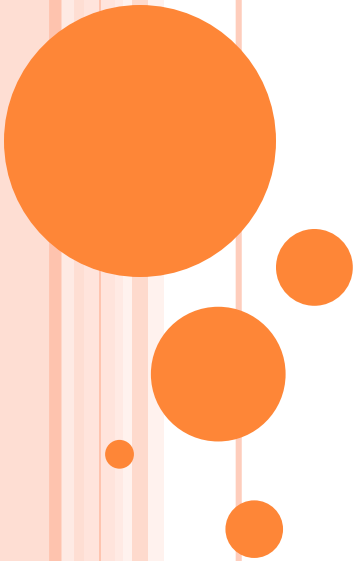
Parallel Search Trees

Yihan Sun

CMU 15-853



Why trees?

- Very important data structure in almost all areas in computer science
- Maintain ordering on the keys – search tree
  - Build index for databases or search engines
  - Support ordered sets and maps, priority queues, …, useful as a subroutine in many algorithms
  - Range searching: useful in database systems and computational geometry algorithms
  - ……

Balance Search Trees (BSTs)

- Many algorithms on trees have cost proportional to tree height
- Need **balancing schemes** to keep tree nearly balanced
  - Balancing scheme: a set of invariants on trees
  - AVL trees, weight-balanced trees, splay trees, treaps, B trees, ……
- In this lecture we use balanced binary search trees
- Not limited to any specific balancing schemes. The algorithms/bounds hold at least for:
  - AVL trees, red-black trees, weight-balance trees and treaps

What we want to do with trees

- Searching
- The first or last entry, the entry of certain rank, …
- Insertion and deletion

Cost is order of the tree height, $O(\log n)$ for balance trees

- Construction
- Filter, map-reduce, …
- Bulk insertion and deletion
- Merging two trees (or getting the common elements)

Can be highly-parallelized

Goal: work-efficient (optimal) and polylogarithmic depth

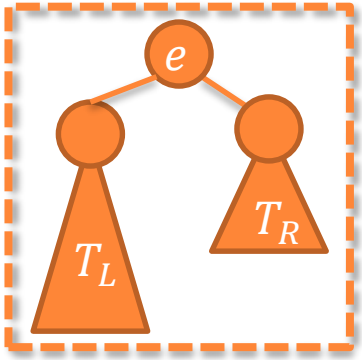
In this lecture

- Parallel algorithms on trees
- Applications that can be solved using assorted tree structures

- Next: Preliminaries

Preliminaries

- $T = \texttt{join}(T_L, e, T_R)$: connects two trees T_L and T_R with e, but get the result balanced

$T =$ 

in-order(T)=[in-order(T_L), e, in-order(T_R)]

Preliminaries

- $T = \texttt{join}(T_L, e, T_R)$: connects two trees T_L and T_R with e, but get the result balanced
- $T = \texttt{join2}(T_L, T_R)$: similar as join but without the middle entry

Both can finish in $O(\log n)$, where n is the larger tree size (for join a tighter bound is $O(|\log|T_L| - \log|T_R||)$)

- $\langle T_L, b, T_R \rangle$= split$(T, k)$
  - T_L contains all keys in T smaller than k
  - T_R contains all keys in T greater than k
  - A bit b indicating whether $k \in T$

costs $O(\log n)$, where n is the size of T

- Next: simple parallel algorithms on trees

Parallel Search trees

- It is very easy to design parallel algorithms on trees using divide-and-conquer scheme
  - Recursively deal with two subtrees in parallel
  - Combine results of recursive calls and the root
  - Usually gives polylogarithmic bounds on depth

```
func(T, …) {
    if (T is empty)
        return base_case;
    M = do_something(T.root);
    in parallel:
       L=func(T.left, …);
       R=func(T.right, …);
    return combine_results(L, R, M, …)
}
```

Map and reduce

- Maps each entry on the tree to a certain value using function `map`, then reduce all the mapped values using `reduce` (with identity `identity`).
- Assume `map` and `reduce` both have constant cost.

```
map_reduce(Tree T, function map, function reduce,
value_type identity) {
    if (T is empty) return identity;
    M=map(t.root);
    in parallel:
       L=map_reduce(T.left, map, reduce, identity);
       R=map_reduce(T.right, map, reduce, identity);
    return reduce(reduce(L, M, R));
```

$O(n)$ work and $O(\log n)$ depth

Filter

- Select all entries in the tree that satisfy function f
- Return a tree of all these entries

```
filter(Tree T, function f) {
    if (T is empty) return an empty tree;
    in parallel:
       L=filter(T.left, f);
       R=filter(T.right, f);
    if (f(T.root)) return join(L, T.root, R);
    else return join2(L, R);
```

$O(n)$ work and $O(\log^2 n)$ depth

Construction

```
T=build(Array A, int size) {
    A′=parallel_sort(A, size);
    return build_sorted(A′, s);
}

T=build_sorted(Arrary A, int start, int end) {
    if (start == end) return an empty tree;
    if (start == end-1) return singleton(A[start]);
    mid = (start+end)/2;
    in parallel:
      L = build_sorted(A, start, mid);
      R = build_sorted(A, mid+1, end);
    return join(L, A[mid], R);
```

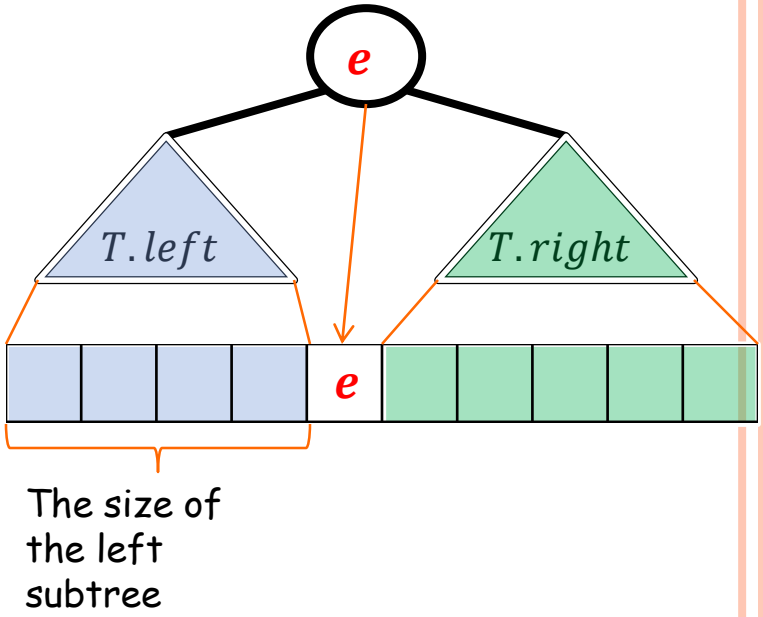
$O(n)$ work and
$O(\log n)$ depth

$O(n \log n)$ work and $O(\log n)$ depth,
bounded by the sorting algorithm

Output to array

- Output the entries in a tree T to an array in its in-order
- Assume each tree node stores its subtree size (an empty tree has size 0)

```
to_array(Tree T, array A, int offset) {
    if (T is empty) return;
    A[offset+T.left.size] = get_entry(T.root);
    in parallel:
       to_array(T.left, A, offset);
       to_array(T.right, A, offset+T.left.size()+1);
```



$O(n)$ work and $O(\log n)$ depth

Brief Summary

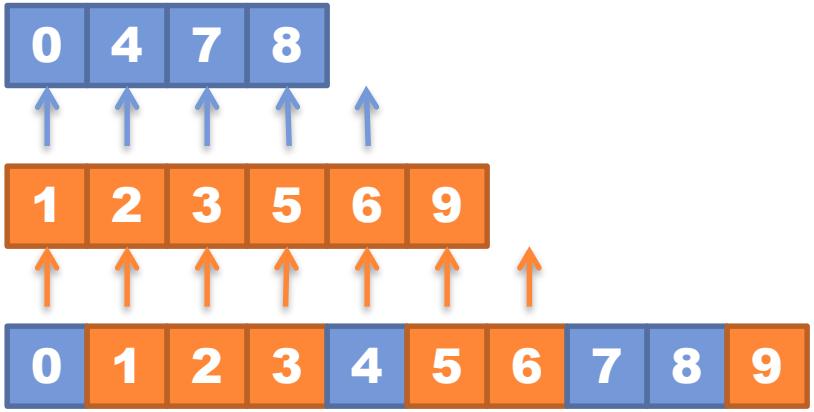
- Parallel algorithms on trees
  - Polylogarithmic depth
- Takeaway: design parallel divide-and-conquer algorithms on trees

- Next: tree-tree operations: union, intersection, difference
  - Combine two indexes in database
  - Subroutine in some applications, e.g., the range tree
  - ……

  In this lecture: lower bound, a divide-and-conquer algorithm, the cost analysis

Merging Two Trees of Size n and m (n≥m)

- Solution 1: flatten trees into arrays, merge with moving pointers:

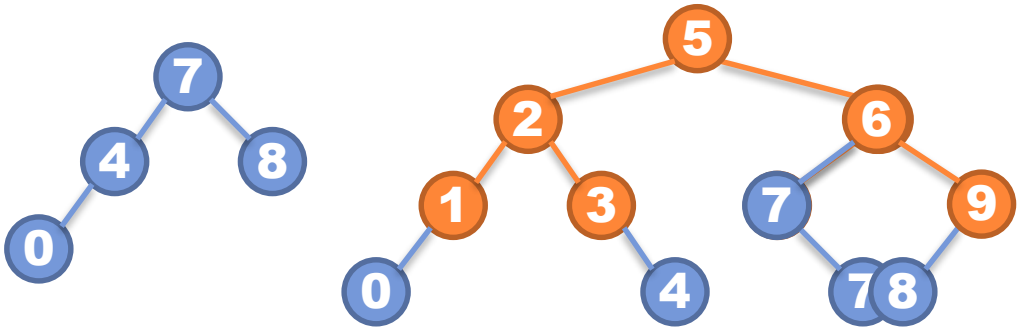


Cost: $O(m+n)$

What if $n \gg m$?

$O(n)$?

- Solution 2: insert the entries in the smaller tree into the larger tree



Cost: $O(m \log n)$

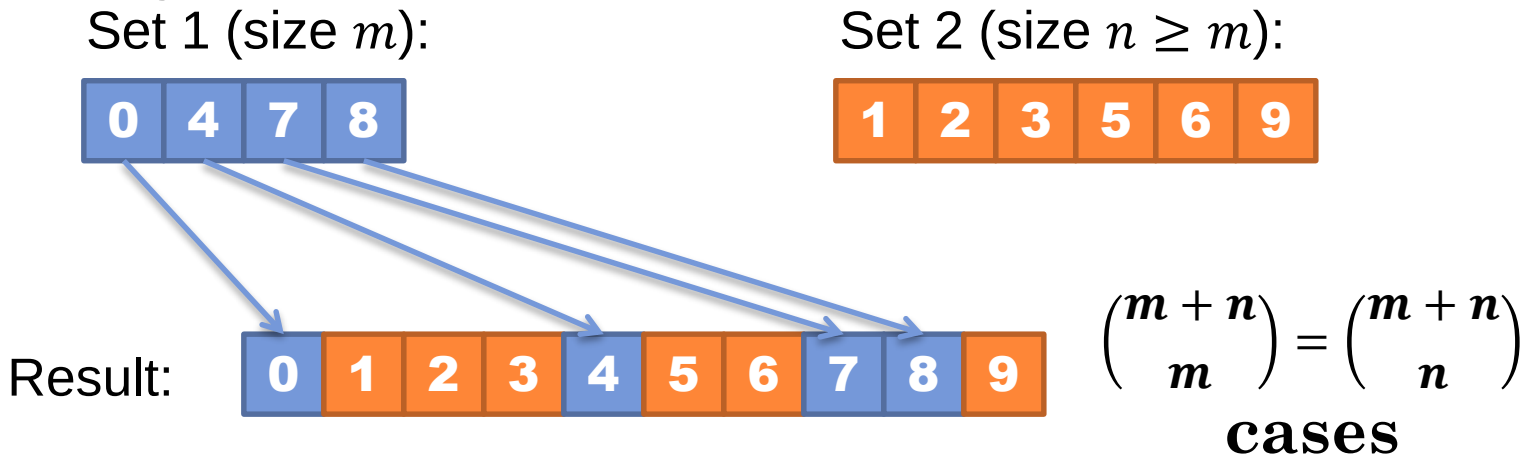
What if $n = m$?

$O(n \log n)$?

- **What is the minimum cost?**

The Lower Bound of Merging Two Ordered Sets

- Choose n slots for the elements in the first set among all $m + n$ available slots in the final result.



- **Lower bound:** $\log_2 \binom{m+n}{m} = \Theta\left(m \log\left(\frac{n}{m} + 1\right)\right)$

The Lower Bound of Merging Two Ordered Sets

- **The lower bound**

$$\boldsymbol{O\left(m \log\left(\frac{n}{m}+1\right)\right)}$$

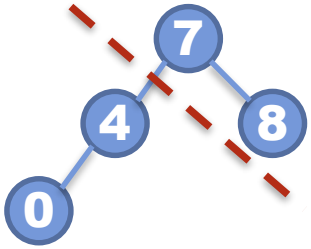
- When $m = n$, it is $O(n)$
- When $\mathrm{n} \gg m$, it is about $O(m \log n)$ (e.g., when $m = 1$, it is $O(\log n)$)

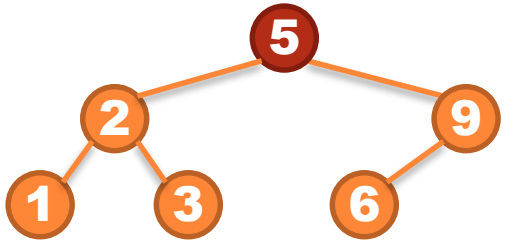
- Can we give an algorithm achieving this bound?

The Union Function

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
       (L1, b, R1) = split(T1, k2);
       TL = union(L1, L2) || TR = union(R1, R2);
       join(TL, k2, TR)
```

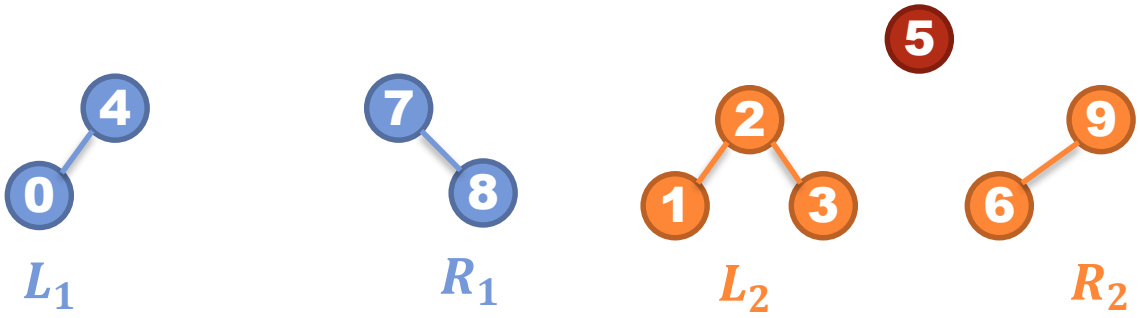
Base case





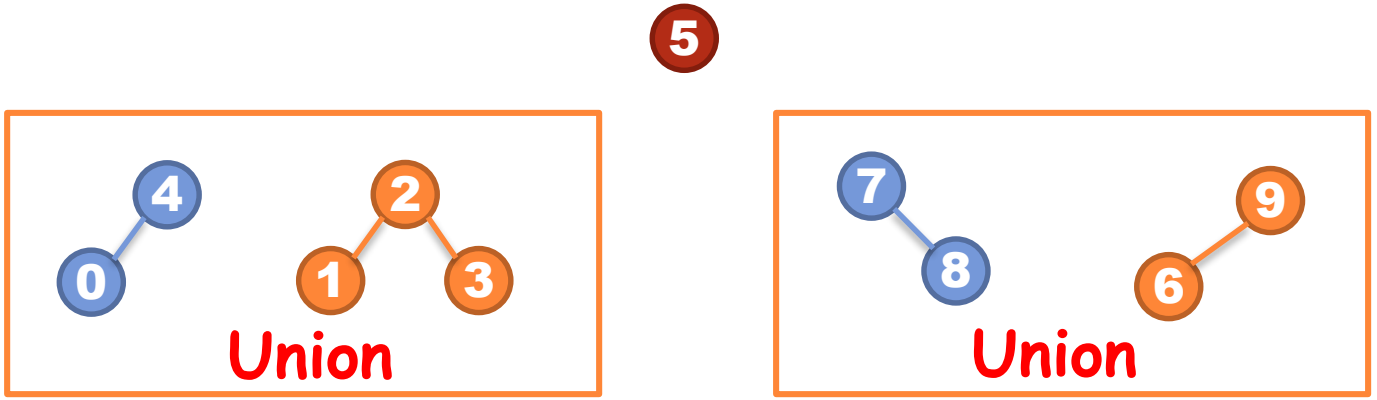
The Union Function

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```



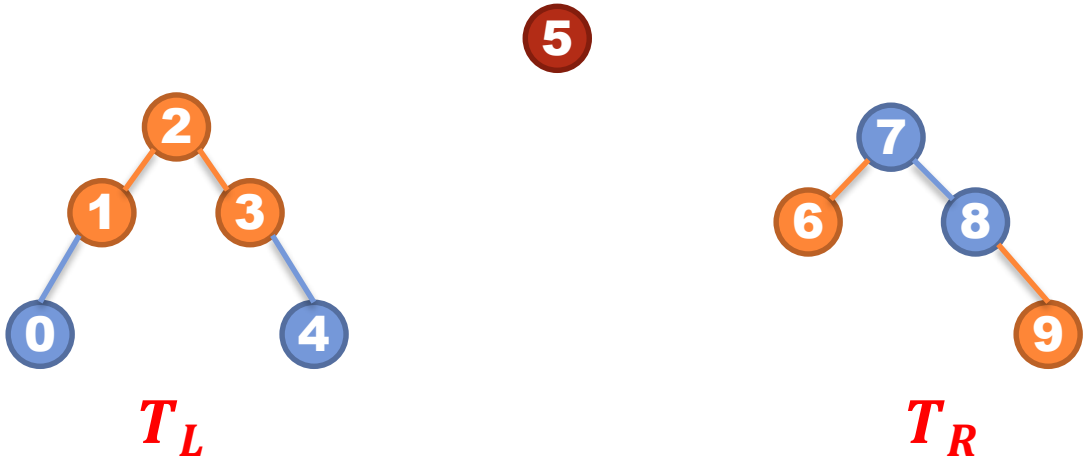
The Union Function

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```



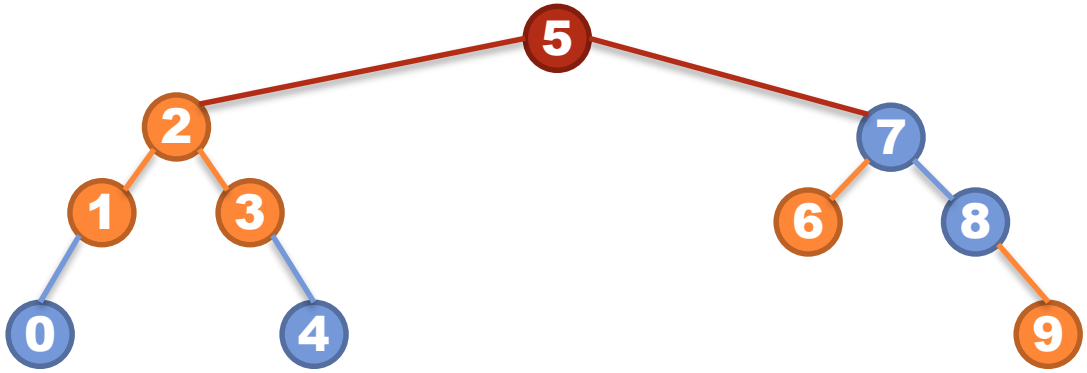
The Union Function

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```



The Union Function

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```



Similarly we can implement intersection and difference.

The Cost

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```

Theorem 1. For AVL trees, red-black trees, weight-balance trees and treaps, the above algorithm of merging two balanced BSTs of sizes m and n ($m \leq n$) have $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$ work and $O(\log m \log n)$ depth (in expectation for treaps).

The Cost

```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```

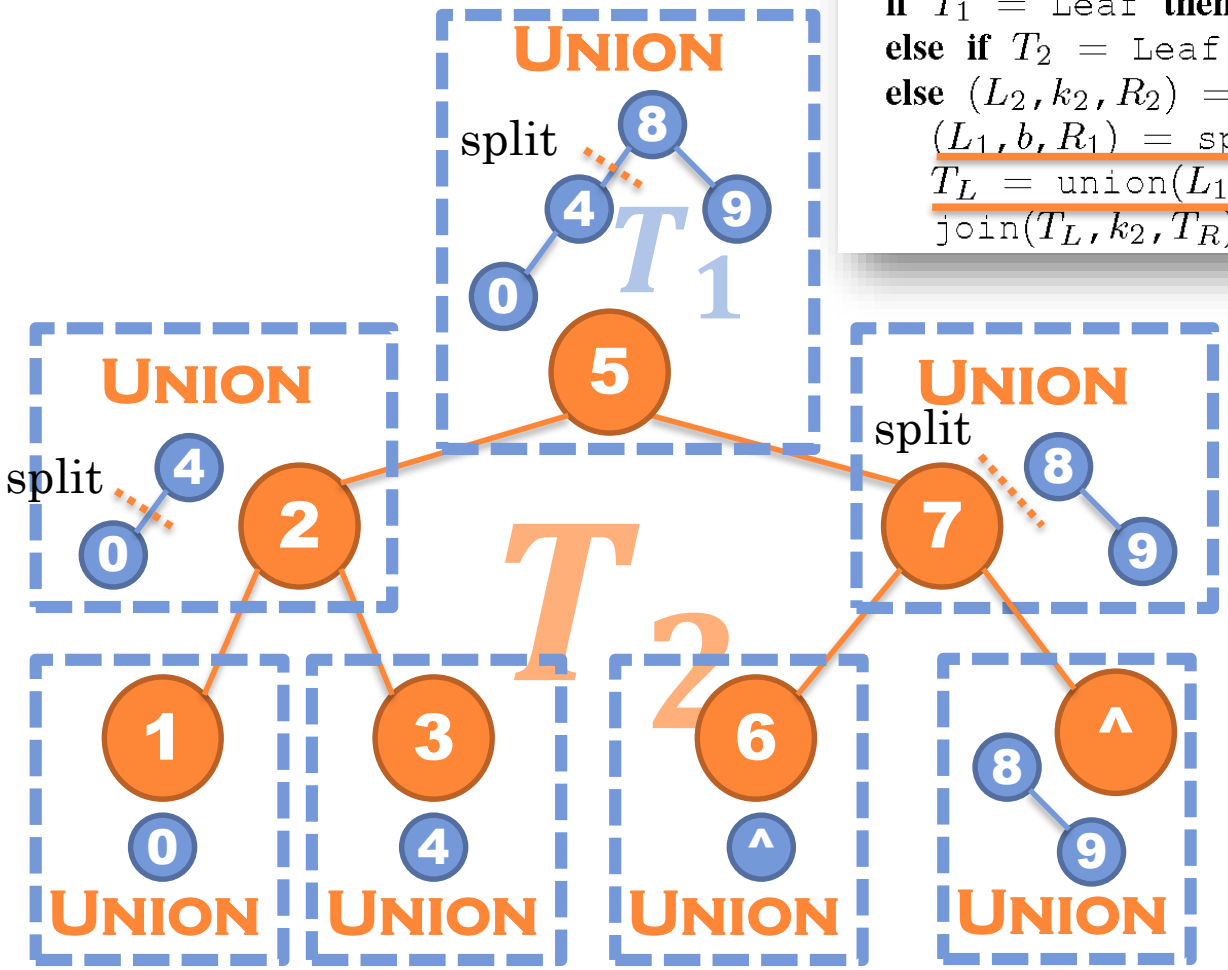
Lemma 1. The Join work can be asymptotically bounded by its corresponding Split.

Lemma 2. Split a tree T of size n costs time $O(\log n)$.

The depth is $O(\log m \log n)$

h_1 steps to reach the base case, $O(h_2)$ cost for each split

The Split Work

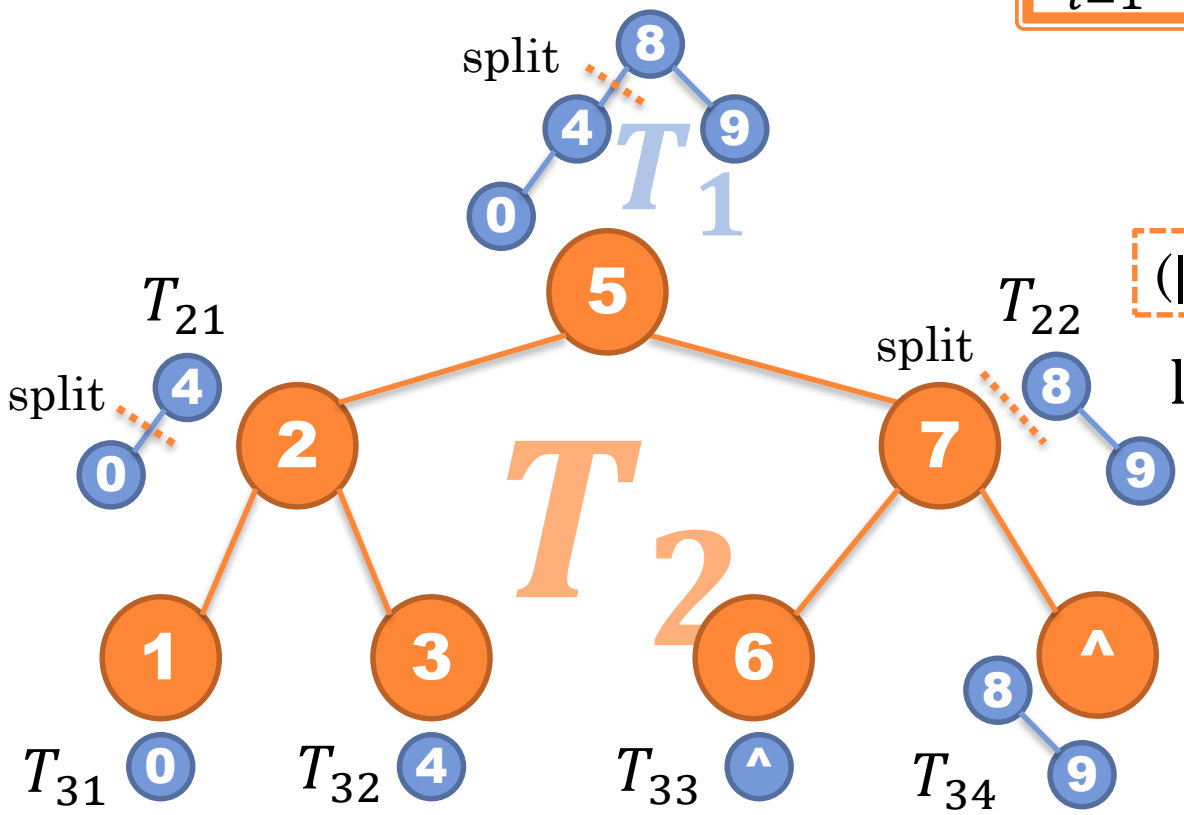



```
union(T1, T2) =
  if T1 = Leaf then T2
  else if T2 = Leaf then T1
  else (L2, k2, R2) = expose(T2);
    (L1, b, R1) = split(T1, k2);
    TL = union(L1, L2) || TR = union(R1, R2);
    join(TL, k2, TR)
```

The Split Work

Concavity:

$$\sum_{i=1}^{k} a_i \le k \log \frac{\sum a_i}{k}$$



$$\log |T_1| = \log n$$

($|T|$: the size of tree T)

$$\log |T_{21}| + \log |T_{22}| \le 2 \log \frac{n}{2}$$

……

$$\log |T_{31}| + \log |T_{32}| + \log |T_{33}| + \log |T_{34}| \le 4 \log \frac{n}{4}$$

$$\log |T_{h1}| + \ldots + \log |T_{h2^h}| \le 2^h \log \frac{n}{2^h}$$

$$\log n + 2 \log \frac{n}{2} + 4 \log \frac{n}{4} \ldots + 2^h \log \frac{n}{2^h} = O\left(m^{\mathbf{c}} \log\left(\frac{n}{m} + 1\right)\right)$$

$\mathbf{c} \log_2 m$ terms (If T_2 is perfectly balanced)

The height of T_2 $h = O(\log m)$

The Split Work

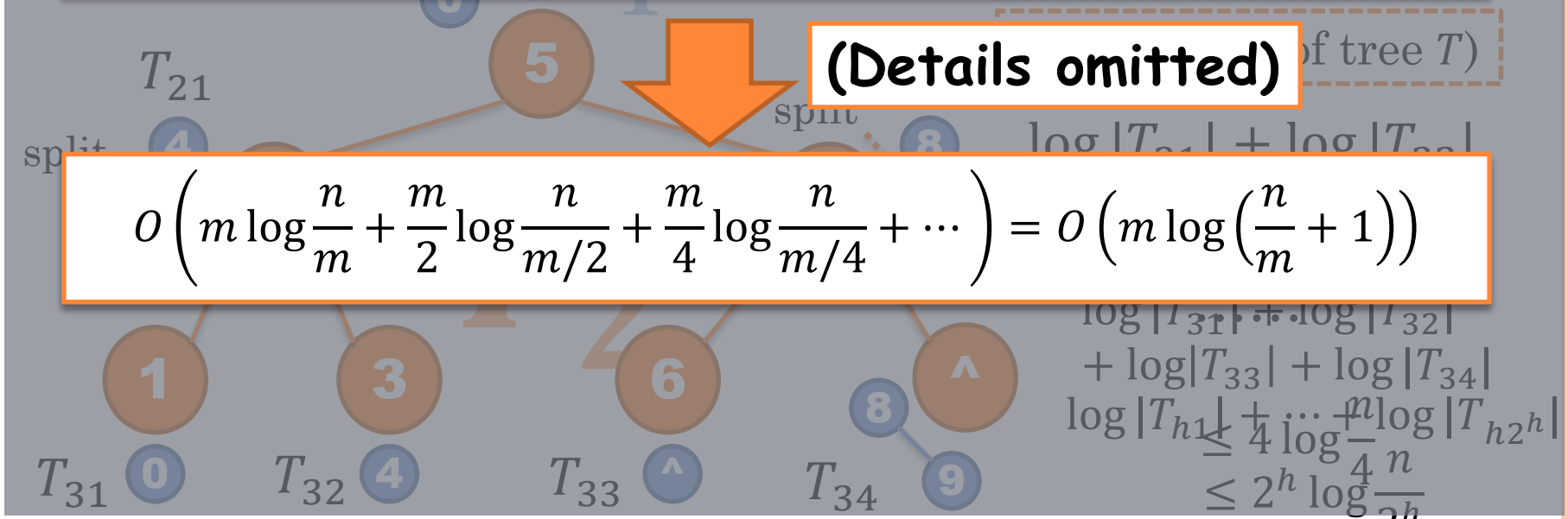
Concavity:

$$\sum_{i=1}^{k} a_i \le k \log \frac{\sum a_i}{k}$$

Lemma 3. In a tree of size N, there are at most $\frac{N}{c^{\lceil i/2 \rceil}}$ nodes in level $-i$.

(Details omitted)

$$O\left(m \log \frac{n}{m} + \frac{m}{2} \log \frac{n}{m/2} + \frac{m}{4} \log \frac{n}{m/4} + \cdots\right) = O\left(m \log\left(\frac{n}{m} + 1\right)\right)$$



$$\log n + 2 \log \frac{n}{2} + 4 \log \frac{n}{4} \ldots + 2^h \log \frac{n}{2^h} = O\left(m^{\mathbf{c}} \log\left(\frac{n}{m} + 1\right)\right)$$

$\mathbf{c} \log_2 m$ terms (If T_2 is perfectly balanced)

The height of T_2 $h = O(\log m)$

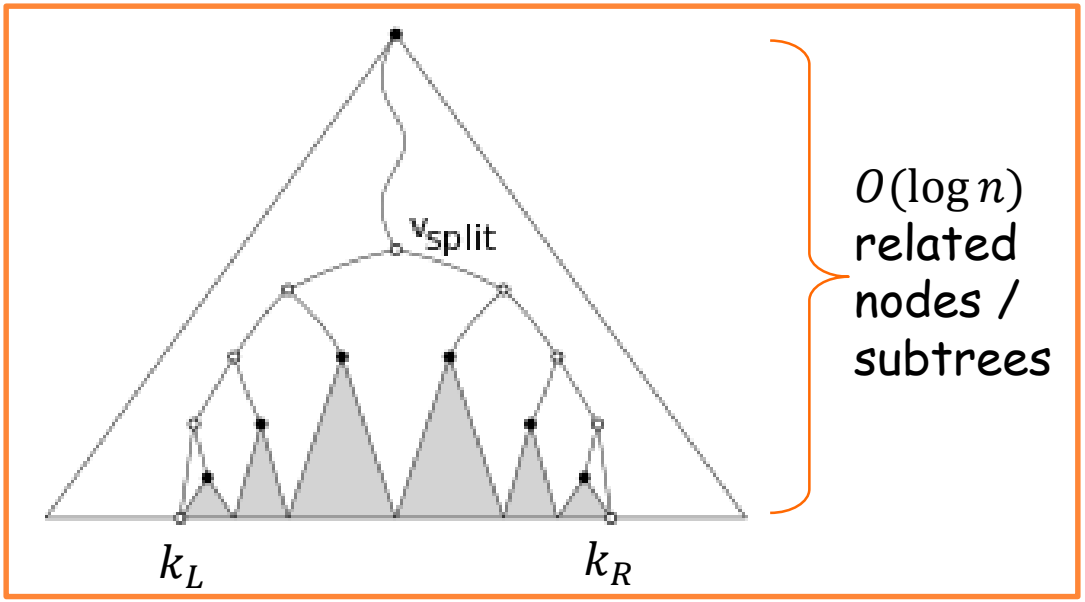
Brief Summary

- Takeaways:
  - The lower bound of merging two ordered structures
  - A work-optimal and polylog-depth parallel algorithms to merge two trees
  - A useful trick in analysis: layering bottom-up

- Next: augmentations on trees and range queries

Range query (1D)

- Report all entries in key range $[k_L, k_R]$.
- Get a tree of them: $O(\log n)$ work and depth
- Flatten them in an array: $O(k + \log n)$ work, $O(\log n)$ depth

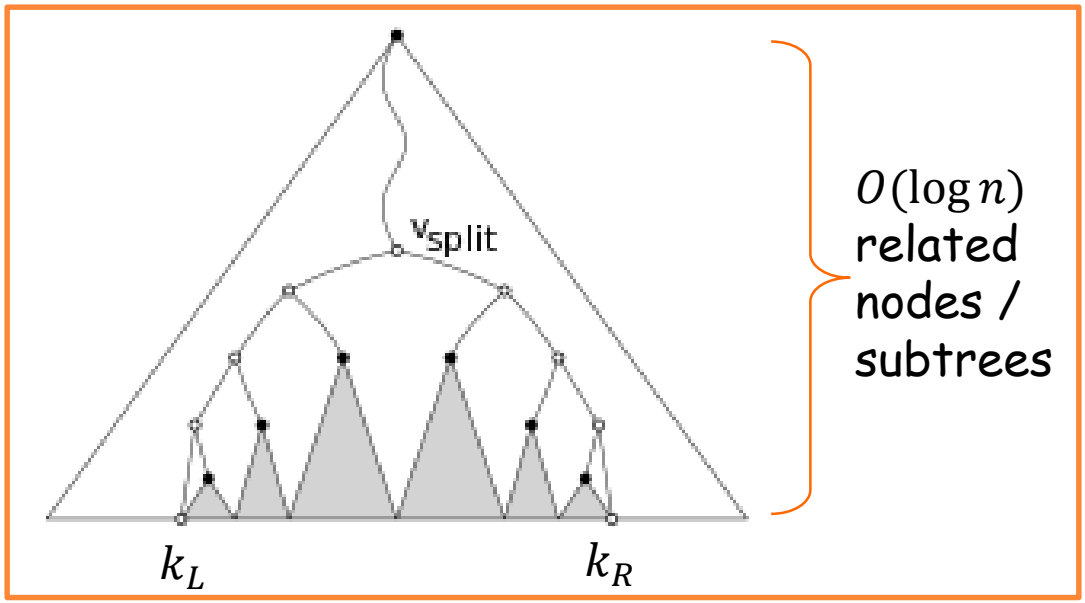


Augmentations

- In practice, most problems can be solved with a “textbook” data structure with some augmentations
- Augmented trees: in each of the tree node, store some additional information about the whole subtree rooted at it.
  - The size, sum of values, the maximum value, ……
  - Another search tree, a heap, ……
- Efficiently maintain this augmented value in your algorithms.

Augmentations for range search

- If we want to fast report the “range sum” of any associative operations, augmentations can help
  - The sum of all values in a key range, the maximum value in the key range, ……
- Augment each tree node with the partial sum in its subtree

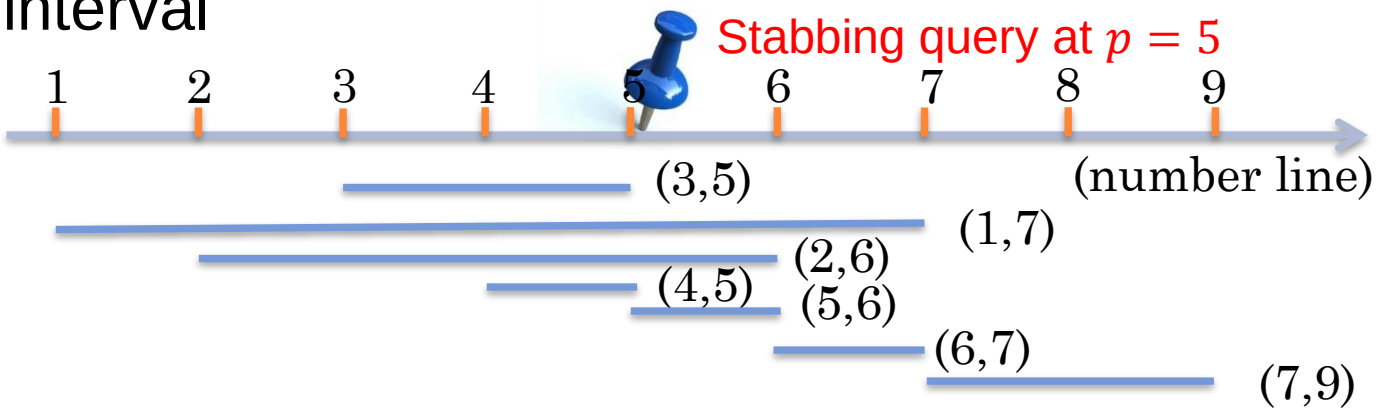


Applications

- 1D stabbing query
- 1D stabbing query: Interval tree
- 2D range query: range tree
- 2D 3-sided query: priority search trees

1D stabbing query

- Given a set of intervals $[l_i, r_i]$ on the number line, and a specific point p, determine whether p is covered by any interval



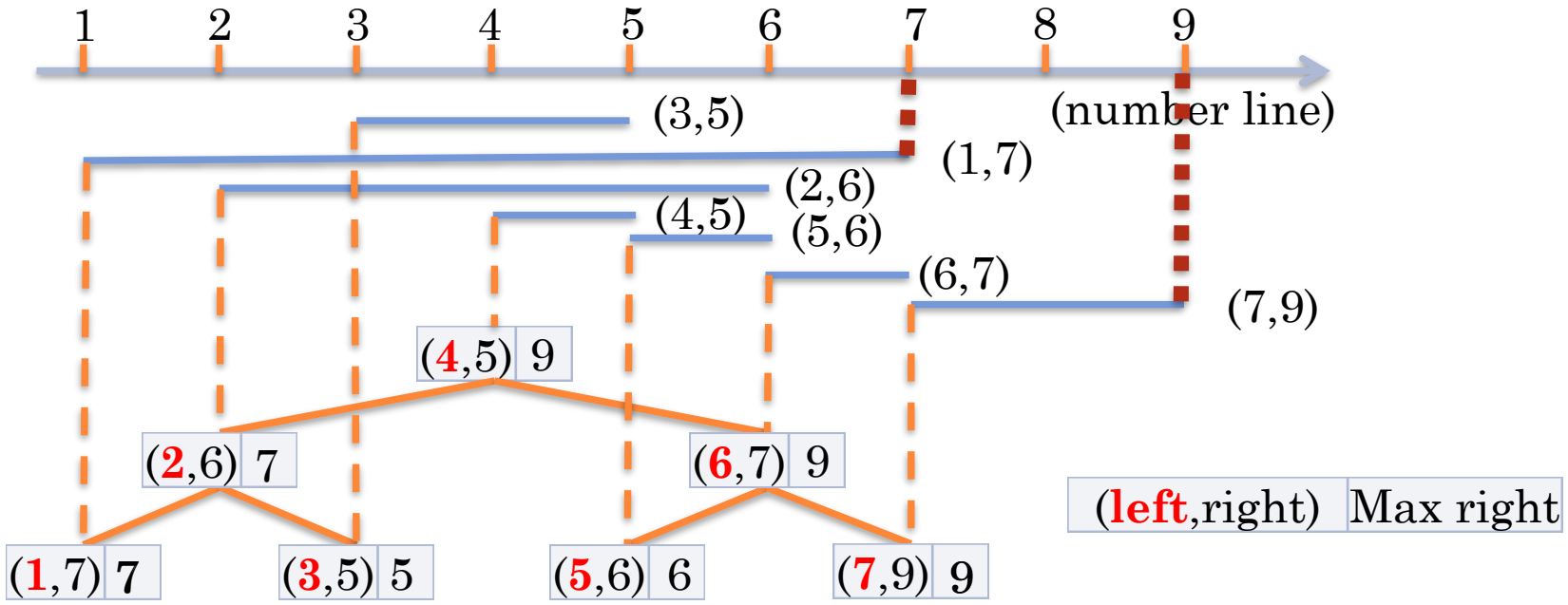
  - Is there an interval that satisfies $l_i \leq p \leq r_i$?
- One possible application: for a website, given the login interval of all users, query at any given time if anyone is online

1D stabbing query

- Given a set of intervals $[l_i, r_i]$ on the number line, and a specific point p, determine whether p is covered by any interval
  - Is there an interval that satisfies $l_i \leq p \leq r_i$?
- Possible solution:
  - Find all intervals that $l_i \leq p$
  - Return the maximum right endpoint r_m among them
  - If $p \leq r_m$ then return true, else return false
- If we sort all intervals by their left endpoint, and view “max” as an abstract “sum” function, then it just requires to fast answer a range sum query

Interval tree

- All intervals in the tree are sorted by the left endpoint. Each node is augmented by the maximum right endpoint in its subtree

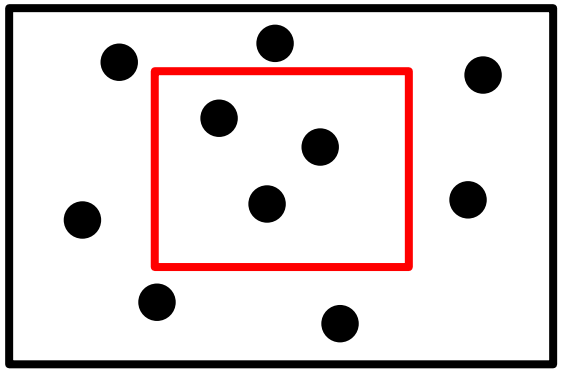


Interval tree

- Parallel construction: use the divide-and-conquer algorithm
  - Maintain the augmented values in the join function
- Query: **return** max_right(T.root, p)>p;
- max_right(node* t, p) {
  - **if** (!t) **return** $-\infty$;
  - **if** (p<t.left_endpoint) **return** max_right(t.left_child, p)
  - l_max = r.left_child.aug_val;
  - r_max = max_right(t.right_child, p);
  - **return** max(l_max, r_max, t.right_endpoint);
- }

2D range query

- Given a set of points on 2D planar, answer some information of all points in a query window $[x_L, x_R] \times [y_L, y_R]$



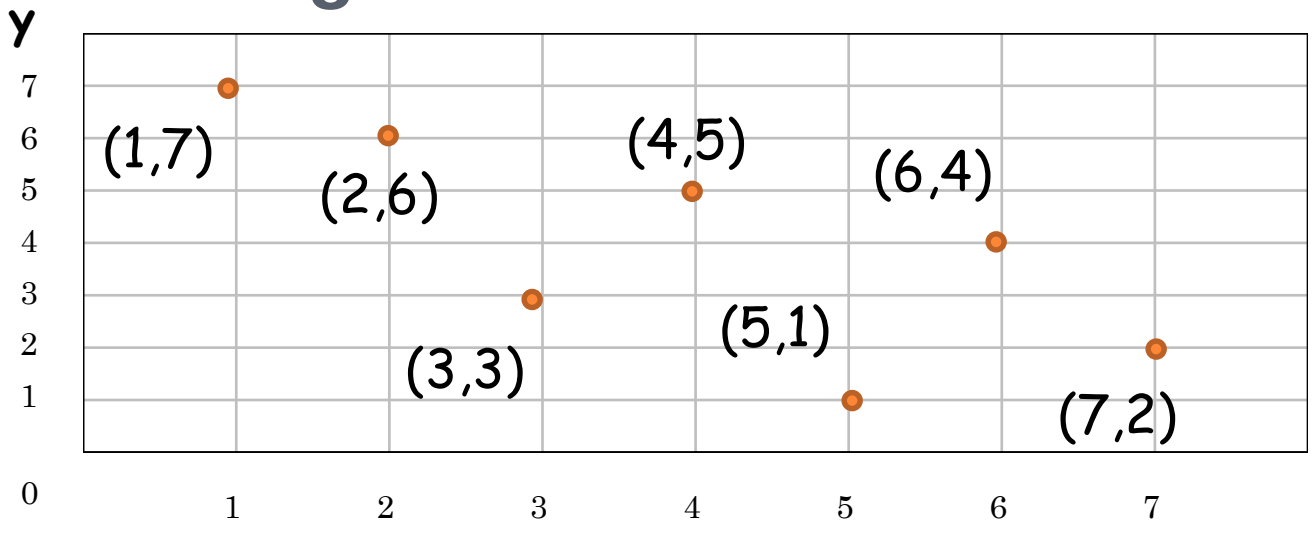
Application: In a database, report all people (or the number of people) in age range $[a_1, a_2]$ and salary range $[s_1, s_2]$

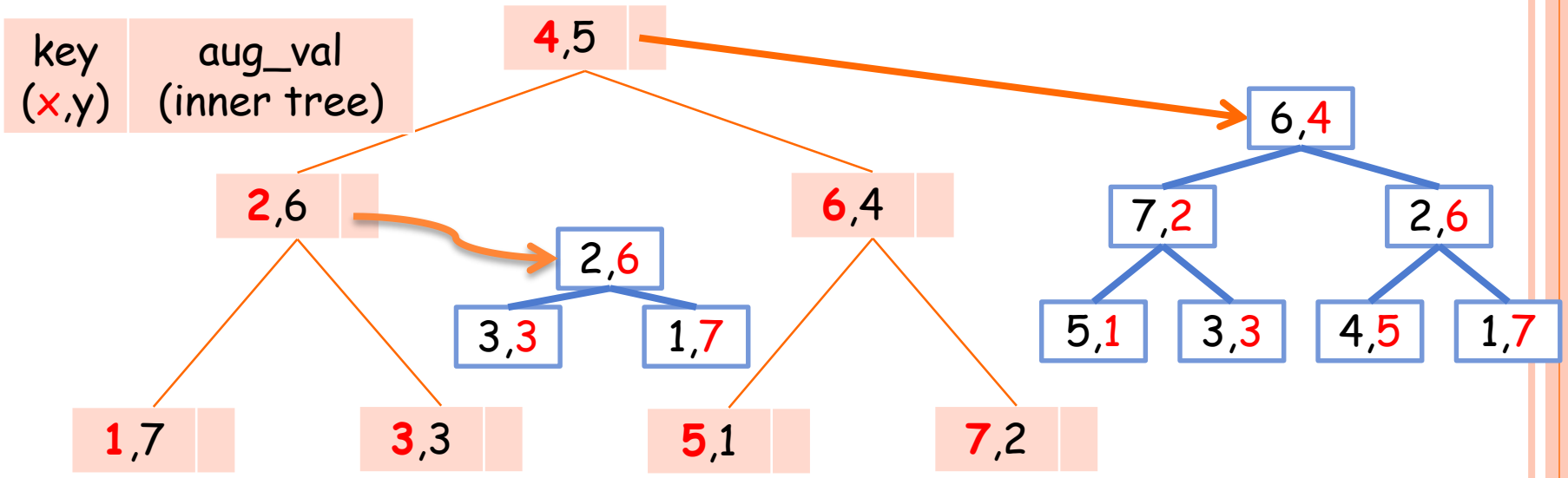
- Possible solution: find all points in $[x_L, x_R]$, find those in $[y_L, y_R]$
  - If the result points in the first search are sorted by their y-coordinate ……

2D Range Tree

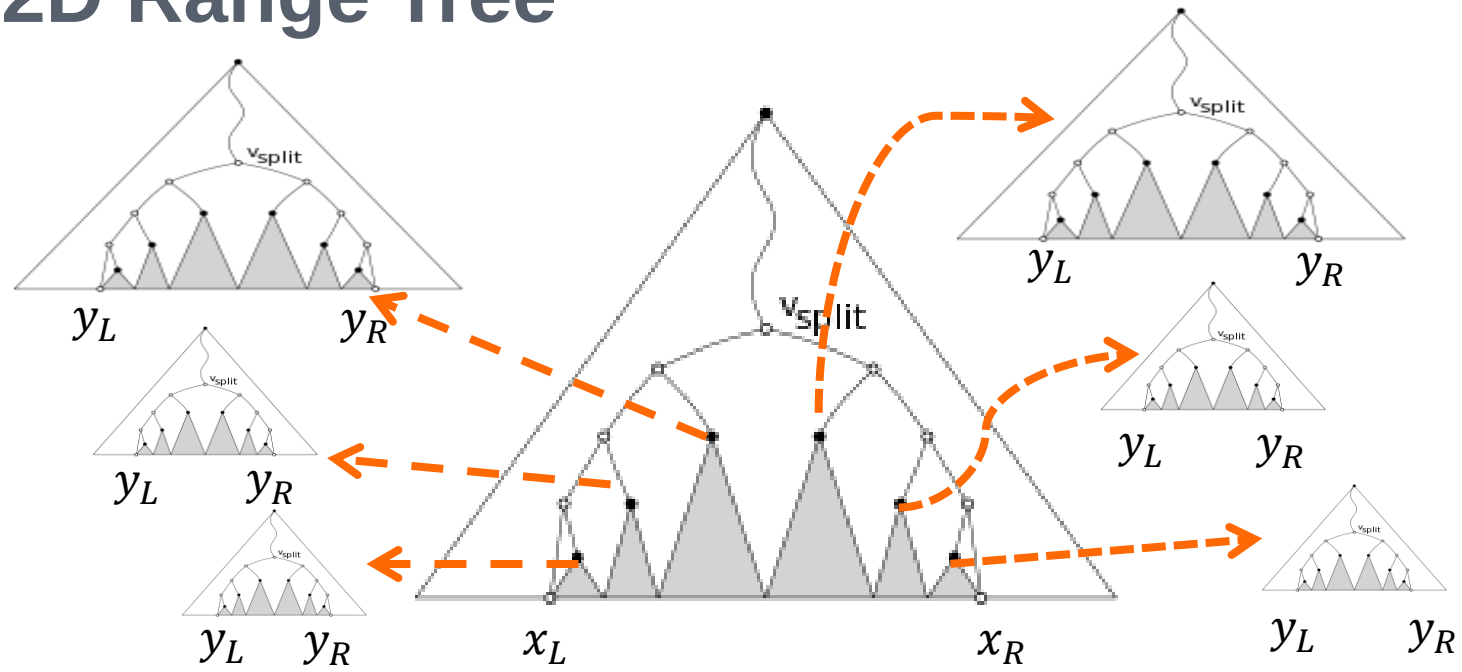
- 2D Range tree: a two-level tree structure
  - The augmented value of the each outer tree node is also a search tree structure
  - Outer level: all points sorted by x-coordinate
  - Inner level (in each tree node): all points in its subtree sorted by y-coordinate

2D Range Tree





2D Range Tree



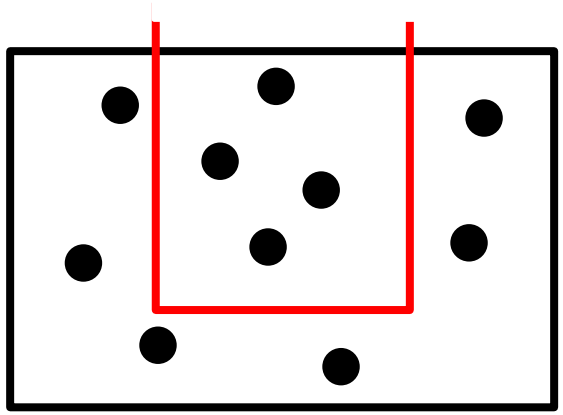
- A range query can be done by two nested range query on outer and inner level respectively
- Report all points in the search range $O(\log^2 n + k)$, where k is the output size
- Report the count in the search range: $O(\log^2 n)$

2D Range Tree

- If the tree is static (i.e., no insertions or deletions), the inner tree can also be maintained by arrays
- Construction: the parallel divide-and-conquer algorithm ($O(n \log n)$ work and polylog depth)
  - Maintain the augmented value (the inner tree) in the join function
  - When the join function is called, the inner trees of the two children have already been settled, so in this step we only need to merge two ordered structures into one
    - The union function we mentioned above, or the array merging algorithm (since the two ordered sets are of the same size)

3-sided query

- Given a set of points on 2D planar, report all points in query window $[x_L, x_R] \times [y_L, +\infty)$



The y coordinate is often called the priority of the point.

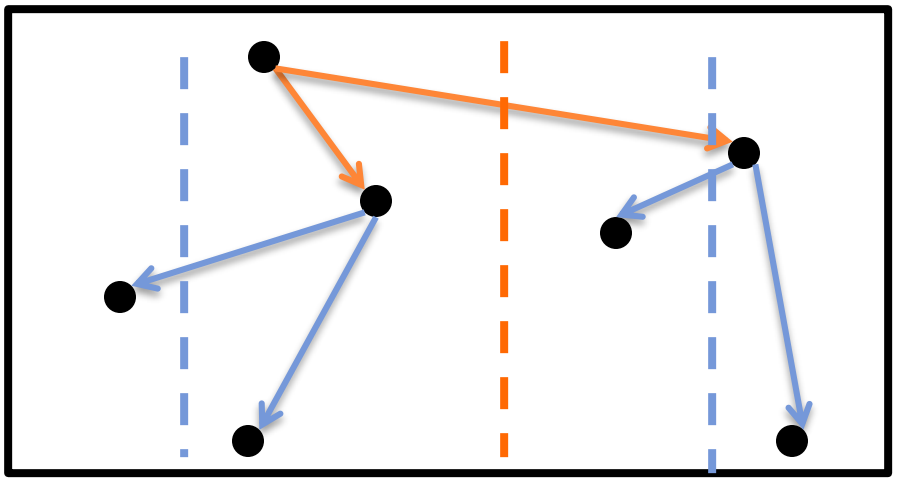
- Using a range tree, this can be done in time $O(\log^2 n + k)$
  - When k is small, $\log^2 n$ dominates
  - Can we do better?

3-sided query

- Given a set of points on 2D planar, report all points in query window $[x_L, x_R] \times [y_L, +\infty)$
- Using a range tree, this can be done in time $O(\log^2 n + k)$
  - When k is small, $\log^2 n$ dominates
  - Can we do better?
- When we do the secondary search, if we can first access the points with higher priority…
  - Instead of a tree, can we use a heap?

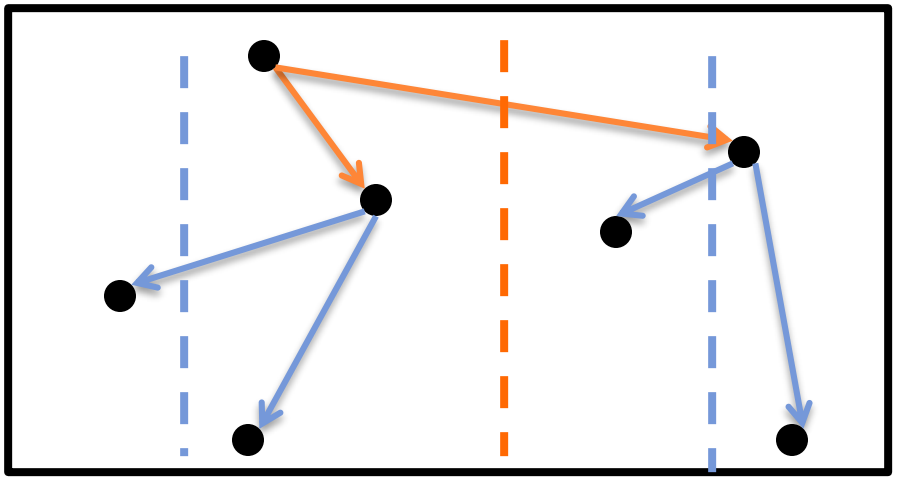
Priority search tree

- Heap on y (priority)
- (Almost) search tree on x



- The root is the point with highest priority
- All the other points are evenly split by the median of their x-coordinate, forming the left and right subtrees respectively
- Recursively build the two subtrees

Priority search tree



- Search $[x_L, x_R] \times [y_L, \infty)$:
  - Search $[x_L, x_R]$ in the tree, finding at most $O(\log n)$ related nodes and subtrees - $O(\log n)$
  - For each node, check if it is in the query range - $O(\log n)$
  - For each subtree, pop all points with priority higher than y_L - $O(k)$

In total $O(k + \log n)$

Priority search tree

- Construct a priority search tree in parallel:
  - Presort all points based on their x-coordinates. Store the sorted list in an array A

  $O(n \log n)$ work and $O(\log n)$ depth
  - Recursively do the follows:
    - In parallel find the point p with the highest priority

    $O(n)$ work and $O(\log n)$ depth
    - Remove p from the input, find the median m of the rest points
    - Copy the left half of the points in A_L (exclude p) recursively build the left subtree L
    - Copy the right half of the points in A_R (exclude p) recursively build the right subtree R
    - Store p in the root, set its left child to L, right child to R, the splitter key to m

  $O(n \log n)$ work and $O(\log^2 n)$ depth

Summary

- Parallel divide-and-conquer algorithms on trees
- Merging two trees in parallel
  - Lower bound
  - A divide-and-conquer algorithm
  - Cost analysis

- Applications:
  - 1d range query and augmentations
  - Solve 1d stabbing query using interval trees
  - Solve 2d range query using range trees
  - Solve 3-sided query using priority search trees