UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

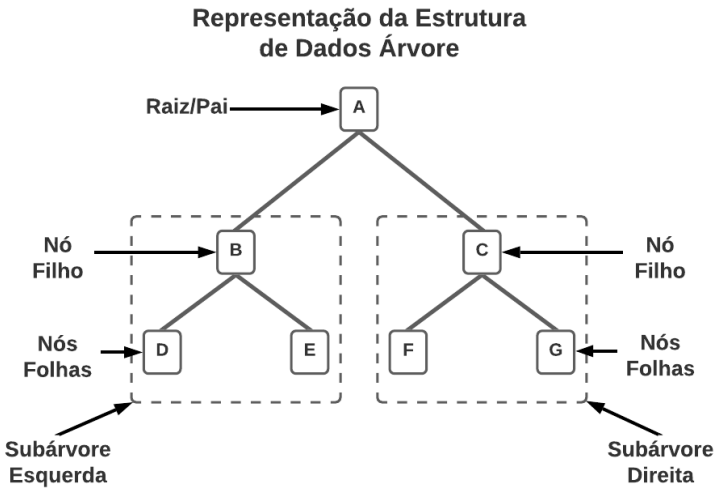
Aluno: **Rosivaldo Lucas da Silva**
Matrícula: **20190028170**
Disciplina: **Estrutura de Dados**
Semestre: **2021.1**

Exercício de Fixação e Aprendizagem III

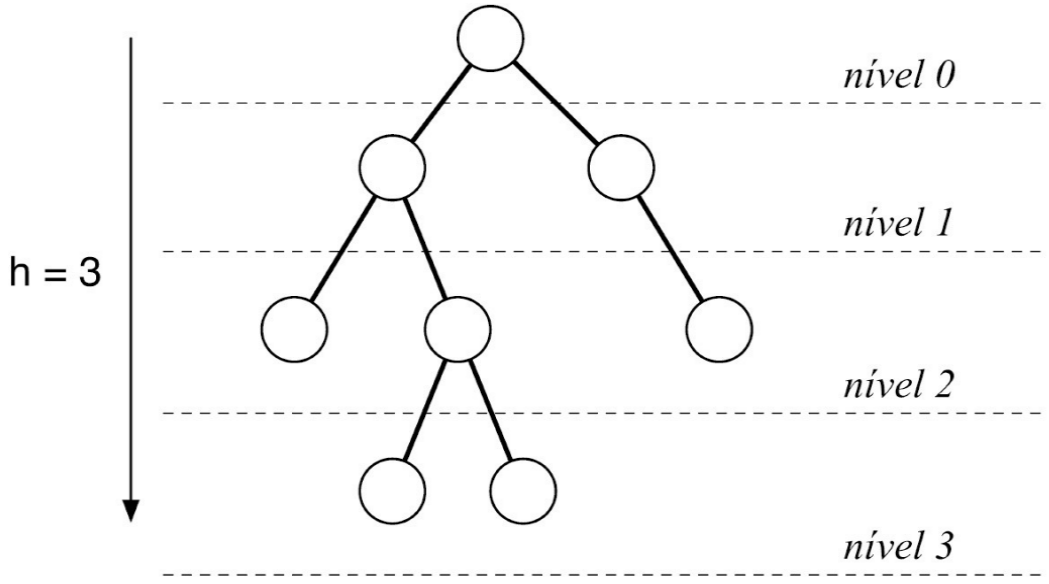
QUESTÃO 1

A.

- **Árvores:**
  Uma árvore é uma estrutura de dados não linear que representa e organiza elementos de forma hierárquica. A estrutura é composta por um conjunto de nós, consistindo de um nó chamado raiz que se encontra no topo da estrutura e abaixo se encontram os demais nós, chamados de nós filhos ou nós internos (descendentes do nó raiz), os nós filhos podem conter zero, um ou mais de um nós filhos. Os nós filhos que não contém outros nós são chamados de nós folhas ou nós externos. Abaixo do nó raiz a árvore forma subárvores que são constituídos pelos seus nós filhos.



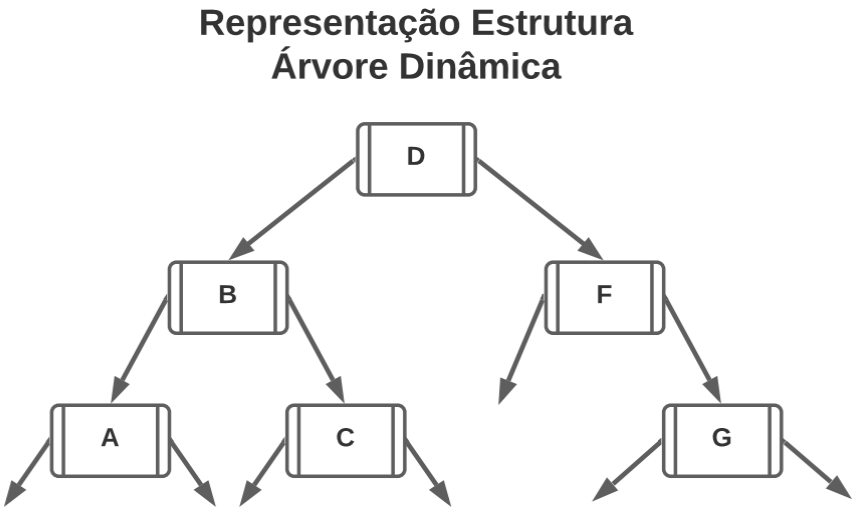
- **Altura de uma Árvore:**
  A altura de uma árvore é um dado importante para realizar a implementação da estrutura, em árvores binárias a altura é utilizada para definir a sua complexidade de busca. Pode-se definir a altura como sendo o comprimento do caminho da raiz até a folha mais distante. Assim dizemos que a altura de uma árvore que possui apenas o nó raiz é zero, e para uma árvore vazia dizemos que sua altura é -1.



