Listas

Listas Circulares

Prof. Edson Alves - UnB/FGA

2018

Sumário

1. Definição

2. Implementação

3. Inserção

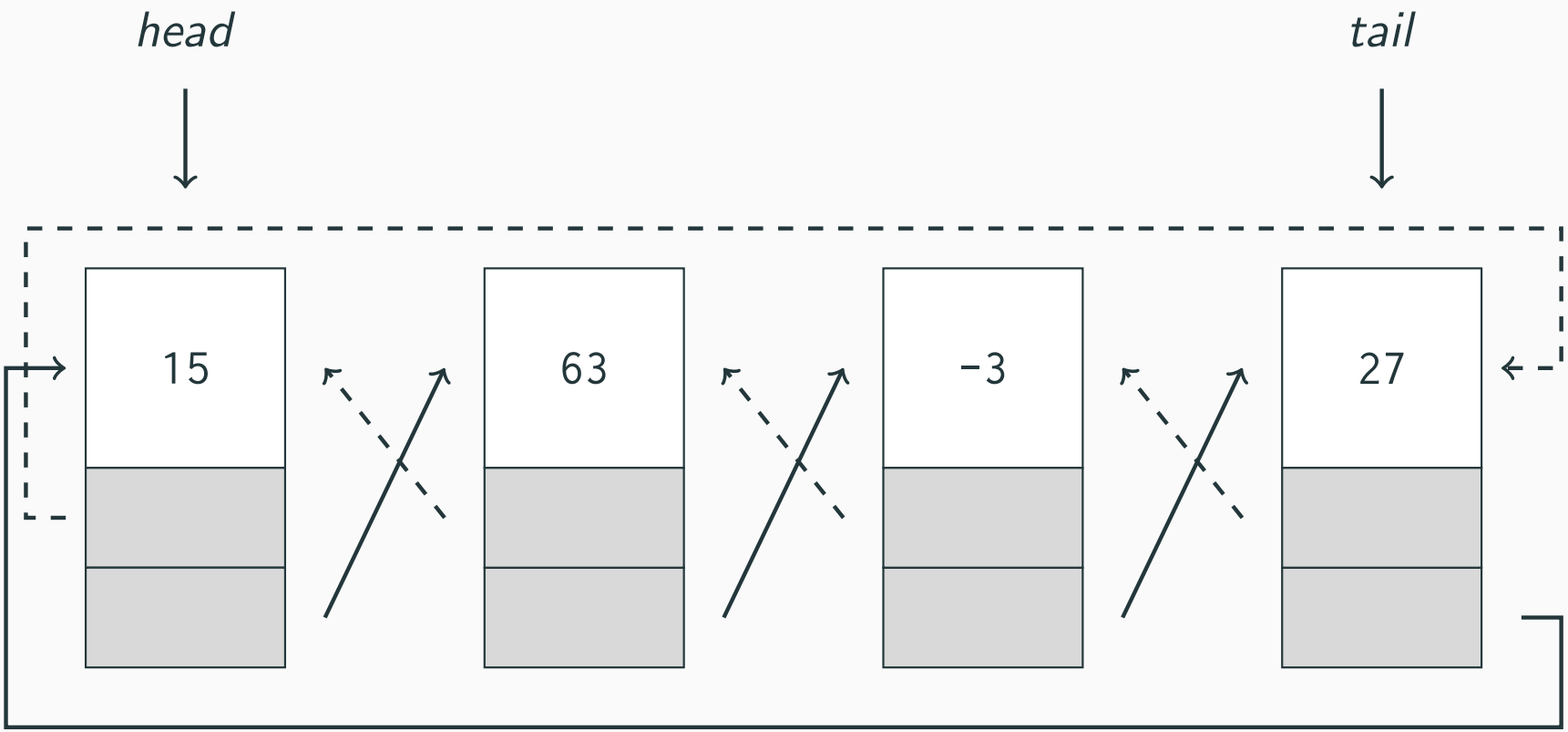
4. Remoção

Definição

Listas circulares

- Uma lista circular é uma lista onde o membro `next` do último elemento aponta para o primeiro elemento, e o campo `prev` do primeiro elemento aponta para o último elemento, quando for o caso
- A lista forma um caminho cíclico entre seus elementos, de modo que a identificação do primeiro ou do último elemento é, de fato, relativa
- Uma travessia deve marcar um ponto de partida e progredir até que atinja este ponto novamente, encerrando assim a travessia
- Em geral, as listas circulares são implementadas como listas encadeadas
- As operações de inserção e remoção em uma posição conhecida tem complexidade constante

Visualização de uma lista circular



Implementação

Implementação de uma lista circular

- Uma lista circular pode ser implementada a partir de adaptações pontuais nas implementações das listas encadeadas ou duplamente encadeadas
- A lista que servirá como ponto de partida deve ser escolhida de acordo com a memória disponível ou dos sentidos de travessia desejados
- Deve ser mantida a invariante que o último elemento aponta para o primeiro após cada operação (e que o primeiro aponta para o último, no caso das listas duplamente encadeadas)
- É preciso oferecer um iterador circular, que permita a travessia circular dos elementos da lista

Exemplo de implementação de uma lista circular

```
1  #ifndef CIRCULAR_LIST_H
2  #define CIRCULAR_LIST_H
3
4  #include <ostream>
5  #include <initializer_list>
6
7  template<typename T>
8  class CircularList {
9
10     friend
11     std::ostream& operator<<(std::ostream& os, const CircularList& list) {
12         os << "[";
13
14         if (not list.empty()) {
15             auto it = list.begin();
16
17             do {
18                 os << " " << *it;
19                 it++;
20             } while (it != list.begin());
21         }
```

Exemplo de implementação de uma lista circular

```
23          os << " ]";
24
25          return os;
26      }
27
28  private:
29      struct Node {
30          T info;
31          Node *prev, *next;
32
33          Node(const T& i, Node *p, Node *n) : info(i), prev(p), next(n) {}
34      };
35
36      Node *head, *tail;
37      unsigned long _size;
38
39  public:
40      CircularList() : head(nullptr), tail(nullptr), _size(0) {}
41
42      CircularList(std::initializer_list<T> source)
43          : head(nullptr), tail(nullptr), _size(0)
```

Exemplo de implementação de uma lista circular

```
44      {
45          for (auto it = source.begin(); it != source.end(); ++it)
46              push_back(*it);
47      }
48
49      ~CircularList()
50      {
51          if (head == nullptr)
52              return;
53
54          auto p = head;
55
56          do {
57              auto next = p->next;
58              delete p;
59              p = next;
60          } while (p != head);
61      }
62
63      const T& front() const
64      {
```

Exemplo de implementação de uma lista circular

```
65          if (head)
66              return head->info;
67          else
68              throw "Empty list!";
69      }
70
71      const T& back() const
72      {
73          if (tail)
74              return tail->info;
75          else
76              throw "Empty list!";
77      }
78
79      bool empty() const { return head == nullptr; }
80
81      unsigned long size() const { return _size; }
82
83      class iterator {
84      public:
85          explicit iterator(Node *n) : node(n) {}
```

Exemplo de implementação de uma lista circular

```
87          // Prefix
88          iterator& operator++() {
89              node = node->next;
90              return *this;
91          }
92
93          // Posfix
94          iterator& operator++(int) {
95              return ++(*this);
96          }
97
98          iterator& operator--() {
99              node = node->prev;
100             return *this;
101         }
102
103         bool operator==(const iterator& it) const
104         {
105             return node == it.node;
106         }
107
```

Exemplo de implementação de uma lista circular

```
08          bool operator!=(const iterator& it) const
09          {
10              return not (*this == it);
11          }
12
13          T& operator*() const {
14
15              if (node == nullptr)
16                  throw "Invalid iterator";
17
18              return node->info;
19          }
20
21      private:
22          Node *node;
23      };
24
25      iterator begin() const {
26          return iterator(head);
27      }
28
```

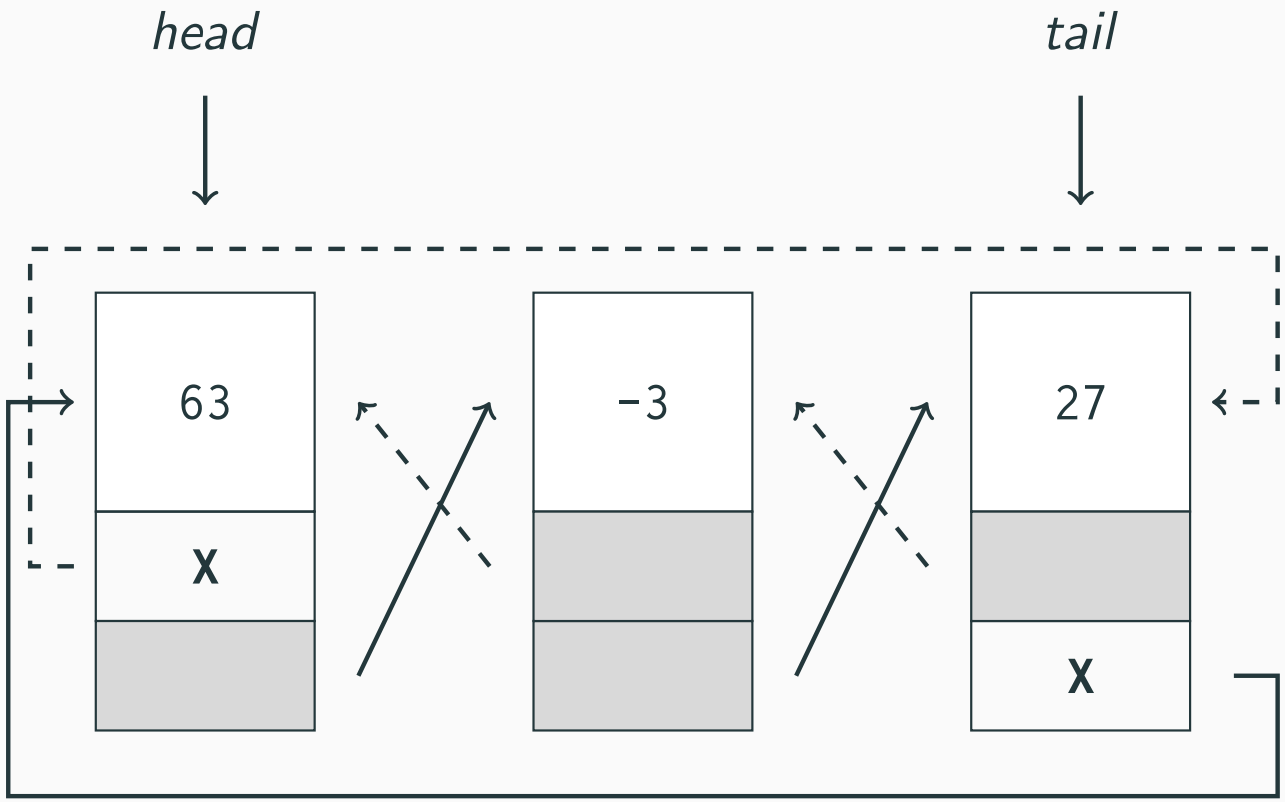
Inserção

Inserção no início

- A inserção no início (push_`front()`) de uma lista circular 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)
- O membro head deve apontar para o novo elemento
- Por fim, é preciso reestabelecer o invariante da lista circular: o campo `next` de `tail` tem que apontar para head e o campo `prev` de head tem que apontar para `tail`

