上海科技大学
ShanghaiTech University

Algorithms and Data Structures Discussion 4 (Week 6)

Designed By: Keyi Yuan
Teaching Assistant
Oct.14th 2019

1. Binary Tree

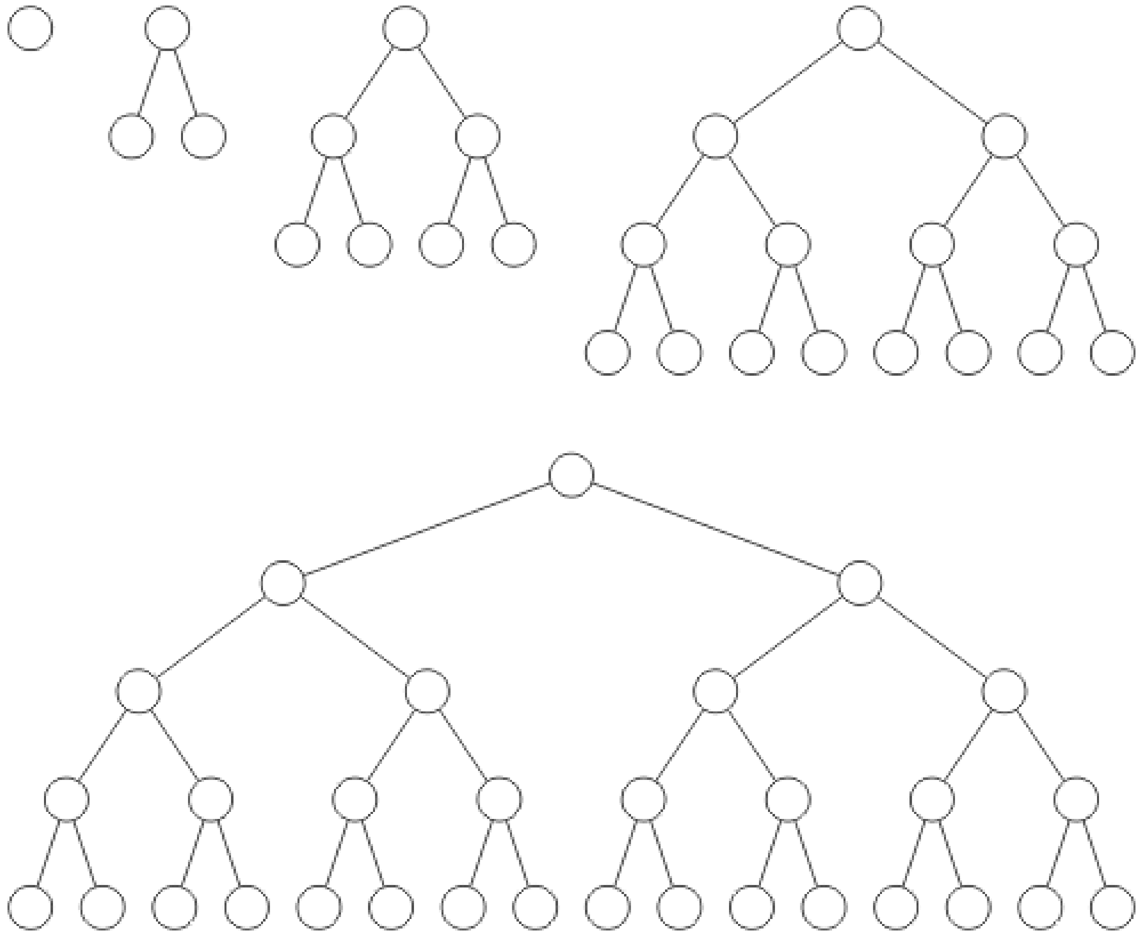
1-1. Perfect Binary Tree

1-1. Perfect Binary Tree

- A perfect binary tree of height h is a binary tree where:
  - all leaf nodes have the same depth, *h*, and
  - all other nodes are full nodes.
- A <u>recursive</u> definition of a perfect binary tree is:
  - A single node with no children is a perfect binary tree of height $h = 0$,
  - A perfect binary tree with height $h > 0$ is a node where both sub-trees are non-overlapping perfect binary trees of height $h - 1$.

1-1. Perfect Binary Tree

- Perfect binary trees of height $h = 0, 1, 2, 3$ and 4



1-1. Perfect Binary Tree Properties

- Four properties of perfect binary trees:
  - A perfect binary tree of height h has $2^{h+1} - 1$ nodes, which means a perfect binary tree with n nodes has height $\lg(n + 1) - 1$.
  - The height is $\Theta(\ln(n))$
  - There are 2^h leaf nodes -> *You should know how to prove it*.
  - The average depth of a node is $\Theta(\ln(n))$

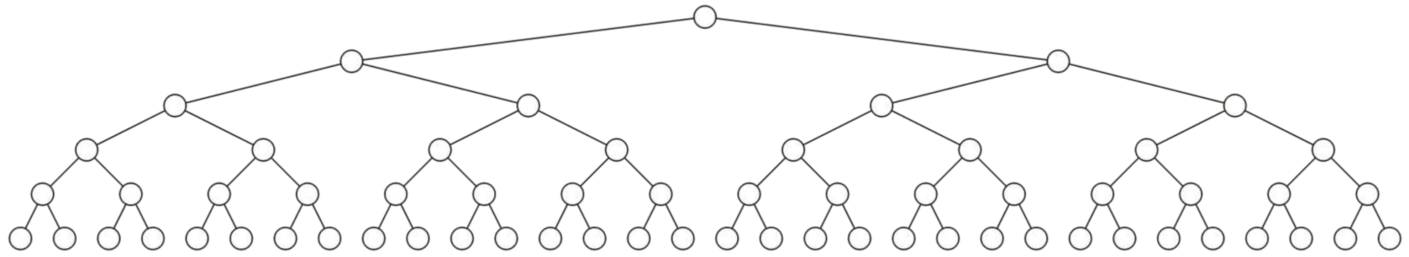
- These theorems will allow us to determine the optimal run-time properties of operations on binary trees.

1-1. Perfect Binary Tree - 2^h Leaf Nodes

- Theorem:
  - A perfect binary tree with height h has 2^h leaf nodes.
- Proof (by induction):
  - When $h = 0$, there is $2^0 = 1$ leaf node.
  - Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^h leaf nodes.
- Consequence: Over half of the nodes are leaf nodes:

1-1. The Average Depth of a Node

- The average depth of a node in a perfect binary tree is:



Depth	Count
0	1
1	2
2	4
3	8
4	16
5	32

Sum of the depths

Number of nodes

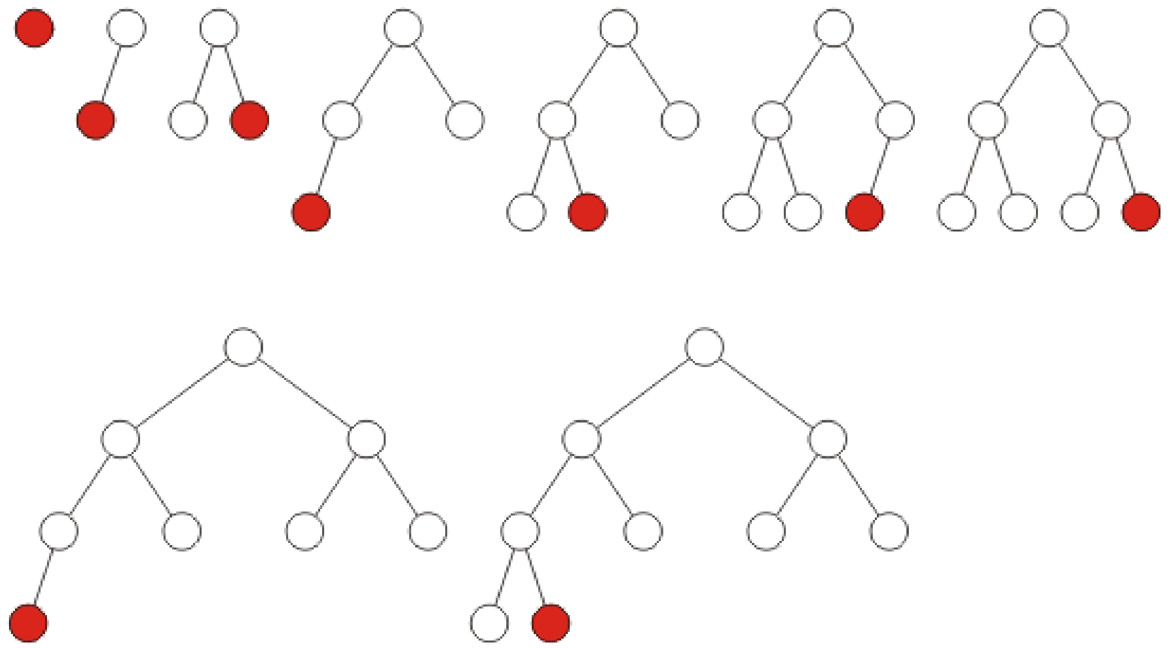
$$\frac{\sum_{k=0}^{h} k2^k}{2^{h+1}-1} = \frac{h2^{h+1}-2^{h+1}+2}{2^{h+1}-1} = \frac{h(2^{h+1}-1)-(2^{h+1}-1)+1+h}{2^{h+1}-1}$$

$$= h-1+\frac{h+1}{2^{h+1}-1} \approx h-1 = \Theta\left(\ln(n)\right)$$

1-2. Complete Binary Tree

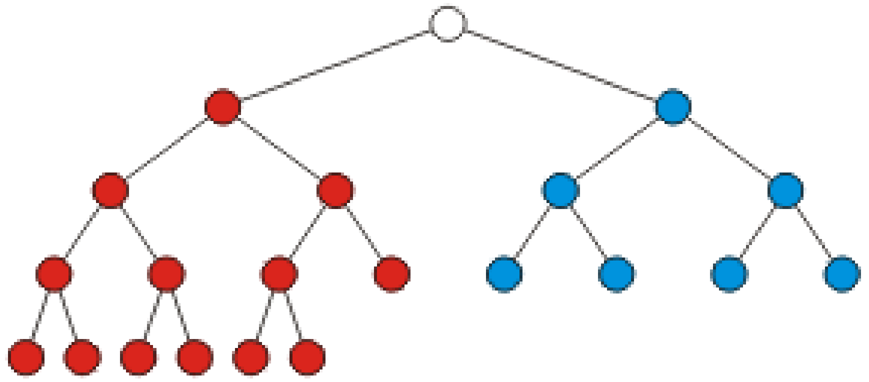
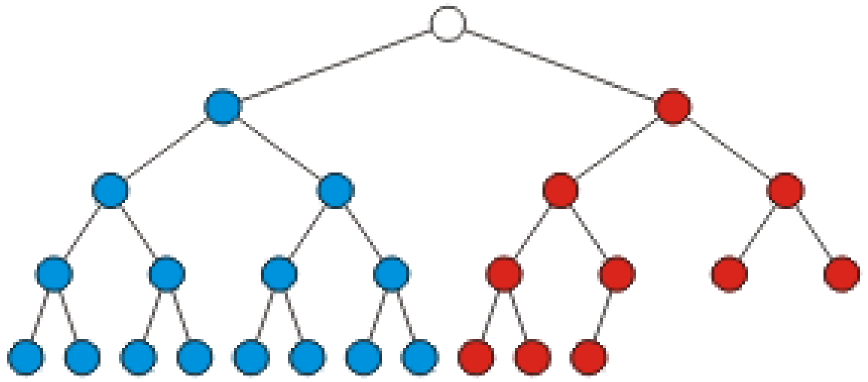
1-2. Complete Binary Tree

- A complete binary tree filled at each depth from left to right
  - Identical order to that of a breadth-first traversal



1-2. Complete Binary Tree

- Recursive definition: a binary tree with a single node is a complete binary tree of height $h = 0$ and a complete binary tree of height h is a tree where either:
  - The left sub-tree is a **complete tree** of height $h - 1$ and the right sub-tree is a **perfect tree** of height $h - 2$, or
  - The left sub-tree is **perfect tree** with height $h - 1$ and the right sub-tree is **complete tree** with height $h - 1$

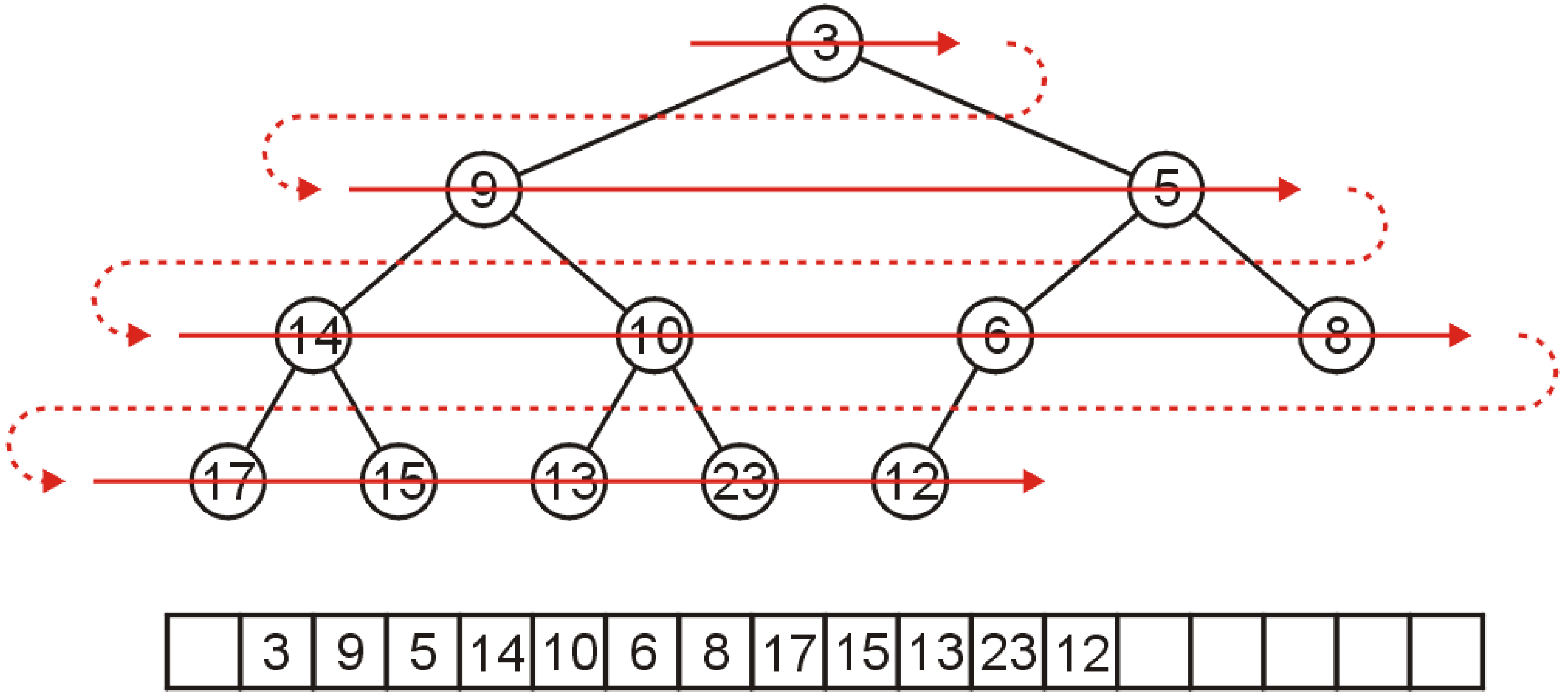



1-2. Complete Binary Tree Properties

- Properties of complete binary tree:
  - The height of a complete binary tree with n nodes is $h = \lfloor \lg(n) \rfloor$
- Proof (By induction):
  - Base case:
    - When $n = 1$ then $\lfloor \lg(1) \rfloor = 0$ and a tree with one node is a complete tree with height $h = 0$
  - Inductive step:
    - Assume that a complete tree with n nodes has height $\lfloor \lg(n) \rfloor$
    - Must show that $\lfloor \lg(n + 1) \rfloor$ gives the height of a complete tree with $n + 1$ nodes
    - Two cases:
      - If the tree with n nodes is perfect, and
      - If the tree with n nodes is complete but not perfect

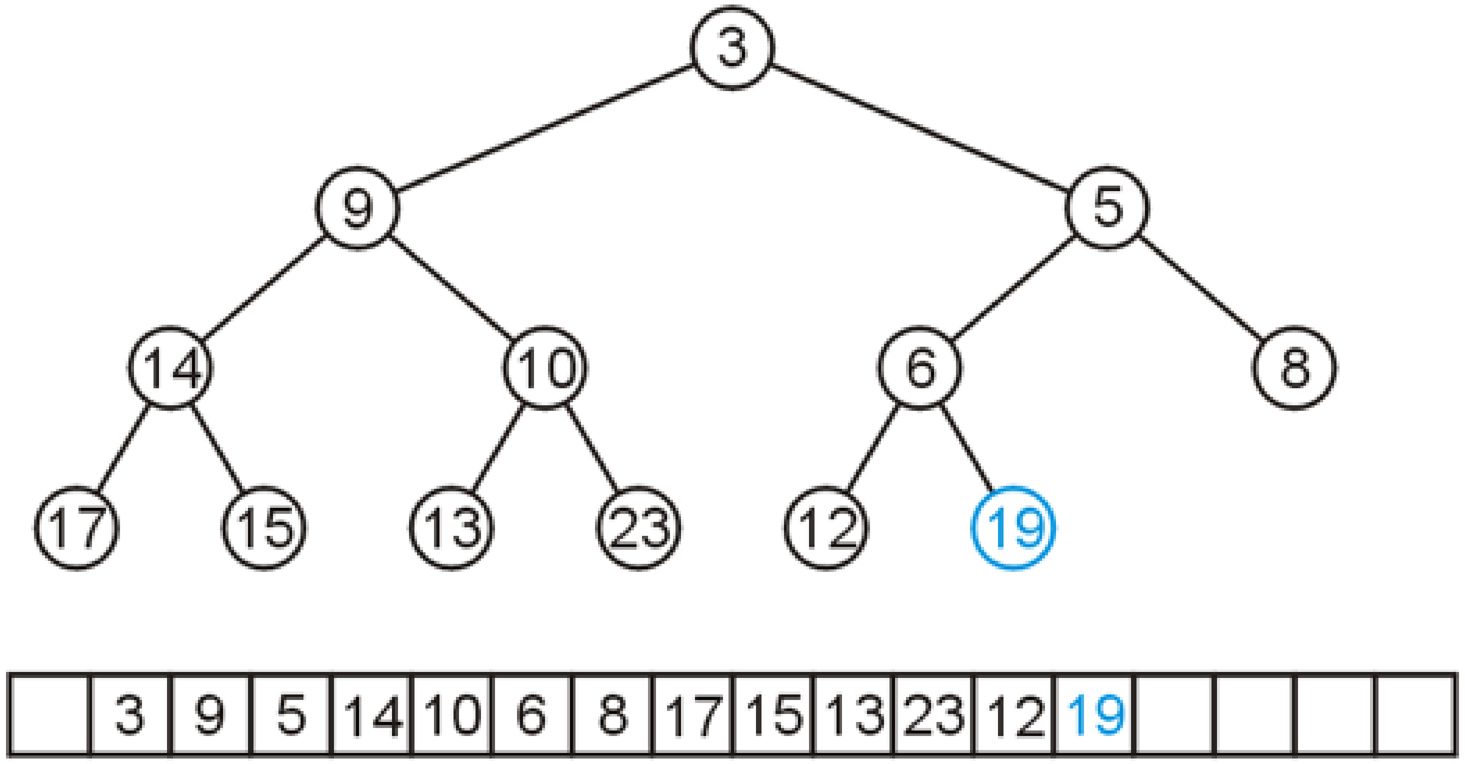
1-2. Complete Binary Tree – Array Storage

- We are able to store a complete tree as an array
  - Traverse the tree in breadth-first order, placing the entries into the array.



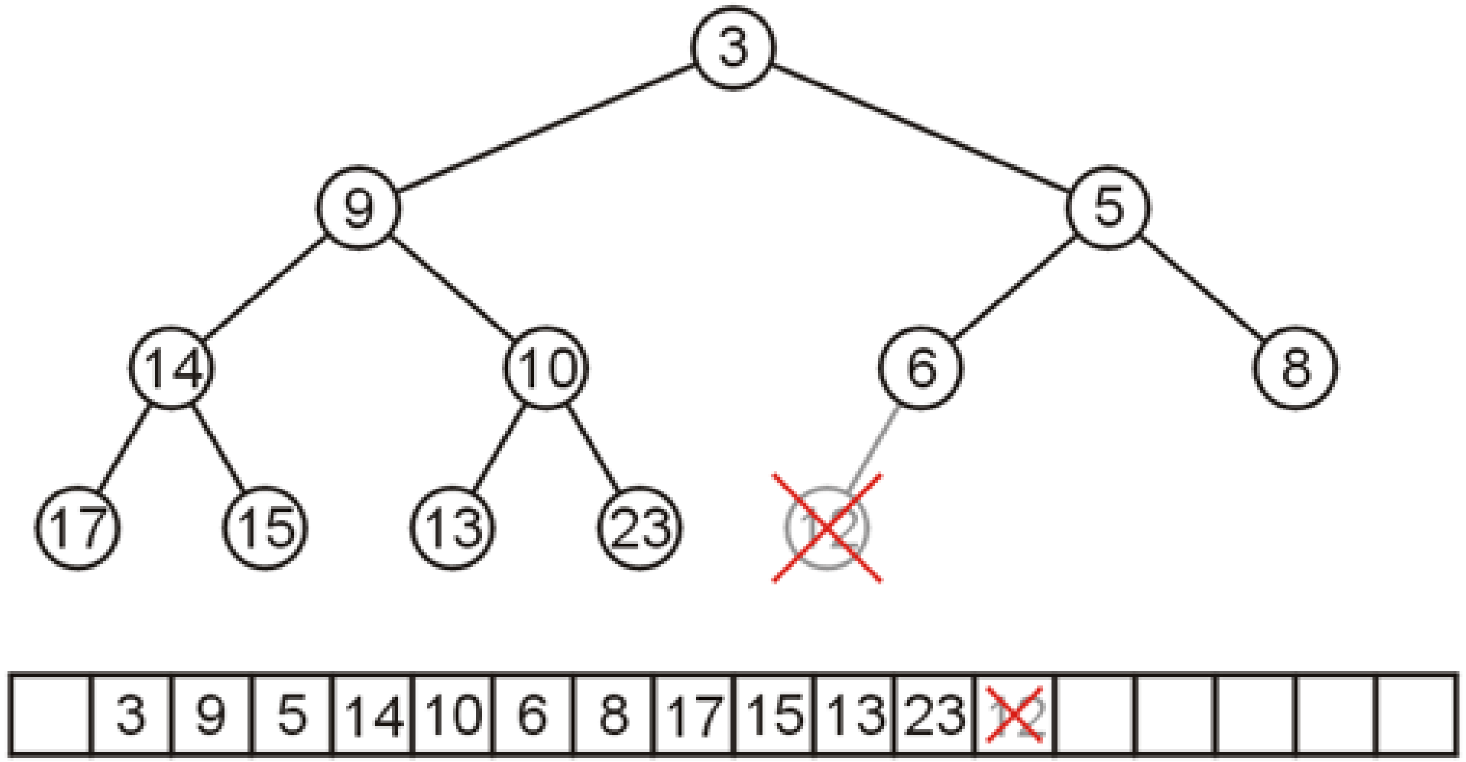
1-2. Complete Binary Tree – Array Storage

- To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location.



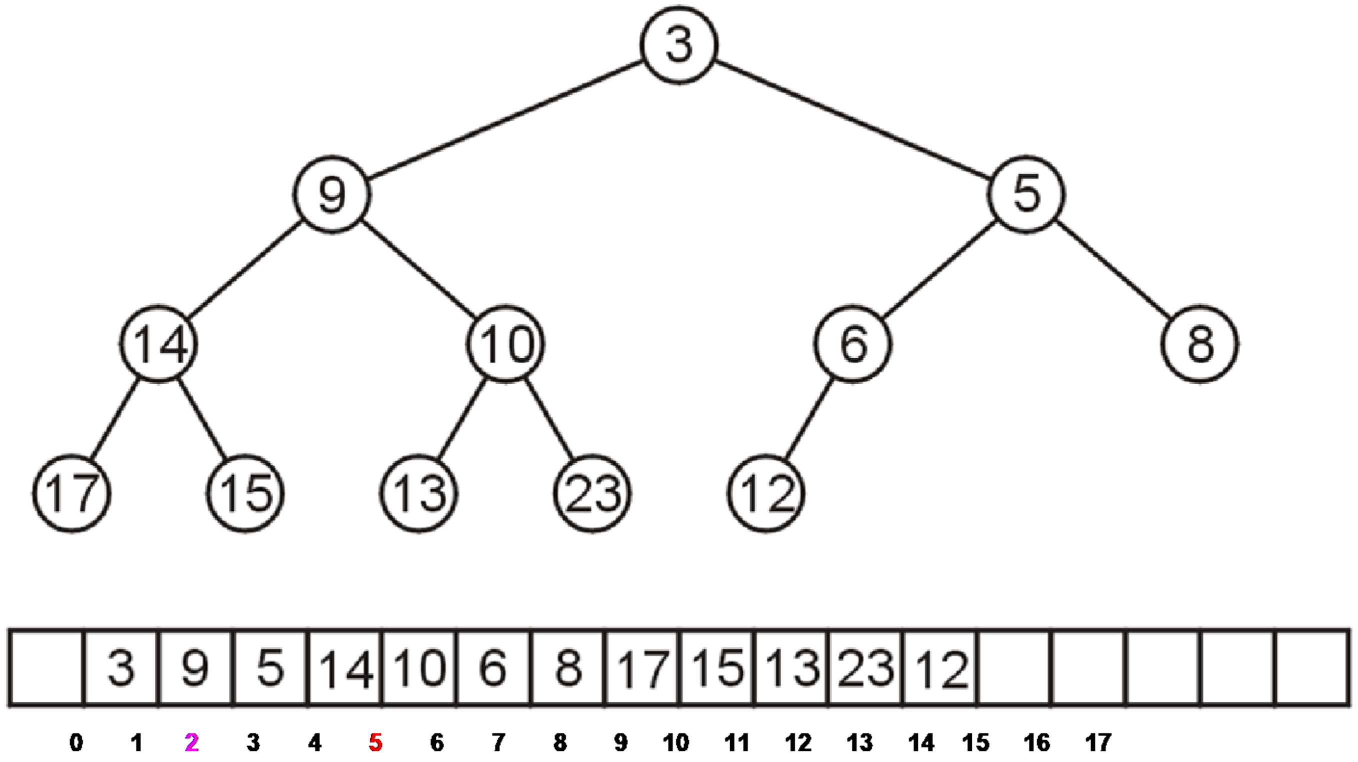
1-2. Complete Binary Tree – Array Storage

- To remove a node while keeping the complete-tree structure, we must remove the last element in the array.



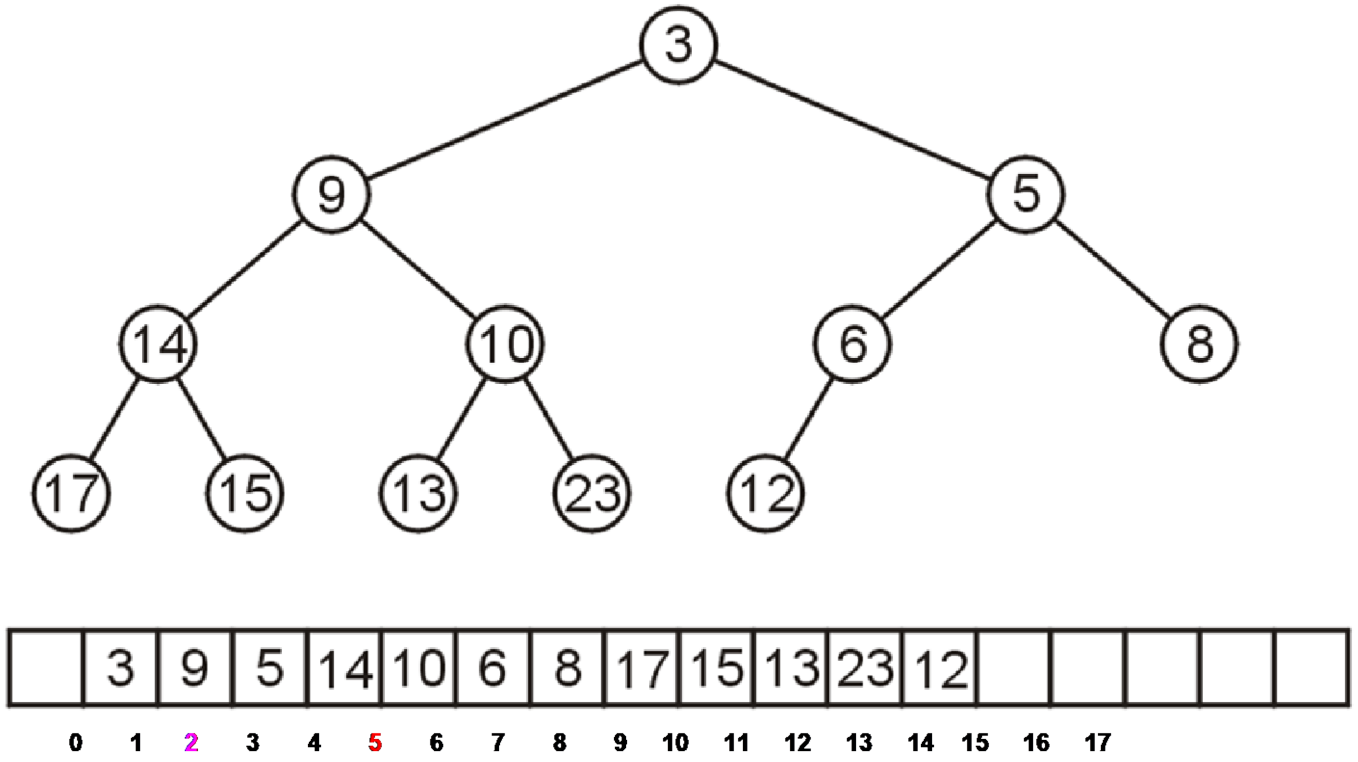
1-2. Complete Binary Tree – Array Storage

- Leaving the first entry blank yields a bonus:
  - The children of the node with index k are in $2k$ and $2k+1$
  - The parent of node with index k is in $k/2$



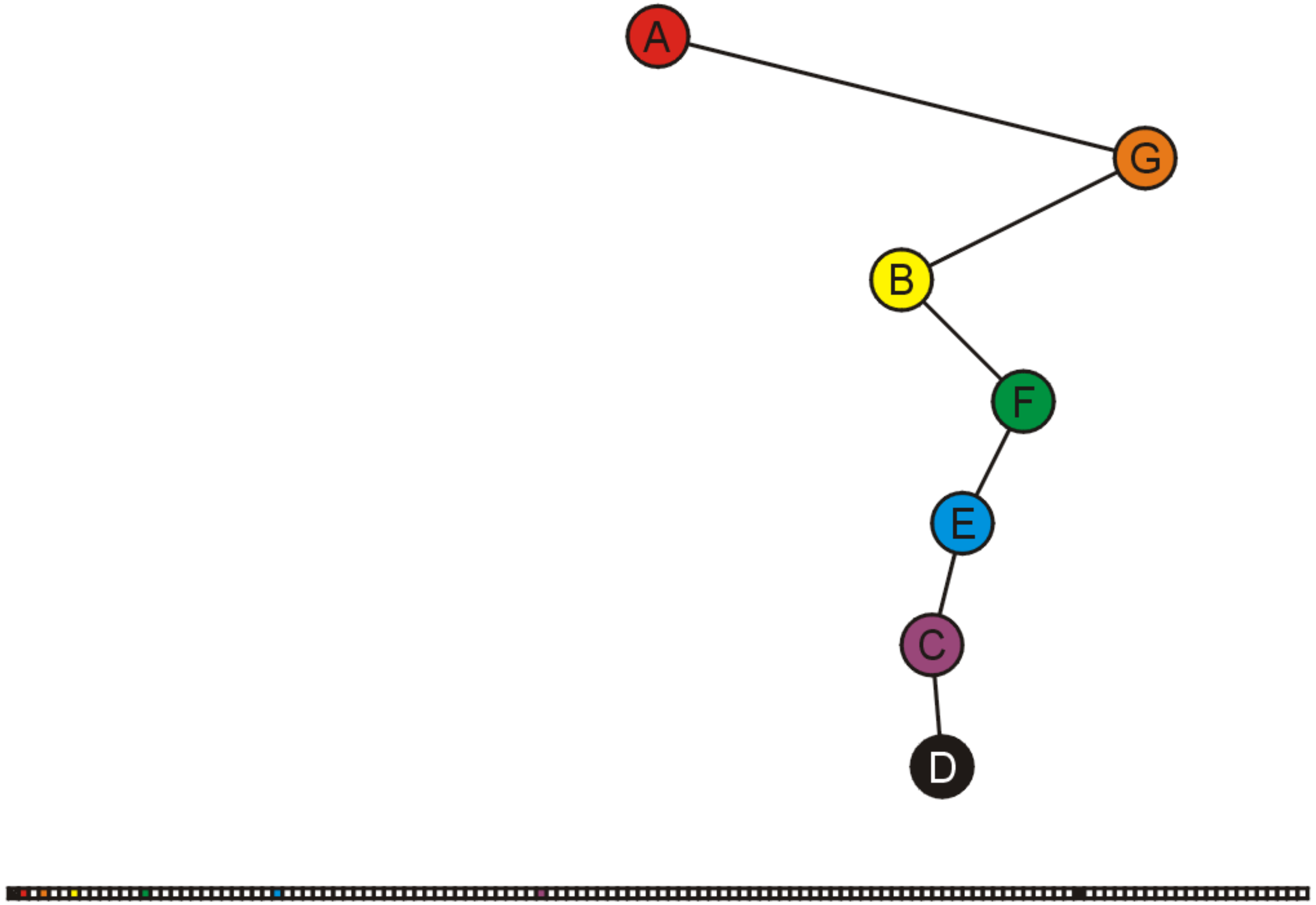
1-2. Complete Binary Tree – Array Storage

- For example, node 10 has index 5:
  - Its children 13 and 23 have indices 10 and 11, respectively
  - Its parent is node 9 with index 5/2 = 2



1-2. Array Storage?

- For any tree, is it a better choice to choose array to store a tree?



1-3. Full Binary Tree

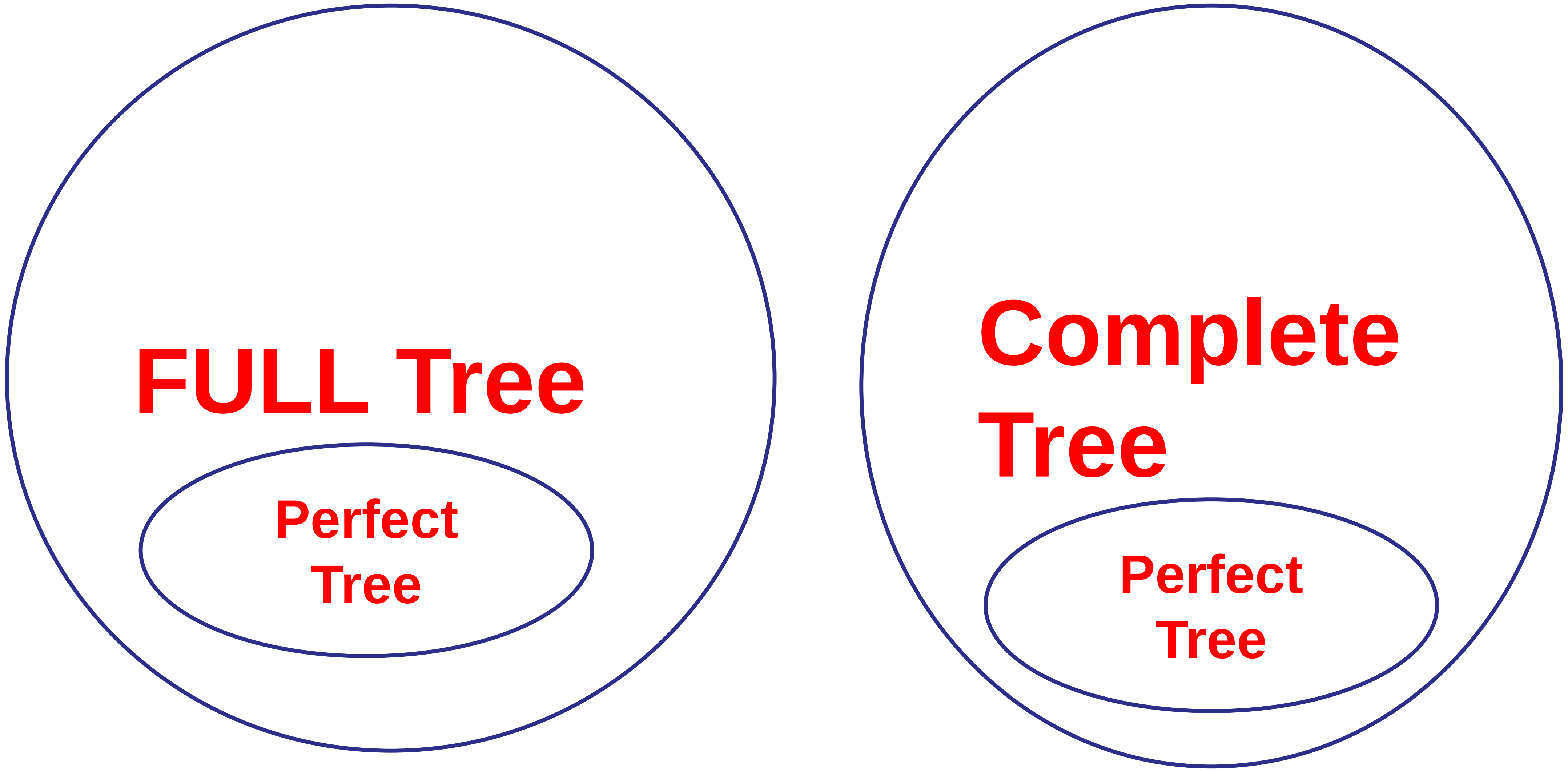
1-3. Full Binary Tree

- A binary tree T is full if each node is either a leaf or possesses exactly two child nodes.

Summary 1

- Binary tree
  - Each node has two children
  - In-order traversal
- Perfect binary tree
  - Number of nodes, height, number of leaf nodes, average depth
- Complete binary tree
  - Height, array storage

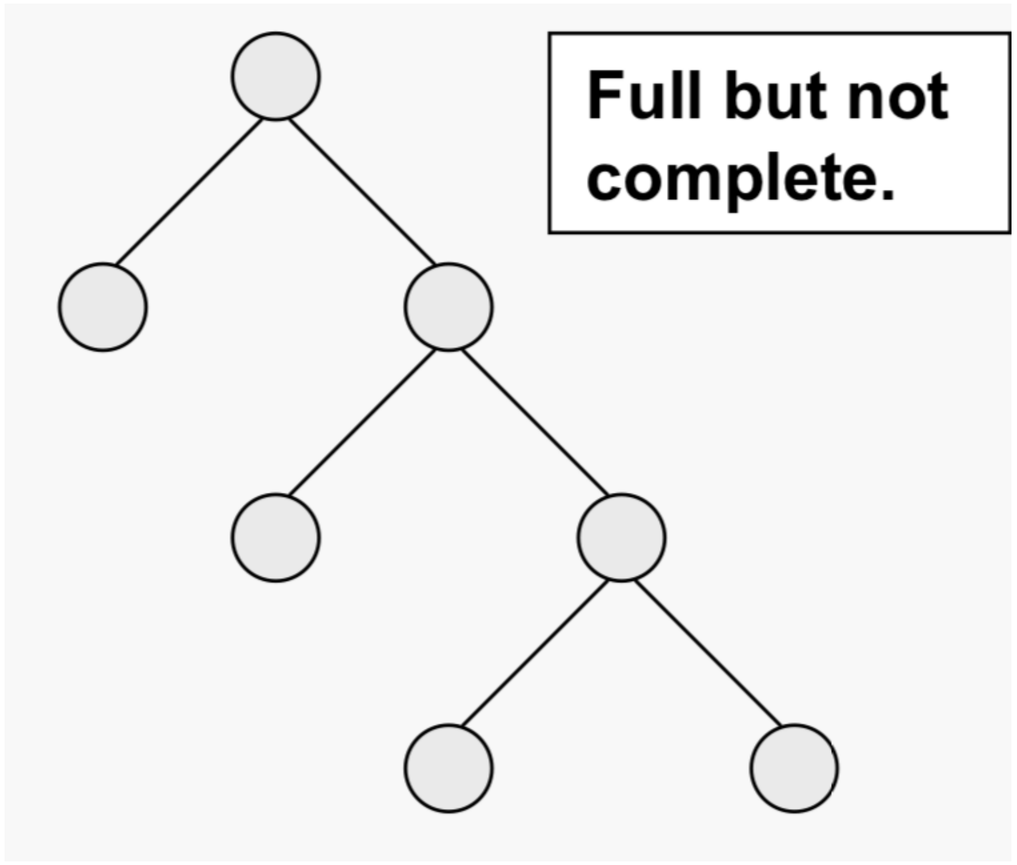
Summary 1



1-4. Exercise Part 1

Exercise 1 - Definition Distinguish

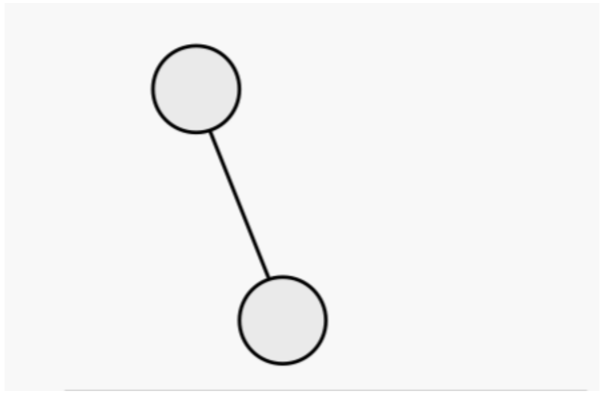
- Judge if it is a perfect, complete or full binary tree.



- Not perfect
- Not complete
- Full

Exercise 1 - Definition Distinguish

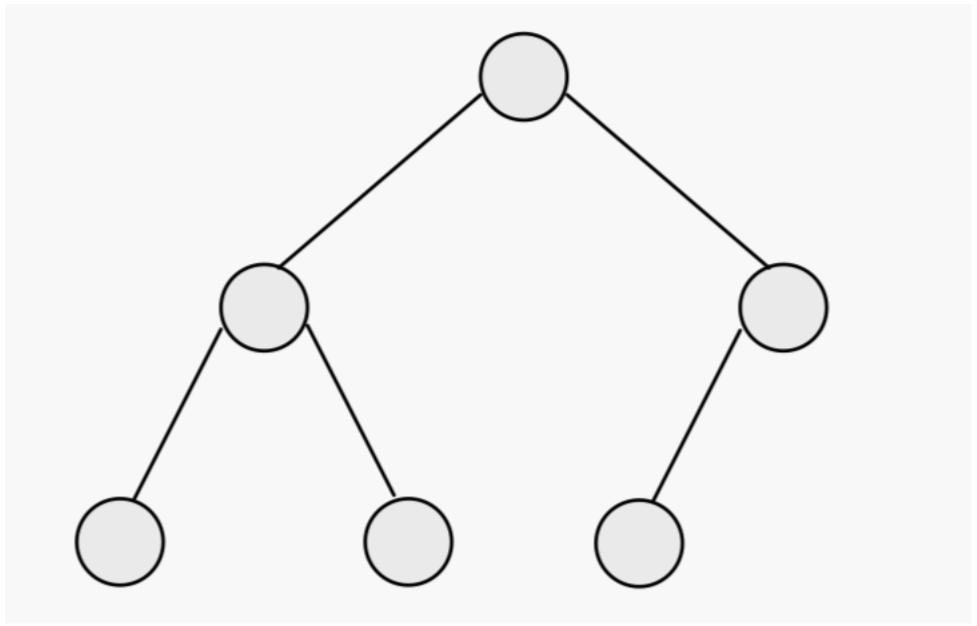
- Judge if it is a perfect, complete or full binary tree.



- Not perfect
- Not complete
- Not full

Exercise 1 - Definition Distinguish

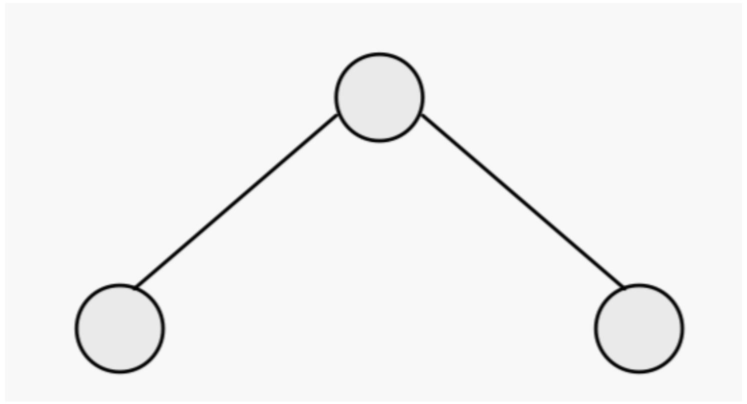
- Judge if it is a perfect, complete or full binary tree.



- Not perfect
- Complete
- Not full

Exercise 1 - Definition Distinguish

- Judge if it is a perfect, complete or full binary tree.



- Perfect
- Complete
- Full

Exercise 2 – Node Number Calculation

- How many internal nodes are there in a full binary tree with 500 leaves?

- *Adapt from ShanghaiTech University, GEE, 2019.*GEE = Graduate Entrance Examination*

Exercise 2 – Node Number Calculation

- How many internal nodes are there in a full binary tree with 500 leaves?

- Solution:
- Take any two leaves and combine them to create an internal node. Now, you can increase by one the number of internal nodes and delete the two used leaves, which transforms than internal node in a new leaf.
- Thus, if we call $f(n)$ the number of internal nodes with n leaves, the previous argument leads us to $f(n) = 1 + f(n-1)$, where $f(2) = 1$. Therefore, $f(n) = n - 1$.
- $f(500) = 500 - 1 = 499$

Exercise 2 – Node Number Calculation

- How many internal nodes are there in a full binary tree with 500 leaves?
- Key:
  - The definition of leaf and full tree.
  - Find the relationship between the property of the tree.

Exercise 3 – Given Orders to Get Another Orders

- Recall:
  - Pre-order: Root -> Left -> Right
  - In-order: Left -> Root -> Right
  - Post-order: Left -> Right -> Root

Exercise 3 – Given Orders to Get Another Orders

- Given the post-order and in-order traversal of a binary tree, print out the pre-order of a tree.
  - In-order: {4,2,5,1,3,6}
  - Post-order: {4,5,2,6,3,1}

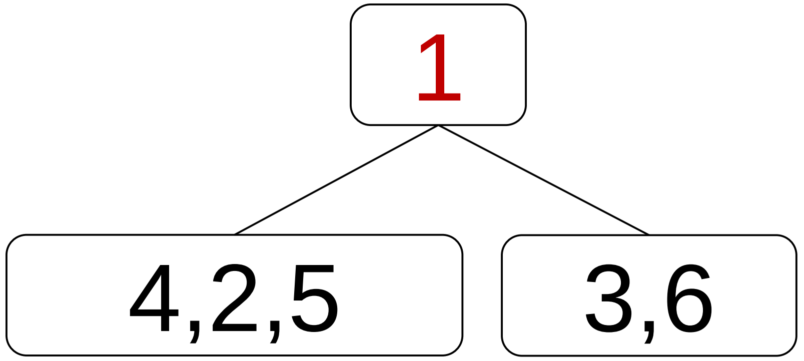
Exercise 3 – Given Orders to Get Another Orders

- *Step1: Find the root*
  - In-order: {4,2,5,**1**,3,6}
    - {4,2,5} is the left subtree, while {3,6} is the right subtree.
  - Post-order: {4,5,2,6,3,**1**} ← *Root of the tree must be at last one of post-order*

- *Step2: Reconstruct root-level subtree*
- *Step3: Recursively apply that process to the left&right subtree the until the subtree includes only a node.*

Exercise 3 – Given Orders to Get Another Orders

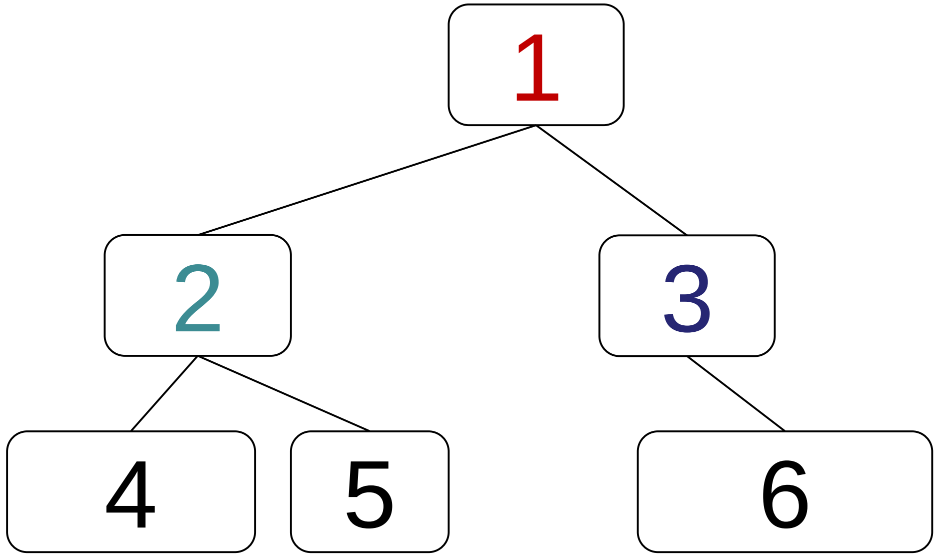
- *Step1: Find the root*
  - In-order: {4,2,5,**1**,3,6}
  - Post-order: {4,5,2,6,3,**1**}



- *Step2: Reconstruct root-level subtree*
- *Step3: Recursively apply that process to the left&right subtree the until the subtree includes only a node.*

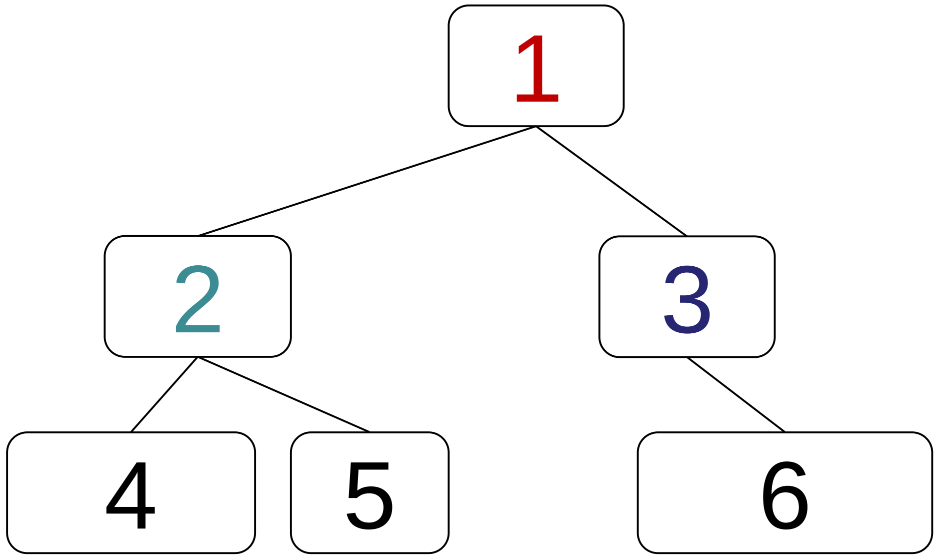
Exercise 3 – Given Orders to Get Another Orders

- *Step1: Find the root*
  - In-order: {4,**2**,5,**1**,**3**,6}
  - Post-order: {4,5,**2**,6,**3**,**1**}
    - {2} is the root of the left subtree.
    - {3} is the root of the right subtree.
- *Step2: Reconstruct root-level subtree*
- *Step3: Recursively apply that process to the left&right subtree the until the subtree includes only a node.*



Exercise 3 – Given Orders to Get Another Orders

- *Step1: Find the root*
  - In-order: {4,**2**,5,**1**,**3**,6}
  - Post-order: {4,5,**2**,6,**3**,**1**}
    - {2} is the root of the left subtree.
    - {3} is the root of the right subtree.
  - **Notice: {6} is the right child of {3}. Why?**
  - Therefore, the pre-order is: {1,2,4,5,3,6}



Exercise 3 – Discussion

- Given pre-order/post-order and in-order, we can easily get post-order/pre-order respectively.

- How about the case given pre-order and post-order to get in-order? Why?

Exercise 4 – In-order Concept

- Consider a node X in a binary tree. Given that X has two children, let Y be in-order successor of X. Which of the following is true about Y?
  - (A) Y has no right child.
  - (B) Y has no left child.
  - (C) Y has both child.
  - (D) None of the above.

- *Adapt from ShanghaiTech University, CS101 Midterm Exam, 2018 Fall*

Exercise 4 – In-order Concept

- Consider a node X in a binary tree. Given that X has two children, let Y be in-order successor of X. Which of the following is true about Y?
  - (A) Y has no right child.
  - (B) Y has no left child.
  - (C) Y has both child.
  - (D) None of the above.

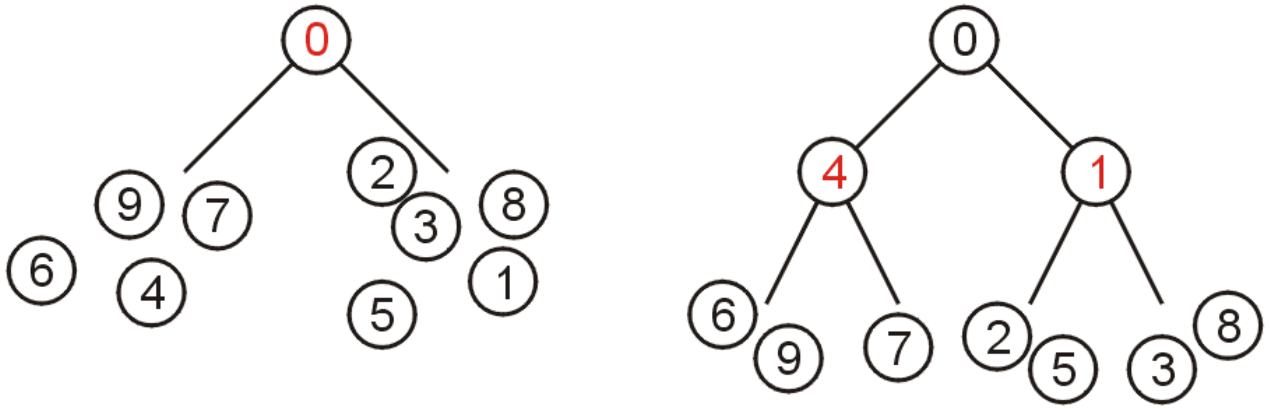
- Recall: Left -> Root -> Right
- X is the root of the subtree.

2. Heap and Heap Sort

2-1. Heap and Operations

2-1. Heap Definition

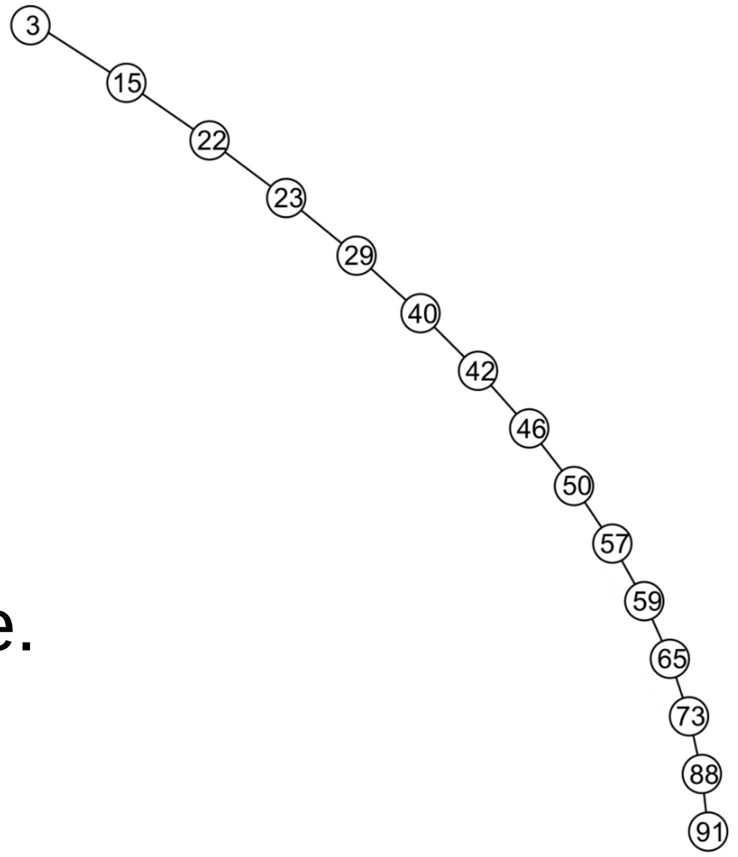
- A non-empty tree is a min-heap if
  - The key associated with the root is less than or equal to the keys associated with the sub-trees (if any).
  - The sub-trees (if any) are also min-heaps. (Recursive)



****There is no other relationship between the elements in the subtrees!*

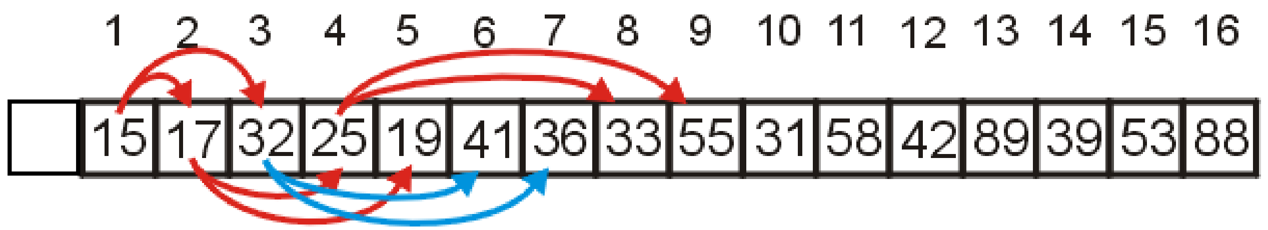
2-1. Heap Storage

- If we use a simple tree structure, then the time complexity may be bad.
  - $O(n)$
  - Worst case: the binary tree is highly unbalanced
- Can we do better?
  - Keep balance – a complete tree structure.



2-1. Heap Storage - Array

- We start at index 1 when filling the array. (like complete tree)



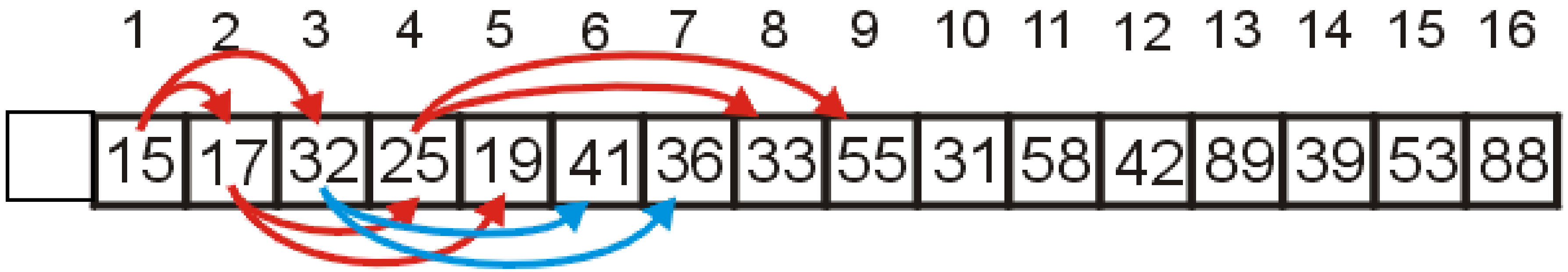
- Given the entry at index k, it follows that:
  - The parent of node is a $k/2$
  - The children are at $2k$ and $2k + 1$

```
parent = k >> 1;

left_child = k << 1;
 right_child = left_child + 1;
```

2-1. Heap Storage – Array Implementation

- If the heap-as-array has $\boldsymbol{count}$ entries, then the next empty node in the corresponding complete tree is at location $\boldsymbol{posn = count + 1}$
- We compare the item at location $\boldsymbol{posn}$ with the item at $\boldsymbol{\frac{posn}{2}}$
- If they are out of order
  - Swap them
  - Set **posn /= 2** and repeat



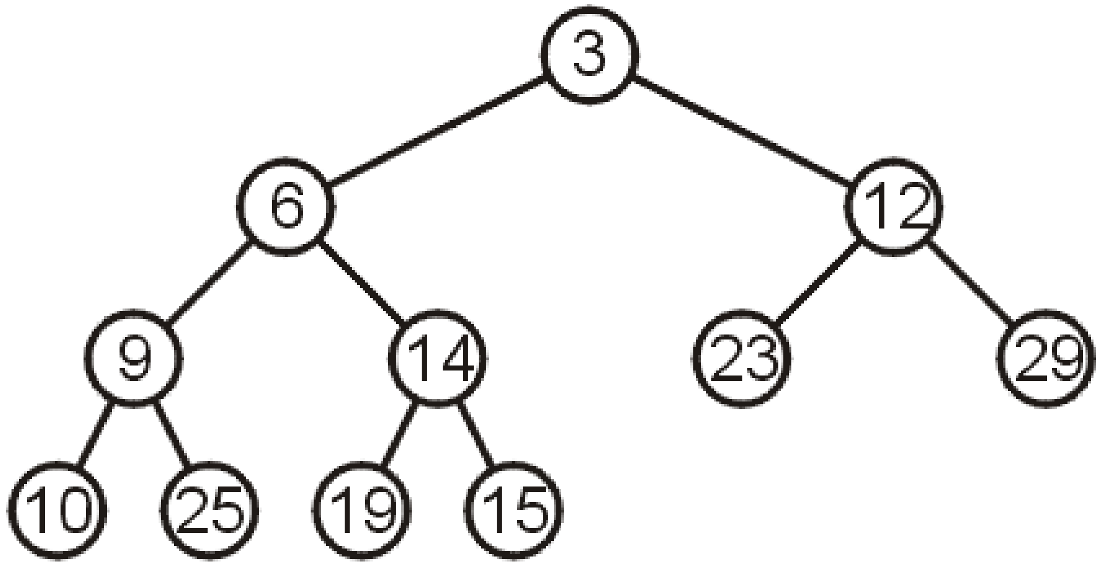
2-1. Heap Operations Guidelines

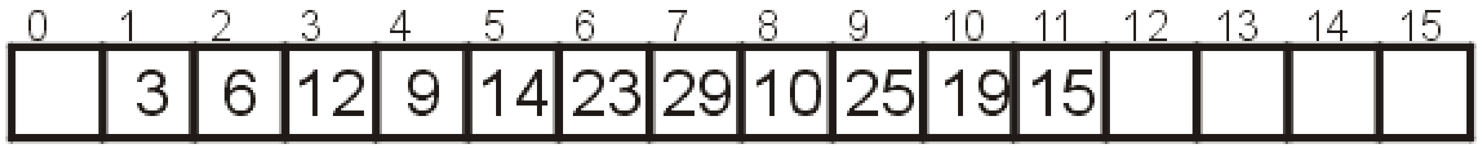
- We have such following operations on heap:
  - Push -> Based on Priority Queue
  - Pop -> Based on Priority Queue
  - Build Heap

- For each operation, the heap properties should always be satisfied.
- **`Heapify()`**

2-1. Heap Operations Example

- Consider the following heap, both as a tree and in its array representation.



