Chapter 15: Graphs

Data Structures and Algorithms with C

Königsberg | Definitions | Matrix and lists | Adjacency multilist | DFS | BFS | Components

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_15_Graphs.html

Learning Outcomes

Define

Use G = (V, E), directed/undirected graphs, adjacency, subgraphs, paths, degree and connectivity.

Prove

Justify that the degree sum = 2e (Theorem 1) and that connectivity needs $e \geq n - 1$.

Implement

Build adjacency matrices, adjacency lists, sequential arrays, orthogonal lists and adjacency multilists in C.

Traverse

Code and trace depth-first (DFS) and breadth-first (BFS) search; analyze them as O(n + e).

Apply

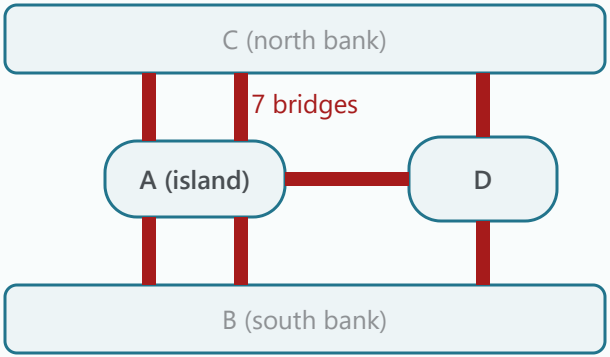
Find connected components, cycles and bipartiteness with DFS/BFS.

Choose

Pick the right representation from the operations the problem needs: “data structure first, then code”.

INTRODUCTION

The Problem: The 7 Bridges of Königsberg

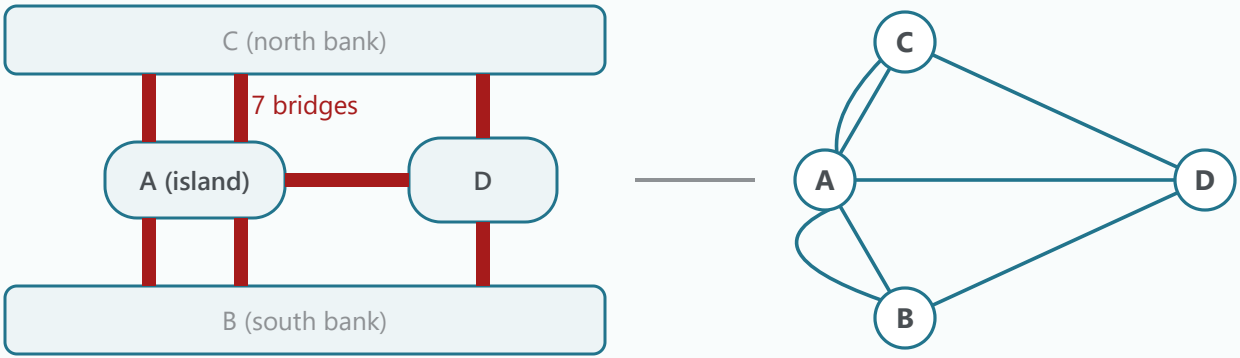


On the Pregel river, two islands (A, D) and two banks (B, C) are joined by 7 bridges. **Can you cross every bridge exactly once and return to the start?**

Step 1 / 3

This is regarded as the birth of graph theory. Euler (pronounced *Oiler*) stated the result as a theorem; its proof is left to **Graph Theory** courses (4th year: “Graph Theory and Combinatorics”).

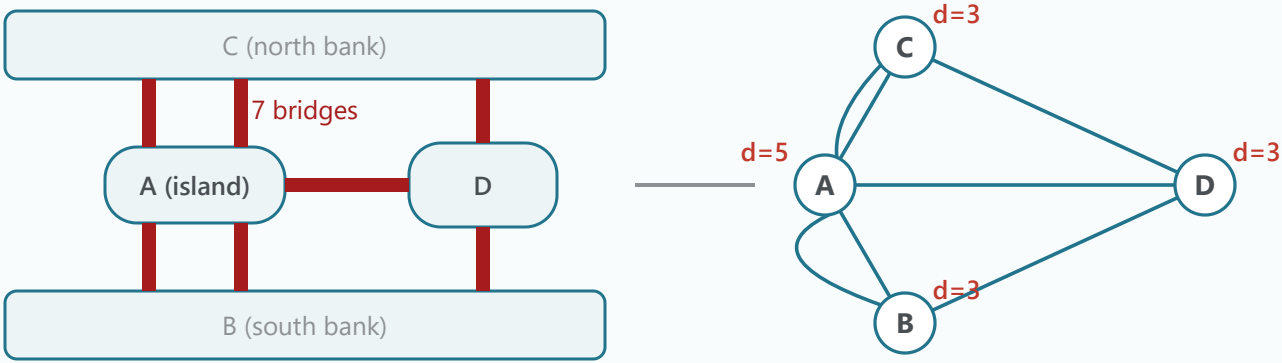
The Problem: The 7 Bridges of Königsberg



Euler (1736) made the land masses **vertices** and the bridges **edges**: what matters is not the shape but the **connections** (notation, not drawing).

This is regarded as the birth of graph theory. Euler (pronounced *Oiler*) stated the result as a theorem; its proof is left to **Graph Theory** courses (4th year: "Graph Theory and Combinatorics").

The Problem: The 7 Bridges of Königsberg



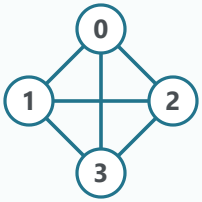
Such a walk exists only if **every vertex has even degree**. Here all degrees are odd (5, 3, 3, 3) → **no such walk**.

Step 3 / 3

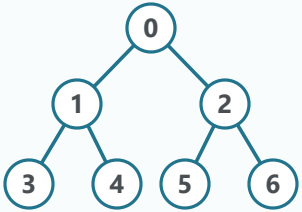
This is regarded as the birth of graph theory. Euler (pronounced *Oiler*) stated the result as a theorem; its proof is left to **Graph Theory** courses (4th year: "Graph Theory and Combinatorics").

Definition: G = (V, E)

A graph G consists of **two finite sets**: a **nonempty** set V of vertices (*nodes*) and a possibly empty set E of edges (*arcs*). Undirected: (v0, v1) ≡ (v1, v0); directed (*digraph*): <v0, v1> ≠ <v1, v0> — <*tail*, *head*>: the arrow leaves the tail and points into the head.



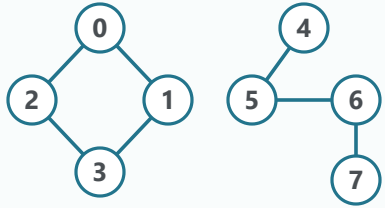
G1: connected (K4)



G2: a tree, no cycles



G3: directed



G4: two components

Graph	V(G)	E(G)
G1	{0, 1, 2, 3}	{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}
G2	{0, …, 6}	{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}
G3	{0, 1, 2}	{<0,1>, <1,0>, <1,2>} — both are listed in a digraph

E is a **subset** of the Cartesian product V × V (it need not contain every pair).

Where Is the Graph Data Structure Used?

Engineering

Analyzing electrical circuits, project planning (activity networks), identifying chemical compounds.

Paths and networks

Shortest paths, routing, many problems on computer networks.

Trees and ordering

Trees are graphs; topological sorting, *finite automata*.

Flow

Flow in transport networks; **max flow**: moving as much oil/water as possible from source to sink.

Spanning trees

Minimum spanning tree: connecting villages with the least cable.

Connectivity

Is the graph connected, how many components, is a network broken?

Instructor: “Where aren't they used! Graphs have a very wide range of uses.”

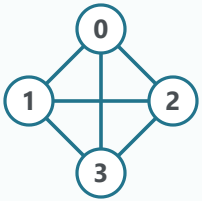
Terminology: Adjacent Vertices, Subgraphs

Adjacent

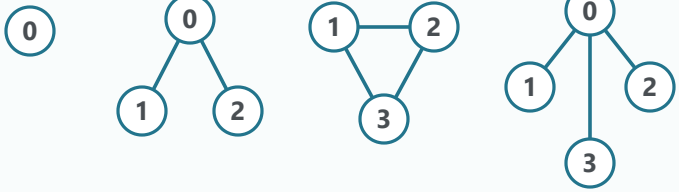
If (v0, v1) is an edge of an undirected graph, v0 and v1 are **adjacent**, and the edge is *incident* on both.

Subgraph

If G′ is a subgraph of G, then V(G′) ⊆ V(G) and E(G′) ⊆ E(G). A single vertex is also a subgraph.



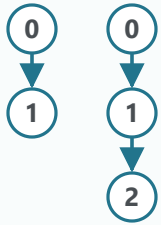
G1



Some of the possible subgraphs of G1



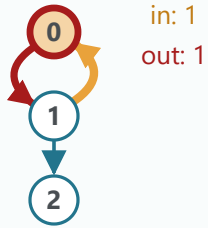
G3



Subgraphs of G3

Path, Connectivity and Degree

- **Path**: a sequence of vertices from v_p to v_q such that (v_p, v_{i1}), (v_{i1}, v_{i2}), …, (v_{in}, v_q) are edges.
- If there is a path from v0 to v1, they are **connected**; if every distinct pair (v_i, v_j) is joined by a path, **G is connected**.
- **Degree**: the number of edges incident on the vertex. In G2 d(0) = 2, in G1 d(1) = 3, in G2 d(4) = 1.
- In a digraph, degree = **in-degree** + **out-degree**. In G3 d(0) = 1 + 1 = 2; d(1) = 1 + 2 = 3; d(2) = 1 + 0 = 1.

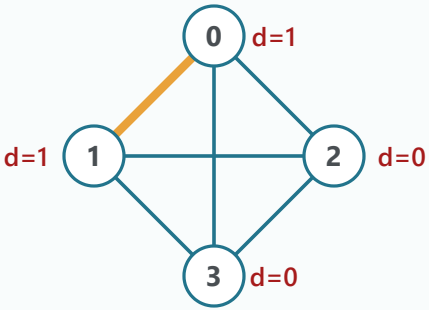


Vertex 0 in G3: d = 1 + 1 = 2

"In G3, d = 1 + 1 = 2" refers to **vertex 0**: a degree is always stated for a particular vertex.

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 1

Σ d = 1 + 1 + 0 + 0 = 2

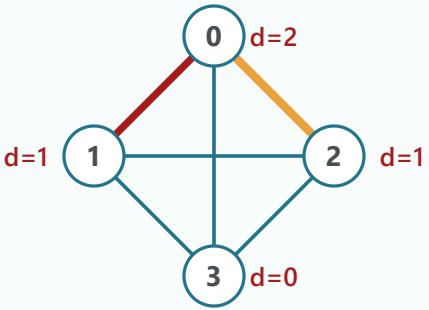
2e = 2

Edge 1 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 2

Σ d = 2 + 1 + 1 + 0 = 4

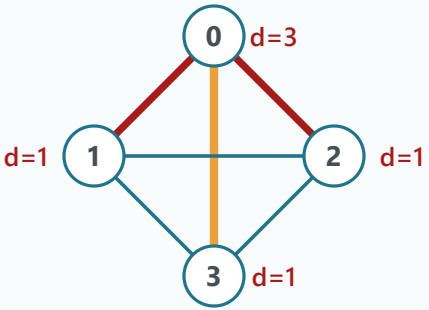
2e = 4

Edge 2 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 3

Σ d = 3 + 1 + 1 + 1 = 6

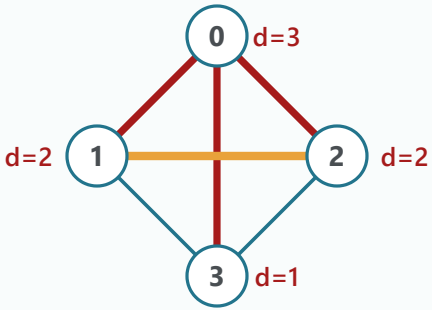
2e = 6

Edge 3 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, $\mathbf{2e = d_0 + d_1 + \ldots + d_{n-1}}$. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 4

Σ d = 3 + 2 + 2 + 1 = 8

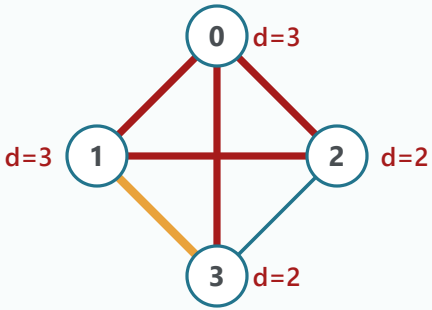
2e = 8

Edge 4 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 5

Σ d = 3 + 3 + 2 + 2 = 10

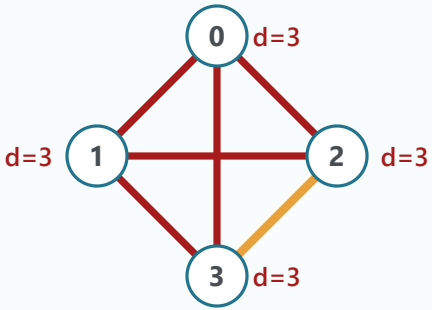
2e = 10

Edge 5 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

Theorem 1: The Sum of Degrees = 2e

In a graph G with n vertices and e edges, **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **Proof:** each edge adds exactly **2** to the degree sum of its two endpoints. It holds for simple and non-simple (multi-edge) graphs. **Simple graph:** at most one edge between two vertices, no *self-loops*.



edges processed: e = 6

Σ d = 3 + 3 + 3 + 3 = 12

2e = 12

Edge 6 added: each endpoint's degree rose by 1 → the sum rose by 2.

Instructor: "'Read the book that will earn you money!' says the teacher; 'No, I won't buy it!' comes the reply. The habit of reading professional books must be built at this age."

How Do We Represent a Graph in Memory?

1. **Adjacency matrix**
2. **Adjacency lists (+ inverse)**
3. **Sequential array representation**
4. **Orthogonal lists (directed)**
5. **Adjacency multilists**
+ **Weighted representation**

Instructor: “First we explain the concept, then its implementation on the computer… Our aim here is not to teach discrete mathematics. It is to teach ‘how do we represent a graph in memory?’”

REPRESENTATIONS

1. Adjacency Matrix

For G with n vertices, an n × n matrix: `a[i][j] = 1` if (i, j) ∈ E. For an undirected graph the matrix is **symmetric**; to save space only the lower or upper triangle may be stored — memory is still **O(n²)**.

G1 — (0,2) ∈ E(G1): a[0][2] = a[2][0] = 1

	0	**1**	**2**	**3**
0	0	1	**1**	1
1	1	0	1	1
2	**1**	1	0	1
3	1	1	1	0

G3 — <0,1> and <1,2> marked; not symmetric

	0	**1**	**2**
0	0	**1**	0
1	1	0	**1**
2	0	0	0

Finding a degree

Count the 1s in the row (or column): **O(n)**. In a digraph the row gives out-degree, the column in-degree; together O(2n). In G3 out-degree d(2) = 0, in-degree d(2) = 1.

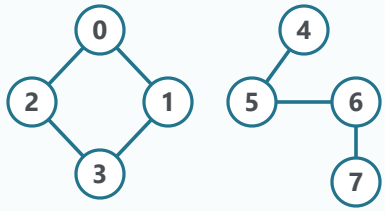
Counting edges

Count all the 1s and divide by 2: **O(n²)**. By symmetry half the matrix is enough; still $O(n^2/2) = O(n^2)$.

G4's Adjacency Matrix — Checking Symmetry

G4; red frame: cells often gotten wrong

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



G4: two components

Common mistake: writing row 1 as `1 0 1 0 …` and row 7 as `… 0 0 1`. The matrix of an undirected graph must be symmetric; these rows break symmetry. Correct: row 1 `1 0 0 1 0 0 0 0` (neighbors 0, 3), row 7 `0 0 0 0 0 0 1 0` (neighbor 6).

The number of components can also be found from the matrix: each DFS scans neighbors in $O(n)$, so $O(n^2)$ in total.

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

`degree(a, 3, 1)` : finding the degree of vertex 1 in G3.

Step 1 / 9

```
▶ int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
4     for (j = 0; j < n; j++)
5         out += a[v][j];       /* row    */
6     for (j = 0; j < n; j++)
7         in += a[j][v];       /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

The counters are set to zero.

Step 2 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
▶       int j, out = 0, in = 0;
4       for (j = 0; j < n; j++)
5           out += a[v][j];       /* row    */
6       for (j = 0; j < n; j++)
7           in += a[j][v];       /* column */
8       return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `out = 0` `in = 0`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Row 1, column 0: a[1][0] = 1 → out = 1.

Step 3 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
▶     for (j = 0; j < n; j++)
▶         out += a[v][j];      /* row    */
6     for (j = 0; j < n; j++)
7         in += a[j][v];      /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 0` `out = 1` `in = 0`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Row 1, column 1: a[1][1] = 0 → out = 1.

Step 4 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
▶     for (j = 0; j < n; j++)
▶         out += a[v][j];      /* row    */
6     for (j = 0; j < n; j++)
7         in += a[j][v];      /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 1` `out = 1` `in = 0`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Row 1, column 2: a[1][2] = 1 → out = 2.

Step 5 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
▶     for (j = 0; j < n; j++)
▶         out += a[v][j];      /* row    */
6     for (j = 0; j < n; j++)
7         in += a[j][v];      /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 2` `out = 2` `in = 0`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Column 1, row 0: a[0][1] = 1 → in = 1.

Step 6 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
4     for (j = 0; j < n; j++)
5         out += a[v][j];       /* row    */
▶     for (j = 0; j < n; j++)
▶         in += a[j][v];       /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 0` `out = 2` `in = 1`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Column 1, row 1: a[1][1] = 0 → in = 1.

Step 7 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
4     for (j = 0; j < n; j++)
5         out += a[v][j];       /* row    */
▶     for (j = 0; j < n; j++)
▶         in += a[j][v];       /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 1` `out = 2` `in = 1`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Column 1, row 2: a[2][1] = 0 → in = 1.

Step 8 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
4     for (j = 0; j < n; j++)
5         out += a[v][j];       /* row    */
▶     for (j = 0; j < n; j++)
▶         in += a[j][v];       /* column */
8     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `j = 2` `out = 2` `in = 1`

CALL STACK

`degree(a, 3, 1)`

`main()`

Code Trace: `degree(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

return 2 + 1 = 3. Both loops run n times → **O(2n) = O(n)**.

Step 9 / 9

```
1 int degree(int a[][MAX_V], int n, int v)
2 {
3     int j, out = 0, in = 0;
4     for (j = 0; j < n; j++)
5         out += a[v][j];       /* row    */
6     for (j = 0; j < n; j++)
7         in += a[j][v];       /* column */
▶     return out + in;
9 }
```

VARIABLES

`v = 1` `n = 3` `out = 2` `in = 1`

CALL STACK

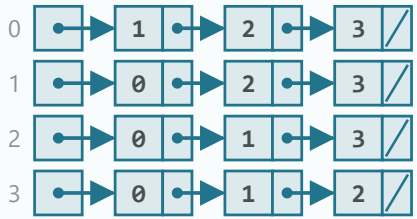
`degree(a, 3, 1)`

`main()`

2. Adjacency Lists

Instead of each matrix row, a **linked list**: `graph[i]` points to the list of i's neighbors. Drawback: an undirected graph uses **2 nodes** per edge.

```
#define MAX_V 50   /* max number of vertices */
typedef struct node *node_ptr;
typedef struct node {
    int vertex;          /* the neighbor's number  */
    node_ptr link;       /* next neighbor in list   */
} node;                  /* don't forget ';' and the name */
node_ptr graph[MAX_V];   /* one list per vertex */
```



G1 represented with adjacency lists

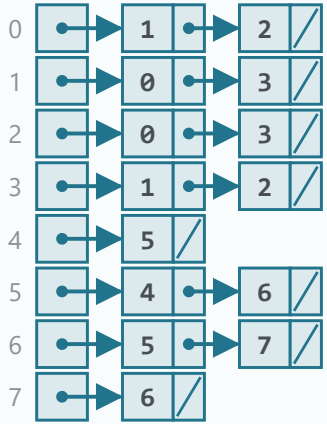
Number of edges

Count all the nodes and divide by 2 (Theorem 1): **O(n + e)**. In graph theory, complexity is usually stated in terms of e.

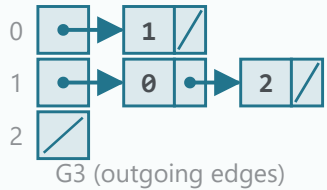
Degree

In an undirected graph, counting the nodes in list i is enough: $O(d_i)$.

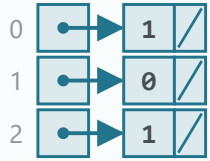
Adjacency Lists: Undirected and Directed Examples



G4's adjacency lists



G3 (outgoing edges)



G3's inverse (incoming edges)



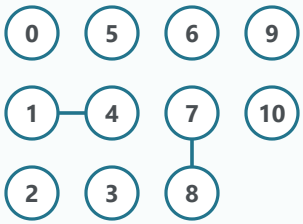
Undirected (G4): the benefit of sorted lists

If the lists are sorted, in each list we can count the nodes whose number is below the head's (e.g. (1,0)) and skip the rest; we visit half the vertices, but it is still **O(n + e)**.

Directed (G3)

<0,1> ≠ <1,0> → every edge must be counted. Out-degree is easy from the lists; in-degree requires scanning every list. The fix: also keep the **inverse (transpose)** lists, built in a single pass.

A Sparse Graph: 11 Vertices, 2 Edges



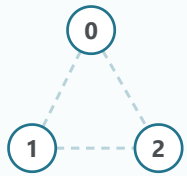
G5

Lists: graph[1] → 4, graph[4] → 1, graph[7] → 8, graph[8] → 7. The other 7 heads are **NULL**.

Total: 11 heads + 2e = 4 nodes. Matrix: 11 × 11 = 121 cells.

11 vertices, 2 edges, **9 components**. The matrix wastes 117 of its 121 cells; the lists hold only 11 heads + 4 nodes. Consider lists for sparse graphs and a matrix for dense ones — but always decide by the **operations to be performed**.

Code Trace: Adding Edges to the Structure



edges added so far are bold



Edge (0,1) → first `add_edge(0, 1)` : memory for a new node is allocated (NULL check!).

Step 1 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

VARIABLES

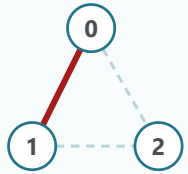
`u = 0` `v = 1` `p = &node(?)` `graph[u] = NULL`

CALL STACK

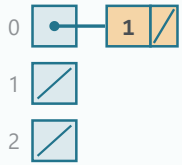
`add_edge(0, 1)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 1` , `p->link = NULL` , `graph[0] = p` : the node went to the **front** of the list.

Step 2 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 0` `v = 1` `p = &node(1)`

`graph[u] = &node(1)`

CALL STACK

`add_edge(0, 1)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold

Edge (0,1) → then `add_edge(1, 0)` : memory for a new node is allocated (NULL check!).

Step 3 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

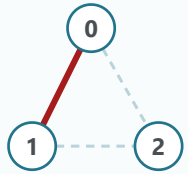
VARIABLES

`u = 1` `v = 0` `p = &node(?)` `graph[u] = NULL`

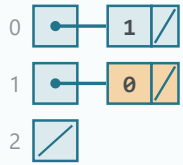
CALL STACK

- `add_edge(1, 0)`
- `main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 0` , `p->link = NULL` , `graph[1] = p` : the node went to the **front** of the list.

Step 4 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 1` `v = 0` `p = &node(0)`

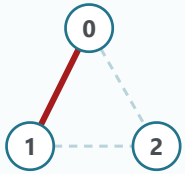
`graph[u] = &node(0)`

CALL STACK

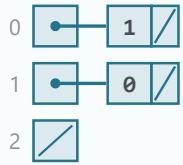
`add_edge(1, 0)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



Edge (0,2) → first `add_edge(0, 2)` : memory for a new node is allocated (NULL check!).

Step 5 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

VARIABLES

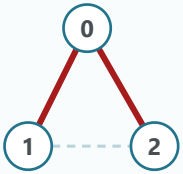
`u = 0` `v = 2` `p = &node(?)` `graph[u] = &node(1)`

CALL STACK

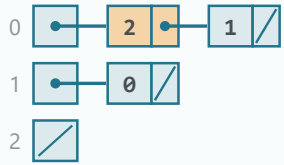
`add_edge(0, 2)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 2` , `p->link = &node(1)` , `graph[0] = p` : the node went to the **front** of the list.

Step 6 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 0` `v = 2` `p = &node(2)`

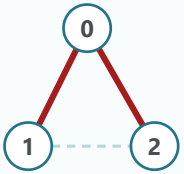
`graph[u] = &node(2)`

CALL STACK

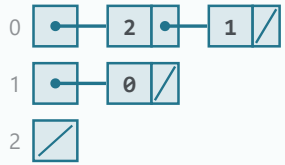
`add_edge(0, 2)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



Edge (0,2) → then `add_edge(2, 0)` : memory for a new node is allocated (NULL check!).

Step 7 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

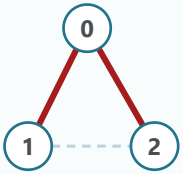
VARIABLES

`u = 2` `v = 0` `p = &node(?)` `graph[u] = NULL`

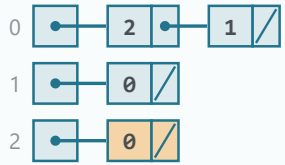
CALL STACK

- `add_edge(2, 0)`
- `main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 0` , `p->link = NULL` , `graph[2] = p` : the node went to the **front** of the list.

Step 8 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 2` `v = 0` `p = &node(0)`

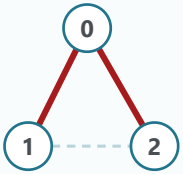
`graph[u] = &node(0)`

CALL STACK

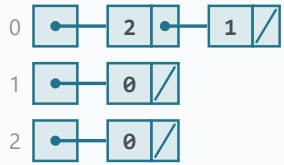
`add_edge(2, 0)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



Edge (1,2) → first `add_edge(1, 2)` : memory for a new node is allocated (NULL check!).

Step 9 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

VARIABLES

`u = 1` `v = 2` `p = &node(?)`

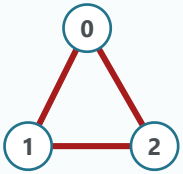
`graph[u] = &node(0)`

CALL STACK

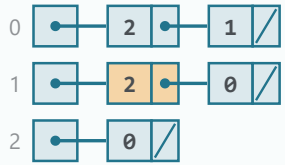
`add_edge(1, 2)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 2` , `p->link = &node(0)` , `graph[1] = p` : the node went to the **front** of the list.

Step 10 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 1` `v = 2` `p = &node(2)`

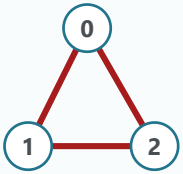
`graph[u] = &node(2)`

CALL STACK

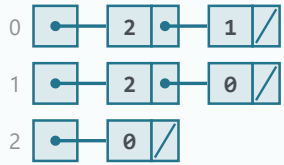
`add_edge(1, 2)`

`main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



Edge (1,2) → then `add_edge(2, 1)` : memory for a new node is allocated (NULL check!).

Step 11 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
▶     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
5     p->vertex = v;
6     p->link = graph[u];   /* at front: O(1) */
7     graph[u] = p;
8 }
```

VARIABLES

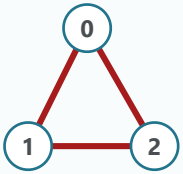
`u = 2` `v = 1` `p = &node(?)`

`graph[u] = &node(0)`

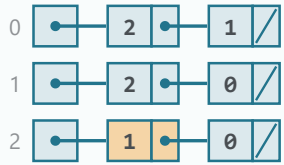
CALL STACK

- `add_edge(2, 1)`
- `main()`

Code Trace: Adding Edges to the Structure



edges added so far are bold



`p->vertex = 1` , `p->link = &node(0)` , `graph[2] = p` : the node went to the **front** of the list.

Step 12 / 12

```
1 void add_edge(int u, int v)   /* u -> v */
2 {
3     node_ptr p = malloc(sizeof(node));
4     if (p == NULL) exit(1);       /* no memory */
▶     p->vertex = v;
▶     p->link = graph[u];   /* at front: O(1) */
▶     graph[u] = p;
8 }
```

VARIABLES

`u = 2` `v = 1` `p = &node(1)`

`graph[u] = &node(1)`

CALL STACK

`add_edge(2, 1)`

`main()`

Who Builds the Graph?

> **Instructor:** “Edges come in, and they must be added to the structure. I looked and drew it; a program can't just look and draw! The algorithm has to do the work.”

Insert at the front

Each edge is placed in O(1); the list ends up in the **reverse** of arrival order. Total build time O(n + e).

Insert at the end

Keeps the order, but each time you must walk to the NULL (O(d)); or keep a `last` pointer for each list.

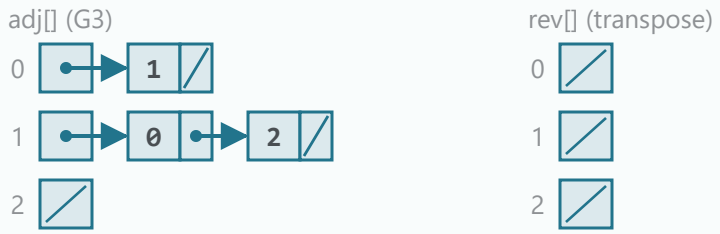
Sorted insertion

Keeping each list sorted by vertex number: edge counting and traversal order become predictable; insertion is O(d).

Freeing

At the end of the program, every `graph[i]` list must be walked and `free` d.

Code Trace: `transpose()`



First the `rev[]` heads are set to NULL.

Step 1 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
▶      for (i = 0; i < n; i++) rev[i] = NULL;
7      for (i = 0; i < n; i++)
8          for (p = adj[i]; p; p = p->link)
9              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

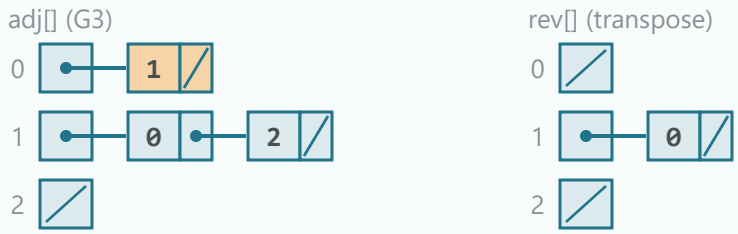
`n = 3` `i = –` `p = NULL`

CALL STACK

`transpose(adj, rev, 3)`

`main()`

Code Trace: `transpose()`



Edge <0,1>: 0 is added to list `rev[1]` (<1,0>).

Step 2 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
6      for (i = 0; i < n; i++) rev[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = adj[i]; p; p = p->link)
▶              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

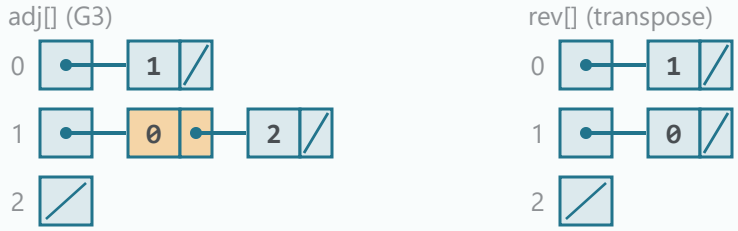
`n = 3` `i = 0` `p = &node(1)`

CALL STACK

`transpose(adj, rev, 3)`

`main()`

Code Trace: `transpose()`



Edge <1,0>: 1 is added to list `rev[0]` (<0,1>).

Step 3 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
6      for (i = 0; i < n; i++) rev[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = adj[i]; p; p = p->link)
▶              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

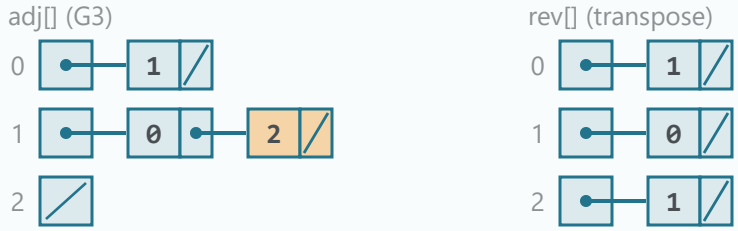
`n = 3` `i = 1` `p = &node(0)`

CALL STACK

`transpose(adj, rev, 3)`

`main()`

Code Trace: `transpose()`



Edge <1,2>: 1 is added to list `rev[2]` (<2,1>).

Step 4 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
6      for (i = 0; i < n; i++) rev[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = adj[i]; p; p = p->link)
▶              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

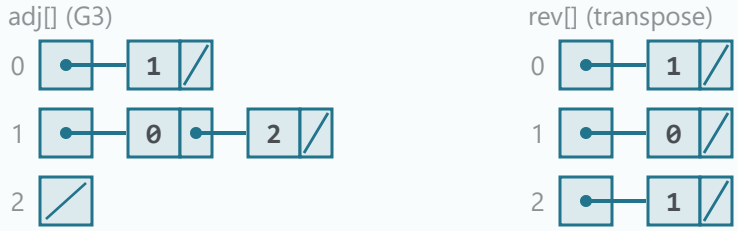
`n = 3` `i = 1` `p = &node(2)`

CALL STACK

`transpose(adj, rev, 3)`

`main()`

Code Trace: `transpose()`



`adj[2]` is empty: no edges leave 2.

Step 5 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
6      for (i = 0; i < n; i++) rev[i] = NULL;
▶      for (i = 0; i < n; i++)
▶          for (p = adj[i]; p; p = p->link)
9              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

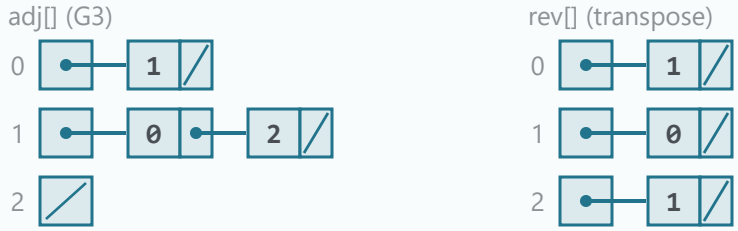
`n = 3` `i = 2` `p = NULL`

CALL STACK

`transpose(adj, rev, 3)`

`main()`

Code Trace: `transpose()`



Each node was looked at once: the transpose was built in **O(n + e)**. Now in-degree is also found in O(d).

Step 6 / 6

```
1  void transpose(node_ptr adj[], node_ptr rev[],
2                 int n)
3  {
4      int i;
5      node_ptr p;
6      for (i = 0; i < n; i++) rev[i] = NULL;
▶      for (i = 0; i < n; i++)
8          for (p = adj[i]; p; p = p->link)
9              insert(rev, p->vertex, i); /* <j,i> */
10 }
```

VARIABLES

`n = 3` `i = 3` `p = NULL`

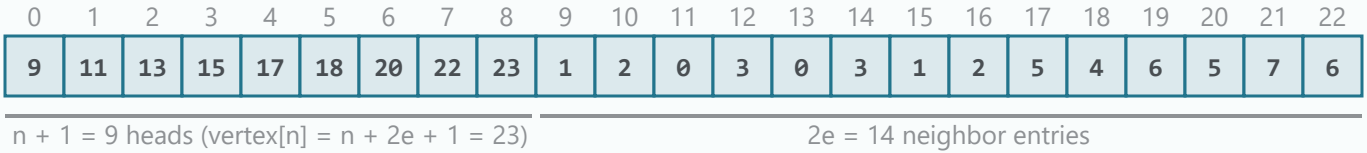
CALL STACK

`transpose(adj, rev, 3)`

`main()`

3. Sequential Representation with an Array

Undirected, n vertices, e edges: a single array `int vertex[n + 2*e + 1]`. The first n + 1 cells are starting indices; the neighbors of i are in `vertex[vertex[i]] … vertex[vertex[i+1] − 1]`.



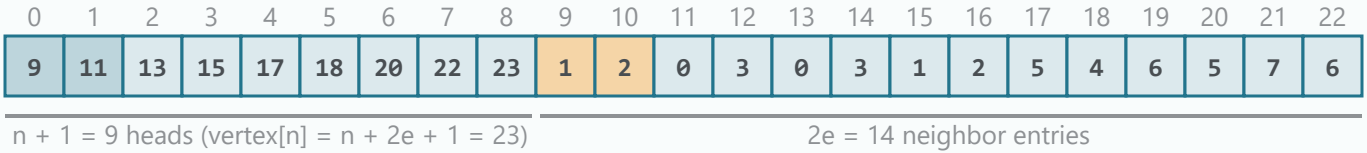
The array for G4. Each edge is still stored twice, but there are **no pointers**.

Step 1 / 4

Instructor: “Storing is one issue; the efficiency of the algorithms that will run on that data is another… This structure is not efficient for many insertions and deletions. The computers in a network, say, are fixed; if the problem is to store them, this structure may suit.”

3. Sequential Representation with an Array

Undirected, n vertices, e edges: a single array `int vertex[n + 2*e + 1]`. The first n + 1 cells are starting indices; the neighbors of i are in `vertex[vertex[i]] … vertex[vertex[i+1] − 1]`.



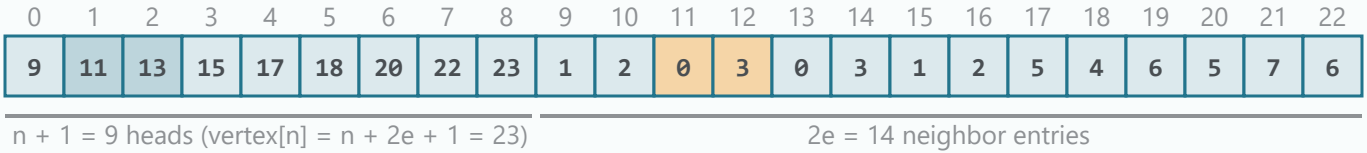
neighbors of 0: vertex[0] = 9 … vertex[1] = 11 (excl.) → 1, 2

Ex. 1: from [0] = 9 (incl.) to [1] = 11 (excl.): indices 9 and 10 → neighbors **1, 2**.

Instructor: “Storing is one issue; the efficiency of the algorithms that will run on that data is another… This structure is not efficient for many insertions and deletions. The computers in a network, say, are fixed; if the problem is to store them, this structure may suit.”

3. Sequential Representation with an Array

Undirected, n vertices, e edges: a single array `int vertex[n + 2*e + 1]`. The first n + 1 cells are starting indices; the neighbors of i are in `vertex[vertex[i]] … vertex[vertex[i+1] − 1]`.



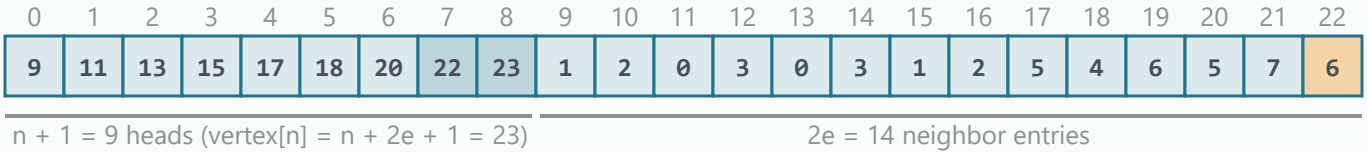
neighbors of 1: vertex[1] = 11 … vertex[2] = 13 (excl.) → 0, 3

Ex. 2: [1] = 11, [2] = 13 → indices 11, 12 → neighbors **0, 3**.

Instructor: “Storing is one issue; the efficiency of the algorithms that will run on that data is another… This structure is not efficient for many insertions and deletions. The computers in a network, say, are fixed; if the problem is to store them, this structure may suit.”

3. Sequential Representation with an Array

Undirected, n vertices, e edges: a single array `int vertex[n + 2*e + 1]`. The first n + 1 cells are starting indices; the neighbors of i are in `vertex[vertex[i]] … vertex[vertex[i+1] − 1]`.



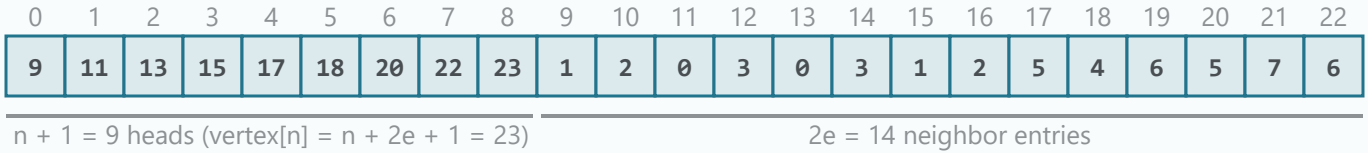
neighbors of 7: vertex[7] = 22 … vertex[8] = 23 (excl.) → 6

Ex. 3: [7] = 22, [8] = 23 → only index 22 → a single neighbor, **6**.

Step 4 / 4

Instructor: “Storing is one issue; the efficiency of the algorithms that will run on that data is another… This structure is not efficient for many insertions and deletions. The computers in a network, say, are fixed; if the problem is to store them, this structure may suit.”

Code Trace: `print_adj(vertex, 5)`



`print_adj(vertex, 5)` : the neighbors of vertex 5.

Step 1 / 4

```
▶ void print_adj(const int vertex[], int i)
2 {
3     int k;
4     for (k = vertex[i]; k < vertex[i + 1]; k++)
5         printf("%d ", vertex[k]);
6     printf("\n");
7 }
```

VARIABLES

`i = 5`

CALL STACK

`print_adj(vertex, 5)`

`main()`

OUTPUT

Code Trace: `print_adj(vertex, 5)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

k = 18

k = 18 < vertex[6] = 20 → `vertex[18] = 4` is printed.

Step 2 / 4

```
1 void print_adj(const int vertex[], int i)
2 {
3     int k;
▶     for (k = vertex[i]; k < vertex[i + 1]; k++)
▶         printf("%d ", vertex[k]);
6     printf("\n");
7 }
```

VARIABLES

`i = 5` `k = 18` `vertex[k] = 4`

CALL STACK

`print_adj(vertex, 5)`

`main()`

OUTPUT

```
4
```

Code Trace: `print_adj(vertex, 5)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

k = 19

k = 19 < vertex[6] = 20 → `vertex[19] = 6` is printed.

Step 3 / 4

```
1 void print_adj(const int vertex[], int i)
2 {
3     int k;
▶     for (k = vertex[i]; k < vertex[i + 1]; k++)
▶         printf("%d ", vertex[k]);
6     printf("\n");
7 }
```

VARIABLES

`i = 5` `k = 19` `vertex[k] = 6`

CALL STACK

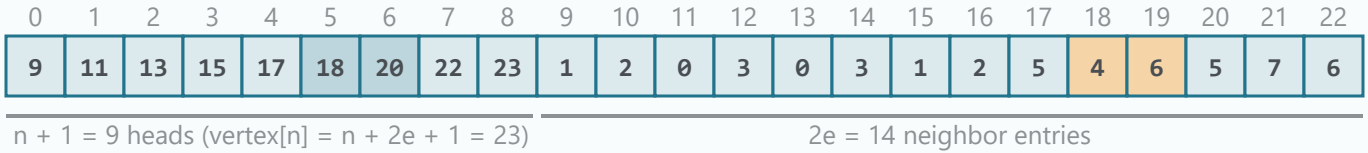
`print_adj(vertex, 5)`

`main()`

OUTPUT

```
4 6
```

Code Trace: `print_adj(vertex, 5)`



neighbors of 5: vertex[5] = 18 … vertex[6] = 20 (excl.) → 4, 6

k = 20 = vertex[6] → the loop ends. Neighbor count = vertex[6] − vertex[5] = the degree: **O(d)**.

