SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

TRABALHO PRÁTICO - AEDS I

SISTEMA DE CONTROLE DE SONDAS E CATALOGAÇÃO DE ROCHAS MINERAIS

MARIA EDUARDA DUARTE LACERDA[5920]
ANA CLARA SOARES OLIVEIRA [5896]
LARA GEOVANA ALMEIDA MELO [5897]

Florestal - MG
2024

SUMÁRIO

1. INTRODUÇÃO

No trabalho prático da disciplina Algoritmos e Estruturas de Dados I, o objetivo era desenvolver o Sistema de Controle e Catalogação de Rochas, que tinha por intuito investigar a composição do solo marciano, por meio da coleta e armazenamento de diferentes tipos de rochas e as informações referentes a elas.

Para a implementação do sistema foi utilizado os conceitos de Tipo Abstrato de Dados(TAD), TAD Lista Linear implementada com vetor e TAD Lista Linear Encadeada.

2. ORGANIZAÇÃO

A organização do trabalho foi feita a partir da criação de um repositório no GitHub[1], onde os três integrantes tiveram acesso para fazer as modificações necessárias.

Na Figura 1 é possível visualizar como foi realizada a organização do projeto. O repositório foi nomeado como **TRABALHOPRÁTICO**[2] e possui a seguinte estrutura.

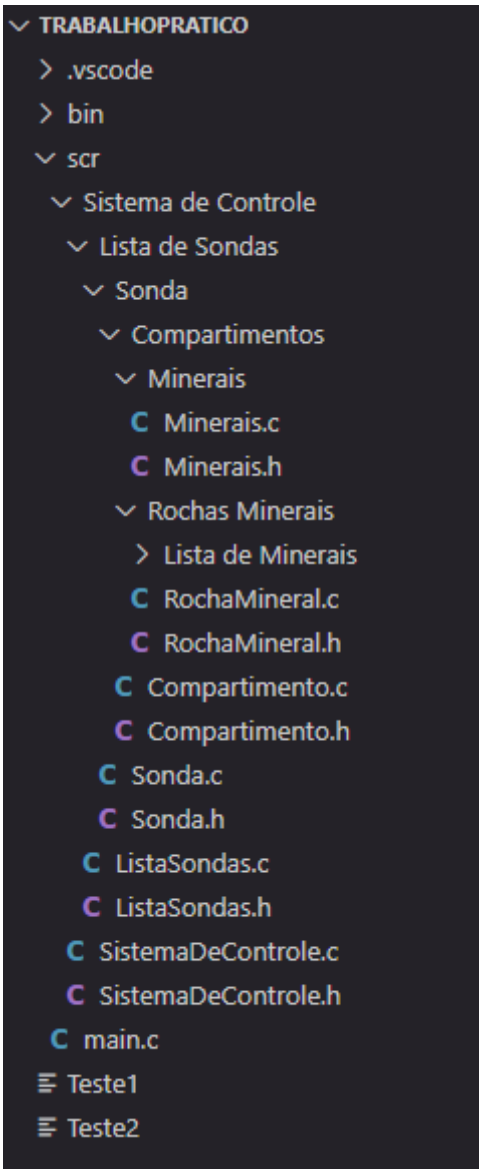


Figura 1 - Repositório do projeto.

Na pasta também estão localizados os dois arquivos(Teste1,Teste2) que utilizamos para testar nosso projeto. Para executar o projeto, foi utilizado o

task.json com os comandos necessários para compilar e executar os códigos, que está localizado na pasta .vscode. Na pasta bin está localizado o arquivo executável. E por último na pasta src(Source) estão os arquivos .c e .h, organizados em um estrutura que mostra como foi implementado os Tipos Abstratos de Dados(TADs).

3. DESENVOLVIMENTO

No desenvolvimento do trabalho decidimos criar o Sistema de Controle separado do main, com o intuito de deixar os TADs encapsulados, logo as funções implementadas no sistema são responsáveis por executar todas as funcionalidades básicas que o projeto deve possuir. Sendo elas, a entrada de dados(arquivo ou terminal), inicialização, coleta de novas rochas, impressão do status atual das sondas e redistribuição de rochas.

Em relação a como será feita a entrada de dados, ou seja, se será por terminal ou arquivo, implementamos isso na função Central, que chama a função de entrada de acordo com a resposta do usuário.

```c
void Central(TSondas *ListaSondas){

    printf("Bem-vindo(a) ao Sistema de Controle e Catalogacao de Rochas Minerais!\n");
    printf("Como voce deseja realizar a entrada de dados?\n");
    printf("(1)Arquivo.\n(2)Terminal.\n");
    int escolha;
    scanf("%d", &escolha);
    switch (escolha){
        case 1:
            LeituraPorArquivo(ListaSondas);
            break;
        case 2:
            LeituraPeloTerminal(ListaSondas);
            break;
        default:
            break;
    }
}
```

Figura 2 - Função Central

Após isso ocorre a implementação da entrada de dados, inicialização e coleta de rochas por meio de três funções: LeituraPorArquivo e LeituraPeloTerminal. E as funcionalidades de Coleta, Impressão e Redistribuição foram implementadas em diferentes funções do Sistema.

3.1 ENTRADA DE DADOS POR ARQUIVO

A leitura de dados por arquivo foi realizada a partir da função LeituraPorArquivo, os pontos principais desta operação foi a utilização de comandos que envolvem a abertura e leitura de arquivos. Os mais importantes para o sucesso do processo foram o fopen(abre o arquivo), fscanf(lê semelhantemente ao scanf), fgets(lê uma linha até o \n). Além desses, utilizamos também o comando strtok, que separa uma string em outras strings menores, ele foi essencial para separarmos os dados das rochas.

```
FILE* entrada;
entrada = fopen(nome, "r");
if(entrada==NULL){
    printf("Erro ao ler o arquivo.");
    return 0;
}

int numsondas;
double lat_i, long_i, c_i;
int v_i, vc_i, identificador;

fscanf(entrada,"%d", &numsondas);
FazListaVazia(ListaSondas);
for(int m = 0; m<numsondas;m++){
    Sonda NovaSonda;
    identificador = m+1;
    fscanf(entrada,"%lf %lf %lf %d %d", &lat_i, &long_i, &c_i, &v_i, &vc_i);
    InicializaSonda(&NovaSonda, lat_i, long_i, c_i, v_i, vc_i, identificador);
    InsereSonda(ListaSondas, &NovaSonda);
}
```

Figura 3 - Utilização do fopen e fscanf

A leitura dos dados da sonda foram feitas utilizando apenas o comando fscanf, um para pegar o número de sondas a ser lido na primeira linha, e outro para a partir de uma estrutura de repetição, ler as informações presentes no número de linhas que foi lido anteriormente, inicializando a sonda a cada iteração.

