Lecture 14
Augmenting a Data Structure

Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

TOPICS

- **Augmentation**
- **Order Statistics Dictionary**
- **Interval Tree**

Augmenting a Data Structure

- Suppose we have a base data structure D that efficiently handles a standard set of operations. For instance, D is a Red-Black Tree that supports operations SEARCH, INSERT, and DELETE.

- In some applications, besides the existing operations, we wish our data structure to support an additional set of operations. For instance, the order-statistics operations SELECT and RANK (see next slides).

- How do we efficiently implement the new operations without degrading the efficiency of the existing ones?

- This can be done by augmenting the data structure, i.e., maintaining added pieces of information in it to assist fast implementation of the new operations.

- However, this forces revision of the existing operations to consistently maintain the augmented info while they modify D.

- Given a new application, we need to figure out the following:
    1. What is the base data structure D we wish to use?
    2. What is the augmented information?
    3. How do we efficiently implement the new ops on the augmented D?
    4. How do we revise the existing operations on the augmented D, (ideally) without performance degradation?

ORDER STATISTICS DICTIONARY

Two new (inverse) operations:

RANK(K,D): return the number of items in data set D that are $\leq$ key K.
SELECT(r,D): return the item in D with rank r (return nil if none exists).

Solution 1: D as an un-ordered set of n items.

RANK and SELECT can be done in O(n) time in the worst-case.
RANK(K,D): Sequentially scan through D and count # items $\leq$ K.
SELECT(r,D): See [CLRS chapter 9] or my CSE3101 LS5 or LN4.

Solution 2: D as a sorted array of n items.

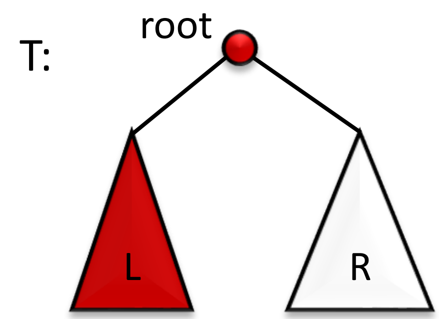
RANK takes O(log n) time by binary-search.
SELECT takes O(1) time by probing rank index position.

What about the dictionary operations?

Solution 3: Augment a search tree. See next slides.

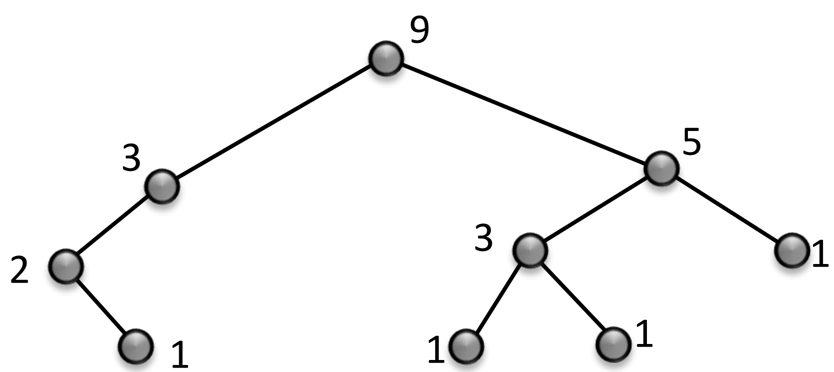
Augmenting a BST

Let T be any BST.
What is rank of the root?
1 + # items in the left subtree.



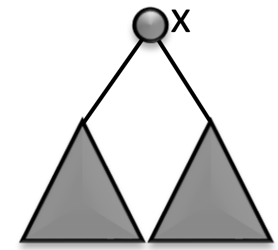
Augmented info in each node x:

size[x] = # items in the subtree rooted at x.
(size[nil] = 0.)

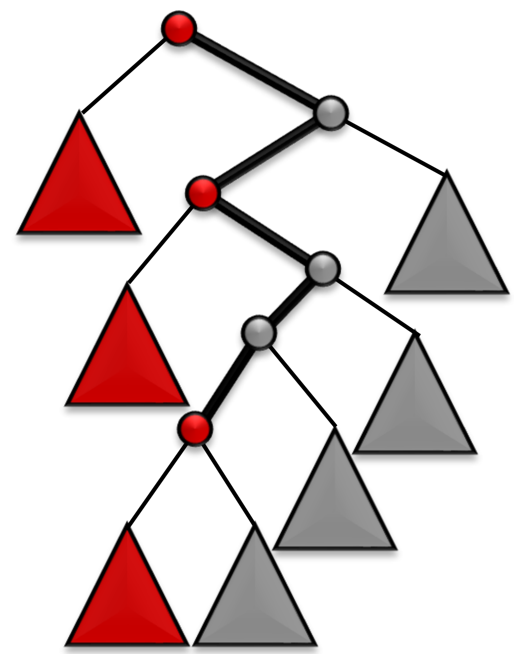


Rank of root = 1 + size[left[root]].

RANK & SELECT on BST




```
Rank(K,x)    (* return rank of key K in BST rooted at x *)
      if x = nil then return 0
      R ← 1+ size[left[x]]          (* root rank  *)
      if K = key[x] then return R
      if K < key[x] then return Rank(K,left[x])
      if K > key[x] then return R + Rank(K, right[x])
end
```

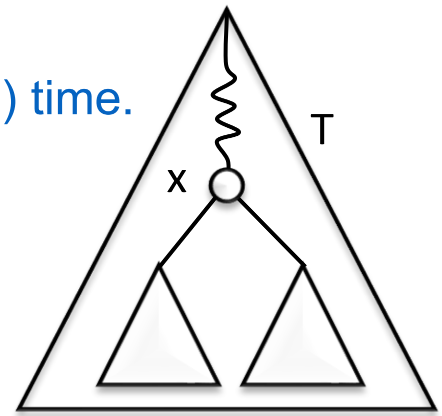


```
Select(r,x)    (* return item of rank r in BST rooted at x *)
      if x = nil then return nil
      R ← 1+ size[left[x]]          (* root rank  *)
      if r = R then return x
      if r < R then return Select(r,left[x])
      if r > R then return Select(r-R, right[x])
end
```

Running time = O(# nodes on the search path).

Maintain Augmented Info

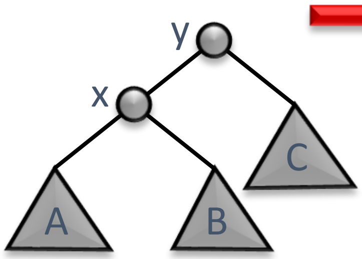
“size[.]” field can be evaluated by a local recurrence in O(1) time.



size[x] = 1 + size[left[x]] + size[right[x]], if $x \neq$ nil
size[nil] = 0.

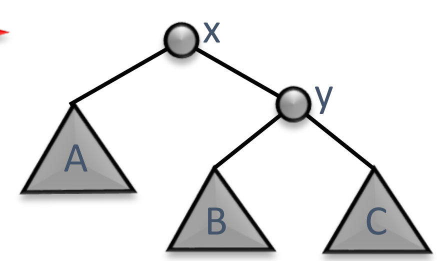
With each dictionary operation (Search, Insert, Delete), update “size” field of affected nodes. What local changes affect the “size” field?

- Insert: attach a new leaf (increment size of all ancestors)
- Delete: splice-out a node (decrement size of all ancestors)
- Rotation:



Rotate(y,x);
size[x] ←size[y];
size[y] ←1+size[left[y]]+size[right[y]]

(Left Rotation is analogous)



Search, Insert, Delete: asymptotic running time unaffected!

Order Statistics Complexity

THEOREM 1:
Augmented Red-Black trees, with the added field size[x] at each node x, support the Order Statistics operations RANK & SELECT, as well as the dictionary operations SEARCH, INSERT, DELETE, in O(log n) worst-case time per operation.

If we use the same augmentation on Splay trees, each of these five operations takes O(log n) amortized time.
[Note: we should "splay the deepest accessed node" after each operation, even after operations RANK & SELECT.]

DEFINITION: Suppose we augment each node x of a BST (or any variant) with a new field f[x]. We say "f" is "O(1) locally composable" if for every node x in the tree, f[x] can be determined in O(1) time from the contents of nodes x, left[x], and right[x] (including their "f" fields).

[For instance "size", as defined above, is O(1) locally composable.]

AUGMENTATION THEOREM

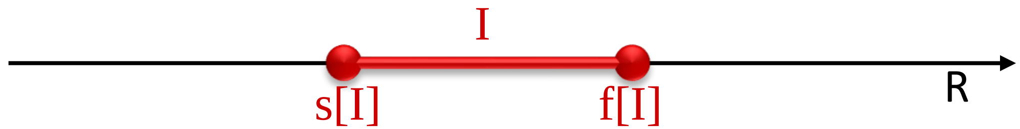
THEOREM 2:
Suppose we augment Red-Black trees with a new O(1) locally composable field f[x] at each node x. Then field "f" in every node of the tree can be consistently maintained by dictionary operations SEARCH, INSERT, DELETE, without affecting their O(log n) worst-case running time per operation.

If we use the same augmentation on Splay trees, each dictionary operation still takes O(log n) amortized time.

Proof: Generalize the "size" field augmentation idea.
If x is the deepest affected accessed node,
then bottom-up update f[y], for every ancestor y of x, inclusive.
Also revise each rotation in O(1) time to update the field f at its affected local nodes.

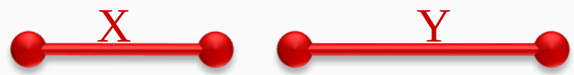
Intervals on the real line

Interval I on the real line = $[s[I], f[I]]$ (from start $s[I]$ to finish $f[I]$, inclusive).

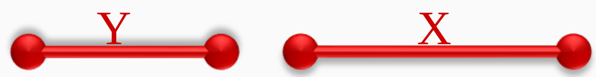


Dichotomy: For intervals X and Y exactly one of the following 3 holds:

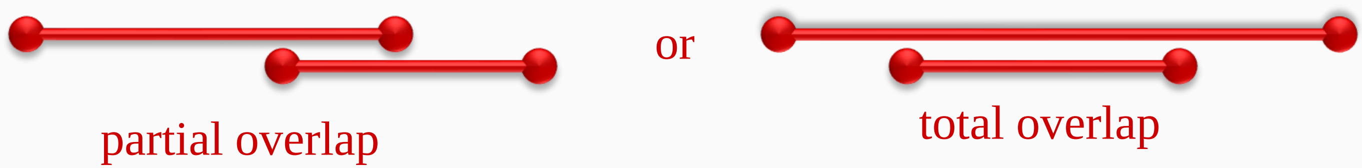
(1) X left of Y: $f[X] < s[Y]$.



(2) Y left of X: $f[Y] < s[X]$.



(3) X and Y overlap: $f[X] \geq s[Y]$ and $f[Y] \geq s[X]$.



Interval Dictionary Problem

PROBLEM:
Maintain a set S of (possibly overlapping) intervals with the following operations:

Insert(I, S): Insert interval I into S.

Delete(I, S): Delete interval I from S.

OverlapSearch(I, S): Return an arbitrarily chosen interval of S that overlaps interval I. (Return nil if none exists.)

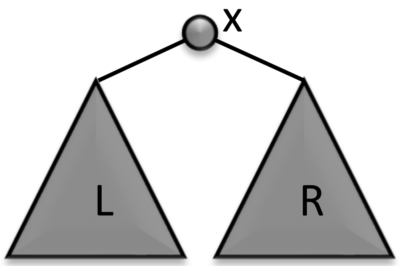
ReportAllOverlaps(I, S): Output all intervals of S that overlap interval I.

CountAllOverlaps(I, S): Output the # of intervals of S that overlap interval I.

SOLUTION:

Augment a Red-Black tree or a Splay tree.
Each node x holds an interval Int[x] ≡ [s[x],f[x]] of S.
For each node x: key[x] ≡ s[x].
So, intervals are inorder sorted by their starting point.
What augmented fields should we maintain?

OverlapSearch(I,x)



Case 1) I and Int[x] overlap: **return** x (* or Int[x] *)

Case 2) I to the left of Int[x]: $f[I] < s[x]$.

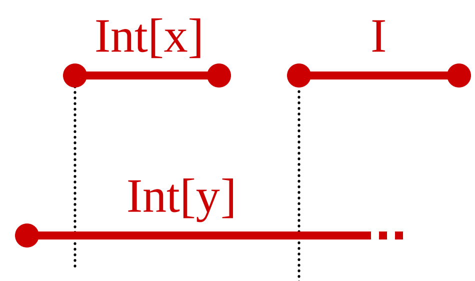
$\therefore \forall y \in R$: $f[I] < s[x] \le s[y]$

I is disjoint from x and from every interval in R.

return OverlapSearch(I,left[x])

Case 3) I to the right of Int[x]: $f[x] < s[I]$.

Where to search next?



$\therefore \forall y \in L$: $s[y] \le s[x] < s[I]$.

$\therefore \forall y \in L$: Int[y] overlap I $\Leftrightarrow f[y] \ge s[I]$.

(R may or may not have overlapping intervals.)

Define: $LAST(L) = \max \{ f[y] \mid y \in L \}$.

$\therefore$ ($\exists y \in L$: Int[y] overlap I) $\Leftrightarrow LAST(L) \ge s[I]$.

if $LAST(L) \ge s[I]$ **then** OverlapSearch(I,left[x])

else OverlapSearch(I,right[x])

factor out

Interval Tree Example

As an augmented Red-Black tree: