CS260 –parallel algorithms

Yihan Sun

Parallel data structures

Last Lecture: SSSP

- **Find a set of vertices as the frontier, update all their neighbors**
- **Dijkstra: 1 vertex in the frontier**
    - Frontier: the one with the smallest tentative distance
    - n rounds to finish
- **Bellman-ford: n vertices in the frontier**
    - Frontier: all vertices
    - n rounds to finish

Last Lecture: SSSP

- **Δ-stepping: a bucket of vertices in the frontier**
  - Fontier: all vertices with tentative distances in range $i\Delta$ to $(i+1)\Delta$
  - Run multiple rounds of Bellman-ford until no tentative distances are updated
  - No useful bounds about #rounds
- **Radius-stepping: a bucket of vertices in the frontier**
  - Shortcut each vertex to its ρ nearest neighbors, call the ρ-th nearest distance to v the radius of v, or $r(v)$
  - Frontier: all unsettled vertices within range d_i
    - $d_i = \min_{unsettled\ v \in V} \delta(v) + r(v)$ after round $i-1$
    - A vertex v can update its neighbors' distances as far as $\delta(v) + r(v)$
    - Run two rounds of Bellman-ford until no tentative distances are updated
  - $O\left(\frac{n}{p} \log \rho L\right)$ rounds to finish, $L =$ max/min edge weight

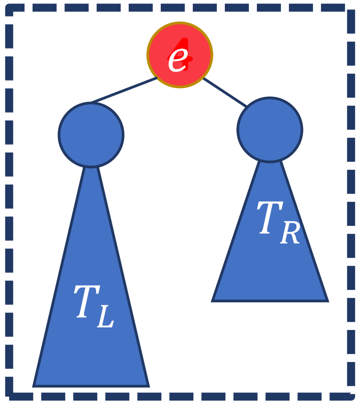
Useful tools

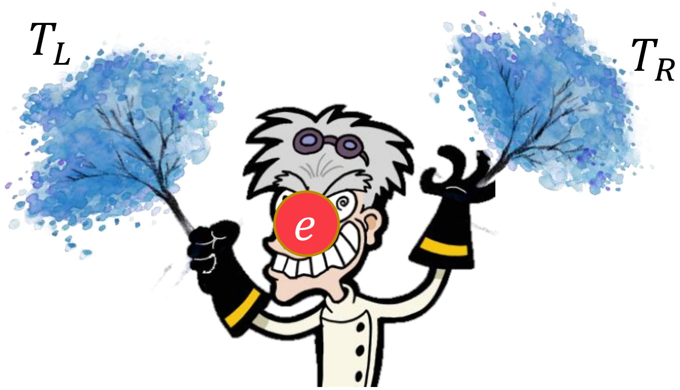
- **Almost all algorithms mentioned in the lectures about parallel algorithms are implemented in libraries:**
  - Ligra/Ligra+ [Shun and Blelloch'13]
    - Frontier-based graph algorithms
    - https://github.com/jshun/ligra
  - Julienne [Dhulipala et al.'17]
    - Bucket-based graph algorithms
    - https://github.com/jshun/ligra/tree/master/apps/bucketing
  - GBBS [Dhulipala et al.'18]
    - Parallel graph algorithm library
    - https://github.com/ldhulipala/gbbs
  - ASPEN [Dhulipala et al.'18]
    - Dynamic graph processing library
    - https://github.com/ldhulipala/aspen

Last lecture: parallel trees

- **Join-based tree algorithms**
- T =**Join**(T_L, e, T_R) : T_L and T_R are two trees of a certain balancing scheme, e is an entry/node (the pivot).
- $\boldsymbol{T_L < e < T_R}$
- It returns a valid tree T, which is $T_L \cup \{e\} \cup T_R$
- Cost: roughly the difference in height

$T =$



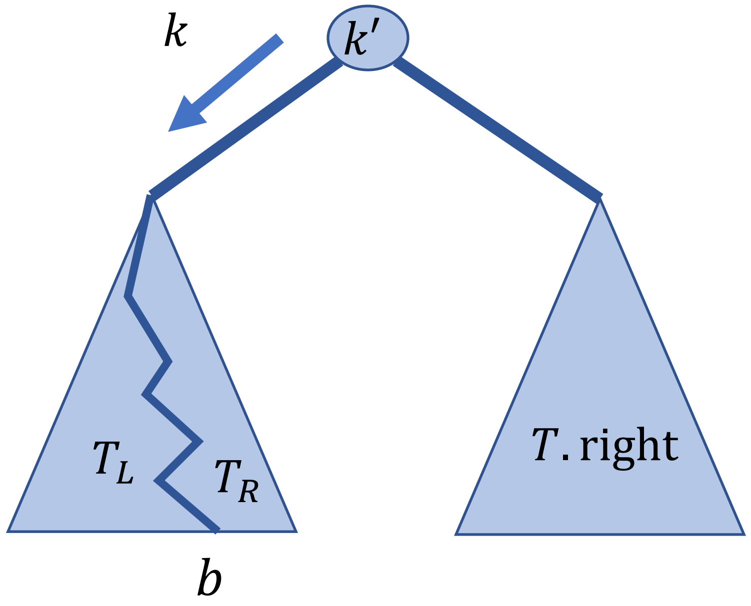


(Rebalance if necessary)

Parallel trees

- **Join-based insertion**$(\boldsymbol{T}, \boldsymbol{k})$
  - Recursively insert the key into the corresponding subtree
  - Join back using the root
- **Join-based split**$(\boldsymbol{T}, \boldsymbol{k})$
  - Find which subtree does the key locate
  - Split that subtree
  - Join back

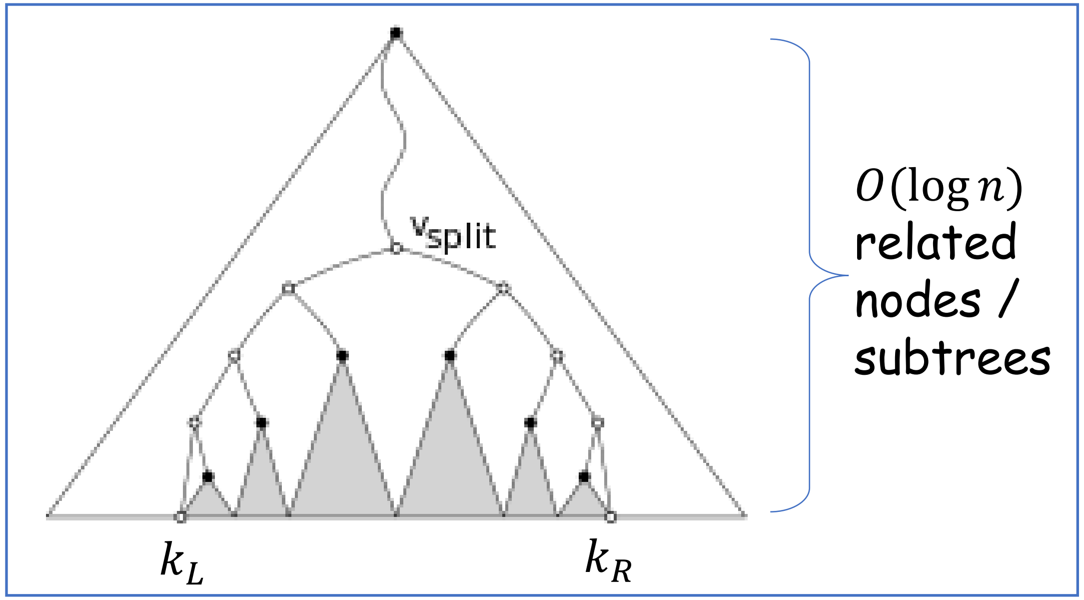
Return
$\langle T_L, b, \text{join}(T_R, k', T.\text{right})\rangle$



- **Join2**$(\boldsymbol{T_L}, \boldsymbol{T_R})$
  - Split the last element k in T_L
  - Join $T_L - k$ and T_R using k

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



Equivalent to using two split algorithms

Join-based parallel algorithms

Parallel Algorithms

- **Parallel algorithms on trees using divide-and-conquer scheme**
  - Recursively deal with two subtrees in parallel (or split the tree into two pieces and handle them in parallel)
  - Combine results of recursive calls and the root (e.g., using join or join2)
  - 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, …)
}
```

Get the maximum value

- In each node we store a key and a value. The nodes are sorted by the keys.

```
get_max(Tree T) {
    if (T is empty) return −∞;
    in parallel:
        L=get_max(T.left);
        R=get_max(T.right);
    return max(max(L, T.root.value), R);
```

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

Similar algorithm work on any map-reduce function

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`).
  - Reduce is associative
- 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);
```

e.g., for all values a_i in the tree, return $\sum a_i^2$

$\Rightarrow$ map(x) = x^2

$\Rightarrow$ reduce(x, y) = $x + y$;

$O(n)$ 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);
```

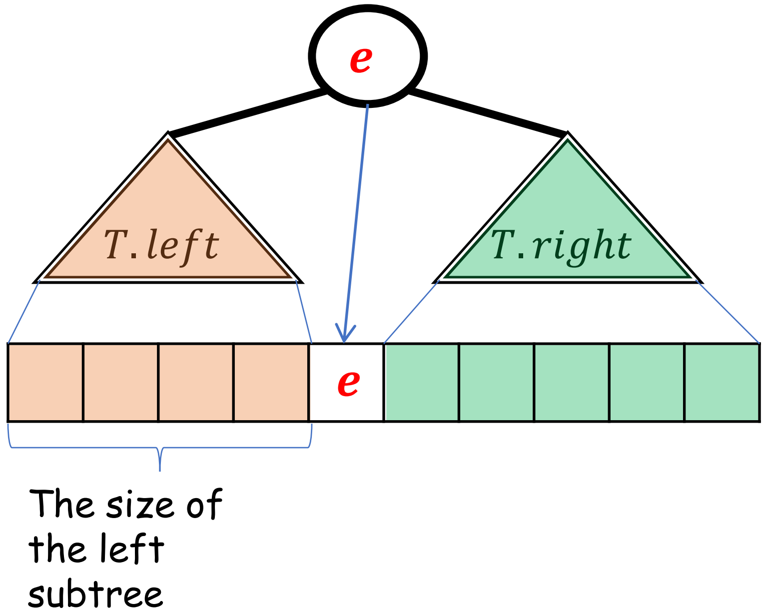
$O(n)$ work and
$O(\log n)$ depth

$O(n \log n)$ 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);
```

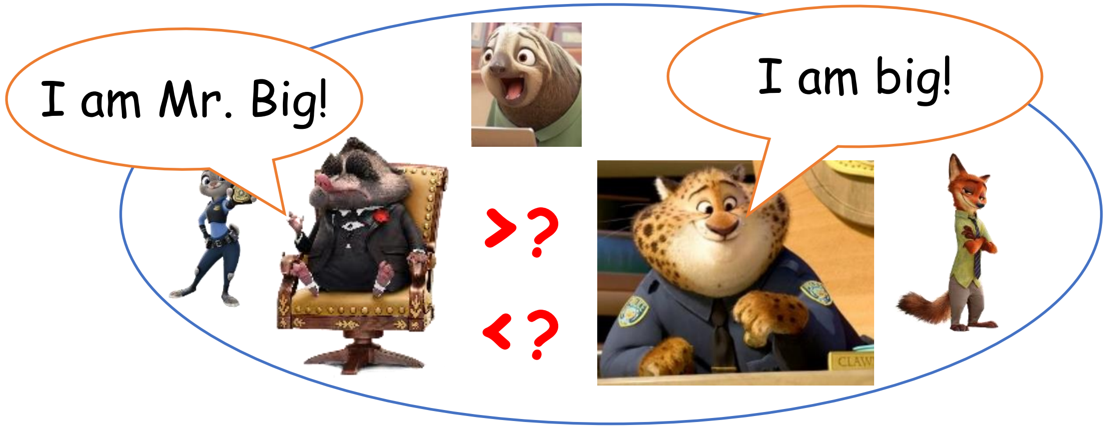


$O(n)$ work and $O(\log n)$ depth

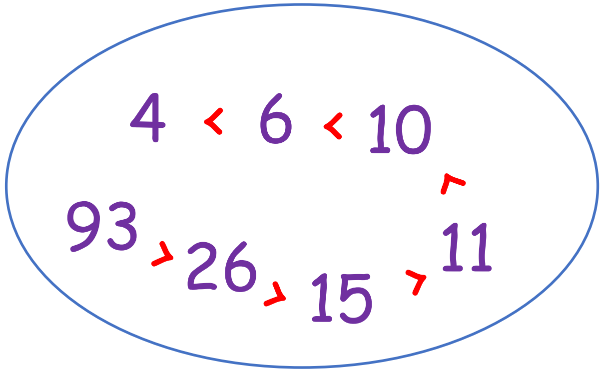
Parallel algorithms for ordered sets

Ordered Sets

- Ordered sets: sets with total ordering



A set of citizens in Zootopia (unordered)

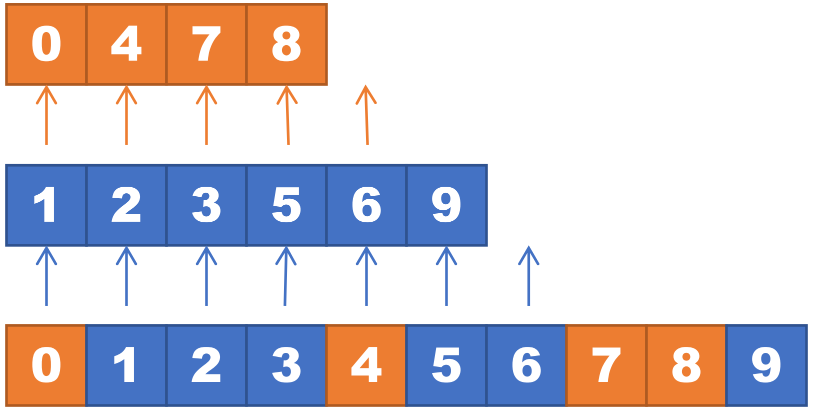


A set of integers (ordered)

- Useful interface:
  - Find, previous, next, size, first, last, k-th, …
  - Filter, reduce, construction, …
  - Union, intersection, difference, symmetric difference, …

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

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

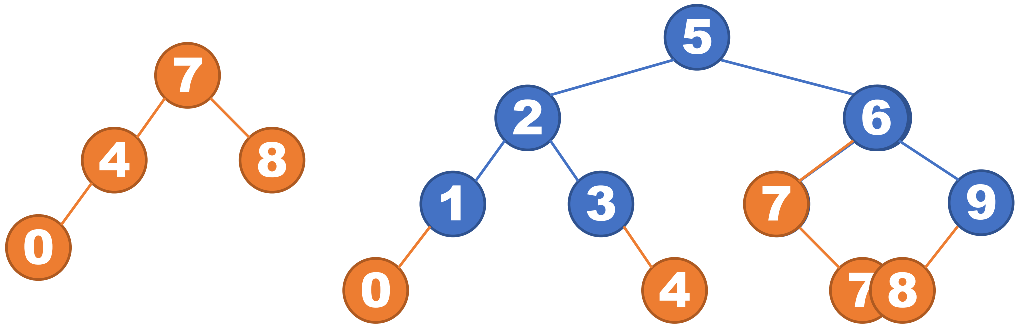


Cost: $O(m+n)$

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 $n = m$?

$O(n \log n)$?

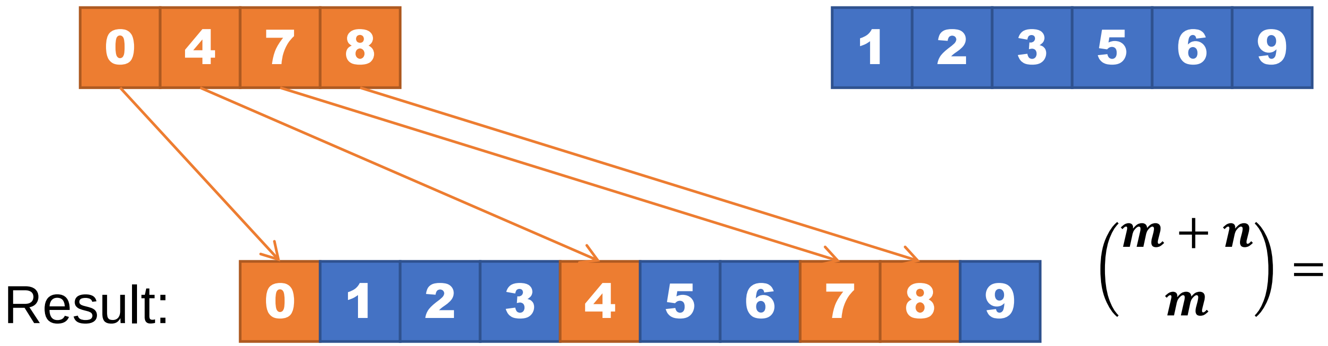
- **What is the minimum cost?**

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.

Set 1 (size m):

Set 2 (size $n \geq m$):



$\binom{\boldsymbol{m}+\boldsymbol{n}}{\boldsymbol{m}} = \binom{\boldsymbol{m}+\boldsymbol{n}}{\boldsymbol{n}}$ cases

- 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?

Join-based Algorithms: Union

union$(\boldsymbol{T_1}, \boldsymbol{T_2})$

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1

Base case

$(L_2, k_2, R_2) = extract(T_2)$

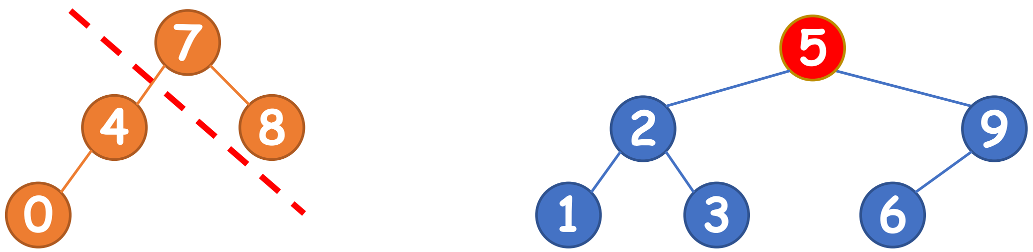
$(L_1, b, R_1) =$split(T_1, k_2)

In parallel:

$T_L =$`Union`(L_1, L_2)

$T_R =$`Union`(R_1, R_2)

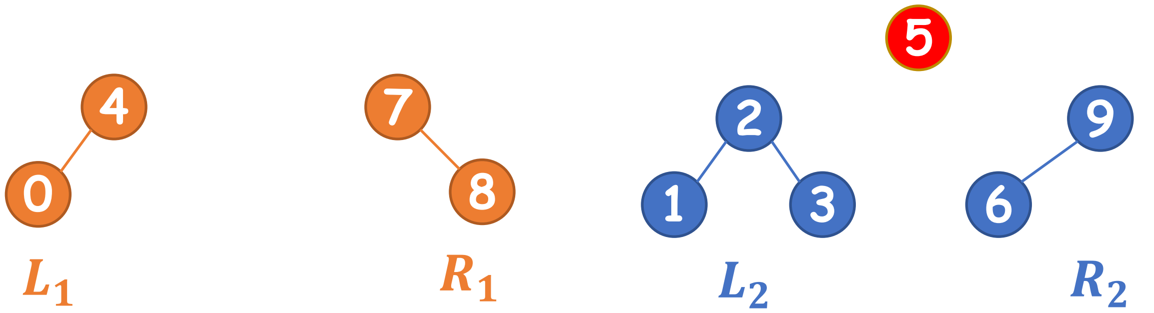
return `Join`(T_L, k_2, T_R)



Join-based Algorithms: Union

union$(\boldsymbol{T_1}, \boldsymbol{T_2})$

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
$\boxed{(L_1, b, R_1) = \text{split}(T_1, k_2)}$
In parallel:
$T_L = \texttt{Union}(L_1, L_2)$
$T_R = \texttt{Union}(R_1, R_2)$
return $\texttt{Join}(T_L, k_2, T_R)$

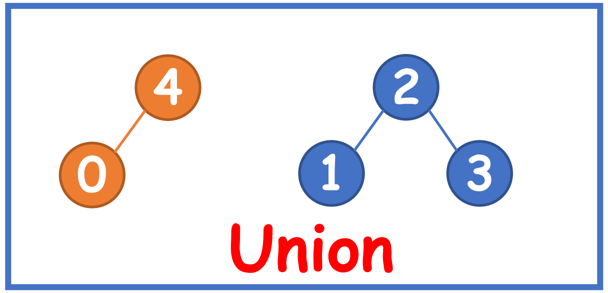


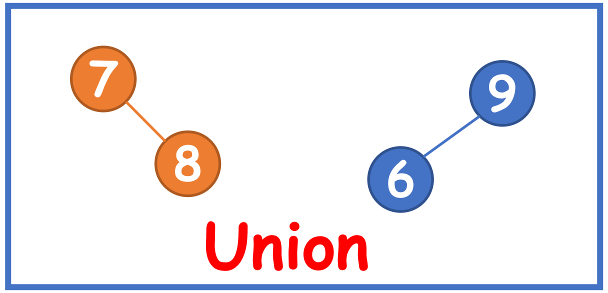
Join-based Algorithms: Union

union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
$(L_1, b, R_1) =$ split(T_1, k_2)
In parallel:
$\boxed{\begin{array}{l} T_L = \texttt{Union}(L_1, L_2) \\ T_R = \texttt{Union}(R_1, R_2) \end{array}}$
return $\texttt{Join}(T_L, k_2, T_R)$

5

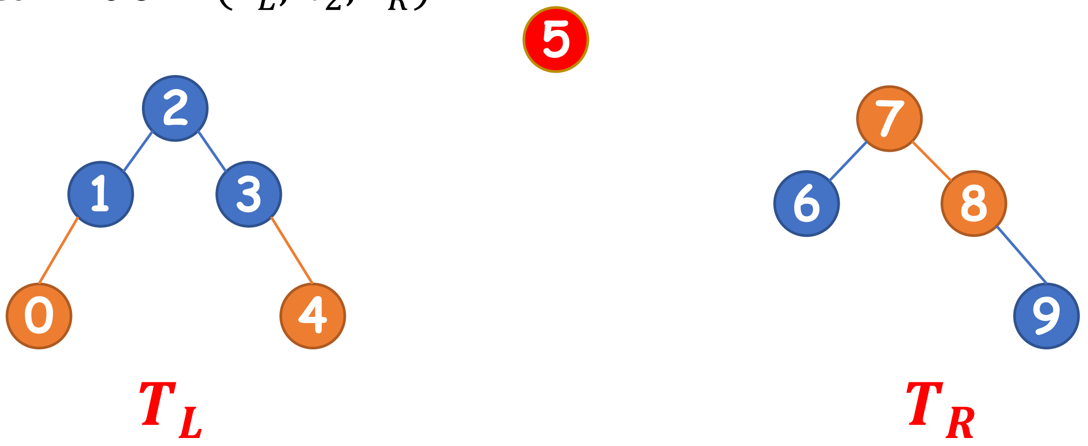




Join-based Algorithms: Union

union($\boldsymbol{T_1}, \boldsymbol{T_2}$)

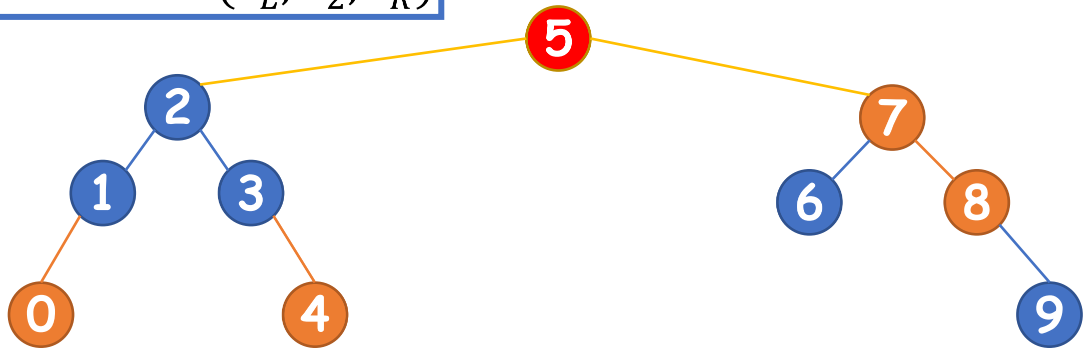
if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
$(L_1, b, R_1) =$ split(T_1, k_2)
In parallel:
$\boxed{\begin{array}{l} T_L = \texttt{Union}(L_1, L_2) \\ T_R = \texttt{Union}(R_1, R_2) \end{array}}$
return $\texttt{Join}(T_L, k_2, T_R)$



Join-based Algorithms: Union

union$(\boldsymbol{T_1}, \boldsymbol{T_2})$

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
$(L_1, b, R_1) =$split(T_1, k_2)
In parallel:
$T_L = \texttt{Union}(L_1, L_2)$
$T_R = \texttt{Union}(R_1, R_2)$
$\boxed{\textbf{return } \texttt{Join}(T_L, k_2, T_R)}$



Similarly we can implement intersection and difference.

Join-based Algorithms: Union

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 ($m \le n$) 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 bound also holds for intersection and difference

Cost analysis of union

Using AVL as an example

union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
$(L_1, b, R_1) =$split(T_1, k_2)
In parallel:
$T_L =$`Union`(L_1, L_2)
$T_R =$`Union`(R_1, R_2)
return `Join`(T_L, k_2, T_R)

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

Split cost: the height of T_1
Join cost: difference of height between T_L and T_R

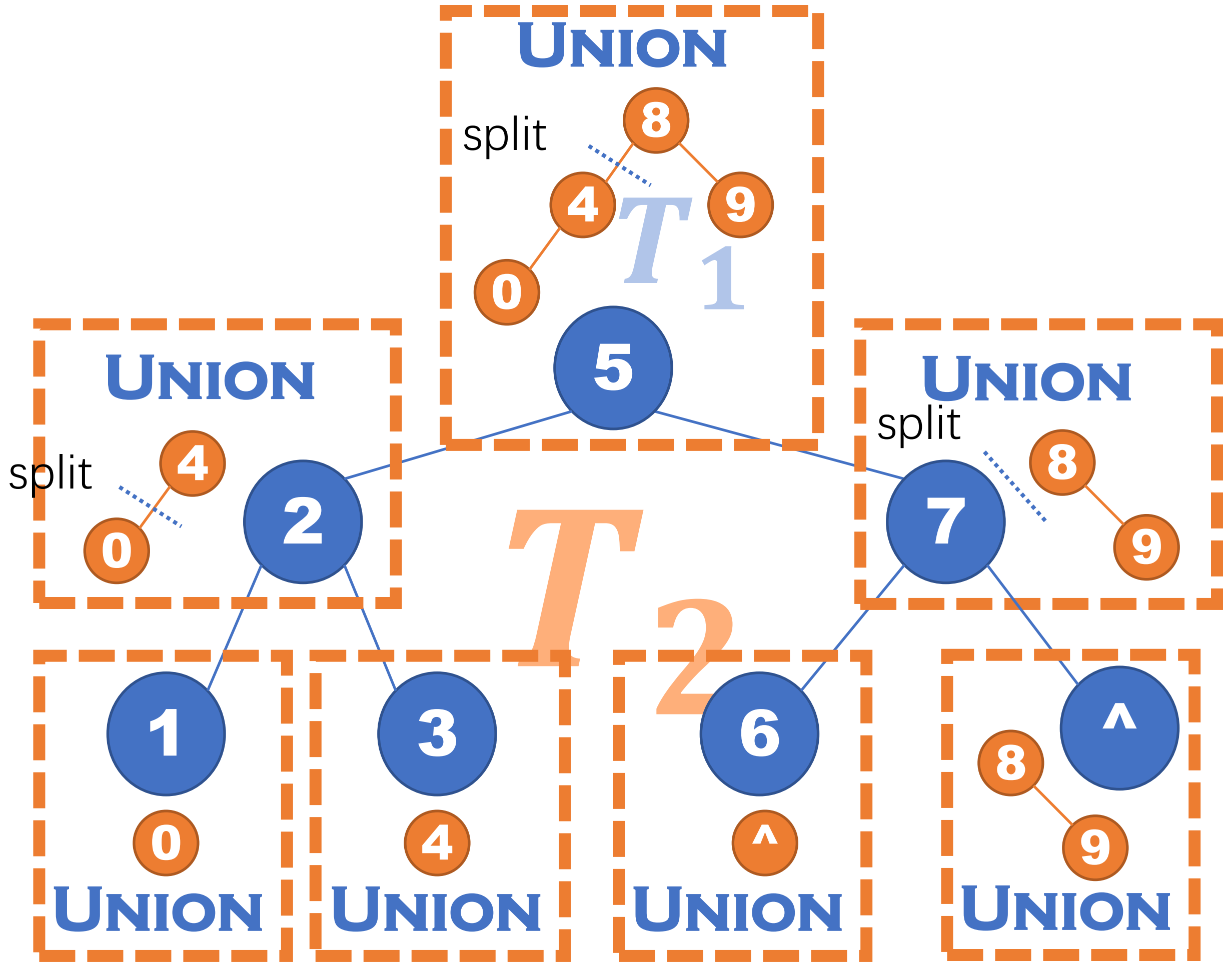
$$T_L = T_2.left \cup a\ subset\ of\ T_1$$
$$T_R = T_2.right \cup a\ subset\ of\ T_1$$

Can prove by induction

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

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

The Split Work



union$(\boldsymbol{T_1}, \boldsymbol{T_2})$

if $T_1 = \emptyset$ **then return** T_2

if $T_2 = \emptyset$ **then return** T_1

$(L_2, k_2, R_2) = extract(T_2)$

$(L_1, b, R_1) =$split(T_1, k_2)

In parallel:

$T_L = \texttt{Union}(L_1, L_2)$

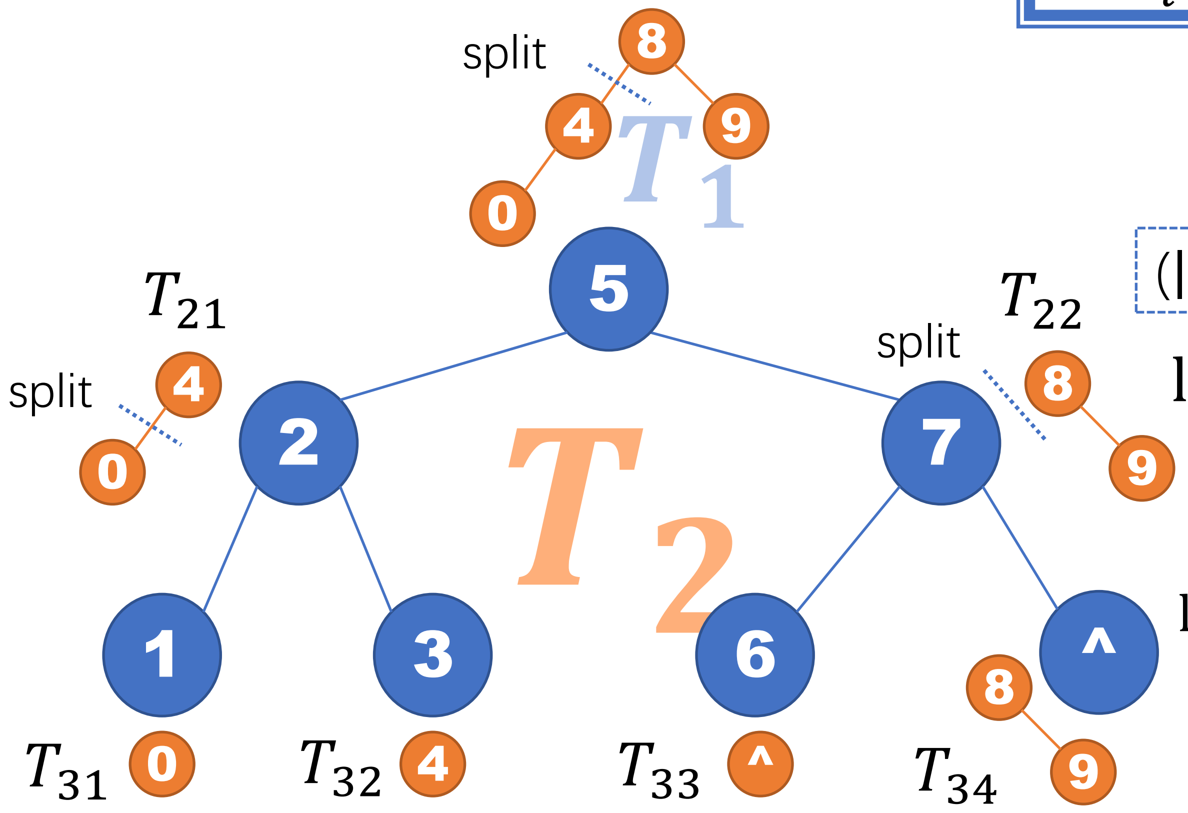
$T_R = \texttt{Union}(R_1, R_2)$

return $\texttt{Join}(T_L, k_2, T_R)$

The Split Work

Concavity:

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



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

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

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

......

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

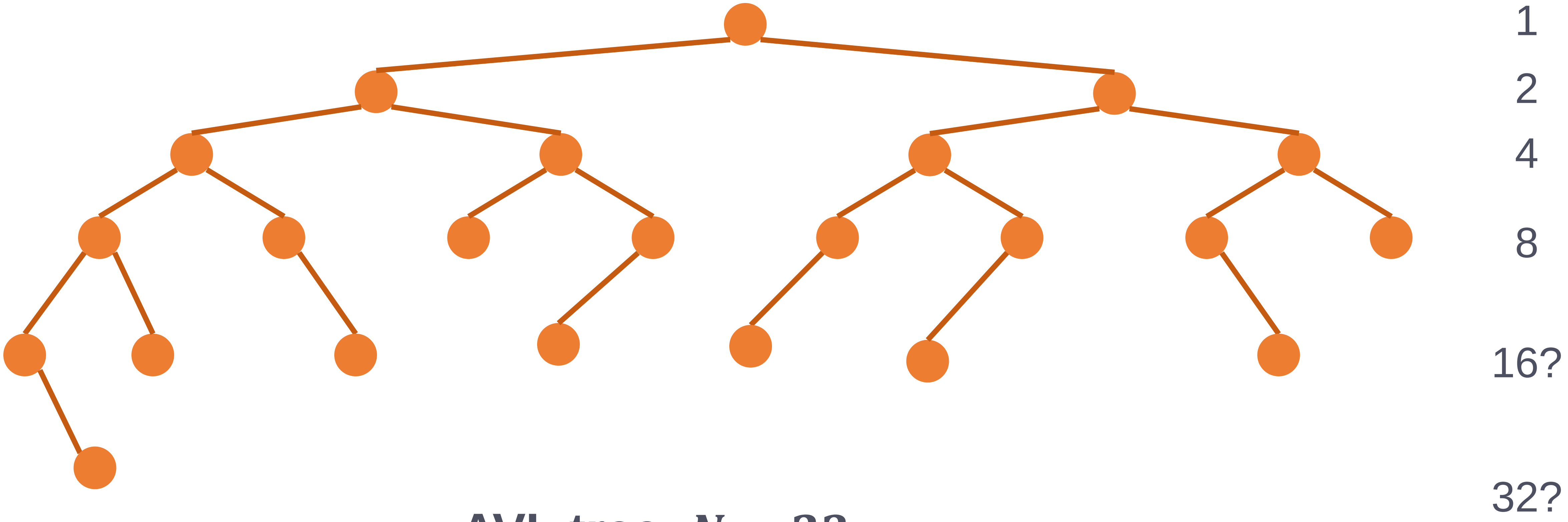
$$\log |T_{h1}| + \cdots + \log |T_{h2^h}|$$
$$\le 2^h \log \frac{n}{2^h}$$

$$\underbrace{\log n + 2 \log \frac{n}{2} + 4 \log \frac{n}{4} \ldots + 2^h \log \frac{n}{2^h}}_{\textbf{c}\ \log_2 m \text{ terms (If } T_2 \text{ is perfectly balanced)}} = O\left(m^{\mathbf{c}} \log\left(\frac{n}{m} + 1\right)\right)$$

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

How can we enumerate the splitters?

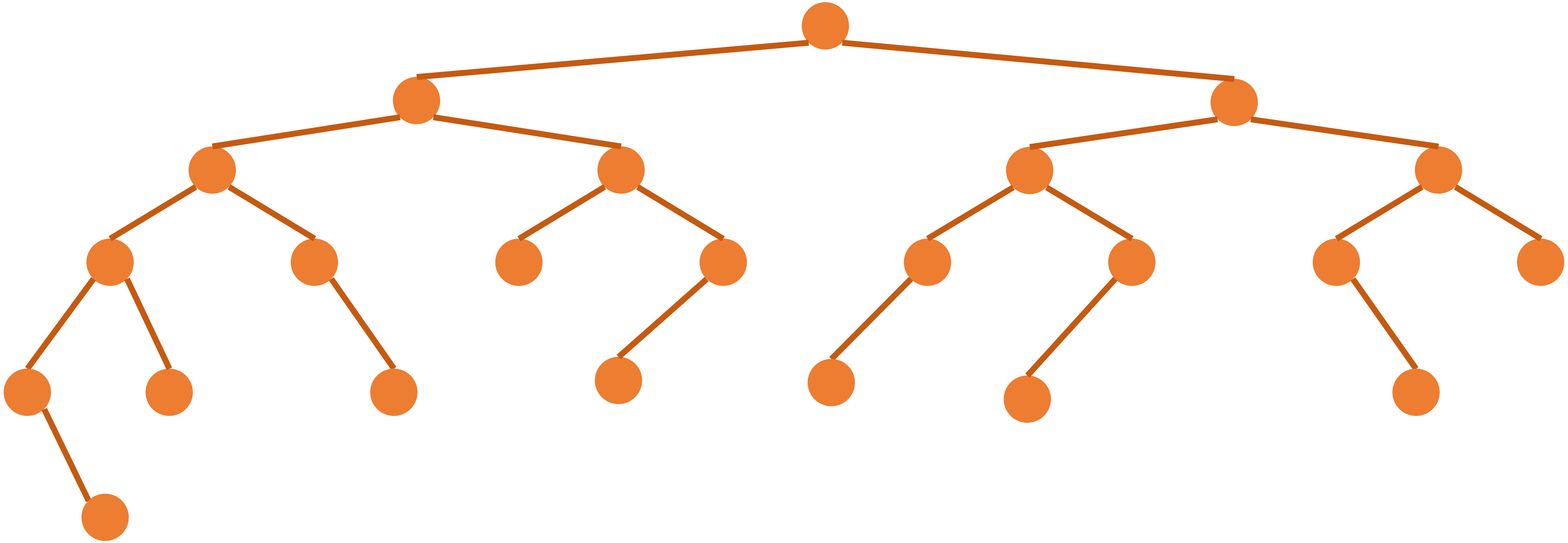
- Height $O(\log n)$
- For the i–th layer, there are at most 2^{i-1} nodes
- For $c\log n$ layers, are there $2^{c\log n} = n^c$ nodes?



AVL tree, $N = 23$

How can we enumerate the splitters?

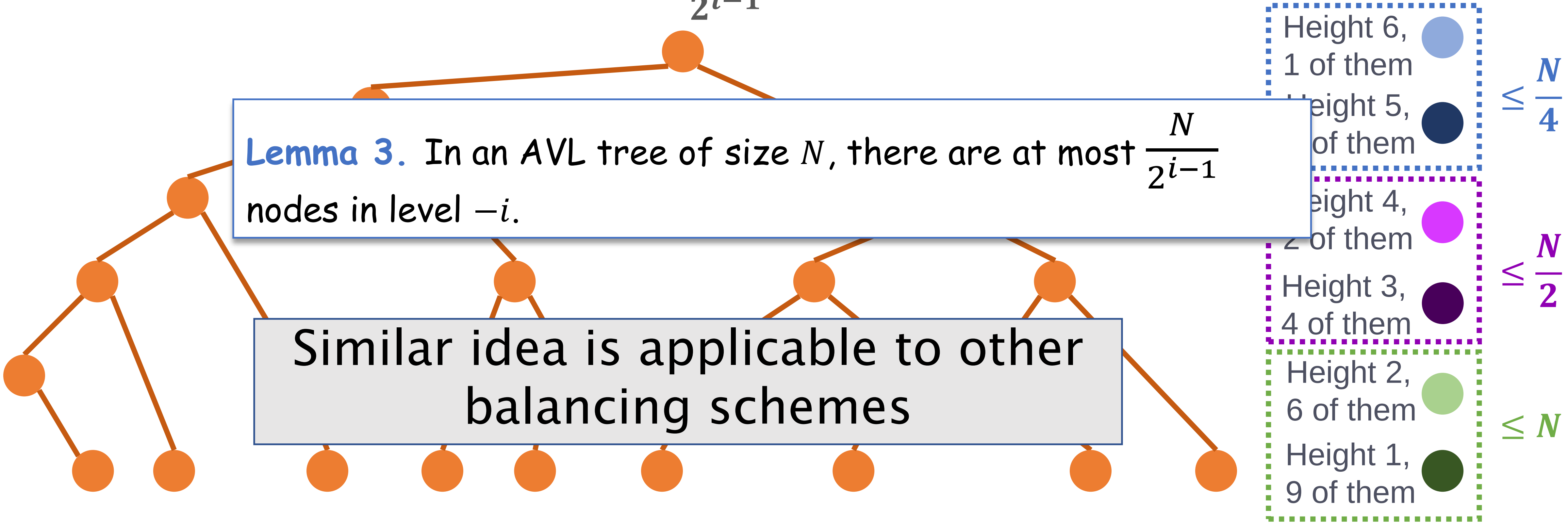
- Idea: group all nodes bottom up
- Example: AVL tree: group all nodes with height $2i - 1$ and $2i$ into level $-i$: no more than $\frac{N}{2^{i-1}}$ of them



AVL tree, $N = 23$

How can we enumerate the splitters?

- Idea: group all nodes bottom up
- Example: AVL tree: group all nodes with height $2i-1$ and $2i$ into level $-i$: no more than $\frac{N}{2^{i-1}}$ of them



Lemma 3. In an AVL tree of size N, there are at most $\frac{N}{2^{i-1}}$ nodes in level $-i$.

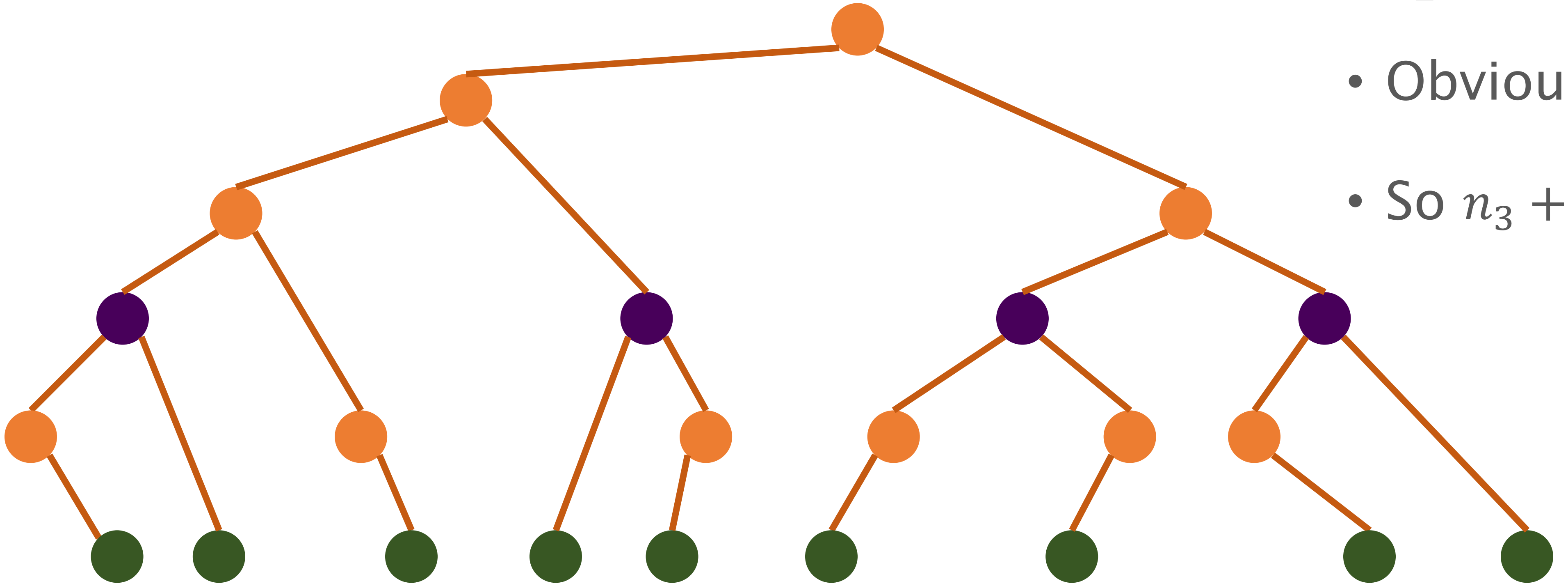
Similar idea is applicable to other balancing schemes

AVL tree, $N = 23$

Proof sketch

Let n_i be the number of nodes with height i

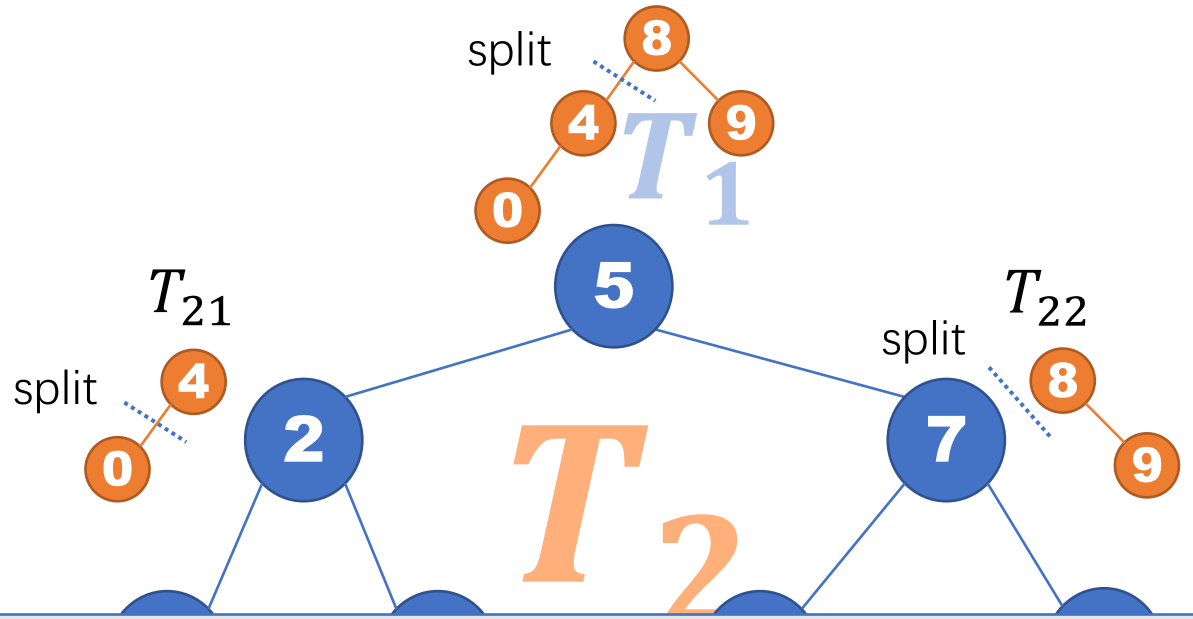
- Every node with height 3 has at least 2 leaf descendants, so $n_3 \leq \frac{n_1}{2}$
  - AVL invariant
- Remove all leaf nodes, it is still an AVL tree, so $n_4 \leq \frac{n_2}{2}$



- Obviously $n_1 + n_2 \leq N$
- So $n_3 + n_4 \leq \frac{N}{2}$

Lemma 3. In an AVL tree of size N, there are at most $\frac{N}{2^{i-1}}$ nodes in level $-i$.

The Split Work



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

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

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

Join-based Algorithms: Union

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 ($m \le n$) 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 bound also holds for intersection and difference

Persistent parallel trees for MVCC

What Are Persistence and MVCC?

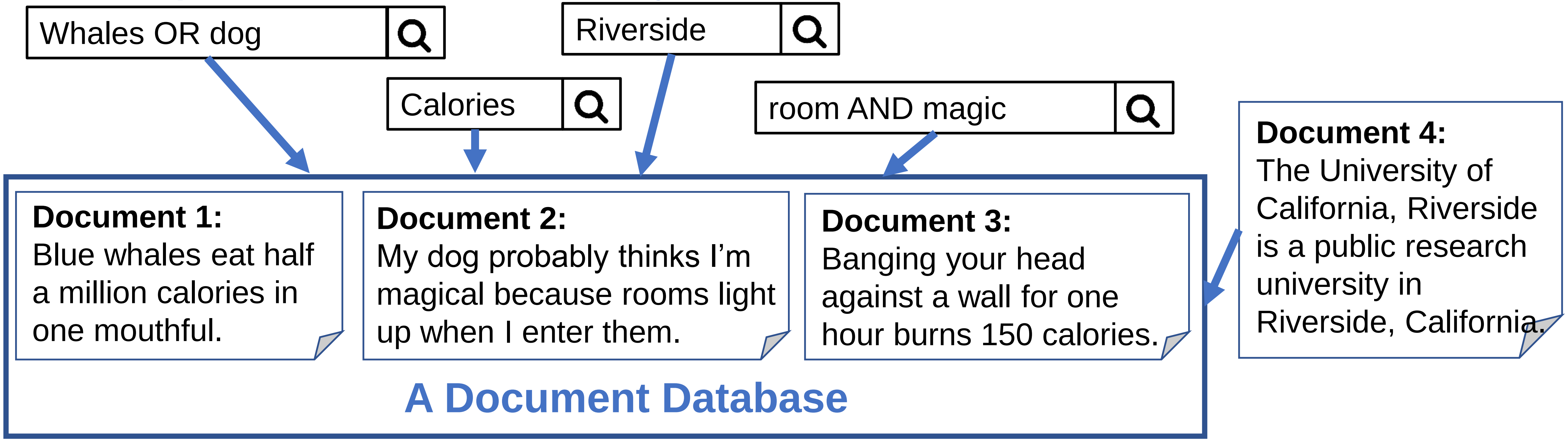
- **Persistence** [DSST'86]**: for data structures**
  - Preserves the previous version of itself
  - Always yields a new version when being updated
- **Multi-version Concurrency Control (MVCC): for databases**
  - Let write transactions create new versions
  - Let ongoing queries work on old versions

Why Persistence and MVCC?

- **To guarantee concurrent updates and queries to work correctly and efficiently**
  - Queries work on a consistent version
  - Writers/readers do not block each other

Why Persistence and MVCC?

An example of a document search engine.



For end-user experience:

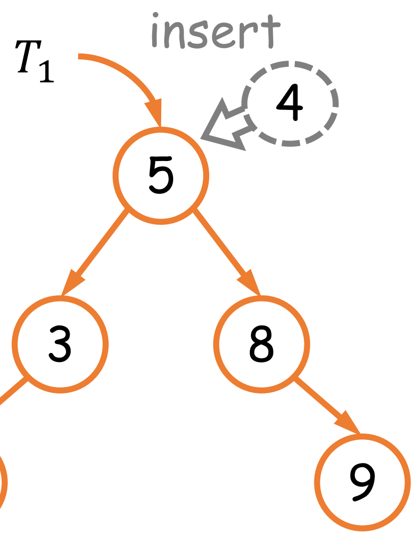
- Queries shouldn't be delayed by updates
- Queries must be done on a consistent version of database

Generally useful for any database systems with concurrent updates and queries.

Hybrid Transactional and Analytical Processing (HTAP) Database System

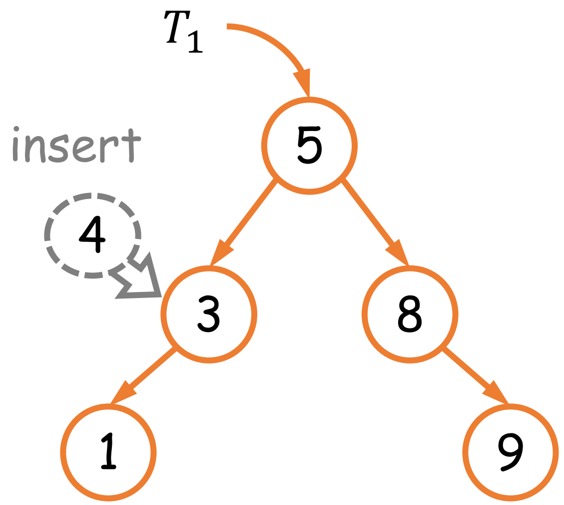
Persistence Using Join

- **Path-copying**: copy the affected path on trees
- **Copying occur only in Join!**



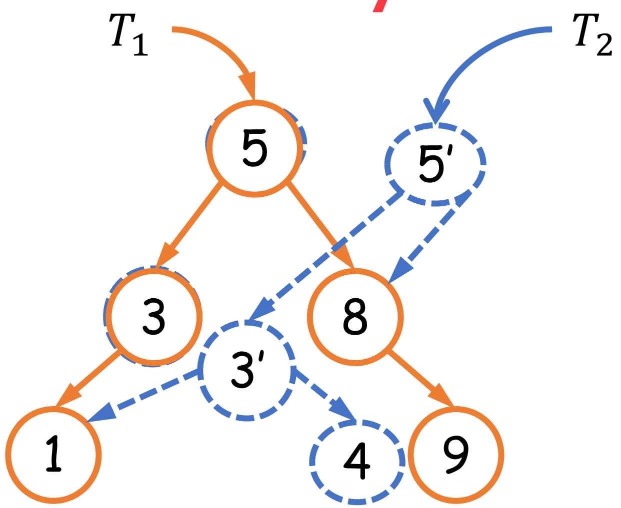
Persistence Using Join

- **Path-copying**: copy the affected path on trees
- **Copying occur only in Join!**



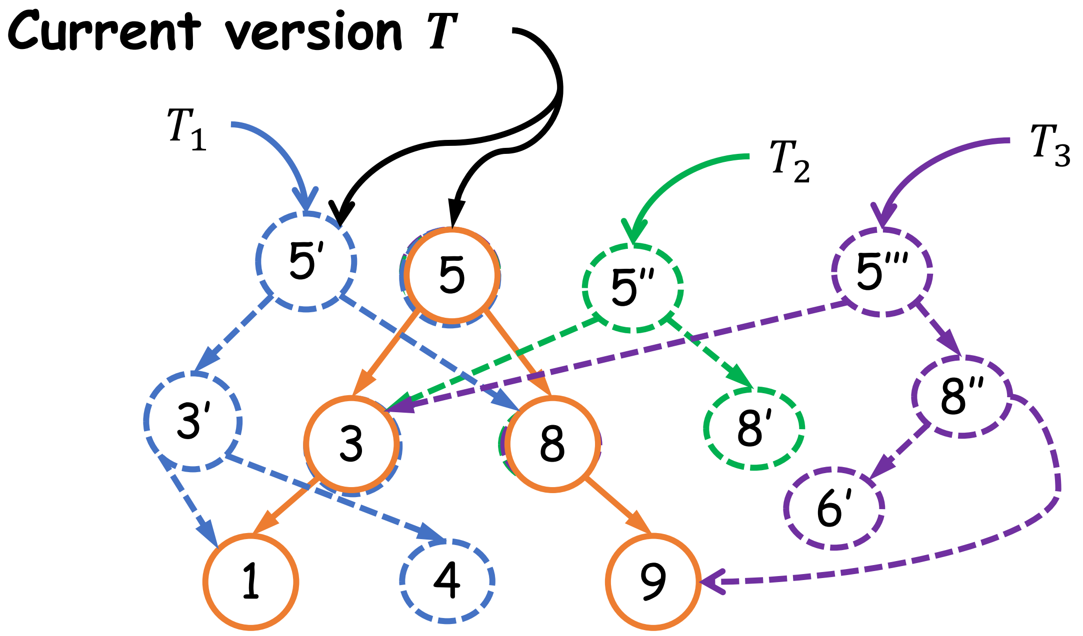
Persistence Using Join

- **Path-copying**: copy the affected path on trees
- Copying occur **only in Join**!
- Always **copy** the middle node
- All the other parts in the algorithm remain **unchanged**
- **No extra cost** in time asymptotically, small overhead in space
- Safe for **concurrency** – multi-version concurrency control (MVCC)



Persistence for MVCC

- Each operate on a snapshot – safe for concurrency
- Multiple updates can be visible atomically – lock-free



P_1: $T_1 = T$.insert(4)
commit(T_1)

P_2: $T_2 = T$.delete(9)

P_3: $T_3 = T$.insert(6)

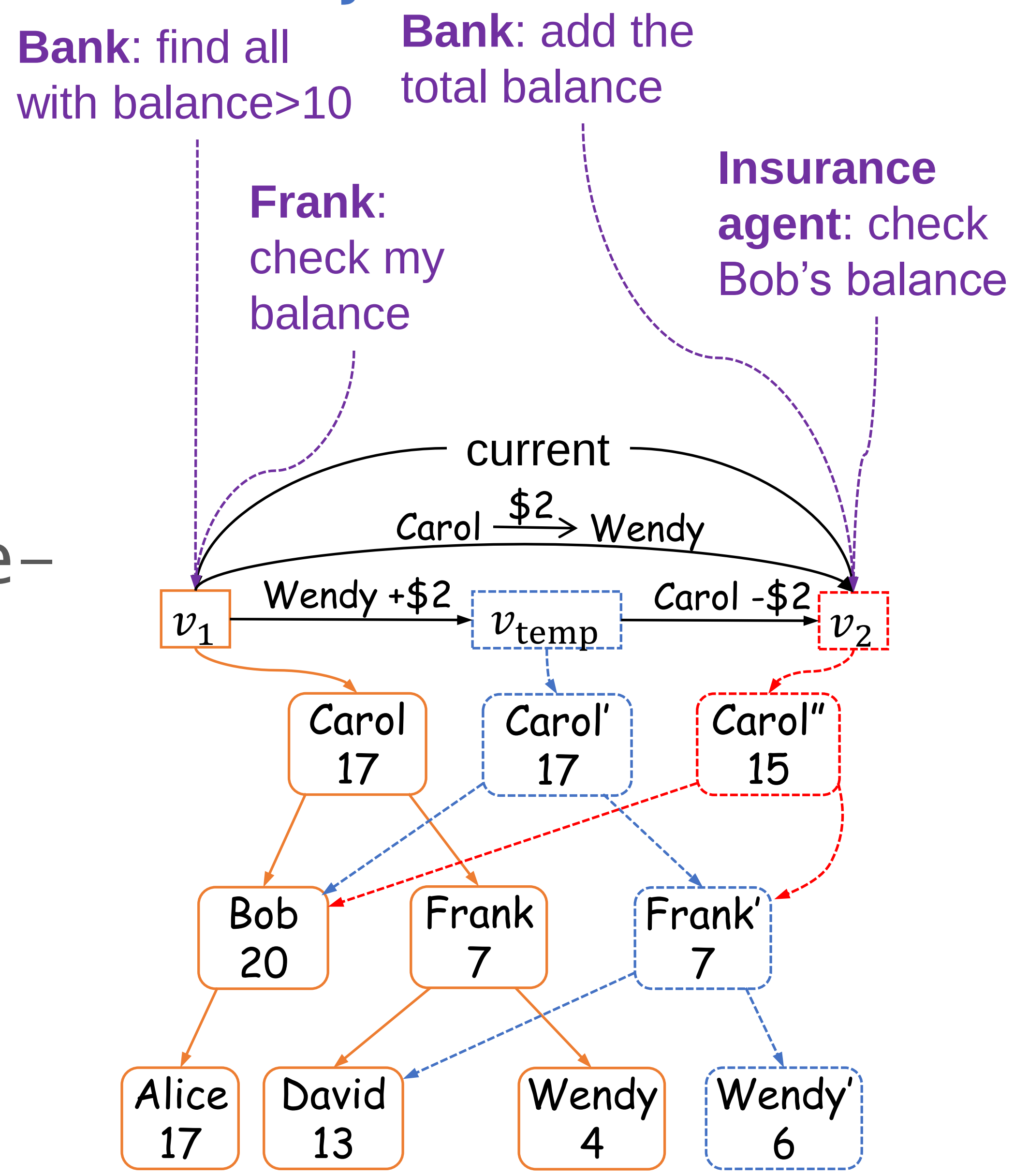
Transactions using Multi-version Concurrency Control (MVCC)

- Lock-free **atomic** updates ☺
  - A series of operations
  - A bulk of operations (e.g., union)
- Easy **roll-back** ☺
- Do not affect other **concurrent** operations ☺
- Any operation works on as if a single-versioned tree with **no extra (asymptotical) cost** ☺

Worst-case

$O(\log n)$ for a lookup

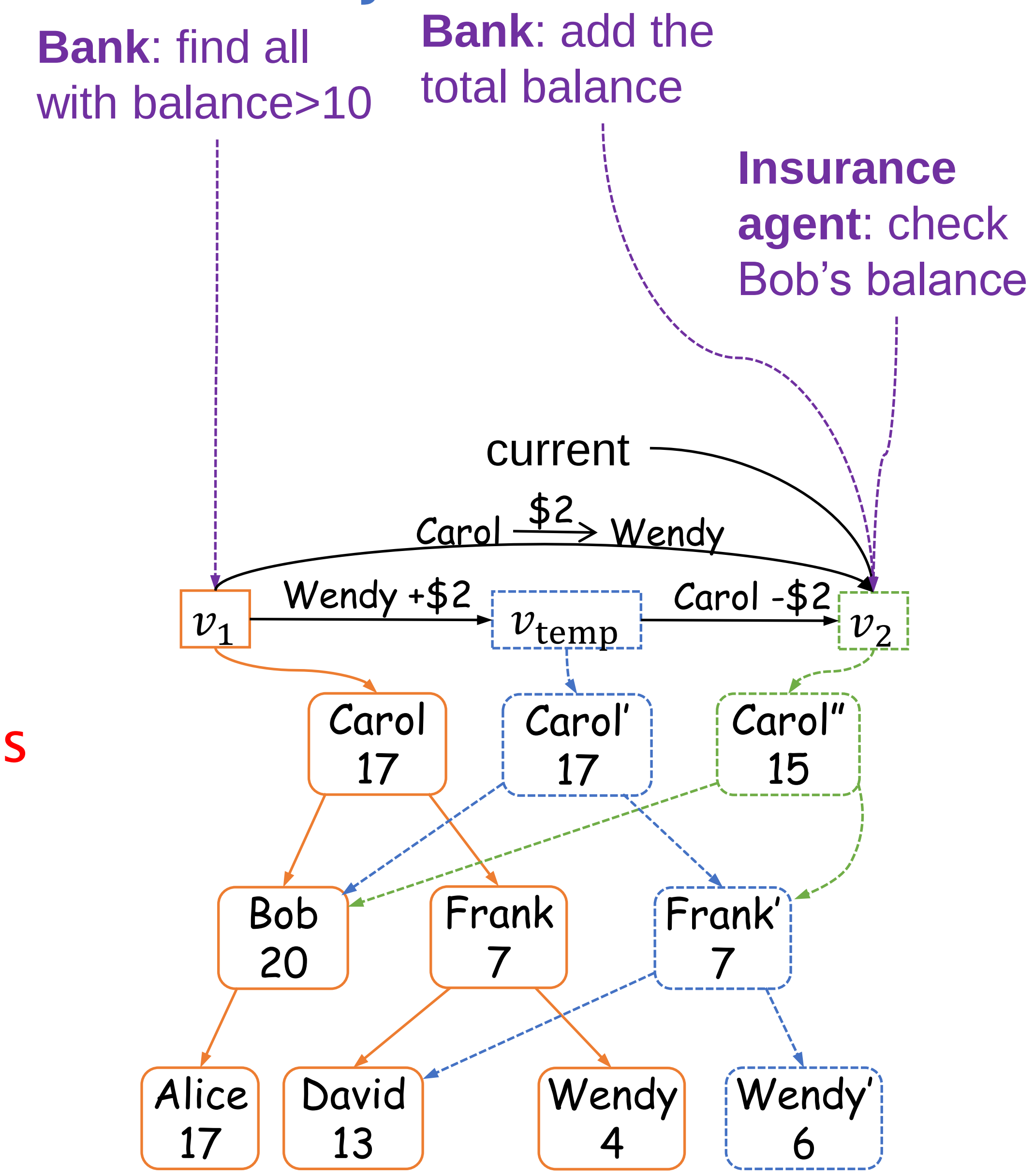
$O(\log n)$ for an insertion



Transactions using Multi-version Concurrency Control (MVCC)

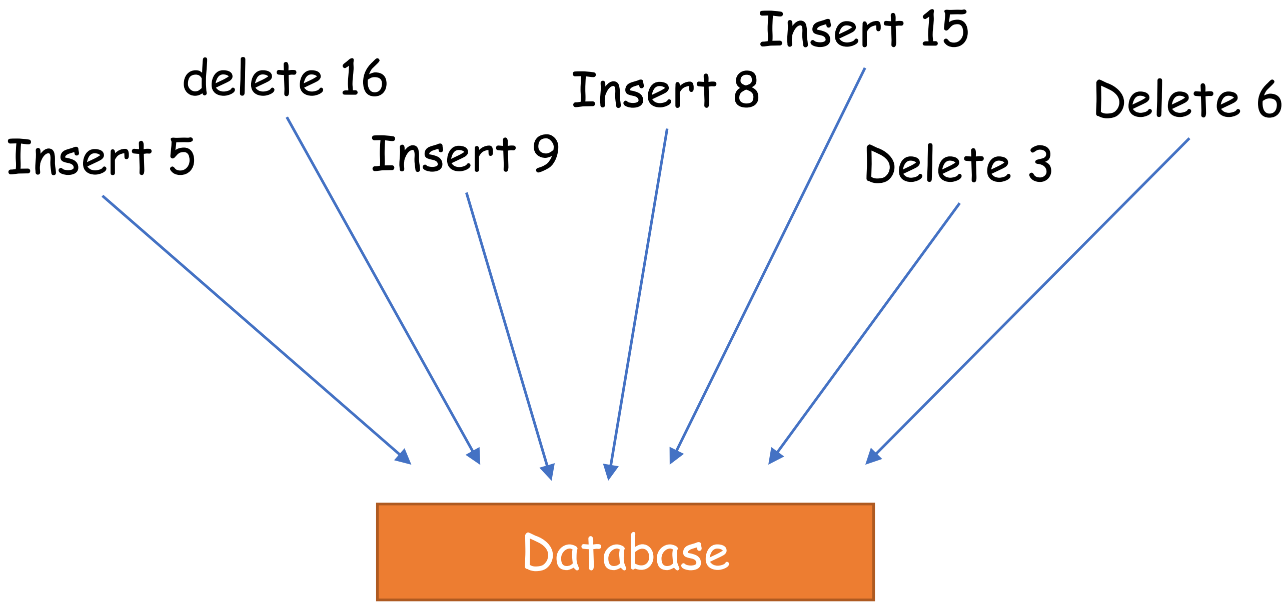
- Lock-free atomic updates ☺
  - A series of operations
  - A bulk of operations (e.g., union)
- Easy roll-back ☺
- Do not affect other concurrent operations ☺
- Any operation works on as if a single-versioned tree with no extra (asymptotical) cost ☺

- Concurrent writes? ☹
  - Concurrent transactions work on snapshots
  - They don't come into effect on the same tree?
- Useless old nodes? ☹
  - Out-of-date nodes should be collected in time



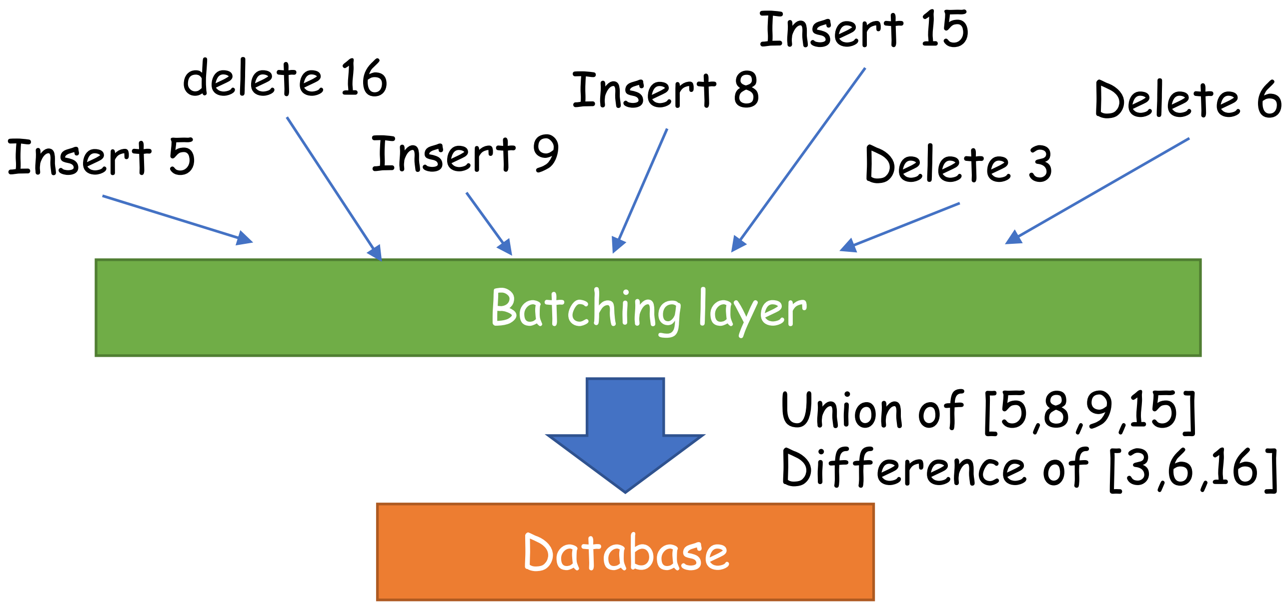
Batching

- Collect all concurrent writes can commit using a single writer once a while



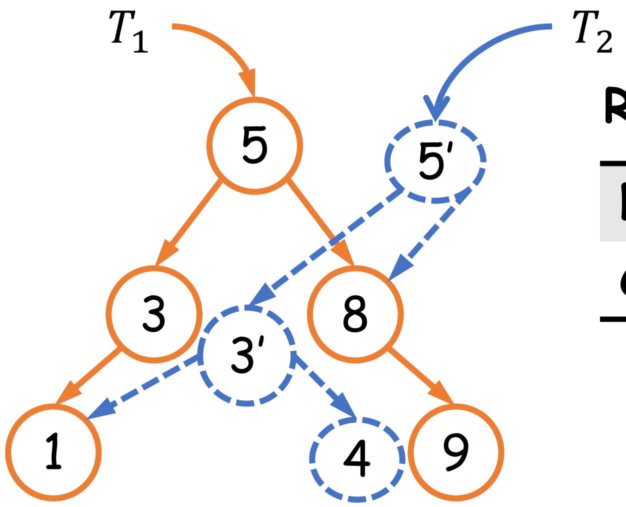
Batching

- Collect all concurrent writes can commit using a single writer once a while



Garbage Collection

- **Reference Counter Garbage Collector**
  - Each tree node records the number of other tree nodes/pointers refers to it
  - Node 8 and 1 in the example have reference counter 2

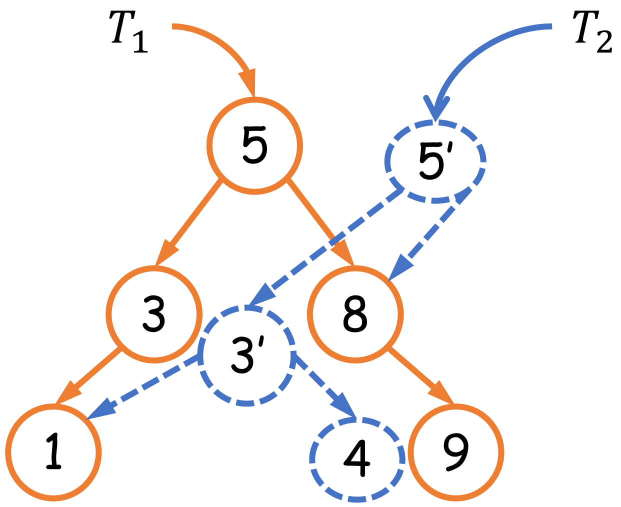


Reference count:

Node	1	3	5	8	9	5′	3′	4
Count	2	1	1	2	1	1	1	1

Garbage Collection

- Each tree node records the number of other tree nodes/pointers refers to it
- Node 8 and 1 in the example have reference counter 2
- Collect a node if and only if its reference count is 1

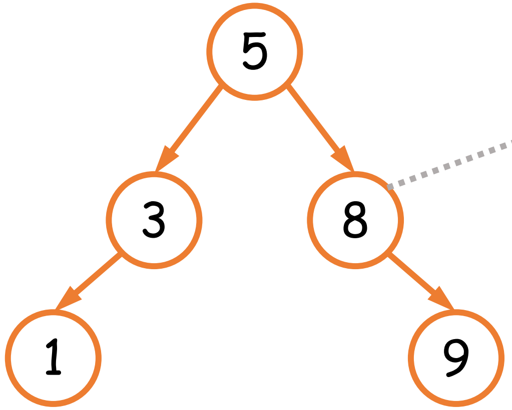



```
collect(node* t) {
  if (!t) return;
  if (t->ref_cnt == 1) {
    node* lc = t->lc, *rc = t->rc;
    free(t);
    in parallel:
      collect(lc);
      collect(rc);
  } else dec(t->ref_cnt);
}
```

Node	1	3	5	8	9	5′	3′	4
Count	1	1	1	1	1	1	1	1

Version chains

- **An alternative way is to use version chains**
  - Stores all versions in one (tree) skeleton
  - Readers need to check the visibility of versions
  - Less space used, but readers can be slow



36

Time stamp	9	11	20	21	34	55	60
Value	4	5	2	7	2	5	9

A reader starting at time 36 would check the version chain and return the value 2 at time stamp 34