Univiçosa – Centro Universitário de Viçosa
Cursos: Análise e Desenvolvimento de Sistemas e Engenharia de Computação
Disciplina: ADS101/ECO131 – Programação de Computadores 2

Aula 10 – Introdução a lista

1) Introdução

Uma **Lista** é uma estrutura de dados que armazena uma sequência de elementos de maneira linear, mas com uma característica especial: os elementos não precisam estar armazenados de forma contígua na memória, como ocorre com vetores ou arrays.

2) Alguns tipos de listas

Existem dois tipos de listas que são mais comuns.

Lista Simplesmente Encadeada (Singly Linked List):

Nessa estrutura, cada elemento (também chamado de **nó**) contém dois componentes:

- **Dados**: que armazena o valor.
- **Ponteiro** para o próximo nó: que aponta para o próximo nó da lista.

O último nó da lista aponta para `nullptr` (ou `NULL`), indicando o final da lista. O ponto de entrada da lista é o **cabeçalho**, que contém o ponteiro para o primeiro nó da lista.

Observe no quadro abaixo um exemplo de lista encadeada usando Struct.

```
#include <iostream>
using namespace std;

// Definição do nó
struct Node {
    int data;       // Dados armazenados no nó
    Node* next;     // Ponteiro para o próximo nó
};

// Função para inserir um novo nó no início da lista
void insertAtBeginning(Node** head, int newData) {
    Node* newNode = new Node();  // Aloca um novo nó
    newNode->data = newData;     // Atribui o dado
```

```
    newNode->next = *head;       // Aponta para o antigo primeiro nó
    *head = newNode;             // Atualiza o ponteiro do cabeçalho
}

// Função para exibir a lista
void printList(Node* node) {
    while (node != nullptr) {
        cout << node->data << " -> ";
        node = node->next;
    }
    cout << "null" << endl;
}

int main() {
    Node* head = nullptr; // Inicializa a lista vazia

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 30);

    printList(head); // Saída: 30 -> 20 -> 10 -> null

    return 0;
}
```

Lista Duplamente Encadeada (Doubly Linked List):

Nessa estrutura, cada nó tem três componentes:

- **Dados**: o valor armazenado.
- **Ponteiro para o próximo nó**.
- **Ponteiro para o nó anterior**.

Isso permite que a navegação na lista seja feita tanto para frente quanto para trás. Da mesma forma, o primeiro nó tem o ponteiro anterior apontando para `nullptr`, e o último nó tem o ponteiro próximo apontando para `nullptr`.

Observe o exemplo no quadro abaixo.

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* prev;
    Node* next;
};
```

```
// Função para inserir um novo nó no início da lista duplamente encadeada
void insertAtBeginning(Node** head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = *head;
    newNode->prev = nullptr;

    if (*head != nullptr)
        (*head)->prev = newNode;

    *head = newNode;
}

// Função para imprimir a lista para frente
void printListForward(Node* node) {
    while (node != nullptr) {
        cout << node->data << " -> ";
        node = node->next;
    }
    cout << "null" << endl;
}

// Função para imprimir a lista para trás
void printListBackward(Node* node) {
    // Ir até o último nó
    Node* last = nullptr;
    while (node != nullptr) {
        last = node;
        node = node->next;
    }

    // Agora imprime para trás
    while (last != nullptr) {
        cout << last->data << " -> ";
        last = last->prev;
    }
    cout << "null" << endl;
}

int main() {
    Node* head = nullptr;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 30);

    cout << "Lista para frente: ";
    printListForward(head); // Saída: 30 -> 20 -> 10 -> null
```

```
    cout << "Lista para trás: ";
    printListBackward(head); // Saída: 10 -> 20 -> 30 -> null

    return 0;
}
```

Diferenças principais entre Lista Simples e Dupla:

- **Memória**: A lista duplamente encadeada usa mais memória, pois cada nó tem dois ponteiros (um para o próximo e outro para o anterior).
- **Acesso**: Na lista duplamente encadeada, é possível navegar nos dois sentidos, enquanto na lista simplesmente encadeada, você só pode percorrer a lista para frente.
- **Inserção e remoção**: A inserção e remoção em uma lista duplamente encadeada pode ser mais eficiente, pois não é necessário percorrer a lista para encontrar o nó anterior.

3) O uso de listas no mundo real

Vamos usar o conceito de lista ligada para representar uma fila de atendimento em uma clínica. Cada paciente que chega é inserido no final da lista, e o atendimento ocorre no início da lista, removendo o primeiro paciente. Isso simula a lógica de **primeiro a entrar, primeiro a sair (FIFO)**, que é comum em filas.

Observe o código abaixo.

