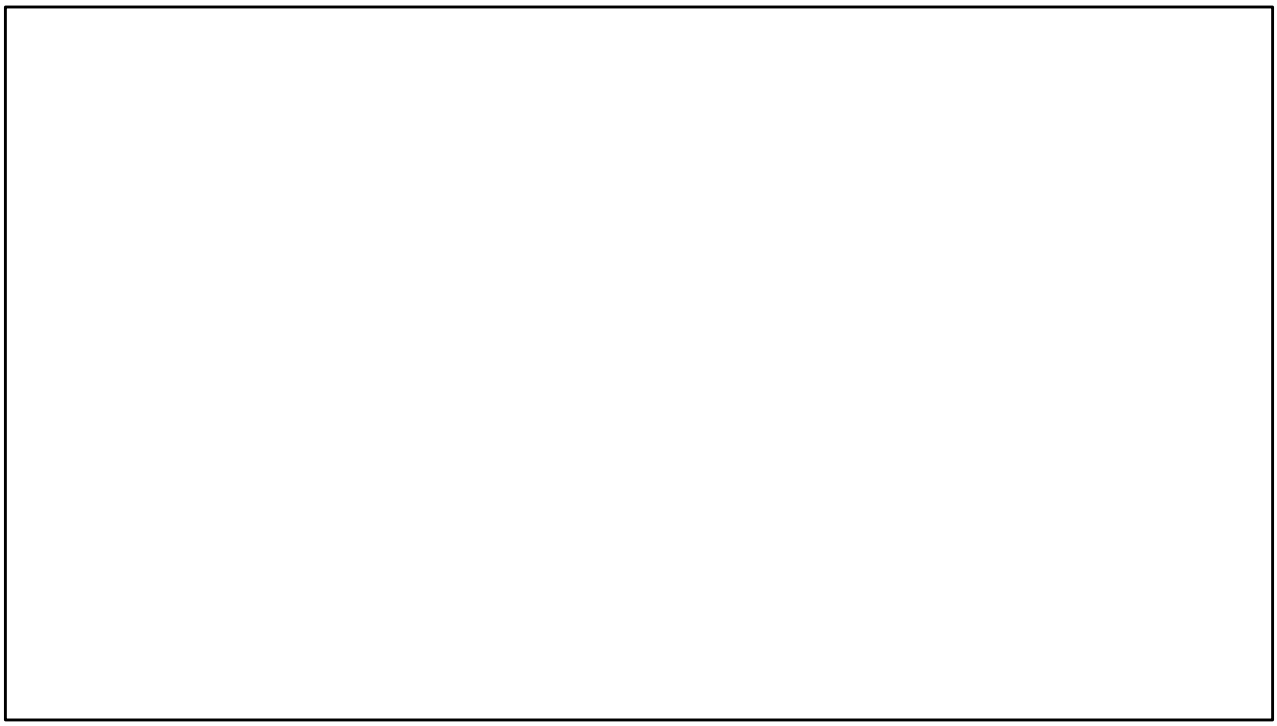


Today's topics

- Heap
- Heapsort

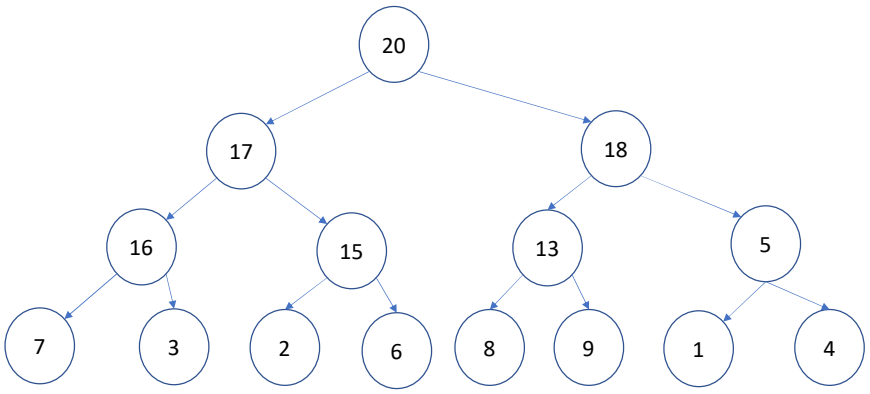
References

- Read chapter-6 from Cormen et al.

Complete binary tree

- A complete binary tree is a binary tree in which all the levels are completely filled

Complete binary tree

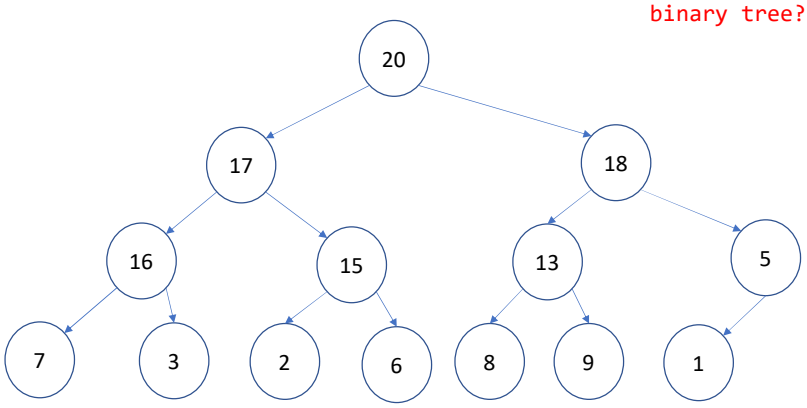


Nearly complete binary tree

- In a nearly complete binary tree, all the levels are completely filled except the last level, which is filled from the left up to a point

Nearly complete binary tree

Is this a nearly complete binary tree?

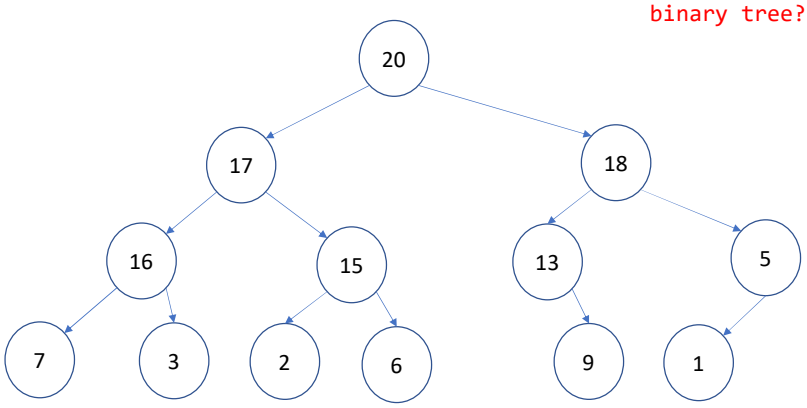


Yes, because all the levels from 0 to 2 are completely filled, and there is no gap between the leftmost position and the position of the last leaf in the last level.

Nearly complete binary tree

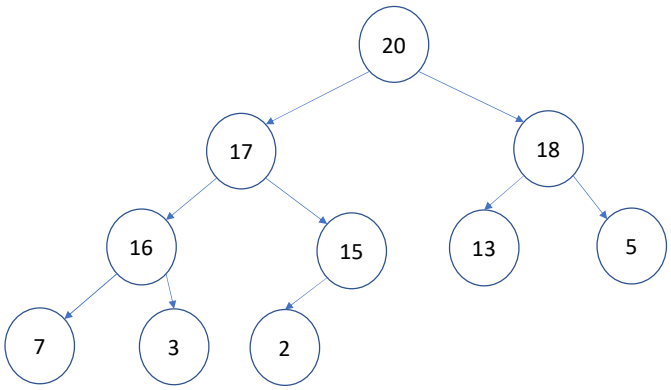
Is this a nearly complete binary tree?

Not



No, because an element (left child of 13) is missing between the leftmost position and the position of the last leaf in the last level.

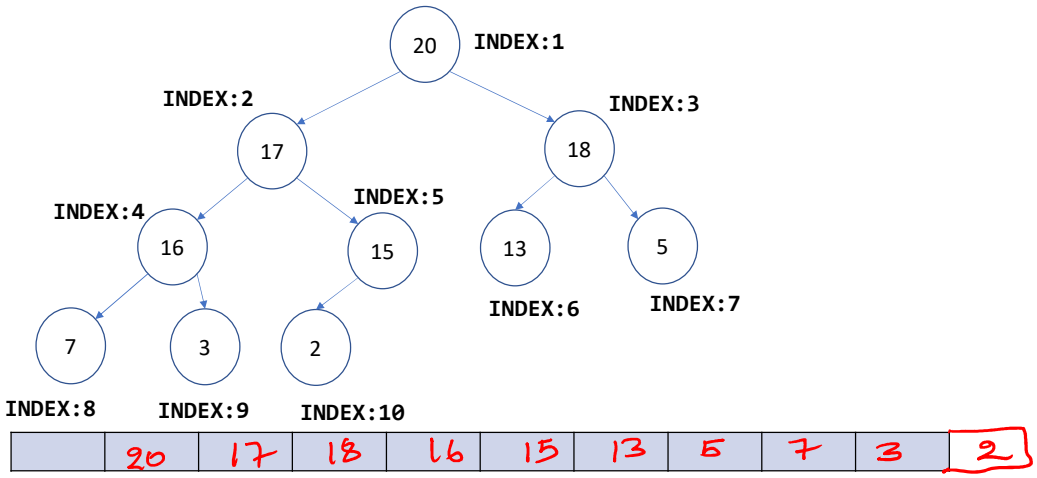
Nearly complete binary tree in an array



Nearly complete binary tree in an array

- We can store a nearly complete binary array in a tree as follows:
  - The starting index of the array is 1
  - The elements are stored one by one in the array in the following order
    - The root is stored first
    - Then, all elements at level-1 are stored one by one from left to right
    - Then, all elements at level-2 are stored one by one from left to right
    - and so on
    - Finally, all elements at the last level are stored one by one from left to right

Nearly complete binary tree in an array



The root is stored at index-1, followed by the elements at level-1 from left to right, followed by the elements at level-2 from left to right, and so on. Notice that for a given node at index i, its left child is stored at index 2i, and the right child is stored at index 2i+1. The parent of i is stored at the index floor(i/2).

Nearly complete binary tree

- The parent of node at index i is stored at index $\left\lfloor \frac{i}{2} \right\rfloor$
- The left child of node at index i is stored at index $2 * i$
- The right child of node at index i is stored at index $2 * i + 1$

Nearly complete binary tree

PARENT(i)

return $\left\lfloor \frac{i}{2} \right\rfloor$

LEFT(i)

return 2*i

RIGHT(i)

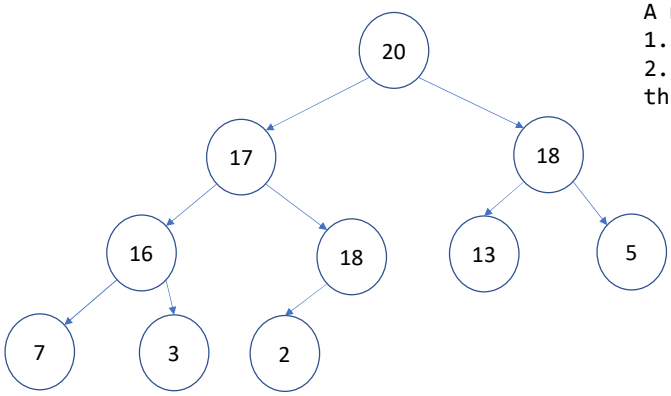
return 2*i + 1

Max heap

Max heap

- The max heap follows the following properties
  - Max heap is a nearly complete binary tree
  - The value stored at any node is greater than or equal to the values of all of its descendants

Max heap



Is this a max heap?

