Objective:

- Learning and Implementing Interval Tree: A variation of Binary Search tree.

Interval Tree

[Taken from: http://www.geeksforgeeks.org/interval-tree/]

Consider the following examples of intervals, which will help you, understand the basic meaning of intervals.

Toy example #1:

I want to book a conference room from 16:00 to 17:00.
Does this time conflict with any previous bookings?

The current bookings are:
9:00 to 9:30
10:00 to 12:00
12:30 to 13:00
14:00 to 15:00
16:30 to 19:30

Toy example #2:

Does my lifeguarding shift overlap with anyone else's shift?
If so, which one?
Given: I am a lifeguard from 9:00 to 12:00.
The other shifts are: 6:00 to 8:00, 10:00 to 13:00, 12:00 to 15:00 and 14:00 to 17:00.

When we say that two intervals overlap:

- Intervals i and j overlap iff:

  low[i] ≤ high[j] and low[j] ≤ high[i]

- Intervals i and j do not overlap iff:

  high[i] < low[j] or high[j] < low[i]

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.

1. Add an interval
2. Remove an interval
3. Given an interval x, find if x overlaps with any of the existing intervals.

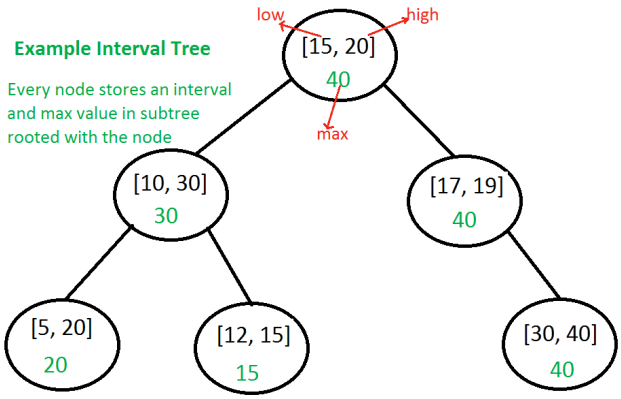
Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like Red Black Tree, AVL Tree, etc (You will study one of them in next week) to maintain set of intervals so that all operations can be done in O(Logn) time (You can work on augmenting BST).

Every node of Interval Tree stores following information:

a) **i**: An interval which is represented as a pair *[low, high]*

b) **max**: Maximum *high* value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.



The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval *x* in an Interval tree rooted with *root*.

Interval *overlappingIntervalSearch*(root, x)

1) If x overlaps with root's interval, return the root's interval.
2) If left child of root is not empty and the *max* in left child is greater than x's low value, recur for left child
3) Else recur for right child.

Implementation of Interval Tree:

Following is C++ implementation of Interval Tree. The implementation uses basic insert operation of BST to keep things simple. Ideally it should be insertion of AVL Tree or insertion of Red-Black Tree. Deletion from BST is left as an exercise.

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

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i;
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert.  Here the low value of interval
// is used tomaintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval goes to
    // left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
```

```
bool doOVerlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *intervalSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOVerlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval is left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return intervalSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return intervalSearch(root->right, i);
}

void inorder(ITNode *root)
{
    if (root == NULL) return;

    inorder(root->left);

    cout << "[" << root->i->low << ", " << root->i->high << "]"
         << " max = " << root->max << endl;

    inorder(root->right);
}

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
        {5, 20}, {12, 15}, {30, 40}
    };
    int n = sizeof(ints)/sizeof(ints[0]);
    ITNode *root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, ints[i]);

    cout << "Inorder traversal of constructed Interval Tree is\n";
    inorder(root);

    Interval x = {6, 7};

    cout << "\nSearching for interval [" << x.low << "," << x.high << "]";
    Interval *res = intervalSearch(root, x);
    if (res == NULL)
        cout << "\nNo Overlapping Interval";
```

```
    else
        cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";
}
```

Output:

Inorder traversal of constructed Interval Tree is
[5, 20] max = 20
[10, 30] max = 30
[12, 15] max = 15
[15, 20] max = 40
[17, 19] max = 40
[30, 40] max = 40

Searching for interval [6,7]
Overlaps with [5, 20]

Applications of Interval Tree: Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source Wiki).

Your Task:

- Implement remove operation for interval tree.
- Extend the intervalSearch() to print all overlapping intervals instead of just one.
- Develop an object oriented ADT for Interval Tree