Relatório tarefa 2 - Lívia Lutz dos Santos- 2211055 - 3WA

Objetivo:

Criar um programa que inicialmente gera 10 valores inteiros aleatoriamente no intervalo [1,20] e os exiba na ordem que foram gerados.

Após isso, o programa deve inserir esses números numa árvore binária por nível e em uma árvore binária de busca, exibindo essas árvores no percurso pré - ordem junto com as informações completas sobre todos os nós da árvore,por exemplo os ponteiros para cada nó (direito e esquerdo) e o valor guardado na raiz que está sendo exibida no momento .

Então, verificamos se as árvores são do tipo árvore binária de busca e calculamos sua altura independentemente de serem uma abb ou não.

Por fim, ocorre a troca dos nós da esquerda pelos da direita de cada nó nas árvores e imprimimos as novas árvores no percurso pré-ordem.

Estrutura do programa:

O módulo arvBin.h possui o protótipo de todas as funções usadas no manuseio de árvores binárias e o módulo abb.h possui as funções usadas para o manuseio de árvores binárias de busca.

```c
/*Struct da arvore binaria*/

typedef struct arv_bin arvBin;

/*Funcao que inicializa cada no da arvore binaria como NULL*/

arvBin* criaArv(void);

/*Funcao que verifica se a arvore binaria esta vazia*/

int arv_vazia(arvBin* a);

/*Funcao que cria cada no da arvore binaria*/

arvBin* arv_criaRaiz(int n);

/*Funcao que imprime a arvore binaria em pre - ordem*/

void arvImprime(arvBin* a);

/*Funcao que calcula a altura da arvore binaria*/

int arvBin_altura(arvBin* no);

/*Funcao que insere um no na arvore binaria por nivel*/

arvBin* insere(arvBin* a, int v[], int i, int n);

/*Funcao que verifica se a arvore binaria e uma abb*/

int eh_abb(arvBin* a);

/*Funcao que troca todos os nos esquerdos da arvore pelos direitos e vice - versa*/

arvBin* trocaDirEsq(arvBin* a);

/*Funcao que libera a memoria ocupada pela arvore binaria*/

arvBin* arv_libera(arvBin* a);
```

```c
/*Struct da abb*/

typedef struct abb Abb;

/*Função que inicializa cada nó da abb como NULL*/

Abb* criaAbbVazia(void);

/*Função que insere um nó na abb*/

Abb* insereAbb(int i, Abb* a);

/*Função que indica se abb é vazia*/

int abbVazia(Abb* a);

/*Função que imprime a abb em pré - ordem*/

void abb_imprime(Abb* a);

/*Função que calcula a altura de uma abb*/

int arvbb_altura(Abb* no);

/*Função que verifica se uma abb é uma abb*/

int is_abb(Abb* a);

/*Função que troca os nós esquerdos e direitos de uma abb*/

Abb* swapLeftRight(Abb* a);

/*Função que libera a memória ocupada por uma abb*/

Abb* abb_libera(Abb* a);
```

-> TAD de uma árvore binária:

Inclui um campo do tipo int para armazenar o número inteiro que será parte de cada nó da árvore, um ponteiro para o nó à esquerda e um ponteiro para o nó à direita, ambos do tipo arvBin como definido em arvBin.h

```c
/*No struct de uma árvore binária temos um inteiro que será inserido em cada nó,
um ponteiro para o nó à esquerda e um para o nó à direita*/

struct arv_bin {
    int num;
    arvBin* esq;
    arvBin* dir;
};
```

->TAD de uma árvore binária de busca (abb):

Possui uma estrutura similar ao TAD da árvore binária descrito acima.

```c
/*No struct de uma abb temos um inteiro que será inserido em cada nó,
um ponteiro para o nó a esquerda e um para o nó a direita*/

struct abb {
    int n;
    Abb* dir;
    Abb* esq;
};
```

Descrevendo as funções implementadas:

Árvore Binária e ABB - funções comuns:

Nas funções descritas abaixo, o único aspecto que as diferem é o tipo do ponteiro passado como parâmetro e o tipo retornado pelas funções, nas funções de árvore binária, o tipo é arvBin, enquanto nas de abb o tipo é Abb.

