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

Trabalho 1 - AEDS 1

Sistema de gerenciamento de sondas espaciais

HEITOR PORTO JARDIM DE OLIVEIRA - 5895
JÚLIO CESAR DE SOUZA OLIVEIRA - 5903
OTÁVIO FRANCISCO SABINO TAVARES - 5912

Florestal - MG
2024

Sumário

1. Introdução

O presente projeto se baseia no desejo da Agência Espacial de Desenvolvimento Sustentável (AEDS) de gerenciar sondas espaciais enviadas para Marte, de maneira a catalogar as rochas minerais presentes no solo do planeta e controlar as sondas. Assim, o projeto visa, por meio do registro de novas rochas e amostras, sua catalogação em compartimentos de armazenamento e funcionalidades de análise e classificação, otimizar as atividades da AEDS na investigação da fertilidade do solo marciano.

A abordagem escolhida no desenvolvimento se baseia no inicial desenvolvimento dos TADs mais simples (mineral, sonda espacial, rocha mineral) e suas funcionalidades básicas (como funções de obtenção e definição de valores e inicialização), seguido do desenvolvimento das estruturas de lista linear (lista de minerais, compartimento, lista de sondas) e suas funcionalidades básicas. Após isso, iniciou-se o desenvolvimento do arquivo ‘main.c’ e dos testes via terminal, seguidos do desenvolvimento da execução por arquivo de entrada.

A estrutura de dados das listas lineares foi feita com o uso de listas encadeadas, com exceção da lista de minerais, que foi feita com o uso de vetor.

Vale citar, ainda, que foram usadas como base as implementações em Ziviani, 2003 [1], além do uso do Guthub [2] para versionamento do projeto.

2. Organização

A organização do projeto é representada na Figura 1 e se dá da seguinte forma: na pasta **src/**, se encontra a implementação do projeto, separada em módulos: uma pasta para cada TAD (com seus respectivos arquivos .h e .c), e o arquivo main.c. Na pasta **input/** estão os arquivos de entrada para teste.

Além disso, está presente no repositório o arquivo **Makefile** construído baseado no material complementar disponibilizado pelos monitores, de forma a compilar os múltiplos arquivos de diferentes pastas. Para tanto, algumas modificações foram necessárias para adaptar o **Makefile** às necessidades específicas do projeto:

```
7   CC=gcc
8   CFLAGS=-Wall -Wextra -g  -I./src/Compartimento -I./src/Lista_sonda_espacial -I./src/ListaMinerais -I./src/Mineral -
    I./src/RochaMineral -I./src/Sonda_espacial ./src/main.c
9   SRC=$(wildcard src/*/*.c src/*/*/*.c)
10  OBJ=$(SRC:.c=.o)
11  TARGET=main.exe
12
13  $(TARGET): $(OBJ)
14          $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)
15
16  %.o: %.c
17          $(CC) $(CFLAGS) -c $< -o $@
18
19  clean:
20          del /Q /S $(OBJ) $(TARGET)
```

Figura 1 - Arquivo Makefile.

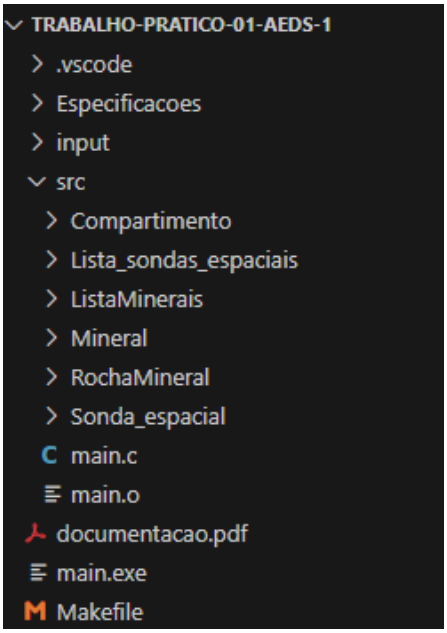

Figura 2 - Repositório do projeto.

3. Desenvolvimento

Observemos, a partir de módulos, as funções mais importantes do projeto.

3.1 TAD ‘Mineral’

No TAD correspondente ao mineral, é preciso citar a função ‘atribui_mineral’:

```c
void atribui_mineral(Mineral * mineral, char nome_mineral[]){//coloca todas as informaçoes em um mineral
//de acordo com o nome recebido
    char cor[20];
    float dureza;
    float reatividade;

    if (strcmp(nome_mineral, "Ferrolita")==0){
        strcpy(cor,"cinza");
        dureza =0.5;
        reatividade =0.7;
    } else if (strcmp(nome_mineral , "Solarium")==0){
        strcpy(cor,"amarelo");
        dureza = 0.9;
        reatividade =0.2;
    }else if (strcmp(nome_mineral , "Aquavitae")==0){
        strcpy(cor,"azul");
        dureza =0.5;
        reatividade =0.8;

    } else if (strcmp(nome_mineral , "Terranita")==0){
        strcpy(cor,"marrom");
        dureza =0.7;
        reatividade =0.6;
    } else if (strcmp(nome_mineral ,"Calaris")==0){
        strcpy(cor,"vermelho");
        dureza =0.6;
        reatividade =0.5;
    }

    inicializa_mineral(mineral,nome_mineral, cor, dureza,reatividade);
}
```

Figura 3 - Função ’atribuiMineral’.

Nela, um mineral é inicializado de acordo apenas com o nome, uma vez que a partir dele sabe-se as outras características (cor, dureza e reatividade). É com isso em mente que se compara a string nome do mineral que vai ser inicializado (recebido como parâmetro junto com o nome) com os nomes de mineral abrangidos pelo projeto, identificando o mineral em questão e atribuindo as características referentes ao mineral com esse nome. Com o fim das comparações e com o mineral devidamente atribuído, inicializa-se o mineral a partir da função ‘inicializa_mineral’, que, como a imagem abaixo mostra, apenas atribui os valores de nome, cor, dureza e reatividade recebidos como parâmetro para o mineral também recebido como parâmetro.

```c
void inicializa_mineral(Mineral * mineral, char nome[], char cor[], float dureza, float reatividade){
    //inicializa um mineral pegando todas as informaçoes
    set_nome(mineral, nome);
    set_cor(mineral, cor);
    set_dureza(mineral, dureza);
    set_reatividade(mineral, reatividade);
}
```

Figura 4 - Função ’inicializa_mineral’.

A função ‘atribui_mineral’ é essencial para o código, uma vez as características cor, dureza, e reatividade do mineral dependem dela para ser definidos.

3.2 TAD ‘ListaMinerais’

No TAD ‘ListaMinerais’, deve-se destacar a função ‘insereMineralLista’, pois é a mais utilizada ao longo do projeto. Ela se dá da seguinte maneira:

```
//função que insere um novo mineral na lista
int insereMineralLista(ListaMinerais* pLista, TItem x){
    //certificando que a lista não está cheia - tamanho 3 ou maior
    if (pLista->Ultimo == MaxTam){
        //se a lista estiver cheia, não é possível adicionar o mineral
        return 0;
    }
    //adicionando o mineral x na última posição disponível
    pLista->minerais[pLista->Ultimo] = x;

    //incrementando o Apontador Ultimo, já que temos mais um elemento na lista
    pLista->Ultimo++;

    //inserção concluída
    return 1;
}
```

Figura 5 - Função ’insereMineralLista’.

A função recebe uma lista de minerais e um item a ser inserido. Inicialmente, verifica-se se a lista está cheia, ou seja, se o tamanho máximo (3) é igual à posição para a qual o apontador ‘Ultimo’ se refere. Se tal condição for verdadeira, o item não é inserido na lista e a função retorna 0, indicando que não foi possível realizar a inserção. Caso contrário, o item é adicionado na posição para a qual o apontador ‘Ultimo’ se refere, ou seja, o item é adicionado na próxima posição disponível. Com isso, o apontador ‘Ultimo’ é incrementado, pois aquela que era a próxima posição disponível está ocupada, e a posição seguinte será a próxima disponível. Após isso, o valor retornado é 1, declarando que a inserção foi concluída e a função termina.

3.3 TAD ‘RochaMineral’

Neste TAD, a função ‘escolheMineral’ merece destaque:

```c
//função que define a categoria da rocha a partir da lista de minerais
void escolheCategoria(RochaMineral *rocha){

    //definindo variáveis booleanas para verificar se existe cada mineral na lista
    int tFerrolita = 0, tSolarium = 0, tAquavitae = 0, tTerranita = 0, tCalaris = 0;
    //imprimeListaMinerais(rocha->lista);
    //passando por todas as posições da lista de minerais
    for(int i = rocha->lista->Primeiro; i < rocha->lista->Ultimo; i++){
        TChave m = rocha->lista->minerais[i].Chave;
        //printf("%s\n", m.nome);
        //verificando, a partir do nome do mineral, qual mineral está na posição i
        if(strcmp(m.nome, "Ferrolita") == 0){
            tFerrolita = 1;
        }else if(strcmp(m.nome, "Solarium") == 0){
            tSolarium = 1;
        }else if(strcmp(m.nome, "Aquavitae") == 0){
            tAquavitae = 1;
        }else if(strcmp(m.nome, "Terranita") == 0){
            tTerranita = 1;
        }else if (strcmp(m.nome, "Calaris") == 0){
            tCalaris = 1;
        }
    }
```

Figura 6.1 - Primeiro processo da função 'escolheCategoria'.

Inicialmente, a função recebe uma rocha mineral como parâmetro e analisa a lista de minerais que ela contém, a partir de variáveis booleanas (valor 0 ou 1) que indicam se os minerais, individualmente, existem na lista. Isso é feito a partir do nome dos minerais da lista em um laço de repetição que passa por todos os minerais da lista. As variáveis são declaradas como 0, mas se o nome do mineral 'm' atual corresponder à Ferrolita, por exemplo, a variável 'tFerrolita' passará a ser 1, indicando que a lista de minerais da rocha em questão possui Ferrolita. Após verificar todos os minerais da lista, sabe-se quais minerais compõem a lista e passamos para o segundo processo da função:

```
    //definindo a categoria da rocha de acordo com os minerais que estão presentes na lista
    if(tAquavitae && tCalaris && tFerrolita){
        setCategoria(rocha, "AQUACALIS");
    }else if(tTerranita && tFerrolita){
        setCategoria(rocha, "TERRALIS");

    }else if(tSolarium && tFerrolita){
        setCategoria(rocha, "SOLARISFER");

    }else if(tCalaris && tAquavitae){
        setCategoria(rocha, "CALQUER");

    }else if(tAquavitae && tTerranita){
        setCategoria(rocha, "AQUATERRA");

    }else if(tTerranita && tSolarium){
        setCategoria(rocha, "TERRASOL");

    }else if(tTerranita && tCalaris){
        setCategoria(rocha, "TERROLIS");

    }else if (tFerrolita && tAquavitae){
        setCategoria(rocha,"AQUAFERRO");

    }else if (tSolarium){
        setCategoria(rocha, "SOLARIS");

    }else if (tFerrolita){
        setCategoria(rocha, "FERROM");
    }
}
```

Figura 6.2 - Segundo processo da função ’escolheCategoria’.

Aqui, atribui-se a categoria à rocha do parâmetro a partir das definições de categoria com base na lista de minerais: se a lista possui Aquavitae, Calaris e Ferrolita, então a categoria da rocha é Aquacalis (primeiro ‘if’). Essa é a lógica seguida nos ‘if’ para definir a categoria correta às rochas. A função ‘escolheCategoria’ é imprescindível para inicializar corretamente as rochas minerais, definindo a categoria a partir apenas da lista de minerais.

3.4 TAD ‘Compartimento’

No compartimento, a função ‘trocar_rocha’ deve ser analisada:

```c
//troca rochas de lugar pela categoria, caso o peso da previamente armazenada seja superior ao da nova
int trocar_rocha(Compartimento* compartimento, RochaMineral* rocha){
    Ccelula* celula = compartimento->primeiro->prox;

    if (!compartimento_eh_vazio(compartimento)){
        while (celula != NULL) {
            //compara primeiro a categoria para achar a rocha a ser trocada
            if (strcmp(celula->rocha.categoria, rocha->categoria) == 0) {
                if (celula->rocha.peso > rocha->peso) { //compara o peso para fazer a troca se necessário
                        compartimento->peso_atual -= celula->rocha.peso;
                        celula->rocha = *rocha;
                        compartimento->peso_atual += celula->rocha.peso;
                        return 1;
                }
            }
            celula = celula->prox;
        }

        return 0;

    }

    return 0;
}
```

Figura 7 - Função 'trocar_rocha'.

