Queue Data Structure

* Last Updated : 19 Dec, 2022
* Read
* Discuss(13)
* Courses
* Practice
* Video

[**Data Structure and Algorithms Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=gfg&utm_medium=header+link+click&utm_campaign=dsa+course+tracker&utm_term=dsa+course+promo&utm_content=queue-lp)

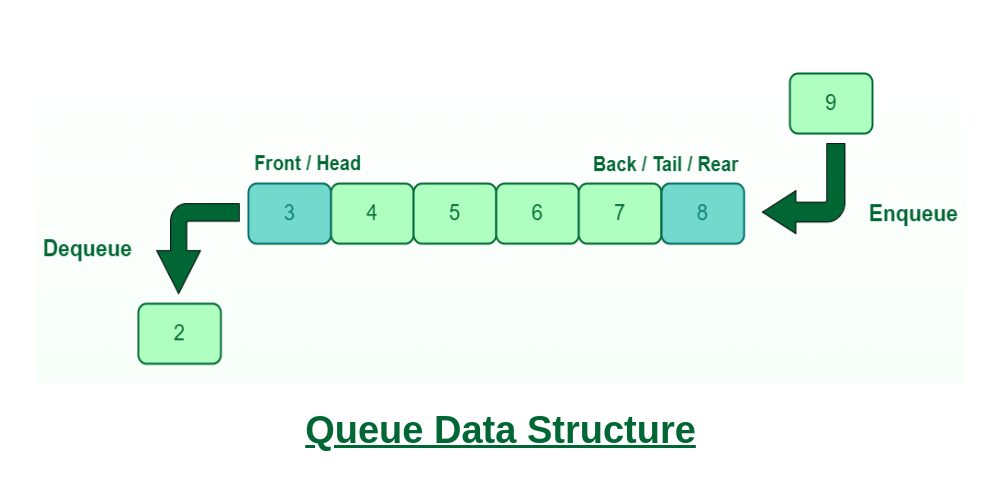
[**Practice Problems on Queue**](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Queue&sortBy=submissions&utm_source=gfg&utm_medium=header+link+click&utm_campaign=practice+tracker&utm_term=practice+promo&utm_content=queue-lp)

[**Recent articles on Queue**](https://www.geeksforgeeks.org/category/queue/?utm_source=gfg&utm_medium=header+link+click&utm_campaign=recent+article+tracker&utm_term=recent+article+tracker&utm_content=queue-lp)

**What is Queue Data Structure?**

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

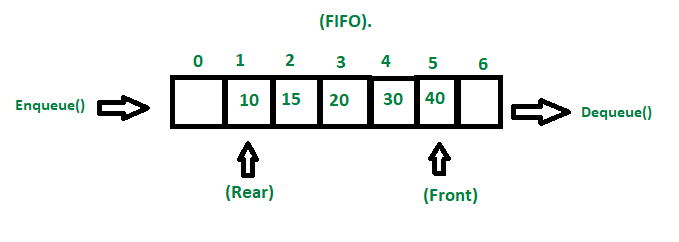
We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.  The element which is first pushed into the order, the operation is first performed on that.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20221213113312/Queue-Data-Structures.png)

*Queue Data Structure*

**FIFO Principle of Queue:**

* A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
* Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20221213111946/fifo-property-in-Queue.png)

*Fifo Property in Queue*

**Characteristics of Queue:**

* Queue can handle multiple data.
* We can access both ends.
* They are fast and flexible.

**Queue Representation:**

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

* **Queue:** the name of the array storing queue elements.
* **Front**: the index where the first element is stored in the array representing the queue.
* **Rear:** the index where the last element is stored in an array representing the queue.

**Topics :**

* [Introduction](https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm#intro)
* [Queue in various Programming Languages](https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm#queueProgLang)
* [Implementation](https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm#implementation)
* [Standard Problems on Queues](https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm#standard)

**Introduction to Queue Data Structure:**

1. [Introduction to Queue – Data Structure and Algorithm Tutorials](https://www.geeksforgeeks.org/introduction-to-queue-data-structure-and-algorithm-tutorials/)
2. [Implementations of Queue Data Structure using Arrays](https://www.geeksforgeeks.org/introduction-and-array-implementation-of-queue/)
3. [Implementations of Queue Data Structure using Linked List](https://www.geeksforgeeks.org/queue-linked-list-implementation/)
4. [Applications, Advantages and Disadvantages of Queue](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/)
5. [Different Types of Queue](https://www.geeksforgeeks.org/different-types-of-queues-and-its-applications/)

**Implementation of Queue in various Programming Languages:**

1. [Queue in C++ Standard Template Library (STL)](https://www.geeksforgeeks.org/queue-cpp-stl/)
2. [Queue Interface In Java](https://www.geeksforgeeks.org/queue-interface-java/)
3. [Queue In Python](https://www.geeksforgeeks.org/queue-in-python/)
4. [Queue In C#](https://www.geeksforgeeks.org/c-sharp-queue-with-examples/)
5. [Queue in Javascript](https://www.geeksforgeeks.org/implementation-queue-javascript/)
6. [Queue in Go Language](https://www.geeksforgeeks.org/queue-in-go-language/)
7. [Queue in Scala](https://www.geeksforgeeks.org/queue-in-scala)

*From <*[*https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm*](https://www.geeksforgeeks.org/queue-data-structure/?ref=ghm)*>*

**Queue in Python**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 06 Jul, 2022
* Read
* Discuss(1)
* Courses
* Practice
* Video

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

* **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
* **Front:** Get the front item from queue – Time Complexity : O(1)
* **Rear:** Get the last item from queue – Time Complexity : O(1)

**Implementation**

There are various ways to implement a queue in Python. This article covers the implementation of queue using data structures and modules from Python library.

Queue in Python can be implemented by the following ways:

* list
* collections.deque
* queue.Queue

**Implementation using list**

List is a Python’s built-in data structure that can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used. However, lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

* Python3

# Python program to

# demonstrate queue implementation

# using list

# Initializing a queue

queue **=** []

# Adding elements to the queue

queue.append('a')

queue.append('b')

queue.append('c')

**print**("Initial queue")

**print**(queue)

# Removing elements from the queue

print("\nElements dequeued from queue")

print(queue.pop(0))

**print**(queue.pop(0))

**print**(queue.pop(0))

**print**("\nQueue after removing elements")

print(queue)

# Uncommenting print(queue.pop(0))

# will raise and IndexError

# as the queue is now empty

**Output:**

Initial queue  
['a', 'b', 'c']

Elements dequeued from queue  
a  
b  
c

Queue after removing elements  
[]

Traceback (most recent call last):  
 File "/home/ef51acf025182ccd69d906e58f17b6de.py", line 25, in   
 print(queue.pop(0))  
IndexError: pop from empty list

**Implementation using collections.deque**

Queue in Python can be implemented using deque class from the collections module. Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity. Instead of enqueue and deque, append() and popleft() functions are used.

* Python3

# Python program to

# demonstrate queue implementation

# using collections.dequeue

**from** collections **import** deque

# Initializing a queue

q **=** deque()

# Adding elements to a queue

q.append('a')

q.append('b')

q.append('c')

**print**("Initial queue")

print(q)

# Removing elements from a queue

**print**("\nElements dequeued from the queue")

**print**(q.popleft())

print(q.popleft())

print(q.popleft())

**print**("\nQueue after removing elements")

print(q)

# Uncommenting q.popleft()

# will raise an IndexError

# as queue is now empty

**Output:**

Initial queue  
deque(['a', 'b', 'c'])

Elements dequeued from the queue  
a  
b  
c

Queue after removing elements  
deque([])

Traceback (most recent call last):  
 File "/home/b2fa8ce438c2a9f82d6c3e5da587490f.py", line 23, in   
 q.popleft()  
IndexError: pop from an empty deque

**Implementation using queue.Queue**

Queue is built-in module of Python which is used to implement a queue. queue.Queue(maxsize) initializes a variable to a maximum size of maxsize. A maxsize of zero ‘0’ means a infinite queue. This Queue follows FIFO rule.

There are various functions available in this module:

* **maxsize**– Number of items allowed in the queue.
* **empty()** – Return True if the queue is empty, False otherwise.
* **full()** – Return True if there are maxsize items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.
* **get()** – Remove and return an item from the queue. If queue is empty, wait until an item is available.
* **get\_nowait()** – Return an item if one is immediately available, else raise QueueEmpty.
* **put(item)** – Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.
* **put\_nowait(item)** – Put an item into the queue without blocking. If no free slot is immediately available, raise QueueFull.
* **qsize()** – Return the number of items in the queue.

* Python3

# Python program to

# demonstrate implementation of

# queue using queue module

**from** queue **import** Queue

# Initializing a queue

q **=** Queue(maxsize **=** 3)

# qsize() give the maxsize

# of the Queue

print(q.qsize())

# Adding of element to queue

q.put('a')

q.put('b')

q.put('c')

# Return Boolean for Full

# Queue

**print**("\nFull: ", q.full())

# Removing element from queue

**print**("\nElements dequeued from the queue")

print(q.get())

print(q.get())

**print**(q.get())

# Return Boolean for Empty

# Queue

print("\nEmpty: ", q.empty())

q.put(1)

**print**("\nEmpty: ", q.empty())

print("Full: ", q.full())

# This would result into Infinite

# Loop as the Queue is empty.

# print(q.get())

**Output:**

0

Full: True

Elements dequeued from the queue  
a  
b  
c

Empty: True

Empty: False  
Full: False

**Easy Questions:**

**Queue – Linked List Implementation**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 13 Dec, 2022
* Read
* Discuss(19)
* Courses
* Practice
* Video

In this article, the Linked List implementation of the [queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) is discussed and implemented.

Recommended Practice

[Implement Queue using Linked List](https://practice.geeksforgeeks.org/problems/implement-queue-using-linked-list/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/implement-queue-using-linked-list/1/)

[](https://practice.geeksforgeeks.org/courses/complete-interview-preparation?utm_source=article&utm_medium=article&utm_campaign=complete-interview-preparation)

**Approach:**To solve the problem follow the below idea:

*we maintain two pointers,****front****, and****rear****. The front**points to the first item of the queue and**rear**points to the last item.*

* ***enQueue():****This operation adds a new node after the rear**and moves the rear**to the next node.*
* ***deQueue():****This operation removes the front node and moves the front**to the next node.*

Follow the below steps to solve the problem:

* Create a class QNode with data members integer data and QNode\* next
* A parameterized constructor that takes an integer x value as a parameter and sets data equal to xand next as NULL
* Create a class Queue with data members QNode frontand rear
* Enqueue Operation with parameter x:
* Initialize QNode\* tempwith data = x
* If the rear is set to NULL then set the front and rear to tempand return(Base Case)
* Else set rear next totemp and then move rear to temp
* Dequeue Operation:
* If the front is set to NULL return(Base Case)
* Initialize QNode tempwith front and set front to its next
* If the front is equal to NULLthen set the rear to NULL
* Delete temp from the memory

Below is the Implementation of the above approach:

* C++
* C
* Java
* Python3
* C#
* Javascript

# Python3 program to demonstrate linked list

# based implementation of queue

# A linked list (LL) node

# to store a queue entry

**class** Node:

**def** \_\_init\_\_(self, data):

        self.data **=** data

        self.next **=** None

# A class to represent a queue

# The queue, front stores the front node

# of LL and rear stores the last node of LL

**class** Queue:

**def** \_\_init\_\_(self):

        self.front **=** self.rear **=** None

**def** isEmpty(self):

**return** self.front **==** None

    # Method to add an item to the queue

**def** EnQueue(self, item):

        temp **=** Node(item)

**if** self.rear **==** None:

            self.front **=** self.rear **=** temp

**return**

        self.rear.next **=** temp

        self.rear **=** temp

    # Method to remove an item from queue

**def** DeQueue(self):

**if** self.isEmpty():

**return**

        temp **=** self.front

        self.front **=** temp.next

**if**(self.front **==** None):

            self.rear **=** None

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    q **=** Queue()

    q.EnQueue(10)

    q.EnQueue(20)

    q.DeQueue()

    q.DeQueue()

    q.EnQueue(30)

    q.EnQueue(40)

    q.EnQueue(50)

    q.DeQueue()

**print**("Queue Front : " **+** str(q.front.data))

    print("Queue Rear : " **+** str(q.rear.data))

**Output**

Queue Front : 40  
Queue Rear : 50

**Time Complexity:**O(1), The time complexity of both operations enqueue() and dequeue() is O(1) as it only changes a few pointers in both operations

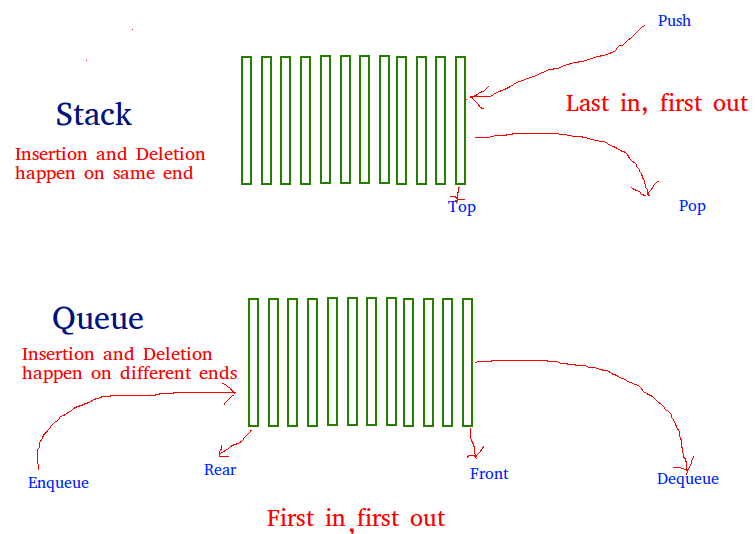
**Auxiliary Space:**O(1), The auxiliary Space of both operations enqueue() and dequeue() is O(1) as constant extra space is required

*From <*[*https://www.geeksforgeeks.org/queue-linked-list-implementation/*](https://www.geeksforgeeks.org/queue-linked-list-implementation/)*>*

**Queue using Stacks**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 14 Dec, 2022
* Read
* Discuss(166)
* Courses
* Practice
* Video

The problem is opposite of [this](https://www.geeksforgeeks.org/implement-stack-using-queue/) post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Recommended Problem

Queue using stack

[Stack](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Stack&sortBy=submissions)

[Queue](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Queue&sortBy=submissions)

+2 more

[Microsoft](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Microsoft&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/queue-using-stack/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 11.4K

**Method 1 (By making enQueue operation costly):** This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

*enQueue(q, x):*

* *While stack1 is not empty, push everything from stack1 to stack2.*
* *Push x to stack1 (assuming size of stacks is unlimited).*
* *Push everything back to stack1.*

*Here time complexity will be O(n)*

*deQueue(q):*

* *If stack1 is empty then error*
* *Pop an item from stack1 and return it*

*Here time complexity will be O(1)*

Below is the implementation of the above approach:

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to implement Queue using

# two stacks with costly enQueue()

**class** Queue:

**def** \_\_init\_\_(self):

        self.s1 **=** []

        self.s2 **=** []

**def** enQueue(self, x):

        # Move all elements from s1 to s2

**while** len(self.s1) !**=** 0:

            self.s2.append(self.s1[**-**1])

            self.s1.pop()

        # Push item into self.s1

        self.s1.append(x)

        # Push everything back to s1

**while** len(self.s2) !**=** 0:

            self.s1.append(self.s2[**-**1])

            self.s2.pop()

    # Dequeue an item from the queue

**def** deQueue(self):

            # if first stack is empty

**if** len(self.s1) **==** 0:

**print**("Q is Empty")

        # Return top of self.s1

        x **=** self.s1[**-**1]

        self.s1.pop()

**return** x

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    q **=** Queue()

    q.enQueue(1)

    q.enQueue(2)

    q.enQueue(3)

**print**(q.deQueue())

**print**(q.deQueue())

**print**(q.deQueue())

# This code is contributed by PranchalK

**Output:**

1  
2  
3

**Complexity Analysis:**

* **Time Complexity:**
* **Push operation:** O(N).   
  In the worst case we have empty whole of stack 1 into stack 2.
* **Pop operation:** O(1).   
  Same as pop operation in stack.
* **Auxiliary Space:** O(N).   
  Use of stack for storing values.

**Method 2 (By making deQueue operation costly):**In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)  
 1) Push x to stack1 (assuming size of stacks is unlimited).  
Here time complexity will be O(1)

deQueue(q)  
 1) If both stacks are empty then error.  
 2) If stack2 is empty  
 While stack1 is not empty, push everything from stack1 to stack2.  
 3) Pop the element from stack2 and return it.  
Here time complexity will be O(n)

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty. So, the amortized complexity of the dequeue operation becomes

Implementation of method 2:

* C++
* C
* Java
* Python3
* C#
* Javascript

# Python3 program to implement Queue using

# two stacks with costly deQueue()

**class** Queue:

**def** \_\_init\_\_(self):

        self.s1 **=** []

        self.s2 **=** []

    # EnQueue item to the queue

**def** enQueue(self, x):

        self.s1.append(x)

    # DeQueue item from the queue

**def** deQueue(self):

        # if both the stacks are empty

**if** len(self.s1) **==** 0 **and** len(self.s2) **==** 0:

**print**("Q is Empty")

**return**

        # if s2 is empty and s1 has elements

**elif** len(self.s2) **==** 0 **and** len(self.s1) > 0:

**while** len(self.s1):

                temp **=** self.s1.pop()

                self.s2.append(temp)

**return** self.s2.pop()

**else**:

**return** self.s2.pop()

    # Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    q **=** Queue()

    q.enQueue(1)

    q.enQueue(2)

    q.enQueue(3)

**print**(q.deQueue())

**print**(q.deQueue())

**print**(q.deQueue())

# This code is contributed by Pratyush Kumar

**Output:**

1 2 3

**Complexity Analysis:**

* **Time Complexity:**
* **Push operation:** O(1).   
  Same as pop operation in stack.
* **Pop operation:** O(N) in general and O(1) amortized time complexity.  
  In the worst case we have to empty the whole of stack 1 into stack 2 so its O(N). Amortized time is the way to express the time complexity when an algorithm has the very bad time complexity only once in a while besides the time complexity that happens most of time. So its O(1) amortized time complexity, since we have to empty whole of stack 1 only when stack 2 is empty, rest of the times the pop operation takes O(1) time.
* **Auxiliary Space:** O(N).   
  Use of stack for storing values.

**Queue can also be implemented using one user stack and one Function Call Stack.**Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

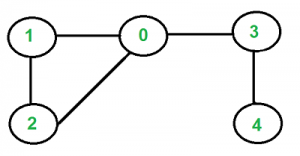
*enQueue(x)*  
 1) Push *x* to *stack1*.

*deQueue:*  
 1) If *stack1* is empty then error.  
 2) If *stack1* has only one element then return it.  
 3) Recursively pop everything from the stack1, store the popped item   
 in a variable *res*, push the *res* back to stack1 and return *res*

**Detect cycle in an undirected graph using BFS**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 15 Dec, 2022
* Read
* Discuss(9)
* Courses
* Practice
* Video

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle of 1-0-2-1.



[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/detect-cycle-in-an-undirected-graph/1)

We have discussed [cycle detection for](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/)the [directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/). We have also discussed a [union-find algorithm for cycle detection in undirected graphs.](https://www.geeksforgeeks.org/union-find/). The time complexity of the union-find algorithm is O(ELogV). Like directed graphs, we can use [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to detect a cycle in an undirected graph in O(V+E) time. We have discussed [DFS based solution for cycle detection in](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/)an [undirected graph](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/).

In this article, the [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) based solution is discussed. We do a BFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not a parent of v, then there is a cycle in the graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.

We use a parent array to keep track of the parent vertex for a vertex so that we do not consider the visited parent as a cycle.

**Implementation:**

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to detect cycle in

# an undirected graph using BFS.

**from** collections **import** deque

**def** addEdge(adj: list, u, v):

    adj[u].append(v)

    adj[v].append(u)

**def** isCyclicConnected(adj: list, s, V,

                      visited: list):

    # Set parent vertex for every vertex as -1.

    parent **=** [**-**1] **\*** V

    # Create a queue for BFS

    q **=** deque()

    # Mark the current node as

    # visited and enqueue it

    visited[s] **=** True

    q.append(s)

**while** q !**=** []:

        # Dequeue a vertex from queue and print it

        u **=** q.pop()

        # Get all adjacent vertices of the dequeued

        # vertex u. If a adjacent has not been visited,

        # then mark it visited and enqueue it. We also

        # mark parent so that parent is not considered

        # for cycle.

**for** v **in** adj[u]:

**if not** visited[v]:

                visited[v] **=** True

                q.append(v)

                parent[v] **=** u

**elif** parent[u] !**=** v:

**return** True

**return** False

**def** isCyclicDisconnected(adj: list, V):

    # Mark all the vertices as not visited

    visited **=** [False] **\*** V

**for** i **in** range(V):

**if not** visited[i] **and** \

               isCyclicConnected(adj, i, V, visited):

**return** True

**return** False

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    V **=** 4

    adj **=** [[] **for** i **in** range(V)]

    addEdge(adj, 0, 1)

    addEdge(adj, 1, 2)

    addEdge(adj, 2, 0)

    addEdge(adj, 2, 3)

**if** isCyclicDisconnected(adj, V):

        print("Yes")

**else**:

        print("No")

# This code is contributed by

# sanjeev2552

**Output**

Yes

**Time Complexity:** The program does a simple BFS Traversal of the graph and the graph is represented using an adjacency list. So the time complexity is**O(V+E)**

**Space Complexity: O(V)**for visited vector.

**Breadth First Search or BFS for a Graph**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 21 Dec, 2022
* Read
* Discuss(168)
* Courses
* Practice
* Video

*The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree’s root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.*

[Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to the Breadth-First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)).

