Trabalho Pra´tico 1

Acesso ao Ensino Superior em Arendelle

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

**Arthur Souto Lima**

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

[arthursl@ufmg.br](mailto:email-do-autor@ufmg.br)

# Introduc¸a˜o

[email@ufmg.br](mailto:email-do-autor@ufmg.br)

O problema proposto foi implementar um sistema semelhante ao Sistema de Selec¸a˜o Unificado (SISU) do Brasil. Para um conjunto de alunos e de cursos, deveria-se clas- sifica´-los em suas respectivas opc¸o˜es de cursos, levando em conta as notas obtidas. Aque- les com maiores notas seriam colocados a` frente. Em caso de empate, tem-se crite´rios para resolver a questa˜o. Ale´m disso, e´ importante manejar as listas de espera dos cur- sos, que sa˜o preenchidas com aqueles candidatos que eventualmente na˜o se classificaram dentre as vagas dispon´ıveis.

# Implementac¸a˜o

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

## Estrutura de Dados

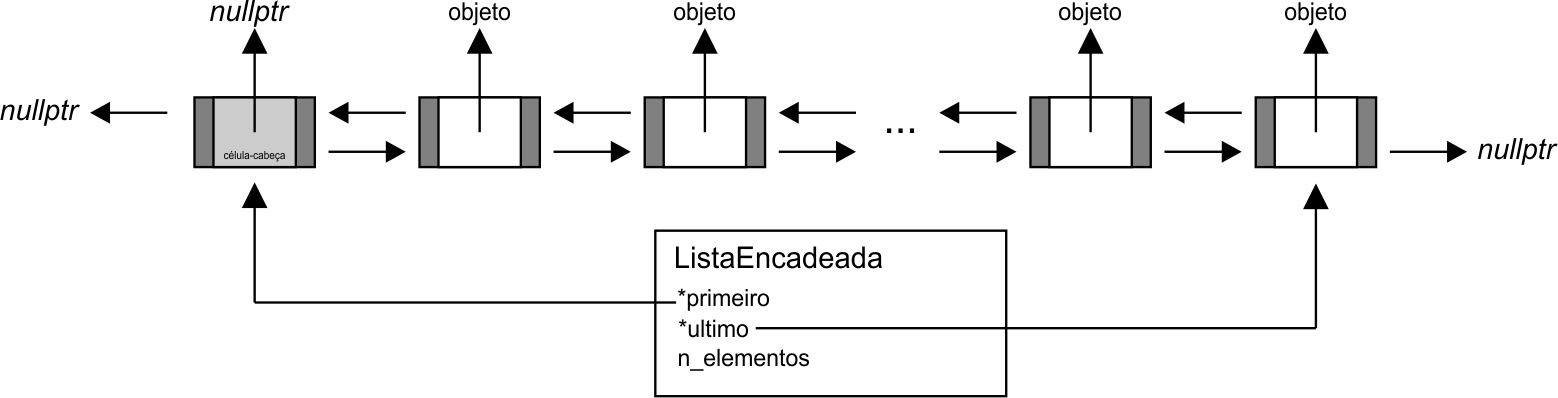
A implementac¸a˜o do programa teve como base a estrutura de dados de uma lista dupla- mente encadeada. A prefereˆncia desta ao inve´s daquela simplesmente encadeada deu-se pois a duplamente encadeada possui custo constante para a remoc¸a˜o de elementos no fim dela, ale´m de ter metade do custo para o 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 ”ce´lula-cabec¸a” antes do primeiro elemento, visando facilitar e, assim, reduzir o custo assinto´tico de algumas operac¸o˜es. Ale´m disso, um dos mem- bros da lista guarda a quantidade de elementos que ela possui, evitando um custo linear desnecessa´rio (de iterar pela lista inteira contando o nu´mero de elementos) caso na˜o hou- vesse esse atributo com essa informac¸a˜o.

Um diagrama esquema´tico da lista duplamente encadeada implementada pode ser visto na figura.

As principais func¸o˜es, como os construtores e os destrutores, a *Vazia*, bem como as operac¸o˜es de inserc¸a˜o e remoc¸a˜o nas va´rias posic¸o˜es da lista sa˜o adaptac¸o˜es dos algo- ritmos vistos em sala. Teve-se, no caso, uma atenc¸a˜o um pouco maior, devido a` existeˆncia tambe´m do ponteiro *anterior* em cada ce´lula da lista.

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



**Figure 1. Diagrama da Lista**

que e´ um ponteiro para elemento da lista. Este possibilita o acesso e tambe´m a

modificac¸a˜o de elementos da lista. Essa funcionalidade utiliza dois me´todos: *primeiro* que coloca o iterador na primeira posic¸a˜o e retorna um ponteiro para o objeto nessa posic¸a˜o (naturalmente imediatamente apo´s a ce´lula cabec¸a); e *proximo*, que ”anda” com o iterador uma posic¸a˜o e retorna o ponteiro para o elemento dessa nova posic¸a˜o. Se at- ing´ıssemos o fim da lista, o iterador se tornava *nulltpr*.

## Classes

Para modularizar a implementac¸a˜o, foram montadas treˆs classes principais e uma struct. A primeira delas e´ a lista duplamente encadeada abordada acima, cujas ce´lulas sa˜o structs, com dois ponteiros, um para a pro´xima ce´lula, outro para a anterior e um ponteiro para o elemento (um objeto) que esta´ sendo guardado nessa ce´lula e, consequentemente na lista.

A lista possui alguns me´todos u´teis, como o *get indice* para receber o ´ındice de um elemento e outros me´todos espec´ıficos como o Adiciona, pertencente a` Lista de Cursos, que faz o in´ıcio do processo de classificac¸a˜o.

As outras duas classes sa˜o Candidato e Curso. Elas foram criadas para ar- mazenarem os dados das entradas e processa´-los. A classe Candidato basicamente guar- dava as informac¸o˜es fornecidas de cada aluno, tal como nome, nota e opc¸o˜es 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 crite´rios de classificac¸a˜o e desempate. Ale´m disso, esta ainda tinha atributos como o nome do curso, a nota de corte e a quantidade de vagas dispon´ıveis.

## Classificac¸a˜o e Desempate

O sistema de classificac¸a˜o e desempate segundo as regras do SISU e´ um dos trechos mais importantes da implementac¸a˜o, sobretudo pois ele e´ quem maneja os candidatos que va˜o sendo adicionados.

Em prol da modularizac¸a˜o, preferiu-se separar obrigac¸o˜es. O Curso ficaria re- sponsa´vel por decidir se o candidato iria ser classificado ou se iria para a lista de es- pera daquele curso. Se ele fosse para a lista de espera, quem teria que controlar as con- sequeˆncias disso era a Lista de Cursos, que e´ criada quando o programa inicia, requisi- tando a entrada do candidato em sua segunda opc¸a˜o, por exemplo.

Essa ”comunicac¸a˜o” entre esses dois mo´dulos e´ feita atrave´s de um valor de re- torno, definida por um inteiro, que a classe Curso retornava para a Lista de Cursos. Como

comentado no co´digo, um retorno 0 significa que o candidato tinha sido colocado na lista de classificados e, assim, a princ´ıpio, na˜o 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 opc¸a˜o, a princ´ıpio. Por fim, um retorno 2 significa que o candidato foi direto para a lista de espera daquele curso, ou seja, devera´, a priori, ser colocado na sua segunda opc¸a˜o tambe´m.

A Lista de Cursos recebe essa comunicac¸a˜o e toma as devidas provideˆncias. Se o retorno foi zero, nada deve ser feito e seguimos para o pro´ximo candidato. Se o retorno for 1, ele deve adicionar o candidato que foi ”empurrado” na sua segunda opc¸a˜o de curso, mas se aquela ja´ for a segunda opc¸a˜o, nada deve ser feito. Se o retorno for 2, ele apenas

deve adicionar o aluno na sua segunda opc¸a˜o, se aquela ja´ na˜o for. Esse processo de

adic¸a˜o a um novo curso tambe´m tem o mesmo estilo de retorno, que leva novamente a uma ”ana´lise do empurrado” caso haja alguma alterac¸a˜o nesse sentido.

# Ana´lise Complexidade

## Tempo

Comec¸aremos a ana´lise para a simples adic¸a˜o de um candidato a um curso, enta˜o partire- mos para a ana´lise tambe´m se ele tiver que ser realocado em outras listas de espera.

Um curso estando vazio, o custo para adicionar um novo candidato e´ o curto de adicionar um novo elemento no in´ıcio da lista de classificados, no caso *O*(1), um custo

constante. Se curso ja´ tiver alguns alunos na lista de classificados, a situac¸a˜o e´ mais

complexa. Primeiro, ha´ uma operac¸a˜o com custo, na pior hipo´tese, *O*(*n*), uma pesquisa sequencial pela lista procurando um candidato com nota menor ou igual a` do novo aluno. Se o novo tiver nota menor, adicionamos a` u´ltima posic¸a˜o da lista de classificados com custo *O*(1).

Se o novo candidato tiver uma nota empatada com outro que ja´ esta´ na lista, tere- mos que iterar por ela (para garantir que na˜o ha´ outros empatados com essa mesma nota), com custo *O*(*n*) no pior caso e decidir o desempate. No desempate: custos constantes para comparac¸a˜o dos ´ındices das opc¸o˜es de cursos. Mas no final das contas, teremos que adicionar o novo candidato ou antes ou depois de algum candidato da lista numa posic¸a˜o espec´ıfica *i*. Isso tem o custo *O*(*n*) para encontrar o ´ındice e mais um *O*(*n*), para adicionar na i-e´sima posic¸a˜o, ambos no pior caso, considerando a implementac¸a˜o 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 classificac¸a˜o.

