Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Estruturas de Dados

Mini SISU

Documentação

Vinícius Julião Ramos

Belo Horizonte, 11 de Dezembro de 2018

1. Introdução

O problema dado, chamado de Mini SISU, é uma implementação do Sistema Integrado de Seleção Unificada que o Ministério da Educação do Brasil usa para disponibilizar vagas nas universidades públicas. Tal sistema consiste na inscrição de diversos alunos em dois cursos diferentes, para se classificarem de acordo com suas notas obtidas em um exame previamente aplicado, sendo que a primeira escolha impera sobre a segunda, de forma que o aluno classificado na primeira opção não é incluído na lista de sua segunda escolha.

Para resolver o problema foram necessários algoritmos de ordenação e um pequena implementação lógica que executasse a regra de inserir o aluno/candidato na segunda opção dependendo de sua escolha principal.

Quanto às ferramentas para resolver o problema: das linguagens de programação propostas, selecionou-se C++11, pois a implementação por orientação à objetos traz o benefício de uma boa leitura de código, além de facilitar a manipulação de strings, que é necessário para ler as entradas do programa. Usou-se o controlador de versionamento GitHub para facilitar as etapas de implementação e o editor de texto utilizado foi o Visual Studio Code que possui alguns plugins para a linguagem C++ e viabilizaram a rápida resolução do problema.

2. Implementação

A implementação da solução do problema se deu pelo uso de orientação de objetos, e por tal motivo existem os métodos *get* e *set* que são base para o encapsulamento. Porém, focaremos na explicação dos métodos principais das classes responsáveis pela resolução da ordenação e na criação e articulação das estruturas de dados.

O uso de uma *Lista Duplamente Encadeada* [1], juntamente com a ordenação através do uso de uma versão um pouco modificada do método *insertion sort* [2], viabilizou que se inserissem itens, de forma ordenada, em uma lista, da qual, inicialmente não se sabe a quantidade de elementos necessários.

Também vale ressaltar, que buscou-se uma implementação simples, logo optou-se por não implementar *templates* em C++ para as classes **Lista** e **ItemLista,** uma vez que, para esse trabalho o conteúdo da lista trata-se apenas da classe **Aluno**. Além disso, os métodos e funções das classes são apenas os necessários para solucionar o problema.

A solução se deu, principalmente, por meio das seguintes classes e métodos:

a. Lista

Atributos:

ItemLista* = Ponteiro para o primeiro elemento da lista.

ItemLista* = Ponteiro para o último elemento da lista.

Métodos:

Lista(): Construtor que inicializa com uma célula cabeça.

void insere_fim(Aluno *aluno): Recebe um ponteiro de aluno e coloca-o como o último elemento da lista.

void insere_ordenado(Aluno *aluno): Recebe um ponteiro de aluno e insere na lista colocando-o como elemento posterior ao aluno de nota maior.

void libera_alunos(): Desaloca a memória alocada para cada **Aluno** inserido no heap.

b. Curso

Classe que herda de **Lista**, tal herança é justificada pelo fato de que cada curso se baseia numa lista com **Alunos** e mais alguns atributos, tais como:

Atributos:

std::string: Nome do curso.

int: quantidade de vagas.

int: quantidade de alunos já inseridos na lista.

float: nota de corte do curso.

Métodos:

Curso(): Construtor que inicializa com uma célula cabeça e com os elementos preenchidos por uma valor nulo.

insere_fim(Aluno *aluno): método sobrescritor que além de inserir um novo item ao fim da lista, também executa manipulações para saber a nota de corte.

void insere_ordenado(Aluno *aluno): método sobrescritor para a inserção do aluno na segunda opção do curso e para fazer manipulações que gravam a nota de corte.

imprime(): método responsável por varrer toda a lista e imprimir a saída esperada.

~Curso(): Destrutor que remove os **ItemLista** alocados em cada inserção de alunos.

c. ItemLista

Classe que faz o controle da *Lista Duplamente Encadeada*. Esta classe apenas salva os apontadores que conduzem a lista nos sentidos fim para início e início para fim.

Atributos:

ItemLista *: Elemento anterior ao atual.

Aluno *: Instância de **Aluno** que já está alocado anteriormente.

ItemLista *: Elemento seguinte ao atual.

d. Aluno

Objeto criado para armazenar os dados da entrada do programa, além de alguns atributos que ajudarão na questão de ordenação e na verificação da nota de corte de cada curso. Trata-se apenas de uma classe de armazenagem de dados, com métodos *get* e *set*.

Atributos:

std::string: Nome do aluno
int: Primeira opção de curso selecionado
int: Segunda opção de curso selecionado
float: Nota do aluno
int: Posição do aluno em sua primeira opção de curso.

e. main

Neste ponto do código ocorrem as chamadas aos métodos que dão a resolução do problema. A sequência de cada chamada pode ser descrita por:

- 1) Criação do array de Cursos.
- 2) Criação da **Lista** inicial de alunos:
  A lista inicial é criada a partir da inserção dos elementos na entrada do programa. Utiliza-se o método **Lista::insere_ordenado()** para cada registro de **Aluno**.
- 3) Inserção dos alunos na primeira opção do curso
  Varredura completa de todos os alunos da lista inicial e inserção dos alunos no fim de cada lista do respectivo **Curso** candidatado; através do uso do método **Curso::insere_fim()**.
- 4) Inserção dos alunos na segunda opção do curso
  Para cada **Aluno** presente na lista inicial, obtém-se o seu respectivo **Curso** de segunda opção e se tal **Aluno** não estiver classificado na primeira opção, então faz-se a inserção ordenada deste candidato na lista do **Curso** através do método **Curso::insere_ordenado()**.
- 5) Imprimir os classificados e listas de espera para cada curso:
  Para cada **Curso** registrado, chama-se o método **Curso::imprime()**.

3. Referencial Teórico e Análise de complexidade

Uma vez que o número de alunos/candidatos é menor ou igual a 1024 e a quantidade de cursos é menor ou igual a 128, e tais números são considerados pequenos quando o assunto é quantidade computacional, o método utilizado para a ordenação dos alunos, de acordo com as notas, envolve inserir todos os alunos em

uma lista à medida que os dados dos candidatos são inseridos no programa. Esse método é uma adaptação do algoritmo chamado de *insertion sort* [2] que possui função de complexidade $\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$.

Visto que o professor solicitou a implementação do problema através do uso de listas encadeadas, preferiu-se utilizar listas de duplo encadeamento, pois possuem um custo baixo quando se deseja adicionar elementos em qualquer posição diferente do fim, além de viabilizar a navegação em duas direções (fim para início e início para o fim), o que tornou a inserção menos custosa quando desejou-se investigar tanto os elementos anteriores quanto os elementos posteriores de um certo item da lista.

Outro ponto a ser abordado é o uso de memória: Cada item da lista (**ItemLista**) recebe um ponteiro de aluno (**Aluno***), como conteúdo principal, dessa maneira, apenas no início do programa se aloca memória para os alunos. Nas demais partes do código, aloca-se apenas memória para os itens da lista, pois os alunos são referenciados a partir da alocação inicial.

a) Montagem da lista inicial:

Como já mencionado acima, nesse instante do código, a cada aluno/candidato inserido na entrada do programa, insere-o de forma ordenada na lista. Para tanto, usa-se o método *insertion sort* modificado e temos uma análise assintótica de $O(n^2)$ na inserção ordenada de n candidatos.

O custo de memória nesse ponto é igual a $n \times$ (**Aluno**) + $n \times$ (**ItemLista**).

b) Vetor de Cursos:

Como é necessário guardar cada índice dos m cursos dos quais os aluno podem se matricular, faz-se necessário criar um vetor de cursos que se responsabilizará por tal armazenamento de candidatos.

O custo de memória nesse ponto é igual a $m \times$ (**Curso**).

c) Inserção da primeira opção dos candidatos

Uma vez que ordenamos os n alunos por nota no início do código, o custo da inserção de todos os alunos em seus respectivos cursos de primeira opção foi apenas de $n = O(n)$, pois bastou percorrer completamente a lista inicial e inserir os candidatos no final da lista de cada curso.

d) Inserção da segunda opção dos candidatos

Na inserção dos alunos na segunda opção de curso também implementou-se o método *insertion sort.* Entretanto, para a análise do custo desta inserção, deve-se imaginar a quantidade máxima de **ItemLista** criados para que se consiga obter a quantidade de vezes se percorreu a lista de cada curso a fim de ordenar os alunos,

no pior caso. É válido lembrar que a quantidade de **ItemLista** representa a quantidade de relações criadas entre **Curso** e **Aluno**.

Tem-se que cada um dos m **Cursos** possui n_m **Alunos** e como cada um dos n **Alunos** dados na entrada podem se inserir em no máximo dois cursos, o valor máximo de **ItemLista** existentes para todos os cursos é de $2n = \sum_{i=1}^{m} n_m = N$. Logo, é possível observar que o pior caso se trata da necessidade de reordenar todos esse N **Alunos** e o custo se baseia no mesmo critério de *insertion sort*: $\sum_{i=1}^{N-1} i = \frac{(N-1)(N)}{2} = \frac{N^2}{2} - \frac{N}{2} = O(N^2) = O(4n^2) = O(n^2)$.

Ainda há uma diferença no uso de memória dessa etapa. A fim de salvar o valor para a nota de corte, para cada inserção, há a necessidade de uma manipulação em alguns dados da lista do curso, por isso fez-se necessário o uso de recursos a mais em comparação ao método que insere na primeira opção. Estes são: 3 inteiros (**int**), 1 ponteiro para **ItemLista** , 2 comparações e mais 1 atribuição.

e) Uso da memória na inserção nos cursos

Como a classe **Curso** herda de uma **Lista**, temos que a cada adição de um item em um curso, precisa-se de 2 apontadores para **ItemLista** e 1 apontador para **Aluno**, porém não é mais necessário alocar memória para os alunos, pois estes foram alocados no início do programa. Assim, como cada **Curso** tem n_m alunos (**ItemLista**), ao final de todas as inserções terá-se N **ItemLista** alocados, sendo que $n \leq N \leq 2n$, onde o cenário mínimo de N representa quando todos os alunos são aprovados na primeira opção (assim eles não entrarão na lista da segunda opção) e o cenário máximo representa todos os candidatos não sendo aprovados na primeira opção de curso (assim os alunos entrarão como excedente da primeira opção e também serão alocados na lista de aprovados ou de espera da segunda opção).

Vale lembrar que 1 unidade da medida que chamamos de "custo" nos itens **c)** e **d)** nesse texto se refere à todas as operações que envolvem a inserção de um novo item na lista dos cursos: troca dos ponteiros anterior e posterior, incremento no número da quantidade de candidatos inseridos, alocação de uma nova posição no "heap", comparação do valor entre o elemento anterior e o elemento posterior, comparação da troca no valor da nota de corte, além da soma executada pelo "navegador" do array de cursos.

f) Custo geral

A análise assintótica das etapas principais do programa (**a, c** e **d**) constituem os seguintes custos: $O(n^2)$, $O(n)$, $O(n^2)$. Dessa maneira o custo geral do dessa solução é $O(n^2)$.

O custo geral de memória, é referente à soma de algumas alocações: N **ItemLista** dos cursos, tal que para n **Alunos** alocados, temos $n \leq N \leq 2n$, m **Cursos**, uma alocação para a **Lista** inicial, a qual possui ainda n **ItemLista**. Deve-se somar, ainda, o método chamado que tem o maior custo de espaço, pois assim sabe-se a necessidade máxima da memória do programa, que refere-se ao método **Curso::insere_ordenado()**. Dessa maneira nosso custo de memória é uma função $f(mn)$ que representa tem custo máximo (assintótico) dado por:

$$1 \times LISTA + 3n \times ITEMLISTA + n \times ALUNO + m \times CURSO.$$

4. Conclusão

Por conta do baixo número de inscritos e de cursos, a solução apresentada atende à demanda do problema com excelência, visto que na questão do espaço, a intenção do programa é poupar o uso de memória, não exercendo novas alocações, em conjunto com um bom desempenho computacional referente aos algoritmos ensinados em sala de aula, até então, e da necessidade do uso da estrutura *Lista Encadeada*.

Porém, em outro cenário, em que o número de inscritos é muito alto, uma solução com custo de execução $O(n^2)$ não é viável, e nem o uso de *Lista Encadeada*, uma vez que tal estrutura depende da alocação de outros ponteiros. Em um caso de otimização, a implementação de vetores e juntamente com um algoritmo de ordenação mais eficaz (como o *quick sort* [3], por exemplo) traria o benefício de um menor custo de complexidade e de uso de memória.

5. Bibliografia

1 - CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L.; STEIN, Clifford. ***Algoritmos:*** *teoria e prática. Rio de Janeiro: Campus, 2002. página 166.*

2 - CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L.; STEIN, Clifford. ***Algoritmos:*** *teoria e prática. Rio de Janeiro: Campus, 2002. página 11.*

3 - CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L.; STEIN, Clifford. ***Algoritmos:*** *teoria e prática. Rio de Janeiro: Campus, 2002. página 117.*