ESTUDO - PROVA

Conteúdo criado para revisão de material de prova segundo os slides apresentados em sala

Código - Pilha.cpp

Exibição do código completo

```
class PilhaDeArray {

    private:
        int *VET;
        int ProximaPosicaoLivre;
        int MAX;

    public:
        PilhaDeArray(int qtde){
            MAX = qtde;
            VET = new int[MAX];
            ProximaPosicaoLivre = 0;
        }


        void Empilha(int n){

            if(ProximaPosicaoLivre < MAX) {
                VET[ProximaPosicaoLivre++] = n;
            }
        }

        void Mostra(){

            for(int i = 0; i<ProximaPosicaoLivre; i++){
                cout <<VET[i] <<"\n";
            }
        }

        int Desempilha(){

            if(ProximaPosicaoLivre > 0){
                                return VET[--ProximaPosicaoLivre];
            }
        }
};

int main(int argc, char *argv[])
{
    PilhaDeArray pilha(50);
    pilha.Empilha(3);
    pilha.Empilha(5);
    pilha.Empilha(7);
    pilha.Empilha(1);
    pilha.Mostra();
    cout <<"\nDesempilha " <<pilha.Desempilha() <<".\n\n";
    cout <<"\nDesempilha " <<pilha.Desempilha() <<".\n\n";
```

```
    pilha.Mostra();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Class **PilhaDeArray{}**

```
class PilhaDeArray {
private:
        int *VET;
        int ProximaPosicaoLivre;
        int MAX;

    public:
        PilhaDeArray(int qtde){
            MAX = qtde;
            VET = new int[MAX];
            ProximaPosicaoLivre = 0;
        }


        void Empilha(int n){

            if(ProximaPosicaoLivre < MAX) {
                VET[ProximaPosicaoLivre++] = n;
            }
        }

        void Mostra(){

            for(int i = 0; i<ProximaPosicaoLivre; i++){
                cout <<VET[i] <<"\n";
            }
        }

        int Desempilha(){

            if(ProximaPosicaoLivre > 0){
                                return VET[--ProximaPosicaoLivre];
            }
        }
};
```

É uma classe nomeada como “PilhaDeArray” responsável por implementar a estrutura de dados Pilha (Stack), utilizando um vetor estático dinamicamente.

A declaração desta classe introduz o uso de conceitos de POO (***Programação Orientada a Objetos*** *),* um paradigma essencial para modularizar estrutura de dados. A classe funciona como um contêiner lógico que agrupa

atributos e comportamentos relacionados à implementação da pilha. isso reforça a ideia de abstração, permitindo que o programador manipule a pilha como um único objeto independente dos detalhes internos de armazenamento ou manipulação de memória.

Em estruturas de dados, essa abordagem possibilita encapsular a lógica da operação LIFO (***Last In - First Out*** *),* facilitando o reuso e evitando que outras partes do código manipulem indevidamente o estado interno da estrutura.

Atributos Privados

```
private:
    int *VET; int ProximaPosicaoLivre; int MAX;
```

- O 1º atributo (***int *VET*** *)* projeta um ponteiro para array de inteiros, criando uma nova instância com o palavra " **new** ". O vetor representa o espaço onde os elementos serão empilhados.
- O 2º atributo (***int ProximaPosicaoLivre*** *)* indica quantos elementos existem na pilha e qual posição será usada para a próxima inserção.
- O 3º atributo (***int MAX*** *)* representa a capacidade máxima na pilha.

Esses atributos privados ilustram o princípio de encapsulamento, pois impedem que outras partes do programa acessem diretamente os dados internos da pilha. O ponteiro ***int *VET*** representa uma área de memória contígua alocada dinamicamente, evidenciando o uso de memória física (RAM), ao passo que a variável ***ProximaPosicaoLivre*** opera sobre a memória lógica (espaço de endereçamento), representando índices e posições abstratas da estrutura.

Já a variável *MAX* define a capacidade total, reforçando a natureza estática da pilha baseada em arranjo, onde o crescimento não é dinâmico. Esses elementos formam a base da representação sequencial contínua típica de estruturas implementadas com arrays.

Constructor

```
public:
    PilhaDeArray(int qtde){
        MAX = qtde;
        VET = new int[MAX];
        ProximaPosicaoLivre = 0;
    }
```

- O constructor solicita um valor definido como quantidade, que é atribuído como parâmetro para operações (***int qtde*** *).*

- A variável (***int MAX*** *)* é atribuída ao valor informado pelo parâmetro do construtor (***int qtde*** *).*
- O ponteiro definido anteriormente aloca dinamicamente a memória para a pilha, indicando a criação de uma nova instância do vetor.
- O uso de *new* indica alocação em heap, ao invés da pilha na CPU, garantindo a flexibilidade em relação ao tamanho da pilha.
- A variável ***ProximaPosicaoLivre*** carrega um valor “neutro”, indicando que a pilha está sem valores. Assim, a próxima operação de ***push()*** será inserida no vetor com o índice definido.

O construtor inicia a estrutura, estabelecendo sua capacidade e alocando memória real através da instrução *new*, demonstrando o funcionamento interno do heap e a manipulação explícita de ponteiros. A variável ***ProximaPosicaoLivre*** é configurada como zero, indicando que a pilha está vazia e que seu topo lógico ainda não contém elementos.

Conceitualmente, este bloco revela a relação entre estruturas abstratas e sua materialização na memória física, além de reforçar a prática de inicialização controlada, sendo um princípio essencial para consistência e segurança em Estrutura de Dados.

Método de Empilhar (***push*** *)*

