# Listas Encadeadas

## Listas Duplamente Encadeadas

Prof. Edson Alves - UnB/FGA

2018

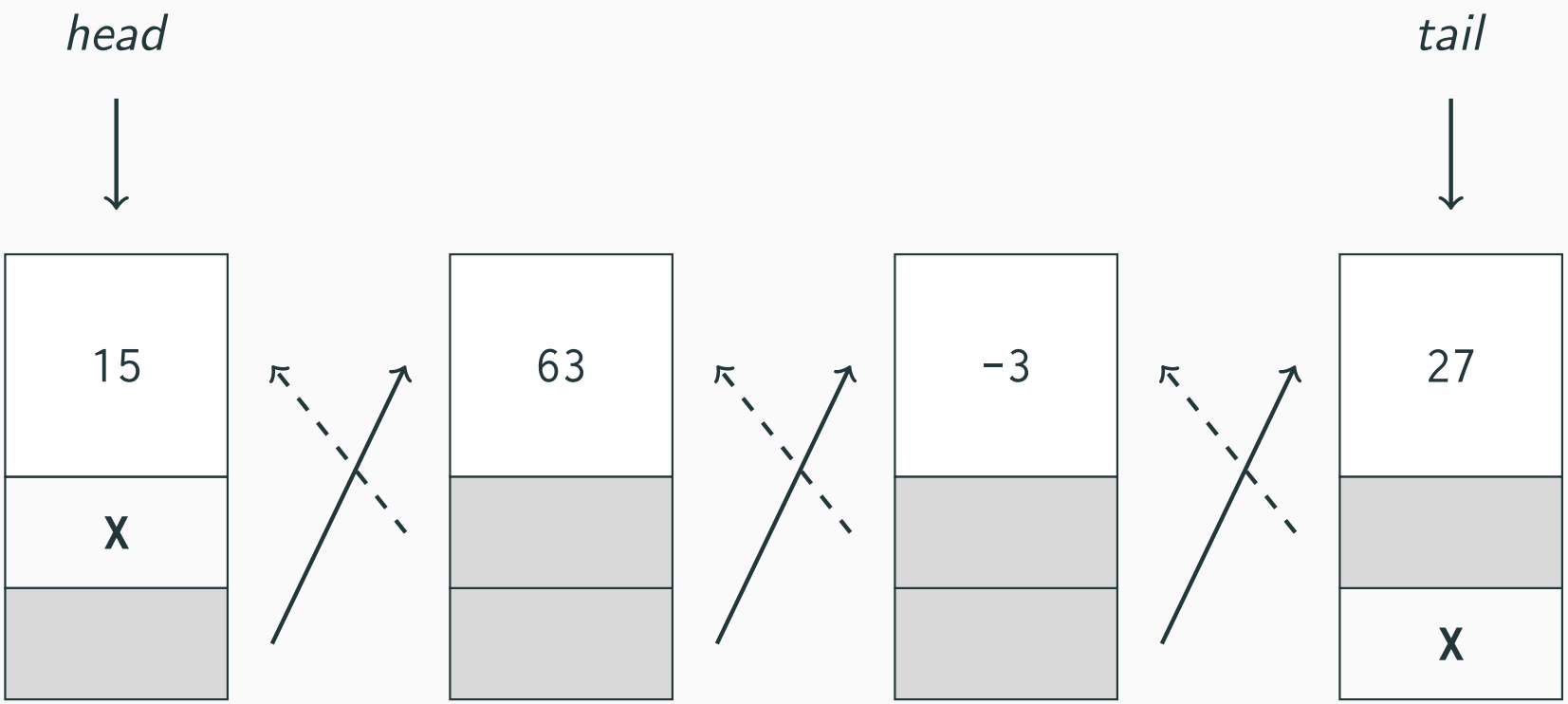
# Sumário

# Definição

# Listas duplamente encadeadas

- Uma lista duplamente encadeada é uma estrutura composta por nós, onde cada nó armazenada uma informação, um ponteiro para o antecessor e um ponteiro para o próximo nó da lista
- A partir de qualquer elemento da lista é possível acessar todos os demais elementos
- Esta maior flexibilidade em relação às listas (simplesmente) encadeadas tem seu custo: cada nó precisa de um ponteiro extra, aumentando o uso de memória
- É uma estrutura de dados linear, devido a travessia sequencial e ordenada de seus elementos
- Por conta da estrutura dos nós, o acesso aleatório em listas duplamente encadeadas tem complexidade $O(N)$

# Visualização de uma lista duplamente encadeada



# Implementação de uma lista duplamente encadeada

- Uma lista duplamente encadeada pode ser implementada como uma `struct` em C ou uma classe em C++
- Ela deve ter um membro para o primeiro (*head*) e para o último (*tail*) elemento da lista
- Cada nó deve ter, no mínimo, três membros: um para armazenar as informações (`info`), um para representar o ponteito para o próximo nó (`next`) e um para representar o ponteiro para o nó anterior (`prev`)
- O primeiro elemento da lista tem membro `prev` nulo
- O último elemento da lista tem membro `next` nulo
- Esta configuração permite inserções e remoções, no íncio e no final, com complexidade $O(1)$

# Exemplo de implementação de uma lista duplamente encadeada

```
1  #ifndef LIST_H
2  #define LIST_H
3
4  template<typename T>
5  class List {
6  public:
7      List() : head(nullptr), tail(nullptr), _size(0) {}
8
9      ~List()
10     {
11         auto p = head;
12
13         while (p)
14         {
15             auto next = p->next;
16             delete p;
17             p = next;
18         }
19     }
20
```

# Exemplo de implementação de uma lista duplamente encadeada

```
21      const T& front() const
22      {
23          if (head)
24              return head->info;
25          else
26              throw "Empty list!";
27      }
28
29      const T& back() const
30      {
31          if (tail)
32              return tail->info;
33          else
34              throw "Empty list!";
35      }
36
37      bool empty() const { return head == nullptr; }
38
39      unsigned long size() const { return _size; }
40
```

# Exemplo de implementação de uma lista duplamente encadeada

```
90  private:
91      struct Node {
92          T info;
93          Node *prev, *next;
94
95          Node(const T& i, Node *p, Node *n) : info(i), prev(p), next(n) {}
96      };
97
98      Node *head, *tail;
99      unsigned long _size;
100 };
101
102 #endif
```

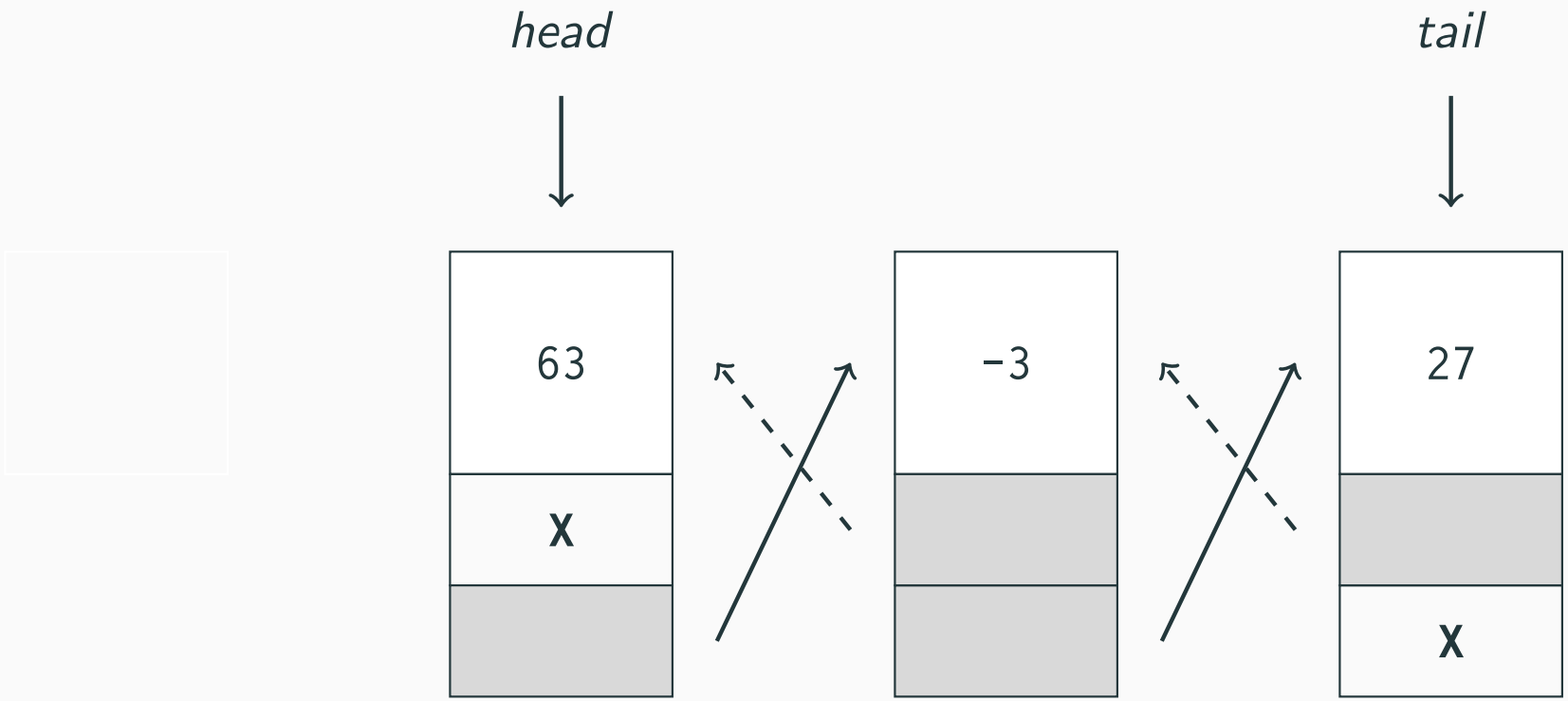
# Inserção

# Inserção no início

- A inserção no início (`push_front()`) de uma lista duplamente encadeada tem complexidade $O(1)$
- O primeiro passo da inserção é criar um novo nó
- Em seguida, deve ser preenchido o campo `info`
- O membro `prev` deve ser nulo
- O membro `next` deve apontar então para o primeiro elemento da lista (head)
- Por fim, o membro head deve apontar para o novo elemento
- *Corner case:* caso o membro head esteja nulo no início da inserção, o membro `tail` também deve apontar para o novo elemento
- Caso a classe tenha o membro `size`, este deve ser incrementado na inserção

