

Estruturas Lineares

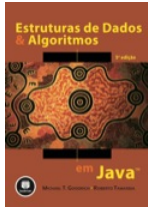
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

25 de setembro de 2024

Introdução

Leitura(s) Recomendada(s)



Seções 3.2 (Listas simplesmente encadeadas), 3.3 (Listas duplamente encadeadas), 6.1 (Listas arranjo), 6.2 (Listas de nodos), 6.4 (Os TADs de lista e o *framework* de coleções)

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados e algoritmos em Java**. Tradução: Bernardo Copstein. 5. ed. Porto Alegre: Bookman, 2013. xxii, 713 p. E-book. ISBN 9788582600191. Tradução de: Data Structures and Algorithms in Java, 5th Edition. Disponível em: <<https://integrada.minhabiblioteca.com.br/#/books/9788582600191/>>. Acesso em: 01 ago. 2023.

Tipos Abstratos de Dados

Abordagem OO

- Princípios da abordagem OO
 - **Abstração**: representação de um objeto do mundo real, “abstraindo-se” os detalhes desnecessários, de forma que o objeto possa ser utilizado sem se preocupar com como ele foi implementado
 - **Encapsulamento**: detalhes da implementação ficam escondidos e a manipulação dos dados acontece através de uma interface pública
 - **Modularidade**: vários componentes que interagem
- Abstração, Encapsulamento, Herança e Polimorfismo são considerados os 4 pilares da POO

Tipos Abstratos de Dados

- A aplicação de abstração ao projeto de estruturas de dados nos leva a **Tipos Abstratos de Dados (TAD)**
- TAD
 - É uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados
 - Modelo matemático de estruturas de dados que especifica
 - O tipo dos dados armazenados
 - As operações definidas sobre esses dados
 - Os tipos dos parâmetros dessas operações
- A separação de especificação e implementação permite usar um TAD sem conhecer nada sobre a sua implementação
 - Assim, um TAD pode ter mais de uma implementação

TAD

- Usa “encapsulamento”
- Princípio: esconder detalhes de representação e direcionar o acesso aos objetos abstratos por meio de operações
- A representação fica protegida contra qualquer tentativa de manipulá-la diretamente (só através das operações disponíveis)
- Define o que cada operação faz, mas não como o faz

Tipos Abstratos de Dados

- Resumindo, TAD é uma estrutura de programa que contém
 - A especificação de uma estrutura de dados
 - Um conjunto de operações que podem ser realizadas sobre os dados encapsulados
- Exemplos de TADs
 - Pilhas, Filas, Deques e Listas
- Essas estruturas são classificadas como lineares
 - Representam coleções de elementos linearmente organizados que oferecem métodos para inserir, acessar e remover elementos
 - Têm a ordem interna de seus elementos definida pela forma como são feitas inserções e remoções na estrutura
 - Costumam ter duas extremidades (esquerda e direita; frente e traseira; cabeça e cauda; ...)

Estruturas Lineares

- **Lista**

- Organiza os dados de maneira “sequencial” (não necessariamente de forma física, mas sempre existe uma ordem lógica entre os elementos)
- Permite inserção, acesso e remoção de elementos

- **Pilha**

- Usa a política *LIFO – Last In First Out* (o último elemento que entrou, é o primeiro a sair)
- Possui apenas uma entrada, chamada de topo, a partir da qual os dados entram e saem dela

- **Fila**

- Usa a política *FIFO – First In First Out* (o primeiro elemento a entrar será o primeiro a sair)
- Os elementos entram por um lado (“cauda” ou parte de trás) e saem por outro (“cabeça” ou parte da frente)

- **Deque (*Double-Ended QUEue*)**

- Os elementos entram e saem por qualquer uma das extremidades (cauda ou cabeça) da lista

Estruturas Lineares

- Permitem representar um conjunto de dados de um mesmo tipo (com alguma afinidade) de forma a preservar a relação de ordem entre seus elementos
- Cada elemento da estrutura é chamado de nó, ou nodo.
- Uma estrutura linear é definida como:
 - Um conjunto de N nós, organizados de forma a refletir a posição relativa dos mesmos
 - Se $N > 0$, os nós da estrutura serão x_1, x_2, \dots, x_N ,
 - x_1 é o primeiro nó
 - Para $1 < k < N$, o nó x_k é precedido pelo nó x_{k-1} e seguido pelo nó x_{k+1}
 - x_N é o último nó
 - Quando $N = 0$, diz-se que a estrutura está vazia

Exemplos de Estruturas Lineares

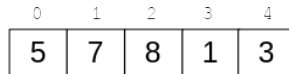
- Pessoas na fila de um caixa (ordem definida pela chegada e posição na fila)
- Pessoas na sala de espera de um consultório (ordem definida pela chegada)
- Conjunto de notas dos alunos de uma turma
- Itens no estoque de uma loja
- Palavras de um texto
- Letras de uma palavra
- Especificação de operações e operandos em uma expressão matemática
- Dias da semana
- Relação de compromissos
- Pilha de livros
- Cartas de um baralho
- etc.

Alocação de Estruturas Lineares

- Estruturas lineares podem ser alocadas de forma:

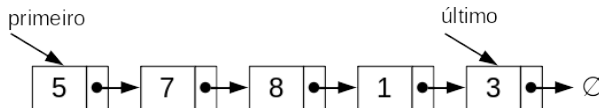
- Sequencial ou Contígua**

Os nós, além de estarem em uma sequência lógica, também estão **fisicamente em sequência**



- Encadeada**

Os nós são alocados dinamicamente e são ligados entre si, de forma que há uma sequência lógica, mas fisicamente os nós **NÃO** precisam estar contíguos



- Cada forma tem as suas vantagens e desvantagens

Alocação Sequencial ou Contígua de Estruturas Lineares

- Nós adjacentes na estrutura são armazenados em endereços contíguos na memória física e o tamanho da estrutura é fixo
- A implementação é feita com vetores (arranjos ou *arrays*), que podem ser alocados de forma estática ou dinâmica
- Pode-se trabalhar com vetores parcialmente preenchidos
- O acesso é rápido
- NÃO é possível ter espaços vazios (não utilizados) no meio da estrutura (a não ser no final, para vetores parcialmente preenchidos)
- Inserção e Remoção de elementos no meio exige movimentação de elementos
- Para estruturas alocadas dinamicamente, pode-se: alocar um novo vetor, copiar os elementos do antigo para o novo, desalocar o antigo e passar a usar o novo – mas isto pode ser custoso

Alocação Sequencial ou Contígua de Estruturas Lineares (vetores.cpp)

```
#include <iostream>
using namespace std;
int main() {

    int vetorEstatico[10]; // ALOCAÇÃO ESTATICA
    for (int i=0; i<10; ++i) vetorEstatico[i] = i+1;
    for (int i=0; i<10; ++i) cout << vetorEstatico[i] << endl;

    const int TAM_MAX = 10;
    int vetorParcial[TAM_MAX]; // VETOR PARCIALMENTE PREENCHIDO
    int tamAtual = 0; // tamanho atual do vetor parcialmente preenchido
    for (int i=0; i<TAM_MAX+1; ++i)
        if ( tamAtual < TAM_MAX)
            vetorParcial[ tamAtual++ ] = i+1;
    for (int i=0; i<tamAtual; ++i) cout << vetorParcial[i] << endl;

    int tam;
    cin >> tam;
    int *vetorDinamico = new int[tam]; // ALOCAÇÃO DINAMICA
    for (int i=0; i<tam; ++i) vetorDinamico[i] = i+1;
    for (int i=0; i<tam; ++i) cout << vetorDinamico[i] << endl;
    delete[] vetorDinamico;

    return 0;
}
```

Alocação Sequencial ou Contígua de Estruturas Lineares (vetores2.cpp)

```
#include <iostream>

using namespace std;

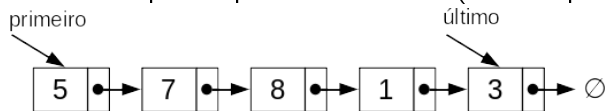
int main() {

    const int TAM = 10;
    int *vetorExpansivel = new int[TAM]; // VETOR PARCIALMENTE PREENCHIDO DINAMICAMENTE EXPANSIVEL
    int tamAtual = 0, tam_max = TAM;
    for (int i=0; i<TAM+5; ++i) {
        if ( tamAtual == tam_max ) {
            int *novo = new int[tam_max + TAM];
            for (int j=0; j<tamAtual; ++j) novo[j] = vetorExpansivel[j];
            delete[] vetorExpansivel;
            vetorExpansivel = novo;
            tam_max += TAM;
        }
        vetorExpansivel[ tamAtual++ ] = i+1;
    }
    for (int i=0; i<tamAtual; ++i) cout << vetorExpansivel[i] << endl;
    delete[] vetorExpansivel;

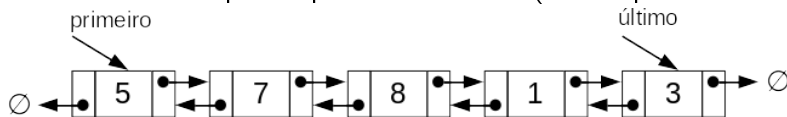
    return 0;
}
```

Alocação Encadeada de Estruturas Lineares

- Os elementos da estrutura seguem uma ordem lógica, mas **NÃO** estão necessariamente armazenados sequencialmente na memória
- A relação lógica de ordem é implementada através de uma ligação (referência ou armazenamento de endereço) entre os nodos
- Estruturas lineares encadeadas são chamadas de **listas encadeadas**, sendo que cada nodo pode armazenar uma referência para o próximo elemento (lista simplesmente encadeada)



- Ou para o elemento anterior e para o próximo elemento (lista duplamente encadeada)



- A estrutura pode aumentar e diminuir em tempo de execução

Alocação Encadeada de Estruturas Lineares

- Quando for necessário inserir um elemento na estrutura, deve-se:
 - Alocar um novo nodo
 - Preencher as informações no nodo
 - Inserir o novo nodo em determinada posição da estrutura (o que exige ajustes em alguns encadeamentos)
- Alocação encadeada será útil quando:
 - Não é possível prever o número de entradas de dados em tempo de compilação
 - For mais fácil aplicar determinada operação sobre a estrutura encadeada

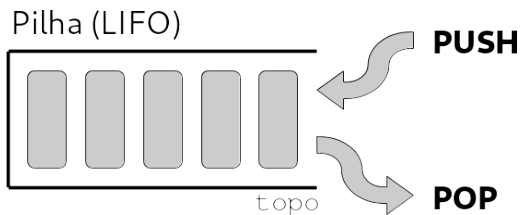
Operações Básicas sobre Estruturas Lineares

- Criação da estrutura
- Destruição da estrutura
- Inserção de um elemento na estrutura
- Remoção de um elemento da estrutura
- Acesso a um elemento da estrutura
- Alteração de um elemento da estrutura
- Combinação de duas ou mais estruturas
- Ordenação dos elementos da estrutura
- Cópia de uma estrutura
- Localização de um nodo através de alguma informação do nodo

Pilha (*Stack*)

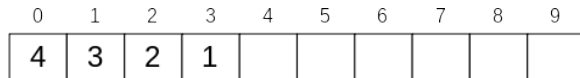
Pilha ou *Stack*

- Usa a política *LIFO* – *Last In First Out* (o último elemento que entrou, é o primeiro a sair)
- Possui apenas uma entrada, chamada de topo, a partir da qual os dados são inseridos e removidos



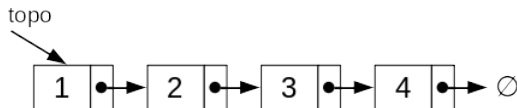
Pilha: Implementações Possíveis

- **Arranjo**

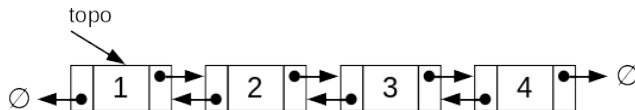


tamMax = 10 / tam = 4 / posTopo = 3 / valTopo = 1

- **Lista Simplesmente Encadeada**



- **Lista Duplamente Encadeada**



Aplicações que Usam Pilha

- Operações de edição de desfazer/refazer
- Histórico de visitação de páginas em navegadores *web* (botão *back*)
- Cadeia de chamada de métodos em interpretadores e máquinas virtuais
- Auxiliar para implementação de outras estruturas de dados e algoritmos
- Implementação de compiladores
- Computação Gráfica (operações com matrizes)
- Manipulação de expressões aritméticas: infixada, pré-fixada, pós-fixada

Manipulação de Expressões Aritméticas

- Expressões aritméticas geralmente são representadas usando a notação infixada

$$2 + 3 * 4 = 14$$

$$(2 + 3) * 4 = 20$$

- Alternativa 1: Notação polonesa (pré-fixada)

- Proposta por Jan lukasiewicz em 1920
- Permite escrever expressões aritméticas com precedência implícita
- Operador aparece antes dos operandos

$$+ 2 * 3 4$$

$$* + 2 3 4$$

- Alternativa 2: Notação polonesa inversa (pós-fixada)

- Proposta por Charles Hamblin em 1950
- Também usa precedência implícita, porém o operador aparece antes dos operandos

$$2 3 4 * +$$

$$2 3 + 4 *$$

Exercício

Converta as seguintes expressões na forma infixada para as formas pré-fixada e pós-fixada:

❶ Infixada: $(1-2)*(3+4)$

Pré-fixada:

Pós-fixada:

❷ Infixada: $(2 + 4)/(3 - 1)$

Pré-fixada:

Pós-fixada:

❸ Infixada: $(2+4)/(3-1)*4$

Pré-fixada:

Pós-fixada:

Exercício (Resposta)

Converta as seguintes expressões na forma infixada para as formas pré-fixada e pós-fixada:

❶ Infixada: $(1-2)*(3+4)$

Pré-fixada: $* - 1 2 + 3 4$

Pós-fixada: $1 2 - 3 4 + *$

❷ Infixada: $(2 + 4)/(3 - 1)$

Pré-fixada: $/ + 2 4 - 3 1$

Pós-fixada: $2 4 + 3 1 - /$

❸ Infixada: $(2+4)/(3-1)*4$

Pré-fixada: $* / + 2 4 - 3 1 4$

Pós-fixada: $2 4 + 3 1 - / 4 *$

Métodos do TAD Pilha

- `bool push(e)`: insere o elemento no topo da pilha (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool pop(&e)`: remove e retorna (por referência) o elemento do topo da pilha (retorna `true`, em caso de sucesso, ou `false`, a pilha estiver vazia)
- `bool top(&e)` ou `bool peek(&e)`: retorna (por referência) o elemento do topo da pilha, mas não o remove da pilha (retorna `true`, em caso de sucesso, ou `false`, a pilha estiver vazia)
- `int size()`: retorna o número de elementos da pilha
- `int maxSize()`: retorna o número máximo de elementos suportado pela pilha
- `bool isEmpty()`: retorna `true`, se a pilha estiver vazia, ou `false`, em caso contrário
- `bool isFull()`: retorna `true`, se a pilha estiver cheia, ou `false`, em caso contrário
- `void clear()`: esvazia a pilha

Exemplo de Implementação: IntStack.hpp

```
#ifndef _INTSTACK_HPP
#define _INTSTACK_HPP

#include <string>

using namespace std;

class IntStack {
private:
    int numElements;
    int maxElements;
    int *stack;
public:
    IntStack(int mxSz = 10);
    ~IntStack();
    int size() const;
    int maxSize() const;
    bool isEmpty() const;
    bool isFull() const;
    void clear();
    bool push(const int e);
    bool pop(int &e);
    bool top(int &e) const;
    string str() const; // APENAS PARA DEPURACAO
};

#endif
```

Exemplo de Implementação: IntStack.cpp

```

#include <sstream>
#include "IntStack.hpp"

IntStack::IntStack(int mxSz) {
    numElements = 0;    maxElements =  ( mxSz < 1 ) ? 10 : mxSz;
    stack = new int[maxElements];
}

IntStack::~IntStack() { delete[] stack; }
int IntStack::size() const { return numElements; }
int IntStack::maxSize() const { return maxElements; }
bool IntStack::isEmpty() const { return numElements == 0; }
bool IntStack::isFull() const { return numElements == maxElements; }
void IntStack::clear() { numElements = 0; }

bool IntStack::push(const int e) {
    if ( numElements == maxElements ) return false;
    else { stack[ numElements++ ] = e; return true; }
}

bool IntStack::pop(int &e) {
    if ( numElements == 0 ) return false;
    else { e = stack[ --numElements ]; return true; }
}

bool IntStack::top(int &e) const {
    if ( numElements < 1 ) return false;
    else { e = stack[ numElements-1 ]; return true; }
}

string IntStack::str() const {
    int i;    stringstream ss;
    ss << "|";
    for (i=0; i<numElements; ++i) ss << stack[i] << "|";
    for (; i<maxElements; ++i) ss << "░|";
    return ss.str();
}

```

Exemplo de Implementação: IntStackMain.cpp

```

#include <iostream>
#include "IntStack.hpp"

using namespace std;

void print(IntStack &stack) {
    cout << "uu" << stack.str() << "uu" << "size=" << stack.size() << "/" << stack.maxSize() << "uu" << "top=";
    int t;    bool res = stack.top(t);
    if (res) cout << t; else cout << "X";
    cout << "uu" << "isEmpty=" << stack.isEmpty() << "uu" << "isFull=" << stack.isFull() << endl;
}

int main() {
    int e;
    bool res;
    cout << "IntStack(4):u"; IntStack stack(4); print(stack);
    e = 1; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 2; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 3; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    e = 4; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 5; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 6; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(X):" << "u"; cout << (res?"OKuu":"ERRO"); print(stack);
    e = 7; cout << "push(" << e << "):" << "u"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    cout << "clear():OKuu"; stack.clear(); print(stack);
    return 0;
}

```

Exemplo de Implementação (Saída)

IntStack(4):		size=0/4	top=X	isEmpty=1	isFull=0
push(1): OK	1	size=1/4	top=1	isEmpty=0	isFull=0
push(2): OK	1 2	size=2/4	top=2	isEmpty=0	isFull=0
push(3): OK	1 2 3	size=3/4	top=3	isEmpty=0	isFull=0
pop(3): OK	1 2	size=2/4	top=2	isEmpty=0	isFull=0
push(4): OK	1 2 4	size=3/4	top=4	isEmpty=0	isFull=0
push(5): OK	1 2 4 5	size=4/4	top=5	isEmpty=0	isFull=1
push(6): ERRO	1 2 4 5	size=4/4	top=5	isEmpty=0	isFull=1
pop(5): OK	1 2 4	size=3/4	top=4	isEmpty=0	isFull=0
pop(4): OK	1 2	size=2/4	top=2	isEmpty=0	isFull=0
pop(2): OK	1	size=1/4	top=1	isEmpty=0	isFull=0
pop(1): OK		size=0/4	top=X	isEmpty=1	isFull=0
pop(X): ERRO		size=0/4	top=X	isEmpty=1	isFull=0
push(7): OK	7	size=1/4	top=7	isEmpty=0	isFull=0
clear(): OK		size=0/4	top=X	isEmpty=1	isFull=0

Exercícios

Exercícios 1-3

- 1 Considerando como base a implementação da classe `IntStack` (apresentadas nas lâminas anteriores), implemente uma classe em C++ para gerenciar uma pilha de caracteres (`CharStack`).
- 2 Usando a classe `CharStack` (do exercício anterior), escreva um programa em C++, que inverte as letras de cada palavra de um texto terminado por ponto (.), preservando a ordem das palavras.

Por exemplo, dado o texto:

ETSE OICICREXE E OTIUM LICAF.

A saída deve ser:

ESTE EXERCICIO E MUITO FACIL.

- 3 Implemente uma função em C++ para criar uma nova cópia de uma pilha de caracteres (objeto da classe `CharStack`). A pilha deve ser criada usando `new`, deve ser uma cópia duplicada exatamente igual à pilha original e deve ser retornada como resultado da execução da função. Sua função deve ter o seguinte protótipo:

`CharStack *copia(CharStack &p);`

No retorno da função, o conteúdo da pilha recebida como parâmetro deve ser o mesmo da chamada.

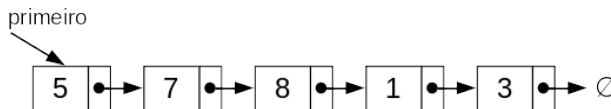
Exercícios 4-5

- 4 Implemente uma função em C++ para testar se duas pilhas de caracteres (objetos da classe CharStack) são iguais. Sua função deve ter o seguinte protótipo:
- ```
bool saoIguais(CharStack &p1, CharStack &p2);
```
- Para que duas pilhas sejam consideradas iguais, elas devem ter o mesmo número de elementos nas mesmas posições; no entanto, o tamanho máximo das pilhas NÃO precisa ser igual. No retorno da função, o conteúdo das pilhas deve ser o mesmo da chamada.
- 5 Considerando ainda a classe CharStack, implemente uma função em C++, usando o conceito de pilha, para testar se uma palavra (string) recebida como parâmetro é um palíndromo ou não. Sua função deve ter o seguinte protótipo:
- ```
bool ehPalindromo(string s);
```

Implementando uma Pilha Encadeada

Estruturas Lineares Encadeadas

- Uma estrutura encadeada é uma estrutura composta por nodos, que possuem campos que apontam para outros nodos
- Por exemplo, em uma lista simplesmente encadeada, tipicamente:
 - Há um ponteiro para o primeiro elemento da lista
 - Cada nodo tem um campo que aponta para o próximo elemento da lista
 - O campo de encadeamento do último elemento contém uma referência inválida



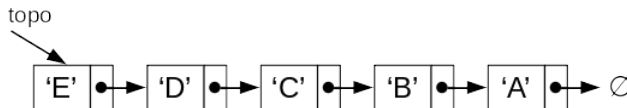
- Ao inserir um elemento, deve-se alocar um nodo, preenchê-lo e colocá-lo na posição desejada na estrutura (ajustando as referências necessárias)
- Ao remover um elemento, é preciso removê-lo (ajustando as referências necessárias) e desalocá-lo

Estruturas Encadeadas em C++

- Pode-se declarar um nodo em C++, para armazenar caracteres, da seguinte forma:

```
struct Node {
    char info;
    Node *next;
    Node(char l) {
        info = l;
        next = nullptr;
    }
};
```

- nullptr é usado para indicar uma estrutura vazia ou fim da estrutura
- Exemplo: construção da seguinte pilha simplesmente encadeada



Exemplo: pilha_simplesmente_encadeada.cpp

```

#include <iostream>

using namespace std;

struct Node {
    char info;
    Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    Node *nodo1 = new Node('A');
    Node *nodo2 = new Node('B');
    Node *nodo3 = new Node('C');
    Node *nodo4 = new Node('D');
    Node *nodo5 = new Node('E');

    Node *topo = nodo5;
    nodo5->next = nodo4;
    nodo4->next = nodo3;
    nodo3->next = nodo2;
    nodo2->next = nodo1;

    for (Node *aux = topo; aux != nullptr; aux = aux->next)
        cout << aux->info << endl;

    while (topo != nullptr) {
        Node *aux = topo;
        topo = topo->next;
        delete aux;
    }

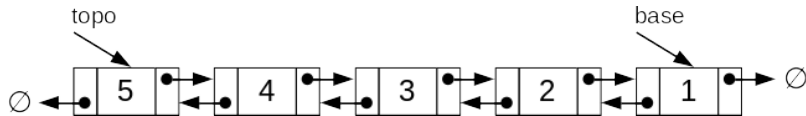
    return 0;
}

```

Exercícios

Exercícios 6 e 7

- 6 Modifique o programa da lâmina anterior para que a lista seja criada **com um laço**, exatamente com o mesmo conteúdo e na mesma ordem lógica. Faça as inserções sempre pela mesma extremidade da estrutura encadeada.
- 7 Usando como modelo o programa da lâmina anterior, construa em C++ a seguinte lista duplamente encadeada. Percorra a lista, mostrando o conteúdo de seus nodos, tanto do início para o fim, quanto do fim para o início.



