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**.
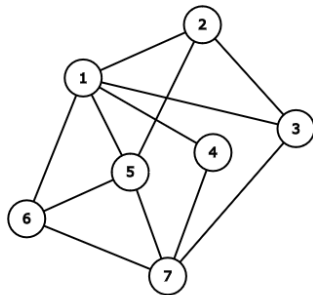

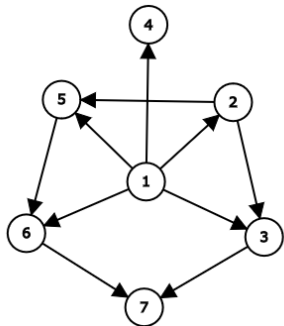Figure: A plain graph example

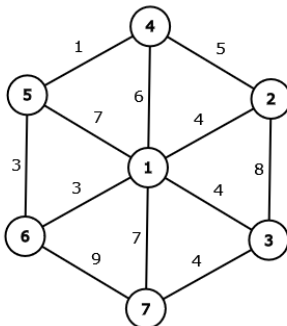# Graphs, Examples

Figure: Directed graph example
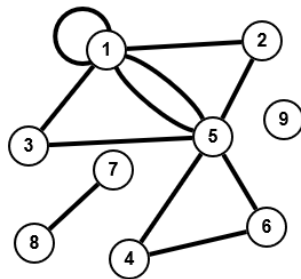
Figure: Weighted graph example

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.
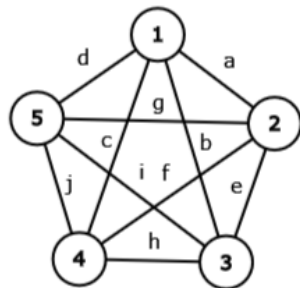- **Hamilton graphs** contains a closed spanning path.
  - ref: inzva, 2018

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.
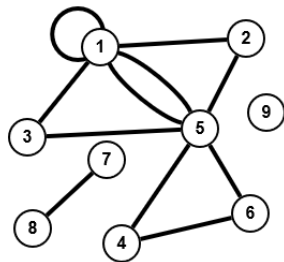- Largest distance in graph is called **diameter** of graph.

Figure: Disconnected graph example

Special Graphs

- **Completely connected graph**s 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.
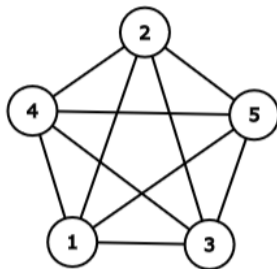
Figure: Completely connected graph example, K_5

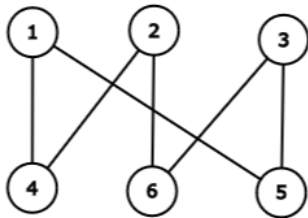# Special Graphs, Examples
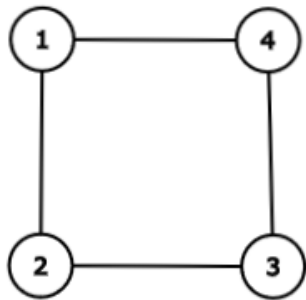
Figure: 2-regular graph example

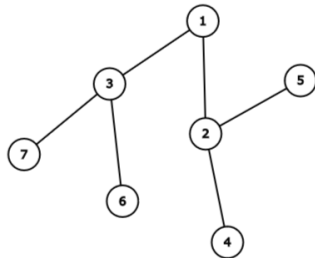Figure: Bipartite graph example

Figure: Tree example

Graph Representation, Adjacency Matrix

Matrix Representation

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

```cpp
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}};
```
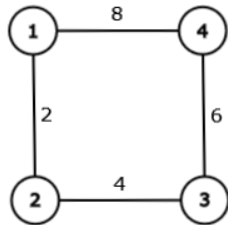

Figure: The graph represented in the code.

Graph Representation, Adjacency List

Matrix Representation

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

```cpp
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}}};
```
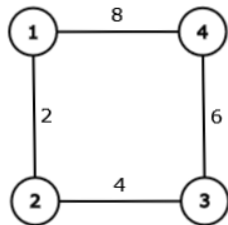

Figure: The graph represented in the code.

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 **O(V + E)** and space complexity **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
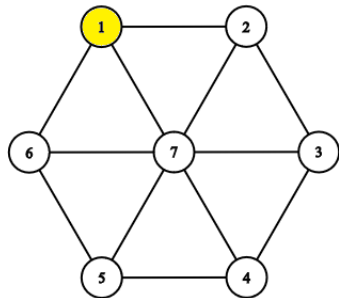
Current node	-
Queue	1
Used nodes	1
BFS	-
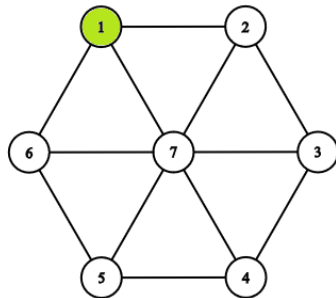
Current node	1
Queue	1
Used nodes	1
BFS	-
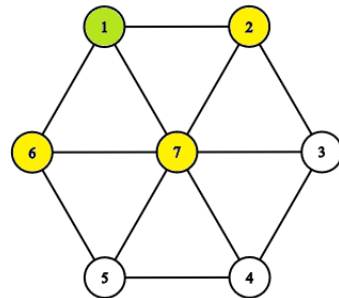
Current node	1
Queue	1 2 7 6
Used nodes	1 2 6 7
BFS	-

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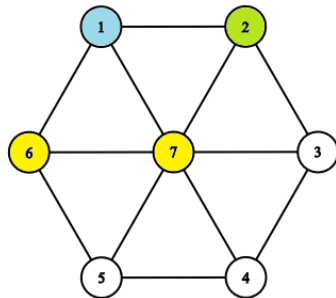
Current node	-		Current node	2		Current node	2	
Queue	2 7 6		Queue	2 7 6		Queue	2 7 6 3	
Used nodes	1 2 6 7		Used nodes	1 2 6 7		Used nodes	1 2 3 6 7	
BFS	1		BFS	1		BFS	1	

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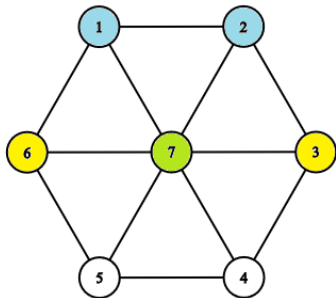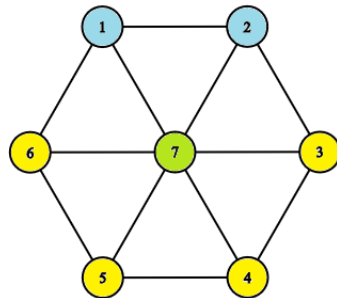
Current node	-	Current node	7	Current node	7
Queue	7 6 3	Queue	7 6 3	Queue	7 6 3 4 5
Used nodes	1 2 3 6 7	Used nodes	1 2 3 6 7	Used nodes	ALL
BFS	1 2	BFS	1 2	BFS	1 2

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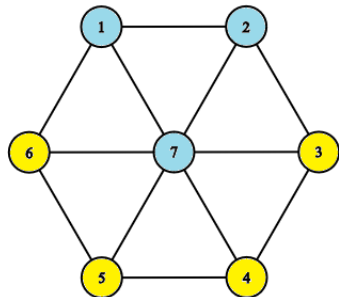
Current node	-
Queue	6 3 4 5
Used nodes	ALL
BFS	1 2 7
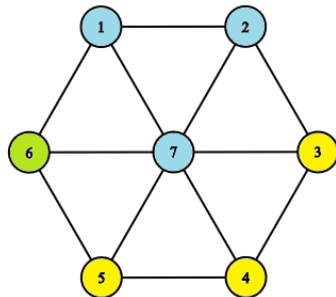
Current node	6
Queue	6 3 4 5
Used nodes	ALL
BFS	1 2 7
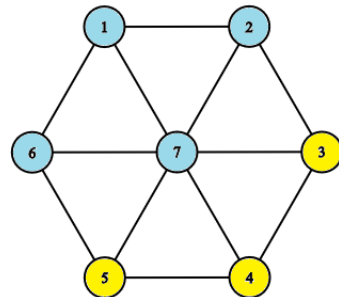
Current node	-
Queue	3 4 5
Used nodes	ALL
BFS	1 2 7 6

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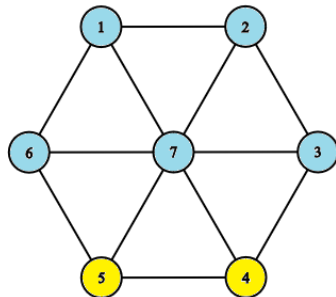
Current node	3	Current node	-	Current node	4
Queue	3 4 5	Queue	4 5	Queue	4 5
Used nodes	ALL	Used nodes	ALL	Used nodes	ALL
BFS	1 2 7 6	BFS	1 2 7 6 3	BFS	1 2 7 6 3

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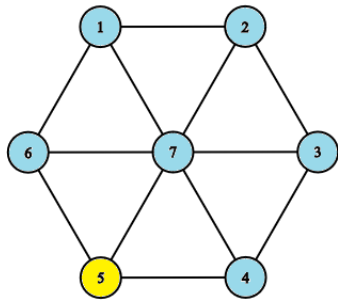
Current node	-
Queue	5
Used nodes	ALL
BFS	1 2 7 6 3 4
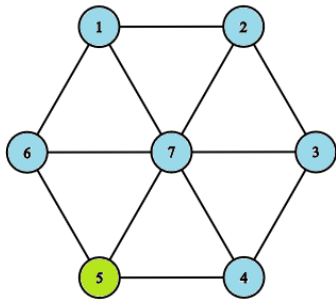
Current node	5
Queue	5
Used nodes	ALL
BFS	1 2 7 6 3 4
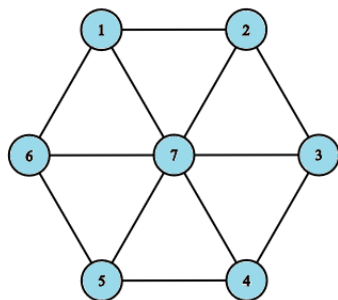
Current node	-
Queue	-
Used nodes	ALL

BFS complete: 1 2 7 6 3 4 5

BFS Code, Declerations

```cpp
1  #include <queue>
2  #include <iostream>
3  #include <vector>
4  #include <unordered_map>
5  using namespace std;
6
7  int main()
8  {
9      vector<vector<int>> adjacency_list = {
10         {},
11         {2, 7, 6},
12         {3, 7, 1},
13         {4, 7, 2},
14         {5, 7, 3},
15         {6, 7, 4},
16         {1, 7, 5},
17         {1, 2, 3, 4, 5, 6}};
18     unordered_map<int, bool> visited;
19     queue<int> bfs;
```
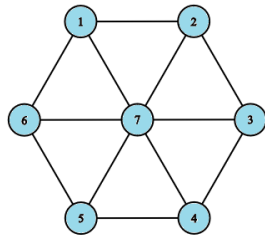

Figure: The graph used in the code

BFS Code, Operation

```
1    bfs.push(1);
2    visited[1] = true;
3    while (!bfs.empty())
4    {
5        int current_node = bfs.front();
6        for (int x : adjacency_list[current_node])
7        {
8            if (!visited[x])
9            {
10               bfs.push(x);
11               visited[x] = true;
12           }
13       }
14       bfs.pop();
15       cout << current_node << " ";
16   }
17   cout << endl;
18   return 0;
19 }
```
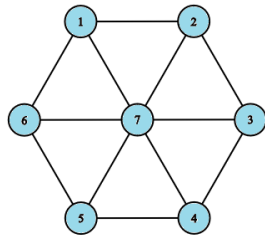

Figure: The graph used in the code

BFS with Saving Node Level

```
1    unordered_map<int, bool> visited;
2    queue<pair<int, int>> bfs; // First : node
3    bfs.push(make_pair(1, 1)); // Second: level
4    visited[1] = true;
5    while (!bfs.empty())
6    {
7        int current_node = bfs.front().first;
8        int current_level = bfs.front().second;
9        for (int x : adjacency_list[current_node])
10       {
11           if (!visited[x])
12           {
13               bfs.push(make_pair(x, current_level + 1));
14               visited[x] = true;
15           }
16       }
17       bfs.pop();
18       cout << current_node << " " << current_level << endl;
19   }
```

DFS (Depth First Search)

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

Its algorithm is basically:

1. Push starting node to stack.
2. Save the top node and pop it from the stack. If the stack is empty, terminate.
3. Push unvisited adjacent nodes to stack.
4. If the current node is not visited, write it down and mark it as visited.
5. Jump to step 2.

DFS Example Part 1
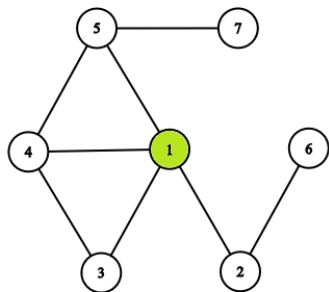
Current node	1
Stack	-
Used nodes	1
DFS	-
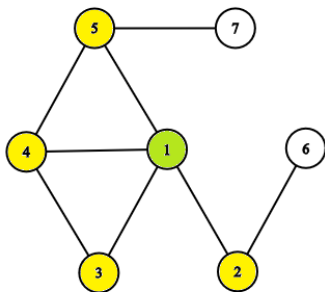
Current node	1
Stack	2 3 4 5
Used nodes	1
DFS	-
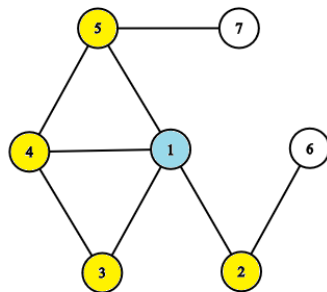
Current node	-
Stack	2 3 4 5
Used nodes	1
DFS	1

DFS Example Part 2
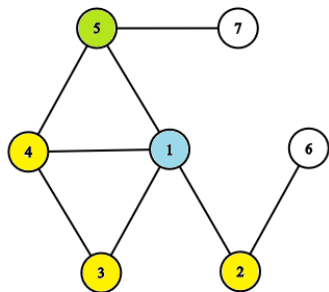
Current node	5
Stack	2 3 4
Used nodes	1 5
DFS	1
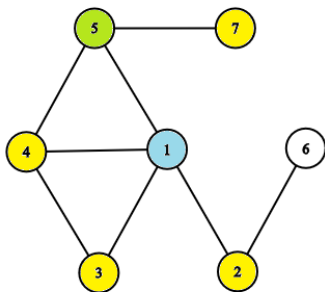
Current node	5
Stack	2 3 4 4 7
Used nodes	1 5
DFS	1
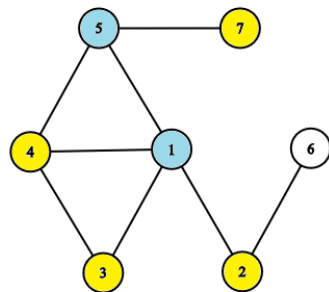
Current node	-
Stack	2 3 4 4 7
Used nodes	1 5
DFS	1 5

DFS Example Part 3
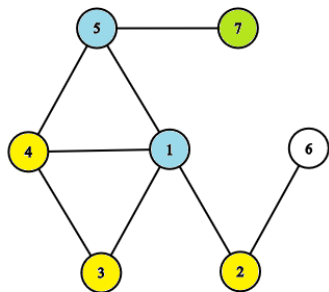
Current node	7
Stack	2 3 4 4
Used nodes	1 5 7
DFS	1 5
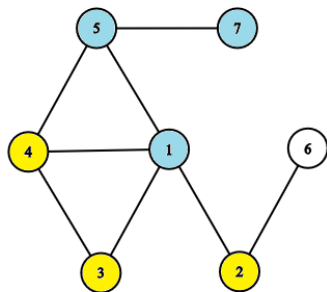
Current node	-
Stack	2 3 4 4
Used nodes	1 5 7
DFS	1 5 7
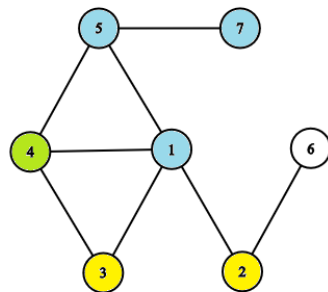
Current node	4
Stack	2 3 4
Used nodes	1 4 5 7
DFS	1 5 7

DFS Example Part 4
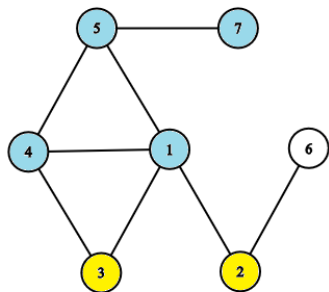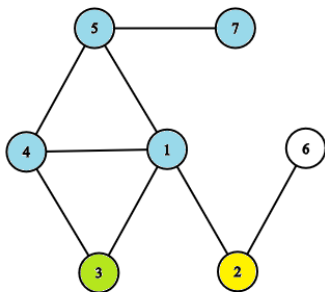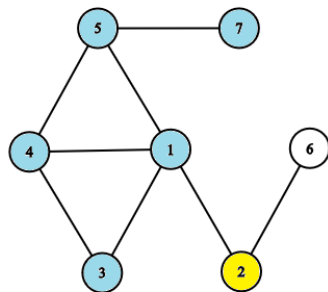
Current node	-		Current node	3		Current node	-
Stack	2 3 4 3		Stack	2 3 4		Stack	2 3 4
Used nodes	1 4 5 7		Used nodes	1 3 4 5 7		Used nodes	1 3 4 5 7
DFS	1 5 7 4		DFS	1 5 7 4		DFS	1 5 7 4 3

DFS Example Part 5
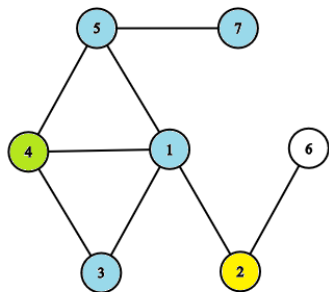
Current node	4
Stack	2 3
Used nodes	1 3 4 5 7
DFS	1 5 7 4 3
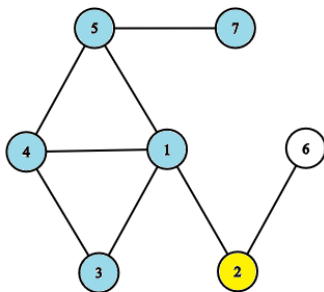
Current node	-
Stack	2 3
Used nodes	1 3 4 5 7
DFS	1 5 7 4 3
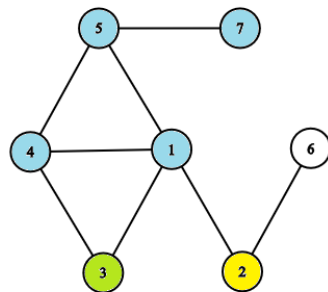
Current node	3
Stack	2
Used nodes	1 3 4 5 7
DFS	1 5 7 4 3

DFS Example Part 6
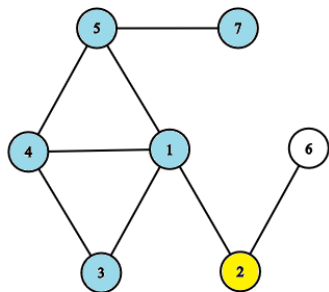
Current node	-
Stack	2
Used nodes	1 3 4 5 7
DFS	1 5 7 4 3
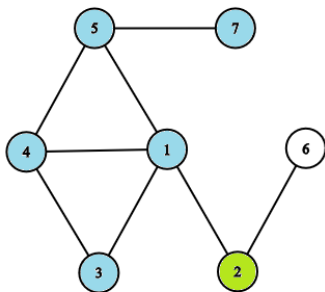
Current node	2
Stack	-
Used nodes	1 2 3 4 5 7
DFS	1 5 7 4 3
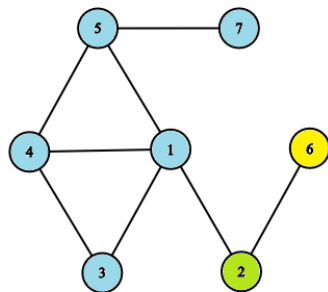
Current node	2
Stack	6
Used nodes	1 2 3 4 5 7
DFS	1 5 7 4 3

DFS Example Part 7
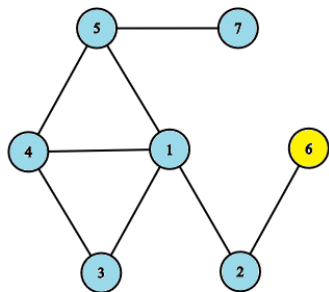
Current node	-
Stack	6
Used nodes	1 2 3 4 5 7
DFS	1 5 7 4 3 2
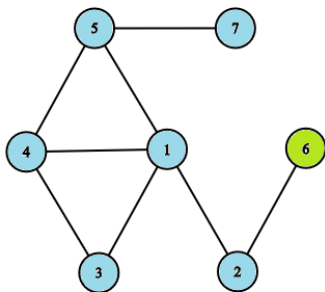
Current node	6
Stack	-
Used nodes	ALL
DFS	1 5 7 4 3 2
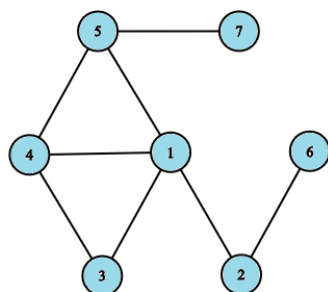
Current node	-
Stack	-
Used nodes	All

DFS complete: 1 5 7 4 3 2 6

Iterative DFS Code, Declerations

```cpp
1  #include <stack>
2  #include <iostream>
3  #include <vector>
4  #include <unordered_map>
5  using namespace std;
6
7  int main()
8  {
9      vector<vector<int>> adjacency_list = {
10         {},
11         {2, 3, 4, 5},
12         {1, 6},
13         {1, 4},
14         {1, 3, 5},
15         {1, 4, 7},
16         {2},
17         {5}};
18     unordered_map<int, bool> visited;
19     stack<int> dfs;
```
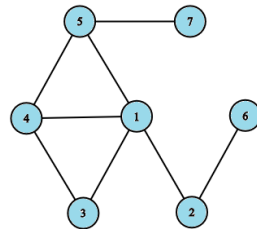

Figure: The graph used in the code

Iterative DFS Code, Operation

```
1       dfs.push(1);
2       while(!dfs.empty())
3       {
4           int current_node = dfs.top();
5           dfs.pop();
6           for(int x: adjacency_list[current_node])
7           {
8               if(!visited[x])
9                   dfs.push(x);
10          }
11          if(!visited[current_node])
12          {
13              visited[current_node] = true;
14              cout << current_node << " ";
15          }
16      }
17      cout << endl;
18      return 0;
19 }
```
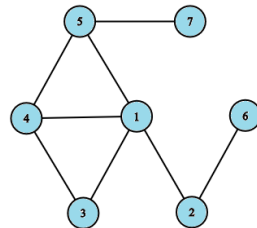

Figure: The graph used in the code

Recursive DFS Code, Declerations

```
1  int main()
2  {
3      vector<vector<int>> adjacency_list = {
4          {},
5          {2, 3, 4, 5},
6          {1, 6},
7          {1, 4},
8          {1, 3, 5},
9          {1, 4, 7},
10         {2},
11         {5}};
12     unordered_map<int, bool> visited;
13     dfs(1, adjacency_list, visited);
14     cout << endl;
15     return 0;
16 }
```
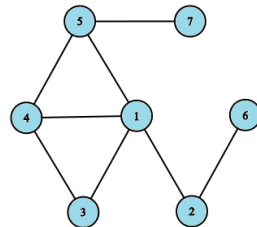

Figure: The graph used in the code

Recursive DFS Code, DFS Function

```cpp
1  void dfs(int current_node,
2          vector<vector<int>>& adjacency_list,
3          unordered_map<int, bool>& visited)
4  {
5      visited[current_node] = true;
6      cout << current_node << " ";
7      for(int x : adjacency_list[current_node])
8      {
9          if(!visited[x])
10             dfs(x, adjacency_list, visited);
11     }
12 }
```


Figure: The graph used in the code

- Note that recursive DFS would give correct but different answer compared to itterative DFS.
- 1 2 6 3 4 5 7
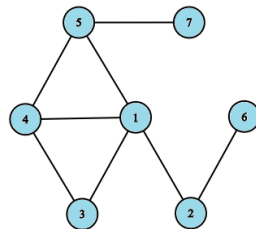
Comparison of BFS and DFS

	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: **O(V + E)**
- Space complexity: **O(V)**

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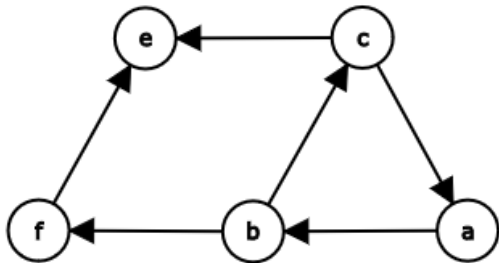
Figure: Not a DAG, not topologically sort-able.
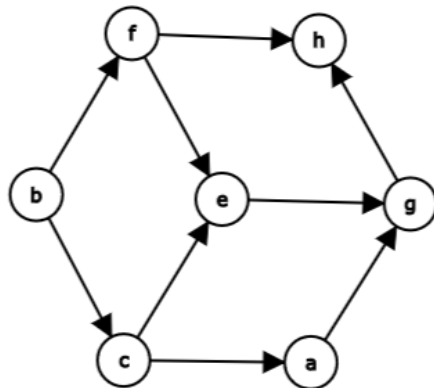
Figure: DAG, topologically sort-able

Topological Sort Example Part 1
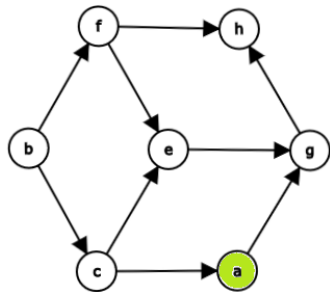
Figure: Topological Sort Stack: -empty-
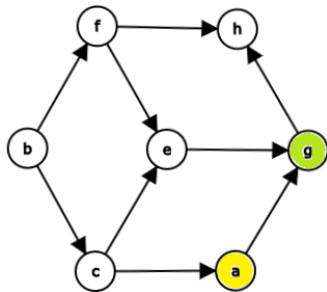
Figure: Topological Sort Stack: -empty-
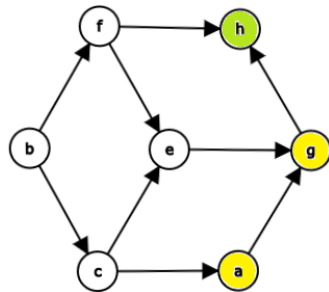
Figure: Topological Sort Stack: -empty-

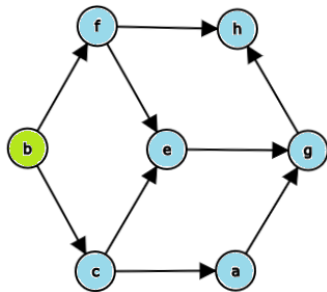# Topological Sort Example Part 2
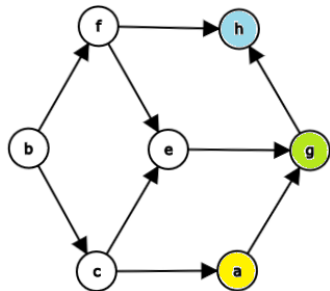
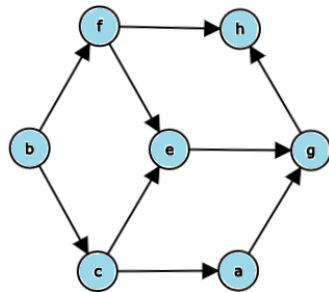Figure: Topological Sort Stack: h
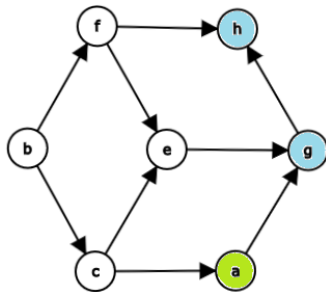
Figure: Topological Sort Stack: h-g
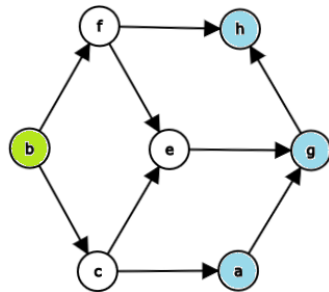
Figure: Topological Sort Stack: h-g-a

Topological Sort Example Part 3
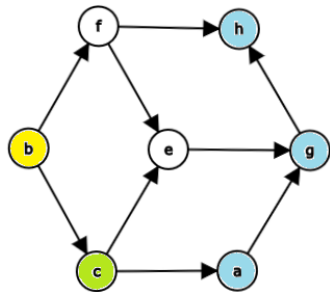
Figure: Topological Sort Stack: h-g-a
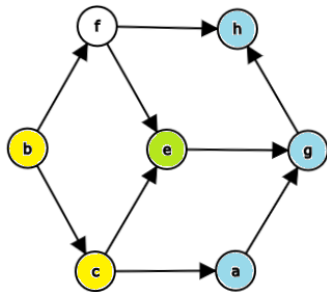
Figure: Topological Sort Stack: h-g-a
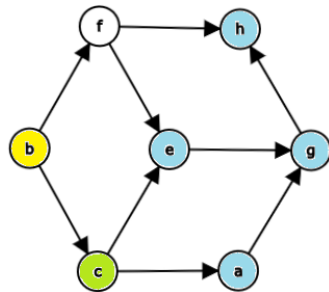
Figure: Topological Sort Stack: h-g-a-e

Topological Sort Example Part 4
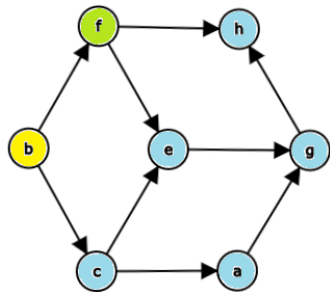
Figure: Topological Sort Stack: h-g-a-e-c

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

```
1  void topSort(int node, vector<vector<int>>& adj_list, unordered_map<int,
       bool>& visited, stack<int>& top_sort_stack)
2  {
3      visited[node] = true;
4      for(int x : adj_list[node])
5      {
6          if(!visited[x])
7              topSort(x, adj_list, visited, top_sort_stack);
8      }
9      top_sort_stack.push(node);
10     return;
11 }
```

Topological Sort Function Call and Printing Stack

```cpp
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	- Inorder traversal	- Postorder traversal
- value, left, right	- left, value, right	- left, right, value

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

Tree Struct Code

Struct:

```cpp
1 struct TreeNode
2 {
3     int value;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode(int x) : value(x), left(nullptr), right(nullptr) {}
7 };
```

Declaration:

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

Preorder Traversal Example Part 1
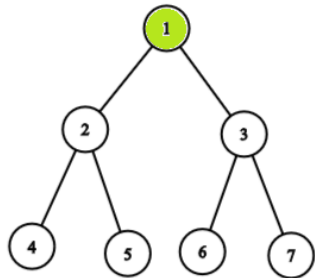
Figure: Preorder Traversal: -empty-
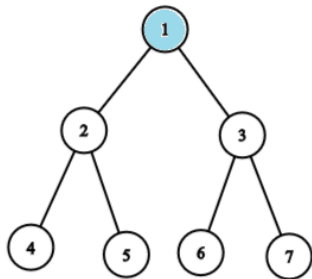
Figure: Preorder Traversal: 1
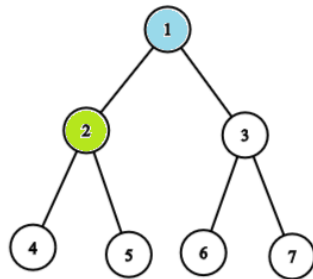
Figure: Preorder Traversal: 1

Preorder Traversal Example Part 2
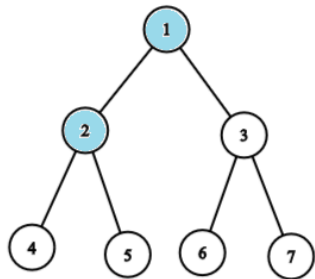
Figure: Preorder Traversal: 1-2
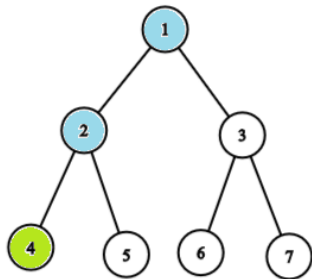
Figure: Preorder Traversal: 1-2
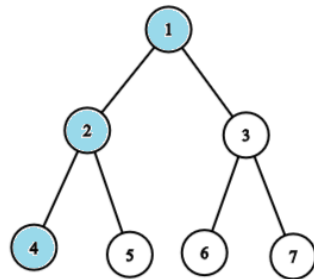
Figure: Preorder Traversal: 1-2-4

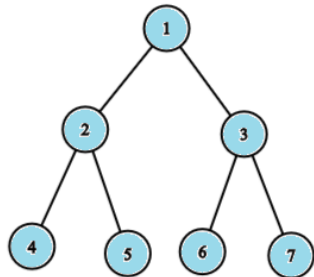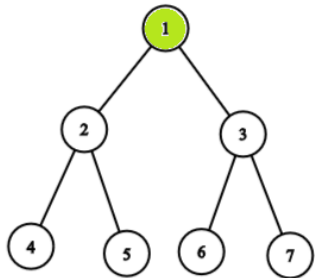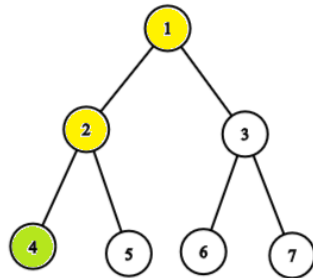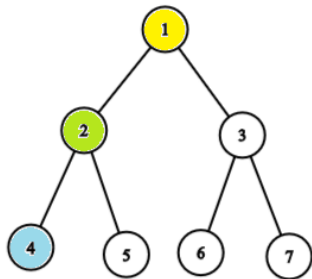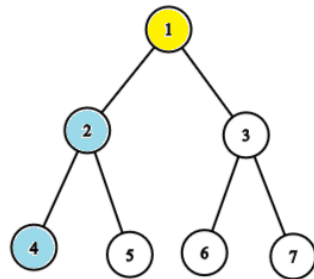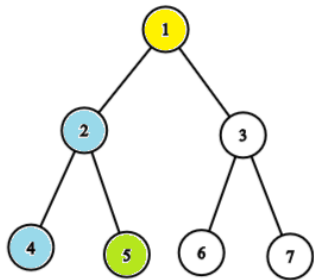# Preorder Traversal Example Part 3
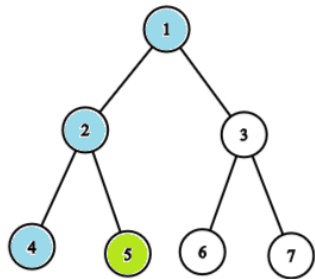
Figure: Preorder Traversal: 1-2-4
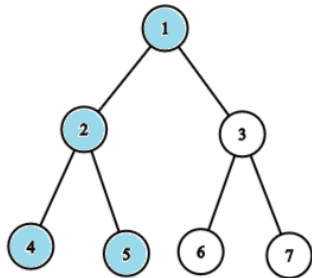
Figure: Preorder Traversal: 1-2-4-5
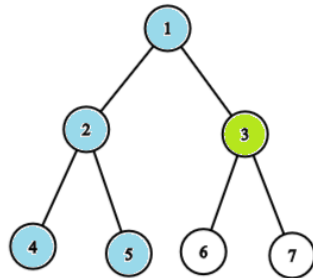
Figure: Preorder Traversal: 1-2-4-5

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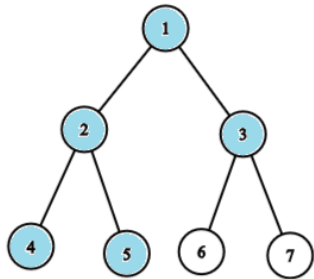
Figure: Preorder Traversal: 1-2-4-5-3
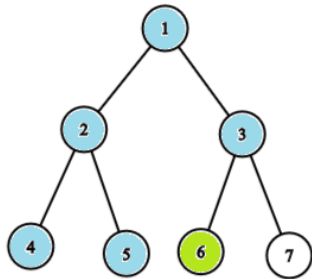
Figure: Preorder Traversal: 1-2-4-5-3
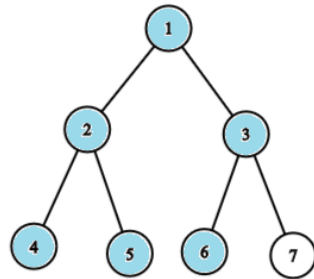
Figure: Preorder Traversal: 1-2-4-5-3-6

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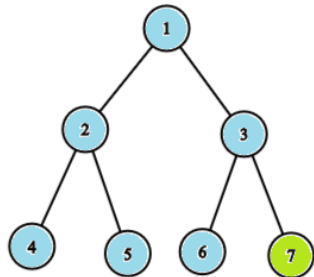
Figure: Preorder Traversal:
1-2-4-5-3-6

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

Preorder Traversal Code

Recursive function:

```cpp
1 void preorderTraversal(TreeNode *root)
2 {
3     if (root == nullptr)
4     {
5         return;
6     }
7     cout << root->value << " ";
8     preorderTraversal(root->left);
9     preorderTraversal(root->right);
10 }
```

Function call:

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

Inorder Traversal Example Part 1

Figure: Preorder Traversal: -empty-
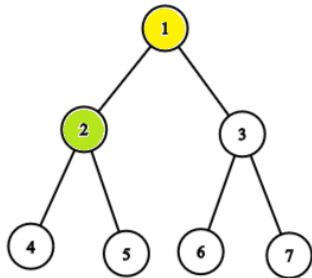
Figure: Inorder Traversal: -empty-

Figure: Inorder Traversal: -empty-

Inorder Traversal Example Part 2
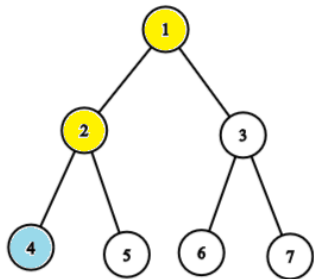
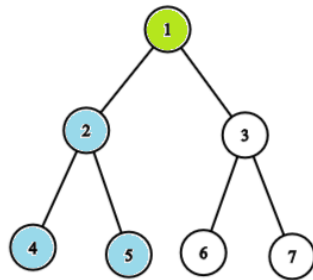Figure: Inorder Traversal: 4

Figure: Inorder Traversal: 4

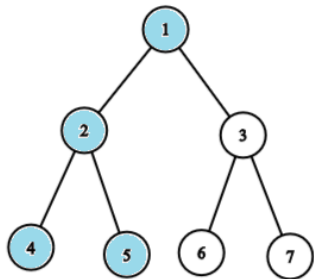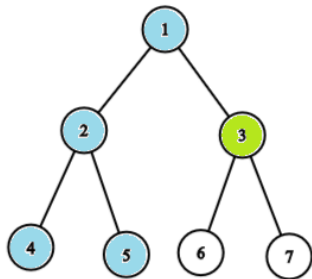Figure: Inorder Traversal: 4-2

Inorder Traversal Example Part 3

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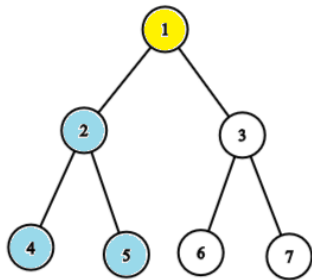
Figure: Inorder Traversal: 4-2-5

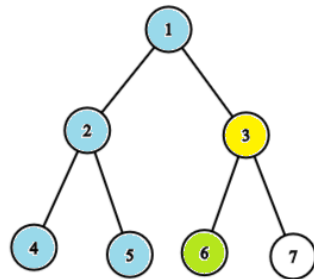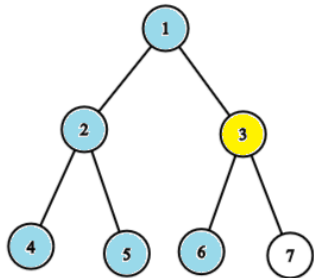Figure: Inorder Traversal: 4-2-5

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

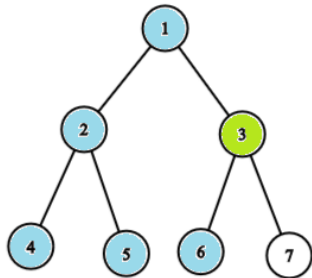# Inorder Traversal Example Part 5

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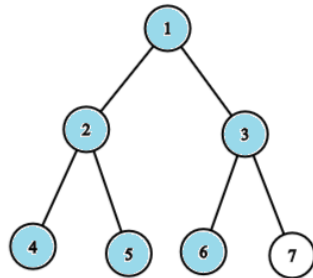4-2-5-1-6

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

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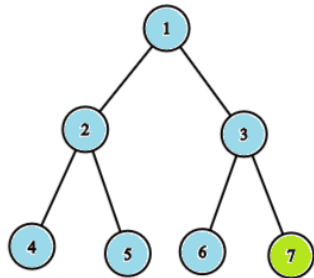4-2-5-1-6-3

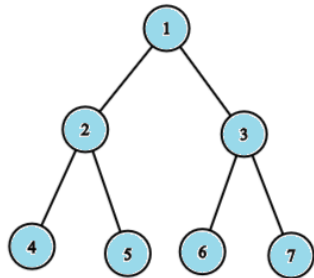# Inorder Traversal Example Part 6

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

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

Inorder Traversal Code

Recursive function:

```cpp
1  void inorderTraversal(TreeNode *root)
2  {
3      if (root == nullptr)
4      {
5          return;
6      }
7      inorderTraversal(root->left);
8      cout << root->value << " ";
9      inorderTraversal(root->right);
10 }
```

Function call:

```cpp
1      // inorder traversal
2      inorderTraversal(root);
3      cout << endl;
```
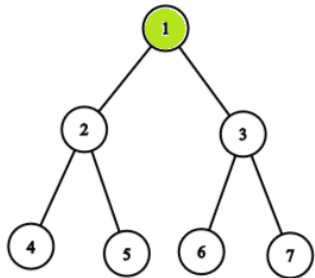
Postorder Traversal Example Part 1

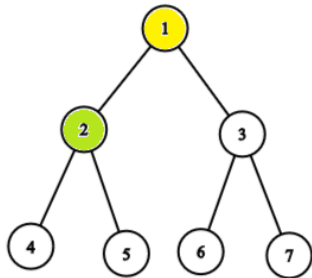Figure: Postorder Traversal: -empty-

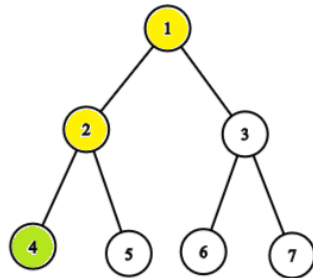Figure: Postorder Traversal: -empty-

Figure: Postorder Traversal: -empty-
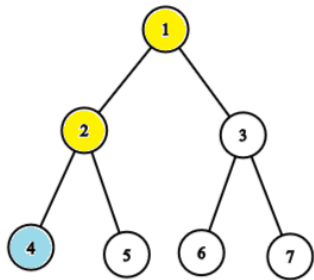
Postorder Traversal Example Part 2
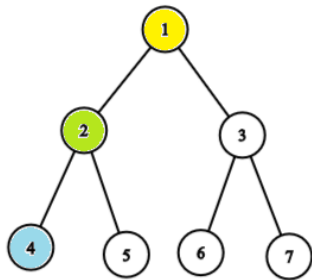

Figure: Postorder Traversal: 4
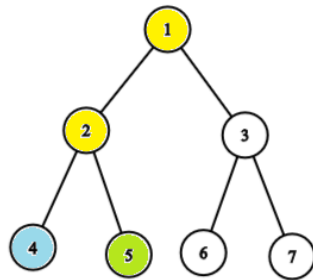
Figure: Postorder Traversal: 4

Figure: Postorder Traversal: 4
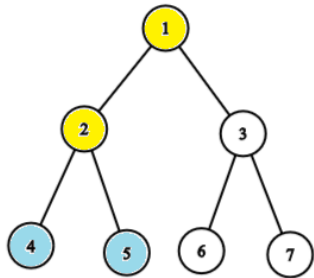
Postorder Traversal Example Part 3

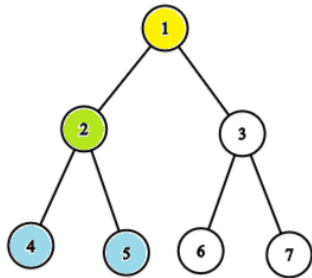Figure: Postorder Traversal: 4-5
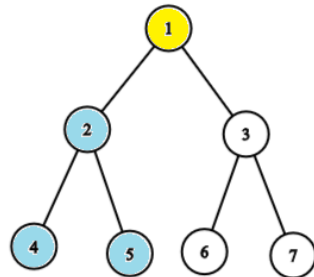
Figure: Postorder Traversal: 4-5

Figure: Postorder Traversal: 4-5-2
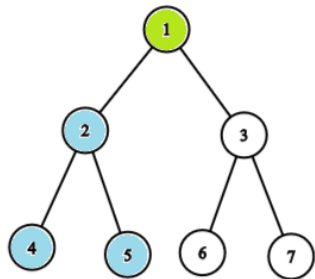
Postorder Traversal Example Part 4

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

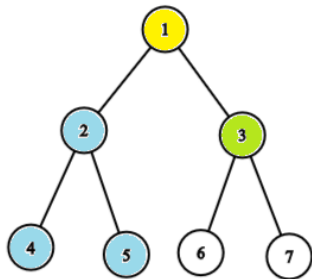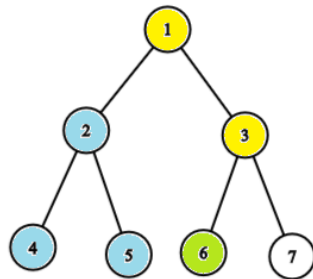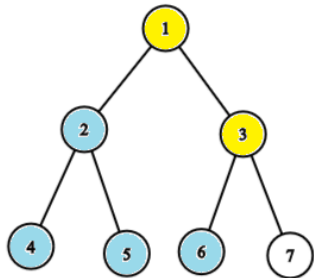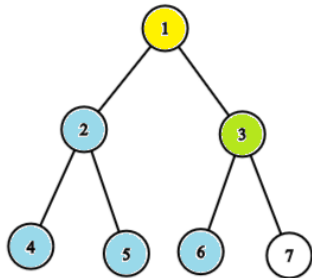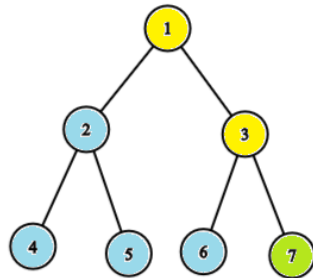Figure: Postorder Traversal: 4-5-2-6

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

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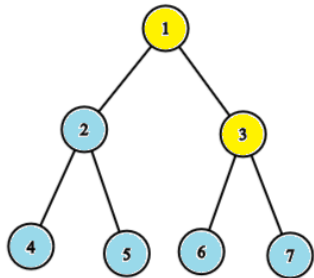
Figure: Postorder Traversal: 4-5-2-6-7
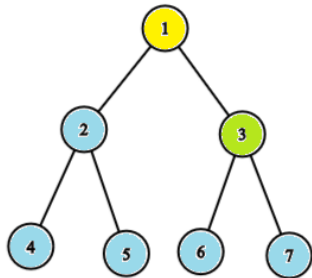
Figure: Postorder Traversal: 4-5-2-6-7
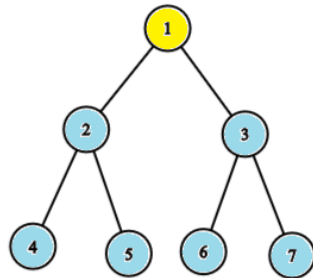
Figure: Postorder Traversal: 4-5-2-6-7-3

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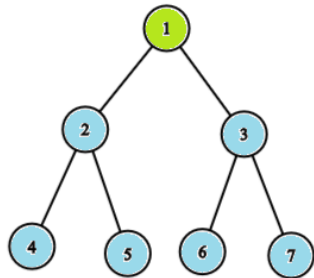
Figure: Postorder Traversal:
4-5-2-6-7-3

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

Postorder Traversal Code

Recursive function:

```cpp
1 void postorderTraversal(TreeNode *root)
2 {
3     if (root == nullptr)
4     {
5         return;
6     }
7     postorderTraversal(root->left);
8     postorderTraversal(root->right);
9     cout << root->value << " ";
10 }
```

Function call:

```cpp
1     // postorder traversal
2     postorderTraversal(root);
3     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

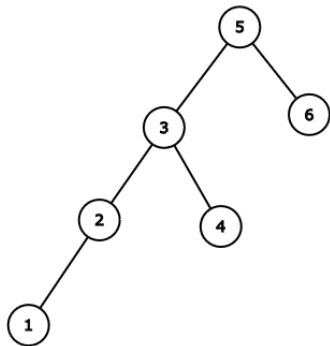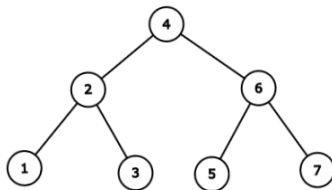Figure: Balanced BST example
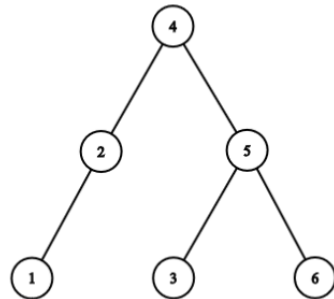
Figure: Invalid BST example

Constructing a BST from Scratch Example

Figure: Midpoint(s): 5
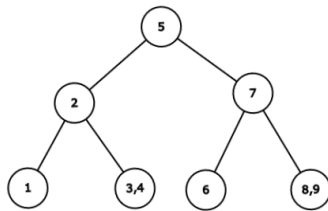
Figure: Midpoint(s): 2, 7
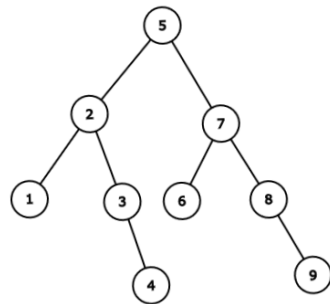
Figure: Balanced BST constructed.

Balanced BST Construction Code, Tree Struct and Function Call

```cpp
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
1 TreeNode* recursiveBSTgen(vector<int>& numbers, int left, int right)
2 {
3     // When there is no element to place
4     if(left > right)
5     {
6         return NULL;
7     }
8     int middle = (left + right) / 2;
9     TreeNode* root = new TreeNode(numbers[middle]);
10     // left and right branches
11     root->left = recursiveBSTgen(numbers, left, middle - 1);
12     root->right = recursiveBSTgen(numbers, middle + 1, right);
13     return root;
14 }
```

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)
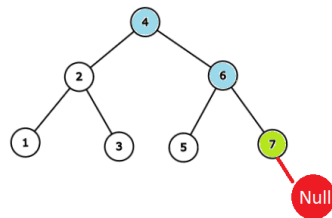
BST Search Example
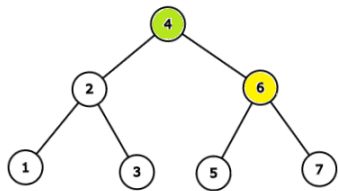
Target   9
Value   4
Action   Go Right
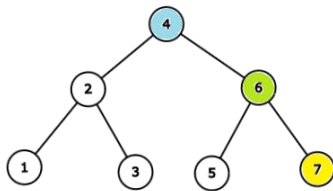
Target   9
Value   6
Action   Go Right

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

```
1 int BSTsearch(TreeNode* node, int target, int previous_level)
2 {
3     if(node == NULL)
4         return -1; // Invalid level or false,
5     if(node->val == target)
6         return previous_level + 1; // Current level or true,
7     if(node->val > target)
8         return BSTsearch(node->left, target, previous_level + 1);
9     if(node->val < target)
10         return BSTsearch(node->right, target, previous_level + 1);
11 }
```

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)*