Interval Tree

Elimizde bir dizi aralığın olduğu ve aşağıdaki 3 işlemi verimli bir şekilde yapmamız gereken bir durumu düşünelim.

1) Aralık ekleme
2) Aralık çıkarma
3) Verilen bir x aralığı için x'in var olan diğer aralıklarla kesişip kesişmediğini bulma

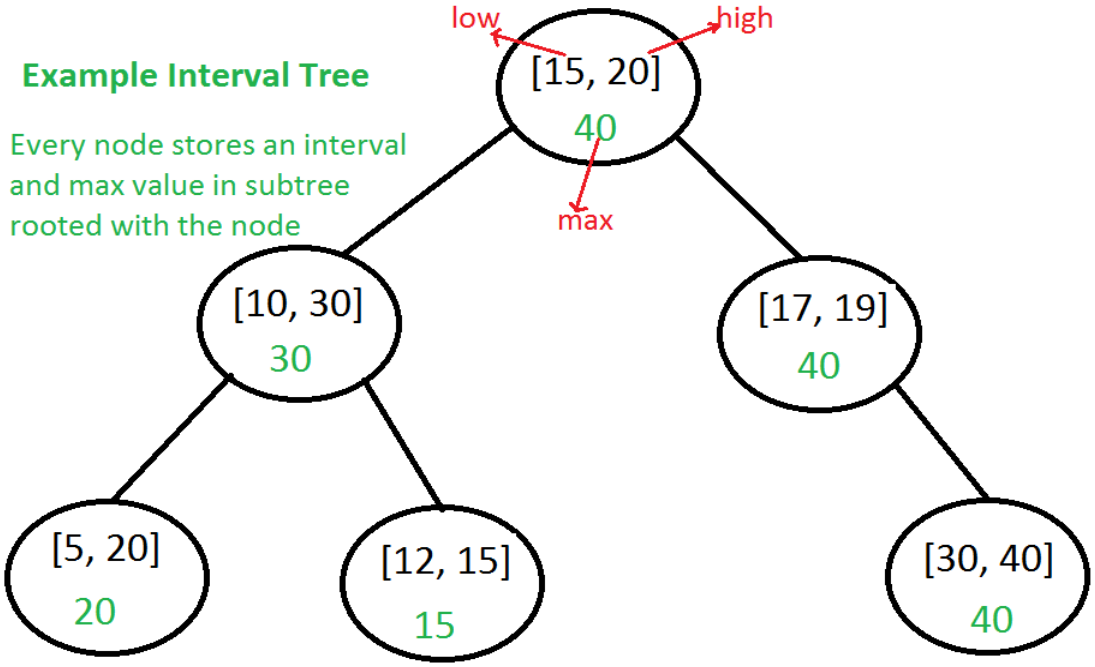
Interval Tree: Interval Tree'de olay, Red Black Tree ve AVL Tree gibi, kendini dengeleyen bir Binary Search Tree yapısı oluşturarak yukarıda bahsettiğimiz işlemleri O(logN) zamanda yapabilmektir.

Interval Tree'de her bir node aşağıdaki değerleri tutar.

a) **i**: Aralığı yani baş ve sonu belirten bir çift değişken. *[low, high]*

b) **max**: Root'un o node olduğu ağaçtaki en büyük high değeri.

Binary search tree yapısını oluştururken her bir node'daki low değerini(aralığın başlangıç noktası) referans alıyoruz. Interval tree'de ekleme ve silme operasyonları kendini dengeleyen (self-balancing) BST'dekinin aynısıdır.



Yapacağımız temel işlem çakışan aralıkları aramak. Aşağıda root'u *root* değişkeni olan bir interval tree'de bir x aralığı ile çakışan aralığı bulan bir algoritma verilmiştir:

Interval overlappingIntervalSearch(root, x)

1) Eğer x root'un aralığıyla çakışıyorsa root'un aralığını return et.

2) Eğer root'un sol çocuğu varsa ve sol çocuğun max değeri x'in low value'sundan küçükse aynı işlemi sol çocuk için tekrarla.

3) Değilse sağ çocuk için tekrarla.

Yukarıdaki algoritma nasıl çalışıyor? How does the above algorithm work?

Aradığımız aralık x olsun. Aşağıdaki iki durum için ispatları yapmamız gerekiyor.

İlk Durum: Sağ alt ağaca gittiğimiz zaman, aşağıdakilerden biri doğru olmalı:

a) Sağ alt ağaçta bir çakışma vardır. Bu durumda çakışan bir aralığı return etmemiz gerektiğinden bu durum bizim aradığımız durum.

b) İki alt ağaçta da çakışma yoktur. Sadece sol alt ağaç olmadığında veya soldaki ağacın max değeri x'in low değerinden daha küçük olduğunda sağ alt ağaca gidiyoruz. Yani aradığımız aralık sol alt ağaçta bulunamaz.

Diğer Durum: Sol alt ağaca gittiğimizde aşağıdakilerden biri doğru olmalı:

a) Sol alt ağaçta bir çakışma vardır. Çakışan bir aralığı bulmamız gerektiğinden bu durum bizim aradığımız durum.

b) İki alt ağaçta da çakışma yoktur: Burası ispatın en önemli bölümü. Şimdi daha önce bildiğimiz bazı durumları tekrar hatırlamamız gerekiyor.

… Sol alt ağaca gidiyoruz. Çünkü sol alt ağaç için x.low <= max

…. Sol alt ağaçtaki max değeri aralıklardan bir tanesinin high değeridir. Bu aralığa [a, max] aralığı diyelim.

…. x sol alt ağaçtaki hiçbir aralıkla çakışmadığından, x.low 'a' değerinden küçüktür.

…. BST'deki tüm node'lar low değerlerine göre sıralı olduğundan, sağ alt ağaçtaki tüm low değerleri 'a' dan büyüktür.

…. Yukarıdaki iki maddeye bakarak, sağ alt ağaçtaki tüm aralıkların x.low'dan daha büyük low değerlerine sahip olduğunu söyleyebiliriz. Böylelikle x'in sağ alt ağaçtaki hiçbir aralıkla çakışmasının mümkün olmadığını görüyoruz.

Interval Tree'nin Implementasyonu:

Aşağıda C++'ta yazılmış bir Interval Tree kodunu görüyorsunuz. Aşağıdaki kodda olay basit tutulmak istendiği için BST'deki temel ekleme işlemini kullanılıyor. En ideal ekleme ise AVL Tree veya Red-Black Tree'deki ekleme şeklidir. BST'den silme ise alıştırma olarak size bırakılmıştır.

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

// Aralığı temsil eden bir struct yapısı
struct Interval
{
    int low, high;
};

// Interval Search Tree'yi temsil eden bir node struct'ı
struct ITNode
{
    Interval *i;  // i pointer değil de normal bir değişken olsa da olur.
    int max;
    ITNode *left, *right;
};

// Yeni bir Interval Search Tree Node'u üretmek için kullanışlı bir fonksiyon
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// Yeni bir Interval Search Tree Node'u eklemek için kullanışlı bir fonksiyon
// BST'deki ekleme işlemine benziyor. BST yapısı aralığın başlangıç değerine yani low'a göre oluşturuluyor.
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Ağaç boş, yeni gelen node root oluyor.
    if (root == NULL)
        return newNode(i);

    // Root'taki aralığın başlangıcını, yani low değerini l'ye atıyoruz.
    int l = root->i->low;

    // Root'un low değeri daha küçükse yeni aralık sol alt ağaca gidiyor.
    if (i.low < l)
```

```
        root->left = insert(root->left, i);

    // Değilse, yeni node sağ alt ağaca gidiyor.
    else
        root->right = insert(root->right, i);

    // Gerekiyorsa atasını high değerini güncelle.
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// İki aralığın çakışıp çakışmadığını kontrol etmek için bir
fonksiyon
bool doOVerlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// Verilen bir i aralığını Interval Tree'de arayan temel bir
fonksiyon.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, ağaç boş
    if (root == NULL) return NULL;

    // Verilen aralık root'la çakışıyorsa
    if (doOverlap(*(root->i), i))
        return root->i;

    // Root'un sol çocuğu varsa ve sol çocuğun max değeri verilen
aralığa büyük ya da eşitse i aralığı sol alt ağaçtaki bir aralıkla
çakışabilir.
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Değilse i sadece sağ alt ağaçtaki bir aralıkla kesişebilir.
    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);
}

// Main fonksiyonu
int main()
{
    // Yukarıdaki şekilde gösterilen Interval Tree'yi oluşturalım.
    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]

Interval Tree Uygulamaları:

Interval tree temel olarak geometrik bir veri yapısıdır. Sıklıkla sorgular (Windowing queries) için kullanılır. Örnek verecek olursak, dikdörtgen biçimindeki bir haritada tüm yolları bulmak için veya 3 boyutlu bir sahnede görünen elemanları bulmak için kullanılır.

Interval Tree vs Segment Tree

İki ağaç yapısı da aralıkları tutar. Segment tree bir nokta için yapılan sorgularda daha iyidir. Interval tree ise çakışan aralıklarda daha iyi performans göstermektedir.

Alıştırma:

1) Interval Tree için silme işlemini kodlayınız.
2) intervalSearch() fonksiyonunu sadece bir değil çakışan tüm aralıkları yazdıracak şekilde tekrardan yazınız.