SISTEMA DE GERENCIAMENTO DE TAREFAS

AEDS II- Rafael Ribeiro Brant Nobre

RESUMO

- "O objetivo deste trabalho foi desenvolver um sistema de gerenciamento de tarefas utilizando estruturas de dados avançadas, **como filas, pilhas e árvores AVL,** e implementar algoritmos de ordenação e pesquisa. Este sistema busca organizar, priorizar e facilitar a busca de tarefas de forma eficiente e balanceada

A motivação para o uso de cada uma dessas estrutura de dados foi:

- **Árvore AVL** foi usada para armazenar a estrutura **tarefa**
- **Filas** foram utilizadas para ordenar a prioridade das **tarefas**
- **Pilhas** foram utilizadas como ‘’backup” das **tarefas**

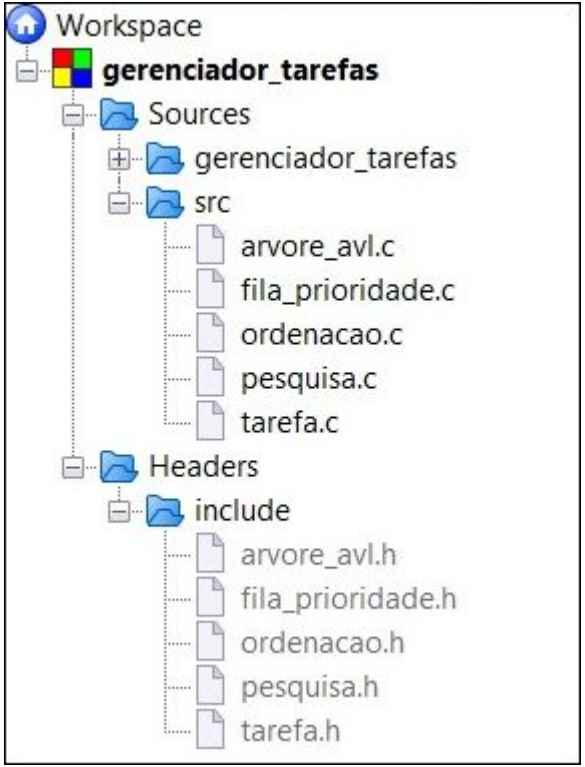
Além disso, foi utilizado algoritmos de **busca binária e ordenação**, como o MergeSort e QuickSort, para ordenar tarefas pelas suas devidas estruturas de dados.

01.

ARQUIVOS E ESTRUTURA PRINCIPAL

ARQUIVOS

A organização do código em vários arquivos foi feita para garantir **modularidade, manutenibilidade, escalabilidade e organização.** Em projetos maiores, essa abordagem permite separar responsabilidades e facilita o desenvolvimento, depuração e evolução do sistema.



ARQUIVOS

01.

MODULARIDADE

dividir o sistema em partes menores e independentes

02.

MANUTENÇÃO

erros podem ser corrigidos em uma única parte do sistema

03.

MAIN.C ISOLADO

mantém a interface separada da implementação..

04.

ESCALABILIDADE

pode ser facilmente expandido com novas funcionalidades ou modificações

MAIN.C

O arquivo main.c atua como o ponto central de controle do sistema, gerenciando a interação do usuário com os módulos de filas, árvores AVL e algoritmos de ordenação e pesquisa.

```c
int main() {
    Tarefa *tarefas = NULL;
    int total_tarefas = 0;
    NoAVL *arvore_ids = NULL;
    Pilha *pilha = criar_pilha();
    int prox_id = 1;

    int opcao;
    do {
        printf("Escolha uma opção:\n");
        printf("1. Inserir nova tarefa\n");
        printf("2. Exibir tarefas ordenadas por ID\n");
        printf("3. Exibir tarefas ordenadas por data de criação\n");
        printf("4. Exibir tarefas ordenadas por prioridade\n");
        printf("5. Salvar tarefas em arquivo\n");
        printf("6. Visualizar dados do arquivo\n");
        printf("7. Excluir tarefa e salvar na pilha\n");
        printf("8. Restaurar tarefa da pilha\n");
        printf("9. Sair\n");
        scanf("%d", &opcao);

        switch (opcao) {
```

