COM 201 – Data Structures And Algorithms
Sample Questions

Assist. Prof. Özge Öztimur Karadağ

ALKÜ

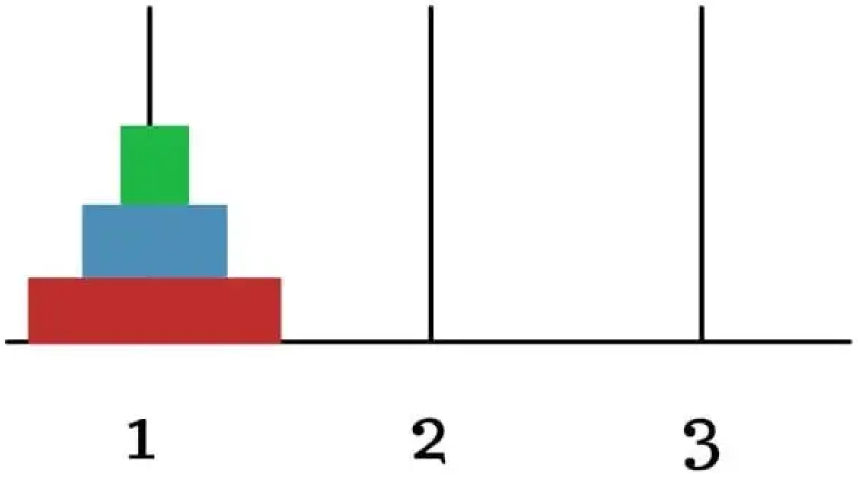
Previously

- Stack
- Queue
- Tree
- Graph

- Recursion

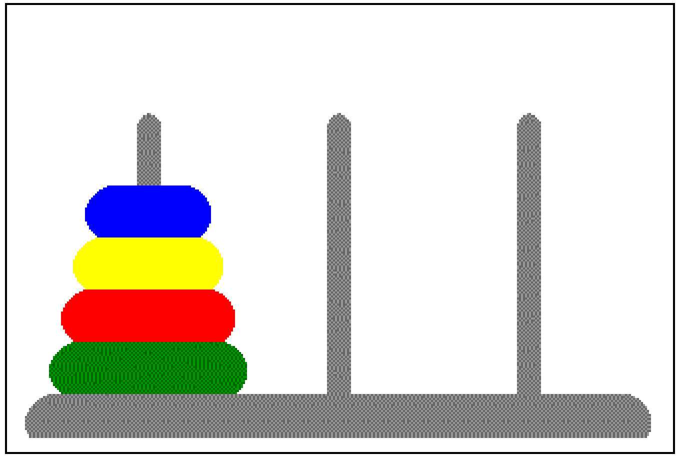
Tower of Hanoi

- It is a mathematical game or puzzle that consists of three rods with ’n’ number of disks of different diameters.
- The objective of the game is to shift the entire stack of disks from one rod to another rod following these three rules :
  - Only one disk can be moved at a time.
  - Only the uppermost disk from one stack can be moved on to the top of another stack or an empty rod.
  - Larger disks cannot be placed on the top of smaller disks.
- **Objective :** To solve the Tower of Hanoi puzzle that contains three disks. The stack of disks has to be shifted from Rod 1 to Rod 3 by abiding to the set of rules that has been mentioned above.



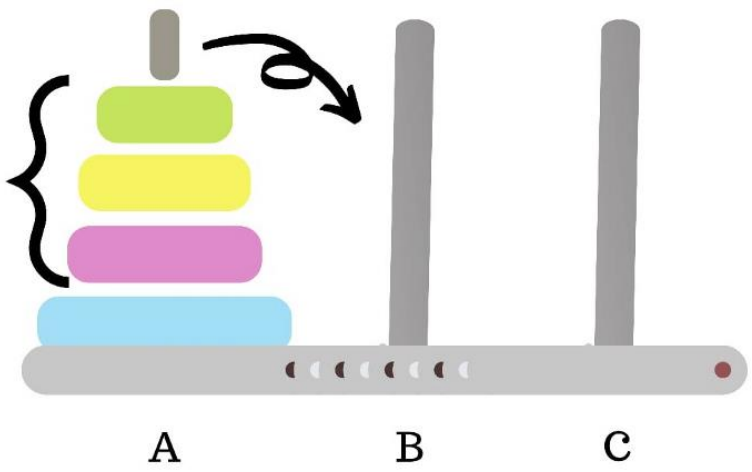
Tower of Hanoi

- **Let us try to solve the Tower of Hanoi puzzle for n=4 disks.**



Tower of Hanoi

- Strategy:
  1. Recursively solve the puzzle of shifting disks 1 , 2 , 3 from Rod A to Rod B.



  2. Then move the largest disk 4 from Rod A to destination Rod C.
  3. Recursively solve the puzzle of shifting the disk 1 , 2 , 3 from Rod B to Rod C.

Tower of Hanoi

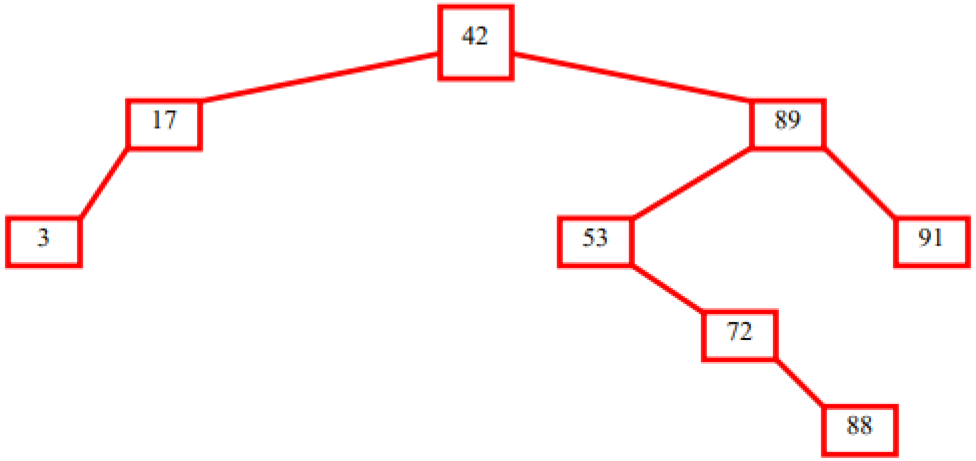
- C++

```
#include <iostream>
using namespace std;

void towerOfHanoi(int n,char from_rod, char to_rod, char aux_rod){

}

int main(){
        int n;
        cout<<"Enter the number of disks"<<endl;
        cin>>n;
        towerOfHanoi(n,'A','C','B');
        return 0;
}
```

Binary Search Tree

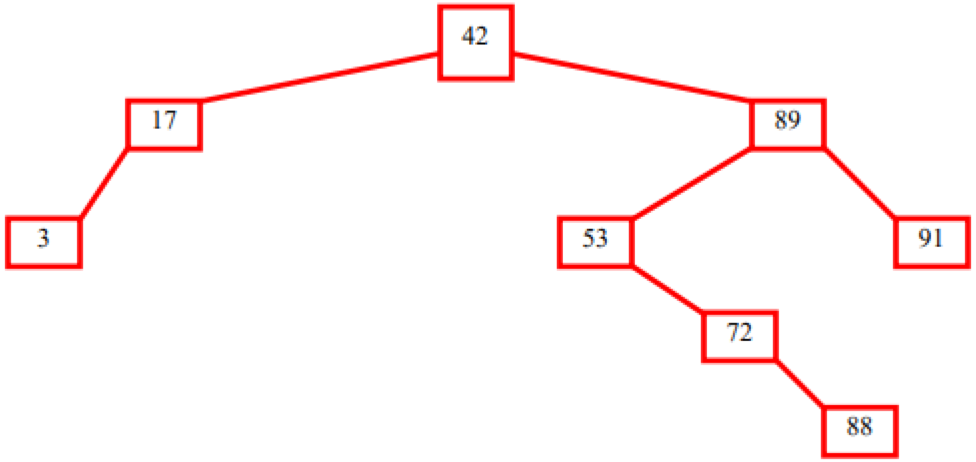
- Draw the BST that results from inserting the following data values in the given order :

  42 17 89 53 72 91 3 88



Binary Search Tree

- Draw what the tree would look like after deleting the value 42?



```
template <class Data> class NodeT {
private:
   Data         Element;
   NodeT<Data>* Left;
   NodeT<Data>* Right;

public:
   // irrelevant members omitted
   Data getData() const;
   NodeT<Data>* getLeft() const;
   NodeT<Data>* getRight() const;
};
```

```
template <class Data> class BSTreeT {
private:
   NodeT<Data>* Root;

   // irrelevant members omitted
public:
   // irrelevant members omitted
   bool Find(const Data& D);
};
```

- Given the above definitions, write the body of the member function Find()

```
template <class Data> bool BST<Data>::Find(const Data& toFind) {

   return FindHelper(Root, toFind);
}

template <class Data>
bool BST<Data>::FindHelper(NodeT<Data>* sRoot,
                           const Data& toFind) {

}
```

Binary Tree

- What are the minimum and the maximum number of nodes in a complete binary tree of height h?

- min = 2^h
- max = 2^{h+1} -1

Graph Implementation in C++

- Edges are represented by adjacency lists, for this purpose list ADT is used.
  - A list is an abstract data type that describes a linear collection of data items in some order, in that each element occupies a specific position in the list. The order could be alphabetic or numeric or it could just be the order in which the list elements have been added. Unlike a set, the elements of a list do not need to be unique.
  - List in C++ Standard Template Library
    - Sample code

Graph

- Count the number of connected components in a graph

- *Do either **BFS** or **DFS** starting from every unvisited vertex, and we get all strongly connected components.*

- *Steps to follow using DFS:*
  - Initialize all vertices as not visited.
  - Do the following for every vertex **v**:
    - If **v** is not visited before, call the DFS. and print the newline character to print each component in a new line
      - Mark **v** as visited and print **v**.
      - For every adjacent **u** of **v**, If **u** is not visited, then recursively call the DFS.

```cpp
#include <bits/stdc++.h>
using namespace std;

// Graph class represents an undirected graph using adjacency list representation
class Graph {
	int V; // No. of vertices

	list<int>* adj; // Pointer to an array containing adjacency lists

	void DFSUtil(int v, bool visited[]); // A function used by DFS

	public:
	Graph(int V); // Constructor

	void addEdge(int v, int w);
	int NumberOfconnectedComponents();
};

// Function to return the number of connected components in an undirected graph
int Graph::NumberOfconnectedComponents()
{

	bool* visited = new bool[V]; // Mark all the vertices as not visited

	int count = 0; // To store the number of connected components
	for (int v = 0; v < V; v++)
		visited[v] = false;

	for (int v = 0; v < V; v++) {
		if (visited[v] == false) {
			DFSUtil(v, visited);
			count += 1;
		}
	}
	return count;
}
```

```cpp
void Graph::DFSUtil(int v, bool visited[])
{

	visited[v] = true; // Mark the current node as visited

	// Recur for all the vertices
	// adjacent to this vertex
	list<int>::iterator i;

	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFSUtil(*i, visited);
}
Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) // Add an undirected edge
{
	adj[v].push_back(w);
	adj[w].push_back(v);
}
int main()
{
	Graph g(5);
	g.addEdge(1, 0);
	g.addEdge(2, 3);
	g.addEdge(3, 4);

	cout << g.NumberOfconnectedComponents();

	return 0;
}
```

Graph

- Given an adjacency list representation undirected graph. Write a function to count the number of edges in the undirected graph.

```
#include<bits/stdc++.h>
using namespace std;

class Graph   // Adjacency list representation of graph
{
        int V ;
        list < int > *adj;
public :
        Graph( int V )
        {
                this->V = V ;
                adj = new list<int>[V];
        }
        void addEdge ( int u, int v ) ;
        int countEdges () ;
};

void Graph :: addEdge ( int u, int v ) // add edge to graph
{
        adj[u].push_back(v);
        adj[v].push_back(u);
}

int Graph :: countEdges() // Returns count of edge in undirected graph
{
        int sum = 0;

        for (int i = 0 ; i < V ; i++) //traverse all vertex
                sum += adj[i].size();           // add all edge that are linked to the
                                                // current vertex

        // The count of edge is always even because in
        // undirected graph every edge is connected
        // twice between two vertices
        return sum/2;
}
```

```
// driver program to check above function
int main()
{
        int V = 9 ;
        Graph g(V);

        // making above shown graph
        g.addEdge(0, 1 );
        g.addEdge(0, 7 );
        g.addEdge(1, 2 );
        g.addEdge(1, 7 );
        g.addEdge(2, 3 );
        g.addEdge(2, 8 );
        g.addEdge(2, 5 );
        g.addEdge(3, 4 );
        g.addEdge(3, 5 );
        g.addEdge(4, 5 );
        g.addEdge(5, 6 );
        g.addEdge(6, 7 );
        g.addEdge(6, 8 );
        g.addEdge(7, 8 );

        cout << g.countEdges() << endl;

        return 0;
}
```