Estruturas Lineares

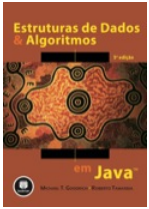
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

29 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 acontede 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

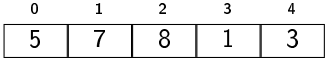
- Permitem representar um conjunto de dados de um mesmo tipo (com alguma afinidade) de forma a preservar a relação de ordem entre seus elementos
- Cada elemento da estrutura é chamado de nó, ou nodo.
- Uma estrutura linear é definida como:
  - Um conjunto de N nós, organizados de forma a refletir a posição relativa dos mesmos
  - Se $N > 0$, os nós da estrutura serão x_1, x_2, ..., 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

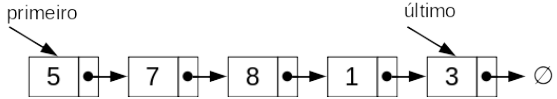
- Pessoas na fila de um caixa (ordem definida pela chegada e posição na fila)
- Pessoas na sala de espera de um consultório (ordem definida pela chegada)
- Conjunto de notas dos alunos de uma turma
- Itens no estoque de uma loja
- Palavras de um texto
- Letras de uma palavra
- Especificação de operações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

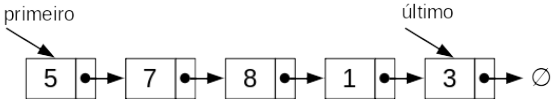
```
#include <iostream>
using namespace std;
int main() {

  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)

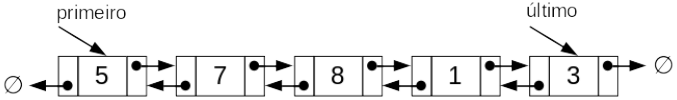
```
#include <iostream>

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

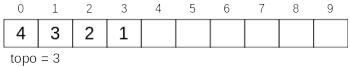
Operações Básicas sobre Estruturas Lineares

- 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

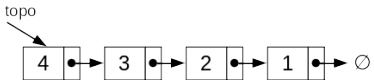
Pilhas

Pilha ou *Stack*

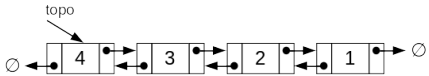
- Usa a política *LIFO – Last In First Out* (o último elemento que entrou, é o primeiro a sair)
- Possui apenas uma entrada, chamada de topo, a partir da qual os dados são inseridos e removidos
- Pode ser implementada usando
  - Arranjo:



  - 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 e 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();
  bool push(const int &e);
  bool pop(int &e);
  bool top(int &e) const;
  int size() const;
  int maxSize() const;
  bool isEmpty() const;
  bool isFull() const;
  void clear();
  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;  }

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; }
}

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; }

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 << "␣␣" << stack.str() << "␣␣size=" << stack.size() << "/" << stack.maxSize() << "␣␣top=";
  int t;   bool res = stack.top(t);
  if (res) cout << t; else cout << "X";
  cout << "␣␣isEmpty=" << stack.isEmpty() << "␣␣isFull=" << stack.isFull() << endl;
}

int main() {
  int e;
  bool res;
  cout << "IntStack(4):␣";  IntStack stack(4);  print(stack);
  e = 1;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO");  print(stack);
  e = 2;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO");  print(stack);
  e = 3;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO");  print(stack);
  res = stack.pop(e); cout << "pop(" << e << "):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  e = 4;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO"); print(stack);
  e = 5;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO"); print(stack);
  e = 6;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO"); print(stack);
  res = stack.pop(e); cout << "pop(" << e << "):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  res = stack.pop(e); cout << "pop(" << e << "):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  res = stack.pop(e); cout << "pop(" << e << "):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  res = stack.pop(e); cout << "pop(" << e << "):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  res = stack.pop(e); cout << "pop(X):␣␣";  cout << (res?"OK␣␣":"ERRO");  print(stack);
  e = 7;  cout << "push(" << e << "):␣"; res = stack.push(e);  cout << (res?"OK␣␣":"ERRO"); print(stack);
  cout << "clear():␣OK␣␣";  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`).

Créditos

Créditos

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