UniSENAI

Trilha de Estrutura de Dados

Módulo 07: Algoritmo de Ordenação

Instruções para a melhor prática de Estudo

1. **Leia atentamente todo o conteúdo:** Antes de iniciar qualquer atividade, faça uma leitura detalhada do material fornecido na trilha, compreendendo os conceitos e os exemplos apresentados.

2. **Não se limite ao material da trilha:** Utilize o material da trilha como base, mas busque outros materiais de apoio, como livros, artigos acadêmicos, vídeos, e blogs especializados. Isso enriquecerá o entendimento sobre o tema.

3. **Explore a literatura:** Consulte livros e publicações reconhecidas na área, buscando expandir seu conhecimento além do que foi apresentado. A literatura acadêmica oferece uma base sólida para a compreensão de temas complexos.

4. **Realize todas as atividades propostas:** Conclua cada uma das atividades práticas e teóricas, garantindo que você esteja aplicando o conhecimento adquirido de maneira ativa.

5. **Evite o uso de Inteligência Artificial para resolução de atividades:** Utilize suas próprias habilidades e conhecimentos para resolver os exercícios. O aprendizado vem do esforço e da prática.

6. **Participe de debates:** Discuta os conteúdos estudados com professores, colegas e profissionais da área. O debate enriquece o entendimento e permite a troca de diferentes pontos de vista.

7. **Pratique regularmente:** Não deixe as atividades para a última hora. Pratique diariamente e revise o conteúdo com frequência para consolidar o aprendizado.

8. **Peça feedback:** Solicite o retorno dos professores sobre suas atividades e participe de discussões sobre os erros e acertos, utilizando o feedback para aprimorar suas habilidades.

Essas instruções são fundamentais para garantir um aprendizado profundo e eficaz ao longo das trilhas.

Algoritmos de Ordenação

1. Introdução aos Algoritmos de Ordenação

Os **algoritmos de ordenação** são fundamentais na ciência da computação, responsáveis por organizar elementos de uma lista em uma ordem específica, geralmente crescente ou decrescente. A eficiência desses algoritmos é medida em termos de **complexidade de tempo**, ou seja, quantas operações são necessárias à medida que o número de elementos cresce.

Existem diferentes algoritmos de ordenação, e eles podem ser classificados como **estáveis** ou **não estáveis**:

- **Estável**: Mantém a ordem relativa dos elementos iguais.
- **Não Estável**: Não garante a manutenção da ordem relativa dos elementos iguais.

---

2. Algoritmos de Ordenação Simples

2.1 Ordenação por Seleção (Selection Sort)

Funcionamento: A ordenação por seleção percorre a lista para encontrar o menor elemento e troca com o primeiro elemento. Repete o processo para os próximos elementos, até que a lista esteja ordenada.

Exemplo em Pseudocódigo:

```
function selectionSort(arr) {
    const n = arr.length;

    for(let i = 0; i < n - 1; i++) {
        let minIndex = i;

        for(let j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) { minIndex = j; }
        }

        // Troca os elementos
        if(minIndex !== i) {
            [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];
        }
    }

    return arr;
}
```

Exemplo de Uso

```
const numbers = [64, 25, 12, 22, 11];
console.log(selectionSort(numbers));

// Saída: [11, 12, 22, 25, 64]
```

- **Complexidade de Tempo**: O(n²) (sempre).
- **Estabilidade**: Não é estável.

2.2 Ordenação por Inserção (Insertion Sort)

Funcionamento: Este algoritmo constrói a lista ordenada um elemento por vez, comparando o elemento atual com os anteriores e movendo-o para a posição correta.

Exemplo em Pseudocódigo:

Insertion Sort em JavaScript

```
function insertionSort(arr) {
    const n = arr.length;

    for(let i = 1; i < n; i++) {
        const key = arr[i];
        let j = i - 1;

        while(j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }

    return arr;
}
```

Exemplo de Uso

```
const numbers = [12, 11, 13, 5, 6];
console.log(insertionSort(numbers));

// Saída: [5, 6, 11, 12, 13]
```

- **Complexidade de Tempo**: O(n²) (no pior caso), O(n) (no melhor caso, quando já está ordenada).
- **Estabilidade**: Estável.

2.3 Ordenação por Bolha (Bubble Sort)

Funcionamento: No bubble sort, os elementos adjacentes são comparados e trocados se estiverem na ordem errada. Esse processo é repetido até que não sejam mais necessárias trocas.

Exemplo em Pseudocódigo:

Bubble Sort em JavaScript

```javascript
function bubbleSort(arr) {
    const n = arr.length;

    for(let i = 0; i < n - 1; i++) {
        for(let j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j + 1]) {
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]]; // Troca dos elementos
            }
        }
    }

    return arr;
}
```

Exemplo de Uso

```javascript
const numbers = [5, 1, 4, 2, 8];
console.log(bubbleSort(numbers));

// Saída: [1, 2, 4, 5, 8]
```

- **Complexidade de Tempo**: O(n²) (sempre).
- **Estabilidade**: Estável.

---

3. Algoritmos de Ordenação Avançados

3.1 Merge Sort

Funcionamento: Merge Sort é um algoritmo baseado na técnica de **divisão e conquista**. Ele divide o array em duas metades, ordena cada metade recursivamente e, em seguida, combina as duas metades ordenadas.

Exemplo em Pseudocódigo:

📌No link vemos o exemplo completo referente a: Merge Sort - Ordenação Avançada

- **Complexidade de Tempo**: O(n log n) (sempre).
- **Estabilidade**: Estável.

3.2 Quick Sort

Funcionamento: Quick Sort também é baseado na técnica de **divisão e conquista**. Ele seleciona um **pivô** e divide o array em duas partes: uma com elementos menores que o pivô e outra com elementos maiores, e recursivamente ordena essas partes.

Exemplo em Pseudocódigo:

📌No link vemos o exemplo completo referente a: Quick Sort - Ordenação Avançada

- **Complexidade de Tempo**: O(n log n) (médio caso), O(n²) (pior caso).
- **Estabilidade**: Não é estável.

3.3 Heap Sort

Funcionamento: Heap Sort é baseado na construção de um **heap** (árvore binária balanceada). O algoritmo constrói um **max-heap**, onde o maior elemento fica no topo, e então remove o elemento do topo e refaz o heap até que todos os elementos estejam ordenados.

Exemplo em Pseudocódigo:

📌No link vemos o exemplo completo referente a: Heap Sort - Ordenação Avançada

- **Complexidade de Tempo**: O(n log n) (sempre).
- **Estabilidade**: Não é estável.

---

4. Ordenação Estável vs. Não Estável

- **Ordenação Estável**: Mantém a ordem relativa dos elementos com chaves iguais. Exemplos: **Merge Sort**, **Insertion Sort**, **Bubble Sort**.
- **Ordenação Não Estável**: Não garante a ordem relativa dos elementos com chaves iguais. Exemplos: **Quick Sort**, **Heap Sort**, **Selection Sort**.

Nota

Os algoritmos de ordenação são ferramentas fundamentais na organização de dados, com várias aplicações em sistemas computacionais. Cada algoritmo tem suas vantagens e desvantagens, dependendo do tipo de dados e do contexto em que será utilizado. A escolha entre algoritmos de ordenação simples e avançados, bem como entre algoritmos estáveis e não estáveis, pode influenciar diretamente a eficiência e o comportamento da aplicação.

Lista de Exercícios de Fixação

1. **Implementação Simples**:
    - Implemente o algoritmo de **Selection Sort** e teste com uma lista de 10 números inteiros. Calcule o número de comparações e trocas realizadas.
    - Modifique o **Insertion Sort** para funcionar com uma lista de strings e ordená-las alfabeticamente.
2. **Análise de Complexidade**:
    - Calcule a complexidade de tempo para o **Bubble Sort** ao ordenar uma lista de 100 elementos. Como a complexidade muda quando a lista já está ordenada?
    - Compare o número de comparações realizadas pelo **Selection Sort** e pelo **Insertion Sort** em uma lista de 10 elementos.
3. **Implementação Avançada**:
    - Implemente o **Merge Sort** e teste com uma lista de 50 números aleatórios. Verifique o tempo de execução comparado com o **Quick Sort**.
    - Modifique o **Quick Sort** para selecionar o pivô como o valor mediano de três elementos (início, meio e fim) e compare o desempenho com a versão original.
4. **Comparação entre Algoritmos**:
    - Compare o desempenho de **Merge Sort**, **Quick Sort** e **Heap Sort** com listas de 100, 1000 e 10.000 elementos aleatórios. Qual algoritmo apresenta o melhor desempenho?
    - Dado um conjunto de elementos com valores repetidos, implemente os algoritmos **Quick Sort** e **Merge Sort** e verifique qual preserva a ordem relativa dos elementos iguais.
5. **Desafio de Estabilidade**:
    - Implemente uma versão do **Heap Sort** que garanta a estabilidade da ordenação (mesmo que naturalmente não seja estável).
    - Implemente o **Selection Sort** de forma que ele se torne um algoritmo estável.