CSCI 2270-305 Recitation 04/16: Priority Queues and Heaps

Varad Deshmukh

Logistics

- We are done with Assignments!
  - If you forgot to click on Submit All and Finish in your past homeworks, email me.
  - Score will default to 0 otherwise.

- Mandatory midterms are done!
  - Good job on the second midterm! Grades will be available soon.
  - **Optional midterm:** Attempt only if you are confident of scoring higher than your least score, or aren't currently meeting the requirements.

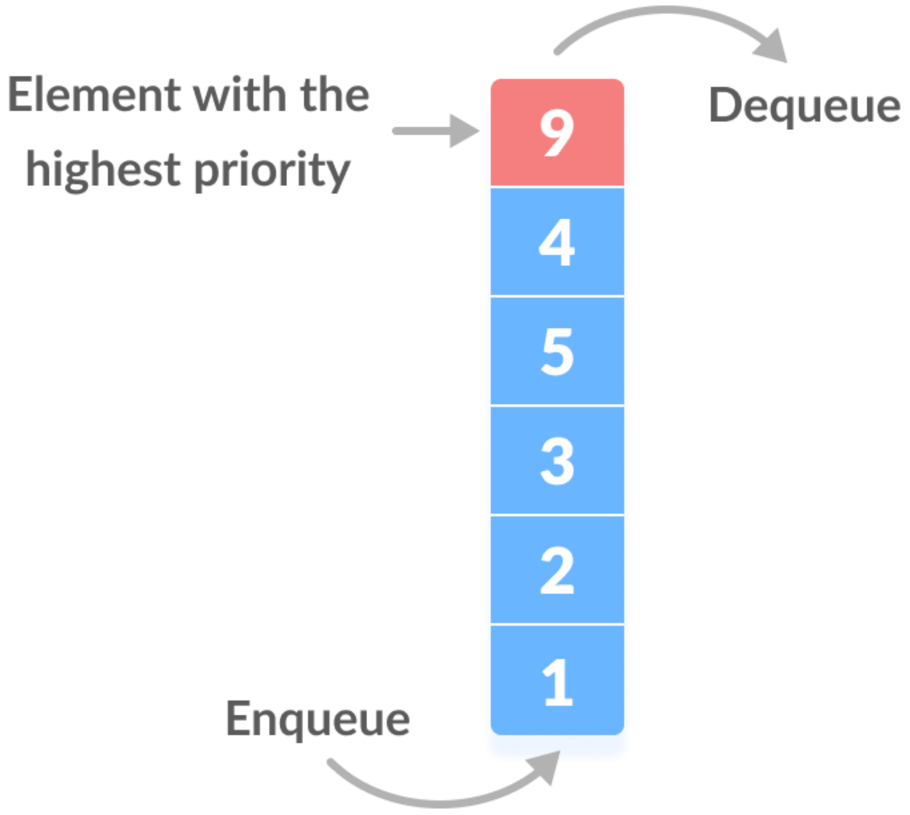
- We are done with Recitations after today! Submit the Recitation 13 work by Sunday.

- Interview Grading next week.
  - Can make up for one assignment from any of the 9 assignments.
  - Schedule: Wednesday 3-5 pm, Thursday 330-5 pm.

- Final project is live, due April 29th.

Priority Queues (ADT)

Priority Queue is a queue that maintains elements based on their priority, and **always dequeues the highest priority element.**



Queues v/s Priority Queues

	Queues	Priority Queues
	First in First Out	Highest priority out
Enqueue	Enqueue at the tail	Enqueue at tail and reorder
Dequeue	Dequeue the oldest item at the front	Dequeue the highest priority at the front
Requires re-ordering	No	Yes

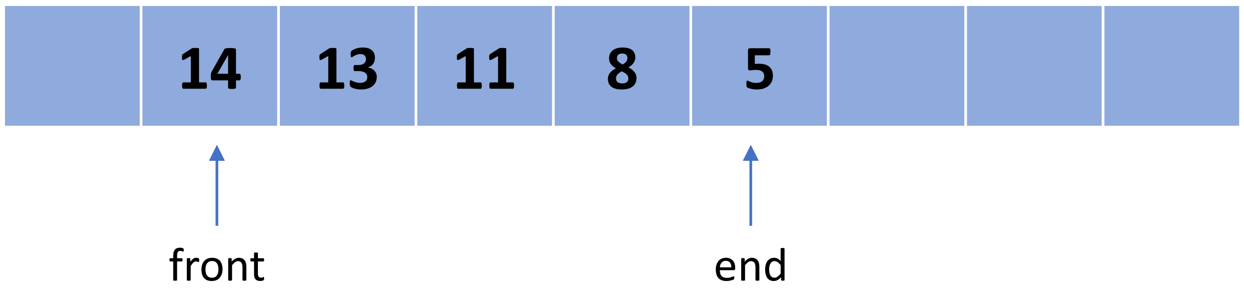
Operations: Queues v/s Priority Queues

Queues	Priority Queues
Enqueue (Tail)	Insert
Dequeue (Front)	DeleteMin/DeleteMax (Highest Priority)

Implementation Ideas (High value = High priority)

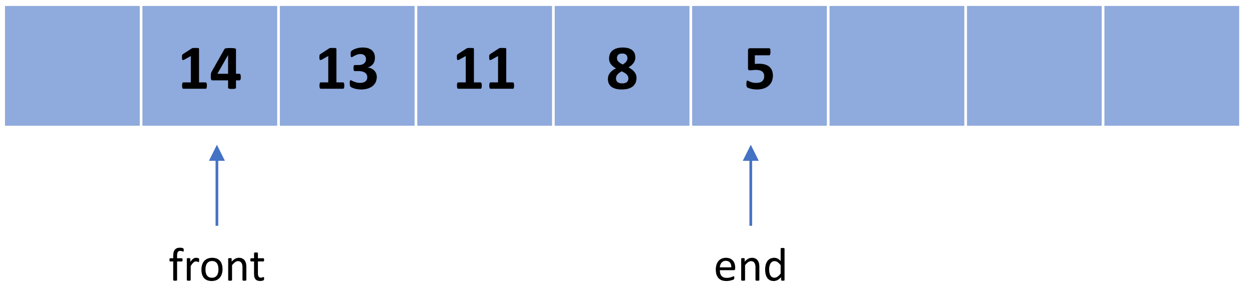
Implementation Ideas (High value = High priority)

1. Sorted Circular Array



Implementation Ideas (High value = High priority)

1. Sorted Circular Array

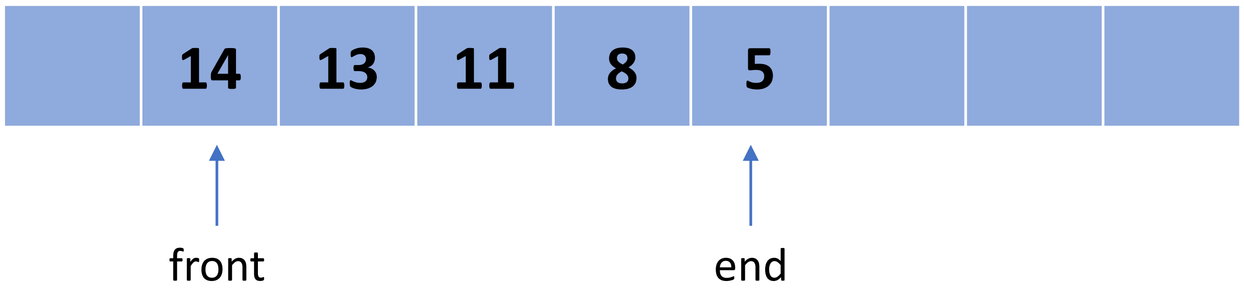


Insertion (9):

Deletion/Dequeueing:

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

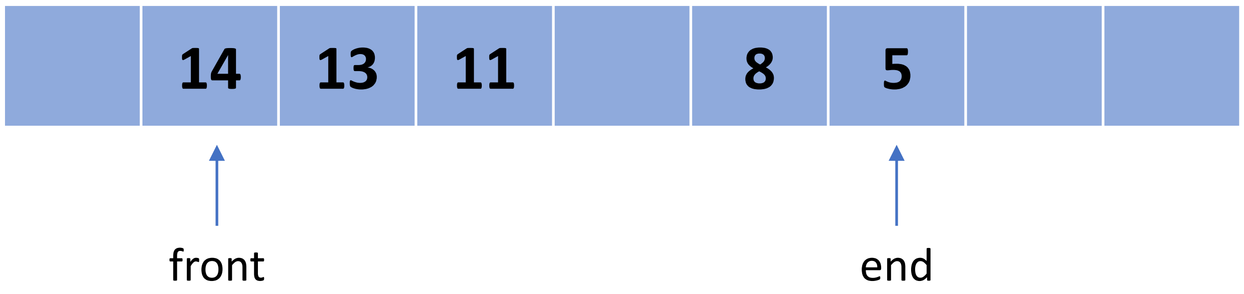


Insertion (9): Modulo shift the end of the queue and insert to keep the array sorted.

Deletion/Dequeueing:

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

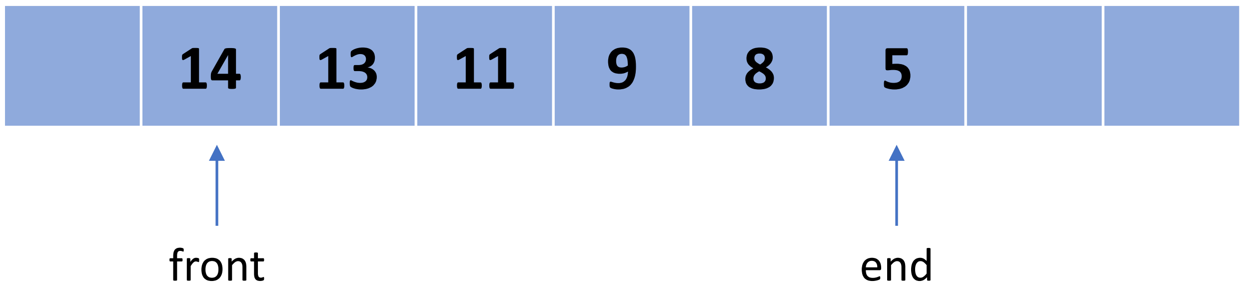


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Deletion/Dequeueing:

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

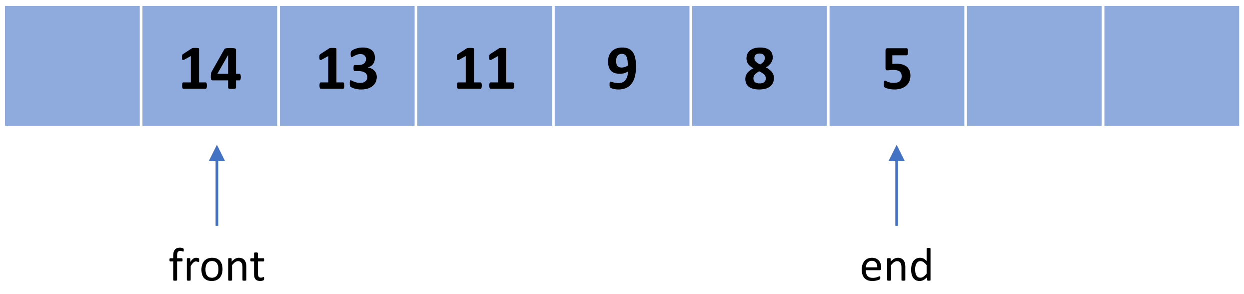


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Deletion/Dequeueing:

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

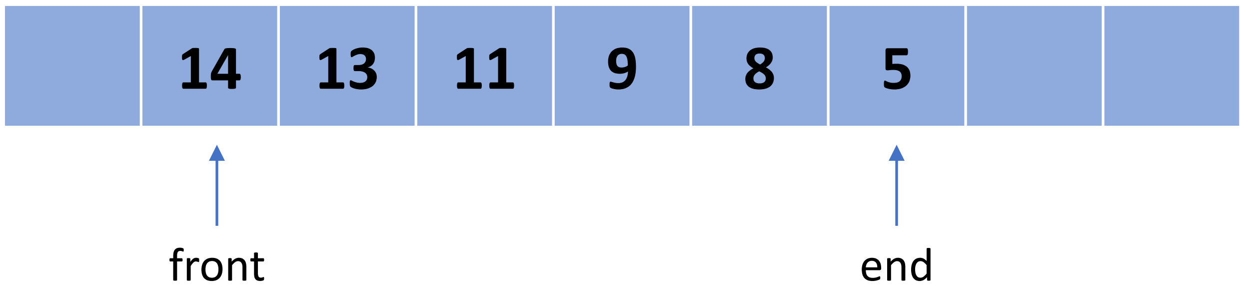


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Deletion/Dequeueing:

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

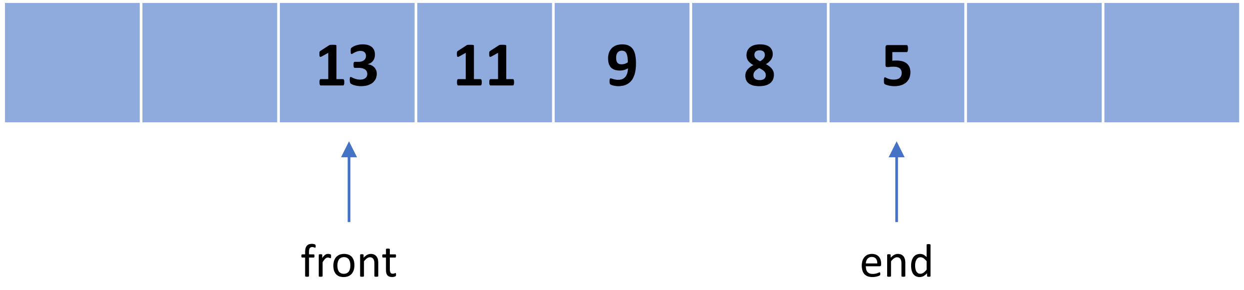


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Deletion/Dequeueing: Modulo increment the front.

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

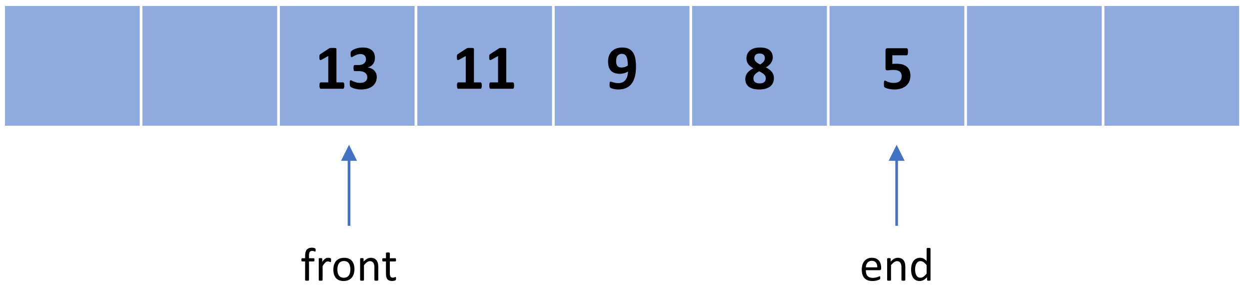


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Deletion/Dequeueing: Modulo increment the front.

Implementation Ideas (High value = High priority)

1. Sorted Circular Array

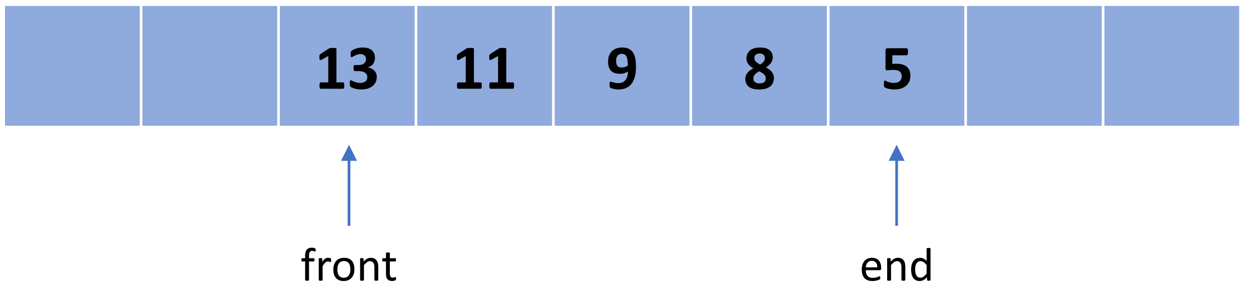


Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)
Complexity =

Deletion/Dequeueing: Modulo increment the front. Complexity =

Implementation Ideas (High value = High priority)

1. Sorted Circular Array



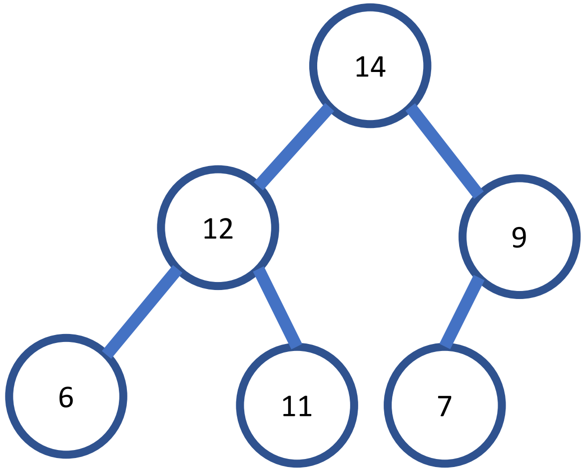
Insertion (9): Modulo shift the end of the queue & insert (keep the array sorted.)

Complexity = O(n).

Deletion/Dequeueing: Modulo increment the front. Complexity = O(1).

Max-Heap as a Priority Queue

2. Heap



Insertion: Insert at the end of the heap, and reorder.

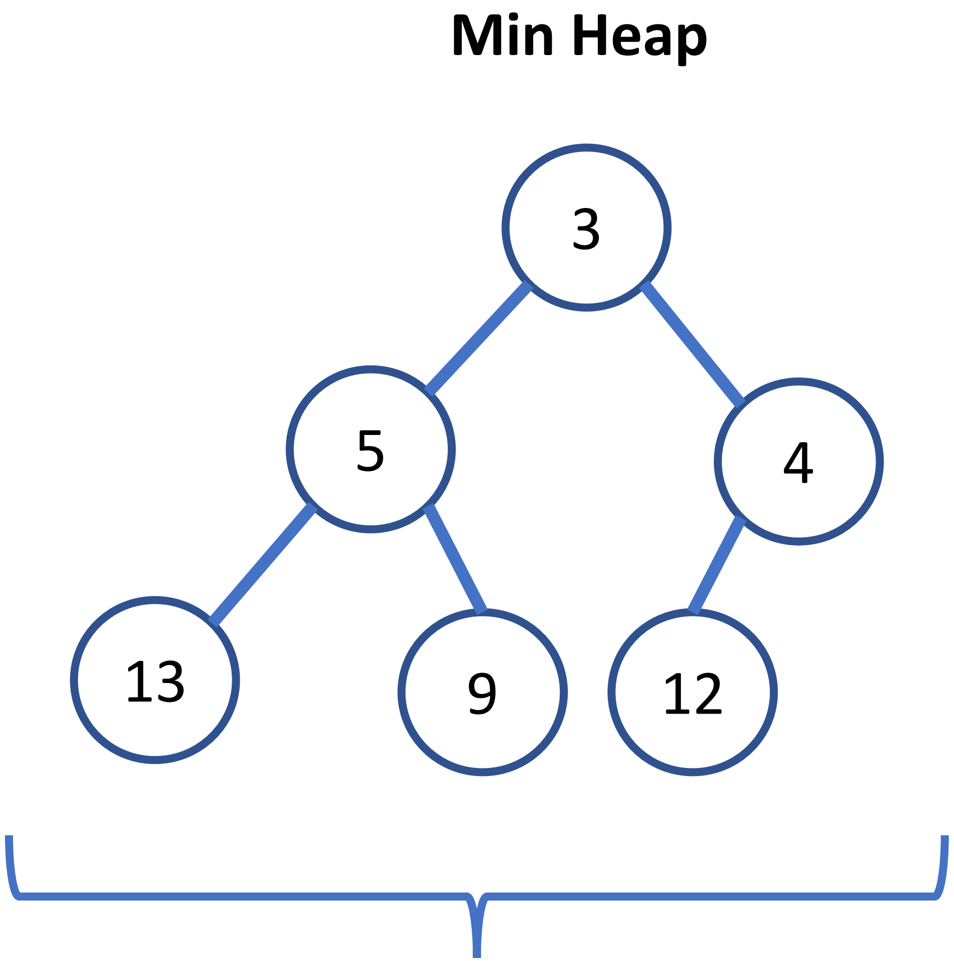
Deletion/Dequeueing: Delete the root, and reorder.

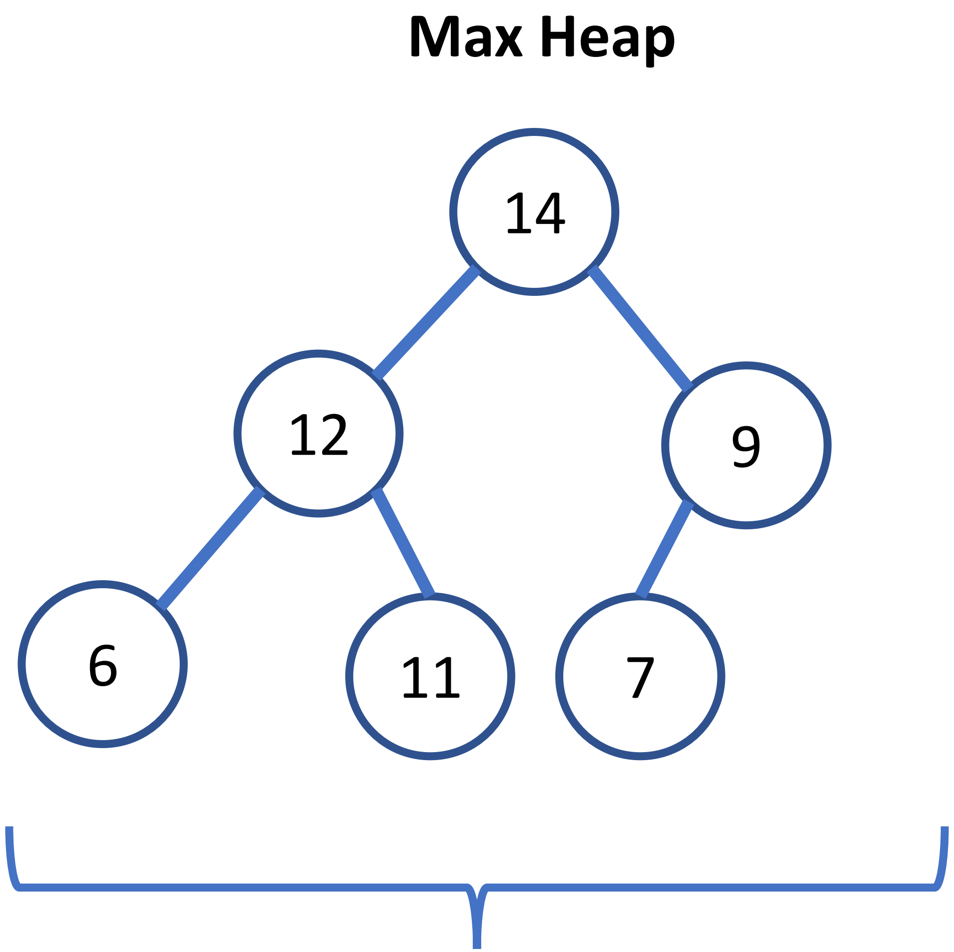
Heaps

- A heap is a tree-like data-structure, which:
  - Is an almost complete tree.
  - Satisfies a certain property of node ordering:
  - Root of a heap is the highest priority element in the heap.

- We will restrict ourselves to studying **binary** heaps.

Min and Max-Heaps

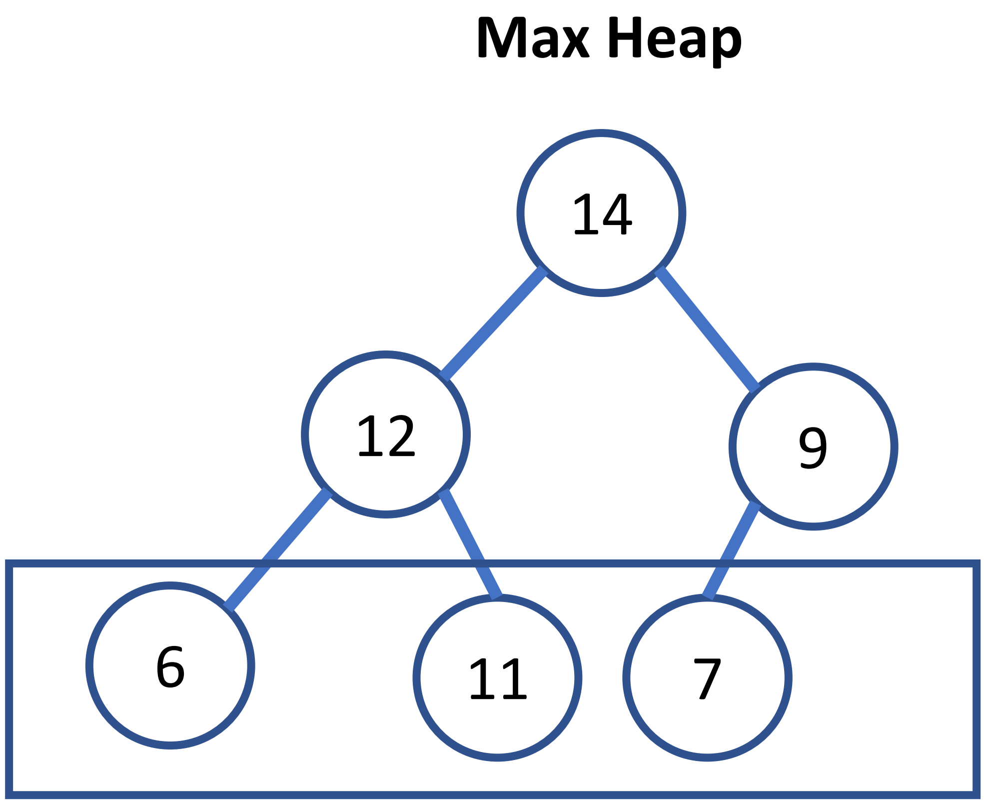




Each parent is smaller than either children.

