UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

1

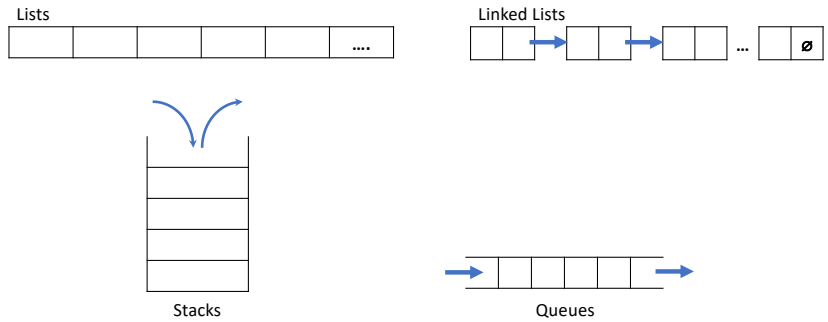
Module 11

Graphs

2

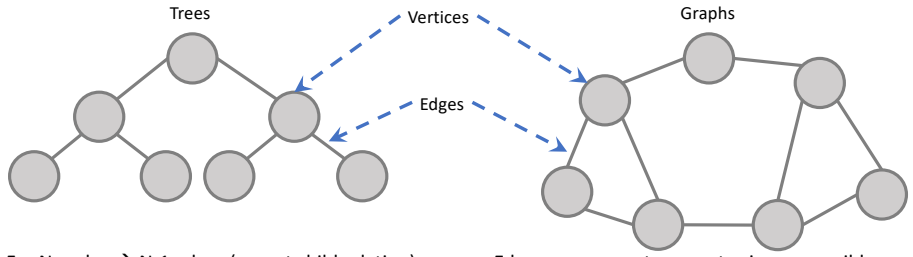
Quick Recap

Linear data structures



Quick Recap

Non-Linear data structures



- For N nodes → N-1 edges (parent child relation)
- All nodes are accessible through root
- Also, there is only one path to reach to any node from root
- Trees are special form of Graphs

- Edges can connect any vertex in any possible way
- Any vertex can be accessed through any path

Graphs

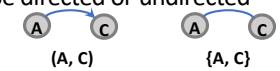
A graph is a way of representing relationships that exist between pairs of objects.

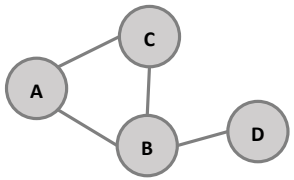
- A Graph **G** can be represented as an ordered pair of a set **V** of vertices and a set **E** of edges as:

  G = (V, E)

- Each vertex in a graph can have a name or a value
  V = {A, B, C, D}

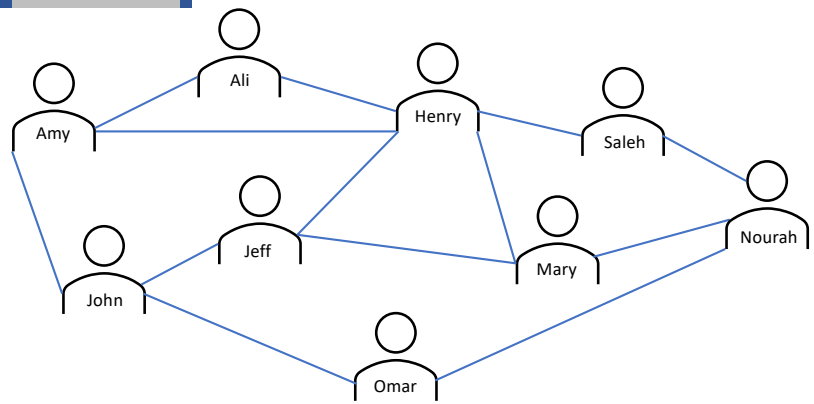
- Edges in graphs can be directed or undirected



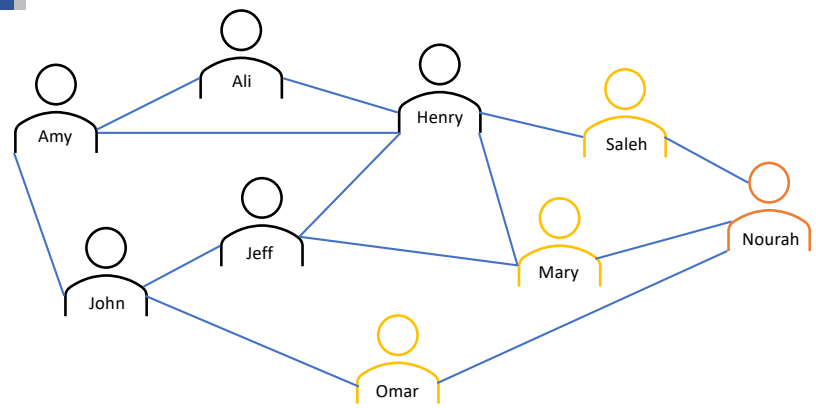


- E = { {A, C}, {A, B}, {B, C}, {B,D} }

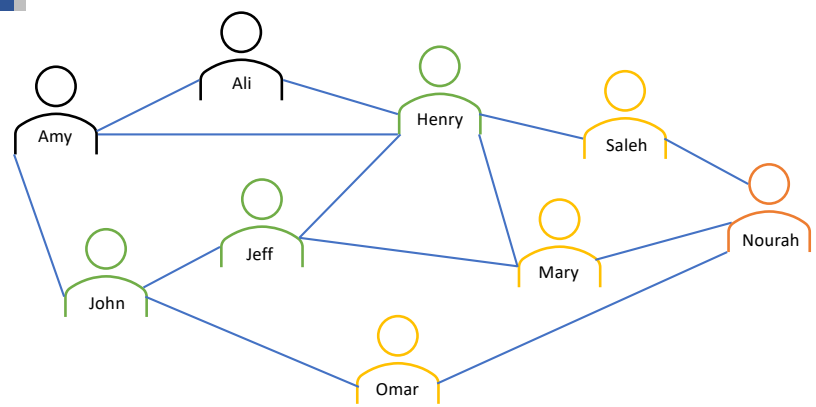
Graphs Applications



Graphs



Graphs

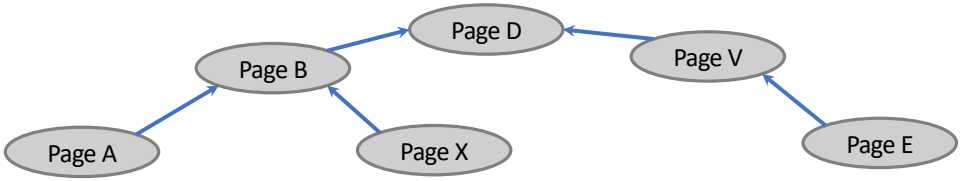


This problem can be described as "finding a vertex having a shortest path of length 2 from vertex **Nourah**".

Graphs Applications

Interlinked web pages

- Webpage, A, can have a link to another webpage, B.
- It may not be necessary that B will also have a link to page A

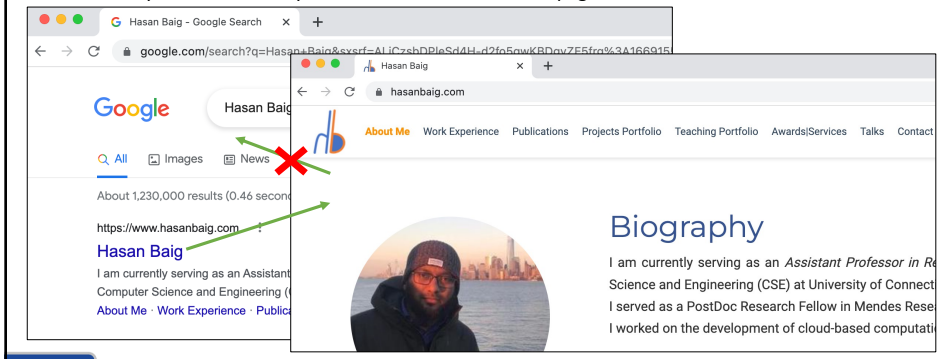


9

Graphs Applications

Interlinked web pages

- Webpage, A, can have a link to another webpage, B.
- It may not be necessary that B will also have a link to page A



10

Graphs Applications

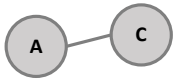
Interlinked web pages

- Webpage, A, can have a link to another webpage, B.
- It may not be necessary that B will also have a link to page A

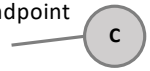
Web search engine as a directed graph
→ Web Crawlers

Graphs Terms

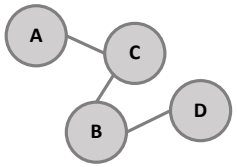
- **Adjacent Vertices**
  - Two vertices, u and v, connected together via the same edge



- An edge is said to be **incident** to a vertex if the vertex is one of the edge's endpoint

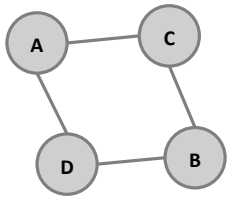


- A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.



Graphs Terms

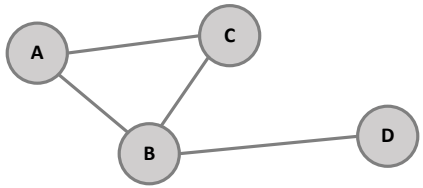
- **Cycle** is a path that starts and ends at the same vertex and includes at least one edge.



Graphs Types

Some important types of graphs are:

1. **Undirected Graphs**
   - Edges do not have a direction
   - Edge (u, v) will be equivalent to the edge (v, u)

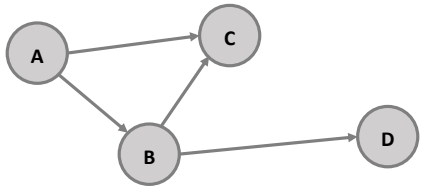


Graphs Types

Some important types of graphs are:

2. Directed Graphs or Digraphs

- Edges have a direction pointing from one vertex to another vertex

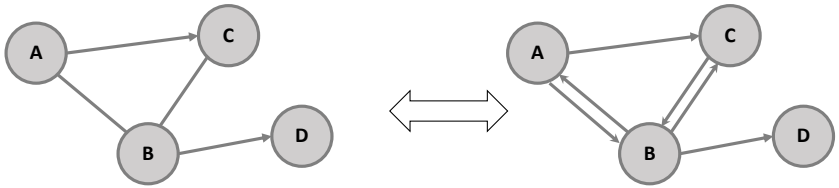


Graphs Types

Some important types of graphs are:

3. Mixed Graphs

- Edges have a direction pointing from one vertex to another vertex
- Some edges are undirected

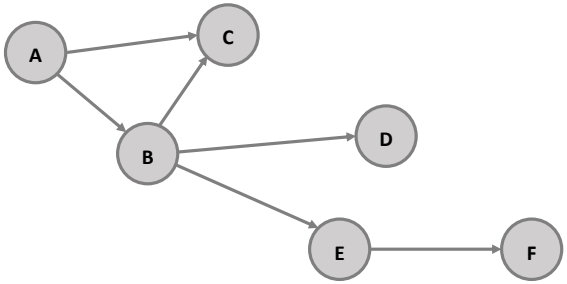


Graphs Types

Some important types of graphs are:

4. Directed Acyclic Graphs

- Finite directed graph with no cycles

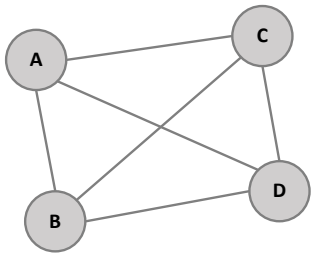


Graphs Types

Some important types of graphs are:

5. Complete Graphs

- All vertices are connected to each other

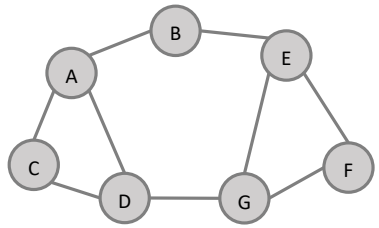


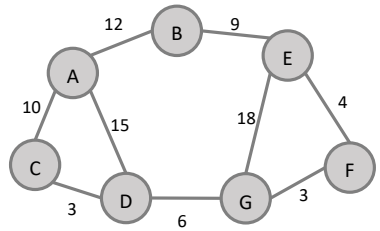
Graphs Types

Some important types of graphs are:

6. Weighted vs unweighted graphs

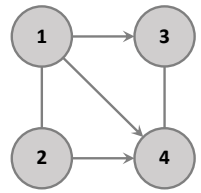
- Unweighted graphs: all edges are of equal weight ~ 1
- Weighted graphs: have different values associated to edges





Graphs Representation

Representation of a graph, G(V,E), in a programming language.



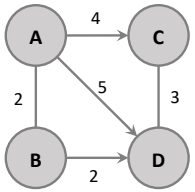
1. Edge Set → Stores a set of vertices and a set of edges
   V = {1, 2, 3, 4}
   E = { (1, 2), (1, 3), (1, 4),
   (2, 1), (2, 4),
   (3,4), (4,3) }

2. Adjacency Set → Stores a set of vertices and a dictionary of neighbors
   V = {1, 2, 3, 4}
   nbrs = { 1: {2, 3, 4},
   2: {1, 4},
   3: {4},
   4: {3}, }

Graphs Representation

For weighted graphs

V = {A, B, C, D}
nbrs = { A: { (2, B), (4, C), (5, D) },
B: { (2, A), (2, D) },
C: { (3, D) },
D: { (3, C) },
}

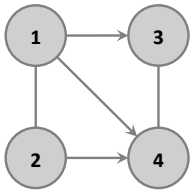


Graphs ADT

- `__init__(V, E)` → creates a graph with vertex and edge sets V and E
- `add_vertex(v)` → add vertex v to graph
- `remove_vertex(v)` → remove vertex v from graph
- `add_edge(e)` → add edge e to graph (assuming e is a 2-tuple)
- `remove_edge(e)` → removes edge e from graph (assuming e is a 2-tuple)
- `neighbors(v)` → returns iterable collection of v's neighbors

Graphs Implementation

Edge Set Implementation



V = {1, 2, 3, 4}
E = { (1, 2), (1, 3), (1, 4),
(2, 1), (2, 4),
(3,4), (4,3)
}

```
class EdgeSet:
    def __init__(self, V = None, E = None):
        self.V = set()
        self.E = set()

        if V is not None:
            for v in V:
                self.add_vertex(v)

        if E is not None:
            for e in E:
                self.add_edge(e)
```

```
    def add_vertex(self, v):
        self.V.add(v)

    def remove_vertex(self, v):
        if v not in self.V:
            raise KeyError(f"No vertex {v} exist.")
        self.V.remove(v)
```

```
    def add_edge(self, e):
        self.E.add(e)

    def remove_edge(self, e):
        if e not in self.E:
            raise KeyError(f"No edge {e} exist.")
        self.E.remove(e)
```

```
    def neighbors(self, v):
        nbrs = set()
        for u, w in self.E:
            if u == v:
                nbrs.add(w)
        return nbrs
```


Graphs Implementation

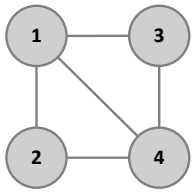
Adjacency Set Implementation

```
class AdjacencySet:
    def __init__(self, V = None, E = None):
        self.V = set()
        self.nbrs = dict()

        if V is not None:
            for v in V:
                self.add_vertex(v)
        if E is not None:
            for e in E:
                self.add_edge(e)
```

```
    def add_vertex(self, v):
        self.V.add(v)

    def remove_vertex(self, v):
        if v not in self.V:
            raise KeyError(f"No vertex {v} exists.")
        self.V.remove(v)
```



Edge Set
V = {1, 2, 3, 4}
E = { (1, 2), (1, 3), (1, 4),
(2, 1), (2, 4),
(3,4), (4,3)
}

Activity # 1

Complete AdjacencySet class by implementing `add_edge()` and `remove_edge()` methods.

```
class AdjacencySet:
    def __init__(self, V = None, E = None):
        self.V = set()
        self.nbrs = dict()

        if V is not None:
            for v in V:
                self.add_vertex(v)
        if E is not None:
            for e in E:
                self.add_edge(e)
```

```
    def add_vertex(self, v):
        self.V.add(v)

    def remove_vertex(self, v):
        if v not in self.V:
            raise KeyError(f"No vertex {v} exists.")
        self.V.remove(v)
```

V = {1, 2, 3, 4}
E = { (1, 2), (1, 3), (1, 4), (2, 1), (2, 4), (3,4), (4,3) }

G = AdjacencySet(V, E)

Output
print(G.nbrs) ➔
{ 1: {2, 3, 4}, 2: {1, 4}, 3: {4}, 4: {3}, }

```
def add_edge(self, e):
```

```
def remove_edge(self, e):
```


Activity # 1 Solution

Complete AdjacencySet class by implementing `add_edge()` and `remove_edge()` methods.

```
class AdjacencySet:
    def __init__(self, V = None, E = None):
        self.V = set()
        self.nbrs = dict()

        if V is not None:
            for v in V:
                self.add_vertex(v)
        if E is not None:
            for e in E:
                self.add_edge(e)
```

```
    def add_vertex(self, v):
        self.V.add(v)

    def remove_vertex(self, v):
        if v not in self.V:
            raise KeyError(f"No vertex {v} exists.")
        self.V.remove(v)
```

V = {1, 2, 3, 4}
E = { (1, 2), (1, 3), (1, 4), (2, 1), (2, 4), (3,4), (4,3) }

G = AdjacencySet(V, E)

Output
print(G.nbrs) ➔
{ 1: {2, 3, 4}, 2: {1, 4}, 3: {4}, 4: {3}, }

```
def add_edge(self, e):
    u, v = e
    if u not in self.nbrs:
        self.nbrs[u] = {v}
    else:
        self.nbrs[u].add(v)

def remove_edge(self, e):
    u, v = e
    if v not in self.nbrs[u]:
        raise KeyError(f"Vertex {u} has no neighbor {v}")
    self.nbrs[u].remove(v)
    if len(self.nbrs[u]) == 0:  #if there exist no neighbor
        self.nbrs.pop(u)
```

UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

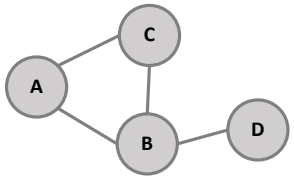
Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

27

Quick Recap

Graphs

- Edges can connect any vertex
- Any vertex can be accessed through any path

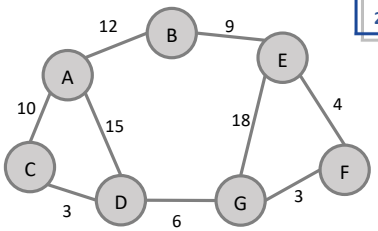


- Edges can be directed or undirected

- G = (V, E)
  - V = {A, B, C, D}
  - E = { {A, C}, {A, B}, {B, C}, {B,D} }

Quick Recap

Graphs Terms and Types
- Cycle, Path,
- Directed, undirected, mixed, complete, weighted