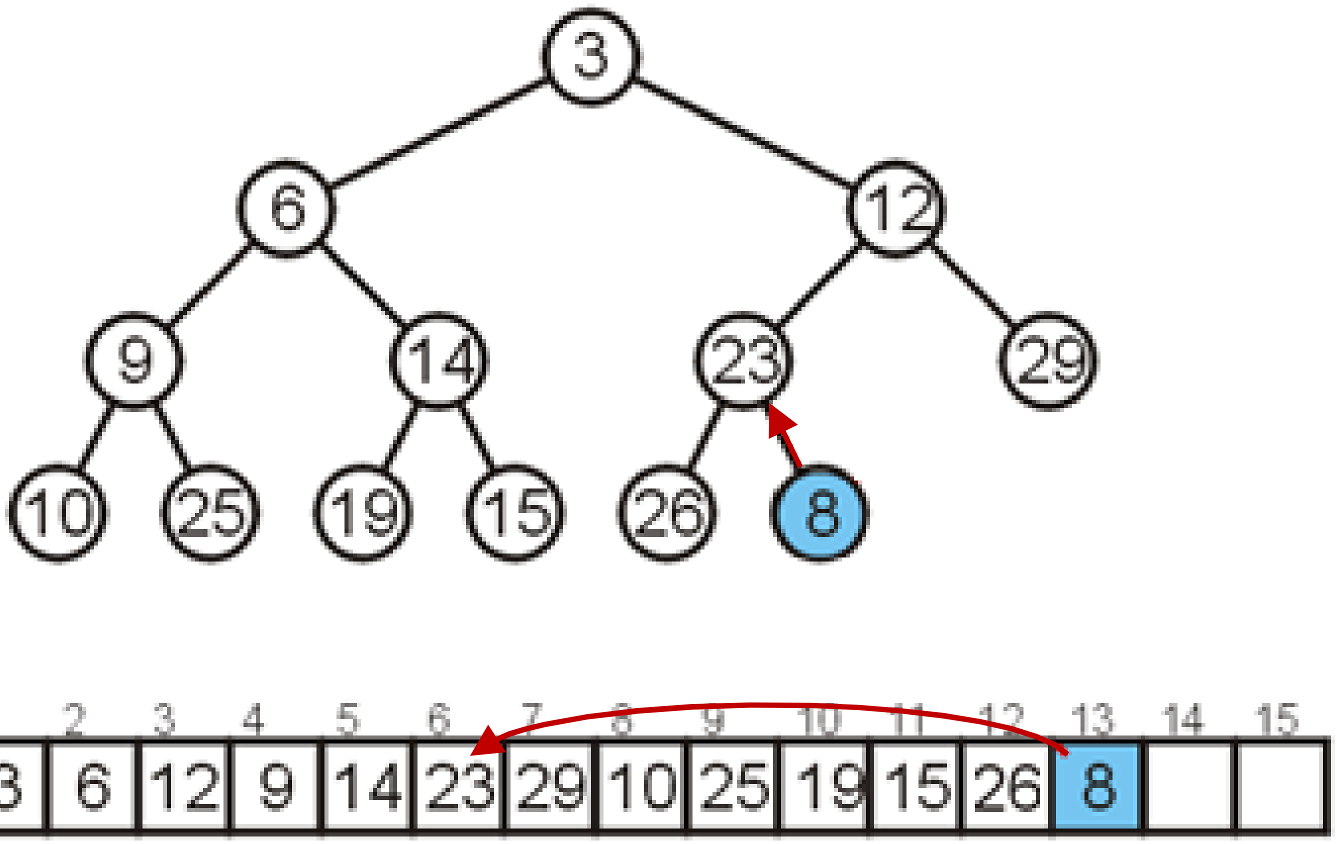


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

(1) Push() Operation

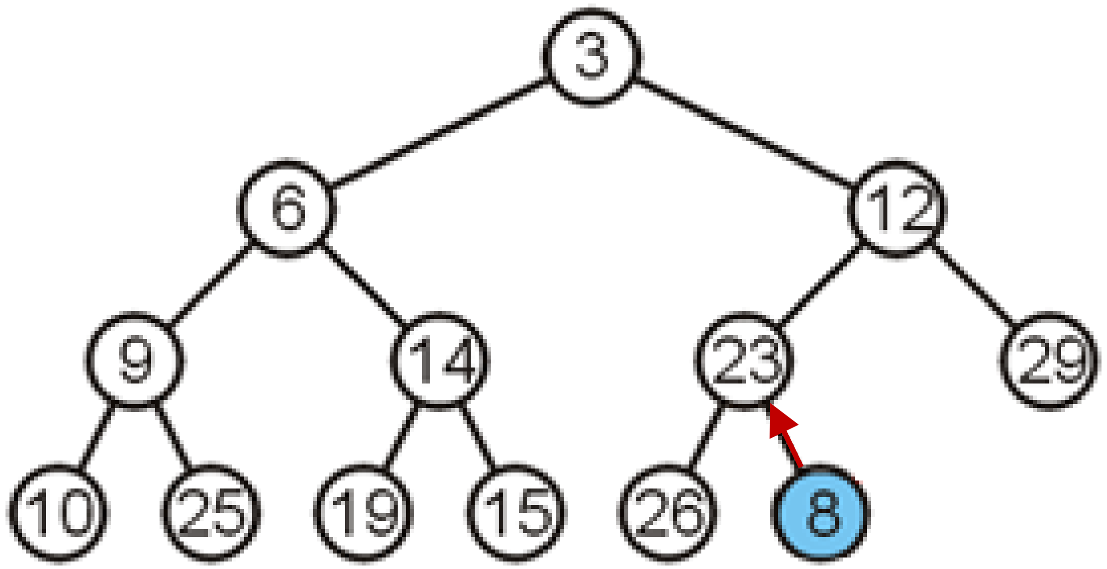
2-1. Heap Operations – Push()

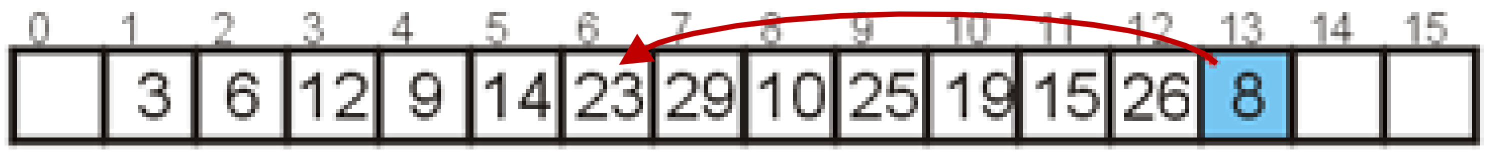
- Suppose we want a min-heap, and we want to insert 8.
  - First, insert it next to the last element in the array.



2-1. Heap Operations – Push()

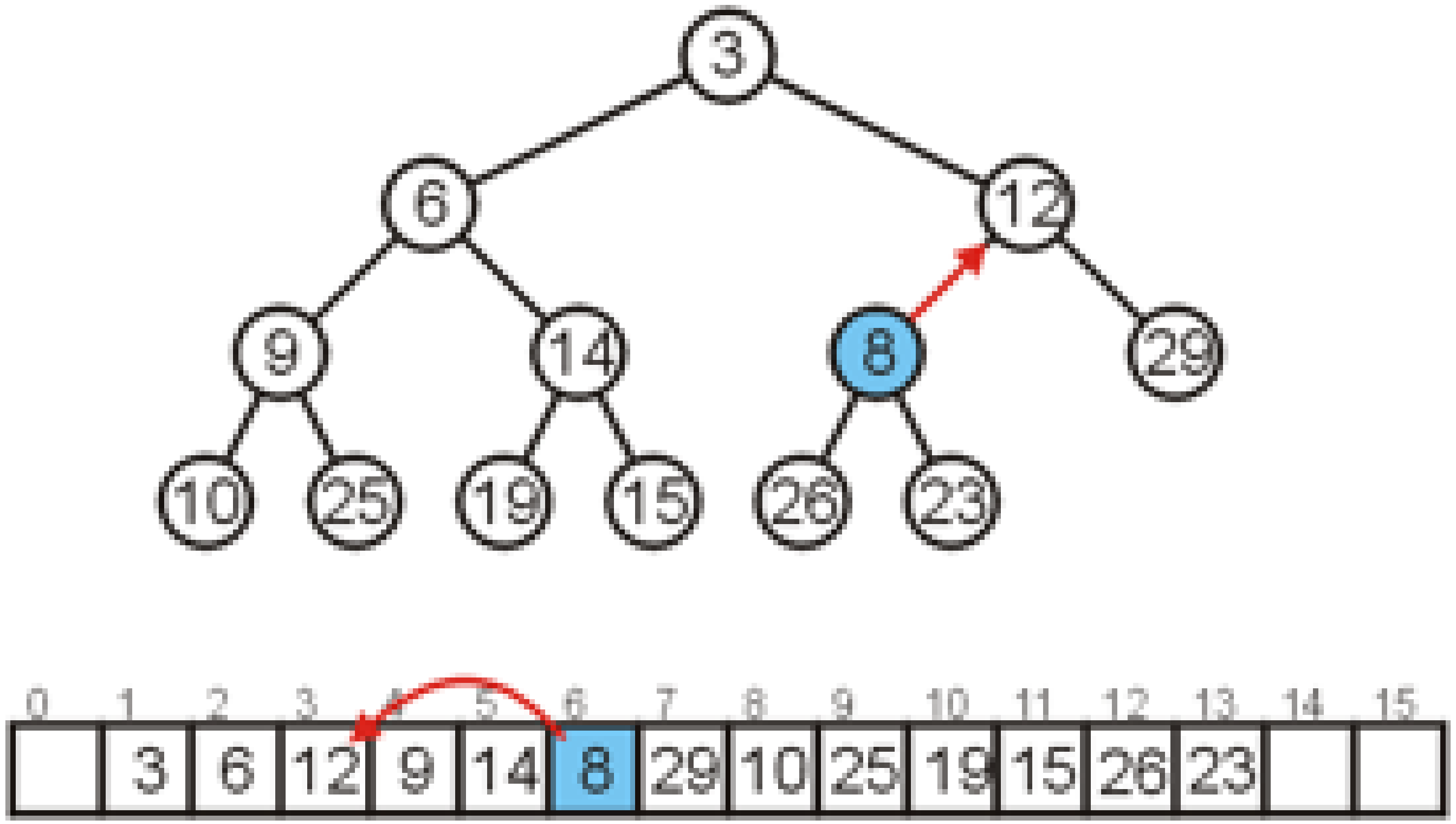
- Inserting 8 requires a few percolations:
  - (1) Swap 8 and 23.





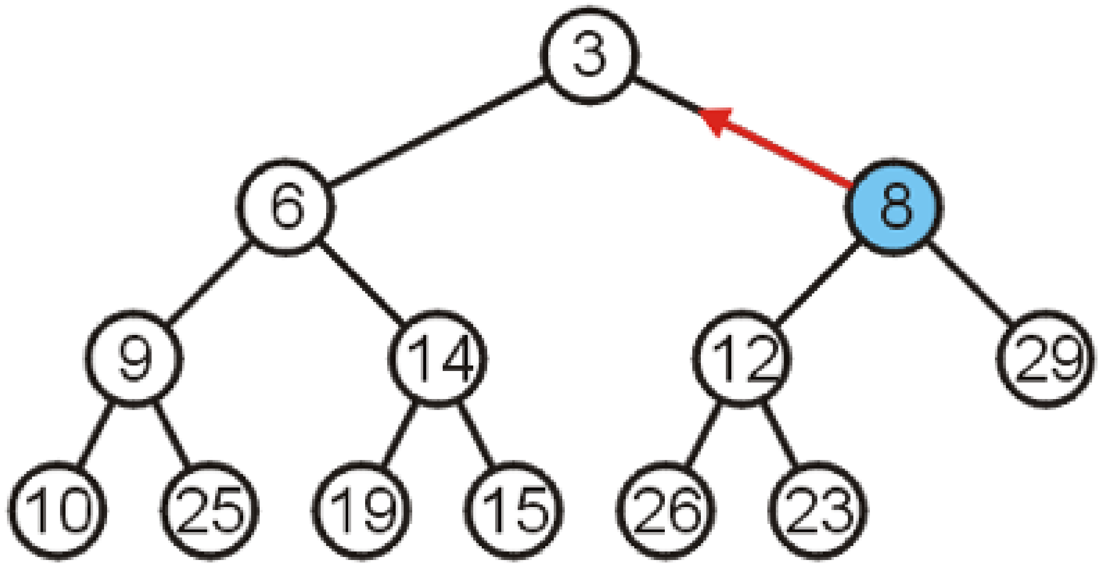
2-1. Heap Operations – Push()

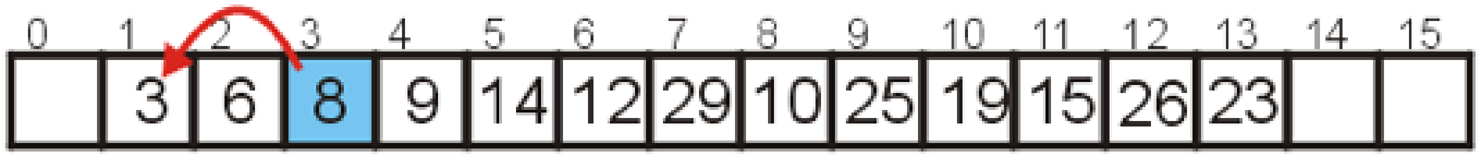
- Inserting 8 requires a few percolations:
  - (2) Swap 8 and 12.



2-1. Heap Operations – Push()

- At this point, it is greater than its parent, so we are finished.





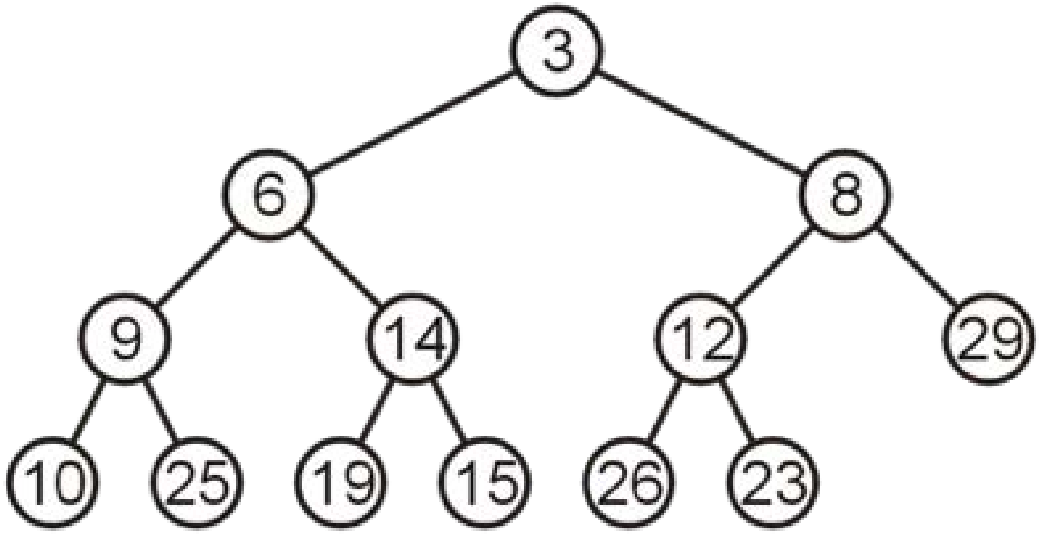
2-1. Heap Operations – Push() Algorithm

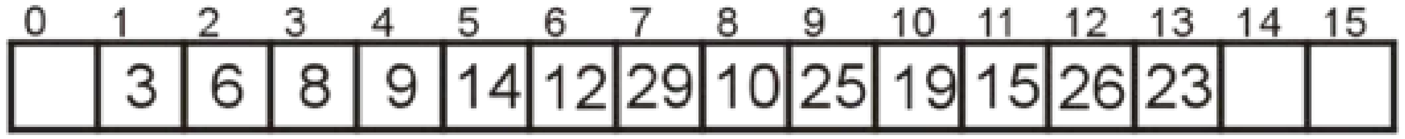
1. Insert it next to the last element in the array.
2. Swap with its parent if the current node is smaller than its parent repeatedly.
    1. If the current node is the root, then the algorithm will be terminated.
    2. If the current node is bigger than its parent, then the algorithm will be terminated.

(2) Pop() Operation

2-1. Heap Operations – Pop()

- As before, popping the top element at index 1.
- Move the last entry to the top, whose index is 1.

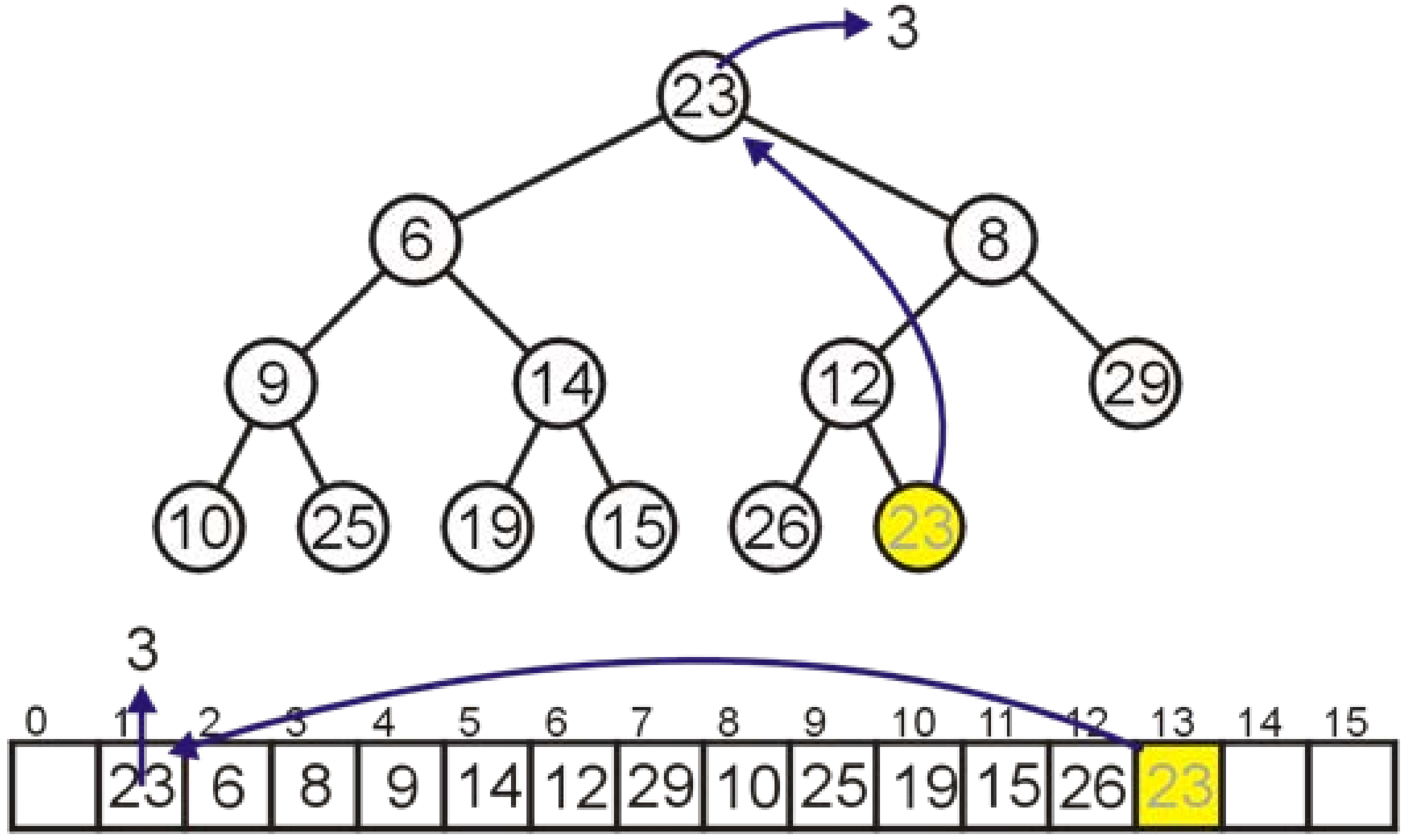




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

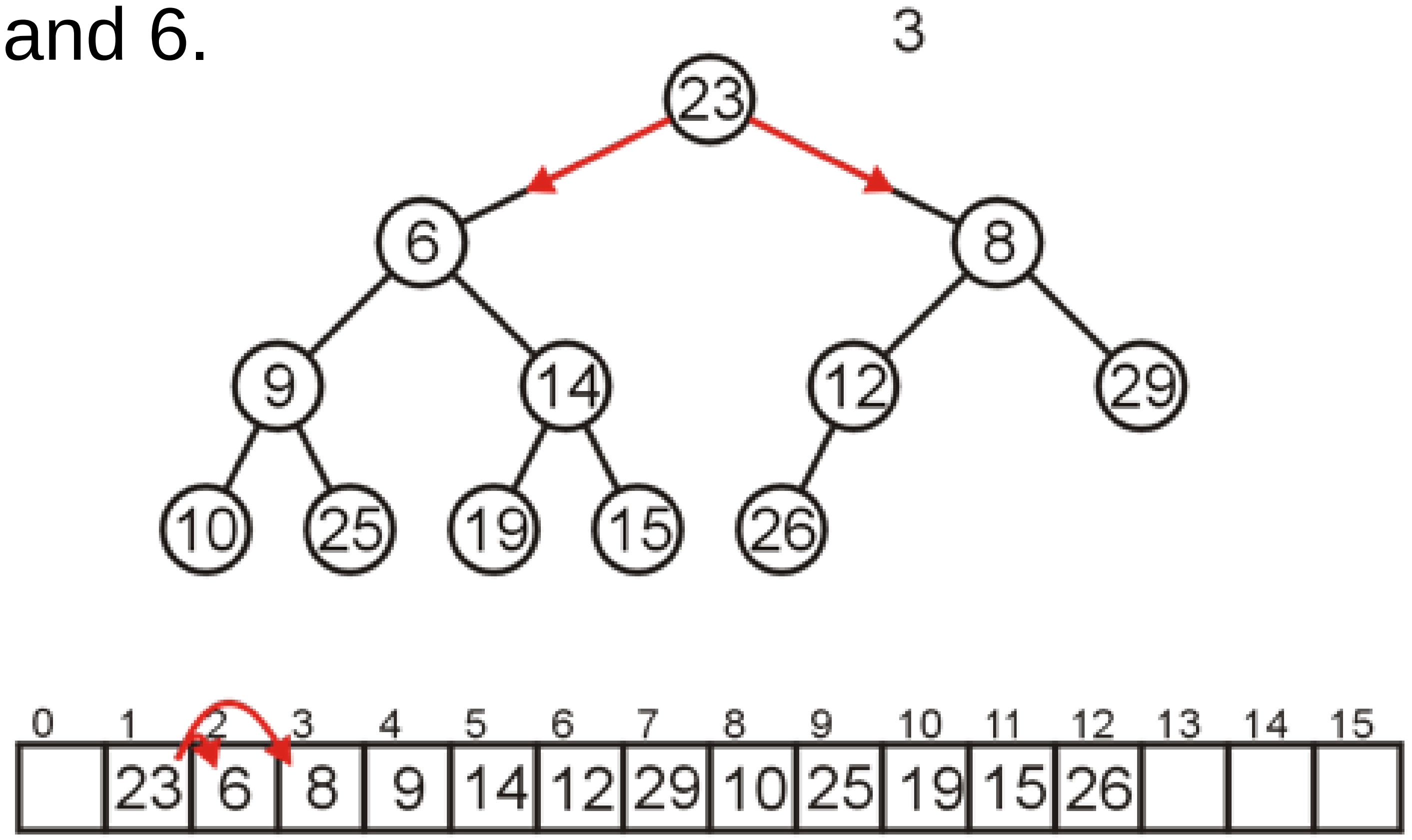
2-1. Heap Operations – Pop()

- As before, popping the top element at index 1.
- Move the last entry to the top, whose index is 1.



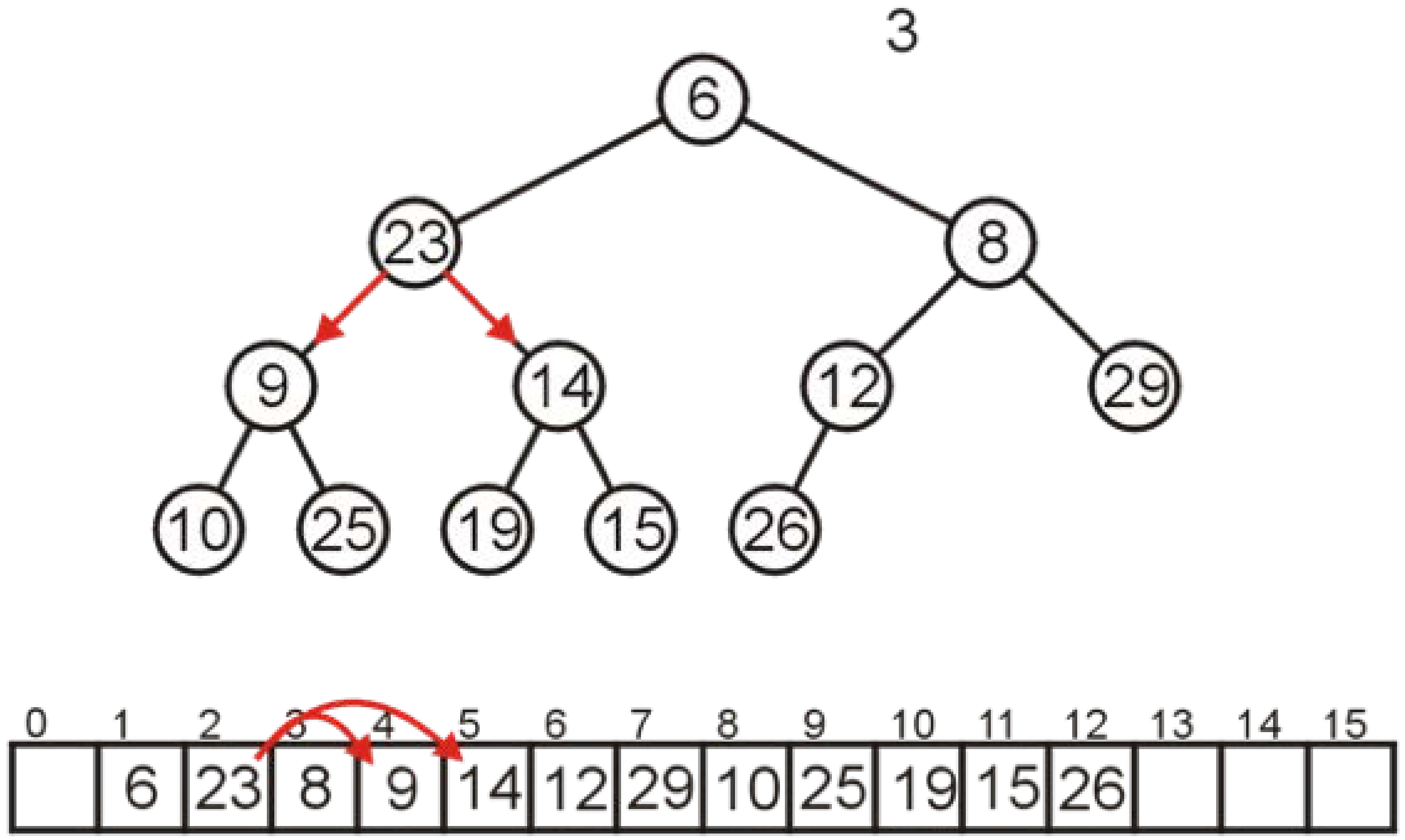
2-1. Heap Operations – Pop()

- Now percolate down. Compare Node 1 with its children: Nodes 2 and 3. Swap with the smallest one.
  - Swap 23 and 6.



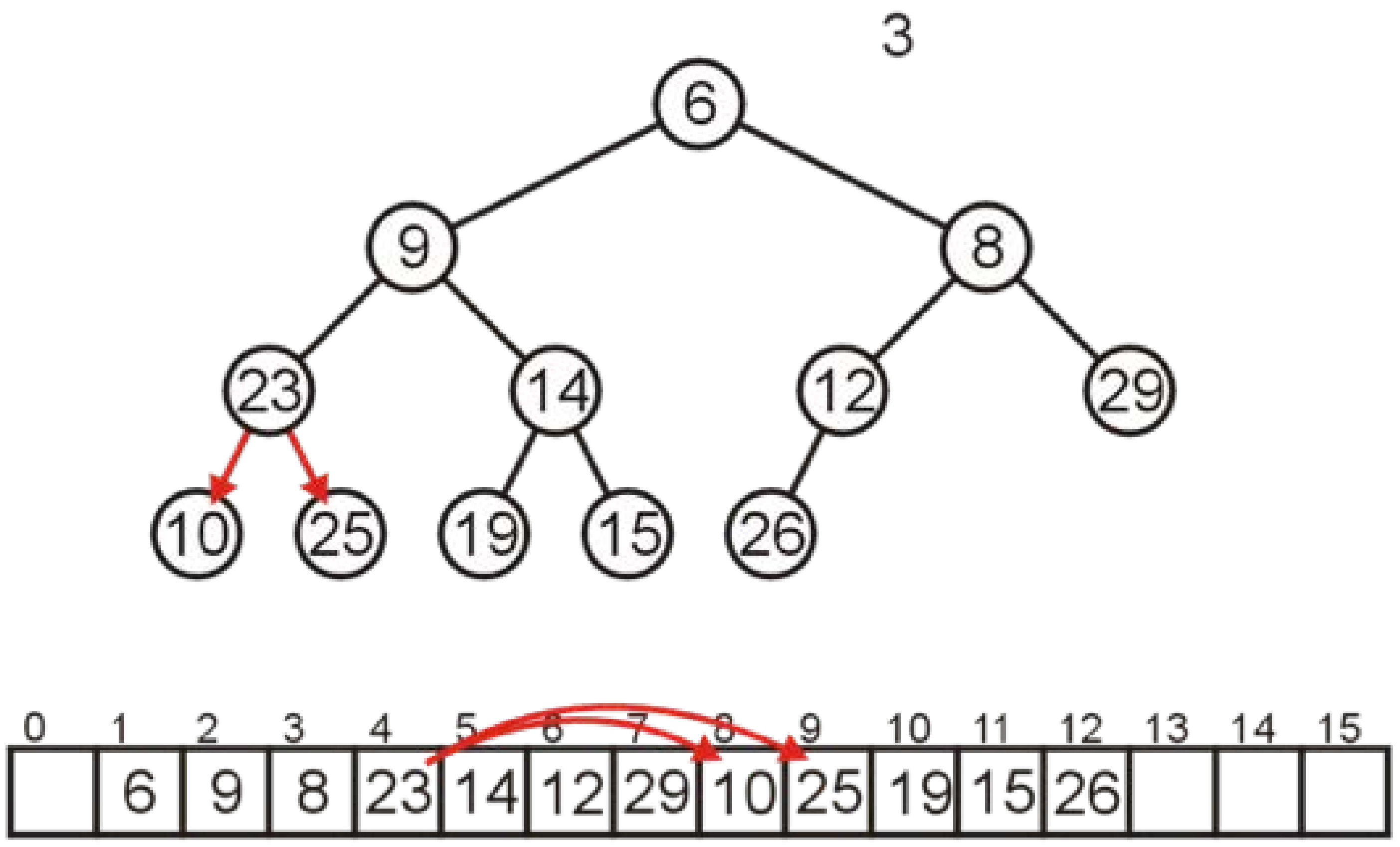
2-1. Heap Operations – Pop()

- Compare Node 2 with its children: Nodes 4 and 5
  - Swap 23 and 9.



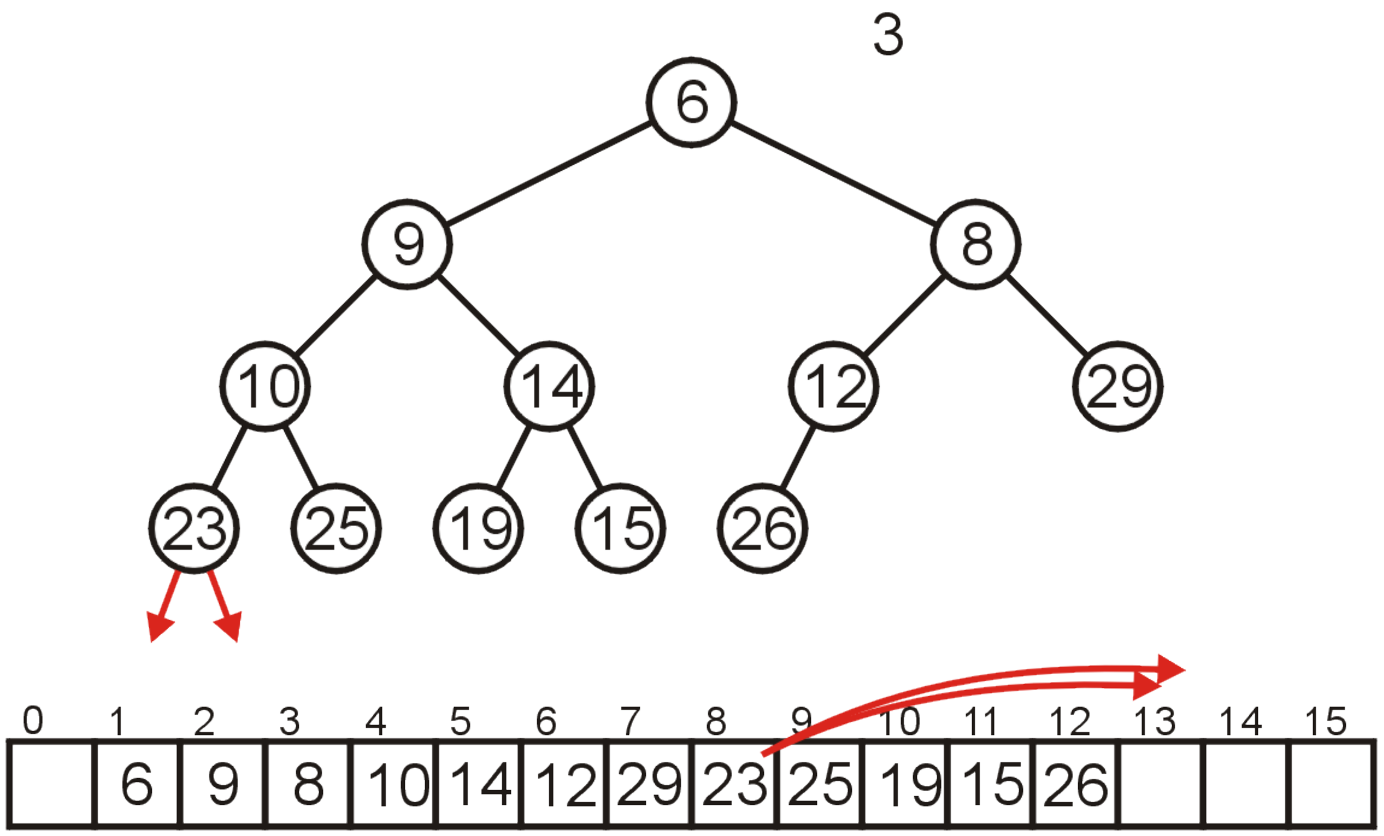
2-1. Heap Operations – Pop()

- Compare Node 4 with its children: Nodes 8 and 9.
    - Swap 23 and 10.



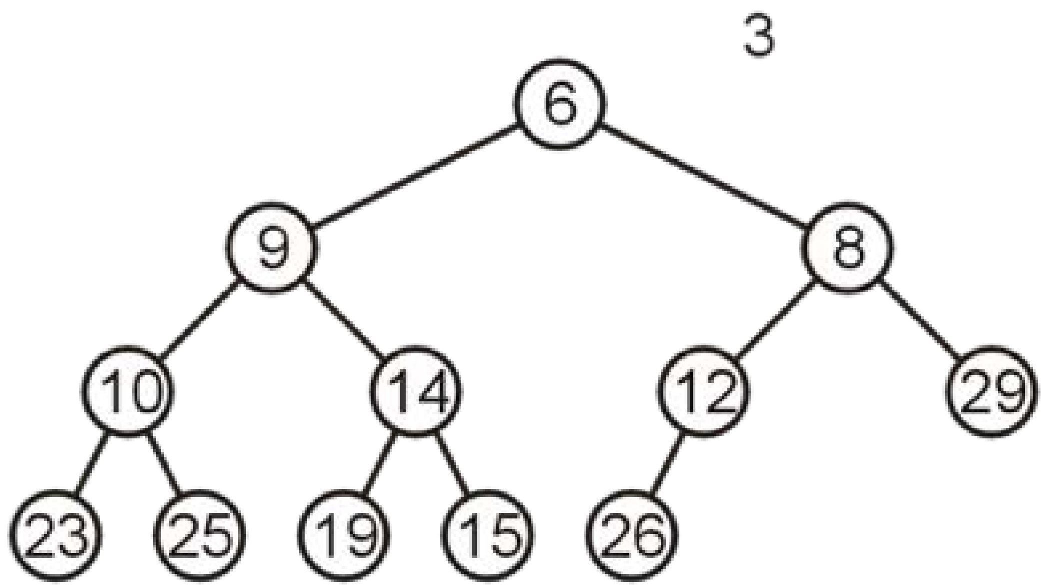
2-1. Heap Operations – Pop()

- The children of Node 8 are beyond the end of the array:
  - Stop.



2-1. Heap Operations – Pop()

- The result is a min-heap. It is still a complete tree.



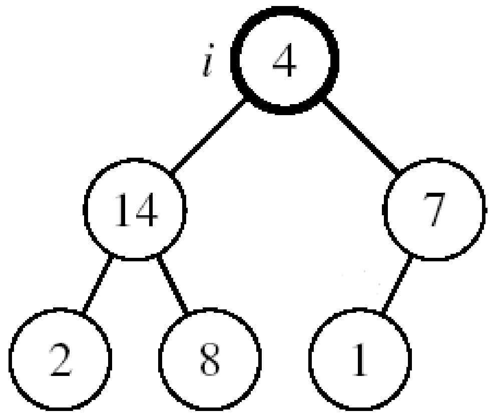
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

2-1. Heap Operations – Pop() Algorithm

1. Pop the top element at index 1.
2. Move the last entry to the top, whose index is 1.
3. Compare the current nodes with its two children, and swap with the smallest one, repeatedly.
    1. If the current node is smaller than both two children, then the algorithm will be terminated.
    2. If the current node has no children, then the algorithm will be terminated.

2-1. Maintaining the Heap Property (Pop())

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children



Alg: MAX-HEAPIFY(*A*, i, n)

```
1.  l ← LEFT(i)
2.  r ← RIGHT(i)
3.  if l ≤ n and A[l] > A[i]
4.      then largest ← l
5.      else largest ← i
6.  if r ≤ n and A[r] > A[largest]
7.      then largest ← r
8.  if largest ≠ i
9.      then exchange A[i] ↔ A[largest]
10.             MAX-HEAPIFY(A, largest, n)
```

[Recursive]

(3) Push(), Pop() Time Complexity

Run-time Analysis

- Accessing the top object is $\mathbf{\Theta}(1)$, since it is an array.

- Popping the top object is $\mathbf{O}(\ln(n))$
  - We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

- Pushing an object is also $\mathbf{O}(\ln(n))$
  - If we insert an object less than the root, it will be moved up to the top

- Space complexity $\mathbf{O}(n)$

- *So binary heap is a better implementation of priority queue*

Run-time Analysis (Push)

- (Worst) If we are inserting an object less than the root (at the front), then the run time will be $\Theta(\ln(n))$
- (Best) If we insert at the back (greater than any object) then the run time will be $\Theta(1)$
- (Average) The run time will be $\Theta(1)$. **Why?**

- For a complete tree, $h = \Theta(\ln(n))$.

Run-time Analysis (Average - Push)

- With each percolation, it will move an object past half of the remaining entries in the tree
  - Therefore after one percolation, it will probably be past half of the entries, and therefore *on average* will require no more percolations

$$\frac{1}{n}\sum_{k=0}^{h}(h-k)2^{k} = \frac{2^{h+1}-h-2}{n}$$

$$= \frac{n-h-1}{n} = \Theta(1)$$

- Therefore, we have an average run time of $\Theta(1)$
- What would be the average run time of pop then?
  - The answer is O(ln(n))

(4) Build Heap

2-1. Build Heap – Simple Method

- Task: Given a set of n keys, build a heap all at once

- Approach 1:
  - Repeatedly perform push

- Complexity
  - $O(n \ln(n))$

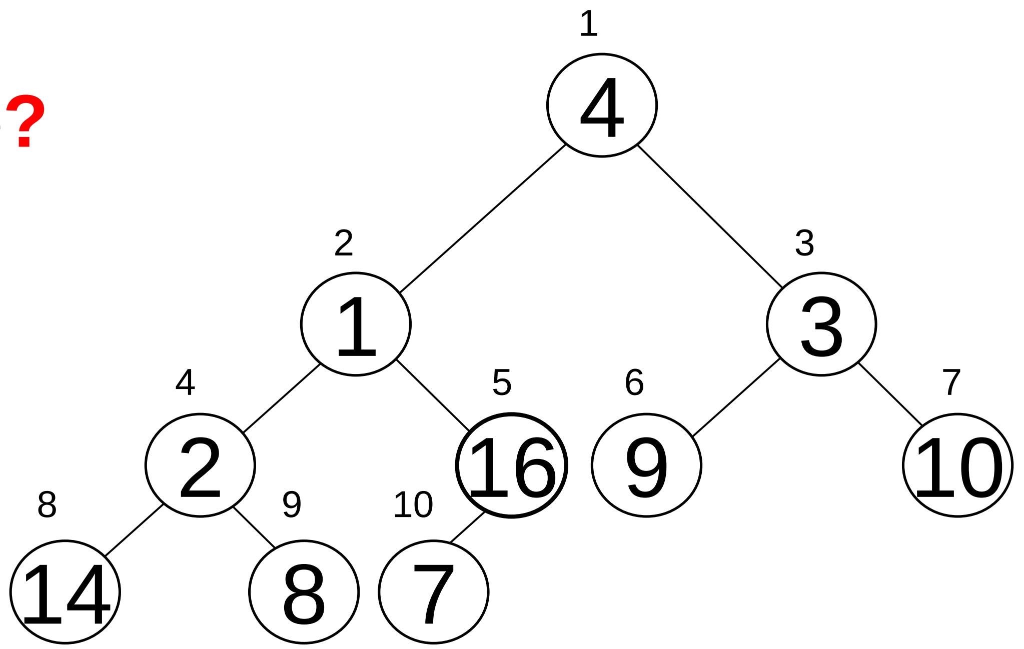
2-1. Build Heap – Floyd’s Method

- Task: Given a set of n keys, build a heap all at once
- Approach 2:
  - Floyd’s Method
  - Put all the keys in a binary tree first and then fix the heap property!
  - You only need to do it at most half keys. Why?
- **Complexity !!!**
  - $\boldsymbol{O(n)}$

2-1. Build Heap – Floyd's Method

- Task: Given a set of n keys, build a heap all at once
- Approach 2:
    - Floyd's Method. **Why half of size?**

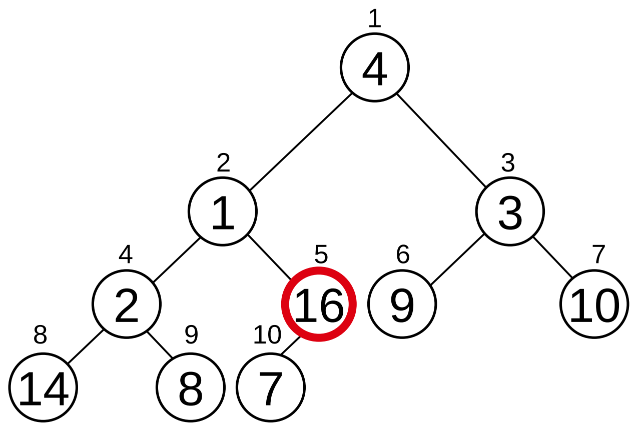
```
buildHeap(){
  for (i=size/2; i>0; i--)
     percolateDown(Array[i]);
}
```



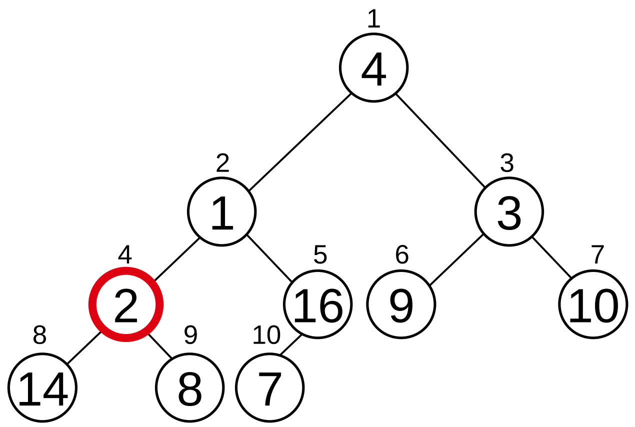
A:									
4	1	3	2	16	9	10	14	8	7

2-1. Build Heap – Floyd's Method (max heap)