MAIN.C

O main.c foi projetado utilizando a estrutura condicional “switch case”, o usuário escolhe valores discretos em um determinado intervalo para selecionar as seguintes opções:

```
int opcao;
do {
    printf("Escolha uma opção:\n");
    printf("1. Inserir nova tarefa\n");
    printf("2. Exibir tarefas ordenadas por ID\n");
    printf("3. Exibir tarefas ordenadas por data de criação\n");
    printf("4. Exibir tarefas ordenadas por prioridade\n");
    printf("5. Salvar tarefas em arquivo\n");
    printf("6. Visualizar dados do arquivo\n");
    printf("7. Excluir tarefa e salvar na pilha\n");
    printf("8. Restaurar tarefa da pilha\n");
```

02 TAREFAS, FILAS E PILHAS

TAREFA.C

Aqui é onde foi definida a estrutura principal. Ela carrega quatro tipo de dados e ponteiros para armazenar os dados, essas escolhas facilitam a organização.

```c
#ifndef TAREFA_H
#define TAREFA_H

typedef struct{
int id;
char descricao[100];
int prioridade;
char data_criacao[11];
}Tarefa;

Tarefa *criar_tarefa();
void imprimir_tarefa(Tarefa *tarefa);
void imprimir_lista_tarefas(Tarefa *tarefas[], int num_tarefas);
Tarefa* copiar_tarefa(Tarefa *original);
```

TAREFA

Para este projeto, foi fundamental utilizar de um struct como estrutura de dados principal, pois cada tarefa armazena 4 tipos de dados diferentes. Além disso, o arquivo conta com as principais funções relacionadas a essa estrutura. Tais como, criar, imprimir e copiar.

```c
#ifndef TAREFA_H
#define TAREFA_H

typedef struct{
int id;
char descricao[100];
int prioridade;
char data_criacao[11];
}Tarefa;

Tarefa *criar_tarefa();
void imprimir_tarefa(Tarefa *tarefa);
void imprimir_lista_tarefas(Tarefa *tarefas[]
Tarefa* copiar_tarefa(Tarefa *original);
```

FILA.C

aqui é onde está a estrutura da fila de prioridade. As funções da fila são projetadas para criar, organizar, remover e adicionar tarefas como dados na fila.

```c
#include "tarefa.h"


typedef struct no_fila {
    Tarefa *tarefa;
    struct no_fila *prox;
} NoFila;


typedef struct {
    NoFila *inicio;
    NoFila *fim;
} Fila;


Fila *criar_fila();
void inserir_na_fila(Fila *f, Tarefa *tarefa);
Tarefa *retirar_da_fila(Fila *f);
int verificar_fila_vazia(Fila *f);
void liberar_fila(Fila *f);
```

FILA

Para fazer um sistema de gerenciamento de prioridade, é fundamental essa estrutura de dados. Pois ela permite ordenar a cada tarefa baseado na ordem em que são inseridos. No caso desse programa, a ordem de inserção na fila está intimamente relacionada com a prioridade atribuída pelo usuário

```c
#include "tarefa.h"


typedef struct no_fila {
    Tarefa *tarefa;
    struct no_fila *prox;
} NoFila;


typedef struct {
    NoFila *inicio;
    NoFila *fim;
} Fila;


Fila *criar_fila();
void inserir_na_fila(Fila *f, Tarefa *tarefa);
Tarefa *retirar_da_fila(Fila *f);
int verificar_fila_vazia(Fila *f);
void liberar_fila(Fila *f);
```

PILHA

Os algoritmos de pilha e suas funções servem neste código como “backup” para as tarefas excluídas pelo usuário. Fornecendo a opção de restaurá-las para as estruturas principais

```c
// Estrutura de um nó da pilha
typedef struct NoPilha {
    void *dado;              // Ponteiro genérico para armazena
    struct NoPilha *proximo; // Próximo elemento da pilha
} NoPilha;

// Estrutura principal da pilha
typedef struct Pilha {
    NoPilha *topo; // Ponteiro para o topo da pilha
    int tamanho;   // Quantidade de elementos na pilha
} Pilha;

// Função para criar uma nova pilha
Pilha *criar_pilha() {
    Pilha *pilha = (Pilha *)malloc(sizeof(Pilha));
    if (pilha == NULL) {
        printf("Erro ao alocar memória pa
        exit(1);
    }
    pilha->topo = NULL;
    pilha->tamanho = 0;
    return pilha;
}
```

```
class Pilha {...}
typedef Pilha Pilha
```

PILHA

Um sistema de backup e histórico de exclusões funciona perfeitamente aplicando uma pilha. Já que, devido a natureza da pilha de ordenar dando preferência das primeiras posições aos últimos elementos inseridos, à medida que as tarefas são excluídas, ficam mais distantes da restauração da pilha.

```c
// Estrutura de um nó da pilha
typedef struct NoPilha {
    void *dado;              // Ponteiro genérico para armazena
    struct NoPilha *proximo; // Próximo elemento da pilha
} NoPilha;

// Estrutura principal da pilha
typedef struct Pilha {
    NoPilha *topo; // Ponteiro para o topo da pilha
    int tamanho;   // Quantidade de elementos na pilha
} Pilha;

// Função para criar uma nova pilha
Pilha *criar_pilha() {
    Pilha *pilha = (Pilha *)malloc(sizeof(Pilha));
    if (pilha == NULL) {
        printf("Erro ao alocar memória pa
        exit(1);
    }
    pilha->topo = NULL;
    pilha->tamanho = 0;
    return pilha;
}
```

```
class Pilha {...}
typedef Pilha Pilha
```

03 ÁRVORE AVL

ARVORE_AVL.C

aqui é onde está a estrutura da árvore avl. A árvore avl é a estrutura principal para o armazenamento e busca das tarefas.

```c
#include "tarefa.h"

// Estrutura de um nó da árvore AVL
typedef struct NoAVL {
    Tarefa *tarefa;
    struct NoAVL *esquerda;
    struct NoAVL *direita;
    int altura;
} NoAVL;

// Função para criar um novo nó da árvore AVL
NoAVL *criar_no(Tarefa *tarefa);

// Funções de inserção específicas para cada critério de ordenação
NoAVL *inserir_na_arvore_por_id(NoAVL *raiz, Tarefa *tarefa);
```

ÁRVORE AVL

A árvore AVL é uma estrutura que se encaixa perfeitamente com o armazenamento dos IDs. Uma vez que cada ID é único e serve como nó da árvore. Além disso, usar a estrutura de uma árvore avl é importante pois as rotações impedem que os IDs fiquem armazenados de forma sequencial, como em uma lista ou vetor. O que facilita a busca

```c
#include "tarefa.h"

// Estrutura de um nó da árvore AVL
typedef struct NoAVL {
    Tarefa *tarefa;
    struct NoAVL *esquerda;
    struct NoAVL *direita;
    int altura;
} NoAVL;
```

04

ORDENAÇÃO E PESQUISA

ORDENAÇÃO

os algoritmos de ordenação principal é o QuickSort MergeSort. Ambos são bons para lidar com grande números de dados e possuem baixa complexidade

```
#ifndef ORDENACAO_H
#define ORDENACAO_H

#include "tarefa.h"



int comparar_data(const char *data1, const char *data2);
void merge(Tarefa *tarefas, int esquerda, int meio, int dire
void mergeSort(Tarefa *tarefas, int esquerda, int direita, i
void ordenar_por_data(Tarefa *tarefas, int n);
void ordenar_por_prioridade(Tarefa *tarefas, int n);
int particao(Tarefa *tarefas, int esquerda, int direita);
void quickSort(Tarefa *tarefas, int esquerda, int direita);
#endif // ORDENACAO_H
```

ORDENAÇÃO

Os algoritmos de ordenação escolhidos, QuickSort e InsertionSort, foram selecionados por sua eficiência e adequação às diferentes necessidades do sistema. O QuickSort é ideal para ordenar grandes volumes de dados devido ao seu desempenho médio de O(nlogn)O(n \log n)O(nlogn), garantindo rapidez na organização de tarefas por data ou prioridade. Já o InsertionSort foi escolhido pela simplicidade e bom desempenho em conjuntos menores de dados, como tarefas armazenadas em memória local. A combinação desses algoritmos permitiu balancear eficiência e simplicidade, dependendo do tamanho e complexidade do conjunto de dados a ser ordenado.