Caso a adic¸a˜o de um novo candidato extrapole o nu´mero de vagas, teremos que remover o u´ltimo da lista dos classificados, a custo constante, pois a lista e´ duplamente encadeada (um dos motivos pela escolha desta ao inve´s da simplesmente) e adiciona´-lo na lista de espera, que teˆm o mesmo custo acima explanado, pois a ideia geral e´ a mesma. Ou seja, o custo para adicionar um candidato num curso tem um custo linear com o tamanho da lista.

Agora temos tambe´m que analisar o que acontece quando temos que colocar um candidato que foi ”empurrado” para a lista de espera em uma das suas opc¸o˜es 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 opc¸a˜o do novo aluno. Em seguida, um custo linear *O*(*n*) com o nu´mero de candidatos que esta˜o nas listas de espera ou de classificados daquele curso. Se foi classificado direto, na˜o ha´ mais custo, seguimos para o pro´ximo candidato. Se o novo foi colocado na lista de espera, temos mais um *O*(*n*) para coloca´-lo na sua segunda opc¸a˜o. Enta˜o ate´ o momento, o custo para adicionar um candidato, entre N cursos e n candidatos ja´ nas listas de classificac¸a˜o e de espera

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

Se o novo foi classificado, mas ”empurrou” algue´m, temos uma situac¸a˜o mais complicada, temos que analisar a situac¸a˜o do ”empurrado” com o me´todo *AnaliseEm- purrado*. Qualquer que seja essa situac¸a˜o, teremos um custo de *O*(*N* ) para pesquisar a segunda opc¸a˜o dele e mais um *O*(*n*) para adiciona´-lo a esse curso, ou simplesmente arru- mar a lista de espera do curso que ele foi empurrado. O problema e´ quando o ”empurrado” empurra outro candidato do segundo curso, o que ira´ criar uma chamada recursiva para o me´todo *AnaliseEmpurrado*. Isso so´ termina, num pior caso em que todos os empurrados se classificam para suas segundas opc¸o˜es e empurram outro candidato, quando todos os cursos na˜o tem mais vagas. Na pior das hipo´teses teremos que chamar essa func¸a˜o para cada um dos *n* candidatos que esta˜o nas listas de classificados. Em outras palavras, veˆ-se que a func¸a˜o e´ 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 e´ da ordem de *n*2, sendo *n* o nu´mero de candidatos nas listas.

## Espac¸o

Vamos considerar que cada candidato ocupe uma unidade de espac¸o, para nossa ana´lise. Assim, na pior hipo´tese do programa, todos os candidatos esta˜o em dois locais: na lista de espera de seu primeiro curso e na lista de classificados ou de espera da segunda opc¸a˜o. Com isso, numa hipo´tese extrema, para uma entrada de *n* candidatos, o programa ocupa o espac¸o de 2*n* candidatos, ou enta˜o, *O*(2*n*) = *O*(*n*)

# Conclusa˜o

Apo´s a implementac¸a˜o 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 a` parte seguinte. Essa segunda parte tinha como objetivo utilizar a estrutra de dados criada anteriormente para criar os algoritmos que efetivamente fariam a selec¸a˜o e a classificac¸a˜o do ”Mini SISU”. Havia ainda uma pequena e mais simples implementac¸a˜o que compreendia a leitura da entrada e escrita da sa´ıda dos dados.

O grande esforc¸o, sobretudo para o desenvolvimento da segunda parte, era de considerar os va´rios casos que poderiam ocorrer e fazer isso tendo em vista o modo como seriam transformados os crite´rios de desempate em algoritmo. Ale´m disso, tinha-se que

estar atento algumas das boas pra´ticas de programac¸a˜o como modularizac¸a˜o de trechos

do co´digo e de encapsulamento, a fim de destinar a` devida classe, no caso Curso ou

ListaEncadeada, a referida responsabilidade. Assim, momentos de refatorac¸a˜o, sobretudo do trecho do co´digo que realizava essas ac¸o˜es, foram bastante necessa´rios para deixar o algoritmo mais leg´ıvel, com menos repetic¸o˜es e com me´todos menores.

Ao fim desse trabalho, ficou ainda mais evidente a importaˆncia dos testes de unidade e da refatorac¸a˜o. Ambas pra´ticas possibilitaram um desenvolvimento mais pro- dutivo e consciente de que, com a implementac¸a˜o de novas funcionalidades ou com a refatorac¸a˜o de trechos do co´digo, as estruturas do programa continuavam corretas e fun- cionais. Ale´m disso, pode-se protagonizar dois pape´is 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 Implementac¸o˜es em Java e C++: Cap´ıtulo 3: Estruturas de Dados Ba´sicas*. Editora Cengage.