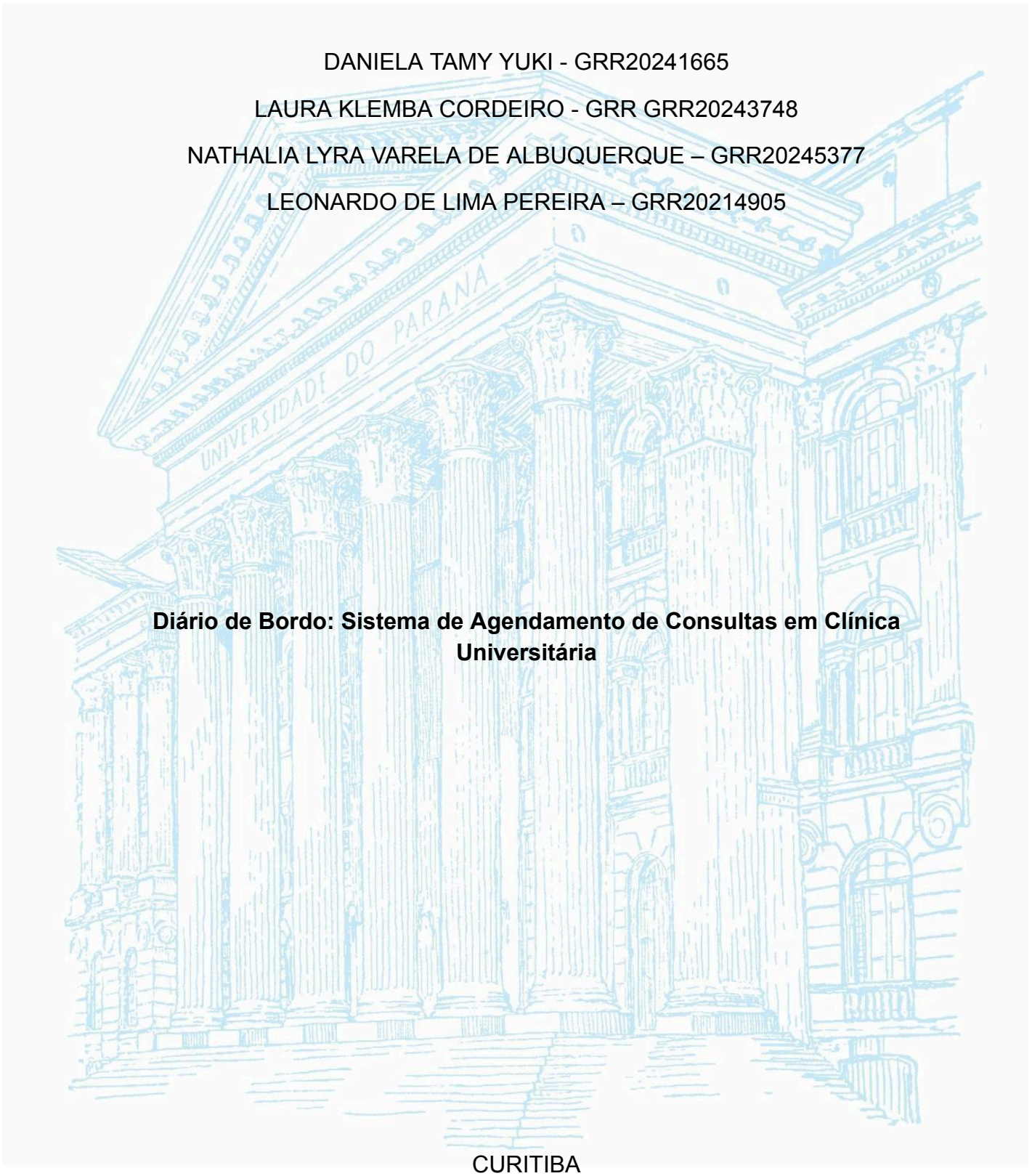
UNIVERSIDADE FEDERAL DO PARANÁ

DANIELA TAMY YUKI - GRR20241665

LAURA KLEMBA CORDEIRO - GRR GRR20243748

NATHALIA LYRA VARELA DE ALBUQUERQUE – GRR20245377

LEONARDO DE LIMA PEREIRA – GRR20214905



Diário de Bordo: Sistema de Agendamento de Consultas em Clínica Universitária

CURITIBA

2025

ESTRUTURA DE DADOS I - DS130

Diário de Bordo: Sistema de Agendamento de Consultas em Clínica Universitária

Tarefa apresentada ao curso Superior Técnico em Análise e Desenvolvimento de Sistemas do Setor de Educação Profissional e Tecnológica da Universidade Federal do Paraná (UFPR), como requisito parcial à obtenção do título de Técnico em Análise e Desenvolvimento de Sistemas.

Professora. Helcio Padilha

CURITIBA

2025

SUMÁRIO

1. Introdução

O projeto “Sistema de Agendamento de Consultas em uma Clínica Universitária” utiliza estrutura de dados lineares implementadas em linguagem C.

Foram criados módulos separados para cada responsabilidade do sistema:

- Lista (lista.c/lista.h): módulo responsável pela implementação das estrutura de dados básicas que são usados nos outros módulos
- Pacientes (pacientes.c/pacientes.h): módulo para cadastro e gerenciamento de pacientes
- Agendamentos (agendamento.c/agendamento.h): módulo para controle de agendamento de consultas
- Menu (main.c): interface interativa do usuário

A lista utilizada para a implementação do módulo Paciente foi a lista simplesmente encadeada devido a sua simplicidade e eficiência, para além de suas características dinâmicas e ao gerenciamento de memória. No caso das características dinâmicas, a lista simplesmente encadeada não possui um array fixo, portanto, ao criar um novo paciente a lista apenas cria um novo nó e aloca na memória.

Já em relação ao gerenciamento de memória, a vantagem está na facilidade de remoção e inserção de nós da lista sem exigir operações complexas, visto que apenas o próximo nó é realocado. Por fim, a desvantagem desse tipo de lista é que é necessário percorrê-la por completo para buscar um elemento. Portanto, não é ideal para grandes volumes de dados.

Já para o módulo agendamento, foi escolhida a lista simplesmente encadeada com header. Essa escolha se deu pois esse módulo realiza maior manipulação dos dados em comparação com a lista de pacientes, visto que está suscetível a inserção, cancelamento e reagendamento de consultas.

Como existe o header nessa lista, isso facilita e torna menos complexa as operações de inserção, busca e remoção já que não é necessário lidar com casos especiais como o null para lista vazia. Embora, esse tipo de lista utilize mais memória em comparação com a simplesmente encadeada, o benefício da maior eficiência e menor complexidade das operações justifica a sua adoção para o módulo agendamento.

Responsabilidades do grupo:

Integrante	Responsabilidade	Arquivo
Daniela Tamy	Módulo Lista	lista.c/lista.h
Laura Klemba	Módulo Paciente	paciente.h/paciente.c
Leonardo Pereira	Módulo Agendamento: Histórico	agendamento.c/agendamento.h
Nathalia Lyra	Módulo Agendamento: Agendamento	agendamento.c/agendamento.h

2. Módulo Lista

2.1. lista.h

Define as estruturas (nó, lista, lista cabeçalho) e as declarações das funções públicas que os módulos podem chamar.

Estruturas:

```
typedef struct No {
        void *dados; // ponteiro genérico: pode ser Paciente, Agendamento, etc.
    struct No *proximo;
} No;
// Estrutura de lista simples (sem cabeçalho)
typedef struct {
    No *inicio;
    int tamanho;
} Lista;
// Estrutura de lista com cabeçalho (para histórico/agendamentos)
typedef struct {
    No *cabeca;   // nó sentinela
    int tamanho;
} ListaCabecalho;
```

- Nó guarda um void *dados que pode apontar para Paciente* ou Agendamento*

- Lista é uma lista encadeada simples para pacientes

- Lista Cabeçalho usa nó sentinela para agendamentos e histórico

Funções Principais:

```
// Funções básicas para lista simples
void inicializarLista(Lista *lista);
void inserirNoFim(Lista *lista, void *dados);
void *buscarElemento(Lista *lista, int (*criterio)(void*, void*),
void *chave);
int removerElemento(Lista *lista, int (*criterio)(void*, void*),
void *chave, void (*liberarDados)(void*));
void liberarLista(Lista *lista, void (*liberarDados)(void*));
void percorrerLista(Lista *lista, void (*mostrar)(void*));


// Funções para lista com cabeçalho
void inicializarListaCabecalho(ListaCabecalho *lista);
void inserirNoFimCabecalho(ListaCabecalho *lista, void *dados);
void percorrerListaCabecalho(ListaCabecalho *lista, void
(*mostrar)(void*));
void liberarListaCabecalho(ListaCabecalho *lista, void
(*liberarDados)(void*));
```

2.2. lista.c

Possui o códigos de manipulação dos nós e ponteiros (inserir, buscar, remover, percorrer, liberar).

Funções lista simples:

- inicializarLista(Lista *lista)
    - inicia lista vazia
- inserirNoFim(Lista *lista, void *dados)
    - cria um novo nó com dados
    - se a lista estiver vazia, inicio = novo

    - caso contrário, percorre até o último nó e conecta
    - incrementa o tamanho
- buscarElemento(Lista *lista, int (*criterio)(void*, void*), void *chave)
    - int (*criterio)(void*, void*) = ponteiro para função de comparação
    - percorre todos os nós e chama critério(aux->dados, chave)
    - critério é uma função definido pelo módulo que usa a lista (ex: compara CPF)
- removerElemento(Lista *lista, int (*criterio)(void*, void*), void *chave)
    - percorre guardando anterior e atual
    - se criterio(atual->dados,chave) for verdadeiro, deve ajudar os ponteiros para pular atual e free(atual) e decrementa tamanho
- percorrerLista(Lista *lista, void (*mostrar)(void*))
    - faz mostrar(aux->dados) para cada nó
    - mostrar é uma função de impressão
- liberarLista (Lista *lista, void (*liberarDados)(void*))
    - percorre a lista e free em cada nó

Funções lista cabeçalho:

- inicializarListaCabecalho(ListaCabecalho, *lista)
    - cria um nó sentinela (nunca é removido)
- inserirNoFimCabecalho(ListaCabelho *lista, void *dados)
    - começa do header e percorre até o último
    - incrementa tamanho
- percorrerListaCabecalho(ListaCabelho *lista, void (*mostrar)(void*))
    - inicia no hearder->proximo
    - mostrar é uma função de impressão
- liberarListaCabecalho(ListaCabecalho*lista,*void (*liberarDados)(void*))
    - liberar os nós e depois free(cabeca)

3. Módulo Agendamento

3.1. agendamento.h

Define a estrutura do módulo de controle dos agendamentos da clínica, para além de declarar as funções relativas ao agendamento.

Estruturas:

```
// Estrutura de agendamento
typedef struct {
    char cpf[15];
    char sala[10];
    char data[11];
    char hora[6];
} Agendamento;

// Estrutura de chave auxiliar para busca/remoção
typedef struct {
    char cpf[15];
    char data[11];
} ChaveAgendamento;

// Estrutura de histórico
typedef struct Historico {
    int total_agendamentos;
    int agendamentos_ativos;
    int agendamentos_cancelados;
    ListaCabecalho registros;
} Historico;
```

A estrutura Agendamento representa um agendamento individual, já a estrutura ChaveAgendamento é utilizada durante operações de busca e remoção, por exemplo.

Funções principais e auxiliares:

```
// Funções principais

Agendamento* criarAgendamento(char *cpf, char *sala, char *data, char
*hora);

void cadastrarAgendamento(ListaCabecalho *lista, Historico *historico,
char *cpf_paciente);

void listarAgendamentosPorCPF(ListaCabecalho *lista, ListaPacientes
*pacientes, char *cpf);

void listarAgendamentosPorSala(ListaCabecalho *lista, ListaPacientes
*pacientes, char *sala);

int removerAgendamento(ListaCabecalho *lista, Historico *historico,
char *cpf, char *data);

void mostrarAgendamento(void *dados);
```

```
// Funções histórico

Historico* criarHistorico();

void adicionarHistorico(Historico *historico, Agendamento
*agendamento);

void exibirHistorico(Historico *historico);

void exibirEstatisticasHistorico(Historico *historico);

void liberarHistorico(Historico *historico);
```

São sete as funções desse módulo que serão utilizadas em outros arquivos, como o main.c.

3.2. agendamento.c

- Agendamento* criarAgendamento(char *cpf, char *sala, char *data, char *hora)
    - Aloca dinamicamente um novo agendamento na memória utilizando-se de ponteiros.
    - Copia os valores recebidos (CPF, sala, data e hora) para a estrutura.
    - Retorna o ponteiro para o agendamento criado.
- void cadastrarAgendamento(ListaCabecalho *lista, Historico *historico, char *cpf_paciente)
    - Após a validação do CPF, lê os dados inseridos pelo usuário, cria um agendamento e insere na lista com cabeçalho. Em seguida, o código:
        - Recebe o CPF do paciente como parâmetro (cpf_paciente).
        - Pede ao usuário os dados restantes: sala, data, hora.
        - Cria um agendamento com esses dados usando criarAgendamento.
        - Insere o agendamento no final da lista com inserirNoFimCabecalho.
        - Adiciona ao histórico.
        - Mostra uma mensagem de sucesso.
- void listarAgendamentosPorCPF(ListaCabecalho *lista, ListaPacientes *pacientes, char *cpf)
    - Busca o paciente pelo CPF usando buscarPacientePorCPF.

