# Tree traversal algorithms (BFS and DFS - Preorder, Inorder, Postorder)

19.505 visualizações Estreou em 13 de abr. de 2022.

BFS (pesquisa em largura) e DFS (pesquisa em profundidade)

Trees are an important data structure because of their non-linear nature, which allows for a faster response time during a search process. So in this video, I decided to teach you about the most important algorithms and ways to search and traverse a binary tree. After watching the video you'll learn and understand the differences between BFS (breadth-first search) and DFS (depth-first search) algorithms, and also learn everything you need to know about Preorder, Inorder, and Postorder algorithms. 👉 Download Visual Assist here: [https://bit.ly/VisualAssistDownload](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbGpSbl9YaEhLYi01RVE0Vlp3RHREaW9GY1VHQXxBQ3Jtc0tualcxSzRJc0VyY2k4TEthd0pQT1lNN2paUHZmV2NCSGxnMm9oX0xJTUZkeXZicjJmNkF4SkctMVVnbkYwM0NNdFBCWXhnS09RNjVxeVBmcWVSWXVtSTVuZk9rX2hJRVFfZ29SSUpJZWl5LUlrZlJqTQ&q=https%3A%2F%2Fbit.ly%2FVisualAssistDownload&v=_IhTp8q0Mm0) And if you want to learn how to write better code or to code faster, use this video tutorial to see some Visual Assist features that I use regularly: [https://youtu.be/so1o7AoP49A](https://www.youtube.com/watch?v=so1o7AoP49A&t=0s) Tree data structure part 1 (introduction): [https://youtu.be/UHxtjVsOTHc](https://www.youtube.com/watch?v=UHxtjVsOTHc&t=0s) Learn about Recursive functions: [https://youtu.be/MwfvXDfaZeI](https://www.youtube.com/watch?v=MwfvXDfaZeI&t=0s) 📚 Learn how to solve problems and build projects with these Free E-Books ⬇️ C++ Lambdas e-book - free download here: [https://bit.ly/freeCppE-Book](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqazZBcGJNV2dRdXUtU1FfRm9laTl4S2pHSm1WUXxBQ3Jtc0tsTVgwbFM5MmhreHhPR1lXbjVIbFJCNVhCTVVoQUo4LTYwRnFyZ0RkeG1GN2ZPZmNwOGIzWWc1aDZWc2tTZWxZaXY5SWM0SkIwSkNMZ0JBdUZERXhHQzdrZEJFRFZTb2ZNaExGYkVqV0h4cGhaVTZMUQ&q=https%3A%2F%2Fbit.ly%2FfreeCppE-Book&v=_IhTp8q0Mm0) Entire Object-Pascal step-by-step guide - free download here: [https://bit.ly/FreeObjectPascalEbook](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbXNqNWVONmNJaWo3NGl0NGRpclRPcFlRQy1Yd3xBQ3Jtc0trd3FoUXpUQlhlc0l6Z0pKd2QxUFFyMGk1b2NiQ0pXSXdZY2lhaTNjblp4VWlheXZPanJtY2E5QkJSV3M0aU92ay1GRHl6THAwNk9Tc2tPNUNhVi1oakZUdEY4TUppT3oyUHZ1MDB6eHlkRE0yRm9Iaw&q=https%3A%2F%2Fbit.ly%2FFreeObjectPascalEbook&v=_IhTp8q0Mm0) Contents: [00:00](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=0s) - Introduction [02:38](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=158s) - BFS and DFS [04:32](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=272s) - Preorder algorithm [08:38](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=518s) - Traversing a tree with recursion (explaining the code) [15:34](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=934s) - Inorder algorithm [19:49](https://www.youtube.com/watch?v=_IhTp8q0Mm0&t=1189s) - Postorder algorithm

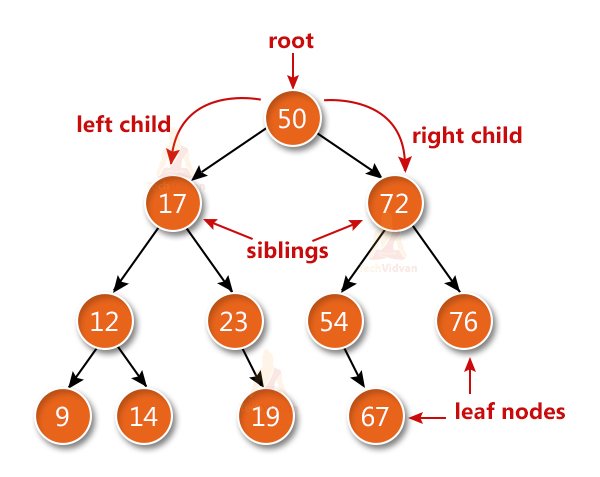
[Skip to content](https://techvidvan.com/tutorials/binary-tree-in-c/#content)

**Binary Tree in C – Types and Implementation**

C programming language provides awesome and useful features and functionalities to the programmers. In C, you can make use of some data structures which will help you in better and efficient coding. So, in this tutorial, we are going to discuss Binary Tree in C. It is one of the popular concepts of the C programming language. But firstly we will learn about Tree in C.

**What are Trees in C?**

As you know that arrays, linked lists, Stacks and Queues are linear data structures. And on the other hand, Trees are hierarchical data structures. A tree includes multiple nodes. In C, we call it a Binary Tree. A tree is referred to as a finite and non-empty set of elements in mathematical terminology.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2021/07/Binary-Trees-in-C-normal-image01.jpg)

**Tree Terminologies:-**

**1. Root:-** A root is a node without a parent. In the above image, 50 is the root node.

**2. Siblings:-** Siblings mean that nodes which have the same parent node. In the above image, 17 and 72 are siblings because they have 50 in common.

**3. Internal Node:-** Internal Node means that a node which has at least a single child. In the above image, 17, 72, 12, 23, 54 are internal nodes.

**4. External Node:-** External Node means that a node which has no children. It is also known as leaf. In the above image, 67 and 76 are external nodes.

**5. Ancestors:-** Ancestors include the parent, grandparent and so on of a node. In the above image, the ancestors of 23 are 17 and 50.

**6. Descendants:-** Descendants are the opposite of ancestors, It includes the child, grandchild and so on of a node. In the above image, the descendants of 17 are 12,23,19,9,14 and 50.

**7. Edge:-** An edge means a connection between one node to another node.

**8. Path:-** Path is a combination of nodes and edges connected with each other. In the above image, 50 to 19 is a path.

**9. Depth:-** You can calculate depth by the number of edges from node to the root of the tree.

**10. Height:-** Height is the maximum depth of a node.

**11. Level:-** Level of a node is equal to depth of the node+1.

**Applications of trees:-**

Following are some of the main applications of trees.

* Used to manipulate hierarchical data or information.
* Through trees, Searching is very easy.
* Router Algorithms.
* Used to manipulate sorted lists of data.
* With this, you can form multi stage decision making.

**Binary Tree in C:-**

A tree is called binary when its elements have at most two children. In a binary tree, each element should have only 2 children and these are known as left and right.

**Representation of Binary Tree in C:-**

The value of root is NULL when a tree is empty. It works on O(logN) for insert, search and delete operations.  
A tree node includes the following parts:-

* Data
* Pointer to the left child
* Pointer to the right child

Using structures in C, you can represent a tree node.

**Example:- A tree node with integer data**

struct node

{

int data;

struct node \*left\_child;

struct node \*right\_child;

};

In the above example, the \*left\_child is the pointer to the left child which can or cannot be NULL and the \*right\_child is the pointer to the right child which can or cannot be NULL.

**Need for binary trees:-**

In C, Binary trees have some exciting and useful applications which you can implement.

* With the help of a binary search tree, you can easily find an element in a huge set because it is fast and efficient.
* Using binary trees, you can implement heapsort.
* To store information in databases, your best way is to make use of binary trees.

**Types of Binary Tree in C:-**

In C, there are two types of binary tree such as:-

**1. Complete Binary Tree:-**

A binary tree is complete when all nodes are as far left as possible and every level except the last level is filled completely.

**2. Full Binary Tree:-**

A binary tree is called Full binary tree when each node of the tree has two children except the leafs(external nodes).

**Implementation of Binary Tree in C:-**

Similarly like Linked Lists, you can implement a binary tree in C. We use structures to implement a binary tree in C.

**1. Declaration of a binary tree:-**

First, you have to declare it before implementing it. Following is the code to declare a binary tree:-

struct node

{

int data;

struct node \*left\_child;

struct node \*right\_child;

};

**2. Creating Nodes in a binary tree:-**

It is like creating data elements in linked lists. A binary tree is created by inserting the root node and its child nodes. Following is the code to create nodes in a binary tree:-

void insert(node \*\* binary\_tree, int value) {

node \*tmp = NULL;

if(!(\*binary\_tree)) {

tmp = (node \*)malloc(sizeof(node));

tmp->left = tmp->right = NULL;

tmp->data = value;

\*binary\_tree = tmp;

return;

}

if(value < (\*binary\_tree)->data) {

insert(&(\*binary\_tree)->left, value);

} else if(value > (\*binary\_tree)->data) {

insert(&(\*binary\_tree)->right, value);

}

}

**3. Searching into a binary tree:-**

In the binary tree, you can search for the values of the nodes. Below is the code to perform a search operation on a binary tree.

node \*search(node \*\* binary\_tree, int value) {

if(!(\*binary\_tree)) {

return NULL;

}

if(value == (\*binary\_tree)->data) {

return \*binary\_tree;

} else if(value < (\*binary\_tree)->data) {

search(&((\*binary\_tree)->left), value);

} else if(value > (\*binary\_tree)->data){

search(&((\*binary\_tree)->right), value);

}

}

The above code will search for the value of a node whether the node of the same value exists or not.

**4. Deletion of a binary tree:-**

You can delete a binary tree by removing the child nodes and the root node. Below is the code snippet to delete a binary tree.

void delete\_tree(node \* binary\_tree) {

if (binary\_tree) {

delete\_tree(binary\_tree->left);

delete\_tree(binary\_tree->right);

free(binary\_tree);

}

}

**5. Displaying the binary tree:-**

You can display a binary tree in three forms such as:-

* **Pre-Order:-** It displays in order. First root node, then left node and then right node.
* **In-Order:-** It displays first left node, then root node and then right node.
* **Post-Order:-** It displays first left node, then right node and then root node.

void display\_preorder(node \* binary\_tree) {

if (binary\_tree) {

printf("%d\n",binary\_tree->data);

display\_preorder(binary\_tree->left);

display\_preorder(binary\_tree->right);

}

}

void display\_inorder(node \* binary\_tree) {

if (binary\_tree) {

display\_inorder(binary\_tree->left);

printf("%d\n",binary\_tree->data);

display\_inorder(binary\_tree->right);

}

}

void display\_postorder(node \* binary\_tree) {

if (binary\_tree) {

display\_postorder(binary\_tree->left);

display\_postorder(binary\_tree->right);

printf("%d\n",binary\_tree->data);

}

}

**Example:- Implementation of Binary Tree in C**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int value;

struct node \*left\_child, \*right\_child;

};

struct node \*new\_node(int value)

{

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

tmp->value = value;

tmp->left\_child = tmp->right\_child = NULL;

return tmp;

}

void print(struct node \*root\_node) // displaying the nodes!

{

if (root\_node != NULL)

{

print(root\_node->left\_child);

printf("%d \n", root\_node->value);

print(root\_node->right\_child);

}

}

struct node\* insert\_node(struct node\* node, int value) // inserting nodes!

{

if (node == NULL) return new\_node(value);

if (value < node->value)

{

node->left\_child = insert\_node(node->left\_child, value);

}

else if (value > node->value)

{

node->right\_child = insert\_node(node->right\_child, value);

}

return node;

}

int main()

{

printf("TechVidvan Tutorial: Implementation of a Binary Tree in C!\n\n");

struct node \*root\_node = NULL;

root\_node = insert\_node(root\_node, 10);

insert\_node(root\_node, 10);

insert\_node(root\_node, 30);

insert\_node(root\_node, 25);

insert\_node(root\_node, 36);

insert\_node(root\_node, 56);

insert\_node(root\_node, 78);

print(root\_node);

return 0;

}

**Output:-**

TechVidvan Tutorial: Implementation of a Binary Tree in C!

10  
25  
30  
36  
56  
78

**Summary**

In this tutorial, we learnt about Binary trees in C. We discussed what are trees and what are binary trees. We also discussed the terminologies of a tree and types of binary trees in C. Then we discussed the need for binary trees. We also talked about how we can implement a binary tree in C. And we performed some basic operations on a binary tree such as insertion, deletion, searching and displaying.

**Your 15 seconds will encourage us to work even harder**  
Please share your happy experience on [**Google**](https://g.page/TechVidvan/review?kd) **|** [**Facebook**](https://www.facebook.com/TechVidvan/reviews/)

Tags: [Basic Operations on Binary tree](https://techvidvan.com/tutorials/tag/basic-operations-on-binary-tree/)[Binary Tree Implementation in C](https://techvidvan.com/tutorials/tag/binary-tree-implementation-in-c/)[Binary Tree in C](https://techvidvan.com/tutorials/tag/binary-tree-in-c/)[Binary Tree Representation in C](https://techvidvan.com/tutorials/tag/binary-tree-representation-in-c/)[Types of Binary Tree in C](https://techvidvan.com/tutorials/tag/types-of-binary-tree-in-c/)

**Leave a Reply**

Parte superior do formulário

Parte inferior do formulário

* [About Us](https://techvidvan.com/tutorials/about-us/)
* [Contact Us](https://techvidvan.com/tutorials/contact-us/)
* [Terms and Conditions](https://techvidvan.com/tutorials/terms-and-conditions/)
* [Privacy Policy](https://techvidvan.com/tutorials/privacy-policy/)
* [Disclaimer](https://techvidvan.com/tutorials/disclaimer/)

TechVidvan © 2022. All Rights Reserved.

* [Home](https://www.tutorialspoint.com/index.htm)
* [Coding Ground](https://www.tutorialspoint.com/codingground.htm)
* [Jobs](https://www.tutorialspoint.com/about/about_careers.htm)
* [Whiteboard](https://www.tutorialspoint.com/whiteboard.htm)
* [Tools](https://www.tutorialspoint.com/online_dev_tools.htm)
* [Business](https://www.tutorialspoint.com/business/index.asp)

[Teach with us](https://www.tutorialspoint.com/market/teach_with_us.jsp)

[](https://www.tutorialspoint.com/index.htm)

* Category

Parte superior do formulário

Parte inferior do formulário

Parte superior do formulário

Parte inferior do formulário

* [Prime Packs](https://www.tutorialspoint.com/latest/prime-packs)
* [Courses](https://www.tutorialspoint.com/market/index.asp)
* [eBooks](https://www.tutorialspoint.com/latest/ebooks)
* [Library](https://www.tutorialspoint.com/tutorialslibrary.htm)
* [Q/A](https://www.tutorialspoint.com/questions/index.php)



* Data Structures & Algorithms
* [DSA - Home](https://www.tutorialspoint.com/data_structures_algorithms/index.htm)
* [DSA - Overview](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_overview.htm)
* [DSA - Environment Setup](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_environment.htm)
* Algorithm
* [DSA - Algorithms Basics](https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm)
* [DSA - Asymptotic Analysis](https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm)
* [DSA - Greedy Algorithms](https://www.tutorialspoint.com/data_structures_algorithms/greedy_algorithms.htm)
* [DSA - Divide and Conquer](https://www.tutorialspoint.com/data_structures_algorithms/divide_and_conquer.htm)
* [DSA - Dynamic Programming](https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm)
* Data Structures
* [DSA - Data Structure Basics](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm)
* [DSA - Array Data Structure](https://www.tutorialspoint.com/data_structures_algorithms/array_data_structure.htm)
* Linked Lists
* [DSA - Linked List Basics](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm)
* [DSA - Doubly Linked List](https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm)
* [DSA - Circular Linked List](https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm)
* Stack & Queue
* [DSA - Stack](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)
* [DSA - Expression Parsing](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm)
* [DSA - Queue](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)
* Searching Techniques
* [DSA - Linear Search](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm)
* [DSA - Binary Search](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm)
* [DSA - Interpolation Search](https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_algorithm.htm)
* [DSA - Hash Table](https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm)
* Sorting Techniques
* [DSA - Sorting Algorithms](https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm)
* [DSA - Bubble Sort](https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm)
* [DSA - Insertion Sort](https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm)
* [DSA - Selection Sort](https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm)
* [DSA - Merge Sort](https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm)
* [DSA - Shell Sort](https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm)
* [DSA - Quick Sort](https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm)
* Graph Data Structure
* [DSA - Graph Data Structure](https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm)
* [DSA - Depth First Traversal](https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm)
* [DSA - Breadth First Traversal](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm)
* Tree Data Structure
* [DSA - Tree Data Structure](https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm)
* [DSA - Tree Traversal](https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm)
* [DSA - Binary Search Tree](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm)
* [DSA - AVL Tree](https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm)
* [DSA - Spanning Tree](https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm)
* [DSA - Heap](https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm)
* Recursion
* [DSA - Recursion Basics](https://www.tutorialspoint.com/data_structures_algorithms/recursion_basics.htm)
* [DSA - Tower of Hanoi](https://www.tutorialspoint.com/data_structures_algorithms/tower_of_hanoi.htm)
* [DSA - Fibonacci Series](https://www.tutorialspoint.com/data_structures_algorithms/fibonacci_series.htm)
* DSA Useful Resources
* [DSA - Questions and Answers](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_questions_answers.htm)
* [DSA - Quick Guide](https://www.tutorialspoint.com/data_structures_algorithms/dsa_quick_guide.htm)
* [DSA - Useful Resources](https://www.tutorialspoint.com/data_structures_algorithms/dsa_useful_resources.htm)
* [DSA - Discussion](https://www.tutorialspoint.com/data_structures_algorithms/dsa_discussion.htm)
* Selected Reading
* [UPSC IAS Exams Notes](https://www.tutorialspoint.com/upsc_ias_exams.htm)
* [Developer's Best Practices](https://www.tutorialspoint.com/developers_best_practices/index.htm)
* [Questions and Answers](https://www.tutorialspoint.com/questions_and_answers.htm)
* [Effective Resume Writing](https://www.tutorialspoint.com/effective_resume_writing.htm)
* [HR Interview Questions](https://www.tutorialspoint.com/hr_interview_questions/index.htm)
* [Computer Glossary](https://www.tutorialspoint.com/computer_glossary.htm)
* [Who is Who](https://www.tutorialspoint.com/computer_whoiswho.htm)

# Data Structure and Algorithms - Tree

Advertisements

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm)

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



## Important Terms

Following are the important terms with respect to tree.

* **Path** − Path refers to the sequence of nodes along the edges of a tree.
* **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
* **Parent** − Any node except the root node has one edge upward to a node called parent.
* **Child** − The node below a given node connected by its edge downward is called its child node.
* **Leaf** − The node which does not have any child node is called the leaf node.
* **Subtree** − Subtree represents the descendants of a node.
* **Visiting** − Visiting refers to checking the value of a node when control is on the node.
* **Traversing** − Traversing means passing through nodes in a specific order.
* **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
* **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

## Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

## Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

In a tree, all nodes share common construct.

## BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

* **Insert** − Inserts an element in a tree/create a tree.
* **Search** − Searches an element in a tree.
* **Preorder Traversal** − Traverses a tree in a pre-order manner.
* **Inorder Traversal** − Traverses a tree in an in-order manner.
* **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

## Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

### Algorithm

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

goto left subtree

endwhile

insert data

end If

### Implementation

The implementation of insert function should look like this −

void insert(int data) {

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

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty, create root node

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

}

//go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

## Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

### Algorithm

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

end if

The implementation of this algorithm should look like this.

struct node\* search(int data) {

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data) {

if(current != NULL)

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

//go to left tree

if(current->data > data) {

current = current->leftChild;

}

//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

}

return current;

}

To know about the implementation of binary search tree data structure, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm).

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm) [Print Page](https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm) [Next Page](https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm)

Advertisements

[](https://www.tutorialspoint.com/)

* [About us](https://www.tutorialspoint.com/about/index.htm)
* [Refund Policy](https://www.tutorialspoint.com/about/return_refund_policy.htm)
* [Terms of use](https://www.tutorialspoint.com/about/about_terms_of_use.htm)
* [Privacy Policy](https://www.tutorialspoint.com/about/about_privacy.htm)
* [FAQ's](https://www.tutorialspoint.com/about/faq.htm)
* [Contact](https://www.tutorialspoint.com/about/contact_us.htm)

© Copyright 2022. All Rights Reserved.

We make use of First and third party cookies to improve our user experience. By using this website, you agree with our Cookies Policy. [Agree](https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm) [Learn more](https://www.tutorialspoint.com/about/about_cookies.htm)

8888888888888888888888888888888888888888888888888888888888888888888888888

# What Is Graph Traversal Algorithms in Data Structure?

Graph traversal is a search technique to find a vertex in a graph. In the search process, graph traversal is also used to determine the order in which it visits vertices. Without producing loops, a graph traversal finds the edges to be employed in the search process. That is, utilizing graph traversal, you can visit all the graph’s vertices without going through a looping path.

There are two methods for traversing graphs, which are as follows:

* **Breadth-First Search** or **BFS** Algorithm
* **Depth- First Search** or **DFS** Algorithm

In this article , we are going to learn the breadth-first search algorithm. Let’s explore.

# What Is the Breadth-First Search Algorithm?

Breadth-First Search Algorithm is the most widely utilized method. BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

According to the BFS, you must traverse the graph in a breadthwise direction:

* To begin, move horizontally and visit all the current layer’s nodes.
* Continue to the next layer.

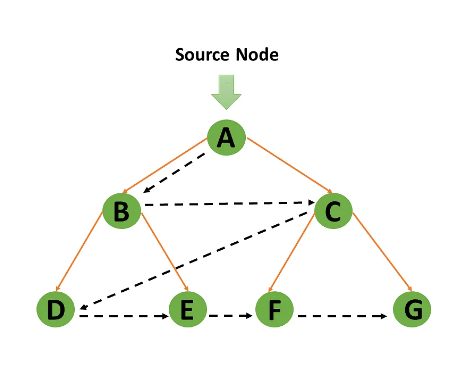


Figure : 01

A queue data structure is used by Breadth-First Search to hold the node and mark it as “visited” until it has marked all of the nearby vertices that are closely related to it. The First In First Out (FIFO) principle governs how the queue functions, thus the neighbors of each node are viewed starting with the node that was inserted first.

The below the definition of breadth-first search, you will look at why we need a breadth-first search algorithm.

# Why Do You Need Breadth-First Search Algorithm?

The BFS Algorithm should be used to traverse graph data structures for a number of reasons. Some of the essential features that make the BFS algorithm necessary include the following.

# ****What is BFS?****

BFS is an algorithm that is used to graph data or searching tree or traversing structures. The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion.

This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them.

Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked. The full form of BFS is the Breadth-first search.

* The architecture of the BFS algorithm is simple and reliable.
* The BFS algorithm supports in node evaluation in a graph and determines the shortest path traverse nodes.
* In a graph can be traversed by the BFS algorithm with the least amount of iterations.
* The BFS algorithm’s iterations are smooth, hence there is no chance for this method to get stuck in an infinite loop.
* The result of the BFS algorithm is highly accurate when compared to the other algorithms.

## Pseudocode Of Breadth-First Search Algorithm

The breadth-first search algorithm’s pseudocode is shown below.

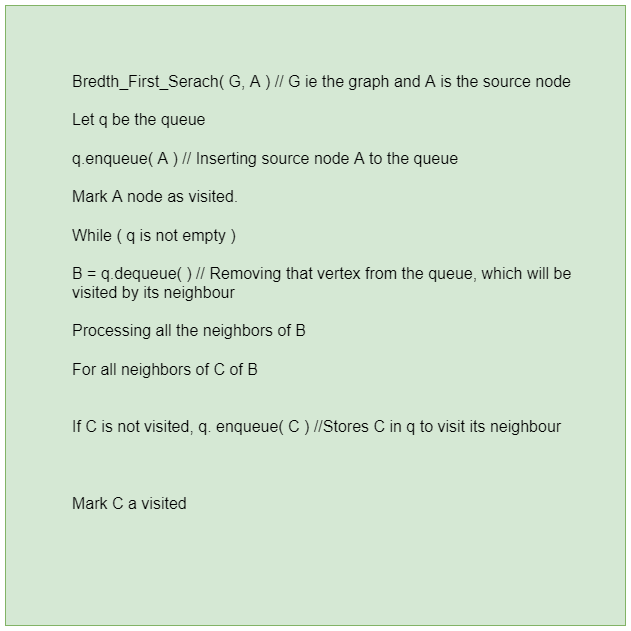


Figure : 02 Displays the breadth-first search algorithm’s pseudocode

# Example of Breadth-First Search Algorithm

In a tree-like structure, graph traversal requires the algorithm to visit, check, and update every single un-visited node. The network nodes are categorized based on the sequence in which graph traversals visit them.

An whole graph is traversed using the BFS algorithm which is starting at the first starting node. After that, traversing the first node successfully, it visits and marks the next non-traversed vertex in the graph.

**Step 1:**

In the graph, every vertex or node is known. Firstly, initialize a queue.

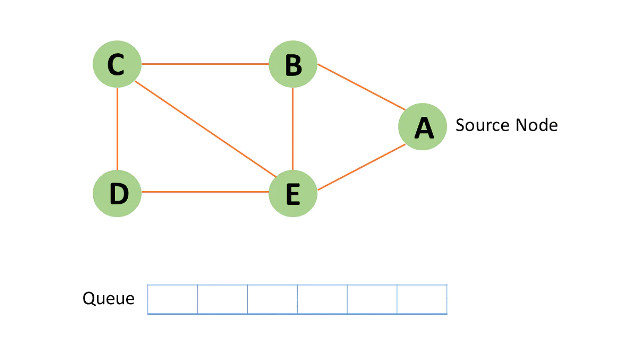


Figure : 03

**Step 2:**

In the graph, start from source node **A** and mark it as visited.

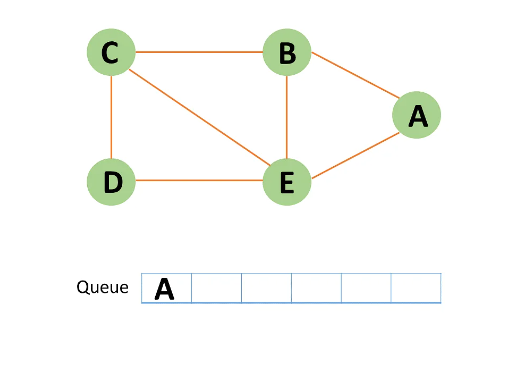


Figure : 04

**Step 3:**

Then we can observe **B** and **E**, which are unvisited nearby nodes from **A**. You have two nodes in this example, but here choose B, mark it as visited, and enqueue it alphabetically.

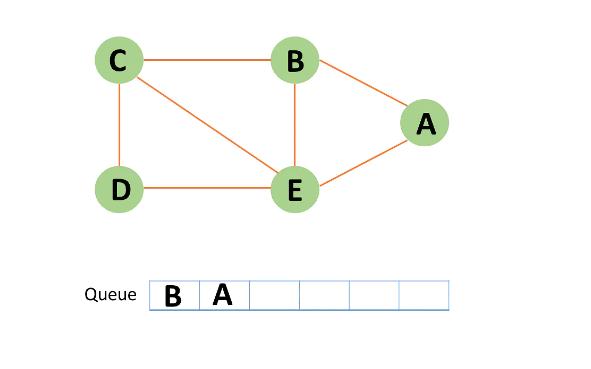


Figure : 05

**Step 4:**

Node **E** is the next unvisited neighboring node from **A**. You enqueue it after marking it as visited.

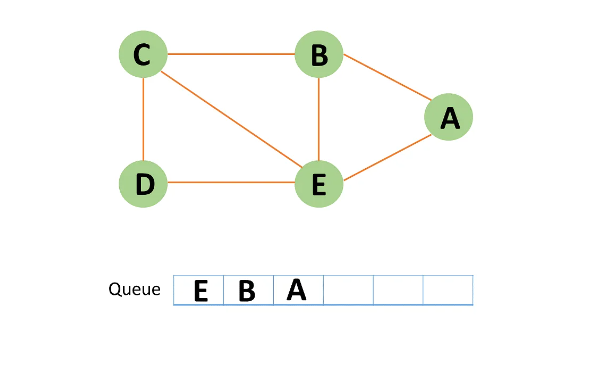


Figure : 06

**Step 5:**

**Now A** has no unvisited nodes in its immediate vicinity. As a result, you dequeue and locate **A**.

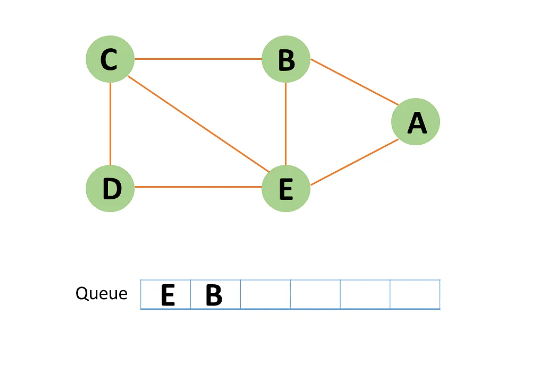


Figure : 07

**Step 6:**

Node **C** is an unvisited neighboring node from **B**. You enqueue it after marking it as visited.

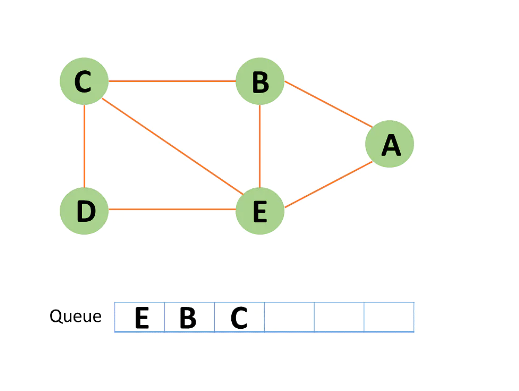


Figure : 08

**Step 7:**

Node **D** is an unvisited neighboring node from **C**. You enqueue it after marking it as visited.

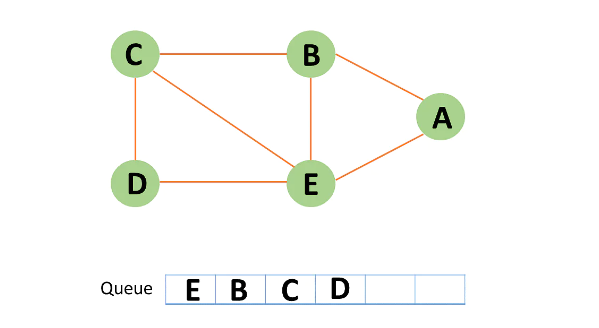


Figure : 09

**Step 8:**

If all of **D**’s adjacent nodes have already been visited, remove **D** from the queue.

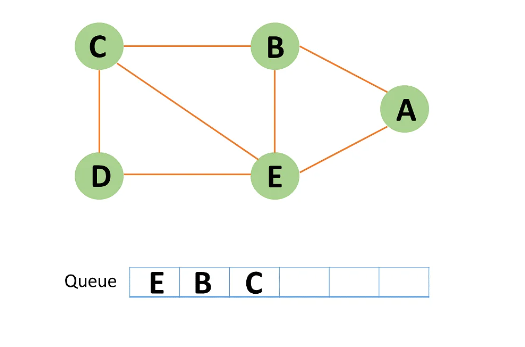


Figure : 10

**Step 9:**

Similarly, all nodes near **E, B**, and **C** nodes have already been visited; therefore, you must remove them from the queue.

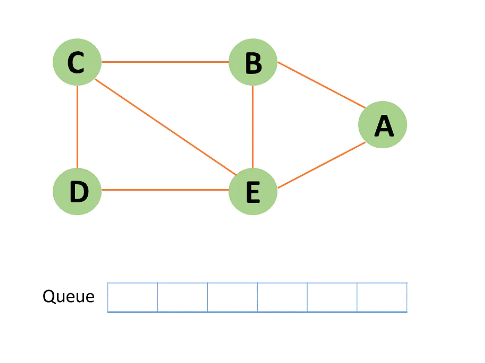


Figure : 11

**Step 10:** Because the queue is now empty, the BFS traversal has ended.

# What is a Depth-First Search Algorithm?

The depth-first search algorithm, often known as DFS, traverses or explores data structures like trees and graphs. The algorithm examines each branch as far as it can go after starting at the root node , before it backtracking. (In the case of a graph, you can choose any random node to serve as the root node)

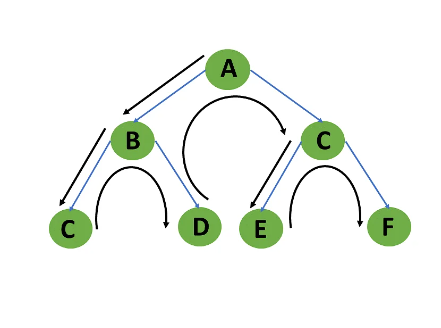


Figure : 12

The Depth First Search (DFS) method explores a network in a deathward motion and uses a stack data structure to keep track of remembering to acquire the next vertex to start a search when a dead-end occurs in any iteration.

For a better understanding, look at a depth-first search method example after reading the definition of the DFS algorithm.

# Example of Depth-First Search Algorithm

The result of a DFS traversal of a graph is a spanning tree. A spanning tree is a graph that is devoid of loops. You must use a stack data structure with a maximum size equal to the total number of vertices in the graph in order to implement DFS traversal.

You must go through the following steps in order to implement DFS traversal.

* **Step 1:**

Create a stack with the total number of vertices in the graph as the size.

* **Step 2:**

Choose any vertex as the traversal’s beginning point. Push a visit to that vertex and add it to the stack.

* **Step 3**

Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

* **Step 4:**

Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

* **Step 5 :**

If there are no new vertices to visit, go back and pop one from the stack using backtracking.

* **Step 6 :**

Continue using steps 3, 4, and 5 until the stack is empty.

* **Step 7:**

When the stack is entirely unoccupied, create the final spanning tree by deleting the graph’s unused edges.

**Consider the graph below as an example of how to apply the DFS algorithm.**

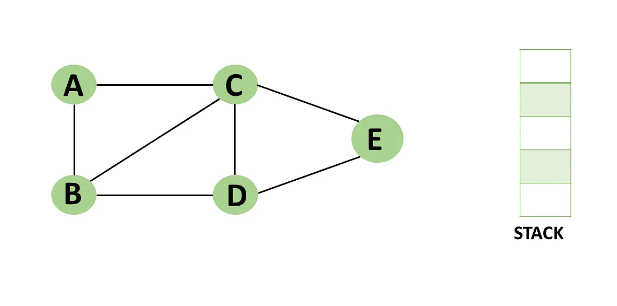


Figure : 13

**Step 1:** Mark vertex **A** as a visited source node by selecting it as a source node.

* You should push vertex **A** to the top of the stack.

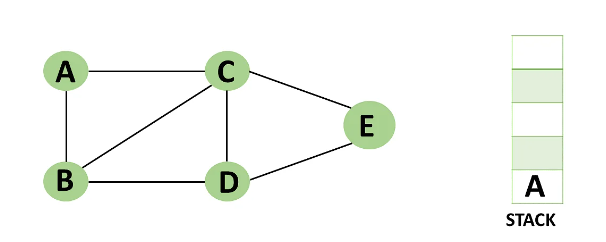


Figure : 14

**Step 2:** Any nearby unvisited vertex of vertex **A**, say **B**, should be visited.

* You should push vertex **B** to the top of the stack.

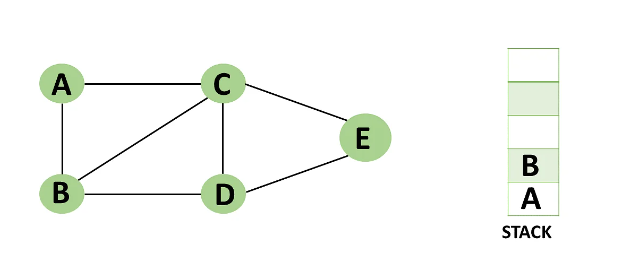


Figure : 15

**Step 3:** From vertex **C** and **D**, visit any adjacent unvisited vertices of vertex **B**. Imagine you have chosen vertex **C**, and you want to make **C** a visited vertex.

* Vertex **C** is pushed to the top of the stack.

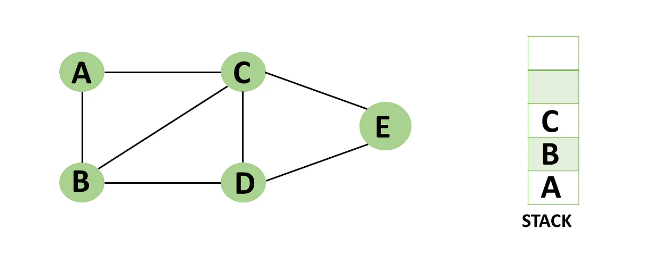


Figure : 16

**Step 4:** You can visit any nearby unvisited vertices of vertex **C**, you need to select vertex **D** and designate it as a visited vertex.

* Vertex **D** is pushed to the top of the stack.

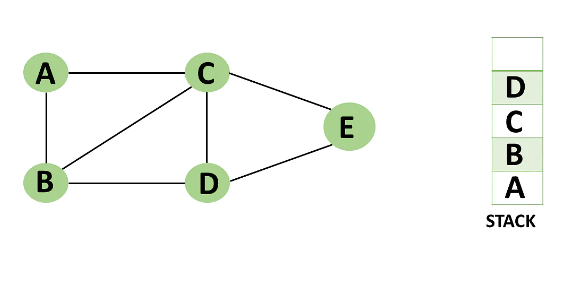


Figure : 17

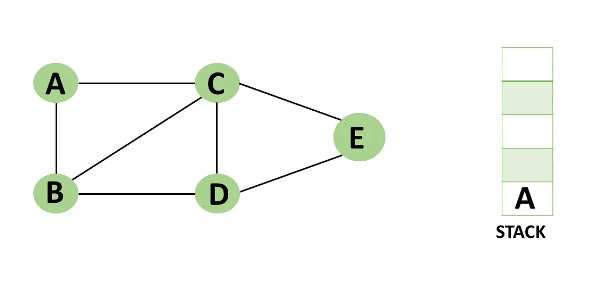


Figure : 18

**Step 5:** Vertex **E** is the lone unvisited adjacent vertex of vertex **D**, thus marking it as visited.

* Vertex **E** should be pushed to the top of the stack.

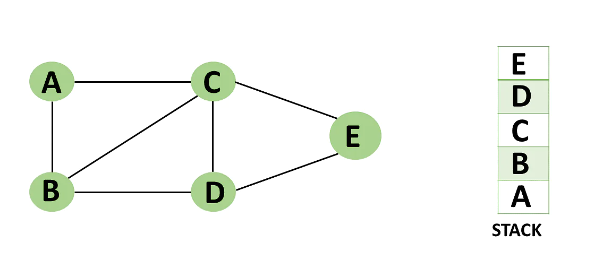


Figure : 19

**Step 6:** Vertex **E**’s nearby vertices, namely vertex **C** and **D** have been visited, pop vertex E from the stack.

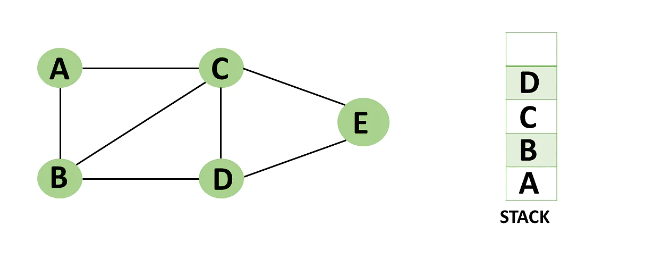


Figure : 20

S**tep 7:** Now that all of vertex **D**’s nearby vertices, namely vertex **B** and **C**, have been visited, pop vertex **D** from the stack.

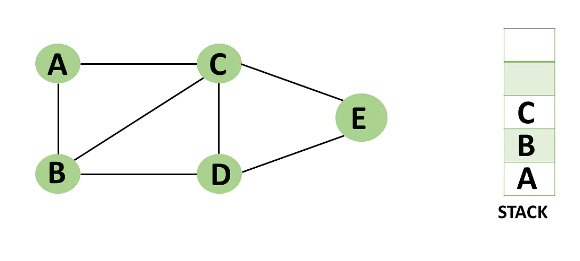


Figure : 21

**Step 8:** Similarly, vertex **C**’s adjacent vertices have already been visited; therefore, pop it from the stack.

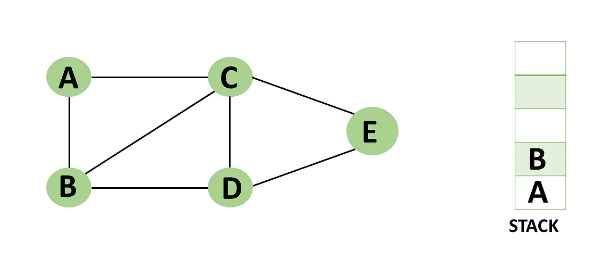


Figure : 22

**Step 9:** There is no more unvisited adjacent vertex of **B**, thus pop it from the stack.

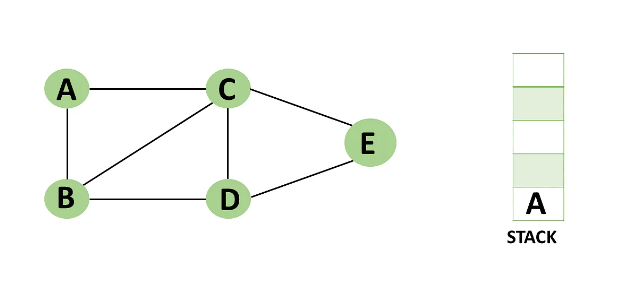


Figure : 23

**Step 10:** All of the nearby vertices of Vertex **A, B**, and **C**, have already been visited, so pop vertex **A** from the stack as well.

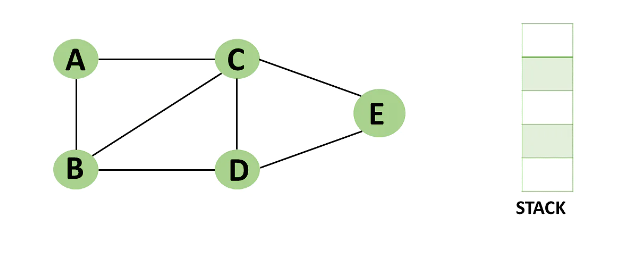


Figure : 24

# Pseudocode of Depth-First Search Algorithm

## Pseudocode of recursive depth-First search algorithm

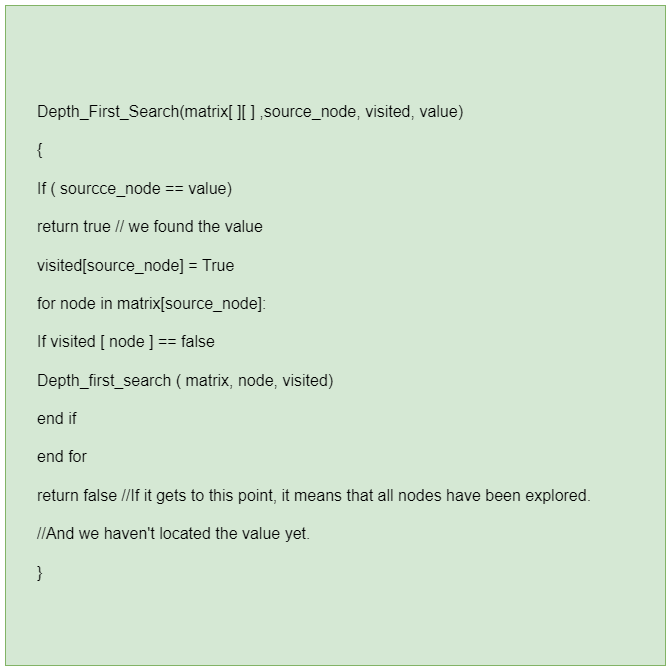


Figure : 25 Displays the Pseudocode of recursive depth-First search algorithm

## 2. Pseudocode of iterative depth-first search algorithm.

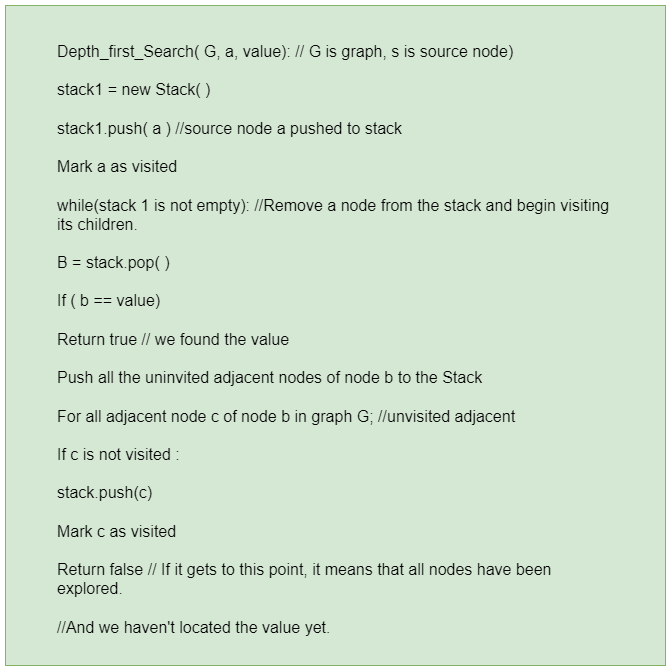


Figure : 26 Displays the Pseudocode of iterative depth-first search algorithm.

# Difference between BFS and DFS Binary Tree

* The below table shows the difference between BFS and DFS.

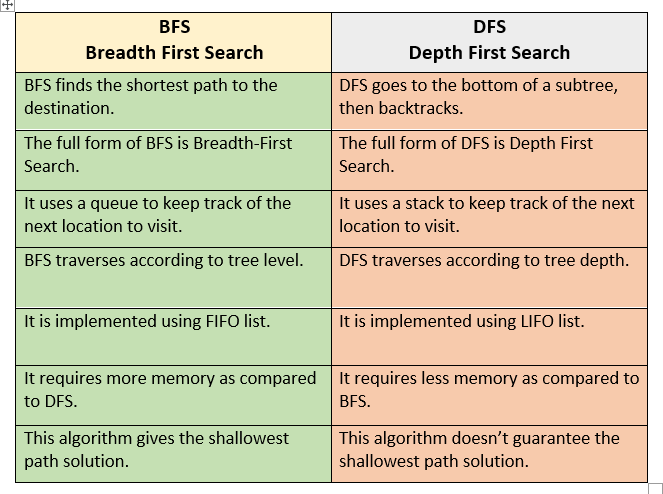


Figure : 27 Displays the difference between BFS and DFS

# When to use BFS and DFS?

The Depth First Search (DFS) should be more efficient in space complexity. Both DFS and BFS should have the same time complexity. With DFS, we only need to use recursion to follow the nodes in a given branch all the down to the end. The greater space needed is because BFS is storing each node in the across the tree. Therefore, DFS could require more space if the tree structure is deep.

**Congratulations..!**, now you have completed the **Breadth First Search (BFS)** and **Depth First Search (DFS)** concepts.

Thanks for reading this article…