```
void Empilha(int n){

        if(ProximaPosicaoLivre < MAX) {
            VET[ProximaPosicaoLivre++] = n;
        }
}
```

Esse método implementa a operação fundamental de inserção em pilhas, ilustrando de forma clara o comportamento ***LIFO***. A verificação condicional garante que não ocorro o estouro da estrutura (conceito importante em Estrutura de Dados e em Gerenciamento de Memória). A expressão ***VET[ProximaPosicaoLivre++] = n*** evidencia o uso de indexação direta em memória lógica, onde o incremento pós-fixado modifica o topo da pilha sem deslocar elementos, garantindo eficiência com custo ***O(1)****.*

Em termos de organização computacional, esse método demonstra o uso de valores como estruturas sequenciais de rápido acesso, comuns em implementações internas de sistemas, incluindo mecanismos de armazenamento temporário em **SGBD’S.**

Método ***mostrar()***

```
void Mostra(){

        for(int i = 0; i<ProximaPosicaoLivre; i++){
            cout <<VET[i] <<"\n";
        }
}
```

Esse método percorre sequencialmente o array até o índice que representa o topo da pilha, exemplificando um percurso lineares típico de estruturas estáticas. Ele demonstra como memórias organizadas de forma contígua possibilitam acesso rápido a qualquer elemento, um princípio crucial para o desempenho em Estruturas de Dados.

Ao mesmo tempo, a apresentação em ordem crescente reflete uma visualização estrutural coerente, mesmo que internamente a pilha opere em modelo LIFO. Em analogia a sistemas de Banco de Dados, percursos desse tipo se assemelham à varredura sequencial de páginas armazenadas em memória ou disco.

Método ***desempilhar*** *(pop())*

```
int Desempilha(){

        if(ProximaPosicaoLivre > 0){
                        return VET[--ProximaPosicaoLivre];
        }
}
```

Este método remove e retorna o último elemento inserido, implementando a característica essencial das pilhas. A verificação inicial impede remoções em estruturas vazias, evitando a corrupção de dados.

A operação ***–ProximaPosicaoLivre*** reduz o topo lógico antes da leitura do elemento, refletindo um comportamento de remoção sem deslocamento, com complexidade ***O(1)***. Conceitualmente, essa eficiência é similar a mecanismos utilizados no controle de transações em Banco de Dados, como pilhas de desfazer (*stacks*), onde operações são revertidas na ordem invertida à aplicação.

Método principal (**Main**)

```
int main(int argc, char *argv[])
{
```

```
    PilhaDeArray pilha(50);
    pilha.Empilha(3);
    pilha.Empilha(5);
    pilha.Empilha(7);
    pilha.Empilha(1);
    pilha.Mostra();
    cout <<"\nDesempilha " <<pilha.Desempilha() <<".\n\n";
    cout <<"\nDesempilha " <<pilha.Desempilha() <<".\n\n";
    pilha.Mostra();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

A função principal demonstra o ciclo completo de operações da pilha : inserções sucessivas, visualização da estrutura e remoções. Isso evidencia não apenas o comportamento da pilha na prática, mas também o caráter didático da implementação, mostrando como estruturas de dados teóricos são manipulados por meio de operações concretas.

Em termos de sistemas, esse fluxo representada padrões reais de uso como carregamento e descarregamento de contextos, rotinas de reversão e gerenciamento de chamadas empilhadas.

Código - Fila

Exibição do código completo

```
#include <cstdlib>
#include <iostream>

using namespace std;

class FilaDeArray{

    private:
      int *VET;
      int ProximaPosicaoLivre;
      int MAX;

   public:
      FilaDeArray(int qtde){

         MAX = qtde;
         VET = new int[MAX];
         ProximaPosicaoLivre = 0;
```

```
    }
        void Insere(int n){
            if(ProximaPosicaoLivre <= MAX) VET[ProximaPosicaoLivre++] = n;
        }

        void Mostra(){
            for(int i = 0; i<ProximaPosicaoLivre; i++) {
                cout <<VET[i] <<"\n";
            }
        }

        int Retira(){

            if(ProximaPosicaoLivre > 0){

                int ValorDeRetorno = VET[0];

                for(int i = 1; i < ProximaPosicaoLivre; i++){
                    VET[i-1] = VET[i];
                }

                ProximaPosicaoLivre--; return ValorDeRetorno;
            }
        }
};

int main(int argc, char *argv[])
{
    FilaDeArray fila(50);
    fila.Insere(3);
    fila.Insere(5);
    fila.Insere(7);
    fila.Insere(1);
    fila.Mostra();
    cout <<"\nRetira da fila o " <<fila.Retira() <<".\n\n";
    cout <<"\nRetira da fila o " <<fila.Retira() <<".\n\n";
    fila.Mostra();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Classe **FilaDeArray{}**