-> Função de criar a árvore:

Retorna um ponteiro NULL para o tipo de árvore criada.

```c
/*criaArv inicializa cada nó da árvore como NULL*/

arvBin* criaArv(void) {
    return NULL;
}
```

```c
/*criaAbbVazia serve para inicializar cada nó da árvore como NULL*/

Abb* criaAbbVazia(void) {
    return NULL;
}
```

->Função de verificar se a árvore está vazia:

Recebe um ponteiro do tipo arvBin ou Abb e retorna um inteiro correspondente a igualdade *árvore == NULL, 1 para True e 0 para False.

```c
/*arv_vazia verifica se a árvore binária passada como parâmetro está vazia, retorna 1 (True) se estiver,
senão, retorna 0 (False)*/

int arv_vazia(arvBin* a) {
    return a == NULL;
}
```

```c
/*abbVazia verifica se a abb passada como parâmetro está vazia, retorna 1 (True) se estiver,
senão, retorna 0 (False)*/

int abbVazia(Abb* a) {
    return a == NULL;
}
```

-> Função de imprimir os elementos em uma árvore:

Recebe um ponteiro do tipo arvBin ou Abb.

Utiliza a técnica de imprimir uma árvore em pré-ordem (Primeiro imprime as informações do nó raiz, seguido de sua sub - árvore esquerda e então sua sub - árvore direita).

Imprime o ponteiro do nó atual, o inteiro armazenado no nó, os ponteiros dos nós à esquerda e direita, respectivamente. Para imprimir informações dos nós usei o formato %p na função printf.

A função é recursiva, portanto para imprimir as sub - árvores chamei a função de novo porém usei cada uma como parâmetro para as chamadas.

```c
/*arvImprime imprime a árvore binária em pré- ordem, ou seja, primeiro as informações no nó raiz, seguido dos nós à esquerda e por fim os nós à direita*/

void arvImprime(arvBin* a){
    /*Até o final da árvore, imprimimos suas informações*/

    if (!arv_vazia(a)) {
        /*Imprimindo as informações do nó raiz, incluindo seu valor inteiro e os ponteiros do nó à esquerda, à direita e seu próprio ponteiro*/

        printf("No:%p Esquerda:%p Valor:%d Direita:%p\n", a, a->esq, a->num, a->dir);

        /*Imprime as informações dos nós à esquerda recursivamente*/

        arvImprime(a->esq);

        /*Imprime as informações dos nós à direita recursivamente*/

        arvImprime(a->dir);
    }
}
```

```c
/*abb_imprime imprime a abb em pré - ordem, ou seja, primeiro as informações no nó raiz, seguido dos nós à esquerda e por fim os nós à direita*/

void abb_imprime(Abb* a){
    /*Imprimindo as informações do nó raiz, incluindo seu valor inteiro e os ponteiros do nó à esquerda, à direita e seu próprio ponteiro*/

    printf("No: %p Esquerda:%p valor:%d direita:%p\n", a, a->esq, a->n, a->dir);

    /*Imprime as informações dos nós à esquerda recursivamente*/

    if (a->esq != NULL) {
        abb_imprime(a->esq);
    }

    /*Imprime as informações dos nós à direita recursivamente*/

    if (a->dir != NULL) {
        abb_imprime(a->dir);
    }
}
```

->Função de calcular a altura de uma árvore:

Recebe um ponteiro do tipo arvBin ou Abb e retorna o valor inteiro correspondente a altura da árvore passada como parâmetro.

Se a árvore for vazia, sua altura será 0, portanto neste caso a função retornará 0.

Guardamos as alturas das sub - árvores à esquerda e à direita em 2 variáveis e comparamos, usando a chamada recursiva da função para cada sub - árvore. Se a altura da esquerda for menor retornamos a altura da direita + 1(Adicionamos +1 pois a contagem começa em 0, e apesar do último nível direito da árvore não estar completo, ele ainda existe e deve ser considerado no cálculo da altura da árvore, por isso consideramos a altura direita ao invés da esquerda), senão a função retorna a altura da esquerda + 1 (mesmo raciocínio da direita).

```c
/*Calcula a altura da árvore binária*/

int arvBin_altura(arvBin* no) {
    /*Se a árvore for vazia, a altura é 0*/

    if (arv_vazia(no)) {
        return 0;
    }

    /*Guardamos as alturas das sub - árvores à esquerda e direita nas variáveis h_esq e h_dir respectivamente
    nas chamadas recursivas da função*/

    else {
        int h_esq = arvBin_altura(no->esq);
        int h_dir = arvBin_altura(no->dir);

        /*Se a altura da esquerda for menor, retornamos o valor da direita + 1*/

        if (h_esq < h_dir) {
            return h_dir + 1;
        }

        /*Senão retornamos o valor da esquerda + 1*/

        else {
            return h_esq + 1;
        }
    }
}
```

```c
/*Calcula a altura da abb*/

int arvbb_altura(Abb* no) {
    /*Se a árvore for vazia, a altura é 0*/

    if (abbVazia(no)) {
        return 0;
    }

    /*Guardamos as alturas das sub - árvores à esquerda e direita nas variáveis h_esq e h_dir respectivamente
    nas chamadas recursivas da função*/

    else {
        int h_esq = arvbb_altura(no->esq);
        int h_dir = arvbb_altura(no->dir);

        /*Se a altura da esquerda for menor, retornamos o valor da direita + 1*/

        if (h_esq < h_dir) {
            return h_dir + 1;
        }

        /*Senão retornamos o valor da esquerda + 1*/

        else {
            return h_esq + 1;
        }
    }
}
```

-> Função que verifica se uma árvore é uma abb:

Recebe um ponteiro do tipo arvBin ou Abb e retorna um valor inteiro correspondente à igualdade (1 para True e 0 para False).

