Roteiro 8

Sávio Francisco - 202050045

1 Árvores Binárias de Pesquisa (ABP)

1.1 Implementação utilizando o código fornecido

1.1.1 Implementação (.h)

```c
/*---------------- File: ABP.h  ------------------------+
|Arvore Binaria de Pesquisa (ABP)                      |
|                                    |
|                                    |
| Implementado por Guilherme C. Pena em 12/10/2023     |
+------------------------------------------------------+ */

#ifndef ABP_H
#define ABP_H

#include <stdio.h>
#include <stdlib.h>

typedef struct NO{
    int info;
    struct NO* esq;
    struct NO* dir;
}NO;

typedef struct NO* ABP;

NO* alocarNO(){
    return (NO*) malloc (sizeof(NO));
}

void liberarNO(NO* q){
    free(q);
}

ABP* criaABP(){
    ABP* raiz = (ABP*) malloc (sizeof(ABP));
    if(raiz != NULL)
        *raiz = NULL;
    return raiz;
}

void destroiRec(NO* no){
    if(no == NULL) return;
    destroiRec(no->esq);
    destroiRec(no->dir);
    liberarNO(no);
    no = NULL;
}

void destroiABP(ABP* raiz){
    if(raiz != NULL){
        destroiRec(*raiz);
        free(raiz);
    }
}

int estaVazia(ABP* raiz){
    if(raiz == NULL) return 0;
    return (*raiz == NULL);
}
```

```
int insereRec(NO** raiz, int elem){
    if(*raiz == NULL){
        NO* novo = alocarNO();
        if(novo == NULL) return 0;
        novo->info = elem;
        novo->esq = NULL; novo->dir = NULL;
        *raiz = novo;
    }else{
        if((*raiz)->info == elem){
            printf("Elemento Existente!\n");
            return 0;
        }
        if(elem < (*raiz)->info)
            return insereRec(&(*raiz)->esq, elem);
        else if(elem > (*raiz)->info)
            return insereRec(&(*raiz)->dir, elem);
    }

    return 1;
}

int insereIte(NO** raiz, int elem){
    NO *aux = *raiz, *ant = NULL;
    while (aux != NULL){
        ant = aux;
        if(aux->info == elem){
            printf("Elemento Existente!\n");
            return 0;
        }
        if(elem < aux->info) aux = aux->esq;
        else aux = aux->dir;
    }
    NO* novo = alocarNO();
    if(novo == NULL) return 0;
    novo->info = elem;
    novo->esq = NULL; novo->dir = NULL;
    if(ant == NULL){
        *raiz = novo;
    }else{
        if(elem < ant->info) ant->esq = novo;
        else ant->dir = novo;
    }
    return 1;
}

int insereElem(ABP* raiz, int elem){
    if(raiz == NULL) return 0;
    return insereRec(raiz, elem);
    //return insereIte(raiz, elem);
}

int pesquisaRec(NO** raiz, int elem){
    if(*raiz == NULL) return 0;
    if((*raiz)->info == elem) return 1;
    if(elem < (*raiz)->info)
        return pesquisaRec(&(*raiz)->esq, elem);
    else
        return pesquisaRec(&(*raiz)->dir, elem);
}

int pesquisaIte(NO** raiz, int elem){
    NO* aux = *raiz;
    while(aux != NULL){
        if(aux->info == elem) return 1;
        if(elem < aux->info)
            aux = aux->esq;
        else
            aux = aux->dir;
    }
    return 0;
}

int pesquisa(ABP* raiz, int elem){
    if(raiz == NULL) return 0;
    if(estaVazia(raiz)) return 0;
    return pesquisaRec(raiz, elem);
    //return pesquisaIte(raiz, elem);
```

```
}

int removeRec(NO** raiz, int elem){
    if(*raiz == NULL) return 0;
    if((*raiz)->info == elem){
        NO* aux;
        if((*raiz)->esq == NULL && (*raiz)->dir == NULL){
            //Caso 1 - NO sem filhos
            printf("Caso 1: Liberando %d..\n", (*raiz)->info);
            liberarNO(*raiz);
            *raiz = NULL;
        }else if((*raiz)->esq == NULL){
            //Caso 2.1 - Possui apenas uma subarvore direita
            printf("Caso 2.1: Liberando %d..\n", (*raiz)->info);
            aux = *raiz;
            *raiz = (*raiz)->dir;
            liberarNO(aux);
        }else if((*raiz)->dir == NULL){
