SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 4

Alunos	**NUSP**
Juan Henriques Passos	15464826
Alec Campos Aoki	15436800

Trabalho prático 4 - Notáveis

Notáveis

Comentário

Dado um arquivo com quantidade de linhas menos uma de alunos, sendo que cada linha contém o nome do aluno, *nota1*, *nota2* e *nota3*, deve-se comparar o desempenho dos alunos, com base na subtração da *nota3* pela *nota1*, gerando uma classificação dos estudantes. Dessa forma, será fornecido um *k*, que representa as *k* melhores notas, sendo necessário imprimir os donos dessas *k* primeiras melhoras notas (observa-se que se pode ter mais de um aluno com uma determinada nota que esteja entre as *k* melhores, e assim, deve-se imprimir o nome de todos). Para resolver esse problema, criou-se uma lista para guardar os estudantes, que inicialmente é inicializada com os primeiros *k* estudantes, sendo ordenada de forma decrescente, assim para sabermos se devemos adicionar um próximo estudante na lista, basta ver se a sua nota é maior ou igual ao último (aluno com a menor nota entre os *k*). Caso seja, deve-se percorrer a lista, até achar sua respectiva posição, porém se essa nota já estiver na lista, ou seja, há mais alunos com esse aumento, adiciona-se esse aluno de forma paralela ao que guarda todos os elementos com essa determinada nota. Cabe ressaltar que caso o elemento não tenha nenhuma nota igual e aumente o tamanho da lista, limita-se o tamanho da lista a *k*, apagando o último, mantendo o tamanho igual a *k*. Tal procedimento foi adotado para evitar o gasto desnecessário de memória, reservando espaço apenas para as *k* melhores notas, conforme especificado no projeto. Portanto, cabe-se analisar a complexidade do algoritmo, tendo em vista que são *n* alunos, será feito a leitura dessas *n* alunos do arquivo, e será inserido os alunos na lista de acordo com a nota, e removendo-os se ultrapassar o tamanho *k* definido, e por últimos impressos os melhores alunos com as *k* melhores notas e enfim apagando-os. Um dos piores casos, ocorre quando todos os alunos têm nota igual e estão em ordem lexicográfica decrescente, assim sempre será necessário passar por todos os alunos para inserir, tanto na busca pela nota, quanto na ordenação lexicográfica que estará de forma paralela. Dessa forma, terá que inserir *n* alunos, e cada inserção terá custo *n*. Assim, a complexidade geral do problema é $O(n^2)$. O melhor dos casos ocorre quando os *k* primeiros alunos lidos são os melhores e ninguém tem nota melhor ou igual que eles, assim será feita *n* leituras, *k* remoções (no final) e *k* para imprimir, sendo assim, o melhor caso é $\Omega(k)$.

Código

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define ERRO -1;

typedef struct no_ NO;
typedef struct lista_ LISTA;
typedef struct aluno_ ALUNO;

struct no_{
   ALUNO *aluno;
   NO *noSeguinte;
   NO *noParalelo;
};

struct lista_{
   NO *inicio;
   NO *fim;
   int tamanho;
   int k;
};

struct aluno_{
   char nome[52];
   int aumento;
};

bool lista_inserir(LISTA *lista, ALUNO *aluno);
void lista_imprimir(LISTA *lista);
void lista_remover(LISTA *lista);
void lista_atualizarTamanho(LISTA *lista);

int main(void){
   char nome_arq[100];
```

```c
/*Criando a lista (encadeada e ordenada)*/
LISTA *lista = (LISTA *)malloc(sizeof(lista));
if(lista == NULL) return ERRO;
lista->inicio = NULL;
lista->fim = NULL;
lista->tamanho = 0;

scanf("%s", nome_arq);
scanf("%d", &lista->k);

FILE *fp = fopen(nome_arq, "r+");
if(fp == NULL) return ERRO;

char string[52];
fscanf(fp, "%s\n", string);

/*Lendo o primeiro aluno do arquivo*/
ALUNO *alunoAux = (ALUNO *)malloc(sizeof(ALUNO));
if(alunoAux == NULL) return ERRO;
float n1, n3;
int aumento;
fscanf(fp, "%[^,],%f,%f,%f\n", alunoAux->nome, &n1, &n3, &n3);
alunoAux->nome[strlen(alunoAux->nome)] = '\0';
alunoAux->aumento = (int)((n3-n1)*10);

/*Inicializando a lista com esse aluno -> evitamos checar se estamos
no caso em que a lista é vazia dentro do while e das funções*/
NO *noNovo = (NO *)malloc(sizeof(NO));
if(noNovo == NULL) return ERRO;
noNovo->aluno = alunoAux;
noNovo->noSeguinte = NULL;
noNovo->noParalelo = NULL;
lista->inicio = noNovo;
lista->fim = noNovo;
lista->tamanho++;
```

