Lista Simplesmente Encadeada

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz

gomes.atilio@ufc.br

Universidade Federal do Ceará

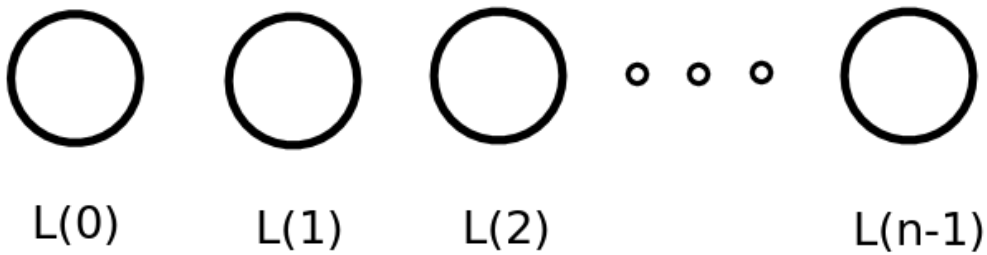
Campus Quixadá

2º semestre/2022

Introdução

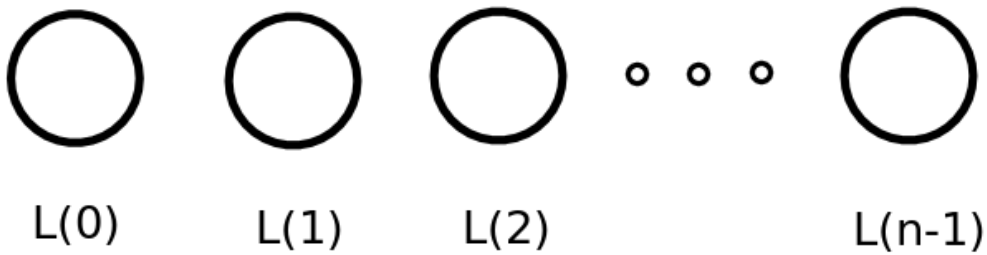
Estrutura de dados: Lista linear

- Uma lista linear L é um conjunto de $n \geq 0$ elementos (nós) $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:

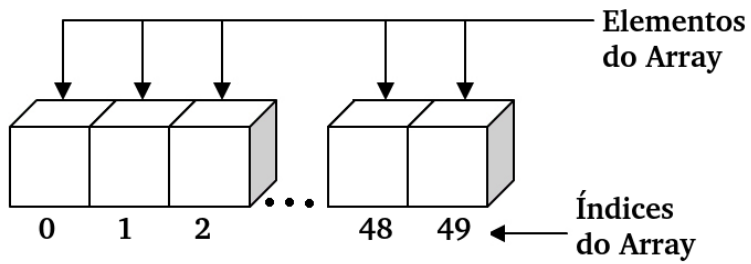


Estrutura de dados: Lista linear

- Uma lista linear L é um conjunto de $n \geq 0$ elementos (nós) $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:



- Vimos que uma lista linear pode ser implementada por meio de um array usando alocação dinâmica de memória (alocação sequencial).



Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Desvantagens do uso de vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Desvantagens do uso de vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- têm um tamanho fixo
  - ou alocamos um vetor pequeno e o espaço pode acabar
  - ou alocamos um vetor grande e desperdiçamos memória
  - Solução: criar lista sequencial redimensionável

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Desvantagens do uso de vetores:

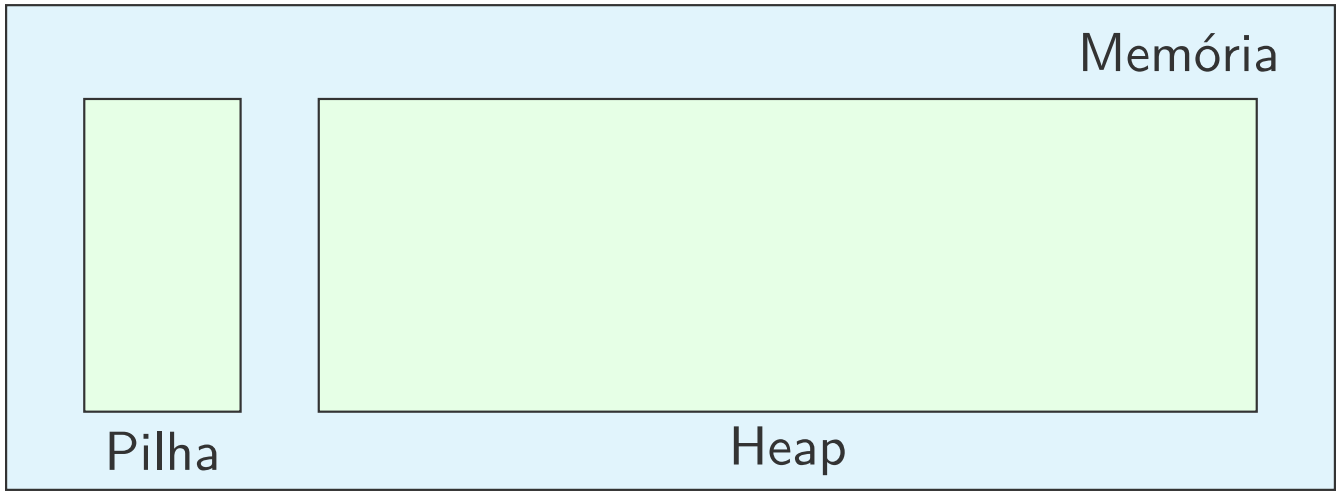
- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- têm um tamanho fixo
  - ou alocamos um vetor pequeno e o espaço pode acabar
  - ou alocamos um vetor grande e desperdiçamos memória
  - Solução: criar lista sequencial redimensionável
- operações de inserção e remoção de elementos são custosas: $O(n)$

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

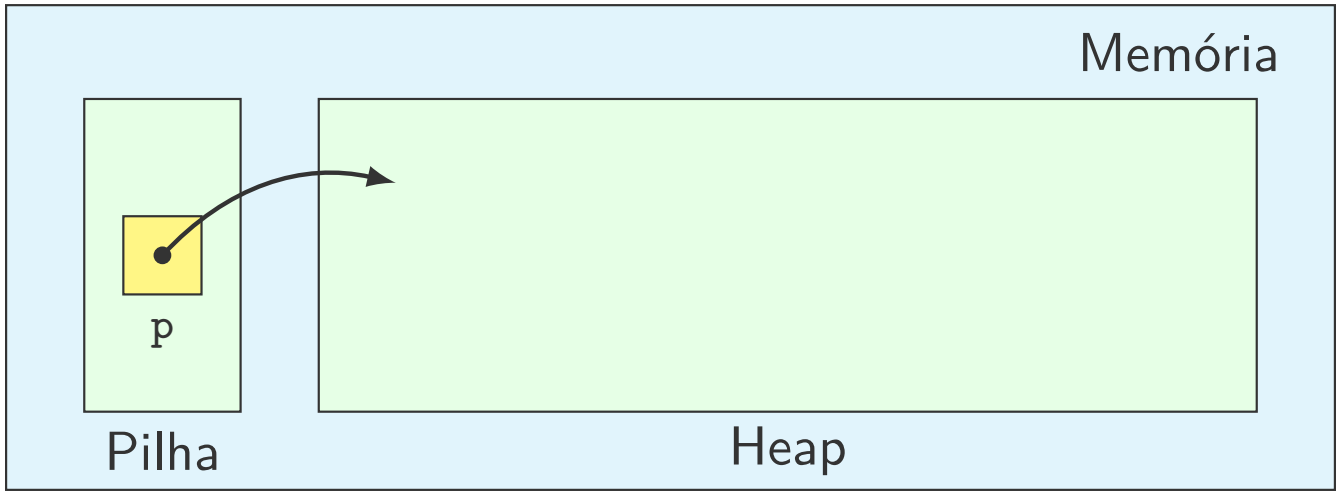
Listas Simplesmente Encadeadas

Alternativa - Lista Simplesmente Encadeada

Alternativa - Lista Simplesmente Encadeada



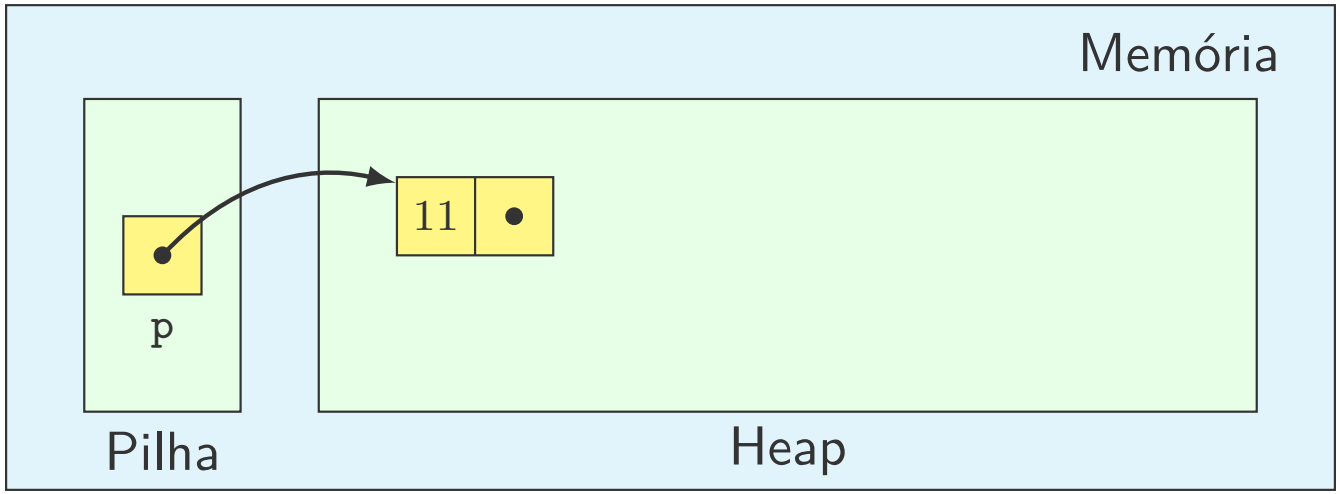
Alternativa - Lista Simplesmente Encadeada



- declaramos um ponteiro para a lista no nosso programa

Alternativa - Lista Simplesmente Encadeada

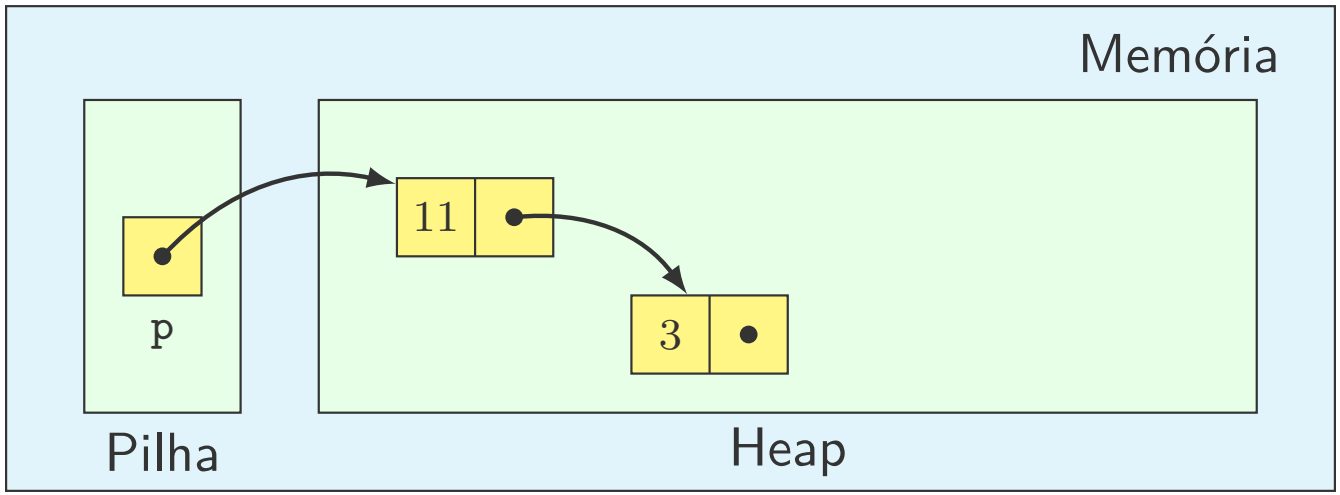
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário

Alternativa - Lista Simplesmente Encadeada

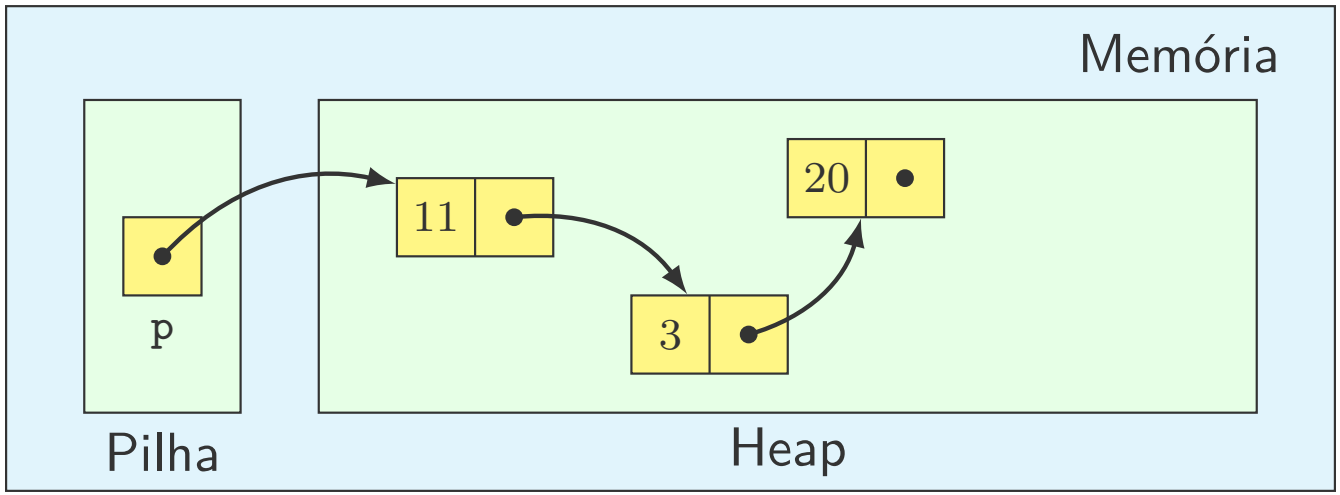
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro nó aponta para o segundo

Alternativa - Lista Simplesmente Encadeada

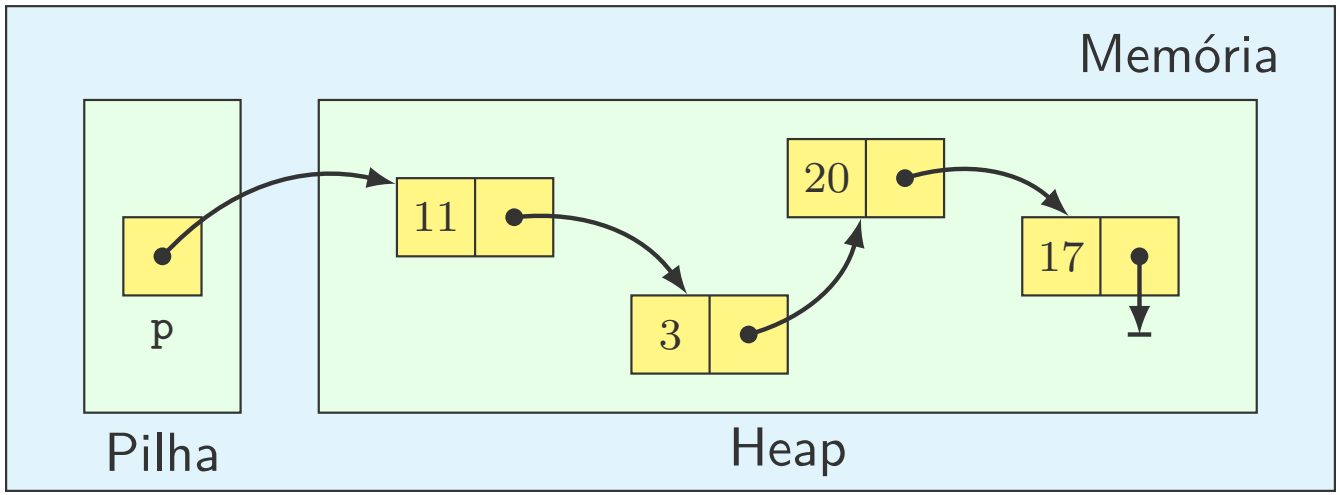
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro

Alternativa - Lista Simplesmente Encadeada

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro
- o último nó aponta para `nullptr`

Lista Simplesmente Encadeada

- O TAD Lista Linear pode ser implementado usando **alocação encadeada** como uma lista simplesmente encadeada.

Lista Simplesmente Encadeada

- O TAD Lista Linear pode ser implementado usando **alocação encadeada** como uma lista simplesmente encadeada.

- A Lista Simplesmente Encadeada mantêm dois atributos:
  - um ponteiro para o primeiro nó (`head`).
  - o número de elementos atualmente na lista (`size`).

Lista Simplesmente Encadeada

- O TAD Lista Linear pode ser implementado usando **alocação encadeada** como uma lista simplesmente encadeada.

- A Lista Simplesmente Encadeada mantêm dois atributos:
  - um ponteiro para o primeiro nó (`head`).
  - o número de elementos atualmente na lista (`size`).

- Operações que podemos querer realizar numa lista:
  - Criar uma nova lista vazia.
  - Deixar a lista vazia.
  - Destruir a lista.
  - Adicionar um elemento em qualquer posição da lista.
  - Remover da lista um elemento em certa posição.
  - Acessar um elemento em uma dada posição.
  - Buscar um elemento.
  - Consultar o tamanho atual da lista.
  - Saber se lista está vazia.
  - Imprimir a lista

Detalhes de Implementação

Listas Encadeadas – Detalhes de Implementação

É formada por um conjunto de objetos chamados nós.

Nó é um elemento alocado dinamicamente que contém:

- o dado armazenado
- um ponteiro para o nó seguinte na lista

Listas Encadeadas – Detalhes de Implementação

É formada por um conjunto de objetos chamados nós.

Nó é um elemento alocado dinamicamente que contém:

- o dado armazenado
- um ponteiro para o nó seguinte na lista

- Um nó pode ser implementado como um `struct` ou como uma `class`.

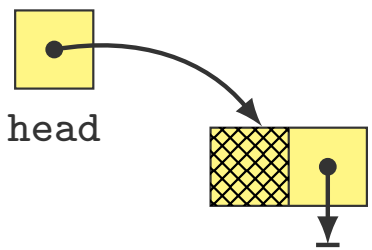
Arquivo Node.h


```
1  #ifndef NODE_H
2  #define NODE_H
3
4  using Item = int;
5
6  class Node {
7      friend class ForwardList;
8  private:
9      Item value;  // valor
10     Node* next;  // ponteiro para o proximo no
11 public:
12     Node(const Item& val, Node *nextPtr) {
13         value = val;
14         next = nextPtr;
15     }
16 };
17
18 #endif
```

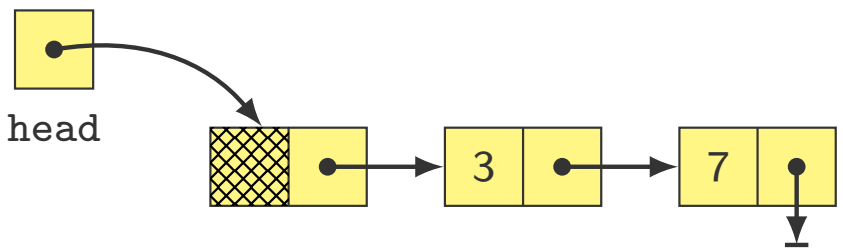
Listas Encadeadas – Detalhes da Implementação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Conjunto de nós ligados entre si de maneira sequencial.
- O ponteiro `head` **sempre** aponta para o nó sentinela.
- Quando a lista está vazia, o nó sentinela é o único nó na lista e seu campo `next` aponta para `nullptr`.



Lista vazia



Lista com 2 elementos

Arquivo ForwardList.h


```
1  #ifndef FORWARDLIST_H
2  #define FORWARDLIST_H
3  #include "Node.h"
4
5  class ForwardList {
6  private:
7      Node* m_head; // aponta para o inicio da lista
8      int m_size;   // numero de elementos na lista
9
10 public:
11     // construtor: cria lista vazia
12     ForwardList();
13
14     // construtor de copia
15     ForwardList(const ForwardList& lst);
16
17     // retorna true sse a lista esta vazia
18     bool empty() const;
19
20     // retorna o numero de elementos na lista
21     int size() const;
```

Arquivo ForwardList.h (const.)


```
22      // deixa a lista vazia: size() == 0
23      void clear();
24
25      // destrutor: libera memoria alocada
26      ~ForwardList();
27
28      // operador[] para acesso a elemento
29      Item& operator[](int index);
30      const Item& operator[](int index) const;
31
32      // insere um elemento no indice especificado
33      void insert_at(int index, const Item& val);
34
35      // remove o elemento no indice especificado
36      void remove_at(int index);
37  };
38
39  #endif
```

Arquivo main.cpp


```
1  #include <iostream>
2  #include "ForwardList.h"
3  using namespace std;
4
5  void print_list(const ForwardList& lst) {
6      for(int i = 0; i < lst.size(); i++) {
7          cout << lst[i] << " ";
8      }
9      cout << endl;
10 }
11
12 int main() {
13     ForwardList lista; // cria lista vazia
14
15     for(int i = 1; i <= 10; i++)
16         lista.insert_at(lista.size(), i);
17
18     print_list(lista); // imprime valores
19 }
```

Exercício

- Implementar as funções-membro da classe ForwardList.

Listas Sequenciais × Encadeadas

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Qual é melhor?

- depende do problema, do algoritmo e da implementação

Exercício 1

Funções Adicionais

Exercício: Implemente as seguintes funções adicionais na ForwardList.

- `ForwardList(int v[], int n)`
  Construtor que recebe um array `v` com n inteiros e inicializa a lista com os n elementos do array `v`.

- `ForwardList(const ForwardList& lst)`
  Construtor de cópia, que recebe uma referência para uma ForwardList `list` e inicializa a nova lista com os elementos de `list`.

- `const ForwardList& operator=(const ForwardList& lst)`
  Implemente uma versão sobrecarregada do operador de atribuição para a ForwardList. O operador de atribuição permite atribuir uma lista a outra.
  Exemplo: `list2 = list1;`
  Após esta atribuição, `list2` e `list1` são duas listas distintas que possuem **o mesmo** conteúdo.

Funções Adicionais

- `bool equals(const ForwardList& lst);`
  Determina se a lista `lst`, passada por parâmetro, é igual a lista em questão. Duas listas são iguais se têm o mesmo tamanho e o valor do k-ésimo elemento da primeira lista é igual ao k-ésimo valor da segunda.

- `void concat(const ForwardList& lst);`
  Concatena a lista atual com a lista `lst`. A lista `lst` não é modificada nessa operação.

- `void reverse();`: Inverte a ordem dos nós (o primeiro nó passa a ser o último, o segundo passa a ser o penúltimo, etc.) Essa operação faz isso sem criar novos nós, apenas altera os ponteiros. Dica: tente usar três ponteiros pra fazer as trocas.

Funções Adicionais

- `void swap(ForwardList& lst);`
  Troca o conteúdo dessa lista pelo conteúdo de lst. Após a chamada para esta função, os elementos nesta lista são aqueles que estavam em `lst` antes da chamada, e os elementos de `lst` são aqueles que estavam nesta lista.

- `void remove(const Item& val);`
  Remove da lista todos os elementos com valor igual a `val`.

- `Item& back();`
  `const Item& back() const;`
  Retorna uma referencia para o ultimo elemento na lista

- `void push_back(const Item& val);`
  Insere um elemento no final da lista.

- `void pop_back();`
  Deleta o ultimo elemento da lista

Funções Adicionais

- `Item& front();`
  `const Item& front() const;`
  Retorna uma referencia para o primeiro elemento na lista

- `void push_front(const Item& val);`
  Insere um elemento no inicio da lista.

- `void pop_front();`
  Deleta o primeiro elemento da lista.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM