Algoritmos e Estruturas de dados

Professora:
Sílvia Campos

IPRJ/UERJ

Tópicos

- Algoritmos
  - O que é?
  - Por que estudar?
- Programação Orientada a Objetos (POO) (Em C++)
- Análise de Complexidade
- Recursão
- Estruturas de dados
- Métodos de ordenação

Algoritmo

- **Processo sistemático** para a resolução de um problema.
- Usado para descrever, de forma lógica, os passos a serem executados no cumprimento de determinada tarefa.
- Ferramenta para resolver um ***problema computacional*** bem especificado. A partir de um valor ou conjunto de valores como *entrada*, produz algum valor ou conjunto de valores como *saída*. Durante o processo de computação, o algoritmo manipula dados, gerados a partir de sua entrada.
- Descreve soluções de problemas do **nosso mundo (ilimitado)** utilizando os recursos do **mundo computacional (limitação de hardware/software)**.
- O conceito de um algoritmo foi formalizado em 1936 pela Máquina de Turing de Alan Turing e pelo cálculo lambda de Alonzo Church, que formaram as primeiras fundações da Ciência da computação.

Exemplos de problemas a serem resolvidos com algoritmos:

- Calcular média aritmética / ponderada.
- Ordenar uma sequência de números.
- Internet – gerenciar e manipular grandes quantidades de informações. Exemplos localização de boas rotas, uso de mecanismos de pesquisa para encontrar com rapidez páginas que residem informações específicas.
- Comércio eletrônico. Criptografia de chave pública e assinaturas digitais (números de cartão de crédito, senhas...)
- Mapa rodoviário. Como encontrar a rota mais curta.

- Um **programa de computador** é essencialmente um algoritmo que diz ao computador os passos específicos e em que ordem eles devem ser executados, como por exemplo, os passos a serem tomados para calcular as notas que serão impressas nos boletins dos alunos.

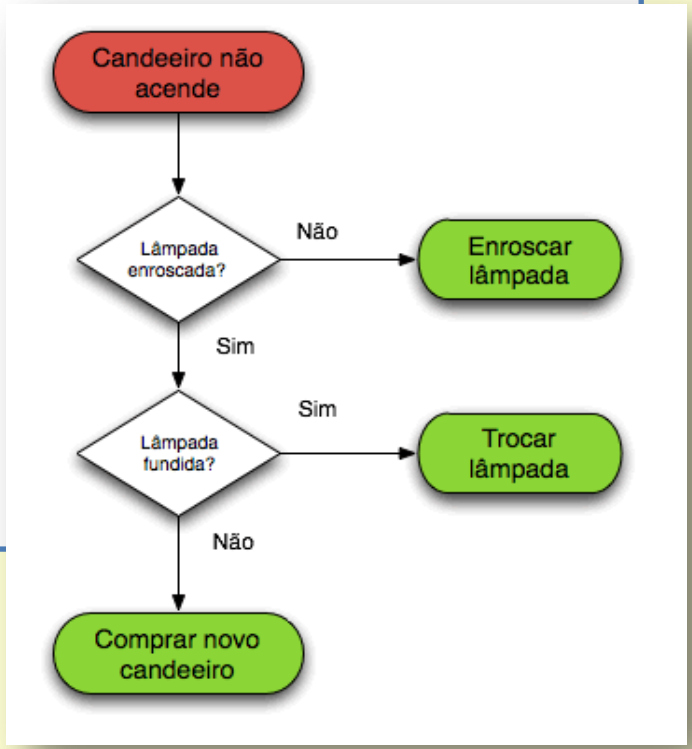
- **Estrutura de dados** é um modo particular de armazenamento e organização de dados em um computador de modo que possam ser usados eficientemente.

- A **corretividade** do algoritmo pode ser provada matematicamente, bem como a quantidade assintótica de tempo e espaço (complexidade) necessários para a sua execução. A análise de algoritmos trata destes aspectos.

- Algoritmo: lista de procedimentos bem definida, na qual as instruções são executadas passo a passo a partir do começo da lista; uma ideia que pode ser visualizada por um fluxograma.

- Tal formalização adota as premissas da programação imperativa, que é uma forma mecânica para visualizar e desenvolver um algoritmo. Concepções alternativas para algoritmos variam em programação funcional e programação lógica.

- Um exemplo de algoritmo imperativo.
  O estado em vermelho indica a entrada do algoritmo enquanto os estados em verde indicam as possíveis saídas.



