Listas Encadeadas

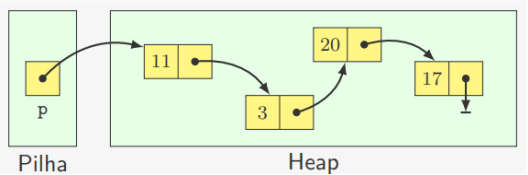
Professor: Rodrigo de Sapienza Luna

16 de outubro de 2025

Vetores

1. Estão alocados contiguamente na memória
   - pode ser que tenhamos espaço na memória
   - mas não para alocar um vetor do tamanho desejado
2. têm um tamanho fixo
   - ou alocamos um vetor pequeno e o espaço pode acabar
   - ou alocamos um vetor grande e desperdiçamos memória

Lista Ligada



- Alocação de memória conforme o necessário
- Guardamos um ponteiro para a estrutura em uma varável
- O primeiro nó aponta para o segundo, o segundo para o terceiro e assim sucessivamente
- O último nó aponta para NULL

Listas Encadeadas

Nó: É composto por duas partes:

1. A informação(o dado em sí);
2. Uma referência para o próximo nó(ponteiro)

Lista Ligada:

1. Conjunto de nós ligados entre sí de maneira sequencial

Observações:

- A lista ligada é acessada a partir de uma varíavel
- um ponteiro pode estar "vazio"

Implementação

- Listas encadeadas são representadas em C utilizando-se estruturas (*struct*).
- A estrutura de cada célula de uma lista ligada pode ser definida da seguinte maneira:

```
1  class celula {
2  public:
3      int dado;
4      celula* prox;
5      // Construtor para inicializar os membros
6      celula(int d, celula* p = nullptr) : dado(d), prox(p) {}
7  };
8
```

Listas encadeadas

- Uma célula c e um ponteiro p para uma célula podem ser declarados da seguinte maneira:

```
1 ●     celula c;
2       celula *p;
```

- Se *c* é uma célula, então *c.dado* é o conteúdo da célula e *c.prox* é o endereço da próxima célula.
- Se *p* é o endereço de uma célula, então $\rightarrow$ é o conteúdo da célula e $\rightarrow$ é o endereço da próxima célula
- Se *p* é o endereço da última célula da lista, então $\rightarrow$ vale *NULL*
- O endereço de uma lista encadeada é o endereço de sua primeira célula. Se *p* é o endereço de uma lista, pode-se dizer simplesmente "p é uma lista"

Listas encadeadas

```
1   class ListaEncadeada {
2   private:
3       Celula *inicio;
4
5   public:
6       ListaEncadeada() {
7           inicio = new Celula(0); // Célula cabeça
8           inicio->prox = nullptr;
9       }
10
11      ~ListaEncadeada() {
12          while (inicio->prox != nullptr) {
13              removeLista(inicio->prox->dado); // Remove todos os nós na destruição
14          }
15          delete inicio; // Libera a célula cabeça
16      }
17  };
18
```

Inserção

- Todas as operações serão apresentadas considerando-se que a lista possui um nó inicial, cujo valor não se tem interesse, denomidado "cabeça"da lista. Esse nó tem apenas a função de apontar para o primeiro elemento inserido na lista.
- A função a seguir deve inserir uma nova célula com conteúdo x após a posição apontada por p(p não pode ser nulo)

```
1  void ListaEncadeada::insereLista (int x)
2  {
3      celula *nova = new celula;
4      nova->dado = x;
5      nova->prox = inicio->prox;
6      inicio->prox = nova;
7  }
8
```

Remoção

```
1  void ListaEncadeada::removeLista(int x) {
2      Celula *p = inicio;        // Ponteiro para percorrer a lista
3      Celula *ant = nullptr;     // Ponteiro para armazenar o nó anterior (não é necessário neste caso)
4
5      // Percorre a lista para encontrar o valor x
6      while (p->prox != nullptr && p->prox->dado != x) {
7          p = p->prox;
8      }
9
10     // Se o valor foi encontrado, remove o nó
11     if (p->prox != nullptr) {
12         Celula *remover = p->prox;   // Nó a ser removido
13         p->prox = remover->prox;     // Ajusta o ponteiro para saltar o nó removido
14         delete remover;              // Libera a memória do nó removido
15     }
16 }
17
```

Impressão

- A função seguinte imprime uma lista a partir da posição apontada por ini → prox.

```
1  void ListaEncadeada::imprime()  {
2      Celula *p = inicio->prox; // Começa do primeiro nó após a cabeça
3      while (p != nullptr) {
4          cout << p->dado << " "; // Imprime o dado seguido de um espaço
5          p = p->prox; // Avança para o próximo nó
6      }
7      cout << endl; // Finaliza a linha após a impressão da lista
8  }
9
```

Complexidade das operações

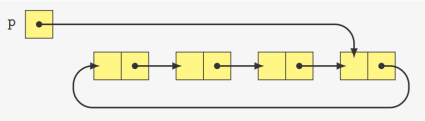
- Inserir: O(1)
- Remover: O(N)
- Imprimir: O(N)

Exercícios:

PDF.