```
   while(!feof(fp)){
       fscanf(fp, "%[^,],%f,%f,%f\n", string, &n1, &n3, &n3);
       string[strlen(string)] = '\0';
       aumento = (int)((n3-n1)*10);

          /*Note que somente alocamos espaço na memória se vamos inserir o aluno na nossa
lista*/
       if((aumento >= lista->fim->aluno->aumento) || (lista->tamanho <= lista->k)){
           ALUNO *alunoAux = (ALUNO *)malloc(sizeof(ALUNO));
           if(alunoAux == NULL) return ERRO;
           strcpy(alunoAux->nome, string);
           alunoAux->nome[strlen(alunoAux->nome)] = '\0';
           alunoAux->aumento = aumento;

           lista_inserir(lista, alunoAux);
       }
   }

   fclose(fp);

   /*Garantindo que a lista tem tamanho k*/
   while(lista->tamanho > lista->k){
       lista_remover(lista);
   }

   lista_imprimir(lista);

   /*Esvazaziando a lista (desalocando a memória)*/
   while(lista->tamanho != 0){
       lista_remover;
   }

   return 0;
}

bool lista_inserir(LISTA *lista, ALUNO *aluno){
```

```
   if(lista == NULL) return false;

   NO *noNovo = (NO *)malloc(sizeof(NO));
   if(noNovo == NULL) return false;
   noNovo->aluno = aluno;
   noNovo->noSeguinte = NULL;
   noNovo->noParalelo = NULL;

   /*
   Preenchemos a lista com os primeiros k alunos do arquivo, independente de seus aumentos
   Para os próximos alunos, verificamos se seu aumento é maior que o último aluno da lista;
       se for, o adicionamos na lista.
       senão, o ignoramos.
   Nos casos em que o aumento de um aluno é igual ao outro, ele será guardado como um nó
   paralelo ao aluno com o mesmo aumento.
        Caso o aluno não seja o último, contamos seus nós paralelos como parte do tamanho da
lista.
       Caso o aluno seja o último, não consideramos seus nós paralelos como parte do tamanho
da lista.
    Toda vez que inserimos um aluno e o tamanho da lista ultrapassar k, apagamos o último
elemento dela
   (e seus nós paralelos, se houver).
   A lista é ordenada em ordem decrescente.

   Como estamos utilizando uma lista encadeada, vamos usar um ponteiro que aponta para o nó
da lista
   que está sendo comparado ao aluno que será inserido (pontAux) e outro ponteiro que aponta
para o nó
    anterior a esse (pontAux_noAnterior). Dessa forma, caso haja necessidade de realizar
qualquer operação
    com o nó anterior a aquele analisado, não precisamos percorrer a lista do começo para
achá-lo.
   */

   NO *pontAux = lista->inicio;
   NO *pontAux_noAnterior = lista->inicio;
```

```
for(int i=0; i<lista->tamanho; i++){
  if(pontAux->aluno->aumento == aluno->aumento){
      /*Inserindo paralelamente*/
      if(strcmp(noNovo->aluno->nome, pontAux->aluno->nome) < 0){
          /*Ordenação pela lexicografia
          Colocamos o aluno lido na sequência de nós principais e o nó do qual ele
          toma lugar se torna um nó paralelo
          */
          if(lista->fim == pontAux) lista->fim = noNovo;
          else if(lista->inicio == pontAux) lista->inicio = noNovo;

          noNovo->noSeguinte = pontAux->noSeguinte;
          noNovo->noParalelo = pontAux;
          pontAux_noAnterior->noSeguinte = noNovo;

          pontAux->noSeguinte = NULL;
      }
      else{
          /*Ainda temos que nos atentar à ordenação pela lexicografia*/
          do{
              pontAux_noAnterior = pontAux;
              pontAux = pontAux->noParalelo;

              if(pontAux == NULL){
                  pontAux_noAnterior->noParalelo = noNovo;
                  noNovo->noParalelo = NULL;

                  break;
              }
              else{
                  if(strcmp(noNovo->aluno->nome, pontAux->aluno->nome) < 0){
                      noNovo->noParalelo = pontAux;
                      pontAux_noAnterior->noParalelo = noNovo;

                      break;
                  }
              }
```

```
            }while(true);
        }

        lista_atualizarTamanho(lista);

        if(pontAux == lista->fim){
            /*Não contamos alunos com aumento igual ao do fim da lista no tamanho, então
              não precisamos remover nenhum aluno para mantermos o tamanho da lista menor
que k
            */
            return true;
        }
        else{
            if(lista->tamanho > lista->k){
                lista_remover(lista);
            }
        }

        return true;
    }

    else if(aluno->aumento > pontAux->aluno->aumento){
        /*Inserindo o aluno ordenadamente na lista encadeada*/
        if(pontAux == lista->inicio){
            noNovo->noSeguinte = lista->inicio;
            lista->inicio = noNovo;
        }
        else{
            noNovo->noSeguinte = pontAux;
            pontAux_noAnterior->noSeguinte = noNovo;
        }
        if(lista->tamanho > lista->k){
              /*Como acabamos de inserir um novo aluno, se o tamanho da lista ultrapassar
k,
            temos que remover o último aluno
            */
            lista_remover(lista);
```

```
            }

            lista_atualizarTamanho(lista);
            return true;
        }

        else{
            /*"Andamos" os ponteiros "para frente" na lista encadeada*/
            pontAux_noAnterior = pontAux;
            pontAux = pontAux->noSeguinte;
            if(pontAux == NULL) break;
        }
    }

    /*Inserindo no fim (tamanho < k)*/
    lista->fim->noSeguinte = noNovo;
    lista->fim = noNovo;
    lista->tamanho++;

    return true;
}

void lista_remover(LISTA *lista){
    if(lista == NULL) return;

    lista_atualizarTamanho(lista);

    NO *noAux = lista->fim;
    NO *noAnterior = lista->inicio;

    /*Movendo noAnterior para o nó anterior ao noAux*/
    while(noAnterior->noSeguinte != noAux){
        noAnterior = noAnterior->noSeguinte;
    }

    free(lista->fim->aluno);
    free(lista->fim);
```

```
    lista->fim = noAnterior;
    noAnterior->noSeguinte = NULL;

    lista_atualizarTamanho(lista);

    return;
}

void lista_atualizarTamanho(LISTA *lista){
    if(lista == NULL) return;

    NO *noAux = lista->inicio;
    NO *proximoNo = noAux;
    int tam=0;

    /*Vamos percorrer a lista, adicionando ao tamanho da lista todos os nós não-paralelos.
     Caso um nó (que não seja o último) tenha nós paralelos, vamos inserí-los no tamanho da lista também.
    */
    while(noAux != lista->fim){
        tam++;
        proximoNo = noAux;
        while(proximoNo->noParalelo != NULL){
            tam++;
            proximoNo = proximoNo->noParalelo;
        }
        noAux = noAux->noSeguinte;
    }
    tam++;

    lista->tamanho = tam;

    return;
}

void lista_imprimir(LISTA *lista){
    if(lista == NULL) return;
```

```
    NO* pontNo = lista->inicio;
    NO* pontParalelos;

    int i=0;

    while(pontNo != NULL){
        pontParalelos = pontNo->noParalelo;
        printf("%s\n", pontNo->aluno->nome);
        while(pontParalelos != NULL){
            printf("%s\n", pontParalelos->aluno->nome);
            pontParalelos = pontParalelos->noParalelo;
        }
        pontNo = pontNo->noSeguinte;
    }
}
```

Saída

```
casosteste/alunos_notas.csv 3
Locsprywhysnizkvhykuyxqc
Vfscctszedcdtqzciowrqwktgyrelgzzqbscq
Anmkfjxacqkfgemunshmowzukhqzbompzhmxjjiixcnqs
Edblkeshgl
Mycvqotroafwkhwjwmjmyrejrxj
Olmjlndxkaxpwlgajoebqrxlkyviwt
Pwfccaqmihhzevtosfvokleirrdpaihcuigqa
Teesciwciacrpjovtviwfvtnuibiragylvq
Tivkgajeoyzirlwkozwzqnflx
Tjrezivzvjxm
Toflddvedob
Uliklxm
Yyqdpemwjnkesdqasxyuajbrfpfjibdngizefuyfqhjb
Zfklmxdeokolfwyoyzifvp

Tempo de execucao: 0.057181ms
```

```
casosteste/alunos_notas.csv 12
Locsprywhysnizkvhykuyxqc
Vfscctszedcdtqzciowrqwktgyrelgzzqbscq
Anmkfjxacqkfgemunshmowzukhqzbompzhmxjjiixcnqs
Edblkeshgl
Mycvqotroafwkhwjwmjmyrejrxj
Olmjlndxkaxpwlgajoebqrxlkyviwt
Pwfccaqmihhzevtosfvokleirrdpaihcuigqa
Teesciwciacrpjovtviwfvtnuibiragylvq
Tivkgajeoyzirlwkozwzqnflx
Tjrezivzvjxm
Toflddvedob
Uliklxm
Yyqdpemwjnkesdqasxyuajbrfpfjibdngizefuyfqhjb
Zfklmxdeokolfwyoyzifvp

Tempo de execucao: 0.059040ms
```