```
class FilaDeArray{
private:
        int *VET;
        int ProximaPosicaoLivre;
        int MAX;

    public:
        FilaDeArray(int qtde){

            MAX = qtde;
            VET = new int[MAX];
            ProximaPosicaoLivre = 0;
        }
            void Insere(int n){
                if(ProximaPosicaoLivre <= MAX) VET[ProximaPosicaoLivre++] = n;
            }

            void Mostra(){
                for(int i = 0; i<ProximaPosicaoLivre; i++) {
                    cout <<VET[i] <<"\n";
                }
            }

            int Retira(){

                if(ProximaPosicaoLivre > 0){

                    int ValorDeRetorno = VET[0];

                    for(int i = 1; i < ProximaPosicaoLivre; i++){
                        VET[i-1] = VET[i];
                    }

                    ProximaPosicaoLivre--; return ValorDeRetorno;
                }
            }
};
```

A declaração da classe define o escopo orientado a objetos da estrutura de FIla, encapsulado tanto os dados quanto as operações que caracterizam o comportamento FIFO (***First In - FIrst Out***). Por meio da abstração, a classe permite manipular a fila como uma entidade única, ocultando detalhes de implementação e reforçando a modularidade.

A representação da fila como objeto permite que outras partes do programa a utilizem sem conhecer sua estrutura interna, um princípio essencial para reutilização

e manutenção de código em sistemas complexos, como os elementos de gerenciamento de tarefas de um banco de dados.

Atributos privados

```
private:
    int *VET;  int ProximaPosicaoLivre; int MAX;
```

Os atributos privados do objeto representam o núcleo da estrutura sequencial da fila. O ponteiro ***VET*** refere-se a um bloco contíguo na memória física alocada dinamicamente, onde cada elemento ocupa uma posição adjacente, exemplificando a organização sequencial típicas dos vetores.

A variável ***ProximaPosicaoLivre*** cumpre o papel de memória lógica, definindo o ponto onde o próximo elemento será inserido, sem expor o funcionamento da estrutura interna. A variável ***MAX*** estabelece o limite da estrutura, caracterizando a fila como estática.

Esses atributos, encapsulados, impedem manipulações externas indevidas, reforçando a segurança e consistência dos dados. Um princípio também é utilizado em sistemas gerenciadores de banco de dados ao controlar *buffers* e filas internas de requisições.

Constructor

```
public:
    FilaDeArray(int qtde){

      MAX = qtde;
      VET = new int[MAX];
      ProximaPosicaoLivre = 0;
    }
```

O constructor inicializa a fila com um tamanho máximo definido, alocando espaço contíguo na memória RAM por meio da instrução ***new int[MAX]***. Essa etapa representa o processo de reserva de bloco físico contínuo para armazenamento, característica fundamental de estruturas sequenciais.

A definição inicial de ***ProximaPosicaoLivre = 0*** indica que nenhum elemento foi inserido, preparando a estrutura para sua primeira operação de enfileiramento. Conceitualmente, este trecho corresponde ao processo de inicialização de buffers de entrada em sistemas reais, como os utilizados pelo gerenciador de consultadas de um SGBD (***Sistemas Gerenciador de Banco de Dados***) para organizar transações pendentes.

Método **Inserir()** (*enqueu)*

```
void Insere(int n){
            if(ProximaPosicaoLivre <= MAX) VET[ProximaPosicaoLivre++] = n;
        }
```

Esse método implementa a operação de enfileiramento em uma estrutura FIFO, inserindo novos elementos sempre ao final da sequência. A condição de verificação impede que a fila ultrapasse sua capacidade máxima, prevenindo corrupção de memória.

A instrução de indexação direta seguida de incremento demonstra a eficiência do acesso sequencial contínuo, permitindo inserção em tempo constante ***O(1)***. O Comportamento reflete fielmente a lógica de sistemas de fila utilizados em bancos de dados, como schedulers que armazenam queries (consultas ou solicitações de informações) em ordem de chegada para processamento posterior.

Método ***Mostrar()***

```
void Mostra(){
            for(int i = 0; i<ProximaPosicaoLivre; i++) {
                cout <<VET[i] <<"\n";
            }
}
```

Este método percorre a fila do primeiro ao último elemento existente, exibindo seu conteúdo de forma linear. O laço utiliza memória lógica baseada na quantidade de elementos efetivamente inseridos, não no tamanho físico da estrutura.

A operação sequencial reforça a propriedade de continuidade da memória real, permitindo acesso rápido e direto às posições. No contexto de banco de dados, esse tipo de varredura sequencial similar à inspeção de páginas armazenadas em memória pelos algoritmos de otimização, durante a análise de blocos de dados ou durante operações de varredura sequencial de tabelas (***full scan***).

Método ***Retirar()***

```
int Retira(){

            if(ProximaPosicaoLivre > 0){

                int ValorDeRetorno = VET[0];
```

```
            for(int i = 1; i < ProximaPosicaoLivre; i++){
                VET[i-1] = VET[i];
            }

