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

Trabalho 1 - AEDS 1

Sistema de gerenciamento de entregas com drones

Luan Barros Reis [6576]
Thomaz Augusto Araújo Silva [6577]
Yasmim Pereira Delfino[5919]

Florestal - MG
2025

Sumário

1. Introdução **3**

2. Organização **4**

3. Desenvolvimento **5**
3.1 Pacote (pacote.c / pacote.h) 5
3.2 Lista (lista.c / lista.h) 5
3.3 Drone (drone.c / drone.h) 7
3.4 Galpão (galpao.c / galpao.h) 8
3.5 Programa Principal (main.c) 8
3.6 Parte Importante 9

4. Compilação e Execução **10**

5. Resultados **12**

6. Conclusão **12**

7. Referências **13**

8. GitHub **13**

1. Introdução

Este trabalho tem como propósito o desenvolvimento de um sistema de gerenciamento de entregas realizado por drones, tendo como objetivo auxiliar pessoas idosas na compra e recebimento de produtos de farmácias e supermercados. O sistema deve controlar a carga máxima suportada pelos drones, distribuir as entregas de maneira eficiente, calcular a distância total percorrida nas viagens e apresentar as informações detalhadas sobre cada entrega realizada.

Para resolver o problema, foi adotada uma abordagem modular, com a implementação dos Tipos Abstratos de Dados (TADs) **Pacote**, **Lista de Pacotes**, **Drone** e **Galpão**. Cada um deles define atributos e operações específicas, permitindo organizar o sistema de forma clara e estruturada por partes.

Para armazenar os pacotes, utilizamos uma **lista encadeada simples sem nó cabeça [1]**. Apesar de a versão com nó cabeça facilitar certas operações, escolhemos a versão sem justamente porque ela exige mais atenção aos casos especiais (como quando a lista está vazia ou tem apenas um elemento). A ideia foi treinar melhor a lógica e entender cada passo da manipulação da estrutura, em vez de abstrair esses detalhes. Dessa forma, conseguimos acompanhar de perto como as inserções e remoções realmente funcionam “por dentro” da lógica.

A implementação foi realizada em linguagem C, utilizando as bibliotecas padrão `<stdio.h>`, `<stdlib.h>`, `<string.h>` e `<math.h>`. Além disso, utilizamos diretivas de pré-processador, como `#ifndef` e `#define`, para organizar melhor os módulos do sistema e evitar problemas de múltiplas chamadas de arquivo.

---

2. Organização

Na Figura 1, é possível visualizar a estrutura geral do projeto. O diretório foi organizado de forma modular para separar claramente os diferentes componentes e facilitar a manutenção.

- **Pasta `codigo/`**: contém a implementação do projeto. Os arquivos `.h` (cabeçalhos) concentram as declarações de funções e structs, enquanto os arquivos `.c` reúnem as implementações correspondentes.
- **Pasta `testes/`**: armazena arquivos utilizados para validação da implementação.
- **Pasta `documentacao/`**: reúne o relatório do trabalho.

Essa organização garante maior clareza ao projeto, permitindo que cada parte cumpra um papel específico dentro do sistema.

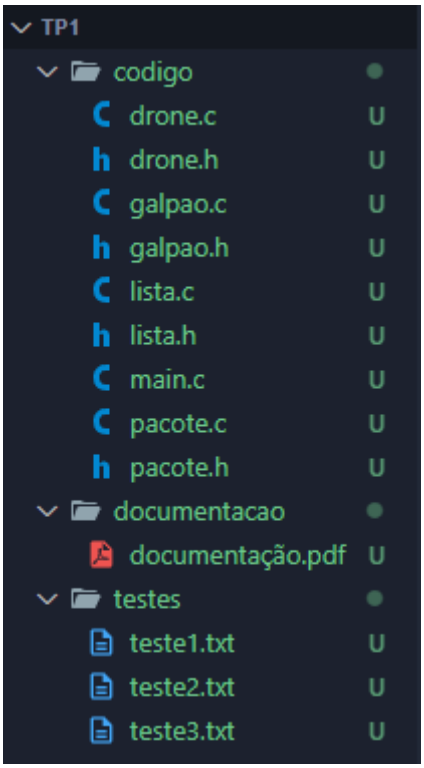


Figura 1 - Repositório do projeto.

