Report On

DATA STRUCTURE

Md. Rubel Hossain

ID: 171442529

Contents

Chapter:01

1D & 2D Array

An array is a collection of data that holds fixed number of values of same type. For example, if we want to store marks of 100 students, we can create an array for it.

float marks[100];

here, marks is the array name and float is its value type and we have declared array's size to 100.

There are two types of array. One is 1D and another is 2D or multidimensional array.

Array declaration for 1D & 2D will be,
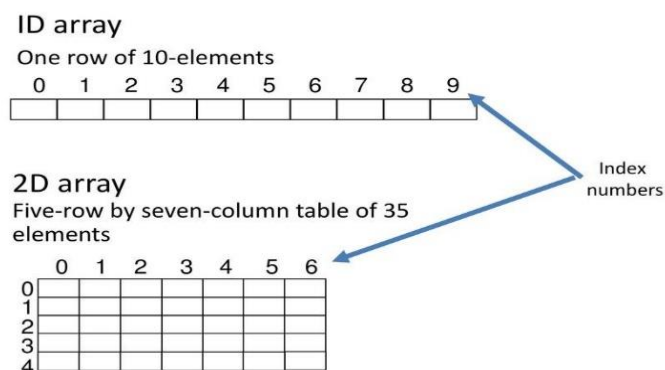
int mark[5];   // 1D array declaration

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,

Simple programs of 1D and 2D array,

1D array:

```c
#include<stdio.h>

int main()

{

int a[10];

inti, n;

printf("How many element you want to save \n");

scanf("%d", &n);

printf("Enter element one by one \n");

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

{

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

}

printf("The List you entered\n");

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

{

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

}

return 1;

}
```

Output will be:

How many element you want to save

3

Enter element one by one

10

20

30

The List you entered

10 20 30

2D array:

```c
#include<stdio.h>

void main(){

int a[10][10];

inti,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
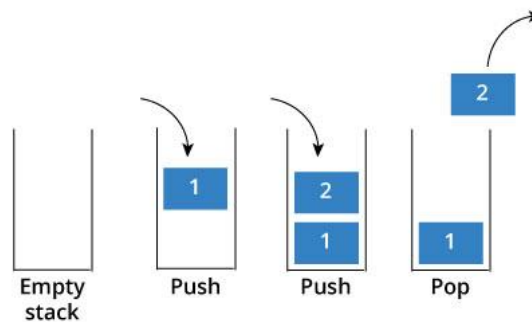
2

2

Your array

2   2

2   2

Chapter:02

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.
Basic concept of stack is showing graphically below,

Empty stack   Push   Push   Pop

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.
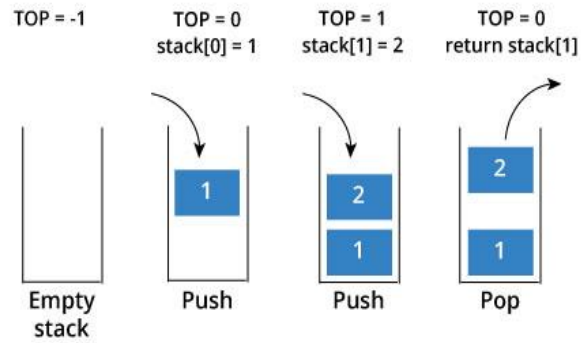
Stack Implementation in C language:

```c
#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;

}

voidprintstack(){
printf("Data in your stack\n");
inti;
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 URL's 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:

Queue arranges element in such a way that new element always gets added in the back and also when comes the removal it removes the last or oldest element or which is the first element in the queue. It is also known as FIFO or First In First Out.
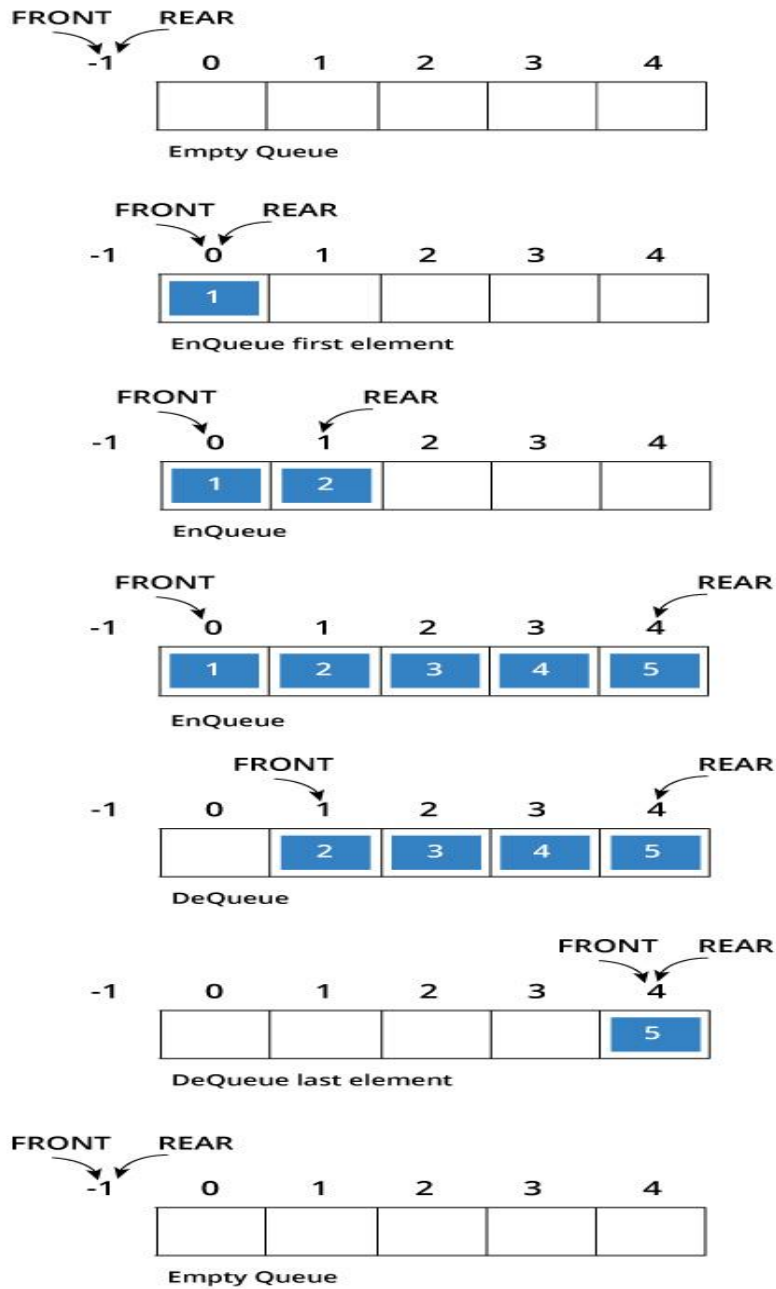
Basic of Queue is showing graphically below,

Empty Queue   EnQueue   EnQueue   DeQueue
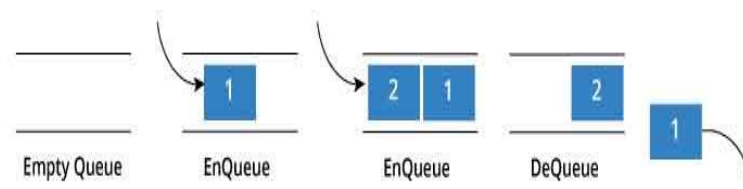
How Queue Works:

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.

FRONT REAR
-1

0 1 2 3 4

Empty Queue

FRONT REAR
-1 0 1 2 3 4

1

EnQueue first element

FRONT REAR
-1 0 1 2 3 4

1 2

EnQueue

FRONT REAR
-1 0 1 2 3 4

1 2 3 4 5

EnQueue

FRONT REAR
-1 0 1 2 3 4

2 3 4 5

DeQueue

FRONT REAR
-1 0 1 2 3 4

5

DeQueue last element

FRONT REAR
-1 0 1 2 3 4

Empty Queue

Queue implementation in C language:

#include<stdio.h>

#define SIZE 100   //Queue size declaration

int queue[SIZE], head = -1, tail = 0; //Empty Queue declaration

```c
voidenQueue(int value){
if(head == SIZE-1)
printf("\nQueue is Full!!!");
else{
head++;
queue[head] = value;
   }
}


voiddeQueue(){
if(head == -1)
printf("\nQueue is Empty!!! ");
else{
printf("\nDeleted: %d", queue[tail]);
tail++;
   }
}
void display(){
if(head == -1)
printf("\nQueue is Empty!!!");
else{
inti;
printf("\nQueue elements are:\n");
for(i=tail; i<=head; i++)
          printf("%d\t",queue[i]);
   }
}
```

```
void main()

{

enQueue(10);

enQueue(20);

enQueue(30);

enQueue(40);

display();

deQueue();

display();

deQueue();

display();

}
```

Output will be:

Queue elements are:

10   20   30   40

Deleted: 10

Queue elements are:

20   30   40

Deleted: 20

Queue elements are:

30   40

Chapter: 04

Linked List

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL. In C, we can represent a node using structures. Linked list consists many node structures those are point each other and create a link.

struct node   //Node Structure

{

int data;

struct node *next;

};

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

{

```c
int data;
struct Node *next;
};
struct Node *head = NULL;
voidinsertAtBeginning(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");
  }
```

```c
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);
  }
}
voidinsertAtEnd(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;
  }
```

```c
}
voidremoveBeginning()
{
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);
     }
   }
}

voidremoveEnd()
{
if(head == NULL)