A max heap is:
1. nearly complete binary tree
2. value at any node is greater than or equal to its descendants

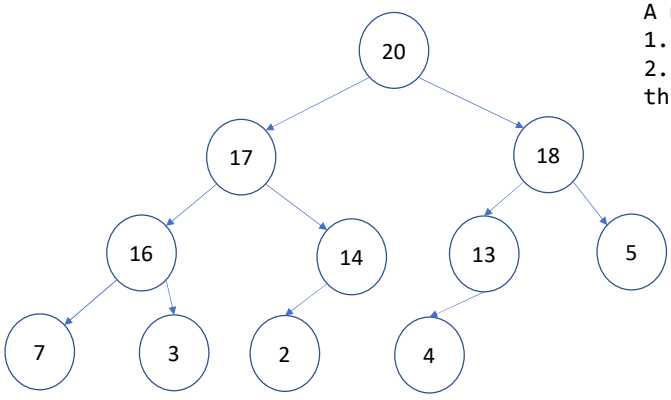
NO

18 > 17

17 is ~~parent~~ of 18

This is not a max heap because the parent of the node containing 18 contains 17, which is less than 18.

Max heap



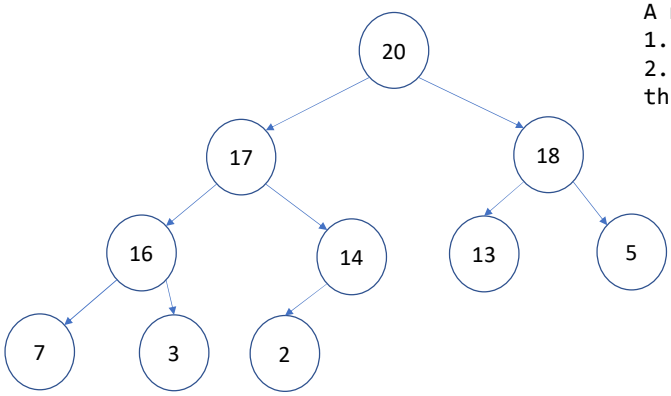
Is this a max heap?

A max heap is:
1. nearly complete binary tree
2. value at any node is greater than or equal to its descendants

Not
Right child of
14 is missing

This is not a max heap because the tree is not a nearly complete binary tree. The right child of 14 is missing between the leftmost position and the position of the last leaf in the last level.

Max heap



Is this a max heap?

A max heap is:
1. nearly complete binary tree
2. value at any node is greater than or equal to its descendants

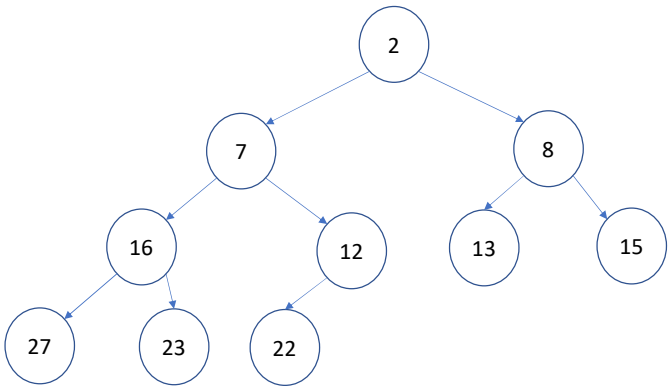
yes

Yes, this is a max heap because all the properties of the max heap are satisfied.

Min heap

- The min heap follows the following properties
  - Min heap is a nearly complete binary tree
  - The value stored at any node is less than or equal to the values of all of its descendants

Min heap



The tree on the left is a nearly complete binary tree.

The value stored at any node is less than or equal to the values of all of its descendants.

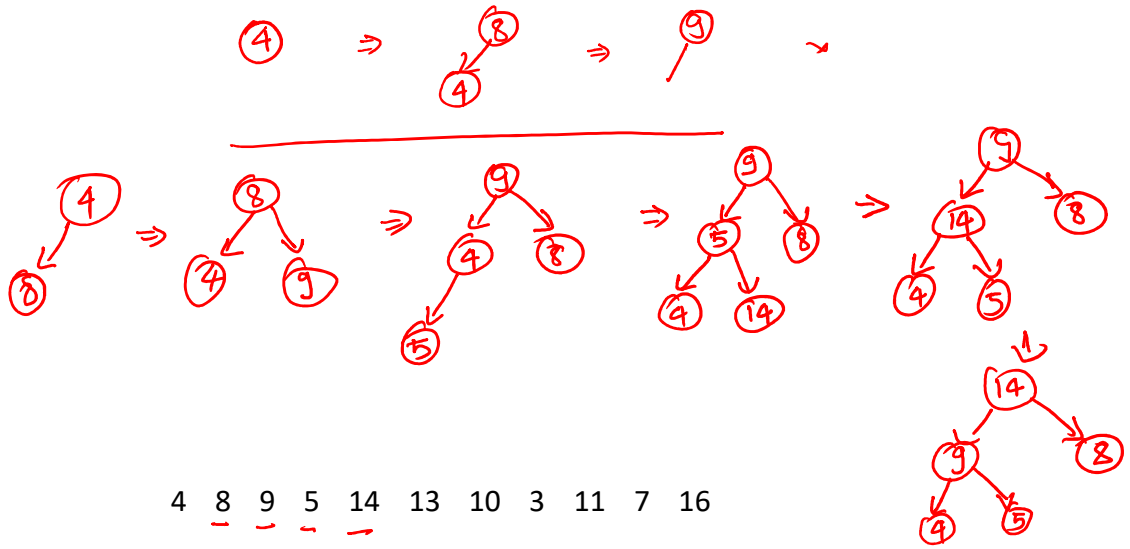
Therefore, the tree is a min heap.

Max heap ADT

- **Max-Heapify(H, i):** Build max heap when the max heap property is satisfied at all descendants of node i
- **Build-Max-Heap(H):** Build max heap from an unsorted array
- **Max-Heap-Maximum(H):** Return the value stored at root (maximum key)
- **Max-Heap-Extract-Max(H):** Delete the root node (maximum key) from H and returns its value
- **Heapsort(H):** Sort the elements of max heap H
- **Max-Heap-Insert(H, k):** Insert a new key (k) in the max heap H
- **Max-Heap-Increase-Key(H, i, keyval):** Increase key at index i to a new value (keyval) in the max heap H

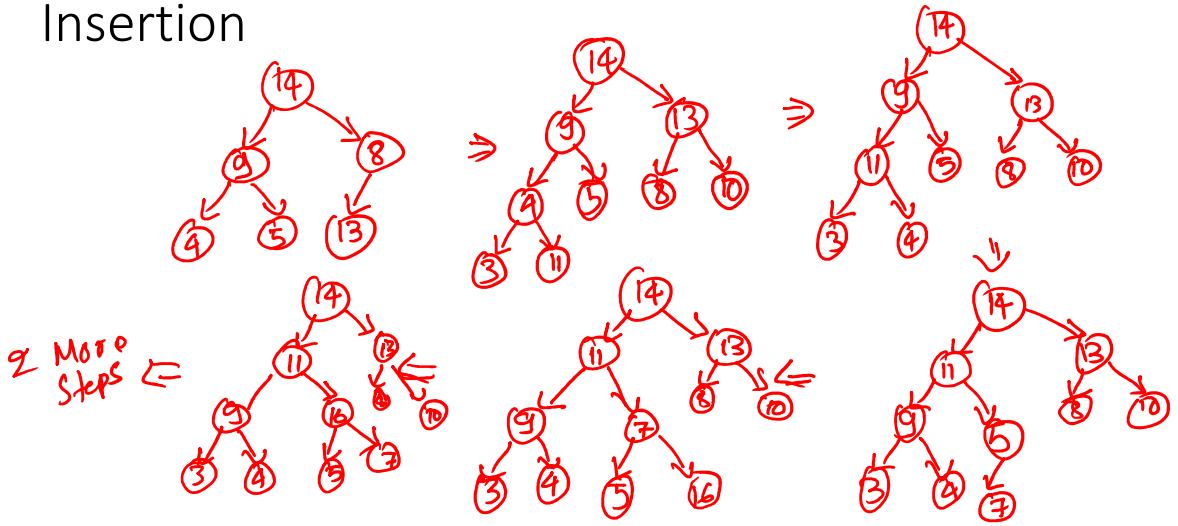
Max heap insert

Insertion



To insert a value in a max-heap, first, we store the new value at the first available index, say i, in the nearly complete binary tree. Afterward, we are done if the parent of i contains a key greater than or equal to the key stored at node i. Otherwise, we exchange the value of node i with its parent. Now, we repeat the same steps for the parent of i, until we are done or reach the root node.