A função recebe um compartimento e uma rocha como parâmetro e começa declarando uma célula que indica o elemento que vem após a célula cabeça. Em seguida, verifica-se se o compartimento é vazio: se for, não podemos realizar uma troca, o retorno é 0 e a função encerra; se não, se inicia um 'while' que é responsável por passar pelas posições do compartimento, até que a célula seja nula, ou seja, passa-se por todas as posições do compartimento. Dentro do 'while', verifica-se (no primeiro 'if') se a categoria da rocha na célula atual é igual à categoria da nova rocha passada no parâmetro: se sim, a troca pode acontecer, e é executado um 'if' que verifica se o peso da rocha na célula atual é maior que o peso da rocha passada no parâmetro. Em caso positivo, o peso do compartimento é subtraído pelo peso da rocha que está na célula atual, a nova rocha é colocada na célula atual, e o peso dessa rocha é somado ao peso do compartimento. O valor retornado é 1, assegurando que a troca entre a rocha presente na célula do compartimento pela nova rocha foi feita, e no fim do 'while' a célula a ser analisada a é a próxima célula do compartimento. Com o fim do 'while', a função é encerrada.
Outra que deve ser citada no presente item é a função remover_rocha:

```c
//remove rocha do compartimento por categoria, e entrega a rocha para o usuario, caso ele precise
int remover_rocha(Compartimento* compartimento, char categoria[], RochaMineral* rocha_retirada){
    Ccelula* anterior = compartimento->primeiro;
    Ccelula* celula = compartimento->primeiro->prox;

    while (celula != NULL) {

        if(strcmp(categoria, celula->rocha.categoria) == 0){ //procura pela rocha da categoria certa e a remove

            anterior->prox = celula->prox;
            compartimento->peso_atual -= celula->rocha.peso;
            *rocha_retirada = celula->rocha;

            if (celula->prox == NULL) compartimento->ultimo = anterior;
            compartimento->tamanho--;

            free(celula);
            return 1;
        }

        anterior = celula;
        celula = celula->prox;
    }

    return 0;
}
```

Figura 8 - Função ’remover_rocha’.

A função trabalha com base em um compartimento, uma categoria e uma rocha a ser retirada. De início, são declaradas as células referentes à célula atual e à célula anterior, de maneira semelhante àquela utilizada na função anterior. Um ‘while’ é executado que passa por todas as posições do compartimento e nele se compara a categoria do parâmetro e a categoria da rocha na célula atual: se são a mesma categoria, a célula anterior passa a ser a seguinte à atual, “pulando” a célula da rocha a ser removida. Também de maneira semelhante à função ‘trocar_rocha’, o peso do compartimento é subtraído pelo peso da rocha a ser removida, a rocha removida é atribuída ao ponteiro ‘rocha_removida’ para que o usuário receba tal rocha. Em seguida, verifica-se se a próxima célula é nula: se sim, a próxima célula corresponde ao último elemento do compartimento, representado pela célula ‘ultimo’. Por fim, o tamanho do compartimento diminui, uma vez que tem um elemento a menos, a célula removida é liberada da memória e a função retorna 1, representando que a remoção foi concluída. No fim do ‘while’, as células atual e anterior são atualizadas para serem tratadas as próximas células na nova execução do laço de repetição.
Ambas as funções tratadas neste item são importantes para o funcionamento das operações desejadas no projeto.

3.5 TAD 'Sonda_espacial’

No TAD referente à sonda espacial, temos a função ‘inicializa_Sonda_Espacial’:

```c
//inicializa uma sonda e preenche os seus campos usando sets
void inicializa_Sonda_Espacial(Sonda_espacial * sonda, char * id, float latitude,
                               float longitude, float capacidade, float velocidade, float combustivel){
    set_Identificador(sonda, id);
    set_Compartimento(sonda, capacidade);
    set_Localizacao_sonda(sonda,latitude,longitude);
    set_Velocidade(sonda, velocidade);
    set_Combustivel(sonda, combustivel);
    set_EstaLigada(sonda, OFF);
}
```

Figura 9 - Função ’inicializa_Sonda_Espacial’.

Nessa função uma sonda é inicializada a partir dos parâmetros que se referem às suas características, usando os métodos ‘set’.

3.6 TAD 'Lista_sonda_espacial’

Neste TAD, tratemos primeiro da função ‘retira_item_lista_sonda_espacial’:

```c
int retira_item_lista_sonda_espacial(Lista_sonda_espacial* lista_sonda, char * id, Sonda_espacial * pitem){
    if (verifica_lista_vazia(lista_sonda)){
        return 0;

    }
    else{
        /*percorre a lista de 2 a 2 de modo que quando encontrar o item a ser removido
        o pProx anterior consiga apontar para o próximo item após remoção.*/
        Celula* pAux = lista_sonda->pPrimeiro->pProx;
        Celula* pAux_ant = lista_sonda->pPrimeiro;
        int count = 1;
        //percorre a lista até encontrar o ultimo item
        while (pAux!= NULL){
            if(*pAux->item_sonda.Identificador == *id){//verifica se o id a ser removido é o da sonda atual
                //verifica se a sonda a ser removida é o último elemento da lista para efetuar as devidas mudanças
                if (count == lista_sonda->QntItens){
                    *pitem = pAux->item_sonda;
                    pAux_ant->pProx = NULL;
                    lista_sonda->pUltimo = pAux_ant;
                    free(pAux);
                    return 1;
                }
                else{
                    *pitem = pAux->item_sonda;
                    pAux_ant->pProx = pAux->pProx;
                    free(pAux);
                    return 1;
                }
            }
```

Figura 10.1 - Primeira parte da função ’retira_item_lista_sonda_espacial’

