CMSC 132: Object-Oriented Programming II

Heaps & Priority Queues

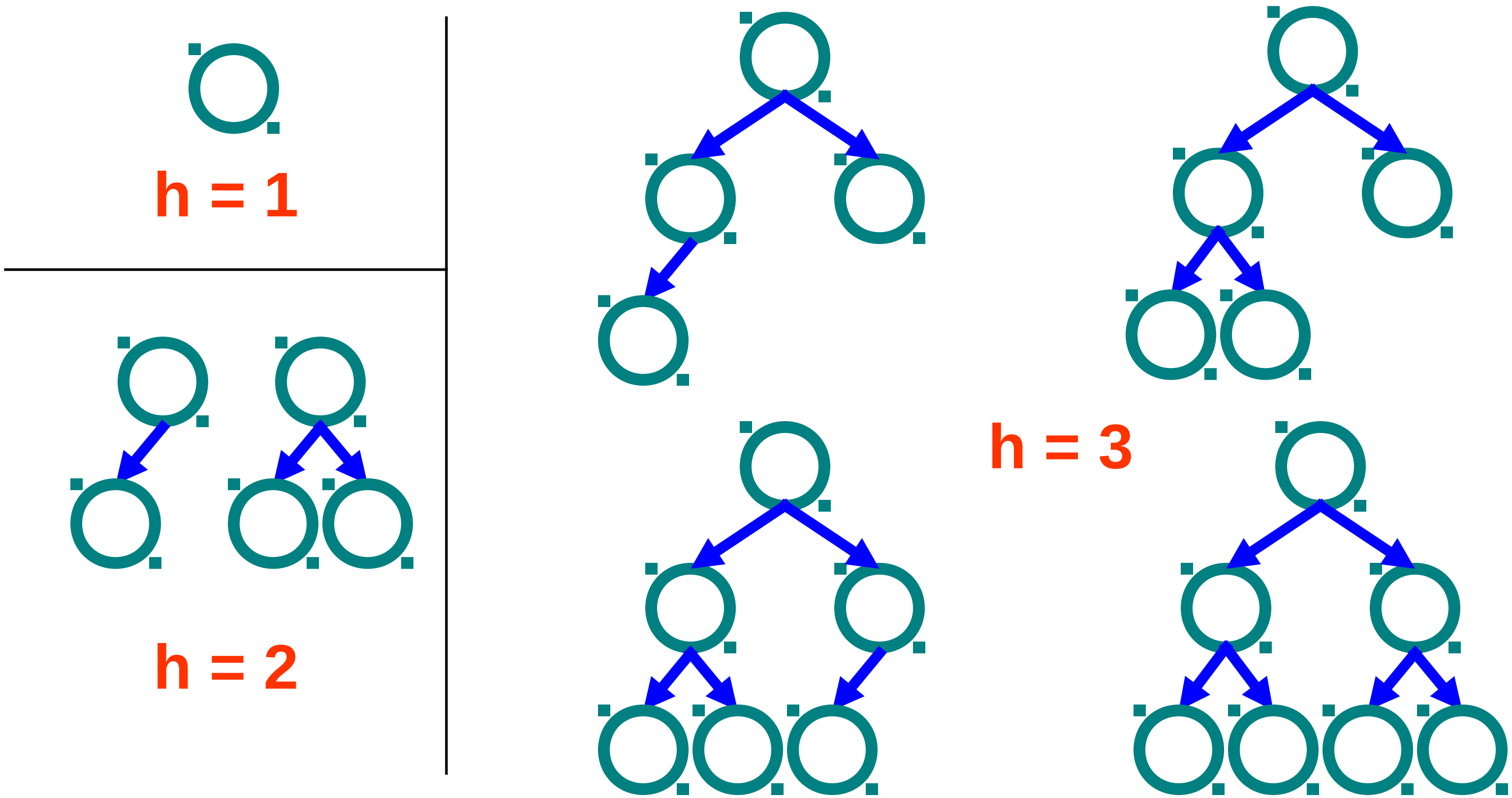
Department of Computer Science
University of Maryland, College Park

Overview

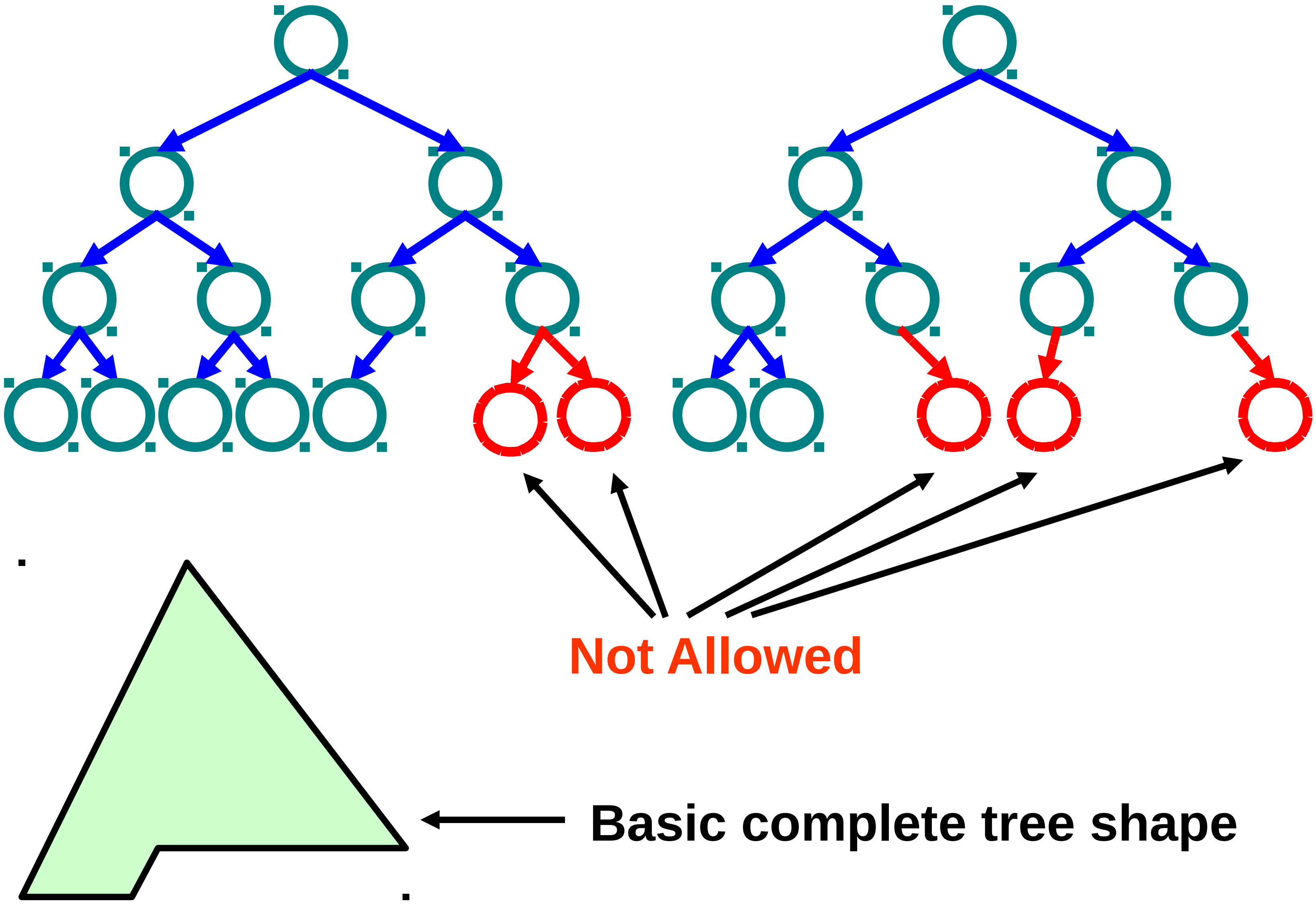
- Binary trees
  - Complete
- Heaps
  - Insert
  - getSmallest
- Heap applications
  - Heapsort
  - Priority queues

Complete Binary Trees

- An binary tree (height h) where
  - Perfect tree to level h-1
  - Leaves at level h are as far left as possible

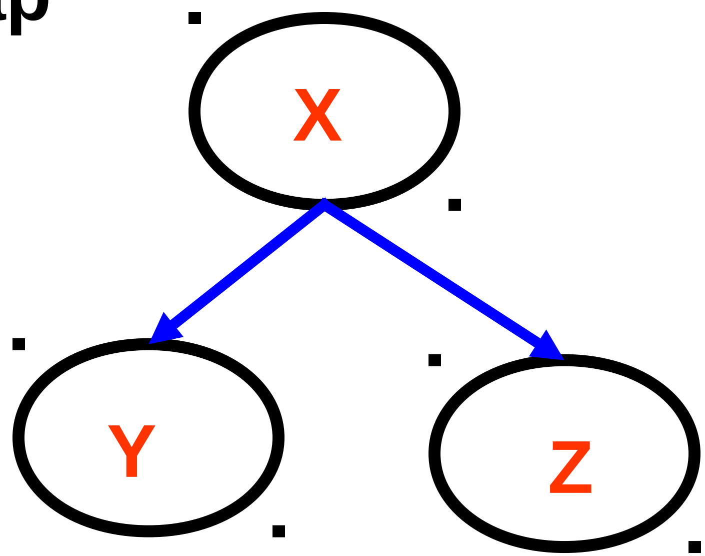


Complete Binary Trees

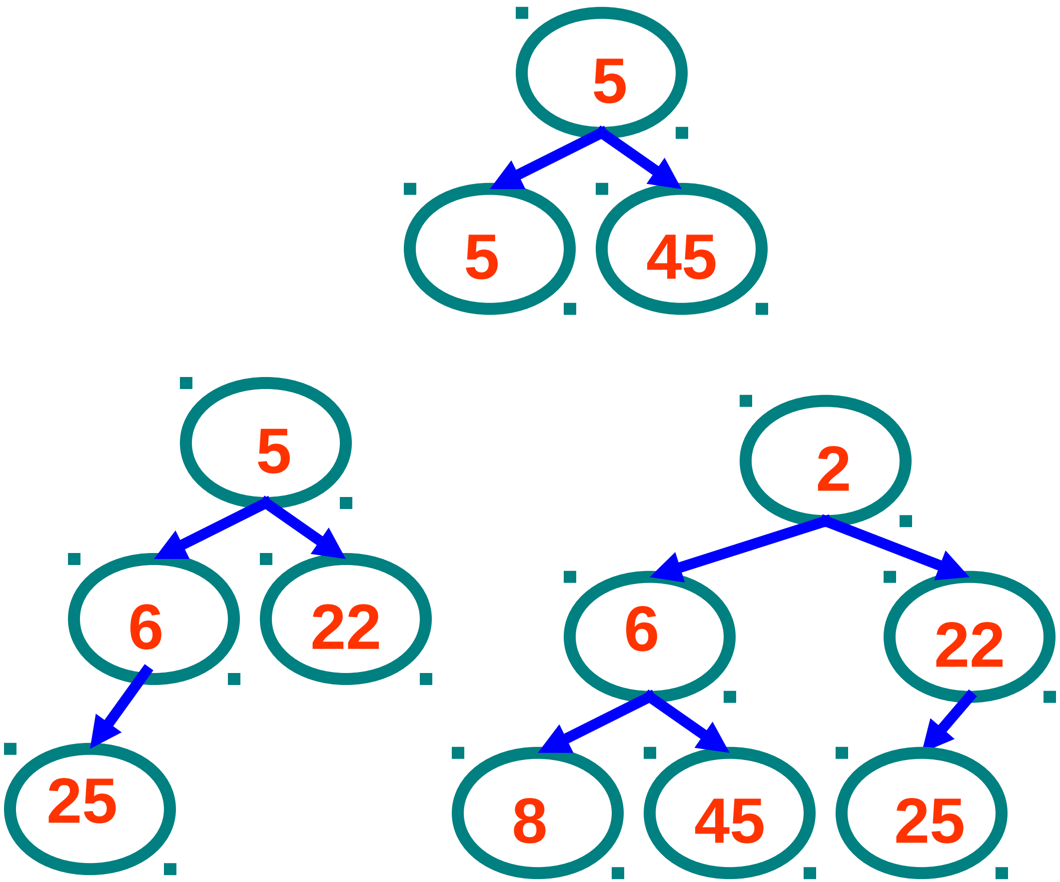


Heaps

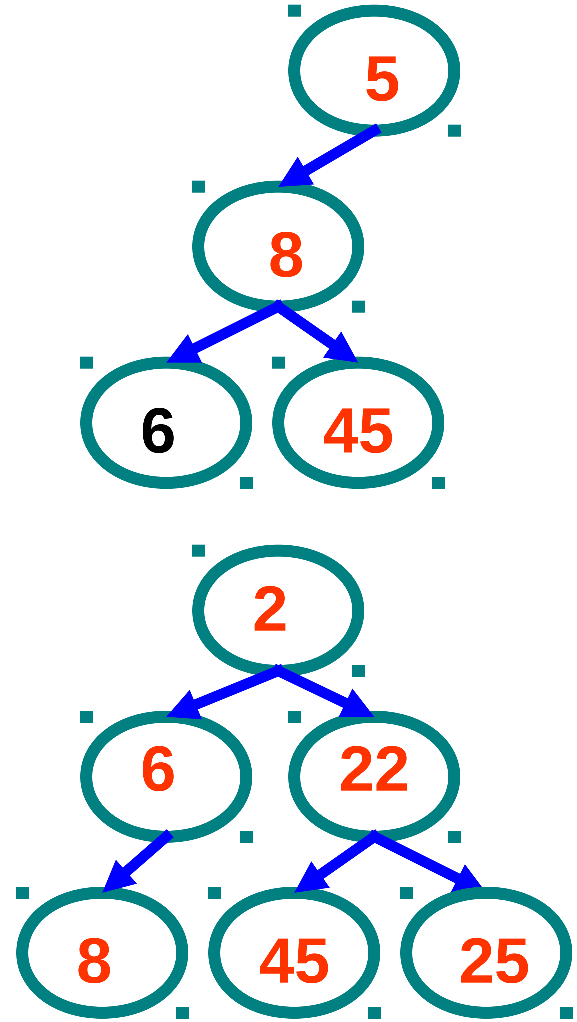
- Two key properties
  - Complete binary tree
  - Value at node
    - Smaller than or equal to values in subtrees
- Example heap
  - $X \le Y$
  - $X \le Z$



Heap & Non-heap Examples



Heaps



Non-heaps

Heap Properties

- Heaps are balanced trees
  - Height = $\log_2(n) = O(\log(n))$

- Can find smallest element easily
