DATA STRUCTURES

UNIT-4

CLASS NOTES

GRAPHS

feedback/corrections: vibha@pesu.pes.edu

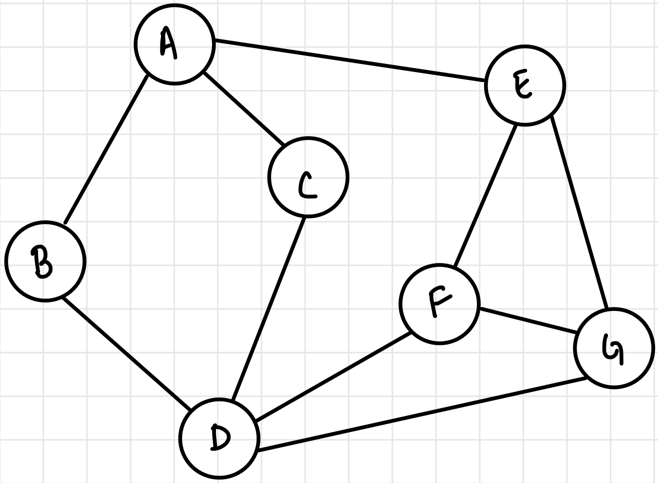
Vibha Masti

GRAPHS

- Non-linear data structure
- Set of vertices and edges
- Set of edges represents the relationship between vertices
- A graph G is defined as

$$G = (V, E)$$

V: set of vertices
E: set of edges

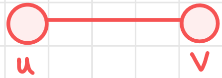


$V = \{A, B, C, D, E, F, G\}$

$E = \{(A,B), (A,C), (B,D), (C,D), (A,E), (E,F), (E,G), (F,G), (F,D), (G,D)\}$

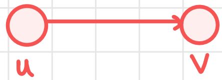
Undirected Graph

- pair of vertices representing an edge is unordered



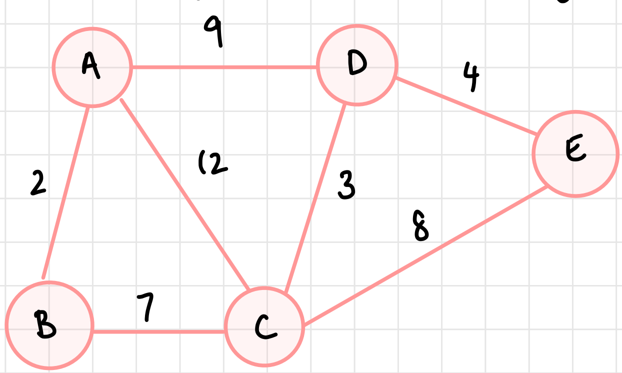
Directed Graph

- edges are directed (order matters)



Weighted Graph

- Each edge has a numerical value attached to it called weight. Eg: distance, difficulty



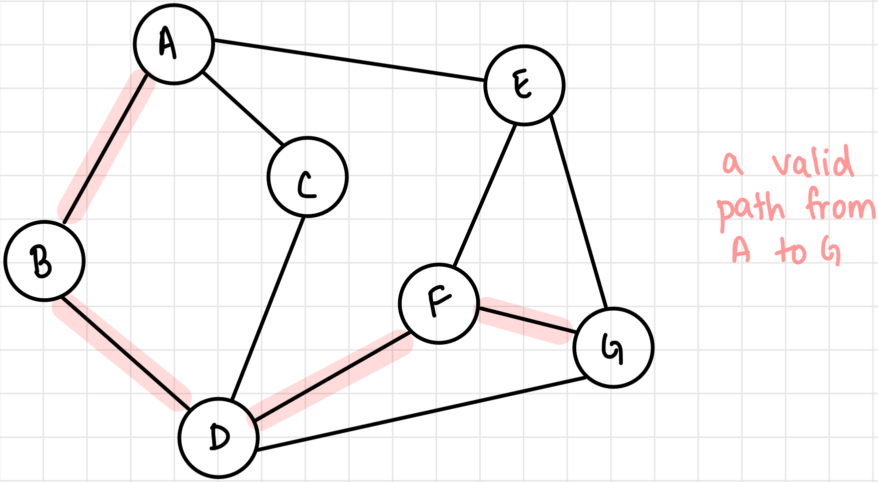
Adjacent Nodes

- Two nodes are adjacent to each other if there exists an edge connecting the two

- If the graphs are directed, the nodes are each others' successor and predecessor

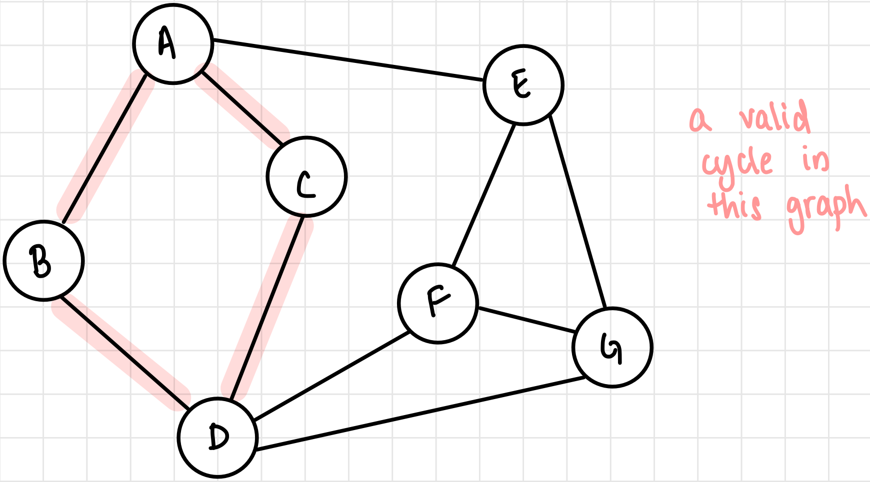
Path

- Sequence of vertices that connect two nodes in a graph



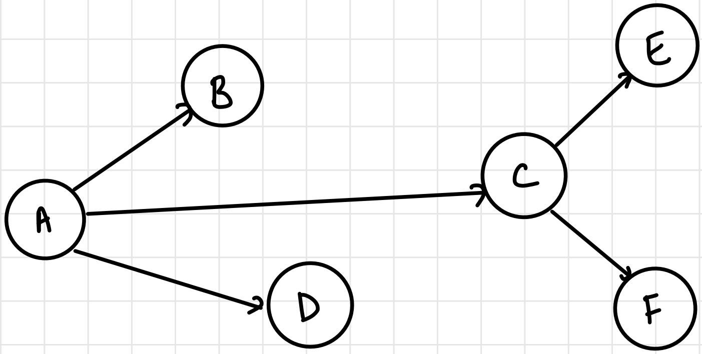
cycle

- Path that starts and ends on same node
- Graphs with at least one cycle are called cyclic graphs and graphs without any cycles are called acyclic



Acyclic graph

- Trees are acyclic

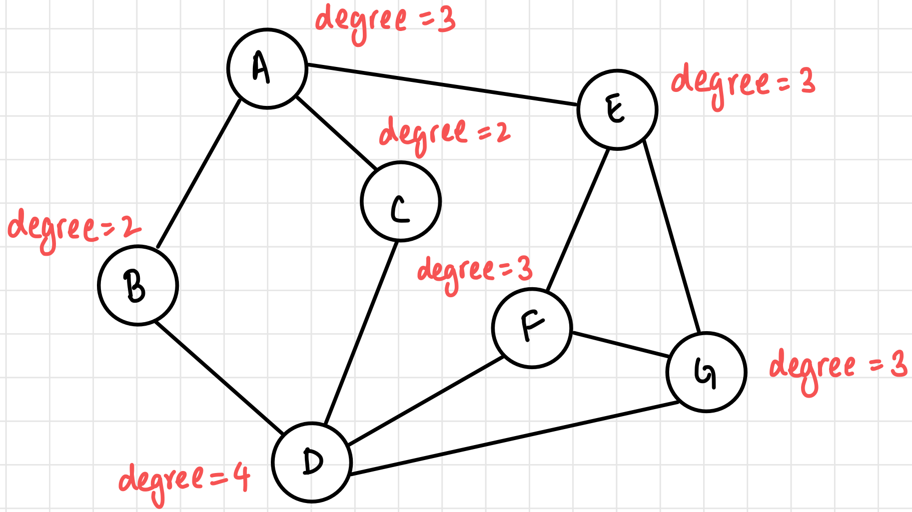


Incident

- A node is incident to an edge if the edge connects that node to another node

Degree

- The degree of a vertex (node) is the number of edges incident on it



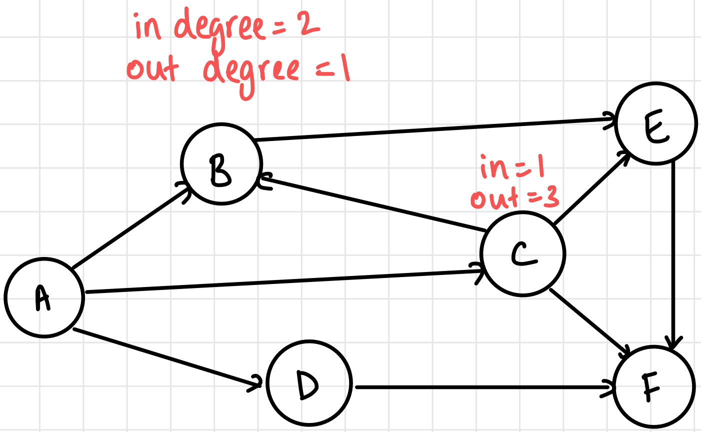
For Directed Graphs

In-degree

- number of edges incident to a vertex

Out-degree

- number of edges incident from a vertex



DIRECTED GRAPH

- Number of possible pairs in an m-vertex graph is $m(m-1)$ [assumming no self connections]

- Number of edges in a directed graph is $\leq m(m-1)$ as the edge $(i,j) \neq$ edge (j,i)

UNDIRECTED GRAPHS

- Number of possible pairs in an m-vertex graph is $m(m-1)$ [assumming no self connections]
- Number of edges in a directed graph is $\leq m(m-1)/2$ as the edge (i,j) = edge (j,i)

REPRESENTATION of GRAPHS

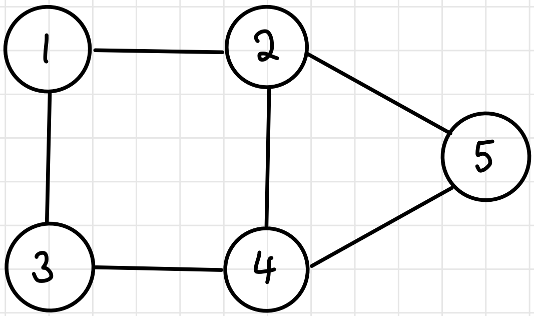
- Information required to represent a graph: set of vertices and their edges
- Depending on density of edges, use and operations performed, graphs are represented in one of two ways

1. Adjacency Matrix
   - 2D-array

2. Adjacency List
   - Linked list

Adjacency Matrix

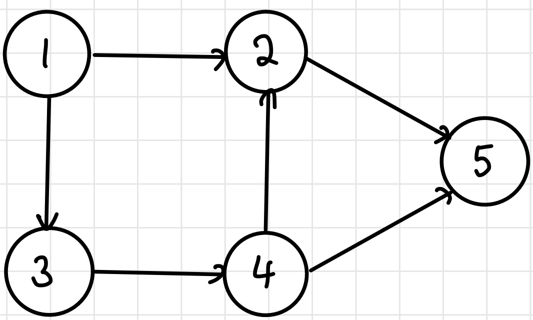
- n x n matrix M
- M[i][j] = 1 if (i, j) is an edge
- M[i][j] = 0 if (i, j) is not an edge
- For undirected graphs, matrix M is symmetric and M[i][j] = M[j][i]
- Assume no node is connected to itself (all diagonals 0)



