**Report on Data Structure**

**MAHBUBUR RAHMAN**

**ID : 171442523**



**MAHBUBUR RAHMAN**

**ID : 171442523**

Contents

[Chapter: 01 4](#_Toc529136399)

[1D & 2D Array 4](#_Toc529136400)

[2D array: 6](#_Toc529136401)

[Chapter: 02 8](#_Toc529136402)

[Stack 8](#_Toc529136403)

[Stack in Programming Terms 8](#_Toc529136404)

[Stack Specification 9](#_Toc529136405)

[How stack works 9](#_Toc529136406)

[Stack Implementation in programming language 9](#_Toc529136407)

[Chapter: 03 12](#_Toc529136408)

[Queue: 12](#_Toc529136409)

[Queue Specifications 12](#_Toc529136410)

[How Queue Works 13](#_Toc529136411)

[Queue Implementation in programming language 15](#_Toc529136412)

[Implementation using C programming 15](#_Toc529136413)

[Chapter: 04 17](#_Toc529136414)

[Linked List 18](#_Toc529136415)

[Linked List Representation 18](#_Toc529136416)

[How another node is referenced? 18](#_Toc529136417)

[Complete program for linked list operations is showing below 22](#_Toc529136418)

[Utility of Linked List 26](#_Toc529136419)

[Chapter: 05 27](#_Toc529136420)

[Binary Search Tree (BST) 27](#_Toc529136421)

[Representation 27](#_Toc529136422)

[Basic Operations 27](#_Toc529136423)

[Chapter: 5.1 28](#_Toc529136424)

[Chapter: 5.2 29](#_Toc529136425)

[Chapter: 06 30](#_Toc529136426)

[Breadth First Search (BFS) 30](#_Toc529136427)

[Implementation in C 33](#_Toc529136428)

[Chapter: 07 38](#_Toc529136429)

[DFS (Depth First Search) 38](#_Toc529136430)

[Implementation in C: 38](#_Toc529136431)

[Chapter: 08 42](#_Toc529136432)

[Sorting Algorithm 42](#_Toc529136433)

[Implementation in C: 43](#_Toc529136434)

[Implementation in C: 45](#_Toc529136435)

[References 51](#_Toc529136436)

# Chapter: 01

## 1D & 2D Array

 C supports a derived data type known as array. • An array is a fixed-sized sequenced collection of elements of the same data type thats shares a common name. • The common name a is the array name and each individual data item is known as an element of the array. • The elements of the array are stored in the subsequent memory locations starting from the memory location given by the array name.

Array can be classified as two types are,

1. One-Dimensional Array

2. Two-Dimensional or multidimensional Array\

 One-Dimensional Array ◦ A one-dimensional array can be used to represent a list of data items. It is also known as a vector.

Two-Dimensional Array ◦ A two dimensional array can be used to represent a table of data items consisting of rows and columns. It is also known as a matrix