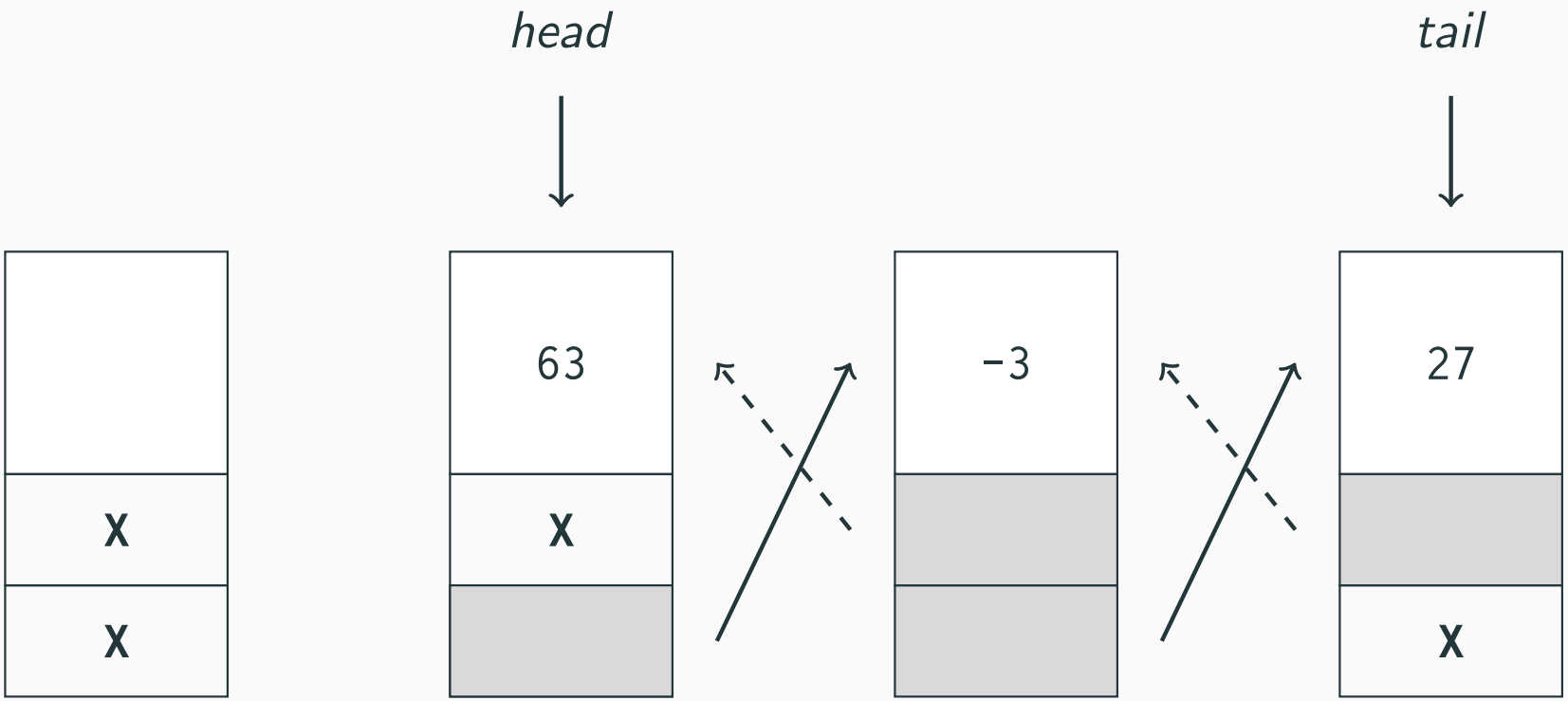
# Visualização da inserção no início

Valor a ser inserido: 88



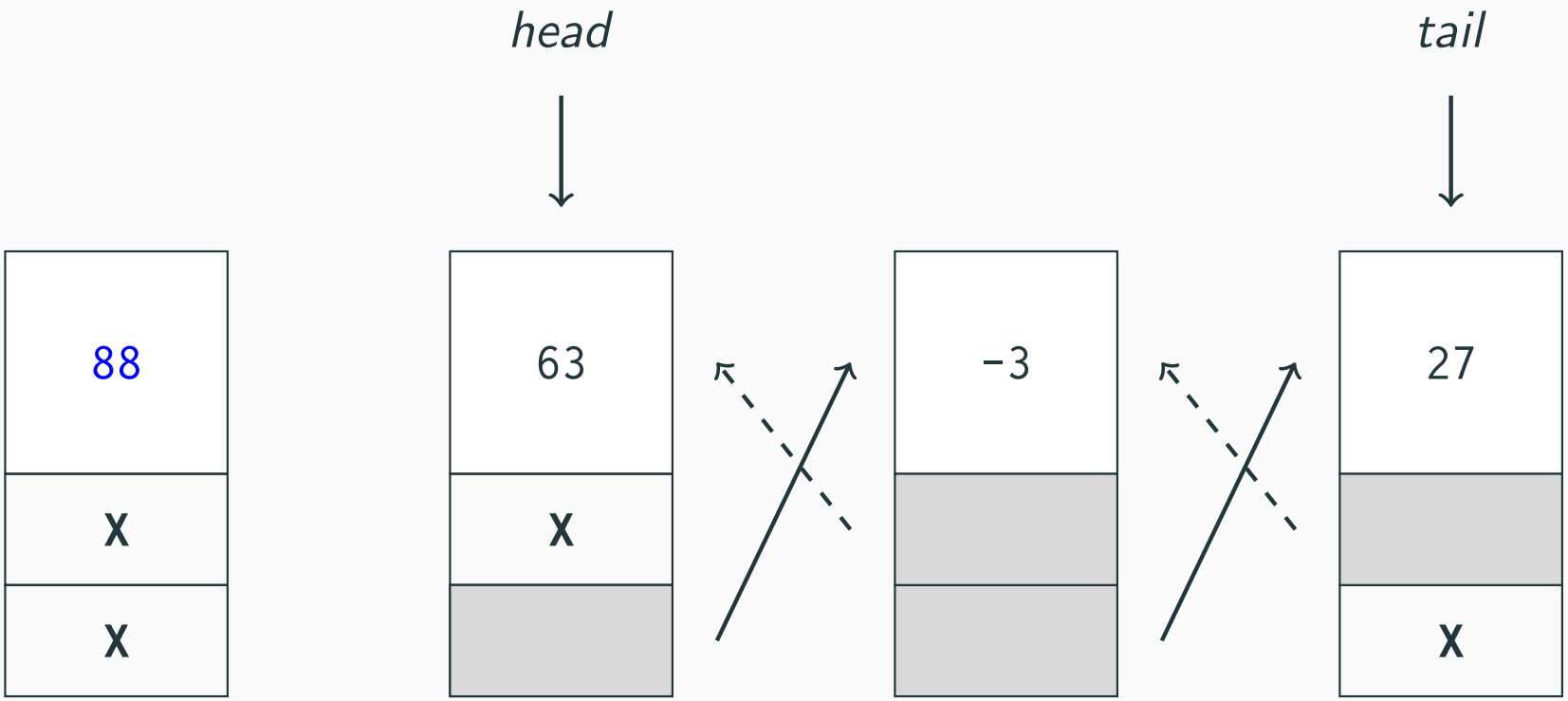
# Visualização da inserção no início

**Passo 01:** Criar um novo nó



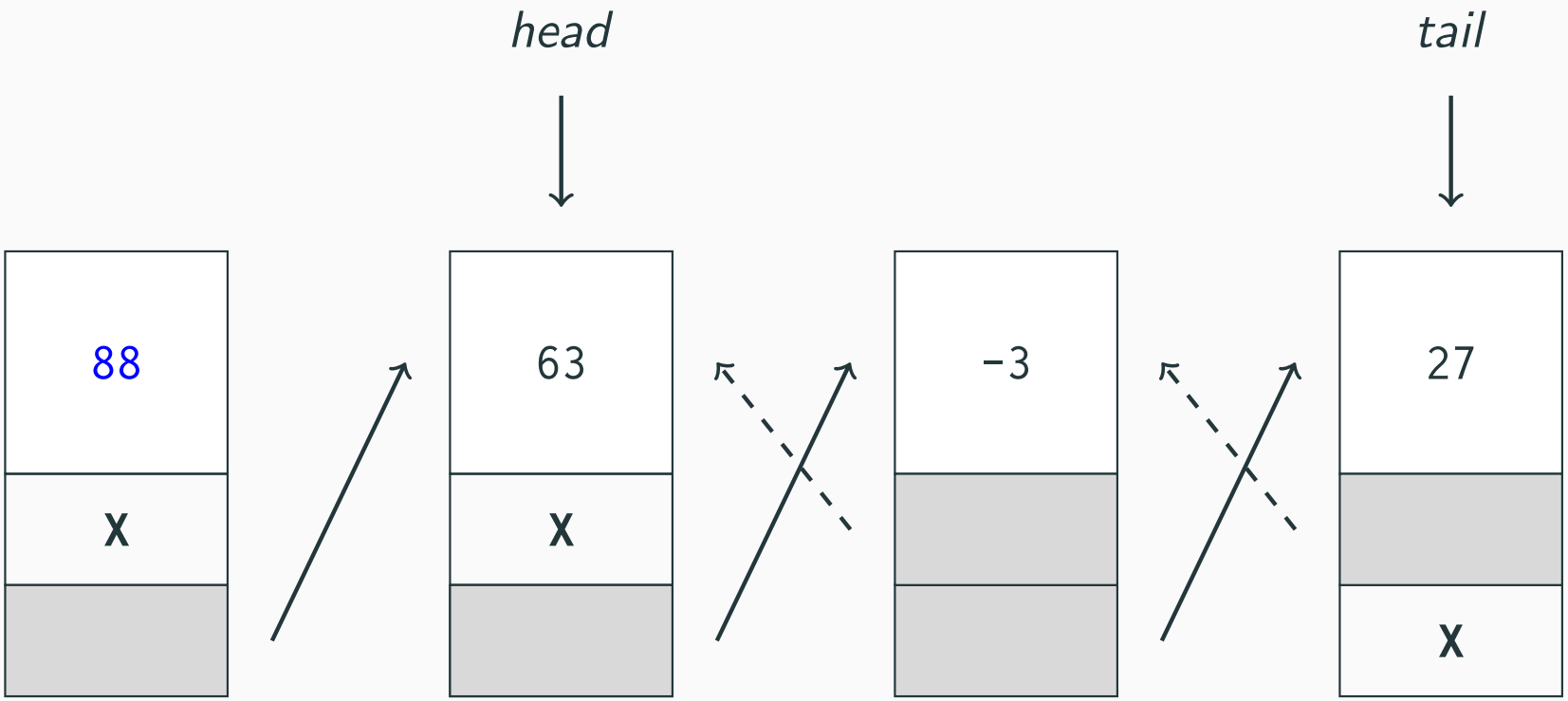
# Visualização da inserção no início

**Passo 02:** Preencher o campo `info`



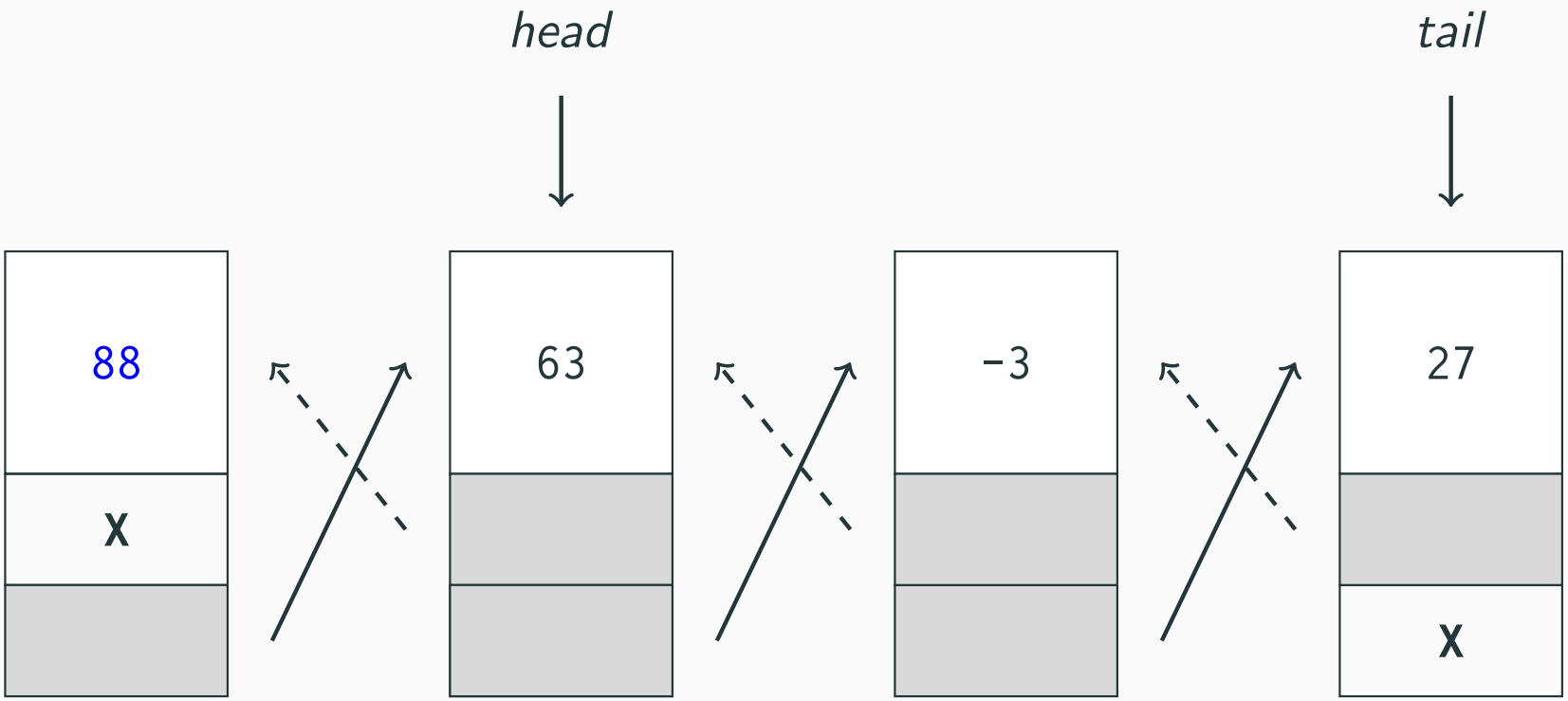
# Visualização da inserção no início

**Passo 03:** Apontar next para o primeiro elemento



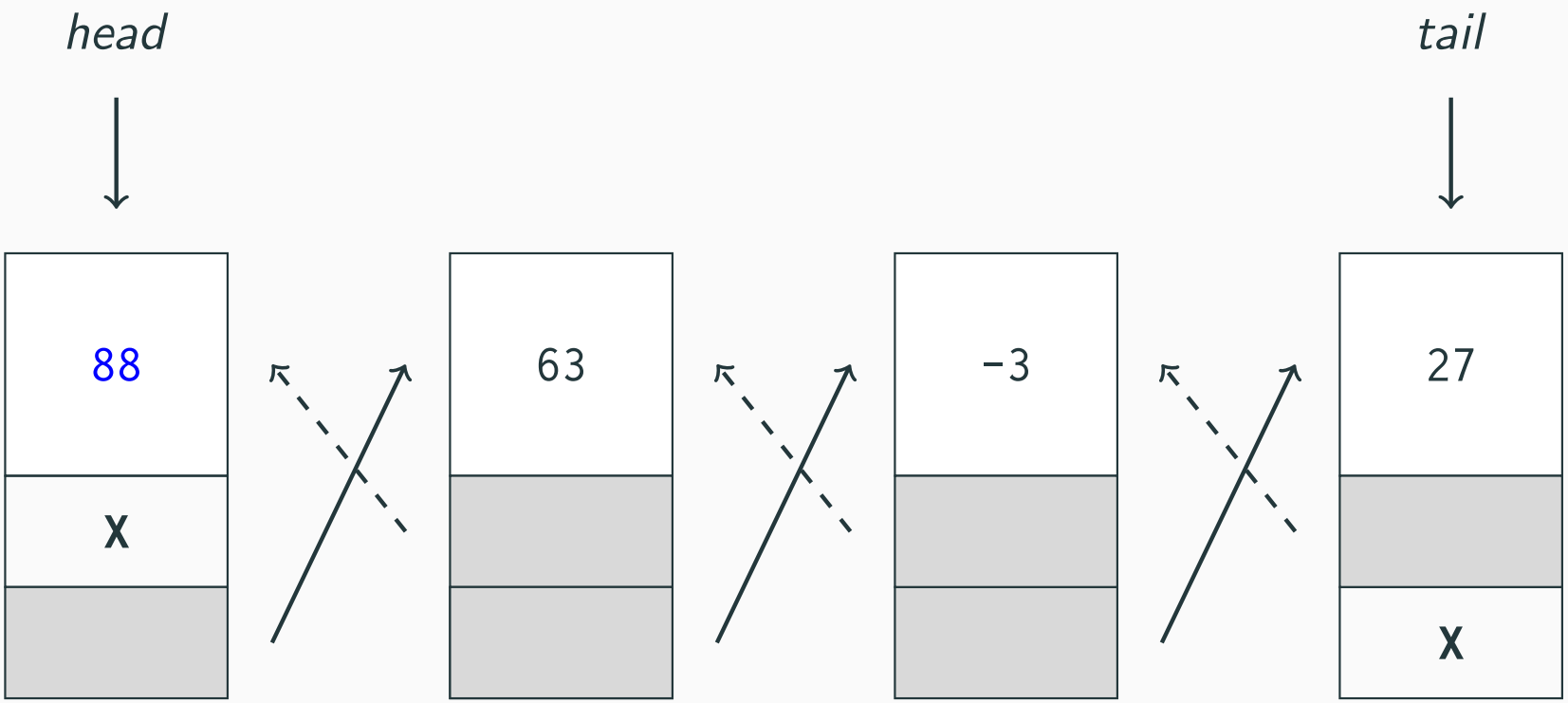
# Visualização da inserção no início

**Passo 04:** Apontar prev de head para o novo elemento



# Visualização da inserção no início

**Passo 05:** Apontar head para o novo elemento



# Implementação da inserção no início

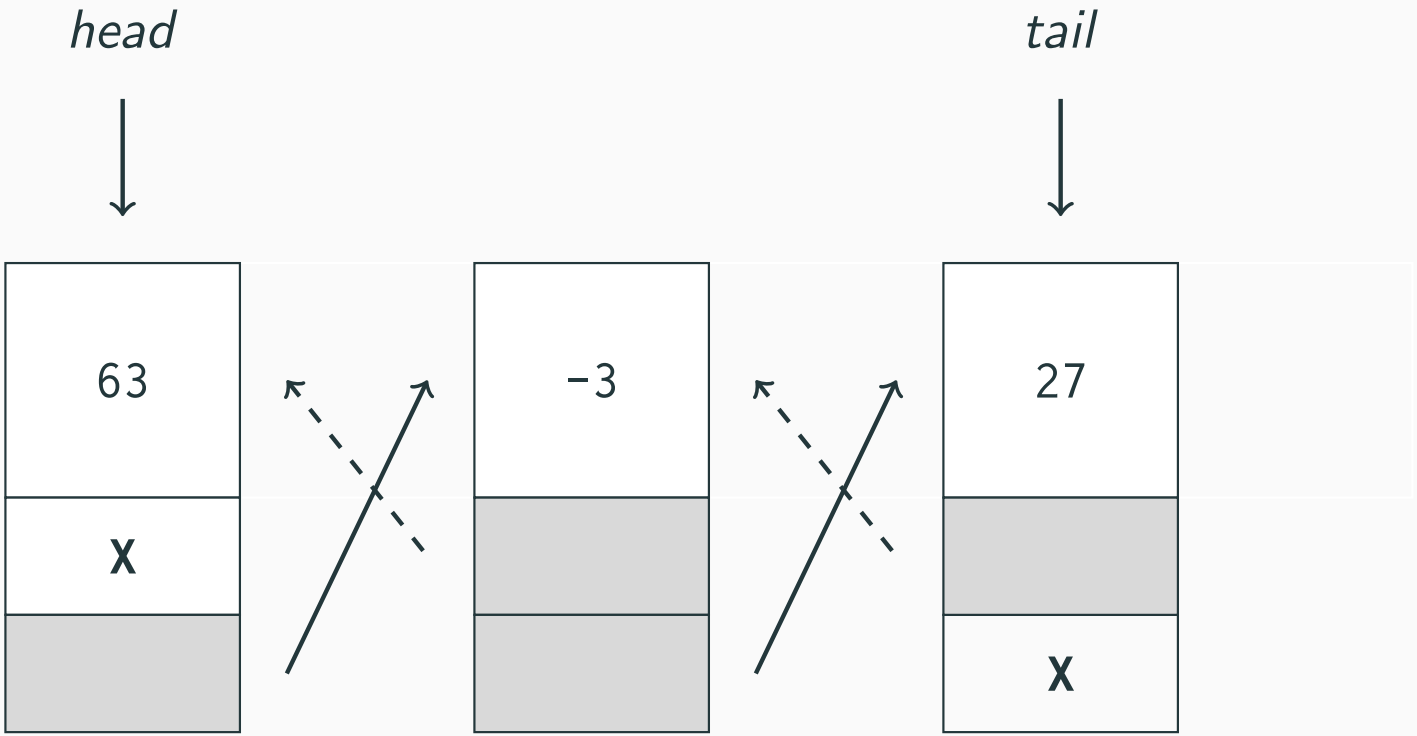
```
41      void push_front(const T& info)
42      {
43          auto n = new Node(info, nullptr, head);
44
45          head ? head->prev = n : tail = n;
46          head = n;
47          _size++;
48      }
```

# Inserção no final

- A inserção no final (push_back()) de uma lista duplamente encadeada tem complexidade $O(1)$
- O primeiro passo da inserção é criar um novo nó
- Em seguida, deve ser preenchido o campo `info`
- O membro `prev` do novo elemento tem que apontar para `tail`
- O membro `next` de `tail` deve apontar então para o novo elemento da lista
- Por fim, o membro `tail` deve apontar para o novo elemento
- *Corner case:* caso o membro `tail` esteja nulo no início da inserção, o membro head também deve apontar para o novo elemento
- Caso a classe tenha o membro `size`, este deve ser incrementado na inserção