Após isso, é inicializada a variável que guarda o número de operações, podendo ser a operação de inicialização(R), impressão(I) e redistribuição de rochas(E), e então utilizando este número e uma estrutura repetição, para que as operações especificadas no arquivo sejam executadas.

Em relação a inicialização das rochas, após a leitura dos dados e inserção dos minerais na Lista, para escolher qual sonda rocha será armazenada, criamos a função Coleta.

3.2 ENTRADA DE DADOS PELO TERMINAL

A leitura dos dados a partir do terminal foi realizada por meio da função LeituraPeloTerminal, que recebe como parâmetro uma lista de sondas anteriormente criada.O grupo decidiu implementar dando como primeira opção de operação apenas a criação de sondas, para que seguisse um padrão semelhante à entrada por arquivo e para facilitar a armazenagem das rochas, de modo que não é possível criar uma rocha sem antes inicializar uma sonda para armazená-la.

```
int LeituraPeloTerminal(TSondas *ListaSondas){
    char data[Data];
    time_t mytime;
    mytime = time(NULL);
    if(numsondas==0){

        printf(" Bem-vindos a central de Inicializaçao de Sondas e Coleta de Rochas!\n"
        "O primeiro passo é criar as sondas necessárias para a analise do solo Marciano."
        "Esta operacao ira criar uma nova sonda de acordo com os atributos digitados.\n"
        "A sonda tambem sera ligada e disparada para o solo Marciano.\n"
        "Quantas sondas gostaria de iniciar? ");

        int sondas;
        scanf("%d", &sondas);       Ana Clara, 3 days ago • .
        numsondas += sondas;
        for(int s = 0; s < numsondas; s++){
            Sonda NovaSonda;

            double lat_i, long_i;
            int  c_i, v_i, nc_i, identificador = s+1;
            printf("Digite as informacoes da sonda %d:\n", s+1);
            getchar();
```

Figura 4 - Função LeituraPeloTerminal

Após a criação das sondas com os atributos digitados pelo usuário e a inserção das mesmas na lista de sondas, é dada ao usuário a opção de escolha da operação a ser executada a seguir, sendo elas a criação de rochas, redistribuição e impressão do estado atual das sondas.

```c
printf("Qual operacao deseja realizar?\n");
printf("(1)Inicializacao de rochas.\n"
"(2)Redistribuicao de rochas.\n"
"(3)Impressao do estado atual das sondas.\n");
int escolha;
scanf("%d", &escolha);
```
*

Figura 5 - Escolha de operação

A partir do número que for digitado pelo usuário, uma operação irá ser iniciada. A operação de inicialização de rochas também foi implementada seguindo o padrão da leitura por arquivo, o usuário será questionado sobre a quantidade de rochas que ele quer criar e após isso irá digitar as informações de cada uma.

```c
if(escolha == 1){
        ListaMinerais ListaMineirais;

        IniVListaM(&ListaMineirais);

        printf("Esta operacao ira coletar a rocha de acordo com os atributos digitados,\n"
        "Alem de adiciona-la na sonda mais proxima.\n"
        "Quantas rochas gostaria de coletar? ");
        int numrochas;
        scanf("%d", &numrochas);
        for(int r = 0; r <numrochas; r++){
            double latrocha, longrocha, pesorocha;


            printf("Digite as informaces da %dª rocha:\n", r+1);

            printf("Latitude:");
            scanf("%lf", &latrocha);
            getchar();

            printf("Longitude:");
            scanf("%lf", &longrocha);
            getchar();

            printf("Peso da rocha:");
            scanf("%lf", &pesorocha);
            getchar();
```

Figura 6 - Coleta de novas rochas

A partir disso, a função Coleta foi chamada para identificar qual sonda é a melhor opção para guardar cada rocha que foi inicializada.Ao final, é dado a opção do usuário realizar outras funções, voltando para o início da função.

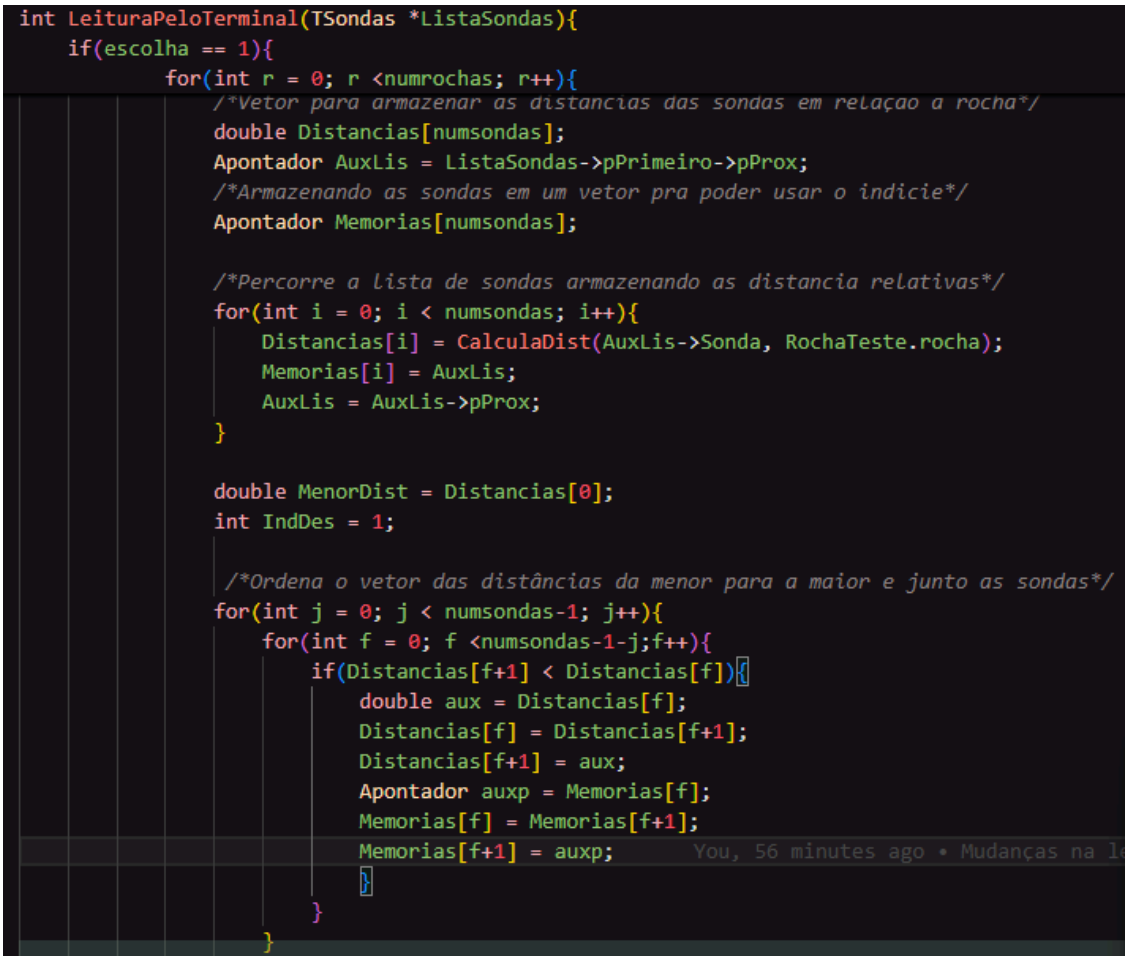
3.3 COLETA

O intuito da função Coleta foi encontrar a sonda ideal para armazenar cada rocha coletada pelas funções “EntradaPeloTerminal” ou “EntradaPorArquivo”. Isso foi feito a partir da lógica explicada a seguir:

A condição para a escolha de qual sonda a rocha seria armazenada foi feita com base na distância da rocha para cada sonda, ou seja, a sonda com menor distância e com espaço disponível será a escolhida.

Para conseguirmos identificar qual a sonda com menor distância, utilizamos a função criada no TAD Sonda, que calcula a distância até a rocha passada como parâmetro. Além disso, para guardar cada distância euclidiana foi utilizado um vetor auxiliar(Distâncias) e para identificar a qual sonda aquela distância pertence foi criado um vetor auxiliar com o endereço de memória de cada sonda(Memórias), de modo que o índice do vetor de distâncias corresponde ao índice do vetor de sondas.

. Após isso, o vetor que armazena as distâncias foi ordenado de forma crescente e o vetor Memórias foi organizado da mesma forma, para que cada Distância[i] corresponda a Memorias[i].



```
int LeituraPeloTerminal(TSondas *ListaSondas){
    if(escolha == 1){
            for(int r = 0; r <numrochas; r++){
                /*Vetor para armazenar as distancias das sondas em relaçao a rocha*/
                double Distancias[numsondas];
                Apontador AuxLis = ListaSondas->pPrimeiro->pProx;
                /*Armazenando as sondas em um vetor pra poder usar o indicie*/
                Apontador Memorias[numsondas];

                /*Percorre a lista de sondas armazenando as distancia relativas*/
                for(int i = 0; i < numsondas; i++){
                    Distancias[i] = CalculaDist(AuxLis->Sonda, RochaTeste.rocha);
                    Memorias[i] = AuxLis;
                    AuxLis = AuxLis->pProx;
                }

                double MenorDist = Distancias[0];
                int IndDes = 1;

                 /*Ordena o vetor das distâncias da menor para a maior e junto as sondas*/
                for(int j = 0; j < numsondas-1; j++){
                    for(int f = 0; f <numsondas-1-j;f++){
                        if(Distancias[f+1] < Distancias[f]){
                            double aux = Distancias[f];
                            Distancias[f] = Distancias[f+1];
                            Distancias[f+1] = aux;
                            Apontador auxp = Memorias[f];
                            Memorias[f] = Memorias[f+1];
                            Memorias[f+1] = auxp;      You, 56 minutes ago • Mudanças na l
                            }
                        }
                    }
```

Figura 7 - Código para encontrar a sonda com menor distância da rocha

No entanto, apenas isso não é suficiente para conseguir armazenar todas as rochas corretamente pois pode ocorrer uma situação onde a sonda com menor distância ultrapassa o peso máximo do compartimento ao armazenar a nova rocha.Então foi necessário implementar após isso a verificação do peso máximo suportado pela sonda com menor distância, e se caso após a inserção da rocha está capacidade fosse ultrapassada, a rocha seria armazenada na sonda com a segunda menor distância, e assim por diante.

```
Apontador MemoriaQueQueremos;

MemoriaQueQueremos = Memorias[0];
/*Insere a rocha no compartimento da sonda com menor distancia*/
for(int d = 0; d < numsondas; d++){
    if(PesoAtual(&MemoriaQueQueremos->Sonda.CompartmentoS) + RochaTeste.rocha.Peso <=
    MemoriaQueQueremos->Sonda.CompartmentoS.PesoMax){
    MoveSonda(&MemoriaQueQueremos->Sonda, RochaTeste.rocha.Latitude, RochaTeste.rocha.Longitude);
    InsereRocha(&MemoriaQueQueremos->Sonda.CompartmentoS, &RochaTeste, MemoriaQueQueremos->Sonda.CompartmentoS.PesoMax);
    break;
}
else {
    MemoriaQueQueremos = Memorias[d+1];
    }       You, 1 second ago • Uncommitted changes
}
EsvaziaLista(&ListaMineirais);
```

Figura 8 - Inserção da rocha na sonda

Após isso, a Lista de Minerais é esvaziada e o processo é repetido até que o usuário crie o número de rochas que foi digitado anteriormente.

```
printf("Deseja realizar outra operacao?(s/n)");
char res1;
scanf("%c", &res1);
if(res1 == 's'){
    LeituraPeloTerminal(ListaSondas);
}
else{
    return 0;
    }
```

