Data Structures

Lecture 5*

More on Trees:
Extensions and Advanced Operations

Shiri Chechik, Or Zamir
Winter semester 2025-6

* based on the TAU course slides, edited by AR and TAUOnline

Plan for Today

We will see several extensions and advanced operations on (AVL) trees.
All can be applied to other types of trees.

1. Rank and Select
2. Finger Trees
3. Split and Join
4. Tree-List

Rank Trees: Motivation

Extending Dictionaries

ADT Dictionary - Reminder

Suppose a dictionary item x contains key and value, in the fields $x.key, x.val$

- Dictionary() — Create an empty dictionary
- Insert(D, x) — Insert x to D
- Delete(D, x) — Delete a given item x from D (assuming it exists)
- Search(D, k) — Return item with a given key k (if exists)
- Min(D) — Return item with the minimal key
- Max(D) — Return item with the maximal key
- Successor(D, x) — Return the successor of a given item x
- Predecessor(D, x) — Return the predecessor of a given item x

Assume that Items have distinct keys.

Additional Operations

• Dictionary()	Create an empty dictionary
• Insert(D, x)	Insert x to D
• Delete(D, x)	Delete a given item x from D (assuming it exists)
• Search(D, k)	Return item with a given key k (if exists)
• Min(D)	Return item with the minimal key
• Max(D)	Return item with the maximal key
• Successor(D, x)	Return the successor of a given item x
• Predecessor(D, x)	Return the predecessor of a given item x

Select(D, k) – return the k^{th} smallest element in D

Rank(D, x) – return the rank of a given element x in D

Rank = position in sorted order

For Example:

Select$(D, 4)$ = 6

Rank(D, x) = 5

Note that:

Select$(D$, Rank$(D,x))$ = x

Rank$(D$, Select$(D, k))$ = k

Rank Trees (aka Order Statistics Trees)

Is it possible to implement all dictionary operations + Select + Rank in $O(\log n)$ time worst case?

We will now see a solution based on an **extension** of AVL trees, called **Rank Trees**.

Definition

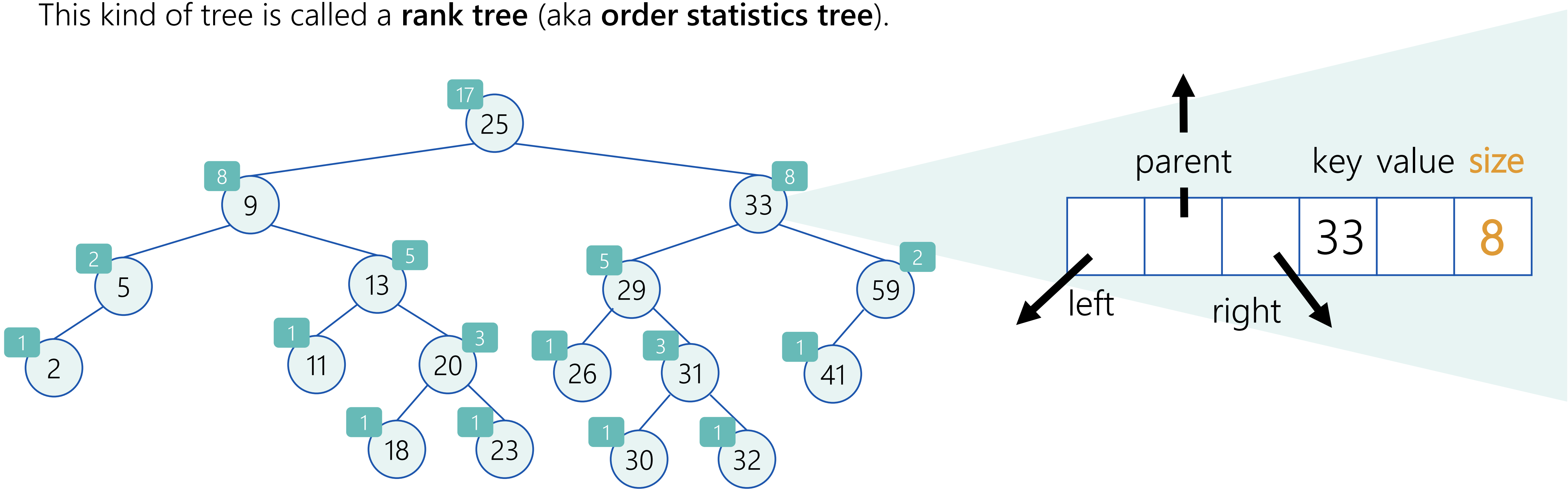
The *size* Attribute

Rank Trees: the *size* attribute

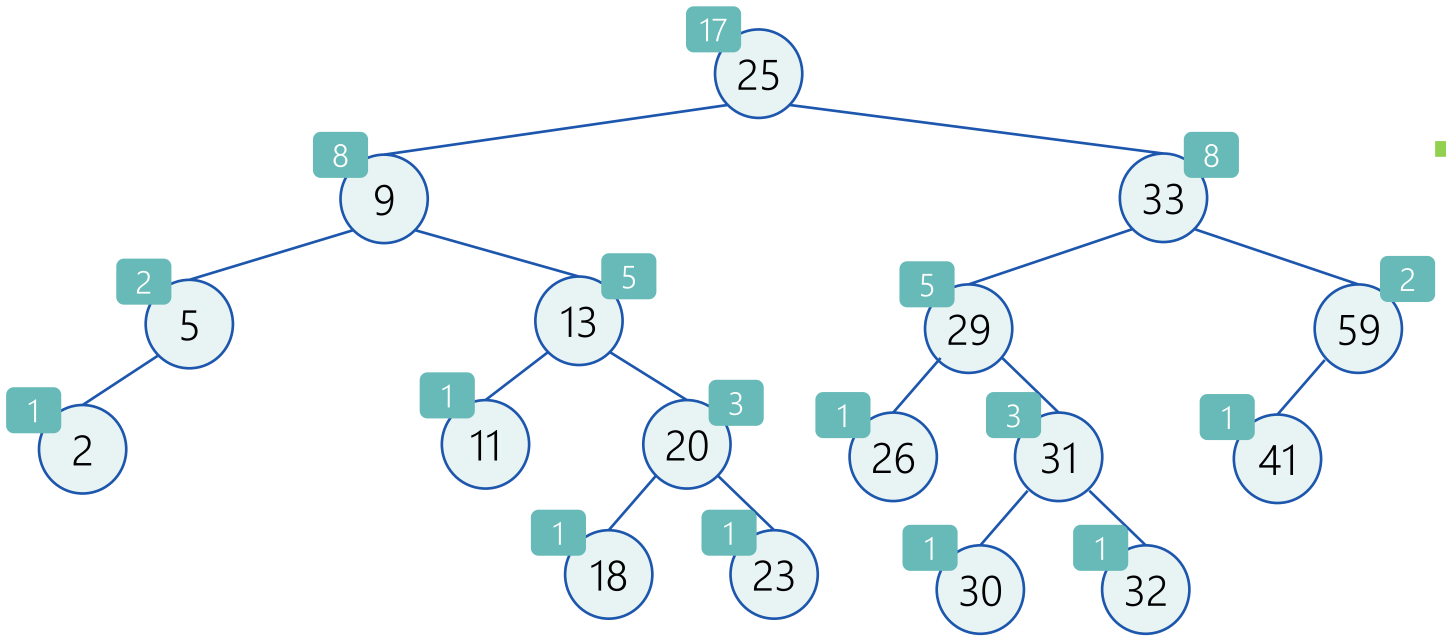
We will use an AVL tree, in which each node v will have a new attribute called *size*.

This attribute will hold the number of nodes in v's subtree (including v itself).

This kind of tree is called a **rank tree** (aka **order statistics tree**).



Rank Trees



Open issues:

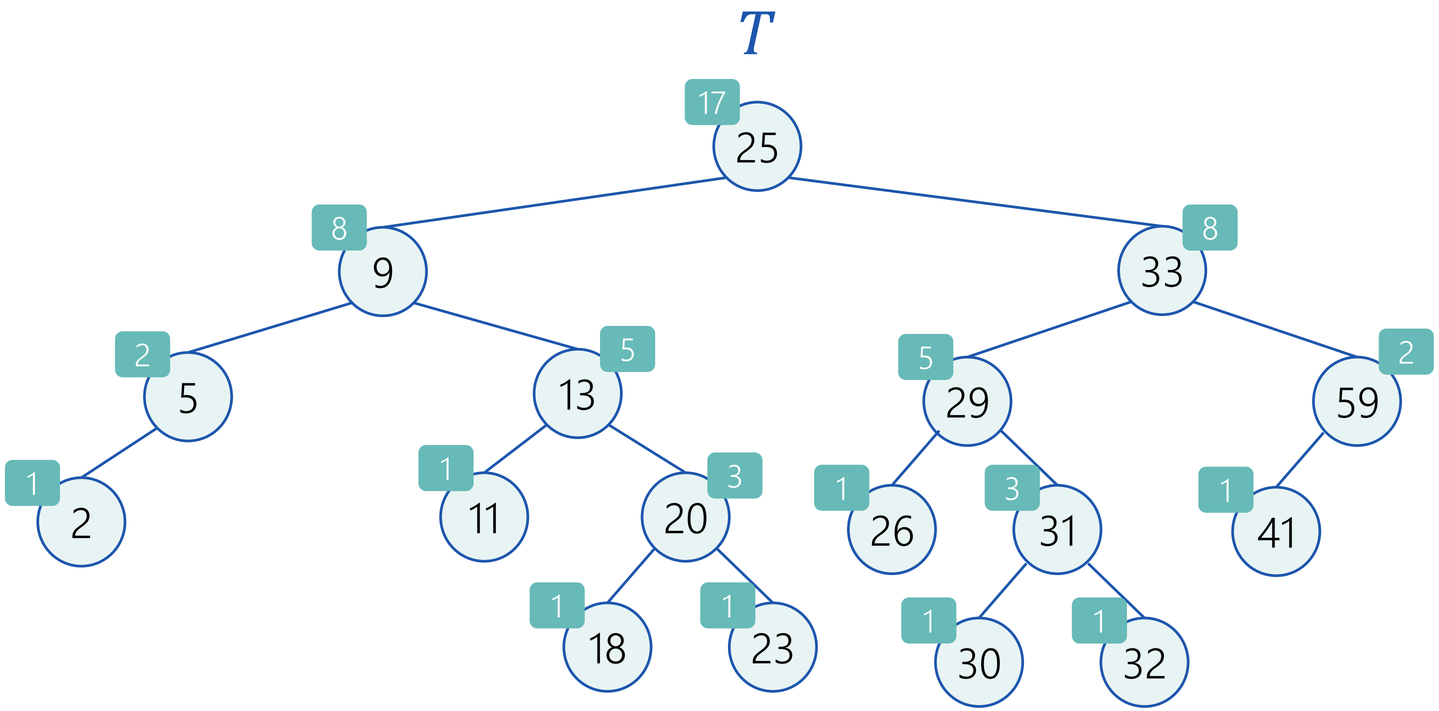
- How to implement **Select**?
- How to implement **Rank**?
- Does it increase the time required for **Insertion and Deletion**?

Tree-Select

Tree-Select

Example

Tree-Select(T,13)

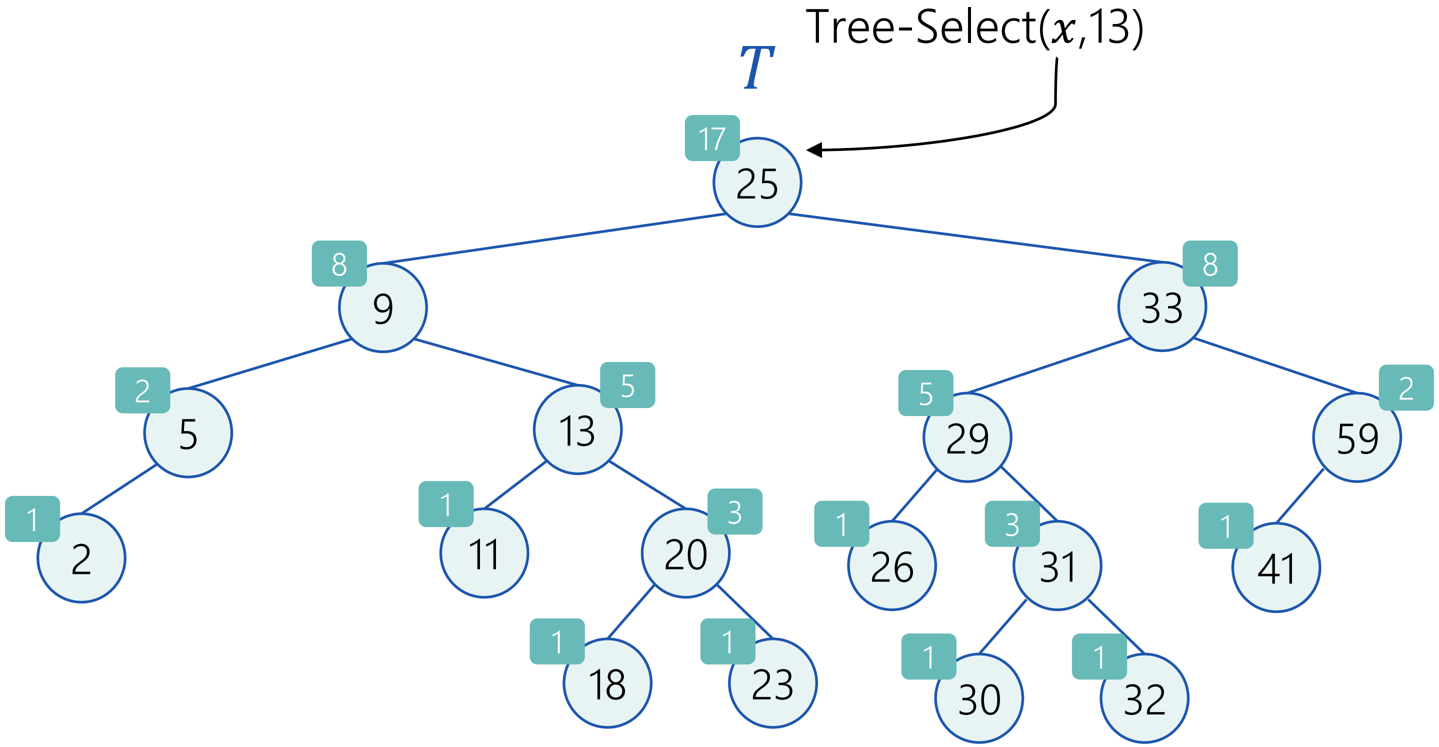


0 0

Tree-Select

Example

Tree-Select(T,13)

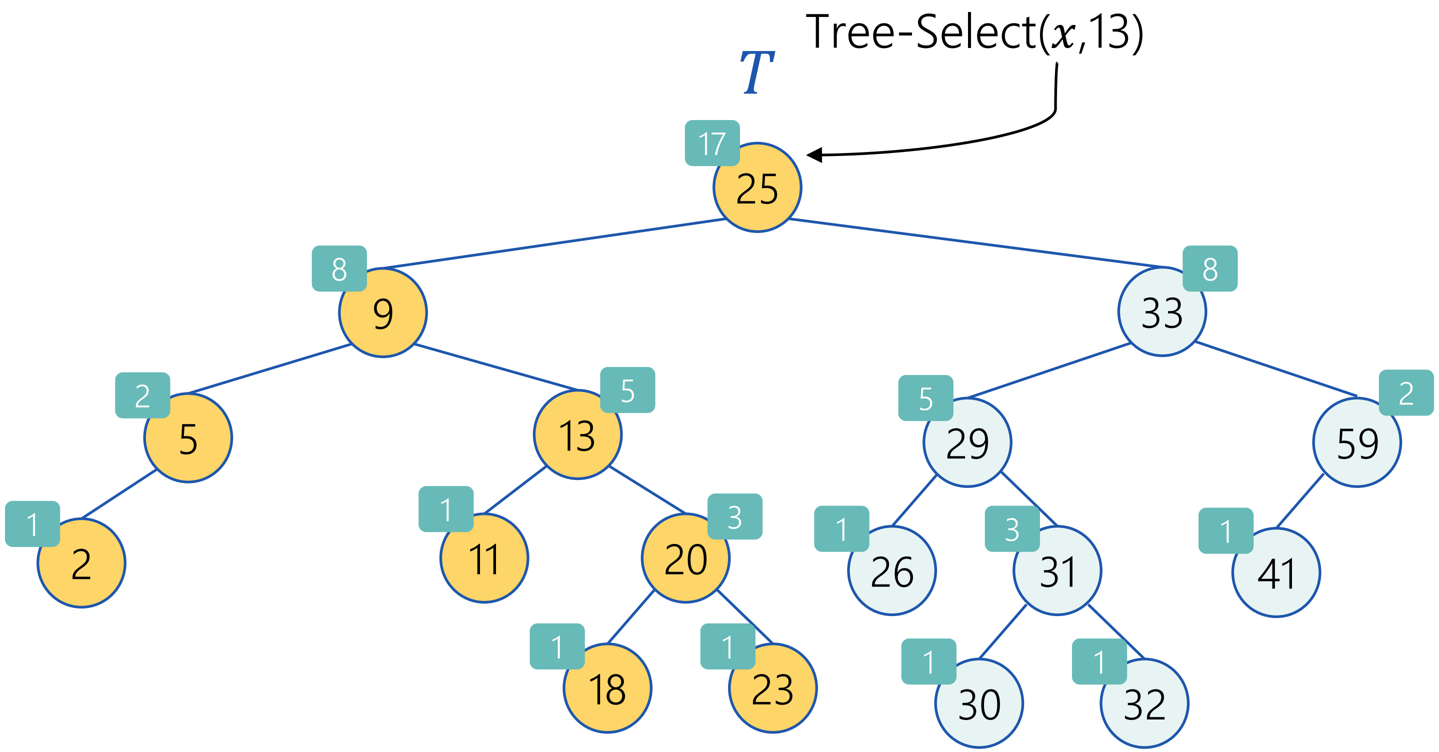


0 | 0

Tree-Select

Example

Tree-Select(T,13)

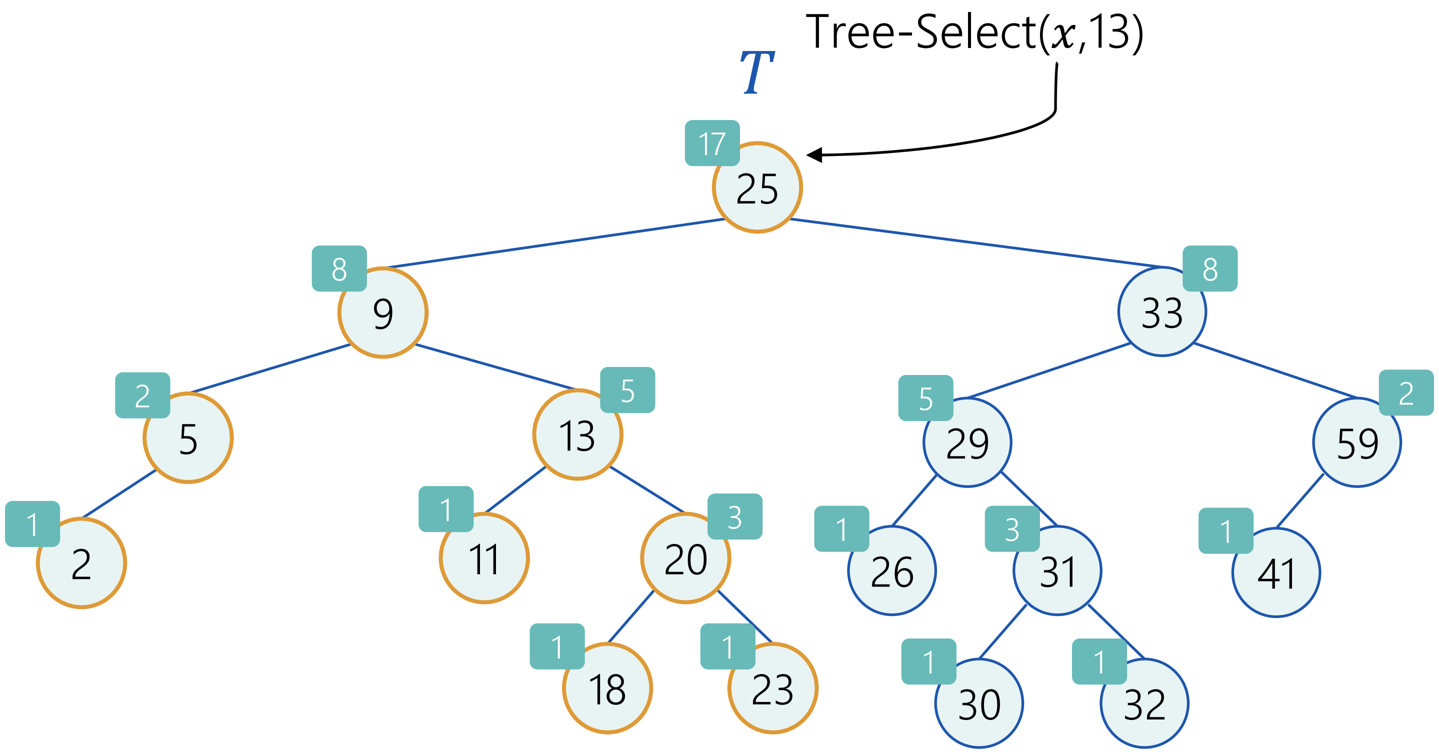


0	9

Tree-Select

Example

Tree-Select(T,13)

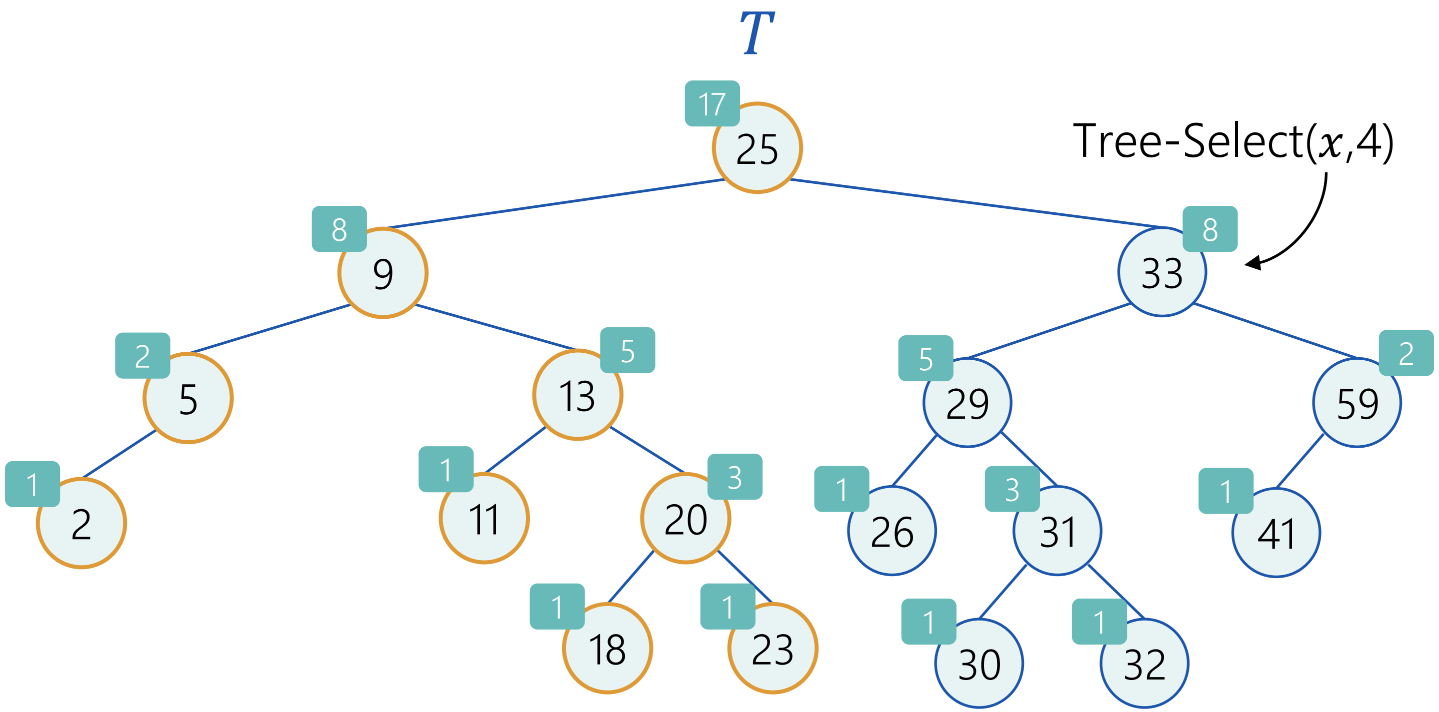


0	9

Tree-Select

Example

Tree-Select(T,13)

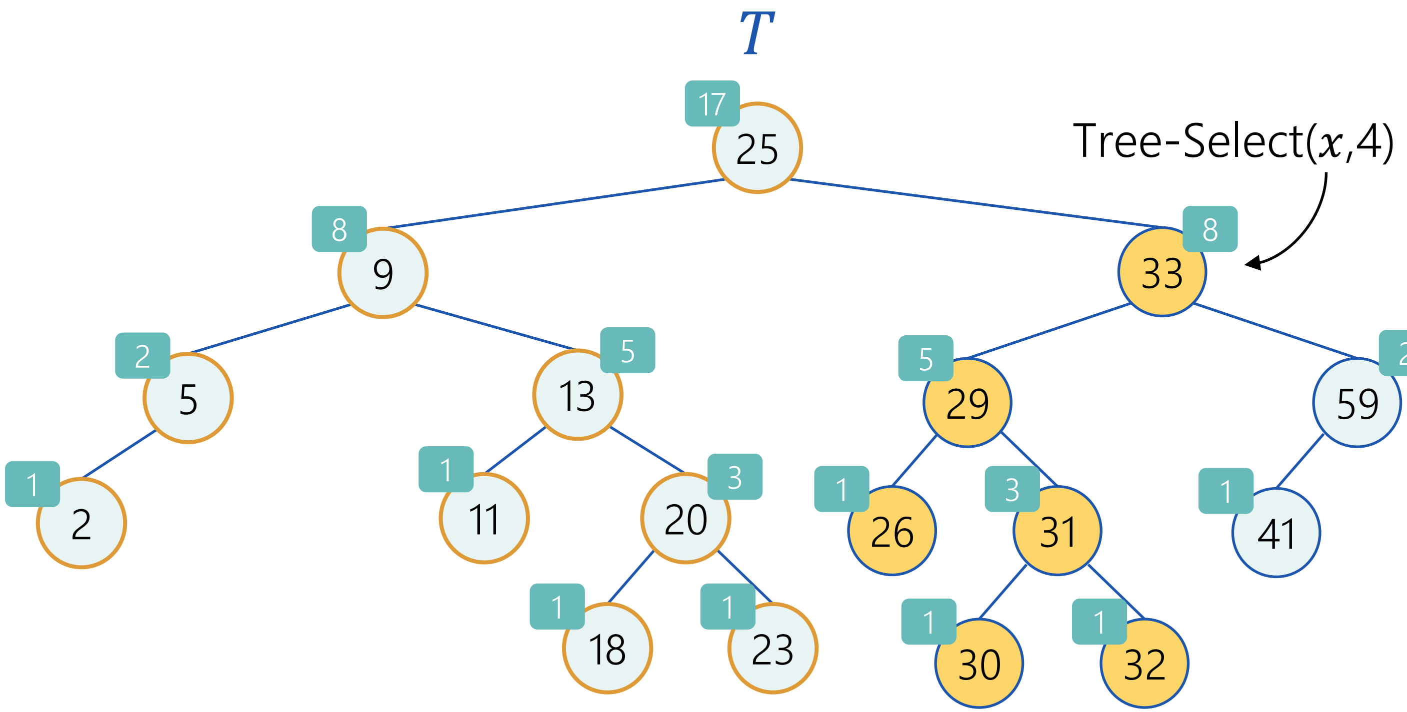


0	9

Tree-Select

Example

Tree-Select(T,13)

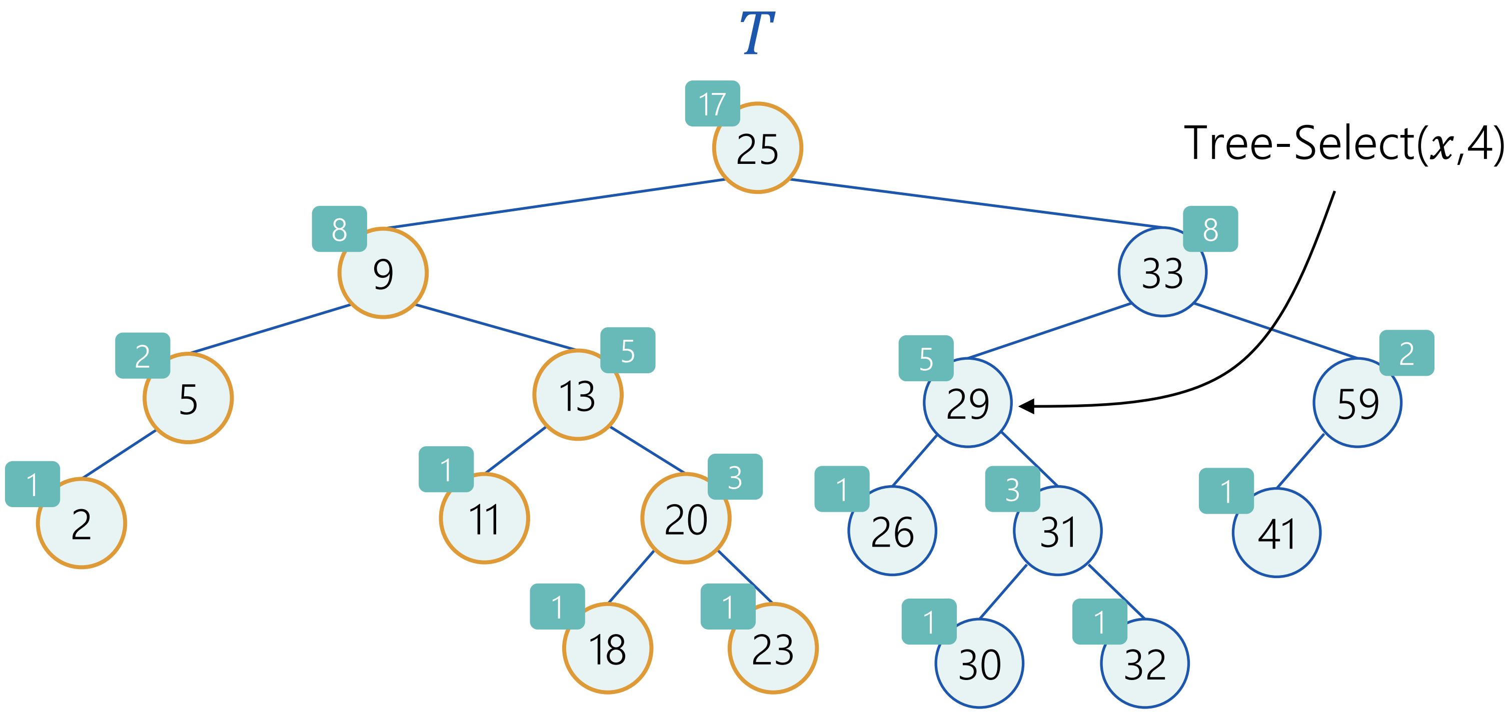


0	9

Tree-Select

Example

Tree-Select(T,13)

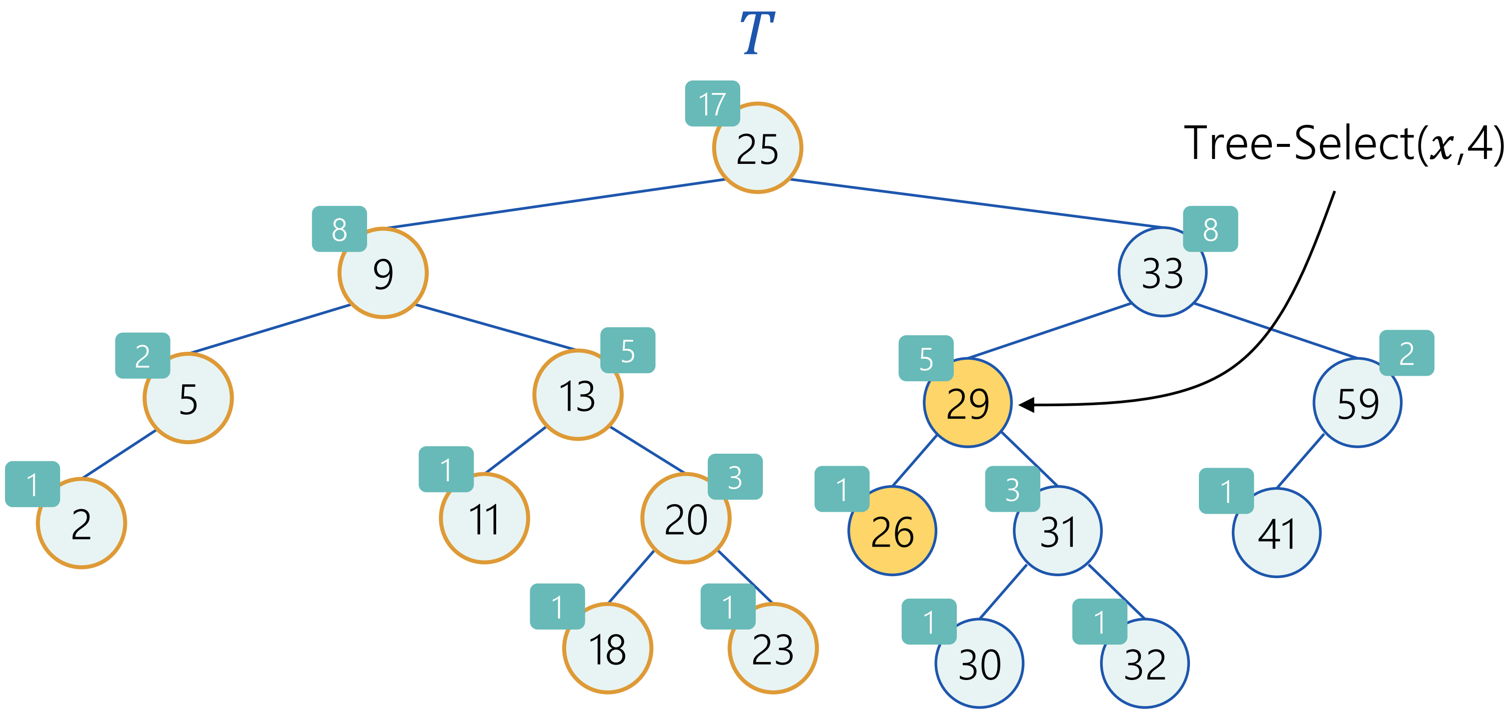


0	9

Tree-Select

Example

Tree-Select(T,13)

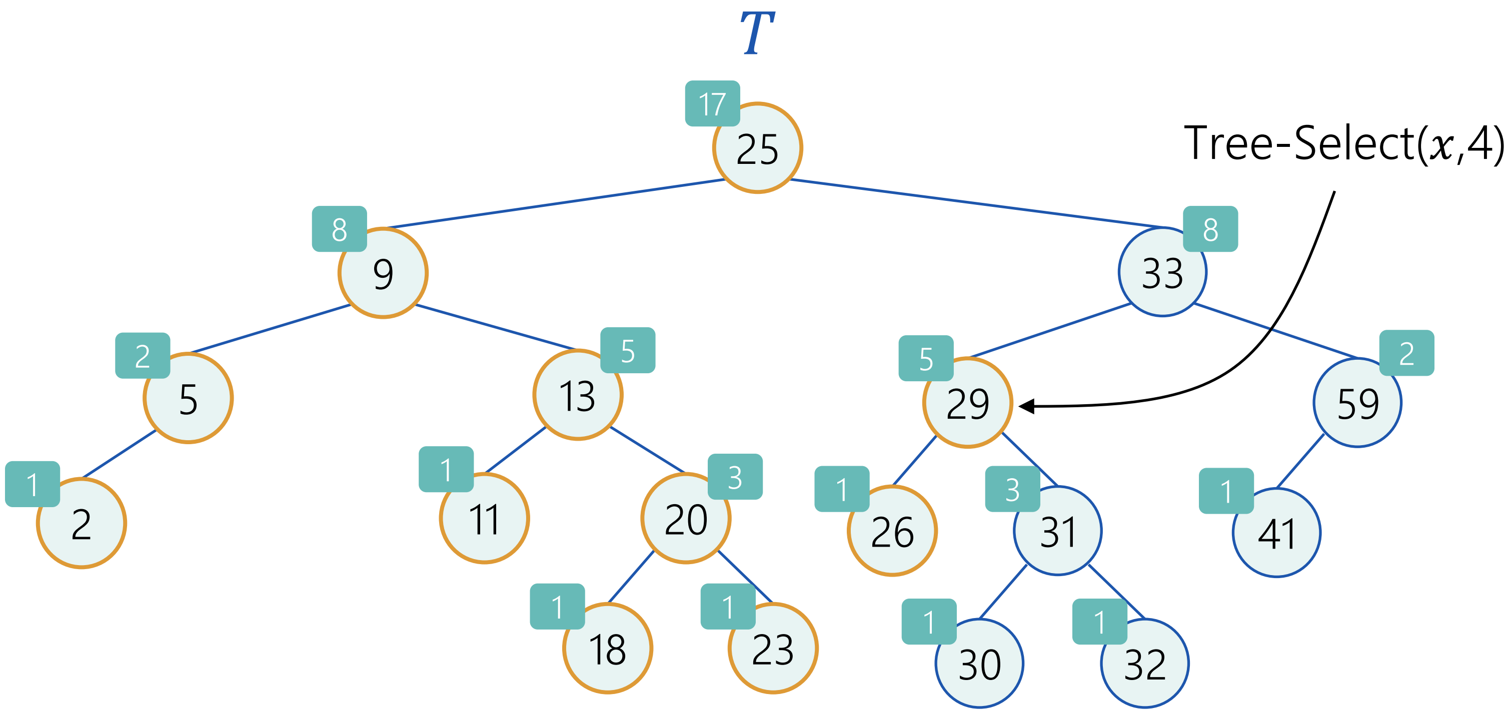


1	1

Tree-Select

Example

Tree-Select(T,13)

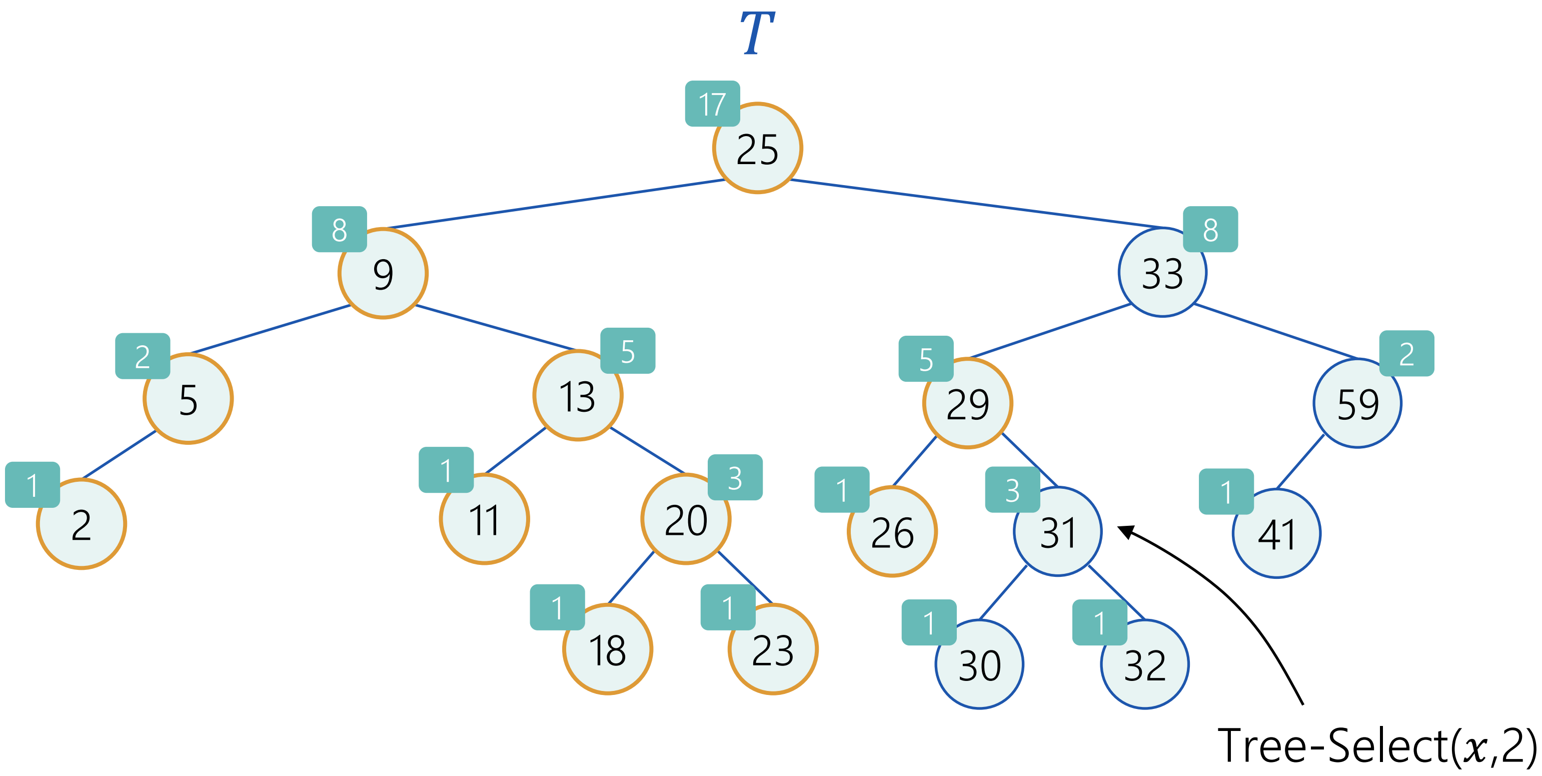


Tree-Select

Example

Tree-Select(T,13)

1	1

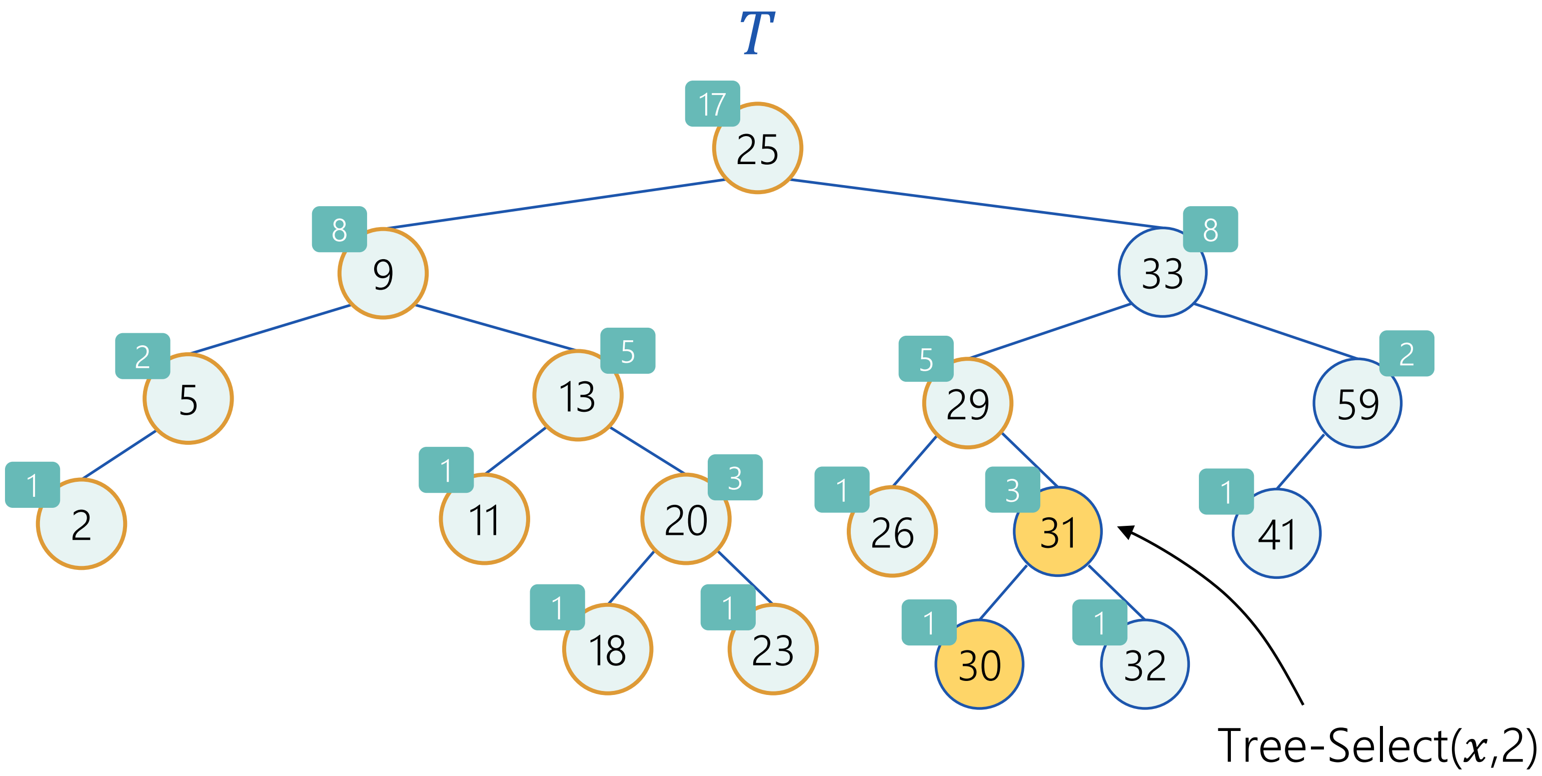


1	1

Tree-Select

Example

Tree-Select(T,13)

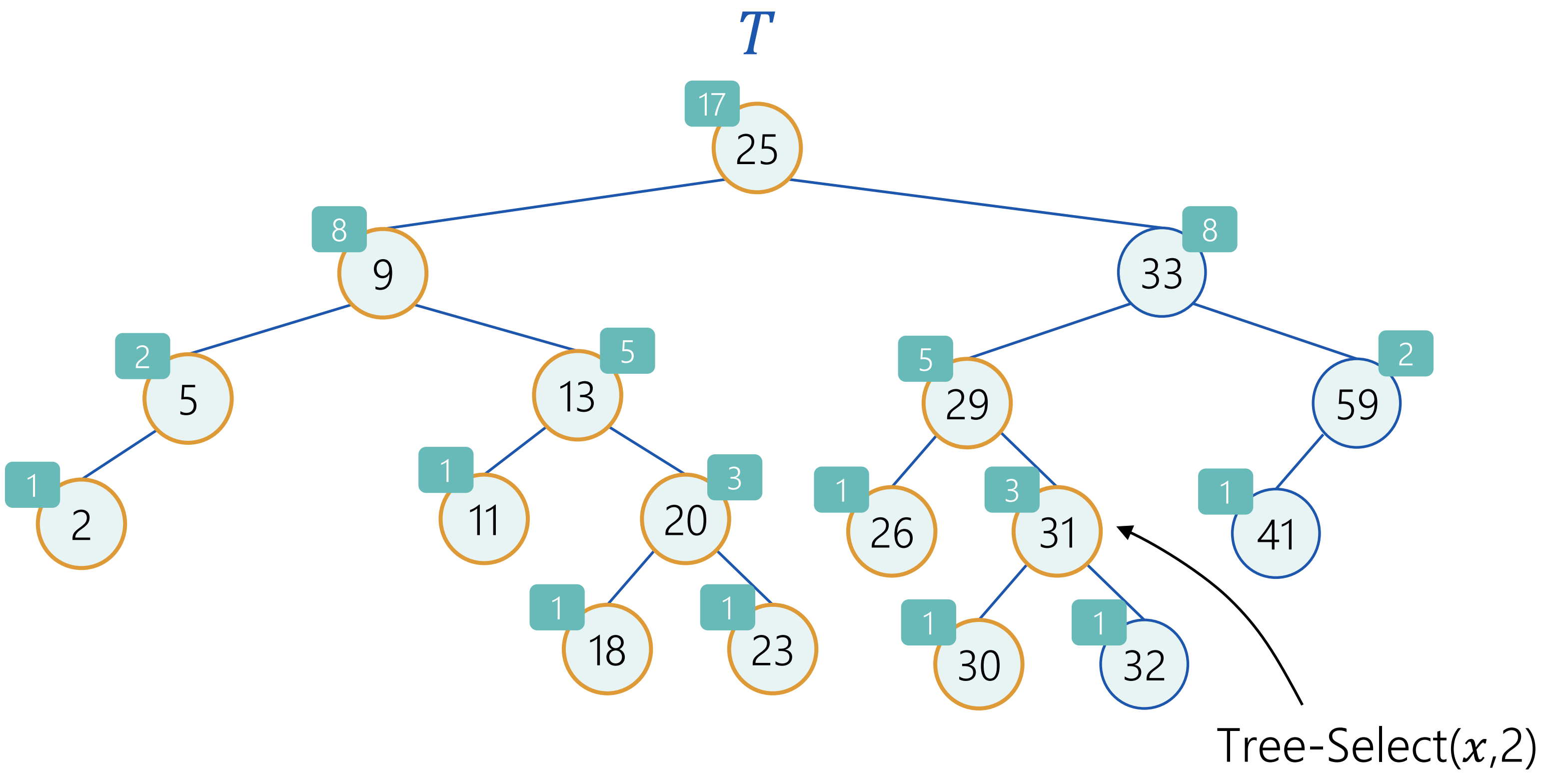


1	3

Tree-Select

Example

Tree-Select(T,13)



Return 31

Tree-Select

The algorithm

Tree-Select(T, k)

1. $x \leftarrow T.root$
2. $r \leftarrow x.left.size + 1$
3. If $k = r$, then the root is the required element
4. Otherwise, if $k < r$, search for the k^{th} smallest item in the **left** subtree of the root
5. Otherwise ($k > r$), search for the $(k - r)^{\text{th}}$ smallest item in the **right** subtree of the root

Tree-Select

The Algorithm in Pseudo-Code

Function Tree-Select(T, k)

1. return Tree-select-rec$(T.root, k)$

Function Tree-select-rec(x, k)

1. $r \rightarrow x.left.size + 1$
2. if $k = r$
   - 2.1 return x
3. else if $k < r$
   - 3.1 return Tree-Select-rec$(x.left, k)$
4. else return Tree-Select-rec$(x.right, k - r)$

Tree-Select

Complexity Analysis

Time Complexity

We “waste” constant time on each level of the tree. Therefore, the time complexity is linear in the height of the tree: $O(\log n)$.

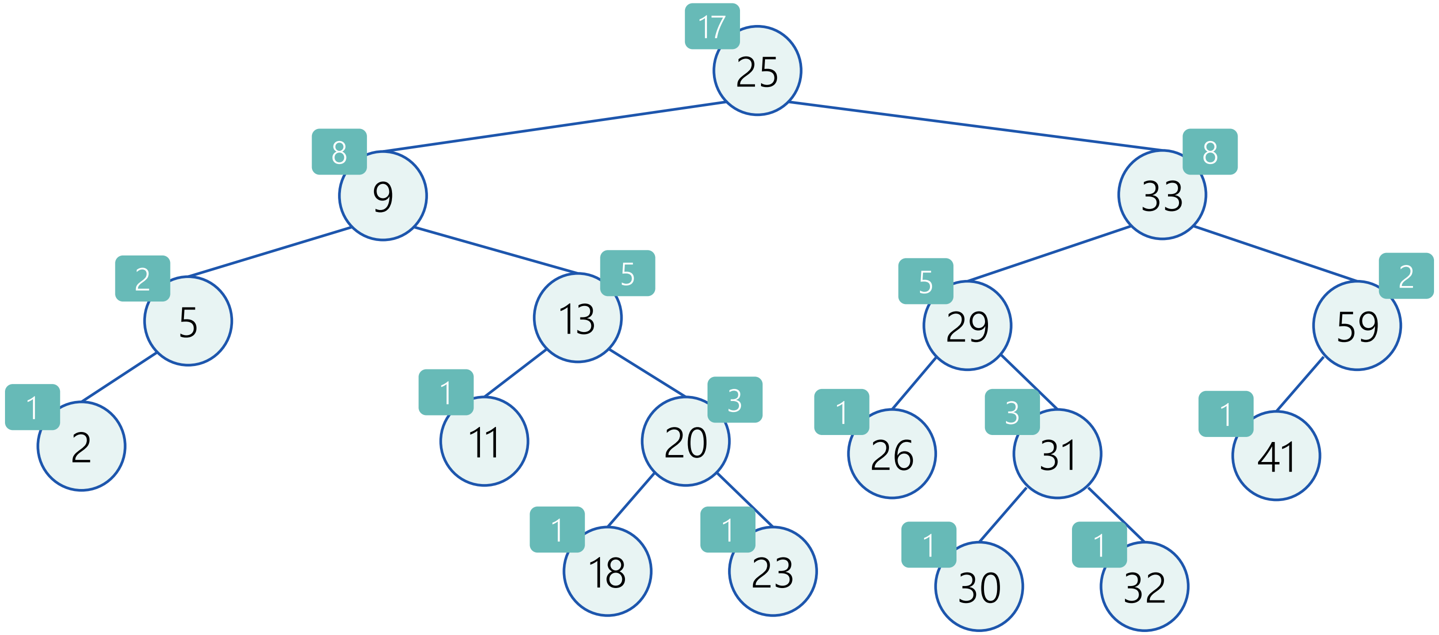
Additional Space Complexity *

Recursion stack- $O(\log n)$. However, an iterative version can be implemented, using $O(1)$ additional memory.

* Of course the size attribute requires $O(n)$ additional space

Tree-rank

Rank Trees

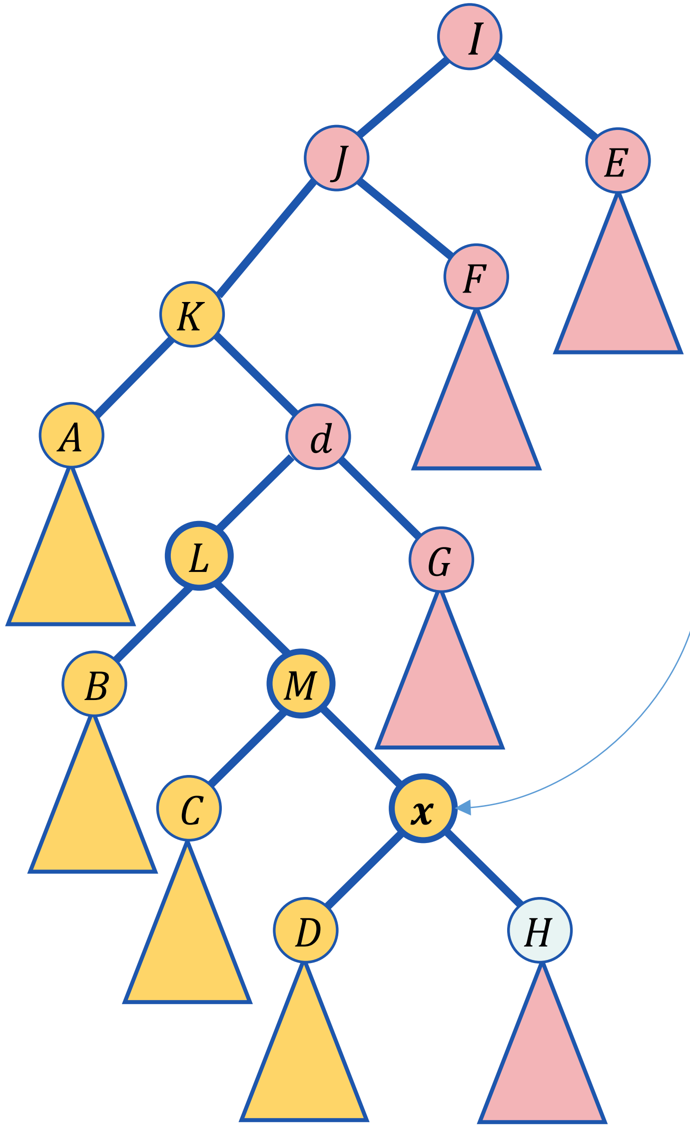


Open issues:

- How to implement **Select**?
- How to implement **Rank**?
- Does it increase the time required for **Insertion and Deletion**?

Tree-Rank

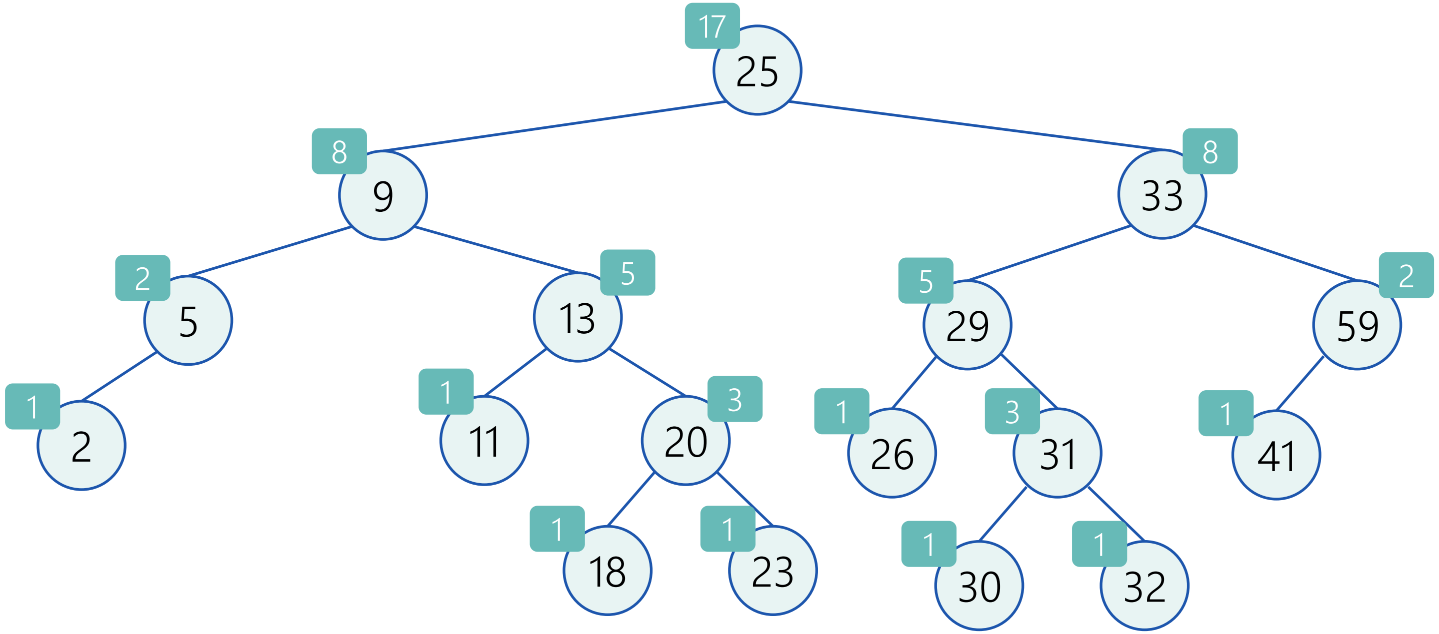
Example



Tree-Rank(x)=
size(D)+1 +
size(C)+1 +
size(B)+1 +
size(A)+1

Tree-Rank

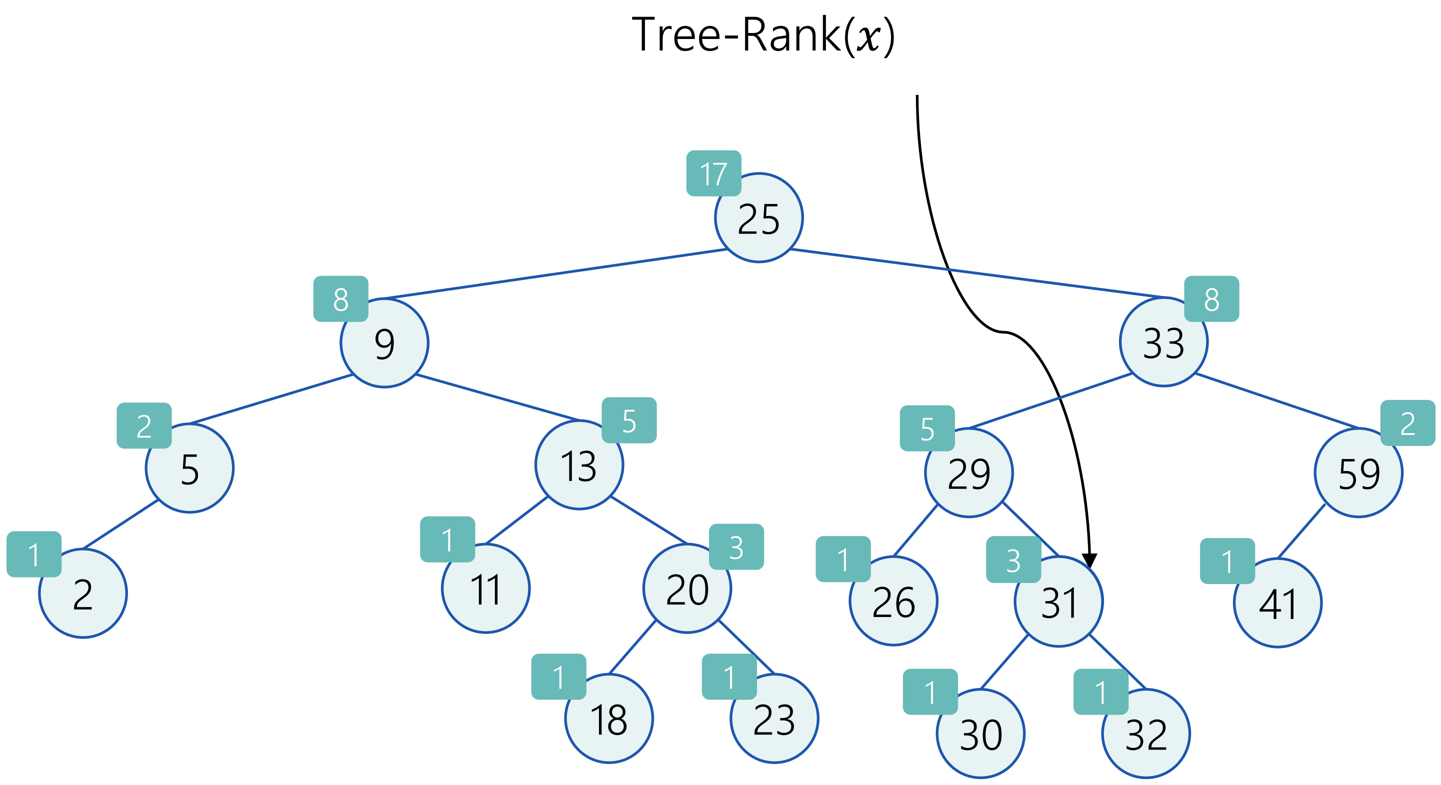
Given a pointer to a node, how can we find the node's rank in logarithmic time?



Tree-Rank

Example

Tree-Rank(31)

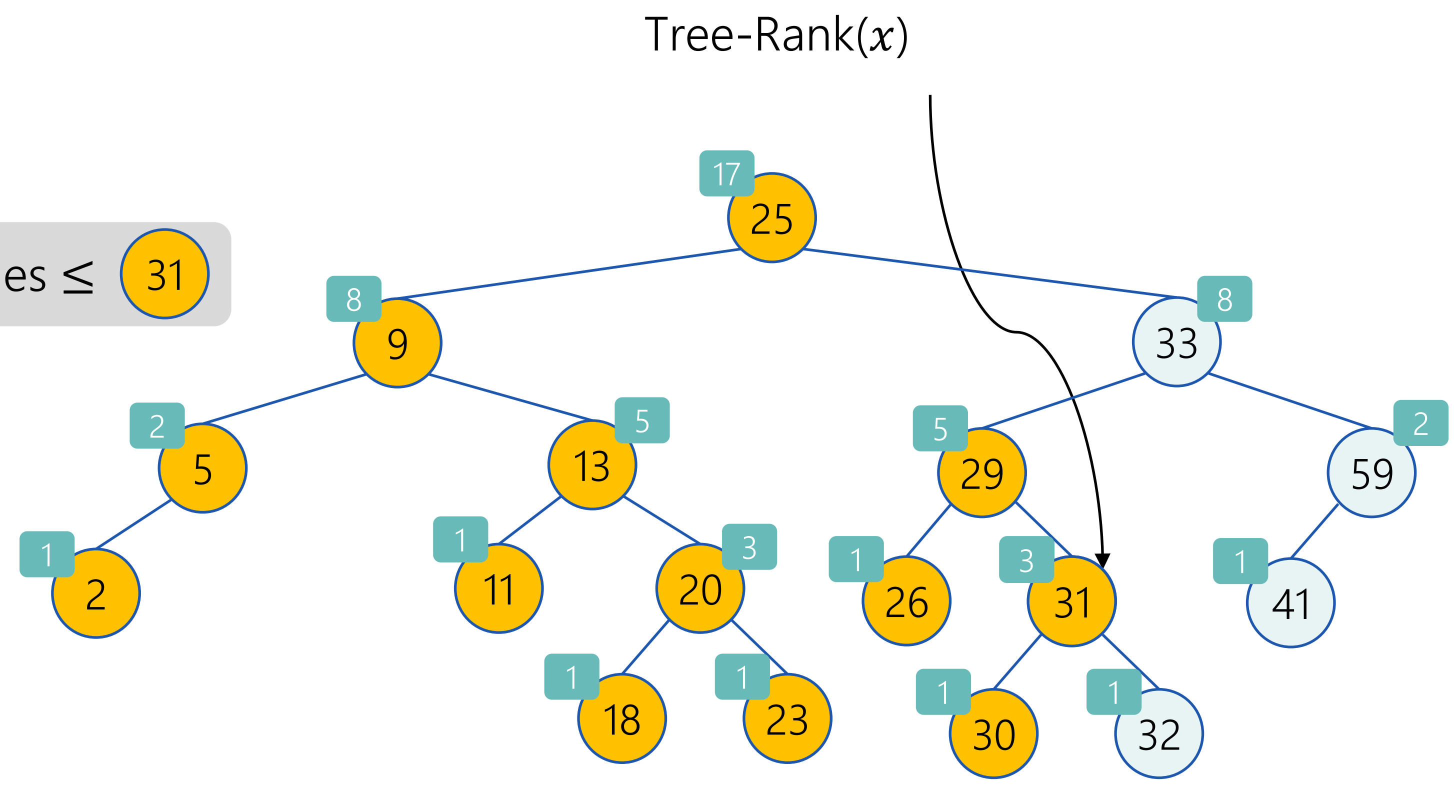


Tree-Rank

Example

Tree-Rank(31)

The orange nodes ≤ 31

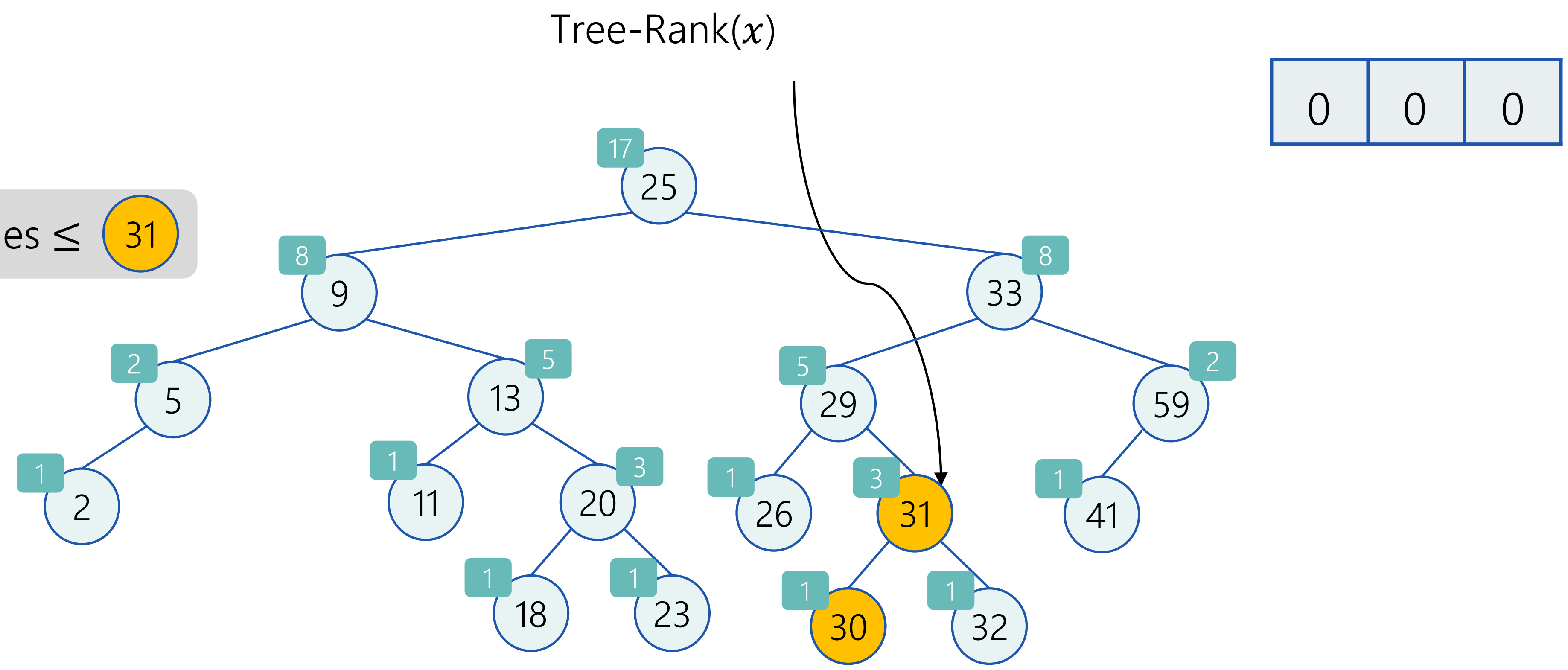


Tree-Rank

Example

Tree-Rank(31)

The orange nodes ≤ 31

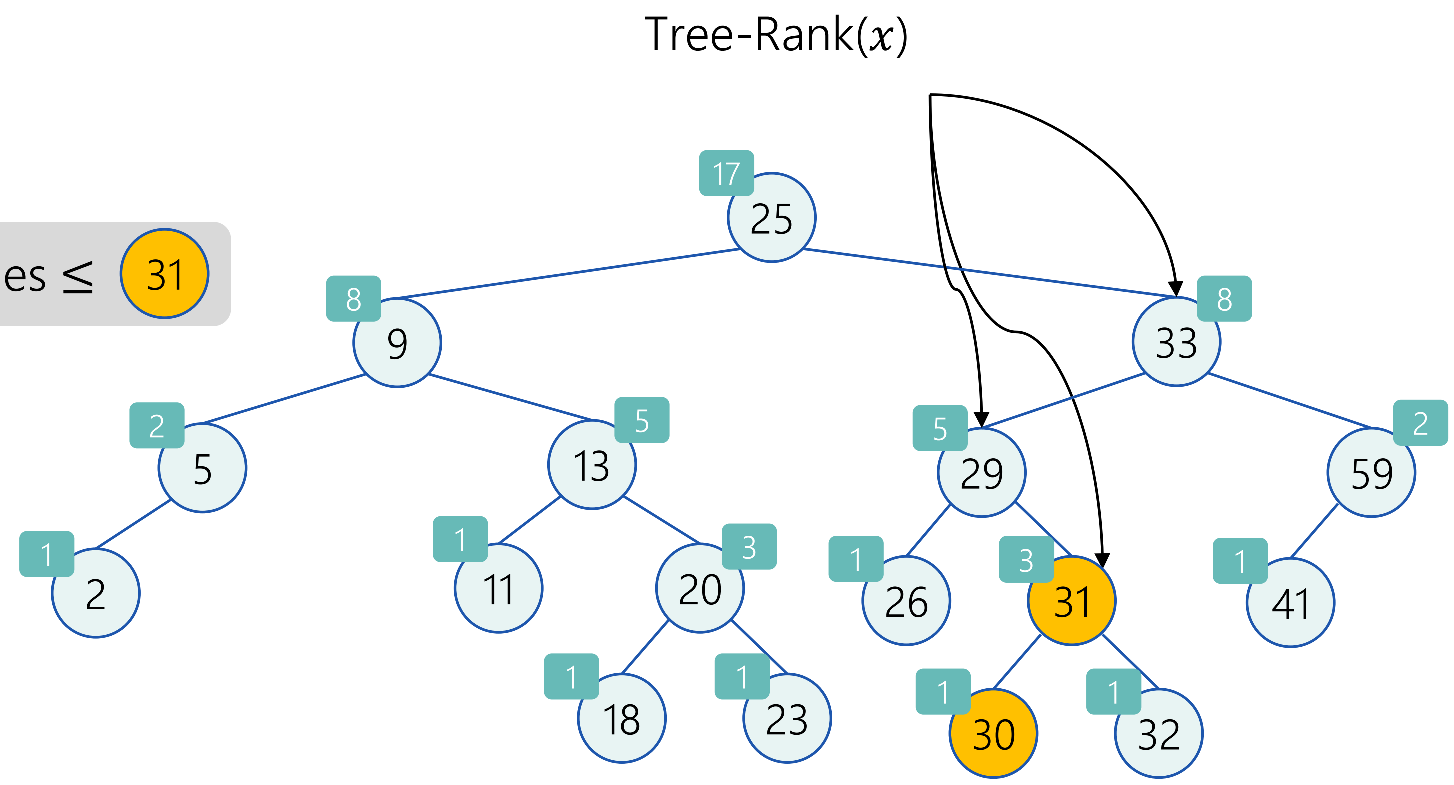


Tree-Rank

Example

Tree-Rank(31)

The orange nodes ≤ 31



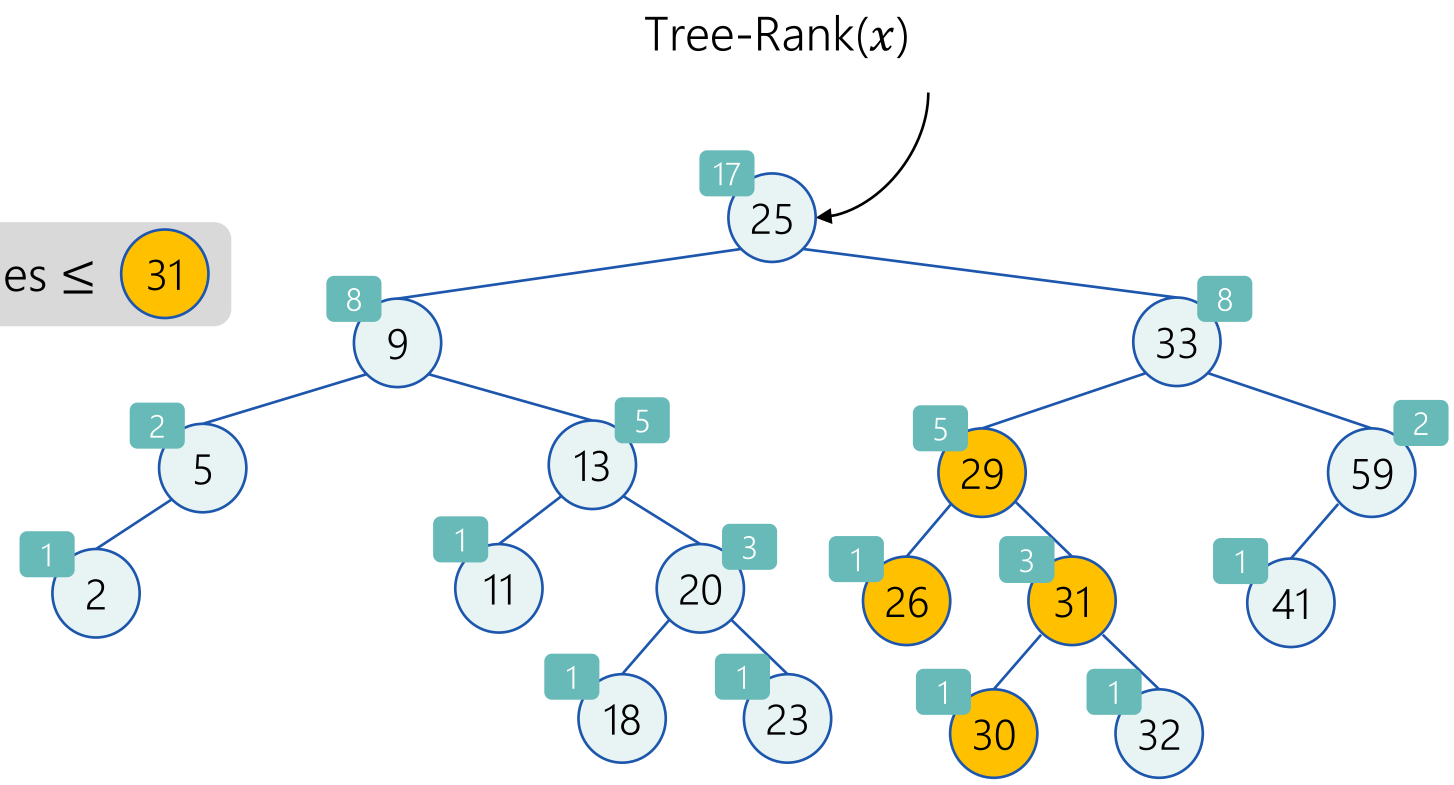
0	0	2

Tree-Rank

Example

Tree-Rank(31)

The orange nodes ≤ 31



0	0	4

Tree-Rank

The algorithm

Tree-Rank(x)

1. Initialize a counter with the number of nodes in x's left subtree, plus 1 (for x itself).
2. Go up to the root, and:
   2.1. Every time you go up left to a node y
   add the number of nodes in y's left subtree + 1
3. Return the final counter value

Tree-Rank

Pseudo-code

Function Tree-rank(x)

$r \leftarrow x.left.size + 1$
$y \leftarrow x$
while $y \neq null$
  if $y = y.parent.right$ # y is a right son
    $r \leftarrow r + y.parent.left.size + 1$
  $y \leftarrow y.parent$
return r

Tree-Rank

Complexity Analysis

Time Complexity

We “waste” constant time on each level of the tree. Therefore, the time complexity is linear in the height of the tree - $O(\log n)$.

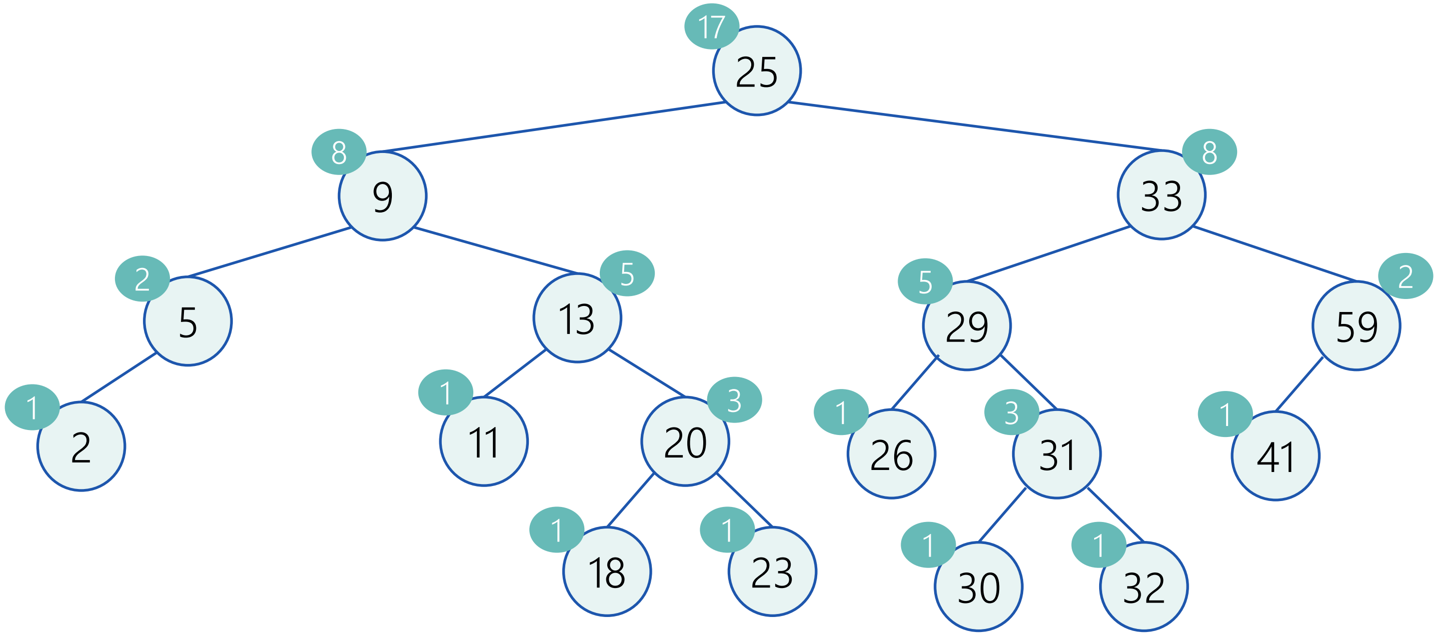
Additional Space Complexity

$O(1)$ assuming parent pointers exist

Rank Trees Insertion and Deletion

How to Maintain *size* During Insertion and Deletion?

Rank Trees



Open issues:

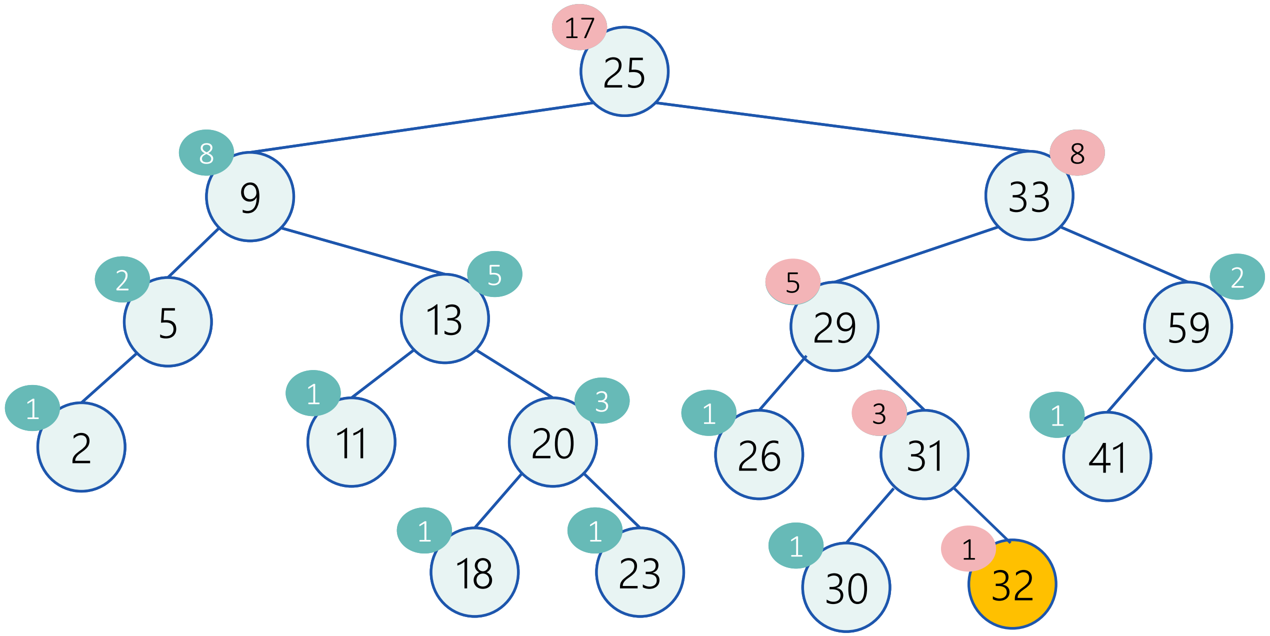
- How to implement **Select**?
- How to implement **Rank**?
- Does it increase the time required for **Insertion and Deletion**?

Maintaining the *size* Attribute After Insert and Delete

- So far, we've seen how adding the *size* attribute enables us to implement select and rank operations in logarithmic time.
- However, we now have to prove that after insert or delete, we can update the *size* attribute without increasing the complexity of these operations.

Maintaining the *size* Attribute After Insert

Tree-Insert(T, 32)

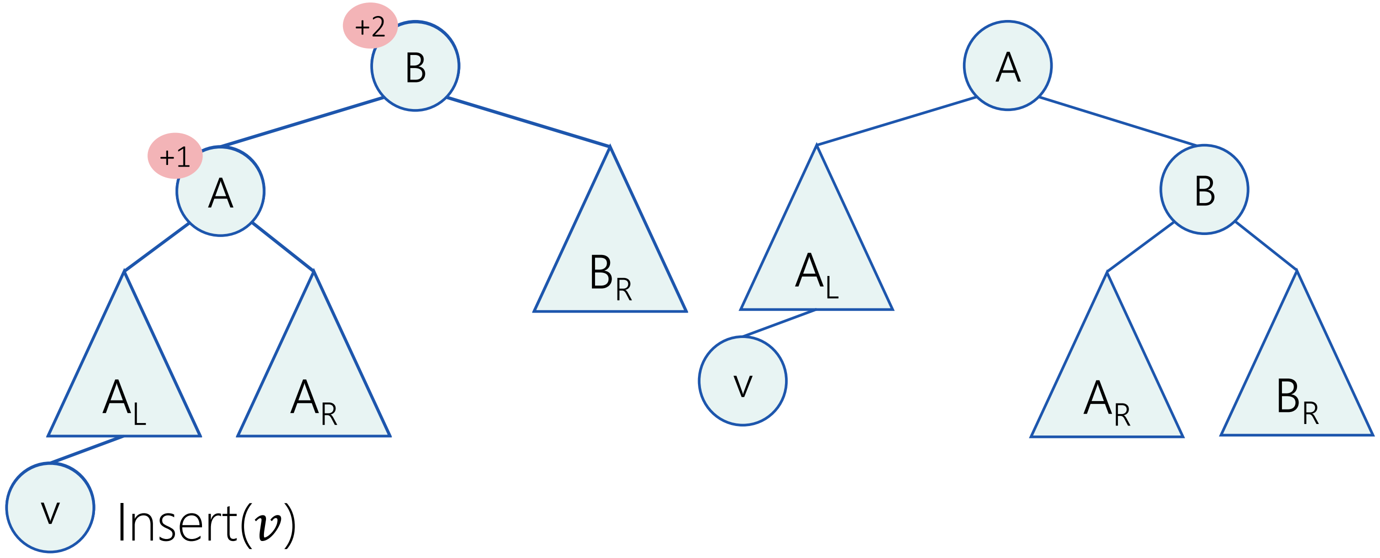


Maintaining the *size* Attribute After Insert

So for each node v on the path from the inserted node to the root:

$$v.size \mathrel{+}= 1$$

Is this all?
No! maybe we had a rotation.



Insert(v)

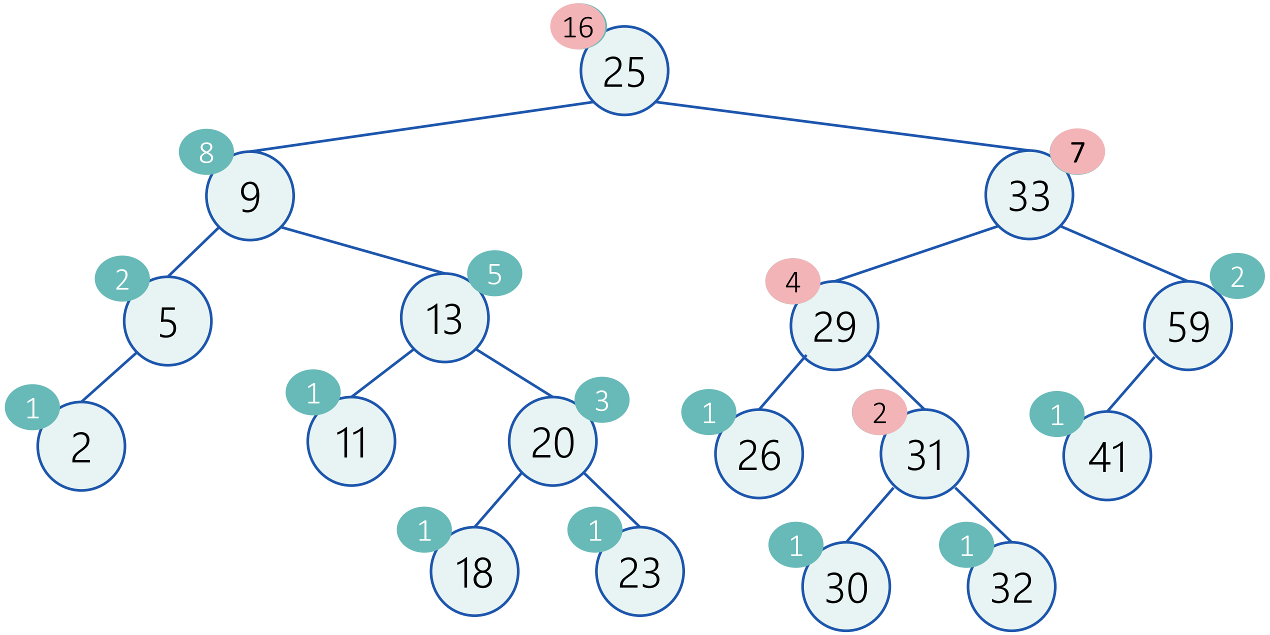
Right after insertion → After rotation

Function Right-Rotation
… *(update required pointers)*
$A.size \leftarrow B.size$
$B.size \leftarrow B.left.size + B.right.size + 1$

Other rotations similar

Maintaining the *size* Attribute After Delete

Tree-Delete(T, 32)



Maintaining the *size* Attribute After Delete

So for each node v on the path from the physically deleted node to the root:

$$v.size \ -= 1$$

Plus correct *size* of nodes involved in rotations.

Extending Data Structures

Sum-Up

Let's sum-up what we just did:

We wanted to implement an ADT (dictionary + Select + Rank), all operations in logarithmic time.
No previous data structure (or a combination of some) suffices.

For this:

1. We chose a familiar data structure as an infrastructure — AVL
2. We extended it by adding extra information — *size*
3. We showed how to implement the new operations — Tree-Select, Tree-Rank
4. We proved time complexity of dynamic operations remains the same — Insert, Delete

Which Attributes Can Be Maintained Efficiently in a Balanced Tree?

Question:

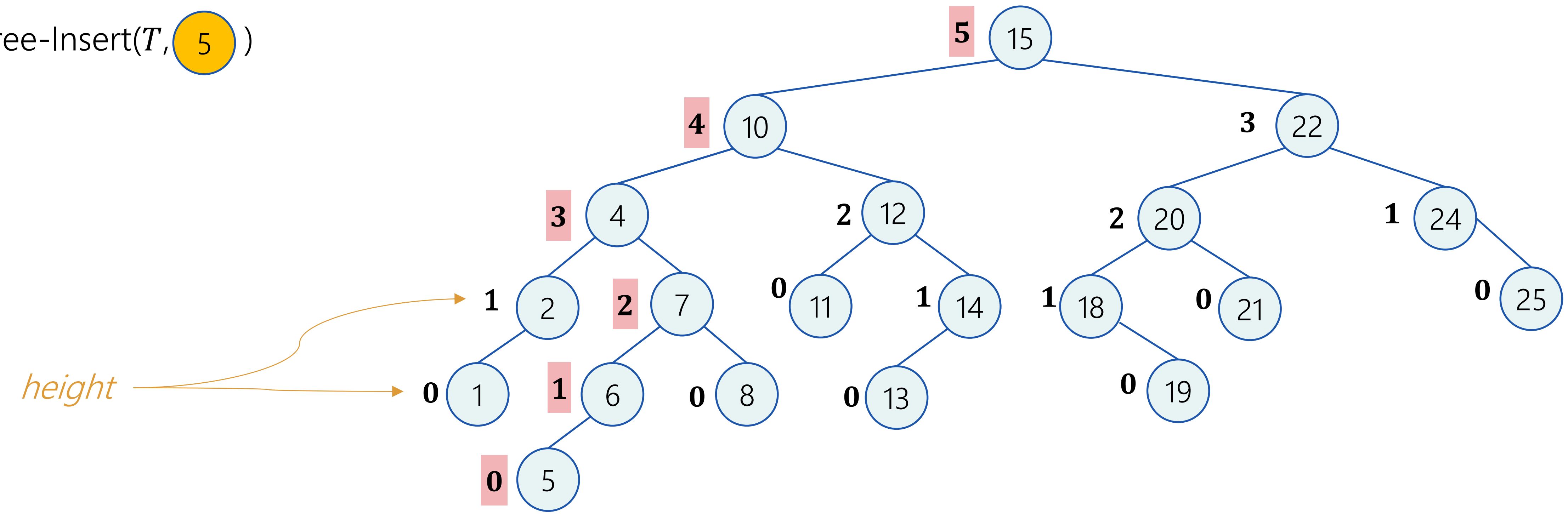
Can the heights of nodes be efficiently maintained during insertion and deletion?

Question:

Can the depths of nodes be efficiently maintained during insertion and deletion?

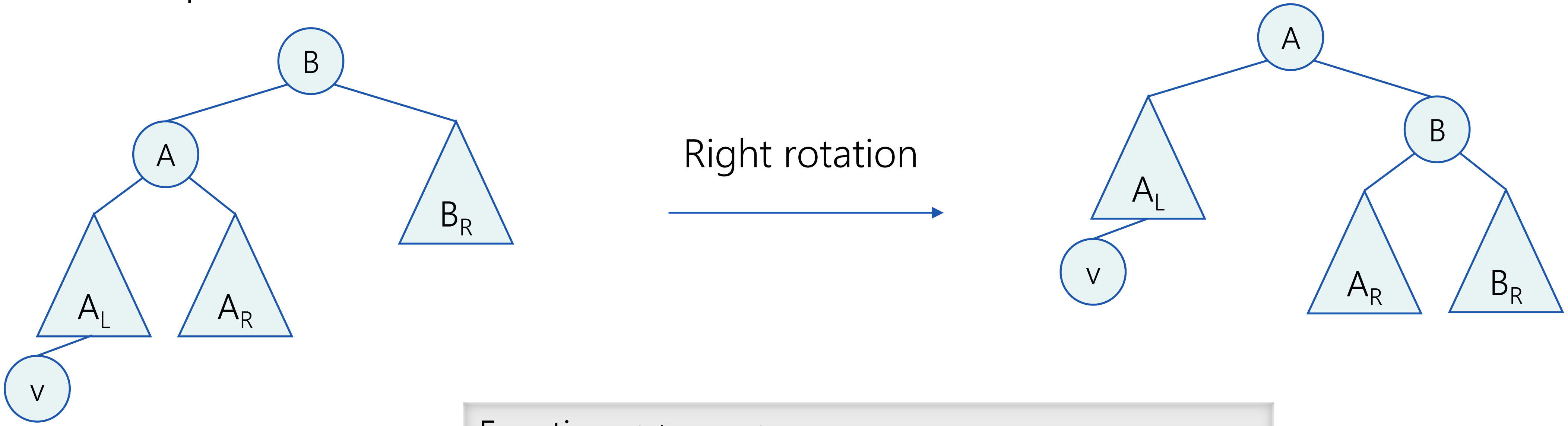
Maintaining *Height* During Insertions

Tree-Insert(T, 5)



Maintaining *Height* During Rotations

For example:



Function Right-Rotation

…

$B.height \leftarrow 1 + \max(B.left.height, B.right.height)$
$A.height \leftarrow 1 + \max(A.left.height, A.right.height)$

Other rotations similar

Which Attributes Can Be Maintained Efficiently in a Balanced Tree?

Question:

Can the heights of nodes be efficiently maintained during insertion and deletion?

Answer:

Yes. The height of a node depends only on the heights of its children.

- Therefore its possible to update this during the fix-up stage of insertion/deletion (each node along the path from the inserted/deleted node up-to the root requires O(1) extra time).
- Furthermore, there's a constant number of nodes whose height is changing during rotations.

Which Attributes Can Be Maintained Efficiently in a Balanced Tree?

Question:

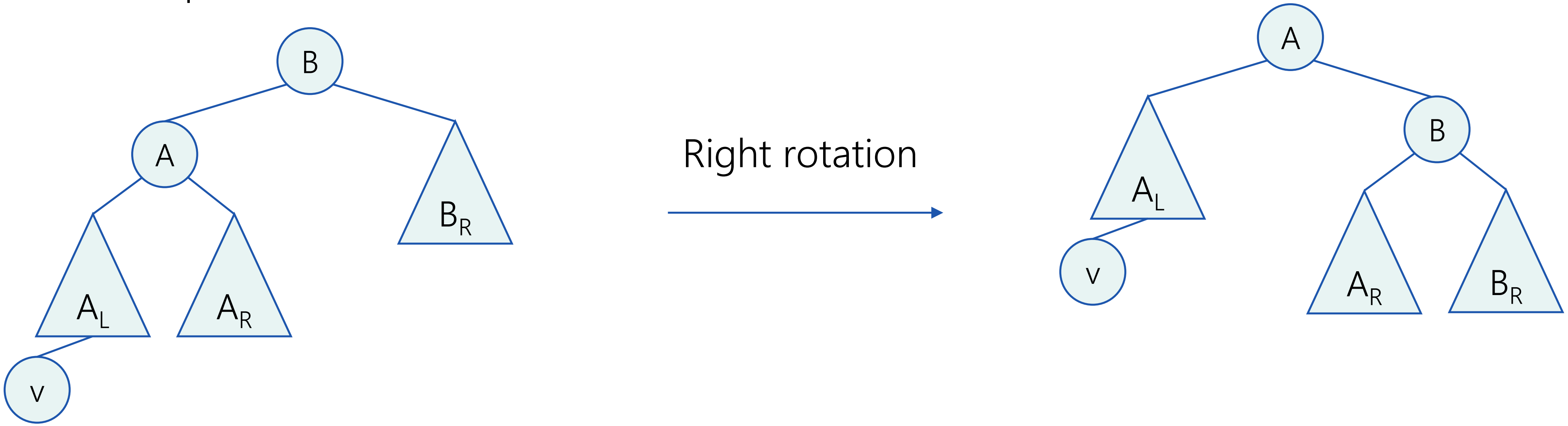
Can the depths of nodes be efficiently maintained during insertion and deletion?

Answer:

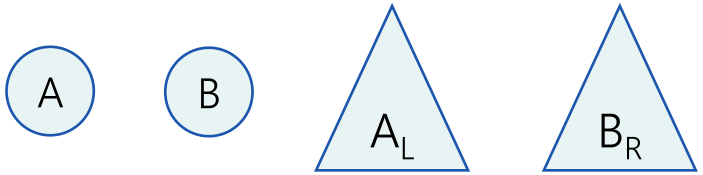
No. there can be scenarios in which the *depth* attribute of $\Theta(n)$ nodes needs to be updated.

Maintaining Depth During Rotations may Require $\Theta(n)$ time

For example:



Depths change:



Which Attributes Can Be Maintained Efficiently in a Balanced Tree?

Theorem

Let f be an attribute that extends an AVL tree T with n nodes.

If the information maintained in a certain node (including its f attribute) can be computed merely from its direct children,

then we can maintain f values after insertion/deletion within the $O(logn)$ time complexity of these operations.

Note: the theorem mentions only a sufficient condition, not a necessary one.

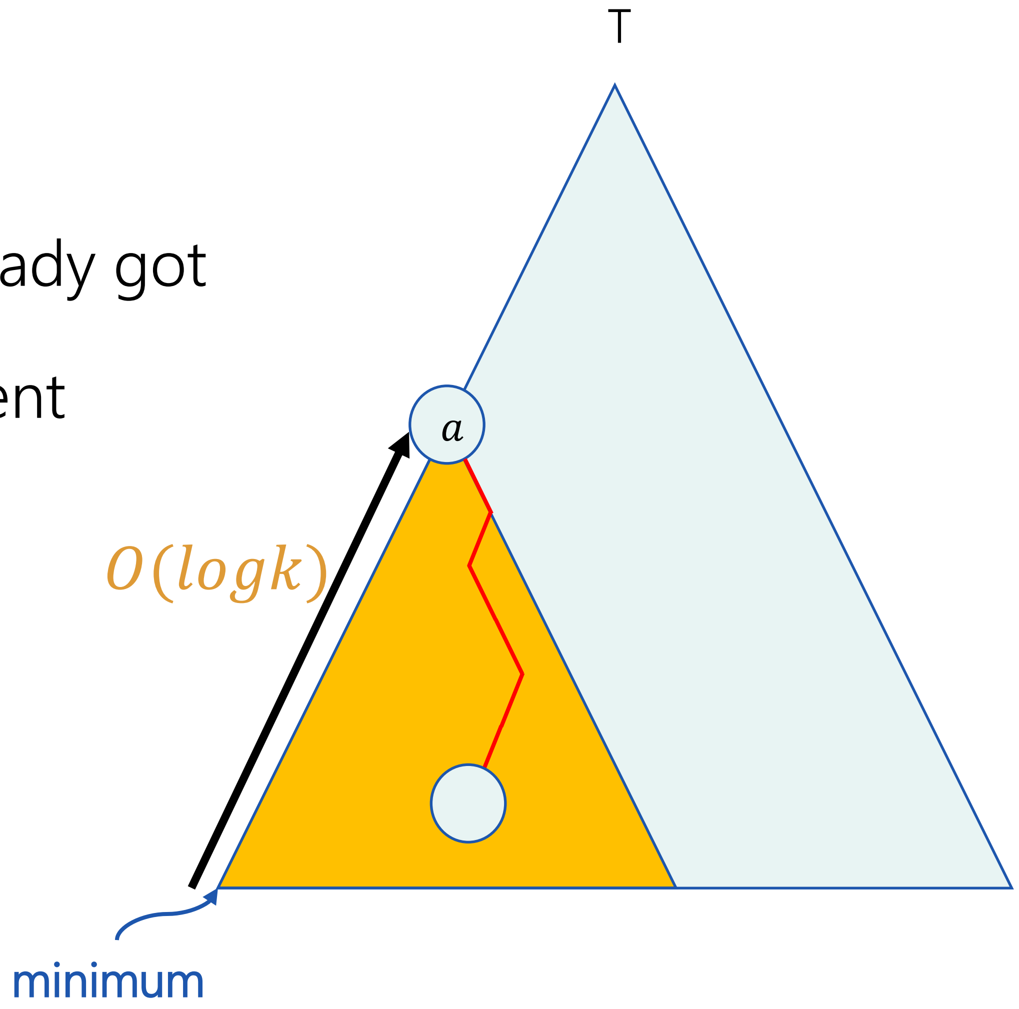
Finger Trees

Finger Trees

- We want $Select(T, k)$ in $O(\log k)$ time
- If $k = o(n)$ this beats the $O(logn)$ we already got

Finger Trees

- We want $Select(T, k)$ in $O(\log k)$ time
- If $k = o(n)$ this beats the $O(logn)$ we already got
- Rank tree + pointer to the minimum element
  (update it in insert/delete!)
- Go up until subtree has at least k nodes
  Denote this node a $(a.size \geq k)$
- Call Tree-Select(a, k)
- Complexity? $O(logk)$



Joining and Splitting AVL Trees

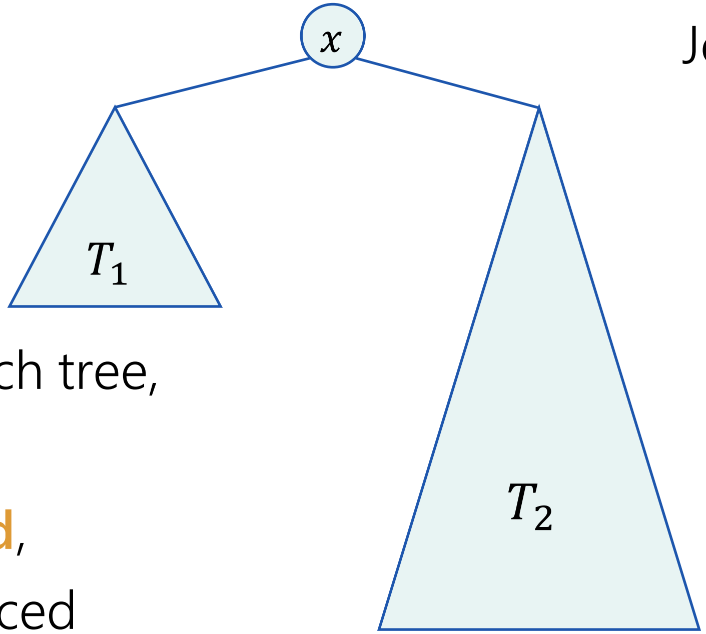
Joining Trees

Joining Two AVL's

Suppose that for all nodes $x_1 \in T_1$ and $x_2 \in T_2$
x_1.key < x.key < x_2.key

"$T_1 < x < T_2$"

Join(T_1,x,T_2) in $O(1)$

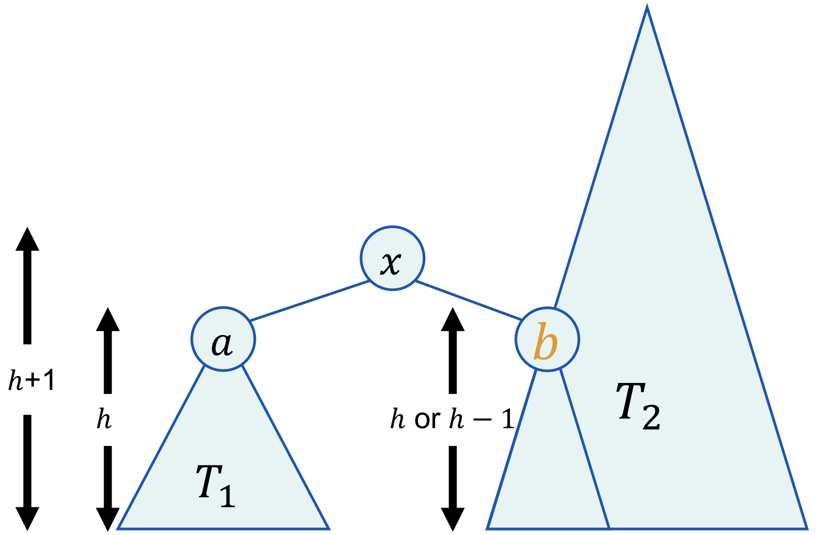


The tree formed is a valid search tree,
but
may be **very unbalanced**,
even if T_1 and T_2 are balanced

Joining Two AVL Trees Efficiently, Maintaining Balance

Join(T_1,x,T_2) when "$T_1 < x < T_2$"

Assume $\boldsymbol{height}(T_1) \leq \boldsymbol{height}\,(T_2)$



Idea: $x.right$ will be a subtree of T_2 of similar height as T_1

Denote $\boldsymbol{height}(T_1) = h$

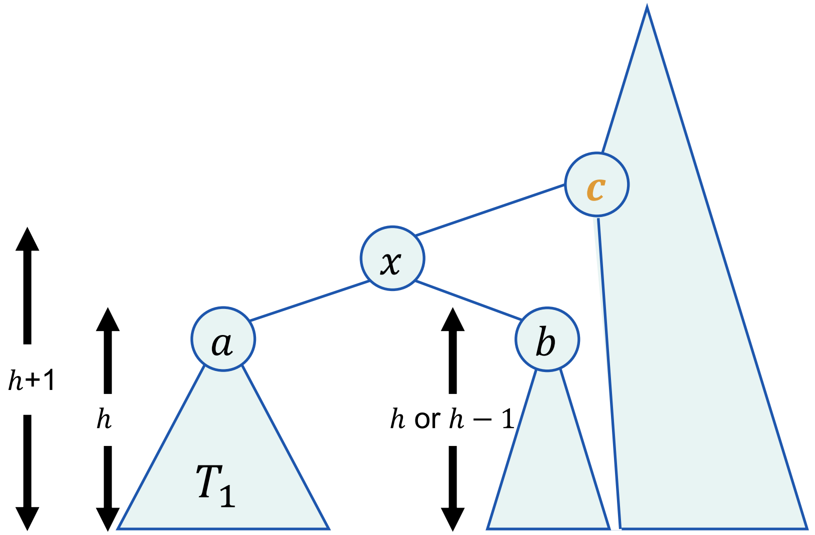
$\boldsymbol{b}$ – first vertex on the left spine of T_2 with $\boldsymbol{height} \leq h$

$\boldsymbol{height}(b) = h$ or $h - 1$

Joining Two AVL Trees Efficiently, Maintaining Balance

Join(T_1,x,T_2) when "$T_1 < x < T_2$"

Assume $height(T_1) \leq height(T_2)$



Attach x to b's former parent (denoted c)

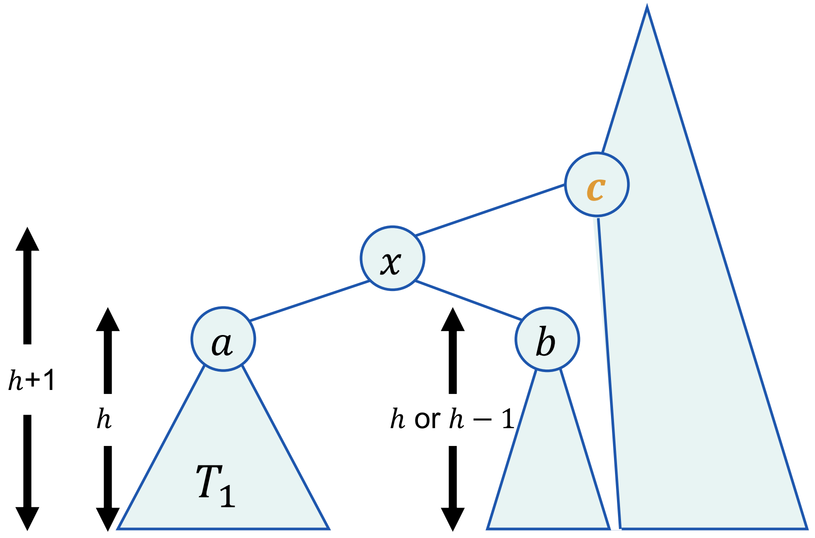
Do rebalancing from x upwards, if needed

(consider all possible cases for c's new BF and height)

Joining Two **AVL** Trees Efficiently, Maintaining Balance

Join(T_1,x,T_2) when "$T_1 < x < T_2$"

Assume $height(T_1) \leq height(T_2)$



$O(\log n)$ time

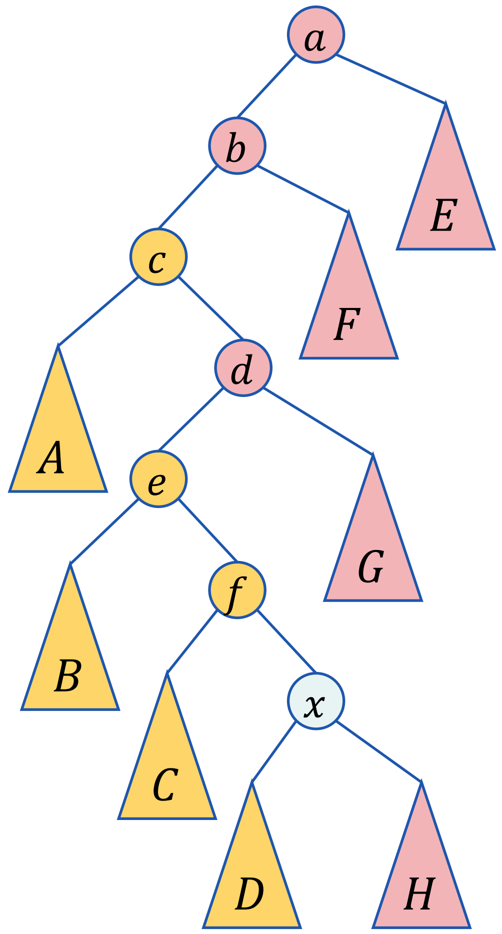
And even $O(height(T_2) - height(T_1) + 1)$ time

(if heights maintained explicitly, so we can **find b** while going **down**)

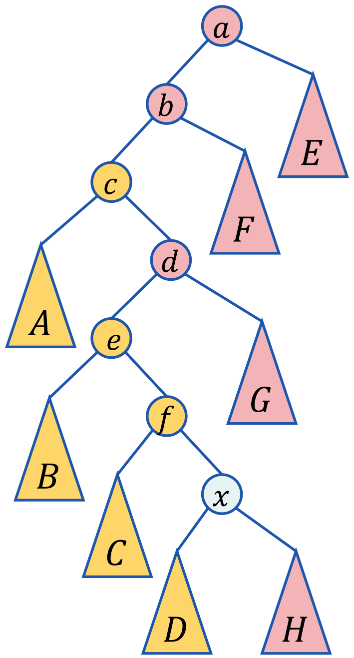
Splitting Trees

Splitting AVL (by Joins)

Given a BST and a node x,
we want to split the tree into T_1, T_2
such that "$T_1 < x < T_2$"



Splitting AVL (by Joins)



Given a BST and a node x,
we want to split the tree into T_1, T_2
such that "$T_1 < x < T_2$"

Time:

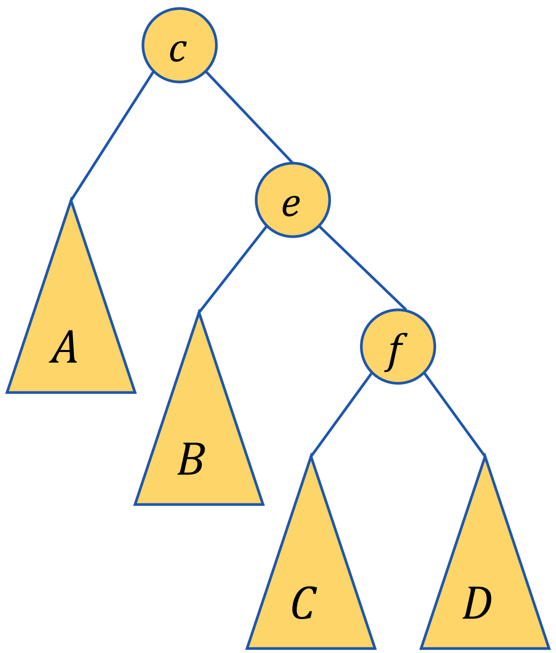
Using simple joins in $O(1)$:

$O(\log n)$, balance may break

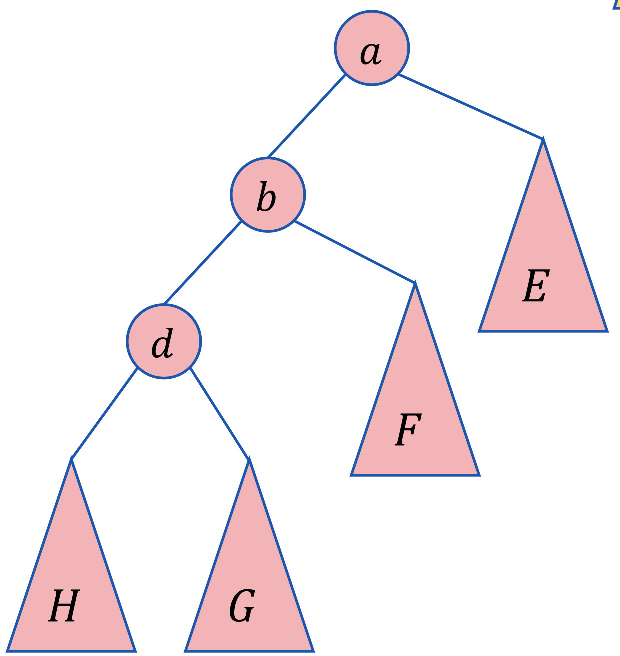
Using smart joins (figure here does not show it): $O(\log^2 n)$, balance maintained.

But…

T_1



T_2

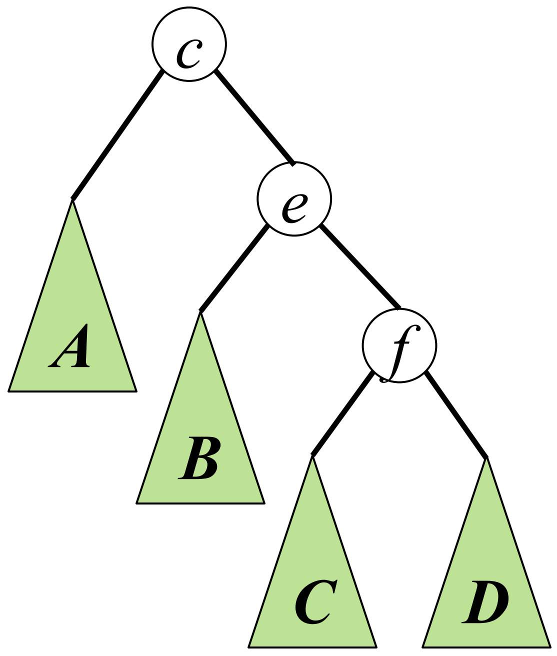


$T_1 = \text{Join}(A, c, \text{Join}(B, e, \text{Join}(C, f, D)))$ $\quad T_2 = \text{Join}(\text{Join}(\text{Join}(H, d, G), b, F), a, E)$

Splitting with efficient joins

Suppose we need to join $T_1, T_2, \ldots, T_k$

Claim: $height(T_1) \leq height(T_2) \leq \ldots \leq height(T_k)$

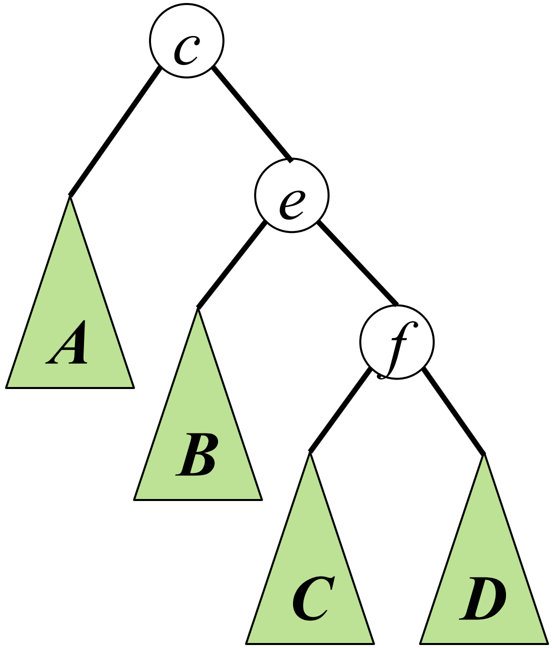


Splitting with efficient joins

Claim: *height*(Join(T_1,…,T_i)) ≤ *height*(T_i) + *c*

Proof:

We prove *height*(Join(T_1,…,T_i)) ≤ *height*(parent (T_i)).



Splitting with efficient joins

Claim: $height(\text{Join}(T_1,\ldots,T_i)) \leq height(T_i) + c$

Proof:

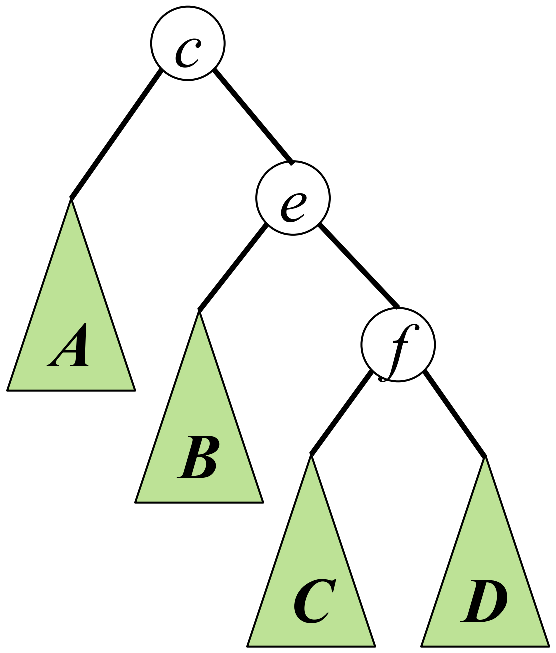
We prove $height(\text{Join}(T_1,\ldots,T_i)) \leq height(\text{parent }(T_i))$.

Base case i=2,

$height(\text{Join}(T_1,T_2)) \leq \max\{height(T_1), height(T_2)\} +1 =$
$height(T_2) +1 \leq height(\text{parent }(T_2))$.

Assume correctness for $< i$ and

$height(\text{Join}(T_1,\ldots,T_i)) \leq$
$\max\{height(\text{Join}(T_1,\ldots,T_{i-1}), height(T_i)\} +1 \leq$
$\max\{height(parent(T_{i-1})), height(T_i)\} +1 \leq$
$\max\{height(parent(T_i))-1, height(parent(T_i))-1\} +1 =$
$height(parent(T_i))$.



Splitting with efficient joins

Claim:

$|\ \text{height}(T_i) - \text{height}(\text{Join}(T_1,..,T_{i-1}))\ | \leq \text{height}(T_i) + c - \text{height}(T_{i-1})$

Proof:

If $\text{height}(T_i) \geq \text{height}(\text{Join}(T_1,..,T_{i-1}))$:

$|\ \text{height}(T_i) - \text{height}(\text{Join}(T_1,..,T_{i-1}))\ | =$

$\text{height}(T_i) - \text{height}(\text{Join}(T_1,..,T_{i-1})) \leq$

$\text{height}(T_i) - \text{height}(T_{i-1})$ (as $\text{height}(\text{Join}(T_1,..,T_{i-1})) \geq \text{height}(T_{i-1})$)

If $\text{height}(T_i) < \text{height}(\text{Join}(T_1,..,T_{i-1}))$:

$|\ \text{height}(T_i) - \text{height}(\text{Join}(T_1,..,T_{i-1}))\ | =$

$\text{height}(\text{Join}(T_1,..,T_{i-1})) - \text{height}(T_i) \leq$

$\text{height}(T_{i-1}) + c - \text{height}(T_i) \leq$

c

Splitting with efficient joins

Suppose we need to join $T_1, T_2, \ldots, T_k$
where $height(T_1) \leq height(T_2) \leq \ldots \leq height(T_k)$

$$time = O\left(\sum_{i=2}^{k} |height(T_i) - height(join(T_1, \ldots, T_{i-1}))| + 1\right)$$

$$= O\left(\sum_{i=2}^{k} height(T_i) - height(T_{i-1}) + 1\right)$$

$$= O(logn)$$

Splitting AVL (by Efficient Joins)

Tighter Analysis

Recall each join really takes only $O(\text{height difference}+1)$

To generate each of T_1, T_2, we make a telescopic join series.

$$time = O\left(\sum_{i=2}^{k} |height(t_i) - height(join(t_1, \ldots, t_{i-1}))| + 1\right)$$

$$\overset{*}{=} O\left(\sum_{i=2}^{k} height(t_i) - height(t_{i-1}) + 1\right) = O(height(t_k) - height(t_1) + k) = O(logn)$$

Lemma 1: $height(t_1) \leq height(t_2) \leq \ldots \leq height(t_k)$

Lemma 2: $height(Join(t_1, \ldots, t_i)) \leq height(t_i) + c$

Back to the ADT List/Sequence:

“Tree-List” Implementation

Implementations of List/Sequence ADT (reminder)

	Circular arrays	Doubly Linked lists
Insert/Delete-First/Last	$O(1)$	$O(1)$
Insert/Delete by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$
Retrieve by index i	$O(1)$	$O(\min\{i+1, n-i\})$
Concatenate two lists (of size n)	$O(n)$	$O(1)$
Split by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$

Implementations of List/Sequence ADT

Can you do this?

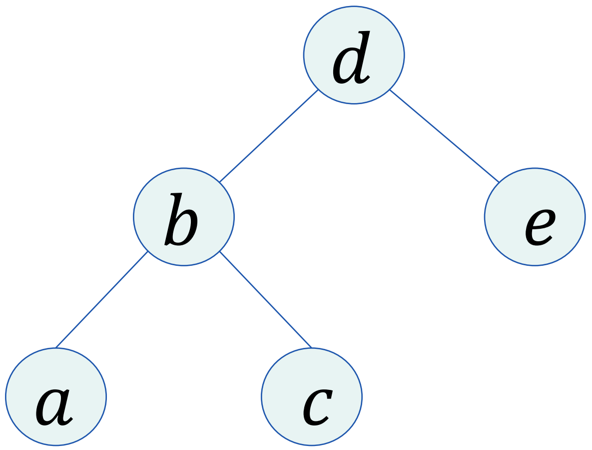
	Circular arrays	Doubly Linked lists	Tree-List
Insert/Delete-First/Last	$O(1)$	$O(1)$	$O(\log n)$
Insert/Delete by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$	$O(\log n)$
Retrieve by index i	$O(1)$	$O(\min\{i+1, n-i\})$	$O(\log n)$
Concatenate two lists (of size n)	$O(n)$	$O(1)$	$O(\log n)$
Split by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$	$O(\log n)$

The ADT List/Sequence implemented as a Rank-Tree

List/sequence ADT

Rank-tree operations

$$a\ b\ c\ d\ e$$



(List indices begin at 0)

ranks implicitly represent list indices (+1)

Tree-Nodes have no explicit keys!

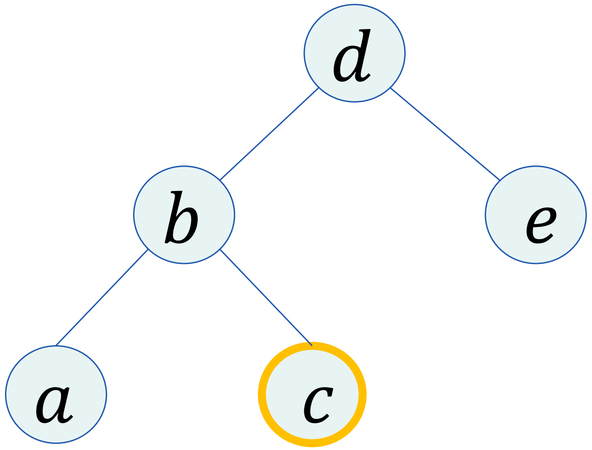
Lists as Trees: Retrieve

List/sequence ADT

Retrieve(L, i) → Tree-Select$(T, i+1)$

Rank-tree operations

$a\ b\ c\ d\ e$



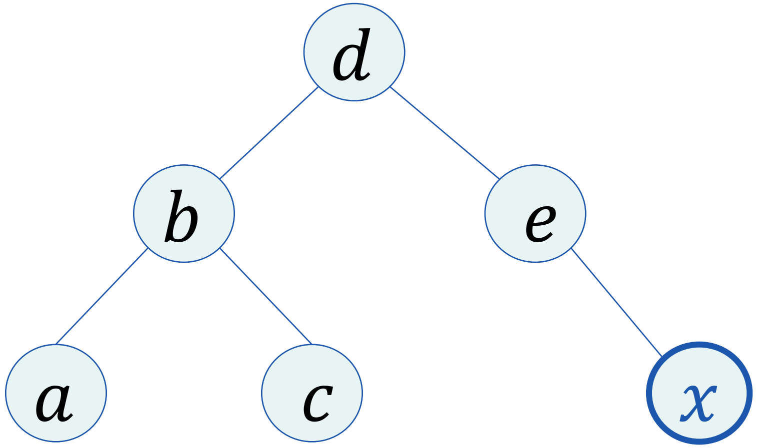
Retrieve$(L, 2)$:

Tree-Select$(T, 3)$:

Lists as Trees: Insert-Last

Insert-Last(L, x):

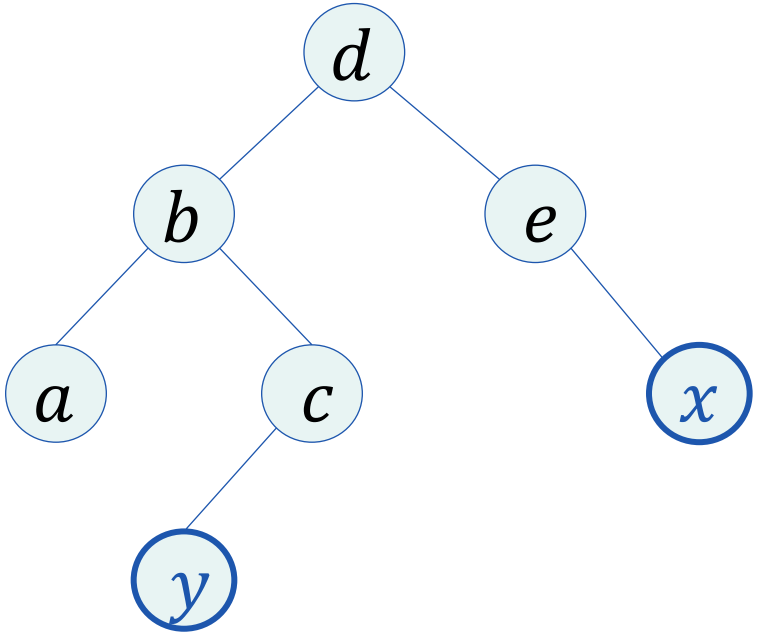
$a\ b\ c\ d\ e\ x$



Lists as Trees: Insert

Insert$(L, 2, y)$:

$a\ b\ y\ c\ d\ e\ x$



All ranks are explicitly updated!

Insert

Insert(L, i, x) (insert x in the i-th position):

if $i = n$ (Insert-Last):

make x the right child of the maximum

else ($0 \leq i < n$):

find the current node of rank $i + 1$ and add x as its predecessor:

if it has no left child, make x its left child

else find its predecessor and make x its right child

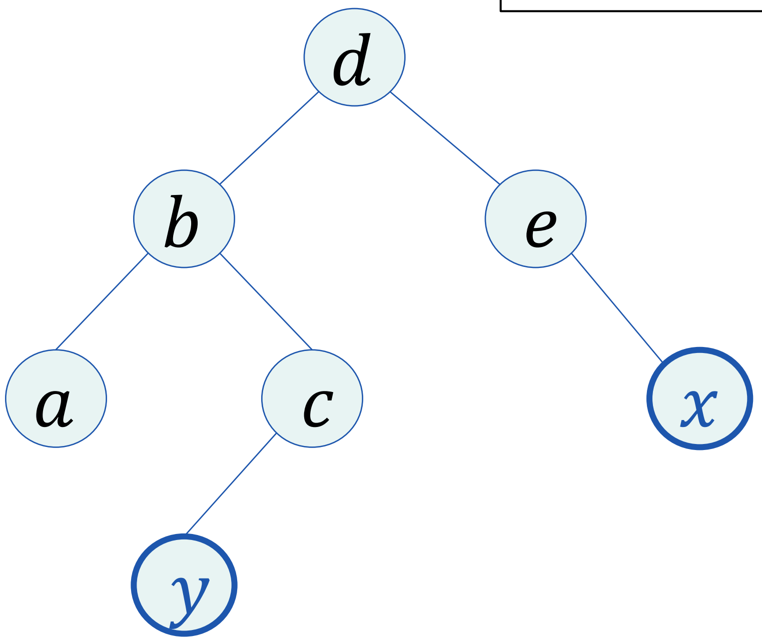
rebalance the tree as usual after insertion

Lists as Trees: Delete

Delete(L, 2):

$a\ b\ \cancel{y}\ c\ d\ e\ x$

Delete(L,i)
  $z \leftarrow$ Tree-Select(T, i)
  delete z as usual (given a pointer to it)



Simply delete a node from tree
All ranks are explicitly updated!

Tree-Lists: Summary

- Implement list L as a rank-tree:
  i-th item is the node of rank $i + 1$

- Tree-Nodes have no explicit keys
  (Implicitly maintained ranks play the role of keys)

- Operations (including the new Insert)
  do not use keys!

- Concat / split achieved by Tree join/split

Implementations of List/Sequence ADT

	Circular arrays	Doubly Linked lists	Tree-List
Insert/Delete-First/Last	$O(1)$	$O(1)$	$O(\log n)$
Insert/Delete by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$	$O(\log n)$
Retrieve by index i	$O(1)$	$O(\min\{i+1, n-i\})$	$O(\log(i+1))$
Concatenate two lists (of size n)	$O(n)$	$O(1)$	$O(\log n)$
Split by index i	$O(\min\{i+1, n-i\})$	$O(\min\{i+1, n-i\})$	$O(\log n)$

Can you do this?

Improving Retrieve

- Yes, using a finger to the minimum!

- Does this make Insert/Delete also work in $O(\log(i+1))$?

- Can be achieved amortized, by lazy update of node information (maybe in the recitations)