Exercício 8

- 8 Usando como ponto de partida a classe `IntStack`, apresentada anteriormente e implementada usando alocação sequencial ou contígua, implemente a classe `IntLinkedStack`, que funciona da mesma forma, porém usando alocação encadeada.

Observações:

- A versão usando alocação encadeada eliminará a necessidade de se trabalhar com um limite máximo de elementos, conseqüentemente, os métodos `maxSize()` e `isFull()` NÃO farão parte da nova implementação.
- A definição da classe (arquivo `IntLinkedStack.hpp`) e um programa de teste (arquivo `IntLinkedStackMain.cpp`) estão listados nas lâminas a seguir.

Exercício 8: IntLinkedStack.hpp

```

#ifndef _INTLINKEDSTACK_HPP
#define _INTLINKEDSTACK_HPP
#include <string>
using namespace std;

class IntLinkedStack {
private:
    int numElements;
    struct Node {
        int data;
        Node *next;
        Node(int d) { data = d; next = nullptr; }
    };
    Node *stack;
public:
    IntLinkedStack();
    ~IntLinkedStack();
    void push(const int e);
    bool pop(int &e);
    bool top(int &e) const;
    int size() const;
    bool isEmpty() const;
    void clear();
    string str() const; // APENAS PARA DEPURACAO
};
#endif

```

Exercício 8: IntLinkedStackMain.cpp

```

#include <iostream>
#include "IntLinkedStack.hpp"

using namespace std;

void print(IntLinkedStack &stack) {
    cout << "uu" << stack.str() << "uu size=" << stack.size() << "uutop=";
    int t;    bool res = stack.top(t);
    if (res) cout << t; else cout << "X";
    cout << "uuisEmpty=" << stack.isEmpty() << endl;
}

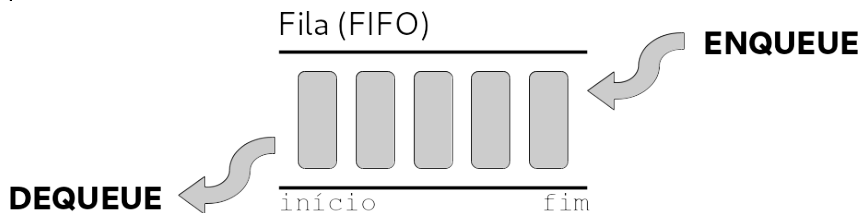
int main() {
    int vet[] = {0,1,2,3,4,5,6,7,8,9};
    int numElements = sizeof(vet)/sizeof(int);
    IntLinkedStack stack;
    cout << "uuuuuuuuuu"; print(stack);
    for (int i=0; i<numElements; ++i) {
        cout << "push(" << vet[i] << "):uu";
        stack.push( vet[i] );
        print(stack);
    }
    for (int i=0; i<numElements+1; ++i) {
        int e;
        bool res = stack.pop( e );
        if ( !res ) cout << "pop(X):uuu";
        else cout << "pop(" << e << "):uuu";
        print(stack);
    }
    return 0;
}

```

Fila (*Queue*)

Fila ou Queue

- Usa a política *FIFO* – *First In First Out* (o primeiro que entrou, é o primeiro a sair)
- Possui uma entrada (fim), a partir da qual os dados são inseridos, e uma saída (início), a partir da qual os dados são removidos

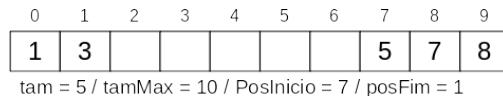
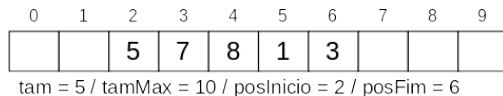


Aplicações que Usam Fila

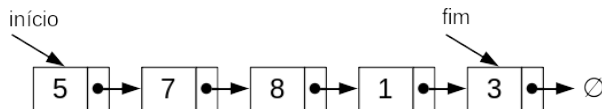
- Desenvolvimento de aplicativos
 - Gerenciamento de transações para aplicativos de lojas, teatros, centros de reserva, etc.
- Simulações
 - Listas de espera na simulação de sistemas de atendimento (banco, supermercado, etc.)
- Sistemas Operacionais
 - Fila de documentos para impressão
 - Escalonamento de processos em um sistema operacional
 - Fila de requisições de acesso a disco
 - Fila de espera por recursos
 - Fila (*buffering*) de mensagens e pacotes
- Estruturas de Dados
 - Suporte na implementação de algoritmos sobre árvores e grafos
- etc.

Fila: Implementações Possíveis

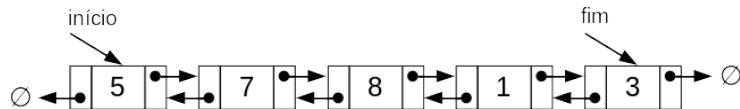
• Arranjo (*buffer*) circular



• Lista Simplesmente Encadeada



• Lista Duplamente Encadeada



Implementação de um Arranjo Circular

- Usam-se índices para inserção (insertPos) e remoção (removePos) que percorrem o arranjo (queue) de forma “circular”
- Para inserir pode-se usar:

```
queue[ insertPos++ ] = e;
insertPos %= maxElements; // Ou: if ( insertPos == maxElements ) insertPos = 0;
```

- Para remover pode-se usar:

```
e = queue[ removePos++ ];
removePos %= maxElements; // Ou: if ( removePos == maxElements ) removePos = 0;
```

- Deve-se verificar as situações de arranjo cheio (na inserção) e arranjo vazio (na remoção)
- Exemplos:

0	1	2	3	4	5	6	7	8	9
		5	7	8	1	3			

tam = 5 / tamMax = 10 / posInício = 2 / posFim = 6

0	1	2	3	4	5	6	7	8	9
1	3						5	7	8

tam = 5 / tamMax = 10 / PosInício = 7 / posFim = 1

Métodos do TAD Fila (*Queue*)

- `bool enqueue(e)`: insere o elemento no final da fila (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool dequeue(&e)`: remove e retorna (por referência) o elemento do início da fila (retorna `true`, em caso de sucesso, ou `false`, a fila estiver vazia)
- `bool head(&e)` ou `bool front(&e)`: retorna (por referência) o elemento do início da fila, mas não o remove da fila (retorna `true`, em caso de sucesso, ou `false`, a fila estiver vazia)
- `int size()`: retorna o número de elementos da fila
- `int maxSize()`: retorna o número máximo de elementos suportado pela fila
- `bool isEmpty()`: retorna `true`, se a fila estiver vazia, ou `false`, em caso contrário
- `bool isFull()`: retorna `true`, se a fila estiver cheia, ou `false`, em caso contrário
- `void clear()`: esvazia a fila

Exemplo de Implementação: IntQueue.hpp

```
#ifndef _INTQUEUE_HPP
#define _INTQUEUE_HPP

#include <string>

using namespace std;

class IntQueue {
private:
    int numElements, maxElements;
    int insertPos, removePos;
    int *queue;
public:
    IntQueue(int mxSz = 10);
    ~IntQueue();
    int size() const;
    int maxSize() const;
    bool isEmpty() const;
    bool isFull() const;
    void clear();
    bool enqueue(const int e);
    bool dequeue(int &e);
    bool head(int &e) const;
    string str() const; // APENAS PARA DEPURACAO
};

#endif
```

Exemplo de Implementação: IntQueue.cpp

```

#include <sstream>
#include "IntQueue.hpp"

IntQueue::IntQueue(int mxSz) {
    numElements = insertPos = removePos = 0;    maxElements =  ( mxSz < 1 ) ? 10 : mxSz;    queue = new int[maxElements];
}

IntQueue::~IntQueue() { delete[] queue; }
int IntQueue::size() const { return numElements; }
int IntQueue::maxSize() const { return maxElements; }
bool IntQueue::isEmpty() const { return numElements == 0; }
bool IntQueue::isFull() const { return numElements == maxElements; }
void IntQueue::clear() { numElements = insertPos = removePos = 0; }

bool IntQueue::enqueue(const int e) {
    if ( numElements == maxElements ) return false;
    else { queue[ insertPos++ ] = e;    insertPos %= maxElements;    ++numElements;    return true; }
}

bool IntQueue::dequeue(int &e) {
    if ( numElements == 0 ) return false;
    else { e = queue[ removePos++ ];    removePos %= maxElements;    --numElements;    return true; }
}

bool IntQueue::head(int &e) const {
    if ( numElements < 1 ) return false;
    else { e = queue[ removePos ];    return true; }
}

string IntQueue::str() const {
    stringstream ss;    ss << "|";
    for (int i=0; i<maxElements; ++i)
        if ( (removePos == insertPos && numElements != 0) ||
            (removePos < insertPos && (i >= removePos && i < insertPos)) ||
            (removePos > insertPos && (i >= removePos || i < insertPos)) ) ss << queue[i] << "|";
        else ss << "␣|";
    return ss.str();
}

```

Exemplo de Implementação: IntQueueMain.cpp

```

#include <iostream>
#include "IntQueue.hpp"

using namespace std;

void print(IntQueue &queue) {
    cout << "uu" << queue.str() << "uusize=" << queue.size() << "/" << queue.maxSize() << "uuhead=";
    int h;    bool res = queue.head(h);
    if (res) cout << h; else cout << "X";
    cout << "uuisEmpty=" << queue.isEmpty() << "uuisFull=" << queue.isFull() << endl;
}

int main() {
    int e;
    bool res;
    cout << "IntQueue(4):uuuu"; IntQueue queue(4); print(queue);
    e = 1; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    e = 2; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    e = 3; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    e = 4; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    e = 5; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    e = 6; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(X):u"; cout << (res?"OKuu":"ERRO"); print(queue);
    e = 7; cout << "enqueue(" << e << "):u"; res = queue.enqueue(e); cout << (res?"OKuu":"ERRO"); print(queue);
    cout << "clear():uuuuOKuu"; queue.clear(); print(queue);
    return 0;
}

```

Exemplo de Implementação (Saída)

IntQueue(4):		size=0/4	head=X	isEmpty=1	isFull=0
enqueue(1): OK	1	size=1/4	head=1	isEmpty=0	isFull=0
enqueue(2): OK	1 2	size=2/4	head=1	isEmpty=0	isFull=0
enqueue(3): OK	1 2 3	size=3/4	head=1	isEmpty=0	isFull=0
dequeue(1): OK	2 3	size=2/4	head=2	isEmpty=0	isFull=0
enqueue(4): OK	2 3 4	size=3/4	head=2	isEmpty=0	isFull=0
enqueue(5): OK	5 2 3 4	size=4/4	head=2	isEmpty=0	isFull=1
enqueue(6): ERRO	5 2 3 4	size=4/4	head=2	isEmpty=0	isFull=1
dequeue(2): OK	5 3 4	size=3/4	head=3	isEmpty=0	isFull=0
dequeue(3): OK	5 4	size=2/4	head=4	isEmpty=0	isFull=0
dequeue(4): OK	5	size=1/4	head=5	isEmpty=0	isFull=0
dequeue(5): OK		size=0/4	head=X	isEmpty=1	isFull=0
dequeue(X): ERRO		size=0/4	head=X	isEmpty=1	isFull=0
enqueue(7): OK	7	size=1/4	head=7	isEmpty=0	isFull=0
clear(): OK		size=0/4	head=X	isEmpty=1	isFull=0

Exercícios

Exercício 9

- 9 Considere duas estrutura de dados do tipo fila, chamadas A e B. Na fila A, foram inseridos (nessa ordem) os seguintes valores: 10, 20 e 30. E, na fila B, foram inseridos (nessa ordem) os seguintes valores: 30, 20 e 10. Para ambas as estruturas, considere as seguintes operações:

- `dequeue(F)`: que remove um elemento da fila F e retorna esse elemento;
- `enqueue(F, E)`: que insere o elemento E na fila F;
- `head(F)`: que retorna o elemento do início da fila, sem removê-lo da estrutura.

Quais serão as sequências de elementos nas filas A e B, após executar a expressão “`enqueue(A, dequeue(A) + dequeue(B) + head(A))`”?

Exercício 10

- 10 Usando como ponto de partida a classe `IntQueue`, apresentada anteriormente e implementada usando alocação sequencial ou contígua, implemente a classe `IntLinkedListQueue`, que funciona da mesma forma, porém usando alocação encadeada.

Observações:

- A versão usando alocação encadeada eliminará a necessidade de se trabalhar com um limite máximo de elementos, consequentemente, os métodos `maxSize()` e `isFull()` NÃO farão parte da nova implementação.
- A definição da classe (arquivo `IntLinkedListQueue.hpp`) e um programa de teste (arquivo `IntLinkedListQueueMain.cpp`) estão listados nas lâminas a seguir.

Exercício 10: IntLinkedListQueue.hpp

```
#ifndef _INTLINKEDQUEUE_HPP
#define _INTLINKEDQUEUE_HPP
#include <string>
using namespace std;

class IntLinkedListQueue {
private:
    int numElements;
    struct Node {
        int data;
        Node *next;
        Node(int d) { data = d; next = nullptr; }
    };
    Node *queueHead, *queueTail;
public:
    IntLinkedListQueue();
    ~IntLinkedListQueue();
    int size() const;
    bool isEmpty() const;
    void enqueue(const int e);
    bool dequeue(int &e);
    bool head(int &e) const;
    void clear();
    string str() const; // APENAS PARA DEPURACAO
};
#endif
```


Exercício 10: IntLinkedListQueueMain.cpp

```

#include <iostream>
#include "IntLinkedListQueue.hpp"

using namespace std;

void print(IntLinkedListQueue &queue) {
    cout << "uu" << queue.str() << "uu size=" << queue.size() << "uu head=";
    int h;    bool res = queue.head(h);
    if (res) cout << h; else cout << "X";
    cout << "uu isEmpty=" << queue.isEmpty() << endl;
}

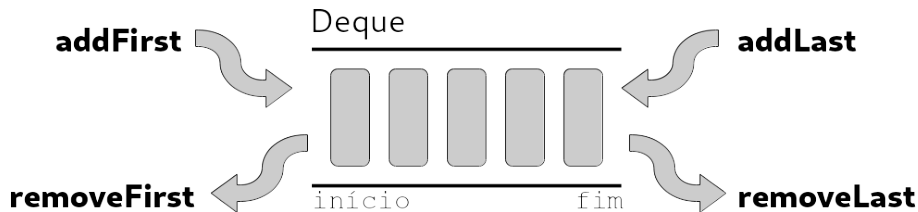
int main() {
    int e;
    bool res;
    cout << "IntLinkedListQueue: u"; IntLinkedListQueue queue; print(queue);
    e = 1; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    e = 2; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    e = 3; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    e = 4; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    e = 5; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    e = 6; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(" << e << "): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    res = queue.dequeue(e); cout << "dequeue(X): u"; cout << (res?"OK uu":"ERRO"); print(queue);
    e = 7; cout << "enqueue(" << e << "): u"; queue.enqueue(e); cout << "OK uu"; print(queue);
    cout << "clear(): uuuu OK uu"; queue.clear(); print(queue);
    return 0;
}

```

Deque

Deque

- É uma abreviação de *Double-Ended Queue*
- Trata-se de uma estrutura linear de dados, um pouco mais flexível do que pilhas ou filas, que permite inserções e remoções tanto no início quanto no final
- Possui, portanto, duas pontas (frente e traseira ou front e back), sendo possível selecionar qual será utilizada tanto para inserção quanto para remoção



Deque: Aplicações e Implementações

- É usado em aplicações onde, por exemplo:
 - Um item é removido da fila e por alguma razão precisa ser reinserido na posição que ocupava anteriormente
 - O último item da fila “desiste” de permanecer nela (por exemplo, devido à demora ou ao tamanho da fila)
- Pode ser implementado da mesma forma que uma fila:
 - Arranjo (*buffer*) circular
 - Lista Simplesmente Encadeada
 - Lista Duplamente Encadeada
- Há implementações bloqueantes, em que a primitiva de inserção fica bloqueada até haver espaço e a primitiva de remoção fica bloqueada até que exista um elemento para ser removido

Métodos do TAD Deque

- `bool addFirst(e)`: insere o elemento no início do deque (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool addLast(e)`: insere o elemento no fim do deque (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool removeFirst(&e)`: remove e retorna (por referência) o elemento do início do deque (retorna `true`, em caso de sucesso, ou `false`, o deque estiver vazio)
- `bool removeLast(&e)`: remove e retorna (por referência) o elemento do fim do deque (retorna `true`, em caso de sucesso, ou `false`, o deque estiver vazio)
- `bool first(&e)` ou `bool head(&e)` ou `bool front(&e)`: retorna (por referência) o elemento do início do deque, mas não o remove do deque (retorna `true`, em caso de sucesso, ou `false`, o deque estiver vazio)
- `bool last(&e)` ou `bool tail(&e)` ou `bool back(&e)`: retorna (por referência) o elemento do fim do deque, mas não o remove do deque (retorna `true`, em caso de sucesso, ou `false`, o deque estiver vazio)
- `int size()`: retorna o número de elementos do deque
- `int maxSize()`: retorna o número máximo de elementos suportado pelo deque
- `bool isEmpty()`: retorna `true`, se o deque estiver vazio, ou `false`, em caso contrário
- `bool isFull()`: retorna `true`, se o deque estiver cheio, ou `false`, em caso contrário
- `void clear()`: esvazia o deque

Exercícios

Exercício 11

11 Considere as seguintes declarações para uma estrutura linear duplamente encadeada do tipo deque:

```
struct Node {
    char info; Node *prev, *next;
    Node(char i) { info = i; prev = next = nullptr; }
};
Node *esquerda = nullptr, *direita = nullptr;
```

E determine o estado final, na ordem correta, dos ponteiros e dos nodos dessa estrutura, depois da execução das seguintes operações:

- Inserir as letras 'D', 'E', 'S', 'C', 'A', 'R', 'T', 'E' e 'S' (nesta ordem), cada uma em um nodo da lista, pela direita;
- Remover 3 letras do deque pela esquerda;
- Remover 4 letras do deque pela direita;
- Inserir as letras 'E', 'D', 'I', 'S', 'O' e 'N' (nesta ordem), cada uma em um nodo da lista, pela esquerda;
- Remover 5 letras do deque pela esquerda;
- Inserir as letras 'R', 'U', 'T', 'H', 'E', 'R', 'F', 'O', 'R' e 'D' (nesta ordem), cada uma em um nodo da lista, pela esquerda;
- Remover 8 letras do deque pela esquerda;
- Inserir as letras 'E', 'I', 'N', 'S', 'T', 'E', 'I' e 'N' (nesta ordem), cada uma em um nodo da lista, pela esquerda;
- Remover 7 letras do deque pela esquerda.

Exercício 12

- 12 Usando como ponto de partida os códigos apresentados e desenvolvidos anteriormente, implemente a classe `IntDoubleLinkedDeque`, que implementa um deque com uma lista duplamente encadeada.

Observações:

- Lembre-se que **NÃO** haverá necessidade de trabalhar com um limite máximo de elementos, consequentemente, os métodos `maxSize()` e `isFull()` **NÃO** farão parte da nova implementação.
- A definição da classe (arquivo `IntDoubleLinkedDeque.hpp`) e um programa de teste (arquivo `IntDoubleLinkedDequeMain.cpp`) estão listados nas lâminas a seguir.

Exercício 12: IntDoubleLinkedListDeque.hpp

```
#ifndef _INTDOUBLELINKEDDEQUE_HPP
#define _INTDOUBLELINKEDDEQUE_HPP
#include <string>
using namespace std;

class IntDoubleLinkedListDeque {
private:
    int numElements;
    struct Node {
        int data;
        Node *prev, *next;
        Node(int d) { data = d; prev = next = nullptr; }
    };
    Node *front, *back;
public:
    IntDoubleLinkedListDeque();
    ~IntDoubleLinkedListDeque();
    int size() const;
    bool isEmpty() const;
    void addFirst(const int e);    bool removeFirst(int &e);    bool first(int &e) const;
    void addLast(const int e);    bool removeLast(int &e);    bool last(int &e) const;
    void clear();
    string str() const;           // APENAS PARA DEPURACAO
    string reverseStr() const;    // APENAS PARA DEPURACAO
};
#endif
```

Exercício 12: IntDoubleLinkedListDequeMain.cpp

```

#include <iostream>
#include "IntDoubleLinkedListDeque.hpp"

using namespace std;

void print(IntDoubleLinkedListDeque &deque) {
    int h; bool res; cout << "uu" << deque.str() << "uu size=" << deque.size();
    cout << "uu first="; res = deque.first(h); if (res) cout << h; else cout << "X";
    cout << "uu last="; res = deque.last(h); if (res) cout << h; else cout << "X";
    cout << "uu isEmpty=" << deque.isEmpty() << "uu reverse=" << deque.reverseStr() << endl;
}

int main() {
    int e; bool res;
    cout << "IntDoubleLinkedListDeque:u"; IntDoubleLinkedListDeque deque; print(deque);
    e = 6; cout << "addFirst(" << e << "):uuuuOKuuuu"; deque.addFirst(e); print(deque);
    e = 7; cout << "addLast(" << e << "):uuuuuOKuuuu"; deque.addLast(e); print(deque);
    e = 5; cout << "addFirst(" << e << "):uuuuuOKuuuu"; deque.addFirst(e); print(deque);
    e = 8; cout << "addLast(" << e << "):uuuuuOKuuuu"; deque.addLast(e); print(deque);
    e = 4; cout << "addFirst(" << e << "):uuuuuOKuuuu"; deque.addFirst(e); print(deque);
    e = 9; cout << "addLast(" << e << "):uuuuuOKuuuu"; deque.addLast(e); print(deque);
    res = deque.removeFirst(e); cout << "removeFirst(" << e << "):u" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeLast(e); cout << "removeLast(" << e << "):uu" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeLast(e); cout << "removeLast(" << e << "):uu" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeFirst(e); cout << "removeFirst(" << e << "):u" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeLast(e); cout << "removeLast(" << e << "):uu" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeFirst(e); cout << "removeFirst(" << e << "):u" << (res?"OKuuuu":"ERROuu"); print(deque);
    res = deque.removeLast(e); cout << "removeLast(X):uu" << (res?"OKuuuu":"ERROuu"); print(deque);
    e = 2; cout << "addLast(" << e << "):uuuuuOKuuuu"; deque.addLast(e); print(deque);
    cout << "clear():uuuuuuuuOKuuuu"; deque.clear(); print(deque);
    return 0;
}

```

Lista

Lista

- Uma lista é uma estrutura de dados que agrupa informações referentes a um conjunto de elementos relacionados entre si
- Há uma ordem lógica entre os elementos, que não corresponde necessariamente à ordem física
- A inserção e exclusão em uma lista é menos restritiva do que em pilhas, filas ou dequeues
- Isto significa que uma implementação de lista poderia ser usada para criar e gerenciar pilhas, filas e dequeues

Operações sobre Listas

- Algumas operações básicas sobre listas, envolvendo elementos (nodos), são:
 - Inserção (no início, no fim ou em uma posição específica)
 - Remoção (no início, no fim ou em uma posição específica)
 - Busca e acesso (através de índice ou através da informação de algum campo)
 - Alteração
- Também é comum aplicar operações sobre toda a lista, tais como:
 - Combinação de duas ou mais listas em uma única lista (concatenação ou *merge*)
 - Ordenação da lista segundo determinado critério

Opções de Implementação

- Usando arranjos
 - Maior desempenho no acesso
 - Apresentam maior dificuldade de inserção e remoção no início ou no meio
- Usando estruturas encadeadas
 - Menor desempenho no acesso
 - Muito mais flexível para inserir e remover nodos
 - Podem ser: simplesmente encadeadas, duplamente encadeadas, simplesmente encadeadas e circulares, duplamente encadeadas e circulares

Lista com Arranjo

Lista com Arranjo

- Consiste em um número fixo de posições **contíguas** e para o armazenamento de elementos do mesmo tipo
- Possui acesso direto facilitado e rápido, ou seja, o tempo de acesso é constante para qualquer elemento
- A inserção e a remoção de nodos no início ou no meio do arranjo tem alto custo, pois requer a movimentação de nodos ou para abrir espaço ou para evitar espaços não utilizados
- Geralmente é preciso definir o número máximo de elementos que serão armazenados (mudar esse tamanho pode ser possível, mas será custoso)

0	1	2	3	4	5	6	7	8	9
4	3	2	1						

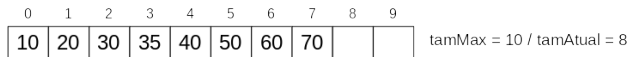
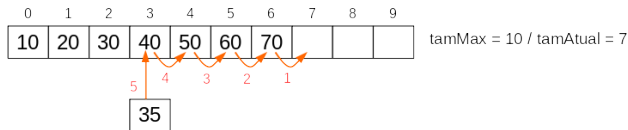
tamMax = 10 / tamAtual = 4

Métodos para um TAD Lista com Arranjo

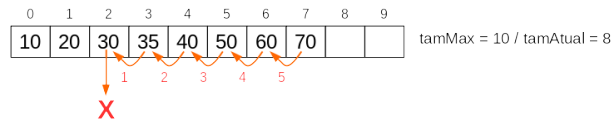
- `bool add(e)`: insere o elemento no final da lista (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool add(index,e)`: insere o elemento em um índice específico da lista (retorna `true`, em caso de sucesso, ou `false`, se NÃO houver espaço)
- `bool get(index, &e)`: retorna (por referência) o elemento do índice especificado (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `bool set(index, e)`: atribui o elemento para a posição do índice especificado (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `bool remove(index)`: remove o elemento do índice especificado da lista (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `int size()`: retorna o número de elementos da lista
- `int maxSize()`: retorna o número máximo de elementos suportado pela lista
- `bool isEmpty()`: retorna `true`, se a lista estiver vazia, ou `false`, em caso contrário
- `bool isFull()`: retorna `true`, se a lista estiver cheia, ou `false`, em caso contrário
- `bool contains(e)`: retorna `true`, se o elemento existir na lista, ou `false`, em caso contrário
- `int indexOf(e)`: retorna o índice da primeira ocorrência do elemento na lista, se o elemento existir na lista, ou -1 em caso contrário
- `int indexOf(pos,e)`: retorna o índice da próxima ocorrência do elemento na lista a partir da posição especificada, se o elemento existir, ou -1 em caso contrário
- `void clear()`: esvazia a lista

Lista com Arranjo: Inserção e Exclusão

- Inserção: `arrayList->add(3,35)`



- Exclusão: `arrayList->remove(2)`



Exercício

Exercício 13

- 12 Usando como base a descrição dos métodos de um TAD Lista com Arranjo, implemente os métodos da classe `StringArrayList`, que implementa uma lista com um arranjo de tamanho predefinido. O arquivo de cabeçalho para esta classe (`StringArrayList.hpp`) e um programa de teste (`StringArrayListMain.cpp`).

Exercício 13: StringArrayList.hpp

```
#ifndef _STRINGARRAYLIST_HPP
#define _STRINGARRAYLIST_HPP

#include <string>

using namespace std;

class StringArrayList {
private:
    int numElements;
    int maxElements;
    string *list;
public:
    StringArrayList(int mxSz = 10);
    ~StringArrayList();
    void clear();
    int size() const;
    int maxSize() const;
    bool isEmpty() const;
    bool isFull() const;
    bool add(const string &s);
    bool add(const int index, const string &s);
    bool remove(const int index);
    bool get(const int index, string &s);
    bool set(const int index, const string &s);
    bool contains(const string &s);
    int indexOf(const string &s);
    int indexOf(int index, const string &s);
    string str() const;
};

#endif
```

Exercício 13: StringArrayListMain.cpp

```

#include <iostream>
#include "StringArrayList.hpp"

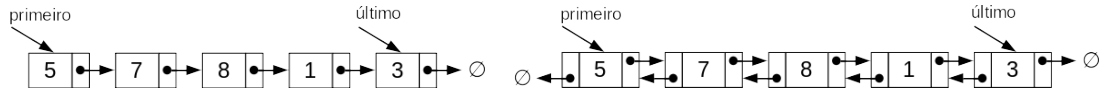
int main() {
    StringArrayList *l = new StringArrayList(28);
    if ( l->maxSize() != 28 || !l->isEmpty() ) cerr << "ERR001" << endl;
    string saudacao[] = { "Bom_","dia,","pessoal!", "Bom_","dia,","Bom_"};
    int tamSaudacao = sizeof(saudacao) / sizeof(string);
    l->add(saudacao[1]); l->add(saudacao[2]); l->add(0,saudacao[0]); l->add(saudacao[3]); l->add(saudacao[4]); l->add(saudacao[5]);
    if ( l->contains("teste") ) cerr << "ERR002" << endl;
    if ( !l->contains(saudacao[0]) || !l->contains(saudacao[1]) || !l->contains(saudacao[2]) ) cerr << "ERR003" << endl;
    for (int i=0; i<l->size(); ++i) if ( l->indexOf(i,saudacao[i]) != i ) cerr << "ERR004" << endl;
    if ( l->indexOf(saudacao[0])!=0 || l->indexOf(saudacao[1])!=1 || l->indexOf(saudacao[2])!=2 ) cerr << "ERR005" << endl;
    l->remove( l->size()-1 ); l->remove( l->size()-1 ); l->remove( l->size()-1 );
    cout << l->str() << endl;
    if ( l->size() != 3 || l->isFull() || l->isEmpty() ) cerr << "ERR006" << endl;
    for (int i=0; i<l->size(); ++i) { string s; bool res = l->get(i,s); if ( !res || s != saudacao[i] ) cerr << "ERR007" << endl; }
    l->add("dinâmica."); l->remove(1); l->remove(0); l->remove(0);
    l->add(0,"Lista_"); l->add(1,"ligada_"); l->add(2,"ê_"); l->add(3,"uma_"); l->add(4,"estrutura_");
    l->add(5,"de_"); l->add(6,"dados_"); l->add(7,"estranha_"); l->add(8,"e_"); l->remove(7);
    l->add(7,"linear_"); l->add(2,"ou_"); l->add(3,"lista_"); l->add(4,"encadeada_");
    cout << l->str() << endl;
    if ( l->size() != 13 || l->isFull() || l->isEmpty() ) cerr << "ERR008" << endl;
    l->clear();
    if ( l->str() != "" || l->size() != 0 || l->maxSize() != 28 || !l->isEmpty() ) cerr << "ERR009" << endl;
    string palavras[] = { "apontam_","que_","referências_","duas_","ou_","uma_","também_","e_","dados_","contém_","nodo_","cada_","que_","sendo_","nodos_","de_","sequência_","uma_","por_","composta_","ê_","Ela_"};
    int tamPalavras = sizeof(palavras) / sizeof(string);
    for (int i=0; i<tamPalavras; ++i) l->add("?");
    for (int i=tamPalavras-1; i>0; --i) l->set(tamPalavras-1-i,palavras[i]);
    if ( l->size() != 22 || l->isFull() || l->isEmpty() ) cerr << "ERR010" << endl;
    l->add("anterior_"); l->add("ou_"); l->add("posterior."); l->add(22,"para_"); l->add(23,"o_"); l->add(24,"nodo_");
    cout << l->str() << endl;
    if ( l->size() != 28 || !l->isFull() || l->isEmpty() || l->add(".") || l->add(0, ".") ) cerr << "ERR011" << endl;
    delete l;
    return 0;
}

```

Lista Encadeada

Lista Encadeada

- Armazena elementos do mesmo tipo em uma estrutura formada por nodos (não necessariamente contíguos na memória), cada nodo contendo uma referência para o nodo seguinte (lista simplesmente encadeada) e eventualmente também uma referência para o nodo anterior (lista duplamente encadeada)



- Para acessar determinado elemento, será preciso percorrer a lista...
- No entanto, inserir elementos na lista ou removê-los não exigirá deslocamentos de nodos, apenas ajustes em algumas poucas referências!

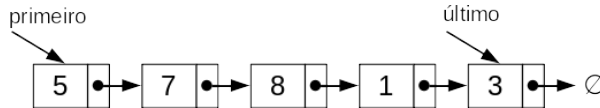
Métodos para um TAD Lista Encadeada

- `int size()`: retorna o número de elementos da lista
- `bool isEmpty()`: retorna `true`, se a lista estiver vazia, ou `false`, em caso contrário
- `void clear()`: esvazia a lista
- `void push_front(e)`: insere o elemento no início da lista
- `void push_back(e)`: insere o elemento no final da lista
- `bool insert(index, e)`: insere o elemento no índice especificado da lista
- `bool pop_front()`: remove o elemento do início da lista (retorna `false` se a lista estiver vazia)
- `bool pop_back()`: remove o elemento do final da lista (retorna `false` se a lista estiver vazia)
- `bool remove(index)`: remove o elemento do índice especificado da lista (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `bool get(index, &e)`: retorna (por referência) o elemento do índice especificado (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `bool set(index, e)`: atribui o elemento para a posição do índice especificado (retorna `true`, em caso de sucesso, ou `false`, se o índice for inválido)
- `bool contains(e)`: retorna `true`, se o elemento existir na lista, ou `false`, em caso contrário
- `int indexOf(e)`: retorna o índice da primeira ocorrência do elemento na lista, ou `-1` se o elemento não existir
- `int indexOf(pos, e)`: retorna o índice da próxima ocorrência do elemento na lista a partir da posição especificada, ou `-1` se o elemento não existir a partir dessa posição

Lista Simplesmente Encadeada

Lista Simplesmente Encadeada

- É uma estrutura encadeada onde cada nodo, além da informação, guarda apenas uma referência para o próximo nodo



- Em C++, usando struct, seria possível declarar, por exemplo, uma lista simplesmente encadeada de valores inteiros usando:

```

struct Node {
    int info;
    Node *next;
    Node(int i) { info = i; next = nullptr; }
};
  
```

- Pode-se trabalhar com dois ponteiros, um para o início (por exemplo, head) e outro (opcional, mas desejável) para o fim da lista (por exemplo, tail)

```

Node *head = nullptr, *tail = nullptr;
  
```

Lista Simplesmente Encadeada

- Inserir ou remover em uma lista simplesmente encadeada exige localizar o elemento anterior, o que pode exigir percorrer a lista a partir do início
 - Inserir ou remover no meio de uma lista duplamente encadeada é mais simples
- Para inserir em uma lista encadeada é preciso: alocar um novo nodo, inserir as informações nele e encadeá-lo na lista
- Para remover de uma lista encadeada é preciso: recuperar a informação (se necessário), ajustar os encadeamentos da lista e desalocar o nodo
- Operações em listas simplesmente encadeadas
 - Inserção no início: simples
 - Inserção no meio: exige referência para o elemento anterior
 - Inserção no final: simples
 - Remoção do início: simples
 - Remoção do meio: exige referência para o elemento anterior
 - Remoção do final: exige referência para o elemento anterior

Lista Simplesmente Encadeada: Inserção

- Inserção no início

```
Node *node = new Node(info);  
node->next = head;  
head = node;  
if ( tail == nullptr ) tail = node;
```

- Inserção no meio

```
// aux aponta para nodo antes do qual se quer inserir  
// ant aponta para nodo anterior a aux  
Node *node = new Node(info);  
ant->next = node;  
node->next = aux;
```

- Inserção no final

```
Node *node = new Node(info);  
tail->next = node;  
tail = node;
```

Lista Simplesmente Encadeada: Remoção

- Remoção do início

```
if ( head != nullptr ) {  
    Node *aux = head;  
    head = head->next;  
    if ( head == nullptr ) tail = nullptr;  
    delete aux;  
}
```

- Remoção do meio

```
// aux aponta para nodo que se quer remover  
// ant aponta para nodo anterior a aux  
ant->next = aux->next;  
if ( aux->next == nullptr ) tail = ant;  
delete aux;
```

- Remoção do final

```
// ant aponta para nodo anterior a tail  
ant->next = nullptr;  
delete tail;  
tail = ant;
```

Exercícios

Exercício 14

- 14 Considere o código abaixo, que cria uma lista simplesmente encadeada formada por 4 nodos, respectivamente, com as letras 'B', 'C', 'E' e 'F' como conteúdo. Modifique este código para inserir na lista, **no ponto indicado no código**, um nodo com a letra 'D', na posição correta da lista, mantendo-se a ordem alfabética crescente. Depois, teste o seu código (e adapte-o se necessário) para inserir, na posição correta, tanto a letra 'A' quanto a letra 'G'.

```
#include <iostream>
using namespace std;
struct Node {
    char info; Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-Node(" << info << ")_destruido..." << endl; }
};

int main() {
    Node *head = nullptr, *tail = nullptr; // Criação
    for (char l='B'; l<='F'; l++) {
        if (l == 'D') continue; // Evita que o Node com a info 'D' seja inserido na lista...
        Node *aux = new Node(l);
        if ( tail == nullptr ) { head = tail = aux; }
        else { tail->next = aux; tail = aux; }
    }

    // Coloque a solução aqui!

    cout << "head-->_"; // Exibição
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << "|" << aux->info << (aux->next==nullptr?"|X|_":"||_>_");
    cout << "_<--tail" << endl;
    while ( head != nullptr ) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}
```


Exercício 15

- 15 Considere o código abaixo, que cria uma lista simplesmente encadeada formada por 5 nodos, respectivamente, com as letras 'A', 'D', 'C', 'B' e 'E' como conteúdo. Modifique este código para trocar, **no ponto indicado no código**, a posição dos nodos que contêm 'D' e 'B' na lista.

```
#include <iostream>
using namespace std;
struct Node {
    char info; Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    string s = "ADCBE"; Node *head = nullptr, *tail = nullptr; // Criação
    for (int i=0; i<s.length(); ++i) {
        Node *aux = new Node( s[i] );
        if ( tail == nullptr ) { head = tail = aux; }
        else { tail->next = aux; tail = aux; }
    }

    // Coloque a solução aqui!

    cout << "head-->_"; // Exibição
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << "|" << aux->info << (aux->next==nullptr?"|X|_":"|_|_>_");
    cout << "_<--tail" << endl;
    while (head != nullptr) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}
```

Exercício 16

16 Implemente uma aplicação para gerenciar uma **lista simplesmente encadeada** em que os nós armazenam uma letra (`char`) e um ponteiro para o próximo elemento. A aplicação deve permitir interativamente a inserção de letras na lista a partir de um pequeno conjunto de operações fornecidas pela entrada padrão (terminal). As operações e seus parâmetros são indicadas pelos seguintes caracteres:

- '`<`' (menor) e letra: insere a letra especificada no início da lista;
- '`>`' (maior) e letra: insere a letra especificada no fim da lista;
- '`+`' (mais), letra e índice: insere a letra especificada no índice especificado da lista;
- '`{`' (abre-chaves): remove a letra/nodo do início da lista (imprime "ERRO" se a lista estiver vazia);
- '`}`' (fecha-chaves): remove a letra/nodo do final da lista (imprime "ERRO" se a lista estiver vazia);
- '`-`' (menos) e índice: remove o elemento no índice especificado (imprime "ERRO" se o índice for inválido);
- '`.`' (ponto): encerra a aplicação.

No laço principal da sua aplicação, sempre antes de ler a especificação de uma operação, mostre o conteúdo da lista. Use o arquivo `exercicio16.input` (dentro do arquivo `src.zip`) como entrada para testar o seu programa.

Exercício 16: exercicio16-template.cpp

```

#include <iostream>

using namespace std;

struct Node {
    char info; Node *next;
    Node(char i) { info = i; next = nullptr; }
};

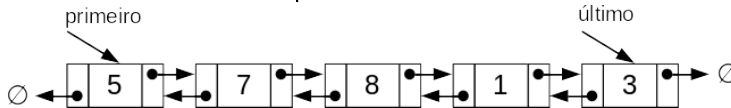
int main() {
    Node *head = nullptr, *tail = nullptr, *aux; char c; bool fim = false; unsigned pos;
    cout << "'<'_PUSH_FRONT'>'>'_PUSH_BACK'>'{'_POP_FRONT'>'}'_POP_BACK'>'+'_INSERT'>'-'_REMOVE'>'.'_QUIT" << endl;
    while (!fim) {
        aux = head; // Mostra a lista
        while (aux != nullptr) { cout << "|" << aux->info; aux = aux->next; }
        cout << "|" << endl;
        cin >> c;
        switch(c) {
            case '<': cin >> c; /* ... */ break;
            case '>': cin >> c; /* ... */ break;
            case '+': cin >> c >> pos; /* ... */ break;
            case '}': /* ... */ break;
            case '{': /* ... */ break;
            case '-': cin >> pos; /* ... */ break;
            case '.': fim = true; break;
        }
    }
    while (head != nullptr) { aux = head; head = head->next; delete aux; } // Desaloca a lista
    return 0;
}

```

Lista Duplamente Encadeada

Lista Duplamente Encadeada

- É uma estrutura encadeada onde cada nodo, além da informação, guarda uma referência para o próximo nodo e uma referência para o nodo anterior



- Em C++, usando struct, seria possível declarar, por exemplo, uma lista simplesmente encadeada de valores inteiros usando:

```
struct Node {
    int info;
    Node *prev, *next;
    Node(int i) { info = i; prev = next = nullptr; }
};
```

- Também são usados dois ponteiros, um para o início (por exemplo, head) e outro (opcional, mas desejável) para o fim da lista (por exemplo, tail)

```
Node *head = nullptr, *tail = nullptr;
```

Lista Duplamente Encadeada

- Para inserir ou remover em uma lista duplamente encadeada basta ter a posição de inserção ou remoção, NÃO sendo necessário percorrer a lista a partir do início
 - É mais fácil inserir ou remover em uma lista duplamente encadeada do que em uma lista simplesmente encadeada
- Para inserir em uma lista encadeada é preciso: alocar um novo nodo, inserir as informações nele e encadeá-lo na lista
- Para remover de uma lista encadeada é preciso: recuperar a informação (se necessário), ajustar os encadeamentos da lista e desalocar o nodo

Lista Duplamente Encadeada: Inserção

- Inserção no início

```
Node *node = new Node(info);  
if ( head == nullptr ) { head = tail = node; }  
else {  
    node->next = head;  
    head->prev = node;  
    head = node;  
}
```

- Inserção no meio

```
// aux aponta para nodo antes do qual se quer inserir  
Node *node = new Node(info);  
node->prev = aux->prev;  
node->next = aux;  
(aux->prev)->next = node;  
aux->prev = node;
```

- Inserção no final

```
Node *node = new Node(info);  
node->prev = tail;  
tail->next = node;  
tail = node;
```

Lista Duplamente Encadeada: Remoção

- Remoção do início

```
if ( head != nullptr ) {  
    Node *aux = head;  
    head = head->next;  
    if ( head == nullptr ) tail = nullptr;  
    else head->prev = nullptr;  
    delete aux;  
}
```

- Remoção do meio

```
// aux aponta para nodo que se quer remover  
(aux->prev)->next = aux->next;  
if ( aux->next == nullptr ) tail = aux->prev;  
else (aux->next)->prev = aux->prev;  
delete aux;
```

- Remoção do final

```
Node *aux = tail;  
tail = tail->prev;  
if ( tail == nullptr ) head = nullptr;  
else tail->next = nullptr;  
delete aux;
```


Exercícios

Exercício 17

- 17 Considere uma **lista simplesmente encadeada**, com referências para o início e para o final, cujos nodos foram declarados da seguinte forma:

```
struct Node {
    int info; Node *next;
    Node(int i) { info = i; next = nullptr; }
};
```

Implemente uma função que recebe os ponteiros para o início e para o final da lista (por referência) e inverte a lista. A função deve ter o seguinte protótipo:

```
void reverse(Node **head, Node **tail);
```

- 18 Considere uma **lista duplamente encadeada**, com referências para o início e para o final, cujos nodos foram declarados da seguinte forma:

```
struct Node {
    int info; Node *prev, *next;
    Node(int i) { info = i; prev = next = nullptr; }
};
```

Implemente uma função que recebe os ponteiros para o início e para o final da lista (por referência) e inverte a lista. A função deve ter o seguinte protótipo:

```
void reverse(Node **head, Node **tail);
```

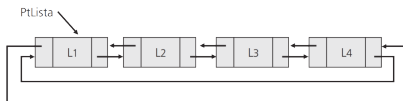
Lista Circular

Lista Circular

- Uma lista circular é um tipo especial de estrutura encadeada onde o último elemento aponta para o primeiro (não usando-se, neste caso, nullptr para indicar o final da lista)
- É possível ter listas circulares:
 - Simplesmente encadeadas: a busca somente ocorre em um sentido

```
struct Node {  
    int info;  Node *next;  
    Node(int i) { info = i;  next = nullptr; }  
};
```

- Duplamente encadeadas: a busca pode ocorrer em qualquer sentido



```
struct Node {  
    int info;  Node *prev, *next;  
    Node(int i) { info = i;  prev = next = nullptr; }  
};
```

Lista Circular

- Observe que nada muda nas declarações em relação às versões não circulares
- Se a lista circular é duplamente encadeada, NÃO há a necessidade de um ponteiro para o final da listas
- Com listas circulares simplesmente encadeadas, um ponteiro para o final da lista é desejável
- Exemplo de aplicação: **Lista de processos prontos para serem executados em um sistema operacional com algoritmo de escalonamento *Round Robin***
 - *Round Robin* é um escalonamento preemptivo que divide uniformemente o tempo da UCP entre todos os processos prontos para a execução
 - Processos são mantidos em uma lista de prontos circular e duplamente encadeada
 - Cada processo tem uma fatia de tempo máxima (*time slice* ou *quantum*) para usar o processador
 - Se o processo termina a sua computação antes do final da fatia de tempo, ele sai da lista
 - Se o processo usa toda a sua fatia de tempo, o ponteiro da lista é apontado para o próximo processo

Créditos

Créditos

- Estas lâminas contêm trechos inspirados em materiais criados e disponibilizados pelos professores Isabel Harb Manssour e Igańska Janiska Weber.

