



UNIVERSIDADE FEDERAL DE RONDÔNIA

NÚCLEO DE TECNOLOGIA

DEPARTAMENTO ACADÊMICO DE CIÊNCIA DA COMPUTAÇÃO

CARLA GABRIELA CAVALCANTE RIBEIRO

T2 - ÁRVORES B

PORTO VELHO

2024

1. Introdução

As Árvores-B são estruturas de dados que, apesar de serem simples em conceito, possuem uma sofisticação que as torna extremamente importantes em sistemas de armazenamento e bancos de dados. Elas organizam grandes volumes de dados de forma eficiente, permitindo operações rápidas de busca, inserção e exclusão. Uma das principais características das Árvores-B é que elas mantêm os dados balanceados, garantindo que todas as folhas estejam no mesmo nível, o que ajuda a otimizar o tempo de acesso, mesmo quando a quantidade de dados é grande.

Este relatório documenta a implementação de uma Árvore-B utilizando o conceito de Tipo Abstrato de Dados (TAD). A atividade foi proposta como parte prática avaliativa da disciplina de Estrutura de Dados II, com o objetivo de aplicar conceitos teóricos de Árvores B, manipulação de strings em C e leitura/escrita em arquivo texto aprendidos em sala de aula. O desenvolvimento foi realizado no sistema operacional Windows 11, utilizando o ambiente de desenvolvimento Visual Studio Code, com a linguagem de programação C, conforme as diretrizes da disciplina.

2. Objetivo

O objetivo proposto para o trabalho é implementar uma Árvore-B utilizando a linguagem de programação C. A Árvore-B deve ser capaz de realizar as operações básicas de inserção, exibição, busca e destruição, mantendo suas propriedades de balanceamento e ordenação. As chaves armazenadas na árvore serão cadeias de caracteres sem espaços, com até 20 posições.

Implementar essa estrutura em C exige o uso de ponteiros e alocação dinâmica de memória para gerenciar os nós e as chaves, garantindo que todas as operações mantenham as propriedades de balanceamento e ordenação da árvore.

A aplicação irá permitir ao usuário interagir com a Árvore-B dinamicamente por meio de um menu de opções, onde ele poderá carregar a árvore a partir de arquivo texto, inserir novas chaves, exibir a árvore em sua forma atual, buscar por chaves específicas e destruir a árvore quando necessário.

3. Implementação

O trabalho foi estruturado de maneira modular utilizando o conceito de Tipo Abstrato de Dados (TAD) que separa a interface (declarações de funções e tipos de dados) da implementação (lógica de funções) facilitando a manutenção e extensibilidade do código, e está organizado nos seguintes arquivos:

- `main.c`: Este arquivo é responsável por exibir o menu e permitir que o usuário execute operações sobre a Árvore-B. Ele inicializa a árvore e interage com o usuário, chamando as funções apropriadas definidas no arquivo `abb.c`.
- `abb.c`: Contém toda a lógica e implementação das funções relacionadas à manipulação da Árvore-B. Isso inclui operações como inserção, busca, remoção e outras operações específicas da estrutura da árvore. Este arquivo é responsável por todas as funcionalidades que a árvore deve fornecer.
- `abb.h`: Este arquivo é o cabeçalho que declara as funções e tipos de dados utilizados no `abb.c`. Ele define a interface pública das funções e estruturas, permitindo que `main.c` e outros arquivos interajam com as funcionalidades da Árvore-B. O cabeçalho implementado se apresenta da seguinte forma.

```

#ifndef ABB_H
#define ABB_H

#define RESET    "\033[0m"
#define BLUE     "\033[34m"
#define GREEN    "\033[32m"
#define YELLOW   "\033[33m"
#define RED      "\033[31m"
#define MAX_CHAVES 5 // Número máximo de chaves em um nó
#define MAX_CARACTERES 21 // Tamanho máximo de uma chave (20 + '\0')

typedef struct No {
    int numChaves;
    char chaves[MAX_CHAVES][MAX_CARACTERES];
    struct No *filhos[MAX_CHAVES + 1];
    int folha; // 1 se o nó é folha, 0 caso contrário
} No;

typedef struct ArvoreB {
    No *raiz;
    int t;
} ArvoreB;

typedef struct {
    No *no;
    int pos;
    int ehInterno;
    int ehRaiz;
    int numFilhos;
    char irmaoEsquerdo[MAX_CARACTERES];
    char irmaoDireito[MAX_CARACTERES];
    No *pai;
} ResultadoBusca;

// Funções para a Árvore-B
ArvoreB* criarArvoreB(int ordem);
void destruirArvoreB(ArvoreB *arvore);
void inserir(ArvoreB *arvore, const char *chave);
void exibirArvoreB(No *no, int nivel, int ehPai);
ResultadoBusca buscar(No *no, const char *chave, No *pai, No *irmaoEsquerdo, No *irmaoDireito, int ehRaiz);
void carregarArquivo(ArvoreB *arvore, const char *nomeArquivo);
void inserirArquivo(ArvoreB *arvore, const char *nomeArquivo, const char *chave);
void emOrdem(No *raiz);

#endif

```

a. Menu

O menu interativo apresenta ao usuário as opções das várias operações implementadas relacionadas a árvore. Utilizando códigos ANSI para realizar a customização de cores e personalizar o menu proporcionando melhor legibilidade.

```

#define RESET    "\033[0m"
#define BOLD     "\033[1m"
#define UNDERLINE "\033[4m"
#define BLUE     "\033[34m"
#define GREEN    "\033[32m"
#define CYAN     "\033[36m"
#define RED      "\033[31m"

void menu() {
    printf("\n");
    printf(BOLD GREEN "===== \n" RESET);
    printf(BOLD CYAN "          ÁRVORE-B IMPLEMENTAÇÃO \n" RESET);
    printf(BOLD GREEN "===== \n" RESET);
    printf("1. " BLUE "Carregar dados do arquivo" RESET "\n");
    printf("2. " BLUE "Inserir chave" RESET "\n");
    printf("3. " BLUE "Exibir árvore" RESET "\n");
    printf("4. " BLUE "Imprimir em ordem" RESET "\n");
    printf("5. " BLUE "Buscar chave" RESET "\n");
    printf("6. " BLUE "Destruir árvore" RESET "\n");
    printf("7. " BLUE "Sair" RESET "\n");
    printf(BOLD GREEN "===== \n" RESET);
    printf("Escolha uma opção: ");
}

```

```

=====
          ÁRVORE-B IMPLEMENTAÇÃO
=====
1. Carregar dados do arquivo
2. Inserir chave
3. Exibir árvore
4. Imprimir em ordem
5. Buscar chave
6. Destruir árvore
7. Sair
=====
Escolha uma opção: |

```

Já na inicialização o programa inicializa a árvore B com variáveis que definem grau mínimo de 3 e define o nome do arquivo a ser usado podendo ser modificado pelo usuário conforme a necessidade.

```

ArvoreB *arvore = criarArvoreB(3);
char nomeArquivo[] = "testeB.txt";

```

Destacam-se das opções do menu:

- **Carregar Dados:** Carrega dados de um arquivo .txt para a árvore.
- **Inserir Chave:** Permite ao usuário inserir uma nova chave na árvore.
- **Exibir Árvore:** Mostra a estrutura atual da árvore.
- **Imprimir em Ordem:** Exibe a árvore em ordem crescente.
- **Buscar Chave:** Procura uma chave específica e exibe o resultado.
- **Destruir Árvore:** Destrói a árvore atual.

b. Inserir

A inserção de chaves pode ser feita de duas formas: (a) carregando as chaves a partir de arquivo texto e (b) inserido manualmente a chave pelo usuário. A inserção manual de chaves além de inserir a chave na árvore também armazena a chave no arquivo .txt usando append, ou seja, caso o arquivo exista, ele irá adicionar a nova chave no fim do arquivo e caso não exista irá criar o arquivo. Dessa forma mesmo que o usuário não possua um conjunto de dados pré-existente o programa permitirá o preenchimento da árvore chave a chave.

```
void carregarArquivo(ArvoreB *arvore, const char *nomeArquivo) {
    FILE *file = fopen(nomeArquivo, "r");
    if (file == NULL) {
        perror(RED "Erro ao abrir o arquivo" RESET);
        return;
    }
    printf(GREEN "Dados carregados do arquivo com sucesso!\n" RESET);
    char chave[MAX_CARACTERES];
    while (fscanf(file, "%s", chave) != EOF) {
        inserir(arvore, chave);
    }
    fclose(file);
}
```

A função *carregarArquivo* tem o objetivo de carregar e inserir chaves em uma Árvore-B a partir de um arquivo texto. Essa função fará todo o procedimento de abertura do arquivo usando fopen e caso tudo ocorra bem, utilizando um buffer de caracteres a função lerá as chaves uma a uma até o fim do arquivo e vai chamar a função inserir que fará a inserção das chaves na árvore. Ou caso o

usuário opte por inserir manualmente, a função responsável por abrir o arquivo e fazer as chamadas da função inserir será ***inserirArquivo***

```
void inserirArquivo(ArvoreB *arvore, const char *nomeArquivo, const char *chave) {
    inserir(arvore, chave);
    FILE *file = fopen(nomeArquivo, "a");
    if (file == NULL) {
        perror("Erro ao abrir o arquivo");
        return;
    }
    fprintf(file, "%s\n", chave);
    fclose(file);
}
```

Quando a função ***inserir*** é chamada, ela primeiro verifica se a raiz da árvore está cheia, ou seja, se contém o número máximo de chaves permitido ($2 * \text{ordem} - 1$). Se a raiz está cheia, o código procede para criar uma nova raiz e dividir a antiga raiz em dois nós filhos. Esse processo de divisão é gerenciado pela função ***dividirFilho***. A divisão promove a chave do meio da raiz original para a nova raiz, fazendo com que a estrutura da árvore se expanda em altura, se necessário. Após essa divisão, a inserção da nova chave continua de maneira recursiva, com a função ***inserirNaoCheio***, que é chamada para inserir a chave na posição correta na nova estrutura.

```

void inserir(ArvoreB *arvore, const char *chave) {
    No *raiz = arvore->raiz;
    if (raiz->numChaves == 2 * arvore->t - 1) {
        No *novaRaiz = criarNo(arvore->t, 0);
        arvore->raiz = novaRaiz;
        novaRaiz->filhos[0] = raiz;
        dividirFilho(novaRaiz, 0, arvore->t);
        inserirNaoCheio(novaRaiz, chave, arvore->t);
    } else {
        inserirNaoCheio(raiz, chave, arvore->t);
    }

    printf(BLUE "Árvore-B após inserção:\n" RESET);
    exibirArvoreB(arvore->raiz, 0, 1);
}

```

```

void dividirFilho(No *pai, int indiceFilho, int ordem) {
    No *filhoEsquerdo = pai->filhos[indiceFilho];
    No *filhoDireito = criarNo(ordem, filhoEsquerdo->folha);
    filhoDireito->numChaves = ordem - 1;

    for (int indiceChave = 0; indiceChave < ordem - 1; indiceChave++) {
        strcpy(filhoDireito->chaves[indiceChave], filhoEsquerdo->chaves[indiceChave + ordem]);
    }

    if (!filhoEsquerdo->folha) {
        for (int indiceFilhoEsquerdo = 0; indiceFilhoEsquerdo < ordem; indiceFilhoEsquerdo++) {
            filhoDireito->filhos[indiceFilhoEsquerdo] = filhoEsquerdo->filhos[indiceFilhoEsquerdo + ordem];
        }
    }

    filhoEsquerdo->numChaves = ordem - 1;

    for (int indicePaiFilho = pai->numChaves; indicePaiFilho >= indiceFilho + 1; indicePaiFilho--) {
        pai->filhos[indicePaiFilho + 1] = pai->filhos[indicePaiFilho];
    }
    pai->filhos[indiceFilho + 1] = filhoDireito;

    for (int indicePaiChave = pai->numChaves - 1; indicePaiChave >= indiceFilho; indicePaiChave--) {
        strcpy(pai->chaves[indicePaiChave + 1], pai->chaves[indicePaiChave]);
    }

    strcpy(pai->chaves[indiceFilho], filhoEsquerdo->chaves[ordem - 1]);
    pai->numChaves++;

    printf(BLUE "Dividido o filho %d do nó pai.\n" RESET, indiceFilho);
}

```



```

void inserirNaoCheio(No *noAtual, const char *chave, int ordem) {
    int indiceChaveAtual = noAtual->numChaves - 1;

    if (noAtual->folha) {
        while (indiceChaveAtual >= 0 && strcmp(chave, noAtual->chaves[indiceChaveAtual]) < 0) {
            strcpy(noAtual->chaves[indiceChaveAtual + 1], noAtual->chaves[indiceChaveAtual]);
            indiceChaveAtual--;
        }
        strcpy(noAtual->chaves[indiceChaveAtual + 1], chave);
        noAtual->numChaves++;
        printf(GREEN "Chave '%s' inserida no nó folha.\n" RESET, chave);
    } else {
        while (indiceChaveAtual >= 0 && strcmp(chave, noAtual->chaves[indiceChaveAtual]) < 0) {
            indiceChaveAtual--;
        }
        indiceChaveAtual++;
        if (noAtual->filhos[indiceChaveAtual]->numChaves == 2 * ordem - 1) {
            dividirFilho(noAtual, indiceChaveAtual, ordem);
            if (strcmp(chave, noAtual->chaves[indiceChaveAtual]) > 0) {
                indiceChaveAtual++;
            }
        }
        inserirNaoCheio(noAtual->filhos[indiceChaveAtual], chave, ordem);
    }
}

```

Caso a raiz não esteja cheia, a função `inserirNaoCheio` é chamada diretamente para inserir a nova chave. Essa função começa verificando se o nó atual (inicialmente a raiz) é uma folha. Se for uma folha, a chave é inserida diretamente no nó, deslocando outras chaves, se necessário, para manter a ordem. Se o nó não for uma folha, a função procura o filho adequado para descer e continuar a inserção. Antes de descer, ela verifica se o nó filho está cheio. Se estiver cheio, o filho é dividido utilizando a função `dividirFilho`, e a chave do meio é promovida para o nó atual. A função então continua a inserção no nó filho apropriado, recursivamente.

A função `inserir` também faz chamadas à função `exibir` a cada inserção para que o usuário possa acompanhar as mudanças na árvore.

c. Exibir

Para a função de exibição foram implementadas duas alternativas distintas. Na primeira implementação a função `exibirArvoreB` é responsável por exibir a estrutura da Árvore-B na tela, mostrando as chaves em cada nó em diferentes cores utilizando o padrão de códigos ANSI, dependendo de sua posição e nível na árvore.

```

void exibirArvoreB(No *no, int nivel, int ehPai) {
    if (no != NULL) {
        int indiceExibir;
        for (indiceExibir = 0; indiceExibir < no->numChaves; indiceExibir++) {
            if (!no->folha) {
                exibirArvoreB(no->filhos[indiceExibir], nivel + 1, 0);
            }
            for (int nivelExibir = 0; nivelExibir < nivel; nivelExibir++) {
                printf("  ");
            }
            if (ehPai) {
                printf(BLUE "%s" RESET "\n", no->chaves[indiceExibir]);
            } else if (!no->folha) {
                printf(GREEN "%s" RESET "\n", no->chaves[indiceExibir]);
            } else {
                printf(YELLOW "%s" RESET "\n", no->chaves[indiceExibir]);
            }
        }
        if (!no->folha) {
            exibirArvoreB(no->filhos[indiceExibir], nivel + 1, 0);
        }
    }
}

```

Ela começa verificando se o nó atual não é nulo. Se o nó não for nulo, a função entra em um laço que percorre todas as chaves do nó. Durante esse laço, se o nó não for uma folha, a função é chamada recursivamente para exibir os filhos à esquerda de cada chave.

Para cada chave no nó, a função imprime o número adequado de espaços em branco, correspondente ao nível do nó na árvore, criando uma indentação visual que reflete a profundidade do nó na estrutura da árvore. As chaves são impressas em cores diferentes, dependendo do contexto: se o nó é o pai de outros nós, as chaves são exibidas em azul; se o nó não é uma folha mas também não é um pai, as chaves são exibidas em verde; e se o nó é uma folha, as chaves são exibidas em amarelo.

Depois que todas as chaves e seus respectivos filhos à esquerda foram exibidos, a função verifica se o nó não é uma folha e, nesse caso, chama recursivamente `exibirArvoreB` para o último filho à direita, completando a exibição do nó e seus filhos. Assim, a função percorre toda a árvore exibindo suas chaves de maneira hierárquica e organizada, a fim de facilitar a visualização da árvore.

Outra alternativa de exibir que foi implementada é a em Ordem

```
void emOrdem(No *raiz) {
    if (raiz == NULL) {
        return;
    }

    if (raiz->filhos[0] != NULL) {
        emOrdem(raiz->filhos[0]);
    }

    for (int i = 0; i < raiz->numChaves; i++) {
        printf("%s ", raiz->chaves[i]);

        if (raiz->filhos[i + 1] != NULL) {
            emOrdem(raiz->filhos[i + 1]);
        }
    }
}
```

A função `emOrdem` realiza a impressão em ordem na árvore, de forma que todas as chaves sejam visitadas e impressas em ordem crescente. Ela começa verificando se o nó atual é nulo; se for, a função retorna imediatamente, indicando que não há mais nós para percorrer. Se o nó não for nulo, a função entra em um laço que percorre todas as chaves do nó atual. Para cada chave, a função primeiro chama recursivamente `emOrdem` no filho à esquerda, se existir, para processar todas as chaves menores antes da chave atual. Em seguida, a chave é impressa, e a função continua para a próxima chave no nó. Após percorrer todas as chaves, a função chama recursivamente `emOrdem` no último filho à direita, garantindo que todas as chaves maiores também sejam processadas em ordem crescente. Esse processo recursivo garante que todas as chaves na árvore sejam visitadas e impressas em ordem, realizando a impressão completa da árvore.

d. Buscar

A busca foi implementada para procurar uma chave específica na árvore e retornar informações detalhadas sobre a chave encontrada (como por exemplo, se a chave é raiz, folha, nó interno, se tem irmãos adjacentes, quem são os irmãos e o pai) ou, se não foi encontrada. A função começa inicializando uma estrutura `ResultadoBusca`, que armazenará todas as informações do resultado da busca.

Primeiramente, a função percorre as chaves no nó atual, comparando cada uma com a chave buscada. Se encontrar a chave, preenche a estrutura ResultadoBusca com as informações para retornarem ao usuário. Se a chave não for encontrada no nó atual e o nó for uma folha, a função retorna o ResultadoBusca inicializado com valores indicando que a chave não foi encontrada. Entretanto, se o nó não for uma folha, a função continua a busca recursivamente no filho adequado, com base na posição onde a chave deveria estar. Antes de fazer a chamada recursiva, a função determina os novos irmãos esquerdo e direito que correspondem ao filho sendo pesquisado. Isso permite que, ao encontrar a chave (ou a posição em que deveria estar), a função forneça uma visão completa da estrutura da árvore ao redor da chave buscada.

```
ResultadoBusca buscar(No *no, const char *chave, No *pai, No *irmaoEsquerdo, No *irmaoDireito, int ehRaiz) {
    ResultadoBusca resultado = {NULL, -1, 0, 0, 0, "", "", NULL};

    int indiceBuscar = 0;
    while (indiceBuscar < no->numChaves && strcmp(chave, no->chaves[indiceBuscar]) > 0) {
        indiceBuscar++;
    }

    if (indiceBuscar < no->numChaves && strcmp(chave, no->chaves[indiceBuscar]) == 0) {
        resultado.no = no;
        resultado.pos = indiceBuscar;
        resultado.ehInterno = !no->folha;
        resultado.ehRaiz = ehRaiz;
        resultado.numFilhos = no->folha ? 0 : no->numChaves + 1;
        resultado.pai = pai;

        if (irmaoEsquerdo != NULL && irmaoEsquerdo->numChaves > 0) {
            strcpy(resultado.irmaoEsquerdo, irmaoEsquerdo->chaves[irmaoEsquerdo->numChaves - 1]);
        } else {
            resultado.irmaoEsquerdo[0] = '\0';
        }

        if (irmaoDireito != NULL && irmaoDireito->numChaves > 0) {
            strcpy(resultado.irmaoDireito, irmaoDireito->chaves[0]);
        } else {
            resultado.irmaoDireito[0] = '\0';
        }

        return resultado;
    }

    if (no->folha) {
        return resultado;
    } else {
        int indiceFilhoBuscar = indiceBuscar;
        No *filhoBuscar = no->filhos[indiceFilhoBuscar];
        No *novoIrmãoEsquerdo = (indiceFilhoBuscar > 0) ? no->filhos[indiceFilhoBuscar - 1] : NULL;
        No *novoIrmãoDireito = (indiceFilhoBuscar < no->numChaves) ? no->filhos[indiceFilhoBuscar + 1] : NULL;
        return buscar(filhoBuscar, chave, no, novoIrmãoEsquerdo, novoIrmãoDireito, 0);
    }
}
```

e. Destruir

```
void destruirNo(No *no) {
    if (no != NULL) {
        for (int i = 0; i <= no->numChaves; i++) {
            destruirNo(no->filhos[i]);
        }
        free(no);
    }
}

void destruirArvoreB(ArvoreB *arvore) {
    destruirNo(arvore->raiz);
    free(arvore);
}
```

Para destruir a árvore foi implementado `destruirArvoreB` pra ser responsável por liberar toda a memória alocada para a árvore como um todo. Ela faz isso chamando a função `destruirNo`, passando a raiz da árvore como argumento. Isso inicia o processo de liberação recursiva para todos os nós da árvore. Após a árvore ter sido totalmente liberada, a própria estrutura da árvore B é então liberada com a função `free`. Assim, `destruirArvoreB` garante que toda a memória utilizada pela árvore B, incluindo todos os seus nós e a estrutura da árvore em si, seja devidamente desalocada e não haja vazamentos de memória. A função `destruirNo` vai liberar a memória locada para um nó da árvore e seus filhos, verificando se o ponteiro para o nó é diferente de `NULL`, indicando que o nó existe e precisa ser destruído, em seguida entra em um loop `for` para interar sobre todos os filhos do nó (de 0 até `no->numChaves`) e chama a si mesma recursivamente para liberar a memória de cada fico percorrento toda subárvore enraizada no nó atual. Depois que todos os filhos foram processados, o nó atual é liberado com `free(no)`.

4. Casos de Teste

Para fim de casos de testes foi utilizado um arquivo .txt contendo as seguintes palavras

Computação	Geolocalização
Programação	Conexão
Desenvolvimento	Internet
Criatividade	BancoDados
Tecnologia	Ambiente
Inovação	Acessibilidade
Aplicativo	Automação
Software	Design
Hardware	Interativo
Inteligência	Funcionalidade
Dados	Eficiência
Processamento	Produtividade
Algoritmo	Mouse
Segurança	Teclado
Interface	Celulares
Usuário	Xiaomi
Gráficos	Android
	Anatel**

```
=====
                        ÁRVORE-B IMPLEMENTAÇÃO
=====
1. Carregar dados do arquivo
2. Inserir chave
3. Exibir árvore
4. Imprimir em ordem
5. Buscar chave
6. Destruir árvore
7. Sair
=====
```

A primeira opção retorna a função de exibir árvore para demonstrar o que ocorre passo a passo a cada inserção de chave linha no arquivo texto de forma dinâmica

```
Dados carregados do arquivo com sucesso!
Chave 'Computação' inserida no nó folha.
Árvore-B após inserção:
Computação
Chave 'Programação' inserida no nó folha.
Árvore-B após inserção:
Computação
Programação
Chave 'Desenvolvimento' inserida no nó folha.
Árvore-B após inserção:
Computação
Desenvolvimento
Programação
Chave 'Criatividade' inserida no nó folha.
Árvore-B após inserção:
Computação
Criatividade
Desenvolvimento
Programação
Chave 'Tecnologia' inserida no nó folha.
Árvore-B após inserção:
Computação
Criatividade
Desenvolvimento
Programação
Tecnologia
Dividido o filho 0 do nó pai.
Chave 'Inovação' inserida no nó folha.
Árvore-B após inserção:
    Computação
    Criatividade
Desenvolvimento
    Inovação
    Programação
    Tecnologia
Chave 'Aplicativo' inserida no nó folha.
```

O mesmo ocorre quando a opção de **inserir chave** é escolhida

```
Digite a chave (até 20 caracteres, sem espaços): Anatel
Chave 'Anatel' inserida no nó folha.
Árvore-B após inserção:
    Acessibilidade
    Algoritmo
Ambiente
    Anatel
    Aplicativo
    Automação
    BancoDados
    Celulares
Computação
    Conexão
    Criatividade
    Dados
Desenvolvimento
    Design
    Eficiência
    Funcionalidade
    Geolocalização
Gráficos
    Hardware
    Inovação
Inteligência
    Interativo
    Interface
Internet
    Mouse
    Processamento
    Produtividade
Programação
    Segurança
    Software
Teclado
    Tecnologia
    Usuário
    Xiaomi
Chave inserida com sucesso!
```

Já quando a opção de **exibir a árvore** de forma tabulada e estilizada para melhor legibilidade o programa retorna ao usuário

Escolha uma opção: 3

Árvore-B:

- Acessibilidade
- Algoritmo
- Ambiente
- Anatel
- Aplicativo
- Automação
- BancoDados
- Celulares
- Computação
- Conexão
- Criatividade
- Dados
- Desenvolvimento
- Design
- Eficiência
- Funcionalidade
- Geolocalização
- Gráficos
- Hardware
- Inovação
- Inteligência
- Interativo
- Interface
- Internet
- Mouse
- Processamento
- Produtividade
- Programação
- Segurança
- Software
- Teclado
- Tecnologia
- Usuário
- Xiaomi

E no caso de **imprimir a árvore em ordem**

Escolha uma opção: 4

Árvore-B em ordem:

Acessibilidade Algoritmo Ambiente Anatel Aplicativo Automação BancoDados Celulares Computação
Conexão Criatividade Dados Desenvolvimento Design Eficiência Funcionalidade Geolocalização
Gráficos Hardware Inovação Inteligência Interativo Interface Internet Mouse Processamento P
rodutividade Programação Segurança Software Teclado Tecnologia Usuário Xiaomi

Na função de **busca** foi testado uma chave que não está contida na árvore para demonstração

Digite a chave a buscar: Geometria
Chave Geometria não encontrada.

```
Digite a chave a buscar: Acessibilidade
Chave Acessibilidade encontrada!
Nó: Folha
Pai: Ambiente
Sem irmão esquerdo.
Irmão direito: Anatel
```

```
Escolha uma opção: Aplicativo
Digite a chave a buscar:
Chave Aplicativo encontrada!
Nó: Folha
Pai: Ambiente
Irmão esquerdo: Algoritmo
Irmão direito: Conexão
```

```
Chave Gráficos encontrada!
Nó: Interno
Pai: Desenvolvimento
Irmão esquerdo: Computação
Sem irmão direito.
```

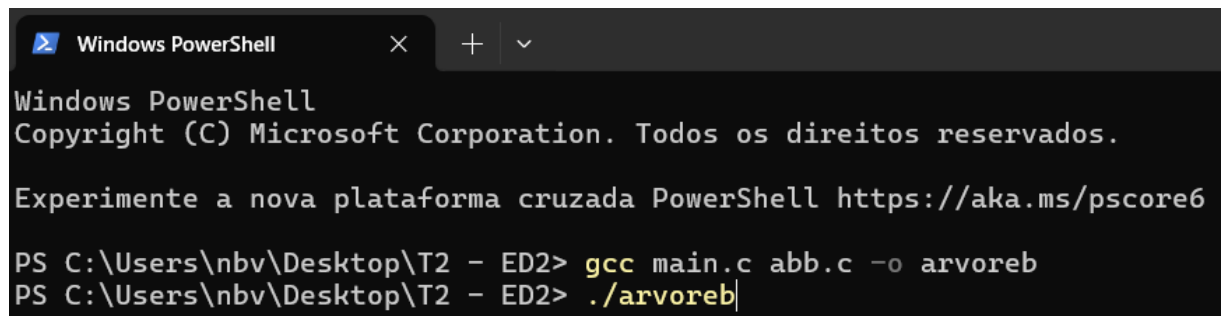
```
Digite a chave a buscar: Desenvolvimento
Chave Desenvolvimento encontrada!
Nó: Raiz
Pai: Sem pai
Sem irmão esquerdo.
Sem irmão direito.
```

```
Digite a chave a buscar: Interface
Chave Interface encontrada!
Nó: Folha
Pai: Gráficos
Irmão esquerdo: Inovação
Irmão direito: Mouse
```

E por fim quando a função de **destruir** a árvore é chamada

```
=====
ÁRVORE-B IMPLEMENTAÇÃO
=====
1. Carregar dados do arquivo
2. Inserir chave
3. Exibir árvore
4. Imprimir em ordem
5. Buscar chave
6. Destruir árvore
7. Sair
=====
Escolha uma opção: 6
Destruindo a árvore...
Árvore destruída com sucesso!
```

5. Instruções de Compilação

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Windows PowerShell' with standard window controls. The terminal text includes the PowerShell header, a copyright notice for Microsoft Corporation, a promotional message for the PowerShell Core 6 platform, and two command-line prompts. The first prompt shows the command 'gcc main.c abb.c -o arvoreb' being entered. The second prompt shows the command './arvoreb' being entered, with the cursor at the end of the line.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Experimente a nova plataforma cruzada PowerShell https://aka.ms/pscore6

PS C:\Users\nbv\Desktop\T2 - ED2> gcc main.c abb.c -o arvoreb
PS C:\Users\nbv\Desktop\T2 - ED2> ./arvoreb|
```

A compilação do código via terminal se dá pelos seguintes comandos:

```
gcc main.c abb.c -o arvoreb
```

```
./arvoreb
```