Estruturas de Dados

Prof. Rodrigo Martins
rodrigo.martins@francomontoro.com.br

Cronograma da Aula

- Listas
  - Lista encadeada tradicional
  - Listas duplamente encadeada
  - Listas circular
  - Lista dinâmica (A classe List)
- Exemplos
- Exercícios

Listas

- São estruturas de dados lineares utilizadas para armazenar e organizar um conjunto de elementos.

- São chamadas de estruturas de dados lineares porque os elementos são organizados em uma sequência linear, ou seja, cada elemento possui uma posição fixa em relação aos outros elementos.

Listas

- Uma lista é composta por nós, onde cada nó contém um elemento e uma referência para o próximo nó da lista.
- O primeiro nó da lista é chamado de "cabeça" (head) e o último nó é chamado de "cauda" (tail).
- A cauda da lista tem uma referência nula, indicando o fim da lista.

Listas

- A principal característica das listas é que elas permitem a inserção e remoção de elementos de forma flexível, sem a necessidade de realocação de memória.

- Isso ocorre porque os nós são encadeados, ou seja, cada nó contém uma referência para o próximo nó, permitindo a manipulação dos elementos de forma eficiente.

Listas

- As listas podem ter diferentes tipos de encadeamento, como encadeamento simples (cada nó tem uma referência para o próximo nó), encadeamento duplo (cada nó tem referências para o próximo e o nó anterior) e encadeamento circular (o último nó tem uma referência para o primeiro nó).

Algumas das principais vantagens e aplicações das listas

- **Flexibilidade na inserção e remoção de elementos:**

  – Uma das principais vantagens das listas é a capacidade de inserir e remover elementos de forma flexível. Diferentemente de outras estruturas de dados, como arrays, as listas não possuem tamanho fixo e não requerem realocação de memória ao adicionar ou remover elementos.

  – As listas são ideais para situações em que a quantidade de elementos pode variar dinamicamente.

Algumas das principais vantagens e aplicações das listas

- **Eficiência na manipulação de elementos:**

  – Devido à estrutura encadeada das listas, a inserção e remoção de elementos podem ser realizadas de forma eficiente. Inserir ou remover um elemento no início ou no final da lista tem complexidade de tempo constante, desde que as referências estejam corretamente atualizadas.

  – No entanto, a busca de um elemento em uma lista pode ser menos eficiente, exigindo uma busca sequencial até encontrar o elemento desejado.

Algumas das principais vantagens e aplicações das listas

- **Implementação de estruturas de dados avançadas:**
  - As listas são utilizadas como base para implementar outras estruturas de dados avançadas, como pilhas, filas e listas duplamente encadeadas.
  - Essas estruturas podem ser facilmente implementadas usando listas como o principal mecanismo de armazenamento e manipulação dos elementos.

Algumas das principais vantagens e aplicações das listas

- **Manipulação de grandes volumes de dados:**

  - As listas são eficientes na manipulação de grandes volumes de dados, pois permitem a adição e remoção de elementos sem a necessidade de realocação de memória.

  - São úteis quando a quantidade de elementos pode variar ao longo do tempo ou quando o tamanho total dos dados não é conhecido antecipadamente.

Lista encadeada tradicional

- Nas estruturas de dados lineares, como as listas, os elementos são armazenados em nós.

- Um nó é uma unidade básica de armazenamento que contém o elemento e uma referência para o próximo nó na lista.

Lista encadeada tradicional

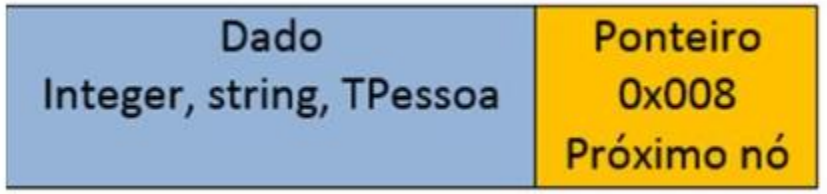
- Cada nó contém dois componentes principais:
  - **Elemento**: É a informação que está sendo armazenada na lista. Pode ser qualquer tipo de dado, como um número, uma string, um objeto personalizado, entre outros.
  - **Referência para o próximo nó**: É um ponteiro ou uma referência que aponta para o próximo nó na lista. Essa referência é essencial para manter a ordem e a conexão entre os nós.

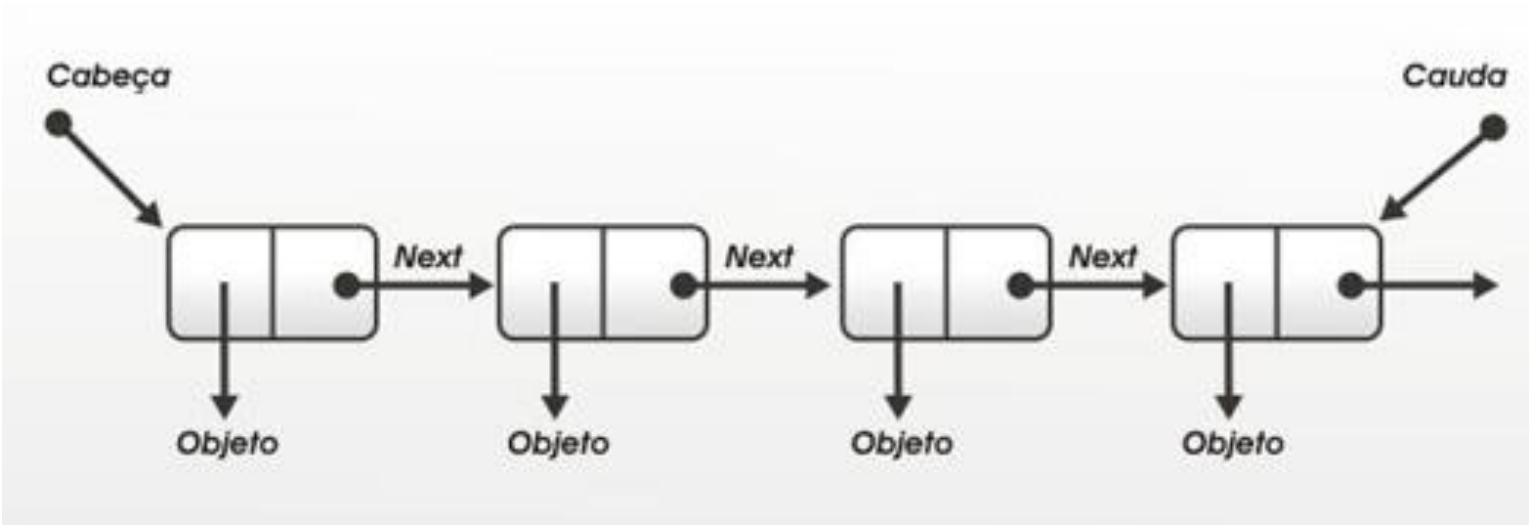
Lista encadeada tradicional

- A ligação entre os nós é o que permite percorrer a lista de forma sequencial, acessando cada elemento. Cada nó, exceto o último nó da lista, possui uma referência para o próximo nó. Essa referência permite navegar da cabeça (primeiro nó) até a cauda (último nó) da lista.

- Quando um novo elemento é inserido na lista, um novo nó é criado e a referência do nó anterior é ajustada para apontar para o novo nó. Dessa forma, a ligação entre os nós é atualizada para incluir o novo elemento na sequência correta.

Lista encadeada tradicional





Exemplo 1 – lista encadeada tradicional

lista-encadeada.cpp

```cpp
#include <iostream>

using namespace std;

// Definição da estrutura do nó da lista encadeada
struct Node {
    int dado;             // Dado do nó
    Node* proximo;        // Ponteiro para o próximo nó
};

// Função para criar um novo nó com o dado fornecido
Node* criarNo(int dado) {
    Node* novoNo = new Node;
    novoNo->dado = dado;
    novoNo->proximo = nullptr;
    return novoNo;
}

```

Exemplo 1 – lista encadeada tradicional

```
19  // Função para inserir um novo nó no início da lista
20  void inserirNoInicio(Node** cabeca, int dado) {
21      // Cria um novo nó
22      Node* novoNo = criarNo(dado);
23
24      // Define o próximo do novo nó como o nó atual no início da lista
25      novoNo->proximo = *cabeca;
26
27      // Atualiza o ponteiro para o novo nó como o novo nó no início da lista
28      *cabeca = novoNo;
29  }
30
31  // Função para remover um nó com o dado fornecido da lista
32  void removerNo(Node** cabeca, int dado) {
33      // Verifica se a lista está vazia
34      if (*cabeca == nullptr) {
35          cout << "Lista vazia. Nenhum no removido." << endl;
36          return;
37      }
38
```

Exemplo 1 – lista encadeada tradicional

```
39      // Verifica se o primeiro nó contém o dado a ser removido
40      if ((*cabeca)->dado == dado) {
41          Node* temp = *cabeca;
42          *cabeca = (*cabeca)->proximo;
43          delete temp;
44          cout << "No removido." << endl;
45          return;
46      }
47
48      // Procura o nó a ser removido na lista
49      Node* anterior = *cabeca;
50      Node* atual = (*cabeca)->proximo;
51      while (atual != nullptr && atual->dado != dado) {
52          anterior = atual;
53          atual = atual->proximo;
54      }
55
```

Exemplo 1 – lista encadeada tradicional

```
56          // Se o nó foi encontrado, remove-o da lista
57          if (atual != nullptr) {
58              anterior->proximo = atual->proximo;
59              delete atual;
60              cout << "No removido." << endl;
61          } else {
62              cout << "No nao encontrado na lista." << endl;
63          }
64      }
65
66      // Função para imprimir os elementos da lista
67      void imprimirLista(Node* cabeca) {
68          cout << "Elementos da lista: ";
69          while (cabeca != nullptr) {
70              cout << cabeca->dado << " ";
71              cabeca = cabeca->proximo;
72          }
73          cout << endl;
74      }
75
```

Exemplo 1 – lista encadeada tradicional

```
76  // Função principal
77  int main() {
78      Node* cabeca = nullptr;      // Ponteiro para o primeiro nó da lista
79
80      // Inserção de elementos na lista
81      inserirNoInicio(&cabeca, 10);
82      inserirNoInicio(&cabeca, 20);
83      inserirNoInicio(&cabeca, 30);
84
85      // Imprime os elementos da lista
86      imprimirLista(cabeca);
87
88      // Remove um nó da lista
89      removerNo(&cabeca, 20);
90
91      // Imprime os elementos atualizados da lista
92      imprimirLista(cabeca);
93
94      return 0;
95  }
```

Listas duplamente encadeadas

- Nas estruturas de dados, um nó é um elemento básico que compõe uma estrutura de dados.

- Ele contém os dados a serem armazenados e uma ou mais ligações para outros nós na estrutura de dados.

- Em algumas estruturas de dados, como as listas duplamente encadeadas, cada nó possui ligações duplas, ou seja, ligações tanto para o nó anterior quanto para o próximo nó na sequência.

Listas duplamente encadeadas

- As ligações duplas permitem que a estrutura de dados seja percorrida em ambas as direções, facilitando operações como a inserção e a remoção de elementos em diferentes posições da lista.

- Com ligações duplas entre os nós, é possível percorrer a lista em ambas as direções.

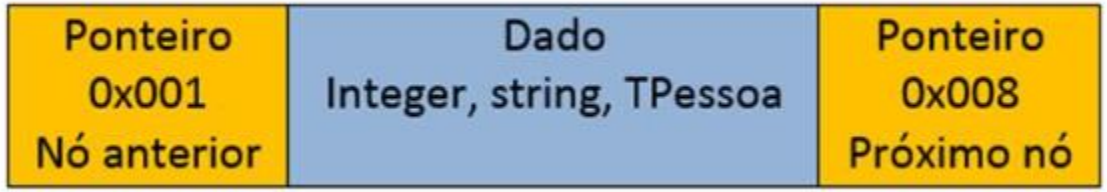
- Por exemplo, dado um nó atual, podemos acessar o nó anterior usando "prev" e o próximo nó usando "next".