            //Caso 2.2 - Possui apenas uma subarvore esquerda
            printf("Caso 2.2: Liberando %d..\n", (*raiz)->info);
            aux = *raiz;
            *raiz = (*raiz)->esq;
            liberarNO(aux);
        }else{
            //Caso 3 - Possui as duas subarvores (esq e dir)
            //Duas estrategias:
            //3.1 - Substituir pelo NO com o MAIOR valor da subarvore esquerda
            //3.2 - Substituir pelo NO com o MENOR valor da subarvore direita
            printf("Caso 3: Liberando %d..\n", (*raiz)->info);
            //Estrategia 3.1:
            NO* Filho = (*raiz)->esq;
            while(Filho->dir != NULL)//Localiza o MAIOR valor da subarvore esquerda
                Filho = Filho->dir;
            (*raiz)->info = Filho->info;
            Filho->info = elem;
            return removeRec(&(*raiz)->esq, elem);
        }
        return 1;
    }else if(elem < (*raiz)->info)
        return removeRec(&(*raiz)->esq, elem);
    else
        return removeRec(&(*raiz)->dir, elem);
}

NO* removeAtual(NO* atual){

    NO* no1, *no2;
    //Ambos casos no if(atual->esq == NULL)
    //Caso 1 - NO sem filhos
    //Caso 2.1 - Possui apenas uma subarvore direita
    if(atual->esq == NULL){
        no2 = atual->dir;
        liberarNO(atual);
        return no2;
    }
    //Caso 3 - Possui as duas subarvores (esq e dir)
    //Estrategia:

    no1 = atual;
    no2 = atual->esq;
    while(no2->dir != NULL){
        no1 = no2;
        no2 = no2->dir;
    }
    if(no1 != atual){
        no1->dir = no2->esq;
        no2->esq = atual->esq;
    }
    no2->dir = atual->dir;
    liberarNO(atual);
    return no2;
}

int removeIte(NO** raiz, int elem){
    if(*raiz == NULL) return 0;
    NO* atual = *raiz, *ant = NULL;
    while(atual != NULL){
        if(elem == atual->info){
```

```
            if(atual == *raiz)
                *raiz = removeAtual(atual);
            else{
                if(ant->dir == atual)
                    ant->dir = removeAtual(atual);
                else
                    ant->esq = removeAtual(atual);
            }
            return 1;
        }
        ant = atual;
        if(elem < atual->info)
            atual = atual->esq;
        else
            atual = atual->dir;
    }
    return 0;
}

int removeElem(ABP* raiz, int elem){
    if(pesquisa(raiz, elem) == 0){
        printf("Elemento inexistente!\n");
        return 0;
    }
    //return removeRec(raiz, elem);
    return removeIte(raiz, elem);
}

void em_ordem(NO* raiz, int nivel){
    if(raiz != NULL){
        em_ordem(raiz->esq, nivel+1);
        printf("[%d, %d] ", raiz->info,nivel);
        em_ordem(raiz->dir, nivel+1);
    }
}

void pre_ordem(NO* raiz, int nivel){
    if(raiz != NULL){
        printf("[%d, %d] ", raiz->info,nivel);
        pre_ordem(raiz->esq, nivel+1);
        pre_ordem(raiz->dir, nivel+1);
    }
}

int qtd_elementos(NO *raiz, int *cont){

    if(raiz != NULL){
        (*cont)++;
        qtd_elementos(raiz->esq, cont);
        qtd_elementos(raiz->dir, cont);
    }

    return *cont;
}

void pos_ordem(NO* raiz, int nivel){
    if(raiz != NULL){
        pos_ordem(raiz->esq, nivel+1);
        pos_ordem(raiz->dir, nivel+1);
        printf("[%d, %d] ", raiz->info,nivel);
    }
}

void imprime(ABP* raiz){
    if(raiz == NULL) return;
    if(estaVazia(raiz)){
        printf("Arvore Vazia!\n");
        return;
    }
    printf("\nEm Ordem: "); em_ordem(*raiz, 0);
    printf("\nPre Ordem: "); pre_ordem(*raiz, 0);
    printf("\nPos Ordem: "); pos_ordem(*raiz, 0);
    printf("\n");
}


#endif
```

A complexidade para a função de procura maior ou menor é O(h) onde h é a altura da árvore. Caso a árvores se encontre desbalanceada, sua procura se torna linear O(n) e quando se encontram balanceadas é O(logN).

