**INTRODUCTION:**

DATA STRUCTURES

Data Structures are structures programme to store ordered data ,so that various operations can be performed on it easily.

It represents the knowledge of data to be organized in memory.

It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

Classification of Data Structures

1.Primitive Data Structure

2.Non-Primitive Data Structure

The most commonly used operations on data structure are:

1.Create

2.Selection

3.Updating

4.Searching

5.Sorting

6.Merging

7.Destroy or Delete

Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

* Linked List
* Tree
* Graph
* Stack, Queue etc.
* All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.

# Algorithm:

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1.Time Complexity

2.Space Complexity

## Usage

Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval ,while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

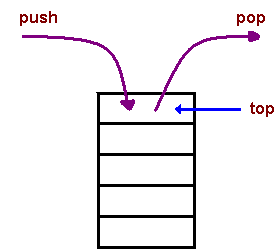
## Implementation

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking).

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

STACK

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



**Methods in stack**

**1. Push Method:** This method is used for adding an item in to the Stack. If stack is full, then Overflow message will be displayed.

**2. Pop Method:** This method is used for Deleting an item from Stack. If stack is empty, then Underflow message will be displayed.

**3. Peek Method:** This method displays top element in the stack.

**4. IsEmpty Method:** This method returns TRUE if stack is empty, else FALSE.

**5. IsFull Method:** This method returns TRUE if stack is full, else FALSE.

**Implementation**

Stack can be implemented in two ways:

1. Using Arrays

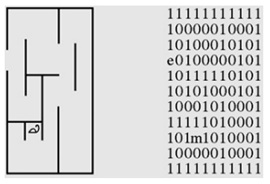
2. Using Linked List

**Applications of stack:**

* Balancing of symbols
* Infix to Postfix /Prefix conversion
* Redo-undo features at many places like editors, photoshop.
* Forward and backward feature in web browsers
* Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
* Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
* In Graph Algorithms like Topological Sorting and Strongly Connected Components

**2.Problem Statement**

Consider a mouse trapped in a maze, trying to find the exit:  
This mouse can only move right, left, down, or up—one step at a time. It applies the following procedure:

[](https://1.bp.blogspot.com/-BMf8rOK0POM/W-UVKVECzwI/AAAAAAAAQ5o/ggEgGJVb8ksnbH54m1gKqx80gnuUIi7jACLcBGAs/s1600/MAZE.jpg)

-> Try moving right, left, down, and up If a route beginning with any one of these fails,  
 -> Try a new route that starts in an untried direction.  
The maze is implemented as a 2D array of char’s. 1 = wall,  0 = open, m = mouse’s initial position, e = exit (could be anywhere),  .(dot) = visited location. We assume the boundaries of the array are walls.  
Find out appropriate data structure to be used to solve the problem. If appropriate data structure found, implement the program, if not, justify your answer why the following data structure cannot be used to implement. (Analyze using Stack, Queue, Linked List (singly/doubly), Tree, Heap).

**3. ANALYSIS :**

**4. DESIGN:**

Select the first Row and column array. Add the amount

The next row is it zero?

Yes

No

No

Add column value to the current one

My next column is Zero?

Yes

No

The final element is whether the rows and

No

The route is finished

Yes

No

There is no way final.

**5**. IMPLEMENTATION:

PROGRAM:

#include <stdio.h>

#include<stdlib.h>

#define SIZE 11

int maze[SIZE][SIZE] = { {1,1,1,1,1,1,1,1,1,1,1}, {1,0,0,0,0,0,1,0,0,0,1},{1,0,1,0,0,0,1,0,1,0,1},{0,0,1,0,0,0,0,0,1,0,1},{1,0,1,1,1,1,1,0,1,0,1},{1,0,1,0,1,0,0,0,1,0,1},{1,0,0,0,1,0,1,0,0,0,1},{1,1,1,1,1,0,1,0,0,0,1},{1,0,1,0,1,0,1,0,0,0,1},{1,0,0,0,0,0,1,0,0,0,1},{1,1,1,1,1,1,1,1,1,1,1}}; //matrix to store the solution

int solution[SIZE][SIZE];

int stack[100],top,y,k;

void push()

{

if(top>=100-1)

printf("Stack is overflow\n");

else {

top++;

stack[top]=1;

}

}

int pop()

{

if(top<=-1)

printf("Stack is underflow\n");

else

{

return stack[top--];

}

}

void printsolution()

{

int i,j;

printf("Input:\n");

for(i=0;i<SIZE;i++)

{

for(j=0;j<SIZE;j++)

{

printf("%d\t",maze[i][j]);

}

printf("\n\n");

}

printf("Output:\n");

for(i=0;i<SIZE;i++)

{ for(j=0;j<SIZE;j++)

{

if(solution[i][j]==0)

{//solution[i][j]='\*';

printf(".\t",solution[i][j]);}

if(solution[i][j]==1)

{

printf("1\t",solution[i][j]);

}}

printf("\n\n");

}

}

int solvemaze(int r, int c)

{

(maze[SIZE-8][SIZE-11]);

if((r==SIZE-8) && (c==SIZE-11))

{

solution[r][c] = 0;

return 1;

}

if(r>=0 && c>=0 && r<SIZE && c<SIZE && solution[r][c] == 1 && maze[r][c] == 0)

{

solution[r][c] = 0;

if(solvemaze(r+1, c))

return 1;

if(solvemaze(r, c+1))

{//return 1;

push();

y=pop();

return y;

}

if(solvemaze(r-1, c))

{//return 1;

push();

y=pop();

return y;

}

if(solvemaze(r, c-1))

{//return 1;

push();

y=pop();

return y;

}

solution[r][c] = 1;

return 0;

}

return 0;

}

int main() {

top=-1;

int i,j,x;

for(i=0; i<SIZE; i++)

{

for(j=0; j<SIZE; j++)

{

solution[i][j] = 1;

}

}

if (solvemaze(8,3)) printsolution();

else

printf("No solution\n");

return 0;

}

6.RESULTS AND OUTCOMES:

