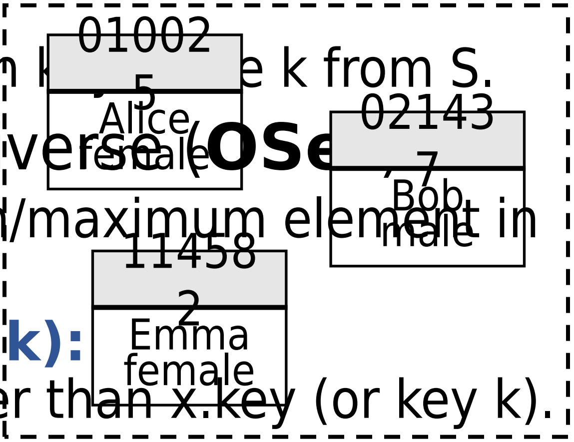
Search Trees

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

The **Set** Abstract Data Type (ADT)

- The **Set** ADT is used to represent a *set* of elements with (usually distinct) *key* values.
  - Each element has a key field and a data field.
- Operations the **Set** ADT should support:
  - **Search(S,k):** Find an element in S with key value k.
  - **Insert(S,x):** Add x to S. (What if element with same key exists?)
  - **Remove(S,x):** Remove element x from S, assuming x is in S.
  - **Remove(S,k):** Remove element with k e k from S.
- If elements are from an ordered universe (**OSe**
  - **Min(S) and Max(S):** Find minimum/maximum element in S.
  - **Successor(S,x) or Successor(S,k):** Find smallest element in S that is larger than x.key (or key k).
  - **Predecessor(S,x) or Predecessor(S,k):** Find largest element in S that is smaller than x.key (or key k).



Efficient implementation of **OSet**

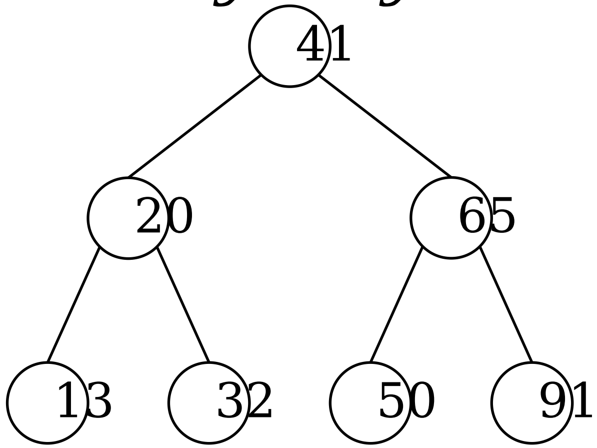
	Search(S,k)	Insert(S,x)	Remove(S,x)
SimpleArray			
SimpleLinkedList			
SortedArray			
SortedLinkedList			
BinaryHeap			

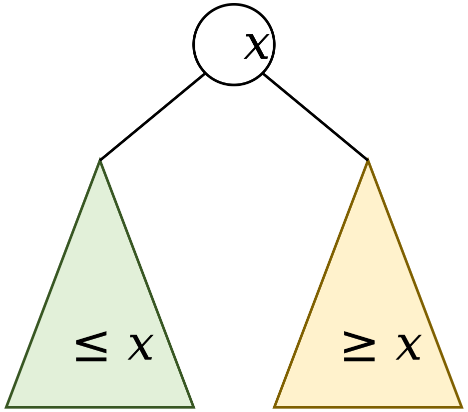
Data structure implementing all these operations efficiently?

Efficient means within $O(\log n)$ time.

Binary Search Tree (BST)

- A **binary search tree** (**BST**) is a binary tree in which each node stores an element, and satisfies the *binary-search-tree property* (*BST property*): for *every* node x in the tree, if y is in the left subtree of x, then $y.key \le x.key$; if y is in the right subtree of x, then $y.key \ge x.key$.

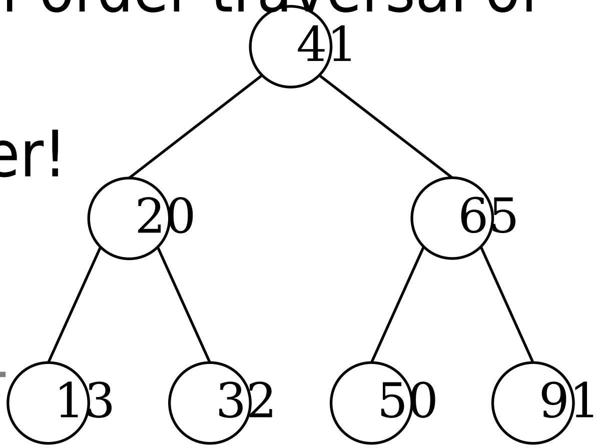




Binary Search Tree (BST)

- A **binary search tree** (**BST**) is a binary tree in which each node stores an element, and satisfies the *binary-search-tree property* (*BST property*): for *every* node x in the tree, if y is in the left subtree of x, then $y.key \leq x.key$; if y is in the right subtree of x, then $y.key \geq x.key$.
- **Q:** Given a BST T, let S be the set of elements stored in T, what is the sequence of the in-order traversal of T?
- **A:** Elements of S in ascending order!

In-order traversal:
13, 20, 32, 41, 50, 65, 91



Search in BST

- Given a BST root x and ke
  key k?
  - If $x.key = k$ then return x
  - If $x.key > k$ then recurse
  - If $x.key < k$ then recurse
- This is tail recursion, and w
  version!

BSTSearch(x,k):

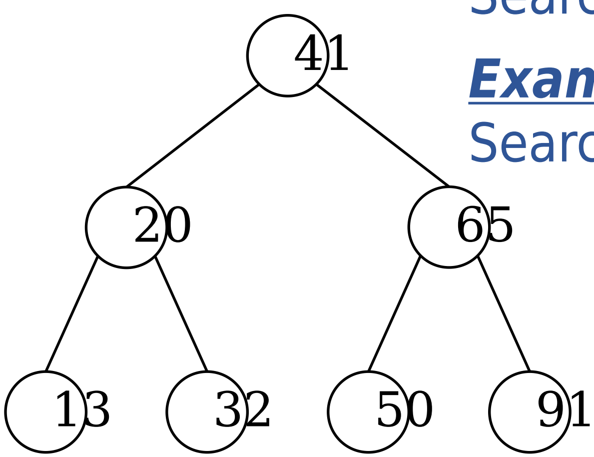
```
if (x==NULL or x.key==k)
  return x
else if (x.key>k)
  return BSTSearch(x.left,k)
else
  return BSTSearch(x.right,k)
```

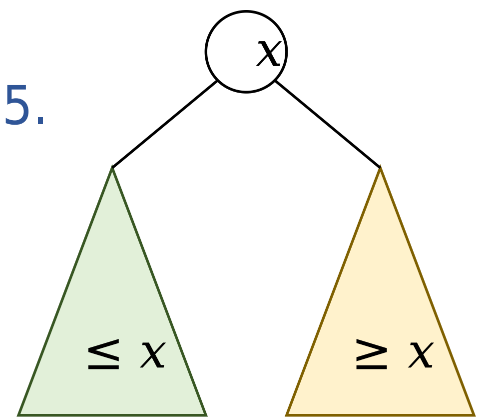
BSTSearchIter(x,k):

```
while (x!=NULL and x.key!=k)
  if (x.key>k)
    x = x.left
  else
    x = x.right
return x
```

Example One:
Search for element with key 50.

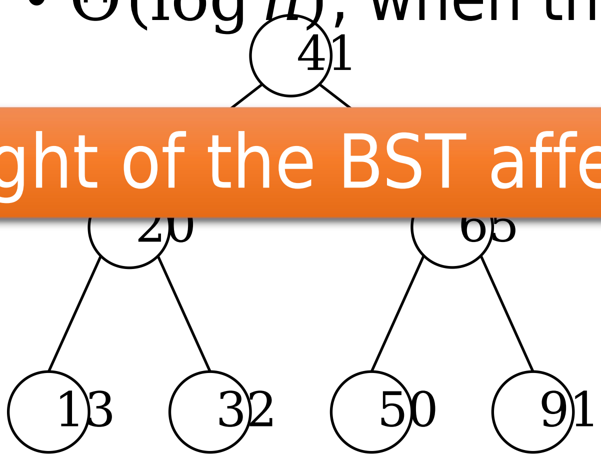
Example Two:
Search for element with key 35.

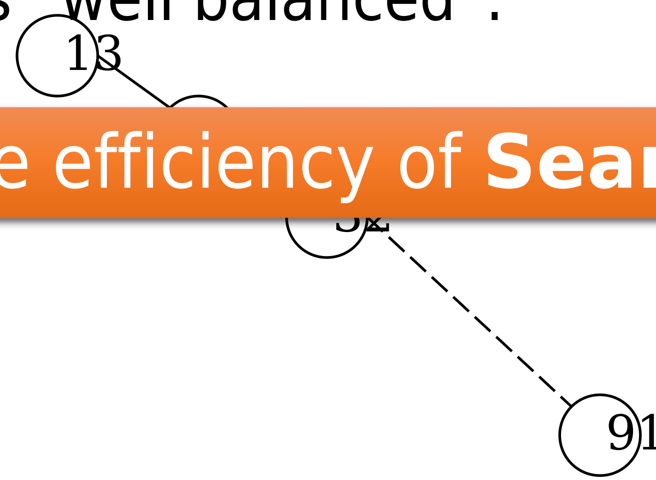




Complexity of **Search** in BST

- **Q:** Worst-case time complexity of **Search** operation?
- **A:** $\Theta(h)$ where h is the height of the BST.
  - How large can h be in an n-node BST?
  - $\Theta(n)$, when the BST is like a "path".
  - How small can h be in an n-node BST?
  - $\Theta(\log n)$, when the BST is "well balanced".

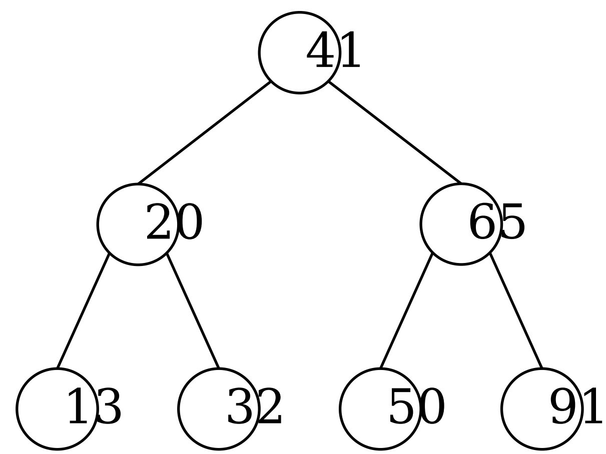




Height of the BST affects the efficiency of **Search**

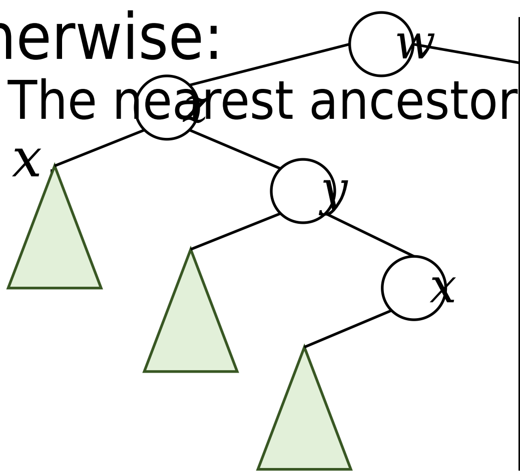
Min and **Max** in BST

- How to find a minimum element in a BST?
  - Keep going left until a node without left child.
- How to find a maximum element in a BST?
  - Keep going right until a node without right child.
- Time complexity of **Min** and **Max** operation?
  - $\Theta(h)$ in the worst-case where h is height.



Successor in BST

- **BSTSuccessor(x):** Find the smallest element in the BST with key value larger than x.key.
- In-order traversal of BST lists the elements in sorted order. Where in the tree does the element following x reside?
- If the right subtree rooted at x is non-empty:
  - The minimum element in BST rooted at $x.right$ is what we want.
- Otherwise:
  - The nearest ancestor



BSTSuccessor(x):

```
if (x.right!=NULL)
  return BSTMin(x.right)
y = x.parent
while (y!=NULL and y.right==x)
  x = y
  y = y.parent
return y
```

Successor in BST

- Time complexity of **BSTSuccessor**?
  - $\Theta(h)$ in the worst-case where h is the height.
- **BSTPredecessor** can be designed and analyzed similarly.

- So far we've seen operations that do not change the BST.
  - **Search**, **Min/Ma**
- How about operatio
  - **Insert** and **Remo**

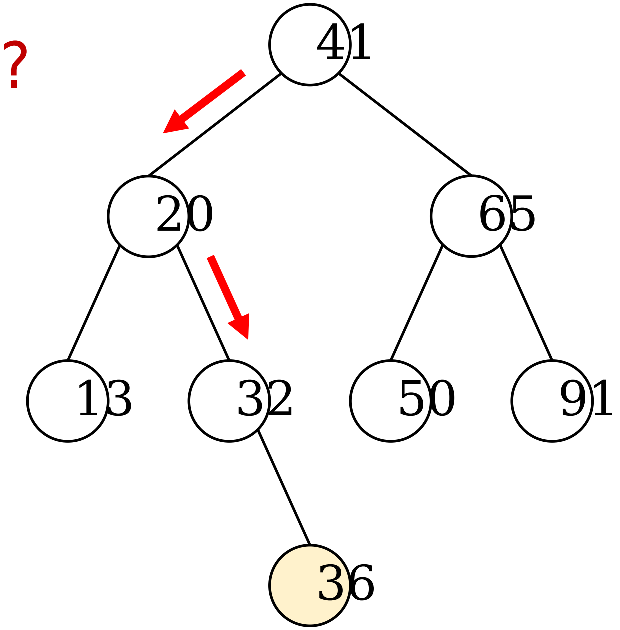
BSTSuccessor(x):

```
if (x.right!=NULL)
  return BSTMin(x.right)
y = x.parent
while (y!=NULL and y.right==x)
  x = y
  y = y.parent
return y
```

Insert in BST

- **BSTInsert(T,z):** Add z to BST T. Notice, insertion should not break the BST property.
- Just like doing a search in T with key $z.key$. This search will fail and end at a leaf y. Insert z as left or right child of y.
- Why above procedure is correct?

Example:
Insert element with key 36.

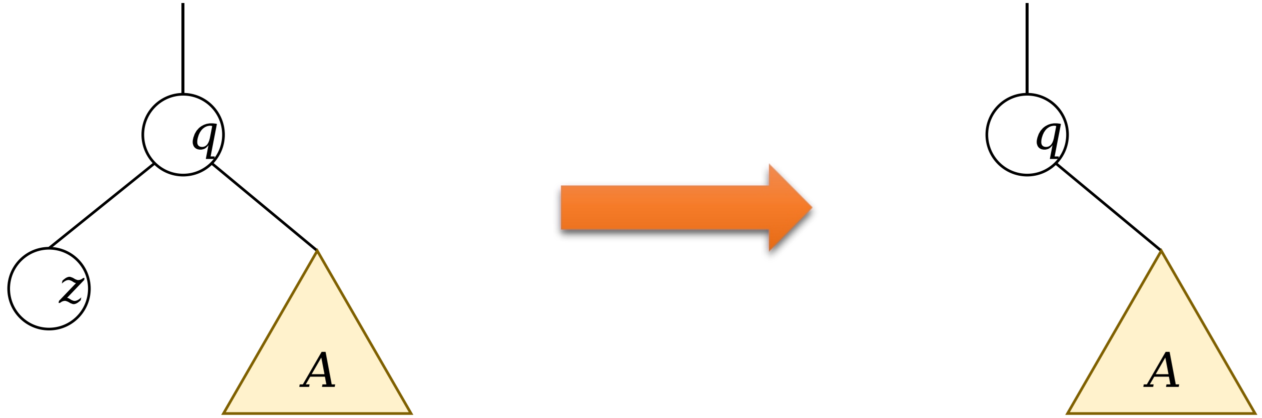


Insert in BST

- **BSTInsert(T,z):** Add z to BST T. Notice, insertion should not break the BST property.
- Just like doing a search in T with key $z.key$. This search will fail and end at a leaf y. Insert z as left or right child of y.
- **Q:** Time complexity of the **Insert** operation?
- **A:** $\Theta(h)$ in the worst-case where h is the height of T.

Remove in BST

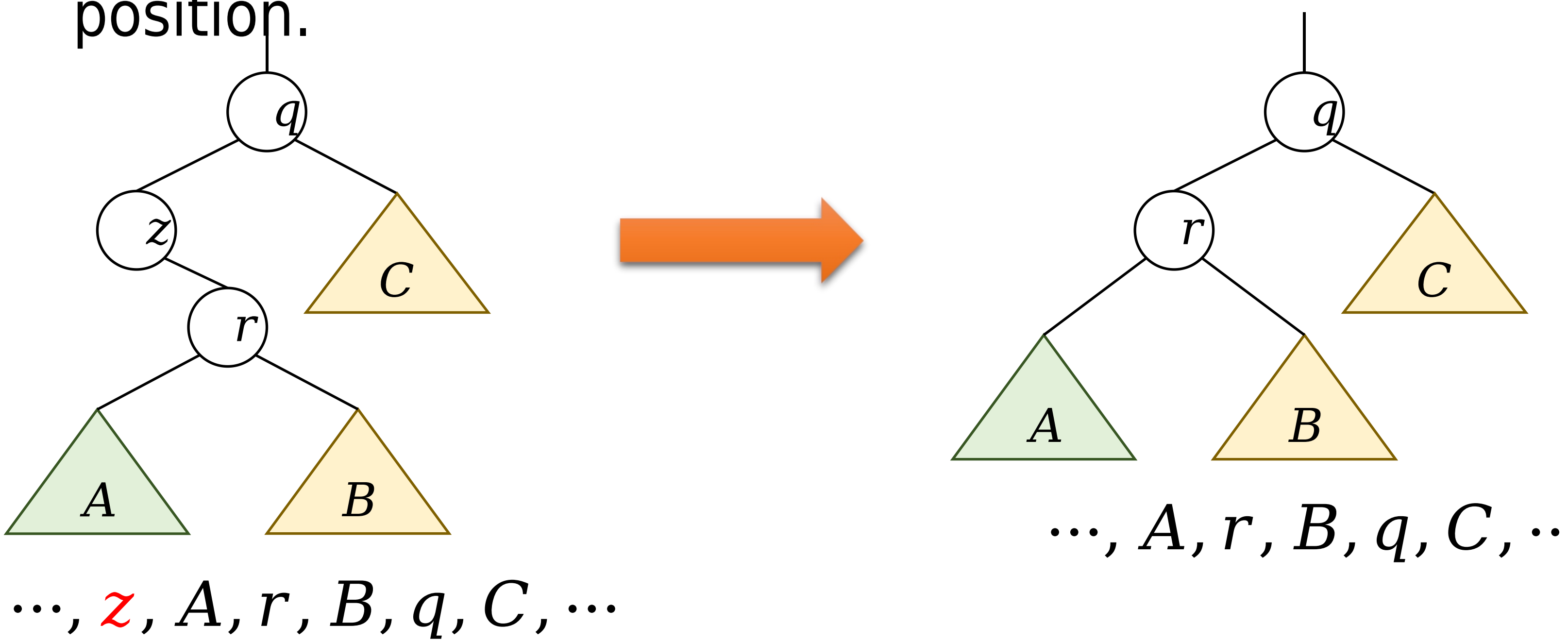
- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 1:** z has no child.
- Easy, simply remove z from the BST tree.



$\cdots, {\color{red}z}, q, A, \cdots$ $\qquad$ $\cdots, q, A, \cdots$

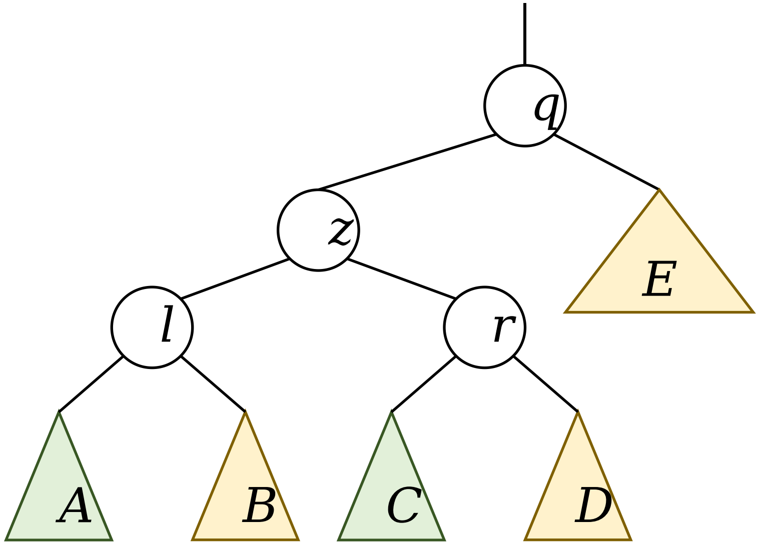
Remove in BST

- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 2:** z has one single child.
- Elevate subtree rooted at z's single child to take z's position.



Remove in BST

- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 3:** z has two children.
  - **Case 3a:** $z.right.left = NULL$
  - **Case 3b:** $z.right.left \neq NULL$



$\cdots, A, l, B, z, C, r, D, q, E, \cdots$

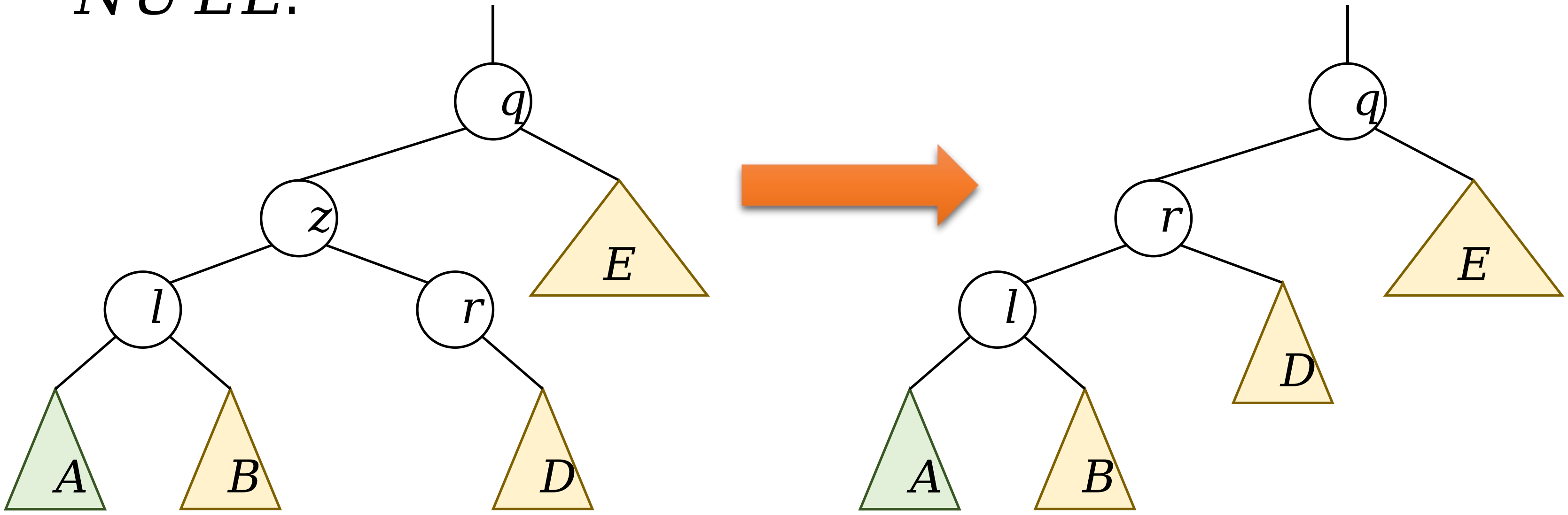
Replace node z with min value node in subtree rooted at $z.right$. ($z.right$ guaranteed to be non-empty)

Replace node z with BSTSuccessor(z)

BSTSuccessor(z) can be:
- r if $r.left = NULL$
- BSTMin(r.left) if $r.left \neq NULL$

Remove in BST

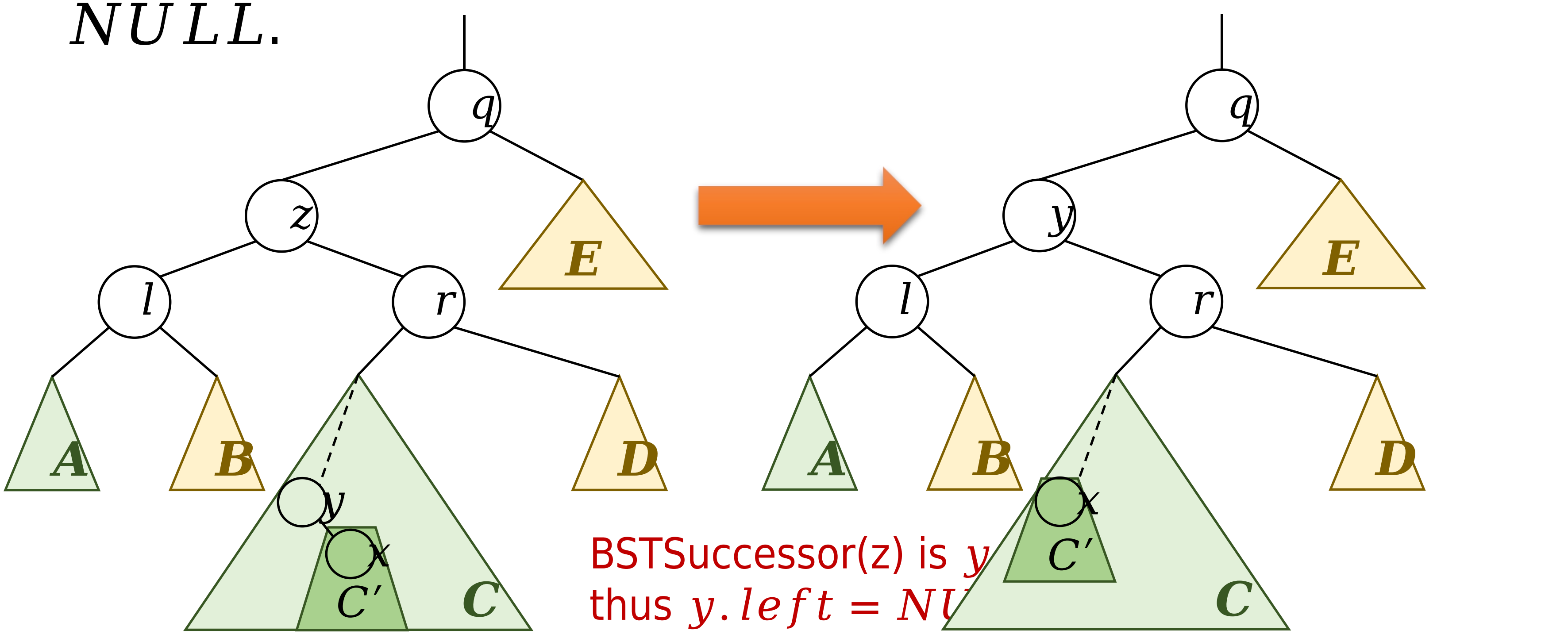
- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 3a:** z has two children and $z.right.left = NULL$.



$\cdots, A, l, B, z, r, D, q, E, \cdots$ $\quad$ $\cdots, A, l, B, r, D, q, E, \cdots$

Remove in BST

- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 3b:** z has two children and $z.right.left \neq NULL$.

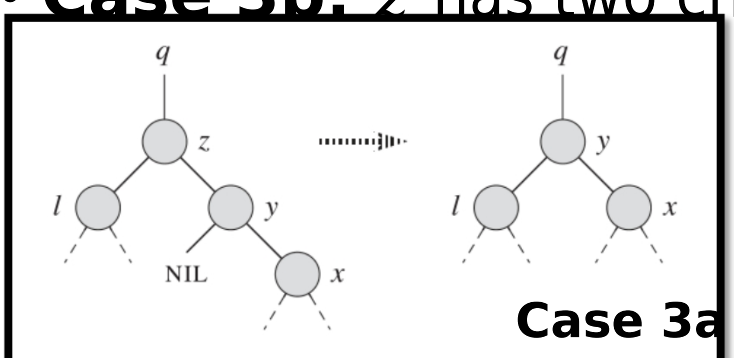


$\cdots, A, l, B, \textcolor{red}{z}, y, C', C \setminus C', r, D, q, E, \cdots$ $\cdots, A, l, B, y, C', C \setminus C', r, D, q, E, \cdots$

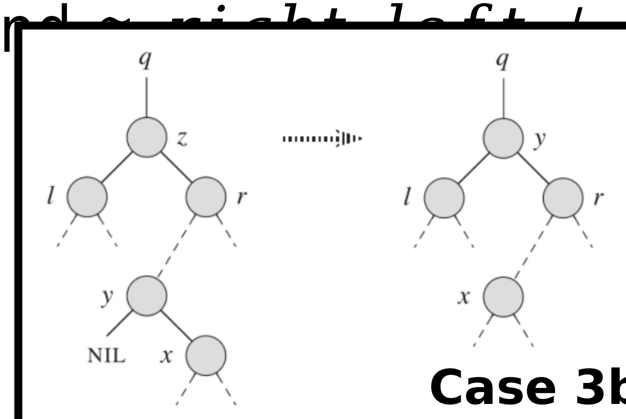
Remove in BST

Worst-case time complexity of **Remove** operation is $\Theta(h)$.

- **BSTRemove(T,z):** Remove element z from T. Notice, removal should not break the BST property.
- **Case 1:** z has no child. $\Theta(1)$
  - Easy, simply remove z from the BST tree.
- **Case 2:** z has a single child. $\Theta(1)$
  - Elevate subtree rooted at z's single child to take z's position.
- **Case 3a:** z has two children and $z.right.left = NULL$. $\Theta(1)$
- **Case 3b:** z has two children and $z.right.left \neq$ $O(h)$



Case 3a



Case 3b

Efficient implementation of **OSet**

	Search(S,k)	**Insert(S,x)**	**Remove(S,x)**
SimpleArray	$O(n)$	$O(1)$	$O(n)$
SimpleLinkedList	$O(n)$	$O(1)$	$O(1)$
SortedArray	$O(\log n)$	$O(n)$	$O(n)$
SortedLinkedList	$O(n)$	$O(n)$	$O(1)$
BinaryHeap	$O(n)$	$O(\log n)$	$O(\log n)$
BinarySearchTree	$O(h)$	$O(h)$	$O(h)$

BST also supports other operations of **OSet**, in $O(h)$ time.

But height of a n-node BST varies between $\Theta(\log n)$ and $\Theta($

Height of BST

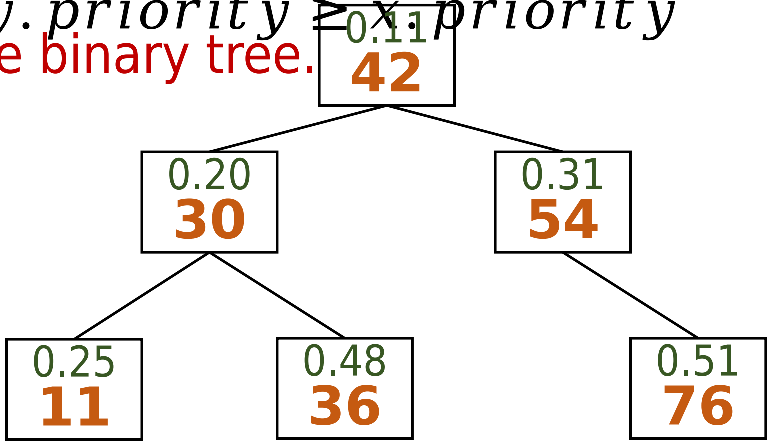
- Consider a sequence of **Insert** operations given by an adversary, the resulting BST can have height $\Theta(n)$.
  - E.g., insert the elements in increasing order.
- What is the expected height of a **randomly** built BST?
  - Build the BST from an empty BST with n **Insert** operations.
  - Each of the $n!$ insertion orders is equally likely to happen.
- The expected height of a randomly built BST is $O(\log n)$.

A randomized BST structure

Treap (Binary-Search-Tree + Heap)

- A **Treap** is a *binary tree* in which each node has a key value, and a priority value.
- The key values must satisfy the BST-property:
  - For each node y in left sub-tree of x: $y.key \le x.key$
  - For each node y in right sub-tree of x: $y.key \ge x.key$
- The priority values must satisfy the MinHeap-property:
  - For each descendent y of x: $y.priority \ge x.priority$

A Treap is not necessarily a complete binary tree.
(Thus it is not a BinaryHeap.)



A randomized BST structure

Treap

- Given a set of n nodes with *distinct* key values and *distinct* priority values, a *unique* **Treap** is determined.
- Proof by induction on n:
  - **[Basis]:** The claim clearly holds when $n = 0$.
  - **[Hypothesis]:** The claim holds when $n \leq n' - 1$.
  - **[Inductive Step]:**
  - Given a set of n' nodes, let r be the node with min priority. By MinHeap-property, r has to be the root of the final Treap.
  - Let L be set of nodes with key values less than $r.key$, and R be set of nodes with key values larger than $r.key$.
  - By BST-property, in the final Treap, nodes in L must in left sub-tree of r, and nodes in R must in right sub-tree of r.
  - By induction hypothesis, nodes in L lead to a unique Treap, and nodes in R lead to a unique Treap.

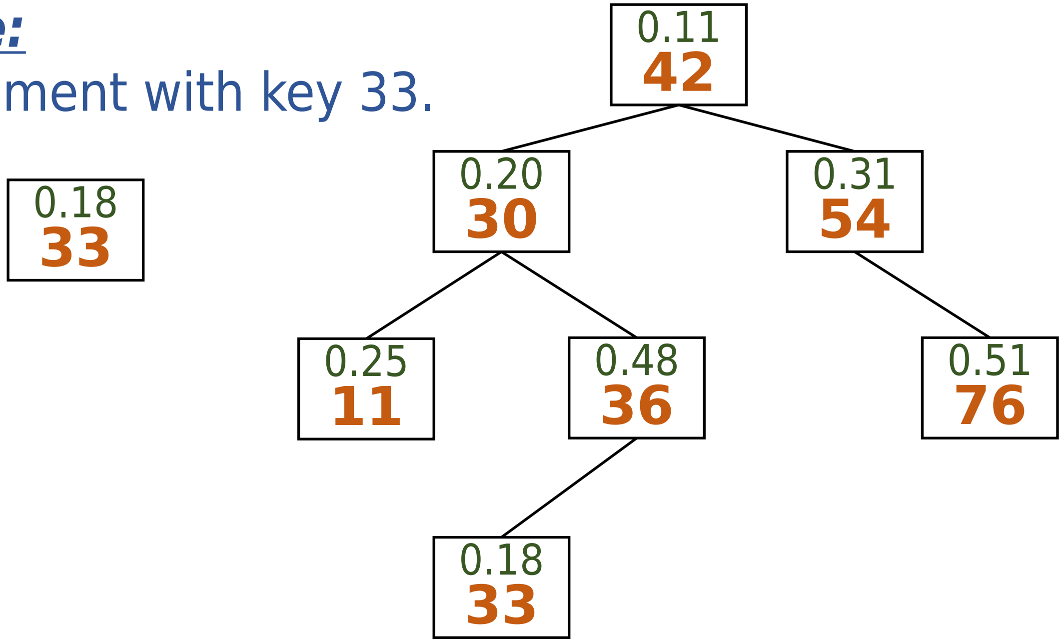
Treap

- **Q:** How do we build a Treap?
- **A:** Starting from an empty Treap, whenever we are given a node x that needs to be added, we assign a random priority for node x, and insert the node into the Treap.
- Alternative view of an n-node Treap: a BST built with n insertions, in the order of increasing priorities. (**Why?**)
  (Only need to worry about BST property if build a Treap in this order.)
- **A Treap is like a randomly built BST, regardless of the order of the insert operations!** (Since we use random priorities!)
- A Treap has height $O(\log n)$ in expectation. Therefore, all **OSet** operations are efficient in

Insert in Treap

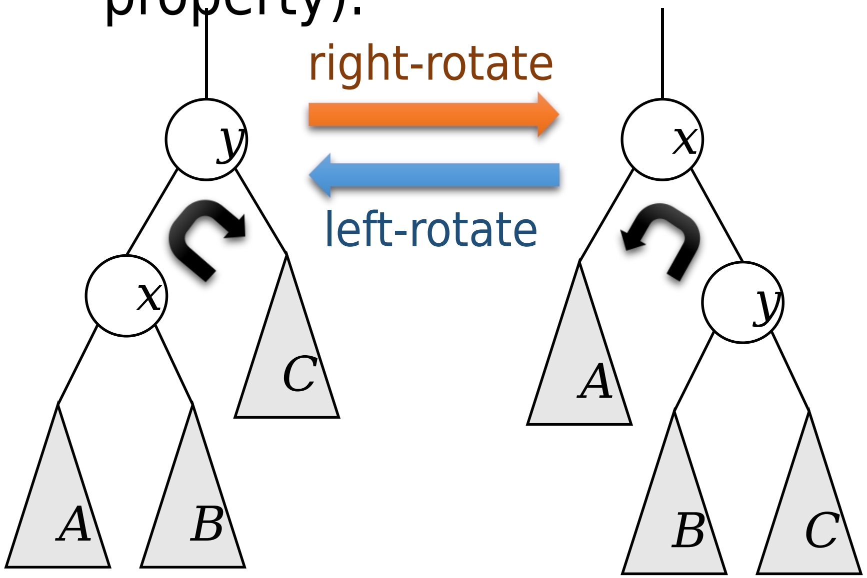
- **Step 1:** Assign a random priority to the node to be added.
- **Step 2:** Insert the node following BST-property.
- **Step 3:** Fix MinHeap-property (without violating BST-property).

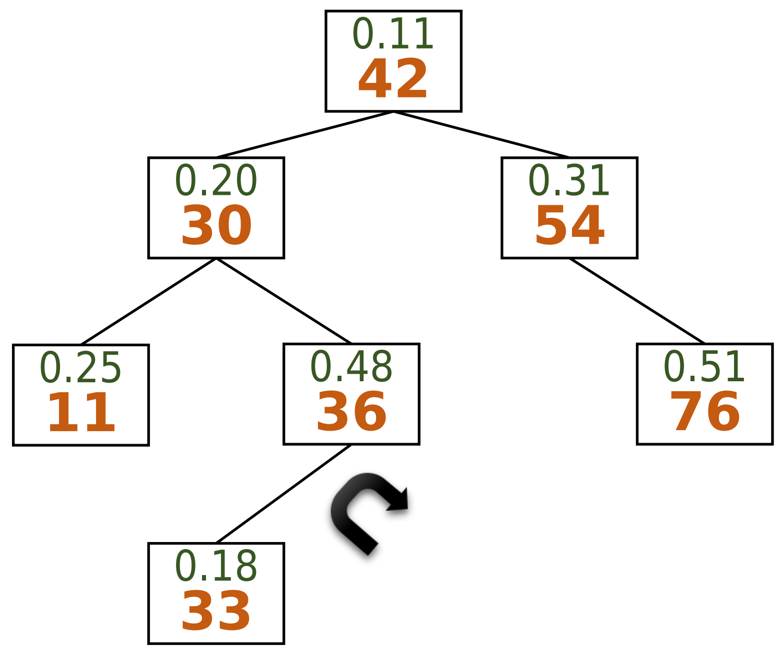
Example:
Insert element with key 33.



Insert in Treap

- **Step 1:** Assign a random priority to the node to be added.
- **Step 2:** Insert the node following BST-property.
- **Step 3:** Fix MinHeap-property (without violating BST-property).

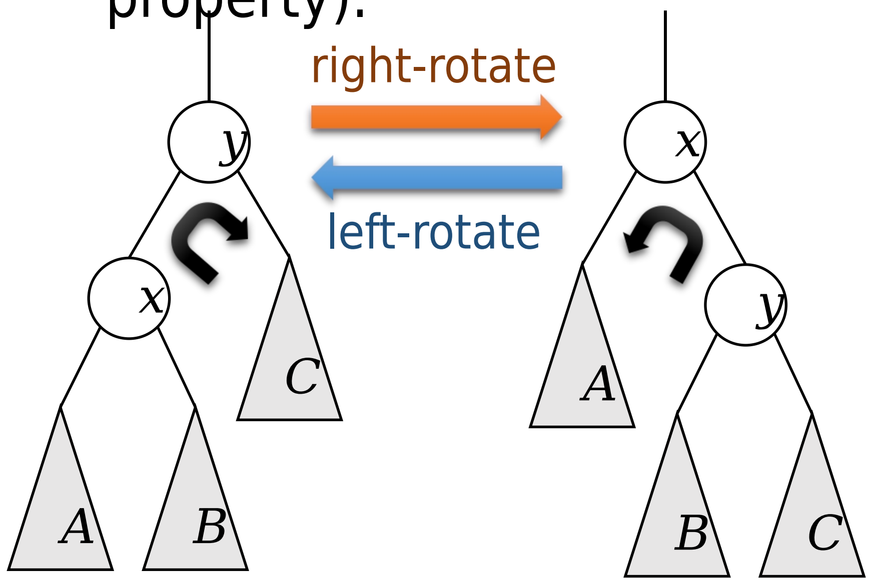




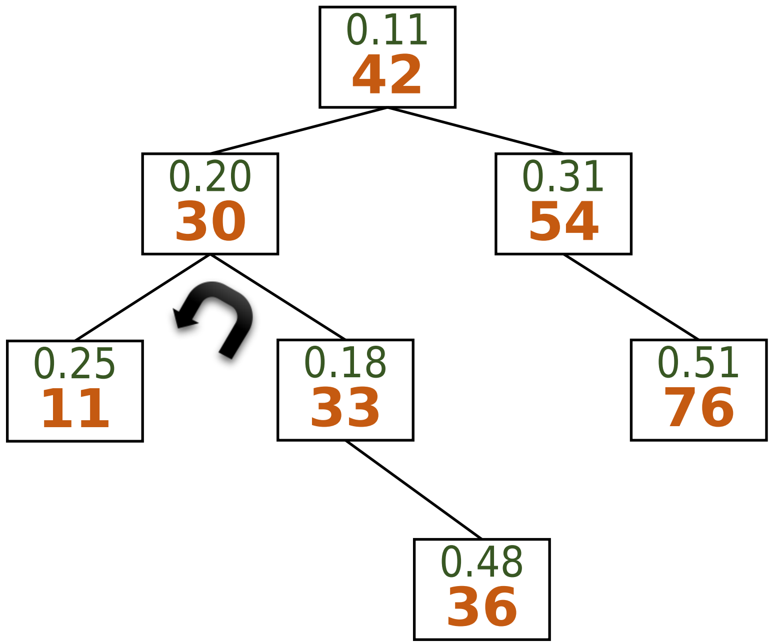
Rotation changes level of x and y, but preserves BST property.

Insert in Treap

- **Step 1:** Assign a random priority to the node to be added.
- **Step 2:** Insert the node following BST-property.
- **Step 3:** Fix MinHeap-property (without violating BST-property).

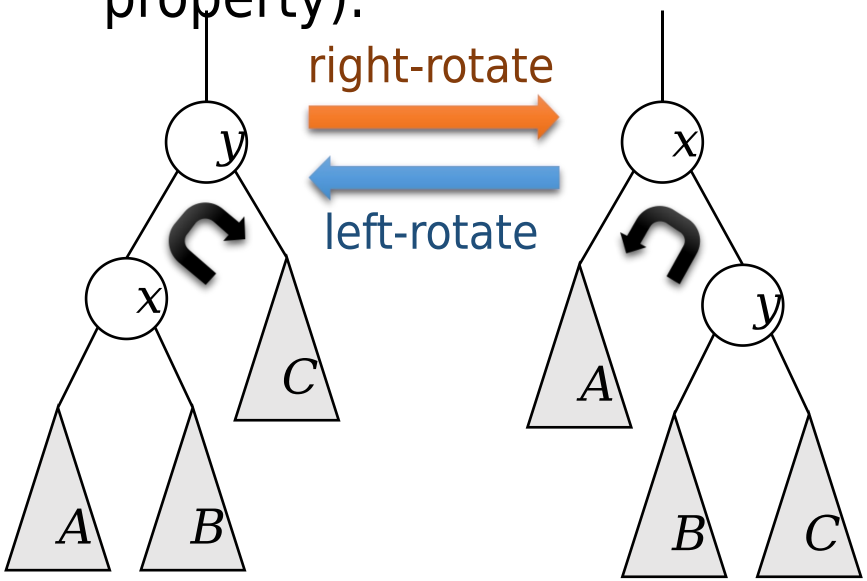


Rotation changes level of x and y, but preserves BST property.

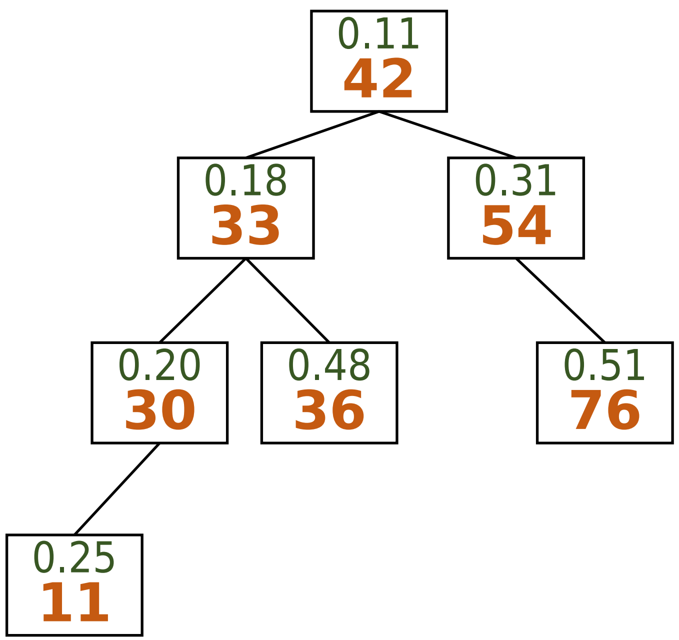


Insert in Treap

- **Step 1:** Assign a random priority to the node to be added.
- **Step 2:** Insert the node following BST-property.
- **Step 3:** Fix MinHeap-property (without violating BST-property).

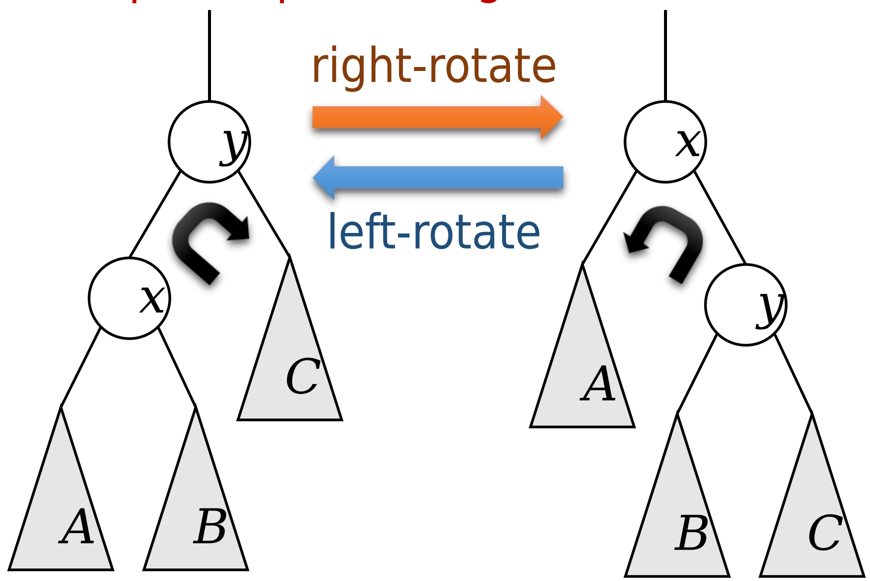


Rotation changes level of x and y,
but preserves BST property.



Insert in Treap

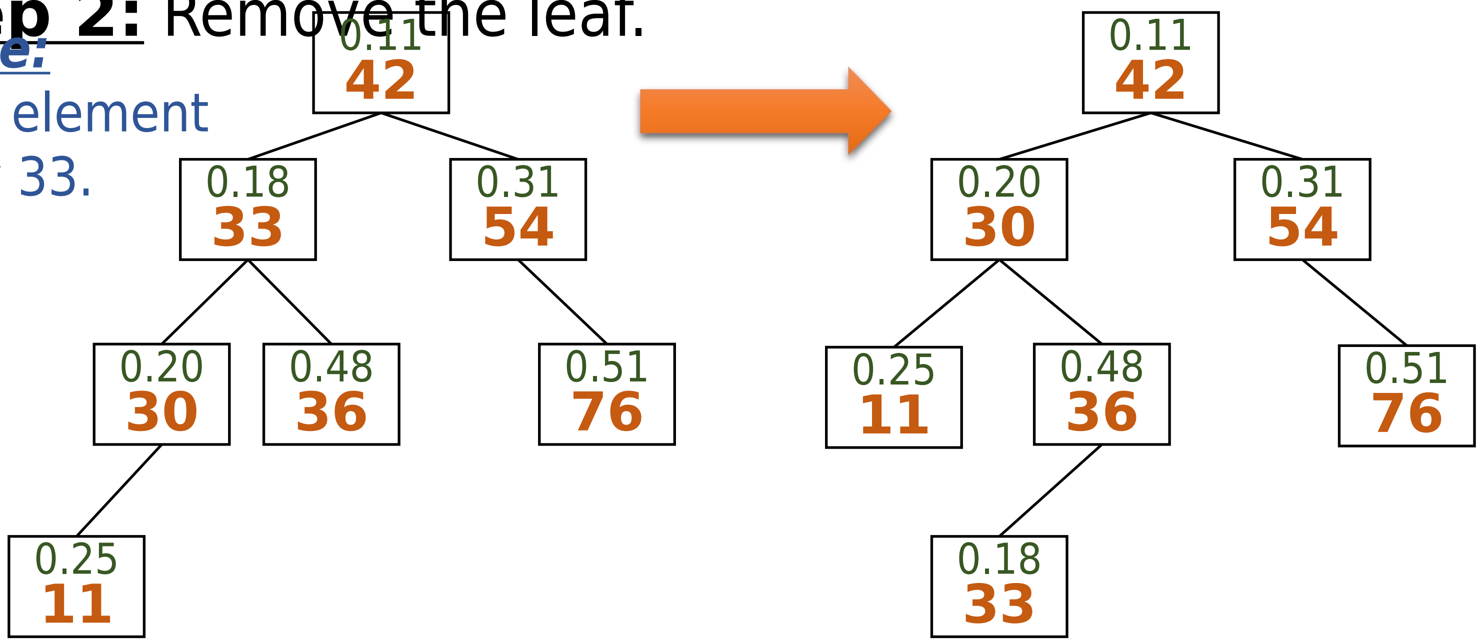
- **Step 1:** Assign a random priority to the node to be added.
- **Step 2:** Insert the node following BST-property.
- **Step 3:** Fix MinHeap-property (without violating BST-property).
    - Use rotations to push-up violating nodes until MinHeap-property restored.



Remove in Treap

- **Q:** Given a pointer to a node, how to remove it?
- **A:** Just invert the process of insertion!
- **Step 1:** Use rotations to push-down the node till it is a leaf.
- **Step 2:** Remove the leaf.

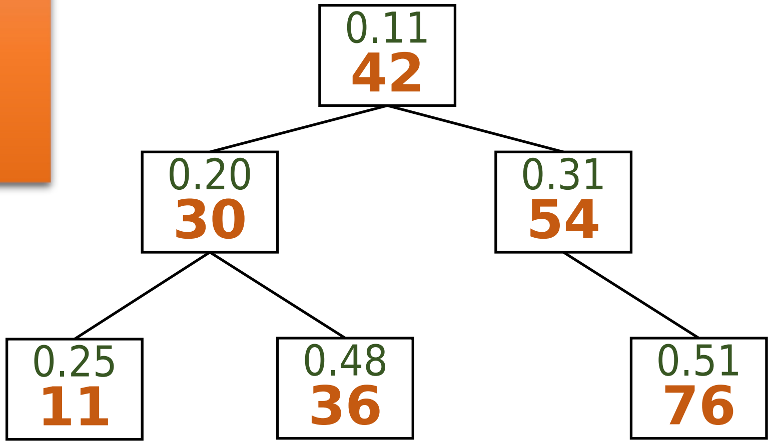
Example:
Remove element with key 33.



Treap

- A probabilistic data structure.
- Like a randomly built BST.
  (Expected height is $O(\log n)$, even for adversarial operation sequence.)
- Support **OSet** operations in $O(\log n)$ time, in expectation

Design a data structure supporting OSet operations in $O(\log n)$ time, even in worst-case?



Reading

- [CLRS] Ch.12
- [Morin] Ch.7 (7.2)