**ÁRVORE BINÁRIA**

- Árvore binária é um conjunto finito de elementos que ou é vazio ou é dividido em três subconjuntos disjuntos:

* raiz;
* sub-árvore da direita (binária);
* sub-árvore da esquerda (binária).

- A árvore binária segue os seguintes princípios:

* O número máximo de nós no nível *i* é 2i;
* O número máximo de nós um uma árvore binária de altura k é 2k+1-1.

**OPERAÇÕES**

IMPLEMENTAÇÃO

- Definimos uma *struct* para nó, onde cada nó guarda seu dado e dois ponteiros, um para o nó da esquerda e outro, para o da direita.

- A árvore binária é indicada por sua raiz, sendo um ponteiro de ponteiro para o nó raiz.

struct No{

int dado;

struct No \*esq;

struct No \*dir;

};

typedef struct No\* Arvore;

- No programa principal:

Arvore\* raiz = cria\_Arvore();

CRIAÇÃO

- A criação da árvore binária consiste em definir um ponteiro de ponteiro para a raiz, através de *malloc*, e que aponta para nulo (árvore vazia).

Arvore\* cria\_arvore(){

Arvore\* raiz = (Arvore\*) malloc(sizeof(Arvore));

if(raiz != NULL)

\*raiz = NULL;

return raiz;

}

- No programa principal:

Arvore\* raiz = cria\_Arvore();

LIBERAÇÃO

- Para destruir a árvore, criamos duas funções: a função *libera\_Arvore* chama recursivamente a função *libera\_No*, responsável por percorrer a árvore e liberar a memória associada a cada nó da árvore.

void libera\_No(struct No\* no){

if(no == NULL)

return;

libera\_No(no->esq);

libera\_No(no->dir);

free(no);

no = NULL;

}

void libera\_Arvore(Arvore\* raiz){

if(raiz == NULL)

return;

libera\_No(\*raiz); //libera cada nó

free(raiz); //libera a raiz

}

- A função para liberação dos nós percorre toda a esquerda, libera o nó folha, passa para a direita, se houver, e faz o mesmo, e vai liberando assim recursivamente os nós ascendentes até a raiz. A função para liberação da árvore termina por eliminar o ponteiro para a árvore.

ALTURA

- A função para calcular a altura da árvore percorre toda a esquerda até o nó folha, compara com a direita, se houver, e a maior altura entre os dois é retornada. A altura naquele ponto é dada por aquela altura do maior filho + 1. Depois, ascende recursivamente ao longo da árvore, esquerda e direita, calculando as alturas até a raiz.

int altura(Arvore \*raiz){

if (raiz == NULL)

return 0;

if (\*raiz == NULL)

return 0;

int alt\_esq = altura(&((\*raiz)->esq));

int alt\_dir = altura(&((\*raiz)->dir));

if (alt\_esq > alt\_dir)

return(alt\_esq + 1);

else

return(alt\_dir + 1);

}

PERCURSOS

- Há três formas de percurso na árvore binária: pré-ordem ou profundidade, em-ordem ou ordem simétrica, e pós-ordem.

Pré-ordem

- Percurso pré-ordem consiste em:

* Mostra o valor do nó;
* Visita o nó esquerdo;
* Visita o nó direito.

- Na função a seguir para impressão, imprime-se primeiro o nó pai (o qual será, primeiramente, o raiz), depois os nós à esquerda. Depois, passa à impressão dos da direita recursivamente. Assim que o nó é visitado, o valor é mostrado.

void preOrdem(Arvore \*raiz){

if(raiz == NULL)

return;