PESQUISA

A busca binária é uma abordagem eficiente quando o array está ordenado, porque ela divide o conjunto de busca pela metade em cada etapa, o que reduz significativamente o tempo de busca em comparação com uma busca linear.

```c
int *null;

Tarefa *buscar_tarefa_por_id(Tarefa *tarefas[], int size, int id) {
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (tarefas[mid]->id == id) {
            return tarefas[mid];
        } else if (tarefas[mid]->id < id) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return null; // Não encontrado
}
```

PESQUISA

A pesquisa binária foi escolhida por sua eficiência em conjuntos de dados ordenados, com complexidade O(logn)O(\log n)O(logn). Este algoritmo é ideal para localizar rapidamente tarefas específicas no sistema, como buscar por ID, pois reduz pela metade o espaço de busca a cada iteração. Sua implementação no sistema aproveita as tarefas previamente ordenadas, garantindo agilidade na recuperação de informações, especialmente em listas grandes, onde métodos mais simples, como a busca linear, seriam menos eficientes

CONCLUSÃO

Principais Pontos do Projeto

1. **Estruturas de Dados**
   - A árvore AVL foi utilizada para garantir um armazenamento balanceado das tarefas, permitindo inserções, buscas e exclusões com eficiência O(logn)O(\log n)O(logn).
   - A fila de prioridade organiza as tarefas com base em níveis de prioridade (alta, média e baixa), garantindo que as tarefas mais importantes sejam processadas primeiro.
   - A pilha foi implementada para armazenar um histórico de tarefas excluídas, possibilitando a restauração de uma tarefa, caso necessário.

CONCLUSÃO

Funcionalidades Principais

- Inserção de Tarefas: As tarefas são criadas com um ID único, descrição, data de criação e prioridade. Elas são simultaneamente inseridas na árvore AVL, na fila de prioridade e armazenadas em um vetor para futuras ordenações.
- Exclusão e Restauração de Tarefas: A exclusão de tarefas remove os dados da árvore AVL e salva uma cópia na pilha. A restauração reintegra a tarefa às estruturas, mantendo a consistência.
- Ordenação e Busca:
    - Algoritmos como QuickSort são utilizados para ordenar as tarefas por prioridade ou data de criação.
    - Algoritmos de busca, como a pesquisa binária e linear, foram aplicados para localizar tarefas específicas.

CONCLUSÃO

Organização do Código

- O sistema foi modularizado em múltiplos arquivos, incluindo cabeçalhos (.h) e implementações (.c). Isso promoveu a separação de responsabilidades, facilitou o entendimento do código e possibilitou sua manutenção.
- O arquivo main.c foi desenvolvido como o ponto de entrada do sistema, gerenciando o fluxo de interação com o usuário e delegando operações às funções específicas de cada módulo.

Desafios Superados

- A manipulação das tarefas entre as diferentes estruturas exigiu atenção ao gerenciamento de memória, especialmente na cópia e armazenamento de dados.
- Problemas relacionados à consistência dos IDs e ao fluxo de restauração de tarefas foram identificados e corrigidos, garantindo que o sistema funcione de maneira confiável.
- A implementação de algoritmos eficientes de ordenação e balanceamento, como os utilizados na árvore AVL e no QuickSort, foi essencial para atingir o desempenho esperado.

CONCLUSÃO

O que poderia ser melhor?

- A ordenação por data e prioridade eventualmente apresentou erros na execução.
- o código não possui interface amigável e nem sempre imprime os dados de forma fácil de visualizar.
- Há escolha de linguagem de programação, software e compilador não foi a melhor possível. Projetos grandes de várias linhas de código podem exigir de muita complexidade dos algoritmos.
- Gostaria de ter trabalhado melhor com a manipulação de arquivos, envolvendo um sistema de adicionar e excluir do arquivo, o que tornaria o programa de fato um ‘’programa’’, por não ser uma exigência direta, deixei de lado.
- Gostaria de ter divido em mais arquivos o programa para facilitar a organização, porém, ao enfrentar alguns problemas com o compilador, desisti e acabei aumentando o código do arquivo main.c

OBRIGADO!