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

diagonals are all 0

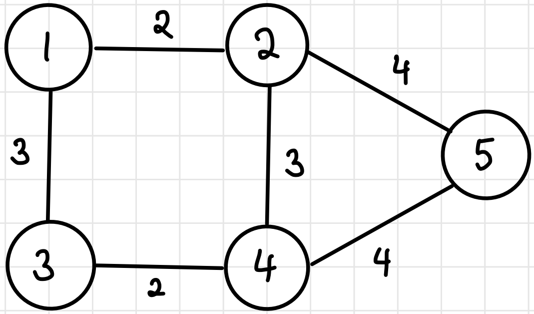
Undirected graph



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

diagonals are all 0

Directed graph



Weighted, undirected graph

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

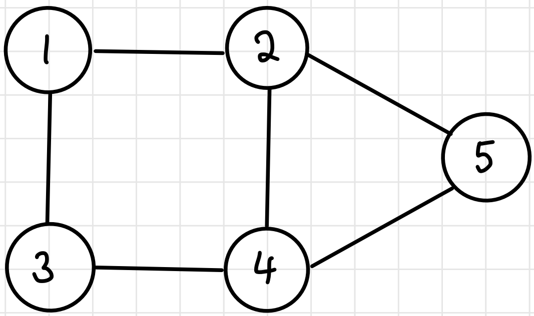
↙ diagonals are all 0

Drawbacks of Adjacency Matrix

- Number of nodes in the graph should be known prior to creation
- To detect presence of edge takes O(1) but to visit all neighbouring nodes takes O(n^2) TODO
- Can become sparse if there are few edges
- Space complexity is O(n^2) ⟶ n^2 locations needed

Adjacency List

- Each node maintains a linked list of its neighbours



array of pointers

1 → 2 → 3 → NULL
2 → 1 → 4 → 5 → NULL
3 → 1 → 4 → NULL
4 → 2 → 3 → 5 → NULL
5 → 2 → 4 → NULL

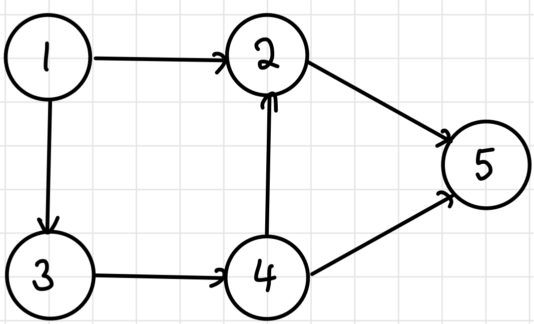
order does not matter

Undirected graph

node 2 → node 1
node 2 → node 4
node 2 → node 5

degree of a vertex: number of elements in linked list of that vertex

degree of 2 : 3



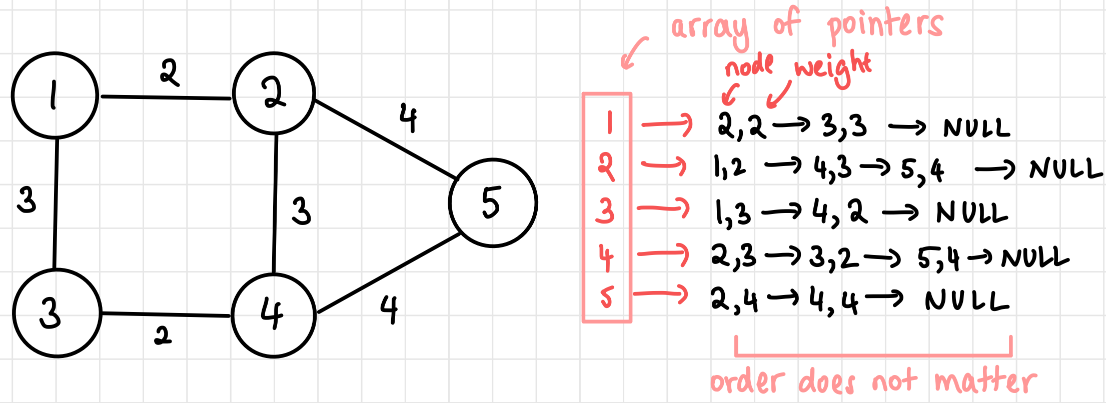
array of pointers

1 → 2 → 3 → NULL
2 → 5 → NULL
3 → 4 → NULL
4 → 2 → 5 → NULL
5 → NULL

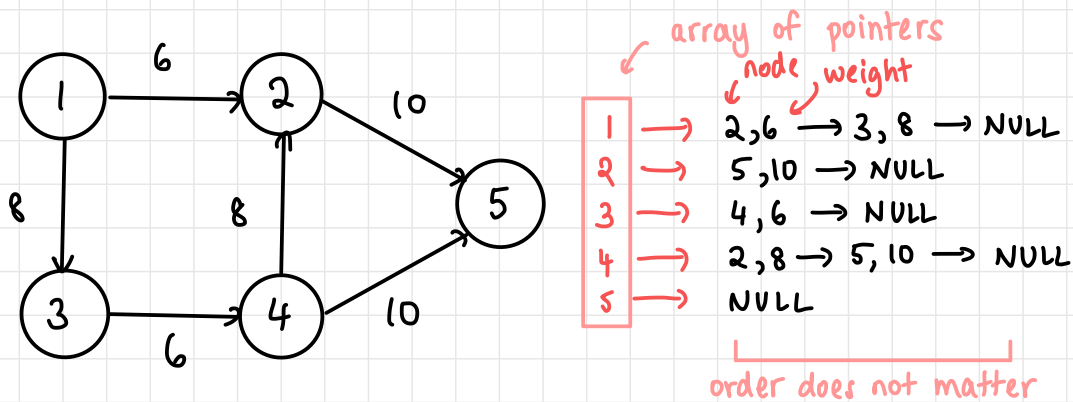
order does not matter

Directed graph

node 2 → node 5



Weighted, undirected graph



Weighted, directed graph

Data Structure representing Adjacency matrix

```c
#define MAX_NODES 50

typedef struct graph {
    int n; /* Number of vertices in graph */
    int adj[MAX_NODES][MAX_NODES]; /* Adjacency matrix */
} Graph;
```

Code Implementation

- create a graph

```
void create_graph(Graph *adj_mat) {
    int i, j;

    for (int i = 0; i < adj_mat->n; ++i) {
        for (int j = 0; j < adj_mat->n; ++j) {
            adj_mat->adj[i][j] = 0;
        }
    }

    while (1) {
        printf("Enter source and destination vertices: ");
        scanf("%d %d", &i, &j);

        if (i < 0 && j <= 0 || i >= adj_mat->n || j >= adj_mat->n) {
            break;
        }

        adj_mat->adj[i][j] = 1;
    }
}
```

- Find the indegree of a node

```
int indegree(Graph *adj_mat, int v) {
    int count = 0;
    for (int i = 0; i < adj_mat->n; ++i) {
        if (adj_mat->adj[i][v] == 1) {
            ++count;
        }
    }
    return count;
}
```

- Find the Outdegree of a node

```
int outdegree(Graph *adj_mat, int v) {
    int count = 0;
    for (int j = 0; j < adj_mat->n; ++j) {
        if (adj_mat->adj[v][j] == 1) {
            ++count;
        }
    }
    return count;
}
```

Data Structure Representing Adjacency List

- use an array of node structures to represent multi-list

```
#define MAX_NODES 50

typedef struct node {
    int data; /* Value of the column of the connection */
    struct node *next;
} Node;


/* Inside main(), initialise the array of nodes */

Node *adj_list[MAX_NODES];
```

Code Implementation

- create a graph

```
void create_graph(Node *adj_list[], int n) {
    int i, j;

    for (int i = 0; i < n; ++i) {
        adj_list[i] = NULL;
    }

    while (1) {
        printf("Enter source and destination vertices: ");
        scanf("%d %d", &i, &j);

        if (i < 0 && j <= 0 || i >= n || j >= n) {
            break;
        }

        // Both for undirected
        insert(adj_list, i, j);
    }
}
```

- Find the Outdegree of a node

```
int outdegree(Node *adj_list[], int n, int v) {
    int count = 0;
    Node *traverse = adj_list[v];

    while (traverse != NULL) {
        ++count;
        traverse = traverse->next;
    }
    return count;
}
```

- Find the indegree of a node

```
int indegree(Node *adj_list[], int n, int v) {
    int count = 0;
    for (int i = 0; i < n; ++i) {
        Node *traverse = adj_list[i];

        while (traverse != NULL) {
            if (traverse->data == v) {
                ++count;
            }
            traverse = traverse->next;
        }
    }
    return count;
}
```

- insert helper function - insert to the end

```
void insert(Node *adj_list[], int i, int j) {
    Node *new_node = (Node *) malloc(sizeof(Node));
    new_node->next = NULL;
    new_node->data = j;

    Node *traverse = adj_list[i];

    if (traverse == NULL) {
        adj_list[i] = new_node;
        return;
    }

    while (traverse->next != NULL) {      <- insert to the end
        traverse = traverse->next;
    }
    traverse->next = new_node;
}
```

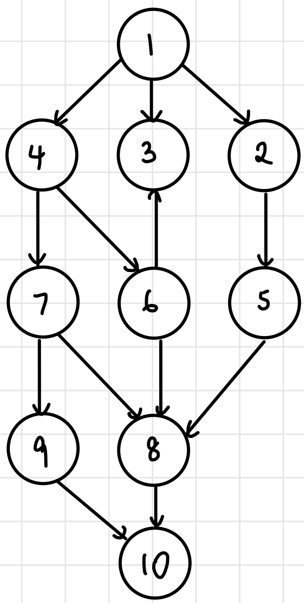
Example Output

```
Enter the number of vertices: 10
Enter source and destination vertices: 0 1
Enter source and destination vertices: 0 2
Enter source and destination vertices: 0 3
Enter source and destination vertices: 1 4
Enter source and destination vertices: 4 7
Enter source and destination vertices: 7 9
Enter source and destination vertices: 3 5
Enter source and destination vertices: 3 6
Enter source and destination vertices: 5 7
Enter source and destination vertices: 5 2
Enter source and destination vertices: 6 7
Enter source and destination vertices: 6 8
Enter source and destination vertices: 8 9
Enter source and destination vertices: -1 -1


MAIN MENU
1. Indegree of a vertex
2. Outdegree of a vertex
3. Display matrix
4. Exit
3
0->1 2 3
1->4
2->
3->5 6
4->7
5->7 2
6->7 8
7->9
8->9
9->
```

GRAPH TRAVERSAL

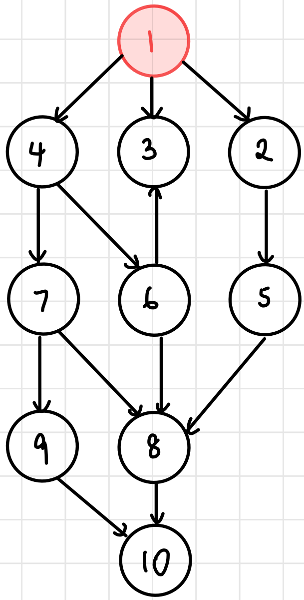
Depth First Search

- DFS — recursive function; can start at any node
- Nodes that have been visited are marked as visited nodes (stored in visited array)
- Analogous to preorder traversal: travels down the depth of one node before backtracking and continuing
- Uses a stack for implementation
- For both directed and undirected graphs
- Consider the following directed graph, starting at 1

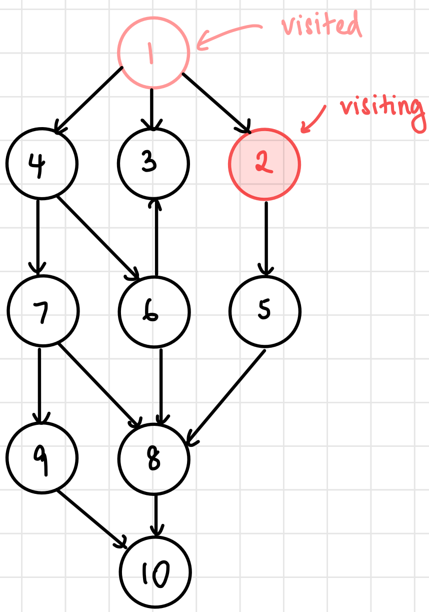


Traversal

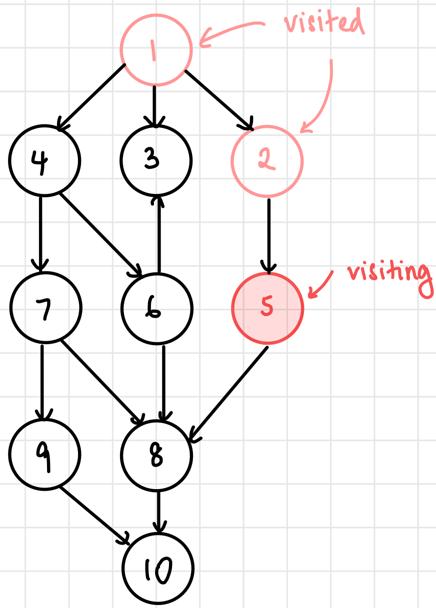
- visit 1, mark as visited



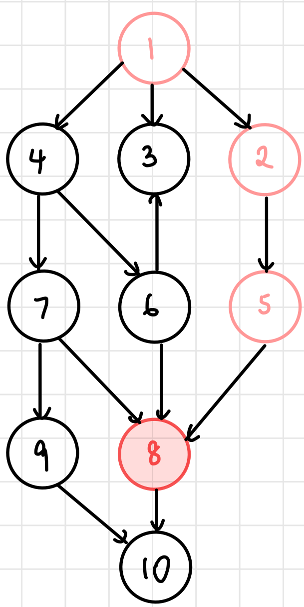
- visit 2, mark as visited



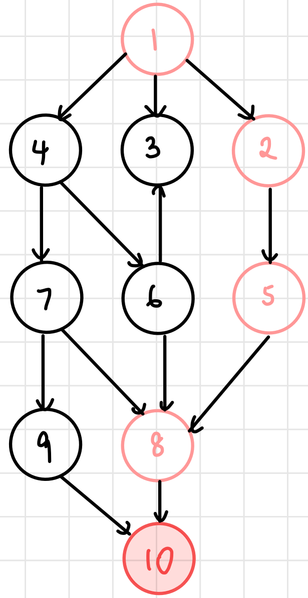
- visit 5, mark as visited



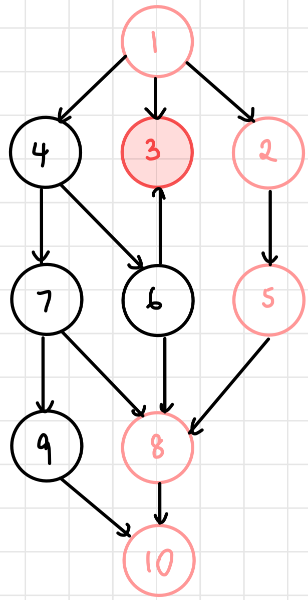
- visit 8, mark as visited



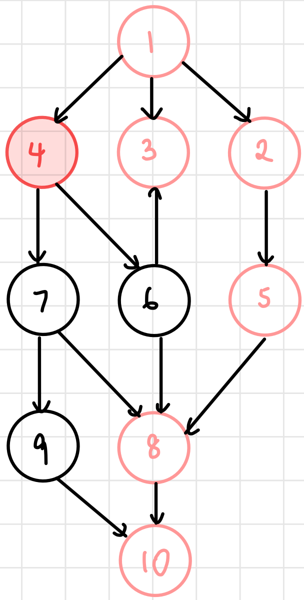
- visit 10, mark as visited



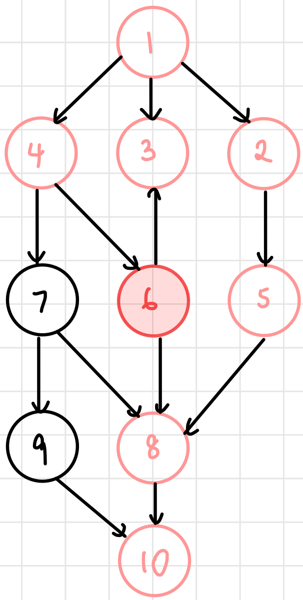
- 10 is a dead-end: backtrack to 8
- 8 has no unvisited children, backtrack to 5
- 5 has no unvisited children, backtrack to 2
- 2 has no unvisited children, backtrack to 1
- visit 3, mark as visited



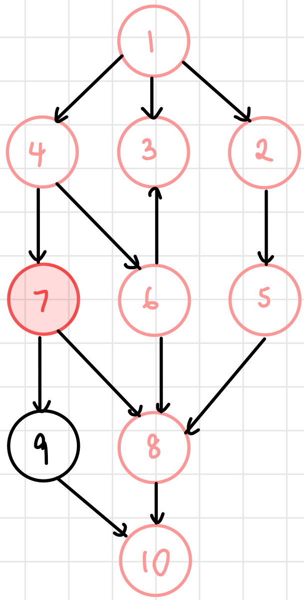
- 3 is a dead-end: backtrack to 1
- visit 4, mark as visited



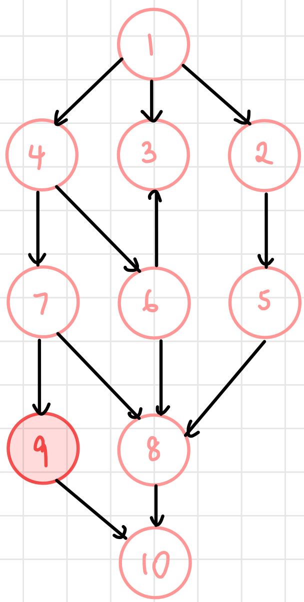
- visit 6, mark as visited



- 6 has no unvisited children, backtrack to 4
- visit 7, mark as visited



- visit 9, mark as visited

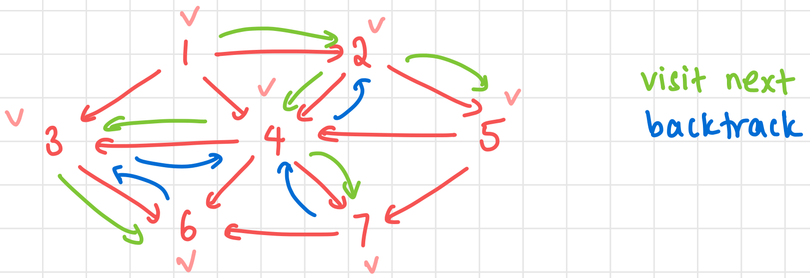


backtrack all the way up; after all have been visited

1, 2, 5, 8, 10, 3, 4, 6, 7, 9

Question 12

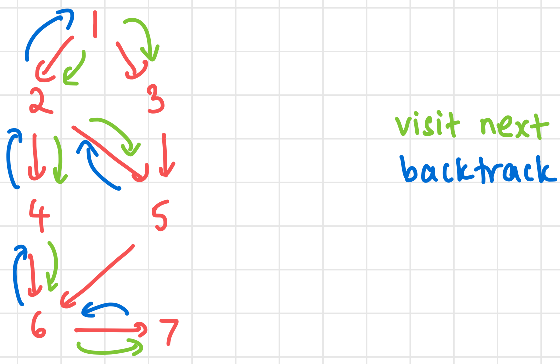
Write DFS traversal for the given graph (starting from vertex 1)



1, 2, 4, 3, 6, 7, 5

Question 13

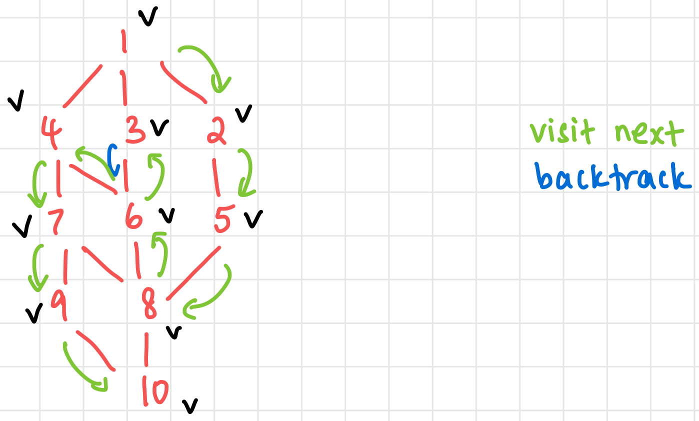
DFS from node 1



1, 2, 4, 6, 7, 5, 3

Question 14

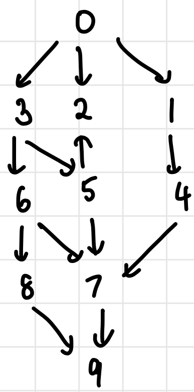
For undirected graph, show DFS



1, 2, 5, 8, 6, 3, 4, 7, 9, 10

Code Implementation - Adjacency Matrix

- Using recursion
- consider this graph



Initialises variables & accepts inputs before calling dfs_helper

```
void dfs(Graph *adj_mat) {
    int vertex, *visited;

    // Accept user input
    printf("Enter source vertex: ");
    scanf("%d", &vertex);

    // Out of bounds
    if (vertex < 0 || vertex >= adj_mat->n) {
        printf("Vertex not in graph.\n");
        return;
    }

    // Initialise visited list and set to 0s
    visited = (int *) calloc(adj_mat->n, sizeof(int));

    // Call recursive function
    dfs_helper(adj_mat, vertex, visited);

    // Free memory used by visited
    free(visited);
}
```

Actually performs DFS

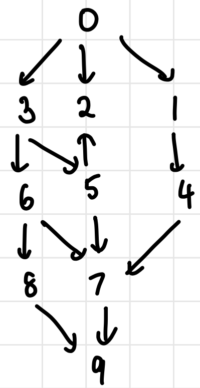
```
void dfs_helper(Graph *adj_mat, int vertex, int *visited) {
    // Mark node as visited and display
    visited[vertex] = 1;
    printf("%d ", vertex);

    // Call dfs_helper on all of its unvisited connections
    for (int i = 0; i < adj_mat->n; ++i) {
        if (adj_mat->adj[vertex][i] == 1 && visited[i] == 0) {
            dfs_helper(adj_mat, i, visited);
        }
    }
}
```

Example Output

```
Enter the number of vertices: 10
Enter source and destination vertices: 0 1
Enter source and destination vertices: 0 2
Enter source and destination vertices: 0 3
Enter source and destination vertices: 1 4
Enter source and destination vertices: 3 6
Enter source and destination vertices: 3 5
Enter source and destination vertices: 5 2
Enter source and destination vertices: 5 7
Enter source and destination vertices: 4 7
Enter source and destination vertices: 7 9
Enter source and destination vertices: 6 8
Enter source and destination vertices: 6 7
Enter source and destination vertices: 8 9
Enter source and destination vertices: -1 -1


MAIN MENU
1. Indegree of a vertex
2. Outdegree of a vertex
3. Display matrix
4. DFS traversal
5. Exit

4
Enter source vertex: 0
0 1 4 7 9 2 3 5 6 8
```

Code Implementation - Adjacency List

- Using recursion
- consider this graph



- new insert function: adds to the front of the list (more efficient)

- insert in opposite (descending) order to achieve same results

```
void insert(Node *adj_list[], int i, int j) {
    Node *new_node = (Node *) malloc(sizeof(Node));
    new_node->next = NULL;
    new_node->data = j;

    Node *temp = adj_list[i];

    adj_list[i] = new_node;
    new_node->next = temp;
}
```

```c
void dfs(Node *adj_list[], int n) {
    int vertex, *visited;

    // Accept user input
    printf("Enter source vertex: ");
    scanf("%d", &vertex);

    // Out of bounds
    if (vertex < 0 || vertex >= n) {
        printf("Vertex not in graph.\n");
        return;
    }

    // Initialise visited list and set to 0s
    visited = (int *) calloc(n, sizeof(int));

    // Call recursive function
    dfs_helper(adj_list, vertex, visited);
    printf("\n");

    // Free memory used by visited
    free(visited);
}
```

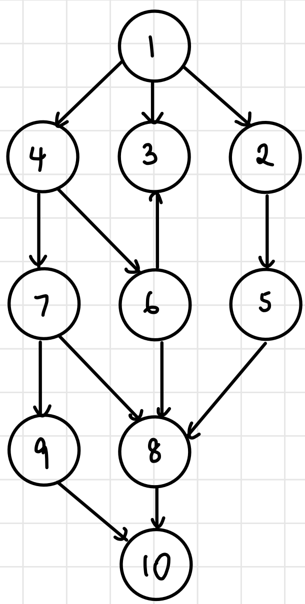
```c
void dfs_helper(Node *adj_list[], int vertex, int *visited) {
    // Mark node as visited and display
    visited[vertex] = 1;
    printf("%d ", vertex);

    Node *traverse = adj_list[vertex];

    while (traverse != NULL) {
        if (visited[traverse->data] == 0) {
            dfs_helper(adj_list, traverse->data, visited);
        }
        traverse = traverse->next;
    }
}
```

Example Output

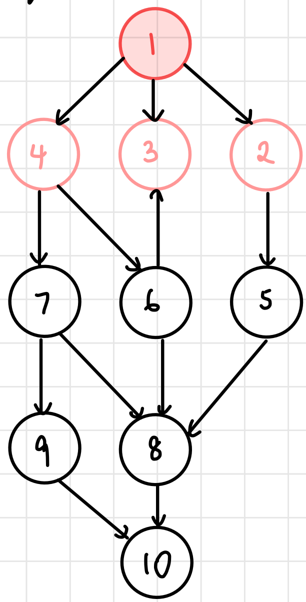
```
Enter the number of vertices: 10
Enter source and destination vertices: 0 3
Enter source and destination vertices: 0 2
Enter source and destination vertices: 0 1
Enter source and destination vertices: 1 4
Enter source and destination vertices: 3 6
Enter source and destination vertices: 3 5
Enter source and destination vertices: 5 7
Enter source and destination vertices: 5 2
Enter source and destination vertices: 4 7
Enter source and destination vertices: 6 8
Enter source and destination vertices: 6 7
Enter source and destination vertices: 8 9
Enter source and destination vertices: 7 9
Enter source and destination vertices: -1 -1


MAIN MENU
1. Indegree of a vertex
2. Outdegree of a vertex
3. Display matrix
4. DFS traversal
5. Exit
4
Enter source vertex: 0
0 1 4 7 9 2 3 5 6 8
```

Breadth First Search

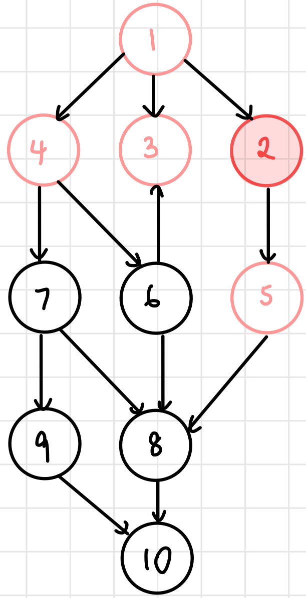
- uses queue data structure with insert, delete, is-empty functions
- Nodes that have been visited are marked as visited nodes (stored in visited array)
- Analogous to level-by-level traversal of a tree
- For both directed and undirected graphs
- Visit all vertices at the same depth at the same time
- Consider the following directed graph, starting at 1



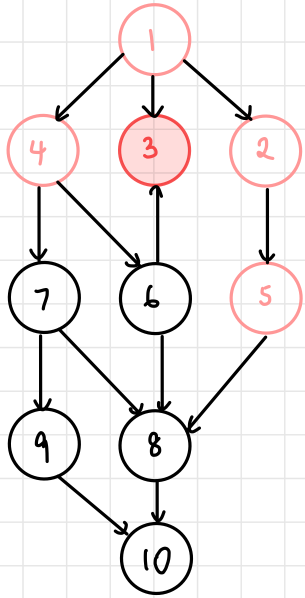
- Visit 1 (first node), append to queue Q = [1]
- Mark as visited, delete from queue
- Append 2, 3, 4 to queue & mark as visited Q = [2, 3, 4]



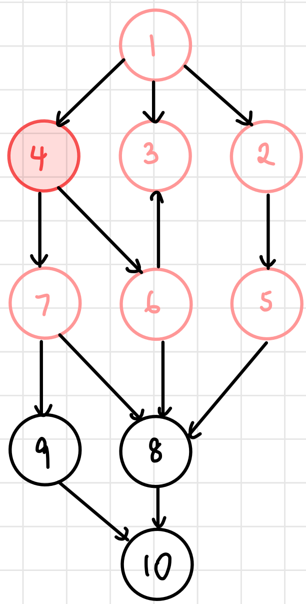
- Delete 2 from queue
- Append 5 to queue, mark as visited Q = [3, 4, 5]



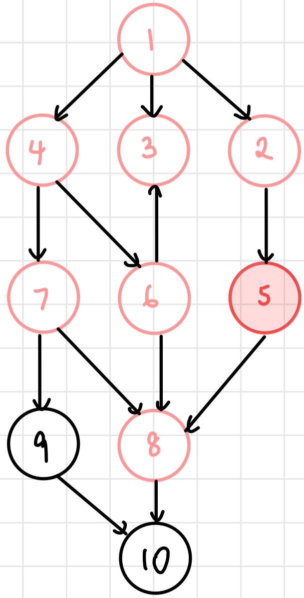
- Delete (dequeue) 3 from the queue
- Append nothing to queue   Q = [4,5]



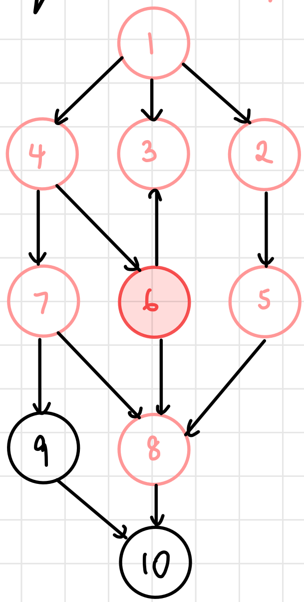
- Delete (dequeue) 4 from the queue
- Append 6,7 to queue, mark as visited Q = [5,6,7]



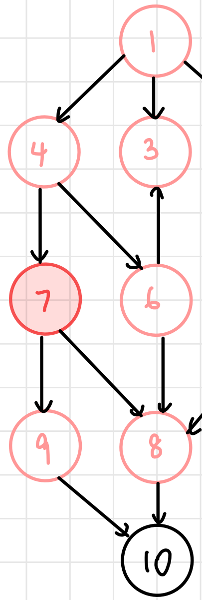
- Delete (dequeue) 5 from the queue
- Append 8 to queue, mark as visited Q = [6,7,8]



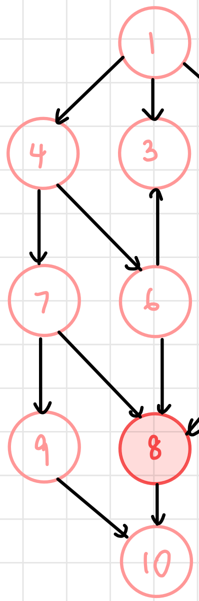
- Delete (dequeue) 6 from the queue
- Append nothing to queue Q = [7,8]



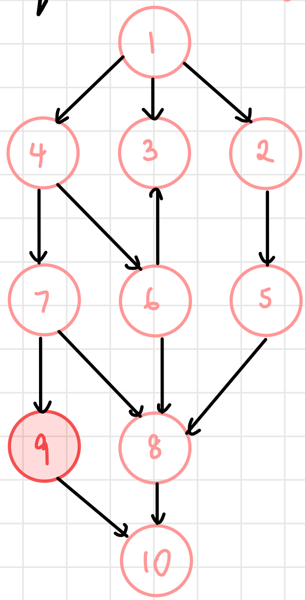
- Delete (dequeue) 7 from the queue
- Append 9 to queue, mark as visited Q = [8, 9]



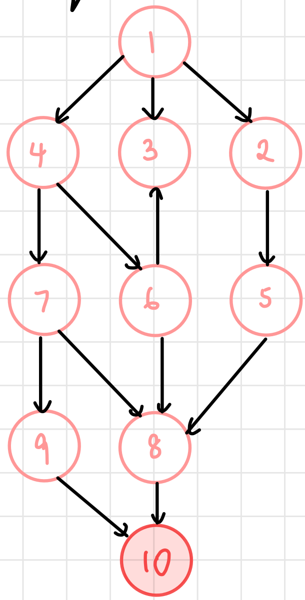
- Delete (dequeue) 8 from the queue
- Append 10 to queue, mark as visited Q = [9, 10]



- Delete (dequeue) 9 from the queue
- Append nothing to queue   Q = [10]



- Delete (dequeue) 10 from the queue
- Append nothing to queue   Q = [] ← empty queue: break



1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Code Implementation adjacency matrix

```
void bfs(Graph *adj_mat) {
    int vertex, *visited, *queue, qr = -1;

    // Accept user input
    printf("Enter source vertex: ");
    scanf("%d", &vertex);

    // Out of bounds
    if (vertex < 0 || vertex >= adj_mat->n) {
        printf("Vertex not in graph.\n");
        return;
    }

    // Initialise visited list and queue (init 0)
    visited = (int *) calloc(adj_mat->n, sizeof(int));
    queue = (int *) calloc(adj_mat->n, sizeof(int));

    // Loop
    append(queue, vertex, &qr);
    visited[vertex] = 1;

    // While queue is not empty
    while (qr != -1) {
        vertex = delete(queue, &qr);
        printf("%d ", vertex);
        for (int i = 0; i < adj_mat->n; ++i) {
            if (adj_mat->adj[vertex][i] == 1 && visited[i] == 0) {
                visited[i] = 1;
                append(queue, i, &qr);
            }
        }
    }
    printf("\n");
    // Free memory used by visited and queue
    free(visited);
    free(queue);
}
```

```c
void append(int *queue, int v, int *pqr) {
    ++(*pqr);
    queue[*pqr] = v;
}

int delete(int *queue, int *pqr) {
    int res = queue[0];

    for (int i = 0; i < *pqr; ++i) {
        queue[i] = queue[i + 1];
    }
    --(*pqr);
    return res;
}
```

adjacency list

```c
void bfs(Node *adj_list[], int n) {
    int vertex, *visited, *queue, qr = -1;

    // Accept user input
    printf("Enter source vertex: ");
    scanf("%d", &vertex);

    // Out of bounds
    if (vertex < 0 || vertex >= n) {
        printf("Vertex not in graph.\n");
        return;
    }

    // Initialise visited list and queue (init 0)
    visited = (int *) calloc(n, sizeof(int));
    queue = (int *) calloc(n, sizeof(int));

    // Loop
    append(queue, vertex, &qr);
    visited[vertex] = 1;
```

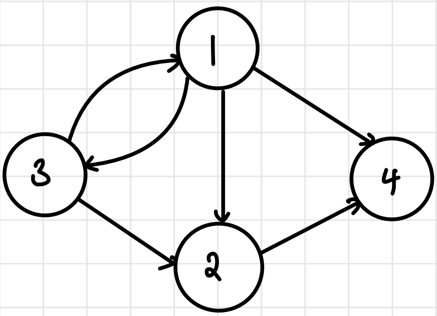
```
    // While queue is not empty
    while (qr != -1) {
        vertex = delete(queue, &qr);
        printf("%d ", vertex);

        Node *traverse = adj_list[vertex];

        while (traverse) {
            if (visited[traverse->data] == 0) {
                visited[traverse->data] = 1;
                append(queue, traverse->data, &qr);
            }
            traverse = traverse->next;
        }
    }

    printf("\n");

    // Free memory used by visited and queue
    free(visited);
    free(queue);
}
```

Output for the same graph

```
MAIN MENU
1. Indegree of a vertex
2. Outdegree of a vertex
3. Display matrix
4. BFS traversal
5. Exit
4
Enter source vertex: 0
0 1 2 3 4 5 6 7 8 9
```

Finding a Path in a Graph

- Find all the paths from a source to a destination
- Example: find all paths from 3 to 4



paths

3 → 1 → 4

3 → 1 → 2 → 4

3 → 2 → 4

USING DFS

- start from source and traverse, storing all vertices in an array
- when destination reached, print
- using adjacency list

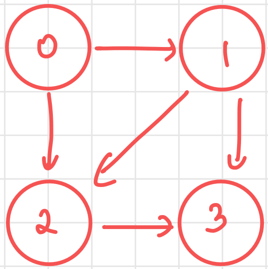
1 → 2 → 3 → 4 → NULL

2 → 4 → NULL

3 → 1 → 2 → NULL

4 → NULL

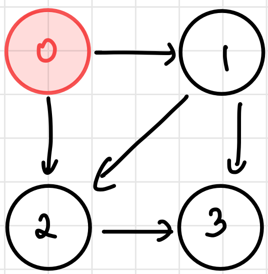
Question 15



find paths from 0 to 3 using DFS

Steps

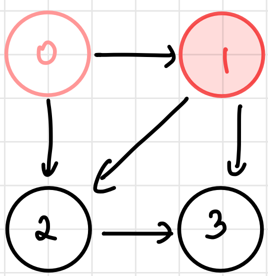
- visit source (0)



visited = [0]

path = [0]

- 0 is not destination
- visit adjacent node (1)

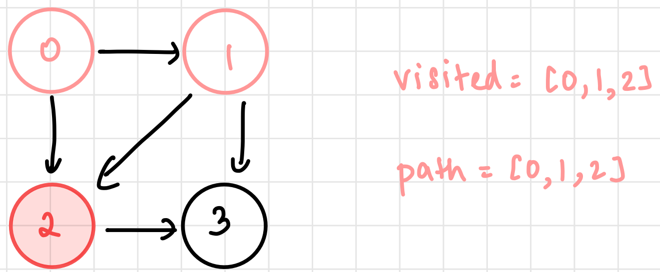


visited = [0,1]

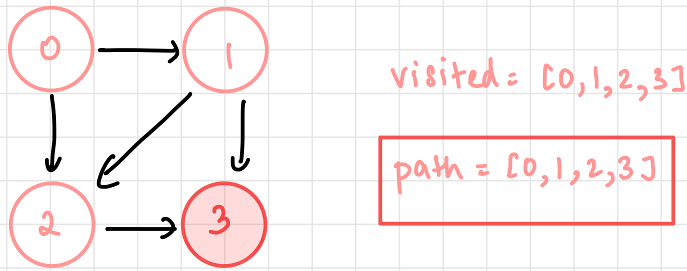
path = [0,1]

- 1 is not destination

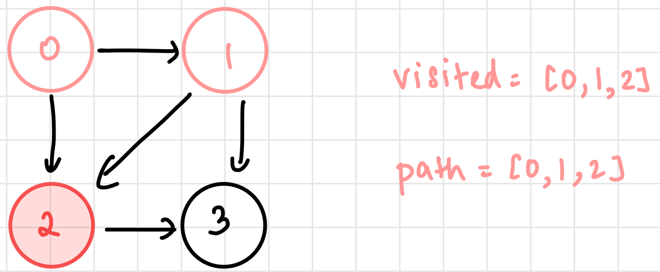
- visit adjacent node (2)



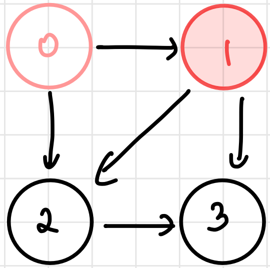
- 2 is not destination
- visit adjacent node (3)



- 3 is destination
- Print path array and mark 3 as unvisited
- Remove 3 from path and backtrack



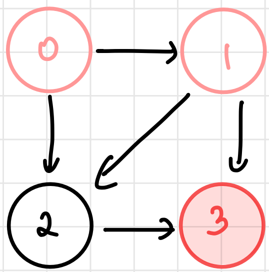
- no unvisited adjacent node of 2
- mark 2 as unvisited, remove from path and backtrack



visited = [0, 1]

path = [0, 1]

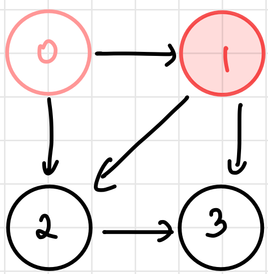
- go to unvisited adjacent node (3)
- mark as visited



visited = [0, 1, 3]

path = [0, 1, 3]

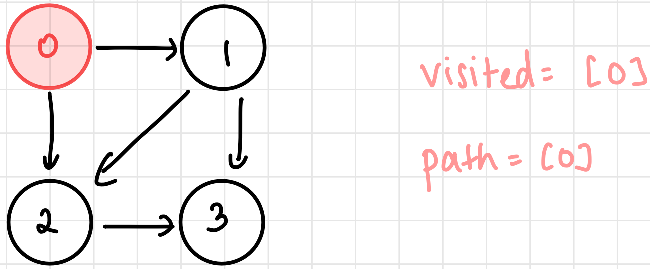
- 3 is destination
- Print path array, mark 3 as unvisited
- Remove 3 from path and backtrack



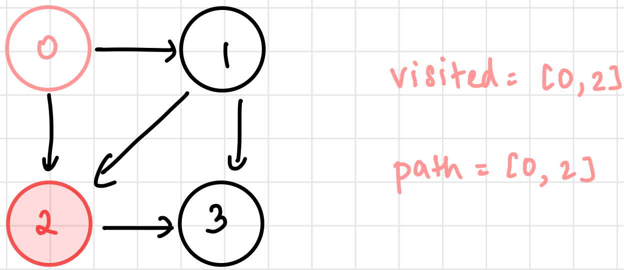
visited = [0, 1]

path = [0, 1]

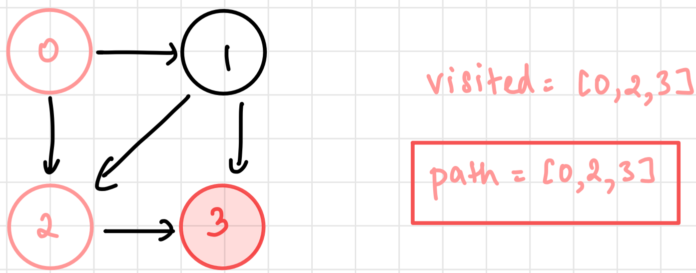
- no unvisited adjacent node of 1
- mark 1 as unvisited, remove from path and backtrack



- visit adjacent neighbour (2)
- mark as visited



- 2 is not destination
- go to unvisited adjacent node (3)
- mark as visited



Code Implementation

- using adjacency list

```
void dfs_path(Node *adj_list[], int n, int source, int dest) {
    int *visited, *path, count = 0;

    // Out of bounds
    if (source < 0 || source >= n) {
        printf("Source not in graph.\n");
        return;
    }
    if (dest < 0 || dest >= n) {
        printf("Destination not in graph.\n");
        return;
    }

    // Initialise visited list and path list to 0s
    visited = (int *) calloc(n, sizeof(int));
    path = (int *) calloc(n, sizeof(int));

    // Call recursive function
    print_path(adj_list, source, dest, visited, path, count);
    printf("\n");

    // Free memory used by visited and path
    free(visited);
    free(path);
}
```

```c
// Recursive function
void print_path(Node *adj_list[], int source, int dest, int *visited, int
*path, int count) {
    // Mark node as visited and display
    visited[source] = 1;
    path[count] = source;
    ++count;
    // Print array if destination reached
    if (source == dest) {
        for (int i = 0; i < count; ++i) {
            printf("%d ", path[i]);
        }
        printf("\n");
    }
    else {
        for (Node *t = adj_list[source]; t != NULL; t = t->next) {
            if (!visited[t->data]) {
                print_path(adj_list, t->data, dest, visited, path, count);
            }
        }
    }
    // Backtrack
    --count;
    visited[source] = 0;
}
```

Output

```
Enter the number of vertices: 4
Enter source and destination vertices: 0 1
Enter source and destination vertices: 0 2
Enter source and destination vertices: 1 2
Enter source and destination vertices: 1 3
Enter source and destination vertices: 2 3
Enter source and destination vertices: -1 -1
Enter source vertex: 0
Enter destination vertex: 3
0 1 2 3
0 1 3
0 2 3
```

USING BFS

- Simply check if there is a path connecting the source and destination vertices
- To store path, each node should store the previous node that was visited and then once a destination is reached, the path can be traced.

Code Implementation

```
int bfs_path(Graph *adj_mat, int source, int dest) {
    int *visited, *queue, qr = -1, vertex;

    // Out of bounds
    if (source < 0 || source >= adj_mat->n) {
        printf("Source not in graph.\n");
        return 0;
    }
    if (dest < 0 || dest >= adj_mat->n) {
        printf("Destination not in graph.\n");
        return 0;
    }

    // Initialise visited list and queue (init 0)
    visited = (int *) calloc(adj_mat->n, sizeof(int));
    queue = (int *) calloc(adj_mat->n, sizeof(int));

    // Loop
    append(queue, source, &qr);
    visited[source] = 1;
```

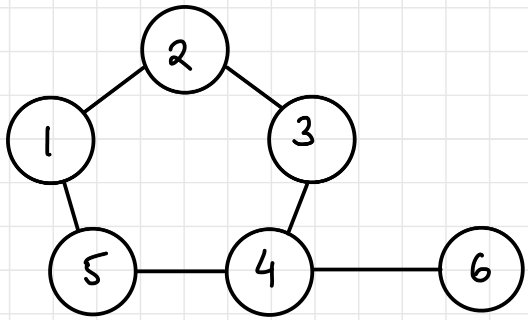
```c
    // While queue is not empty
    while (qr != -1) {
        vertex = delete(queue, &qr);

        // Destination reached
        if (vertex == dest) {
            return 1;
        }

        for (int i = 0; i < adj_mat->n; ++i) {
            if (adj_mat->adj[vertex][i] && !visited[i]) {
                visited[i] = 1;
                append(queue, i, &qr);
            }
        }
    }
    // Free memory used by visited and queue
    free(visited);
    free(queue);
    return 0;
}
```

Helper functions for queues

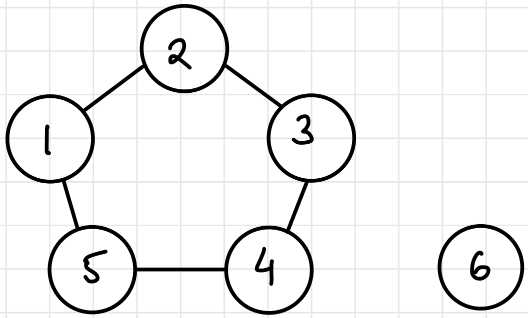
```c
void append(int *queue, int v, int *pqr) {
    ++(*pqr);
    queue[*pqr] = v;
}

int delete(int *queue, int *pqr) {
    int res = queue[0];

    for (int i = 0; i < *pqr; ++i) {
        queue[i] = queue[i + 1];
    }
    --(*pqr);
    return res;
}
```

Connected Graphs

- If all other nodes can be visited from one node, the graph is connected (strongly connected)



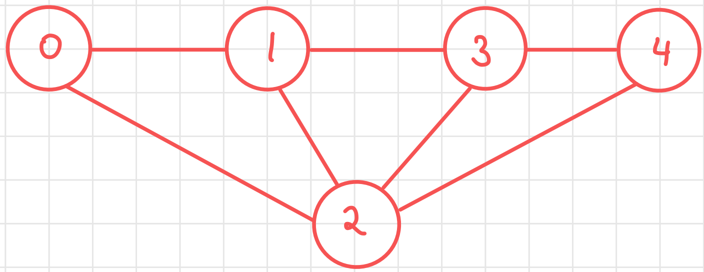
- Otherwise, disconnected



- Can be checked for using BFS or DFS
- weakly connected: all nodes visitable from any one node

Question 16

Check for connectivity



Adjacency matrix

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

Using BFS (start with 0)

f r

queue: 0

visited:

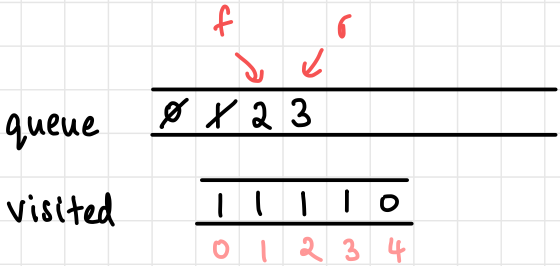
0	1	2	3	4
1	0	0	0	0

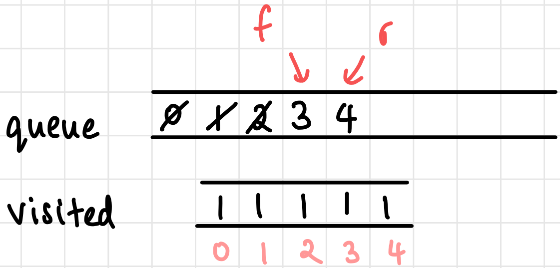
f r

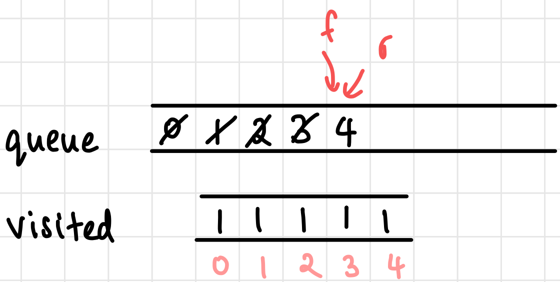
queue: ~~0~~ 1 2

visited:

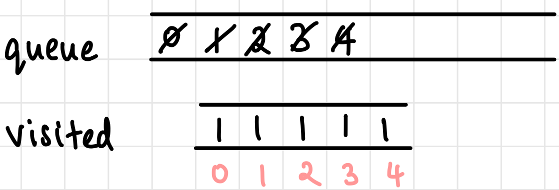
0	1	2	3	4
1	1	1	0	0







empty – checked for 0



- repeat with all nodes

Code Implementation

- create graph function for directed & undirected graphs
- Using BFS

```
void create_graph(Graph *adj_mat, char undir) {
    int i, j;

    // Is graph undirected?
    int un = (undir == 'y' || undir == 'Y');

    for (int i = 0; i < adj_mat->n; ++i) {
        for (int j = 0; j < adj_mat->n; ++j) {
            adj_mat->adj[i][j] = 0;
        }
    }

    while (1) {
        printf("Enter source and destination vertices: ");
        scanf("%d %d", &i, &j);

        if (i < 0 && j <= 0 || i >= adj_mat->n || j >= adj_mat->n) {
            break;
        }

        adj_mat->adj[i][j] = 1;
        if (un) {
            adj_mat->adj[j][i] = 1;
        }
    }
}
```

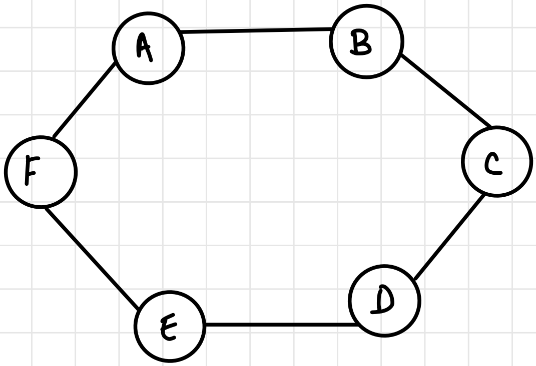
- bfs_con for strongly connected graphs
- can also do with DFS

```c
int bfs_con(Graph *adj_mat) {
    int *visited, *queue, qr = -1;
    visited = (int *) calloc(adj_mat->n, sizeof(int));   ] initialise to 0
    queue = (int *) calloc(adj_mat->n, sizeof(int));

    for (int start = 0; start < adj_mat->n; ++start) {
        // Initialise visited array
        for (int i = 0; i < adj_mat->n; ++i) {
            visited[i] = 0;
        }
        append(queue, start, &qr);     ] append start vertex to queue
        visited[start] = 1;              and mark as visited
        int vertex;
        // While queue is not empty
        while (qr != -1) {
            // Dequeue first element
            vertex = delete(queue, &qr);

            // Enqueue all unvisited connections
            for (int i = 0; i < adj_mat->n; ++i) {
                if (adj_mat->adj[vertex][i] && !visited[i]) {
                    visited[i] = 1;
                    append(queue, i, &qr);
                }
            }
        }
        // Check visited array
        for (int i = 0; i < adj_mat->n; ++i) {
            if (!visited[i]) {
                free(visited);
                free(queue);
                return 0;
            }
        }
    }
    // Free memory used by visited and queue
    free(visited);
    free(queue);
    return 1;
}
```

Computer Network Topology

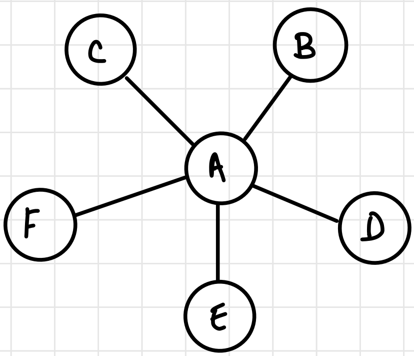
1. Ring Topology (cycle)

- all vertices have degree =2
- no of edges = no of vertices



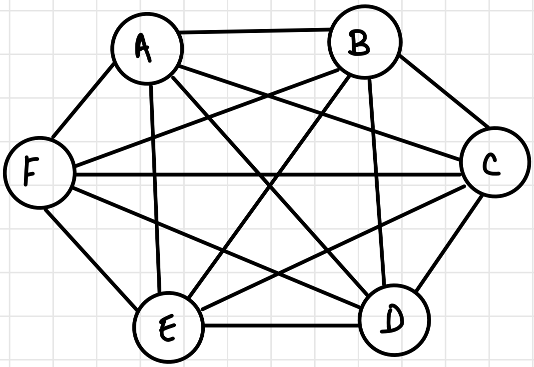
2. Star Topology

- no. of links = no. of nodes -1
- one central vertex connected to all others



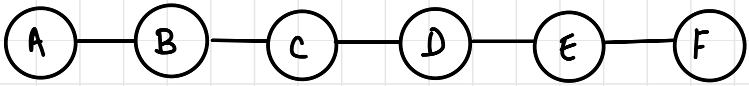
3. Mesh topology

- complete graph



4. Bus Topology

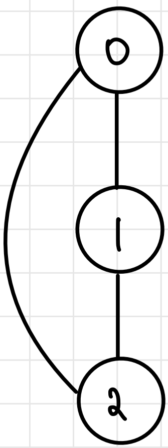
- every node has degree = 2 except ending nodes which have degree = 2



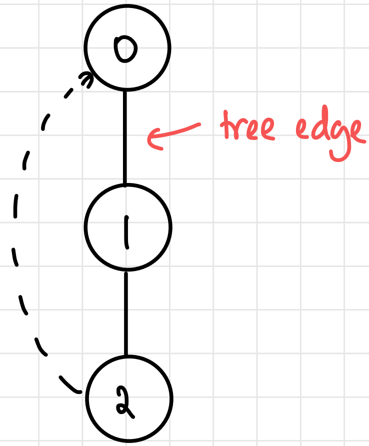
- most networks are combination of all

Presence of cycle in Graph

Graph



tree generated while traversing



0 is adjacent to 2 & has already been visited (and is not a parent)

- If a non-parent adjacent node to a node has already been visited, there is a cycle in the graph
- More than one way to get to a node

Code Implementation

called by main

```c
int dfs_cycle(Graph *adj_mat) {
    /* For a connected graph */
    int *visited;

    /* Initialise visited list and queue (init 0) */
    visited = (int *) calloc(adj_mat->n, sizeof(int));

    int res = dfs(adj_mat, 0, visited, -1);

    /* Free memory used by visited and queue */
    free(visited);
    return res;
}
```

parent

vertex

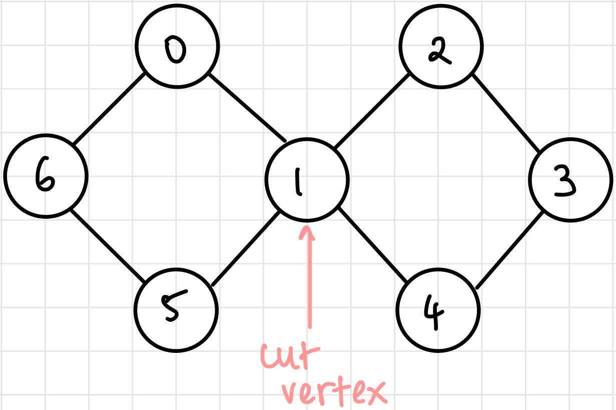
```c
int dfs(Graph *adj_mat, int vertex, int *visited, int parent) {
    int res;
    visited[vertex] = 1;
    for (int i = 0; i < adj_mat->n; ++i) {
        /* If the connection exists and is not the parent */
        if (adj_mat->adj[vertex][i] && i != parent) {
            /* If the child is visited */
            if (visited[i]) {
                return 1;
            }
            /* If child is not visited */
            else {
                res = dfs(adj_mat, i, visited, vertex);
                if (res) return res;
            }
        }
    }
    return 0;
}
```

mark visited

new parent

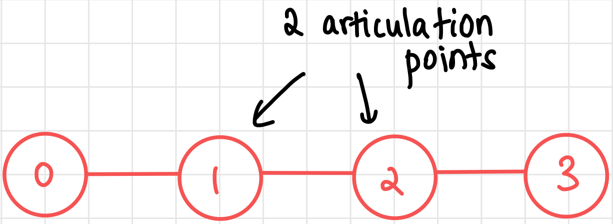
Articulation Point

- Also called cut vertex
- Vertex that when removed makes a graph disconnected (can be multiple)
- Important to identify in computer networks as failure of this point can result in splitting of network



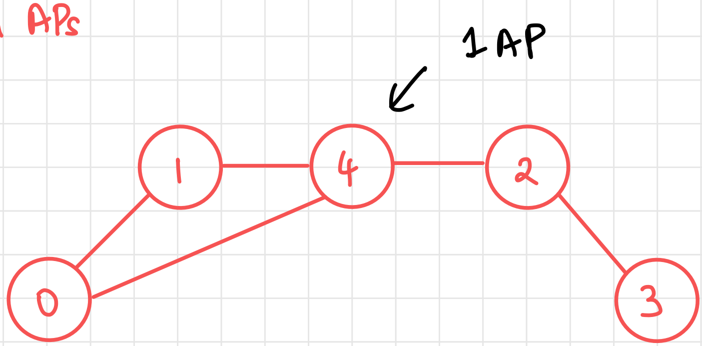
Question 17

Find articulation points



Question 18

Find APs

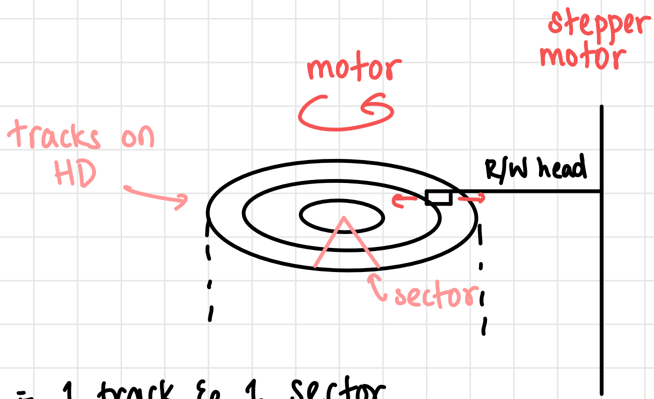


Algorithm

1) Remove one vertex and check for connectivity
2) Repeat for all vertices
3) Can find all articulation points
4) Can perform either DFS or BFS
5) If graph disconnected when any one vertex has been removed, that vertex is an articulation point

INDEXING USING B-TREES

- Large data records to be stored on secondary memory when RAM is not large enough



- Block = 1 track & 1 sector

  eg: t3, s1

- Each track has several blocks, each of same size (based on number of sectors)

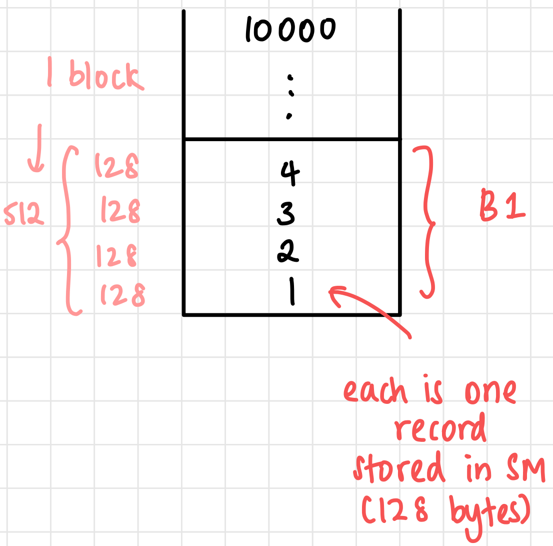
- Block 1 : 512 bytes → example

- Disc access: make required block come under the read/write head (moves linearly)

- Entire block (eg: 512 bytes) transferred to RAM, even if only single character required

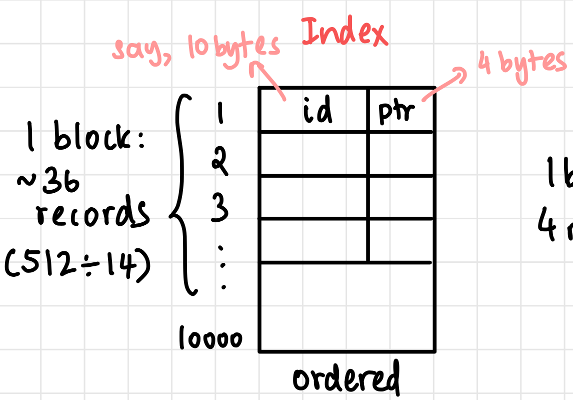
- HD is block-controlled device, while keyboard is character-controlled device

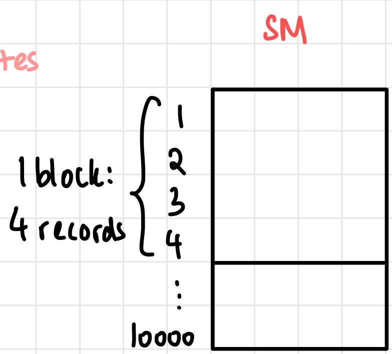
- Transferred through buffer



- can be sorted (ordered) or unordered
- For sorted data, binary search
- Need to access all blocks for linear search (unordered) for worst case scenario
- Want to minimise number of block accesses (search time)
- need 2500 blocks

- Create an index page





10000 ÷ 36 ≈ 280 blocks

10000 ÷ 4 = 2500 blocks

worst case: 280

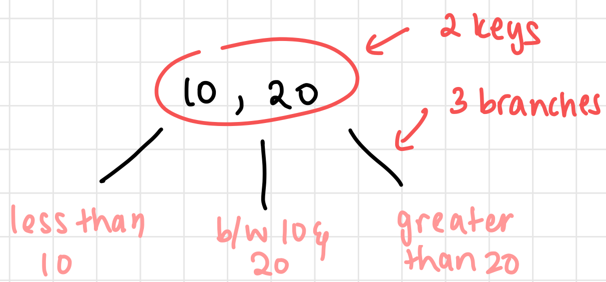
worst case: 2500

- ignoring access within block of 4

- Index page for an index page (reduce even further)
- Range of ids stored in 2nd index table / page

B-Tree

- Multiway search tree ; based on BST
- Good for creating indices
- Can have as many keys & branches in a multiway search tree



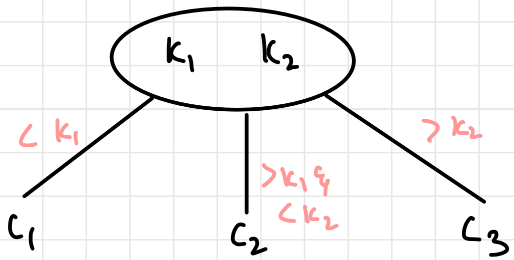
- For strict non-linearity and O(log(n)) time, multiway search tree must be balanced
- Balanced multiway search tree : B-tree (Bayer tree / balanced tree / Fat tree)
- Restrictions for preventing skewedness in multiway search tree
- Order m: at most m children

B-Tree of Order m

1) All leaves on same level

2) All internal nodes except the root have at most (m) non-empty children and at least $\lceil m/2 \rceil$ non-empty children, at most (m-1) keys and at least $\lceil m/2 - 1 \rceil$ keys (non-root)

3) Number of keys in each internal node is one less than the number of non-empty children and partitioning is based on search tree concept



4) Root max : m children and min : 2 children / 0 children

Construction of a B-Tree / Insertion

Question 19

B-Tree of order 5

50, 60, 20, 10, 30, 40, 5, 8, 80, 100

$$\text{min keys} = \left\lceil \frac{5}{2} - 1 \right\rceil = 2$$

$$\text{max keys} = 5 - 1 = 4$$

$$\text{min nodes} = \left\lceil \frac{5}{2} \right\rceil = 3$$

$$\text{max nodes} = 5$$

- note: keys will stop getting added only after they have reached max

Step 1

Read 50

(50) ← key

Step 2

Read 60

(50, 60) ← keys

Step 3

Read 20

(20, 50, 60) ← in order

Step 4

Read 10

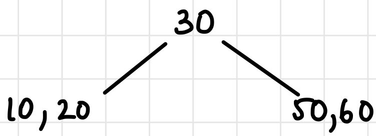
(10, 20, 50, 60) ← in order

Step 5

Read 30

(10, 20, 30, 50, 60) ← overflow

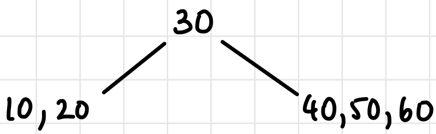
- Take middle & split (for even, can choose bias)



Step 6

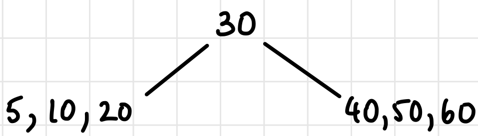
Read 40

- Added to leaves (bottom-up)



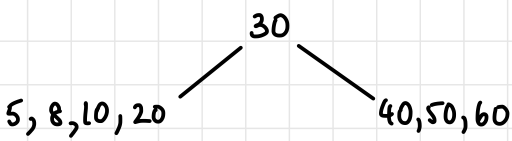
Step 7

Read 5



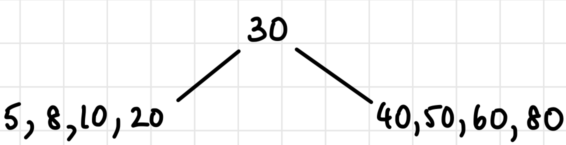
Step 8

Read 8



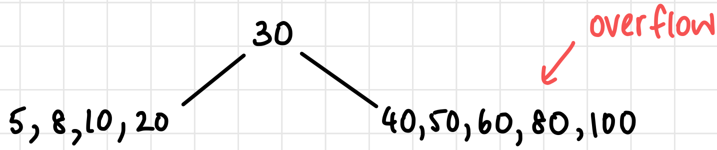
Step 9

Read 80

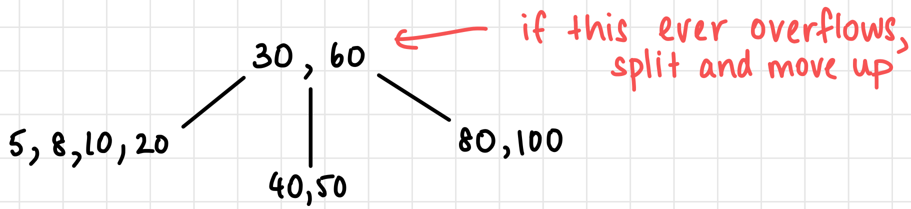


Step 10

Read 100



- Take middle & split (add to topmost node)



- Each level is a level of indexing
- Also called fat tree (short & wide)

Deletion in B-Trees

- Deletion can be internal node or leaf node

1) Non-leaf / internal node

  - its immediate predecessor / successor will be in a leaf
  - promote immediate predecessor / successor to position of deleted node

2) Leaf node

(i) Case 1 - leaf contains keys > min no. of keys

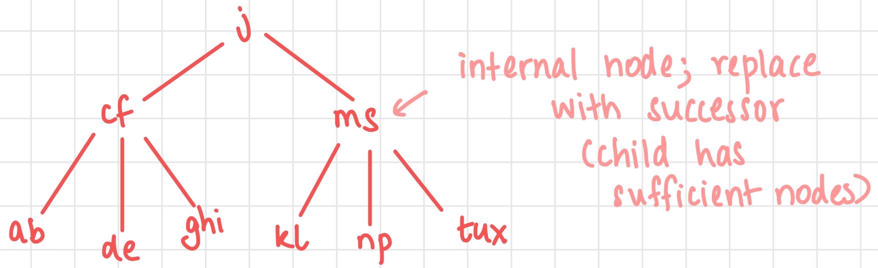
- simply delete the key

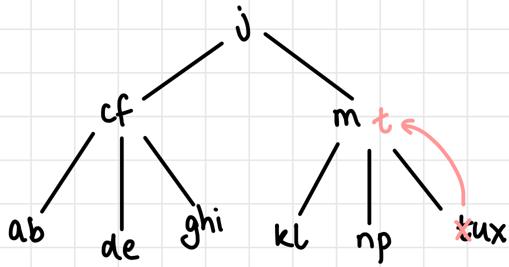
(ii) Case 2 - leaf contains min no. of keys

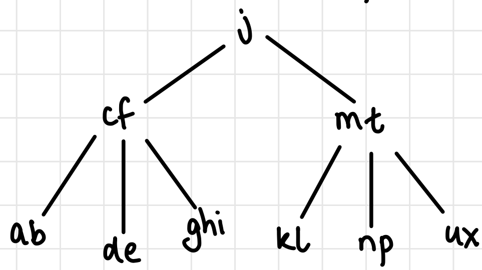
- first look at two adjacent leaves (immediate) and are children of same parent
- if one of them has more than min, move key to parent and move parent to deletion position
- if adjacent leaf has only minimum number of entries, then two leaves and the median entry from parent are combined as new leaf which will contain no more than the maximum no. of entries
- If this step leaves the parent node with few entries, the process propagates upwards

Question 20

Delete node s

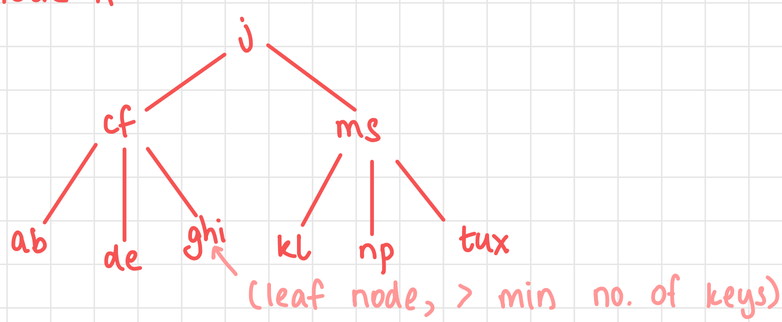


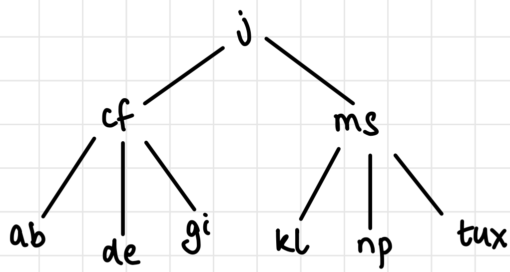




Question 21

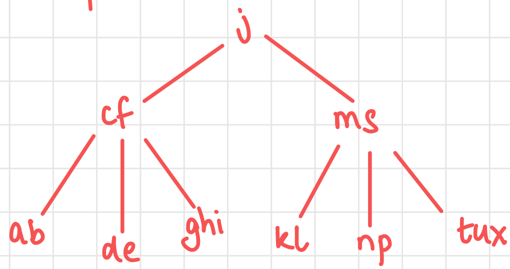
Delete node h

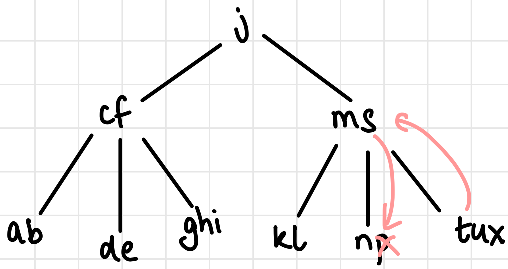


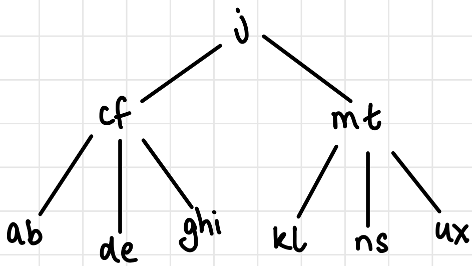


Question 22

Delete node p

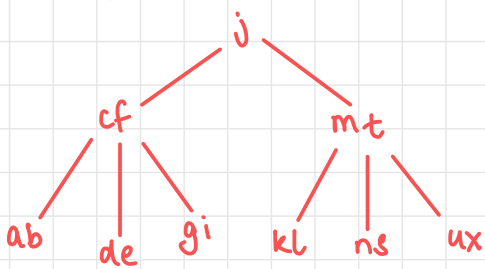


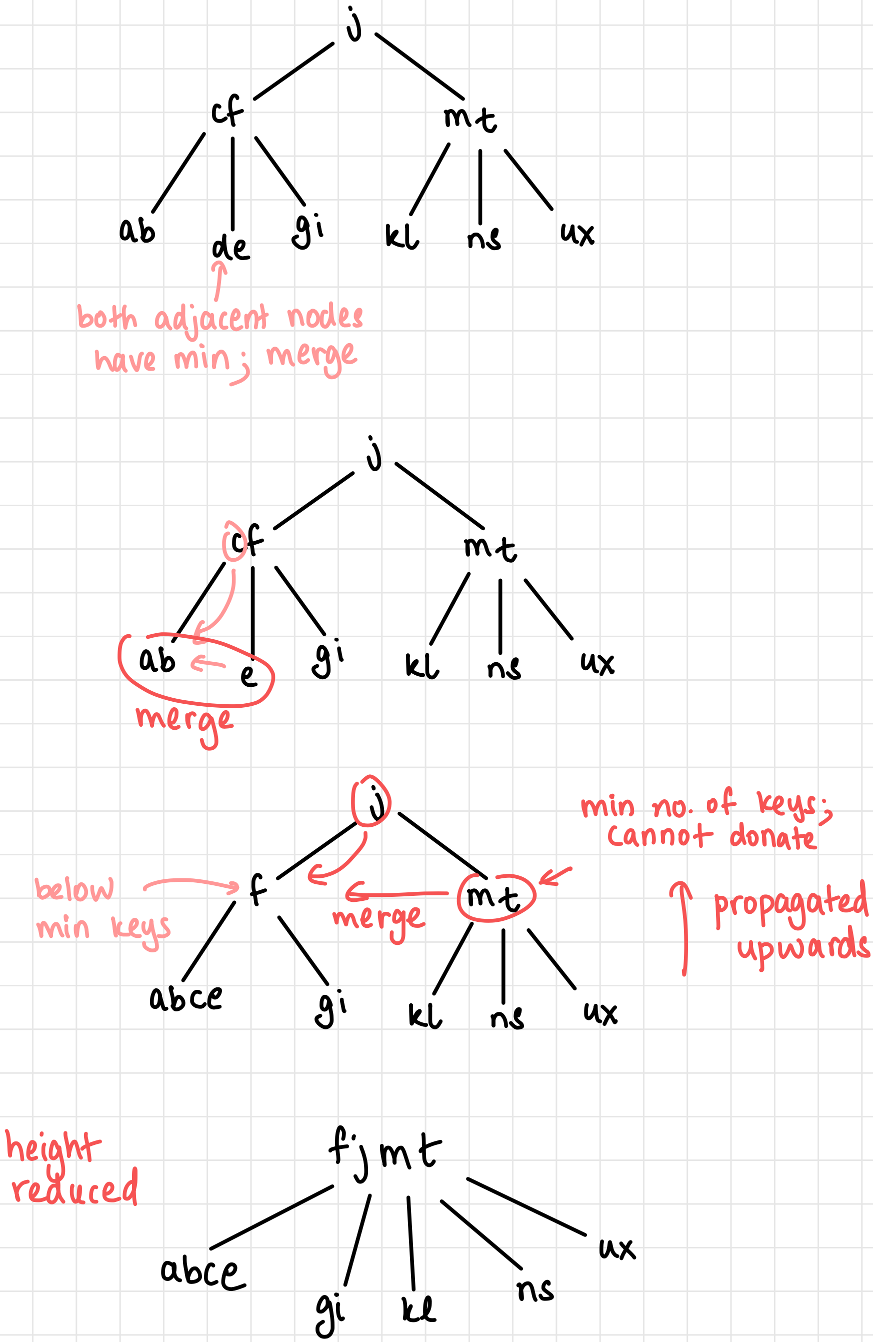




Question 23

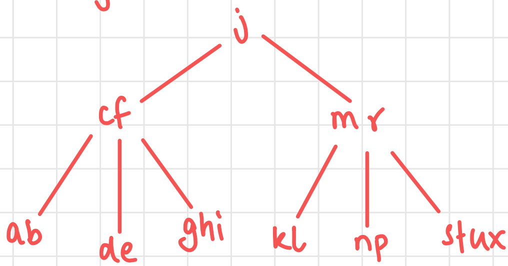
Delete node d

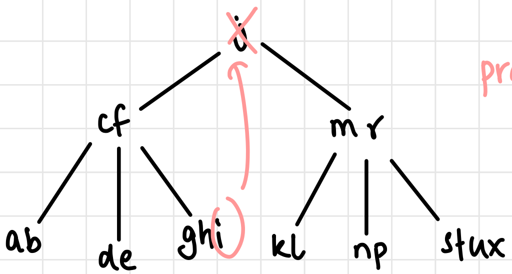



j
cf
mt
ab
de
gi
kl
ns
ux
both adjacent nodes have min; merge
j
cf
mt
ab
e
gi
kl
ns
ux
merge
j
f
mt
abce
gi
kl
ns
ux
below min keys
merge
min no. of keys; cannot donate
propagated upwards
fjmt
abce
gi
kl
ns
ux
height reduced

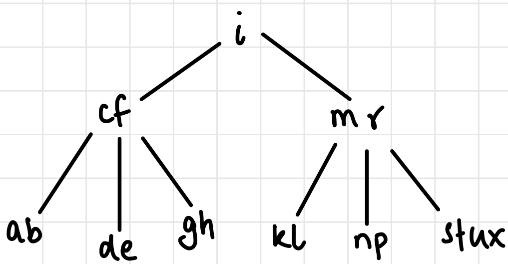
Question 24

Delete node j



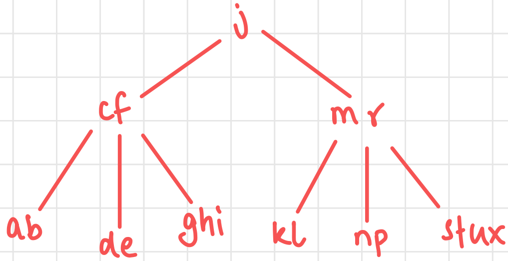


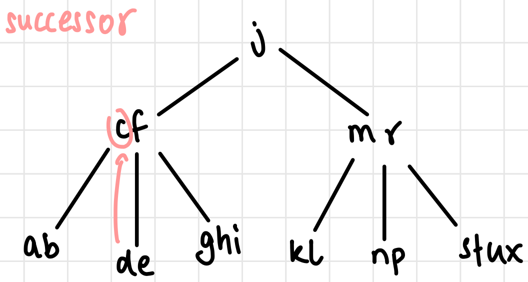
predecessor

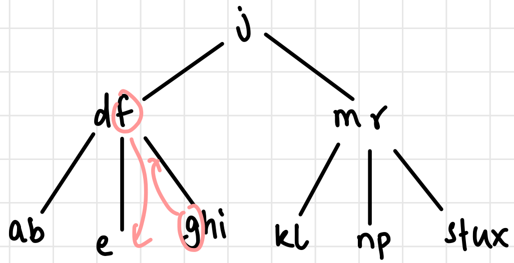


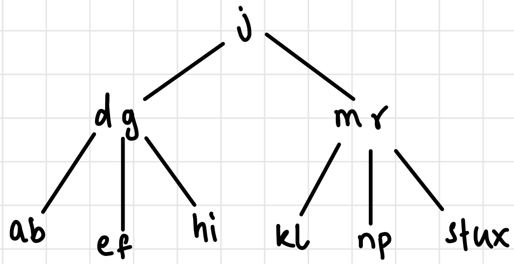
Question 25

Delete node c









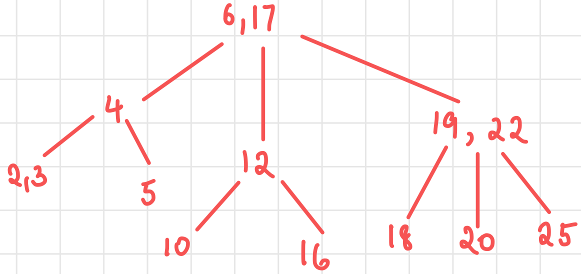
Question 26

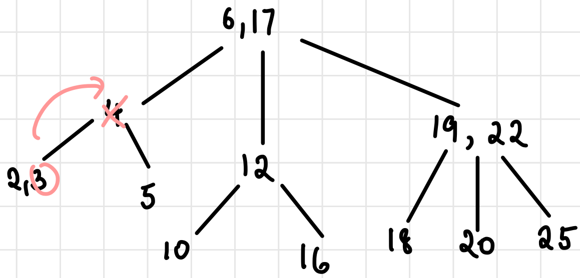
Delete node 4

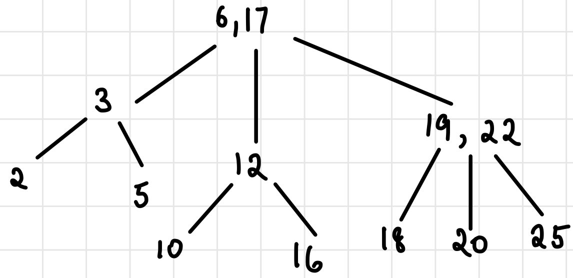
(m=3)

min keys = $\lceil \frac{3}{2} - 1 \rceil = 1$

max keys = $\lceil 3-1 \rceil = 2$

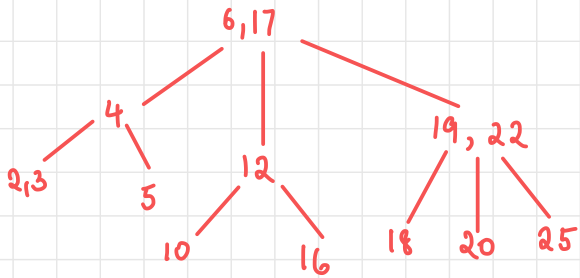


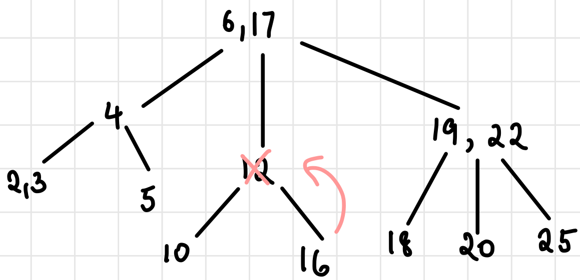


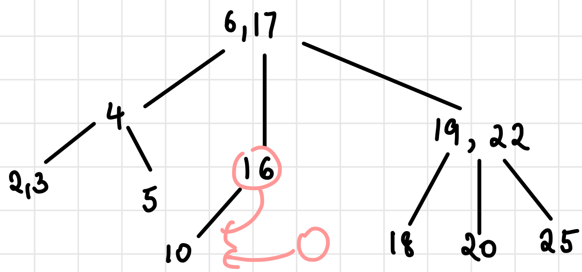


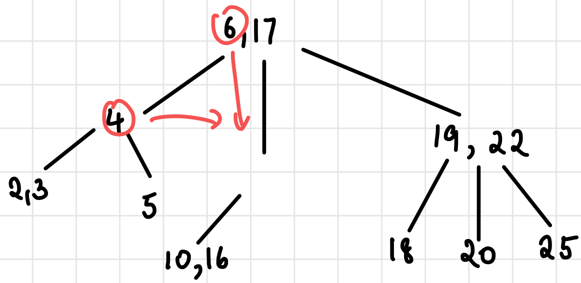
Question 27

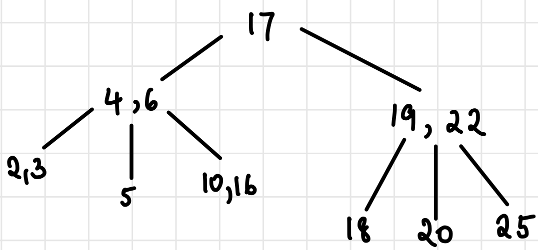
Delete node 12






6,17
4
16
19, 22
2,3
5
10
18
20
25


6,17
4
19, 22
2,3
5
10,16
18
20
25


17
4,6
19, 22
2,3
5
10,16
18
20
25