Se a árvore passada como parâmetro é vazia ela é considerada uma abb, pois não há nós para comparar, a função retorna 1.

Guardamos o resultado da comparação das sub árvores à direita e à esquerda em 2 variáveis, usando a chamada recursiva da função para cada sub - árvore, se as 2 forem True, a árvore é uma abb e a função retorna 1, senão retorna 0.

Para ser uma abb, os valores à esquerda devem ser menores que o valor em seu nó raiz e os valores à direita devem ser maiores que o valor em seu nó raiz.

```c
/*Verifica se a árvore binária é uma abb*/

int eh_abb(arvBin* a) {
    /*Se a árvore for vazia, ela ainda é considerada uma abb
    já que não possui nós para comparação*/

    if (arv_vazia(a)) {
        return 1;
    }

    /*Se o inteiro à esquerda for maior, a árvore não está em ordem e, portanto
    não é uma abb, a função retorna 0 (False)*/

    if (a->esq != NULL && a->esq->num > a->num) {
        return 0;
    }

    /*Se o inteiro à direita for menor, a árvore não está em ordem e a função retorna 0*/

    if (a->dir != NULL && a->dir->num < a->num) {
        return 0;
    }

    /*Usamos as variáveis arvEsq e arvDir para guardar o valor das chamadas recursivas da função
    para a sub - árvore à esquerda e à direita respectivamente*/

    int arvEsq = eh_abb(a->esq);
    int arvDir = eh_abb(a->dir);

    /*Se ambas as sub - árvores forem uma abb (inteiros forem ambos == 1 (True)), então a função retorna 1 (True)
    pois a árvore toda é uma abb*/

    if (arvEsq == 1 && arvDir == 1) {
        return 1;
    }

    /*Caso contrário, a árvore não é uma abb e a função retorna 0 (False)*/

    else {
        return 0;
    }
}
```

```c
'*Verifica se a abb é uma abb*/

int is_abb(Abb* a) {

    /*Se a abb for vazia, ela ainda é considerada uma abb já que não possui nós para a comparação*/

    if (abbVazia(a)) {
        return 1;
    }

    /*Se o inteiro à esquerda for maior, a árvore não está em ordem e, portanto
    não é uma abb, a função retorna 0 (False)*/

    if (a->esq != NULL && a->esq->n > a->n) {
        return 0;
    }

    /*Se o inteiro à direita for menor, a árvore não está em ordem e a função retorna 0*/

    if (a->dir != NULL && a->dir->n < a->n) {
        return 0;
    }

    /*Usamos as variáveis abbEsq e abbDir para guardar o valor das chamadas recursivas da função
    para a sub árvore à esquerda e à direita respectivamente*/

    int abbEsq = is_abb(a->esq);
    int abbDir = is_abb(a->dir);

    /*Se ambas as sub árvores forem uma abb (inteiros forem ambos == 1 (True)),
    entao a função retorna 1 (True) pois a árvore toda é uma abb*/

    if (abbEsq == 1 && abbDir == 1) {
        return 1;
    }

    /*Caso contrário, a árvore não é uma abb e a função retorna 0 (False)*/

    else {
        return 0;
    }
}
```

->Função de trocar todos os nós esquerdos pelos direitos e vice - versa:

Recebe um ponteiro do tipo arvBin ou Abb.

Se a árvore for vazia, retornamos ela mesma pois não há nós para trocar.

Fazemos 2 chamadas recursivas da função para cada sub - árvore e guardamos seus resultados em 2 ponteiros para árvore binária ou abb.