    - Percorre a lista de agendamentos a partir do primeiro nó.
    - Para cada agendamento com CPF igual ao informado exibe data, hora e sala do agendamento.
    - Caso o paciente exista, exibe também nome, GRR e curso.
    - Se não houver agendamentos, exibe uma mensagem informando isso.
- void listarAgendamentosPorSala(ListaCabecalho *lista, ListaPacientes *pacientes, char *sala)
    - Percorre a lista de agendamentos a partir do primeiro nó.
    - Para cada agendamento cuja sala coincide com a fornecida busca os dados do paciente pelo CPF.
    - Se encontrado, exibe nome, CPF, curso, data e hora.
    - Se não encontrado, exibe apenas CPF, data e hora.
    - Informa se nenhum agendamento for encontrado para a sala.
- int removerAgendamento(ListaCabecalho *lista, Historico *historico, char *cpf, char *data);
    - Percorre a lista usando dois ponteiros ant e atual. O ant começa no header e o atual começa no primeiro nó.
    - Procura por um agendamento cujo CPF e data coincidam com os parâmetros.
    - Se encontrado modifica o status no histórico
    - Se encontrado remove o nó da lista, alterando os ponteiros de ant e atual.
    - Libera a memória do agendamento e do nó removido.
    - Atualiza o tamanho da lista, exibe mensagem de sucesso e retorna 1.
    - Se não encontrado exibe mensagem de erro e retorna 0.
- void mostrarAgendamento(void *dados)
    - Converte status do agendamento para um formato textual.
    - Mostra os dados de agendamento e o status no histórico.
- void exibirHistorico(Historico *historico)
    - Mostra todos os registros armazenados no histórico (ativos e cancelados).
- void exibirEstatisticasHistorico(Historico *historico)
    - Mostra os totais:
        - Total de agendamentos criados

        - Ativos atualmente
        - Cancelados
- void liberarHistorico(Historico *historico)
    - Libera toda a memória associada à estrutura de histórico.

3.3 Dificuldades

A maior dificuldade na implementação do módulo agendamento foi a manipulação correta dos nós nas operações de inserção, remoção e atualização da lista. Além disso, a interação entre o módulo agendamento com paciente e histórico também foi um elemento mais desafiador, à exemplo da função cadastraAgendamento que necessita validação dos dados da lista de pacientes e da função adicionarHistorico que é criada e manipulada a partir da lista de agendamentos.

4. Módulo Paciente

4.1. paciente.h

Define a estrutura do módulo de controle dos pacientes da clínica, para além de declarar as funções relativas ao cadastro, busca, etc.

Estrutura:

```
typedef struct {
    char nome[100];
    char cpf[20];
    char grr[15];
    char curso[50];
} Paciente;

// estrutura para a lista de pacientes -> em lista.h
typedef Lista ListaPacientes;
```

A estrutura “typedef struct”, uma estrutura que não possui nome próprio, representa o cadastro individual de pacientes e a estrutura “typedef Lista ListaPacientes” é utilizada para a criação da lista de pacientes.

Funções principais e auxiliares:

```
// funções importantes da lsta
ListaPacientes* criarListaPacientes();
void liberarListaPacientes(ListaPacientes *lista);
void cadastrarPaciente(ListaPacientes *lista, char nome[], char cpf[], char grr[], char curso[]);
Paciente* buscarPacientePorCPF(ListaPacientes *lista, char cpf[]);
void exibirPacientes(ListaPacientes *lista);
int removerPacientePorCPF(ListaPacientes *lista, char cpf[]);
```

São 6 funções totais no módulo Paciente.

4.2. paciente.c

Em paciente.c foram criadas 3 funções que são chamadas de “callback”, ou seja, são funções auxiliares. Essas funções tem o objetivo de se passar como um argumento para outra função, nesse caso, as funções implementadas em lista.c.

Funções de callback:

- int compararCPF(void *dadosPaciente, void *chaveCPF) {
  - criada para comparar e conferir CPFs
  - Recebe o ponteiro da lista e “traduz” mostrando que é um paciente
  - Recebe a chave de busca genérica (`chaveCPF`) e a traduz para uma string (`char*`).
  - Compara o CPF do paciente (`p->cpf`) com essa chave de busca. Ela vai retornar 1 se forem iguais - como se dissesse à lista: "encontrou o cpf".
- void mostrarPaciente(void *dadosPaciente) {
  - criada para exibir os dados do paciente na tela
  - imprime os campos (dados) do paciente
  - chamando a função exibirPacientes(Lista) que chama função genérica percorrerLista(lista, mostrarPaciente) –> vai percorrer cada nó da lista e para cada nó vai chamar a função callback, para que ela retorne os dados do paciente, se existir.
- void liberarDadosPaciente(void *dadosPaciente) {
  - criada para mostrar à lista como liberar a memória alocada para cada paciente
  - usa a função free para liberar a memória que foi utilizada para cada paciente em struct Paciente
- ListaPacientes* criarListaPacientes() {
  - função que retorna um ponteiro para a lista
  - usa malloc para separar um pedaço de memória
  - verifica se o sistema possui memória suficiente, se não, retorna NULL
  - chama a função InicializarLista(Lista) e garante que a lista comece vazia

- void liberarListaPacientes(ListaPacientes *lista) {
    - função que usa o callback (liberarDadosPaciente) para que, quando estiver percorrendo os nós da lista, para cada nó, chama-se a função, a fim de liberar espaço na memória. Antes apaga o paciente que está ali alocado, para assim, apagar o nó.
    - Após, usa o **free** para liberar esses espaços.
- void cadastrarPaciente(ListaPacientes *lista, char nome[], char cpf[], char grr[], char curso[]) {
    - criada para adicionar um paciente na lista
    - antes de fazer qualquer coisa, a função já chama a função de callback (compararCPF) para que se já existir um cpf igual, não continue rodando. No caso, se aparecer qualquer coisa além de NULL, já se sabe que existe algo lá.
    - utiliza o malloc para alocar memória também
    - verifica se a memória está disponível e correta para alocar
    - utiliza a função para copiar a string **nome** para o campo **novoPaciente→nome**. Utilizada também para conferir se a memória não vai alocar em algum lugar que não seja definido para ele
    - depois da estrutura, é chamada a função InserirNoFim. Essa é uma função genérica criada em lista.h e lista.c que faz a parte de criar um struct No, fazer esse novoPaciente ser apontado por No->dados, e conectar esse novo Nó no final da lista.
- Paciente* buscarPacientePorCPF(ListaPacientes *lista, char cpf[]) {
    - chama a função genérica buscarElemento e entrega pra ela os parâmetros que ela precisa: a lista (lista), a instrução de como comparar (compararCPF), e o que procurar (cpf). O (Paciente*) no início converte o resultado genérico (void*) de volta para um ponteiro de Paciente.
- void exibirPacientes(ListaPacientes *lista) {
    - primeiro faz a verificação se a lista é NULL (inválida) ou está vazia, se for verdadeiro, printa na tela que não há pacientes
    - se for falso, ele passa para a próxima parte e chama a função genérica “percorrerLista” que tem dois parâmetros.
    - os parâmetros são: (lista, mostrarPaciente) → lista é o ponteiro que passa por todos os nós da lista de pacientes e mostrarPaciente é a função auxiliar callback.
    - para cada nó da lista, a função callback é chamada para retornar e traduzir os dados genéricos obtidos em dados de pacientes.
- int removerPacientePorCPF(ListaPacientes *lista, char cpf[]) {
- essa função tem dois parâmetros - A lista que o paciente deve ser retirado e o cpf que deseja retirar

- chama a função genérica “removerElemento” (que está em lista.c) e coloca 4 parâmetros para ela analisar → lista, compararCPF, cpf e liberarDadosPaciente
- A lista é a que vamos procurar o paciente a ser removido. Para cada nó da lista, chama-se a função callback para comparar o cpf com o cpf digitado/desejado e apenas quando esses cpfs forem iguais, é chamada a segunda função de callback que fica responsável por apagar todos os dados desse paciente antes do nó ser apagado. A fim de evitar vazamento de memória.

4.3. Dificuldades

A maior dificuldade na criação e implementação do módulo de paciente foram as funções de callback. Como os arquivos paciente.c e paciente.h utilizam funções genéricas de lista.c e lista.h, elas retornam dados brutos, sem um significado em si. Por este motivo, tive que desenvolver essas funções auxiliares para "traduzir" esses dados para as informações que o módulo realmente use, como o cpf, por exemplo.

5. Conclusão

Conclui-se que o projeto do Sistema de Agendamento de Consultas em uma Clínica Universitária permitiu os conceitos aprendidos na disciplina de Estrutura de Dados I, especialmente em relação ao uso de listas encadeadas e modularização de códigos. A escolha da lista simplesmente encadeada para o módulo de pacientes e da lista com cabeçalho para o módulo de agendamentos mostrou-se adequada às necessidades do sistema. Apesar das dificuldades com ponteiros, callbacks e integração entre os módulos, o trabalho foi importante para o entendimento das estruturas de listas, da maneira como ocorre a alocação dinâmica, do uso de ponteiros e da forma como a memória pode ser gerenciada.