Unit 4.1 Breadth First Search

Algorithms

EE3980

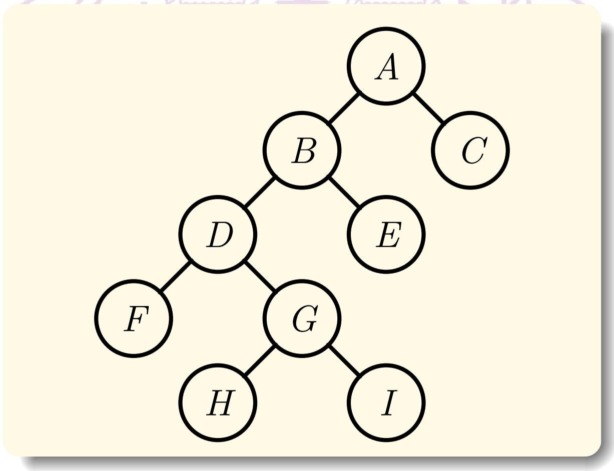
Mar. 28, 2019

Binary Tree Traversal

- Given a binary tree, some applications need to visit every node of the tree.
- It is assumed that each node of the tree has the underlying structure as

```
struct node {
    Type data; // store data of specified Type
    node *lchild, *rchild;
}
```

- Example

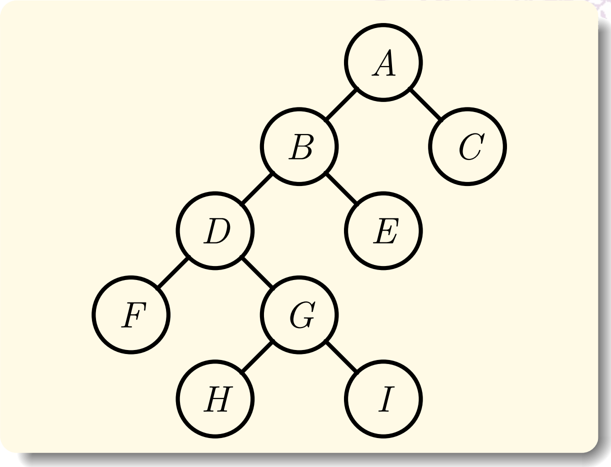


- Three ways to traverse a binary tree

Binary Tree — In-order Traversal

Algorithm 4.1.1. In Order Traversal

// To visit every node of the binary tree *in–order*.
// Input: tree T; Output: none.
Algorithm InOrder(T)
{
    if ($T \neq$ NULL) then {
        InOrder($T \to lchild$) ; Visit(T) ; InOrder($T \to rchild$) ;
    }
}



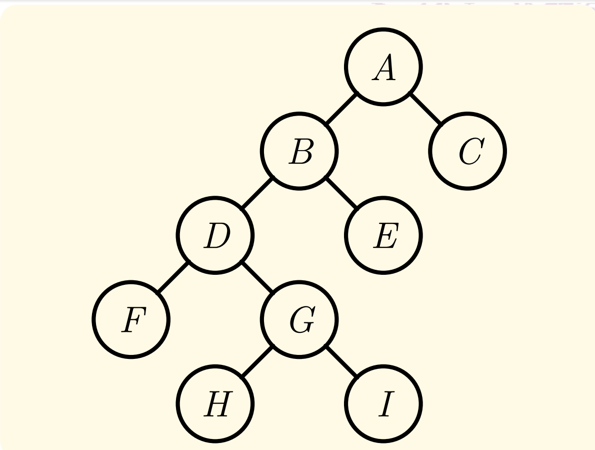
- Execution sequence

InOrder A	visit G
InOrder B	InOrder I
InOrder D	visit I
InOrder F	visit B
visit F	InOrder E
visit D	visit E
InOrder G	visit A
InOrder H	InOrder C
visit H	visit C

Binary Tree — Pre-order Traversal

Algorithm 4.1.2. Pre-Order Traversal

// To visit every node of the binary tree *pre–order*.
// Input: tree T; Output: none.
Algorithm PreOrder(T)
{
    if ($T \neq$ NULL) then {
        Visit(T) ; PreOrder($T \to lchild$) ; PreOrder($T \to rchild$) ;
    }
}



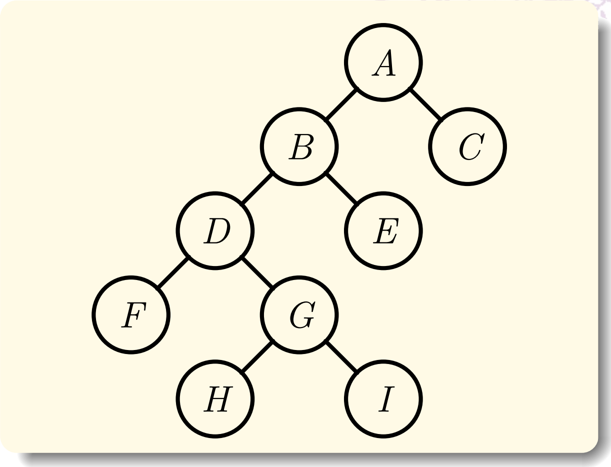
- Execution sequence

PreOrder A	visit G
visit A	PreOrder H
PreOrder B	visit H
visit B	PreOrder I
PreOrder D	visit I
visit D	PreOrder E
PreOrder F	visit E
visit F	PreOrder C
PreOrder G	visit C

Binary Tree — Post-order Traversal

Algorithm 4.1.3. Post-Order Traversal

```
// To visit every node of the binary tree post–order.
// Input: tree T; Output: none.
Algorithm PostOrder(T)
{
    if (T ≠ NULL ) then {
        PostOrder(T → lchild); PostOrder(T → rchild); Visit(T);
    }
}
```



- Execution sequence

PostOrder A	visit I
PostOrder B	visit G
PostOrder D	visit D
PostOrder F	PostOrder E
visit F	visit E
PostOrder G	visit B
PostOrder H	PostOrder C
visit H	visit C
PostOrder I	visit A

Binary Tree Traversal — Complexities

- In traversing the tree, each node is reached three times
  - From its root; when returning from *lchild* and *rchild*
- Thus, the time complexity is $T(n) = \Theta(n)$ for an n-node binary tree.
- The space needed for an n-node binary tree is $\Theta(n)$.
- Traversing the tree using recursive calls would need a heap space proportional to the depth, d, of the tree.
- Since $d \le n$, the space complexity is $\mathcal{O}(n)$.

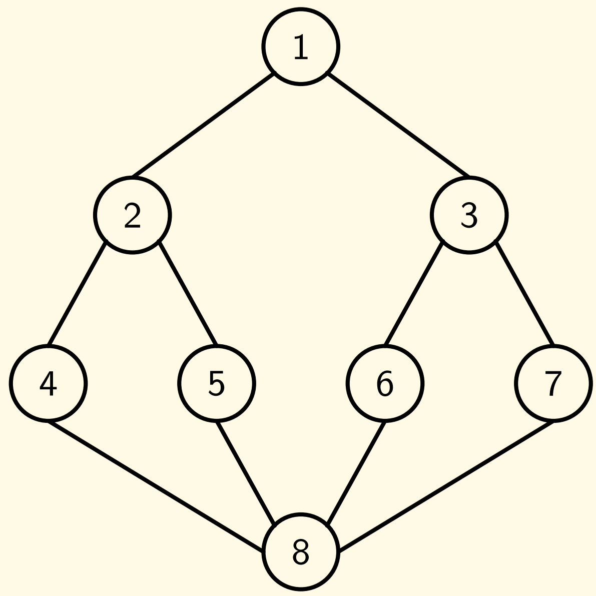
Theorem 4.1.4. Binary Tree Traversal

Let $T(n)$ and $S(n)$ be the time and space complexities of any of the binary traversing algorithms above, then $T(n) = \Theta(n)$ and $S(n) = \mathcal{O}(n)$.

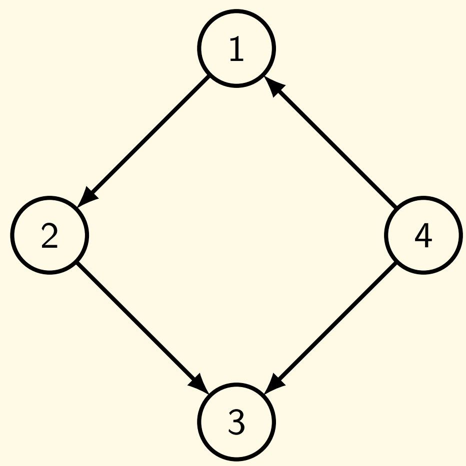
- Proof, please see textbood [Horowitz], pp. 335-337.

Graph Traversal

- Given a graph $G = (V, E)$ with vertex set V and edge set E, a typical graph traversal problem is to find all vertices that is reachable from a particular vertex, for example $v \in V$.
  - Note that G can be either a directed graph or undirected graph.



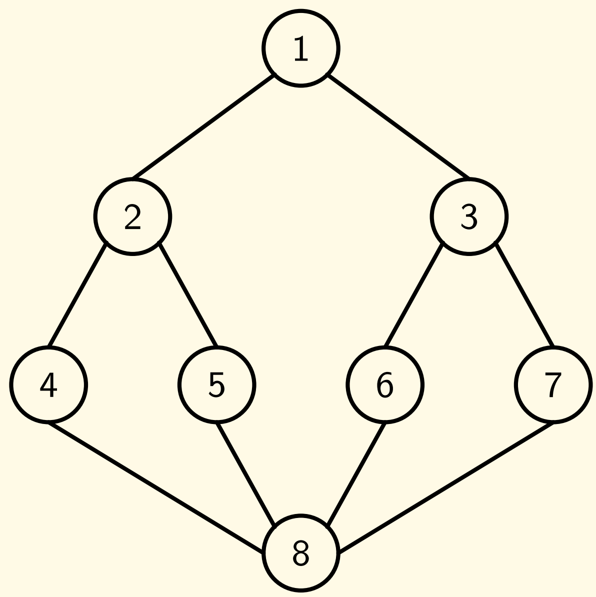
An undirected graph.

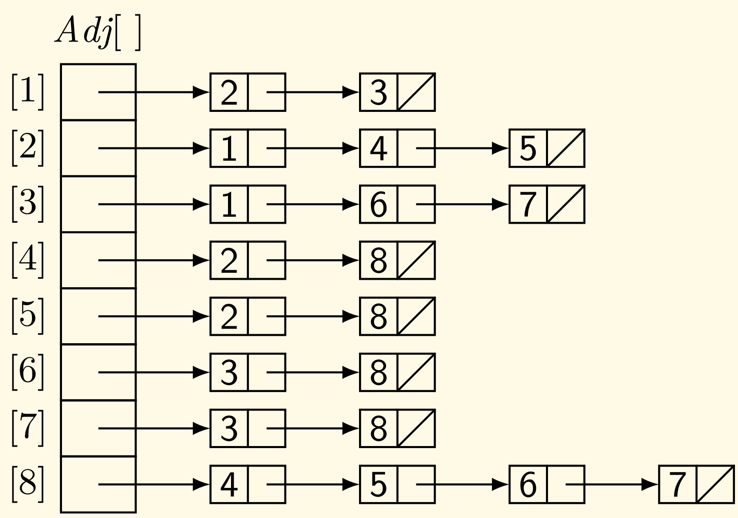


A directed graph.

Graph and Adjacency Lists

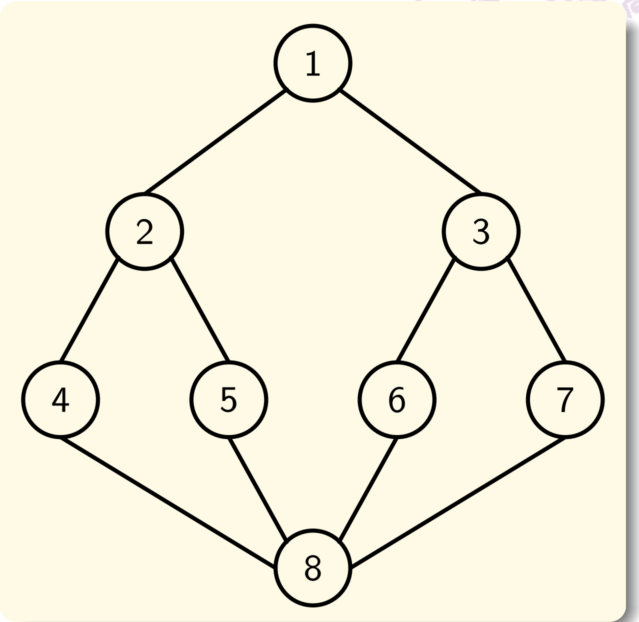
- One way to represent the adjacency information of a graph $G = (V, E)$ is the adjacency list.
  - Both directed and undirected graphs can be represented.
  - In a undirected graph, each edge should appear twice.
  - More efficient if the graph is sparse, $|E| \ll |V|^2$.
  - Weighted graphs can also be represented with more space for each edge.





Graph and Adjacency Matrix

- The other way to keep the adjacent information of a graph $G = (V, E)$ is the adjacency matrix.
  - For undirected graphs, symmetric matrices are obtained.
  - Asymmetric matrices for directed graphs.
  - Weighted graphs can also be represented.
  - More applicable when the graph is dense, $|E| \approx |V|^2$, or faster search of an edge (i, j) is needed.



$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1.1}$$

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

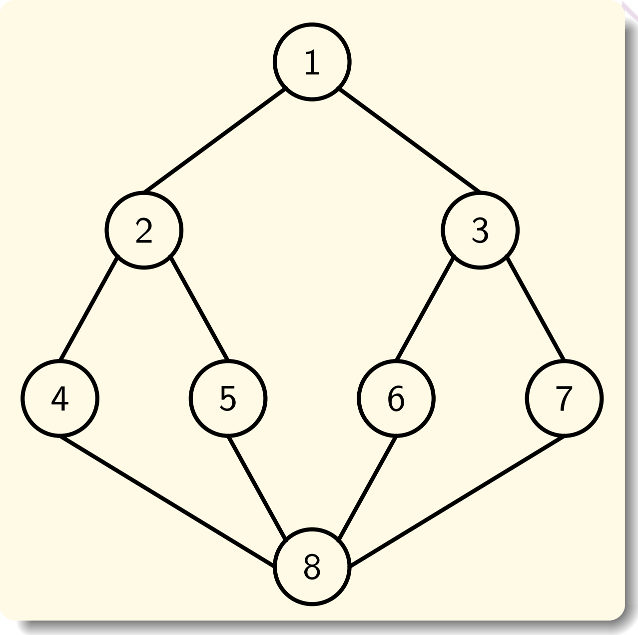
Breadth First Search

- A popular graph traversal algorithm for both directed and undirected graphs is

Algorithm 4.1.5. Breadth First Search

```
// Breadth first search starting from vertex v of graph G.
// Input: v starting node ; Output: none.
Algorithm BFS(v) // Queue Q is global; Array visited initialized to 0.
{
    u := v; visited[v] := 1 ; // Visit v first.
    repeat {
        for all vertices w adjacent to u do { // Visit and enqueue adj. nodes.
            if (visited[w] = 0) then {
                Enqueue(w) ; visited[w] := 1 ;
            }
        }
        if not Qempty() then u := Dequeue() ; // get the next vertex.
    } until ( Qempty()) ;
}
```

BFS Example



- BFS calling sequence

visit 1	$Queue = \{2, 3\}$
visit 2	$Queue = \{3, 4, 5\}$
visit 3	$Queue = \{4, 5, 6, 7\}$
visit 4	$Queue = \{5, 6, 7, 8\}$
visit 5	$Queue = \{6, 7, 8\}$
visit 6	$Queue = \{7, 8\}$
visit 7	$Queue = \{8\}$
visit 8	$Queue = \{\ \}$

Breadth First Search – Properties

Theorem 4.1.6. BFS Complexities

Let $T(n, e)$ and $S(n, e)$ be the maximum time and maximum *additional* space taken by algorithm BFS on any graph G with n vertices and e edges.

1. $T(n, e) = \Theta(n + e)$ and $S(n, e) = \Theta(n)$ if G is represented by its adjacency lists,
2. $T(n, e) = \Theta(n^2)$ and $S(n, e) = \Theta(n)$ if G is represented by its adjacency matrix.

- Proof please see textbook [Horowitz], pp. 341-343.
    - The additional space refers to array $visited[1 : n]$, $\Theta(n)$, and memory needed for the queue, $\mathcal{O}(n)$.

Theorem 4.1.7. BFS Reachability

Algorithm BFS visits all vertices of G reachable from v.

- Proof please see textbook [Horowitz], p. 340.

Shortest Path

Definition 4.1.8. Shortest Path.

Given a graph $G = (V, E)$, the shortest-path distance, $\delta(s, v)$, between any two vertices, $s, v \in V$, is the minimum number of edges in any path from s to v. If there is no path from s to v then $\delta(s, v) = \infty$. A path of length $\delta(s, v)$ from s to v is a shortest path from s to v.

Lemma 4.1.9.

Given a directed or undirected graph $G = (V, E)$ and an arbitrary vertex $s \in V$, then for any edge $(u, v) \in E$ we have

$$\delta(s, v) \leq \delta(s, u) + 1. \tag{4.1.2}$$

- Proof please see textbook [Cormen], p. 598.

Shortest Path and Breadth First Search

- The breadth first search algorithm can be modified to find the shortest distance to other vertices.

Algorithm 4.1.10. Shortest path – Breadth First Search

```
// Breadth first search starting from v to find all shortest path length.
// Input: v; Output: array d, distance from v, p predecessor on the path.
Algorithm BFS_d(v, d, p)
{
    u := v; visited[v] := 1;
    d[v] := 0; p[v] := 0; // Both d, p initialized to 0.
    repeat {
        for all vertices w adjacent to u do { // Breadth first traversal.
            if (visited[w] = 0) then {
                Enqueue(w); visited[w] := 1;
                d[w] := d[u] + 1; p[w] := u; // update d and p arrays.
            }
        }
        if not Qempty() then u := Dequeue(); // Get the next vertex.
    } until (Qempty());
}
```

Shortest Path and Breadth First Search, II

Lemma 4.1.11.

Given a graph $G = (V, E)$, if the BFS_d(s, d) is called for a source vertex $s \in V$, then upon the termination of the algorithm we have for any $v \in V$, $d[v] \geq \delta(s, v)$.

- Proof please see textbook [Cormen], p. 598.

Lemma 4.1.12.

Suppose that during the execution of the BFS_d(s, d) algorithm on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of the queue and v_r is the tail. Then, we have

$$d[v_r] \leq d[v_1] + 1, \tag{4.1.3}$$

$$d[v_i] \leq d[v_{i+1}] \quad \text{for } i = 1, 2, \ldots, r-1. \tag{4.1.4}$$

- Proof please see textbook [Cormen], p. 599.

Shortest Path and Breadth First Search, III

Corollary 4.1.13.

Suppose that during the execution of the BFS_d(s, d) algorithm on a graph $G = (V, E)$, both vertices v_i and v_j are enqueue and v_i is enqueued before v_j, then $d[v_i] \leq d[v_j]$.

- Proof please see textbook [Cormen], p. 599.

Theorem 4.1.14.

Given a graph $G = (V, E)$ and a source vertex $s \in V$, if the algorithm BFS_d(s, d) is called, then for every vertex $v \in V$ reachable from s, upon termination we have $d[v] = \delta(s, v)$.

- Proof please see textbook [Cormen], p. 600.

Shortest Path and Breadth First Search – Print Path

- A shortest path from source s to any vertex $v \in V$ can be printed using the array p.
  - Note that array p records the predecessor information.
  - $p[w]$ is the vertex preceding vertex w in the shortest path.
  - For source vertex v, $p[v] = 0$.