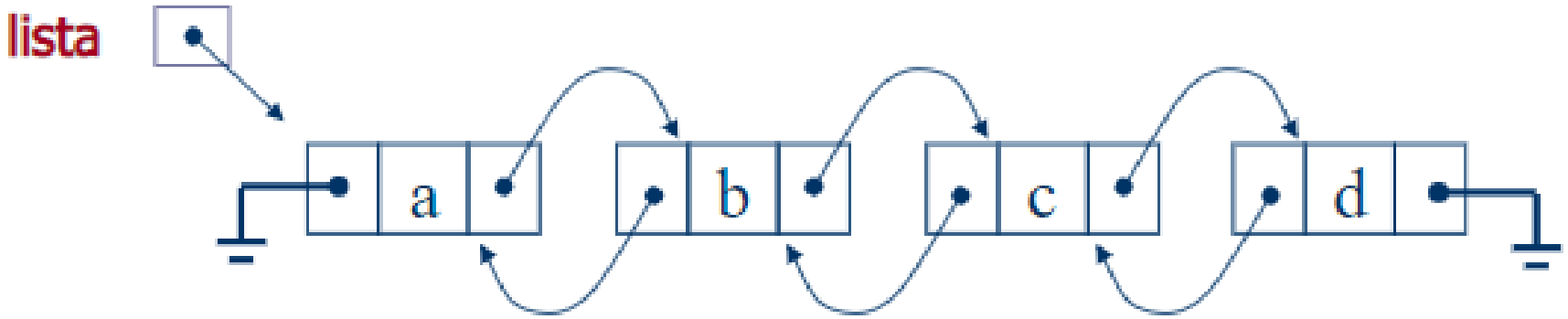
Listas duplamente encadeadas

- A utilização de ligações duplas entre os nós em estruturas de dados como as listas duplamente encadeadas oferece maior flexibilidade e eficiência em operações que envolvem a manipulação e a busca de elementos na estrutura.

- No entanto, também requer um pouco mais de complexidade em relação às listas encadeadas simples, já que é necessário atualizar corretamente os ponteiros ao realizar inserções ou remoções de nós.

Listas duplamente encadeadas





Exemplo 2 - Listas duplamente encadeadas

*lista-dupla-encadeada.cpp

```cpp
#include <iostream>

using namespace std;

// Definição da estrutura do nó da lista duplamente encadeada
struct Node {
    int dado;       // Dado do nó
    Node* anterior; // Ponteiro para o nó anterior
    Node* proximo;  // Ponteiro para o próximo nó
};

// Função para criar um novo nó com o dado fornecido
Node* criarNo(int dado) {
    Node* novoNo = new Node;
    novoNo->dado = dado;
    novoNo->anterior = nullptr;
    novoNo->proximo = nullptr;
    return novoNo;
}

```

Exemplo 2 - Listas duplamente encadeadas

```
21  // Função para inserir um novo nó no início da lista
22  void inserirNoInicio(Node** cabeca, int dado) {
23      // Cria um novo nó
24      Node* novoNo = criarNo(dado);
25
26      // Atualiza os ponteiros do novo nó e do nó atual no início da lista
27      novoNo->proximo = *cabeca;
28      if (*cabeca != nullptr) {
29          (*cabeca)->anterior = novoNo;
30      }
31
32      // Atualiza o ponteiro para o novo nó como o novo nó no início da lista
33      *cabeca = novoNo;
34  }
35
36  // Função para remover um nó com o dado fornecido da lista
37  void removerNo(Node** cabeca, int dado) {
38      // Verifica se a lista está vazia
39      if (*cabeca == nullptr) {
40          cout << "Lista vazia. Nenhum no removido." << endl;
41          return;
42      }
43
```

Exemplo 2 - Listas duplamente encadeadas

```
44      // Verifica se o primeiro nó contém o dado a ser removido
45      if ((*cabeca)->dado == dado) {
46          Node* temp = *cabeca;
47          *cabeca = (*cabeca)->proximo;
48          if (*cabeca != nullptr) {
49              (*cabeca)->anterior = nullptr;
50          }
51          delete temp;
52          cout << "No removido." << endl;
53          return;
54      }
55
56      // Procura o nó a ser removido na lista
57      Node* atual = *cabeca;
58      while (atual != nullptr && atual->dado != dado) {
59          atual = atual->proximo;
60      }
61
```

