Trabalho Prático 1: Acesso ao Ensino Superior em Arendelle

Marcelo Medeiros de Lima
marcelolima@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Resumo. *Este trabalho aborda um problema de seleção e ordenação. Se trata de uma versão simplificada do Sistema de Seleção unificado (SISU), responsável pelo processo seletivo de entrada das universidades públicas brasileiras. O objetivo é simular um processo de seleção seguindo os fundamentos do SISU e aplicando os conceitos lecionados na disciplina de Estruturas de Dados pela Universidade Federal de Minas Gerais para os alunos de Sistemas de Informação.*

1. Introdução

O Sistema de Seleção Unificado (SISU) foi criado pelo Ministério da Educação do Brasil em 2010, com o objetivo de unificar o processo seletivo de entrada para as universidades públicas brasileiras. Aberto a participantes do Exame Nacional do Ensino Médio (ENEM) do ano anterior, o SISU recebe milhões de inscrições todo ano, para milhares de vagas em instituições de ensino de nível superior distribuídas pelo país.

Elsa, rainha de Arendelle, ficou maravilhada ao saber da existência do SISU, e gostaria de implementar um sistema parecido em seu reino. A infraestrutura de universidades p´ublicas em Arendelle é muito parecida com a do Brasil, porém o sistema lá ainda é ligeiramente medieval, as inscrições dos alunos às universidades são feitas via pombo-correio, e este método tem se provado extremamente ineficiente. Elsa foi alertada ao fato de que os alunos de Estruturas de Dados da UFMG são programadores extremamente habilidosos, e portanto, resolveu pedir a sua ajuda para modernizar o sistema.

2. Solução proposta

Este é um problema de ordenação e seleção. Grande parte das regras de negócio se baseiam nesses dois princípios. A linguagem usada nessa solução proposta foi o C99. Os cursos e o alunos registrados no SISU são armazenados em vetores. Lista de aprovados e de espera de cada curso são listas duplamente encadeadas completamente independentes.

Após registrados, o algoritmo percore o array de alunos, inserindo-os nos respectivos cursos, começando na primeira opção e caso necessário seguindo com a regra de negócio para a segunda opção.

2.1. Estruturas de dados

As estruturas de dados utilizadas para solução do problema foram:
1. Course
2. Student
3. Node

2.1.1. Course:

Um Curso possui nome, número de vagas disponíveis na lista de aprovado, uma lista de aprovados e uma lista de espera.

```
typedef struct course Course;
struct course {
  char *name;
  int numberOfVacancies;
  Node *vacancies;
  Node *waitingList;
};
```

2.1.2. Student:

Um aluno possui nome, nota no SISU, primeira opção de curso e segunda opção de curso

```
typedef struct student Student;
struct student {
  char *name;
  float score;
  int firstOption;
  int secondOption;
};
```

2.1.3. Node:

Um nó possui um estudante, um próximo e um anterior. Reforçando que essa é uma lista duplamente encadeada.

```
typedef struct node Node;
struct node {
  Student *student;
  Node *next;
  Node *previous;
};
```

2.2. Algoritmos

O código possui partes iterativas e partes recursivas. Grande parte das regras de negócio giram em torno da **void manageCourseVacancies(Course *courses, Node *last, Node *node, Student *student, int size, int option)**. Essa função é chamada em caso de listas não nulas que podem estar cheias ou não. Quase toda a regra de negócio é feita neste gerenciador. A grande ideia dessa implementação foi criar funções capazes de trabalhar em diferentes contextos sem precisar de explicitar isso aos métodos com mais código ou mais regras de negócio. Em caso de mais dúvidas, favor consultar o código. Os métodos estão enxutos e bastante autoexplicativos. Devido ao primeiro for do código de processamento do sisu:

```
  for (i = 0; i < numberOfStudents; i++) {
    firstOption = students[i].firstOption;
    secondOption = students[i].secondOption;
```

```
4
5
6       Node *node = courses[firstOption].vacancies;
7       Node *last;
8       int size = getListSize(node, &last);
9
10      if (checkListEmpty(size)) {
11        insertAtBeginning(
12        &courses[firstOption].vacancies,
                                        &students[i]);
13      } else {
14        manageCourseVacancies(
15        courses,
16        last,
17        node,
18        &students[i],
19        size,
20        firstOption);
21      }
22
23      rewindList(&courses[firstOption]);
24      rewindList(&courses[secondOption]);
25    }
26
```

E a chamada recursiva de **manageCourseVacancies()** o algoritmo possui ordem de complexidade de

```
O(n)^2
```

O código foi separado em main.c sisu.c course.c linkedList.c student.c utils.c.

3. Compilação e execução

Para compilar o projeto, existe um **makefile** na pasta **cmake-build-debug** que está na raiz do projeto do CLion (IDE usada no projeto). Para executá-lo, basta digitar **make** dentro desta pasta. Para executar o projeto basta digitar

```
./TP1_Estrutura_de_Dados
```

dentro da mesma pasta para rodar o executável. Caso não consiga executar, o comando

```
sudo chmod +x TP1_Estrutura_de_Dados
```

deve resolver o problema.

4. Conclusão

O trabalho foi concluído com sucesso, sendo aprovado em todos os casos de teste fornecidos previamente. A grande dificuldade pessoal na execução do trabalho prático foi a codificação da regra de negócio. Como foi dito anteriormente, grande parte do código é altamente reutilizado e muito modularizado. Isso agrega no fator dificuldade da entrega. Outro fator relevante foram as passagens de argumento entre funções. Todos os argumentos estão sendo usados, necessariamente. Além disso passagens por ”referência”, que não existem em C, são evitadas a fim de diminuir o número de funções impuras no código.