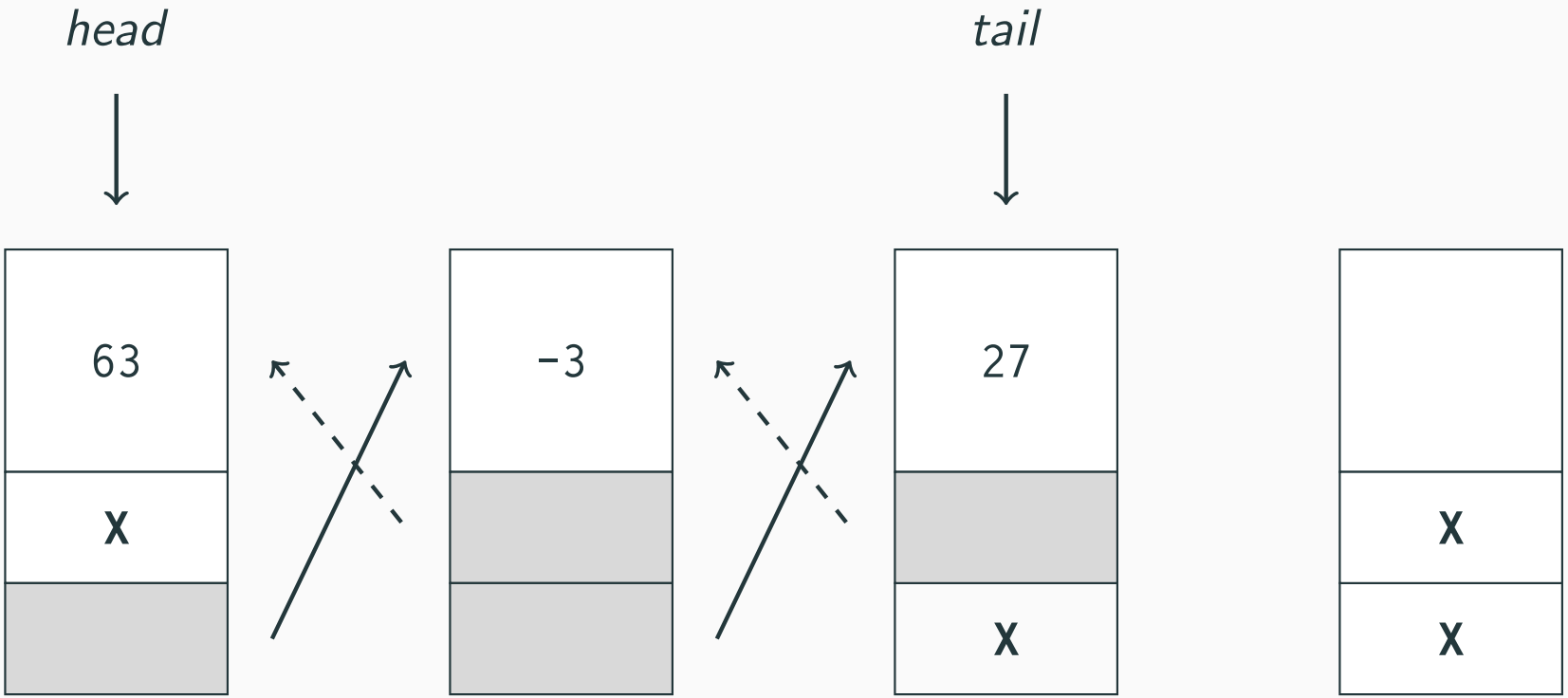
# Visualização da inserção no final

Valor a ser inserido: 88



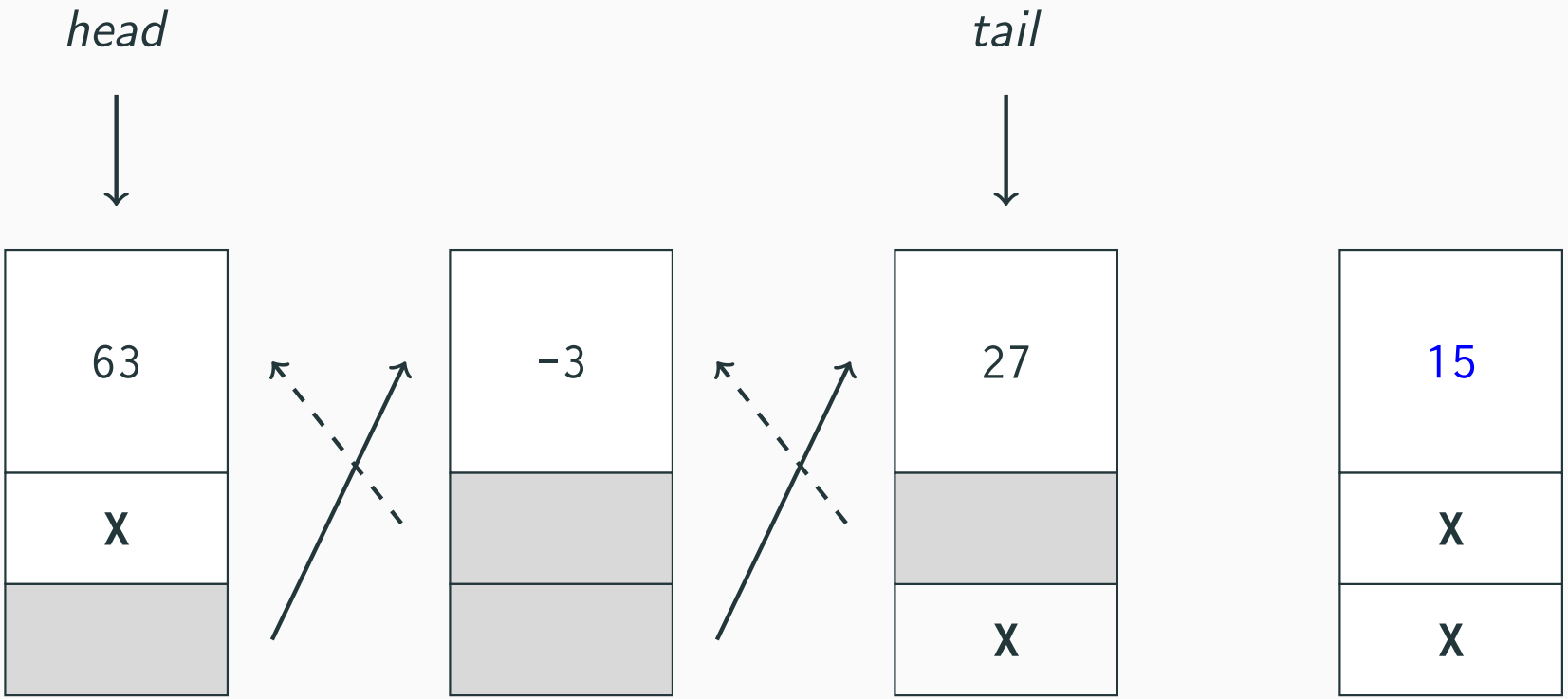
# Visualização da inserção no final

**Passo 01:** Criar um novo nó



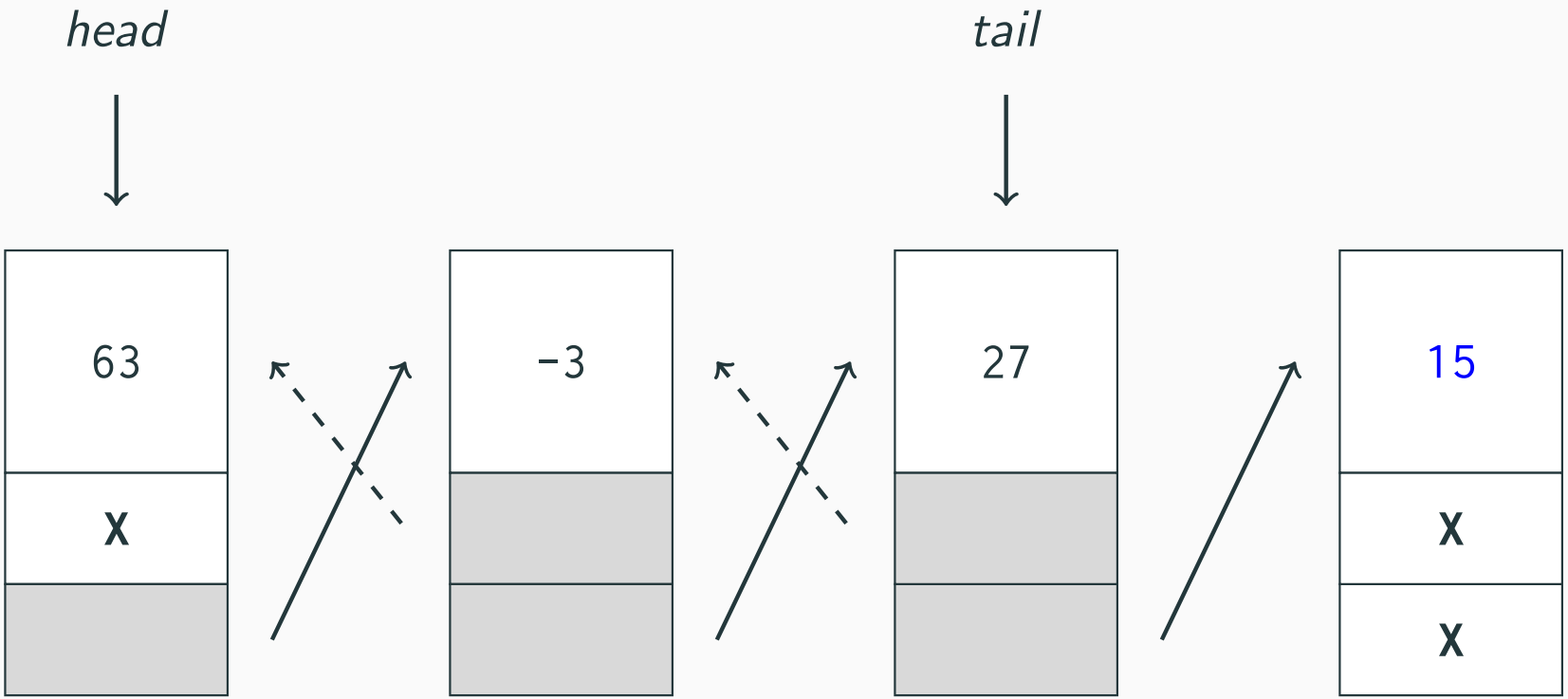
# Visualização da inserção no final

**Passo 02:** Preencher o campo `info`



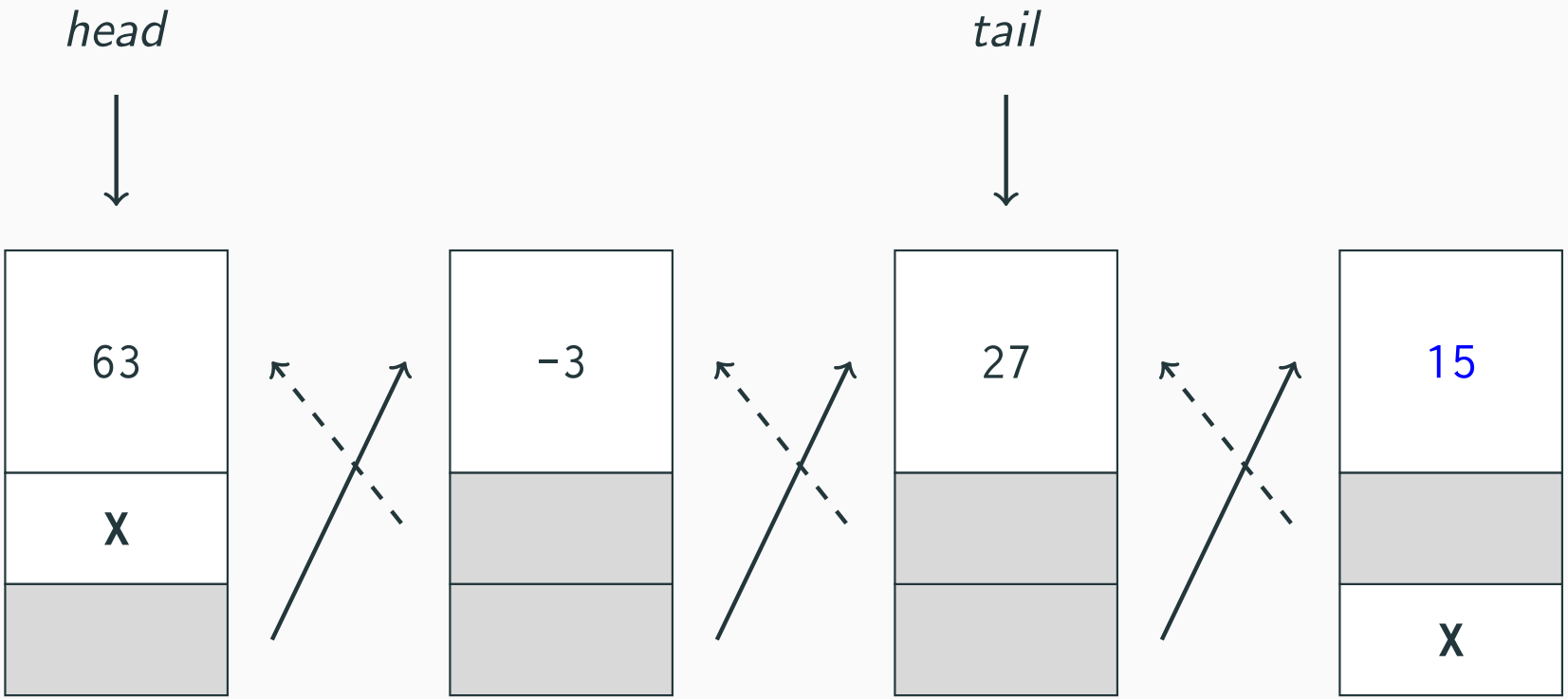
# Visualização da inserção no final

**Passo 03:** Apontar o membro next de `tail` para o novo nó



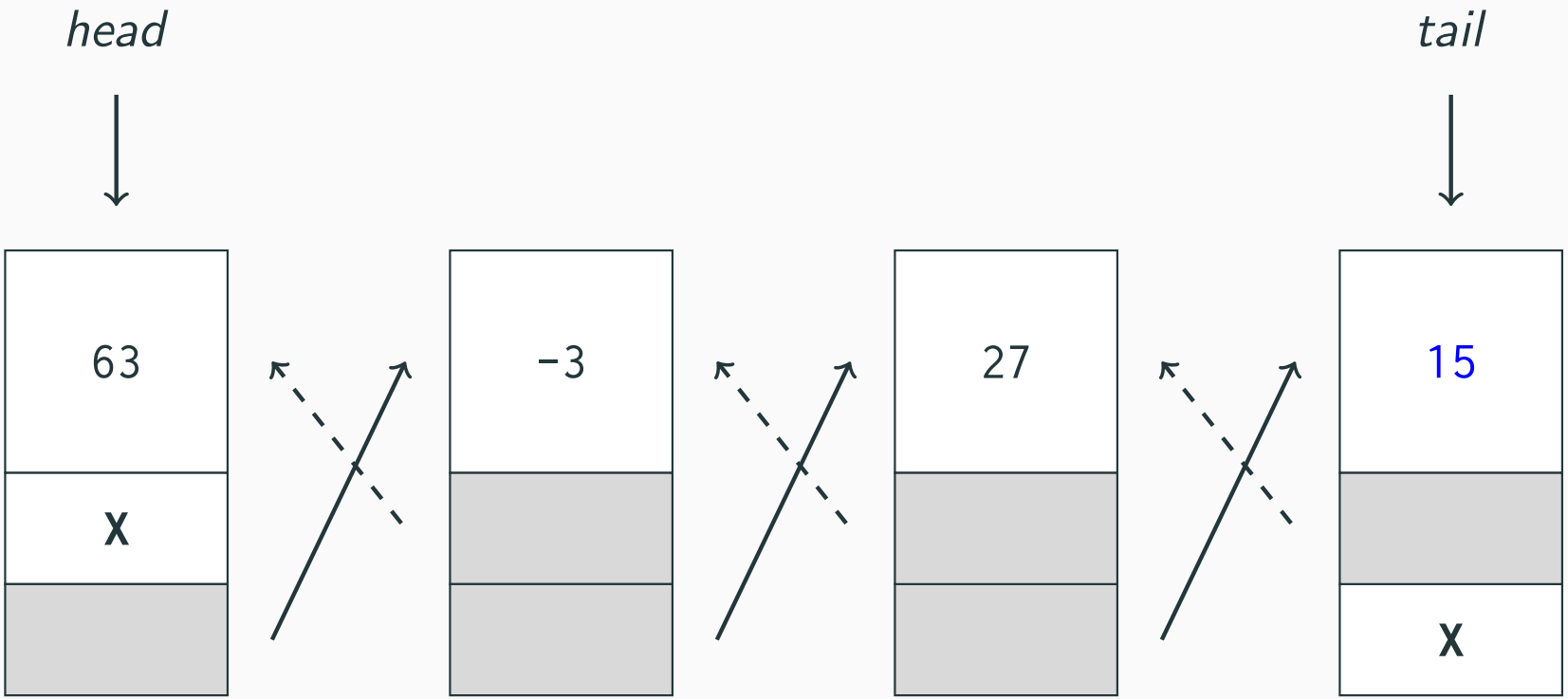
# Visualização da inserção no final

**Passo 04:** Apontar o membro prev do novo nó para `tail`



# Visualização da inserção no final

**Passo 05:** Apontar `tail` para o novo nó



# Implementação da inserção no final

```
50      void push_back(const T& info)
51      {
52          auto n = new Node(info, tail, nullptr);