Exemplo 2 - Listas duplamente encadeadas

```
62      // Se o nó foi encontrado, remove-o da lista
63      if (atual != nullptr) {
64          if (atual->anterior != nullptr) {
65              atual->anterior->proximo = atual->proximo;
66          }
67          if (atual->proximo != nullptr) {
68              atual->proximo->anterior = atual->anterior;
69          }
70          delete atual;
71          cout << "No removido." << endl;
72      } else {
73          cout << "No nao encontrado na lista." << endl;
74      }
75  }
76
77  // Função para imprimir os elementos da lista
78  void imprimirLista(Node* cabeca) {
79      cout << "Elementos da lista: ";
80      while (cabeca != nullptr) {
81          cout << cabeca->dado << " ";
82          cabeca = cabeca->proximo;
83      }
84      cout << endl;
85  }
```

Exemplo 2 - Listas duplamente encadeadas

```
86
87  // Função principal
88  int main() {
89      Node* cabeca = nullptr; // Ponteiro para o início da lista
90
91      // Inserindo alguns nós no início da lista
92      inserirNoInicio(&cabeca, 3);
93      inserirNoInicio(&cabeca, 5);
94      inserirNoInicio(&cabeca, 7);
95
96      // Imprimindo os elementos da lista
97      imprimirLista(cabeca);
98
99      // Removendo um nó da lista
100     removerNo(&cabeca, 5);
101
102     // Imprimindo os elementos da lista após a remoção
103     imprimirLista(cabeca);
104
105     return 0;
106 }
```

Listas circular

- É uma estrutura de dados linear em que os nós estão conectados em forma de círculo, onde o último nó aponta para o primeiro nó.

- Diferentemente de uma lista encadeada tradicional, em uma lista circular não há um nó de sentinela ou um ponto de parada explícito, pois a lista continua indefinidamente.

- Em uma lista circular, cada nó possui um ponteiro para o próximo nó da lista. O último nó da lista aponta para o primeiro nó, fechando assim o círculo.

- Essa estrutura permite a iteração contínua pelos elementos da lista, sem a necessidade de voltar ao início manualmente.

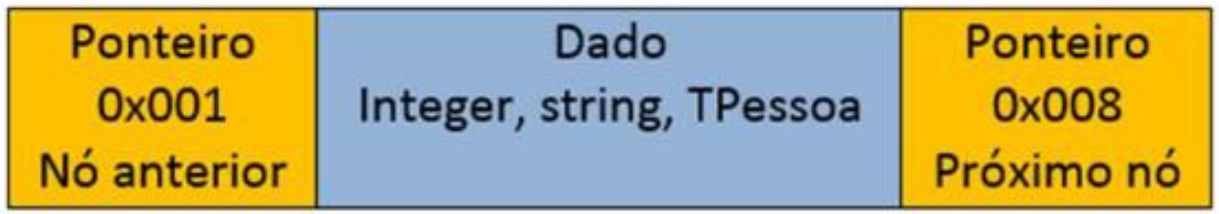
Listas circular

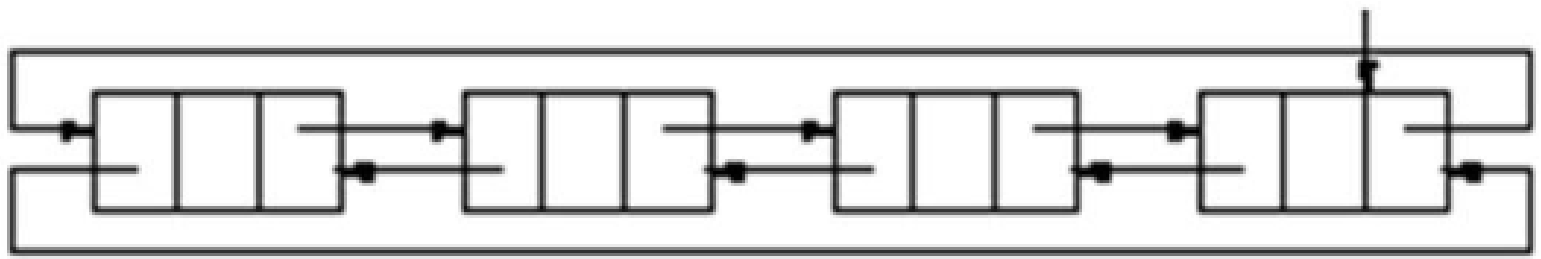
- A principal vantagem de uma lista circular é a facilidade de percorrer todos os elementos da lista a partir de qualquer ponto, sem a necessidade de percorrer toda a lista para chegar ao início novamente.