Análise de algoritmos

- Estuda as **técnicas de projeto de algoritmos de forma abstrata**, sem se preocupar com a forma de implementação. E sim, com os **recursos necessários** para a execução do algoritmo, como o **tempo de execução e o espaço de armazenamento de dados.**

- Para um dado algoritmo, pode-se ter diferentes quantidades de recursos alocados de acordo com os parâmetros passados na entrada. Por exemplo, se definirmos que o fatorial de um número natural é igual ao fatorial de seu antecessor multiplicado pelo próprio número, fica claro que a execução de fatorial(10) consome mais tempo que a execução de fatorial(5).

Um meio de exibir um algoritmo para analisá-lo é através da implementação por pseudocódigo em português estruturado. Exemplo de um algoritmo que retorna (valor de saída) a soma de dois valores (conhecidos como parâmetros ou argumentos, valores de entrada) que são introduzidos na chamada da função:

```
Algoritmo "SomaDeDoisValores";
variável:
      SOMA,A,B: inteiro;
inicio
      Escreva("Digite um numero");
      Leia(A);
      escreva("digite outro numero");
      leia(B);
      SOMA ← A + B;
      escreva(SOMA);
fim.
```

Classificação por implementação

Classificação de algoritmos pela maneira pelo qual foram implementados

- **Iterativo ou Recursivo** - Algoritmos iterativos usam estruturas de repetição (tais como laços), ou ainda estruturas de dados adicionais (tais como pilhas), na resolução de problemas. Algoritmos recursivos possuem a característica de invocarem a si mesmos repetidamente até que certa condição seja satisfeita, para o término; método comum em programação funcional. Cada algoritmo recursivo possui um algoritmo iterativo equivalente e vice-versa, com mais ou menos complexidade em sua construção.

- **Lógico** - um algoritmo pode ser visto como uma dedução lógica controlada. O componente lógico expressa os axiomas usados na computação e o componente de controle determina a maneira como a dedução é aplicada aos axiomas. Tal conceito é base para a programação lógica.

- **Serial** ou **paralelo**.

  Serial: Executa cada instrução individualmente, como uma lista de execução. Base para a programação imperativa.

  Paralelo: levam em conta as arquiteturas de computadores com mais de um processador para executar mais de uma instrução ao mesmo tempo. **Dividem os problemas em subproblemas** e os delegam a quantos processadores estiverem disponíveis, agrupando no final o resultado dos subproblemas em um resultado final ao algoritmo. Base para a programação paralela.

- **Determinístico** ou **não-determinístico** - Determinísticos: resolvem o problema com uma decisão exata a cada passo. Não-determinísticos: resolvem o problema ao deduzir os melhores passos através de estimativas sob forma de heurísticas.

- **Exato** ou **aproximado** - enquanto alguns algoritmos encontram uma resposta exata, algoritmos de aproximação procuram uma resposta próxima a verdadeira solução, por estratégia determinística ou aleatória. Possuem aplicações práticas sobretudo para problemas complexos, quando uma resposta correta é inviável devido a sua complexidade computacional.

Tipos abstratos de dados (TAD)

A **fase de delineamento** do programa
contendo seus **requisitos** deve **preceder** o **processo de codificação**!

Ideal: adiar os detalhes da implementação e das estruturas de dados particulares a serem usadas

É importante **especificar cada tarefa** em **termos de entrada e saída** e focar nossa atenção no **que o programa deveria fazer** (e não em como).

Estrutura de dados: implementação concreta de um Tipo Abstrato de Dados (TAD).

Tipos abstratos de dados (TAD)

- **TAD: Item especificado em termos de operações;**
- **Modelo matemático**, acompanhado das **operações** definidas sobre o modelo. Exemplo: O conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- **Pode ser definido por meio de um par (v,o)** em que: *v* é um conjunto de valores e *o* é um conjunto de operações sobre esses valores. Exemplo: Tipo *real*. v=$\mathcal{R}$ e o={+,-,*, /, =, <, >, <=, >=}.
- A definição do tipo de dados e todas as operações definidas sobre ele podem ser localizadas em uma única seção do programa (na Linguagem C, por exemplo, em um arquivo de cabeçalhos).
- Uma Linguagem Orientada a Objetos (LOO), como C++, tem vínculo direto com os tipos de dados, implementando-os como uma **classe**.

