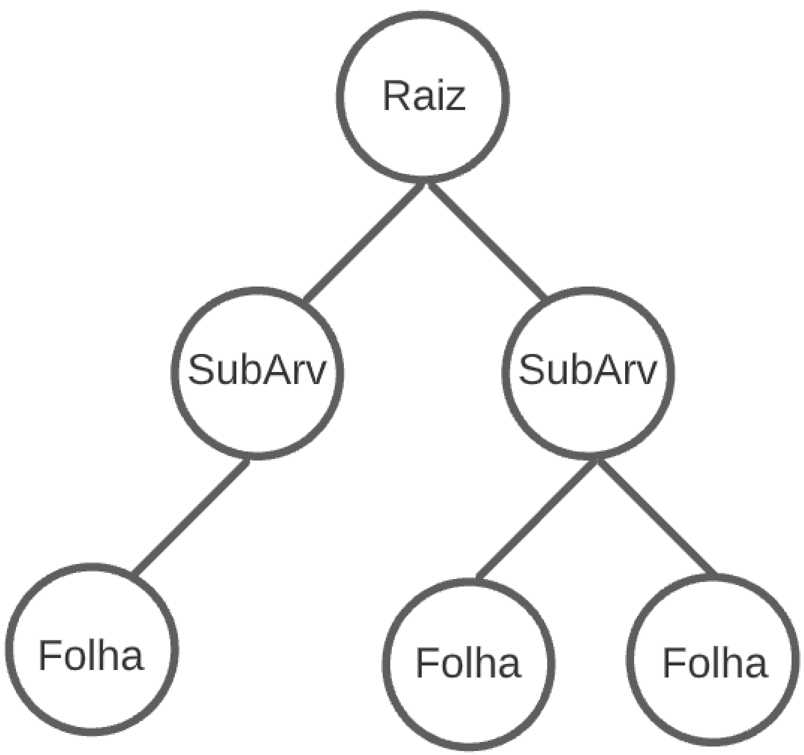
Exercício de Fixação e Aprendizagem III

Questão 1

- Letra **a)**
  - Árvores

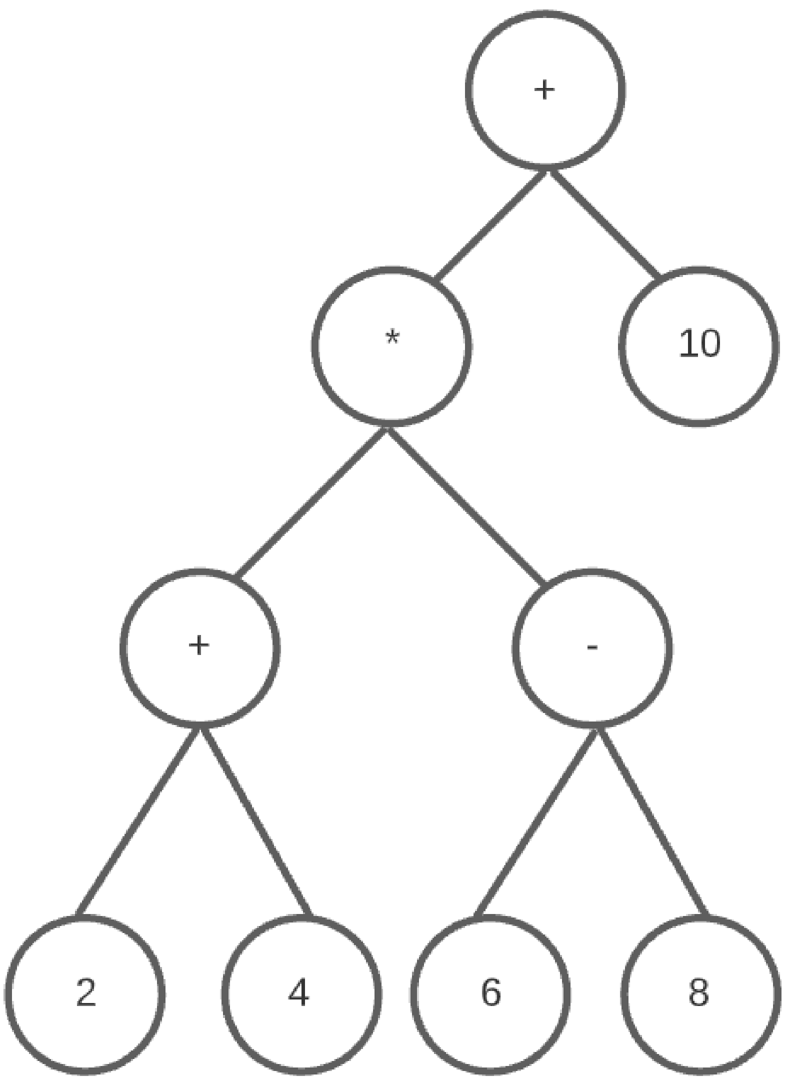
    A árvores é uma estrutura de dados utilizada para a representação hierárquica. Uma árvore é composta por um conjunto de nós, sendo eles nó raiz que é chamado de pai, e os nós subárvores e folhas são chamados de filhos. Os nós que possuem filhos são chamados de nós internos e os que não possuem filhos são chamados de folha ou nós externos.



▼ Árvores Binárias

São árvores que possuem em cada nó no máximo dois filhos. Ela é representada pelo seu nó raiz, e de maneira recursiva pode ser definida como: uma árvore vazia (ponteiro para a raiz nulo) ou um nó raiz que possuem duas subárvores, sendo elas a da direita (sad) e a da esquerda (sae).

Um exemplo de como pode ser utilizado a árvore binária é a avaliação de expressões. Os nós internos representam os operadores e os nós esternos as folha representam os operandos. Na imagem abaixo podemos ver como é representada a expressão (2 + 4) * (6 - 8) + 10.



- ▼ Árvore Binária de Busca

  Na árvore binária de busca usa-se a estrutura da árvore para permitir buscas através de uma chave. A altura de uma árvore é uma medida de busca para encontrar uma determinada chave, a busca de uma determinada chave deve ser feito usando a propriedade de ordem de percurso.

  Abaixo podemos ver a implementação de um código de busca de árvore binária.

```
static ArvNo* busca (ArvNo* r, int v){
  if (r == NULL)
    return NULL;
  else if (r-> info > v)
    return busca (r->esq, v);
  else if(r-> info < v)
    return busca (r->dir, v);
  else
    return r;
}
ArvNo* add_busca (Arv* a, int v){
  return busca(a->raiz, v)
}
```

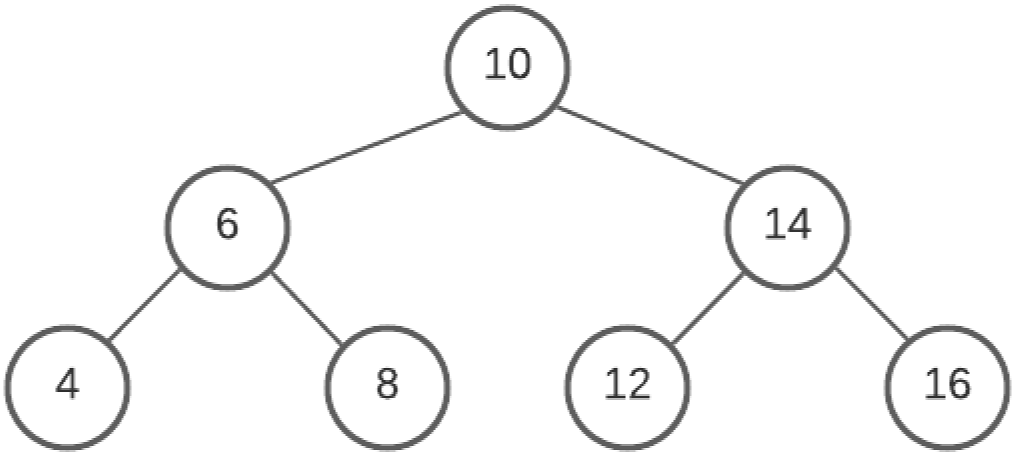
- ▼ Árvore de Busca Balanceada

  A árvore binária balanceada (ou árvore AVL), é uma forma de balanceamento de árvore, pois após várias operações de inserções e remoções na árvore, a mesma tende a ficar desbalanceada, então para acessar qualquer nó de maneira eficiente é indicado usar a árvore binária balanceada.

  O grau de balanceamento depende da ordem na qual as chaves são inseridas na árvore, para que elas já sejam inseridas de maneira eficiente e balanceada. O balanceamento de um nó é definido com a altura da sua subárvore esquerda menos a altura de subárvore a direita, e a altura das subárvores de todo nó nunca diferem de mais 1.

▼ Letra **b)**

Imagem de exemplo para pré-ordem e ordem simétrica



▼ Pré-fixado

- Percorre a árvore de maneira tradicional, ou seja, mostra primeiro a raiz, depois mostra o lado esquerdo ate que chegue no nó *folha* após isso subirá a árvore e mostrará o lado direito da mesma forma do lado esquerdo, até o nó folha.
- Usando como base a árvore apresentada acima, percorrendo-a de maneira **pré-fixada** o resultado seria: `10, 6, 4, 8, 14, 12, 16`
- Percurso **pré-fixado** em código:

```
void imprime(ArvoreNo *raiz) {
  if(!raiz) return;

  printf(" %d ", raiz->info);
  imprime(raiz->esq);
  imprime(raiz->dir);
}
```

▼ Ordem simétrica

- Percorre a árvore de maneira simétrica como o próprio nome já diz, se a árvore estiver balanceada irá mostrar os valores em ordem crescente, do menor para o maior, dessa forma irá percorrer primeiro todo o lado esquerdo, após percorrer o lado esquerdo irá mostrar a raiz e logo em seguida o lado direito.

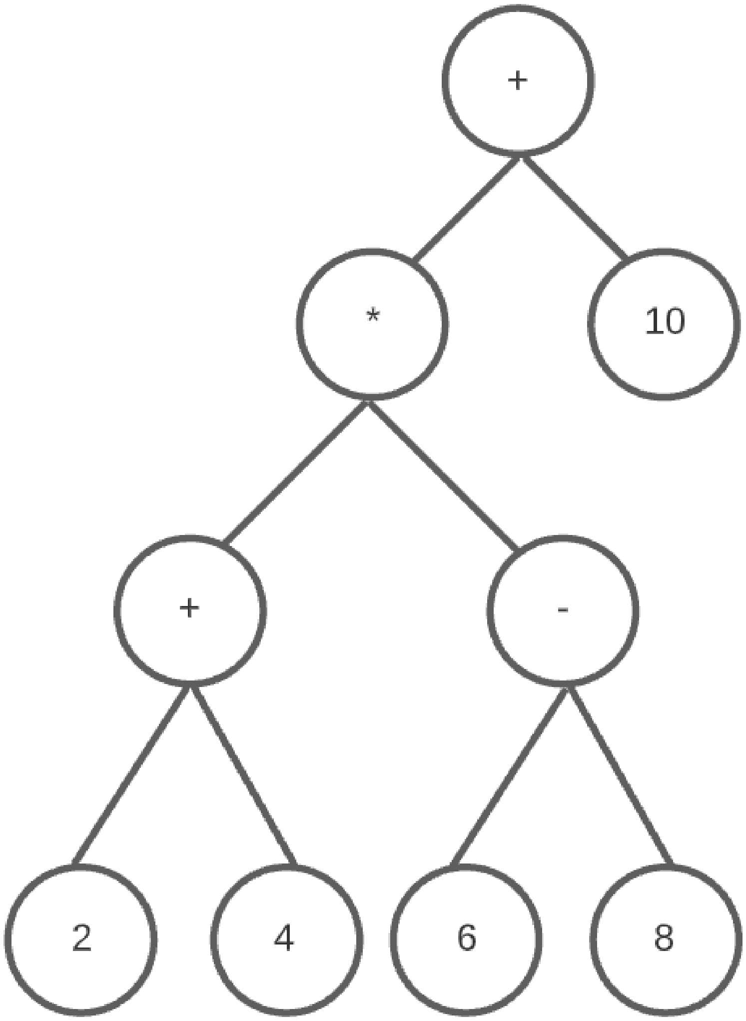
- Diferentemente do modo pré-fixado que vai descendo na árvore e mostrando os elementos, na **ordem simétrica** a árvore será percorrida da seguinte forma: Primeiramente, descerá pela esquerda até que seja encontrado um nó que não tenha filhos, logo no exemplo da imagem acima, a árvore será percorrida até o nó de valor **4**, será verificado que esse nó não possuí filho a esquerda nem a direita e irá mostrar na tela seu valor. Logo em seguida, irá retornar recursivamente para o nó de valor **6**, seu conteúdo será mostrado, depois irá para o filho do lado direito do nó de valor **6** e irá verificar que esse nó não tem filhos, após concluir a verificação irá mostrar seu valor **8** e repetir esse processo até que toda árvore seja mostrada.
- O resultado final será: `4, 6, 8, 10, 14, 12, 16`
- Percurso **ordem simétrica** em código:

```
void imprime(ArvoreNo *raiz) {
  if(!raiz) return;

  imprime(raiz->esq);
  printf(" %d ", raiz->info);
  imprime(raiz->dir);
}
```

▼ **Pós-ordem**

- Percorre a árvore da seguinte maneira, primeiro o lado esquerdo, depois o lado direito e por último mostra a raiz.
- Assim como o percurso na **ordem simétrica** a árvore será percorrida, primeiramente, pelo lado esquerdo até encontrar o valor **NULL**, apos isso irá percorrer o lado direito até encontrar o valor **NULL**, e por último irá a mostrar a raiz, no exemplo da imagem anexada abaixo, o resultado ficaria assim: ` 2 4 + 6 8 - * 10 +`



- Primeiro irá chegar no valor **2**, irá verificar que não existe nó a esquerda e irá que verificar que o nó a direita não existe e irá mostrar o valor **2**, voltará na pilha recursiva para o **+** e irá descer a direita e verificar que os filhos de **4** não existem e irá mostrar o **4** e irá para o **+** como os filhos a esquerda e a direita já foram percorridos irá mostrar o **+** e esse processo será repetido por toda a árvore.
- Percurso **pós-fixado** em código:

```
void imprime(ArvoreNo *raiz) {
  if(!raiz) return;
```

```
    imprime(raiz->esq);
    imprime(raiz->dir);
    printf(" %d ", raiz->info);
  }
```

▼ Letra **c)**

- *main.c*

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

int main(void) {
  Arvore *arvore = NULL;
  int numeros_aleatorios[7] = {10, 6, 8, 4, 14, 16, 12};

  arvore = arv_cria();

  for (size_t i = 0; i < 7; i++) {
    arv_insere(arvore, numeros_aleatorios[i]);
  }

  arv_imprime(arvore);

  printf("A arvore possui %d folha(s)\n", arv_qntd_folhas(arvore));

  arv_retira(arvore, 14);

  arv_imprime(arvore);

  printf("A arvore possui %d folha(s)\n", arv_qntd_folhas(arvore));

  arv_libera(arvore);
}
```

- *arvore.h*

```
#ifndef _ARVORE_H
#define _ARVORE_H

typedef struct arvno ArvoreNo;
typedef struct arv Arvore;

Arvore* arv_cria(void);
void arv_insere(Arvore *arvore_principal, int valor);
void arv_retira(Arvore *arvore_principal, int valor);
void arv_libera(Arvore *arvore_principal);
void arv_imprime(Arvore *arvore_principal);
```

```
int arv_pertence(Arvore *arvore_principal, int valor);
void arv_busca(Arvore *arvore_principal, int valor);
int arv_qntd_folhas(Arvore *arvore_principal);