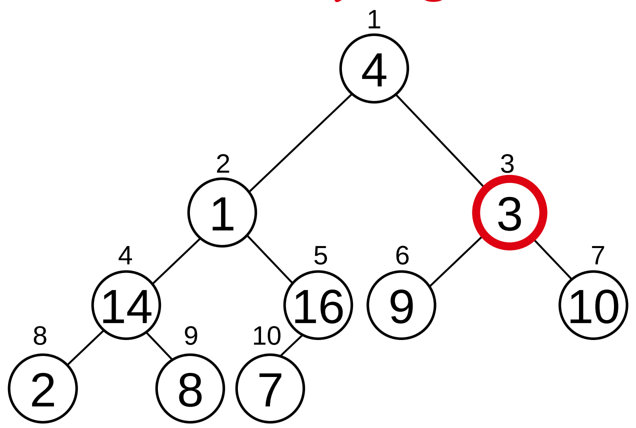
$i = 5$



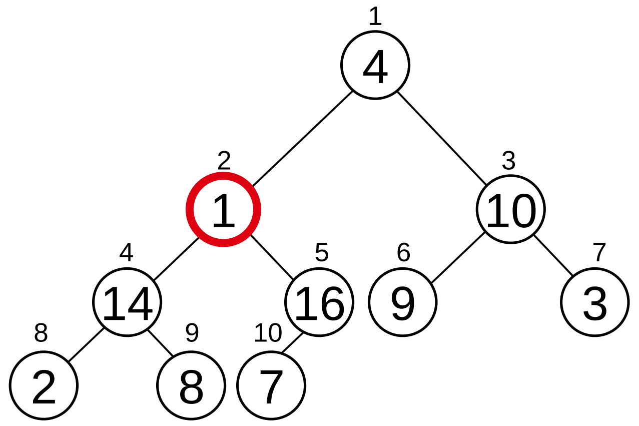
$i = 4$



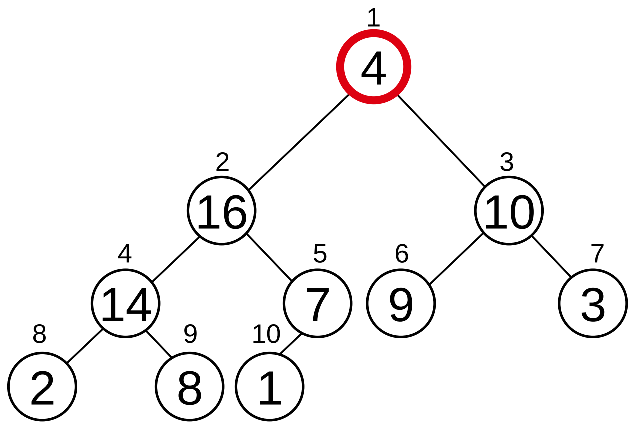
$i = 3$

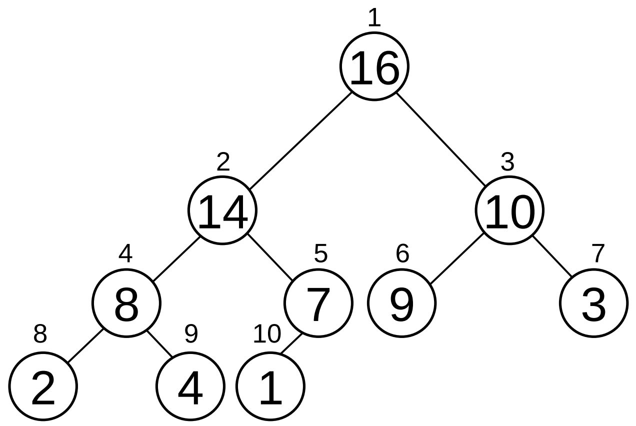


$i = 2$



$i = 1$



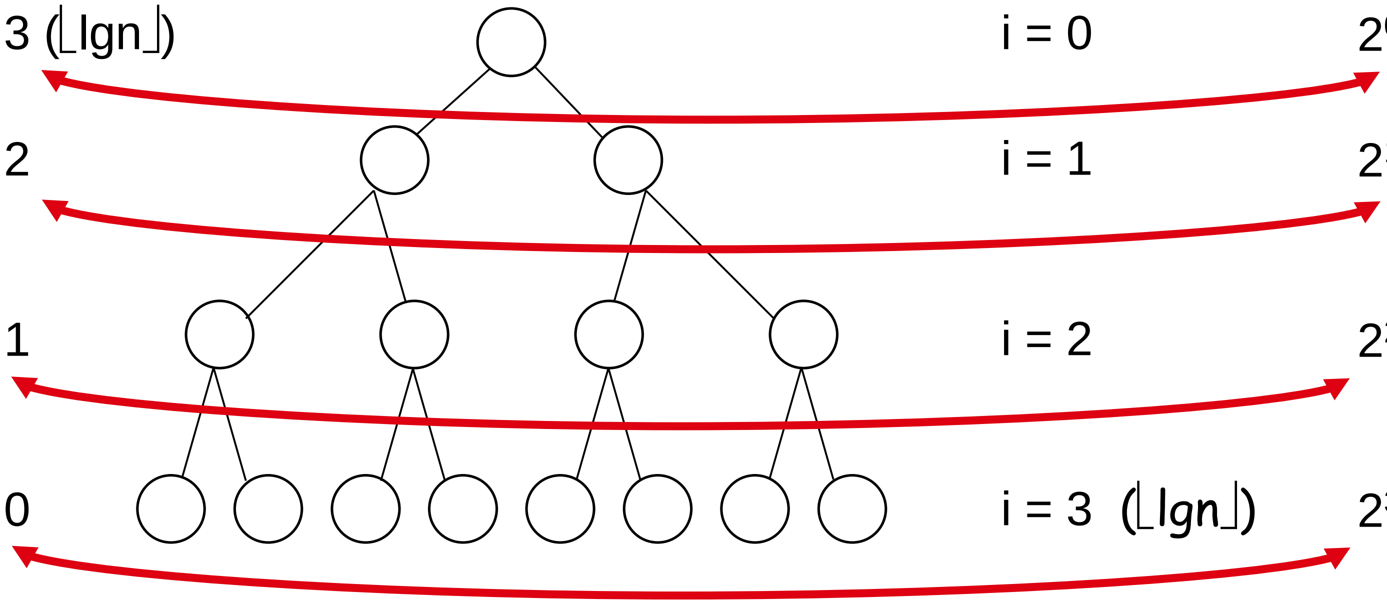


2-1 Running Time of Floyd's Method

- HEAPIFY takes O(h) $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3 ($\lfloor \lg n \rfloor$)	2^3



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

2-1. Build Heap – Floyd's Method

- No percolation for the leaf nodes (n/2 nodes)
- At most n/4 nodes percolate down 1 level
  At most n/8 nodes percolate down 2 levels
  At most n/16 nodes percolate down 3 levels
  …

$$1\frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + \cdots = \sum_{i=1}^{\log n} i\frac{n}{2^{i+1}} = \frac{n}{2}\sum_{i=1}^{\log n}\frac{i}{2^i} = n = \Theta(n)$$

2-1 Running Time of Floyd’s Method

$$T(n) = \sum_{i=0}^{h} n_i h_i$$

Cost of HEAPIFY at level i $*$ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$

Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

Change variables: k = h - i

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

The sum above is smaller than the sum of all elements to ∞ and h = lgn

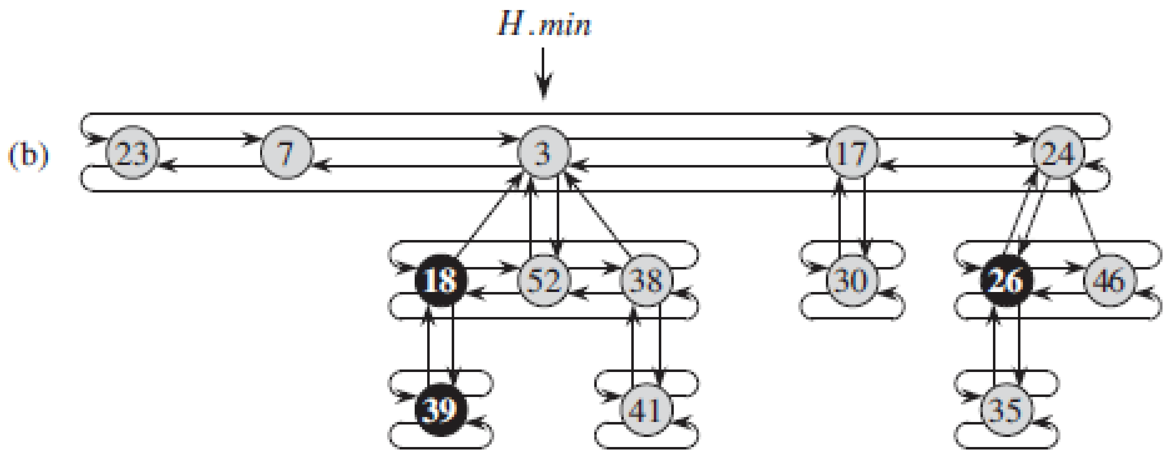
$$= O(n)$$

The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

Summary 2

- Binary heap
  - Based on priority queue, complete binary tree.
  - Implementation using arrays
  - *Heap != Tree.



Fibonacci Heap

Summary 2 – Binary Heap

	Average	**Worst**
Top	$\Theta(1)$	$\Theta(1)$
Push	$\underline{\Theta(1)}$	$\Theta(\log n)$
Pop	$O(\log n)$	$\Theta(\log n)$
Build Heap		$\Theta(n)$

2-2. Heap Sort

2-2. Heapsort

- Heapsort
  - Place the objects into a heap
    - $O(n)$ time
  - Repeatedly popping the top object until the heap is empty
    - $O(n \ln(n))$ time
  - Time complexity: $O(n \log n)$

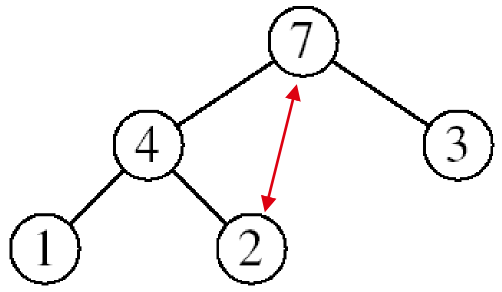
2-2. Heapsort

- Goal:
  - Sort an array using heap representations (increasing order)
- Idea:
  - Convert the unordered array to a **max-heap.**
  - Swap the root (the maximum element) with the last element in the array *-> like the pop() operation*.
  - “Discard” this last node by decreasing the heap size.
  - Call MAX-HEAPIFY on the new root. *-> like the pop() operation*
  - Repeat this process until only one node remains.

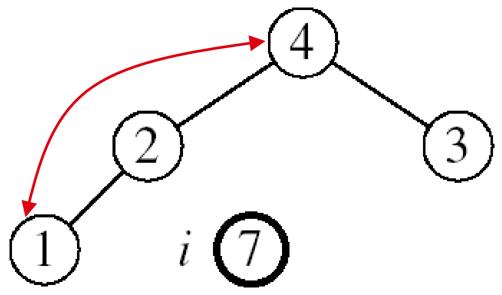
2-2. Heapsort Algorithm

```
1. BUILD-MAX-HEAP(A)                              $O(n)$
2.  for i ← length[A] downto 2
3.        do exchange A[1] ↔ A[i]                 $n$-1 times
4.              MAX-HEAPIFY(A,1,i-1)              $O(lgn)$
```

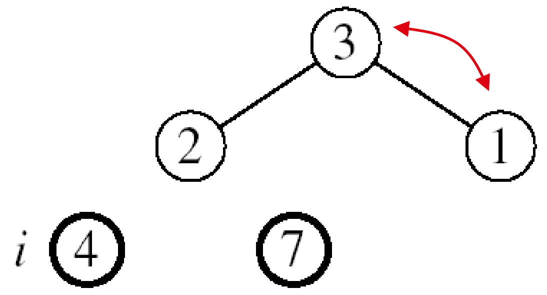
2-2. Heap Sort Example 1



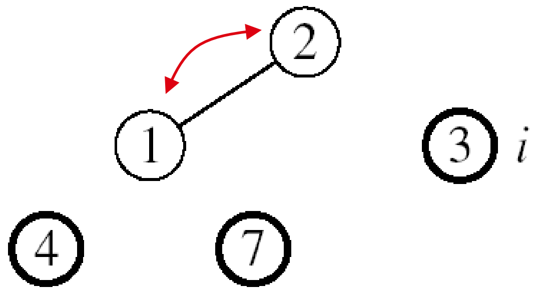
MAX-HEAPIFY(A, 1, 4)



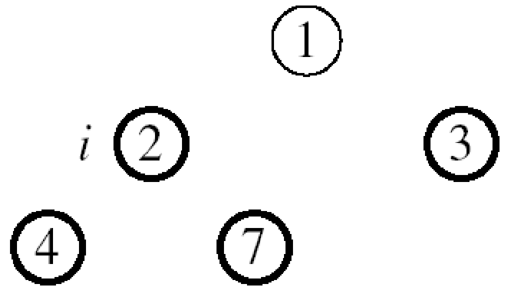
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



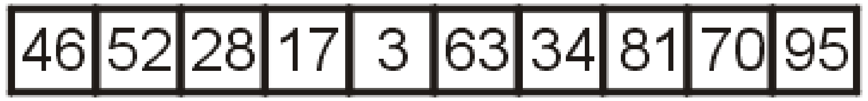
MAX-HEAPIFY(A, 1, 1)



A	1	2	3	4	7

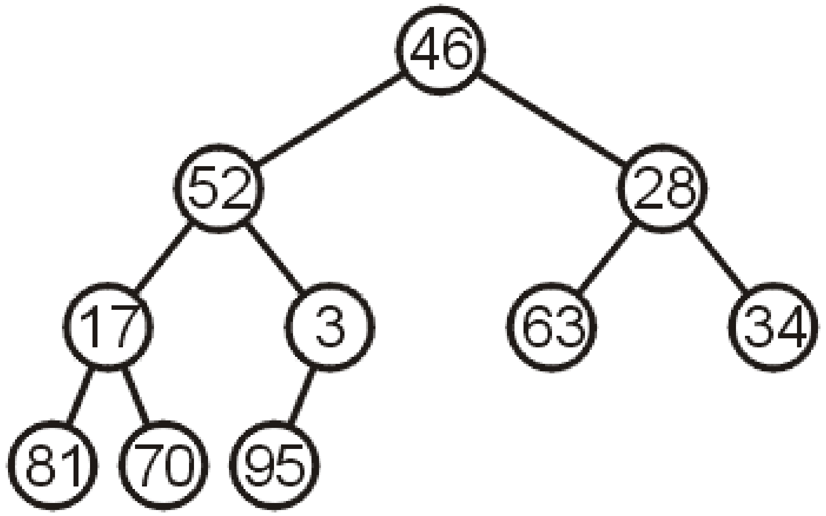
2-2. Heap Sort Example 2

- First, we must convert the unordered array with $n = 10$ elements into a max-heap.



46	52	28	17	3	63	34	81	70	95

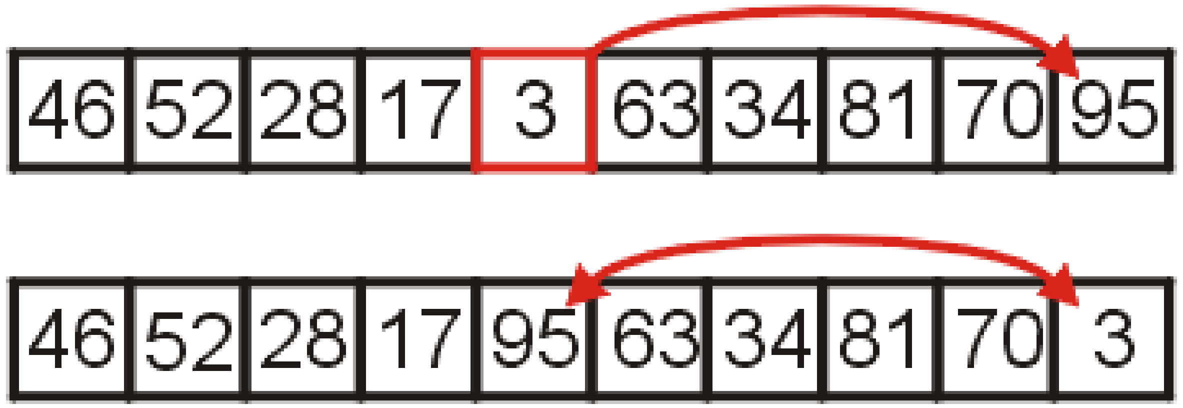
- None of the leaf nodes need to be percolated down, and the last non-leaf node is in position $n/2$



- Thus we start with position $10/2\text{-}1 = 5$

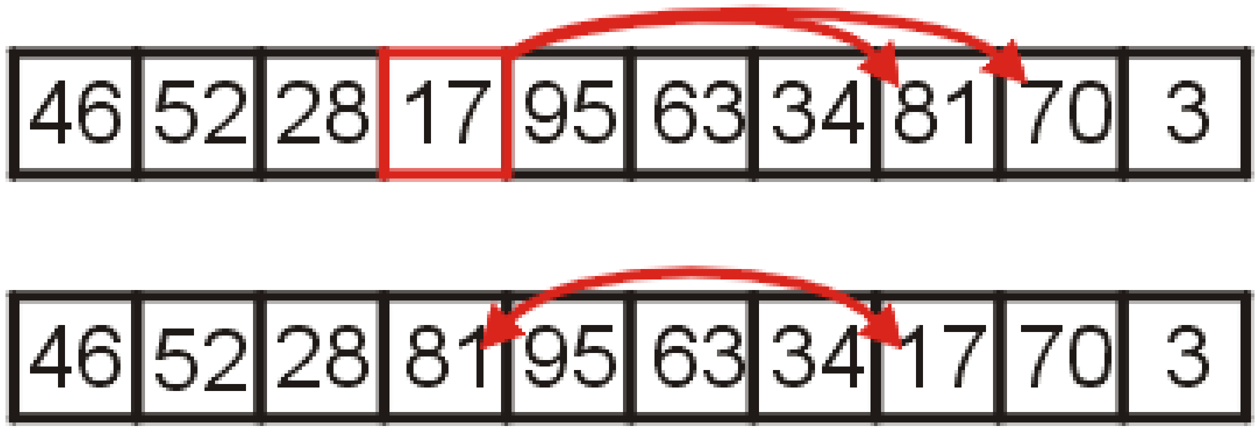
2-2. Example Heap Sort – Build Heap

- We compare 3 with its child and swap them



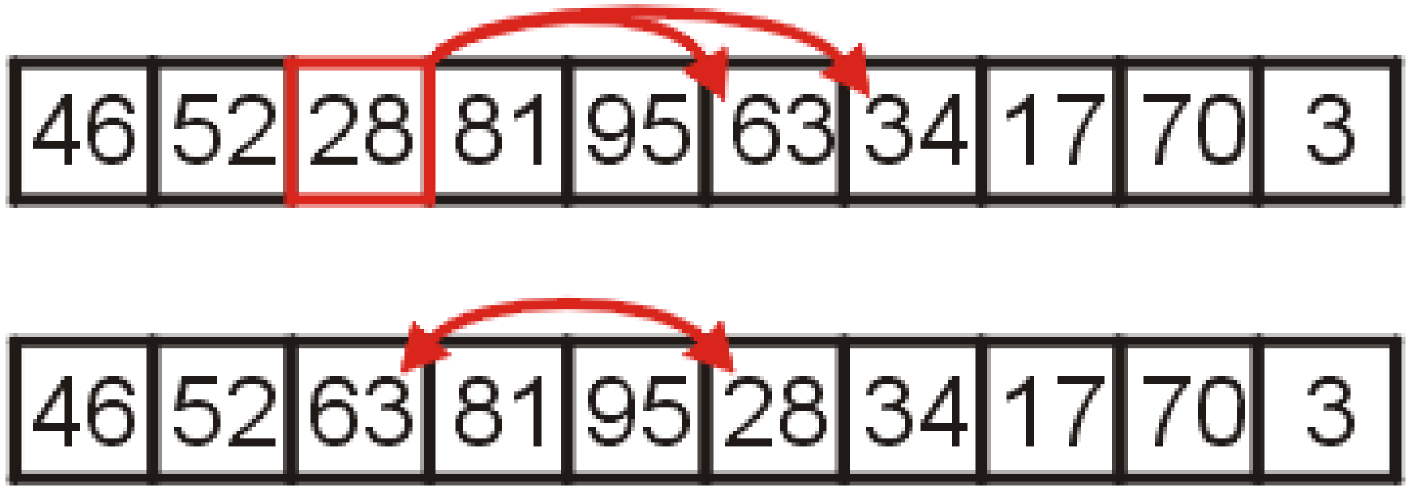
2-2. Example Heap Sort – Build Heap

- We compare 17 with its two children and swap it with the maximum child (81)



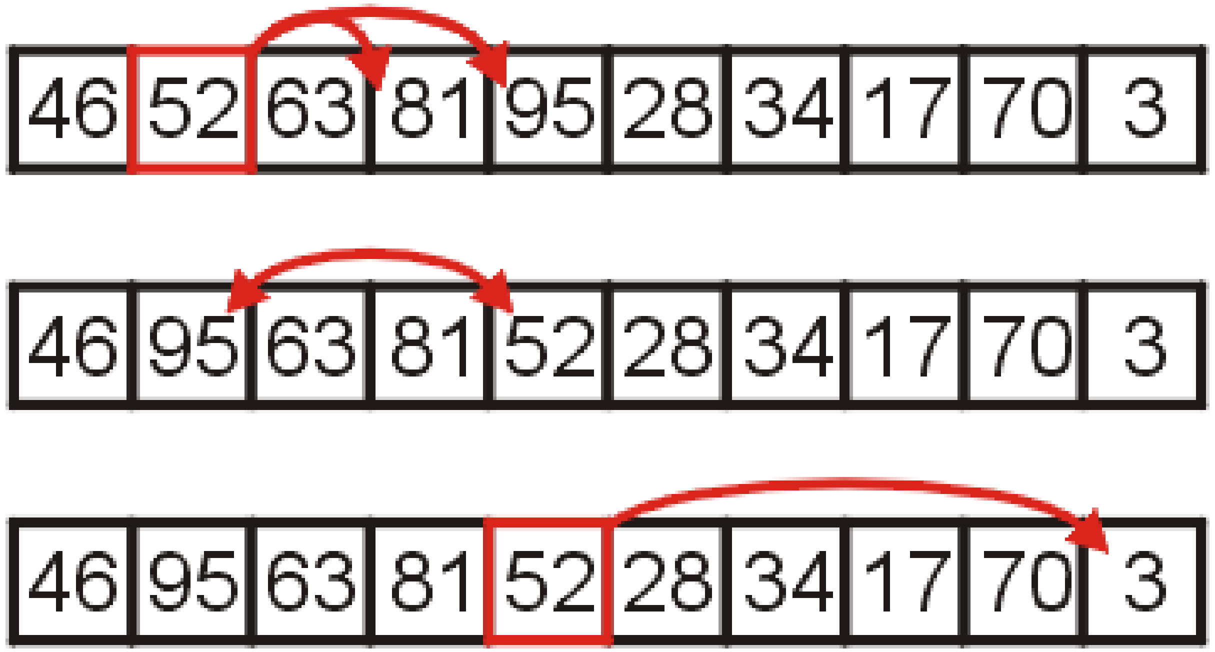
2-2. Example Heap Sort – Build Heap

- We compare 28 with its two children, 63 and 34, and swap it with the largest child.



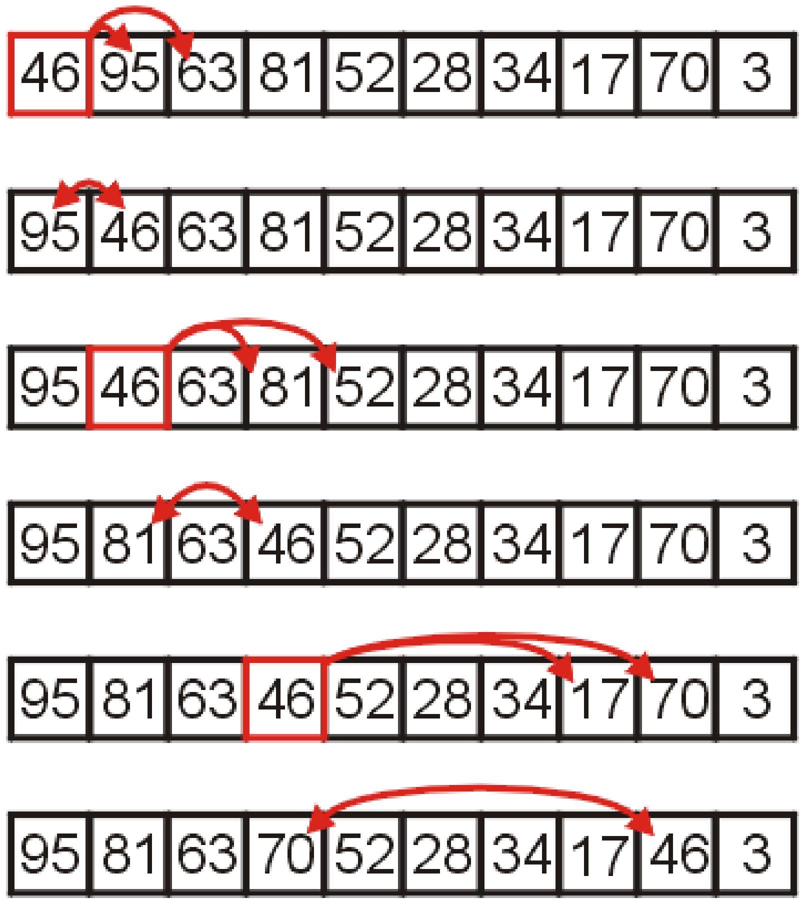
2-2. Example Heap Sort – Build Heap

- We compare 52 with its children, swap it with the largest
  - Recursing, no further swaps are needed



2-2. Example Heap Sort – Build Heap

- Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70



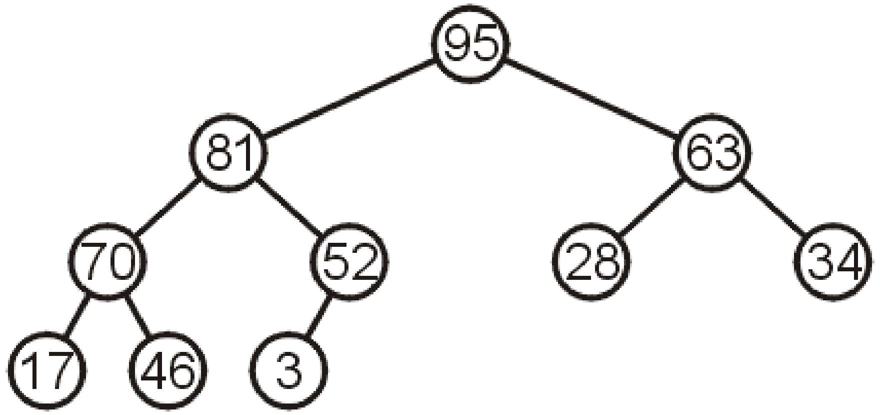
2-2. Example Heap Sort – Build Heap

- We have now converted the unsorted array:

46	52	28	17	3	63	34	81	70	95

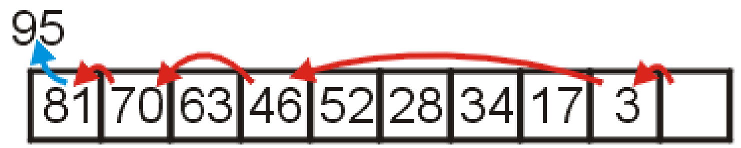
  into a max-heap:

95	81	63	70	52	28	34	17	46	3

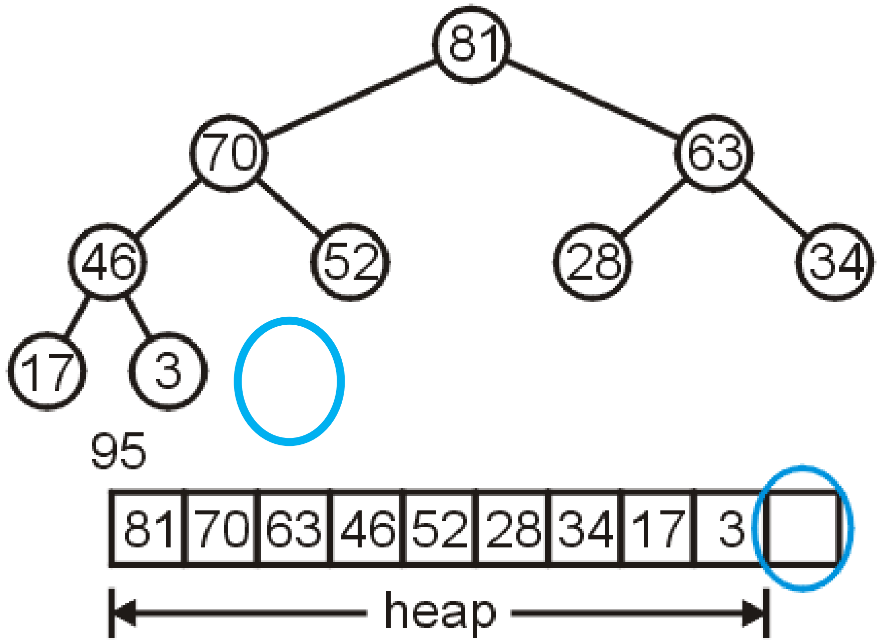


2-2. Example Heap Sort – Get Largest

- We pop the maximum element of this heap

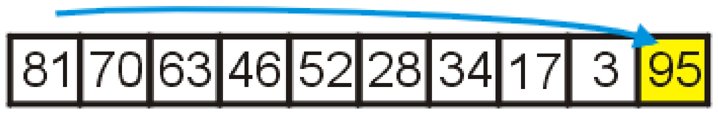


- This leaves a gap at the back of the array:

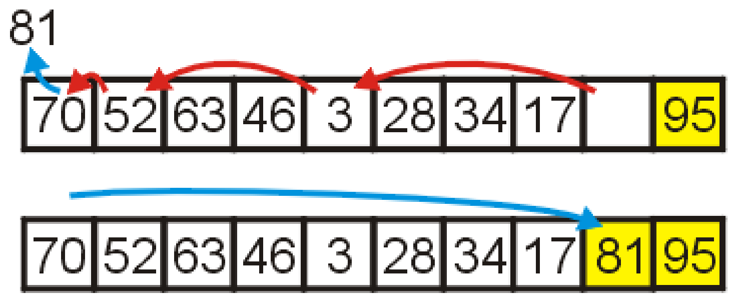


2-2. Example Heap Sort – Get Largest

- This is the last entry in the array, so why not fill it with the largest element?

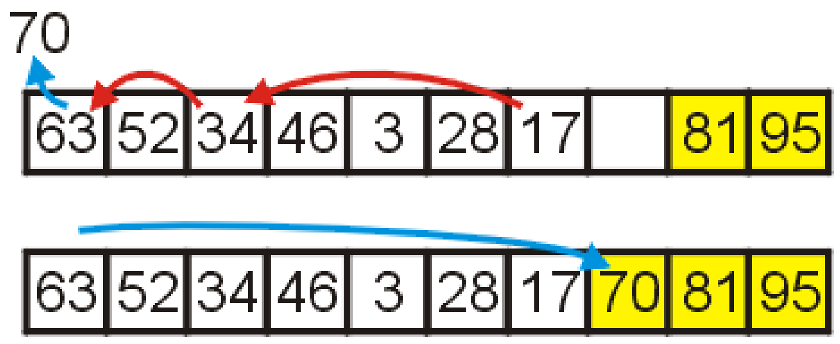


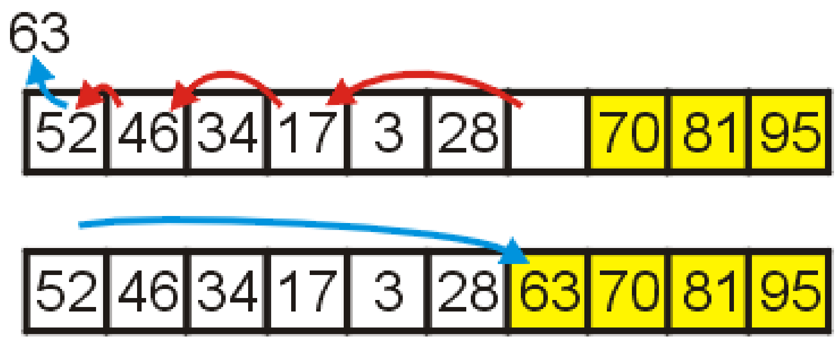
- Repeat this process: pop the maximum element, and then insert it at the end of the array:



2-2. Example Heap Sort – Get Largest

- Repeat this process
  - Pop and append 70

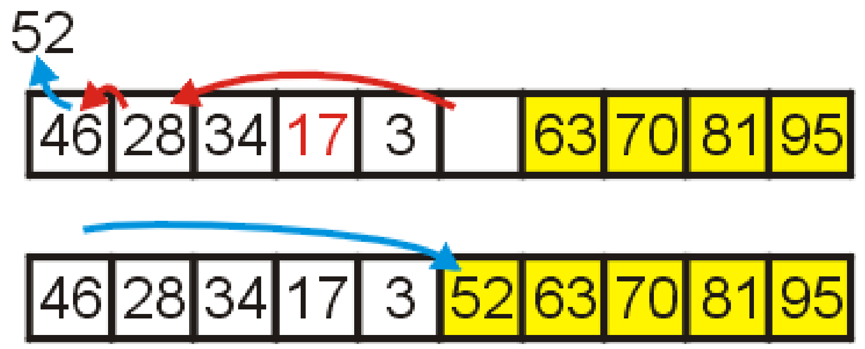


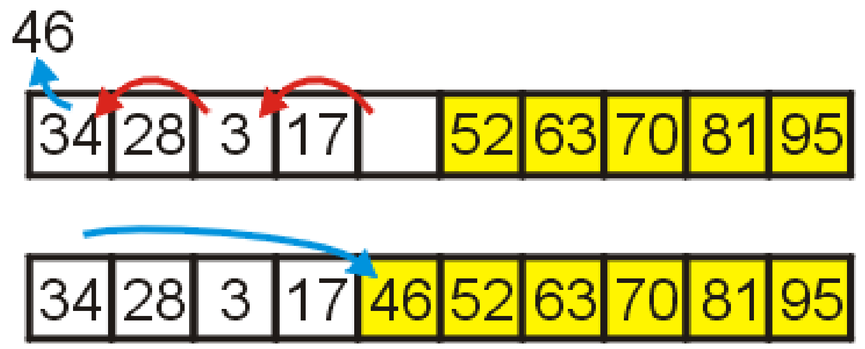


  - Pop and append 63

2-2. Example Heap Sort – Get Largest

- We have the 4 largest elements in order
  - Pop and append 52

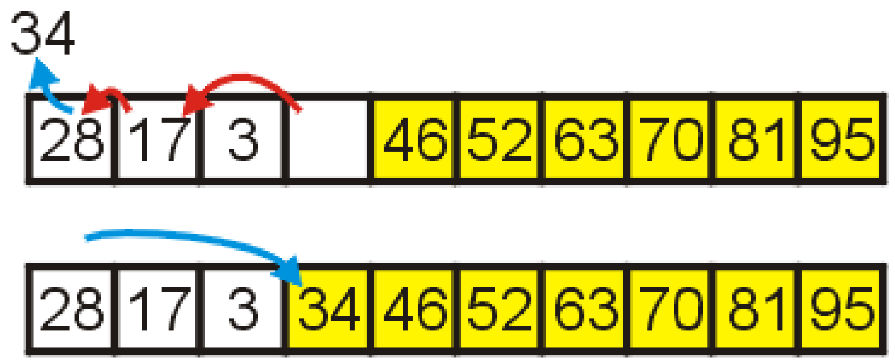


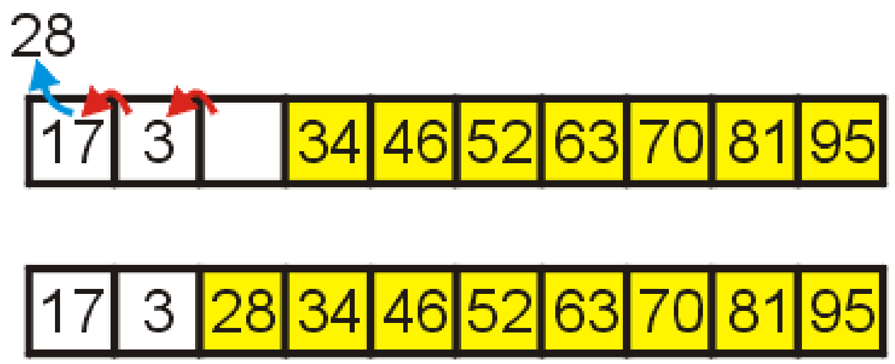


  - Pop and append 46

2-2. Example Heap Sort – Get Largest

- Continuing...
  - Pop and append 34

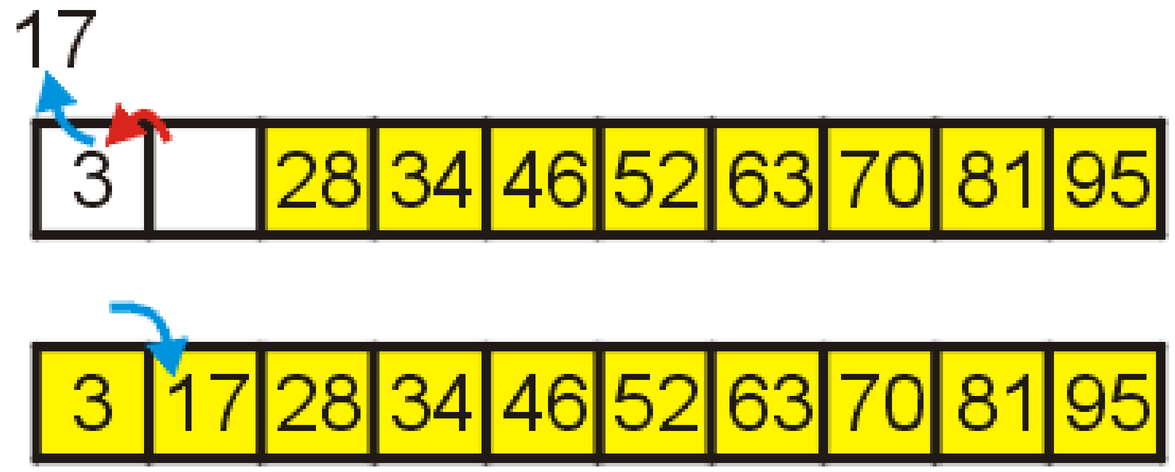




  - Pop and append 28

2-2. Example Heap Sort – Get Largest

- Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted.



Summary 3

- Heap Sort
  - Time Complexity: O(nln(n))
  - Step: (1) Build Heap (2) Pop and Add it to the back
  - Additional Space Complexity: O(1), in-place sorting!

2-3. Exercise Part 2

Exercise 5 – Judge Statement

- T/F: We can always find the maximum in a min-heap in $O(\log n)$ time.

- *Adapt from MIT, Quiz 1, 6.006 2010 Spring.*

Exercise 5 – Judge Statement

- T/F: We can always find the maximum in a min-heap in $O(\log n)$ time.
- Solution: FALSE. The maximum element in a min-heap can be anywhere in the bottom level of the heap. There are up to $n/2$ elements in the bottom level, so finding the maximum can take up to $O(n)$ time.

Exercise 6 – Judge Statement

- T/F: Given a binary min-heap of n distinct elements and a real number x, there exists an algorithm to determine whether the kth ($k \leq n$) smallest element is less than x in $O(k)$ time. (Not $O(n)$ time. No other operation.)

- *Adapt from MIT, Quiz 1, 6.006 2010 Spring.*

Exercise 6 – Judge Statement

- T/F: Given a binary min-heap of n distinct elements and a real number x, there exists an algorithm to determine whether the kth ($k \leq n$) smallest element is less than x in $O(k)$ time. (Not $O(n)$ time. No other operation.)
- Solution: TRUE.
- A solution is to start at the top of the heap and recursively visit the subheaps keeping a global count of the keys seen that have value less than x. If some key has value at least x, then we do not need to visit any of its children. If the global count reaches , then we terminate after $O(k)$ steps and output “yes”. If we have visited all the nodes with keys of value less than x and the counter is less than k, then we also terminate in $O(k)$ steps and output “no”.

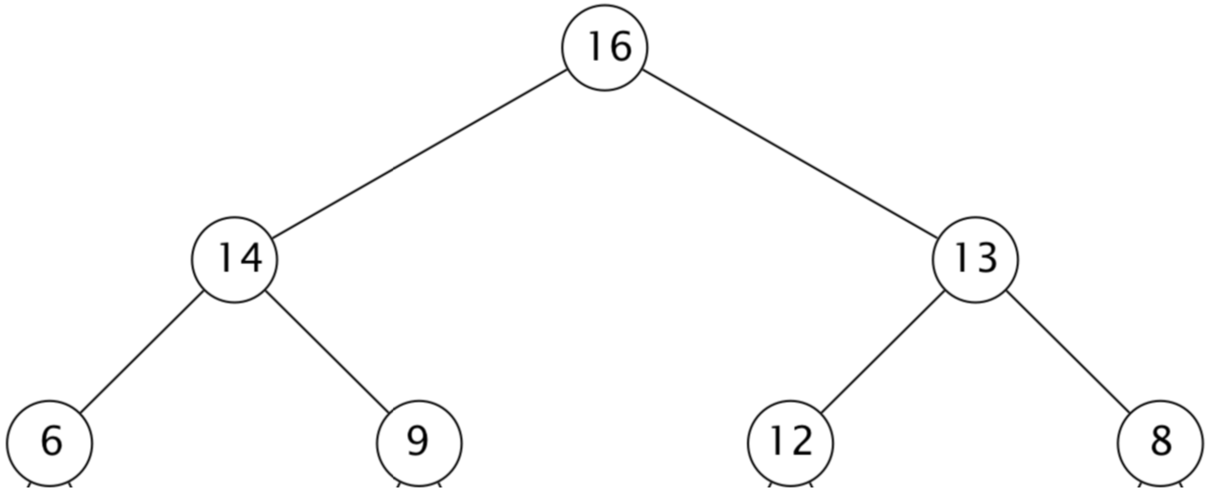
Exercise 6 – Judge Statement

- T/F: Given a binary min-heap of n distinct elements and a real number x, there exists an algorithm to determine whether the kth ($k \leq n$) smallest element is less than x in $O(k)$ time. (Not $O(n)$ time. No other operation.)
- Solution: TRUE.
- We give pseudocode for the procedure KTHLESSTHANX below. The algorithm takes as input a heap H, a real number x, and a positive integer k.

```
KTHLESSTHANX($H, x, k$):
1  $count \leftarrow 0$.
2  RECURSE($H, x, k$):
3      if ($count < k$) and ($H! = null$):
4              if ($H.min < x$):
5                  $count \leftarrow count + 1$.
6                  RECURSE($H.left, x, k$).
7                  RECURSE($H.right, x, k$).
8  return ($count == k$).
```

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.

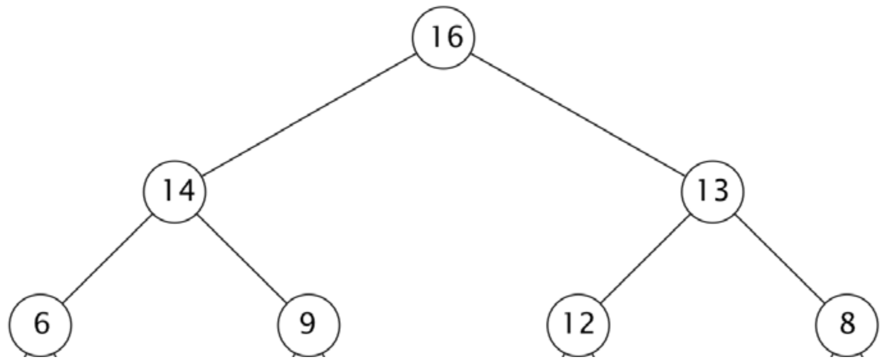


- If we want to insert 15, what will the heap like to be?
- If we want the delete the maximum, what will the heap like to be?

- *Adapt from MIT, Quiz 1, 6.006 2010 Spring.*

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.

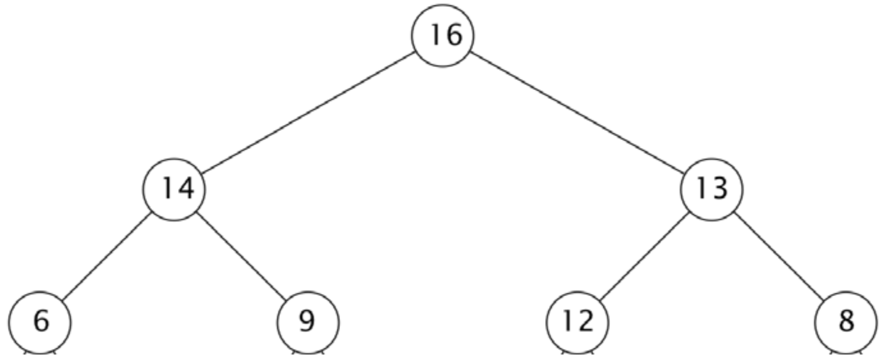


- If we want to insert 15, what will the heap like to be?
- Step 1: Transform it into array.

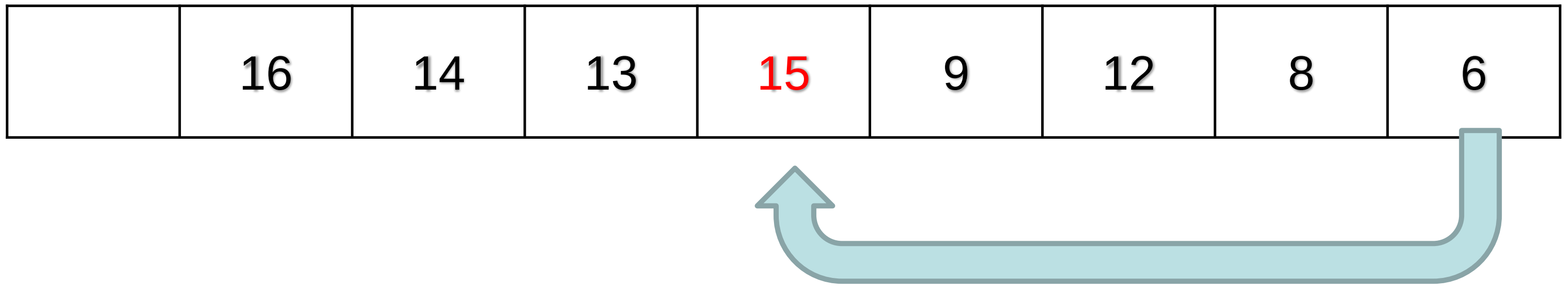
	16	14	13	6	9	12	8	15

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



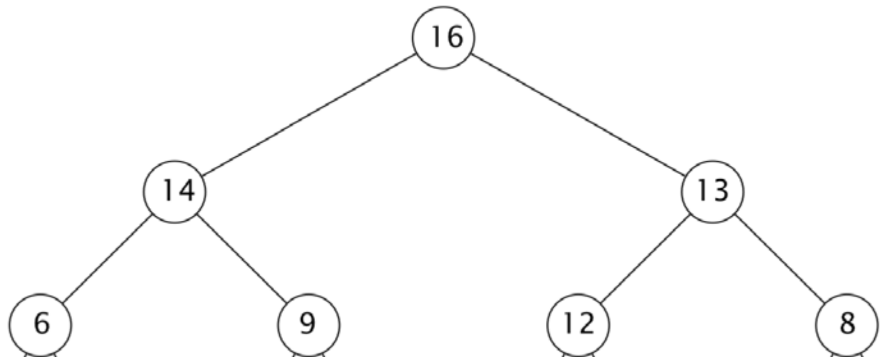
- If we want to insert 15, what will the heap like to be?
- Step 2: Compare with its parent repeatedly.



	16	14	13	15	9	12	8	6

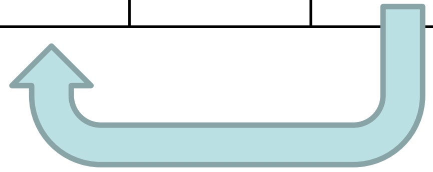
Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



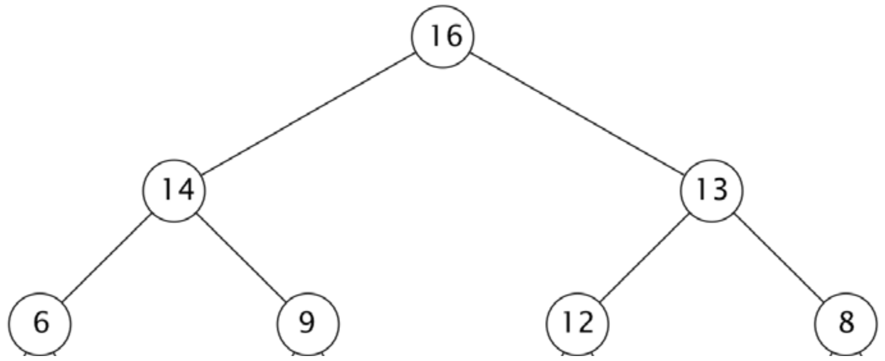
- If we want to insert 15, what will the heap like to be?
- Step 2: Compare with its parent repeatedly.

	16	15	13	14	9	12	8	6



Exercise 7 – Heap Sort

- Consider the following maximum binary heap.

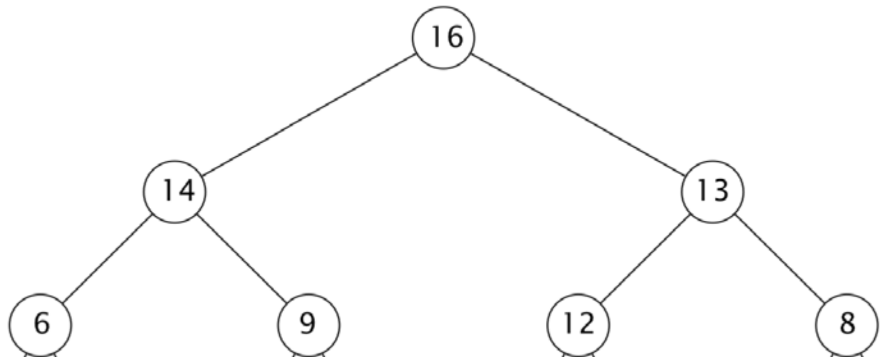


- If we want to insert 15, what will the heap like to be?
- Step 2: Compare with its parent repeatedly.

	16	15	13	14	9	12	8	6

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.

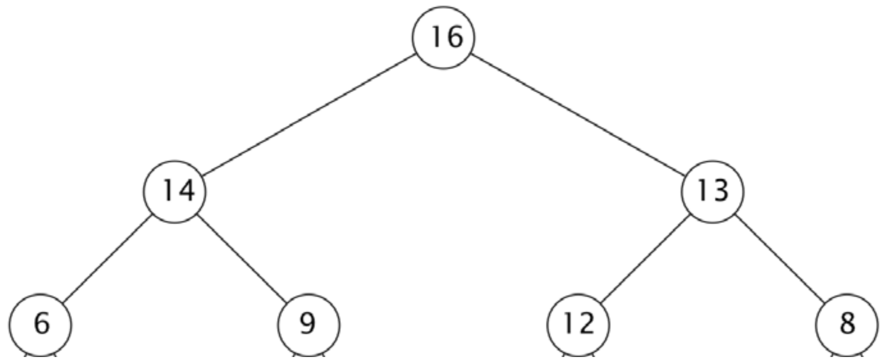


- If we want to insert 15, what will the heap like to be?
- Step 3: Stop. Transform it back to a tree.

	16	15	13	14	9	12	8	6

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.

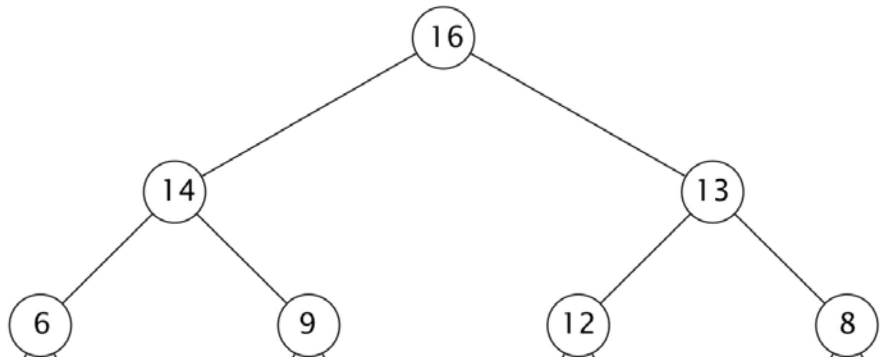


- If we want the delete the maximum, what will the heap like to be?
- Step 1: Transform it into array.

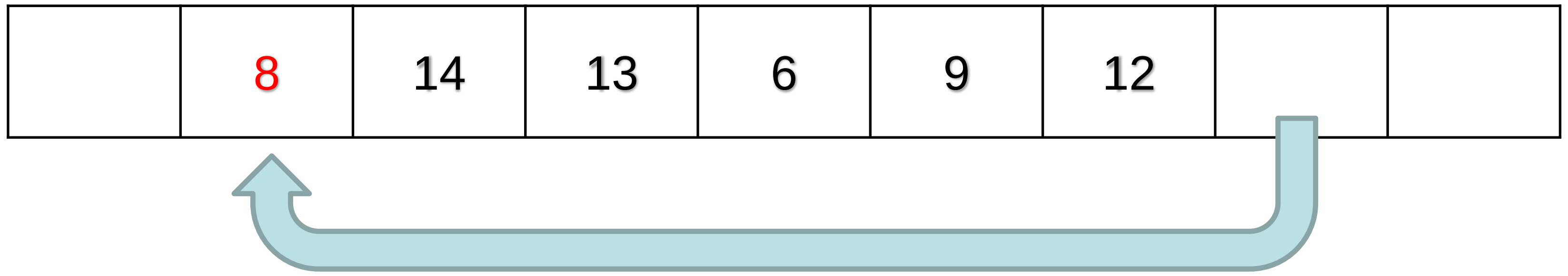
	16	14	13	6	9	12	8	15

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



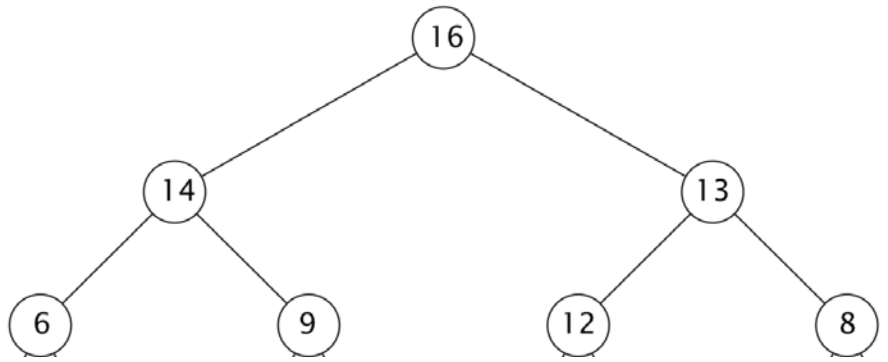
- If we want the delete the maximum, what will the heap like to be?
- Step 2: Pop the queue. Move the last one to the top



	8	14	13	6	9	12		

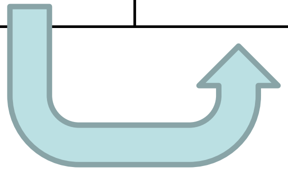
Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



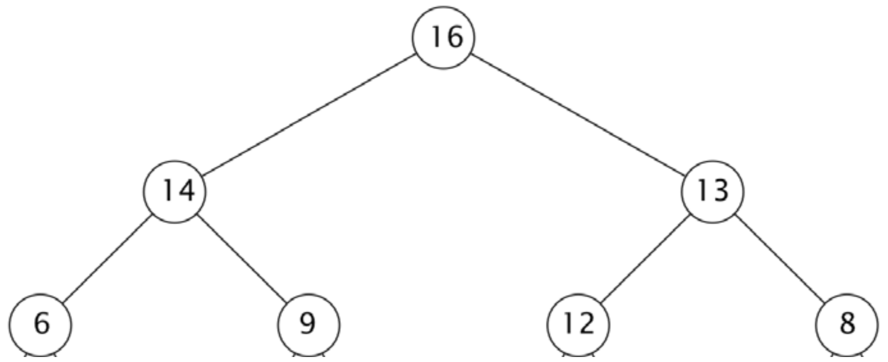
- If we want the delete the maximum, what will the heap like to be?
- Step 3: Compare with its 2 children (left first, then right, swap with maximum).

	14	8	13	6	9	12		

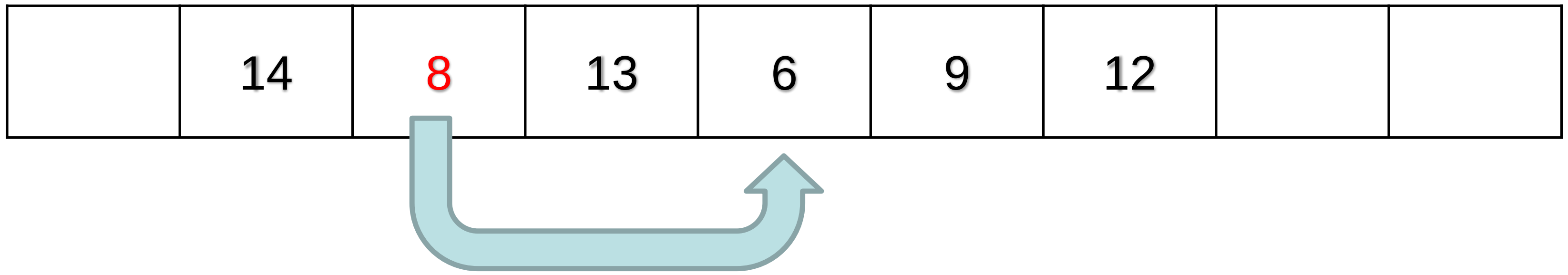


Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



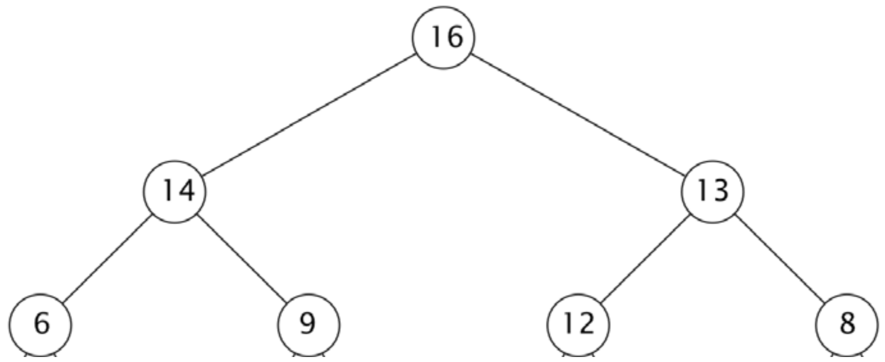
- If we want the delete the maximum, what will the heap like to be?
- Step 3: Compare with its 2 children (left first, then right, swap with maximum).



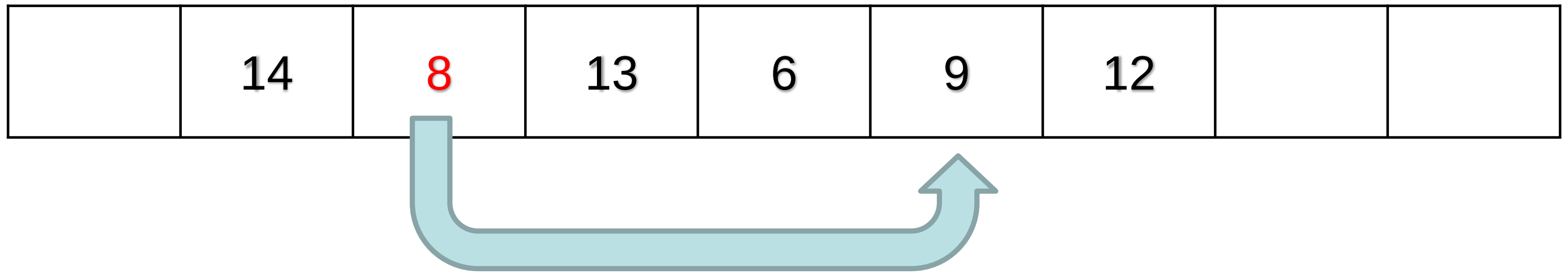
	14	8	13	6	9	12		

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



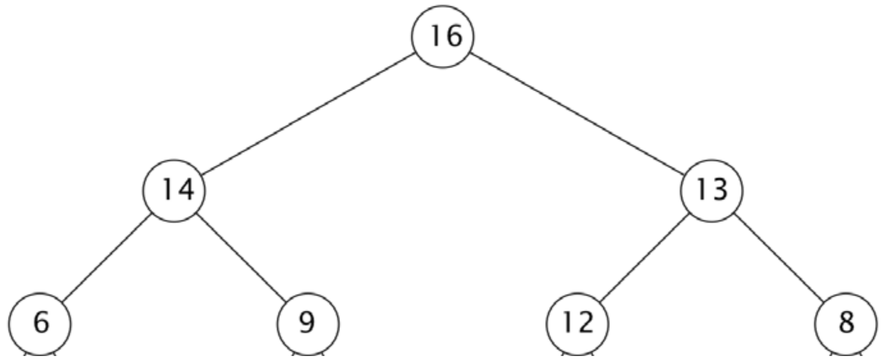
- If we want the delete the maximum, what will the heap like to be?
- Step 3: Compare with its 2 children (left first, then right, swap with maximum).



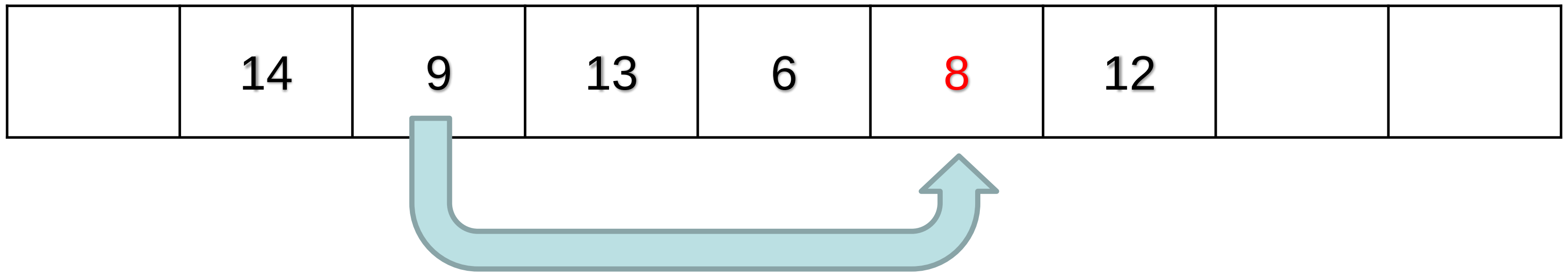
	14	8	13	6	9	12		

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



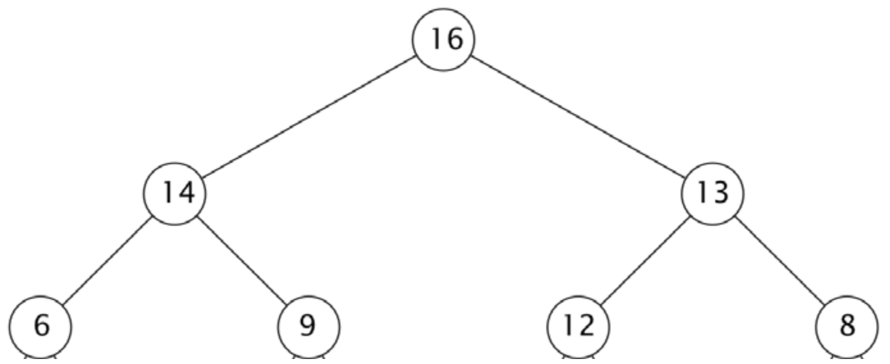
- If we want the delete the maximum, what will the heap like to be?
- Step 3: Compare with its 2 children (left first, then right, swap with maximum).



	14	9	13	6	8	12		

Exercise 7 – Heap Sort

- Consider the following maximum binary heap.



- If we want the delete the maximum, what will the heap like to be?
- Step 3: Stop. Transform it back to a tree. If you want to sort, insert 16 at the back.

	14	9	13	6	8	12		

3. Quiz