



Maratona**CIn**



SELETIVA AULA 3

Grafos 1

Giovanna Mafra <gmm8>
Guilherme Maranhão <gmsmr>

rev. 1.1





Maratona**CIn**



O que é um Grafo?

O que é um Grafo?



Maratona

CIn

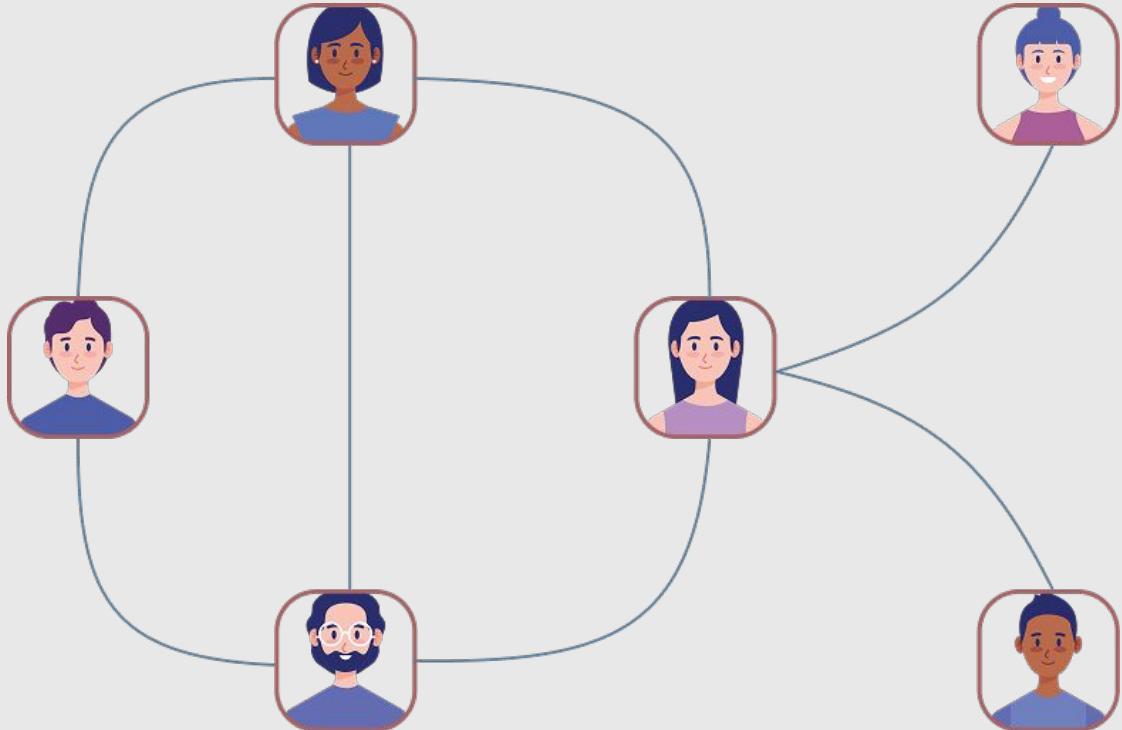
Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Um grafo é uma estrutura matemática composta por vértices (ou nós) e arestas (ou ligações) que conectam pares de vértices. Componentes principais:

- Vértices: Representam os objetos. Por exemplo, as pessoas em uma rede social ou cidades em um mapa.
- Arestas: Representam as relações entre os vértices. Por exemplo, a amizade entre duas pessoas ou ruas entre duas cidades.

Exemplo de Grafos



- *Relação de amizade entre pessoas*



- *Mapa de uma cidade*

Conceitos

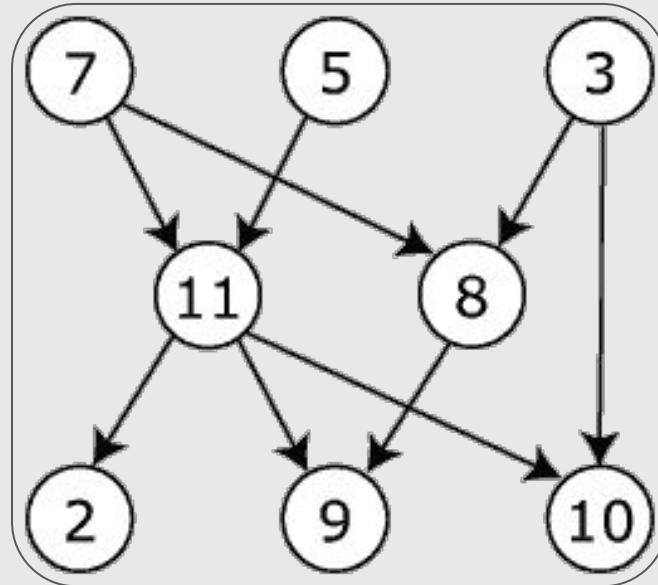


MaratonaCIn

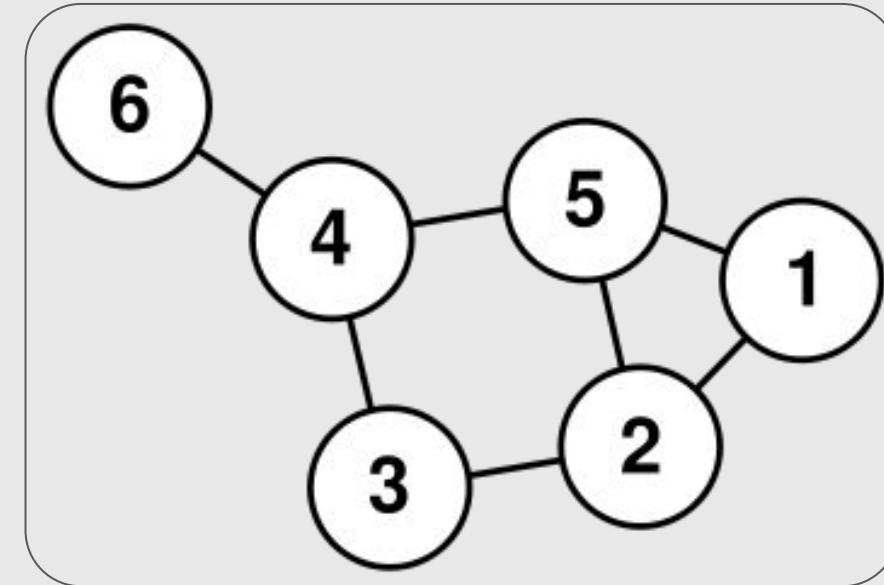
Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 UFPE

- Areias podem possuir direção ou não (bidirecional)



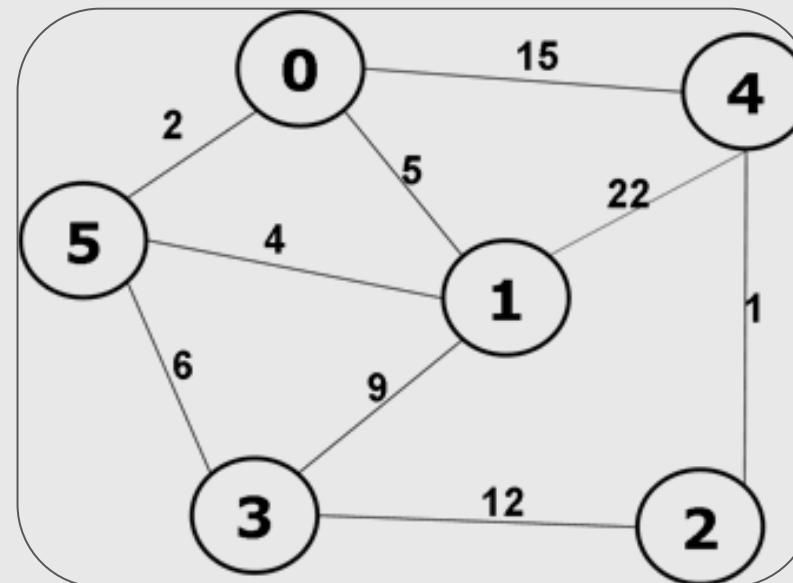
Grafo direcionado



Grafo não-direcionado

Conceitos

- Arestas podem possuir peso/custo



Conceitos

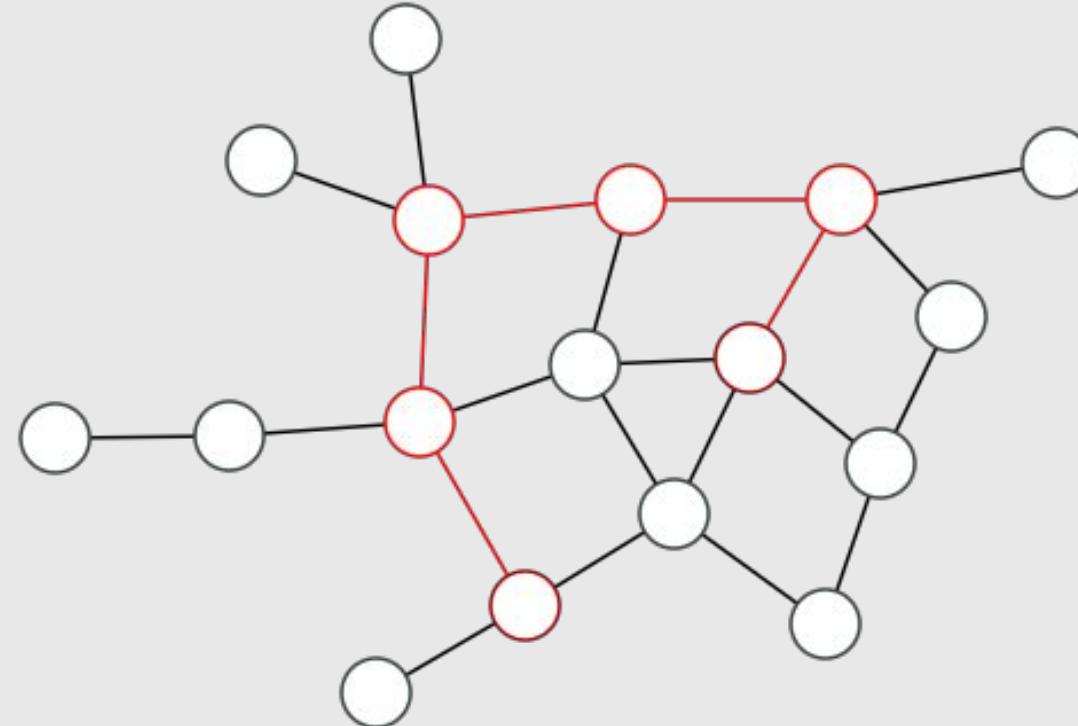


MaratonaCIn

Centro de
Informática
UFPE

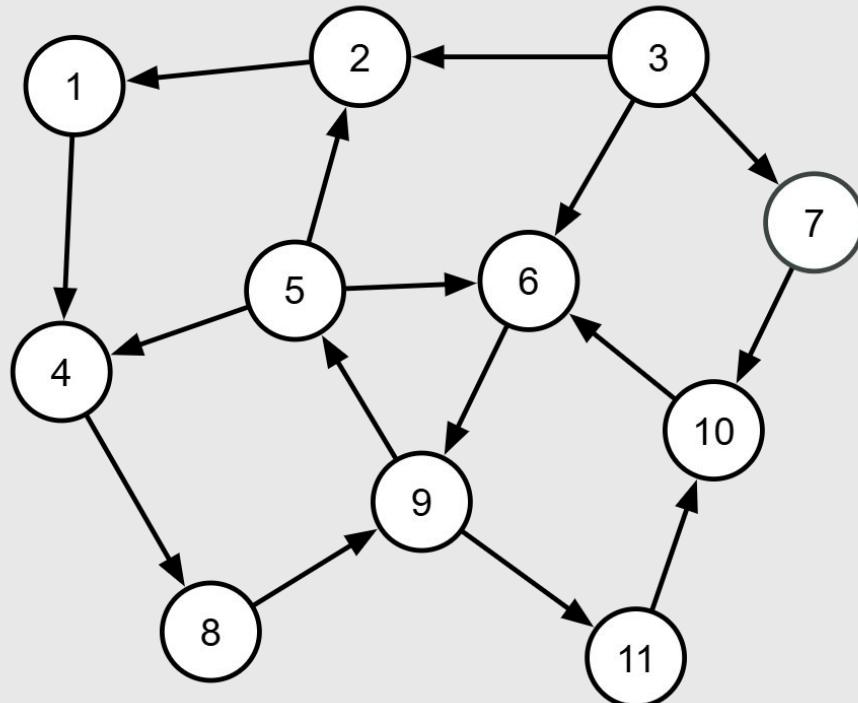
UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
The logo of the Federal University of Pernambuco, featuring a stylized lion holding three torches.

- **Caminho** é uma sequência de arestas (não necessariamente distintas) que levam de um nó origem até um nó destino.



Conceitos

- Um **ciclo** é um caminho fechado em um grafo (termina no mesmo nó inicial).



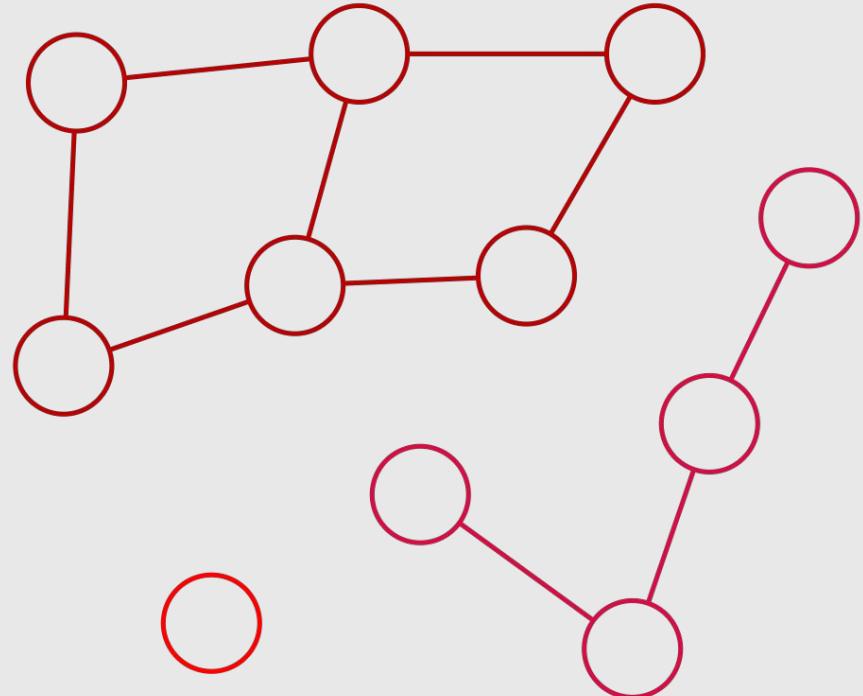
Alguns ciclos no grafo:

→ 6 - 9 - 11 - 10 - 6

→ 4 - 8 - 9 - 5 - 4

→ 1 - 4 - 8 - 9 - 5 - 2 - 1

Conceitos



*Grafo desconexo:
3 componentes*



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
VIRTUS IMPAVIDA

- Dois vértices estão em componentes diferentes se não existe um caminho entre eles.
- Um grafo é conexo se, para qualquer par de nós, existe pelo menos um caminho entre eles.

Conceitos

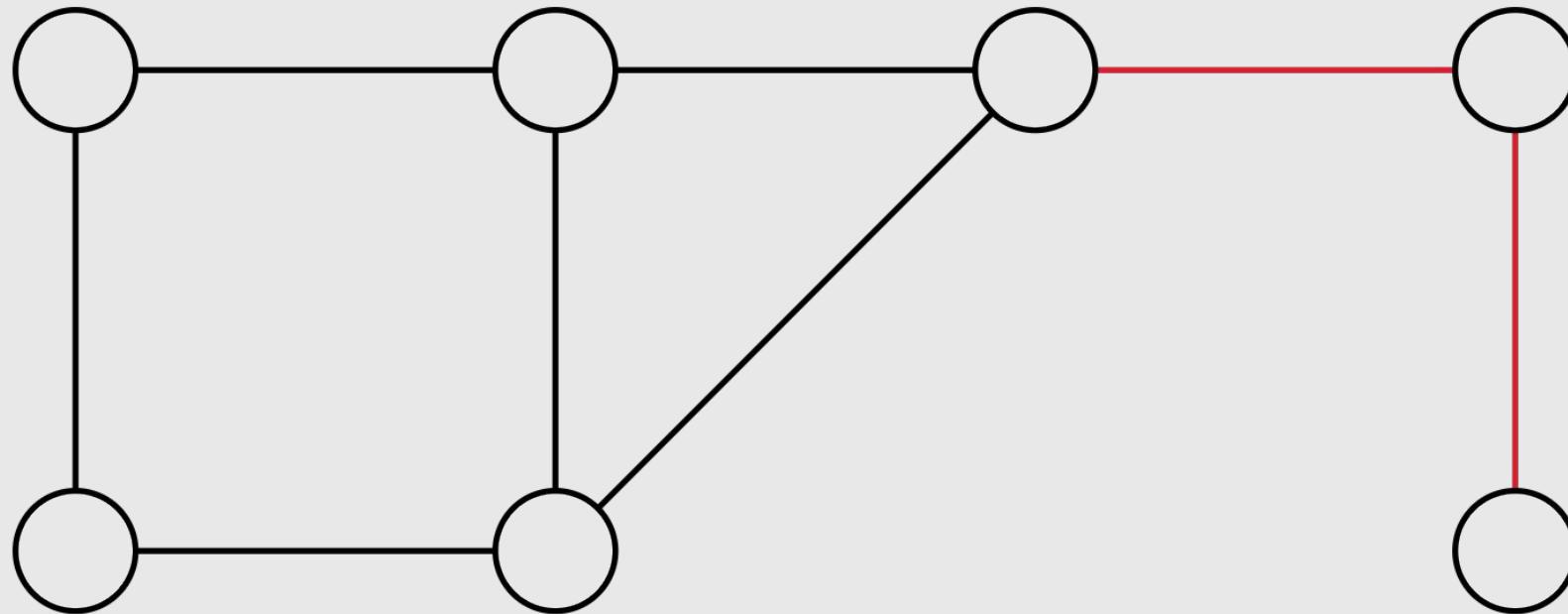


Maratona**CIn**

Centro de
Informática
UFPE

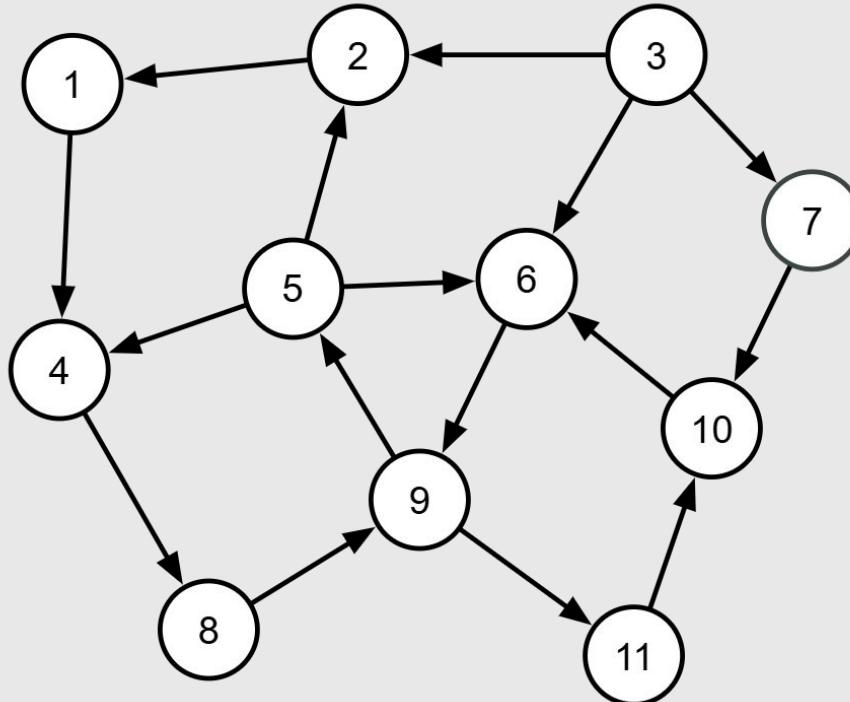
UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


- Pontes (ou Bridges) em um grafo são arestas que, se removidas, aumentam o número de componentes conexos desse grafo.



Conceitos

- Grau de entrada: quantas arestas chegam em um nó
- Grau de saída: quantas arestas saem de um nó



Exemplo:

Grau de **saída** de 4 = 1

Grau de **entrada** de 4 = 2

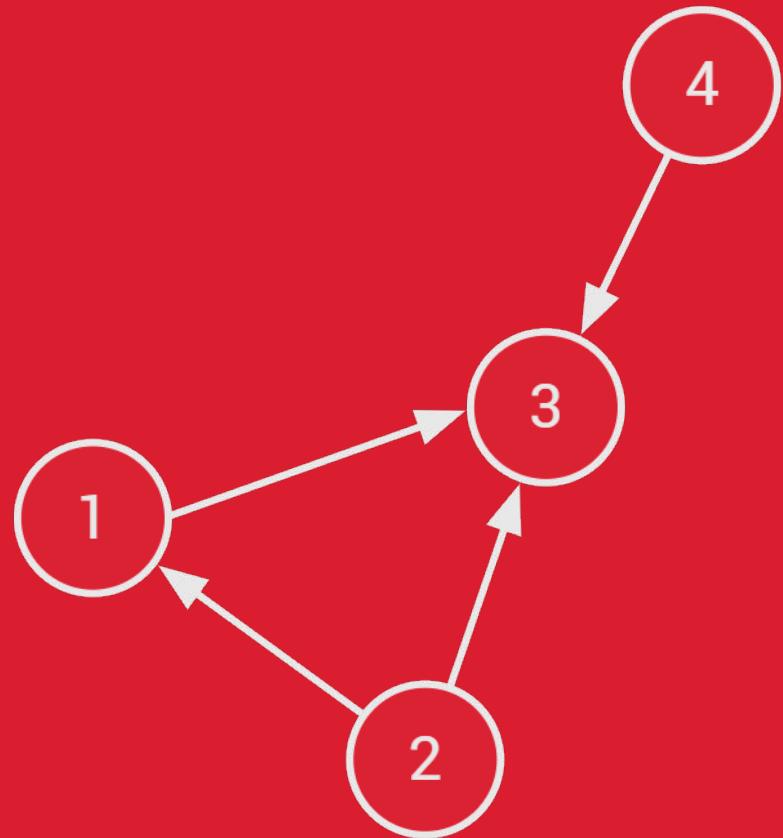


Maratona**CIn**



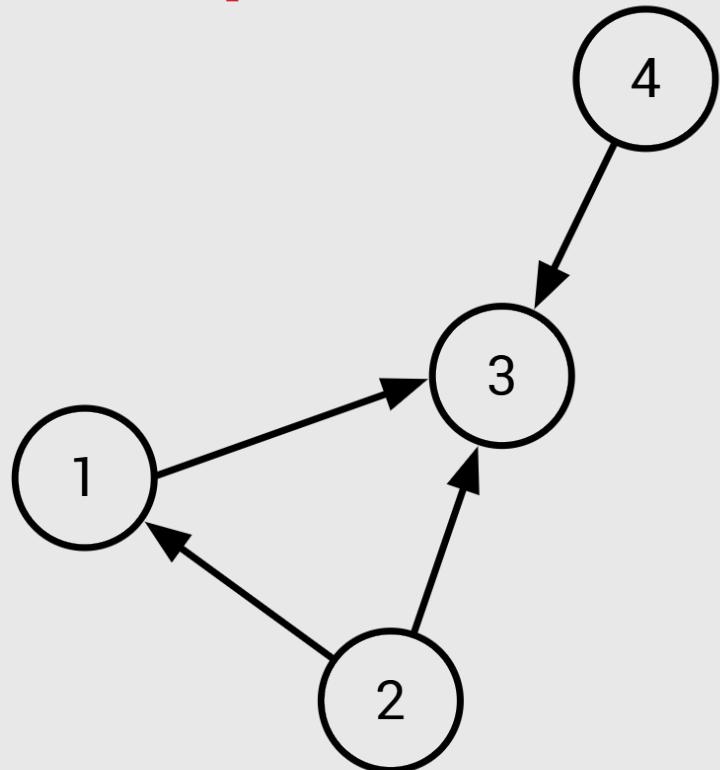
Representação de um Grafo

Matriz de Adjacências



	1	2	3	4
1	0	0	1	0
2	1	0	1	0
3	0	0	0	0
4	0	0	1	0

Matriz de Adjacências (sem peso)



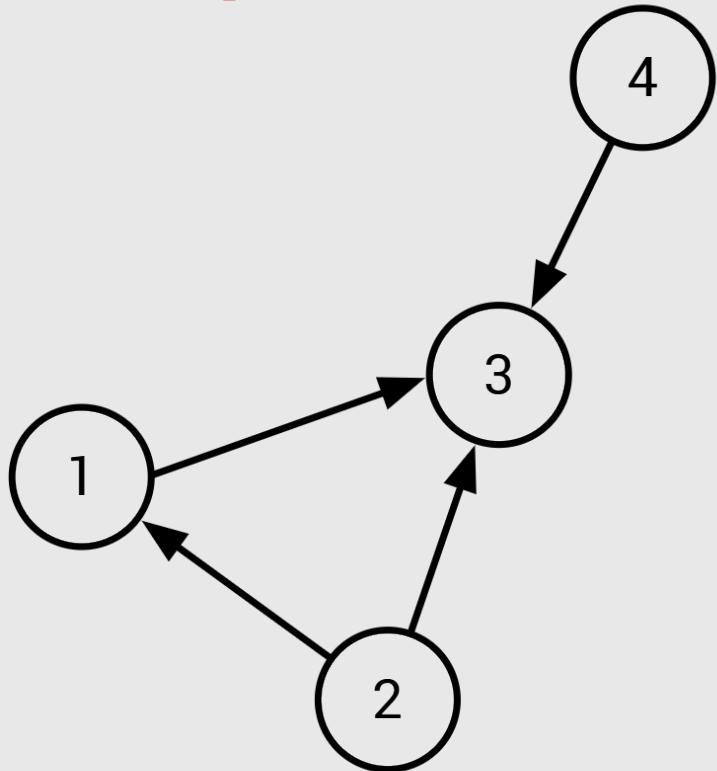
MatrixRepresentation.cpp

```
// N até mais ou menos 5e3
const int MAXSZ = 1e3 + 5;
bool g[MAXSZ][MAXSZ];

void addEdge(int u, int v){
    g[u][v] = 1;
    // g[v][u] = 1;
}

void removeEdge(int u, int v){
    g[u][v] = 0;
    // g[v][u] = 0;
}
```

Matriz de Adjacências (com peso)



MaratonaCIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

MatrixRepresentation.cpp

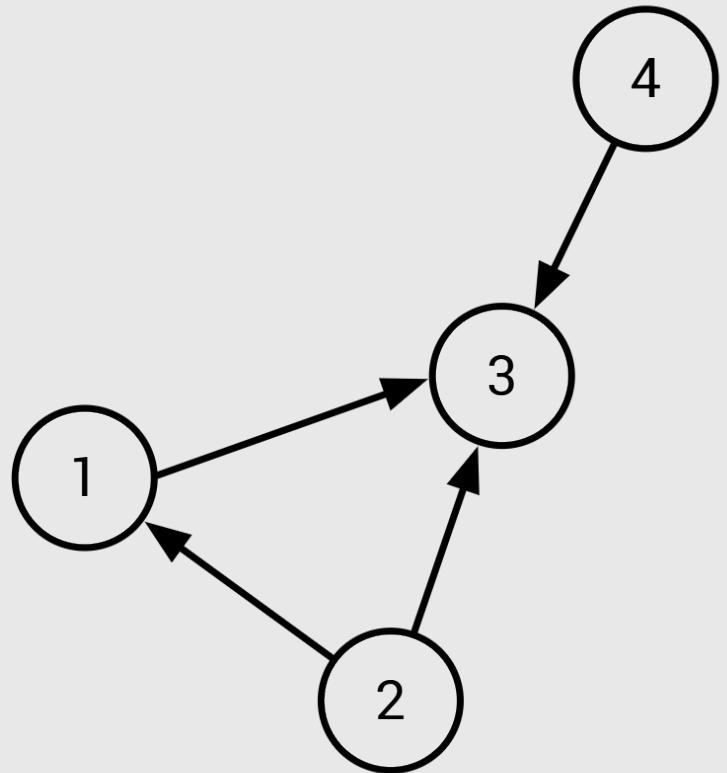
```
// N até mais ou menos 5e3
const int MAXSZ = 1e3 + 5;
const int INF = 2e9;
int g[MAXSZ][MAXSZ];

void init(){
    for(int i = 0; i<n; i++){
        for(int j = 0; j<n; j++){
            g[i][j] = INF;
        }
    }
}

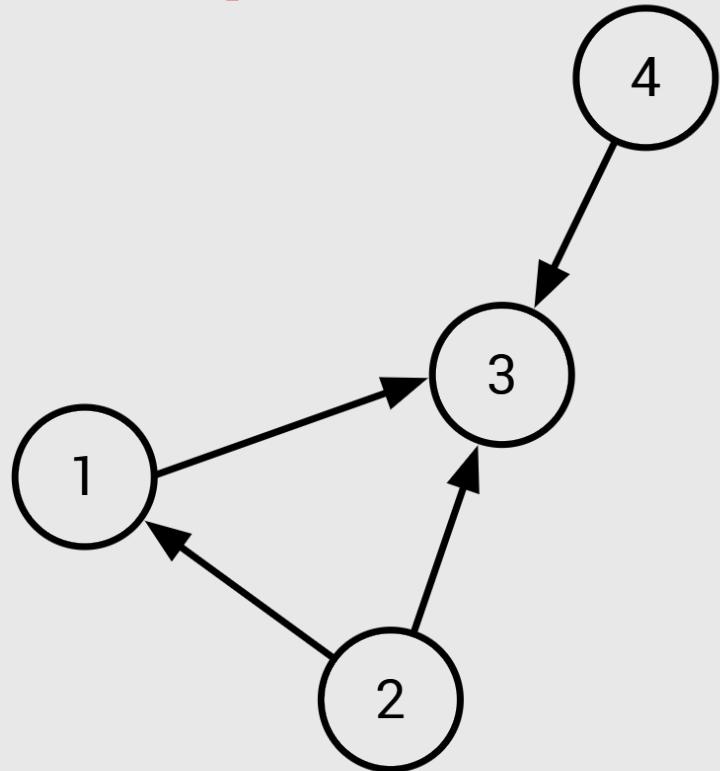
void addEdge(int u, int v, int w){
    g[u][v] = w;
    // g[v][u] = w;
}

void removeEdge(int u, int v, int w){
    g[u][v] = INF;
    // g[v][u] = INF;
}
```

Lista de Adjacências



Lista de Adjacências (sem peso)



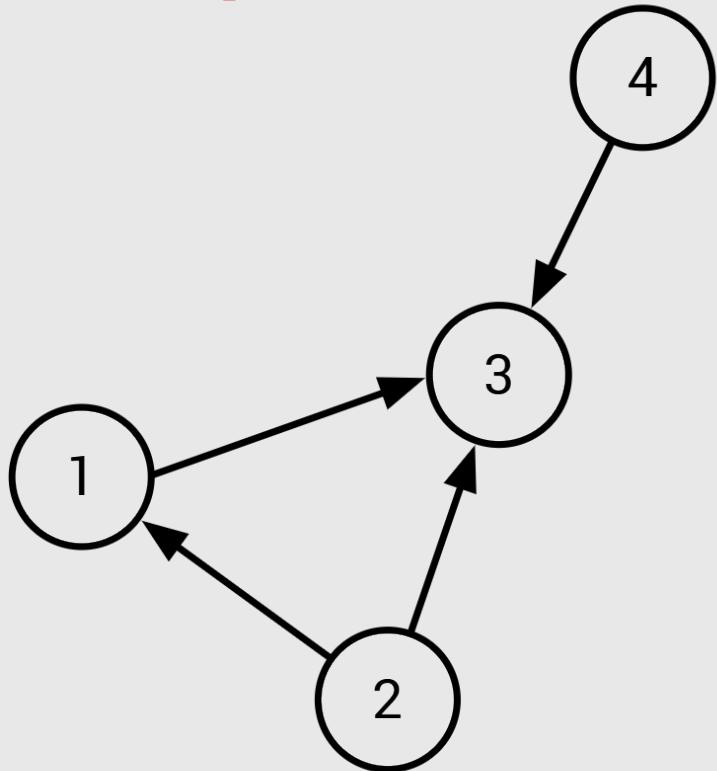
MatrixRepresentation.cpp

```
const int MAXSZ = 1e3 + 5;
int n, m;

vector<int>adj[MAXSZ];

void addEdge(int u, int v){
    adj[u].push_back(v);
    // adj[v].push_back(u);
}
```

Lista de Adjacências (com peso)



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
VIRTUS IMPAVIDA

C++ MatrixRepresentation.cpp

```
#define pii pair<int, int>
const int MAXSZ = 1e3 + 5;
int n, m;

vector<pii>adj[MAXSZ];

void addEdge(int u, int v, int w){
    adj[u].push_back({v, w});
    // adj[v].push_back({u, w});
}
```



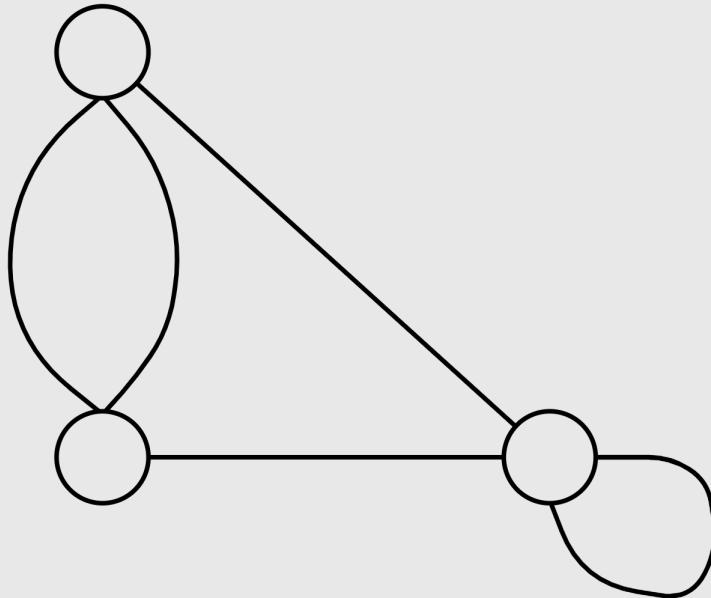
Maratona**CIn**



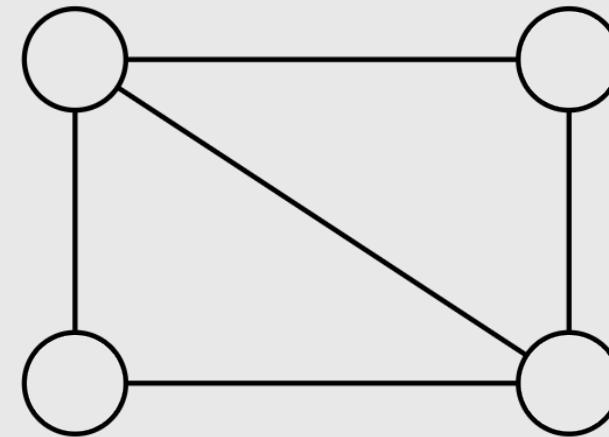
Tipos de Grafos

Tipos de Grafos

- **Grafo simples:** Sem laços e arestas múltiplas



Não é um grafo simples



Grafo simples

Tipos de Grafos

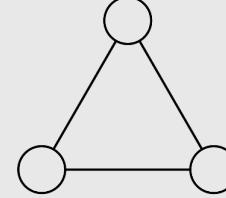


Maratona**CIn**

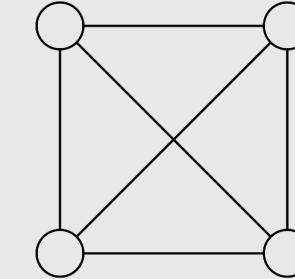
Centro de
Informática
UFPE



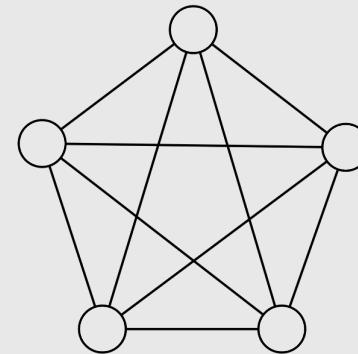
$K2$



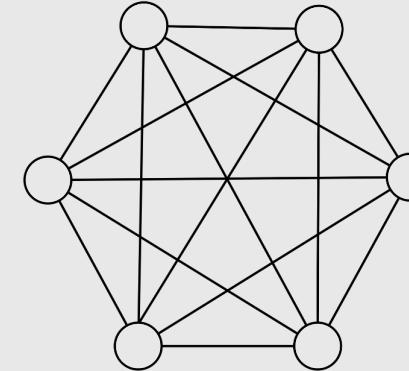
$K3$



$K4$



$K5$



$K6$

Grafo completo: Existe uma aresta entre todo par de vértices

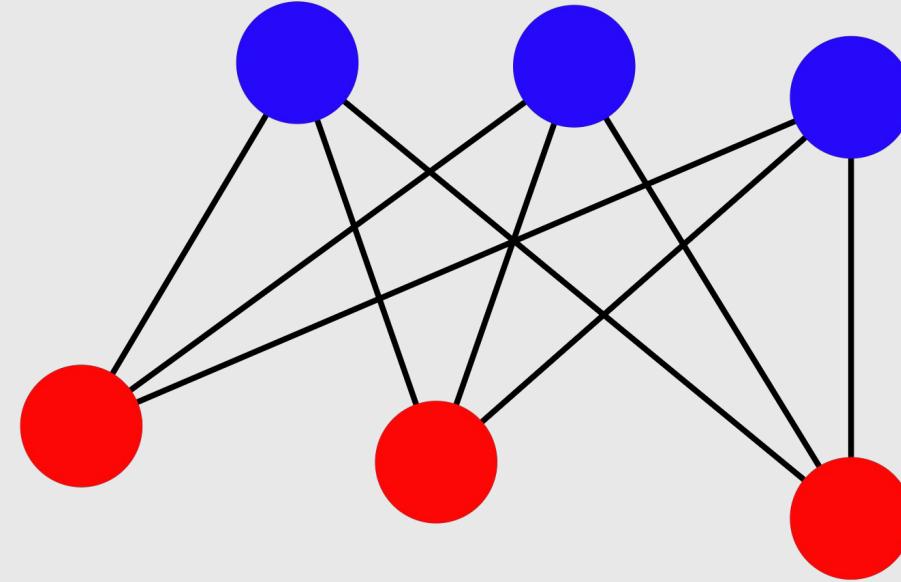
Tipos de Grafos



MaratonaCIn

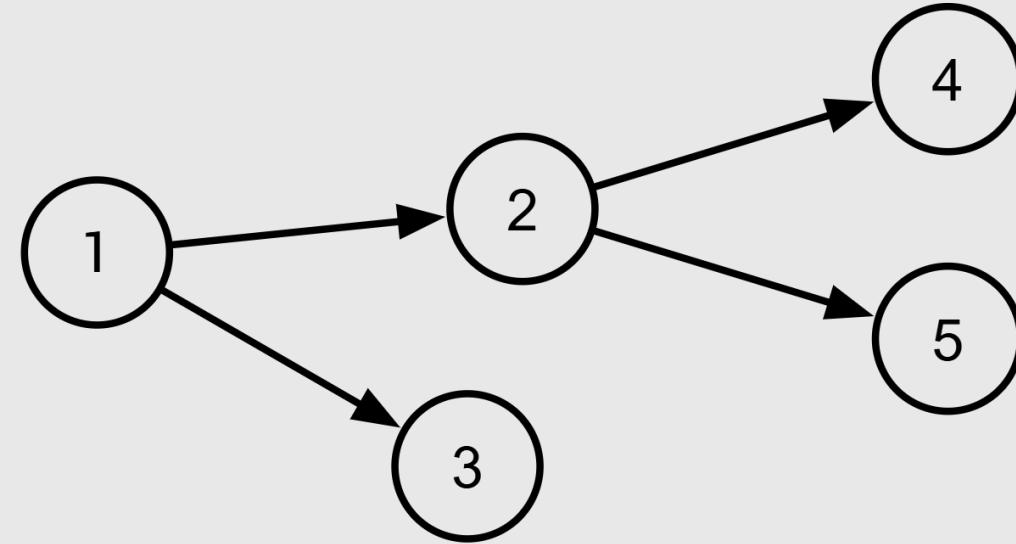
Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Grafo bipartido: Existem dois subconjuntos complementares de vértices de modo que não existe aresta de um vértice de um grupo para outro do mesmo subconjunto

Tipos de Grafos



DAG (Directed acyclic graph): Chamamos nós de grau de entrada 0 de source e os de grau de saída 0 de sink

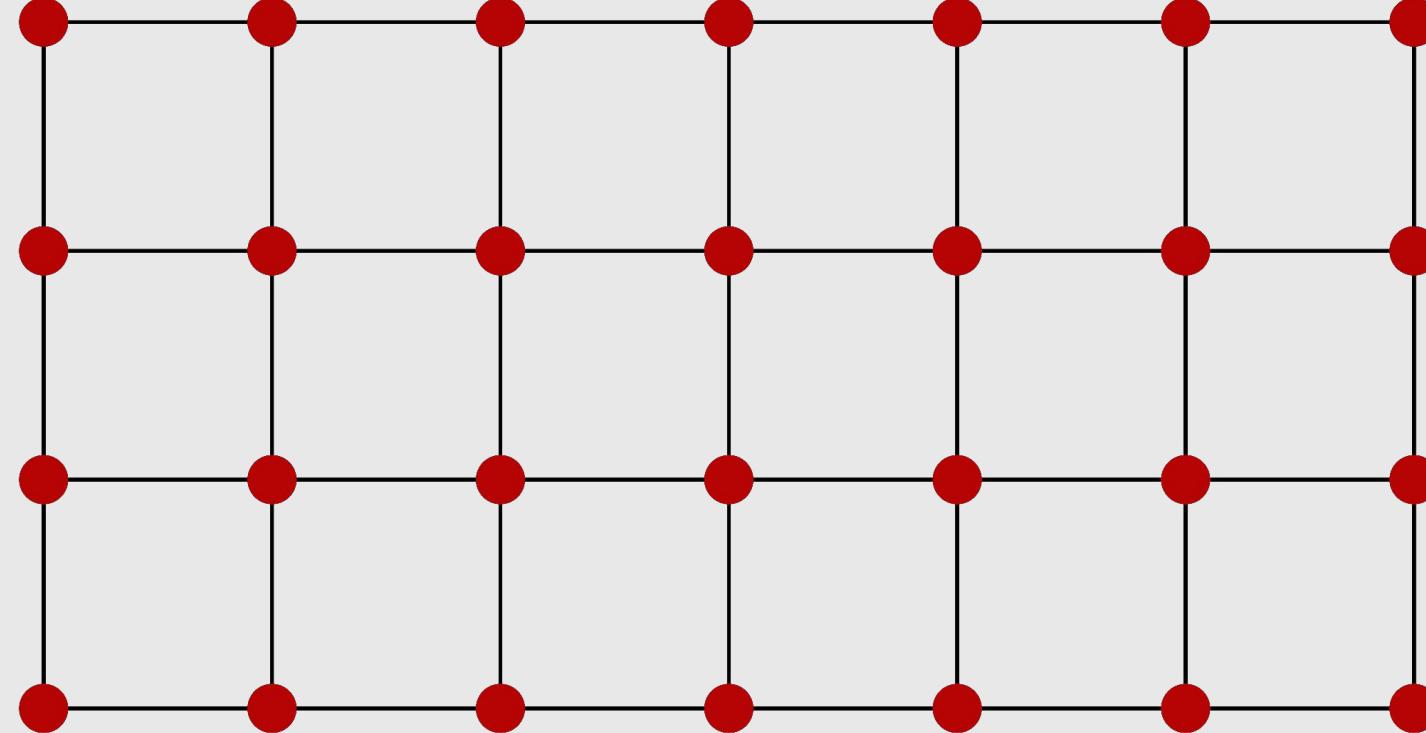
Tipos de Grafos



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Grid: Areias estão implícitas:
(4-adjacência/8-adjacência)

Percorrer a Grid



MaratonaCIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

grid.cpp

- □ ×

```
const int dx[] = {1, 0, 0, -1};
const int dy[] = {0, 1, -1, 0};

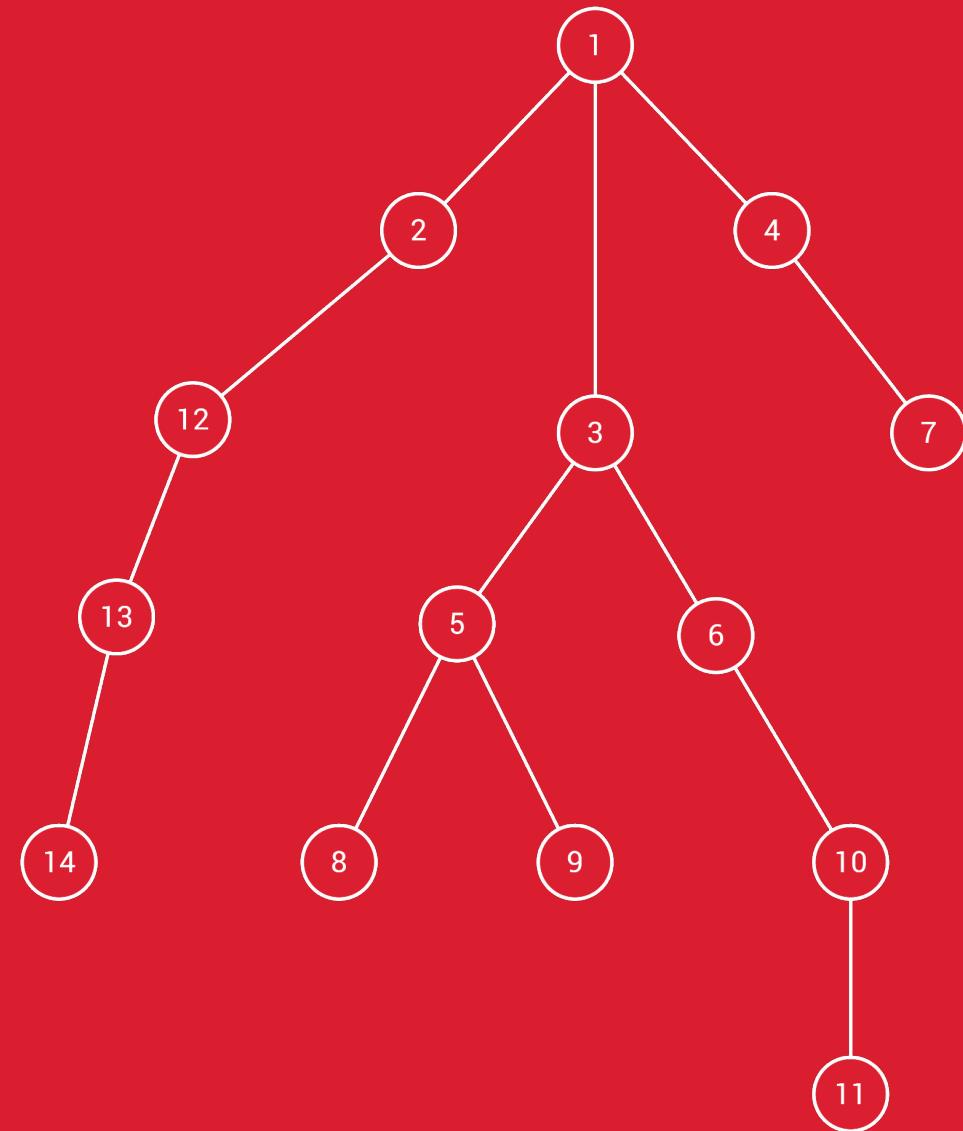
bool valid(int x, int y){
    return x≥0 && x < n && y ≥ 0 && y < m && grid[x][y] ≠ '#';
}

void rec(int x, int y){
    grid[x][y] = '#';
    for(int d = 0; d<4; d++){
        int ax = x + dx[d], ay = y + dy[d];
        if(!valid(ax, ay)) continue;
        rec(ax, ay);
    }
}
```

Árvores

Chamamos um grafo conexo e sem ciclos de árvore. Essa definição faz com que a árvore possua algumas propriedades interessantes:

- Uma árvore sempre tem exatamente $N-1$ arestas.
- Existe um único caminho entre qualquer par de nós.
- Se qualquer aresta for removida o grafo se torna desconexo (todas arestas são pontes).
- Se qualquer aresta for adicionada, forma-se um ciclo.



Árvores



MaratonaCIn

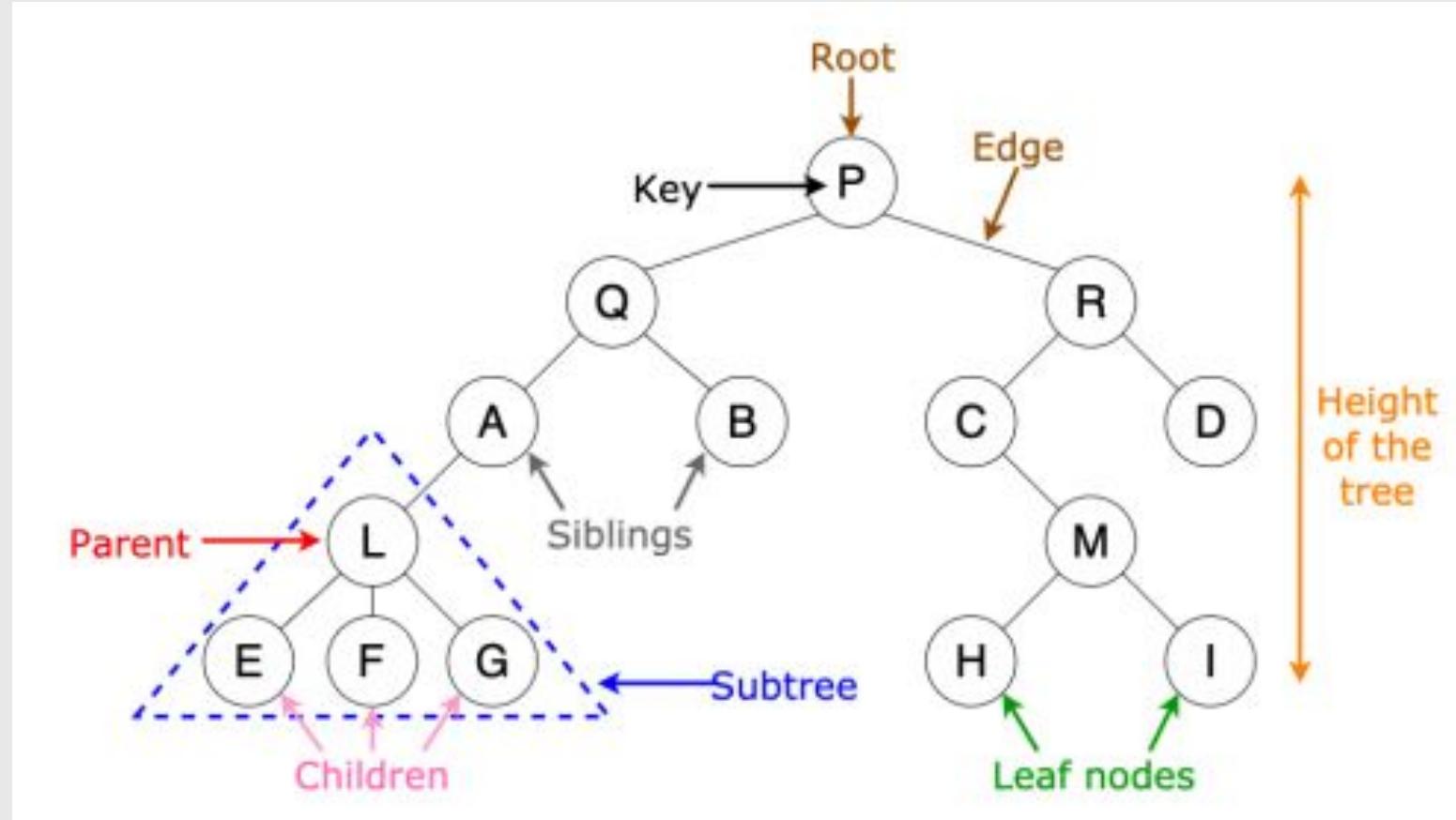
Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
The logo of the Federal University of Pernambuco (UFPE), which includes a stylized lion and three torches.

Além disso podemos falar em Árvore enraizada. Podemos definir um nó como a “raiz” da árvore e a partir disso explorar mais algumas propriedades e conceitos que surgem. Dado um vértice V podemos definir:

- Pai: o vértice diretamente acima dele na árvore.
- Filhos: são os vértices diretamente abaixo dele na árvore.
- Ancestrais: todos os vértices acima dele na árvore, até a raiz.
- Subárvore: todos os “descendentes” de um nó.
- Folha: vértice que não possui filhos

Árvores





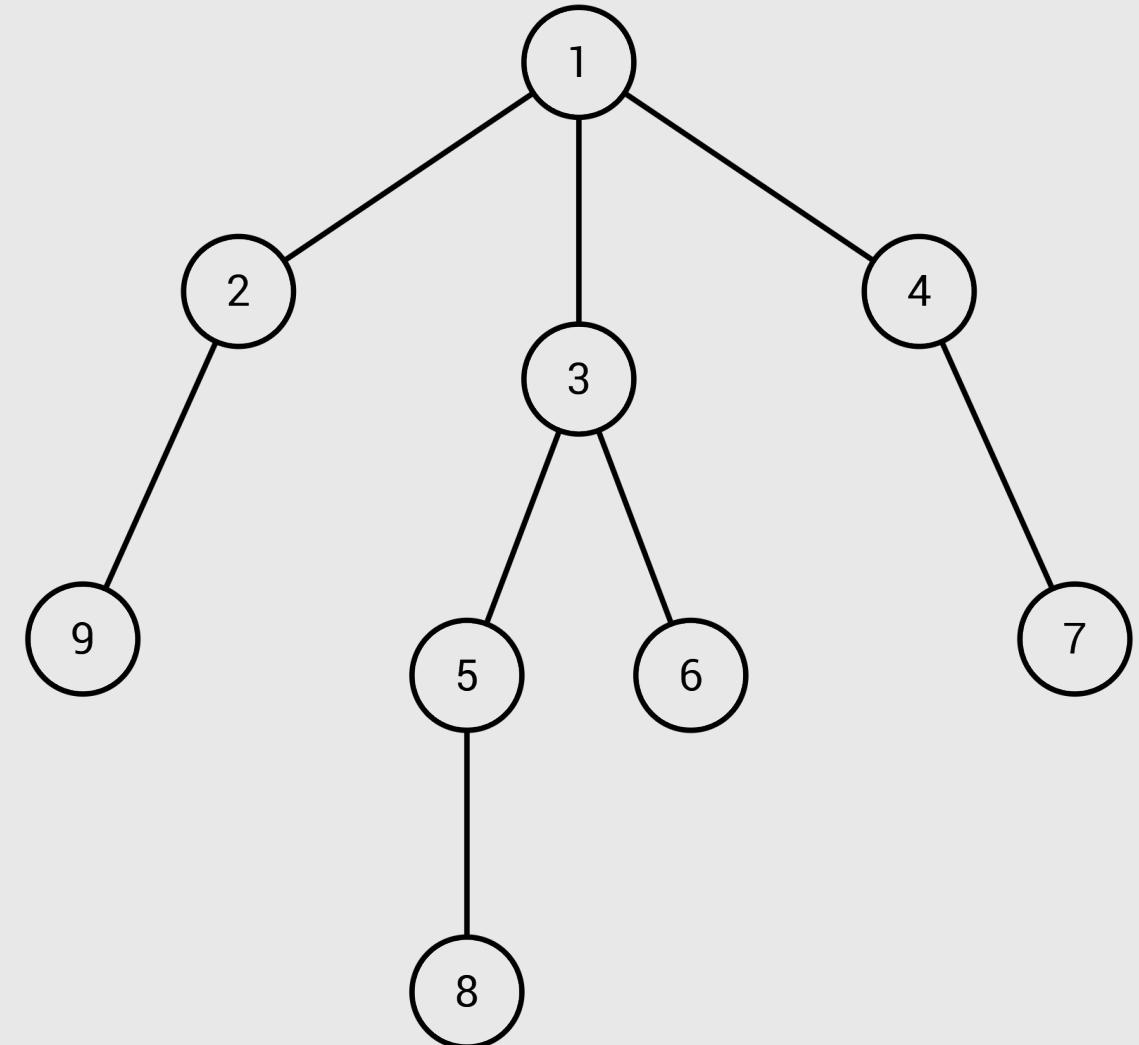
DFS

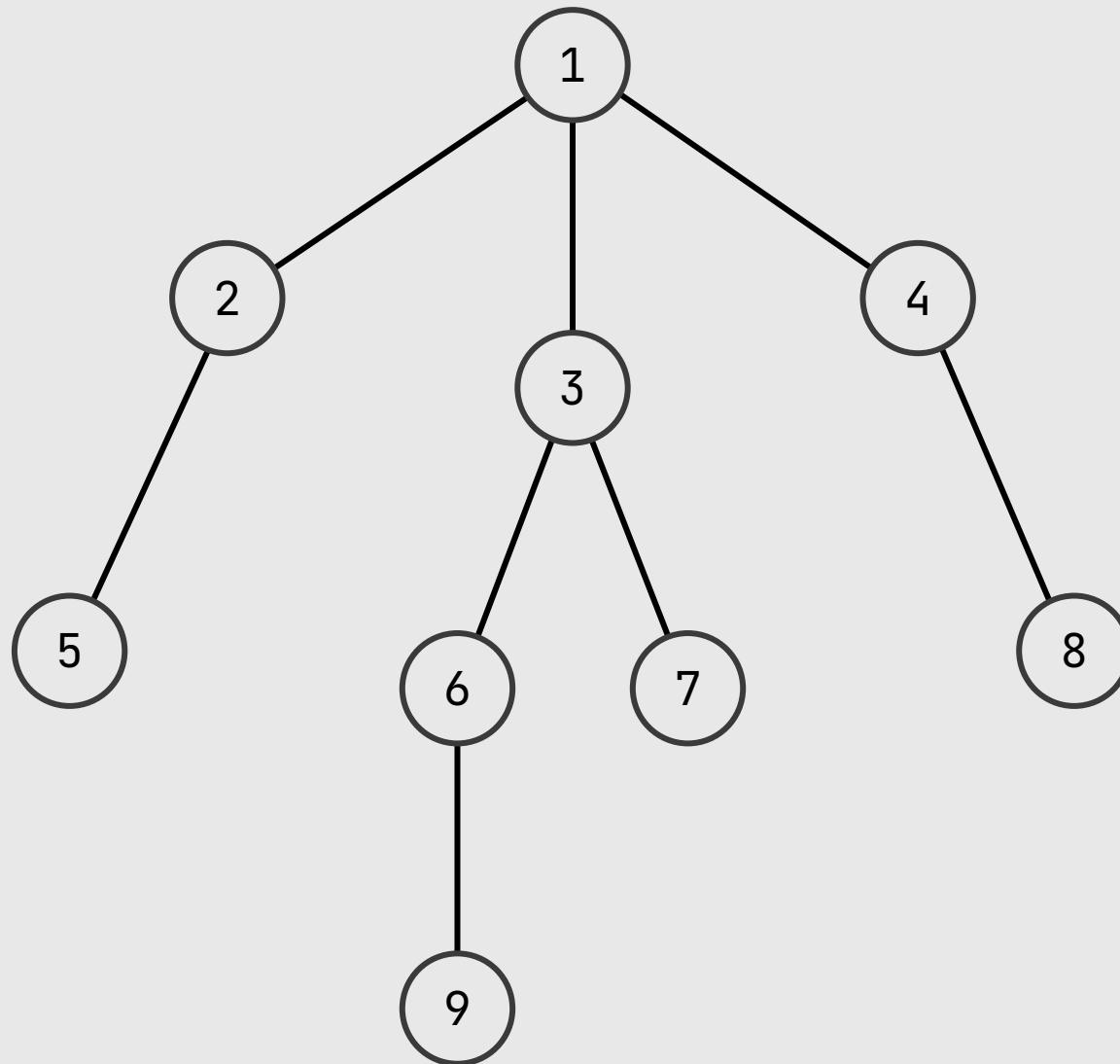
DEPTH-FIRST SEARCH / BUSCA EM PROFUNDIDADE

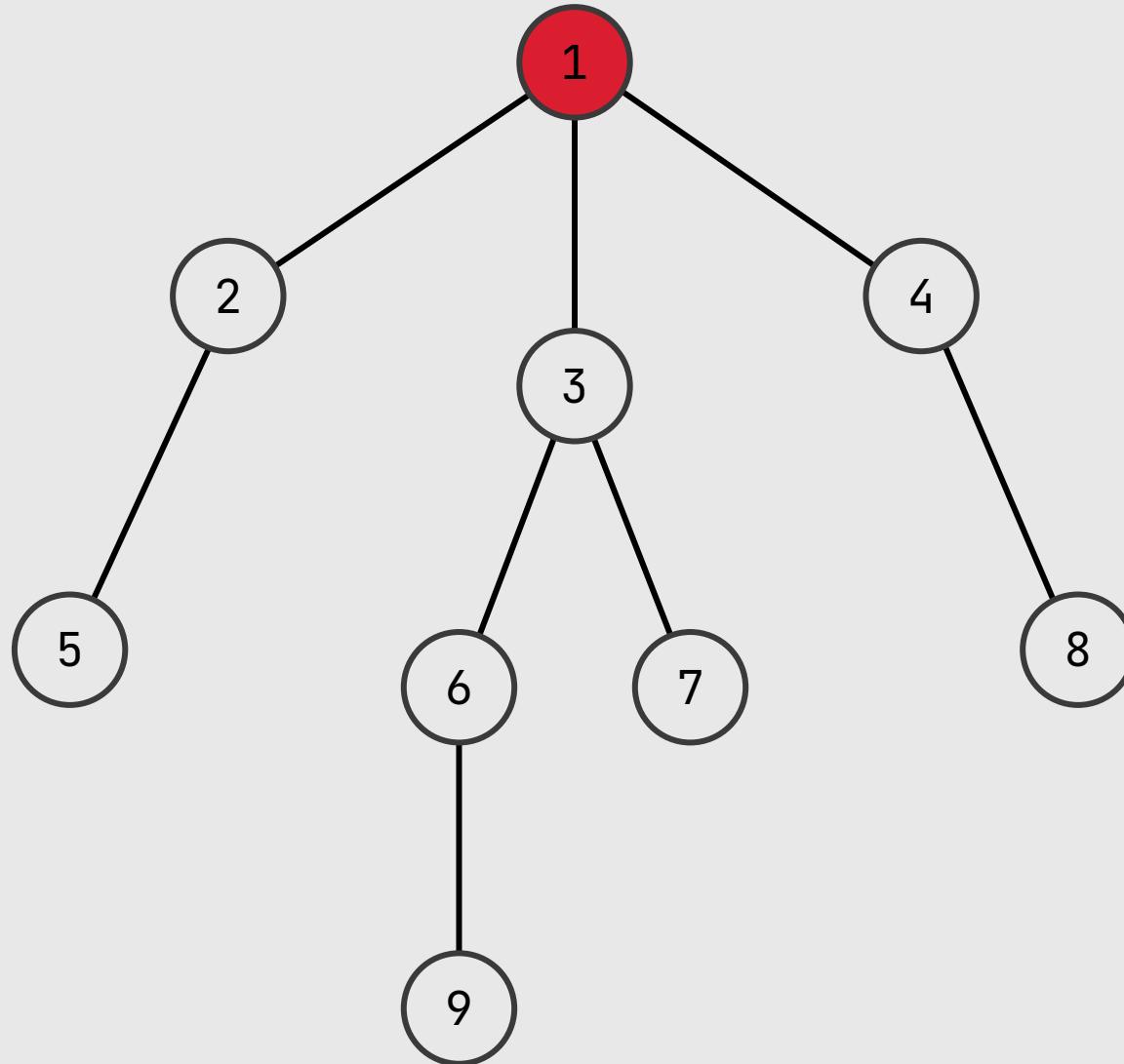
DFS

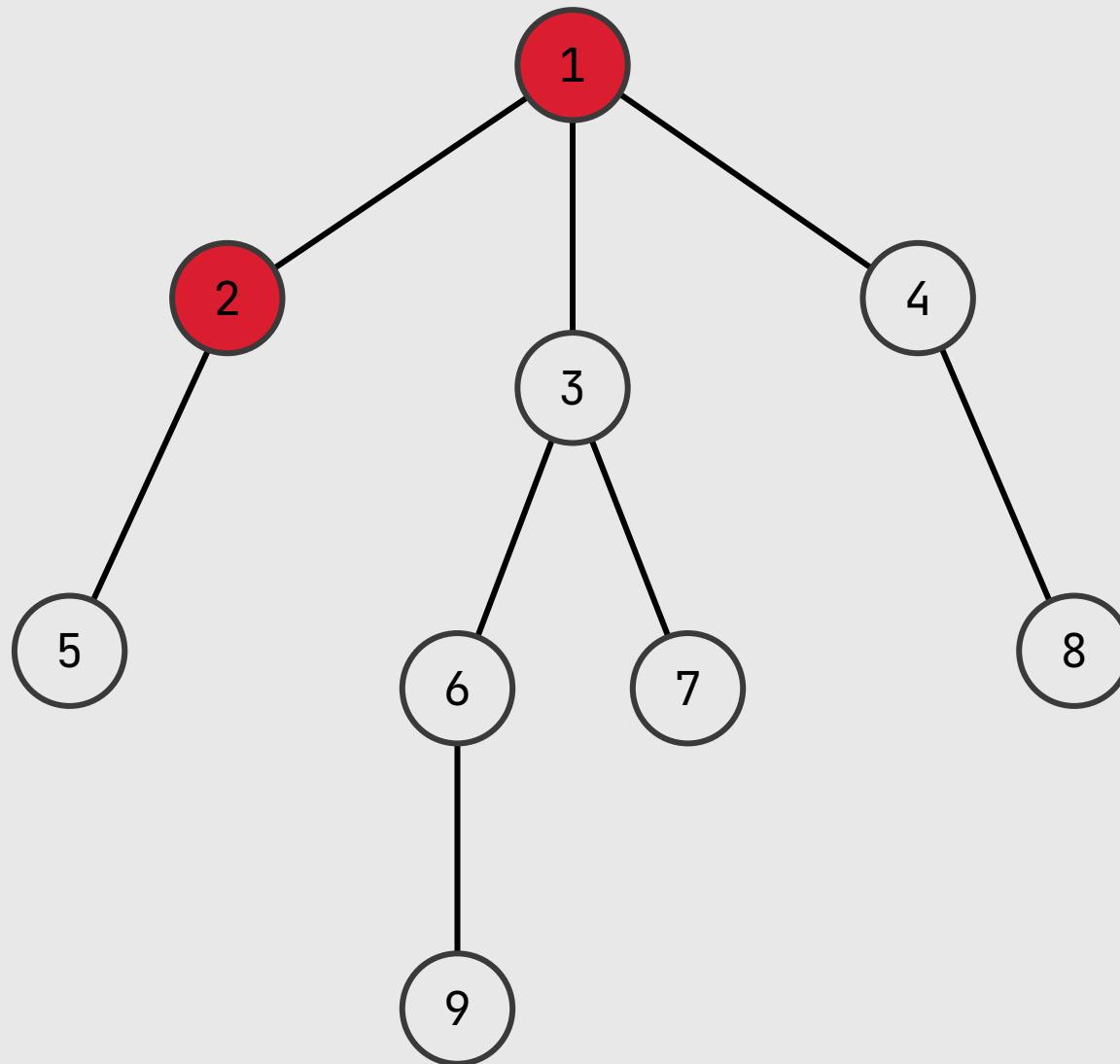
Na DFS fazemos a busca por meio de uma PILHA, ou de recursão.

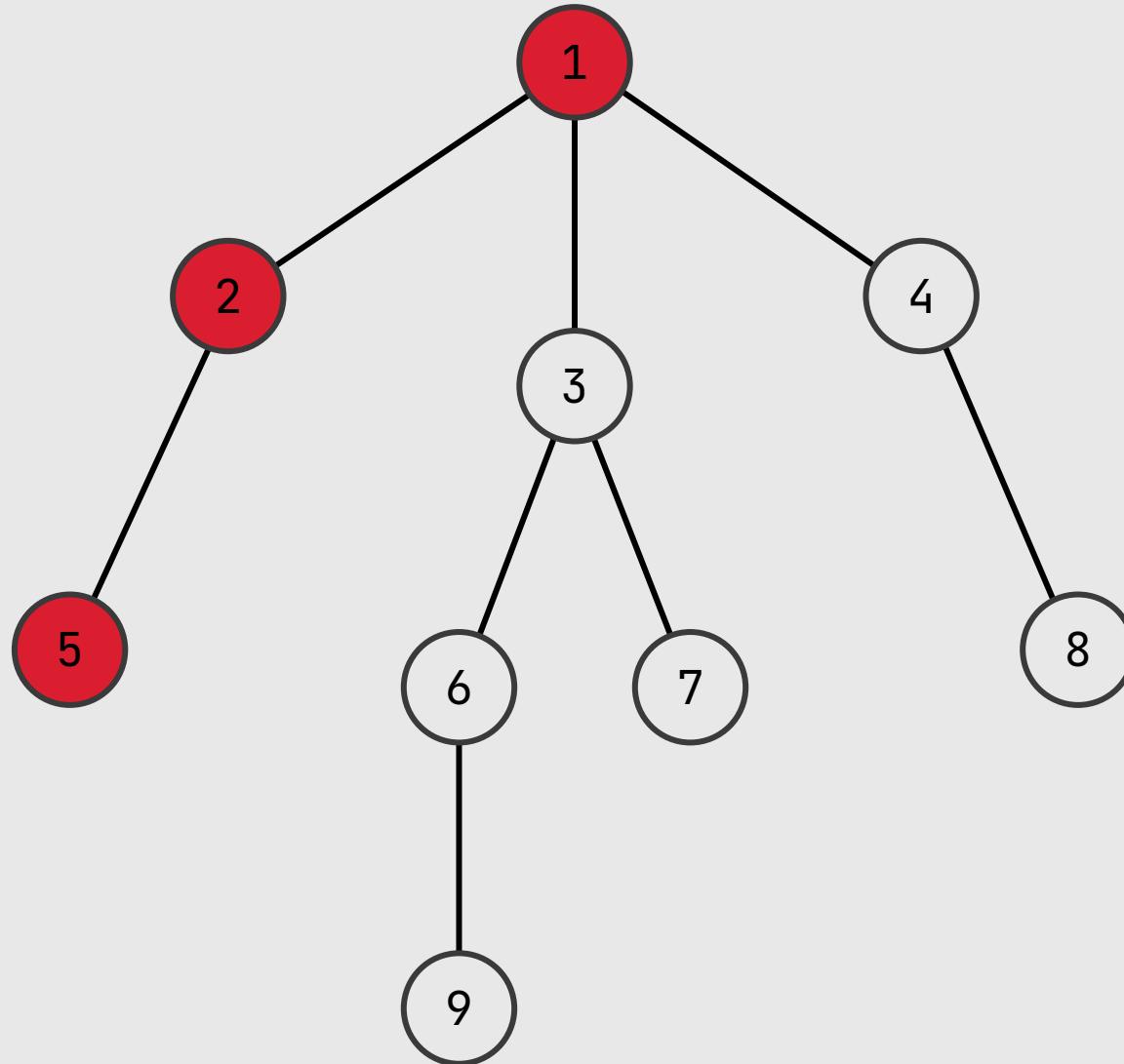
Começamos de um vértice inicial e então visitamos um vizinho dele que ainda não foi visitado. A partir desse vértice vamos para outro que ainda não foi visitado e assim por diante. Isto é o equivalente a adicionar os vizinhos em uma pilha (a recursão também forma uma pilha).

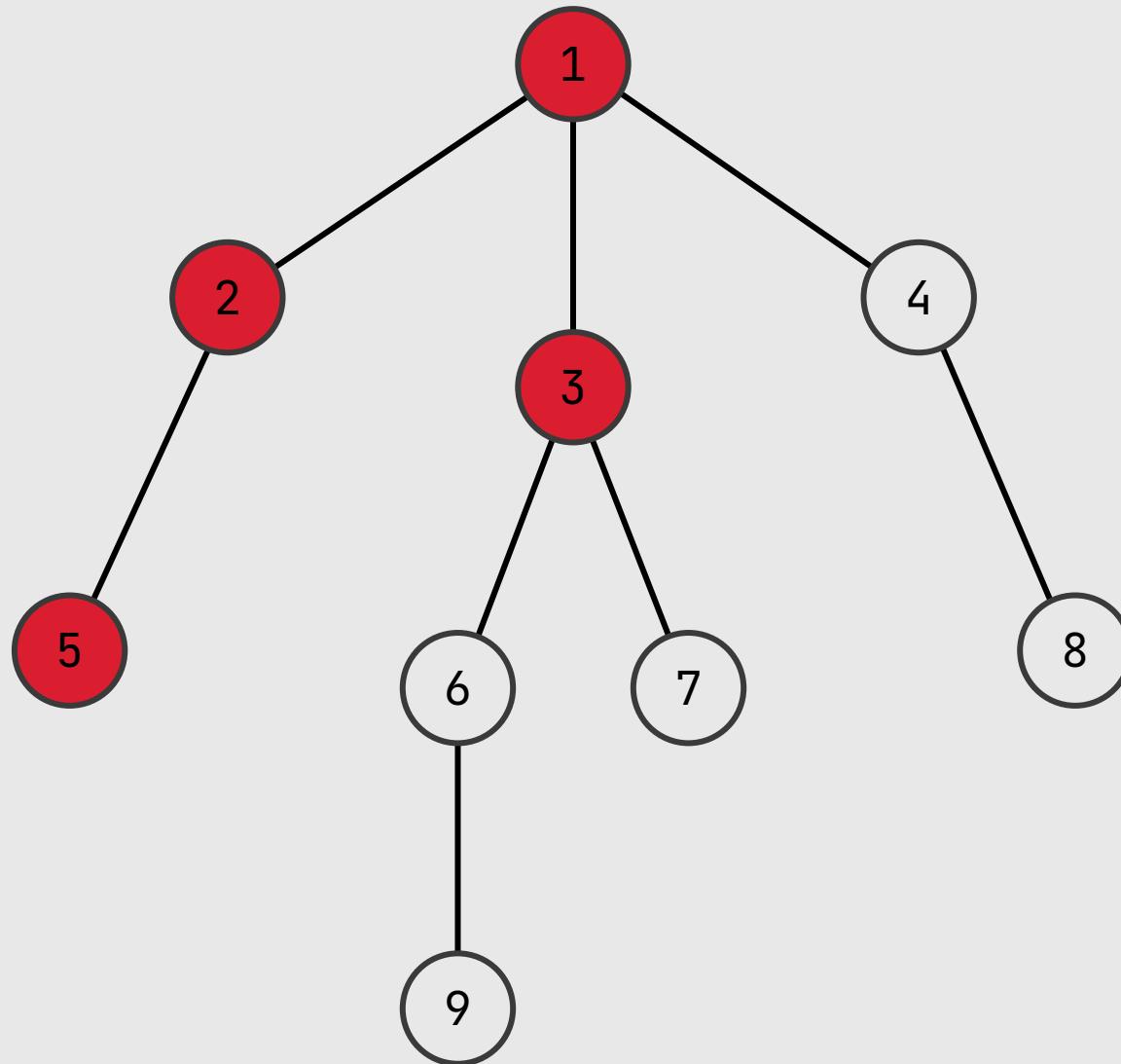


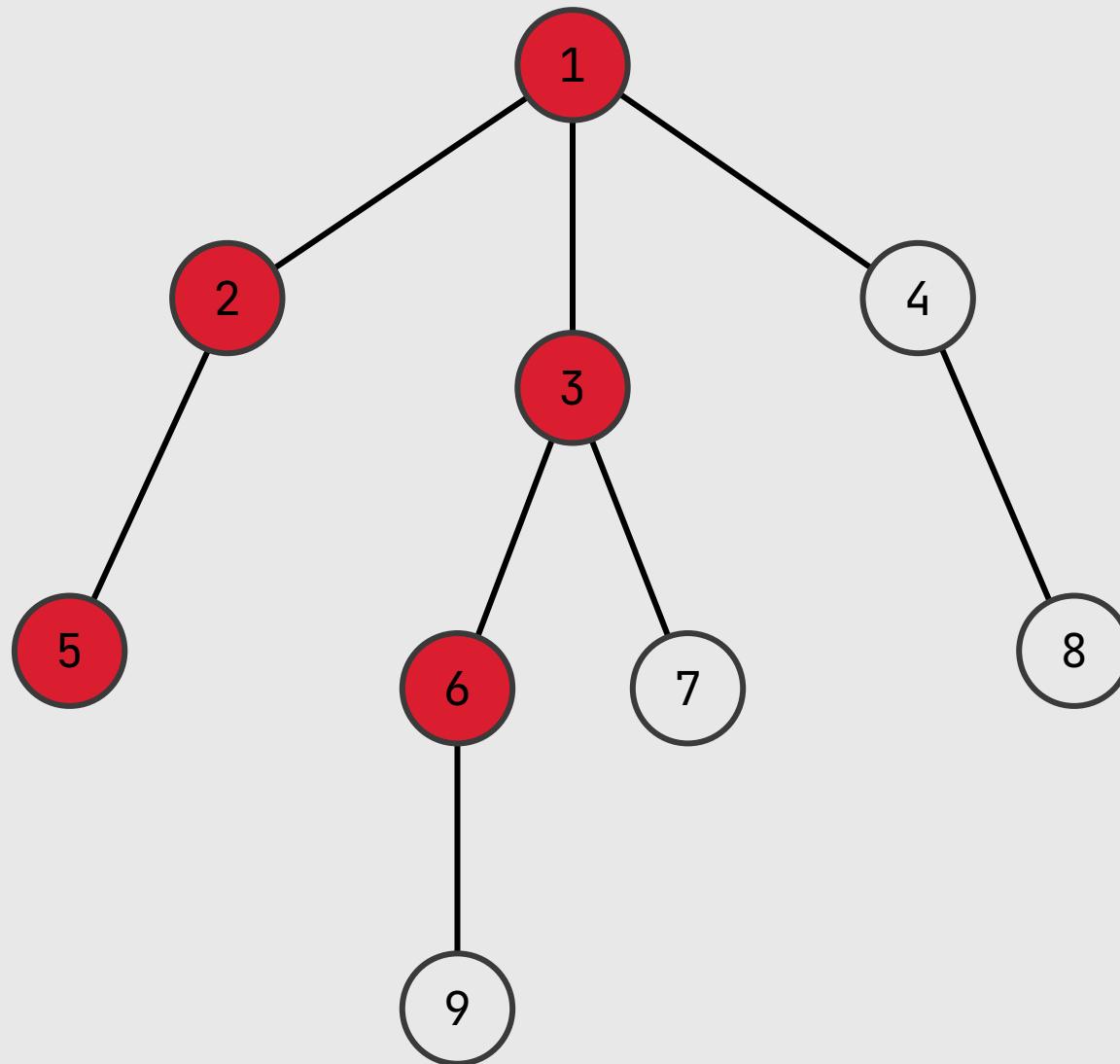


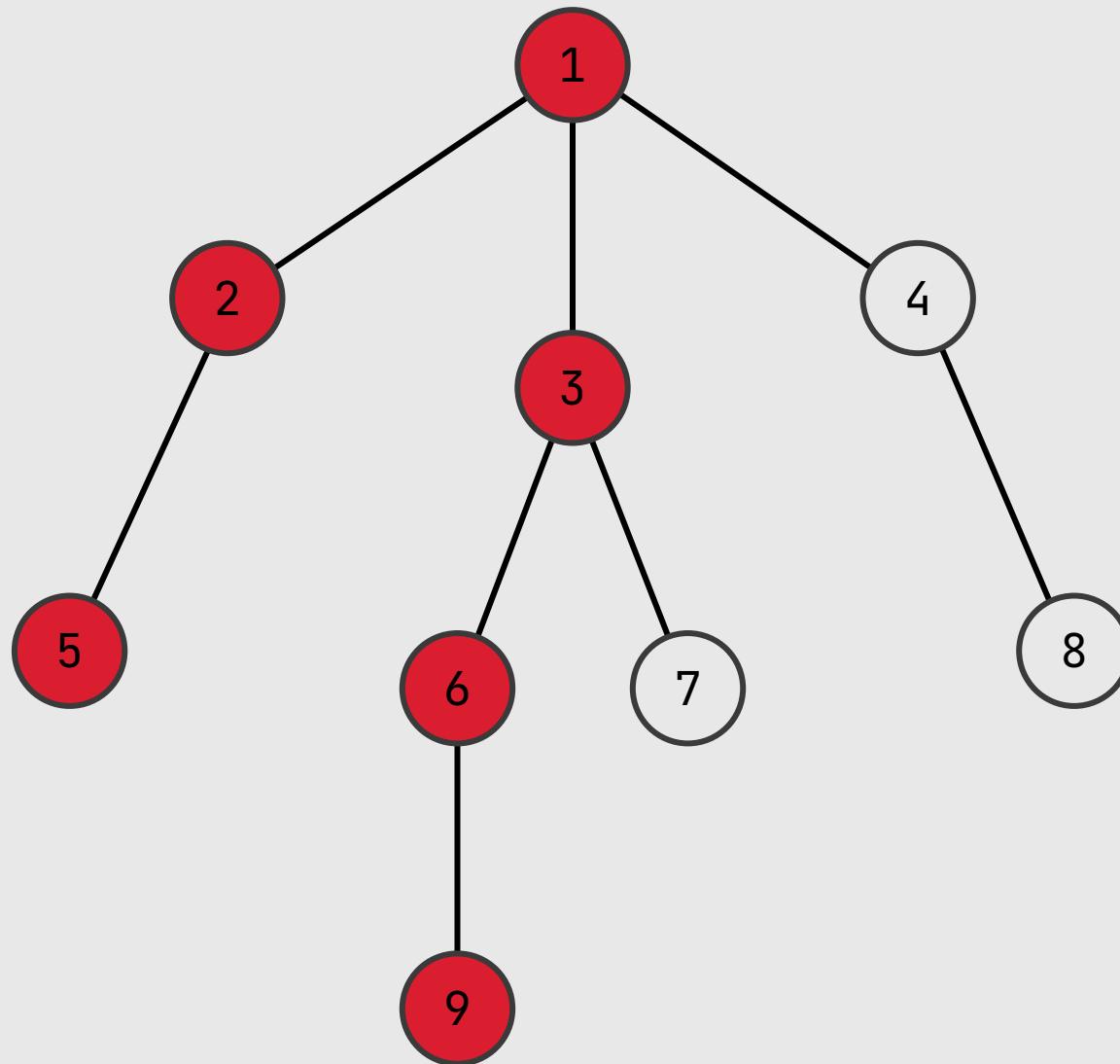


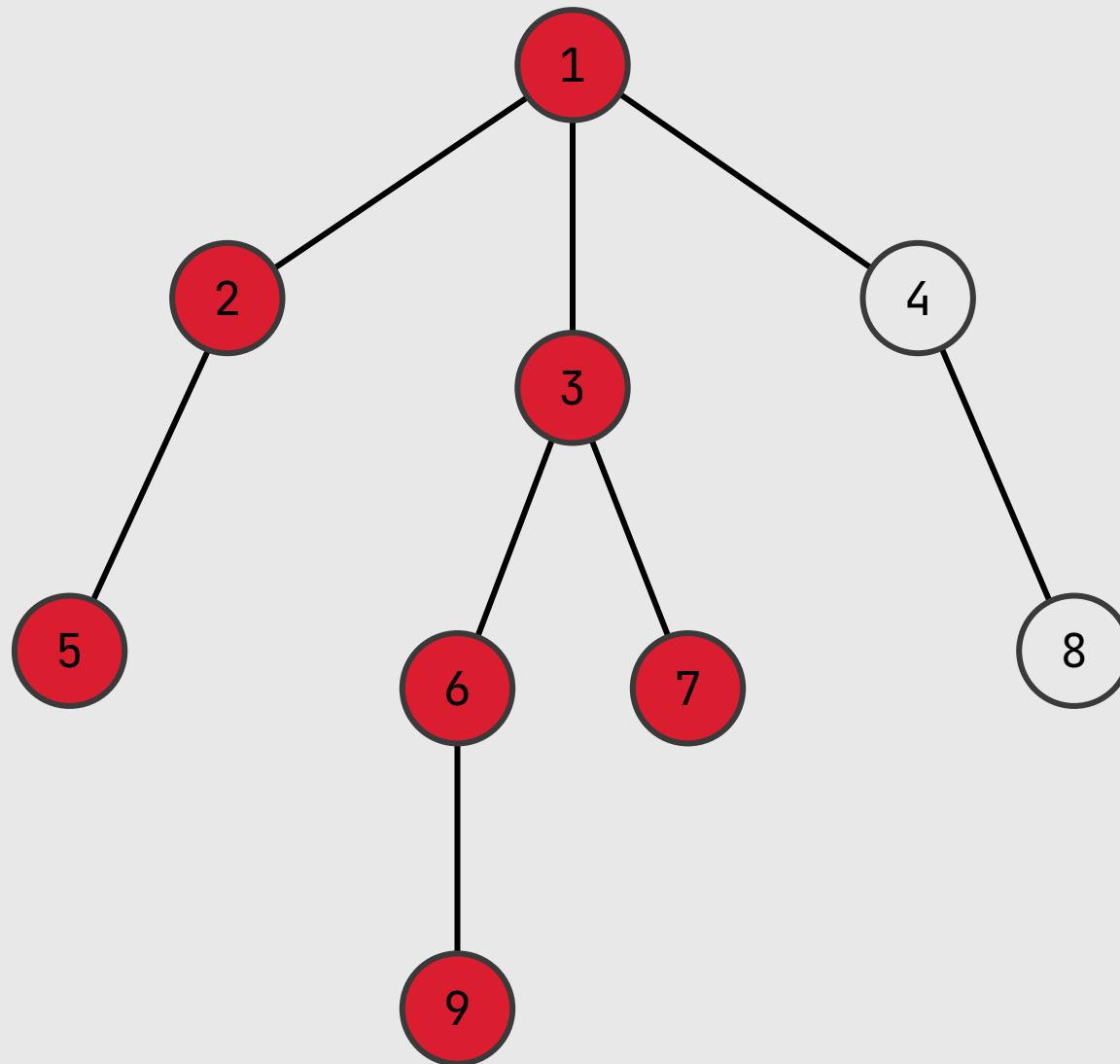


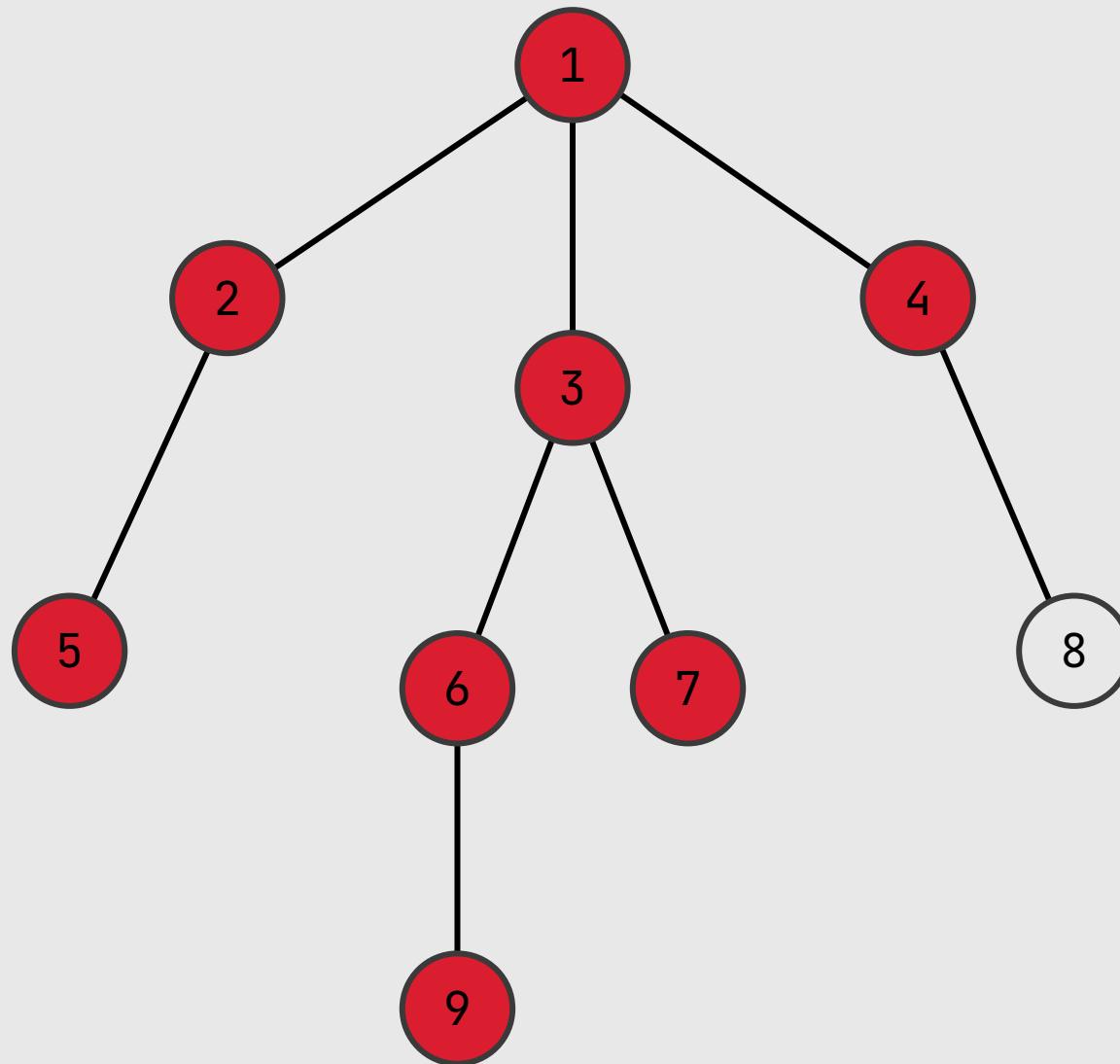


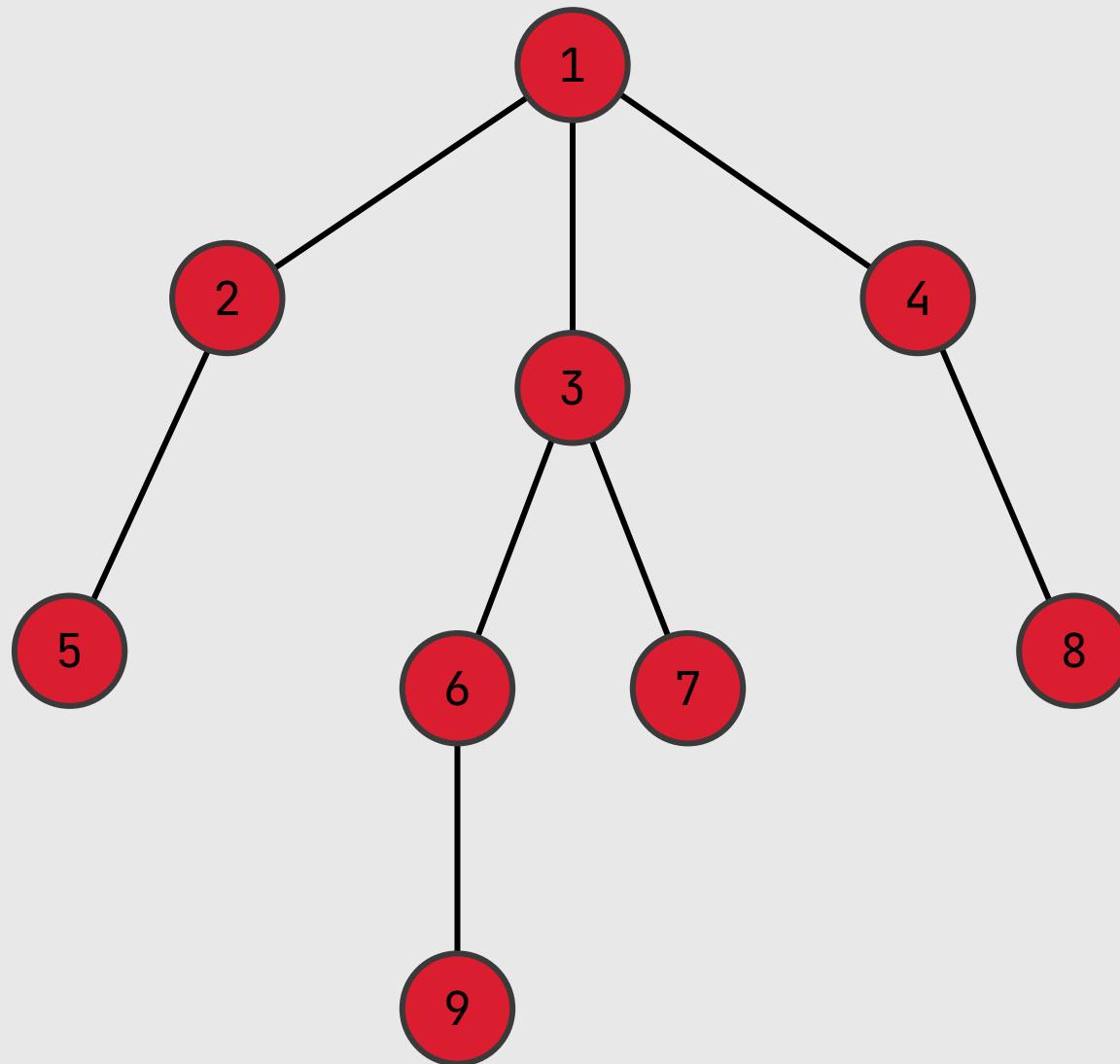












DFS

Na DFS fazemos a busca por meio de uma PILHA, ou de recursão.

Começamos de um vértice inicial e então visitamos um vizinho dele que ainda não foi visitado. A partir desse vértice vamos para outro que ainda não foi visitado e assim por diante. Isto é o equivalente a adicionar os vizinhos em uma pilha (a recursão também forma uma pilha).

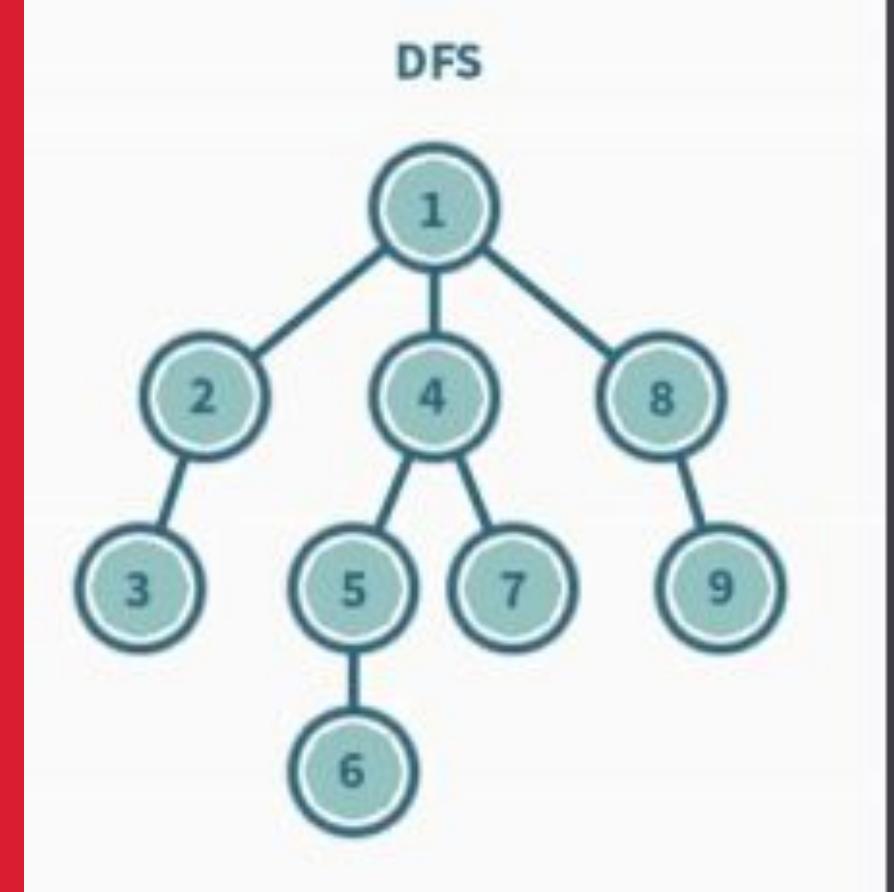


Maratona

Cln

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
VIRTUS IMPAVIDA



Código DFS



Maratona

CIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


C++ dfs.cpp

- □ ×

```
vector<int>adj[MAXN];
bool vis[MAXN]; // false para todos

void dfs(int u){
    vis[u] = true;
    for(int v: adj[u]){
        if(!vis[v]) dfs(v);
    }
}
```

Com isso, a complexidade do pior caso do algoritmo DFS é $O(V + E)$.

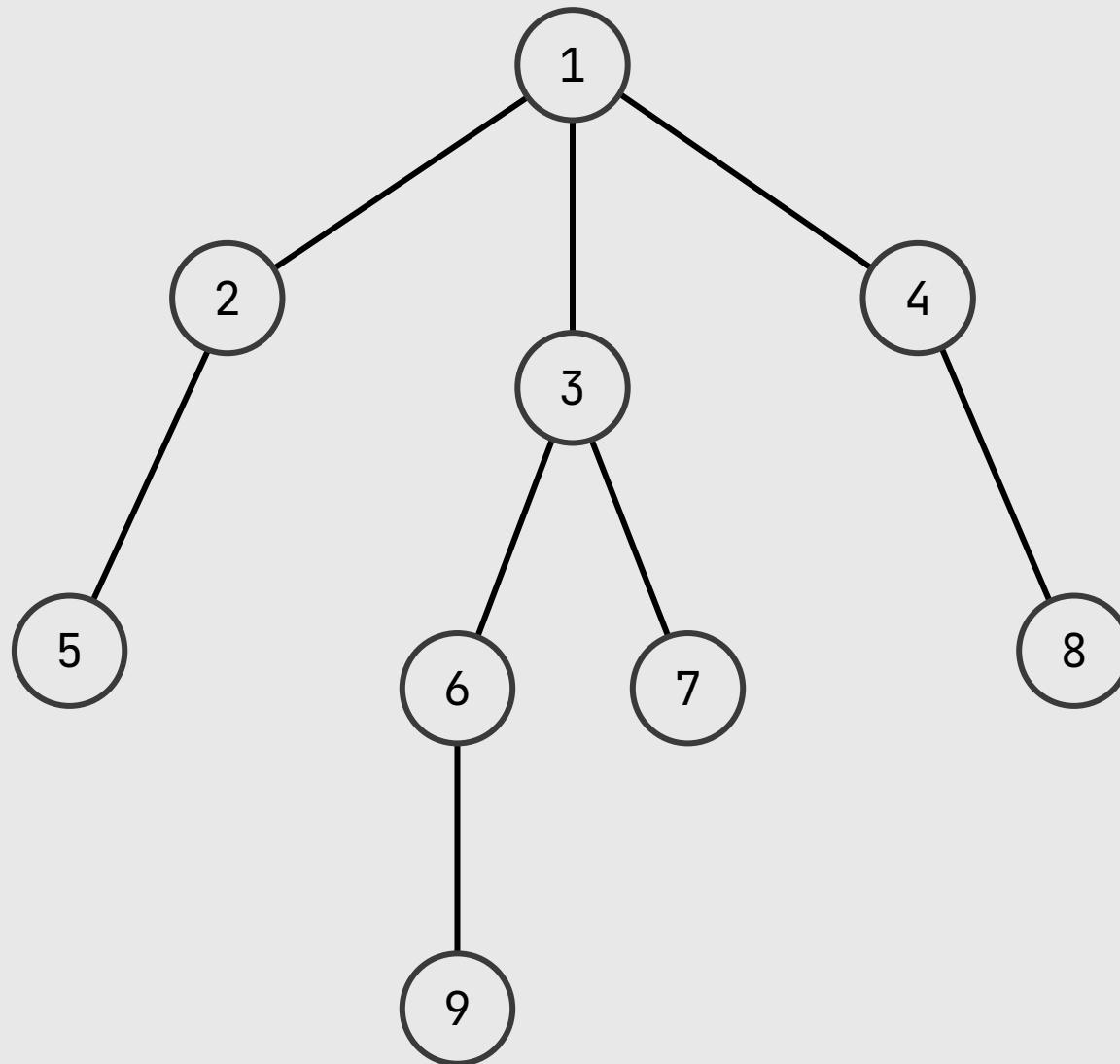


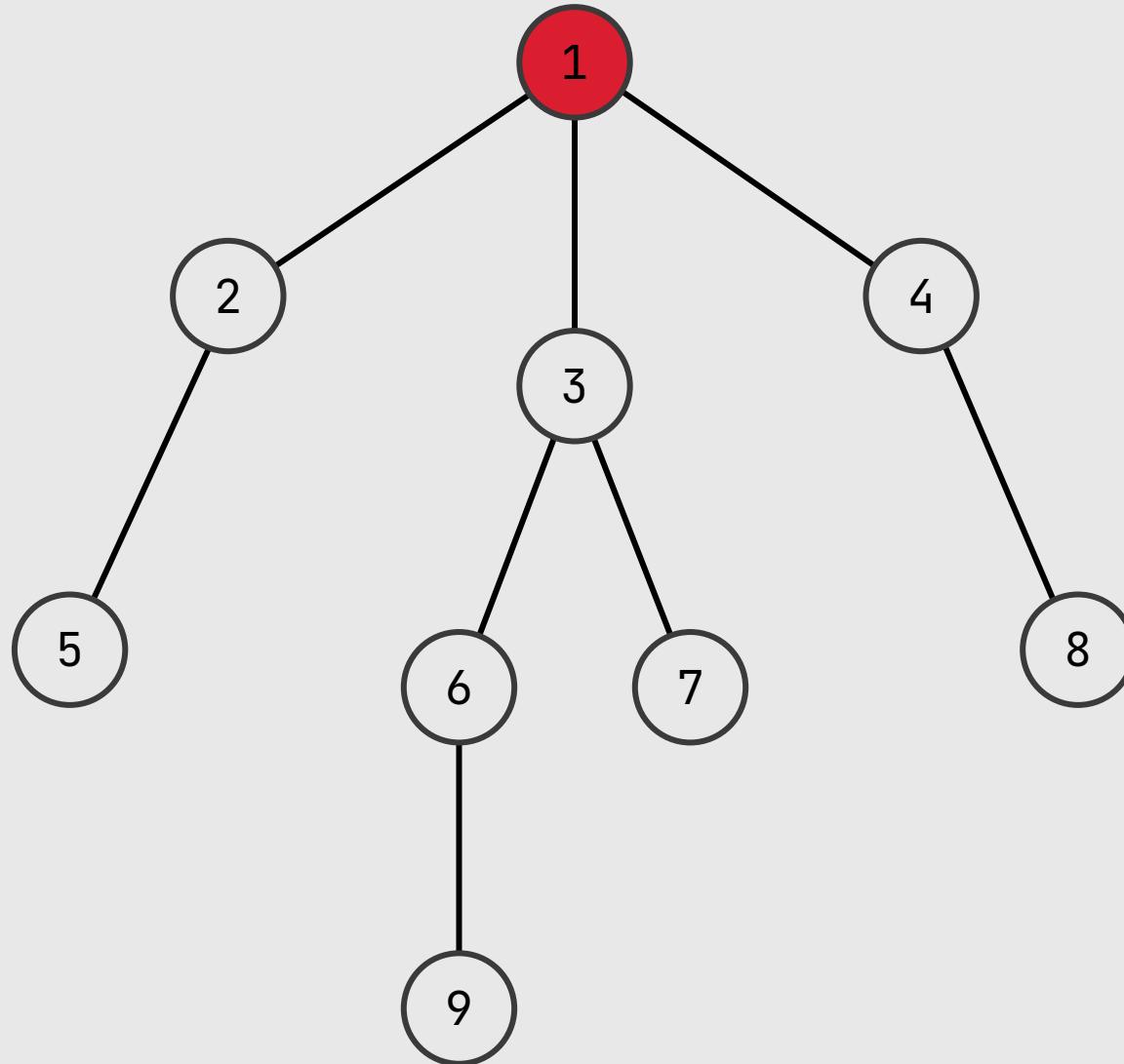
BFS

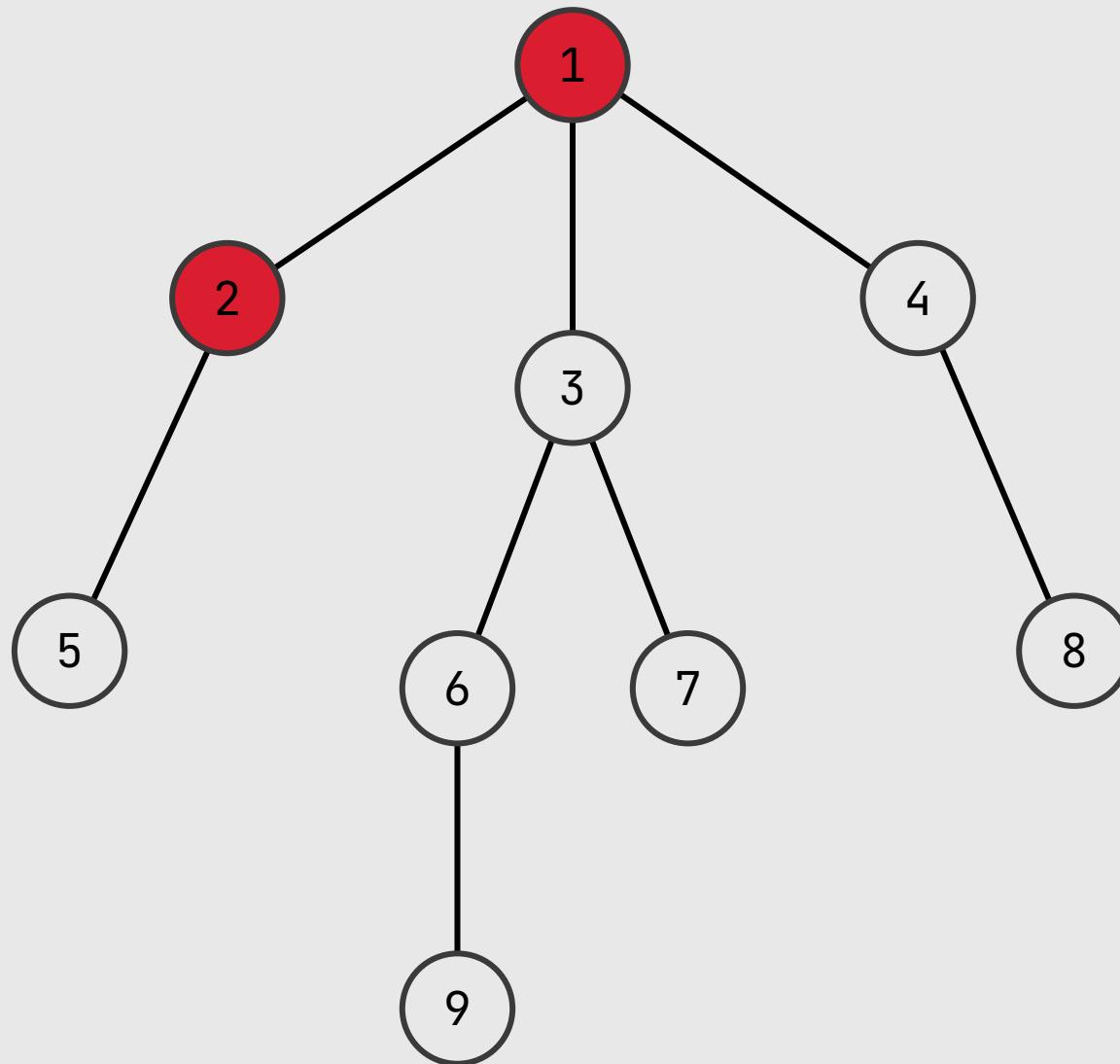
BREADTH-FIRST SEARCH / BUSCA EM LARGURA

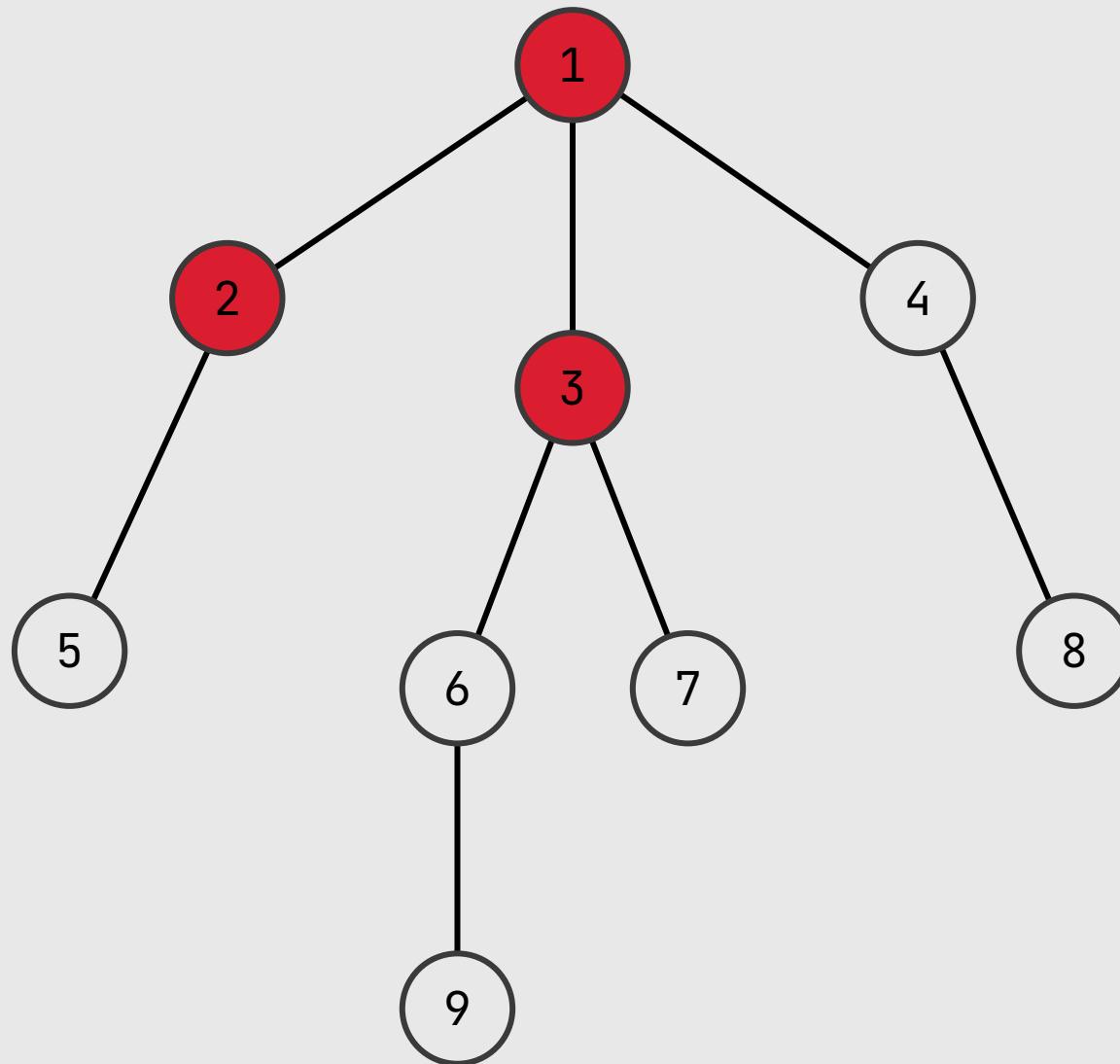


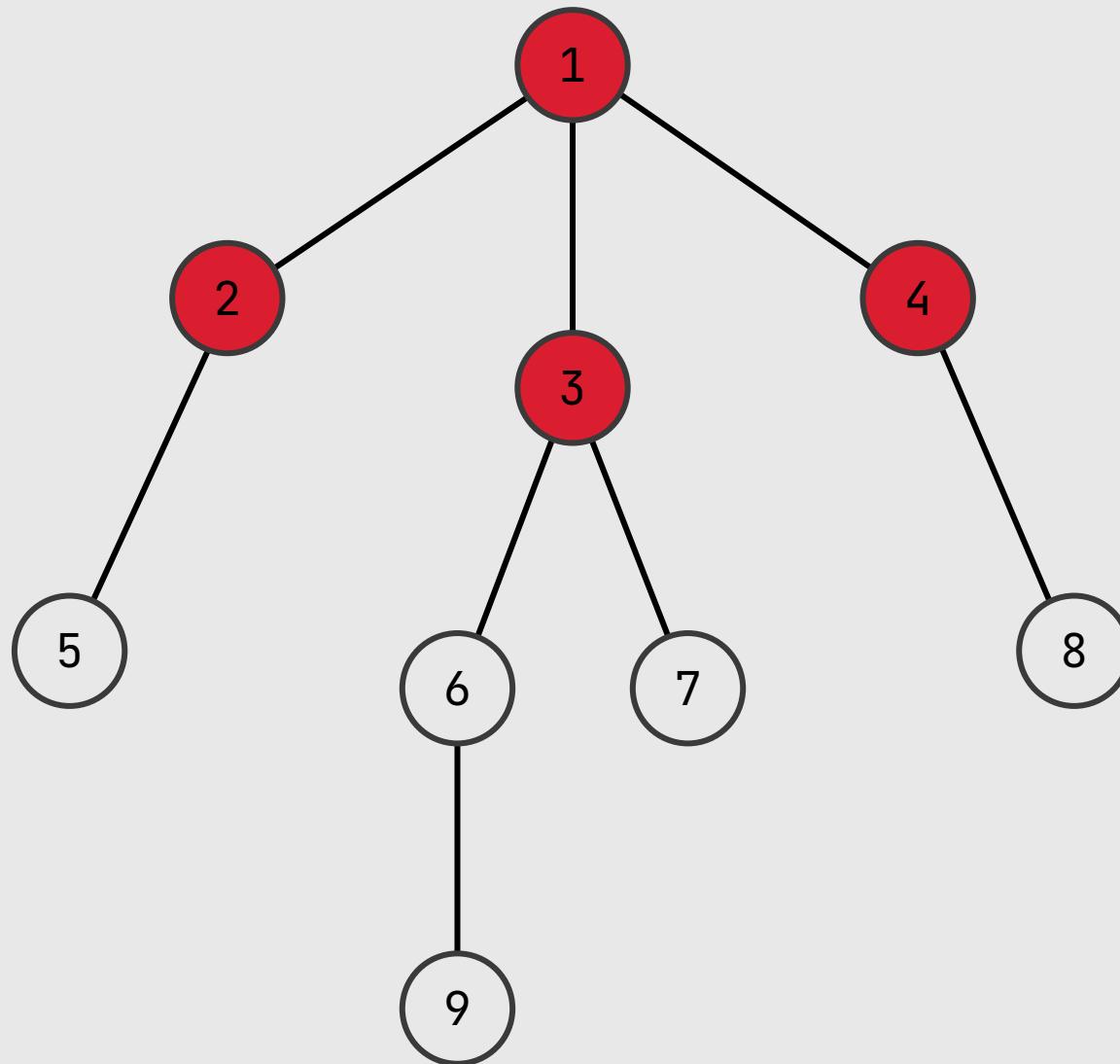
A BFS é uma busca por meio de uma FILA. Nós começamos por um vértice inicial e quando “visitamos” ele, adicionamos todos os seus “vizinhos” na fila para serem visitados. Esses então colocarão seus vizinhos que não foram visitados ainda na fila também! Uma consequência disso é que nós visitamos os vértices em “camadas”, em que cada camada representa a distância para o ponto inicial. Podemos usar a BFS para encontrar a menor distância em um grafo sem pesos.