Figura 9 - Usuário escolhe se irá realizar outra operações

3.4 IMPRESSÃO

A operação de impressão foi feita utilizando a função já criada no TAD Lista de Sondas, está função percorre a lista encadeada por meio de um while, verificando se o compartimento está vazio por meio da função VerificaListaVazia, que está localizada no TAD compartimento, se ele estiver vazio imprime a frase “Compartimento Vazio!”, se não chama a função ImprimeLista, também localizada no TAD compartimento, e ela imprime as informações pedidas na especificação sendo elas, a categoria e o peso das rochas armazenadas em cada sonda.

```c
int ImprimeSonda(TSondas* Sondas){
    TCelula_S* pAux;
    pAux = Sondas->pPrimeiro->pProx;
    if (pAux == NULL) {
        return 0;
    }
    while (pAux != NULL) {
        printf("Identificador: %d\n", pAux->Sonda.Identificador);
        if(VerificaListaVazia(&pAux->Sonda.CompartmentoS)){
            printf("Compartimento Vazio!\n");
        } else{
            ImprimiLista(&pAux->Sonda.CompartmentoS);
        }
        pAux = pAux->pProx;
    }
    return 1;       Ana Clara, 3 days ago • .
}
```

Figura 10 - Função ImprimeSonda

```c
int VerificaListaVazia(Compartimento *lista){
    return(lista->primeiro==lista->ultimo);
}

void ImprimiLista(Compartimento *lista){
    Celula* pAux;
    pAux = lista->primeiro->pProx;
    while(pAux != NULL){
    printf("%s %.0lf\n", pAux->rocha.Categoria, pAux->rocha.Peso);
    pAux = pAux->pProx; /* próxima célula */
    }
}
```

Figura 11 - Função VerificaListaVazia e ImprimiLista

3.5 REDISTRIBUIÇÃO DAS ROCHAS

A função de redistribuição de rochas tinha como intuito deixar todas as sondas com aproximadamente a média do peso total delas em seu compartimento, para executar isso foi criada a função RedistribuiRochas no Sistema de Controle.

O primeiro passo foi mover as sondas para o ponto (0,0) e calcular a média do peso total armazenado nas sondas.Isto foi realizado por meio de uma estrutura de repetição percorrendo a lista e variáveis auxiliares para armazenar os valores.

```c
void RedistribuiRochas(TSondas *ListaSondas, int numsondas){
    double PesoTotal = 0;

    Apontador AuxiliarSondas = ListaSondas->pPrimeiro->pProx;

    while (AuxiliarSondas != NULL){
        MoveSonda(&AuxiliarSondas->Sonda,0,0); /*Move as sondas ao ponto (0,0)*/

        PesoTotal += PesoAtual(&AuxiliarSondas->Sonda.CompartmentoS); /*Calcula o Peso conjunto de todas as Sondas*/

        AuxiliarSondas = AuxiliarSondas->pProx; /*Percorre a Lista*/
    }

    double Med = PesoTotal/(double)numsondas;
```

Figura 12 - Cálculo da média do peso entre as sondas

Após isso, a abordagem escolhida foi a de encontrar quais sondas possuíam o seu peso total de armazenamento acima da média+5 e retirar rochas até que elas estivessem com o peso menor ou igual a esse valor. Entretanto, antes de retirar rochas das sondas sendo percorridas, verifica-se se existe alguma outra sonda que pode receber tais usando a função VerificaSeTemQuemReceber, também implementada no SistemadeControle.

Caso sim, as rochas retiradas são guardadas em um compartimento auxiliar criado anteriormente chamado “ComTemporario”

```c
  while(AuxiliarSondas != NULL){
      RochaEmQuestao = AuxiliarSondas->Sonda.CompartmentoS.primeiro->pProx;
      //Começa da primeira rocha de cada compartimento

      while(PesoAtual(&AuxiliarSondas->Sonda.CompartmentoS) > Med+5){
      //Itera quando o peso da sonda sendo percorrida é maior que a média + 5

          while(RochaEmQuestao->pProx != NULL){
          // Itera até chegar na penúltima rocha do compartimento

              if(VerificaSeTemQuemReceber(ListaSondas, &AuxiliarSondas->Sonda.CompartmentoS, RochaEmQuestao->rocha.Peso)){
                  //Vê se tem outra sonda pra ficar com a rocha sendo percorrida

                  Celula* RochaRemovida = RemoveRocha(&AuxiliarSondas->Sonda.CompartmentoS,
                  RochaEmQuestao);

                  InsereRocha(&ComTemporario,RochaRemovida,PESOMAXIMO);
              }

              if(PesoAtual(&AuxiliarSondas->Sonda.CompartmentoS) <= Med + 5){
                      break;
                  }
              RochaEmQuestao = RochaEmQuestao->pProx;
          }
      break;
      }

  AuxiliarSondas = AuxiliarSondas->pProx;
  }
```

Figura 13 - Verificando se existem sondas que podem receber as rochas da sonda sendo percorrida e, caso sim, retirando as rochas e colocando em um compartimento auxiliar

Após isso, é criado um vetor para armazenar o endereço de memória das sondas existentes, esse vetor é preenchido utilizando a função PreencheVetor. O intuito desse vetor é organizar as sondas de forma crescente, ou seja, a sonda com menor peso em seu compartimento estaria na primeira posição. A organização das sondas de acordo com seu peso foi feita na função OrdenaPesos.

```
void RedistribuiRochas(TSondas *ListaSondas, int numsondas){

    Sonda *VetorSondas[numsondas];

    PreencheVetor(ListaSondas, VetorSondas, numsondas);

    AuxiliarSondas = ListaSondas->pPrimeiro->pProx;

    OrdenaPesos(VetorSondas, numsondas);
```

Figura 14 - Vetor Sondas

Seguindo a lógica, de forma parecida com como escolhemos a sonda mais próxima para ser aquela que armazena uma rocha, percorremos novamente a lista de sondas. Primeiro, confirmando que o compartimento auxiliar não está vazio, pois caso esteja, não tem nenhuma rocha para ser realocada. Assim, o código procura a através do identificador “indm” (VetorSondas[0]->Identificador), qual sonda de menor peso atual, verificando se ela comporta a última rocha do compartimento auxiliar utilizando seu peso máximo (caso ela não comporte, indm é atualizado para a próxima sonda de menor peso atual usando a variável “ver”, que inicialmente é zero e a cada vez que a sonda de menor peso não comporta a última rocha, passa a ser ver+1”.