3. Desenvolvimento

Nesta etapa são apresentados os principais pontos da implementação, destacando os TADs desenvolvidos e algumas funções determinantes para o funcionamento do projeto. Não serão mostrados todos os detalhes.

3.1 Pacote (pacote.c / pacote.h)

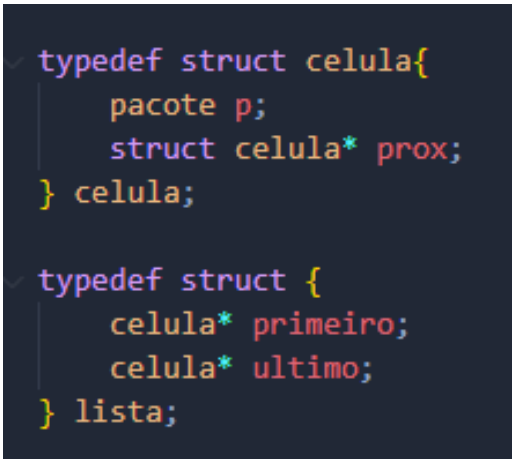
O TAD **Pacote** é responsável por armazenar os dados básicos de uma entrega: conteúdo, destinatário, peso e distância até o destino.

A função principal é a de inicialização:

```c
// inicializa um pacote com conteúdo, destinatário, peso e distância
void inicializarpacote(pacote* p, char* conteudo, char* destinatario,
double peso, double distancia_km) {
    strcpy(p->conteudo, conteudo);
    strcpy(p->destinatario, destinatario);
    p->peso = peso;
    p->distancia_km = distancia_km;
}
```

Além dela, o módulo possui diversos **getters** e **setters** para manipular os atributos do pacote. Esses métodos seguem a mesma estrutura de acesso e atualização, garantindo encapsulamento.

3.2 Lista (lista.c / lista.h)



O TAD **Lista** foi criado para gerenciar coleções de pacotes, tanto no galpão quanto no drone. Ele utiliza alocação dinâmica para permitir a inserção e remoção de pacotes sem tamanho fixo.

Para armazenar os pacotes, foi usada uma **lista encadeada simples sem nó cabeça**, controlada pelos ponteiros para o primeiro e o último elementos, que exigiu um cuidado a mais na implementação.

Um ponto determinante foi a inserção, que exigiu cuidado com alocação de memória:

```
// insere pacote no final da lista
void insercaolista(lista* l, pacote* p){
    celula* new = malloc(sizeof(*new)); //aloca memória dinamicamente
no heap
    if (new == NULL) {
        printf("Memória insuficiente.\n"); //trata falha de alocação
        exit(EXIT_FAILURE);
    }
    new->p = *p; // copia os dados do pacote
    new->prox = NULL; // será o último, próximo é NULL
    if(l->primeiro == NULL){ // lista estava vazia
        l->primeiro = new;
        l->ultimo = new; //ambos apontam para a nova celula
    } else { // lista já tinha elementos
        l->ultimo->prox = new; // adiciona a nova celula no final
        l->ultimo = new; // atualiza o ponteiro do ultimo
    }
}
```

Outro ponto importante é a remoção, que além de retirar o pacote da lista, também libera memória para evitar vazamentos:

```
// remove pacote do início da lista
void remocaolista(lista* l, pacote* p) {
    if (l->primeiro != NULL) { // se houver algo para remover
        celula* c = l->primeiro;
        *p = c->p; // copia os dados do pacote removido
        l->primeiro = c->prox; // atualiza ponteiro para o próximo
        if (l->primeiro == NULL) { // lista ficou vazia
            l->ultimo = NULL; // ponteiros primeiro e ultimo -> NULL
        }
        free(c); // libera a célula antiga
    } else {
        printf("Lista vazia.\n");
    }
}
```

3.3 Drone (drone.c / drone.h)

O TAD **Drone** representa o veículo de entrega, controlando sua capacidade de carga, lista de pacotes carregados e distância total percorrida.

O carregamento de pacotes depende do peso máximo suportado:

```
// tenta carregar um pacote no drone
// retorna 1 se o carregamento foi bem-sucedido, 0 caso contrário
int carregamentopacote(drone* d, pacote* p) {
    // Verifica se o drone suporta o peso adicional do pacote
    if(p->peso + d->peso_carregado <= d->pesoMAX){
        d->peso_carregado += p->peso;    // atualiza o peso carregado
        insercaolista(&(d->l), p);       // Adiciona o pacote à lista
        return 1;  // Sucesso no carregamento
    } else {
        return 0;  // Falha: pacote muito pesado
    }
}
```

A função mais determinante do projeto foi a realizarentrega, pois nela ocorre o cálculo da distância percorrida e a atualização da quilometragem total:

```
// realiza a entrega de todos os pacotes do drone
// retorna a distância total percorrida nesta viagem
double realizarentrega(drone* d, lista* l) {
    inicializarlista(l); // inicia a lista vazia

    celula* c = d->l.primeiro;
    double dist_viagem = 0.0;
    double pos_atual = 0.0;
    while (c != NULL) {
        insercaolista(l, &(c->p)); //insere os pacotes no drone
        dist_viagem += fabs(c->p.distancia_km - pos_atual); // calcula
distância percorrida até o próximo pacote
        pos_atual = c->p.distancia_km;
        c = c->prox;
    }
    dist_viagem += pos_atual; //calcula a volta do drone ao galpão
    d->dist_total += dist_viagem;
    // esvazia o drone
    while (d->l.primeiro != NULL) {
        pacote tmp;
```

```
        remocaolista(&(d->l), &tmp);
    }
    d->peso_carregado = 0;
    return dist_viagem;
}
```

3.4 Galpão (galpao.c / galpao.h)

O TAD **Galpão** representa o local onde os pacotes aguardam para serem carregados no drone. A função central é a de carregamento, que transfere pacotes até atingir o limite de peso do drone:

```
//essa função vai carregar os pacotes no drone até ele atingir o peso máximo
void carregarpacotes(galpao* g, drone* d) {
    celula* c = g->l.primeiro;
    while(c != NULL) {
        pacote p_atual = c->p;
        if(carregamentopacote(d, &p_atual)) { //carrega os pacotes no drone conforme suportado
            pacote temp;
            remocaolista(&(g->l), &temp); // Remove pacote do galpão após ser carregado no drone
            c = g->l.primeiro; //atualiza o ponteiro c
        } else {
            break; // drone atingiu peso máximo
        }
    }
}
```

Essa função garante a integração entre galpão e drone, sendo responsável por manter a lógica de carregamento.

3.5 Programa Principal (main.c)

O arquivo principal integra todos os TADs. Inicialmente, o programa lê os dados de entrada a partir de um arquivo de texto, instanciando o galpão e o drone. Em seguida, os pacotes são adicionados ao galpão.

A lógica principal está no loop de entregas:

```
while(meuGalpao.l.primeiro != NULL) {
        carregarpacotes(&meuGalpao, &meuDrone); // carrega os pacotes
no drone até atingir peso máximo
        lista l;
        inicializarlista(&l);
        double dist_viagem = realizarentrega(&meuDrone, &l); // realiza
a entrega e obtém distância da viagem
        imprimirdrone(&l, dist_viagem, id_viagem); //imprime a viagem e
limpa a lista temporária
        id_viagem++; // viagem 1:, viagem 2: ...
    }
```

Esse trecho garante que, enquanto houver pacotes no galpão, o drone fará viagens sucessivas, entregando os pacotes e acumulando a distância total percorrida no dia.

3.6 Parte Importante

Um ponto importante a ser resolvido durante a implementação foi a relação entre as funções realizarentrega e imprimirdrone. O enunciado exige que, a cada entrega, os pacotes sejam removidos da lista do drone (operação realizarentrega) e somente depois as entregas sejam impressas (imprimirdrone).

O desafio é que, se os pacotes fossem removidos diretamente dentro de realizarentrega, não haveria mais dados para exibir na impressão. Por outro lado, imprimir dentro de realizarentrega quebraria a ordem estabelecida (primeiro calcular/entregar, depois imprimir).

Para resolver isso, optamos por criar uma **lista temporária** no **main**. Durante a chamada de realizarentrega, os pacotes são copiados para essa lista antes de serem removidos do drone. Assim, o cálculo da distância e a limpeza da lista do drone ocorrem corretamente dentro do realizarentrega, e ainda é possível imprimir as informações no passo seguinte, utilizando a lista temporária.

```
pacote tmp;
while(l->primeiro != NULL) {
    remocaolista(l, &tmp);
}
```

Dessa forma, conseguimos cumprir integralmente as instruções do trabalho sem comprometer a clareza do fluxo de execução. Vale ressaltar que a limpeza da lista temporária também foi devidamente tratada dentro do imprimirdrone.

---

4. Compilação e Execução

Para a compilação e execução do projeto, utilizou-se o **sistema operacional Windows 11**, com o compilador **GCC** disponível no pacote *MinGW*.

- A compilação deve ser realizada a partir da pasta raiz do projeto (`TP1/`), com o seguinte comando:
  - `gcc codigo/main.c codigo/pacote.c codigo/lista.c codigo/drone.c codigo/galpao.c -o programa.exe`
- Esse comando gera o arquivo executável **`programa.exe`** diretamente na raiz do projeto.
- A execução do programa pode ser feita com:
  - ./programa.exe

O sistema lê automaticamente os arquivos de entrada localizados na pasta **`testes/`**. Por exemplo, no código principal, a leitura ocorre da seguinte forma:

```
FILE* arq = fopen("testes/teste1.txt", "r");
```

Dessa forma, ao executar o programa, o conteúdo de **`teste1.txt`** será utilizado. Para utilizar outros arquivos de teste (como `teste2.txt` ou `teste3.txt`), basta alterar o nome do arquivo indicado no código ou adaptá-lo conforme necessário.

4.1 Verificação de Memória

Durante o desenvolvimento, foi utilizado o Valgrind, ferramenta de análise de memória do heap, disponível no Linux, para garantir que não houvesse vazamentos de memória dinâmicos no programa.

No Windows 11, a verificação foi feita através do WSL (Windows Subsystem for Linux). O processo de instalação seguiu os seguintes passos:

- wsl --install -d Ubuntu

Após instalar o Ubuntu, foram instalados o compilador C e o Valgrind:

- sudo apt update
- sudo apt upgrade -y
- sudo apt install build-essential valgrind -y

Dentro do WSL, o projeto foi compilado normalmente:

- gcc codigo/main.c codigo/pacote.c codigo/lista.c codigo/drone.c codigo/galpao.c -o programa.exe

E em seguida executado com o Valgrind:

- valgrind --leak-check=full ./programa.exe

A primeira execução revelou problemas de memória:

- ==3264== 440 (264 direct, 176 indirect) bytes in 3 blocks are definitely lost
- ==3264== at 0x4846828: malloc
- ==3264== by 0x109666: insercaolista
- ==3264== by 0x1099B8: realizarentrega
- ==3264== by 0x109443: main

Após ajustes no gerenciamento da lista, a execução passou a não apresentar erros:

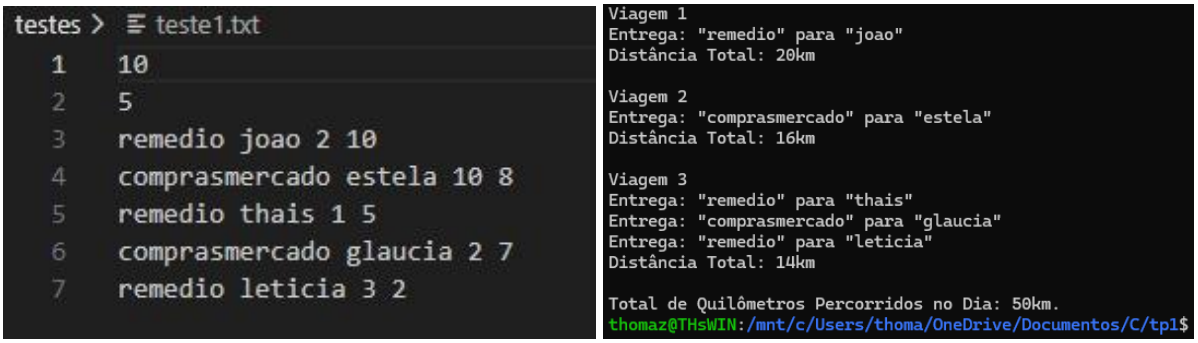
- ==544== HEAP SUMMARY:
- ==544== in use at exit: 0 bytes in 0 blocks
- ==544== total heap usage: 18 allocs, 18 frees, 6,912 bytes allocated
- ==544== All heap blocks were freed -- no leaks are possible
- ==544== ERROR SUMMARY: 0 errors from 0 contexts

Observação: Além da verificação de memória, a execução via WSL também solucionou problemas de codificação de caracteres observados no terminal do Windows. Enquanto no PowerShell alguns acentos eram exibidos de forma incorreta, no ambiente Linux/WSL a saída apareceu exatamente conforme especificado no enunciado, com todos os caracteres acentuados corretamente.

5. Resultados

Para validar o funcionamento do sistema, foram utilizados os arquivos de teste fornecidos no enunciado: `teste1.txt` e `teste2.txt` (além de outros testes tanto iterativos quanto por arquivo realizados pelo grupo)e executados conforme descrito na seção de compilação.

As figuras abaixo mostram a execução com o arquivo `teste1.txt`. O programa carregou corretamente os pacotes respeitando o limite de peso do drone e organizou as viagens:



Além dos testes oficiais, foram criados arquivos adicionais para verificar casos específicos, como pacotes com peso igual ao limite do drone e cenários com apenas um pacote. Em todos os casos, o programa se comportou conforme o esperado, realizando as entregas e acumulando corretamente a quilometragem percorrida.

---

6. Conclusão:

Ao desenvolvermos o sistema de gerenciamento de entregas com drones, pudemos aplicar de forma prática os conceitos de Tipos Abstratos de Dados e listas encadeadas. A modelagem modular, separando Pacote, Lista, Drone e Galpão, trouxe organização ao projeto e possibilitou que cada parte fosse implementada e testada de maneira independente.

Um ponto central foi a escolha pela lista encadeada simples sem nó cabeça, que, embora exigisse maior atenção aos casos especiais, proporcionou um aprendizado mais profundo sobre o funcionamento interno das operações de inserção e remoção.

Outro aspecto relevante foi a solução encontrada para a pendência entre as funções **realizarentrega** e **imprimirdrone**. A criação de uma lista temporária para armazenar os pacotes durante o cálculo da distância e a remoção no drone mostrou-se uma estratégia eficaz, conciliando corretamente as etapas exigidas no modelo do TP1.

Além disso, o uso da ferramenta Valgrind, via WSL no Windows, permitiu validar o gerenciamento de memória do programa, garantindo que todas as alocações dinâmicas fossem devidamente liberadas. Esse processo foi essencial para identificar e corrigir vazamentos nas primeiras versões da implementação, resultando em uma versão final sem erros de memória.

Portanto, conclui-se que os objetivos foram alcançados com sucesso, tanto no aspecto funcional do sistema quanto no aprofundamento dos conceitos de estruturas de dados dinâmicas, modularização e boas práticas de programação em C.

---

7. Referências

[1] SILVA, Thais Regina de Moura Braga. *Notas de Aula de CCF 221 - Algoritmos e Estruturas de Dados I* . Universidade Federal de Viçosa – Campus Florestal, 2025. Disponível no PVANet. Acesso em: 23 set. 2025.

[2] Linguagem C Descomplicada. Disponível em: <https://programacaodescomplicada.wordpress.com/indice/linguagem-c/> Acesso em: 23 set. 2025.

[3] N. Ziviani, Projeto de Algoritmos com Implementações em Java e C++, Editora Thomson, 2007.

---

8. GitHub

https://github.com/essashr/tp1-C