Etapa 1 - Introdução a Estrutura de dados.

1. Introdução às Estruturas de Dados

As estruturas de dados são fundamentais na Ciência da Computação, representando a forma como os dados são organizados, armazenados e gerenciados na memória de um computador. A escolha da estrutura correta não é arbitrária; ela impacta diretamente a eficiência e performance dos algoritmos que as utilizam.

- Conceito Central: É a relação intrínseca entre os dados e as operações que podem ser realizadas sobre eles (como busca, inserção, remoção e travessia). Uma estrutura bem escolhida permite que essas operações sejam executadas de forma rápida e com consumo otimizado de recursos (memória).
- Importância: Elas são os alicerces sobre os quais algoritmos complexos são construídos. Uma estrutura inadequada pode tornar um problema intratável, enquanto a correta pode resolvê-lo de forma eficiente.

---

2. Estruturas de Dados Lineares

Nestas estruturas, os elementos são dispostos em uma sequência linear, um após o outro.

- Arrays (Vetores e Matrizes):
  - São coleções de elementos armazenados em posições contíguas de memória.
  - Permitem acesso direto a qualquer elemento através de um índice em tempo constante, O(1).
  - A principal desvantagem é a rigidez de tamanho (em arrays estáticos) e o custo para inserir/remover elementos no meio, que pode exigir o deslocamento de todos os outros.

- Listas Encadeadas:
  - Compostas por nodos independentes, onde cada um armazena o dado e um ponteiro para o próximo nodo (no caso das listas simples).
  - Inserções e remoções são muito eficientes (O(1)) se houver uma referência direta ao local, pois não exigem deslocamento de elementos.
  - Tipos:
    - Simples: Navegação apenas em uma direção.
    - Duplamente Encadeada: Cada nodo aponta para o anterior e o próximo, permitindo navegação bidirecional.
    - Circular: O último nodo aponta de volta para o primeiro.

- Pilhas (Stack):
  - Seguem o princípio LIFO ("Last-In, First-Out"): o último elemento a entrar é o primeiro a sair.
  - Operações Básicas:
    - `Push`: Adiciona um elemento ao topo.
    - `Pop`: Remove e retorna o elemento do topo.
    - `Peek`: Espia o elemento do topo sem removê-lo.

- Aplicações: Mecanismo de "undo" em editores, chamadas de funções (a "call stack") e avaliação de expressões.

- Filas (Queue):
  - Seguem o princípio FIFO ("First-In, First-Out"): o primeiro elemento a entrar é o primeiro a sair.
  - Tipos Comuns:
    - Fila Simples: Operações básicas de enfileirar (`enqueue`) e desenfileirar (`dequeue`).
    - Fila Circular: Reutiliza o espaço na memória quando elementos são removidos.
    - Fila de Prioridade: Elementos são processados por prioridade, não pela ordem de chegada (geralmente implementada com um *Heap*).
  - Aplicações: Fila de impressão, buffers de dados e processamento de tarefas em ordem.

---

3. Estruturas de Dados Não-Lineares

Os elementos são organizados de maneira hierárquica ou com conexões arbitrárias, não seguindo uma sequência linear.

- Árvores:
  - Estrutura hierárquica composta por nós, com um nó raiz no topo.
  - Árvore Binária: Cada nó possui no máximo dois filhos (esquerdo e direito).
  - Árvore Binária de Busca (BST): Uma árvore binária com uma propriedade de ordenação: para qualquer nó, todos os valores na subárvore esquerda são menores, e os da direita são maiores. Isso permite buscas eficientes (O(log n) em casos balanceados).
  - Árvores Balanceadas (AVL, Red-Black): São BSTs que se auto-ajustam para manter uma altura aproximadamente logarítmica, garantindo eficiência mesmo após sucessivas inserções e remoções.
  - Heap: Uma árvore binária completa que satisfaz a propriedade de heap (o valor do nó pai é maior/menor que os dos filhos). Usada para implementar filas de prioridade.

- Grafos:
  - A estrutura mais geral, composta por vértices (ou nós) conectados por arestas (ou arcos).
  - Tipos:
    - Direcionados: As arestas têm uma direção (ex: A -> B).
    - Não-Direcionados: As arestas são conexões bidirecionais.
    - Ponderados: As arestas possuem um "peso" ou custo associado.
  - Representação:
    - Matriz de Adjacência: Uma matriz 2D onde `matriz[i][j]` indica uma conexão.
    - Lista de Adjacência: Um array de listas, onde cada lista contém os vizinhos de um vértice.

---

4. Estruturas de Dados por Hashing

Focam em fornecer acesso extremamente rápido aos dados através de um mapeamento direto.

- **Tabelas Hash:**
  - Armazenam pares **chave-valor**. Uma **função hash** transforma a chave em um índice de um array, onde o valor é armazenado.
  - Oferecem complexidade de tempo **O(1)** em média para operações de inserção, busca e remoção.
  - O grande desafio são as **colisões** (quando duas chaves diferentes geram o mesmo índice). Isso é resolvido com técnicas como **Encadeamento** (cada posição do array é uma lista) ou **Endereçamento Aberto** (procura-se outra posição vazia no array).

---

5. Estruturas Avançadas

Projetadas para resolver problemas específicos de forma altamente otimizada.

- **Tries (Árvore de Prefixos):**
  - Uma árvore especializada em armazenar strings. Cada nó representa um caractere, e os caminhos da raiz até as folhas formam palavras.
  - É extremamente eficiente para operações como busca de prefixos e autocompletar, com complexidade O(L), onde L é o comprimento da string.

- **Estruturas para Consultas em Intervalos:**
  - **Segment Tree (Árvore de Segmentos):** Permite consultas de agregação (soma, mínimo, máximo) em um intervalo de um array e atualizações de forma eficiente (O(log n)).
  - **Fenwick Tree (ou Binary Indexed Tree):** Uma estrutura mais simples e com menos memória que a Segment Tree, ideal para cálculos de prefixo e soma cumulativa.

---

6. Análise de Complexidade

- **Notação Big O:** Descreve o comportamento assintótico de um algoritmo, ou seja, como seu tempo de execução ou uso de memória cresce à medida que o tamanho da entrada (n) aumenta.
- **Complexidade de Tempo vs. Espaço:** Avalia o tempo de processamento e a quantidade de memória utilizada. Muitas vezes, há um *trade-off* entre eles.
- **Comparação:** A análise de complexidade é a ferramenta crucial para comparar diferentes estruturas de dados e escolher a mais adequada para um problema. Por exemplo, uma busca em uma BST balanceada é O(log n), enquanto em uma lista encadeada é O(n).

Ordem Recomendada de Estudo e Aplicações Práticas

- **Ordem de Estudo:**
    1. **Iniciante:** Domine as estruturas lineares básicas (Arrays, Listas, Pilhas, Filas).
    2. **Intermediário:** Avance para estruturas não-lineares fundamentais (Árvores Binárias de Busca, Tabelas Hash, Grafos).
    3. **Avançado:** Explore estruturas de otimização (AVL/Red-Black, Tries, Segment Trees).

- **Aplicações Práticas:**
    - **Bancos de Dados:** Usam **Índices** (frequentemente B-Trees, uma variação de árvores) para acelerar buscas.
    - **Sistemas Operacionais:** Utilizam **filas** para escalonamento de processos e **listas encadeadas** para gerenciamento de memória.
    - **Redes de Computadores:** **Tabelas de roteamento** são essencialmente estruturas de dados (como tabelas hash ou tries) para direcionar pacotes de dados.
    - **Inteligência Artificial:** **Grafos** são usados para representar redes de conhecimento, mapas (para algoritmos de navegação como A*) e relacionamentos entre entidades.

Este resumo oferece um panorama geral do vasto campo das estruturas de dados, destacando que sua compreensão é indispensável para a criação de software eficiente e escalável.