Fazemos a sub - árvore à esquerda apontar para o resultado da troca da subárvore à direita e vice - versa e a função retorna a árvore atualizada.

```c
/*Troca todos os nós da esquerda pelos da direita e vice - versa, retornando a árvore atualizada*/

arvBin* trocaDirEsq(arvBin* a) {
    /*Se a árvore for vazia, retornamos a árvore*/

    if (arv_vazia(a)) {
        return a;
    }

    /*Guardamos o resultado da troca das sub árvores esquerda e direita em duas variáveis
    de ponteiro para árvore binária esq e dir respectivamente*/

    arvBin*esq = trocaDirEsq(a->esq);
    arvBin*dir = trocaDirEsq(a->dir);

    /*Fazemos com que a sub - árvore a esquerda seja o resultado da troca da direita e vice - versa*/

    a->esq = dir;
    a->dir = esq;

    /*Retornamos a árvore atualizada*/

    return a;
}
```

```c
/*Troca todos os nós da esquerda pelos da direita e vice - versa, retornando a árvore atualizada*/

Abb* swapLeftRight(Abb* a) {

    /*Se a árvore for vazia, retornamos a árvore*/

    if (abbVazia(a)) {
        return a;
    }

    /*Guardamos o resultado da troca das sub - árvores esquerda e direita em duas variáveis
    de ponteiro para abb left e right respectivamente*/

    Abb* left = swapLeftRight(a->esq);
    Abb* right = swapLeftRight(a->dir);

    /*Fazemos com que a sub - árvore à esquerda seja o resultado da troca da direita e vice versa*/

    a->esq = right;
    a->dir = left;

    /*Retornamos a árvore atualizada*/

    return a;
}
```

-> Função de liberar a memória ocupada pela árvore:

Recebe um ponteiro do tipo arvBin ou Abb.

Se a árvore for vazia, a função retorna NULL pois não haverá espaço para liberar.

Caso contrário fazemos 2 chamadas recursivas da função para liberar a memória de cada sub - árvore e por fim liberar a memória do nó raiz.

A função retorna a árvore atualizada (NULL caso liberar toda a memória).

```c
/*Libera a memória ocupada pela árvore binária*/

arvBin* arv_libera(arvBin* a) {
    /*Se a árvore não for vazia, liberamos a memória, senão a função retorna NULL*/

    if (!arv_vazia(a)) {
        /*Libera a sub - árvore à esquerda*/

        arv_libera(a->esq);

        /*Libera a sub - árvore à direita*/

        arv_libera(a->dir);

        /*Libera o nó raiz*/

        free(a);
    }
    return NULL;
}
```

```c
/*Libera a memória ocupada pela abb*/

Abb* abb_libera(Abb* a) {
    /*Se a árvore não for vazia, liberamos a memória, senão a função retorna NULL*/

    if (!abbVazia(a)) {
        /*Libera a sub - árvore à esquerda*/

        abb_libera(a->esq);

        /*Libera a sub - árvore à direita*/

        abb_libera(a->dir);

        /*Libera o nó raiz*/

        free(a);
    }
    return NULL;
}
```

As funções a seguir são exclusivas de árvores binárias de busca ou árvores binárias, conforme será especificado.

-> Função de inserir um elemento numa árvore binária de busca:

Recebe um ponteiro do tipo Abb e um valor do tipo int.

Nesta função apenas para árvores binárias de busca, inserimos um novo nó na árvore com repetição, ou seja, mesmo que esse nó já esteja na árvore, vamos inseri-lo novamente em outra posição respeitando a estrutura de uma abb.

Se a árvore passada como parâmetro estiver vazia, iremos alocar um espaço na memória para seu nó, guardando seu inteiro e iniciando seus ponteiros para as sub - árvores como NULL.

Caso contrário, verificamos se o inteiro passado como parâmetro é menor que o inteiro no nó raiz, se for, esse nó será alocado à esquerda da raiz, caso contrário será alocado à direita da raiz.

Retorna a árvore atualizada.

```c
/*Inserindo na abb com repetição (Números repetidos serão inseridos na quantidade de vezes que
são gerados)*/

Abb* insereAbb(int i, Abb* a) {
    /*Se o nó passado como parâmetro estiver vazio (NULL), alocamos seu espaço na
    memória para guardar seu inteiro e inicialmente seus ponteiros são NULL*/

    if (abbVazia(a)) {
        a = (Abb*)malloc(sizeof(Abb));
        a->n = i;
        a->esq = a->dir = NULL;
    }

    /*Se o nó do parâmetro não for NULL, comparamos o valor inteiro passado com seu inteiro*/

    /*Se o número passado for menor, inserimos seu nó a esquerda recursivamente*/

    else if (i < a->n) {
        a->esq = insereAbb(i, a->esq);
    }

    /*Se o número for maior, inserimos seu nó a direita recursivamente*/

    else {
        a->dir = insereAbb(i,a->dir);
    }

    /*Retornamos a abb atualizada*/

    return a;
}
```

-> Função de criar um novo nó de árvore binária:

Recebe um inteiro como parâmetro.

