Trabalho Prático 1
Acesso ao Ensino Superior em Arendelle

~~**Nome do(a) autor(a)**~~

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

email@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema semelhante ao Sistema de Seleção Unificado (SISU) do Brasil. Para um conjunto de alunos e de cursos, deveria-se classificá-los em suas respectivas opções de cursos, levando em conta as notas obtidas. Aqueles com maiores notas seriam colocados à frente. Em caso de empate, tem-se critérios para resolver a questão. Além disso, é importante manejar as listas de espera dos cursos, que são preenchidas com aqueles candidatos que eventualmente não se classificaram dentre as vagas disponíveis.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de uma lista duplamente encadeada. A preferência desta ao invés daquela simplesmente encadeada deu-se pois a duplamente encadeada possui custo constante para a remoção de elementos no fim dela, além de ter metade do custo para o acesso a um elemento na segunda metade da lista.

Essa estrutura de dados foi montada numa classe e colocada como um *template*, uma ferramenta disponibilizada pela linguagem C++. Como visto em aula, implementou-se a lista com a chamada ”célula-cabeça” antes do primeiro elemento, visando facilitar e, assim, reduzir o custo assintótico de algumas operações. Além disso, um dos membros da lista guarda a quantidade de elementos que ela possui, evitando um custo linear desnecessário (de iterar pela lista inteira contando o número de elementos) caso não houvesse esse atributo com essa informação.

Um diagrama esquemático da lista duplamente encadeada implementada pode ser visto na figura.

As principais funções, como os construtores e os destrutores, a *Vazia*, bem como as operações de inserção e remoção nas várias posições da lista são adaptações dos algoritmos vistos em sala. Teve-se, no caso, uma atenção um pouco maior, devido à existência também do ponteiro *anterior* em cada célula da lista.

Tendo como base a implementação da lista encadeada observada em [Ziviani 2006] ainda foi criado um iterador, um outro atributo da classe ListaEncadeada,

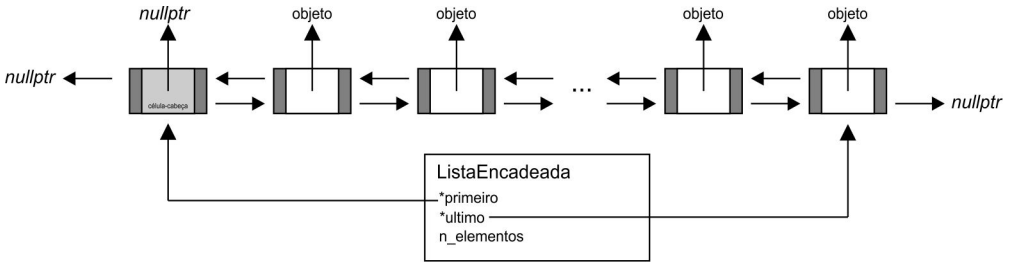

Figure 1. Diagrama da Lista

que é um ponteiro para elemento da lista. Este possibilita o acesso e também a modificação de elementos da lista. Essa funcionalidade utiliza dois métodos: *primeiro* que coloca o iterador na primeira posição e retorna um ponteiro para o objeto nessa posição (naturalmente imediatamente após a célula cabeça); e *proximo*, que "anda" com o iterador uma posição e retorna o ponteiro para o elemento dessa nova posição. Se atingíssemos o fim da lista, o iterador se tornava *nulltpr*.

2.2. Classes

Para modularizar a implementação, foram montadas três classes principais e uma struct. A primeira delas é a lista duplamente encadeada abordada acima, cujas células são structs, com dois ponteiros, um para a próxima célula, outro para a anterior e um ponteiro para o elemento (um objeto) que está sendo guardado nessa célula e, consequentemente na lista.

A lista possui alguns métodos úteis, como o *get_indice* para receber o índice de um elemento e outros métodos específicos como o Adiciona, pertencente à Lista de Cursos, que faz o início do processo de classificação.

As outras duas classes são Candidato e Curso. Elas foram criadas para armazenarem os dados das entradas e processá-los. A classe Candidato basicamente guardava as informações fornecidas de cada aluno, tal como nome, nota e opções de cursos. A classe Curso era mais complexa pois deveria compreender duas listas: a de classificados e a de espera, tendo que gerir os candidatos que estavam sendo adicionados, alocando eles apropriadamente segundo os critérios de classificação e desempate. Além disso, esta ainda tinha atributos como o nome do curso, a nota de corte e a quantidade de vagas disponíveis.

2.3. Classificação e Desempate

O sistema de classificação e desempate segundo as regras do SISU é um dos trechos mais importantes da implementação, sobretudo pois ele é quem maneja os candidatos que vão sendo adicionados.

Em prol da modularização, preferiu-se separar obrigações. O Curso ficaria responsável por decidir se o candidato iria ser classificado ou se iria para a lista de espera daquele curso. Se ele fosse para a lista de espera, quem teria que controlar as consequências disso era a Lista de Cursos, que é criada quando o programa inicia, requisitando a entrada do candidato em sua segunda opção, por exemplo.

Essa "comunicação" entre esses dois módulos é feita através de um valor de retorno, definida por um inteiro, que a classe Curso retornava para a Lista de Cursos. Como

comentado no código, um retorno 0 significa que o candidato tinha sido colocado na lista de classificados e, assim, a princípio, não precisaria ser colocado em quaisquer outras listas ou cursos. Um retorno 1 significa que o candidato foi classificado, mas "empurrou" outro aluno para a lista de espera. Este "empurrado" deve ser organizado na lista de espera daquele curso e ser adicionado no sua segunda opção, a princípio. Por fim, um retorno 2 significa que o candidato foi direto para a lista de espera daquele curso, ou seja, deverá, a priori, ser colocado na sua segunda opção também.

A Lista de Cursos recebe essa comunicação e toma as devidas providências. Se o retorno foi zero, nada deve ser feito e seguimos para o próximo candidato. Se o retorno for 1, ele deve adicionar o candidato que foi "empurrado" na sua segunda opção de curso, mas se aquela já for a segunda opção, nada deve ser feito. Se o retorno for 2, ele apenas deve adicionar o aluno na sua segunda opção, se aquela já não for. Esse processo de adição a um novo curso também tem o mesmo estilo de retorno, que leva novamente a uma "análise do empurrado" caso haja alguma alteração nesse sentido.

3. Análise Complexidade

3.1. Tempo

Começaremos a análise para a simples adição de um candidato a um curso, então partiremos para a análise também se ele tiver que ser realocado em outras listas de espera.

Um curso estando vazio, o custo para adicionar um novo candidato é o curto de adicionar um novo elemento no início da lista de classificados, no caso $O(1)$, um custo constante. Se curso já tiver alguns alunos na lista de classificados, a situação é mais complexa. Primeiro, há uma operação com custo, na pior hipótese, $O(n)$, uma pesquisa sequencial pela lista procurando um candidato com nota menor ou igual à do novo aluno. Se o novo tiver nota menor, adicionamos à última posição da lista de classificados com custo $O(1)$.

Se o novo candidato tiver uma nota empatada com outro que já está na lista, teremos que iterar por ela (para garantir que não há outros empatados com essa mesma nota), com custo $O(n)$ no pior caso e decidir o desempate. No desempate: custos constantes para comparação dos índices das opções de cursos. Mas no final das contas, teremos que adicionar o novo candidato ou antes ou depois de algum candidato da lista numa posição específica *i*. Isso tem o custo $O(n)$ para encontrar o índice e mais um $O(n)$, para adicionar na i-ésima posição, ambos no pior caso, considerando a implementação feita. Em suma, temos

$$O(n) + O(n) + O(n) + O(n) = O(n)$$

para o custo de adicionar um candidato, no pior caso, na lista de classificação.

Caso a adição de um novo candidato extrapole o número de vagas, teremos que remover o último da lista dos classificados, a custo constante, pois a lista é duplamente encadeada (um dos motivos pela escolha desta ao invés da simplesmente) e adicioná-lo na lista de espera, que têm o mesmo custo acima explanado, pois a ideia geral é a mesma. Ou seja, o custo para adicionar um candidato num curso tem um custo linear com o tamanho da lista.

Agora temos também que analisar o que acontece quando temos que colocar um candidato que foi "empurrado" para a lista de espera em uma das suas opções de curso. A

lista de cursos tem N cursos, definidos pela entrada. Para candidato, teremos o seguinte. Um custo $O(N)$ para pesquisar na lista a primeira opção do novo aluno. Em seguida, um custo linear $O(n)$ com o número de candidatos que estão nas listas de espera ou de classificados daquele curso. Se foi classificado direto, não há mais custo, seguimos para o próximo candidato. Se o novo foi colocado na lista de espera, temos mais um $O(n)$ para colocá-lo na sua segunda opção. Então até o momento, o custo para adicionar um candidato, entre N cursos e n candidatos já nas listas de classificação e de espera

$$O(N) + O(n) + O(n) = O(max(N, n))$$

Se o novo foi classificado, mas "empurrou" alguém, temos uma situação mais complicada, temos que analisar a situação do "empurrado" com o método *AnaliseEmpurrado*. Qualquer que seja essa situação, teremos um custo de $O(N)$ para pesquisar a segunda opção dele e mais um $O(n)$ para adicioná-lo a esse curso, ou simplesmente arrumar a lista de espera do curso que ele foi empurrado. O problema é quando o "empurrado" empurra outro candidato do segundo curso, o que irá criar uma chamada recursiva para o método *AnaliseEmpurrado*. Isso só termina, num pior caso em que todos os empurrados se classificam para suas segundas opções e empurram outro candidato, quando todos os cursos não tem mais vagas. Na pior das hipóteses teremos que chamar essa função para cada um dos n candidatos que estão nas listas de classificados. Em outras palavras, vê-se que a função é da ordem de $O(n^2)$.

No final, tem-se que

$$O(max(N, n)) + O(n^2) = O(n^2)$$

ou seja, o custo para adicionar um candidato é da ordem de n^2, sendo n o número de candidatos nas listas.

3.2. Espaço

Vamos considerar que cada candidato ocupe uma unidade de espaço, para nossa análise. Assim, na pior hipótese do programa, todos os candidatos estão em dois locais: na lista de espera de seu primeiro curso e na lista de classificados ou de espera da segunda opção. Com isso, numa hipótese extrema, para uma entrada de n candidatos, o programa ocupa o espaço de $2n$ candidatos, ou então, $O(2n) = O(n)$

4. Conclusão

Após a implementação do programa, pode-se notar que havia duas partes distintas para o desenvolvimento do trabalho. A primeira era implementar a estrutura de dados escolhida de forma correta, coesa e funcional para poder dar suporte à parte seguinte. Essa segunda parte tinha como objetivo utilizar a estrutra de dados criada anteriormente para criar os algoritmos que efetivamente fariam a seleção e a classificação do "Mini SISU". Havia ainda uma pequena e mais simples implementação que compreendia a leitura da entrada e escrita da saída dos dados.

O grande esforço, sobretudo para o desenvolvimento da segunda parte, era de considerar os vários casos que poderiam ocorrer e fazer isso tendo em vista o modo como seriam transformados os critérios de desempate em algoritmo. Além disso, tinha-se que

estar atento algumas das boas práticas de programação como modularização de trechos do código e de encapsulamento, a fim de destinar à devida classe, no caso Curso ou ListaEncadeada, a referida responsabilidade. Assim, momentos de refatoração, sobretudo do trecho do código que realizava essas ações, foram bastante necessários para deixar o algoritmo mais legível, com menos repetições e com métodos menores.

Ao fim desse trabalho, ficou ainda mais evidente a importância dos testes de unidade e da refatoração. Ambas práticas possibilitaram um desenvolvimento mais produtivo e consciente de que, com a implementação de novas funcionalidades ou com a refatoração de trechos do código, as estruturas do programa continuavam corretas e funcionais. Além disso, pode-se protagonizar dois papéis como programador: aquele que cria a estrutura de dados e aquele que utiliza a estrutura de dados.

References

Ziviani, N. (2006). *Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas*. Editora Cengage.