Soluções

Exercício 1: CharStack.hpp

```
#ifndef _CHARSTACK_HPP
#define _CHARSTACK_HPP

#include <string>

using namespace std;

class CharStack {
private:
    int numElements;
    int maxElements;
    char *stack;
public:
    CharStack(int mxSz = 10);
    ~CharStack();
    bool push(const char e);
    bool pop(char &e);
    bool top(char &e) const;
    int size() const;
    int maxSize() const;
    bool isEmpty() const;
    bool isFull() const;
    void clear();
    string str() const; // APENAS PARA DEPURACAO
};

#endif
```

Exercício 1: CharStack.cpp

```
#include <sstream>
#include "CharStack.hpp"

CharStack::CharStack(int mxSz) {
    numElements = 0;    maxElements = ( mxSz < 1 ) ? 10 : mxSz;
    stack = new char[maxElements];
}

CharStack::~CharStack() { delete[] stack; }

bool CharStack::push(const char e) {
    if ( numElements == maxElements ) return false;
    else { stack[ numElements++ ] = e; return true; }
}

bool CharStack::pop(char &e) {
    if ( numElements == 0 ) return false;
    else { e = stack[ --numElements ]; return true; }
}

bool CharStack::top(char &e) const {
    if ( numElements < 1 ) return false;
    else { e = stack[ numElements-1 ]; return true; }
}

int CharStack::size() const { return numElements; }
int CharStack::maxSize() const { return maxElements; }
bool CharStack::isEmpty() const { return numElements == 0; }
bool CharStack::isFull() const { return numElements == maxElements; }
void CharStack::clear() { numElements = 0; }

string CharStack::str() const {
    int i;    stringstream ss;
    ss << "|";
    for (i=0; i<numElements; ++i) ss << stack[i] << "|";
    for (; i<maxElements; ++i) ss << "░|";
    return ss.str();
}
```

Exercício 2: exercicio02.cpp

```

#include <iostream>
#include "CharStack.hpp"

using namespace std;

string invertePalavrasDaFrase(string frase) {
    char ch;
    CharStack stack(frase.length());
    string res = "";
    for (int i=0; i<frase.size(); ++i) {
        char c = frase[i];
        if (c == ' ' || c == '\n') {
            while ( stack.pop(ch) ) res += ch;
            res += c;
        }
        else if ( !stack.push(c) ) cerr << "ERRO!" << endl;
    }
    while ( stack.pop(ch) ) res += ch;
    return res;
}

int main() {
    string frase1 = "ETSE_OICICREXE_E_OTIUM_LICAF.";
    cout << frase1 << endl;
    cout << invertePalavrasDaFrase(frase1) << endl;
    return 0;
}

```

Exercício 3: exercicio03.cpp

```

#include <iostream>
#include "CharStack.hpp"

using namespace std;

CharStack *copia(CharStack &p) {
    char c;
    int tam = p.size(), tamMax = p.maxSize();
    CharStack *res = new CharStack(tamMax);
    CharStack *aux = new CharStack(tam);
    while ( !p.isEmpty() ) { p.pop(c); aux->push(c); }
    while ( !aux->isEmpty() ) { aux->pop(c); p.push(c); res->push(c); }
    delete aux;
    return res;
}

int main() {
    CharStack p1(28), *p2;
    for (int i=0; i<26; ++i) { char c = 'A'+i; p1.push(c); }
    cout << "p1:␣" << p1.str() << "␣" << p1.size() << "/" << p1.maxSize() << endl;
    cout << "Copiando..." << endl;
    p2 = copia(p1);
    cout << "p1:␣" << p1.str() << "␣" << p1.size() << "/" << p1.maxSize() << endl;
    cout << "p2:␣" << p2->str() << "␣" << p2->size() << "/" << p2->maxSize() << endl;
    delete p2;
    return 0;
}

```

Exercício 4: exercicio04.cpp

```

#include <iostream>
#include "CharStack.hpp"

using namespace std;

bool saoIguais(CharStack &p1, CharStack &p2) {
    int tam = p1.size();
    if ( tam != p2.size() ) return false;
    CharStack *aux = new CharStack(tam);
    bool res = true;
    while (tam > 0) {
        char t1, t2; p1.pop(t1); p2.pop(t2);
        if (t1 != t2) { p1.push(t1); p2.push(t2); res = false; break; }
        aux->push(t1);
        --tam;
    }
    while ( !aux->isEmpty() ) { char c; aux->pop(c); p1.push(c); p2.push(c); }
    delete aux;
    return res;
}

int main() {
    CharStack p1(10), p2(15), p3(20), p4(5);
    for (int i=0; i<5; ++i) { char c = 'A'+i; p1.push(c); p2.push(c); p3.push(c); p4.push(c); }
    p1.push('F'); p2.push('G'); p3.push('F');
    for (int i=0; i<3; ++i) { char c = 'H'+i; p1.push(c); p2.push(c); p3.push(c); p4.push(c); }
    cout << "p1:_" << p1.str() << endl << "p2:_" << p2.str() << endl << "p3:_" << p3.str() << endl << "p4:_" << p4.str() << endl;
    cout << "p1==p2?" << (saoIguais(p1,p2)?"S":"N") << endl;
    cout << "p1==p3?" << (saoIguais(p1,p3)?"S":"N") << endl;
    cout << "p1==p4?" << (saoIguais(p1,p4)?"S":"N") << endl;
    cout << "p2==p3?" << (saoIguais(p2,p3)?"S":"N") << endl;
    cout << "p2==p4?" << (saoIguais(p2,p4)?"S":"N") << endl;
    cout << "p3==p4?" << (saoIguais(p3,p4)?"S":"N") << endl;
    cout << "p1:" << p1.str() << endl << "p2:" << p2.str() << endl << "p3:" << p3.str() << endl << "p4:" << p4.str() << endl;
    return 0;
}

```

Exercício 5: exercicio05.cpp

```

#include <iostream>
#include "CharStack.hpp"

using namespace std;

bool ehPalindromo(string s) {
    CharStack p( s.size() );
    char c;
    string sInv = "";
    for (int i=0; i<s.size(); ++i)
        p.push(s[i]);
    while ( !p.isEmpty() ) { p.pop(c); sInv.append(1,c); }
    return s == sInv;
}

int main() {
    string s;
    s = "ABASEDOTETODESABA"; cout << s << " :␣" << (ehPalindromo(s)?"PALINDROMO":"-") << endl;
    s = "LUZAZUL";           cout << s << " :␣" << (ehPalindromo(s)?"PALINDROMO":"-") << endl;
    s = "oloboamaobolo";     cout << s << " :␣" << (ehPalindromo(s)?"PALINDROMO":"-") << endl;
    s = "ABCDED CB";          cout << s << " :␣" << (ehPalindromo(s)?"PALINDROMO":"-") << endl;
    return 0;
}

```

Exercício 6: pilha_simplesmente_encadeada2.cpp

```

#include <iostream>

using namespace std;

struct Node {
    char info;
    Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    Node *topo = nullptr;
    for (char c = 'A'; c <= 'E'; ++c) {
        Node *aux = new Node(c);
        aux->next = topo;
        topo = aux;
    }

    for (Node *aux = topo; aux != nullptr; aux = aux->next)
        cout << aux->info << endl;

    while (topo != nullptr) {
        Node *aux = topo;
        topo = topo->next;
        delete aux;
    }

    return 0;
}

```

Exercício 7: pilha_duplamente_encadeada.cpp

```
#include <iostream>

using namespace std;

struct Node {
    int info;
    Node *prev, *next;
    Node(int i) { info = i; prev = next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    Node *node1 = new Node(1);
    Node *node2 = new Node(2);
    Node *node3 = new Node(3);
    Node *node4 = new Node(4);
    Node *node5 = new Node(5);

    Node *topo = node5;    node5->next = node4;
    node4->prev = node5;    node4->next = node3;
    node3->prev = node4;    node3->next = node2;
    node2->prev = node3;    node2->next = node1;
    node1->prev = node2;    Node *base = node1;

    for (Node *aux = topo; aux != nullptr; aux = aux->next)
        cout << aux->info << endl;
    for (Node *aux = base; aux != nullptr; aux = aux->prev)
        cout << aux->info << endl;

    while (topo != nullptr) {
        Node *aux = topo;
        topo = topo->next;
        delete aux;
    }

    return 0;
}
```


Exercício 8: IntLinkedStack.cpp

```
#include <sstream>
#include "IntLinkedStack.hpp"

IntLinkedStack::IntLinkedStack() { numElements = 0; stack = nullptr; }
IntLinkedStack::~IntLinkedStack() { clear(); }
int IntLinkedStack::size() const { return numElements; }
bool IntLinkedStack::isEmpty() const { return numElements == 0; }

void IntLinkedStack::push(const int e) {
    Node *aux = new Node(e);
    aux->next = stack; stack = aux;
    ++numElements;
}

bool IntLinkedStack::pop(int &e) {
    if ( numElements == 0 ) return false;
    --numElements; e = stack->data;
    Node *aux = stack; stack = stack->next; delete aux;
    return true;
}

bool IntLinkedStack::top(int &e) const {
    if ( numElements == 0 ) return false;
    else { e = stack->data; return true; }
}

void IntLinkedStack::clear() {
    while (stack != nullptr) { Node *aux = stack; stack = stack->next; delete aux; }
    numElements = 0;
}

string IntLinkedStack::str() const {
    stringstream ss;
    ss << "|";
    for (Node *aux = stack; aux != nullptr; aux = aux->next)
        ss << aux->data << "|";
    return ss.str();
}
```

Exercício 9

```
// inicial  
A = { 10, 20, 30 }    B = { 30, 20, 10 }  
  
// após dequeue(A) tem-se "10 + dequeue(B) + head(A)"  
A = { 20, 30 }        B = { 30, 20, 10 }  
  
// após dequeue(B) tem-se "10 + 30 + head(A)"  
A = { 20, 30 }        B = { 20, 10 }  
  
// após head(A) tem-se "10 + 30 + 20 = 60"  
A = { 20, 30 }        B = { 20, 10 }  
  
// resposta final, após "enqueue(A, 60)"  
A = { 20, 30, 60 }    B = { 20, 10 }
```

Exercício 10: IntLinkedListQueue.cpp

```
#include <sstream>
#include "IntLinkedListQueue.hpp"

IntLinkedListQueue::IntLinkedListQueue() { numElements = 0; queueHead = queueTail = nullptr; }
IntLinkedListQueue::~IntLinkedListQueue() { clear(); }
int IntLinkedListQueue::size() const { return numElements; }
bool IntLinkedListQueue::isEmpty() const { return numElements==0; }

void IntLinkedListQueue::enqueue(const int e) {
    Node *aux = new Node(e);
    if ( queueHead == nullptr) { queueHead = queueTail = aux; }
    else { queueTail->next = aux; queueTail = aux; }
    ++numElements;
}

bool IntLinkedListQueue::dequeue(int &e) {
    if ( numElements == 0 ) return false;
    --numElements; e = queueHead->data; Node *aux = queueHead; queueHead = queueHead->next; delete aux;
    return true;
}

bool IntLinkedListQueue::head(int &e) const {
    if ( numElements == 0 ) return false;
    else { e = queueHead->data; return true; }
}

void IntLinkedListQueue::clear() {
    while (queueHead != nullptr) { Node *aux = queueHead; queueHead = queueHead->next; delete aux; }
    numElements = 0;
}

string IntLinkedListQueue::str() const {
    stringstream ss;
    ss << "|";
    for (Node *aux = queueHead; aux != nullptr; aux = aux->next)
        ss << aux->data << "|";
    return ss.str();
}
```

Exercício 11

```
esquerda --> || <-- direita
```

```
esquerda --> |'D'|'E'|'S'|'C'|'A'|'R'|'T'|'E'|'S'| <-- direita
```

```
esquerda --> |'C'|'A'|'R'|'T'|'E'|'S'| <-- direita
```

```
esquerda --> |'C'|'A'| <-- direita
```

```
esquerda --> |'N'|'O'|'S'|'I'|'D'|'E'|'C'|'A'| <-- direita
```

```
esquerda --> |'E'|'C'|'A'| <-- direita
```

```
esquerda --> |'D'|'R'|'O'|'F'|'R'|'E'|'H'|'T'|'U'|'R'|'E'|'C'|'A'| <-- direita
```

```
esquerda --> |'U'|'R'|'E'|'C'|'A'| <-- direita
```

```
esquerda --> |'N'|'I'|'E'|'T'|'S'|'N'|'I'|'E'|'U'|'R'|'E'|'C'|'A'| <-- direita
```

```
esquerda --> |'E'|'U'|'R'|'E'|'C'|'A'| <-- direita
```

Exercício 12: IntDoubleLinkedList.cpp

```
#include <sstream>
#include "IntDoubleLinkedList.hpp"

IntDoubleLinkedList::IntDoubleLinkedList() { numElements = 0; front = back = nullptr; }
IntDoubleLinkedList::~IntDoubleLinkedList() { clear(); }
int IntDoubleLinkedList::size() const { return numElements; }
bool IntDoubleLinkedList::isEmpty() const { return numElements == 0; }

void IntDoubleLinkedList::addFirst(const int e) {
    Node *aux = new Node(e);
    if ( front == nullptr) { front = back = aux; }
    else { front->prev = aux; aux->next = front; front = aux; }
    ++numElements;
}

void IntDoubleLinkedList::addLast(const int e) {
    Node *aux = new Node(e);
    if ( front == nullptr) { front = back = aux; }
    else { aux->prev = back; back->next = aux; back = aux; }
    ++numElements;
}

bool IntDoubleLinkedList::removeFirst(int &e) {
    if ( numElements == 0 ) return false;
    --numElements; e = front->data; Node *aux = front; front = front->next; delete aux;
    if ( front == nullptr) back = nullptr;
    else front->prev = nullptr;
    return true;
}

bool IntDoubleLinkedList::removeLast(int &e) {
    if ( numElements == 0 ) return false;
    --numElements; e = back->data; Node *aux = back; back = back->prev; delete aux;
    if ( back == nullptr ) front = nullptr;
    else back->next = nullptr;
    return true;
}
```

Exercício 12: IntDoubleLinkedList.cpp (continuação)

```
bool IntDoubleLinkedList::first(int &e) const {
    if ( numElements == 0 ) return false;
    else { e = front->data; return true; }
}

bool IntDoubleLinkedList::last(int &e) const {
    if ( numElements == 0 ) return false;
    else { e = back->data; return true; }
}

void IntDoubleLinkedList::clear() {
    while (front != nullptr) { Node *aux = front; front = front->next; delete aux; }
    numElements = 0; back = nullptr;
}

string IntDoubleLinkedList::str() const {
    stringstream ss;
    ss << "|";
    for (Node *aux = front; aux != nullptr; aux = aux->next)
        ss << aux->data << "|";
    return ss.str();
}

string IntDoubleLinkedList::reverseStr() const {
    stringstream ss;
    ss << "|";
    for (Node *aux = back; aux != nullptr; aux = aux->prev)
        ss << aux->data << "|";
    return ss.str();
}
```

Exercício 13: StringArrayList.cpp

```
#include <sstream>
#include "StringArrayList.hpp"

StringArrayList::StringArrayList(int mxSz) {
    numElements = 0;    maxElements = ( mxSz < 1 ) ? 10 : mxSz;
    list = new string[maxElements];
}

StringArrayList::~StringArrayList() { delete[] list; }
void StringArrayList::clear() { numElements = 0; }
int StringArrayList::size() const { return numElements; }
int StringArrayList::maxSize() const { return maxElements; }
bool StringArrayList::isEmpty() const { return numElements == 0; }
bool StringArrayList::isFull() const { return numElements == maxElements; }

bool StringArrayList::add(const string &s) {
    if ( numElements == maxElements ) return false;
    list[ numElements++ ] = s;
    return true;
}

bool StringArrayList::add(const int index, const string &s) {
    if ( numElements == maxElements || index < 0 || index > numElements ) return false;
    for (int i=numElements; i>index; --i)
        list[i] = list[i-1];
    list[index] = s;
    ++numElements;
    return true;
}

bool StringArrayList::remove(const int index) {
    if ( index < 0 || index >= numElements ) return false;
    --numElements;
    for (int i=index; i<numElements; ++i)
        list[i] = list[i+1];
    return true;
}
```

Exercício 13: StringArrayList.cpp (continuação)

```

bool StringArrayList::get(const int index, string &s) {
    if (index < 0 || index >= numElements) return false;
    s = list[index];
    return true;
}

bool StringArrayList::set(const int index, const string &s) {
    if (index < 0 || index >= numElements) return false;
    list[index] = s;
    return true;
}

bool StringArrayList::contains(const string &s) {
    for (int i=0; i<numElements; ++i) if (list[i] == s) return true;
    return false;
}

int StringArrayList::indexOf(const string &s) { return indexOf(0,s); }

int StringArrayList::indexOf(int index, const string &s) {
    if ( index < 0 || index >= numElements ) return -1;
    for (int i=index; i<numElements; ++i) if (list[i] == s) return i;
    return -1;
}

string StringArrayList::str() const {
    stringstream ss;
    for (int i=0; i<numElements; ++i) ss << list[i];
    return ss.str();
}

```


Exercício 14: exercicio14-solucao.cpp

```

#include <iostream>
using namespace std;
struct Node {
    char info; Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    Node *head = nullptr, *tail = nullptr; // Criação
    for (char l='B'; l<='F'; l++) {
        if (l == 'D') continue; // Evita que o Node com a info 'C' seja inserido na lista...
        Node *aux = new Node(l);
        if (tail == nullptr) { head = tail = aux; }
        else { tail->next = aux; tail = aux; }
    }

    // Solução
    Node *novo = new Node('G'), *prev = nullptr, *aux = head;
    while (aux != nullptr && aux->info < novo->info) { prev = aux; aux = aux->next; }
    if (prev == nullptr) { novo->next = head; head = novo; } // Inserção no início
    else if (aux == nullptr) { prev->next = novo; tail = novo; } // Inserção no final
    else { prev->next = novo; novo->next = aux; } // Inserção no meio

    cout << "head-->_"; // Exibição
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << "|" << aux->info << (aux->next==nullptr?"|X|_":"||_>_");
    cout << "_<--tail" << endl;
    while (head != nullptr) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}

```

Exercício 15: exercicio15-solucao.cpp

```

#include <iostream>
using namespace std;
struct Node {
    char info; Node *next;
    Node(char i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

int main() {
    string s = "ADCBE"; Node *head = nullptr, *tail = nullptr; // Criação
    for (int i=0; i<s.length(); ++i) {
        Node *aux = new Node( s[i] );
        if ( tail == nullptr ) { head = tail = aux; }
        else { tail->next = aux; tail = aux; }
    }

    // Solução
    Node *prev_d = head, *nodo_d = prev_d->next,
        *prev_b = head->next->next, *nodo_b = prev_b->next;
    prev_d->next = nodo_b; prev_b->next = nodo_d;
    Node *aux = nodo_b->next; nodo_b->next = nodo_d->next; nodo_d->next = aux;

    cout << "head-->_"; // Exibição
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << "|" << aux->info << (aux->next==nullptr?"|X|_":"||_>_");
    cout << "_--tail" << endl;
    while (head != nullptr) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}

```

Exercício 17: exercicio17.cpp

```

#include <iostream>
using namespace std;
struct Node {
    int info; Node *next;
    Node(int i) { info = i; next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

void reverse(Node **head, Node **tail) {
    Node *newHead = nullptr, *newTail = nullptr;
    while ( *head != nullptr ) {
        Node *aux = *head; *head = (*head)->next; // Retira o primeiro da lista
        aux->next = newHead; newHead = aux; if ( newTail == nullptr ) newTail = aux; // Insere ele no início da "
    }
    *head = newHead; *tail = newTail;
}

int main() {
    Node *head = nullptr, *tail = nullptr; // Criação
    for (int i=10; i<=50; i+=10) { // Insere 10, 20, 30, 40, 50 pelo final da lista
        Node *aux = new Node( i );
        if ( tail == nullptr ) { head = tail = aux; }
        else { tail->next = aux; tail = aux; }
    }
    reverse(&head, &tail);
    cout << "head-->"; // Exibição a partir do início: 50, 40, 30, 20, 10
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << "|" << aux->info << (aux->next==nullptr?"|X|_":"||_>");
    cout << "_<--tail" << endl;
    while (head != nullptr) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}

```

Exercício 18: exercicio18.cpp

```

#include <iostream>
using namespace std;
struct Node {
    int info; Node *prev, *next;
    Node(int i) { info = i; prev = next = nullptr; cout << "+_Node(" << info << ")_criado..." << endl; }
    ~Node() { cout << "-_Node(" << info << ")_destruido..." << endl; }
};

void reverse(Node **head, Node **tail) {
    Node *aux = *head;
    while (aux != nullptr) {
        Node *sw = aux->prev; aux->prev = aux->next; aux->next = sw;
        aux = aux->prev;
    }
    Node *sw = *head; *head = *tail; *tail = sw;
}

int main() {
    Node *head = nullptr, *tail = nullptr; // Criação
    for (int i=10; i<=50; i+=10) { // Insere 10, 20, 30, 40, 50 pelo final da lista
        Node *aux = new Node( i );
        if ( tail == nullptr ) { head = tail = aux; }
        else { aux->prev = tail; tail->next = aux; tail = aux; }
    }
    reverse(&head, &tail);
    cout << "head-->"; // Exibição a partir do início: 50, 40, 30, 20, 10
    for (Node *aux = head; aux != nullptr; aux = aux->next)
        cout << (aux->prev==nullptr?"|X|":"_<==>_||") << aux->info << (aux->next==nullptr?"|X|_":"||");
    cout << "_<--tail" << endl;
    cout << "tail-->"; // Exibição a partir do final: 10, 20, 30, 40, 50
    for (Node *aux = tail; aux != nullptr; aux = aux->prev)
        cout << (aux->next==nullptr?"|X|":"_<==>_||") << aux->info << (aux->prev==nullptr?"|X|_":"||");
    cout << "_<--head" << endl;
    while (head != nullptr) { Node *aux = head; head = head->next; delete aux; } // Desalocação
    return 0;
}

```