

# Listas

## Listas Duplamente Encadeadas

---

Prof. Edson Alves - UnB/FGA

2018



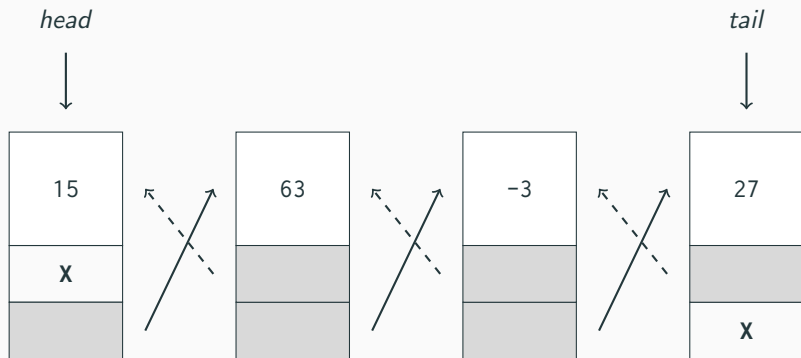
# Definição

---

## 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 ponteiro 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 início 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

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

# Inserção

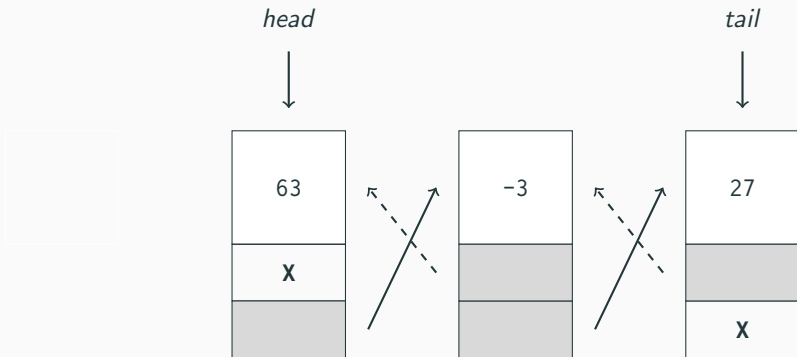
---

# 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

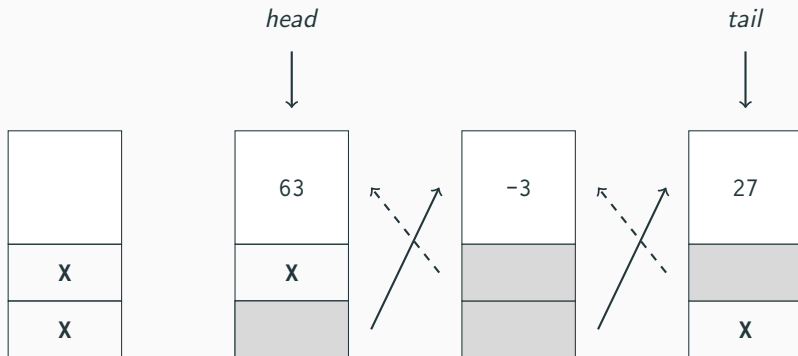
# Visualização da inserção no início

Valor a ser inserido: 88



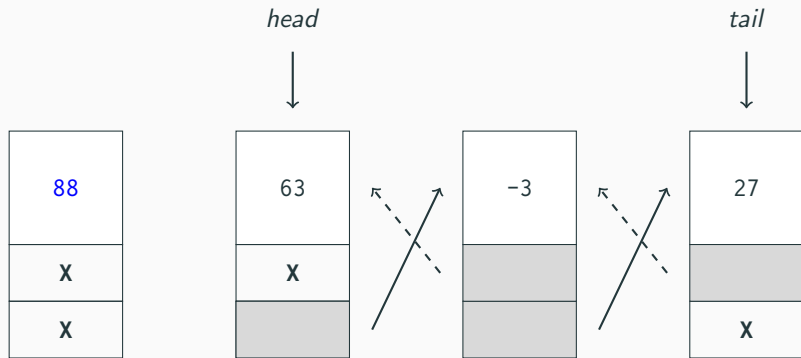
# Visualização da inserção no início

**Passo 01:** Criar um novo nó



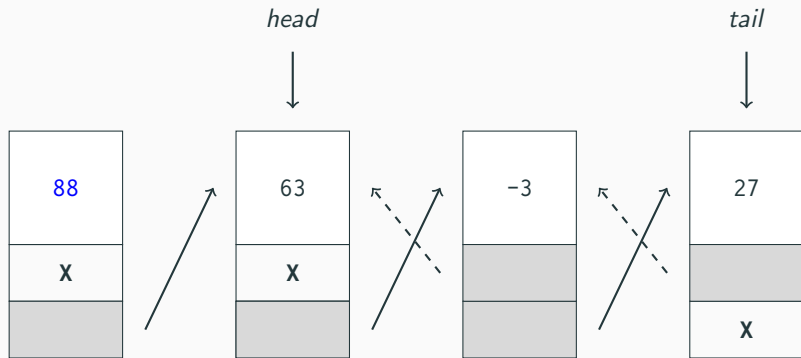
# Visualização da inserção no início

## Passo 02: Preencher o campo info



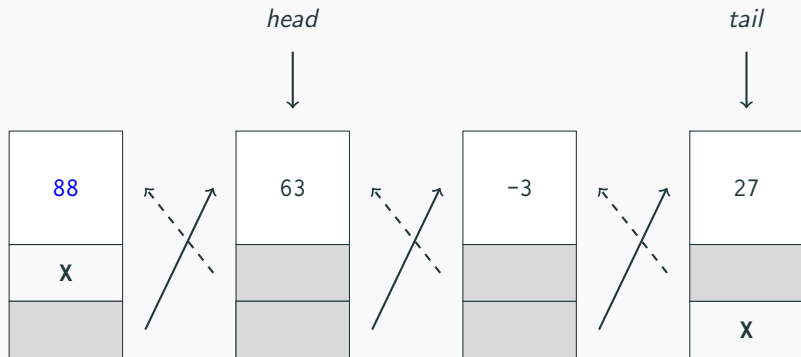
# Visualização da inserção no início

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



# Visualização da inserção no início

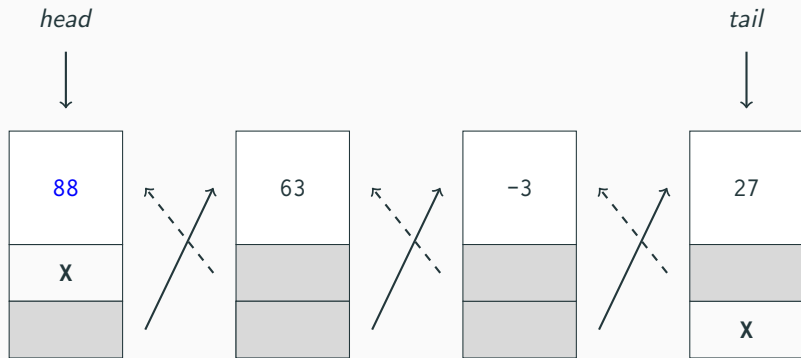
**Passo 04:** Apontar prev de head para o novo elemento





# Visualização da inserção no início

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



# Implementação da inserção no início

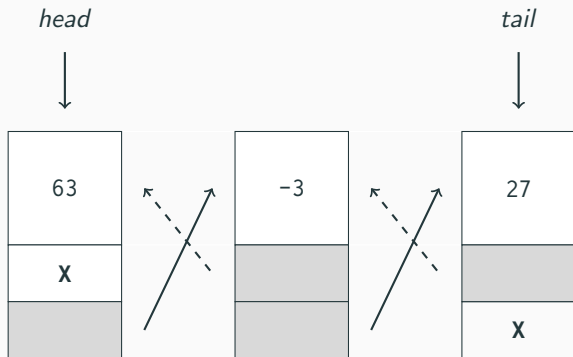
```
41 void push_front(const T& info)
42 {
43     auto n = new Node(info, nullptr, head);
44
45     head ? head->prev = n : tail = n;
46     head = n;
47     _size++;
48 }
```

## 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

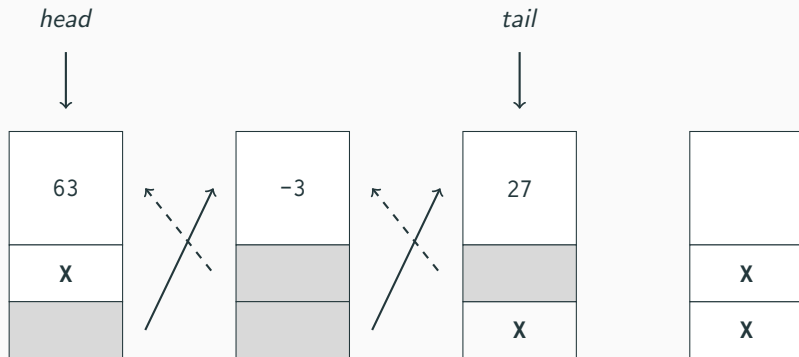
# Visualização da inserção no final

Valor a ser inserido: 15



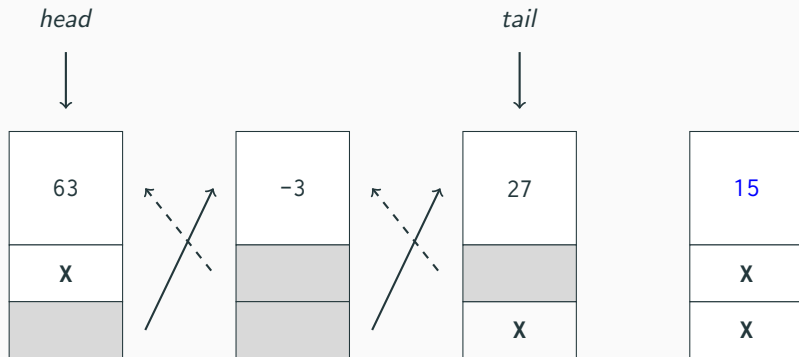
# Visualização da inserção no final

**Passo 01:** Criar um novo nó



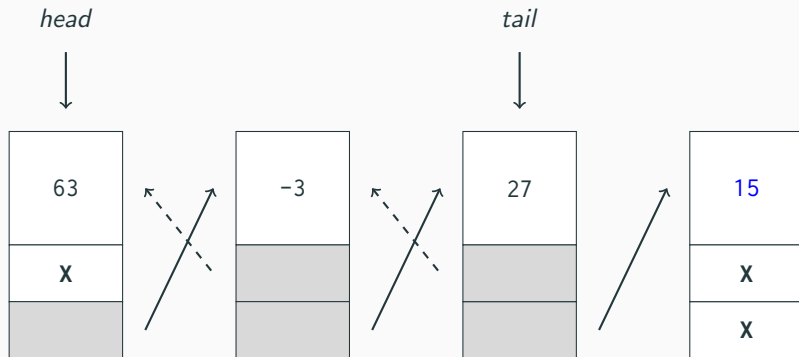
# Visualização da inserção no final

## Passo 02: Preencher o campo info



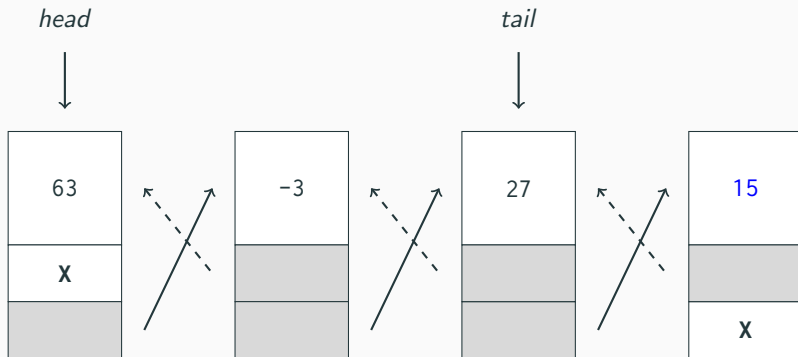
# Visualização da inserção no final

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



# Visualização da inserção no final

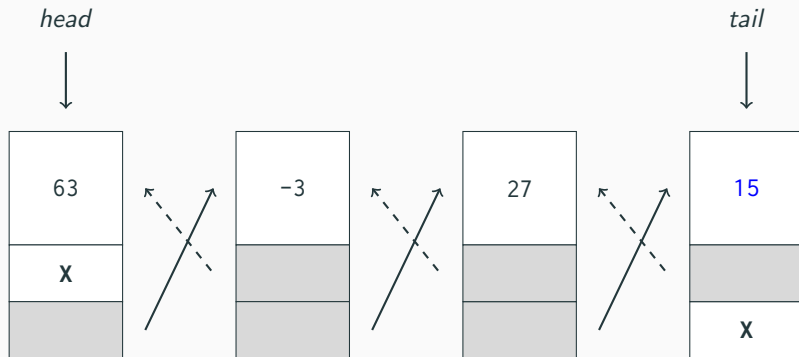
**Passo 04:** Apontar o membro prev do novo nó para tail