53
54          tail ? tail->next = n : head = n;
55          tail = n;
56          _size++;
57      }
```

# Inserção em posição arbitrária

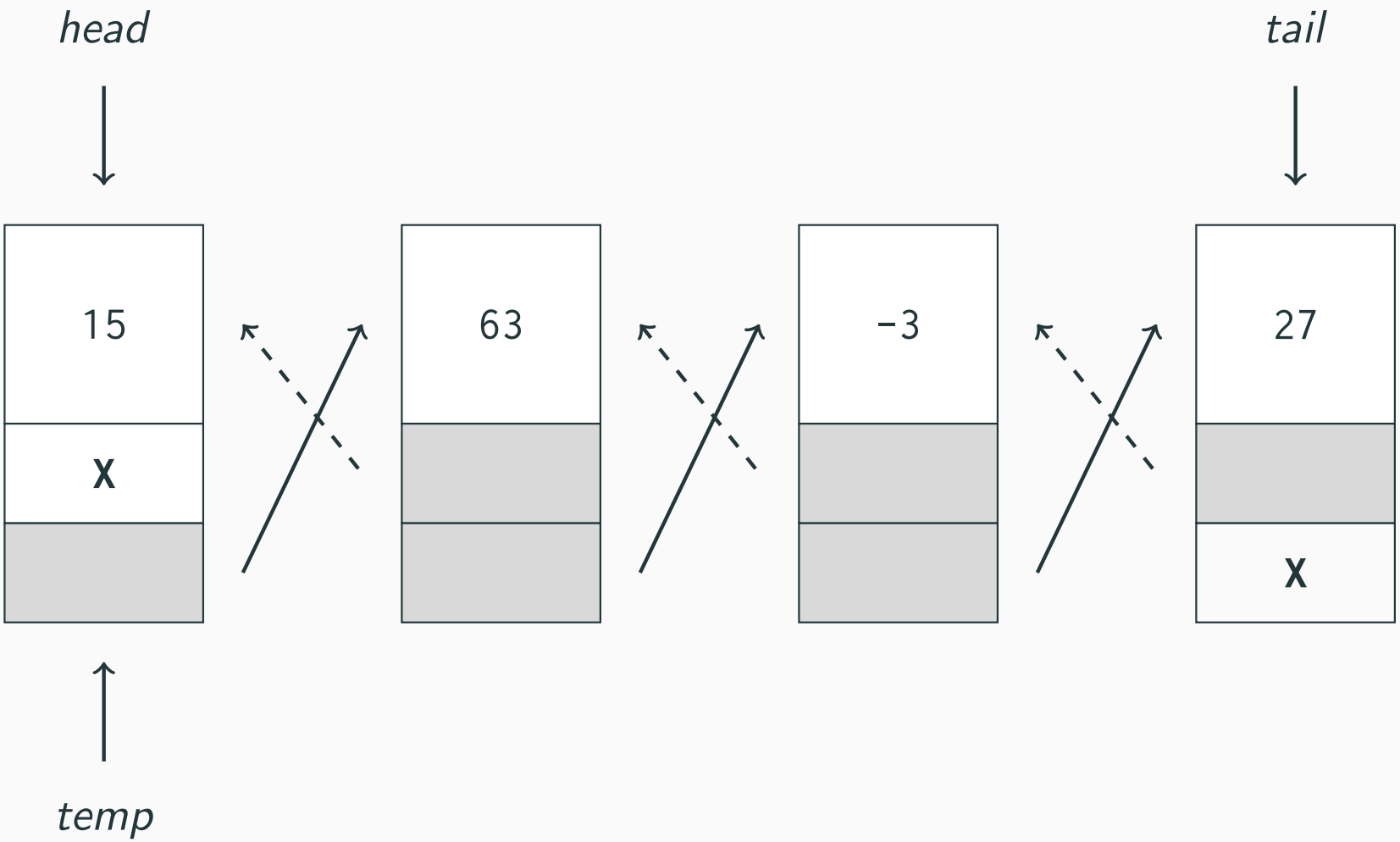
- A inserção em posição arbitrária tem complexidade $O(N)$, onde $N$ é o número de elementos da lista, ou $O(1)$, caso o ponteiro para a posição onde o elemento será inserido seja conhecido
- Observe que, no caso dos vetores, mesmo que a posição de inserção seja conhecida, a complexidade permanece sendo $O(N)$
- É preciso ajustar adequadamente os membros `prev` e `next` do novo nó, de seu antecessor e de seu sucessor, caso existam
- Também é preciso ter cuidado com os ponteiros head e `tail`
- Há vários *corner cases*:
  1. Lista vazia
  2. Apenas um elemento na lista
  3. Inserção na primeira posição
  4. Posição de inserção inválida

# Remoção

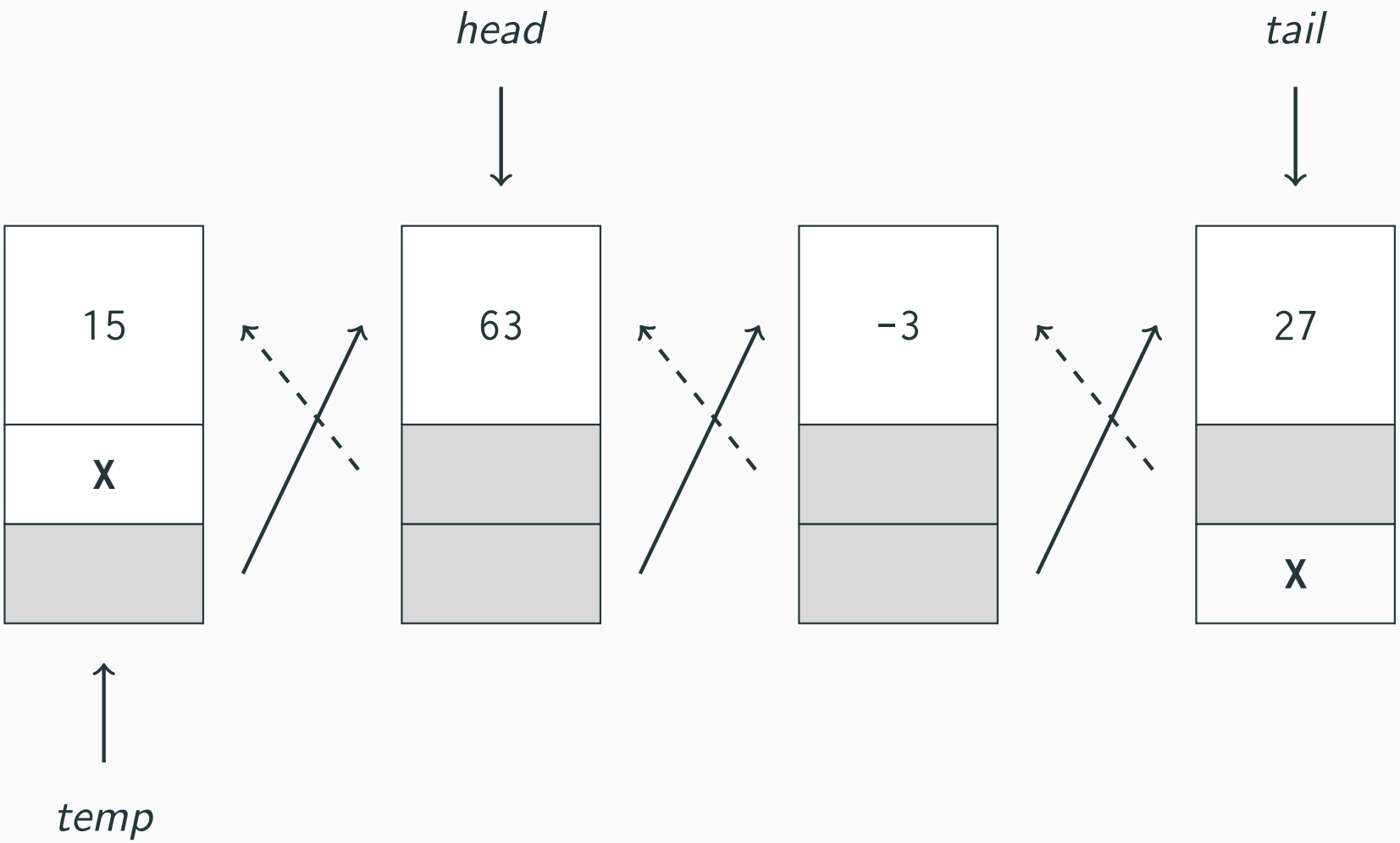
# Remoção do início

- A remoção de um elemento do início de uma lista duplamente encadeada (`pop_front()`) tem complexidade $O(1)$
- O primeiro passo da remoção é armazenar o membro head em uma variável temporária
- Em seguida, o membro head deve apontar para o próximo elemento da lista
- O membro `next` de head deve ser atualizado com o valor nulo, caso exista
- Por fim, o ponteiro armazenado na variável temporária é deletado
- O membro `size` deve ser decrementado, se existir
- *Corner case:* a tentativa de remoção em uma lista vazia deve ser tratada de alguma maneira (código de erro, exceção, etc.)
- *Corner case:* se a lista tiver exatamente um nó no momento da remoção, o membro `tail` deve receber o valor nulo

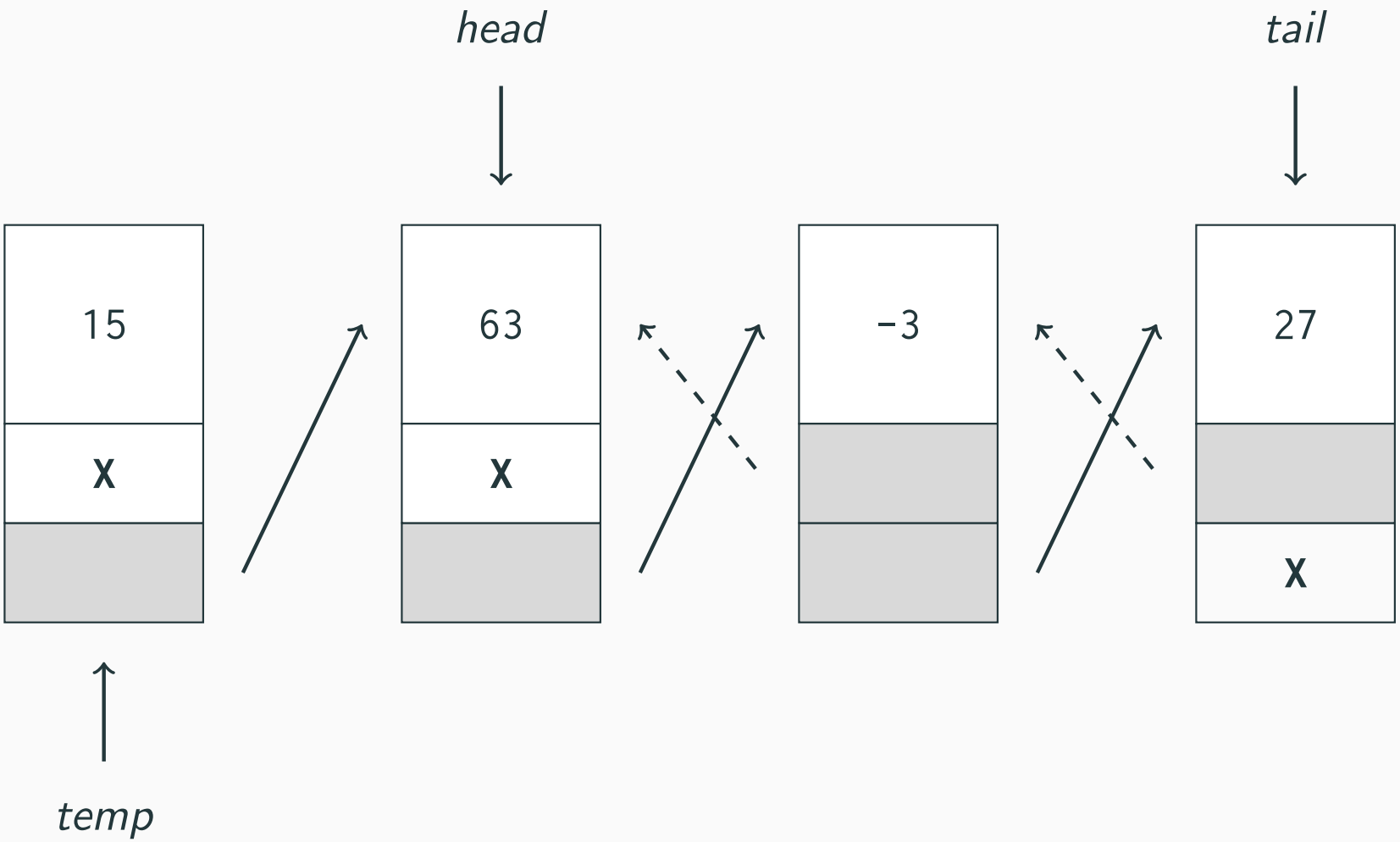
# Visualização da remoção no início da lista



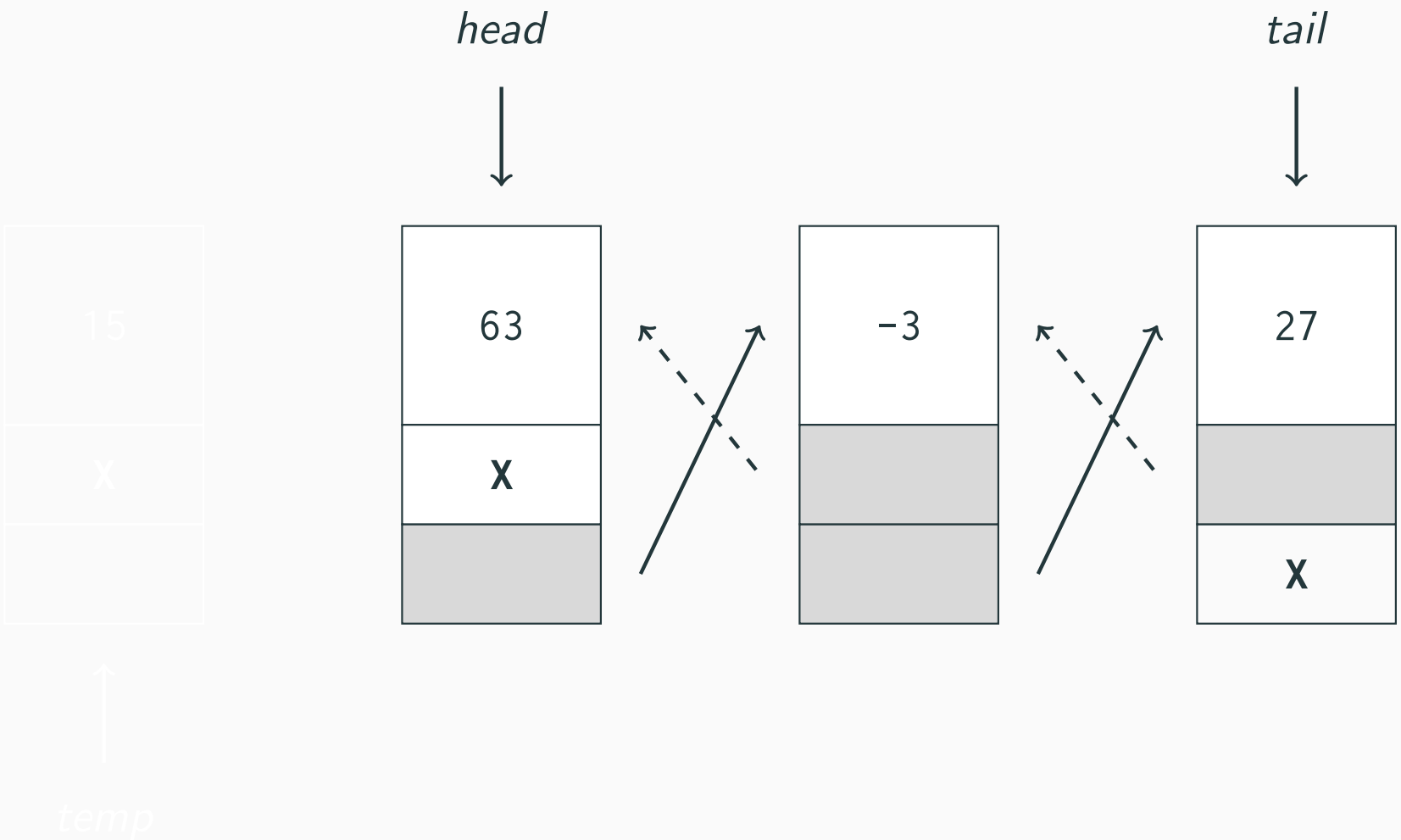
# Visualização da remoção no início da lista