A função toma como parâmetros uma lista de sondas, um identificador e uma sonda. No início, verifica-se se a lista é vazia: se sim, a função é encerrada. Se não, são criados os ponteiros ‘pAux’, correspondente à primeira célula após a cabeça, e ‘pAux_ant’, que corresponde à cabeça. Além disso, é criado o ‘cont’, para contar a posição dos itens. Logo, tem-se o ‘while’ que percorre todas as posições da lista de sondas. Dentro do ‘while’, verifica-se de o identificador da sonda na célula atual é igual ao identificador do parâmetro. Se sim, verifica-se se a célula atual é a última: se sim, a sonda do parâmetro passa a corresponder à sonda da célula atual, enquanto

a célula atual passa a ser nula. Já o ‘pUltimo’ passa a apontar para a célula anterior, fazendo com que o último item da lista seja o penúltimo. Assim, a memória da célula ‘pAux’ é liberada. Caso este último ‘if’ seja falso, ou seja, a sonda a ser removida não é o último item, o ponteiro ‘pProx’ do item anterior é atualizado para apontar para o item seguinte de ‘pAux’, removendo a célula ‘pAux’ da lista. Caso a sonda não seja encontrada, é exibida a seguinte mensagem antes da função ser encerrada:

```
            }
            printf("sonda com identificador %s não encontrado.", id);
            return 0;
        }
    }
```

Figura 10.2 - Segunda parte da função ‘retira_item_lista_sonda’.

Agora, sobre a função ‘insere_item_lista_sonda_espacial’:

```
void insere_item_lista_sonda_espacial(Lista_sonda_espacial * lista_sonda, Sonda_espacial* pItem){
    lista_sonda->pUltimo->pProx = (Celula*) malloc (sizeof(Celula));
    lista_sonda->pUltimo = lista_sonda->pUltimo->pProx;
    lista_sonda->pUltimo->item_sonda = *pItem;
    lista_sonda->QntItens ++;
    lista_sonda->pUltimo->pProx = NULL;

}
```

Figura 11 - Função ’insere_item_lista_sonda_espacial’.

Nessa função, que leva como parâmetros uma lista de sondas e uma sonda, se aloca dinamicamente a memória para a nova célula a ser adicionada e o ponteiro ‘pUltimo->prox’ passa a apontar à nova célula. Depois, o ponteiro referente ao último item da lista passa a apontar para a nova célula. Em seguida, os atributos da sonda do parâmetro são atribuídos à nova célula. Por fim, a quantidade de itens na lista aumenta, e se indica que a nova célula é a última.

3.7 'main.c’

Abordemos a diferenciação de leitura por terminal e por arquivo, começando por arquivo, que se inicia com a verificação dos parâmetros de entrada e com a verificação se o arquivo está integro ou não:

```
int main(int argc, char **argv){
    if (leitura_arq(argc,argv) != 0){
        FILE *file = leitura_arq(argc,argv);
        if (file == NULL){//verifica se o arquivo tem algo a ser lido
            printf("Arquivo de entrada Invalido");
            return 0;
        }
```

Figura 12 – Verificação da entrada por arquivo

Uma vez que se obtém um arquivo válido, se inicia a execução da seguinte forma:

```c
//cria e inicializa sonda espacial
Lista_sonda_espacial lista_de_sondas_file;
inicializa_lista_sonda_espacial(&lista_de_sondas_file);

//recebe numero de sondas
int numero_sondas = 0;
fscanf(file, "%d", &numero_sondas); fgetc(file);
if(numero_sondas == 0){//verifica se e numero valido
    printf("Nenhuma sonda foi enviada pela AEDS\n");
    return 0;
}

preenche_sonda_arq(numero_sondas, file, &lista_de_sondas_file);

//le o numero de instrucoes a serem executadas
int N_instrucao = 0;
fscanf(file,"%d",&N_instrucao); fgetc(file);
```

Figura 12.1 - Início da leitura por arquivo.

Nesse momento, se inicializa uma lista de sondas e é lido o número de sondas que serão utilizadas. Assim, são preenchidas as sondas com o uso da função ‘preenche_sonda_arq’ abaixo:

```c
//percorre as sondas criadas, preenchendo com os valores do arquivo e inserindo em uma lista de sondas
void preenche_sonda_arq(int numero_sondas, FILE * file, Lista_sonda_espacial * lista_de_sondas_file){
    for (int i = 0; i < numero_sondas; i++){
            char id[20];
            sprintf(id, "%d", i+1);
            Sonda_espacial sonda;
            float lat_i,long_i,capacidade_i,velocidade_i,combustivel_i;

            fscanf(file, "%f %f %f %f %f", &lat_i,&long_i,&capacidade_i,&velocidade_i,&combustivel_i);
            fgetc(file);

            inicializa_Sonda_Espacial(&sonda, id, lat_i,long_i,capacidade_i,velocidade_i,combustivel_i);
            insere_item_lista_sonda_espacial(lista_de_sondas_file,&sonda);
    }
}
```

Figura 12.1.1 - Função ’preenche_sonda_arq’.

Nela, para um dado número de sondas, lê-se do arquivo as características de cada uma e, dada também a lista de sondas, as sondas são inicializadas e adicionadas na lista.

Após a execução da função acima, é lido o número de instruções, e temos:

```c
//executa a quantidade de instrucoes informadas
for(int i=0;i<N_instrucao;i++){
    char instrucao;

    //le a instrucao informada
    fscanf(file,"%c", &instrucao); fgetc(file);

    //Verifica qual foi o caractere inserido e aciona o "case" correspondente
    switch (instrucao)
    {
    case 'R':{
        ListaMinerais lista_minerais_file;
        fListaMineraisVazia(&lista_minerais_file);
        case_R_file(file, &lista_minerais_file,  &lista_de_sondas_file);
        break;}
    case 'I':
        //chamada da operacao I com apenas a lista de sondas como parametro
        operacao_I(&lista_de_sondas_file);
        break;
    case 'E':
        //chamada da operacao E com apenas a lista de sondas como parametro
        operacao_E(&lista_de_sondas_file);
        break;
    default:
        //caso seja fornecido alguma operacao invalida, ele desconsidera a operacao e apssa para a proxima
        printf("operacao invalida\n");
        break;
    }
}
```

Figura 12.2 - Recebendo as instruções

Nesse momento, é executado um laço de repetição que lê as ‘N_instrucao’ (número de instruções) e realiza o ‘switch’ de acordo com a operação lida no arquivo. Antes de entrar nas especificidades da operação R por arquivo, deve-se tratar das operações R, I e E.