1.1.2 Implementação (Main.c)

```
#include "Abp.h"
#define endl printf("\n")

int main(){

    ABP* A;;
    int op;
    int *cont = (int*) calloc(1,sizeof(int));

    do{

        int elem;

        printf("0 - Sair");
        endl;
        printf("1 - Criar ABP");
        endl;
        printf("2 - Inserir um elemento");
        endl;
        printf("3 - Buscar um elemento");
        endl;
        printf("4 - Remover um elemento");
        endl;
        printf("5 - Imprimir a ABP em ordem");
        endl;
        printf("6 - Imprimir a ABP em pre-ordem");
        endl;
        printf("7 - Imprimir a ABP em pos-ordem");
        endl;
        printf("8 - Mostrar a quantidade de nos na ABP");
        endl;
        printf("9 - Destruir a ABP");
        endl;

        scanf("%d", &op);

        switch (op) {
            case 1:
               A = criaABP();
               printf("ABP criada com sucesso");
               endl;
                break;
            case 2:
                printf("Digite o elemento que deseja inserir:");
                scanf("%d", &elem);
                insereElem(A, elem);
                break;
            case 3:
                printf("Digite o elemento que deseja procurar:");
                scanf("%d", &elem);
                pesquisa(A, elem);
                break;
            case 4:
                printf("Digite o elemento que deseja remover:");
                scanf("%d",&elem);
                removeElem(A, elem);
                break;
            case 5:
                em_ordem(*A, 0);
                endl;
                break;
            case 6:
                pre_ordem(*A, 0);
                endl;
                break;
            case 7:
                pos_ordem(*A, 0);
                endl;
                break;
```

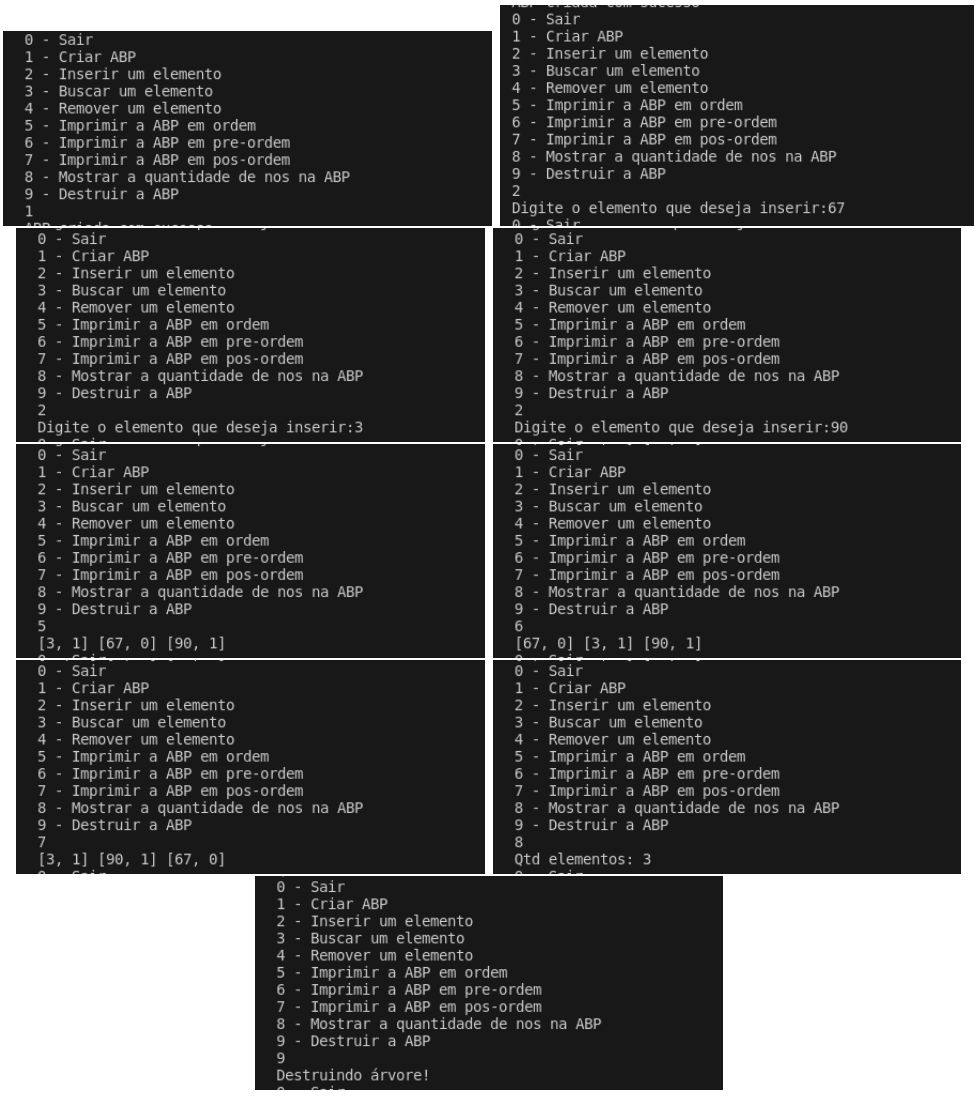
```
                case 8:
                    *cont = 0;
                    printf("Qtd elementos: %d", qtd_elementos(*A, cont));
                    endl;
                    break;
                case 9:
                    printf("Destruindo arvore!");
                    endl;
                    destroiABP(A);
                    break;
            }

        }while(op != 0);

        free(cont);

        return 0;
    }
```

1.1.3 Saída do programa:



1.2 Implementação com a *struct* Aluno

1.2.1 Implementação (.h)

```
/*---------------- File: ABP.h  ------------------------+
|Arvore Binaria de Pesquisa (ABP) Modificada            |
|                                     |                 |
|                                     |                 |
| Implementado com base no codigo de Guilherme C. Pena  |
|                                                       |
+-------------------------------------------------------+ */

#ifndef ABP_H
#define ABP_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct ALUNO{

    char nome[150];
    int mat;
    double nota;