# Visualização da remoção no início da lista



# Visualização da remoção no início da lista



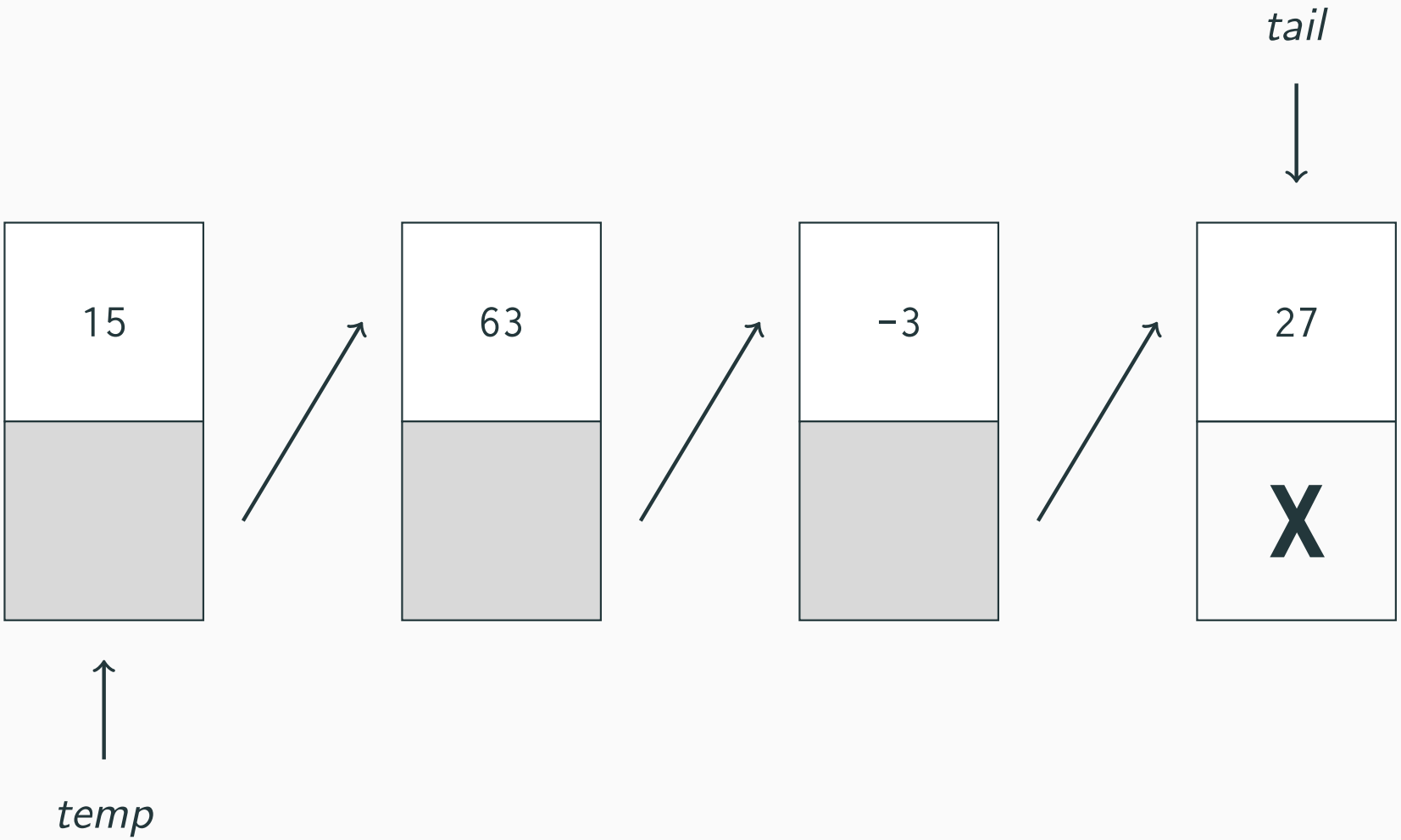
# Implementação da remoção do início

```
59      void pop_front()
60      {
61          if (!head)
62              throw "Lista vazia";
63
64          auto temp = head;
65          head = head->next;
66          delete temp;
67
68          head ? head->prev = nullptr : tail = nullptr;
69          _size--;
70      }
```

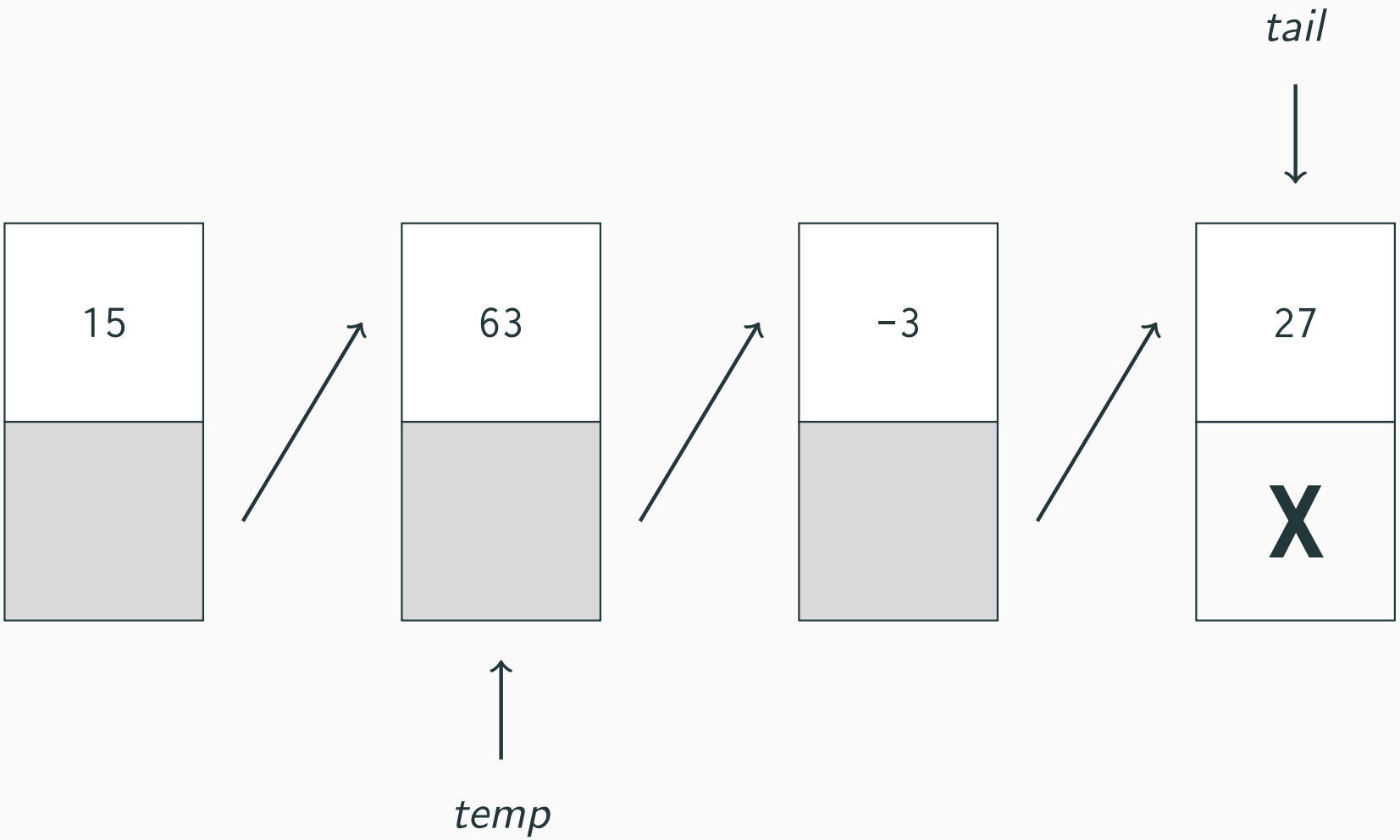
# Remoção do final da lista

- Mesmo com o membro `tail`, a remoção do último elemento de uma lista encadeada (pop_back()) tem complexidade $O(N)$, onde $N$ é o número de nós da lista
- Isto acontece porque é preciso localizar o elemento que antecede o último elemento (prev), processo que tem complexidade linear
- Localizado o elemento prev, a remoção é semelhante à remoção do início: o elemento `tail` é deletado e `tail` passa a apontar para prev
- Por fim, o membro next de prev deve se tornar nulo
- Novamente, o membro `size` deve ser decrementado, se existir
- *Corner case:* lista vazia
- *Corner case:* se a lista tiver exatamente um nó no momento da remoção, o membro head deve receber o valor nulo

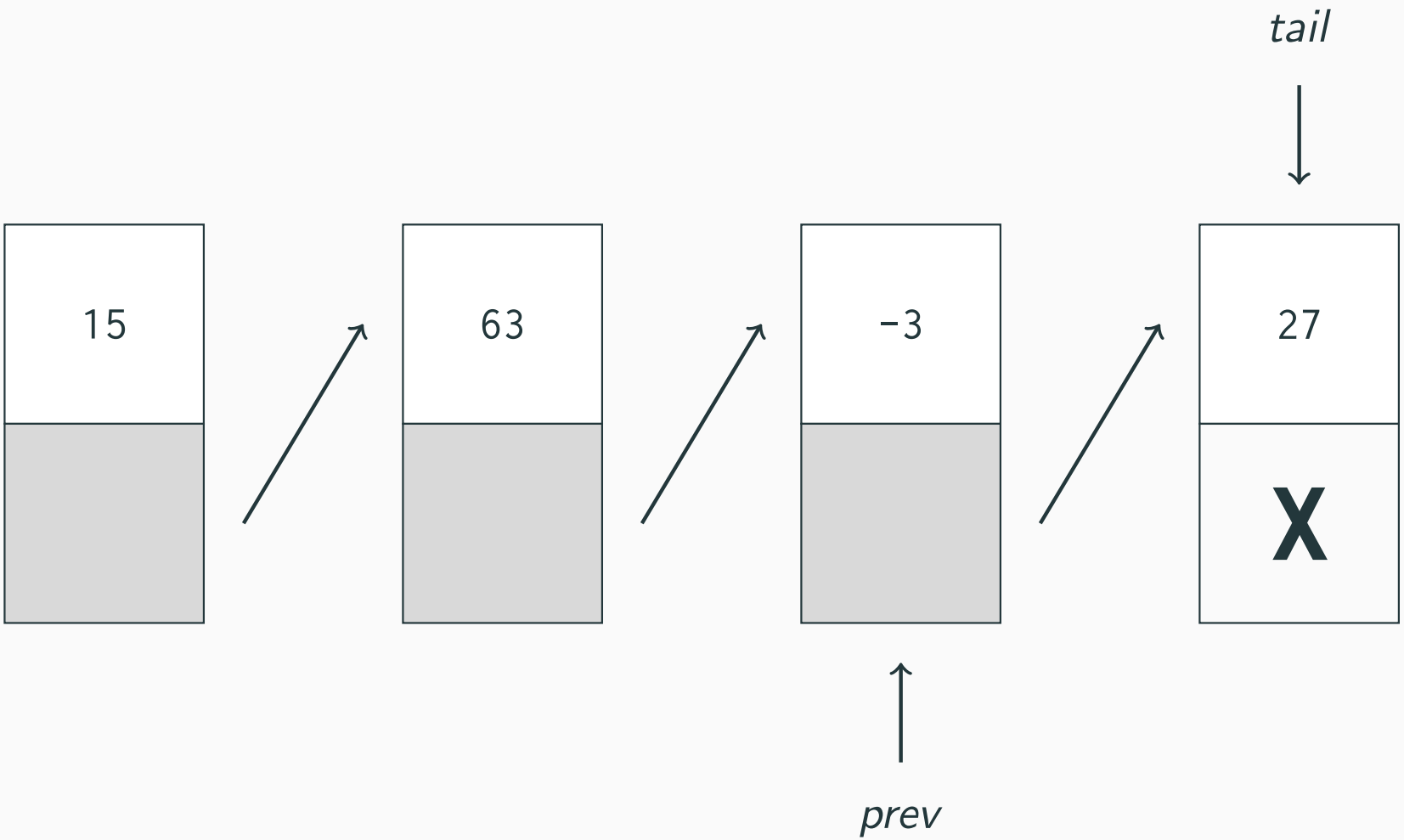
# Visualização da remoção no final da lista