Each parent is greater than either children.

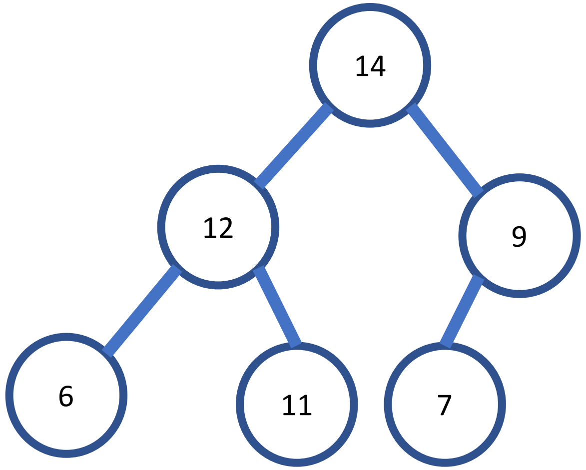
An Almost complete tree



The last level may not be complete (contains 3 out of 4 nodes.)

Max-Heap as a Priority Queue

2. Heap



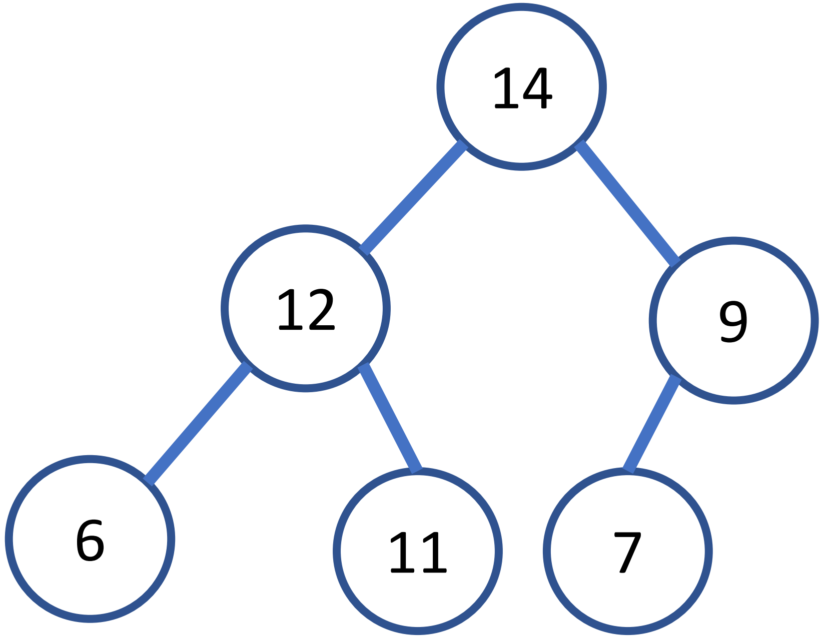
Insertion: Insert at the end of the heap, and reorder. **Complexity = O (lg n).**

Deletion/Dequeueing: Delete the root, and reorder. **Complexity = O (lg n).**

Implementing a Binary Max Heap

We could use the traditional BST structure we have studied in the class,

Max Heap

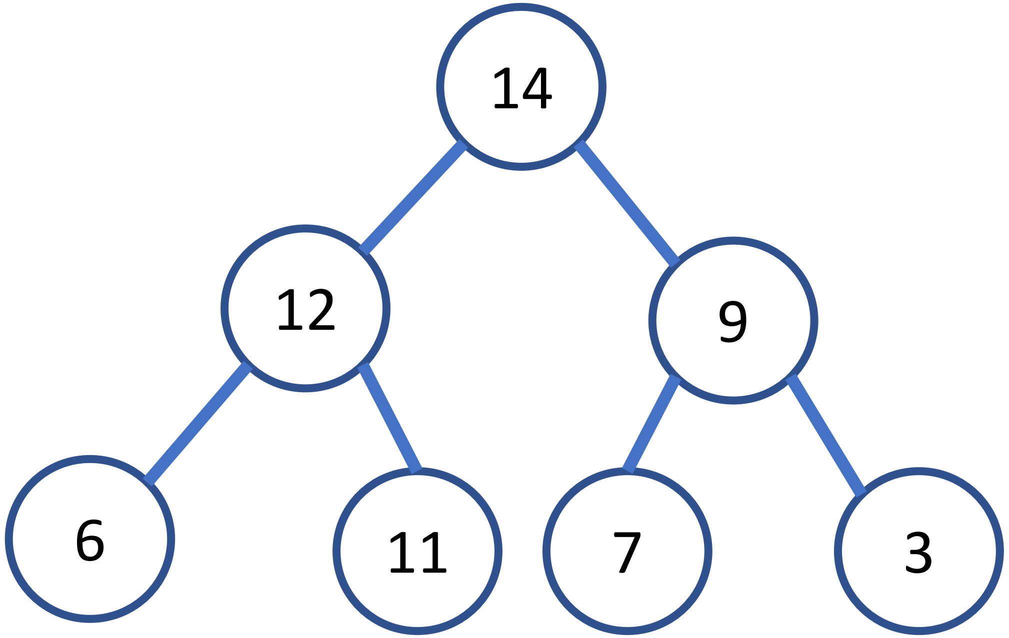



```
struct BSTNode {
    int key;
    BSTNode * left;
    BSTNode * right;
}
```

Implementing a Binary Max Heap

or use a more efficient array implementation.

Max Heap



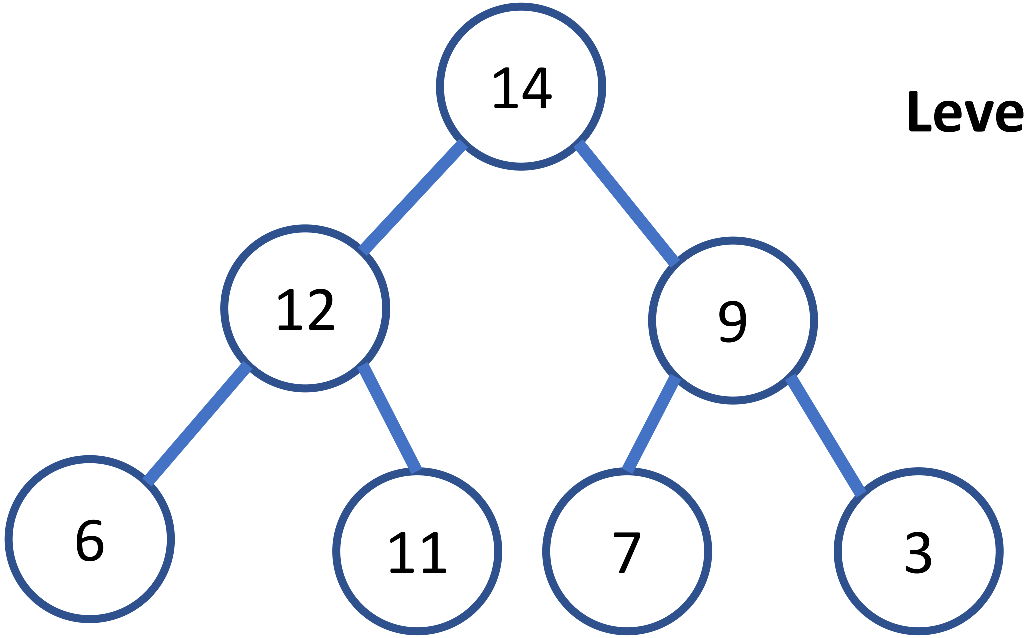
14	**12**	**9**	**6**	**11**	**7**	**3**	

… by exploiting the almost-complete tree structure.

Implementing a Binary Max Heap

or use a more efficient array implementation.

Max Heap



Level	**0**	**1**	**1**	**2**	**2**	**2**	**2**	
	14	**12**	**9**	**6**	**11**	**7**	**3**	

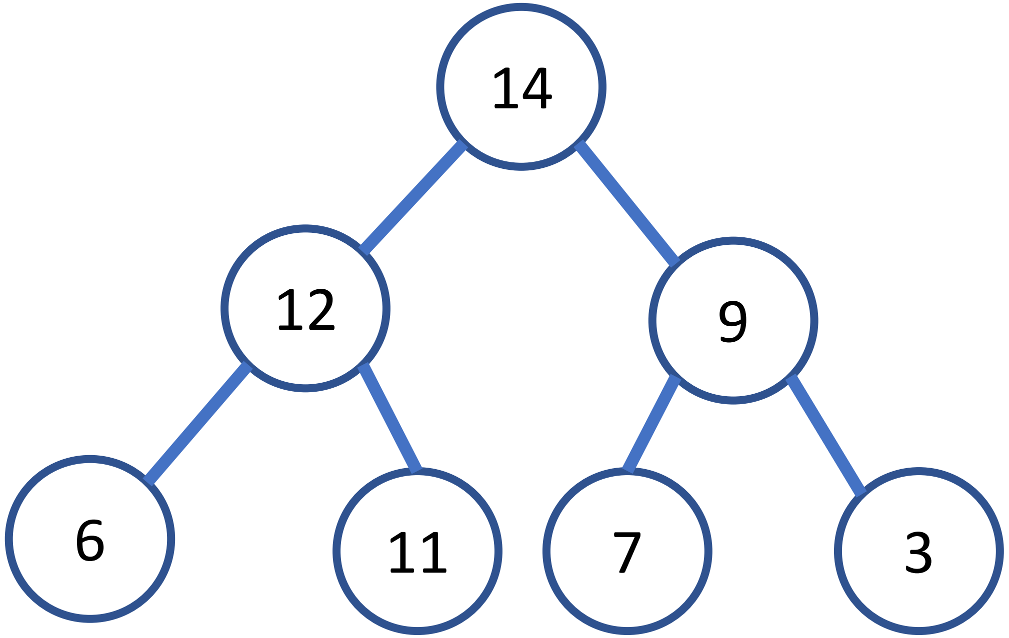
… by exploiting the almost-complete tree structure.

Implementing a Binary Max Heap

or use a more efficient array implementation.

Intermediate values are always valid

Max Heap

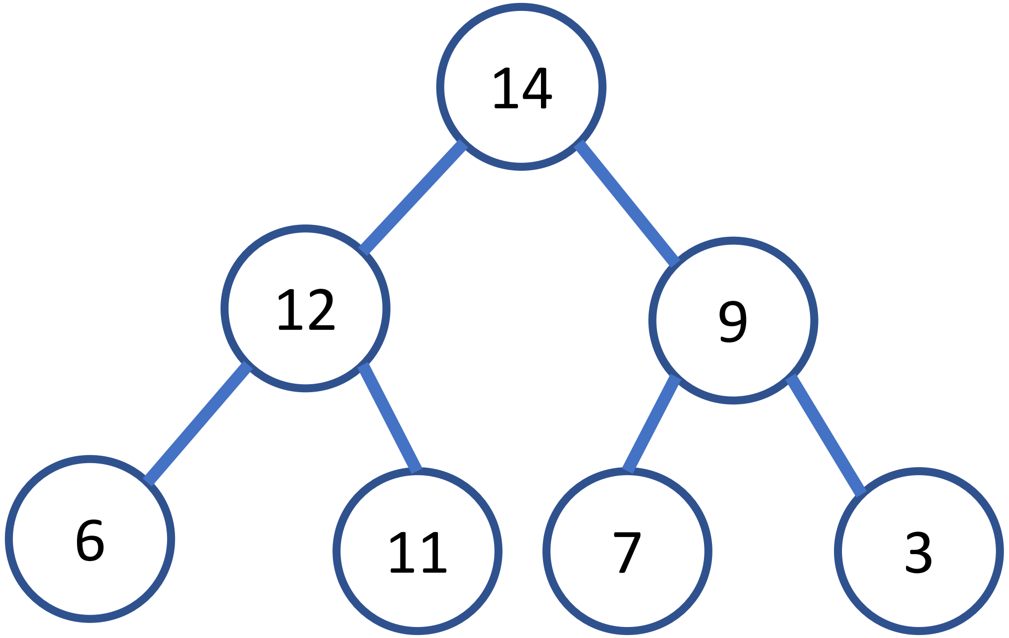


0	1	1	2	2	2	2	
14	**12**	**9**	**6**	**11**	**7**	**3**	

… by exploiting the almost-complete tree structure.