}Aluno;

typedef struct NO{

    Aluno aluno;
    struct NO* esq;
    struct NO* dir;

}NO;

typedef struct NO* ABP;

NO* alocarNO(){
    return (NO*) malloc (sizeof(NO));
}

void liberarNO(NO* q){
    free(q);
}

ABP* criaABP(){
    ABP* raiz = (ABP*) malloc (sizeof(ABP));
    if(raiz != NULL)
        *raiz = NULL;
    return raiz;
}

void destroiRec(NO* no){
    if(no == NULL) return;
    destroiRec(no->esq);
    destroiRec(no->dir);
    liberarNO(no);
    no = NULL;
}

void destroiABP(ABP* raiz){
    if(raiz != NULL){
        destroiRec(*raiz);
        free(raiz);
    }
}

int estaVazia(ABP* raiz){
    if(raiz == NULL) return 0;
    return (*raiz == NULL);
}

int comparaAluno(Aluno a1, Aluno a2) {
    return strcmp(a1.nome, a2.nome);
}
```

```
    int insereRec(NO** raiz, Aluno aluno){

        if(*raiz == NULL){

            NO* novo = alocarNO();

            if(novo == NULL){
                return 0;
            }

            novo->aluno = aluno;
            novo->esq = NULL;
            novo->dir = NULL;
            *raiz = novo;

        }else{

            int cpm = comparaAluno(aluno, (*raiz)->aluno);

            //se for zero significa que o nomes sao iguais
            if(cpm == 0){
                printf("Elemento Existente!\n");
                return 0;
            }
            //o aluno no qual se encontra na raiz e maior que o elemento que desja
inserir
            if(cpm < 0){
                return insereRec(&(*raiz)->esq, aluno);

            }else{//o elemento no qual quero inserir e maior que minha raiz, entao a
inser  o e feita na direta
                return insereRec(&(*raiz)->dir, aluno);
            }
        }
        return 1;
    }

    int insereIte(NO** raiz, Aluno aluno){
        NO *aux = *raiz, *ant = NULL;

        while (aux != NULL){
            ant = aux;

            int cpm = comparaAluno(aluno, aux->aluno);

            if(cpm == 0){
                printf("Elemento Existente!\n");
                return 0;
            }
            if(cpm < 0) {
                aux = aux->esq;
            }else {
                aux = aux->dir;
            }
        }

        NO* novo = alocarNO();

        if(novo == NULL) return 0;

        novo->aluno = aluno;
        novo->esq = NULL;
        novo->dir = NULL;

        if(ant == NULL){
            *raiz = novo;

        }else{

            int cpm = comparaAluno(aluno, ant->aluno);

            if(cpm < 0) {
                ant->esq = novo;
            } else {
                ant->dir = novo;
            }
        }
        return 1;
```

```
}

int insereElem(ABP* raiz, Aluno aluno){

    if(raiz == NULL) return 0;

    return insereRec(raiz, aluno);
    //return insereIte(raiz, elem);
}

int pesquisaRec(NO** raiz, Aluno aluno){

    if(*raiz == NULL) return 0;

    int cpm = comparaAluno(aluno, (*raiz)->aluno);

    if(cpm == 0) {
        return 1;
    }
    if(cpm < 0){