#endif
```

- *arvore.c*

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

#ifndef _ARVORE_C
#define _ARVORE_C

struct arvno {
  int info;
  ArvoreNo* esq;
  ArvoreNo* dir;
};

struct arv {
  ArvoreNo* raiz;
};

Arvore* arv_cria(void) {
  Arvore *nova_arvore = NULL;

  nova_arvore = (Arvore*) malloc(sizeof(Arvore));
  if(!nova_arvore) return 0;

  nova_arvore->raiz = NULL;

  return nova_arvore;
}

ArvoreNo* insere(ArvoreNo *raiz, int valor) {
  if(!raiz) {
    ArvoreNo *novo_elemento = NULL;

    novo_elemento = (ArvoreNo*) malloc(sizeof(ArvoreNo));
    if(!novo_elemento) return NULL;

    novo_elemento->info = valor;
    novo_elemento->dir = NULL;
    novo_elemento->esq = NULL;

    return novo_elemento;
  } else {
    if (valor < raiz->info) {
      raiz->esq = insere(raiz->esq, valor);
```

```
    } else {
      raiz->dir = insere(raiz->dir, valor);
    }

    return raiz;
  }
}

void arv_insere(Arvore* arvore_principal, int valor) {
  arvore_principal->raiz = insere(arvore_principal->raiz, valor);
}

ArvoreNo* retira(ArvoreNo *raiz, int valor) {

  if(!raiz) {
    // Caso o elemento nao exista na arvore
    return NULL;
  } else if(valor > raiz->info) {
    raiz->dir = retira(raiz->dir, valor);
  } else if(valor < raiz->info) {
    raiz->esq = retira(raiz->esq, valor);
  } else {
    // Se o no nao possuir filhos
    if(!raiz->esq && !raiz->dir) {
      free(raiz);
      raiz = NULL;
    } else if(!raiz->esq) {
      ArvoreNo *aux = raiz;
      raiz = raiz->dir;
      free(aux);
    } else if(!raiz->dir) {
      ArvoreNo *aux = raiz;
      raiz = raiz->esq;
      free(aux);
    } else {
      ArvoreNo *aux = raiz->esq;

      // Ira percorrer a arvore ate achar um filho da direita que seja nulo
      while(aux->dir) {
        aux = aux->dir;
      }

      // Troca as informacoes com o elemento que sera removido
      raiz->info = aux->info;
      aux->info = valor;

      raiz->esq = retira(raiz->esq, valor);
    }
  }

  return raiz;
}

void arv_retira(Arvore *arvore_principal, int valor) {
```

```
  arvore_principal->raiz = retira(arvore_principal->raiz, valor);
}

void libera(ArvoreNo *raiz) {
  if(!raiz) return;

  libera(raiz->esq);
  libera(raiz->dir);
  free(raiz);
}

void arv_libera(Arvore *arvore_principal) {
  if(arvore_principal) {
    libera(arvore_principal->raiz);
    free(arvore_principal);
  }
}

void imprime(ArvoreNo *raiz) {
  if(!raiz) return;

  printf(" %d ", raiz->info);
  imprime(raiz->esq);
  imprime(raiz->dir);
}

void arv_imprime(Arvore *arvore_principal) {
  imprime(arvore_principal->raiz);
  printf("\n");
}

int pertence(ArvoreNo *raiz, int valor) {
  if(!raiz) return 0;

  return valor == raiz->info ||
    pertence(raiz->esq, valor) ||
    pertence(raiz->dir, valor);
}

int arv_pertence(Arvore *arvore_principal, int valor) {
  return (pertence(arvore_principal->raiz, valor));
}

ArvoreNo* busca(ArvoreNo *raiz, int valor) {
  if(!raiz) {
    return NULL;
  } else if(valor > raiz->info) {
    return busca(raiz->dir, valor);
  } else if(valor < raiz->info) {
    return busca(raiz->esq, valor);
  } else {
    return raiz;
  }
}
```

```
void arv_busca(Arvore *arvore_principal, int valor) {
  arvore_principal->raiz = busca(arvore_principal->raiz, valor);
}

int conta_folha(ArvoreNo *raiz) {
  if(!raiz) {
    return 0;
  } else if(raiz->esq == NULL && raiz->dir == NULL) {
    return 1;
  } else {
    return conta_folha(raiz->esq) + conta_folha(raiz->dir);
  }
}

int arv_qntd_folhas(Arvore *arvore_principal) {
  return conta_folha(arvore_principal->raiz);
}

#endif
```

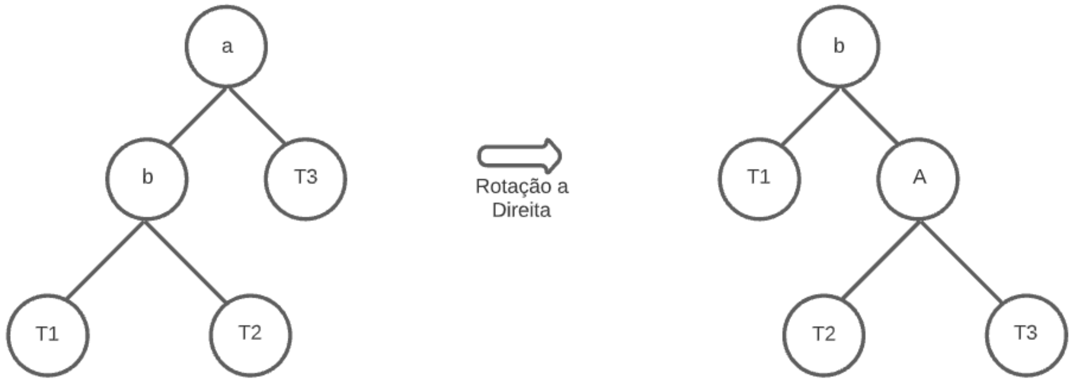
▼ Letra d)

As operações que devem ser aplicadas nas árvores binárias para torna-las árvores AVL são: rotação direita, rotação esquerda, rotação dupla direita e rotação dupla esquerda.

▼ Rotação Direita

Quando um nó estiver desbalanceado do lado esquerdo, ou seja, quando o lado esquerdo for maior que o lado direito, é aplicado a rotação a direita.

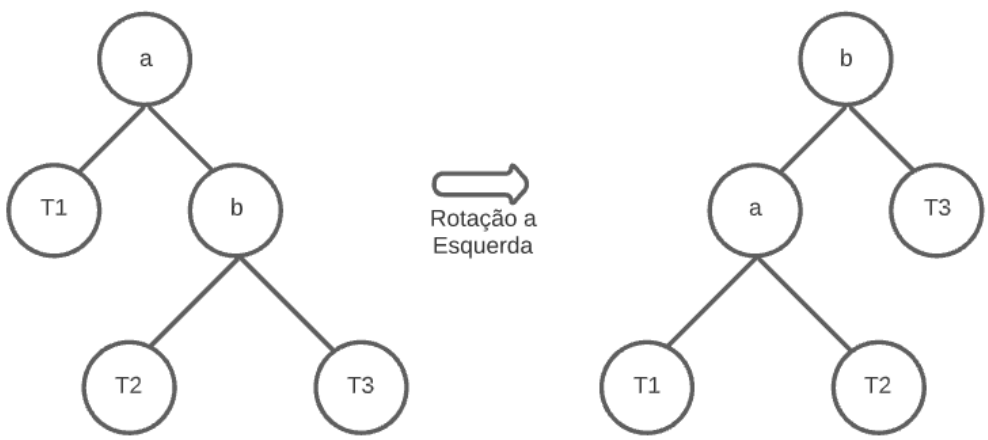
Como podemos ver na imagem abaixo, o nó "a" está desbalanceado e é feita a rotação a direita.



▼ Rotação Esquerda

Quando um nó estiver desbalanceado do lado direito, ou seja, quando o lado direito for maior que o lado esquerdo, é aplicado a rotação a esquerda.

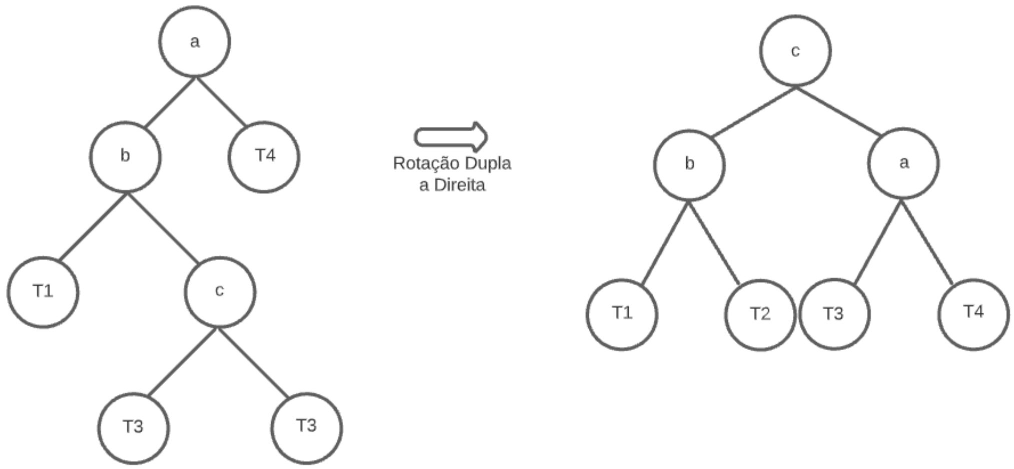
Como podemos ver na imagem abaixo, o nó "a" está desbalanceado e é feita a rotação a esquerda.



▼ Rotação Dupla Direita

É aplicada a rotação dupla a Direita quando um nó tem seu lado esquerdo maior, mas o lado direito do filho desse nó também é maior, então é feita a rotação para balancear e ficar de tamanho igual.

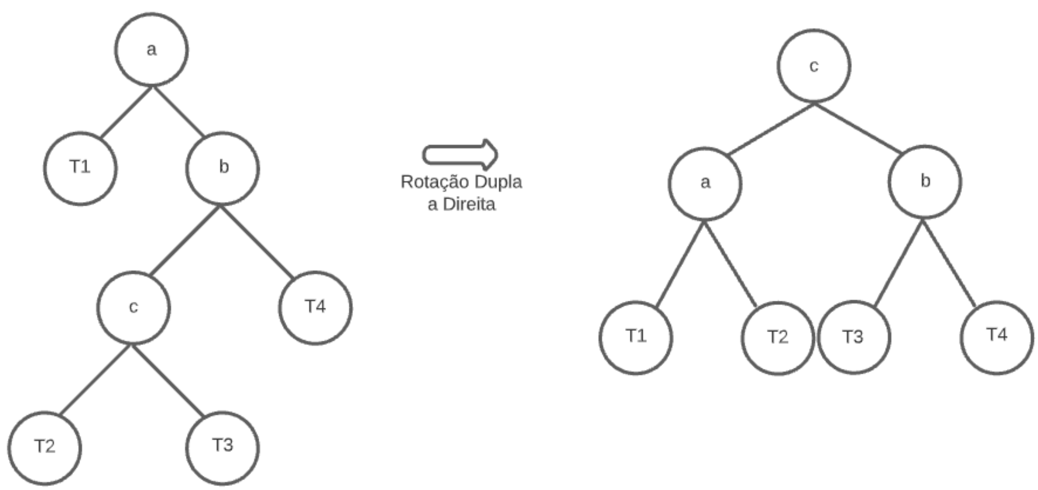
Para ficar mais didático a imagem abaixo ilustrara como é feita a rotação dupla a direita:



- ▼ Rotação Dupla Esquerda

  A rotação dupla a esquerda, é feita quando um nó tem seu lado direito maior, mas o lado esquerdo do filho desse nó também é maior, então é feita a rotação para balancear e ficar de tamanho igual.

  A imagem abaixo mostra como é feita a rotação dupla a esquerda:



Questão 2

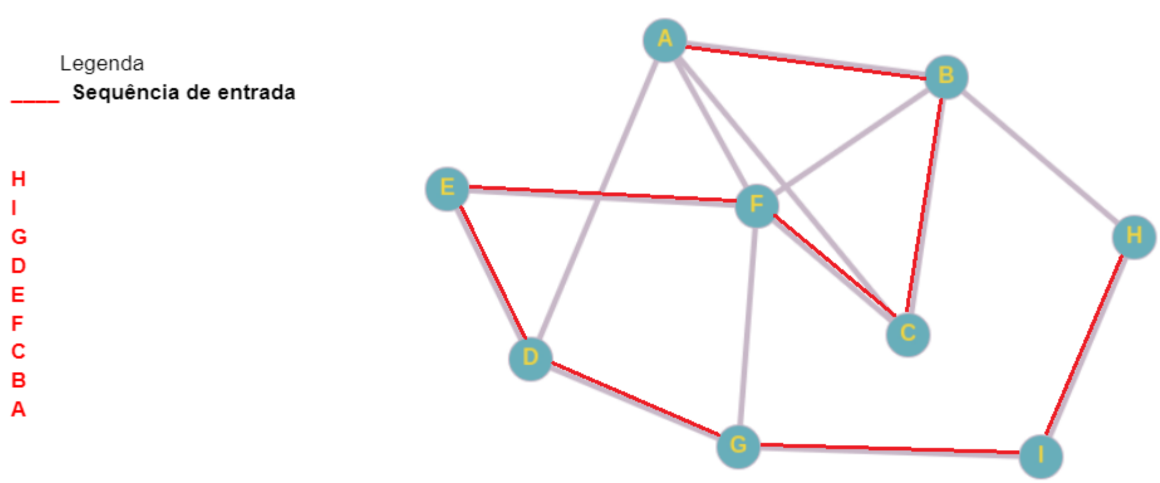
- ▼ Letra **a)**
  - Matriz de adjacência

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

▼ Letra **b)**

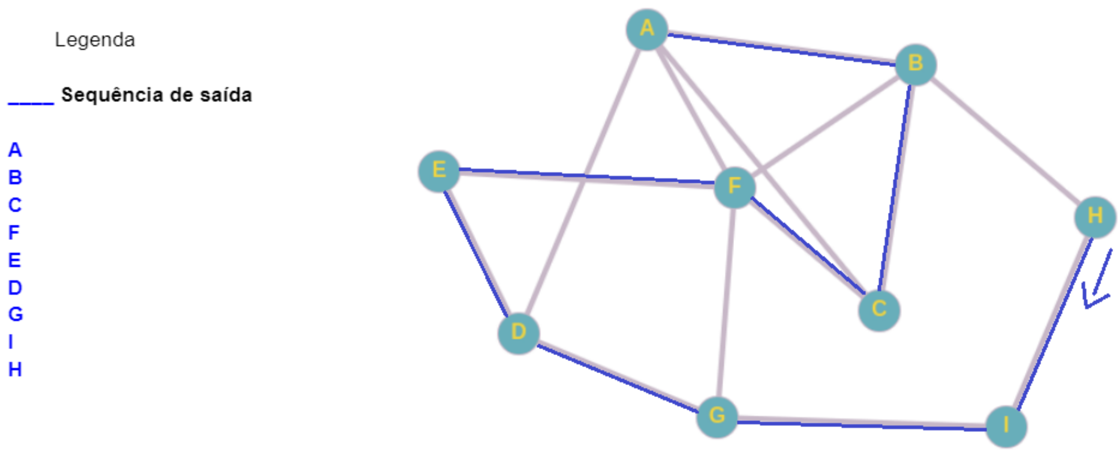
Primeiramente explicando como funciona a busca em profundidade, como o nome já implica, o objetivo é buscar o "mais fundo" do grafo, ele explora as arestas partindo de um vértice primário conhecido e do qual ainda arestas inexploradas. Logo após todas as arestas do vértice primário for explorada a busca "regressa pelo mesmo caminho", para explorar as outras arestas que partem do vértice inicial. Além disso, vale salientar que a busca de profundidade usa algoritmo de pilha.

Partindo do nó A é desse jeito que o grafo deve ser percorrido em profundidade:

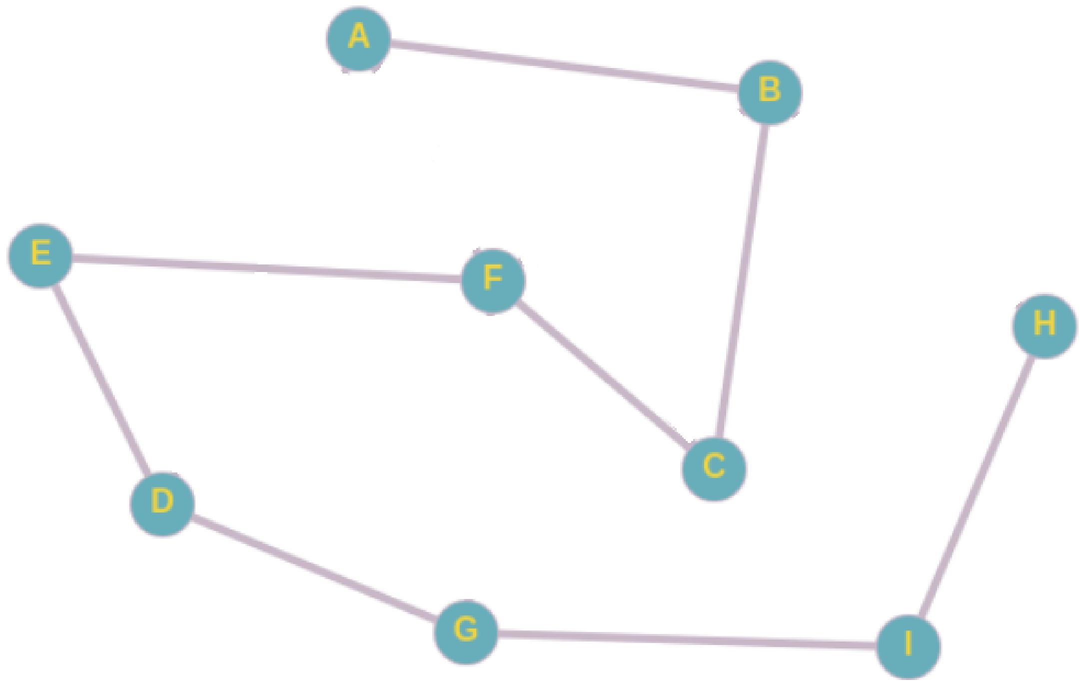


Começando com o nó **A**, logo em seguida vai para o nó **B**, vai para a aresta maior que é a **C**, depois o único lugar que ele pode ir é o **F**, chega a maior aresta e vai para o **E**, como não tem outra opção ele vai para o **D**, depois para o **G**, em seguida vai para o nó **I**, e por fim vai para o **H**.

Sabendo que todos os nós foram visitados, agora é hora de "regredir" retirando da pilha começando de onde terminou, o **H**:



Assim, todos os nós vão ser retirados da pilha, e a Busca por Profundidade foi realizada.



E essa foi o caminho percorrido.

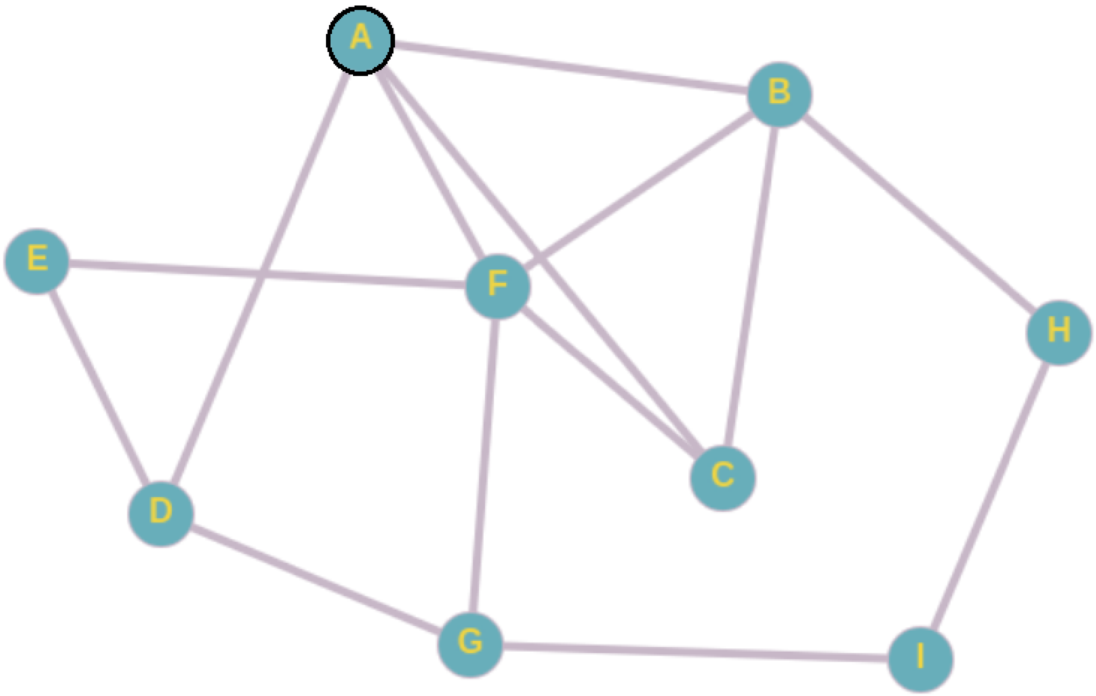
▼ Letra **c)**

A Busca em Largura ou BFS (breadth-first search), explora todos as arestas a partir de um vértice inicial, com o objetivo de "descobrir" cada vértice que pode ser alcançado a partir do vértice fonte. Essa busca sempre parte de um vértice que é considerado "raiz", e deve-se visitar todos os vértices próximos da raiz primeiro, antes de ir para os vértices mais distantes do grafo. Além disso, a estrutura que mais é usada para implementar a Busca em Largura é a fila.

Partindo do nó **A** é desse jeito que o grafo deve ser percorrido em largura:

▼ FILA

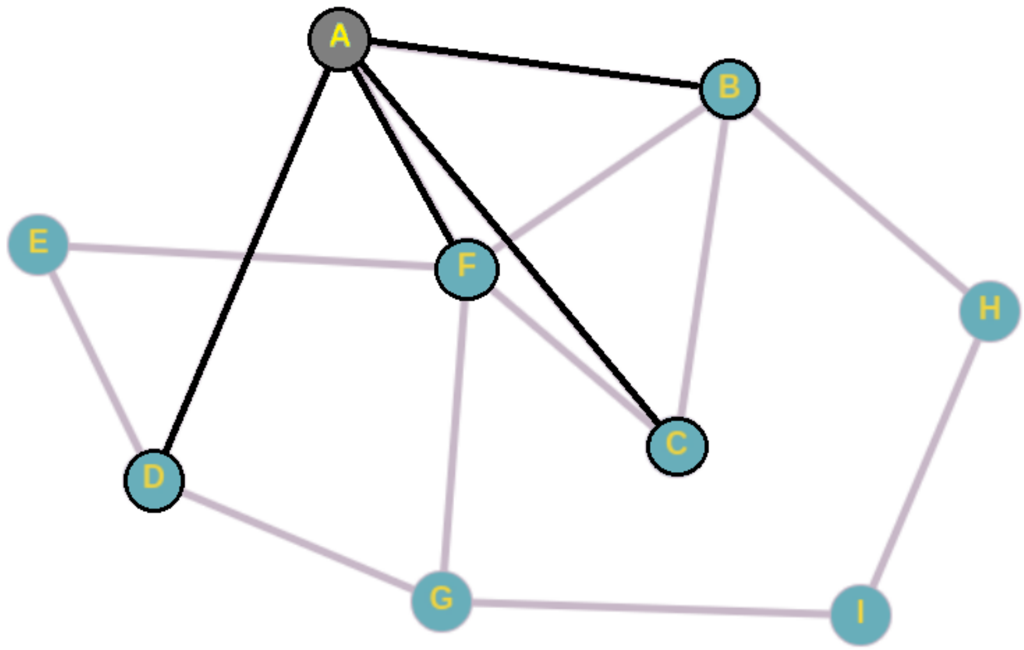
A



Como o nó **A** é o raiz, agora vai visitar os nós que então ligados a ele e remover o **A** da fila, que são o **D**, **F**, **C** e **B:**

▼ FILA

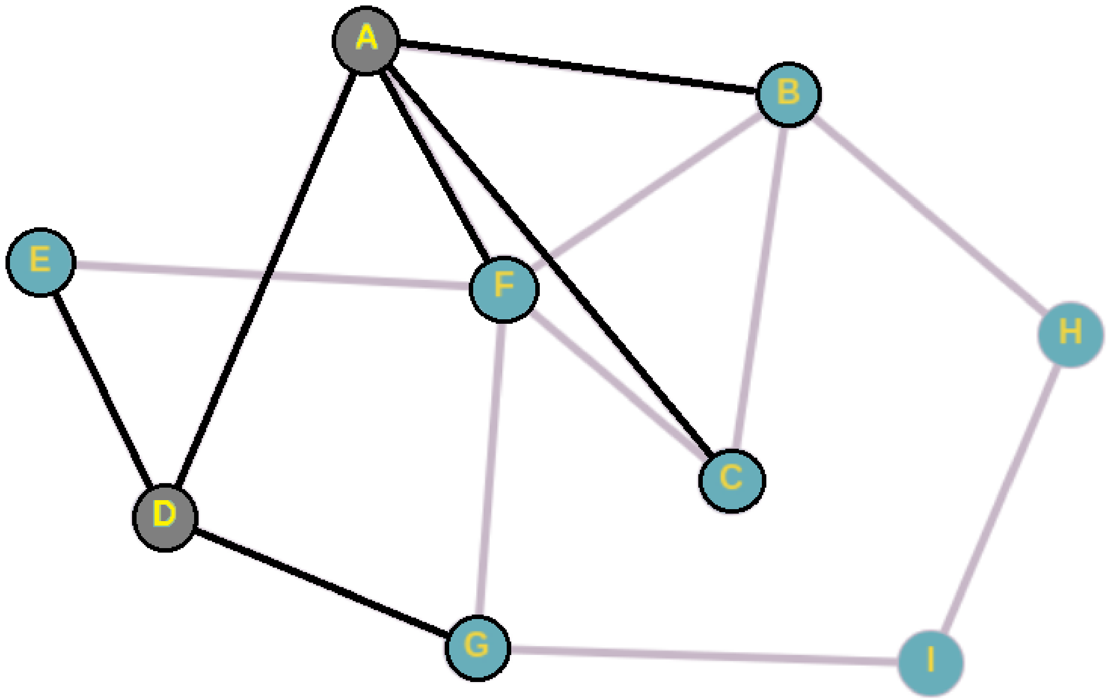
D F C B



Escolhendo o nó **D** como raiz deve-se visitar suas arestas que são o **E** e o **G**, e remover o **D** da fila:

▼ FILA

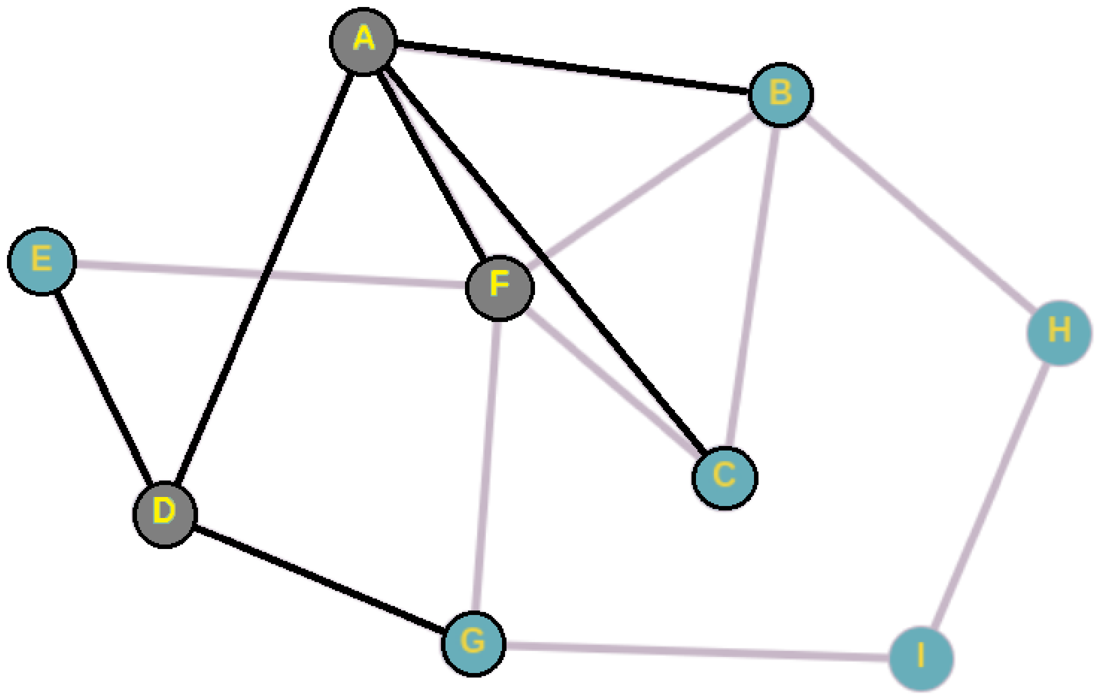
F C B E G



Escolhendo o nó **F** como nó raiz, deve-se visitar suas arestas, mas como suas arestas já foram visitadas ela apenas vai sair na fila:

▼ Fila

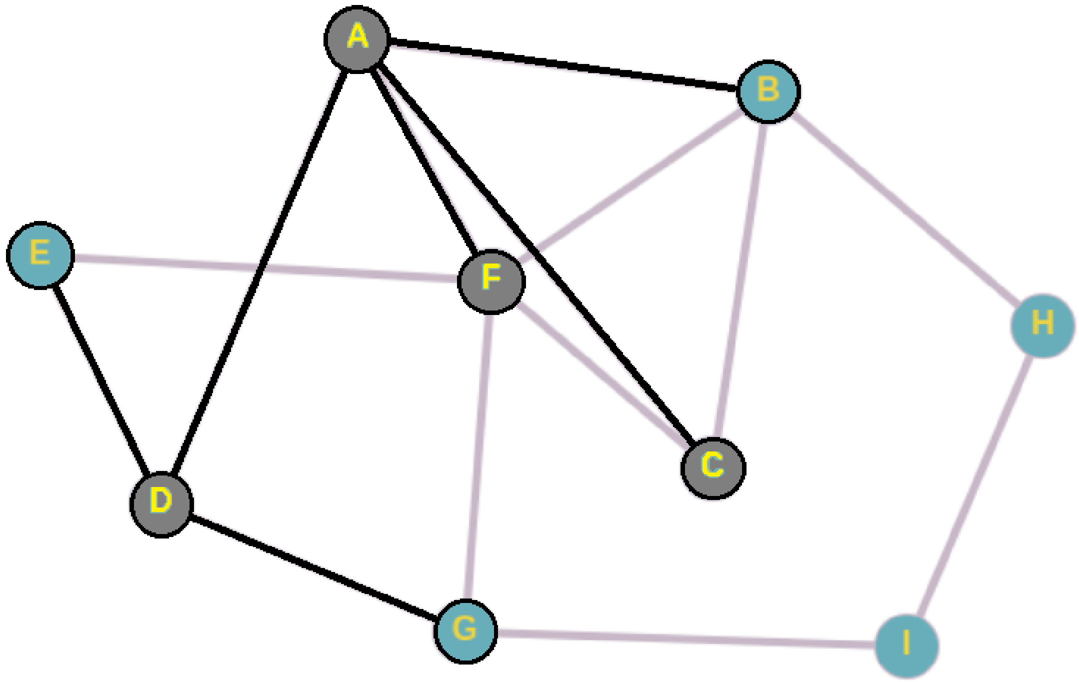
C B E G



Escolhendo o nó **C** como nó raiz, deve-se visitar suas arestas, mas como suas arestas já foram visitadas ela apenas vai sair na fila:

▼ Fila

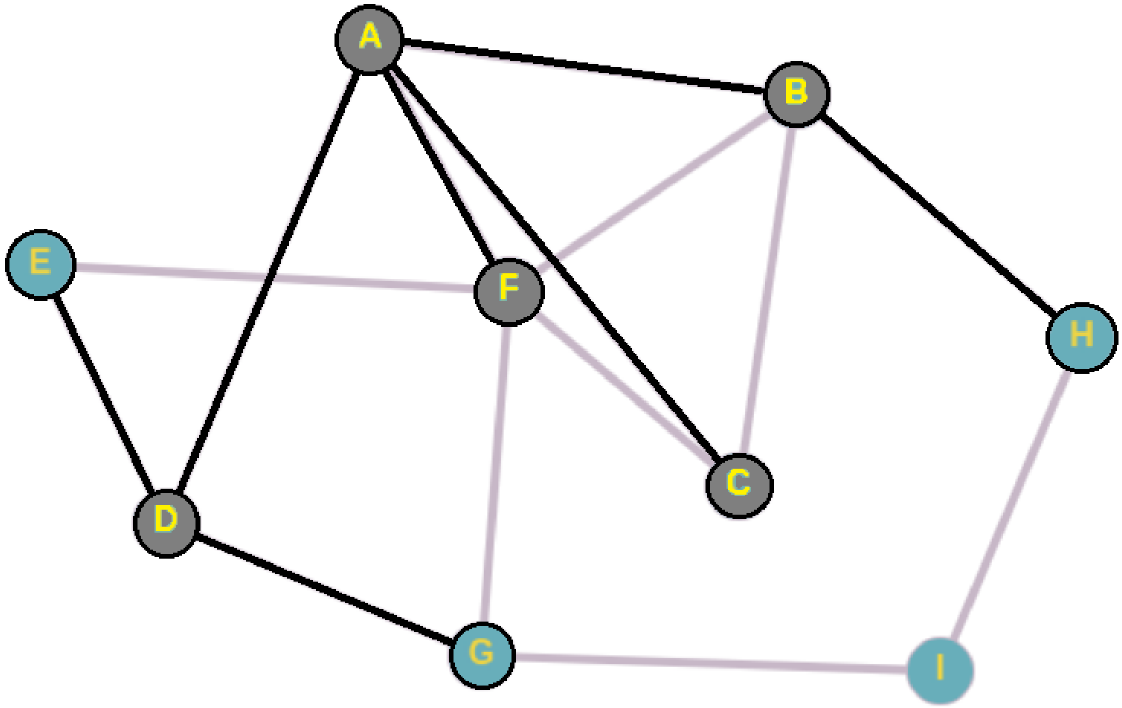
B E G



Escolhendo o nó **B** como raiz a aresta que vai ser visitada é a **H**, então vai ser adicionada o nó **H** na fila e retirar o nó **B**:

▼ Fila

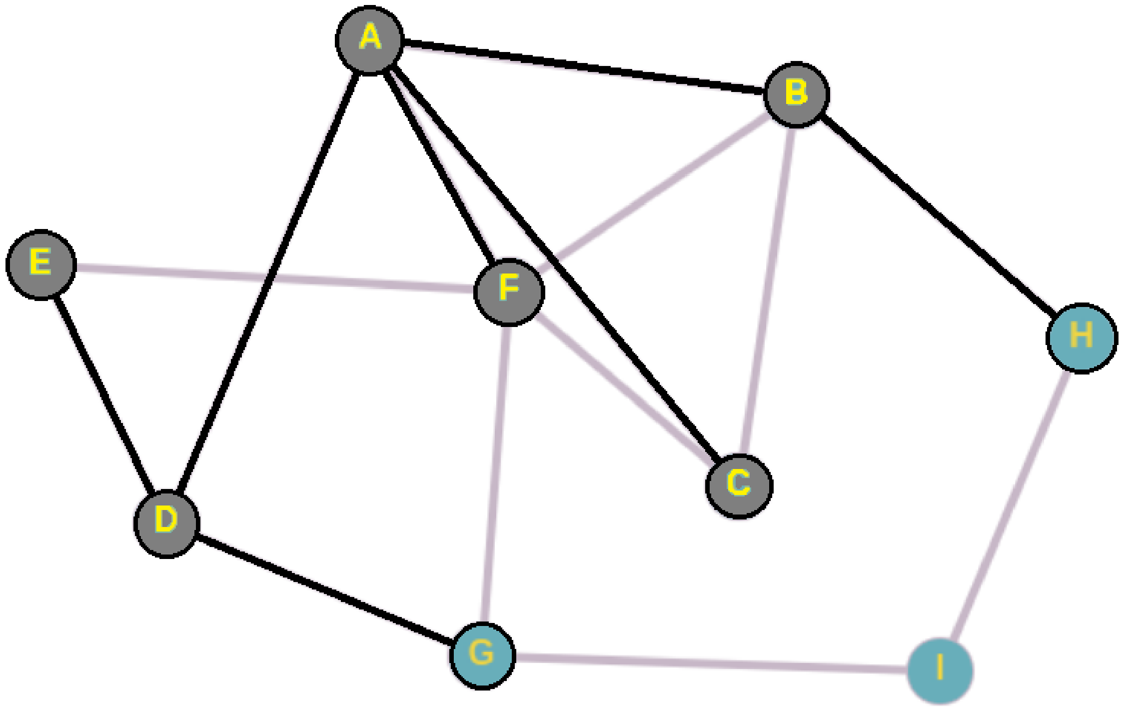
E G H



Escolhendo o nó **E** como raiz, e suas arestas já foram visitadas, ele só sai da fila:

▼ Fila

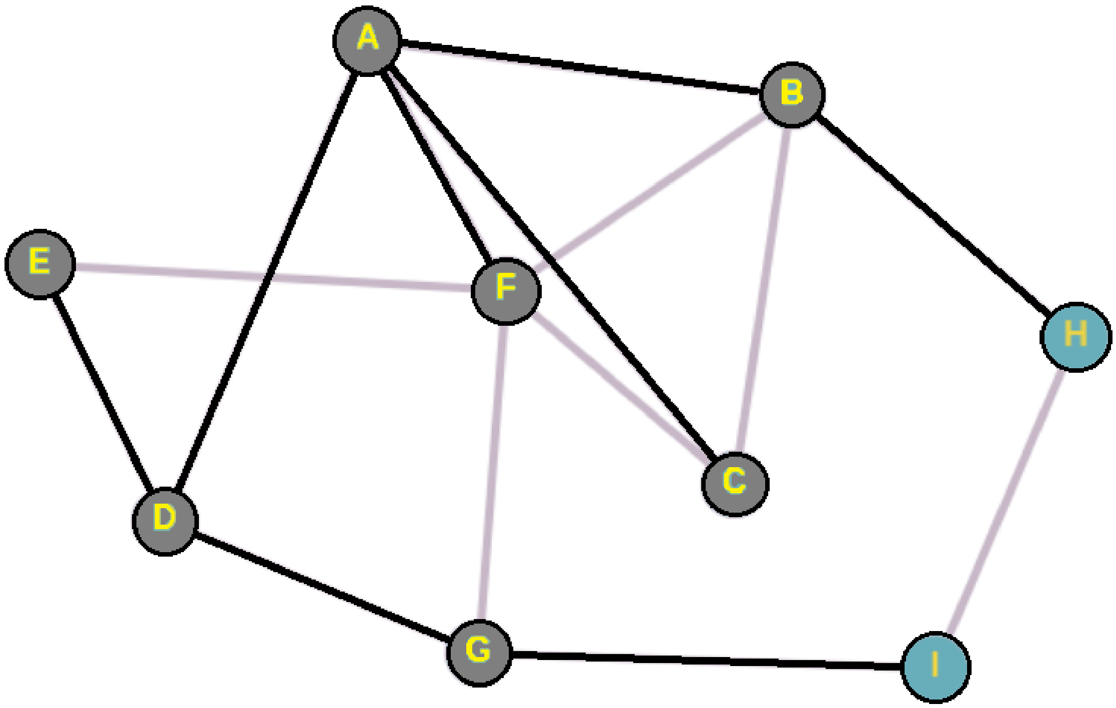
G H



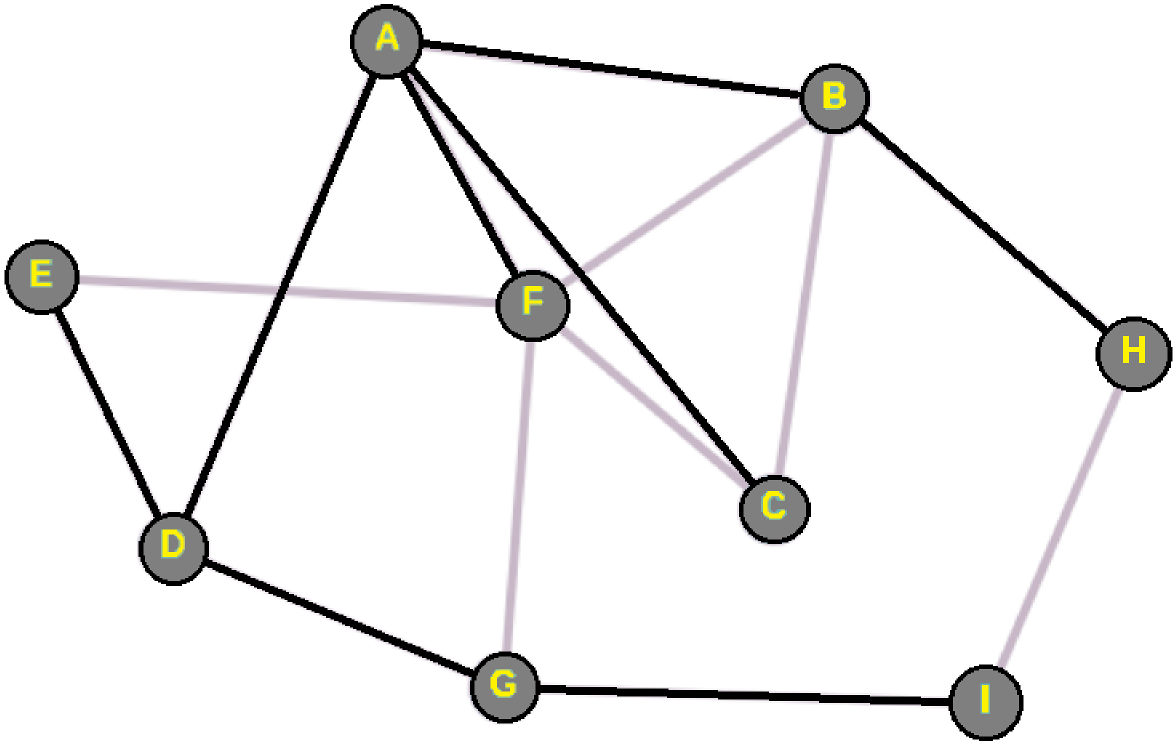
Escolhendo o nó **G** como raiz a aresta que vai ser visitada é a **I**, então vai ser adicionada o nó **I** na fila e retirar o nó **G**:

▼ Fila

H I



Escolhendo o nó **H** como raiz e sabendo que suas arestas já foram visitadas ele só sai da fila, igualmente com o nó **I** que vai ser escolhido depois como raiz e sairá da fila, terminando a Busca por Largura:



▼ Letra **d)**

Algoritmo de Bellman-Ford

- *pseudo-código*

Algoritmo grafo_bellman_ford(grafo_principal, inicial)

inicializa(grafo_principal) % Atribui infinito a todos os vertices do grafo
grafo_principal[inicial].custo $\leftarrow$ 0.0

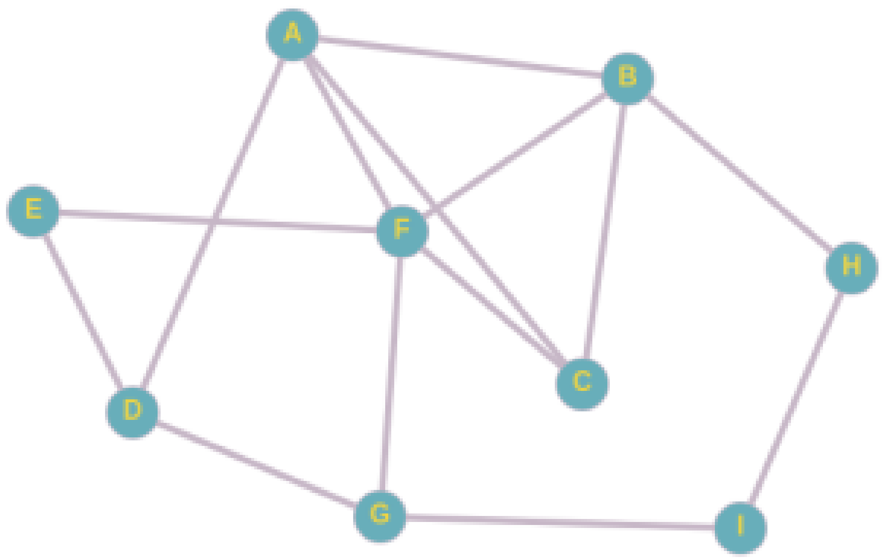
para $k \leftarrow 1$ **ate** $n-1$ **faça**
para $i \leftarrow 0$ **ate** $n-1$ **faça**
para toda aresta ij $\in$ grafo_principal **faça**
para toda aresta ij $\in$ grafo_principal **faça**
j $\leftarrow$ aresta.vertice
relaxa(grafo_principal, i, j, aresta.custo)
fimpara

fimpara

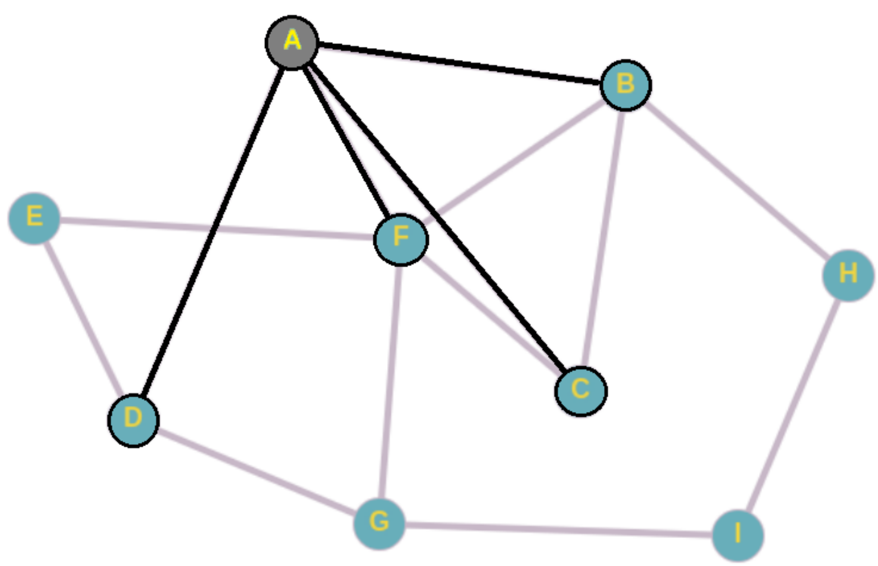
fimpara

fimpara

- Funcionamento
  - É a estratégia mais simples para achar os caminhos mínimos de um vértice para qualquer outro vértice de um grafo.
  - Seu funcionamento consiste em relaxar todas as arestas do grafo até que todas as arestas estejam relaxadas
  - Inicialmente, o custo atrelado ao vértice origem é **zero** e o custo atrelado aos demais vértices é infinito
  - Como foi dito anteriormente, todas as arestas precisam ser relaxadas. Na primeira iteração apenas as arestas ligadas ao vértice serão relaxadas e o processo de relaxamento será realizado para todos as arestas restantes do grafo até que nenhum precise ser mais relaxa, ou seja, a condição de parada do processamento de relaxamento é todos os nós terem sido relaxados.
  - Dada a imagem abaixo, um exemplo prático do algoritmo de Bellman-Ford seria o seguinte:



- Assumindo que a origem é o **A**
- Inicialmente, o custo de **A** será **zero** e todos os outros terão custo **infinito**
- Após isso as aresta que adjacentes ao vértice **A** precisam ser relaxadas essas arestas serão: `AB, AC, AD, AF`



- Esse processo será repetido até que não sobre nenhuma aresta sem estar relaxada, ou seja, até que todos estejam relaxados

▼ Letra **e)**

Algoritmo de Dijkstra

- *pseudo-código*

  Algoritmo grafo_dijkstra(grafo_principal, inicial, final)

  inicializa(grafo_principal) % Atribui infinito a todos os vertices do grafo

  grafo_principal[inicial].custo $\leftarrow$ 0.0

  grafo_principal[inicial].cor $\leftarrow$ CINZA

  $i \leftarrow \emptyset$

  enquanto ($i \leftarrow$ extrai_minimo(grafo_principal)) $\neq 0$) **faça**

  grafo_principal[i].cor $\leftarrow$ PRETO

  se (i == final)

  interrompa

  fimse

  para toda aresta $ij \in$ grafo_principal **faça**

  se (grafo_principal[aresta.vertice].cor $\neq$ PRETO)

  relaxa(grafo_principal, inicial, aresta.vertice, aresta.custo)

  grafo_principal[aresta.vertice].cor $\leftarrow$ CINZA

  fimse

  fimpara

  fimenquanto

- Funcionamento
  - O algoritmo de Dijkstra busca otimizar a procura do menor caminho ou do caminho de menor custo em grafo.

- Durante a execução busca-se dar prioridade as arestas que pertecem a uma *"frente de avanço"*. Inicialmente, assim como no algoritmo de Bellman-Ford o vértice de origem é zero o vértice de origem Dijsktra será o único na frente de avanço, apenas inicialmente.
- O processo é iniciado e o vértice de origem será removido da lista de busca e suas arestas serão relaxadas e o vértice cuja a aresta possuí o menor custo será adicionado na *"frente de avanço"*, esse processo será repetido n vezes, onde o n é o número de vértices.
- Implementando esse algoritmo em código a *"frente de avanço"* pode ser marcada com uma cor CINZA e o vértice anterior pode ser marcado de PRETO e dessa forma após a execução os vértices que compõem o caminho otimizado estarão marcados de PRETO.