Implementation Details

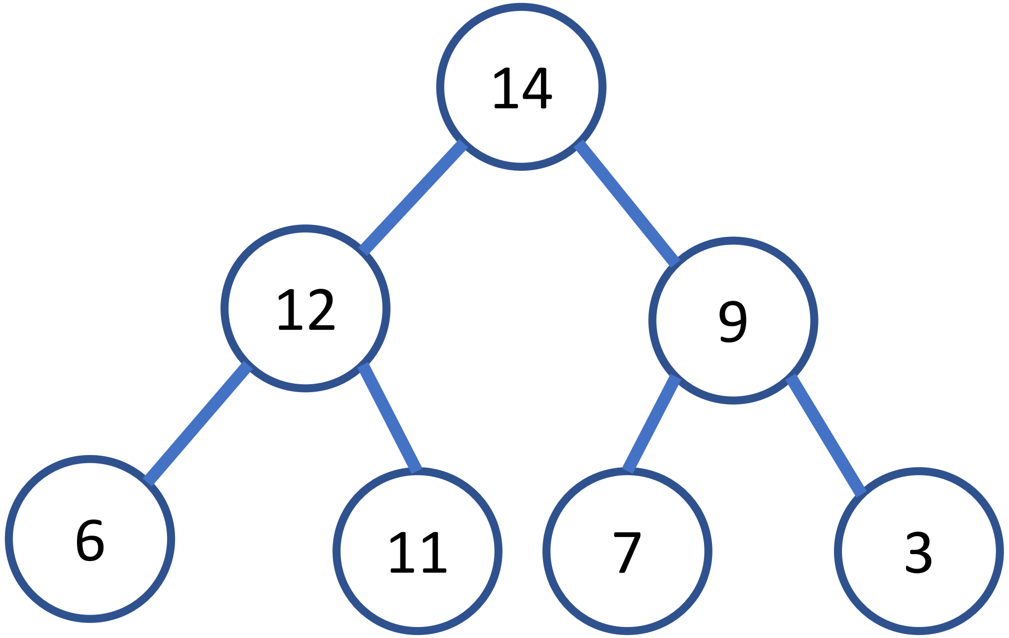
Max Heap



0	1	1	2	2	2	2	
14	**12**	**9**	**6**	**11**	**7**	**3**	
0	1	2	3	4	5	6	

Implementation Details

Max Heap



0	1	1	2	2	2	2	
14	12	9	6	11	7	3	
0	1	2	3	4	5	6	

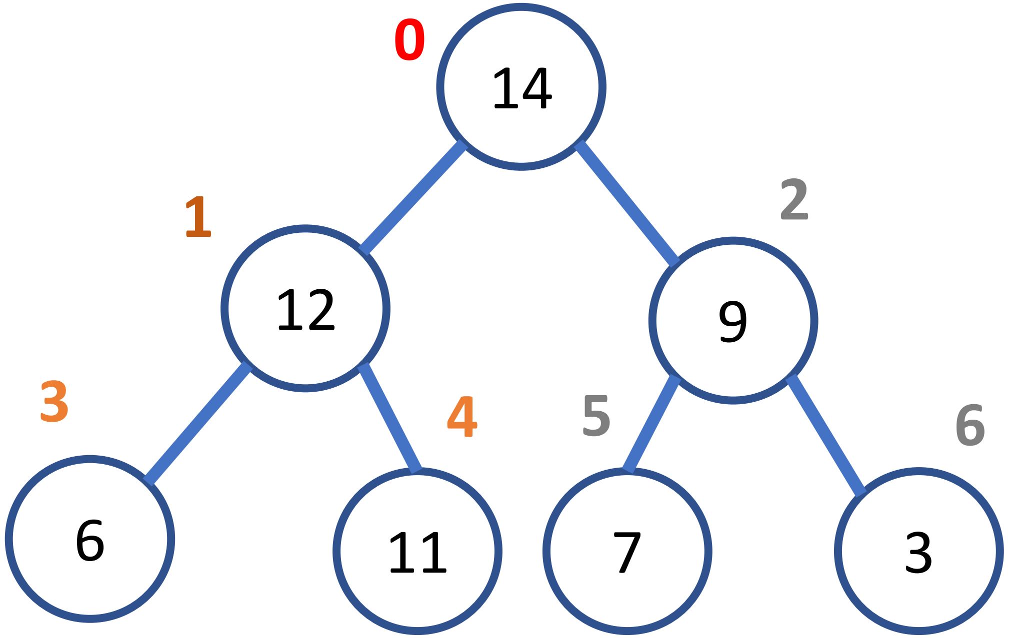
Parent

Node

Children

Implementation Details

Max Heap



0	1	1	2	2	2	2	
14	**12**	**9**	**6**	**11**	**7**	**3**	
0	1	2	3	4	5	6	

For each node at index i,

Index of left child =

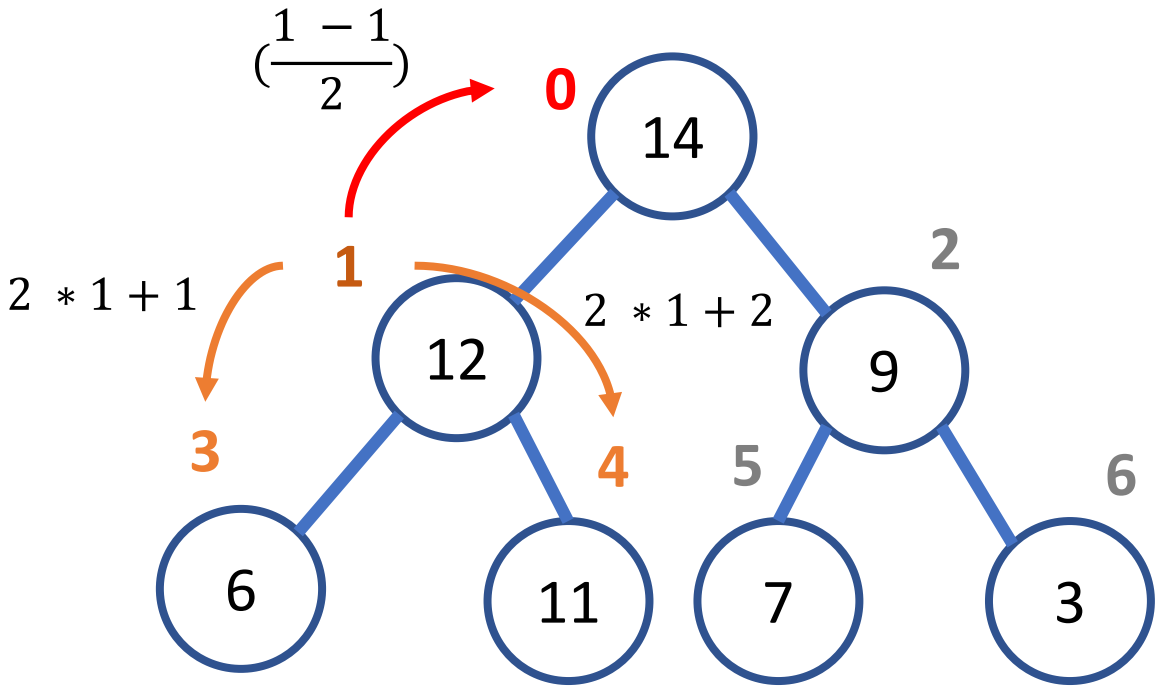
Index of right child =

Index of parent =

Implementation Details

0	1	1	2	2	2	2	
14	12	9	6	11	7	3	
0	1	2	3	4	5	6	

Max Heap



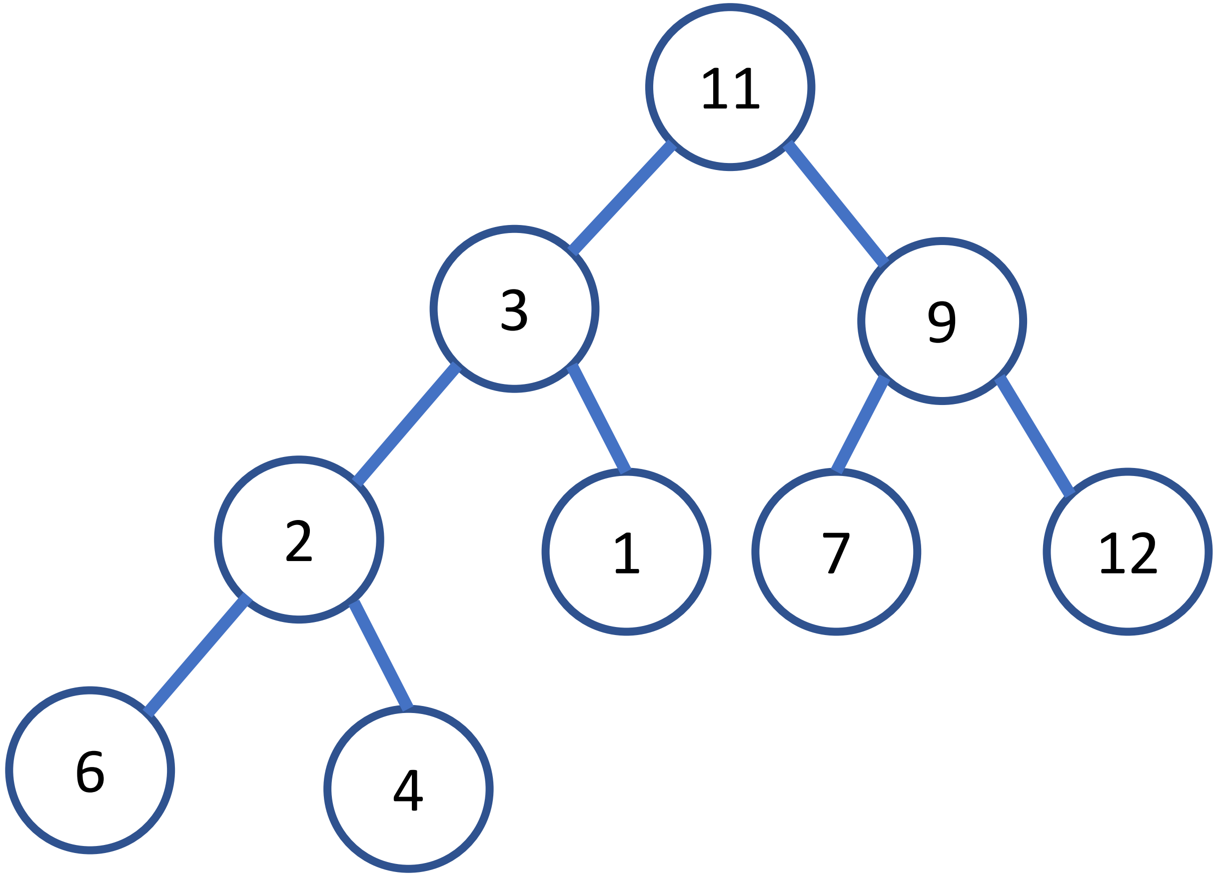
For each node at index i,

Index of left child $= 2i + 1$
Index of right child $= 2i + 2$

Index of parent $= (\frac{i-1}{2})$

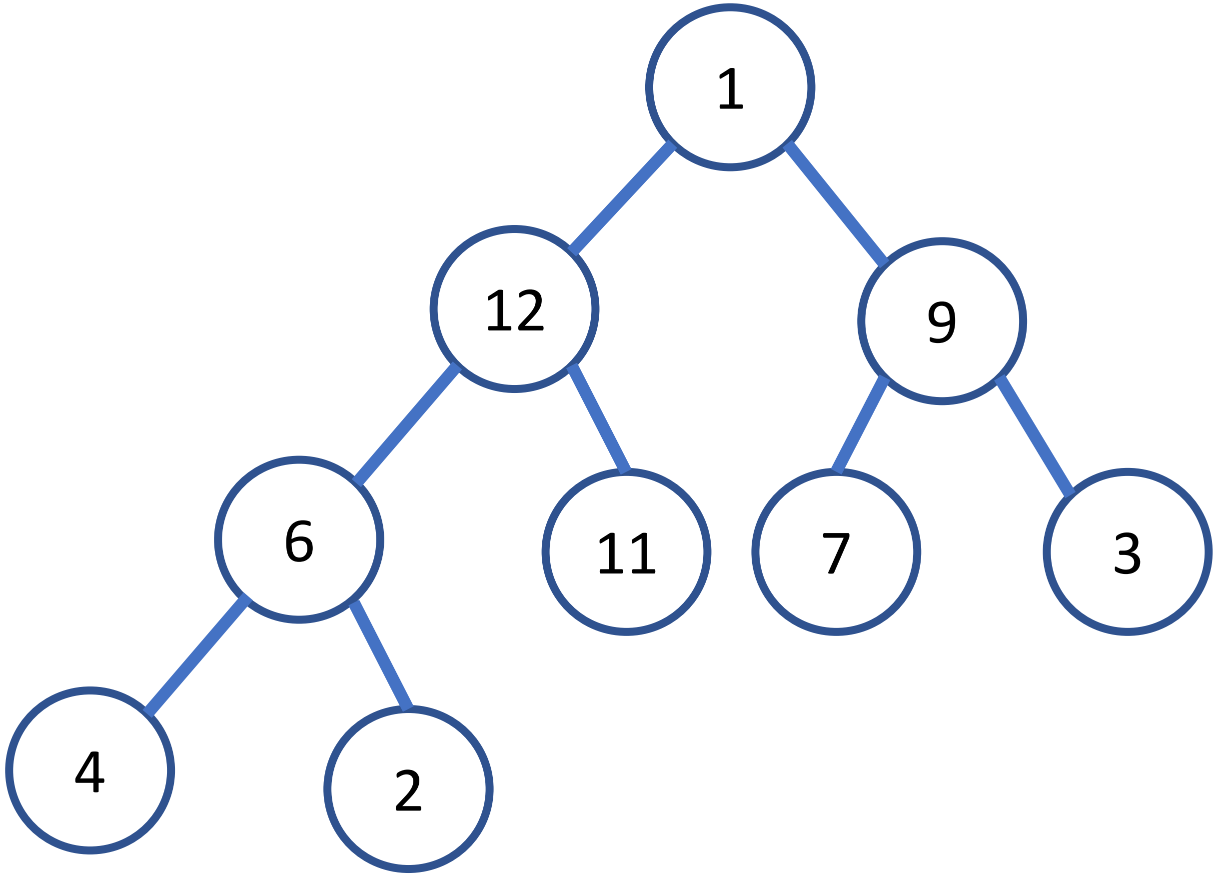
Max-Heapifying your data

Convert the below Random Tree/Array into a Max-Heap



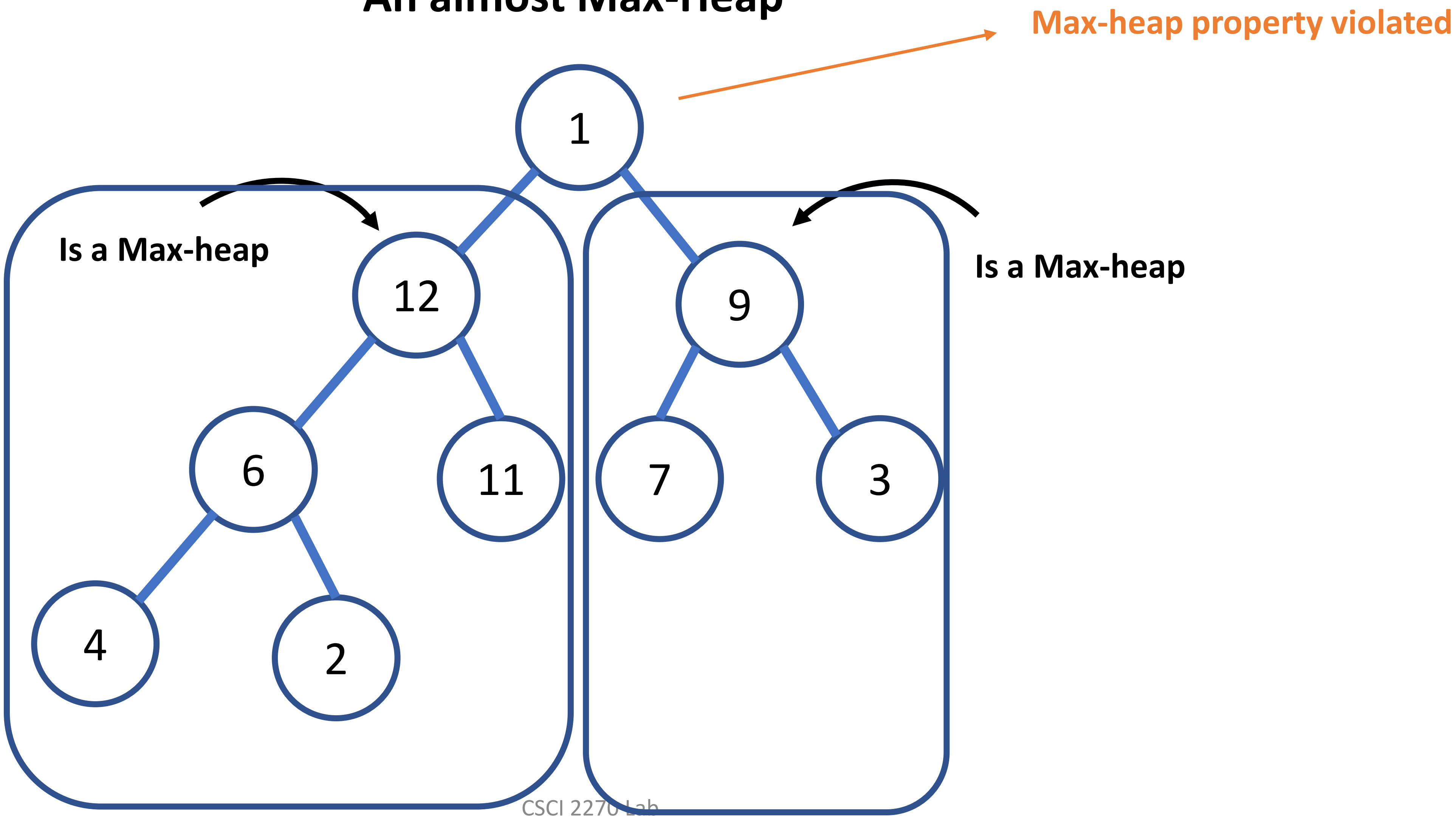
Max-Heapifying your data

Let’s consider a simpler example:
An almost Max-Heap



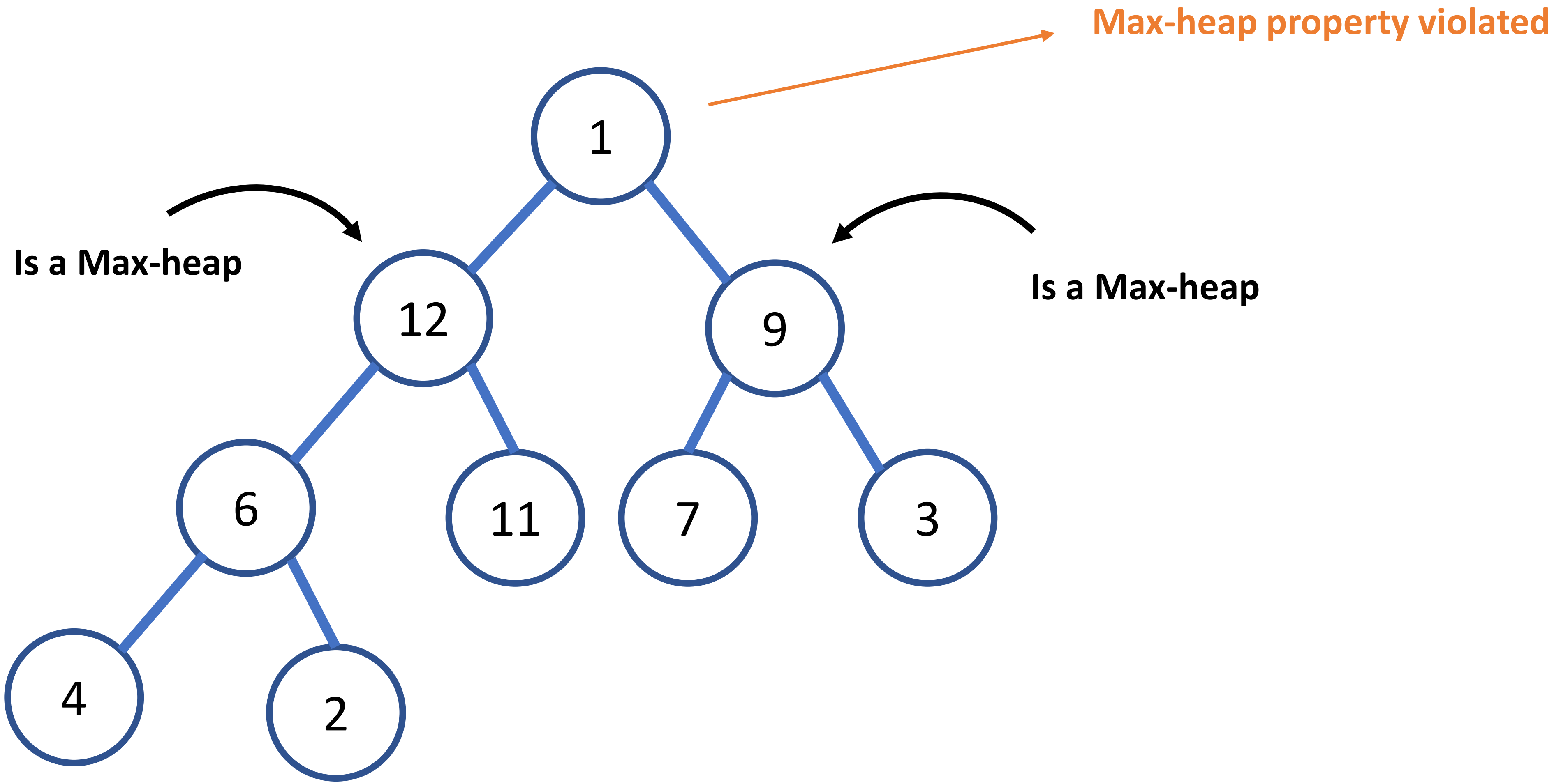
Max-Heapifying an almost Max-heap

Let's consider a simpler example:
An almost Max-Heap



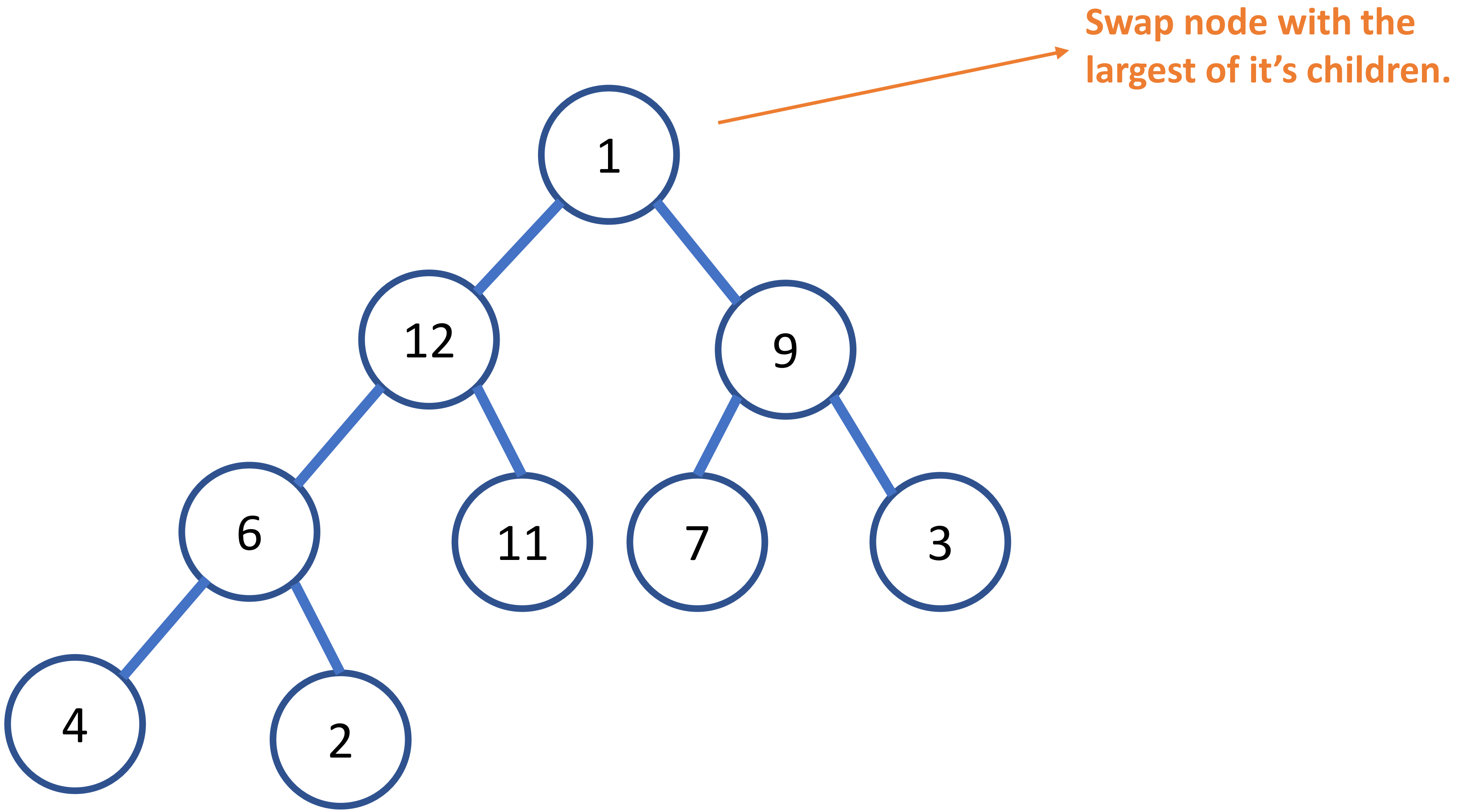
Max-Heapifying an almost Max-heap

Swap violating node with the largest of its children until max-heap property satisfied.



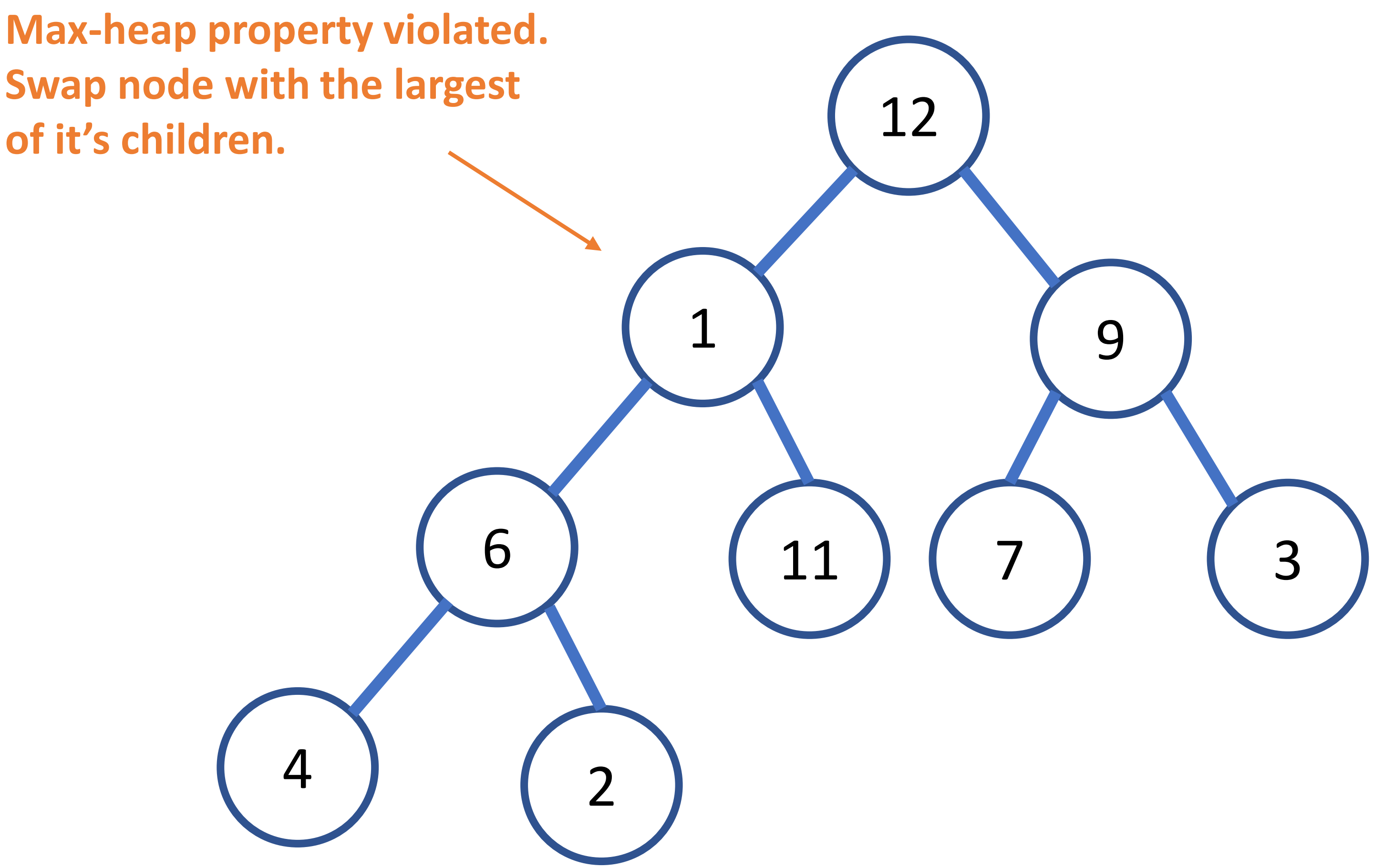
Max-Heapifying an almost Max-heap

Swap violating node with the largest of its children until max-heap property satisfied.



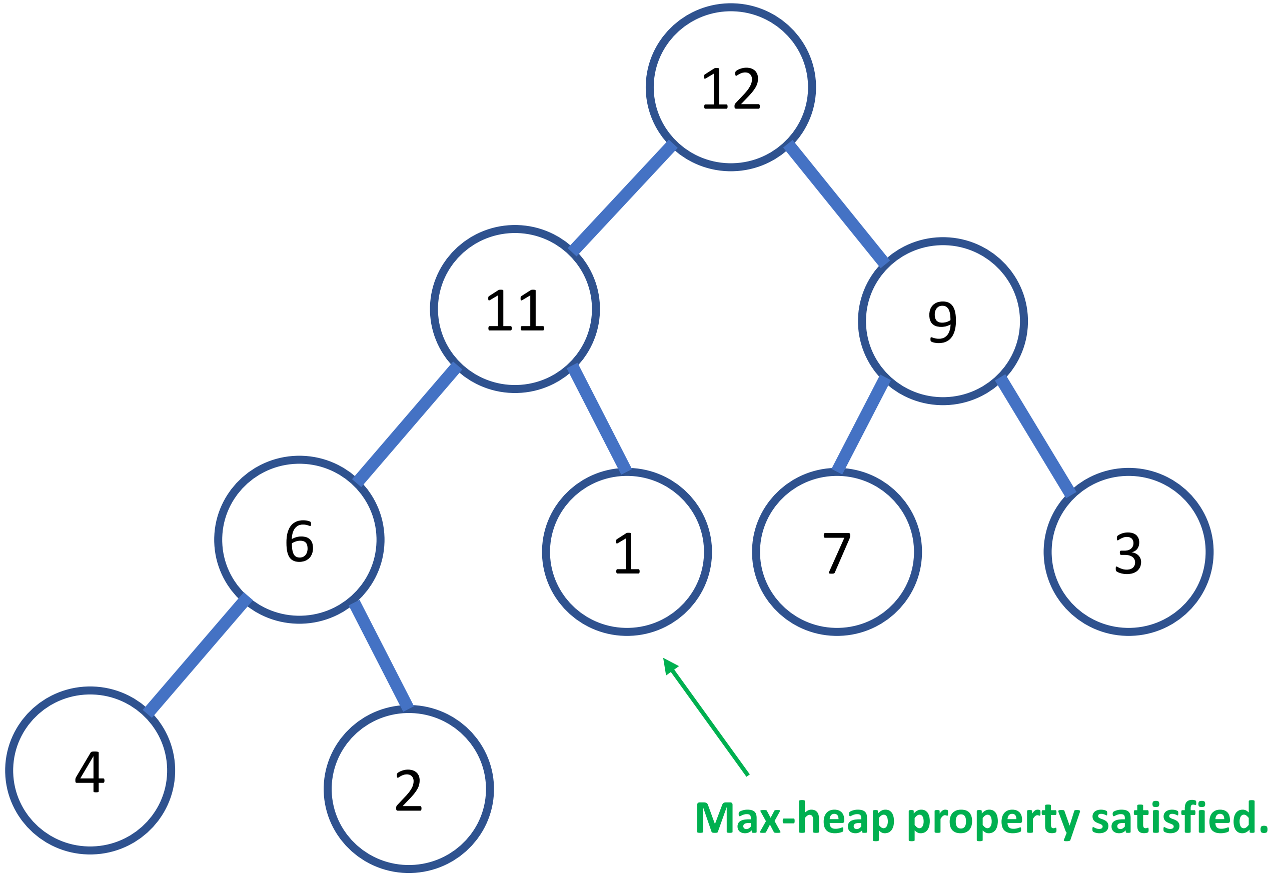
Max-Heapifying an almost Max-heap

Swap violating node with the largest of its children until max-heap property satisfied.



Max-Heapifying an almost Max-heap

Swap violating node with the largest of its children until max-heap property satisfied.



Max-Heapifying an almost Max-heap

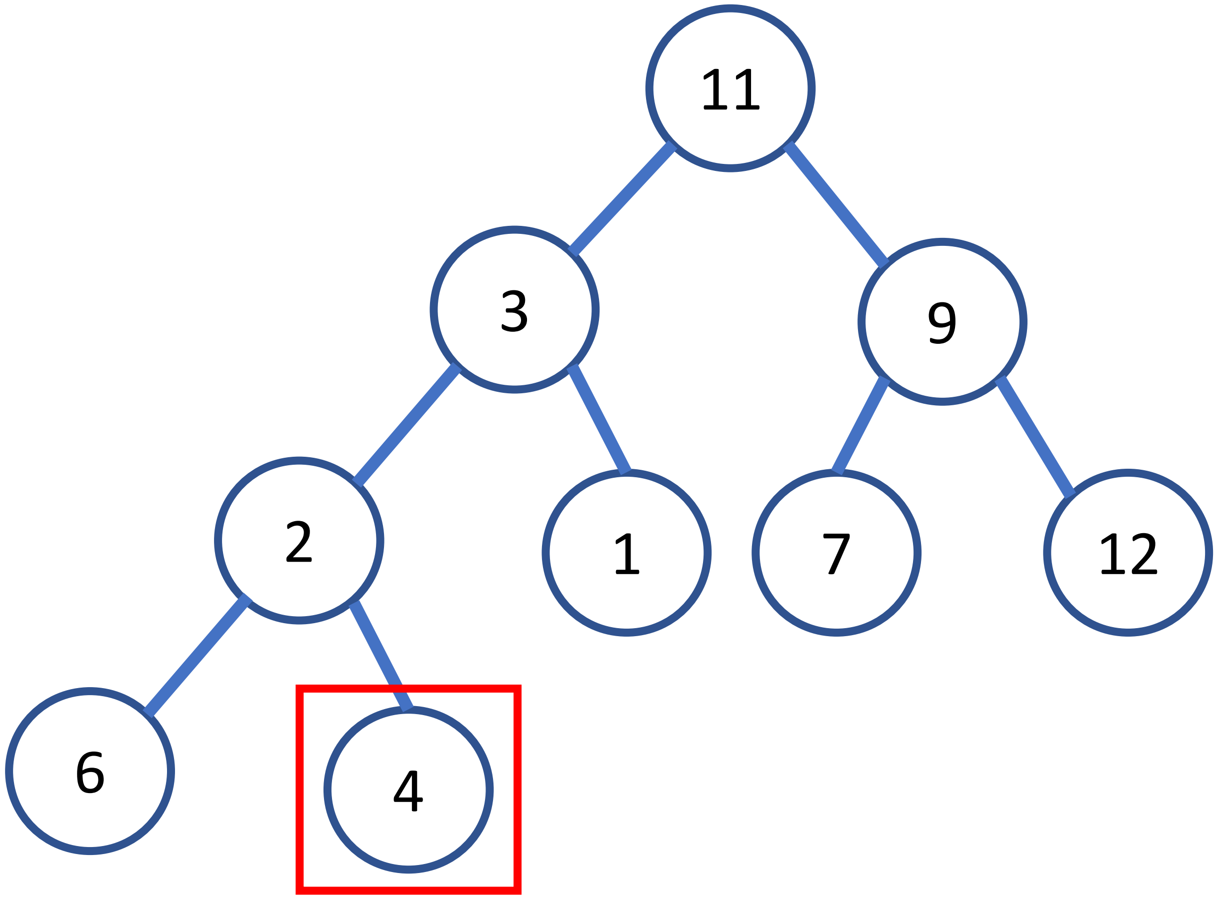
Swap violating node with the largest of its children until max-heap property satisfied.

```
 void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i + 1; //left child
    int right = 2*i + 2; //right child

    if(left< N && arr[left] > Arr[i] )
          largest = left;
    else
         largest = i;
    if(right < N && arr[right] > arr[largest] )
        largest = right;

    if(largest != i)
    {
        swap (&Ar[i], &Arr[largest]);
        max_heapify (Arr, largest,N);
    }
 }
```

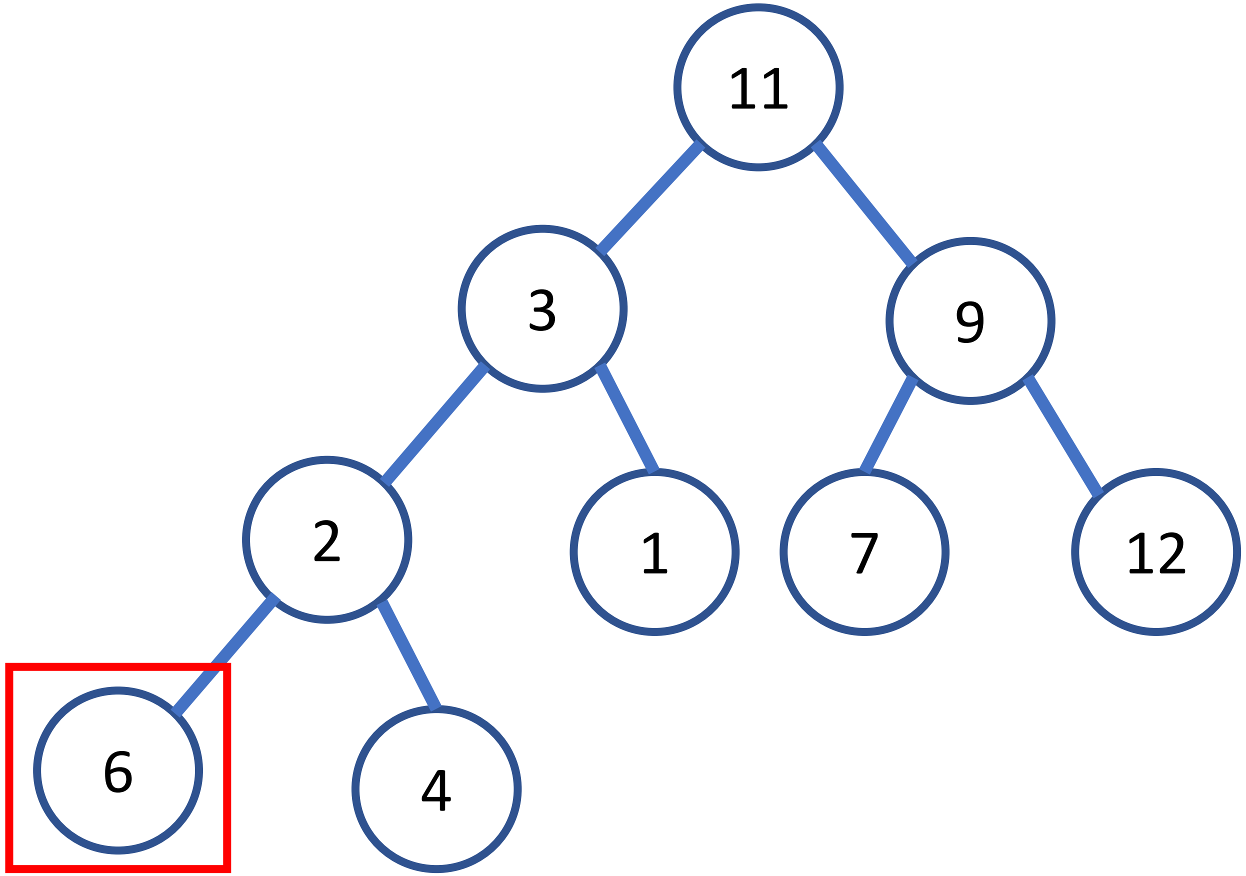
Max-Heapifying a random array




```
max_heapify(Arr, 8, 9);
```

- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

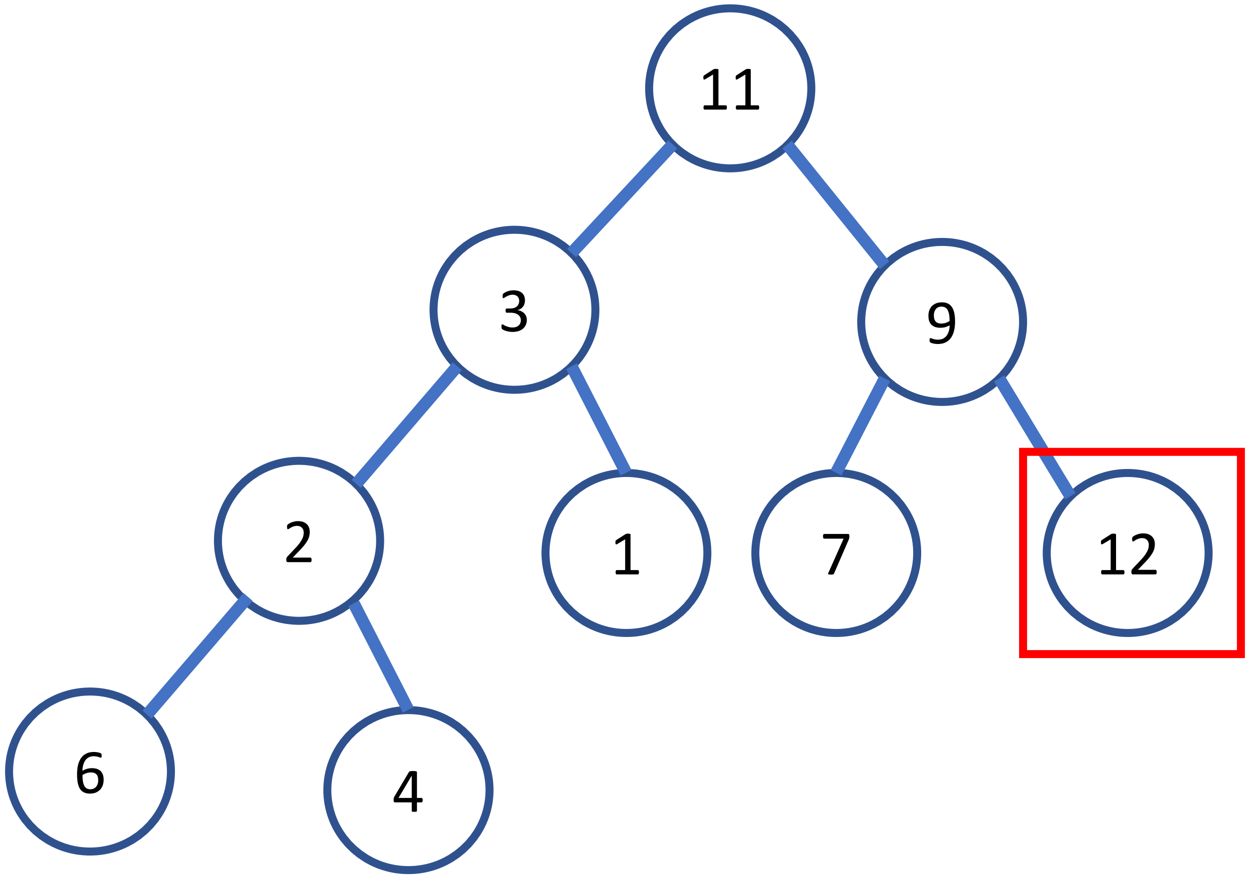
Max-Heapifying a random array



- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

`max_heapify(Arr, 7, 9);`

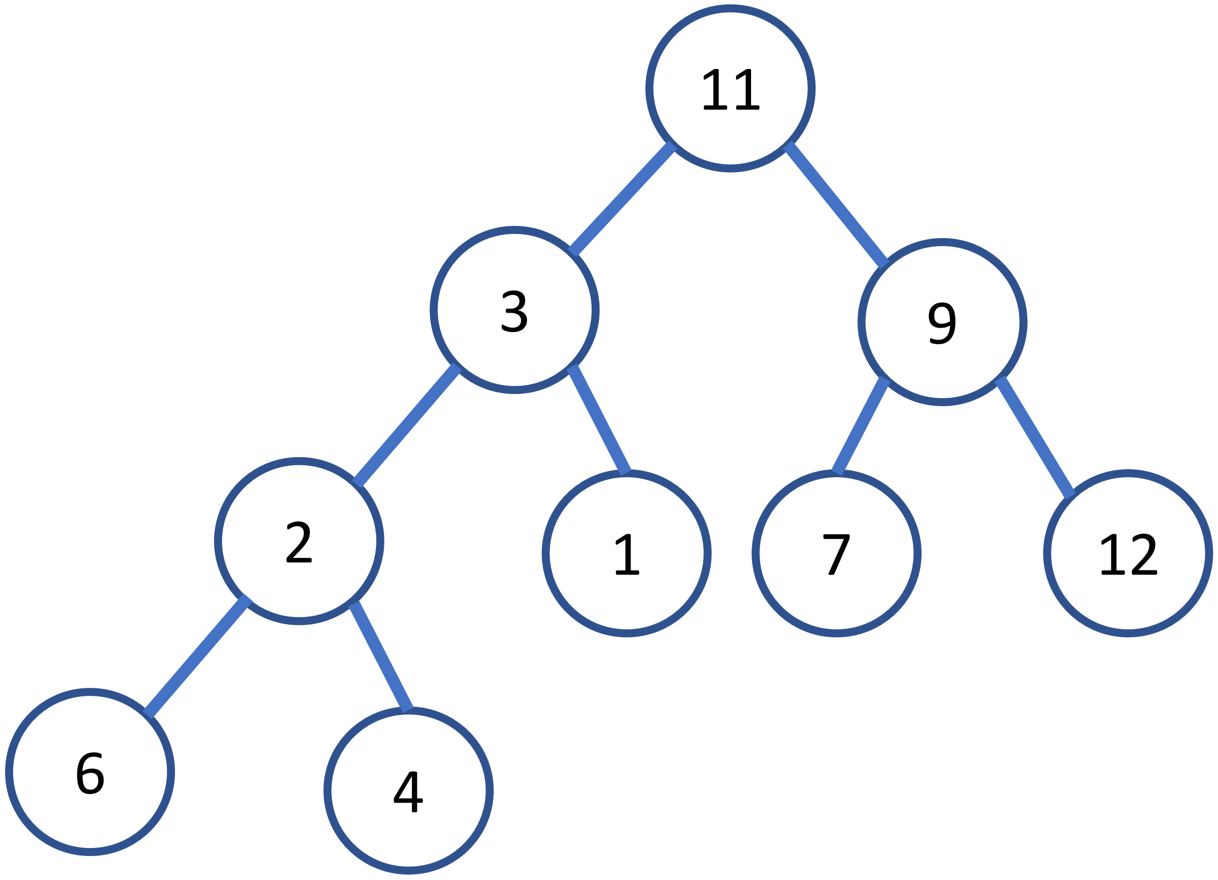
Max-Heapifying a random array



`max_heapify(Arr, 6, 9);`

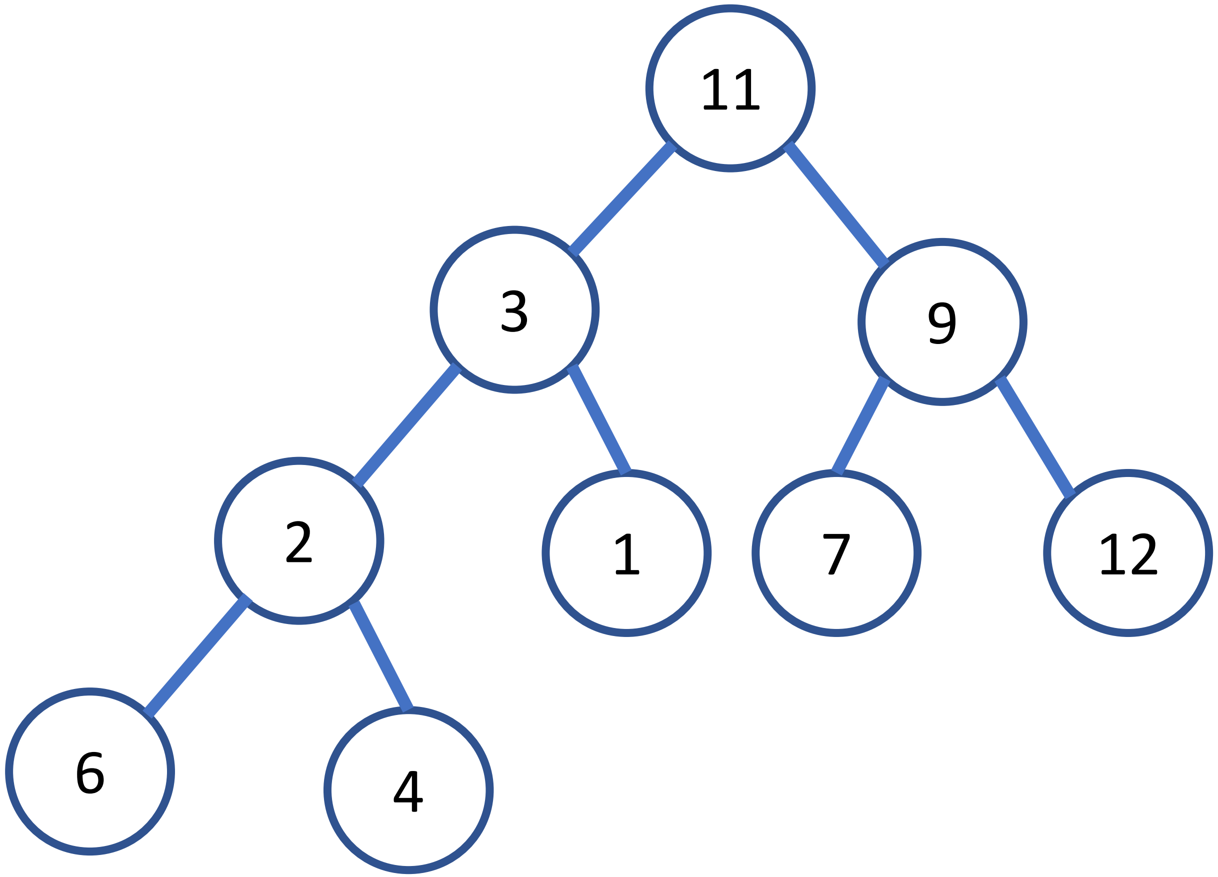
- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

Max-Heapifying a random array



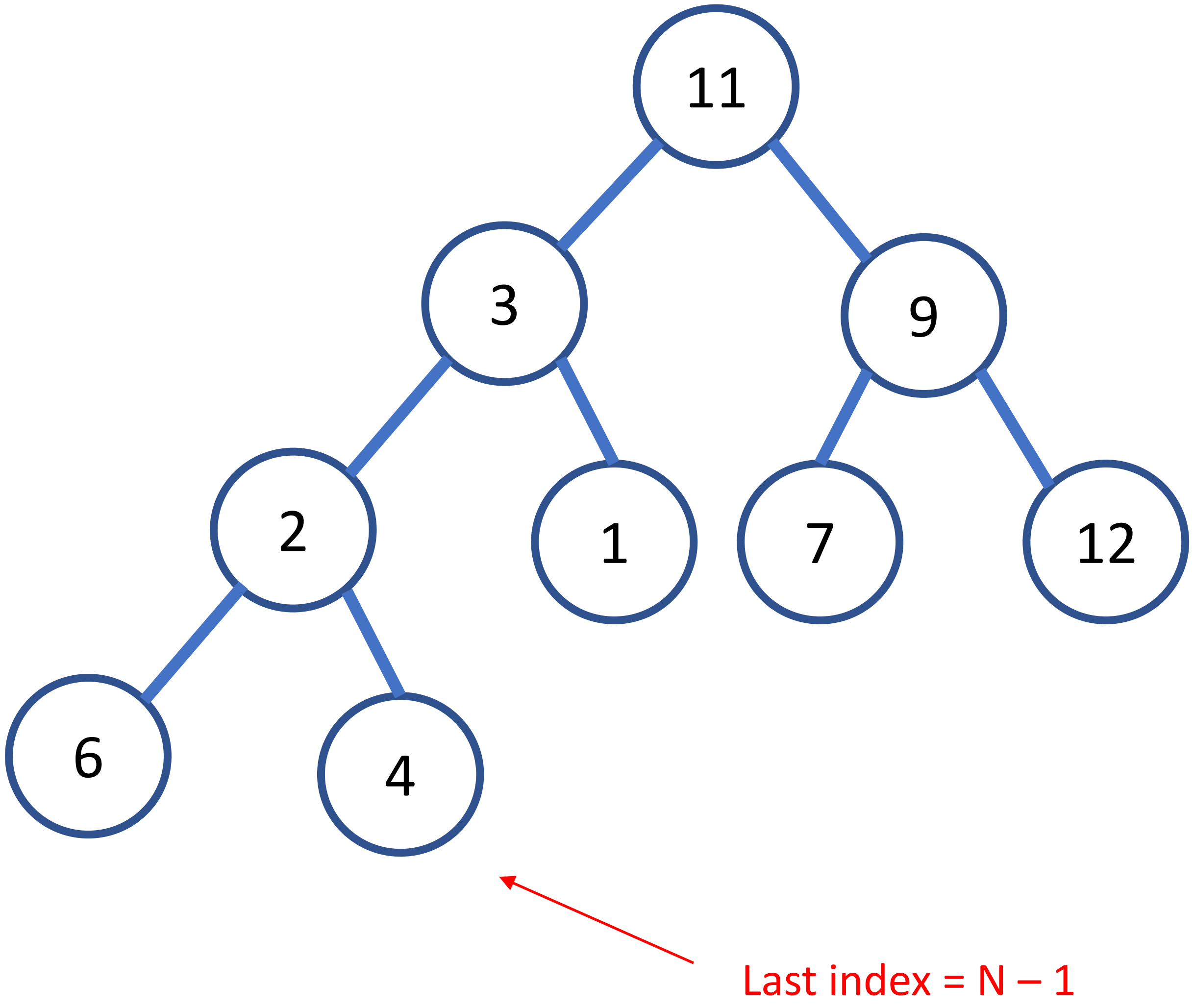
- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.
- Since the last few nodes are leaves, they represent a heap of 1 node, and are already max-heapified.

Max-Heapifying a random array



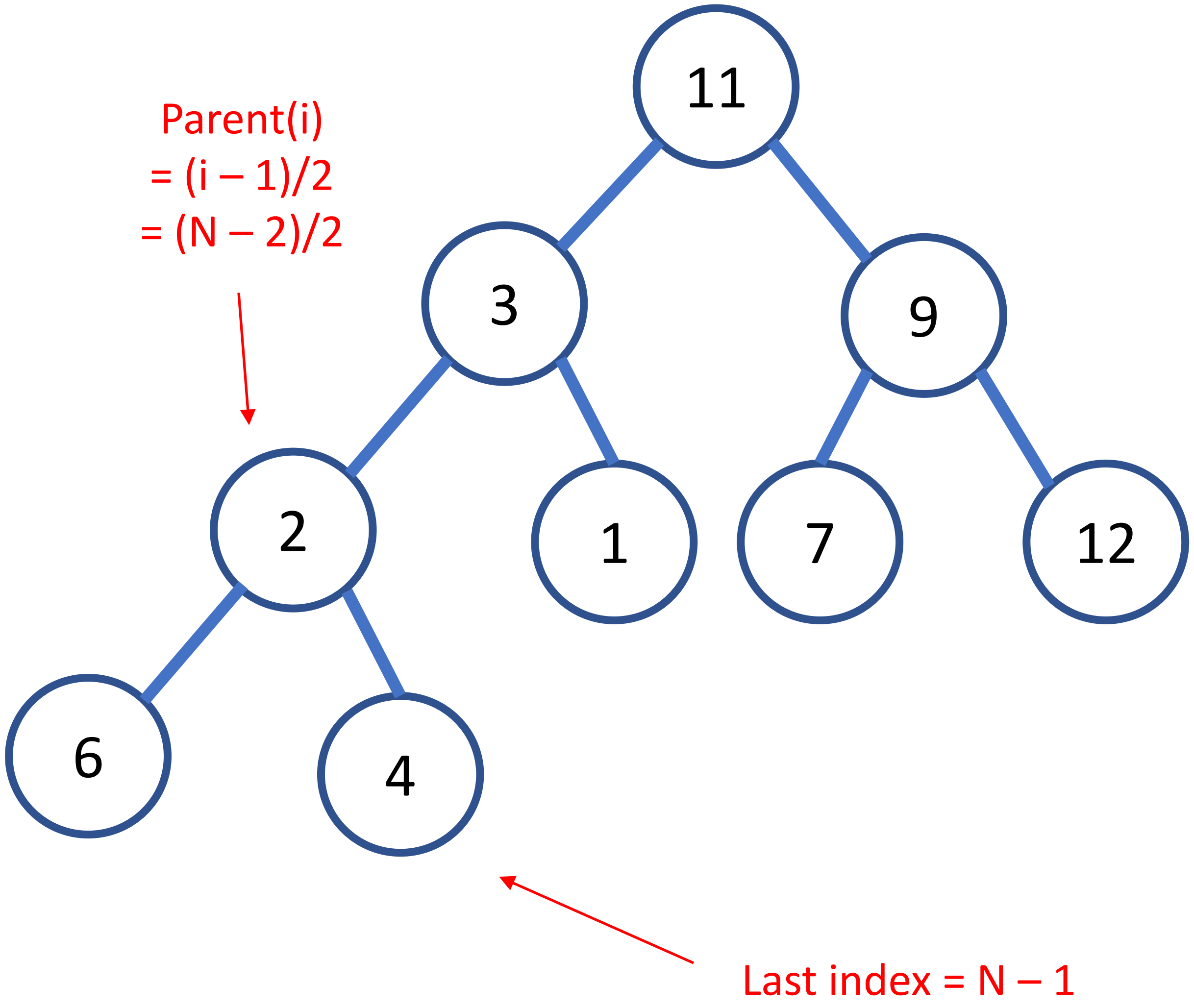
- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.
- Since the last few nodes are leaves, they represent a heap of 1 node, and are already max-heapified.
- We can skip these nodes and start with the highest index non-leaf node.
- What is the index of this node?

Max-Heapifying a random array



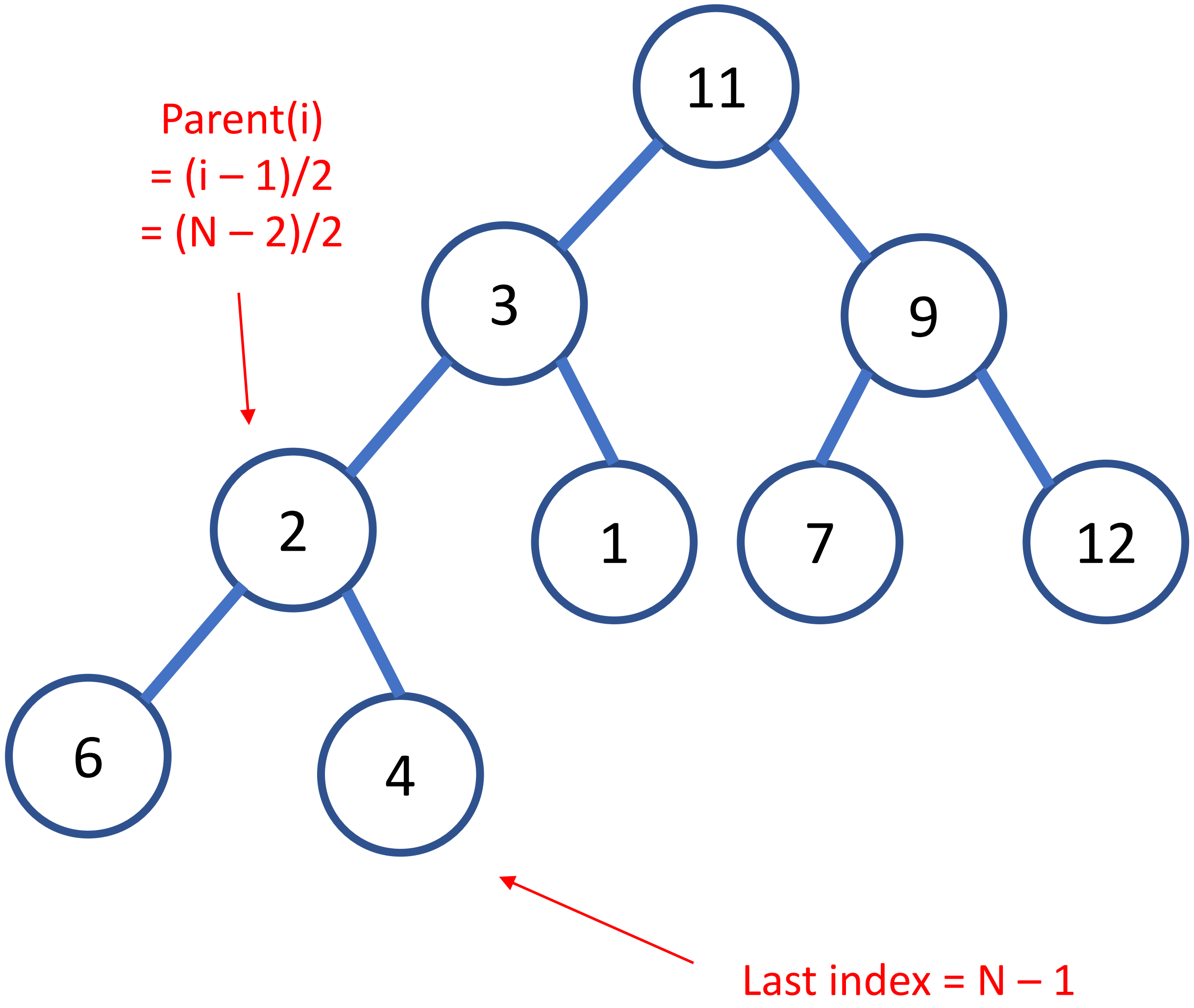
- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.
- Since the last few nodes are leaves, they represent a heap of 1 node, and are already max-heapified.
- We can skip these nodes and start with the highest index non-leaf node.
- What is the index of this node?

Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.
- Since the last few nodes are leaves, they represent a heap of 1 node, and are already max-heapified.
- We can skip these nodes and start with the highest index non-leaf node.
- What is the index of this node?

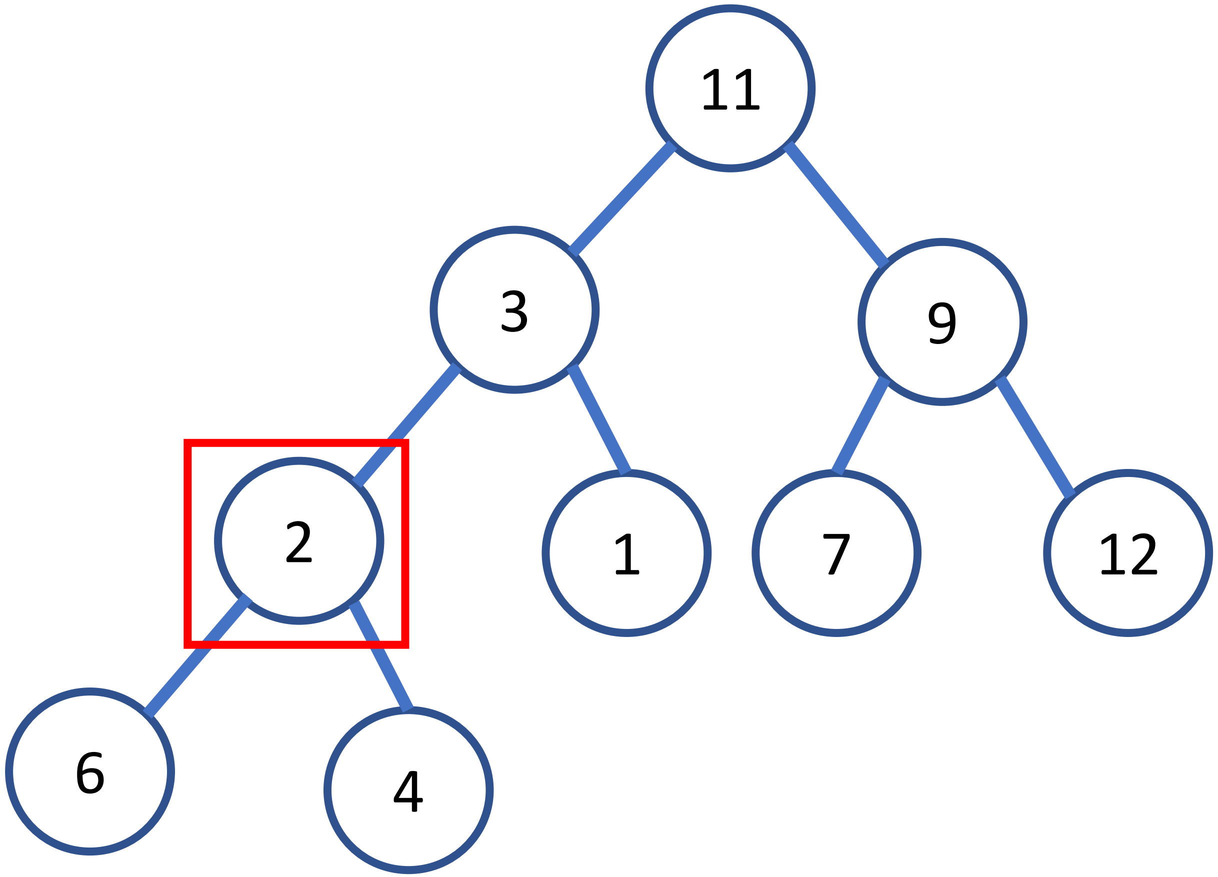
Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.
- Since the last few nodes are leaves, they represent a heap of 1 node, and are already max-heapified.
- We can skip these nodes and start with the highest index non-leaf node.
- What is the index of this node?

$$\frac{N}{2} - 1$$

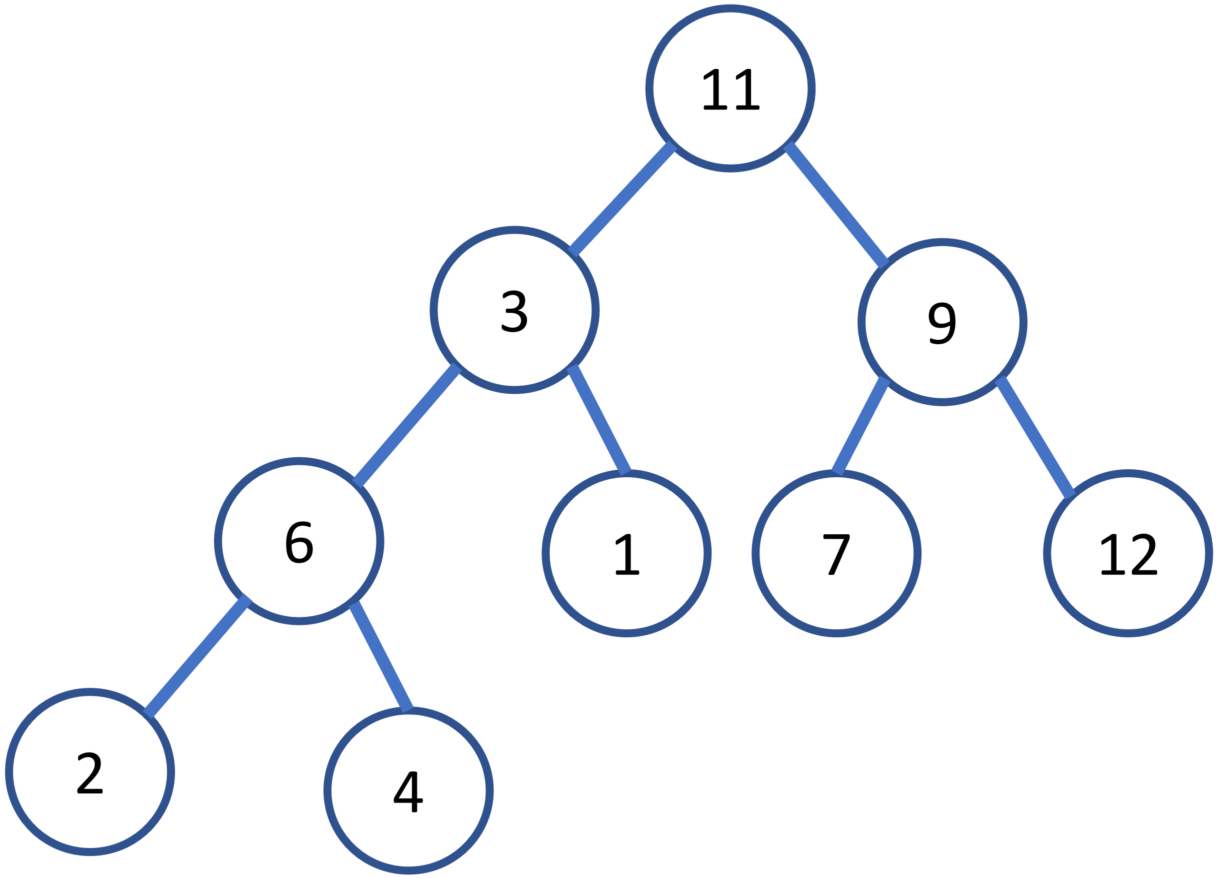
Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

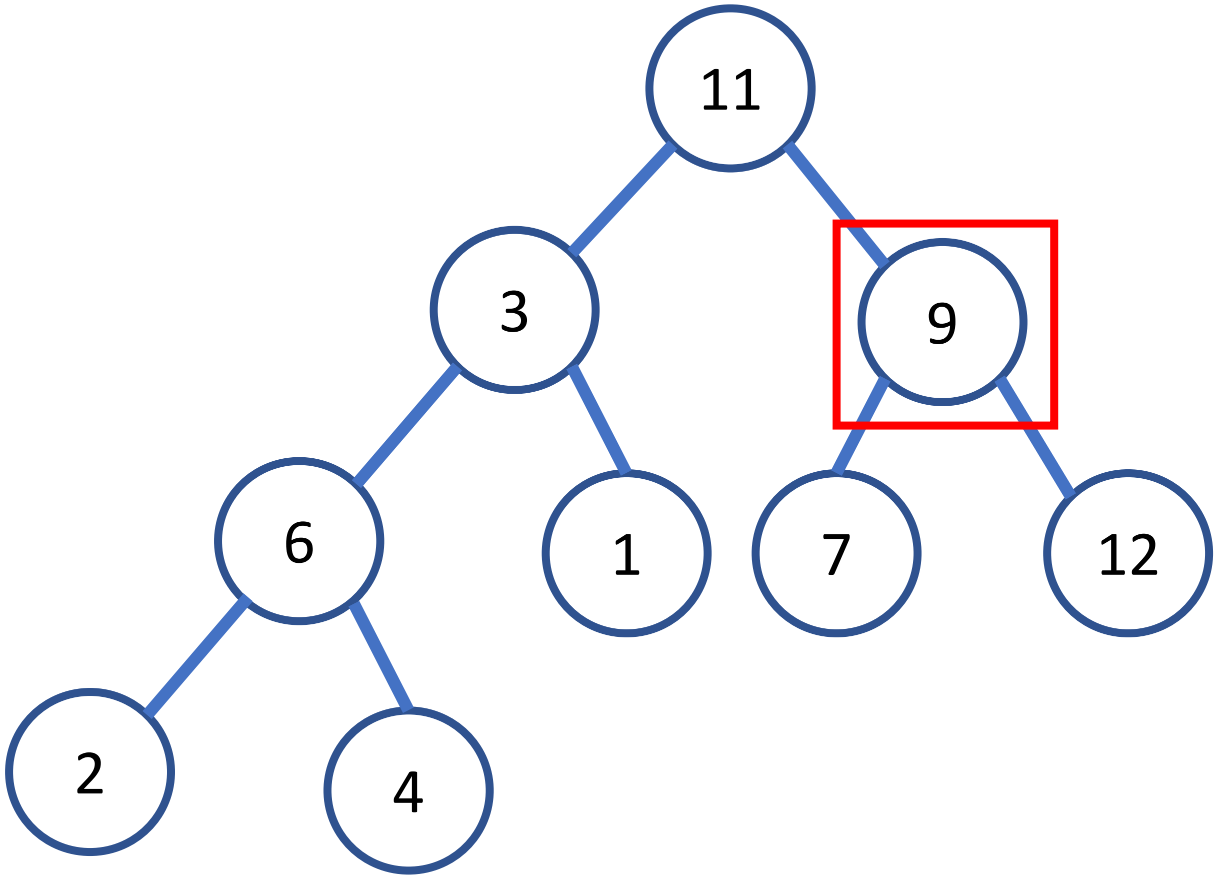
```
max_heapify(Arr, 3, 9);
```

Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

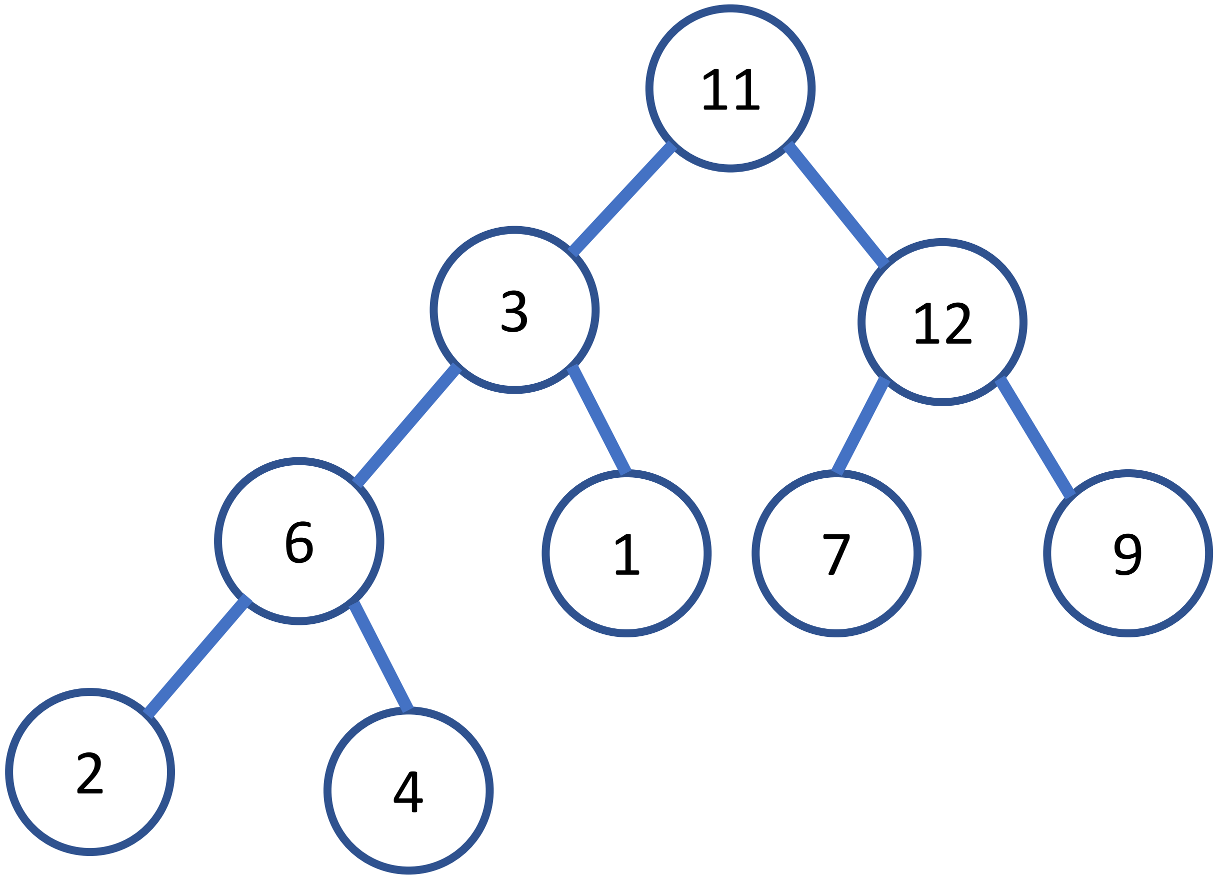
Max-Heapifying a random array



`max_heapify(Arr, 2, 9);`

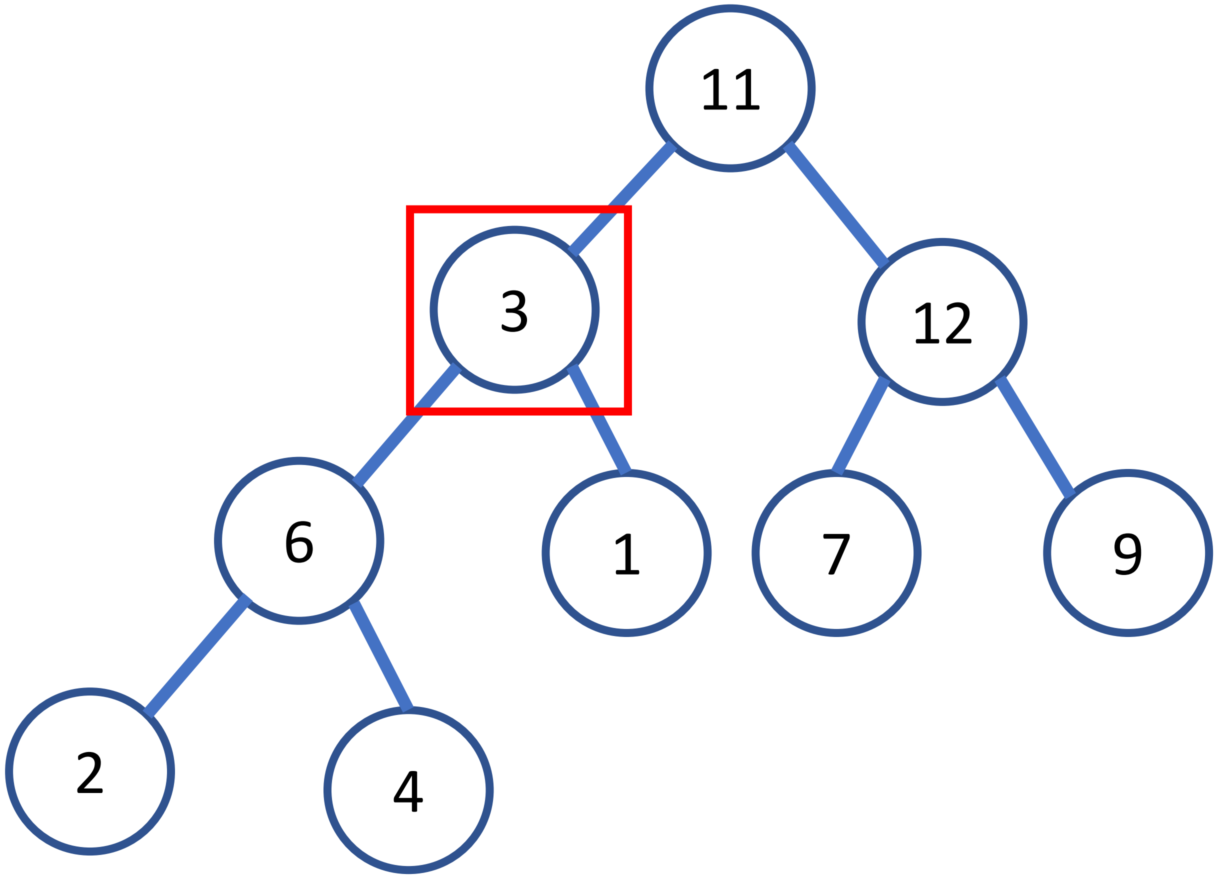
- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

Max-Heapifying a random array



- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

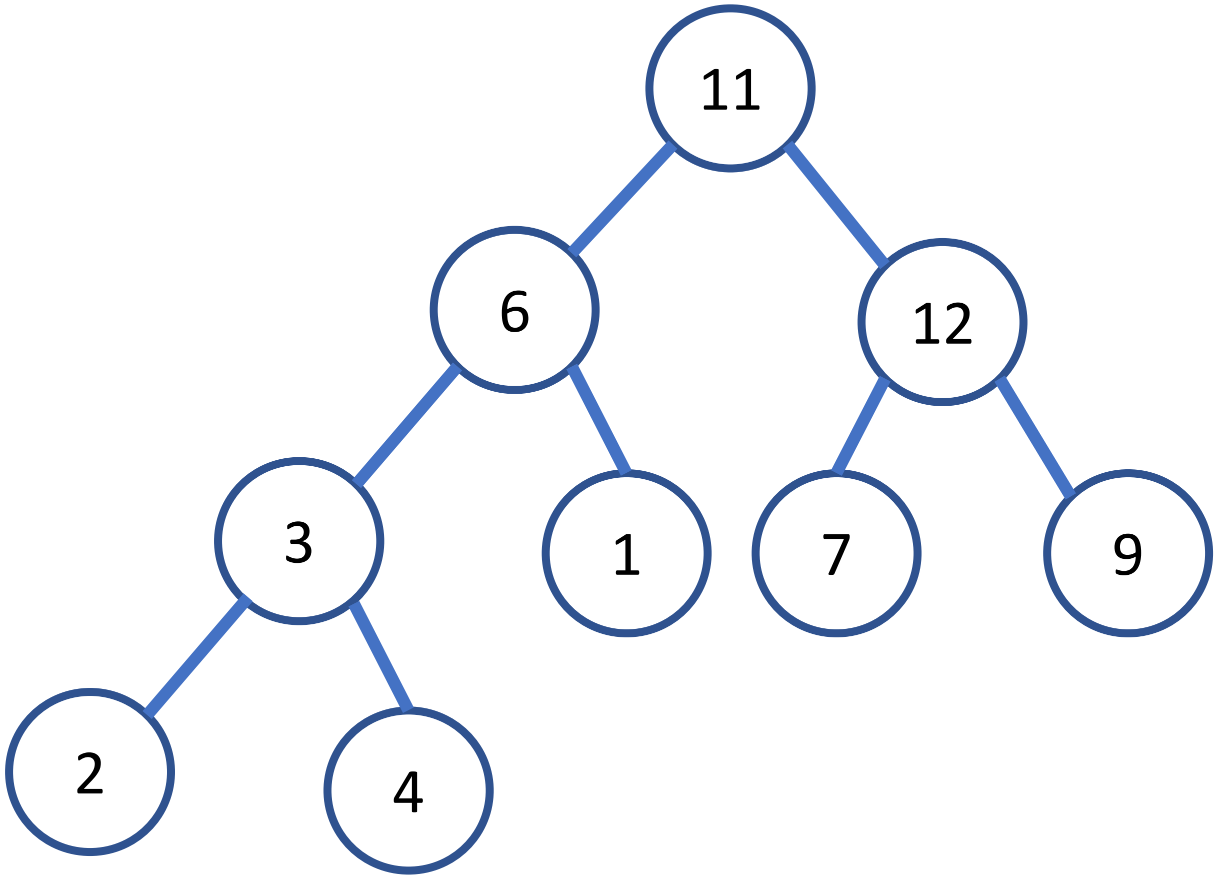
Max-Heapifying a random array




```
max_heapify(Arr, 1, 9);
```

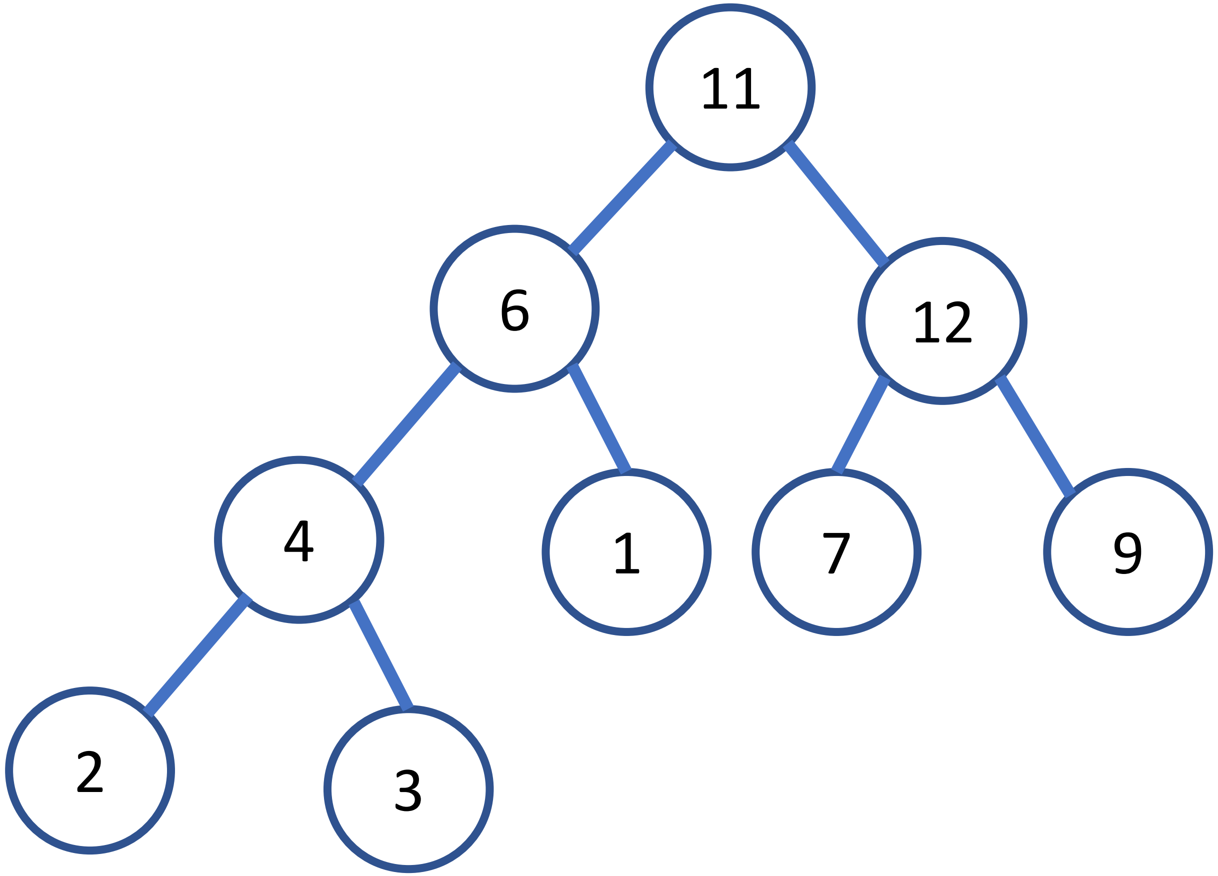
- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

Max-Heapifying a random array



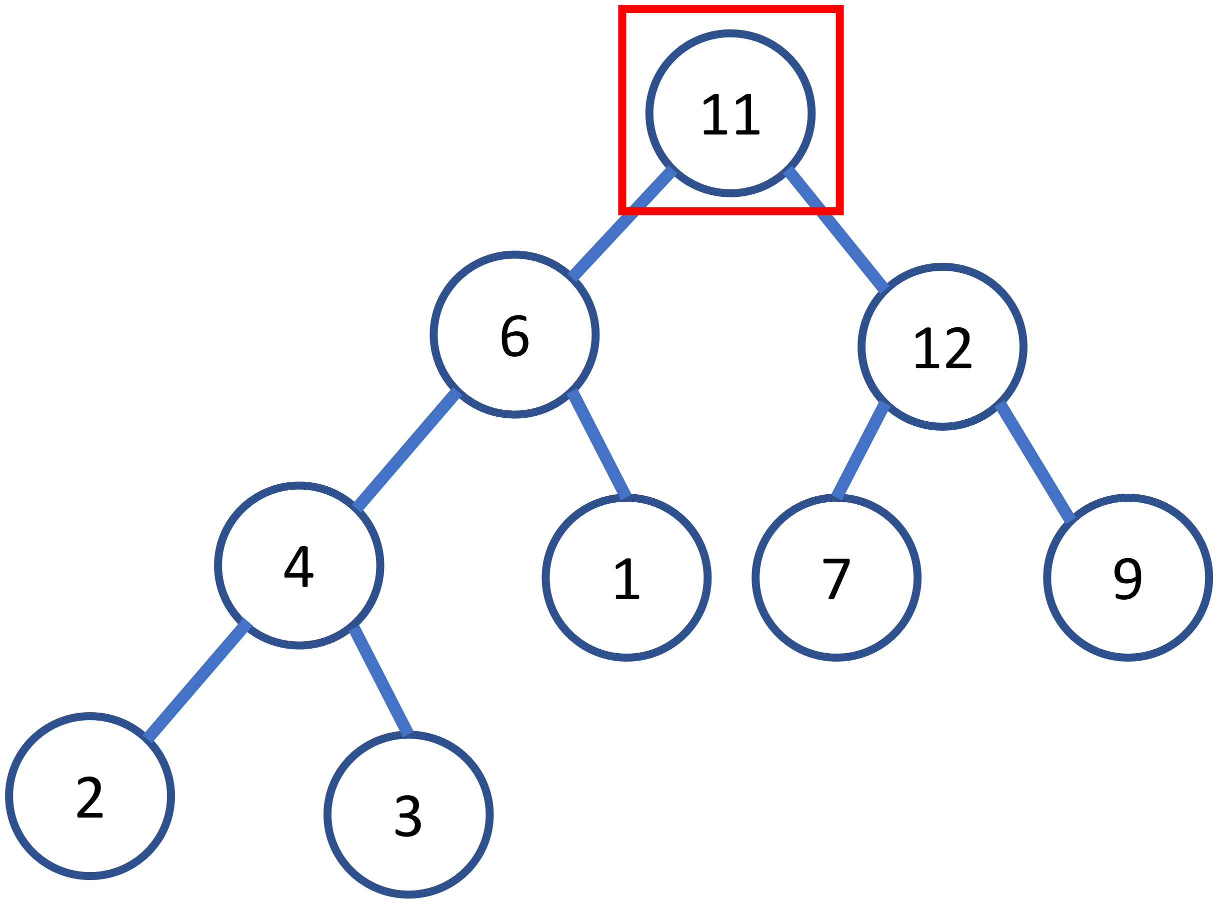
- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

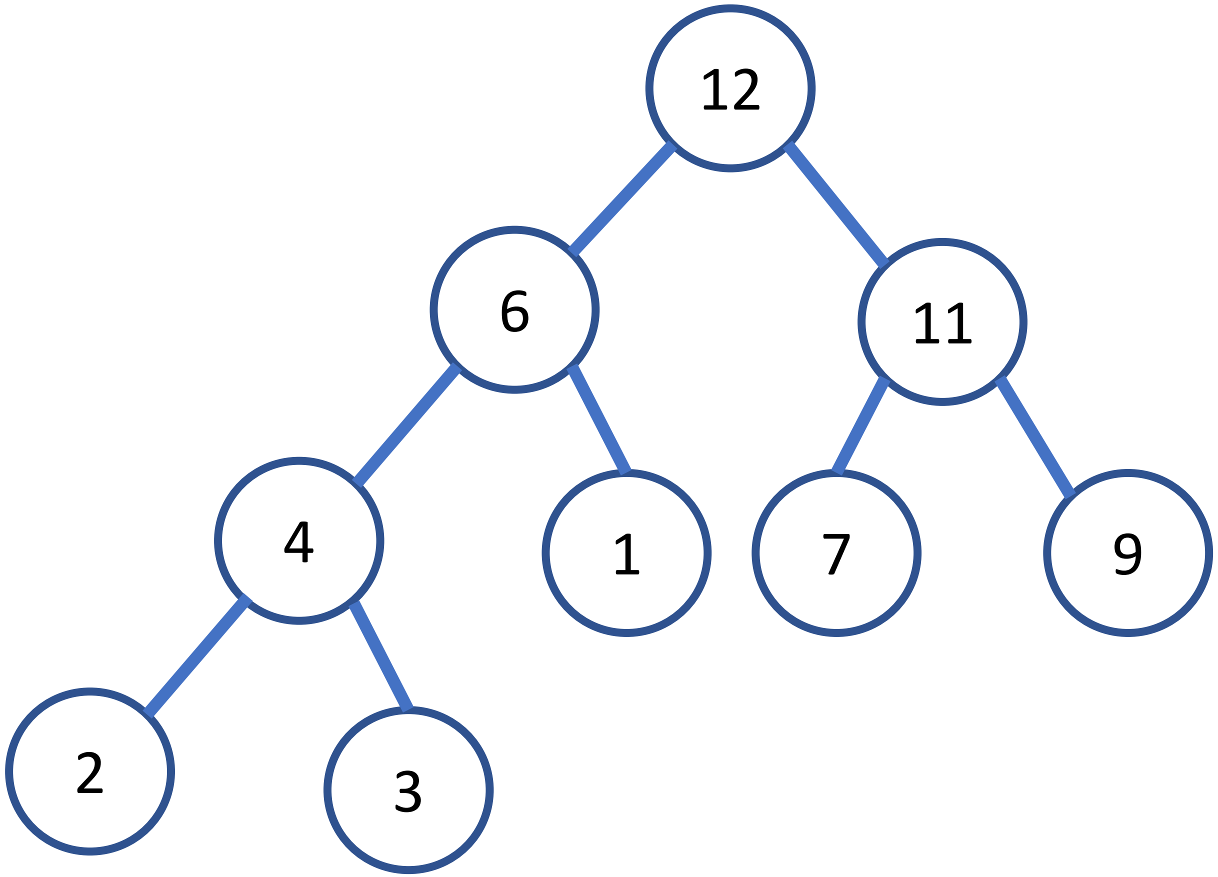
Max-Heapifying a random array



- Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.

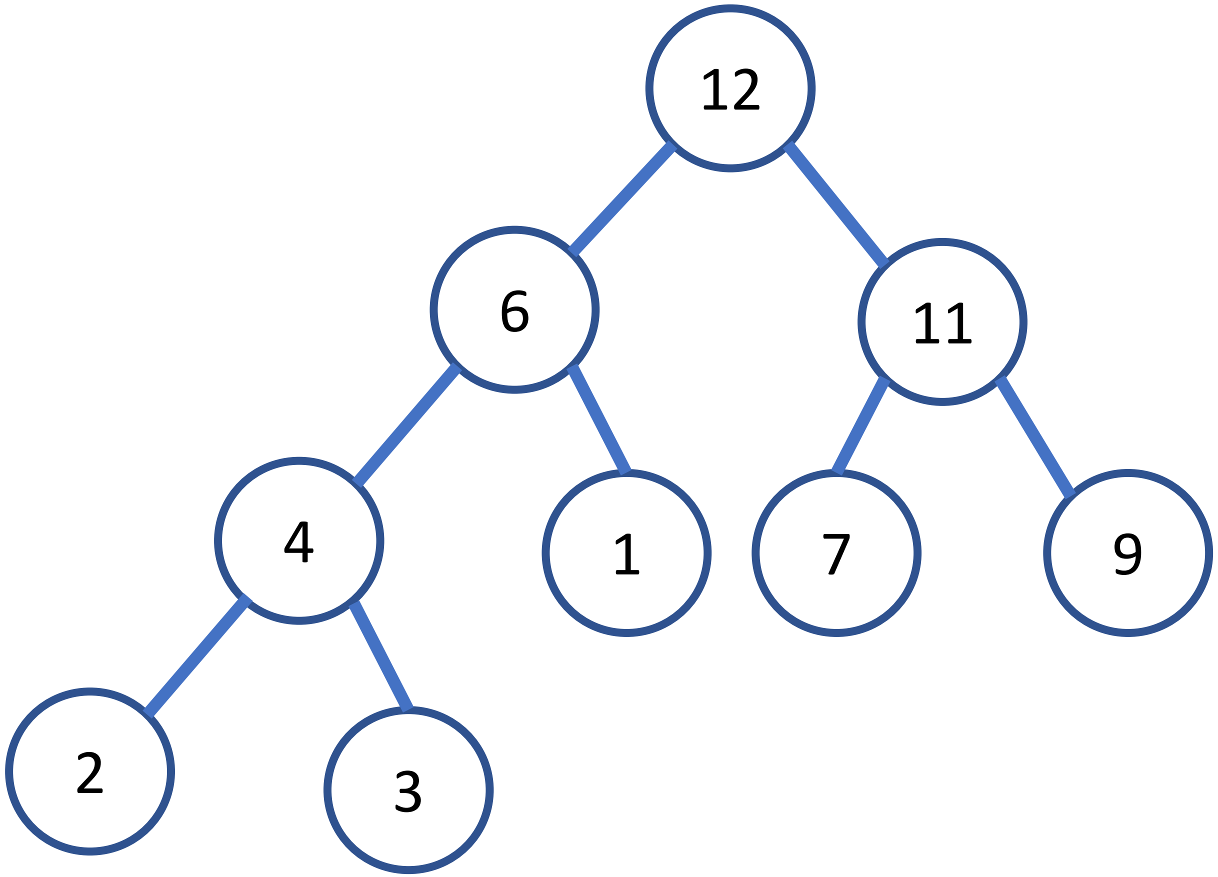
```
max_heapify(Arr, 0, 9);
```

Max-Heapifying a random array



- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

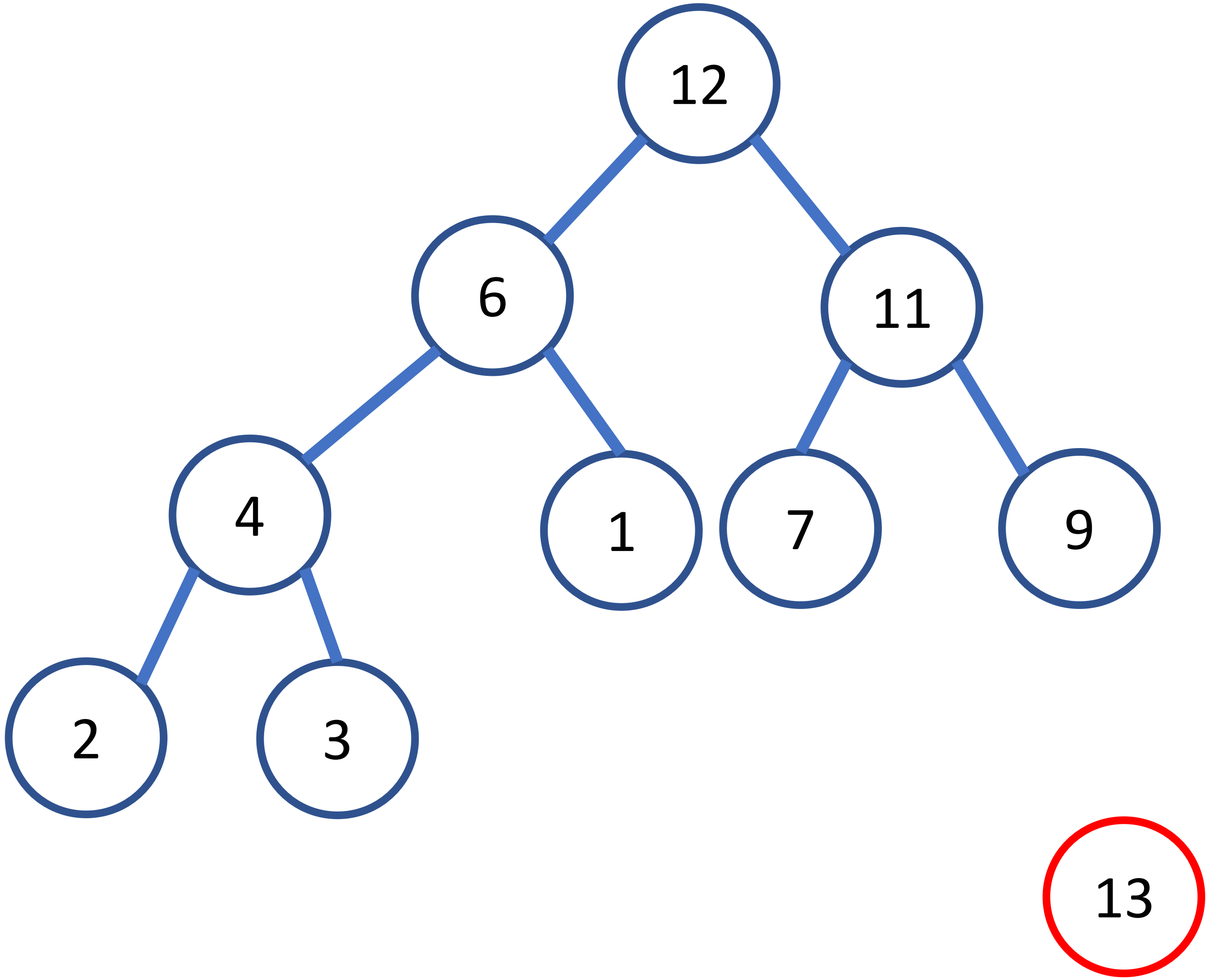
Max-Heapifying a random array



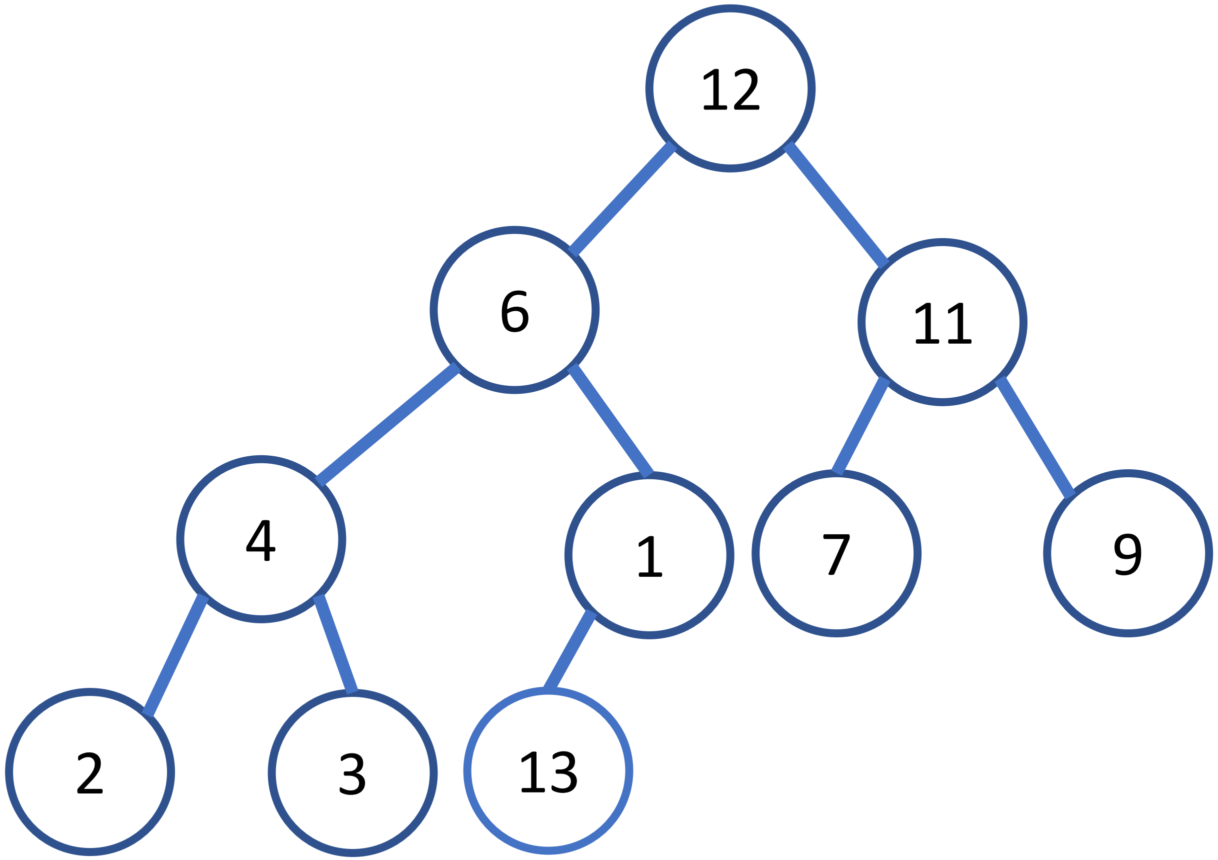
- `Starting with the highest indexed node and moving backwards, max-heapify the node into the subtree.`

```
for (int i =  N/2 – 1; i >= 0; i--)
{
      max_heapify(arr, i, N);
}
```

Inserting a new element into a max-heap

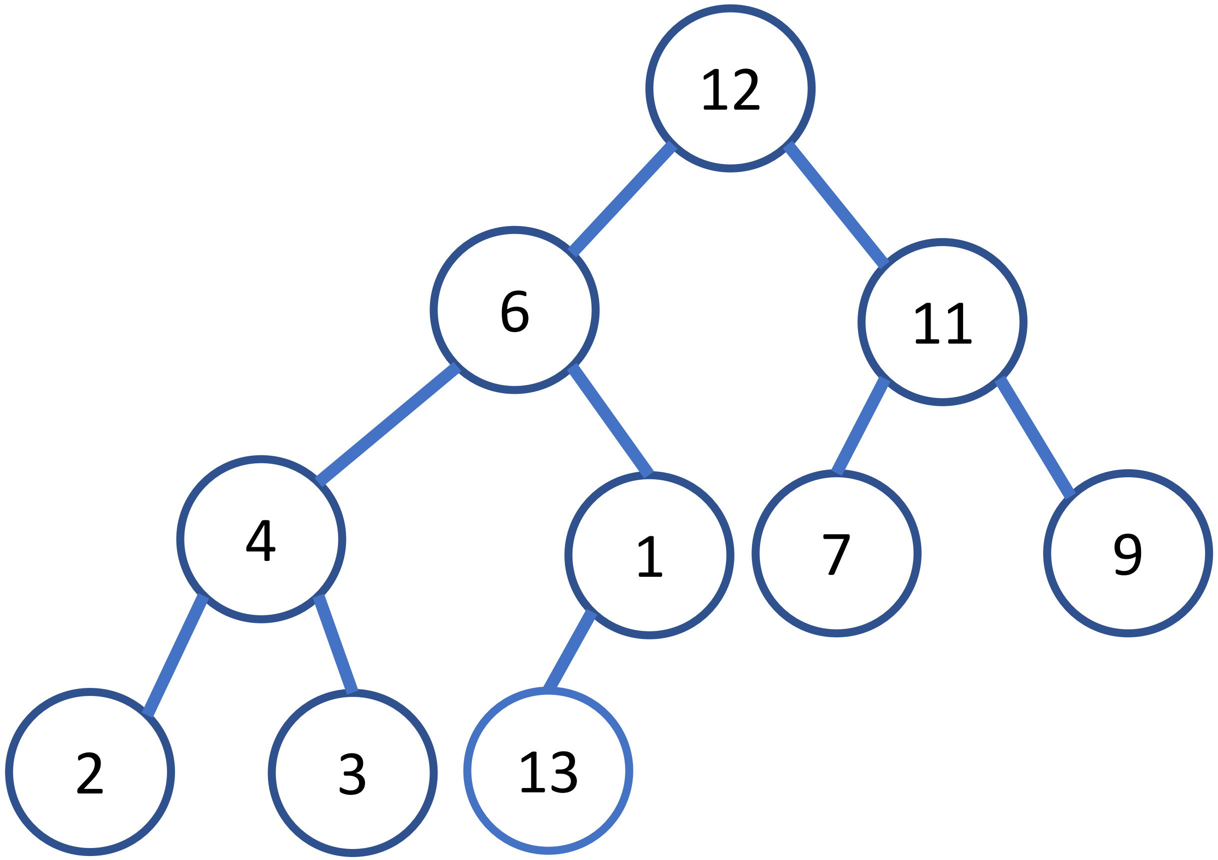


Inserting a new element into a max-heap



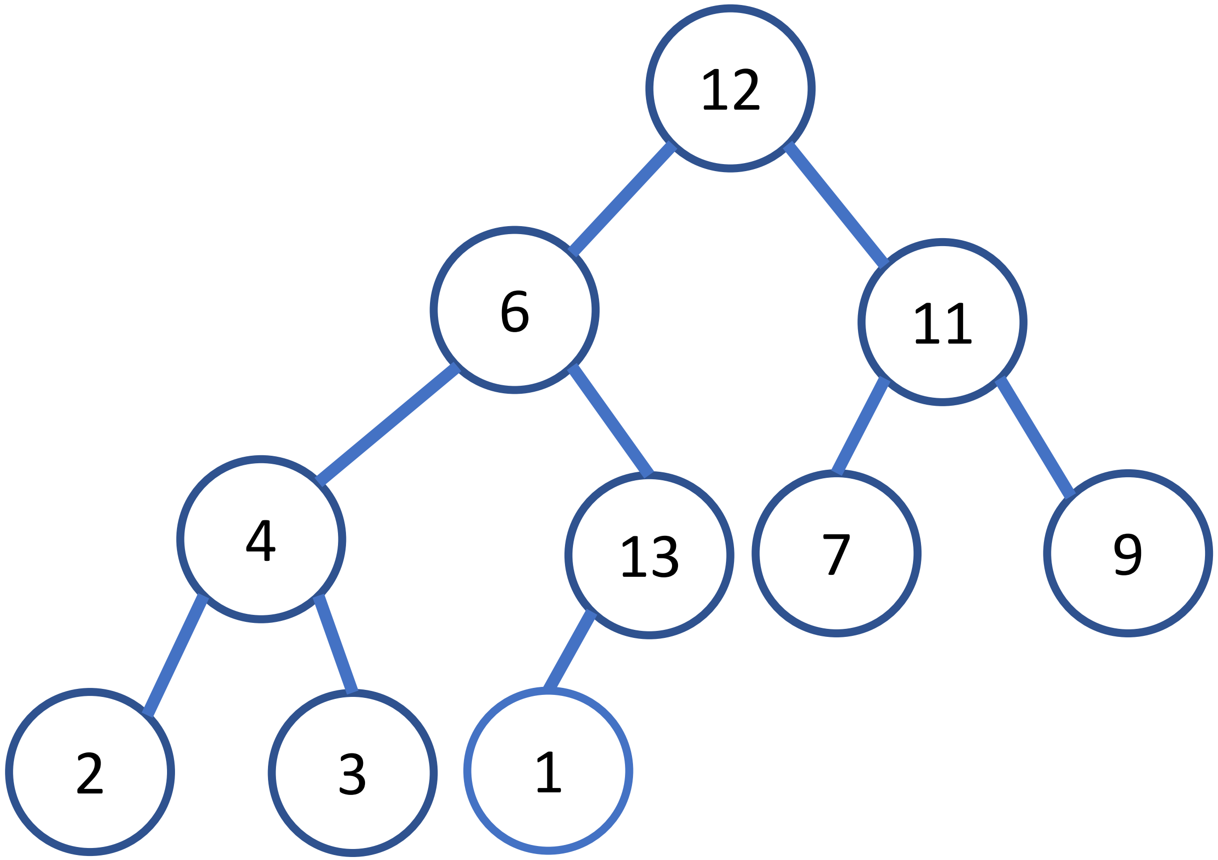
- Insert the new node at the last position in the heap (index N+1).

Inserting a new element into a max-heap



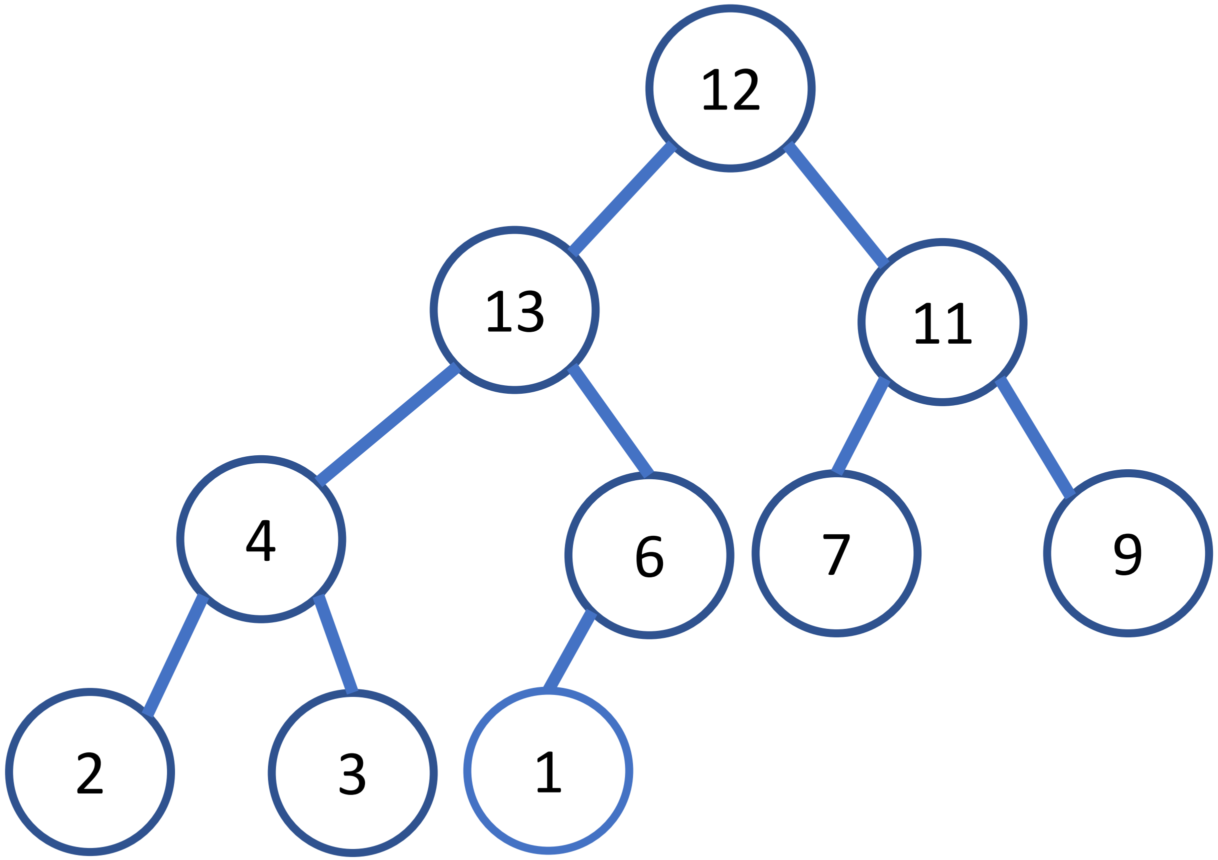
- `Insert the new node at the last position in the heap (index N+1).`
- `While the node is larger is greater than the parent, swap with parent. Repeat.`

Inserting a new element into a max-heap



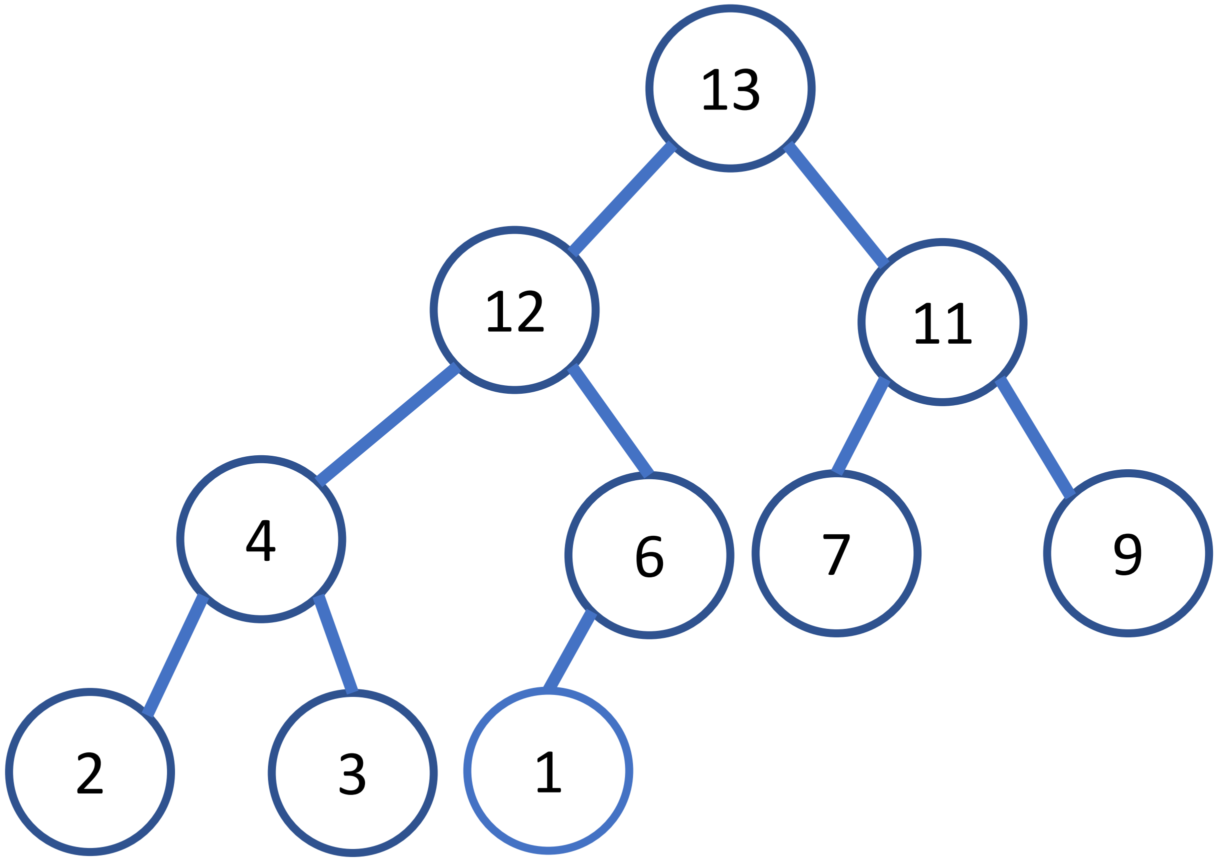
- Insert the new node at the last position in the heap (index N+1).
- While the node is larger is greater than the parent, swap with parent. Repeat.

Inserting a new element into a max-heap



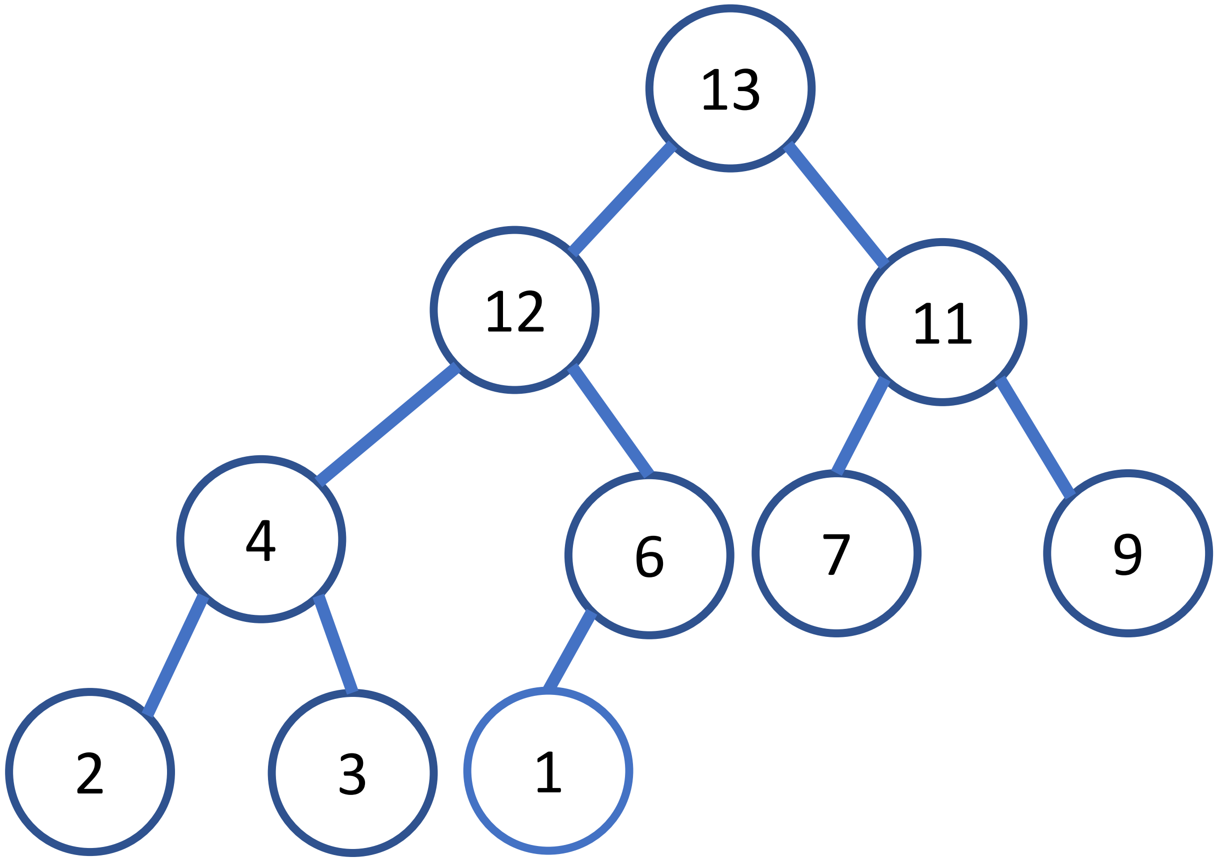
- Insert the new node at the last position in the heap (index N+1).
- While the node is larger is greater than the parent, swap with parent. Repeat.

Inserting a new element into a max-heap



- Insert the new node at the last position in the heap (index N).
- While the node is larger is greater than the parent, swap with parent. Repeat.

Inserting a new element into a max-heap

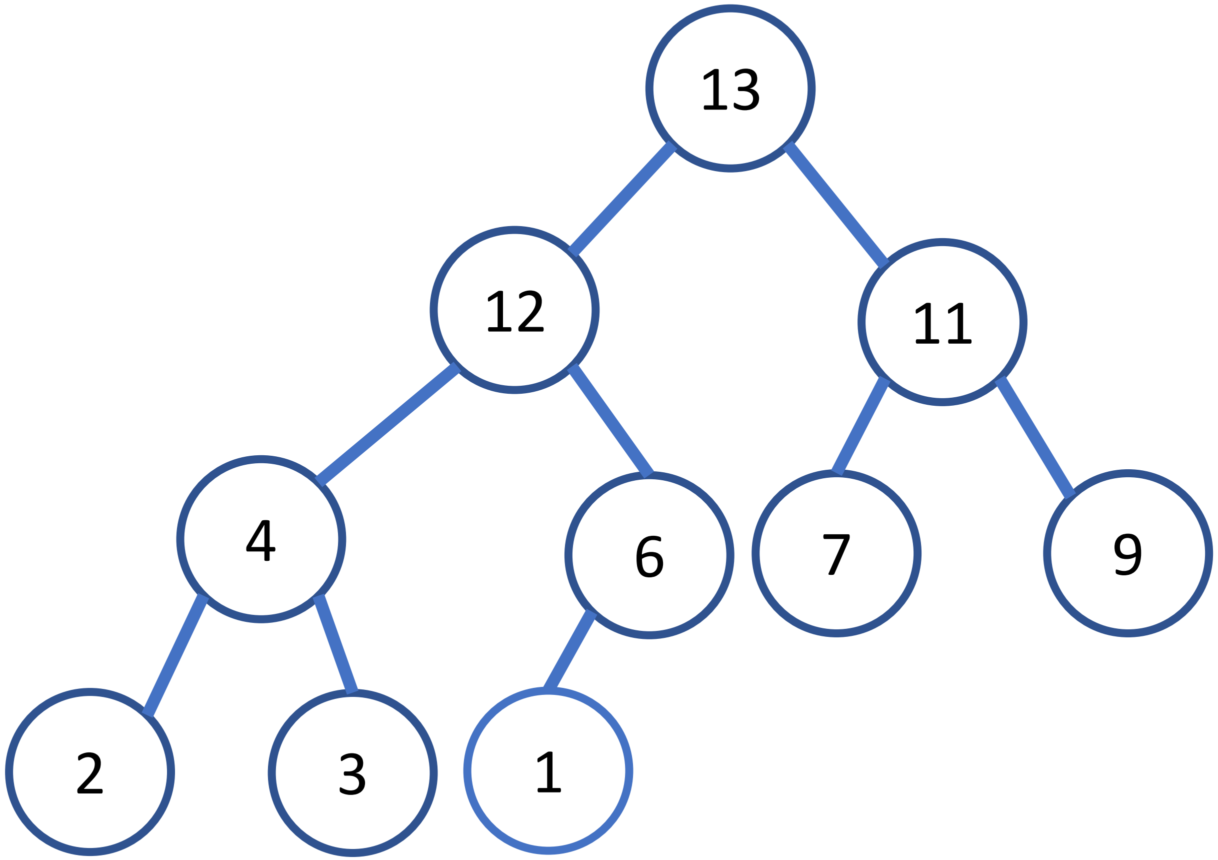


- Insert the new node at the last position in the heap (index N+1).
- While the node is larger is greater than the parent, swap with parent. Repeat.

```
Arr[N] = key;
int i = N;
N = N+1;

while(i > 0 && Arr[i] > Arr[i/2-1])
{
    Swap(&Arr[i], &Arr[i/2 - 1]);
    i = i/2 – 1;
}
```

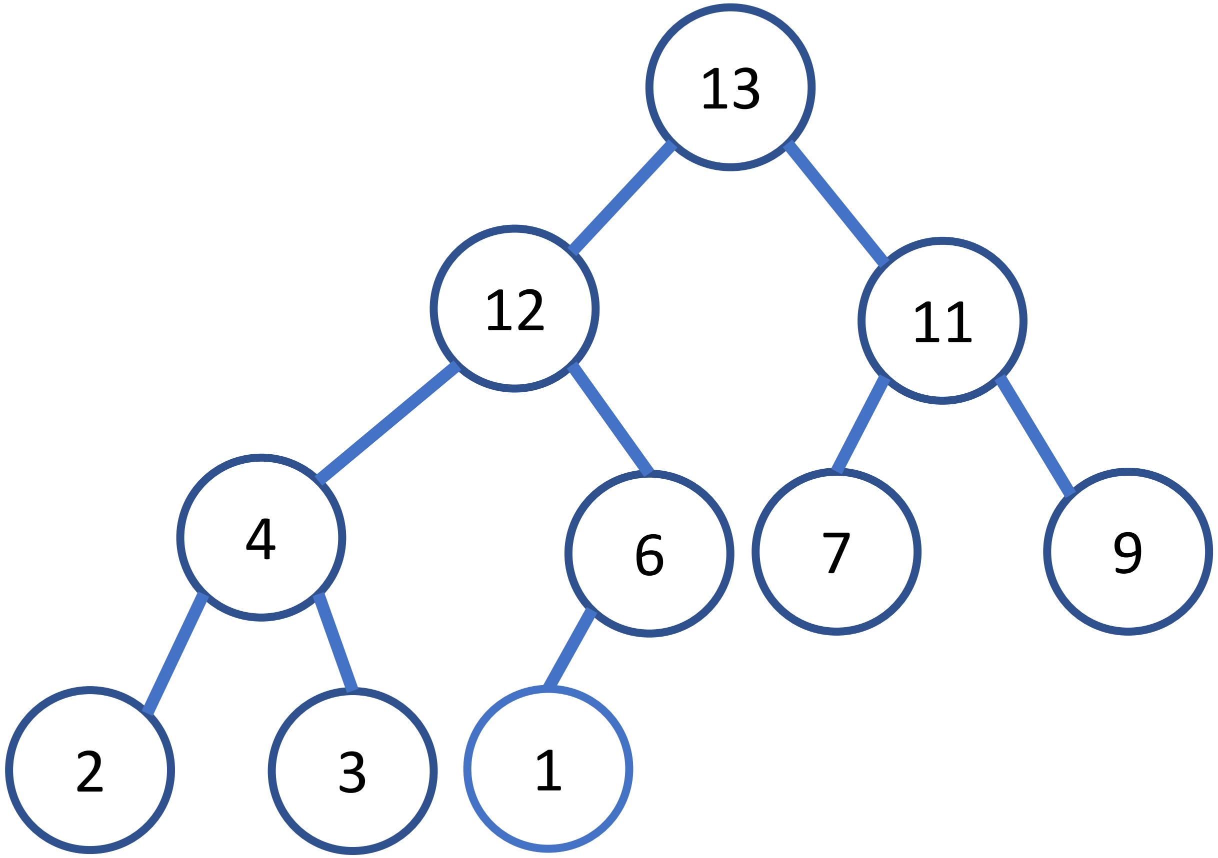
Inserting a new element into a max-heap



- Insert the new node at the last position in the heap (index N+1).
- While the node is larger is greater than the parent, swap with parent. Repeat.

How many operations?

Inserting a new element into a max-heap

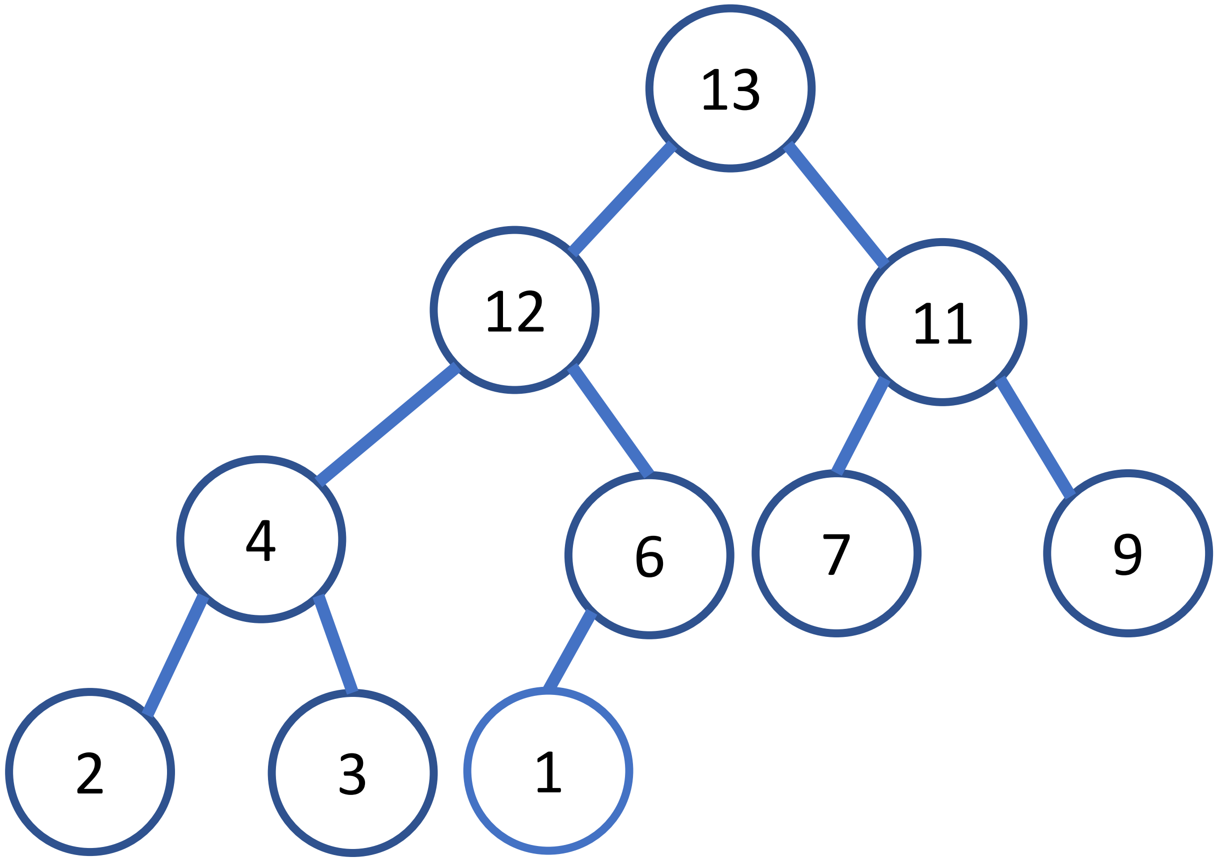


- `Insert the new node at the last position in the heap (index N+1).`
- `While the node is larger is greater than the parent, swap with parent. Repeat.`

How many operations?

- For a tree with height h, takes an average of O (h) operations.
-

Inserting a new element into a max-heap

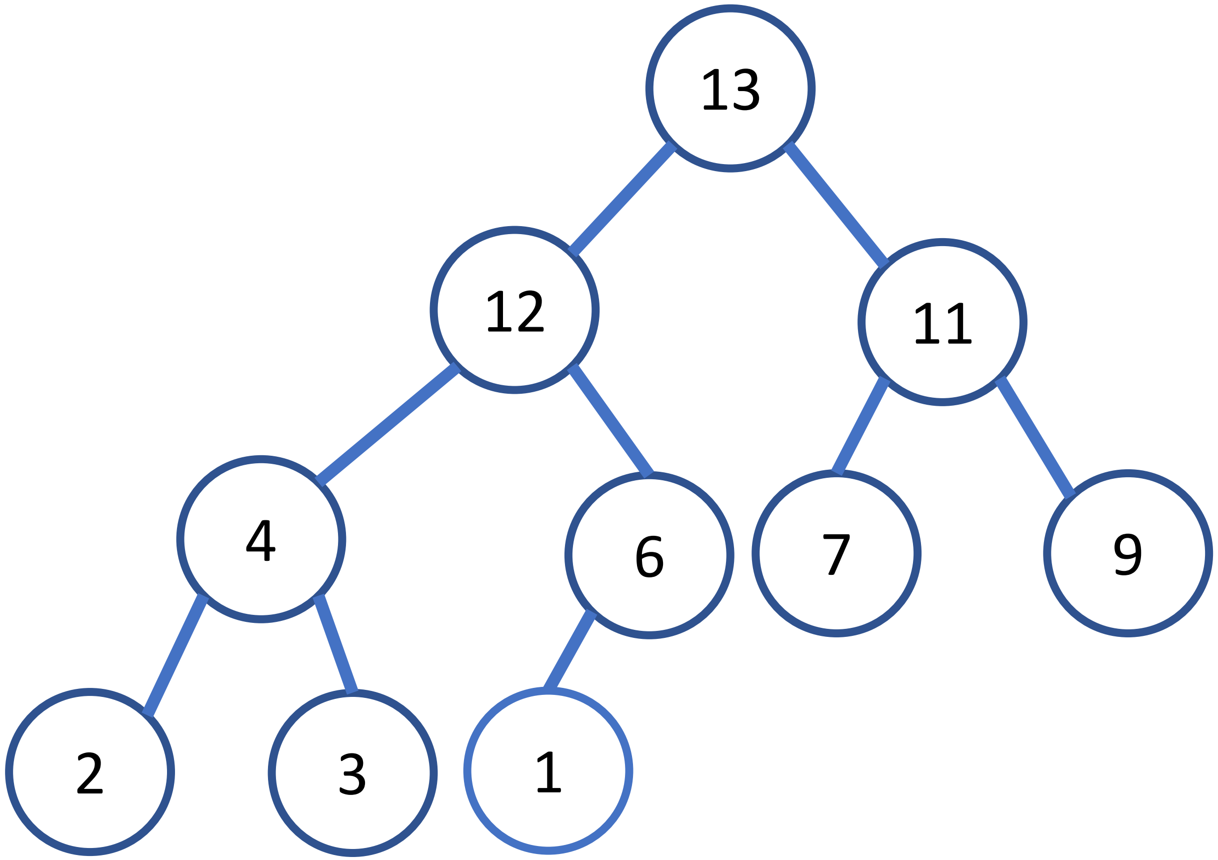


- `Insert the new node at the last position in the heap (index N+1).`
- `While the node is larger is greater than the parent, swap with parent. Repeat.`

How many operations?

- For a tree with height h, takes an average of O (h) operations.
- For an almost complete tree with N nodes, h = lg N.

Inserting a new element into a max-heap

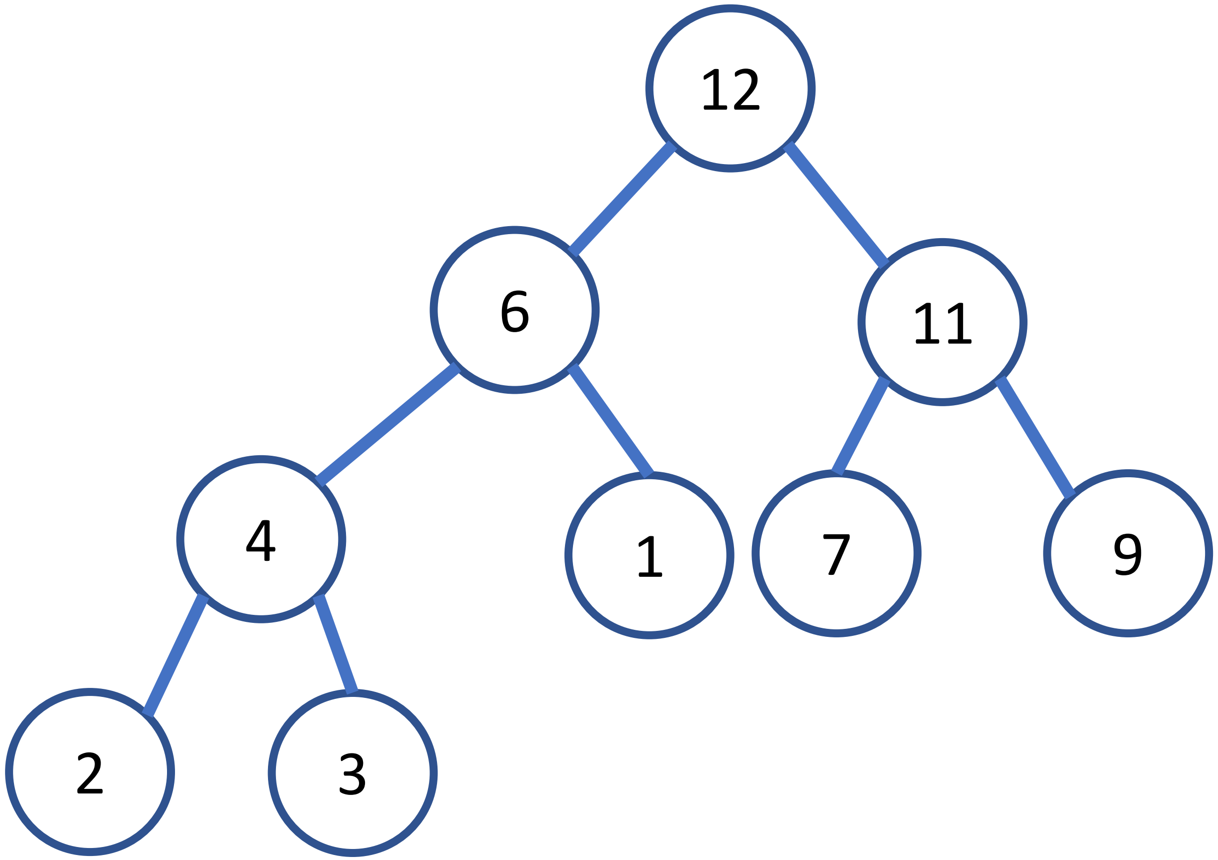


- `Insert the new node at the last position in the heap (index N+1).`
- `While the node is larger is greater than the parent, swap with parent. Repeat.`

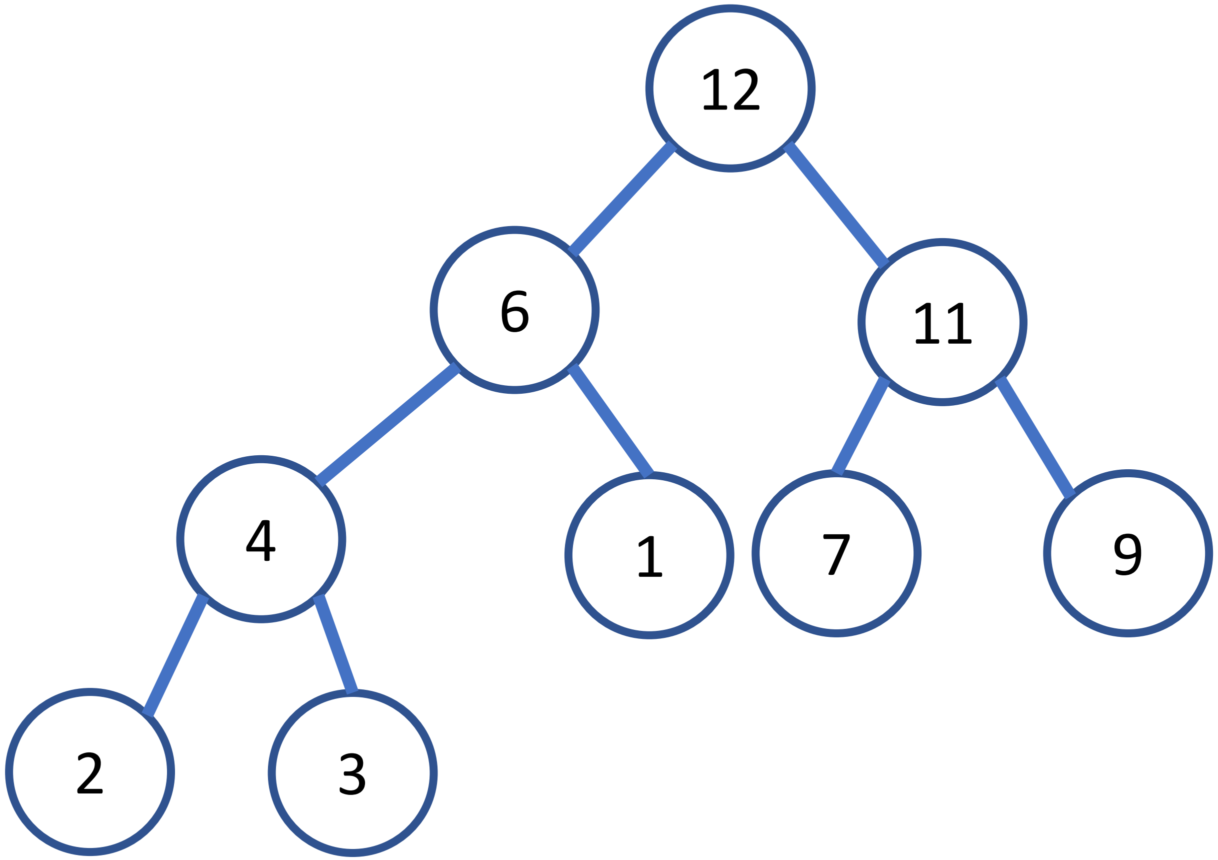
How many operations?

- For a tree with height h, takes an average of O (h) operations.
- For an almost complete tree with N nodes, h = lg N.
- Insertion complexity = O (lg N).

DeleteMax in a Max-heap

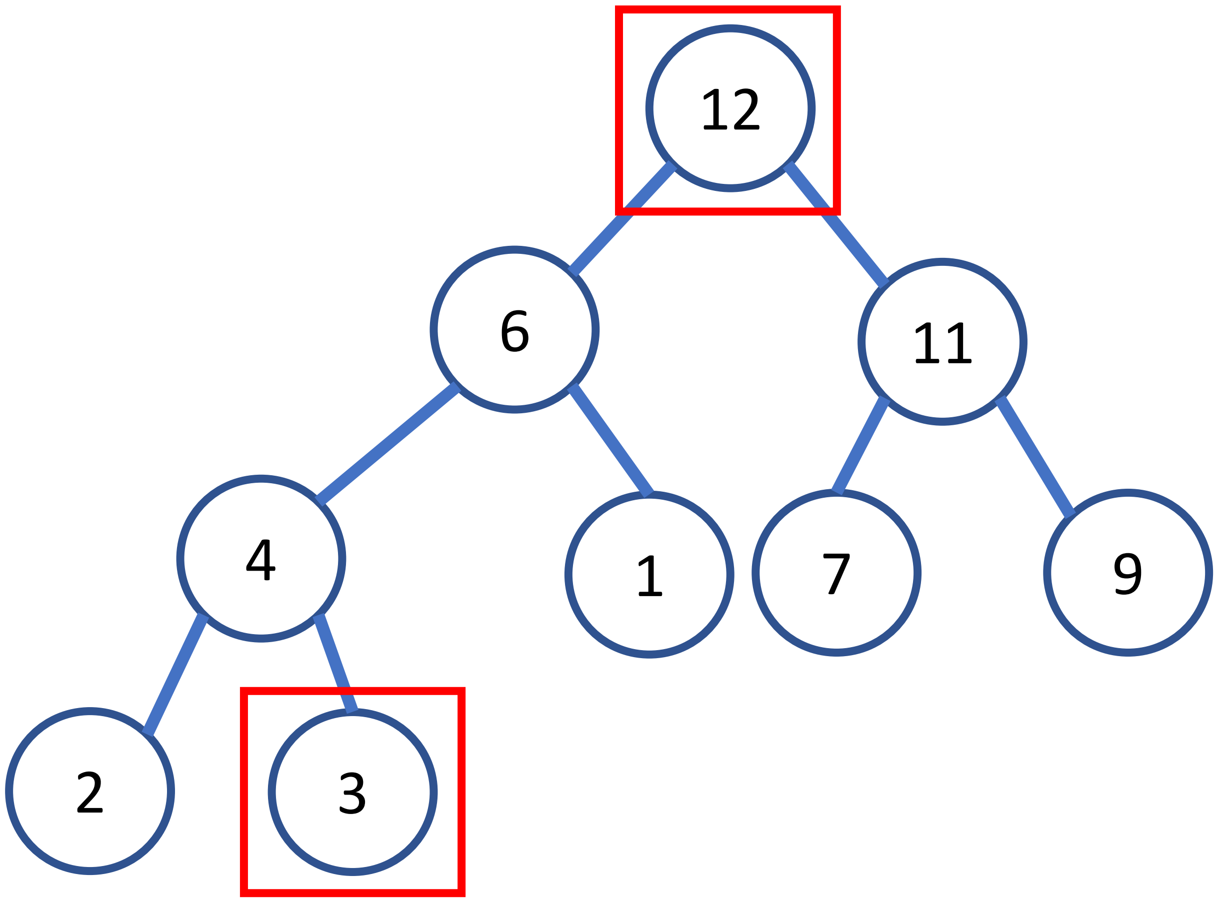


DeleteMax in a Max-heap



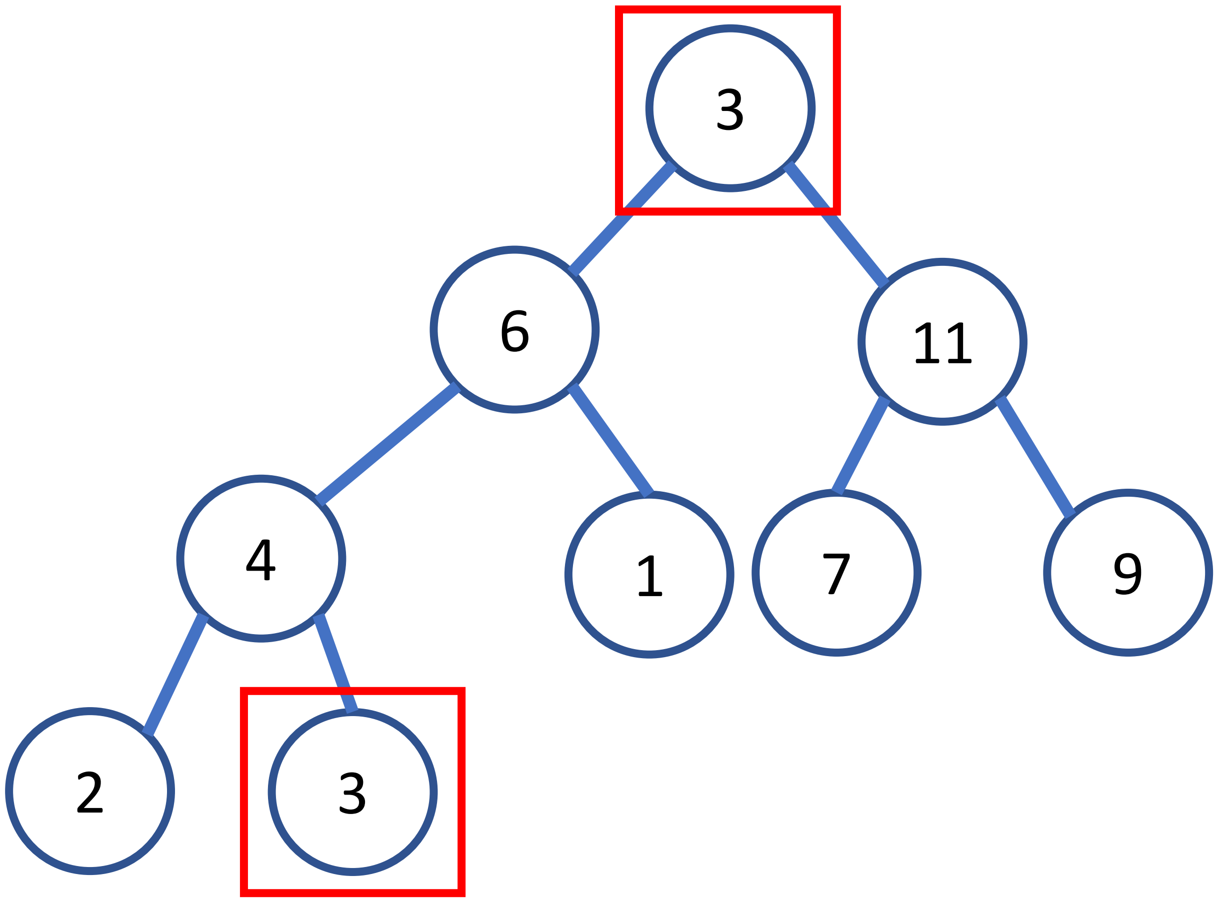
- Replace the root of the heap with the last element.
- Delete the last element.
- Max_heapify the root.

DeleteMax in a Max-heap



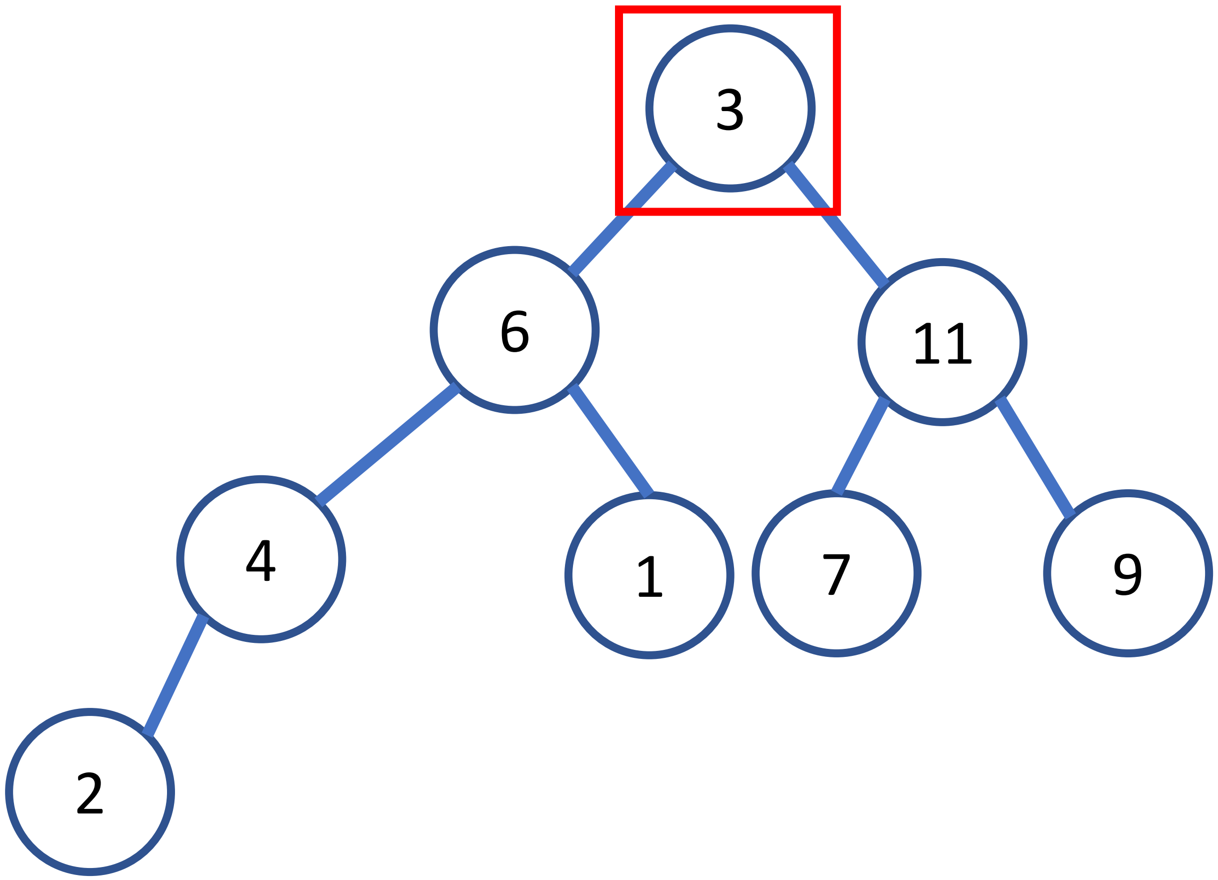
- Replace the root of the heap with the last element.
- Delete the last element.
- Max_heapify the root.

DeleteMax in a Max-heap



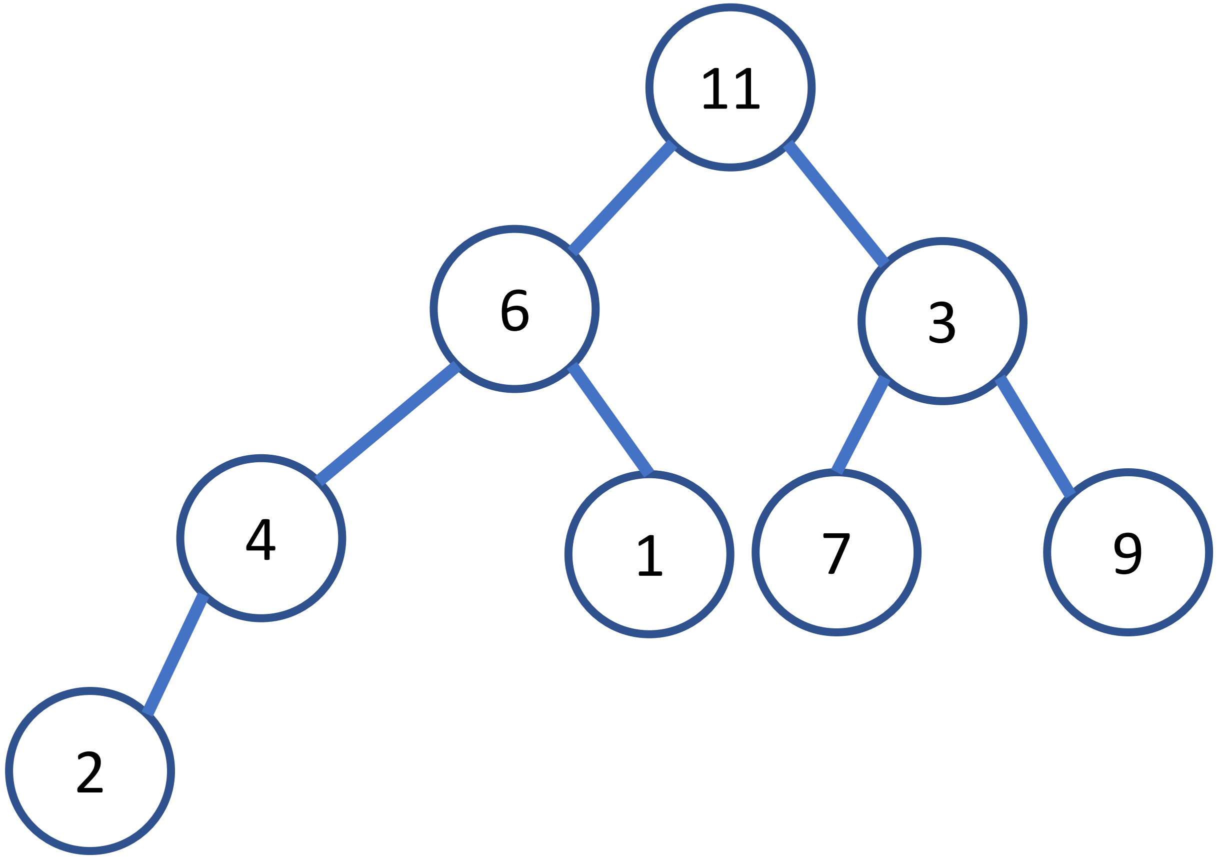
- Replace the root of the heap with the last element.
- Delete the last element.
- Max_heapify the root.

DeleteMax in a Max-heap



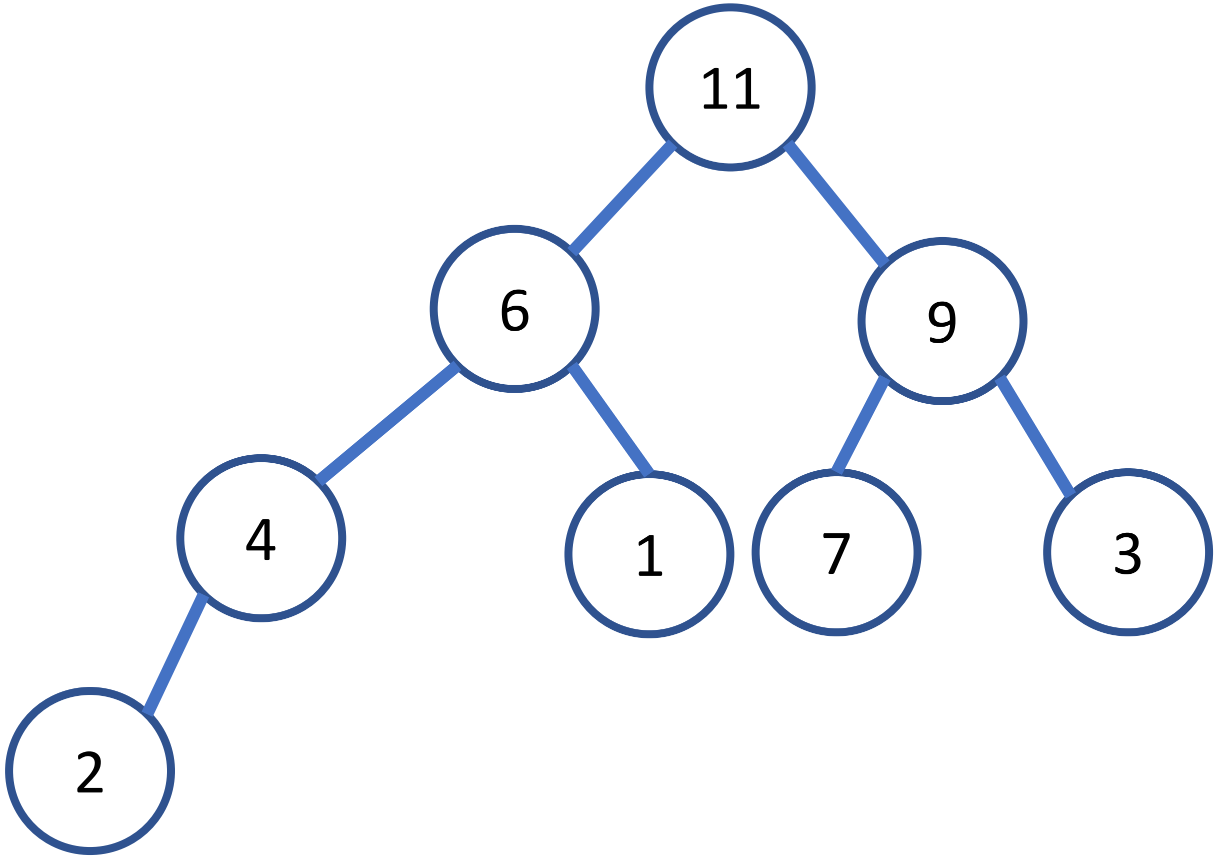
- `Replace the root of the heap with the last element.`
- `Delete the last element.`
- `Max_heapify the root.`

DeleteMax in a Max-heap



- Replace the root of the heap with the last element.
- Delete the last element.
- Max_heapify the root.

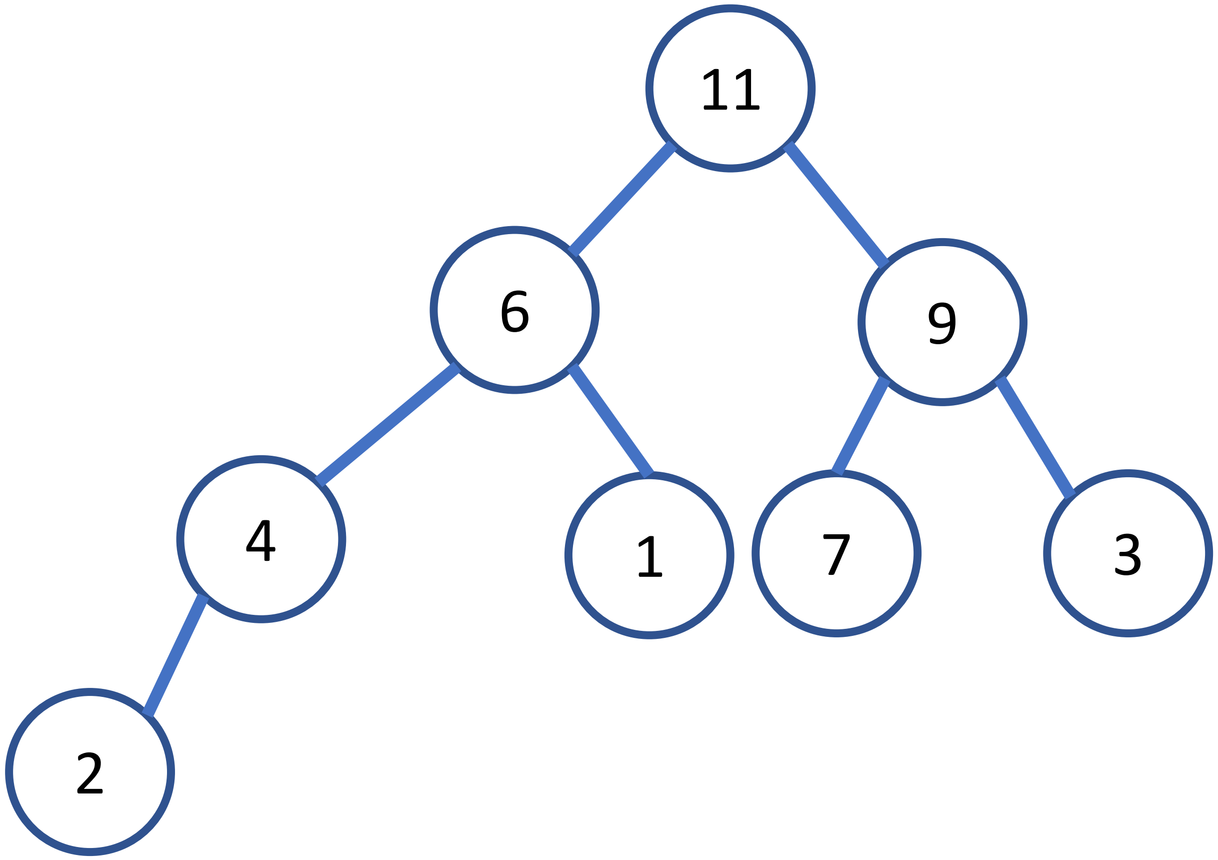
DeleteMax in a Max-heap



- Replace the root of the heap with the last element.
- Delete the last element.
- Max_heapify the root.

```
Arr[0] = Arr[N-1];
N = N – 1;
max_heapify(Arr, 0, N);
```

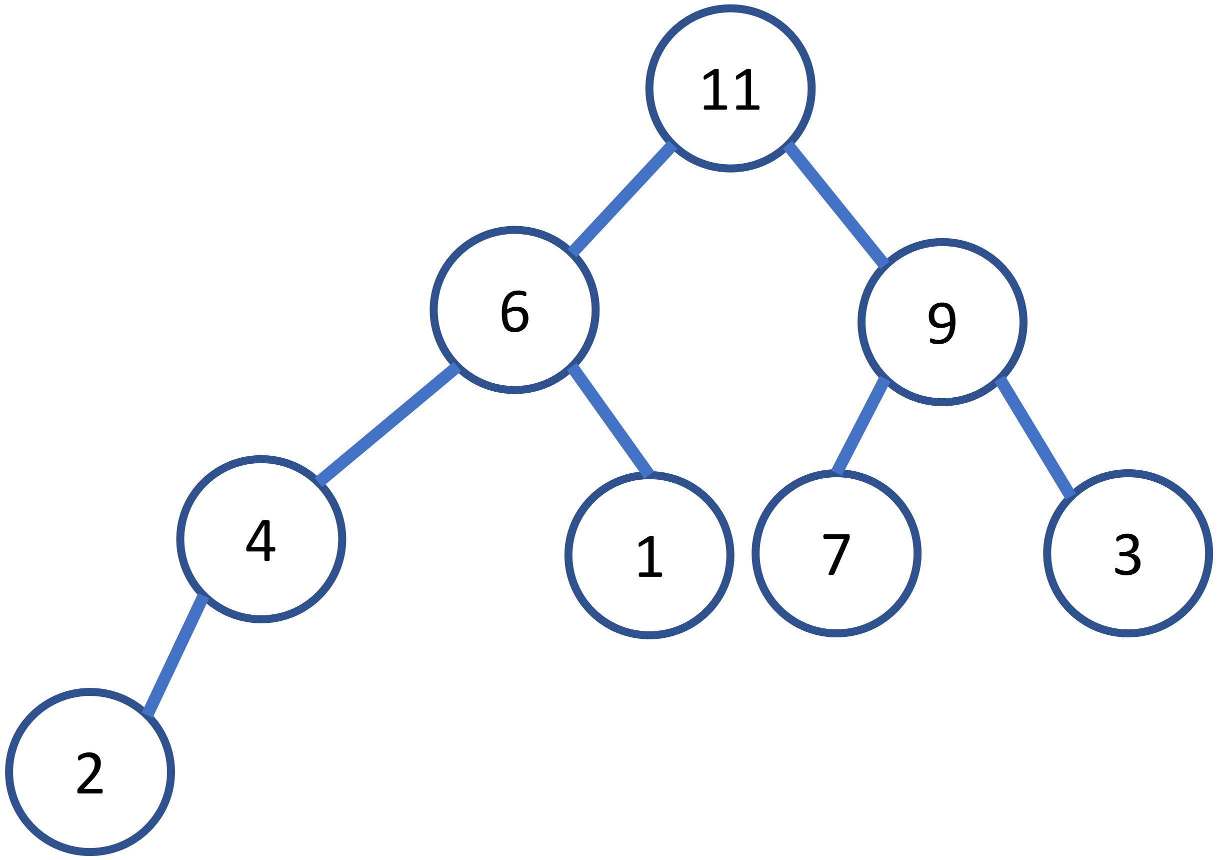
DeleteMax in a Max-heap



- `Replace the root of the heap with the last element.`
- `Delete the last element.`
- `Max_heapify the root.`

How many operations?

DeleteMax in a Max-heap



- `Replace the root of the heap with the last element.`
- `Delete the last element.`
- `Max_heapify the root.`

How many operations?

- For a tree with height h, takes an average of O (h) operations.
- For an almost complete tree with N nodes, h = lg N.
- DeleteMax complexity = O (lg N).

A final word on the complexity

	Circular Array	Heap
Enqueue	O(n)	O (lg N)
Dequeue	O(1)	O (lg N)
Peek the highest priority	O(1)	O(1)

Silver Problem

- Implement the functions extractMin and heapify for a Min-Heap.
- *Hints:*
  - *Same as the Max-Heap, just reverse the comparison signs.*
  - *In heapify, swap with the smallest.*
  - *You don't have to do the leftChild, rightChild and parent calculations. Use the available functions in the starter code.*

```
/*
When an element is removed/deleted. This method make sures the array
satisfies the heap property.
*/
void MinHeap::heapify(int i)
{
  int l = leftChild(i);
  int r = rightChild(i);
  int smallest = i;

  //TODO finsh the heapify function


}

```

```
// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
  if (currentSize <= 0)
    return INT_MAX;

  if (currentSize == 1)
  {
    currentSize--;
    return heapArr[0];
  }
  //TODO finsh the function

}
```

Gold Problem

- Implement the function deleteKey for a Min-Heap.
- *Hint: Use logic similar to deleteMax, except choose the right node whose value needs to be replaced.*

Questions?

Good luck for your finals!