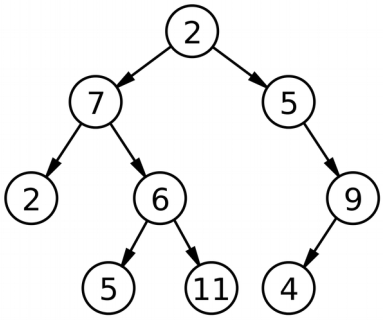
Universidade Federal de Goiás
Engenharia da Computação
Estruturas de Dados 1

2ª Avaliação – 2019.1

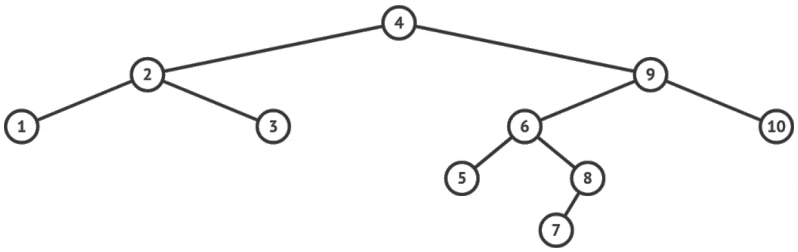
1. **(2.0)** Analise a árvore binária abaixo e escreva a sequência em que os nós são visitados em cada tipo de percurso: pré-ordem, em-ordem, pós-ordem e por nível.



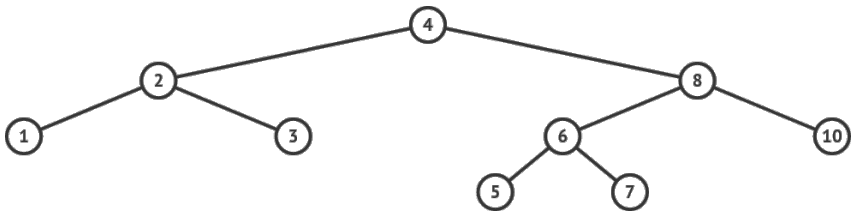
Pré-ordem: 2 7 2 6 5 11 5 9 4
Em-ordem: 2 7 5 6 11 2 5 4 9
Pós-ordem: 2 5 11 6 7 4 9 5 2
Por nível: 2 7 5 2 6 9 5 11 4

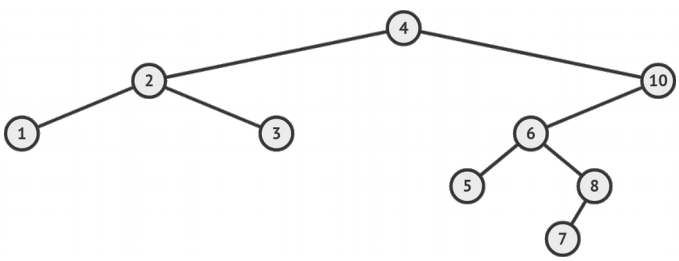
2. **(1.5)** Desenhe a árvore binária de busca obtida a partir da inserção dos itens [4, 9, 2, 6, 3, 10, 8, 7, 5, 1] exatamente nesta ordem. Em seguida, desenhe a árvore obtida a partir da exclusão do item 9 da árvore obtida anteriormente.

Árvore binária de busca gerada



Soluções possíveis removendo o 9. Substituindo pelo predecessor (8) ou sucessor (10).





3. **(1.0)** Analise a função abaixo. Qual o tipo de percurso em árvore implementado pela função?

```
 1. void percorre(NoArvore *raiz) {
 2.     Pilha *p = pilha_nova();
 3.     NoArvore *no = raiz;
 4.     while (no != NULL || !pilha_esta_vazia(p)) {
 5.         while (no != NULL) {
 6.             pilha_empilha(p, no);
 7.             no = no->esq;
 8.         }
 9.         no = pilha_desempilha(p);
10.         printf("%d ", no->chave);
11.         no = no->dir;
12.     }
13.     pilha_libera(p);
14. }
```