Aloca um espaço na memória para um ponteiro para arvBin, alocando o inteiro em seu campo e inicializando suas sub - árvores como NULL e retorna o ponteiro criado.

Caso o ponteiro seja NULL, houve problemas de alocação e a função retornará NULL.

```c
/*arv_criaRaiz cria um novo ná da árvore binária que guardará o inteiro passado
como parâmetro*/

arvBin* arv_criaRaiz(int n) {

    /*Criamos um novo nó usando malloc para alocarmos seu espaço na memória*/

    arvBin* a = (arvBin*)malloc(sizeof(arvBin));

    /*Caso o nó não seja alocado adequadamente,a função retorna NULL */

    if (a == NULL) {
        return NULL;
    }
    /*Guardamos o inteiro no espaço para o inteiro no nó e inicializamos os nós da esquerda
    e direita como NULL*/

    a->num = n;
    a->esq = NULL;
    a->dir = NULL;

    /*Retornamos o novo nó*/

    return a;
}
```

-> Função de inserir um elemento numa árvore binária:

Recebe como parâmetro um ponteiro para o tipo arvBin, um vetor de números inteiros (será gerado no programa principal), um inteiro correspondente ao índice de um elemento no vetor e outro inteiro que indica a quantidade de elementos no vetor.

Se o índice do elemento for menor que a quantidade de elementos, podemos inserir o nó. Para isso, criamos um nó temporário com elemento do índice específico no vetor usando a função arv_criaRaiz descrita acima e igualamos o ponteiro do parâmetro ao nó temporário criado.

Assim fazemos 2 chamadas recursivas da função para cada sub - árvore.Porém nas chamadas recursivas das funções vamos atualizando o índice do elemento do vetor que queremos inserir, sabemos que vamos inserir elementos ímpares nos nós a esquerda e pares nos nós à direita, portanto, na inserção da subárvore à esquerda, o índice sempre será 2*i + 1 (ímpar) e na subárvore à direita, o índice sempre será 2*i + 2 (par).

OBS : adicionei 1 pois a contagem de índices começa em 0, senão, a representação comum de (2*i) e (2*i +1) para par e ímpar, respectivamente, seria usada.

A função retorna a árvore atualizada.

```c
/*Função de inserir os nós na árvore binária por nível*/

/*Passamos um vetor de inteiros como parâmetro para inserir todos os seus elementos na árvore
(Esse vetor guarda os números gerados aleatoriamente no programa principal)*/

/*Passamos tambem o índice de um elemento no vetor (i) a quantidade de elementos no vetor (n)
e o nó da árvore que o elemento será inserido*/

arvBin* insere(arvBin* a, int v[], int i, int n) {

    /*Se o índice for menor que o tamanho do vetor, podemos inserir os elementos*/

    if (i < n) {
        /*Criamos um novo nó da árvore contendo o elemento do índice atual*/

        arvBin* temp = arv_criaRaiz(v[i]);

        /*Passamos esse nó para a árvore passada como parâmetro*/

        a = temp;

        /*Repetimos o processo recursivamente nas sub - árvores à esquerda e à direita*/

        /*Porém nas chamadas recursivas das funções vamos atualizando o índice do elemento
        do vetor que queremos inserir.

        Por exemplo: Sabemos que vamos inserir elementos ímpares nos nós a esquerda e pares nos nós à direita
        portanto, na inserção da sub - árvore à esquerda, o índice sempre será 2*i + 1 (ímpar)
        e na sub - árvore à direita, o índice sempre será 2*i + 2 (par).

        OBS : adicionei 1 pois a contagem de índices começa em 0, senão, a representação comum de
        2*i e 2*i +1 para par e ímpar, respectivamente, seria usada*/

        a->esq = insere(a->esq, v, (2 * i + 1), n);

        a->dir = insere(a->dir, v, (2 * i + 2), n);
    }

    /*Retornamos a árvore atualizada*/

    return a;
}
```

Solução:

Usei as bibliotecas <stdio.h>,para usar nas chamadas de printf e, <stdlib.h>,para fazer alocações dinâmicas de memória e a função free e importei as bibliotecas “abb.h” e “arvBin.h” que contém as funções e structs de árvores binárias de busca e árvores binárias respectivamente.

Usando #define, declarei um número N que tem valor 10 para usá-lo como a quantidade de números gerados que serão usados em todo programa principal, assim fica mais fácil acessar esse valor.

Na função main, declarei 5 variáveis inteiras para guardar o valor do número gerado aleatoriamente, a altura da árvore binária, altura da abb e os retornos das funções de verificação se cada árvore é uma abb, respectivamente. Também declarei um vetor de inteiros de tamanho N para guardar todos os números gerados aleatoriamente, um ponteiro para árvore binária e outro para uma abb.

