UNIVERSIDADE FEDERAL DE MINAS GERAIS
Faculdade de Engenharia de Produção

Trabalho Prático 1 - Múltiplas Ordenações

Tasso A. T. Pimmenta
2021072198

30 de janeiro de 2025

Sumário

Resumo

Este trabalho apresenta a implementação de uma estrutura de dados de lista sequencial e três algoritmos de ordenação: QuickSort, HeapSort e QuickInsertionSort. A lista sequencial foi escolhida devido à sua eficiência no acesso direto aos elementos, o que é benéfico para a ordenação. A ordenação é realizada de forma indireta, manipulando apenas os índices dos elementos, o que minimiza a realocação de memória. A análise de complexidade dos algoritmos é discutida, destacando suas eficiências e limitações. Resultados experimentais demonstram a eficácia dos algoritmos implementados, com destaque para a eficiência do QuickSort e QuickInsertionSort em comparação ao HeapSort. Conclui-se que a escolha da estrutura de dados e a técnica de ordenação indireta são adequadas para o problema proposto.

1 Introdução

O problema que vai ser enfrentado nesse trabalho se trata de uma ordenação por chaves. Basicamente, possuo três chaves: Nome, CPF e Endereço, e uma informação. O objetivo é a ordenação dos arquivos de acordo com a escolha de uma dessas chaves. A solução é criar uma estrutura de dados própria para armazenar os dados e que permita a ordenação por meio de três algoritmos de ordenação: QuickSort, HeapSort e QuickInsertionSort, por uma ordenação indireta, de modo que a movimentação de dados seja somente por parte dos índices e evite a realocação excessiva de memória.

2 Método

2.1 Estrutura da Lista Sequencial

A construção da lista sequencial foi realizada com o objetivo de armazenar os dados de forma eficiente e permitir a ordenação indireta por meio de três chaves: Nome, CPF e Endereço. A lista sequencial é implementada como uma classe template 'SequentialList', que armazena os elementos em um array dinâmico e mantém um array de índices para facilitar a ordenação indireta. A escolha de aplicar uma lista sequencial é primeiramente pela descrição do problema não ter uma limitação da estrutura. A lista sequencial é particularmente boa para ordenação por causa do acesso. Para uma pilha, acessar um elemento na posição k, eu preciso percorrer k elementos até o item (no caso de um acesso pelo começo, sem um ponteiro para o fim). Então, como o problema não me limitou e só mostrou um exemplo de uma para aplicação, eu percebi que tinha a liberdade para escolher a melhor estrutura que julgo ser a melhor para a ordenação, visto a quantidade de acessos que o algoritmo faz, evitando uma distância de pilha gigantesca. Com a lista sequencial, eu mostrarei que a distância de pilha é nula, pois, eu acesso o elemento diretamente.

2.2 Ordenação Indireta

A ordenação indireta é uma técnica onde a movimentação dos dados é feita apenas através dos índices, evitando a realocação excessiva de memória. Isso é particularmente útil quando se trabalha com grandes volumes de dados, pois minimiza o custo de cópia dos elementos. Na implementação, os

métodos de ordenação manipulam apenas os índices, deixando os dados originais armazenados para vizualização.

2.3 Métodos Principais

Os métodos principais da lista sequencial incluem:

- **insert**: Insere um novo elemento na lista, atualizando o array de índices.
- **operator[]**: Acessa um elemento da lista através do índice indireto.
- **swap**: Troca dois elementos na lista, manipulando apenas os índices.

2.4 Ordenação por Três Chaves

A ordenação por três chaves é implementada através da função 'getValue', que retorna o valor da chave especificada (Nome, CPF ou Endereço) para um dado 'DataRow'. A chave de ordenação é definida globalmente pela variável 'sortKey'.

