

Estruturas Lineares

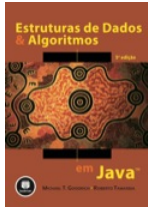
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

30 de agosto de 2023

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
 - Tem a ordem interna de seus elementos definida pela forma como são feitas inserções e remoções na estrutura
 - Costuma 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

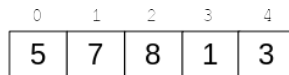
- Pessoas na fila de um caixa (ordem definida pela chegada e posição na fila)
- Pessoas na sala de espera de um consultório (ordem definida pela chegada)
- Conjunto de notas dos alunos de uma turma
- Itens no estoque de uma loja
- Palavras de um texto
- Letras de uma palavra
- Especificação de operações 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:

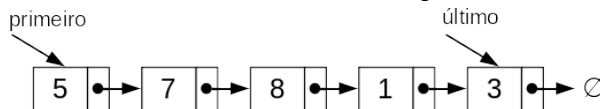
- **Sequencial ou Contígua**

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]; // ALOCACAO 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]; // ALOCACAO 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;

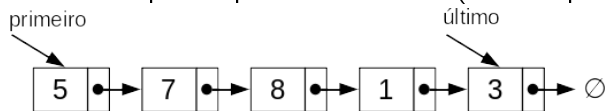
int main() {

    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;

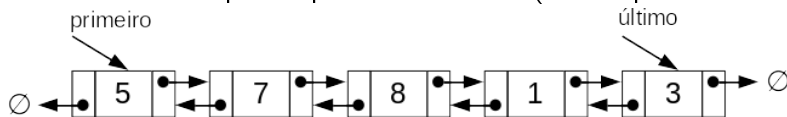
    return 0;
}
```

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

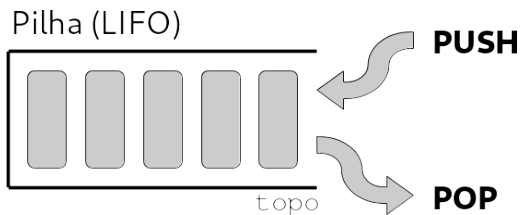
Operações Básicas sobre Estruturas Lineares

- 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*)

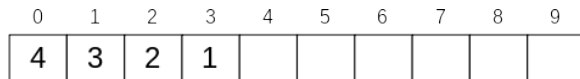
Pilha ou *Stack*

- 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



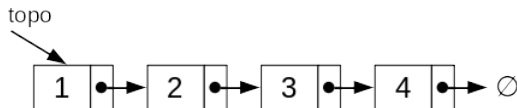
Pilha: Implementações Possíveis

• Arranjo

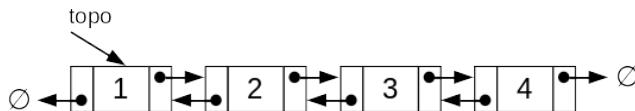


tamMax = 10 / tam = 4 / posTopo = 3 / valTopo = 1

• 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, pós-fixada, pré-fixada

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;
};

#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):uu"; IntStack stack(4); print(stack);
    e = 1; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 2; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 3; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    e = 4; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 5; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    e = 6; cout << "push(" << e << "):" << "uu"; res = stack.push(e); cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(" << e << "):" << "uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    res = stack.pop(e); cout << "pop(X):uu"; cout << (res?"OKuu":"ERRO"); print(stack);
    e = 7; cout << "push(" << e << "):" << "uu"; 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 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 da questão 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:

ESTE EXERCICIO E MUITO FACIL.

A saída deve ser:

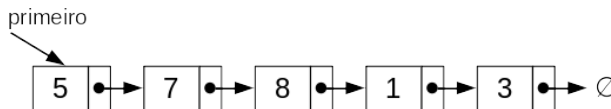
ETSE OICICREXE E OTIUM LICAF

- 3 Considerando ainda a classe `CharStack`, escreva uma função que verifique se uma palavra (`string`) é um palíndromo.
- 4 Implemente uma função em C++ para testar se duas pilhas de caracteres (objetos da classe `CharStack`), P1 e P2, são iguais.
- 5 Implemente uma função em C++ para copiar os elementos de uma pilha. Desenvolva uma operação para copiar elementos de uma pilha P1 para uma pilha P2 (sem destruir P1).

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



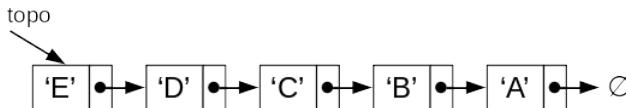
- Ao inserir um elemento, deve-se alocar um nodo, preenchê-lo e colocá-lo na posição desejada na estrutura (ajustando as referências necessárias)
- Ao remover um elemento, é preciso removê-lo (ajustando as referências necessárias) e desalocá-lo

Estruturas Encadeadas em C++

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

```
struct Nodo {  
    char letra;  
    Nodo *prox;  
    Nodo(char l) {  
        letra = 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 Nodo {
    char letra;
    Nodo *prox;
    Nodo(char l) { letra = l; prox = nullptr; cout << "Nodo_" << letra << "_criado..." << endl; }
    ~Nodo() { cout << "Nodo_" << letra << "_destruido..." << endl; }
};

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

    Nodo *topo = nodo5;
    nodo5->prox = nodo4;
    nodo4->prox = nodo3;
    nodo3->prox = nodo2;
    nodo2->prox = nodo1;

    for (Nodo *aux = topo; aux != nullptr; aux = aux->prox)
        cout << aux->letra << endl;

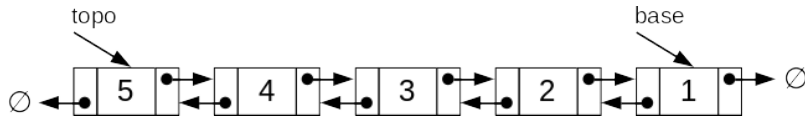
    while (topo != nullptr) {
        Nodo *aux = topo;
        topo = topo->prox;
        delete aux;
    }

    return 0;
}

```

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 7: IntLinkedStack.hpp

```

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

class IntLinkedStack {
private:
    int numElements;
    struct Reg {
        int data;
        Reg *next;
        Reg(int d) { data = d; next = nullptr; }
    };
    Reg *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;
};
#endif

```

Exercício 7: 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;
}

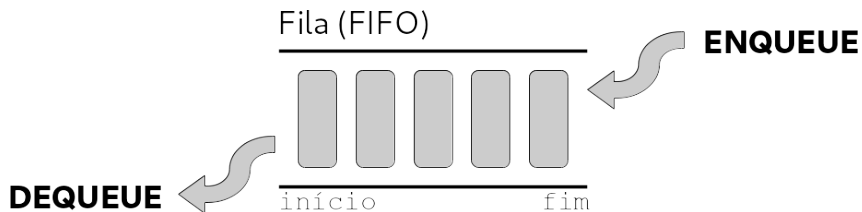
int main() {
    int vet[] = {0,1,2,3,4,5,6,7,8,9};
    int numElements = sizeof(vet)/sizeof(int);
    IntLinkedStack stack;
    cout << "uuuuuuuuuu"; print(stack);
    for (int i=0; i<numElements; ++i) {
        cout << "push(" << vet[i] << "):uu";
        stack.push( vet[i] );
        print(stack);
    }
    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



Fila: Implementações Possíveis

• Arranjo circular

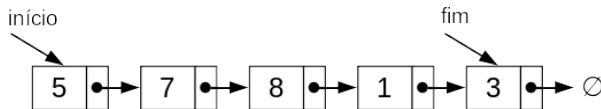
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | 3 | 1 | 8 | 7 | 5 | | | |

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

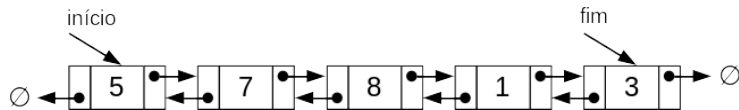
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 5 | | | | | | 3 | 1 | 8 |

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

• Lista Simplesmente Encadeada



• Lista Duplamente Encadeada



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.

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)`: 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;
};

#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 |

Créditos

Créditos

- Estas lâminas contêm trechos inspirados em materiais criados e disponibilizados pelos professores Isabel Harb Manssour e Iaçanã Janiski 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;
};

#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: questao2.cpp

```

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

using namespace std;

string invertePalavrasDaFrase(string frase) {
    char ch;
    CharStack stack(1000);
    string res = "";
    for (int i=0; i<frase.size(); ++i) {
        char c = frase[i];
        if (c == '.' || c == '_') {
            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 = "ESTE_EXERCICIO_E_MUITO_FACIL.";
    cout << frase1 << endl;
    cout << invertePalavrasDaFrase(frase1) << endl;
    return 0;
}

```

Exercício 6: pilhla_simplesmente_encadeada2.cpp

```

#include <iostream>

using namespace std;

struct Nodo {
    char letra;
    Nodo *prox;
    Nodo(char l) { letra = l; prox = nullptr; cout << "Nodo_" << letra << "_criado..." << endl; }
    ~Nodo() { cout << "Nodo_" << letra << "_destruido..." << endl; }
};

int main() {
    Nodo *topo = nullptr;
    for (char c = 'A'; c <= 'E'; ++c) {
        Nodo *nodo = new Nodo(c);
        nodo->prox = topo;
        topo = nodo;
    }

    for (Nodo *aux = topo; aux != nullptr; aux = aux->prox)
        cout << aux->letra << endl;

    while (topo != nullptr) {
        Nodo *aux = topo;
        topo = topo->prox;
        delete aux;
    }

    return 0;
}

```

Exercício 7: pilha_duplamente_encadeada.cpp

```
#include <iostream>

using namespace std;

struct Nodo {
    int valor;
    Nodo *prev;
    Nodo *prox;
    Nodo(int v) { valor = v; prev = nullptr; prox = nullptr; cout << "Nodo_" << valor << "_criado..." << endl; }
    ~Nodo() { cout << "Nodo_" << valor << "_destruido..." << endl; }
};

int main() {
    Nodo *nodo1 = new Nodo(1);
    Nodo *nodo2 = new Nodo(2);
    Nodo *nodo3 = new Nodo(3);
    Nodo *nodo4 = new Nodo(4);
    Nodo *nodo5 = new Nodo(5);

    Nodo *topo = nodo5;    nodo5->prox = nodo4;
    nodo4->prev = nodo5;    nodo4->prox = nodo3;
    nodo3->prev = nodo4;    nodo3->prox = nodo2;
    nodo2->prev = nodo3;    nodo2->prox = nodo1;
    nodo1->prev = nodo2;    Nodo *base = nodo1;

    for (Nodo *aux = topo; aux != nullptr; aux = aux->prox)
        cout << aux->valor << endl;
    for (Nodo *aux = base; aux != nullptr; aux = aux->prev)
        cout << aux->valor << endl;

    while (topo != nullptr) {
        Nodo *aux = topo;
        topo = topo->prox;
        delete aux;
    }

    return 0;
}
```

Exercício 8: IntLinkedList.cpp

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

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

void IntLinkedList::push(const int e) {
    Reg *novo = new Reg(e);
    novo->next = stack; stack = novo;
    ++numElements;
}

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

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

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

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