RB-Tree Augmentation

- Augment x with **Size(x)**, where
  - Size(x) = size of subtree rooted at x
  - Size(NIL) = 0

OS-Select

```
OS-SELECT(x,i) //page 304
// Select the node with rank i
// in the subtree rooted at x
1.  r ← size[left[x]]+1
2.  if i = r then
3.          return x
4.  elseif  i < r then
5.          return OS-SELECT (left[x], i)
6.  else    return OS-SELECT (right[x], i-r)
```

Time Complexity O(log n)

OS-Rank

```
OS-RANK(x,y)
// Different from text (recursive version)
// Find the rank of y in the subtree rooted at x
r = size[left[y]] + 1
if x = y then return r
else if ( key[x] < key[y] ) then
      return OS-RANK(x,left[y])
else return r + OS-RANK(x,right[y] )
```

Time Complexity O(log n)

Augmenting RB-Trees

Theorem 14.1, page 309

Let f be a field that augments a red-black tree T with n nodes, and f(x) can be computed using only the information in nodes x, left[x], and right[x], including f[left[x]] and f[right[x]].

Then, we can maintain f(x) during insertion and deletion without asymptotically affecting the O(lgn) performance of these operations.

For example,

size[x] = size[left[x]] + size[right[x]] + 1

rank[x] = ?

Examples of augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous

Interval Trees

- **Need**: Dynamic data structure to store time intervals
- **Application:** Maintain schedule for set of seminars
- **Operations:** Insert, Delete
- Every interval j has: low[j], high[j]
- **Data Structure:**
  - Augment RB-Tree so that it can store intervals.
  - Ordering based on what key? low values? high values? (high+low)/2 values? (high-low) values?
  - Note that insert and delete are still efficient.
- **New Operation:** Search (find any overlapping interval)
  - Problem with Search!

Augmented Information

- low, high, max
- max[x] = rightmost high value of all intervals in subtree rooted at x
- The value max[x] of each node can be written as:

  max[x] = Max { high[int[x]], max[left[x]], max[right[x]] }
- Therefore it can be maintained efficiently under insertions and deletions

Interval-Search

```
INTERVAL-SEARCH (T, j)
// finds an interval in tree T that overlaps interval j,
// else return NIL.
1.  x = root[T]
2.  while x ≠ NIL and j does not overlap int[x] do
3.        if left[x] ≠ NIL and max[left[x]]>=low[j] then
4.              x = left[x]
5.        else  x = right[x]
6.  return x
```

Time Complexity O(log n)