Insertion



4 8 9 5 14 13 10 3 11 7 16

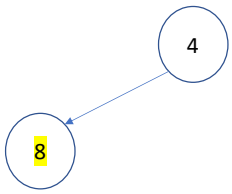
Insertion

4 8 9 5 14 13 10 3 11 7 16

Insertion

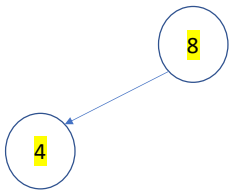
4 8 9 5 14 13 10 3 11 7 16

Insertion



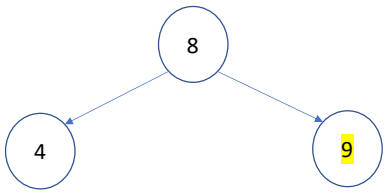
4 8 9 5 14 13 10 3 11 7 16

Insertion



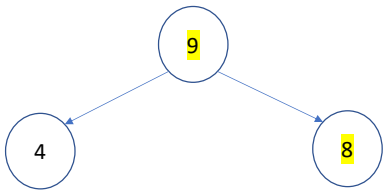
4 8 9 5 14 13 10 3 11 7 16

Insertion



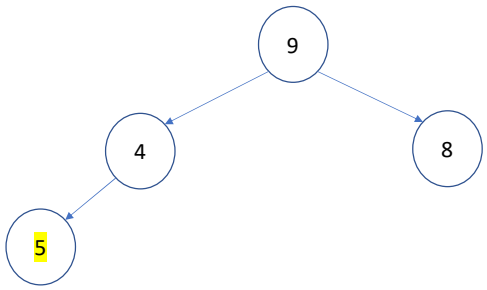
4 8 9 5 14 13 10 3 11 7 16

Insertion



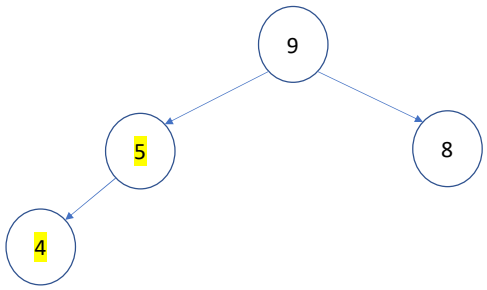
4 8 9 5 14 13 10 3 11 7 16

Insertion



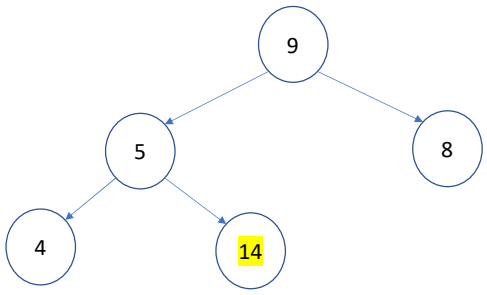
4 8 9 5 14 13 10 3 11 7 16

Insertion



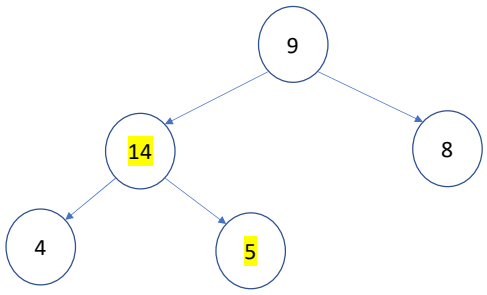
4 8 9 5 14 13 10 3 11 7 16

Insertion



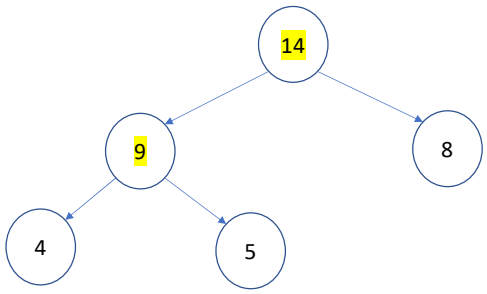
4 8 9 5 14 13 10 3 11 7 16

Insertion



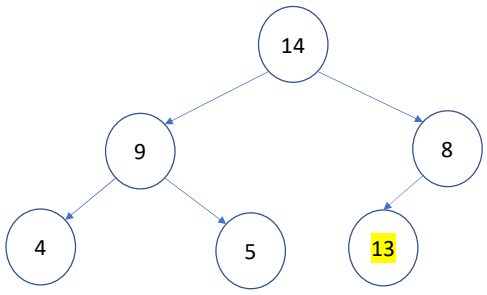
4 8 9 5 14 13 10 3 11 7 16

Insertion



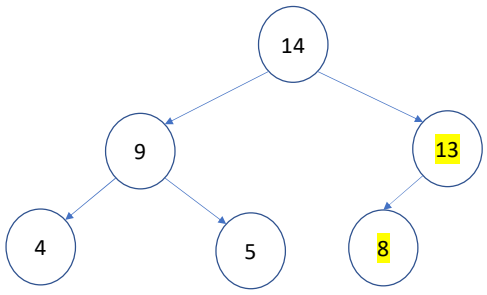
4 8 9 5 14 13 10 3 11 7 16

Insertion



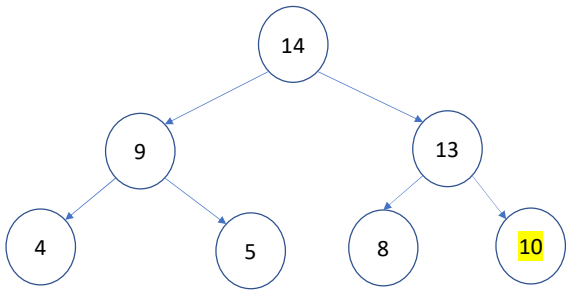
4 8 9 5 14 13 10 3 11 7 16

Insertion



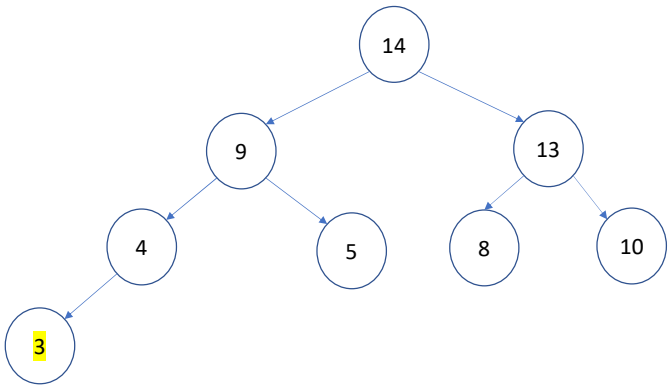
4 8 9 5 14 13 10 3 11 7 16

Insertion



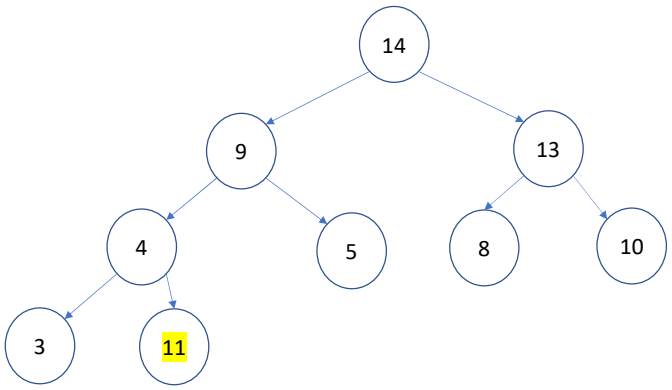
4 8 9 5 14 13 10 3 11 7 16

Insertion



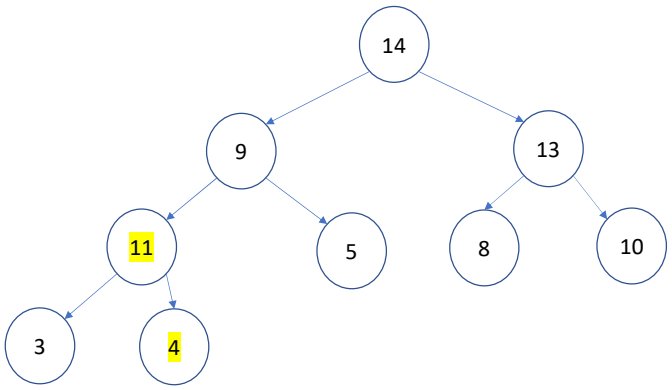
4 8 9 5 14 13 10 3 11 7 16

Insertion



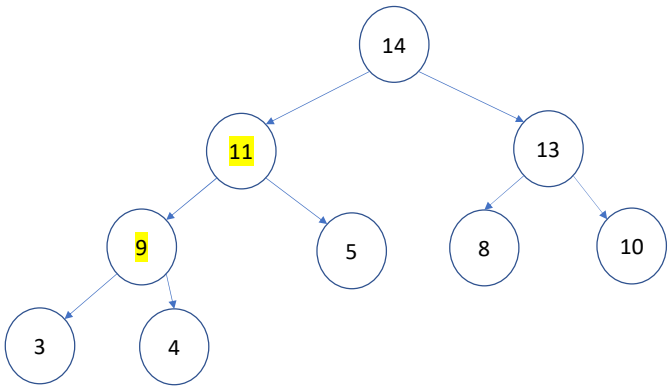
4 8 9 5 14 13 10 3 11 7 16

Insertion



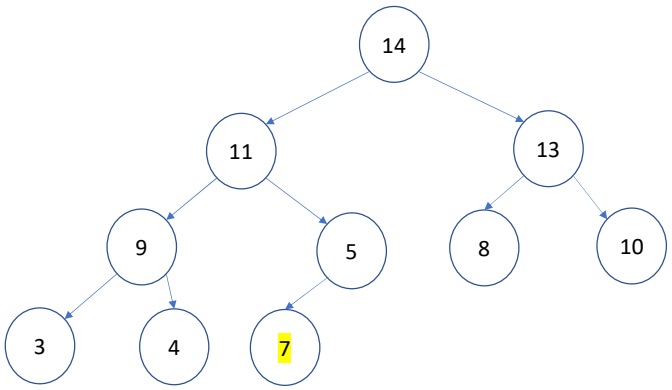
4 8 9 5 14 13 10 3 11 7 16

Insertion



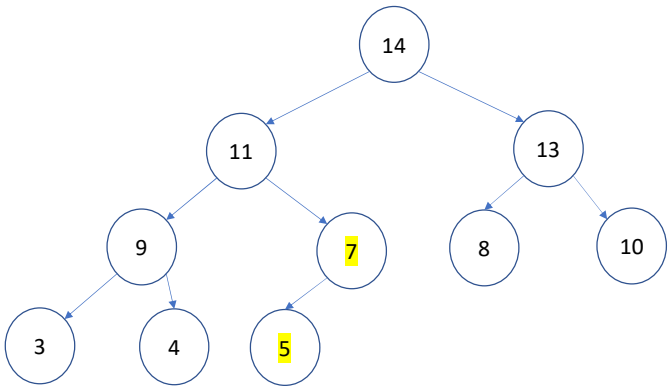
4 8 9 5 14 13 10 3 11 7 16

Insertion



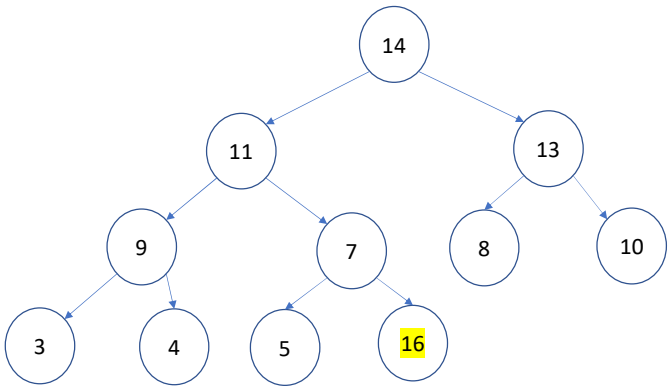
4 8 9 5 14 13 10 3 11 7 16

Insertion



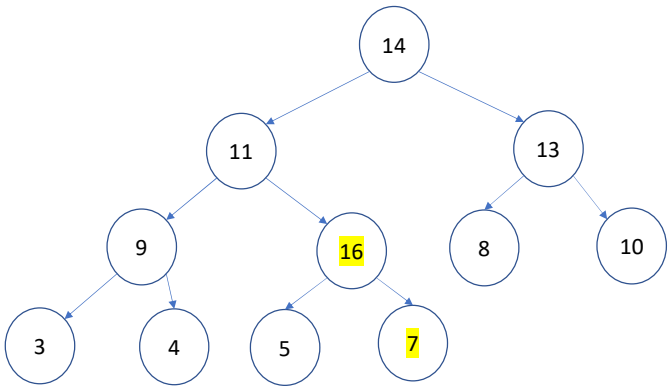
4 8 9 5 14 13 10 3 11 7 16

Insertion



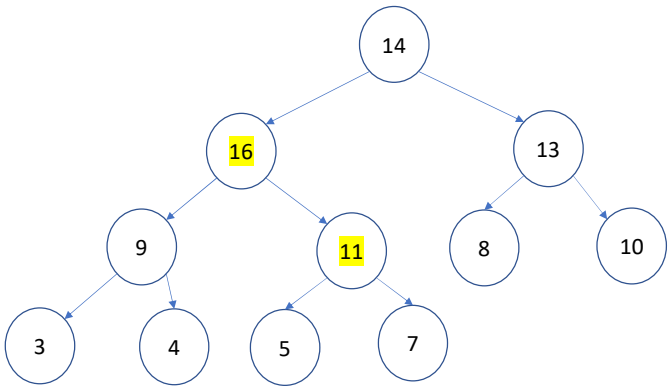
4 8 9 5 14 13 10 3 11 7 16

Insertion



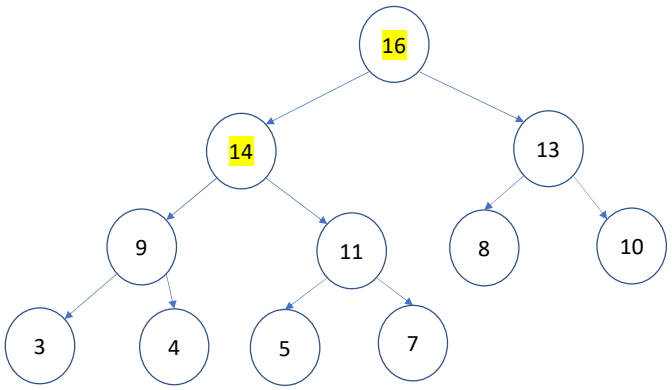
4 8 9 5 14 13 10 3 11 7 16

Insertion



4 8 9 5 14 13 10 3 11 7 16

Insertion



4 8 9 5 14 13 10 3 11 7 16

Insertion algorithm

```
1.  struct Heap {
2.    int *arr;
3.    int capacity;    // maximum size of heap
4.    int heap_size;  // number of elements in the heap
5.  };

6.  Max_Heap_insert(H, x)
7.  // H is a reference to the heap of type struct Heap
8.  // x is an integer input that needs to be inserted
9.  // Output: insert x in the heap

10. if H->heap_size == H->capacity
11.    perror("heap overflow")
12. H->heap_size = H->heap_size + 1
13. H->arr[H->heap_size] = x

14. i = H->heap_size
15. while i > 1 and H->arr[PARENT(i)] < H->arr[i]
16.    exchange H->arr[i] with H->arr[PARENT(i)]
17.    i = PARENT(i)
```

In this algorithm, we are using a fixed-size array. In "struct Heap", capacity is the maximum number of elements that can be stored in the heap. The heap_size contains the actual number of elements in the heap. This algorithm first checks the overflow condition. If there is sufficient space in the heap, at line-13, it inserts the element at the first available position in the nearly complete binary tree. The loop at line 15 iterates from the leaf node to the root, exchanging the value of the child with its parent until the heap property is satisfied at the parent.

Insertion

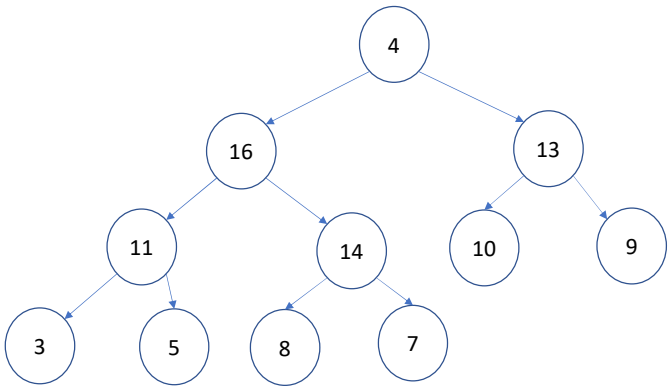
- What is the time complexity of a single insertion? $\log n$
- What is the time complexity of n insertions? $n \log n$

Max-Heapify

Max-Heapify

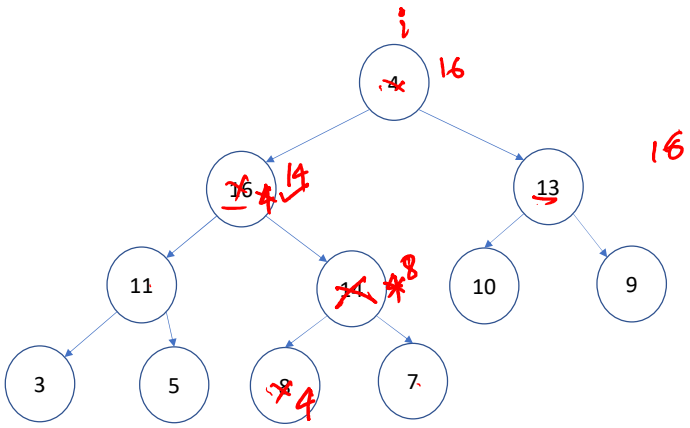
- Max_Heapify(H, i)
  - Binary trees rooted at indices LEFT(i) and RIGHT(i) are max heaps.
  - The Max_Heapify procedure ensures that the binary tree rooted at index i follows the max heap property

Max-Heapify

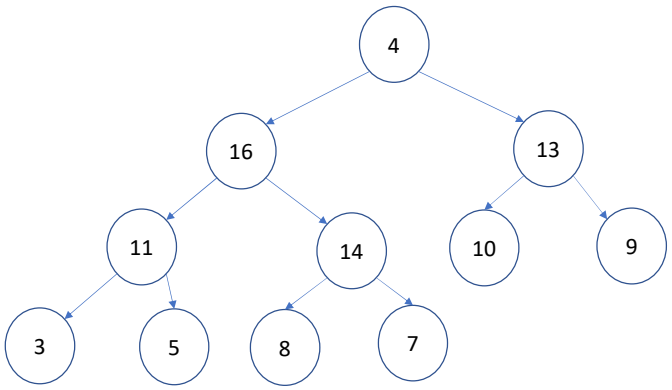


The Max-Heapify function can be applied on a node whose left and right children are already the max heaps. If Max-Heapify is applied to node n, it ensures that node n follows the max heap property after the Max-Heapify operation. The Max-Heapify works as follows. First, we compute the largest key among the key at node n, the key at the left child of n, and the key at the right child of n. If the largest key is the same as the key at n, then we are done. Otherwise, the key at n is exchanged with the largest key, and the same steps are performed at the node that previously contained the largest key.

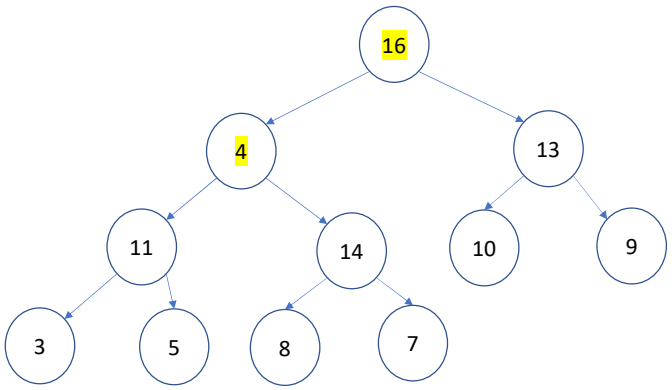
Max-Heapify



Max-Heapify

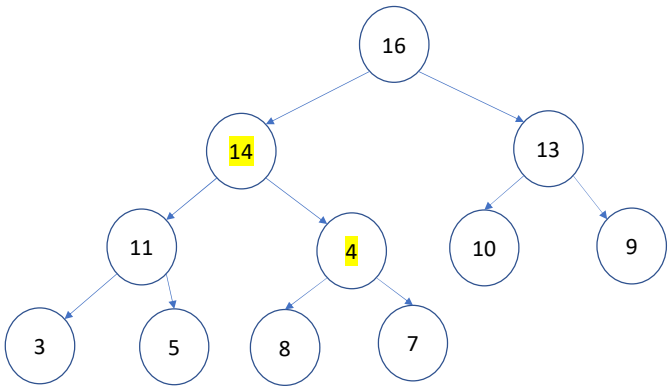


Max-Heapify



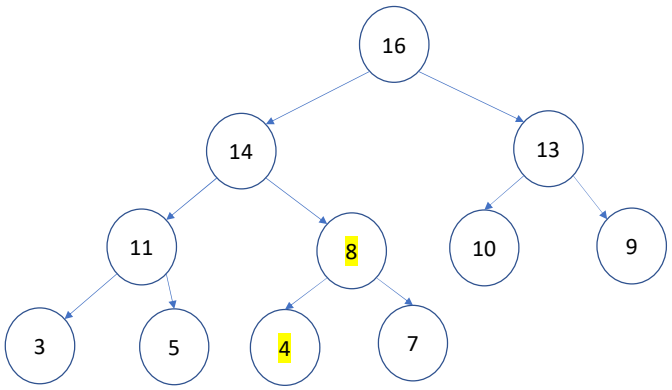
STEP-1

Max-Heapify



STEP-2

Max-Heapify



STEP-3

Max-Heapify

- Base case: if key at index i greater or equal to key at both LEFT(i) and RIGHT(i), then the tree is already a max heap, return
- Recursive step: Swap the value of the root node with the largest of left and right children (say node j)
  - recursively call Max-Heapify for the node j
- Time complexity : O(log n)

Max_Heapify

```
1.  struct Heap {
2.    int *arr;
3.    int capacity;    // maximum size of heap
4.    int heap_size;  // number of elements in the heap
5.  };

6.  Max_Heapify(H, i)
7.  // H is a reference to the heap of type struct Heap
8.  // i is the index at which the Max_Heapfify needs to be performed
9.  // Output: node at index i satisfies the max heap property

10. l = LEFT(i)
11. r = RIGHT(i)
12. if (l <= H->heap_size and H->arr[l] > H->arr[i]
13.    largest = l
14. else largest = i
15. if r <= H->heap_size and H->arr[r] > H->arr[largest]
16.    largest = r
17. if largest != i
18.    exchange H->arr[i] with H->arr[largest]
19.    Max_Heapify(H, largest)
```

If the Max_Heapify is performed at the node at index i, at lines 12-16, we are computing an index among i, LEFT(i), and RIGHT(i) that contains the largest key, called largest. If the largest is not i, then at line-18, we exchange the node's value at index i with the node's value at index largest, and at line-19, we perform the Max_Heapify operation at index largest.

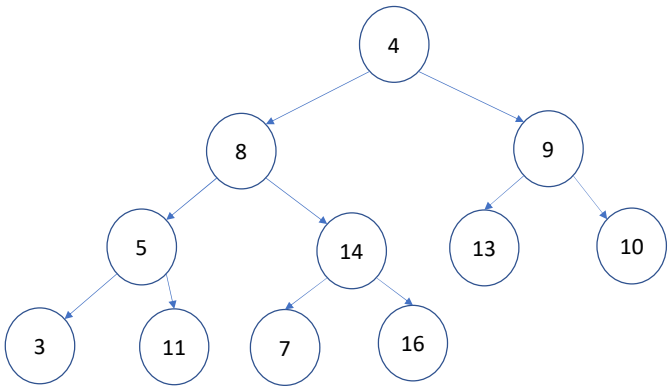
Build-Max-Heap

Build-Max-Heap

- The goal is to make max heap from an unordered array

Build-Max-Heap

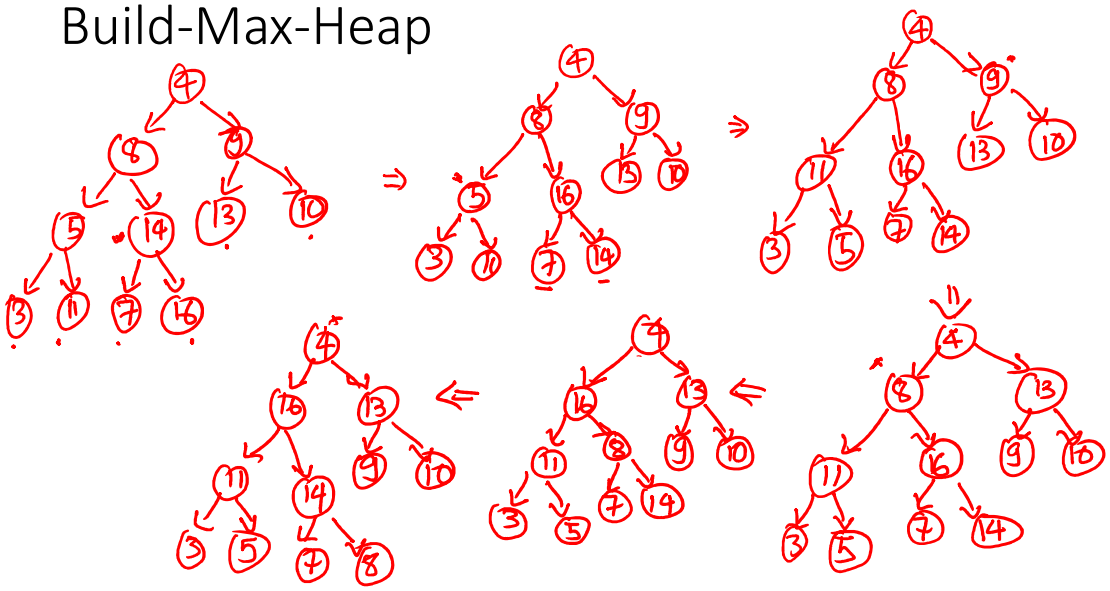
Can we use the Max_Heapify operation in a way that the final tree is a max heap?



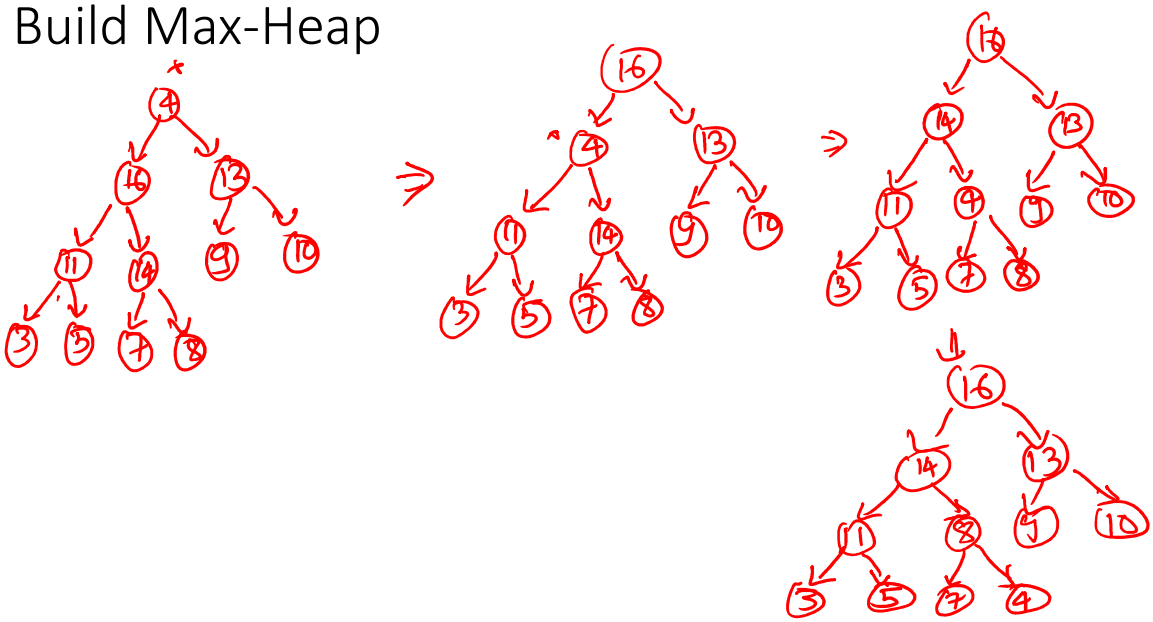
4 8 9 5 14 13 10 3 11 7 16

To build the max heap from a nearly complete binary tree of height h, we first apply the Max_Heapify operations on all the nodes at level h-1 that have at least one child, followed by Max_Hapify operations at all the nodes at level h-2, followed by Max_Heapify operations at all the nodes at level h-3, and so on, until we reach the root node.

Build-Max-Heap



Build Max-Heap



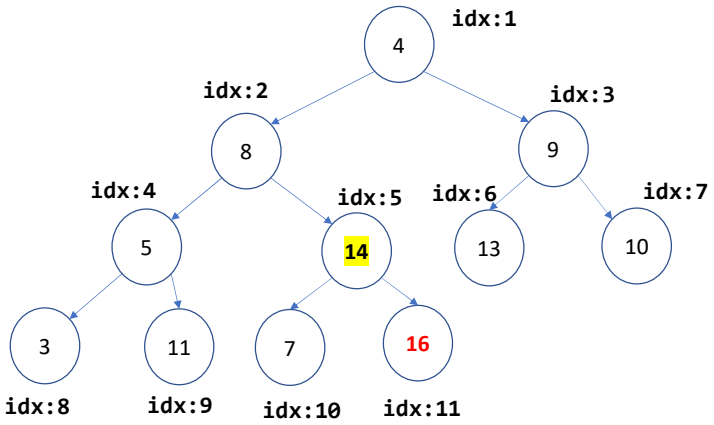
Build-Max-Heap

- The leaf nodes are already max heaps

for i = $\left\lfloor \frac{n}{2} \right\rfloor$ **downto** 1

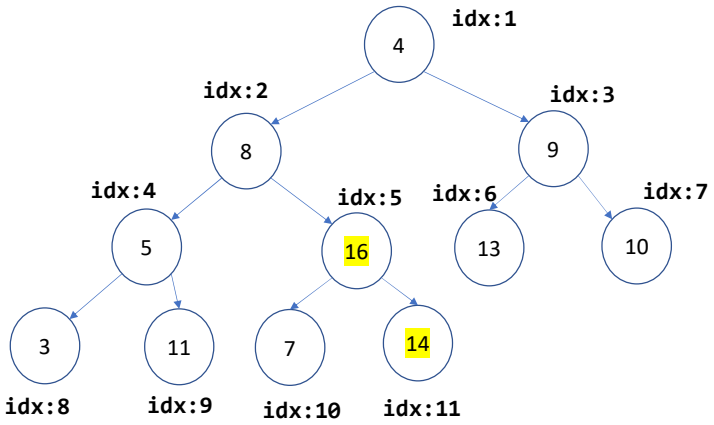
Max_Heapify(H, i)

Build-Max-Heap



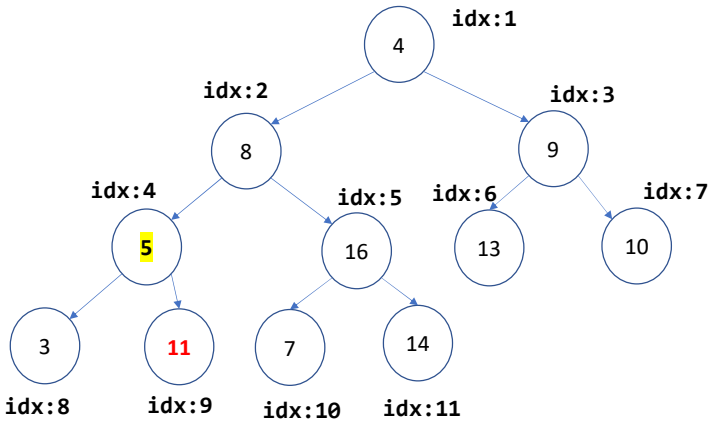
heapify idx - 5

Build-Max-Heap



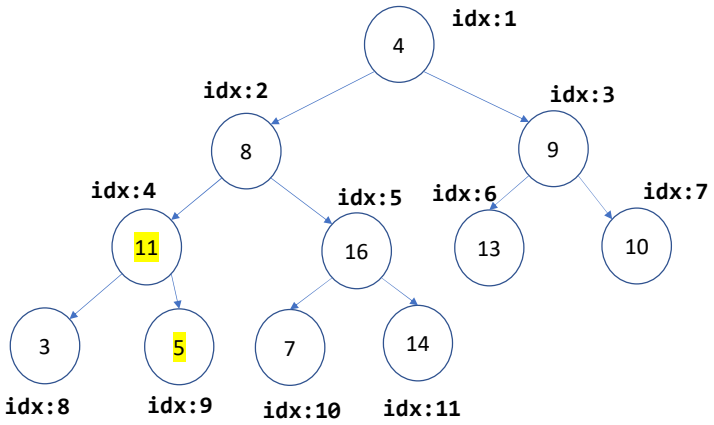
heapify idx - 5

Build-Max-Heap



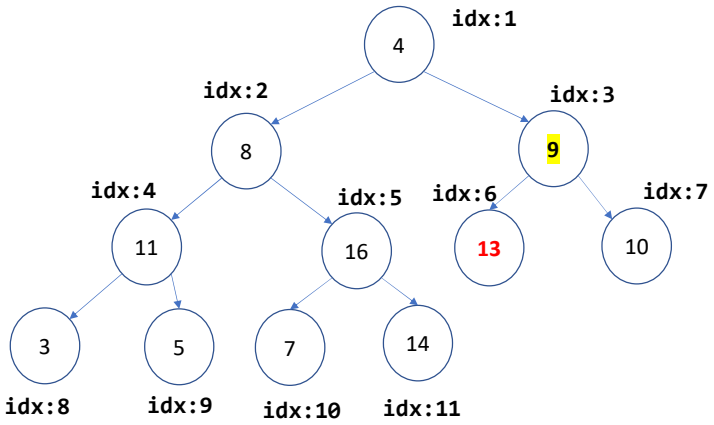
heapify idx – 5
heapify idx – 4

Build-Max-Heap



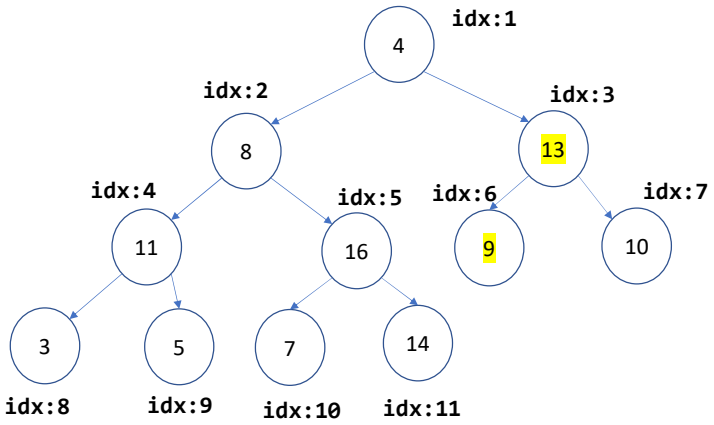
heapify idx – 5
heapify idx – 4

Build-Max-Heap



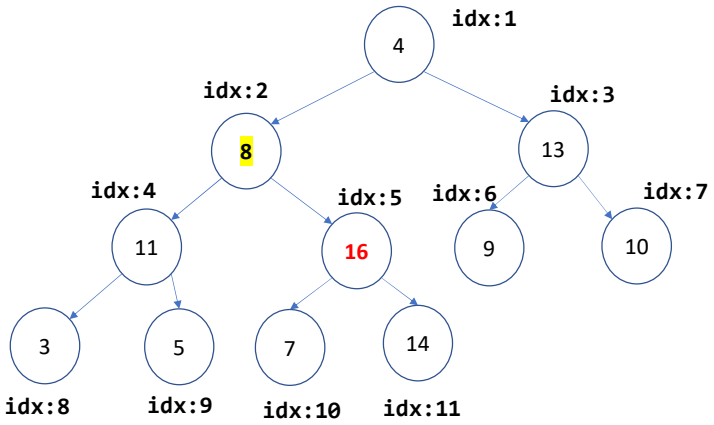
heapify idx – 5
heapify idx – 4
heapify idx – 3

Build-Max-Heap



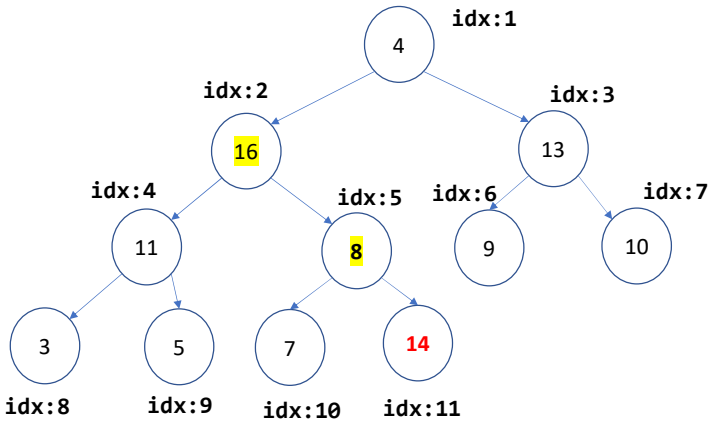

```
heapify idx – 5
heapify idx – 4
heapify idx – 3
```

Build-Max-Heap



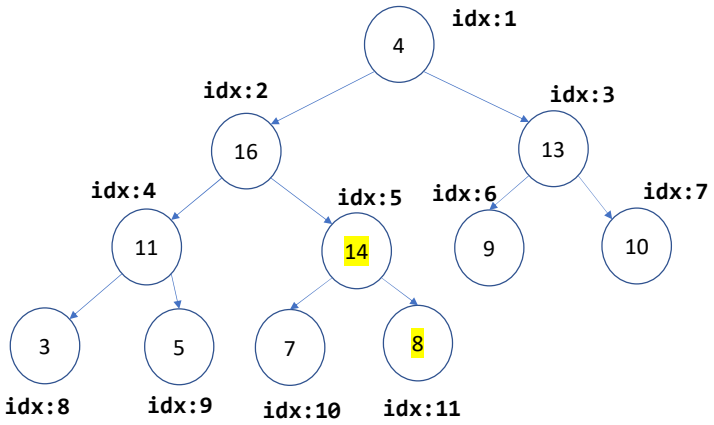
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2

Build-Max-Heap



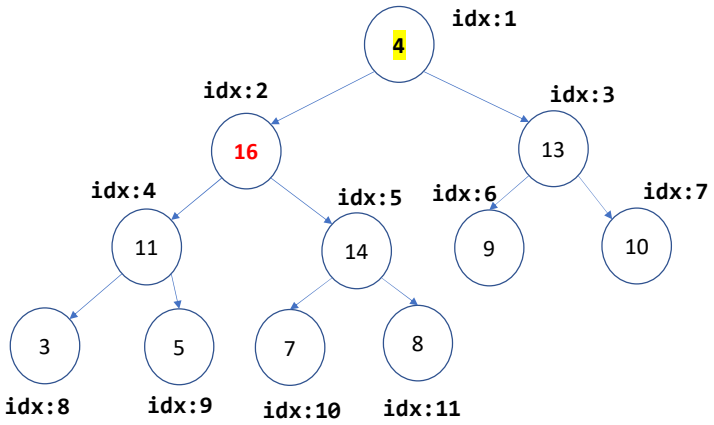
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2

Build-Max-Heap



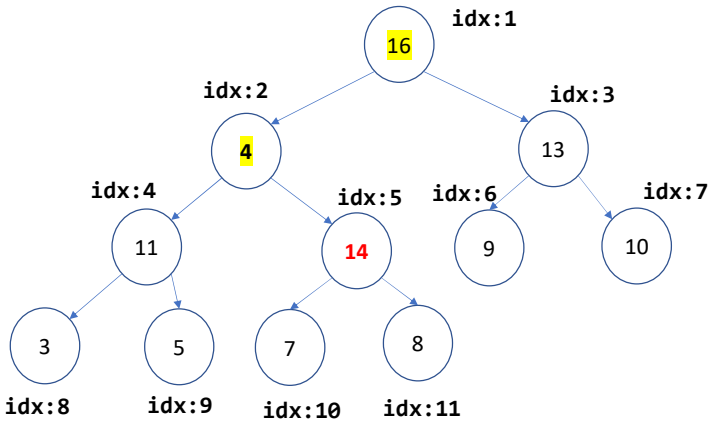
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2

Build-Max-Heap



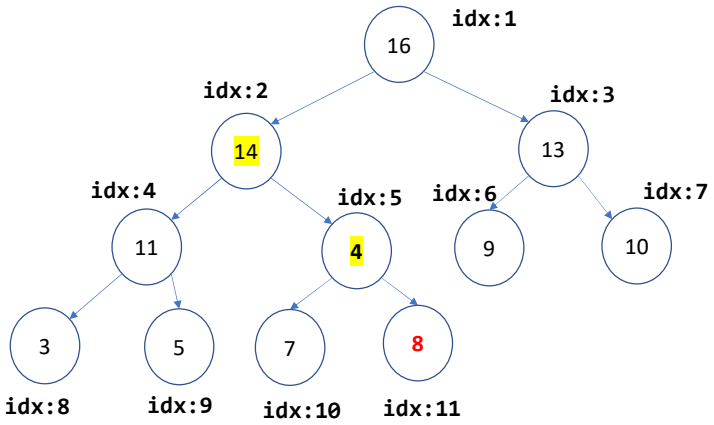

```
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2
heapify idx - 1
```

Build-Max-Heap



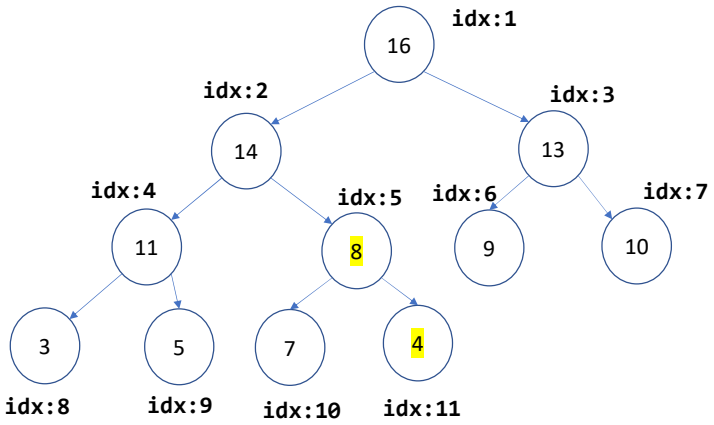

```
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2
heapify idx - 1
```

Build-Max-Heap




```
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2
heapify idx - 1
```

Build-Max-Heap




```
heapify idx – 5
heapify idx – 4
heapify idx – 3
heapify idx – 2
heapify idx - 1
```

Build-Max-Heap time complexity

$$T(n) = c\left(2^{h-1}\times 1 + 2^{h-2}\times 2 + 2^{h-3}\times 3 + \cdots + 2^{0}\times h\right)$$

$$= c\times 2^{h}\left(\frac{1}{2} + \frac{2}{2^{2}} + \frac{3}{2^{3}} + \cdots + \frac{h}{2^{h}}\right)$$

$$= c\times 2^{h}\sum_{i=1}^{h}\frac{i}{2^{i}}$$

$$2^{h} \le n \le 2^{h+1}-1$$

$$\sum_{i=1}^{h}\frac{i}{2^{i}} < 2$$

$$T(n) < c\times 2^{h}\times 2 < c\times 2^{h+1}$$

$$T(n) = O(n)$$

The Max_Heapify operation at index i takes O(h) operations, where h is the height of the subtree rooted at index i. Therefore, for some constant c, the Max_Heapify operations at level h-1 take c * 1 operations for each node. The Max_Heapify operations at level h-2 take c * 2 operations for each node. The Max_Heapify operations at level h-3 take c * 3 operations for each node, and so on. Finally, the Max-Heapify operation will take c * h operations for the node at height 0. In a nearly complete binary tree, the number of nodes at level h-1 is 2^(h-1), the number of elements at level h-2 is 2^(h-2), and so on. The number of elements at level 0 is 1. Plugging all these values, we get an equation for the time complexity as listed on this slide.

Build-Max-Heap time complexity

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \text{if } x < 1$$

$$\sum_{i=0}^{\infty} i x^{i-1} = \frac{d(1-x)^{-1}}{d(1-x)} * \frac{d(1-x)}{dx} = \frac{-1}{(1-x)^2} \times (-1)$$

$$\sum_{i=0}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$$

$$\sum_{i=0}^{\infty} i x^{i} = \frac{x}{(1-x)^2}$$

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2 \quad \Rightarrow \quad \sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

Build-Max-Heap time complexity

$$T(n) \leq c * (2^{h-1} * 1 + 2^{h-2} * 2 + 2^{h-3} * 3 + \ \ldots + 2^0 * h)$$
$$\leq c * 2^h \left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \ \ldots . + \frac{h}{2^h}\right)$$
$$\leq c * 2^h \sum_{i=1}^{h} \frac{i}{2^i}$$

$\sum_{i=1}^{h} \frac{i}{2^i} < 2$

$\mathrm{T(n)} < c * 2^{h+1}$

$2^h \leq n \leq 2^{h+1} - 1$
$T(n) = O(n)$

Build-Max-Heap time complexity

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \qquad where\ x < 1$$

Differentiating both sides with respect to x

$$\sum_{i=0}^{\infty} ix^{i-1} = \frac{d(1-x)^{-1}}{d(1-x)} * \frac{d(1-x)}{d(x)} = (-1) * \frac{1}{(1-x)^2} * (-1) = \frac{1}{(1-x)^2}$$

$\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$

Substituting x = 1/2, we get

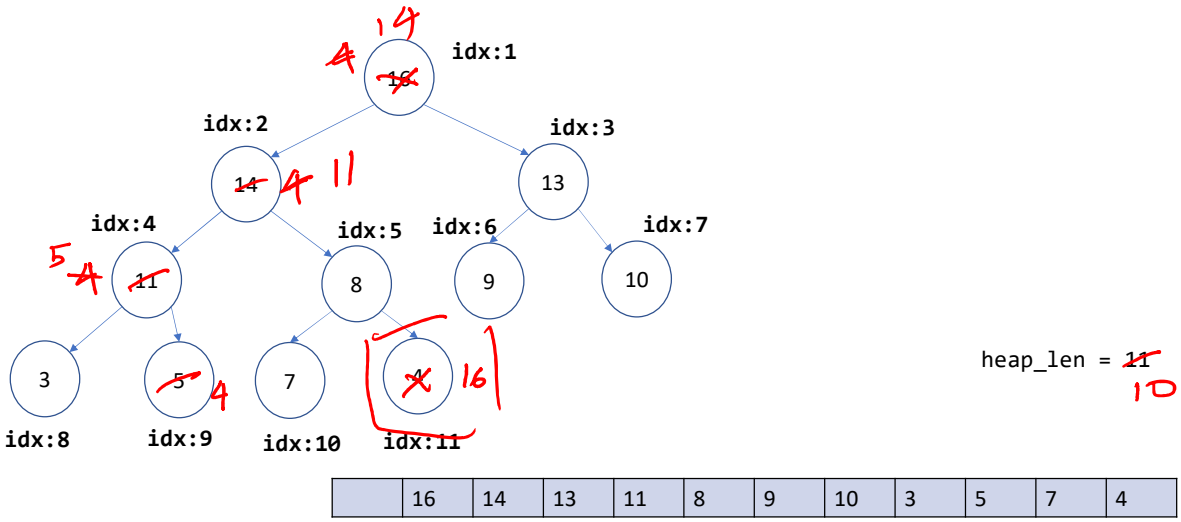
$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

Max-Heap-Extract-Max

Max-Heap-Extract-Max

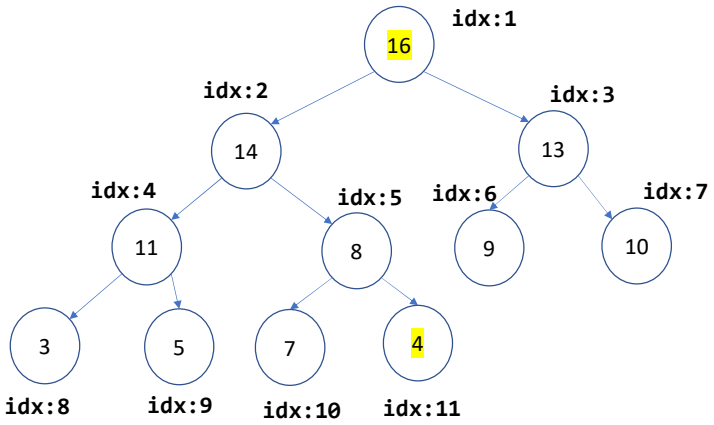
- The goal is to remove and return the maximum key

Max-Heap-Extract-Max



To delete the root and return its value, we can first save the value of the node at index 1 in some variable. After that, we copy the value of the rightmost leaf at the last level to the node at index 1 and decrement the number of elements in the heap by one. This will make sure that the root node is deleted from the heap. But at this point, the heap at index 1 doesn't necessarily satisfy the max heap property even though the subtree rooted at its left and right children are the max heaps. Therefore, to enforce the max heap property at index 1 we perform the Max_Heapify operation at index 1.

Max-Heap-Extract-Max

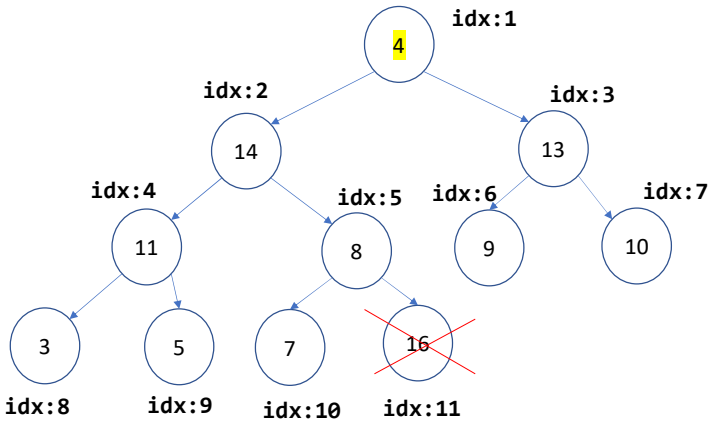


heap_len = 11

	16	14	13	11	8	9	10	3	5	7	4

Max-Heap-Extract-Max

```
replace arr[1] with arr[heap_len]
heap_len -= 1
```

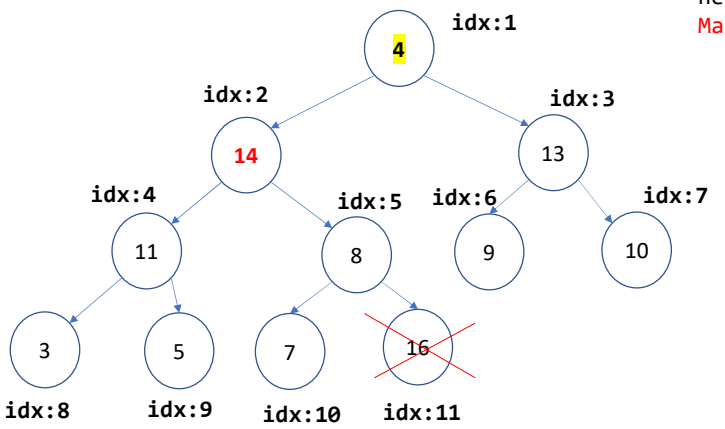



```
heap_len = 10
```

	4	14	13	11	8	9	10	3	5	7	16

Max-Heap-Extract-Max

```
replace arr[1] with arr[heap_len]
heap_len -= 1
Max_Heapify(H, 1)
```

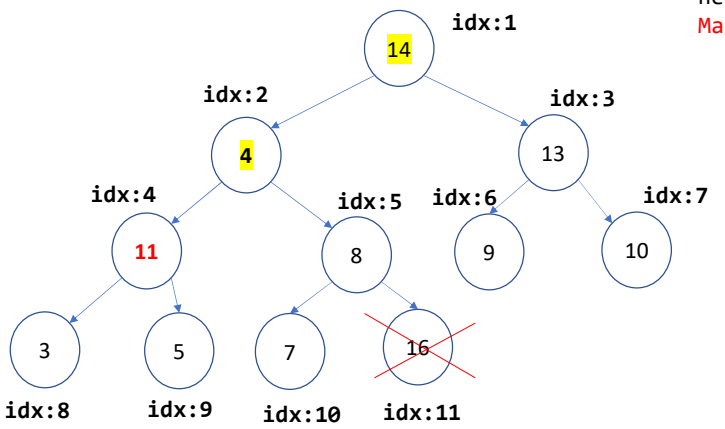



```
heap_len = 10
```

	4	14	13	11	8	9	10	3	5	7	16

Max-Heap-Extract-Max

```
replace arr[1] with arr[heap_len]
heap_len -= 1
Max_Heapify(H, 1)
```

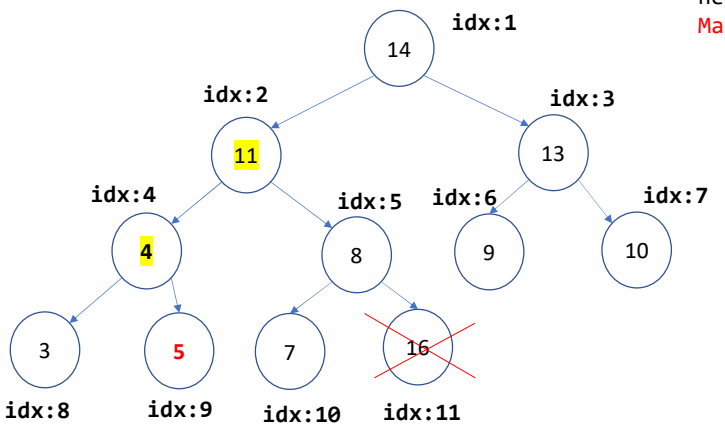



```
heap_len = 10
```

	14	4	13	11	8	9	10	3	5	7	16

Max-Heap-Extract-Max

```
replace arr[1] with arr[heap_len]
heap_len -= 1
Max_Heapify(H, 1)
```

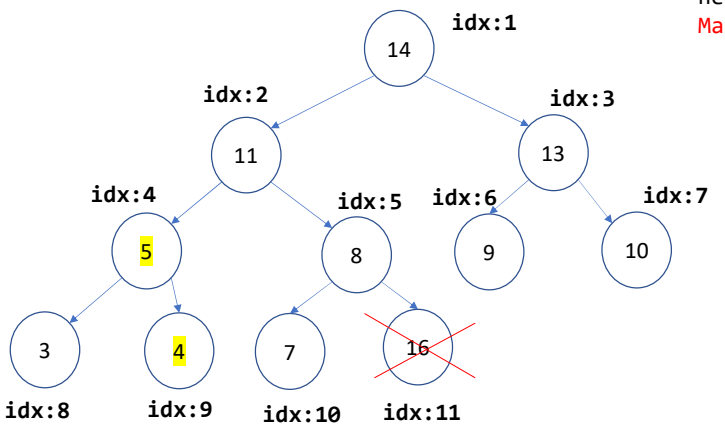



```
heap_len = 10
```

	14	11	13	4	8	9	10	3	5	7	16

Max-Heap-Extract-Max

```
replace arr[1] with arr[heap_len]
heap_len -= 1
Max_Heapify(H, 1)
```




```
heap_len = 10
```

	14	11	13	5	8	9	10	3	4	7	16

Max-Heap-Extract-Max

- Time complexity:

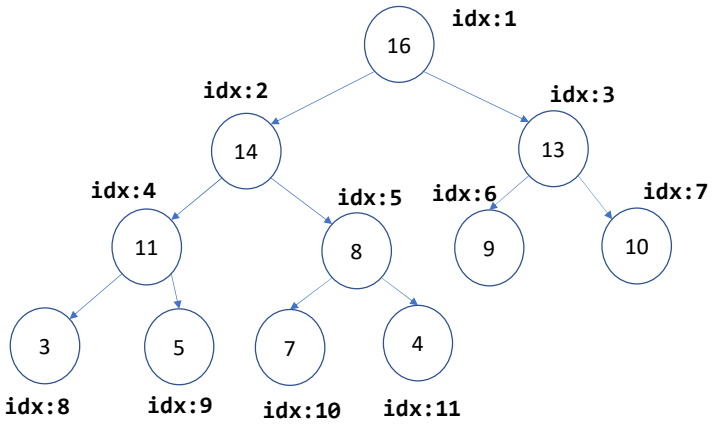
```
1.  struct Heap {
2.    int *arr;
3.    int capacity;    // maximum size of heap
4.    int heap_size;  // number of elements in the heap
5.  };

6.  Max_Heap_Max(H)
7.  // H is a reference to the heap of type struct Heap
8.  // Output: return the value of the maximum element
9.    if H->heap_size == 0
10.       perror("Heap underflow")
11.   return H->arr[1]

12. Max_Heap_Extract_Max(H)
13. // H is a reference to the heap of type struct Heap
14. // Output: delete the maximum element and return its value
15.   max = Max_Heap_Max(H)
16.   H->arr[1] = H->arr[H->heap_size]
17.   H->heap_size = H->heap_size – 1
18.   Max_Heapify(H, 1)
19.   return max
```

At line-15, we save the value of the root in the variable max. The Max_Heap_Max routine at line-6 returns the value of the root node if the heap is not empty. At line 16, we copy the value of the rightmost leaf at the last level to the node at index 1. At line-17, we are reducing the number of elements in the heap by one. At line-18, we are performing the Max_heapify operation at index 1.

Heapsort

Heap-Sort

How to implement heap sort using Max-Heap-Extract-Max.



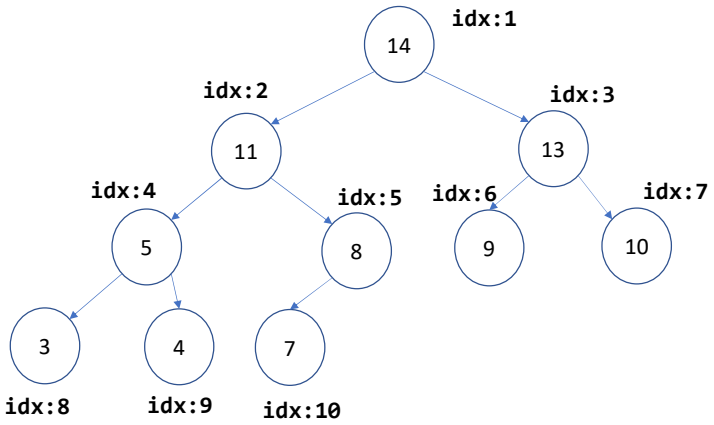
heap_len = 11

	16	14	13	11	8	9	10	3	5	7	4

Because the Max-Heap-Extract-Max always deletes the maximum element, we can use it to delete the maximum element n-1 times. Instead of copying the value of the rightmost leaf at the last level (say node at index j) to index 1, we can swap the value at index 1 with the value at index j. Notice that index j is not part of the heap after the deletion, but it is still a valid array index. After deleting the n-1 element in this manner, the array contains the element in a sorted, increasing order.

Heap-Sort

Max-Heap-Extract-Max(H)

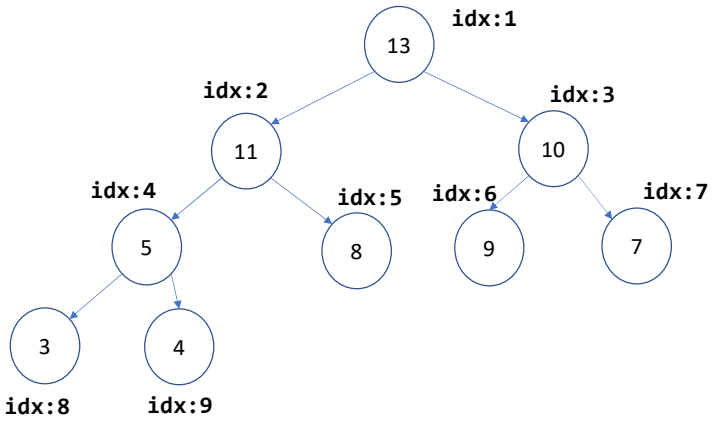


heap_len = 10

	14	11	13	5	8	9	10	3	4	7	16

Heap-Sort

Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)

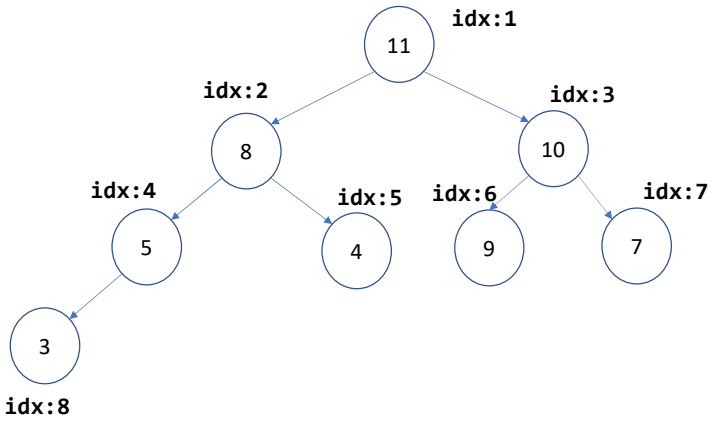


heap_len = 9

	13	11	10	5	8	9	7	3	4	14	16

Heap-Sort

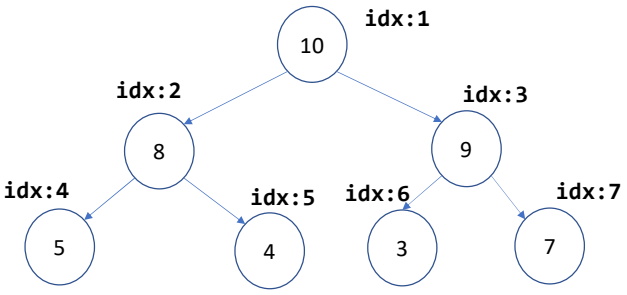
```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```




```
heap_len = 8
```

	11	8	10	5	4	9	7	3	13	14	16

Heap-Sort

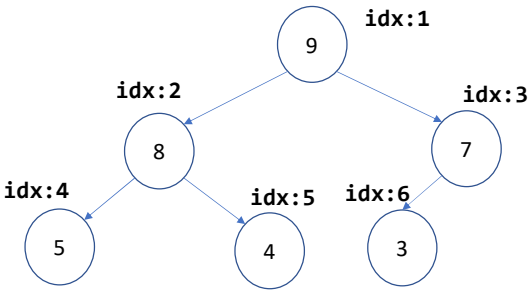



```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 7
```

	10	8	9	5	4	3	7	11	13	14	16

Heap-Sort

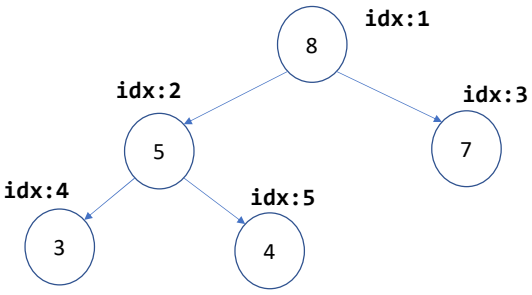



```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 6
```

	9	8	7	5	4	3	10	11	13	14	16

Heap-Sort

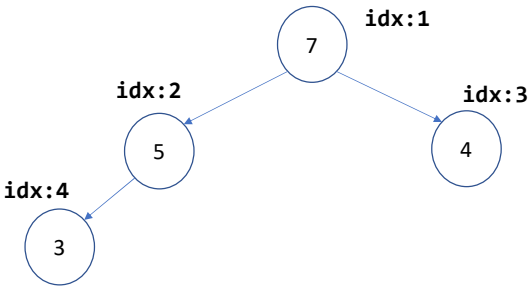



```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 5
```

	8	5	7	3	4	9	10	11	13	14	16

Heap-Sort

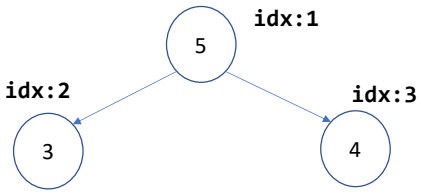



```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 4
```

	7	5	4	3	8	9	10	11	13	14	16

Heap-Sort

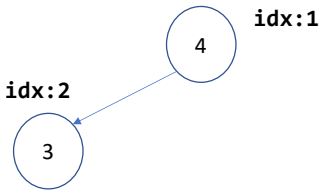



```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 3
```

	5	3	4	7	8	9	10	11	13	14	16

Heap-Sort




```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 2
```

	4	3	5	7	8	9	10	11	13	14	16

Heap-Sort

idx:1

```
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
Max-Heap-Extract-Max(H)
```

```
heap_len = 1
```

	3	4	5	7	8	9	10	11	13	14	16

Heapsort

- Build a max heap from the array elements $O(n)$
- Perform the Max-Heap-Extract-Max operation n-1 times $n \log n$
- Time complexity: $n \log n$

Building max heap

- Option-1
  - Insert n elements in the array
  - Use the Build_Max_Heap algorithm on the unordered array
- Option-2
  - Insert n elements one by one starting from an empty heap

Building max heap

- Why does Build-Max-Heap take O(n) time, whereas inserting n elements take O(nlog n) time?

Building max heap

- Why does Build-Max-Heap take O(n) time, whereas inserting n elements take O(nlog n) time?
  - Except for the last level, the number of elements at a given level is twice the number of elements at the preceding level
  - Per-node number of operations increases for a higher level in the Insert-Max-Heap algorithm
  - Per-node number of operations decreases for a higher level in the Build-Max-Heap algorithm
  - There are roughly (n/2) leaf nodes for which the insertion takes O(log n) operations
  - Build-Max-Heap doesn't require any operations on the leaf node

Exercise

- Max_Heap_Increase_Key(H, i, keyval): Increase key at index i to a new value (keyval) in the max heap H