3.7.1 Operação R

A função referente à operação R é a que se segue:

```c
//Operacao responsavel por coletar uma nova rocha
void operacao_R(Lista_sonda_espacial * lista_sondas, float lat_rocha, float long_rocha, float peso_rocha, ListaMinerais* lista_minerais){
    static int contId = 1;

    LocalRochaMineral local = { lat_rocha, long_rocha };
    RochaMineral rocha_file;

    //inicializa a rocha a ser coletada
    inicializaRochaMineral(&rocha_file, contId, peso_rocha , lista_minerais,local,"00:00:00");

    int cont = lista_sondas->QntItens;
    float menor_d = INFINITY;
    float distancia = 0;

    Celula* sonda_mais_perto = NULL;
    Celula* aux = lista_sondas->pPrimeiro->pProx;
```

Figura 13.1 – Primeira parte da operação R.

A função tem como parâmetros a lista de sondas e as características da rocha. Inicialmente, é declarada uma variável estática do tipo inteiro, que será útil para gerar um identificador único para cada rocha coletada. Assim, inicializa-se a rocha coletada, ‘cont’ recebe o número de sondas na lista, ‘menor_d’ recebe infinito e ‘distancia’

recebe 0. Em seguida, ‘sonda_mais_perto’ aponta para a célula na qual se encontra a sonda mais próxima, enquanto ‘aux’ aponta para a primeira célula.

```
if (!aux) return;
//inicia o processo de percerrer todas as sondas e verificar se atende aos requisitos para coletar a rocha
for (int i = 0;i< cont; i++) {
    distancia = calcula_distancia(aux->item_sonda.Localizacao_sonda.Longitude,aux->item_sonda.Localizacao_sonda.Latitude,
                                  long_rocha, lat_rocha);
    //verifica a primeira condicao para coletar uma rocha: ser a sonda mais perto
    if (distancia < menor_d){//se a sonda em questao for a mais perto no momento, verifica o segundo requisito
        if(trocar_rocha(&aux->item_sonda.Compartimento, &rocha_file) == 1){//Ja ter uma rocha da mesma categoria mais pesada
            menor_d = distancia;
            sonda_mais_perto = aux;
            move_Sonda_Espacial(&sonda_mais_perto->item_sonda,lat_rocha,long_rocha);
            return;
        }else if ((aux->item_sonda.Compartimento.peso_atual + rocha_file.peso) <= aux->item_sonda.Compartimento.peso_maximo){
            //ou ter capacidade de armazenar a rocha
            menor_d = distancia;
            sonda_mais_perto = aux;
        }
    }
    aux = aux->pProx;


}
```

Figura 13.2 - Segunda parte da operação R.

Nessa etapa, verifica-se, inicialmente, se ‘aux’ é nulo (lista vazia); se sim, a função termina. Se não, é executado um laço de repetição responsável por passar por todas as posições da lista de sondas. Dentro dele, se calcula a distância entre a localização da sonda e a localização da rocha, da seguinte forma:

```
//recebe como parametro as coordenadas x e y da sonda e da rocha e retorna o valor da distancia euclidiana
float calcula_distancia(float x1, float y1, float x2, float y2){
    return (sqrt(pow((x1 - x2),2) + pow((y1 - y2),2)));
}
```

Figura 13.2.1 - Função ’calcula_distancia’.

Nessa função, é usada a fórmula de cálculo de distância euclidiana:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Após a atribuição dessa distância, são verificados os requisitos para inserção da rocha nas sondas: o ‘if(distancia < menor_d)’ verifica se a sonda é a mais próxima; se sim, temos mais dois ‘if’: o primeiro verifica se já existe uma rocha mais pesada da mesma categoria; o segundo verifica se a sonda tem capacidade para armazenar a rocha. Se um dos dois ‘if’ for verdadeiro, a sonda é movida até a rocha por meio da função ‘move_Sonda_Espacial’, que apenas define a localização da sonda com base nos parâmetros de latitude e longitude.

```
    if(sonda_mais_perto == NULL) return;
    //se nenhuma sonda contem a rocha, a sonda mais perto encontrada ira mover-se ate a rocha e coleta-la
    move_Sonda_Espacial(&sonda_mais_perto->item_sonda,lat_rocha,long_rocha);
    inserir_rocha(&sonda_mais_perto->item_sonda.Compartimento, &rocha_file);
    contId++;
}
```

Figura 13.3 - Final da operação R.

Se nenhuma sonda atender aos requisitos para coletar a rocha, o laço termina e o código verifica se ‘sonda_mais_perto’ foi atualizada para representar uma sonda

válida: se ‘sonda_mais_perto’ for nula, a sonda mais próxima será movida para a localização da rocha, a rocha será inserida na sonda e ‘contId’ será incrementado para que a próxima rocha tenha um identificador diferente.

3.7.2 Operação I

A operação I se dá da seguinte forma:

```
//Operacao que imprime as rochas insertidas nas sondas
void operacao_I(Lista_sonda_espacial * lista_sondas){

    int cont_sondas  = lista_sondas->QntItens;
    Celula* aux = lista_sondas->pPrimeiro->pProx;
    //percorre toda a lista de sodas e imprime o compartimento
    for (int i = 0;i < cont_sondas; i++){
        printf("%s\n", aux->item_sonda.Identificador);

        imprime_compartimento(&aux->item_sonda.Compartimento);
        aux = aux->pProx;
    }
}
```

Figura 14 - Operação I.

A função recebe a lista de sondas e, imprime a situação de cada compartimento.

3.7.3 Operação E

A função começa verificando se a lista é vazia. Em seguida, são declaradas variáveis de maneira análoga a como já foi feito anteriormente. Em seguida, um ‘for’ passa por todas as sondas, movendo-as para o ponto de origem:

```
//operacao responsavel por realizar a redistribuicao das rochas entre as sondas
int operacao_E(Lista_sonda_espacial * lista_sondas){
    if (verifica_lista_vazia(lista_sondas)){//verifica se existem rochas a serem redistribuídas
        return 0;
    }

    int cont_sondas = lista_sondas->QntItens;
    int quantidade_de_rochas = 0;
    Celula *aux = lista_sondas->pPrimeiro->pProx;

    //move todas as sondas para a origem (ponto (0,0)) e conta quantas rochas foram coletadas no total
    for (int i = 0; i < cont_sondas; i++){
        move_Sonda_Espacial(&aux->item_sonda, 0 , 0);
        quantidade_de_rochas += aux->item_sonda.Compartimento.tamanho;
        aux = aux->pProx;
    }
```

Figura 15.1 - Primeira parte da operação E.