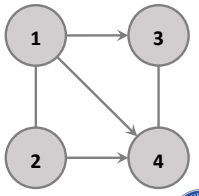


- Representation
  - Edge Set Representation

    V = {1, 2, 3, 4}
    E = { (1, 2), (1, 3), (1, 4), (2, 1), (2, 4), (3,4), (4,3) }

  - Adjacency Set Representation

    V = {1, 2, 3, 4}
    nbrs = { 1: {2, 3, 4}, 2: {1, 4}, 3: {4}, 4: {3}, }

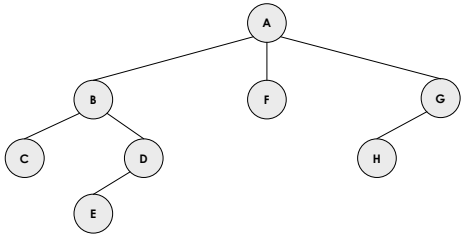


  - Implementation

Graph Traversal

Purpose of graph traversal is to visit all the nodes/vertices of a graph

- Breadth-First Search (BFS)
  ➔ Visit order → A, B, F, G, C, D, H, E



- Depth-First Search (DFS)
  ➔ Visit order → A, B, C, D, E, F, G, H

Graph Traversal | Breadth-First Search (BFS)

- Visit the neighbors then the neighbors of these new vertices and so on
- Running time complexity: O(V+E)
- Memory complexity is not good: Store many references
  → DFS is usually preferred
- It constructs a shortest path for unweighted graphs (smaller number of edges)

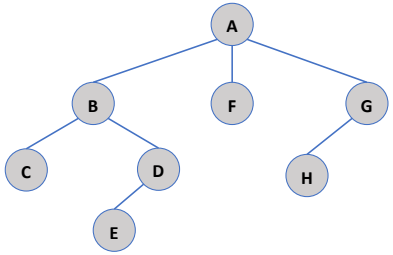
Graph Traversal | Breadth-First Search (BFS)

Iterative Implementation

- Maintain a Queue, Q
- Check if the node is visited

1. Start off with any vertex by enqueuing it and set it to visited
2. Keep checking until queue is empty
   a. Dequeue a vertex from front of the queue
   b. Enqueue the neighbors of dequeued vertex

Graph Traversal | Breadth-First Search (BFS)



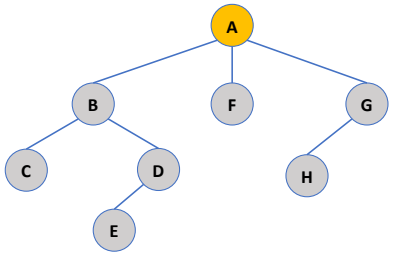
Operations

Q

V

33

Graph Traversal | Breadth-First Search (BFS)



Enqueue first vertex
Visited

Q

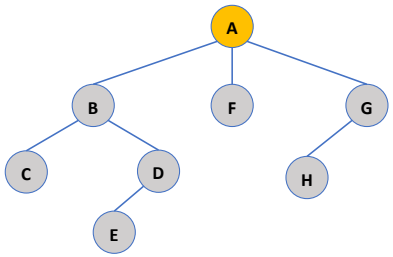
A

Add first vertex in the visited list

V = [A]

34

Graph Traversal | Breadth-First Search (BFS)



Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

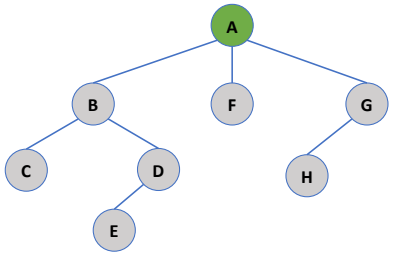
Q

A

V = [A]

35

Graph Traversal | Breadth-First Search (BFS)



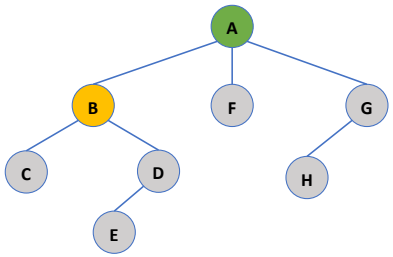
Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q

V = [A]

36

Graph Traversal — Breadth-First Search (BFS)



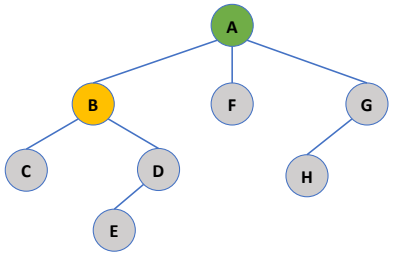
Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B

V = [A]

37

Graph Traversal — Breadth-First Search (BFS)



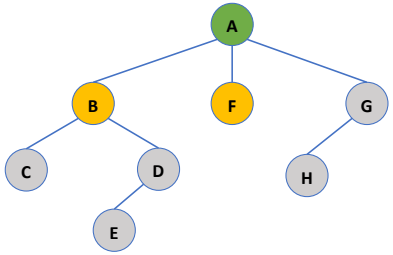
Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B

V = [A, B]

38

Graph Traversal — Breadth-First Search (BFS)

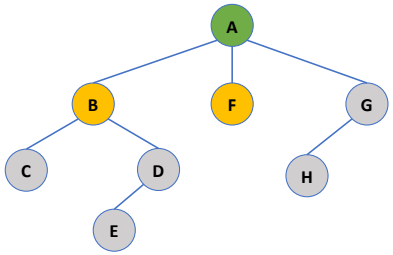


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B F

V = [A, B]

Graph Traversal — Breadth-First Search (BFS)

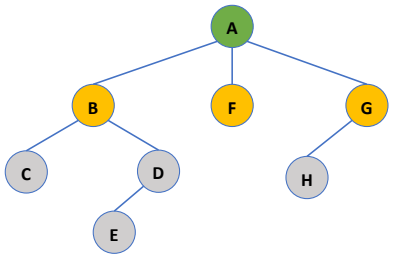


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B F

V = [A, B, F]

Graph Traversal — Breadth-First Search (BFS)

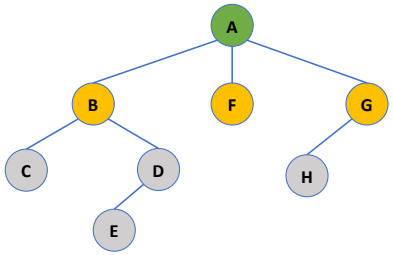


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B F G

V = [A, B, F]

Graph Traversal — Breadth-First Search (BFS)

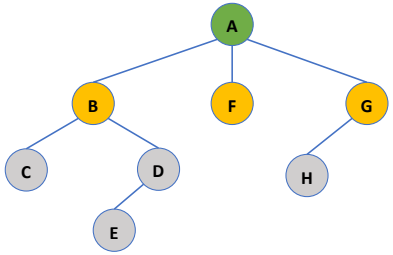


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of A
Add in V

Q
B F G

V = [A, B, F, G]

Graph Traversal — Breadth-First Search (BFS)

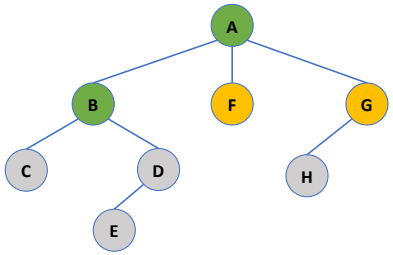


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
B F G

V = [A, B, F, G]

Graph Traversal — Breadth-First Search (BFS)

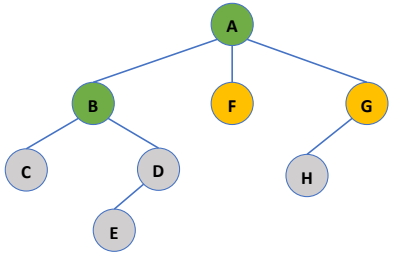


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
F G

V = [A, B, F, G]

Graph Traversal — Breadth-First Search (BFS)

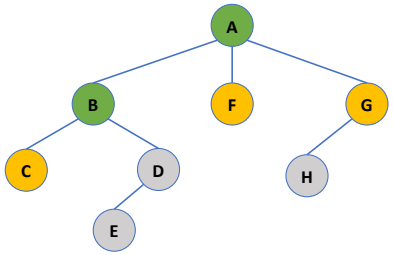


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of B
Add in V

Q
F G

V = [A, B, F, G]

Graph Traversal — Breadth-First Search (BFS)

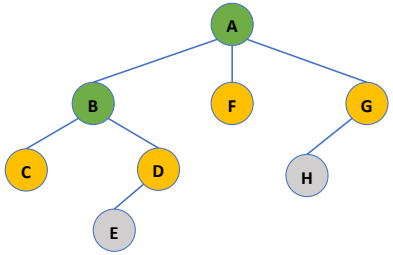


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of B
Add in V

Q
F G C

V = [A, B, F, G, C]

Graph Traversal — Breadth-First Search (BFS)

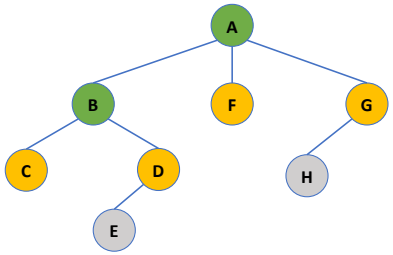


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of B
Add in V

Q
F G C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

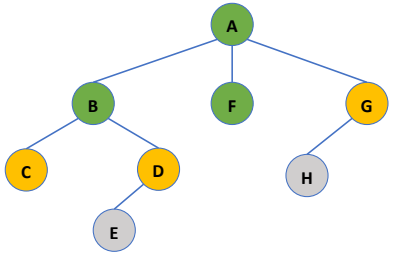


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
F G C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

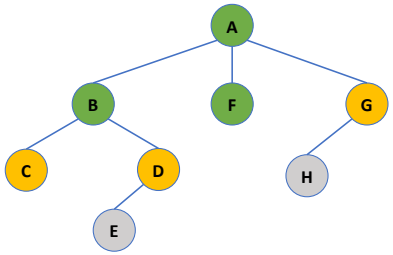


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
G C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

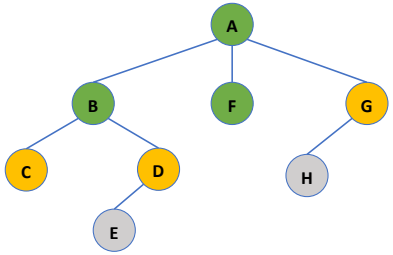


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of F
Add in V

Q
G C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

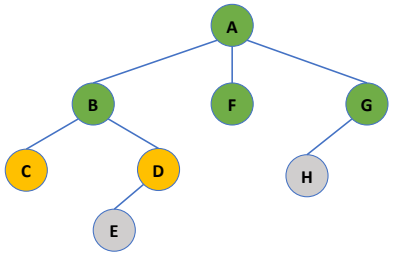


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
G C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

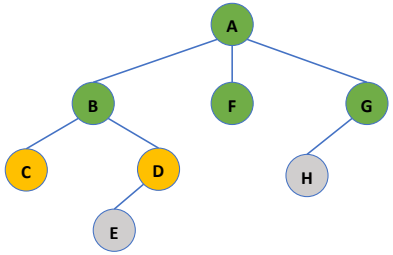


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor
Add in V

Q
C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

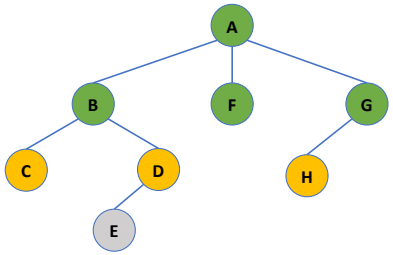


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of G
Add in V

Q
C D

V = [A, B, F, G, C, D]

Graph Traversal — Breadth-First Search (BFS)

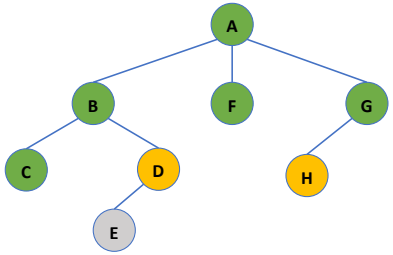


Is Q empty? → No
Dequeue
If not in V, Enqueue neighbor of G
Add in V

Q
C D H

V = [A, B, F, G, C, D, H]

Graph Traversal — Breadth-First Search (BFS)



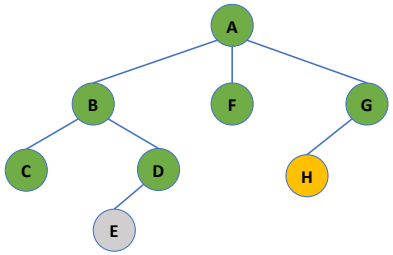
Q

D H

V = [A, B, F, G, C, D, H]

55

Graph Traversal — Breadth-First Search (BFS)



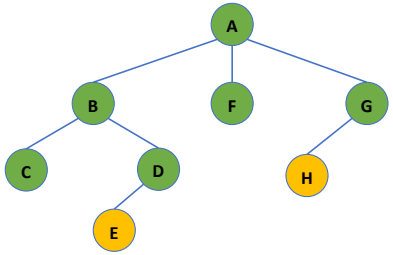
Q

H

V = [A, B, F, G, C, D, H]

56

Graph Traversal — Breadth-First Search (BFS)



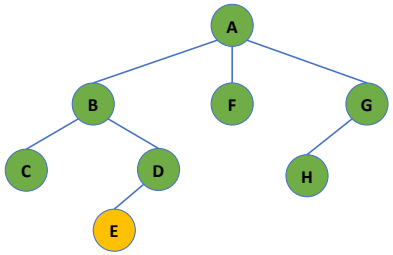
Q

H E

V = [A, B, F, G, C, D, H, E]

57

Graph Traversal — Breadth-First Search (BFS)



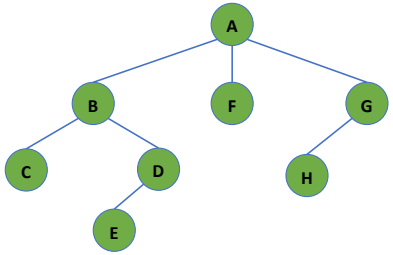
Q

E

V = [A, B, F, G, C, D, H, E]

58

Graph Traversal | Breadth-First Search (BFS)



Q _____

_____

V = [A, B, F, G, C, D, H, E]

Graph Traversal | Depth-First Search (DFS)

- It explores as far as possible along each branch
- Running time complexity: O(V+E)
- Memory complexity is better than BFS
- It is used to solve mazes

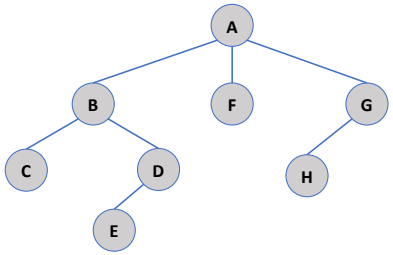
Graph Traversal | Depth-First Search (DFS)

Iterative Implementation

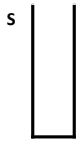
- Maintain a Stack, S
- Check if the node is visited

1. Start off with any vertex by pushing it on the stack
2. Keep checking until stack is empty
   a. Pop vertex from the top and mark it as visited
   b. Push the neighbors of popped vertex

Graph Traversal | Depth-First Search (DFS)



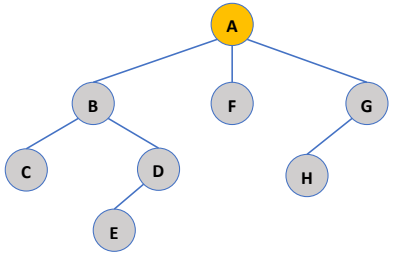
Operations



Add first vertex in the visited list

V

Graph Traversal | Depth-First Search (DFS)



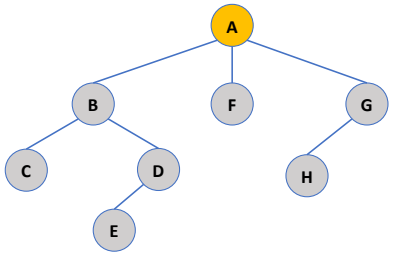
Push first vertex

S

V

63

Graph Traversal | Depth-First Search (DFS)



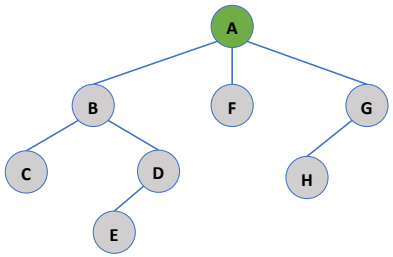
Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S

V

64

Graph Traversal | Depth-First Search (DFS)



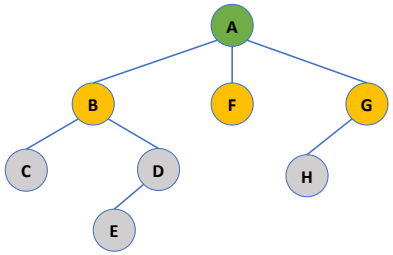
Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S

V = [A]

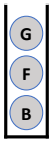
65

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S



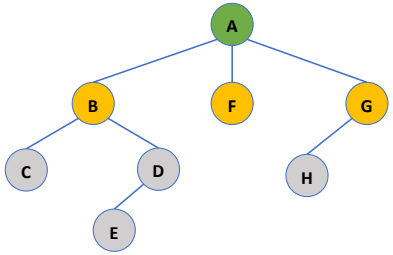
V = [A]

66

CSE-2050 – Data Structures and Object-Oriented Design

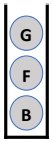
Graph Traversal | Depth-First Search (DFS)

67



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S



V = [A]

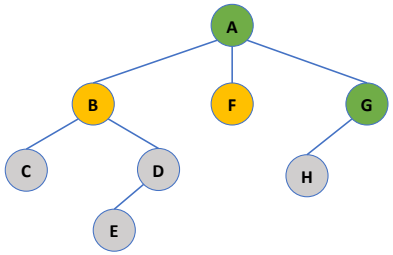
Hasan Baig

67

CSE-2050 – Data Structures and Object-Oriented Design

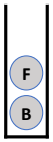
Graph Traversal | Depth-First Search (DFS)

68



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S

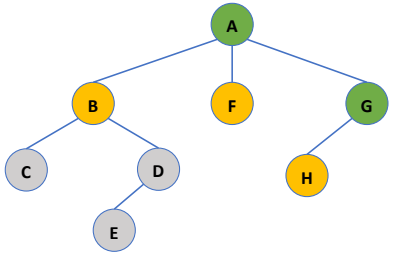


V = [A, G]

Hasan Baig

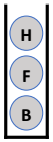
68

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

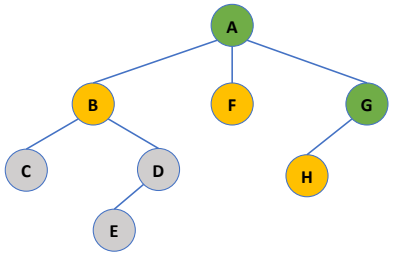
S



V = [A, G]

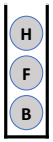
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S



V = [A, G]

Graph Traversal | Depth-First Search (DFS)

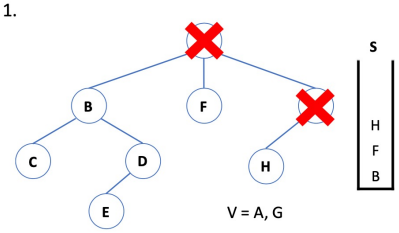
Complete it yourself

In the graph shown, vertices marked as X are visited according to Depth-First Search Approach. Consider the current state of stack, S, and visited list, V, in step 1, indicate in the next steps how the S and V are updated. Mark the visited vertices with X on the graph. The instructions that must be followed in each step are also given in the first box.

Instructions

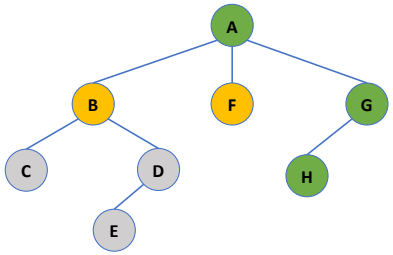
a. Check if **S** is empty? If NO, follow the steps below, else stop execution.
b. Pop a vertex from the stack S.
c. If neighbors of popped vertex is not in **V**, Push neighbor(s) in the stack in alphabetical order.
d. Add the pushed vertices in list V.

1.

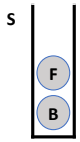


V = A, G

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

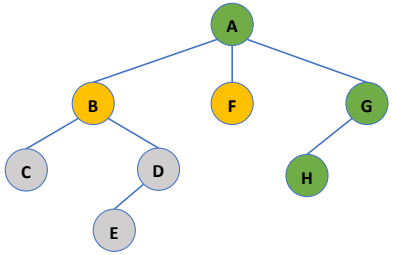


V = [A, G, H]

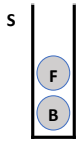
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

73



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H]

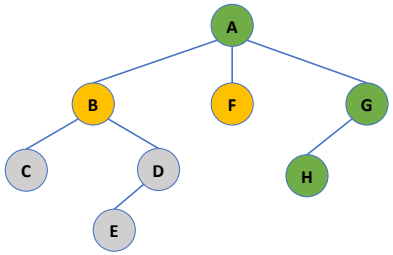
Hasan Baig

73

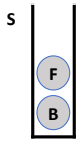
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

74



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H]

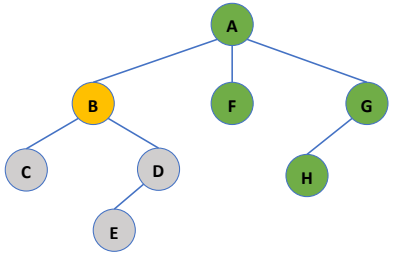
Hasan Baig

74

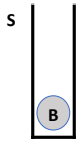
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

75



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H, F]

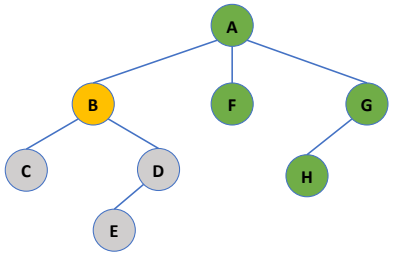
Hasan Baig

75

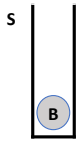
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

76



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H, F]

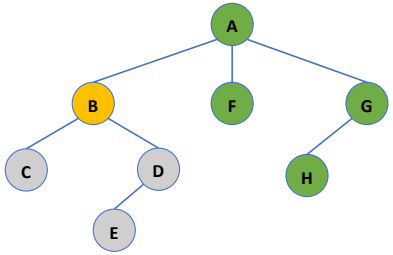
Hasan Baig

76

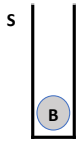
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

77



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H, F]

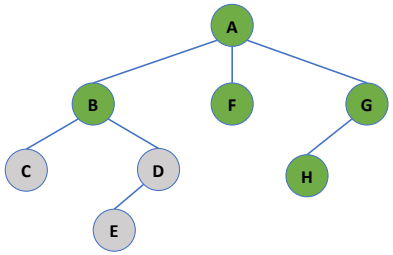
Hasan Baig

77

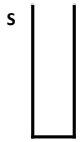
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

78



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

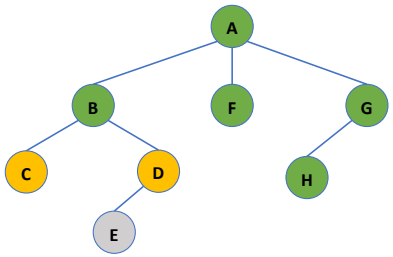


V = [A, G, H, F, B]

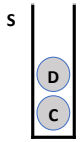
Hasan Baig

78

Graph Traversal | Depth-First Search (DFS)



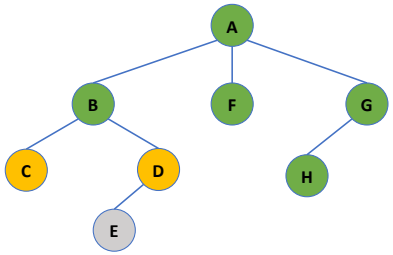
Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



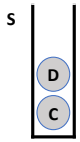
V = [A, G, H, F, B]

CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

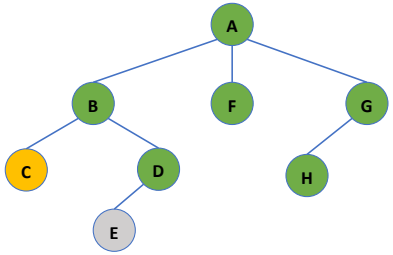


Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

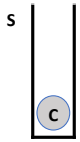


V = [A, G, H, F, B]

Graph Traversal | Depth-First Search (DFS)



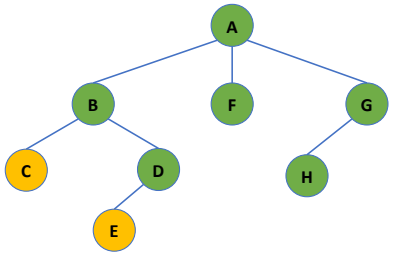
Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



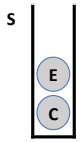
V = [A, G, H, F, B, D]

CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

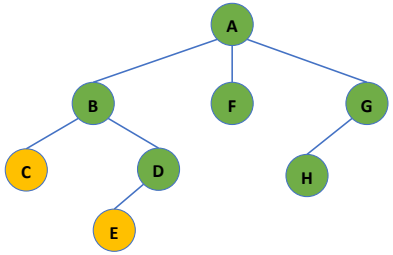


V = [A, G, H, F, B, D]

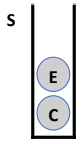
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

83



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H, F, B, D]

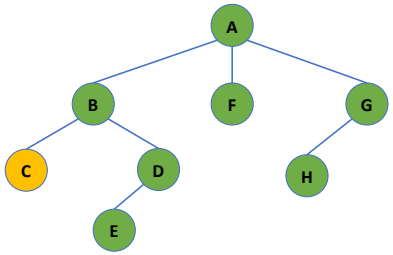
Hasan Baig

83

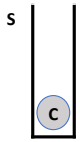
CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

84



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

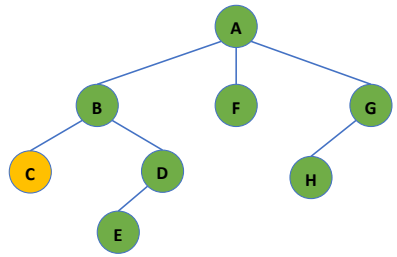


V = [A, G, H, F, B, D, E]

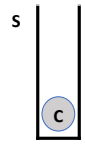
Hasan Baig

84

Graph Traversal | Depth-First Search (DFS)



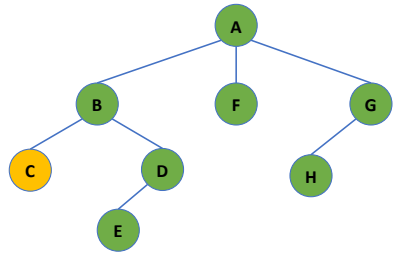
Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



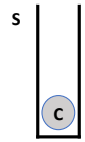
V = [A, G, H, F, B, D, E]

CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

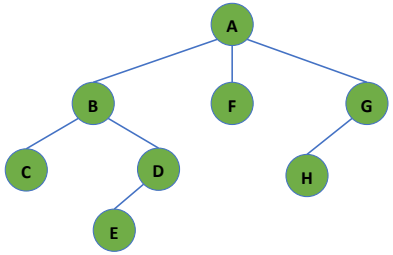


V = [A, G, H, F, B, D, E]

CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

87



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

S

V = [A, G, H, F, B, D, E, C]

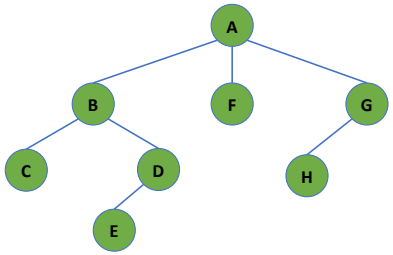
Hasan Baig

87

CSE-2050 – Data Structures and Object-Oriented Design

Graph Traversal | Depth-First Search (DFS)

88



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)

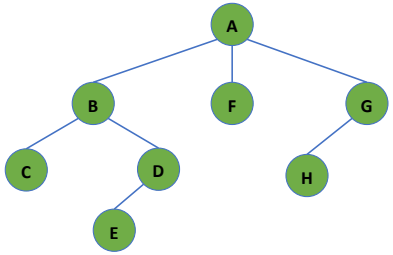
S

V = [A, G, H, F, B, D, E, C]

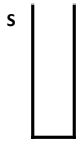
Hasan Baig

88

Graph Traversal | Depth-First Search (DFS)



Is S empty? → No
Pop
Add in V
If not in V, Push neighbor(s)



V = [A, G, H, F, B, D, E, C]

BFS vs DFS

BFS	DFS

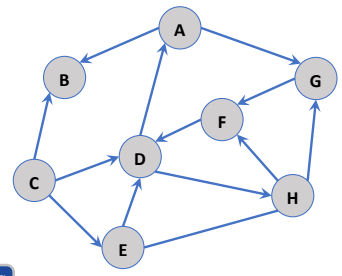
• Search breadth-wise
  • Walk through all nodes on the same level
	• Search depth-wise
	  • Goes as deep as possible until there is no neighbor left

• Based on FIFO (queues)	• Based on LIFO (stacks)
• Suitable for searching nearby vertices	• Suitable for searching vertices farther away from the source
• No concept of backtracking	• Can recursively backtrack

Activity # 2

Using BFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is the front of Q.
After dequeuing a vertex, enqueue (if not already in V) all its neighbors in Q in alphabetical order. Add them in V.

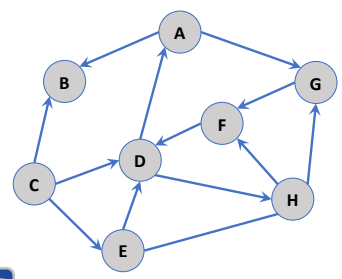


	Q	V
0	[C]	[C]
1	[B, D, E]	[C, B, D, E]
2		
3		
4		
5		
6		
7		
8		
9		
10		

Activity # 2 Solution

Using BFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is the front of Q.
After dequeuing a vertex, enqueue (if not already in V) all its neighbors in Q in alphabetical order. Add them in V.



	Q	V
0	[C]	[C]
1	[B, D, E]	[C, B, D, E]
2	[D, E]	[C, B, D, E]
3	[E, A, H]	[C, B, D, E, A, H]
4	[A, H]	[C, B, D, E, A, H]
5	[H, G]	[C, B, D, E, A, H, G]
6	[G, F]	[C, B, D, E, A, H, G, F]
7	[F]	[C, B, D, E, A, H, G, F]
8	[]	[C, B, D, E, A, H, G, F]
9		
10		

CSE-2050 – Data Structures and Object-Oriented Design

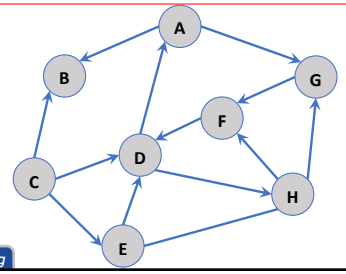
93

Activity # 3

Using DFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is Top of the stack

After popping a vertex out, push its neighbors (if not already in V) in S in alphabetical order. Also, add popped element in V.



	S	V
0	[C]	[ ]
1	[B, D, E]	[C]
2		
3		
4		
5		
6		
7		
8		
9		
10		

Hasan Baig

93

CSE-2050 – Data Structures and Object-Oriented Design

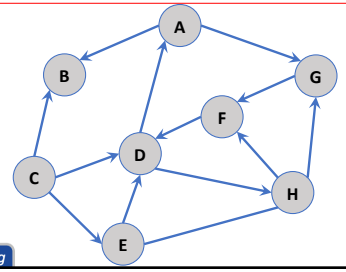
94

Activity # 3 Solution

Using DFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is Top of the stack

After popping a vertex out, push its neighbors (if not already in V) in S in alphabetical order. Also, add popped element in V.



	S	V
0	[C]	[ ]
1	[B, D, E]	[C]
2	[D, E]	[C, B]
3	[A, H, E]	[C, B, D]
4	[G, H, E]	[C, B, D, A]
5	[F, H, E]	[C, B, D, A, G]
6	[H, E]	[C, B, D, A, G, F]
7	[E]	[C, B, D, A, G, F, H]
8	[]	[C, B, D, A, G, F, H, E]
9		
10		

Hasan Baig

94