   {
printf("\nList is Empty!!!\n");
   }
```

```c
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

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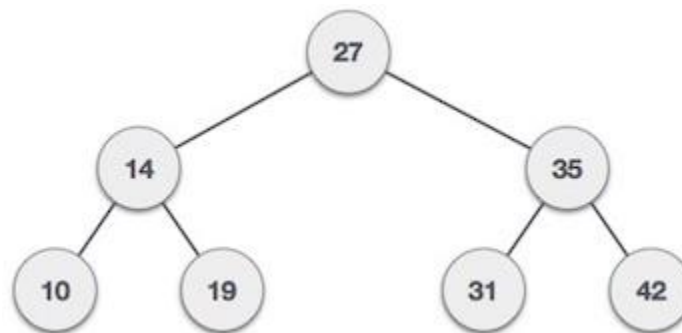
We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

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** 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
```
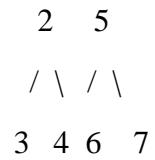
Output:
```
        1
       / \
```

```
   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
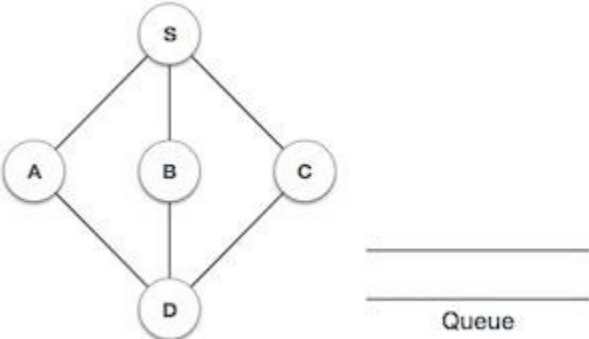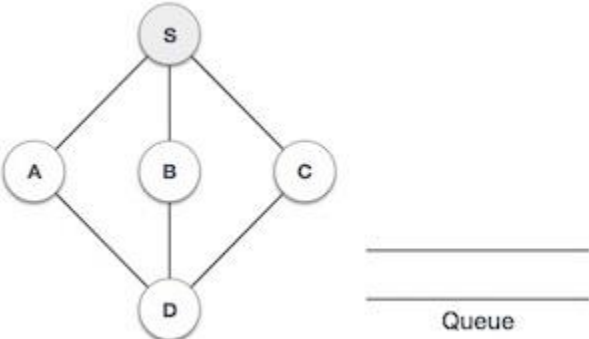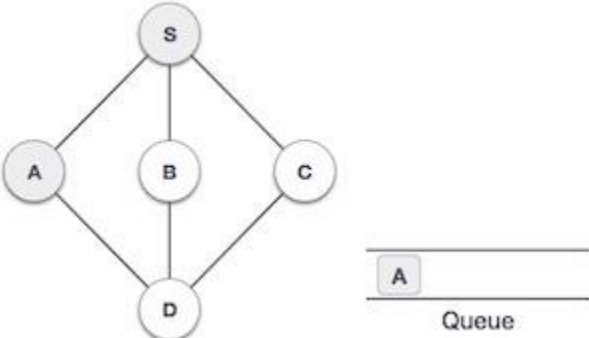
subtree of the node.
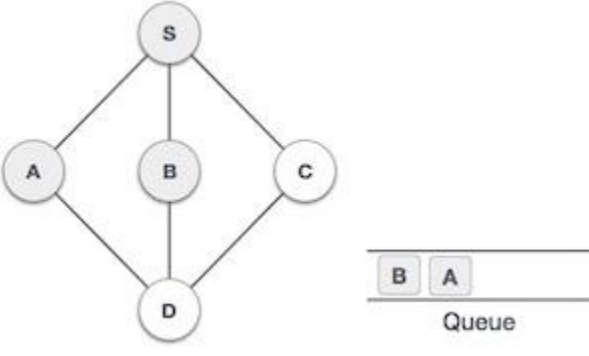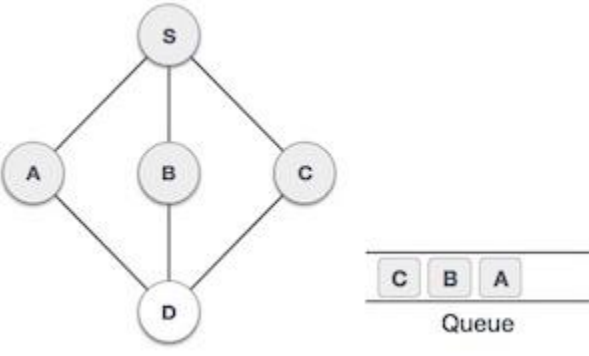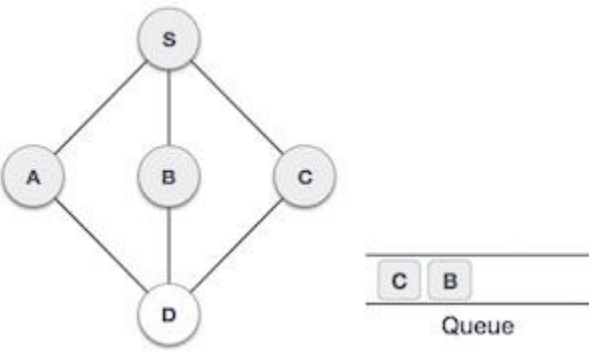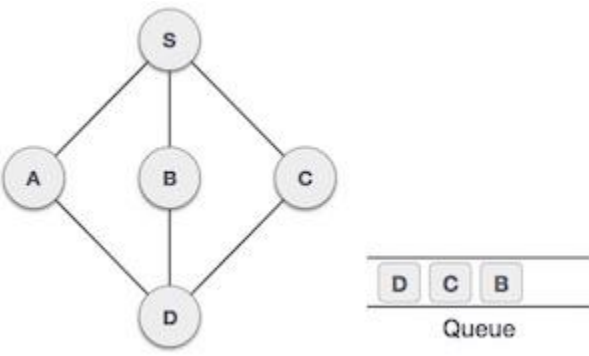
Chapter: 06

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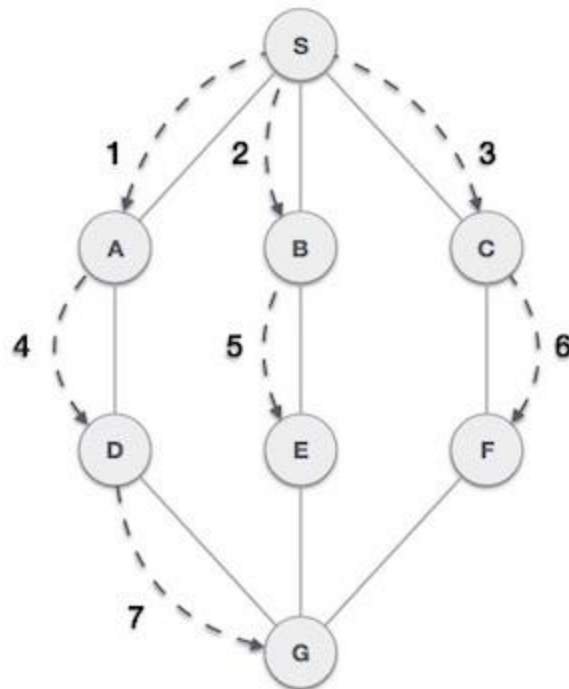
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **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	Queue	Initialize the queue.
2	Queue	We start from visiting S(starting node), and mark it as visited.
3	A Queue	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		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		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 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 −
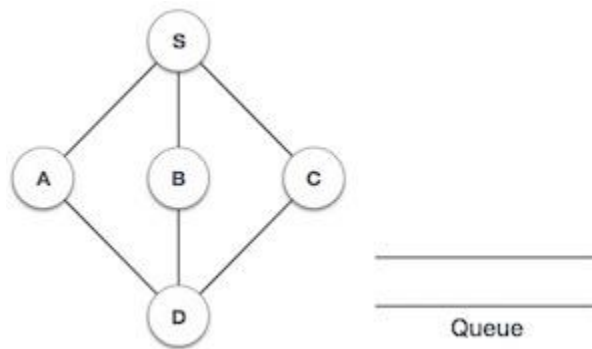
Implementation in C

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX 5


struct Vertex {

char label;

bool visited;

};


//queue variables
```

```c
int queue[MAX];

int rear = -1;

int front = 0;

intqueueItemCount = 0;


//graph variables


//array of vertices

struct Vertex* lstVertices[MAX];


//adjacency matrix

intadjMatrix[MAX][MAX];


//vertex count

intvertexCount = 0;


//queue functions


void insert(int data) {

queue[++rear] = data;

queueItemCount++;

}


intremoveData() {

queueItemCount--;

return queue[front++];

}
```

```c
boolisQueueEmpty() {
returnqueueItemCount == 0;
}


//graph functions


//add vertex to the vertex list
voidaddVertex(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
voidaddEdge(intstart,int end) {
adjMatrix[start][end] = 1;
adjMatrix[end][start] = 1;
}


//display the vertex
voiddisplayVertex(intvertexIndex) {
printf("%c ",lstVertices[vertexIndex]->label);
}


//get the adjacent unvisited vertex
intgetAdjUnvisitedVertex(intvertexIndex) {
inti;
```

```
for(i = 0; i<vertexCount; i++) {
if(adjMatrix[vertexIndex][i] == 1 &&lstVertices[i]->visited == false)
returni;
   }

return -1;
}

voidbreadthFirstSearch() {
inti;

   //mark first node as visited
lstVertices[0]->visited = true;

   //display the vertex
displayVertex(0);

   //insert vertex index in queue
insert(0);
intunvisitedVertex;

while(!isQueueEmpty()) {
    //get the unvisited vertex of vertex which is at front of the queue
inttempVertex = 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() {

inti, 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

- In-order Traversal (Left-Root-Right)

- Pre-order Traversal (Root-Left-Right)

- Post-order Traversal (Left-Right-Root)
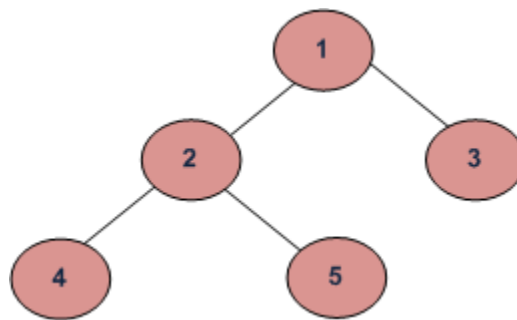
DFS of given Tree

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 {