  - Always at top of heap!

- Can organize heap to find maximum value
  - Value at node larger than values in subtrees
  - Heap can track either min or max, but not both

Heap

- **Key operations**
  - **Insert (X)**
  - **getSmallest ( )**

- **Key applications**
  - **Heapsort**
  - **Priority queue**

Heap Operations – Insert(X)

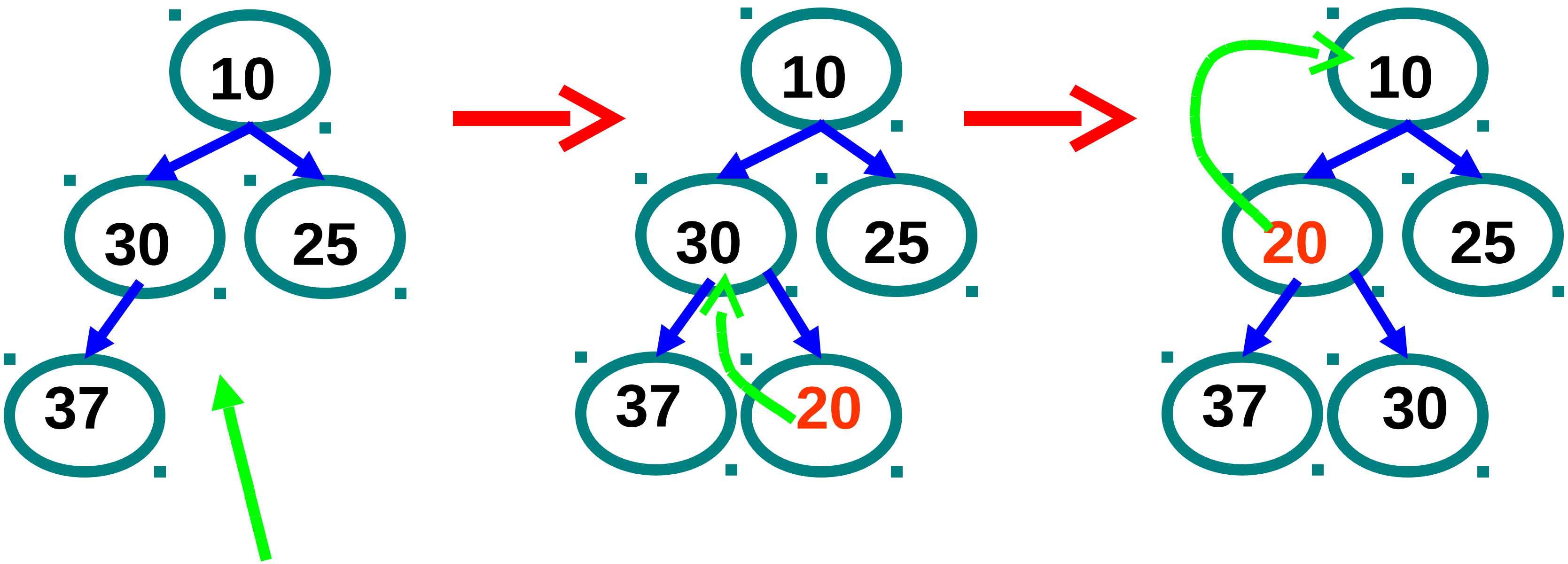
- **Algorithm**
  1. **Add X to end of tree**
  2. **While (X < parent)**

     Swap X with parent // X bubbles up tree

- **Complexity**
  - **# of swaps proportional to height of tree**
  - **O(log(n))**

Heap Insert Example

- Insert (20)



