Lecture 8: Advanced Binary Trees

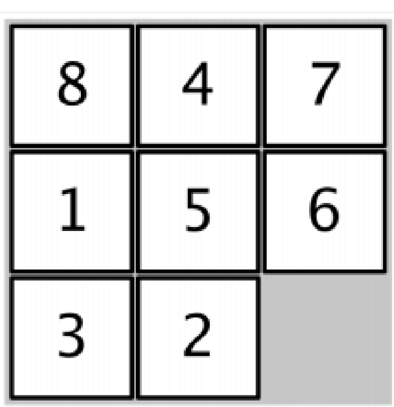
Bo Tang @ SUSTech, Fall 2023

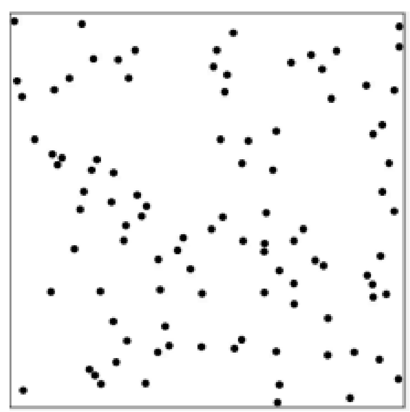
Our Roadmap

- Priority Queue (binary heap)
  - Min-heap insert / delete-min
- Binary Heaps in Dynamic Arrays
  - O(n) time to build min-heap
- Binary Search Tree (BST)
  - BST operators
  - Balanced BST (AVL-tree)

Priority Queue

- A priority queue stores a set S of n integers and supports the following operations:
  - *Insert(e)*: adds a new integer to S
  - *Delete-min:* removes the smallest integer in S, and returns it.
- Priority Queue applications:
  - Artificial intelligence (A* algorithm)
  - Operating systems (load balancing)
  - Graph searching (Shortest path algorithms)





Priority Queue Example

- Suppose that the following integers are inserted into an initially empty priority queue
  - S={93, 39, 1, 26, 8, 23, 79, 54}
  - Perform Delete-min, the operation returns 1, and S ={93, 39, 26, 8, 23, 79, 54}
  - Perform Delete-min, the operation returns 8, and S ={93, 39, 26, 23, 79, 54}
  - Perform …..
- Unlike an ordinary queue (FIFO), a priority queue guarantees that the elements always leave in ascending order (or descending order with *Delete-max*), regardless of the order by which they are inserted.

Priority Queue Implementation

- We will implement a priority queue using a data structure called the "binary heap" to achieve the following guarantees:
  - O(n) space consumption
  - O(log n) insertion time
  - O(log n) delete-min time

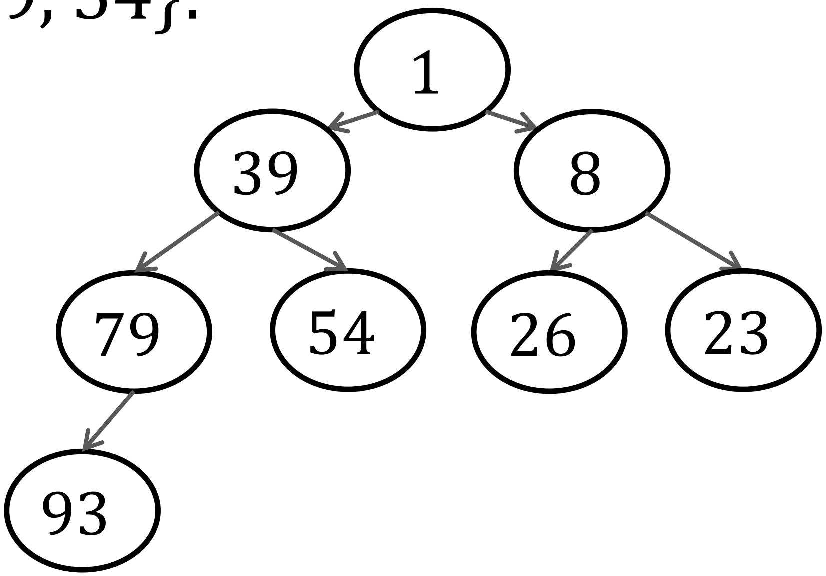
- The binary heap data structure is an array object that we can view as a complete binary tree.
  - Level 0 to h-1 are full
  - Leaf nodes in level h are "as far left as possible"

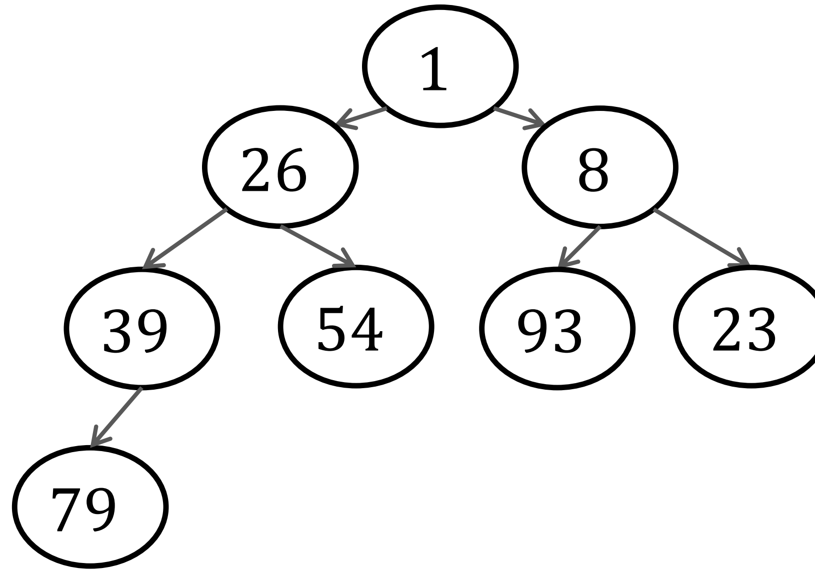
Binary Heap

- Let S be a set of n integers. A binary heap on S is a binary tree T satisfying:
  - (1) T is complete
  - (2) Every node u in T corresponds to a distinct integer in S, the integer is called the key of u (and is stored at u)
  - (3) If u is an internal node, the key of u is smaller than those of its child nodes
- Note that:
  - Condition 2 implies that T has n nodes
  - Condition 3 implies that the key of u is the smallest in the subtree of u

Binary Heap Example

- Two possible binary heaps on S = {93, 39, 1, 26, 8, 23, 79, 54}:





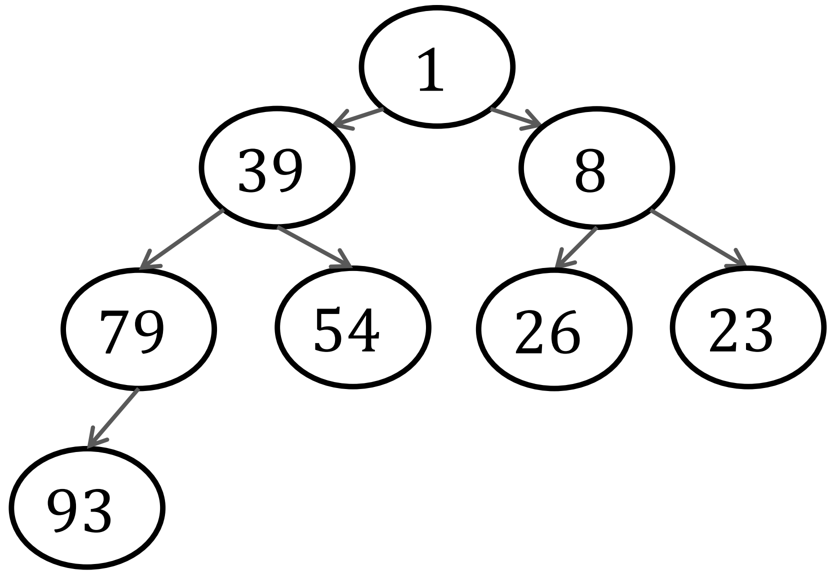
- The binary heaps of a set S is not unique.
- The smallest integer of S must be the key of the root.

Binary Heap Insertion

- We perform insert(e) on a binary heap T as follows:
  - Step 1: Create a leaf node z with key e, while ensuring that T is a complete binary tree, it means there is only one place where z could be added.
  - Step 2: Set u $\leftarrow$ z
  - Step 3: If u is the root, return.
  - Step 4: If the key of u > the key of its parent p, return
  - Step 5: Otherwise, swap the keys of u and p. Set u $\leftarrow$ p, and repeat from Step 3.

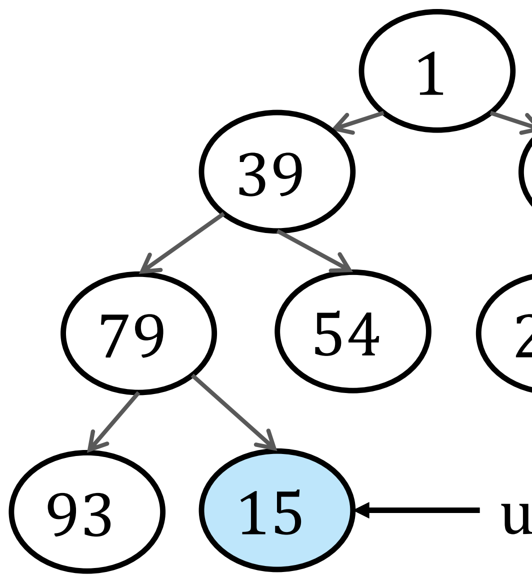
Binary Heap Insertion

- Suppose we want to insert 15 into the binary heap below:



Binary Heap Insertion

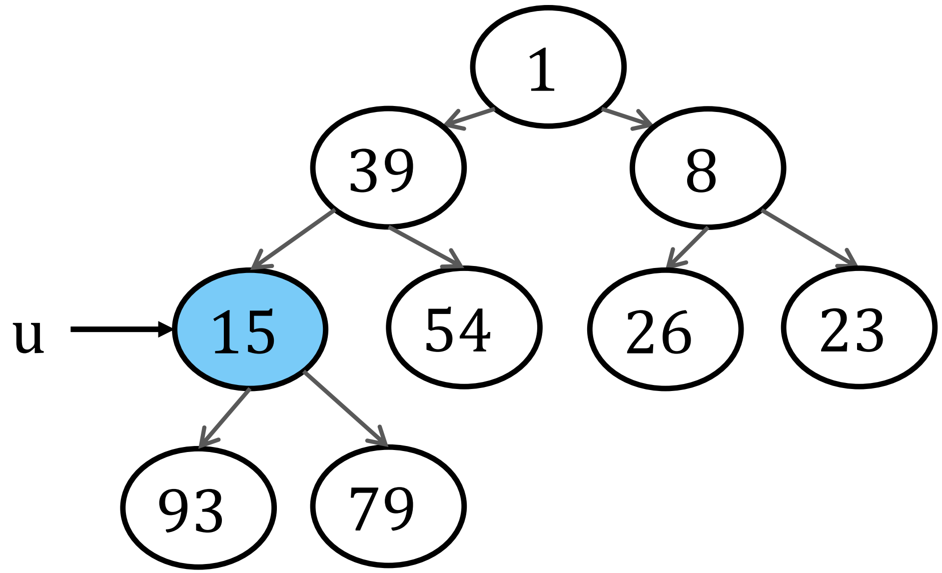
- First add 15 as a new leaf, making sure that we still have a complete binary tree.



- Step 3 is not true, go to Step 4,
- Step 4 is not true, go to step 5.

Binary Heap Insertion

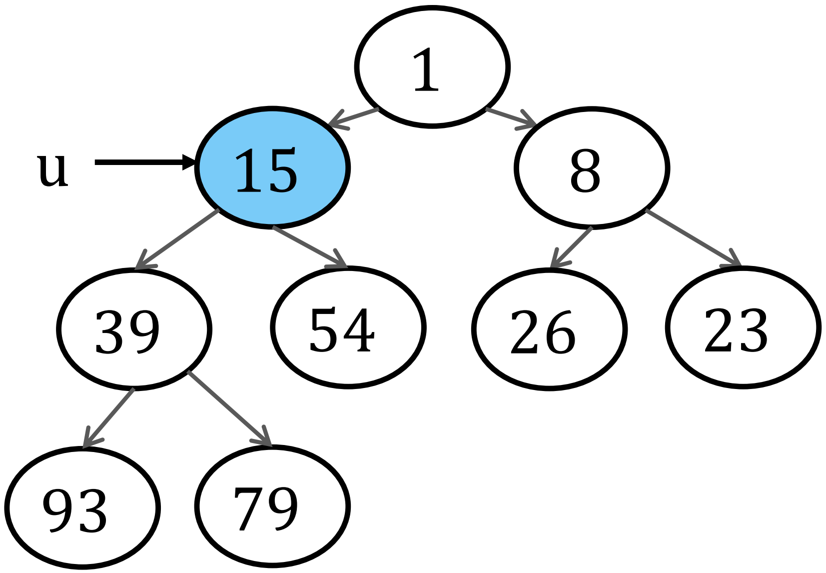
- First add 15 as a new leaf, making sure that we still have a complete binary tree.



- Swap the keys of u and its parent p
- Set u ← p, go back to Step 3)
- Step 3 and Step 4 are not true, go to Step 5.

Binary Heap Insertion

- First add 15 as a new leaf, making sure that we still have a complete binary tree.



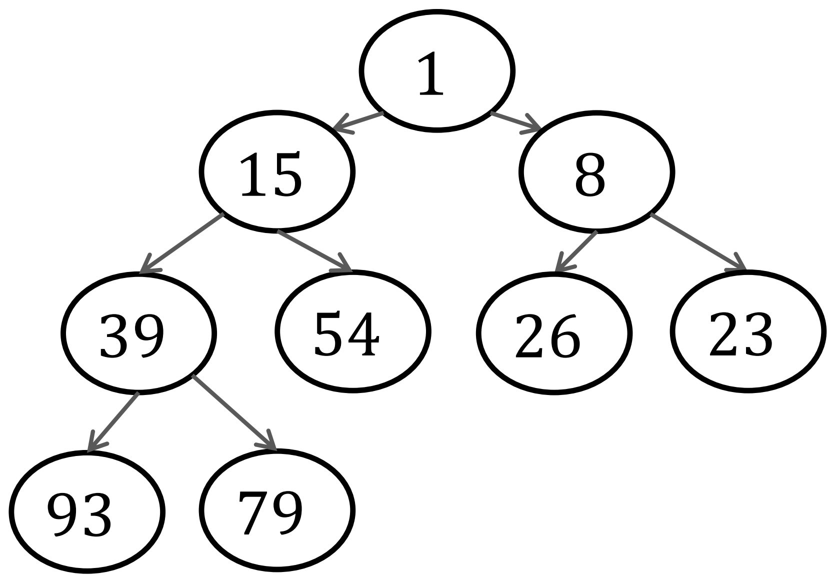
- Swap the keys of u and p
- Set u ← p, go back to Step 3)
- Step 3 is not true, Step 4 is true, return. Insertion complete.

Binary Heap Delete-min

- We perform delete-min on a binary heap T as follows:
  - Step 1: Report the key of the root
  - Step 2: Identify the rightmost leaf z at the bottom level of T
  - Step 3: Delete z, and store the key of z at the root
  - Step 4: Set u $\leftarrow$ the root
  - Step 5: If u is leaf, return
  - Step 6: If the key of u < the keys of the children of u, return
  - Step 7: Otherwise, let v be the child of u with a smaller key
    Swap the keys of u and v. Set u $\leftarrow$ v, and repeat from Step 5

Binary Heap Delete-min

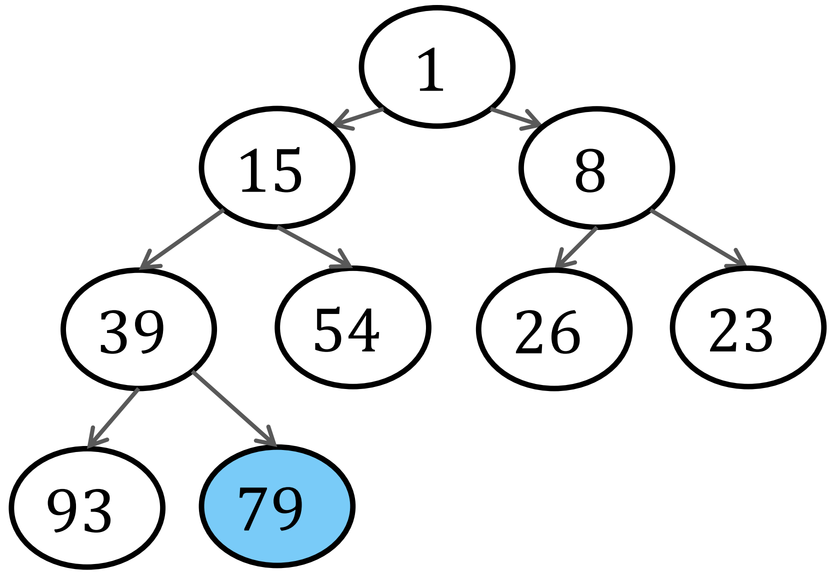
- Assume that we perform a delete-min from the binary heap below:



- Delete-min delete root node, and we should maintain the rest nodes as a complete binary tree.

Binary Heap Delete-min

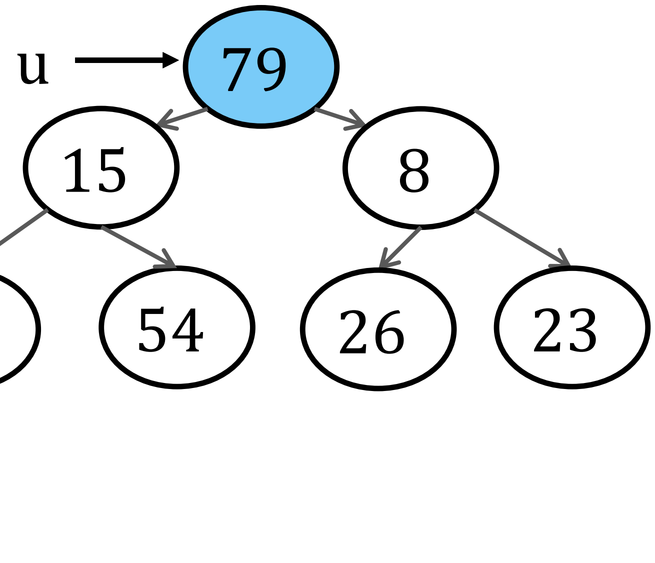
- Frist, find the rightmost leaf at the bottom level, it is node with key 79.



- Note that the tree is still a complete binary tree after removing this leaf.

Binary Heap Delete-min

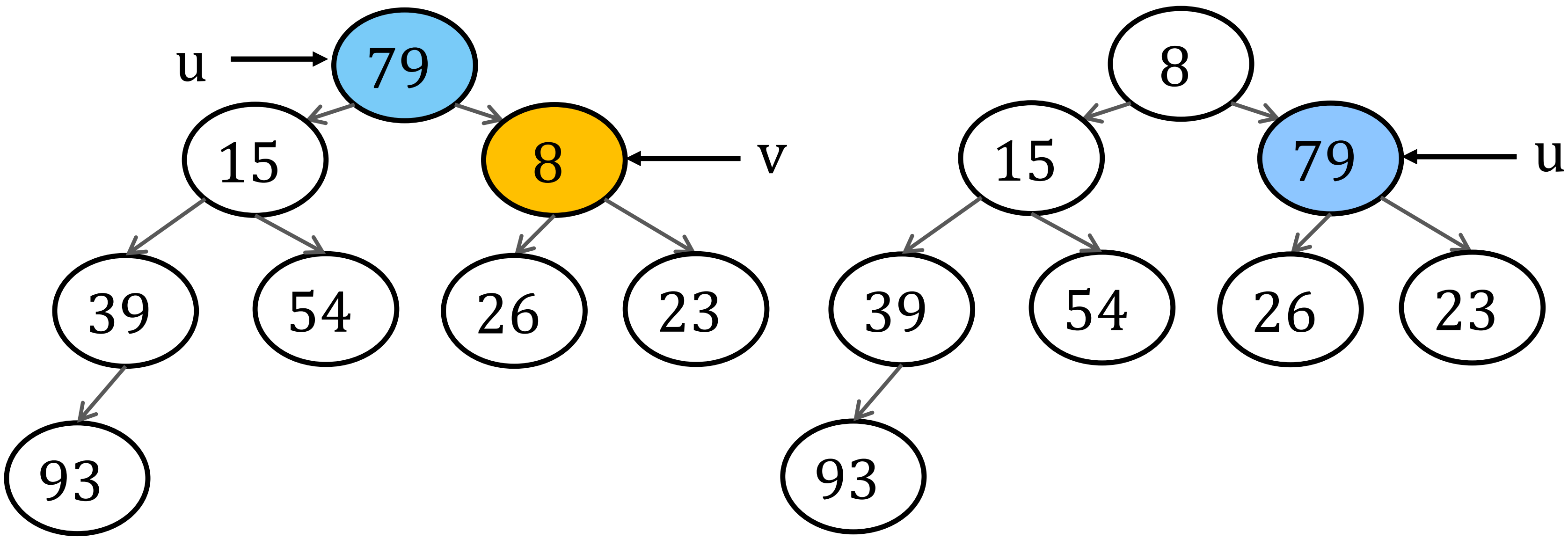
- Remove the leaf, but place the key value 79 in the root.



- Step 4: set u as the root.
- Step 5 and 6 are not true,
- Go to Step 7.

Binary Heap Delete-min

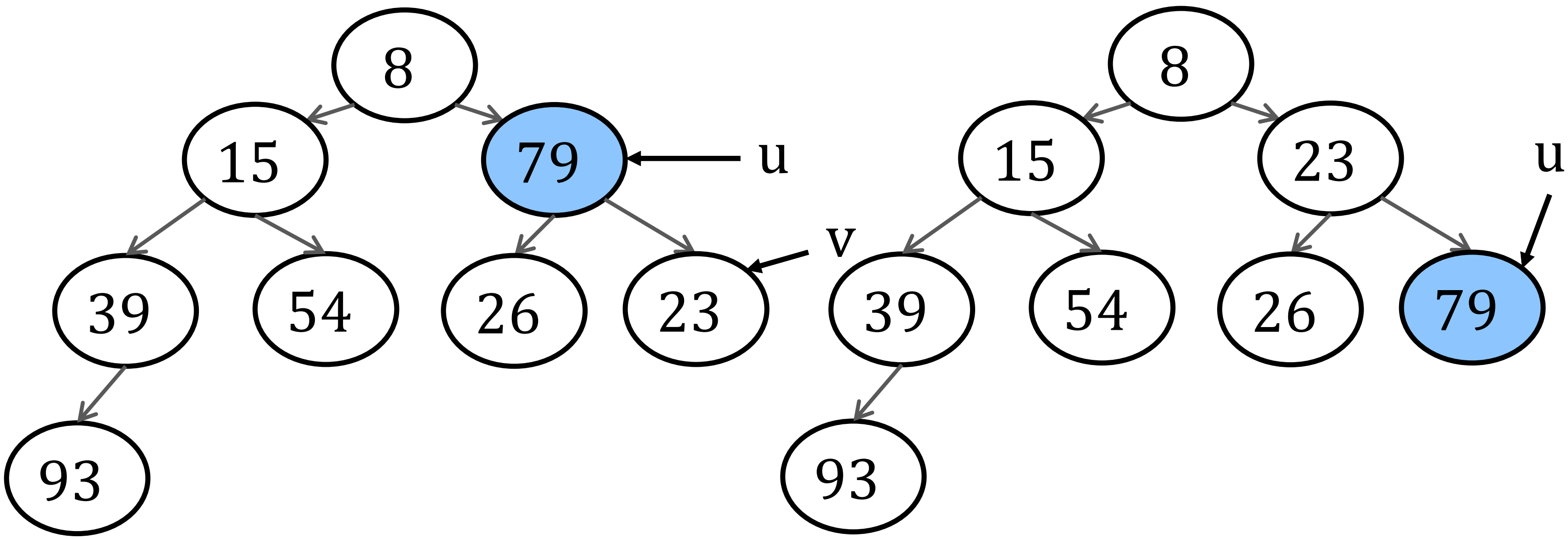
- Let v be the child of u with a smaller key.
- Swap the keys of u and v, and set u ← v



- Go to Step 5
- Step 5 and Step 6 are not true, go to Step 7

Binary Heap Delete-min

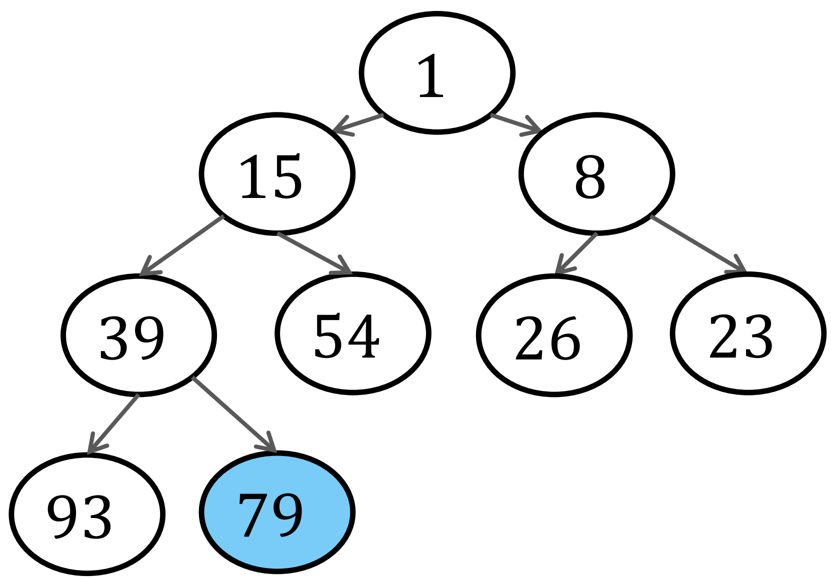
- Let v be the child of u with a smaller key.
- Swap the keys of u and v, and set u ← v



- Go to Step 5
- Step 5 is true, return. Delete-min complete.

How to find rightmost leaf?

- Before we analyzing the time complexity of insert and delete-min, let us first consider a sub-problem:
- Given a complete binary tree T with n nodes, how to identify quickly the rightmost leaf node at the bottom level of T (i.e., colored node in below tree).
  - It is Step 1 in insert algorithm, and Step 2 in delete-min algorithm

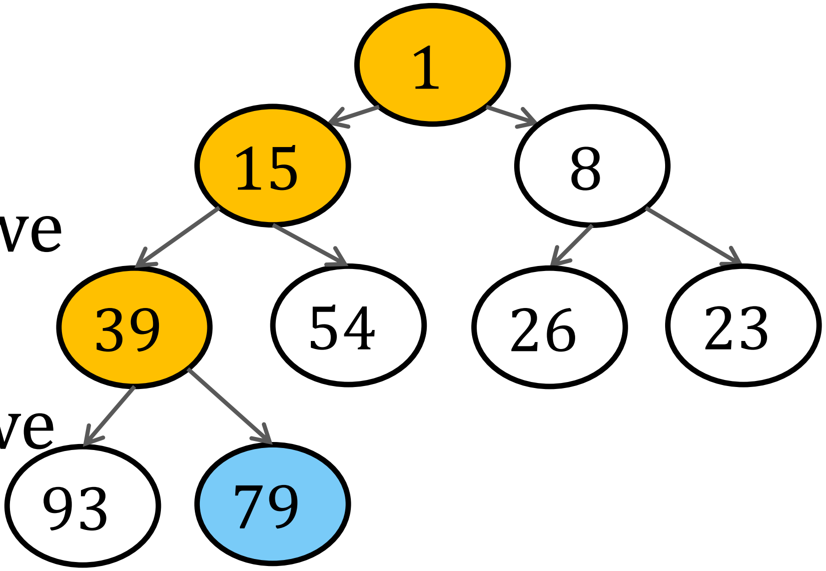


How to find rightmost leaf?

- We give a clever algorithm for solving the sub-problem in O(log n) time.
- Write the value n in binary form. We can do that in O(log n) time.
- Skip the most significant bit. We will scan the remaining bits from left to right, start from root,
  - If the bit is 0, we go to the left child of the current node
  - Otherwise, go to right child

Find Rightmost Leaf Example

- Here n = 9, binary form: 1001
- Skip the first bit '1'
- We scan the remaining bits
- Start from root node 1.
- The 2nd leftmost bit is 0, so we visit left, and go to node 15
- The 3rd leftmost bit is 0, so we visit left, and go to node 39
- The 4th leftmost bit is 1, so we turn right, and go to node 79 (done).



Time Complexity Analysis

- We are now ready to prove that our insertion and delete-min algorithms finish in O(log n) time.
- It suffices to point out the key facts:
  - Step 1 of the insertion algorithm (page 8) and Step 2 of the delete-min algorithm (page 13) can be performed in O(log n) time, using our solution to previous sub-problem
  - The rest of insertion ascends a root-to-leaf path, while that of delete-min descends a root-to-leaf path. The time is O(log n) in both cases.
- Thus, we guarantee: (1) O(n) space consumption, (2) O(log n) insertion / delete-min operations.

Our Roadmap

- Priority Queue (binary heap)
  - Min-heap insert / delete-min
- Binary Heaps in Dynamic Arrays
  - O(n) time to build min-heap
- Binary Search Tree (BST)
  - BST operators
  - Balanced BST (AVL-tree)

Binary Heaps in Dynamic Arrays

- We have already learned that the binary heap serves as an efficient implementation of a priority queue. Our previous discussion was based on pointers (for getting a parent node connected with its children). In this lecture, we will see a "pointerless" way to implement a binary heap, which in practice achieves much lower space consumption

- We will also see a way to build a heap from n integers in just O(n) time, improving the obvious O(n log n) bound.

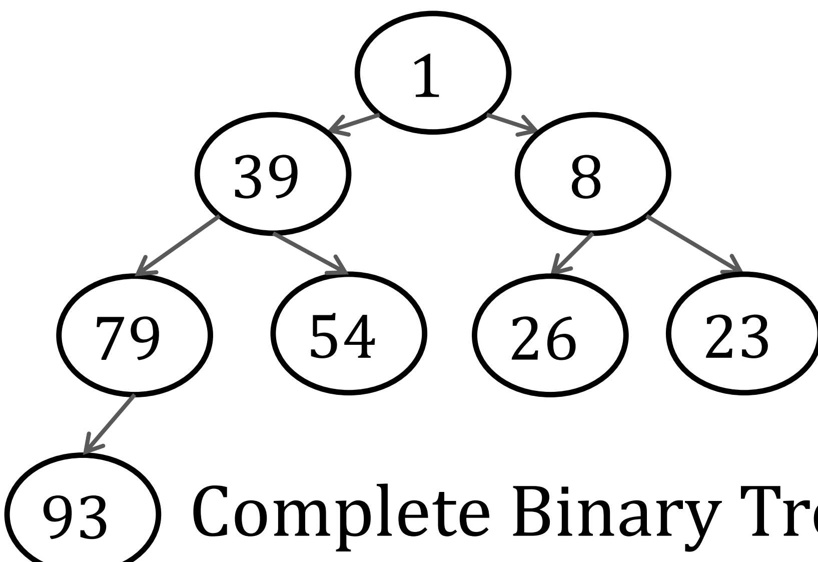
Recall

- A **priority queue** stores a set S of n integers and supports the following operations:
  - *Insert(e)*: adds a new integer to S
  - *Delete-min:* removes the smallest integer in S, and returns it.

- Let S be a set of n integers. A **binary heap** on S is a binary tree T satisfying:
  - (1) T is complete
  - (2) Every node u in T corresponds to a distinct integer in S, the integer is called the key of u (and is stored at u)
  - (3) If u is an internal node, the key of u is smaller than those of its child nodes

Storing a Complete Binary Tree

- Storing a complete binary tree using an array
- Let T be any complete binary tree with n nodes, let us linearize the nodes in the following manner:
  - Put nodes at a higher level before those at a lower level
  - Within the same level, order the nodes from left to right
- Let us store the linearized sequence of nodes in an array A of length n. Example:



Complete Binary Tree

1	39	8	79	54	26	23	93

Storing in an array

Property 1

- Let us refer to the i-th element of A as A[i], for simplicity, we assume the index of A starts from 1.

- `Lemma: Suppose that node u of T is stored at A[i]. Then, the left child of u is stored at A[2i], and the right child at A[2i+1].`
- Observe this from the example of the previous slide
- Proof leaves as your homework.
- Hints, consider the number of nodes after u, but before its left child.

More Properties

- The following is an immediate corollary of the previous lemma:
- `Corollary: Suppose that node u of T is stored at A[i]. Then, the parent of u is stored at A[`$\lfloor i/2 \rfloor$`].`
- The following is a simple yet useful fact:
- `Lemma: the rightmost leaf node at the bottom level is stored at A[n].`
- Now we have got everything we need to implement the insertion and delete-min algorithms on the array representation of a binary heap.

Insert 15

1	39	8	79	54	26	23	93

1	39	8	79	54	26	23	93	15

1	39	8	15	54	26	23	93	79

1	15	8	39	54	26	23	93	79

Delete-min

1	15	8	39	54	26	23	93	79

79	15	8	39	54	26	23	93

8	15	79	39	54	26	23	93

8	15	23	39	54	26	79	93

Performance Guarantees

- Combining our analysis on (i) binary heaps and (ii) dynamic arrays, we obtain the following guarantees on binary heap implemented with a dynamic array:
  - Space consumption O(n)
  - Insertion: O(log n) time amortized
  - Delete-min: O(log n) time amortized

Build a binary heap in array

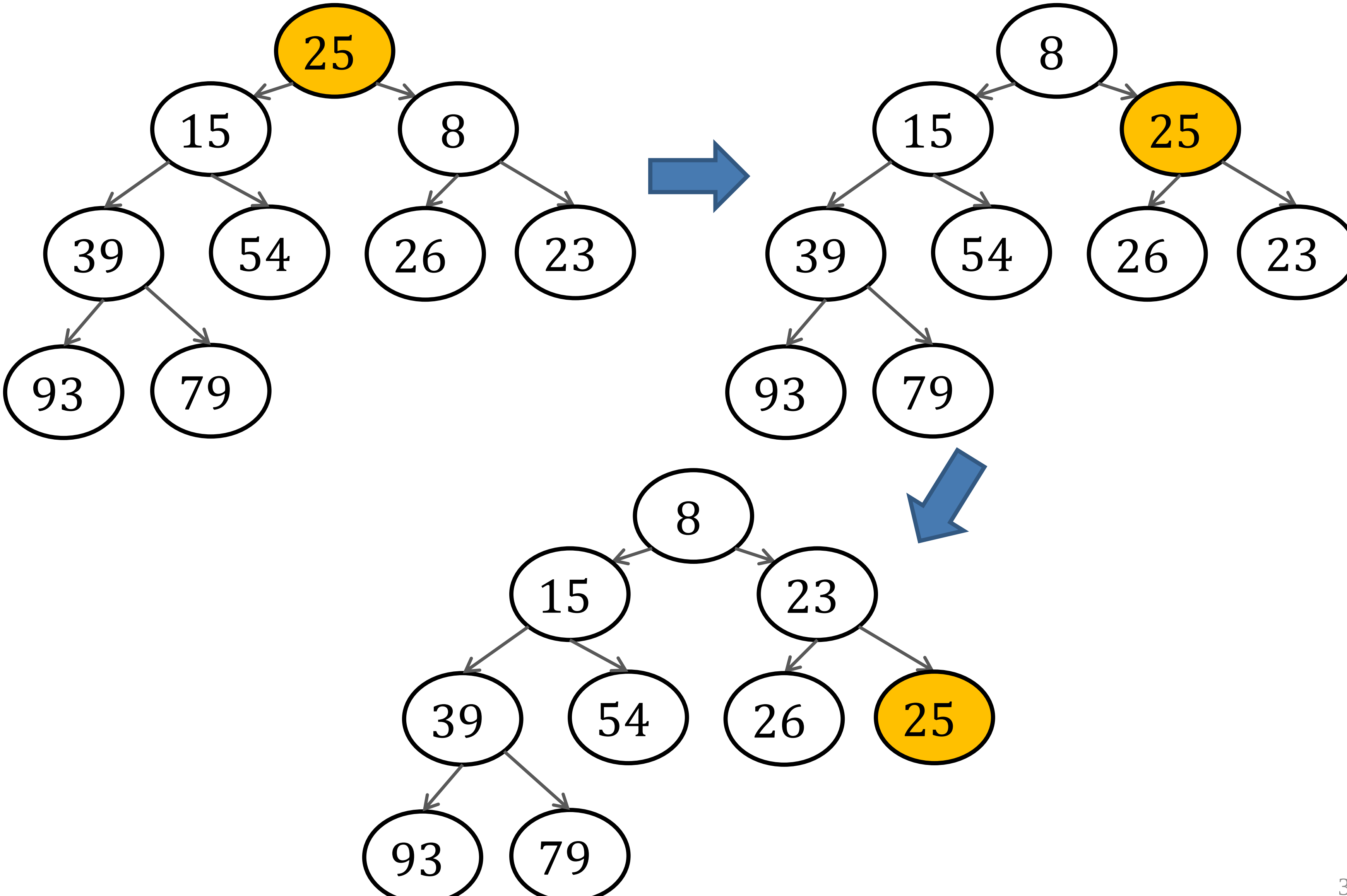
- Next we consider the problem of creating a binary heap on a set S of n integers. Obviously, we can do so in O(n log n) time by doing n insertions. However, this is an overall kill because the binary heap does not need to support any delete-min operations until all the n numbers have been inserted. This raises the question whether we can build the heap faster?

- The answer is positive: we will see an algorithm that does so in O(n) time.

Root-fix operator

- We are given a complete binary tree T with root r. It guaranteed that:
  - The left subtree of r is binary heap
  - The right subtree of r is a binary heap
  - However, the key of r may not be smaller than the keys of its children.
- We define the root-fix operation, it fixes the issue and makes T a binary heap.
- Root-fix can be done in O(log n) time – in the same manner as the delete-min algorithm (step 4 - 7)

Root-fix Example



Building a Heap

- Create an array A that stores a set S of n integers, we can turn A into a binary heap on S using the following simple algorithm, which view A as a complete binary tree T:

- `For each i=n downto 1:`
  - `Perform root-fix on the subtree of T rooted at A[i]`

- Think: why are the conditions of root-fix always satisfied?

Building a Heap example

54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
54	8	1	39	26	15	23	93

Root-fix

54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
56	8	1	39	26	15	23	93
1	8	15	39	26	54	23	93

Complexity Analysis

- Lemma: The time complexity of turn array A into a binary heap on S is O(n).

- Proof as follows:
  - view A as a complete binary tree
  - The height of T is h.
  - Without loss of generality, assume that all the levels of T are full, i.e., $n=2^{h+1}-1$.
    - Why no generality is lost?
  - Analyze the total running time of Build heap algorithm
  - Proof that $\sum_{i=0}^{h} O\left(i * 2^{h-i}\right) = O(n)$ with $n=2^{h+1}-1$.

Our Roadmap

- Priority Queue (binary heap)
  - Min-heap insert / delete-min
- Binary Heaps in Dynamic Arrays
  - O(n) time to build min-heap
- Binary Search Tree (BST)
  - BST operators
  - Balanced BST (AVL-tree)

Binary Search Tree (BST)

Binary Search Tree (especially, balanced BST) is the most powerful data structure of this course. This is without a doubt one of the most important data structures in computer science.

In extreme case, BST is equivalent to a linked list, thus, we guarantee the operations performance of BST by study AVL-tree.

Dynamic Predecessor Search

- Let S be a set of integers. We want to store S in a data structure to support the following operations:
  - A predecessor query: give an integer q, find its predecessor in S, which is the largest integer in S that does not exceed q.
  - Insertion: adds a new integer to S
  - Deletion: removes a integer from S
- Suppose that S={3,10,15,20,30,40,60,73,80}
  - The predecessor of 23 is 20
  - The predecessor of 15 is 15
  - The predecessor of 2 does not exist
- Note that a predecessor query is more general than a “dictionary look-up”. Why?

Binary Search Tree (BST)

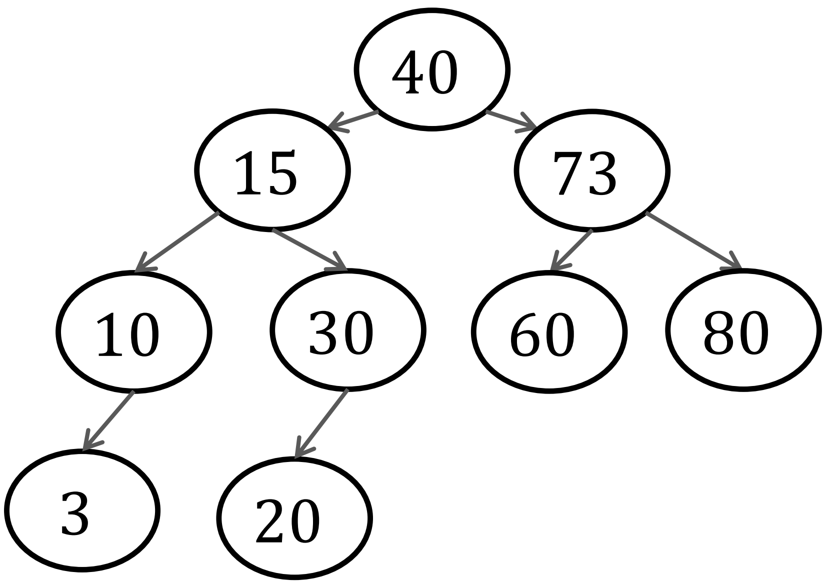
- We will learn a version of the BST that guarantees:
  - O(n) space consumption
  - O(h) time per predecessor query (hence, also per dictionary lookup)
  - O(h) time per insertion
  - O(h) time per deletion
- where n = |S|, h is the height of BST, Note that all the above complexities hold in the worst case.

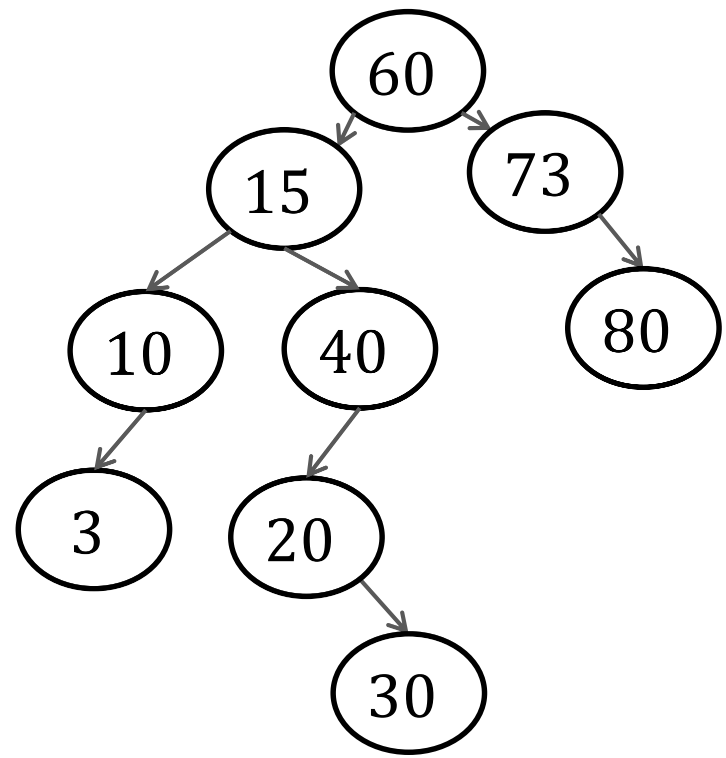
Binary Search Tree (BST)

- A BST on a set S of n integers in a binary tree T satisfying all the following requirements:
  - T has n nodes
  - Each node u in T stores a distinct integer in S, which is called the key of u
  - For every internal u, it holds that:
    - The key of u is larger than all the keys in the left subtree of u.
    - The key of u is smaller than all the keys in the right subtree of u.

BST Example

- Two possible BSTs on S = {3,10,15,20,30,40,60,73,80}



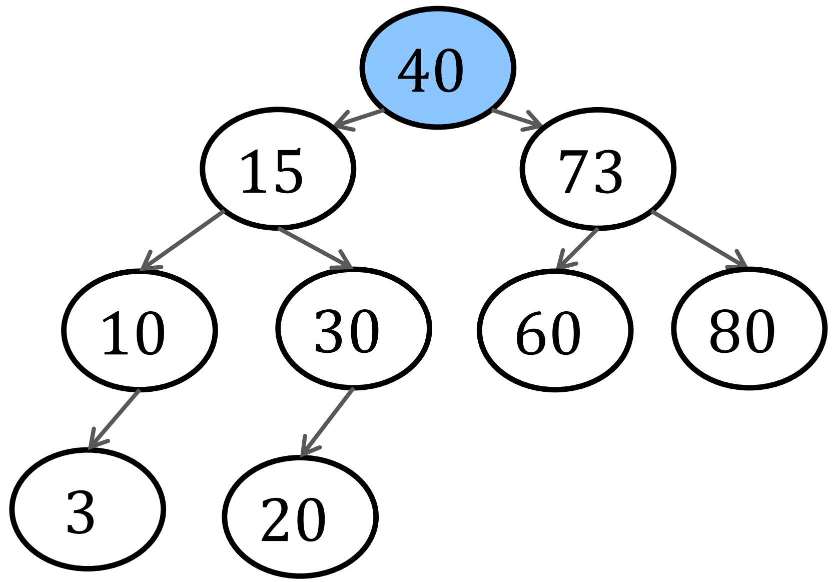


Predecessor Query

- Suppose that we have created a BST T on a set S of n integers. A predecessor query with search value q can be answered by descending a single root-to-leaf path:
  - (1) Set p ← $-\infty$ (p will contain the final answer at the end)
  - (2) Set u ← the root of T
  - (3) If u = nil, then return p
  - (4) If key of u = q, the set p to q, and return p
  - (5) If key of u > q, then set u to the left child (now u = nil if there is no left child), and repeat from Step (3)
  - (6) Otherwise, set p to the key of u and u to the right child, and repeat from Step (3)

Predecessor Query Example

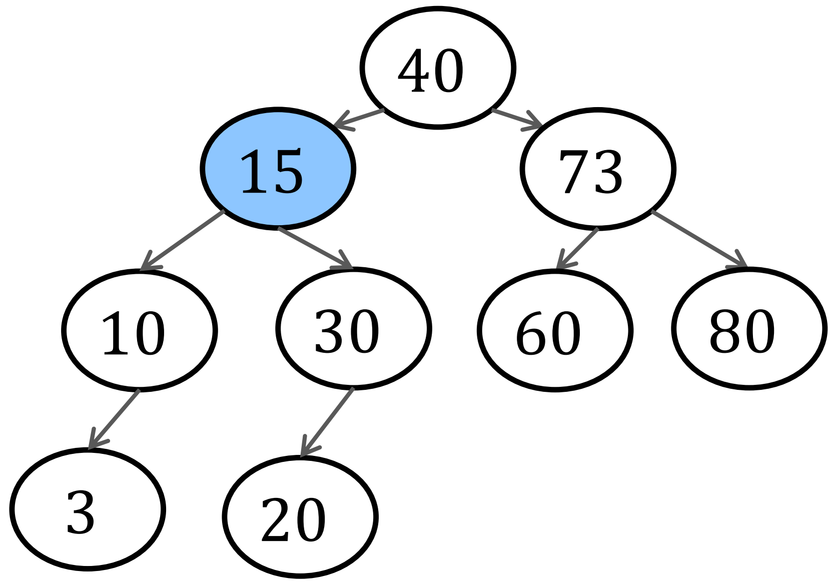
- Suppose that we want to find the predecessor of 35



- Set p $\leftarrow -\infty$, u = root 40
- (3) and (4) are not true, go to (5)
- Since 40>35, the predecessor cannot be in the right subtree of 40, so we move to the left child of 40, now u = node 15.

Predecessor Query Example

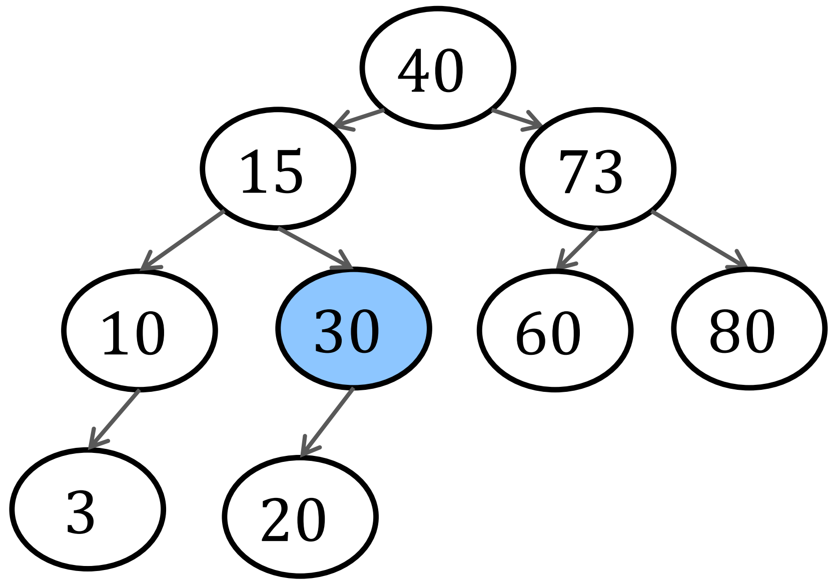
- Suppose that we want to find the predecessor of 35



- (3), (4) and (5) are not true, go to (6)
- Since 15 < 35, p ← 15, since this is the predecessor of 35 so far.
- The predecessor cannot be in the left subtree of 15, so we move u to the right child, now u = node 30.

Predecessor Query Example

- Suppose that we want to find the predecessor of 35



- (3), (4) and (5) are not true, go to (6)
- Since 30 < 35, p ← 30, since this is the predecessor of 35 so far.
- The predecessor will be in the right subtree of 30, but 30 does not have a right child. So algorithm terminates here with p = 30 as the final answer.

Time complexity Analysis

- Obviously, we spend O(1) time at each node visited. Since the height of BST is h, therefore the total query time is O(h).

Successor Query

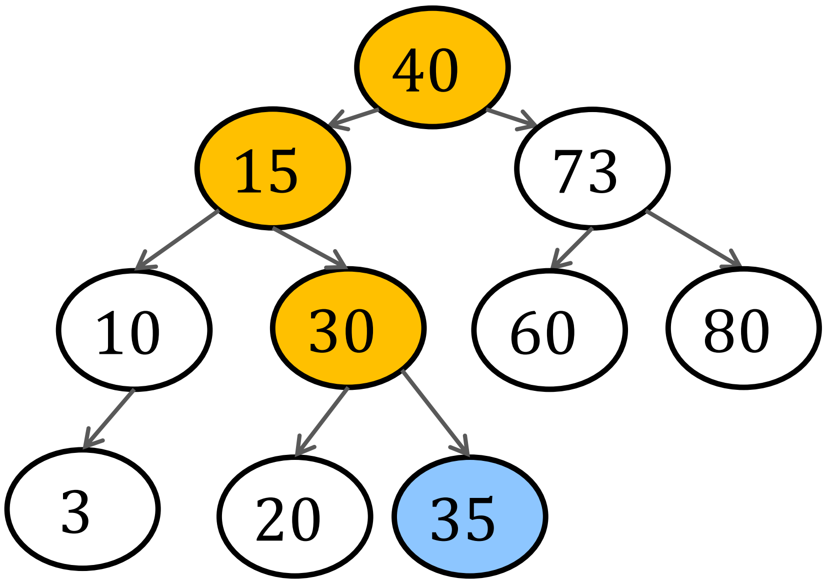
- The opposite of predecessors are successors.
- The successors of n integer q in S is the smallest integer in S that is no smaller than q.
- Suppose that S={3,10,15,20,30,40,60,73,80}
  - The successor of 23 is 30
  - The successor of 15 is 15
  - The successor of 81 does not exist
- Given an integer q, a successor query returns the successor of q in S.
- By symmetry, we know from the earlier discussion (on predecessor queries) that a successor query can be answered using a BST in O(h) time.

BST Insertion

- Suppose that we need to insert a new integer e. First create a new leaf z storing the key e. This can be done by descending a root-to-leaf path:
  - 1. Set u $\leftarrow$ the root of T
  - 2. If e < the key of u
    - 2.1 If u has a left child, then set u to the left child
    - 2.2 Otherwise, make z the left child of u, and done
  - 3. Otherwise:
    - 3.1 If u has a right child, then set u to the right child
    - 3.2 Otherwise, make z the right child of u, and done.
  - Repeat from Step 2.
- The total cost is proportional to the height of T, i.e., O(h)

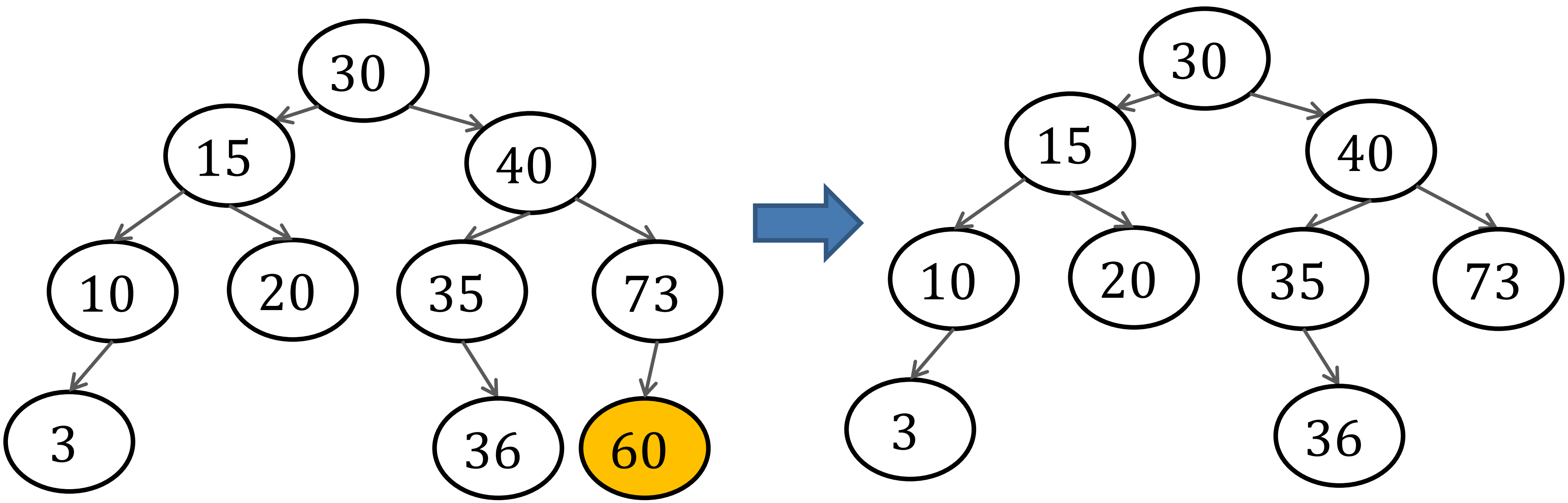
BST Insertion Example

- Inserting 35:
- u is root 40, e < the key of u, u has a left child, u ← node 15
- u is node 15, e > the key of u u has a right child, u ← node 30
- u is node 30, e > the key of u, u's right child is nil, then set z as the right child of u. Done.



BST Deletion

- Suppose that we want to delete an integer e. Frist, find the node u whose key equals to e in O(h) time (through a predecessor query).
- Case 1: if u is a leaf node, simply remove it from T.
- Example: remove 60

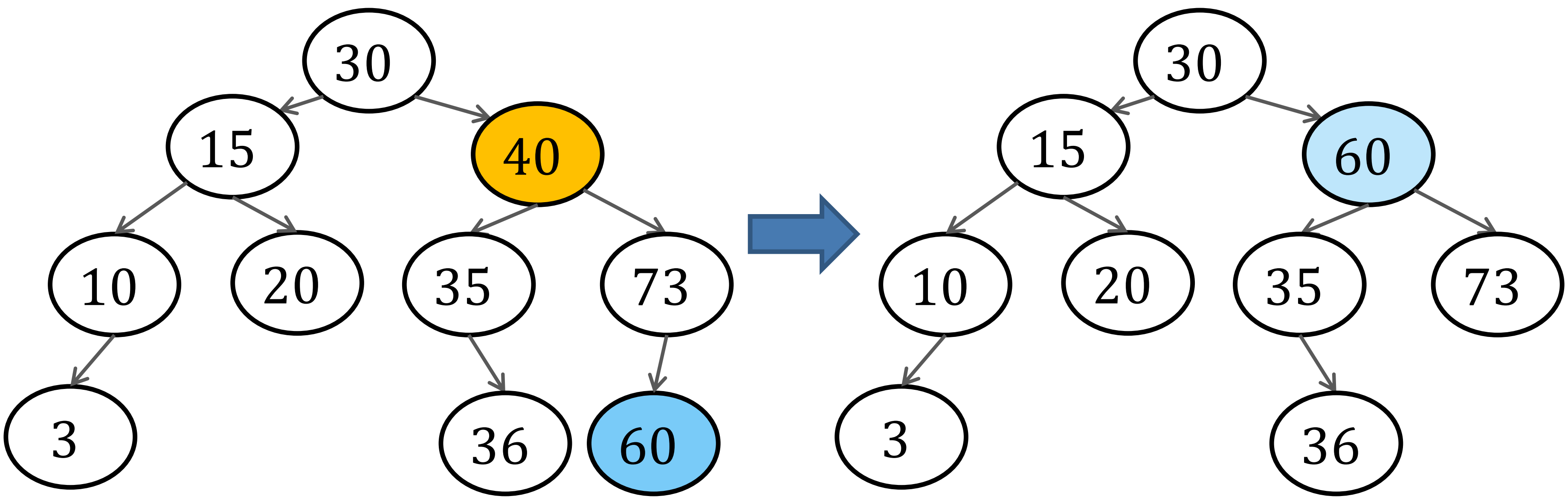


BST Deletion

- What happens if node u is not a leaf node?
- Case 2: if u has a right subtree:
  - Find the node v storing the successor s of e.
  - Set the key of u to s
  - Case 2.1: if v is a leaf node, them remove it from T
  - Case 2.2: otherwise, it must hold that v has a right child w, but not left child. Replace node v by subtree which rooted at w.
- Case 3: if u has no right subtree:
  - It must hold that u has a left child v, Replace node u by the subtree rooted at v.

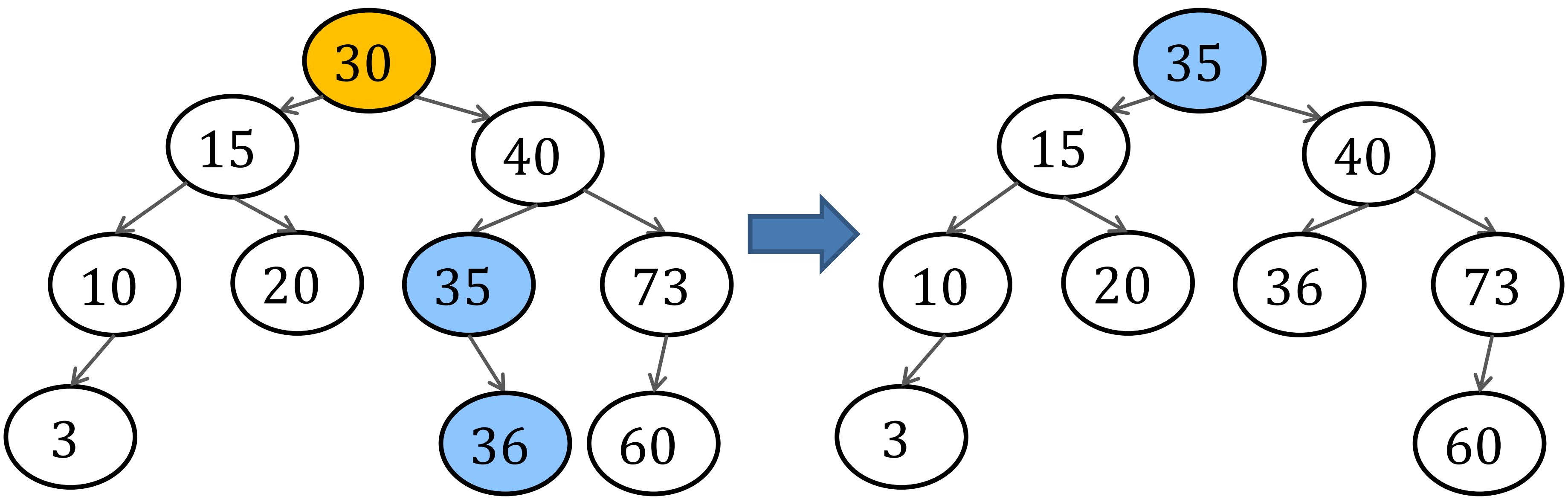
Case 2.1 Example

- Delete 40:
- u has a right subtree, node v (60) is the successor of 40.
- Set the key of u to 60
- v is a leaf, remove node v, done.



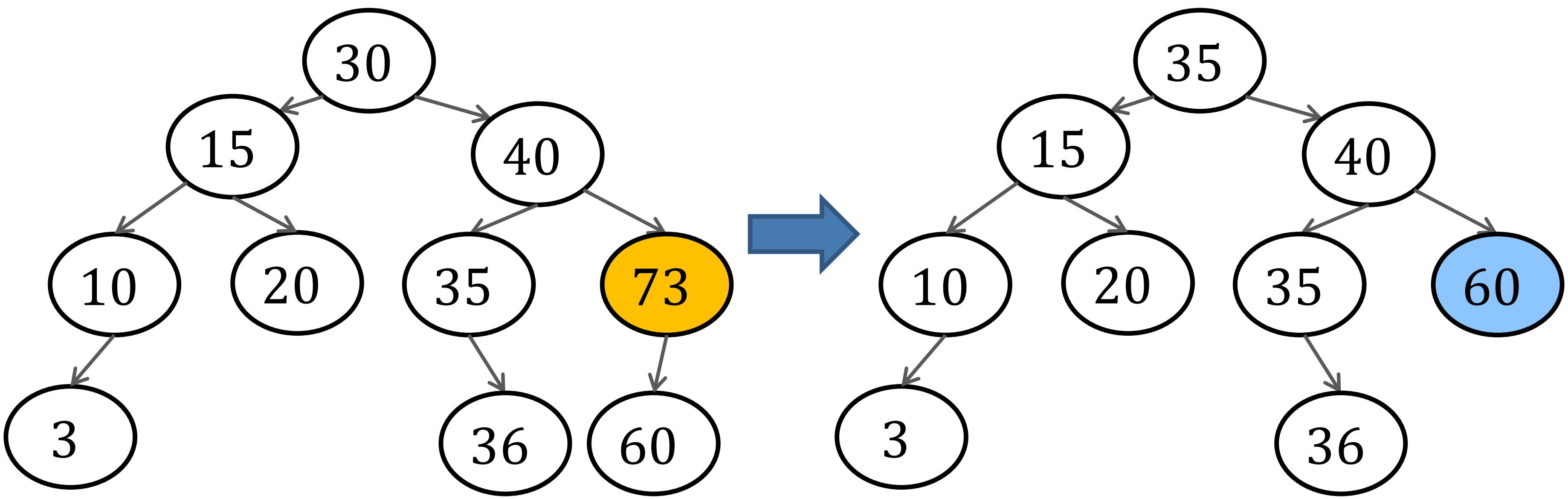
Case 2.2 Example

- Delete 30:
- u has a right subtree, node v (35) is the successor of 30.
- Set the key of u to 35
- v is not leaf node, it has right child w (36), replace node v by subtree rooted at w(36).



Case 3 Example

- Delete 73:
- u has no right subtree, and u must have a left child v (60), replace node u by node v(60).
- done.



BST Deletion

- In all above cases, we have essentially descended a root-to-leaf path (call it deletion path), and removed a leaf node.
- The cost so far is O(h), recall that the successor of an integer can be found in O(h) time.

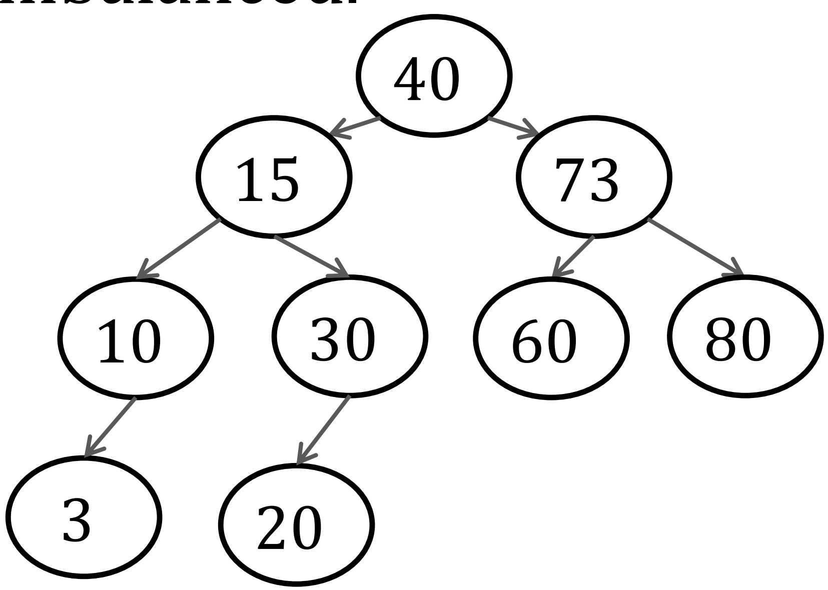
- Given a set S of n integers, what is the maximum possible height of its BST?
  - h = n, why?
  - So what is the worst-case query cost? O(n)
  - However, we can guarantee h = O(log n) is the BST is balanced BST.

What is the height of tree

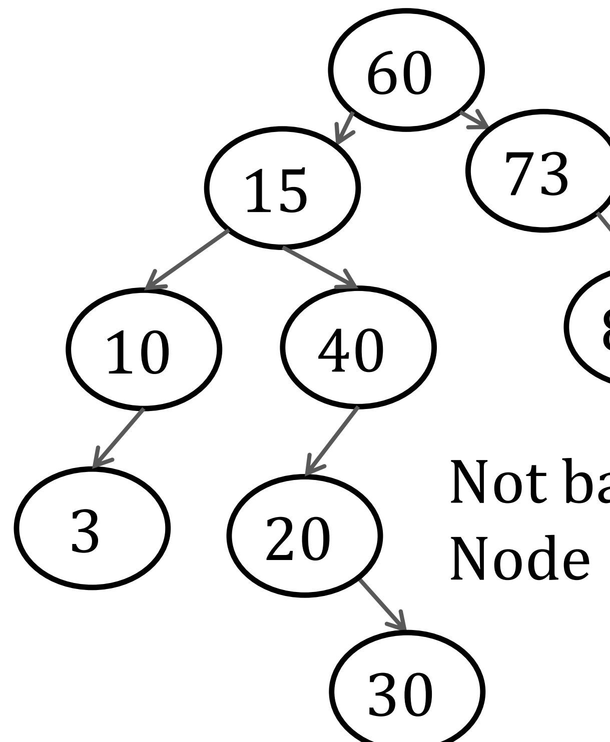
- Given a set S of n integers, what is the maximum possible height of its BST?
  - h = n, why?
- What is the worst-case query / insertion / deletion cost?
  - O(n) !!!
- How to achieve O(log n) time per operation?
  - Balanced Binary Search Tree

Balanced Binary Tree

- A binary tree T is balanced if the following holds on every internal node u of T:
  - The height of the left subtree of u differs from that the right subtree of u by at most 1.
- If u violates the above requirement, we say that u is imbalanced.



Balanced



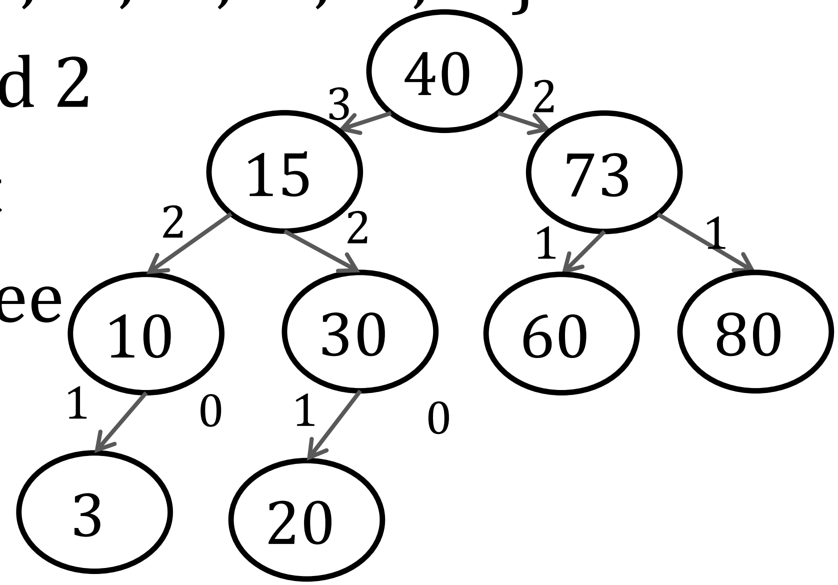
Not balanced
Node 60, 40

Height of a Balanced Binary Tree

- `Theorem: a balanced binary tree with n nodes has height O(log n).`
- Proof. (left as homework)
- Hints:
  - 1) consider minimum number of nodes in a balanced binary tree with height h
  - 2) recursive equation
  - 3) analysis two cases: case 1) h is even, case 2) h is odd.
- With the height of balanced binary tree is O(log n), we can conclude that the cost of query operation is O(log n) on a balanced binary search tree.
- How about the cost of insertion and deletion on it?

Balanced BST

- An AVL-tree on a set S of n integers is a balanced binary search tree T, where the following hold on every internal node u
  - u stores the heights of its left and right subtrees.
- An AVL-tree on S = {3,10,15,20,30,40,60,73,80}
- For example, the number 3 and 2 near root 40 indicate that its left subtree has height 3, right subtree has height 2.
- By storing the subtree heights an internal node know whether it has become imbalanced

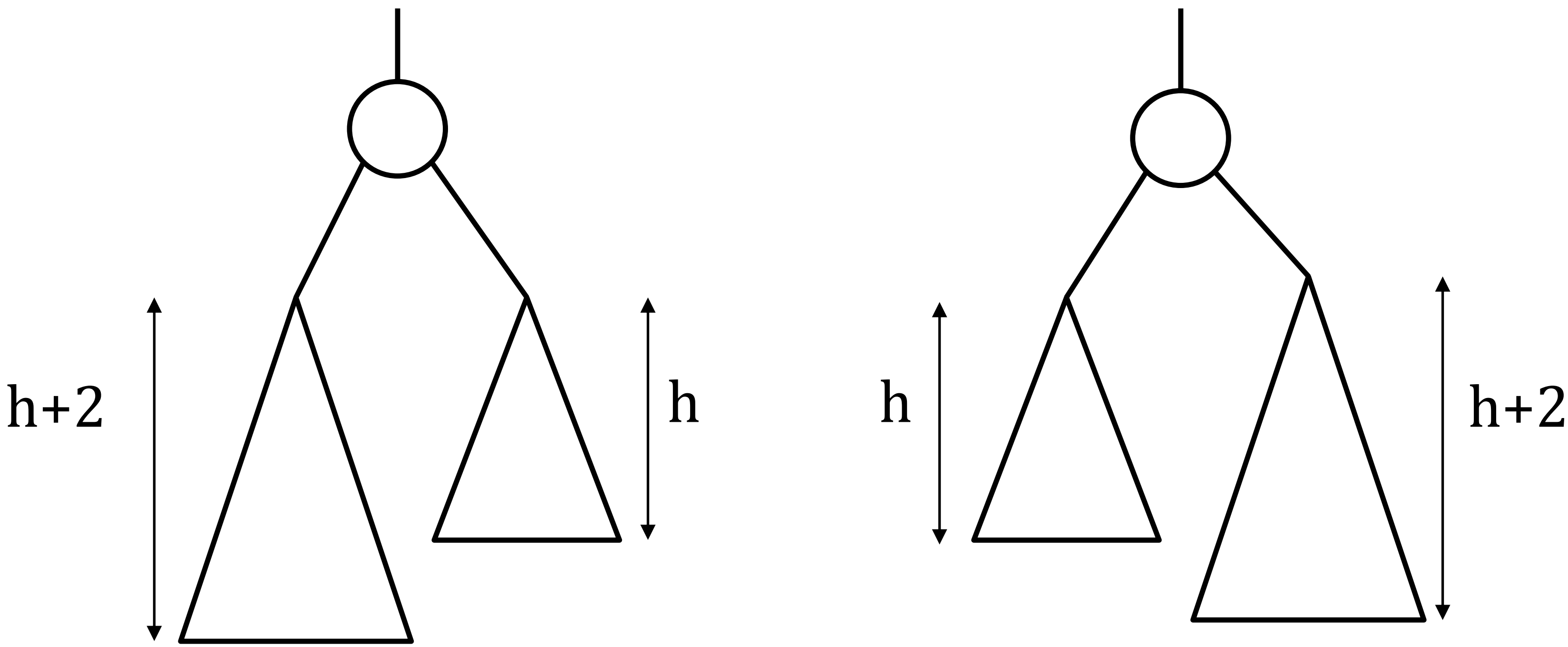


Balanced BST

- Next we will explain how to perform updates. The most important step is remedy a node u when it becomes imbalanced.
- It suffices to consider a scenario called 2-level imbalance. In this situation, two conditions apply:
  - There is a difference of 2 in the heights of the left and right subtree of u.
  - All the proper descendants of u are balanced
- We will first explain how to rebalance u in the above situation

2-level imbalance

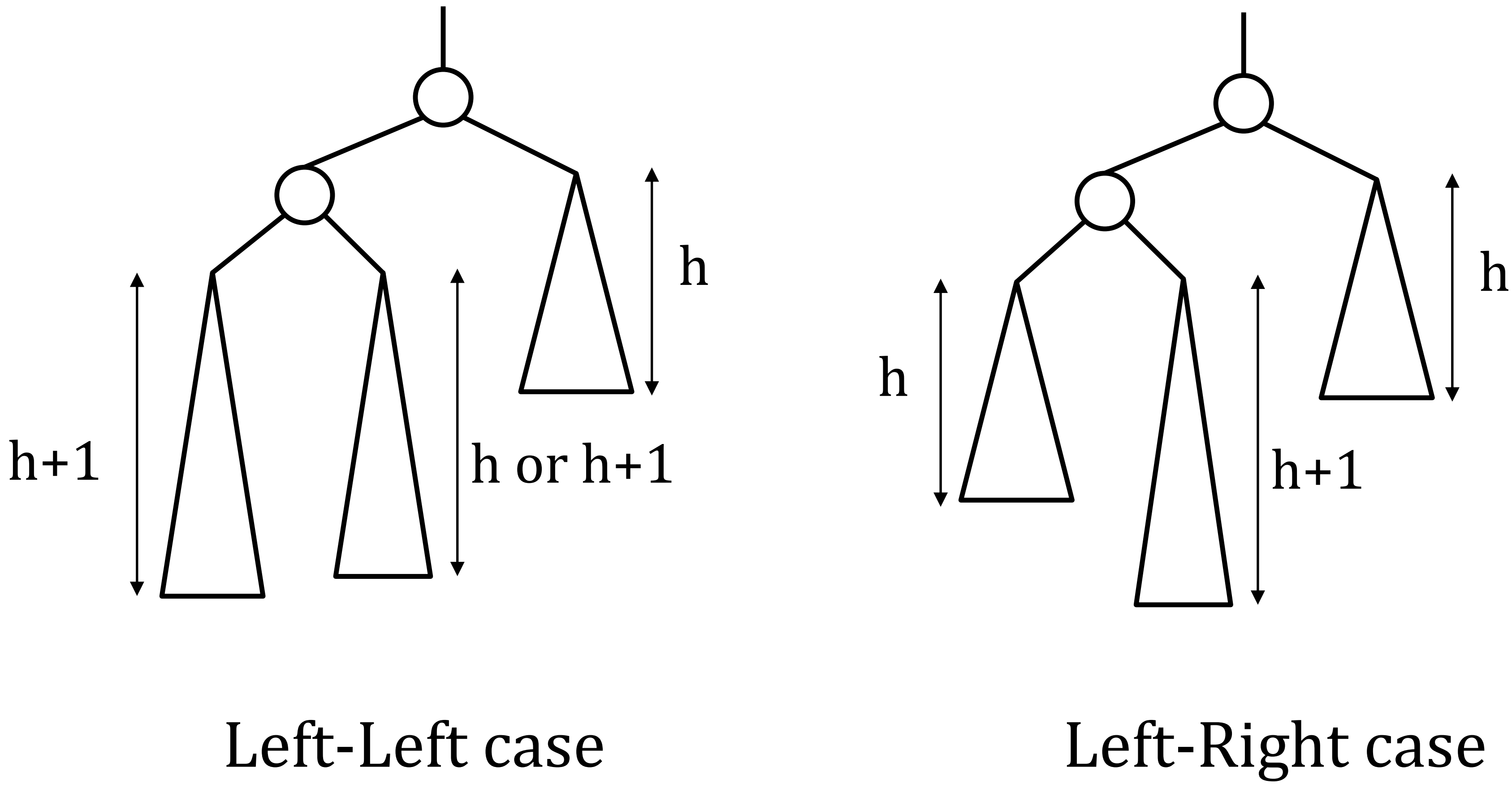
- There are two cases:



- Due to symmetry, it suffices to explain only the left case, which can be further divide to a left-left and a left-right case, as shown next.

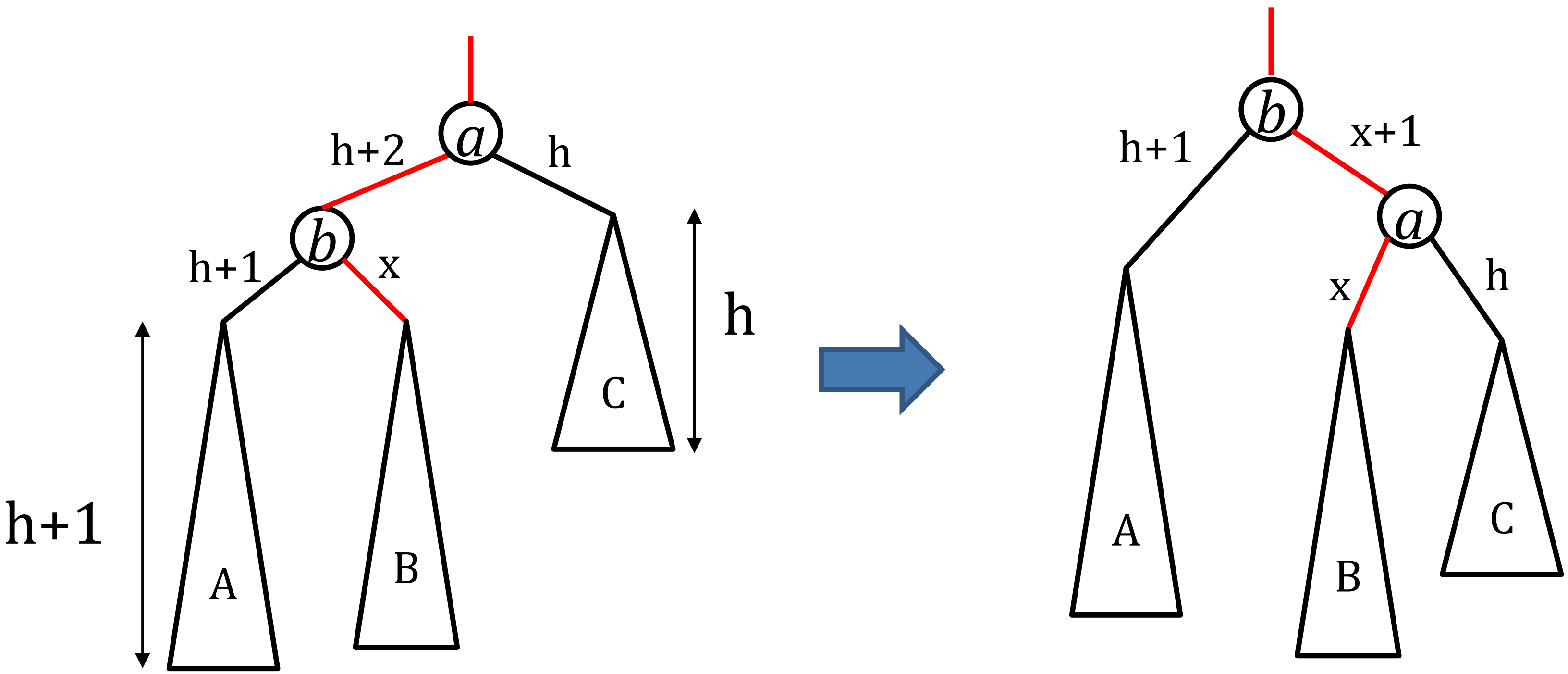
2-level imbalance

- There are two cases:



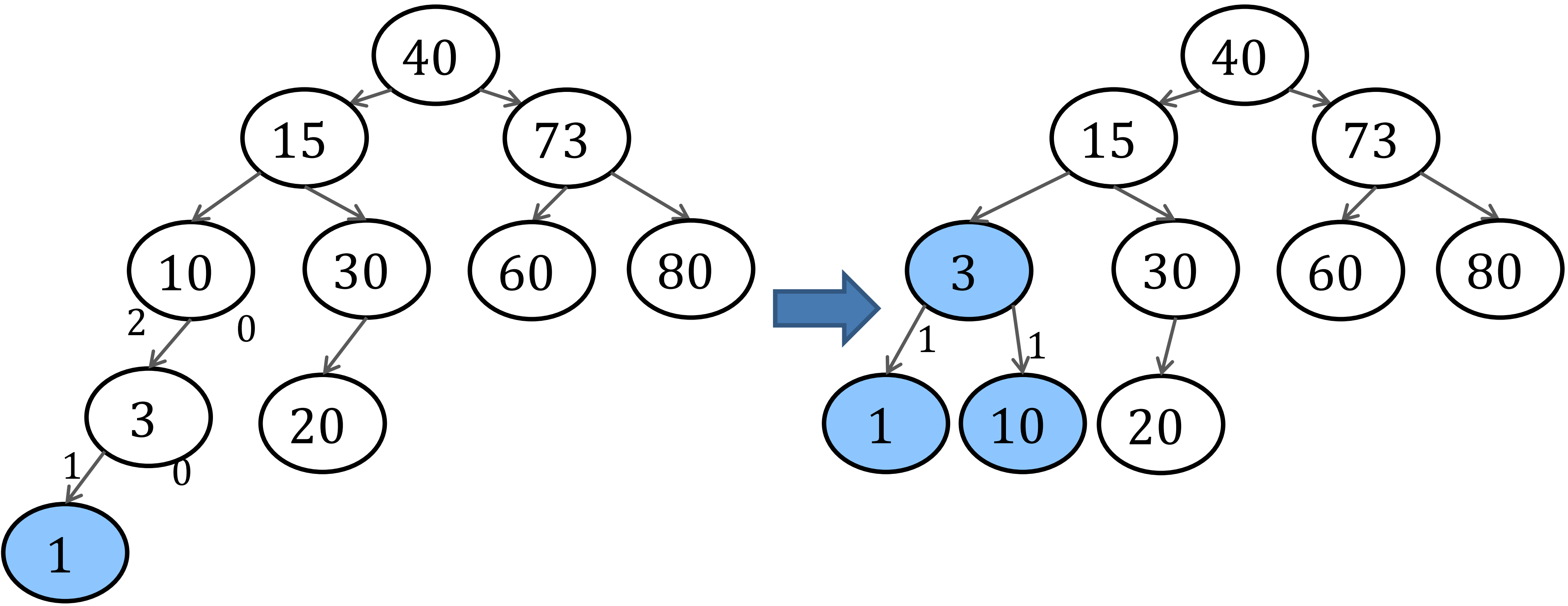
Rebalance Left-Left

- By a rotation:



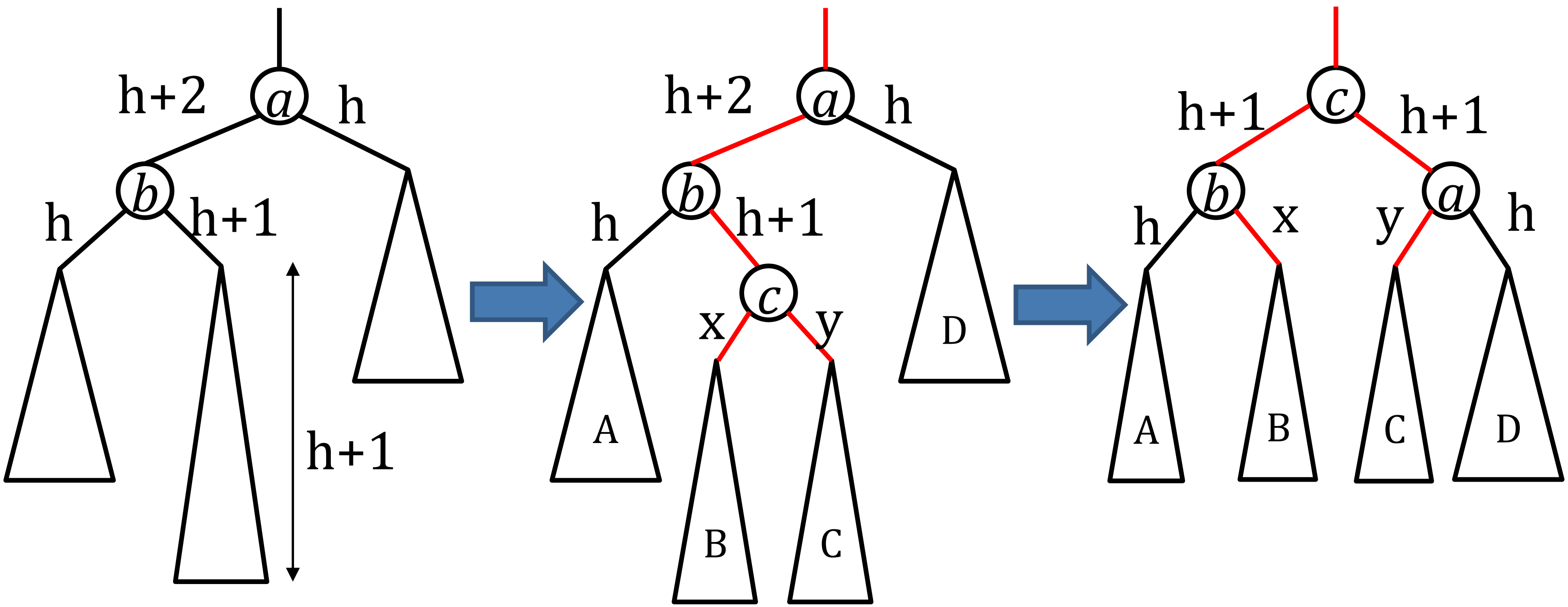
- Only 3 pointers to change (the red ones). The cost is O(1).
- Recall that x = h or h+1

Rebalance Left-Left Example



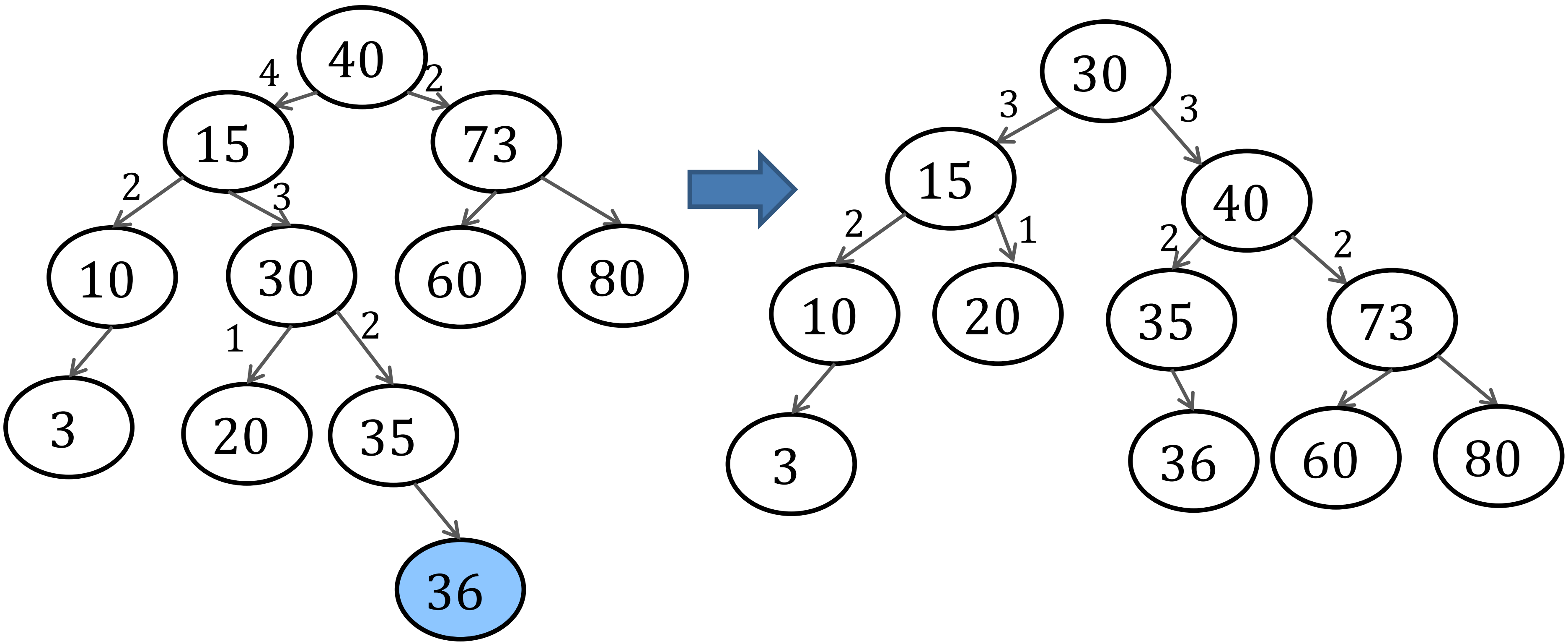
Rebalance Left-Right

- By a double rotation:



- Only 5 pointers to change (see above). Hence, the cost is O(1).
- Note that x and y must be h or h-1. Furthermore, at least one of them must be h (why?)

Rebalance Left-Right Example



Insertion and Deletion Time

- Insertion time analysis
  - It will be left as an exercise for you to prove
    - Only 2-level imbalance can occur in an insertion
    - Once we have remedied the lowest imbalance node, all the nodes in the tree will become balanced again
  - Thus, we can conclude the insertion cost in a balanced BST is O(log n), why?
- Deletion time analysis
  - It will be left as an exercise for you to prove
    - Only 2-level imbalance can occur after a deletion
  - Thus, we can conclude the deletion cost in a balanced BST is O(log n)

Balanced BST

- We now conclude our discussion on the AVL-tree, which provides the following guarantees:
  - O(n) space consumption
  - O(log n) time per predecessor query (hence, also per dictionary lookup)
  - O(log n) time per insertion
  - O(log n) time per deletion
- All the above complexities hold in the worst case.

Thank You!