

# Listas Encadeadas

## Fundamentos

---

Prof. Edson Alves - UnB/FGA

2018

1. Definição
2. Inserção
3. Remoção
4. Listas encadeadas em C++

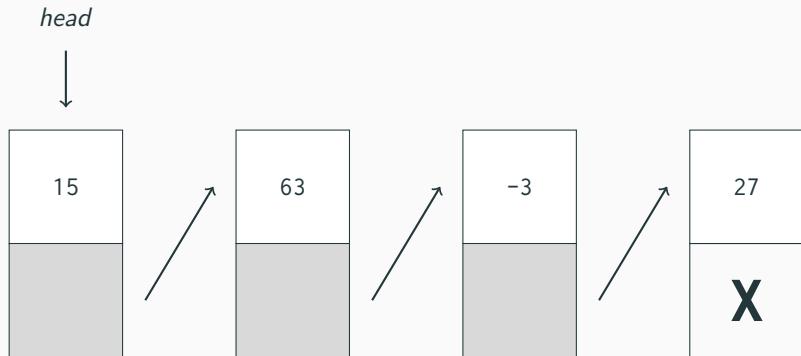
# Definição

---

# Listas encadeadas

- Uma lista encadeada, ou simplesmente lista, é uma estrutura composta por nós, onde cada nó armazenada uma informação e um ponteiro para o próximo nó da lista
- Conhecido o primeiro elemento da lista (*head*), é possível acessar todos os demais elementos
- Uma lista é uma estrutura de dados linear, devido a travessia sequencial e ordenada de seus elementos
- As listas são uma alternativa aos vetores: a vantagem do acesso aleatório imediato dos vetores é substituída pela inserção e remoção eficientes
- Por conta da estrutura dos nós, o acesso aleatório em listas encadeadas tem complexidade  $O(N)$

## Visualização de uma lista encadeada



# Implementação de uma lista encadeada

- Uma lista pode ser implementada como uma **struct** em C ou uma classe em C++
- A lista deve ter, no mínimo, um membro para o primeiro elemento da lista (*head*)
- Cada nó deve ter, no mínimo, dois membros: um para armazenar as informações (*info*) e outro para representar o ponteiro para o próximo nó (*next*)
- O último elemento da lista tem membro *next* nulo
- A adição de um ponteiro para o último elemento da lista (*tail*) aumenta a memória usada pela lista, mas permite a inserção ao final em complexidade  $O(1)$

# Exemplo de implementação de uma lista 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 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 encadeada

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

# Inserção

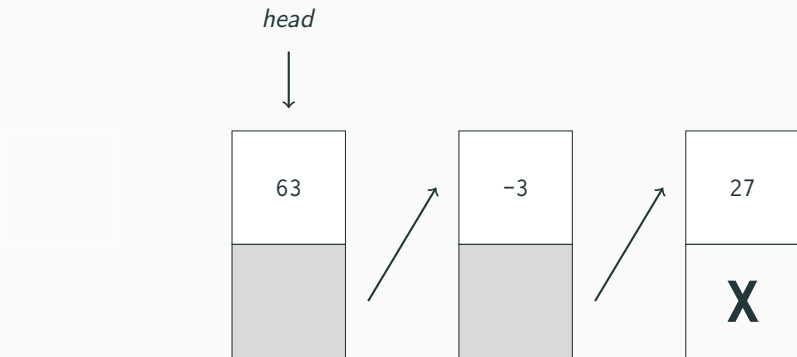
---

# Inserção no início

- A inserção no início (`push_front()`) de uma lista 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 `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

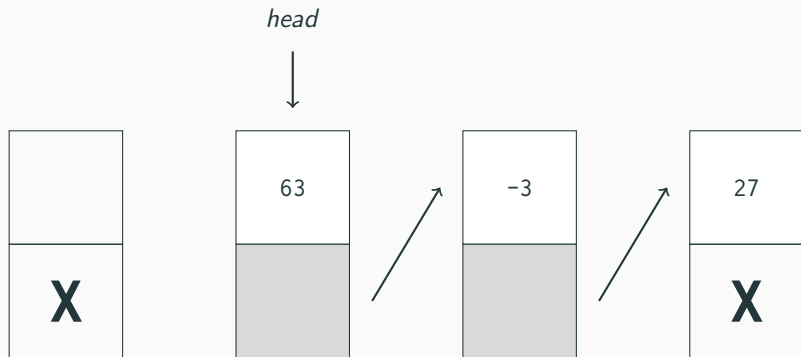
## Visualização da inserção no início

Informação a ser inserida: 88



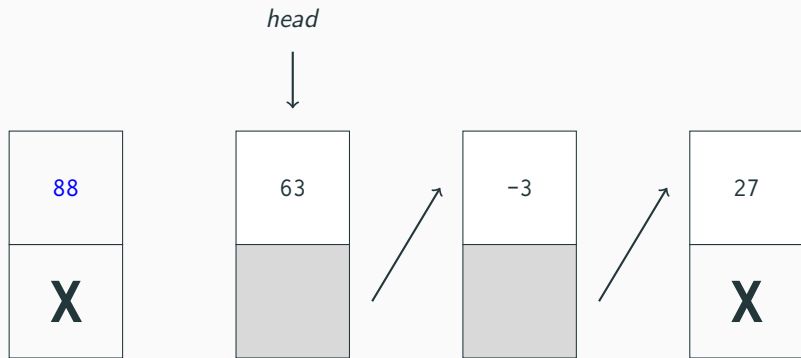
# Visualização da inserção no início

**Passo 01:** Criar um novo nó



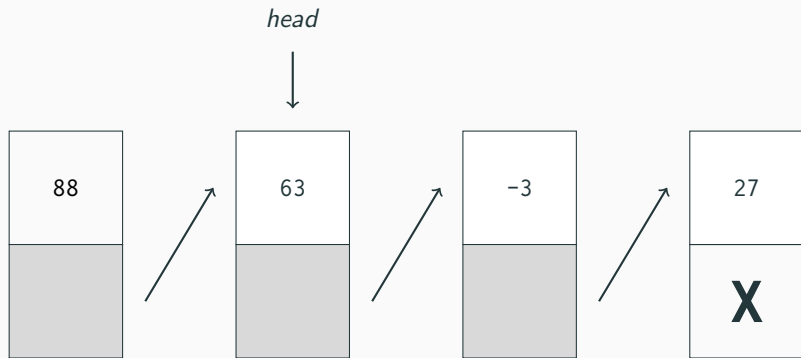
# Visualização da inserção no início

**Passo 02:** Preencher o campo info



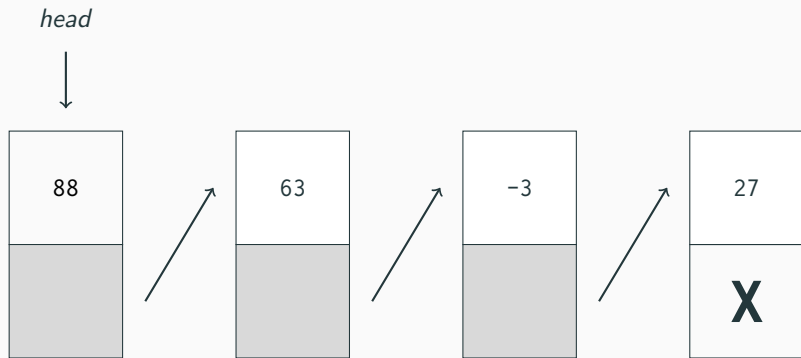
## Visualização da inserção no início

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



# Visualização da inserção no início

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





# Implementação da inserção no início

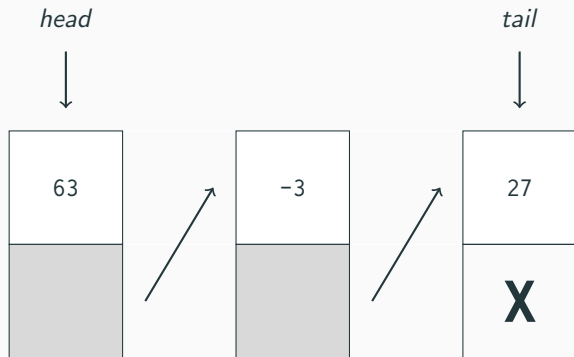
```
41 void push_front(const T& info)
42 {
43     auto n = new Node(info, head);
44
45     tail = head ? tail : n;
46     head = n;
47     _size++;
48 }
```

# Inserção no final

- A inserção no final (`push_back()`) de uma lista encadeada tem complexidade  $O(1)$ , desde que a classe tenha o membro `tail`
- O primeiro passo da inserção é criar um novo nó
- Em seguida, deve ser preenchido o campo `info`
- 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

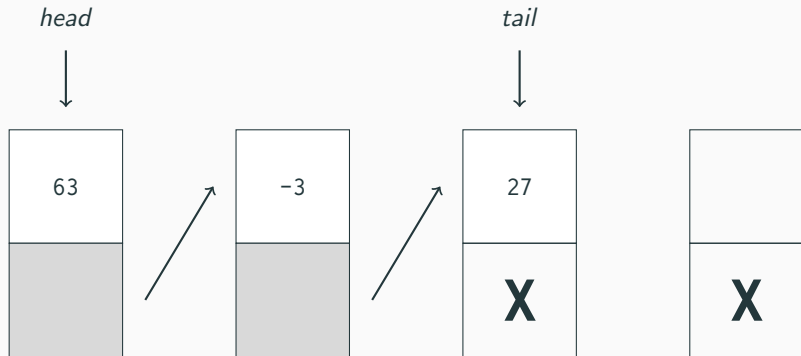
## Visualização da inserção no final

Informação a ser inserida: 88



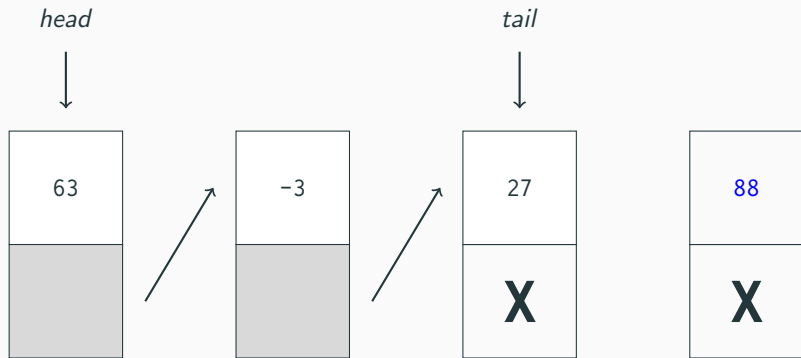
# Visualização da inserção no final

**Passo 01:** Criar um novo nó



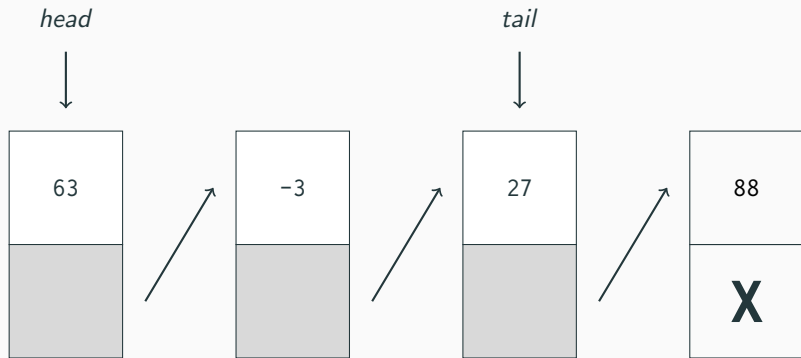
## Visualização da inserção no final

**Passo 02:** Preencher o campo info



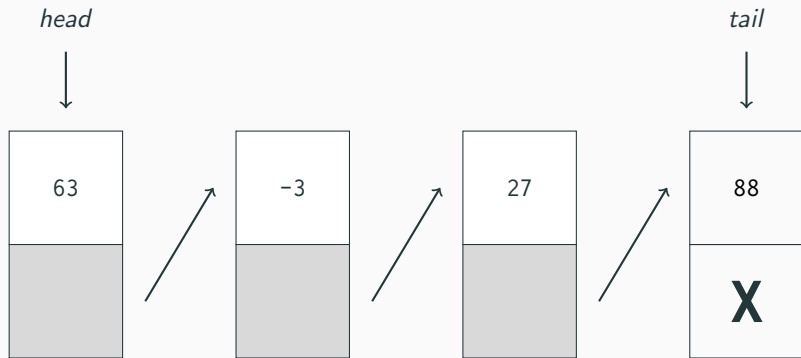
# Visualização da inserção no final

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



## Visualização da inserção no final

**Passo 04:** Apontar tail para o novo nó



# Implementação da inserção no final

```
50 void push_back(const T& info)
51 {
52     auto n = new Node(info, 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
- Primeiramente é necessário localizar a posição da inserção
- Além disso, é preciso identificar, se existir, o elemento que sucede o elemento que ocupa a posição de inserção
- O membro next do novo elemento deve apontar para o elemento que ocupa a posição de inserção
- O membro next do nó que antecedia o elemento da posição de inserção deve apontar para o novo nó
- É preciso tomar cuidado com vários *corner cases*:
  1. Lista vazia
  2. Apenas um elemento na lista
  3. Inserção na primeira posição
  4. Posição solicitada inválida

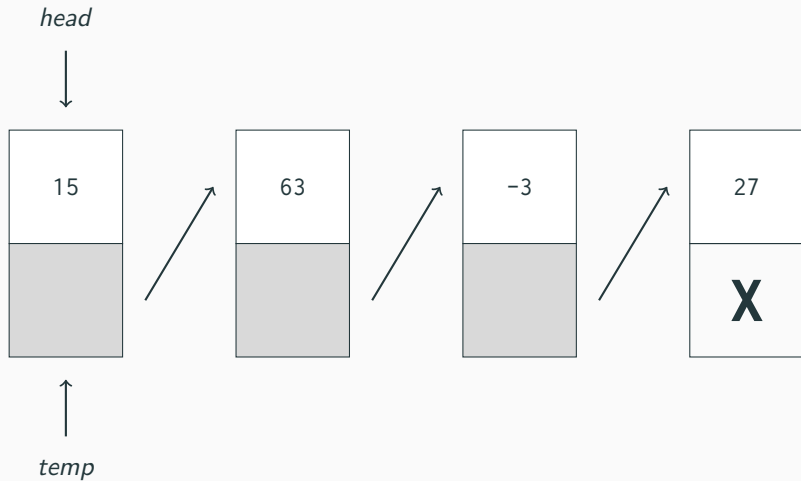
# Remoção

---

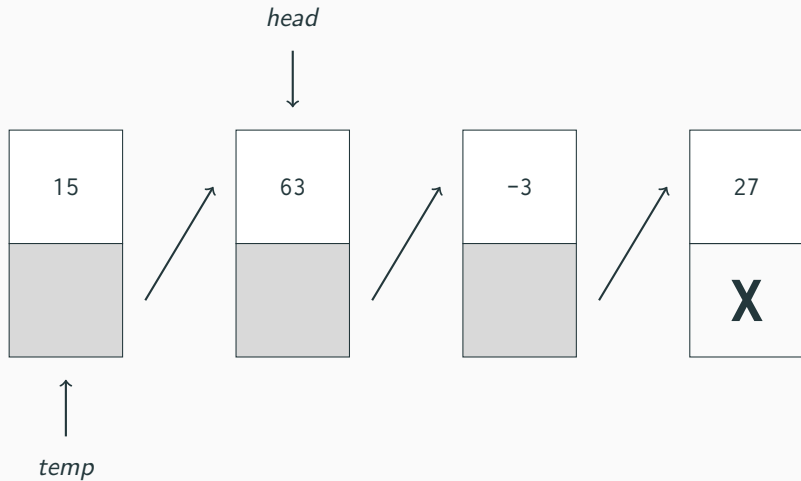
## Remoção do início

- A remoção de um elemento do início de uma lista (`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
- 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

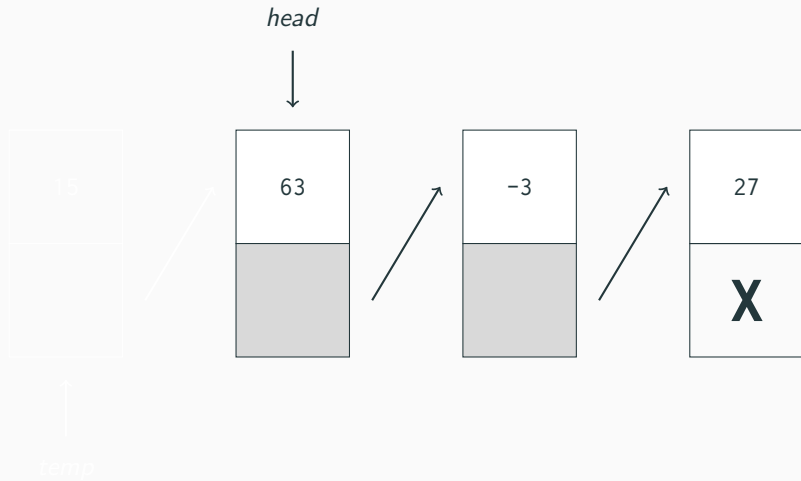
## Visualização da remoção no início da lista



## Visualização da remoção no início da lista



## Visualização da remoção no início da lista



# Implementação da remoção do início

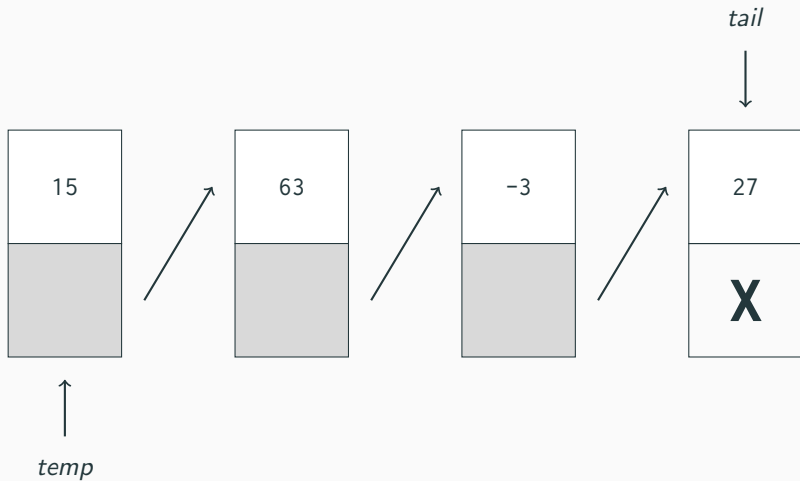
```
59 void pop_front()
60 {
61     if (!head)
62         throw "Lista vazia";
63
64     auto temp = head;
65     head = head->next;
66     delete temp;
67
68     tail = head ? tail : nullptr;
69     _size--;
70 }
```

## Remoção do final da lista

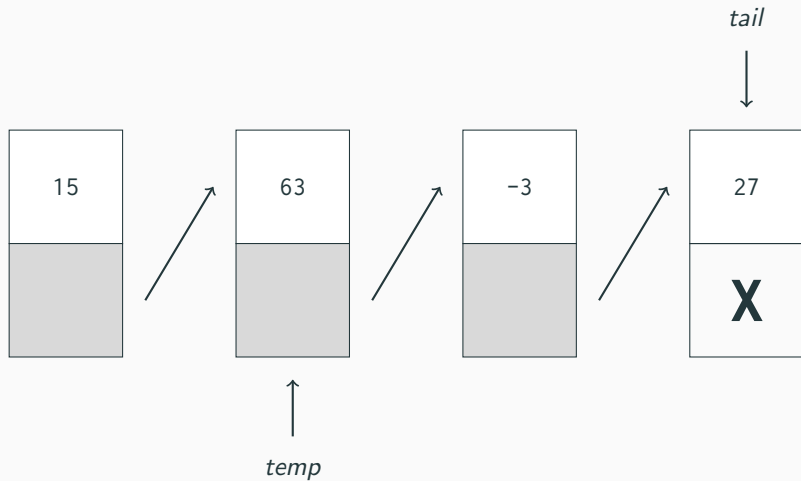
- Mesmo com o membro `tail`, a remoção do último elemento de uma lista encadeada (`pop_back()`) tem complexidade  $O(N)$ , onde  $N$  é o número de nós da lista
- Isto acontece porque é preciso localizar o elemento que antecede o último elemento (`prev`), processo que tem complexidade linear
- Localizado 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



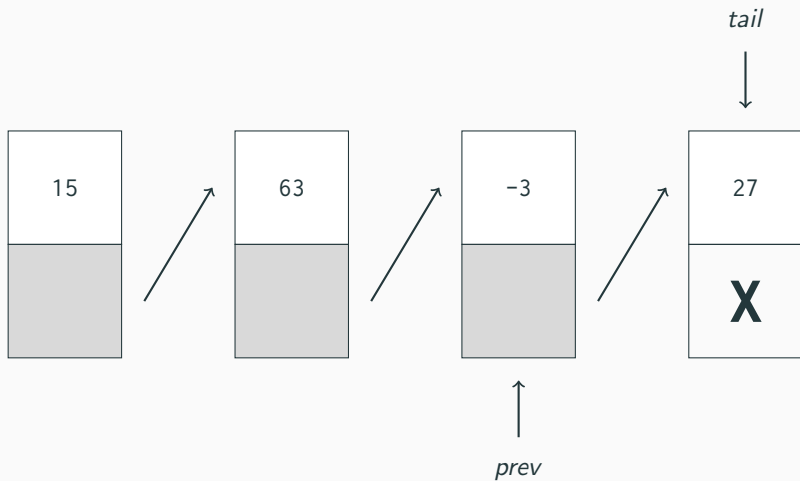
## Visualização da remoção no final da lista



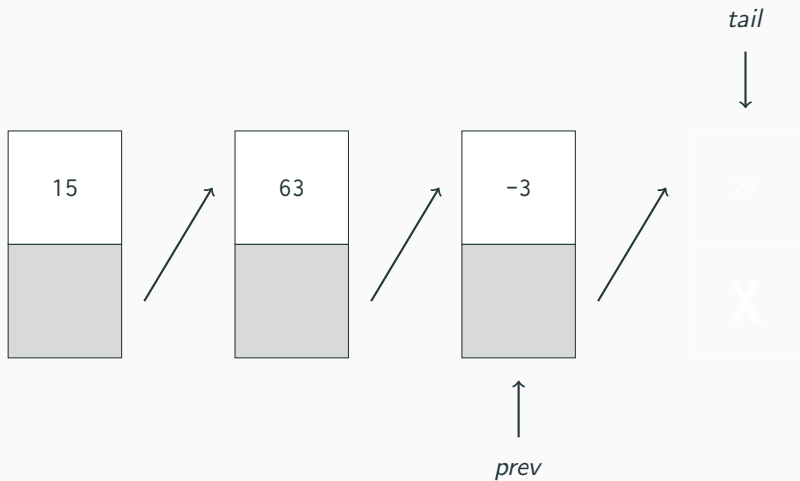
## Visualização da remoção no final da lista



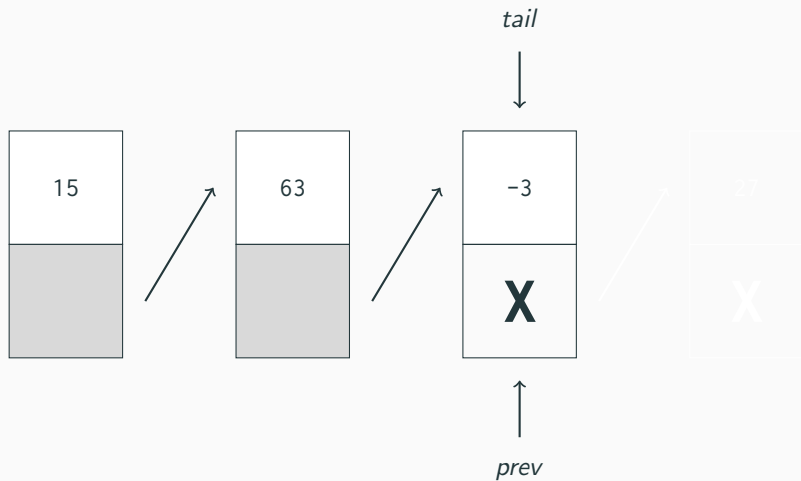
## Visualização da remoção no final da lista



## 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 antecede 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

```
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_front(n);
14
15    list.pop_back();
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 encadeadas em C++

---

- 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

```
1 #include <iostream>
2 #include <algorithm>
3 #include <forward_list>
4
5 using namespace std;
6
7 int main()
8 {
9     forward_list<int> list;
10
11     cout << "Lista vazia? " << (list.empty() ? "Sim" : "Nao") << endl;
12
13     for (int n = 1; n <= 10; ++n)
14         list.push_front(n);
15
16     list.pop_front();
17
18     // Complexidade linear
19     int size = distance(list.begin(), list.end());
20     cout << "Tamanho da lista: " << size << endl;
21
```

## 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 }
```

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/>