Variações - Lista circulares

A lista encadeada circular é quase igual à lista encadeada simples, mas com uma pequena mudança. Em uma lista encadeada circular, o ponteiro next do último nó aponta para o primeiro nó da lista encadeada, em vez de apontar para NULL. Isso torna essa estrutura de dados circular por natureza.



Aplicações

- Devido a natureza cirurlar, é utilizado em diversas aplicações
- Players de mídia.
- Execução de processos no sistema operacional
- Controlar de quem é a vez em um jogo de tabuleiro

Lista Circular

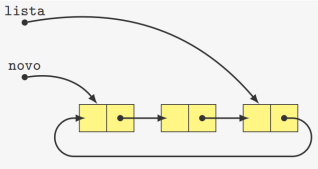
```
1   class ListaCircular {
2   public:
3       Celula *inicio; // Ponteiro para o nó cabeça
4
5       ListaCircular() {
6           inicio = new Celula(0); // Célula cabeça
7           inicio->prox = inicio; // A cabeça aponta para ela mesma
8       }
9
10      ~ListaCircular() {
11          while (inicio->prox != inicio) {
12              remove(inicio->prox->dado); // Remove todos os nós na destruição
13          }
14          delete inicio; // Libera a célula cabeça
15      }
16
17      void insere(int x);
18      void remove(int x);
19      void imprime() const;
20      bool busca(int x) const;
21  };
```

Inserindo em lista circular

```
1    // Método para inserir um elemento na lista
2   void ListaCircular::insere(int x) {
3       Celula *nova = new Celula(x);
4       Celula *p = inicio;
5
6       // Percorre a lista até o último nó
7       while (p->prox != inicio) {
8           p = p->prox;
9       }
10
11      p->prox = nova;  // O último nó aponta para o novo nó
12      nova->prox = inicio; // O novo nó aponta para a cabeça
13  }
14
```

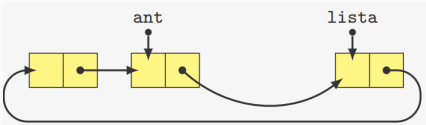
Inserindo em lista circular

- A lista sempre aponta para o último elemento
- O dado do primeiro né elemento é lista → prox → dado
- O dado do último nó elemento é lista → dado
- Para inserir no final, basta devolver novo ao invés de lista



Removendo de lista circular

```
1   // Método para remover um elemento da lista
2   void ListaCircular::remove(int x) {
3       Celula *p = inicio;
4       Celula *ant = nullptr;
5
6       // Percorre a lista para encontrar o valor x
7       do {
8           ant = p;
9           p = p->prox;
10      } while (p != inicio && p->dado != x);
11
12      // Se o valor foi encontrado, remove o nó
13      if (p != inicio) {
14          ant->prox = p->prox; // Ajusta o ponteiro do nó anterior
15          delete p; // Libera a memória do nó removido
16      }
17  }
```



Percorrendo uma lista circular

```
1   // Método para imprimir os elementos da lista
2   void ListaCircular::imprime() const {
3       Celula *p = inicio->prox; // Começa do primeiro nó após a cabeça
4       if (p == inicio) {
5           cout << "Lista vazia." << endl;
6           return;
7       }
8
9       do {
10          cout << p->dado << " ";
11          p = p->prox; // Avança para o próximo nó
12      } while (p != inicio);
13      cout << endl; // Finaliza a linha após a impressão da lista
14  }
```

Busca em lista circular

```cpp
// Método para buscar um elemento na lista
bool ListaCircular::busca(int x) const {
    Celula *p = inicio->prox;
    while (p != inicio) {
        if (p->dado == x) {
            return true; // Elemento encontrado
        }
        p = p->prox; // Avança para o próximo nó
    }
    return false; // Elemento não encontrado
}
```

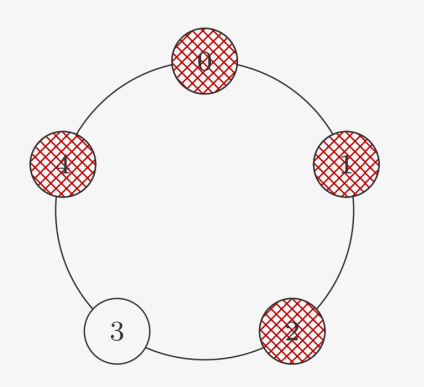
Desafio - Problema de Josephus

Vamos eleger um líder entre N pessoas

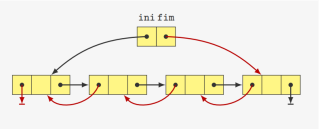
- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos (M + 1)-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos quando chegamos ao final

Exemplo

Exemplo: $N = 5$ e $M = 2$



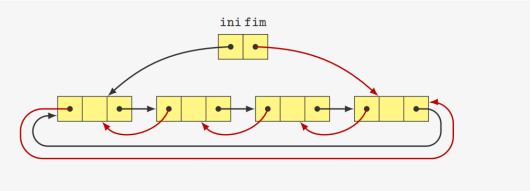
Lista Duplamente ligada



Exemplos:

- Operações desfazer/refazer em softwares
- Player de música (música anterior e próxima)

Lista dupla Circular



Permite inserção e remoção em O(1)

- Variável fim é opcional (fim == ini $\rightarrow$ *ant*)

Lista duplamente ligada

Célula normal

```
1  // Classe que representa um nó na lista duplamente ligada
2  class Celula {
3  public:
4      int dado;            // Valor armazenado no nó
5      Celula *prox;       // Ponteiro para o próximo nó
6      Celula *ant;        // Ponteiro para o nó anterior
7
8      Celula(int valor) : dado(valor), prox(nullptr), ant(nullptr) {}
9  };
```

Lista duplamente ligada

```
1   class ListaDuplamenteLigada {
2   private:
3       Celula *inicio;    // Ponteiro para o primeiro nó
4       Celula *fim;       // Ponteiro para o último nó
5
6   public:
7       // Construtor da lista
8       ListaDuplamenteLigada::ListaDuplamenteLigada() {
9       inicio = nullptr; // Inicializa o ponteiro de início como nulo
10      fim = nullptr;    // Inicializa o ponteiro de fim como nulo
11      }
12     // Destrutor da lista
13      ListaDuplamenteLigada::~ListaDuplamenteLigada() {
14          Celula *p = inicio;
15          while (p != nullptr) {
16              Celula *temp = p;
17              p = p->prox; // Avança para o próximo nó
18              delete temp; // Libera a memória do nó
19          }
20      }
21  };
```

Lista duplamente ligada - Inserção no início

```
1   // Método para inserir um elemento no início da lista
2   void ListaDuplamenteLigada::insereInicio(int x) {
3       Celula *nova = new Celula(x);
4       if (inicio == nullptr) { // Se a lista está vazia
5           inicio = nova;
6           fim = nova;
7       } else {
8           nova->prox = inicio; // O novo nó aponta para o antigo início
9           inicio->ant = nova;  // O antigo início aponta para o novo nó
10          inicio = nova;       // Atualiza o início para o novo nó
11      }
12  }
```

Lista duplamente ligada - Inserção no fim

```
1   // Método para inserir um elemento no final da lista
2   void ListaDuplamenteLigada::insereFim(int x) {
3       Celula *nova = new Celula(x);
4       if (fim == nullptr) { // Se a lista está vazia
5           inicio = nova;
6           fim = nova;
7       } else {
8           nova->ant = fim;    // O novo nó aponta para o antigo fim
9           fim->prox = nova;   // O antigo fim aponta para o novo nó
10          fim = nova;         // Atualiza o fim para o novo nó
11      }
12  }
```

Lista duplamente ligada - Remoção

```
1   // Método para remover um elemento da lista
2   void ListaDuplamenteLigada::remove(int x) {
3       Celula *p = inicio;
4       while (p != nullptr) {
5           if (p->dado == x) { // Se o valor for encontrado
6               if (p->ant != nullptr) {
7                   p->ant->prox = p->prox; // Ajusta o próximo do anterior
8               } else {
9                   inicio = p->prox; // Atualiza o início se necessário
10              }
11              if (p->prox != nullptr) {
12                  p->prox->ant = p->ant; // Ajusta o anterior do próximo
13              } else {
14                  fim = p->ant; // Atualiza o fim se necessário
15              }
16              delete p; // Libera a memória do nó removido
17              return;
18          }
19          p = p->prox; // Avança para o próximo nó
20      }
21  }
```

Lista duplamente ligada - Imprime

```
1  // Método para imprimir os elementos da lista
2  void ListaDuplamenteLigada::imprime() const {
3      Celula *p = inicio;
4      while (p != nullptr) {
5          cout << p->dado << " ";
6          p = p->prox; // Avança para o próximo nó
7      }
8      cout << endl; // Finaliza a linha após a impressão da lista
9  }
```

Lista duplamente ligada - Busca

```
1   // Método para buscar um elemento na lista
2   bool ListaDuplamenteLigada::busca(int x) const {
3       Celula *p = inicio;
4       while (p != nullptr) {
5           if (p->dado == x) {
6               return true; // Elemento encontrado
7           }
8           p = p->prox; // Avança para o próximo nó
9       }
10      return false; // Elemento não encontrado
11  }
```

Lista duplamente ligada - Complexidade

- Inserção no início: O(1)
- Inserção no fim: O(1)
- Remoção do nó por valor: O(N)
- Busca: O(N)