Trabalho prático 1 de Estruturas de Dados

Professor Luiz Chaimowicz, 2019/1

Ábner de Marcos Neves
2018054605

1. Introdução

Vendo o sistema brasileiro de seleção de universitários, Elsa, rainha de Arendelle, ficou interessada em implantar uma plataforma parecida em seu reino. Para isso, contratou os alunos de Estruturas de Dados da UFMG, aproveitando também para fazê-los praticar o conteúdo visto em sala de aula até então: listas encadeadas e análise de complexidade.

2. Implementação

O algoritmo criado toma base em 5 classes, que são explanadas a seguir:

1. **Aluno**: objetos dessa classe têm atributos referentes a seu nome, nota, primeira e segunda opções de curso. Os métodos são aqueles básicos: construtor, destrutor, *getters* e um que imprime as informações.

2. **Curso**: essa classe tem atributos para o nome do curso, quantidade de vagas, nota de corte e duas listas da classe Lista: uma para os alunos classificados e outra para a lista de espera. Além dos métodos básicos, ela possui um mais importante: o método *classificar*, que recebe como parâmetro um ponteiro para Aluno e o insere na sua lista de classificados, caso haja vaga. Se houver uma vaga e ela for a última, esse método também define a nota de corte do curso como sendo a nota desse aluno. Se não tiver vaga, então o aluno é inserido na lista de espera. Ainda, se o aluno foi classificado, então esse método retorna *true*; caso contrário, retorna *false*.

3. **Celula**: é uma classe *template* que, nesse trabalho, é usada para os tipos Aluno* e Curso*. Foi feita para integrar a classe Lista. Possui um atributo que guarda sua posição em relação à lista de que faz parte e também tem dois apontadores pra outras células do tipo T: um para a próxima e outro para a anterior. Além dos métodos básicos como *getters* e *setters*, ela possui o método *trocar*, que recebe como parâmetro uma outra célula e troca as duas de lugar, manipulando os apontadores dentro de cada uma das duas células e daquelas ao redor, tomando cuidado também quando são células das extremidades.

4. **Lista**: escolhi implementar uma lista duplamente encadeada pensando em diminuir o custo da ordenação da lista e deslocamentos em geral: ao trazer para frente algum aluno, não seria necessário percorrer novamente do início da lista até os alunos anteriores àquele em questão, pois teríamos um ponteiro diretamente para eles. Essa classe possui ponteiros para a primeira e última

células e também os métodos padrão vistos em aula, como inserir e retirar. Esse último, entretanto, é "subdividido" em três: *get_celula*, que retorna a célula numa determinada posição; *get_objeto*, que retorna o objeto presente na célula em uma dada posição; e *remover*, que apenas remove a célula da lista. Ao longo do programa, percebi que fazer essa divisão seria mais útil do que um único método que, ao mesmo tempo, remove e retorna um elemento. Lista também é uma classe *template* que, nesse trabalho, foi usada para os tipos Aluno* e Curso*.

5. **MiniSisu**: essa classe possui uma lista para os cursos e outra para os alunos. O método de leitura de dados recebe da entrada padrão como especificado no enunciado do trabalho, cria objetos do tipo Aluno* e Curso* com o que leu e povoa essas duas listas com eles. Além desse e de outros mais básicos, MiniSisu tem também os métodos *ordenar_alunos*, *desempatar* e *classificar*, que são explicados com mais detalhes nos itens a seguir.

2.1. Ordenação

Para classificar os alunos, a classe MiniSisu precisa ordená-los decrescentemente de acordo com suas notas. O algoritmo escolhido foi o *insertion sort*, pois vimos em sala de aula e eu estava mais familiarizado com ele. No pior caso, sua complexidade em relação às operações de troca é $O(n^2)$, que acontece quando a lista está inversamente ordenada. Quanto à memória, o *insertion sort* é $O(n)$.

Essa ordenação foi muito útil também para os critérios de desempate, já que os alunos com notas iguais não são trocados de lugar e, assim, permanecem na ordem de inscrição, facilitando a classificação mais tarde.

2.2. Desempate

O método para desempate recebe como parâmetros dois inteiros que indicam posições dentro da lista de Alunos inscritos. Essas posições delimitam um intervalo em que todos os alunos têm a mesma nota.

Como o primeiro critério de desempate é a prioridade por primeira opção de curso, esse intervalo é percorrido do início ao fim e tenta classificar, em ordem, cada um dos alunos em suas primeiras opções. Se um aluno é classificado, ele é retirado da lista que a classe MiniSisu mantém com todos os alunos a serem selecionados. Mas se ele não for classificado, então ele continua nela e nada mais é feito por enquanto. Vale lembrar que o método *classificar* que é chamado pela primeira opção de curso automaticamente insere o aluno na lista de espera, caso ele não seja aprovado.

Depois disso, ainda dentro do intervalo (com as posições devidamente atualizadas), acaba sobrando apenas os alunos que não foram classificados em suas primeiras opções. Então, passamos por eles e tentamos classificá-los em suas segundas opções. Assim, eles serão automaticamente inseridos na lista de classificados ou na lista de espera do respectivo curso e, de qualquer forma, serão agora removidos da lista mantida pela classe MiniSisu.

Aquela vantagem do *insertion sort* citada anteriormente entra aqui. Note que não foi preciso me preocupar com o segundo critério de desempate, a ordem de chegada. Como

os alunos só são levados para a frente da lista enquanto suas notas são *maiores* que as dos anteriores, aqueles que têm notas iguais permanecem em ordem de chegada entre si. Dessa forma, tentar classificá-los sequencialmente em suas primeiras opções já leva em conta o primeiro critério e o segundo. Do mesmo jeito, tentar classificar sequencialmente em suas segundas opções os alunos que sobraram também já leva em conta o segundo critério de desempate.

Estruturalmente, esse método tem dois *loops* (um depois do outro) que percorrem, no pior caso, todos os alunos do intervalo. E esse intervalo, no pior caso, tem todos os alunos inscritos. Portanto, a complexidade do método de desempate é O(n), no pior caso.

2.3. Classificação

Antes de classificar os alunos, eles são ordenados em ordem decrescente de nota. Depois disso, começa do primeiro aluno, o que tem a maior nota, e verifica se ele está empatado com outros. Se não estiver empatado com ninguém, o método tenta classificá-lo na primeira opção de curso e, depois, na segunda, se for o caso. Se estiver empatado, o algoritmo encontra o intervalo que contém todos os alunos com a nota igual à dele e chama o método de desempate descrito acima. Dessa forma, tenta classificar todos os empatados de uma vez e o processo se repete a partir do próximo aluno, que já vai ter uma nota menor.

3. Análise de complexidade

A princípio, pode parecer que o método *classificar* é $O(n^2)$ por ter um loop dentro do outro, mas, na verdade, isso não acontece. Tomando como operações relevantes as iterações ao longo da lista de alunos inscritos, esse método é O(n). Isso acontece porque os loops encadeados são, digamos, mutuamente exclusivos: os elementos iterados pelo loop mais externo não são iterados pelo mais interno.

A cada aluno iterado pelo primeiro loop, o segundo passa pelos próximos alunos que têm notas iguais ao aluno em questão. Se tiver mais de um, então eles serão passados para o desempate e lá acabarão sendo removidos da lista que está sendo iterada, o que impede que eles passem pelo laço externo depois. Agora, se nem o aluno seguinte tem nota igual, então o aluno em questão já passará diretamente pelo método de classificação do seu curso, que tem custo constante.

Dessa forma, ou um aluno passa pelo loop externo uma vez e é classificado com custo constante, o que gera uma complexidade resultante de O(n), pois $n*O(1) = O(n)$; ou o aluno passa pelo loop interno junto com um grupo de outros alunos, resultando numa complexidade também O(n), e, depois de saírem do loop interno, passam pelo desempate, que também é O(n). Isso tudo resulta em $3*O(n) = O(n)$.

Entretanto, fiz essa análise ignorando o método de ordenação do vetor, que é $O(n^2)$. Por causa dele, o algoritmo todo também se torna $O(n^2)$.

5. Considerações

Esse trabalho foi um bom exercício do conteúdo estudado em sala de aula. A implementação das estruturas e da ordenação foram o que mais demandou tempo, principalmente pela manipulação dos ponteiros. Mas minha maior dificuldade foi na logística, pois meu computador estragou quando comecei o trabalho e fiquei dependendo dos computadores da UFMG, o que atrapalhou muito o desenvolvimento e impediu que eu terminasse de debugar o processo de classificação, que não está funcionando para todos os casos. Gostaria de ter conseguido corrigir o algoritmo e tê-lo refatorado mais.

4. Referências

https://www.stackoverflow.com

https://www.cprogramming.com/debugging/segfaults.html