Depois, cria-se uma lista temporária de rochas, dinamicamente alocada, reinicializa-se o ponteiro ‘aux’ para passar por todas as posições da lista de sondas.
O laço ‘for’ que vem a seguir passa por todas as sondas e a inicialização do ‘apontador’ serve para que ele passe por todas as posições do compartimento no ‘while’ que vem em seguida. Esse ‘while’ percorre as rochas do compartimento de

cada sonda, removendo-as usando a função ‘remover_rocha’ e armazenando as rochas no vetor ‘lista_rochas’:

```
//cria uma lista temporaria de acordo com essa quantidade total de rochas coletadas
RochaMineral* lista_rochas= (RochaMineral*) malloc(quantidade_de_rochas * sizeof(RochaMineral)); //faz lista de rochas para redistribuir depois
aux = lista_sondas->pPrimeiro->pProx;
Ccelula* comp_aux = NULL;
Ccelula* apontador = NULL;
int indice = 0;
//percorre toda a lista de sondas esvaziando os seus compartimentos e armazenando as rochas em um vetor de rochas temporario
for (int counter = 0; counter < cont_sondas; counter++){
    apontador = aux->item_sonda.Compartimento.primeiro->prox;

    while (apontador != NULL) { // percorre todas as rochas do compartimento de cada sonda, removendo-as e adicionando no vetor
        RochaMineral rocha_retirada;
        comp_aux = apontador;
        apontador = apontador->prox;
        remover_rocha(&aux->item_sonda.Compartimento, comp_aux->rocha.categoria, &rocha_retirada);
        lista_rochas[indice++] = rocha_retirada;
    }

    aux = aux->pProx;
}
```

Figura 15.2 - Segunda parte da operação E.

Os dois ‘for’ encapsulados que vêm em seguida servem para ordenar 'lista_rochas’ colocando as rochas mais pesadas no início do vetor:

```
//ordena o vetor de forma decrescente a partir do peso das rochas
for (int i = 0; i < quantidade_de_rochas; i++){
    for (int j = i; j < quantidade_de_rochas; j++){
        if (lista_rochas[i].peso<lista_rochas[j].peso){
            RochaMineral auxiliar = lista_rochas[i];
            lista_rochas[i] = lista_rochas[j];
            lista_rochas[j] = auxiliar;
        }
    }
}
```

Figura 15.3 - Terceira parte da operação E.

Por fim, é feita a redistribuição das rochas entre as sondas:

```
//distribui as rochas nas sondas, sempre priorizando a sonda com menor carga atual e com capacidade para armazenar a rocha
for (int i = 0; i < quantidade_de_rochas; i++){
    aux = lista_sondas->pPrimeiro->pProx;
    Celula* menor = lista_sondas->pPrimeiro->pProx;
    for (int j = 0 ; j < lista_sondas->QntItens ; j++){//descobre qual a sonda com o menor peso atual

        if (aux->item_sonda.Compartimento.peso_atual < menor->item_sonda.Compartimento.peso_atual &&
        aux->item_sonda.Compartimento.peso_atual + lista_rochas[i].peso <= aux->item_sonda.Compartimento.peso_maximo){
            menor = aux;
        }

        aux = aux->pProx;
    }
    //após percorrer todas as sondas, insere a rocha na sonda adequada
    inserir_rocha(&menor->item_sonda.Compartimento, &lista_rochas[i]);
}

return 1;
```

Figura 15.4 - Final da operação E.

O primeiro laço ‘for’ percorre todas as rochas ordenadas e para cada rocha, a função percorre todas as sondas e encontra a sonda com a menor carga e que tenha capacidade suficiente para armazenar a rocha, utilizando o laço interno ‘for’. A sonda com o menor peso atual e capacidade disponível é escolhida para armazenar a rocha em questão e após encontrar a sonda adequada, a função ‘inserir_rocha’ é chamada para inserir a rocha no compartimento da sonda.

3.7.4 Operação R na execução por arquivo/terminal

Retomando o assunto tratado no começo do item 3.7, recordemos que o ‘switch’ da operação na execução por arquivo se dá da seguinte forma:

```
case 'R':{
    ListaMinerais lista_minerais_file;
    fListaMineraisVazia(&lista_minerais_file);
    case_R_file(file, &lista_minerais_file,  &lista_de_sondas_file);
    break;}
```

Figura 16.1 - ‘case ‘R’’ na leitura por arquivo.

A função ‘case_R_file’ é estruturada a seguir:

```
//inicia a leitura dos dados para a operacao R, dispostos em uma linha do arquivo
void case_R_file(FILE * file, ListaMinerais * lista_minerais_file, Lista_sonda_espacial * lista_de_sondas_file){
    char linha[255];
    char *buffer = NULL;
    const char delim[2] = " ";
    float lat_rocha = 0, long_rocha = 0, peso_rocha = 0;
    fgets(linha,255,file);
    Mineral minerais[3];
    char nome_mineral[15];
    linha[strlen(linha) - 1] = '\0';//captura a linha logo após a instrucao R

    /*secciona a linha capturada a partir do padrao definido no delim ou seja,
    separa os 3 primeiros tokens entre espaco e atribui para sua variavel correspondente.*/
    buffer = strtok(linha,delim);
    lat_rocha = atof(buffer);

    buffer = strtok(NULL,delim);
    long_rocha = atof(buffer);

    buffer = strtok(NULL,delim);
    peso_rocha = atof(buffer);

    int m = 0;
    fListaMineraisVazia(lista_minerais_file);
```

Figura 16.2 - Primeira parte da função ’case_R_file’.

