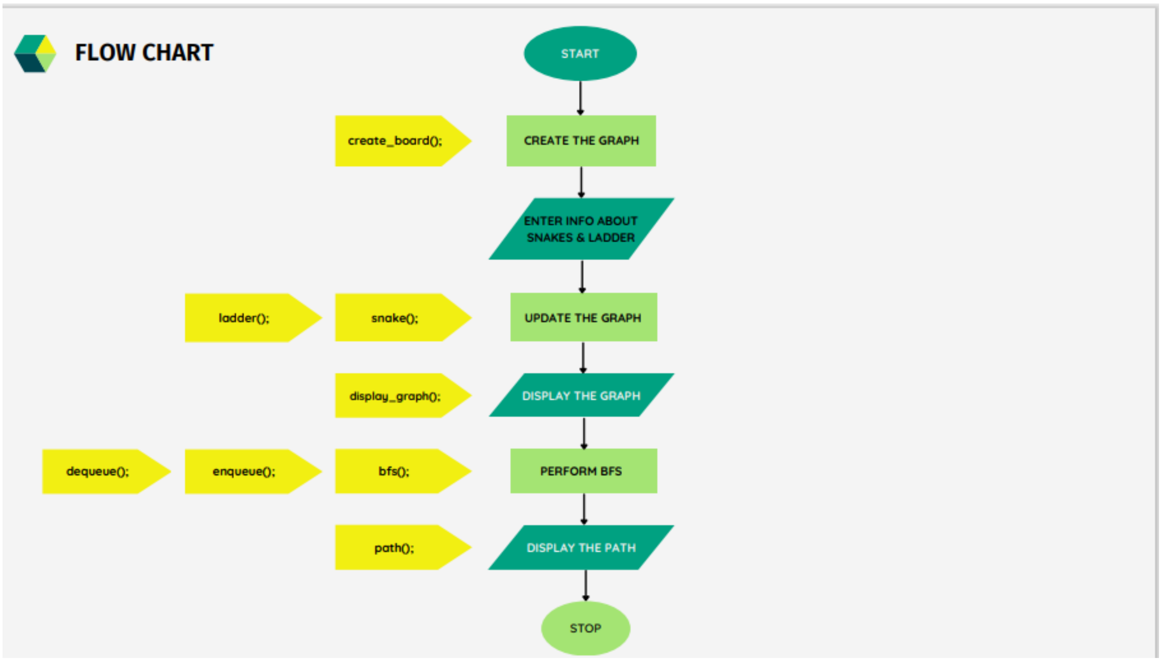
INDEX

SNAKE AND LADDER PROBLEM

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. This is done by considering that the player can determine which number appears in the dice being biased. The player rolls the dice and if reaches a base of a ladder then he can move up the ladder to a different position/cell and if he reaches a snake then it brings him down away from the destination cell/position.

This problem can be solved using a Breadth-First Search (BFS)
Data Structure used: Graph.

FLOWCHART



DATA STRUCTURES USED

1. Queue
2. Linked List
3. Graph

1. QUEUE

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. Some common applications of Queue data structure:

Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

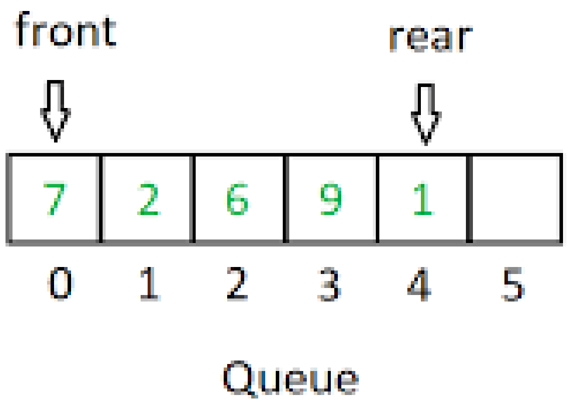
Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

Message Buffering: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

Event Handling: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

Traffic Management: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.



2. LINKED LIST

Array is a linear data structure that is a collection of similar data types. Arrays are stored in contiguous memory locations. It is a static data structure with a fixed size. It combines data of similar types. Below are some real-time applications of arrays:

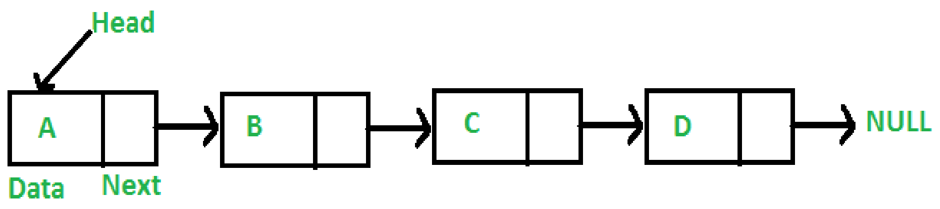
Signal Processing: Arrays are used in signal processing to represent a set of samples that are collected over time.

Data Mining: Arrays are used in data mining applications to represent large datasets. This allows for efficient data access and processing, which is important in real-time applications.

Robotics: Arrays are used in robotics to represent the position and orientation of objects in 3D space. This can be used in applications such as motion planning and object recognition.

Real-time Monitoring and Control Systems: Arrays are used in real-time monitoring and control systems to store sensor data and control signals.

Scientific Computing: Arrays are used in scientific computing to represent numerical data, such as measurements from experiments and simulations.



3. GRAPHS

A Graph is a non-linear data structure consisting of vertices and edges. Vertices are also referred to as nodes. Edges are lines or arcs that connect any two nodes in the graphs. Graph is composed of a set of vertices(V) and a set of edges(E). Denoted by G(V, E). In Computer science graphs are used to represent the flow of computation.
Google maps uses graphs for building transportation systems, where intersection of two (or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms:

Vertex: Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

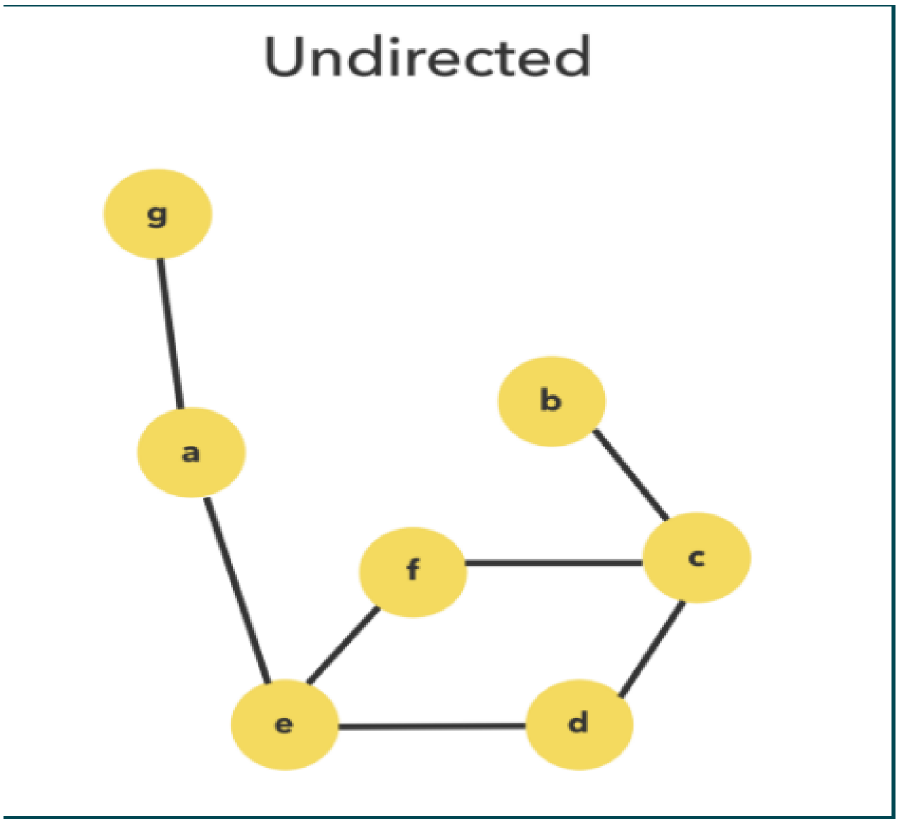
Edge: Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency: Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path: Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

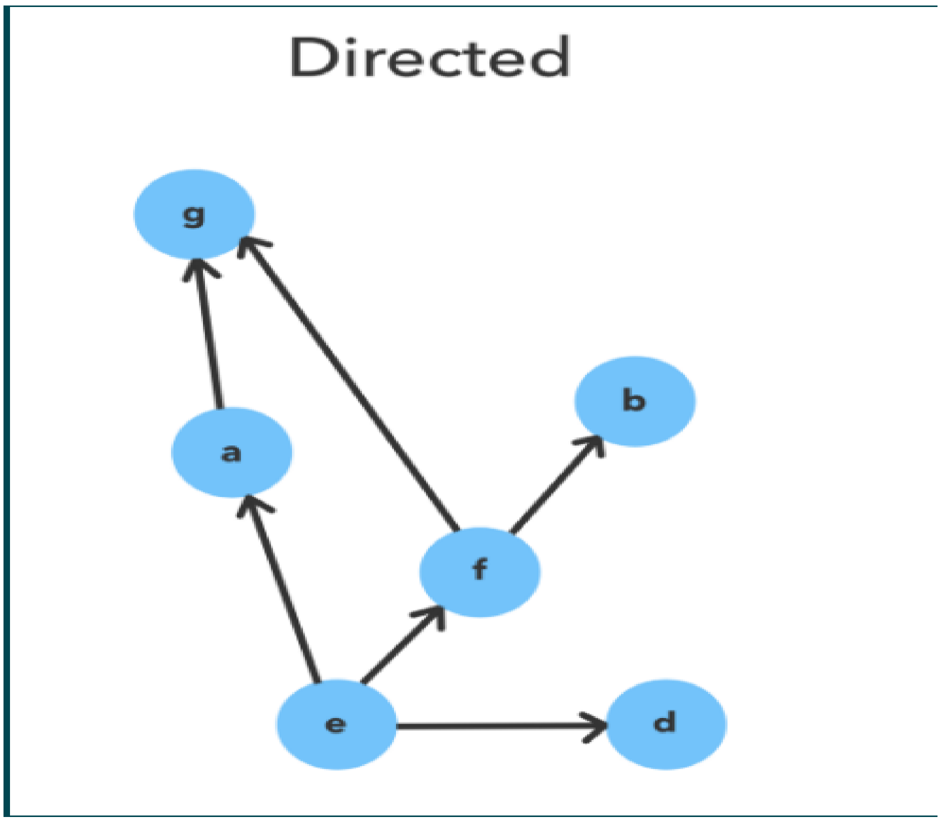
TYPES OF GRAPHS

- UNDIRECTED GRAPH



Undirected graphs have edges that do not have a direction. The edges indicate a two-way relationship, in that each edge can be traversed in both directions. This figure shows a simple undirected graph with three nodes and three edges.

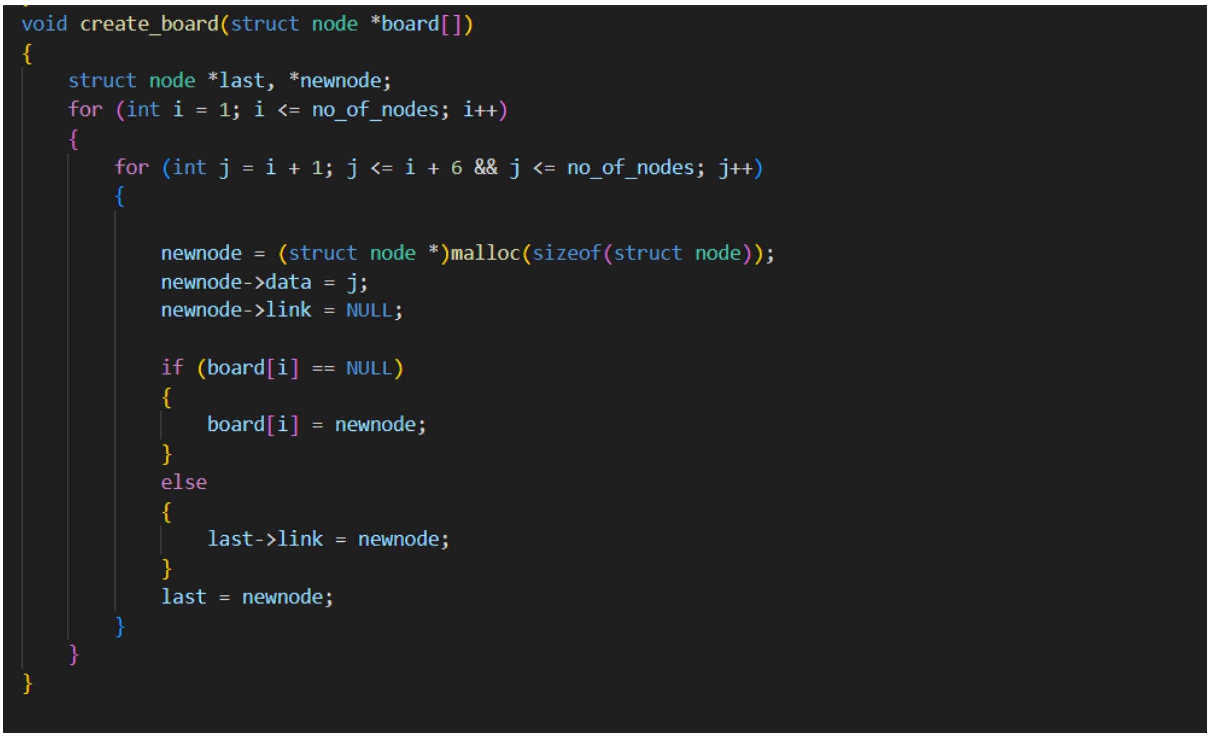
- DIRECTED GRAPH



A directed graph, also called a digraph, is a graph in which the edges have a direction. This is usually indicated with an arrow on the edge; more formally, if v and w are vertices, an edge is an unordered pair {v,w}, while a directed edge, called an arc, is an ordered pair (v,w) or (w,v). The arc (v,w)is drawn as an arrow from v to w. If a graph contains both arcs (v,w) and (w,v), this is not a "multiple edge", as the arcs are distinct. It is possible to have multiple arcs, namely, an arc (v,w).

FUNCTION USED

1. Creation of board [void create_board( ) ;]



```c
void create_board(struct node *board[])
{
    struct node *last, *newnode;
    for (int i = 1; i <= no_of_nodes; i++)
    {
        for (int j = i + 1; j <= i + 6 && j <= no_of_nodes; j++)
        {

            newnode = (struct node *)malloc(sizeof(struct node));
            newnode->data = j;
            newnode->link = NULL;

            if (board[i] == NULL)
            {
                board[i] = newnode;
            }
            else
            {
                last->link = newnode;
            }
            last = newnode;
        }
    }
}
```

# Creates the board as a graph by representing it as an adjacency list or adjacency matrix

* It takes the struct node pointer array(board) as arguments

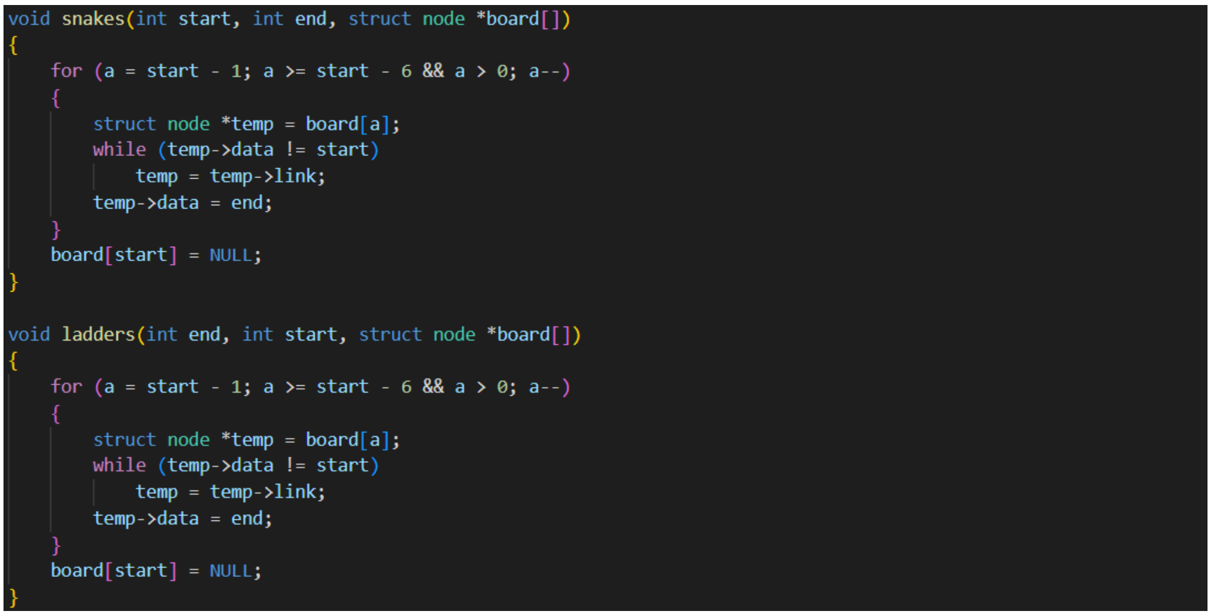
* Create a for loop from cell 1 to to the last cell

* Create another for loop inside the main for loop for adding the adjacent elements to each vertex

* Create a new node dynamically for each element and link it to the previous node in LinkedList.

* The board is created in the form of graph represented as adjacency list

2. Snakes & Ladder [void snakes( ) ;] [void ladders( ) ;]



```c
void snakes(int start, int end, struct node *board[])
{
    for (a = start - 1; a >= start - 6 && a > 0; a--)
    {
        struct node *temp = board[a];
        while (temp->data != start)
            temp = temp->link;
        temp->data = end;
    }
    board[start] = NULL;
}

void ladders(int end, int start, struct node *board[])
{
    for (a = start - 1; a >= start - 6 && a > 0; a--)
    {
        struct node *temp = board[a];
        while (temp->data != start)
            temp = temp->link;
        temp->data = end;
    }
    board[start] = NULL;
}
```

snakes ():

* Updates the already created board by adding the snakes in the graph representation

* It takes the arguments head of the snake as start to the tail of the snake as end

* Replaces the starting vertex of snake with ending vertex of snake wherever it occurs by using for loop

* it replaces the starting node of snake by making it as null

ladders ():

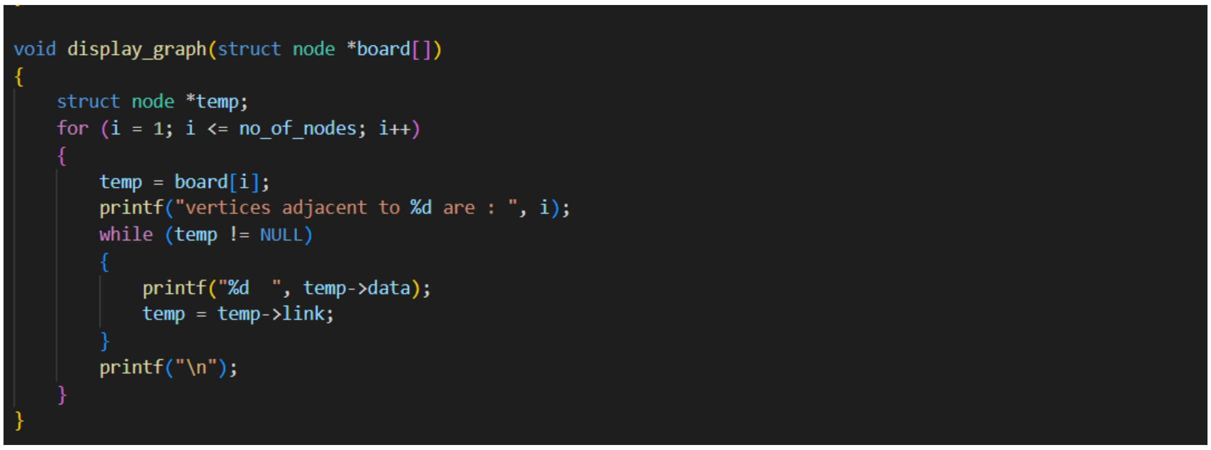
* Updates the already created board by adding the snakes in the graph representation

* It takes the arguments start of the ladder to the end destination of the ladder

* Replaces the ending vertex of ladder with starting vertex of ladder wherever it occurs

* it replaces the ending node of ladder by making it as null

3. Display of the graph [void display_graph( ) ;]



```c
void display_graph(struct node *board[])
{
    struct node *temp;
    for (i = 1; i <= no_of_nodes; i++)
    {
        temp = board[i];
        printf("vertices adjacent to %d are : ", i);
        while (temp != NULL)
        {
            printf("%d  ", temp->data);
            temp = temp->link;
        }
        printf("\n");
    }
}
```

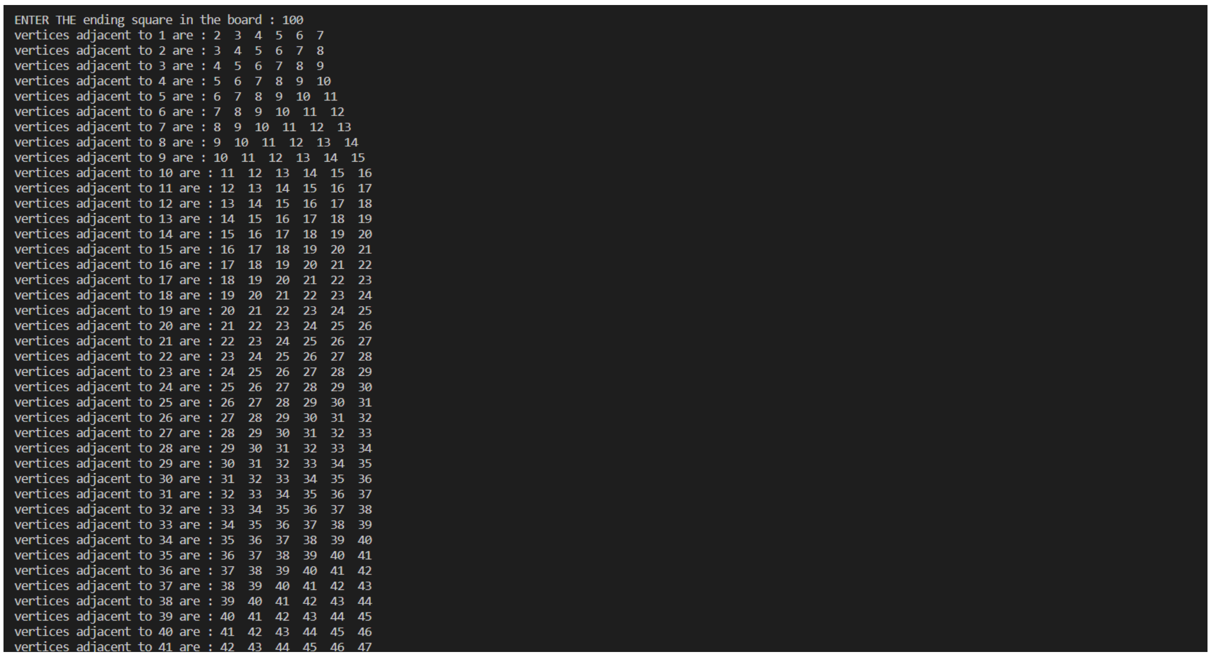
display_graph():

* Displays the board in the form of graph representation either in adjacency matrix or adjacency list

* It takes the arguments as board

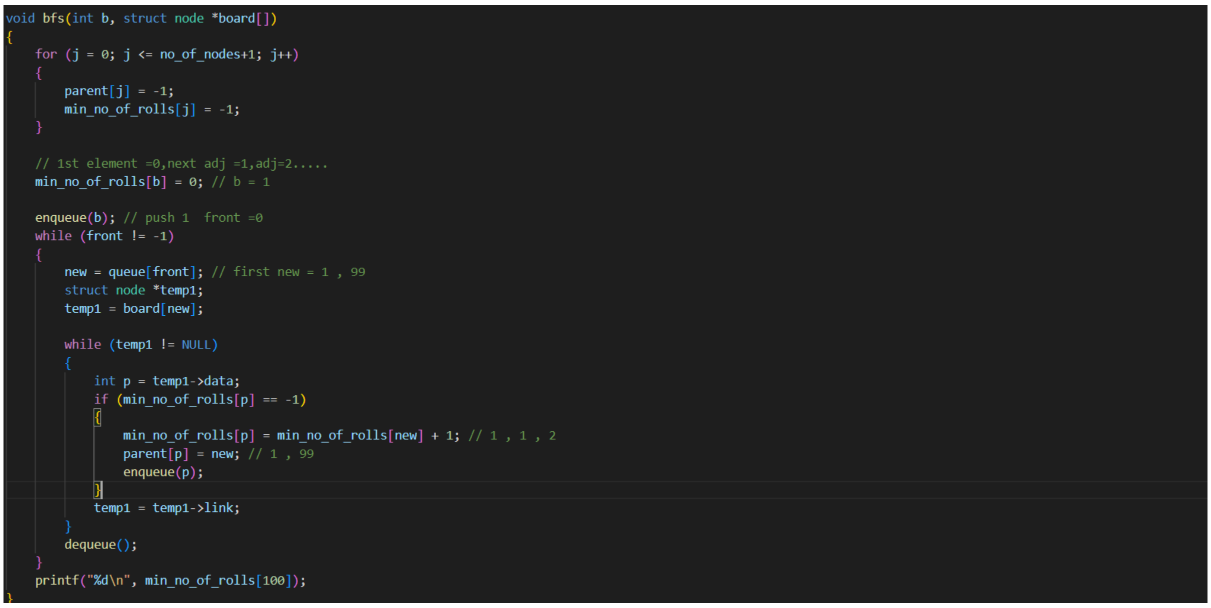
* By taking the for loop it prints every vertex with all of its adjacent vertices

OUTPUT: -



```
ENTER THE ending square in the board : 100
vertices adjacent to 1 are : 2  3  4  5  6  7
vertices adjacent to 2 are : 3  4  5  6  7  8
vertices adjacent to 3 are : 4  5  6  7  8  9
vertices adjacent to 4 are : 5  6  7  8  9  10
vertices adjacent to 5 are : 6  7  8  9  10  11
vertices adjacent to 6 are : 7  8  9  10  11  12
vertices adjacent to 7 are : 8  9  10  11  12  13
vertices adjacent to 8 are : 9  10  11  12  13  14
vertices adjacent to 9 are : 10  11  12  13  14  15
vertices adjacent to 10 are : 11  12  13  14  15  16
vertices adjacent to 11 are : 12  13  14  15  16  17
vertices adjacent to 12 are : 13  14  15  16  17  18
vertices adjacent to 13 are : 14  15  16  17  18  19
vertices adjacent to 14 are : 15  16  17  18  19  20
vertices adjacent to 15 are : 16  17  18  19  20  21
vertices adjacent to 16 are : 17  18  19  20  21  22
vertices adjacent to 17 are : 18  19  20  21  22  23
vertices adjacent to 18 are : 19  20  21  22  23  24
vertices adjacent to 19 are : 20  21  22  23  24  25
vertices adjacent to 20 are : 21  22  23  24  25  26
vertices adjacent to 21 are : 22  23  24  25  26  27
vertices adjacent to 22 are : 23  24  25  26  27  28
vertices adjacent to 23 are : 24  25  26  27  28  29
vertices adjacent to 24 are : 25  26  27  28  29  30
vertices adjacent to 25 are : 26  27  28  29  30  31
vertices adjacent to 26 are : 27  28  29  30  31  32
vertices adjacent to 27 are : 28  29  30  31  32  33
vertices adjacent to 28 are : 29  30  31  32  33  34
vertices adjacent to 29 are : 30  31  32  33  34  35
vertices adjacent to 30 are : 31  32  33  34  35  36
vertices adjacent to 31 are : 32  33  34  35  36  37
vertices adjacent to 32 are : 33  34  35  36  37  38
vertices adjacent to 33 are : 34  35  36  37  38  39
vertices adjacent to 34 are : 35  36  37  38  39  40
vertices adjacent to 35 are : 36  37  38  39  40  41
vertices adjacent to 36 are : 37  38  39  40  41  42
vertices adjacent to 37 are : 38  39  40  41  42  43
vertices adjacent to 38 are : 39  40  41  42  43  44
vertices adjacent to 39 are : 40  41  42  43  44  45
vertices adjacent to 40 are : 41  42  43  44  45  46
vertices adjacent to 41 are : 42  43  44  45  46  47
```

4. Breadth First Search [void bfs( ) ;]



```c
void bfs(int b, struct node *board[])
{
    for (j = 0; j <= no_of_nodes+1; j++)
    {
        parent[j] = -1;
        min_no_of_rolls[j] = -1;
    }

    // 1st element =0,next adj =1,adj=2.....
    min_no_of_rolls[b] = 0; // b = 1

    enqueue(b); // push 1  front =0
    while (front != -1)
    {
        new = queue[front]; // first new = 1 , 99
        struct node *temp1;
        temp1 = board[new];

        while (temp1 != NULL)
        {
            int p = temp1->data;
            if (min_no_of_rolls[p] == -1)
            {
                min_no_of_rolls[p] = min_no_of_rolls[new] + 1; // 1 , 1 , 2
                parent[p] = new; // 1 , 99
                enqueue(p);
            }
            temp1 = temp1->link;
        }
        dequeue();
    }
    printf("%d\n", min_no_of_rolls[100]);
}
```

bfs ():

* Implements the breadth first search algorithm on the board and returns the minimum number of dice rolls required to reach the last square

* It takes two arguments as board and the starting argument

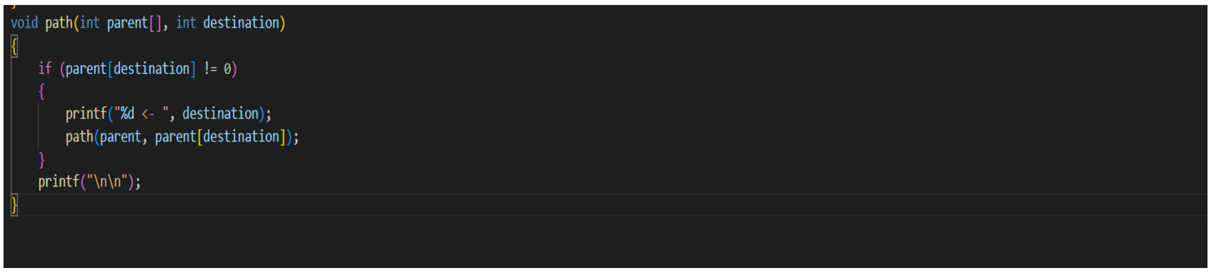
* we define two arrays as parent: and min no of nodes: by setting all the entries as -1

- parent node is to keep track of the path of the bfs

- min no of roles[i] gives the no of die rolls to the ith cell

* We perform bfs by using a queue

5. Shortest path [void path( ) ;]



```c
void path(int parent[], int destination)
{
    if (parent[destination] != 0)
    {
        printf("%d <- ", destination);
        path(parent, parent[destination]);
    }
    printf("\n\n");
}
```

path ():

* Gives the shortest path for reaching the last cell of the board

* It takes the arguments as parent and destination

* Until parent[destination]! = 0

- we recursively call this function until it reaches the destination

* To display the path taken by the bfs algorithm

/*Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. This is done by

considering that the player can determine which number appears in the dice being biased. The player rolls the dice and if reaches a base of a ladder then he can move up

the ladder to a different position/cell and if he reaches a snake then it brings him down away from the destination cell position.*/

```
#include <stdio.h>

#include <stdlib.h>

#define max 100

int no_of_nodes, i, j, ns, end, start, a, nl, star, en, x;

int queue[max], front = -1, rear = -1, new;

int parent[101],var5,var6;

int min_no_of_rolls[101];

int var = 1;

struct node

{

    int data;

    struct node *link;

};

void path(int[], int);

int dequeue();

void enqueue(int);

void display_graph(struct node *[]);

void bfs(int, struct node *[]);

void snakes(int, int, struct node *[]);
void ladders(int, int, struct node *[]);
void create_board(struct node *[]);
void main()
{
    printf("\nENTER THE ending square in the board : ");
    scanf("%d", &no_of_nodes);

    struct node *board[no_of_nodes + 1];
    for (i = 1; i <= no_of_nodes; i++)
    {
       board[i] = NULL;
    }
    create_board(board);

    display_graph(board);

    printf("\nEnter the number of snakes : ");
    scanf("%d", &ns);
    for (i = 1; i < ns + 1; i++)
    {
       printf("Enter the starting cell and ending value of snake %d : \n ", i);
       scanf("%d%d", &end, &start);
       snakes(end, start, board);
    }

    printf("\nEnter the number of Ladders  : ");
    scanf("%d", &nl);
```

```
    for (i = 1; i < nl + 1; i++)
    {
        printf("Enter the starting cell and ending value of  Ladder %d : \n ", i);
        scanf("%d%d", &star, &en);
        ladders(en, star, board);
    }

    display_graph(board);

    //printf("\nEnter from which element you want to search : ");
    //scanf("%d",&var5);
    bfs(1, board);
    //printf("\nEnter the destination cell : ");
    //scanf("%d",&var6);
    //printf("\nshortest path from %d to %d is : ",var5,var6);
    path(parent, 100);
}
int dequeue()
{
    int y;
    if (front == -1)
        printf("\nunderflow\n");
    else
    {
        y = queue[front];
        if (front == rear)
        {
            front = -1;
            rear = -1;
        }
        else
        {
            front++;
        }
        return y;
    }
}
void enqueue(int x)
{
    if (rear == max - 1)
        printf("overflow\n");
    else
    {
        if (rear == -1 && front == -1)
        {
            front = 0;
            rear = 0;
        }
        else
            rear++;
        queue[rear] = x;
    }
}
```

```
void display_graph(struct node *board[])
{
    struct node *temp;
    for (i = 1; i <= no_of_nodes; i++)
    {
        temp = board[i];
        printf("vertices adjacent to %d are : ", i);
        while (temp != NULL)
        {
            printf("%d  ", temp->data);
            temp = temp->link;
        }
        printf("\n");
    }
}

void bfs(int b, struct node *board[])
{
    for (j = 0; j <= no_of_nodes + 1; j++)
    {
        parent[j] = -1;
        min_no_of_rolls[j] = -1;
    }

    // 1st element =0,next adj =1,adj=2.....
    min_no_of_rolls[b] = 0; // b = 1
    enqueue(b); // push 1  front =0
    while (front != -1)
    {
        new = queue[front]; // first new = 1 , 99
        struct node *temp1;
        temp1 = board[new];

        while (temp1 != NULL)
        {
            int p = temp1->data;
            if (min_no_of_rolls[p] == -1)
            {
                min_no_of_rolls[p] = min_no_of_rolls[new] + 1; // 1 , 1 , 2
                parent[p] = new;                               // 1 , 99
                enqueue(p);
            }
            temp1 = temp1->link;
        }
        dequeue();
    }
    printf("%d\n", min_no_of_rolls[100]);
}
void snakes(int start, int end, struct node *board[])
{
    for (a = start - 1; a >= start - 6 && a > 0; a--)
    {
        struct node *temp = board[a];
        while (temp->data != start)
            temp = temp->link;
        temp->data = end;
    }
    board[start] = NULL;
}

void ladders(int end, int start, struct node *board[])
{
    for (a = start - 1; a >= start - 6 && a > 0; a--)
    {
        struct node *temp = board[a];
        while (temp->data != start)
            temp = temp->link;
        temp->data = end;
    }
    board[start] = NULL;
}
void create_board(struct node *board[])
{
    struct node *last, *newnode;
    for (int i = 1; i <= no_of_nodes; i++)
    {
        for (int j = i + 1; j <= i + 6 && j <= no_of_nodes; j++)
        {

            newnode = (struct node *)malloc(sizeof(struct node));
            newnode->data = j;
            newnode->link = NULL;

            if (board[i] == NULL)
            {
                board[i] = newnode;
            }
            else
            {
                last->link = newnode;
            }
            last = newnode;
        }
    }
}
void path(int parent[], int destination)
{
    if (parent[destination] != 0)
    {
        printf("%d <- ", destination);
        path(parent, parent[destination]);
    }
    printf("\n\n");
}
```