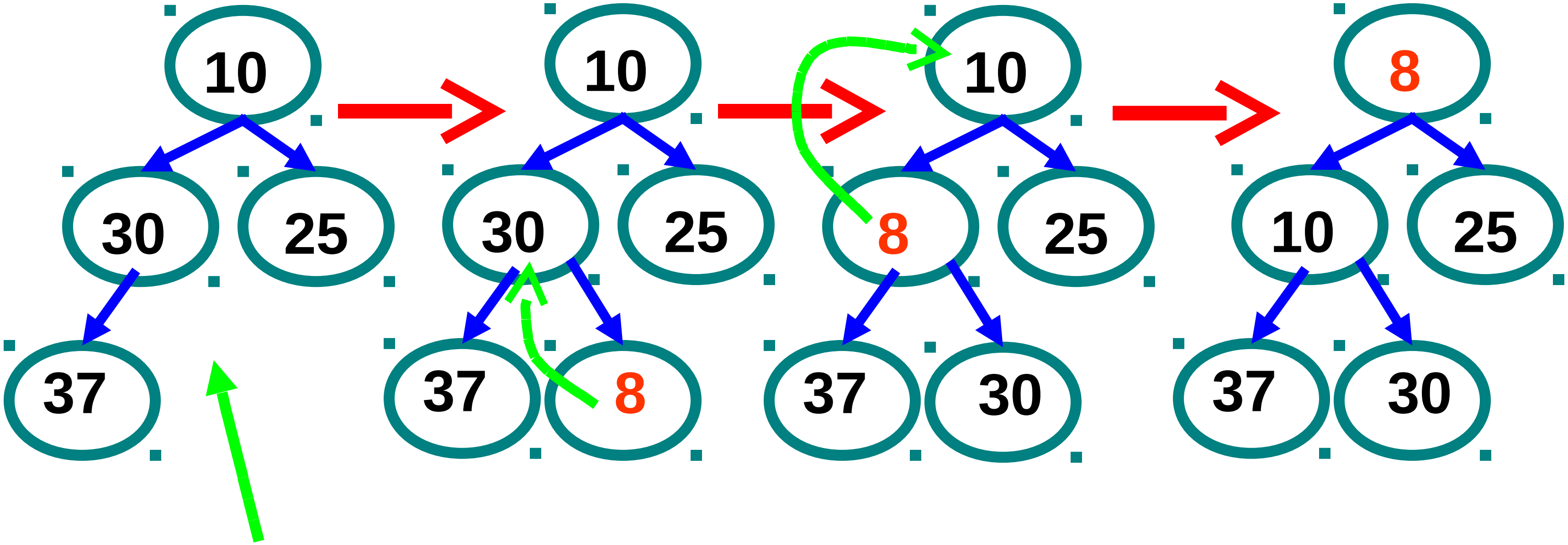
1) Insert to end of tree

2) Compare to parent, swap if parent key larger

3) Insert complete

Heap Insert Example

- Insert (8)



1) Insert to end of tree

2) Compare to parent, swap if parent key larger

3) Insert complete

Heap Operation – getSmallest()

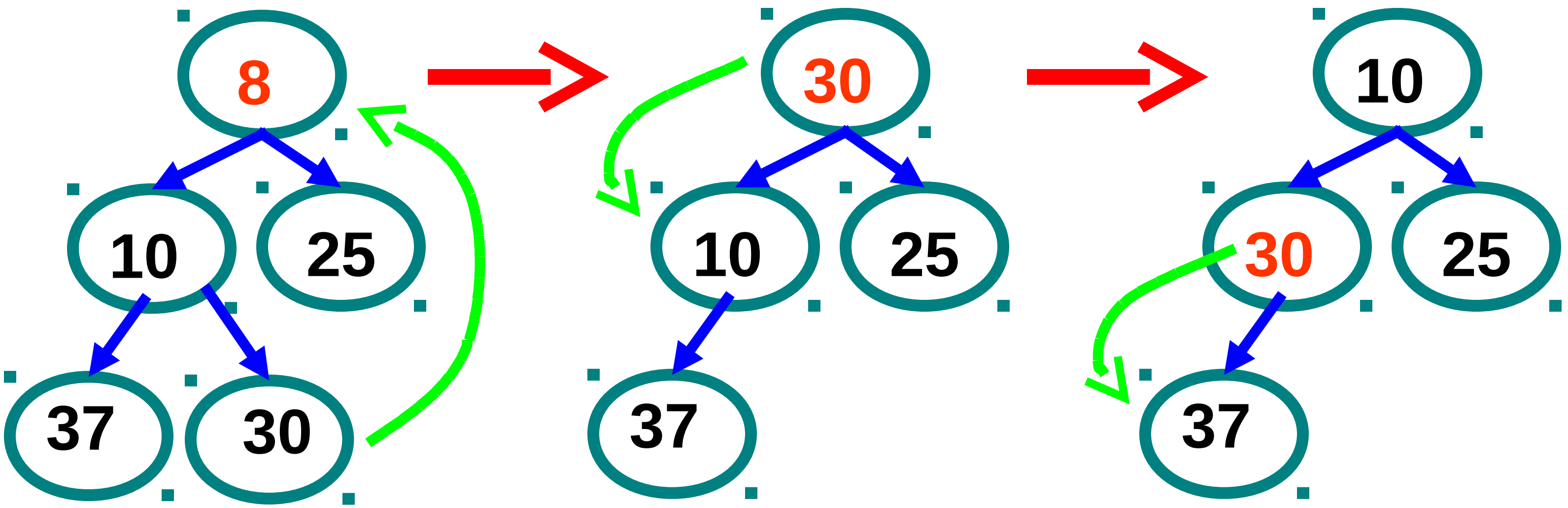
- **Algorithm**
  1. **Get smallest node at root**
  2. **Replace root with X at end of tree**
  3. **While (X > child)**

     Swap X with smallest child   // X drops down tree
  4. **Return smallest node**

- **Complexity**
  - **# swaps proportional to height of tree**
  - **O(log(n))**