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

* Visited and
* Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a [**queue data structure**](https://www.geeksforgeeks.org/queue-data-structure/) for traversal.

**Example:**

*In the following graph, we start traversal from vertex 2.*



*When we come to****vertex 0****, we look for all adjacent vertices of it.*

* *2 is also an adjacent vertex of 0.*
* *If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.*

*There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :*

*2, 3, 0, 1*

*2, 0, 3, 1*

Recommended Problem

BFS of graph

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

[Graph](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Graph&sortBy=submissions)

+2 more

[Adobe](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Adobe&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+5 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 2.2L

**Implementation of BFS traversal on Graph:**

**Pseudocode:**

*Breadth\_First\_Serach( Graph, X ) // Here, Graph is the graph that we already have and X is the source node*

*Let Q be the queue*

*Q.enqueue( X ) // Inserting source node X into the queue*

*Mark X node as visited.*

*While ( Q is not empty )*

*Y = Q.dequeue( ) // Removing the front node from the queue*

*Process all the neighbors of Y, For all the neighbors Z of Y*

*If Z is not visited, Q. enqueue( Z ) // Stores Z in Q*

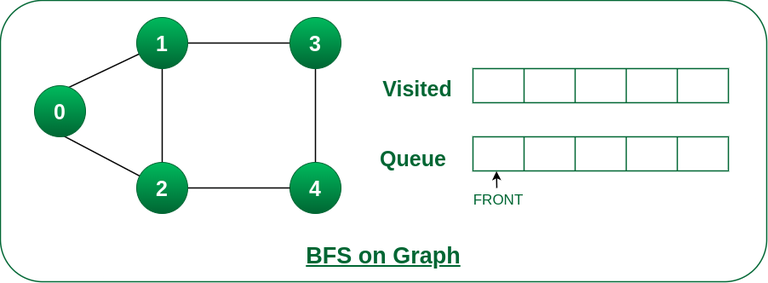
*Mark Z as visited*

Follow the below method to implement BFS traversal.

* Declare a queue and insert the starting vertex.
* Initialize a **visited** array and mark the starting vertex as visited.
* Follow the below process till the queue becomes empty:
* Remove the first vertex of the queue.
* Mark that vertex as visited.
* Insert all the unvisited neighbors of the vertex into the queue.

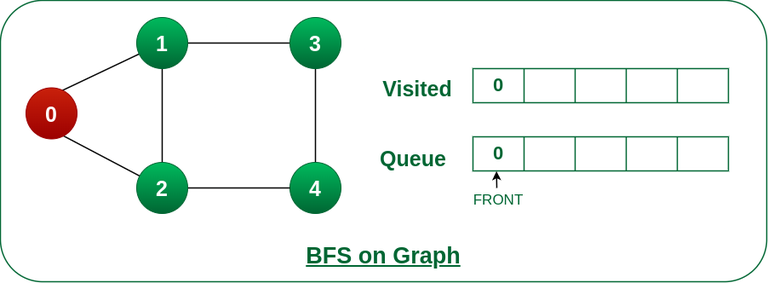
**Illustration:**

***Step1:****Initially queue and visited arrays are empty.*



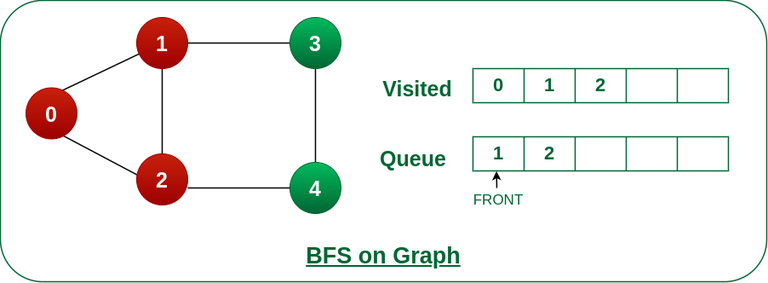
*Queue and visited arrays are empty initially.*

***Step2:****Push node 0 into queue and mark it visited.*



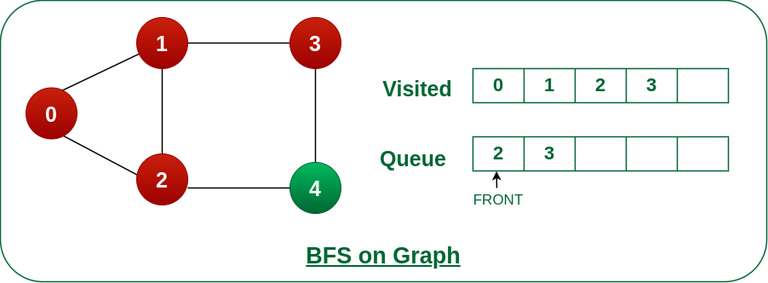
*Push node 0 into queue and mark it visited.*

***Step 3:****Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



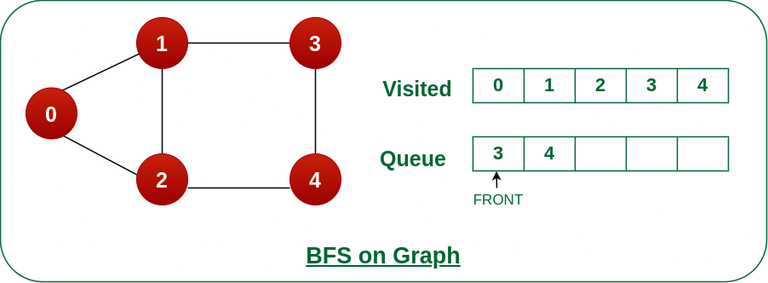
*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

***Step 4:****Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

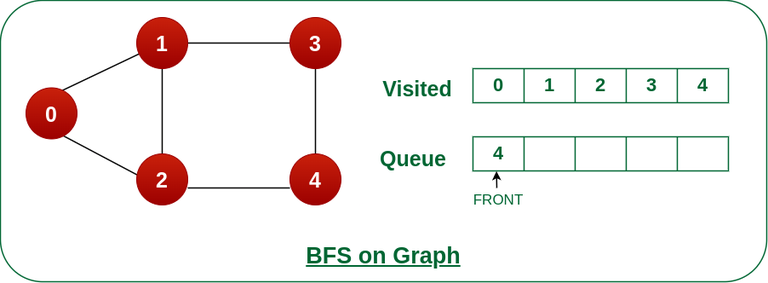
***Step 5:****Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Step 6:****Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

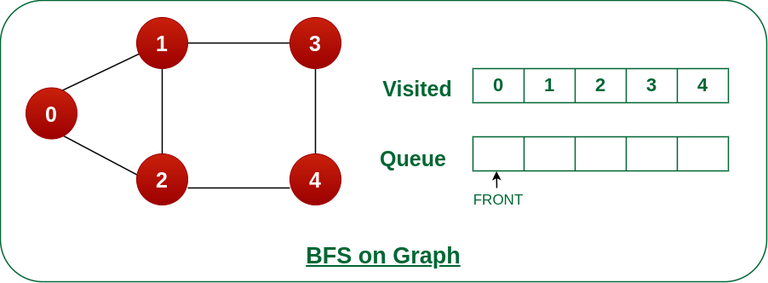
*As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*



*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Steps 7:****Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.*

*As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*



*Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.*

*Now, Queue becomes empty, So, terminate these process of iteration.*

The implementation uses an [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) stores lists of adjacent nodes and the queue of nodes needed for BFS traversal.

* C++
* Java
* Python3
* C#
* Javascript

# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

**from** collections **import** defaultdict

# This class represents a directed graph

# using adjacency list representation

**class** Graph:

    # Constructor

**def** \_\_init\_\_(self):

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

    # function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # Function to print a BFS of graph

**def** BFS(self, s):

        # Mark all the vertices as not visited

        visited **=** [False] **\*** (max(self.graph) **+** 1)

        # Create a queue for BFS

        queue **=** []

        # Mark the source node as

        # visited and enqueue it

        queue.append(s)

        visited[s] **=** True

**while** queue:

            # Dequeue a vertex from

            # queue and print it

            s **=** queue.pop(0)

            print(s, end**=**" ")

            # Get all adjacent vertices of the

            # dequeued vertex s. If a adjacent

            # has not been visited, then mark it

            # visited and enqueue it

**for** i **in** self.graph[s]:

**if** visited[i] **==** False:

                    queue.append(i)

                    visited[i] **=** True

# Driver code

# Create a graph given in

# the above diagram

g **=** Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

**print**("Following is Breadth First Traversal"

      " (starting from vertex 2)")

g.BFS(2)

# This code is contributed by Neelam Yadav

**Output**

Following is Breadth First Traversal (starting from vertex 2)   
2 0 3 1

**Time Complexity:**O(V+E), where V is the number of nodes and E is the number of edges.

**Auxiliary Space:**O(V)

**BFS for Disconnected Graph:**

Note that the above code traverses only the vertices reachable from a given source vertex. In every situation, all the vertices may not be reachable from a given vertex (i.e. for a disconnected graph).

*To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the*[*DFS modified version*](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)*).*

Below is the implementation for BFS traversal for the entire graph (valid for directed as well as undirected graphs) with possible multiple disconnected components:

* C++
* Java
* Python3
* C#
* Javascript

'''

Generic Function for BFS traversal of a Graph

 (valid for directed as well as undirected graphs

 which can have multiple disconnected components)

-- Inputs --

-> V - represents number of vertices in the Graph

-> adj[] - represents adjacency list for the Graph

-- Output --

-> bfs\_traversal - a vector containing bfs traversal

for entire graph

'''

**def** bfsOfGraph(V, adj):

    bfs\_traversal **=** []

    vis **=** [False]**\***V

**for** i **in** range(V):

        # To check if already visited

**if** (vis[i] **==** False):

            q **=** []

            vis[i] **=** True

            q.append(i)

            # BFS starting from ith node

**while** (len(q) > 0):

                g\_node **=** q.pop(0)

                bfs\_traversal.append(g\_node)

**for** it **in** adj[g\_node]:

**if** (vis[it] **==** False):

                        vis[it] **=** True

                        q.append(it)

**return** bfs\_traversal

  # This code is contributed by Abhijeet Kumar(abhijeet19403)

**Problems related to BFS:**

|  |  |  |
| --- | --- | --- |
| S.no | Problems | Practice |
| 1. | [Find the level of a given node in an Undirected Graph](https://www.geeksforgeeks.org/find-the-level-of-given-node-in-an-undirected-graph/) | [Link](https://ide.geeksforgeeks.org/) |
| 2. | [Minimize maximum adjacent difference in a path from top-left to bottom-right](https://www.geeksforgeeks.org/minimize-maximum-adjacent-difference-in-a-path-from-top-left-to-bottom-right/) | [Link](https://ide.geeksforgeeks.org/) |
| 3. | [Minimum jump to the same value or adjacent to reach the end of an Array](https://www.geeksforgeeks.org/minimum-jumps-to-same-value-or-adjacent-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 4. | [Maximum coin in minimum time by skipping K obstacles along the path in Matrix](https://www.geeksforgeeks.org/maximum-coin-in-minimum-time-by-skipping-k-obstacles-along-path-in-matrix/) | [Link](https://ide.geeksforgeeks.org/) |
| 5. | [Check if all nodes of the Undirected Graph can be visited from the given Node](https://www.geeksforgeeks.org/check-if-all-nodes-of-undirected-graph-can-be-visited-from-given-node/) | [Link](https://ide.geeksforgeeks.org/) |
| 6. | [Minimum time to visit all nodes of a given Graph at least once](https://www.geeksforgeeks.org/minimum-time-to-visit-all-nodes-of-given-graph-at-least-once/) | [Link](https://ide.geeksforgeeks.org/) |
| 7. | [Minimize moves to the next greater element to reach the end of the Array](https://www.geeksforgeeks.org/minimize-moves-to-next-greater-element-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 8. | [Shortest path by removing K walls](https://www.geeksforgeeks.org/shortest-path-by-removing-k-walls/) | [Link](https://ide.geeksforgeeks.org/) |
| 9. | [Minimum time required to infect all the nodes of the Binary tree](https://www.geeksforgeeks.org/minimum-time-required-to-infect-all-the-nodes-of-binary-tree/) | [Link](https://ide.geeksforgeeks.org/) |
| 10. | [Check if destination of given Matrix is reachable with required values of cells](https://www.geeksforgeeks.org/check-if-destination-of-given-matrix-is-reachable-with-required-values-of-cells/) | Link |

**Applications of BFS:**

* **Shortest Path and Minimum Spanning Tree for unweighted graph:**In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
* **Peer-to-Peer Networks:** In Peer-to-Peer Networks like [BitTorrent](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.
* **Crawlers in Search Engines:** Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.
* **Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.
* **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
* **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
* **In Garbage Collection:** Breadth First Search is used in copying garbage collection using [Cheney’s algorithm](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Refer [this](https://lambda.uta.edu/cse5317/notes/node48.html)and for details. Breadth First Search is preferred over Depth First Search because of the better locality of reference:
* [**Cycle detection in the undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use [BFS to detect cycle in a directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/) also,
* [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)**:**In the Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst-case time complexity to O(VE2).
* [**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/)**:** We can either use Breadth First or Depth First Traversal.
* **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
* **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

*From <*[*https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/*](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)*>*

**Breadth First Search or BFS for a Graph**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 21 Dec, 2022
* Read
* Discuss(168)
* Courses
* Practice
* Video

*The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree’s root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.*

[Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to the Breadth-First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)).

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

* Visited and
* Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a [**queue data structure**](https://www.geeksforgeeks.org/queue-data-structure/) for traversal.

**Example:**

*In the following graph, we start traversal from vertex 2.*



*When we come to****vertex 0****, we look for all adjacent vertices of it.*

* *2 is also an adjacent vertex of 0.*
* *If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.*

*There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :*

*2, 3, 0, 1*

*2, 0, 3, 1*

Recommended Problem

BFS of graph

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

[Graph](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Graph&sortBy=submissions)

+2 more

[Adobe](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Adobe&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+5 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 2.2L

**Implementation of BFS traversal on Graph:**

**Pseudocode:**

*Breadth\_First\_Serach( Graph, X ) // Here, Graph is the graph that we already have and X is the source node*

*Let Q be the queue*

*Q.enqueue( X ) // Inserting source node X into the queue*

*Mark X node as visited.*

*While ( Q is not empty )*

*Y = Q.dequeue( ) // Removing the front node from the queue*

*Process all the neighbors of Y, For all the neighbors Z of Y*

*If Z is not visited, Q. enqueue( Z ) // Stores Z in Q*

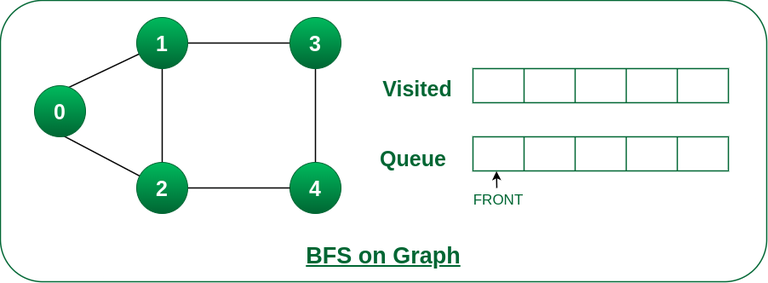
*Mark Z as visited*

Follow the below method to implement BFS traversal.

* Declare a queue and insert the starting vertex.
* Initialize a **visited** array and mark the starting vertex as visited.
* Follow the below process till the queue becomes empty:
* Remove the first vertex of the queue.
* Mark that vertex as visited.
* Insert all the unvisited neighbors of the vertex into the queue.

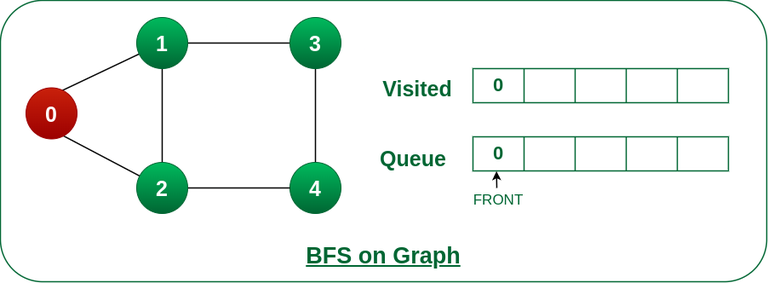
**Illustration:**

***Step1:****Initially queue and visited arrays are empty.*



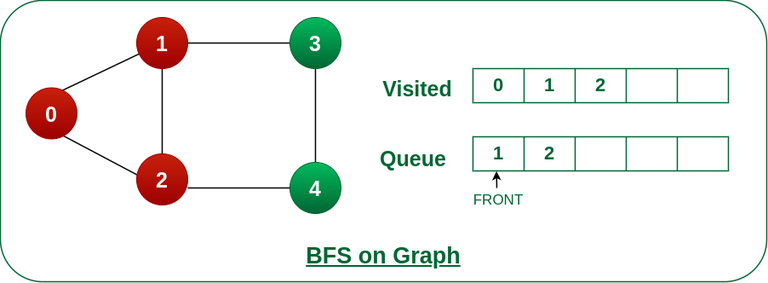
*Queue and visited arrays are empty initially.*

***Step2:****Push node 0 into queue and mark it visited.*



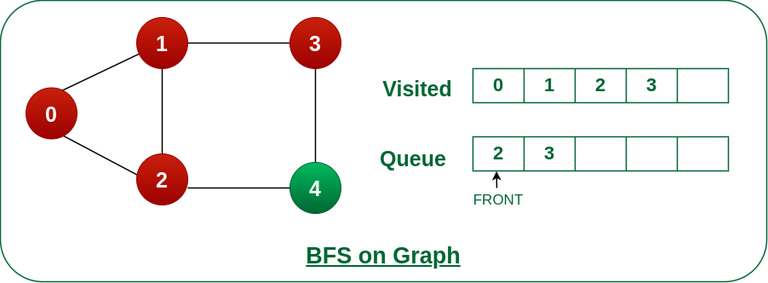
*Push node 0 into queue and mark it visited.*

***Step 3:****Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



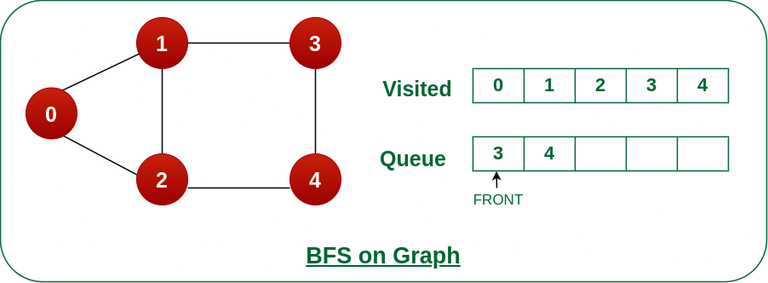
*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

***Step 4:****Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

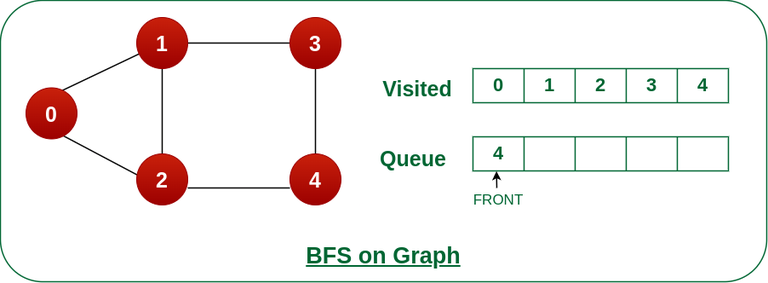
***Step 5:****Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Step 6:****Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

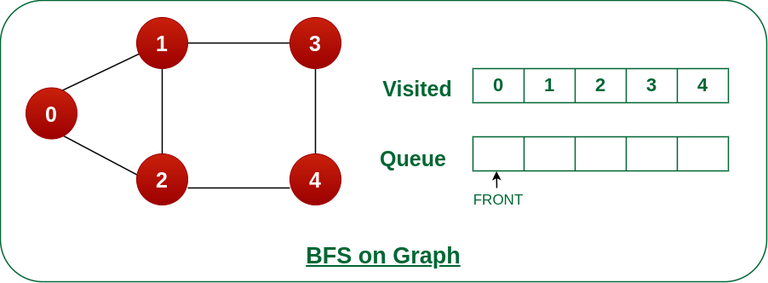
*As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*



*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Steps 7:****Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.*

*As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*



*Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.*

*Now, Queue becomes empty, So, terminate these process of iteration.*

The implementation uses an [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) stores lists of adjacent nodes and the queue of nodes needed for BFS traversal.

* C++
* Java
* Python3
* C#
* Javascript

# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

**from** collections **import** defaultdict

# This class represents a directed graph

# using adjacency list representation

**class** Graph:

    # Constructor

**def** \_\_init\_\_(self):

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

    # function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # Function to print a BFS of graph

**def** BFS(self, s):

        # Mark all the vertices as not visited

        visited **=** [False] **\*** (max(self.graph) **+** 1)

        # Create a queue for BFS

        queue **=** []

        # Mark the source node as

        # visited and enqueue it

        queue.append(s)

        visited[s] **=** True

**while** queue:

            # Dequeue a vertex from

            # queue and print it

            s **=** queue.pop(0)

            print(s, end**=**" ")

            # Get all adjacent vertices of the

            # dequeued vertex s. If a adjacent

            # has not been visited, then mark it

            # visited and enqueue it

**for** i **in** self.graph[s]:

**if** visited[i] **==** False:

                    queue.append(i)

                    visited[i] **=** True

# Driver code

# Create a graph given in

# the above diagram

g **=** Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

**print**("Following is Breadth First Traversal"

      " (starting from vertex 2)")

g.BFS(2)

# This code is contributed by Neelam Yadav

**Output**

Following is Breadth First Traversal (starting from vertex 2)   
2 0 3 1

**Time Complexity:**O(V+E), where V is the number of nodes and E is the number of edges.

**Auxiliary Space:**O(V)

**BFS for Disconnected Graph:**

Note that the above code traverses only the vertices reachable from a given source vertex. In every situation, all the vertices may not be reachable from a given vertex (i.e. for a disconnected graph).

*To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the*[*DFS modified version*](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)*).*

Below is the implementation for BFS traversal for the entire graph (valid for directed as well as undirected graphs) with possible multiple disconnected components:

* C++
* Java
* Python3
* C#
* Javascript

'''

Generic Function for BFS traversal of a Graph

 (valid for directed as well as undirected graphs

 which can have multiple disconnected components)

-- Inputs --

-> V - represents number of vertices in the Graph

-> adj[] - represents adjacency list for the Graph

-- Output --

-> bfs\_traversal - a vector containing bfs traversal

for entire graph

'''

**def** bfsOfGraph(V, adj):

    bfs\_traversal **=** []

    vis **=** [False]**\***V

**for** i **in** range(V):

        # To check if already visited

**if** (vis[i] **==** False):

            q **=** []

            vis[i] **=** True

            q.append(i)

            # BFS starting from ith node

**while** (len(q) > 0):

                g\_node **=** q.pop(0)

                bfs\_traversal.append(g\_node)

**for** it **in** adj[g\_node]:

**if** (vis[it] **==** False):

                        vis[it] **=** True

                        q.append(it)

**return** bfs\_traversal

  # This code is contributed by Abhijeet Kumar(abhijeet19403)

**Problems related to BFS:**

|  |  |  |
| --- | --- | --- |
| S.no | Problems | Practice |
| 1. | [Find the level of a given node in an Undirected Graph](https://www.geeksforgeeks.org/find-the-level-of-given-node-in-an-undirected-graph/) | [Link](https://ide.geeksforgeeks.org/) |
| 2. | [Minimize maximum adjacent difference in a path from top-left to bottom-right](https://www.geeksforgeeks.org/minimize-maximum-adjacent-difference-in-a-path-from-top-left-to-bottom-right/) | [Link](https://ide.geeksforgeeks.org/) |
| 3. | [Minimum jump to the same value or adjacent to reach the end of an Array](https://www.geeksforgeeks.org/minimum-jumps-to-same-value-or-adjacent-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 4. | [Maximum coin in minimum time by skipping K obstacles along the path in Matrix](https://www.geeksforgeeks.org/maximum-coin-in-minimum-time-by-skipping-k-obstacles-along-path-in-matrix/) | [Link](https://ide.geeksforgeeks.org/) |
| 5. | [Check if all nodes of the Undirected Graph can be visited from the given Node](https://www.geeksforgeeks.org/check-if-all-nodes-of-undirected-graph-can-be-visited-from-given-node/) | [Link](https://ide.geeksforgeeks.org/) |
| 6. | [Minimum time to visit all nodes of a given Graph at least once](https://www.geeksforgeeks.org/minimum-time-to-visit-all-nodes-of-given-graph-at-least-once/) | [Link](https://ide.geeksforgeeks.org/) |
| 7. | [Minimize moves to the next greater element to reach the end of the Array](https://www.geeksforgeeks.org/minimize-moves-to-next-greater-element-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 8. | [Shortest path by removing K walls](https://www.geeksforgeeks.org/shortest-path-by-removing-k-walls/) | [Link](https://ide.geeksforgeeks.org/) |
| 9. | [Minimum time required to infect all the nodes of the Binary tree](https://www.geeksforgeeks.org/minimum-time-required-to-infect-all-the-nodes-of-binary-tree/) | [Link](https://ide.geeksforgeeks.org/) |
| 10. | [Check if destination of given Matrix is reachable with required values of cells](https://www.geeksforgeeks.org/check-if-destination-of-given-matrix-is-reachable-with-required-values-of-cells/) | Link |

**Applications of BFS:**

* **Shortest Path and Minimum Spanning Tree for unweighted graph:**In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
* **Peer-to-Peer Networks:** In Peer-to-Peer Networks like [BitTorrent](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.
* **Crawlers in Search Engines:** Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.
* **Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.
* **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
* **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
* **In Garbage Collection:** Breadth First Search is used in copying garbage collection using [Cheney’s algorithm](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Refer [this](https://lambda.uta.edu/cse5317/notes/node48.html)and for details. Breadth First Search is preferred over Depth First Search because of the better locality of reference:
* [**Cycle detection in the undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use [BFS to detect cycle in a directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/) also,
* [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)**:**In the Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst-case time complexity to O(VE2).
* [**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/)**:** We can either use Breadth First or Depth First Traversal.
* **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
* **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

**Vertical order traversal of Binary Tree using Map**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 06 Dec, 2022
* Read
* Discuss(210)
* Courses
* Practice
* Video

Given a binary tree, print it vertically. The following examples illustrate the vertical order traversal.

**Examples:**

***Input:****1*

*/    \*

*2      3*

*/ \   /   \*

*4   5  6   7*

*/  \*

*8    9*

***Output:***

*4*

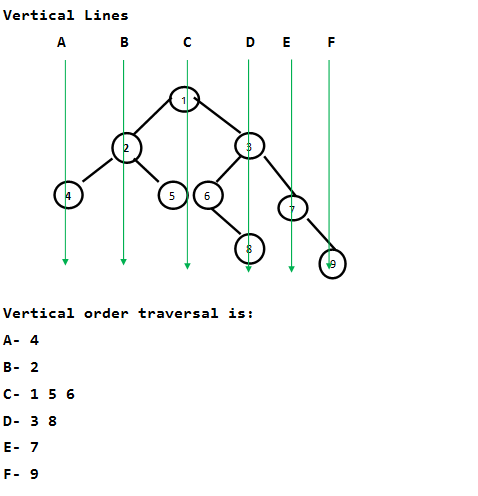
*2*

*1 5 6*

*3 8*

*7*

*9*



[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/print-a-binary-tree-in-vertical-order/1)

**Vertical order traversal of the binary tree using**[hashing](https://www.geeksforgeeks.org/hashing-data-structure/)**:**

To solve the problem follow the below idea:

*We need to check the Horizontal Distances from the root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on the same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance.*

Follow the below steps to solve the problem:

* Do a preorder traversal of the given Binary Tree.
* While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for the root.
* For the left subtree, we pass the Horizontal Distance as the Horizontal distance of the root minus 1.
* For the right subtree, we pass the Horizontal Distance as the Horizontal Distance of the root plus 1.
* For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in a map.

Below is the implementation of the above approach, thanks to Chirag for providing the below C++ implementation:

* C++
* Java
* Python3
* C#

# Python3 program for printing vertical order of a given

# binary tree

# A binary tree node

**class** Node:

    # Constructor to create a new node

**def** \_\_init\_\_(self, key):

        self.key **=** key

        self.left **=** None

        self.right **=** None

# Utility function to store vertical order in map 'm'

# 'hd' is horizontal distance of current node from root

# 'hd' is initially passed as 0

**def** getVerticalOrder(root, hd, m):

    # Base Case

**if** root **is** None:

**return**

    # Store current node in map 'm'

**try**:

        m[hd].append(root.key)

**except**:

        m[hd] **=** [root.key]

    # Store nodes in left subtree

    getVerticalOrder(root.left, hd**-**1, m)

    # Store nodes in right subtree

    getVerticalOrder(root.right, hd**+**1, m)

# The main function to print vertical order of a binary

# tree ith given root

**def** printVerticalOrder(root):

    # Create a map and store vertical order in map using

    # function getVerticalORder()

    m **=** dict()

    hd **=** 0

    getVerticalOrder(root, hd, m)

    # Traverse the map and print nodes at every horizontal

    # distance (hd)

**for** index, value **in** enumerate(sorted(m)):

**for** i **in** m[value]:

**print**(i, end**=**" ")

**print**()

# Driver program to test above function

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    root **=** Node(1)

    root.left **=** Node(2)

    root.right **=** Node(3)

    root.left.left **=** Node(4)

    root.left.right **=** Node(5)

    root.right.left **=** Node(6)

    root.right.right **=** Node(7)

    root.right.left.right **=** Node(8)

    root.right.right.right **=** Node(9)

**print**("Vertical order traversal is")

    printVerticalOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

Vertical order traversal is   
4   
2   
1 5 6   
3 8   
7   
9

**Time Complexity:**O(N log N). The hashing based solution can be considered as O(N) under the assumption that we have a good hashing function that allows insertion and retrieval operations in O(1) time. In the above C++ implementation, [map of STL](http://www.cplusplus.com/reference/map/map/) is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take O(Log N) time.

**Auxiliary Space:**O(N)

**Note:**The above solution may not print nodes in the same vertical order as they appear in the tree.

For example, the above program prints 12 before 9. See [this](https://ide.geeksforgeeks.org/TPyOLR) for a sample run.

*1*

*/    \*

*2       3*

*/  \     /  \*

*4    5  6    7*

*/  \*

*8     9*

*/   \*

*10     11*

*\*

*12*

Refer below post for a level order traversal-based solution. The below post makes sure that nodes of a vertical line are printed in the same order as they appear in the tree: [Print a Binary Tree in Vertical Order | Set 3 (Using Level Order Traversal)](https://www.geeksforgeeks.org/print-a-binary-tree-in-vertical-order-set-3-using-level-order-traversal/)

**Print vertical order traversal of the binary tree in the same order as they appear:**

To solve the problem follow the below idea:

*We can also maintain the order of nodes in the same vertical order as they appear in the tree. Nodes having the same horizontal distance will print according to level order.*

*For example, In below diagram 9 and 12 have the same horizontal distance. We can make sure that  if a node like 12 comes below in the same vertical line, it is printed after a node like 9*

***Idea:****Instead of using horizontal distance as a key in the map, we will use horizontal distance + vertical distance as the key. We know that the number of nodes can’t be more than the integer range in a binary tree.*

*We will use the first 30 bits of the key for horizontal distance [MSB to LSB] and will use the 30 next bits for vertical distance. Thus keys will be stored in the map as per our requirement.*

Follow the below steps to solve the problem:

* Declare a map to store the value of nodes at each level
* If the root is null then return from the function(Base case)
* Create an integer val and set its value to horizontal distance << 30 OR vertical distance
* Push root->data in the map using val as the key
* Recur for root->left and root->right with horizontal distance – 1, vertical distance + 1 and horizontal distance + 1, vertical distance -1 respectively
* Print the solution using map

Below is the implementation of the above approach:

* C++14
* Java
* Python3
* C#

**import** sys

# Python program for printing

# vertical order of a given binary

# tree

**class** Node:

**def** \_\_init\_\_(self, data):

        self.data **=** data

        self.left **=** None

        self.right **=** None

# Store vertical order

# in map "m", hd = horizontal

# distance, vd = vertical distance

**def** preOrderTraversal(root, hd, vd, m):

**if not** root:

**return**

    # key = horizontal

    # distance (30 bits) + vertical

    # distance (30 bits) map

    # will store key in sorted

    # order. Thus nodes having same

    # horizontal distance

    # will sort according to

    # vertical distance.

    val **=** hd << 30 | vd

    # insert in map

**if** val **in** m:

        m[val].append(root.data)

**else**:

        m[val] **=** [root.data]

    preOrderTraversal(root.left, hd**-**1, vd**+**1, m)

    preOrderTraversal(root.right, hd**+**1, vd**+**1, m)

**def** verticalOrder(root):

    mp **=** dict()

    preOrderTraversal(root, 0, 0, mp)

    # print dictionary

    prekey **=** sys.maxsize

**for** i **in** sorted(mp.keys()):

**if** (prekey !**=** sys.maxsize) **and** (i >> 30 !**=** prekey):

**print**()

        prekey **=** i >> 30

**for** j **in** mp[i]:

**print**(j, end**=**" ")

# Driver code

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

root.right.left **=** Node(6)

root.right.right **=** Node(7)

root.right.left.right **=** Node(8)

root.right.right.right **=** Node(9)

**print**("Vertical order traversal :- ")

verticalOrder(root)

# This code is contributed by prashantchandelme.

**Output**

Vertical order traversal :-   
4   
2   
1 5 6   
3 8   
7   
9

**Time Complexity:** O(N Log N)

**Auxiliary Space:**O(N)

*From <*[*https://www.geeksforgeeks.org/vertical-order-traversal-of-binary-tree-using-map/*](https://www.geeksforgeeks.org/vertical-order-traversal-of-binary-tree-using-map/)*>*

**Print Right View of a Binary Tree**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 28 Sep, 2022
* Read
* Discuss(115)
* Courses
* Practice
* Video

Given a Binary Tree, print the Right view of it.

*The****right view of a Binary Tree****is a set of nodes visible when the tree is visited from the Right side.*

**Examples:**

***Input:***

*1*

*/     \*

*2        3*

*/   \       /  \*

*4     5   6    7*

*\*

*8*

***Output****: Right view of the tree is 1 3 7 8*

***Input:***

*1*

*/*

*8*

*/*

*7*

***Output****: Right view of the tree is 1 8 7*

Recommended Practice

[Right View of Binary Tree](https://practice.geeksforgeeks.org/problems/right-view-of-binary-tree/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/right-view-of-binary-tree/1/)

**Right View of a Binary Tree using**[Recursion](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)**:**

*The idea is to use recursion and keep track of the maximum level also. And traverse the tree in a manner that the right subtree is visited before the left subtree.*

Follow the steps below to solve the problem:

* Perform **Postorder** traversal to get the rightmost node first
* Maintain a variable name **max\_level**which will store till which it prints the right view
* While traversing the tree in a postorder manner if the current level is greater than max\_level then print the current node and update max\_level by the current level

Below is the implementation of the above approach:

* C++
* C
* Java
* Python
* C#
* Javascript

# Python program to print right view of Binary Tree

# A binary tree node

**class** Node:

    # A constructor to create a new Binary tree Node

**def** \_\_init\_\_(self, item):

        self.data **=** item

        self.left **=** None

        self.right **=** None

# Recursive function to print right view of Binary Tree

# used max\_level as reference list ..only max\_level[0]

# is helpful to us

**def** rightViewUtil(root, level, max\_level):

    # Base Case

**if** root **is** None:

**return**

    # If this is the last node of its level

**if** (max\_level[0] < level):

**print** "%d   " **%** (root.data),

        max\_level[0] **=** level

    # Recur for right subtree first, then left subtree

    rightViewUtil(root.right, level**+**1, max\_level)

    rightViewUtil(root.left, level**+**1, max\_level)

**def** rightView(root):

    max\_level **=** [0]

    rightViewUtil(root, 1, max\_level)

# Driver program to test above function

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

root.right.left **=** Node(6)

root.right.right **=** Node(7)

root.right.left.right **=** Node(8)

rightView(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

1 3 7 8

[Right view of Binary Tree using Queue](https://www.geeksforgeeks.org/right-view-binary-tree-using-queue/)

**Time Complexity:** O(N), Traversing the Tree having N nodes

**Auxiliary Space:**O(N), Function Call stack space in the worst case.

**Right View of a Binary Tree using**[Level Order Traversal](https://www.geeksforgeeks.org/level-order-tree-traversal/)**:**

*The idea is to use****Level Order Traversal****as the last node every level gives the right view of the binary tree.*

Follow the steps below to solve the problem:

* Perform level order traversal on the tree
* At every level print the last node of that level

Below is the implementation of above approach:

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to print right

# view of Binary Tree

**from** collections **import** deque

# A binary tree node

**class** Node:

    # A constructor to create a new

    # Binary tree Node

**def** \_\_init\_\_(self, val):

        self.data **=** val

        self.left **=** None

        self.right **=** None

# Function to print Right view of

# binary tree

**def** rightView(root):

**if** root **is** None:

**return**

    q **=** deque()

    q.append(root)

**while** q:

        # Get number of nodes for each level

        n **=** len(q)

        # Traverse all the nodes of the

        # current level

**while** n > 0:

            n **-=** 1

            # Get the front node in the queue

            node **=** q.popleft()

            # Print the last node of each level

**if** n **==** 0:

                print(node.data, end**=**" ")

            # If left child is not null push it

            # into the queue

**if** node.left:

                q.append(node.left)

            # If right child is not null push

            # it into the queue

**if** node.right:

                q.append(node.right)

# Driver code

# Let's construct the tree as

# shown in example

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

root.right.left **=** Node(6)

root.right.right **=** Node(7)

root.right.left.right **=** Node(8)

rightView(root)

# This code is contributed by Pulkit Pansari

**Output**

1 3 7 8

**Time Complexity:** O(N), where N is the number of nodes in the binary tree.

**Auxiliary Space**: O(N) since using auxiliary space for queue

**Find Minimum Depth of a Binary Tree**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 21 Dec, 2022
* Read
* Discuss(106)
* Courses
* Practice
* Video

Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

For example, minimum height of below Binary Tree is 2.



Note that the path must end on a leaf node. For example, the minimum height of below Binary Tree is also 2.

10  
 /   
 5

Recommended Problem

Minimum Depth of a Binary Tree

[Tree](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Tree&sortBy=submissions)

[Data Structures](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Data%20Structures&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

[Facebook](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Facebook&sortBy=submissions)

+1 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/minimum-depth-of-a-binary-tree/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 52.1K

**Method 1:**The idea is to traverse the given Binary Tree. For every node, check if it is a leaf node. If yes, then return 1. If not leaf node then if the left subtree is NULL, then recur for the right subtree. And if the right subtree is NULL, then recur for the left subtree. If both left and right subtrees are not NULL, then take the minimum of two heights.

Below is implementation of the above idea.

* C++
* C
* Java
* Python3
* C#
* Javascript

# Python program to find minimum depth of a given Binary Tree

# Tree node

**class** Node:

**def** \_\_init\_\_(self , key):

        self.data **=** key

        self.left **=** None

        self.right **=** None

**def** minDepth(root):

    # Corner Case.Should never be hit unless the code is

    # called on root = NULL

**if** root **is** None:

**return** 0

    # Base Case : Leaf node.This accounts for height = 1

**if** root.left **is** None **and** root.right **is** None:

**return** 1

    # If left subtree is Null, recur for right subtree

**if** root.left **is** None:

**return** minDepth(root.right)**+**1

    # If right subtree is Null , recur for left subtree

**if** root.right **is** None:

**return** minDepth(root.left) **+**1

**return** min(minDepth(root.left), minDepth(root.right))**+**1

# Driver Program

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

**print** (minDepth(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

The minimum depth of binary tree is : 2

**Time Complexity:**O(n), as it traverses the tree only once.

**Auxiliary Space:**O(h), where h is the height of the tree, this space is due to the recursive call stack.

**Method 2:**The above method may end up with complete traversal of Binary Tree even when the topmost leaf is close to root. A **Better Solution** is to do Level Order Traversal. While doing traversal, returns depth of the first encountered leaf node.

Below is the implementation of this solution.

* C++
* Java
* Python3
* C#
* Javascript

# Python program to find minimum depth of a given Binary Tree

# A Binary Tree node

**class** Node:

    # Utility to create new node

**def** \_\_init\_\_(self , data):

        self.data **=** data

        self.left **=** None

        self.right **=** None

**def** minDepth(root):

    # Corner Case

**if** root **is** None:

**return** 0

    # Create an empty queue for level order traversal

    q **=** []

    # Enqueue root and initialize depth as 1

    q.append({'node': root , 'depth' : 1})

    # Do level order traversal

**while**(len(q)>0):

        # Remove the front queue item

        queueItem **=** q.pop(0)

        # Get details of the removed item

        node **=** queueItem['node']

        depth **=** queueItem['depth']

        # If this is the first leaf node seen so far

        # then return its depth as answer

**if** node.left **is** None **and** node.right **is** None:

**return** depth

        # If left subtree is not None, add it to queue

**if** node.left **is not** None:

            q.append({'node' : node.left , 'depth' : depth**+**1})

        # if right subtree is not None, add it to queue

**if** node.right **is not** None:

            q.append({'node': node.right , 'depth' : depth**+**1})

# Driver program to test above function

# Lets construct a binary tree shown in above diagram

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

**print** (minDepth(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

2

**Time Complexity:**O(n), where n is the number of nodes in the given binary tree. This is due to the fact that we are visiting each node once.

**Auxiliary Space:** O(n), as we need to store the elements in a queue for level order traversal.

**Method 3:**

* C++
* Java
* Python3
* C#
* Javascript

# Python implementation to find minimum depth

# of a given Binary tree

# Class containing left and right child of current

# Node and key value

**class** Node:

    # Constructor to create a new node

**def** \_\_init\_\_(self, d):

        self.data **=** d

        self.left **=** None

        self.right **=** None

# Function to calculate the minimum depth of the tree

**def** minimumDepth(root, level):

**if** (root **==** None):

**return** level;

    level **+=** 1;

**return** min(minimumDepth(root.left, level),

                        minimumDepth(root.right, level))

# Driver program to test above functions

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

print("The minimum depth of ","binary tree is : ", minimumDepth(root, 0))

# This code is contributed by ab2127

**Output**

2

**Time Complexity:**O(n), where n is the number of nodes

**Auxiliary Space:**O(h), where h is the height of the tree, this space is due to the recursive call stack.

*From <*[*https://www.geeksforgeeks.org/find-minimum-depth-of-a-binary-tree/*](https://www.geeksforgeeks.org/find-minimum-depth-of-a-binary-tree/)*>*

**Difference between BFS and DFS**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 12 Sep, 2022
* Read
* Discuss(13)
* Courses
* Practice
* Video

**Breadth-First Search:**

**BFS, Breadth-First Search,** is a vertex-based technique for finding the shortest path in the graph. It uses a [Queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

**Example**:

**Input:**  
 A  
 / \  
 B C  
 / / \  
 D E F

**Output:**

A, B, C, D, E, F

**Depth First Search:**

**DFS,** [**Depth First Search**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/), is an edge-based technique. It uses the [Stack data structure](https://www.geeksforgeeks.org/stack-data-structure/) and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

**Example:**

**Input:**  
 A  
 / \  
 B D  
 / / \  
 C E F

**Output:**

A, B, C, D, E, F

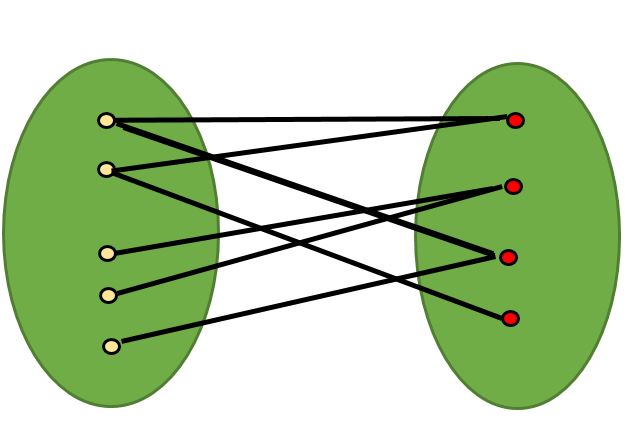
**BFS vs DFS**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | Parameters | BFS | DFS |
| 1. | Stands for | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| 2. | Data Structure | BFS(Breadth First Search) uses Queue data structure for finding the shortest path. | DFS(Depth First Search) uses Stack data structure. |
| 3. | Definition | BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level. | DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes. |
| 4. | Technique | BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex. | In DFS, we might traverse through more edges to reach a destination vertex from a source. |
| 5. | Conceptual Difference | BFS builds the tree level by level. | DFS builds the tree sub-tree by sub-tree. |
| 6. | Approach used | It works on the concept of FIFO (First In First Out). | It works on the concept of LIFO (Last In First Out). |
| 7. | Suitable for | BFS is more suitable for searching vertices closer to the given source. | DFS is more suitable when there are solutions away from source. |
| 8. | Suitable for Decision Treestheirwinning | BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop. |
| 9. | Time Complexity | The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. | The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. |
| 10. | Visiting of Siblings/ Children | Here, siblings are visited before the children. | Here, children are visited before the siblings. |
| 11. | Removal of Traversed Nodes | Nodes that are traversed several times are deleted from the queue. | The visited nodes are added to the stack and then removed when there are no more nodes to visit. |
| 12. | Backtracking | In BFS there is no concept of backtracking. | DFS algorithm is a recursive algorithm that uses the idea of backtracking |
| 13. | Applications | BFS is used in various applications such as bipartite graphs, shortest paths, etc. | DFS is used in various applications such as acyclic graphs and topological order etc. |
| 14. | Memory | BFS requires more memory. | DFS requires less memory. |
| 15. | Optimality | BFS is optimal for finding the shortest path. | DFS is not optimal for finding the shortest path. |
| 16. | Space complexity | In BFS, the space complexity is more critical as compared to time complexity. | DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node. |
| 17. | Speed | BFS is slow as compared to DFS. | DFS is fast as compared to BFS. |
| 18. | When to use? | When the target is close to the source, BFS performs better. | When the target is far from the source, DFS is preferable. |

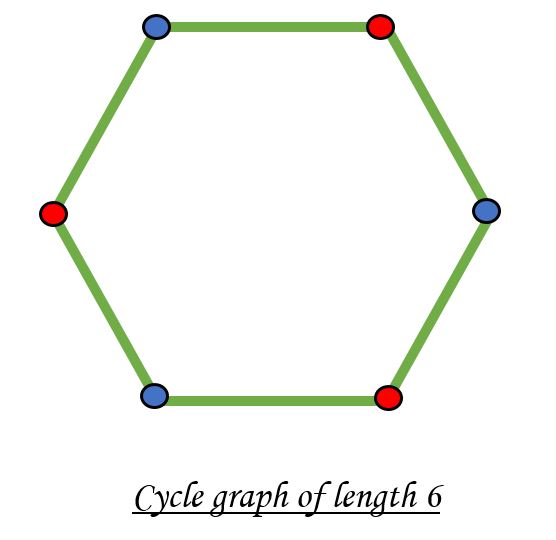
**Check whether a given graph is Bipartite or not**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 15 Dec, 2022
* Read
* Discuss(153)
* Courses
* Practice
* Video

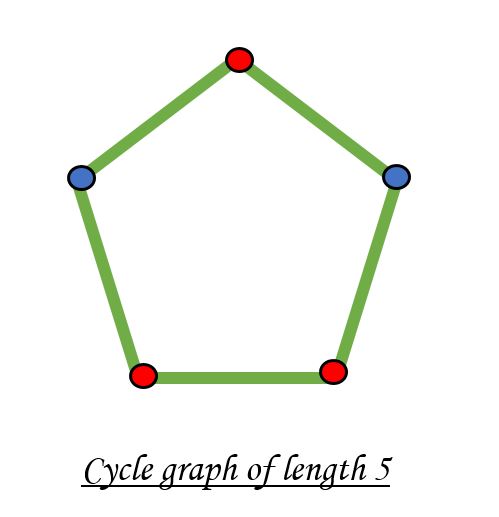
A [Bipartite Graph](http://en.wikipedia.org/wiki/Bipartite_graph) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



It is not possible to color a cycle graph with odd cycle using two colors.



*Algorithm to check if a graph is Bipartite:*

One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm m coloring problem](https://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/).

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).

2. Color all the neighbors with BLUE color (putting into set V).

3. Color all neighbor’s neighbor with RED color (putting into set U).

4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.

5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Recommended Problem

Bipartite Graph

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

[DFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=DFS&sortBy=submissions)

+3 more

[Samsung](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Samsung&sortBy=submissions)

[Microsoft](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Microsoft&sortBy=submissions)

+1 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/bipartite-graph/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 96K

* C++
* Java
* Python3
* C#
* Javascript

# Python program to find out whether a

# given graph is Bipartite or not

**class** Graph():

**def** \_\_init\_\_(self, V):

        self.V **=** V

        self.graph **=** [[0 **for** column **in** range(V)] \

**for** row **in** range(V)]

    # This function returns true if graph G[V][V]

    # is Bipartite, else false

**def** isBipartite(self, src):

        # Create a color array to store colors

        # assigned to all vertices. Vertex

        # number is used as index in this array.

        # The value '-1' of  colorArr[i] is used to

        # indicate that no color is assigned to

        # vertex 'i'. The value 1 is used to indicate

        # first color is assigned and value 0

        # indicates second color is assigned.

        colorArr **=** [**-**1] **\*** self.V

        # Assign first color to source

        colorArr[src] **=** 1

        # Create a queue (FIFO) of vertex numbers and

        # enqueue source vertex for BFS traversal

        queue **=** []

        queue.append(src)

        # Run while there are vertices in queue

        # (Similar to BFS)

**while** queue:

            u **=** queue.pop()

            # Return false if there is a self-loop

**if** self.graph[u][u] **==** 1:

**return** False;

**for** v **in** range(self.V):

                # An edge from u to v exists and destination

                # v is not colored

**if** self.graph[u][v] **==** 1 **and** colorArr[v] **== -**1:

                    # Assign alternate color to this

                    # adjacent v of u

                    colorArr[v] **=** 1 **-** colorArr[u]

                    queue.append(v)

                # An edge from u to v exists and destination

                # v is colored with same color as u

**elif** self.graph[u][v] **==** 1 **and** colorArr[v] **==** colorArr[u]:

**return** False

        # If we reach here, then all adjacent

        # vertices can be colored with alternate

        # color

**return** True

# Driver program to test above function

g **=** Graph(4)

g.graph **=** [[0, 1, 0, 1],

            [1, 0, 1, 0],

            [0, 1, 0, 1],

            [1, 0, 1, 0]

            ]

**print** ("Yes" **if** g.isBipartite(0) **else** "No")

# This code is contributed by Divyanshu Mehta

**Output**

Yes

**Time Complexity**: O(V\*V) as adjacency matrix is used for graph but can be made O(V+E) by using adjacency list

**Auxiliary Space:** O(V) due to queue and color vector.

**The above algorithm works only if the** **graph is connected**. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly called above method for all not yet visited vertices.

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find out whether a

# given graph is Bipartite or not

**class** Graph():

**def** \_\_init\_\_(self, V):

        self.V **=** V

        self.graph **=** [[0 **for** column **in** range(V)]

**for** row **in** range(V)]

        self.colorArr **=** [**-**1 **for** i **in** range(self.V)]

    # This function returns true if graph G[V][V]

    # is Bipartite, else false

**def** isBipartiteUtil(self, src):

        # Create a color array to store colors

        # assigned to all vertices. Vertex

        # number is used as index in this array.

        # The value '-1' of self.colorArr[i] is used

        # to indicate that no color is assigned to

        # vertex 'i'. The value 1 is used to indicate

        # first color is assigned and value 0

        # indicates second color is assigned.

        # Assign first color to source

        # Create a queue (FIFO) of vertex numbers and

        # enqueue source vertex for BFS traversal

        queue **=** []

        queue.append(src)

        # Run while there are vertices in queue

        # (Similar to BFS)

**while** queue:

            u **=** queue.pop()

            # Return false if there is a self-loop

**if** self.graph[u][u] **==** 1:

**return** False

**for** v **in** range(self.V):

                # An edge from u to v exists and

                # destination v is not colored

**if** (self.graph[u][v] **==** 1 **and**

                        self.colorArr[v] **== -**1):

                    # Assign alternate color to

                    # this adjacent v of u

                    self.colorArr[v] **=** 1 **-** self.colorArr[u]

                    queue.append(v)

                # An edge from u to v exists and destination

                # v is colored with same color as u

**elif** (self.graph[u][v] **==** 1 **and**

                      self.colorArr[v] **==** self.colorArr[u]):

**return** False

        # If we reach here, then all adjacent

        # vertices can be colored with alternate

        # color

**return** True

**def** isBipartite(self):

        self.colorArr **=** [**-**1 **for** i **in** range(self.V)]

**for** i **in** range(self.V):

**if** self.colorArr[i] **== -**1:

**if not** self.isBipartiteUtil(i):

**return** False

**return** True

# Driver Code

g **=** Graph(4)

g.graph **=** [[0, 1, 0, 1],

           [1, 0, 1, 0],

           [0, 1, 0, 1],

           [1, 0, 1, 0]]

print ("Yes" **if** g.isBipartite() **else** "No")

# This code is contributed by Anshuman Sharma

**Output**

Yes

**Time complexity**: **O(V+E)**.

**Auxiliary Space: O(V),**because we have a V-size array.

**If Graph is represented using Adjacency List** .Time Complexity will be O(V+E).

Works for connected as well as disconnected graph.

* C++
* Java
* Python3
* C#
* Javascript

**def** isBipartite(V, adj):

    # vector to store colour of vertex

    # assigning all to -1 i.e. uncoloured

    # colours are either 0 or 1

    # for understanding take 0 as red and 1 as blue

    col **=** [**-**1]**\***(V)

    # queue for BFS storing {vertex , colour}

    q **=** []

    #loop incase graph is not connected

**for** i **in** range(V):

        # if not coloured

**if** (col[i] **== -**1):

            # colouring with 0 i.e. red

            q.append([i, 0])

            col[i] **=** 0

**while** len(q) !**=** 0:

                p **=** q[0]

                q.pop(0)

                # current vertex

                v **=** p[0]

                # colour of current vertex

                c **=** p[1]

                # traversing vertexes connected to current vertex

**for** j **in** adj[v]:

                    # if already coloured with parent vertex color

                    # then bipartite graph is not possible

**if** (col[j] **==** c):

**return** False

                    # if uncoloured

**if** (col[j] **== -**1):

                        # colouring with opposite color to that of parent

**if** c **==** 1:

                            col[j] **=** 0

**else**:

                            col[j] **=** 1

                        q.append([j, col[j]])

    # if all vertexes are coloured such that

    # no two connected vertex have same colours

**return** True

V, E **=** 4, 8

# adjacency list for storing graph

adj **=** []

adj.append([1,3])

adj.append([0,2])

adj.append([1,3])

adj.append([0,2])

ans **=** isBipartite(V, adj)

# returns 1 if bipartite graph is possible

**if** (ans):

    print("Yes")

# returns 0 if bipartite graph is not possible

**else**:

**print**("No")

    # This code is contributed by divyesh072019.

**Output**

Yes

**Time Complexity:**O(V+E)

**Auxiliary Space:**O(V)

**Exercise:**

**1.** Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

Solution :

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find out whether a given

# graph is Bipartite or not using recursion.

V **=** 4

**def** colorGraph(G, color, pos, c):

**if** color[pos] !**= -**1 **and** color[pos] !**=** c:

**return** False

    # color this pos as c and all its neighbours and 1-c

    color[pos] **=** c

    ans **=** True

**for** i **in** range(0, V):

**if** G[pos][i]:

**if** color[i] **== -**1:

                ans &**=** colorGraph(G, color, i, 1**-**c)

**if** color[i] !**=-**1 **and** color[i] !**=** 1**-**c:

**return** False

**if not** ans:

**return** False

**return** True

**def** isBipartite(G):

    color **=** [**-**1] **\*** V

    #start is vertex 0

    pos **=** 0

    # two colors 1 and 0

**return** colorGraph(G, color, pos, 1)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    G **=** [[0, 1, 0, 1],

         [1, 0, 1, 0],

         [0, 1, 0, 1],

         [1, 0, 1, 0]]

**if** isBipartite(G): print("Yes")

**else**: **print**("No")

# This code is contributed by Rituraj Jain

**Output**

Yes

**Time Complexity:**O(V+E)

**Auxiliary Space:**O(V)

**Find the Maximum Depth or Height of given Binary Tree**

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 10 Nov, 2022
* Read
* Discuss(311)
* Courses
* Practice
* Video

Given a binary tree, the task is to find the height of the tree. Height of the tree is the number of edges in the tree from the root to the deepest node, Height of the empty tree is **0**.



Recommended Problem

Height of Binary Tree

[Tree](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Tree&sortBy=submissions)

[Data Structures](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Data%20Structures&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

[Cadence India](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Cadence%20India&sortBy=submissions)

+12 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/height-of-binary-tree/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 1.8L

Recursively calculate height of **left**and **right** subtrees of a node and assign height to the node as**max of the heights of two children plus 1**. See below pseudo code and program for details.

**Illustration:**

*Consider the following graph:*



*maxDepth(‘1’) = max(maxDepth(‘2’), maxDepth(‘3’)) + 1 = 2 + 1*

*because recursively*

*maxDepth(‘2’) =  max (maxDepth(‘4’), maxDepth(‘5’)) + 1 = 1 + 1 and  (as height of both ‘4’ and ‘5’ are 1)*

*maxDepth(‘3’) = 1*

Follow the below steps to Implement the idea:

* Recursively do a Depth-first search.
* If the tree is empty then return -1
* Otherwise, do the following
* Get the max depth of the left subtree recursively  i.e. call maxDepth( tree->left-subtree)
* Get the max depth of the right subtree recursively  i.e. call maxDepth( tree->right-subtree)
* Get the max of max depths of **left**and **right** subtrees and **add 1**to it for the current node.
* Return max\_depth.

Below is the Implementation of the above approach:

* C++
* C
* Java
* Python3
* C#
* Javascript

# Python3 program to find the maximum depth of tree

# A binary tree node

**class** Node:

    # Constructor to create a new node

**def** \_\_init\_\_(self, data):

        self.data **=** data

        self.left **=** None

        self.right **=** None

# Compute the "maxDepth" of a tree -- the number of nodes

# along the longest path from the root node down to the

# farthest leaf node

**def** maxDepth(node):

**if** node **is** None:

**return** 0

**else**:

        # Compute the depth of each subtree

        lDepth **=** maxDepth(node.left)

        rDepth **=** maxDepth(node.right)

        # Use the larger one

**if** (lDepth > rDepth):

**return** lDepth**+**1

**else**:

**return** rDepth**+**1

# Driver program to test above function

root **=** Node(1)

root.left **=** Node(2)

root.right **=** Node(3)

root.left.left **=** Node(4)

root.left.right **=** Node(5)

**print**("Height of tree is %d" **%** (maxDepth(root)))

# This code is contributed by Amit Srivastav

**Output**

Height of tree is 3

**Time Complexity:**O(N) (Please see our post [Tree Traversal](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)for details)

**Auxiliary Space:** O(N) due to recursive stack.

**Find the Maximum Depth or Height of a Tree using**[Level Order Traversal](https://www.geeksforgeeks.org/level-order-tree-traversal/)**:**

*Do*[*Level Order Traversal*](https://www.geeksforgeeks.org/level-order-tree-traversal/)*, while adding Nodes at each level to*[*Queue*](https://www.geeksforgeeks.org/queue-data-structure/)*, we have to add****NULL Node****so that whenever it is encountered, we can increment the value of variable and that level get counted.*

Follow the below steps to Implement the idea:

* Traverse the tree in level order traversal starting from **root**.
* Initialize an empty queue **Q**, a variable **depth**and push **root**, then push **null**into the **Q**.
* Run a while loop till **Q**is not empty.
* Store the front element of **Q**and Pop out the front element.
* If the front of **Q**is **NULL**then increment **depth**by one and if queue is not empty then push **NULL**into the **Q**.
* Else if the element is not **NULL**then check for its **left**and **right**children and if they are not **NULL**push them into **Q**.
* Return **depth**.

Below is the Implementation of the above approach:

* C++
* Java
* Python3
* Javascript

# Python code to implement the approach

# A Tree node

**class** Node:

**def** \_\_init\_\_(self):

        self.key **=** 0

        self.left, self.right **=** None, None

# Utility function to create a new node

**def** newNode(key):

    temp **=** Node()

    temp.key **=** key

    temp.left, temp.right **=** None, None

**return** temp

# Function to find the height(depth) of the tree

**def** height(root):

    # Initialising a variable to count the

    # height of tree

    depth **=** 0

    q **=** []

    # appending first level element along with None

    q.append(root)

    q.append(None)

**while**(len(q) > 0):

        temp **=** q[0]

        q **=** q[1:]

        # When None encountered, increment the value

**if**(temp **==** None):

            depth **+=** 1

        # If None not encountered, keep moving

**if**(temp !**=** None):

**if**(temp.left):

                q.append(temp.left)

**if**(temp.right):

                q.append(temp.right)

        # If queue still have elements left,

        # append None again to the queue.

**elif**(len(q) > 0):

            q.append(None)

**return** depth

# Driver program

# Let us create Binary Tree shown in above example

root **=** newNode(1)

root.left **=** newNode(2)

root.right **=** newNode(3)

root.left.left **=** newNode(4)

root.left.right **=** newNode(5)

**print**(f"Height(Depth) of tree is: {height(root)}")

# This code is contributed by shinjanpatra

**Output**

Height(Depth) of tree is: 3

**Time Complexity:**O(N)

**Auxiliary Space:**O(N)