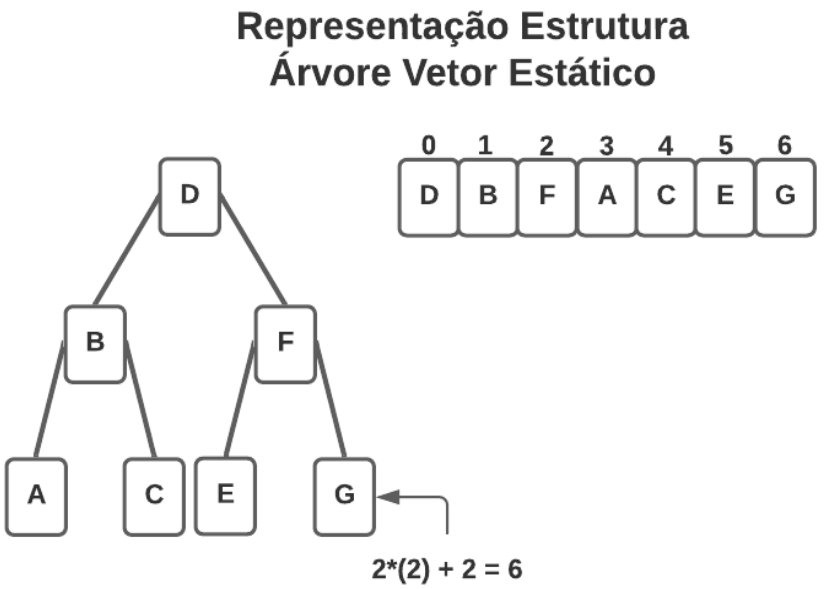
- **Aplicações:**
  - Sistemas de arquivos.
  - Organogramas.
  - Ambientes de Programação.

- **Características:**
  - **Raiz:** Toda árvore possui o nó raiz que é o nó inicial.
  - **Grau:** É o número de nós filhos que um nó possui.
  - **Nível:** É a distância de um nó filho até o nó raiz.
  - **Altura:** O maior nível encontrado na árvore.
  - **Folha:** O nó que não possui filhos.

- **Representação Dinâmica:**
  Na representação dinâmica da estrutura da árvore são criados dois ponteiros onde irão armazenar as referências para as subárvores da esquerda e direita.



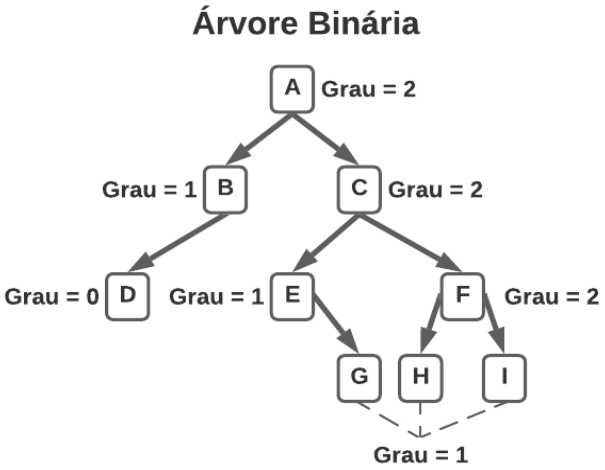
- **Representação Estática com Vetor:**
  Nessa representação é usado um vetor estático onde o seu primeiro elemento (índice 0) será ocupado pelo nó raiz, o segundo e terceiro elemento pelos nós filhos a esquerda e a direita respectivamente. Para realizar a adição de mais nós na estrutura é necessário seguir as fórmulas de que os filhos do índice i serão **2i+1** para o filho esquerdo e **2i+2** para o filho esquerdo.



- **Árvores Binárias**
  Uma árvore binária é uma especificação do conceito geral de árvores, onde abaixo de cada nó é permitido existir no máximo duas subárvores. Logo o grau de cada nó da árvore pode assumir apenas

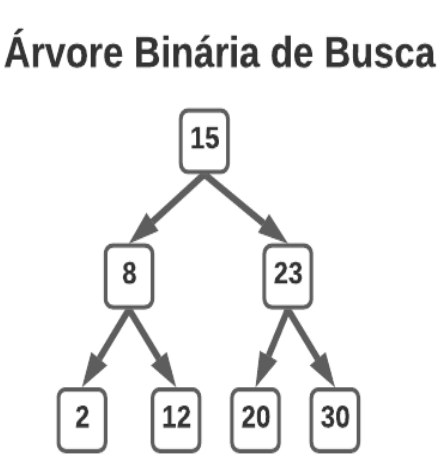
os valores 0, 1 ou 2. A representação de árvores binárias é feita através de estruturas dinâmicas.

Uma aplicação desse tipo de estrutura é na utilização para avaliar expressões.

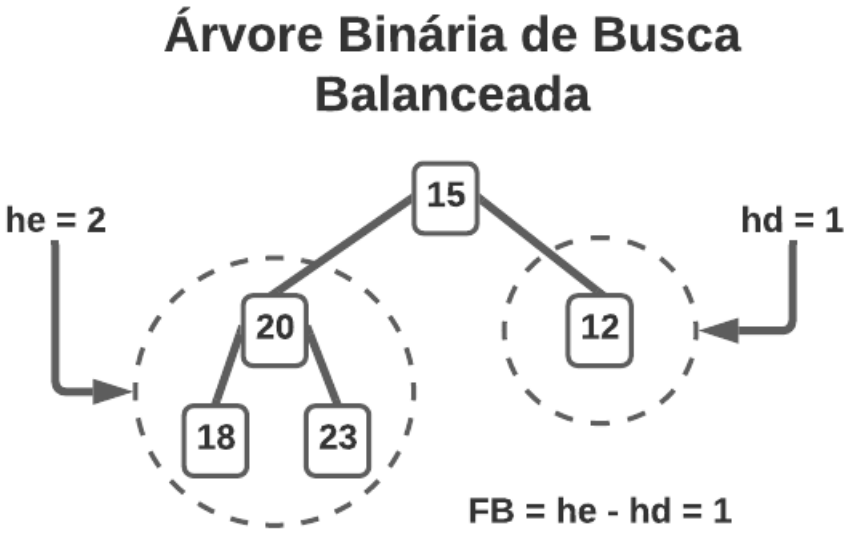


- **Árvores Binárias de Busca**
  Árvore binária de busca é uma árvore binária em que, a cada nó, todos os registros com chaves menores que a do nó em questão estão na subárvore da esquerda, e os nós que contenham chaves maior do que o nó em questão estão na subárvore da direita.

  Esse tipo de estrutura torna possível a realização de busca binária entre as chaves da árvore. A altura da árvore determina a sua complexidade de buscar o elemento desejado, podendo ter uma complexidade de $O(log2(n))$ou $O(n)$ quando é necessário descer até os nós folhas. Nessa estrutura não é garantido o balanceamento da árvore, então dependendo da ordem de inserção dos elementos essa árvore pode se tornar uma lista ligada, assim causando lentidão na busca.

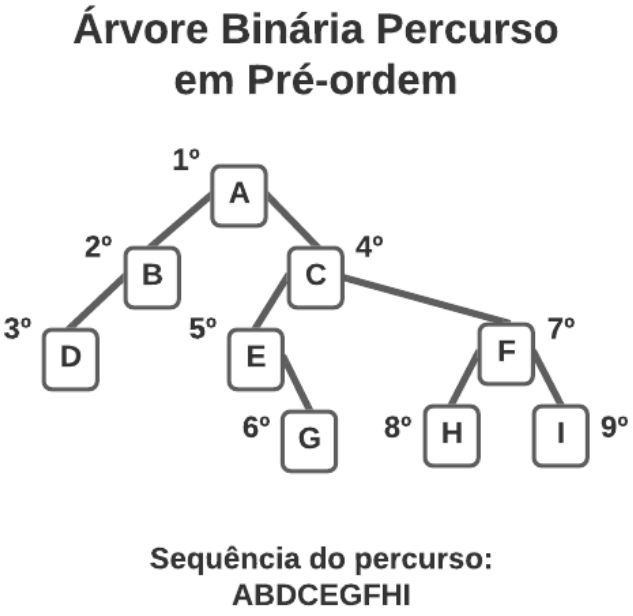


- **Árvores de Busca Balanceadas**
  É uma árvore binária de busca, mas balanceada em relação à altura de suas subárvoras. Esse balanceamento é fundamental para garantir que a busca na árvore continue tendo complexidade de $O(log2(n))$.
  A árvore binária é dita balanceada, se para cada nó a altura das suas subárvores diferem de no máximo 1. Caso contrário é necessário realizar o balanceamento através de operações de rotação que podem ser rotações simples e rotações duplas à direita e à esquerda.

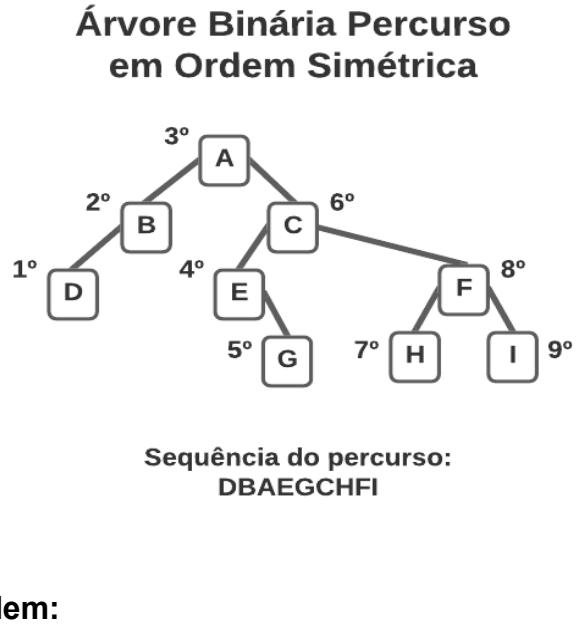


B.

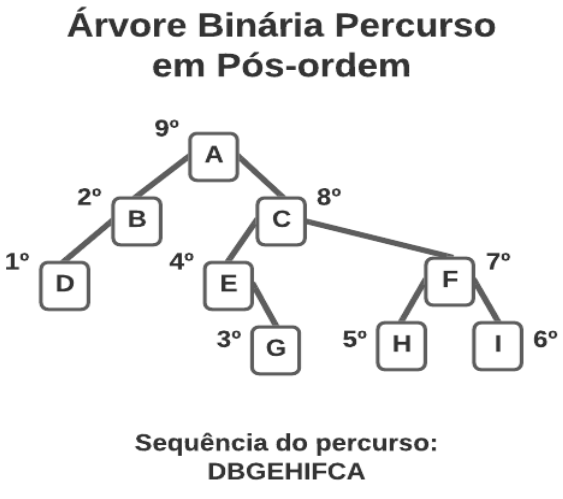
- **Pré-ordem:**
  Nessa abordagem é feito primeiro a visita ao nó raiz e depois são visitadas as subárvores esquerda e direita.



- **Ordem Simétrica:**
  Primeiro é feita a visita a subárvore da esquerda, depois é visitado o nó raiz e por fim é visitada a subárvore da direita.



- **Pós-ordem:**
  A subárvore da esquerda é a primeira a ser visitada, depois a subárvore da direita e por fim é visitado o nó raiz.



C.

Implementação de uma árvore binária de busca.

- Estruturas e funções implementadas

```
typedef int chave;
typedef struct no no_t;
typedef struct abb abb_t;

abb_t *abb_cria();
void abb_libera(abb_t *abb);
int abb_insere(abb_t *abb, chave c);
int abb_remove(abb_t *abb, chave c);
no_t *abb_busca(abb_t *abb, chave c);
int abb_qtd_folha(abb_t *abb);
void abb_mostra(abb_t *abb);
```

- Bibliotecas e estruturas utilizadas

```
#include <stdio.h>
#include <stdlib.h>
#include "abb.h"

struct no {
   chave c;
   no_t *esq, *dir;
};

struct abb {
   no_t *raiz;
};
```

- Função que cria uma árvore binária busca vazia

```
abb_t *abb_cria() {
    abb_t *abb = (abb_t *) malloc(sizeof(abb_t));

    if (abb == NULL) return NULL;

    abb->raiz = NULL;

    return abb;
}
```

- Função que libera os nós da árvore binária de busca

```
void libera(no_t *raiz) {
    if (raiz != NULL) {
        libera(raiz->esq);
        libera(raiz->dir);
        free(raiz);
    }
}
void abb_libera(abb_t *abb) {
    if (abb != NULL) libera(abb->raiz);
}
```

- Função que realiza a inserção de um novo nó na árvore binária de busca

  Casos considerados na inserção de um nó na árvore binária de busca:

  1. **raiz == NULL:** Insere o elemento que passará a ser a raiz, fim do algoritmo.
  2. **raiz->c == c:** Elemento já inserido na árvore, fim do algoritmo.
  3. **raiz->c > c:** Realiza uma chamada recursiva para a subárvore da esquerda até inserir o novo nó como folha.
  4. **raiz->c < c:** Realiza uma chamada recursiva para a subárvore da direita até inserir o novo nó como folha.

```
no_t *insere(no_t *raiz, chave c) {
    if (raiz == NULL) {
        raiz = (no_t *) malloc(sizeof(no_t));
        raiz->c = c;
        raiz->esq = NULL;
        raiz->dir = NULL;
    } else if (raiz->c == c) {
        return raiz;
    } else if (raiz->c > c) {
        raiz->esq = insere(raiz->esq, c);
```

```
    } else {
        raiz->dir = insere(raiz->dir, c);
    }
    return raiz;
}
int abb_insere(abb_t *abb, chave c) {
    if (abb == NULL) return 0;
    abb->raiz = insere(abb->raiz, c);
    return 1;
}
```

- Função que remove um nó da árvore binária de busca

  Casos considerados na remoção de um nó na árvore binária de busca:
  1. Nó a ser removido é folha:
     a. Desaloca o nó da memória
     b. E faz ele apontar para NULL.
  2. Remover um nó que tem apenas 1 filho:
     a. Desaloca o nó da memória
     b. Sobe o nó filho para o nó pai
  3. Remover um nó que tem dois nós filhos:
     a. Procura o maior valor da subárvore esquerda
     b. Faz o nó receber o valor da chave encontrado
     c. Desaloca a maior chave da subárvore esquerda

```
no_t *remover(no_t *raiz, chave c) {
    if (raiz == NULL) {
        return NULL;
    } else if (raiz->c > c) {
        raiz->esq = remover(raiz->esq, c);
    } else if (raiz->c < c) {
        raiz->dir = remover(raiz->dir, c);
    } else {
        if (raiz->esq == NULL && raiz->dir == NULL) {
            free(raiz);
            raiz = NULL;
        } else if (raiz->esq == NULL) {
            no_t *aux = raiz;
            raiz = raiz->dir;
            free(aux);
        } else if (raiz->dir == NULL) {
            no_t *aux = raiz;
            raiz = raiz->esq;
            free(aux);
```

```
        } else {
            no_t *aux = raiz->esq;

            while (aux->dir != NULL) {
                aux = aux->dir;
            }

            raiz->c = aux->c;

            raiz->esq = remover(raiz->esq, aux->c);
        }
    }

    return raiz;
}
int abb_remove(abb_t *abb, chave c) {
    if (abb == NULL) return 0;

    abb->raiz = remover(abb->raiz, c);
}
```

- Função que realiza a busca binária de busca na árvore

  Casos considerados na busca binária em uma árvore:
  1. **raiz == NULL:** Chave não se encontra na árvore, fim do algoritmo.
  2. **raiz->c == c:** Achou o elemento buscado no nó raiz, retorna o nó e finaliza o algoritmo.
  3. **raiz->c > c:** O elemento pode estar na subárvore esquerda, realiza uma chamada recursiva para a função busca.
  4. **raiz->c < c:** O elemento pode estar na subárvore direita, realiza uma chamada recursiva para a função busca.

```
no_t *busca(no_t *raiz, chave c) {
    if (raiz == NULL) {
        return NULL;
    } else if (raiz->c == c) {
        return raiz;
    } else if (raiz->c > c) {
        return busca(raiz->esq, c);
    } else {
        return busca(raiz->dir, c);
    }
}
no_t *abb_busca(abb_t *abb, chave c) {
```

```
    if (abb == NULL) return NULL;

    return busca(abb->raiz, c);
}
```

- Função que retorna a quantidade de folhas na árvore binária de busca

  Casos considerados na busca binária em uma árvore:
  1. **raiz == NULL:** Árvore vazia, retorna zero e finaliza o algoritmo.
  2. **Nó esq e nó dir == NULL:** Chegou em uma folha, retorna 1.
  3. Realiza uma chamada recursiva para a função passando a subárvore da esquerda e da direita e armazena o valor retornado em uma variável.
  4. Soma as variáveis da esq e dir e retorna a quantidade de folhas na árvore.

```
int qtd_folha(no_t *raiz) {
    int qtd_e = 0;
    int qtd_d = 0;

    if (raiz == NULL) {
        return 0;
    } else if (raiz->esq == NULL || raiz->dir == NULL) {
        return 1;
    }

    qtd_e = qtd_folha(raiz->esq);
    qtd_d = qtd_folha(raiz->dir);

    return (qtd_e + qtd_d);
}
int abb_qtd_folha(abb_t *abb) {
    if (abb == NULL) return -1;

    return qtd_folha(abb->raiz);
}
```

- Função que imprime a árvore binária de busca

```
void mostra(no_t *raiz) {
    if (raiz != NULL) {
        printf("<%d", raiz->c);
        mostra(raiz->esq);
        mostra(raiz->dir);
        printf(">");
```

```
    } else {
        printf("<>");
    }
}
void abb_mostra(abb_t *abb) {
    if (abb != NULL) mostra(abb->raiz);
}
```

D.

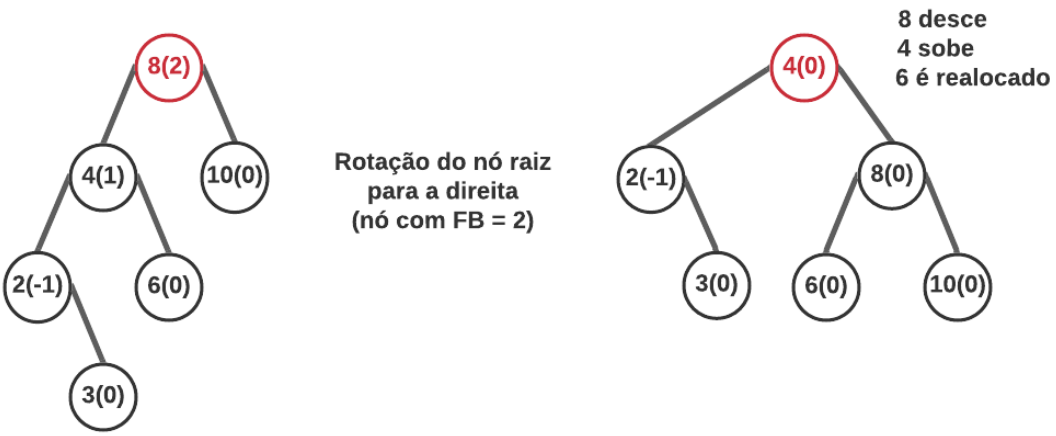
O balanceamento de árvores binárias de busca é realizado para garantir eficiência de $O(log2(n))$ na busca, caso a árvore não esteja balanceada o pior caso de ineficiência pode ser de $O(n)$, tornando ineficiente realizar uma busca na estrutura. Abaixo é mostrado as estratégias para se balancear uma árvore binária de busca.

O balanceamento de árvores binárias é feito realizando uma comparação entre as alturas das subárvores. Caso a árvore tenha uma diferença de no máximo 1, essa árvore é dita balanceada.

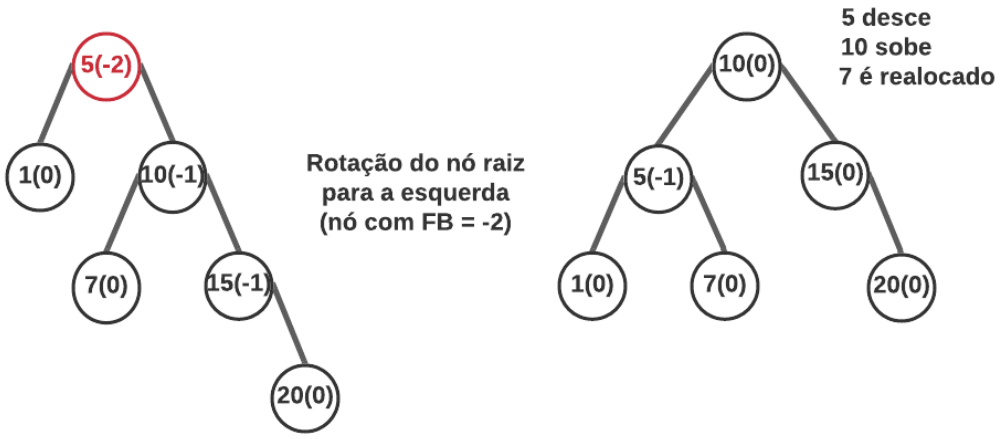
- **Fator de balanceamento (FB):**
  - Altura da subárvore esquerda menos altura da subárvore direita (FB = he - hd).
  - Atualizar o FB sempre que é realizada uma inserção ou uma remoção.
  - Árvore balanceada quando os FB forem 0, 1 ou -1.
  - Árvore desbalanceada quando os FB forem 2 ou -2.
  - Se a árvore pende para a esquerda (FB positivo), rotaciona-se para a direita.
  - Se a árvore pende para a direita (FB negativo), rotaciona-se para a esquerda.

- **1 Caso:**
  Acontece quando os fatores de balanceamento da subárvore e filho têm sinais iguais. Subárvore do nó raiz e do filho pendem para o mesmo lado.

  - **1.1 Rotação simples à direita:**
    É realizado quando os fatores de balanceamento do nó raiz e do nó filho são positivos, então a subárvore está pendendo para a esquerda, logo é realizado um balanceamento para a direita.

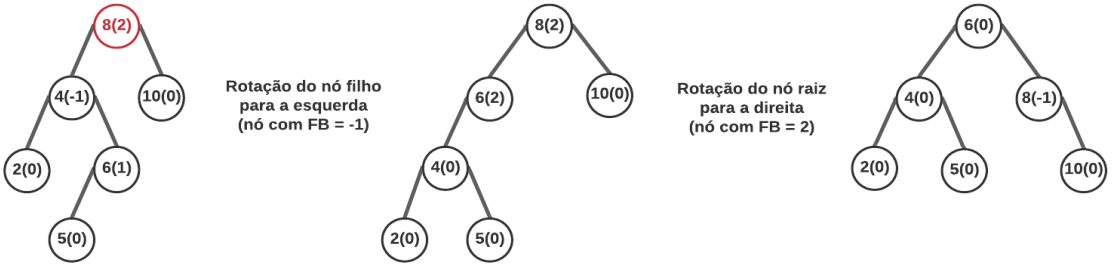


- **1.2 Rotação simples à esquerda:**
  É realizado quando os fatores de balanceamento do nó raiz e do nó filho são negativos, então a subárvore está pendendo para a direita, logo é realizado um balanceamento para a esquerda.



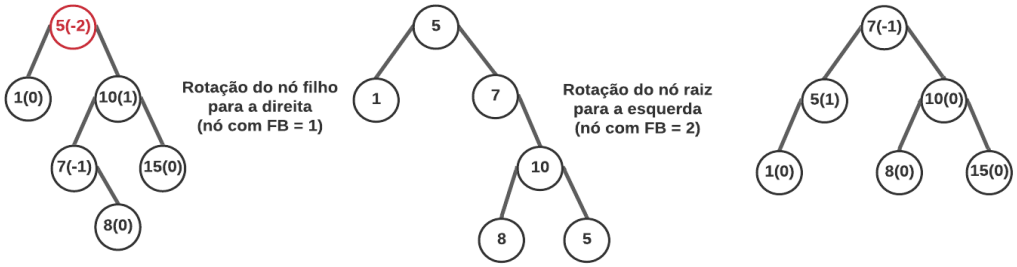
- **2 Caso:**
  Acontece quando os fatores de balanceamento da subárvore e filho têm sinais opostos. Subárvore do nó raiz pende para um lado e subárvore do nó filho pende para outro lado.

  - **1.1 Rotação dupla esquerda direita:**
    Nesse tipo de rotação é necessário realizar primeiro uma rotação para a esquerda e depois uma rotação para a direita.



Subárvore do nó 4 está pendendo para a direita e a árvore do nó 8 está pendendo para a esquerda

- **1.2 Rotação dupla direita esquerda:**
  Nesse tipo de rotação é necessário realizar primeiro uma rotação para a direita e depois uma rotação para a esquerda.



Subárvore do nó 10 está pendendo para a esquerda e a árvore do nó 5 está pendendo para a direita

QUESTÃO 2

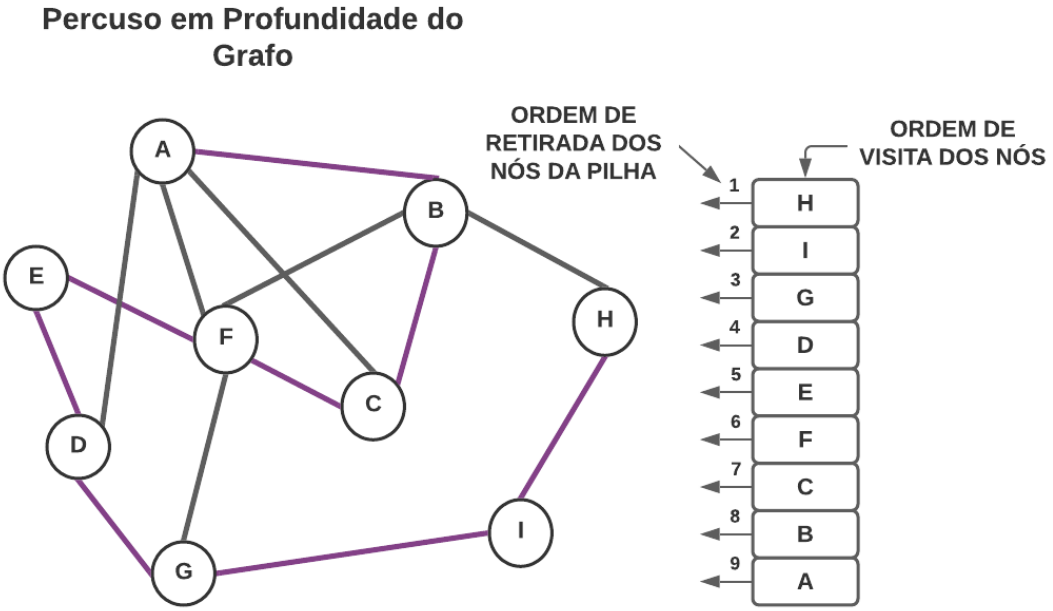
A. Matriz de adjacências

	0(A)	1(B)	2(C)	3(D)	4(E)	5(F)	6(G)	7(H)	8(I)
0(A)	0	1	1	1	0	1	0	0	0
1(B)	1	0	1	0	0	1	0	1	0
2(C)	1	1	0	0	0	1	0	0	0
3(D)	1	0	0	0	1	0	1	0	0
4(E)	0	0	0	1	0	1	0	0	0
5(F)	1	1	1	0	1	0	1	0	0
6(G)	0	0	0	1	0	1	0	0	1
7(H)	0	1	0	0	0	0	0	0	1
8(I)	0	0	0	0	0	0	1	1	0

B. Busca em profundidade

Esse tipo de busca é semelhante a busca em pré-ordem em uma árvore. A estratégia seguida pelo algoritmo de busca em profundidade é buscar mais fundo no grafo sempre que possível. A busca pára quando é encontrada a chave procurada ou quando todos os vértices do grafo forem visitados.
A busca parte de um vértice v e segue explorando as arestas que ainda não foram exploradas, assim indo cada vez mais fundo na estrutura grafo, quando todas as arestas do vértice v são exploradas a busca volta ao vértice anterior ao vértice v para voltar a explorar arestas que ainda não foram exploradas. A busca continua até que tenham sido exploradas todas as arestas do grafo.

Analisando como o grafo da questão pode ser percorrido usando a busca em profundidade.



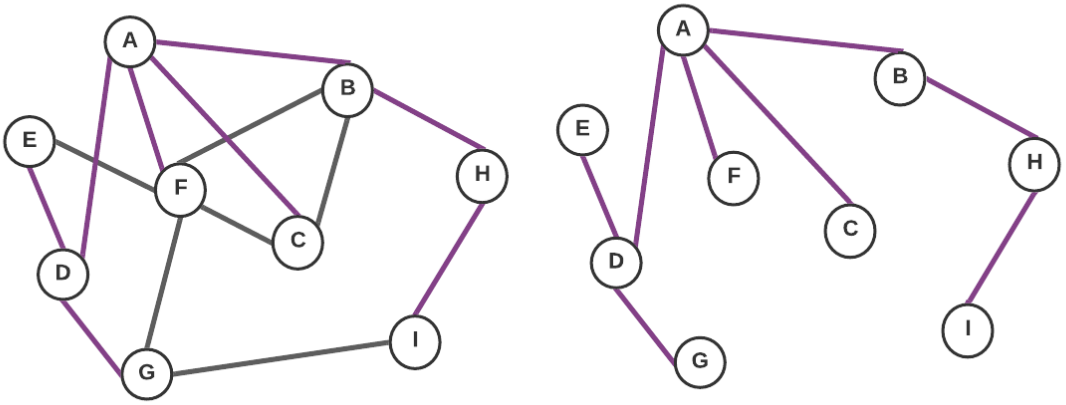
C. Busca em largura

O algoritmo de busca em largura (breadth-first-search - BFS) percorre as arestas do grafo descobrindo todos os vértices atingíveis a partir do vértice inicial.

O algoritmo BFS gera uma árvore BFS com raiz no vértice inicial escolhido, que contém todos os vértices acessíveis determinando o caminho mais curto do vértice inicial até os vértices acessíveis.

O algoritmo utiliza uma fila (FIFO) para armazenar os vértices que já foram visitados, depois os retira até que todos os vértices sejam visitados.

Percuso em Largura do Grafo



FILA (F)	PREDECESSOR (P)
F(0) = [A]	P(A) = []
F(1) = [B, C, D, F]	P(B, C, D, F) = A
F(2) = [C, D, F, H]	P(C, D, F) = A P(H) = B
F(3) = [D, F, H]	P(D, F) = A P(H) = B
F(4) = [F, H, E, G]	P(F) = A P(H) = B P(E, G) = D
F(5) = [H, E, G]	P(H) = B P(E, G) = D
F(6) = [E, G, I]	P(E, G) = D P(I) = H
F(7) = [G, I]	P(G) = D P(I) = H
F(8) = [I]	P(I) = H
F(9) = [] - FILA VAZIA	

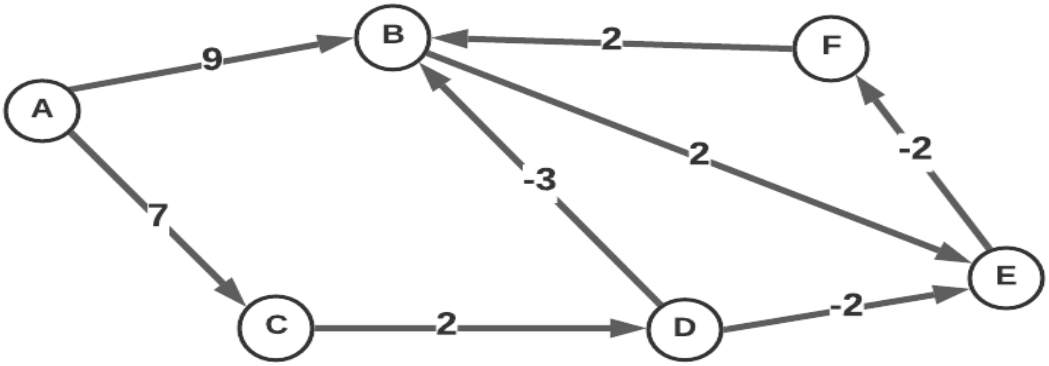
D. Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford indica todas as distâncias mínimas a partir de um determinado vértice inicial. Este algoritmo pode ser utilizado para grafos com arestas de pesos negativos. Para tratar grafos com pesos positivos é

normalmente usado o algoritmo de Dijkstra, pois o mesmo resolve o problema em um tempo menor.

- **Aplicações do Algoritmo:**
  - Problemas de logística
  - Redes de computadores
  - Tráfego humano
  - Sistemas rodoviários e aéreos

- **Exemplo do Algoritmo:**



O número de iterações é dado por $n^{\underline{o}}\ v\acute{e}rtices\ -\ 1\ =\ n^{\underline{o}}\ itera\c{c}\tilde{o}es.$

CHEGADA	SAÍDA
A	

CHEGADA	SAÍDA
B	A, D, F

CHEGADA	SAÍDA
C	A

CHEGADA	SAÍDA
D	C

CHEGADA	SAÍDA
E	D, B

CHEGADA	SAÍDA
F	E

Nº Ite. / Vértices	**A**	**B**	**C**	**D**	**E**	**F**
1	**(0, A)**	**(9, A)**	**(7, A)**	**∞**	**∞**	**∞**
2	**(0, A)**	**(9, A)**	**(7, A)**	**(9, C)**	**(11, B)**	**∞**
3	**(0, A)**	**(6, D)**	**(7, A)**	**(9, C)**	**(7, D)**	**(9, E)**
4	**(0, A)**	**(6, D)**	**(7, A)**	**(9, C)**	**(7, D)**	**(5, E)**
5	**(0, A)**	**(6, D)**	**(7, A)**	**(9, C)**	**(7, D)**	**(5, E)**

Exemplo: Para ir do vértice A até D:

- **Custo total = 9**
- **Caminho = A-C-D**

- **Código do algoritmo:**

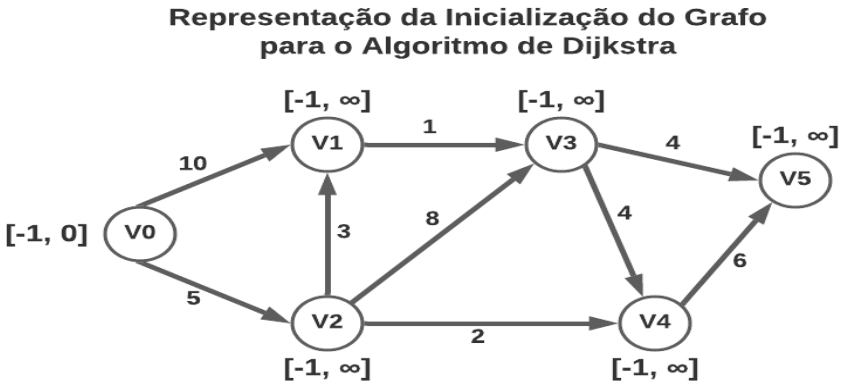
```
void relaxa_aresta(grafo_t *g, int i, int j, float c){
   if(g->v[j].custo > g->v[i].custo + c) {
       g->v[j].custo = g->v[i].custo + c;
       g->v[j].vant = i;
   }
}

void grafo_bellman_ford(grafo_t *g, int s){
   inicializa(g);

   g->v[s].custo = 0.0f;

   for (int k = 1; k < g->n; k++) {
       for (int i = 0; i < g->n; i++) {
           for (aresta_t *a = g->v[i].lista; a != NULL; a = a->prox) {
               int j = a->v;
               relaxa_aresta(g, i, j, a->custo);
           }
       }
   }
}
```

E. Algoritmo de Dijkstra

O algoritmo de Dijkstra calcula o caminho mais curto, em termos do peso total das arestas, entre um nó inicial e todos os demais nós do grafo. O peso total das arestas é a soma dos pesos das arestas que compõem o caminho.

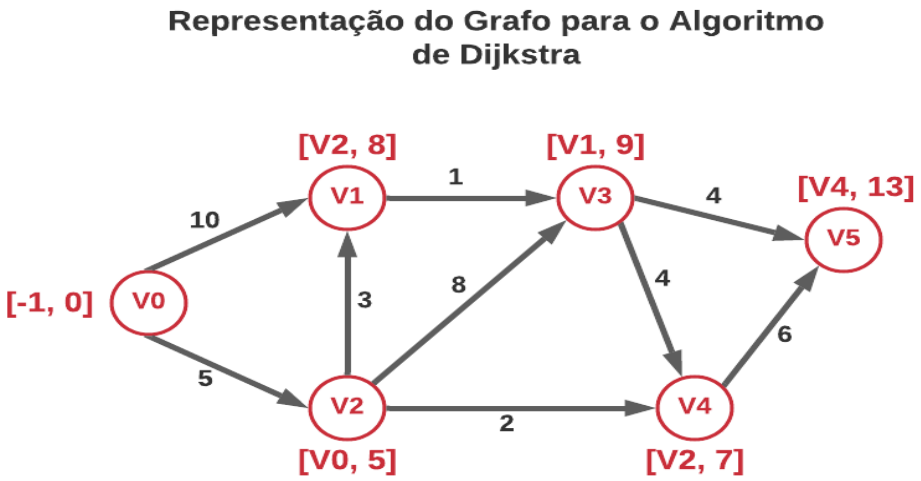
- **Funcionamento do algoritmo:**
  - O algoritmo inicia todos os vértices do grafo como abertos.
  - Inicializa a distância do vértice inicial até ele mesmo como zero, a distância dos demais vértices como infinito e os vértices predecessores como -1 (depende do que se está modelando).
  - Enquanto houver vértices abertos:
    - É escolhido um vértice cuja estimativa da distância seja menor dentre os nós abertos. (No início do algoritmo o vértice escolhido é o vértice inicializado com zero).
    - Quando encontrado, esse vértice é fechado.
    - Relaxa a aresta de todos os vértices adjacentes do vértice em questão.

- **Implementação do algoritmo:**



1. Inicializa todos os vértices com as condições iniciais
2. Enquanto houver vértice aberto:
    a. Pega o V0 que contém a menor estimativa
    b. Fecha o vértice V0
    c. Enquanto houver vértice aberto na adjacência do vértice V0:
        i. Relaxa a aresta:
            1. (V0.0 + 10) < V1.∞, V1 -> [V0, 10]
            2. (V0.0 + 5) < V1.∞, V2 -> [V0, 5]

Esse procedimento é repetido até que todas as arestas tenham sido relaxadas. Resultado final do algoritmo para o grafo de exemplo:



- **Código do algoritmo:**

```
void relaxa_aresta(grafo_t *g, int i, int j, float c){
   if(g->v[j].custo > g->v[i].custo + c) {
```

```
        g->v[j].custo = g->v[i].custo + c;
        g->v[j].vant = i;
    }
}

int extrai_minimo(grafo_t *g) {
    int imin = -1;

    for (int i = 0; i < g->n; i++) {
        if (g->v[i].cor == CINZA) {
            if (imin < 0 || g->v[i].custo < g->v[imin].custo) {
                imin = i;
            }
        }
    }

    return imin;
}

void grafo_dijkstra(grafo_t *g, int s, int t) {
    int i;

    // inicializa vértices como BRANCO que indica ainda não verificados
    inicializa(g);

    g->v[s].custo = 0.0f;
    g->v[s].cor = CINZA ; // CINZA representa frente de avanço

    while ((i = extrai_minimo(g)) >= 0) {
        g->v[i].cor = PRETO ; // PRETO representa vértice processado

        if (i == t) {
            break;
        }

        for (aresta_t *a = g->v[i].lista; a != NULL; a = a->prox) {
            if (g->v[a->v].cor != PRETO) {
                relaxa_aresta(g, i, a->v, a->custo);
                g->v[a->v].cor = CINZA;
            }
        }
    }
}
```