Heap GetSmallest Example

- **getSmallest ()**



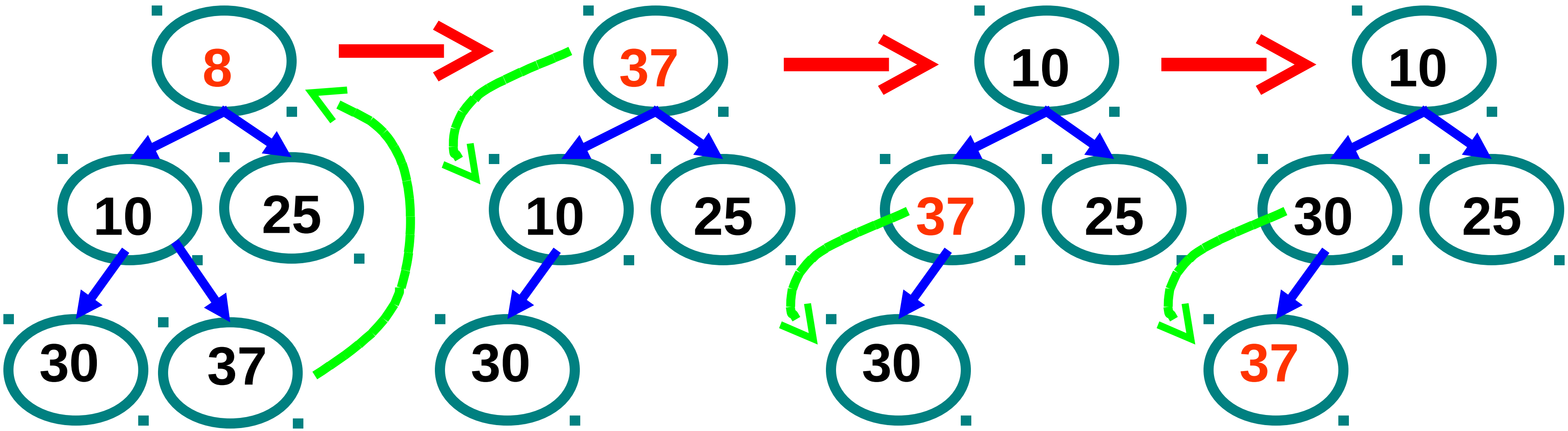
1) Replace root with end of tree

2) Compare node to children, if larger swap with smallest child

3) Repeat swap if needed

Heap GetSmallest Example

- **getSmallest ()**



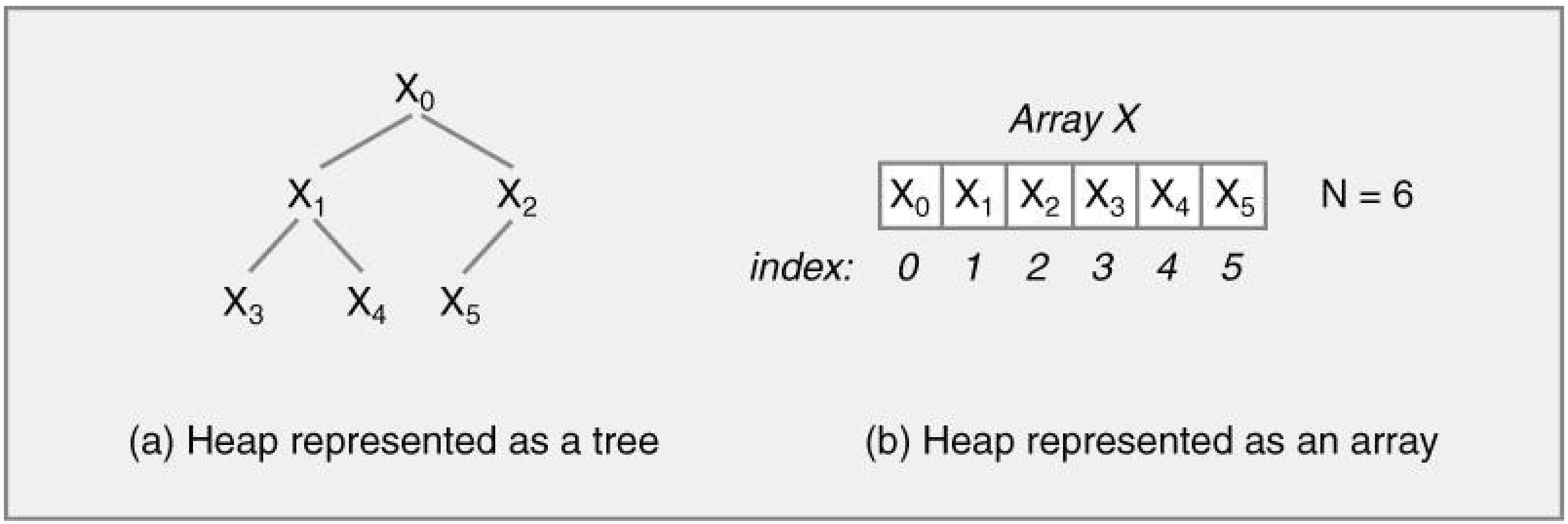
1) Replace root with end of tree

2) Compare node to children, if larger swap with smallest child

3) Repeat swap if needed

Heap Implementation

- **Can implement heap as array**
  - **Store nodes in array elements**
  - **Assign location (index) for elements using formula**

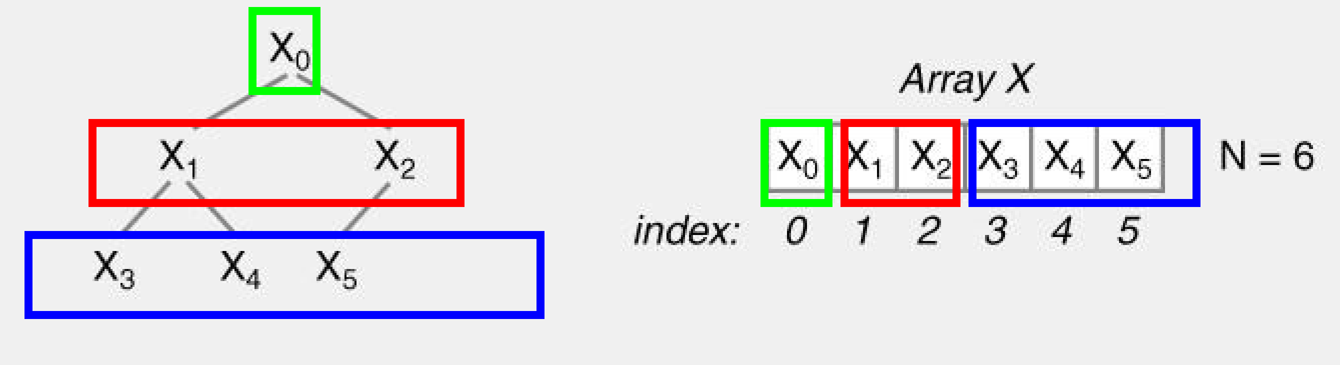


Heap Implementation

- Observations
  - Compact representation
  - Edges are implicit (no storage required)
  - Works well for complete trees (no wasted space)

Heap Implementation

- **Calculating node locations**
  - **Array index i starts at 0**
  - **Parent(i) = $\lfloor (i - 1) / 2 \rfloor$**
  - **LeftChild(i) = $2 \times i + 1$**
  - **RightChild(i) = $2 \times i + 2$**

