Relatório do trabalho de Estruturas de Dados e Algoritmos

Universidade de Évora 24 de junho de 2025

Alunos:

60585 - Rafael Faria Delgado

62901 - André Garcia

63481 - Andre Mansinho

Introdução

Este relatório apresenta e descreve o desenvolvimento do software “Agenda de telemóvel”. O principal objetivo do trabalho no desenvolvimento de um software pratico e eficiente para a gerência de contactos telefónicos e registar chamadas realizadas.

O relatório está organizado de forma a fornecer uma visão completa e clara de todas as etapas do trabalho. Começamos por explicar como foi desenvolvido primeiramente o código-fonte em ficheiros.c na pasta “CFile” para permitir a realização de teste isolados de cada modulo, em seguida criamos as interfaces(.h) na pasta “Include” e por último, o desenvolvimento do ficheiro principal (main.c) que integra todas as funcionalidades.

O processo de desenvolvimento foi baseado nos aprendizados das aulas da disciplina e foi utilizado o IDE VSCode.

Desenvolvimento

1. Escolha das estruturas de dados utilizadas.
2. Desenvolvimento dos códigos
3. Testes.

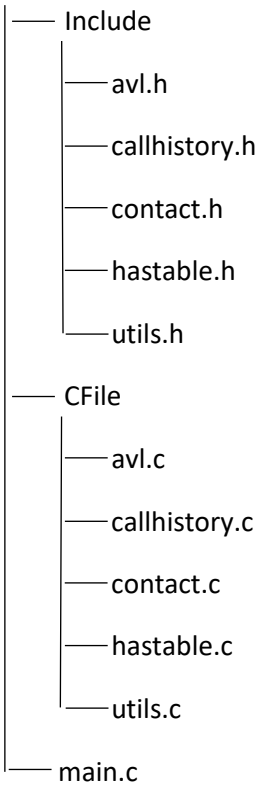
Estruturas de Dados Utilizados

A escolha das estruturas de dados exigiu uma análise para selecionar as mais adequadas e também foi seguido as recomendações da professora, considerando a eficiência em operações e facilidade na gestão dos dados. Foram escolhidas três estruturas cada uma com funções especificas para alcançar os objetivos do trabalho.

- **AVL**: foi utlizada esta estrutura devido a sua excelente performance em manter os dados ordenados alfabeticamente, assim garantindo operações rápidas de introdução, remoção e pesquisas rápidas. Todas complexidade O(log n).
- **Hashtable**: esta estrutura se destacou pela capacidade de realizar operações de forma rápida O(1) em média de introdução, remoção e pesquisa. Sendo útil para operação frequentes que necessitam de agilidade, como a gestão e pesquisa exata de contactos por nome.
- **Lista duplamente ligado**: foi utilizado porque permite uma navegação eficiente em ambas as direções, permitindo movimentar-se rapidamente tanto para trás tanto para frente. Inserção e remoção são feitas de forma rápida e simples rápida O(1), entregando uma boa flexibilidade e controlo.

Organização do Código

O trabalho se encontra organizado em três diretórios para facilitar a gestão e a manutenção do código:



- **CFile**: foi criado primeiro, contem todos os ficheiros do código-fonte que realizam as funcionalidades essências do trabalho, em concreto: avl.c, contact.c, hashtable.c, callhistory.c e utils.c. A criação inicial destes ficheiros permitiu-nos testar de forma isolada cada módulo antes de avançar para as etapas seguintes.
  - **Avl.c**: este modulo representa a implementação de uma robusta e bem estruturada arvore AVL, utilizar uma arvore AVL desempenha um papel critico para manter os contactos organizados alfabeticamente, que permite consultas rápidas.

    Este modulo inclui várias funções fundamentais, como:

    - **Criação de nó AVL**(create_avlnode): responsável pela criação e inicialização de novos nós. Um nó contém o contacto, ponteiros para subárvores esquerda e direita e a altura do nó. static AVLNode* create_avlnode(Contact *contact).

    - **Inserção de Contactos** (avlnode_insert): insere novos contactos e realiza automaticamente o balanceamento da arvore através de rotações, mantendo a estrutura otimizada.

```c
// Insere um contacto recursivamente na subárvore e aplica balanceamentos
static AVLNode* avlnode_insert(AVLNode *node, Contact *contact) {
    if (!node) return create_avlnode(contact);

    // Insere recursivamente na esquerda ou direita, conforme a ordem alfabética
    if (compare_contacts(contact, node->contact) < 0) {
        node->left = avlnode_insert(node->left, contact);
    } else {
        node->right = avlnode_insert(node->right, contact);
    }

    // Atualiza a altura do nó atual
    node->height = 1 + max(height(node->left), height(node->right));

    // Obtém o fator de balanceamento
    int balance = get_balance(node);

    // Casos de desbalanceamento:

    // Left Left Case
    if (balance > 1 && compare_contacts(contact, node->left->contact) < 0) {
        return right_rotate(node);
    }

    // Right Right Case
    if (balance < -1 && compare_contacts(contact, node->right->contact) > 0) {
        return left_rotate(node);
    }

    // Left Right Case
    if (balance > 1 && compare_contacts(contact, node->left->contact) > 0) {
        node->left = left_rotate(node->left);
        return right_rotate(node);
    }

    // Right Left Case
    if (balance < -1 && compare_contacts(contact, node->right->contact) < 0) {
        node->right = right_rotate(node->right);
        return left_rotate(node);
    }

    return node;
}
```

- **Callhistory.c**: é responsável pela gestão do histórico de chamadas telefónicas, permitindo armazenar, navegar e exibir informações detalhadas das chamadas efetuadas.
  A sua implementação utiliza uma lista duplamente ligada, que oferece eficiência e facilidade para inserir novos registos no início e navegar entre chamadas rapidamente.

  Este modulo inclui várias funções fundamentais, como:

  - **Adição de chamadas**(add-call): insere novas chamadas no inicio da lista, garantindo que as chamadas mais recentes estejam disponíveis.

```c
// Adiciona uma nova chamada ao início da lista (mais recente primeiro)
void add_call(CallHistory *history, CallRecord call) {
    if (!history) return;

    // Cria um novo nó de chamada
    CallNode *new_node = (CallNode*)malloc(sizeof(CallNode));
    if (!new_node) return;

    new_node->call = call;
    new_node->next = history->head;
    new_node->prev = NULL;

    // Atualiza ponteiros se a lista não estiver vazia
    if (history->head) {
        history->head->prev = new_node;
    } else {
        history->tail = new_node;
    }

    // Atualiza o início da lista e a posição atual
    history->head = new_node;
    history->current = new_node;
    history->size++;
}
```

- **Liberação do histórico**(free_callhistory): liberta toda a memoria alocada, evitando vazamentos e garantindo boa gestão de recursos.

```c
// Liberta toda a memória alocada para o histórico
void free_callhistory(CallHistory *history) {
    if (!history) return;

    CallNode *current = history->head;
    while (current) {
        CallNode *temp = current;
        current = current->next;
        free(temp); // Liberta cada nó da lista
    }
    free(history); // Liberta a estrutura principal
}
```

- **contact.c**: este modulo desempenha um papel na gestão segura e eficiente das informações pessoais dos utilizadores. Fornece operações básicas, porem crucias, tais como, criação, impressão e libertação de contactos. Garante que todos os outros módulos possam manipular os contactos facilmente.
  Este modulo inclui várias funções fundamentais, como:
  - Comparação de dois contactos(compare_contacts): esta função compara os nomes de dois contactos utilizando a função padrão C.

```c
// Compara dois contactos com base no nome (ordem alfabética)
int compare_contacts(const Contact *a, const Contact *b) {
    return strcmp(a->name, b->name);
}
```

- Criação de um contacto(create_contact): esta função e responsável por criar e inicializar um novo contacto, recebe o nome e o número de telefone como parâmetros. Envolve alocar a memoria dinamicamente para a estrutura e copiar as strings recebidas usando a função strdup.

```c
// Cria um novo contacto a partir do nome e número fornecidos
Contact* create_contact(const char *name, const char *phone) {
    // Aloca memória para a estrutura Contact
    Contact *new_contact = (Contact*)malloc(sizeof(Contact));
    if (!new_contact) return NULL;

    // Duplica as strings fornecidas (aloca nova memória para elas)
    new_contact->name = strdup(name);
    new_contact->phone = strdup(phone);

    // Verifica se houve falha na duplicação e limpa, se necessário
    if (!new_contact->name || !new_contact->phone) {
        free_contact(new_contact);
        return NULL;
    }

    return new_contact;
}
```

- **Hashtable.c: implementa uma tabela de dispersão, utilizada para armazenar e gerir contactos de forma eficiente. Hashtable é o núcleo de gerência de contactos.**

  Este modulo inclui várias funções fundamentais, como:
  - **Função de dispersão(hash_function):** usa o algoritmo DJB2, um método clássico e eficaz para dispersão de strings, transformando o nome do contacto em um índice da tabela.

```c
// Função de hash simples baseada no algoritmo DJB2 (muito comum para strings)
int hash_function(const char *key) {
    unsigned long hash = 5381;
    int c;

    // Processa cada caractere da string chave
    while ((c = *key++)) {
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    }

    return hash % TABLE_SIZE;
}
```

- **Pesquisa de Contactos(hashtable_search):** esta função percorre a lista ligada no índice correspondente até encontrar o nome desejado.

```c
// Procura por um contacto pelo nome
Contact* hashtable_search(HashTable *table, const char *name) {
    if (!table || !name) return NULL;

    int index = hash_function(name);
    HashNode *current = table->buckets[index];

    // Percorre a lista do bucket correspondente
    while (current) {
        if (strcmp(current->contact->name, name) == 0) {
            return current->contact;
        }
        current = current->next;
    }

    return NULL;
}
```

Ficheiros include: nessa pasta definimos os ficheiros de interfacer(.h), que inclui avl.h, contact.h, hashtable.h, callhistory.h e utils.h. Cada ficheiro define os tipos abstratos de dados(TADs) e protótipos das funções disponíveis para utilização externa.

Main.c: esse é o coração da aplicação, aqui que tudo se junta. Atuando como a interface entre o utilizador e as estruturas de dados implementadas nos restantes módulos, realizando a interação com o utilizador através dum menu simples. Entre as principais funcionalidades do main encontra-se

1. **Adicionar Contacto**

- Cria um novo contacto via input_contact().
- Insere-o na hashtable (hashtable_insert) e na AVL (avltree_insert).

2. **Editar Contacto**

- Pesquisa o contacto na hashtable.
- Remove-o da hashtable (e da AVL implicitamente).
- Cria novo com dados atualizados e reinsere.

3. **Remover Contacto**

- Remove da hashtable (não da AVL — indicado como simplificação).

4. **Pesquisar Contacto**

- Realiza uma pesquisa exata na hashtable pelo nome.

5. **Listar Contactos Ordenadamente (opção 5)**

- Usa avltree_inorder() para listar contactos por ordem alfabética.

6. **Registar. Chamada**

- Pede os números e a duração.
- Usa add_call() para inserir no histórico.

7. **Ver Histórico de Chamadas**

- Mostra as últimas N chamadas via print_recent_calls().

8. **Navegar no Histórico**

- Usa navigate_history() para ir à chamada anterior ou seguinte e apresenta os dados da chamada formatados com strftime().

Conclusão

O desenvolvimento deste projeto proporcionou uma oportunidade única para consolidar os conhecimentos teóricos e práticos adquiridos na disciplina de Estrutura de Dados e Algoritmos.

O código desenvolvido cumpre os requisitos, o trabalho esta estrutura de forma mais simples e eficiente possível considerando o nosso nível de conhecimento.