Usando as funções de criar árvores, inicializei os 2 ponteiros para árvores como NULL com as funções adequadas para cada tipo.

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

/*Incluindo os .h's que contém as funções para arvores binárias e abb's respectivamente*/

#include "arvBin.h"
#include "abb.h"

/*Definindo a quantidade de números aleatórios que serão gerados nesse programa*/

#define N 10

int main(void) {

    /*Declarando variáveis inteiras para guardar o número gerado aleatoriamente,
    a altura da árvore binária, a altura da abb, valores de retorno da função de verificação
    de abb para a árvore binária e a abb respectivamente*/

    int r,alturaBin,alturaBinBusca,t,v;

    /*Declarando um array de tipo inteiro para armazenar os números gerados aleatoriamente*/

    int arr[N];

    /*A variável do tipo arvBin é declarada para criar a árvore binária e a do tipo Abb cria a abb*/

    arvBin* arvore;
    Abb* a;

    /*Inicializando as 2 árvores chamando suas respectivas funçães de inicialização*/

    arvore = criaArv();
    a = criaAbbVazia();
```

Na parte a), começamos a gerar os números aleatoriamente, para isso, criei um loop que funciona N vezes, nelas a variável r guarda o valor de retorno da função rand() , o %20 serve para estabelecer um limite superior ao número gerado e o +1 estabelece um limite inferior, portanto r está no intervalo [1,20] (rand () retorna qualquer número, para que esse número seja no máximo 20, pegamos o resto de sua divisão por 20, mas como ele não pode

ser 0, adicionamos 1 ao resultado caso o número gerado seja 20 e o resto seja 0). Então, para cada volta do loop adicionamos esse número no vetor de acordo com a ordem que foi gerado (0 a 9) e exibimos seu valor.

```c
/*a) Exibir os números gerados aleatoriamente na ordem que foram gerados*/

printf("Ordem gerada:\n");
printf("\n");

/*Loop para gerar a quantidade de números que foi anteriormente estabelecida com define*/

for (int i = 0; i < N; i++) {
    /*A função rand gera um número aleatoriamente, o operador %20 é para que o limite superior desta geração
    seja o número 20 (o número gerado será o resto da sua divisão por 20) e adicionamos mais um para estabelecer
    o limite inferior de 1 (caso o resto da divisão seja igual a 0, adicionamos 1 para que seja um número gerado no
    intervalo de [1,20]*/

    r = rand() % 20 + 1;

    /*Adicionamos os números no array de acordo com a ordem que são gerados*/
    arr[i] = r;

    /*Por fim, imprimimos o número gerado a cada loop*/
    printf("%d\n", r);
}

printf("\n");
```

Na parte b) inserimos os números gerados na parte a) na árvore binária por ordem de nível, para isso chamei a função de inserir na árvore binária. O ponteiro inicializado como NULL anteriormente será passado como parâmetro, assim como o vetor de inteiros, o índice 0 (índice se atualiza nas chamadas recursivas da função, portanto passei o primeiro índice como parâmetro para fazer a inserção do primeiro ao último elemento do vetor) e o N que representa a quantidade de números gerados definido com o #define, e atribuímos o valor retornado pela função ao ponteiro de árvore binária NULL passado como parâmetro.

```c
/*b) Inserir os números gerados numa árvore binária por nível*/

/*Passamos o índice como 0 para começarmos a inserir do começo do array,
o índice se atualiza dentro da função conforme cada chamada recursiva*/

arvore = insere(arvore, arr, 0, N);
```

Já na parte c) inserimos os mesmos números numa abb. Para isso fiz um loop que percorre todos os índices do vetor de inteiros e em cada “volta” a função de inserção em abb era chamada com o elemento e o ponteiro NULL para abb como parâmetros da função e atribuímos o valor de retorno ao ponteiro NULL de abb.

```c
/*c) Inserir os números gerados numa abb com repetição
(todos os valores gerados serão inseridos na abb, incluindo valores repetidos)*/

/*Loop que percorre o array de inteiros para que cada elemento seja inserido na abb*/

for (int j = 0; j < N; j++) {
    a = insereAbb(arr[j], a);
}
```

Na parte d), imprimimos cada nó de ambas as árvores em pré - ordem, para isso chamei suas respectivas funções de imprimir ,exibindo mensagens adequadas para identificar quais árvores estavam sendo exibidas por vez.