        return pesquisaRec(&(*raiz)->esq, aluno);
    }else{
        return pesquisaRec(&(*raiz)->dir, aluno);
    }
}

int pesquisaIte(NO** raiz, Aluno aluno){
    NO* aux = *raiz;

    while(aux != NULL){

        int cpm = comparaAluno(aluno, aux->aluno);

        if(cpm == 0){
            return 1;
        }
        if(cpm < 0){
            aux = aux->esq;

        }else{
            aux = aux->dir;
        }
    }
    return 0;
}
void maiorNota(NO *raiz){

    if(raiz == NULL){
        return ;
    }

    while (raiz->dir != NULL){

        raiz = raiz->dir;
    }

    printf("Aluno com a maior nota:\n");
    printf("Nome: %s\n", raiz->aluno.nome);
    printf("Matricula: %d\n", raiz->aluno.mat);
    printf("Nota: %.2f\n", raiz->aluno.nota);

}
void menorNota(NO *raiz){

    if(raiz == NULL){
        return;
    }
    while (raiz ->esq != NULL){
        raiz = raiz->esq;
    }

    printf("Aluno com a menor nota:\n");
    printf("Nome: %s\n", raiz->aluno.nome);
    printf("Matr cula: %d\n", raiz->aluno.mat);
    printf("Nota: %.2f\n", raiz->aluno.nota);
}

int pesquisa(ABP* raiz, Aluno aluno){
```

```
    if(raiz == NULL) return 0;

    if(estaVazia(raiz)) return 0;

    return pesquisaRec(raiz, aluno);
    //return pesquisaIte(raiz, elem);
}

int removeRec(NO** raiz, Aluno aluno){

    if(*raiz == NULL) return 0;

    int cpm = comparaAluno(aluno, (*raiz)->aluno);

    if(cpm == 0){

        NO* aux;

        if((*raiz)->esq == NULL && (*raiz)->dir == NULL){
            //Caso 1 - NO sem filhos

            printf("Caso 1: Liberando %s..\n", (*raiz)->aluno.nome);
            printf("Caso 1: Liberando %d..\n", (*raiz)->aluno.mat);
            printf("Caso 1: Liberando %lf..\n", (*raiz)->aluno.nota);

            liberarNO(*raiz);
            *raiz = NULL;

        }else if((*raiz)->esq == NULL){
            //Caso 2.1 - Possui apenas uma subarvore direita

            printf("Caso 2.1: Liberando %s..\n", (*raiz)->aluno.nome);
            printf("Caso 2.1: Liberando %d..\n", (*raiz)->aluno.mat);
            printf("Caso 2.1: Liberando %lf..\n", (*raiz)->aluno.nota);

            aux = *raiz;
            *raiz = (*raiz)->dir;

            liberarNO(aux);
        }else if((*raiz)->dir == NULL){
            //Caso 2.2 - Possui apenas uma subarvore esquerda

            printf("Caso 2.2 Liberando %s..\n", (*raiz)->aluno.nome);
            printf("Caso 2.2 Liberando %d..\n", (*raiz)->aluno.mat);
            printf("Caso 2.2 Liberando %lf..\n", (*raiz)->aluno.nota);

            aux = *raiz;
            *raiz = (*raiz)->esq;

            liberarNO(aux);
        }else{

            //Caso 3 - Possui as duas subarvores (esq e dir)
            //Duas estrategias:
            //3.1 - Substituir pelo NO com o MAIOR valor da subarvore esquerda
            //3.2 - Substituir pelo NO com o MENOR valor da subarvore direita

            printf("Caso 3: Liberando %s..\n", (*raiz)->aluno.nome);