2.5 Algoritmos de Ordenação

Os métodos de ordenação, como QuickSort, HeapSort e QuickInsertionSort, utilizam a lista sequencial e a ordenação indireta para realizar a ordenação de forma eficiente. Cada método de ordenação é implementado como uma classe template que recebe a lista sequencial como parâmetro e utiliza a função 'getValue' para comparar os elementos com base na chave de ordenação escolhida.

Essa abordagem modular e eficiente permite que a lista sequencial seja facilmente ordenada por diferentes chaves, utilizando diferentes algoritmos de ordenação, sem a necessidade de modificar a estrutura dos dados originais.

3 Análise de Complexidade

Nesta seção, analisamos a complexidade de tempo e espaço dos algoritmos de ordenação implementados, utilizando a notação assintótica.

3.1 QuickSort

O algoritmo QuickSort é um algoritmo de ordenação eficiente que, em média, possui uma complexidade de tempo de $O(n \log n)$. No entanto, no pior caso, sua complexidade de tempo é $O(n^2)$, que ocorre quando o pivô escolhido é sempre o menor ou o maior elemento da lista, podendo ser evitado escolhendo o elemento medio. A complexidade de espaço do QuickSort é $O(\log n)$ devido à profundidade da pilha de recursão.

3.2 QuickInsertionSort

O algoritmo QuickInsertionSort é uma combinação do QuickSort e do InsertionSort. Para sublistas pequenas (tamanho menor ou igual a 50), ele usa o InsertionSort, que tem uma complexidade de tempo de $O(n^2)$. Para listas maiores, ele usa o QuickSort, com a complexidade de tempo média de $O(n \log n)$. A complexidade de espaço é $O(\log n)$ devido à recursão do QuickSort. No geral, a complexidade de tempo do QuickInsertionSort é $O(n \log n)$ em média, mas pode se aproximar de $O(n^2)$ se a lista for quase ordenada e o InsertionSort for usado com frequência.

3.3 HeapSort

O algoritmo HeapSort é um algoritmo de ordenação in-place que possui uma complexidade de tempo de $O(n \log n)$ tanto no caso médio quanto no pior caso. Isso ocorre porque a construção do heap tem uma complexidade de $O(n)$ e cada remoção do maior elemento do heap tem uma complexidade de $O(\log n)$. A complexidade de espaço do HeapSort é $O(1)$, pois ele não requer espaço adicional além do necessário para armazenar a lista original.

4 Estratégias de Robustez

Contém a descrição, justificativa e implementação dos mecanismos de programação defensiva e tolerância a falhas implementados.

- **Validação de Entrada** Para garantir que o programa funcione corretamente, é importante validar as entradas fornecidas pelo usuário. No código, isso é feito verificando se o número de argumentos passados

para o programa é suficiente e se o arquivo de entrada pode ser aberto corretamente. Caso contrário, uma mensagem de erro é exibida e o programa é encerrado.

- **Tratamento de Exceções**

  O tratamento de exceções é utilizado para capturar e lidar com erros que possam ocorrer durante a execução do programa. No código, isso é feito ao tentar abrir o arquivo de entrada. Se o arquivo não puder ser aberto, uma exceção é lançada e uma mensagem de erro é exibida.

- **Gerenciamento de Memória**

  Para evitar problemas de memória, o código utiliza a função 'iniciaMemLog' para iniciar o registro de memória e 'finalizaMemLog' para finalizar o registro de memória após a execução dos algoritmos de ordenação. Isso garante que todos os acessos à memória sejam monitorados e que não haja vazamentos de memória.

- **Criação de Diretórios**

  Antes de salvar os arquivos de log, o código verifica se os diretórios necessários existem e os cria, se necessário. Isso é feito utilizando o comando 'mkdir -p', que cria os diretórios de forma recursiva, garantindo que a estrutura de diretórios esteja correta antes de salvar os arquivos.

- **Modularidade e Reutilização de Código**

  O código é organizado de forma modular, com cada algoritmo de ordenação implementado como uma classe separada. Isso facilita a manutenção e a reutilização do código, permitindo que novos algoritmos de ordenação sejam adicionados facilmente no futuro. Além disso, a utilização de templates permite que os algoritmos de ordenação sejam aplicados a diferentes tipos de listas, aumentando a flexibilidade do código.

5 Análise Experimental

Meu código foi testado com o dataset 'cad.r5000.p5000.xcsv', que era o maior dataset disponível para a análise. A ordenação de todos os algoritmos resultou no mesmo resultado para as chaves. Utilizei o 'AnalisaMem.cpp' (tive que refazer o arquivo porque o original não funcionava, não sei se isso causou algum erro).

5.0.1 QuickSort

- **Acessos e Tempo de Execução:** 135370 e 0.449738337s

5.0.2 QuickInsertionSort

- **Acessos e Tempo de Execução:** 134952 e 0.416776268s

5.0.3 HeapSort

- **Acessos e Tempo de Execução:** 329574 e 1.171602985s

Para demonstrar a eficiência do código, apresento os resultados da análise de memória para a chave Nome (o comportamento é o mesmo independentemente da chave, apenas o resultado muda) para os três algoritmos:

5.1 Análise dos Resultados

Os gráficos acima mostram o comportamento de acessos à memória de cada algoritmo. A seguir, destacamos alguns pontos importantes:

- **HeapSort:** O algoritmo HeapSort apresenta uma área de acesso muito grande devido à reordenação constante da lista (heapify). Isso resulta em um número elevado de leituras/escritas (329574) e um tempo de execução maior (1.171602985s).
- **QuickSort e QuickInsertionSort:** Ambos os algoritmos apresentam comportamentos semelhantes, com QuickInsertionSort mostrando uma leve vantagem em termos de tempo de execução (0.416776268s) e número de acessos (134952). A adição do InsertionSort reduz a distância dos endereços de memória acessados.

6 Conclusões

Os resultados mostram que os algoritmos 'QuickSort' e 'QuickInsertionSort' são mais eficientes em termos de tempo de execução e número de acessos à memória em comparação ao 'HeapSort'. A lista sequencial contribuiu para a eficiência dos algoritmos, facilitando o acesso direto aos elementos. A técnica de ordenação indireta também se mostrou eficaz, minimizando a realocação de memória.

É possível perceber o comportamento de cada algoritmo a partir dos graficos. A 'HeapSort' é um algoritmo muito interessante pois a ordenação dele é refeita quando retiramos ou inserimos o elementos, pois a heapfy é chamada para reordenar a lista, com isso a área de acesso fica muito grande.

É possivel perceber um comportamento paracido entre os ordenadores 'Quick', porém com a adição do insertion, há uma diminuição nos picos para leituras/escritas. Enquanto o 'Quick' normal tem vários picos até chegar no fim, com o insertion diminui para dois grandes apenas. Outro detalhe importante é que no gráfico é bem perceptível a escolha do elemento pivô.

6.0.1 Distância de Pilha

A distância de pilha foi 0, sendo sincero não sei se foi algum problema no log de registro, no código 'Analisamem.cpp' que eu tive que mexer ou se é por causa da estrutura de dados que escolhi. A minha conclusão sobre isso é que foi pela estrutura de dados, pois se não é uma pilha não existe distância de pilha, logo por acessar o elemento diretamente a referência de localidade não encontra distância.

Neste trabalho, implementei uma estrutura de dados de uma lista sequencial, além de três algoritmos de ordenação por chaves. Fiz por meio de templates de modo que eu conseguiria usar os algoritmos para outras estruturas de dados, como pilhas, por exemplo. Estava planejando implementar uma para comparações de distância de pilha e eficiência, mas visto a limitação de tamanho, não fiz.

Encontrei resultados positivos, a ordenação indireta por chaves foi bem-sucedida, encontrei resultados bons em tempo de execução. Na parte de distância de pilha a escolha da estrutura não foi tão satisfatório para a demonstração do comportamento.

Referências

[1] Eu fiz tudo sozinho