Graphs

A graph is a set of **nodes**, some of which are connected with **edges**.

- **node:** A node or a **vertex** is a structure that contains data.
- **edge:** An edge is the connection between two nodes, they are also called **arcs**. Edges can be
  - directed/undirected
  - weighted/unweighted

Graphs (cont'd)

- Two nodes are said to be **adjacent** if there exists an edge between them. A graph node can be adjacent to zero or more nodes.
- A graph is **connected** if there is a path from any node to any other node in the graph.

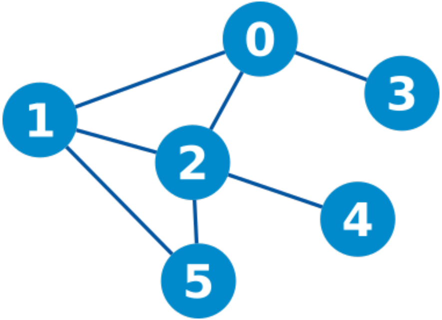


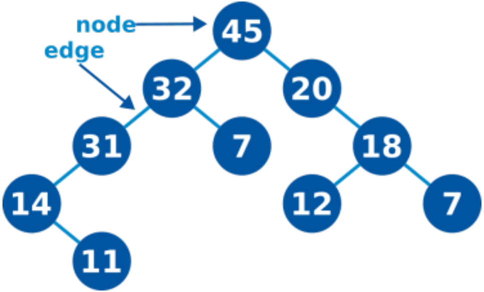
Figure: An undirected, unweighted graph.

Trees

Tree is a nonlinear data structure that stores data hierarchically. A tree is made of **nodes** and **edges**. Trees are specialized types of graphs.

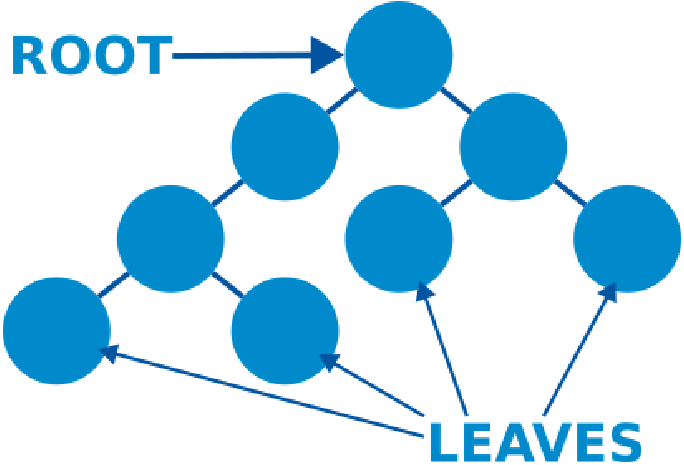
- **node:** A tree node is where the data of the tree is stored. It's also called a **vertex**.
- **edge:** A tree edge is the connection between two tree nodes. Edges can be weighted and unweighted.

Each node has zero or more **children** and except the first node of the tree, **root**, each node has exactly one **parent**.



Trees (cont'd)

- The first node of the tree is called the tree's **root**.
- Outer nodes of the tree are called the **leaves** of the tree.



Trees (cont'd)

- Children of the same parent are called **siblings**.
- Node a is **ancestor** of node b if a $=$ b or a is an ancestor of b's parent.
- Node b is **descendant** of node a if a is an ancestor of b.
- A **subtree** with root v is all descendent nodes of v.

Graph and Tree Representations

There are many ways to represent graphs and trees in computers. We will be covering:

- Node based implementations
- Adjacency List
- Adjacency Matrix

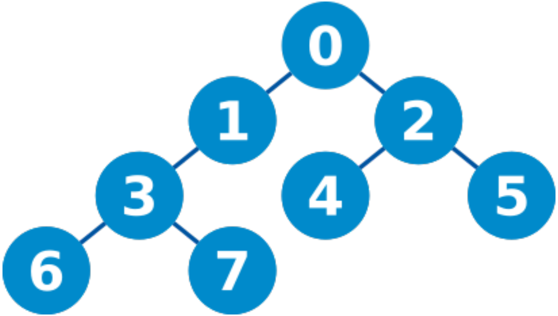
C++ Node Based Implementations

```cpp
class GraphNode {
public:
    int data;
    vector<GraphNode*> adj;
};

class Graph {
public:
    vector<GraphNode*> nodes;
    /*
    GraphNode* startingNode;
    */
};
```

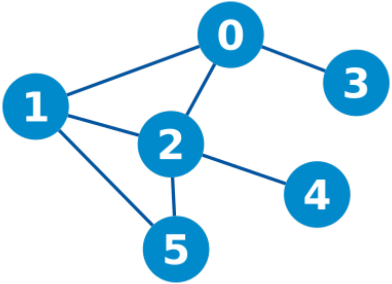
```cpp
class TreeNode {
public:
    int data;
    vector<TreeNode*> children;
};

class Tree {
public:
    TreeNode* root;
};
```

Adjacency List

For each node, we keep a list of its adjacent nodes.



Adj. List: [[1, 2], [3], [4, 5], [6, 7], [], [], [], []]



Adj. List:
[[1, 2, 3], [0, 2, 5], [0, 1, 4, 5], [0], [2], [1, 2]]

Adjacency Matrix

We will create a matrix sized n x n where n is the number of nodes in the graph/tree we want to represent. We place a "1" if there is an edge between nodes, 0 otherwise.

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	0	0	0	1	0	0	0	0
2	0	0	0	0	1	1	0	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Table: Adjacency Matrix of the tree

	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	0	0	0	1
2	1	0	0	0	1	1
3	1	0	0	0	0	0
4	0	1	0	0	0	0
5	1	1	0	0	0	0

Table: Adjacency Matrix of the graph

References

Michael T. Goodrich, Roberto Tamassia, David M. Mount - Data Structures and Algorithms in C++ 2nd Edition - Wiley (2011)

Reviewer: Novruz Amirov

ALGO-101

Week 5 - Graphs and Trees

Fatih Baskın

ITU ACM

November 2022

Topics

Topics covered at week 5:

- Graphs
  - DFS and BFS
  - Topological Sort
- Trees
  - Tree Traversals
  - Binary Search Tree

Graphs, Definitions

- **Node** is a data element of a graph. Also called **vertex**.
- **Edge** is a line that connects two nodes.
  - Shown as **e = (v1, v2)**.
  - In this case, edge **e** is **incident** to nodes **v1** and **v2**.
  - Can be weighted, if so they are called **weighted edge**.
  - If graph is directed, they are called **arc**.
- **Adjacent** nodes are connected by an edge.
- **Self-loop** is an edge that connects a node to itself.
- Two nodes can be connected by more than one edges, these edges are called **parallel edges**.
- A **plain graph** does not contain any self-loops or parallel edges. If so, that graph is called **multigraph**.

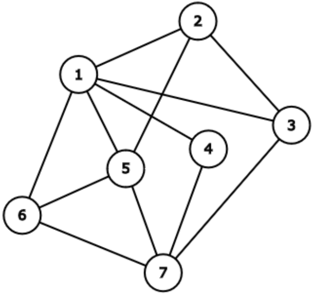


Figure: A plain graph example

Graphs, Examples

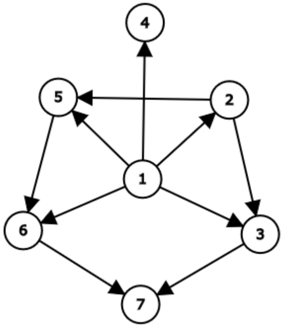


Figure: Directed graph example

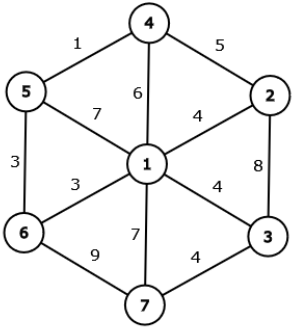


Figure: Weighted graph example

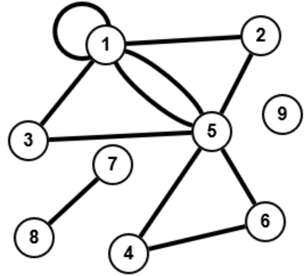


Figure: Multigraph example

Paths, Walks, Trails, Circuits, Cycles

- **Walk** is a sequence of nodes and edges in a graph.
- **Trail** is a walk without visiting the same edge.
- **Circuit** is a trail that has the same node at the start and end.
- **Path** is a walk without visiting same node.
- **Cycle** is a circuit without visiting same node.
- **Spanning trail** covers all edges.
- **Spanning cycle** covers all nodes.
- **Euler graphs** contains closed spanning trail.
- **Hamilton graphs** contains a closed spanning path.
    - ref: inzva, 2018

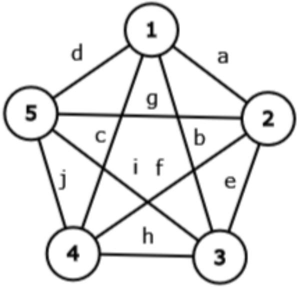


Figure: Find paths, walks, trails, circuits, cycles

Graphs, Definitions Cont.

- **Degree** of a node means number of incident edges.
  - $d_1 = 6$, $d_2 = 2$ ...
  - **In-degree** and **out-degree** for weighted graphs.
- **Connected graphs** have a path between every pair of nodes.
- Disconnected graphs can be divided into **connected components**.
  - 1, 2, 3, 4, 5, 6
  - 7, 8
  - 9
- **Distance** between nodes is the length/weight of shortest path between those nodes.
- Largest distance in graph is called **diameter** of graph.

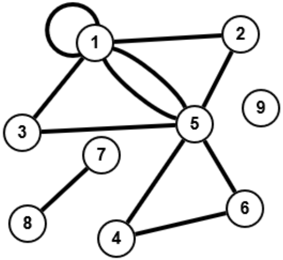


Figure: Disconnected graph example

Special Graphs

- **Completely connected graphs** have an edge between every pair of nodes.
  - Number of nodes: n
  - Special name K_n
  - Number of edges: $\binom{n}{k} = \frac{n(n-1)}{2}$
- **Tree** is a special type of graph.
  - Undirected,
  - Connected,
  - No cycles, exactly one path between every pair of nodes,
  - If number of nodes is n, number of edges are (n - 1).
- **Bipartite graph** is a graph whose vertices can be divided into two disjoint and independent sets.
  - ref: Uyar et al., 2016
- All nodes of a **regular graph** have the same degree.
  - **n-regular**: All nodes have degree n.

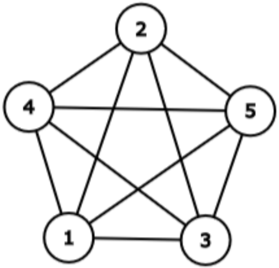


Figure: Completely connected graph example, K_5

Special Graphs, Examples

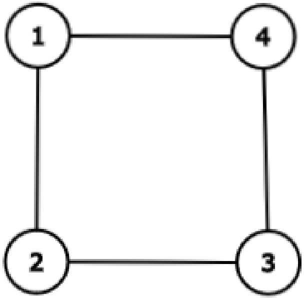


Figure: 2-regular graph example

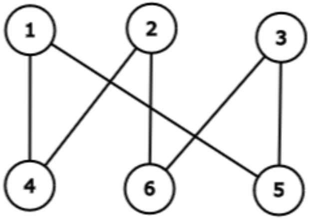


Figure: Bipartite graph example

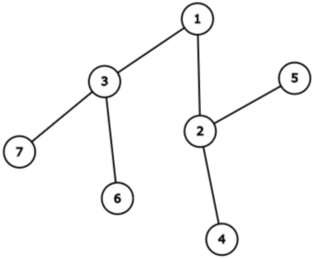


Figure: Tree example

Graph Representation, Adjacency Matrix

Matrix Representation

- Store Boolean, adjacent or not?
- Store integer, weight.

```
vector<vector<bool>> adjacency_matrix = {{},
     {false, true, false, true},
     {true, false, true, false},
     {false, true, false, true},
     {true, false, true, false}};
// You can determine a distinct value for
    unconnected
vector<vector<int>> adjacency_matrix_w = {{},
     {-1, 2, -1, 8},
     {2, -1, 4, -1},
     {-1, 4, -1, 6},
     {8, -1, 6, -1}};
```

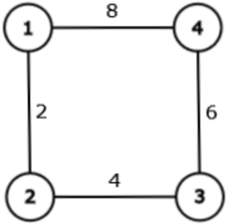


Figure: The graph represented in the code.

Graph Representation, Adjacency List

Matrix Representation

- Store adjacent nodes' numbers.
- Store pair<int, int>, weight.

```
vector<vector<int>> adjacency_list = {{},
    {2, 4},
    {1, 3},
    {2, 4},
    {1, 3}};
vector<vector<pair<int, int>>> adjacency_list_w =
    {{},
    {{2,2}, {4,8}},
    {{1,2}, {3,4}},
    {{2,4}, {4,6}},
    {{1,8}, {3,6}}};
```

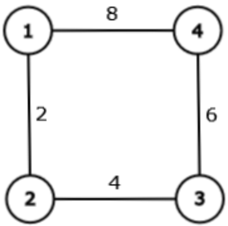


Figure: The graph represented in the code.

Graph Exploration Methods

- Two ways to explore a graph, **BFS** and **DFS**.
- Both do the same task using different methods.
- Requires a **starting node**, a **hash map** and adequate data structure (**queue or stack**).
- **BFS (Breadth First Search)**: Explore using a **queue**.
- **DFS (Depth First Search)**: Explore using a **stack**.
- Both have time complexity $\mathbf{O(V + E)}$ and space complexity $\mathbf{O(V)}$.
  - V: number of nodes, E: number of edges.

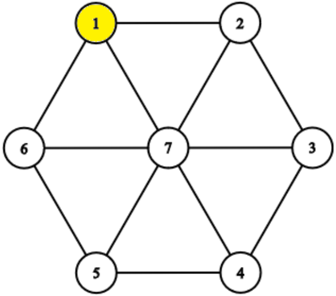
BFS (Breadth First Search)

- From a starting node, explore all nodes level by level.
- First explore nodes one step away, then two step away...
- This logic implies an usage of a queue.

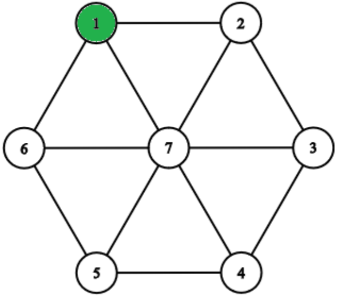
Its algorithm is basically:

1. Push starting node to queue and mark it as used.
2. Take node **V** at the front of the queue. If the queue is empty, terminate.
3. Push the unvisited nodes adjacent to **V** into the queue and mark them as used.
4. Write down **V** and pop it from the queue.
5. Jump to step 2.

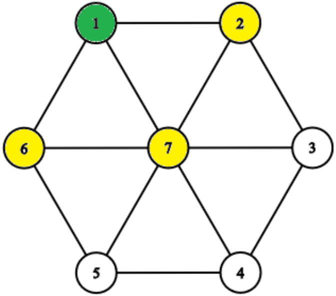
BFS Example Part 1



Current node	-
Queue	1
Used nodes	1
BFS	-

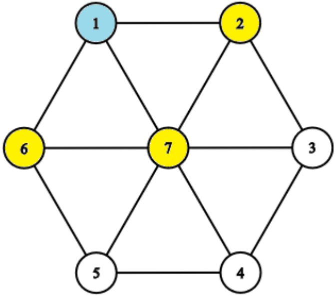


Current node	1
Queue	1
Used nodes	1
BFS	-

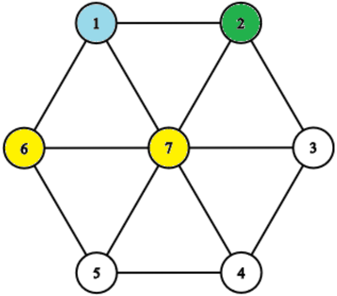


Current node	1
Queue	1 2 7 6
Used nodes	1 2 6 7
BFS	-

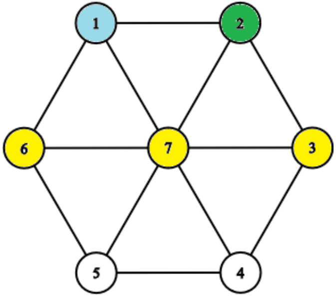
BFS Example Part 2



Current node	-
Queue	2 7 6
Used nodes	1 2 6 7
BFS	1

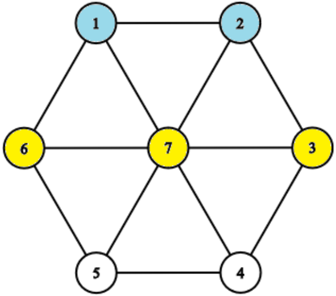


Current node	2
Queue	2 7 6
Used nodes	1 2 6 7
BFS	1

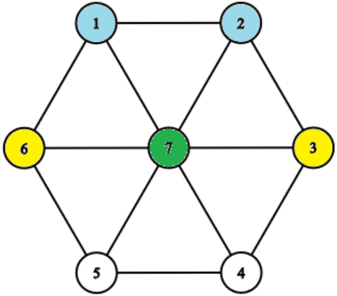


Current node	2
Queue	2 7 6 3
Used nodes	1 2 3 6 7
BFS	1

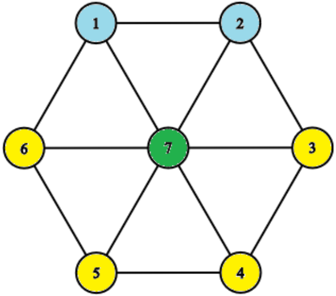
BFS Example Part 3



Current node	-
Queue	7 6 3
Used nodes	1 2 3 6 7
BFS	1 2

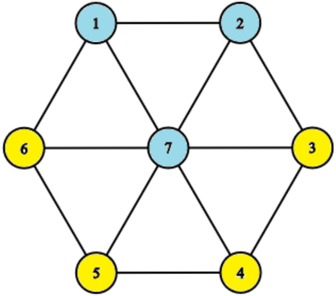


Current node	7
Queue	7 6 3
Used nodes	1 2 3 6 7
BFS	1 2

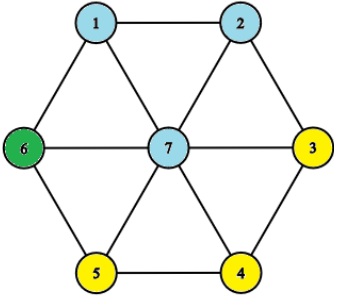


Current node	7
Queue	7 6 3 4 5
Used nodes	ALL
BFS	1 2

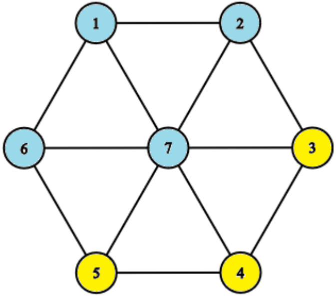
BFS Example Part 4



Current node	-
Queue	6 3 4 5
Used nodes	ALL
BFS	1 2 7

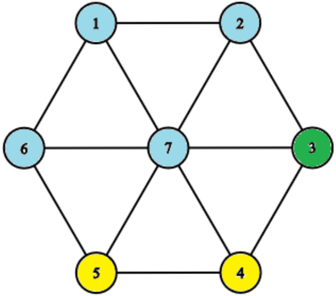


Current node	6
Queue	6 3 4 5
Used nodes	ALL
BFS	1 2 7

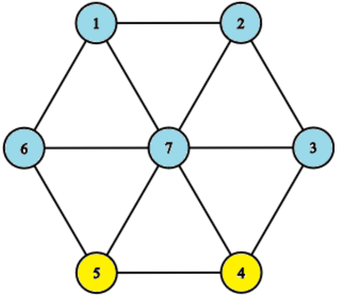


Current node	-
Queue	3 4 5
Used nodes	ALL
BFS	1 2 7 6

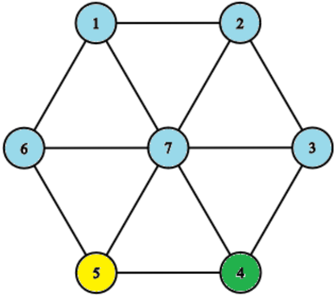
BFS Example Part 5



Current node	3
Queue	3 4 5
Used nodes	ALL
BFS	1 2 7 6

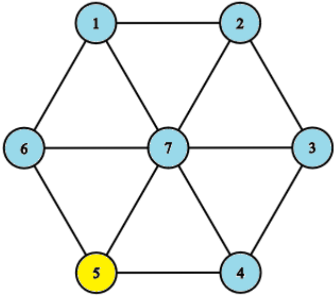


Current node	-
Queue	4 5
Used nodes	ALL
BFS	1 2 7 6 3

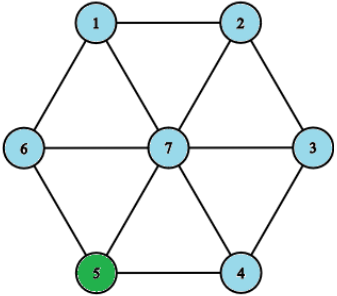


Current node	4
Queue	4 5
Used nodes	ALL
BFS	1 2 7 6 3

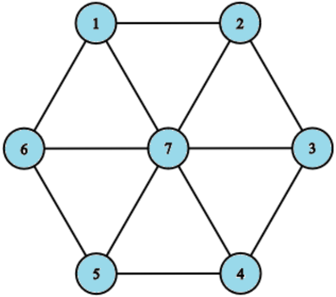
BFS Example Part 6



Current node	-
Queue	5
Used nodes	ALL
BFS	1 2 7 6 3 4



Current node	5
Queue	5
Used nodes	ALL
BFS	1 2 7 6 3 4



Current node	-
Queue	-
Used nodes	ALL

BFS complete: 1 2 7 6 3 4 5

BFS Code, Declerations

```cpp
#include <queue>
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

int main()
{
    vector<vector<int>> adjacency_list = {
        {},
        {2, 7, 6},
        {3, 7, 1},
        {4, 7, 2},
        {5, 7, 3},
        {6, 7, 4},
        {1, 7, 5},
        {1, 2, 3, 4, 5, 6}};
    unordered_map<int, bool> visited;
    queue<int> bfs;
```

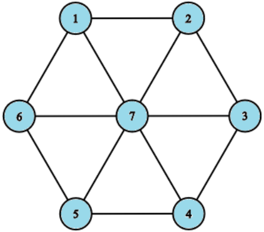


Figure: The graph used in the code

BFS Code, Operation

```
    bfs.push(1);
    visited[1] = true;
    while (!bfs.empty())
    {
        int current_node = bfs.front();
        for (int x : adjacency_list[current_node])
        {
            if (!visited[x])
            {
                bfs.push(x);
                visited[x] = true;
            }
        }
        bfs.pop();
        cout << current_node << " ";
    }
    cout << endl;
    return 0;
}
```

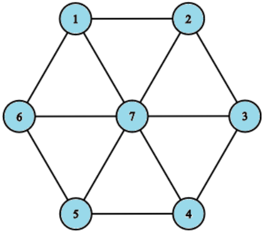


Figure: The graph used in the code

BFS with Saving Node Level

```
    unordered_map<int, bool> visited;
    queue<pair<int, int>> bfs; // First : node
    bfs.push(make_pair(1, 1)); // Second: level
    visited[1] = true;
    while (!bfs.empty())
    {
        int current_node = bfs.front().first;
        int current_level = bfs.front().second;
        for (int x : adjacency_list[current_node])
        {
            if (!visited[x])
            {
                bfs.push(make_pair(x, current_level + 1));
                visited[x] = true;
            }
        }
        bfs.pop();
        cout << current_node << " " << current_level << endl;
    }
```

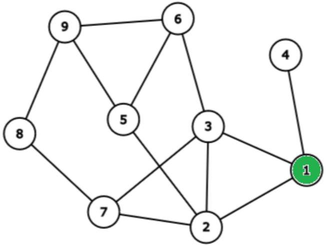
DFS (Depth First Search)

- From a starting node, explore to the furthest depth possible.
- Trackback to next unexplored branches until complete.
- This logic implies the usage of a stack.
- It is possible to implement DFS with iterative and recursive methods.
- Recursive method is easier to understand and implement.

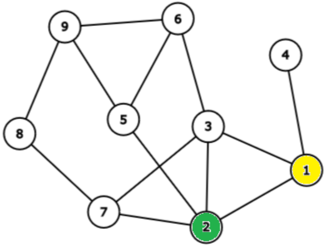
Its recursive algorithm is basically:

1. Call the algorithm for starting node.
2. Mark the current node as used.
3. Write down the current node.
4. Call the function for all unvisited adjacent nodes. Calls them recursively.

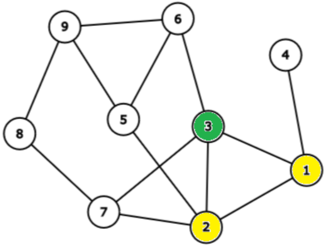
DFS Example Part 1



Current node	1
Call Stack	1
Used nodes	1
DFS	1

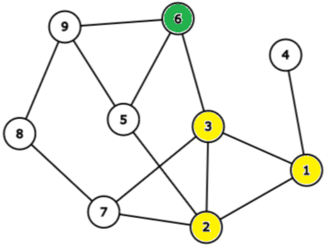


Current node	2
Call Stack	1 2
Used nodes	1 2
DFS	1 2

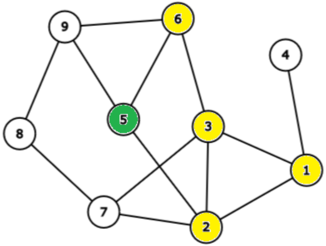


Current node	3
Call Stack	1 2 3
Used nodes	1 2 3
DFS	1 2 3

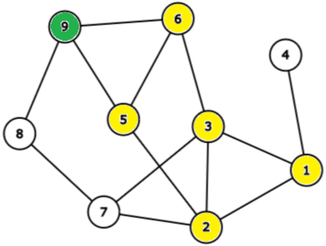
DFS Example Part 2



Current node	6
Call Stack	1 2 3 6
Used nodes	1 2 3 6
DFS	1 2 3 6

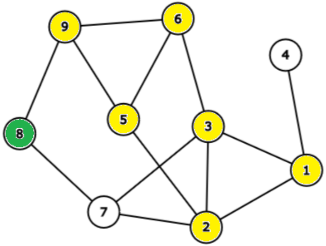


Current node	5
Call Stack	1 2 3 6 5
Used nodes	1 2 3 5 6
DFS	1 2 3 6 5

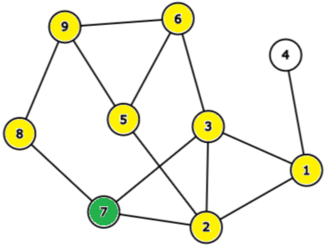


Current node	9
Call Stack	1 2 3 6 5 9
Used nodes	1 2 3 5 6 9
DFS	1 2 3 6 5 9

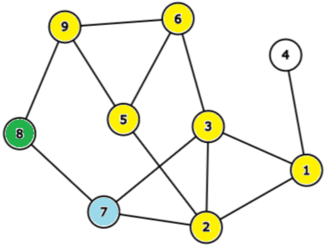
DFS Example Part 3



Current node	8
Call Stack	1 2 3 6 5 9 8
Used nodes	1 2 3 5 6 8 9
DFS	1 2 3 6 5 9 8

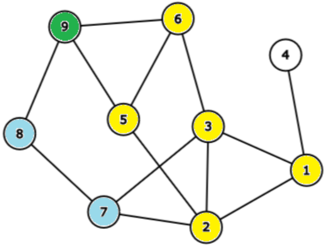


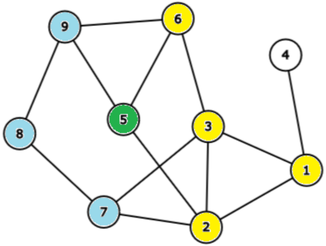
Current node	7
Call Stack	1 2 3 6 5 9 8 7
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

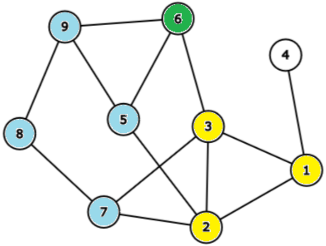


Current node	8
Call Stack	1 2 3 6 5 9 8
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

DFS Example Part 4





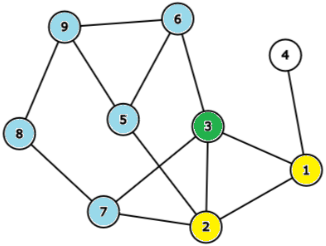


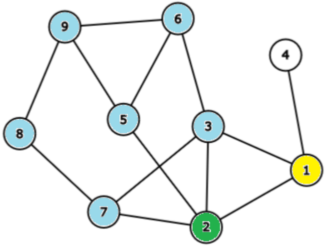
Current node	9
Call Stack	1 2 3 6 5 9
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

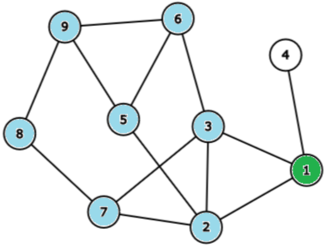
Current node	5
Call Stack	1 2 3 6 5
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

Current node	6
Call Stack	1 2 3 6
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

DFS Example Part 5





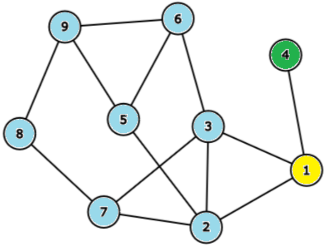


Current node	3
Call Stack	1 2 3
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

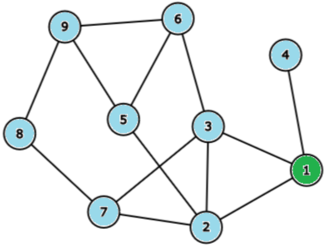
Current node	2
Call Stack	1 2
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

Current node	1
Call Stack	1
Used nodes	1 2 3 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7

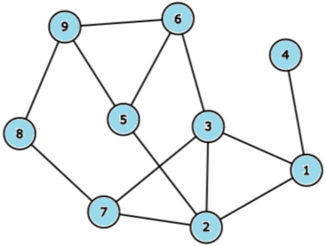
DFS Example Part 6



Current node	4
Call Stack	1 4
Used nodes	1 2 3 4 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7 4



Current node	1
Call Stack	1
Used nodes	1 2 3 4 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7 4



Current node	-
Call Stack	-
Used nodes	1 2 3 4 5 6 7 8 9
DFS	1 2 3 6 5 9 8 7 4

Recursive DFS Code, Declerations

```cpp
int main()
{
    vector<vector<int>> adjacency_list =
    {
        {}, {2, 3, 4},
        {1, 3, 5, 7}, {1, 2, 6, 7},
        {1}, {2, 6, 9},
        {3, 5, 9}, {2, 3, 8},
        {7, 9}, {5, 6, 8}
    }
    unordered_map<int, bool> visited;
    dfs(1, adjacency_list, visited);
    cout << endl;
    return 0;
}
```

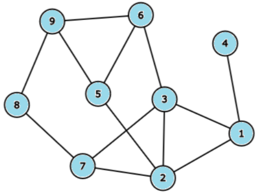


Figure: The graph used in the code

Recursive DFS Code, DFS Function

```cpp
void dfs(int current_node,
         vector<vector<int>>& adjacency_list,
         unordered_map<int, bool>& visited)
{
    visited[current_node] = true;
    cout << current_node << " ";
    for(int x : adjacency_list[current_node])
    {
        if(!visited[x])
            dfs(x, adjacency_list, visited);
    }
}
```

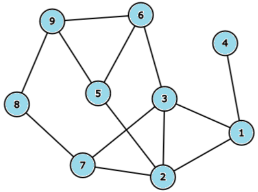


Figure: The graph used in the code

Iterative DFS Code, Declerations

```
int main()
{
    vector<vector<int>> adjacency_list =
    {
        {}, {2, 3, 4},
        {1, 3, 5, 7}, {1, 2, 6, 7},
        {1}, {2, 6, 9},
        {3, 5, 9}, {2, 3, 8},
        {7, 9}, {5, 6, 8}
    };
    unordered_map<int, bool> visited;
    stack<int> dfs;
    dfs.push(1);
```

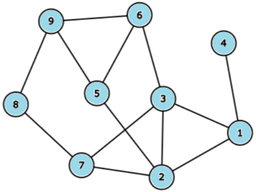


Figure: The graph used in the code

Iterative DFS Code, Operation

```
    while(!dfs.empty())
    {
        int current_node = dfs.top();
        dfs.pop();
        for(int x: adjacency_list[current_node])
        {
            if(!visited[x])
                dfs.push(x);
        }
        if(!visited[current_node])
        {
            visited[current_node] = true;
            cout << current_node << " ";
        }
    }
    cout << endl;
    return 0;
}
```

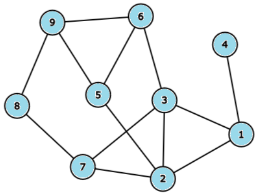


Figure: The graph used in the code

Comparison of BFS and DFS

	DFS	BFS
Exploration order	Depth	Level
Data structure	Stack	Queue
Time complexity	$O(V+E)$	$O(V+E)$
Space complexity	$O(V)$	$O(V)$
Exploration tree	Narrow and long	Wide and short

Table: Ref: opengenus.org

Topological Sort

- With using a DFS, entirely traverse a **directed uncyclic graph (DAG)**.
- Use a stack to save the traversed nodes in a stack.
- Write down stack from top to bottom to store:
  - For dependencies, install order,
  - For lectures requiring previously taken courses, order of lectures that you should take,
  - Order of tasks in task list in which certain tasks require previously completed tasks.
- This algorithm fails if the graph is not a **DAG**.
  - It is not the weakness of the algorithm, such graphs are impossible to topologically sort.
  - If A requires B and B requires A to be done, it is impossible to do both tasks.
- Time complexity: $\mathbf{O(V + E)}$
- Space complexity: $\mathbf{O(V)}$

DAG and Not DAG Graph Examples

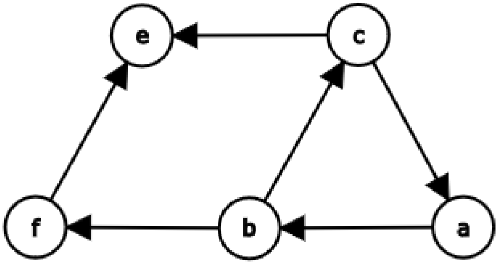


Figure: Not a DAG, not topologically sort-able.

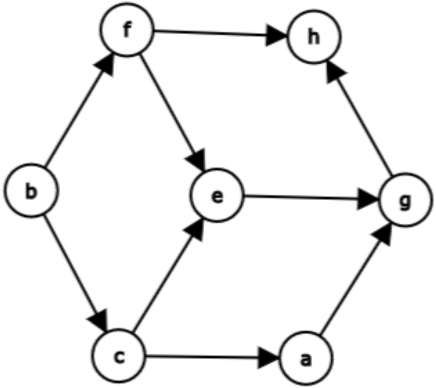


Figure: DAG, topologically sort-able

Topological Sort Example Part 1

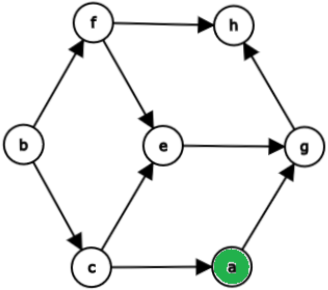


Figure: Topological Sort Stack: -empty-

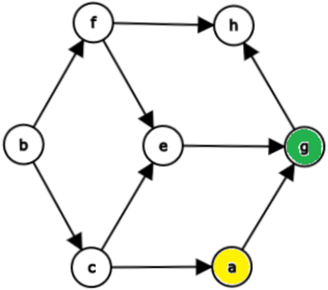


Figure: Topological Sort Stack: -empty-

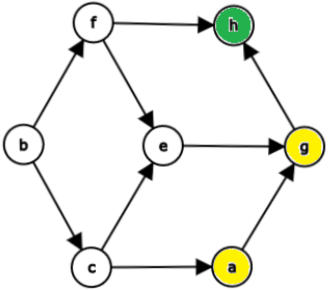


Figure: Topological Sort Stack: -empty-

Topological Sort Example Part 2

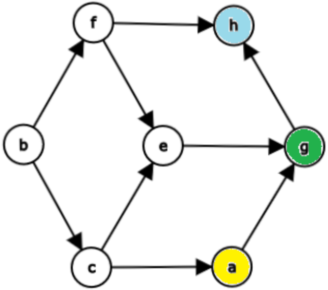


Figure: Topological Sort Stack: h

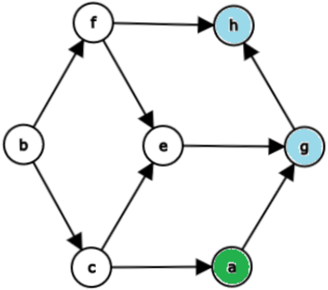


Figure: Topological Sort Stack: h-g

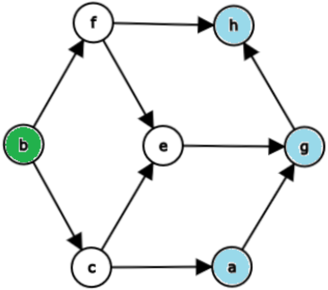


Figure: Topological Sort Stack: h-g-a

Topological Sort Example Part 3

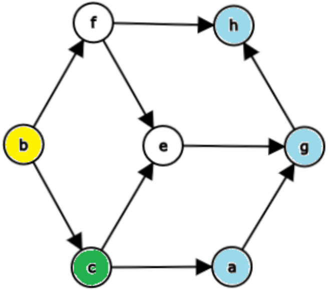


Figure: Topological Sort Stack: h-g-a

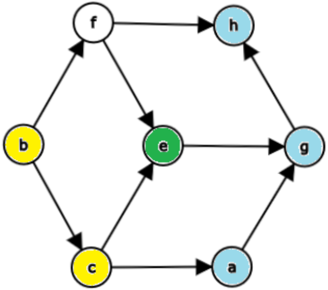


Figure: Topological Sort Stack: h-g-a

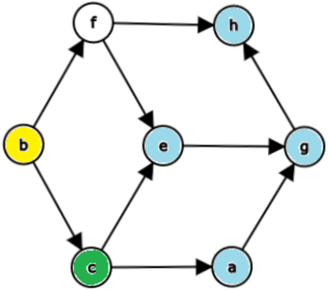


Figure: Topological Sort Stack: h-g-a-e

Topological Sort Example Part 4

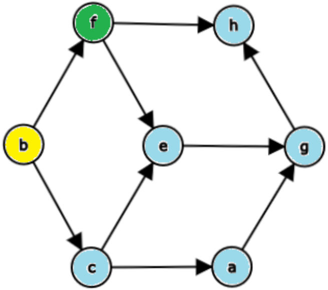


Figure: Topological Sort Stack: h-g-a-e-c

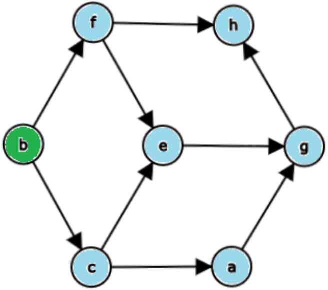


Figure: Topological Sort Stack: h-g-a-e-c-f

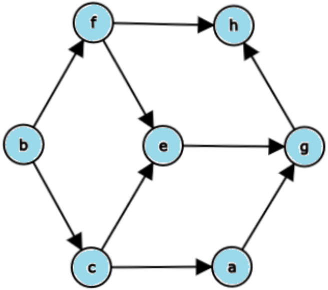


Figure: Topological Sort Stack: h-g-a-e-c-f-b

Topological Sort Recursive DFS Code

```
void topSort(int node, vector<vector<int>>& adj_list, unordered_map<int,
    bool>& visited, stack<int>& top_sort_stack)
{
    visited[node] = true;
    for(int x : adj_list[node])
    {
        if(!visited[x])
            topSort(x, adj_list, visited, top_sort_stack);
    }
    top_sort_stack.push(node);
    return;
}
```

Topological Sort Function Call and Printing Stack

```
for(int i = 1; i < adj_list.size(); i++)
{
    if(!visited[i])
        topSort(i, adj_list, visited, top_sort_stack);
}
while(!top_sort_stack.empty())
{
    cout << top_sort_stack.top() << " ";
    top_sort_stack.pop();
}
cout << endl;
```

Tree Traversals

- Trees are called complete m-ary if nodes have either **0** or **m** child nodes.
- **Height** of a tree is the longest path from the **root** of the tree and its **leaves.**
- Three ways to traverse a complete binary tree.

- Preorder traversal
- value, left, right

- Inorder traversal
- left, value, right

- Postorder traversal
- left, right, value

- Traversals are used to notate a tree.
- Using recursive functions to traverse.
- Complexity is $\mathbf{O(n + m)}$ but for trees, $\mathbf{m = n - 1}$ so complexity is $\mathbf{O(n)}$.
- Space complexity is $\mathbf{O(1)}$.
- **Note:** Postorder traversal is also known as **reverse Polish notation**.

Tree Struct Code

Struct:

```
struct TreeNode
{
    int value;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : value(x), left(nullptr), right(nullptr) {}
};
```

Declaration:

```
    TreeNode *root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);
```

Preorder Traversal Example Part 1

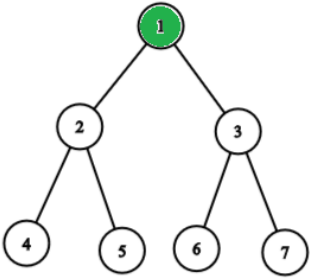


Figure: Preorder Traversal: -empty-

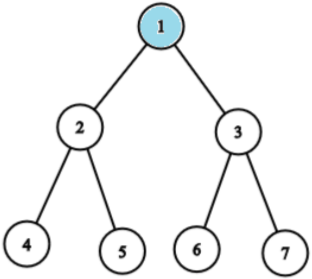


Figure: Preorder Traversal: 1

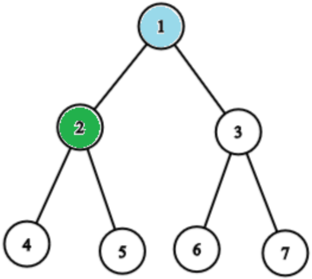


Figure: Preorder Traversal: 1

Preorder Traversal Example Part 2

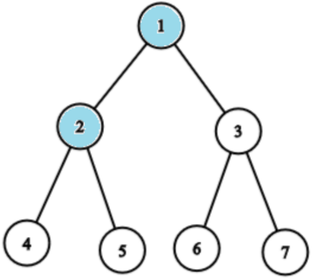


Figure: Preorder Traversal: 1-2

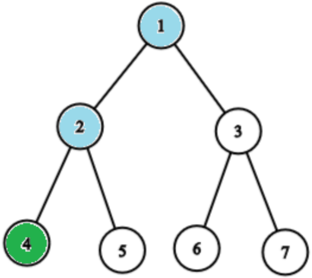


Figure: Preorder Traversal: 1-2

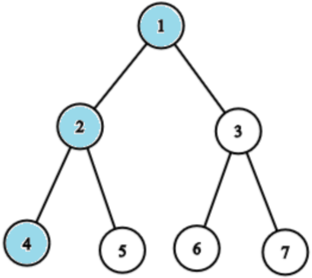


Figure: Preorder Traversal: 1-2-4

Preorder Traversal Example Part 3

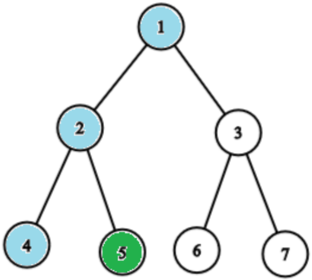


Figure: Preorder Traversal: 1-2-4

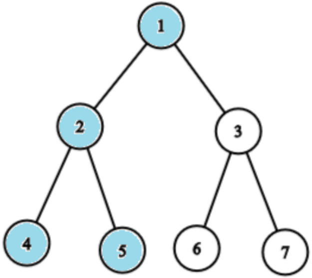


Figure: Preorder Traversal: 1-2-4-5

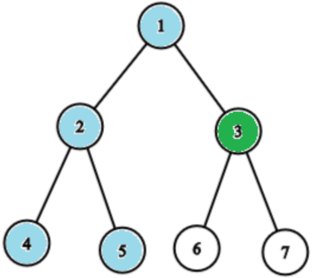


Figure: Preorder Traversal: 1-2-4-5

Preorder Traversal Example Part 4

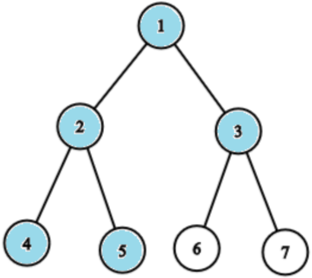


Figure: Preorder Traversal: 1-2-4-5-3

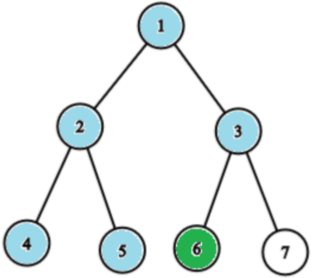


Figure: Preorder Traversal: 1-2-4-5-3

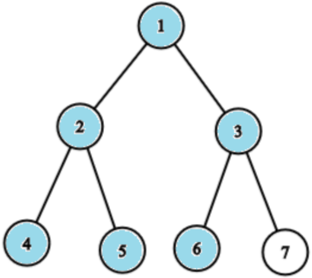


Figure: Preorder Traversal: 1-2-4-5-3-6

Preorder Traversal Example Part 5

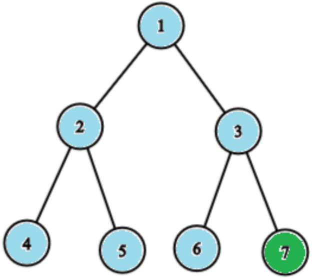


Figure: Preorder Traversal: 1-2-4-5-3-6

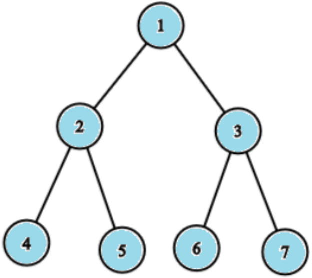


Figure: Preorder Traversal: 1-2-4-5-3-6-7

Preorder Traversal Code

Recursive function:

```cpp
void preorderTraversal(TreeNode *root)
{
    if (root == nullptr)
    {
        return;
    }
    cout << root->value << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

Function call:

```cpp
    // preorder traversal
    preorderTraversal(root);
    cout << endl;
```

Inorder Traversal Example Part 1

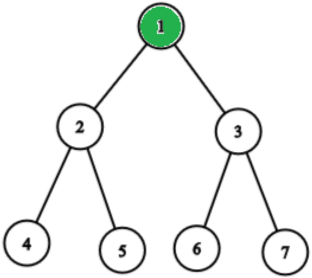


Figure: Preorder Traversal:
-empty-

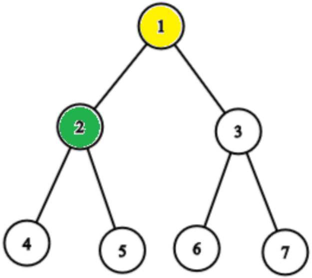


Figure: Inorder Traversal:
-empty-

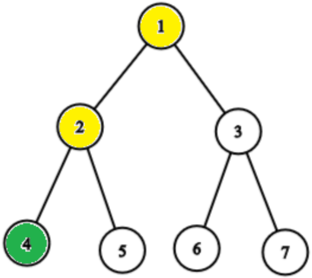


Figure: Inorder Traversal:
-empty-

Inorder Traversal Example Part 2

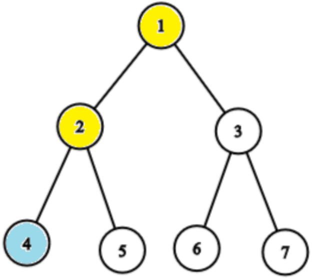


Figure: Inorder Traversal: 4

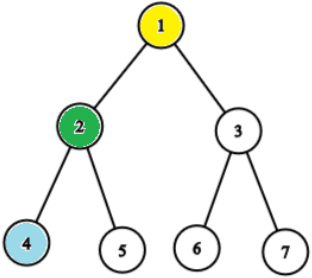


Figure: Inorder Traversal: 4

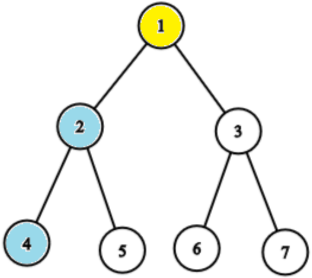


Figure: Inorder Traversal: 4-2

Inorder Traversal Example Part 3

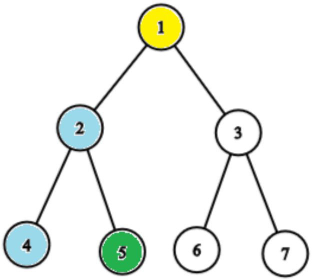


Figure: Inorder Traversal: 4-2

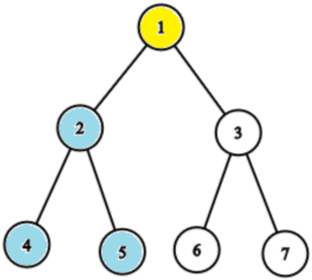


Figure: Inorder Traversal: 4-2-5

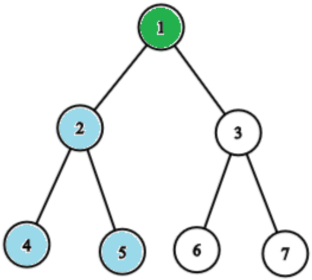


Figure: Inorder Traversal: 4-2-5

Inorder Traversal Example Part 4

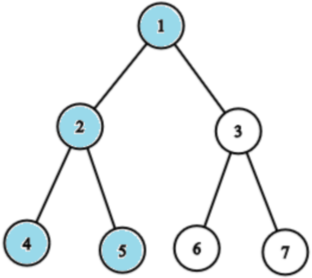


Figure: Inorder Traversal: 4-2-5-1

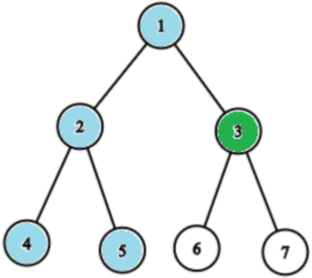


Figure: Inorder Traversal: 4-2-5-1

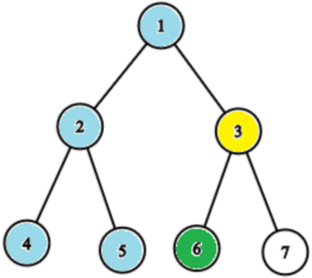


Figure: Inorder Traversal: 4-2-5-1

Inorder Traversal Example Part 5

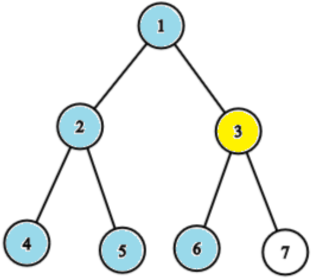


Figure: Inorder Traversal: 4-2-5-1-6

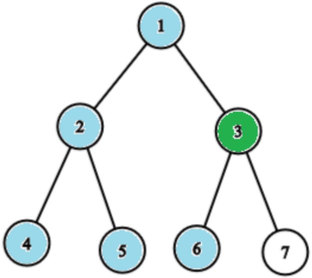


Figure: Inorder Traversal: 4-2-5-1-6

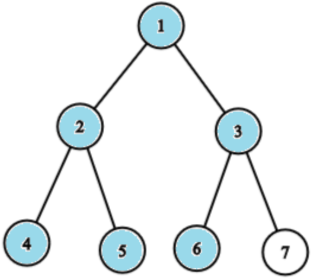


Figure: Inorder Traversal: 4-2-5-1-6-3

Inorder Traversal Example Part 6

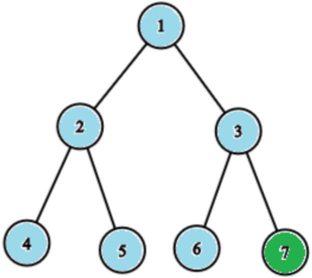


Figure: Inorder Traversal:
4-2-5-1-6-3

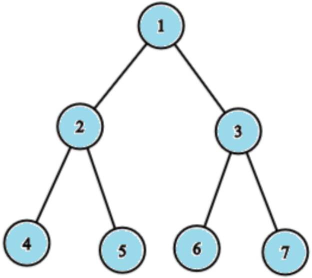


Figure: Inorder Traversal:
4-2-5-1-6-3-7

Inorder Traversal Code

Recursive function:

```
void inorderTraversal(TreeNode *root)
{
    if (root == nullptr)
    {
        return;
    }
    inorderTraversal(root->left);
    cout << root->value << " ";
    inorderTraversal(root->right);
}
```

Function call:

```
    // inorder traversal
    inorderTraversal(root);
    cout << endl;
```

Postorder Traversal Example Part 1

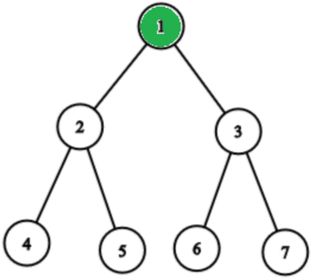


Figure: Postorder Traversal:
-empty-

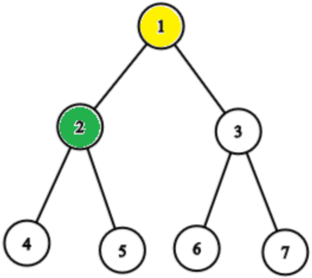


Figure: Postorder Traversal:
-empty-

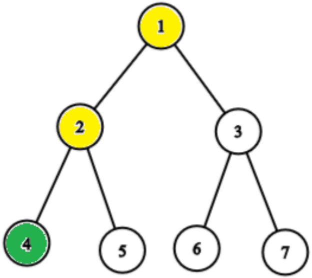


Figure: Postorder Traversal:
-empty-

Postorder Traversal Example Part 2

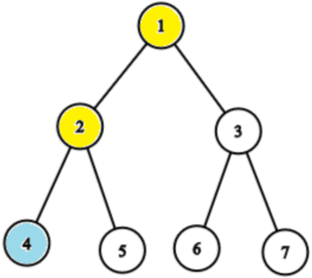


Figure: Postorder Traversal: 4

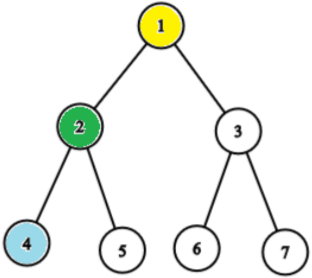


Figure: Postorder Traversal: 4

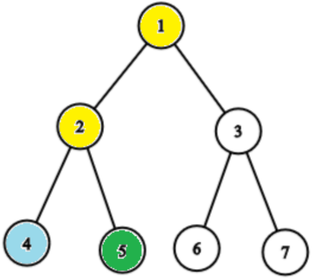


Figure: Postorder Traversal: 4

Postorder Traversal Example Part 3

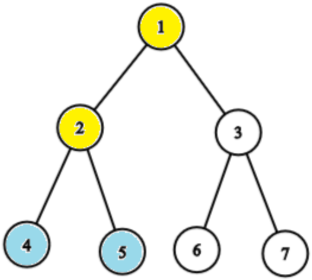


Figure: Postorder Traversal: 4-5

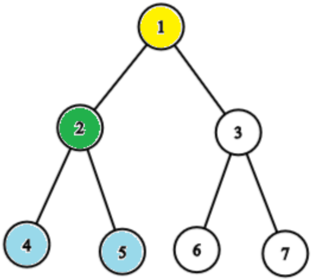


Figure: Postorder Traversal: 4-5

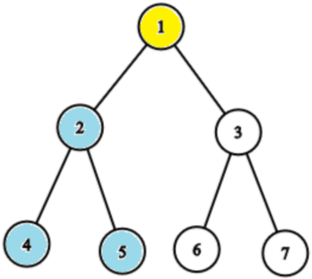


Figure: Postorder Traversal: 4-5-2

Postorder Traversal Example Part 4

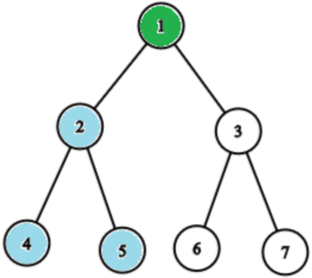


Figure: Postorder Traversal: 4-5-2

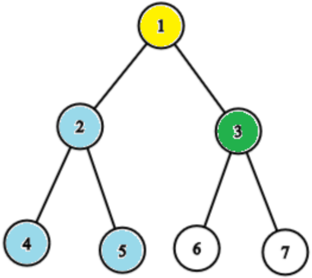


Figure: Postorder Traversal: 4-5-2

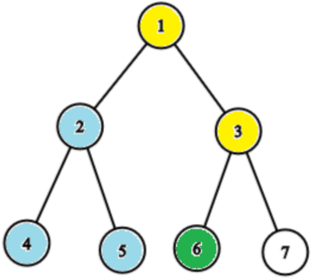


Figure: Postorder Traversal: 4-5-2

Postorder Traversal Example Part 5

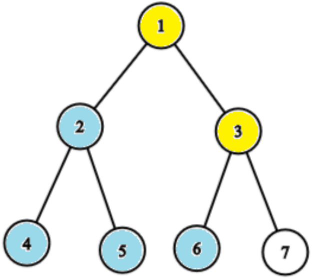


Figure: Postorder Traversal: 4-5-2-6

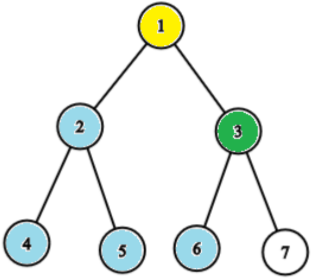


Figure: Postorder Traversal: 4-5-2-6

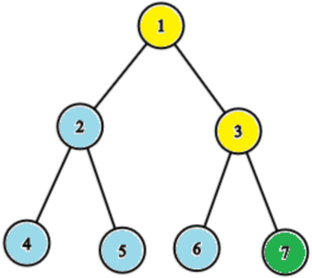


Figure: Postorder Traversal: 4-5-2-6

Postorder Traversal Example Part 6

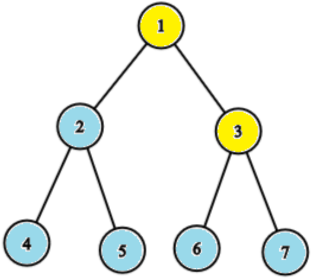


Figure: Postorder Traversal: 4-5-2-6-7

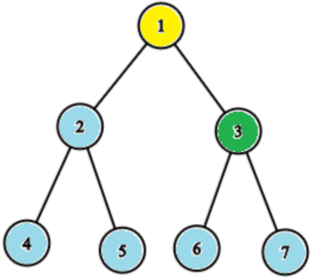


Figure: Postorder Traversal: 4-5-2-6-7

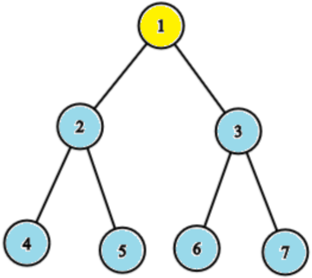


Figure: Postorder Traversal: 4-5-2-6-7-3

Postorder Traversal Example Part 7

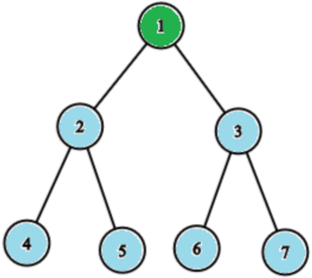


Figure: Postorder Traversal:
4-5-2-6-7-3

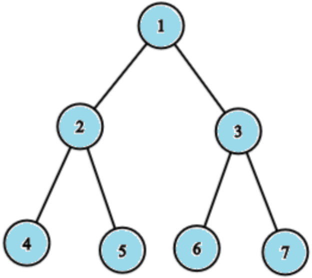


Figure: Postorder Traversal:
4-5-2-6-7-3-1

Postorder Traversal Code

Recursive function:

```
void postorderTraversal(TreeNode *root)
{
    if (root == nullptr)
    {
        return;
    }
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->value << " ";
}
```

Function call:

```
    // postorder traversal
    postorderTraversal(root);
    cout << endl;
```

Binary Search Tree

- In the form of the binary tree data structure.
- Values of left sub-tree $<$ node's value.
- Values of right sub-tree $>$ node's value.
- Left-most value is the smallest.
- Right-most value is the greatest.
- If traversed inorder, you get a sorted array.
- **Balanced BST**: Depth of the leaves differ at most by 1.
- It is possible to construct a BST from any traversal of the BST or from an array.
- In a BST, you can find an item in **O(logn)** complexity.

Binary Search Tree Examples

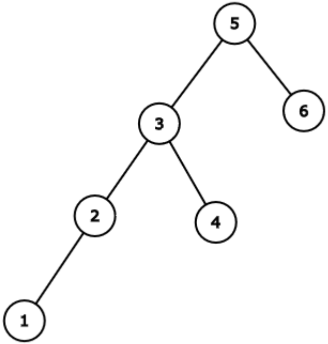


Figure: Unbalanced BST example

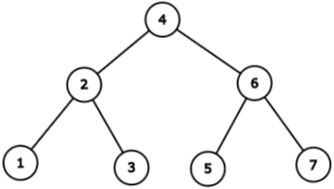


Figure: Balanced BST example

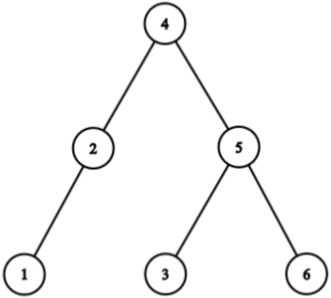


Figure: Invalid BST example

Constructing a BST from Scratch Example

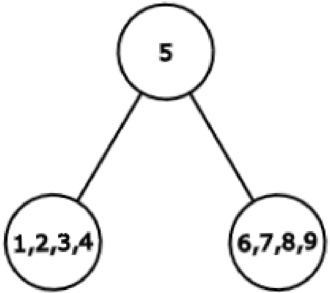


Figure: Midpoint(s): 5

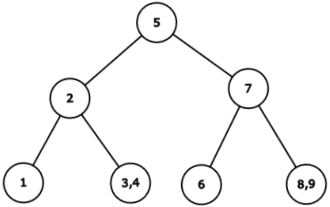


Figure: Midpoint(s): 2, 7

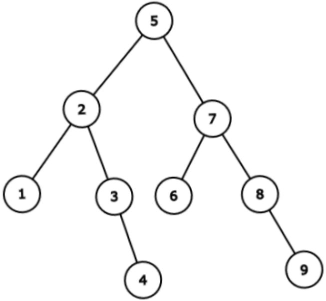


Figure: Balanced BST constructed.

Balanced BST Construction Code, Tree Struct and Function Call

```
struct TreeNode
{
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
     right(right) {}
};
vector<int> numbers = {4, 1, 2, 6, 3, 9, 7, 8, 5};
    sort(numbers.begin(), numbers.end());
    TreeNode* balancedBST = recursiveBSTgen(numbers, 0, numbers.size() -
    1);
```

Balanced BST Construction Code, Recursive Construction Code

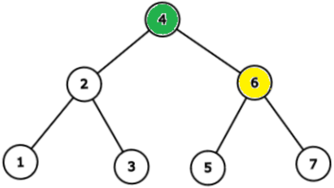
```cpp
TreeNode* recursiveBSTgen(vector<int>& numbers, int left, int right)
{
    // When there is no element to place
    if(left > right)
    {
        return NULL;
    }
    int middle = (left + right) / 2;
    TreeNode* root = new TreeNode(numbers[middle]);
    // left and right branches
    root->left = recursiveBSTgen(numbers, left, middle - 1);
    root->right = recursiveBSTgen(numbers, middle + 1, right);
    return root;
}
```

BST Search Algorithm

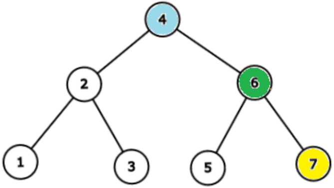
- One of the benefits of BST is the ease of searching an item.
- Rather than searching linearly in an array, search using divide & conquer.
- **O(logn)** complexity in a tree, rather than **O(n)** in an array.
- There exists dynamically configurable BSTs but they are out of today's scope.
- Those are called red & black trees.

1. If current node pointer is a null pointer return false.
2. If current value is equal to required, return level or true & false.
3. If required value is bigger than current value, go right. (Go back to step 1)
4. If required value is smaller than current value, go left. (Go back to step 1)

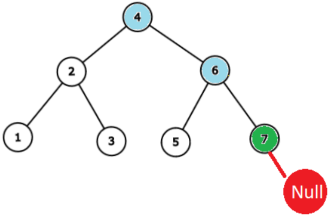
BST Search Example



Target	9
Value	4
Action	Go Right



Target	9
Value	6
Action	Go Right



Target	9
Value	7
Action	Go Right

After going right, function will encounter a null pointer therefore it will return either an **invalid level** or **false**.

BST Search Code

```
int BSTsearch(TreeNode* node, int target, int previous_level)
{
    if(node == NULL)
        return -1; // Invalid level or false,
    if(node->val == target)
        return previous_level + 1; // Current level or true,
    if(node->val > target)
        return BSTsearch(node->left, target, previous_level + 1);
    if(node->val < target)
        return BSTsearch(node->right, target, previous_level + 1);
}
```

Example Questions

- Path Sum II *(DFS & Tree & Recursion)*
- Number of Provinces *(DFS)*
- Course Schedule *(Topological Sort)*
- Convert Sorted Array to BST *(Binary Search Tree)*
- Validate Binary Search Tree *(BST & Tree Traversal)*