Pontifícia Universidade Católica de Goiás
Departamento de Ciência da Computação

Apostila

Estrutura de Dados e Algoritmos

Curso: Ciência da Computação

Prof.: Rafael Viana de Carvalho

https://github.com/r-carvalho/Computacao

Índice

Fundamentos: Análise e projeto de algoritmos

Os algoritmos são o objeto central de estudo em muitas áreas da ciência da computação. Vários autores fazem sua própria definição de algoritmo. De acordo com Cormen, algoritmo é um procedimento computacional que toma valores ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída. Dessa forma um conceito geral para algoritmo seria a sequência de passos computacionais que transformam a entrada em saída. (CORMEN et al, 2002)

De forma semelhante, Ziviani (2004) afirma que "Algoritmo pode ser visto como uma sequência de ações executáveis para obtenção de uma solução para um determinado tipo de problema". Já Sedgewick (1998) define algoritmos na área da ciência da computação como um método para resolução de problemas, adequado para implementação como um programa de computador.

De forma geral, Um algoritmo é uma sequência extremamente precisa de instruções que, quando lida e executada por uma outra pessoa, produz o resultado esperado, isto é, a solução de um problema. Esta sequência de instruções é nada mais nada menos que um registro escrito da sequência de passos necessários que devem ser executados para manipular informações, ou dados, para se chegar na resposta do problema. Podemos pensar em algoritmo como uma receita que dão cabo de uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser claras e precisas. Como exemplos de algoritmos podemos citar os algoritmos das operações básicas (adição, multiplicação, divisão e subtração) de números reais decimais. Outros exemplos seriam os manuais de aparelhos eletrônicos, ou uma receita de comida, que explicam passo a passo como, por exemplo, fazer um bolo:

> *Bata em uma batedeira a manteiga e o açúcar. Junte as gemas uma a uma até obter um creme homogêneo. Adicione o leite aos poucos. Desligue a batedeira e adicione a farinha de trigo, o chocolate em pó, o fermento e reserve. Bata as claras em neve e junte-as à massa de chocolate misturando delicadamente. Unte uma forma retangular pequena com manteiga e farinha e leve para assar em forno médio pré-aquecido por aproximadamente 30 minutos. Desenforme o bolo ainda quente e reserve.*

Os Algoritmos são o cerne da computação que toma como parâmetro de entrada um valor (ou um conjunto de valores) e que produz como saída um valor (ou um conjunto de valores). Ou seja, é uma sequência de passos computacionais que transformam um "input" num "output". Um algoritmo feito para um computador executar deve tornar explícito todas as informações implícitas. Também deve evitar o uso de frases ambíguas ou imprecisas e deve ser o mais detalhado possível e não pode ter frases de significado desconhecido. Além disso, é necessário que o algoritmo tenha um inicio e um fim, um conjunto de regras finito bem como um tempo finito de execução.

Os algoritmos podem ser representados de diversas formas:

- Em uma língua (português, inglês) através dos pseudocódigos: Uma descrição textual, estruturada e regida por regras; que descrevem os passos executados no algoritmo. Ela **utiliza expressões pré-definidas** para representar ações e fluxos de controle. Ex.:

```
algoritmo_Media_Final
    declare
      inteiro: P1, P2, P3, P4;
      real: media;
    inicio
      leia(P1, P2, P3, P4);
      media ← (P1+P2+P3+P4)/4;
      escreva(media);
fim fim_algoritmo
```

- Em uma linguagem de programação (C++, Java, Python, etc.): Utiliza apenas as instruções existentes na linguagem específica e é muito detalhada nas preocupações com a sintaxe e a semântica.
- Em representações gráficas: Fluxograma, diagramas, etc.

Referências

1. P. Feofiloff. Algoritmos em Linguagem C. Campus-Elsevier, 2009.
2. H. M. Deitel, P. J. Deitel. C - Como Programar (6a. ed.), Pearson Education, 2011.
3. B. W. Kernighan, D. M. Ritchie. The C Programming Language (2a. ed.), Prentice-Hall, 1988
4. J. L. Szwarcfiter, L. Markenzon. Estruturas de Dados e seus Algoritmos (3a. ed.), Editora LTC, 2010.
5. W. Celes, R. Cerqueira, J.L. Rangel. Introdução a Estruturas de Dados, Editora Campus, 2004.
6. N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C (3a. ed.), Editora Cengage Learning, 2011.
7. T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos - Teoria e Prática (3a. ed.), Editora Campus, 2012.
8. R. Sedgewick, K. Wayne. Algorithms (4a. ed.), Addison-Wesley, 2011.
9. H. Schildt. C - Completo e Total, Makron Books, Makron Books, 1996.
10. Kelley and I. Pohl. Book on C: Programming in C (4a. ed.), Pearson, 2007.
11. Farrer H. et ali. Algoritmos Estruturados., Editora Guanabara S. A., Rio de Janeiro, 1985.
12. Guimarães, Angelo M & Lages, Newton A. de C. Algoritmos e Estrutura de dados. LTC - Livros Técnicos e Científicos Editora S.A.; 216 – 1985
13. Tremblay, J. P; Bunt R. Ciência dos Computadores - Uma Abordagem Algorítmica. Editora McGraw Hill do Brasil, são Paulo, 1983.
14. Villas, Marcos V, & Villasboas, Luiz F.P. Programação Conceitos, Técnicas e Linguagens. Ed. Campus; 195 p. 1988.
15. Filho, José Vanni, Construção de Algoritmos. PUC-Rio . 4a edição – Setembro de 1995

Um algoritmo é um processo sistemático para a resolução de um problema. O desenvolvimento de algoritmos é particularmente importante para problemas a serem solucionados em um computador, pela própria natureza do instrumento utilizado. Existem dois aspectos básicos no estudo de algoritmos: a *correção e a análise*. O primeiro consiste em verificar a exatidão do método empregado, o que é realizado através de uma prova matemática. A análise visa à obtenção de parâmetros que possam avaliar a eficiência do algoritmo em termos de tempo de execução e memória ocupada. A análise é realizada através de um estudo do comportamento do algoritmo.

Um algoritmo computa uma saída, o resultado do problema, a partir de uma entrada, as informações inicialmente conhecidas e que permitem encontrar a solução do problema. Durante o processo de computação, o algoritmo manipula dados, gerados a partir de sua entrada. Quando os dados são dispostos e manipulados de uma forma homogênea, constituem um tipo abstrato de dados. Este é composto por um modelo matemático acompanhado por um conjunto de operações definido sobre esse modelo. Um algoritmo é projetado em termos de tipos abstratos de dados. Para implementá-los numa linguagem de programação, é necessário encontrar uma forma de reprensetar-los nessa linguagem, utilizando tipos e operações suportadas pelo computador. Na representação do modelo matemático, emprega-se uma estrutura de dados.

As estruturas diferem umas das outras pela disposição ou manipulação de seus dados. A disposição dos dados em uma estrutura obedece a condições preestabelecidas e caracteriza a estrutura. O estudo de estruturas de dados não pode ser desvinculado de seus aspectos algorítmicos. A escolha correta da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente. O conhecimento de princípios de complexidade computacional é portanto, requisito básico para se avaliar corretamente a adequação de uma estrutura de dados.

A construção de um algoritmo para a resolução de um problema deve seguir uma metodologia seguindo os seguintes critérios:

1 – Entender o problema;

2 – Formar um esboço da solução;

3 – Definir a estrutura de dados necessária;

4 – Fazer o projeto, revendo os passos originais, detalhados;

5 – Se o algoritmo estiver suficientemente detalhado, testar com um conjunto de dados significativos

6 – Implementar em uma linguagem de programação

Mesmo para um problema simples existem diversas soluções e algumas soluções são melhores do que outras sob algum critério. Além disso, alguns problemas são casos particulares de outros similares, logo, através da solução de uma classe de problemas é possível achar a solução de um problema específico. A escolha da melhor solução depende de vários fatores. Por exemplo, se a resposta deve ser exata ou não ou se os conhecimentos prévios necessários estão disponíveis, e assim por diante. Para análise da qualidade de um algoritmo cinco fatores relevantes devem ser considerados:

1 – Clareza: Refere-se à facilidade de leitura do algoritmo. Um algoritmo claro permite que outros programadores seguir sua lógica e seu entendimento

2 – Simplicidade: Refere-se à precisão da estrutura do algoritmo quanto ao seu objetivo.

3 – Eficiência: Refere-se à velocidade de processamento e a utilização de espaço (memória) utilizado. Um algoritmo deve ter performance suficiente para atender à necessidade do problema e do usuário de forma mais rápida e utilizando menos recurso possível, dentro das limitações do problema

4 – Modularização: Durante a fase de projeto, a solução do problema vai sendo fatorada em soluções de subproblemas em módulos com sub-funções claramente definidas. O conjunto desses módulos e a interação entre eles permitem a resolução do problema de forma mais simples e clara.

5 – Generalidade: É interessante que um algoritmo seja genérico de forma a permitir a reutilização de seus componentes em outros projetos.

Pode-se realizar a análise de um algoritmo particular, na qual se verifica o custo de um determinado algoritmo na resolução do problema. Para isso, características importantes do algoritmo em questão devem ser investigadas, geralmente uma análise do número de vezes que cada parte do algoritmo deve ser executada (complexidade de tempo), seguida do estudo da quantidade de memória necessária (complexidade de espaço).

Também é possível fazer a análise de uma classe de algoritmos, na qual se verifica qual é o algoritmo de menor custo possível para resolver um problema. Para isso, toda uma classe de algoritmos é estudada com o objetivo de identificar um que seja o melhor possível. Isso significa colocar limites para a complexidade computacional dos algoritmos pertencentes à classe.

Noções de Complexidade: Complexidade temporal e espacial, notação assintótica, crescimento de funções, problemas P e NP

A complexidade de um algoritmo reflete o esforço computacional requerido para executá-lo sobre um conjunto de entradas (tamanho dos dados de entrada). Esse esforço computacional mede a quantidade de trabalho, em termos de tempo de execução ou da quantidade de memória requerida. Para medir esse esforço computacional, duas medidas de desempenho são consideradas: A complexidade espacial e a complexidade temporal que medem respectivamente o espaço (quantidade de memória) e o tempo (velocidade) requeridos por um algoritmo para sua execução.

A complexidade espacial, usa como medida de desempenho a quantidade de memória necessária para execução de um algoritmo em um problema de tamanho *n*. A memória usada por um programa, bem como o tempo requerido para a sua execução, depende muito da implementação do algoritmo. Entretanto, algumas conclusões sobre espaço utilizado podem ser tiradas, examinando-se o algoritmo. Um programa requer uma área para guardar suas instruções, suas constantes, suas variáveis e os dados. Podem também utilizar uma área de trabalho para manipular os dados e guardar informações para levar adiante a computação. Os dados podem ser representados de várias formas, as quais requerem uma área maior ou menor de memória. Se os dados têm uma representação natural, por exemplo, como uma matriz, são considerados para análise de espaço somente o espaço extra, além do espaço utilizado para guardar as instruções do programa e os dados. Porém, se os dados podem ser representados de várias formas, como um grafo, o espaço requerido para guarda-os também de ser levado em conta.

A complexidade temporal, usa como medida de desempenho o tempo necessário para a execução de um algoritmo em um problema de tamanho *n*. É possível determinar o tempo de execução através de métodos empíricos, isto é, obter o tempo de execução através da execução propriamente dita do algoritmo, considerando entradas diversas. Esse método nem sempre é simples de concretizar e muitas vezes não é conclusivo pois não só depende da implementação, como também das condições de processamento, compilação e linguagem de programação. Uma outra forma de se determinar o tempo de execução é utilizando-se de métodos analíticos. O objetivo desses métodos é determinar uma expressão matemática que traduza o comportamento do tempo de um algoritmo. Ao contrário do método empírico, o analítico visa aferir o tempo de execução de forma independente do computador utilizado, da linguagem e compiladores empregados e das condições locais de processamento.

A tarefa de obter uma expressão matemática para avaliar o tempo de um algoritmo em geral não é simples, mesmo considerando uma expressão aproximada. Logo, é necessário fazer simplificações, uma delas é em relação à quantidade de dados a serem manipulados pelo algoritmo onde é considerado uma massa de dados suficientemente grande, algoritmos com entradas de dados reduzidas são desconsiderados. Nesse caso, somente o comportamento assintótico será avaliado, ou seja, a expressão matemática fornecerá valores de tempo que serão válidos unicamente quando a quantidade de dados correspondente crescer o suficiente. Outra simplificação desconsidera constantes aditivas ou multiplicativas na expressão matemática obtida.

O processo de execução de um algoritmo pode ser dividido em etapas elementares denominadas **passos**. Cada passo consiste na execução de um número fixo de operações básicas cujos tempos de execução são considerados constantes. A operação básica de maior frequência de execução no algoritmo é denominada **operação dominante**. Como a expressão do tempo de execução do algoritmo será obtida a menos de constantes aditivas e multiplicativas, o número de passos de um algoritmo pode ser interpretado como sendo o número de execuções da operação dominante. Por exemplo:

1 - Soma de vetores para i de1até N faça S[i] ← X[i] + Y[i] fim para Número de passos = número de somas (N)	2 - Soma de matrizes para i de 1 até N faça para j de 1 até N faça C[i,j] ←A[i,j] + B[i,j] fim para fim para Número de passos = número de somas (N*N)

Logo, o número de passos de um algoritmo constitui a informação de que se necessita para avaliar o seu comportamento de tempo. Assim, um algoritmo de um único passo (ex. 1) possui tempo de execução constante, já um algoritmo de mais de um passo (ex. 2) possui um tempo quadrático (N*N).

A expressão matemática de avaliação do tempo de execução de um algoritmo é definida como sendo uma função que fornece o número de passos efetuados pelo algoritmo a partir de uma certa entrada. Sua complexidade pode ser qualificada quanto ao seu comportamento como: **Polinomial** onde a medida que N aumenta, o fator que estiver sendo analisado aumenta linearmente; ou **Exponencial** onde a medida que N aumenta, o fator que estiver sendo analisado aumenta exponencialmente. A ordem de crescimento do tempo de execução de um algoritmo fornece uma caracterização simples de eficiência do algoritmo. Este comportamento do crescimento de funções é chamado de crescimento assintótico.

Existem três escalas de complexidade usadas para analisar os algoritmos, cujo a função f(N) retorna a complexidade de um algoritmo com entrada de N elementos:

- **Complexidade de melhor caso:** É definido pela letra grega Ω (Ômega). Corresponde ao menor tempo de execução de uma entrada de tamanho N. Essa escala especifica um limite inferior para valores assintóticos da função f(N). Exemplo: Se tivermos uma lista de N números e quisermos encontrar algum deles assume-se que a complexidade no melhor caso é **f(N) = Ω (1)**, pois assume-se que o número estaria logo na cabeça da lista.
- **Complexidade de caso médio:** É definido pala letra grega θ (Theta). Retrata o comportamento médio do algoritmo, quando se consideram todas as entradas possíveis e as respectivas probabilidades de ocorrência (esperança matemática). Das três escalas, é a mais difícil de se determinar, Deve-se obter a média dos tempos de execução de todas as entradas de tamanho N, ou baseado em probabilidade de determinada condição ocorrer. Exemplo: Considerando o problema da lista anterior, a complexidade média é **P(1) + P(2) + ... + P(N)**. Para calcular a complexidade média, basta conhecer as probabilidades de P*i*: **P*i* = 1/N, 1 <= i <= N**, logo temos que: **P(1/N) + P(2/N) + ... + P(N/N) => 1/N(1+2+...+N) => 1/N[N(N+1)/2] => f(N) = θ[N+1)/2].**
- **Complexidade de pior caso:** Representado pela letra grega O (ômicron). É o método mais fácil de se obter. Baseia-se no maior tempo de execução sobre todas as entradas de tamanho N. . Essa escala especifica um limite superior para valores assintóticos da função f(N). Exemplo: Se tivermos uma lista de N números e quisermos encontrar algum deles, a complexidade no pior caso é **O (N)**, pois assume-se que o número estaria, no pior caso, no final da lista.

Acontece que um problema pode ter mais de um algoritmo para resolve-lo e em geral, deseja-se selecionar o melhor, isto é, o mais eficiente, aquele com menor tempo de execução no pior dos casos **O(N)**. Para saber qual é a complexidade de um determinado algoritmo, foram definidas Classes de Problemas (ou ordem de complexidade) de acordo com o parâmetro que afeta o algoritmo de forma mais significativa em relação ao seu tempo de execução. Essas classes são:

Complexidade constante – O(1): Independe do tamanho N de entradas. É o único em que as instruções dos algoritmos são executadas em um tamanho fixo de vezes. Exemplo: Criar uma lista vazia.

Complexidade linear – O(N): Uma operação é realizada em cada elemento de entrada. Típico quando algum processamento é feito para cada dado de entrada. Nesse caso, Cada vez que N dobra de tamanho, o tempo de execução dobra. Exemplo: pesquisa de elementos em uma lista.

Complexidade Logarítmica – O (logN): Ocorre tipicamente em algoritmos que dividem o problema em problemas menores. Pode-se considerar o tempo de execução como menor que uma constante grande. Para dobrar o valor de logN temos de considerar o quadrado de N. Exemplo: O algoritmo de Busca Binária.

Complexidade O(NlogN): Ocorre tipicamente em algoritmos que dividem o problema em problemas menores, porém juntando posteriormente a solução dos problemas menores. A maioria dos algoritmos de ordenação externa são de complexidade *logarítmica* ou *N Log N.*

Complexidade Quadrática – O(N^2): Itens são processados aos pares, geralmente com um *loop* dentro do outro. Sempre que n dobra, o tempo de execução é multiplicado por 4. Prático apenas em pequenos problemas, exemplo: produto de vetores.

Complexidade cúbica – O(N^3): Itens são processados três a três, geralmente com um *loop* dentro do outros dois. Sempre que n dobra, o tempo de execução fica multiplicado por 8. Úteis apenas para resolver pequenos problemas, exemplo: produto de matrizes.

Complexidade exponencial – O(2^N): Ocorrem na solução de problemas quando se usa força bruta para resolvê-los. Geralmente não são úteis sob o ponto de vista prático. Para $N = 20$, $2^N = 1$ milhão; se n duplica, o tempo passa a ser o quadrado.

A complexidade por ser vista como uma **propriedade do problema.** Logo, é possível obter uma medida independente do tratamento dado ao problema ou do caminho percorrido na busca da solução, portanto independente do algoritmo. Problemas tratáveis são aqueles que o tempo de execução da maioria dos algoritmos é limitado por alguma função polinomial do tamanho da entrada. Problemas intratáveis são aqueles em que o tempo de execução da maioria dos algoritmos é limitado por alguma função não polinomial (exponencial) do tamanho da entrada. Dessa forma é possível classificar os problemas como:

Classe P - consiste nos problemas que podem ser **resolvidos** em tempo Polinomial (Problemas tratáveis). Alguns exemplos de algoritmos de tempo polinomial:

- O algoritmo de ordenação quicksort em N inteiros executa, no máximo, A N^2 operações para algumas constantes A. Assim, é executado em tempo O (n^2) e é um algoritmo de tempo polinomial.
- Todas as operações aritméticas básicas (adição, subtração, multiplicação, divisão e comparação) pode ser feito em tempo polinomial.
- Emparelhamento máximo em grafos podem ser encontrados em tempo polinomial.

Classe NP se referem a algoritmos "não determinísticos polinomiais" no tempo. Ela consiste nos problemas que podem ser **verificados** em tempo polinomial (Problemas Intratáveis). Dada uma entrada, é possível verificar se ela corresponde a uma solução do problema em tempo polinomial. Alguns exemplos de algoritmos NP:

- O problema do isomorfismo de grafos: determinar se dois grafos podem ser desenhados de forma idêntica.
- O problema do caixeiro viajante, onde queremos saber se existe uma rota de qualquer comprimento que passe através de todos os nodos de uma certa rede.
- O Problema de Roteamento de Veículos, onde queremos alocar uma frota veicular para o atendimento de um conjunto de consumidores. É semelhante ao problema do caixeiro viajante, mas possui mais de um veículo para entregas e coletas de mercadorias.
- O problema da factibilidade lógica, onde nós queremos conhecer se uma certa formula em uma proposição lógica com variáveis lógicas pode ser satisfeita ou não.

Classe NP-completo são problemas NP que possuem a característica de que se um deles puder ser resolvido em tempo polinomial então todo problema NP- Completo terá uma solução em tempo polinomial. Um aspecto interessante é que vários problemas NP-Completos são, a principio, semelhantes a problemas que tem algoritmos de tempo polinomial. Um exemplo de problema NP-completo é o problema de isomorfismo de subgrafos.

Recursão

Recursão é um método de programação no qual uma função pode chamar a si mesma. O termo é usado de maneira mais geral para descrever o processo de repetição de um objeto de um jeito similar ao que já fora mostrado. Muitos problemas em computação tem a propriedade de que cada instância sua contém uma instância menor do mesmo problema. A Recursividade permite descrever algoritmos de forma mais clara, concise e simplificadas. Um método comum de simplificação é dividir o problema em subproblemas do mesmo tipo. Como técnica de programação, este método é conhecido como dividir e conquistar e é a chave para a construção de muitos algoritmos importantes, bem como uma parte fundamental da programação dinâmica.

Normalmente, as funções recursivas são divididas em duas partes: A chamada recursiva e a condição de parade. A chamada recursiva pode ser direta (mais comum) ou indireta (A chama B que chama A novamente). A condição de parada é fundamental para evitar a execução de loops infinitos. Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um Registro de Ativação na Pilha de Execução do programa. O registro de ativação armazena os parâmetros e variáveis locais da função bem como o "ponto de retorno" no programa ou subprograma que chamou essa função. Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Um exemplo clássico de recursão é a definição do cálculo do fatorial de um número, dada aqui no algoritmo a seguir:

```
ALGORITMO: Fatorial(n)
ENTRADA: inteiro n
SAÍDA: fatorial de n
REQUISITOS: n >= 0
SE n <= 1
    RETORNE 1
SENÃO
    RETORNE n * Fatorial(n-1)
```

A função chama a si mesma recursivamente em uma versão menor da entrada (n - 1) e multiplica o resultado da chamada por n, até que alcance o caso base, de modo análogo à definição matemática de fatorial. Como pode-se notar pela primeira condição, todo processo recursivo necessita de uma **condição de parada** para evitar um *loop* infinito. Neste caso, quando a função fatorial atinge o valor menor ou igual a 1 ele passa a retornar o valor de volta para a função.

A vantagem em se construer algoritmos recursivos está na redução do tamanho do código fonte além de permitir descrever algoritmos de forma mais clara e consisa. No entanto, a recursão não é sempre a melhor opção. Em programas recursivos, hà uma redução do desempenho de execução devido ao tempo para o gerenciamento de chamadas. Além disso, pode haver uma dificuldade na depuração de programas recursivos, especialmente se a recursão for muito profunda. No exemplo do factorial, a complexidade de tempo do fatorial recursivo é O(n) e a complexidade de espaço também é O(n), devido a pilha de execução. Ja no fatorial não recursivo a complexidade de espaço é O(1). Portanto, a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos.

Para montar um algoritmo recursive é necessario primeiramente definir pelo menos um caso básico (condição de terminação). Depois, Quebra-se o problema em problemas menores, definindo o(s) caso(s) com recursão(ões). Em seguida, é necessário fazer o teste de finitude, isto é, certificar-se de que as sucessivas chamadas recursivas levam obrigatoriamente, e numa quantidade finita de vezes, ao(s) caso(s) básico(s).

Algoritmos recursivos aparecem bastante na prática, podemos citar Torre de Hanoi, Algoritmos de ordenação (Quicksort, Mergesort), sequência de fibonaci, implementação de lista encadeada, etc.

Recursividade vale a pena para Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha como os algoritmos de ordenação (quicksort) ou caminho em árvore (pesquisa, backtracking). Nesses algoritmos, basicamente há duas chamadas recursivas onde cada uma resolve a metade do problema (dividir para conquistar). Nesse caso, a recursividade é essencial pois produz uma solução eficiente do problema devido à sua decomposição em problemas menores. Além disso, ela não produz recomputação execessiva (como na séria de Fibonacci) pois trabalham porções diferentes do problema.

Técnicas de Projeto de Algoritmo: Programação Dinâmica, Divisão e Conquista, Algoritmo Guloso e Algoritmos Aproximativos

Para a maioria dos problemas, as técnicas simples de Análise Estruturada ou de Análise Orientada a Objetos aliadas a um pouco de bom senso e conhecimentos sobre Estruturas de Dados são suficientes para elaborar um algoritmo para resolver este problema. Por exemplo: criar um algoritmo para ler um arquivo, criar um algoritmo da receita de um bolo. Existem alguns problemas porém, onde o caminho a ser seguido para a solução não é tão direto, ou requerem abordagens mais adequadas de forma a otimizar o desempenho de sua solução. Por exemplo: Criar um algoritmo eficiente para ordenar endereços, criar um algoritmo para distribuir tarefas entre máquinas de uma empresa afim de reduzir o tempo ocioso delas. Estes são exemplos de problemas onde é possível aplicar técnicas de projeto de algoritmos mais elaboradas para resolvê-los. Dentre essas técnicas podemos citar o método guloso, a divisão e conquista, a programação dinâmica e algoritmos aproximativos.

O método guloso é aplicado em problemas de otimização, isto é, problemas que envolvem pesquisa em um conjunto de ***configurações*** para encontrar uma que minimize ou maximize uma ***função objetivo*** definida para essas configurações. A fórmula genérica do método guloso não poderia ser mais simples. Para resolver um problema de otimização, é feito uma seqüência de escolhas. Essa seqüência se inicia a partir de uma configuração bem entendida e então, iterativamente, escolhe a que parece ser a melhor do conjunto das possíveis. Ou seja, é feito uma escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global. Exemplos de problemas que utilizam o método guloso:

Problema do troco – No Brasil. Temos moedas de 1 real, 50, 25, 5 e 1 centavos. O problema do troco consiste em pagar um troco com a menor quantidade possível de moedas. Por exemplo, se o troco for de R$2,78, a solução ótima seria: 2 moedas de 1 real, 1 moeda de 50 centavos, 1 moeda de 25 centavos e 3 moedas de 1 centavo. Esse problema é considerado guloso porque a cada passo escolhe a maior moeda possível, uma vez escolhida a moeda, esta não será trocada.

Problema do empacotador – Deseja-se otimizar a colocação de produtos em sacolas de farmácia ou supermercado. Onde em cada iteração é colocado na sacola o primeiro item que aparecer, verificando a cada passo a capacidade da sacola para que não seja ultrapassada.

Outros exemplos são os algoritmos para a **Árvore Expandida de Custo Mínimo de Kruskal de de Dijkstra**. Em ambos os algoritmos, escolhe-se o próximo vértice somente pelo critério "*qual é o vértice livre que está mais próximo ?*".

Em geral o algoritmo guloso tem cinco componentes:

1. Um conjunto candidato, a partir do qual é criada uma solução.
2. Uma função de seleção, que selecciona o melhor candidato para ser adicionado à solução.
3. Uma função de viabilidade, que é usada para determinar se um candidato pode ser utilizado para contribuir para uma solução.
4. Uma função objetivo, que atribui um valor a uma solução, ou uma solução parcial.
5. Uma função de solução, que indicará quando nós descobrimos uma solução completa.

A abordagem gulosa nem sempre tende para a solução ótima. Existem vários problemas, entretanto, em que funciona bem, e tais problemas são os ditos possuidores da propriedade da ***escolha gulosa***. Essa propriedade diz que a solução global ótima é alcançada a partir de um conjunto de escolhas locais ótimas (isto é, escolhas que são as melhores dentre um conjunto de possibilidades disponíveis no momento), começando por uma configuração bem definida.

Vantagens

- Simples e de fácil implementação;
- Algoritmos de rápida execução;
- Podem fornecer a melhor solução (estado ideal).

Desvantagens

- Nem sempre conduz a soluções ótimas globais. Podem efetuar cálculos repetitivos.
- Escolhe o caminho que, à primeira vista, é mais econômico.
- Pode entrar em *loop* se não detectar a expansão de estados repetidos.

- Pode tentar desenvolver um caminho infinito.

A técnica de ***divisão e conquista*** envolve a solução de um problema computacional particular dividindo-o em um ou mais subproblemas de menor tamanho, resolvendo cada subproblema recursivamente e, então, "juntando" ou "casando" as soluções dos subproblemas para produzir a solução do problema original. A Divisão e Conquista emprega modularização de programas e frequentemente conduz a um algoritmo simples e eficiente. Esta técnica é bastante utilizada em desenvolvimento de algoritmos paralelos, onde os subproblemas são tipicamente independentes um dos outros, podendo assim serem resolvidos separadamente. Exemplos clássicos de problemas que utilizam a técnica de divisão e conquista são os problemas de busca de elementos, ordenação de lista, o problema da torre de Hanoi, etc. Exemplo de algoritmos são: Busca binária, SelectionSort, MergeSort, QuickSort, Fibonacci recursivo, etc.

A técnica soluciona o problema através de três fases:

1. **Divisão**: o problema maior é dividido em problemas menores e os problemas menores obtidos são novamente divididos sucessivamente de maneira recursiva.
2. **Conquista**: o resultado do problema é calculado quando o problema é pequeno o suficiente.
3. **Combinação**: o resultado dos problemas menores são combinados até que seja obtida a solução do problema maior

Vantagens

- Indicado para aplicações que tem restrição de tempo;
- Algoritmos de fácil implementação;
- Simplifica problemas complexos.

Desvantagens

- Necessita de memoria auxiliary;
- Repetição de subproblemas;
- Tamanho da pilha (número de chamadas recursivas e/ou armazenadas pode causar estouro de memoria

Problemas que utilizam esta técnica podem tirar proveito de máquinas com múltiplos processadores pois a fase de divisão em problemas menores proporciona uma divisão natural do trabalho. Cada um dos problemas menores obtidos pode ser calculado separadamente em um processador sem depender dos demais.

A técnica de programação dinâmica é similar à técnica de divisão e conquista na medida em que pode ser aplicada a uma grande gama de problemas. A particularidade desta metodologia diz respeito à divisão do problema original: o problema é decomposto somente uma vez e os subproblemas menores são gerados antes dos subproblemas maiores; dessa forma, esse método é claramente ascendente (problemas recursivos são descendentes). Na Programação Dinâmica resolvem-se os problemas de pequena dimensão e guardam-se as soluções em tabelas dinâmicas, a fim de evitar redundância de cálculo, ou seja, uma solução parcial obtida somente é calculada uma única vez. Este procedimento é importante porque, diferentemente da técnica de divisão e conquista, aqui cada subproblema é dependente de pelo menos um outro. A solução final é obtida combinando as soluções dos problemas menores. Pode-se dizer que a Programação Dinâmica é uma maneira esperta de transformar recursão em iteração, daí ser chamada de Otimização Recursiva.

Um problema clássico para a aplicação desta técnica é a ordem de produto de matrizes, que ilustra bem a técnica de programação dinâmica por meio de determinar a melhor forma de avaliar o produto de várias matrizes. Por exemplo, se multiplicar M = M1*M2*M3*M4 com dimensões 200x2, 2x30, 30x20, e 20x5 respectivamente, teríamos as seguintes possibilidades de realizar o produto:

Ordem de multiplicação	Computação do Custo	Custo
M1*((M2*M3)*M4)	2*30*20 + 2*20*15 + 200*2*5	3.400
(M1*(M2*M3))*M4	2*30*20 + 200*2*20 + 200*20*5	29.200
(M1*M2)*(M3*M4)	200*2*30 + 30*20*5 + 200*30*5	45.000

Passos gerais de um algoritmo baseado em Programação Dinâmica:

1. Dividir o problema em subproblemas;
2. Computar os valores de uma solução de forma bottom-up e armazená-los (memorização);
3. Construir a solução ótima para cada subproblema utilizado os valores computados.

Vantagens

- Pode ser utilizada em um grande número de problemas de otimização discreta;
- Não necessita de muita precisão numérica;
- Útil para aplicar em problemas que exigem teste de todas as possibilidades.

Desvantagens

- Necessita de grande espaço de memória;
- A complexidade espacial pode ser exponencial

A técnica de programação dinâmica é usada sobretudo quando a solução ótima de um subprobelma pode ser composta pelas soluções ótimas dos subproblemas ou quando o cálculo da solução ótima implica muitas vezes no cálculo do mesmo subproblema.

Algoritmos de aproximação são geralmente associados com problemas NP-difíceis, já que estes problemas não podem ser resolvidos em tempo polinomial. Também em alguns problemas resolvidos em tempo polinomial no qual o tamanho da entrada pode fazer com que mesmo algoritmos polinomiais sejam custosos, estão sendo cada vez mais usados algoritmos de aproximação. Neste caso procura-se por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima. O Algoritmo aproximado gera soluções aproximadas dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado (comportamento monitorado sob o ponto de vista da qualidade dos resultados).

Nem todos os algoritmos de aproximação são usuais na prática. Eles costumam usar estruturas de dados complexos ou sofisticadas técnicas algorítmicas que dificultam sua implementação. Além disso, alguns algoritmos de aproximação tem tempos de execução não viaveis, embora sejam de tempo polinomial, por exemplo, $O(n^{2000})$. No entanto, o estudo de alguns algoritmos mesmo muito caros podem fornecer conhecimentos valiosos.

Um exemplo clássico é o PTAS para o problema do caxeiro viajante euclidiano (PCV euclidiano, problema PCV utilizando distâncias euclidianas) concebido por Sanjeev Arora, que possui um tempo de execução de um ano, redefinindo os conceitos de algoritmo de tempo linear. Esses algoritmos são úteis em algumas aplicações onde os tempos de execução e de custo podem ser justificados, como por exemplo: biologia computacional, engenharia financeira, planejamento, transporte e gerenciamento de inventário.

Outra limitação da abordagem é que ela se aplica somente a problemas de otimização e não a problemas de decisão em essência como o problema da satisfatibilidade, embora muitas vezes é possível conceber versões de otimização de tais problemas. (Max SAT). Exemplo de algoritmos aproximados são: Algoritmos genéticos, Programação Genética, Simulated Annealing, Particle Swarm Optimization, Ant Colony Optimization. Etc.

Representação de Dados

O ser humano consegue guardar informações sob forma de imagens e cenas. Entretanto, o computador é uma máquina, sendo muito difícil construir circuitos para que ele guarde essas informações tal como o cérebro. Internamente o computador possui um modelo que representa a realidade, criando um modelo numérico e aritmético. Essas representações são de tipos variados como símbolos, textos, imagens, vídeos, som etc. Para isso, ele utiliza uma representação binária para fazer o armazenamento e manipulação dos dados. Dados neste caso podem ser programas armazenados, imagens, sons, textos, vídeos, dentre outros.

O computador é um equipamento eletrônico, portanto, só reconhece dois estados físicos: Ligado/ desligado ou presença/ausência de energia isso equivale ao nível de tensão elétrica que pode variar **entre** 0v e +5v. Para representar esses dois estágios o computador utiliza um sistema de numeração chamado binário cujos algarismos são representados pelo número 0 e 1, esses números individualmente são chamados de ***Bi****nary Digi****t*** **- BIT.**

Para representar o estado de ligado o computador utiliza o BIT 1, consequentemente o BIT 0 representa o estado inverso. Contudo, utilizando apenas dois bits o computador não conseguiria representar todas as letras, símbolos e números do mundo real. Mesmo realizando combinações de diferentes posições, o máximo de possibilidade alcançada seriam quatro, como mostra a tabela abaixo.

1	**1**
1	**0**
0	**1**
0	**0**

Outras bases de representações utilizadas para facilitar a escrita são as bases Octal (8 dígitos) e Hexadecimal (16 dígitos). Essas bases não são representadas internamente pelo computador. Ambos são utilizados pelos tradutores com o propósito de gradativamente realizar a codificação para um nível mais baixo de linguagem.

Base 10	Base 2	Base 8	Base 16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Para aumentar o número de possibilidades e conseqüentemente o número de representações do mundo real, em 1960 a IBM desenvolveu o *Extended Binary Coded Decimal Interchange Code*- **EBCDIC,** derivado do antigo *Binary-coded decimal*– **BCD (**uma combinação de 4 bits também conhecida como nibble). Com o EBCIDIC ficou convencionado que os dados seriam representados utilizando uma combinação de 8 bits denominada **BYTE.** Esse conjunto de bits passou a ser mundialmente utilizado nos sistemas informatizados. Com 8 bits é possível reprentar 256 combinações numéricas, todas a letras do alfabetos , além de símbolos. Para reprentar todas essas cominações foi desenvolvido o acrônimo para *American Standard Code for Information Interchange*- ***ASCII,*** uma tabela com a relação dos dígitos binários e suas respectivas representações do alfabeto

inglês, além todos os símbolos possíveis de serem representados em sistema informatizado. A tabela abaixo ilustra parte do código ASCII e suas representações.

Para formar palavras, imagens e outras estruturas mais complexas são necessarios milhoões de Bytes. Então, para facilitar a quantificação desses *bytes* foram desenvolvidas representações que equivalem a mudanças de unidades como 1.000, 1.000.000 e assim sucessivamente.

Nome	**Valor aproximado**
8 Bits	1 Byte
1024 Bytes	1 KB
1024 KiloByte	1 MB
1024 MegaByte	1 GB
1024 GigaByte	1 TB
1024 TeraByte	1 PB

Os bytes são usados para representar caracteres, números, figuras, ou qualquer outro tipo de dado armazenado ou processado em um computador. Na maioria dos códigos alfanuméricos cada caractere é representado através de um byte. Por exemplo, no código ASCII a letra 'A' é representada pelo byte "0100 0001". Uma seqüência de caracteres é expressa por uma cadeia de bytes sucessivos. Nem todos os tipos de códigos utilizam os 8 bits de um byte para a representação de caracteres.

Os primeiros códigos utilizados foram os de 6 bits, que permitiam a representação de 26 = 64 caracteres que correspondem a:

- 26 letras maiúsculas.
- 10 algarismos (0 1 2 3 4 5 6 7 8 9).
- 28 caracteres chamados especiais, incluindo SP (caractere em espaço em branco).

Com o desenvolvimento das linguagens de programação de alto nível começaram a ser utilizados os códigos de 7 bits que permitiam a representação de minúsculas e de caracteres cujo significado são ordens de controle para periféricos. Um exemplo desse tipo de códigos é o ASCII de 7 bits. O EBCDIC (Extended Binary Coded Decimal Interchange Code) é uma codagem de caracteres de 8 bits (Figura 3) e se trata de um padrão proprietário desenvolvido pela IBM. Posteriormente, foi desenvolvido uma extensão do conjunto ASCII básico. Os caracteres ASCII estendido usam 8 bits para representar os caracteres. Os caracteres extras representam caracteres de línguas mortas e caracteres especiais para desenhas figures.

Para representar números positivos, utiliza-se normalmente o valor do próprio número binário. Por exemplo, o número 6 é representado por 0101 e o número 12 é representado por 1100.

Existem quatro maneiras de se representar números negativos: módulo e sinal (MS); complemento de 1 (C-1), complemento de 2 (C-2) e excesso de 2 elevado a (N-1). Dentre esses, destaca-se o MS, e o C-1:

- Módulo e Sinal (MS) – Onde o bit que está situado mais à esquerda representa o sinal, e o seu valor será 0 para o sinal + e um para o sinal -. Os bits restantes (N-1) representam o módulo do número. Por exemplo, supondo que exista a limitação de 8 bits (N=8), o valor 00101010 representa o número +42 e o valor 10101010 representa o número -42.
- Complemento de 1 (C-1) – Similar ao MS, onde o simétrico de um número positivo é obtido pelo complemento de todos os seus dígitos (trocando 0 por 1 e vice- versa), incluindo o bit de sinal. Por exemplo, supondo que exista a limitação de 8 bits (N=8), o valor 00101010 representa o número +42 e o valor 11010101 representa o número -42.

Uma vez que número de bits que representa um número real é limitado, os números reais sofrem truncamento na sua parte fracionária. É importante lembrar de que, por utilizar a vírgula flutuante, nem todos os números têm representação, razão pela qual estes números são representados de forma aproximada, acarretando pequenos erros de representação. Os números decimais ponto flutuante são representados na forma $Fx10^{E}$, onde F é a fração e E o expoente. Apenas a fração e o expoente são fisicamente representados em termos computacionais. A base 10 e o ponto decimal da fração são assumidos e não são mostrados explicitamente.

Os arquivos são conjuntos de bytes armazenados em alguma mídia (dispositivo de armazenamento permanente) que representam uma informação, como uma imagem, um texto ou uma música. A extensão do arquivo é dada em função do seu conteúdo. Por exemplo:

- TXT, HTM, BAT – representa um arquivo texto
- DOC – representa um documento do MS Word
- PCX, BMP, JPG, GIF, TIF – formatos de imagem
- MPG, AVI – formatos de vídeo
- XLS – planilha do MS Excel PPT – apresentação do MS PowerPoint
- EXE, DLL – programa executável

No arquivo, seus dados são armazenados em um conjunto de bytes e sua estrutura depende do seu formato/extensão.

Logo, existem diversas formas de representação de dados que são utilizadas e compreendidas pelo hardware dos sistemas, cabe aos programas e programadores a definição para o sistema de como cada dado será armazenado e manipulado. Vale ressaltar que o dado é a representação física de um evento no tempo e espaço que não agrega fundamento para quem o sente ou recebe, não podendo ser possível entender o que ele representa ou para que ele existe. Podemos ter como exemplo um número, se somente esse número for disponibilizado para alguém ou para o tempo e espaço, por alguém ou por um evento, não é possível saber o que ele significa ou o que ele representa, podendo representar qualquer coisa ou não representar nada. Porém no momento que existir uma agregação com outro dado ele passa a ser ou não uma informação.

Tipos de Dados primitivos e estruturados

Todo o trabalho realizado por um computador é baseado na manipulação das informações contidas em sua memoria. Estas informações podem ser classificadas em dois tipos:

– As **instruções**, que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados.

– Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

Em uma linguagem de programação, é importante classificar constantes, variáveis e valores gerados por expressões/funções de acordo com o seu **tipo de dados**. Um **tipo de dados** deve caracterizar o conjunto de valores a que uma constante pertence ou o conjunto de valores que pode ser assumido por uma variável ou gerado por uma expressão/função. Os tipos de dados são comumente classificados em **tipos de dados elementares (primitivos)** e **tipos de dados compostos (estruturados)**.

Os tipos de dados elementares, denominados tipos primitivos, não possuem uma estrutura sobre seus valores, ou seja, não é possível decompor o tipo primitivo em partes menores. Os tipos básicos são, portanto, indivisíveis. Por exemplo, uma variável do tipo lógico, somente pode assumir valores verdadeiro e falso. Dentre os tipos de dados primitivos em uma linguagem de programação, podemos citar:

- **Inteiros** – compreende ao conjunto de todos os números inteiros (negativo, zero e positivos). Ex.: 1, -5, 687. As operações permitidas para esse tipo de dado são: Soma, Subtração, Multiplicação, Divisão inteira (DIV), Resto da divisão (MOD), e operação de comparação (=, ≠, >, <, ≥, ≤)
- **Reais** – são os números racionais (positivos e negativos), isto é, números compostos por uma parte inteira e uma parte fracionaria. Ex.: 5.47, 3.14852, -8.75. As operações permitidas para esse tipo de dado são: Soma, Subtração, Multiplicação, Divisão (/), e operação de comparação (=, ≠, >, <, ≥, ≤)
- **Lógicos** – composto por apenas dois valores: **verdadeiro** e **falso** sendo que este tipo de dado poderá representar apenas um dos dois valores. As operações permitidas para esse tipo de dado são: OU (OR – conjunção), E (AND – disjunção), NÃO (NOT – negação), OU EXCLUSIVO (XOR – disjunção exclusiva).
- **Caracteres** – são os caracteres (ou uma sequência de caracteres) alfanuméricos, isto é, dígitos decimais (0...9), letras (a...z) e (A...Z) e sinais especiais (sinais de pontuação, símbolos, espaço em branco, etc.). Caracteres geralmente são armazenados em códigos (usualmente o código ASCII) e indicado através de aspas aspas (“ “). Ex.: “%”(32 na tabela ASCII), “@” (64 na tabela ASCII), “D”(68 na tabela ASCII). As operações permitidas para esse tipo de dado são de comparação (=, ≠)

Os tipos Primitivos (inteiro, real, caracter e lógico) não são suficientes para representar todos os tipos de dados. Geralmente são utilizados os tipos primitivos para construir outras estruturas de dados mais complexas.

Os tipos de dados estruturados permitem agregar mais do que um valor em uma variável, existindo uma relação estrutural entre os seus elementos. Os tipos de dados compostos são divididos em duas formas fundamentais: homogêneas (Arranjos) e heterogêneas (registros). As estruturas homogêneas são conjuntos de dados formados pelo mesmo tipo de dado primitivo. As estruturas heterogêneas são conjuntos de dados formados por tipos de dados primitivos diferentes (campos do registro) em uma mesma estrutura.

Dentre os tipos de dados compostos em uma linguagem de programação, podemos citar:

- **Arranjos** (estrutura vetorial/matricial) – Composta de um número fixo de elementos do mesmo tipo com índices variando de a_1 até a_n. Ex.: considerando os elementos a_{ij}, onde i = [1...n] e j = [1...m]. O conjunto de m x n variáveis pode ser representado por um arranjo bidimensional A, onde m é o numero de linhas e n o numero de colunas.
- **Registro** (estrutura) – estrutura de campos que pode ser utilizada para ligar componentes de diferentes tipos. Cada campo tem um nome próprio chamado de identificador do campo. Ex.:

```
Tipo Aluno estrutura
        nome Tipo caráter
        matrícula Tipo inteiro
        média Tipo real
```

Outros tipos de dados compostos são:

- **Arquvo** (file) – um conjunto de registros (linhas) que identificam a informação através de uma chave ou índice agilizando a manipulação das informações ex. (.TEXT, .DOC, .JPG, etc.)
- **Banco de dados** – Conjunto de arquivos ou tabelas com informações que podem ser compartilhadas com vários usuários e assim são relacionadas.

Existem ainda os tipos de dados definidos pelo usuário. São também tipos de dados estruturados, construídos hierarquicamente através de componentes, os quais são de fato tipos de dados primitivos e/ou estruturados. Um tipo definido pelo usuário é construído por um conjunto de componentes, que podem ser de tipos diferentes, agrupados sob um único nome. Normalmente, os elementos desse tipo estruturado tem alguma relação semântica. Essa construção hierárquica de tipos é a maneira fornecida pelas linguagens de programação para estruturar os dados e manipulá-los de forma organizada.

Dentro desse contexto surgem as estruturas de dados que, por sua vez especificam conceitualmente os dados, de forma a refletir um relacionamento lógico entre os dados e o domínio do problema considerado. Além disso, as estruturas de dados incluem operações para manipulação dos seus dados, que também desempenham o

papel de caracterização do mínimo de problema considerado. Cabe lembrar que esse nível de conceitual de abstração não é fornecido diretamente pelas linguagens de programação, as quais fornecem os tipos de dados e os operadores que permitem a construção de uma estrutura de dados flexível para o problema que está sendo definido. A forma mais próxima de implementação de uma estrutura de dados é através dos tipos abstratos de dados.

Tipos Abstratos de Dados

Todo o trabalho realizado por um computador é baseado na manipulação das informações contidas em sua memoria. Estas informações podem ser classificadas em dois tipos:

– As **instruções**, que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados.

– Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

Em uma linguagem de programação, é importante classificar constantes, variáveis e valores gerados por expressões/funções de acordo com o seu **tipo de dados**. Um **tipo de dados** deve caracterizar o conjunto de valores a que uma constante pertence ou o conjunto de valores que pode ser assumido por uma variável ou gerado por uma expressão/função.

Uma outra perspectiva de se interpretar o conceito de **Tipos de Dados** diferente da perspectiva do que um computador pode fazer (interpretar os bits), é interpretar o que os programadores (usuários) desejam fazer (somar dois inteiros). O programador não se importa muito com a representação no hardware, mas sim com as operações e as estruturas que um tipo de dado possa representar. Este conceito de **Tipo de Dado** divorciado do hardware é chamado **Tipo Abstrato de Dado - TAD**.

Os TADs são estruturas de dados capazes de representar os tipos de dados que não foram previstos no núcleo das linguagens de programação e que, normalmente, são necessários para aplicações específicas. Essas estruturas são dividias em duas partes: os dados e as operações. A especificação de um TAD corresponde à escolha de uma boa maneira de armazenar os dados e à definição de um conjunto adequado de operações para atuar sobre eles.

A característica essencial de um TAD é a separação entre conceito e implementação, ou seja, existe uma distinção entre a definição do tipo e a sua representação. Um TAD é, portanto, uma forma de definir um novo tipo de dado juntamente com as operações que manipulam esse novo tipo de dado. As aplicações que utilizam esse TAD são denominadas clientes do tipo de dado.

Abstraída qualquer linguagem de programação, um **TAD** pode ser visto como um modelo matemático que encapsula um **modelo de dados** e um conjunto de **procedimentos** que atuam com exclusividade sobre os dados encapsulados. Em nível de abstração mais baixo, associado à implementação, esses procedimentos são implementados por subprogramas denominados operações, métodos ou serviços.

Estrutura de Dados é um método particular de se implementar um **TAD**. Uma estrutura de dado armazena dados na memoria do computador a fim de permitir o acesso eficiente dos mesmos. A implementação de um TAD escolhe uma estrutura de dados para representá-lo. Cada ED é construída dos tipos primitivos (inteiro,

real, char,...) ou dos tipos compostos (arranjo, registro,...) de uma linguagem de programação. Quando bem projetada, a **Estrutura de Dados** permite a manipulação eficiente, em tempo e em espaço, dos dados armazenados através de operações especificas

Qualquer processamento a ser realizada sobre os dados encapsulados em um TAD só poderá ser executada por intermédio dos procedimentos definidos no modelo matemático do TAD, sendo esta restrição a característica operacional mais útil dessa estrutura. Existem quatro operações básicas de processamento: a primeira operação a ser efetuada em um TAD é a **criação**. Depois, podemos realizar **inclusões** e **remoções** de dados. A operação que varre todos os dados armazenados num TAD é o **percurso**, podendo também ser realizada uma **busca** por algum valor dentro da estrutura.

Existem basicamente dois tipos de **Estruturas de Dados** que implementam um TAD:

1. **Estruturas lineares** mantém os itens de informação de forma independente de seus valores. A única informação utilizada pela estrutura é a posição do item; qualquer manipulação relativa ao conteúdo ou valor desse item é atribuição da aplicação. São exemplo de estruturas lineares: **Listas ordenadas, Pilhas, Filas**
2. **Estrutura não-lineares (associativas)** permitem o acesso a seus elementos de forma independente de sua posição, com base apenas em seu valor. São exemplos de estruturas não-lineares: **Árvores e Grafos.**

Uma razão importante para programar em termos de *TAD* é o fato de que os elementos da estrutura passiva do *TAD* são acessíveis somente através dos elementos da estrutura ativa.

Uma segunda razão, também importante, para programar em termos de *TAD* é o fato de que seu uso permite introduzir alterações nas estruturas definidas no nível de implementação — visando, por exemplo, aumento de eficiência — livre da preocupação de gerar erros no restante do programa. Tais erros não ocorrem porque a única conexão entre o *TAD* e o restante do programa é aquela constituída pela interface **imutável** dos algoritmos que implementam a estrutura ativa do *TAD*.

Por fim, pode-se dizer que um *TAD* bem construído pode tornar-se uma porção de código confiável e genérica, permitindo e aconselhando seu **reuso** em outros programas. Dessa forma, aumenta-se a produtividade na construção de programas e, sobretudo, garante-se a qualidade dos produtos gerados.

Listas lineares, Pilhas e Filas – Conceitos e Algoritmos

Dentre as estruturas de dados não primitivas, as listas lineares são as de manipulação mais simples. Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si. Ela pode constituir por exemplo, de informações sobre funcionários de uma empresa, sobre notas de alunos, itens de estoque, etc.

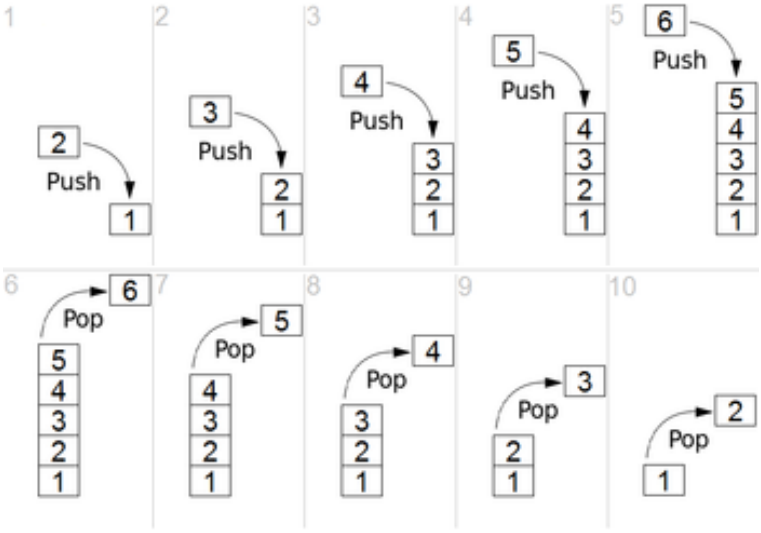
Pode-se definir uma lista como um conjunto de n elementos (n>=0) organizados de tal forma que a sua estrutura reflete diretamente as posições relativas dos elementos que compõem a lista. Formalmente, uma **lista linear** é um conjunto de n>=0 de nós L[1], L[2],...,L[n] tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear. Tem-se:

- Se n> 0, L[1] é o primeiro nó
- Para 1<k>=n, o nó L[k] é precedido por l[k-1]

As operações mais frequentes em listas são a busca, a inclusão e a remoção de um determinado elemento. Outras operações, também importantes são: alteração de um elemento da lista, a combinação de duas ou mais listas lineares, a ordenação, a determinação do primeiro/ultimo nó da lista, etc.

Casos particulares de listas são de especial interesse considerando a sua disciplina de acesso, ou seja, forma como os elementos de uma lista linear são acessados, inseridos e removidos. Se as inserções e remoções são permitidas apenas nas extremidades da lista, ela recebe o nome de *deque*; Se as inserções e as remoções são realizadas somente em um extremo, a lista é chamada *pilha* **(LIFO)**; Já se as inserções são realizadas em uma extremidade e as remoções na outra extremidade, a lista é denominada *fila* **(FIFO)**.

Uma pilha (**stack** em inglês) é um tipo abstrato de dado e estrutura de dados baseado no princípio de *Last In First Out* (**LIFO**), ou seja "o último que entra é o primeiro que sai" caracterizando um empilhamento de dados. Essa estrutura compõe-se de duas operações: *push* (empilhar) que adiciona um elemento no topo da pilha e *pop* (desempilhar) que remove o último elemento adicionado.



As operações associadas a uma pilha são: Criar uma pilha, empilhar um elemento, desempilhar um elemento e destruir uma pilha. Sua implementação pode ser dada por:

Criando uma Pilha:

```
struct pilha {
        int elementos[MAX];
        int topo;
}
struct pilha * cria(void) {
        struct pilha *p;
        p = malloc(sizeof(struct pilha));
        if(!p) {
                perror(NULL);
                exit(1);
        }
        /* IMPORTANTE: */
        p->topo = 0;
}
```

Empilhando um elemento:

```
struct pilha {
        int elementos[MAX];
        int topo;
}
Void empilha (struct pilha *p, int Numero)
{
        p->elementos[p->topo] = A;
        p ->topo = p->topo +1;
}
Void main () {
...
empilha (p, 7123);
...
}
```

Desempilhando um elemento

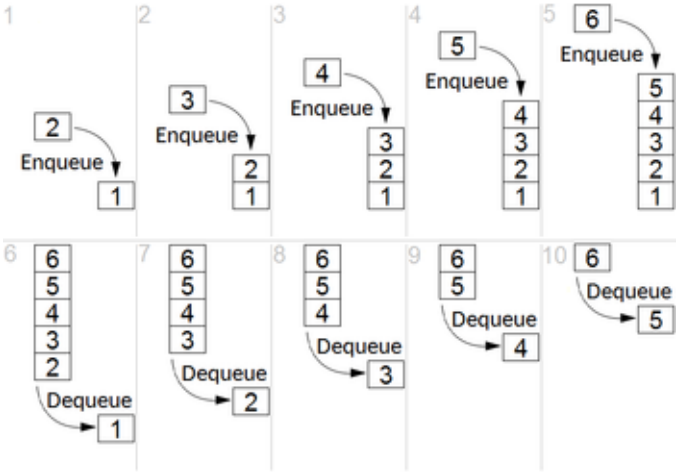
```
struct pilha {
        int elementos[MAX];
        int topo;
}
int desempilha (struct pilha *p) {
        p->topo = p->topo-1;
        return p->elementos[p->topo];
}
Void main () {
...
Int t = desempilha (p);
...
}
```

Destruindo uma pilha

```
struct pilha {
        int elementos[MAX];
        int topo;
}
int destroi (struct pilha *p) {
        free(pilha);
}
Void main () {
...
destroi (p);
...
}
```

O conceito de pilha é amplamente utilizado na informática, como, por exemplo, durante a execução de um programa, para o armazenamento de valores de variável local a um bloco e também para conter o endereço de retorno do trecho de programa que chamou a função ou procedimento atualmente em execução.

Uma Fila (queue em inglês**)** é um tipo abstrato de dado e estrutura de dados baseado no princípio de *First In First Out* (**FIFO**), ou seja o "o primeiro que entra é o primeiro que sai" caracterizando uma fila de dados onde a inserção é feita em uma extremidade e a eliminação em outra. Essa estrutura compoe-se de iniciar uma fila, inserir elementos no fim da fila (*Enqueue*) e remover o primeiro elemento da fila (*Dequeue*).



As operações associadas a uma Fila são: Criar uma fila, enfileirar um elemento, desenfileirar o primeiro elemento e destruir uma fila. Sua implementação pode ser dada por:

Criando uma Fila:

```
struct fila {
        int elementos[MAX];
        int primeiro, ultimo;
}
struct fila * cria(void) {
        struct fila *f;
        f = malloc(sizeof(struct fila));
        if(!f) {
                perror(NULL);
                exit(1);
        }
        f->primeiro = 0;
        f->ultimo = 0;
}
```

Enfileirando um elemento:

```
struct fila {
        int elementos[MAX];
        int primeiro, ultimo;
}
void enfilera (struct fila *f, int A) {
        f->elementos [f->ultimo] = A;
        f->ultimo += 1;
}
void main () {
...
        enfilera (f, 325);
...
}
```

Desenfileirando o primeiro elemento	**Destruindo uma fila**

struct fila {
int elementos[MAX];
int primeiro, ultimo;
}
int desenfilera (struct fila *f) {
int r = f->elementos [f->primeiro];
f->primeiro += 1;
Retunr r;
}
void main () {
...
int p = desenfilera (f);
...
}
	struct fila {
	int elementos[MAX];
	int primeiro, ultimo;
	}
	}
	void destroi(struct fila *f) {
	free(f);
	}
	void main () {
	...
	destroi (f);
	...
	}

As filas são amplamente utilizadas em programação para implementar *filas de espera*. Como exemplo de aplicação para filas, pode-se citar a fila de processos de um sistema operacional.

Ao desenvolver uma implementação para listas lineares, o primeiro problema que surge é: *como armazenar os elementos da lista?*

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a forma em que os espaços de memórias são alocados para as listas. **Na alocação estática**, o espaço de memória ocupado pelas variáveis é determinado no momento da compilação. Já na **alocação dinâmica**, o espaço de memória é alocado em tempo de execução.

Outra classificação das listas que está ligado diretamente ao tipo de armazenamento é em relação ao acesso aos elementos de uma lista linear. O acesso aos elementos de uma lista pode se dar de forma **sequencial** ou **encadeada**. No Acesso sequencial, assume-se que os elementos de uma lista são armazenados de forma consecutiva na memória. Já no acesso encadeado, os elementos de uma lista podem ocupar quaisquer área de memória não necessariamente consecutivas, eles possuem em seus nós encadeados por ponteiros que guardam o endereço do registro sucessor.

Logo, considerando o tipo de armazenamento e acesso de uma lista linear, temos as possíveis combinações:

- **Alocação Estática/Acesso Sequencial**
- Alocação Estática/Acesso Encadeado
- Alocação Dinâmica/Acesso Sequencial

- **Alocação Dinâmica/Acesso Encadeado**

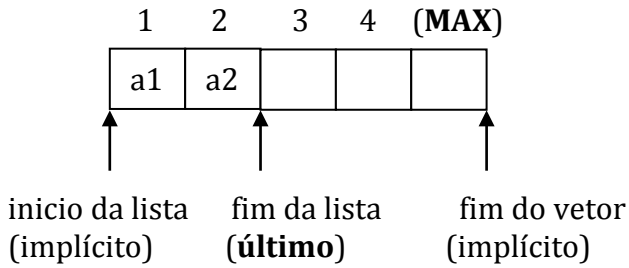
As implementações de Listas lineares mais estudas são:

Alocação Estática/Sequencial

Uma lista estática sequencial pode ser implementada através de um vetor. Exemplos de listas estáticas/sequenciais: lista telefônica, lista de alunos, entre outras. Listas lineares de alocação Estática/Sequencial são caracterizadas por:

- Armazenar itens em posições consecutivas na memória
- A lista pode ser percorrida em qualquer direção
- A inserção de um novo elemento pode ser realizada após o último elemento inserido com custo constante
- A inserção de um elemento na posição A[i] causa o deslocamento à direita do elemento A[i] de todos os elementos posteriores até o último
- A retirada de um elemento A[i] requer o deslocamento à esquerda do A[i+1] de todos os outros elementos posteriores, até o último.

Na implementação, os itens são armazenados em um vetor de tamanho suficiente para armazenar a lista. É definido um campo ***último*** contendo a posição após último elemento da lista. O i-ésimo item da lista está armazenado na i-ésima posição do vetor onde 0 <= i >= *ultimo.* É definido uma constante **MAX** com o tamanho máximo permitido na lista. *A* estrutura de uma lista:



Dentre as vantagens de se utilizar uma Lista Sequencial Estática podemos citar:

- Acesso direto indexado a qualquer elemento da lista
- Tempo constante para acessar o elemento i dependerá somente do índice

Já as desvantagens temos:

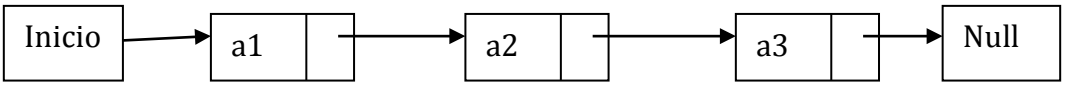
- Um custo de movimentação para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens no pior caso.
- Em aplicações em que não existe previsão sobre o crescimento da lista pode ser um problema pois, em algumas linguagens de programação, o tamanho máximo da lista tem de ser pré-estimado e definido em tempo de compilação.

Alocação Dinâmica/Encadeado

Em uma lista linear encadeada dinâmica, cada item é encadeado com o item seguinte mediante uma variável do tipo ponteiro que aponta para o próximo elemento da lisa. A alocação dinâmica/encadeada permite utilizar posições não contíguas de memória. É possível inserir e retirar elementos sem a necessidade de deslocar os itens seguintes da

lista. Para isso é definido uma célula cabeça (inicio da lista) para simplificar as operações com a lista.

A lista encadeada/dinâmica é representada por um ponteiro *inicio* que aponta para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim sucessivamente. O último elemento da lista aponta para NULL, sinalizando que não existe próximo elemento. Na implementação, é necessário definir explicitamente um ponteiro para o inicio da lista e um nó que contenha um campo para a informação e um campo para um ponteiro que aponte para o próximo nó.



Dentre as vantagens de se utilizar uma Lista Encadeada Dinâmica, podemos citar:

- Melhor utilização dos recursos de memória pois não é necessário pré-definir um espaço que talvez não seja utilizado
- Não requer movimentação de dados nas operações de inserção e remoção pois tais operações são feitas através dos ponteiros que apontam e/ou deixam de apontar para os elementos
- Não requer gerenciamento de espaço livre

Dentre as desvantagens podemos citar que o acesso aos elementos da lista não é indexado, os conteúdos são acessados através dos ponteiros.

Tabelas Hash: Funções hash; Tratamento de colisões; Complexidade

As tabelas de hash são um tipo de estruturação para o armazenamento de informação, de uma forma extremamente simples, fácil de se implementar e intuitiva de se organizar grandes quantidades de dados.

Possui como ideia central a divisão de um universo de dados a ser organizado em subconjuntos mais gerenciáveis.

A estruturação da informação em tabelas de hash, visa principalmente permitir armazenar e **procurar** rapidamente grande quantidade de dados.

As tabelas de hash são constituídas por 2 conceitos fundamentais:

- **Tabela de Hash**: Estrutura que permite o acesso aos subconjuntos.
- **Função de Hashing**: Função que realiza um mapeamento entre valores de chaves e entradas na tabela.

Uma tabela **hash** é construída através de um vetor de tamanho **n**, no qual se armazenam as informações. Nele, a localização de cada informação é dada a partir do cálculo de um índice através de uma função de indexação, a **função de hash.** A posição de um elemento é obtida aplicando-se ao elemento a função hash que devolve a sua posição na tabela. Em seguida, basta verificar se o elemento realmente está nesta posição.

Ex.: Construir uma tabela com os elementos 34, 45, 67, 78, 89, supondo que a tabela possui 10 elementos e a função hash é dada por **x%10** (resto da divisão por 10):

i	0	1	2	3	4	5	6	7	8	9
A[i]	0	0	0	0	34	45	0	67	78	79

A grande vantagem na utilização da tabela *hash* está no desempenho -- enquanto a busca linear tem complexidade temporal O(N) e a busca binária tem complexidade O(log N), o tempo de busca na tabela *hash* é praticamente independente do número de chaves armazenadas na tabela, ou seja, tem complexidade temporal O(1). Aplicando a função *hash* no momento de armazenar e no momento de buscar a chave, a busca pode se restringir diretamente àquela posição da tabela gerada pela função.

Idealmente, cada chave processada por uma função *hash* geraria uma posição diferente na tabela. No entanto, na prática existem **sinônimos** -- chaves distintas que resultam em um mesmo valor de *hashing*. Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma **colisão**.

Alguns dos problemas que se colocam quando usamos tabelas de hash são:

- Determinar uma função de hashing que minimize o número de colisões;
- Obter os mecanismos eficientes para tratar as colisões.

Uma boa função *hash* deve apresentar duas propriedades básicas: seu cálculo deve ser rápido e deve gerar poucas colisões. Além disso, é desejável que ela leve a uma ocupação uniforme da tabela para conjuntos de chaves quaisquer. Duas funções *hash* usuais são descritas a seguir:

Divisão

Nesse tipo de função, a chave é interpretada como um valor numérico que é dividido por um valor **M**. O resto dessa divisão inteira, um valor entre **0** e **M-1**, é considerado o endereço em uma tabela de **M** posições. Para reduzir colisões, é recomendável que **M** seja um número primo.

Enlaçamento

Neste método a chave é dividida em diversas partes que são combinadas ou "enlaçadas" e transformadas para criar o endereço. Existem 2 tipos de enlaçamento:

1. **Enlaçamento deslocado -** As partes da chave são colocadas uma embaixo da outra e processadas. Por exemplo, um código 123-456-789 pode ser dividido em 3 partes: 123-456-789 que são adicionadas resultando em 1368.
2. **Enlaçamento limite -** As partes da chave são colocadas em ordem inversa. Ex.: Considerando as mesmas divisões do código 123-456-789. Alinha-se as partes sempre invertendo as divisões da seguinte forma 321-654-987. O resultado da soma é 1566.

Meio do quadrado

Nesse tipo de função, a chave é interpretada como um valor numérico que é elevado ao quadrado. Os **r** bits no meio do valor resultante são utilizados como o endereço em uma tabela com $\mathbf{2^r}$ posições.

Transformação da raiz

A chave é transformada para outra base numérica. O valor obtido é aplicado no método da divisão valor%S para obter o endereço.

Como nem sempre a chave é um valor inteiro que possa ser diretamente utilizado, a técnica de *folding* pode ser utilizada para reduzir uma chave a um valor inteiro. Nessa técnica, a chave é dividida em segmentos de igual tamanho (exceto pelo último deles, eventualmente) e cada segmento é considerado um valor inteiro. A soma de todos os valores assim obtidos será a entrada para a função *hash*.

O processamento de tabelas *hash* demanda a existência de algum mecanismo para o tratamento de colisões. As formas mais usuais de tratamento de colisão são por endereçamento aberto ou por encadeamento.

Na técnica de tratamento de colisão por endereçamento aberto, a estratégia é utilizar o próprio espaço da tabela que ainda não foi ocupado para armazenar a chave que gerou a colisão. Quando a função *hash* gera para uma chave uma posição que já está ocupada, o procedimento de armazenamento verifica se a posição seguinte também está ocupada; se estiver ocupada, verifica a posição seguinte e assim por diante, até encontrar uma posição livre. (Nesse tipo de tratamento, considera-se a tabela como uma estrutura circular, onde a primeira posição sucede a última posição.) A entrada é então armazenada nessa posição. Se a busca termina na posição inicialmente determinada pela função *hash*, então a capacidade da tabela está esgotada e uma mensagem de erro é gerada.

No momento da busca, essa varredura da tabela pode ser novamente necessária. Se a chave buscada não está na posição indicada pela função *hashing* e aquela posição está ocupada, a chave pode eventualmente estar em outra posição na tabela. Assim, é necessário verificar se a chave não está na posição seguinte. Se, por sua vez, essa posição

estiver ocupada com outra chave, a busca continua na posição seguinte e assim por diante, até que se encontre a chave buscada ou uma posição sem nenhum símbolo armazenado.

Na técnica de tratamento de colisão por encadeamento, para cada posição onde ocorre colisão cria-se uma área de armazenamento auxiliar, externa à área inicial da tabela *hash*. Normalmente essa área é organizada como uma **lista ligada** que contém todas as chaves que foram mapeadas para a mesma posição da tabela. No momento da busca, se a posição correspondente à chave na tabela estiver ocupada por outra chave, é preciso percorrer apenas a lista ligada correspondente àquela posição até encontrar a chave ou alcançar o final da lista.

Hashing é uma técnica simples e amplamente utilizada na programação de sistemas. Quando a tabela *hash* tem tamanho adequado ao número de chaves que irá armazenar e a função *hash* utilizada apresenta boa qualidade, a estratégia de manipulação por *hashing* é bastante eficiente.

Conceitos e algoritmos de Árvores: Árvores Binárias de Pesquisa, Árvores Balanceadas; Árvores B e B+

As listas não lineares são exemplo de estrutura de dados não linear onde seus elementos podem ter mais de um predecessor ou mais de um sucessor. Essa forma de organizar dados permite que outros tipos de relação entre os dados possam ser representados, como por exemplo, hierarquia e composição. Um dos exemplos mais significativos de estruturas não lineares é a árvore. Essas estruturas são adequadas para representação de hierarquias, Ex.: hierarquia de pastas, árvores genealógicas.

Uma árvore em computação é uma estrutura hierárquica formada por vértices (nós) e ligações. Cada nó tem zero ou mais sucessores, mas tem apenas um predecessor, exceto o primeiro nó, chamado de raiz. Chamamos os elementos relacionados com um vértice de filhos e vértices sem filhos são chamado de folhas.

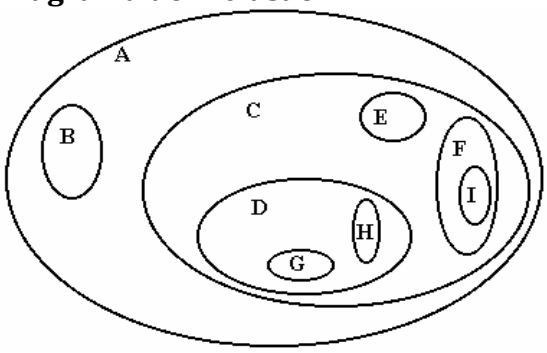
Formalmente, temos que uma árvore **T** é um conjunto finito de elementos denominados nós tais que:

- **T** = ∅, e a árvore é dita vazia, ou
- Existe um nó especial **r**, chamado raiz de **T,** com zero ou mais sub-árvores, cujas raízes estão ligadas a **r**; Os nós raízes destas sub-árvores são os filhos de **r**; os nós internos da árvore são os nós com filhos; as folhas ou nós externos da árvore são os nós sem filhos

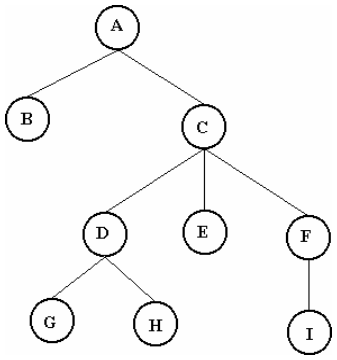
As três formas mais comuns de representação gráfica de uma árvore são:

1. **Representação por parênteses aninhados**
   (A (B) (C (D (G) (H)) (E) (F (I))))

2. **Diagrama de Inclusão**



3. **Representação Hierárquica**

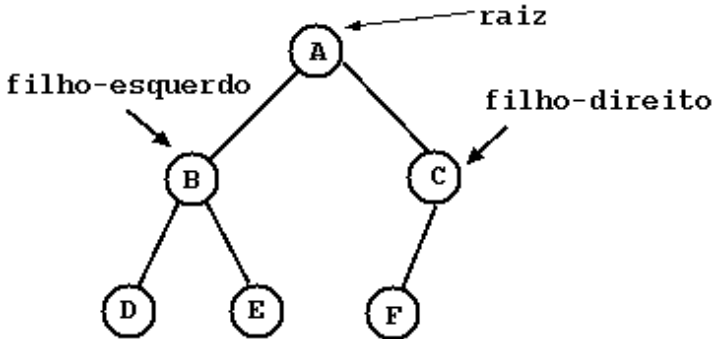


Chama-se **caminho** a uma sequência de ramos entre dois nós. Uma propriedade importante de uma árvore é que existe um e apenas um caminho entre dois quaisquer nós de uma árvore. O comprimento de um caminho é o número de ramos nele contido. A profundidade de um nó **n** é o comprimento do caminho de n até a raiz, já a profundidade da raiz é zero. A **altura de um nó** é o comprimento do caminho deste nó até o seu nó folha mais profundo (a altura de um nó folha é zero). Já a **altura de uma árvore** é o comprimento do maior caminho de um nó folha até a raiz.

Um dos tipos estrutura de dados em árvore mais fácil de se armazenar e manipular são as **árvores binárias.** Uma **árvore binária** é constituída por um conjunto finito de nós. Se o conjunto for vazio, a árvore diz-se vazia, caso contrário, obedece a seguinte regra:

1. Possui um nó especial, a raiz da árvore
2. Cada nó possui no máximo dois filhos, o **filho esquerdo** e o **filho-direito**
3. Cada nó, exceto a raiz, possui exatamente um **nó-pai**

De forma mais simples, **Árvores Binárias** são árvores em que cada nó tem 0, 1 ou 2 filhos



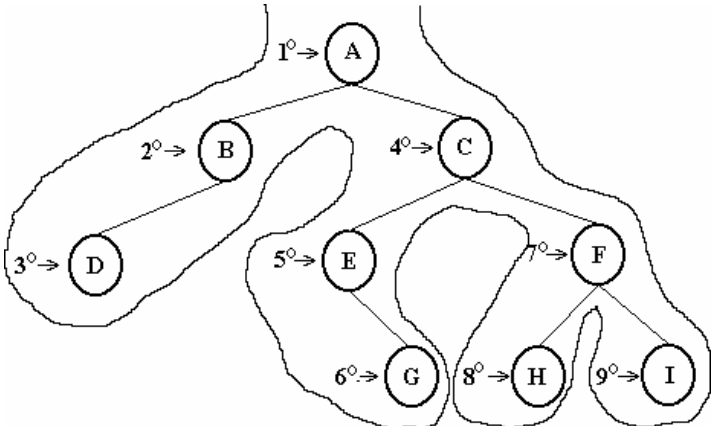
Uma derivação das árvores binárias são as **árvores binárias de busca.** Elas são estruturas usadas para representar conjuntos com as seguintes operações: busca, máximos, mínimos, inserção, remoção, sucessor e predecessor. A árvore binária de busca mantém uma ordem em seus elementos tal que:

- Todos os nós da sub-árvore esquerda são menores que o nó raiz
- Todos os nós da sub-árvore direita são maiores que o nó raiz
- As sub-árvores direita e esquerda são também Árvores Binárias de Busca

A vantagem da árvore binária de busca é que ela permite consultas rápidas, mesmo quando a quantidade de elementos é grande. A eficiência do uso de árvores binárias de pesquisa exige que estas sejam mantidas balanceadas.

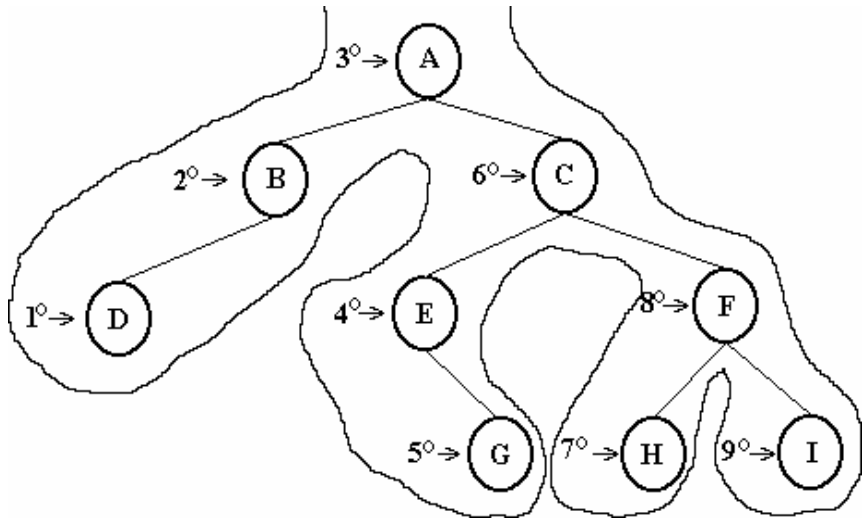
A busca ou percurso em uma árvore binária de busca é o processo de percorrer uma árvore visitando cada nó uma única vez. Esse processo gera uma sequencia linear de nós, onde é possível classificar o sucessor e o predecessor de um nó segundo um determinado percurso. A escolha por uma ou outra forma de percurso depende fundamentalmente da aplicação da árvore Há três maneiras de se percorrer árvores binárias:

1. **Travessia em Pré-ordem:**
   - Trata a raiz, percorre a Sub-árvore Esquerda, percorre a Sub-arvore Direita
   - Ex.:



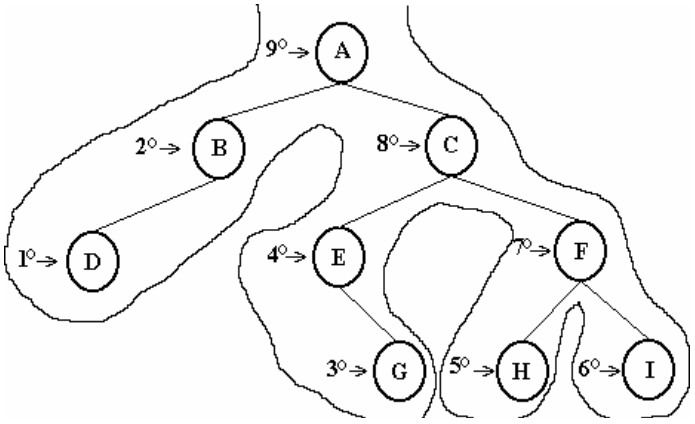
- **ABDCEGFHI**

2. **Travessia em Ordem Simétrica(In-ordem)**
   - Percorre a Sub-árvore Esquerda, Trata a raiz, percorre a Sub-arvore Direita
   - Ex.:



   - **DBAEGCHFI**

3. **Travessia em Pós-ordem**
   - Percorre a Sub-árvore Direita, percorre a Sub-arvore Esquerda, Trata a raiz
   - Ex.:



   - **DBGEHIFCA**

As árvores binárias de pesquisa são, em alguns casos, pouco recomendáveis para operações básicas (inserção, remoção e busca), principalmente quando elas estão degeneradas. Em uma árvore binária de busca, o custo da busca para o pior caso é igual à altura da árvore. Ou seja, sua complexidade é O(n), onde n é o número de nós da árvore.

Para que uma árvore seja, de fato, um mecanismo eficiente, é preciso que os seus elementos estejam distribuídos de forma relativamente homogênea pela estrutura (sub-árvores). Como as operações de inserção e remoção são aleatórias, é preciso ter um operador capaz de manter os elementos distribuídos de forma homogênea entre as sub-árvores. Esse é o operador de **balanceamento.**

O balanceamento de uma árvore pode ser feito segundo duas estratégias: **Global** – envolvendo toda a árvore, ex. Algoritmo DSW; e **Local** – envolvendo apenas uma parte da árvore, ex.: Árvore AVL ou RB.

No balanceamento global a árvore balanceada pode ser construída a partir de uma estrutura externa. O algoritmo DSW envolve dois passos para a criação de uma árvore perfeitamente balanceada. O primeiro passo transforma uma árvore desbalanceada em uma espinha dorsal. A espinha dorsal é simplesmente uma lista linear ordenada que contém os elementos da árvore binária de busca. A segunda etapa converte a espinha dorsal em uma árvore perfeitamente balanceada por meio da realização de uma série de rotações. O número de rotações é dado em função do número de nós e na altura da árvore final.

O balanceamento local faz uso de algoritmos que trabalham apenas em parte da árvore, a cada inserção ou remoção. Algoritmos desse tipo podem ser representados pelas árvores AVL ou árvores Rubro-negras. Uma árvore AVL é uma árvore binária de pesquisa onde a diferença em altura entre as sub-árvore esquerda e direita é no máximo 1 (positivo ou negativo). Quando a diferença chega a 2 ou –2, deve ser refeito o balanceamento através de rotações. Chamamos essa diferença de "fator de balanceamento".

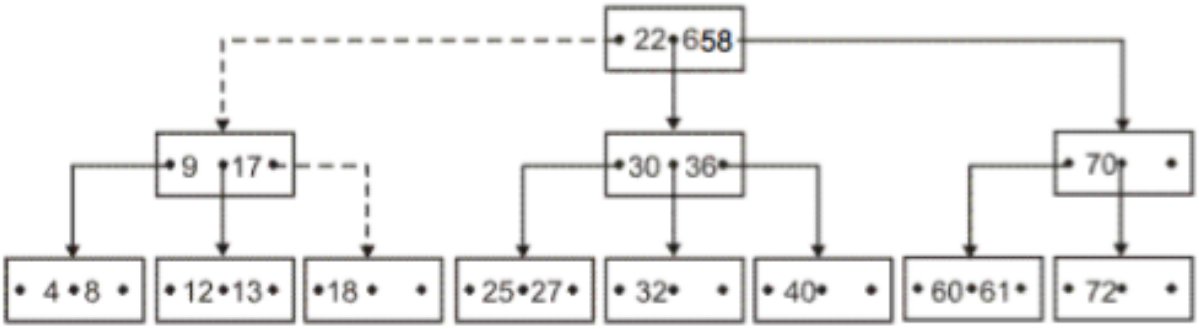
Na implementação das árvores binárias pode ser utilizado matrizes (ou vetor de vetores) ou ponteiros. Através de matrizes, usa-se um vetor para guardar o conteúdo efetivo do nó e outros dois vetores para armazenar os índices dos nós raízes de cada uma das sub-árvores desse nó. Se o nó não possuir sub-árvore, então o vetor correspondente recebe o valor-1. O nó raiz da árvore ocupa a primeira posição do vetor ou, opcionalmente, é apontado por uma variável externa que marca o índice da posição ocupada pela raiz.

Na implementação através de ponteiros, cada nó da árvore é representado por uma estrutura contendo tanto as informações do nó, quanto os apontadores para as suas sub-árvores. Se um nó não possui uma sub-árvore, o ponteiro respectivo é marcado como nulo. A raiz é identificada através de um ponteiro específico.

As árvores B são uma generalização das árvores binárias de busca. Ao invés de armazenar em cada nó uma única chave de busca como as árvores binárias, as árvores B armazenam mais de uma chave de busca em cada nó da estrutura, proporcionando uma organização de ponteiros tal que que as operações sejam executadas rapidamente.

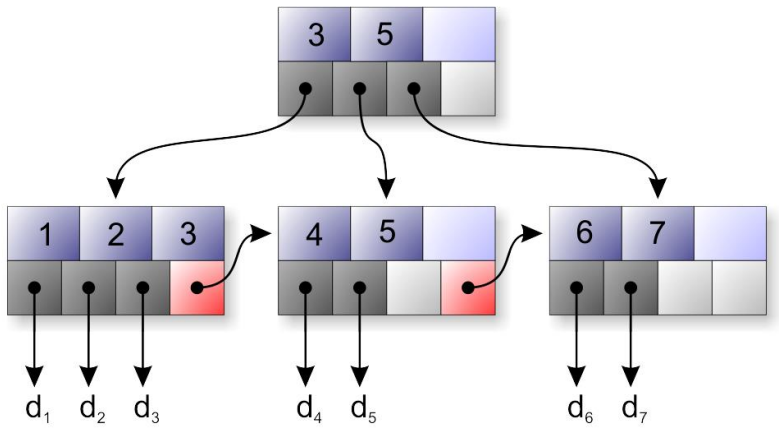
As **árvores B** são organizadas por nós, tais como os das árvores binárias de busca, mas estes apresentam um conjunto de chaves maior do que um e são usualmente chamados de páginas. As chaves em cada página são, no momento da inserção, ordenadas de forma

crescente e para cada chave há dois endereços para páginas filhas, sendo que, o endereço à esquerda é para uma página filha com um conjunto de chaves menor e o à direita para uma página filha com um conjunto de chaves maior. A figura abaixo demonstra essa organização de dados característica.



Nas árvores B, os registros são mantidos em um arquivo, e ponteiros (**esq.** e **dir.**) indicam onde estão os registros filhos. Esta estrutura pode ser mantida em memória secundária: os ponteiros para os filhos dariam o número dos registros correspondentes aos filhos, i.e., a sua localização no arquivo.

As **árvores B** não são as únicas estruturas de dados usadas em aplicações que demandam a manipulação de grande volume de dados, também existem variações desta que proporcionam determinadas características como as árvores B+. **As árvores B+** possuem seus dados armazenados somente em seus nós folha e, seus nós internos e raiz, são apenas referências para as chaves que estão em nós folha. Assim é possivel manter ponteiros em seus nós folha para um acesso sequencial ordenado das chaves contidas no arquivo. A principal característica proporcionada por esta variação é o fato de permitir o acesso sequencial das chaves por meio de seu *sequence set* de maneira mais eficiente do que o realizado em **árvores B**. Além do mais, as páginas utilizadas em seu *index set* podem conter mais apontadores para páginas filha permitindo reduzir a altura da árvore. Elas são bastante utilizadas para criar indices de banco de dados (Usado pelo NTFS do Windows)



Grafos: Representação (Matriz de Adjacencia, Lista de Adjacencia) Caminho (Largura, Profundidade, Menor Caminho). Complexidade

Um **grafo** é uma forma de especificar relações entre coleção de objetos. A **teoria dos grafos** é muito utilizada como ferramenta de modelagem onde a abstração permite codificar pares de objetos (pessoas, cidades, empresas) e seus relacionamentos (amizade, distancia, produção).

Por definição, Um grafo é um conjunto de pontos, chamados vértices, conectados por linhas, chamadas de arestas. Dessa forma, temos o grafo G(V,A) onde:

- V é um conjunto não vazio de objetos denominados vértices;
- A é um subconjunto de pares não ordenados de V, chamados arestas que definem o relacionamento entre os vértices

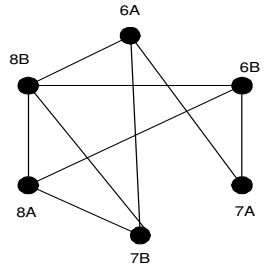
Um exemplo seria jogos de futebol interclasse onde:

- Objetos: turmas 6A, 6B, 7A, 7B, 8A e 8B
- Relacionamento: partida de futebol

Existem basicamente 2 formas de se representar a estrutura desse grafo:

- Por uma lista, dizendo quem se relaciona com quem.
- Por um desenho, isto é, uma representação gráfica.

$6A$	jogou com	$7A$, $7B$, $8B$
$6B$	jogou com	$7A$, $8A$, $8B$
$7A$	jogou com	$6A$, $6B$
$7B$	jogou com	$6A$, $8A$, $8B$
$8A$	jogou com	$6B$, $7B$, $8B$
$8B$	jogou com	$6A$, $6B$, $7B$, $8A$



Quando existe uma aresta ligando dois vértices dizemos que os vértices são adjacentes e que a aresta é incidente aos vértices. No nosso exemplo podemos representar o grafo de forma sucinta como:

- V ={6A;6B;7A;7B;8A;8B}
- A = {(6A; 7A); (6A; 7B); (6A; 8B); (6B; 7A); (6B; 8A); (6B; 8B); (7B; 8A); (7B; 8B); (8A; 8B)}

Observe que não precisamos colocar (8A;7B) no conjunto de arestas pois já tínhamos colocado (7B; 8A). O número de vértices será simbolizado por |V | ou pela letra n. O número de arestas será simbolizado por |A| ou pela letra m. No nosso exemplo n = 6 e m = 9.

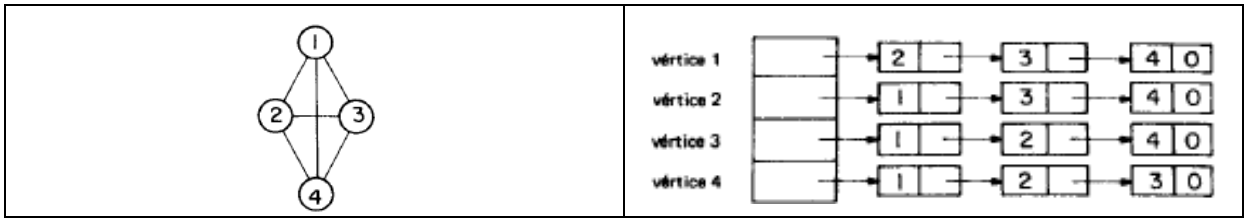
O número de vezes que as arestas incidem sobre o vértice v é chamado grau do vértice v, simbolizado por d(v). No nosso exemplo, d(6A) = 3; d(7A) = 2.

Há diversas maneiras de armazenarmos grafos em computadores. A estrutura de dados usada dependerá tanto da estrutura do grafo quanto do algoritmo usado para manipulá-lo. Teoricamente, podemos dividir entre estruturas do tipo lista e do tipo matriz, mas em aplicações reais, a melhor estrutura é uma combinação de ambas. Estruturas do tipo lista são frequentemente usadas em grafos esparsos (grafos que possuem um pequeno

número de arestas em relação ao número de vértices, oposto ao grafo denso, onde o número de arestas se aproxima do máximo de arestas possível) já que exigem menor uso da memória. Por outro lado, estruturas do tipo matriz fornecem um rápido acesso em algumas aplicações, mas podem consumir uma grande quantidade de memória.

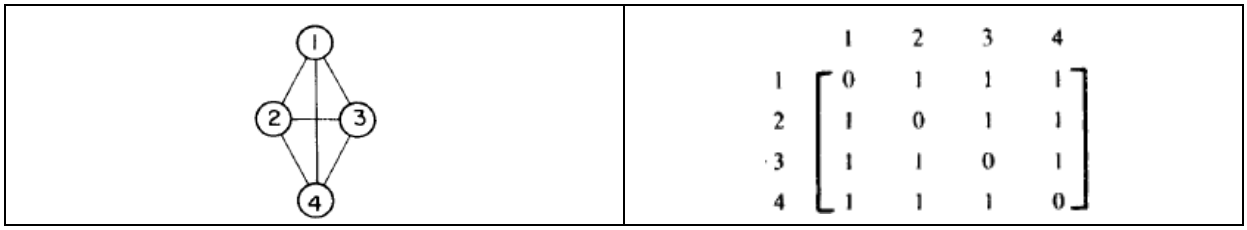
Estruturas do tipo lista incluem a lista de adjacência que associa a cada vértice do grafo uma lista de todos os outros vértices com os quais ele tem uma aresta. Nesta estrutura, seja G(V,A) um grafo com *n* vértices, onde *n >= 1*. Supomos que os vértices são numerados 1, 2, ..., |V| de alguma maneira arbitrária. Existe uma lista para cada vértice em *G*. Os nós na relação *i* representam os vértices que são adjacentes do vértice *i*. Cada nó possui, pelo menos, dois campos: VERTEX e LINK. Os campos VERTEX contém dois índices dos vértices adjacentes do vértice *i*. Cada relação possui um nó de cabeça. Os nós de cabeça são sequenciais, permitindo fácil acesso aleatório à lista de adjacências de determinado vértice.

Ex.:



Estruturas do tipo matriz incluem a matriz de adjacência, uma matriz de 0's e 1's onde ambas linhas e colunas possuem vértices. Dado um grafo *G* com *n* vértices, podemos representá-lo em uma matriz *n* x *n A(G)*=[a_{ij}] (ou simplesmente *A*). Para representar um grafo não direcionado, simples e sem pesos nas arestas, basta que as entradas a_{ij} da matriz *A* contenham 1 se v_i e v_j são adjacentes e 0 caso contrário. Se as arestas do grafo tiverem pesos, a_{ij} pode conter, ao invés de 1 quando houver uma aresta entre v_i e v_j, o peso dessa mesma aresta.

Ex.:



Em relação à sua complexidade, a matriz de adjacências no pior caso gasta-se tempo O(1) para decidir se dois vértices são vizinhos. Já a lista de adjacência Gasta-se tempo O(n) no pior caso para decidir se dois vértices são vizinhos.

Estruturas que podem ser representadas por grafos estão em toda parte e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos. Um dos problemas mais estudados é o caminhamento em grafo. Um percurso ou caminhamento é uma sequência de arestas sucessivamente adjacentes, cada uma tendo uma extremidade adjacente à anterior e a outra a subsequente (à exceção da primeira e da última). O comprimento do caminho é dado pelo número de arestas (o que faz sentido: é o número de "passos" que gastamos para percorrer o caminho). Explorando um grafo de forma sistêmica, muitas aplicações e algoritmos são abstraídos. Problemas como identificar o número de arestas que um grafo tem (comprimento do grafo) ou o caminho mais curto entre dois vértices (distância entre dois vértices) são explorados

pela busca em grafo. Os três tipos de caminhamento em grafo mais estudados são: **Busca em largura**, **Busca em profundidade** e o **Menor Caminho**.

Busca em Largura (busca por extensão ou Breadth-First Search - BFS)

A busca por largura é um algoritmo de busca em grafos utilizado para realizar uma busca ou travessia num grafo e estrutura de dados do tipo árvore. A propriedade especial está no fato de a árvore não possuir ciclos: dados dois vértices quaisquer, existe exatamente 1 caminho entre eles. Um percurso em extensão é visitar cada nó começando do menor nível e move-se para os níveis mais altos nível após nível, visitando cada nó da esquerda para a direita. Sua implementação é direta quando uma fila é utilizada. Depois que um nó é visitado, seus filhos, se houver algum, são colocados no final da fila e o nó no início da fila é visitado. Assim, os nós do nível n+1 serão visitados somente depois de ter visitados todos os nós do nível n. Computa a menor distância para todos os vértices alcançáveis. O sub-grafo contendo os caminhos percorridos é chamado de breadth-first tree.

Considerando um grafo representado em listas de adjacência, o pior caso, aquele em que todos os vértices e arestas são explorados pelo algoritmo, a complexidade de tempo pode ser representada pela seguinte expressão O (|A| + |V|), onde |A| significa o tempo total gasto nas operações sobre todas as arestas do grafo onde cada operação requer um tempo constante O(1) sobre uma aresta, e |V| que significa o número de operações sobre todos os vértices que possui uma complexidade constante O (1) para cada vértice uma vez que todo vértice é enfileirado e desenfileirado uma única vez.

Busca em Profundidade (Depth-First Search - DFS)

A busca em profundidade também é um algoritmo para realizar busca em um grafo tipo árvore. Um algoritmo de busca em profundidade realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda cada vez mais, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (backtrack) e começa no próximo nó. Numa implementação não-recursiva, todos os nós expandidos recentemente são adicionados a uma pilha, para realizar a exploração. A complexidade espacial de um algoritmo de busca em profundidade é muito menor que a de um algoritmo de busca em largura porem a complexidade temporal, ela é equivalente (proporcionais ao número de vértices somados ao número de arestas dos grafos aos quais eles atravessam).

Menor Caminho (Algoritmo de Dijkstra)

O algoritmo de Dijkstra assemelha-se ao BFS, mas é um algoritmo guloso, ou seja, toma a decisão que parece ótima no momento. O algoritmo considera um conjunto S de menores caminhos, iniciado com um vértice inicial I. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a I e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por I estejam em S. Arestas que ligam vértices já pertencentes a S são desconsideradas. Em tempo computacional, a complexidade para o pior caso é dada por O([m+n]log n) onde m é o número de arestas e n é o número de vértices.

Spanning trees (árvore de extensão)

Dado um grafo não orientado conectado, uma árvore de extensão deste grafo é um sub-grafo o qual é uma árvore que conecta todos os vértices. Um único grafo pode ter diferentes árvores de extensão. Nós podemos assinalar um *peso* a cada aresta, que é um número que representa quão desfavorável ela é, e atribuir um peso a árvore de extensão calculado pela soma dos pesos das arestas que a compõem. Uma **árvore de extensão mínima** (também conhecida como **árvore de extensão de peso mínimo** ou **árvore geradora mínima**) é então uma árvore de extensão com peso menor ou igual a cada uma das outras árvores de extensão possíveis. Generalizando mais, qualquer grafo não direcional (não necessariamente conectado) tem uma **floresta de árvores mínimas**, que é uma união de árvores de extensão mínimas de cada uma de suas componentes conexas.

Um exemplo de uso de uma árvore de extensão mínima seria a instalação de fibras óticas num campus de uma faculdade. Cada trecho de fibra ótica entre os prédios possui um custo associado (isto é, o custo da fibra, somado ao custo da instalação da fibra, mão de obra, etc.). Com esses dados em mãos (os prédios e os custos de cada trecho de fibra ótica entre todos os prédios), podemos construir uma *árvore de extensão* que nos diria um jeito de conectarmos todos os prédios sem redundância. Uma *árvore geradora mínima* desse grafo nos daria uma árvore com o menor custo para fazer essa ligação.

Uma árvore de extensão/dispersão apresenta as seguintes propriedades:

- Define um subconjunto de arestas que mantém o grafo conectado em um único componente;
- Em um grafo não-valorado qualquer árvore de dispersão é mínima;
- Podem ser calculadas em tempo polinomial;

Os algoritmos usuais para a determinação de árvores de extensão/dispersão são o algoritmo de Prim (1957) e o algoritmo de Kruskal (1956).

Algoritmos de ordenação interna. Complexidade.

Um algoritmo de ordenação é um algoritmo que recebe uma lista de elementos e gera como saída uma nova lista de elementos ordenados (classificados, organizados) em uma determinada ordem. O processo de organizar os dados em uma determinada ordem é parte importante de muitos métodos e técnicas em computação. Uma série de algoritmos de busca, intercalação/fusão, utilizam ordenação como parte do processo, por exemplo, aplicações em geometria computacional, bancos de dados, entre outras necessitam de listas ordenadas para funcionar.

A saída de um algoritmo de ordenação deve satisfazer duas condições:

1. estar em uma ordem crescente ou decrescente,
2. ser uma permutação da entrada.

Ex.:

- **Entrada:** 6 5 7 1 4 3 2
- **Saídas possíveis:** 1 2 3 4 5 6 7 **ou** 7 6 5 4 3 2 1

Todo algoritmo de ordenação deve realizar as seguintes tarefas: receber os dados para poderem ser ordenados, comparar os elementos dos dados que estabelecerão a ordem e verificar a necessidade de realizar a troca para se chegar à ordem. É possível classificar os algoritmos em ordenação interna e externa. Os algoritmos de ordenação interna consomem apenas a memória primária para realizar a ordenação. O método externo usa memória externa, secundária, já que o arquivo é muito grande para caber na memória principal.

Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação. Sendo n o número registros no arquivo, as medidas de complexidade relevantes são:

- Número de comparações C (n) entre chaves.
- Número de movimentações M (n) de item do arquivo.

O uso econômico da memória disponível é um requisito primordial na ordenação interna. Métodos de ordenação in situ são os preferidos, por utilizarem a estrutura de vetores.

Podemos classificar os métodos de ordenação em subclasses. Existem os algoritmos elementares chamados assim devido a sua fácil implementação, geralmente eles possuem uma complexidade de tempo maior, e uma complexidade de espaço menor. Outra subclasse não os algoritmos eficientes possuem códigos maiores e mais complexos, geralmente consomem menos tempo e mais espaço que os métodos elementares.

Os Algoritmos elementares são adequados para pequenos arquivos, eles requerem $O(n^2)$ comparações e produzem programas pequenos. Exemplo de algoritmos elementares são**: Selection sort, Insertion sort, Bubble sort**.

Os algoritmos eficientes são adequados para arquivos maiores. Eles requerem O(n log n) comparações, ou seja, usam menos comparações, porém mais complexas nos detalhes. Exemplo de algoritmos elementares são: **Merge sort, Heapsort, Quick sort**.

Ordenação por seleção **(Selection Sort)**

é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos. Sua complexidade para o pior caso é $O(n^2)$.

- **Vantagens:** Custo linear no tamanho da entrada para o número de movimentos de registros – a ser utilizado quando há registros muito grandes
- **Desvantagens**: Não adaptável, não importa se o arquivo está parcialmente ordenado ele irá percorrer todo o arquivo, além disso, o Algoritmo não é estável.

Ordenação por inserção **(Insertion Sort)**

Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer. Sua complexidade para o pior caso é $O(n^2)$.

- **Vantagens**: Laço interno é eficiente, inserção é adequado para ordenar vetores pequenos. Esse método é vantajoso para ser utilizado quando o arquivo está "quase" ordenado e deseja-se adicionar poucos itens a um arquivo ordenado, pois o custo é linear e o algoritmo é **estável .**
- **Desvantagens**: Número de comparações tem crescimento quadrático, além do alto custo de movimentação de elementos no vetor.

Ordenação pelo método da bolha **(bubble sort)**

O ***bubble sort***, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo. Sua complexidade para o pior caso é $O(n^2)$.

- **Vantagens**: Algoritmo simples e estável
- **Desvantagens:** Não adaptável e realiza muitas trocas de itens

Ordenação por mistura **(merge sort)**

O ***merge sort***, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Sua ideia básica consiste em Dividir (o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade) e Conquistar (após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas). Sua complexidade para o pior caso é $O(n \log_2 n)$.

Vantagens: O algoritmo é estável, mais fácil de ser paralelizado e possui uma implementação não recursiva simples

Desvantagens: Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas .

Ordenação **Heapsort**

O algoritmo **heapsort** é um algoritmo de ordenação generalista que utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.

A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor. Para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máxima (o maior elemento fica na raiz). Em termo de comparações, sua complexidade para o pior caso é $2n \log_2 n + O(n)$, que é o mesmo que $2n \lg n + O(n)$. Já para as trocas, no piór caso a sua complexidade é $n \log_2 n + O(n)$, que é o mesmo que $n \lg n + O(n)$.

- **Vantagens**: O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada
- **Desvantagens**: O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort, além disso o algoritmo não é estável.

Ordenação por **Quicksort**

O Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o Quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Os passos são:

. Escolha um elemento da lista, denominado *pivô*;

. Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada *partição*;

. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte. Sua complexidade para o pior caso é $O(n^2)$.

Vantagens: É extremamente eficiente para ordenar arquivos de dados, necessitando de apenas uma pequena pilha como memória auxiliar. Requer cerca de n log n comparações em média para ordenar n itens.

Desvantagens: Tem um pior caso O(n2) comparações e sua implementação é muito delicada e difícil: um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados. Além disso, o método não é estável.

Há limites fundamentais sobre o desempenho dos algoritmos de comparação. Um algoritmo de comparação deve ter um limite inferior de $\Omega(n \log n)$ operações de comparação. Esta é uma consequência da escassa informação disponível, através de comparações sozinhas - ou, dito de outra forma, da vaga estrutura algébrica dos conjuntos totalmente ordenados. Entretanto, existem algoritmos de ordenação em

tempo linear desde que a entrada de dados possua características especiais, algumas restrições sejam respeitadas, o algoritmo não seja puramente baseado em comparações e sua implementação seja feita de maneira adequada. Dentre esses algoritmos podemos citar os de Ordenação por Contagem **(counting sort), Radix sort e Bucket sort**. Esses algoritmos exigem um conhecimento prévio sobre os dados a serem ordenados, portanto não são tão genéricos como os algoritmos de comparação.

Ordenação por contagem

O algoritmo por contagem (counting sort) Pressupõe que cada elemento da entrada é um inteiro na faixa de 0 a k, para algum inteiro k. A ideia básica do counting sort é determinar, para cada entrada x, o número de elementos menor que x. Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no array de saída. Por exemplo, se há 17 elementos menor que x, então x pertence a posição 18. Esse esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor, uma vez que não se deve colocar eles na mesma posição.

O Counting Sort ordena exclusivamente números inteiros pelo fato de seus valores servirem como índices no vetor de contagem. Uma desvantagem é que esse algoritmo utiliza vetores auxiliares em sua implementação.

Radix sort

O ***Radix sort*** surgiu no contexto de ordenação de cartões perfurados; a máquina ordenadora so era capaz de ordenar os cartões segundo um de seus dígitos. Ele é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o *radix sort* ordena estas chaves numa qualquer ordem relacionada com a lexicografia. O algoritmo Radix Sort ordena um vetor A de n números inteiros com um número constante d de dígitos, através de ordenações parciais dígito a dígito
. Existem duas classificações do radix sort, que são:

- Least significant digit (LSD – Dígito menos significativo) radix sort; - Most significant digit (MSD – Dígito mais significativo) radix sort.

O radix sort LSD começa do dígito menos significativo até o mais significativo, ordenando tipicamente da seguinte forma: chaves curtas vem antes de chaves longas, e chaves de mesmo tamanho são ordenadas lexicograficamente. Isso coincide com a ordem normal de representação dos inteiros, como a seqüência "1, 2, 3, 4, 5, 6, 7, 8, 9, 10". Os valores processados pelo algoritmo de ordenação são frequentemente chamados de “chaves”, que podem existir por si próprias ou associadas a outros dados. As chaves podem ser strings de caracteres ou números.

Já o radix sort MSD trabalha no sentido contrário, usando sempre a ordem lexicográfica, que é adequada para ordenação de strings, como palavras, ou representações de inteiros com tamanho fixo. A seqüência "b, c, d, e, f, g, h, i, j, ba" será ordenada lexicograficamente como "b, ba, c, d, e, f, g, h, i, j". Se essa ordenação for usada para ordenar representações de inteiros com tamanho variável, então a representação dos números inteiros de 1 a 10 terá a saída "1, 10, 2, 3, 4, 5, 6, 7, 8, 9".

Bucket sort

Bucket sort, ou **bin sort**, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipients (baldes). Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o

algoritmo bucket sort recursivamente. O bucket sort tem complexidade linear ($\Theta(n)$) quando o vetor a ser ordenado contém valores que são uniformemente distribuídos. O seu algoritmo divide o intervalo [0, 1) em n sub-intervalos iguais, denominados buckets (baldes), e então distribui os n números reais nos n buckets. Como a entrada é composta por dados distribuídos uniformemente, espera-se que cada balde possua, ao final deste processo, um número equivalente de elementos (usualmente 1). Para obter o resultado, basta ordenar os elementos em cada bucket e então apresentá-los em ordem.

Algoritmos de pesquisa: pesquisa sequencial, pesquisa binária

Quando temos um Vetor (ou Matriz) com muitos elementos e precisamos descobrir se um determinado elemento que procuramos se encontra no vetor, uma solução que certamente nos vem à mente é comparar o elemento que procuramos com cada elemento do vetor, até que encontremos ou até que concluamos que o elemento procurado não está no vetor.

Esta é a base do raciocínio dos algoritmos de pesquisa ou busca ("Search"), que como sugere o nome, "Pesquisam", em um vetor, a existência ou não existência de um elemento procurado. A diferença entre um e outro algoritmo de busca, fica por conta da rapidez com que "varremos" o vetor para encontrar o elemento ou para concluirmos que ele não existe.

Um fator que influencia em muito nessa rapidez é a disposição dos elementos no vetor. Se estão desordenados, seguramente teremos que verificar do primeiro ao último elemento para concluir, com certeza, que o elemento não existe. Já se estão ordenados, ao encontrarmos um elemento maior (ou menor) que o elemento procurado, poderemos concluir pela sua não existência. Os algoritmos de ordenação ou classificação ("Sort"), por sua vez, são utilizados para ordenar os elementos de um vetor de forma a facilitar a pesquisa posterior de um elemento, no conjunto de elementos existentes.

Para fazermos qualquer pesquisa em vetor (ou matriz) precisamos de quatro parâmetros:

a) vetor no qual realizaremos a pesquisa
b) número de elementos desse vetor que devem ser pesquisados
c) elemento procurado
d) Um índice que vai ser preenchido com a posição onde o elemento foi encontrado ou retornará com 0 (zero) caso o elemento não exista.

O tempo de pesquisa depende do algoritmo de pesquisa utilizado e a escolha desse algoritmo depende diretamente da quantidade de dados envolvidos, da frequência das operações de inserção e de exclusão dos registros e da sua ordenação. Quando a operação de pesquisa é muito mais frequente do que a inserção, deve-se minimizer o tempo de pesquisa através da ordenação dos registros. Existem vários algoritmos de busca dos quais podemos destacar os algoritmos de busca sequencial e de busca binária.

O algoritmo de pesquisa sequencial Ou busca linear é o método mais simples e claro para uma variedade de estrutura de dados. Ele verifica elemento por elemento da lista, de modo que a função do tempo de execução do algoritmo em relação ao número de elementos é linear, ou seja, cresce proporcionalmente. O seu algoritmo inicia a pesquisa pelo primeiro registro, avança sequencialmente (registro por registro) até encontrar o registro (busca com sucesso) ou quando todos os registros forem pesquisados e a chave não for encontrada (busca sem sucesso). Seu algoritmo é descrito da seguinte forma:

1. i <-- 0
2. Se i = n, escreve "não existe" e termina.
3. Se x = a[i], escreve "existe" e termina.
4. i <-- i+1. Volta ao passo 2.

Analisando esse algoritmo, temos que para uma pesquisa com sucesso, o algoritmo no melhor caso vai realizar 1 iteração (chave no primeiro registro); no pior caso irá realizar N iterações (chave no ultimo registro); e (N+1)/2 iterações para o caso médio (registro no meio). Em uma pesquisa sem sucesso, o algoritmo irá realizar N+1 iterações. De forma geral o algoritmo é O(n) em complexidade e é considerado a melhor solução para o problema da pesquisa em tabelas desordenadas com poucos registros.

A pesquisa binária é um método que só se aplica a vetores previamente ordenados. A sua grande vantagem é a rapidez e, por isso, ela é muito recomendada para vetores grandes. Para saber se uma chave está presente na tabela, compare a chave com o registro que está no meio da tabela. Se a chave é menor, então o registro procurado está na primeira metade da tabela. Se a chave é maior, então o registro procurado está na segunda metade da tabela. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso. No seu algoritmo, é usado duas variáveis: esquerda e direita, que significam os limites esquerdo e direito do vetor. A idéia é que se o elemento x existe numa posição a[i] do vetor, então i tem de estar compreendido entre esq e dta (esq ≤ i ≤ dta). Seu aglgoritmo é descrito da seguinte forma:

1. esq <-- 0 , dta <-- n-1
2. Se esq > dta, escreve "não existe" e termina.
3. i <-- parte inteira de (esq+dta)/2
4. Se x = a[i], escreve "existe" e termina.
   Se x < a[i], dta <-- i-1. Volta ao passo 2.
   Se x > a[i], esq <-- i+1. Volta ao passo 2.

Analisando esse algoritmo, temos que a cada iteração o número de elementos a

serem pesquisados é reduzido à metade (N, N/2, N/4, N/8, ..., $N/2^k$). Queremos que $N/2^k \leq 1$, logo $k \geq \log_2 N$. A chave pesquisada deve ser comparada com o último elemento restante, portanto o número máximo de comparações é $1 + \log_2 N$. Conclusão: Sua ordem de complexidade é $O(\log_2 N)$. O log2n cresce muito devagar com o aumento de n.

Entretanto, não se pode concluir que o algoritmo de busca binária é melhor que o de pesquisa sequencial. Há vários pontos que se deve ter em consideração:

1. Se n é pequeno (inferior a 100) não vale a pena fazer pesquisas binárias porque o computador é muito rápido para varrer uma coleção de 100 elementos por exemplo.
2. O algoritmo de pesquisa binária assume que o vetor está ordenado. Ordenar um vetor também tem um custo
3. Se for para fazer uma só pesquisa, não vale a pena ordenar o vetor. Por outro lado, se pretendermos fazer muitos pesquisas, o esforço da ordenação já poderá valer a pena.