if(\*raiz != NULL){

printf("%d\n",(\*raiz)->dado); // começa pela raiz

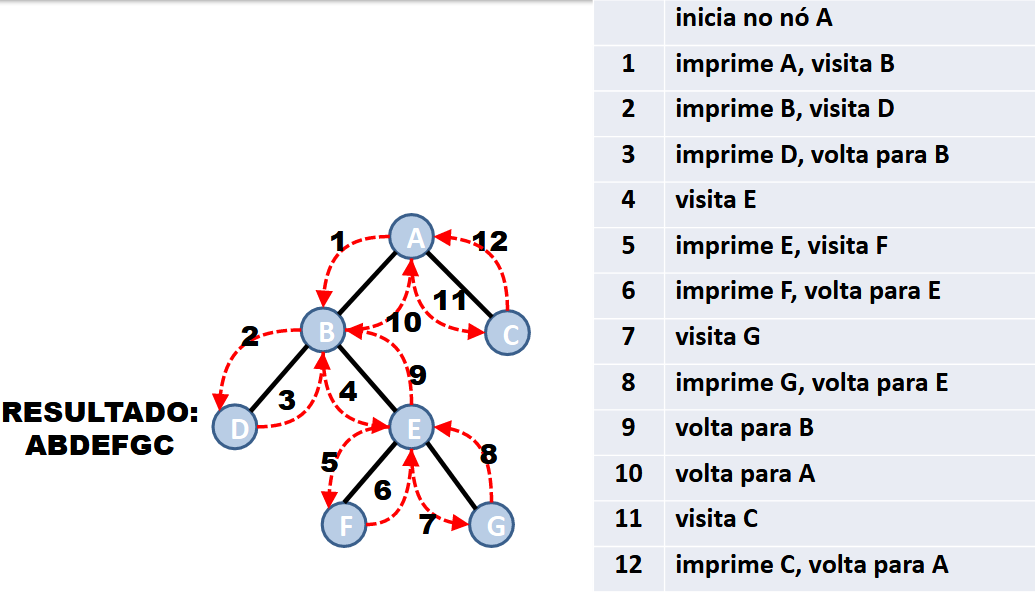
preOrdem(&((\*raiz)->esq));

preOrdem(&((\*raiz)->dir));

}

}

- Exemplo:



Em-ordem

- Percurso em-ordem consiste em:

* Visita o nó esquerdo;
* Mostra o valor do nó;
* Visita o nó direito.

- Inicia o percurso pela raiz da árvore, visitando primeiramente todos os nós à esquerda. Imprime os valores quando todos à esquerda já tiverem sido visitados. Depois, imprime os da direita recursivamente.

void emOrdem(Arvore \*raiz){

if(raiz == NULL)

return;

if(\*raiz != NULL){

emOrdem(&((\*raiz)->esq));

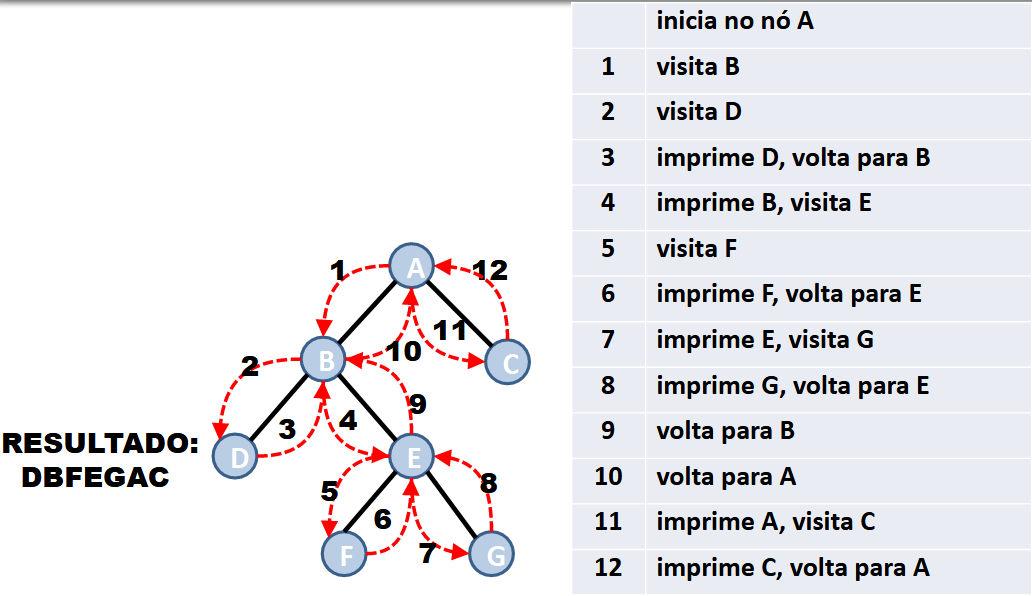
printf("%d\n",(\*raiz)->dado);

emOrdem(&((\*raiz)->dir));

}

}

- Exemplo:



Pós-ordem

- Percurso pós-ordem consiste em:

* Visita o nó esquerdo;
* Visita o nó direito;
* Mostra o valor do nó.

- Inicia o percurso pela raiz e visita primeiramente os nós à esquerda, depois à direita, e imprime os descendentes depois que eles foram visitados. Posteriormente, imprime o nó pai.

void posOrdem(Arvore \*raiz){

if(raiz == NULL)

return;

if(\*raiz != NULL){

posOrdem(&((\*raiz)->esq));

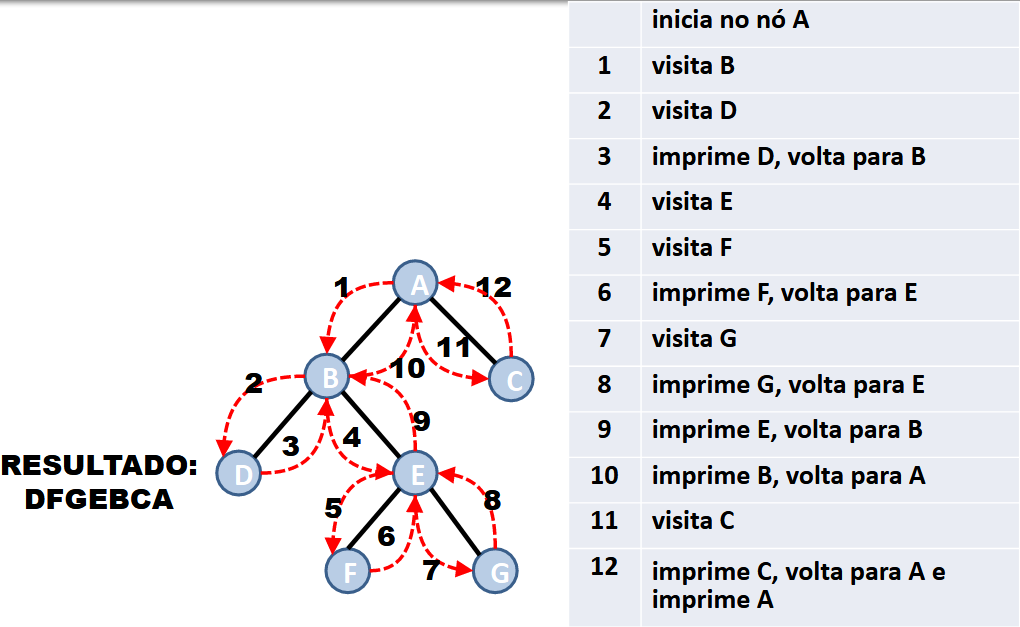
posOrdem(&((\*raiz)->dir));

printf("%d\n",(\*raiz)->dado);

}

}

- Exemplo:



**ÁRVORE BINÁRIA DE BUSCA**

- A árvore binária de busca é caracterizada por cada nó conter um valor (chave) associado a ele, e esse valor determina a posição do nó na árvore.

- Para cada nó pai, todos os valores da sub-árvore da esquerda são menores do que o do nó pai, e todos os valores da sub-árvore da direita são maiores do que o do nó pai.

- Uma árvore criada a partir de um conjunto de valores não é única; o resultado depende da sequência de inserção de dados.

- Um percurso em-ordem resulta na sequência de valores em ordem crescente.

- Grande utilidade da árvore de busca é armazenar dados contra os quais outros dados são verificados (busca).

- Na busca binária em um vetor, a cada divisão, desprezamos toda a porção de valores maior ou menor que a chave (posição do meio a cada iteração). A árvore de busca binária pode apresentar o mesmo desempenho desde que sua altura seja mínima, isto é, se ela for perfeitamente balanceada. O número de comparações então será igual a: hmin= menor inteiro maior ou igual a log2(n+1).

- As operações de inserção e remoção devem ser realizadas respeitando a lógica de maiores e menores valores.

**OPERAÇÕES**

INSERÇÃO

- Passo-a-passo da inserção:

* Procura a partir do nó raiz;
* Para cada nó v, compara: se o novo nó possui valor menor que v, o que nos leva para a sub-árvore da esquerda, ou se possui valor maior que v, o que nos leva para a sub-árvore da direita.
* Se uma referência nula (em uma folha) é atingida, coloca o novo nó como sendo filho do nó em questão.

int insere(Arvore\* raiz, int valor){

if(raiz == NULL)

return 0;

struct No\* novo;

novo = (struct No\*) malloc(sizeof(struct No));

if(novo == NULL)

return 0;

novo->dado = valor;

novo->dir = NULL;

novo->esq = NULL;

// procura o local para o novo nó

if(\*raiz == NULL)

\*raiz = novo;

else{

struct No\* atual = \*raiz;

struct No\* ant = NULL;

// navega nos nós até chegar em um nó folha

while(atual != NULL){

ant = atual;

if(valor == atual->dado){

free(novo);

return 0; // elemento já existe

}

if(valor > atual->dado)

atual = atual->dir;

else

atual = atual->esq;

}

// insere como filho do nó folha

if(valor > ant->dado)

ant->dir = novo;

else

ant->esq = novo;

}

return 1;

}

REMOÇÃO

- Utilizamos duas funções: uma busca pelo nó a ser removido, enquanto a outra resolve as dependências em relação à remoção.

- *remove\_Arvore* percorre a árvore procurando o nó desejado. Utilizamos duas variáveis para percorrer: *ant*, para guardar o nó pai do nó visitado, e *atual*, para o nó visitado.

*ant* recebe o *atual*, e o percurso segue à direita caso o valor desejado seja maior que o valor do atual, ou à esquerda, caso seja menor.

- Quando o valor é encontrado na variável *atual*, dois casos são tratados: um em que o nó em questão é raiz, e outro em que não é. No primeiro caso, a raiz recebe a próxima função, *remove\_atual*. No segundo caso, testamos se o nó em questão é direita ou esquerda a partir de seu pai, *ant*, comparando se *ant->dir* é igual a *atual*. É necessário atualizar essa direita ou esquerda do pai após a remoção, fazendo com que passe a apontar para o nó retornado pela função *remove\_atual*.

int remove\_Arvore(Arvore \*raiz, int valor){

if(raiz == NULL)

return 0;

struct No\* ant = NULL;

struct No\* atual = \*raiz;

while(atual != NULL){

// se achou o nó a ser removido, invoca remove\_atual

if(valor == atual->dado){

if(atual == \*raiz)

\*raiz = remove\_atual(atual);

else{

if(ant->dir == atual)

ant->dir = remove\_atual(atual);

else

ant->esq = remove\_atual(atual);

}

return 1;

}

// se não achou, continua percorrendo a árvore

ant = atual;

if(valor > atual->dado)

atual = atual->dir;

else

atual = atual->esq;

}

return 0;

}

- A segunda função, então, trata do caso em que o nó é folha ou tem um filho apenas. Se não houver nó esquerdo, uma variável recebe o nó da direita e a função retorna ela. No caso de nó folha, o nó da direita é null, será retornado null.

- Em seguida, trata do caso em que o nó possui dois filhos, em que é necessário buscar o nó mais à direita na sub-árvore da esquerda. Depois copia o nó achado no lugar do nó removido.

struct No\* remove\_atual(struct No\* atual) {

struct No \*no1, \*no2;

// primeiro caso: se o nó é folha ou tem um filho apenas

if(atual->esq == NULL){

no2 = atual->dir;

free(atual);

return no2;

}

// segundo caso: se o nó possui dois filhos, vai buscar o nó mais à direita na sub-árvore da esquerda

no1 = atual;

no2 = atual->esq;

while(no2->dir != NULL){

no1 = no2;

no2 = no2->dir;

}

// copia o nó achado em questão para o lugar do nó removido

if(no1 != atual){

no1->dir = no2->esq;

no2->esq = atual->esq;

}

no2->dir = atual->dir;

free(atual);

return no2;

}