Advanced Heaps

Leftist Heaps & Skew Heaps

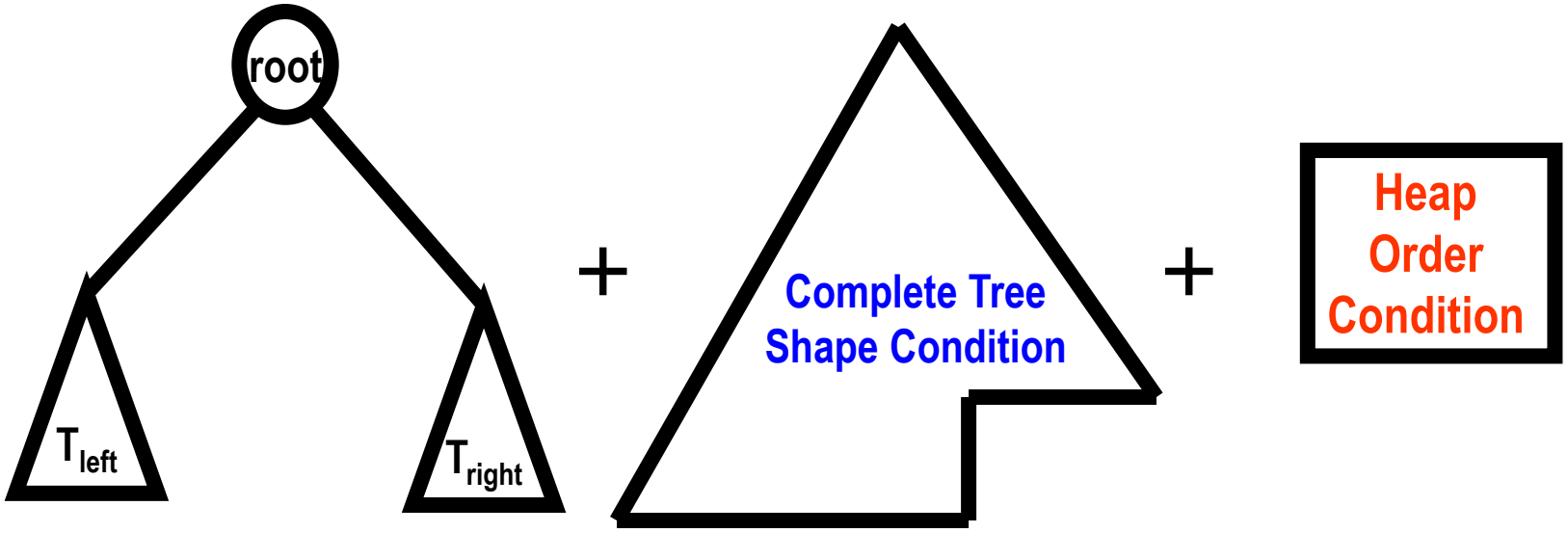
Prof. Young Park

Binary Heaps

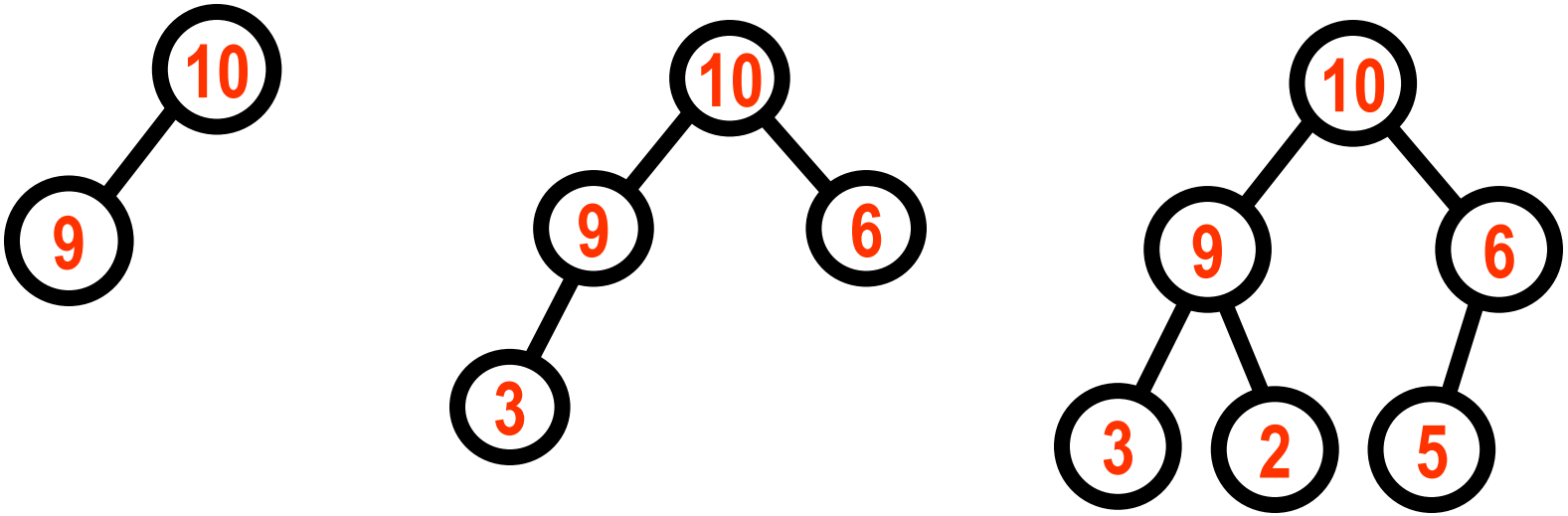
A Binary Heap?

- A binary tree that satisfies two conditions:
  - The **shape/structure** condition
    - **Complete binary tree**
    - **O(log n) height!**
  - The **(partial) order** condition between every node and the nodes in its left and right subtrees.
    - Heap order property
    - **There is no necessary ordering relationship between a node and its siblings!**
    - **Max-Heap or Min-Heap**

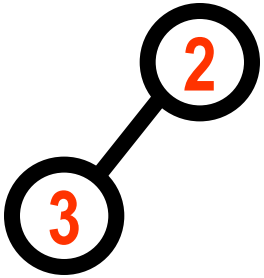
A Binary Heap

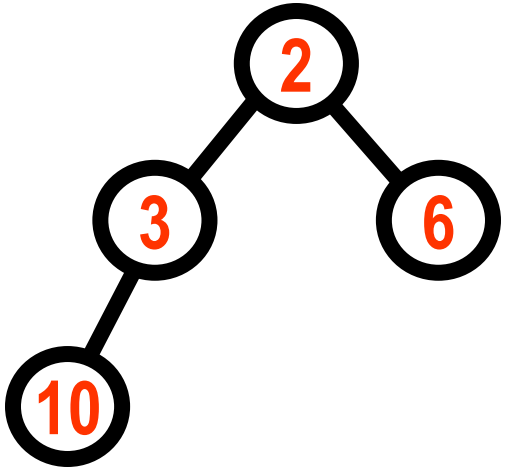


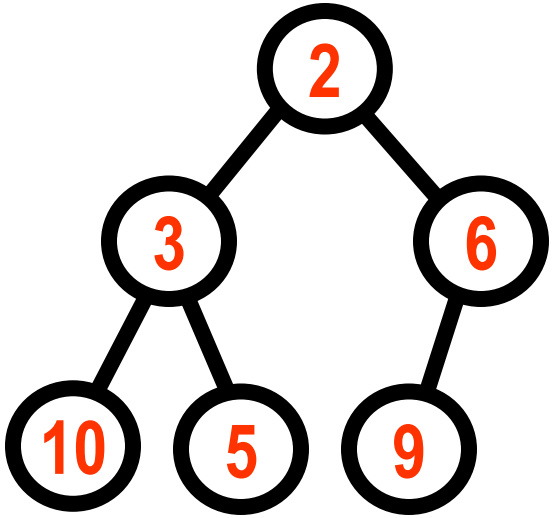
Example: Binary Max Heap?



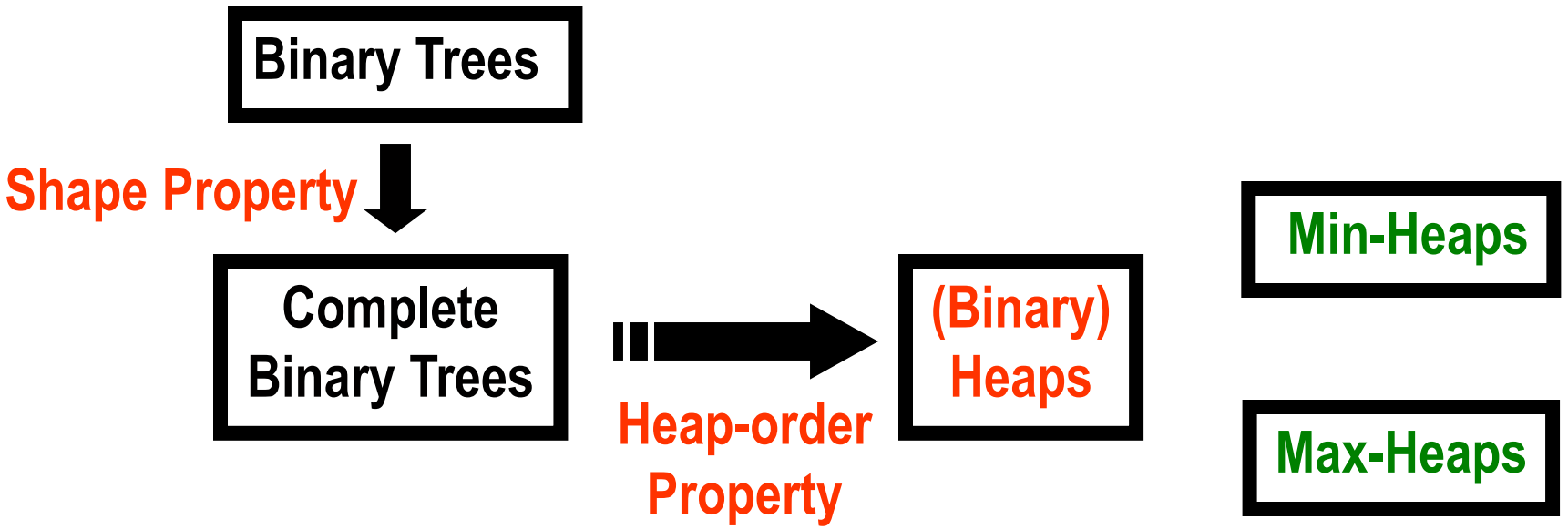
Example: Binary Min-Heap?







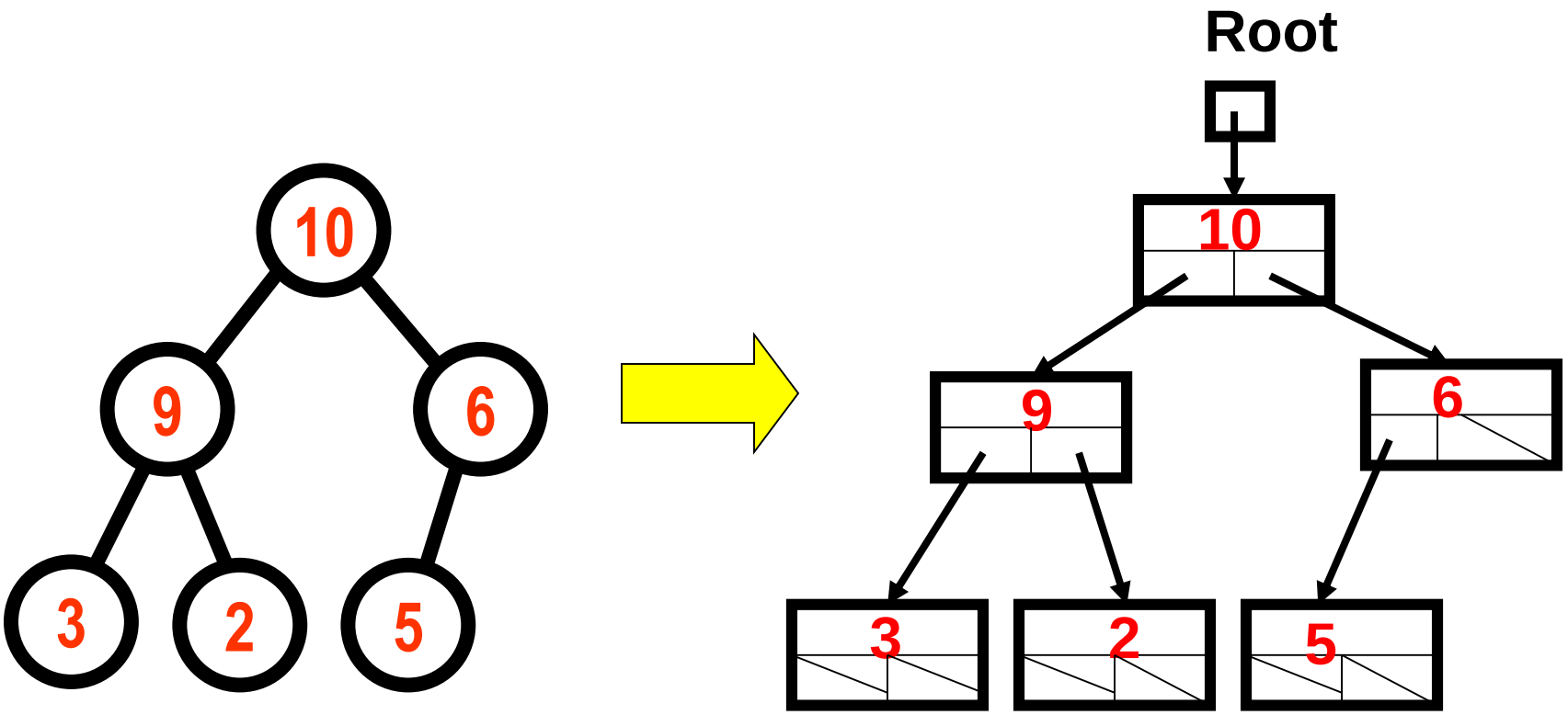
The World of Binary Heaps



How to Represent a Binary Heap?

- Pointer-based Representation
- Array-based Representation

A Pointer-Based Representation of a Binary Heap

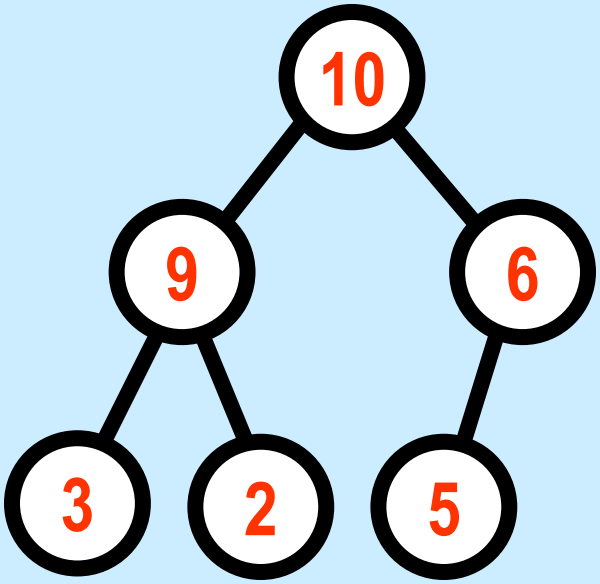


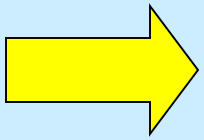
Array-Based Representation?

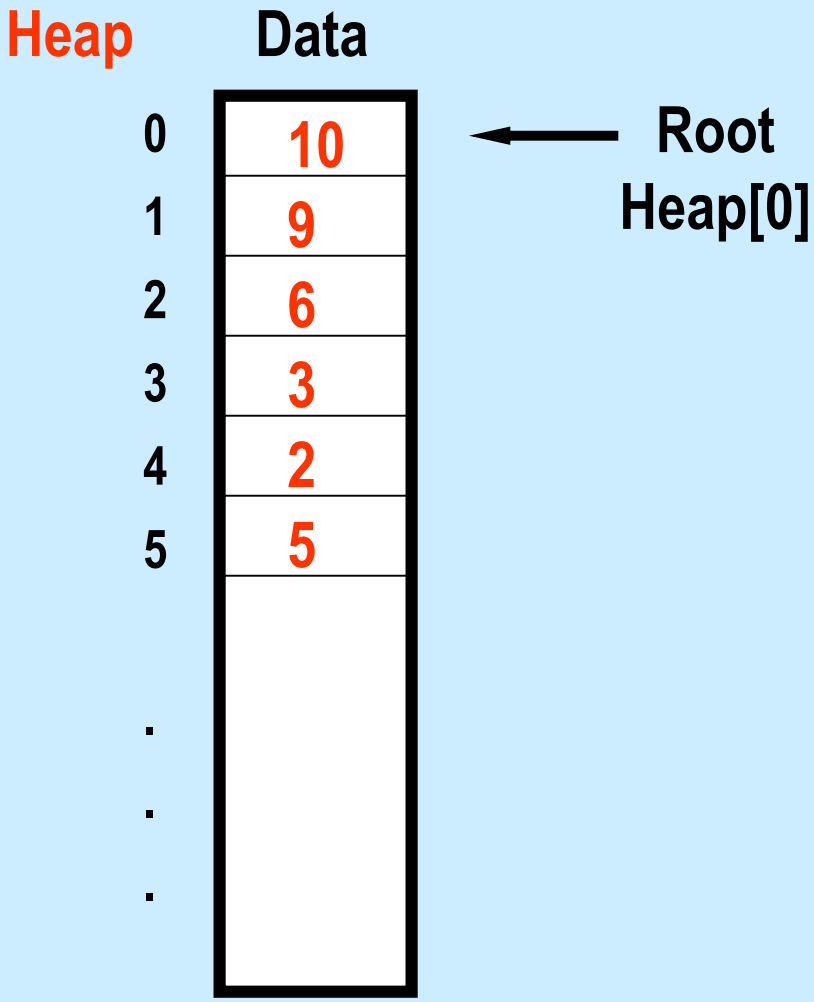
- A heap is a **complete** binary tree!
- **So, an efficient array-based representation!**

  - Represent a node in the complete binary tree as
    - A data
  - Represent the complete binary tree as an array.
- **Array Implementation for Complete Binary Trees!**

An Array-Based Representation of a Binary Heap







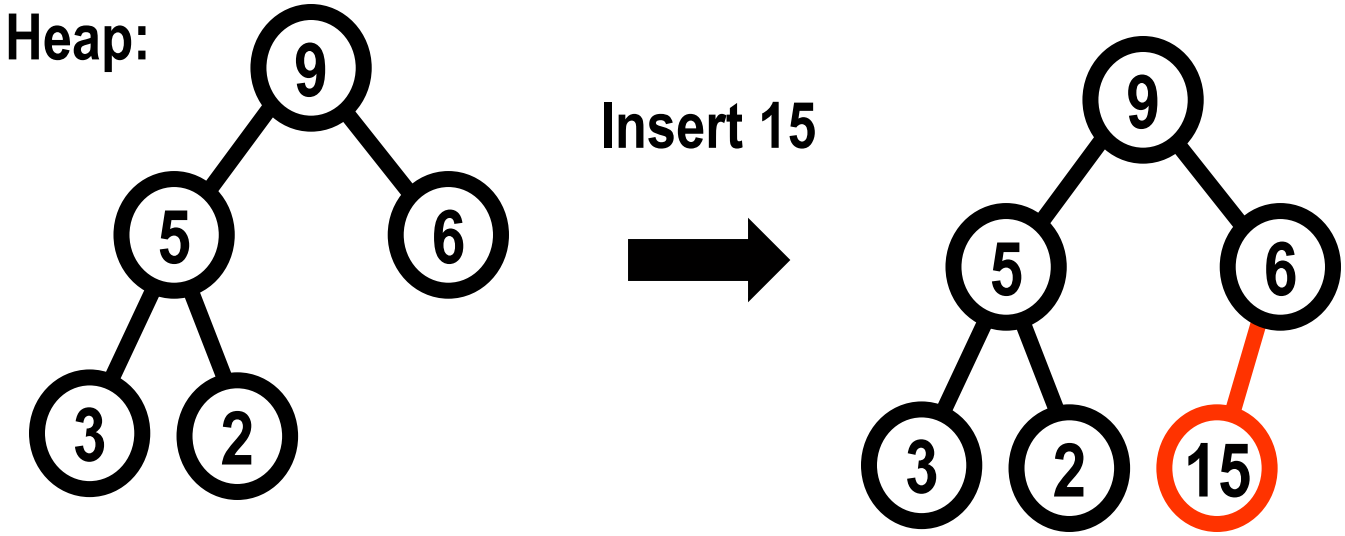
Finding Parent and Children in a Binary Heap

- For any node Heap[**i**]
  - Its left child =
    - Heap [**2 * i + 1**]
  - Its right child =
    - Heap [**2 * i + 2**]
  - Its parent =
    - Heap [**(i -1) / 2**]

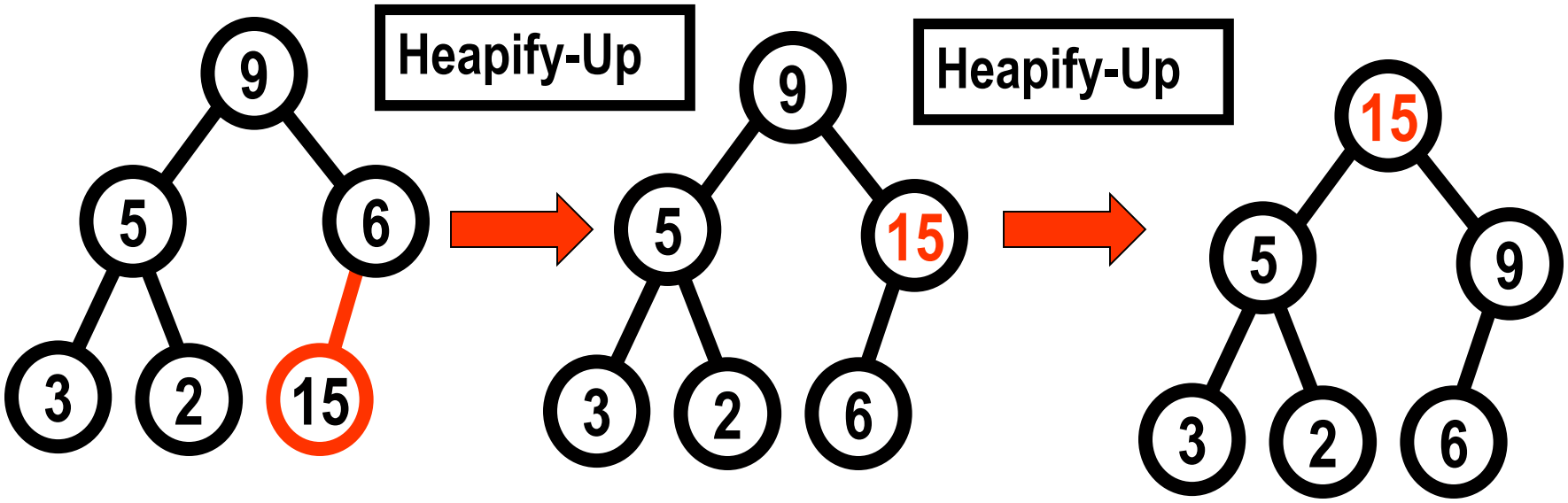
Operations on Binary Max-heaps

- **Insert (add)** a new item to the heap.
- **Retrieve** and then **delete a heap's root item**. (This item has the largest search key.)
- ...

Insert an Item Into a Binary Max-heap



Example: Insert 15



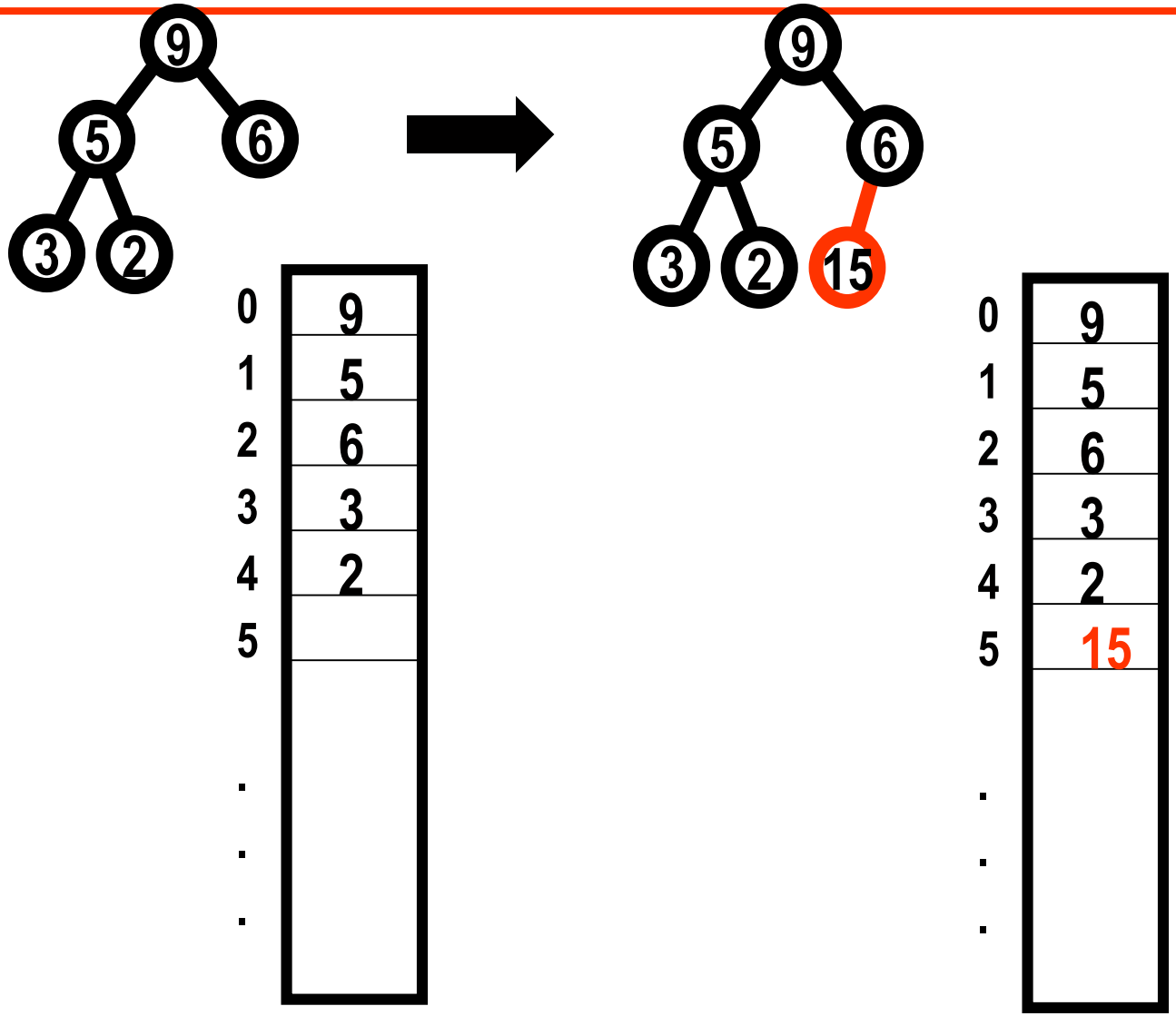
Heapify-Up

- Heapify-Up (Heap, root, bottom):
  - The heap order may be violated only at the bottom node.
  - Restores the heap order between the root and the bottom.
  - If bottom > root then
    - Set Parent to the index of bottom node;
    - If heap[Parent] < heap[bottom] then
      - Swap heap[Parent] with heap[bottom];
      - Heapify-Up (Heap, root, Parent);

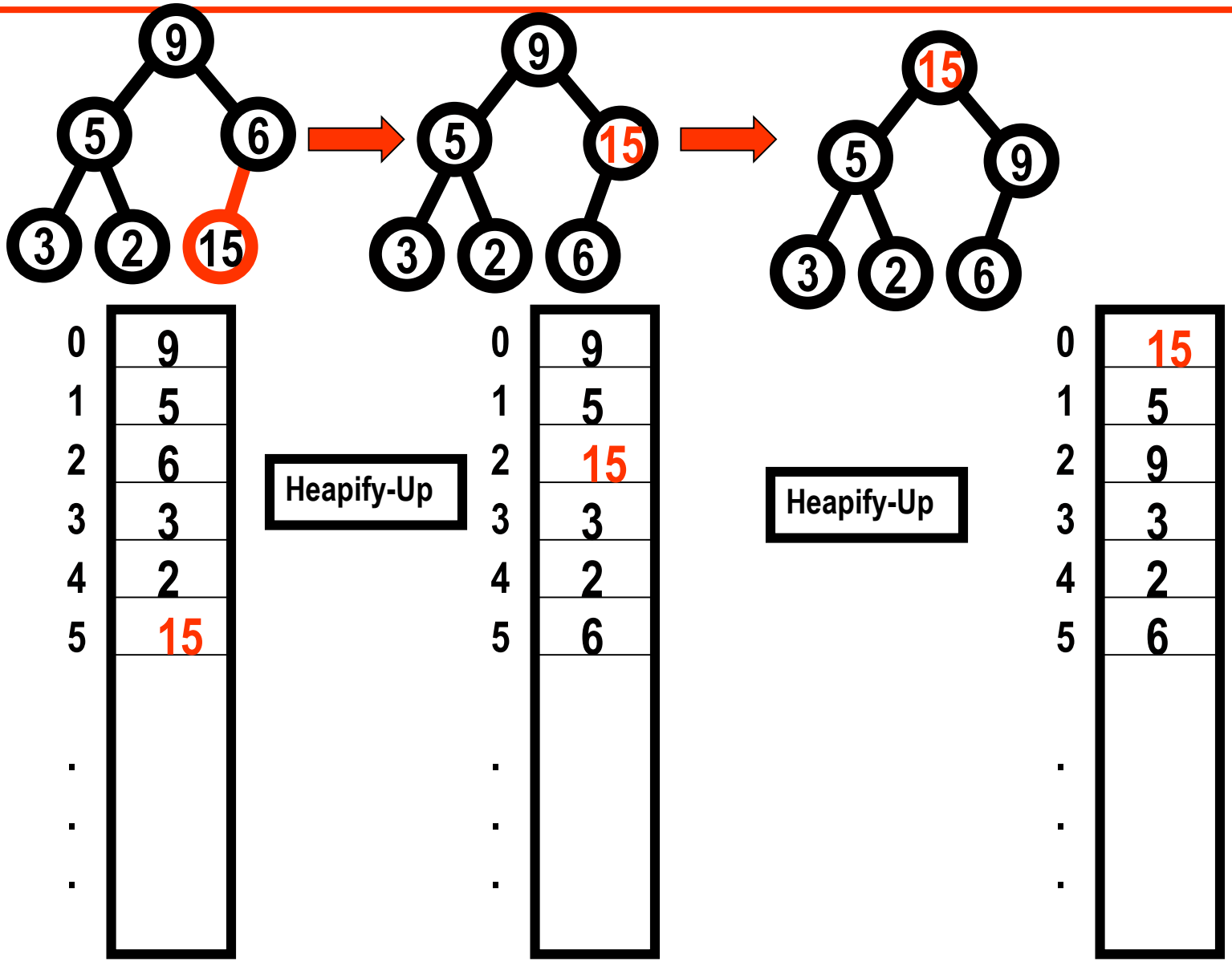
Insert an Item Into a Binary Max-heap

- Insert (Heap, item):
  - Increment the size of the heap;
  - Put item in the next available position;
  - Heapify-Up (Heap, 0, size-1);

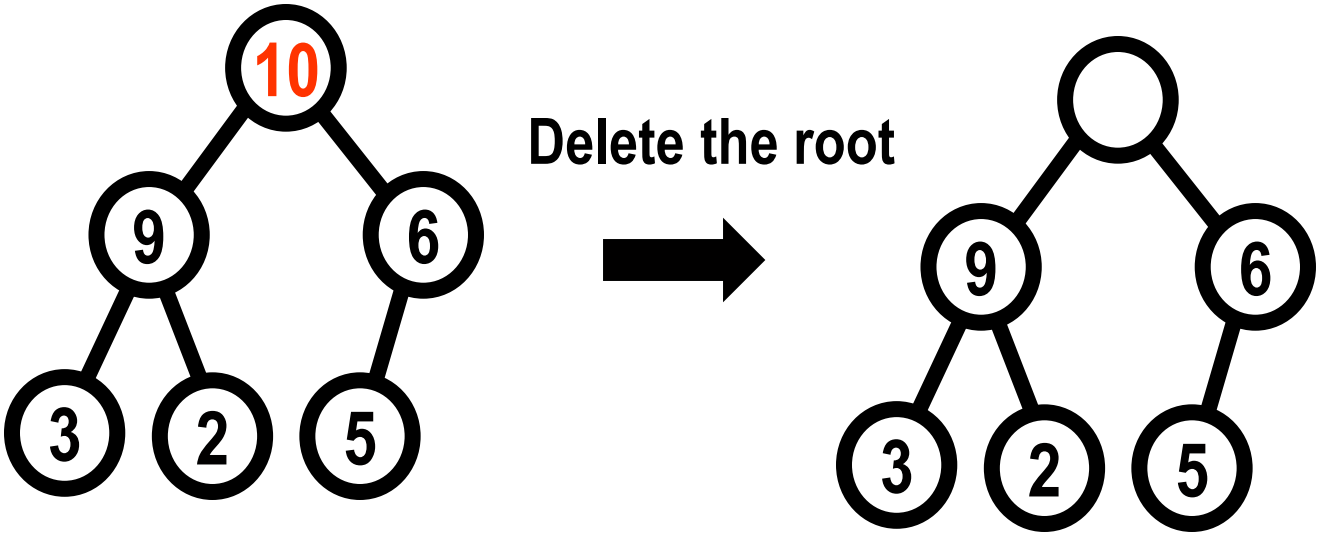
Example: Insert 15



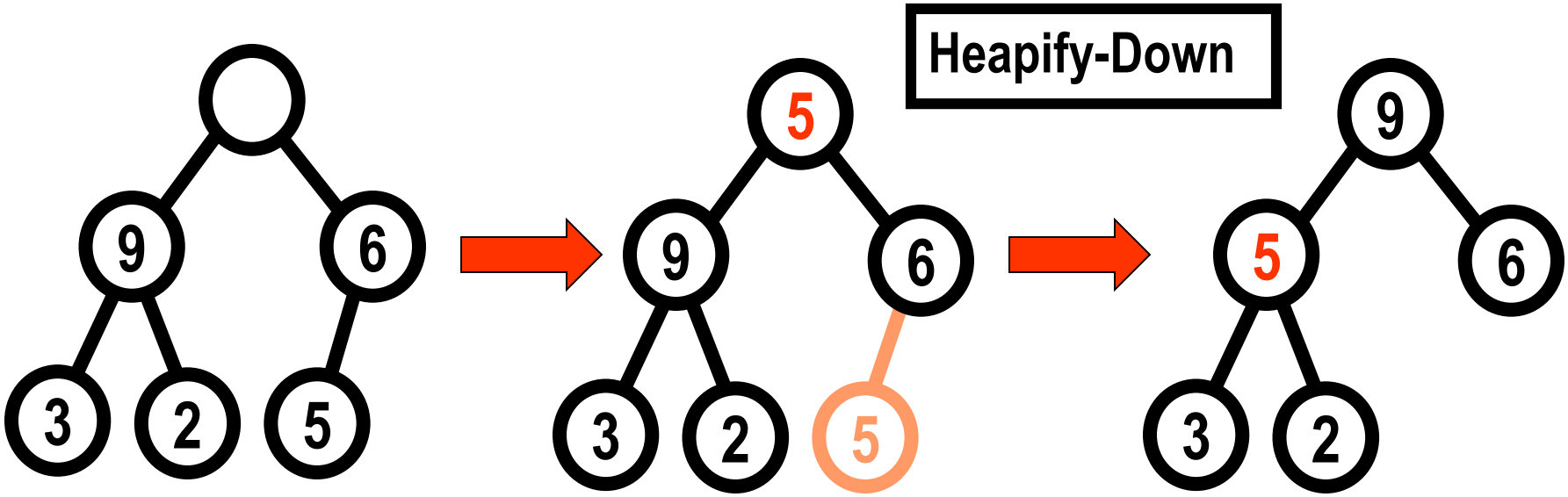
Example: Insert 15



Retrieve & Delete a heap's root item



Example: Delete the root



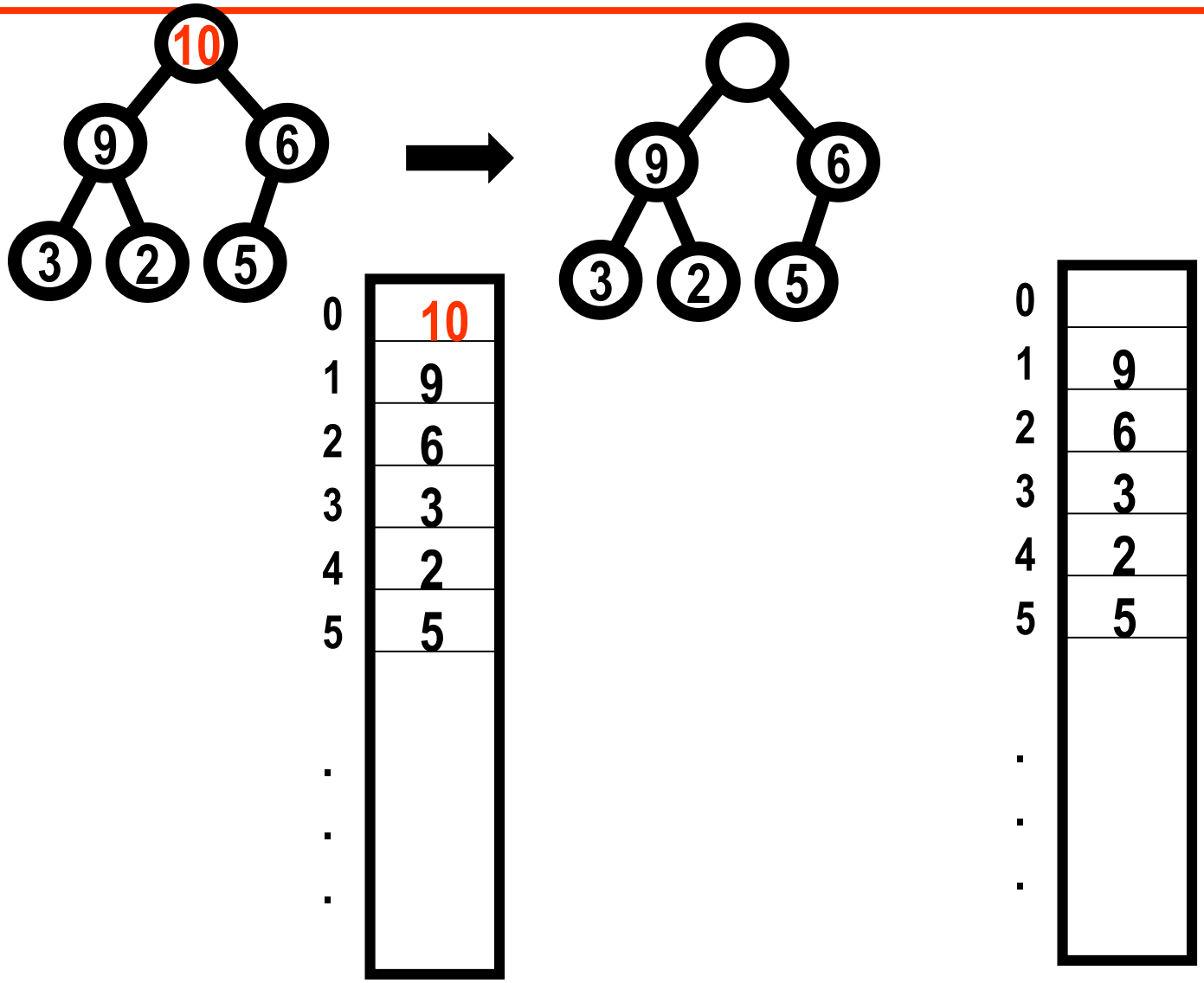
Heapify-Down

- Heapify-Down (Heap, root, bottom):
  - The heap order may be violated only at the root node.
  - Restores the heap order between the root and the bottom.
  - If root is not a leaf then
    - Set BiggerChild to the index of child with larger value.
    - If heap[root] < heap[BiggerChild] then
      - Swap heap[root] with heap[BiggerChild];
      - Heapify-Down (BiggerChild, bottom);

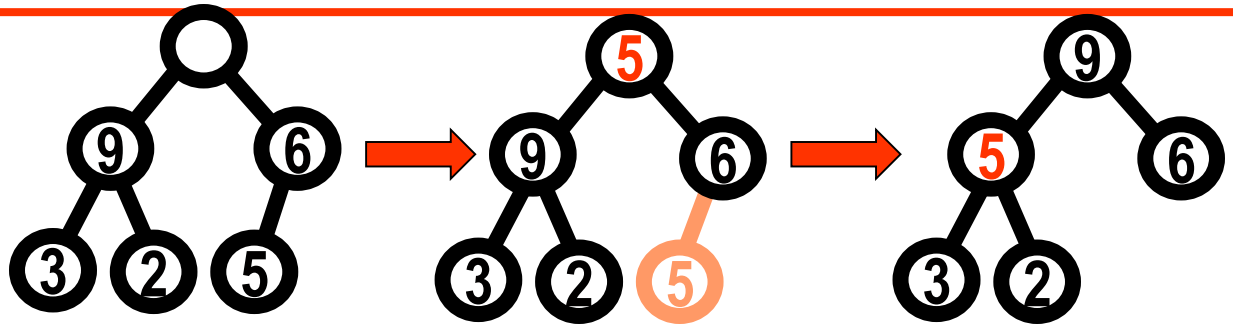
Retrieve & Delete a heap's root item

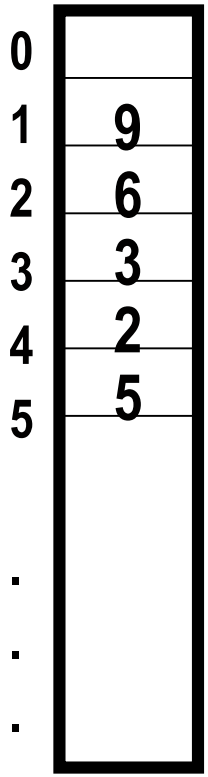
- DeleteTheRoot (Heap):
  - Delete the root value;
  - Move the last leaf value into the root position;
  - Decrement the size of the heap;
  - Heapify-Down (Heap, 0, size-1);

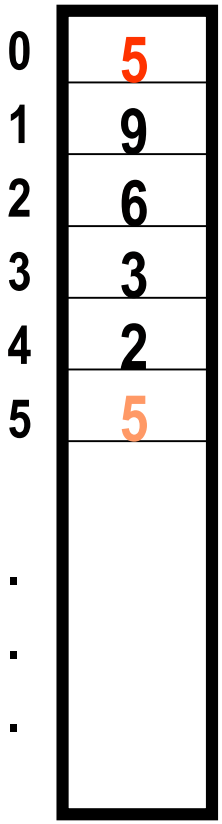
Example: Delete the root



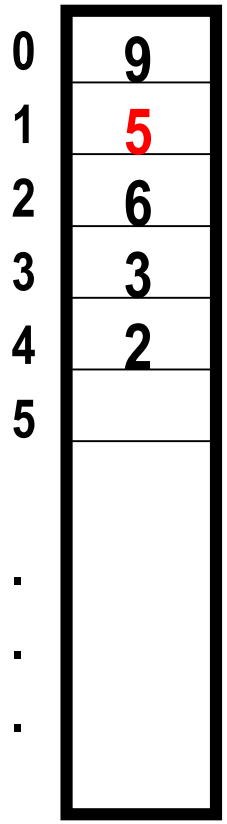
Example: Delete the root







Heapify-Down



Implementation of Binary Heap

```
template < class DT >
class BHeap
{
  public:
    BHeap ( int maxNumber = defMaxHeapSize );
    ~BHeap ();
    void insert ( const DT &newElement );    // Insert element
    DT removeMax ();  // Remove max pty element

private:
    int maxSize,   // Maximum number of elements in the heap
        size;      // Actual number of elements in the heap
   DT *dataItems; // Array containing the heap elements
};
```

Binary Heap Operations - Analysis

- The height of a complete binary tree with N nodes is
  - Shortest height!
  - **O(log N)**
- Worst-case running time: All **O(log n)**
  - **Heapify-Up**
    - **O(log N)**
  - **Insert**
    - **O(log N)**
  - **Heapify-Down**
    - **O(log N)**
  - **DeletetheRoot**
    - **O(log N)**

D-ary (D-way) Heaps (d-Heaps)

d-Heaps

- A **complete d-ary (d-way) tree** with heap ordering
  - ➔ $O(\log_d n)$
- A binary heap is a d-heap where d=2!
- When d=3: **Ternary heap**
  - ➔ We can implement **ternary heap as an array** just like the binary heap.
  - ➔ 3i+1, 3i+2, 3i+3, (i-1)/3
  - ➔ Worst-case running time: All **O(log n)**

►QUIZ?

- Insert 5, 4, 3, 2 & 1 into an empty ternary min-heap?

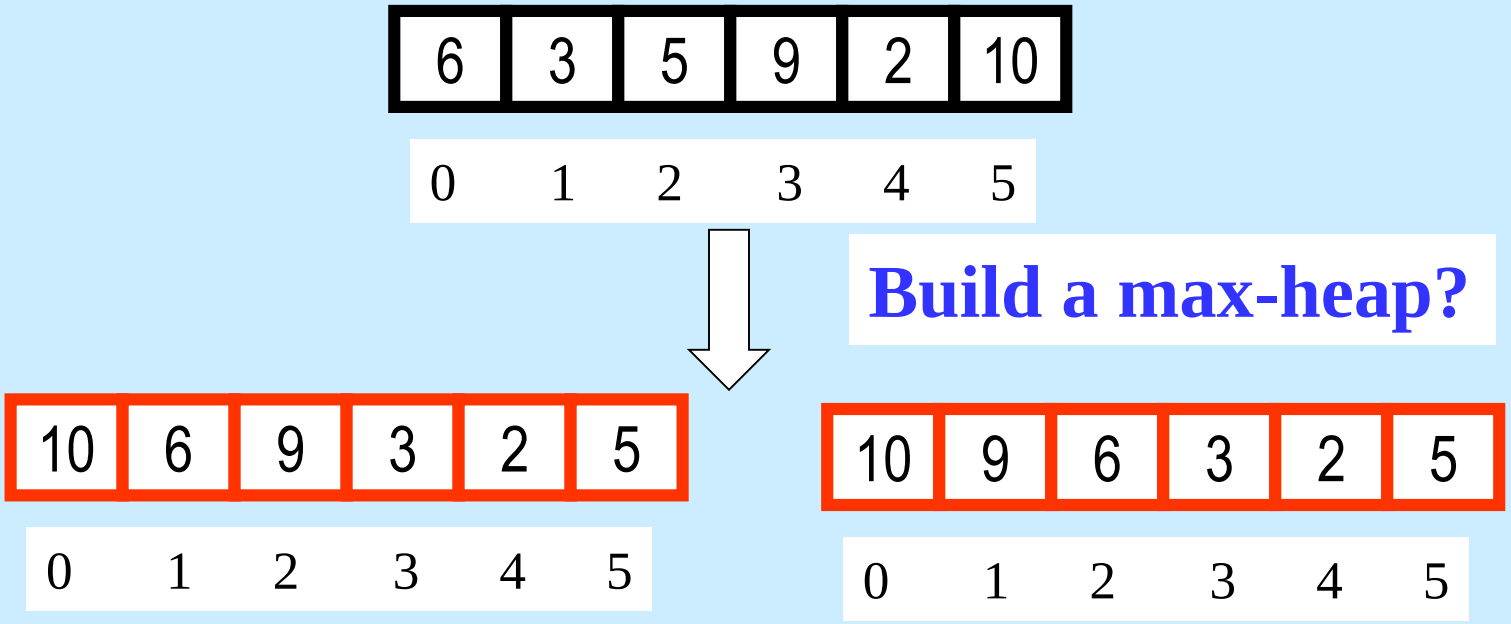
►QUIZ?

- An array-based representation of Ternary Heap (3-Heap)?
  - ➔3i+1, 3i+2, 3i+3, (i-1)/3

Transform an Array To a Binary (Max- or Min-) Heap

Build a Binary Max-Heap

- Convert an array into a max-heap?



- Idea?

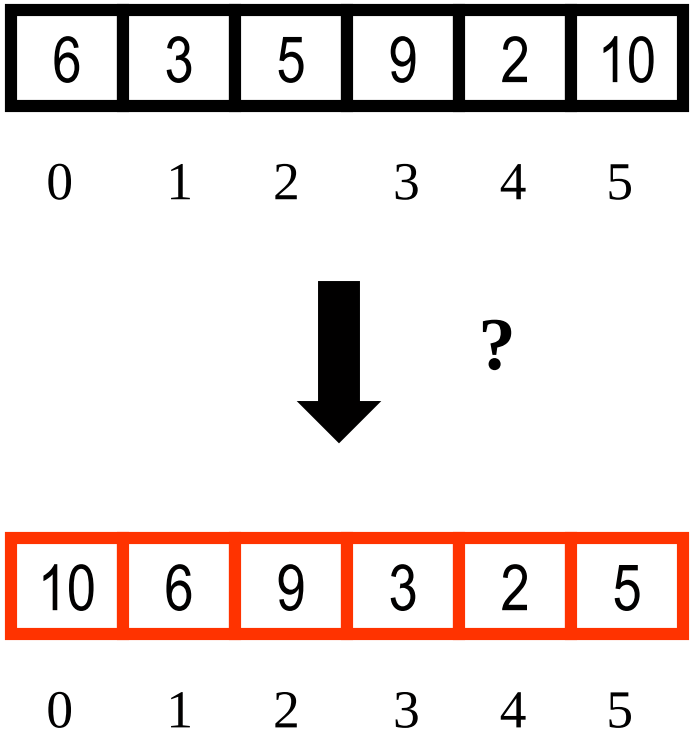
Building a Binary Max-Heap

- **Approach 1:**
  - Using **Heapify-Up**
  - **Top-down heap building**
- **Approach 2:**
  - Using **Heapify-Down**
  - **Bottom-up heap building**

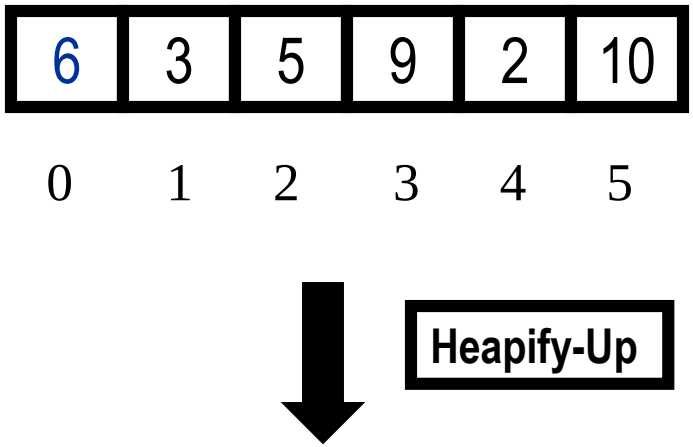
1. Building a Binary Max-Heap - Approach 1

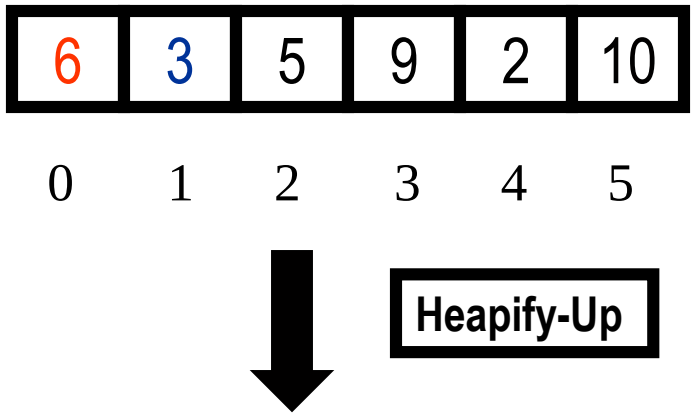
- **Approach 1:**
  - ➔ Using Heapify-Up
  - ➔ **Top-down heap building**

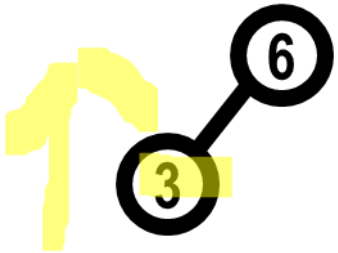
Example: Transform an Array into a Binary Max-Heap - Approach 1



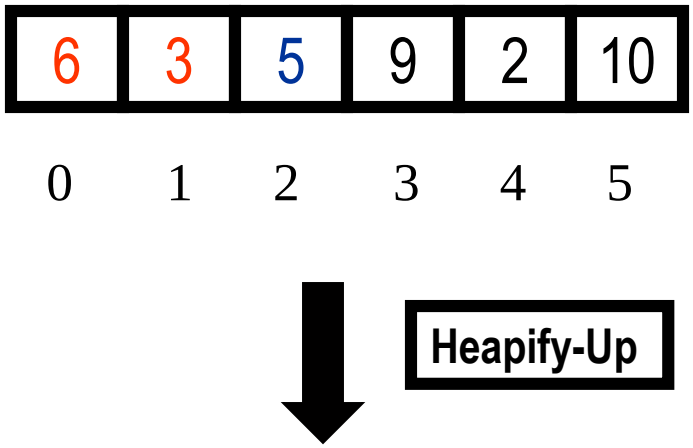
Example: Approach 1

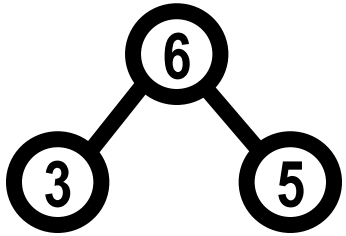


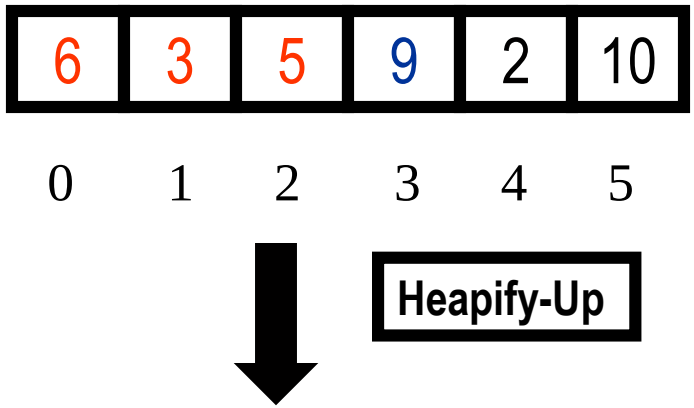


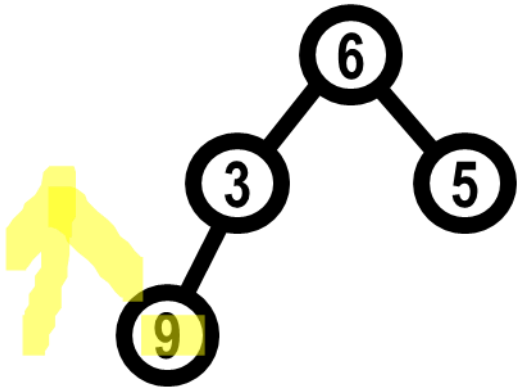


Example: Approach 1

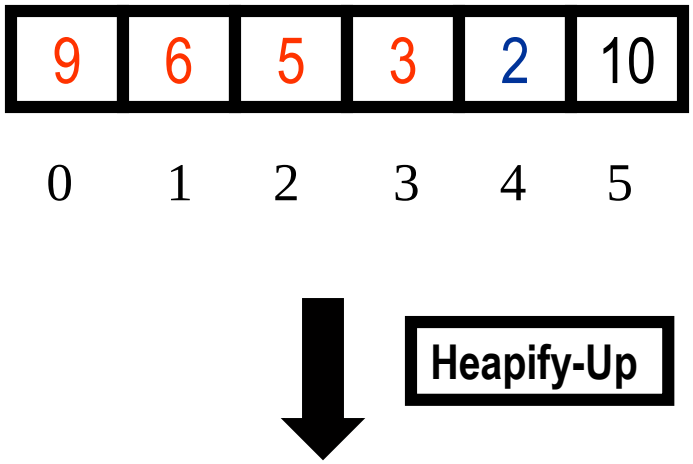


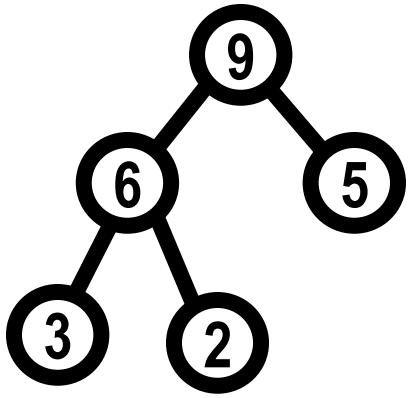


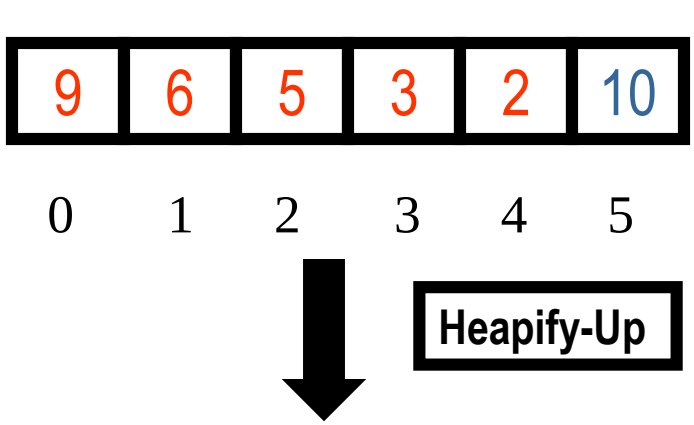


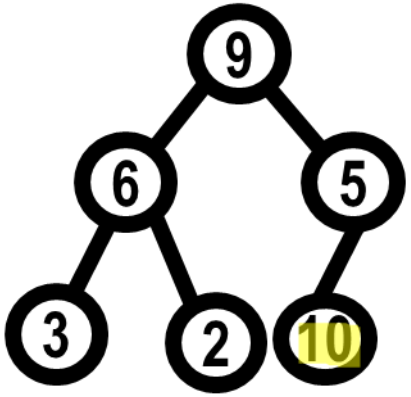


Example: Approach 1

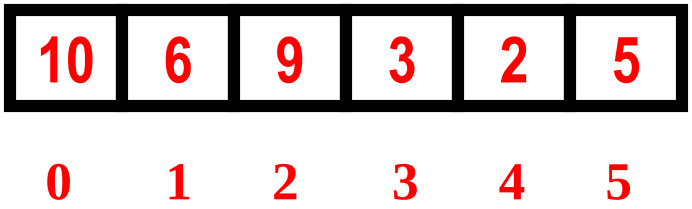


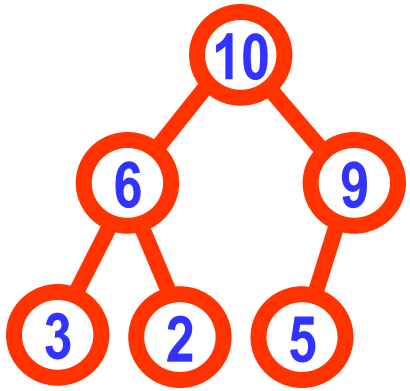




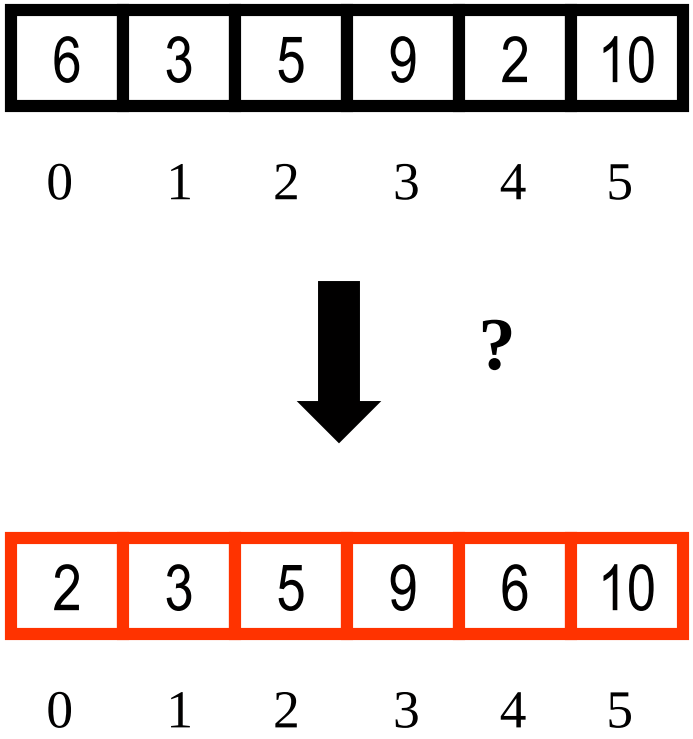


Example: Approach 1





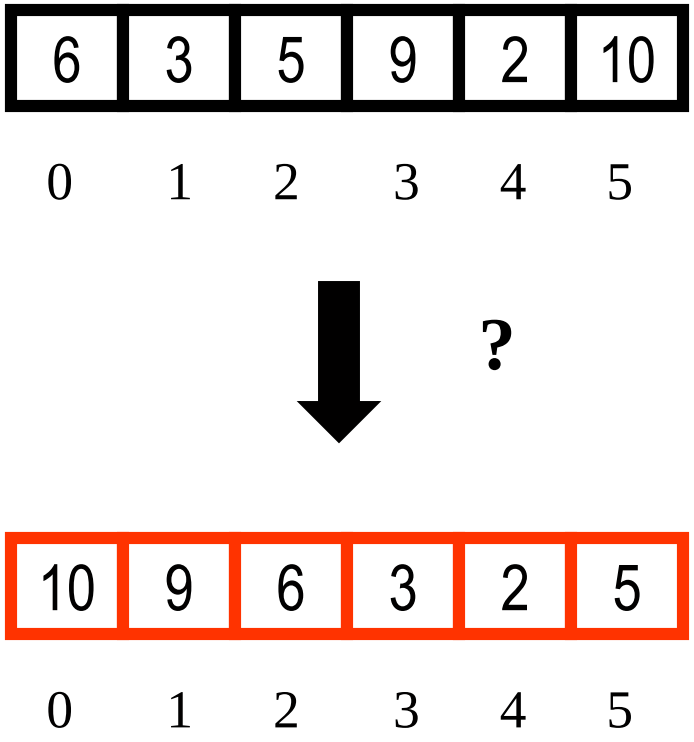
►QUIZ? Into a Binary Min-Heap



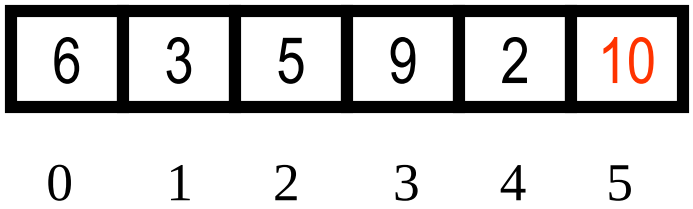
2. Building a Binary Heap- Approach 2

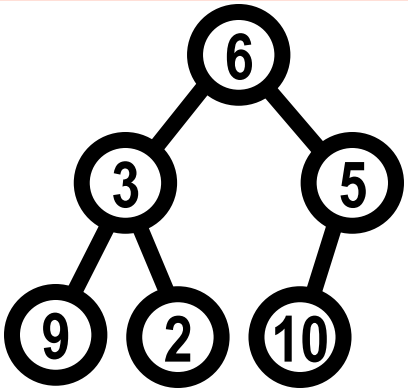
- **Approach 2:**
  - ➔Using Heapify-Down
  - ➔**Bottom-up heap building**

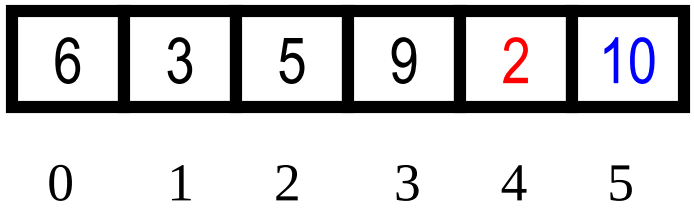
Example: Transform an Array into a Binary Max-Heap - Approach 2

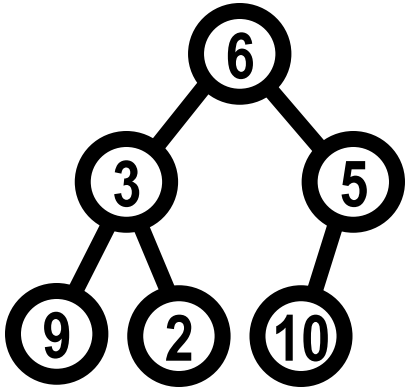


Example: Transform an Array into a Binary Max-Heap - Approach 2

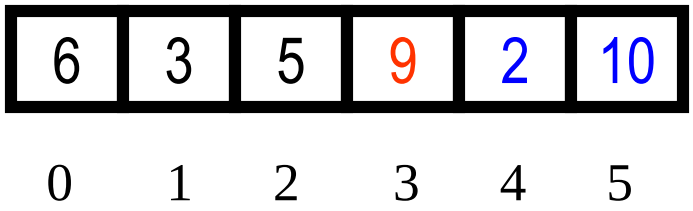


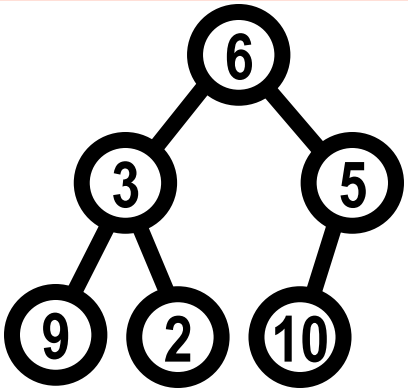


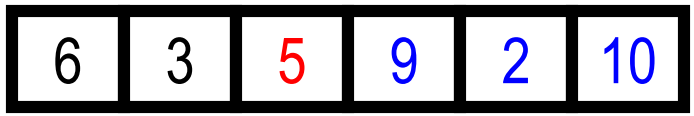


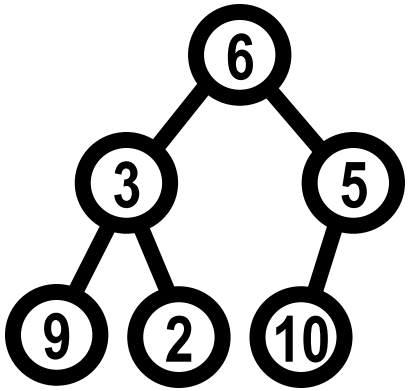


Example: Approach 2

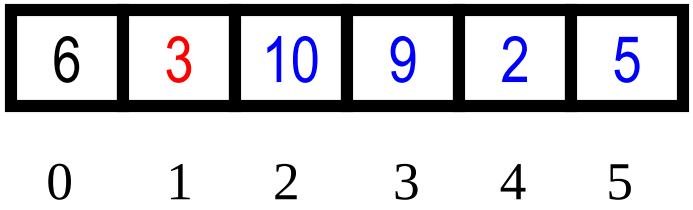


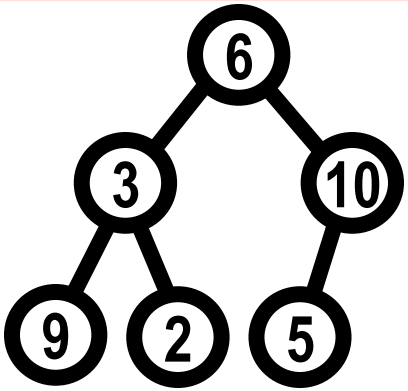


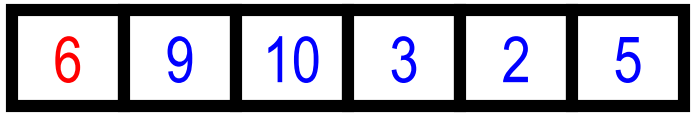


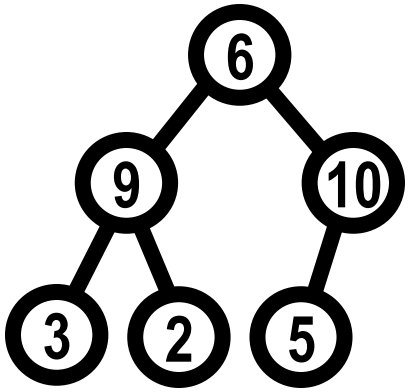


Example: Approach 2

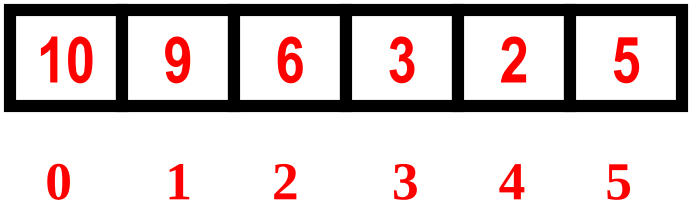


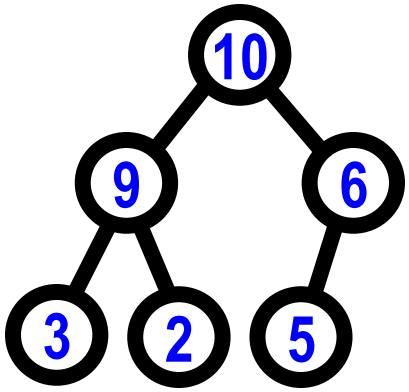






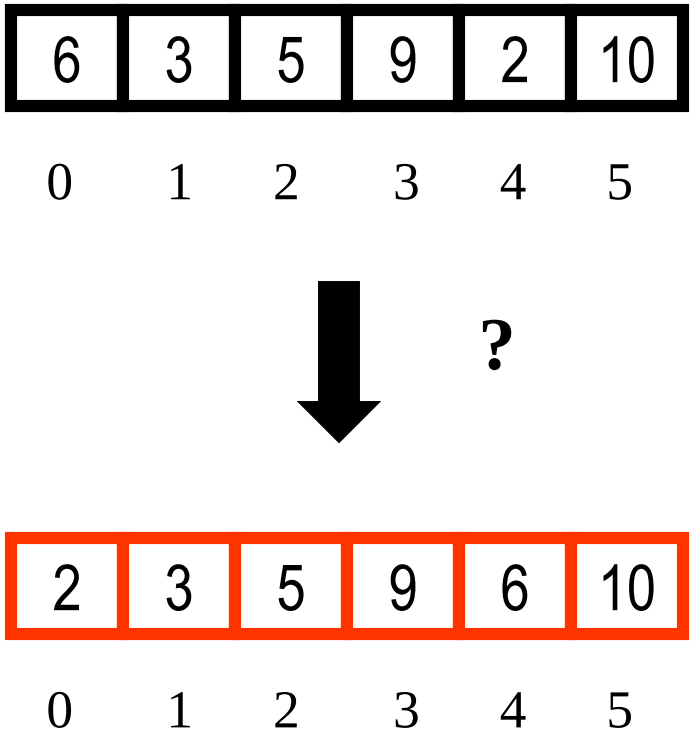
Example: Approach 2





Heap!

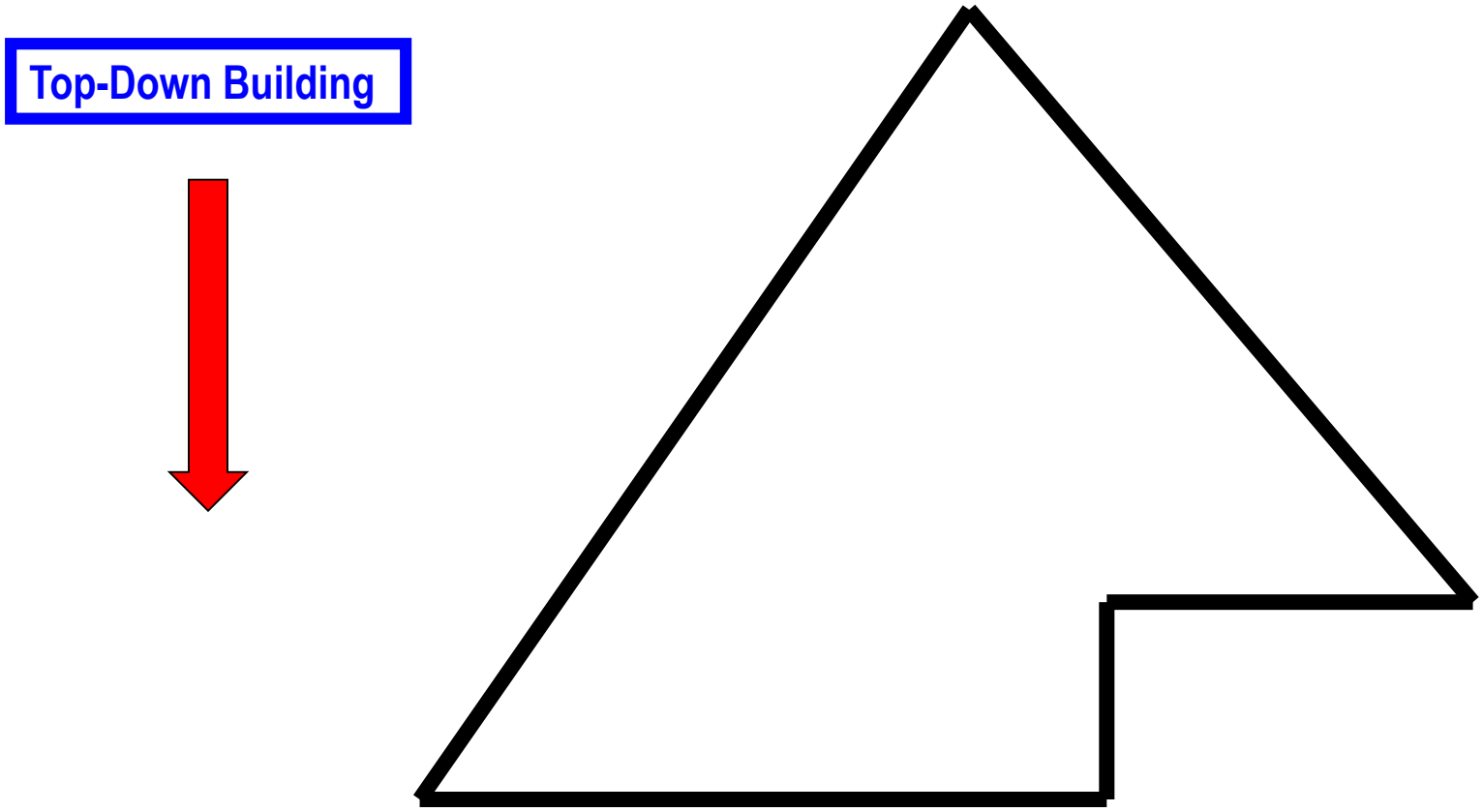
►QUIZ? Into a Binary Min-Heap



Building a Binary Max-Heap - Analysis

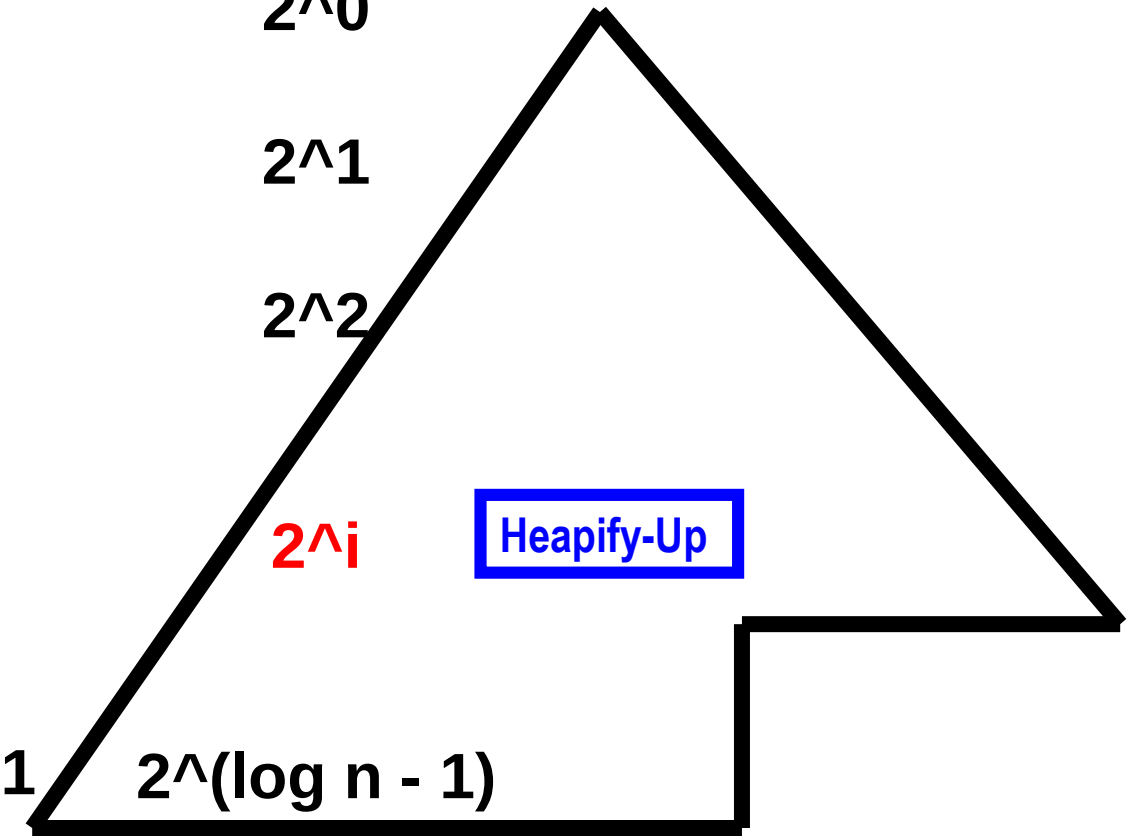
- **Approach 1:**
  - Using Heapify-Up
  - **Top-down heap building**
- **Approach 2:**
  - Using Heapify-Down
  - **Bottom-up heap building**

Top-Down Building a Binary Heap - Analysis



Top-Down Building a Binary Heap - Analysis

Depth	#Nodes	#Compare
0	2^0	0
1	2^1	1
2	2^2	2
i	2^i	i
log n - 1	2^(log n - 1)	log n - 1



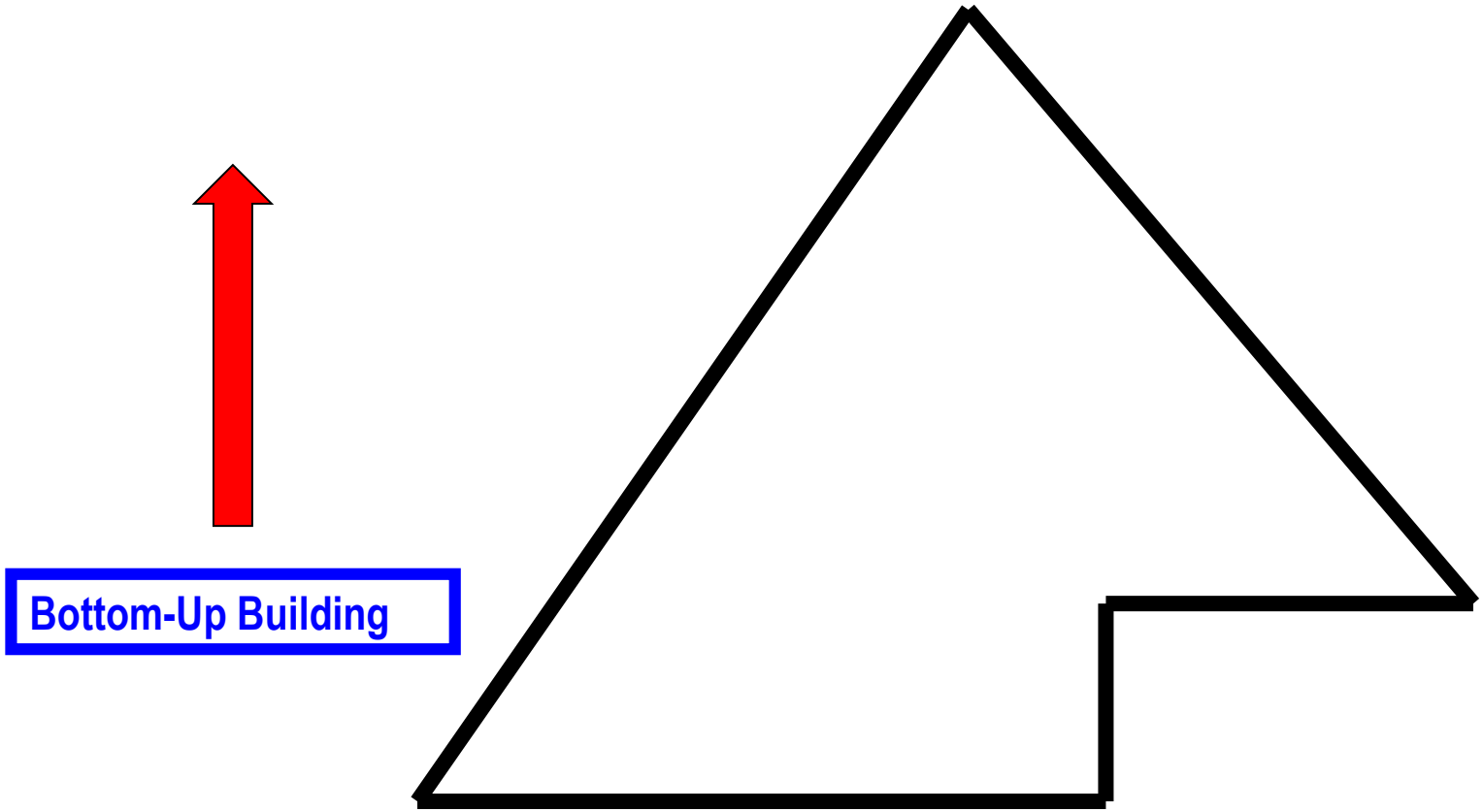
Build a Binary Heap - Analysis

- **Top-down Approach 1:**
  - N Heapify-Up
  - O(log n) for Heapify-Up
  - **$T(n) = \sum_{i=0,\log n - 1} (i * 2^i)$** **Example A.5**
  - Worst-case running time for heap building:
    - **O(N log N)**

▶QUIZ?

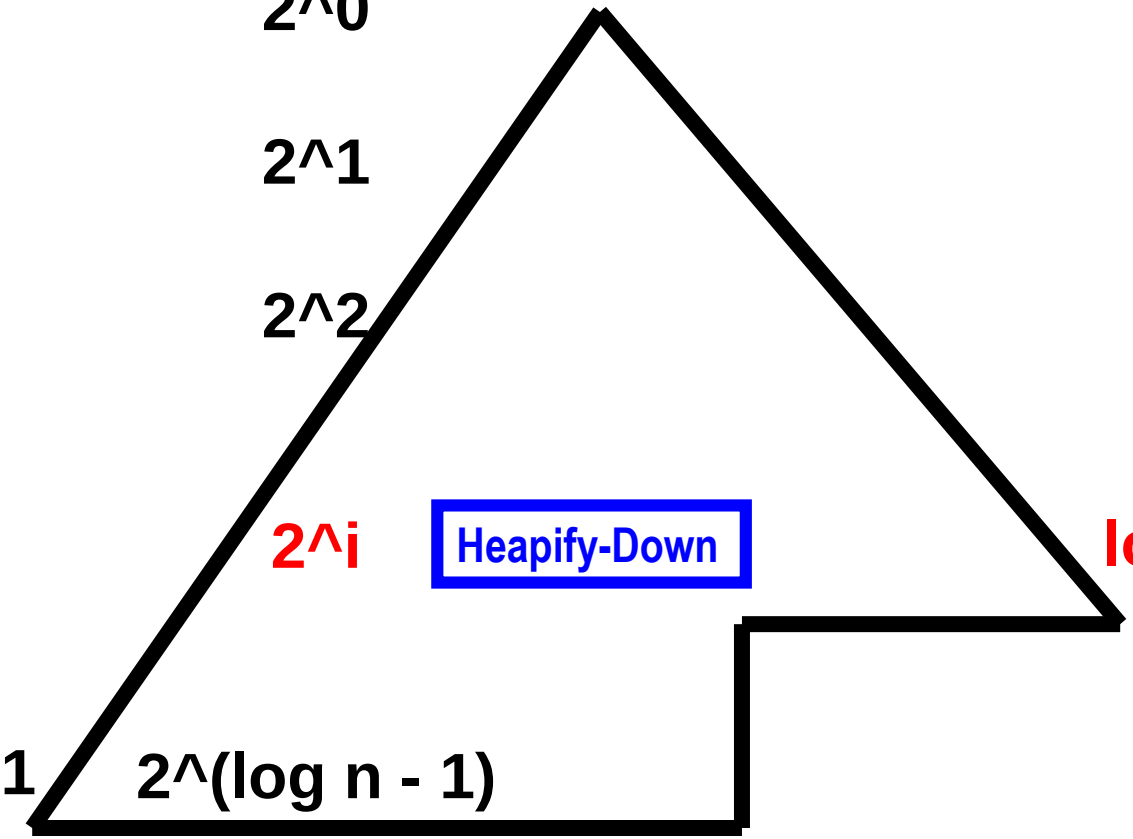
- Worst-case running time for top-down heap building?

Bottom-Up Building a Binary Heap - Analysis



Bottom-Up Building a Binary Heap - Analysis

Depth	#Nodes	#Compare
0	2^0	log n - 1
1	2^1	log n - 2
2	2^2	log n - 3
i	2^i	log n - 1 - i
log n - 1	2^(log n - 1)	0



Build a Binary Heap - Analysis

- **Bottom-up Approach 2:**
  - N Heapify-Down
  - O(log n) for HeapifyDown

Example A.3

Example A.5

  - $T(n) = \sum_{i=0,\log n - 1} (\log n - 1 - i) * 2^i)$

☞ **O(N) linear time!**

▶QUIZ?

- Worst-case running time for bottom-up heap building?

Binary Heap Visualization

- ***Binary Heap Visualization***

►QUIZ?

- Compare Top-down heap building with Bottom-up heap building?

Advanced Heaps

Binary Heaps

- **Basic operations** on Binary Heaps:
  - ➔ Insert (add) a new item to the binary heap?
    - ☞ **O(log N) time at worst-case**
  - ➔ Retrieve and then delete a binary heap's root item?
    - ☞ **O(log N) time at worst-case**

Merging Binary Heaps

- How about **merging** two Binary Heaps?
  - **Merge (join, meld, union)** two binary heaps into one binary heap?
  - How?

Merging Binary Heaps

- How? Merge(BH1, BH2)?
  - ➔ Approach 1:
  - ✓ **Insert each element of the smaller heap into the larger.**
    - ✓ **O(N log N) worst-case time**
  - ➔ Approach 2:
  - ✓ **Concatenate binary heaps' arrays and run BuildHeap.**
    - ✓ **O(N) worst-case time**
- Can we do better?

Why Advanced Heaps?

- Why?
  - **Merge (join, meld)** two heaps into one heap.
  - **A better merge/join/meld operation than O(n)?**
    - **O(log N) worst time & amortized time?**

- SO, **all three heap operations** are O(log N) time at worst-case & amortized case?
  - **Mergable Heaps!**

What Advanced Heaps?

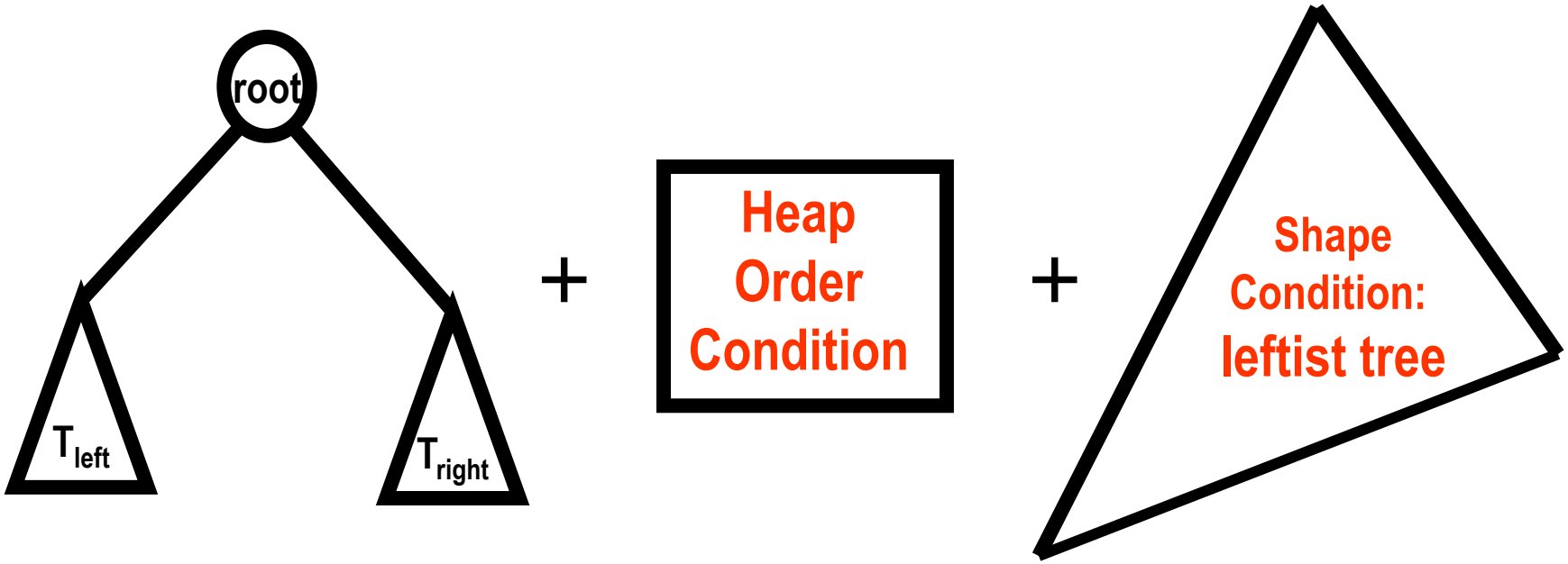
- How?
  - O(log N) Merge ***worst time***?
    - **Leftist Heaps**
    - **Binomial Heaps**
    - **Fibonacci Heaps**
  - O(log N) Merge ***amortized time***?
    - **Skew Heaps – Self-Adjusting Heaps!**

Leftist Heaps

What Is A Leftist Heap?

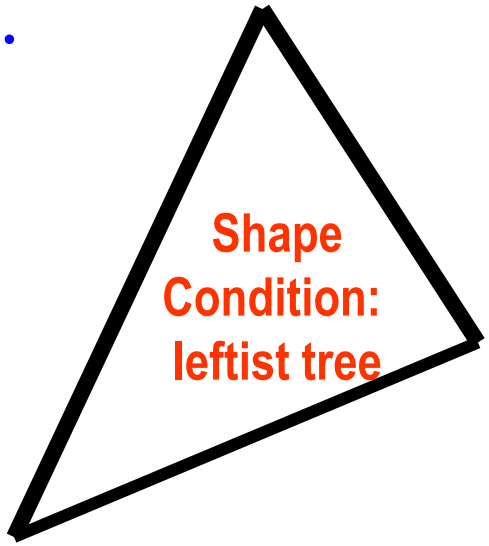
- A binary tree that satisfies two conditions:
  - The (partial) order condition between every node and the nodes in its left and right subtrees.
    - Heap order property
  - The shape condition
    - A special shape called a **leftist tree** for fast merge.

A Leftist Heap



What is a Leftist Tree?

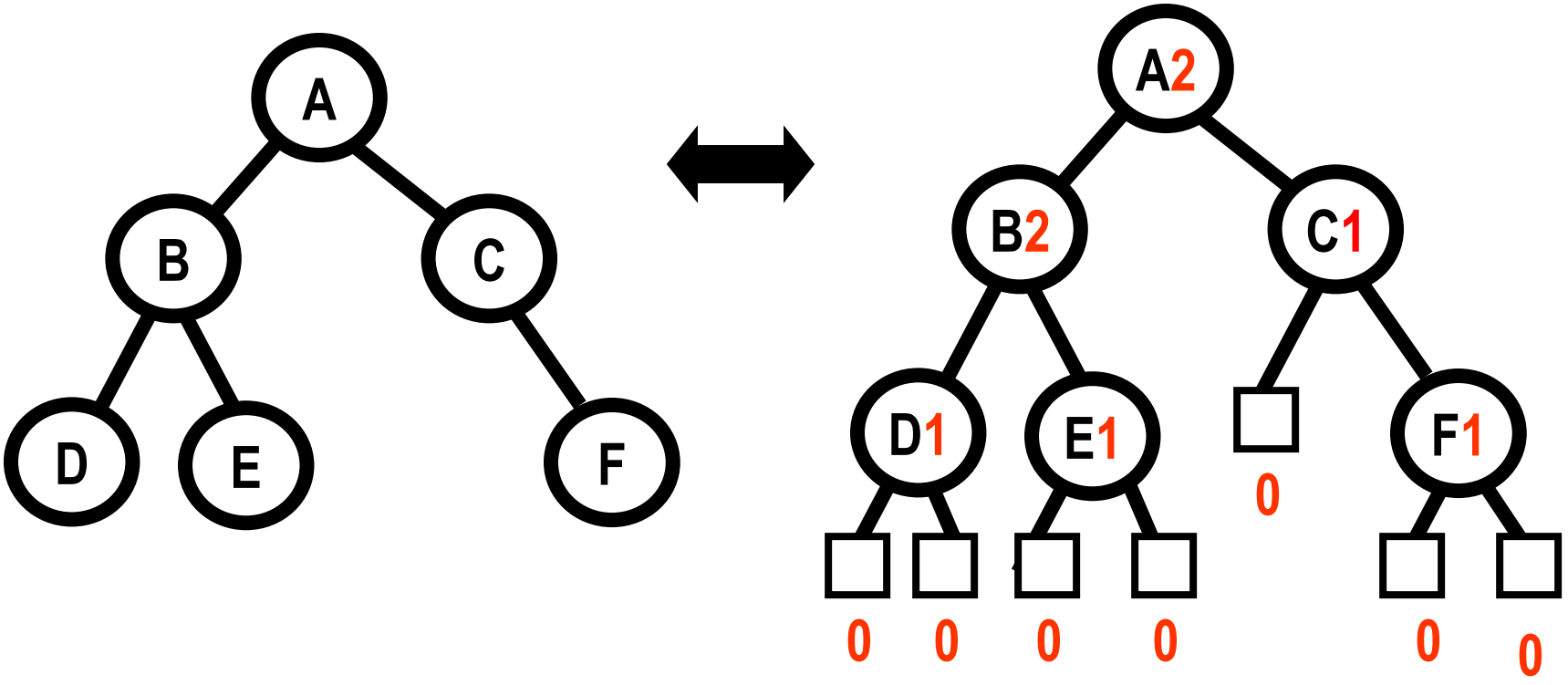
- A binary tree which tends to “lean” to the left.
- The tendency to lean to the left is defined in terms of
  - ➔The **shortest path** from an internal to an external node.



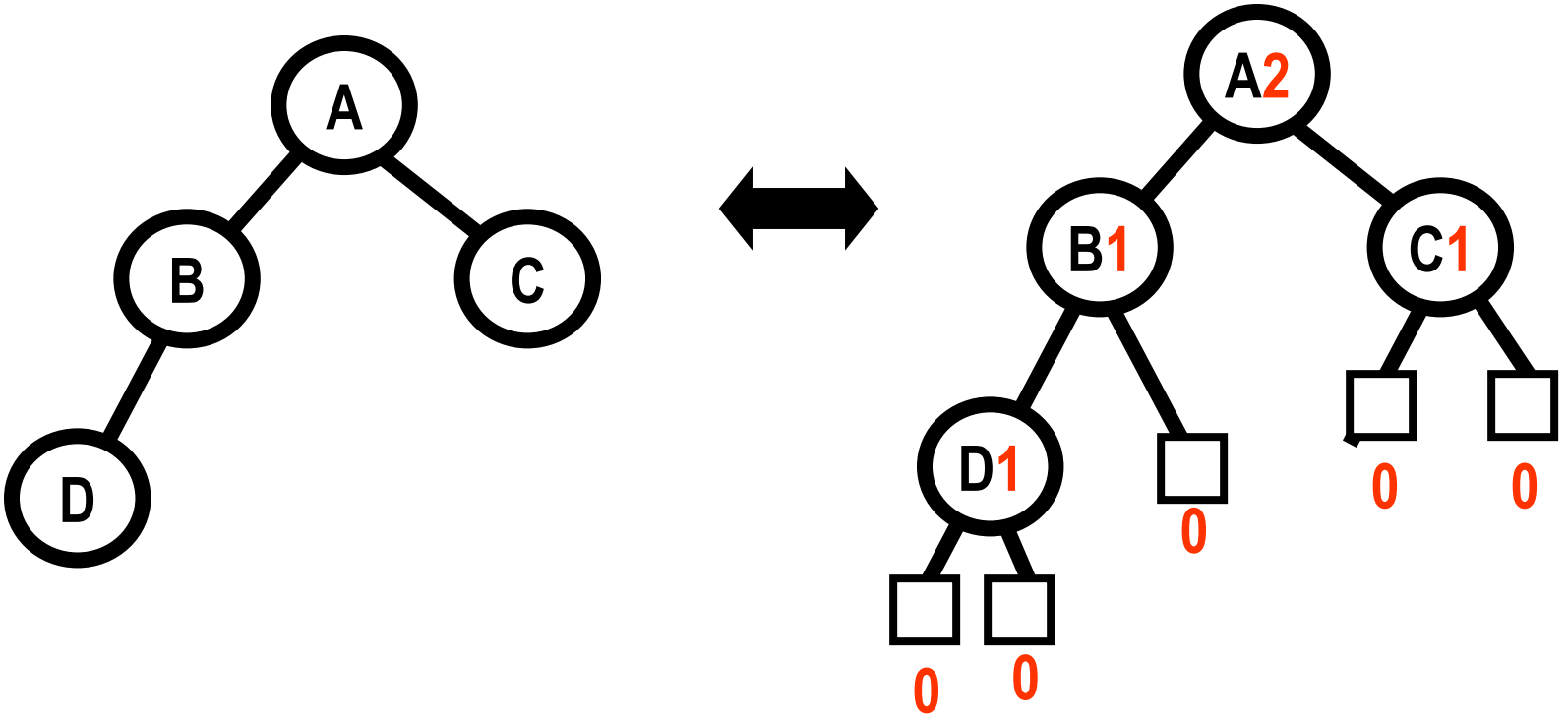
The Shortest Path in A Leftist Tree

- For a node x: **s**hortest **p**ath **l**ength (**spl**)
  - **spl(x) = The shortest path length from the node x to an external node.**
- **spl(x) =**
  - **0** if x is an external node
  - **1 + min (spl(LeftChild of x), spl(RightChild of x))** if x is an internal node

Example: The Shortest Path Length



►QUIZ? The Shortest Path Length



►QUIZ? Shortest Path vs. Height

For a node x:

- spl (x) = **?**
- height(x) = **?**

►QUIZ: Shortest Path vs. Height

- spl(x) =
  - ➔0 if x is an external node
  - ➔1 + min (spl(LeftChild of x), spl(RightChild of x)) if x is an internal node
- height(x) = longest path length
  - ➔0 if x is an external node
  - ➔1 + max (height(LeftChild of x), height(RightChild of x)) if x is an internal node

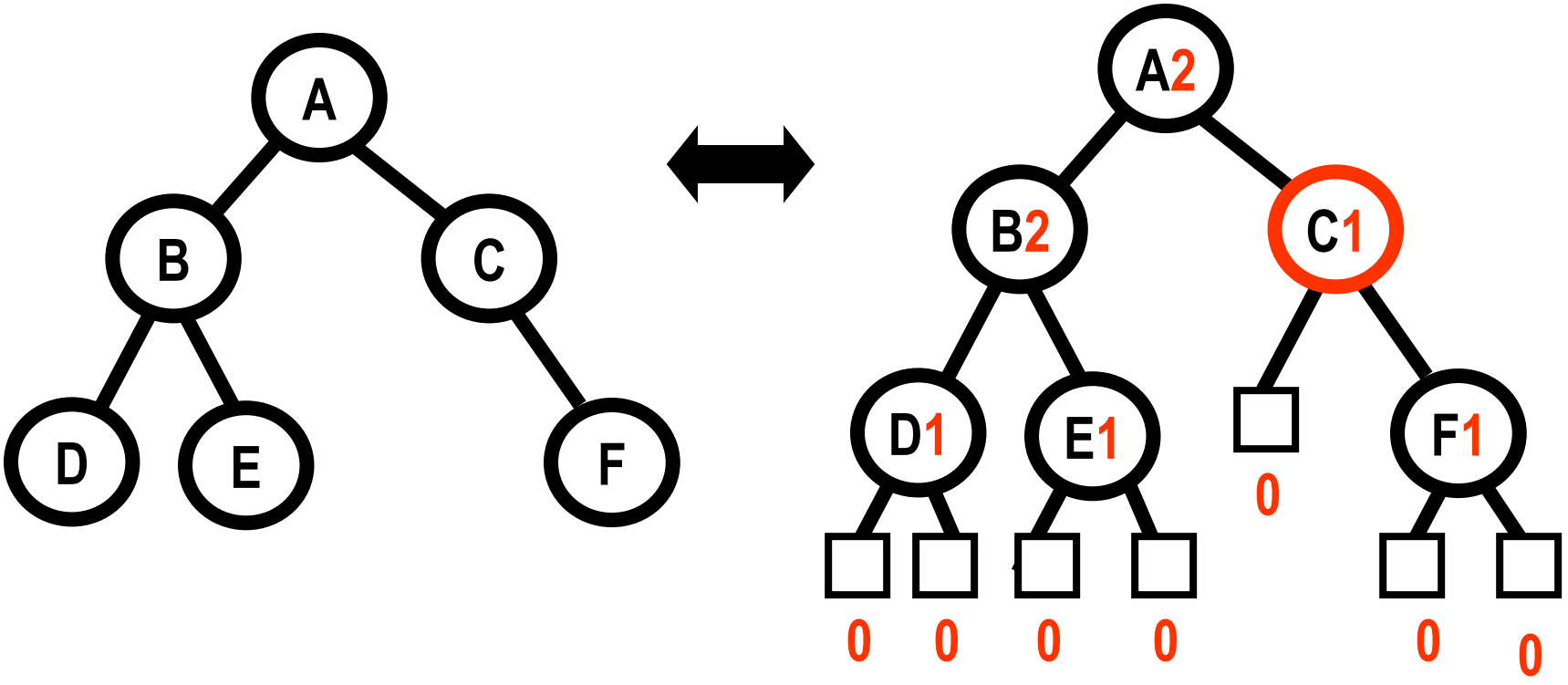
Leftist Tree Condition

- A leftist tree is a binary tree s.t.
  - ➔Either an empty tree
  - ➔Or for every internal node x,

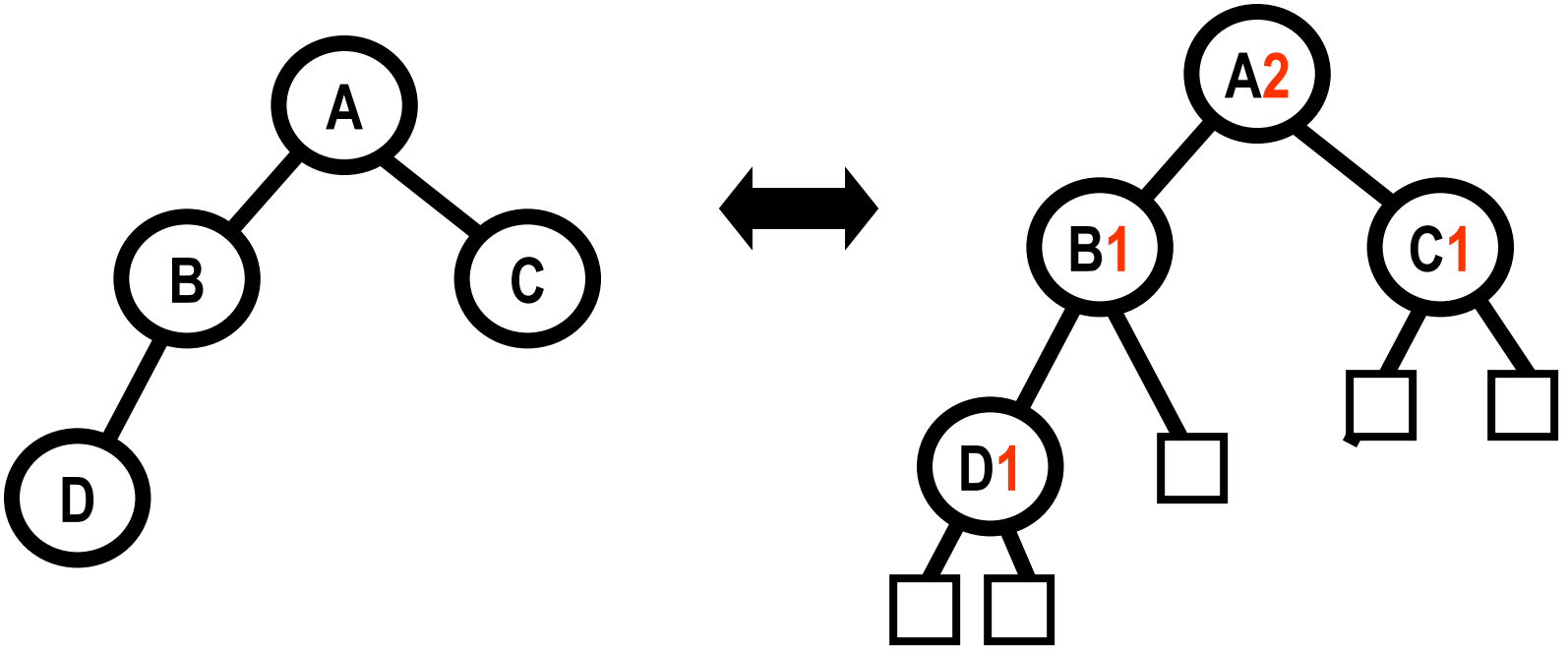
☞SPL(LeftChild of x) ≥ SPL(RightChild of x)

- **Leftist tree is at least as "heavy" on the left as the right!**
- **Unbalanced!**

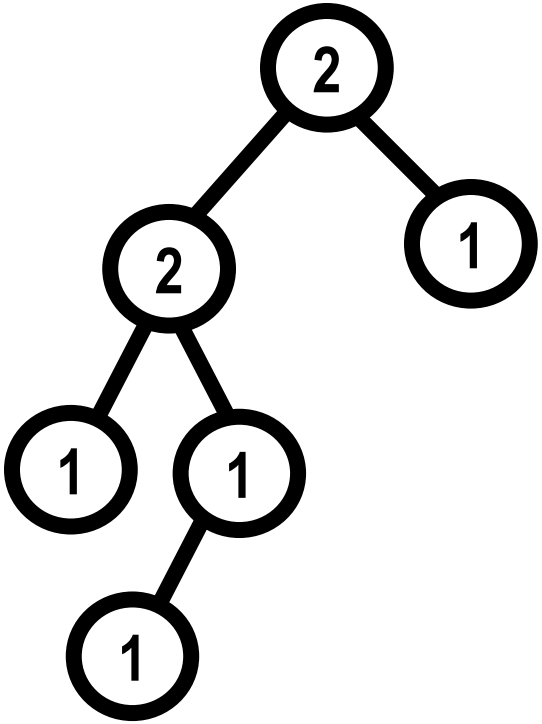
Example: Leftist Tree?

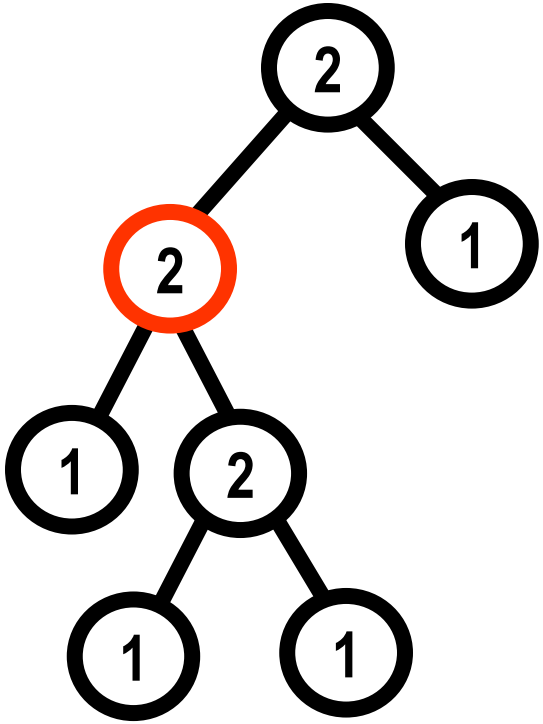


Example: Leftist Tree?

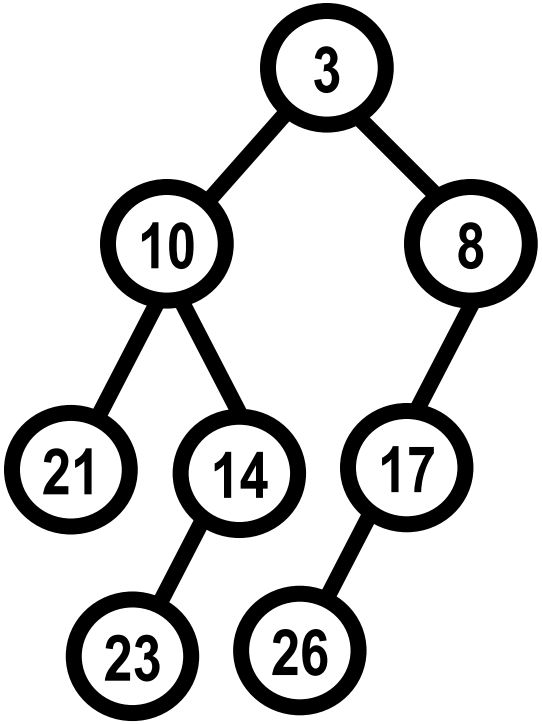


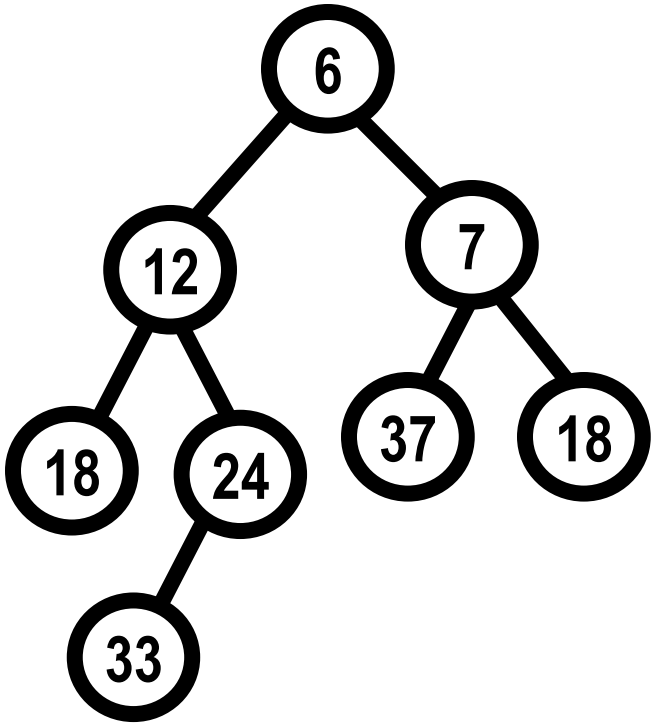
▶QUIZ?



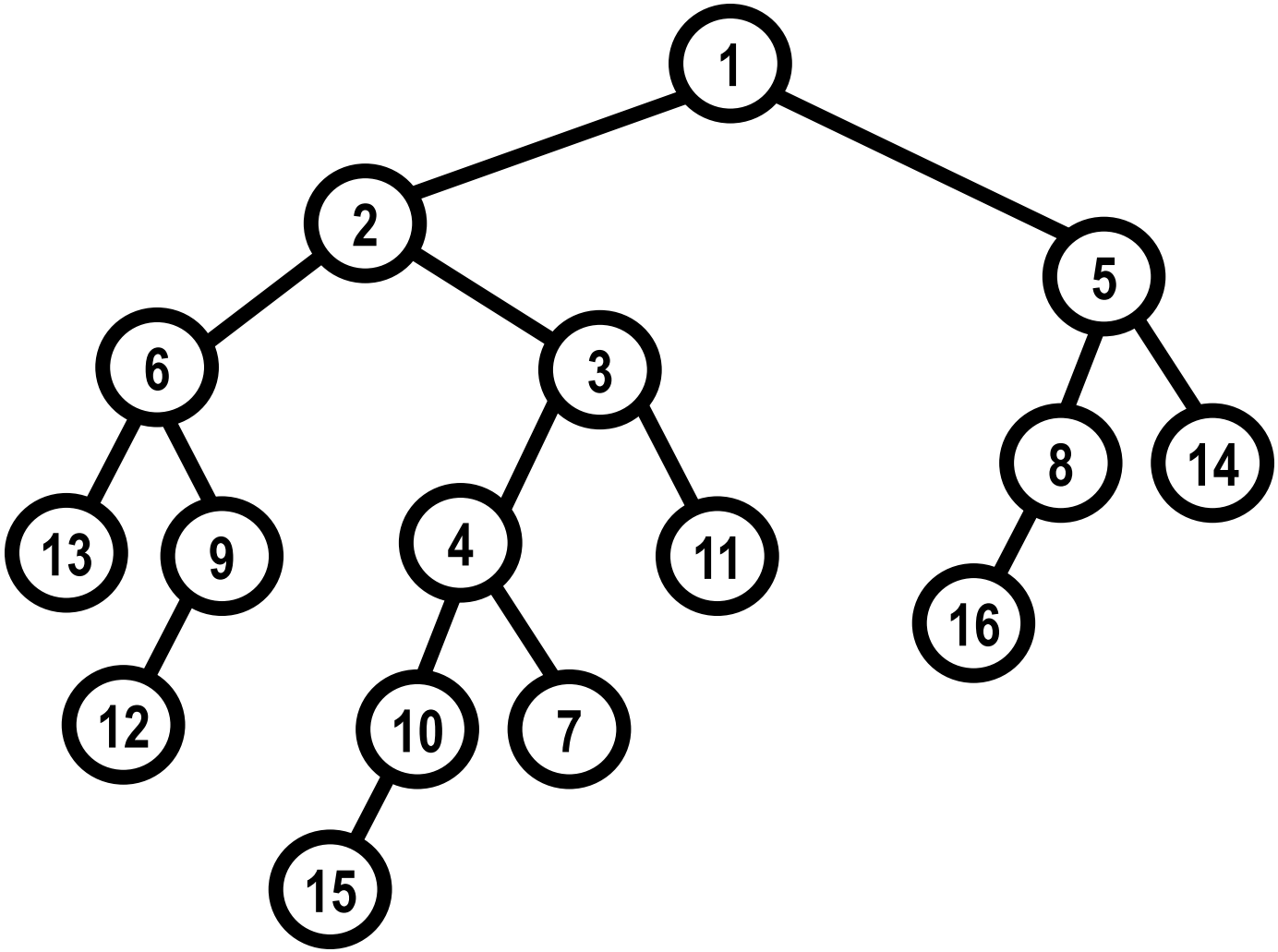


▶QUIZ?





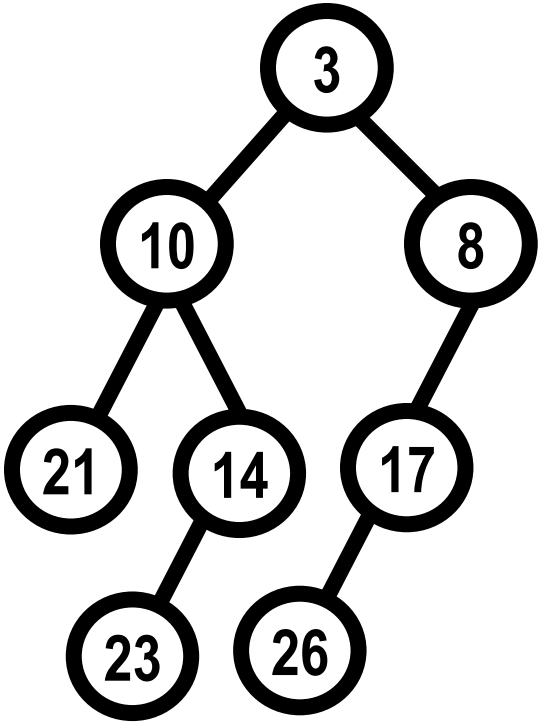
▶QUIZ?

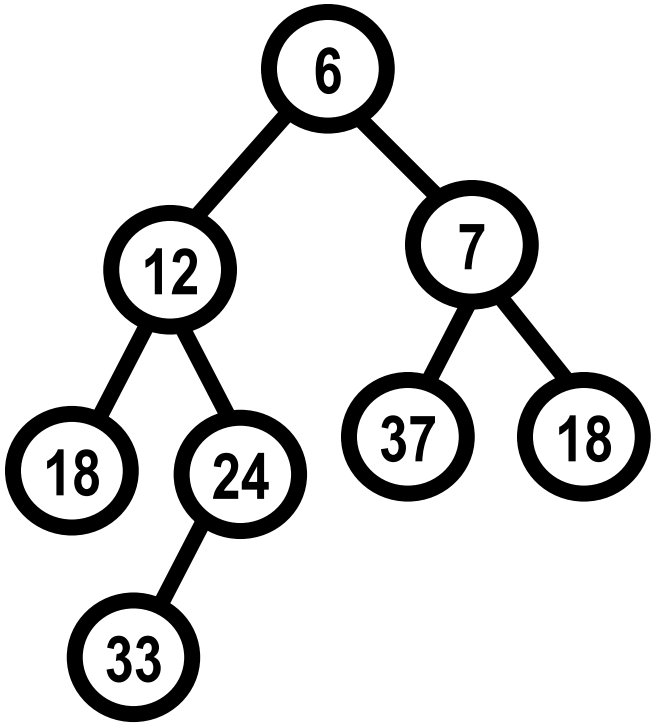


Leftist Heaps

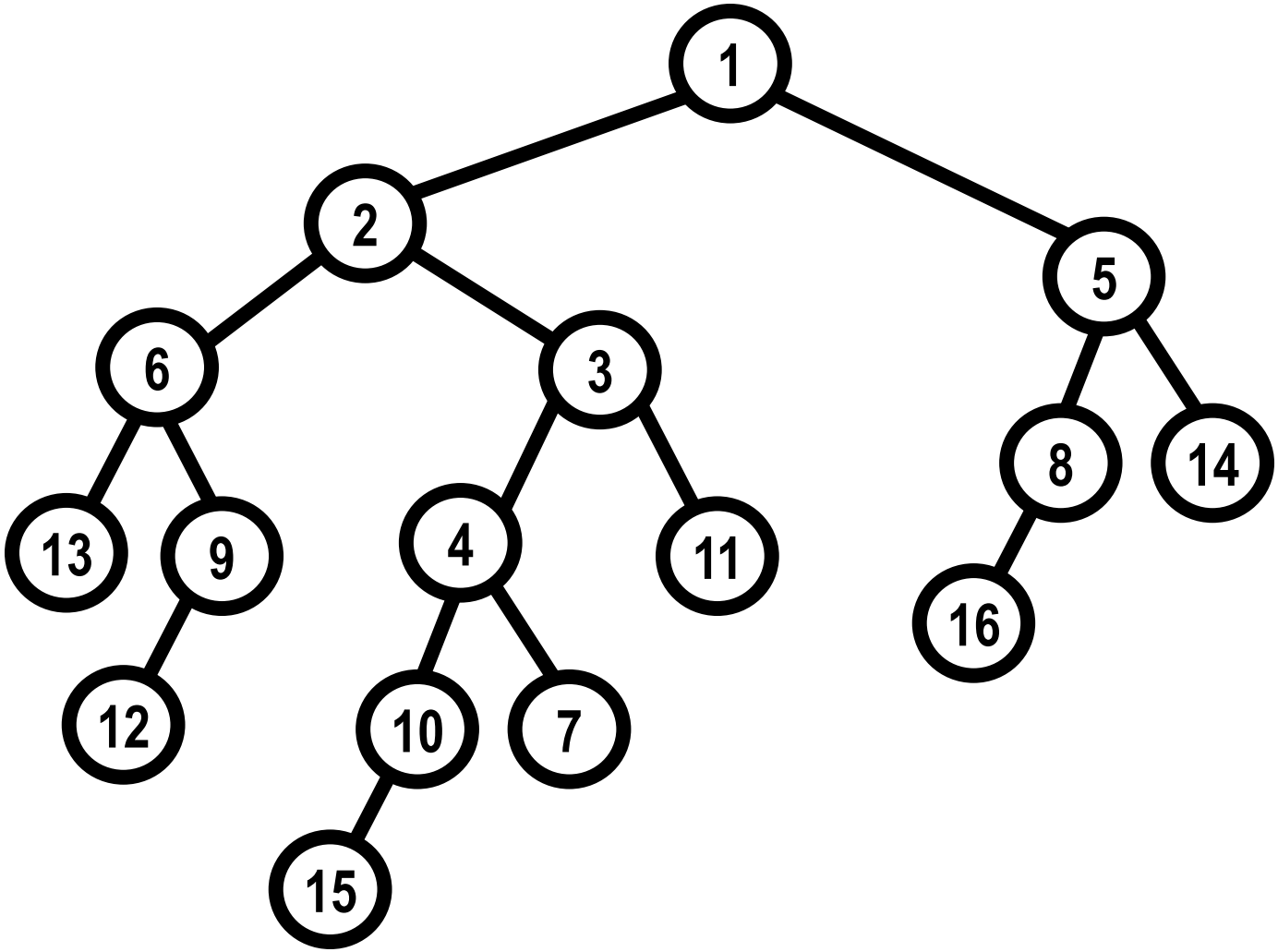
- Leftist min-heaps
- Leftist max-heaps

►QUIZ? Leftist Min-Heap?





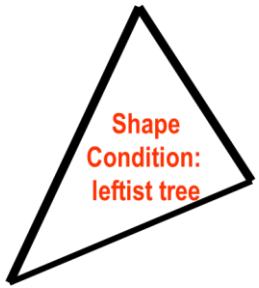
►QUIZ? Leftist Min-Heap?



Paths in a Leftist Heap

- Observation!
  - ➔Leftist heaps tend to have deep left paths.
  - ➔The right path of a leftist heap should be short.

- **The shortest path from the root to an external node is always found on the right!**



Properties of Leftist Trees

- What is the minimum number of nodes that a leftist tree whose **right path length** is h can have?
  - ➔ **$2^h - 1$**

Properties of Leftist Trees

- N= The number of nodes in a leftist tree.
- h = The length of the **right path** of a leftist tree.

➔ $2^h - 1 \leq N$

➔ $h \leq \log (N+1)$

➔ **Upper Bound: h = O (log N)**

The Right Path of Leftist Heap is Short

- In a leftist heap with N nodes, **the shortest path from the root to an external node is always found on the right and has**
  - ➔ **O(log N) nodes!**

Operations on Leftist Heaps

- **Insert (add)** a new item to the leftist heap.
- **Retrieve** and then **delete a leftist heap's root item**.
- **Join (merge, meld) two leftist heaps**

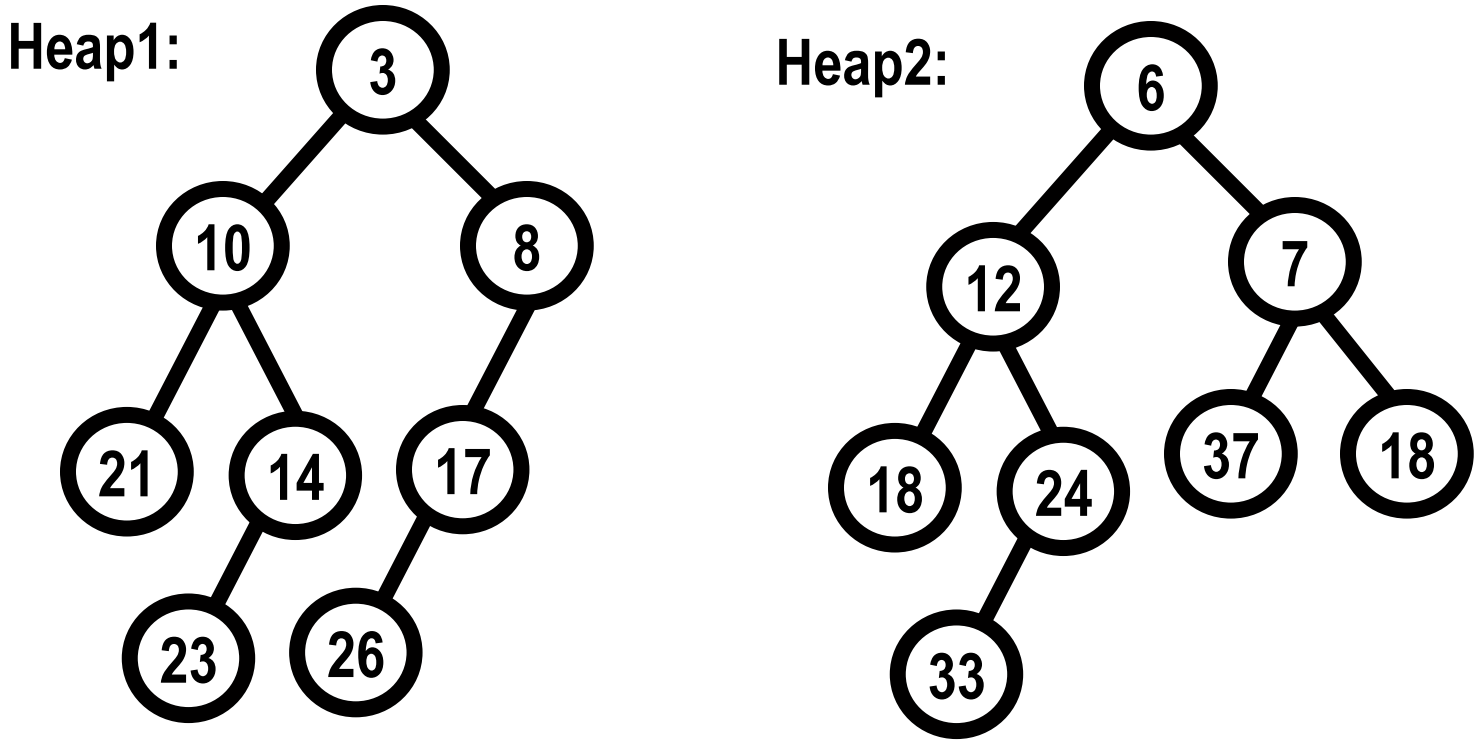
General Idea

- In a leftist tree with N nodes,
  - When a new node or subtree is attached to a given tree, it is always attached in place of an external node.
  - The shortest path from the root to an external node gives a lower bound on the cost of insertion.
- **Idea?**
  - **Perform all the work on the right path that is guaranteed to be shortest!**

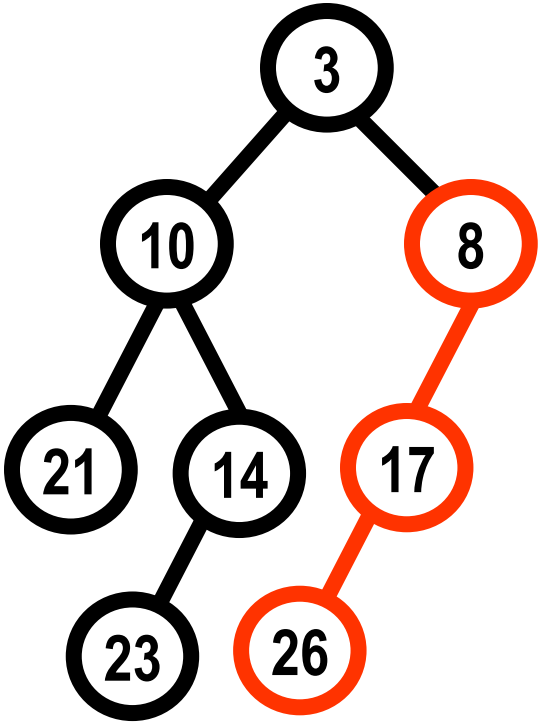
Merge Two Leftist-heaps

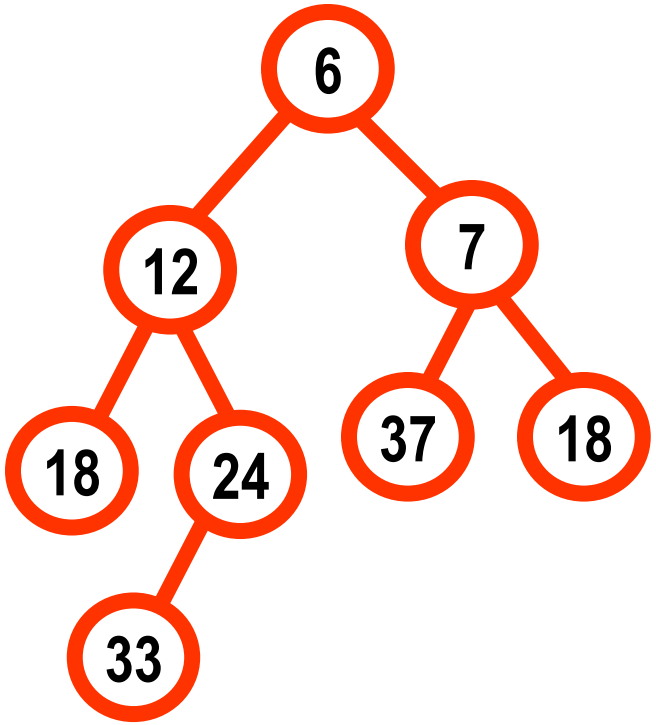
- **Merge**: The fundamental operation!
- How?
  - Put the root with smaller value as the new root.
  - **Recursively merge its right subtree and the other tree.**
  - Before returning from recursion:
    - **Swap** left and right subtrees just below root, **if needed**, **to keep leftist property of merged result**.
    - **Update spl** of merged root.

Example: Merging Two Leftist Heaps

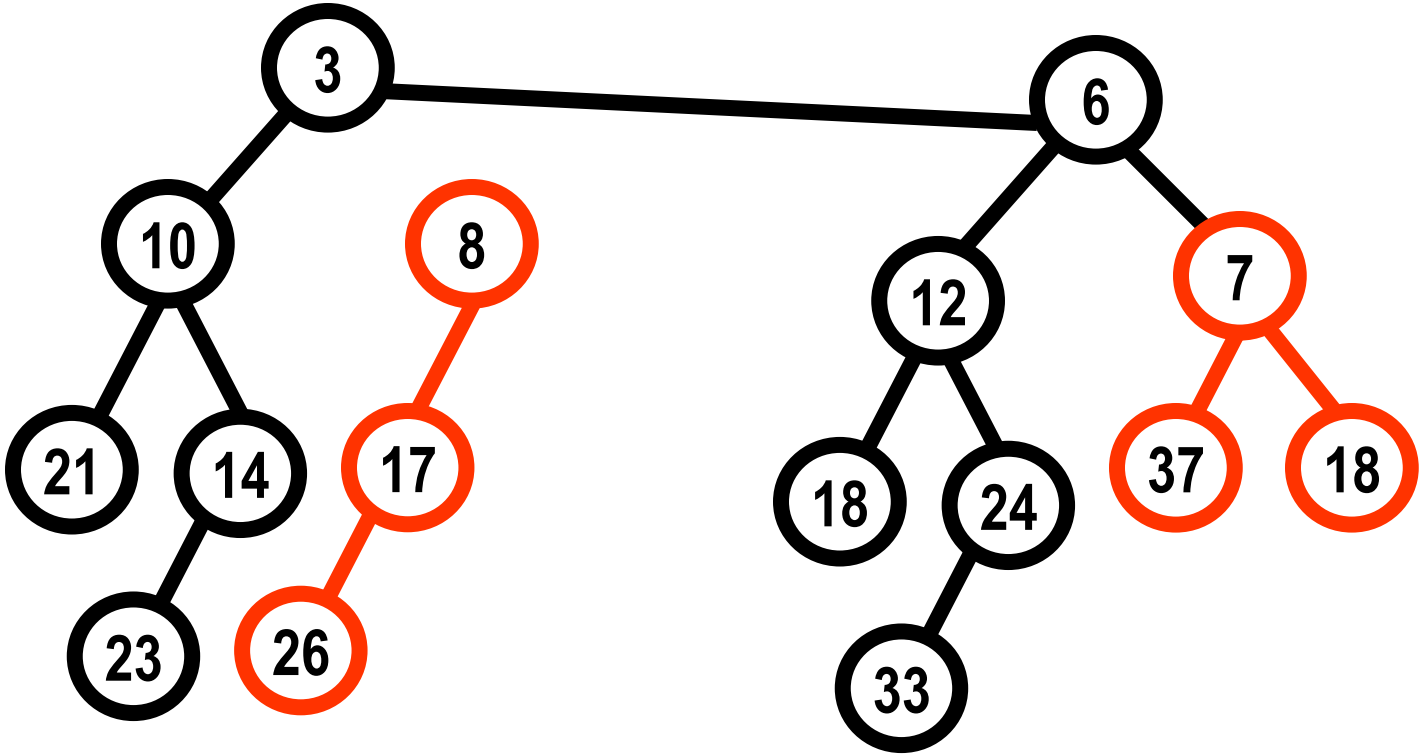


Example: Merging Two Leftist Heaps

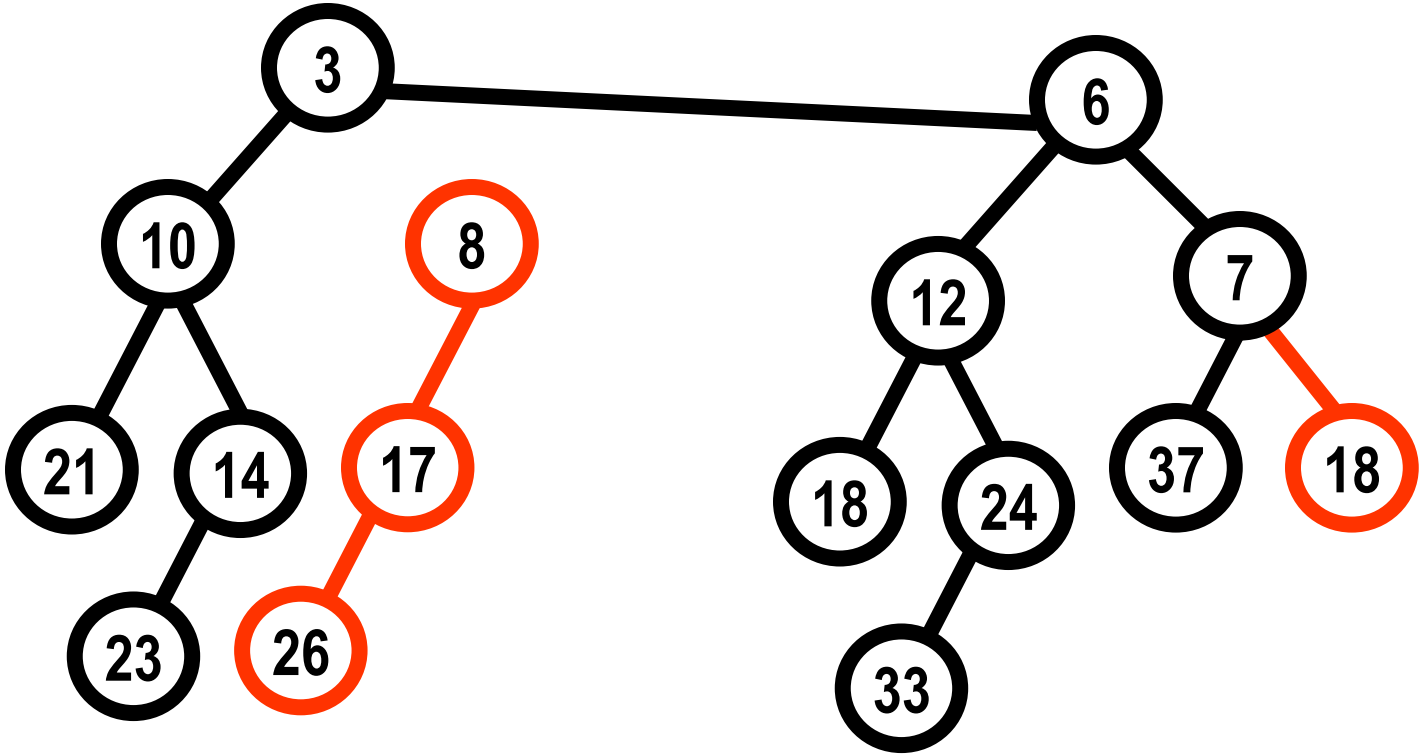




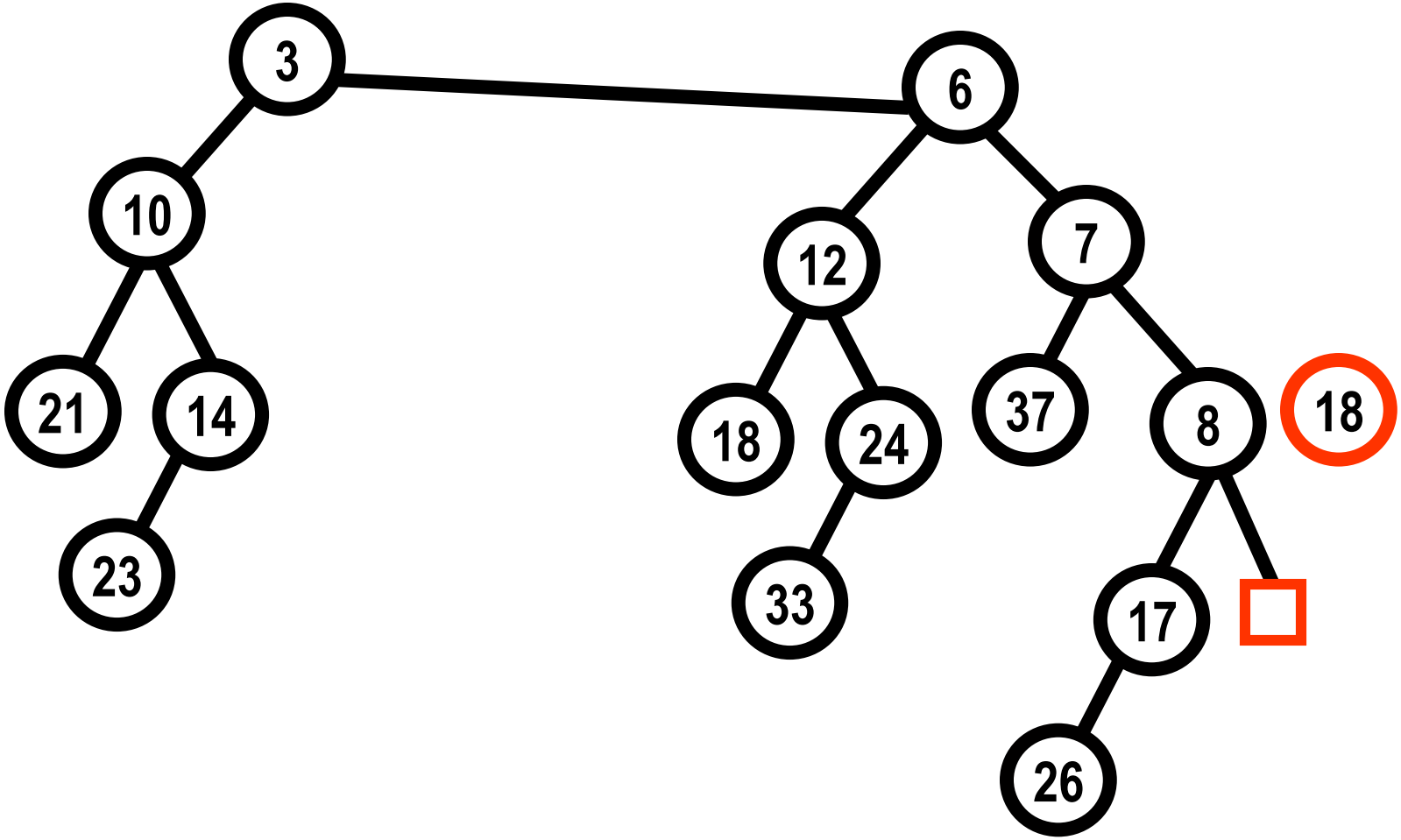
Example: Merging Two Leftist Heaps



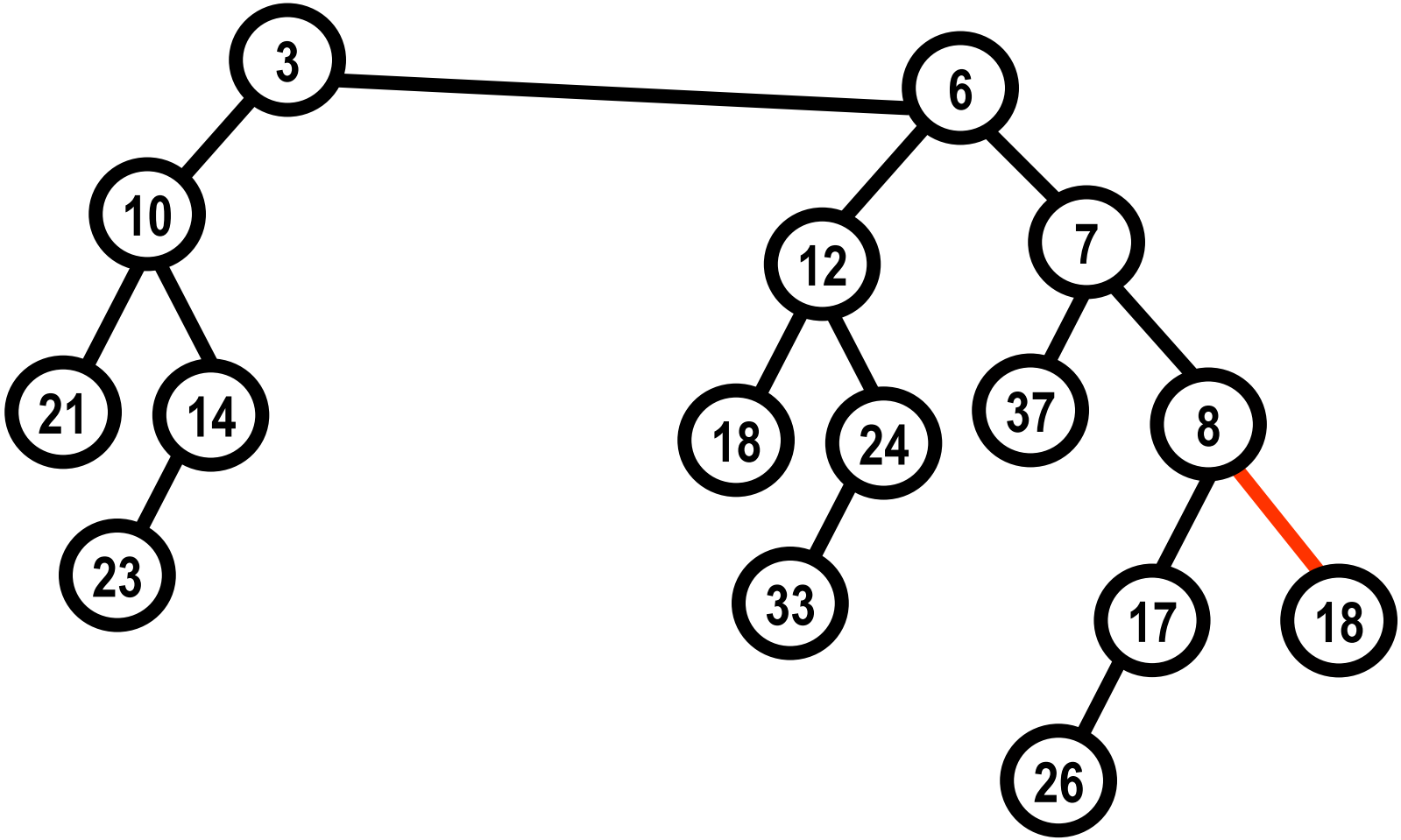
Example: Merging Two Leftist Heaps



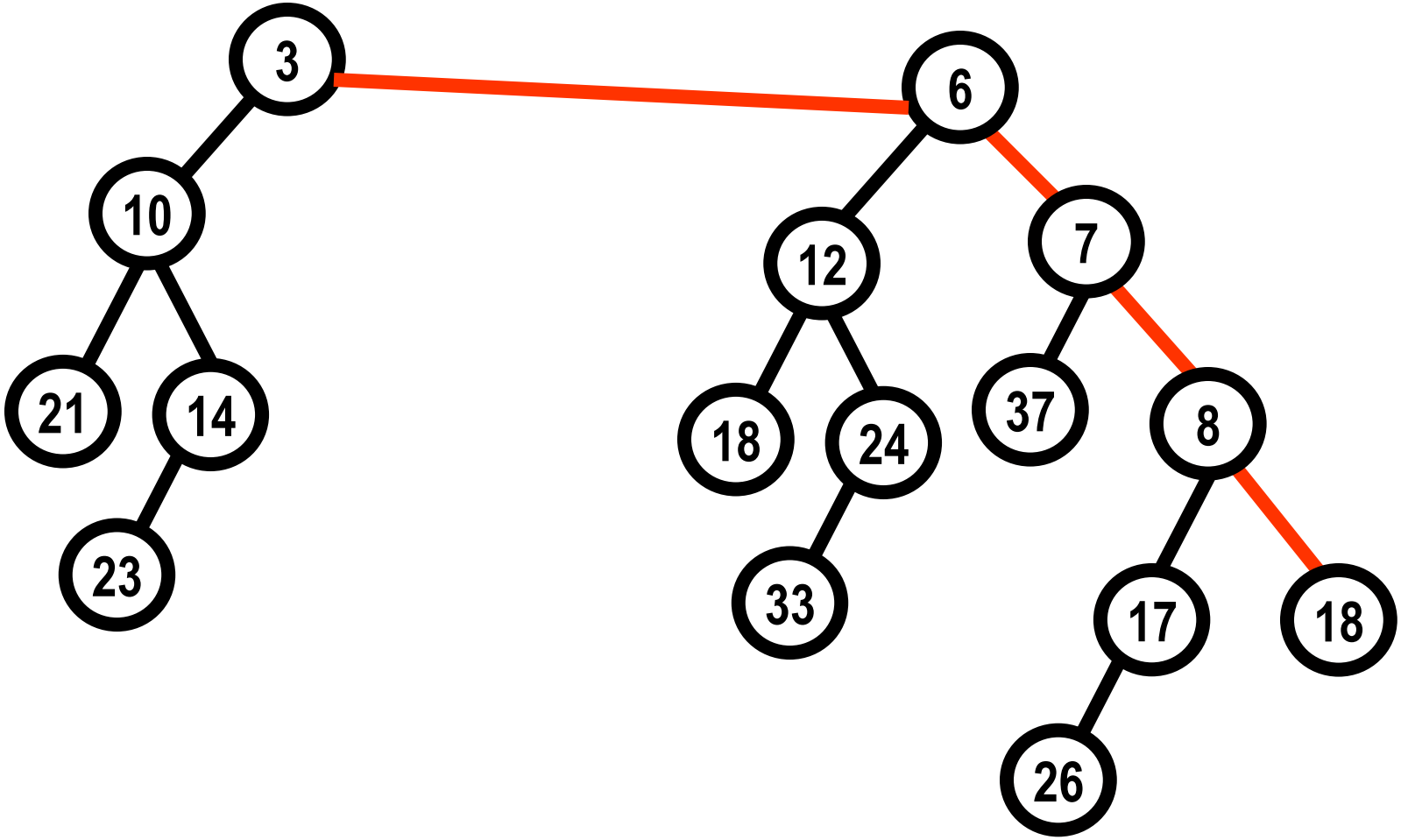
Example: Merging Two Leftist Heaps



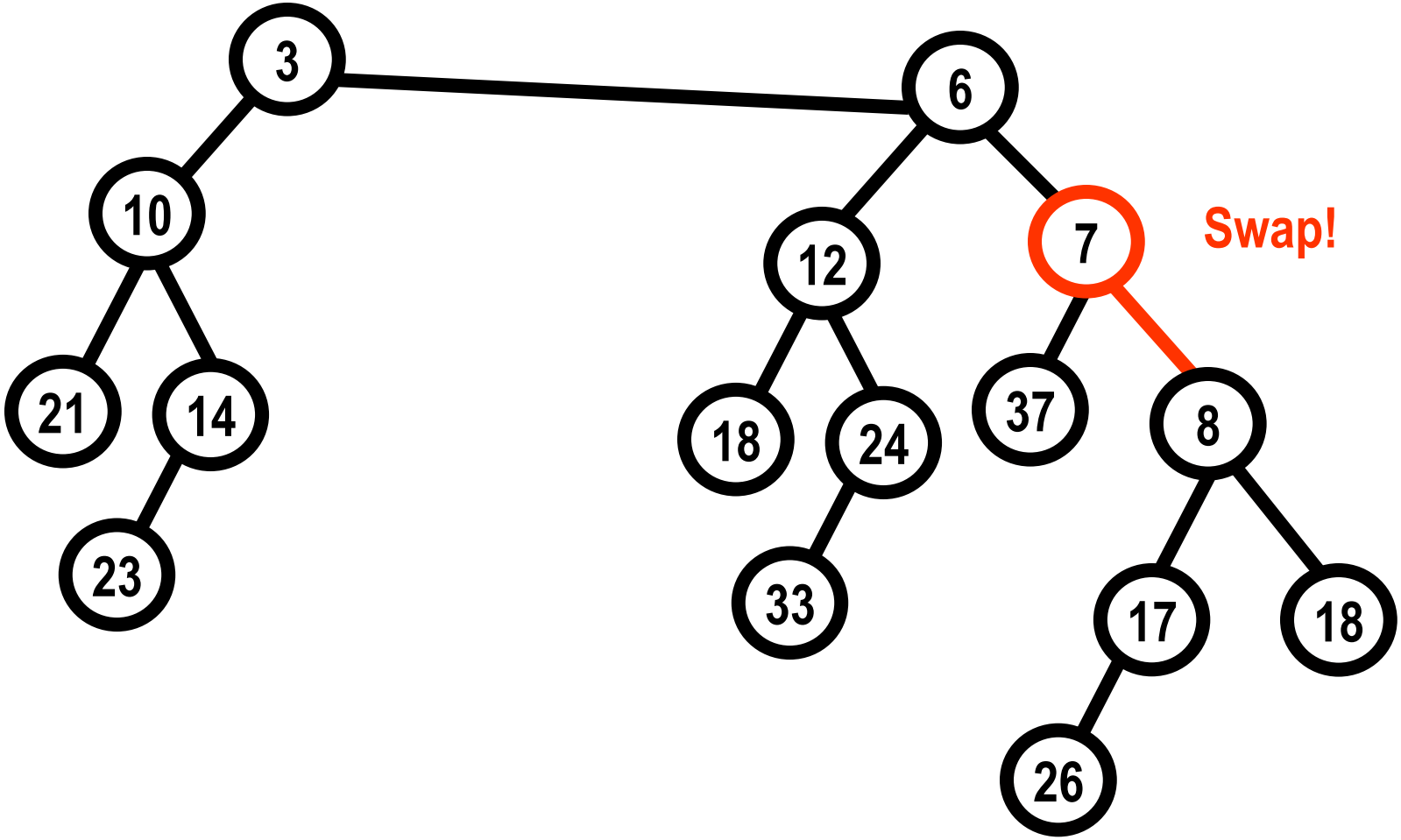
Example: Merging Two Leftist Heaps



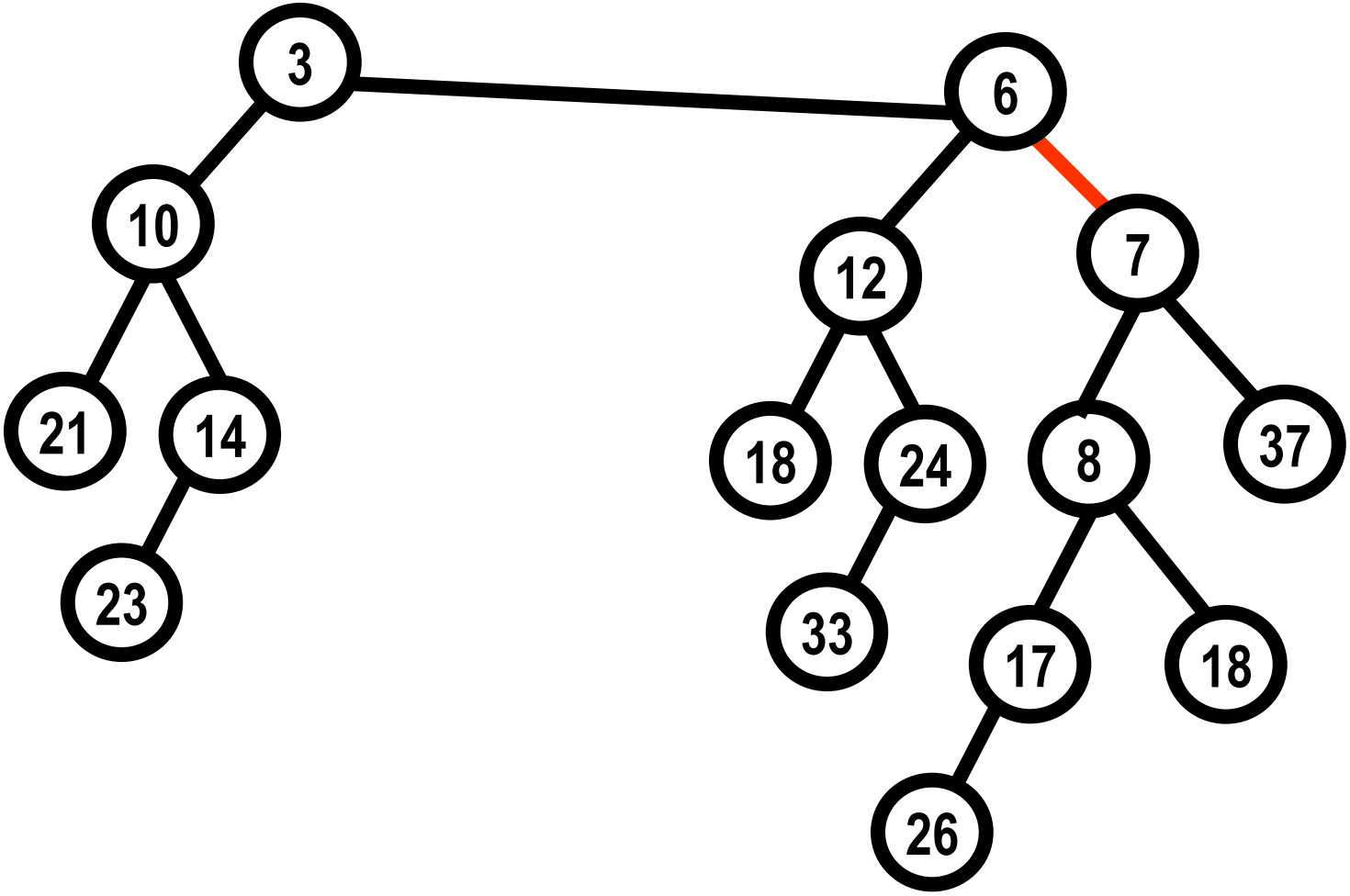
Is This a Leftist Heap Now?



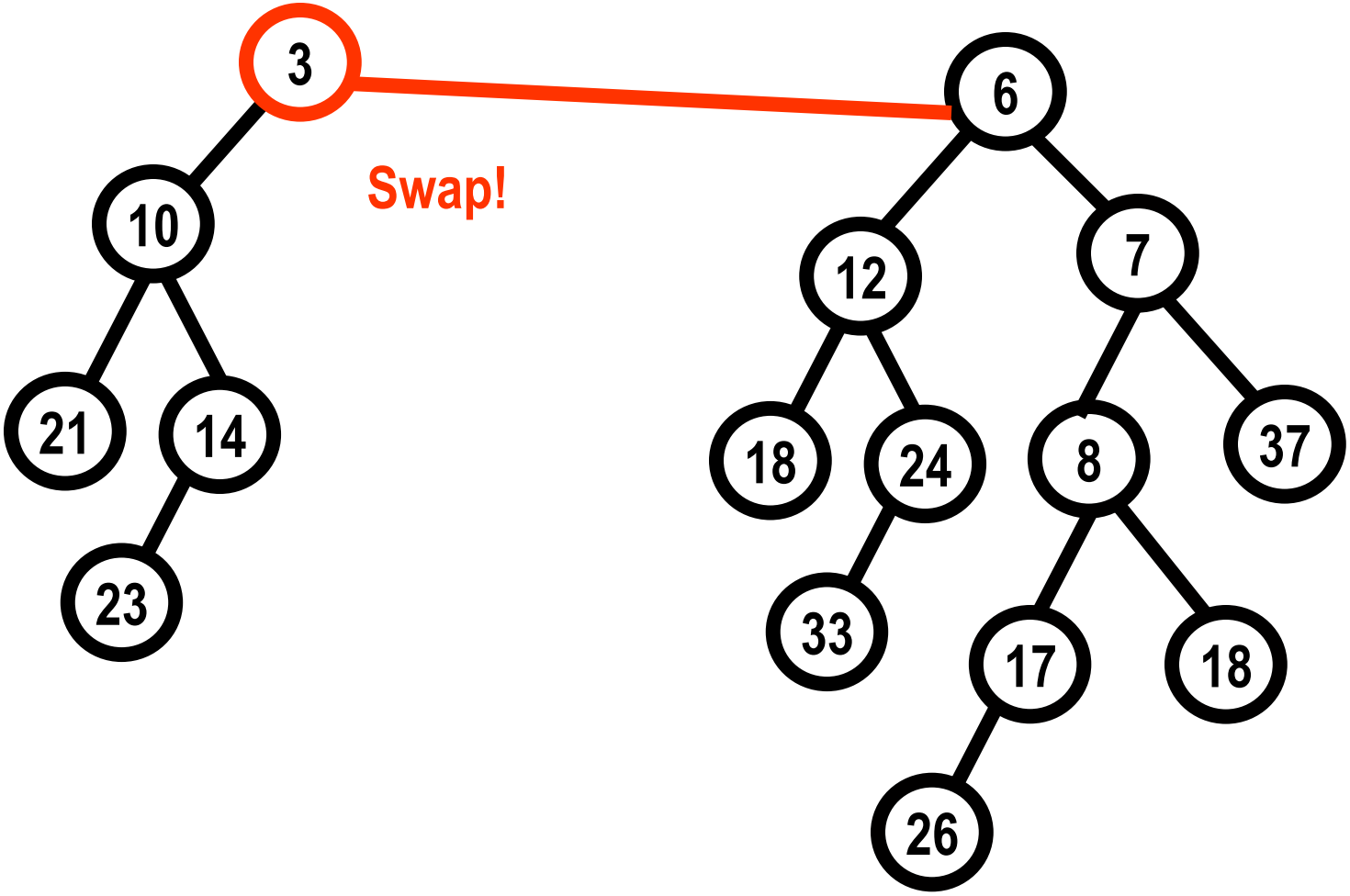
Example: Merging Two Leftist Heaps



Example: Merging Two Leftist Heaps

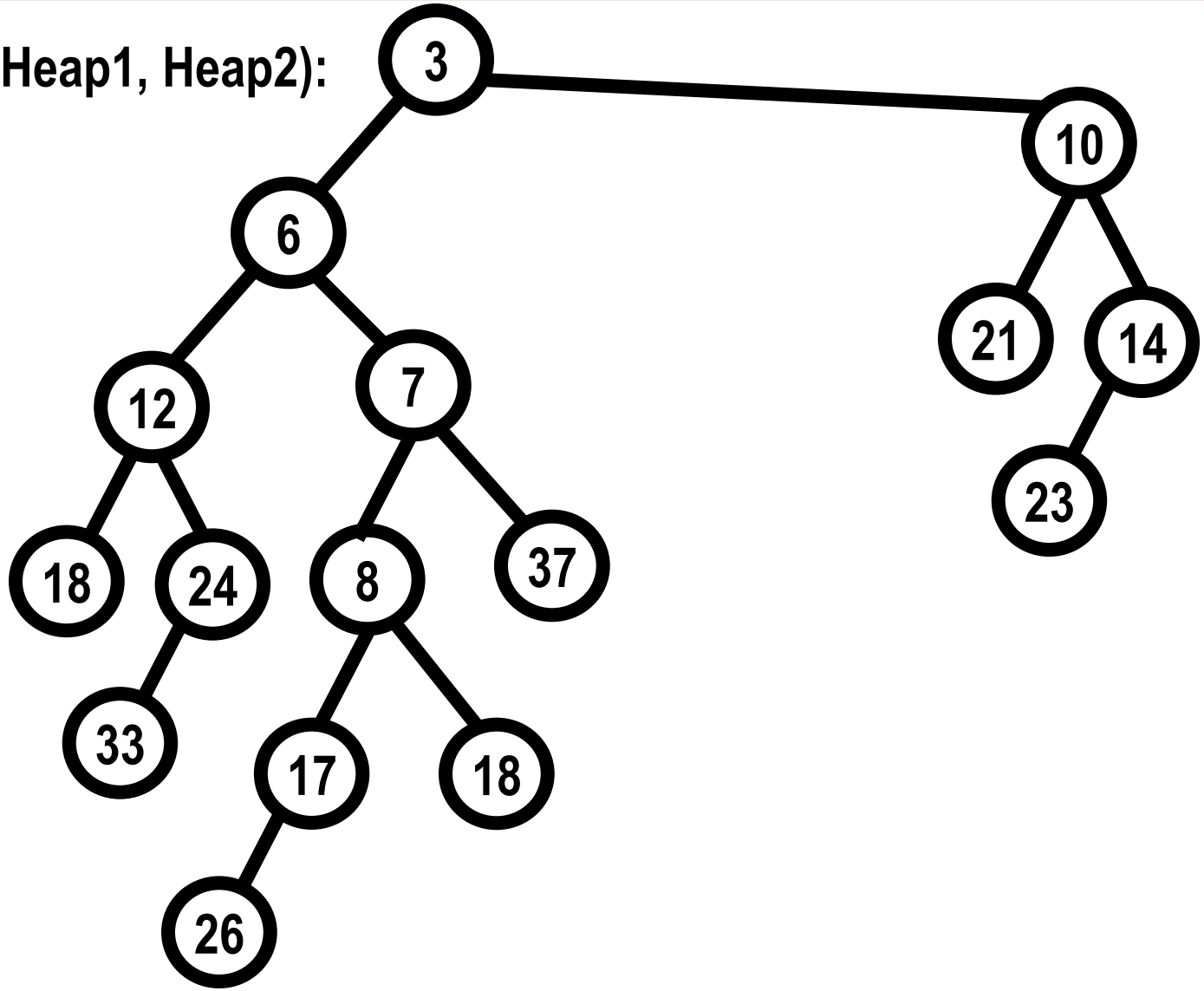


Example: Merging Two Leftist Heaps

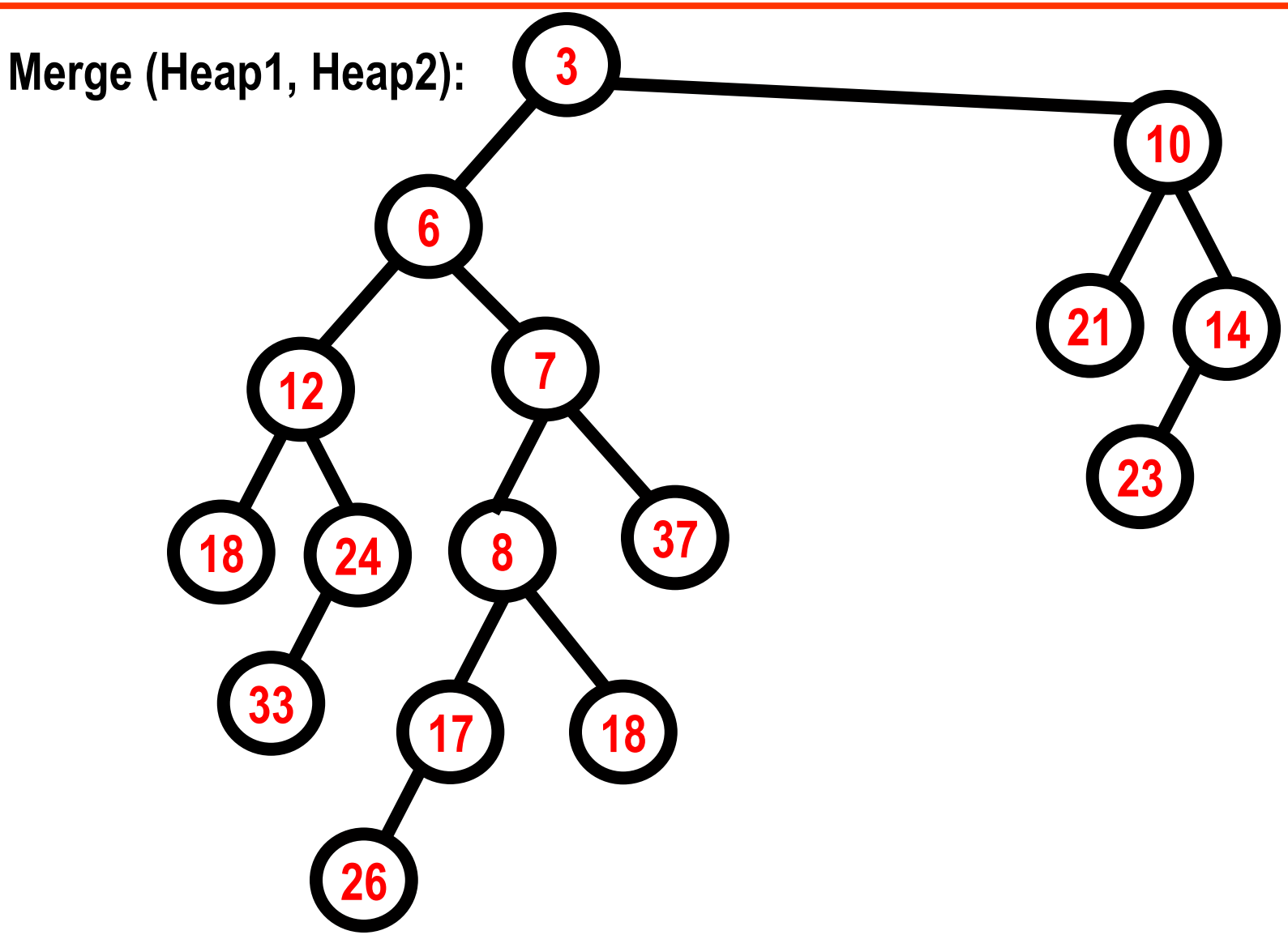


Example: Merging Two Leftist Heaps

Merge (Heap1, Heap2):



Is This a Leftist Heap Now?

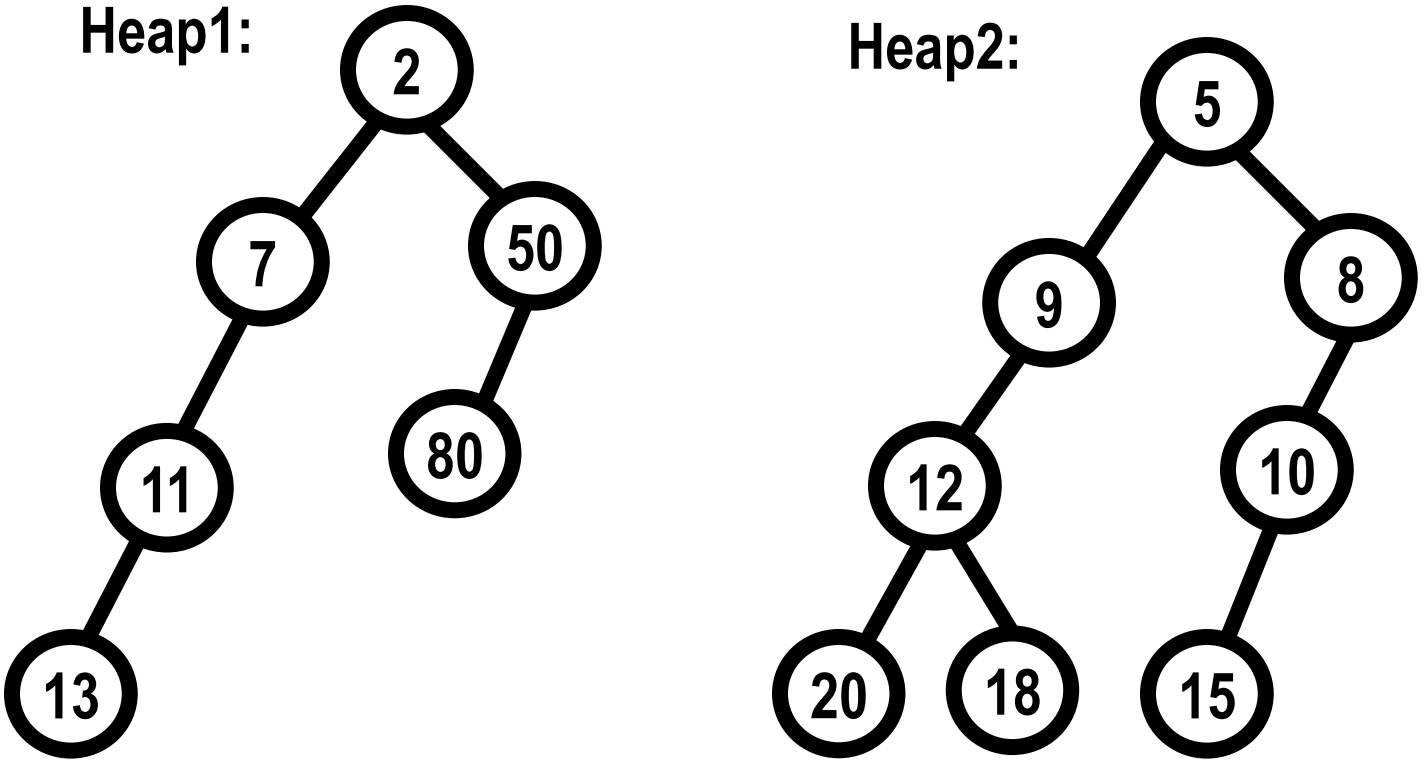


Merge Two Leftist Heaps

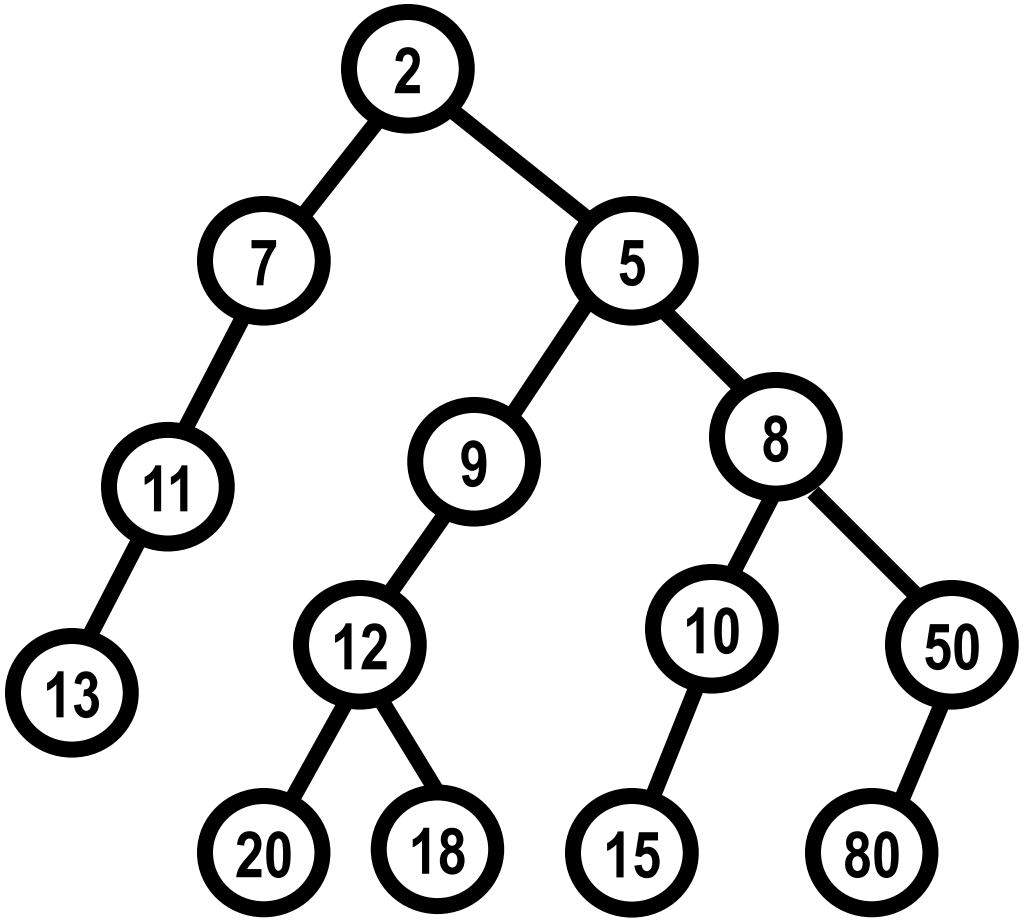
- A **two-pass Merge** process:
  - Top down
  - Bottom-up

- Similar to AVL tree **insertion**!

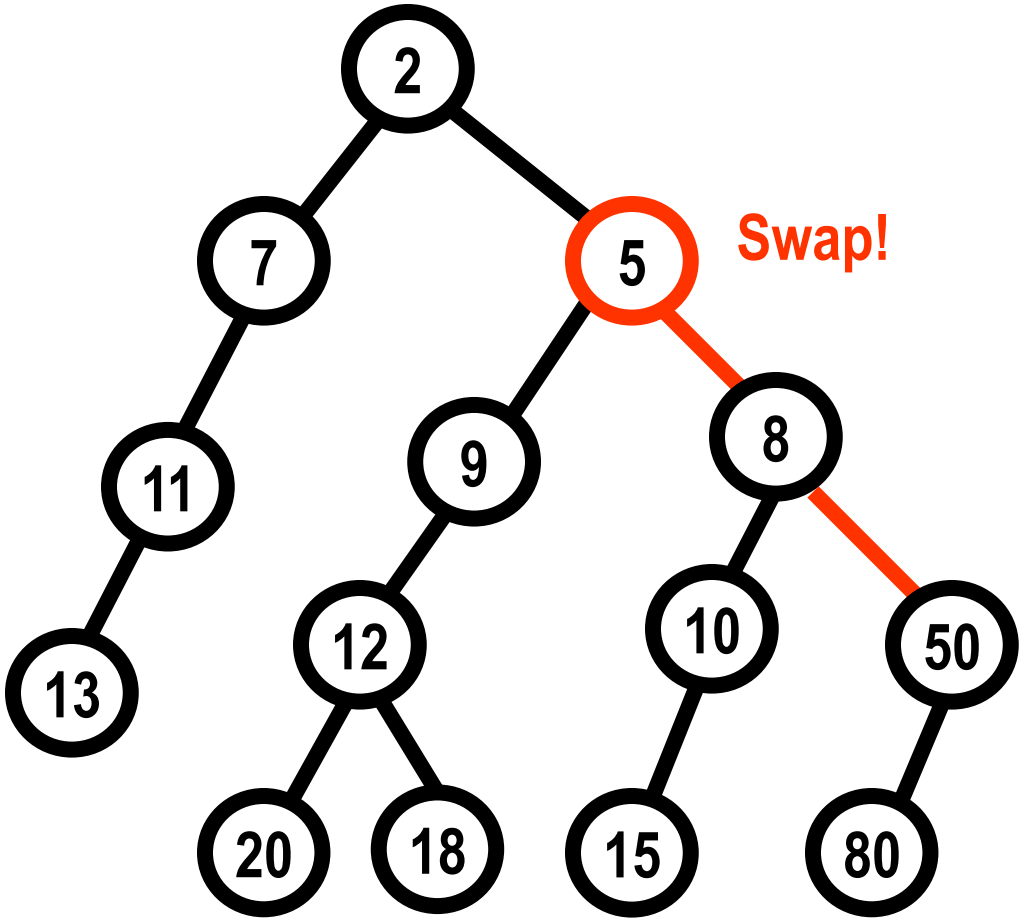
►QUIZ? Merging Two Leftist Heaps?



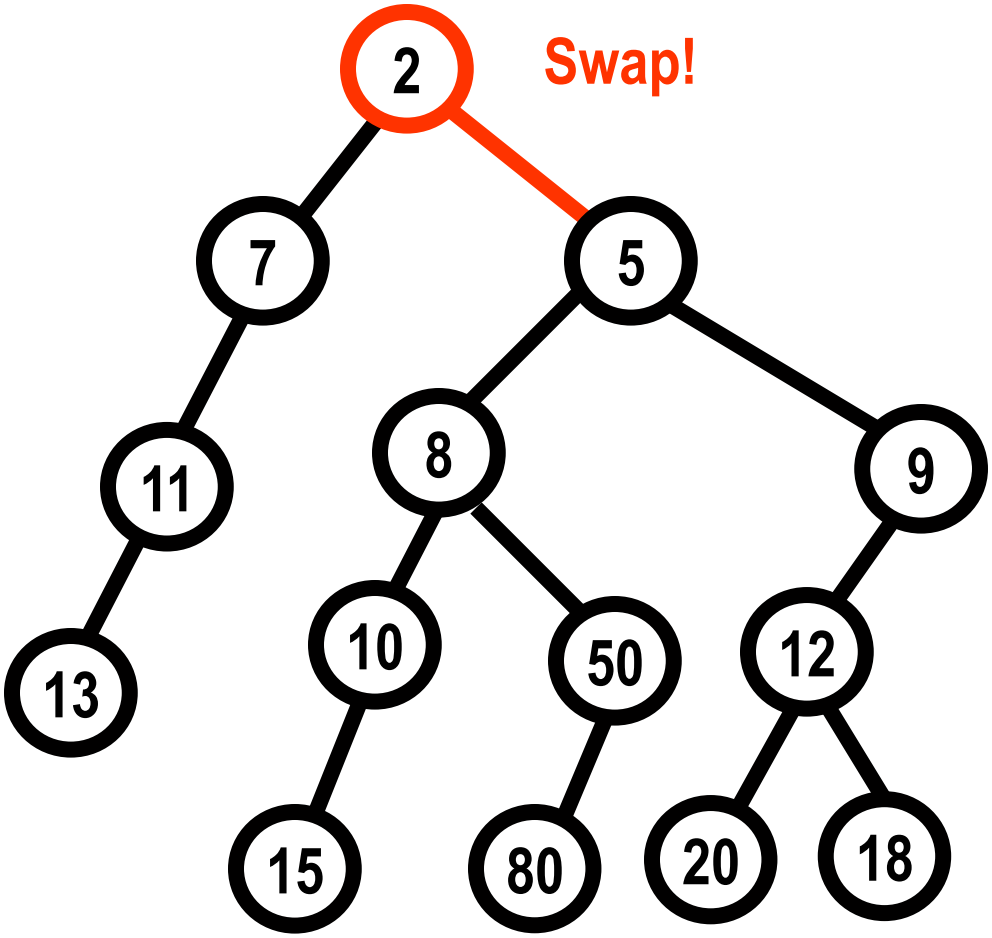
After The Top-Down Pass



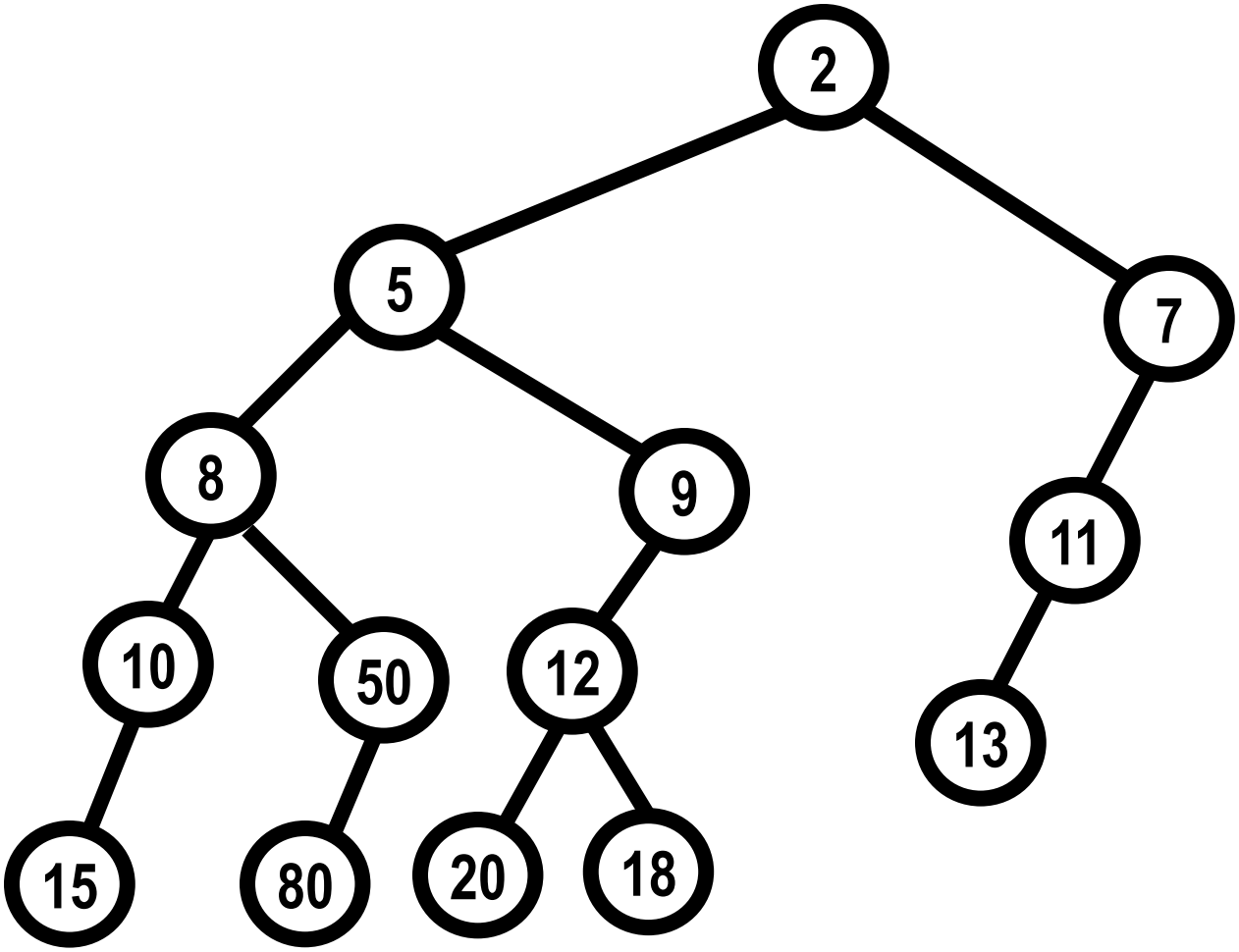
The Bottom-Up Pass



The Bottom-Up Pass

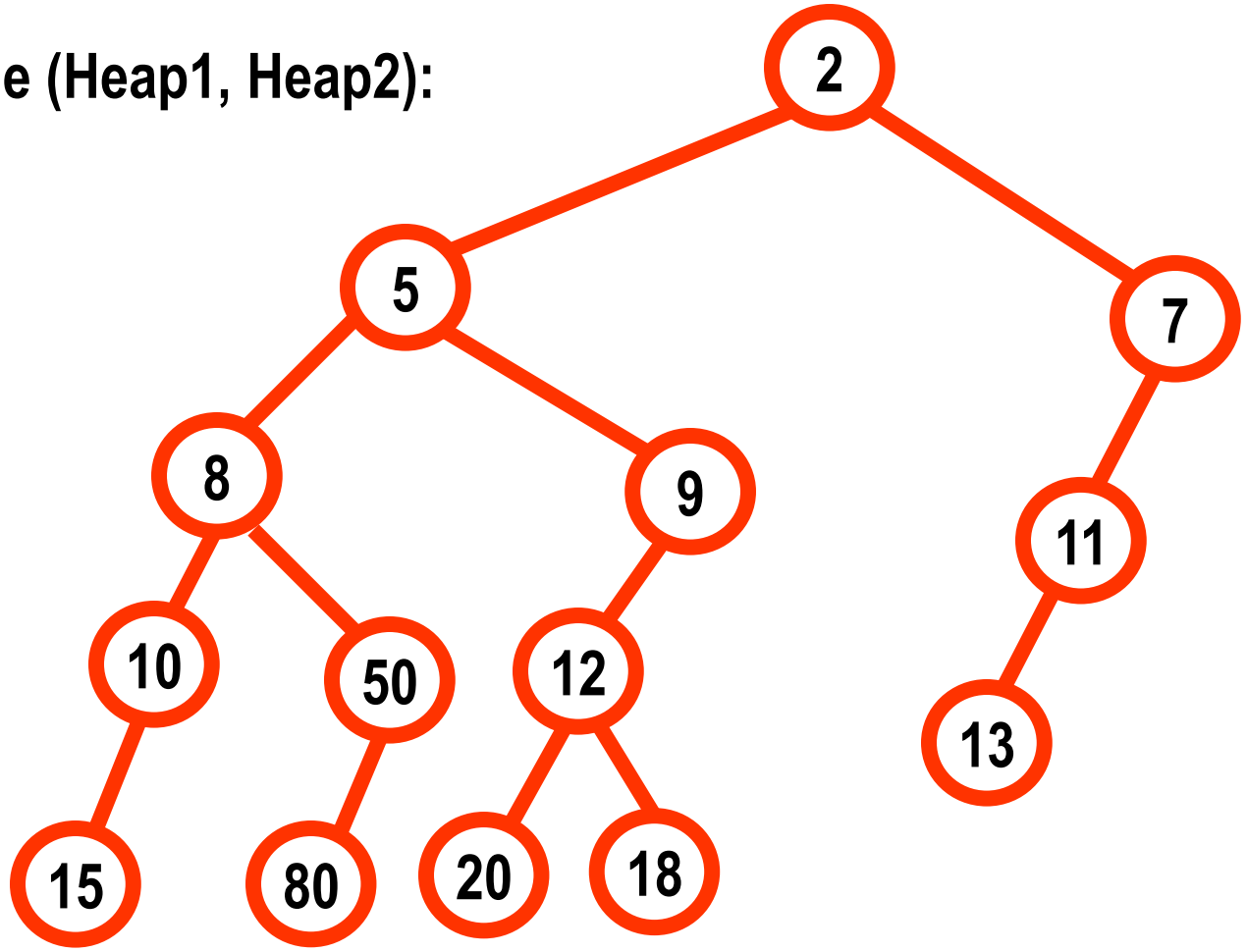


After The Bottom-Up Pass



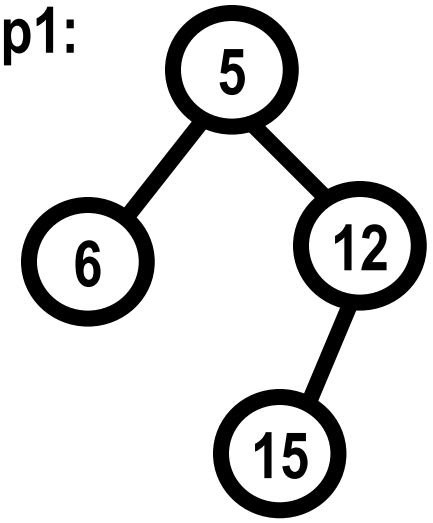
►QUIZ: Merging Two Leftist Heaps

Merge (Heap1, Heap2):

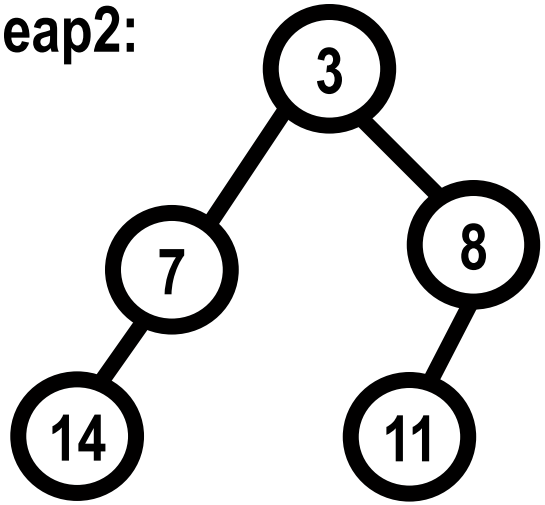


►QUIZ? Merging Two Leftist Heaps?

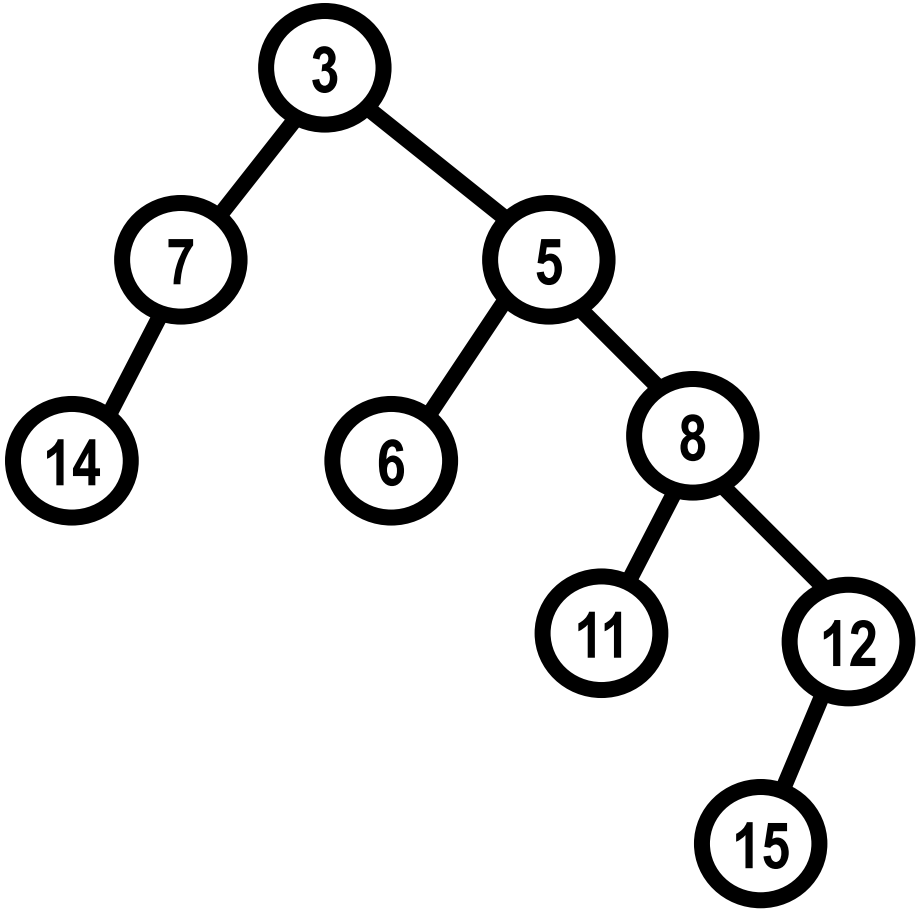
Heap1:



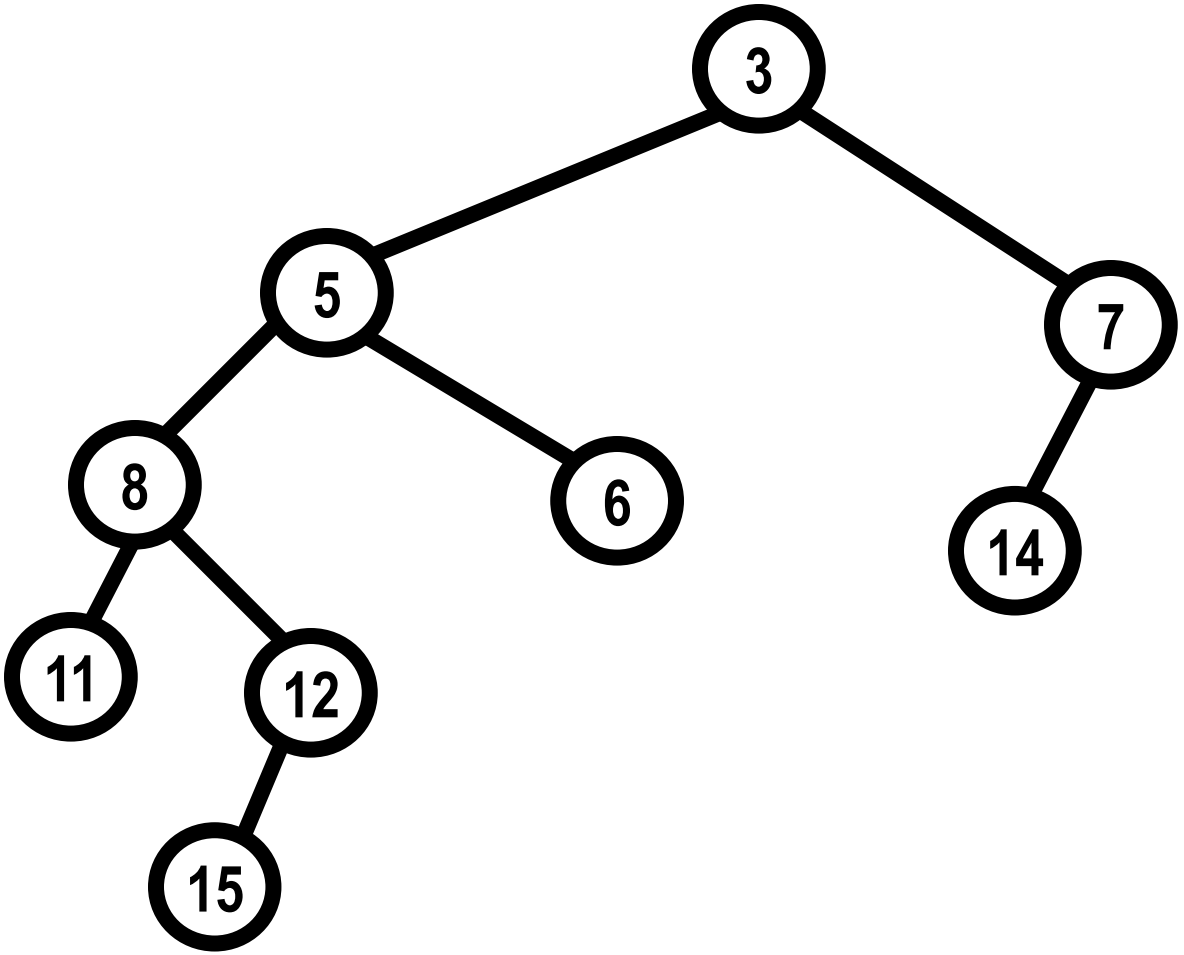
Heap2:



After The Top-Down Pass

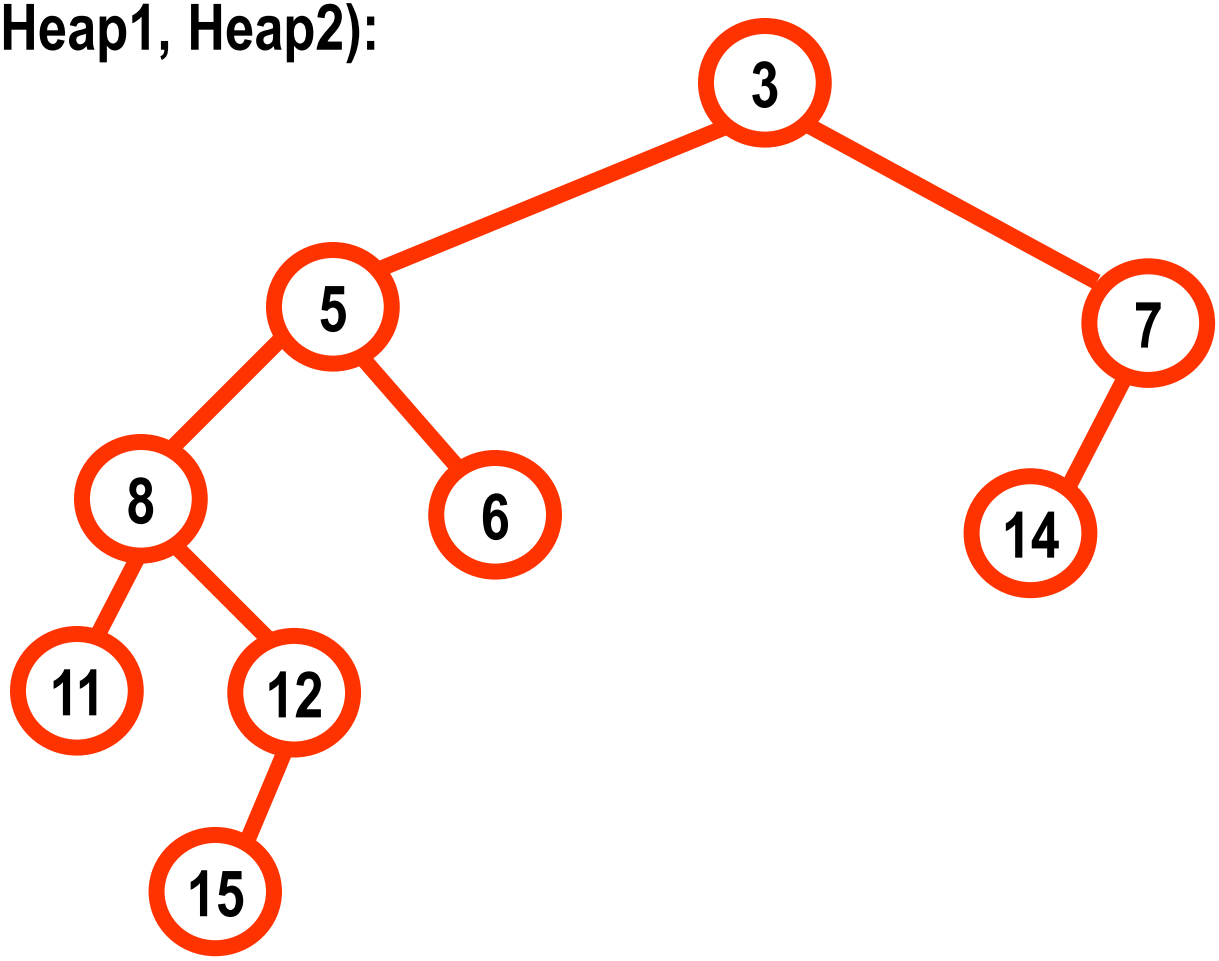


After The Bottom-Up Pass



►QUIZ: Merging Two Leftist Heaps

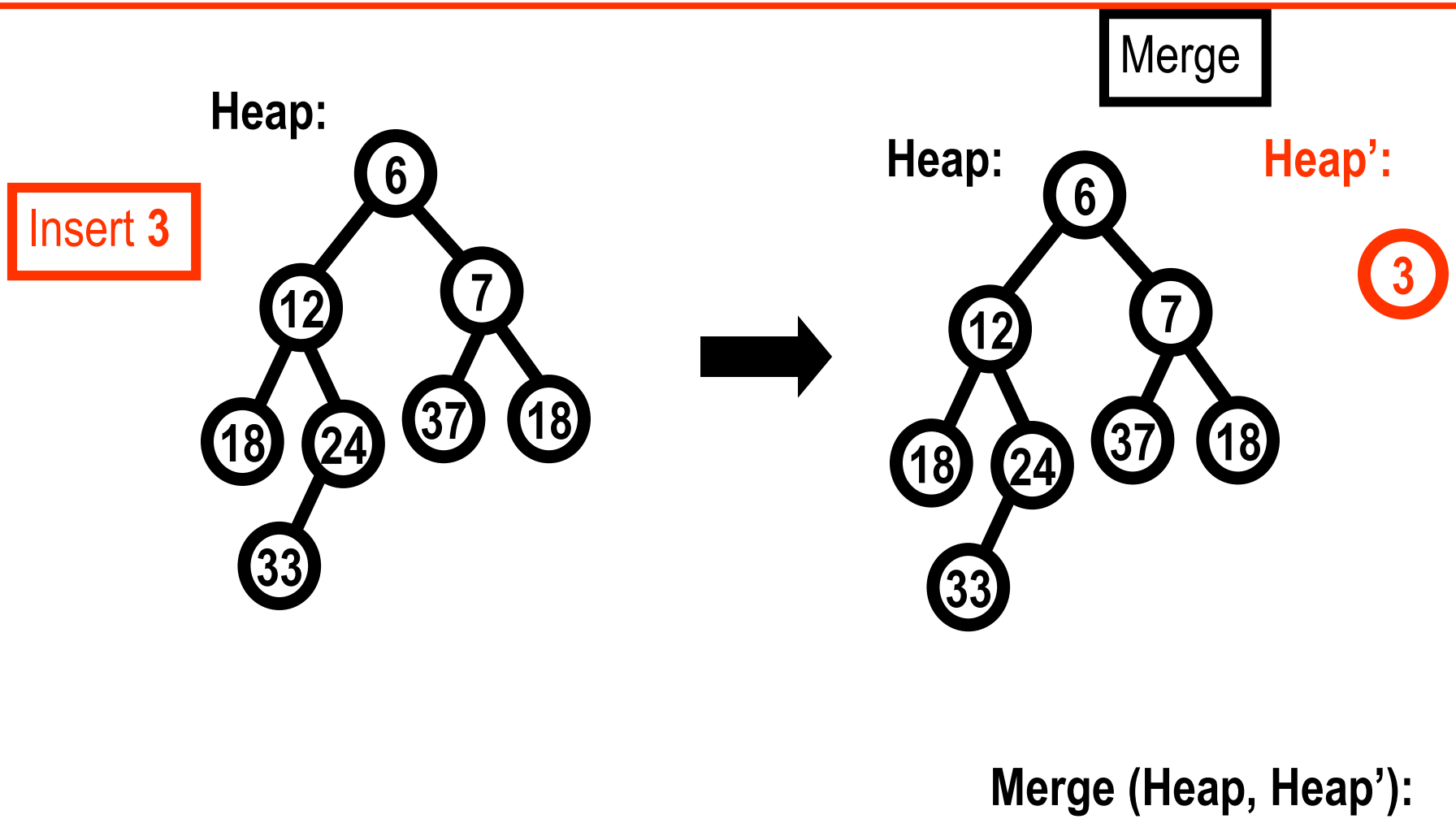
Merge (Heap1, Heap2):



Insert an Item Into a Leftist-heap

- Merge the item leftist heap and the leftist heap.

Example: Insert an item into a leftist-heap



Merge (Heap, Heap'):

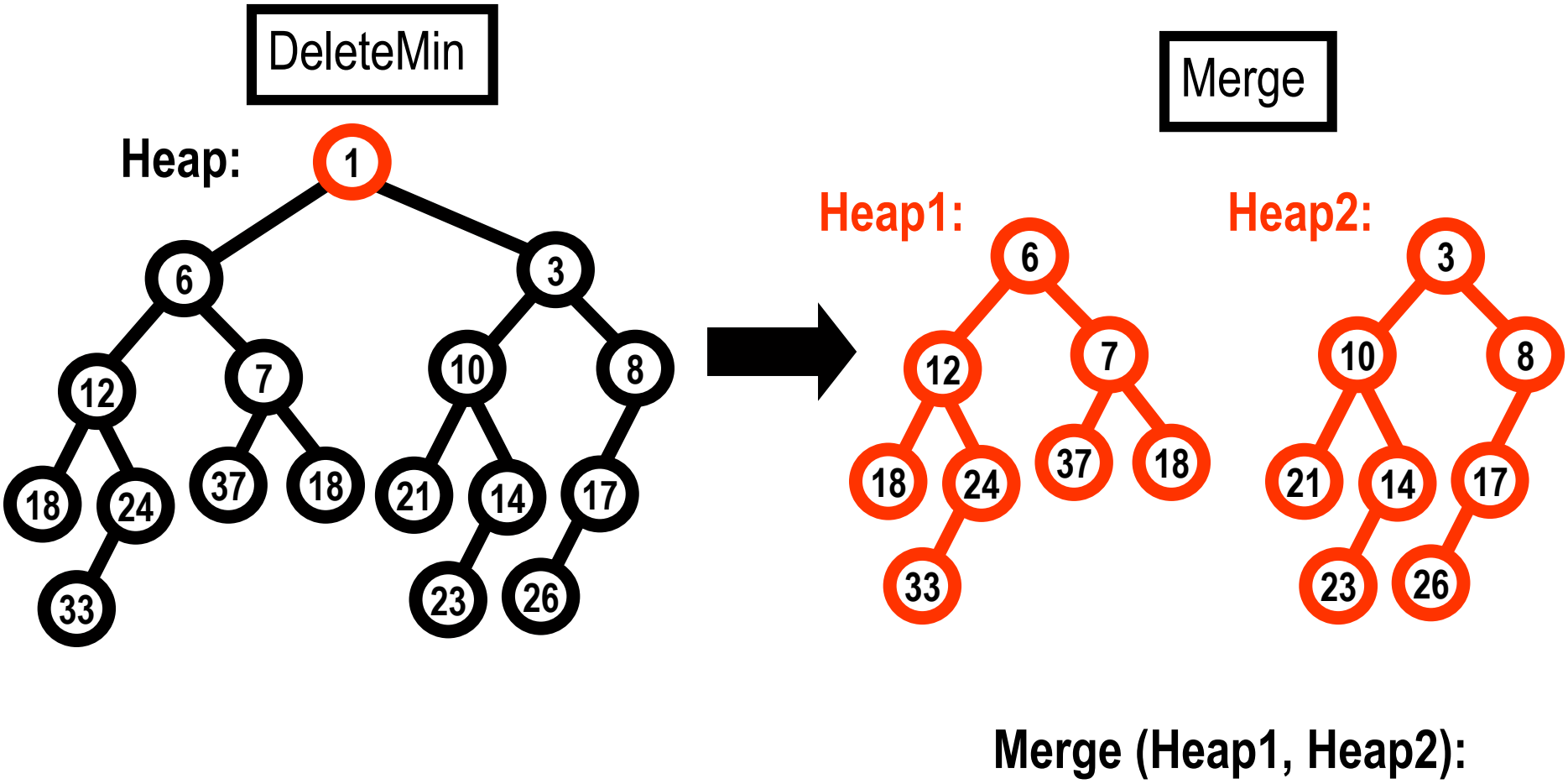
►QUIZ?

- Insert 5, 3, 4, 1 & 2 into an empty leftist min-heap?

DeleteTheRoot From a Leftist-heap

- Merge the left subtree leftist heap and the right subtree leftist heap.

Example: DeleteTheRoot from a leftist-heap



Merge (Heap1, Heap2):

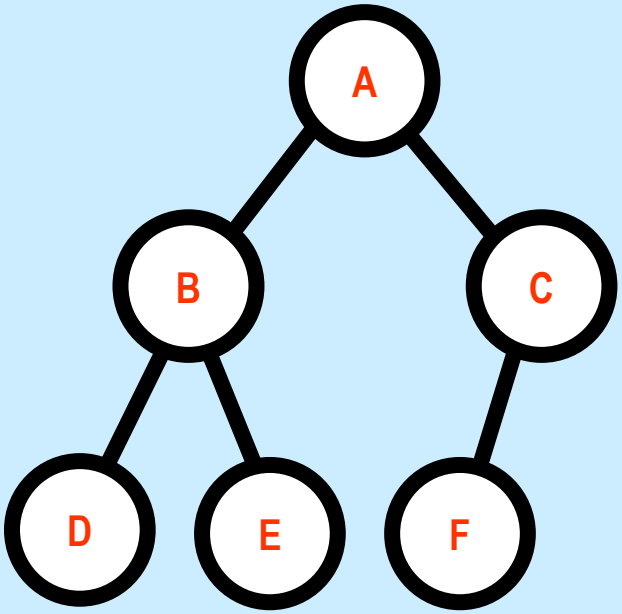
►QUIZ?

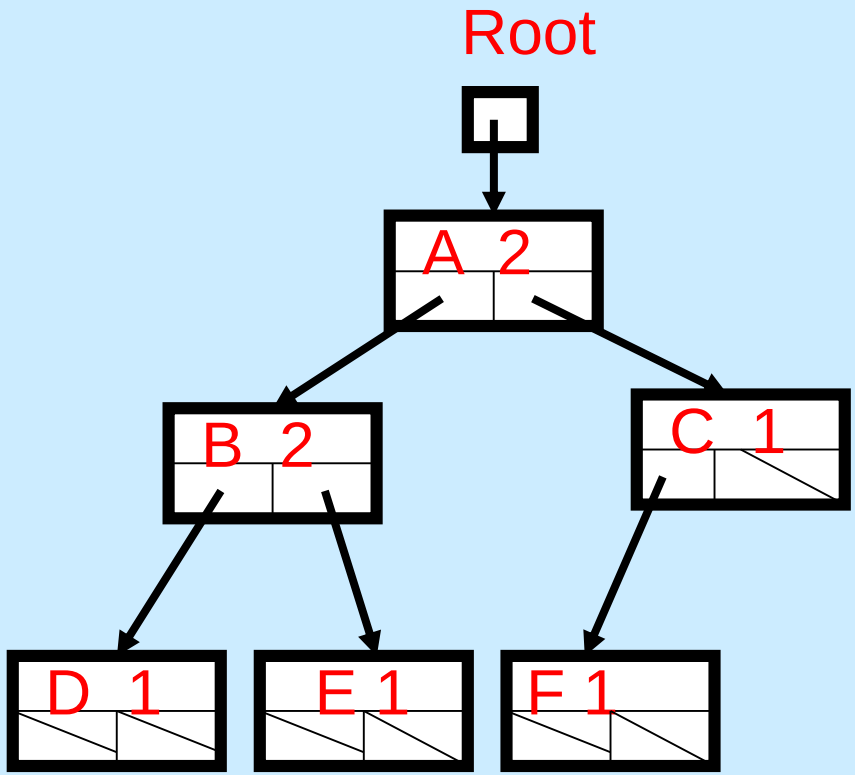
- Insert 5, 3, 4, 1 & 2 into an empty leftist min-heap & DeleteRoot?

How to Represent a Leftist Heap?

- Pointer-based Representation
- Not array-based!

A Pointer-Based Representation of Leftist Heaps





Implementation of Leftist Heaps

```
template <class DT>
class LeftistHeap;

template <class DT>
class LeftistNode
{
   DT   element;
   LeftistNode *left;
   LeftistNode *right;
   int   spl;

   LeftistNode( const DT & theElement, LeftistNode *lt = NULL,
                LeftistNode *rt = NULL, int np = 1 )
     : element( theElement ), left( lt ), right( rt ), spl( np ) { }

   friend class LeftistHeap<DT>;
};
```

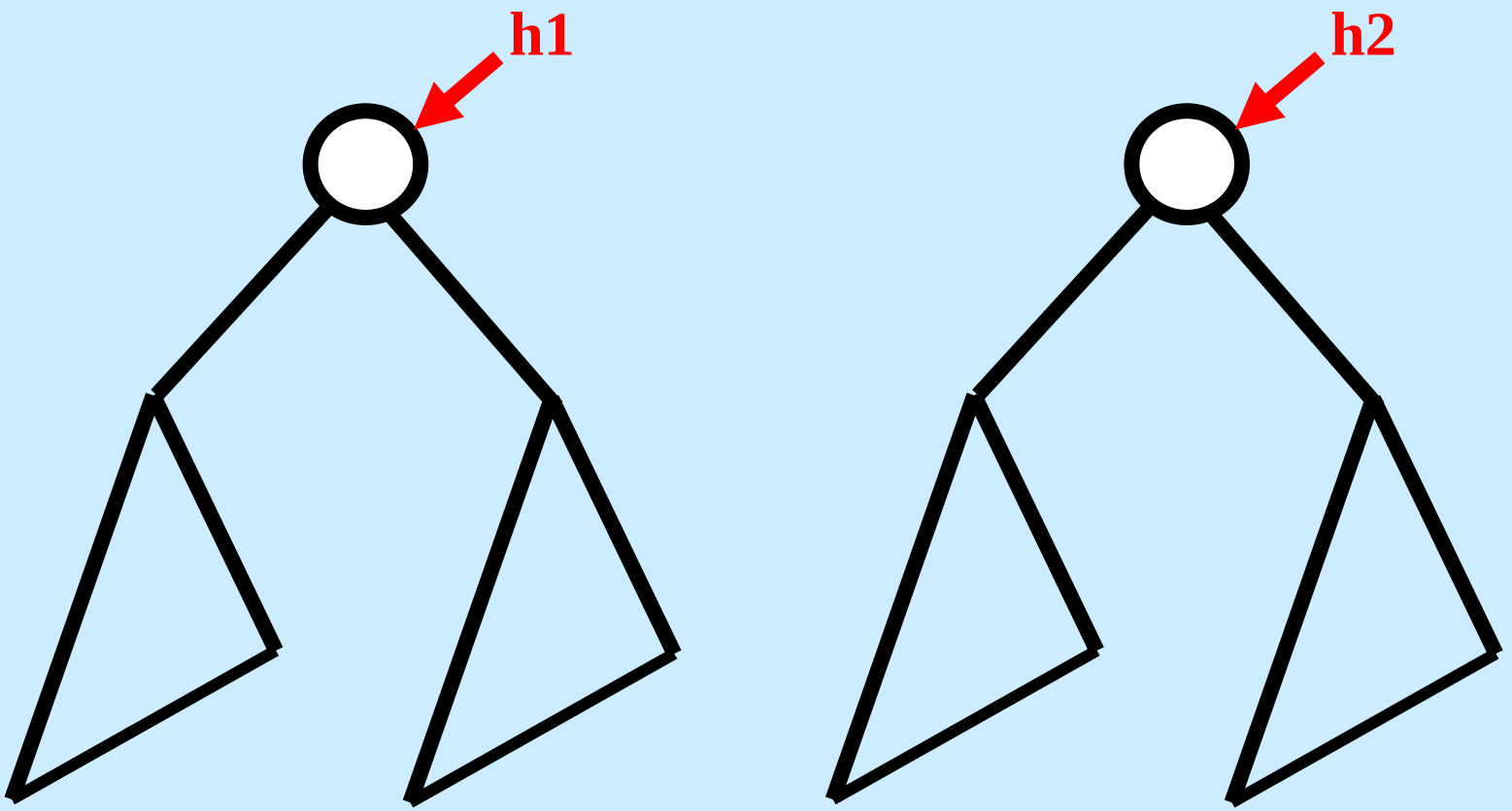
Implementation of Leftist Heaps

```
template <class DT>
class LeftistHeap
{
  public:
    LeftistHeap( );
    LeftistHeap( const LeftistHeap & rhs );
    ~LeftistHeap( );
    const LeftistHeap & operator=( const LeftistHeap & rhs );
    void insert( const DT & x );
    void deleteMin( DT & minItem );
    void merge( LeftistHeap & rhs );
    void printLHeap( ) const;
  private:
    LeftistNode<DT> *root;
    LeftistNode<DT> * merge1 ( LeftistNode<DT> *h1,
                               LeftistNode<DT> *h2 ) const;
    LeftistNode<DT> * merge2( LeftistNode<DT> *h1,
                              LeftistNode<DT> *h2 ) const;
    void swapChildren( LeftistNode<DT> * t ) const;
};
```

Merge

```
/**
 * Merge the rhs leftist heap into the leftist heap.
 * rhs becomes empty.
 */

template <class DT>
void LeftistHeap<DT>::merge( LeftistHeap & rhs )
{

    root = merge1( root, rhs.root );

    rhs.root = NULL;
}
```

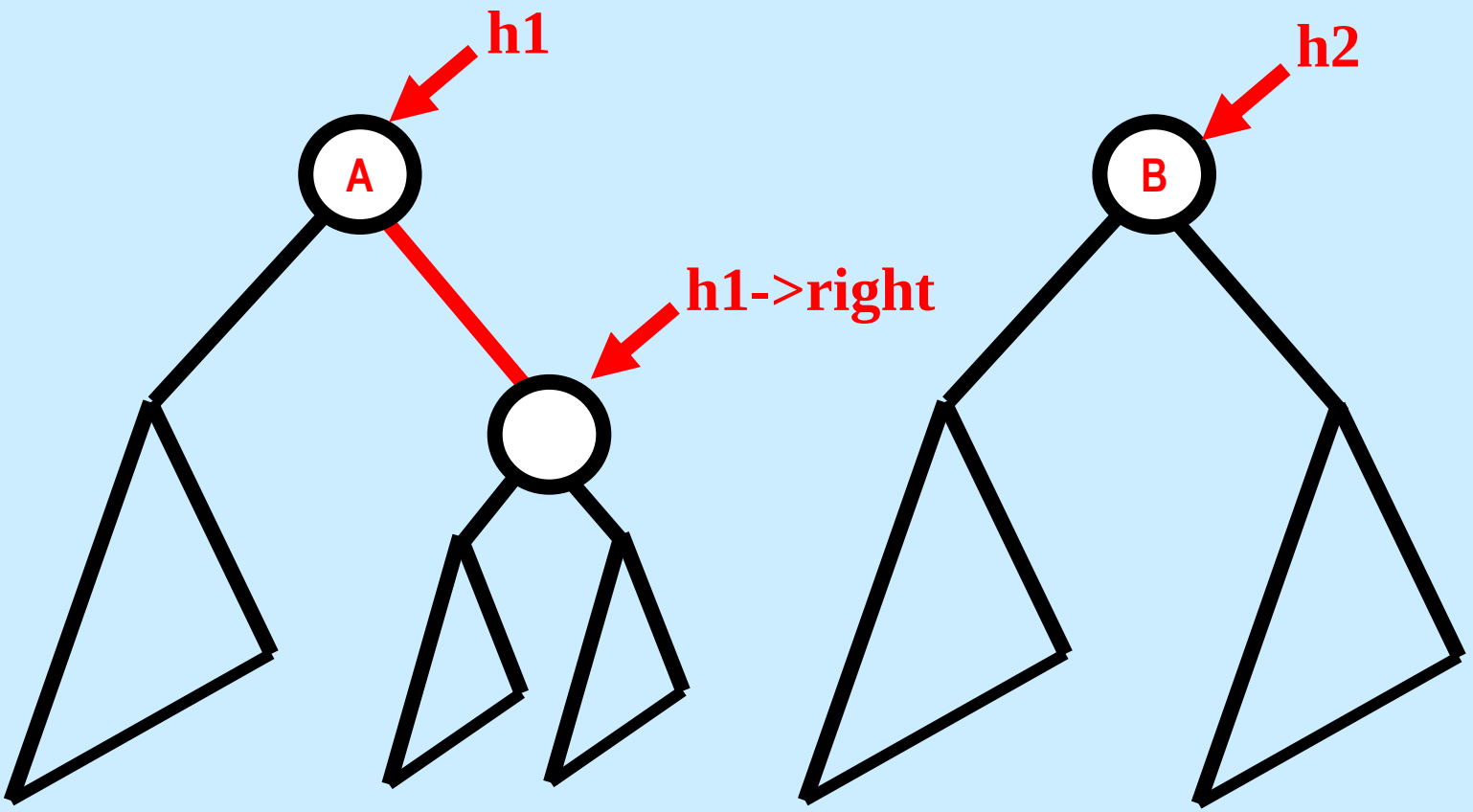
Merge1



Merge1

```
/**
 * Merge two roots h1 and h2.
 */
 template <class DT>
 LeftistNode<DT> *
 LeftistHeap<DT>::merge1( LeftistNode<DT> * h1,
                                        LeftistNode<DT> * h2 )
{
    if ( h1 == NULL )  return h2;
    if ( h2 == NULL )  return h1;

    if (h1->element < h2->element ) return merge2(h1, h2);
    else return merge2(h2, h1);

}
```

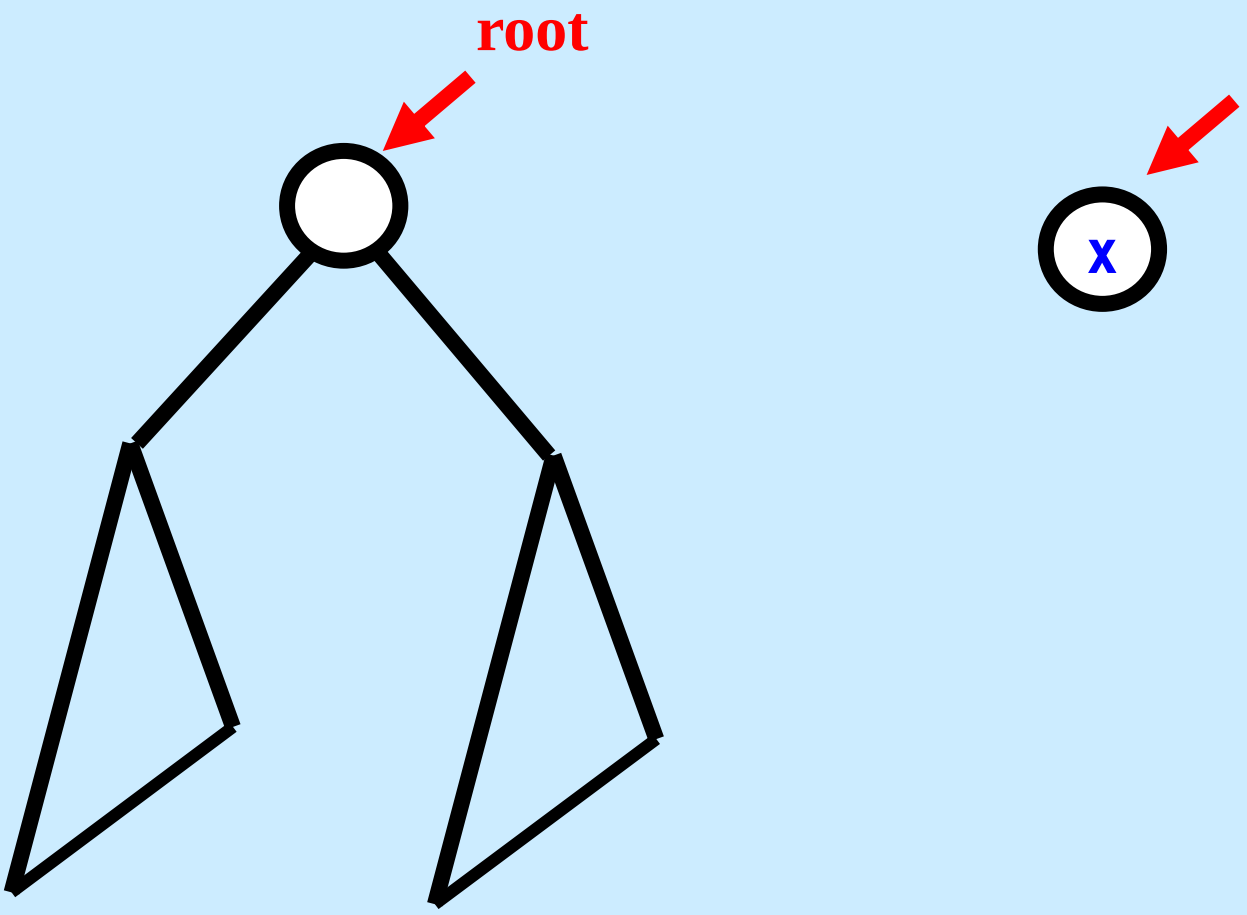
Merge2



Merge2

```
/**
 * Merge two not empty roots h1 and h2.
 * h1's root contains smallest item.
 */
template <class DT>
LeftistNode<DT> *
LeftistHeap<DT>::merge2( LeftistNode<DT> * h1,
                                    LeftistNode<DT> * h2 ) const
{
    // Recursively merge its right subtree and the other tree h2.
    h1->right = merge1( h1->right, h2 );
    // Swap if needed.
    if( h1->left->spl < h1->right->spl )
      swapChildren( h1 );
    // Update the spl of the merged root.
    h1->spl = h1->right->spl + 1;
    return h1;
}
```

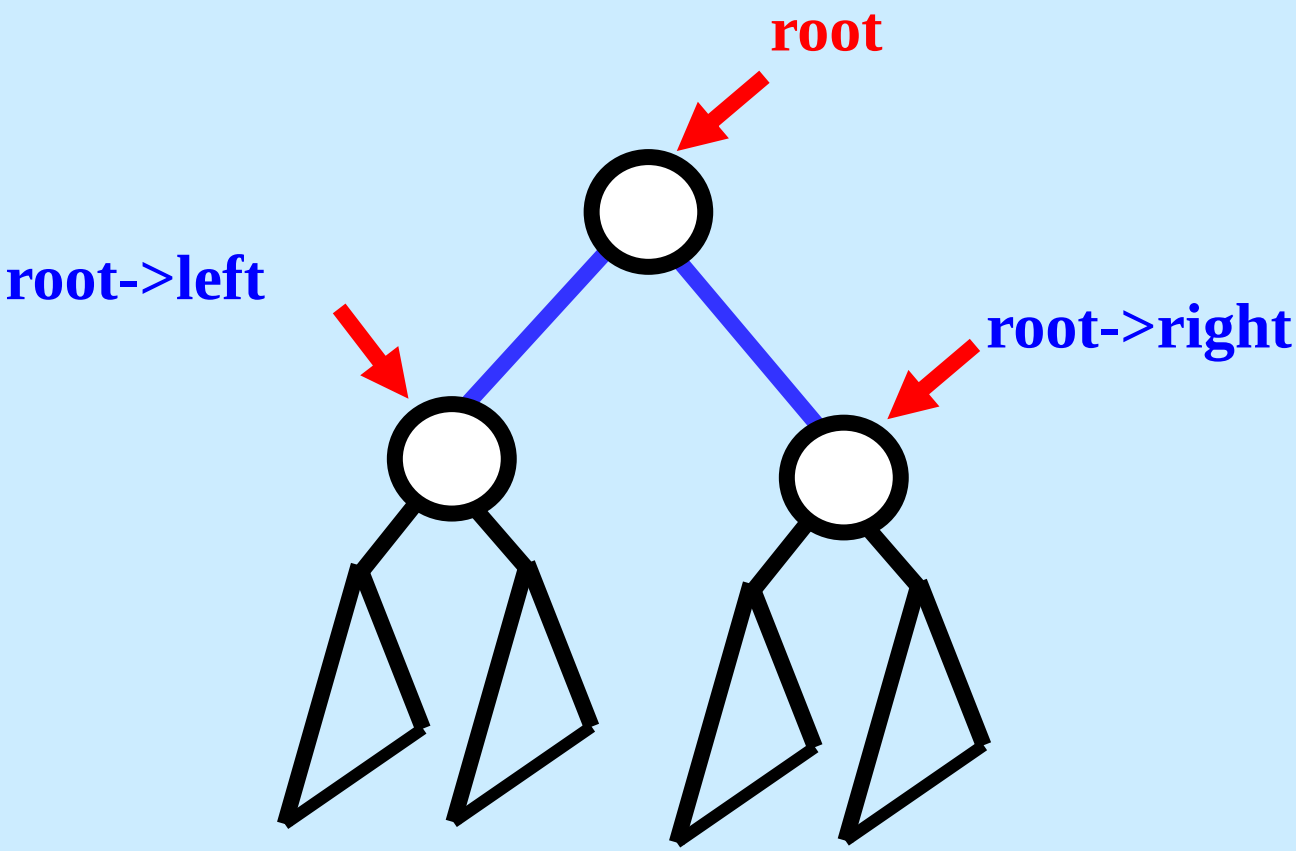
Insert



Insert

```
template <class DT>
void LeftistHeap<DT>::insert( const DT & x )
{
    root = merge1( new LeftistNode<DT>( x ), root );
}
```

DeleteMin



DeleteMin

```
template <class DT>
void LeftistHeap<DT>::deleteMin( DT & minItem )
{
    minItem = findMin( );
    LeftistNode<DT> *oldRoot = root;
    root = merge1( root->left, root->right );
    delete oldRoot;
 }
```

Leftist Heap Operations - Analysis

- The height of **the right path** of a leftist tree with N nodes is O(log N).
- Worst-case time:
  - Merge
    - **O(log N)**
  - Insert
    - **O(log N)**
  - DeletetheRoot
    - **O(log N)**

Leftist Heap History

- Crane, Clark A. (1972), **Linear Lists and Priority Queues as Balanced Binary Trees** (Ph.D. thesis), Department of Computer Science, Stanford University.

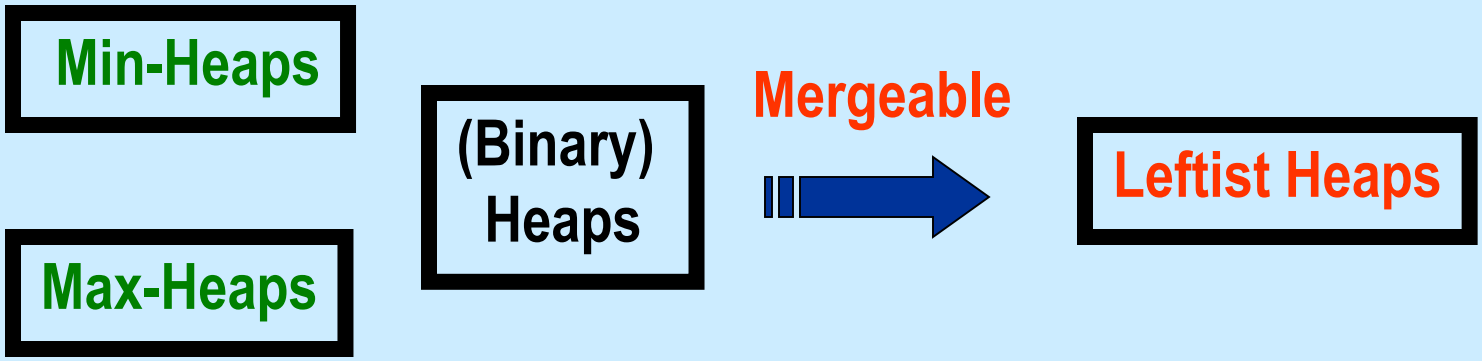
Leftist Heap Visualization

- ***Leftist Heap Visualization***

►QUIZ?

- Compare Binary Heap with Leftist Heap?

The World of Heaps



Skew Heaps

(Self-Adjusting Heaps)

Skew Heaps

- A **skew heap** is a **self-adjusting** version of a leftist heap!
  - ➔ **Self-Adjusting Heaps!**
  - ➔ Similar to Skip lists-to-Self-organizing lists!
  - ➔ Similar to AVL trees-to-Splay trees!

Skew Heaps

- Problems with leftist heaps:
  - Extra storage for spl
  - Extra complexity/logic to maintain and check spl
- Observation!
  - Right side of leftist heap is “often” heavy and requires a switch!

Skew Heaps

- Why?
  - ➔ To improve the **amortized (average time over operations) time** of **merge**!

Skew Heaps

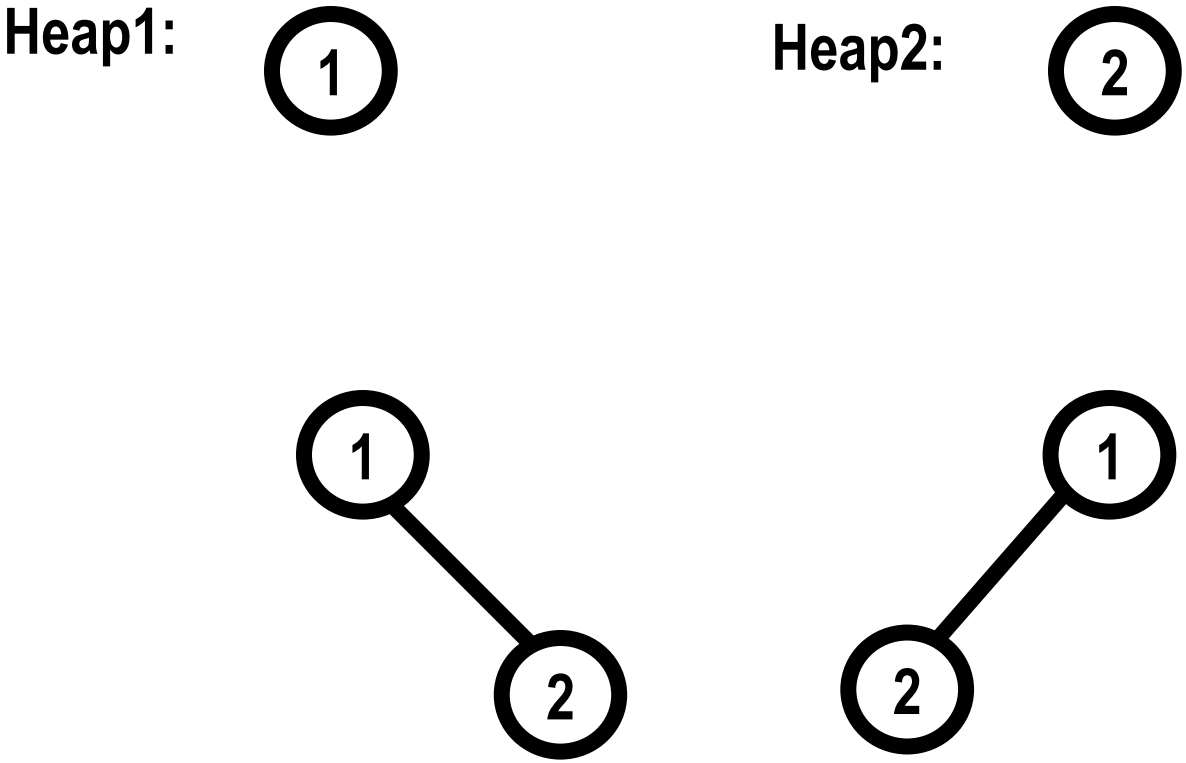
- How?
  - **Merge** "**Blindly**" **adjusting** version of leftist heaps **for future operations**!
  - **Merge always swaps** children when fixing right path.

Skew Heaps

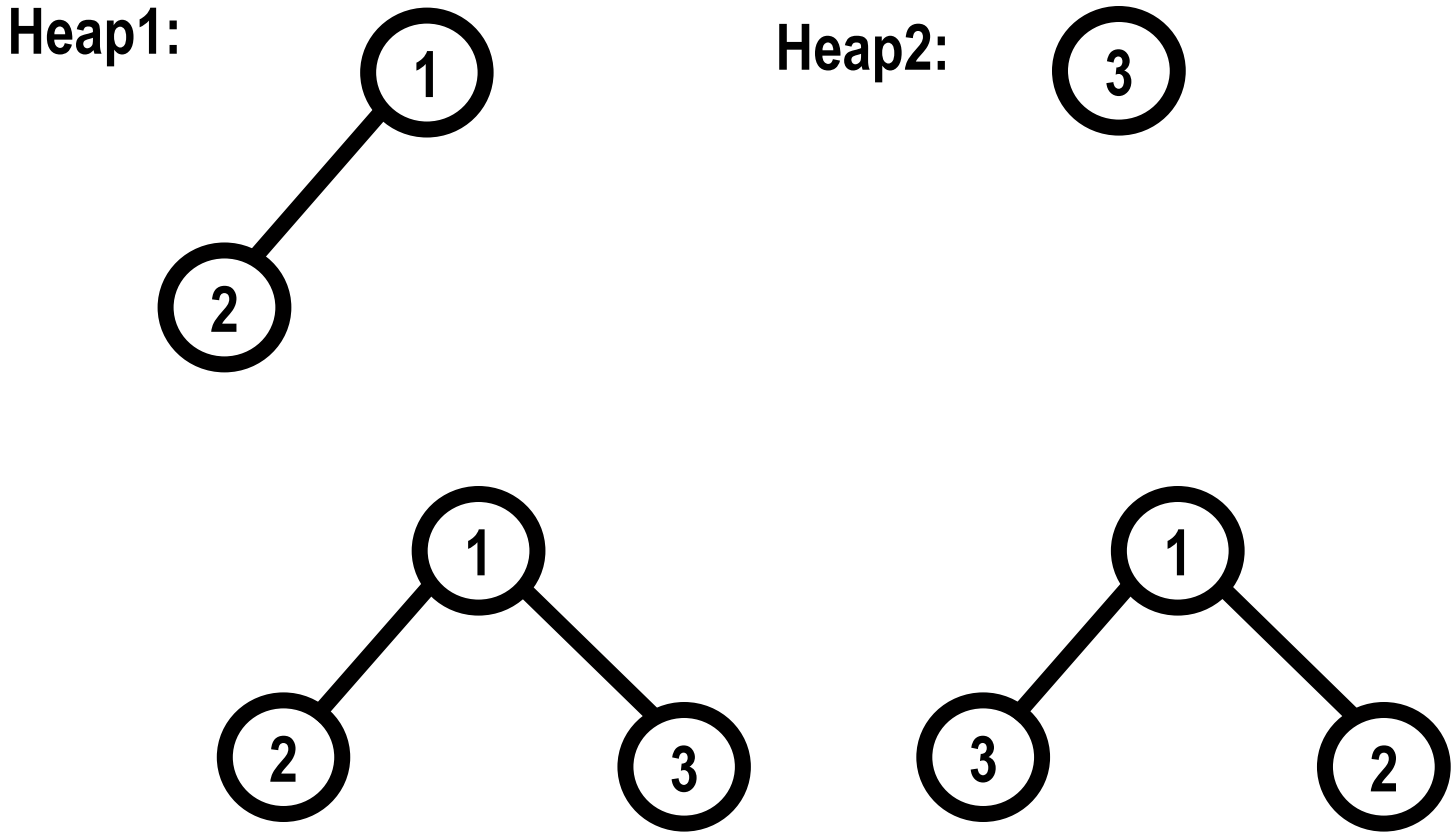
- No worst case guarantee on right path length!
- Worst case time: merge, insert, deleteMin = O(n) time

- BUT,
- **Amortized time**: merge, insert, deleteMin= **O(log n)** time

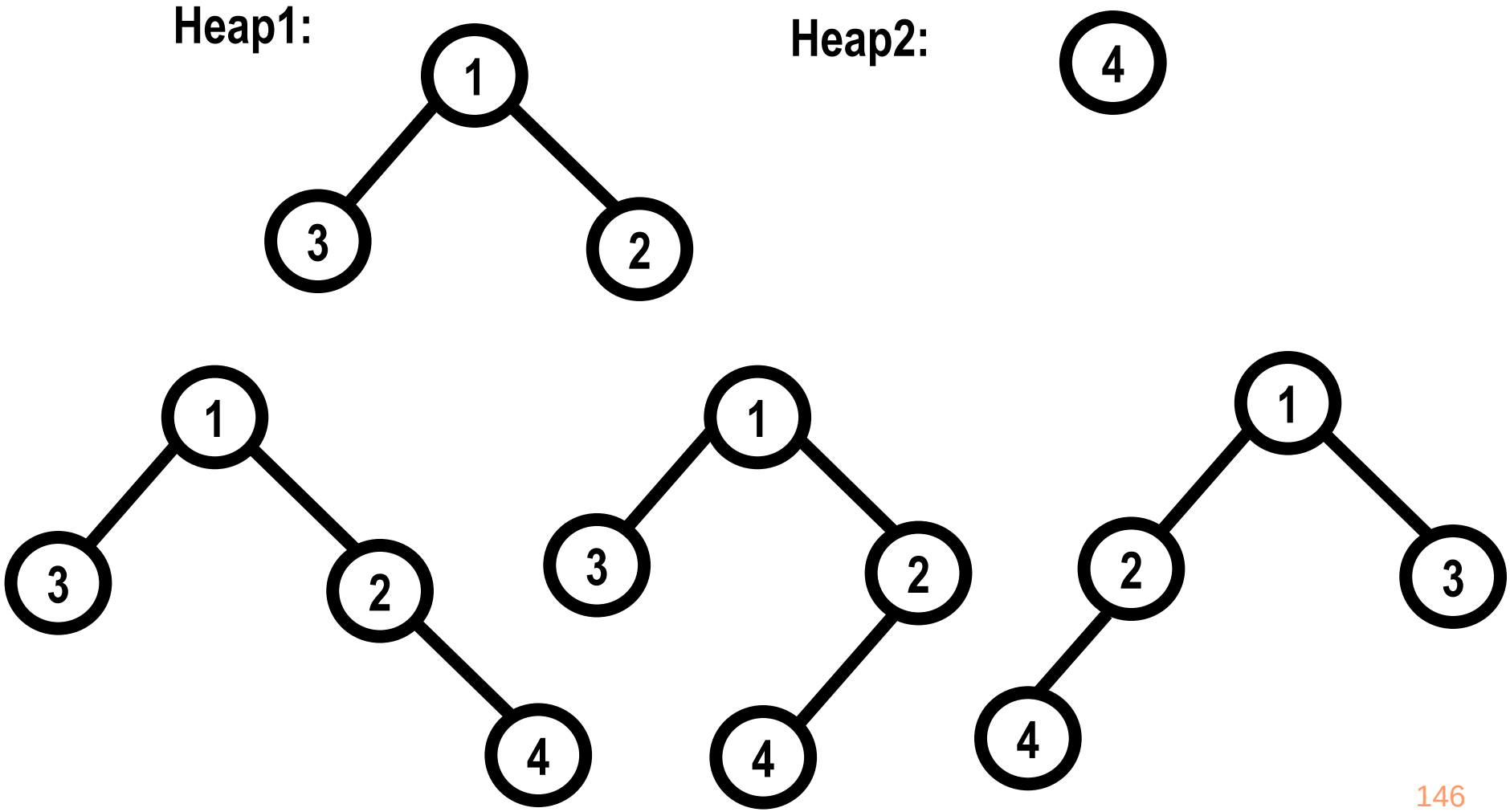
Merging Two Skew Heaps



Merging Two Skew Heaps

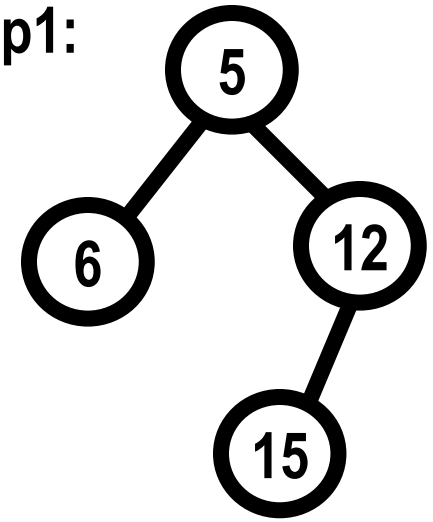


Merging Two Skew Heaps

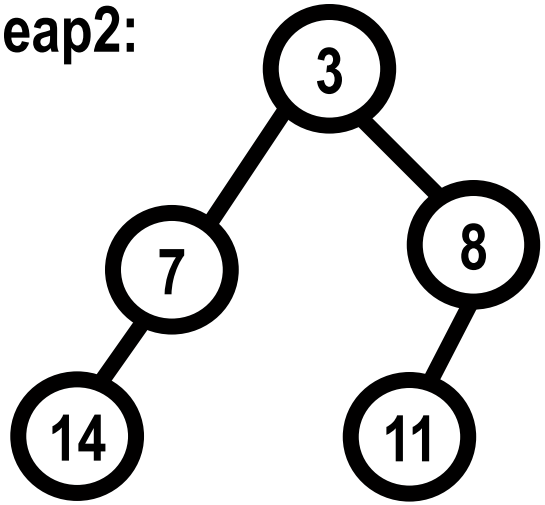


►QUIZ? Merging Two Skew Heaps?

Heap1:



Heap2:



▶QUIZ?

- Insert 5, 3, 4, 1 & 2 into an empty skew min-heap?

Implementation of Skew Heaps

```
template <class DT>
class SkewHeap;

template <class DT>
class LeftistNode
{
   DT   element;
   LeftistNode *left;
   LeftistNode *right;
   int   spl;

   LeftistNode( const DT & theElement, LeftistNode *lt = NULL,
                LeftistNode *rt = NULL, int np = 1 )
     : element( theElement ), left( lt ), right( rt ), spl( np ) { }

   friend class LeftistHeap<DT>;
};
```

Merge2

```
/**
 * Merge two not empty roots h1 and h2.
 * h1's root contains smallest item.
 */
template <class DT>
LeftistNode<DT> *
LeftistHeap<DT>::merge2( LeftistNode<DT> * h1,
                                    LeftistNode<DT> * h2 ) const
{
    // Recursively merge its right subtree and the other tree h2.
    h1->right = merge1( h1->right, h2 );
    // Swap if needed.
    if( h1->left->spl < h1->right->spl )
      swapChildren( h1 );
    // Update the spl of the merged root.
    h1->spl = h1->right->spl + 1;
    return h1;
}
```

Skew Heaps

- Skew heaps, in contrast to leftist heaps, use less space and are easier to implement.
- The **amortized time** of an insert, deletemin, or merge/meld/union skew heap operation is **O(log n).**

Skew Heap History

- Sleator,Tarjan, "**Self-Adjusting Heaps**," SIAM J. Computing, 15(1), 1986, pp. 52-69.

Skew Heap Visualization

- ***Skew Heap Visualization***

►QUIZ?

- Compare Leftist Heap with Skew Heap?

Binomial Heaps

Fibonacci Heaps

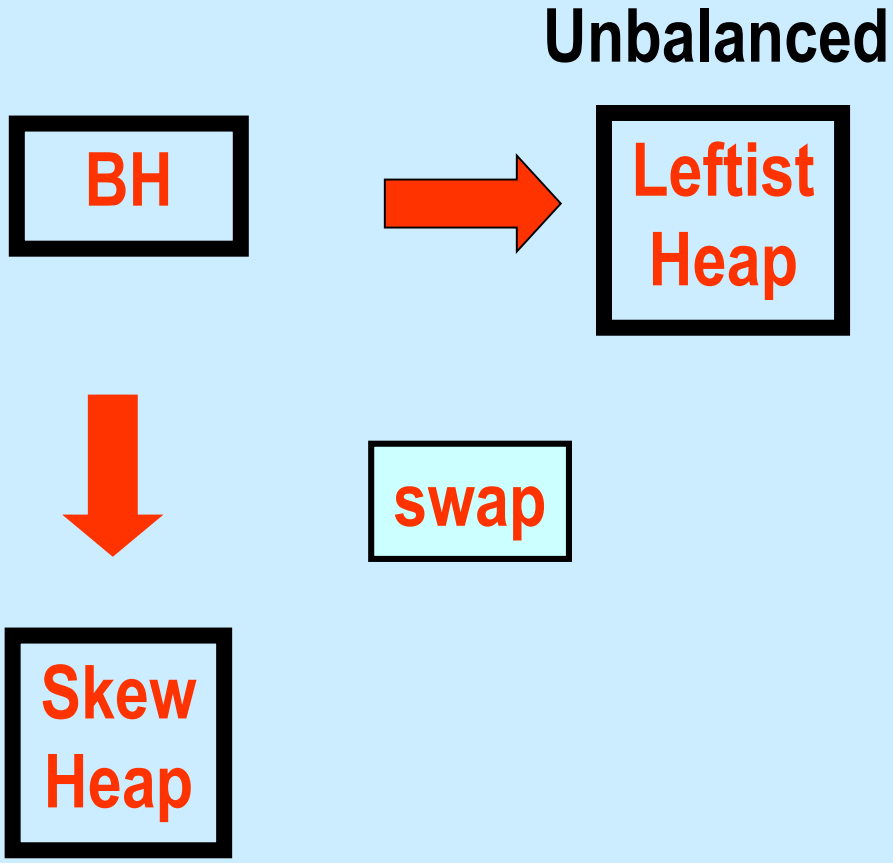
(Mergeable/Meldable Heaps)

Binomial & Fibonacci Heaps

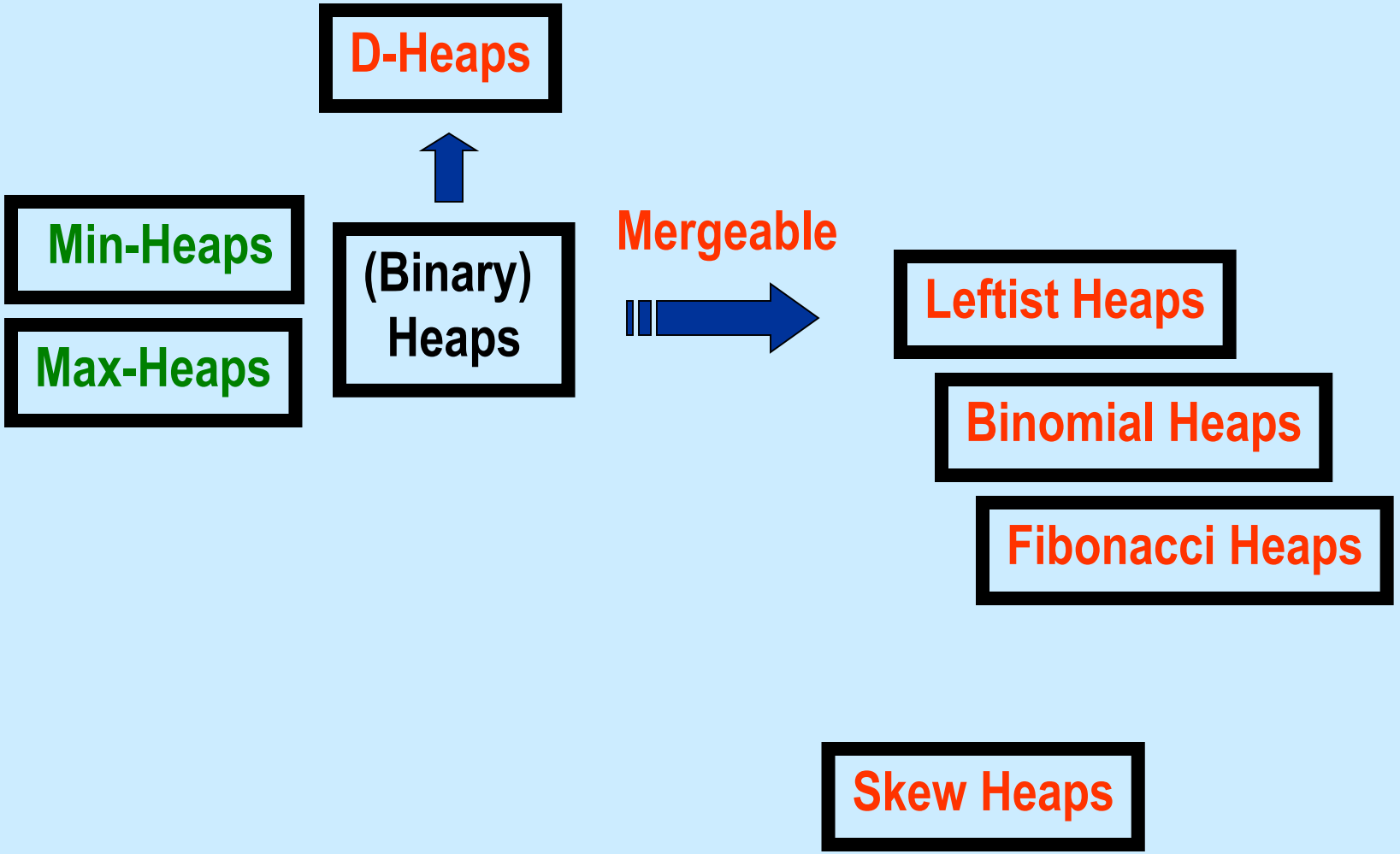
- Binomial Heaps
  - ➔Binomial trees
- Fibonacci Heaps
  - ➔Fibonacci Trees

  - ➔Supports quickly **merging** two heaps.
  - ➔**Mergeable/ Meldable** Heaps
  - ➔O(log n) worst time ~ O(1) amortized time

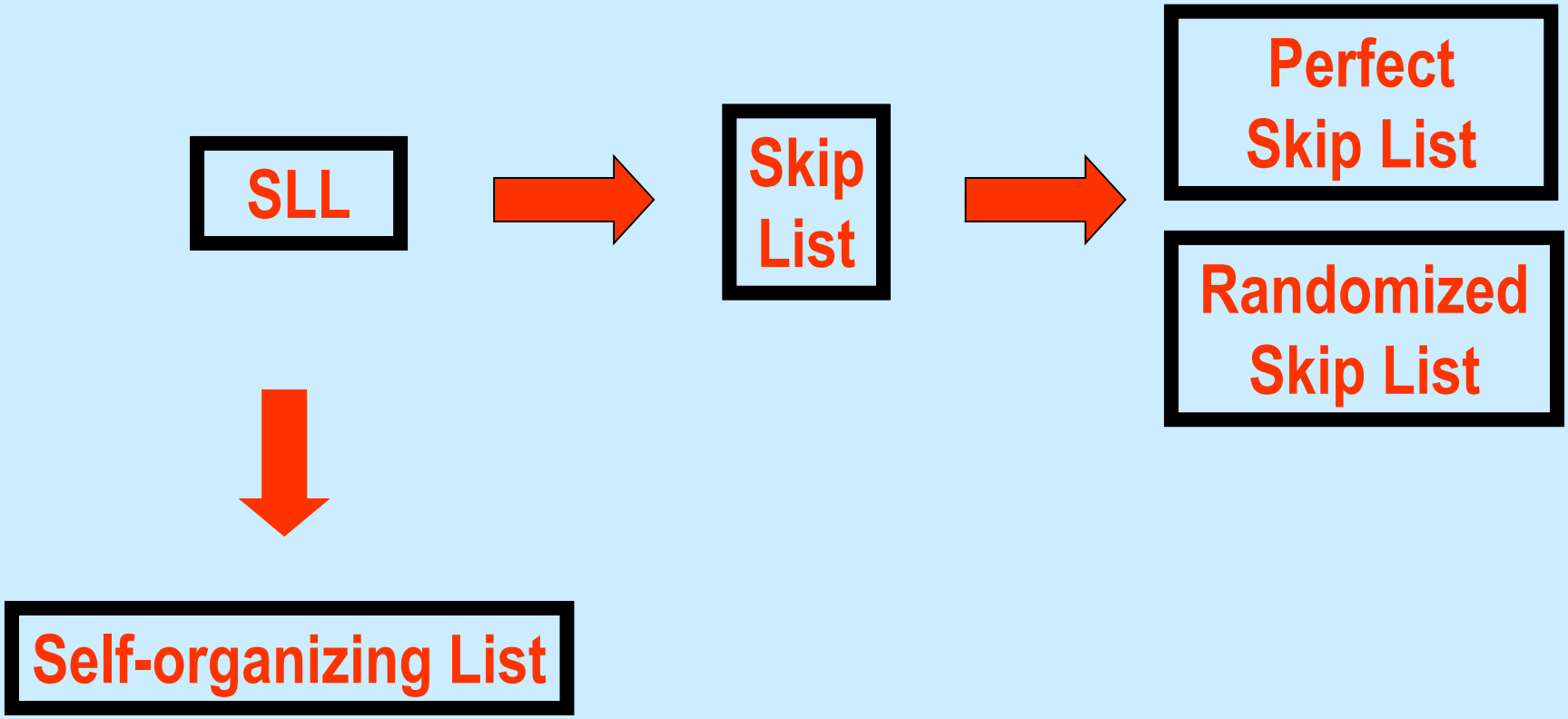
Binary Heaps, Leftist Heaps & Skew Heaps



The Wide World of Heaps



SLLs, Skip Lists & Self-organizing Lists



Homework Assignment

►Homework Assignment?

- What is the worst-case time complexity of the **top-down heap building**?
- What is the worst-case time complexity of the **bottom-up heap building**? Explain.

►Homework Assignment?

- Draw the **leftist min-heap** that results when you insert items with the keys 77, 22, 9, 68, 16, 34, 13 and 8 in that order into an initially empty leftist min-heap.
- Draw the **skew min-heap** that results when you insert items with the keys 77, 22, 9, 68, 16, 34, 13 and 8 in that order into an initially empty skew min-heap.

►Homework Assignment?

- ***Advanced Heap*: The Leftist Min-Heap**
- Design and implement the Leftist Heap ADT.

- You should include ***merge, insert, deleteRoot, showLH*** and ***showSPL***.
  - The operation ***showLH*** prints the *priority* values in the leftist heap as rotated counterclockwise 90 degrees from its conventional orientation using a "reverse" inorder tree traversal.
  - The operation ***showSPL*** prints the *spl* values in the leftist heap as rotated counterclockwise 90 degrees from its conventional orientation using a "reverse" inorder tree traversal.

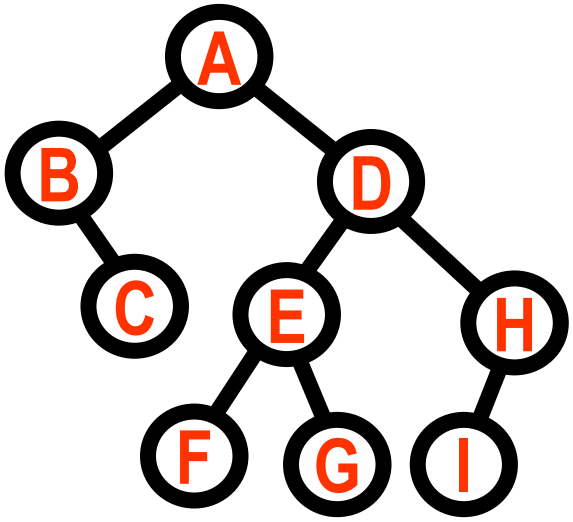
Test Cases

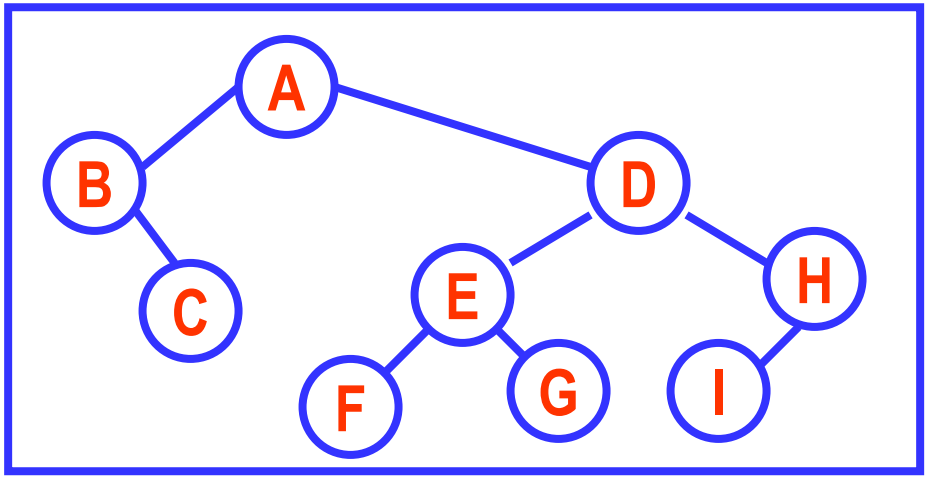
- **LH1: Insert 5, 6, 12 and 15. LH2: Insert 7, 14, 3, 8, and 11. Merge LH1 with LH2.**
- **LH3: Insert 77, 22, 9, 68, 16, 34, 13 and 8. DeleteRoot (LH3) & DeleteRoot (LH3).**

Printing Leftist Heaps?

- **Leftist Heaps & Skew Heaps** are special Binary Trees!

Printing Binary Trees?



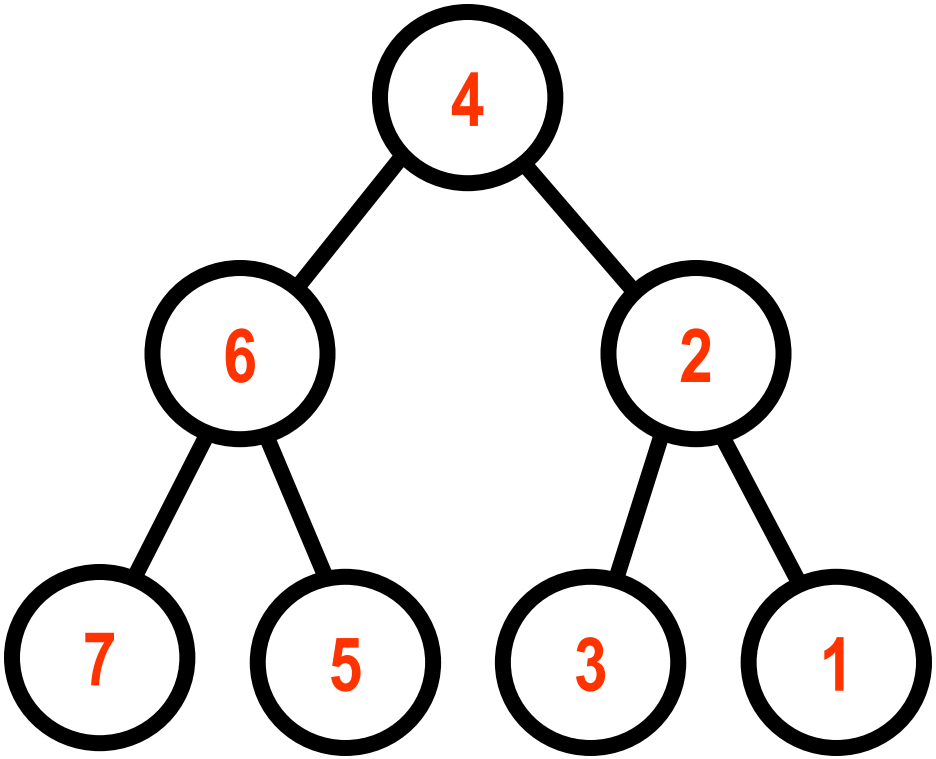


- **Idea?**
  - **Backward In-order Traversal & Print**

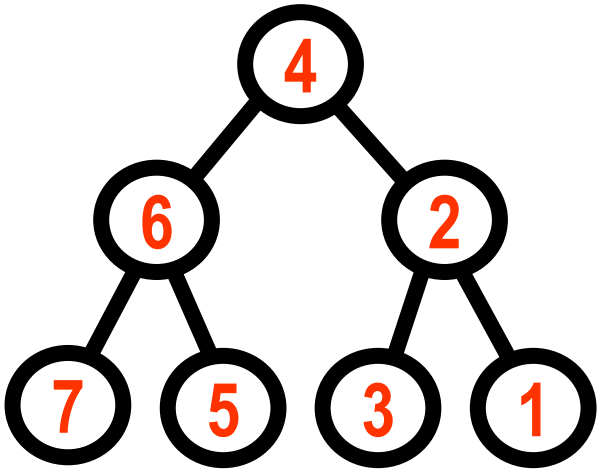
Backward In-order Traversal of Binary Trees

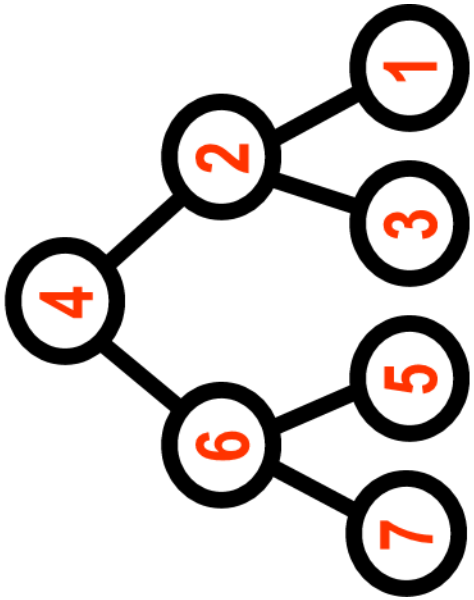
- Process right-child subtree, then a node, then left-child subtree.
- If the tree is not empty then
  - ➔Backward Inorder traverse the right subtree recursively.
  - ➔Visit the root & process!
  - ➔Backward Inorder traverse the left subtree recursively.

Backward In-order Traversal - Processing Order

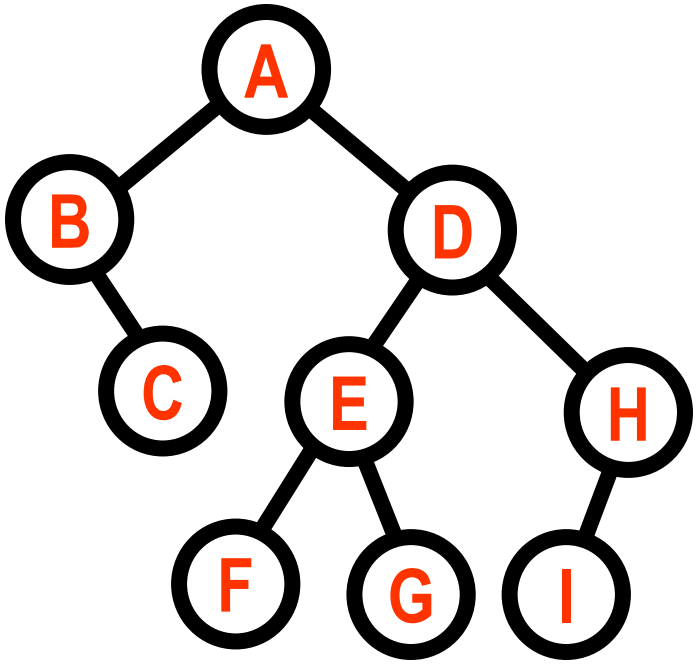


Backward In-order Traversal - Processing Order



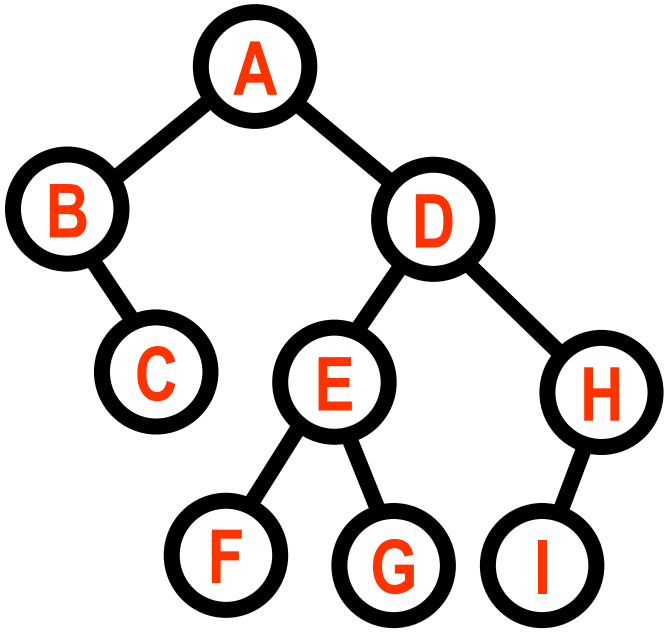


Example: Backward Inorder Traversal & Printing



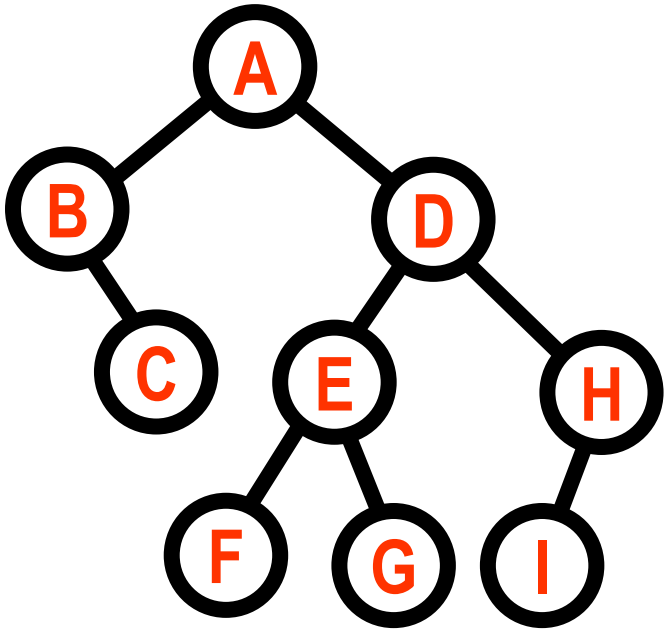
H I D G E F A C B

Example: Backward Inorder Traversal & Printing

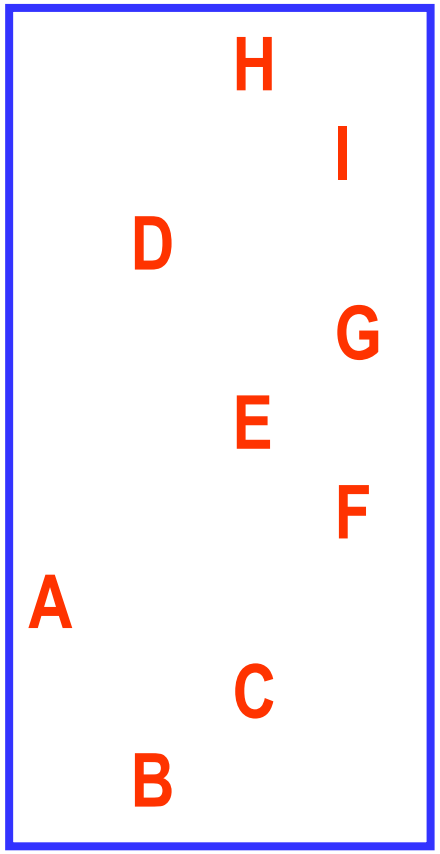


H
I
D
G
E
F
A
C
B

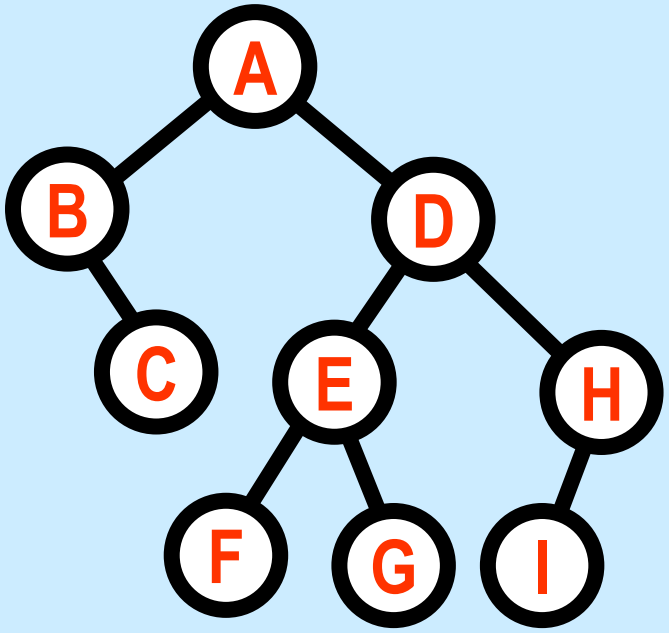
Example: Backward Inorder Traversal & Printing



H
I
D
G
E
F
A
C
B



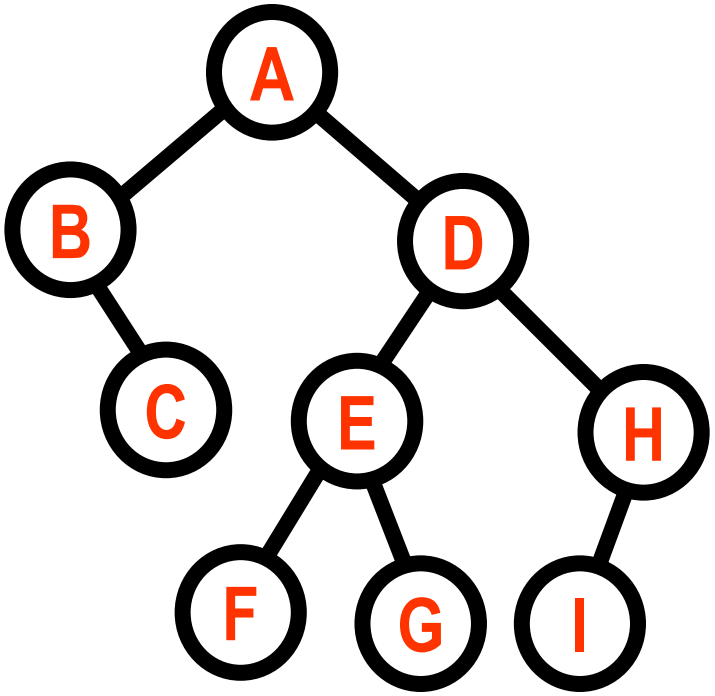
Example: Backward Inorder Traversal & Printing

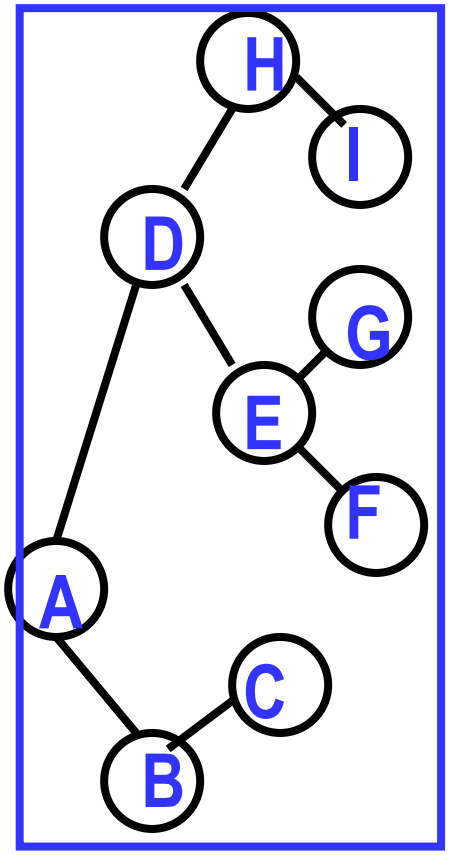



```
H
I
D
G
E
F
A
C
B
```

```
          H \
              I
    D <
              G
          E <
              F
A <
          C
    B /
```

Example: Backward Inorder traversal & Printing





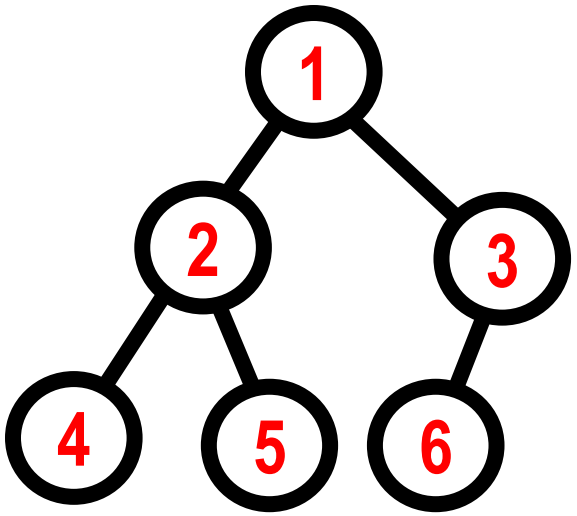
PrintBTree

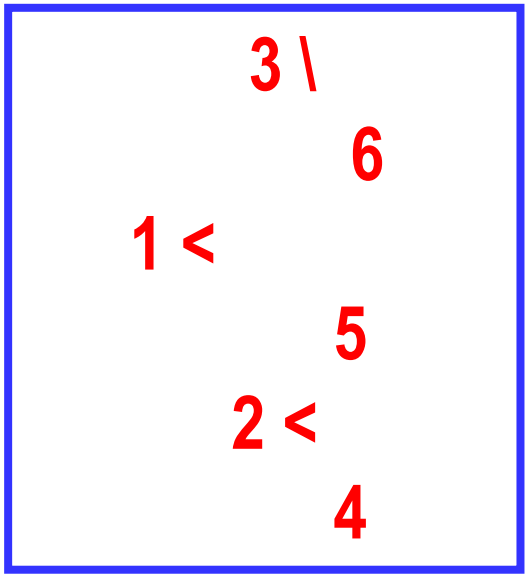
```
void printBTree() const

// Outputs the keys in a binary tree.
// The tree is output rotated counterclockwise 90 degrees
// using a "reverse" inorder traversal.

{
   if ( root == 0 )
      cout << "Empty tree" << endl;
   else
   {
      cout << endl;
      printBTreeHelper(root,1);
      cout << endl;
   }
}
```

PrintBTreeHelper

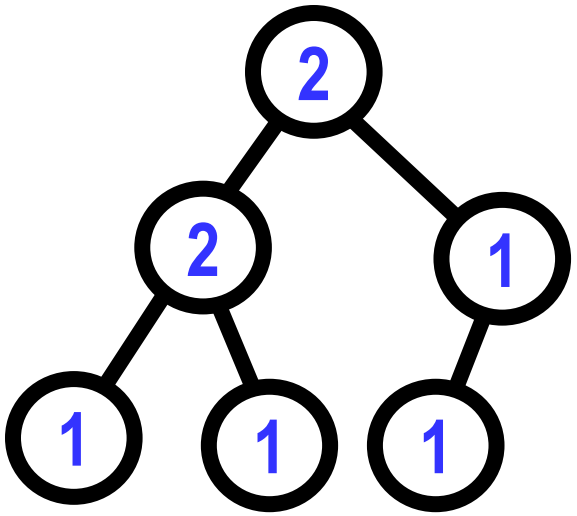
```
void printBTreeHelper( BTreeNode *p, int level ) const

// Recursive helper for printBTree.
// Outputs the subtree whose root node is pointed to by p.
// level is the level of this node within the tree.
{
     int j;
     if ( p != 0 )
     {
        printBTreeHelper(p->right,level+1);        // Output right subtree
        for ( j = 0 ; j < level ; j++ ) cout << "\t";
        cout << " " << p->Key;   // Output key
        if ( ( p->left != 0 ) && ( p->right != 0 ) ) cout << "<";
        else if ( p->right != 0 ) cout << "/";
        else if ( p->left != 0 ) cout << "\\";
        cout << endl;
        printBTreeHelper(p->left,level+1);         // Output left subtree
     }
}
```

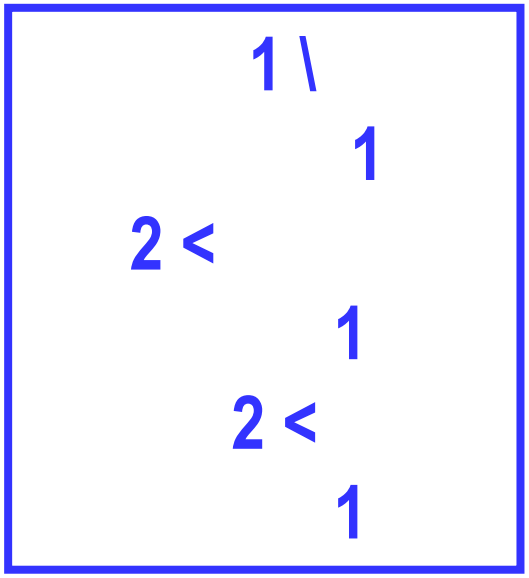
►QUIZ? Printing Leftist Min-Heap Priority Values?





►QUIZ? Printing Leftist Min-Heap SPLs?





the right majors, minors &

ucation?

or students' academic and career success

ging for many students - *Many students change their during college!*

e prediction of student success in MMC could dual students

d their right MMC

chieve their academic goals

END

Prof. Young Park