# 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

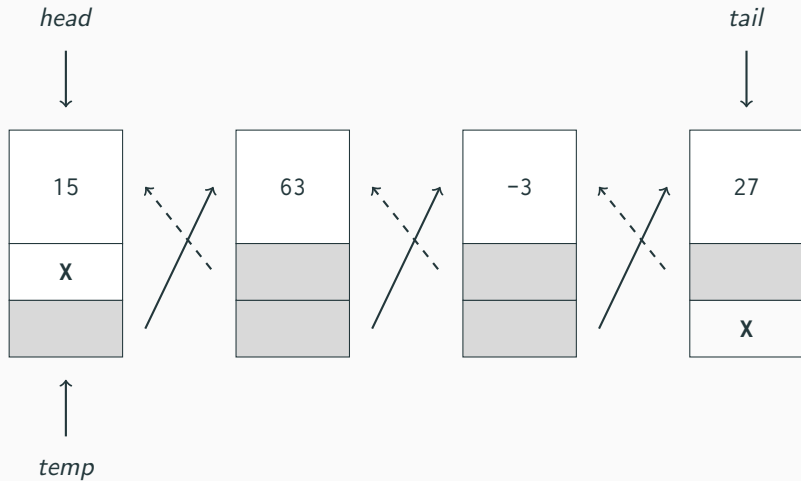
# Remoção

---

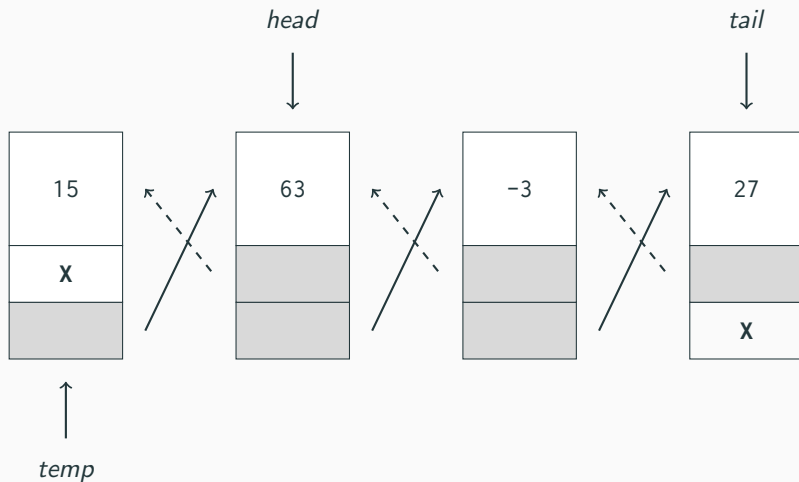
# Remoção do início

- 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

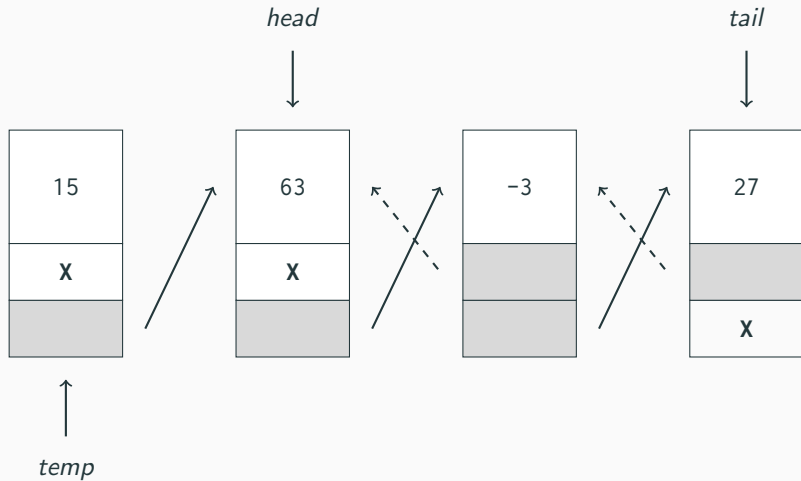
## Visualização da remoção no início da lista