            printf("Caso 3: Liberando %d..\n", (*raiz)->aluno.mat);
            printf("Caso 3: Liberando %lf..\n", (*raiz)->aluno.nota);

            //Estrategia 3.1:
            NO* Filho = (*raiz)->esq;

            while(Filho->dir != NULL){//Localiza o MAIOR valor da subarvore esquerda
                Filho = Filho->dir;
            }

            (*raiz)->aluno = Filho->aluno;
            Filho->aluno = aluno;

            return removeRec(&(*raiz)->esq, aluno);
        }
        return 1;

    }else if(cpm  < 0){
        return removeRec(&(*raiz)->esq, aluno);
```

```
        }
        else{
            return removeRec(&(*raiz)->dir, aluno);
        }
    }

    NO* removeAtual(NO* atual){

        NO* no1, *no2;
        //Ambos casos no if(atual->esq == NULL)
        //Caso 1 - NO sem filhos
        //Caso 2.1 - Possui apenas uma subarvore direita

        if(atual->esq == NULL){
            no2 = atual->dir;
            liberarNO(atual);
            return no2;
        }

        //Caso 3 - Possui as duas subarvores (esq e dir)
        //Estrategia:

        no1 = atual;
        no2 = atual->esq;

        while(no2->dir != NULL){
            no1 = no2;
            no2 = no2->dir;
        }
        if(no1 != atual){
            no1->dir = no2->esq;
            no2->esq = atual->esq;
        }
        no2->dir = atual->dir;

        liberarNO(atual);

        return no2;
    }

    int removeIte(NO** raiz, Aluno aluno){

        if(*raiz == NULL) return 0;

        NO* atual = *raiz, *ant = NULL;

        int cpm = comparaAluno(aluno, atual->aluno);

        while(atual != NULL){

            if(cpm == 0){

                if(atual == *raiz){
                    *raiz = removeAtual(atual);

                }else{

                    if(ant->dir == atual){
                        ant->dir = removeAtual(atual);
                    }
                    else{
                        ant->esq = removeAtual(atual);
                    }
                }
                return 1;
            }
            ant = atual;

            if(cpm < 0){
                atual = atual->esq;

            }else{
                atual = atual->dir;
            }
        }
        return 0;
    }
```

```
    int removeElem(ABP *raiz, Aluno aluno){

        if(pesquisa(raiz, aluno) == 0){
            printf("Elemento inexistente!\n");
            return 0;
        }
        //return removeRec(raiz, elem);
        return removeIte(raiz, aluno);
    }

    void em_ordem(NO *raiz, int nivel){

        if(raiz != NULL){
            em_ordem(raiz->esq, nivel+1);
            printf("[%s, %d, %lf, %d] ", raiz->aluno.nome, raiz->aluno.mat, raiz->aluno.
nota, nivel);
            em_ordem(raiz->dir, nivel+1);
        }
    }

    void pre_ordem(NO* raiz, int nivel){

        if(raiz != NULL){
            printf("[%s, %d, %lf, %d] ", raiz->aluno.nome, raiz->aluno.mat, raiz->aluno.