Step 4 / 4

```
1 void print_adj(const int vertex[], int i)
2 {
3     int k;
▶     for (k = vertex[i]; k < vertex[i + 1]; k++)
5         printf("%d ", vertex[k]);
▶     printf("\n");
7 }
```

VARIABLES

`i = 5` `k = 20`

CALL STACK

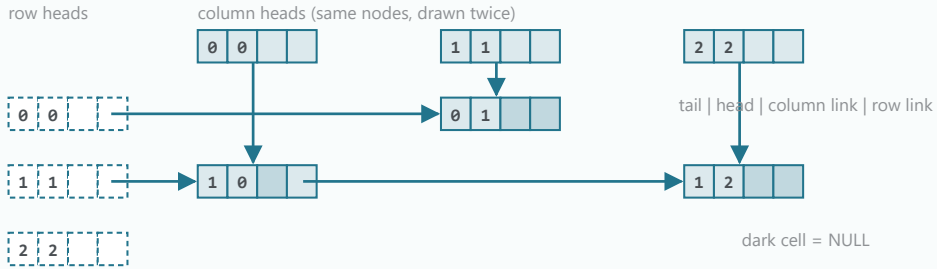
`print_adj(vertex, 5)`

`main()`

OUTPUT

```
4 6
```

4. Orthogonal Representation of Digraphs



G3: each edge node sits in one **row** list (out) and one **column** list (in) at the same time

Two links

Column link, for the head: where we come into this vertex from. **Row link, for the tail:** where we go from this vertex. **One node** per edge is enough.

Like a sparse matrix

It resembles the linked sparse matrix structure; it can be made circular, doubly linked, with head nodes. “Using 2e nodes bothered us, so we found this one.”

Instructor: “Never trust the first solution that comes to mind, or its code. If the underlying data structure were something else, could I solve it more efficiently?”

5. Adjacency Multilists

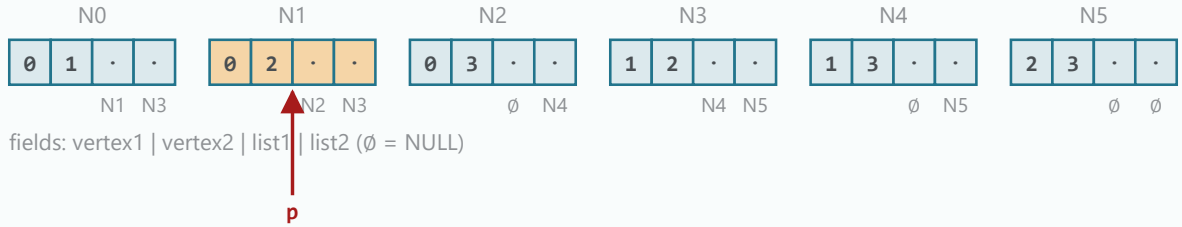
```
typedef struct edge *edge_ptr;
typedef struct edge {
    short marked;         /* processed? */
    int vertex1, vertex2;  /* ends    */
    edge_ptr list1;     /* next for vertex1 */
    edge_ptr list2;     /* next for vertex2 */
} edge;
edge_ptr graph[MAX_V];
```

The edge is **the basic unit**: there is **one node** for (v_i, v_j), but it is on both v_i's and v_j's lists. `marked` stops a processed edge from being handled again. To carry data on a vertex, add a field such as `char data`.

Node	vertex1	vertex2	list1	list2
N0	0	1	N1	N3
N1	0	2	N2	N3
N2	0	3	NULL	N4
N3	1	2	N4	N5
N4	1	3	NULL	N5
N5	2	3	NULL	NULL

Heads for G1: graph[0] = N0, graph[1] = N0, graph[2] = N1, graph[3] = N2. A vertex with no edges has a NULL head.

Code Trace: Walking over the Neighbors of 2



`p = graph[2]` = N1: the first edge of 2.

Step 1 / 8

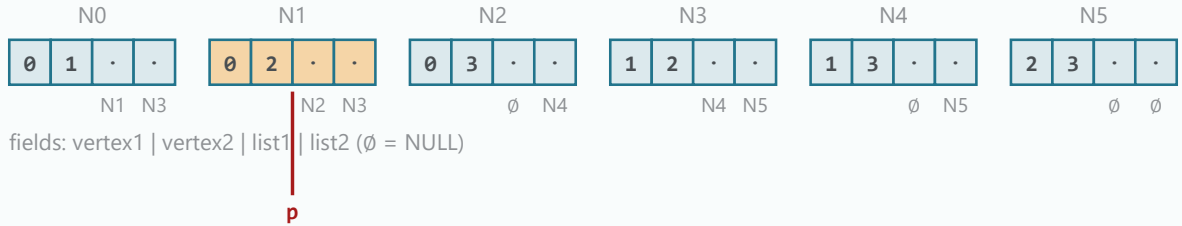
```
1  void print_adj(int v)
2  {
▶      edge_ptr p = graph[v];
4      while (p) {
5          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
8              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

VARIABLES

`v = 2` `p = &N1`

OUTPUT

Code Trace: Walking over the Neighbors of 2



N1 = (0,2): v is **vertex2** → print the other end, 0.

Step 2 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
▶          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
▶              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

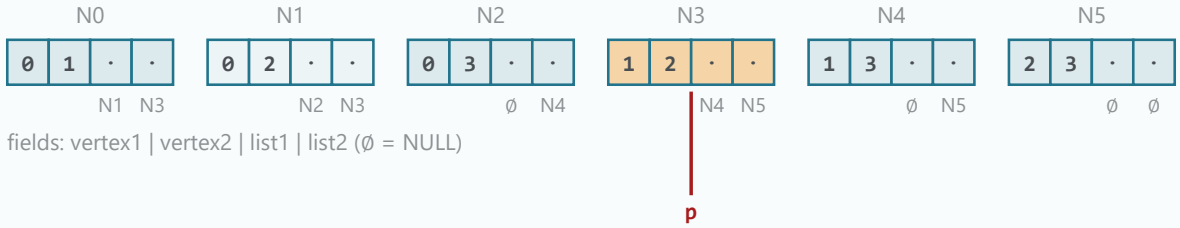
VARIABLES

`v = 2` `p = &N1`

OUTPUT

```
0
```

Code Trace: Walking over the Neighbors of 2



Follow the link: `p = p→list2` = N3.

Step 3 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
5          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
▶              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

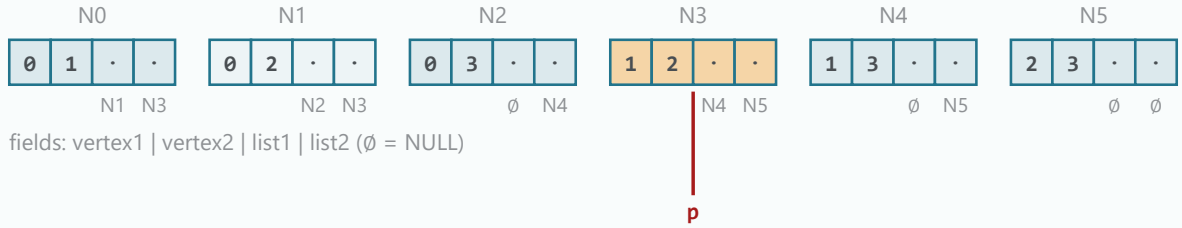
VARIABLES

`v = 2` `p = &N3`

OUTPUT

```
0
```

Code Trace: Walking over the Neighbors of 2



N3 = (1,2): v is **vertex2** → print the other end, 1.

Step 4 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
▶          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
▶              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

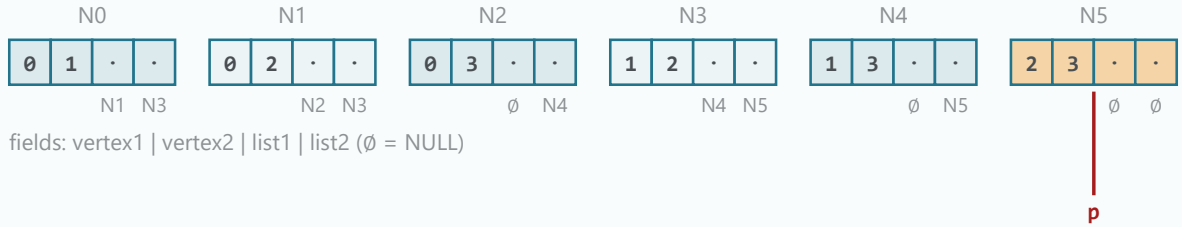
VARIABLES

v = **2** p = **&N3**

OUTPUT

```
0 1
```

Code Trace: Walking over the Neighbors of 2



Follow the link: `p = p→list2` = N5.

Step 5 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
5          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
▶              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

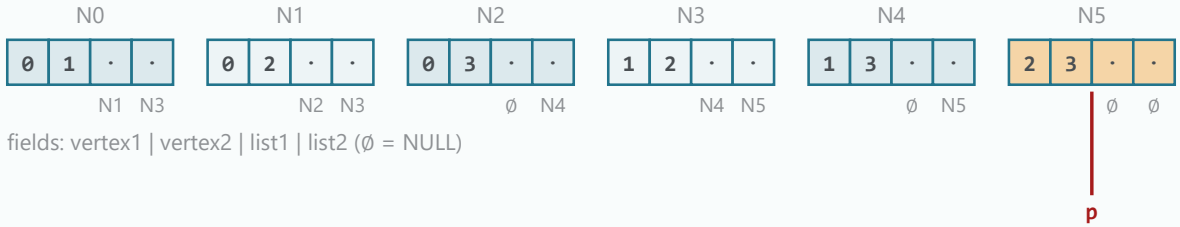
VARIABLES

`v = 2` `p = &N5`

OUTPUT

```
0 1
```

Code Trace: Walking over the Neighbors of 2



N5 = (2,3): v is **vertex1** → print the other end, 3.

Step 6 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
▶          if (p->vertex1 == v) {      /* v: vertex1 */
▶              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
8              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

VARIABLES

v = **2** p = **&N5**

OUTPUT

```
0 1 3
```

Code Trace: Walking over the Neighbors of 2



Follow the link: `p = p→list1` = NULL.

Step 7 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
4      while (p) {
5          if (p->vertex1 == v) {      /* v: vertex1 */
▶              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
8              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

VARIABLES

`v = 2` `p = NULL`

OUTPUT

```
0 1 3
```

Code Trace: Walking over the Neighbors of 2



NULL: no more matching nodes. The neighbors of 2 are 0, 1, 3 — each edge was stored only **once**.

Step 8 / 8

```
1  void print_adj(int v)
2  {
3      edge_ptr p = graph[v];
▶      while (p) {
5          if (p->vertex1 == v) {      /* v: vertex1 */
6              printf("%d ", p->vertex2);  p = p->list1;
7          } else {                    /* v: vertex2 */
8              printf("%d ", p->vertex1);  p = p->list2;
9          }
10     }
11 }
```

VARIABLES

v = **2** p = **NULL**

OUTPUT

```
0 1 3
```

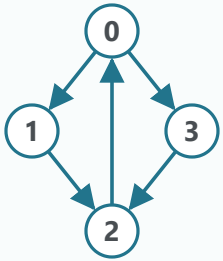
Adding an Edge to an Adjacency Multilist

- **Traversal:** follow list1 if v is vertex1, list2 if v is vertex2. Stop at NULL. "Any more? Follow the link!"
- **Insertion:** you could walk the list to NULL and append; but **inserting at the front** saves the walk: O(1).
- An undirected graph needs one **head array**; a digraph keeps a **tail array** as well.
- The gain: only **e** nodes instead of 2e.

```
void ml_add_edge(int u, int v)
{
    edge_ptr p = malloc(sizeof(edge));
    if (p == NULL) exit(1);
    p->marked = 0;
    p->vertex1 = u;  p->vertex2 = v;
    p->list1 = graph[u];  /* u's list */
    p->list2 = graph[v];  /* v's list */
    graph[u] = graph[v] = p;  /* front of both */
}
```

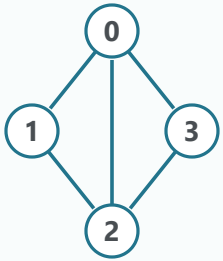
Instructor: "If the edges come in this order and you are told 'place each in O(1)', insert at the front of the list! I give no credit to anyone who draws it this way (appending to the end), because I can see they appended."

Exercise: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>



For the digraph with this edge list, give the **adjacency matrix**, the **adjacency lists** (and their inverse) and the **orthogonal lists**.

Exercise: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>

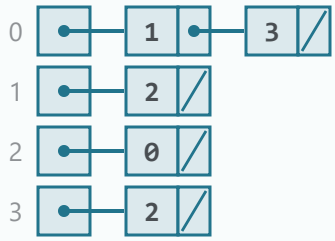


row = tail, column = head

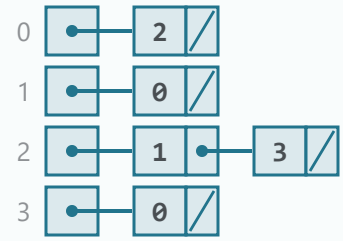
	0	1	2	3
0	0	**1**	0	**1**
1	0	0	**1**	0
2	**1**	0	0	0
3	0	0	**1**	0

Matrix: a[i][j] = 1 if <i,j> exists. Row sums are out-degrees (2, 1, 1, 1), column sums in-degrees (1, 1, 2, 1).

Exercise: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>



outgoing (adjacency lists)



incoming (transpose of the lists)

Lists and inverse: 2e = 10 nodes in all.

Exercise: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>

Node	tail	head	tail link	head link
N1	0	1	N2	NULL
N2	0	3	NULL	NULL
N3	1	2	NULL	N5
N4	2	0	NULL	NULL
N5	3	2	NULL	NULL

Vertex	tail (out)	head (in)
0	N1	N4
1	N3	N1
2	N4	N3
3	N5	N2

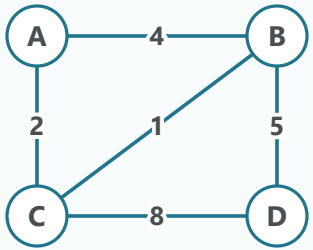
Orthogonal: only e = 5 nodes (N1…N5). Each node is on its tail's out-list and its head's in-list.

Takeaway: First Decide How the Data Is Stored

Representation	Memory	(u,v) present?	u's neighbors	Insert/delete
Adjacency matrix	$O(n^2)$	O(1)	O(n)	O(1)
Adjacency lists	$O(n + 2e)$	$O(d_u)$	$O(d_u)$	front O(1) / search O(d)
Sequential array	$n + 2e + 1$	$O(d_u)$	$O(d_u)$	expensive (shifting)
Orthogonal (directed)	O(n + e)	O(d)	in and out O(d)	O(1) at front
Multilist	O(n + e)	$O(d_u)$	$O(d_u)$	O(1) at front

Instructor: "The first solution that comes to mind may not be a good one. Maybe an n^3 algorithm can be brought down to n. That is where you must design the data structure well. Before writing any code, I think about how to store the data. Storing data is not that easy!"

Weighted Graph Representation



weight = distance, cost, time…

cost matrix: ∞ when there is no edge (a large constant in C)

	A	B	C	D
A	0	4	2	∞
B	4	0	1	5
C	2	1	0	8
D	∞	5	8	0

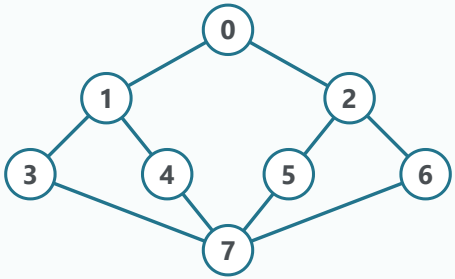
```c
#define INF 1000000   /* not INT_MAX: sums overflow */
int cost[MAX_V][MAX_V];

typedef struct wnode *wnode_ptr;
typedef struct wnode {
    int vertex;
    int weight;        /* cost of the edge */
    wnode_ptr link;
} wnode;
wnode_ptr wgraph[MAX_V];
```

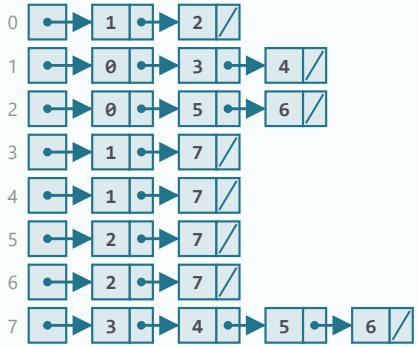
- In the matrix, a **weight** instead of 1, and **∞** when there is no edge.
- In the lists, a `weight` field is added to the node.
- Using `INT_MAX` for ∞ overflows in the sum `d + w`; choose a safe large constant.
- MST and shortest-path algorithms use this representation.

Basic Graph Operations: Traversal

- Given an undirected G = (V, E) and a vertex v, visit **every vertex** reachable from v (visit = print its data field).
- **Depth-first search** (DFS) resembles **preorder** traversal of a tree.
- **Breadth-first search** (BFS) resembles **level-order** traversal of a tree.
- Both use a graph stored as **adjacency lists**; the lists are sorted.



Sample graph



graph[] adjacency lists

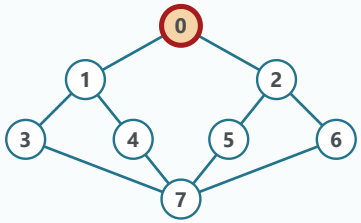
Depth-First Search (DFS)

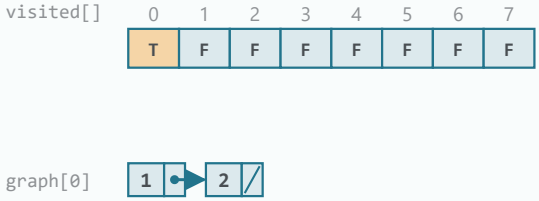
- Visit v; pick an **unvisited** w from v's list and continue the search **from w**.
- A **stack** is needed to remember our place in the list; when u has nowhere left to go, pop a vertex from the stack and continue.
- A **recursive routine** makes this seemingly complex scheme easy: the stack is the call stack.
- The global array `visited[]` must start out FALSE.

```c
#include <stdio.h>
#include <stdbool.h>
#define TRUE  true
#define FALSE false
bool visited[MAX_V];   /* don't forget ';' */
```

```c
void dfs(int v)   /* depth-first search */
{
    node_ptr w;
    visited[v] = TRUE;    /* visited */
    printf("%d ", v);
    for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

Code Trace: `dfs(0)`





`dfs(0)` is called: **0** is marked and printed.

Step 1 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 0` `w = –`

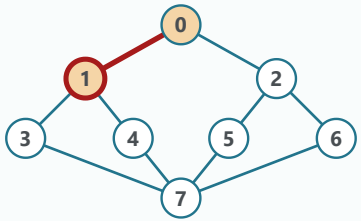
CALL STACK

`dfs(0)` `main()`

OUTPUT

```
0
```

Code Trace: `dfs(0)`



visited[]

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

graph[1] → 0 → 3 → 4 /

0 → 1: 1 is unvisited → `dfs(1)`. 1 is marked and printed.

Step 2 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 1` `w = –`

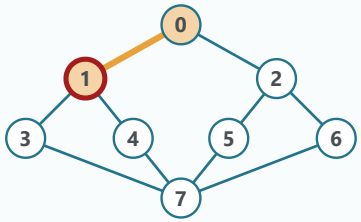
CALL STACK

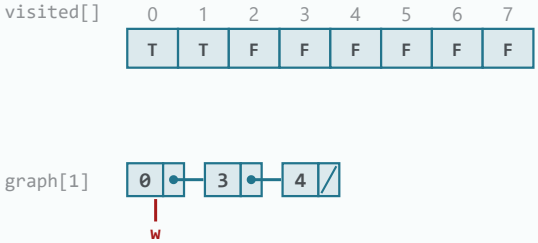
`dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1
```

Code Trace: `dfs(0)`





w → 0: `visited[0]` is TRUE; skip.

Step 3 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 1` `w = &node(0)`

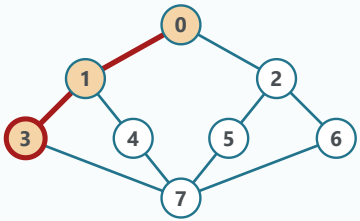
CALL STACK

`dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1
```

Code Trace: `dfs(0)`



visited[]

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

graph[3] → 1 → 7

1 → 3: 3 is unvisited → `dfs(3)`. 3 is marked and printed.

Step 4 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 3` `w = –`

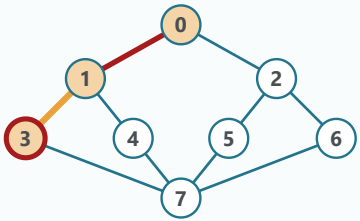
CALL STACK

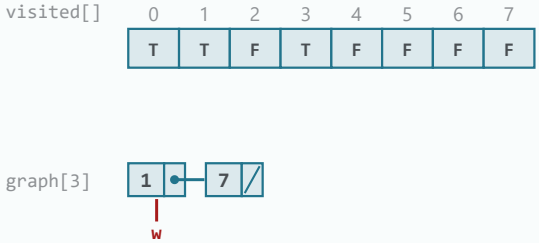
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3
```

Code Trace: `dfs(0)`





w → 1: `visited[1]` is TRUE; skip.

Step 5 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 3` `w = &node(1)`

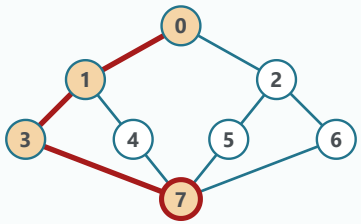
CALL STACK

`dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3
```

Code Trace: `dfs(0)`



visited[]

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

graph[7]



3 → 7: 7 is unvisited → `dfs(7)`. 7 is marked and printed.

Step 6 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 7` `w = –`

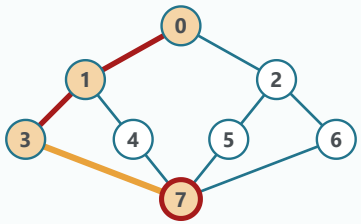
CALL STACK

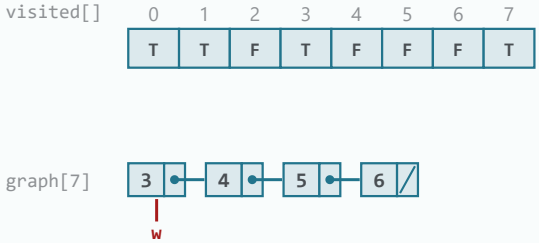
`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7
```

Code Trace: `dfs(0)`





w → 3: `visited[3]` is TRUE; skip.

Step 7 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 7` `w = &node(3)`

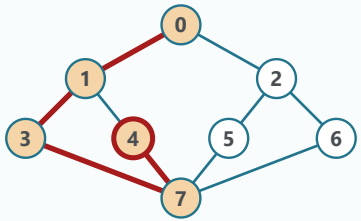
CALL STACK

`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7
```

Code Trace: `dfs(0)`



visited[]

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

graph[4]



7 → 4: 4 is unvisited → `dfs(4)`. 4 is marked and printed.

Step 8 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 4` `w = –`

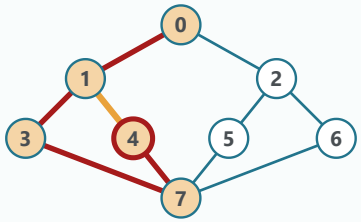
CALL STACK

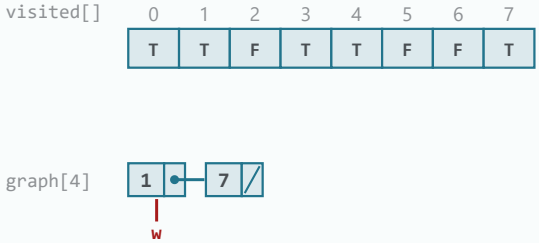
`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4
```

Code Trace: `dfs(0)`





w → 1: `visited[1]` is TRUE; skip.

Step 9 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 4` `w = &node(1)`

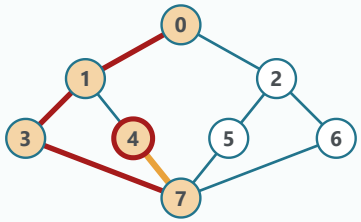
CALL STACK

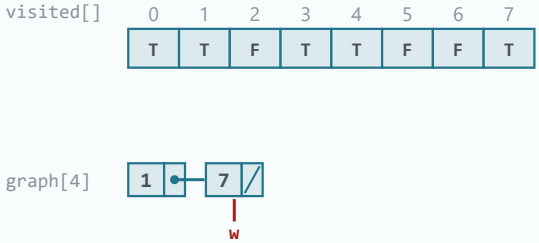
`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4
```

Code Trace: `dfs(0)`





w → 7: `visited[7]` is TRUE; skip.

Step 10 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 4` `w = &node(7)`

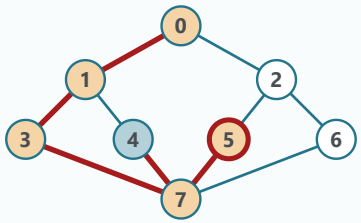
CALL STACK

`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4
```

Code Trace: `dfs(0)`



visited[]

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

graph[5] → 2 → 7

(`dfs(4)` is done.) 7 → 5: 5 is unvisited → `dfs(5)`. 5 is marked and printed.

Step 11 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 5` `w = –`

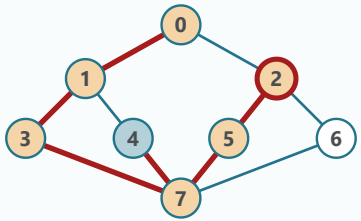
CALL STACK

`dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4 5
```

Code Trace: `dfs(0)`



visited[]

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

graph[2] → 0 → 5 → 6 /

5 → 2: 2 is unvisited → `dfs(2)`. 2 is marked and printed.

Step 12 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 2` `w = –`

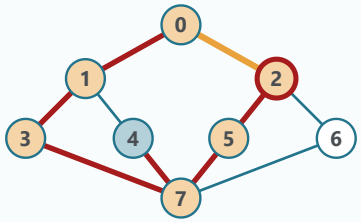
CALL STACK

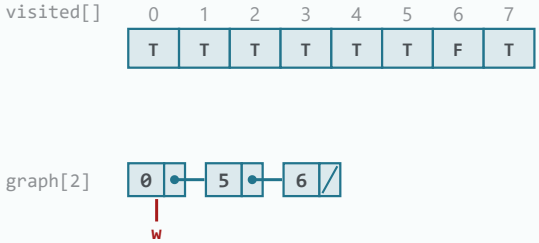
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4 5 2
```

Code Trace: `dfs(0)`





w → 0: `visited[0]` is TRUE; skip.

Step 13 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 2` `w = &node(0)`

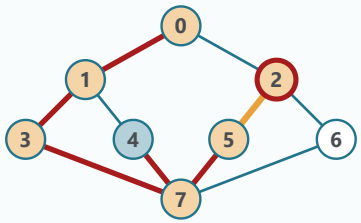
CALL STACK

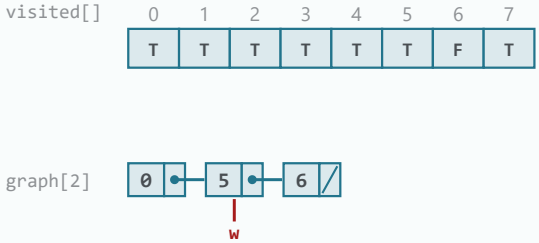
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4 5 2
```

Code Trace: `dfs(0)`





w → 5: `visited[5]` is TRUE; skip.

Step 14 / 14

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 2` `w = &node(5)`

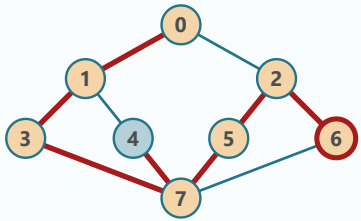
CALL STACK

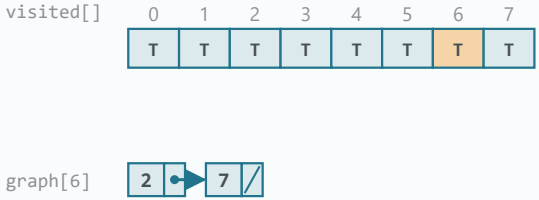
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

```
0 1 3 7 4 5 2
```

Code Trace: `dfs(0)` (Cont.)





2 → 6: 6 is unvisited → `dfs(6)`. 6 is marked and printed.

Step 1 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
▶     visited[v] = TRUE;    /* visited */
▶     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 6` `w = –`

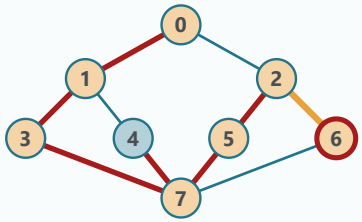
CALL STACK

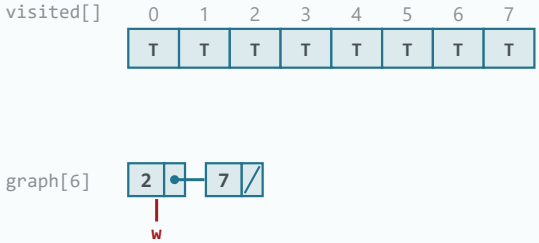
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)`
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





w → 2: `visited[2]` is TRUE; skip.

Step 2 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 6` `w = &node(2)`

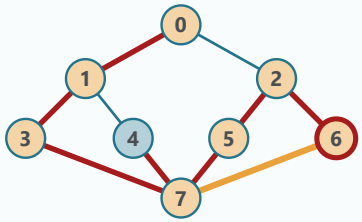
CALL STACK

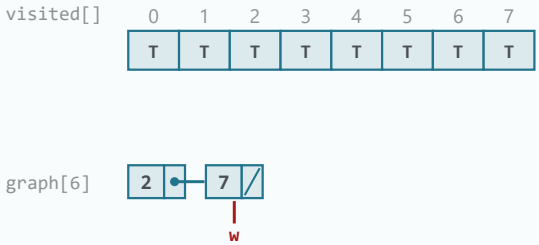
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)`
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





w → 7: `visited[7]` is TRUE; skip.

Step 3 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 6` `w = &node(7)`

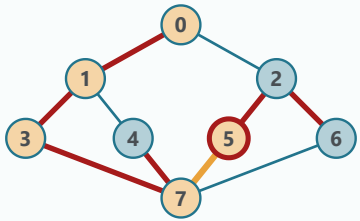
CALL STACK

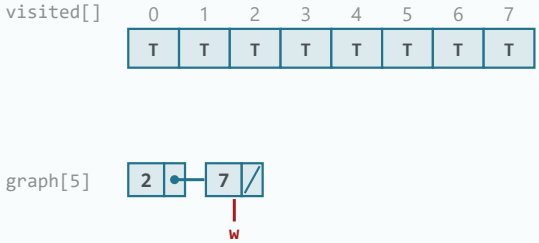
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)`
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





`dfs(6), dfs(2)` are done; resume where 5 left off. w → 7: `visited[7]` is TRUE; skip.

Step 4 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 5` `w = &node(7)`

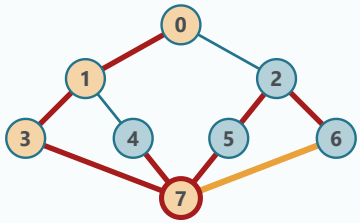
CALL STACK

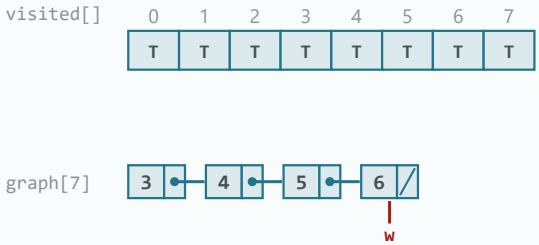
`dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





`dfs(5)` is done; resume where 7 left off. w → 6: `visited[6]` is TRUE; skip.

Step 5 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 7` `w = &node(6)`

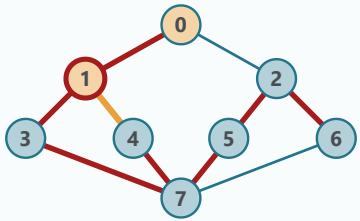
CALL STACK

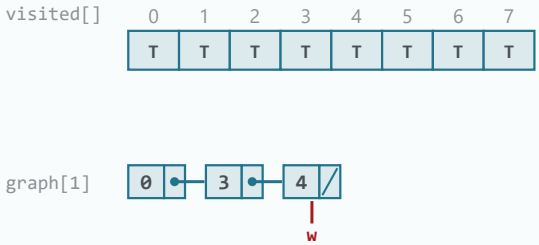
`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





`dfs(7), dfs(3)` are done; resume where 1 left off. w → 4: `visited[4]` is TRUE; skip.

Step 6 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 1` `w = &node(4)`

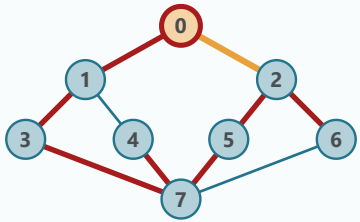
CALL STACK

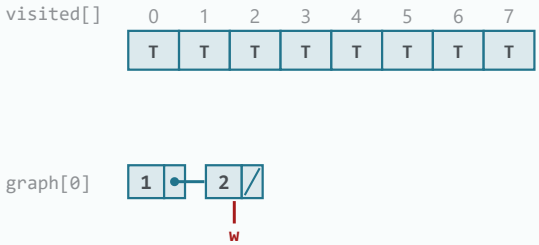
`dfs(1)` `dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)





`dfs(1)` is done; resume where 0 left off. w → 2: `visited[2]` is TRUE; skip.

Step 7 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
▶     for (w = graph[v]; w; w = w->link)
▶         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```

VARIABLES

`v = 0` `w = &node(2)`

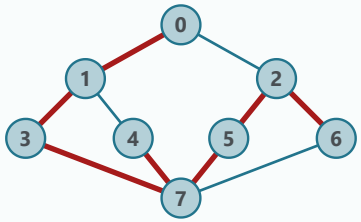
CALL STACK

`dfs(0)` `main()`

OUTPUT

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

Code Trace: `dfs(0)` (Cont.)



visited[]

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

`dfs(0)` is done. Visit order **0 1 3 7 4 5 2 6**; the orange edges form a **tree**.

Step 8 / 8

```
1 void dfs(int v)   /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;    /* visited */
5     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
▶ }
```

VARIABLES

`v = 0` `w = NULL`

CALL STACK

`main()`

OUTPUT

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

What Does the DFS Result Depend On?

Start vertex

From 0: 0 1 3 7 4 5 2 6. The start need not be 0: from 6, 6 2 0 1 3 7 4 5; from 4, 4 1 0 2 5 7 3 6. Same vertices, different order.

Effect of list order

Had 0's list been **{2, 1}** instead of {1, 2}, the order would be 0 2 5 7 3 1 4 6. Whether lists were built by inserting at the front or end, or kept sorted, changes the traversal.

Analysis

dfs is called at most **once** per vertex; each list is scanned once: **O(n + e)**. There are at most n(n − 1)/2 edges.

Data structure

With a matrix, finding v's neighbors would take O(n), **$O(n^2)$** in total. With lists we traverse in linear time.

The traversal produced a **tree** (the depth-first spanning tree). Adding one more edge to it creates a cycle, and it is no longer a tree.

Breadth-First Search (BFS)

- Start at v and mark it; then visit **all** the vertices on v's list.
- Then the unvisited neighbors of the **first** vertex on v's list, then those of the second…
- A visited vertex goes into a **queue** (first in, first out); when a list is finished, a vertex is taken from the queue and its list is scanned.
- The search ends when the queue is empty.

> **Instructor:** "Just as depth-first search by its nature required a stack, this one requires a queue."

```
void bfs(int v)   /* breadth-first search */
{
    node_ptr w;
    queue_ptr front = NULL, rear = NULL;
    printf("%d ", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front, &rear);
        for (w = graph[v]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%d ", w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
    }
}
```

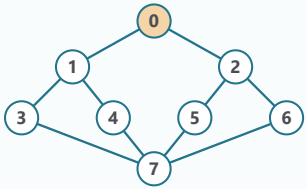
The Dynamic Queue Used by BFS

```
typedef struct queue *queue_ptr;
typedef struct queue {
    int vertex;
    queue_ptr link;
} queue;
void addq(queue_ptr *f,
          queue_ptr *r, int d)
{
    queue_ptr p = malloc(sizeof(queue));
    if (p == NULL) exit(1);
    p->vertex = d;  p->link = NULL;
    if (*f) (*r)->link = p;  else *f = p;
    *r = p;
}
```

```
int deleteq(queue_ptr *f,
            queue_ptr *r)
{
    queue_ptr p = *f;   /* f != NULL */
    int d = p->vertex;
    *f = p->link;
    if (*f == NULL) *r = NULL;
    free(p);
    return d;
}
```

A prototype such as `int deleteq(queue_ptr *)`, taking only the front pointer, is not enough. When the queue empties, the rear pointer must become NULL too, so `deleteq` also gets `r`.

Code Trace: `bfs(0)`





0 is printed, marked and added to the queue.

Step 1 / 12

```
1  front = rear = NULL;
▶  printf("%d ", v);  visited[v] = TRUE;
▶  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

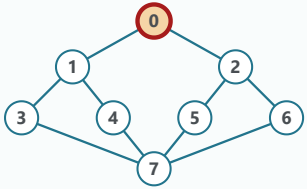
VARIABLES

`v = 0` `w = –`

OUTPUT

```
0
```

Code Trace: `bfs(0)`



visited[]

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

queue: empty (front = rear = NULL)

graph[0]: 1 → 2 /

The queue is not empty: the front item **0** is removed; now 0's list is scanned.

Step 2 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

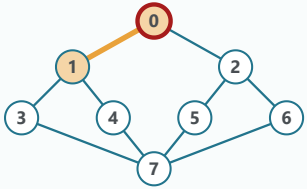
VARIABLES

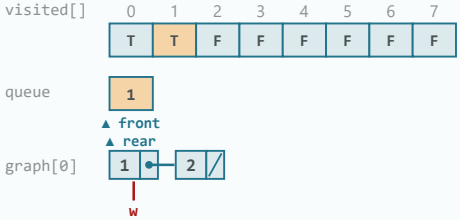
`v = 0` `w = –`

OUTPUT

```
0
```

Code Trace: `bfs(0)`





w → 1: unvisited → printed, added to the **rear** of the queue, marked.

Step 3 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

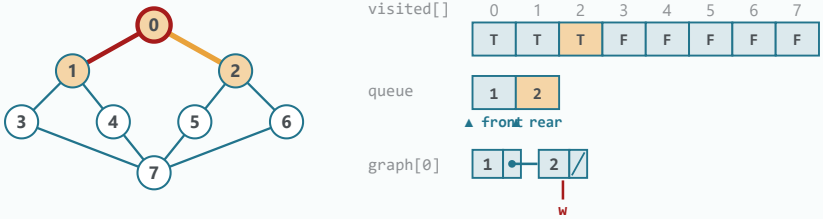
VARIABLES

`v = 0` `w = &node(1)`

OUTPUT

```
0 1
```

Code Trace: `bfs(0)`



w → 2: unvisited → printed, added to the **rear** of the queue, marked.

Step 4 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

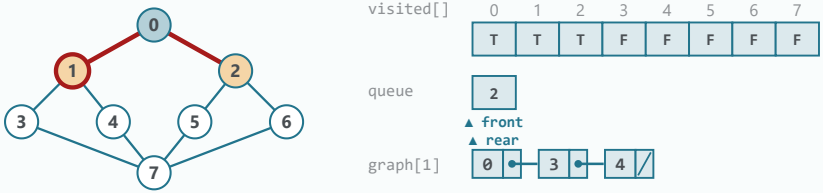
VARIABLES

`v = 0` `w = &node(2)`

OUTPUT

```
0 1 2
```

Code Trace: `bfs(0)`



The queue is not empty: the front item **1** is removed; now 1's list is scanned.

Step 5 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

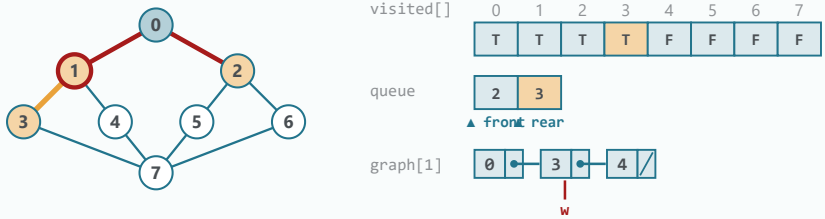
VARIABLES

`v = 1` `w = –`

OUTPUT

```
0 1 2
```

Code Trace: `bfs(0)`



w → 3: unvisited → printed, added to the **rear** of the queue, marked. (0 was already marked.)

Step 6 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

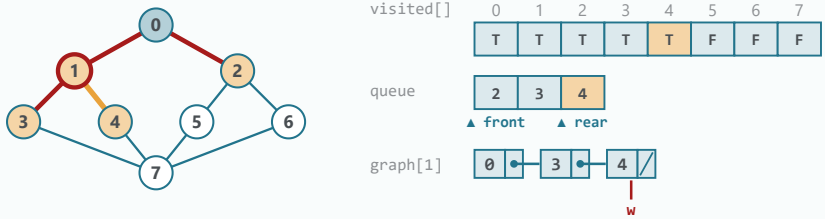
VARIABLES

`v = 1` `w = &node(3)`

OUTPUT

```
0 1 2 3
```

Code Trace: `bfs(0)`



w → 4: unvisited → printed, added to the **rear** of the queue, marked.

Step 7 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

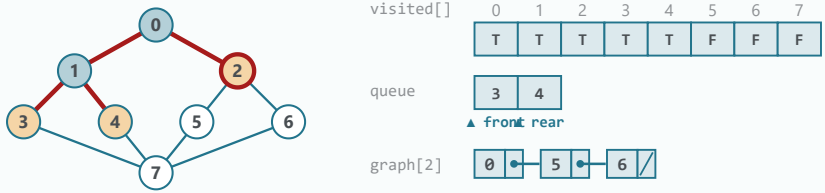
VARIABLES

`v = 1` `w = &node(4)`

OUTPUT

```
0 1 2 3 4
```

Code Trace: `bfs(0)`



The queue is not empty: the front item **2** is removed; now 2's list is scanned.

Step 8 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

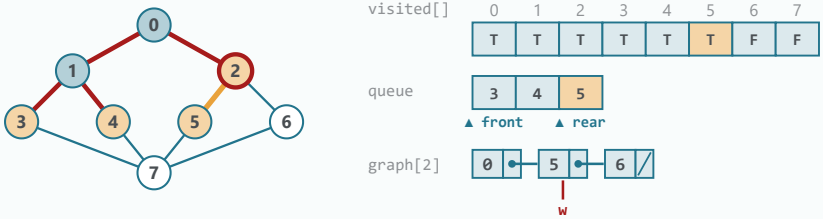
VARIABLES

`v = 2` `w = –`

OUTPUT

```
0 1 2 3 4
```

Code Trace: `bfs(0)`



w → 5: unvisited → printed, added to the **rear** of the queue, marked. (0 was already marked.)

Step 9 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

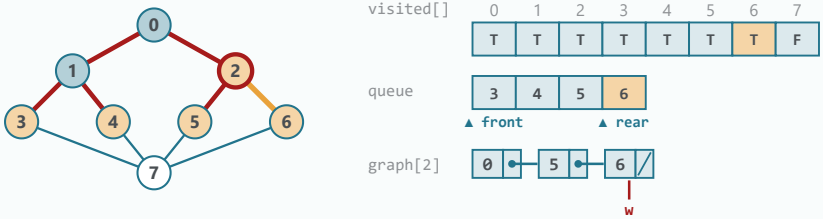
VARIABLES

`v = 2` `w = &node(5)`

OUTPUT

```
0 1 2 3 4 5
```

Code Trace: `bfs(0)`



w → 6: unvisited → printed, added to the **rear** of the queue, marked.

Step 10 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

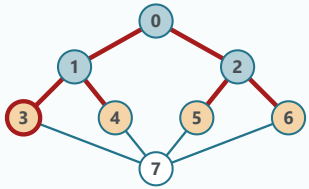
VARIABLES

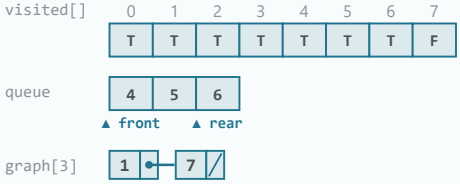
`v = 2` `w = &node(6)`

OUTPUT

```
0 1 2 3 4 5 6
```

Code Trace: `bfs(0)`





The queue is not empty: the front item **3** is removed; now 3's list is scanned.

Step 11 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

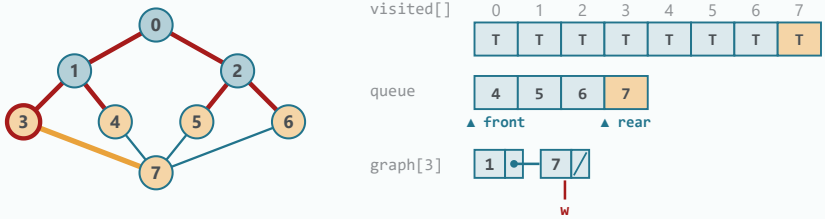
VARIABLES

`v = 3` `w = –`

OUTPUT

```
0 1 2 3 4 5 6
```

Code Trace: `bfs(0)`



w → 7: unvisited → printed, added to the **rear** of the queue, marked. (1 was already marked.)

Step 12 / 12

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
4  while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
▶          if (!visited[w->vertex]) {
▶              printf("%d ", w->vertex);
▶              addq(&front, &rear, w->vertex);
▶              visited[w->vertex] = TRUE;
11         }
12 }
```

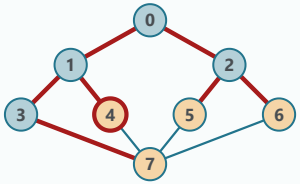
VARIABLES

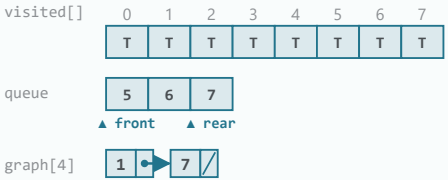
`v = 3` `w = &node(7)`

OUTPUT

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

Code Trace: `bfs(0)` (Cont.)





The queue is not empty: the front item **4** is removed; now 4's list is scanned.

Step 1 / 5

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

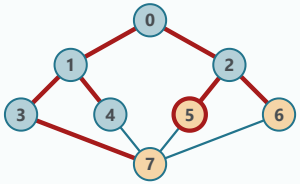
VARIABLES

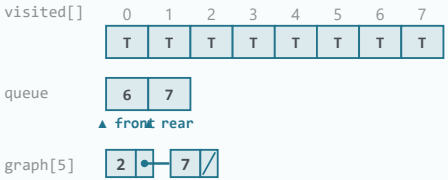
`v = 4` `w = –`

OUTPUT

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

Code Trace: `bfs(0)` (Cont.)





The queue is not empty: the front item **5** is removed; now 5's list is scanned.

Step 2 / 5

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

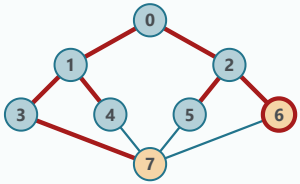
VARIABLES

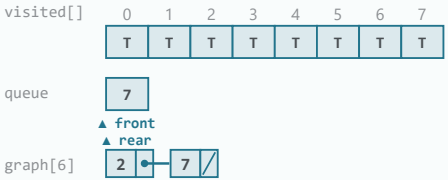
`v = 5` `w = –`

OUTPUT

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

Code Trace: `bfs(0)` (Cont.)





The queue is not empty: the front item **6** is removed; now 6's list is scanned.

Step 3 / 5

```
1 front = rear = NULL;
2 printf("%d ", v);  visited[v] = TRUE;
3 addq(&front, &rear, v);
▶ while (front) {
▶     v = deleteq(&front, &rear);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex]) {
8             printf("%d ", w->vertex);
9             addq(&front, &rear, w->vertex);
10            visited[w->vertex] = TRUE;
11        }
12 }
```

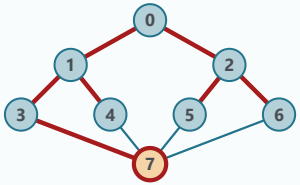
VARIABLES

`v = 6` `w = –`

OUTPUT

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

Code Trace: `bfs(0)` (Cont.)



visited[]

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

queue: empty (front = rear = NULL)

graph[7]: 3 → 4 → 5 → 6 → /

The queue is not empty: the front item **7** is removed; now 7's list is scanned.

Step 4 / 5

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶  while (front) {
▶      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

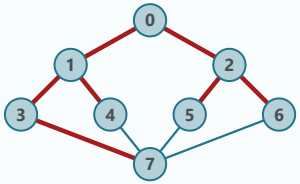
VARIABLES

`v = 7` `w = –`

OUTPUT

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

Code Trace: `bfs(0)` (Cont.)



visited[]

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

queue — empty (front = rear = NULL)

The queue is empty → the search ends. Order **0 1 2 3 4 5 6 7**: level by level. The orange edges are the breadth-first spanning tree.

Step 5 / 5

```
1  front = rear = NULL;
2  printf("%d ", v);  visited[v] = TRUE;
3  addq(&front, &rear, v);
▶ while (front) {
5      v = deleteq(&front, &rear);
6      for (w = graph[v]; w; w = w->link)
7          if (!visited[w->vertex]) {
8              printf("%d ", w->vertex);
9              addq(&front, &rear, w->vertex);
10             visited[w->vertex] = TRUE;
11         }
12 }
```

VARIABLES

`v = 7` `w = NULL`

OUTPUT

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

BFS: Analysis and Beyond

Analysis

Each vertex enters the queue **once** → the while loop runs n times. The lists hold 2e nodes in all: $d_0 + \ldots + d_{n-1} = 2e$. Total **O(n + e)**; with a matrix **$O(n^2)$**.

What does it give?

Visits in order of **number of edges** (level) from the start; the shortest-path tree of an unweighted graph.

Beyond

The ideas of depth-first/breadth-first search reappear in AI when a search space is traversed as a tree: minimax, B* search, *branch and bound*.

DFS or BFS?

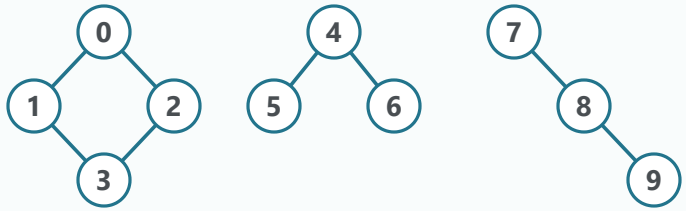
DFS: stack/recursion, goes deep (components, cycles, topological thinking).
BFS: queue, layer by layer (path with fewest edges).

> **Instructor:** “In game problems too, if they say ‘the moves could be these,’ that means it expands breadth-first.”

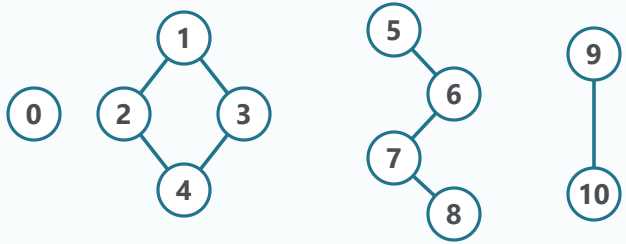
Connected Components

```
void connected(int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (!visited[i]) {   /* new */
            dfs(i);
            printf("\n");
        }
}
```

Is the undirected graph connected, and how many components does it have? Each DFS (or BFS) call marks one whole component; increment a counter on each new line and you get the **number of components** too.



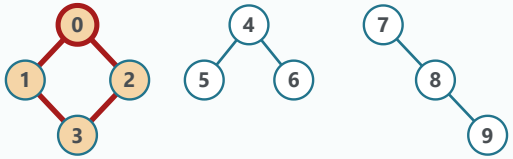
Output: 0 1 3 2 / 4 5 6 / 7 8 9



Output: 0 / 1 2 4 3 / 5 6 7 8 / 9 10

The output of line 2 is **1 2 4 3**; 5 belongs to the next component.

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

components: 1

i = 0: unmarked → **a new component**; `dfs(0)` visits 0 1 3 2.

Step 1 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 0`

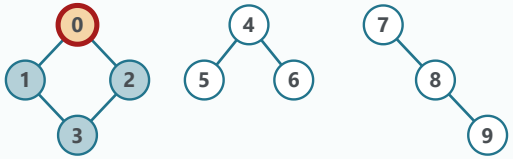
CALL STACK

`dfs(0)` `connected(10)` `main()`

OUTPUT

```
0 1 3 2
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

components: 1

dfs returned; go to a new line.

Step 2 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!visited[i]) {    /* new */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 0`

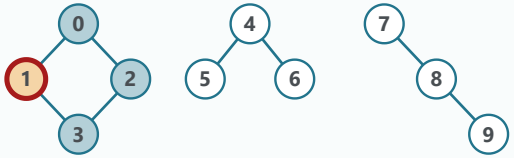
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

components: 1

i = 1: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 3 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 1`

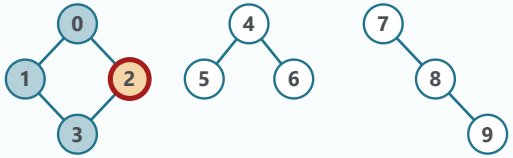
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

components: 1

i = 2: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 4 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 2`

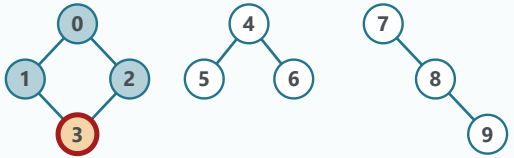
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

components: 1

i = 3: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 5 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 3`

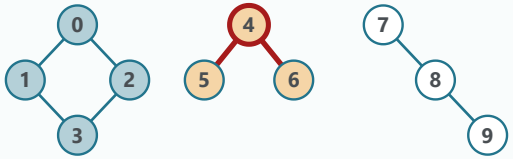
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

components: 2

i = 4: unmarked → **a new component**; `dfs(4)` visits 4 5 6.

Step 6 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 4`

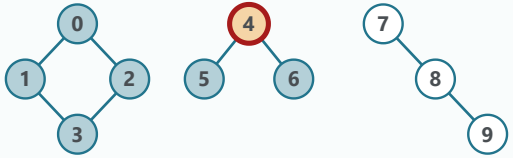
CALL STACK

`dfs(4)` `connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

components: 2

dfs returned; go to a new line.

Step 7 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!visited[i]) {    /* new */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 4`

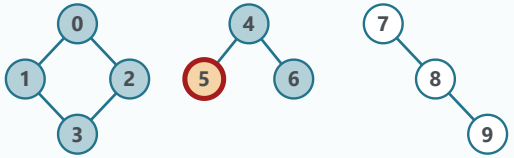
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

components: 2

i = 5: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 8 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 5`

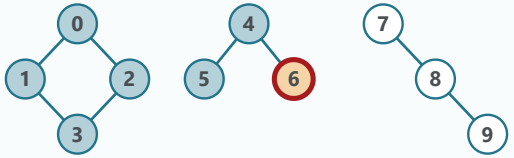
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

components: 2

i = 6: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 9 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 6`

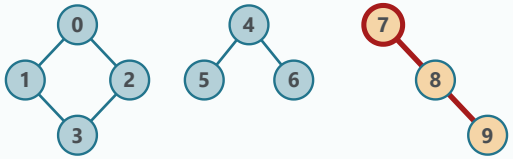
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

components: 3

i = 7: unmarked → **a new component**; `dfs(7)` visits 7 8 9.

Step 10 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 7`

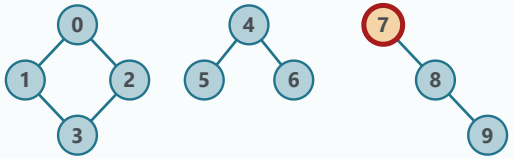
CALL STACK

`dfs(7)` `connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
7 8 9
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

components: 3

dfs returned; go to a new line.

Step 11 / 14

```
1 void connected(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!visited[i]) {    /* new */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 7`

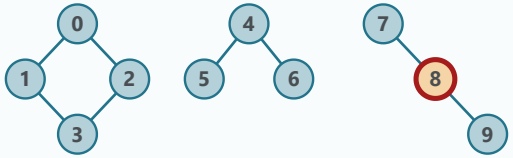
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
7 8 9
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

components: 3

i = 8: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 12 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 8`

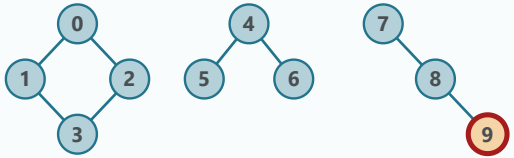
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
7 8 9
```

Code Trace: `connected(10)`



visited[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

components: 3

i = 9: already marked (visited by the previous DFS) → the if is skipped, **O(1)**.

Step 13 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!visited[i]) {   /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 9`

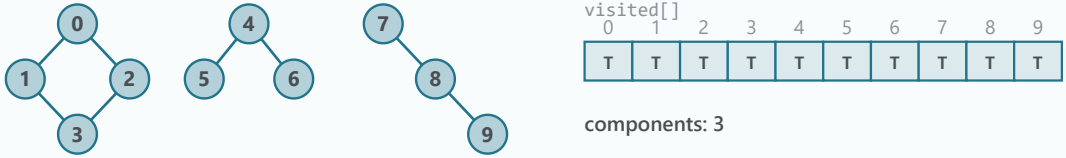
CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
7 8 9
```

Code Trace: `connected(10)`



The loop is done: **3 components**. Every vertex and every list was processed once in total.