A função começa lendo uma linha do arquivo com a função ‘fgets()’, que captura a linha contendo os dados da rocha a ser coletada e a linha é armazenada. Em seguida, o caractere ‘\n’ no final da linha é removido com para garantir que a string fique corretamente formatada. Depois, as variáveis para armazenar a latitude, longitude, e o peso da rocha são inicializadas com valores 0, um vetor de minerais é declarado para armazenar os minerais que serão associados à rocha e uma variável é usada para capturar os nomes dos minerais lidos da linha.

Após isso, a linha é separada com a função ‘strtok’, que usa o delimitador (espaço).

```c
/*conclui a leitura da linha lendo os ultimos tokens e inserindo na lista. Esses tokens varia de 1 a 3,
por isso a verificacao que encerra o while ao chegar em um valor NULL*/
while((buffer = strtok(NULL, delim)) != NULL){
    strcpy(nome_mineral, buffer);
    atribui_mineral(&minerais[m], nome_mineral);
    TItem a = {minerais[m]};
    insereMineralLista(lista_minerais_file, a);
    m++;
}
//chamada da operacao R, que tem como parametro a lista de sondas e os dados da rocha
operacao_R(lista_de_sondas_file,lat_rocha, long_rocha, peso_rocha, lista_minerais_file);
```

Figura 16.3 - Final da função 'case_R_file'.

No 'while', cada mineral é atribuído a 'minerais[m]' inserido na lista.
Por fim, é acionada a já conhecida 'operação_R'.

Deve-se perceber, por fim, que a execução da operação R por terminal se dá de maneira análoga.

4. Compilação e Execução

Para compilar o projeto, é usado o comando **make** do mingw:

```
C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-01-AEDS-1> mingw32-make
```

Figura 17 - Comando de compilação

Para realizar a execução, deve-se ter em mente as duas formas diferentes de entrada: por arquivo e por terminal.

Para executar por terminal, deve-se digitar no terminal o comando **.\main**:

```
C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-01-AEDS-1> .\main
```

Figura 18.1 - Comando de execução via terminal

Já na execução por arquivo, deve-se digitar no terminal **.\main -f** seguido do endereço do arquivo em questão:

```
C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-01-AEDS-1> .\main -f .\input\teste1.txt
```

Figura 18.2 - Comando de execução via arquivo de entrada

5. Resultados

5.1 Execução via terminal

A figura abaixo mostra o início da entrada de dados: o número lido de sondas é 8, e em seguida são lidas as informações de cada uma das 8 sondas:

```
PS C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-01-AEDS-1> .\main
Iniciando missao espacial
Digite o numero de sondas enviada pela AEDS:
8
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 1 (ex: -2 10 50 12 100):
-87.557319 -131.619105 32 16 79
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 2 (ex: -2 10 50 12 100):
-29.658319 54.649626 42 11 57
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 3 (ex: -2 10 50 12 100):
48.656635 -159.429645 73 13 73
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 4 (ex: -2 10 50 12 100):
-89.965611 127.148451 67 11 90
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 5 (ex: -2 10 50 12 100):
-73.09033 43.000027 28 11 67
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 6 (ex: -2 10 50 12 100):
76.156641 174.918481 61 13 59
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 7 (ex: -2 10 50 12 100):
-24.109513 149.231262 50 13 56
Digite a latitude, longitude, capacidade, velocidade e combustivel da sonda 8 (ex: -2 10 50 12 100):
-15.708167 -120.506503 57 12 67
```

Figura 19.1 - Leitura de dados das sondas

Em seguida, é lido o número de operações que serão realizadas (7) e, abaixo, são lidas as quatro primeiras operações, que são operações R, nas quais são informados os dados da rocha coletada:

```
Digite o numero de operacoes que as sondas realizarao:
7
Insira a operacao 1:
R
Digite a latitude, longitude, peso, categoria e os minerais da rocha (ex: -4.6 137.5 20 Ferrolita Aquavitae):
-34.552032 -136.291597 16 Solarium
Insira a operacao 2:
R
Digite a latitude, longitude, peso, categoria e os minerais da rocha (ex: -4.6 137.5 20 Ferrolita Aquavitae):
71.602894 -172.649438 19 Calaris Aquavitae
Insira a operacao 3:
R
Digite a latitude, longitude, peso, categoria e os minerais da rocha (ex: -4.6 137.5 20 Ferrolita Aquavitae):
78.782967 42.951206 25 Calaris Aquavitae
Insira a operacao 4:
R
Digite a latitude, longitude, peso, categoria e os minerais da rocha (ex: -4.6 137.5 20 Ferrolita Aquavitae):
80.71773 -27.109002 7 Aquavitae Terranita
```

Figura 19.2 - Realização das operações R.

Logo, é lida a operação 5, que é uma operação I, e a saída desejada é mostrada:

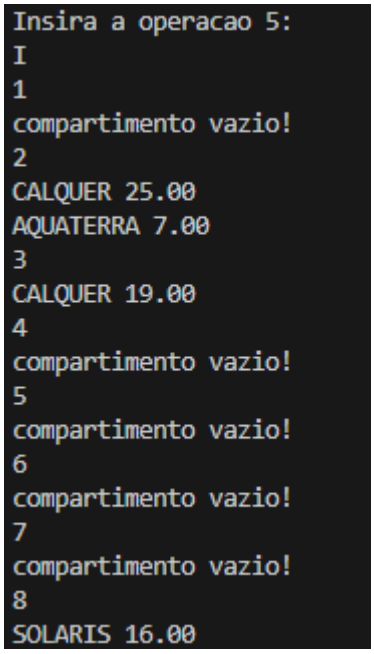

Figura 19.3 - Entrada da operação I e sua saída.

Por fim, o programa recebe a operação 7, que é a operação E, e após reorganizar as rochas nas sondas, é lida a operação 8, que é uma operação I; sua saída é mostrada em seguida:

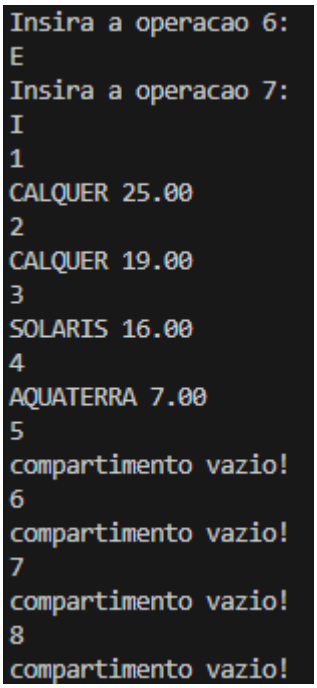

Figura 19.4 - Leitura das últimas operações e saída da operação I.

