Tree

1.- A tree is a type of hierarchical data structure composed of nodes. Nodes, which are joined by edges, represent value. The primary qualities of a tree are as follows:

1. The root node appears to be the tree's single node. This is where the tree originates, although it has no parents.

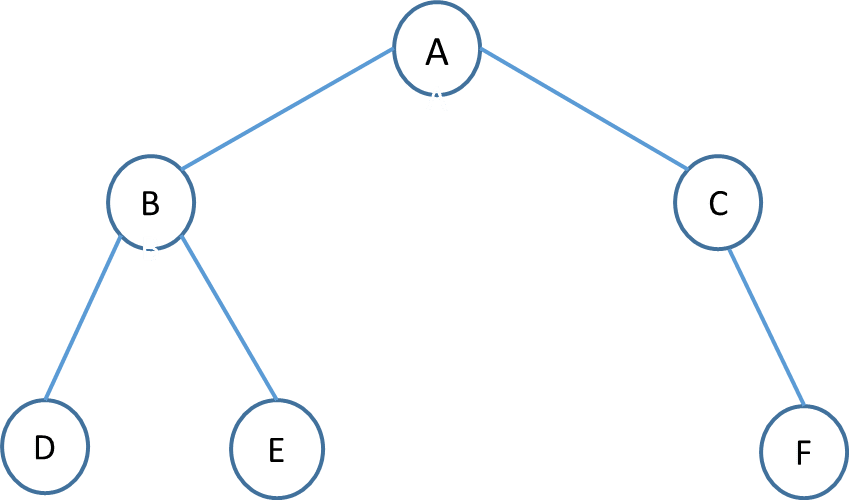
2. There is only one parent for each node, but there are several children.

3. Each node has an edge that connects it to its descendants.

The amount of children a node has is determined on the tree type. Trees are divided into two categories:

- A general tree is one in which the maximum number of children a node can have is unrestricted. Two examples are the family tree and the folder structure.

-Every node in a binary tree can only have two offspring, left and right. In the diagram below, B and D are left children, whereas C, E, and F are right children.



The Advantages of Trees:

-The structural connections in the data can be seen in trees.

-Hierarchies are illustrated using trees.

-Insertion and searching was made much easier using trees.

-Trees are indeed an incredibly versatile type of data since they allow you to shift subtrees around with little effort.

The Disadvantages of Trees:

-When reading a specific node, sequential storage may consume space (when it is not a complete binary tree), but it is more efficient. O(0)

-When compared to binary trees, chain storage takes less space, but it is less efficient when reading a specific node. O(nlogn)

1. Used by compilers to build syntax trees.
2. Used to implement expression parsers and expression solvers.
3. Used to store router-tables in routers.
4. Used in many search applications where data are constantly entering and leaving.

As a base for data structures used in computational geometry

Implementation in C

**struct** node {

**int** data;

**struct** node\* left;

**struct** node\* right;

};

**struct** node\* newNode(**int** data)

{

**struct** node\* node

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

    node->data = data;

    node->left = NULL;

    node->right = NULL;

**return** (node);

}

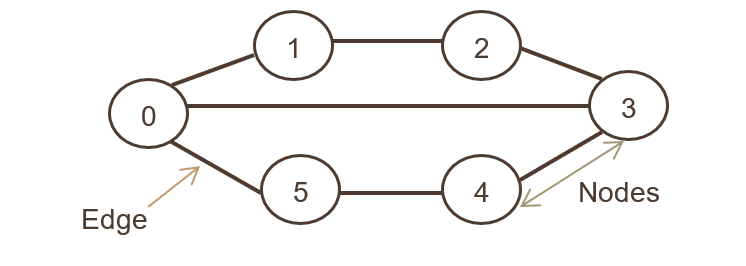
Graph

Graphs are non-linear data structures made up of nodes and edges. The edges are lines or arcs that connect any two nodes in the graph, and the nodes are also known as vertices. The main qualities of a graph are as follows:

1. Graphs are mostly used to characterize graphs based on their structure.

2. Data that is kept in a network of interconnected edges (paths) and vertices (nodes).

3. A graph data structure (N, E) is made up of a number of nodes and edges. Both nodes and vertices must have a finite number of nodes and vertices.



Advantage of graphs, that graphs is used to makes large data simpler to work with

Disadvantage of graphs is the large memory complexity.

Graph is good for data structures that you use every day through social media and other media platforms. Graph is used connect elements that share connections.

**struct** AdjListNode

{

**int** dest;

**struct** AdjListNode\* next;

};

**struct** AdjList

{

**struct** AdjListNode \*head;

};

**struct** Graph

{

**int** V;

**struct** AdjList\* array;

};

**struct** AdjListNode\* newAdjListNode(**int** dest)

{

**struct** AdjListNode\* newNode =

     (**struct** AdjListNode\*) **malloc**(**sizeof**(**struct** AdjListNode));

    newNode->dest = dest;

    newNode->next = NULL;

**return** newNode;

}

**struct** Graph\* createGraph(**int** V)

{

**struct** Graph\* graph =

        (**struct** Graph\*) **malloc**(**sizeof**(**struct** Graph));

    graph->V = V;

      graph->array =

      (**struct** AdjList\*) **malloc**(V \* **sizeof**(**struct** AdjList));

**int** i;

**for** (i = 0; i < V; ++i)

        graph->array[i].head = NULL;

**return** graph;

}

**void** addEdge(**struct** Graph\* graph, **int** src, **int** dest)

{

**struct** AdjListNode\* newNode = newAdjListNode(dest);

    newNode->next = graph->array[src].head;

    graph->array[src].head = newNode;

    newNode = newAdjListNode(src);

    newNode->next = graph->array[dest].head;

    graph->array[dest].head = newNode;

}

**void** printGraph(**struct** Graph\* graph)

{

**int** v;

**for** (v = 0; v < graph->V; ++v)

    {

**struct** AdjListNode\* pCrawl = graph->array[v].head;

**printf**("\n Adjacency list of vertex %d\n head ", v);

**while** (pCrawl)

        {

**printf**("-> %d", pCrawl->dest);

            pCrawl = pCrawl->next;

        }

**printf**("\n");

    }

}

Tree and Graph implementation

1. In order Tree traversal without Recursion

The first step is creating an empty stack S, and then set the current node as root

Push the current node to S and set the current = current-> until the current read as NULL

If the current is NULL and the stack isn’t empty then

Pop the top item from the stack, print and set current = current\_item->right

Back to pushing the current node to S until the current read as NULL

And if the current is NULL and stack is empty the program is done

Let’s say we have a

1

/ \

2 3

/ \

4 5

The first step is to create an empty Stack which is S = NULL

Then set the current as address of root current->1

Push the current node and set current = current->left until the current is NULL

current -> 1

push 1: Stack S -> 1

current -> 2

push 2: Stack S -> 2, 1

current -> 4

push 4: Stack S -> 4, 2, 1

current = NULL

the next step is pop from S

pop 4 from Stack S 🡪 2,1 and print “4” current will be null (right side of 4) and go to the previous step, pushes 5 tro stack and makes current NULL

Stack S 🡪 5, 1

current = NULL

pop from S, pop 5 Stack S 🡪 1, print “5”, current = NULL (right side of 5) and back to step previous step before pop 4 since the current is NULL this step doesn’t do anything

keep repeating until the stack is Stack S 🡪 NULL then print the last one “3” that makes current = NULL (right side of 3), the traversal is done and stack S is empty and the current is NULL

1. Detect Cycle in a Directed Graph

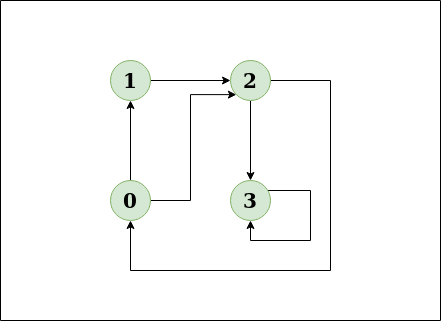
Given a directed graph, check if the graph contains a cycle or doesn’t contain a cycle, if the graph return a cycle then the return should be true else return false.

Say the input is n = 4, e = 6

0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output: Yes

The diagram :



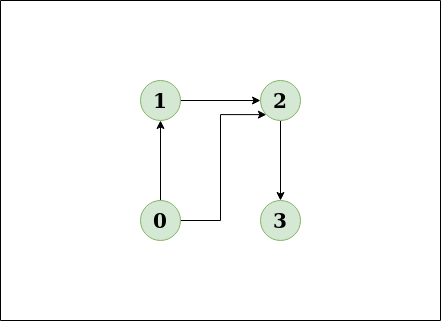
The diagram clearly how that the cycle is 0🡪2🡪0

If the input n = 4, e = 4

0 -> 1, 0 -> 2, 1 -> 2, 2 -> 3

Output:No

The diagram :



This diagram doesn’t shows any cycle.

The algorithm as following :

1. Make a graph with the specified number of edges and vertices.

2. Create a recursive function that sets the current index or vertex, visited, and recursion stack to their initial values.

3. Mark the index in the recursion stack for the current node that has previously been visited.

4. Locate all of the unvisited vertices that are near to the current node. Call the function for those vertices recursively, and if the recursive function returns true, return true.

5. If the adjacent vertices have previously been marked on the recursion, return true.

6. Make a wrapper class that calls recursive functions for all vertices and returns true if any function returns true. Otherwise, return false.

References :

<https://www.javatpoint.com/tree>

<https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/>

<https://www.geeksforgeeks.org/graph-and-its-representations/>

<https://www.geeksforgeeks.org/basic-properties-of-a-graph/>

<https://www.baeldung.com/cs/graphs>

<https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>

<https://www.upgrad.com/blog/graphs-in-data-structure/>