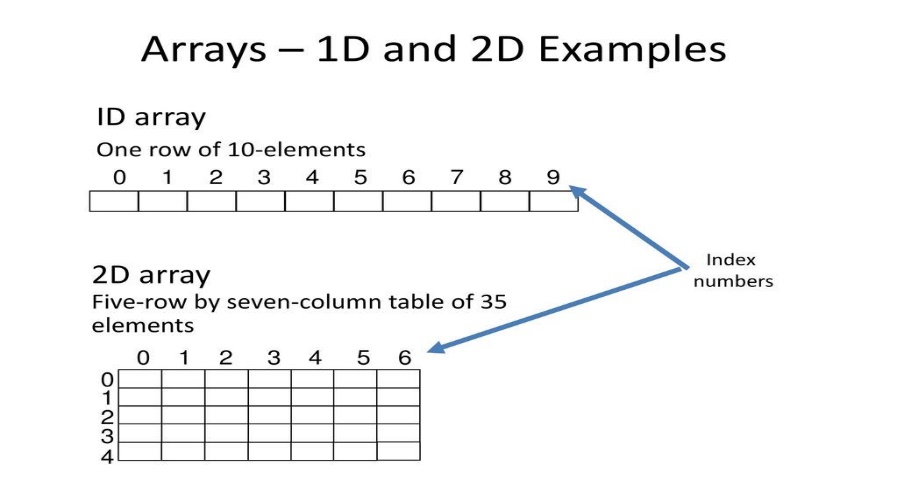
Array declaration for 1D & 2D will be,

**int** mark[5]; // 1D array declaration

**int** mark[2][3]; // 2D array declaration

in 2D array there’s will be row and column number.

Graphical examples of 1D and 2D are showing below,



Easy Programs of 1D and 2D array :

The following program prints the sum of elements of an array.

#include<stdio.h>

int main()

{

int arr[5], i, s = 0;

for(i = 0; i < 5; i++)

{

printf("Enter a[%d]: ", i);

scanf("%d", &arr[i]);

}

for(i = 0; i < 5; i++)

{

s += arr[i];

}

printf("\nSum of elements = %d ", s);

// signal to operating system program ran fine

return 0;

}

Expected Output:

Enter a[0]: 22

Enter a[1]: 33

Enter a[2]: 56

Enter a[3]: 73

Enter a[4]: 23

Sum of elements = 207

### 2D array:

#include<stdio.h>

void main(){

int a[10][10];

int i,j,rows, columns;

printf("How many rows you want \n");

scanf("%d", &rows);

printf("How many columns you want \n");

scanf("%d", &columns);

printf("Enter your array Element one by one \n");

for(i=0;i<rows;i++){

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

scanf("%d", &a[i][j]);

}

}

printf("Your array \n");

for(i=0;i<rows;i++){

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

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

}

printf("\n");

}

}

**Output will be:**

How many rows you want

2

How many columns you want

2

Enter your array Element one by one

2

2

# Chapter: 02

## Stack

A stack is a useful data structure in programming. It is just like a pile of plates kept on top of each other.

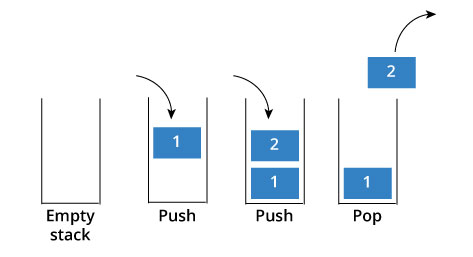


Think about the things you can do with such a pile of plates

* Put a new plate on top
* Remove the top plate

If you want the plate at the bottom, you have to first remove all the plates on top. Such kind of arrangement is called **Last In First Out** - the last item that was placed is the first item to go out.

**Stack in Programming Terms**

In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop"

In the above image, although item 2 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

We can implement stack in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

**Stack Specification**

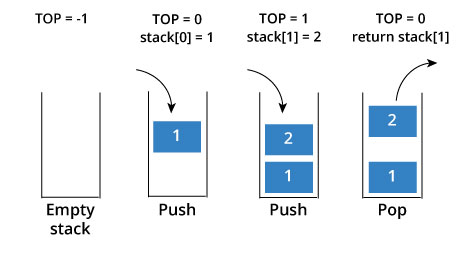
A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

* Push: Add element to top of stack
* Pop: Remove element from top of stack
* IsEmpty: Check if stack is empty
* IsFull: Check if stack is full
* Peek: Get the value of the top element without removing it

**How stack works**

The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if stack is already full
6. Before popping, we check if stack is already empty



### Stack Implementation in programming language

The most common stack implementation is using arrays, but it can also be implemented using lists.

Here is an implementation using arrays and C programming language.