Com isso, a execução por terminal chega ao fim.

5.2 Execução via arquivo de entrada

Abaixo, lê-se o arquivo de entrada referente a este exemplo:

```
3
-39.341947 142.872653 91 20 79
-60.450806 -112.832771 73 11 55
7.969169 168.985042 41 7 63
12
R
55.489421 -150.457196 12 Aquavitae Terranita
R
16.718537 -148.127484 18 Calaris Aquavitae
I
R
-28.452065 84.537518 21 Solarium
R
-73.005243 -76.093169 29 Terranita Ferrolita
R
-15.677243 -44.787665 19 Calaris Aquavitae
R
9.344208 172.900976 3 Terranita Calaris
I
R
76.926237 147.437399 12 Ferrolita
R
13.629488 -76.551737 25 Ferrolita
E
I
```

Figura 20 - Arquivo de entrada ‘teste1.txt’.

Tendo como entrada o arquivo acima, a execução no terminal se dá da seguinte forma:

```
PS C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-01-AEDS-1> .\main -f .\input\teste1.txt
1
compartimento vazio!
2
AQUATERRA 12.00
CALQUER 18.00
3
compartimento vazio!
1
SOLARIS 21.00
CALQUER 19.00
2
AQUATERRA 12.00
CALQUER 18.00
TERRALIS 29.00
3
TERROLIS 3.00
1
TERRALIS 29.00
AQUATERRA 12.00
FERROM 12.00
2
FERROM 25.00
CALQUER 18.00
TERROLIS 3.00
3
SOLARIS 21.00
CALQUER 19.00
```

Figura 21 - Execução tendo ’teste1.txt’ como arquivo de entrada.

Analisemos a entrada e a saída simultaneamente.

No começo da execução temos na entrada:

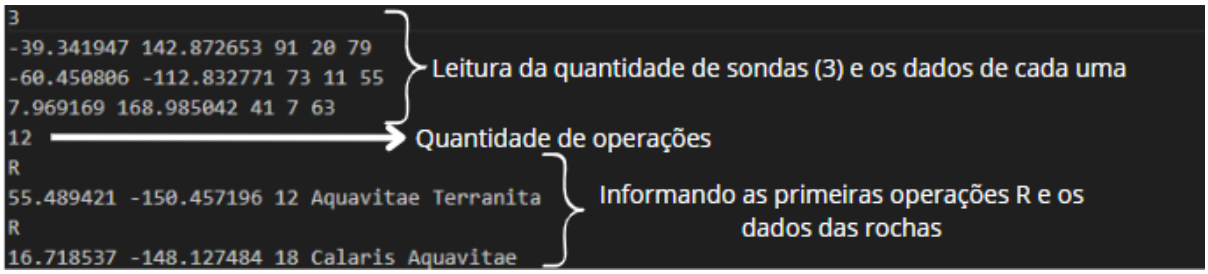

Figura 20.1 - Primeira parte da entrada.

Na linha seguinte, lê-se a operação I e, na saída temos:

```
1
compartimento vazio!
2
AQUATERRA 12.00
CALQUER 18.00
3
compartimento vazio!
```

Figura 21.1 - Saída da primeira operação I.

Em seguida, na entrada temos, de maneira semelhante, a leitura de operações R e os dados dos minerais coletados:

```
R
-28.452065 84.537518 21 Solarium
R
-73.005243 -76.093169 29 Terranita Ferrolita
R
-15.677243 -44.787665 19 Calaris Aquavitae
R
9.344208 172.900976 3 Terranita Calaris
I
```

Figura 20.2 - Segunda parte da entrada, após primeira operação I.

Com a função I representada na imagem acima, obtemos a seguinte saída relativa à situação das sondas:

```
1
SOLARIS 21.00
CALQUER 19.00
2
AQUATERRA 12.00
CALQUER 18.00
TERRALIS 29.00
3
TERROLIS 3.00
```

Figura 21.2 - Saída da segunda operação I.

Na sequência, são lidas as últimas operações - duas R com os dados das rochas, uma E e uma I:

```
R
76.926237 147.437399 12 Ferrolita
R
13.629488 -76.551737 25 Ferrolita
E
I
```

Figura 20.3 - Última parte da entrada

Como há uma operação E antes da I, as rochas são reorganizadas nas sondas e a saída da função I é:

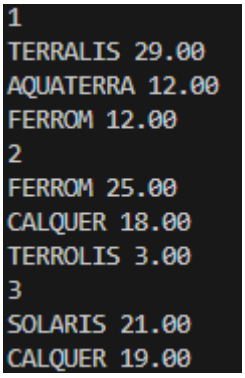

Figura 21.3 - Saída da última função I, após a reorganização

Assim, a execução pelo arquivo ‘teste1.txt’ chega ao fim.

6. Conclusão

Este projeto simula um sistema de gerenciamento de sondas espaciais e coleta de rochas minerais, no qual é necessária ampla interação entre os TADs, desenvolvimento consciente de listas lineares – usando vetor e listas encadeadas -, além da implementação de várias funcionalidades que visam o melhor funcionamento das ferramentas de organização e gerenciamento.

Por fim, destaca-se o trabalho realizado como uma implementação prática de todo a teoria aprendida nas aulas acerca de TADs e listas lineares. Com esse trabalho fomos capazes de entender e aplicar melhor os conceitos, o conhecimento da linguagem, e o conhecimento da lógica de programação.

7. Referências

[1] ZIVIANI, Nivio. **Projeto de algoritmos com implementações em Pascal e C.** 4ª Edição, São Paulo. Editora Pioneira, 1999. Disponível em: <https://www.cin.ufpe.br/~jndm/edados/referencias/Projeto%20de%20Algoritmos%20Com%20Implementa%C3%A7%C3%B5es%20em%20Pascal%20e%20C%20(Nivio%20Ziviani,%204ed).pdf>. Último acesso em: 25 de novembro de 2024.
[2] Github. Disponível em: <https://github.com/> Último acesso em: 25 de novembro de 2024.