```
#include <iostream>
#include <string>

using namespace std;

// Estrutura que representa um nó (paciente) na lista ligada
struct Paciente {
    string nome;
    int idade;
    Paciente* proximo;
};

// Função para adicionar um paciente no final da fila
void adicionarPaciente(Paciente** frente, Paciente** fim, string nome, int idade) {
    Paciente* novoPaciente = new Paciente();
    novoPaciente->nome = nome;
    novoPaciente->idade = idade;
    novoPaciente->proximo = nullptr;
```

```
    // Se a fila está vazia, o novo paciente será o primeiro da fila
    if (*fim == nullptr) {
        *frente = novoPaciente;
        *fim = novoPaciente;
    } else {
        (*fim)->proximo = novoPaciente;
        *fim = novoPaciente;
    }

    cout << "Paciente " << nome << " adicionado à fila." << endl;
}

// Função para atender o primeiro paciente da fila (remover do início)
void atenderPaciente(Paciente** frente) {
    if (*frente == nullptr) {
        cout << "A fila está vazia. Ninguém para atender." << endl;
        return;
    }

    Paciente* pacienteAtendido = *frente;
        cout << "Atendendo paciente: " << pacienteAtendido->nome << ", Idade: " <<
pacienteAtendido->idade << endl;

    *frente = pacienteAtendido->proximo;

    delete pacienteAtendido;  // Libera a memória do paciente atendido
}

// Função para exibir a fila de pacientes
void exibirFila(Paciente* frente) {
    if (frente == nullptr) {
        cout << "A fila está vazia." << endl;
        return;
    }

    cout << "Fila de Pacientes: " << endl;
    while (frente != nullptr) {
        cout << "Paciente: " << frente->nome << ", Idade: " << frente->idade << endl;
        frente = frente->proximo;
    }
}

int main() {
    Paciente* frente = nullptr;  // Ponteiro para o primeiro paciente (início da fila)
    Paciente* fim = nullptr;     // Ponteiro para o último paciente (fim da fila)

    // Adicionando pacientes à fila
    adicionarPaciente(&frente, &fim, "Ana", 30);
    adicionarPaciente(&frente, &fim, "João", 45);
    adicionarPaciente(&frente, &fim, "Carlos", 50);
```

```
    cout << endl;

    // Exibindo a fila de pacientes
    exibirFila(frente);

    cout << endl;

    // Atendendo pacientes
    atenderPaciente(&frente);
    atenderPaciente(&frente);

    cout << endl;

    // Exibindo a fila novamente após alguns atendimentos
    exibirFila(frente);

    return 0;
}
```

Explicando o funcionamento do código:

1. **Estrutura Paciente**: Representa cada paciente na fila, com o nome, a idade e um ponteiro para o próximo paciente.
2. **Função adicionarPaciente**: Adiciona um novo paciente ao final da fila. Se a fila estiver vazia, o novo paciente será tanto o início quanto o final da fila.
3. **Função atenderPaciente**: Remove o primeiro paciente da fila, simulando o atendimento. O paciente removido tem sua memória liberada.
4. **Função exibirFila**: Exibe a fila de pacientes mostrando o nome e a idade de cada um.
5. **Função principal (main)**: Adiciona alguns pacientes à fila, os exibe, realiza atendimentos e depois exibe a fila novamente após remover dois pacientes.

OBS.: Neste código há uma estrutura que será explicada no próximo tópico:

```
void atenderPaciente(Paciente** frente)
```

O “**Paciente****” é um recurso de ponteiro que será explicado logo a seguir.

4) Um pequeno parênteses para falarmos sobre ponteiro.

O que é um Ponteiro?

Um **ponteiro** é uma variável que **armazena o endereço de memória** de outra variável. Em vez de armazenar diretamente um valor (como `int` ou `float`), o ponteiro armazena a **localização** (endereço) onde esse valor está armazenado na memória.

Conceitos principais:

1. **Endereço de memória**: Todo valor armazenado no computador (variáveis, arrays, objetos) reside em um endereço específico na memória RAM.
2. **Ponteiro**: Uma variável que guarda o endereço de memória de outra variável.

Sintaxe básica:

- Para declarar um ponteiro, usamos o operador `*`:

```
int* p; // Declara um ponteiro para um inteiro
```

O operador **&** (referência) é usado para obter o **endereço de uma variável**:

```
int x = 10;
p = &x; // Agora 'p' aponta para o endereço de 'x'
```

O operador ***** (desreferência) é usado para acessar o valor armazenado no endereço de memória para o qual o ponteiro aponta:

```
cout << *p; // Exibe o valor de 'x' através do ponteiro (neste caso, 10)
```

Observe no quadro abaixo um breve exemplo do uso do ponteiro.

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;    // Uma variável comum
    int* p;        // Declaração de um ponteiro para um inteiro
```

```
    p = &x;       // O ponteiro 'p' recebe o endereço de 'x'

    cout << "Valor de x: " << x << endl;              // Imprime 10
    cout << "Endereço de x: " << &x << endl;          // Mostra o endereço de 'x'
    cout << "Endereço armazenado em p: " << p << endl; // Mostra o mesmo endereço (onde 'x' está
                                                          armazenado)
      cout << "Valor apontado por p: " << *p << endl;     // Imprime 10 (valor de 'x' através do
                                                          ponteiro)

    return 0;
}
```

Observe a saída deste código no quadro abaixo:

```
Valor de x: 10
Endereço de x: 0x7ffc12345678   // O endereço de memória será diferente em cada execução
Endereço armazenado em p: 0x7ffc12345678
Valor apontado por p: 10
```

Explicando a execução:

1. `x` é uma variável comum que guarda o valor `10`.
2. `p` é um ponteiro que guarda o **endereço de memória** onde `x` está armazenado.
3. Quando acessamos o valor de `p` com `*p`, estamos na verdade acessando o valor de `x` porque `p` aponta para o endereço de `x`.

Usos comuns de Ponteiros:

1. **Acesso a memória dinâmica**: Ponteiros permitem alocar e acessar memória dinamicamente, especialmente com o uso de `new` e `delete`.
2. **Passagem de parâmetros por referência**: Em vez de passar uma cópia de uma variável para uma função, podemos passar um ponteiro para a função acessar diretamente o valor original, economizando memória e tempo.
3. **Estruturas de dados dinâmicas**: Estruturas como **listas encadeadas**, **árvores** e **grafos** dependem de ponteiros para conectar seus elementos, como nós.

Supondo que eu tenho uma Struct definida deste modo:

```
struct Node{

};
```

- **Node**: Representa um nó. É o tipo da estrutura.
- **Node***: É um ponteiro para um nó. Ele guarda o **endereço de memória** de um nó.
- **Node****: Para modificar o ponteiro principal que aponta para o primeiro nó da lista (chamado de `head`), precisamos de um **ponteiro de ponteiro**. Caso contrário, qualquer alteração no ponteiro dentro da função não será refletida fora dela
- **Node***: Um **ponteiro de ponteiro** para um nó. Ele guarda o **endereço de memória de um ponteiro** que, por sua vez, guarda o endereço de um nó.

5) Considerações da aula

Nesta aula aprendemos como a estrutura lista é aplicada em um sistema. O domínio dos conceitos de Pilha, Fila e Lista estudados em Estrutura de Dados e aplicados em Programação Orientada a objetos é uma forma de consolidar estes recursos para a manipulação de dados dentro do sistema.

Bons estudos.