# Lecture 5 Stacks/Queues and Trees

| | |
|---|---|
| Class | Cohort 2 Year 1 |
| Created | @Oct 24, 2020 4:34 PM |
| Materials | |
| Reviewed | ☐ |
| Type | Lecture |

## Stacks And Queues

- Implement a function string* findBin(int n) which will generate binary numbers from 1 to n stored in an array of type string by making use of a queue.
- Implement the a functions to implement two stacks using a single array to store elements. You can make changes to the constructor as well.
- Implement the function myQueue reverseK(myQueue queue, int k) which takes a queue and a number k as input and reverses the first k elements of the queue. An illustration is also provided for your understanding.
- You have to implement enqueue() and dequeue() functions in the newQueue class using the myStack class we created earlier. enqueue() will insert a value into the queue and dequeue will remove a value from the queue.
- You have to implement myStack sortStack(myStack stack, int size) function which will take a stack and sort all its elements in ascending order.
- In this problem, you have to implement the int evaluatePostFix(string exp) function which will take a compute a postfix expression given to it in a string. Your code should handle these four operators: +,-,*,/.
- You must implement the int* nextGreaterElement(int *arr, int size) function. For each element in an array, it finds the next greater element in that array.

- In this problem, you have to implement the `isBalanced()` function which will take a string containing only curly `{}`, square `[]`, and round `()` parentheses. The function will tell us whether all the parentheses in the string are balanced or not.
  For all the parentheses to be balanced, every opening parenthesis must have a closing one. The order in which they appear also matters. For example, `{[]}` is balanced, but `{[}]` is not.
- You have to implement the minStack class which will have a min() function. Whenever min() is called, the minimum value of the stack is returned in O(1) time. The element is not popped from the stack. Its value is simply returned.

## Trees

Trees consist of vertices (nodes) and edges that connect them. Unlike the linear data structures that we have studied so far, trees are hierarchical. They are similar to Graphs, except that a **cycle** cannot exist in a Tree - they are **acyclic**. In other words, there is always exactly one path between any two nodes.

- **Root Node**: A node with no parent nodes.
- **Child Node**: A Node that is linked to an upper node (*Parent Node*).
- **Parent Nodes**: A Node that has links to one or more child nodes.
- **Sibling Node**: Nodes that share the same *Parent Node*.
- **Leaf Node**: A node that doesn't have any *Child Nodes*.
- **Ancestor Nodes**: the nodes on the path from a node *d* to the root node. Ancestor nodes include node *d*'s parents, grandparents, and so on.
- **Sub-tree**: A subtree is a portion of a tree that can be viewed as a complete tree on its own. Any node in a tree, together with all the connected nodes below it, comprise a subtree of the original tree.
- **Degree of a node**: Total number of children of a node.
- **Length of a path**: The number of edges in a path.

- **Depth of a node *n***: The length of the path from a node *n* to the root node. The depth of the root node is 0.
- **Level of a node *n***: (Depth of a Node)+1.
- **Height of a node *n***: The length of the path from *n* to its deepest descendant. So the height of the tree itself is the height of the root node, and the height of leaf nodes is always 0.
- **Height of a Tree**: Height of its root node.

## Some Tree Types

- Binary Tree
- Binary Search Trees
- AVL Trees
- Red-Black Trees
- 2-3 Trees

## Types of Binary Trees

- Complete Binary Trees
- Full Binary Trees
- Perfect Binary Trees
- Balanced Binary Tree
- Skewed Binary Tree

How to check if a binary tree is balanced?

### Binary Search Tree

NodeValues(leftsubtree) <= CurrentNodeValue <= NodeValues(rightsubtree)

Bare bone implementation of BST

```
#include "BST.h"

Node::Node() {
  value = 0;
  leftChild = NULL;
  rightChild = NULL;
}

Node::Node(int val) {
  value = val;
  leftChild = NULL;
  rightChild = NULL;
}

BinarySearchTree::BinarySearchTree() {
  root = NULL;
}

BinarySearchTree::BinarySearchTree(int rootValue) {
  root = new Node(rootValue);
}

Node * BinarySearchTree::getRoot() {
  return root;
}
```

### BST Insertion

### Iterative

```
void insertBST(int val) {
    if(getRoot()==NULL){
            root=new Node(val);
            return;
    }
    //starting from the root
    Node* currentNode = root;
    Node* parent;
    //while we get to the null node
    while (currentNode) {
        parent = currentNode; //update the parent
        if (val < currentNode->value) {
            //if newValue < currentNode.val,
            //iterate to the left subtree
            currentNode = currentNode->leftChild;
        } else {
            //if newValue >= currentNode.val,
            //iterate to the right subtree
            currentNode = currentNode->rightChild;
        }
    }
    //by now, we will have the parent of the null
    //node where we have to insert the newValue
    if (val < parent->value) {
        //if newValue < parent.val
        //insert into the leftChild
        parent->leftChild = new Node(val);
    } else {
        //if newValue >= parent.val
        //insert into the rightChild
        parent->rightChild = new Node(val);
    }
}
```

Insert BST (recursive)

```
Node* insert(Node* currentNode, int val) {
    if(currentNode==NULL) {
        return new Node(val);
    }
    else if(currentNode->value > val) {

        currentNode->leftChild=insert(currentNode->leftChild,val);

    }
    else {
        currentNode->rightChild=insert(currentNode->rightChild,val);
    }

    return currentNode;

}

void insertBST(int value) {

    if(getRoot()==NULL){
        root=new Node(value);
        return;
    }
    insert(this->getRoot(),value);
}
```

## Examples of Balanced Binary Trees

1. AVL trees
2. Red-Black Trees
3. 2-3 trees