            ProximaPosicaoLivre--; return ValorDeRetorno;
        }
    }
```

Este método remove o primeiro elemento da fila, implementando a característica central do modelo FIFO. Após recuperar o valor inicial, todos os demais elementos precisam ser deslocados uma posição para a esquerda, compensando o fato de que arrays sequenciais não possuem ponteiros encadeados.

Tal deslocamento representa uma operação de custo *O(n),* pois exige realocação lógica de todos os elementos subsequentes. Esse comportamento está alinhado ao organizar páginas no buffer que devem ser substituídas ou reordenadas, como nas políticas de gerenciamento de memória.

Método principal (**Main**)

```
int main(int argc, char *argv[])
{
    FilaDeArray fila(50);
    fila.Insere(3);
    fila.Insere(5);
    fila.Insere(7);
    fila.Insere(1);
    fila.Mostra();
    cout <<"\nRetira da fila o " <<fila.Retira() <<".\n\n";
    cout <<"\nRetira da fila o " <<fila.Retira() <<".\n\n";
    fila.Mostra();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

A função principal demonstra o funcionamento prático da estrutura : inserções em ordem crescente, visualização do conteúdo, remoções respeitando o modelo FIFO e nova exibição da estrutura resultante.

Esse fluxo simula a lógica de requisições em espera, tal como ocorre em sistemas operacionais ou em bancos de dados, onde consultas, transações e eventos aguardam sua vez em filas de processamento.

A implementação didática evidencia como operações abstratas de Estruturas de Dados se traduzem em comportamento real e previsível em softwares profissionais.

Código - Lista{}

Exibição completa do código

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Lista{

   private:
      int *VET;
      int ProximaPosicaoLivre;
      int MAX;

   public:
      Lista(int qtde){
         MAX = qtde;
         VET = new int[MAX];
         ProximaPosicaoLivre = 0;
      }

         void InserirFim( int n ){
            if ( ProximaPosicaoLivre < MAX ) {
               VET[ ProximaPosicaoLivre++ ] = n;
            }
         }

         void InserirInicio( int n ) {

            if(ProximaPosicaoLivre <= MAX){

                  for ( int i = ProximaPosicaoLivre; i >= 0 ; i-- ){
                           VET[ i ] = VET[ i - 1 ];
                  }

                  VET[ 0 ] = n;
                  ProximaPosicaoLivre++;
            }
         }

         void Mostrar( ){

            for(int i = 0; i < ProximaPosicaoLivre; i++){
               cout <<VET[i] <<"\n";
            }
         }
```

```
        int RetirarInicio( ){

            if(ProximaPosicaoLivre > 0){

                int ValorDeRetorno = VET[0];

                for(int i = 0; i < ProximaPosicaoLivre; i++) {
                    VET[i] = VET[i + 1];
                }

                ProximaPosicaoLivre--; return ValorDeRetorno;
            }
        }

        int RetirarFim( ) {

            if ( ProximaPosicaoLivre > 0 ){
                return VET[--ProximaPosicaoLivre];
            }
         }
};

int main(int argc, char *argv[])
{
    Lista list(50);

    list.InserirInicio(3);
    list.InserirInicio(5);
    list.Mostrar();

    cout <<"\nRetira da fila o " << list.RetirarFim() <<".\n\n";
    cout <<"\nRetira da fila o " << list.RetirarFim() <<".\n\n";
    list.Mostrar();

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Classe **Lista{}**

```
class Lista{

   private:
      int *VET;
      int ProximaPosicaoLivre;
      int MAX;

   public:
      Lista(int qtde){
         MAX = qtde;
         VET = new int[MAX];
         ProximaPosicaoLivre = 0;
      }

         void InserirFim( int n ){
            if ( ProximaPosicaoLivre < MAX ) {
               VET[ ProximaPosicaoLivre++ ] = n;
            }
         }

         void InserirInicio( int n ) {

            if(ProximaPosicaoLivre <= MAX){

                  for ( int i = ProximaPosicaoLivre; i >= 0 ; i-- ){
                           VET[ i ] = VET[ i - 1 ];
                  }

                  VET[ 0 ] = n;
                  ProximaPosicaoLivre++;
            }
         }

         void Mostrar( ){

            for(int i = 0; i < ProximaPosicaoLivre; i++){
               cout <<VET[i] <<"\n";
            }
         }

         int RetirarInicio( ){

            if(ProximaPosicaoLivre > 0){

               int ValorDeRetorno = VET[0];
```

```
            for(int i = 0; i < ProximaPosicaoLivre; i++) {
                VET[i] = VET[i + 1];
            }

            ProximaPosicaoLivre--; return ValorDeRetorno;
        }
    }

    int RetirarFim( ) {

        if ( ProximaPosicaoLivre > 0 ){
            return VET[--ProximaPosicaoLivre];
        }
    }
};
```

A definição da classe estabelece uma abstração orientada a objetos para representar uma lista sequencial, permitindo centralizar operações de inserção, remoção e visualização dos elementos dentro de uma única entidade lógica.

Isso promove encapsulamento, ocultando a forma como os dados são organizados internamente, e reforça o uso de estruturas lineares como componentes modulares. O conceito essencial tanto em Estruturas de Dados quanto em mecanismos internos de bancos de dados, como listas de páginas, listas de registros e catálogos.