Step 14 / 14

```
1 void connected(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
5         if (!visited[i]) {    /* new */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

VARIABLES

`n = 10` `i = 10`

CALL STACK

`connected(10)` `main()`

OUTPUT

```
0 1 3 2
4 5 6
7 8 9
```

Analysis: n × O(n + e)?

The outer for runs n times, and the inner `dfs()` was O(n + e). Is the complexity their product, $O(n^2)$?

> **Instructor:** "Is it? Is it? Is it? Does it really run that long? No! … 'I'll just multiply them all and state the result.' Wrong!"

Why O(n + e)?

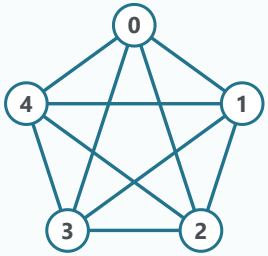
With a single component, the first iteration runs in O(n + e) and every `visited[]` becomes TRUE; for the other values of i the if is skipped. The components are disjoint, so the total work is **O(n + e)**.

With a matrix?

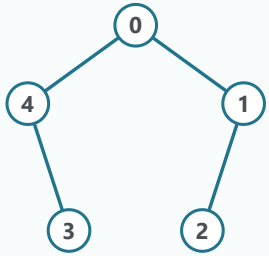
Each dfs call searches for neighbors in O(n) → finding components is **$O(n^2)$**. "If that is written efficiently too…"

Theorem: A Connected Graph Has at Least n − 1 Edges

- **Theorem:** a graph needs at least **n − 1** edges to be connected.
- **It is one-way:** n − 1 edges are *necessary* but *not sufficient*; a graph with n − 1 edges may or may not be connected. With fewer it is certainly not connected.
- At most: if all vertices are adjacent to each other (the complete graph K_n), (n − 1) + (n − 2) + … + 1 = **n(n − 1)/2**. So e can be quadratic in n.
- In a connected graph: **n − 1 ≤ e ≤ n(n − 1)/2**.



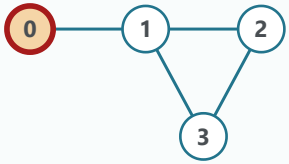
K5: 10 edges



n − 1 = 4 edges, connected

Instructor: "Tic-tac-toe… 'How do I spoil my opponent's move?' I expect these concepts will settle better once you read about it."

Code Trace: Does an Undirected Graph Have a Cycle?



visited[]

0	1	2	3
T	F	F	F

`has_cycle(0, -1)` : 0 is marked.

Step 1 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
▶      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
8          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

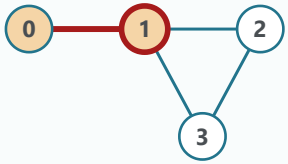
VARIABLES

`v = 0` `parent = -1`

CALL STACK

`has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?



visited[]

0	1	2	3
T	T	F	F

`has_cycle(1, 0)` : 1 is marked.

Step 2 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
▶      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
8          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

VARIABLES

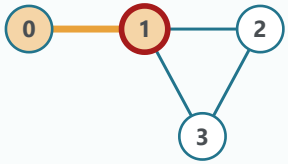
`v = 1` `parent = 0`

CALL STACK

`has_cycle(1, 0)`

`has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?



visited[]

0	1	2	3
T	T	F	F

w → 0: marked, but it is 1's parent; the edge we came in on is not a cycle.

Step 3 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
4      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
▶          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

VARIABLES

`v = 1` `parent = 0`

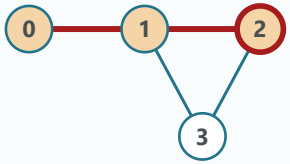
`w = &node(0)`

CALL STACK

`has_cycle(1, 0)`

`has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?



visited[]

0	1	2	3
T	T	T	F

`has_cycle(2, 1)` : 2 is marked.

Step 4 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
▶      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
8          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

VARIABLES

`v = 2` `parent = 1`

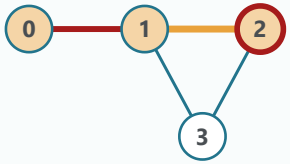
CALL STACK

`has_cycle(2, 1)`

`has_cycle(1, 0)`

`has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?



visited[]

0	1	2	3
T	T	T	F

w → 1: marked, but it is 2's parent; the edge we came in on is not a cycle.

Step 5 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
4      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
▶          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

VARIABLES

`v = 2` `parent = 1`

`w = &node(1)`

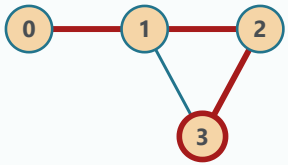
CALL STACK

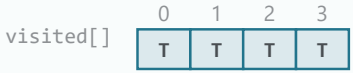
`has_cycle(2, 1)`

`has_cycle(1, 0)`

`has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?





`has_cycle(3, 2)` : 3 is marked.

Step 6 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
▶      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
8          } else if (w->vertex != parent)
9              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

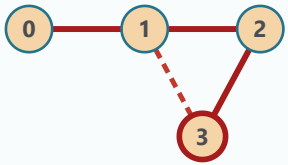
VARIABLES

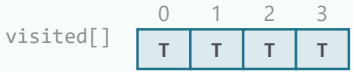
`v = 3` `parent = 2`

CALL STACK

- `has_cycle(3, 2)`
- `has_cycle(2, 1)`
- `has_cycle(1, 0)`
- `has_cycle(0, -1)` `main()`

Code Trace: Does an Undirected Graph Have a Cycle?





w → 1: marked **and** not the parent (2) → a **back edge** (3,1): there is a cycle! TRUE is passed up.

Step 7 / 7

```
1  bool has_cycle(int v, int parent)
2  {
3      node_ptr w;
4      visited[v] = TRUE;
5      for (w = graph[v]; w; w = w->link)
6          if (!visited[w->vertex]) {
7              if (has_cycle(w->vertex, v)) return TRUE;
▶          } else if (w->vertex != parent)
▶              return TRUE;          /* back edge */
10     return FALSE;
11 }
```

VARIABLES

`v = 3` `parent = 2`

`w = &node(1)`

CALL STACK

`has_cycle(3, 2)`

`has_cycle(2, 1)`

`has_cycle(1, 0)`

`has_cycle(0, -1)` `main()`

Cycle Detection: Undirected and Directed

Undirected

In DFS, a visited neighbor that is **not the parent** = a back edge = a cycle. O(n + e).

Directed

An edge to a vertex “currently on the stack” (gray) is a cycle; three colors: white/gray/black. A topological sort routine also catches cycles.

Why does it matter?

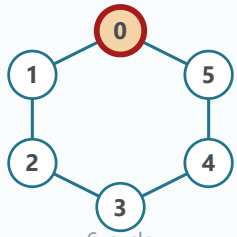
With prerequisite tasks, a cycle means the project never finishes; in an MST, a cycle means an extra edge.

With a matrix

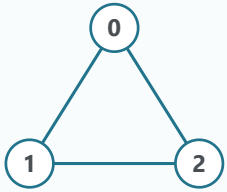
The same DFS becomes $O(n^2)$ with a matrix.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle



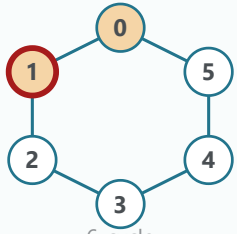
triangle

BFS: 0 colored (color 0). Each neighbor gets the **opposite** color.

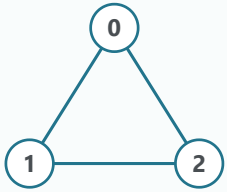
Step 1 / 10

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle

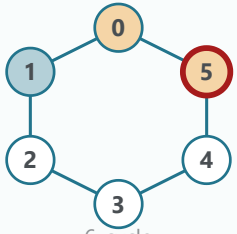


triangle

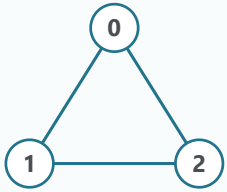
BFS: 1 colored (color 1). Each neighbor gets the **opposite** color.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle

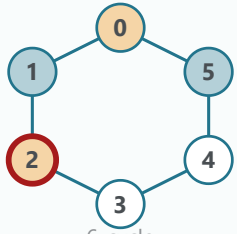


triangle

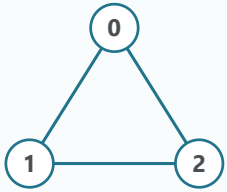
BFS: 5 colored (color 1). Each neighbor gets the **opposite** color.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle

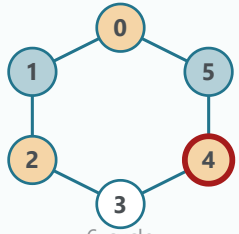


triangle

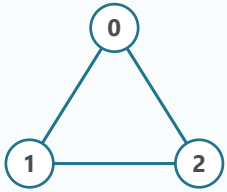
BFS: 2 colored (color 0). Each neighbor gets the **opposite** color.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle

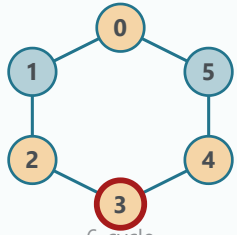


triangle

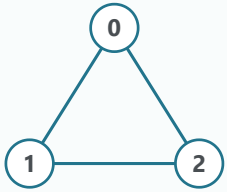
BFS: 4 colored (color 0). Each neighbor gets the **opposite** color.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle

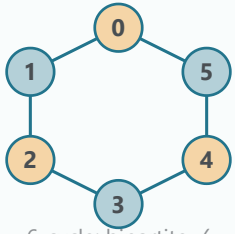


triangle

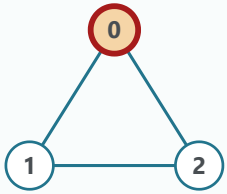
BFS: 3 colored (color 1). Each neighbor gets the **opposite** color.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle: bipartite ✓

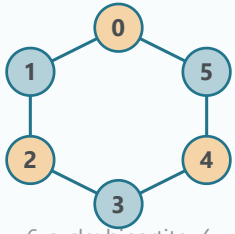


triangle

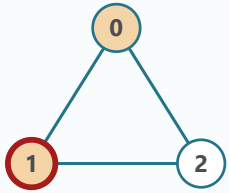
Triangle: 0 colored.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle: bipartite ✓

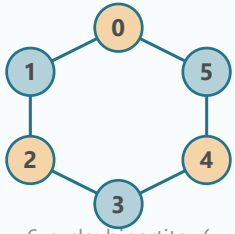


triangle

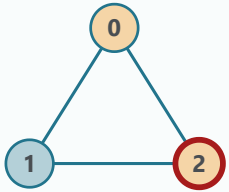
Triangle: 1 colored.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle: bipartite ✓

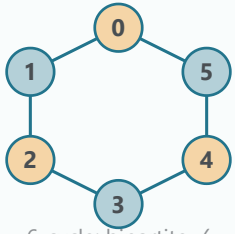


triangle

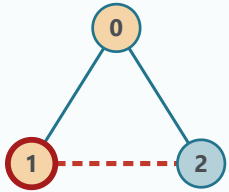
Triangle: 2 colored.

Bipartite Graph Test

- V can be split into two sets with every edge joining **different** sets: a bipartite graph (e.g. $K_{3,3}$, students–courses).
- Test: two-color with BFS, giving each neighbor the opposite color. If an edge joins two same-colored vertices, it is not bipartite.
- Theorem: a graph is bipartite ⇔ it has **no odd-length cycle**.
- Same cost as BFS: **O(n + e)**; start it separately for each component.



6-cycle: bipartite ✓



triangle

Two adjacent vertices with the same color → **not bipartite** (there is an odd cycle).

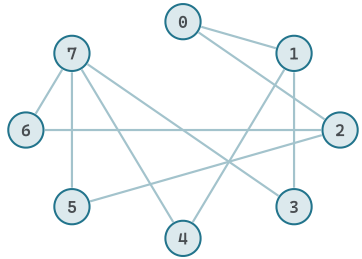
Lab: DFS and BFS

DFS: the 8-vertex graph

Input (undirected edges, e.g. 0-1, 0-2) `0-1, 0-2, 1-3, 1-4, 2-5, 2-6, 3-7, 4-7, 5-7, 6-7` Apply

CODE: dfs

```
1 void dfs(int v) /* depth-first search */
2 {
3     node_ptr w;
4     visited[v] = TRUE;      /* visited */
5     printf("%d ", v);
6     for (w = graph[v]; w; w = w->link)
7         if (!visited[w->vertex])
8             dfs(w->vertex);
9 }
```



`v –` `w NULL` `calls –`

OUTPUT (none yet)

Adjacency lists (in increasing order): 0: 1 2; 1: 0 3 4; 2: 0 5 6; 3: 1 7; 4: 1 7; 5: 2 7; 6: 2 7; 7: 3 4 5 6. The traversal starts at vertex 0.

SUMMARY

Complexity Summary

Operation	Matrix	Lists	Note
Memory	$O(n^2)$	$O(n + e)$	2e nodes in undirected lists
Degree (undirected)	$O(n)$	$O(d)$	inverse lists for in-degree in digraphs
Number of edges	$O(n^2)$	$O(n + e)$	Theorem 1: sum / 2
DFS / BFS	$O(n^2)$	$O(n + e)$	each vertex once
Connected components	$O(n^2)$	$O(n + e)$	not a product!
Cycles / bipartiteness	$O(n^2)$	$O(n + e)$	on top of DFS / BFS
Building transpose lists	—	$O(n + e)$	in a single pass

n: number of vertices · e: number of edges · d: degree of a vertex · $n - 1 \leq e \leq n(n - 1)/2$ (connected simple graph)

SUMMARY

Common Mistakes

1. Assuming <v,u> exists in a digraph because <u,v> does; taking the matrix to be symmetric.
2. In BFS, marking a vertex when it leaves the queue instead of when it enters → the same vertex is queued several times.
3. Forgetting to reset `visited[]` to FALSE before each traversal (a global array!).
4. Multiplying the outer for by DFS and calling components O(n(n + e)).
5. Not checking the result of `malloc` when building lists; not `free`ing the lists at the end of the program.
6. Appending each edge at the end and walking the list from the start each time; missing the O(1) insert at the front.

Exercises

1. Is there a simple undirected graph with degree sequence (3, 3, 2, 2, 1)? How many edges does it have?
2. In the sequential array representation of G4, which indices hold the neighbors of vertex 5?
3. Run DFS and BFS on the 8-vertex sample graph starting at 7 (lists sorted).
4. What happens if the line `visited[w→vertex] = TRUE` in the BFS code is moved after `deleteq`?
5. Write a routine that counts the components of a graph given as an adjacency matrix; what is its complexity?
6. To report in-degree in O(1) for a digraph, which structure would you keep, and how would you update it?

▶ **Answers / hints**

SUMMARY

Chapter Summary

Concepts

G = (V, E); directed/undirected, adjacent, subgraph, path, degree; Σd = 2e; connectivity needs e ≥ n − 1.

Representations

Matrix $O(n^2)$; lists O(n + e); sequential array; orthogonal (directed); multilist (one node per edge).

Traversals

DFS = recursion/stack; BFS = queue. Both O(n + e) with lists, $O(n^2)$ with a matrix.

Applications

Connected components in O(n + e); cycle detection; bipartite test; groundwork for spanning trees.

Method

First the problem, then how the data is stored; don't trust the first solution.

SUMMARY

References

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 6 (Graphs).
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Chapter 20 (Elementary Graph Algorithms).
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed.

← Chapter 14: Hash Tables

Course page

Chapter 16: Sets and Union-Find

The End

Chapter 15: Graphs

Next chapter: Chapter 16: Sets and Union-Find · Course page