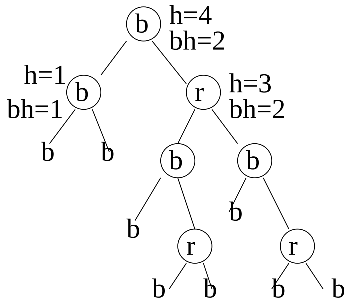
Lecture 3: Red-black trees. Augmenting data structures

A red-black tree is a binary search tree with the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black. Hence there cannot be two consecutive red nodes along a simple path (path without repeated nodes).
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Height h is the number of edges in a longest path to a leaf. **Black-height** $bh(x)$ is the number of black nodes on any path from, but not including x, down to a leaf.

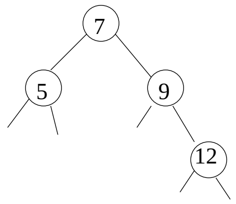
Example of tree, with node colors instead of keys.



Note that the tree is perfectly balanced wrt the black nodes. By property 4 there are $\leq h/2$ red nodes along a path (not counting the subroot), and hence $\geq h/2$ black. So a node of height h has black-height $\geq h/2$.

It is straightforward to prove by induction that a red-black tree with n internal nodes (nonleaves) has height $h \leq 2\,\log(n+1)$.

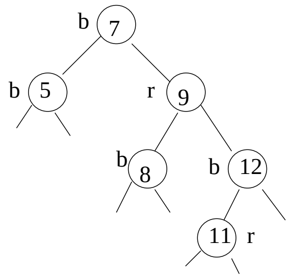
We shall study how the properties of red-black trees are maintained on insertions.



Color the tree interactively and justify the color choices, starting in node 12 (forced choices: 12 red, 5 and 9 black).

Insert 8 (as left child of 9); no problem, color it red.

Insert 11 (as left child of 12); cannot be red (property 4) or black (property 5). Recolor the tree: 11 red, change 9 to red, and 8 and 12 to black. (**Case 1 below.**)



Insert 10 (as left child of 11); now it's not enough to recolor because of unbalance in the tree. We cannot satisfy property 5 without violating property 4. We must change the tree structure by **rotation**, without destroying the search-tree property.

Figure 13.2 shows how rotation works. RIGHT-ROTATE: x keeps its left child, gets y as right child, and inherits y's parent; y keeps its right child and gets x's right child as its left child. The inorder between keys is preserved.

RB-INSERT(z): First find correct leaf; it becomes red internal node z. Property 2 is violated if z is the root. Property 4 is violated when z and $p[z]$ both are red. Move the conflict upwards in the tree until it can be fixed.

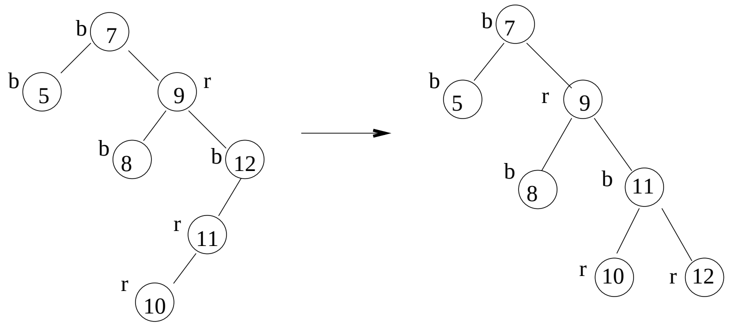
```
TREE-INSERT($z$)                                   usual tree insertion
$color[z] \leftarrow$ RED
**while** $color[p[z]] =$ RED **do**
    **if** $p[z] = left[p[p[z]]]$ **then**         $p[z]$ is left child
        $y \leftarrow right[p[p[z]]]$              its sibling
        **if** $color[y] =$ RED **then**
            $color[p[z]] \leftarrow$ BLACK         Case 1
            $color[y] \leftarrow$ BLACK            Case 1
            $color[p[p[z]]] \leftarrow$ RED        Case 1
            $z \leftarrow p[p[z]]$                 Case 1
        **else if** $z = right[p[z]]$ **then**
                $z \leftarrow p[z]$                Case 2
                LEFT-ROTATE($z$)                   Case 2
            $color[p[z]] \leftarrow$ BLACK         Case 3
            $color[p[p[z]]] \leftarrow$ RED        Case 3
            RIGHT-ROTATE($p[p[z]]$)                Case 3
    **else** same as **then** above by exchanging right and left
$color[root] \leftarrow$ BLACK
```

Note that $p[p[z]]$ exists since $p[z]$ is red and hence not root.

In Case 1, we walk up the tree, which only changes color. In Case 2 or 3, one or two rotations are performed, after which we are done since $p[z]$ is black in the next iteration of **while**.

Hence, time is $O(\log n)$ with only $O(1)$ rotations. Other trees, like AVL-trees, perform $O(\log n)$ rotations at an insertion.

Inserting 10 in the tree above result in Case 3: 11 becomes black, 12 red, followed by a right rotation around 12.



Deletion in red-black trees also takes $O(\log n)$ time, doing at most three rotations. When an internal node is deleted an extra black is introduced and moved up the tree until the red-black properties are satisfied. We skip the details.

Augmenting data structures

1. Choose underlying data structure, for instance a red-black tree.
2. Determine additional information to be maintained, for instance sizes of subtrees.
3. Verify that additional information is updated correctly for the operations on the data structure.
4. Develop new operations.

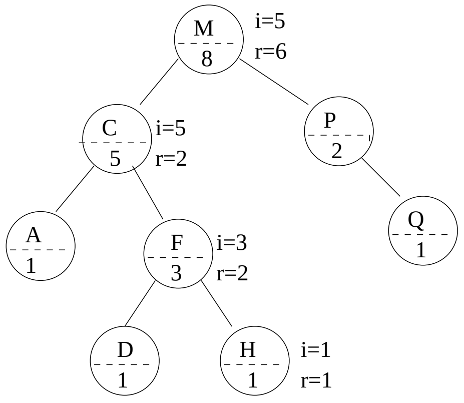
We will illustrate this methodology through two examples:

Dynamic order statistics wants to support the usual dynamic set operations[1], plus:

OS-SELECT(x, i) – returns ith smallest key in subtree rooted at x

OS-RANK(T, x) – returns rank of x in the linear order determined by an inorder traversal of T

Idea: Store sizes of subtrees in the nodes in a red-black tree.



We do not color the nodes red or black since that does not affect the correctness of the algorithm, only its speed. Use letters as keys to separate from sizes in the nodes.

[1]Includes insertions and deletions

```
OS-SELECT($x, i$)

$r \leftarrow size[left[x]] + 1$
**if** $i = r$ **then return** $x$
**elseif** $i < r$ **then return** OS-SELECT($left[x], i$)
**else return** OS-SELECT($right[x], i - r$)
```

Execute OS-SELECT$(root, 5) \rightarrow H$ on the tree above and write i and r beside each visited node. The size of the left subtree determines which subtree contains the answer.

```
OS-RANK($T, x$)

$r \leftarrow size[left[x]] + 1$
$y \leftarrow x$
**while** $y \neq root[T]$ **do**
    **if** $y = right[p[y]]$ **then**
        $r \leftarrow r + size[left[p[y]]] + 1$
    $y \leftarrow p[y]$
**return** $r$
```

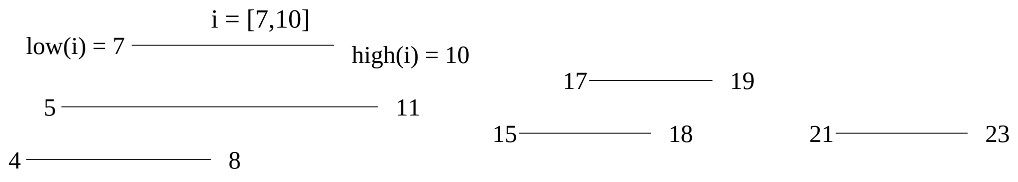
OS-RANK$(root, F)$ gives the answer $2 + 2 = 4$. Find F and back up the search.

Both OS-SELECT and OS-RANK takes $O(\log n)$ time.

Can the data structure be maintained during tree modification? If not, we have to traverse the tree on each update, which takes $\Omega(n)$ time.

During insertion, sizes are incremented by 1 along the traversal path. We can update sizes during rotation in $O(1)$ time by looking at the children of the node (as in Figure 14.2). Deletion is analogous to insertion. Hence, INSERT and DELETE do take $O(\log n)$ time.

Interval trees: to maintain a set of intervals, for instance time intervals.



Find *one* interval in the set that overlaps a given query interval.

For example: $[14, 16] \rightarrow [15, 18]$; $[16, 19] \rightarrow [15, 18]$ or $[17, 19]$; $[12, 14] \rightarrow$ NIL.

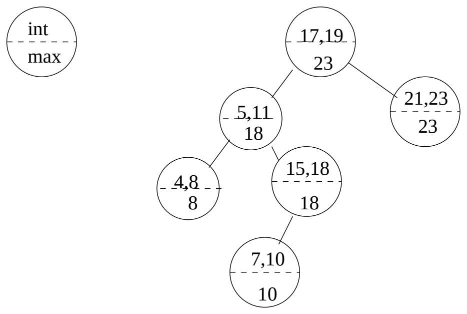
We ignore whether intervals are open or closed by choosing examples where it doesn't matter.

Following the methodology to augment data structures:

1. Underlying data structure: red-black tree of intervals with endpoints *high* and *low*, where the search key is *low*.

2. Additional information: store in each node *max*, the largest endpoint of the intervals in its subtree.

   Example according to above (without node colors).

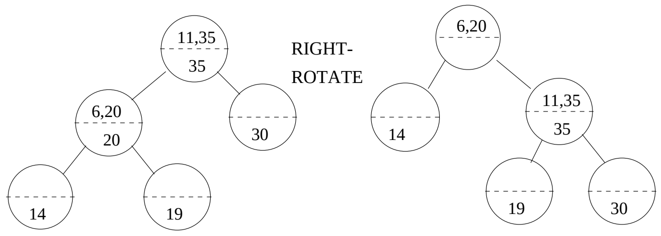


Note that $max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$, since ordered on *low*.

3. Maintain information.

   INSERT:

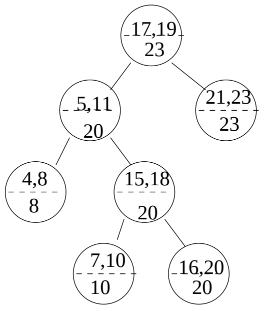
   - Update *max* for subtrees on the downward traversal.
   - If rotations are needed update *max* accordingly:



   Compute new *max* of [11,35] from formula above. *max* of [6,20] can be computed similarly, but is faster copied from old *max* of [11,35].

   - Update *max* after rotation takes $O(1)$ time, i.e. $O(\log n)$ total update time.

   Example: Insert [16,20] in the tree at the top of the page.



   Note: [16,20] overlaps [17,19], but if [16,22] was inserted it would also overlap [21,23].

   DELETE is similar to INSERT.

4. New operation: INTERVAL-SEARCH(T, i), find one interval that overlaps query interval i.

```
$x \leftarrow root[T]$
**while** $x \neq$ NIL and $i$ does not overlap $int[x]$ **do**
        **if** $left[x] \neq$ NIL and $max[left[x]] \geq low[i]$ **then** $x \leftarrow left[x]$
        **else** $x \leftarrow right[x]$
**return** $x$
```

Search for [14,16], and [12,14], in the example tree above. Answers: [15,18] and NIL.

Time is $O(\log n)$, since red-black tree is used.

The key idea is that we only need to check one of node's two subtrees.

If search goes right:

- If there is an overlap in the right subtree, then we are done.
- If there is no overlap in right then there is no overlap in left subtree, since we went right because:
  - $left[x] =$ NIL $\Rightarrow$ no overlap in left, or
  - $max[left[x]] < low[i]$ $\Rightarrow$ no overlap in left.

If search goes left:

- If there is an overlap in the left subtree, then we are done.
- If there is no overlap in left, then there is no overlap in right subtree.
  - Went left because: $low[i] \leq max[left[x]] = high[j]$ for some interval j in left subtree.
  - Since there is no overlap in left subtree, i and j do not intersect.
  - Recall: no overlap if $low[i] > high[j]$ or $low[j] > high[i]$.
  - Since $low[i] \leq high[j]$, must have $low[j] > high[i]$.
  - Consider any interval k in *right* subtree.
  - But since keys are low endpoint: $low[j] \leq low[k]$.
  - Therefore, $high[i] < low[j] \leq low[k]$.
  - Giving $high[i] < low[k]$, so intervals i and k do not intersect.