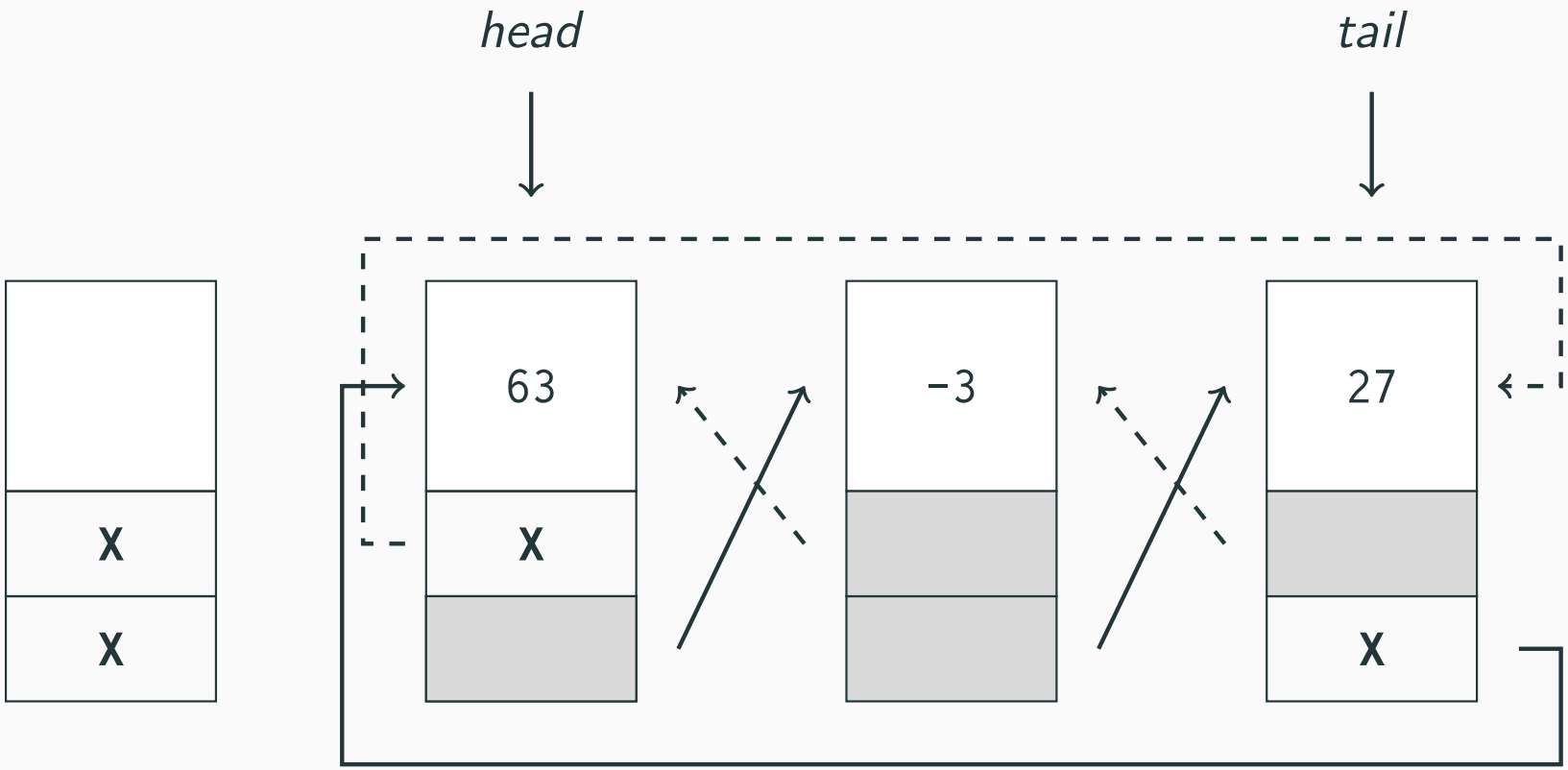
Visualização da inserção no início

Valor a ser inserido: 88



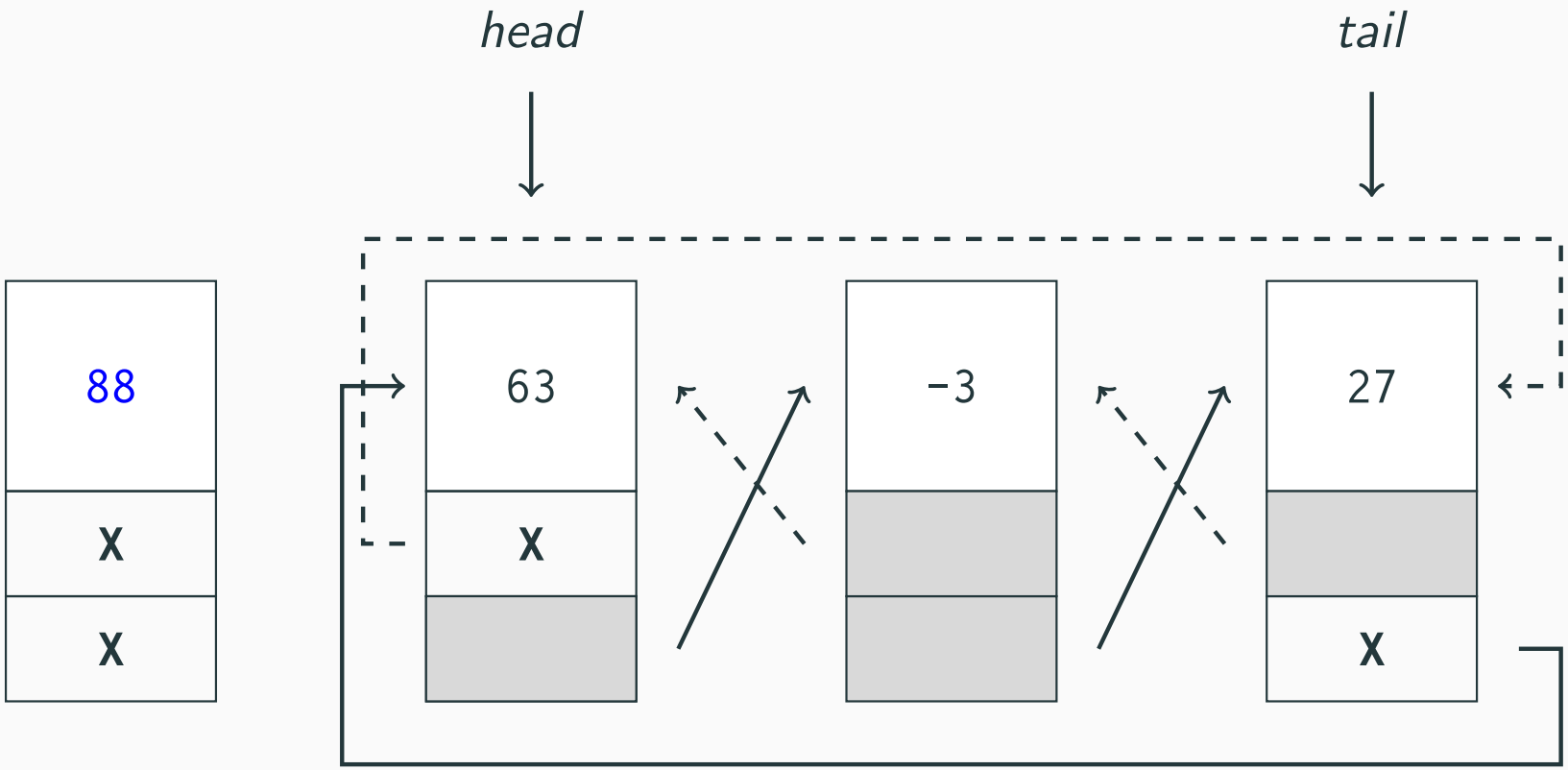
Visualização da inserção no início

Passo 01: Criar um novo nó



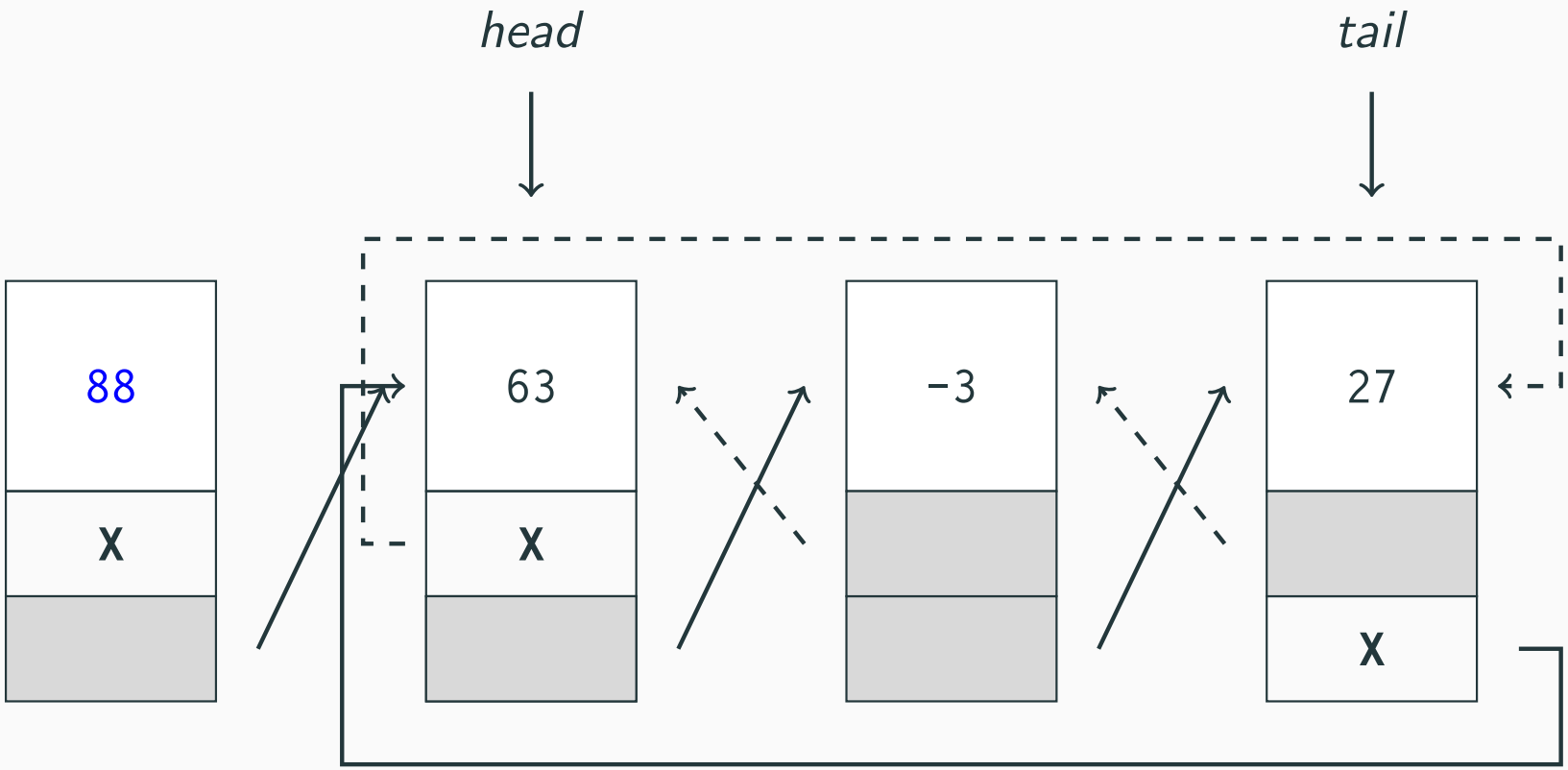
Visualização da inserção no início

Passo 02: Preencher o campo `info`



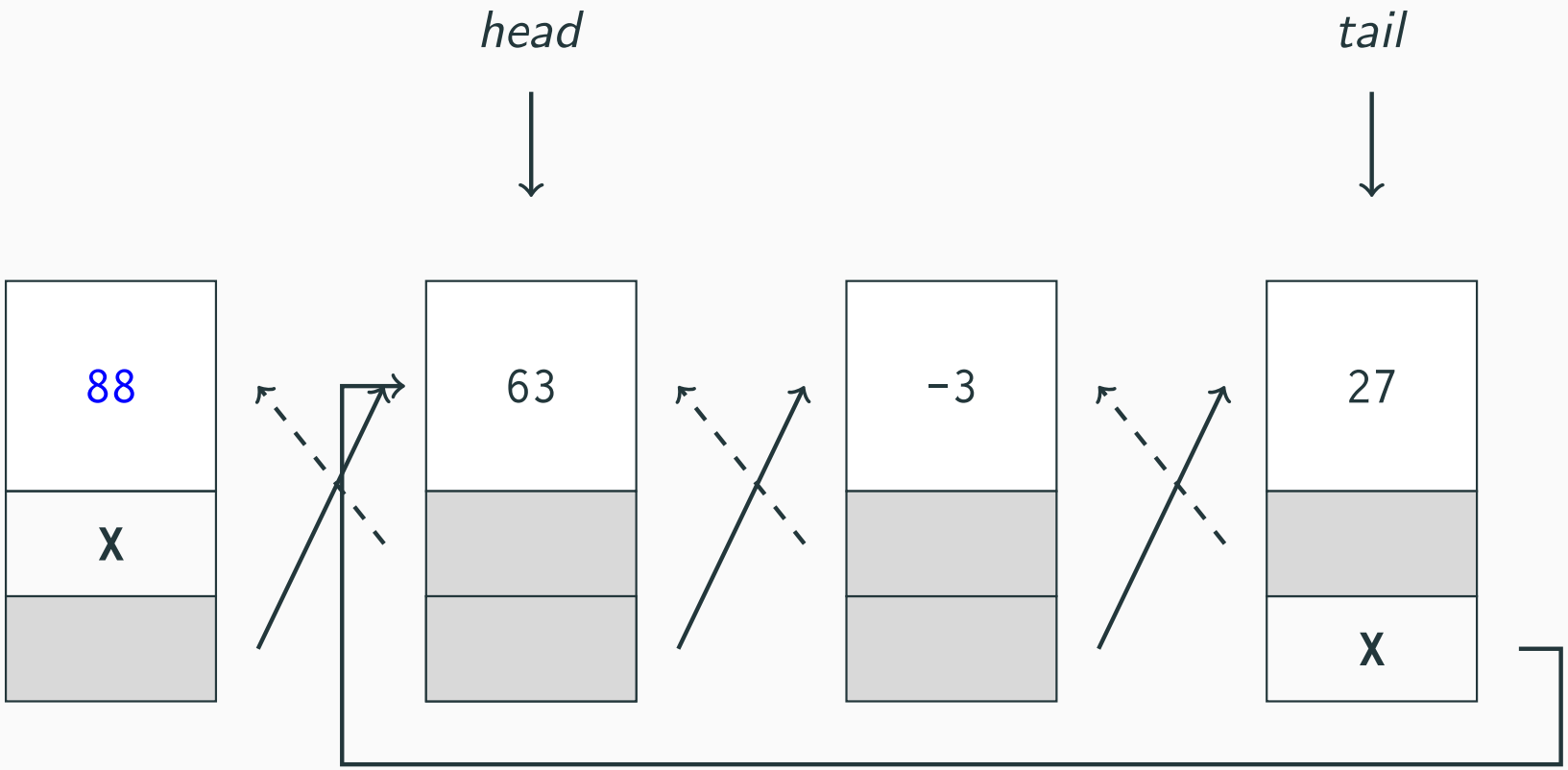
Visualização da inserção no início

Passo 03: Apontar next para o primeiro elemento



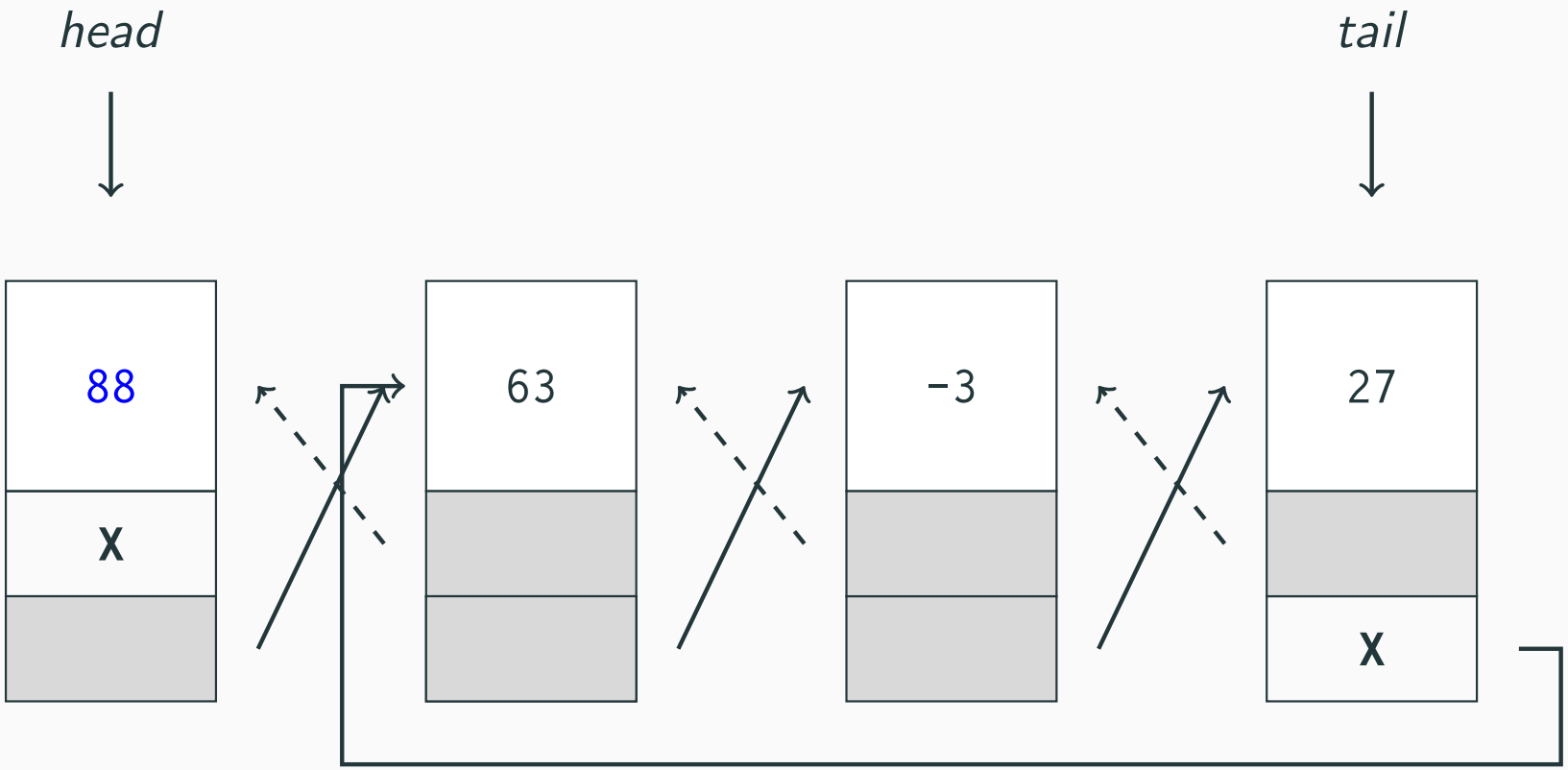
Visualização da inserção no início