## Visualização da remoção no início da lista

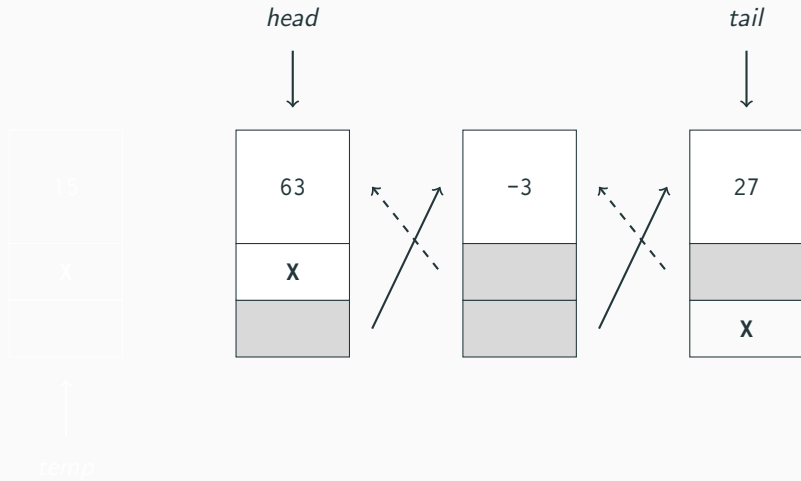


## Visualização da remoção no início da lista





## Visualização da remoção no início da lista



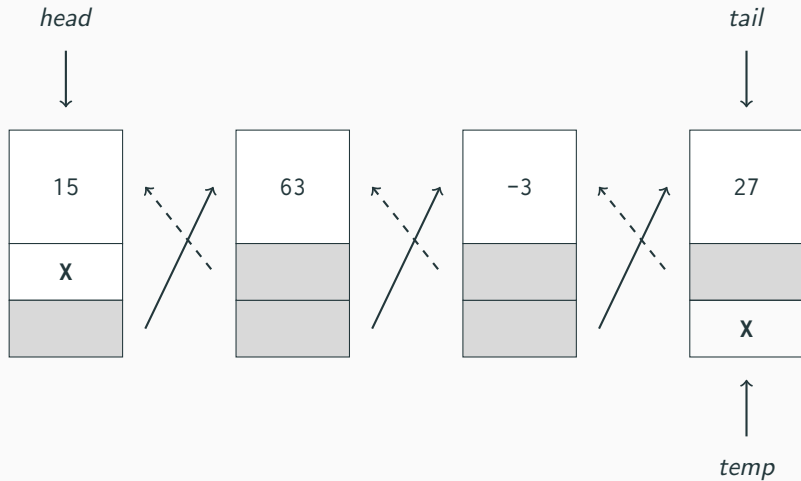
## Implementação da remoção do início

```
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 }
```

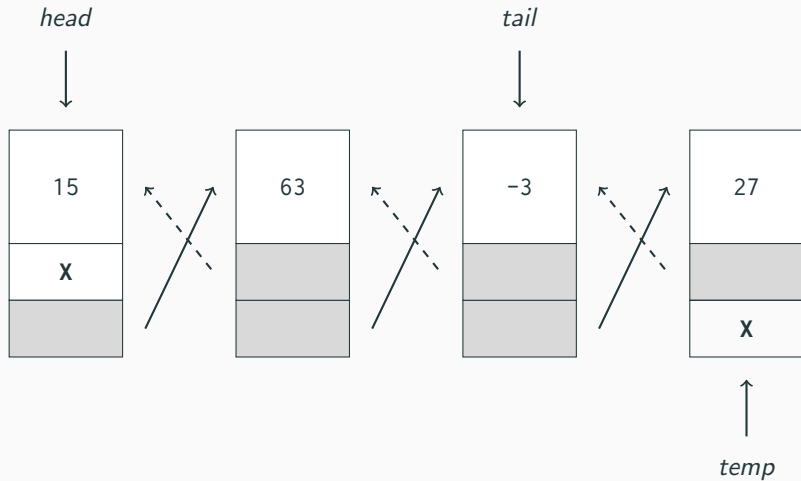
## Remoção do final da lista

- 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

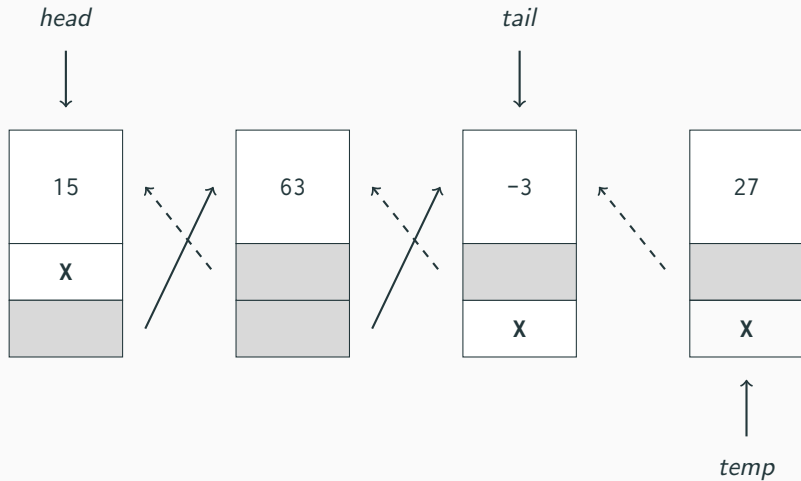
## Visualização da remoção no final da lista



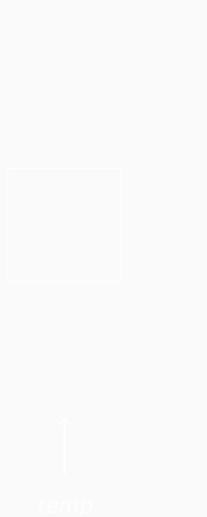
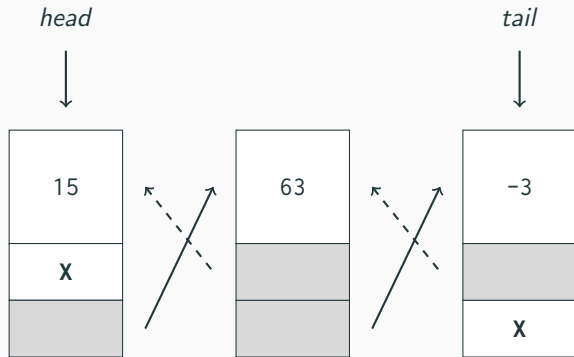
## Visualização da remoção no final da lista



## Visualização da remoção no final da lista



## Visualização da remoção no final da lista



## Implementação da remoção do final

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



## Remoção em posição arbitrária

- A remoção em posição arbitrária também tem complexidade  $O(1)$  ou  $O(N)$ , caso esteja disponível ou não um ponteiro para o elemento a ser removido
- Esta é uma rotina de implementação complexa, dado o grande número de *corner cases*
- 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

```
1 #include <iostream>
2 #include "list.h"
3
4 using namespace std;
5
6 int main()
7 {
8     List<int> list;
9
10    cout << "Lista vazia? " << (list.empty() ? "Sim" : "Nao") << endl;
11
12    for (int n = 1; n <= 10; ++n)
13        list.push_back(n);
14
15    list.pop_front();
16    cout << "Tamanho da lista: " << list.size() << endl;
17    cout << "Primeiro elemento: " << list.front() << endl;
18    cout << "Último elemento: " << list.back() << endl;
19
20    return 0;
21 }
```

## Listas duplamente encadeadas em C++

---

- Desde a versão C++, a linguagem C++ oferece uma implementação de listas duplamente encadeadas: o contêiner `list`
- Ao contrário da `forward_list`, seus iteradores são bidirecionais, permitindo a travessia em ambos sentidos
- Também de forma diferente da `forward_list`, este contêiner possui o método `size()`
- As inserções e remoções constantes devem ser feitas através dos métodos `push_front()`, `push_back()`, `pop_front()` e `pop_back()`
- Também está disponível o método `insert()`, que insere um elemento logo após o iterador indicado em complexidade constante

## Exemplo de uso de list

```
1 #include <iostream>
2 #include <list>
3
4 using namespace std;
5
6 int main()
7 {
8     list<int> list;
9
10    cout << "Lista vazia? " << (list.empty() ? "Sim" : "Nao") << endl;
11
12    for (int n = 1; n <= 10; ++n)
13        list.push_front(n);
14    list.pop_back();
15
16    cout << "Tamanho da lista: " << list.size() << endl;
17    cout << "Primeiro elemento: " << list.front() << endl;
18    cout << "Último elemento: " << list.back() << endl;
19
20    return 0;
21 }
```

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<sup>1</sup>.

---

<sup>1</sup><https://en.cppreference.com/w/>