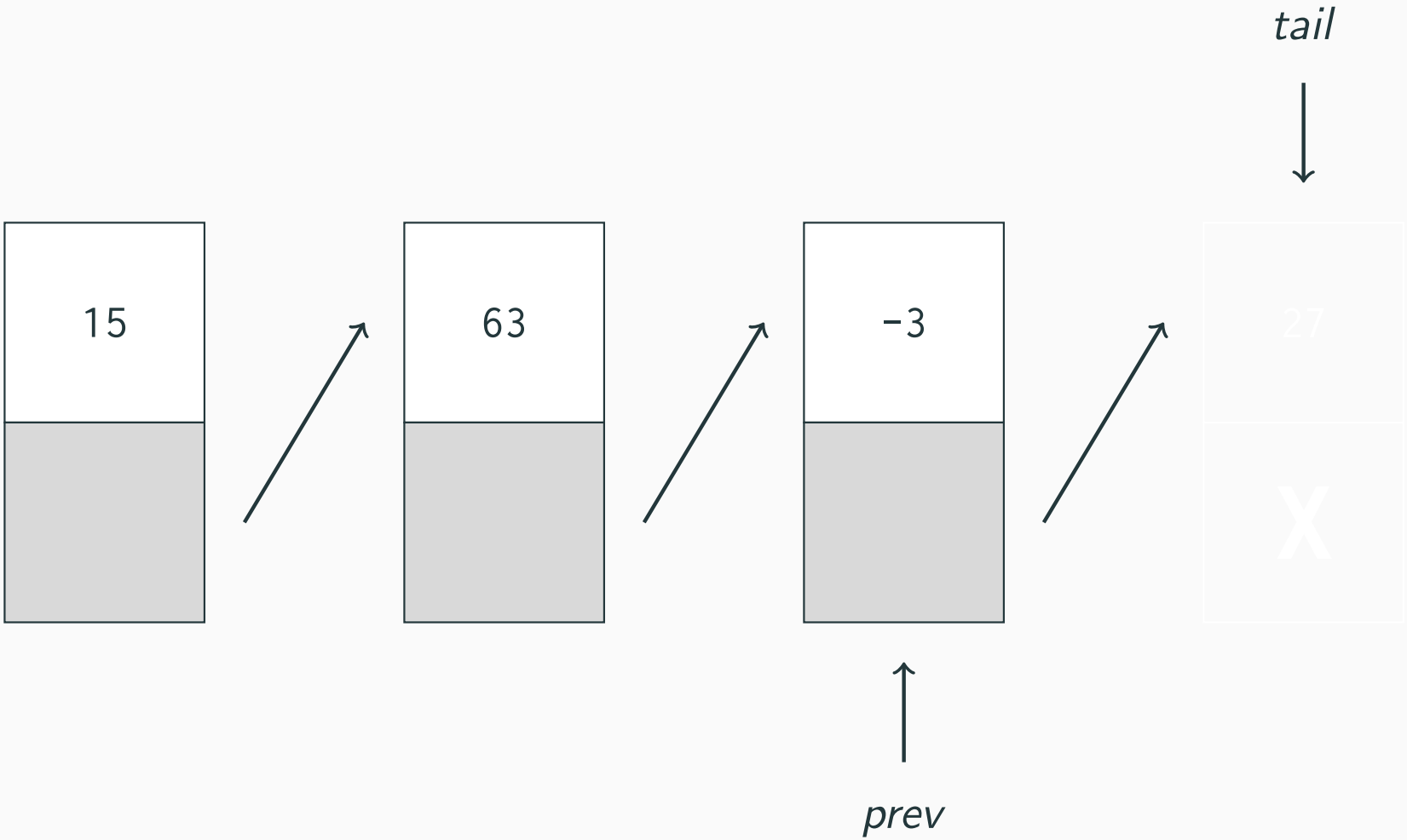
# Visualização da remoção no final da lista



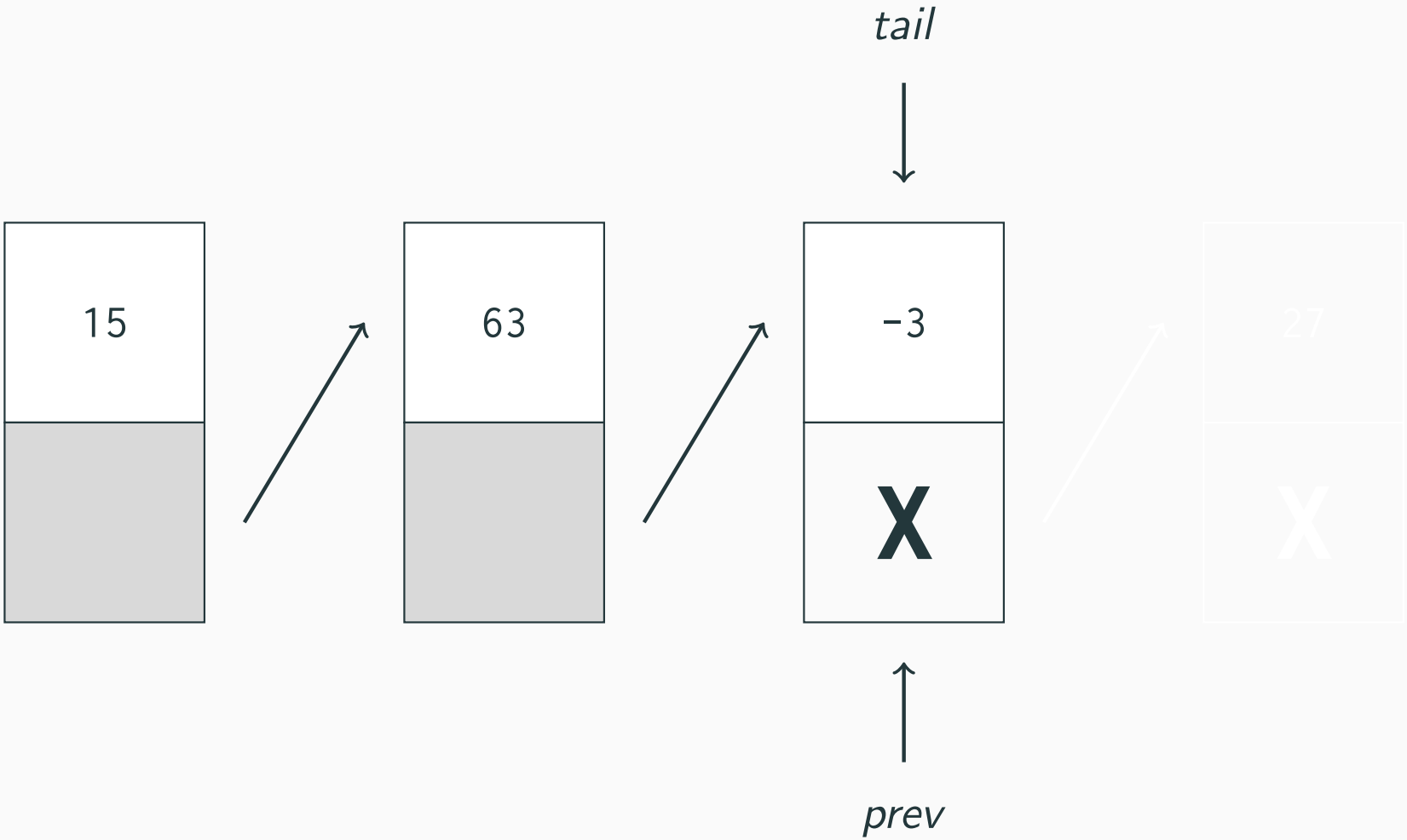
# Visualização da remoção no final da lista



# Visualização da remoção no final da lista



# Visualização da remoção no final da lista



# Implementação da remoção do final

```
72      void pop_back()
73      {
74          if (!head)
75              throw "Lista vazia";
76
77          auto prev = head;
78
79          while (prev->next and prev->next != tail)
80              prev = prev->next;
81
82          delete tail;
83
84          tail == head ? (head = tail = nullptr)
85              : (tail = prev, tail->next = nullptr);
86
87          _size--;
88      }
```

# Remoção em posição arbitrária

- A remoção em posição arbitrária também tem complexidade $O(N)$
- Esta é uma rotina de implementação complexa, dado o grande número de *corner cases*
- É preciso localizar o elemento que antece o elemento a ser removido, como no caso da remoção ao final
- Os membros head e `tail` deve ser devidamente tratados e atualizados, quando for o caso
- O membro `next` de `prev` também precisa ser atualizado corretamente

# Exemplo de uso da lista encadeada

```cpp
#include <iostream>
#include "list.h"

using namespace std;

int main()
{
    List<int> list;

    cout << "Lista vazia? " << (list.empty() ? "Sim" : "Nao") << endl;

    for (int n = 1; n <= 10; ++n)
        list.push_front(n);

    list.pop_back();
    cout << "Tamanho da lista: " << list.size() << endl;
    cout << "Primeiro elemento: " << list.front() << endl;
    cout << "Último elemento: " << list.back() << endl;

    return 0;
}
```

# Listas encadeadas em C++

# forward_list

- Desde a versão C++, a linguagem C++ oferece uma implementação de listas simplesmente encadeadas: o contêiner `forward_list`
- Por padrão a implementação deve ser tão eficiente quanto a implementação de uma lista simplesmente encadeada em C
- Por este motivo, é o único dentre os contêiners da STL que não tem um método `size()`
- As inserções e remoções constantes devem ser feitas através dos métodos `push_front()` e `pop_front()`
- Também está disponível o método `insert_after()`, que insere um elemento logo após o iterador indicado em complexidade constante

# Exemplo de uso da forward_list

```cpp
#include <iostream>
#include <algorithm>
#include <forward_list>

using namespace std;

int main()
{
    forward_list<int> list;

    cout << "Lista vazia? " << (list.empty() ? "Sim" : "Nao") << endl;

    for (int n = 1; n <= 10; ++n)
        list.push_front(n);

    list.pop_front();

    // Complexidade linear
    int size = distance(list.begin(), list.end());
    cout << "Tamanho da lista: " << size << endl;

```

# Exemplo de uso da forward_list

```
22      cout << "Primeiro elemento: " << list.front() << endl;
23
24      // Complexidade linear
25      int last = -1;
26
27      for (auto it = list.begin(); it != list.end(); ++it)
28          last = *it;
29
30      cout << "Último elemento: " << last << endl;
31
32      return 0;
33  }
```

# Referências


1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
3. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.
4. C++ Reference[1].


[1] https://en.cppreference.com/w/