Algorithm 4.1.15. Print Shortest Path

```
// To print the shortest path that ends at w using array p.
// Input: vertex w, path array p; Output: shortest path that ends at w.
Algorithm BFSpath(w, p)
{
    if (p[w] ≠ 0) BFSpath(p[w]);
    write ( " w " );
}
```


Spanning Trees of Connected Graphs

- The BFS algorithm can be modified to find the spanning tree of a connected graph.

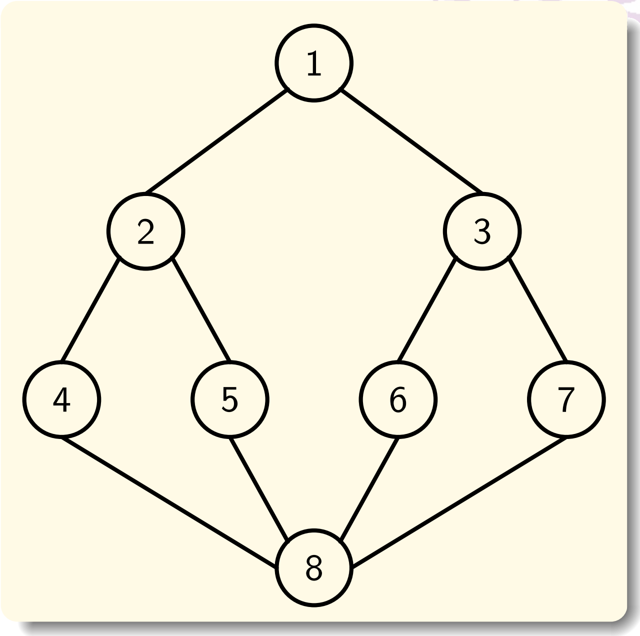
Algorithm 4.1.16. BFS to find a spanning tree

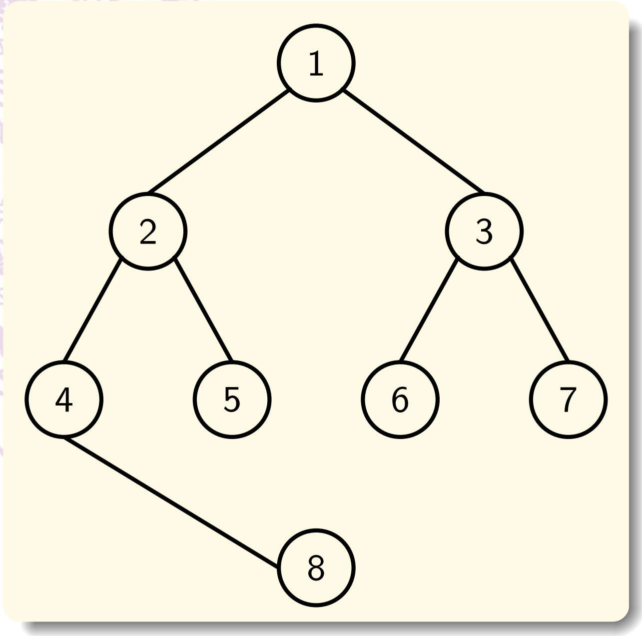
```
// Breadth first search to find the spanning tree from vertex v.
// Input: source node v; Output: spanning tree t.
Algorithm BFS*(v, t)
{
    u := v; visited[v] := 1; t := ∅; // t initialized to empty set.
    repeat {
        for all vertices w adjacent to u do {
            if (visited[w] = 0) then {
                Enqueue(w); visited[w] := 1;
                t := t ∪ {(u, w)}; // Add edge to spanning tree.
            }
        }
        if not Qempty() then u := Dequeue(u); // Get the next vertex.
    } until ( Qempty());
}
```

- On termination, t is the set of edges that forms a spanning tree of G.

BFS Spanning Tree

- The spanning tree found by Algorithm `BFS*` can be called BFS spanning tree.
- This tree has the property that the path from the root s to any vertex $v \in V$ is a shortest path.
- Example





- The time and space complexity of `BFS*` is the same as `BFS`.

Summary

- Binary tree traversal
- Graph traversal
- Breadth first search
- Spanning tree