```
/*d) Imprimir a árvore binária e a abb em pré - ordem*/
/*Primeiro exibimos a árvore binária em pré - ordem seguida da abb*/

printf("Arvore binaria em pre ordem:\n");
printf("\n");
arvImprime(arvore);
printf("\n");
printf("Abb em pre ordem:\n");
printf("\n");
abb_imprime(a);
printf("\n");
```

Na parte e), verificamos se cada uma das árvores geradas são abb's. Para isso, chamei suas respectivas funções de verificação e guardei seus resultados em 2 variáveis inteiras,t e v, para árvore binária e abb, respectivamente. Caso a variável fosse 1, o programa exibe a mensagem de que a árvore da respectiva variável é uma abb, caso contrário, o programa exibe a mensagem de que a árvore em questão não é uma abb.

```
/*e) Verificar se a abb e a árvore binária são abb's*/
/*Chamada da função de verifição de abb, guardamos seu retorno nas variáveis inteiras t e v
para árvore binária e abb respectivamente*/

t = eh_abb(arvore);
v = is_abb(a);

/*Ambas as funções retornam 1 se a árvore for uma abb, retornam 0 caso contrário*/
/*Informamos se a árvore for ou não uma abb com uma mensagem adequada dependendo
dos valores de t e v*/

if (t == 1) {
    printf("Arvore binaria eh uma abb\n");
}

else {
    printf("Arvore binaria nao eh uma abb\n");
}

printf("\n");

if (v == 1) {
    printf("Abb eh uma abb\n");
}
else {
    printf("Abb nao eh uma abb\n");
}

printf("\n");
```

Na parte f) temos que calcular a altura de cada árvore, para isso chamei suas respectivas funções de calcular altura e guardei seus resultados nas variáveis inteiras definidas anteriormente para cada tipo de árvore e exibi seus resultados com mensagens para identificar a altura de cada tipo de árvore.

```c
/*f) Indicar a altura das 2 árvores*/
/*Usamos as variáveis alturaBin e alturaBinBusca para guardar o valor da
altura de cada árvore e imprimimos os valores com uma mensagem adequada dizendo
a qual árvore eles pertencem*/

alturaBin = arvBin_altura(arvore);

printf("Altura da arvore binaria: %d\n",alturaBin);
printf("\n");

alturaBinBusca = arvbb_altura(a);

printf("Altura da abb: %d\n", alturaBinBusca);
printf("\n");
```

Já para a parte g) temos que trocar todos os nós da direita pelos da esquerda e vice - versa de cada árvore e imprimir em pré - ordem a árvore resultante dessa troca. Para isso, chamei as funções de troca para cada tipo de árvore e atribuí suas árvores retornadas para o ponteiro de árvore adequado para cada função, após isso, chamei a função de imprimir de cada árvore para exibir o resultado da troca mostrando uma mensagem adequada para indicar a árvore que está sendo exibida.

```c
/*g) Trocar todos os nós da direita e da esquerda de cada árvore*/

/*Atualizamos cada árvore com a chamada da sua respectiva função de troca de nós
e exibimos as novas árvores em pré - ordem*/

arvore = trocaDirEsq(arvore);
printf("Arvore binaria apos a troca:\n");
printf("\n");
arvImprime(arvore);
printf("\n");

a = swapLeftRight(a);
printf("Abb apos a troca:\n");
printf("\n");
abb_imprime(a);
printf("\n");
```

Por fim, liberei a memória ocupada pelas árvores chamando as respectivas funções de liberar e como uma forma de demonstrar que ocorreu tudo de forma correta, conferi se os dois ponteiros para árvore estavam nulos, se estivessem isso significa que a memória foi liberada e o programa exibe uma mensagem indicando isso.

```c
/*Liberando a memória usada pelas 2 árvores*/

/*Atualizamos as árvores usando a função de liberar a memória de cada tipo de árvore,
se ambas as árvores forem NULL no final de cada chamada, quer dizer que a memória foi
liberada adequadamente e imprimimos uma mensagem adequada para indicar isso*/

arvore = arv_libera(arvore);
a = abb_libera(a);

if (arv_vazia(arvore) && abbVazia(a)) {
    printf("Memoria liberada!\n");
}

return 0;
```

Conclusões:

Uma parte difícil para mim nesta tarefa foi implementar a função de inserir um nó na árvore binária por nível, acabei utilizando um método mais matemático usando arrays e seus índices ao invés de usar filas como em um dos exemplos em sala de aula, acredito que eu tenha preferido dessa forma pois apesar de já ter aprendido a manipular uma fila, ainda não estou muito familiarizada com suas aplicações em programas.

A função de trocar os nós foi um pouco trabalhosa, mas percebi que usar recursão facilitava meu entendimento,portanto apliquei recursão em outras funções também.

Em relação ao resto do programa, não tive maiores dificuldades.

Observações:

No struct da abb não inclui um ponteiro para o nó pai e também permiti a inserção de elementos duplicados na abb.

Resultados:

```
Ordem gerada:

2
8
15
1
10
5
19
19
3
5

Arvore binaria em pre ordem:

No:00000283A374D7B0 Esquerda:00000283A374DB10 Valor:2 Direita:00000283A3752B80
No:00000283A374DB10 Esquerda:00000283A3753120 Valor:8 Direita:00000283A3752940
No:00000283A3753120 Esquerda:00000283A3752F40 Valor:1 Direita:00000283A37529A0
No:00000283A3752F40 Esquerda:0000000000000000 Valor:19 Direita:0000000000000000
No:00000283A37529A0 Esquerda:0000000000000000 Valor:3 Direita:0000000000000000
No:00000283A3752940 Esquerda:00000283A3753000 Valor:10 Direita:0000000000000000
No:00000283A3753000 Esquerda:0000000000000000 Valor:5 Direita:0000000000000000
No:00000283A3752B80 Esquerda:00000283A3752CA0 Valor:15 Direita:00000283A3753300
No:00000283A3752CA0 Esquerda:0000000000000000 Valor:5 Direita:0000000000000000
No:00000283A3753300 Esquerda:0000000000000000 Valor:19 Direita:0000000000000000

Abb em pre ordem:

No: 00000283A37536C0 Esquerda:00000283A3753780 valor:2 direita:00000283A3753480
No: 00000283A3753780 Esquerda:0000000000000000 valor:1 direita:0000000000000000
No: 00000283A3753480 Esquerda:00000283A3752FA0 valor:8 direita:00000283A3752C40
No: 00000283A3752FA0 Esquerda:00000283A37531E0 valor:5 direita:00000283A3752BE0
No: 00000283A37531E0 Esquerda:0000000000000000 valor:3 direita:0000000000000000
No: 00000283A3752BE0 Esquerda:0000000000000000 valor:5 direita:0000000000000000
No: 00000283A3752C40 Esquerda:00000283A3752A00 valor:15 direita:00000283A3753180
No: 00000283A3752A00 Esquerda:0000000000000000 valor:10 direita:0000000000000000
No: 00000283A3753180 Esquerda:0000000000000000 valor:19 direita:00000283A3752AC0
No: 00000283A3752AC0 Esquerda:0000000000000000 valor:19 direita:0000000000000000
```

```
Arvore binaria nao eh uma abb

Abb eh uma abb

Altura da arvore binaria: 4

Altura da abb: 5

Arvore binaria apos a troca:

No:00000283A374D7B0 Esquerda:00000283A3752B80 Valor:2 Direita:00000283A374DB10
No:00000283A3752B80 Esquerda:00000283A3753300 Valor:15 Direita:00000283A3752CA0
No:00000283A3753300 Esquerda:0000000000000000 Valor:19 Direita:0000000000000000
No:00000283A3752CA0 Esquerda:0000000000000000 Valor:5 Direita:0000000000000000
No:00000283A374DB10 Esquerda:00000283A3752940 Valor:8 Direita:00000283A3753120
No:00000283A3752940 Esquerda:0000000000000000 Valor:10 Direita:00000283A3753000
No:00000283A3753000 Esquerda:0000000000000000 Valor:5 Direita:0000000000000000
No:00000283A3753120 Esquerda:00000283A37529A0 Valor:1 Direita:00000283A3752F40
No:00000283A37529A0 Esquerda:0000000000000000 Valor:3 Direita:0000000000000000
No:00000283A3752F40 Esquerda:0000000000000000 Valor:19 Direita:0000000000000000

Abb apos a troca:

No: 00000283A37536C0 Esquerda:00000283A3753480 valor:2 direita:00000283A3753780
No: 00000283A3753480 Esquerda:00000283A3752C40 valor:8 direita:00000283A3752FA0
No: 00000283A3752C40 Esquerda:00000283A3753180 valor:15 direita:00000283A3752A00
No: 00000283A3753180 Esquerda:00000283A3752AC0 valor:19 direita:0000000000000000
No: 00000283A3752AC0 Esquerda:0000000000000000 valor:19 direita:0000000000000000
No: 00000283A3752A00 Esquerda:0000000000000000 valor:10 direita:0000000000000000
No: 00000283A3752FA0 Esquerda:00000283A3752BE0 valor:5 direita:00000283A37531E0
No: 00000283A3752BE0 Esquerda:0000000000000000 valor:5 direita:0000000000000000
No: 00000283A37531E0 Esquerda:0000000000000000 valor:3 direita:0000000000000000
No: 00000283A3753780 Esquerda:0000000000000000 valor:1 direita:0000000000000000

Memoria liberada!
```