Atributos privados

```
private:
    int *VET; int ProximaPosicaoLivre; int MAX;
```

Como ocorre nas demais estruturas, os atributos privados representam os fundamentos técnicos da estrutura sequencial. O ponteiro ***VET*** refere-se a uma área contínua de memória alocada dinamicamente, onde cada elemento ocupa um bloco adjacente, modelo fundamental para listas lineares baseadas em arranjo.

A variável ***ProximaPosicaoLivre*** atua em nível lógico, indicando quantos elementos a lista contém, enquanto a variável ***MAX*** define seu limite físico na RAM. O encapsulamento desses atributos permite que o controle de integridade e a validação de operações sejam gerenciados internamente, assim como ocorre em estruturas internas de bancos de dados que mantêm controle rigoroso sobre suas listas de metadados e páginas alocadas.

Constructor

```
public:
      Lista(int qtde){
         MAX = qtde;
         VET = new int[MAX];
         ProximaPosicaoLivre = 0;
      }
```

A etapa de construção da lista inicia seus limites e cria um vetor de tamanho fixo, alocando memória contígua no heap. A variável ***ProximaPosicaoLivre*** começa em zero, indicando que a lista está vazia. Esse processo é equivalente à alocação inicial de estruturas auxiliares em SGBDs, como tabelas temporárias, listas de buffer ou catálogos de índices, onde a definição do tamanho e da estrutura física precede qualquer operação de manipulação.

Método ***InserirFim()***

```
void InserirFim(int n){
   if (ProximaPosicaoLivre < MAX){
      VET[ProximaPosicaoLivre++] = n;
   }
}
```

Este método realiza a inserção de novos valores no final da lista, utilizando a indexação direta típica de estruturas contíguas. A operação possui custo constante O(1), refletindo a eficiência esperada em listas sequenciais estáticas.

A verificação preventiva assegura que a capacidade máxima não seja ultrapassada, preservando a integridade da estrutura. Em paralelo com bancos de dados, este comportamento lembra o processo de inserção de registros no final de uma página de armazenamento, enquanto ainda há espaço disponível.

Método ***InserirInicio()***

```
void InserirInicio( int n ) {

            if(ProximaPosicaoLivre <= MAX){

                  for ( int i = ProximaPosicaoLivre; i >= 0 ; i-- ){
                        VET[ i ] = VET[ i - 1 ];
                  }
```

```
                VET[ 0 ] = n;
                ProximaPosicaoLivre++;
            }
}
```

A inserção no início da lista demonstra a limitação inerente de estruturas sequenciais estáticas: para abrir espaço na primeira posição, todos os elementos precisam ser deslocados para a direita. Este comportamento reflete a lógica de manipulação de memória contínua, onde cada posição tem endereço físico consecutivo.

O custo **O(n)** dessa operação evidencia conceitos de eficiência e impacto computacional. Esse tipo de deslocamento também é encontrado em sistemas de bancos de dados, por exemplo, ao organizar tabelas heap ou ao manter arrays ordenados de índices

Método ***Mostrar()***

```
void Mostrar( ){

            for(int i = 0; i < ProximaPosicaoLivre; i++){
                cout <<VET[i] <<"\n";
            }
        }
```

O método de exibição percorre a lista linearmente, revelando o arranjo contíguo dos elementos. Ele reforça a noção de acesso sequencial (***linear scan***), conceito amplamente utilizado em bancos de dados para operações como table scan ou varredura de índices. A manipulação do laço demonstra como a estrutura lógica mapeia diretamente para o layout físico no vetor.

Método ***RemoverInicio()***

```
int RetirarInicio( ){

            if(ProximaPosicaoLivre > 0){

                int ValorDeRetorno = VET[0];

                for(int i = 0; i < ProximaPosicaoLivre; i++) {
                    VET[i] = VET[i + 1];
                }

                ProximaPosicaoLivre--; return ValorDeRetorno;
```

```
        }
    }
```

A remoção do primeiro elemento obriga a realocação lógica de todos os demais itens para preencher o espaço vazio. Isso demonstra a diferença entre estruturas sequenciais e encadeadas, onde deslocamentos não seriam necessários.

O custo **O(n)** reforça conceitos de complexidade e impacto de operações sobre listas estáticas. Em modelos de bancos de dados, esse comportamento é análogo a operações de compactação de páginas ou reordenação interna após deleções.

Método ***RemoverFim()***

```
int RetirarFim( ) {

        if ( ProximaPosicaoLivre > 0 ){
            return VET[--ProximaPosicaoLivre];
        }
    }
```

A operação de remoção no fim é extremamente eficiente, realizada em tempo constante O(1). A instrução pré-decremento ajusta o tamanho lógico da estrutura antes da leitura do elemento.

Tal operação demonstra como elementos localizados no final de uma estrutura sequencial podem ser removidos sem necessidade de reorganização. Em bancos de dados, o comportamento é semelhante à liberação do último slot disponível em uma página de dados.

Método principal (**Main**)

```
int main(int argc, char *argv[])
{
    Lista list(50);

    list.InserirInicio(3);
    list.InserirInicio(5);
    list.Mostrar();

    cout <<"\nRetira da fila o " << list.RetirarFim() <<".\n\n";
    cout <<"\nRetira da fila o " << list.RetirarFim() <<".\n\n";
    list.Mostrar();
```

```
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

O método main demonstra inserção no início, exibição da lista e remoções no fim, simulando o comportamento completo da estrutura. Esse fluxo representa de forma didática as operações fundamentais da lista sequencial, aproximando o estudante da lógica de manipulação de estruturas internas usadas em bancos de dados, como listas de páginas, catálogos e registros ordenados.

Código - Lista Encadeada

Exibição completa do código:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class ListaDeArray
{
     private:
          int *VET;
          int ProximaPosicaoLivre;
     public:
          ListaDeArray(int qtde)
          {
             VET = new int[qtde];
             ProximaPosicaoLivre = 0;
          }
          void Insere(int n)
          {
              int pos = ProximaPosicaoLivre - 1;
              while((n < VET[pos]) && (pos > -1))
              {
                     VET[pos+1] = VET[pos];
                     pos--;
              }
              VET[pos+1] = n;
              ProximaPosicaoLivre++;
          }
          void Mostra()
          {
              for(int i = 0; i<ProximaPosicaoLivre; i++)
```

```
            {
                cout <<VET[i] <<"\n";
            }
        }
        int Busca(int n)
        {
            for(int i = 0; i<ProximaPosicaoLivre; i++)
            {
                if(n == VET[i]) return i;
            }
            return -1;
        }
        void Remove(int n)
        {
            int pos = Busca(n);
            if(pos > -1)
            {
                for(int i = pos+1; i < ProximaPosicaoLivre; i++)
                {
                    VET[i-1] = VET[i];
                }
                ProximaPosicaoLivre--;
            }
        }
};

int main(int argc, char *argv[])
{
    ListaDeArray lista(50);
    lista.Insere(1);
    lista.Insere(12);
    lista.Insere(3);
    lista.Insere(7);
    lista.Remove(200);
    lista.Remove(3);
    lista.Mostra();
    cout <<"\n\n\n";
    cout <<lista.Busca(5) <<"\n\n\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Classe **ListaDeArray{}**

```
class ListaDeArray
{
 private:
          int *VET;
          int ProximaPosicaoLivre;
     public:
          ListaDeArray(int qtde)
          {
             VET = new int[qtde];
             ProximaPosicaoLivre = 0;
          }
          void Insere(int n)
          {
              int pos = ProximaPosicaoLivre - 1;
              while((n < VET[pos]) && (pos > -1))
              {
                     VET[pos+1] = VET[pos];
                     pos--;
              }
              VET[pos+1] = n;
              ProximaPosicaoLivre++;
          }
          void Mostra()
          {
              for(int i = 0; i<ProximaPosicaoLivre; i++)
              {
                     cout <<VET[i] <<"\n";
              }
          }
          int Busca(int n)
          {
              for(int i = 0; i<ProximaPosicaoLivre; i++)
              {
                     if(n == VET[i]) return i;
              }
              return -1;
          }
          void Remove(int n)
          {
              int pos = Busca(n);
              if(pos > -1)
              {
                    for(int i = pos+1; i < ProximaPosicaoLivre; i++)
                    {
                           VET[i-1] = VET[i];
                    }
                    ProximaPosicaoLivre--;
              }
          }
```

```
};
```

A definição da classe estabelece uma abstração orientada a objetos para representar uma lista sequencial, permitindo centralizar operações de inserção, remoção e visualização dos elementos dentro de uma única entidade lógica.

Isso promove encapsulamento, ocultando a forma como os dados são organizados internamente, e reforça o uso de estruturas lineares como componentes modulares, conceito essencial tanto em Estruturas de Dados quanto em mecanismos internos de bancos de dados, como listas de páginas, listas de registros e catálogos.

Atributos privados

```
private:
        int *VET;  int ProximaPosicaoLivre;
```

Como ocorre nas demais estruturas, os atributos privados representam os fundamentos técnicos da estrutura sequencial. O ponteiro *VET* refere-se a uma área contínua de memória alocada dinamicamente, onde cada elemento ocupa um bloco adjacente, modelo fundamental para listas lineares baseadas em arranjo.

A variável *ProximaPosicaoLivre* atua em nível lógico, indicando quantos elementos a lista contém, enquanto a variável *MAX* define seu limite físico na RAM. O encapsulamento desses atributos permite que o controle de integridade e a validação de operações sejam gerenciados internamente, assim como ocorre em estruturas internas de bancos de dados que mantêm controle rigoroso sobre suas listas de metadados e páginas alocadas."

Constructor

```
public:
        ListaDeArray(int qtde)
        {
           VET = new int[qtde];
           ProximaPosicaoLivre = 0;
        }
```

A etapa de construção da lista inicializa seus limites e cria um vetor de tamanho fixo, alocando memória contígua no heap. A variável *ProximaPosicaoLivre = 0* indica que a lista está vazia. Esse processo é equivalente à alocação inicial de estruturas auxiliares em SGBDs, como tabelas temporárias, listas de buffer ou

catálogos de índices, onde a definição do tamanho e da estrutura física precede qualquer operação de manipulação.

Método ***Insere()***