Caso todas essas condições sejam satisfeitas, é verificada a existência de uma rocha de mesma categoria na sonda em questão. De forma que, se existir, a rocha de maior peso é apenas descartada e a sonda fica com a mais leve. Se não, a última rocha do compartimento auxiliar é tirada dele e colocada ao final do compartimento da sendo percorrida.

Por último, para o caso da sonda sendo percorrida ser a última, ela não cumprir com as condições imposta e o compartimento auxiliar ainda não ter ficado completamente vazio, o auxiliar usado para percorrer a lista de sondas volta a ser a primeira e a lista é percorrida novamente, até o compartimento auxiliar ficar vazio.

```
while (AuxiliarSondas != NULL) {

    if(!VerificaListaVazia(lista: &ComTemporario)){

        if(PesoAtual(lista: &AuxiliarSondas->Sonda.CompartmentoS)+ComTemporario.ultimo->rocha.Peso
            <= AuxiliarSondas->Sonda.CompartmentoS.PesoMax &&
            AuxiliarSondas->Sonda.Identificador == VetorSondas[ver]->Identificador){

                if(VerificaRochaExistente(lista: &AuxiliarSondas->Sonda.CompartmentoS, &ComTemporario.ultimo->rocha)){

                    TrocaRocha(lista: &AuxiliarSondas->Sonda.CompartmentoS, &ComTemporario.ultimo->rocha);
                    Celula* RochaRemovida = RemoveRocha(lista: &ComTemporario, rocha: ComTemporario.ultimo);

                    PreencheVetor(ListaSondas, VetorSondas, numsondas);
                    OrdenaPesos(VetorSondas, numsondas);
                    ver = 0;

                    if(VerificaListaVazia(lista: &ComTemporario)){
                        break;
                    }

                }
                else {
                    Celula* RochaRemovida = RemoveRocha(lista: &ComTemporario, rocha: ComTemporario.ultimo);

                    InsereRocha(lista: &AuxiliarSondas->Sonda.CompartmentoS,
                    rocha: RochaRemovida, AuxiliarSondas->Sonda.CompartmentoS.PesoMax);

                    PreencheVetor(ListaSondas, VetorSondas, numsondas);
                    OrdenaPesos(VetorSondas, numsondas);
                    ver = 0;
                    if(VerificaListaVazia(lista: &ComTemporario)){
                        break;
                    }
                }

                if(!VerificaListaVazia(lista: &ComTemporario) && AuxiliarSondas->pProx == NULL){
                AuxiliarSondas = ListaSondas->pPrimeiro->pProx;
                }
            }
            else if(PesoAtual(lista: &AuxiliarSondas->Sonda.CompartmentoS)+ComTemporario.ultimo->rocha.Peso
                > AuxiliarSondas->Sonda.CompartmentoS.PesoMax &&
                 AuxiliarSondas->Sonda.Identificador == indm){

                ver++;

                if(!VerificaListaVazia(lista: &ComTemporario) && AuxiliarSondas->pProx == NULL){
                AuxiliarSondas = ListaSondas->pPrimeiro->pProx;
                }

            }
            else {
                AuxiliarSondas = AuxiliarSondas->pProx;
            }
        }
        else {
            break;
        }
    }
```

Figura 15 - Inserção das Rochas do compartimento auxiliar nas Sondas da Lista

3.6 TIPOS ABSTRATOS DE DADOS

Seguindo as especificações, os TADs foram implementados da seguinte forma:

O TAD Minerais cria um tipo de dado enum (enumeração: conjunto de valores constantes) chamado Cor e uma estrutura chamada Minerais que armazena os dados do mineral (tipos e nomes de suas características);

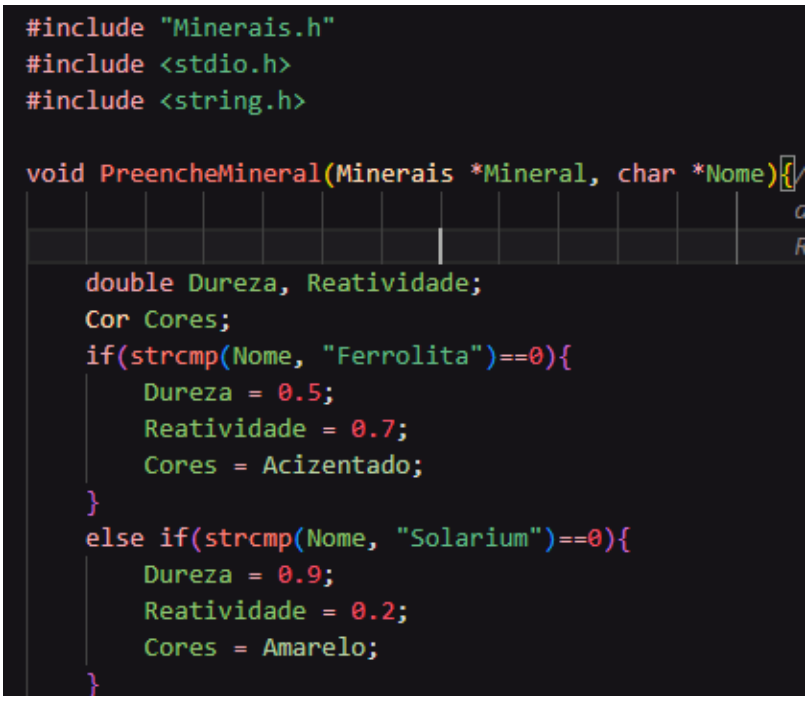
```c
#ifndef MINERAIS_H_
#define MINERAIS_H_
#define max_tam 100

typedef enum{//Possíveis cores de um Mineral
    Acizentado,
    Amarelo,
    Azulado,
    Marrom,
    Vermelho
}Cor;

typedef struct{
    char Nome[max_tam];
    double Dureza;
    double Reatividade;
    Cor Cores;
}Minerais;
```

Figura 16 - Minerais.h

A principal função deste TAD é a PreencheMineral, ela recebe os parâmetros Mineral* mineral (estrutura) e char* nome. Com base no nome do passado, retorna os dados do mineral (pré-definidos). Utiliza strcmp() para comparar os nomes. Após isso, Inicializa o mineral com InicializaMineral(Mineral, Nome, Dureza, Reatividade, Cores), passando o resultado obtido com os laços condicionais, atribuindo os valores. Segue uma parte da função que exemplifica como foi feita a inicialização dos dados dos minerais:



```c
#include "Minerais.h"
#include <stdio.h>
#include <string.h>

void PreencheMineral(Minerais *Mineral, char *Nome){

    double Dureza, Reatividade;
    Cor Cores;
    if(strcmp(Nome, "Ferrolita")==0){
        Dureza = 0.5;
        Reatividade = 0.7;
        Cores = Acizentado;
    }
    else if(strcmp(Nome, "Solarium")==0){
        Dureza = 0.9;
        Reatividade = 0.2;
        Cores = Amarelo;
    }
```

Figura 17 - Função PreencheMineral

O TAD Lista Minerais define uma estrutura chamada ListaMinerais que contém um array de minerais e dois índices (Primeiro e Último) que controlam o início e fim de uma lista.Após isso possui funções que inicializam uma lista vazia, verificam se a lista está vazia e adiciona um mineral através desses índices.

- Para inicializar, iguala Primeiro e último.

- Para retirar, verifica se a lista não é vazia e compara os nomes contidos em cada índice, removendo e realocando os itens posteriores.

- Adiciona mineral ao final da lista.

```c
#ifndef LISTAMINERAIS_H_
#define LISTAMINERAIS_H_
#define InicioArranjo 0
#define MaxTam 3
#include "Minerais.h"

typedef struct {

 Minerais listaminerais[MaxTam];
 int Primeiro, Ultimo;

} ListaMinerais;

void IniVListaM(ListaMinerais* ListaM);

void InsMineral(ListaMinerais* ListaM, char *NomeNov);

int RetMineral(ListaMinerais* ListaM, char *Nomed);

void ImprimeListaM(ListaMinerais* ListaM);

void EsvaziaLista(ListaMinerais *ListaM);

#endif // LISTAMINERAIS_H_
```

Figura 18 - ListaMinerais.h

```c
void InsMineral(ListaMinerais* ListaM, char *NomeNov){
    Minerais NovoM;
    PreencheMineral(&NovoM, NomeNov);
    ListaM->listaminerais[ListaM->Ultimo] = NovoM;
    ListaM->Ultimo++;
} /*Não tratei pro caso da lista estar cheia,
    porque obrigatoriamente só receberemos no
    máximo três mineirais de entrada em cada lista*/

int RetMineral(ListaMinerais* ListaM, char *Nomed){
    if(ListaM->Ultimo == ListaM->Primeiro){
        return 0;
    }

    int pos;

    for(int i = 0; i<ListaM->Ultimo;i++){
        if (ListaM->listaminerais[i].Nome == Nomed){
            pos = i;
        }
    }
    for (int j = pos+1; j <= ListaM->Ultimo; j++){
        ListaM->listaminerais[j-1] = ListaM->listaminerais[j];
    }

    ListaM->Ultimo--;

    return 1;
}
```

Figura 19 - Funções insere e retira mineral

O TAD Rocha Mineral representa a rocha formada por um ou até 3 minerais, contendo outras características como identificador e a localização onde ela foi encontrada.

```c
Ana Clara, 3 hours ago | 2 authors (You and one othe
#ifndef ROCHAMINERAL_H_INCLUDED
#define ROCHAMINERAL_H_INCLUDED
#define Data 11
#include "ListaMinerais.h"

You, last week | 1 author (You)
typedef struct{
    int Identificador;
    double Peso;
    char Categoria[max_tam];
    ListaMinerais ListaM;
    double Latitude, Longitude;
    char DataC[Data];
} RochaMineral;
```

Figura 20 - RochaMineral.h

O TAD Compartimento define a estrutura lista de rochas minerais, contendo o peso máximo suportado por aquele compartimento e as rochas armazenadas na lista, foi implementado utilizando o conceito de lista encadeada com célula cabeça e possui a seguinte estrutura.

```c
#ifndef COMPARTIMENTO_H_
#define COMPARTIMENTO_H_
#include "RochaMineral.h"

typedef struct Celula{
    RochaMineral rocha;
    struct Celula* pProx;
} Celula;

typedef struct Compartimento{
    double PesoMax;
    Celula* primeiro;
    Celula* ultimo;
} Compartimento;

void CriaListaRocha(Compartimento *lista, int PesoMax);//Cria uma Lista de Rocha Mineral Vazia
int TamanhoListaRocha(Compartimento *lista);//Retorna o numero(int) de Rochas armazenadas no compartimento
int VerificaListaVazia(Compartimento *lista);//Verifica se a Lista esta vazia
void ImprimiLista(Compartimento *lista);//Exibe todo o conteudo do compartimento
double PesoAtual(Compartimento *lista);//Retorna o peso total do compartimento/lista
void TrocaRocha(Compartimento *lista, RochaMineral *rocha);//Adiciona uma rocha mais leve no lugar da rocha mais pesada d
int InsereRocha(Compartimento *lista, RochaMineral *rocha, int PesoMax);//Insere uma rocha encontrada no final da lista,
RochaMineral *RemoveRocha(Compartimento *lista, RochaMineral *rocha);//Remove a rocha de acordo com a categoria(nome)

#endif
```

Figura 21 - Compartimento.h

As funções básicas deste TAD seguem a lógica apresentada em sala de aula para a implementação de listas encadeadas, algo importante a destacar seria a função Remove rocha, que é utilizada na operação de redistribuição, ela retornará a a rocha a ser removida, para que as informações referentes a ela não sejam perdidas.

```c
RochaMineral *RemoveRocha(Compartimento *lista, RochaMineral *rocha){
    if (VerificaListaVazia(lista)){
        return 0;
    }
    Celula* pAux;
    Celula* pAux2;
    pAux = lista->primeiro->pProx;//Primeira rocha da lista
    while((strcmp(pAux->rocha.Categoria,rocha->Categoria))!=0){
        pAux2 = pAux;
        pAux = pAux->pProx;
    }
    RochaMineral* pAux3 = &pAux->rocha;//Rocha Mineral que queremos remover
    pAux2->pProx = pAux->pProx;
    return pAux3;//Retorna a rocha Mineral que queremos remover
}
```

Figura 22 - Função remove rocha

O TAD Sonda define a estrutura da sonda, criando um tipo de dado e permitindo o encapsulamento. A inicialização recebe as informações referentes a sonda digitados pelo usuário, de acordo com seus respectivos tipos, que serão inseridos na Sonda. Esse trecho é essencial para a criação de uma lista encadeada criada em outro arquivo (ListaSondas.c), permitindo acesso aos dados que serão utilizados nas operações implementadas no sistema.

Além disso, a função calcula distância é parte essencial, pois ela calcula qual a distância euclidiana da sonda para a rocha a ser coletada, permitindo então que no Sistema seja decidido utilizado esta operação qual a melhor sonda para armazenar a rocha coletada.

```c
#ifndef SONDA_H_
#define SONDA_H_

#include "Compartimento.h"

typedef struct {
    int Identificador;
    Compartimento CompartmentoS;
    double Latitude;
    double Longitude;
    int EstaLigada;
    int Velocidade;
    int Combustivel;
} Sonda;

void InicializaSonda(Sonda *NovaSonda, double Latitude, double Longitude,
 int PesoMax, int Velocidade, int Combustivel, int Identificador);

void LigaSonda(Sonda *SondaDes);

void DesligaSonda(Sonda *Sondalig);

double MoveSonda(Sonda *SondaMov, double Latitude, double Longitude);

double CalculaDist(Sonda Sonda, RochaMineral Rocha);

#endif
```

Figura 23 - Sonda.h

O TAD Lista de Sondas foi implementado utilizando lista encadeada com cabeça e possui a seguinte estrutura:

```c
#ifndef LISTASONDAS_H_
#define LISTASONDAS_H_

#include "Sonda.h"

typedef struct Celula_S* Apontador;

typedef struct Celula_S {
    Sonda Sonda;
    struct Celula_S* pProx;
} TCelula_S;


typedef struct {
    Apontador pPrimeiro;
    Apontador pUltimo;
} TSondas;

void FazListaVazia(TSondas* Sondas);
void InsereSonda(TSondas* Sondas, Sonda* sonda);
int RetiraSonda(TSondas* Sondas, int retirar);
void ImprimeSonda(TSondas* Sondas);

#endif
```

Figura 24 - ListaSondas.h

O ListaSondas.h cria um ponteiro para célula da lista encadeada permitindo navegar por ela, facilitando encontrar as células desejadas, que contêm a sonda. Após isso, cria uma célula chamada TCelula_S contendo uma estrutura de sonda e um ponteiro para a próxima célula e, ao final, cria a lista encadeada que possui um ponteiro para a primeira célula e outro para a última, permitindo criar e verificar o seu início e fim.

4. COMPILAÇÃO E EXECUÇÃO

Para a compilação e execução do código no Visual Studio Code, foi usado um tasks.json com duas tasks: Compilacao e Execucao.

Para compilar é definido como label: “Compilacao”; como type: “shell”; o comando como: “gcc” e são passados como argumentos “-o”, “bin/AEDS1” (que é o caminho para usar o executável) e depois o caminho de todos os arquivos de código.

```
{
  "label": "Compilacao",
  "type": "shell",
  "command": "gcc",
  "args": [
    "-o", "bin/AEDS1",
    "src/main.c",
    "src/Sistema de Controle/SistemadeControle.c",
    "src/Sistema de Controle/Lista de Sondas/ListaSondas.c",
    "src/Sistema de Controle/Lista de Sondas/Sonda/Sonda.c",
    "src/Sistema de Controle/Lista de Sondas/Sonda/Compartimentos/Compatimento.c",
    "src/Sistema de Controle/Lista de Sondas/Sonda/Compartimentos/Rochas Minerais/RochaMineral.c",
    "src/Sistema de Controle/Lista de Sondas/Sonda/Compartimentos/Rochas Minerais/Lista de Minerias/ListaMinerais.c",
    "src/Sistema de Controle/Lista de Sondas/Sonda/Compartimentos/Rochas Minerais/Lista de Minerias/Minerais/Minerais.c"
  ],
  "group": {
    "kind": "build",
    "isDefault": true
  },
  "problemMatcher": ["$gcc"]
},
```

Figura 25 - Estrutura no tasks.json para compilação do código

Para executar é definido como label: “Execucao”; como type: “shell” e o comando como “./bin/AEDS1”.

```
{
  "label": "Execucao",
  "type": "shell",
  "command": "./bin/AEDS1",
  "group": {
    "kind": "test",
    "isDefault": true
  }
}
```

Figura 26 - Estrutura no tasks.json para execução do código

Dessa forma, para rodar o código deve-se abrir o terminal integrado e executar o comando "**./bin/AEDS1**".

No caso de ser leitura por arquivo, o arquivo deve estar na mesma pasta do projeto, de forma que após escolher a opção 1, o nome do arquivo deve ser digitado.

Para a leitura interativa, diferentemente, após escolher a opção 2, cada informação deve ser digitada separadamente, respeitando os prints.

5. RESULTADOS

Em relação aos resultados obtidos na criação do Sistema de Controle e Catalogação de Sondas, os resultados que obtivemos utilizando a entrada por terminal e os dados escritos no arquivo disponibilizado no PVanet Moodle, contendo apenas dados para teste, a seguir o Teste 1:

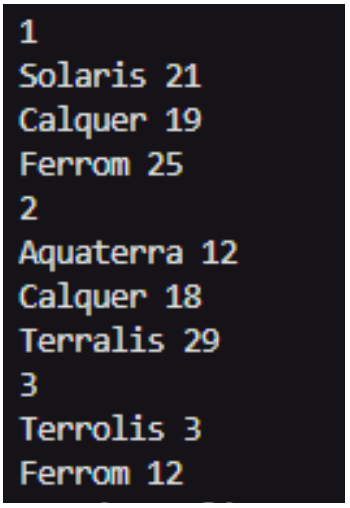


Figura 27 - Resultado antes da operação E

Figura 28 - Resultado após a operação E

Utilizando o os dados do Teste 2 os resultados foram:

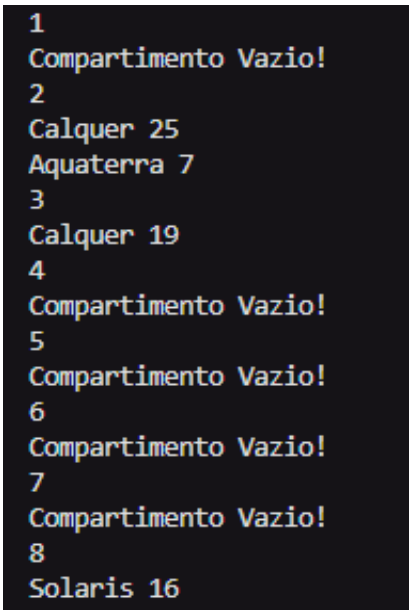


Figura 29 - Resultados antes da operação E

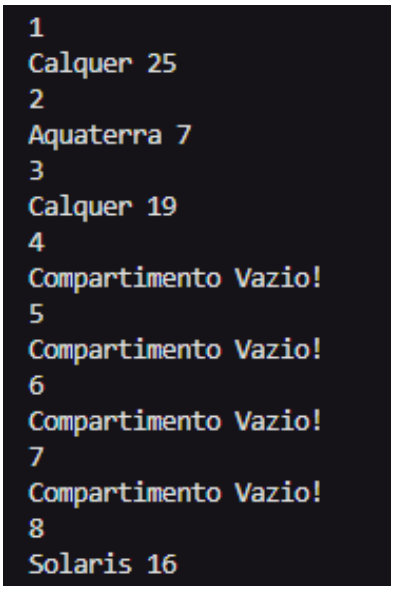


Figura 30 - Resultados após a operação E

Os resultados que obtivemos utilizando a leitura por arquivo e o arquivo Teste1 foram:

```
1
Compartimento Vazio!
2
Aquaterra 12
Calquer 18
3
Compartimento Vazio!
1
Solaris 21
Calquer 19
2
Aquaterra 12
Calquer 18
Terralis 29
3
Terralis 29
3
Terrolis 3
1
Solaris 21
Calquer 19
Ferrom 25
2
Aquaterra 12
Calquer 18
Terralis 29
3
Terrolis 3
Ferrom 12
1
Calquer 19
Ferrom 25
Solaris 21
2
Calquer 18
Terralis 29
3
Terrolis 3
Ferrom 12
Aquaterra 12
```

Figura 31 - Resultados utilizando Teste1

Já utilizando o arquivo Teste2 foram:

```
1
Compartimento Vazio!
2
Calquer 25
Aquaterra 7
3
Calquer 19
4
Compartimento Vazio!
5
Compartimento Vazio!
6
Compartimento Vazio!
7
Compartimento Vazio!
8
Solaris 16
1
Calquer 25
2
Aquaterra 7
3
Calquer 19
4
Compartimento Vazio!
5
Compartimento Vazio!
6
Compartimento Vazio!
7
Compartimento Vazio!
8
Solaris 16
```

Figura 32 - Resultados utilizando Teste2

6. CONCLUSÃO

Em suma, para a realização do trabalho, cada TAD foi criado em uma sequência de dependência “em cadeia”, de forma que o primeiro (Minerais) é o único independente, o segundo depende do primeiro, o terceiro do segundo e assim em diante.

Para a implementação do Sistema de Controle, foram criados também arquivos “.h” e “.c”, com as funções: “Central”, que é chamada pela main e define se a entrada de arquivos será feita por arquivo ou de forma interativa (pelo terminal); “LeituraPorArquivo”, que lê os dados por arquivo; “LeituraPeloTerminal”, que lê os dados de forma interativa; “Coleta”, que escolhe a sonda mais próxima da rocha para inserir; “RedistribuiRochas”, que faz a redistribuição das rochas; e funções auxiliares que foram necessárias para facilitar o funcionamento.

```c
#ifndef SISTEMADECONTROLE_H_      maria eduarda, last week • Criação do Sistema de Controle
#define SISTEMADECONTROLE_H_

#define STRING 100
#define Data 11

#include "ListaSondas.h"

void Central(TSondas *lista);//Contem um menu interativo dando ao usuario a opção de decidir o que ele quer fazer
int LeituraPorArquivo(TSondas *lista);//Lê o arquivo teste e passa os valores lidos para as outras funções
int LeituraPeloTerminal(TSondas *lista);//Lê as informações digitadas no terminal e inicializa o que for passado
void Coleta(Celula Rocha,TSondas *ListaSondas, int numsondas);//Escolhe a sonda mais próxima da rocha para inserir
void RedistribuiRochas(TSondas *ListaSondas, int numsondas);//Faz a redistribuição de rochas


//Funções auxiliares
int VerificaSeTemQuemReceber(TSondas *lista, Compartimento *Comp, double peso);//
void OrdenaPesos(Sonda **Sondas, int numsondas);//
void PreencheVetor(TSondas *Sondas, Sonda **VetorSondas, int numsondas);//

#endif
```

Figura 33 - Funções presentes no Sistema de Controle

Assim, os dois tipos de entrada diferem apenas na hora de receber os dados, usando as mesmas funções para: inicializar e inserir rochas (‘R’ para o arquivo e ‘1’ para o terminal), redistribuir (‘E’ para o arquivo e ‘2’ para o terminal) e imprimir os estados das sondas (‘I’ para o arquivo e ‘3’ para o terminal).

Isso acontece levando em consideração o seguinte detalhe: antes de inicializar rochas, sondas devem ser inicializadas primeiro. De maneira que tanto para o arquivo quanto para o terminal, o primeiro dado a ser lido é a quantidade de sondas da lista.

Sendo assim, comparando o resultado final obtido com o esperado na documentação, não foi atingido exatamente o mesmo, devido a função de redistribuição. Entretanto, todos os TADS funcionam bem e são bem utilizados entre si e pelos arquivos principais, de maneira que o sistema como um todo trata bem de casos diferentes de entradas de dados.

7. REFERÊNCIAS

[1] Github. Disponível em: <https://github.com/> Último acesso em: 24 de novembro de 2024.

[2] Repositório do Trabalho. Disponível em: <https://github.com/lacerdamadu/Trabalho-Pratico> Último acesso em: 24 de novembro de 2024.