```c
    int data;

    struct node* left;

    struct node* right;

};


voidinorder(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);

}


voidpostorder(struct node* root) {

if(root == NULL) return;

postorder(root->left);

postorder(root->right);

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

}


struct node* createNode(value){
```

```c
struct node* newNode = malloc(sizeof(struct node));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;


returnnewNode;

}


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 ➔

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   SA   IN=   =   =>A   B   H   II   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.
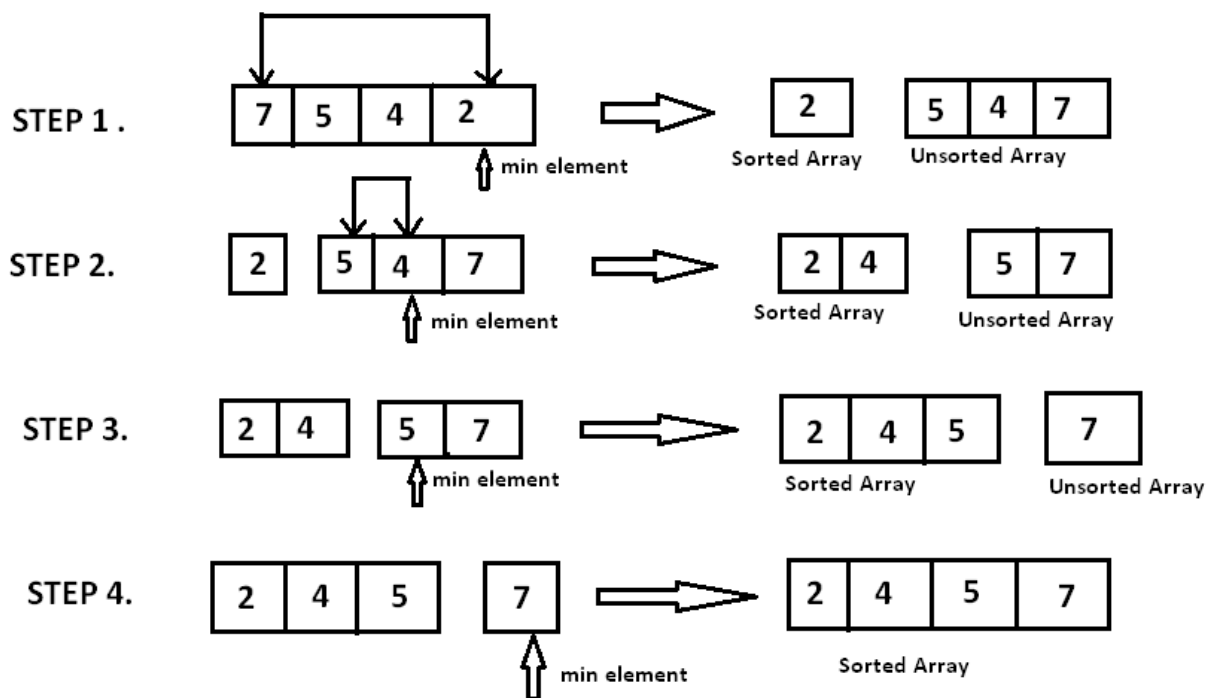
Implementation in C:

```c
#include <stdio.h>

voidselectionSort(int A[], int n){

inttemp,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()

 {

inti;

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.

Insertion sort compares the first two elements.

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

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

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.

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

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

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

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:

```c
#include <stdio.h>

voidinsertionSort(int A[], int n){

inthole,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()

 {

inti;

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;

}
```
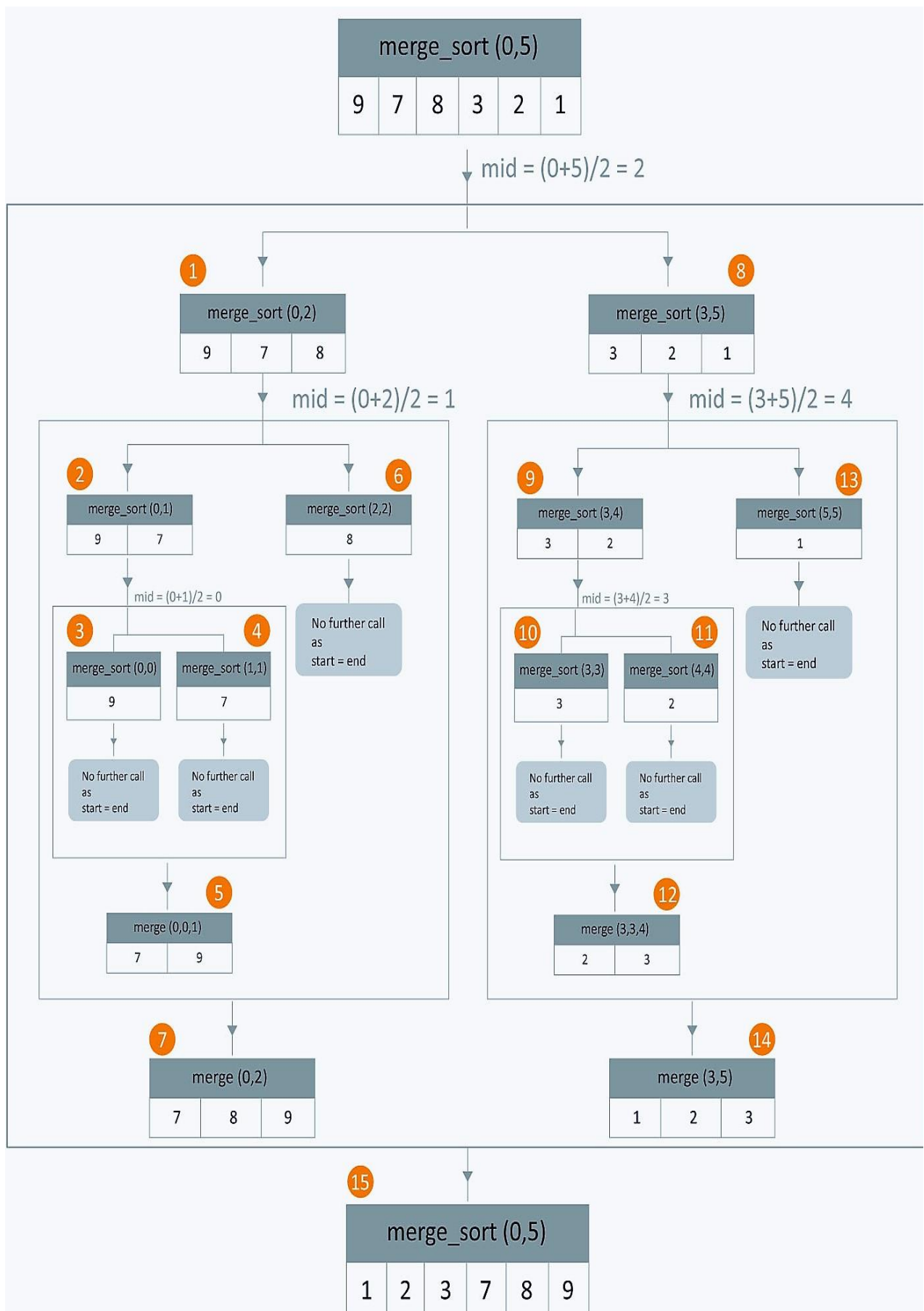
Output:

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,

merge_sort (0,5)

9	7	8	3	2	1

mid = (0+5)/2 = 2

(1) **merge_sort (0,2)**

9	7	8

(8) **merge_sort (3,5)**

3	2	1

mid = (0+2)/2 = 1

mid = (3+5)/2 = 4

(2) **merge_sort (0,1)**

9	7

(6) **merge_sort (2,2)**

8

(9) **merge_sort (3,4)**

3	2

(13) **merge_sort (5,5)**

1

mid = (0+1)/2 = 0

No further call
as
start = end

mid = (3+4)/2 = 3

No further call
as
start = end

(3) **merge_sort (0,0)**

9

(4) **merge_sort (1,1)**

7

(10) **merge_sort (3,3)**

3

(11) **merge_sort (4,4)**

2

No further call
as
start = end

No further call
as
start = end

No further call
as
start = end

No further call
as
start = end

(5) **merge (0,0,1)**

7	9

(12) **merge (3,3,4)**

2	3

(7) **merge (0,2)**

7	8	9

(14) **merge (3,5)**

1	2	3

(15) **merge_sort (0,5)**

1	2	3	7	8	9

Implementation in C:

```c
#include<stdio.h>

voidmergesort(int a[],inti,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;
}

voidmergesort(int a[],inti,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

inti,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. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/sorting-algorithms/

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

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

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

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

Anon., n.d. *indiaclass.com.* [Online]
Available at: http://www.indiaclass.com/project-report-index-format/

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

Anon., n.d. *tutorialspoint.* [Online]
Available at:
https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal_in_c.htm

Anon., n.d. *tutorialspoint.* [Online]
Available at:
https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal_in_c.htm