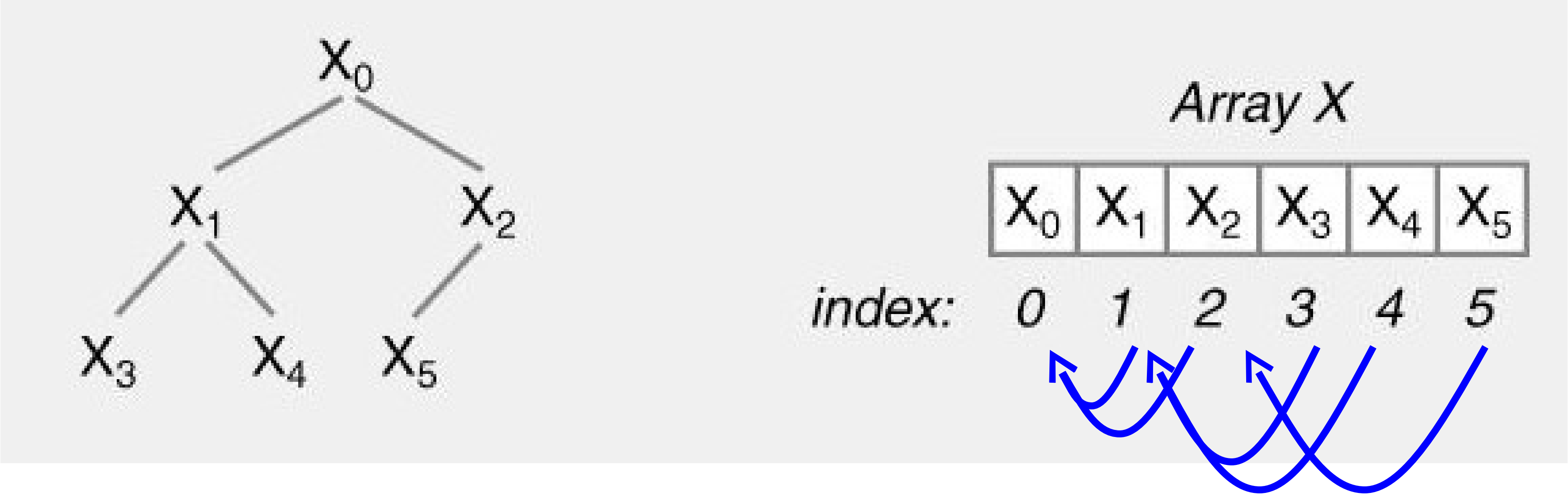


(a) Heap represented as a tree

(b) Heap represented as an array

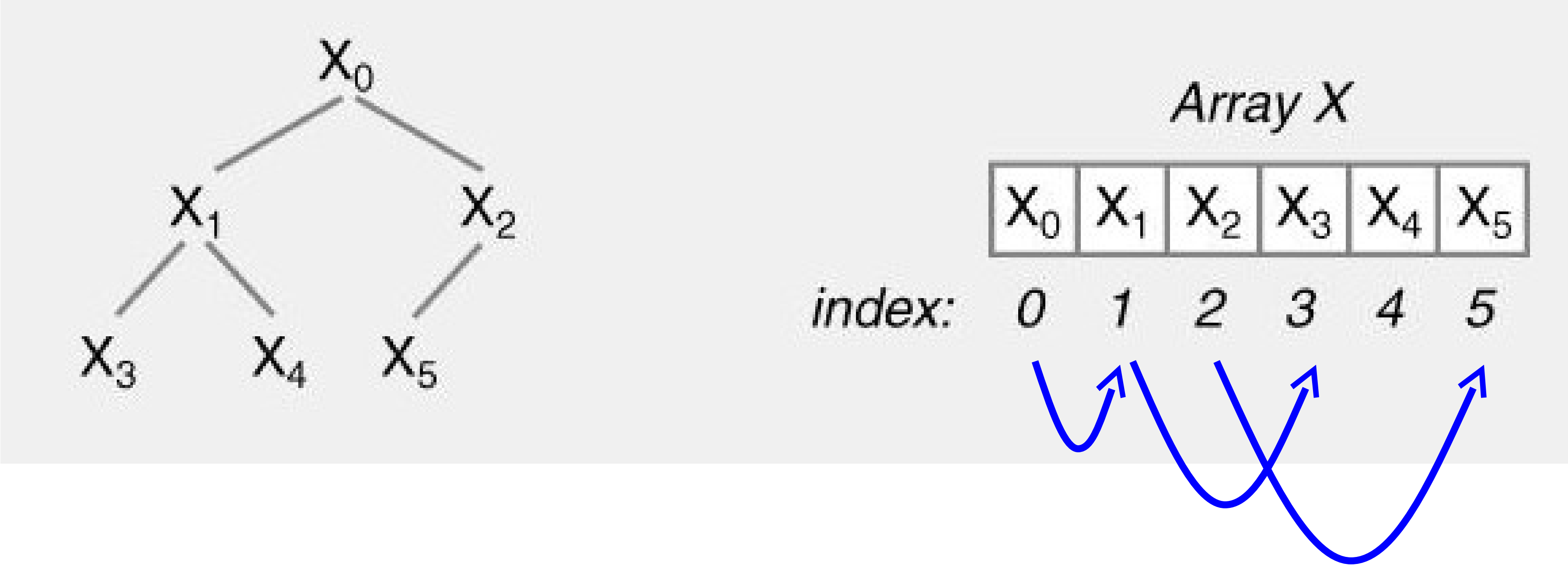
Heap Implementation

- **Example**
  - **Parent(1) = ⌊ (1 – 1) / 2 ⌋ = ⌊ 0 / 2 ⌋ = 0**
  - **Parent(2) = ⌊ (2 – 1) / 2 ⌋ = ⌊ 1 / 2 ⌋ = 0**
  - **Parent(3) = ⌊ (3 – 1) / 2 ⌋ = ⌊ 2 / 2 ⌋ = 1**
  - **Parent(4) = ⌊ (4 – 1) / 2 ⌋ = ⌊ 3 / 2 ⌋ = 1**
  - **Parent(5) = ⌊ (5 – 1) / 2 ⌋ = ⌊ 4 / 2 ⌋ = 2**



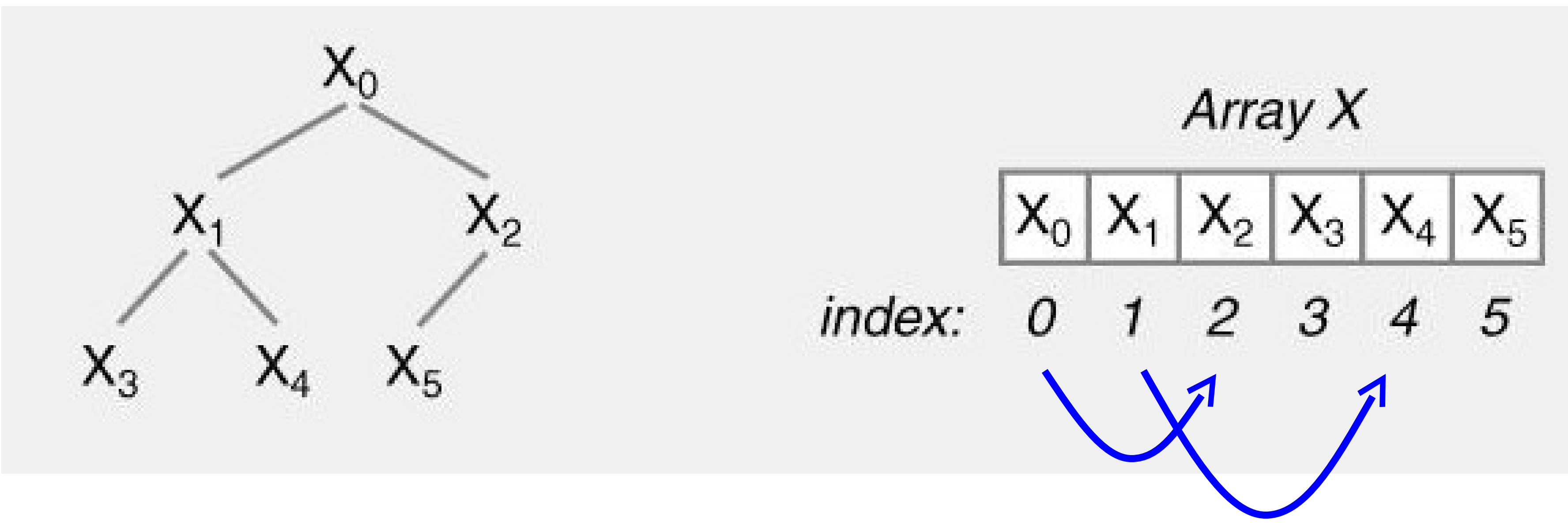
Heap Implementation

- **Example**
  - **LeftChild(0) = 2 × 0 +1 = 1**
  - **LeftChild(1) = 2 × 1 +1 = 3**
  - **LeftChild(2) = 2 × 2 +1 = 5**



Heap Implementation

- **Example**
  - **RightChild(0) = 2 × 0 +2 = 2**
  - **RightChild(1) = 2 × 1 +2 = 4**



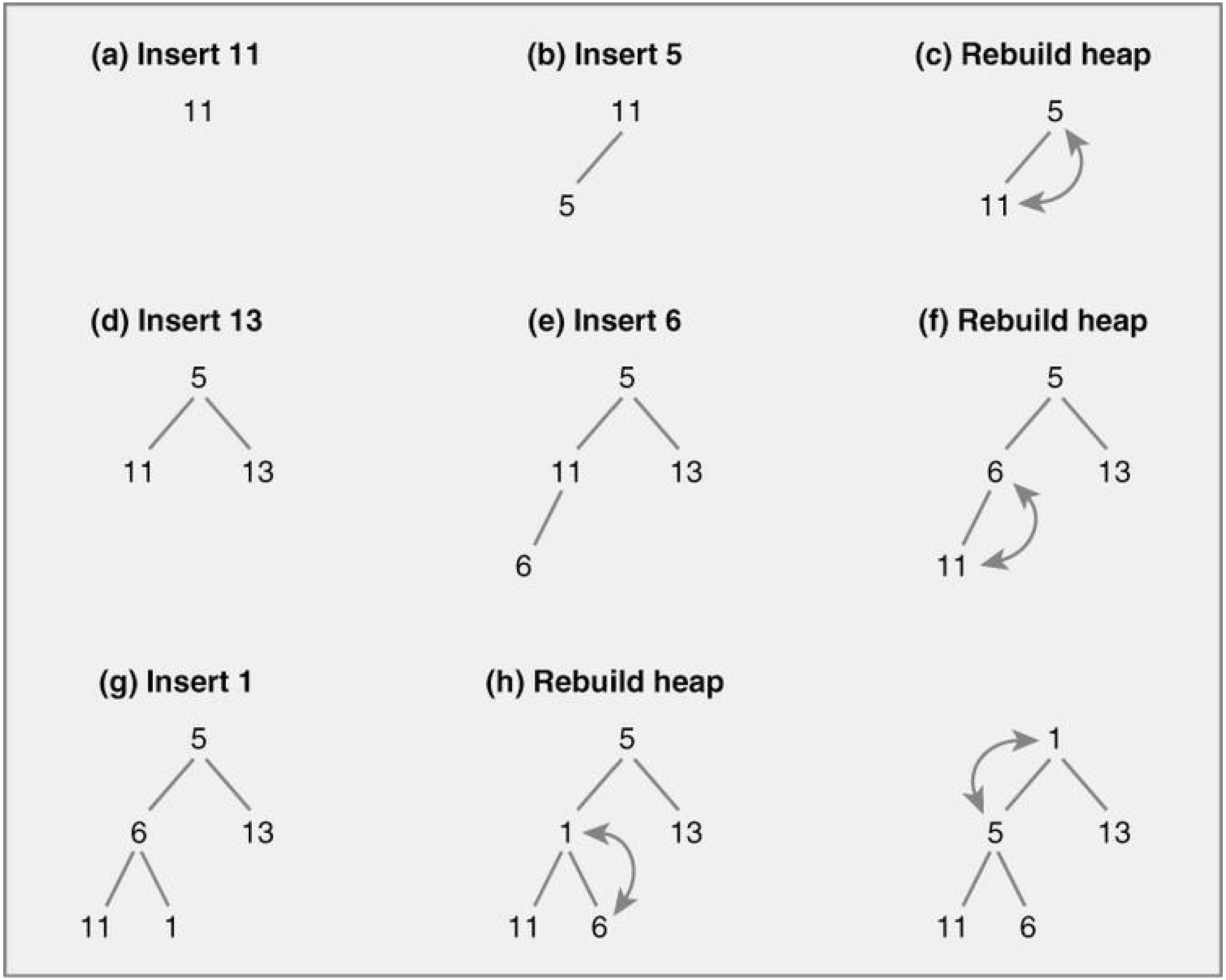
Heap Application – Heapsort

- **Use heaps to sort values**
  - **Heap keeps track of smallest element in heap**
- **Algorithm**
  1. **Create heap**
  2. **Insert values in heap**
  3. **Remove values from heap (in ascending order)**
- **Complexity**
  - **O(nlog(n))**

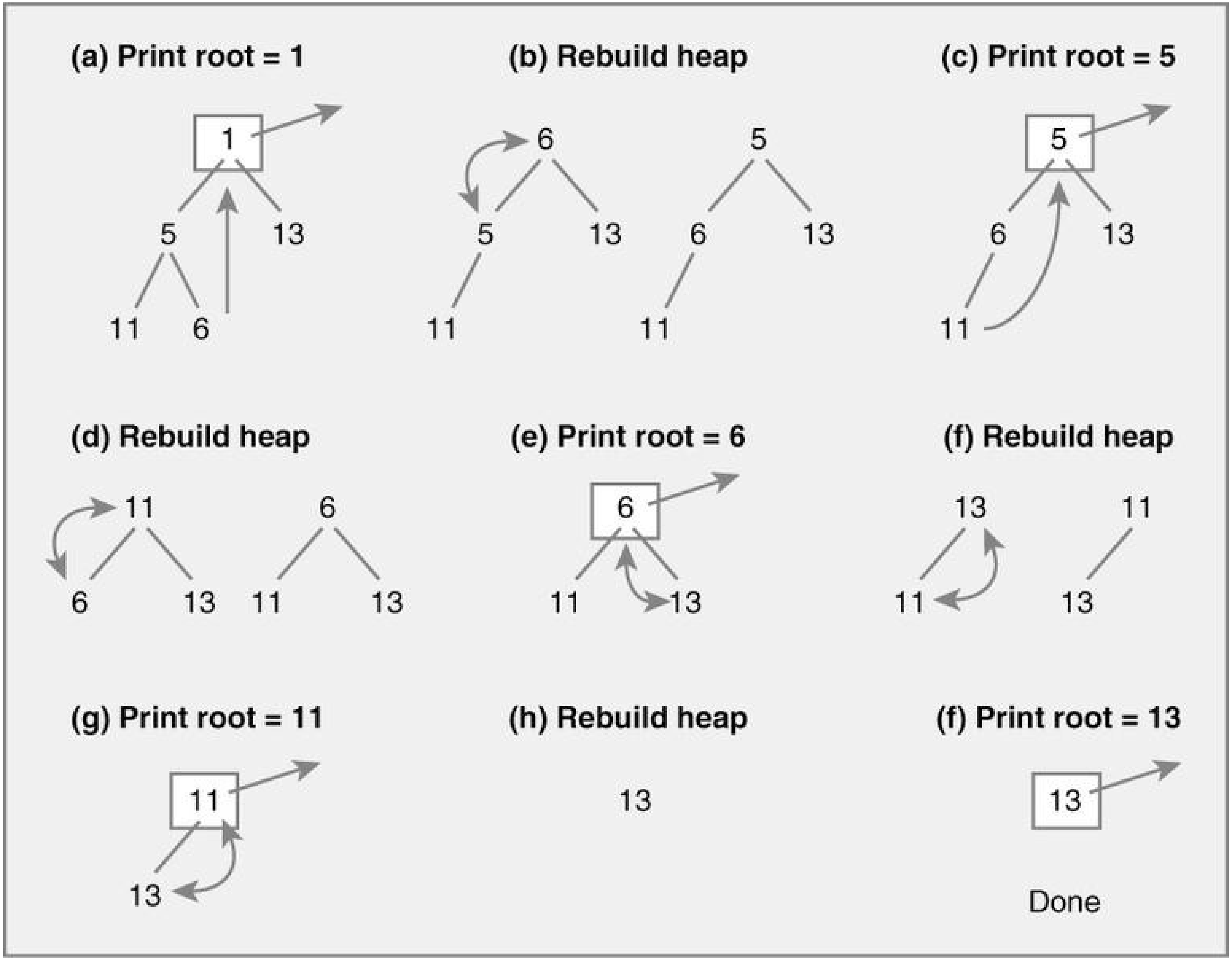
Heapsort Example

- Input
  - 11, 5, 13, 6, 1
- View heap during insert, removal
  - As tree
  - As array

Heapsort – Insert Values



Heapsort – Remove Values



Heapsort – Insert in to Array 1

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Insert 11	11				

Heapsort – Insert in to Array 2

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Insert 5	11	5			
Swap	5	11			

Heapsort – Insert in to Array 3

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Insert 13	5	11	13		

Heapsort – Insert in to Array 4

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Insert 6	5	11	13	6	
Swap	5	6	13	11	

…

Heapsort – Remove from Array 1

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Remove root	1	5	13	11	6
Replace	6	5	13	11	
Swap w/ child	5	6	13	11	

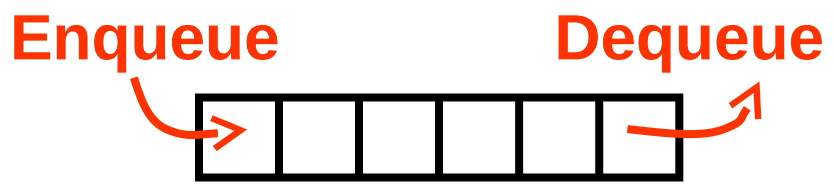
Heapsort – Remove from Array 2

- **Input**
  - **11, 5, 13, 6, 1**

Index =	0	1	2	3	4
Remove root	5	6	13	11	
Replace	11	6	13		
Swap w/ child	6	11	13		

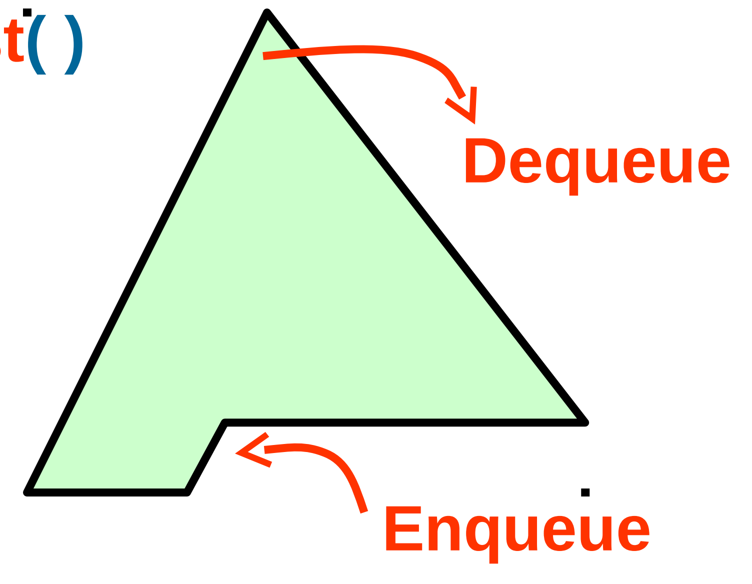
Heap Application – Priority Queue

- Queue
  - Linear data structure
  - First-in First-out (FIFO)
  - Implement as array / linked list



Heap Application – Priority Queue

- **Priority queue**
  - **Elements are assigned priority value**
  - **Higher priority elements are taken out first**
  - **Implement as heap**
    - **Enqueue ⇒ insert( )**
    - **Dequeue ⇒ getSmallest( )**



Priority Queue

- **Properties**
  - **Lower value = higher priority**
  - **Heap keeps highest priority items in front**

- **Complexity**
  - **Enqueue ⇒ insert( ) = O(log(n))**
  - **Dequeue ⇒ getSmallest( ) = O(log(n))**
  - **For any heap**

Heap vs. Binary Search Tree

- **Binary search tree**
  - **Keeps values in sorted order**
  - **Find any value**
    - **O(log(n)) for balanced tree**
    - **O(n) for degenerate tree (worst case)**
- **Heap**
  - **Keeps smaller values in front**
  - **Find minimum value**
    - **O(log(n)) for any heap**