EECS 4101/5101

Prof. Andy Mirzaian

Augmenting Data Structures

TOPICS

- Augmentation
- Order Statistics Dictionary
- Interval Tree
- Overlapping Windows

References:

✂[CLRS] chapter 14

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

ORDER STATISTICS DICTIONARY

Two new (inverse) operations:

RANK(K,D): return the number of items in data set D that are ≤ key K.

SELECT(r,D): return the item in D with rank r (return nil if none exists).

Solution 1: D as an 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 ≤ K.

SELECT(r,D): See [CLRS chapter 9] or my EECS3101 LS5 or LN4.

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

RANK takes O(log n) time by binary-search.

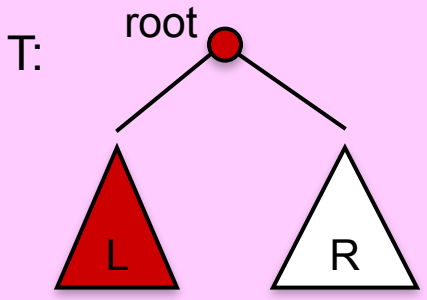
SELECT takes O(1) time by probing rank index position.

What about the dictionary operations?

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

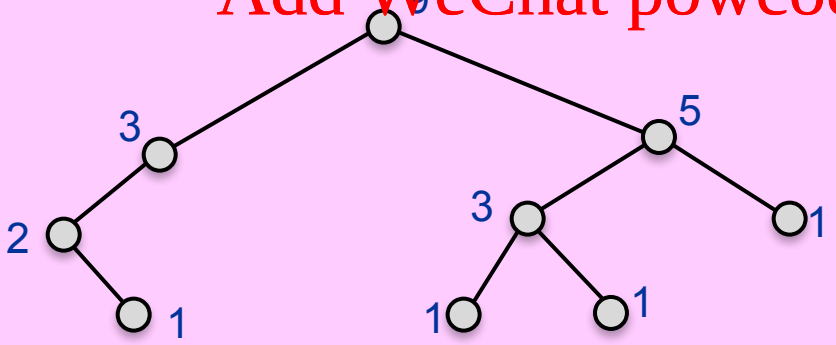
Augmenting a BST

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



Augment BST: info in each node x

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



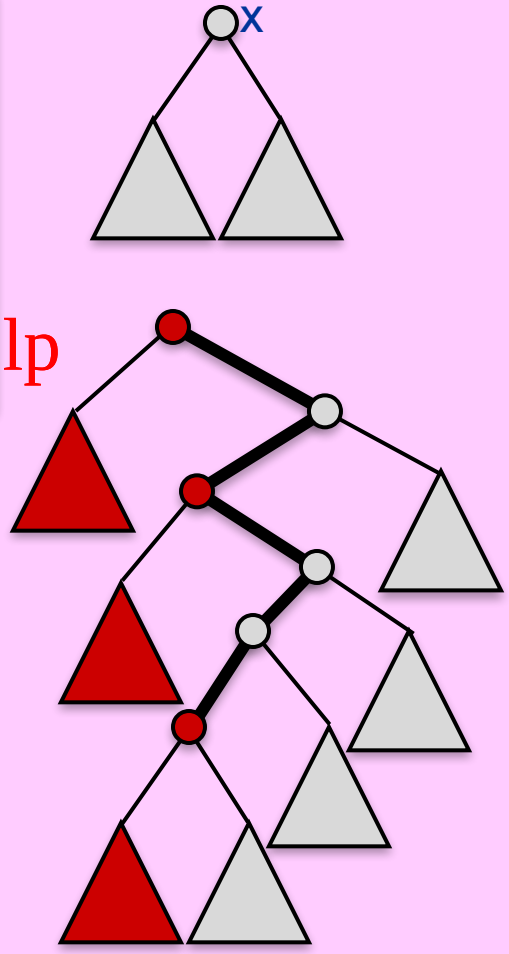
Rank of root = 1 + size[left[root]].

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
```

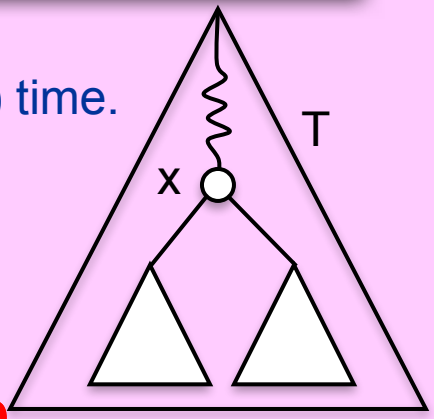


Running time = O(# nodes on the search path).

Maintain Augmented Info

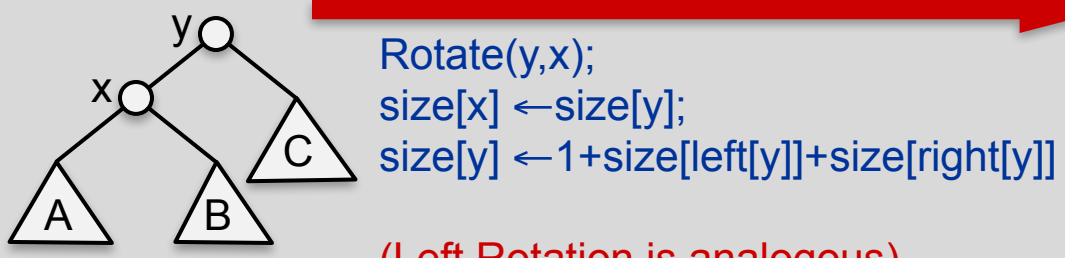
"size[.]" field can be evaluated by a local recurrence in O(1) time.

size[x] = 1 + size[left[x]] + size[right[x]], if x≠ 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:



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.

NOTE: In general, "f" may be not just one scalar field, but a record of a constant number of scalar fields.

INTERVAL TREES

Intervals on the real line

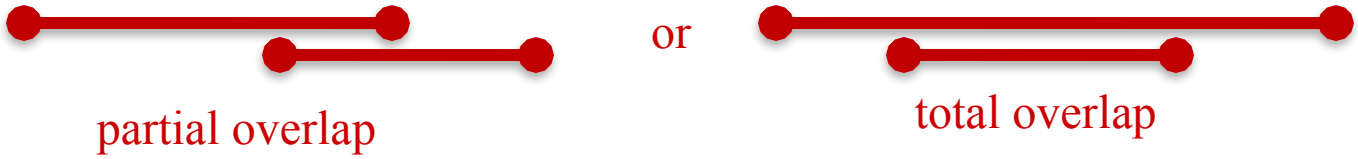
Interval I on the real line = [s[I], f[I]] (from start s[I] to finish f[I], inclusive).

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] s[Y] and f[Y] s[X].



Interval Dictionary Problem

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

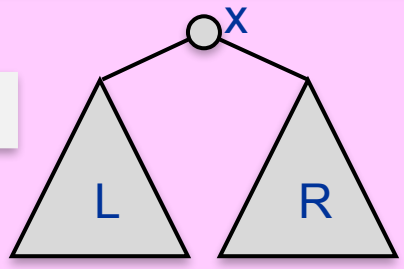
Insert(I, S):	Insert interval I into S.
Delete(I, S):	Delete interval I from S.
OverlapSearch(I, S):	Return an arbitrarily chosen interval of S that overlaps interval I. (Return nil if none exists.)
ReportAllOverlaps(I, S):	Output all intervals of S that overlap interval I.
CountAllOverlaps(I, S):	Output 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].

∴ ∀y∈R: f[I] < s[y].

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

return OverlapSearch(I,left[x])

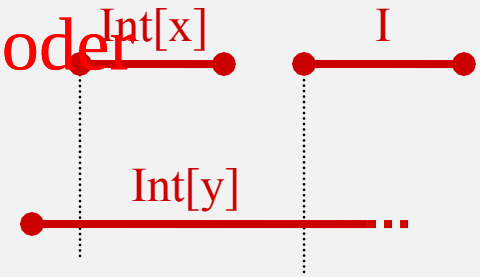
Case 3) I to the right of Int[x]: f[x] < s[I].

Where to search next?

∴ ∀y∈L: s[y] ≤ s[x] < s[I].

∴ ∀y∈L: Int[y] overlap I ⇔ f[y] s[I].

(R may or may not have overlapping intervals.)



Define: LAST(L) = max { f[y] | y∈L }.

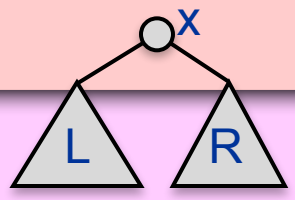
∴ (∃y∈L: Int[y] overlap I) ⇔ LAST(L) s[I].

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

else OverlapSearch(I,right[x])

factor out

OverlapSearch cases



f[x] < s[I] ∨ f[I] < s[x]

f[x] s[I] ∧ f[I] s[x]

L ≠∅ ∧ LAST[L] s[I]

L =∅ ∨ LAST[L] < s[I]

∃y∈L: f[y] = LAST[L] s[I]

∀y∈L: f[y] ≤ LAST[L] < s[I]

f[x] < s[I]

f[I] < s[x]

∀y∈L:
s[y]≤s[x]≤f[x]<s[I]≤f[I]

∀y∈R:
f[I]<s[x]≤s[y]

∃y∈L: s[y] ≤ f[I] ∧ s[I] ≤ f[y]

Int[x]∩I ≠∅

∃y∈L: Int[y]∩I ≠∅

∀y∈R : Int[y]∩I =∅

∀y∈L : Int[y]∩I =∅

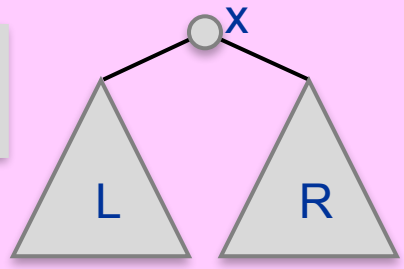
Output x

Search L

Search R

Augmented field: LAST[x]

DEFINITION:
LAST[x] = max { f[y] | y is a descendant of x, inclusive }.

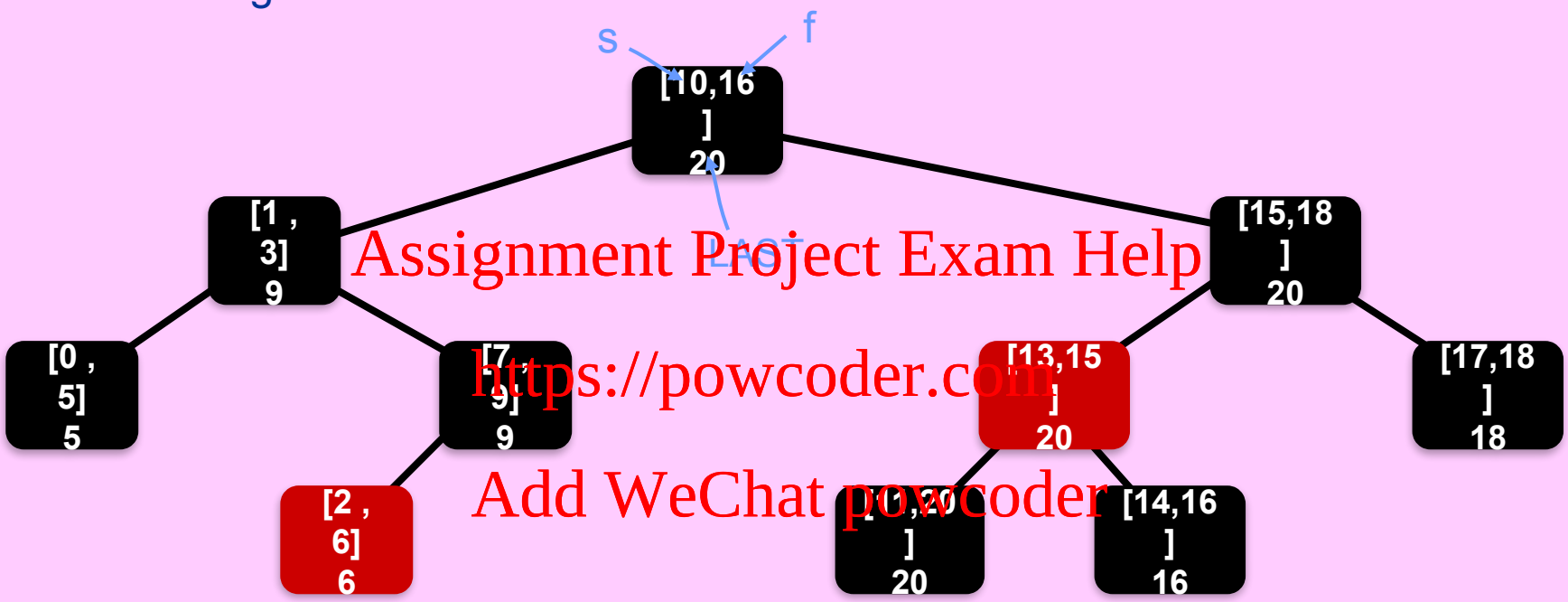


$$\text{LAST}[x] = \begin{cases} \max \begin{Bmatrix} f[x], \\ \text{LAST}[\text{left}[x]], \\ \text{LAST}[\text{right}[x]] \end{Bmatrix} & \text{if } x \neq \text{nil} \\ -\infty & \text{if } x = \text{nil} \end{cases}$$

NOTE: "LAST" is O(1) locally composable.

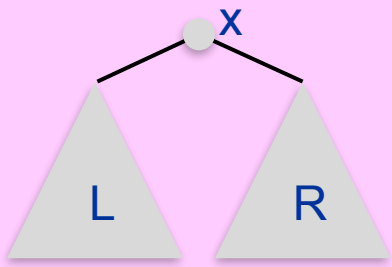
Interval Tree Example

As an augmented Red-Black tree:



OverlapSearch(I,x)

```
OverlapSearch(I,x)
    if  x = nil  then return nil
    if I overlaps Int[x]   then return x
    if left[x] ≠ nil and LAST[left[x]]  s[I]
        then return OverlapSearch(I,left[x])
        else  return OverlapSearch(I,right[x])
end
```



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

Interval Tree Complexity

THEOREM 3:
Interval trees, as augmented Red-Black trees with one interval per node and the O(1) locally composable augmented field "LAST" as described above, support the operations OverlapSearch, Insert, Delete, in O(log n) worst-case time per operation.

If we use the same augmentation on Splay trees, each of these operations takes O(log n) amortized time.

EXERCISE:

Without degrading the O(log n) time complexity of earlier operations, do:

INPUT: interval I and an Interval tree T (possibly augmented or redefined).

a) Design and analyze an implementation of ReportAllOverlaps(I,T) that takes O(R + log n) time, where n is the number of intervals in T, and R is the output size, i.e., the number of intervals in T that overlap I.

b) Show CountAllOverlaps(I,T) can be done in O(log n) time.

An Application of Interval Trees

Overlapping Windows

PROBLEM:

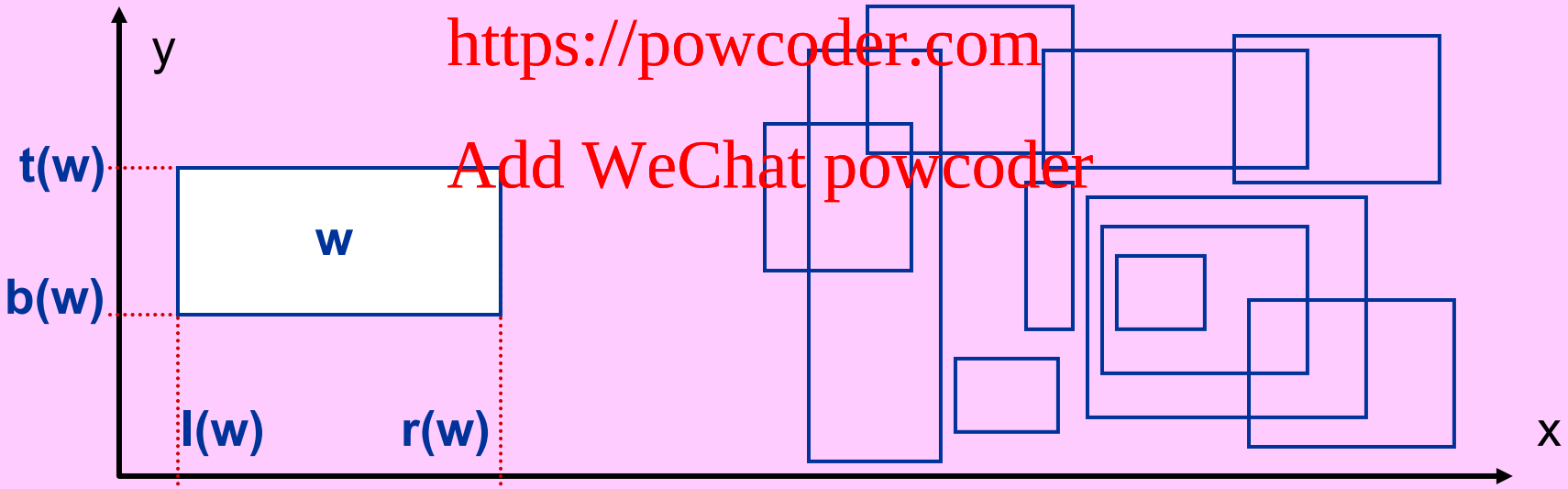
INPUT: We are given a set $S = \{w_1, w_2, \ldots, w_n\}$ of n axis-parallel rectangular windows; $w = [l(w), r(w), b(w), t(w)]$, for $w \in S$.

OUTPUT MODE:

Reporting: Report all (R) pairs of overlapping input windows.

Counting: How many pairs of overlapping input windows? (=R)

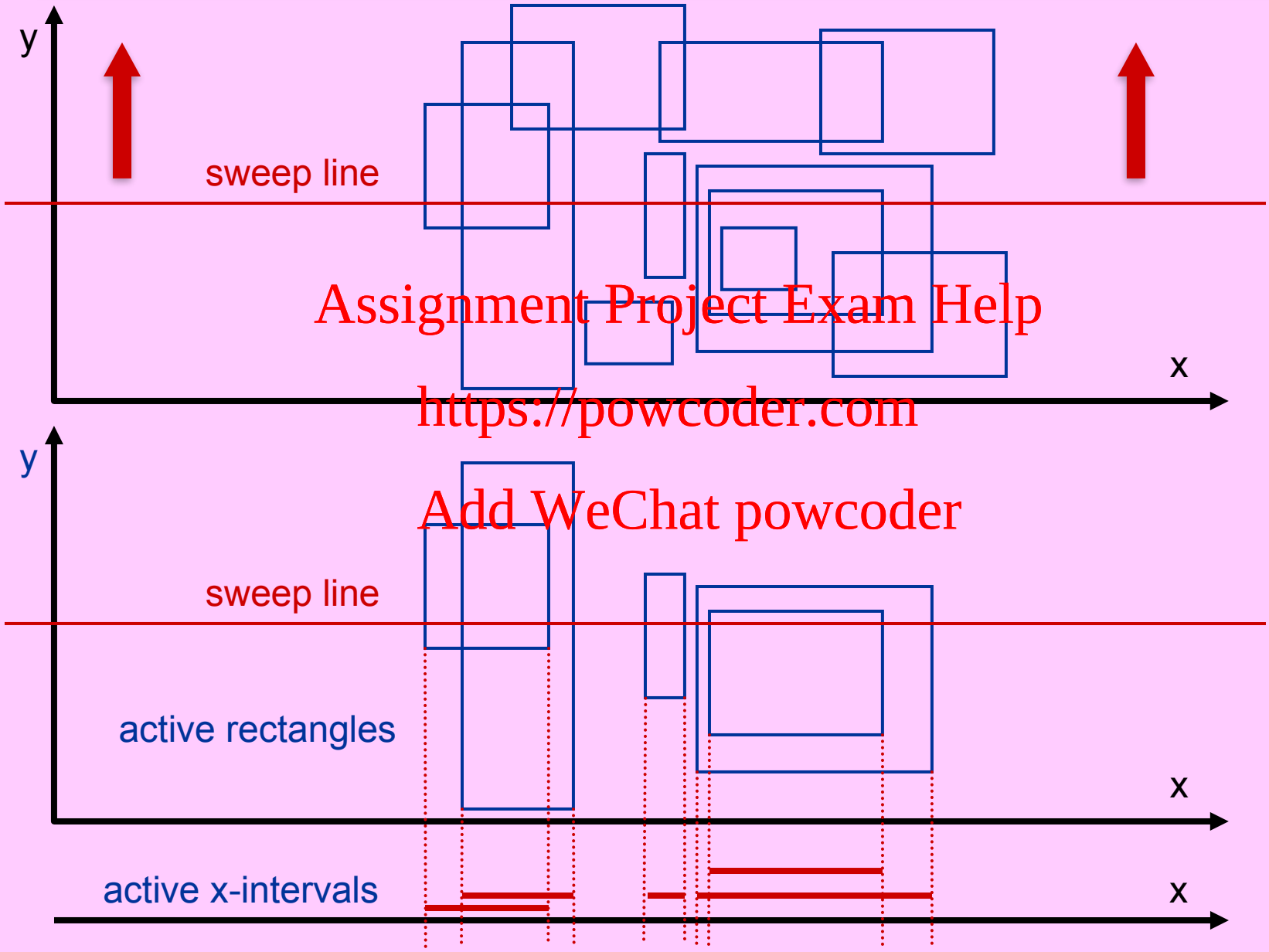
Disjointness: Is there an overlapping pair of input windows? (R>0?)



Can we do better than the obvious $O(n^2)$ time brute-force solution?

YES: Counting & Disjointness in O(n log n) time,
Reporting in O(R + n log n) time.

Plane Sweep Method



Sweep Schedule & Sweep Status

Sweep Schedule is the 2n events (corresponding to the bottom and top y coordinates of the n windows in input set S) in ascending order.
We can pre-sort these 2n events in O(n log n) time.

Sweep Status is the dictionary of the active windows maintained in an interval tree T. For each window w, its x-interval is [l(w), r(w)], starting at l(w) and finishing at r(w). Dictionary size: $|T| \leq n$.

Operations:
Insert (w, T): insert (x-interval of) activated window w into T.
Delete(w,T): delete (x-interval of) deactivated window w from T.
OverlapSearch(w,T): return a window in T that overlaps w.
CountAllOverlaps(w,T): return count of # windows in T that overlap w.
ReportAllOverlaps(w,T): report all windows in T that overlap w.

All but the last operation take O(log n) time each.
ReportAllOverlaps takes $O(R_w + \log n)$ time, where R_w is the output size.

$$\sum_{w \in S} O(R_w + \log n) = O((\sum_{w \in S} R_w) + n \log n) = O(R + n \log n).$$

Plane Sweep Algorithm

ALGORITHM ReportOverlappingPairs (S,n)

Step 0: Initialize sweep status interval tree T:

$T \leftarrow \emptyset$

Step 1: Initialize sweep schedule of events E:

$E = \{ \langle b(w), w \rangle \mid w \in S \} \cup \{ \langle t(w), w \rangle \mid w \in S \}$.

Sort E in ascending order of the first component.

[Tie breaking rule: if b(w)=t(w'), b has precedence over t]

Step 2: Sweep the plane according to sweep schedule E:

```
for each ⟨y,w⟩ ∈ E, in ascending order of y do
        if y = b(w)
        then do
                ReportAllOverlaps(w,T)
                Insert(w,T)
        end-then
        else Delete(w,T)    (* y = t(w)  *)
end-for
end
```

Running time = O(R + n log n).

Exercises

1. **Range-Search-Counting:** Given a Red-Black tree T and a pair of keys a and b, a < b (not necessarily in T), we wish to output the number of items x in T such that a ≤ key[x] ≤ b. Augment T so that, without degrading the time complexities of the dictionary operations Search, Insert, Delete, operation RangeSearchCount can be done in O(log n) time in the worst case, where n is the number of items in T.

2. **Dictionary with Average and Median:** Operations **Average(K,T)** and **Median(K,T)** return, respectively, the average and median of those items in dictionary T that are ≤ the given key K.
   (a) Describe the augmented dictionary data structure to support these new operations.
   (b) Describe the algorithms for Average and Median, and explain their running times.
   (c) What are the required modifications to Search, Insert, Delete? Are their running times affected?

3. **[CLRS, Exercise 14.1-7, p. 345]** We wish to output the number of inversions in a given array A[1..n] of n real numbers. Show how to use Order Statistics Dictionary (OSD) to solve this problem in O(n log n) time.

4. **Selection in a Pair of Order Statistics Dictionaries:**
   As we learned in the course, an Order Statistics Dictionary (OSD) can be implemented efficiently by an augmented Red-Black Tree in which each node x has the extra field size[x] which stores the number of descendants of node x. We are given two OSDs S and T with n keys in total, and a positive integer k, k ≤ n. The elements within an OSD have distinct keys, but the elements in the two OSDs may have some equal keys. The problem is to find the k^{th} smallest key among the n keys stored in S and T (without changing S and T).
   Design and analyze an algorithm that solves this problem in worst-case O(log n) time.

5. **[CLRS, Exercise 14.1-8, p. 345]** Consider n given chords on a circle, each defined by its two endpoints. Describe an O(n log n) time algorithm to determine the number of pairs of chords that intersect inside the circle. [For instance, if the n chords are all diameters that meet at the center, the correct answer would be n(n-1)/2.] For simplicity, assume that no two chords share an end-point.

6. **[CLRS, Problem 14-1, p. 354]** We wish to keep track of a **Point of Maximum Overlap (PMO)** in a dynamic set T of intervals, i.e., a point on the real line that is overlapped by the largest number of intervals in data set T.

(a) Show that there is always a PMO that is an end-point of an interval in T (if not empty).

(b) Design and analyze an augmented interval data structure T that efficiently supports the old operations Insert, Delete, and the new operation **FindPMO(T)**. The latter returns a point of maximum overlap with the intervals in the data set T.
[Hint: Keep a Red-Black tree of all endpoints. Associate a value of +1 with each left end-point, and associate a value of –1 with each right end-point. Augment each node of the tree with extra information to maintain the PMO.]

7. **[CLRS, Exercise 14.3-6, p. 354] Dynamic Closest Pair:** Show how to maintain a dynamic data set Q of numbers that supports the operation **ClosestPair**, which returns the pair of numbers in Q with minimum absolute value difference. For example, if Q = {1, 5, 9, 15, 18, 22}, then ClosestPair(Q) returns (15,18), since |18 – 15| = 3 is the minimum difference among any distinct pair of numbers in Q.

Design and analyze an augmented data structure to support such a data set with the operations Search, Insert, Delete, and ClosestPair as efficiently as possible.

8. **Closest Bichromatic Pair:** We want to maintain a dynamic data set Q of items. Each item has two attributes: a **key**, which is a real number, and a **colour**, which is either **blue** or **green**. In addition to the usual dictionary operations, we wish to support a new operation called **Closest Bichromatic Pair (CBP).** CBP(Q) returns a pair of items in Q that have different colours and minimum possible absolute value difference in their key values. (Note, we are referring to the values of their keys, not their index or position in the data set.) CBP returns nil if all items in Q have the same colour.
Design and analyze an augmented data structure to support such a data set with the operations Search, Insert, Delete, and CBP as efficiently as possible.

9. **[CLRS, Exercise 14.3-3, p. 353] EarliestOverlapSearch(I,T):** We are given an interval tree T and an interval I. For simplicity, assume all intervals have distinct starting points. Design and analyze an efficient implementation of EarliestOverlapSearch(I,T), which returns the interval with minimum starting time among those intervals in T that overlap interval I. It returns nil if no interval in T overlaps I.

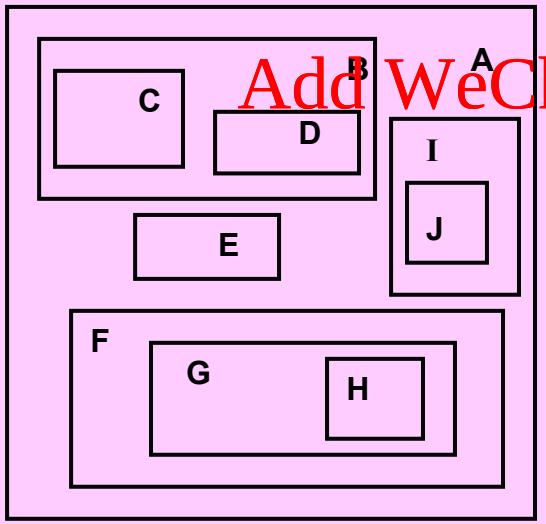
10. **[CLRS, Exercise 14.3-4, p. 354] ReportAllOverlaps(I,T):**
(a) Show that ReportAllOverlaps(I,T) can be done in O(min{n, (R+1)log n}) time, where n is the number
of intervals in Interval tree T, and R is the output size, i.e., the number of intervals in T that overlap interval I. You should not redefine or augment Interval trees in any way. However, the operation may

temporarily modify T during its process (e.g., delete an item, then reinsert it back into T).
(b) Answer part (a) but with ReportAllOverlaps not modifying T in any way, not even temporarily.
(c) [Extra credit and optional:] Improve the time complexity of ReportAllOverlaps(I,T) to O(R + log n), without degrading the time complexities of the other operations. [You may redefine the structure.]

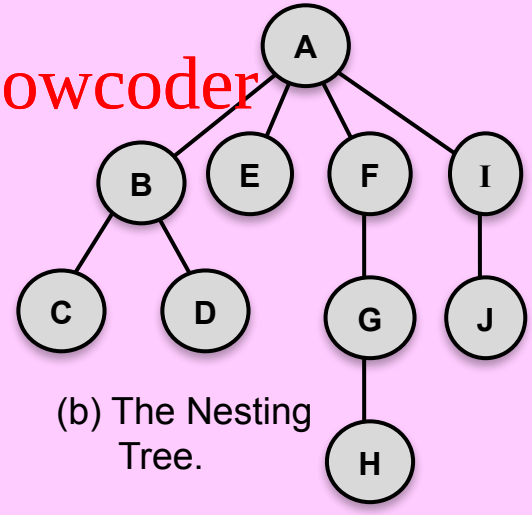
11. **Nested Rectangles:** We are given a set R= {R_1 , ... , R_n} of n axis-parallel rectangles in the plane. Each rectangle is given by its two pairs of extreme x and y coordinates. One of these rectangles is the bounding box and encloses all the remaining n-1 rectangles in R. All rectangles in R have disjoint boundaries. So, for any pair of rectangles R_i and R_j in R, either they are completely outside each other, or one encloses the other. Therefore, we may have several levels of nesting among these rectangles. (See Fig. (a) below.) We say R_i **immediately encloses** R_j, if R_i encloses R_j and there is no other rectangle R_k in R such that R_i encloses R_k and R_k encloses R_j.

(a) The **immediate enclosure** relationship on R defines a directed graph, where we have a directed edge from R_i to R_j iff R_i immediately encloses R_j. Prove this digraph is indeed a rooted tree (called the **Nesting Tree** of R). That is, you need to prove that the said digraph is acyclic, and every node has in-degree 1 except for one node that has in-degree 0. What rectangle in R corresponds to the root of that tree? (As an illustration, see Fig. (b) corresponding to Fig. (a).)

(b) Design and analyze an efficient algorithm that given the set R, constructs its corresponding Nesting Tree by **parent** pointers. That is, outputs the parent array p[1..n] such that p[j] = i means R_i immediately encloses R_j. Carefully explain the correctness and running time of your algorithm.



(a) A set of axis parallel rectangles with disjoint boundaries.



(b) The Nesting Tree.

END