Passo 04: Apontar prev de head para o novo elemento



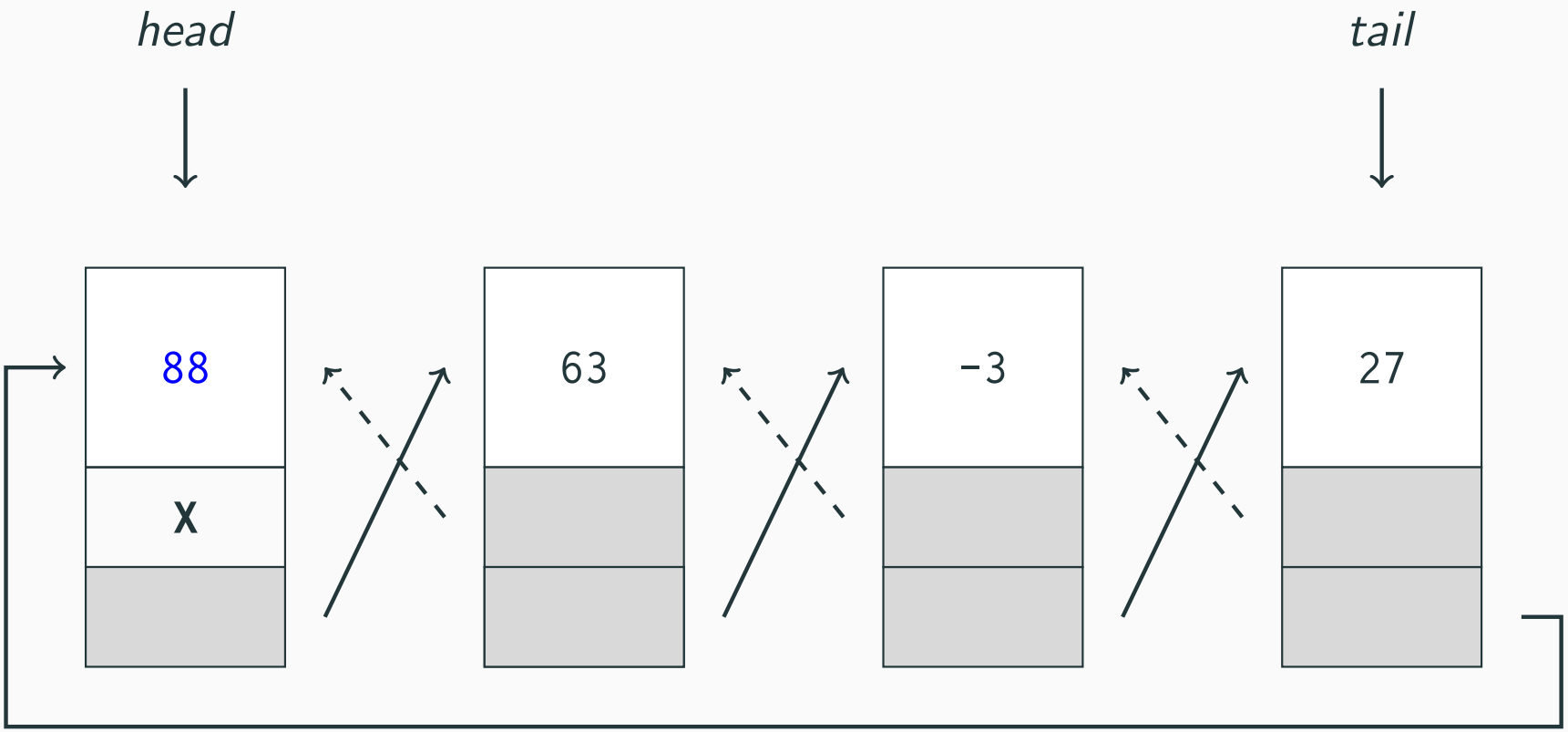
Visualização da inserção no início

Passo 05: Apontar head para o novo elemento



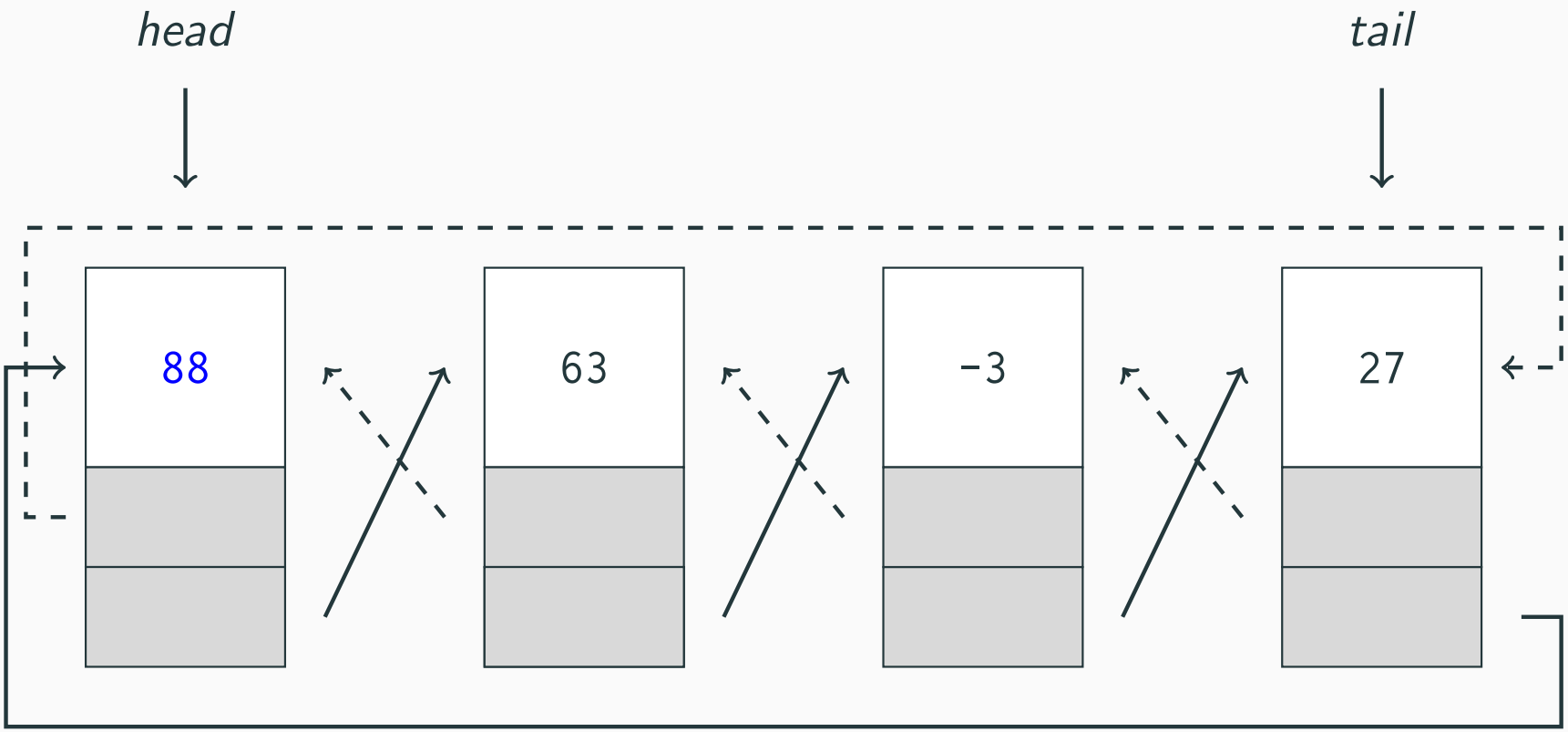
Visualização da inserção no início

Passo 06: Apontar next de `tail` para head



Visualização da inserção no início

Passo 07: Apontar prev de head para `tail`



Implementação da inserção no início

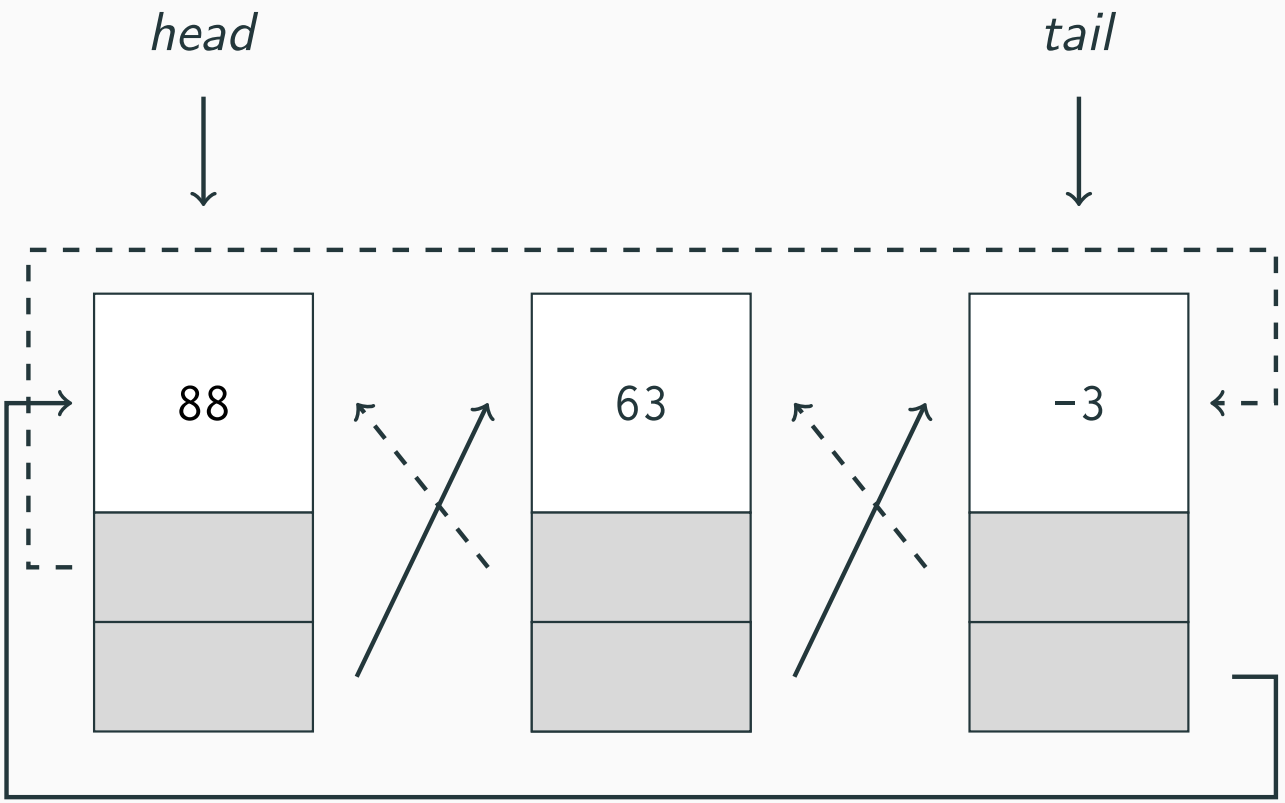
```
29      void push_front(const T& info)
30      {
31          auto n = new Node(info, nullptr, head);
32
33          head ? head->prev = n : tail = n;
34          head = n;
35
36          head->prev = tail;
37          tail->next = head;
38
39          _size++;
40      }
```

Inserção no final

- A inserção no final (push_back()) de uma lista circular 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
- O membro `tail` deve apontar para o novo elemento
- Por fim, é preciso reestabelecer o invariante da lista circular: o campo `next` de `tail` tem que apontar para head e o campo `prev` de head tem que apontar para `tail`

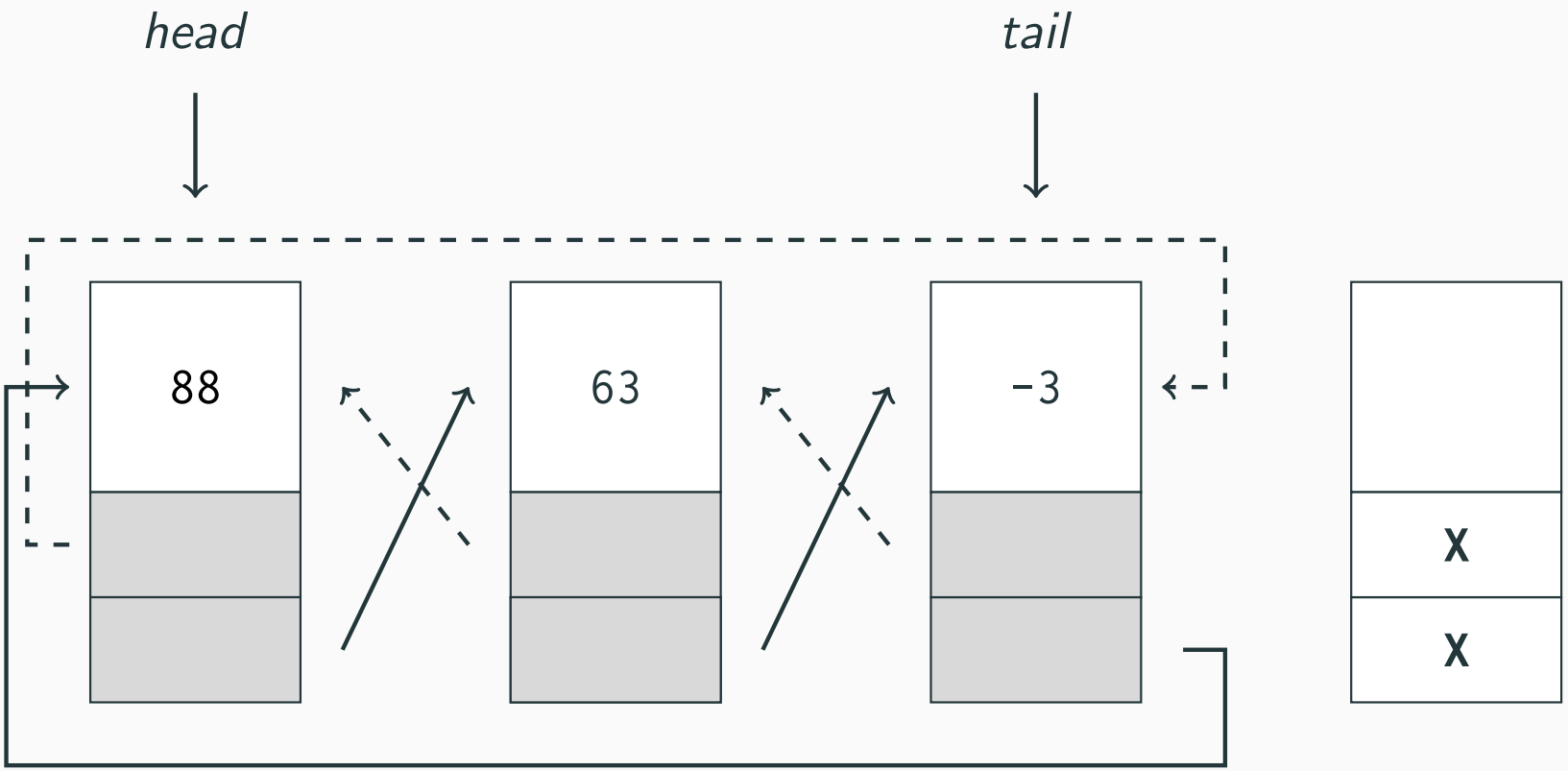
Visualização da inserção no final

Valor a ser inserido: 15



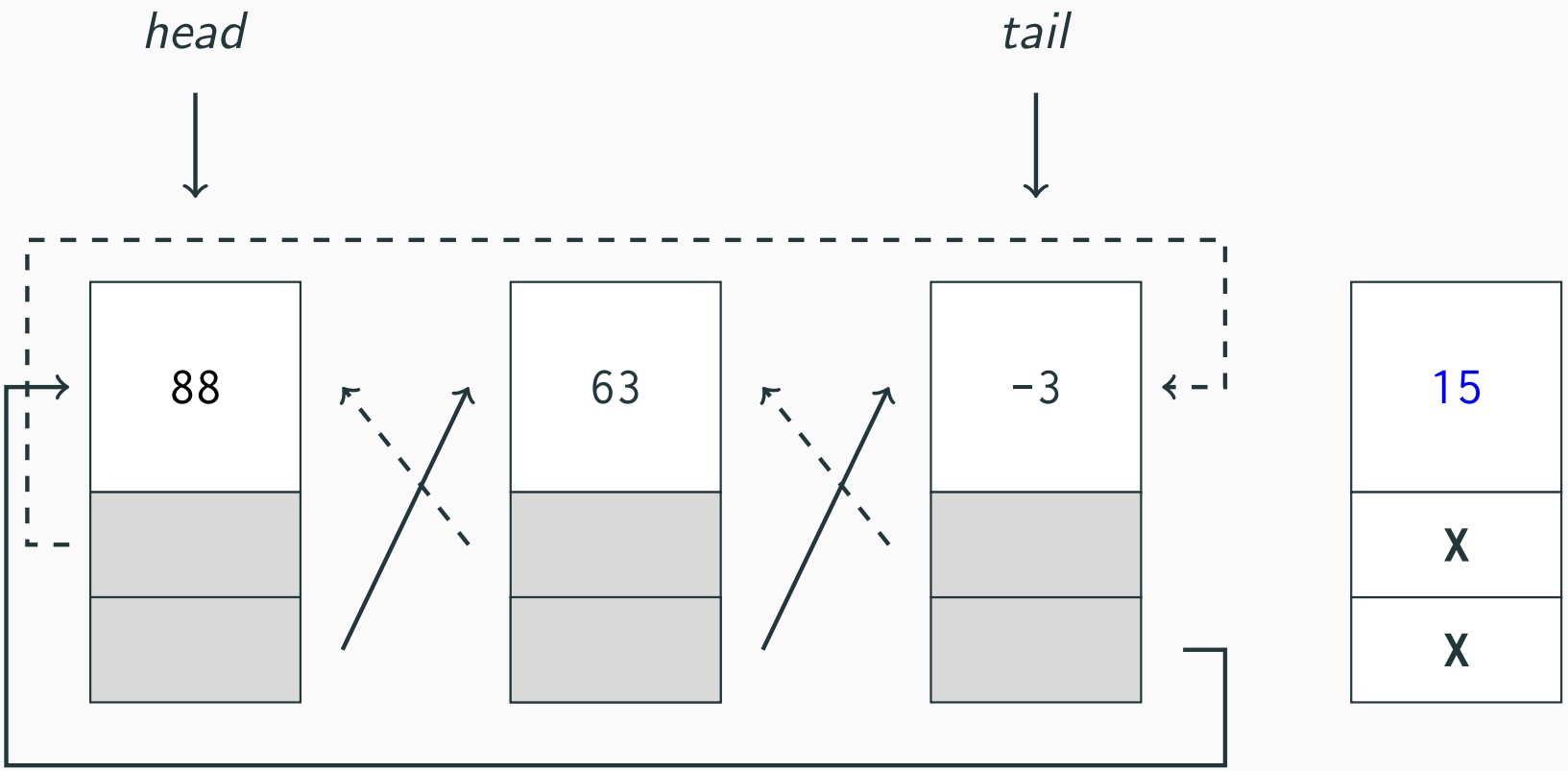
Visualização da inserção no final

Passo 01: Criar um novo nó



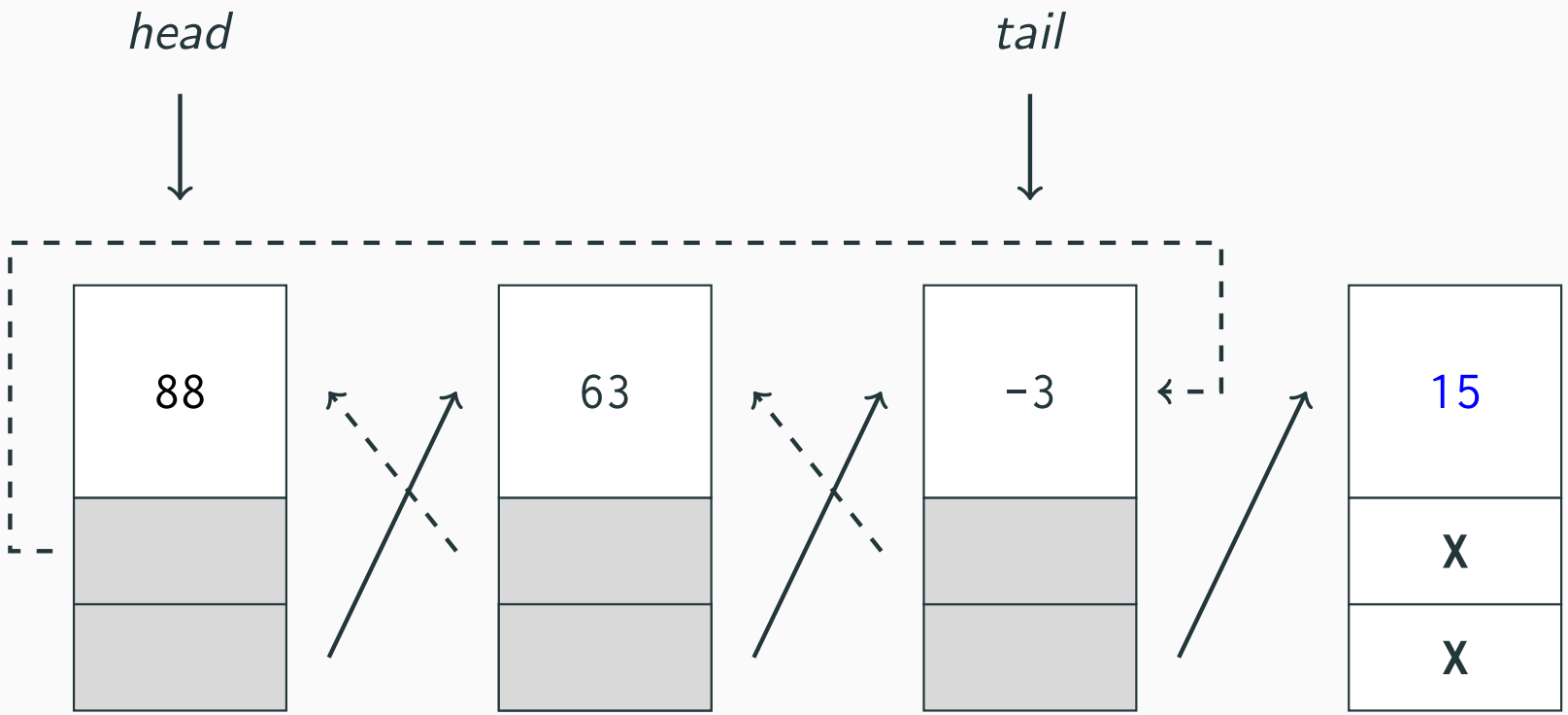
Visualização da inserção no final

Passo 02: Preencher o campo `info`



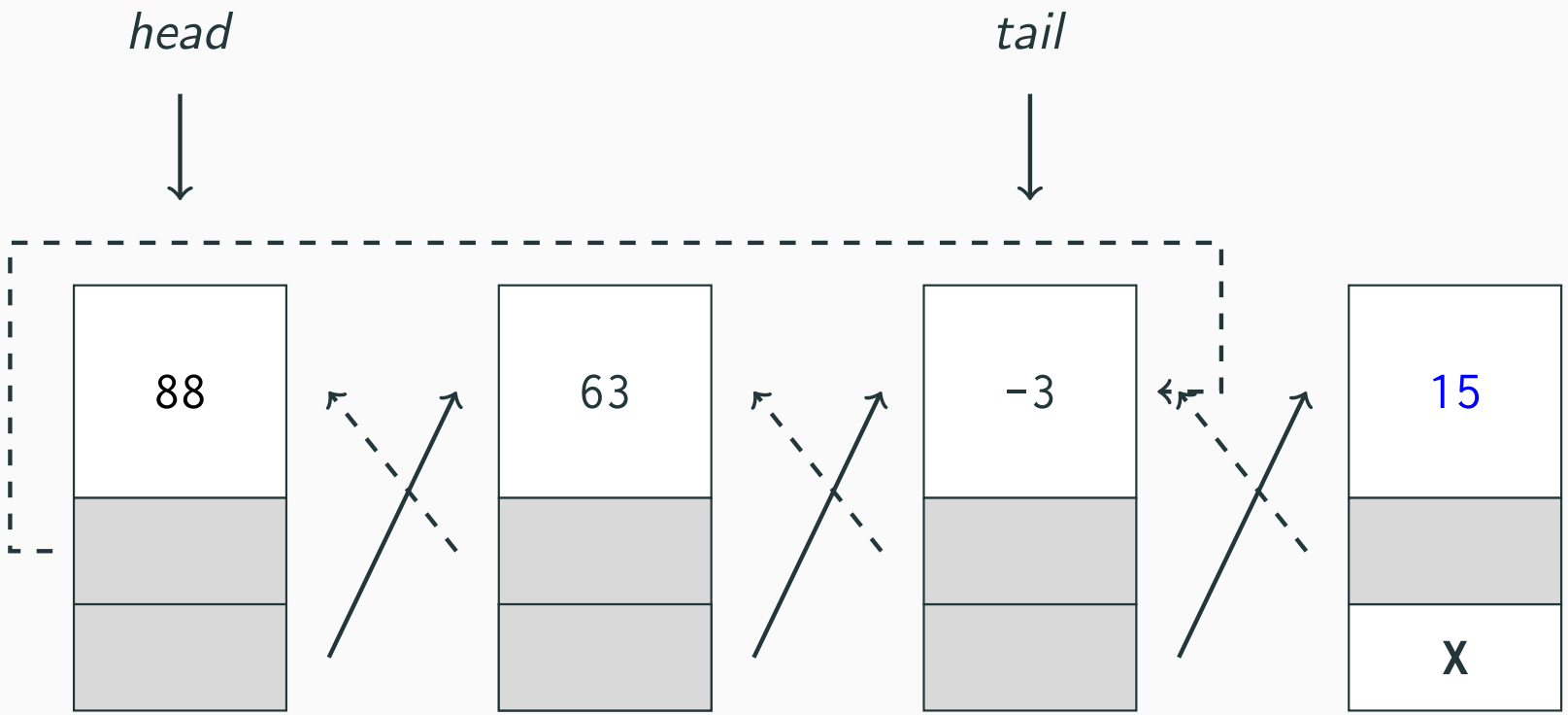
Visualização da inserção no final

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



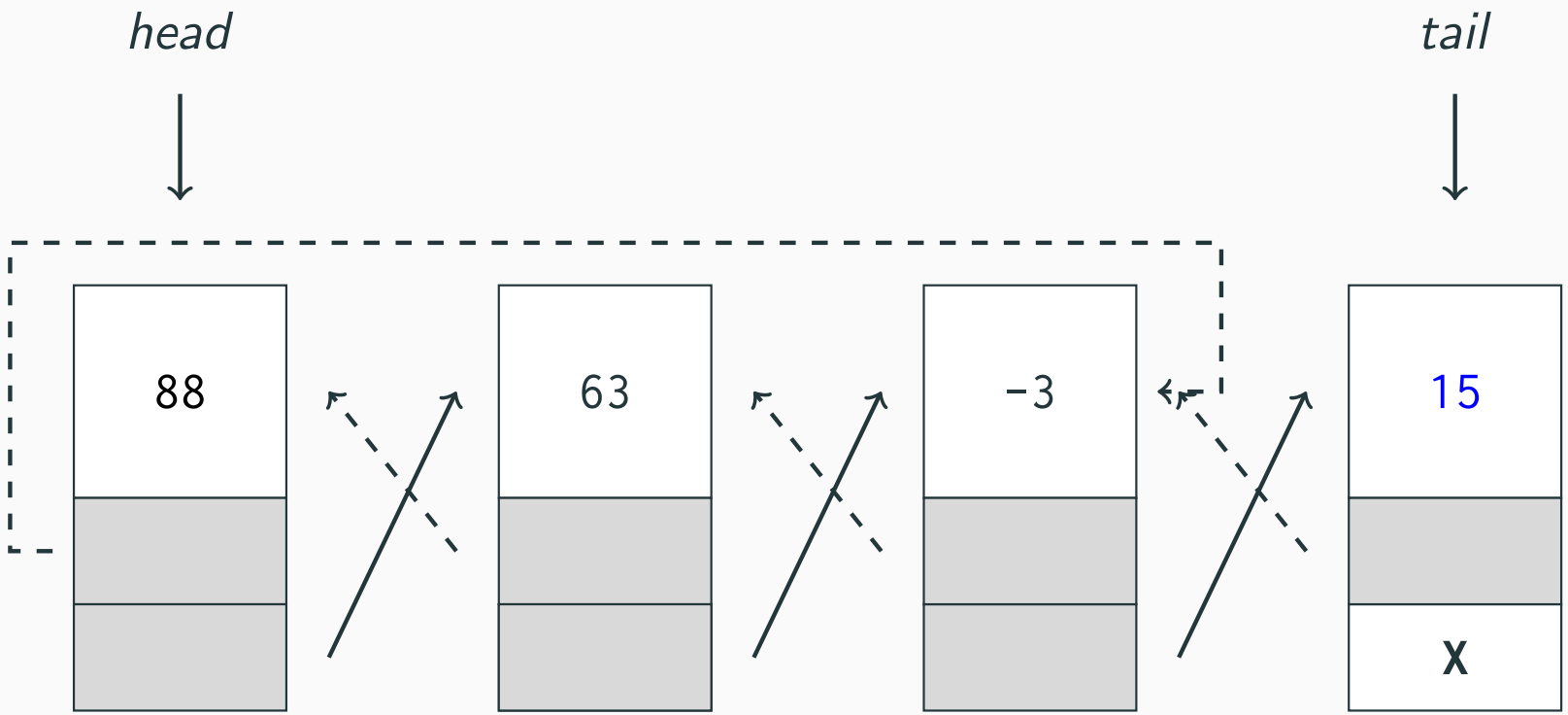
Visualização da inserção no final

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



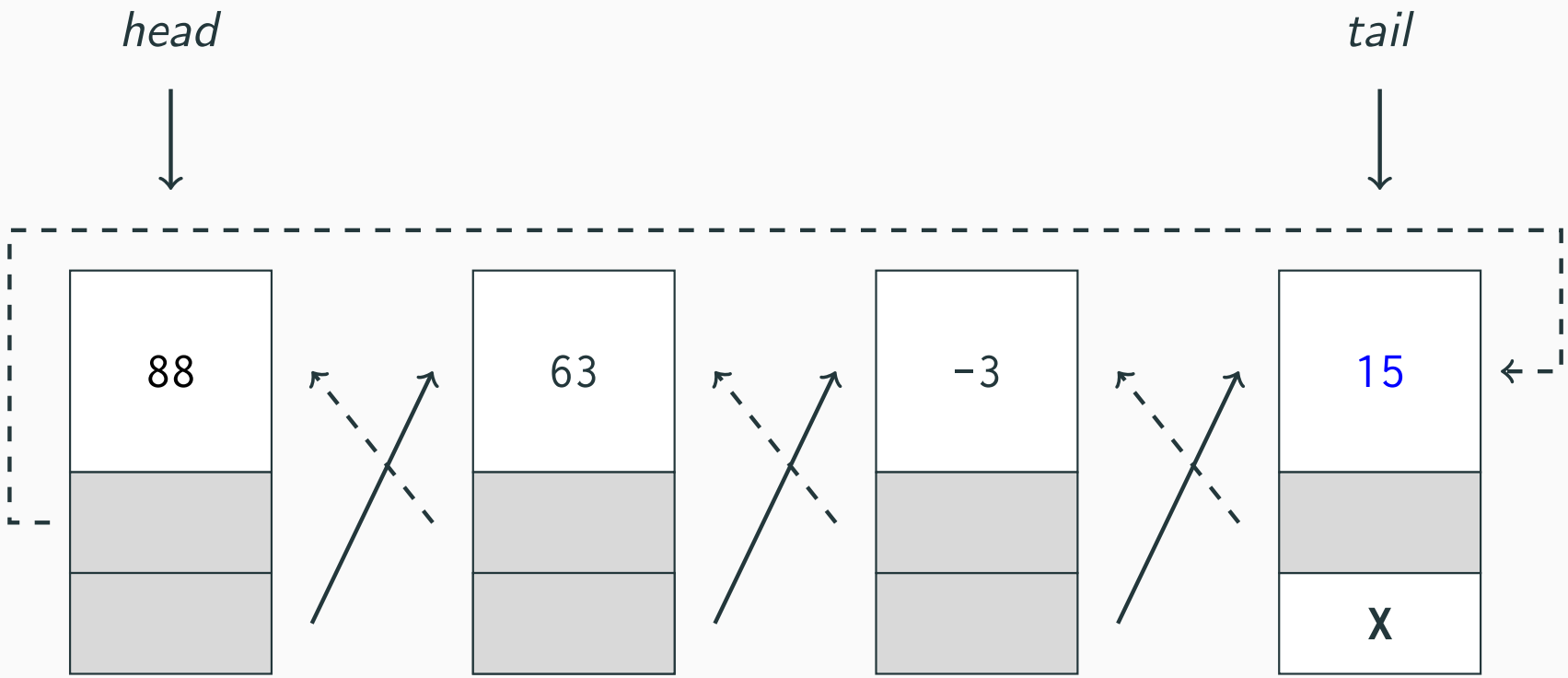
Visualização da inserção no final

Passo 05: Apontar `tail` para o novo nó



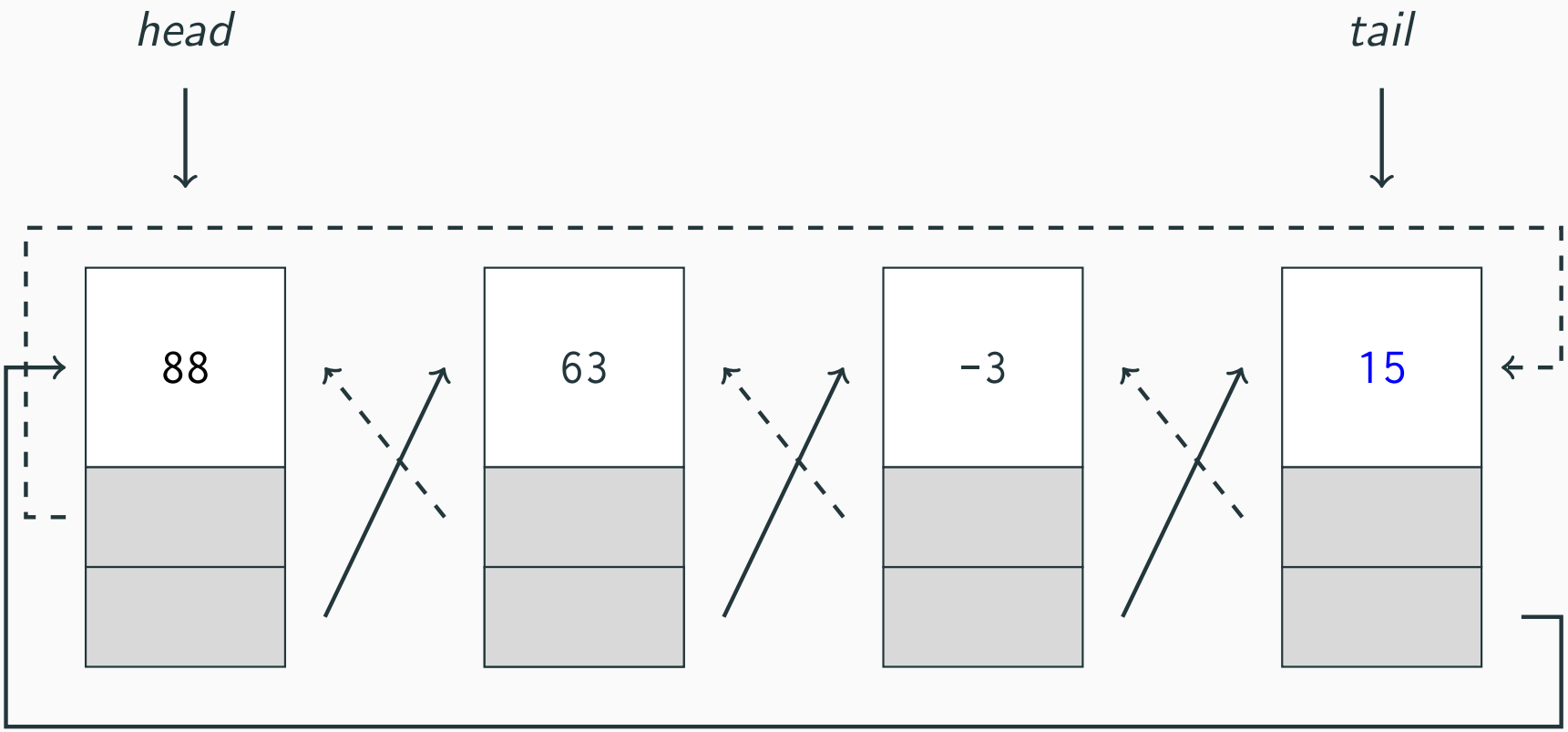
Visualização da inserção no final

Passo 06: Apontar prev de head para `tail`



Visualização da inserção no final

Passo 07: Apontar next de `tail` para head



Implementação da inserção no final

```
42      void push_back(const T& info)
43      {
44          auto n = new Node(info, tail, nullptr);
45
46          tail ? tail->next = n : head = n;
47          tail = n;
48
49          tail->next = head;
50          head->prev = tail;
51
52          _size++;
53      }
```

Inserção em posição arbitrária

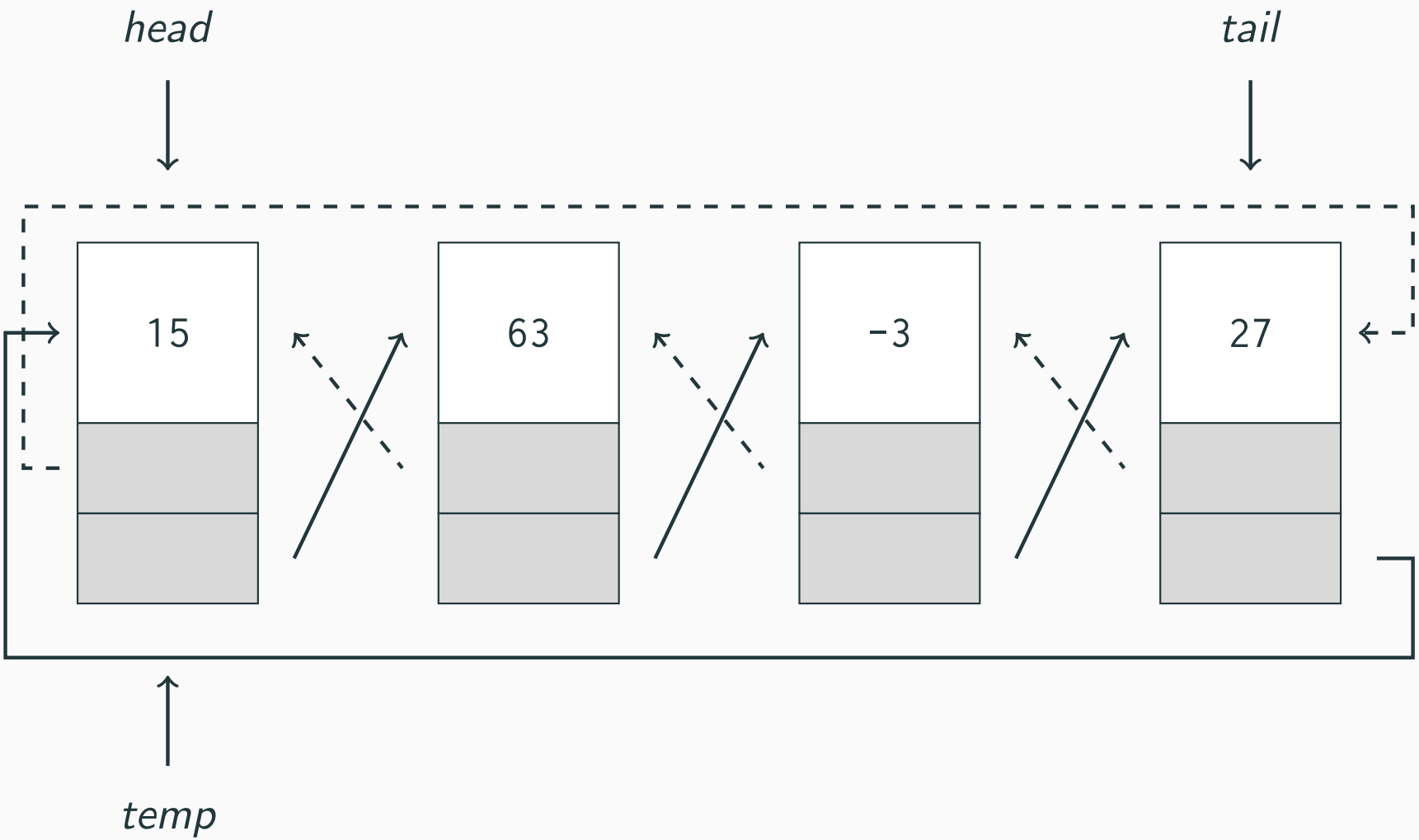
- A inserção em posição arbitrária tem complexidade $O(N)$, onde N é o número de elementos da lista, ou $O(1)$, caso o ponteiro para a posição onde o elemento será inserido seja conhecido
- Observe que, no caso dos vetores, mesmo que a posição de inserção seja conhecida, a complexidade permanece sendo $O(N)$
- 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

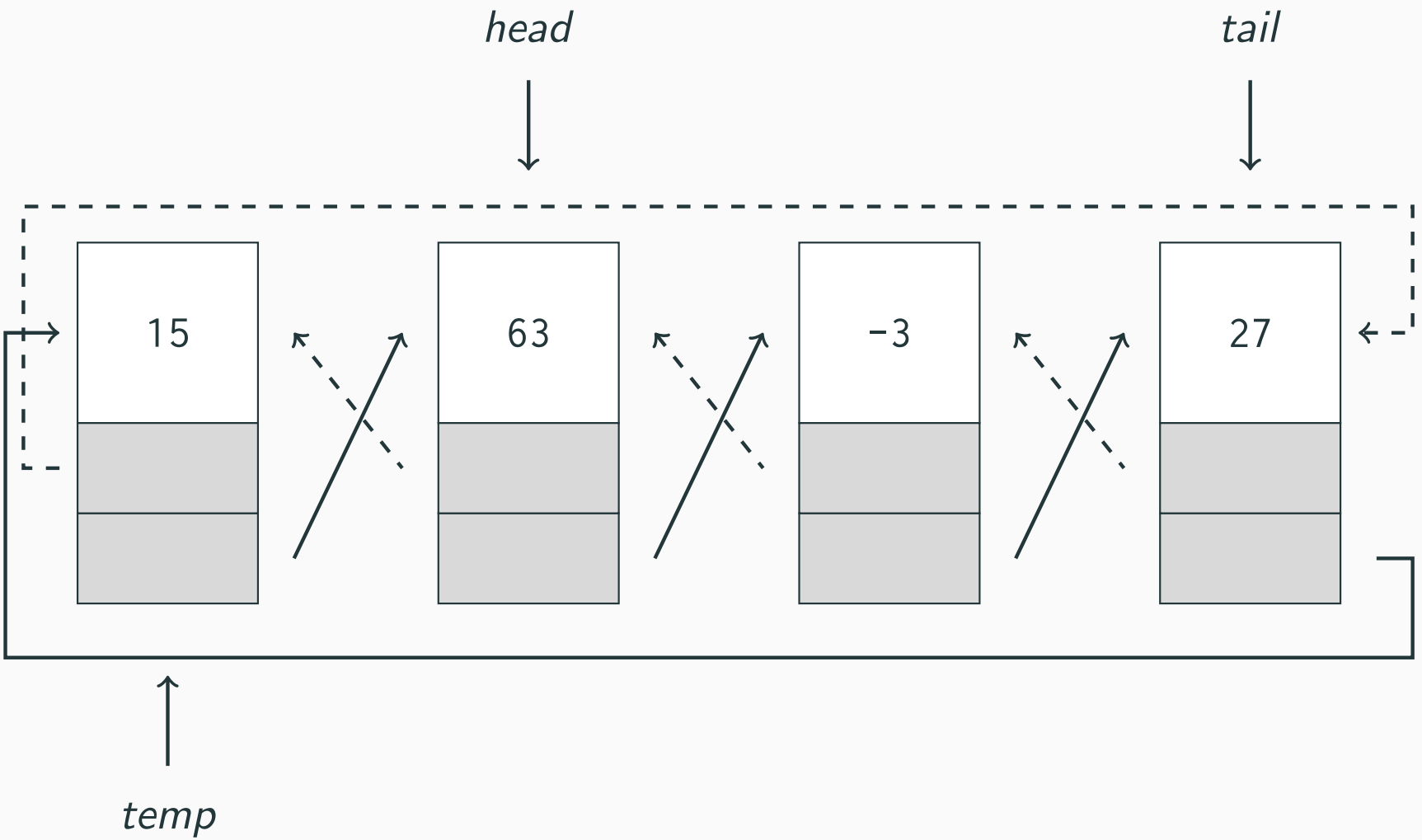
Remoção do início

- A remoção de um elemento do início de uma lista circular (`pop_front()`) tem complexidade $O(1)$
- O primeiro passo da remoção é armazenar o membro head em uma variável temporária
- Em seguida, head deve apontar para o próximo elemento da lista
- O membro `prev` de head deve ser apontar para o último elemento (`tail`)
- O membro `next` de `tail` deve ser apontar para o primeiro elemento (head)
- Por fim, o ponteiro armazenado na variável temporária é deletado
- O membro `size` deve ser decrementado, se existir

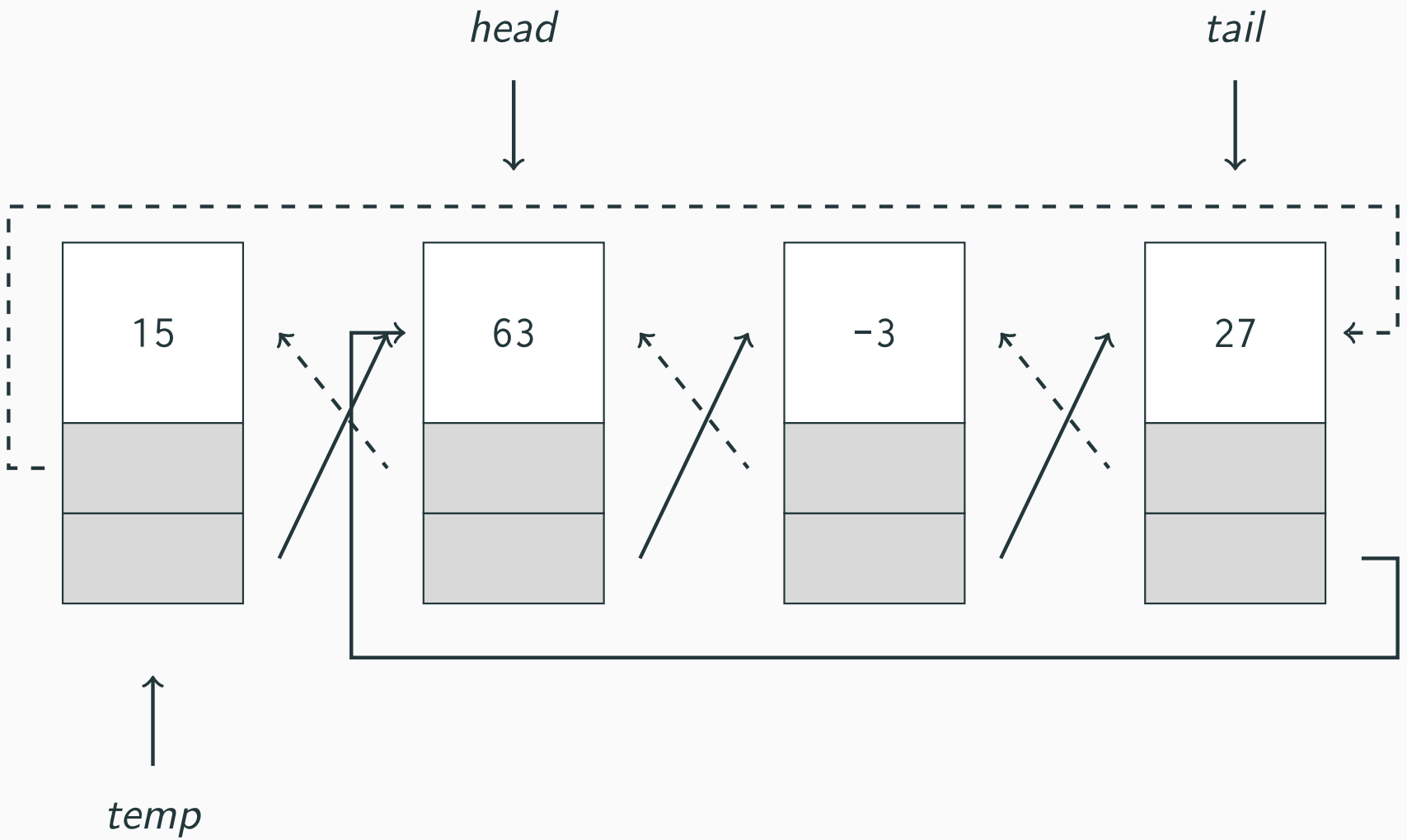
Visualização da remoção do início da lista



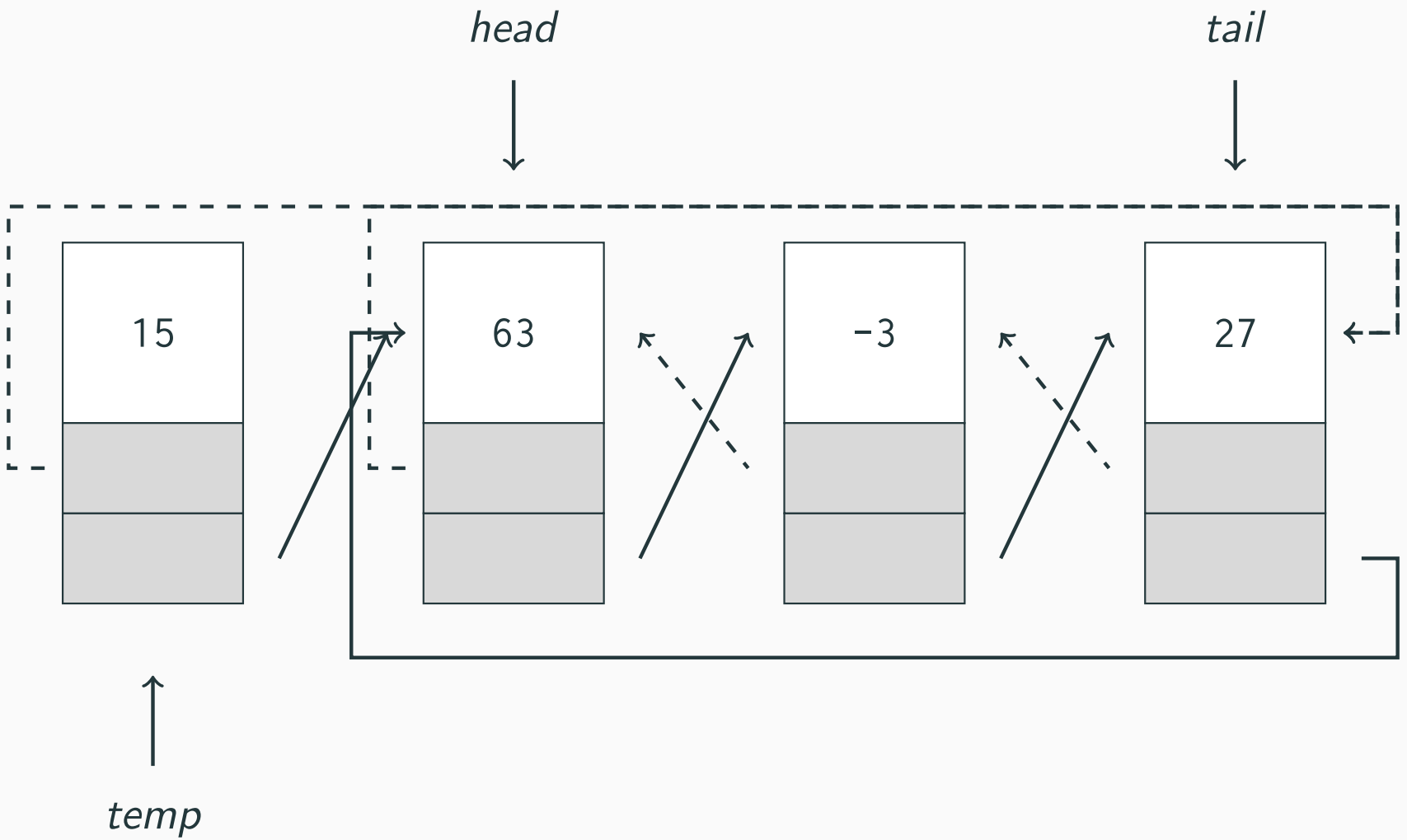
Visualização da remoção do início da lista



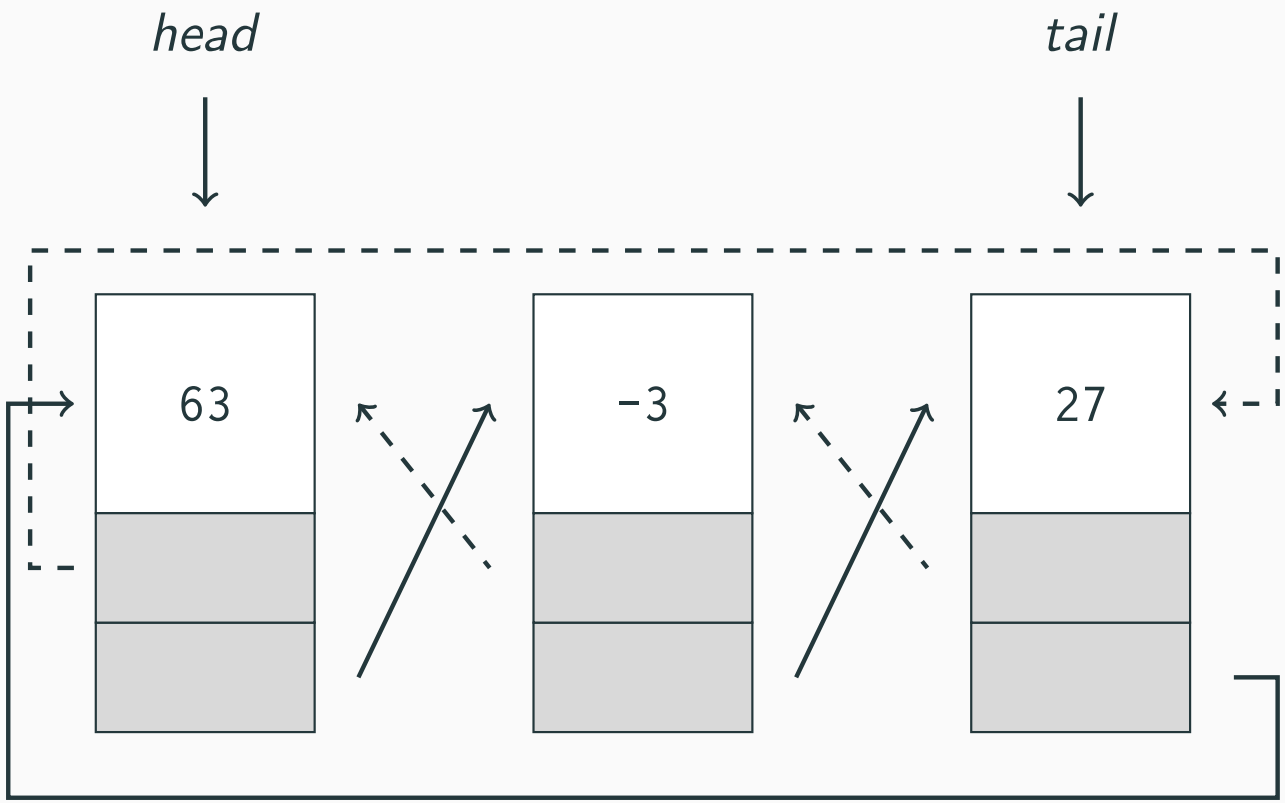
Visualização da remoção do início da lista



Visualização da remoção do início da lista



Visualização da remoção do início da lista



Implementação da remoção do início

```
55      void pop_front()
56      {
57          if (!head)
58              throw "Lista vazia";
59
60          auto temp = head;
61          head = _size > 1 ? head->next : nullptr;
62          delete temp;
63
64          head ? head->prev = tail, tail->next = head : tail = nullptr;
65          _size--;
66      }
```

Remoção do final da lista

- A remoção do último elemento de uma lista circular (pop_back()) tem complexidade $O(1)$
- Efetivamente, ela pode ser reduzida à uma remoção do início: basta delocar tanto head quanto `tail` uma posição para trás
- Esta simplificação evita a duplicidade de código, e mantém os invariantes da lista

Implementação da remoção do final

```
68      void pop_back()
69      {
70          if (!tail)
71              throw "Lista vazia";
72
73          tail = tail->prev;
74          head = head->prev;
75
76          pop_front();
77      }
78  };
79
80  #endif
```

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
- Novamente é possível reduzir este problema para o da remoção do início: basta avançar head e `tail` simultaneamente até que head se iguale com o ponteiro do elemento a ser removido
- Assim como nas demais remoções, a tentativa de excluir um elemento de uma lista vazia constitui um erro

Exemplo de uso da lista encadeada

```
1  #include <iostream>
2  #include "circular_list.h"
3
4  using namespace std;
5
6  int main()
7  {
8      CircularList<int> clist { 1, 2, 3, 4, 5 };
9
10     cout << clist << '\n';
11
12     clist.push_back(6);
13     clist.push_front(7);
14
15     cout << clist << '\n';
16
17     auto it = clist.begin();
18
19     for (int i = 0; i < 20; ++i)
20         cout << *it++ << ' ';
```

Exemplo de uso da lista encadeada

```
21      cout << '\n';
22
23      int counter = 0;
24
25      while (not clist.empty())
26      {
27          counter++ % 2 ? clist.pop_front() : clist.pop_back();
28          cout << clist << '\n';
29      }
30
31      cout << "Empty? " << clist.empty() << '\n';
32
33      clist.push_back(10);
34      clist.push_front(20);
35      clist.push_back(30);
36
37      cout << clist << '\n';
38
39      return 0;
40  }
```

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/