```
void Insere(int n)
        {
            int pos = ProximaPosicaoLivre - 1;
            while((n < VET[pos]) && (pos > -1))
            {
                  VET[pos+1] = VET[pos];
                  pos--;
            }
            VET[pos+1] = n;
            ProximaPosicaoLivre++;
        }
```

Este método implementa a inserção ordenada em uma lista sequencial. O algoritmo percorre o vetor de trás para a frente, deslocando elementos para abrir espaço para o novo valor n na posição correta. Isso cria uma lista sempre ordenada, útil para buscas mais previsíveis.

A operação possui custo O(n) no pior caso, devido aos deslocamentos necessários. Esse comportamento é equivalente ao funcionamento de estruturas ordenadas em bancos de dados, como arrays internos usados para construção temporária de índices ou estruturas auxiliares em memória durante certas operações de ordenação (por exemplo, sorting interno).

Além disso, o método demonstra claramente a dualidade entre memória lógica e memória física: logicamente, os elementos mudam de posição; fisicamente, os valores são copiados de um endereço contíguo para o seguinte no bloco de memória. Esse tipo de operação sequencial é típico de estruturas estáticas e reforça a importância da contiguidade física no desempenho.

Método ***Mostra()***

```
void Mostra()
        {
            for(int i = 0; i<ProximaPosicaoLivre; i++)
            {
                  cout <<VET[i] <<"\n";
            }
        }
```

O método percorre a lista sequencialmente e exibe seu conteúdo. A lógica mostra claramente a relação entre memória lógica (posição i) e memória física (endereço real do vetor).

Trata-se de uma varredura linear, equivalente a um table scan em bancos de dados. O custo da operação é O(n). Essa operação ilustra como estruturas sequenciais são ideais para leituras completas e análises ordenadas de dados.

Método ***Busca()***

```
int Busca(int n)
        {
            for(int i = 0; i<ProximaPosicaoLivre; i++)
            {
                  if(n == VET[i]) return i;
            }
            return -1;
        }
```

A busca linear percorre os elementos até encontrar o valor desejado. Embora a lista esteja ordenada, o algoritmo não utiliza busca binária; ainda assim, sua organização pode reduzir comparações médias.

O custo é O(n). Esse procedimento se assemelha a varreduras lineares utilizadas em SGBDs sobre páginas em memória quando não há índices adequados, demonstrando a importância da ordenação e da análise de complexidade.

Método ***Remove()***

```
void Remove(int n)
        {
            int pos = Busca(n);
            if(pos > -1)
            {
                  for(int i = pos+1; i < ProximaPosicaoLivre; i++)
                  {
                        VET[i-1] = VET[i];
                  }
                  ProximaPosicaoLivre--;
            }
        }
```

O método de remoção localiza o elemento através da busca e, em seguida, desloca todos os elementos posteriores uma posição à esquerda para preencher o espaço vazio.

Esta operação também possui custo O(n), reforçando a característica de estruturas sequenciais de custo elevado para remoções internas. Em bancos de dados, comportamento semelhante ocorre na compactação de páginas, quando registros removidos exigem reorganização para mitigar fragmentação.

Método principal (**Main**)

```
int main(int argc, char *argv[])
{
    ListaDeArray lista(50);
    lista.Insere(1);
    lista.Insere(12);
    lista.Insere(3);
    lista.Insere(7);
    lista.Remove(200);
    lista.Remove(3);
    lista.Mostra();
    cout <<"\n\n\n";
    cout <<lista.Busca(5) <<"\n\n\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

A função principal demonstra a utilização da estrutura: criação da lista, inserção ordenada de elementos, tentativa de remoção de um valor inexistente, remoção bem-sucedida, exibição e busca.

Esse fluxo simula operações típicas de manipulação de dados em memória, similares a tarefas realizadas por um SGBD ao manipular estruturas auxiliares ordenadas, tabelas temporárias ou buffers de operações. A lógica demonstra como as estruturas sequenciais conseguem manter ordenação interna sem uso de ponteiros, apenas através de realocação lógica.

Possíveis Resoluções

1) **ListadeArray** - Inserção com prioridade (Triagem)

Resumo do enunciado: pacientes com cartão verde (V) são inseridos no final; pacientes com cartão amarelo (A) têm prioridade e são inseridos no início. O professor pede criar no main os objetos e inserir os cartões na ordem informada (10 - verde, 11 - verde, 5 - amarelo, 12 - verde, 6 - amarelo) e mostrar a lista.