nota, nivel);
            pre_ordem(raiz->esq, nivel+1);
            pre_ordem(raiz->dir, nivel+1);
        }
    }

    int qtd_elementos(NO *raiz, int *cont){

        if(raiz != NULL){
            (*cont)++;
            qtd_elementos(raiz->esq, cont);
            qtd_elementos(raiz->dir, cont);
        }

        return *cont;
    }

    void pos_ordem(NO* raiz, int nivel){

        if(raiz != NULL){
            pos_ordem(raiz->esq, nivel+1);
            pos_ordem(raiz->dir, nivel+1);
            printf("[%s, %d, %lf, %d] ", raiz->aluno.nome, raiz->aluno.mat, raiz->aluno.
nota, nivel);
        }
    }

    void imprime(ABP* raiz){

        if(raiz == NULL) return;

        if(estaVazia(raiz)){
            printf("Arvore Vazia!\n");
            return;
        }

        printf("\nEm Ordem: "); em_ordem(*raiz, 0);
        printf("\nPre Ordem: "); pre_ordem(*raiz, 0);
        printf("\nPos Ordem: "); pos_ordem(*raiz, 0);
        printf("\n");
    }

    #endif
```

1.2.2 Implementação (Main.c)

```
#include "Abp.h"
#include <stdio.h>

#define endl printf("\n")

int main(){

    ABP* A;
    Aluno aluno, busca;

    int *cont = (int*) calloc(1,sizeof(int));
    char nome[100];
    double nota = 0.0;
    int op, mat;

    do{

        int elem;

        printf("0 - Sair");
        endl;
        printf("1 - Criar ABP");
        endl;
        printf("2 - Inserir um Aluno");
        endl;
        printf("3 - Buscar um Aluno pelo nome e imprimir suas informa  es");
        endl;
        printf("4 - Remover um Aluno pelo nome");
        endl;
        printf("5 - Imprimir a ABP em ordem");
        endl;
        printf("6 - Imprimir as informa  es do aluno com a maior nota");
        endl;
        printf("7 - Imprimir as informa  es do aluno com a menor nota");
        endl;
        printf("8 - Mostrar a quantidade de nos na ABP");
        endl;
        printf("9 - Destruir a ABP");
        endl;

        scanf("%d", &op);

        switch (op) {
            case 1:
               A = criaABP();
               printf("ABP criada com sucesso");
               endl;
                break;
            case 2:

                printf("Digite o nome do aluno:");
                endl;
                while (getchar() != '\n');
                fgets(nome, sizeof(nome), stdin);

                printf("Digite a matricula do aluno:");
                endl;
                scanf("%d", &mat);

                printf("Digite a nota do aluno:");
                endl;
                scanf("%lf", &nota);

                strcpy(aluno.nome,nome);
                aluno.mat = mat;
                aluno.nota = nota;

                insereElem(A, aluno);
                break;
            case 3:

                printf("Digite o nome do aluno que deseja procurar:");
                while (getchar() != '\n');
                fgets(nome, sizeof(nome), stdin);
```

```
                    strcpy(aluno.nome, nome);

                    if(pesquisa(A, aluno)){
                        printf("Nome aluno: %s \nMatricula: %d \nNota: %lf", aluno.nome,
aluno.mat, aluno.nota);
                        endl;
                    }else{
                        printf("Aluno nao encontrado");
                        endl;
                    }
                    break;
                case 4:
                    printf("Digite o nome do aluno que deseja remover:");
                    while (getchar() != '\n');
                    fgets(nome, sizeof(nome), stdin);

                    strcpy(aluno.nome, nome);

                    removeElem(A, aluno);
                    break;
                case 5:
                    em_ordem(*A, 0);
                    endl;
                    break;
                case 6:
                    menorNota(*A);
                    endl;
                    break;
                case 7:
                    maiorNota(*A);
                    endl;
                    break;
                case 8:
                    *cont = 0;
                    printf("Qtd elementos: %d", qtd_elementos(*A, cont));
                    endl;
                    break;
                case 9:
                    printf("Destruindo arvore!");
                    endl;
                    destroiABP(A);
                    break;
            }

        }while(op != 0);

        free(cont);

        return 0;
}
```

1.2.3 Saída do programa: