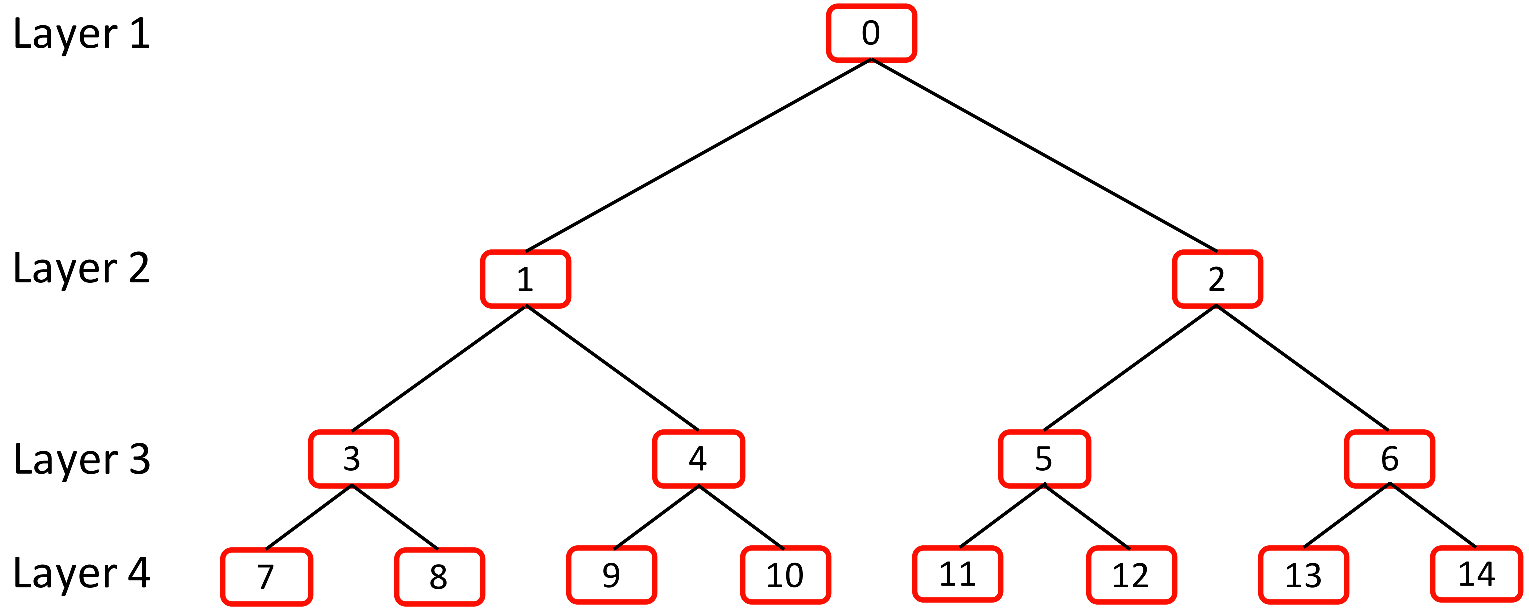
CS 313: Intermediate Data Structure
Project 2: Max-Heap Priority Queue

Viet Lai

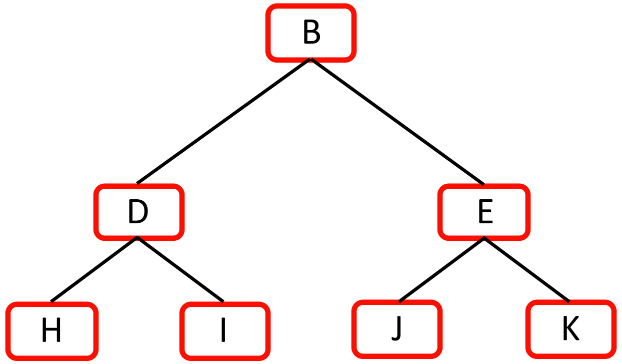
vietl@uoregon.edu

Binary tree

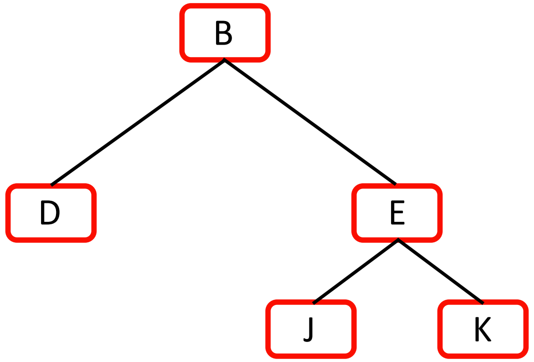
- A node has:
  - 1 parent (except the root)
  - At most 1 left child
  - At most 1 right child
- Number of node: N
- Depth of a perfect tree: $log_2(N+1)$

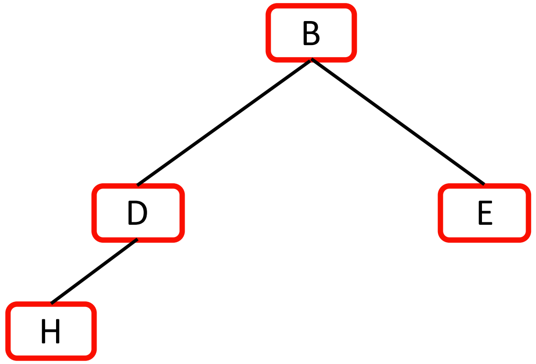


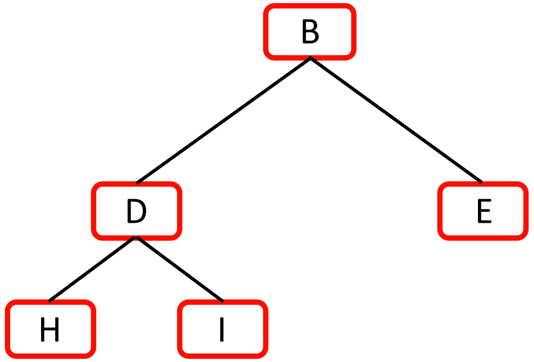
Full-Complete-Perfect Binary Tree



- Full tree:
  - Each node has either 0 or 2 children
- Complete tree:
  - All of the levels of a binary tree are entirely filled
  - Except for the last level
    - Can contain 1 or 2 children nodes
    - Is **filled from the left**
- Perfect tree:
  - All interior nodes have two children
  - All leaves have the same depth

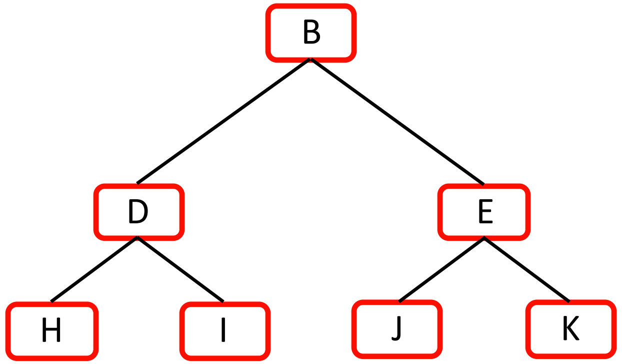




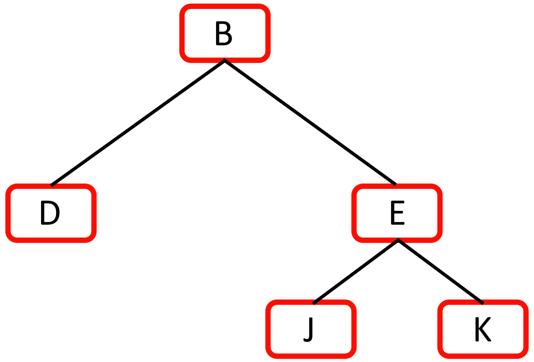


Full-Complete-Perfect Binary Tree

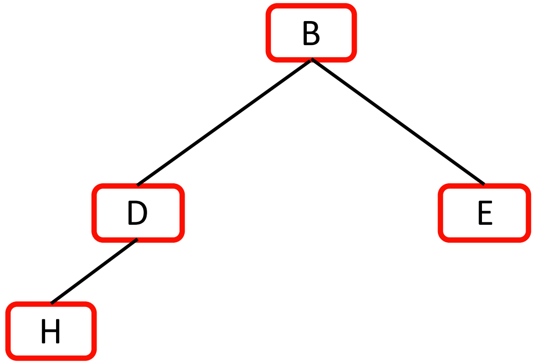
- Full tree:
  - Each node has either 0 or 2 children
- Complete tree:
  - All of the levels of a binary tree are entirely filled
  - Except for the last level
    - Can contain 1 or 2 children nodes
    - Is **filled from the left**
- Perfect tree:
  - All interior nodes have two children
  - All leaves have the same depth



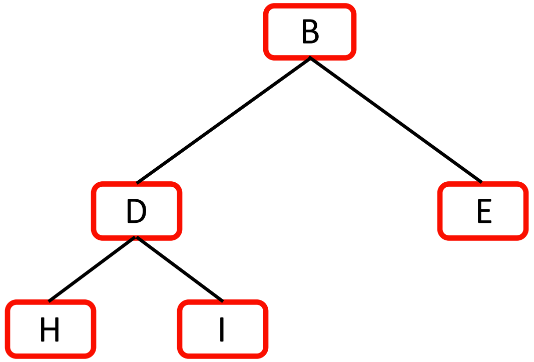
Full: Yes
Complete: Yes
Perfect: Yes



Full: Yes
Complete: No
Perfect: No

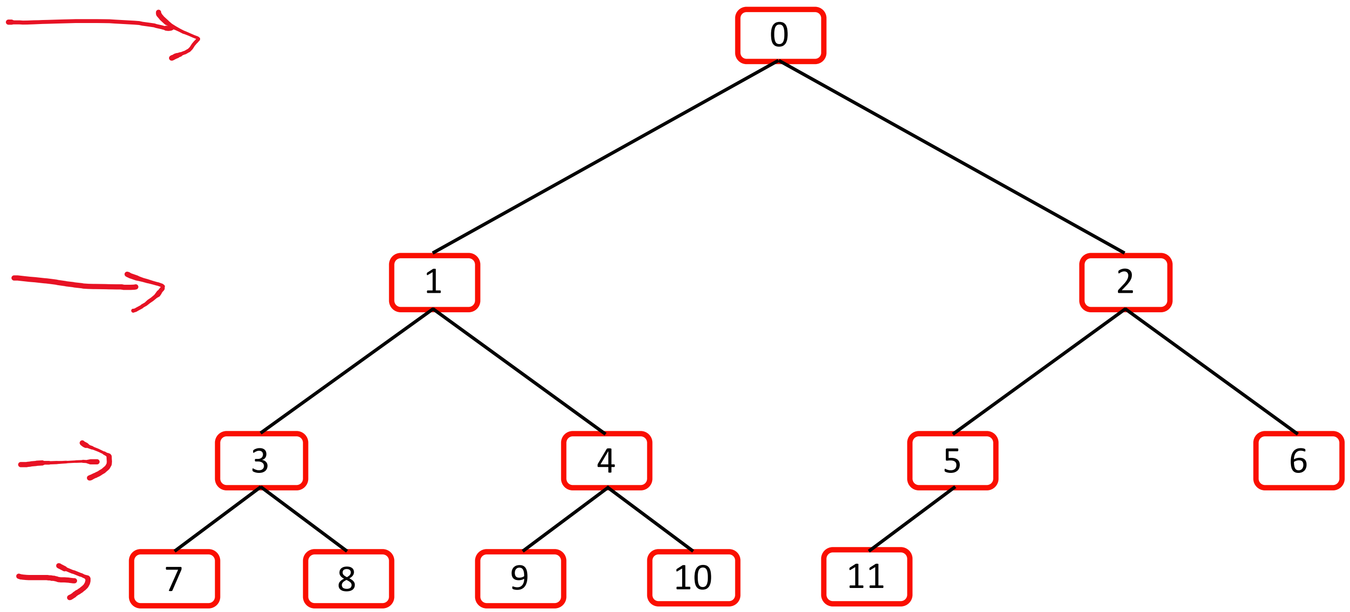


Full: No
Complete: Yes
Perfect: No



Full: Yes
Complete: Yes
Perfect: No

Linearize a complete tree



Read order:

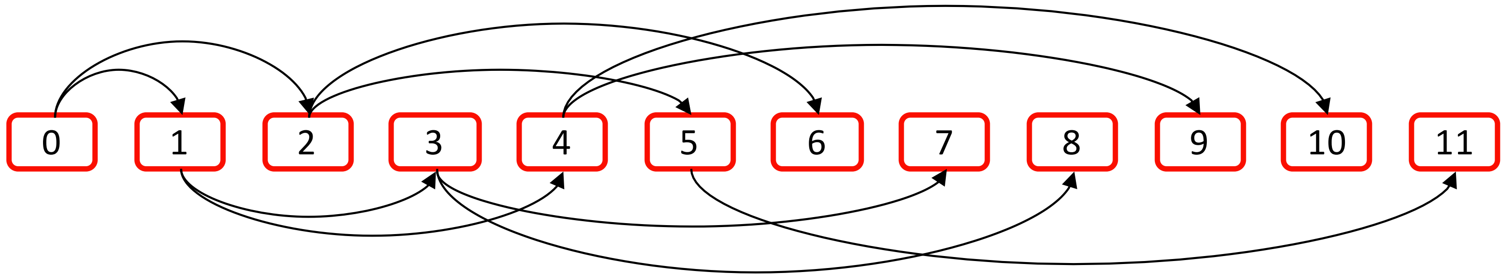
- Left - right
- Top - down

A list as a complete tree

- Parent: (index-1)//2
- Left child: 2*index + 1
- Right child: 2*index + 2

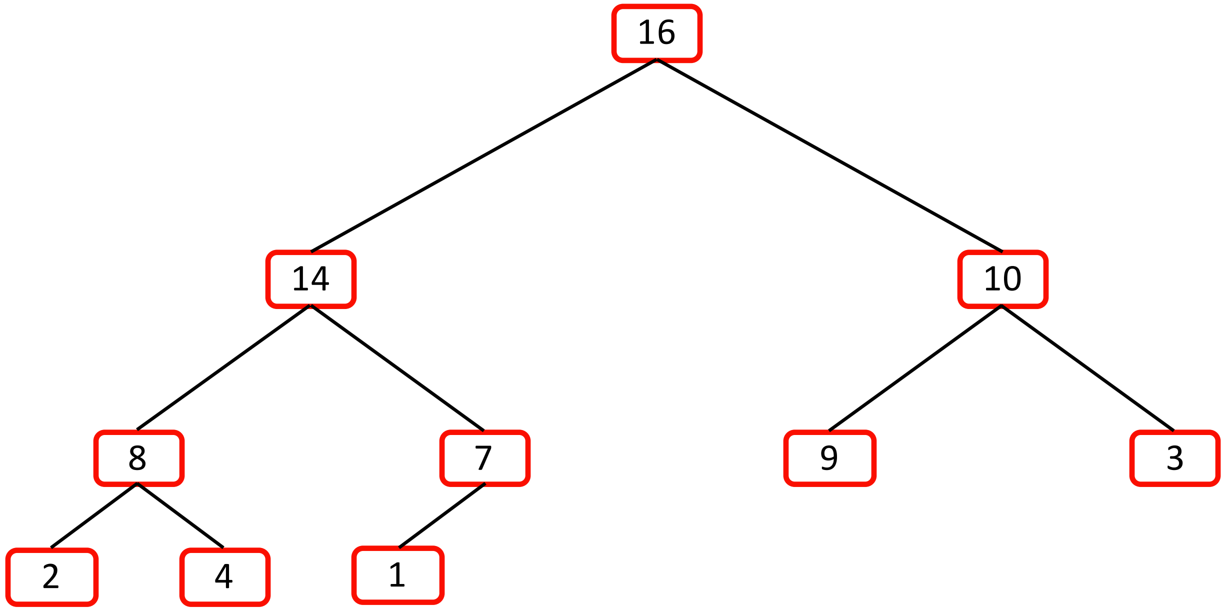
Advantages:

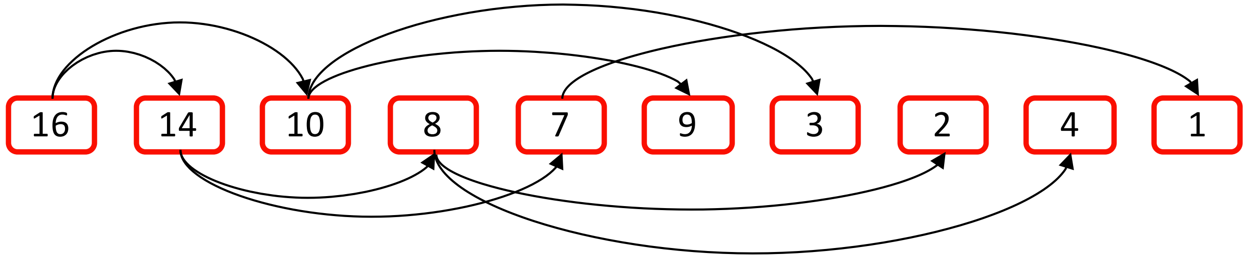
- Tree-like structure
- Fast access



Heap

- Heap is a complete tree
- Heap property
  - Max-heap:
    - Parent's value >= Children's values
    - Aka: parent is the max
    - **Root** is the max of the whole heap
  - Min-heap:
    - Parent's value <= Children's values
    - Aka: parent is the min
    - **Root** is the min of the whole heap





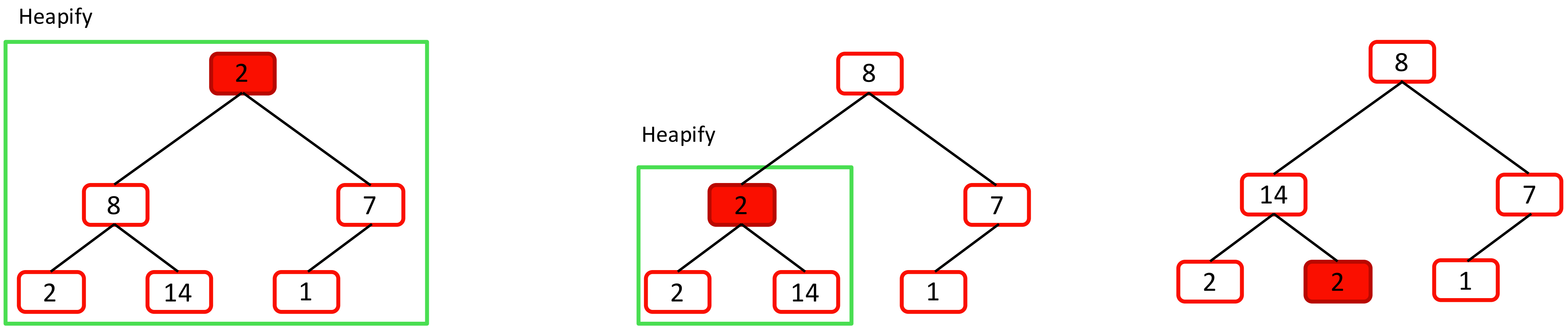
A max heap

Maintain the max-heap property

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Heapify for a heap "A", starting at the index "i"
Get the index of the left child
Get the index of the left child
Compare the node with its left child
Largest is the left if left > current node
Largest is the current node if left <= current node

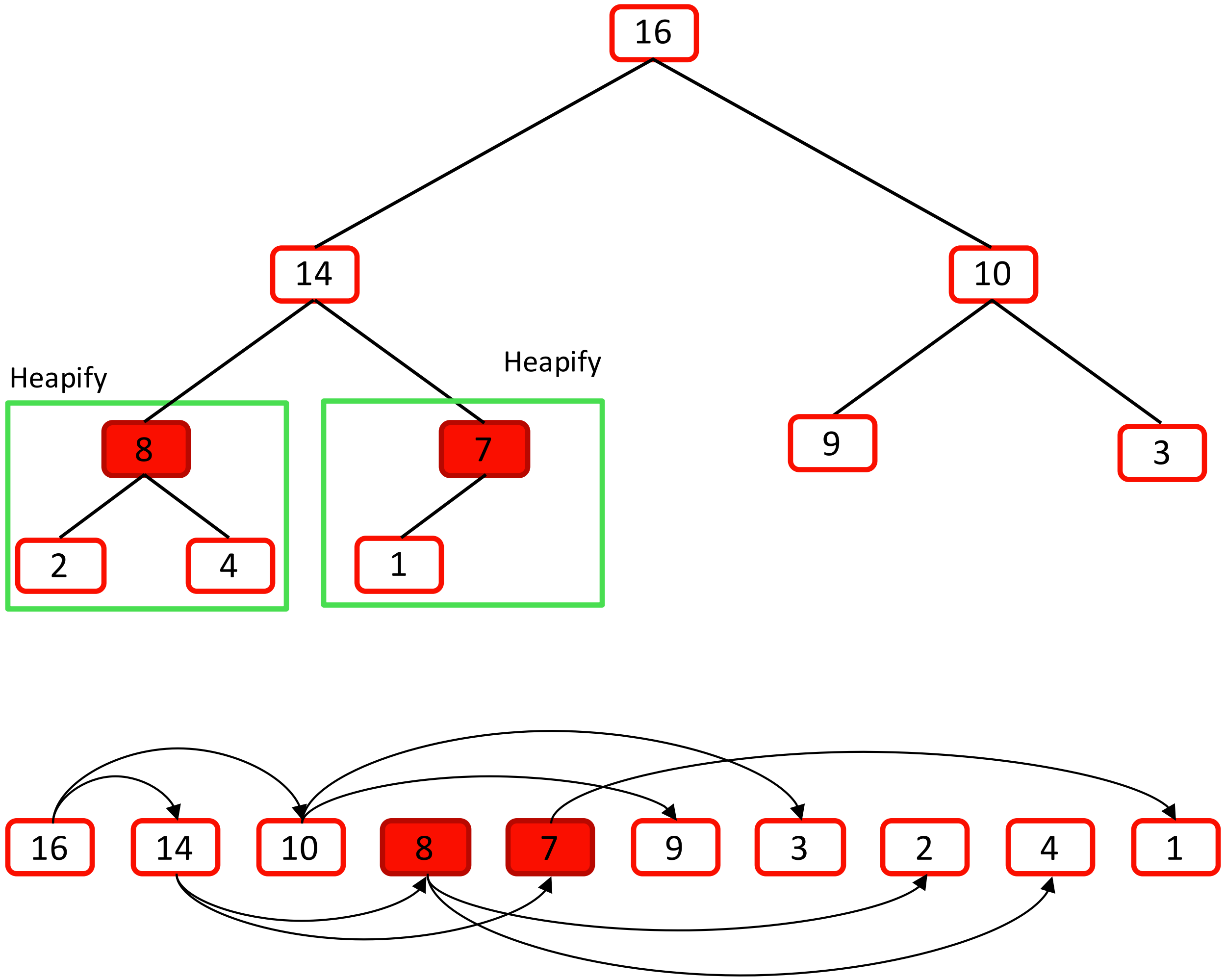
If the largest is not the current node
Swap the current node and the largest (either left or right)
Recursively heapify the subtree that has just been swapped



Building a heap from a given list (1)

A bottom up method

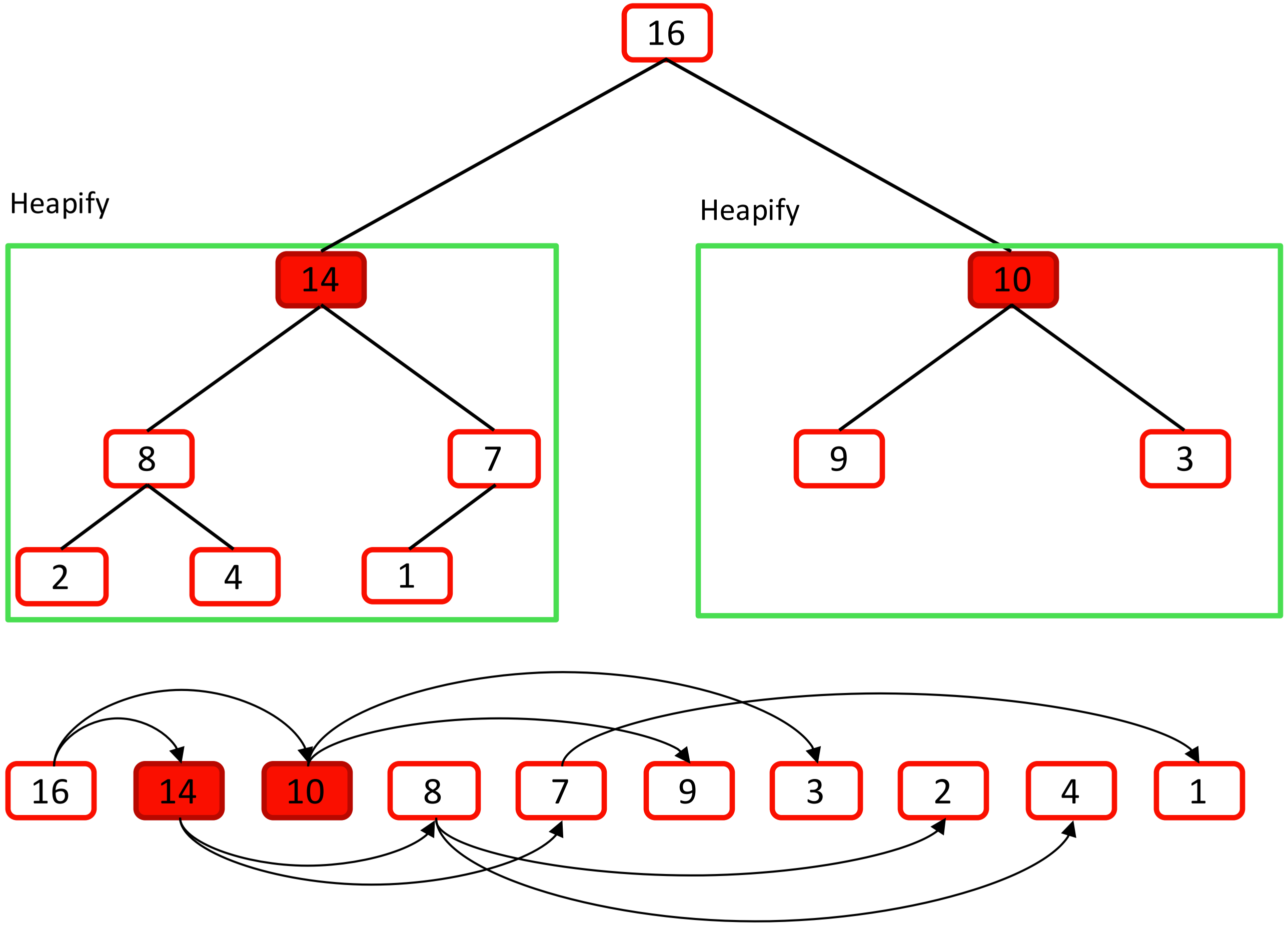
- Try to maintain the heap property from the bottom up



Building a heap from a given list (2)

A bottom up method

- Try to maintain the heap property from the bottom up

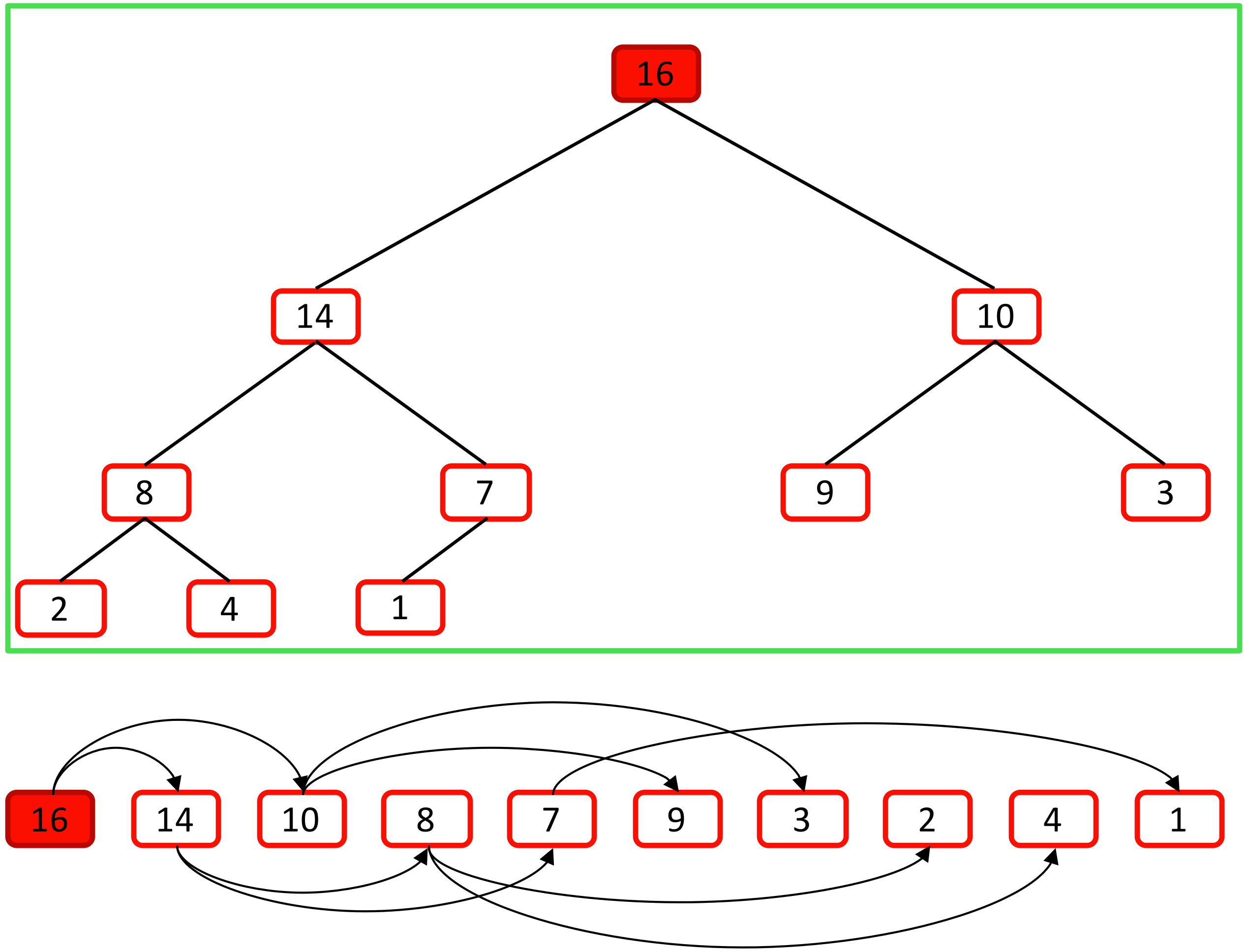


Building a heap from a given list (3)

A bottom up method

- Try to maintain the heap property from the bottom up

Heapify



Building a heap from a given list (4)

While don’t we heapify the leaf nodes?

- *Leaf nodes does not have child, so heapifying is not needed*

How to implement bottom up using linearized heap?

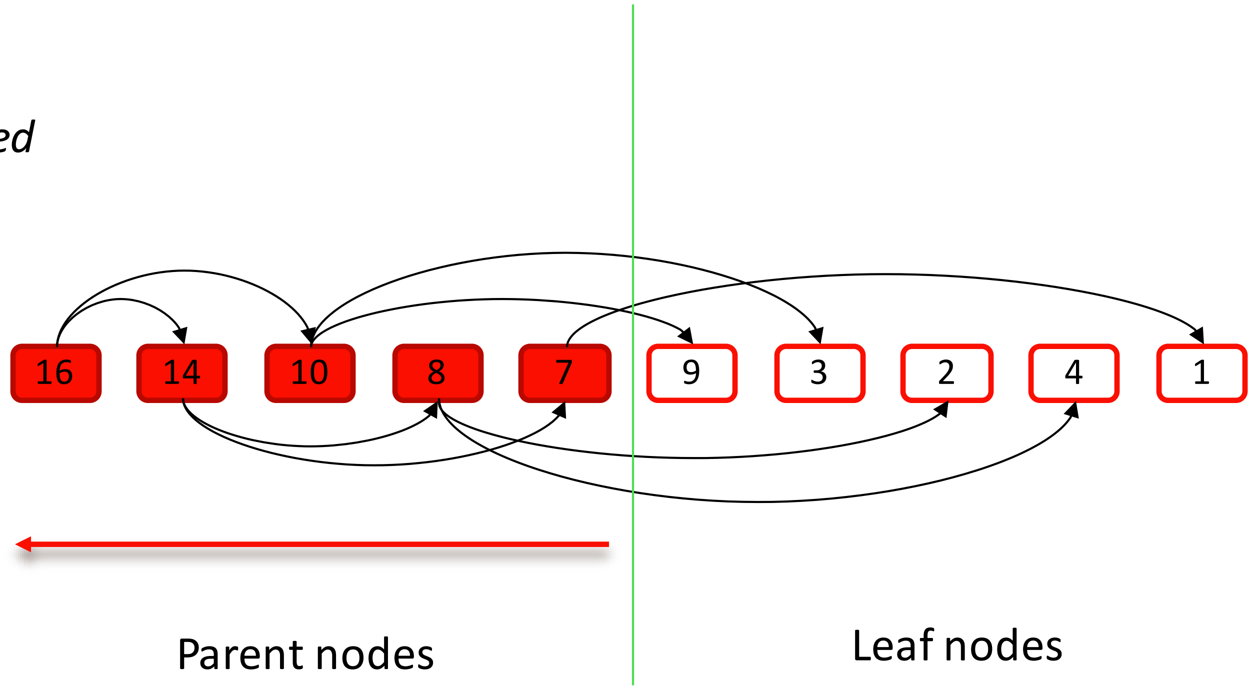
- *Loop backward (from the end to the beginning)*

What are the starting/ending indices?

- *Starting: length//2*
- *Ending: 0*

Why length//2?

- $2^0 + 2^1 + \dots + 2^{L-1} = 2^L - 1 < 2^L$




```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Build a max heap from a given list

- Assign the size of the heap
- Loop from the middle of the list to the left
- Heapify