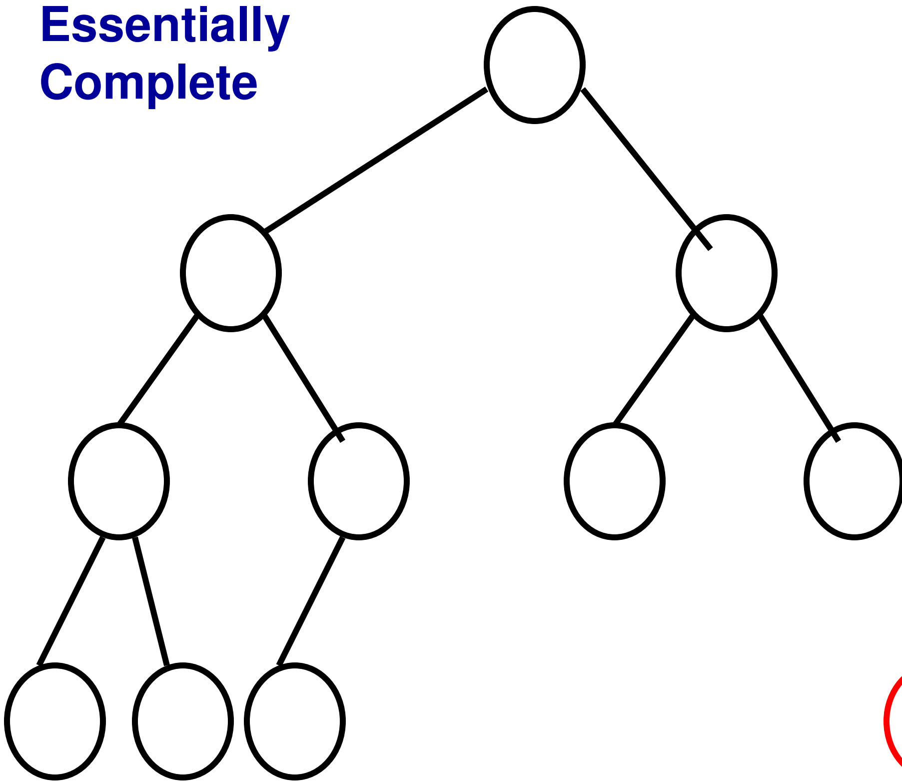
Module 8: Heap

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

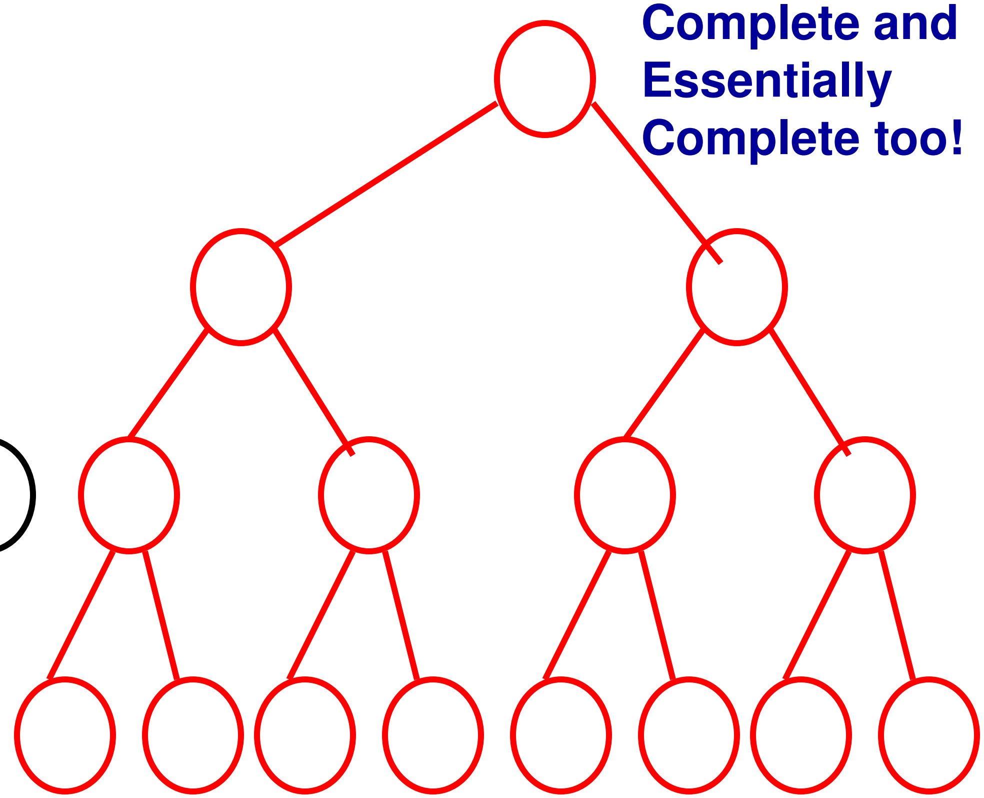
Essentially Complete Binary Tree

- A binary tree of height ‘h’ is essentially complete if it is a complete binary tree up to level h-1 and the nodes at level h are as far to the left as possible.
- Note: A complete binary tree is also essentially complete.

Essentially Complete

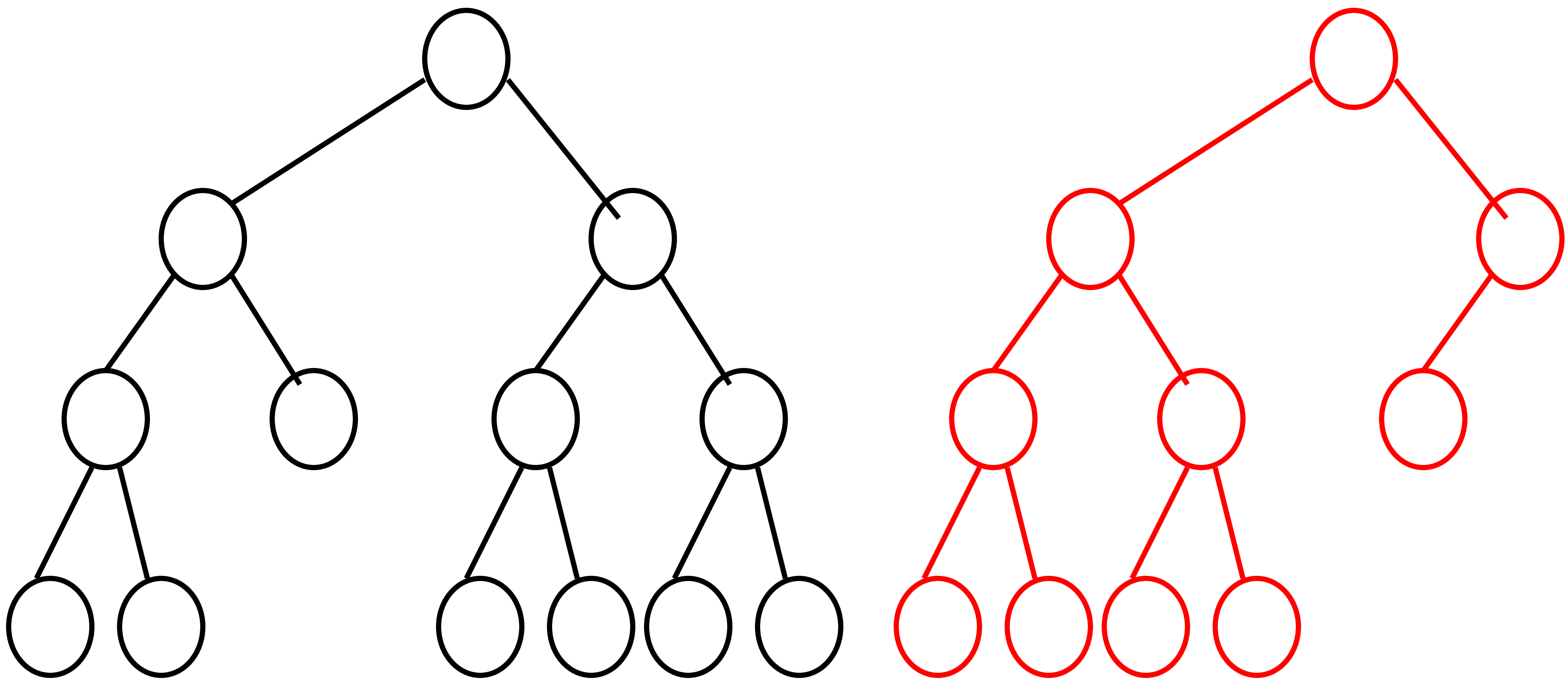


Complete and Essentially Complete too!



Essentially Complete Binary Tree

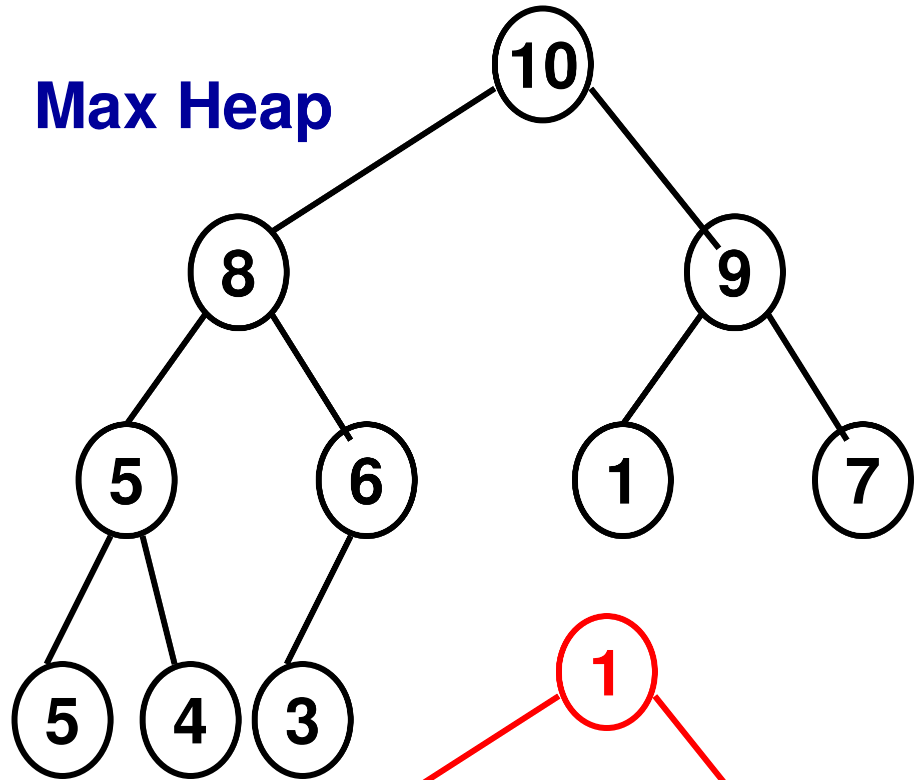
- The trees shown below are not essentially complete.

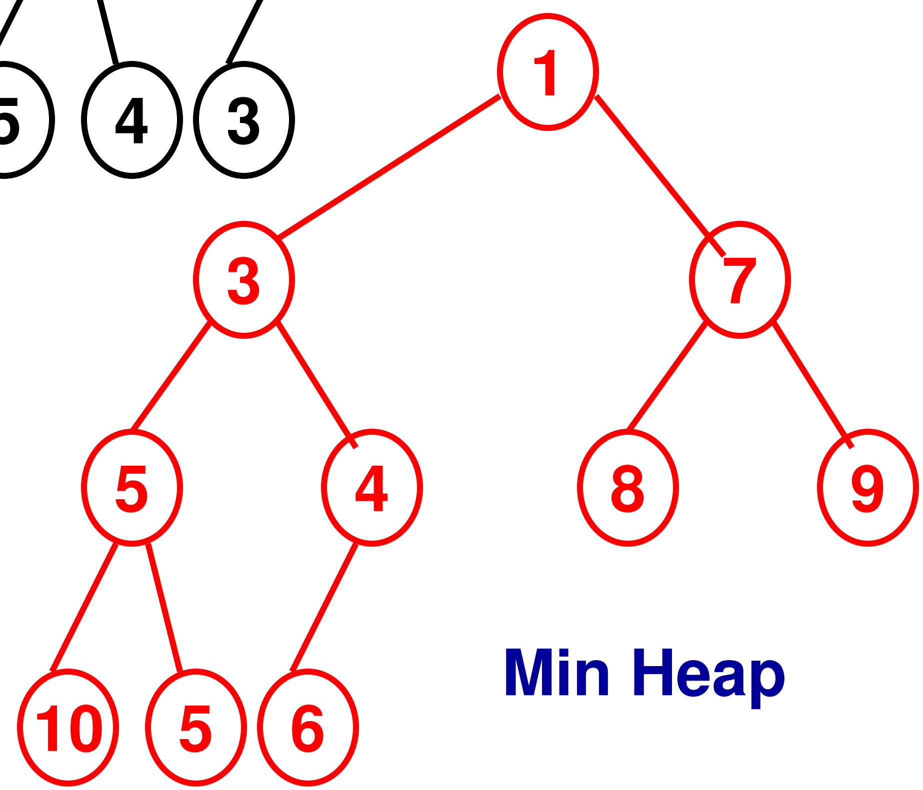


Heap

- A heap is a binary tree that satisfies the following two properties:
  - Essentially complete or complete
  - Max/Min heap
    - Max heap: The data at each internal node is greater than or equal to the data of its immediate child nodes
    - Min heap: The data at each internal node is lower than or equal to the data of its immediate child nodes

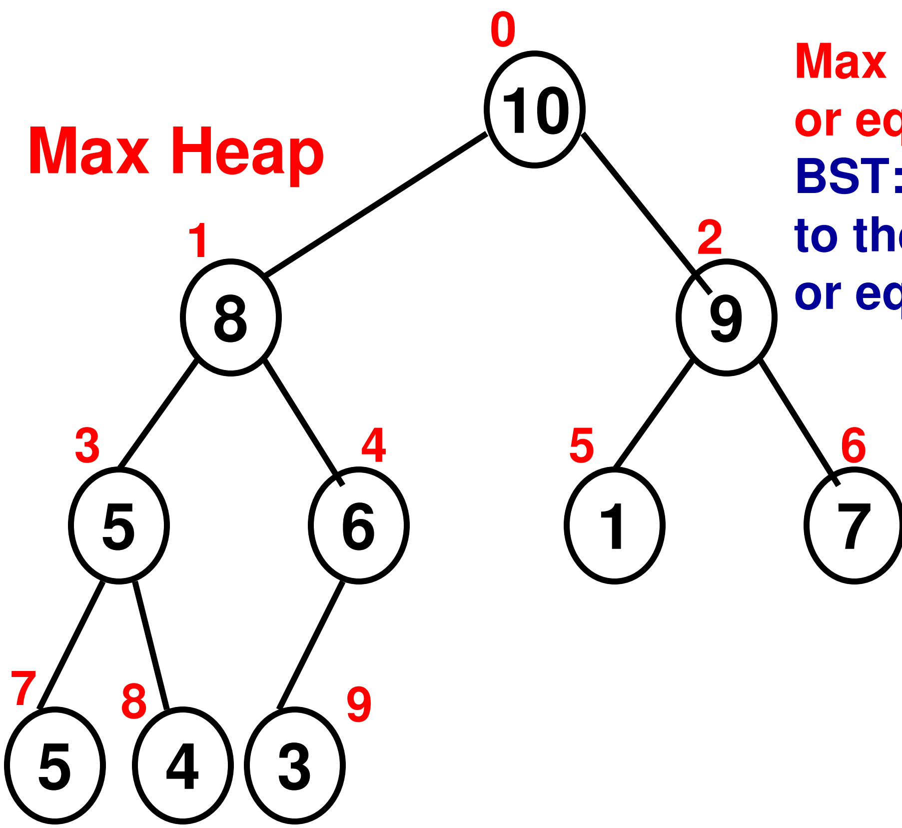
Max Heap





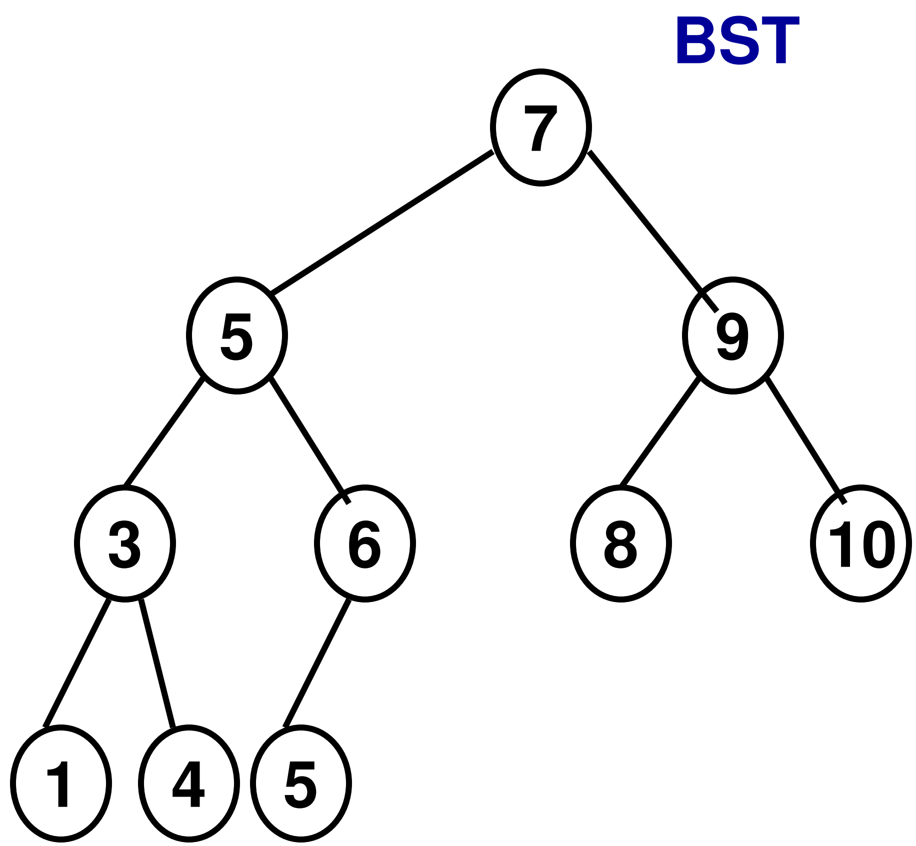
Min Heap

Difference between BST and Heap



Max Heap: Each internal node is greater than or equal to its child nodes.

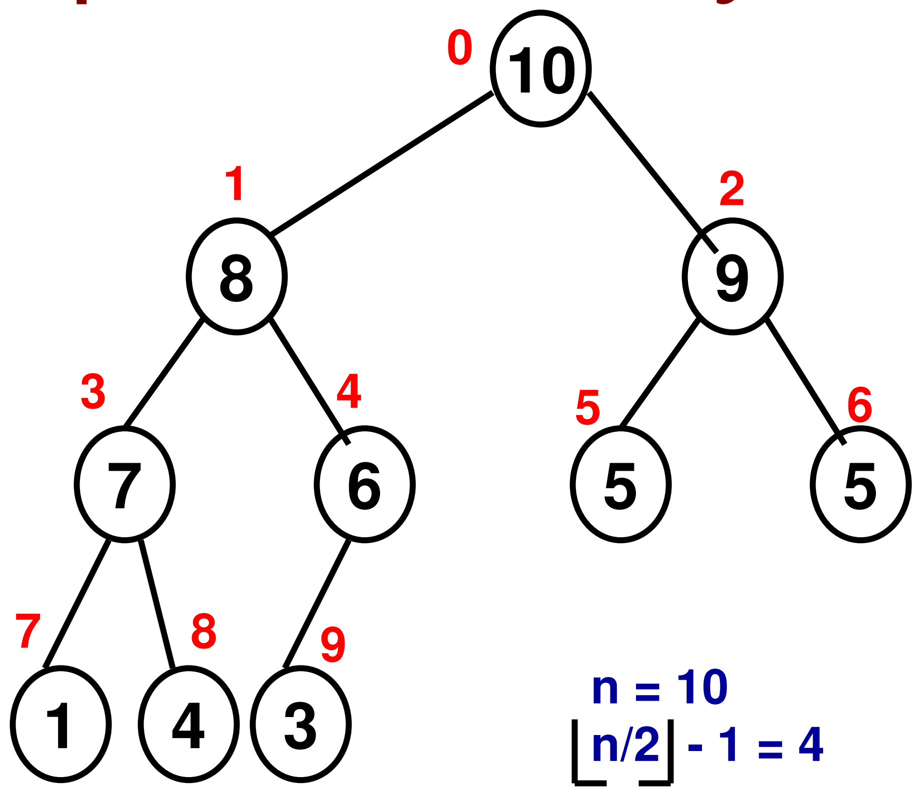
BST: An internal node is greater than or equal to the nodes in its left sub tree and lower than or equal to the nodes in its right sub tree.



7	3	8	1	9	4	0	5	2	6
5	5	4	8	3	6	10	1	9	7
1	3	4	5	5	6	7	8	9	10

Storing the Heap as an Array

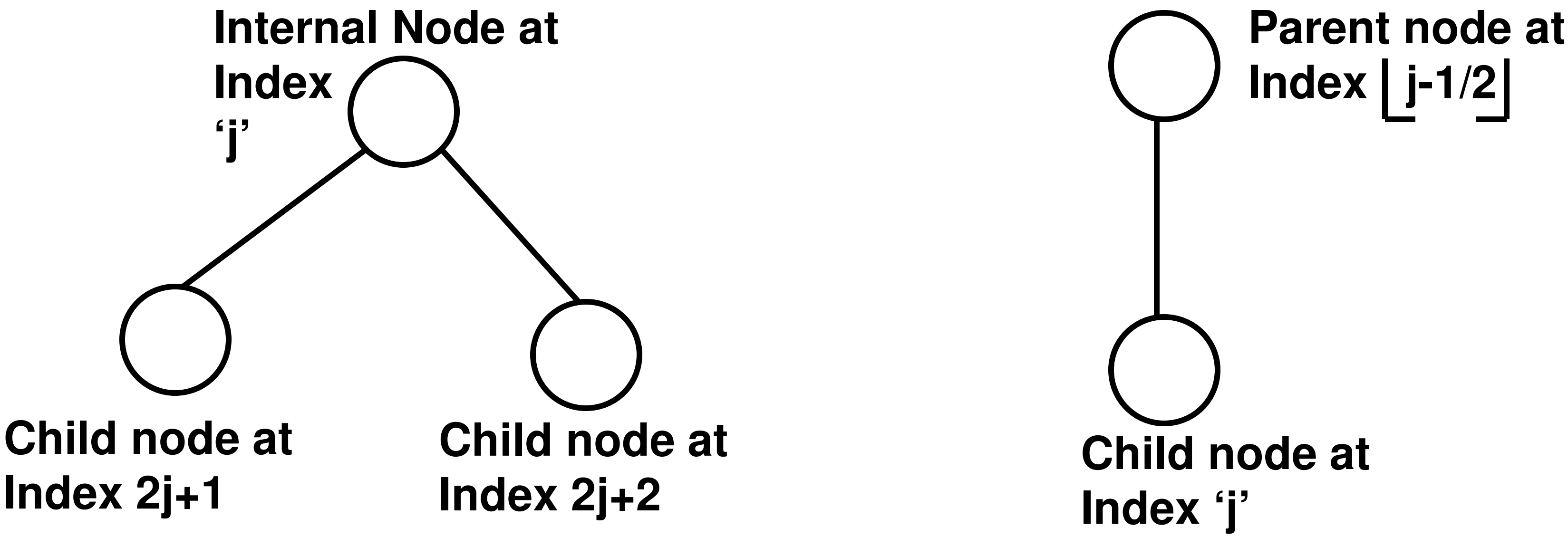
- A heap of ‘n’ elements can be stored in an array (index starting from 0) such that the internal nodes (in the top-down, left-right order) are represented as elements from index 0 to $\lfloor n/2 \rfloor - 1$ and the leaf nodes (again, top-down, left-right order) are represented as elements from index $\lfloor n/2 \rfloor$ to n-1.
- The child nodes of an internal node at index ‘j’ are at indexes 2j+1 and 2j+2.
- The parent node for a node at index j is at index $\lfloor (j-1)/2 \rfloor$



0	1	2	3	4	5	6	7	8	9
10	8	9	7	6	5	5	1	4	3

The child nodes of internal node ‘8’ at index 1 are at indexes 2*1+1 = 3 and 2*1 + 2 = 4. The parent node for node ‘7’ at index 3 is at index $\lfloor (3-1)/2 \rfloor = 1$

Storing the Heap as an Array



For the rest of this module, we will construct and employ a ‘max’ heap unless otherwise specified.

The data for an internal node must be greater than or equal to that of its child nodes.

Using BFS to check whether a Binary Tree is Essentially Complete

The moment we come across an internal node with a missing Child node (left node or right node), we set the boolean 'noChildZoneStarts' to true.

