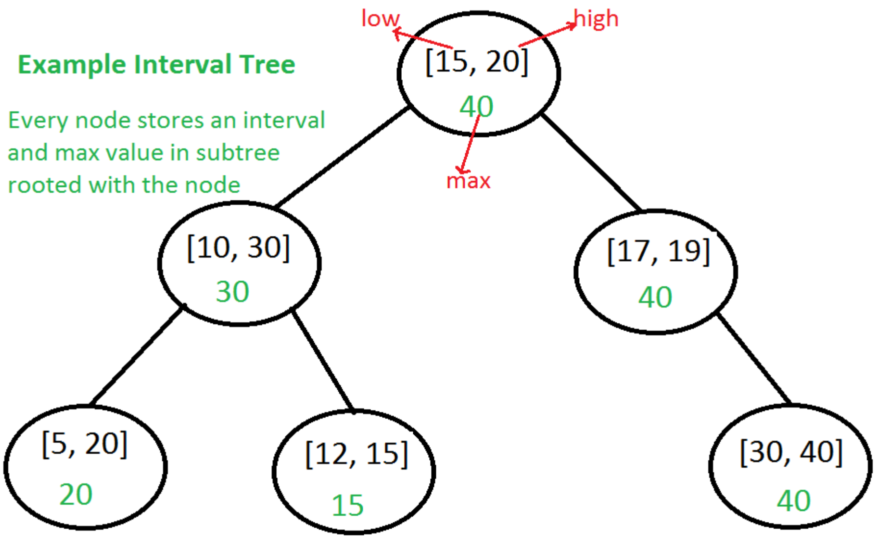
Interval Tree

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 to maintain set of intervals so that all operations can be done in O(Logn) time.

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.
```

How does the above algorithm work?
Let the interval to be searched be x. We need to prove this in for following two cases.

Case 1: *When we go to right subtree, one of the following must be true.*
a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than *x.low*. So the interval cannot be present in left subtree.

Case 2: *When we go to left subtree, one of the following must be true.*

a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.
… We went to left subtree because *x.low* <= *max* in left subtree
…. max in left subtree is a high of one of the intervals let us say *[a, max]* in left subtree.
…. Since *x* doesn't overlap with any node in left subtree *x.low* must be smaller than '*a*'.
…. All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than '*a*'.
…. From above two facts, we can say all intervals in right subtree have low value greater than*x.low*. So *x* cannot overlap with any interval in right subtree.

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;  // 'i' could also be a normal variable
    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 *overlapSearch(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 overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(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 = overlapSearch(root, x);
    if (res == NULL)
        cout << "\nNo Overlapping Interval";
    else
        cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";
    return 0;
}
```

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).

Interval Tree vs Segment Tree
Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:
1) Implement delete operation for interval tree.
2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree
http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
https://www.youtube.com/watch?v=dQF0zyaym8A