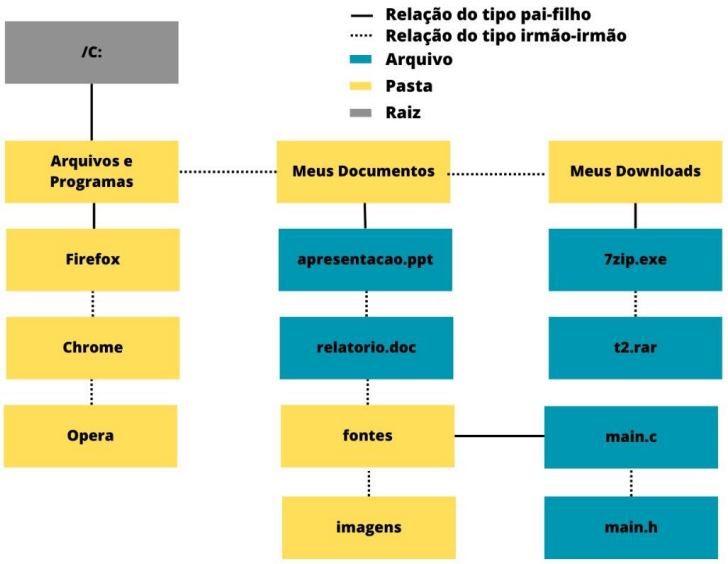
Integrantes: Luis Henrique da Silva Resende, Kauan Felipe de Moura, Iuri Almeida Pereira.




Estrutura da Arvore gerada pelo programa.

```
Diretorio* cria_usuario(){
    Diretorio *dir;
    dir = (Diretorio*) malloc(sizeof(Diretorio));
    if(dir != NULL){
        strcpy(dir->tipo, "");
        dir->irmao = NULL;
        dir->filho = NULL;
        strcpy( dir->caminho, "C:");
        return dir;
    }else {
        return NULL;
    }

}
```

Função responsável por realizar a criação da raiz na árvore. Aloco a memória, defino os ponteiros como NULL, o tipo como “”(pasta),  e defino o caminho como “C:”, que será o nosso diretório pai.

```
void mkdir(Diretorio* dir, char caminho[256], char tipo[10])
```

Função responsável pela criação de novas pastas/arquivos.

```
if(dir == NULL){ // Retornando para a main caso dir for nulo
    return;
}
if(strcmp(dir->tipo, "") != 0){ // Verificando se não é uma pasta
    printf("Retornando, tentando criar dentro do arquivo\n");
    return;
}
Diretorio* novo = (Diretorio*) malloc(sizeof(Diretorio));
```

Verifico se o diretório é nulo, e se o usuário está tentando criar um novo arquivo/pasta dentro de um arquivo. Caso um deles for validado, retorno para a função.

```
char caminhoPai[256]; // Criando string para armazenar o caminho do diretorio pai
strcpy(caminhoPai, dir->caminho);
if(strcmp(tipo, "") != 0){ // Se for um arquivo:
    strcpy(novo->tipo, tipo); // Tipo recebe o tipo
    strcpy(novo->caminho, strcat(strcat(strcat(strcat(caminhoPai, "/"), caminho), "."), tipo));
}else {
    strcpy(novo->tipo, ""); // Definindo o tipo da pasta como "", ou seja, vazio, pois é uma pas
    strcpy(novo->caminho, strcat(strcat(caminhoPai, "/"), caminho)); // Definindo o caminho da n
}

novo->irmao = NULL; //Definindo os ponteiros do novo arquivo/pasta como NULL
novo->filho = NULL;
```

Crio uma string (caminhoPai) que irá receber o caminho do pai, afim de realizar manipulações com ele.

Verifico se o tipo do arquivo desejado é diferente de “”, se for, significa que se trata de um novo arquivo. Declaro o tipo como a string “tipo” passada por parâmetro e defino o caminho.

Se não for diferente de “”, se trata de uma nova pasta. Declaro o tipo como “”, e o novo caminho.

Aponto os ponteiros do novo arquivo/pasta como nulos.

```
Diretorio* aux = dir->filho; // Criando um diretório aux para percorrer até
while (aux != NULL) {
    if (strcmp(aux->caminho, novo->caminho) == 0) {
        printf("Pasta ou arquivo j existente!\n");
        free(novo); // Libera a mem ria alocada para o novo diret rio
        return;
    }
    aux = aux->irmao;
}
```

Crio um Diretorio aux para percorrer entre os irmãos, e verificar se a pasta/arquivo já existe, caso existir, imprimo o erro, libero a memória e retorno para a main.

```
if(dir->filho == NULL){ // Se o novo arquivo/pas
    dir->filho = novo;
} else {
    Diretorio* aux = dir->filho;
    while(aux->irmao != NULL){ // Percorrendo pa
        aux = aux->irmao;
    }
    aux->irmao = novo;
}
return;
```

Verifico se o filho do diretório pai é nulo, e caso for, defino o novo elemento como filho do diretório pai,

Caso contrário, percorro todos os irmãos, afim de chegar no ultimo e realizar o apontamento de ultimo->irmão = novo.

```
void help(){ // Função que imprime todos os comandos disponíveis no sistema
    printf(
            "Comandos dispon\ufffdveis:\n"
            "- cd     : entrar em uma pasta\n"
            "- search : busca uma pasta ou arquivo\n"
            "- rm     : remover uma pasta\n"
            "- list   : lista os componentes dentro da pasta em questao\n"
            "- mkdir  : cria uma nova pasta\n"
            "- clear  : limpa o conteudo da tela\n"
            "- help   : exibe a relacao completa dos comandos\n"
            "- exit   : fechar o programa\n"
            "- ..     : retornar\n"
    );
}
```

Função que imprime todos os comandos existentes.

```
char cortarString(char *str, int indice) { //
    if (indice >= strlen(str)) {
        return ' ';  // \ufffdndice fora dos limit
    }
    strcpy(str, str + indice);
}
```

Função que corta a string recebida por parâmetro (diretamente na memória), no índice informado pelo usuário, que foi informado passado por parâmetro.

```
int printar_nome(char caminho[265], int pos){ /
    if(caminho[pos] == '/'){
        return pos;
    }else {
        pos = printar_nome(caminho, pos-1);
    }
    return pos;
}
```

Função recursiva que retorna a posição exata da ultima barra existente no diretório. Quando utilizada em conjunto com corta string, dentro da função list, é possível imprimir os nomes das pastas e arquivos.

```
void list(Diretorio* dir, int modo, char caminho[256]){
```

Função para listar os diretórios disponíveis dentro de uma pasta.

No modo 1, ela apenas imprime todos os diretórios.

Em outros modos (!= 1), ela verifica se o começo da string que guarda os diretórios é igual ao caminho passado, e se for, informa como um diretório possível, afim de auxiliar o usuário a navegar pelos diretórios. É chamada apenas pela função “cd”.

```
if(strcmp(dir->tipo, "") != 0){ // Veri
    printf("Diretorio invalido!\n");
    return;
}
if(dir->filho == NULL){ // Se o filho f
    printf("Diretorio vazio\n");
    return;
}
```

Verifico se o usuário está tentando listar os diretórios disponíveis dentro de um arquivo, caso sim, informo que não é um diretório válido e retorno.

Verifico se o diretório passado por parâmetro possui um filho(outros diretório(s)), e caso não possua, informo que o retorno está vazio e retorno.

```
Diretorio* aux = dir->filho;
char nome[256] = "";
int printados = 0;
```

Crio um diretório auxiliar para percorrer nos próximos passos.

Crio uma string nome, que irá receber o caminho dos diretórios

Crio uma variável do tipo inteiro, que recebe +1, toda vez que um diretório é printado fora do modo 1.

```
do{
    strcpy(nome, aux->caminho);
    cortarString(nome, printar_nome(aux->caminho, strlen(aux->caminho) - 1));
    if(modo == 1){ // Se for modo 1, apenas printa
        printf("%s\n", nome);
```

Abro um do while, e passo o caminho de aux para nome. Chamo a função cortarString, que irá receber no final, apenas o nome do arquivo com a barra.

Se o modo for igual a 1, apenas printo o nome(caminho).

```
}else{ // Se não for, faço as verificações para cada diretório,
    int igual = 0;
    for(int i = 0; i < strlen(caminho); i++){ // Verifico se ca
        if(caminho[i] != nome[i+1]){
            igual = 1; // Se for diferente, vai receber 1
        }
    }
```

Caso o modo não for o 1, crio um int igual, que irá receber 0.

Dentro de um for, verifico se o caminho informado pelo usuário no parâmetro, é igual ao caminho do atual diretório (aux). Caso algum elemento for diferente, igual receberá 1.

```
if(igual == 0){ // Printo as possiveis alternativ
    if(printados==0){ // Se o numero de alternati
        printf("Possiveis alternativas:\n");
    }
    printf("%s\n", nome); // Printo o nome do dir
    printados++; // Printados recebe mais um
}
```

Após percorrer toda o nome do arquivo que foi passado pelo usuário, verifico se é igual (igual == 0), e se for, verifico se printados é igual a 0, caso sim, significa que ainda não foi printado nenhum diretório, e imprimo a frase auxiliar (“Possiveis alternativas:”), e logo em seguida imprimo o caminho do possível diretório que o usuário poderá escolher.

```
    aux = aux->irmao;
}while(aux != NULL);
```

Isso é feito para todos os diretórios dentro do diretório pai, caminhando entre os irmãos.

```
if(modo != 1 && printados == 0){ // Se chega
    printf("Diretorio nao encontrado!\n"); /
}
```

Após de percorrer todos os diretórios, caso não esteja no modo 1, e o numero de elementos printados for 0, informo que não foi encontrado nenhum diretório.

```c
void rm_recursiva(Diretorio* dir){ //
    if(dir->irmao != NULL){ // Caminho
        rm_recursiva(dir->irmao);
    }
    if(dir->filho != NULL){ // Caminho
        rm_recursiva(dir->filho);
    }
    free(dir); // Libero a memória
    return;
}
```

Função recursiva para remover todos os irmãos, e o próprio elemento do diretório passado por parâmetro. Primeiro a função percorre até chegar no ultimo elemento (folha), e depois vem eliminando os elementos.

```c
void rm(Diretorio* dir, char nome[256]){
```

Função responsável por eliminar um arquivo/pasta.

```c
Diretorio* filho = dir->filho;
Diretorio* ant = filho; // Crio um diretorio ant p
char diretorioPai[256]; // Crio uma string para re
strcpy(diretorioPai, dir->caminho);
nome = strcat(strcat(diretorioPai, "/"), nome);
```

Declaro um novo diretório filho como o filho do diretório pai, e declaro um novo diretório ant, e igualando ele à filho. Crio uma nova string diretórioPai, que recebe o caminho completo concatenando o nome do arquivo passado por parâmetro.

```c
while(filho != NULL && strcmp(filho->caminho, nome) != 0){ //
    ant = filho;
    filho = filho->irmao;
}
```

Percorro todos os irmãos, afim de encontrar o diretório solicitado pelo usuário.

```c
if(filho == NULL){ // Se filho for null, info
    printf("Diretorio nao encontrado!\n");
    return;
}
```

Se ao final do while, filho for nulo, significa que não foi encontrado o diretório solicitado, informo o erro e retorno.

```c
if(dir->filho == filho){ // Se filho
    ant = filho->irmao; // Anterior
    dir->filho = ant; // Filho do di
}else{
    ant->irmao = filho->irmao; // Ir
}
```

Se o filho do diretório filho, for igual ao filho(ultimo irmão do filho do diretório pai), declaro ant como o irmão do filho, e o novo filho do diretório pai igual a ant.

Caso contrário, declaro o irmão de ant como o irmão do filho.

Os processos acima são a jogada de ponteiros, para reorganiza-los após a exclusão de um diretório principal.

```
if(filho->filho != NULL){ // Se
    rm_recursiva(filho->filho);
}
free(filho); // Libero o filho
```

Se o filho do filho for nulo, apenas libero a sua memória. Caso contrário, chamo a função de remover recursiva, que irá apagar todos os diretórios que estão dentro dele, e ao final, libero a memoria.

```
void search(Diretorio* dir, char nome[256]){
```

Função que busca um arquivo/pasta que existe no diretório atual.

```
while(filho  != NULL && strcmp(filho->caminho, nome) != 0){
    filho = filho->irmao;
}
```

Procuro um filho que tenha um caminho correspondente ao que o usuário inseriu.

```
if(filho == NULL){ // Se filho for nulo, informo que não encontr
    printf("Diretorio nao encontrado!\n");
    return;
}else {
    if(strcmp(filho->tipo, "") == 0){ // Se o tipo do elemento e
        printf("Pasta encontrada!\n"
               "Caminho: %s\n", filho->caminho);
    }else { // Se o tipo do elemento encontrato não for nulo, in
        printf("Arquivo %s encontrado!\n"
               "Caminho: %s\n", filho->tipo ,filho->caminho);
    }
}
```

Se não encontrar print que não existe a pasta/arquivo que o usuário informou. Caso existe eu verifico se é uma pasta ou um arquivo e print as informações relevantes de cada um

```
void executar(Diretorio* dir){
```

Função recursiva que serve como uma 'main', ela é responsável por andar pela árvore. Ela sempre está no diretório atual do usuário e, por ser recursiva, consegue voltar para seu diretório pai.

```
while(1){
    printf("%s ", dir->caminho);
    char comando[256] = "";
    char comando_interno[256] ="";
    int pos;
    scanf(" %[^\n]", comando);
    int i;
    for(i = 0; i < 256; i++){
        if(comando[i] != ' '){
            comando_interno[i] = comando[i];
        }else{
            cortarString(comando, i+1);
            break;
        }
    }
```

A execução fica dentro de um while, pois o usuário pode digitar o comando errado. Portanto, caso isso aconteça, eu preciso continuar pedindo a ele um novo comando. O comando é escaneado e dividido entre comando e comando interno. O comando interno identifica os comandos do programa, como cd, mkdir e help, e o comando é o restante da string. Há uma verificação para saber se o usuário digitou apenas um comando interno ou também digitou algo diferente, que é guardado usando a função cortarString.

```
switch (verificarComando(comando_interno)) {
```

Então, usando um switch, identificamos o comando interno que o usuário inseriu e chamamos sua função correspondente.

```
case 1:
    novoDiretorio = cd(dir, comando);
    if(novoDiretorio != NULL) {
        executar(novoDiretorio);
    }
    break;
```

Case 1: Entra em um diretório ou arquivo existente e chama a função 'executar' passando esse diretório como parâmetro. Dessa maneira, você consegue navegar pelos diretórios e voltar, se necessário.

```
case 2:
    search(dir, comando);
    break;
```

Case 2: Procuramos uma pasta/arquivo existente no diretório atual usando a função 'Search'.

```
case 3:
    rm(dir, comando);
    break;
```

Case 3: Deletamos um arquivo/pasta passado. Caso seja uma pasta e possua filhos, deletamos todos os seus filhos e os filhos dos filhos, e assim por diante, até liberar todos os descendentes desse diretório atual. Dessa maneira, evitamos que a memória fique cheia, pois ao deletar uma pasta, tudo que existe dentro dela será deletado em conjunto.

```
case 4:
    list(dir, 0, "");
    break;
```

Case 4: Lista todos os arquivos/pastas existentes no diretório atual.

```
case 5:
    printf("Voce deseja criar uma pasta ou um arquivo?\n"
           "1 - Arquivo\n"
           "2 - Pasta\n"
           "Escolha: ");
    scanf("%d", &i);
    if(i == 1){
        printf("Informe o tipo: ");
        scanf(" %[^\n]", tipo);
        printf("Informe o nome do arquivo: ");
        scanf(" %[^\n]", caminho);
        mkdir(dir, caminho, tipo);
    }else if(i == 2){
        strcpy(tipo, "");
        printf("Informe o nome da pasta: ");
        scanf(" %[^\n]", caminho);
        mkdir(dir, caminho, "");
    }else {
        printf("Comando invalido!\n");
    }
    break;
```

Case 5: Criação de diretórios e arquivos. Nesse comando, é possível criar tanto pastas como arquivos. O usuário escreve apenas "mkdir" e, em seguida, perguntamos se ele deseja criar um arquivo ou uma pasta. Caso ele queira criar um arquivo, nós também guardamos o tipo dele. Depois, utilizamos essa informação para identificar se o nó atual em que estou é uma pasta ou um arquivo, pois não é possível criar pastas/arquivos dentro de um arquivo, apenas em pastas.

```
case 6:
    system("cls");
    break;
```

Case 6: Limpa o terminal

```
case 7:
    help();
    break;
```

Case 7: chama a função help para informar os comandos que existe no programa.

```
case 8:
    rm(dir, dir->caminho);
    system("cls");
    exit(1);
    break;
```

Case 8: Fecha o programa, mas antes libera toda a árvore, desde a raiz até o último descendente.

```
case 9:
    return;
    break;
```

Case 9: Volta para a Raiz anterior

```
case 0:
    printf("Comando invalido! Utilize 'help' para obter os comandos.'\n");
```

Case 0: Caso o usuário inseriu um comando invalido

```
int verificarComando(char comando[256]){
```

A função "verificarComando" recebe o comando que o usuário inseriu e retorna um valor inteiro, que será usado no switch. Isso nos permite identificar qual operação o usuário deseja realizar.

```
Diretorio* cd(Diretorio* dir, char nome[256]) {
```

A função "cd" é responsável por navegar pela árvore. Ela recebe o diretório pai e o nome do diretório para o qual o usuário deseja entrar.

```
while (filho != NULL && strcmp(filho->caminho, nome) != 0) {
    filho = filho->irmao;
}
```

Dentro da função "cd", você pode percorrer os filhos que estão em uma lista encadeada usando um loop while. Verifique se o caminho de cada filho é igual ao caminho fornecido pelo usuário. Se for o caso, retorne o diretório encontrado. Caso contrário, liste os diretórios existentes e retorne o diretório pai, imprimindo que o diretório não foi encontrado.

```c
void ler_txt(Diretorio* dirPai, const char* nomeArquivo) {
```

Função responsável por ler o arquivo txt e criar os diretórios e subdiretórios.

```c
while (fgets(linha, sizeof(linha), arquivo)) {
    // Remover o caractere de quebra de linha no final da linha
    linha[strcspn(linha, "\n")] = '\0';

    char* token = strtok(linha, "/");
    Diretorio* dirAtual = dirPai;
```

Ela funciona inserindo linha por linha do txt. Então, eu inicio um while que possui o diretório pai de todos, que é o "C:", e substituo as quebras de linha por '\0' para conseguir identificar que eu cheguei ao final de uma linha do arquivo. Em seguida, utilizo a função strtok para guardar a string até um delimitador, que é a barra "/". Por exemplo, se tivermos "Meus Documentos/Fontes", ele irá guardar a string até encontrar a barra, resultando em "Meus Documentos

```c
char caminho_atual[256] = "";
strcat(caminho_atual, "C:");
while (token != NULL) {
    // Verificar se o token já existe como filho do diretório atual
    int encontrado = 0;
    Diretorio* filho = dirAtual->filho;
    strcat(caminho_atual, "/");
    strcat(caminho_atual, token);

    while (filho != NULL) {
        if (strcmp(filho->caminho,caminho_atual) == 0) {
            encontrado = 1;
            break;
        }
        filho = filho->irmao;
    }
```

Então, eu começo um segundo while que é responsável pela criação dos diretórios. Primeiro, verifico se existe um diretório com o caminho especificado. Se existir, eu entro nele; caso contrário, crio um novo diretório. Durante esse processo, vou guardando o caminho atual para a verificação dos subdiretórios, já que o programa sempre mantém o caminho completo.

Repito esse processo até o final da linha e, em seguida, passo para a próxima linha do arquivo de texto. Ao fazer isso, volto para o diretório pai, que é o "C:", e repito o procedimento, sempre verificando se o diretório já existe para evitar duplicatas. No final, fecho o arquivo, volto para a função principal (main) e chamo a função "executar()". Agora, quando a função "executar()" for iniciada, esses diretórios já estarão criados.

Principais Desafios e dificuldades:

Dificuldades na Manipulação de Strings em C:

Durante o desenvolvimento do projeto, nos deparamos com várias dificuldades relacionadas à manipulação de strings em C. É importante destacar que a linguagem C apresenta limitações significativas nesse aspecto, o que exigiu um esforço adicional para superar essas restrições. Algumas das principais dificuldades encontradas incluem:

Funções Limitadas: A linguagem C oferece um conjunto limitado de funções embutidas para manipulação de strings. Isso significa que tarefas básicas, como concatenação, busca e substituição de substrings, exigem implementações personalizadas e cuidadosas para obter os resultados desejados.

Tamanho Fixo de Strings: Em C, o tamanho das strings é fixo e precisa ser definido antecipadamente. Isso pode ser problemático quando não sabemos com antecedência o tamanho máximo que uma string pode ter, levando a possíveis estouros de buffer e problemas de alocação de memória.

Ausência de Tipo String: Ao contrário de algumas linguagens modernas, como Python ou JavaScript, C não possui um tipo de dado "string" nativo. As strings são tratadas como arrays de caracteres, o que torna as operações de manipulação mais verbosas e suscetíveis a erros.

Desafio na Inserção de Documento TXT:

Além das dificuldades específicas na manipulação de strings, encontramos um desafio adicional ao tentar inserir um documento de texto no código. Uma vez que o projeto foi concebido com interação com o usuário em mente, a inserção do documento de texto de forma adequada se tornou uma tarefa complicada.