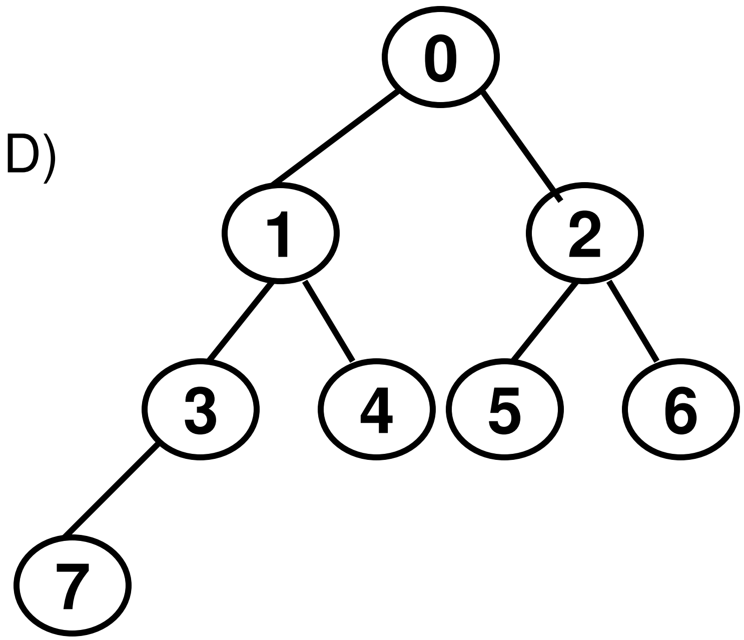
If we come across an internal node with a child node when the noChildZoneStarts boolean is already true, we declare the tree is not essentially complete!

```
Queue queue
queue.enqueue(root node id 0)
noChildZoneStarts = false

Begin BFS_BinaryTree
while (!queue.isEmpty()) do
  FirstNodeID = queue.dequeue();
  if (noChildZoneStarts == false AND FirstNode.leftChildNodeID == -1)
        noChildZoneStarts = true
  else if (noChildZoneStarts == true AND FirstNode.leftChildNodeID != -1)
        return "the binary tree is not essentially complete"

  if (FirstNode.leftChildNodeID != -1) then
     queue.enqueue(FirstNode.leftChildNodeID)
  end if
```

noChildZoneStarts	LeftChild	
True	!= -1 (Exists)	Not essentially complete
True	== -1 (Does not exist)	OK (continue)
False	!= -1 (Exists)	OK enqueue the left child node
False	== -1 (Does not exist)	No child zone has begun

Using BFS to check whether a Binary Tree is Essentially Complete

```
if (noChildZoneStarts == false AND FirstNode.rightChildNodeID == -1)
        noChildZoneStarts = true
else if (noChildZoneStarts == true AND FirstNode.rightChildNodeID != -1)
        return "the binary tree is not essentially complete"

if (FirstNode.rightChildNodeID != -1) then
   queue.enqueue(FirstNode.rightChildNodeID)
end if

end while

return "the binary tree is essentially complete"

End BFS_BinaryTree
```



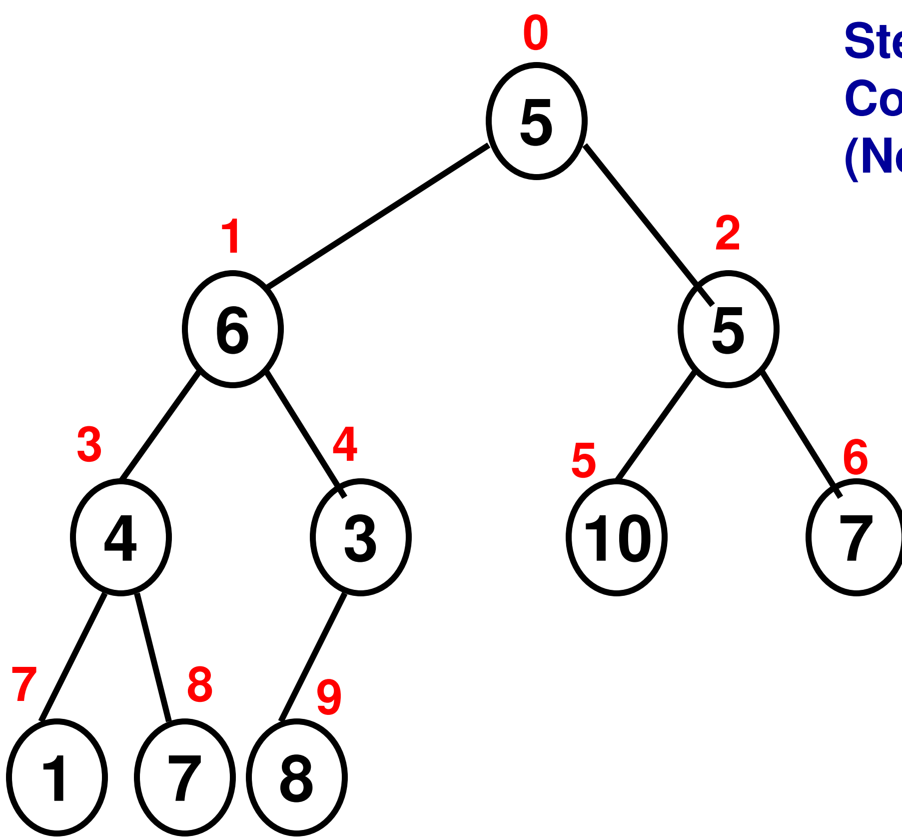
Once we find out that node '3' does not have a right child, all the nodes explored further in BFS should not have any child node. Otherwise, the binary tree is not essentially complete.

Heap Construction

- Given an array of 'n' elements,
- Step 1: Construct an essentially complete binary tree and then reheapify the internal nodes of the tree to make sure the max or min heap property is satisfied for each internal node.
- Step 2: Reheapify an internal node for 'max' heap: If the data at an internal node is lower than that of one or both of its child nodes, then swap the data for the internal node with the larger of the data of its two child nodes.
  - If any internal node further down is affected because of this swap, the reheapify operation is recursively continued all the way until a leaf node is reached.
- The reheapify operation is started from the node at index $\lfloor n/2 \rfloor - 1$ and continued all the way to the node at index 0.

Heap Construction Example 1

0	1	2	3	4	5	6	7	8	9
5	6	5	4	3	10	7	1	7	8

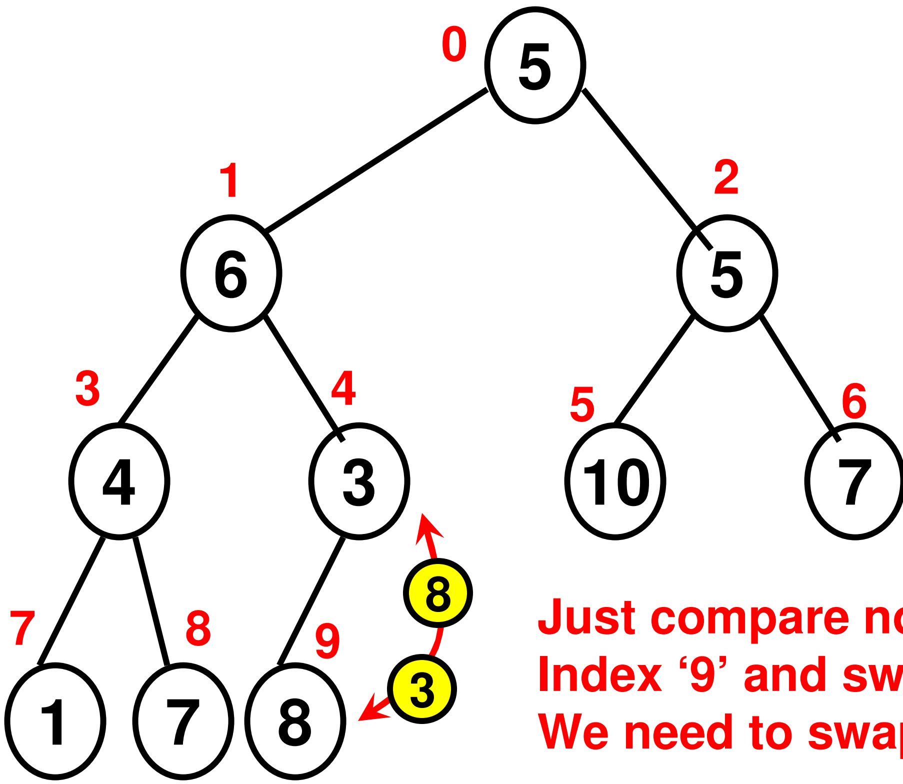


Step 1: Essentially Complete Binary Tree (Not a heap yet!)

Heap Construction Example 1

Before (Reheapify at Index '4'):

0	1	2	3	4	5	6	7	8	9
5	6	5	4	3	10	7	1	7	8



Step 2: Reheapify node at index '4' and down further if needed

Compare the node at index '4' with its child nodes at index 2*4 + 1 = 9 and index 2*4 + 2 = 10. Since index '10' does not exist and index 9 exists, it implies we have reached a leaf node (at index 9) and there is no need to proceed further down.

Just compare node at index '4' with the child node at Index '9' and swap them, if needed. In this case: Yes, We need to swap.

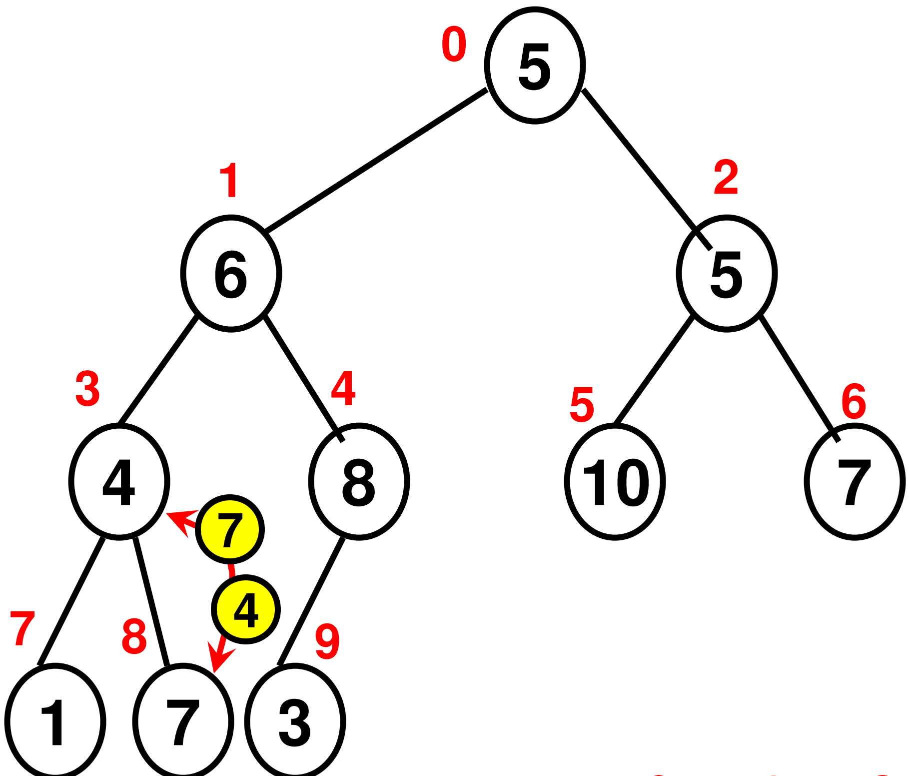
After (Reheapify at Index '4'):

0	1	2	3	4	5	6	7	8	9
5	6	5	4	8	10	7	1	7	3

Heap Construction Example 1

Before (Reheapify at Index '3'):

0	1	2	3	4	5	6	7	8	9
5	6	5	4	8	10	7	1	7	3



Step 2: Reheapify node at index '3' and down further if needed

Compare the node at index '3' with its child nodes at index 2*3 + 1 = 7 and index 2*3 + 2 = 8. In this case, We swap element at index '3' with element at index '8'. Since 8 is already a leaf node, we do not proceed down further.

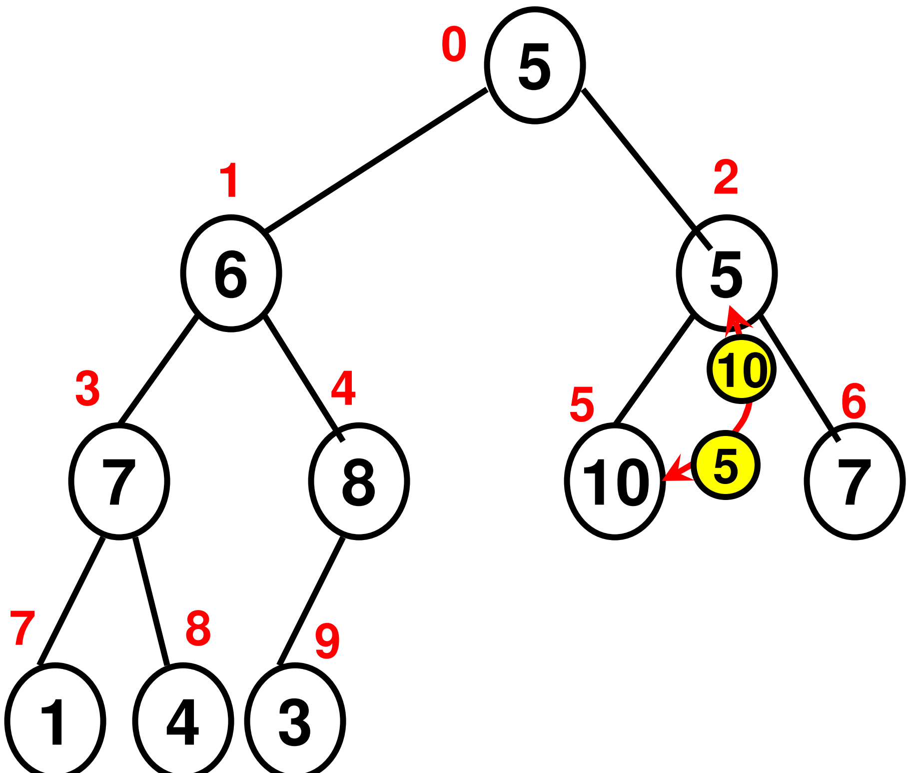
After (Reheapify at Index '3'):

0	1	2	3	4	5	6	7	8	9
5	6	5	7	8	10	7	1	4	3

Heap Construction Example 1

Before (Reheapify at Index ‘2’):

0	1	2	3	4	5	6	7	8	9
5	6	5	7	8	10	7	1	4	3



Step 2: Reheapify node at index ‘2’ and down further if needed

Compare the node at index ‘2’ with its child nodes at index 2*2 + 1 = 5 and index 2*2 + 2 = 6. In this case, We swap element at index ‘2’ with element at index ‘5’. Since 5 is already a leaf node, we do not proceed down further.

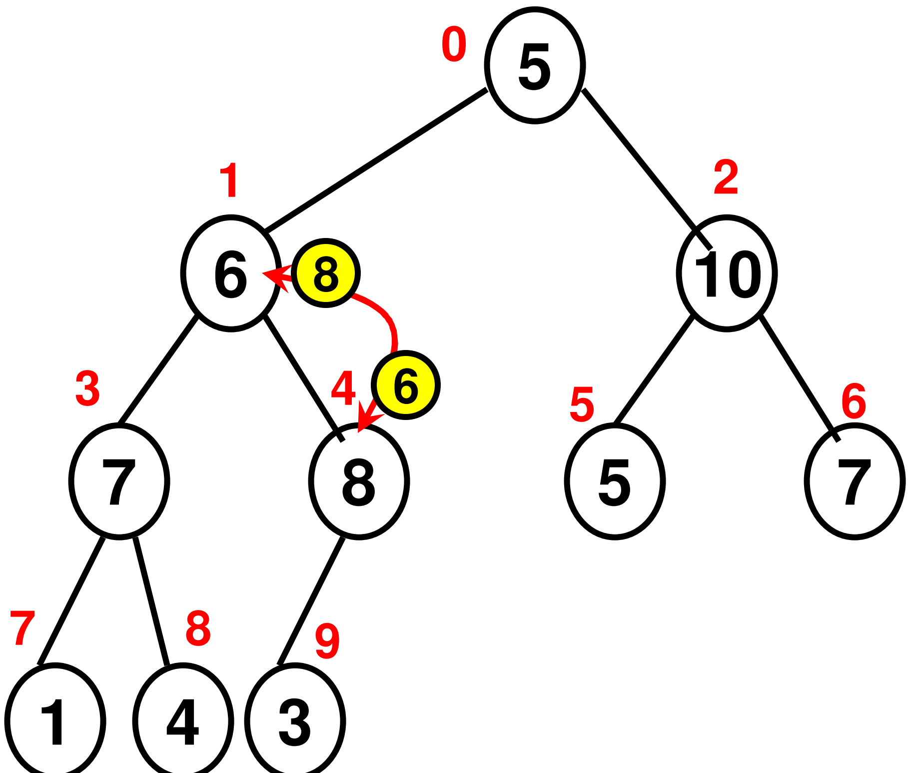
After (Reheapify at Index ‘2’):

0	1	2	3	4	5	6	7	8	9
5	6	10	7	8	5	7	1	4	3

Heap Construction Example 1

Before (Reheapify at Index ‘1’):

0	1	2	3	4	5	6	7	8	9
5	6	10	7	8	5	7	1	4	3



Step 2: Reheapify node at index ‘1’ and down further if needed

Compare the node at index ‘1’ with its child nodes at index 2*1 + 1 = 3 and index 2*1 + 2 = 4. In this case, We swap element at index ‘1’ with element at index ‘4’.
Again do a reheapify at index ‘4’, if needed and continue in a recursive fashion until it is no longer needed.

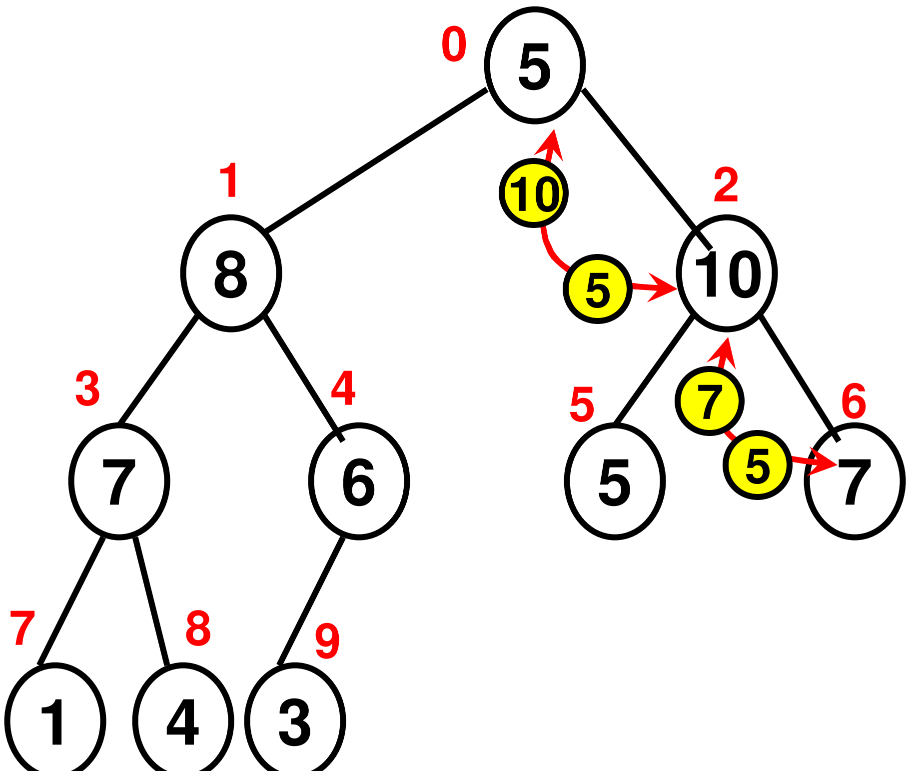
After (Reheapify at Index ‘1’):

0	1	2	3	4	5	6	7	8	9
5	8	10	7	6	5	7	1	4	3

Heap Construction Example 1

Before (Reheapify at Index ‘0’):

0	1	2	3	4	5	6	7	8	9
5	8	10	7	6	5	7	1	4	3



Step 2: Reheapify node at index ‘0’ and down further if needed

Compare the node at index ‘0’ with its child nodes at index 2*0 + 1 = 1 and index 2*0 + 2 = 2. In this case, We swap element at index ‘0’ with element at index ‘2’.
Again do a reheapify at index ‘2’ as the element now at index ‘2’ (which is 5) is lower than the maximum of its two child nodes (which is 9 at index ‘6’).

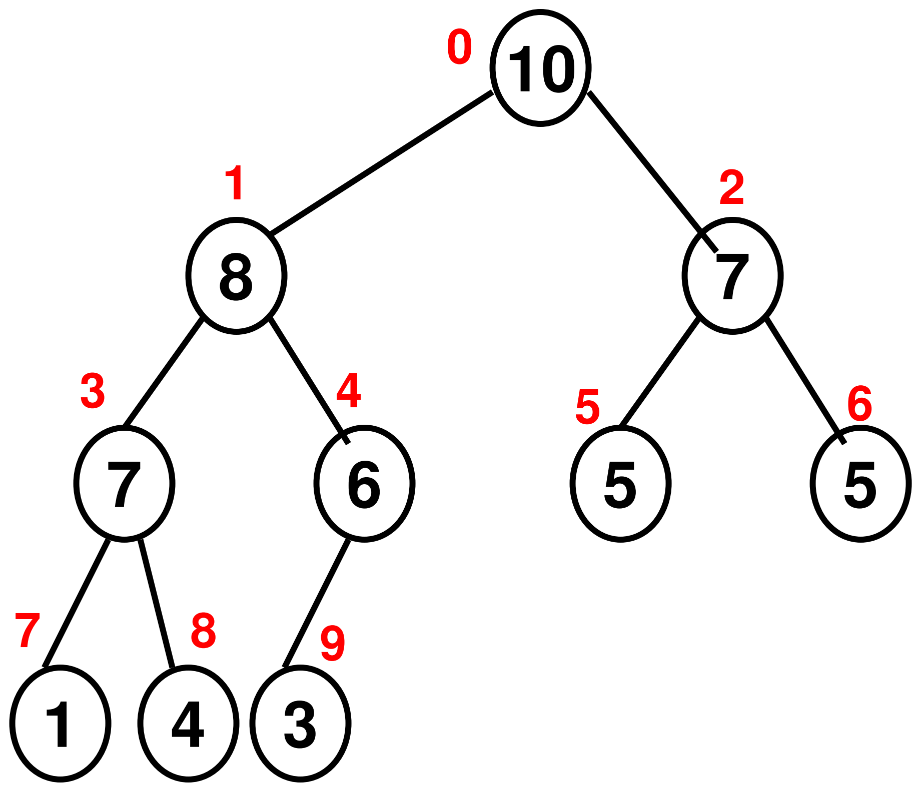
After (Reheapify at Index ‘0’):

0	1	2	3	4	5	6	7	8	9
10	8	7	7	6	5	5	1	4	3

Heap Construction Example 1

Final Array Representing Max Heap

0	1	2	3	4	5	6	7	8	9
10	8	7	7	6	5	5	1	4	3



Max Heap Construction (Code 8.1: C++)

Main Function

```
int arraySize;
cout << "Enter array size: ";
cin >> arraySize;

int array[arraySize];

int maxValue;
cout << "Enter the max. value for any element: ";
cin >> maxValue;

srand(time(NULL));

cout << "Generated array: ";
for (int i = 0; i < arraySize; i++){
    array[i] = rand() % maxValue;
    cout << array[i] << " ";
}
```

```
//max. heap construction

for (int index = (arraySize/2)-1; index  >= 0; index--)
        rearrangeHeapArray(array, arraySize, index);
}


cout << "After Heap construction..." << endl;
for (int index = 0; index < arraySize; index++)
        cout  << array[index] << " ";

cout << endl;
```

7.1: Reheapify Code (C++)

```
void rearrangeHeapArray(int *array, int arraySize, int index){
	// max heap construction

	int leftChildIndex = 2*index + 1;
	int rightChildIndex = 2*index + 2;
			// If the node at 'index' does not have a left child (implies it does
	if (leftChildIndex >= arraySize)	// not have right child too), then there
		return;				// is no need to reheapify at that index

	// If the node at 'index' does not have a right child (if the control reaches
	if (rightChildIndex >= arraySize){	// here, it implies the node
						// at 'index' has a left child)
// Check if the data for the
// node at		if (array[index] < array[leftChildIndex]){
// 'index' is less		int temp = array[index];
// than that of its		array[index] = array[leftChildIndex];
// left child. If so,		array[leftChildIndex] = temp;
// swap
		}

		return;
	}
```

// If the control reaches here, it means the node at 'index' has both left child
// and right child

// If the node at 'index' has data that is greater than or equal to both its left child and right child, then there is no need to reheapify for this index

```
if (array[index] >= array[leftChildIndex] &&
    array[index] >=  array[rightChildIndex])
        return;
```

// If the control reaches here, it implies the node at 'index' has data that is less than at least one of its two child nodes

```
int maxIndex = leftChildIndex;
if (array[leftChildIndex] < array[rightChildIndex])
        maxIndex = rightChildIndex;
```

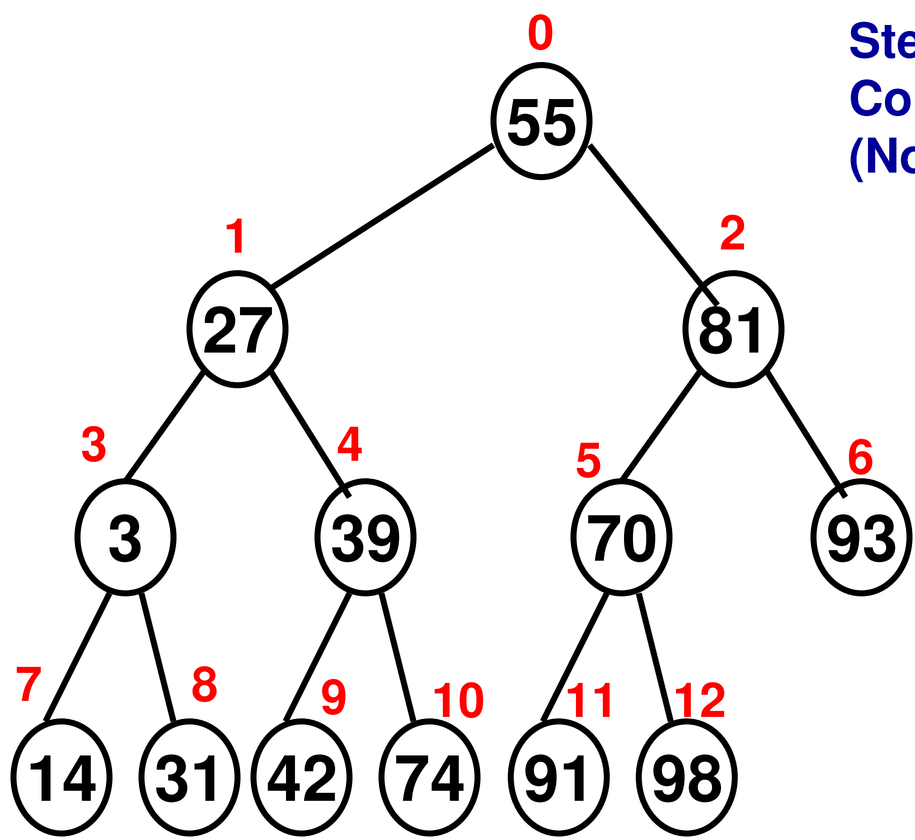
// Between the left and right
// child nodes, find the node
// that has relatively larger
// data, call the index of this
// as 'maxIndex' and swap
// its value with the node at
// 'index'.

```
int temp = array[maxIndex];
array[maxIndex] = array[index];
array[index] = temp;

rearrangeHeapArray(array, arraySize, maxIndex);
```

// Call the rearrangeHeap function in a recursive fashion
// to see if further rearrangements need to be done starting
// from maxIndex

Heap Construction Example 2

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	39	70	93	14	31	42	74	91	98



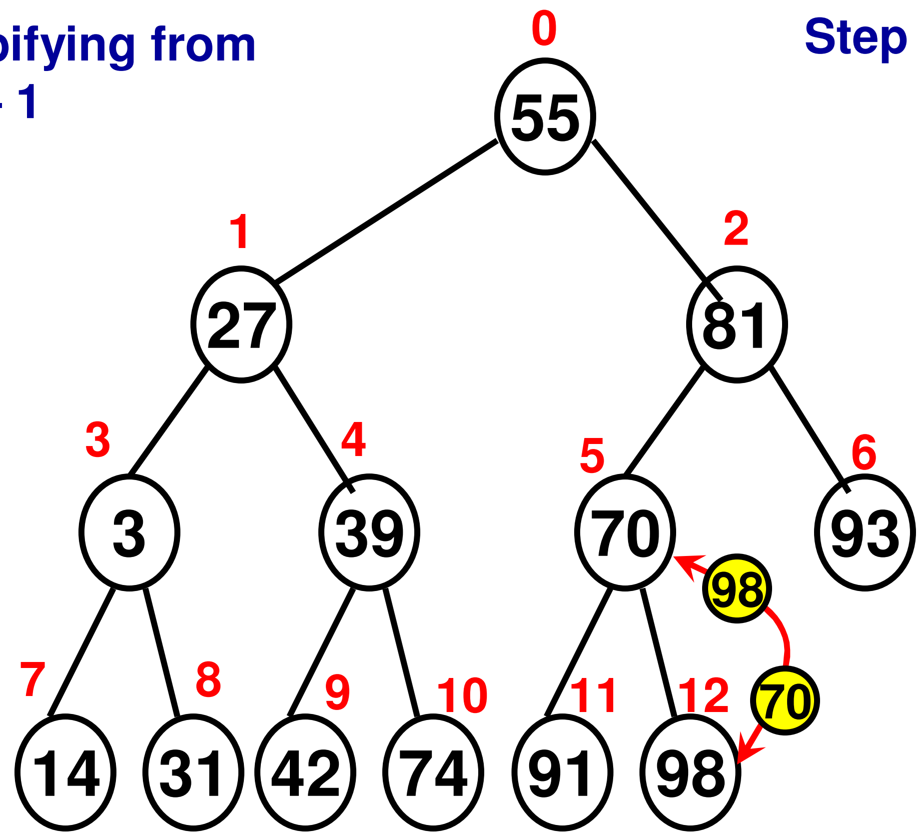
Step 1: Essentially Complete Binary Tree (Not a heap yet!)

Before (Reheapify at Index '5'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	39	70	93	14	31	42	74	91	98

Start reheapifying from Index $\lfloor 13/2 \rfloor - 1$

Step 2: Reheapify at index 5



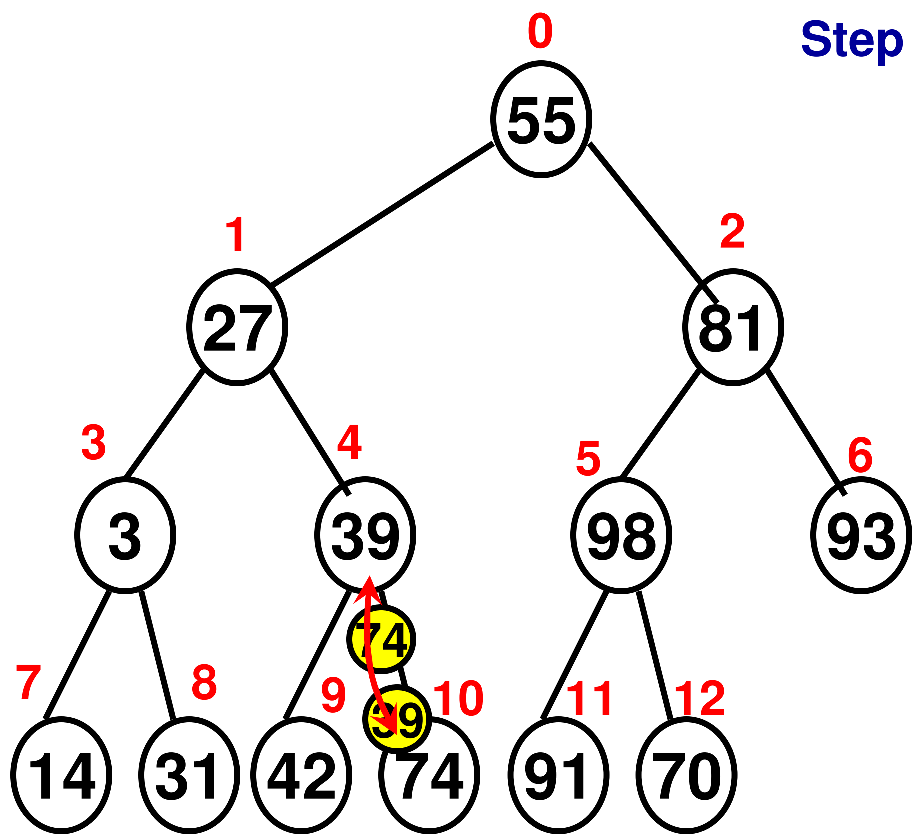
After (Reheapify at Index '5'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	39	98	93	14	31	42	74	91	70

Before (Reheapify at Index ‘4’):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	39	98	93	14	31	42	74	91	70

Step 2: Reheapify at index 4



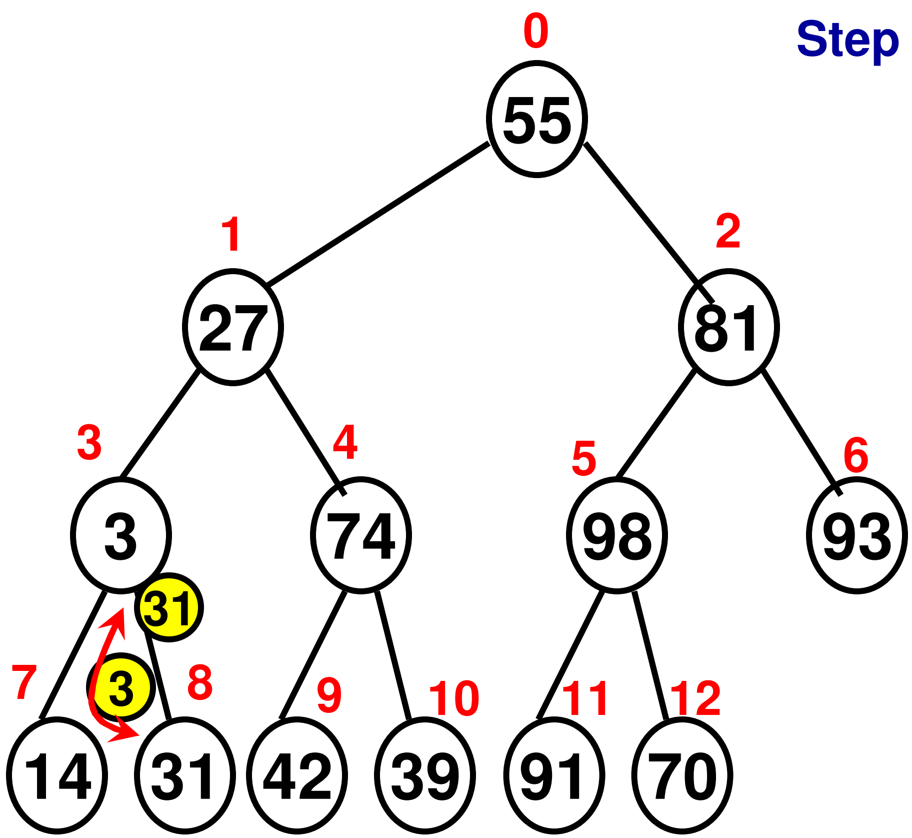
After (Reheapify at Index ‘4’):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	74	98	93	14	31	42	39	91	70

Before (Reheapify at Index '3'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	3	74	98	93	14	31	42	39	91	70

Step 2: Reheapify at index 3



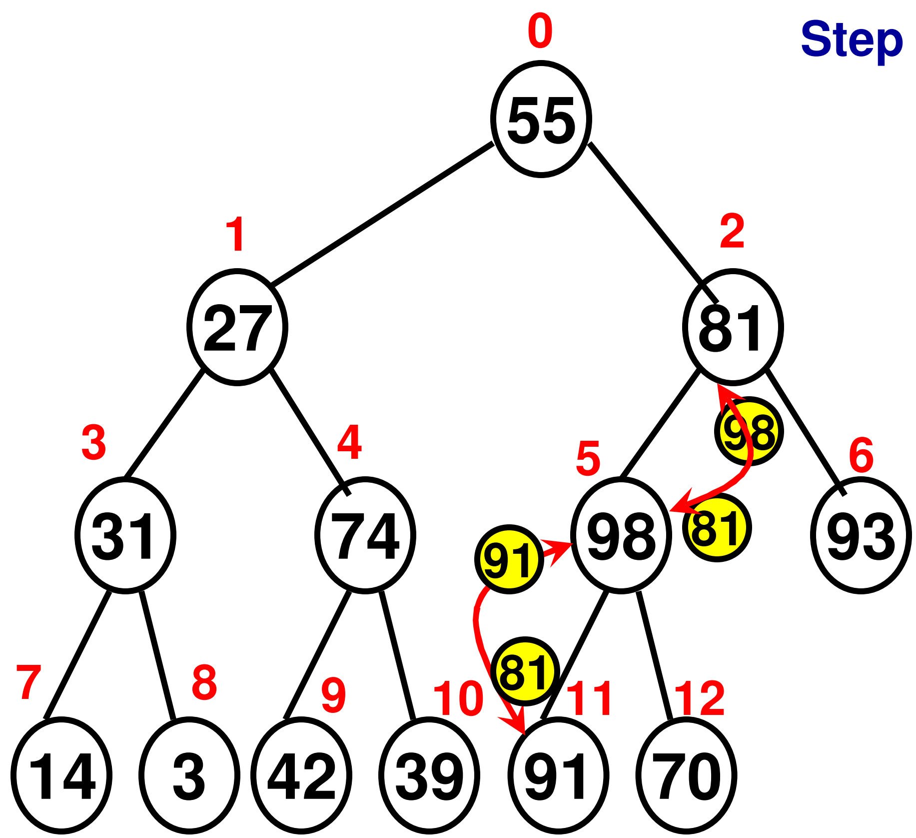
After (Reheapify at Index '3'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	31	74	98	93	14	3	42	39	91	70

Before (Reheapify at Index '2'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	81	31	74	98	93	14	3	42	39	91	70

Step 2: Reheapify at index 2



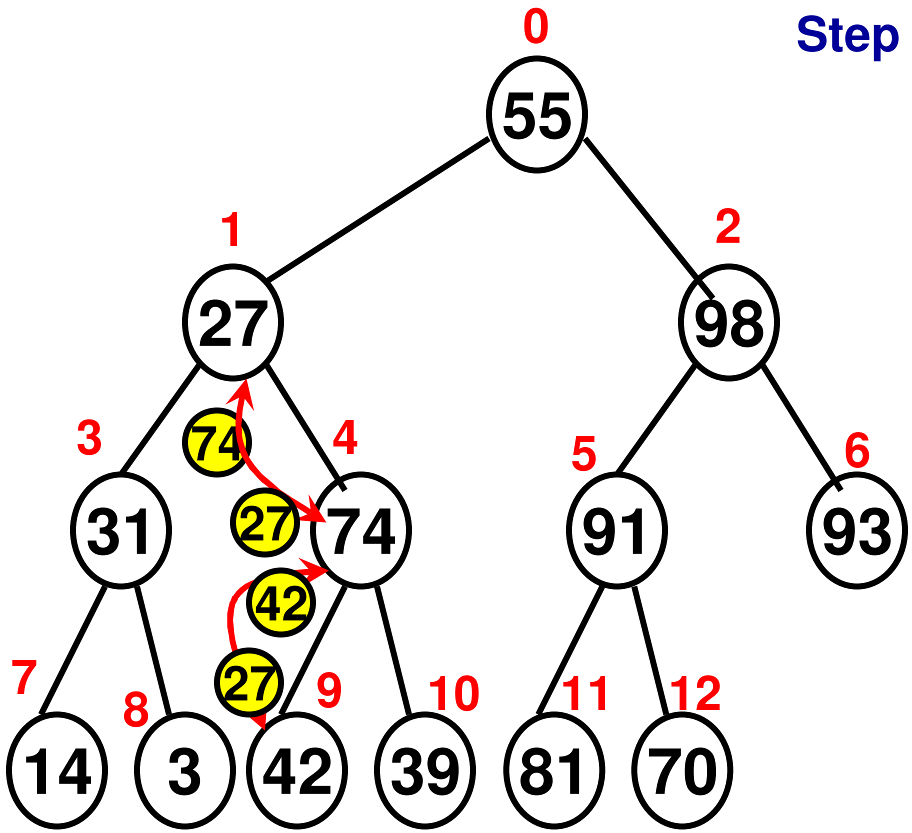
After (Reheapify at Index '2'):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	98	31	74	91	93	14	3	42	39	81	70

Before (Reheapify at Index ‘1’):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	27	98	31	74	91	93	14	3	42	39	81	70

Step 2: Reheapify at index 1



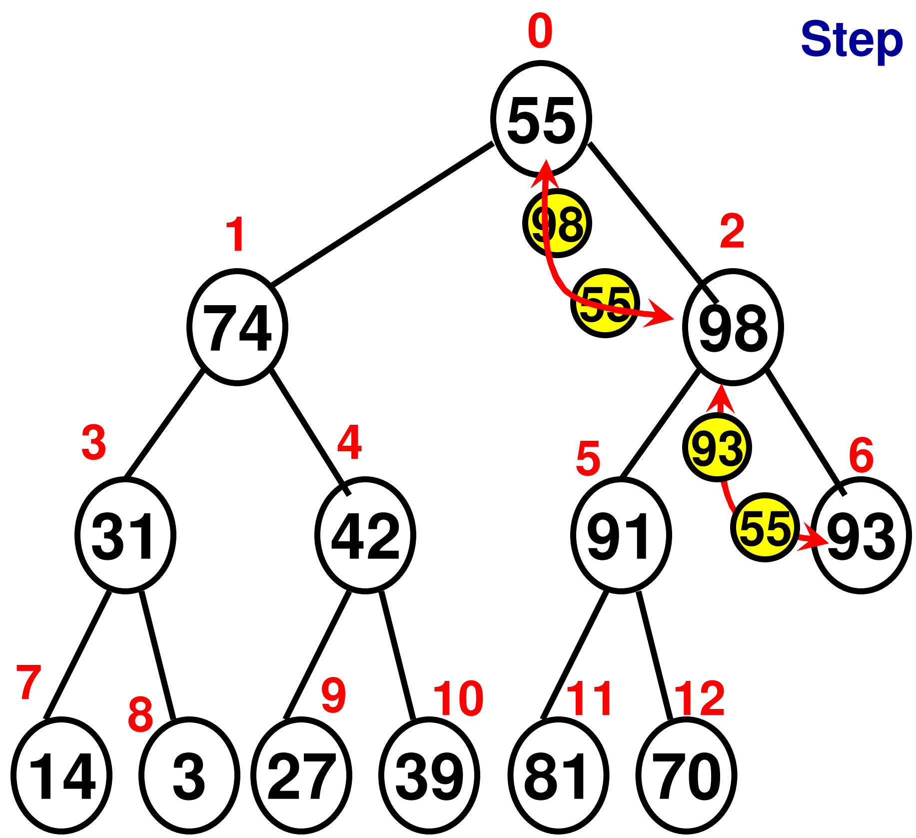
After (Reheapify at Index ‘1’):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	74	98	31	42	91	93	14	3	27	39	81	70

Before (Reheapify at Index ‘0’):

0	1	2	3	4	5	6	7	8	9	10	11	12
55	74	98	31	42	91	93	14	3	27	39	81	70

Step 2: Reheapify at index 0



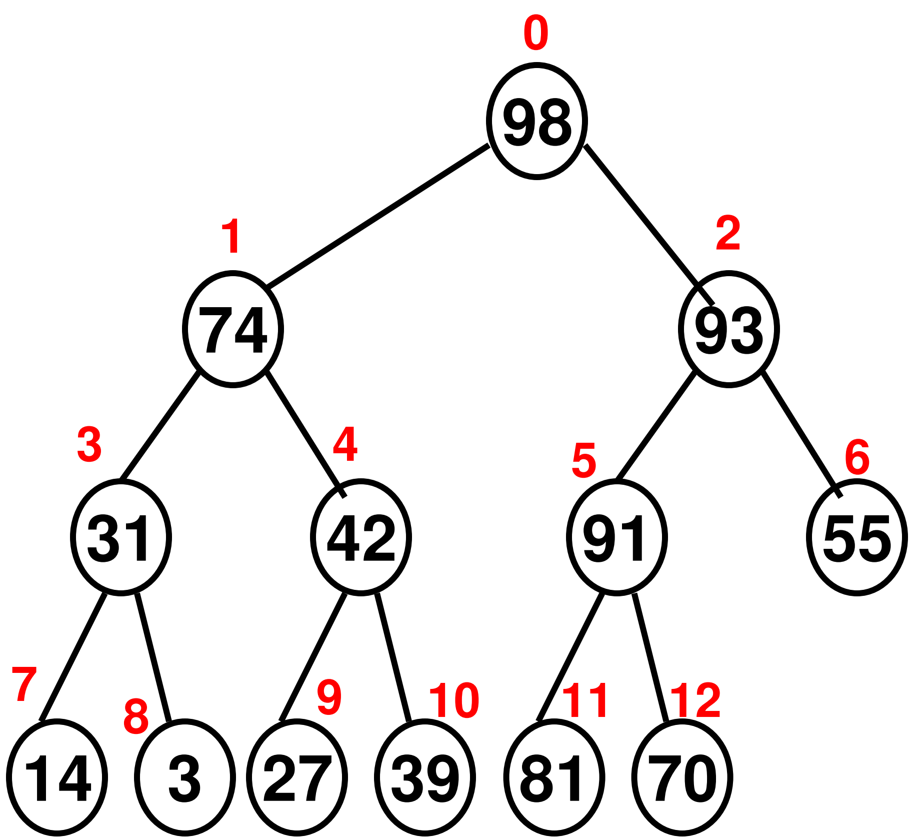
After (Reheapify at Index ‘0’):

0	1	2	3	4	5	6	7	8	9	10	11	12
98	74	93	31	42	91	55	14	3	27	39	81	70

Heap Construction Example 2

0	1	2	3	4	5	6	7	8	9	10	11	12
98	74	93	31	42	91	55	14	3	27	39	81	70

Final Array Representing Max Heap



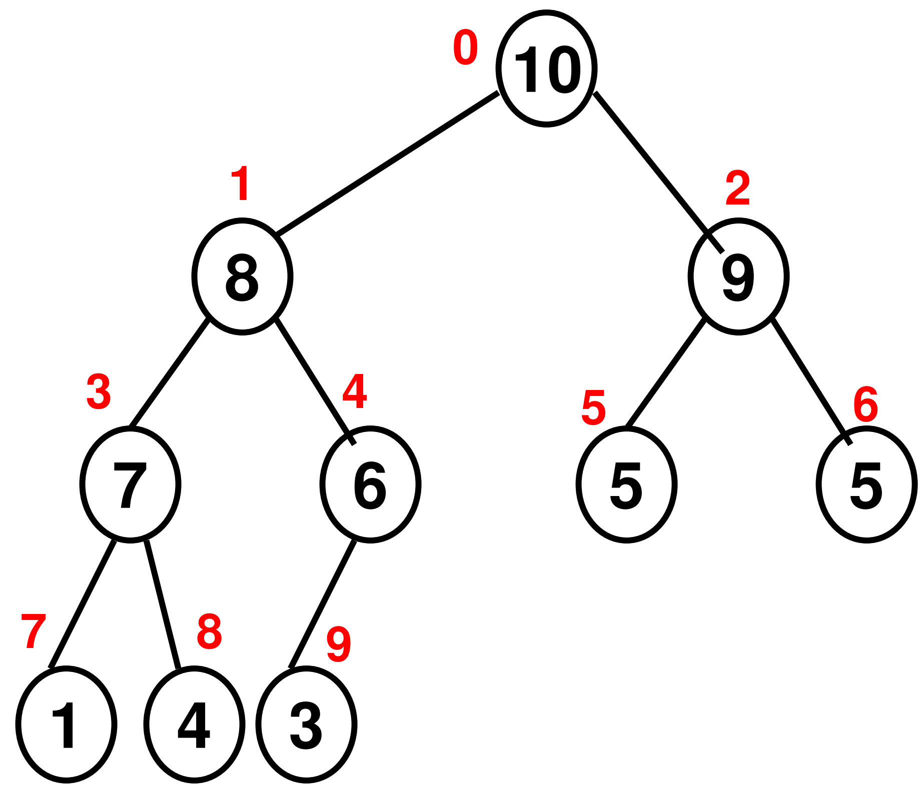
Heap Sort

- Given an array of size ‘n’, first construct a max-heap version of the array.
- Run ‘n-1’ iterations (iteration index 0 to n-1)
  – Swap element at index “0” with element at index “n-1-iteration index”
  – Element at index “0” has now moved to its final location “n-1-iteration index” in the sorted array
  – Reheapify the array as a result of this swap with the array index values ranging from “0” to “n-1-iteration index – 1”.
- Each iteration would require “logn” swappings at the worst case, across the entire height of the binary tree.
- For a total of ‘n-1’ iterations, the time complexity of heap sort is O(nlogn).

Heap Sort: Example 1

Original Array	5	6	5	4	3	10	9	1	7	8
	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	10	8	9	7	6	5	5	1	4	3

Max-Heap

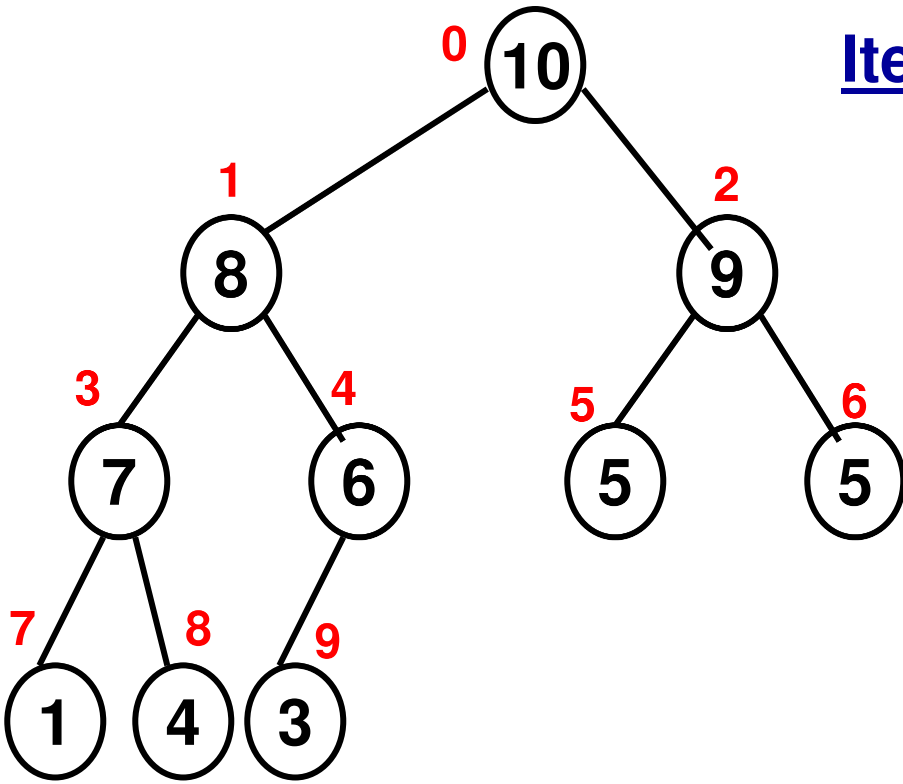


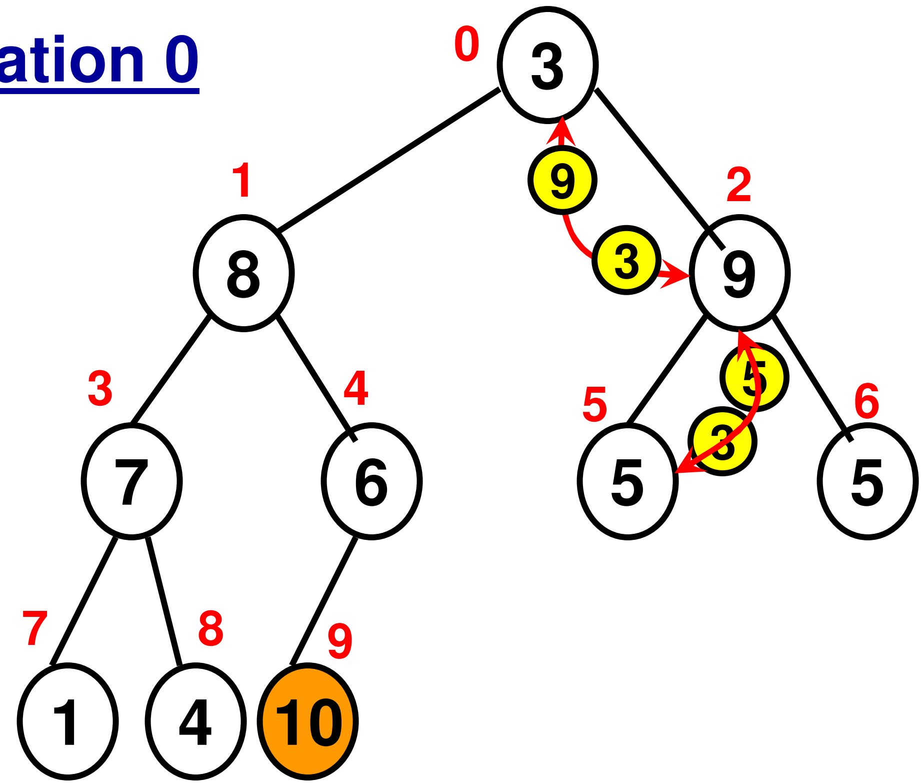
	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	10	8	9	7	6	5	5	1	4	3

	0	1	2	3	4	5	6	7	8	9
After Swap	3	8	9	7	6	5	5	1	4	10

	0	1	2	3	4	5	6	7	8	9
After Reheapify	9	8	5	7	6	3	5	1	4	10

Iteration 0





Max-Heap Version

0	1	2	3	4	5	6	7	8	9
9	8	5	7	6	3	5	1	4	10

4 9

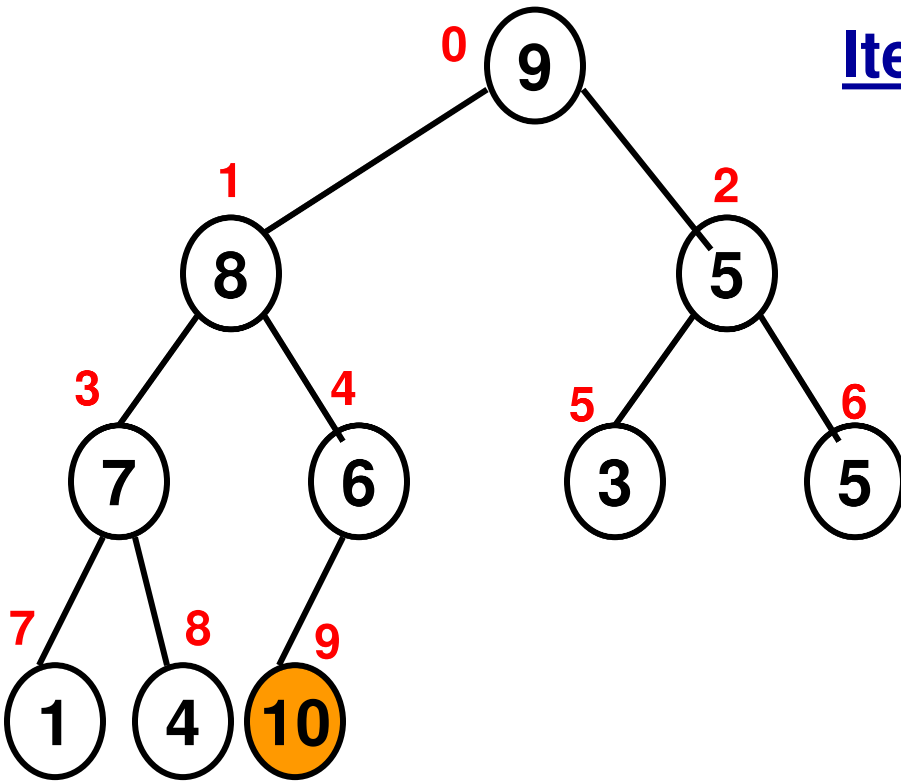
After Swap

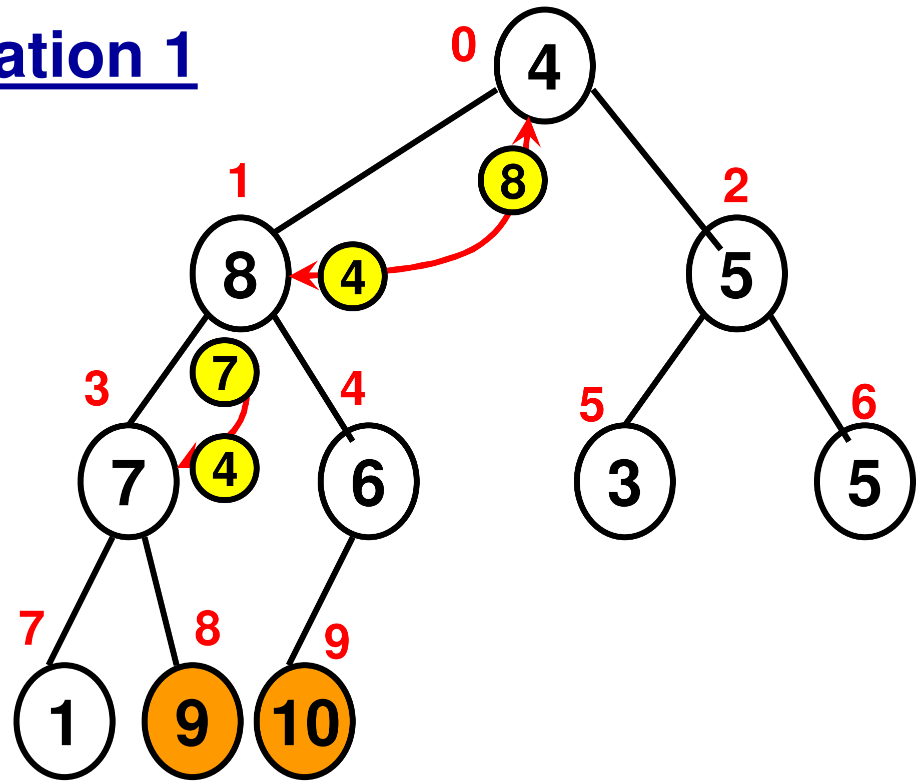
0	1	2	3	4	5	6	7	8	9
4	8	5	7	6	3	5	1	9	10

After Reheapify

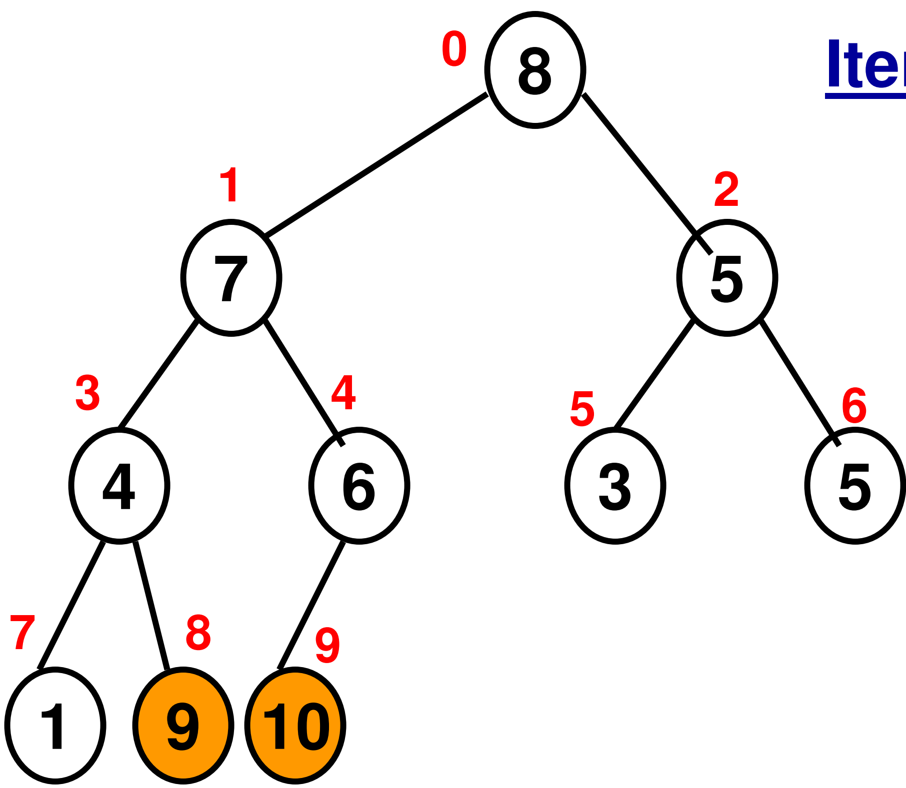
0	1	2	3	4	5	6	7	8	9
8	7	5	4	6	3	5	1	9	10

Iteration 1

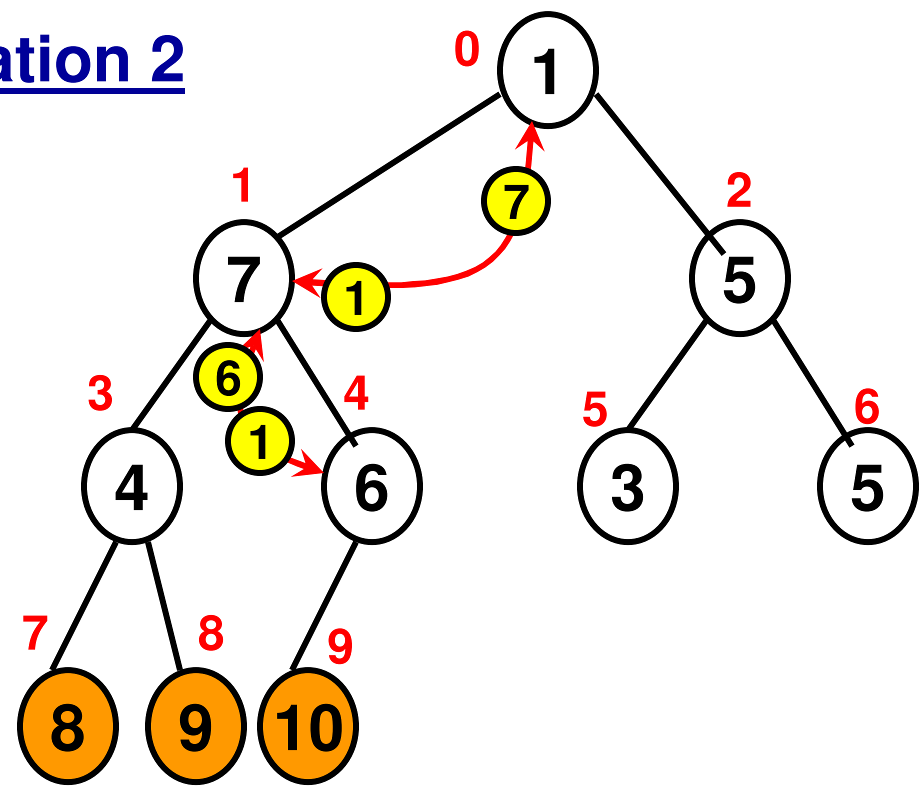




	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	8	7	5	4	6	3	5	1	9	10
After Swap	1	7	5	4	6	3	5	8	9	10
After Reheapify	7	6	5	4	1	3	5	8	9	10

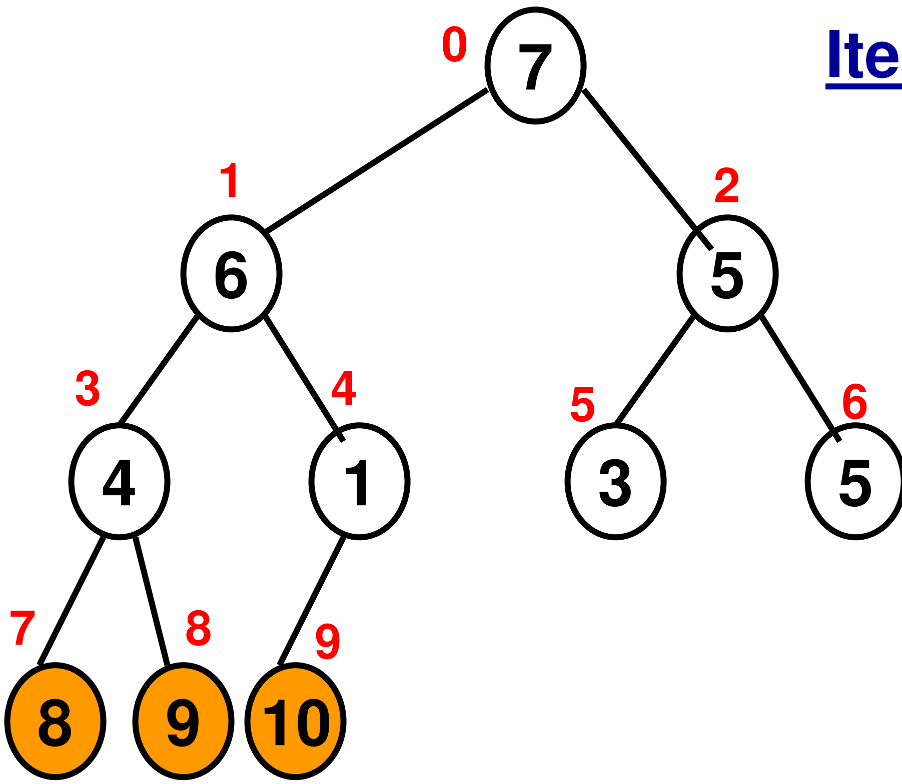


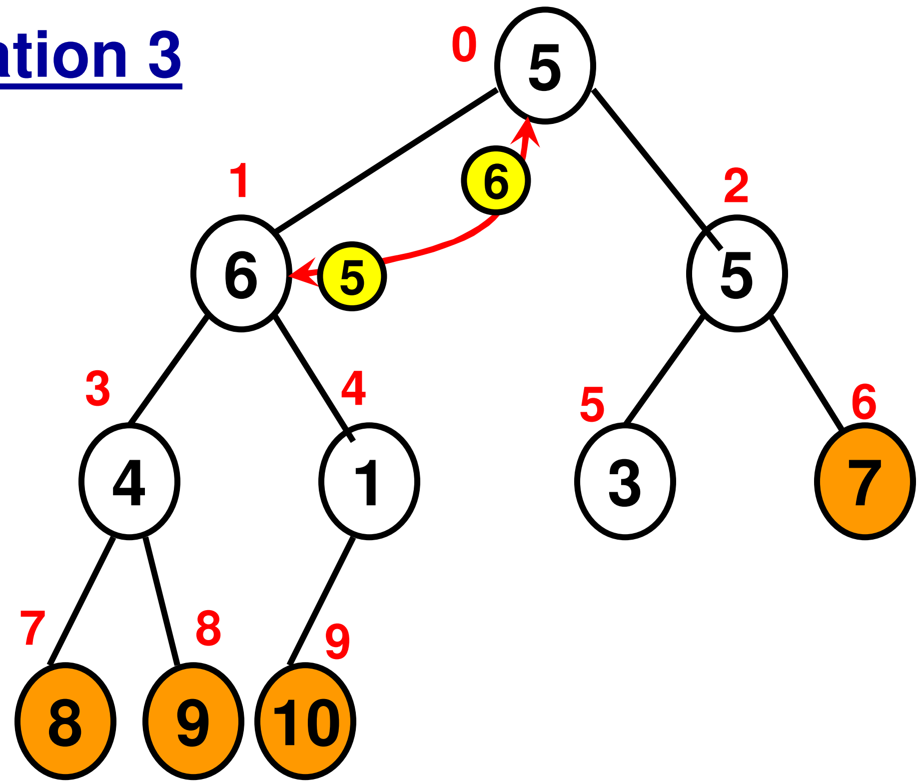
Iteration 2



	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	7	6	5	4	1	3	5	8	9	10
After Swap	5	6	5	4	1	3	7	8	9	10
After Reheapify	6	5	5	4	1	3	7	8	9	10

Iteration 3





Max-Heap Version

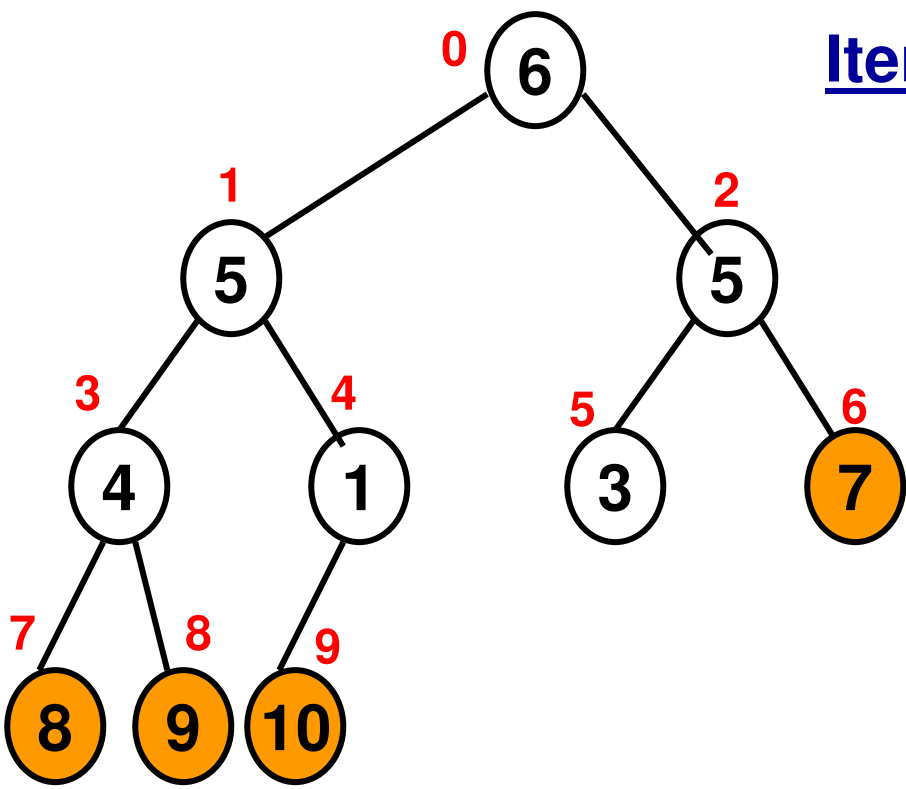
0	1	2	3	4	5	6	7	8	9
6	5	5	4	1	3	7	8	9	10

After Swap

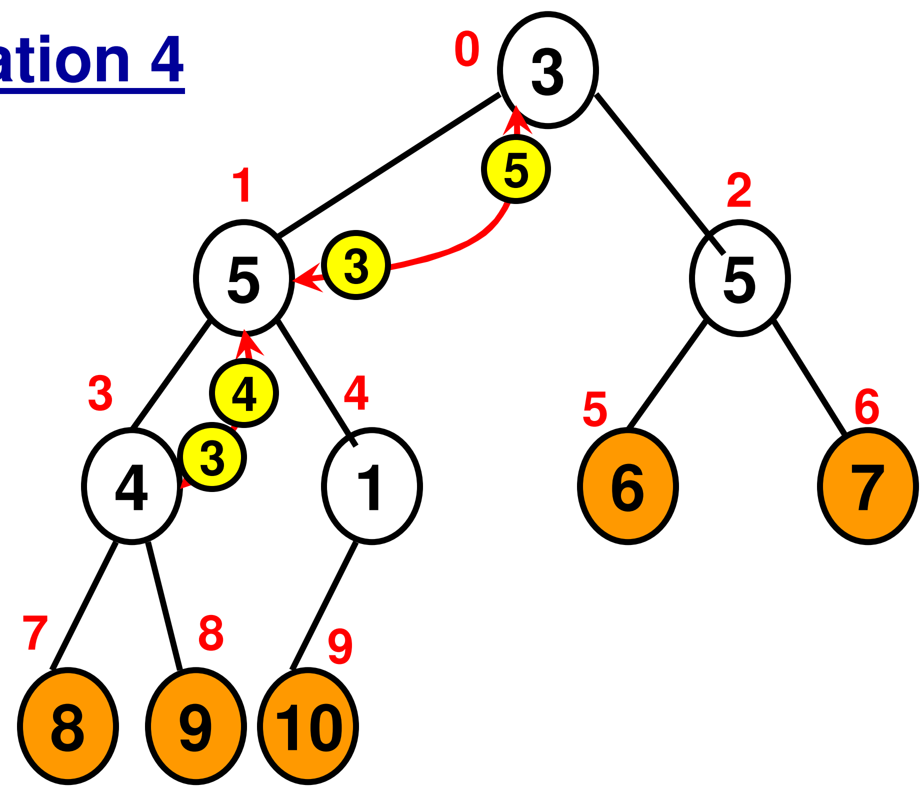
0	1	2	3	4	5	6	7	8	9
3	5	5	4	1	6	7	8	9	10

After Reheapify

0	1	2	3	4	5	6	7	8	9
5	4	5	3	1	6	7	8	9	10



Iteration 4



Iteration 5

Max-Heap Version

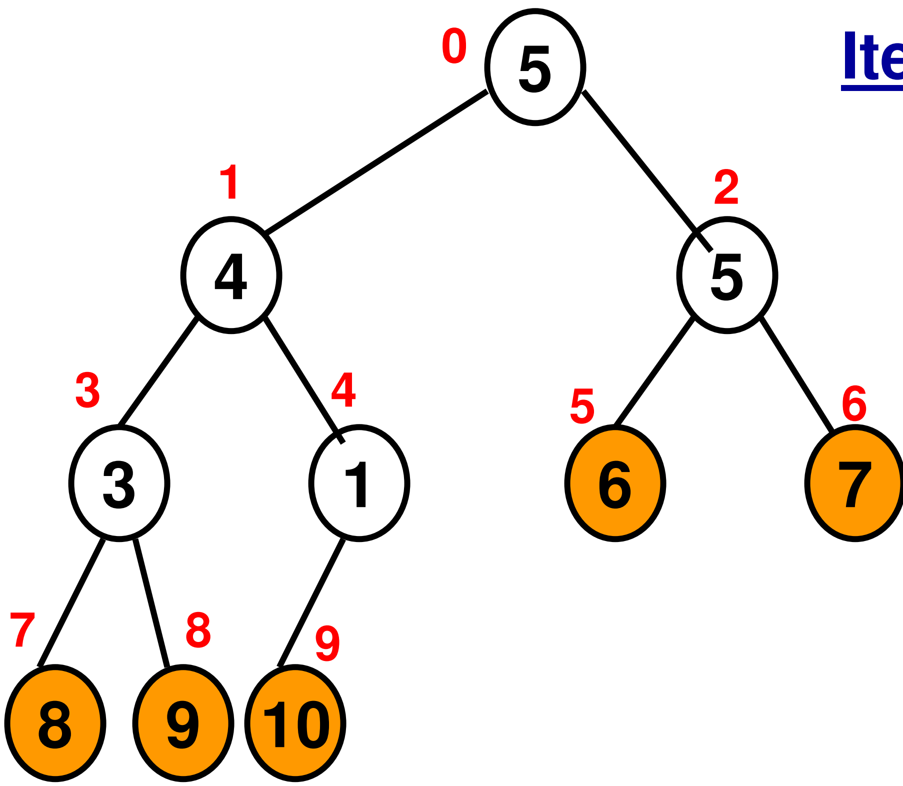
0	1	2	3	4	5	6	7	8	9
5	4	5	3	1	6	7	8	9	10

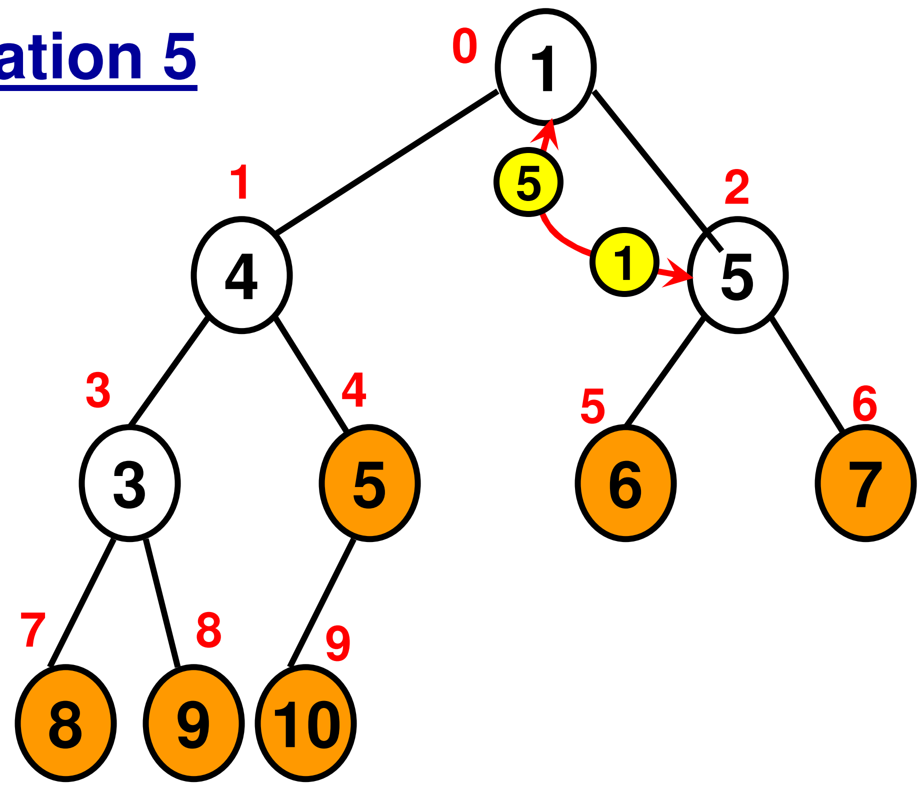
After Swap

0	1	2	3	4	5	6	7	8	9
1	4	5	3	5	6	7	8	9	10

After Reheapify

0	1	2	3	4	5	6	7	8	9
5	4	1	3	5	6	7	8	9	10

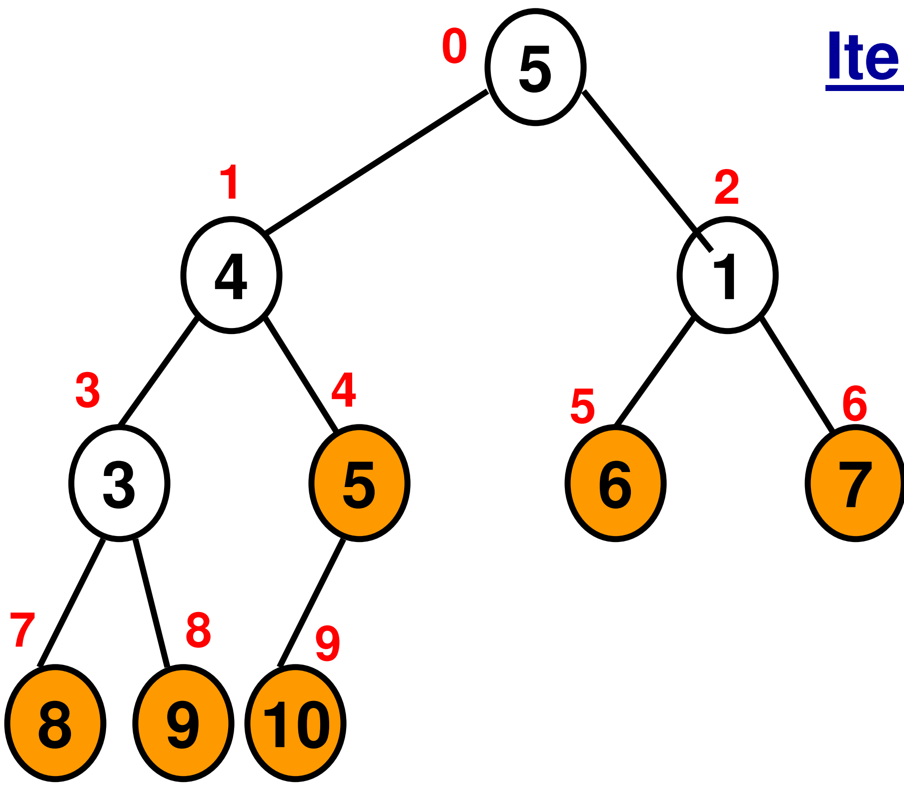




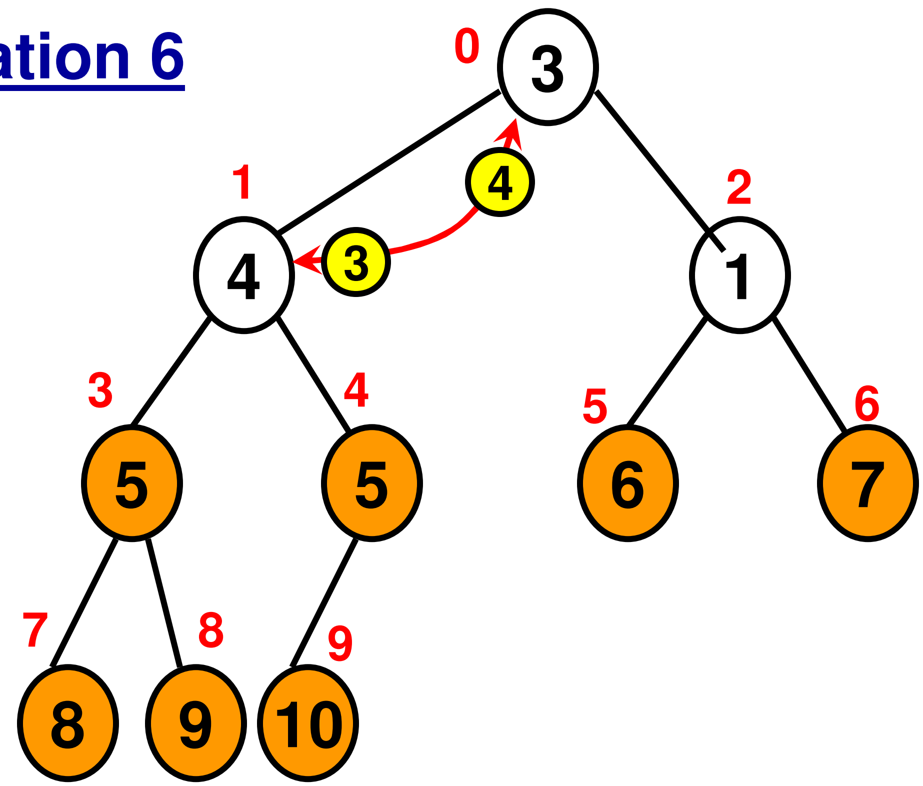
	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	5	4	1	3	5	6	7	8	9	10

	0	1	2	3	4	5	6	7	8	9
After Swap	3	4	1	5	5	6	7	8	9	10

	0	1	2	3	4	5	6	7	8	9
After Reheapify	4	3	1	5	5	6	7	8	9	10



Iteration 6

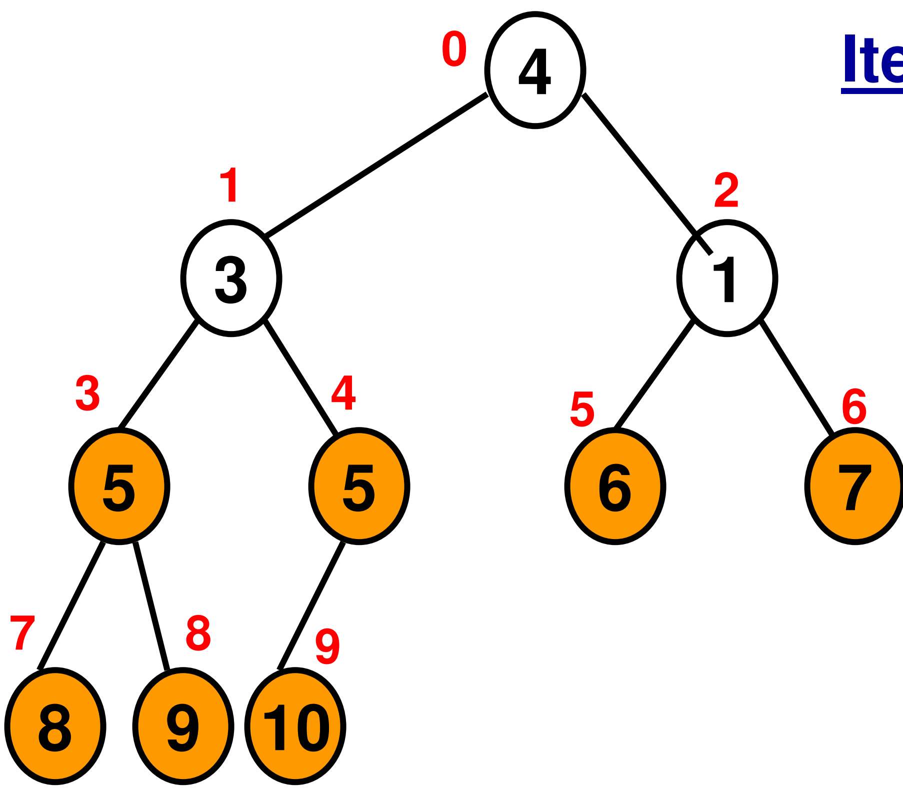


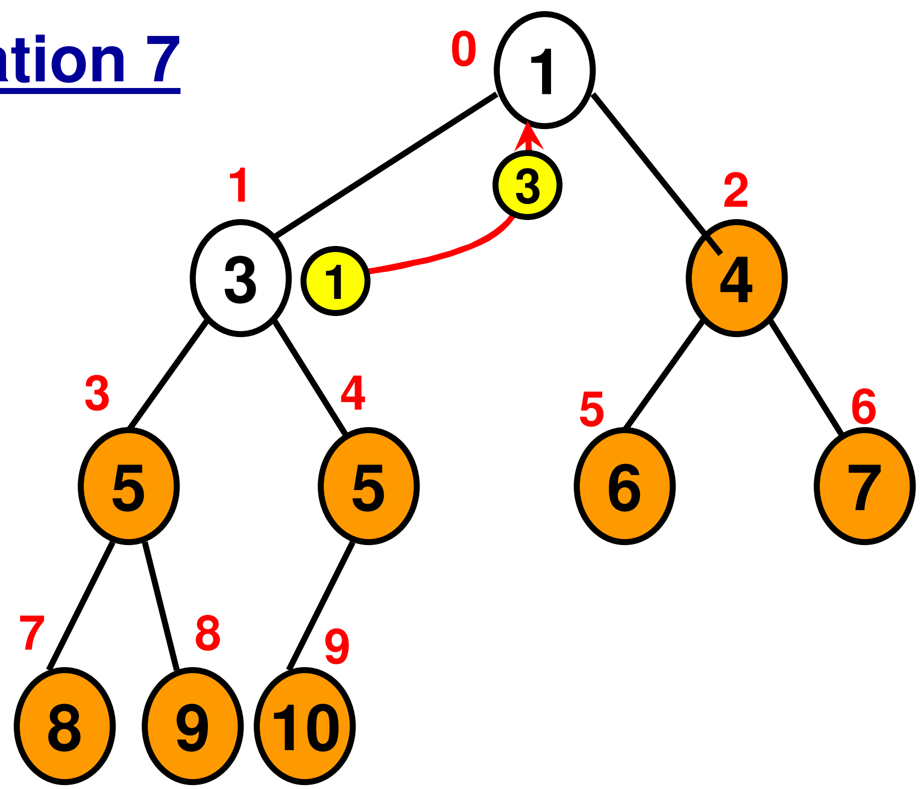
	0	1	2	3	4	5	6	7	8	9
Max-Heap Version	4	3	1	5	5	6	7	8	9	10

	0	1	2	3	4	5	6	7	8	9
After Swap	1	3	4	5	5	6	7	8	9	10

	0	1	2	3	4	5	6	7	8	9
After Reheapify	3	1	4	5	5	6	7	8	9	10

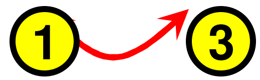
Iteration 7





Max-Heap Version

0	1	2	3	4	5	6	7	8	9
3	1	4	5	5	6	7	8	9	10

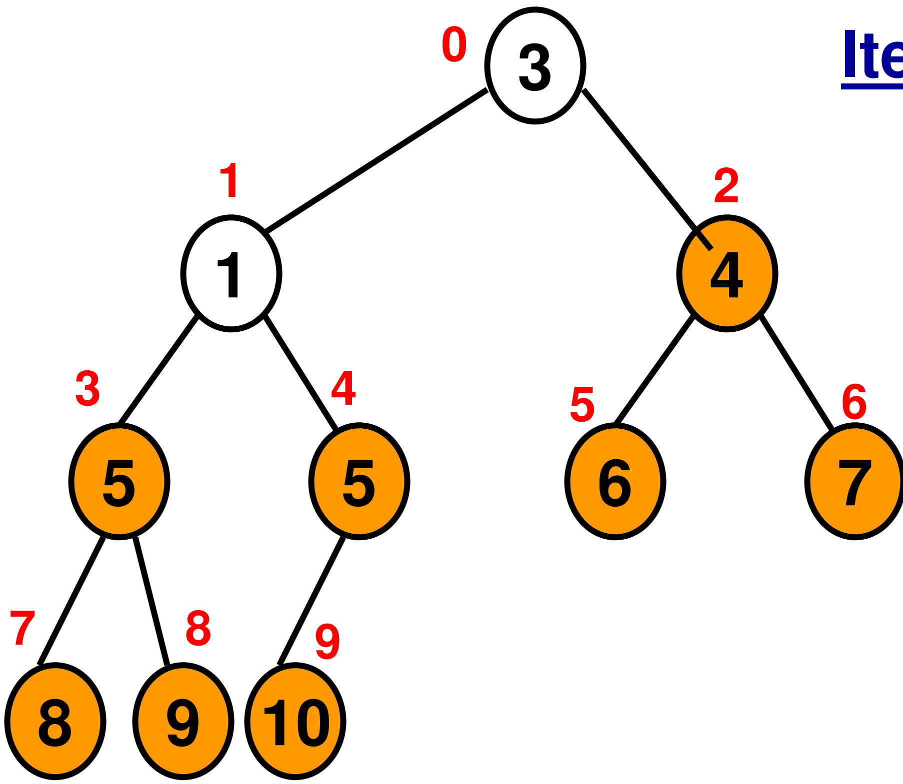


After Swap

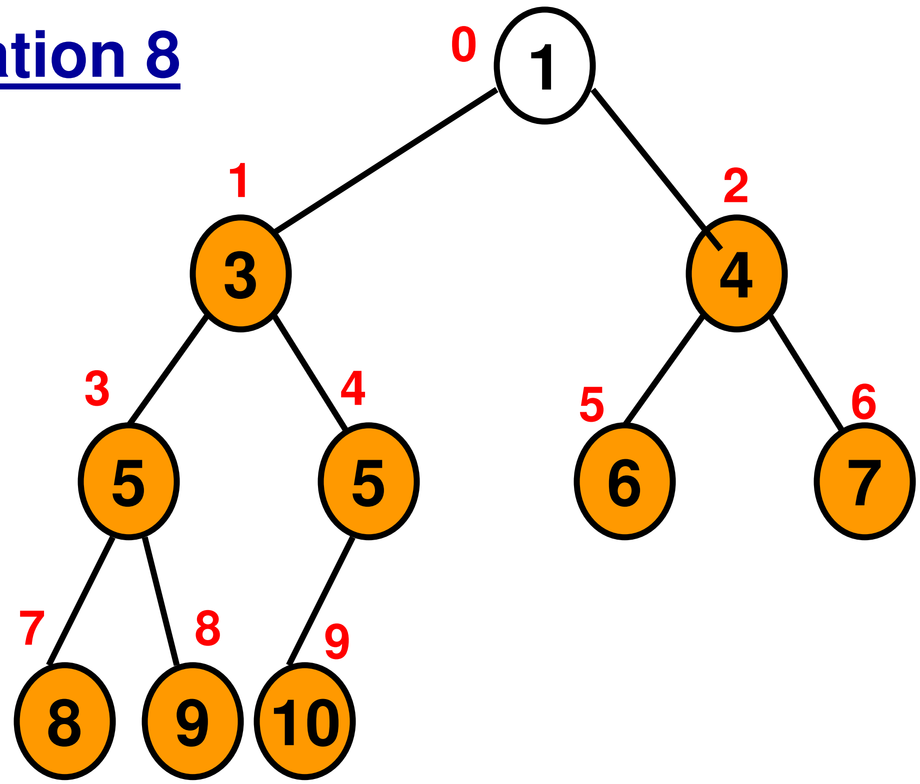
0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

After Reheapify

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

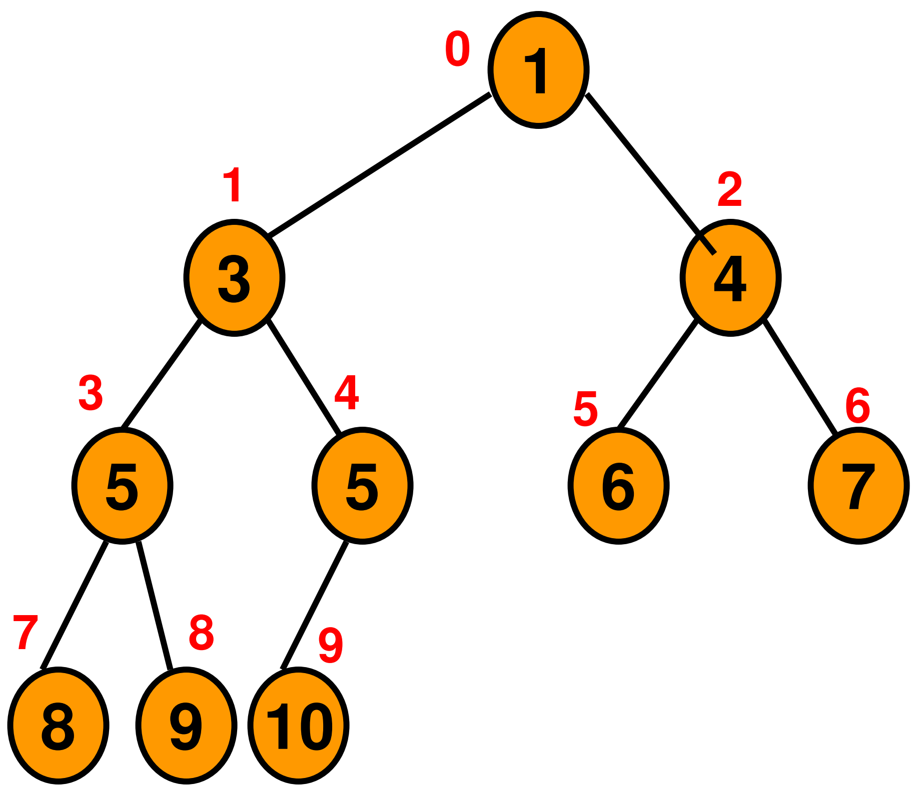


Iteration 8



Heap Sort: Example
Final Sorted Array

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10



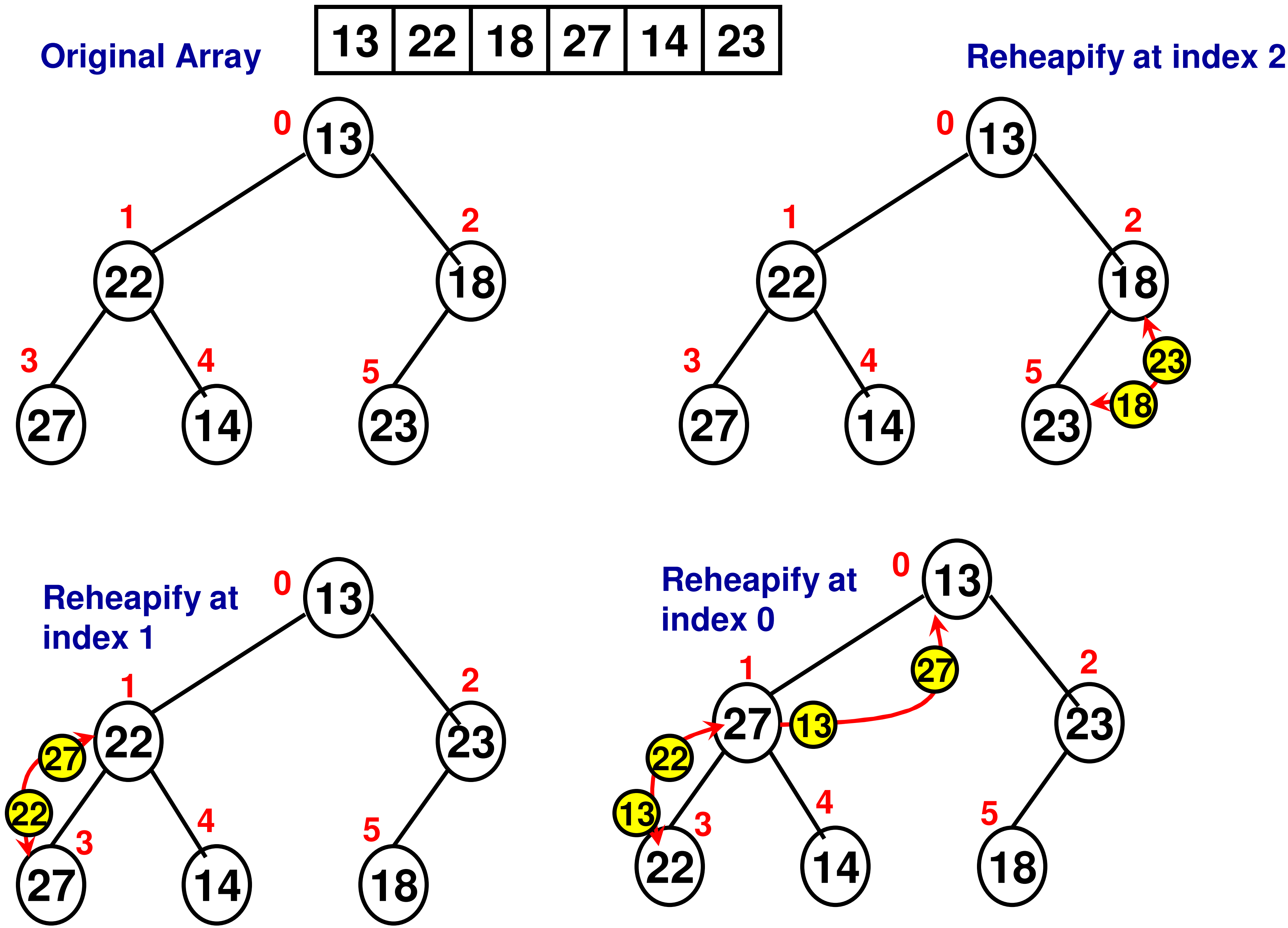
Heap Sort (Code 8.1: C++)

```
for (int iterationIndex = 0; iterationIndex < arraySize; iterationIndex++){

        int temp = array[0];
        array[0] = array[arraySize-1-iterationIndex];
        array[arraySize-1-iterationIndex] = temp;

        rearrangeHeapArray(array, arraySize-1-iterationIndex, 0);

        cout << "Iteration " << iterationIndex << " : ";
        for (int index = 0; index < arraySize; index++)
                cout << array[index] << " ";

        cout << endl;

}
```

Swap the element at the top of the heap with the element at the last index (arraySize-1-iterationIndex) in the active portion of the array

The active portion of the array ranges from Index '0' to 'arraySize-1-iterationIndex'

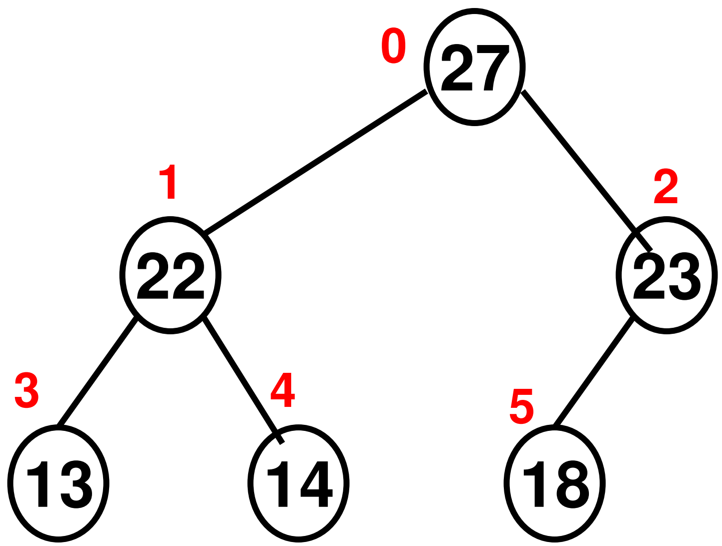
After the swap, the size of the active portion of the array is 'arraySize-1-iterationIndex'


Original Array
13 22 18 27 14 23
Reheapify at index 2
Reheapify at index 1
Reheapify at index 0

Heap Sort: Example 2

Original Array	13	22	18	27	14	23
	0	1	2	3	4	5
Max-Heap Version	27	22	23	13	14	18

Max-Heap



Max-Heap Version

0	1	2	3	4	5
27	22	23	13	14	18

18 27

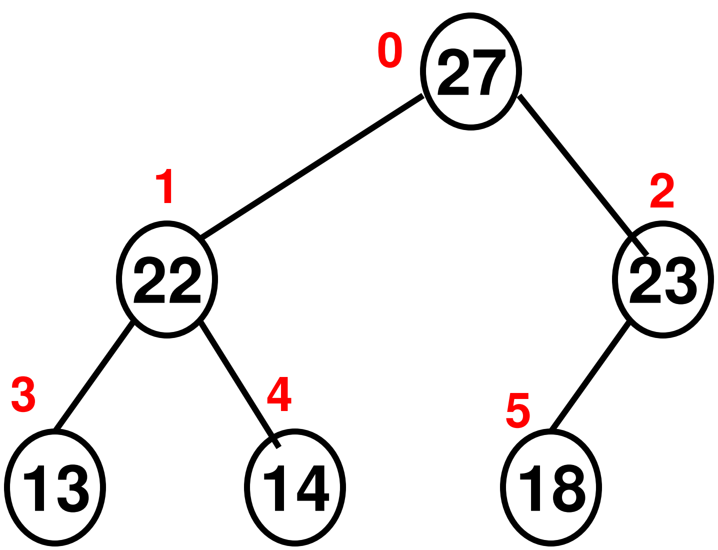
After Swap

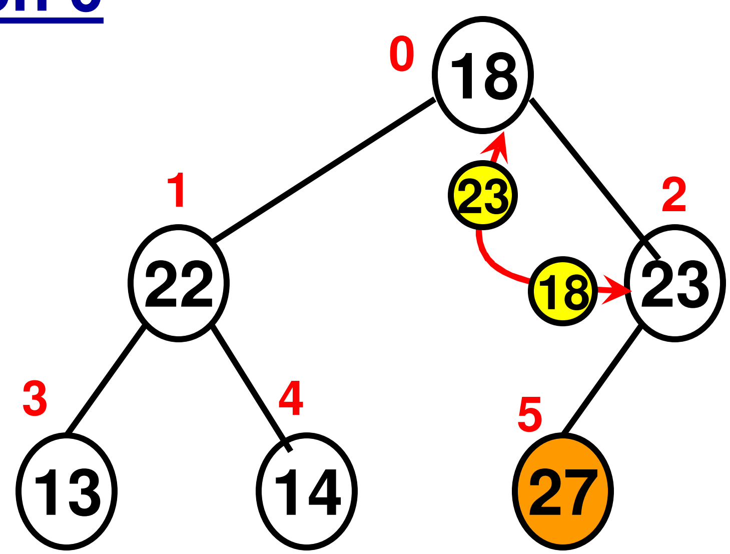
0	1	2	3	4	5
18	22	23	13	14	27

After Reheapify

0	1	2	3	4	5
23	22	18	13	14	27

Iteration 0





Max-Heap Version

0	1	2	3	4	5
23	22	18	13	14	27

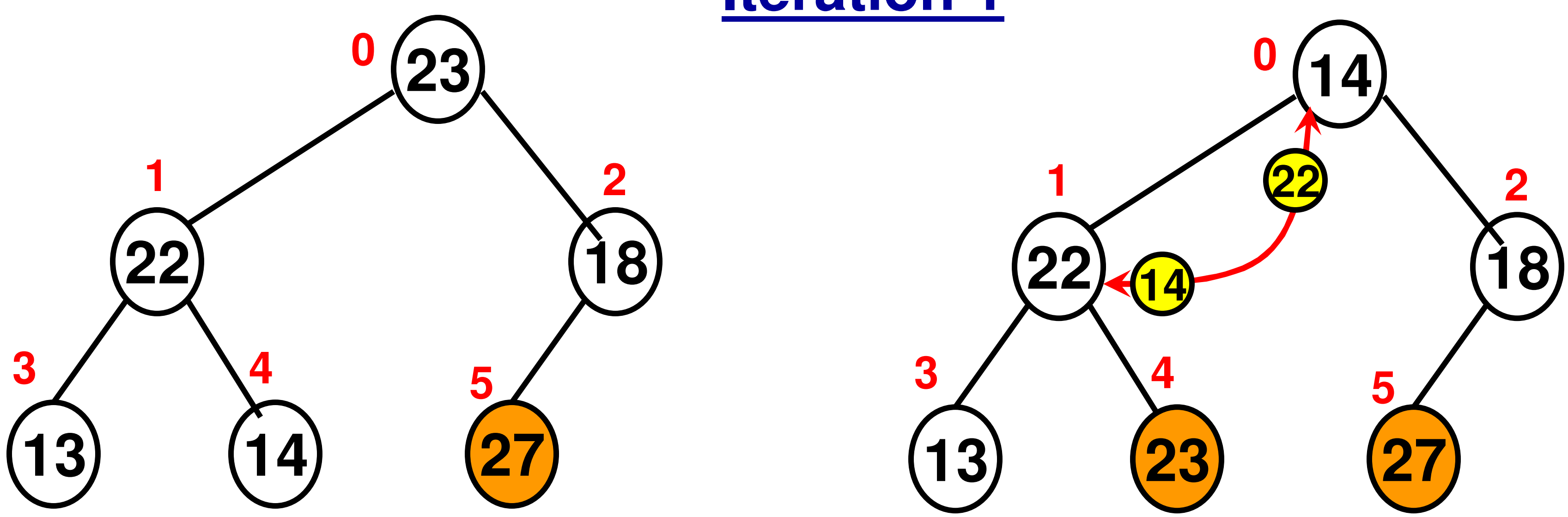
After Swap

0	1	2	3	4	5
14	22	18	13	23	27

After Reheapify

0	1	2	3	4	5
22	14	18	13	23	27

Iteration 1



Max-Heap Version

0	1	2	3	4	5
22	14	18	13	23	27

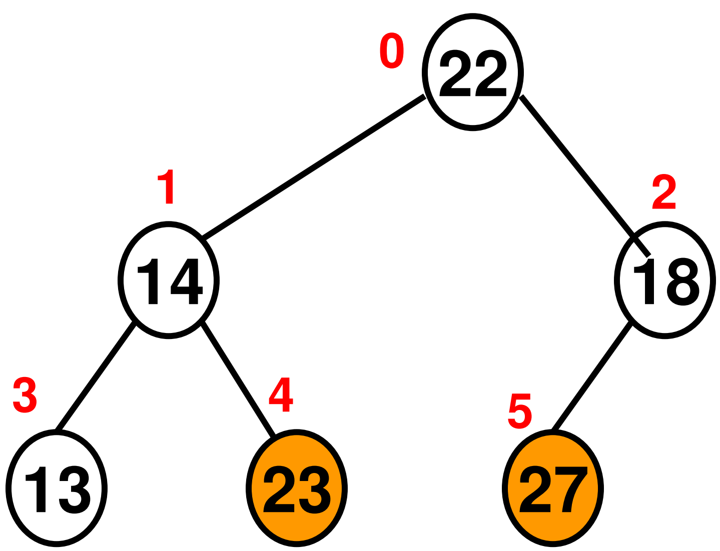
After Swap

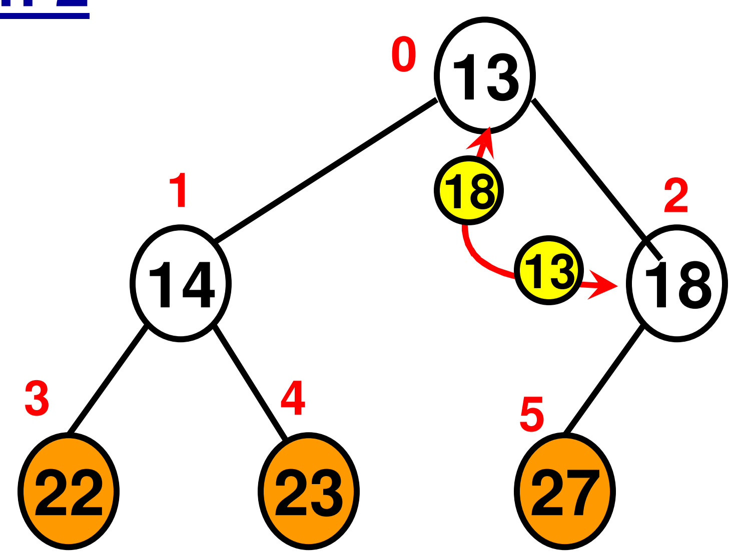
0	1	2	3	4	5
13	14	18	22	23	27

After Reheapify

0	1	2	3	4	5
18	14	13	22	23	27

Iteration 2





Max-Heap Version

0	1	2	3	4	5
18	14	13	22	23	27

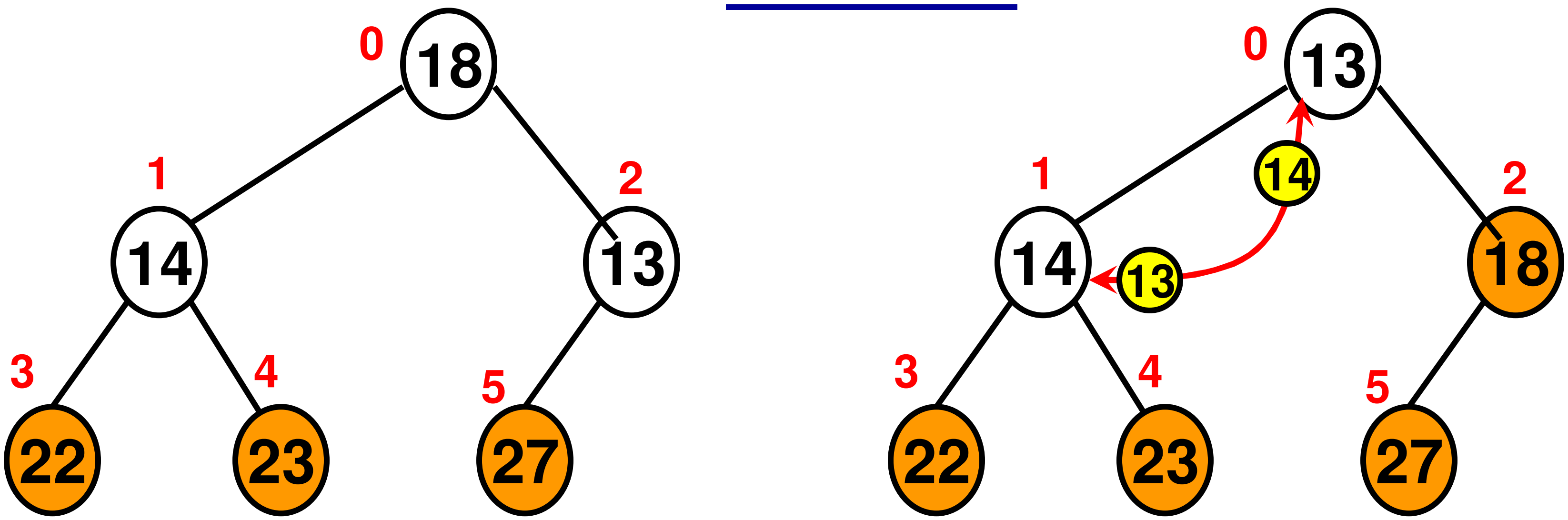
After Swap

0	1	2	3	4	5
13	14	18	22	23	27

After Reheapify

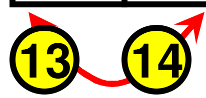
0	1	2	3	4	5
14	13	18	22	23	27

Iteration 3



Max-Heap Version

0	1	2	3	4	5
14	13	18	22	23	27



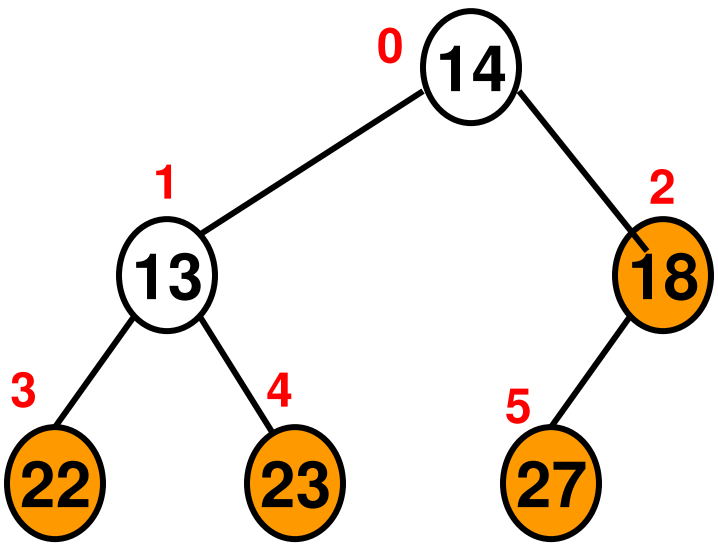
After Swap

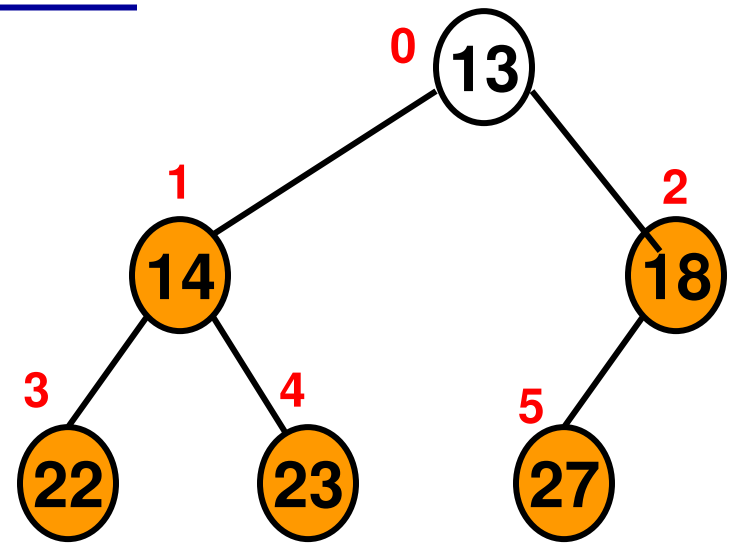
0	1	2	3	4	5
13	14	18	22	23	27

After Reheapify

0	1	2	3	4	5
13	14	18	22	23	27

Iteration 4

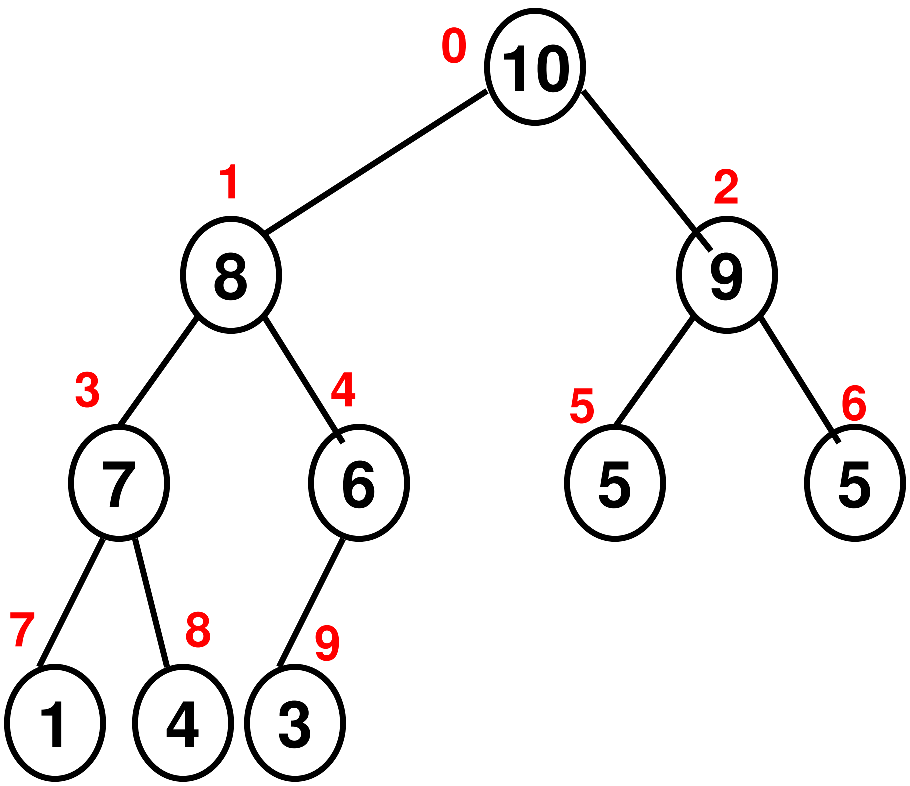


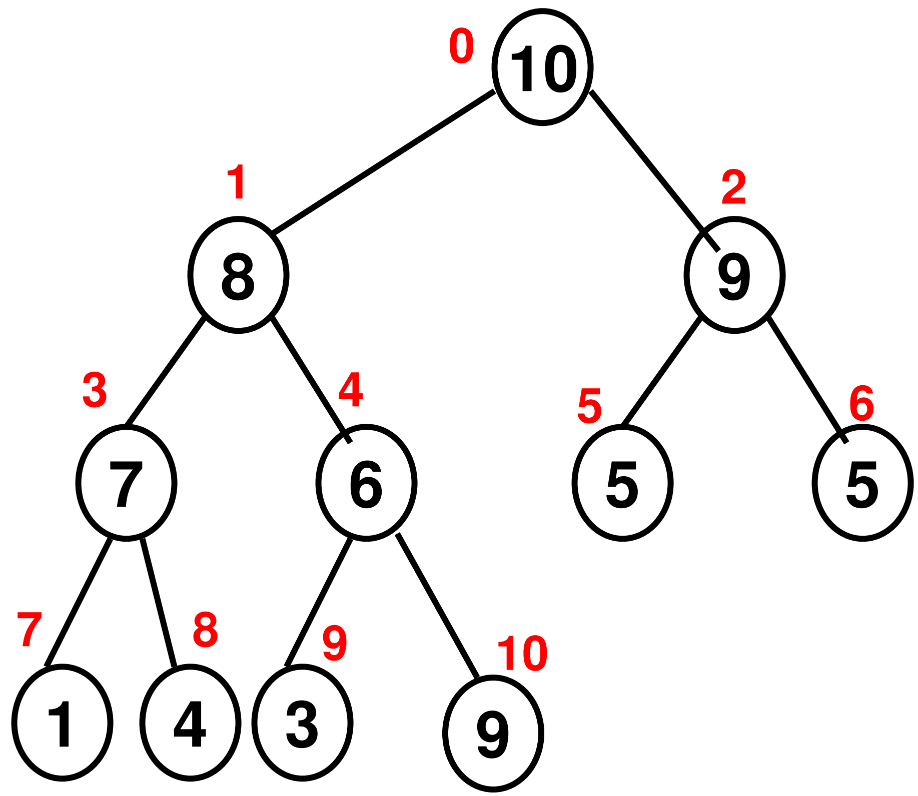


Inserting Data to a (Max) Heap

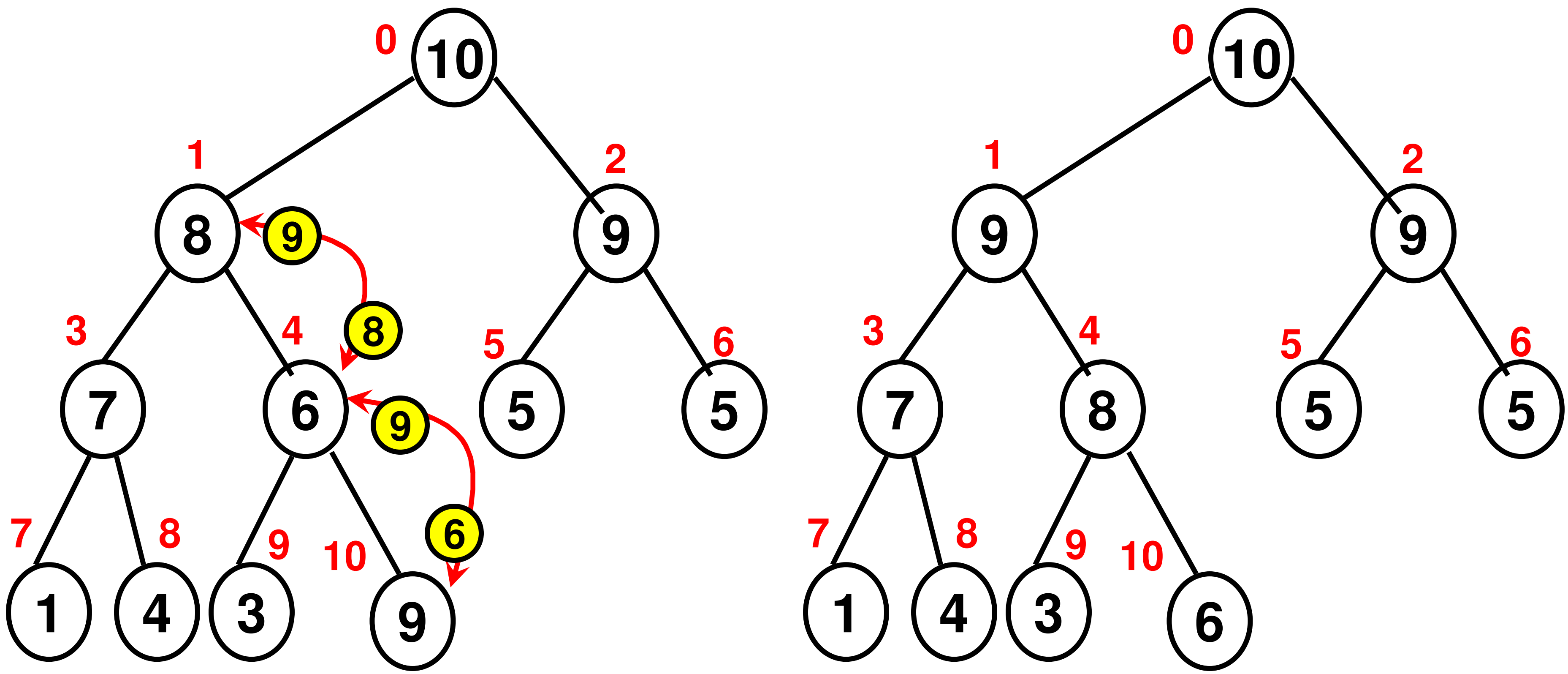
- Insert the data at the bottommost level at the leftmost position. Then reheapify starting from the parent node of the inserted node and recursively all the way to the root node or the internal node at which the heap property is satisfied. This will take O(logn) time.

Assume we want to insert data '9'
Initially, insert at index 10 and reheapify starting from index (10-1)/2 = 4.





Inserting Data to a (Max) Heap

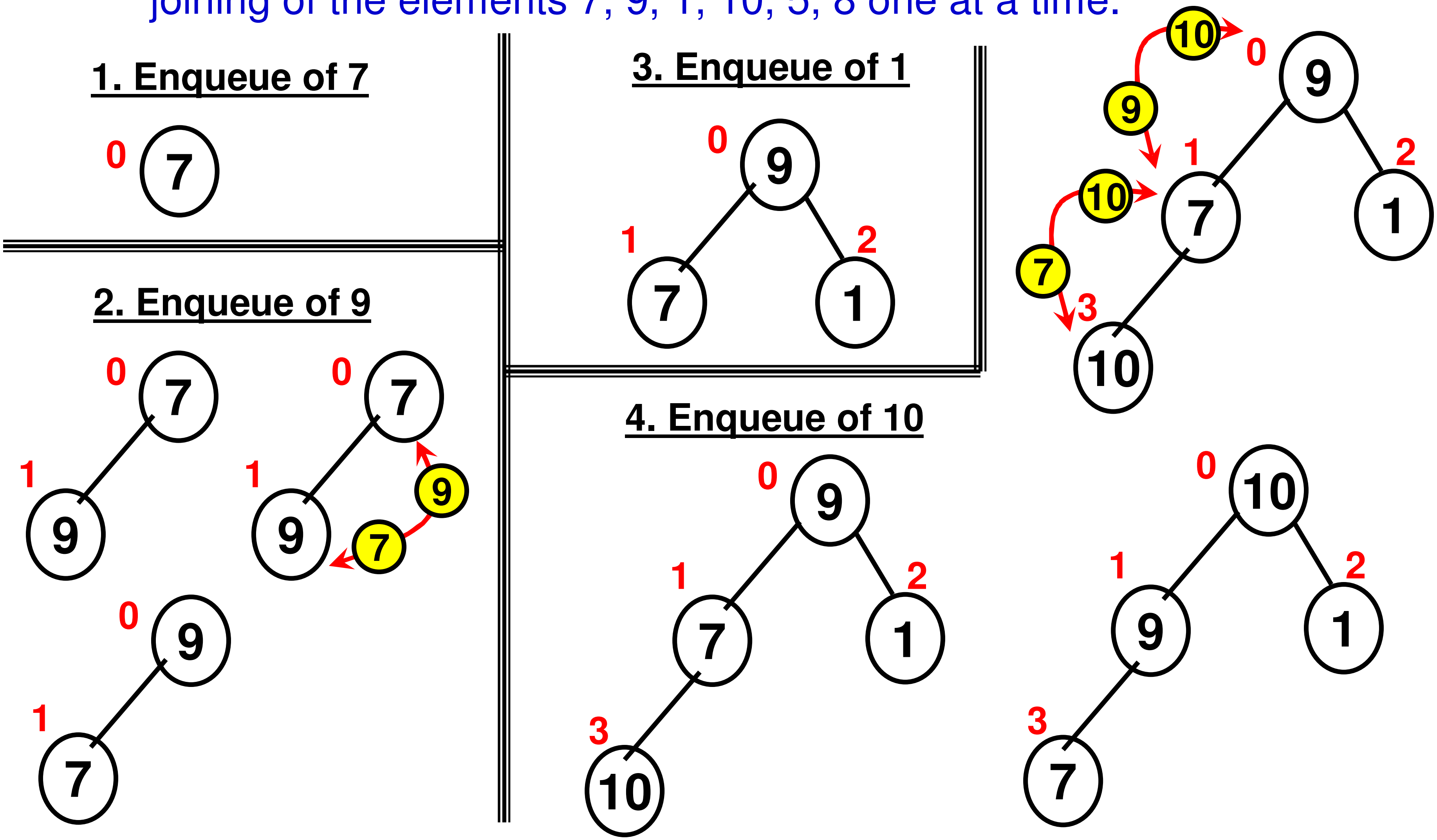


Heap – Priority Queue

- A heap can be used to implement a priority queue.
- Each element in the queue has a priority (typically, the numerical value of the element is its priority).
- The elements in the queue are arranged as a max or min heap (depending on how we define priority: the element with the largest value has the highest priority – max heap; the element with the lowest value has the highest priority – min heap).
- A dequeue operation on the priority queue will remove the root node of the heap and it will take O(logn) time to reheapify the heap.
- An enqueue operation on the priority queue will insert the node initially at the last index and then reheapify all the way to the root node if needed: O(logn) time.
- <u>Tradeoff:</u> We saw earlier that a regular FIFO queue could be implemented as a doubly linked list with O(1) time for the enqueue and dequeue operations.

Priority Queue Construction: Example

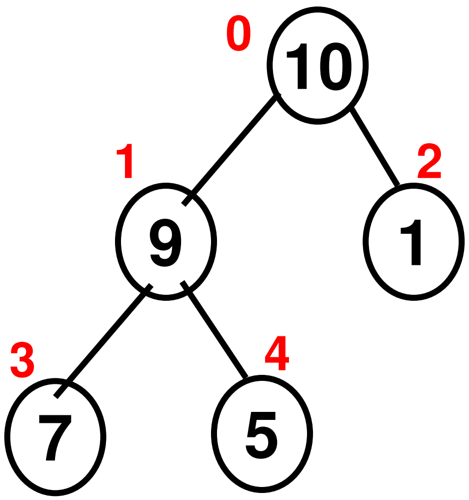
- Construct a sequence of priority queues (max heaps) with the joining of the elements 7, 9, 1, 10, 5, 8 one at a time.



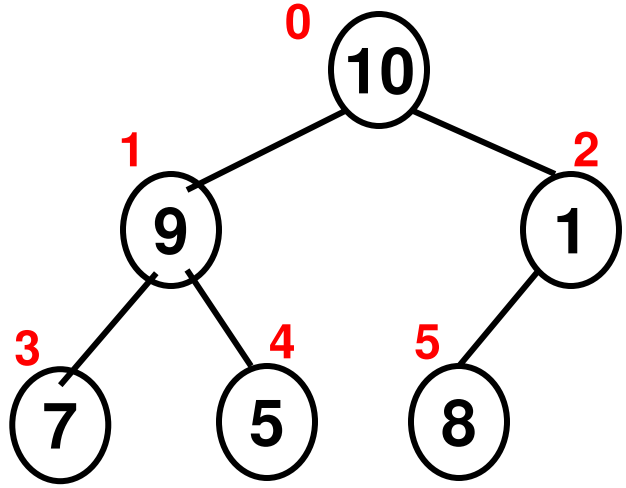
Priority Queue Construction: Example

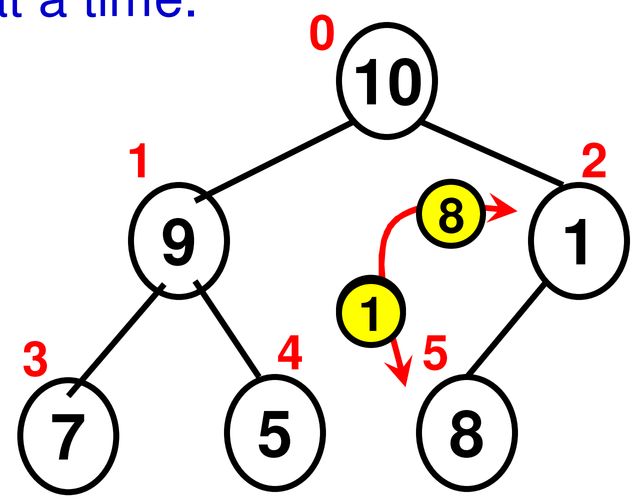
- Construct a sequence of priority queues (max heaps) with the joining of the elements 7, 9, 1, 10, 5, 8 one at a time.

<u>5. Enqueue of 5</u>

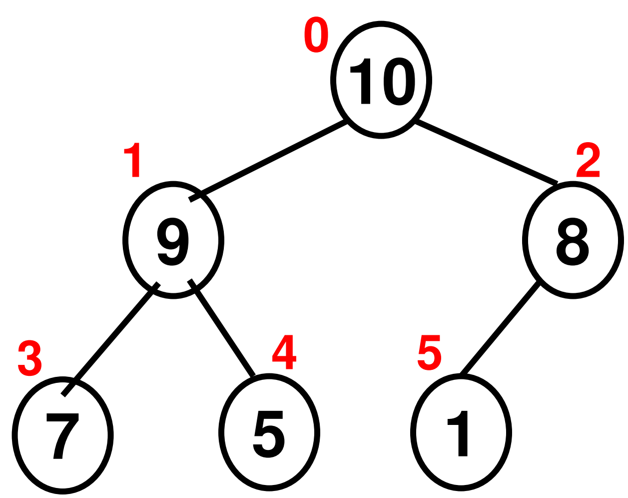


<u>6. Enqueue of 8</u>



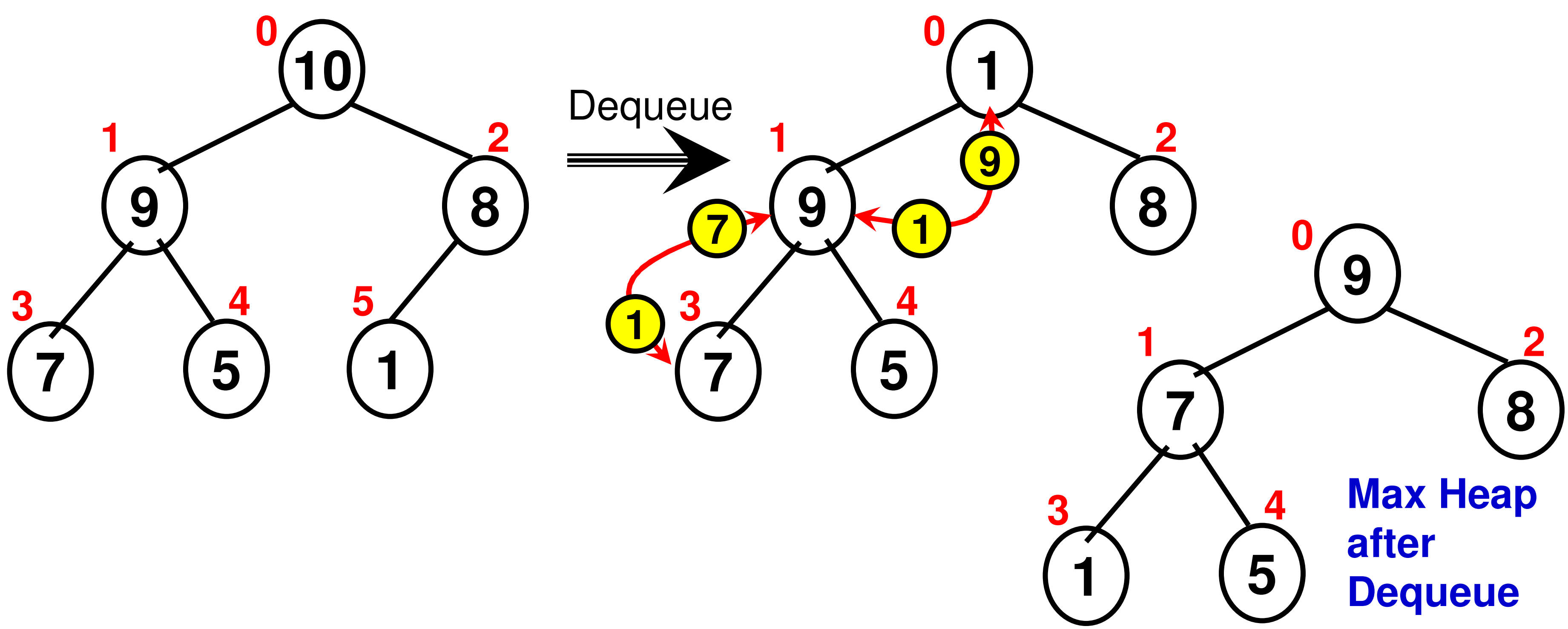


Final Max Heap Priority Queue



Dequeue of a Priority Queue (Max Heap)

- Remove the root node.
- Replace the data for the root node with the data of the element at the rightmost leaf node at the bottommost level, and remove the latter.
- Reheapify starting from the root node.

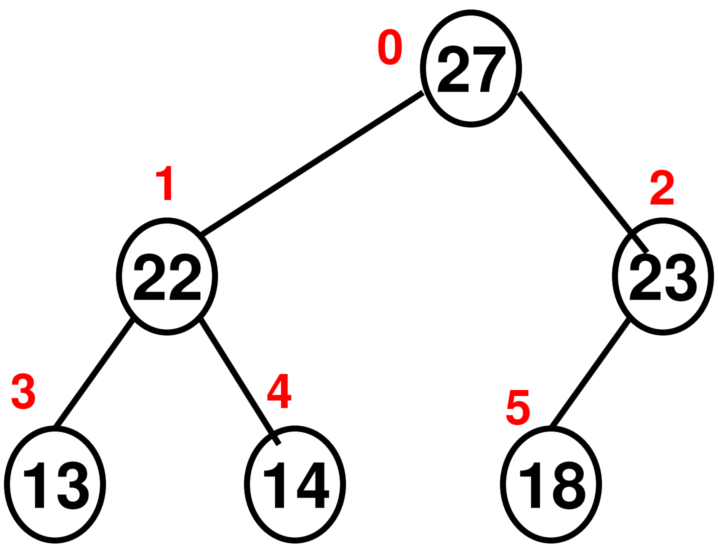


Binary Tree Transformations

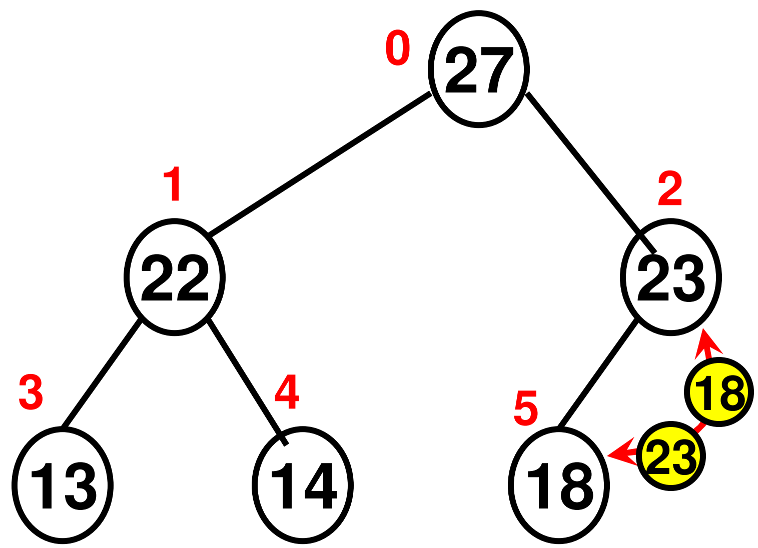
- Max Heap to Min Heap
  - Reheapify starting from the last internal node: at each internal node (and recursively down to a leaf node, if needed) so that the “min heap” property is satisfied.
- Min Heap to Max Heap
  - Reheapify starting from the last internal node: at each internal node (and recursively down to a leaf node, if needed) so that the “max heap” property is satisfied.
- Max Heap or Min Heap to a BST
  - Superimpose the sorted order of the data with the inorder listing of the indices of the heap
- BST to a Min Heap
  - Superimpose the inorder listing of the BST data with the Preorder listing of the indices of the BST
- BST to a Max Heap
  - Superimpose the inorder listing of the BST data with the Postorder listing of the indices of the BST

Max Heap to Min Heap (reheapify internal nodes): Example 1

- Given a Max Heap, reheapify (starting from the last internal node) every internal node to make sure the data at the internal node is lower than or equal to the data of its immediate child nodes.
- This would take O(n) time (like the transformation of an arbitrary essentially complete binary tree to max heap or min heap).

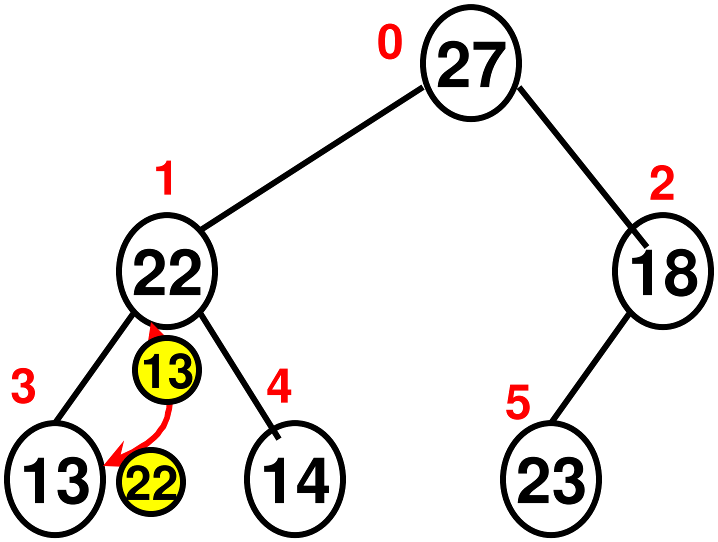


Given Max Heap

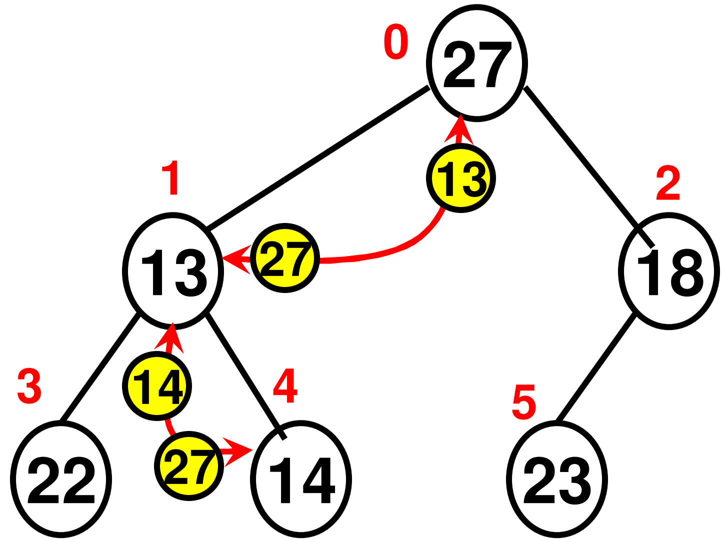


Reheapify at index 2

Max Heap to Min Heap: Example 1

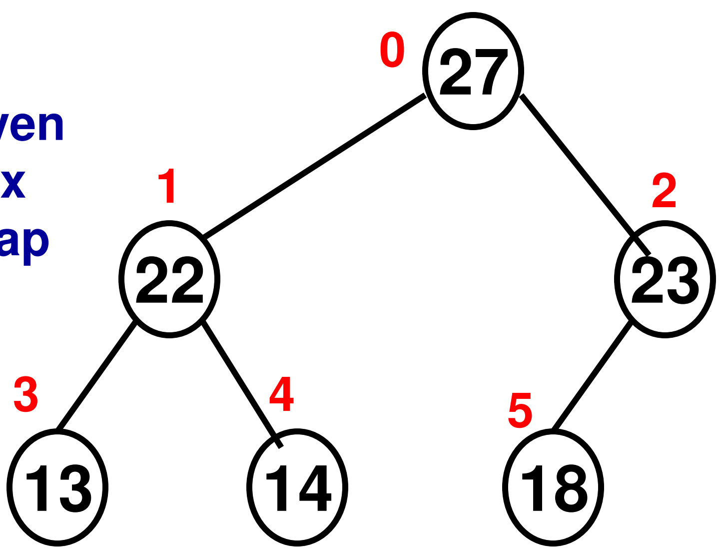


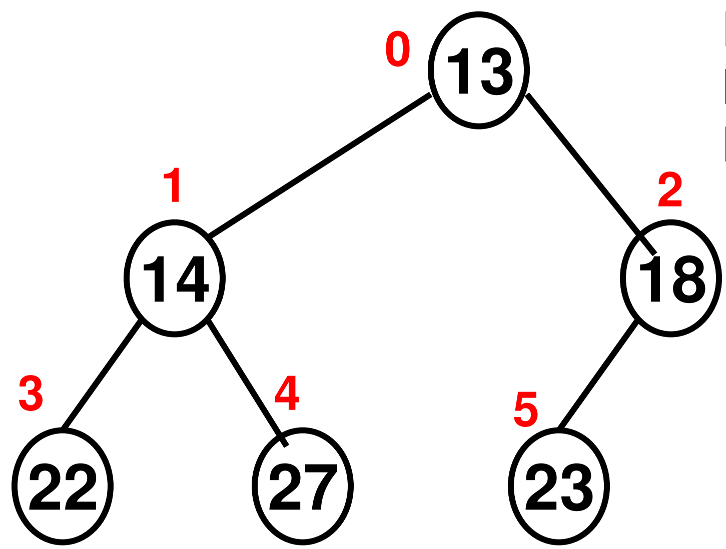
Reheapify at index 1



Reheapify at index 0

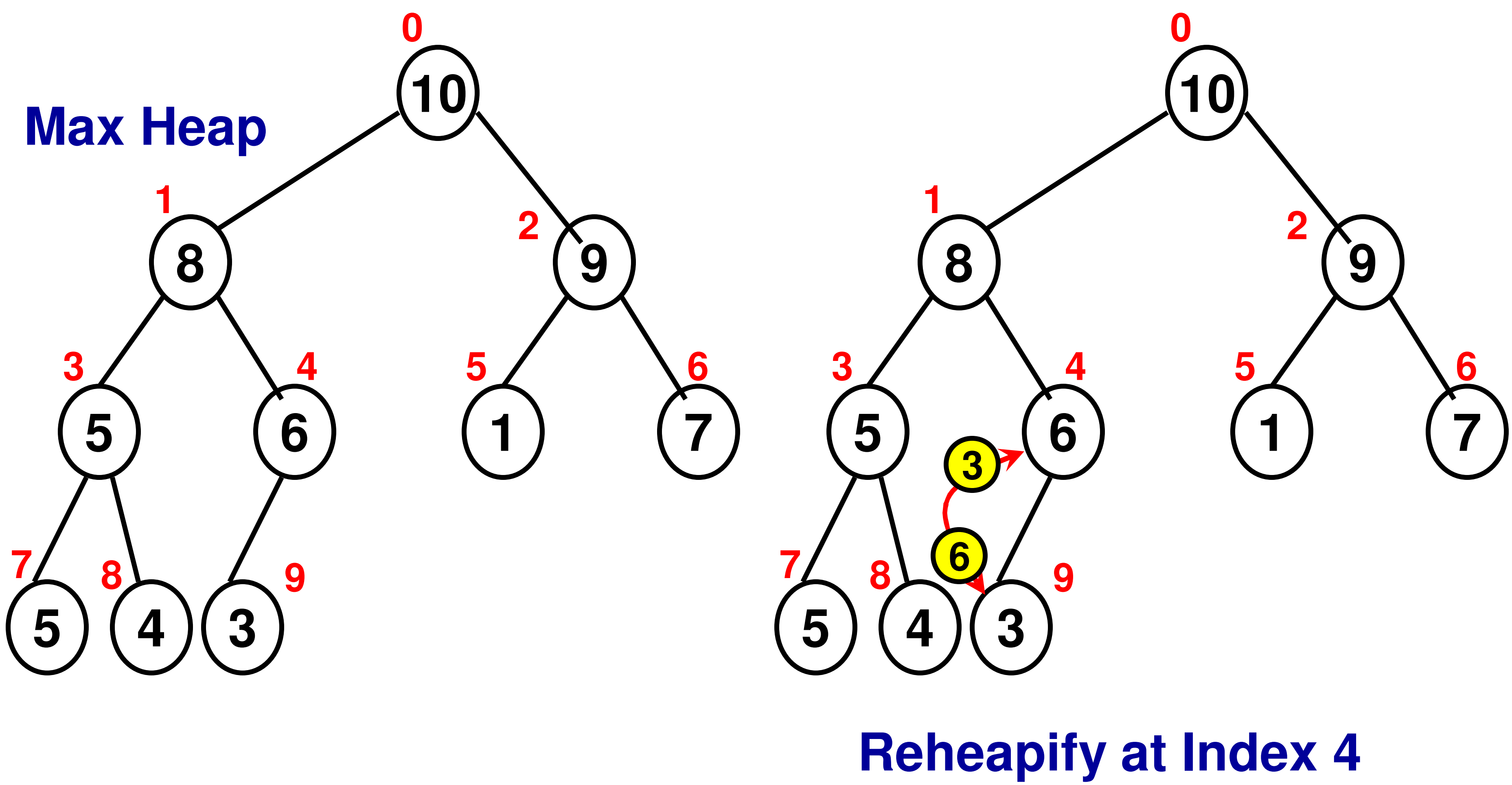
Given Max Heap



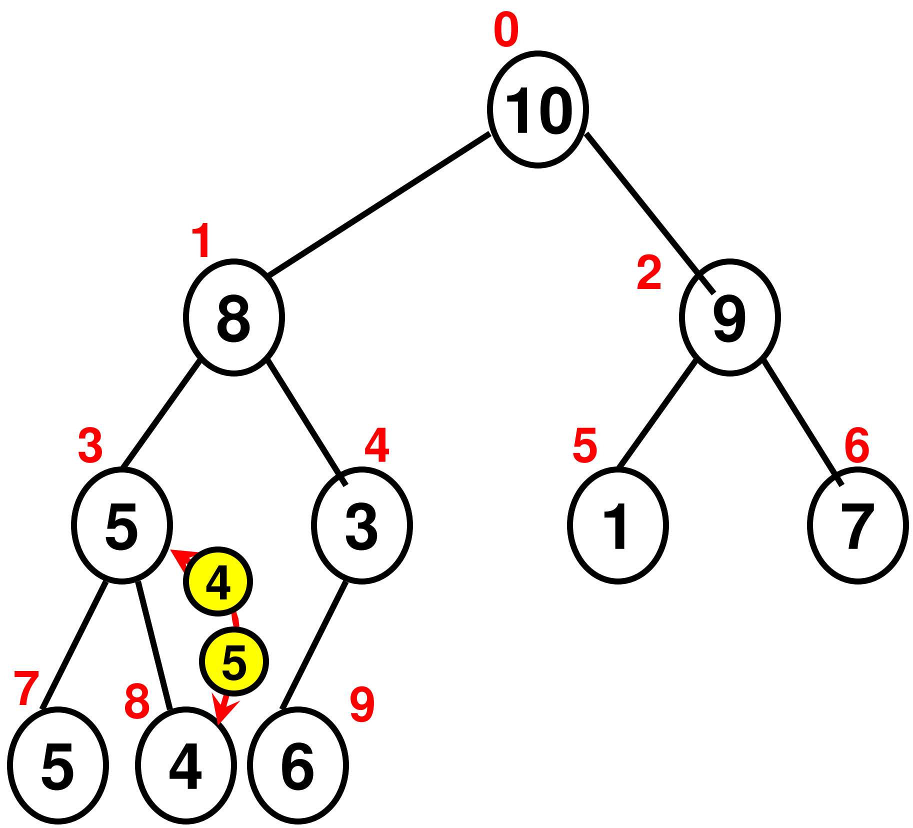


Final Min Heap

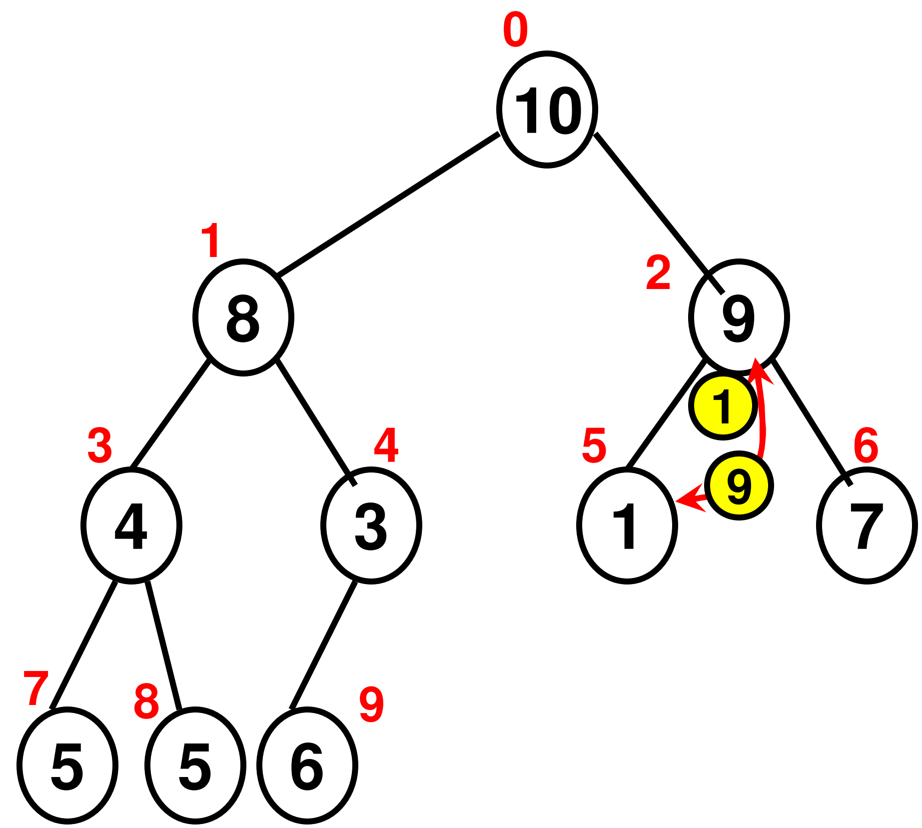
Max Heap to Min Heap: Example 2



Max Heap to Min Heap: Example 2

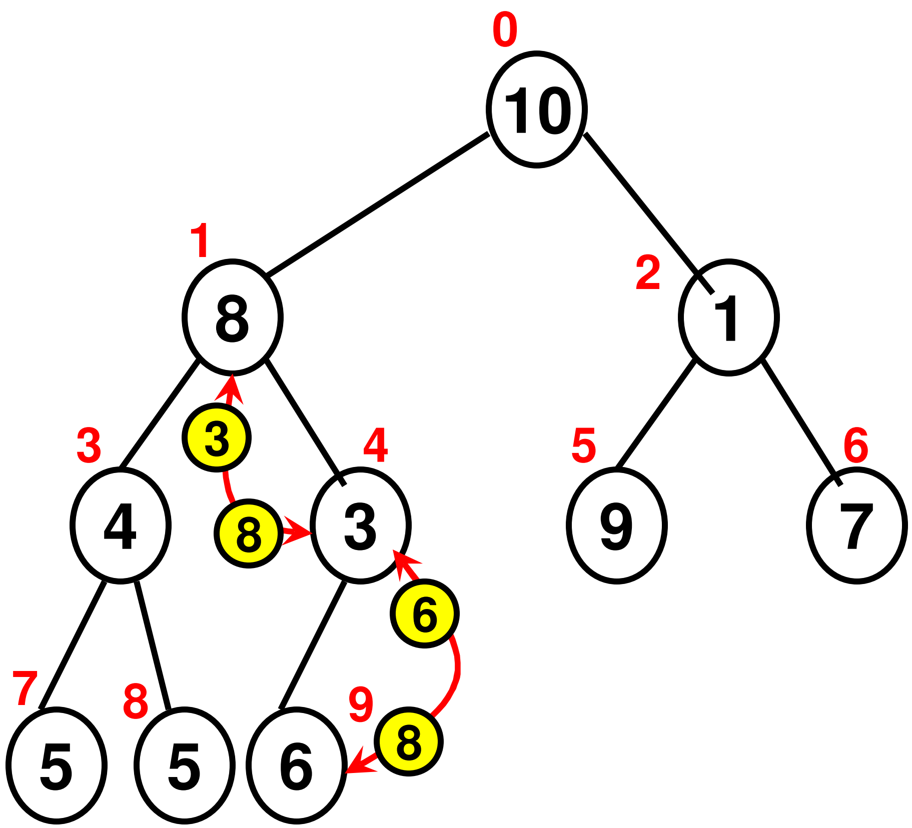


Reheapify at Index 3

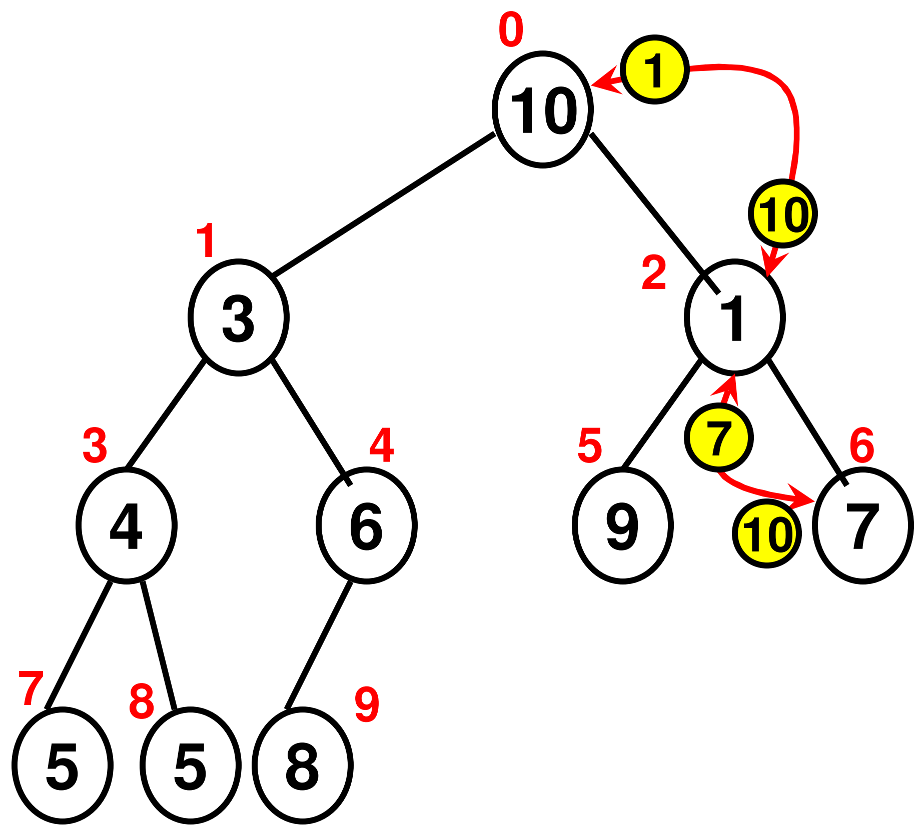


Reheapify at Index 2

Max Heap to Min Heap: Example 2

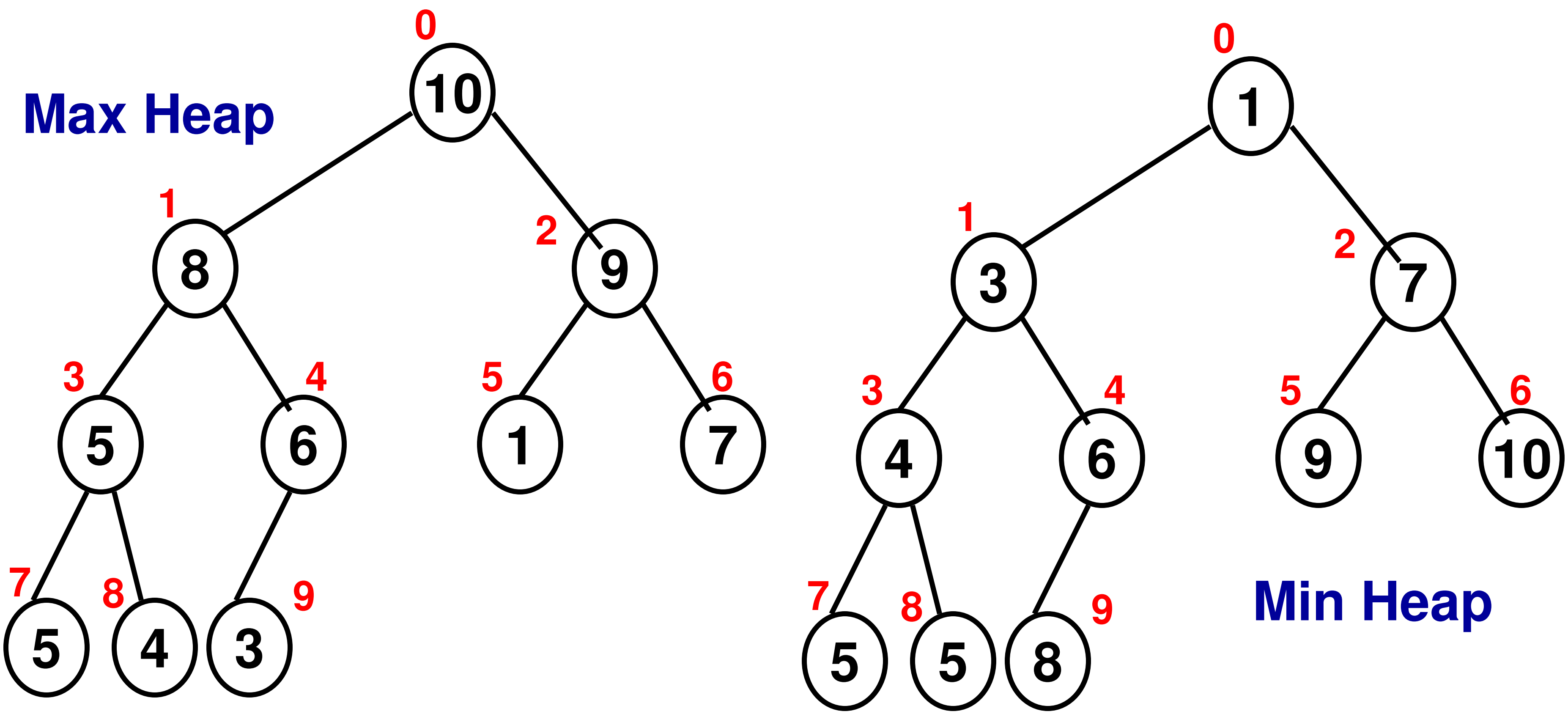


Reheapify at Index 1



Reheapify at Index 0

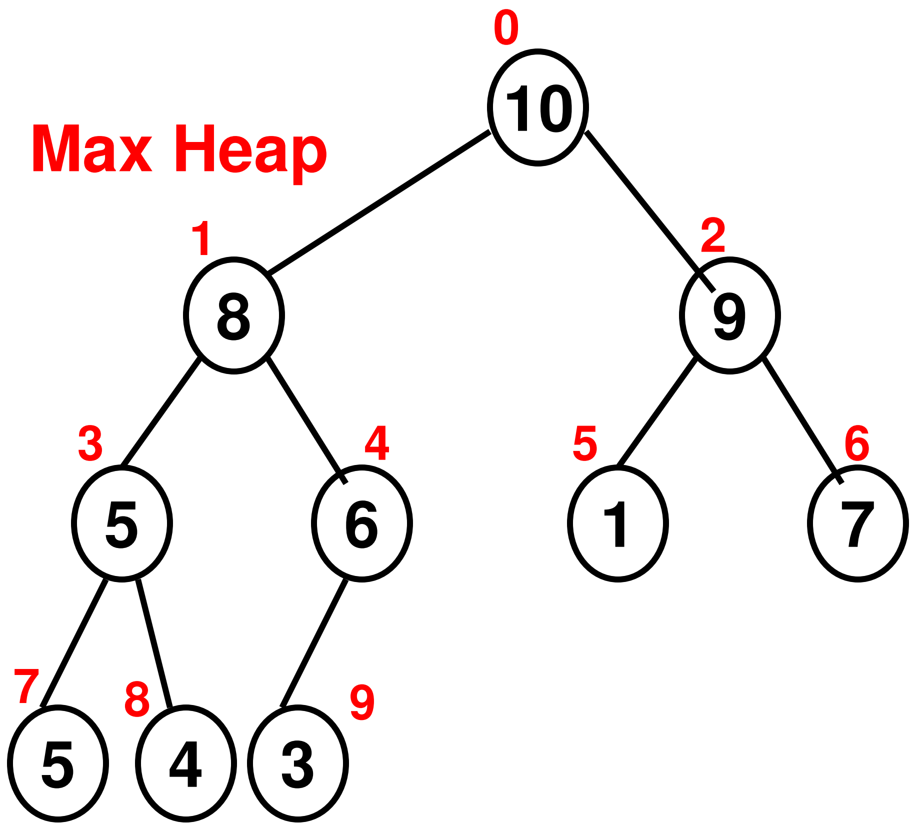
Max Heap to Min Heap: Example 2



Min Heap or Max Heap to a BST

- Superimpose the sorted order of the data with the inorder listing of the indices.

Max Heap



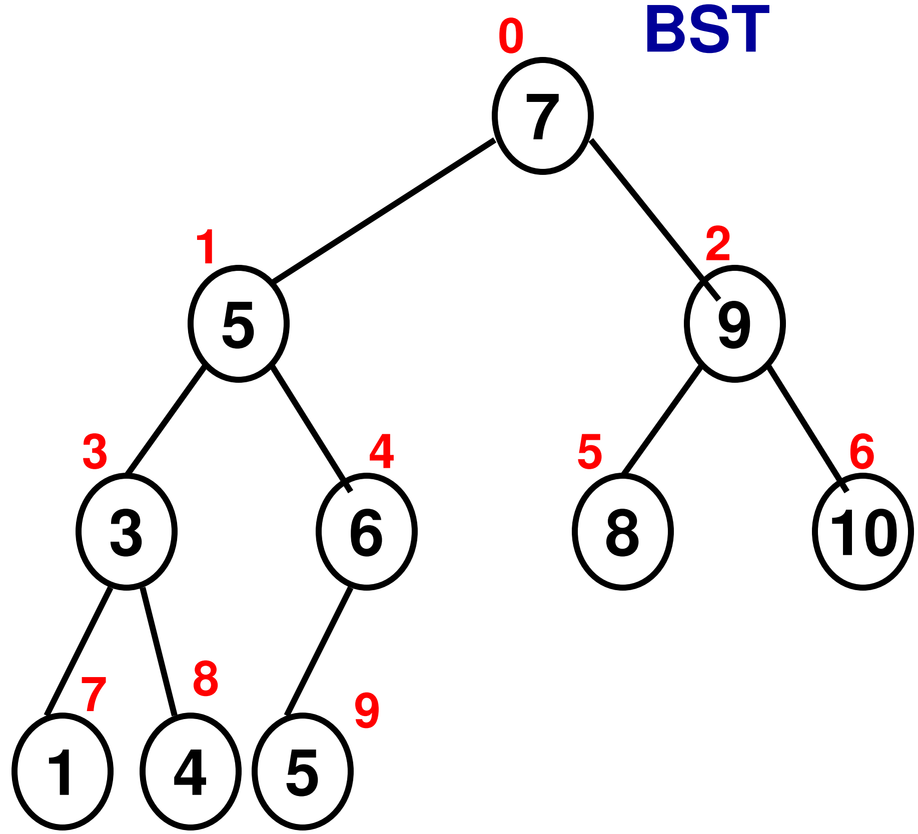
Inorder Traversal

Index	7	3	8	1	9	4	0	5	2	6
Data	5	5	4	8	3	6	10	1	9	7

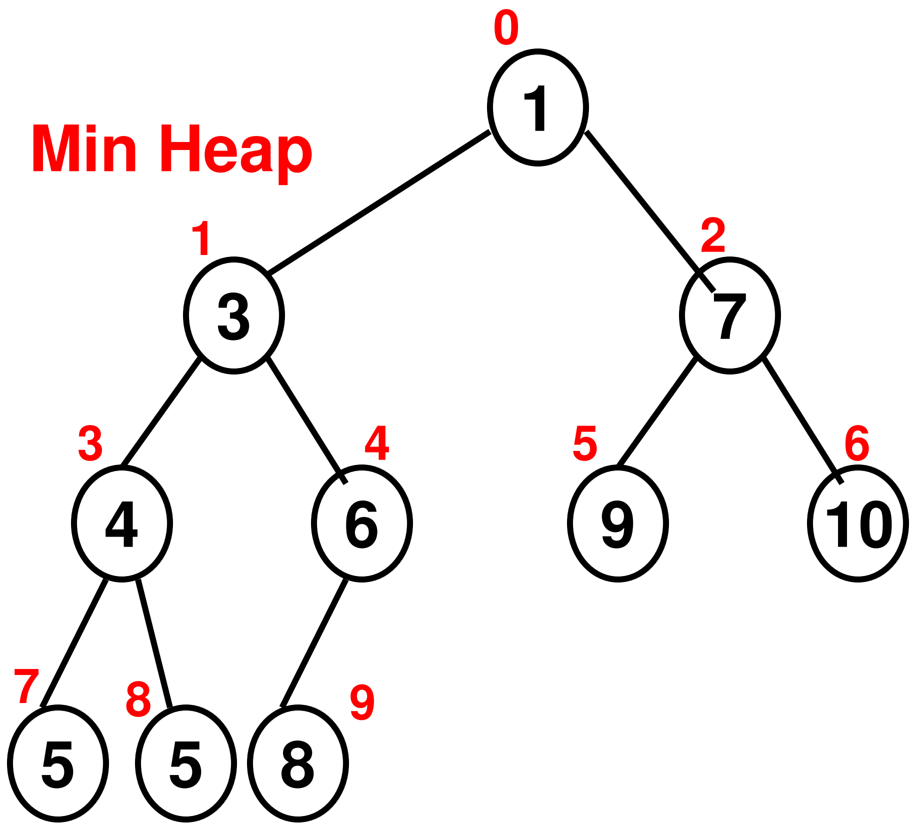
Sort just the data

Index	7	3	8	1	9	4	0	5	2	6
Data	1	3	4	5	5	6	7	8	9	10

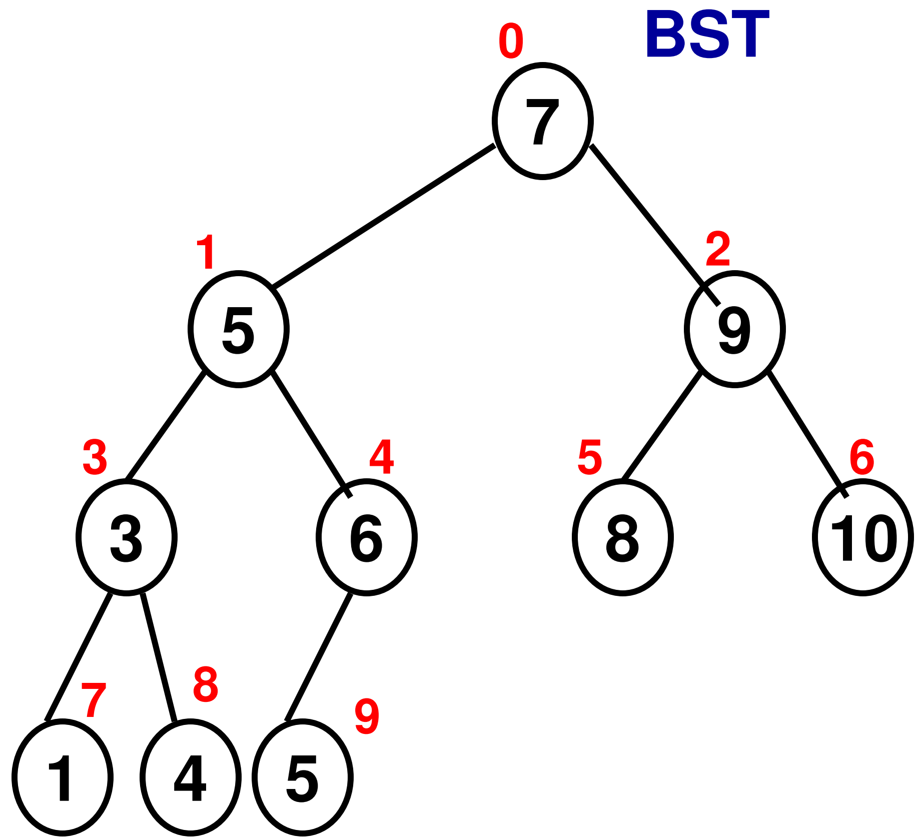
BST



Min Heap or Max Heap to a BST



- Superimpose the sorted order of the data with the inorder listing of the indices.



Inorder Traversal

Index	7	3	8	1	9	4	0	5	2	6
Data	5	4	5	3	8	6	1	9	7	10

Sort just the data

Index	7	3	8	1	9	4	0	5	2	6
Data	1	3	4	5	5	6	7	8	9	10

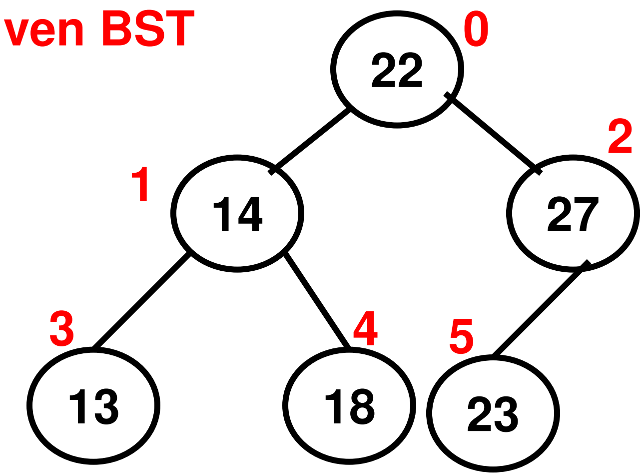
Binary Search Tree to a Min (Max) Heap

- **BST to Min Heap**
- Superimpose the inorder listing of the BST data with the Preorder listing of the indices of the BST
  - **Details**
  - Step 1: Perform an inorder traversal of the BST and create an array of the sorted integers of the data corresponding to the nodes in the BST.
  - Step 2: Perform a preorder traversal of the BST. While performing the preorder traversal, replace the data at each node visited with the values of the inorder array. The resulting binary tree is a min heap.
- **Since the min heap is generated from a BST, the min heap has the following property (need not be observed when directly obtained from a max heap):**
  - **For any internal node: the data of all the nodes in the left sub tree are less than or equal to the data of all the nodes in the right sub tree.**

- **BST to Max Heap**
- Superimpose the inorder listing of the BST data with the Postorder listing of the indices of the BST
  - **Details**
  - **Do Step 1 as above (i.e., inorder traversal of the BST)**
  - **For Step 2, do a postorder traversal of the BST.**

BST to Min Heap: Example 1

Given BST



Inorder-based data

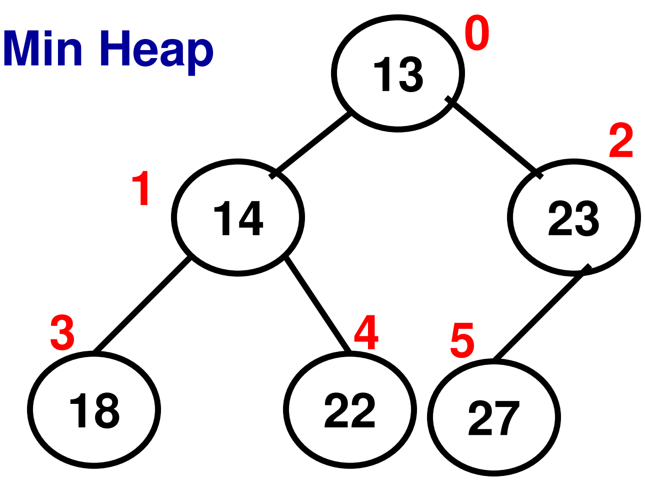
13 14 18 22 23 27

Preorder Listing of the node indices

0 1 3 4 2 5

Superimpose the inorder data with the Preorder listing of the indices

Min Heap

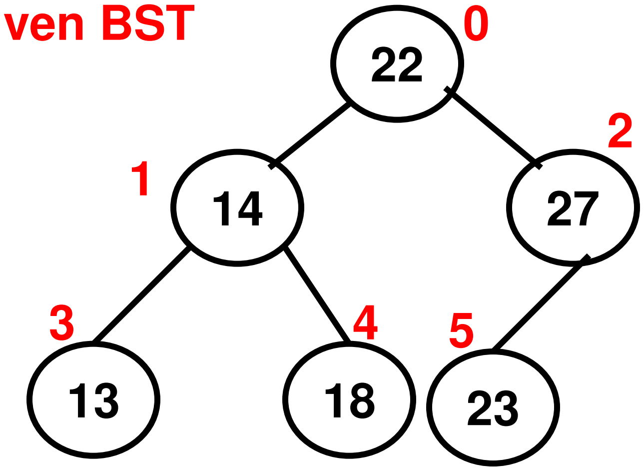


Min heap

0	1	3	4	2	5
13	14	18	22	23	27

BST to Max Heap: Example 1

Given BST



Inorder-based data

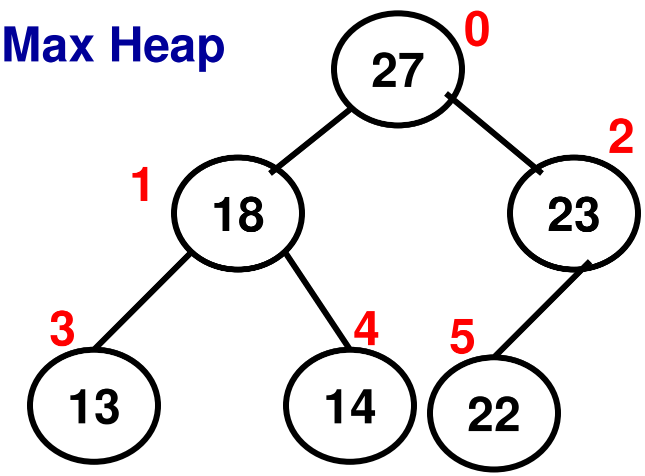
13 14 18 22 23 27

Postorder Listing of the node indices

3 4 1 5 2 0

Superimpose the inorder data with the Postorder listing of the indices

Max Heap

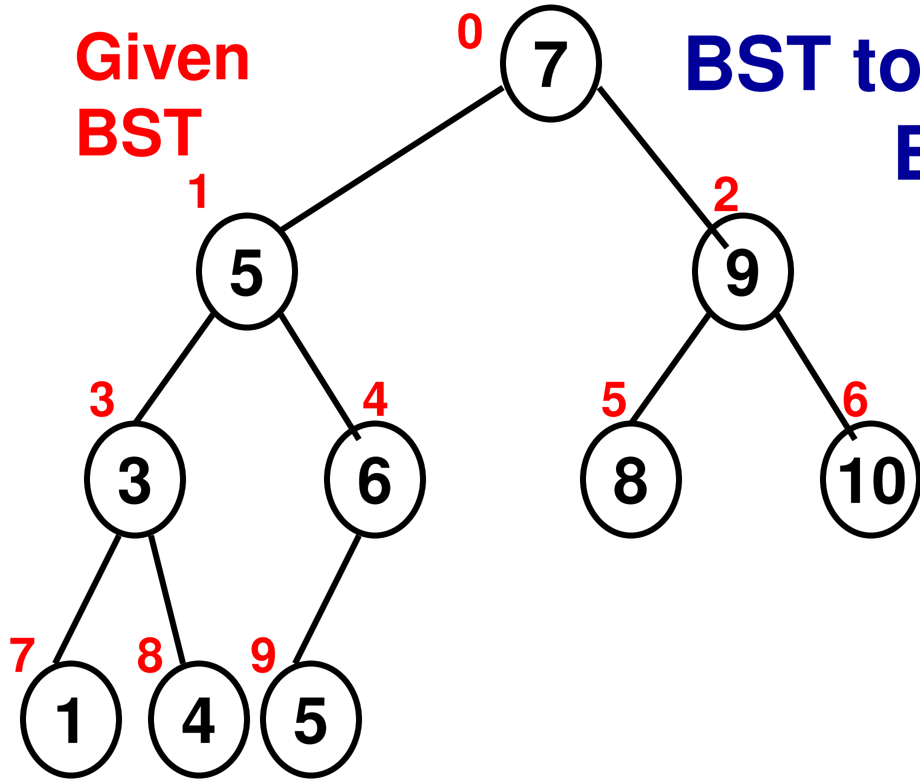


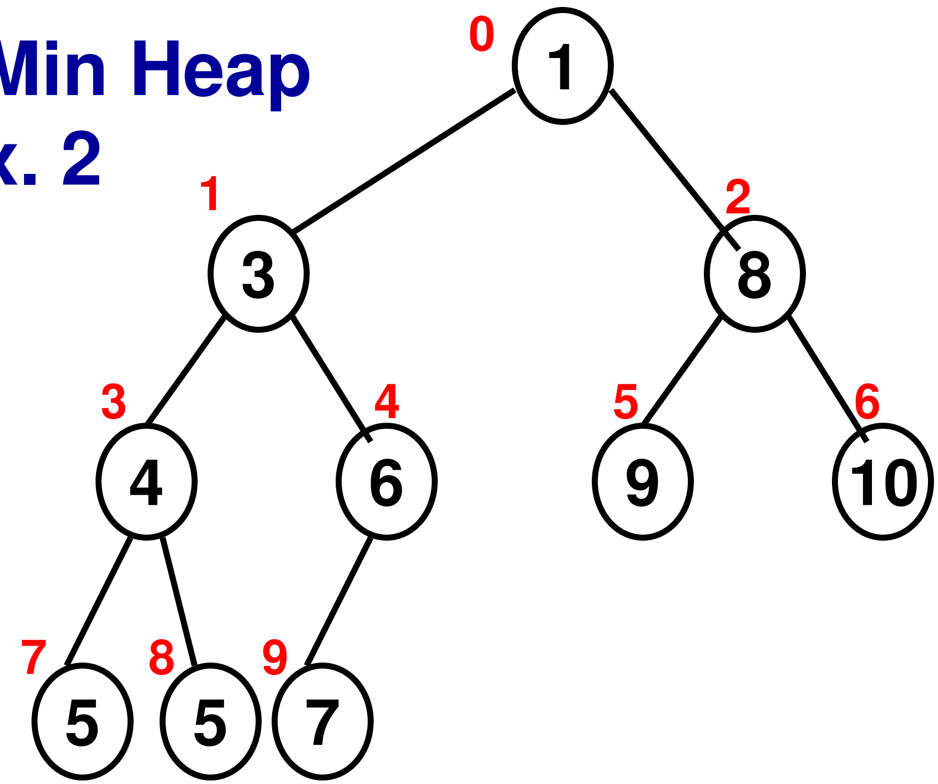
Max heap

3	4	1	5	2	0
13	14	18	22	23	27

BST to Min Heap
Ex. 2

Given BST





Inorder-based data

1	3	4	5	5	6	7	8	9	10

Preorder Listing of the node indices

0	1	3	7	8	4	9	2	5	6

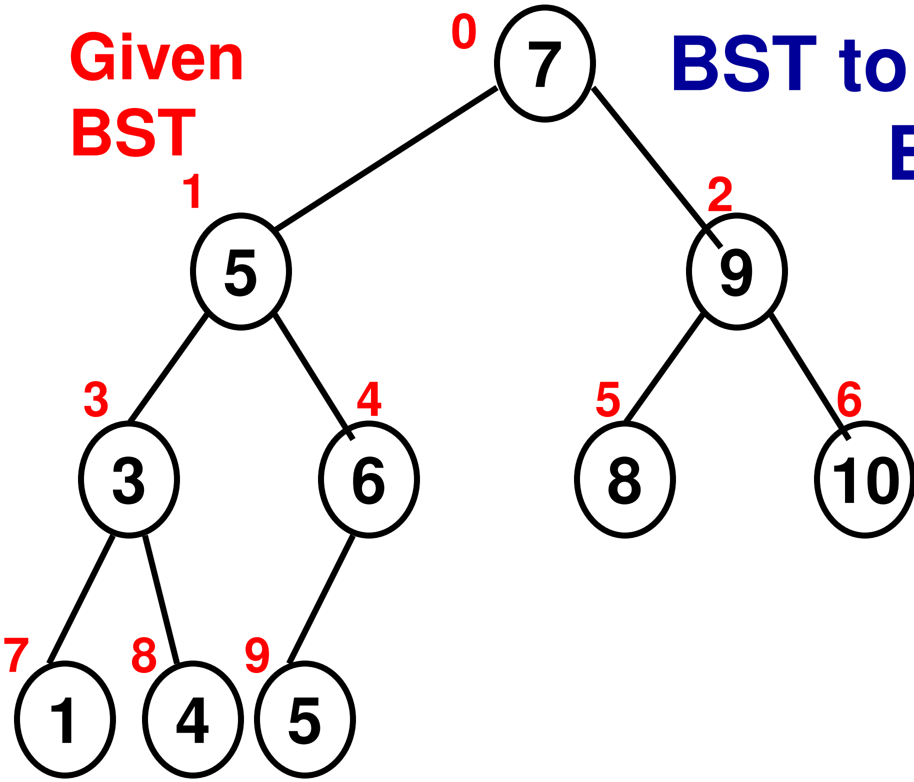
Min Heap

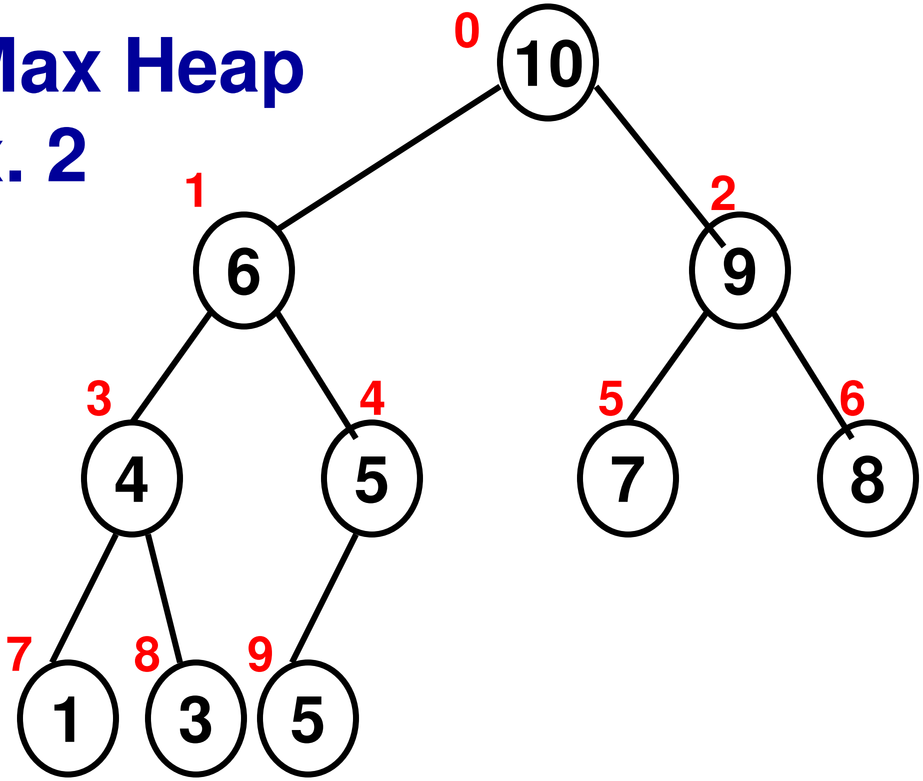
0	1	3	7	8	4	9	2	5	6
1	3	4	5	5	6	7	8	9	10

Note that all the nodes in the left sub tree of an Internal node have data that is less than or equal to the data of the nodes in the right sub tree.

BST to Max Heap
Ex. 2

Given BST





Inorder-based data

1	3	4	5	5	6	7	8	9	10

Postorder Listing of the node indices

7	8	3	9	4	1	5	6	2	0

Max Heap

7	8	3	9	4	1	5	6	2	0
1	3	4	5	5	6	7	8	9	10

Note that all the nodes in the left sub tree of an Internal node have data that is less than or equal to the data of the nodes in the right sub tree.