Tipos abstratos de dados

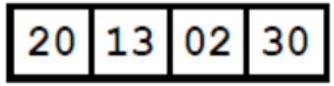
- Exemplo: lista de números

  - Operações:
    - Criar uma nova lista vazia
    - Inserir um número no final da lista

Programa usuário do TAD:

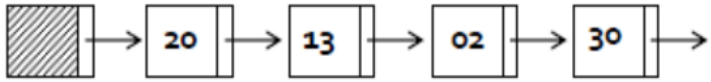
```
int main() {
   Lista *L;
   int x = 20;

   L = cria_lista();
   insere(L, x);
   ...
}
```

TAD implementado com vetor:




```
void insere(Lista *l, int x) {
   l->arranjo[...] = x;
   ...
}
```

TAD implementado com lista:




```
void insere(Lista *l, int x) {
   Celula *c = cria_celula(x);
   l->ultimo = c;
}
```

Orientação a objetos

Aborda os conceitos de:

- Classe:
  - tipo de dado; inclui a especificação de dados e as funções que operam sobre esses dados.
- Objeto:
  - instância de uma classe.
- Herança:
  - relacionamento entre classes.
- Encapsulamento:
  - combina dados e operações relacionadas.
- Polimorfismo:
  - Mesmo nome para identificar diferentes procedimentos.

```
class C {
public:
        C  (char *s = "",  int i = 0,  double d = 1) {
                strcpy(dadosMembro1,s);
                dadosMembro2 = i;
                dadosMembro3 = d;
        }
        void funcaoMembro1 () {
                cout    << dadosMembro1 << ' ' << dadosMembro2 << ' '   << dadosMembro3 << endl;
        }

        void funcaoMembro2 (int i, char *s = "desconhecido") {
                dadosMembro2 = i;
                cout << i << " recebido de " << s << endl;
        }
protected:
        char dadosMembro1[20];
        int dadosMembro2;
        double dadosMembro3;
};

C objeto1 ("objeto1", 100, 2000),  objeto2 ("objeto2"), objeto3;
```

Vantagens da POO

- **Forte acoplamento dos dados e das operações** que pode ser bem usado na modelagem de eventos do mundo real;
- **Os objetos permitem que os erros sejam descobertos mais facilmente**, porque as operações são localizadas dentro de seus objetos. Mais fácil de rastrear os efeitos colaterais;
- **Princípio de ocultamento de informação** – os objetos nos permitem esconder, de outros objetos, certos detalhes de suas operações, de modo que essas operações não possam ser desfavoravelmente afetadas por outros objetos;
- **Facilita o desenvolvimento e entendimento** de programas de grande porte.
- **Possibilita a reutilização do código**, reduz custos de desenvolvimento.

Análise de Complexidade

Como avaliar a eficiência de um algoritmo?

- A **complexidade computacional:** indica quanto **esforço** é necessário para aplicar um algoritmo ou quão **custoso** é.

- **Critérios de eficiência**: **tempo** e **espaço**. O tempo de operação é sempre dependente do sistema, da linguagem, se o programa é compilado ou interpretado, etc..

- **Unidades lógicas** usadas no lugar das unidades de tempo real (*ms* ou *ns*) -> expressam uma relação entre o tamanho *n* de um arquivo ou de uma matriz e a quantidade de tempo *t* exigida para processar os dados.

- **Complexidade assintótica: medida de eficiência** usada para aproximação de funções. Notações: O-Grande, Ω e θ.

Recursão

- Técnica que define um problema em termos de uma ou mais versões menores deste mesmo problema.
- Expressa a solução de um problema em função do próprio problema. **Tipo de procedimento que contém, em sua descrição, uma ou mais chamadas a si mesmo – procedimento recursivo.** A chamada a si mesmo é dita *chamada recursiva*.
- Programa recursivo - usa um procedimento ou sub-rotina, que permite dar um nome a um comando, o qual pode chamar a si próprio.
- Todo procedimento, recursivo ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ele (chamada externa). Procedimento não-recursivo: todas as chamadas são externas!

Ex.: Soma dos N números inteiros (N=5):

S(5) = 1+2+3+4+5 = 15 -> S(5) = S(4) + 5 -> 10 + 5 = 15

S(4) = 1+2+3+4 =10 -> S(4) = S(3) + 4 -> 6 + 4 = 10

S(3) = 1+2+3 = 6 -> S(3) = S(2) + 3 -> 3 + 3 = 6

S(2) = 1+2 = 3 -> S(2) = S(1) + 2 -> 1 + 2 = 3

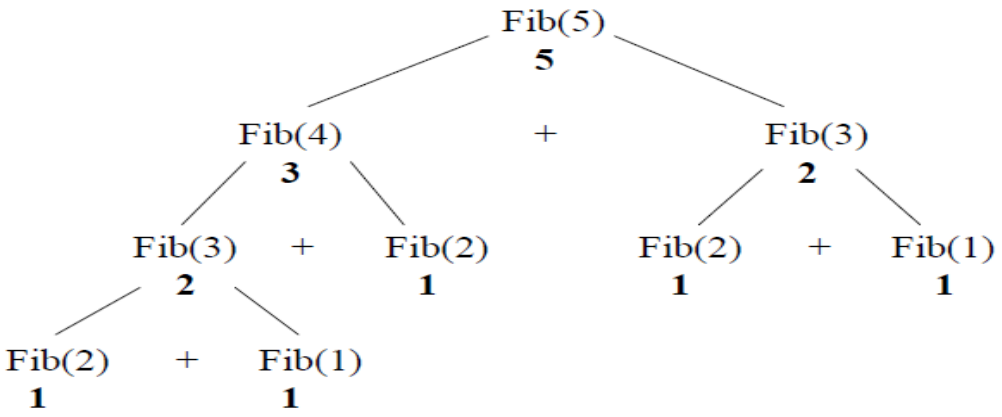
S(1) = 1 =1 -> S(1) = 1 -----> solução trivial

Seqüência de Fibonacci

$$F(N)=\begin{cases}1 & se\ \ N=1\ ou\ N=2 \quad \text{solução trivial}\\ F(N-1)+F(N-2), & se\ \ N>2 \quad \text{chamada recursiva}\end{cases}$$

```
void main()
{int n;
 scanf("%d",&n);
 printf("%d", fibonacci(n));
}
int fibonacci(int n)
{
 if ((n == 1)||(n == 2)) return (1);
 else  return(fibonacci(n-1)+fibonacci(n-2));
}
```

Seqüência de Fibonacci



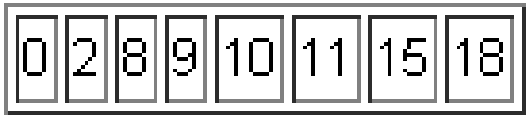
Estrutura de dados

- Vetores
- Matrizes
- Listas encadeadas
- Pilhas
- Filas
- Árvores binárias
- Grafos

Vetores

- Em computação, um **Vetor** (Array) ou **Arranjo** é o nome de uma matriz unidimensional considerada a mais simples das estruturas de dados. São constituídos por dados do mesmo tipo (homogêneos) e pelo tamanho; são agrupados continuamente na memória e acessados por sua posição (índice - geralmente um número inteiro) dentro do vetor.

- Exemplo de um vetor. Os valores internos são os dados alocados no vetor, e o seu tamanho é dado pelo número de casas disponíveis (no caso 8) e o índice representa a posição do dado no vetor.



- Em C/C++, o primeiro índice é o 0.

Do exemplo, o número 0 tem o índice 0 ($vet[0] = 0$), o número 2 tem índice 1 ($vet[1] = 2$), o número 8 tem índice 2 ($vet[2] = 8$), e assim sucessivamente, até chegar ao número 18, de índice 7 ($vet[7] = 18$).

Matrizes

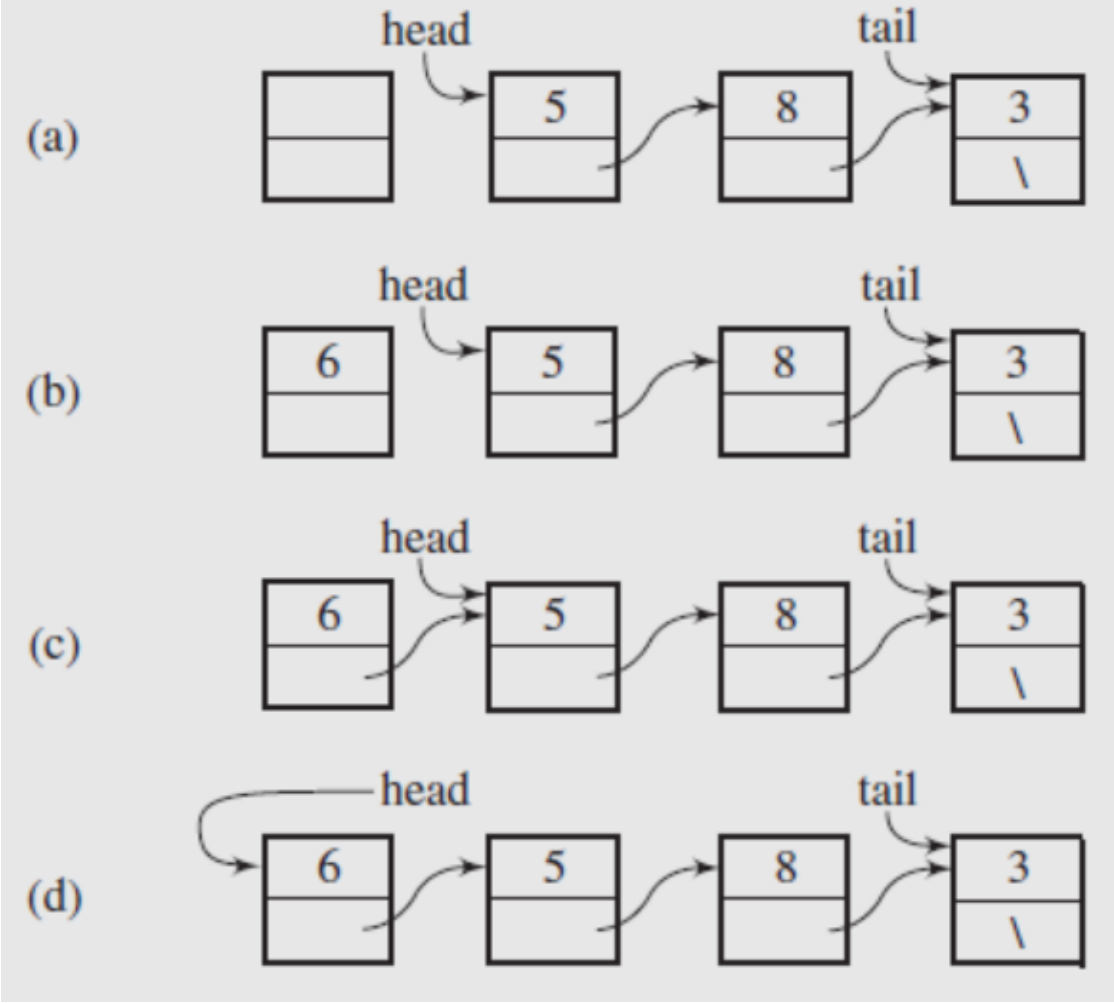
- **Arranjos ordenados que, ao contrário dos vetores,** podem ter *n* dimensões.
- Uma matriz de duas dimensões será chamada bidimensional, uma de três dimensões tridimensional, e assim por diante.
- Tal como os vetores, o tamanho também deve ser conhecido no momento da compilação.
- Os dados em uma matriz estão separados na memória do computador pela mesma distância; inserir um item na matriz exige que se movam outros dados nela;

Listas ligadas

- **Estrutura ligada:** coleção de nós, que armazenam dados, e de ligações com outros nós;
- Os nós podem estar em qualquer lugar na memória;
- A passagem de um nó para outro é realizada armazenando-se os endereços de outros nós;
- A implementação mais flexível é por meio de ponteiros.
- **Listas singularmente ligadas, listas duplamente ligadas, listas circulares**, listas de salto, listas auto-organizadas, tabelas esparsas.

Lista singularmente ligada

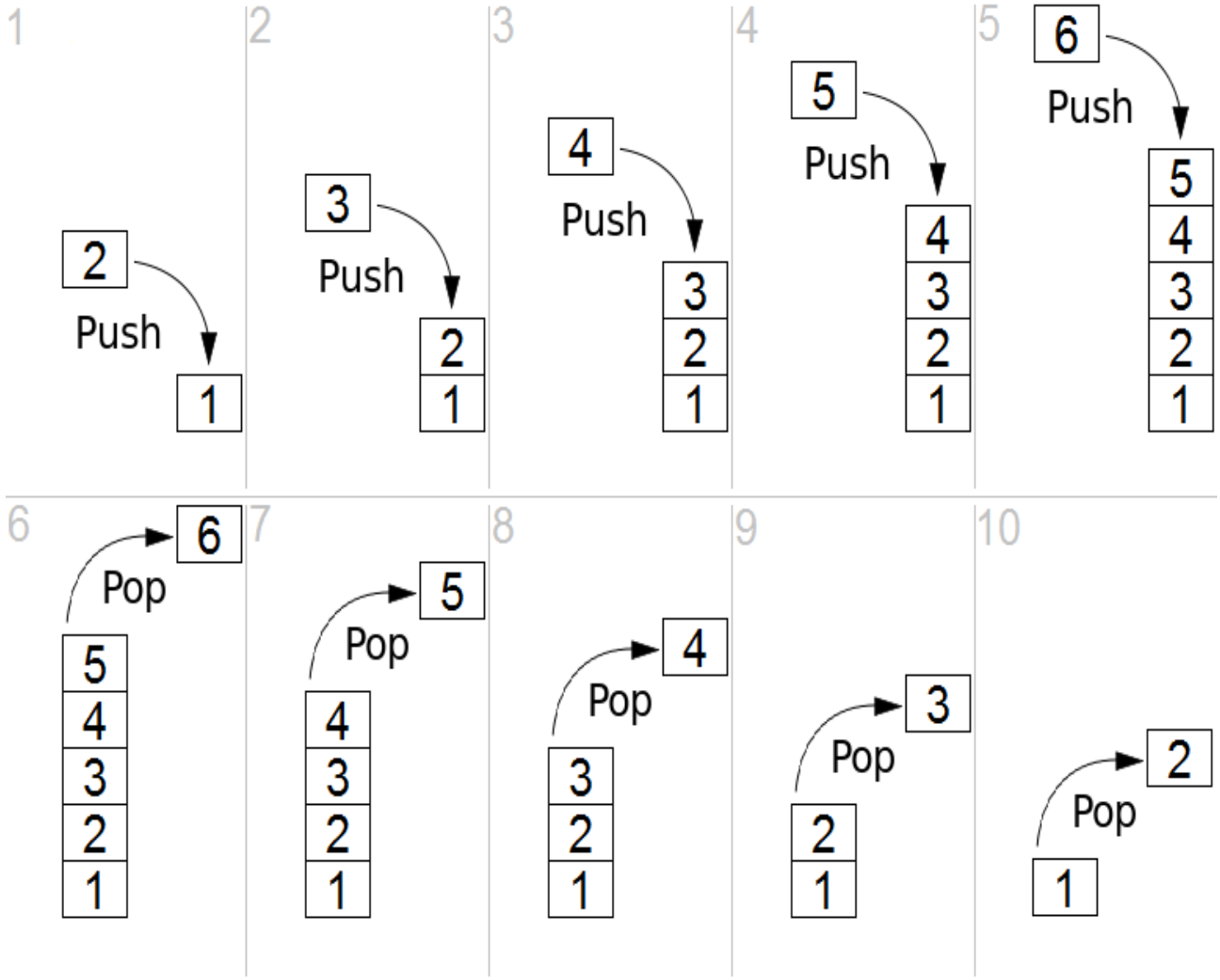
Exemplo de inserção de nó no início da lista



Pilhas

- Estrutura linear de dados que pode ser acessada somente por uma de suas extremidades para armazenar e recuperar dados;
- Estrutura LIFO – ***Last in/First out****;*
- Exemplo: pilha de bandejas em uma lanchonete, que são colocadas e retiradas sempre do topo;
- *Definida em termos das operações que modificam e das que verificam seu status;*
- *Operações principais:* ***push*** *(colocar) /* ***pop*** *(extrair)*

Pilhas

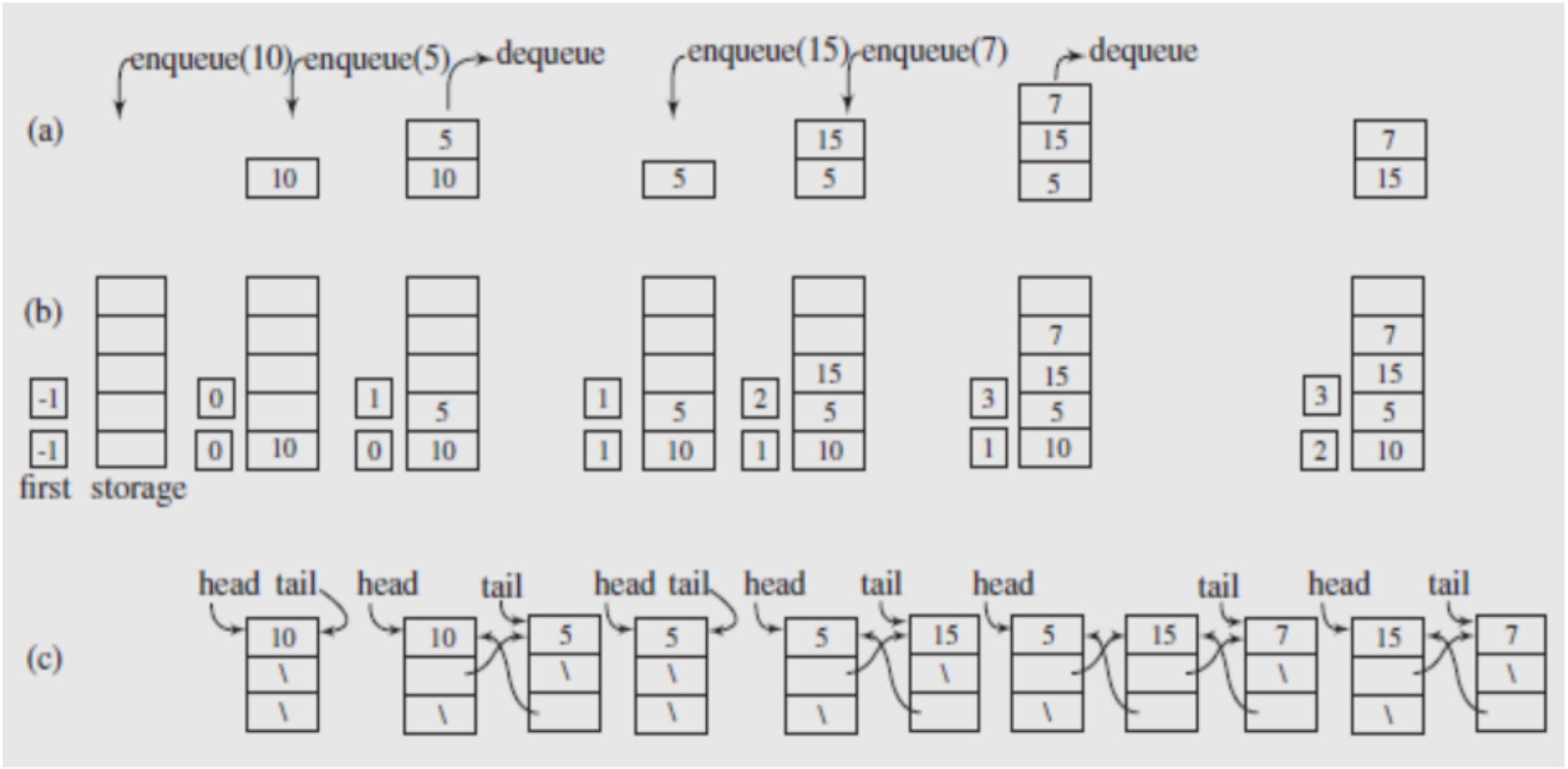

1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
Pop
6
5
4
3
2
1
7
Pop
5
4
3
2
1
8
Pop
4
3
2
1
9
Pop
3
2
1
10
Pop
2
1

Filas

- Estrutura linear de dados que cresce somando elementos ao seu final e decresce retirando elementos à sua frente. **Diferente de pilha, ambas as extremidades são usadas:** uma para adicionar novos elementos e outra para removê-los;
- Estrutura FIFO – ***FIRST IN/FIRST OUT****;*
- Exemplo: fila de banco;
- *Definida em termos das operações que modificam e das que verificam seu status.*
- *Operações principais:* ***enqueue*** *(colocar no final) /* ***queue*** *(extrair o primeiro elemento).*
- *Existem as Filas com prioridades e deques.*

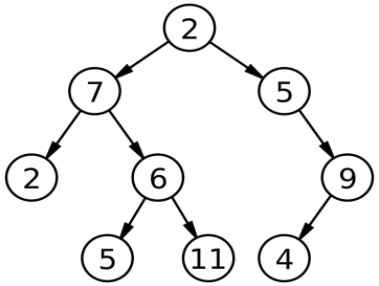
Filas

(a) Fila abstrata , (b) Implementação por vetor, (c) Implementação por lista duplamente ligada.



Árvores binárias

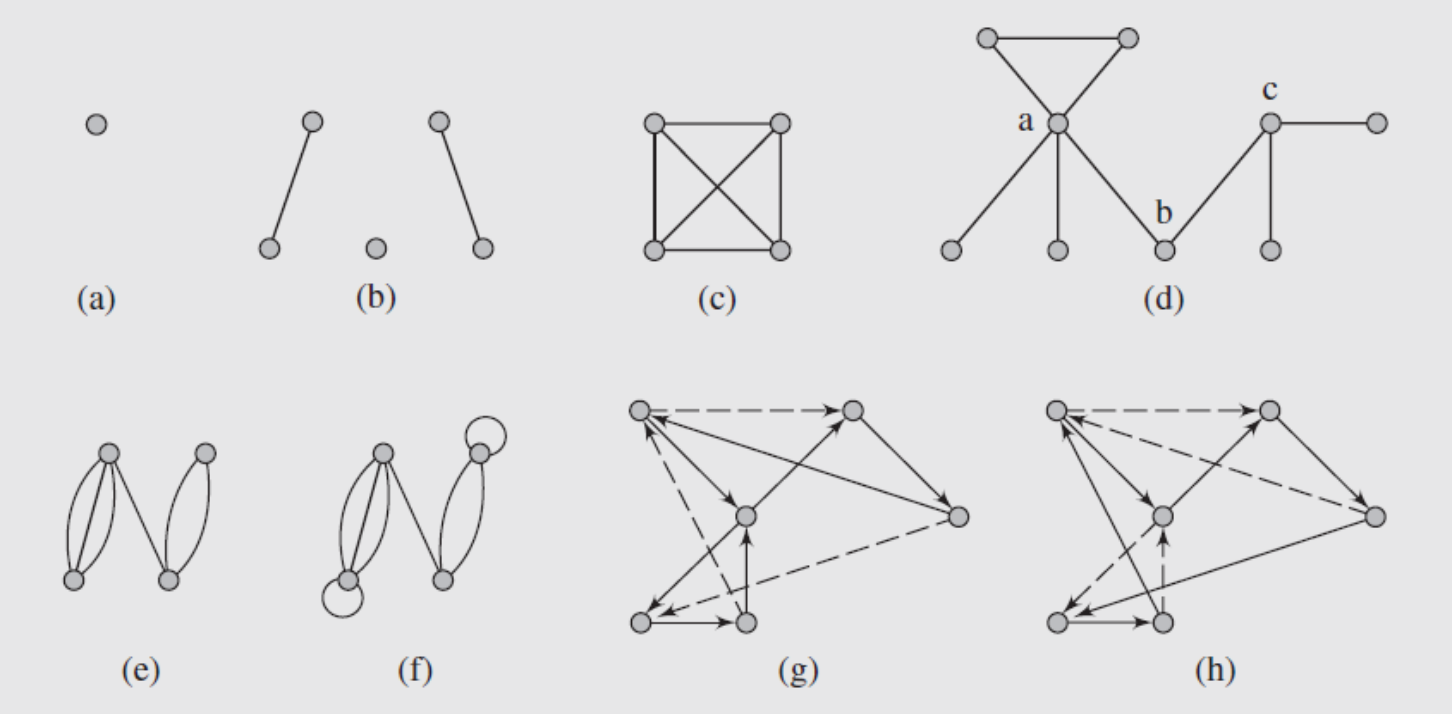
- Embora pilhas e filas reflitam alguma hierarquia, são limitas a uma só dimensão.



- Uma árvore consiste em nós e arcos.
- Representadas de cima para baixo, com a *raiz* no topo e as *folhas* na base.
- Uma árvore binária é aquela cujos nós tem dois filhos (possivelmente vazios) e cada filho é designado como filho à esquerda ou filho à direita.
- Árvores, árvores binárias e árvores binárias de busca.

Grafos

- Coleção de vértices (ou nós) e as conexões entre eles



Examples of graphs: (a–d) simple graphs; (c) a complete graph K_4; (e) a multigraph; (f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph.

Métodos de ordenação

- Ordenação por inserção *(Insertion Sort)*
- Ordenação por seleção *(Selection Sort)*
- Ordenação por borbulhamento *(bubble Sort)*
- Ordenação rápida (*Quick Sort*)
- Ordenação por fusão (*Merge Sort*)

Exercícios

- Pesquisar sobre dois exemplos de algoritmos e de duas diferentes estruturas de dados. Fazer um pequeno resumo de cada exemplo.

- Pesquisar sobre exemplos de diferentes tipos de TAD.

Referência bibliográficas

- **Estrutura de dados e Algoritmos em C++**
  Autor: Adam Drosdek.
  Editora: Cengage Learning.

- **Algoritmos – Teoria e Prática**
  Autores: Cormen, Leiserson, Rivest, Stein
  Editora: Campus

- **Treinamento em Linguagem C++**. Módulo 2, Victorine Viviane, Makron Books.