#include <stdio.h>

int stack[20]; //Stack declaration

int head = -1; //Stack initially empty

void push(int data){

head++;

stack[head] = data;

}

int pop(){

int data = stack[head];

head--;

return data;

}

void printstack(){

printf("Data in your stack\n");

int i;

for(i=0;i<=head;i++){

printf("%d ",stack[i]);

}

}

void main()

{

push(5);

push(7);

push(10);

printstack();

int data = pop();

printf("\nYour pop data: %d \n",data);

printstack();

}

**Output will be:**

Data in your stack

5 7 10

Your pop data: 10

Data in your stack

5 7

#### Use of stack

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

* **To reverse a word** - Put all the letters in a stack and pop them out. Because of LIFO order of stack, you will get the letters in reverse order.
* **In compilers** - Compilers use stack to calculate the value of expressions like 2+4/5\*(7-9) by converting the expression to prefix or postfix form.
* **In browsers** - The back button in a browser saves all the urls you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack and the previous url is accessed.

# Chapter: 03

## Queue:

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

We can implement queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

## Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

* Enqueue: Add element to end of queue
* Dequeue: Remove element from front of queue
* IsEmpty: Check if queue is empty
* IsFull: Check if queue is full
* Peek: Get the value of the front of queue without removing it

## How Queue Works

Queue operations work as follows:

1. Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 0.
8. When dequeing the last element, we reset the values of FRONT and REAR to -1



## Queue Implementation in programming language

The most common queue implementation is using arrays, but it can also be implemented using lists.

**Implementation using C programming**

#include<stdio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int items[SIZE], front = -1, rear = -1;

int main()

{

//deQueue is not possible on empty queue

deQueue();

//enQueue 5 elements

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

//6th element can't be added to queue because queue is full

enQueue(6);

display();

//deQueue removes element entered first i.e. 1

deQueue();

//Now we have just 4 elements

display();

return 0;

}

void enQueue(int value){

if(rear == SIZE-1)

printf("\nQueue is Full!!");

else {

if(front == -1)

front = 0;

rear++;

items[rear] = value;

printf("\nInserted -> %d", value);

}

}

void deQueue(){

if(front == -1)

printf("\nQueue is Empty!!");

else{

printf("\nDeleted : %d", items[front]);

front++;

if(front > rear)

front = rear = -1;

}

}

void display(){

if(rear == -1)

printf("\nQueue is Empty!!!");

else{

int i;

printf("\nQueue elements are:\n");

for(i=front; i<=rear; i++)

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

}

}

When you run this program, you get the output

Queue is Empty!!

Inserted -> 1

Inserted -> 2

Inserted -> 3

Inserted -> 4

Inserted -> 5

Queue is Full!!

Queue elements are:

1 2 3 4 5

Deleted : 1

Queue elements are:

2 3 4 5

# 

# Chapter: 04

## Linked List

In a game of Treasure Hunt, you start by looking for the first clue. When you find it, instead of having the treasure, it has the location of the next clue and so on. You keep following the clues until you get to the treasure.

A linked list is similar. It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.

## Linked List Representation



You have to start somewhere, so we give the address of the first node a special name called HEAD.

Also, the last node in the linkedlist can be identified because its next portion points to NULL.

## How another node is referenced?

Some pointer magic is involved. Let's think about what each node contains:

* A data item
* An address of another node

We wrap both the data item and the next node reference in a struct as:

struct node

{

int data;

struct node \*next;

};

Understanding the structure of a linked list node is the key to having a grasp on it.

Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data=3;

/\* Connect nodes \*/

one->next = two;

two->next = three;

three->next = NULL;

/\* Save address of first node in head \*/

head = one;

If you didn't understand any of the lines above, all you need is a refresher on [pointers and structs](https://www.programiz.com/c-programming/c-structures-pointers).

In just a few steps, we have created a simple linkedlist with three nodes.



The power of linkedlist comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

* Create a new struct node and allocate memory to it.
* Add its data value as 4
* Point its next pointer to the struct node containing 2 as data value
* Change next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

**How to traverse a linked list**

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is NULL, we know that we have reached the end of linked list so we get out of the while loop.

struct node \*temp = head;

printf("\n\nList elements are - \n");

while(temp != NULL)

{

printf("%d --->",temp->data);

temp = temp->next;

}

The output of this program will be:

List elements are -

1 --->2 --->3 --->

**How to add elements to linked list**

We can add elements to either beginning, middle or end of linked list. Here we will see only the beginning & end operations.

**Add to beginning**

* Allocate memory for new node
* Store data
* Change next of new node to point to head
* Change head to point to recently created node

For an example,

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

newNode->next = head;

head = newNode;

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

newNode->next = NULL;

struct node \*temp = head;

while(temp->next != NULL){

temp = temp->next;

}

temp->next = newNode;

**Add to end**

* Allocate memory for new node
* Store data
* Traverse to last node
* Change next of last node to recently created node

**How to delete from a linked list**

You can delete either from beginning, end or from a particular position.

**Delete from beginning**

* Point head to the second node

head = head->next;

**Delete from end**

* Traverse to second last element
* Change its next pointer to null

struct node\* temp = head;

while(temp->next->next!=NULL){

temp = temp->next;

}

temp->next = NULL;

Complete program for linked list operations is showing below**,**

#include <stdio.h>

struct Node

{

int data;

struct Node \*next;

};

struct Node \*head = NULL;

void insertAtBeginning(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if(head == NULL)

{

head = newNode;

}

else

{

newNode->next = head;

head = newNode;

}

}

void display()

{

if(head == NULL)

{

printf("\nList is Empty\n");

}

else

{

struct Node \*temp = head;

printf("\n\nList elements are - \n");

while(temp->next != NULL)

{

printf("%d ",temp->data);

temp = temp->next;

}

printf("%d ",temp->data);

}

}

void insertAtEnd(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if(head == NULL)

head = newNode;

else

{

struct Node \*temp = head;

while(temp->next != NULL){

temp = temp->next;

}

temp->next = newNode;

}

}

void removeBeginning()

{

if(head == NULL)

printf("\n\nList is Empty!!!");

else

{

struct Node \*temp = head;

if(temp->next == NULL)

{

head = NULL;

free(temp);

}

else

{

head = temp->next;

free(temp);

}

}

}

void removeEnd()

{

if(head == NULL)

{

printf("\nList is Empty!!!\n");

}

else

{

struct Node \*temp1 = head,\*temp2;

if(temp1->next == NULL)

{

head = NULL;

}

else

{

while(temp1->next != NULL)

{

temp2 = temp1;

temp1 = temp1->next;

}

temp2->next = NULL;

}

free(temp1);

}

}

int main()

{

insertAtEnd(10);// 10

insertAtEnd(20); // 10 20

insertAtBeginning(40); // 40 10 20

insertAtBeginning(50); // 50 40 10 20

insertAtEnd(100); // 50 40 10 20 100

display();

removeBeginning(); // 40 10 20 100

display();

removeEnd(); // 40 10 20

display();

return 0;

}

**Output will be:**

List elements are -

50 40 10 20 100

List elements are -

40 10 20 100

List elements are -

40 10 20

### Utility of Linked List

Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java and C#.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

# Chapter: 05

## Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties,

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

**left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)**

## Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

## Basic Operations

Following are the basic operations of a tree −

* **Search** − Searches an element in a tree.
* **Insert** − Inserts an element in a tree.
* **Pre-order Traversal** − Traverses a tree in a pre-order manner.
* **In-order Traversal** − Traverses a tree in an in-order manner.
* **Post-order Traversal** − Traverses a tree in a post-order manner.

**Node**

Define a node having some data, references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

# Chapter: 5.1

Given a [**Binary Search Tree**](https://www.geeksforgeeks.org/binary-search-tree-data-structure/) which is also a Complete Binary Tree. The problem is to convert a given BST into a Max Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Max Heap.

**Examples:**

**Input:** 4

/ \

2 6

/ \ / \

1 3 5 7

**Output:** 7

/ \

3 6

/ \ / \

1 2 4 5

The given **BST** has been transformed into a

**Max Heap.**

All the nodes in the Max Heap satisfies the given

condition, that is, values in the left subtree of

a node should be less than the values in the right

subtree of the node.

# Chapter: 5.2

Given a binary search tree which is also a complete binary tree. The problem is to convert the given BST into a Min Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Min Heap.

**Examples:**

**Input**: 4

/ \

2 6

/ \ / \

1 3 5 7

**Output**: 1

/ \

2 5

/ \ / \

3 4 6 7

The given **BST** has been transformed into a

Min Heap.

All the nodes in the Min Heap satisfies the given

condition, that is, values in the left subtree of

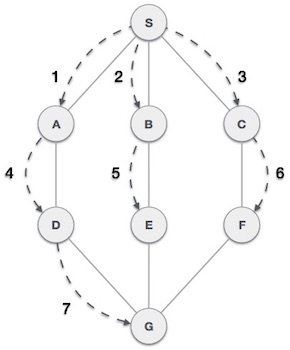
a node should be less than the values in the right

subtree of the node.

# Chapter: 06

## Breadth First Search (BFS)

Breadth First Search (BFS) algorithm traverses a graph in a Breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

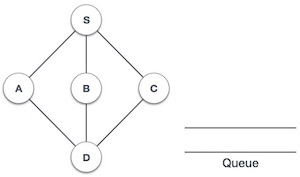
* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Breadth First Search Step One | Initialize the queue. |
| 2 | Breadth First Search Step Two | We start from visiting S(starting node), and mark it as visited. |
| 3 | Breadth First Search Step Three | We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it. |
| 4 | Breadth First Search Step Four | Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it. |
| 5 | Breadth First Search Step Five | Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it. |
| 6 | Breadth First Search Step Six | Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A. |
| 7 | Breadth First Search Step Seven | From A we have D as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

The implementation of this algorithm in C programming language can be [seen](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal_in_c.htm)below,

We shall not see the implementation of Breadth First Traversal (or Breadth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model −



## Implementation in C

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 5

struct Vertex {

char label;

bool visited;

};

//queue variables

int queue[MAX];

int rear = -1;

int front = 0;

int queueItemCount = 0;

//graph variables

//array of vertices

struct Vertex\* lstVertices[MAX];

//adjacency matrix

int adjMatrix[MAX][MAX];

//vertex count

int vertexCount = 0;

//queue functions

void insert(int data) {

queue[++rear] = data;

queueItemCount++;

}

int removeData() {

queueItemCount--;

return queue[front++];

}

bool isQueueEmpty() {

return queueItemCount == 0;

}

//graph functions

//add vertex to the vertex list

void addVertex(char label) {

struct Vertex\* vertex = (struct Vertex\*) malloc(sizeof(struct Vertex));

vertex->label = label;

vertex->visited = false;

lstVertices[vertexCount++] = vertex;

}

//add edge to edge array

void addEdge(int start,int end) {

adjMatrix[start][end] = 1;

adjMatrix[end][start] = 1;

}

//display the vertex

void displayVertex(int vertexIndex) {

printf("%c ",lstVertices[vertexIndex]->label);

}

//get the adjacent unvisited vertex

int getAdjUnvisitedVertex(int vertexIndex) {

int i;

for(i = 0; i<vertexCount; i++) {

if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)

return i;

}

return -1;

}

void breadthFirstSearch() {

int i;

//mark first node as visited

lstVertices[0]->visited = true;

//display the vertex

displayVertex(0);

//insert vertex index in queue

insert(0);

int unvisitedVertex;

while(!isQueueEmpty()) {

//get the unvisited vertex of vertex which is at front of the queue

int tempVertex = removeData();

//no adjacent vertex found

while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {

lstVertices[unvisitedVertex]->visited = true;

displayVertex(unvisitedVertex);

insert(unvisitedVertex);

}

}

//queue is empty, search is complete, reset the visited flag

for(i = 0;i<vertexCount;i++) {

lstVertices[i]->visited = false;

}

}

int main() {

int i, j;

for(i = 0; i<MAX; i++) // set adjacency {

for(j = 0; j<MAX; j++) // matrix to 0

adjMatrix[i][j] = 0;

}

addVertex('S'); // 0

addVertex('A'); // 1

addVertex('B'); // 2

addVertex('C'); // 3

addVertex('D'); // 4

addEdge(0, 1); // S - A

addEdge(0, 2); // S - B

addEdge(0, 3); // S - C

addEdge(1, 4); // A - D

addEdge(2, 4); // B - D

addEdge(3, 4); // C - D

printf("\nBreadth First Search: ");

breadthFirstSearch();

return 0;

}

If we compile and run the above program, it will produce the following result −

**Output:**

Breadth First Search: S A B C D

# Chapter: 07

## DFS (Depth First Search)

[Depth First Traversals](https://www.geeksforgeeks.org/618/)

* + In-order Traversal (Left-Root-Right)
  + Pre-order Traversal (Root-Left-Right)
  + Post-order Traversal (Left-Right-Root)



**DFS** of given Tree

**Depth First Traversals:**

**Pre-order Traversal:** 1 2 4 5 3

**In-order Traversal:** 4 2 5 1 3

**Post-order Traversal:** 4 5 2 3 1

## Implementation in C:

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node\* left;

struct node\* right;

};

void inorder(struct node\* root){

if(root == NULL) return;

inorder(root->left);

printf("%d ->", root->data);

inorder(root->right);

}

void preorder(struct node\* root){

if(root == NULL) return;

printf("%d ->", root->data);

preorder(root->left);

preorder(root->right);

}

void postorder(struct node\* root) {

if(root == NULL) return;

postorder(root->left);

postorder(root->right);

printf("%d ->", root->data);

}

struct node\* createNode(value){

struct node\* newNode = malloc(sizeof(struct node));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

struct node\* insertLeft(struct node \*root, int value) {

root->left = createNode(value);

return root->left;

}

struct node\* insertRight(struct node \*root, int value){

root->right = createNode(value);

return root->right;

}

int main(){

struct node\* root = createNode(1);

insertLeft(root, 12);

insertRight(root, 9);

insertLeft(root->left, 5);

insertRight(root->left, 6);

insertLeft(root->right, 10);

insertRight(root->right, 12);

insertRight(root->left->left, 20);

insertRight(root->left->right, 30);

printf("In-order traversal \n");

inorder(root);

printf("\nPre-order traversal \n");

preorder(root);

printf("\nPost-order traversal \n");

postorder(root);

}

**Output of this program will be:**

**In-order traversal**

5 🡺20 🡺12 🡺6 🡺30 🡺1 🡺10 🡺9 🡺12 🡺

**Pre-order traversal**

1 🡺12 🡺5 🡺20 🡺6 🡺30 🡺9 🡺10 🡺12 🡺

**Post-order traversal**

20 🡺5 🡺30 🡺6 🡺12 🡺10 🡺12 🡺9 🡺1 🡺

Which one is better between BFS & DFS?

* In BFS the space complexity is more critical as compared to time complexity. while

In DFS has lesser space complexity, because at a time it needs to store only single path from the root to leaf node.

* Breadth First Search can be done with the help of queue i.e, FIFO(First In First Out)while

Depth First Search can be done with the help of stack i.e. LIFO(Last In First Out).

* BFS is slower than DFS while

DFS is more faster than BFS.

* BFS requires more memory compare to DFS while

DFS require less memory compare to BFS.

* BFS is useful in finding shortest path while

DFS in not so useful in finding shortest path.

So, in my opinion DFS is better than BFS

# 

# Chapter: 08

## Sorting Algorithm

A Sorting Algorithm is used to rearrange a given array elements according to a comparison operator on the elements. For an example,

A B I R H O S S A I N = = = > A B H I I N O R S S

**How Selection Sort Works?**

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.



## Implementation in C:

#include <stdio.h>

void selectionSort(int A[], int n){

int temp,iMin,i,j;

for( i=0;i<n;i++){

iMin= i;

for( j=i+1;j<n;j++){

if(A[j]<A[iMin])

{

iMin=j;

}

}

/\* To sort in descending order, change > to <. \*/

temp=A[i];

A[i]=A[iMin];

A[iMin]=temp;

}

}

int main()

{

int i;

int A[] ={10,30,6,7,11,1};

selectionSort(A,6);

printf("In ascending order: ");

for(i=0;i<6;i++)

printf("%d ",A[i]);

return 0;

}

**Output:**

**In ascending order: 1 6 7 10 11 30**

#### How Insertion Sort Works?

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

### Implementation in C:

#include <stdio.h>

void insertionSort(int A[], int n){

int hole,value,i;

for( i=0;i<n;i++){

value= A[i];

hole = i;

while(hole>0 && A[hole-1]>value){

A[hole] = A[hole-1];

hole = hole -1;

}

A[hole] = value;

}

}

int main()

{

int i;

int A[] ={10,30,6,7,11,1};

insertionSort(A,6);

printf("In ascending order: ");

for(i=0;i<6;i++)

printf("%d ",A[i]);

return 0;

}

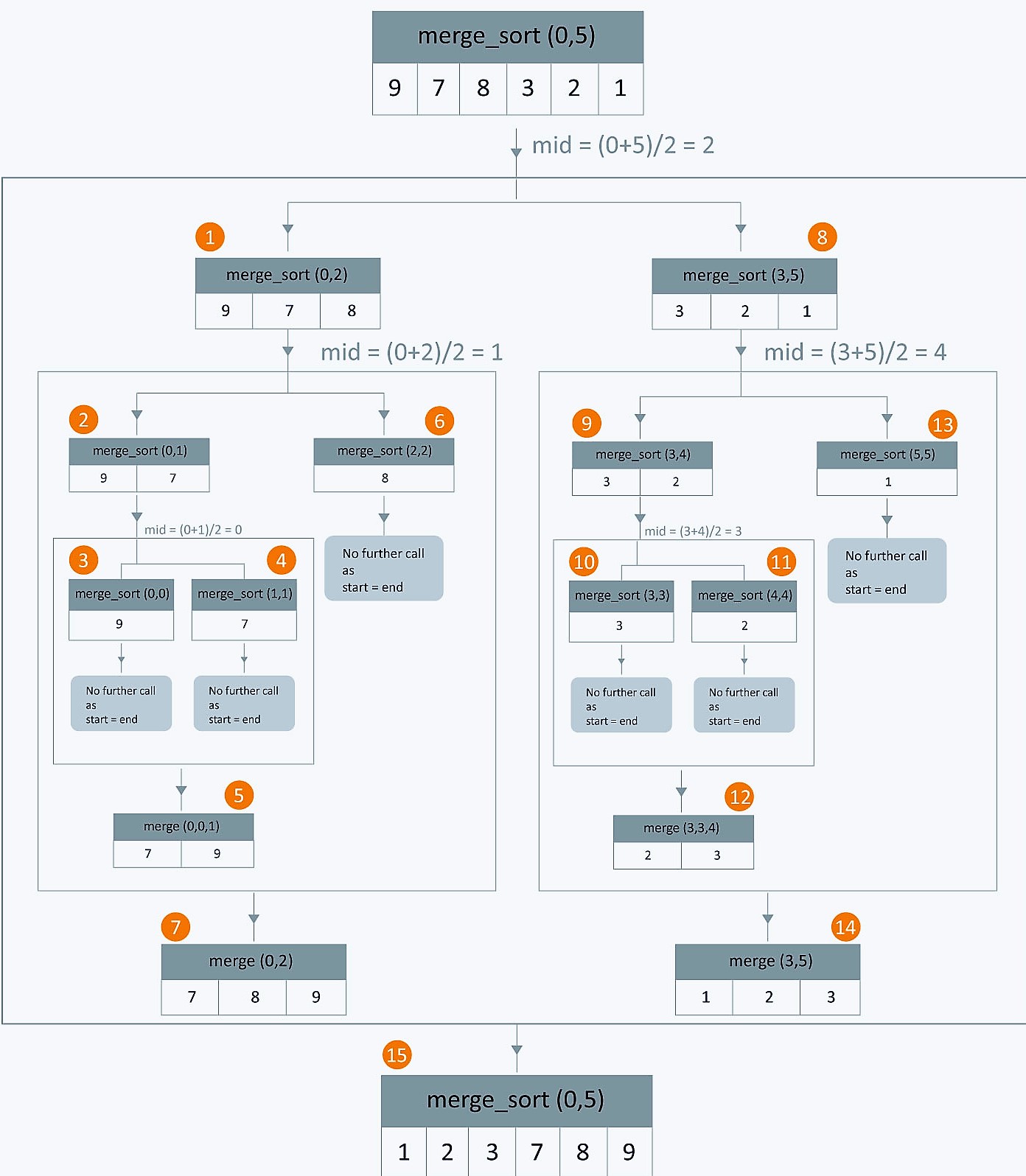
**Output:**

**In ascending order: 1 6 7 10 11 30**

##### How Merge Sort Works?

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

For an example a picture is given below,



#### Implementation in C:

#include<stdio.h>

void mergesort(int a[],int i,int j);

void merge(int a[],int i1,int j1,int i2,int j2);

int main()

{

int a[30],n,i;

printf("Enter no of elements: ");

scanf("%d",&n);

printf("Enter array elements: ");

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

scanf("%d",&a[i]);

mergesort(a,0,n-1);

printf("\nSorted array is: ");

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

printf("%d ",a[i]);

return 0;

}

void mergesort(int a[],int i,int j)

{

int mid;

if(i<j)

{

mid=(i+j)/2;

mergesort(a,i,mid); //left recursion

mergesort(a,mid+1,j); //right recursion

merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays

}

}

void merge(int a[],int i1,int j1,int i2,int j2)

{

int temp[50]; //array used for merging

int i,j,k;

i=i1; //beginning of the first list

j=i2; //beginning of the second list

k=0;

while(i<=j1 && j<=j2) //while elements in both lists

{

if(a[i]<a[j])

temp[k++]=a[i++];

else

temp[k++]=a[j++];

}

while(i<=j1) //copy remaining elements of the first list

temp[k++]=a[i++];

while(j<=j2) //copy remaining elements of the second list

temp[k++]=a[j++];

//Transfer elements from temp[] back to a[]

for(i=i1,j=0;i<=j2;i++,j++)

a[i]=temp[j];

}

**Output will be:**

**Enter no of elements: 5**

**Enter array elements: 1**

**2**

**3**

**4**

**5**

**Sorted array is: 1 2 3 4 5**

# References

Anon., n.d. [Online]   
Available at: https://www.slideshare.net/appilivamsikrishna/arrays-1d-and-2d-and-multi-dimensional

Anon., n.d. [Online]   
Available at: https://www.hackerearth.com/practice/algorithms/sorting

Anon., n.d. [Online]   
Available at: https://www.geeksforgeeks.org/convert-bst-min-heap/

Anon., n.d. [Online]   
Available at: https://www.geeksforgeeks.org/convert-bst-to-max-heap/

Anon., n.d. [Online]   
Available at: https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/

Anon., n.d. [Online]   
Available at: https://www.geeksforgeeks.org/sorting-algorithms/

Anon., n.d. [Online]   
Available at: https://www.programiz.com/dsa/linked-list

Anon., n.d. [Online]   
Available at: https://www.programiz.com/dsa/queue

Anon., n.d. [Online]   
Available at: https://www.programiz.com/dsa/stack

Anon., n.d. [Online]   
Available at: https://overiq.com/c-programming/101/one-dimensional-array-in-c/