- Além disso, uma lista circular pode ser útil em situações em que é necessário implementar estruturas de dados cíclicas, como um buffer circular em um sistema de comunicação.

- No entanto, é importante tomar cuidado ao trabalhar com listas circulares para evitar loops infinitos durante a iteração pelos elementos da lista. É necessário garantir que o código de iteração saiba quando parar, considerando a condição de parada apropriada.

Listas circular





Exemplo 3- Lista circular

*lista-circular.cpp

```cpp
#include <iostream>

using namespace std;

// Definição da estrutura do nó da lista circular
struct Node {
    int dado;       // Dado do nó
    Node* proximo; // Ponteiro para o próximo nó
};

// Função para criar um novo nó com o dado fornecido
Node* criarNo(int dado) {
    Node* novoNo = new Node;
    novoNo->dado = dado;
    novoNo->proximo = nullptr;
    return novoNo;
}

```

Exemplo 3- Lista circular

```
19  // Função para inserir um novo nó no início da lista circular
20  void inserirNoInicio(Node** cabeca, int dado) {
21      // Cria um novo nó
22      Node* novoNo = criarNo(dado);
23
24      if (*cabeca == nullptr) {
25          // Se a lista estiver vazia, o novo nó será o início e o fim da lista
26          *cabeca = novoNo;
27          novoNo->proximo = novoNo; // Aponta o próximo nó para ele mesmo
28      } else {
29          // Se a lista não estiver vazia, insere o novo nó no início e atualiza os ponteiros
30          novoNo->proximo = *cabeca;
31          Node* atual = *cabeca;
32          while (atual->proximo != *cabeca) {
33              atual = atual->proximo;
34          }
35          atual->proximo = novoNo;
36          *cabeca = novoNo;
37      }
38  }
39
```

Exemplo 3- Lista circular

```
40  // Função para remover o nó do início da lista circular
41  void removerNoInicio(Node** cabeca) {
42      if (*cabeca == nullptr) {
43          // Verifica se a lista está vazia
44          cout << "Lista vazia. Nenhum no removido." << endl;
45      } else if ((*cabeca)->proximo == *cabeca) {
46          // Verifica se há apenas um nó na lista
47          delete *cabeca;
48          *cabeca = nullptr;
49          cout << "No removido. Lista vazia." << endl;
50      } else {
51          // Remove o nó do início e atualiza os ponteiros
52          Node* atual = *cabeca;
53          while (atual->proximo != *cabeca) {
54              atual = atual->proximo;
55          }
56          atual->proximo = (*cabeca)->proximo;
57          Node* temp = *cabeca;
58          *cabeca = (*cabeca)->proximo;
59          delete temp;
60          cout << "No removido." << endl;
61      }
62  }
63
```

Exemplo 3- Lista circular

```
64  // Função para imprimir os elementos da lista circular
65  void imprimirLista(Node* cabeca) {
66      if (cabeca == nullptr) {
67          cout << "Lista vazia." << endl;
68          return;
69      }
70
71      cout << "Elementos da lista: ";
72      Node* atual = cabeca;
73      do {
74          cout << atual->dado << " ";
75          atual = atual->proximo;
76      } while (atual != cabeca);
77      cout << endl;
78  }
79
```

Exemplo 3- Lista circular

```
80  // Função principal
81  int main() {
82      Node* cabeca = nullptr; // Ponteiro para o início da lista circular
83
84      // Inserindo alguns nós no início da lista circular
85      inserirNoInicio(&cabeca, 3);
86      inserirNoInicio(&cabeca, 5);
87      inserirNoInicio(&cabeca, 7);
88
89      // Imprimindo os elementos da lista circular
90      imprimirLista(cabeca);
91
92      // Removendo um nó do início da lista circular
93      removerNoInicio(&cabeca);
94
95      // Imprimindo os elementos da lista circular após a remoção
96      imprimirLista(cabeca);
97
98      return 0;
99  }
```

A classe List

- A classe list em C++ é parte da biblioteca padrão do C++ e é usada para implementar uma lista duplamente encadeada.

- Ela fornece uma série de funcionalidades para manipulação de listas, como inserção, remoção e acesso aos elementos.

- A classe list está definida no cabeçalho <list> e é implementada como uma estrutura de dados encadeada em que cada elemento (nó) possui um ponteiro para o elemento anterior e outro ponteiro para o próximo elemento da lista.

- Isso permite inserções e remoções eficientes em qualquer posição da lista.

A classe List

- Para caminhar entre os elementos da lista, usa-se iteradores, da classe iterator.

```
#include <list>
#include <iterator>
```

Métodos mais Importantes

- **push_back(valor):** Insere um elemento no fim da lista.
- **push_front(valor):** Insere um elemento no início da lista.
- **pop_back(valor):** Remove o último elemento da lista.
- **pop_front(valor):** Remove o primeiro elemento da lista.
- **insert(posição, valor):** Insere um elemento na posição especificada.
- **erase(posição):** Remove o elemento na posição especificada.
- **size():** Retorna o número de elementos na lista.
- **empty():** Verifica se a lista está vazia.
- **front():** Acessa o primeiro elemento da lista.

Métodos mais Importantes

- **back():** Acessa o último elemento da lista.
- **begin():** Retorna um iterador para o primeiro elemento da lista.
- **end():** Retorna um iterador para o próximo ao último elemento da lista.
- **rbegin():** Retorna um iterador reverso para o último elemento da lista.
- **rend():** Retorna um iterador reverso para o próximo ao primeiro elemento da lista.
- **clear():** Remove todos os elementos da lista.
- **sort():** Ordena os elementos da lista em ordem crescente.
- **reverse():** Inverte a ordem dos elementos na lista.

Exemplo 4 – Classe List

```
1  #include <iostream>
2  #include <list>
3  #include <iterator>
4
5  using namespace std;
6
7  void ImprimirLista(const list<int>& lista)
8  {
9      list<int>::const_iterator elemento; // [classe]::[método]
10     if (lista.empty())
11     {
12         cout << "Lista vazia\n";
13     }
14     else
15     {
16         elemento = lista.begin(); //aponta para o primeiro elemento.
17         do {
18             cout << *elemento << " ";
19             elemento++;
20         } while (elemento != lista.end());
21         cout << endl;
22     }
23 }
```

Exemplo 4 – Classe List

```
24
25  int main()
26  {
27
28      list<int> lista;
29      lista.push_back(1);  // [1]
30      lista.push_back(5);  // [1,5]
31      lista.push_front(3); // [3,1,5]
32      lista.push_front(2); // [2,3,1,5]
33      ImprimirLista(lista);
34      cout << "A lista tem " << lista.size() << " elementos" << endl << endl;
35      lista.sort(); // Ordena a lista
36      ImprimirLista(lista); // [1,2,3,5]
37      lista.remove(5); // Remove todas as ocorrencias de 5 na lista. -> [1,2,3]
38      ImprimirLista(lista); // [1,2,3]
39      lista.reverse(); // Inverte a ordem dos elementos -> [3,2,1]
40      ImprimirLista(lista);
41      lista.push_back(2); // [3,1,2,2]
42      ImprimirLista(lista);
43      lista.push_front(2); // [2,2,3,1,2]
44      ImprimirLista(lista);
45      lista.sort();
46      ImprimirLista(lista); //[1,2,2,2,3]
47  }
```

Exercício 1

- Você foi contratado para criar um programa em C++ que permite ao usuário digitar uma lista de números inteiros e, em seguida, exibir os números na ordem de inserção e em ordem inversa.

- **Instruções:**

- Utilize a classe list para armazenar uma lista de números inteiros.
- Solicite ao usuário que digite 5 números inteiros.
- Insira os números na lista, um por vez.
- Após a inserção dos números, exiba os números da lista na ordem em que foram inseridos, separados por espaços.
- Em seguida, exiba os números da lista em ordem inversa, separados por espaços.
- Compile e execute o programa para testá-lo.
- Verifique se a saída do programa está correta e se os números são exibidos corretamente na ordem de inserção e em ordem inversa.

Exercício 2

- Construa uma lista de funcionários em C++:
  - O programa deve pedir quantos funcionários serão cadastrados;
  - O programa deve avisar se lista vazia;
  - O programa deve imprimir esta lista na ordem de cadastro;
  - O programa deve imprimir a lista ordenada;
  - O programa deve imprimir a lista reversa;
  - O programa deve mostrar quantos funcionários estão cadastrados.

Referência desta aula

- http://www.cplusplus.com/reference/

Obrigado

Rodrigo