CESED – CENTRO DE ENSINO SUPERIOR E DESENVOLVIMENTO
CENTRO UNIVERSITÁRIO UNIFACISA

APLICAR ALGORITMOS DE APOIO À PESQUISA OPERACIONAL

PROF. JONHNANTHAN OLIVEIRA

EXPLICANDO A ESTRUTURA DE DADOS ESCOLHIDA
EXPLICANDO O ALGORITMO DE ORDENAÇÃO ESCOLHIDO

ALUNOS:
GUSTAVO TOMIO MAGALHÃES KUBO
SÉRGIO MAGNO CASTOR PINHEIRO
THIAGO LIMEIRA DE ALENCAR

CAMPINA GRANDE – PB
2024.2

Explicando a estrutura de dados escolhida

Ao analisarmos os requisitos do projeto buscamos entender quais as vantagens e desvantagens da utilização de cada uma das estruturas de dados propostas, conforme segue:

1. **Lista Encadeada**:

- **Vantagens:**
  - Muito boa para situações onde é preciso adicionar ou remover elementos com frequência, pois isso é rápido (tempo constante) apenas ajustando os ponteiros.
  - Não desperdiça espaço na memória, pois ocupa só o espaço necessário para os elementos que são adicionados.
- **Desvantagens**:
  - Usa mais memória, pois cada elemento precisa de um ponteiro extra que aponta para o próximo (ou anterior, se for uma lista duplamente encadeada).
  - Procurar um elemento específico pode ser mais lento, já que precisamos percorrer a lista inteira.

2. **Lista Sequencial (Estática)**:

- **Vantagens:**
  - Acesso rápido aos elementos, pois eles estão todos em sequência na memória, o que também facilita buscas diretas se soubermos a posição.
  - Muito eficiente para elementos de tamanho fixo, pois não exige ponteiros adicionais.
- **Desvantagens:**
  - Inserir e remover elementos no início ou no meio é complicado, pois é preciso deslocar os elementos para abrir ou fechar espaço, o que toma mais tempo.
  - O tamanho precisa ser decidido logo no início, ou ajustado depois, o que limita a flexibilidade e pode causar fragmentação de memória.

Em nosso caso concreto, entendemos que se fossemos utilizar a lista sequencial para implementação poderíamos nos deparar com dificuldades em relação aos seguintes aspectos:

1. **Inserção e Remoção de Elementos:**

- Para inserir um livro em uma posição específica ou remover um do meio, é preciso mover todos os elementos que estão depois dessa posição. Isso toma mais tempo, especialmente se a lista for grande.

- Adicionar um livro no começo da lista é ainda mais trabalhoso, pois precisamos mover todos os elementos um espaço para frente.

2. **Tamanho Fixo:**
   - A lista sequencial precisa de um tamanho fixo. Isso significa que, se ela encher, teremos que programar uma forma de expandir a lista, o que geralmente envolve criar uma lista maior e copiar tudo para a nova.
   - Se a lista for muito grande e não preenchermos tudo, teremos espaço desperdiçado.

3. **Complexidade de Memória**:
   - Quando aumentamos o tamanho da lista, isso pode criar fragmentação de memória, especialmente se o uso de inserções e remoções for constante.

4. **Ordenação**
   - Organizar uma lista sequencial exige mover elementos de suas posições atuais para novas posições.

5. **Acesso a Elementos**
   - Embora a lista sequencial seja rápida para acessar elementos por posição, se precisarmos procurar um elemento específico, ainda precisamos percorrer toda a lista. A lista encadeada pode ser implementada para ter um acesso mais prático em operações de ordenação e busca.

Desse modo, considerando as características do projeto, no qual a adição e remoção de livros são operações frequentes e não há um limite previamente definido para a quantidade de livros a ser armazenada optamos pela utilização da **lista encadeada,** que diferentemente de uma lista sequencial (ou estática), nos permite adicionar e remover elementos com mais flexibilidade e eficiência, sem a necessidade de reconfigurar ou deslocar elementos na memória. Essa abordagem evita problemas de desperdício de espaço e fragmentação de memória.

Explicando o algoritmo de ordenação escolhido

O Bubble Sort (ou Ordenação por Flutuação) é um algoritmo de ordenação muito simples e intuitivo. Ele é chamado assim porque funciona de forma semelhante ao modo como as bolhas sobem até a superfície da água. Em cada rodada do algoritmo, o maior elemento vai "subindo" até o fim da lista, enquanto os menores elementos vão se ajustando antes dele.

Seu funcionamento ocorre basicamente com os seguintes passos:

1. **Comparação de Pares**: O algoritmo começa comparando o primeiro elemento com o segundo. Se o primeiro é maior que o segundo, eles trocam de lugar. Em caso contrário, permanecem onde estão.
2. **Movimento Progressivo**: Depois de comparar o primeiro e o segundo, o algoritmo avança para o segundo e o terceiro elementos, e assim por diante, até alcançar o fim da lista.
3. **Repetição**: O processo de comparação e troca é repetido várias vezes em toda a lista. Cada vez que o algoritmo passa pela lista, o maior elemento "flutua" para o fim, ficando na posição correta.
4. **Parada do Algoritmo**: A ordenação termina quando o algoritmo passa pela lista sem precisar fazer nenhuma troca. Isso significa que todos os elementos já estão ordenados.

Porque utilizamos o do – while e não um for:

No do-while, a execução do bloco ocorre **pelo menos uma vez** antes de verificar a condição de repetição. Isso é importante para o Bubble Sort quando queremos garantir que a lista seja percorrida até que nenhuma troca seja necessária, sem nos preocuparmos com o estado inicial da lista. No do-while, o fluxo é o seguinte:

1. **Executa uma vez**: Realiza todas as comparações e trocas em uma passagem da lista.
2. **Verifica a condição**: Após cada passagem, verifica se houve trocas.
    - Se houve troca, o do-while repete a passagem para garantir que a lista fique ordenada.
    - Se não houve trocas, o loop termina, indicando que a lista já está ordenada.

Esse comportamento é conveniente para o Bubble Sort, pois precisamos sempre executar a primeira passagem para verificar se já está ordenado.

Um for também poderia fazer múltiplas passagens pela lista, mas ele não é ideal aqui porque a condição para encerrar o loop no Bubble Sort depende de uma **verificação após a execução** — ou seja, precisamos saber se houve trocas **depois de cada passagem completa**. Para conseguir isso com um for, teríamos que adicionar uma lógica extra para controlar manualmente o número de passagens e a verificação de trocas, o que deixaria o código menos intuitivo.

No caso do Bubble Sort:

Usar um for exigiria que nós inicializássemos e verificássemos uma variável de controle (troca) fora e dentro do loop para rastrear se o loop deve continuar, o que torna o código mais complexo.

Em resumo, o do-while é mais apropriado aqui porque:

1. Ele permite que o código execute pelo menos uma vez, independentemente do estado da lista.

2. A condição é verificada após a execução da passagem, simplificando a lógica de controle de trocas e repetições.

Esses pontos tornam o do-while uma escolha mais intuitiva e direta para o Bubble Sort, pois ele garante que a lista seja processada de maneira contínua até que não haja mais trocas a serem feitas.

Em face disso, ele foi escolhido neste projeto devido a simplicidade e adequação para listas pequenas, como a coleção de livros na nossa aplicação. Por ser um algoritmo fácil de entender, ele também facilita o aprendizado dos conceitos básicos de ordenação.