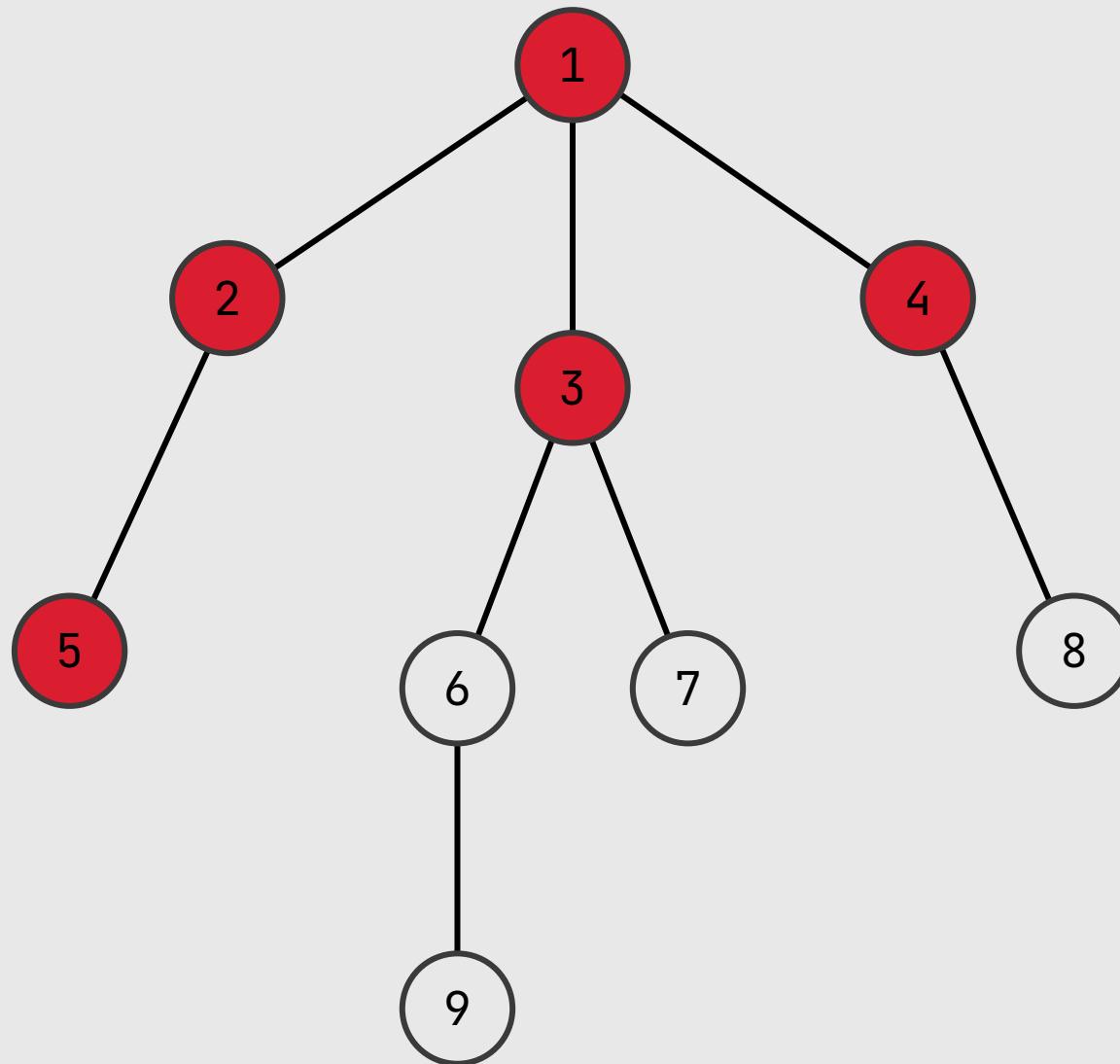


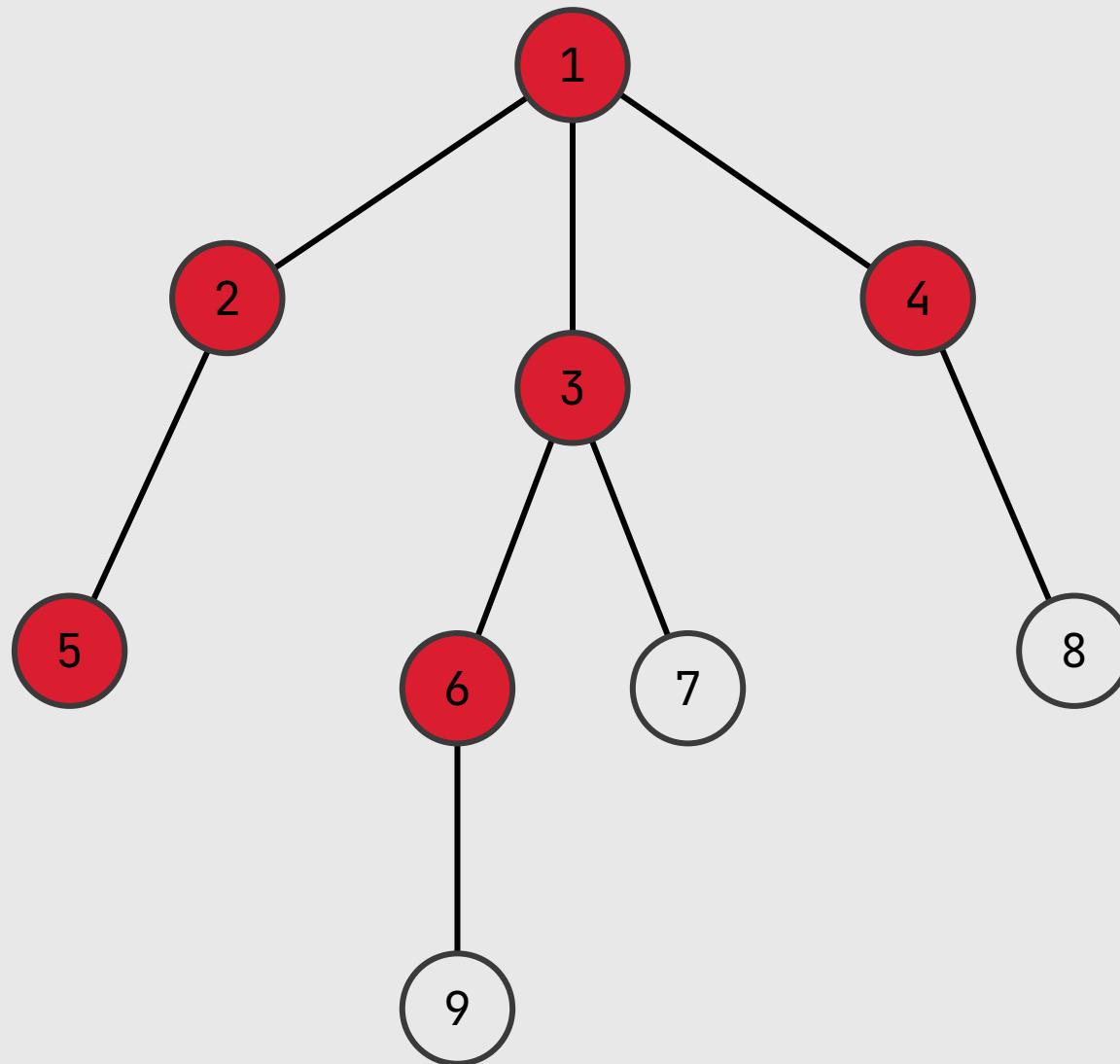


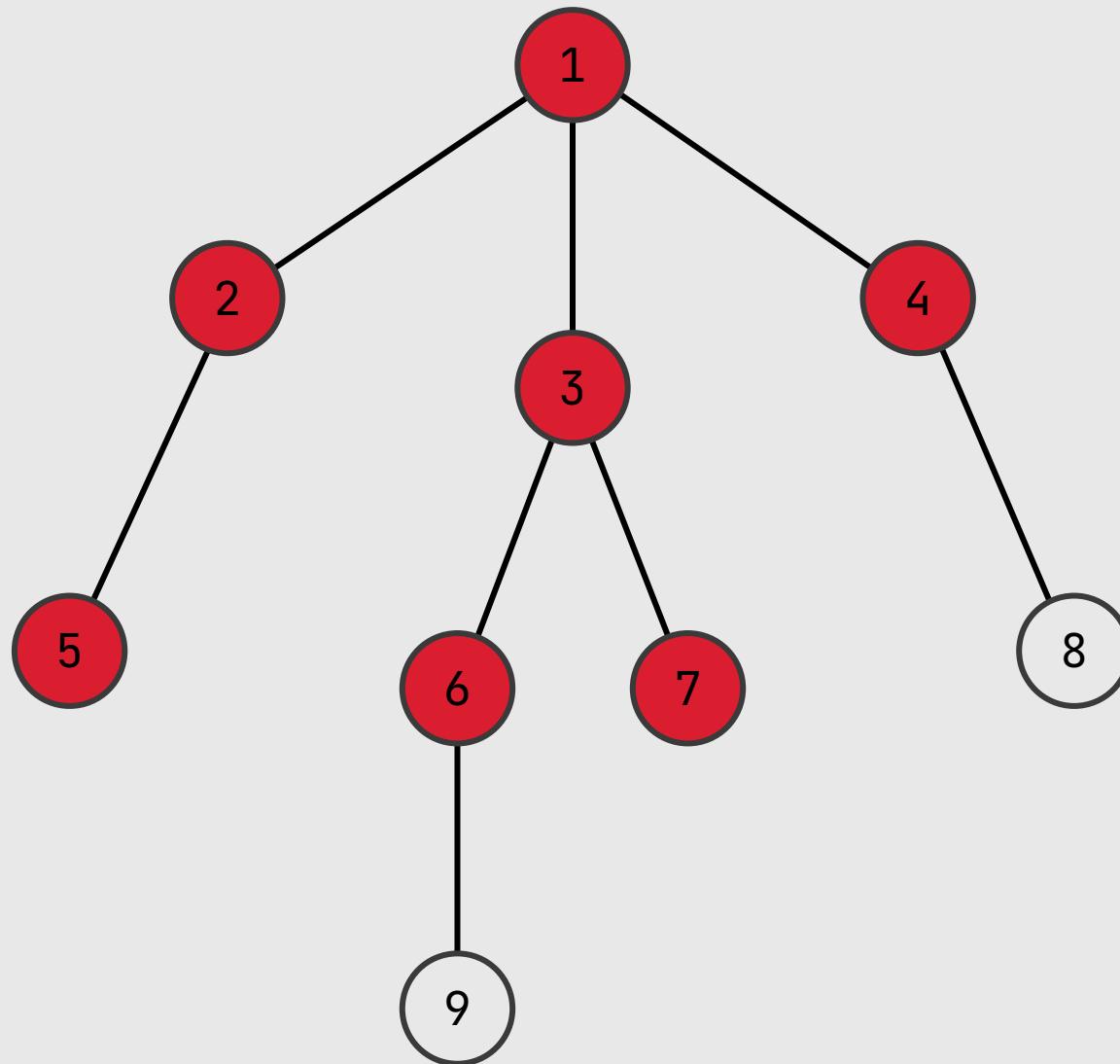


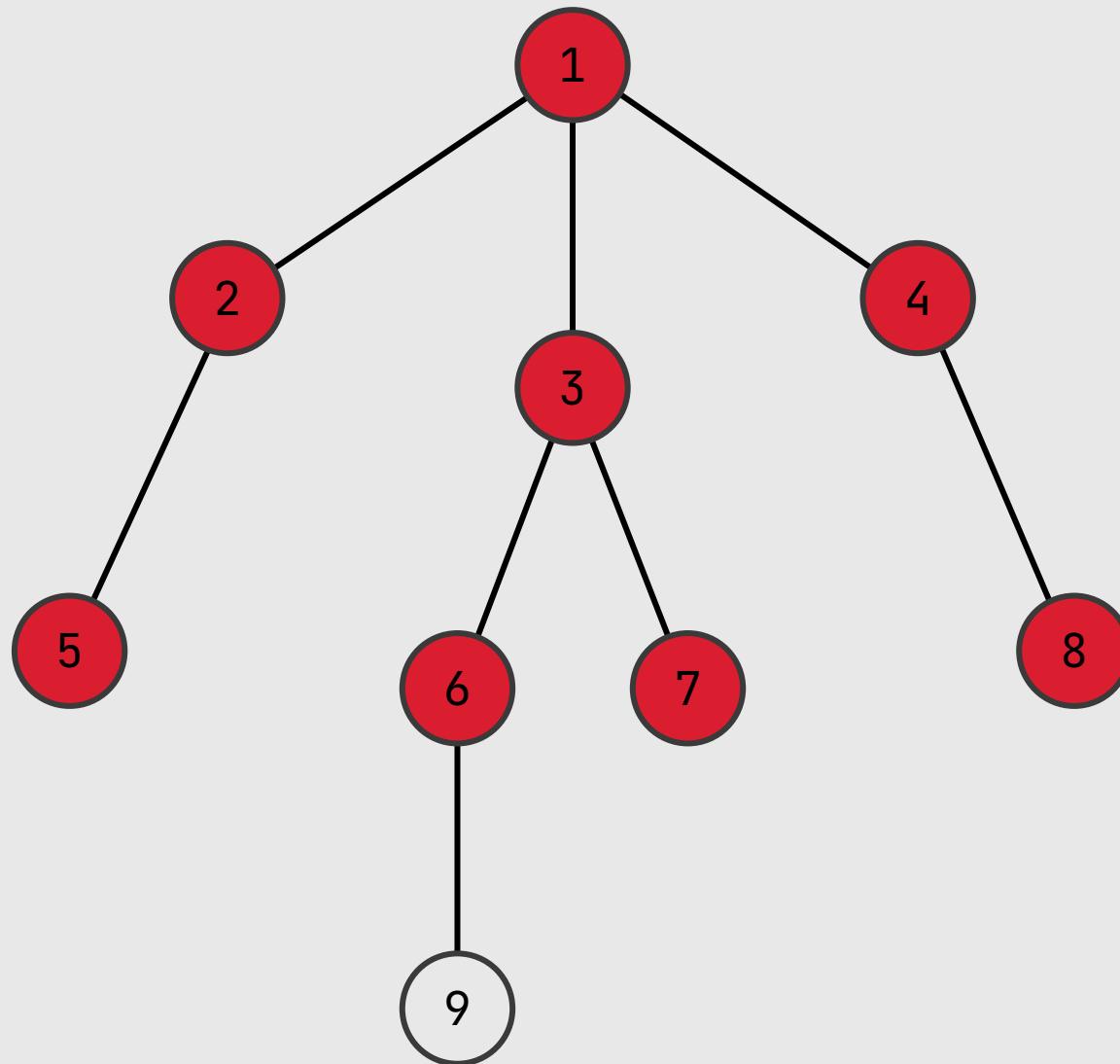


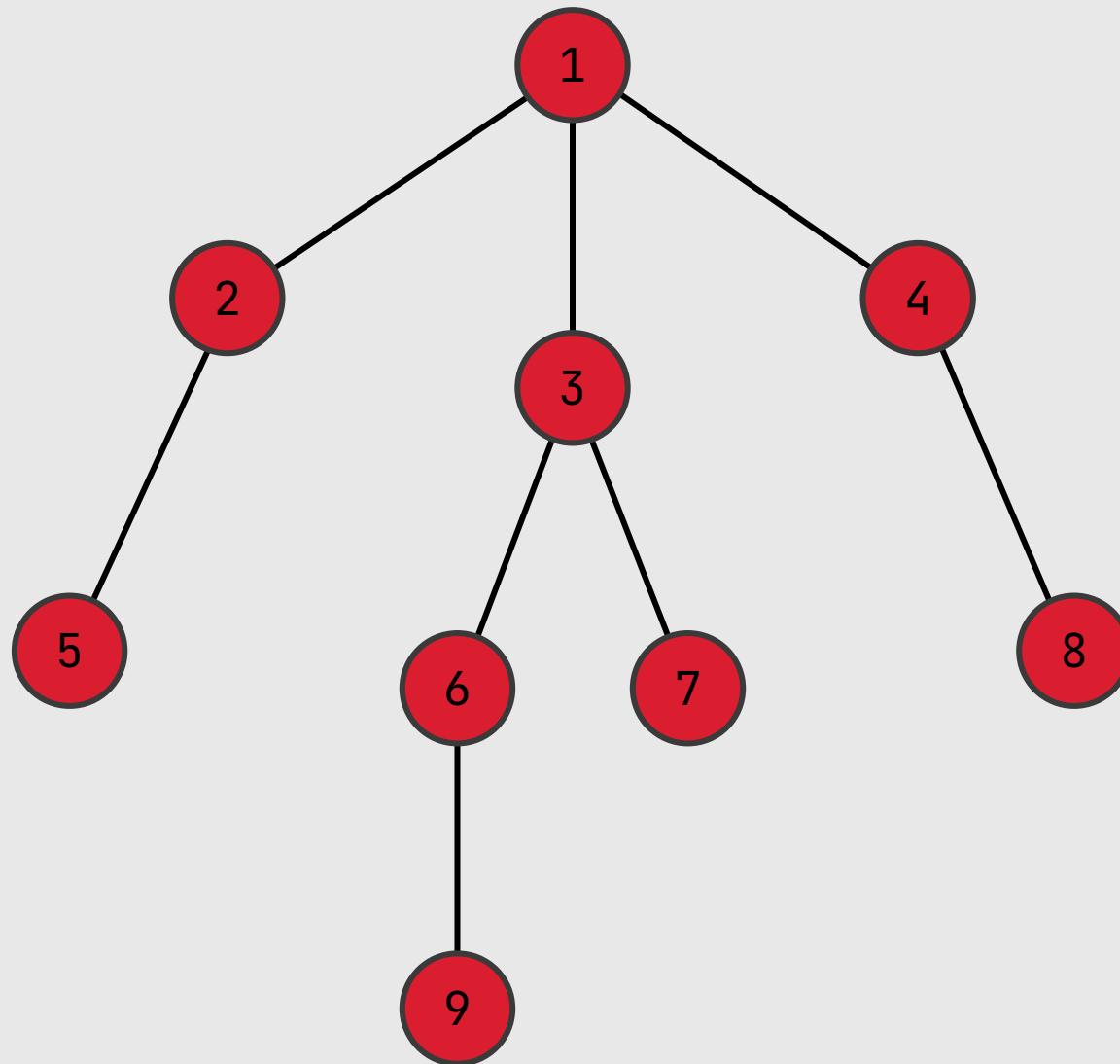












Pseudocódigo BFS



Maratona

CIn



```
bfs.cpp

int dist[MAXN];

void bfs(int b){
    memset(dist, -1, sizeof(dist));
    dist[b] = 0;
    queue<int>q;
    q.push(b);

    while(!q.empty()){
        int u = q.front(); q.pop();
        for(int v: adj[u]){
            if(dist[v] == -1){
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

A complexidade do algoritmo BFS é $O(V + E)$.

BFS em Grid



Maratona

CIn
Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


```
bfsgrid.cpp

char grid[MAXN][MAXN];
bool vis[MAXN][MAXN];

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

bool inRange(int x, int y){
    return x ≥ 0 && x < n && y ≥ 0 && y < m;
}

void bfs(){
    queue<pii>q;
    memset(vis, false, sizeof(vis));
    vis[0][0] = true;
    q.push({0, 0});
    int d = 0;
    while(!q.empty()){
        auto [x, y] = q.front(); q.pop();
        for(int i = 0; i<4; i++){
            int ax = x + dx[i], ay = y + dy[i];
            if(inRange(ax, ay) && !vis[ax][ay] && grid[ax][ay] == '.'){
                vis[ax][ay] = true;
                q.push({ax, ay});
            }
        }
    }
}
```



Bicoloridade

GRAFOS BIPARTIDOS

Bicoloridade

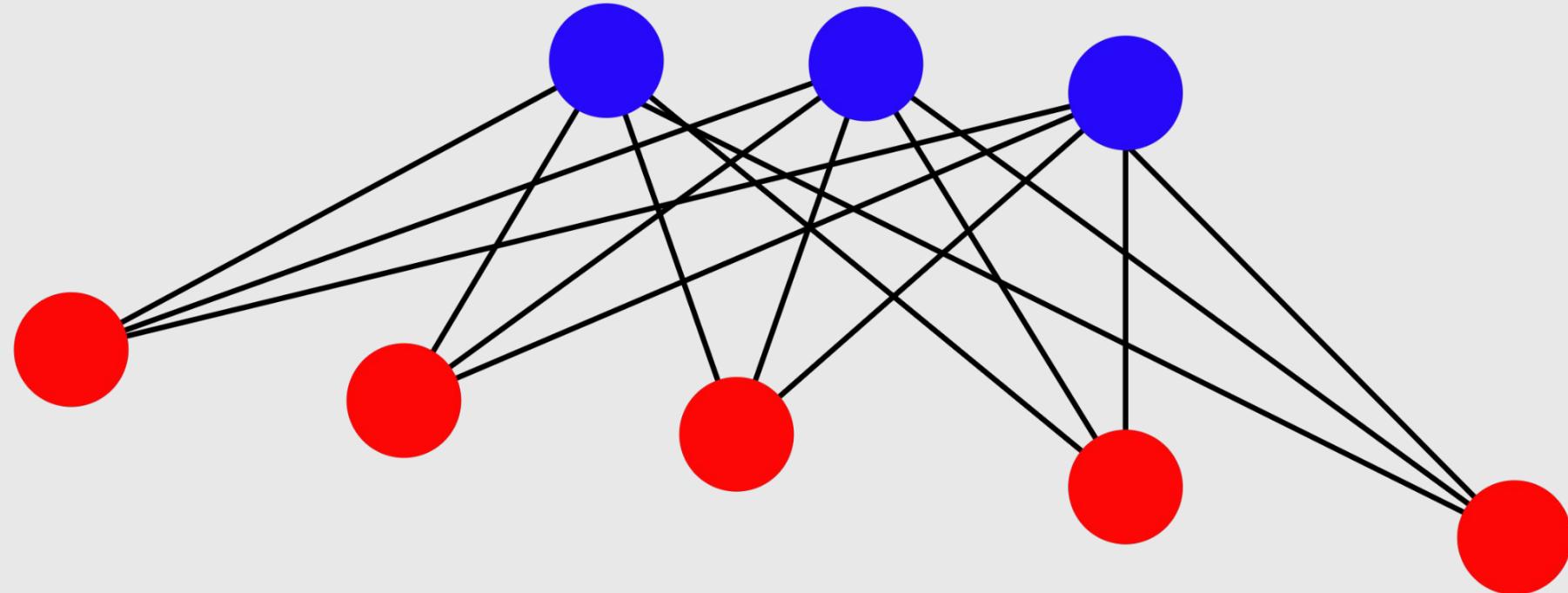


MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

- Um grafo bicolorível é um grafo bipartido.
- Queremos colorir os nós de um grafo com apenas duas cores, em que dois nós adjacentes não podem ter cores iguais.



Bicoloridade



MaratonaCIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

bcolorability.cpp

- □ ×

```
vector<vector<int>>adj;
vector<bool>vis, color;

bool dfs(int v){
    vis[v] = true;
    for(int u: adj[v]){
        if(!vis[u]){
            color[u] = !color[v];
            if(!dfs(u)) return false;
        }
        else if(color[v] == color[u]) return false;
    }
    return true;
}
```



Toposort

ORDEM TOPOLOGICA

Toposort

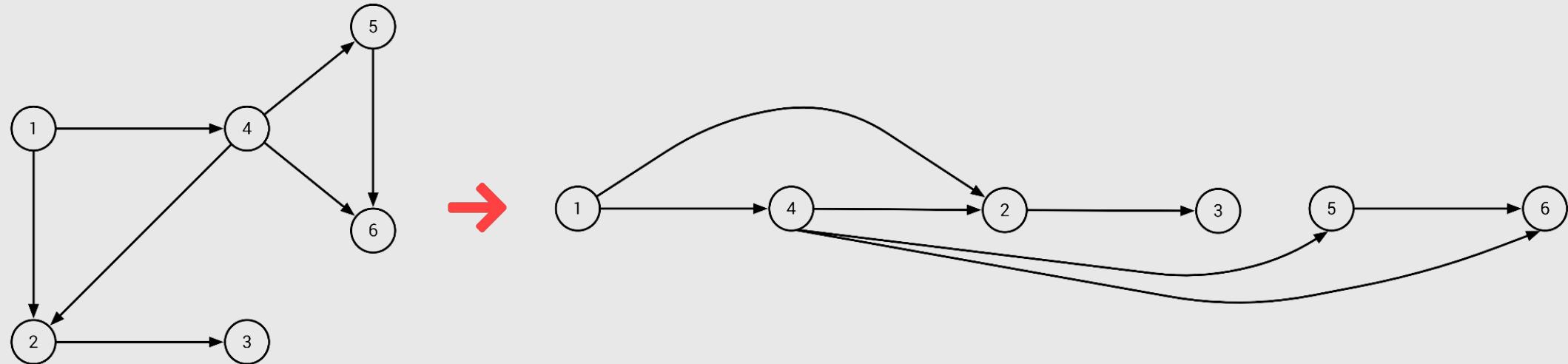


Maratona**CIn**

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


Em um grafo acíclico direcionado (DAG), uma ordem topológica é uma forma de listar todos os vértices de modo que, para cada aresta que vai de U para V, o vértice U apareça antes de V na sequência. Ou seja, nenhum vértice aparece antes de outro do qual ele depende — se há um caminho de U até V, então U vem antes de V na ordem.



Toposort

Para encontrar uma ordem topológica, usamos o Algoritmo de Kahn.

Primeiro, identificamos todos os vértices com grau de entrada igual a 0, ou seja, aqueles que não recebem nenhuma aresta. Esses são os primeiros da sequência, pois não dependem de ninguém.

Depois, removemos esses vértices do grafo, junto com as arestas que saem deles. Isso pode fazer com que outros vértices passem a ter grau de entrada 0. Repetimos então esse passo de escolher os vértices com grau 0 e em seguida “excluir” eles do grafo.



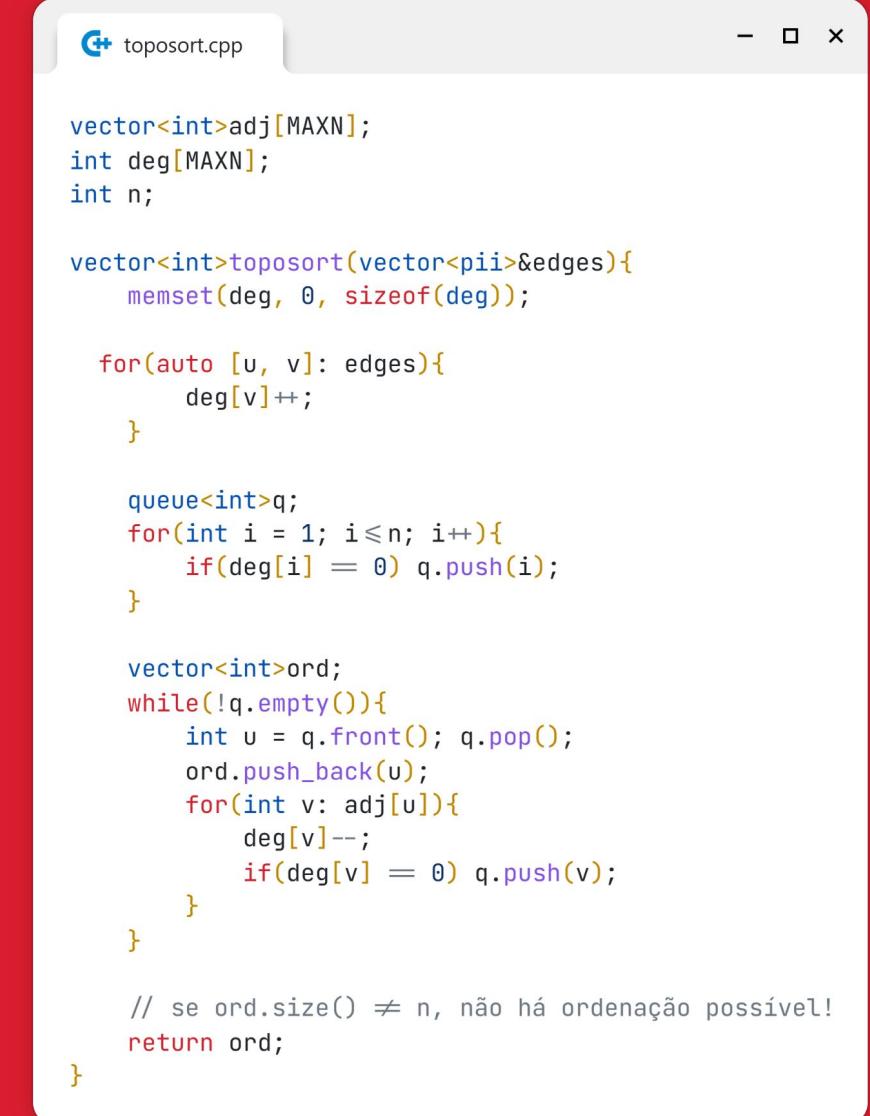
Maratona

CIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

VIRTUS IMPAVIDA



A screenshot of a code editor window titled "toposort.cpp". The code implements the Kahn's algorithm for topological sorting. It uses vectors to store adjacency lists, degrees, and the sorted order. A queue is used to identify vertices with zero incoming edges. The code iterates through these vertices, removing them from the graph and adding them to the sorted order until all vertices have been processed or it is determined that a topological ordering is impossible.

```
vector<int>adj[MAXN];
int deg[MAXN];
int n;

vector<int>toposort(vector<pii>&edges){
    memset(deg, 0, sizeof(deg));

    for(auto [u, v]: edges){
        deg[v]++;
    }

    queue<int>q;
    for(int i = 1; i <= n; i++){
        if(deg[i] == 0) q.push(i);
    }

    vector<int>ord;
    while(!q.empty()){
        int u = q.front(); q.pop();
        ord.push_back(u);
        for(int v: adj[u]){
            deg[v]--;
            if(deg[v] == 0) q.push(v);
        }
    }

    // se ord.size() != n, não há ordenação possível!
    return ord;
}
```



Maratona**CIn**



Problemas Clássicos



BFS MULTISOURCE



Problema: Dado um grafo com K vértices marcados. Diga para cada vértice a menor distância dele para um vértice marcado.

Solução: No **início** da BFS defina a **distância de cada vértice marcado como 0** e **inicie a fila da BFS com todos eles**.

Retomando a analogia, podemos notar que vamos ter **camadas se formando ao redor de cada nó inicial** e um vértice qualquer vai ser alcançado primeiro justamente pela origem mais próxima dele.

Além disso, cada vértice **continua sendo visitado uma única vez**.



ENCONTRAR CICLOS

Encontrar Ciclos

Para encontrar ciclos em um grafo (direcionado ou não), podemos usar uma busca em profundidade (DFS). Começamos a percorrer o grafo visitando vértices livremente. Se, em algum momento, chegarmos a um vértice que já foi visitado e **não é o vértice anterior** (de onde viemos), isso indica que existe um ciclo.

Nesse caso, podemos voltar pelo caminho percorrido até reencontrar o mesmo vértice e registrar todos os nós que fazem parte do ciclo.

```
ciclos.cpp

bool vis[MAXN];
vector<int> ciclo;

int dfs(int atual, int anterior) {
    if (vis[atual]) return atual;
    vis[atual] = true;

    for (int nxt : adj[atual]) {
        if (nxt == anterior) continue;
        int fim = dfs(nxt, atual);

        if (fim == -2) return -2;
        if (fim != -1) {
            ciclo.push_back(atual);
            if (atual == fim) return -2;
            else return fim;
        }
    }

    return -1;
}
```

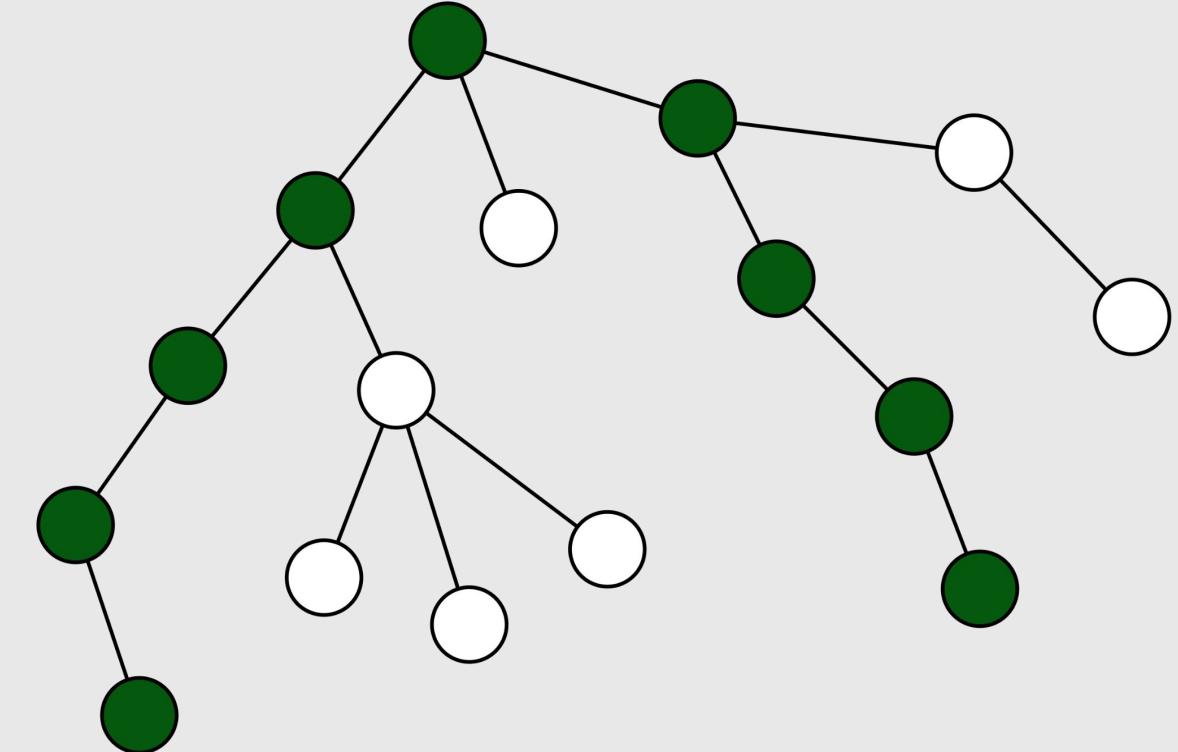


TREE DIAMETER

DOIS VÉRTICES MAIS DISTANTES DE UMA ÁRVORE

Diâmetro de Árvore

Dado uma árvore, o seu diâmetro é a maior distância entre dois vértices. Podem existir diferentes pares que representam o diâmetro, mas a distância entre cada par é a maior possível.



Diâmetro de Árvore



Primeiro, escolhe-se qualquer nó e realiza-se uma busca (BFS ou DFS) a partir dele para encontrar o nó mais distante, chamado de A. Em seguida, faz-se uma nova busca começando em A para encontrar o nó mais distante dele, chamado de B.

A distância entre A e B é o diâmetro da árvore, ou seja, o comprimento do maior caminho simples entre dois nós

OBS: Em uma árvore só existe um único caminho entre quaisquer 2 vértices, então a distância pode ser calculada tanto com BFS como com DFS.



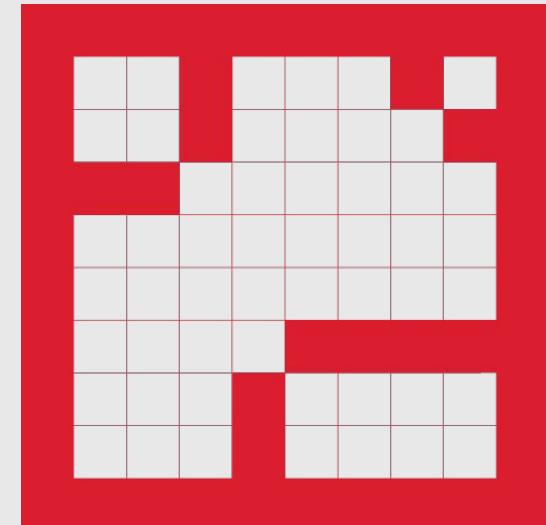
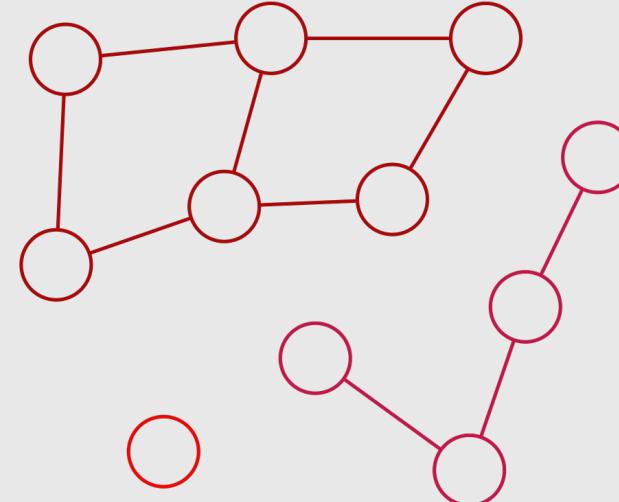
Maratona**CIn**



COMPONENTES CONEXOS & FLOOD FILL

Comp. Conexos & Flood Fill

Um componente conexo é um subconjunto de vértices do grafo original em que todo mundo consegue chegar em todo mundo. Podemos encontrar o componente conexo de um vértice com apenas uma DFS/BFS. Em matrizes chamamos essa mesma ideia de Flood Fill. Funciona como se fosse “pintar um desenho dentro das linhas”.



Comp. Conexos & Flood Fill

Um componente conexo é um subconjunto de vértices do grafo original em que todo mundo consegue chegar em todo mundo. Podemos encontrar o componente conexo de um vértice com apenas uma DFS/BFS. Em matrizes chamamos essa mesma ideia de Flood Fill. Funciona como se fosse “pintar um desenho dentro das linhas”.

```
compconexos.cpp

int componente[MAXN];

void dfs(int u, int comp){
    componente[u] = comp;
    for(int v: adj[u]){
        if(componente[v] == -1){
            dfs(v, comp);
        }
    }
}

void init(){
    memset(componente, -1, sizeof(componente));

    int num = 0;
    for(int i = 1; i <= n; i++){
        if(componente[i] == -1){
            num++;
            dfs(i, num);
        }
    }
}
```



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

VIRTUS IMPAVIDA

Dúvidas?

Próximo passo:
Fazer o Homework!



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

That's all, Folks!