Solução (código simples em C++ usando a classe ListaDeArray dada):

```
int main() {
    ListaDeArray lista(20); // capacidade 20
    // cartões: 10 (V) -> insere fim
    lista.InsereFim(10);
    // 11 (V)
    lista.InsereFim(11);
    // 5 (A) -> insere no início (prioridade)
    lista.InsereInicio(5);
    // 12 (V)
    lista.InsereFim(12);
    // 6 (A)
    lista.InsereInicio(6);

    lista.Mostra(); // mostra a lista organizada
    return 0;
}
```

2) Métodos **RetiraFila** (Array)

RetiraInicio: remove e retorna o primeiro elemento; desloca todos os elementos para a esquerda e decrementa ***ProximaPosicaoLivre.***

Implementação adaptada

```
int RetiraInicio() {
    if (ProximaPosicaoLivre > 0) {
        int ValorDeRetorno = VET[0];
        for (int i = 1; i < ProximaPosicaoLivre; i++) {
            VET[i-1] = VET[i];
        }
        ProximaPosicaoLivre--;
        return ValorDeRetorno;
    }
    // se vazia, retornar valor sentinela (ex.: -1)
    return -1;
}
```

RetiraFim : retorna o último elemento e decrementa ***ProximaPosicaoLivre***

```
int RetiraFim() {
    if (ProximaPosicaoLivre > 0) {
        ProximaPosicaoLivre--;
        return VET[ProximaPosicaoLivre];
    }
    return -1;
}
```

3) Método **Remove** (ListaArray)

Remove(int n): procura posição com Busca(n). Se encontrada, desloca os elementos à direita para a esquerda para cobrir e decrementa ***ProximaPosicaoLivre.***

Exemplo de implementação (para Lista de Array):

```
void Remove(int n) {
    int pos = Busca(n);
    if (pos > -1) {
        for (int i = pos + 1; i < ProximaPosicaoLivre; i++) {
            VET[i-1] = VET[i];
        }
        ProximaPosicaoLivre--;
    }
}
```

4) Questões de **múltipla escolha**

Segundo o trecho

- Funcionário objFuncionario("Rogério Ceni", "São Paulo F.C.", "Goleiro", 300.000,00);

Resposta:

- Os valores entre parênteses são parâmetros passados (nome, time, posição, salário) para o construtor da classe Funcionário.

5) Métodos que insere **algoritmo de ordenação** na Lista (análise)

Código exibido realiza inserção ordenada em vetor: percorre da direita para a esquerda enquanto ***n < VET[pos]*** , desloca e coloca ***n*** na posição correta. Portanto: "**Ele insere dados ordenados na Lista**."

6) **Programação Orientada a Objetos**

1. **Encapsulamento** : Interfaces visíveis, atributos internos não acessíveis.
2. **Polimorfismo** : Objetos podem herdar e utilizar métodos de uma superclasse e sobrescrever-los.
3. **Herança :** O conceito não faz duas operações com o mesmo nome na mesma classe, permitindo reutilizar e estender a estrutura.

7) Verificar sequência de **Fibonacci** usando Pilha

Pedir ao usuário para digitar 8 números num vetor. Usar uma pilha para empilhar quando encontrar um número diferente da sequência esperada. Se no final a pilha estiver cheia (ou não vazia) significa erro. Abaixo um exemplo em C++ (simples):

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

bool ehFiboSeq(int v[], int n) {
    if (n < 3) return true;
    for (int i = 2; i < n; i++) {
        if (v[i] != v[i-1] + v[i-2]) return false;
    }
    return true;
}

int main() {
    const int N = 8;
    int vet[N];
    for (int i = 0; i < N; i++) {
        cout << "Digite elemento " << i << ": ";
        cin >> vet[i];
    }
    if (ehFiboSeq(vet, N)) cout << "Sequencia correta\n";
    else cout << "Sequencia incorreta\n";
    return 0;
}
```

8) Implementar classe **Número (Y1, Y2, Y3)**

Exemplificação :

```
class Numero {
private:
   int Y1, Y2, Y3;
public:
   // a) construtor parametrizado
   Numero(int a, int b, int c) : Y1(a), Y2(b), Y3(c) {}

   // b) incrementa todos
   void incrementa() { Y1++; Y2++; Y3++; }

   // c) decrementa todos
   void decrementa() { Y1--; Y2--; Y3--; }

   // d) retorna o maior
   int maior() {
      int m = Y1;
      if (Y2 > m) m = Y2;
      if (Y3 > m) m = Y3;
      return m;
   }
};
```

Exemplo de uso no main :

```
Numero n(3,5,1); n.incrementa(); cout << n.maior();
```

9) Usar ListaArray para inserir: vet[5, 3, 10, 8, 9) e remover dois inexistentes e existentes

Procedimento simples

- Inserir os valores pela função *InsereFim* ou *InsereOrdenado* conforme a classe disponibilizada. Se tentar remover um valor inexistente, a função *Busca* retorna *-1* e nada acontece.
- Se remover existente, o elemento é removido e os seguintes deslocados para esquerda.

Exemplo de execução (resultado esperado)

- **Inserção** : [5, 3, 10, 8, 9] (dependendo de *InsereFim* a ordem fica igual
- **Remover dois existentes :** nenhum valor alterado.
- **Remover dois existentes :** vet[5, 8, 9] (- 10, -3)

10) Para que é utilizada a classe **Busca?**

Explicação

- A classe / função ***Busca*** procura um elemento no vetor / lista e retorna sua posição (índice) ou -1, caso não encontrar um valor. Também é utilizada no método ***Remove*** para saber qual posição deve ser apagada.