امیرحسین نجفی زاده 9831065

به وسیله یک تغییر در یک درخت قرمز–سیاه ساختمان داده‌ای بسازید که اعمال زیر را با این هزینه‌ها انجام دهد:

تابع	هزینه
Insert(x)	O(log(n))
Delete(x)	O(log(n))
Find(x)	O(log(n))
Count(x)	O(log(n))

تابع `Find`به عنوان ورودی یک عدد می‌گیرد و در صورت وجود این عدد یک اشاره گر به گره‌ای که این مقدار را دارد برمی‌گرداند.تابع `Count` نیز تعداد گره های کوچکتر از `x`را نشان می دهد.

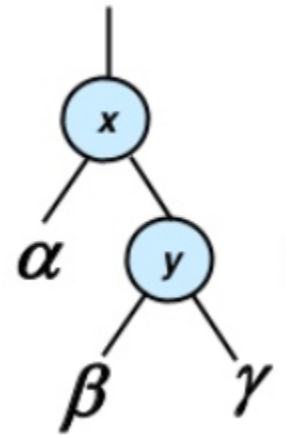
عملیات های Insert و Delete می دانیم با هزینه $O(\log n)$ انجام می شوند پس مشکلی ندارند.

عملیات Find هم در صورت بدترین حالت خودش باید یک بار تا انتهای درخت برود (به اندازه ی ارتفاع درخت پایین برود) و چون ارتفاع درخت قرمز – سیاه برابر $\log n$ است پس هزینه این عملیات هم خود به خود $O(\log n)$ است.

و اما تابع Count نیاز دارد تا تعداد گره های کمتر از یک عدد را بشمارد. برای اینکار از الگوریتم زیر استفاده می کنیم.

منتها قبل الگوریتم یک ویژگی جدید برای هر نود در درخت تعریف می کنیم. آن ویژگی تعداد نود های داخل زیر درختی با root آن نود تعریف می شود (شامل خود آن نود) و با نام Nodes Under تعریف می کنیم. نحوه مقدار دهی به این ویژگی را نیز به این صورت تعریف می کنیم :

1. برای nil ها این مقدار را 0 قرار بده.
2. به هنگام insert کردن مقدار را برای نود جدید 1 قرار بده. از نود جدید تا root درخت اصلی پیمایش کن (به صورت x = p[x]) و مقدار Nodes Under هر یک از نود ها در مسیر را + 1 کن.
3. به هنگام delete کردن ، از نود پدر (p[z]) شروع کن و تا root درخت اصلی برو و مقدار Nodes Under را –1 کن.
4. حال دو عملیات Left-Rotate و Right-Rotate باقی مانده ، که در اینجا left rotate را بررسی می کنیم و right rotate مشابه همین منتها قرینه ی این توضیح عمل می کند.
    1. ابتدا فرض کنید شکل قسمتی از درخت به صورت زیر باشد :



2. از مقدار Nodes Under[x] مقدار Nodes Under[y] را کم کن ، و به آن مقدار Nodes Under[ß] را اضافه کن.
3. از مقدار Nodes Under[y] مقدار Nodes Under[ß] را کم کن ، و به آن مقدار Nodes Under[x] را اضافه کن.
4. حال مقادیر Nodes Under درست می باشند.

با توجه به اینکه نحوه مقدار دهی و تغییر وضعیت ویژگی جدید را توضیح دادیم به سراغ الگوریتم می رویم :

1. ابتدا تابع Find(x) را صدا بزن => O (log n)
2. از root شروع کن تا به مقدار x برسی => O (log n) = O (h)
3. اگر مقدار key نود حاضر از مقدار x کمتر بود ، به اندازه1 + nodes under [left [current]] به تعداد اضافه کن و Constant <= current = right[current]
4. اگر مقدار key نود حاضر از مقدار x بیشتر بود ، Constant <= current = left[current]
5. و در صورت رسیدن به x مقدار Node Under[left[x]] را به تعداد اضافه کن و از تابع خارج بشو (که در شرط while آورده شده)

الگوریتم فوق تعداد نود های با مقدار کمتر از مقدار نود x را به ما می دهد و زمان اجرای آن :

Log n + (C1 + C2) Log n + (1 + Log n) = C * Log n == O(log n)

Here is the pseudo code of algorithm :

```
x <= find(key[x]);

current <= root[T];

int total <= 0;
```

```
while (current != x)
{
    if (key[current] < key[x])
    {
        total += nodes_under[left[current]] + 1;
        current <= right[current];
    } else
    {
        current <= left[current];
    }
}
total += nodes_under + 1;
return total;
```