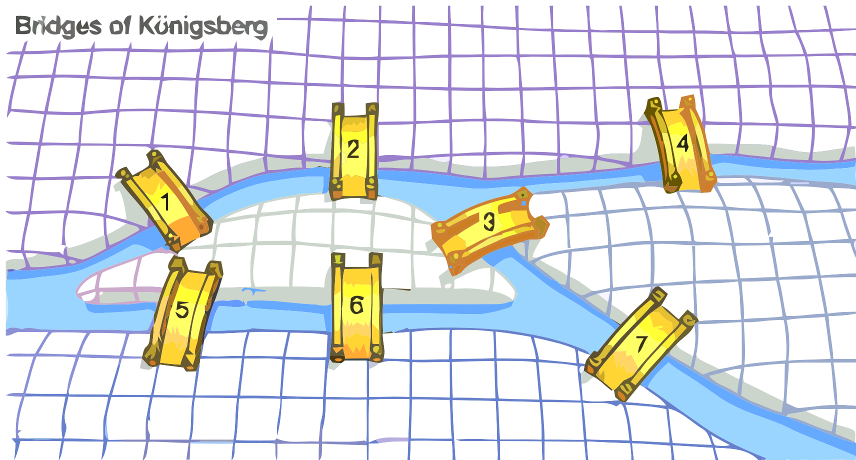
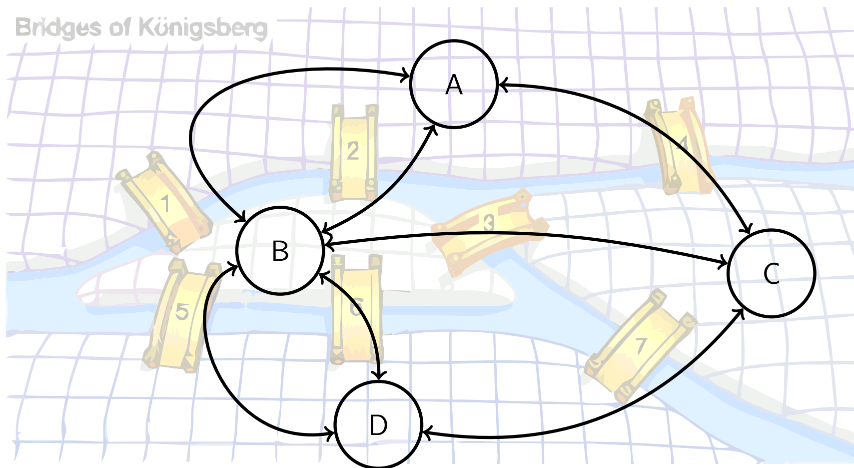
Grafos I

~~Nico Lehmann~~ ~~Benjamín Rubio~~
~~Martín~~ Muñoz
Ignacio
Departamento de Ciencias de la Computación,
~~Universidad de Chile~~ PUC

Invernal Campamento de Programación Competitiva

PUENTES DE KÖNIGSBERG

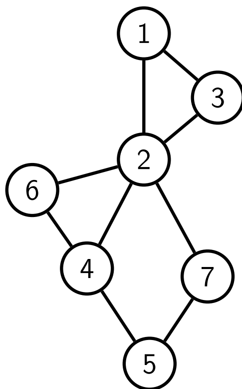



Bridges of Königsberg
A
B
C
D
1
2
3
4
5
6
7

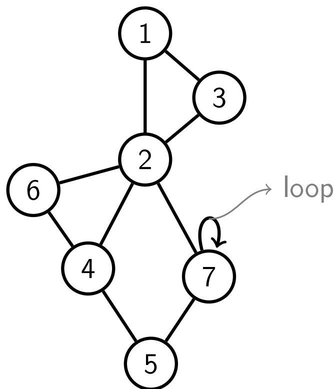
DEFINICIÓN DE GRAFO

Grafo Un grafo G es un par (V, E), donde V es el conjunto de nodos (o vértices), y $E \subseteq V \times V$ es el conjunto de aristas.

Grafo Un grafo G es un par (V, E), donde V es el conjunto de nodos (o vértices), y $E \subseteq V \times V$ es el conjunto de aristas.

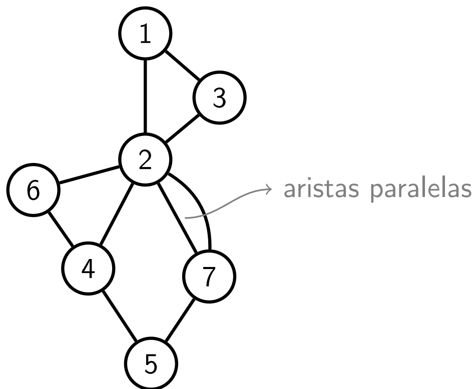


Grafo Un grafo G es un par (V, E), donde V es el conjunto de nodos (o vértices), y $E \subseteq V \times V$ es el conjunto de aristas.

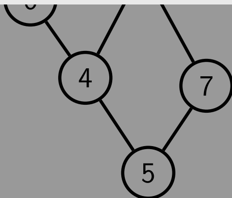


DEFINICIÓN DE GRAFO

Grafo Un grafo G es un par (V, E), donde V es el conjunto de nodos (o vértices), y $E \subseteq V \times V$ es el conjunto de aristas.



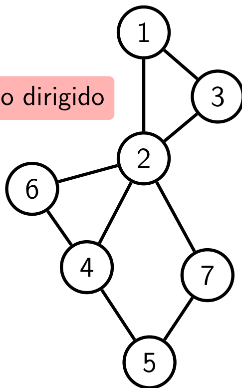
Grafo Un grafo G es un par (V, E), donde V es el conjunto de nodos (o vértices), y $E \subseteq V \times V$ es el conjunto de aristas.

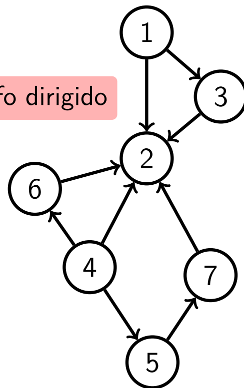


Grafo Simple

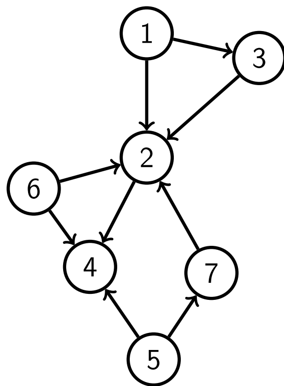
Si un grafo G no contiene loops ni aristas paralelas decimos que es un grafo simple.

DIRECCIÓN ARISTAS



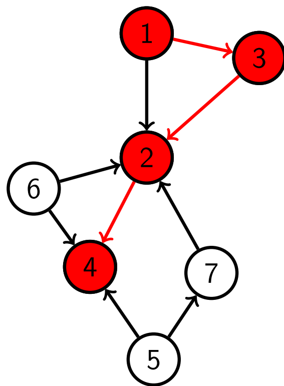


CAMINO (SIMPLE)

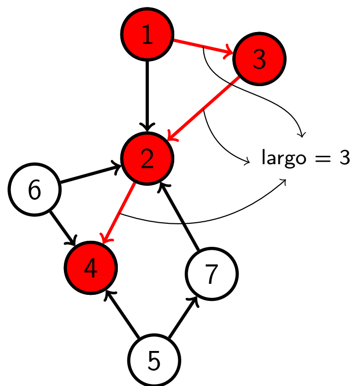


Camino Secuencia de nodos (distintos) unidos por aristas.

CAMINO (SIMPLE)



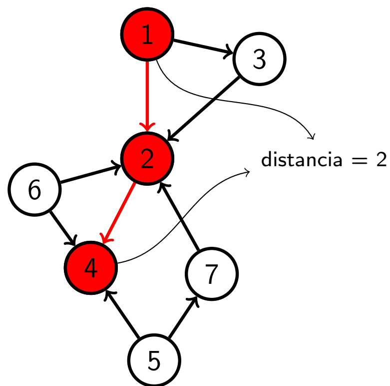
Camino Secuencia de nodos (distintos) unidos por aristas.



Camino Secuencia de nodos (distintos) unidos por aristas.

Largo Cantidad de aristas en el camino.

CAMINO (SIMPLE)



Camino Secuencia de nodos (distintos) unidos por aristas.

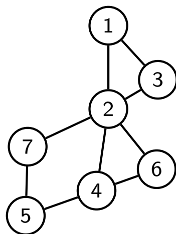
Largo Cantidad de aristas en el camino.

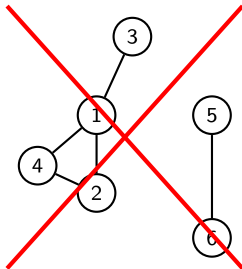
Distancia Largo del camino más corto entre dos nodos.

CONECTIVIDAD (GRAFOS NO DIRIGIDOS)

Un grafo no dirigido es **conexo** si existe un camino entre cada par de nodos.

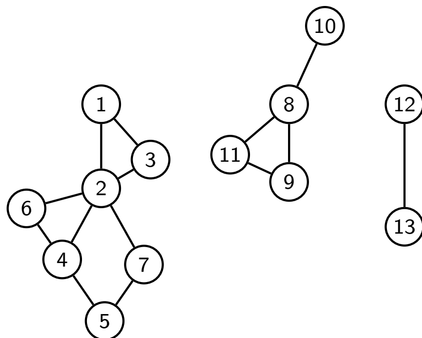
Un grafo no dirigido es **conexo** si existe un camino entre cada par de nodos.



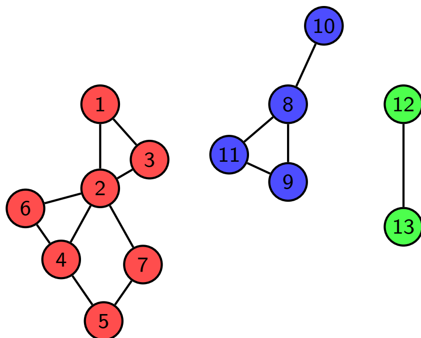


COMPONENTE CONEXA (GRAFOS NO DIRIGIDOS)

Una **componente conexa** es un subgrafo conexo maximal.

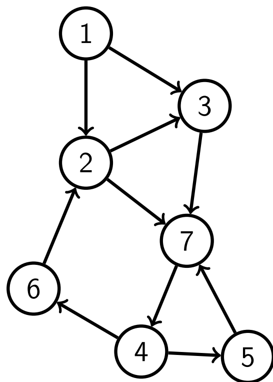


Una **componente conexa** es un subgrafo conexo maximal.



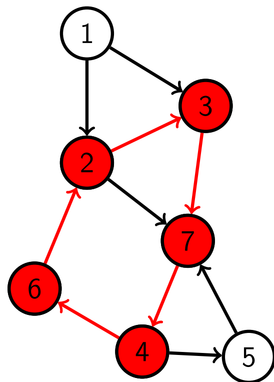
CICLO (SIMPLE)

Ciclo Un ciclo (simple) es un camino (simple) que parte y termina en el mismo nodo.

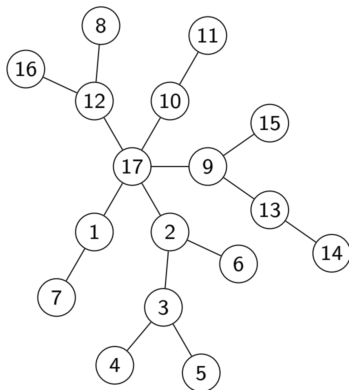


CICLO (SIMPLE)

Ciclo Un ciclo (simple) es un camino (simple) que parte y termina en el mismo nodo.

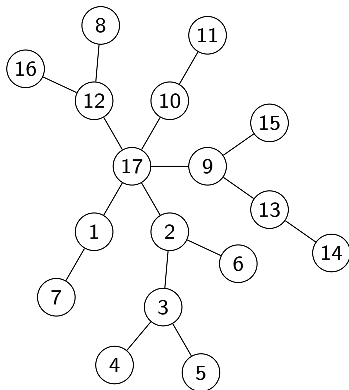


Un **árbol** es un grafo conexo G tal que



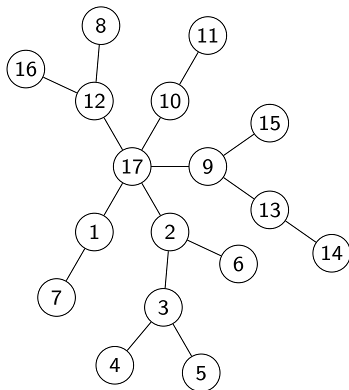
Un **árbol** es un grafo conexo G tal que

- G no tiene ciclos.



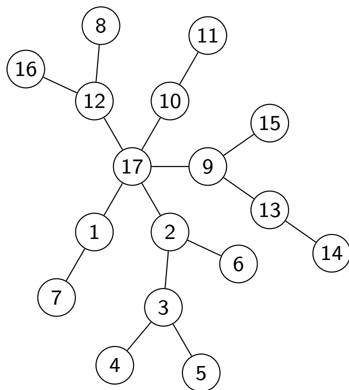
Un **árbol** es un grafo conexo G tal que

- G no tiene ciclos.
- Si se le quita una arista deja de ser conexo.



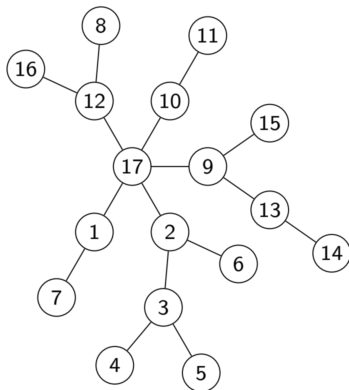
Un **árbol** es un grafo conexo G tal que

- G no tiene ciclos.
- Si se le quita una arista deja de ser conexo.
- Si se le agrega una arista G pasa a tener ciclos.



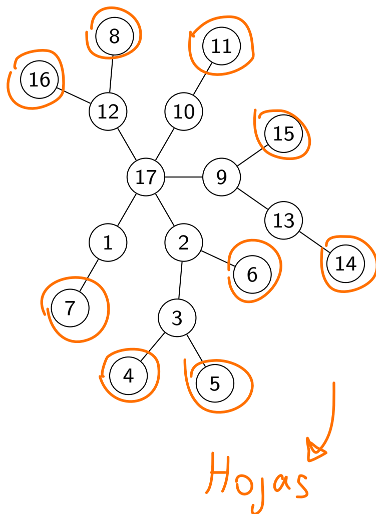
Un **árbol** es un grafo conexo G tal que

- G no tiene ciclos.
- Si se le quita una arista deja de ser conexo.
- Si se le agrega una arista G pasa a tener ciclos.
- Existe un solo camino entre cada par de nodos de G.



Un **árbol** es un grafo conexo G tal que

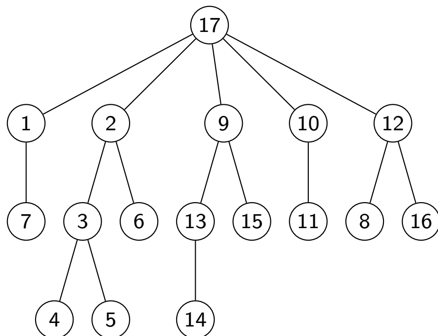
- G no tiene ciclos.
- Si se le quita una arista deja de ser conexo.
- Si se le agrega una arista G pasa a tener ciclos.
- Existe un solo camino entre cada par de nodos de G.
- G tiene n nodos y $n-1$ aristas.



Un **árbol** es un grafo conexo G tal que

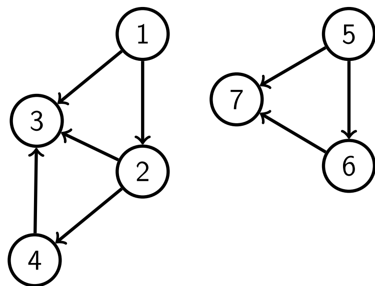
- G no tiene ciclos.
- Si se le quita una arista deja de ser conexo.
- Si se le agrega una arista G pasa a tener ciclos.
- Existe un solo camino entre cada par de nodos de G.
- G tiene n nodos y $n-1$ aristas.

Árbol enraizado (rooteado):

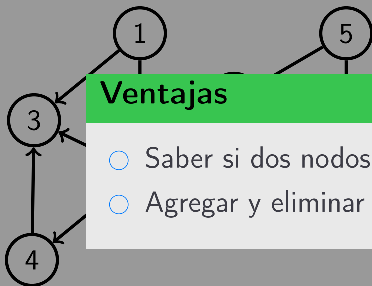


REPRESENTACIÓN DE GRAFOS

MATRIZ DE ADYACENCIA



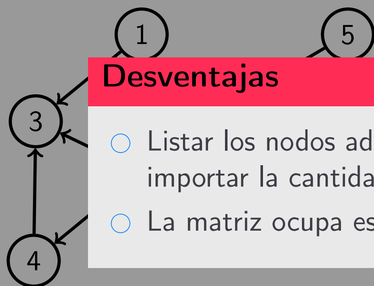
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	1	1	0	0	0
3	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0



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

Ventajas

- Saber si dos nodos están conectados es $\mathcal{O}(1)$.
- Agregar y eliminar aristas es $\mathcal{O}(1)$.

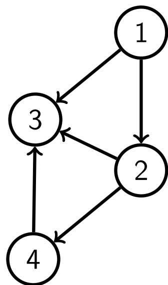


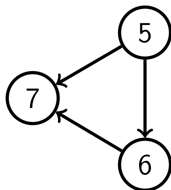
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	1
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	1
7	0	0	0	0	0	0	0

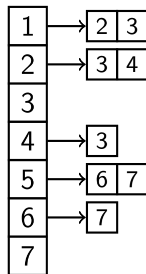
Desventajas

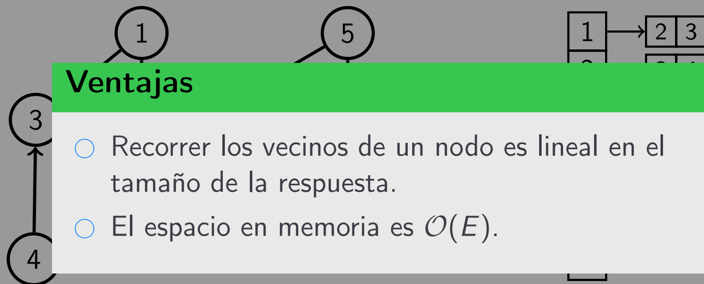
- Listar los nodos adyacentes a otro es $\mathcal{O}(|V|)$, sin importar la cantidad de vecinos que tenga.
- La matriz ocupa espacio $\mathcal{O}(|V|^2)$ siempre.

LISTA DE ADYACENCIA



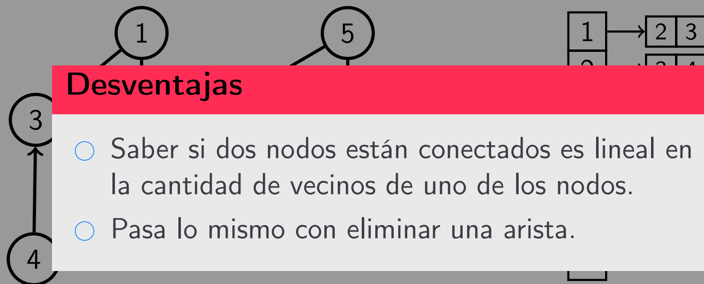






Ventajas

- Recorrer los vecinos de un nodo es lineal en el tamaño de la respuesta.
- El espacio en memoria es $\mathcal{O}(E)$.



Desventajas

- Saber si dos nodos están conectados es lineal en la cantidad de vecinos de uno de los nodos.
- Pasa lo mismo con eliminar una arista.

LISTA DE ADYACENCIA EN C++

```
vector<vector<int> > graph;

int main() {
  int N = 3;
  graph.resize(N);

  // 0 -> 1, 2
  graph[0].push_back(1);
  graph[0].push_back(2);

  for (int v : graph[0])
    printf("%d\n", v);

  return 0;
}
```

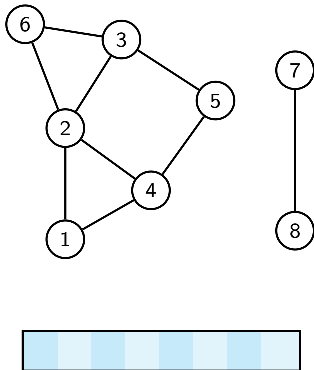
GRAFOS IMPLÍCITOS

- Algunas veces la estructura de un problema tiene asociado un grafo *implícito*. Ejemplo típico: un laberinto.

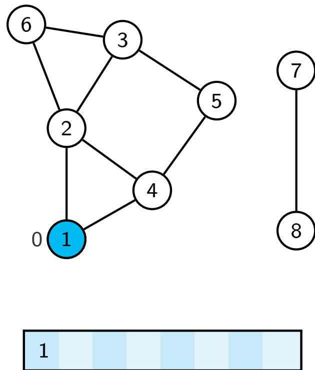
RECORRER UN GRAFO: BFS Y DFS

BREADTH FIRST SEARCH (BFS)

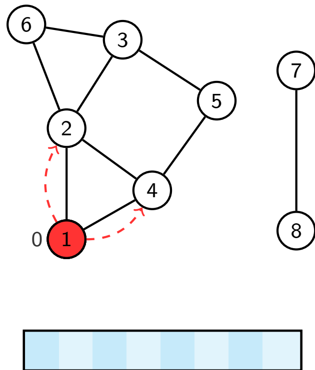
BREADTH FIRST SEARCH (BFS)

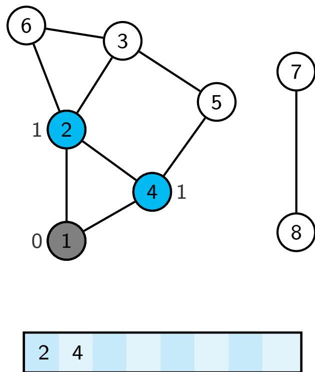


BREADTH FIRST SEARCH (BFS)

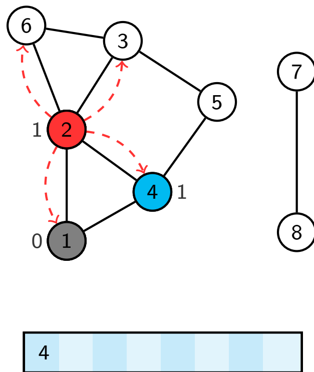


BREADTH FIRST SEARCH (BFS)

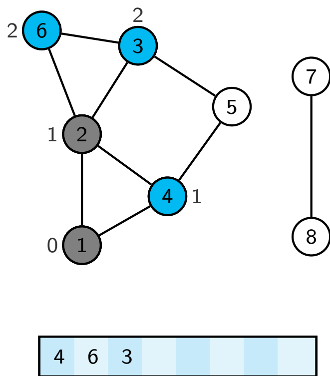


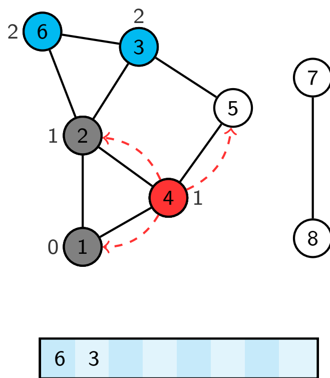

6
3
7
5
1
2
4
1
0
1
8
2
4

BREADTH FIRST SEARCH (BFS)

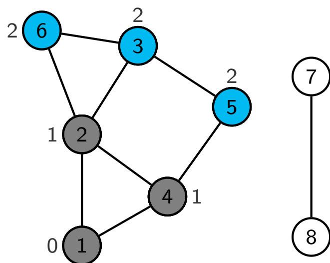


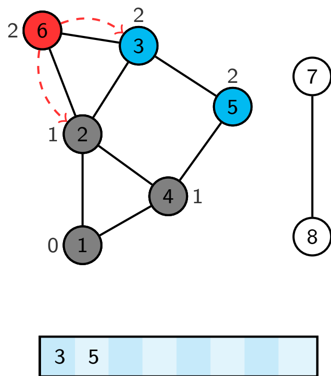
BREADTH FIRST SEARCH (BFS)



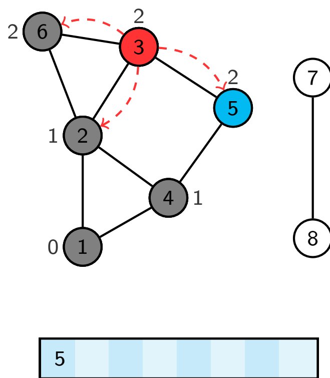

2
6
2
3
7
5
1
2
4
1
0
1
8
6
3

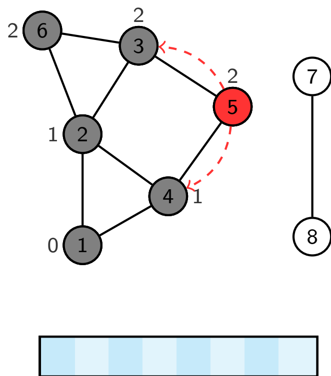
BREADTH FIRST SEARCH (BFS)

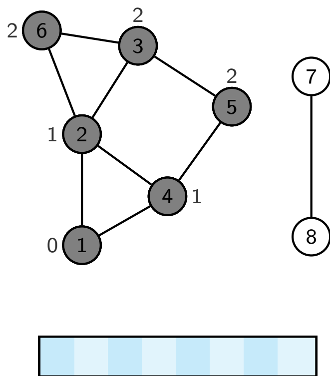



2
6
2
3
2
5
7
1
2
4
1
0
1
8
3
5

BREADTH FIRST SEARCH (BFS)




2
6
2
3
2
5
7
1
2
4
1
0
1
8


2
6
2
3
2
5
7
1
2
4
1
0
1
8

2 6 2

Notar que

- Un BFS corre en tiempo $\mathcal{O}(|V| + |E|)$

Notar que

- Un BFS corre en tiempo $\mathcal{O}(|V| + |E|)$
- Con un BFS podemos calcular la distancia de un nodo a todos los demás.

Notar que

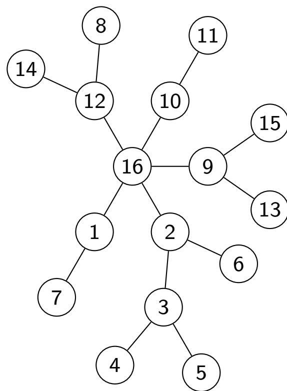
- Un BFS corre en tiempo $\mathcal{O}(|V| + |E|)$
- Con un BFS podemos calcular la distancia de un nodo a todos los demás.
- Podemos incluso guardar el camino que une los nodos con un arreglo `parent[u]` que almacena el nodo desde que se visito `u`.

BREATH FIRST SEARCH (BFS)

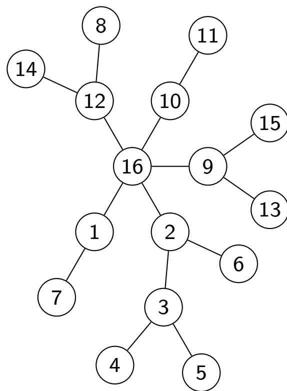
```
vector<vector<int> > graph;
vector<int> dist;
void bfs_visit(int s) {
  queue<int> Q;
  dist[s] = 0;
  Q.push(s);
  while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int v : graph[u])
      if (dist[v] == -1) {
        dist[v] = dist[u] + 1;
        parent[v] = u;
        Q.push(v);
      }
  }
}
void bfs() {
  dist.resize(graph.size(), -1);
  parent.resize(graph.size(), -1);
  for (int u = 0; u < graph.size(); ++u)
    if (dist[u] == -1)
      bfs_visit(u);
}
```

Diámetro El diámetro de un grafo G es la distancia entre los nodos más alejados.

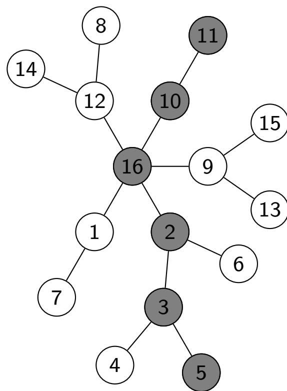


- Puede haber más de un camino con el mismo largo que el diámetro.

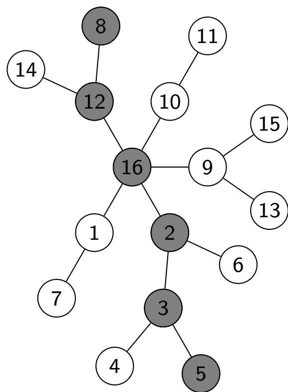


DIÁMETRO EN UN ÁRBOL

- Puede haber más de un camino con el mismo largo que el diámetro.

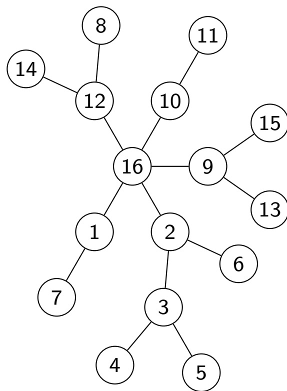


- Puede haber más de un camino con el mismo largo que el diámetro.

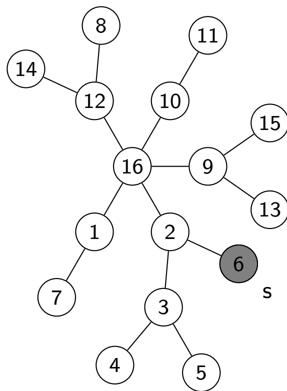


DIÁMETRO EN UN ÁRBOL

- Puede haber más de un camino con el mismo largo que el diámetro.
- Encontrar el diámetro en un árbol se puede hacer en tiempo $\mathcal{O}(|V| + |E|)$

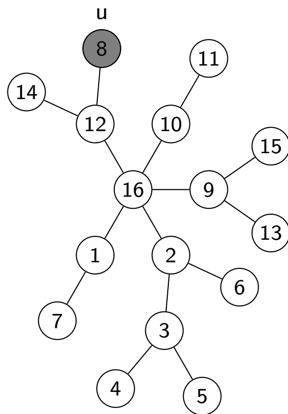


- Puede haber más de un camino con el mismo largo que el diámetro.
- Encontrar el diámetro en un árbol se puede hacer en tiempo $\mathcal{O}(|V| + |E|)$
  - Escogemos un nodo s cualquiera.



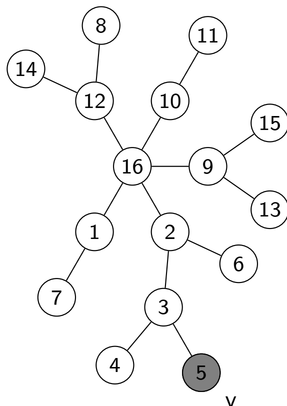
DIÁMETRO EN UN ÁRBOL

- Puede haber más de un camino con el mismo largo que el diámetro.
- Encontrar el diámetro en un árbol se puede hacer en tiempo $\mathcal{O}(|V| + |E|)$
  - Escogemos un nodo s cualquiera.
  - Desde s hacemos bfs para encontrar algún nodo u a distancia máxima.



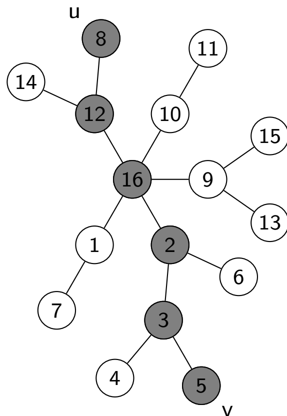
DIÁMETRO EN UN ÁRBOL

- Puede haber más de un camino con el mismo largo que el diámetro.
- Encontrar el diámetro en un árbol se puede hacer en tiempo $\mathcal{O}(|V| + |E|)$
  - Escogemos un nodo s cualquiera.
  - Desde s hacemos bfs para encontrar algún nodo u a distancia máxima.
  - Desde u hacemos bfs para encontrar algún nodo v a distancia máxima.



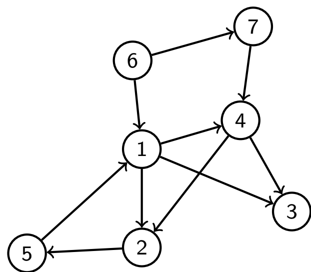
DIÁMETRO EN UN ÁRBOL

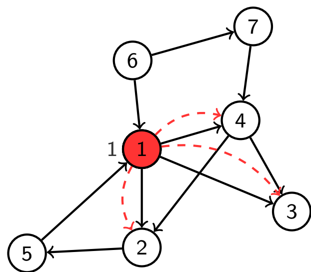
- Puede haber más de un camino con el mismo largo que el diámetro.
- Encontrar el diámetro en un árbol se puede hacer en tiempo $\mathcal{O}(|V| + |E|)$
  - Escogemos un nodo s cualquiera.
  - Desde s hacemos bfs para encontrar algún nodo u a distancia máxima.
  - Desde u hacemos bfs para encontrar algún nodo v a distancia máxima.
  - La distancia entre u y v será el diámetro.



DEPTH FIRST SEARCH (DFS)

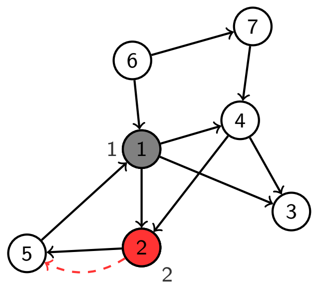
DEPTH FIRST SEARCH (DFS)





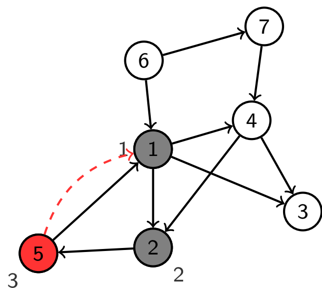
$S : 1$

DEPTH FIRST SEARCH (DFS)

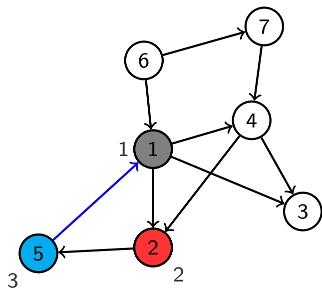


$S : 2, 1$

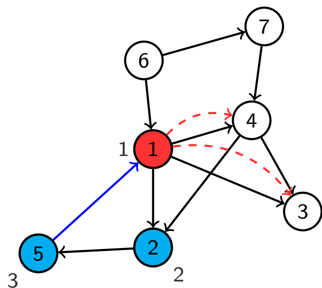
DEPTH FIRST SEARCH (DFS)



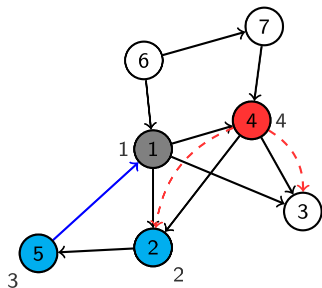
$S : 5, 2, 1$



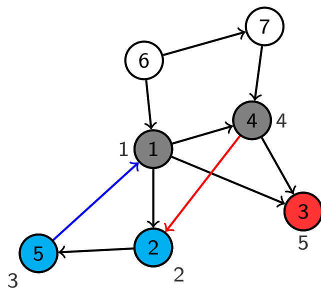
$S : 2, 1$



$S: 1$

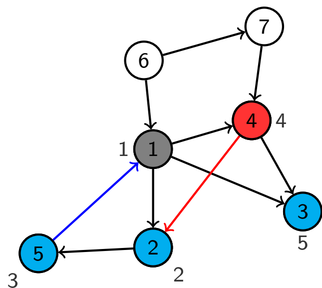


$S : 4, 1$



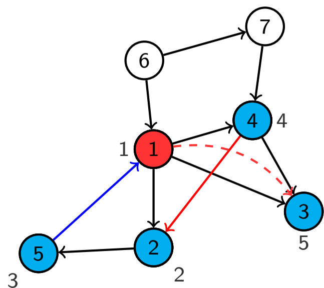
$S : 3, 4, 1$

DEPTH FIRST SEARCH (DFS)

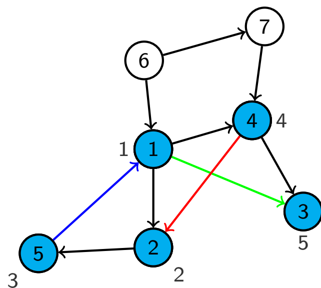


$S : 4, 1$

DEPTH FIRST SEARCH (DFS)

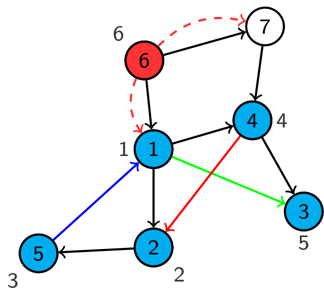


$S : 1$

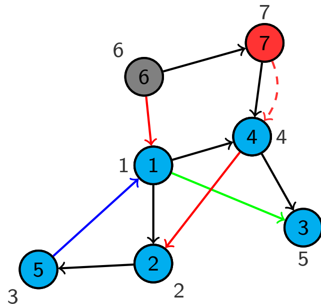


S :

DEPTH FIRST SEARCH (DFS)

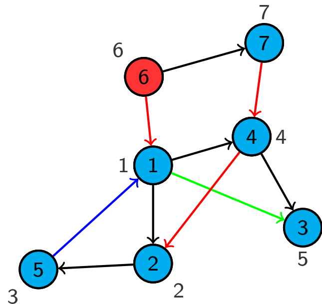


$S : 6$

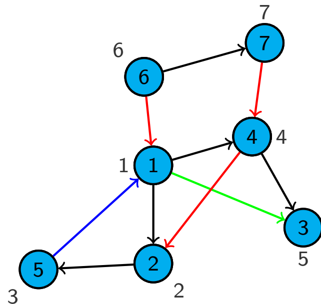


$S : 7, 6$

DEPTH FIRST SEARCH (DFS)

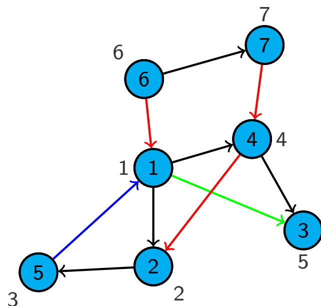


$S : 6$

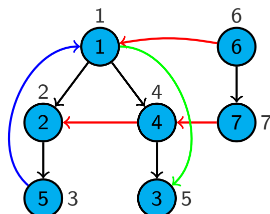


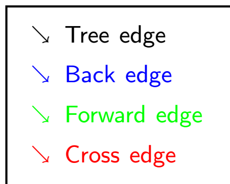
S :

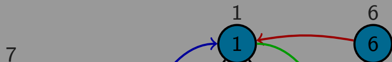
DEPTH FIRST SEARCH (DFS)



S :







Notar que

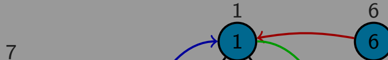
- El DFS corre en tiempo $\mathcal{O}(|V| + |E|)$.

3

S :

Forward edge

Cross edge



3

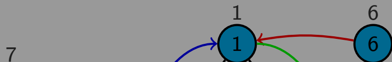
Notar que

- El DFS corre en tiempo $\mathcal{O}(|V| + |E|)$.
- Una *back edge* evidencia la existencia de un ciclo.

S :

↘ Forward edge

↘ Cross edge



Notar que

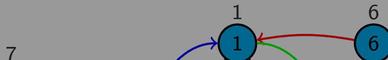
- El DFS corre en tiempo $\mathcal{O}(|V| + |E|)$.
- Una *back edge* evidencia la existencia de un ciclo.
- En un grafo no dirigido solo ocurren *back edges*.

3

S :

Forward edge

Cross edge



3

Notar que

- El DFS corre en tiempo $\mathcal{O}(|V| + |E|)$.
- Una *back edge* evidencia la existencia de un ciclo.
- En un grafo no dirigido solo ocurren *back edges*.
- En un grafo no dirigido hay que tener cuidado de no recorrer la misma arista dos veces (volver hacia el padre o por una *back edge*).

S :

↘ Forward edge

↘ Cross edge

DEPTH FIRST SEARCH (DFS)

```
1  vector<vector<int> > graph;
2  int t;
3  vector<int> entry_time;
4  enum color {UNVISITED, IN_STACK, VISITED};
5  vector<color> state;
6
7  void dfs_visit(int u) {
8    entry_time[u] = t++;
9    state[u] = IN_STACK;
10
11   for (int v : graph[u])
12     if (state[v] == UNVISITED)  // TREE EDGE
13       dfs_visit(v);
14     else if (state[v] == IN_STACK)  // BACK EDGE
15       printf("Back Edge\n");
16     else {
17       if (entry_time[v] > entry_time[u])  // FORWARD EDGE
18         printf("Forward Edge\n");
19       else  // CROSS EDGE
20         printf("Cross Edge\n");
21     }
22   }
23
24   state[u] = VISITED;
25 }
```

DEPTH FIRST SEARCH (DFS)

```
26      void dfs() {
27        t = 0;
28        state.resize(graph.size(), UNVISITED);
29        entry_time.resize(graph.size(), -1);
30
31        for (int u =0; u < graph.size(); ++u)
32          if (state[u] == UNVISITED)
33            dfs_visit(u);
34      }
```