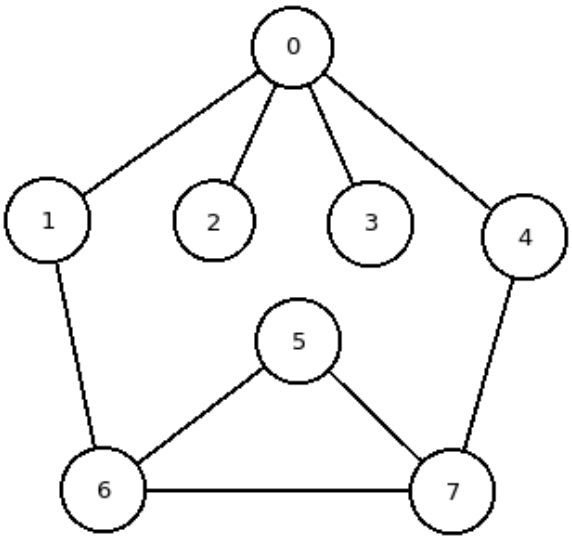
Em-ordem

4. **(2.5)** Escreva o código ou algoritmo de uma função que receba o nó raiz de uma árvore binária e retorne a altura dessa árvore

```c
typedef struct no_arvore NoArvore;
struct no_arvore {
    int chave;
    struct no_arvore *esq, *dir;
};

int altura(NoArvore *raiz) {
    if (raiz == NULL)
        return -1;
    int altura_esq = altura(raiz->esq);
    int altura_dir = altura(raiz->dir);
    if (altura_esq > altura_dir)
        return altura_esq + 1;
    else
        return altura_dir + 1;
}
```

5. **(2.0)** Analise o grafo abaixo.



a. Represente o grafo como matriz de adjacência e lista de adjacência

	0	1	2	3	4	5	6	7
0	0	1	1	1	1	0	0	0
1	1	0	0	0	0	0	1	0
2	1	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1	1
6	0	1	0	0	0	1	0	1
7	0	0	0	0	1	1	1	0

0 → 1 → 2 → 3 → 4
1 → 0 → 6
2 → 0
3 → 0
4 → 0 → 7
5 → 6 → 7
6 → 1 → 5 → 7
7 → 4 → 5 → 6

b. Escreva uma sequencia em que os vértices podem ser visitados ao percorrer o grafo utilizando um algoritmo de busca em profundidade a partir do vértice 0.
0 1 6 5 7 4 2 3

c. Escreva uma sequencia em que os vértices podem ser visitados ao percorrer o grafo utilizando um algoritmo de busca em largura a partir do vértice 0.
0 1 2 3 4 6 7 5

6. **(1.0)** O código abaixo deveria implementar o algoritmo de busca em profundidade, porém possui um erro de lógica, fazendo que a função não se comporte corretamente. Descreva o problema existente no algoritmo e como poderia ser solucionado.

```
 1. void dfs(Grafo *grafo, int origem) {
 2.     Pilha *pilha = pilha_nova();
 3.     pilha_empilha(pilha, origem);
 4.     while (!pilha_vazia(pilha)) {
 5.         int v = pilha_desempilha(pilha);
 6.         printf("%d ", v);
 7.         NoAresta *vizinho = grafo->adjacencia[v];
 8.         while (vizinho != NULL) {
 9.             pilha_empilha(pilha, vizinho->vertice);
10.             vizinho = vizinho->prox;
11.         }
12.     }
13.     while (!pilha_vazia(pilha))
```

```
14.         pilha_desempilha(pilha);
15.     free(pilha);
16. }
```

Antes de visitar um vértice o algoritmo não verifica se esse vértice já foi visitado anteriormente. Caso o grafo possua algum ciclo o algoritmo entrará em repetição infinita. Para solucionar o problema seria necessário marcar os vértices como visitados assim que são processados pela primeira vez. Caso o vértice seja encontrado novamente por um outro caminho e já tenha sido marcado como visitado, deve ser ignorado.