HEAPS (PRIORITY QUEUE)

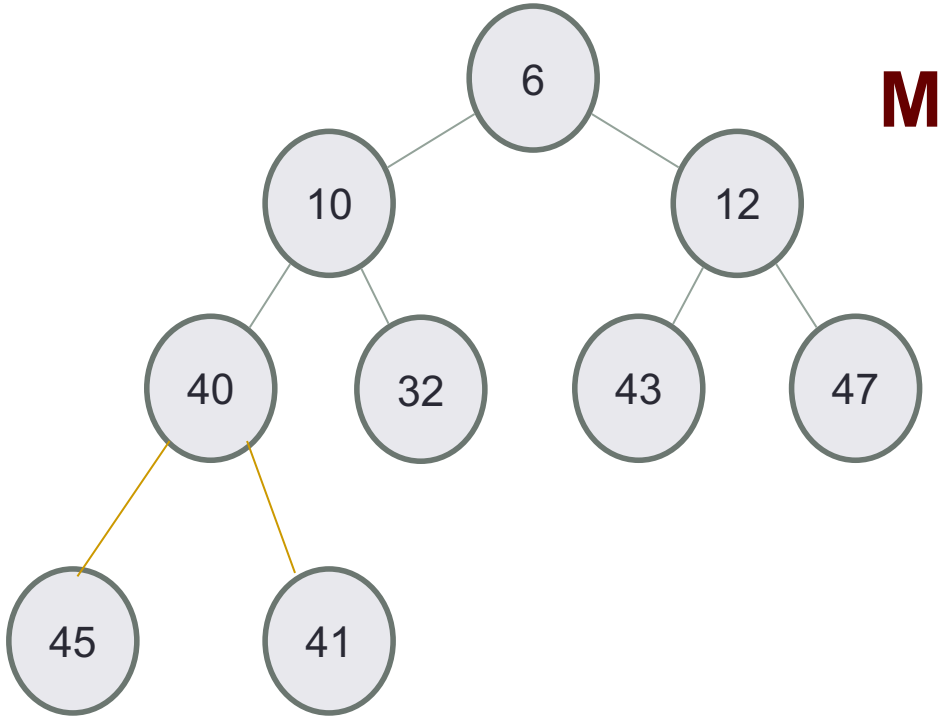
Heaps: Supported Operations

1. Min Heaps:
   - Insert : O(log N)
   - Min: O(1)
   - Delete Min: O(log N)
   - Batch insert: O(N)
2. Max Heaps
   - Insert : O(log N)
   - Max: O(1)
   - Delete Max: O(log N)
   - Batch insert: O(N)

Choose heap if you are doing repeated insert/delete/(min OR max) operations

Heaps as binary trees

- **Rooted binary tree that is as complete as possible**
- **Each node satisfies the heap property:**
  - **For a min heap: key(x)<= key(children of x)**
  - **Ex: Store {12, 41, 47, 45, 43, 32, 6, 10, 40} in a minHeap**

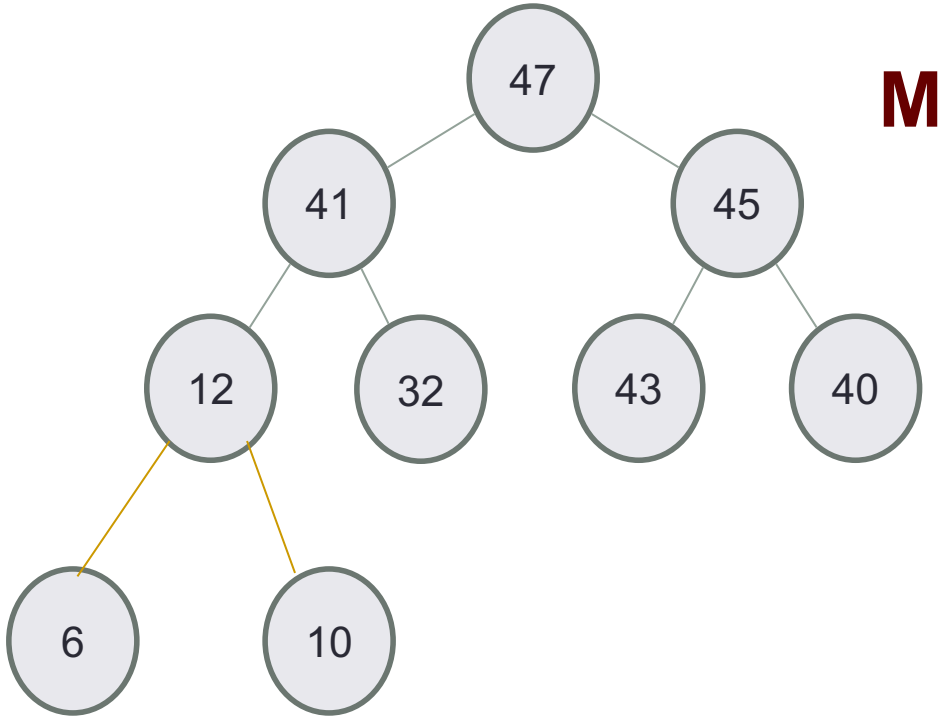


Min Heap with 9 nodes

Where is the minimum element?

Heaps as binary trees

- **Rooted binary tree that is as complete as possible**
- **Each node satisfies the heap property:**
  - **For a max heap: key(x)>= key(children of x)**
  - **Ex: Store {12, 41, 47, 45, 43, 32, 6, 10, 40} in a maxHeap**



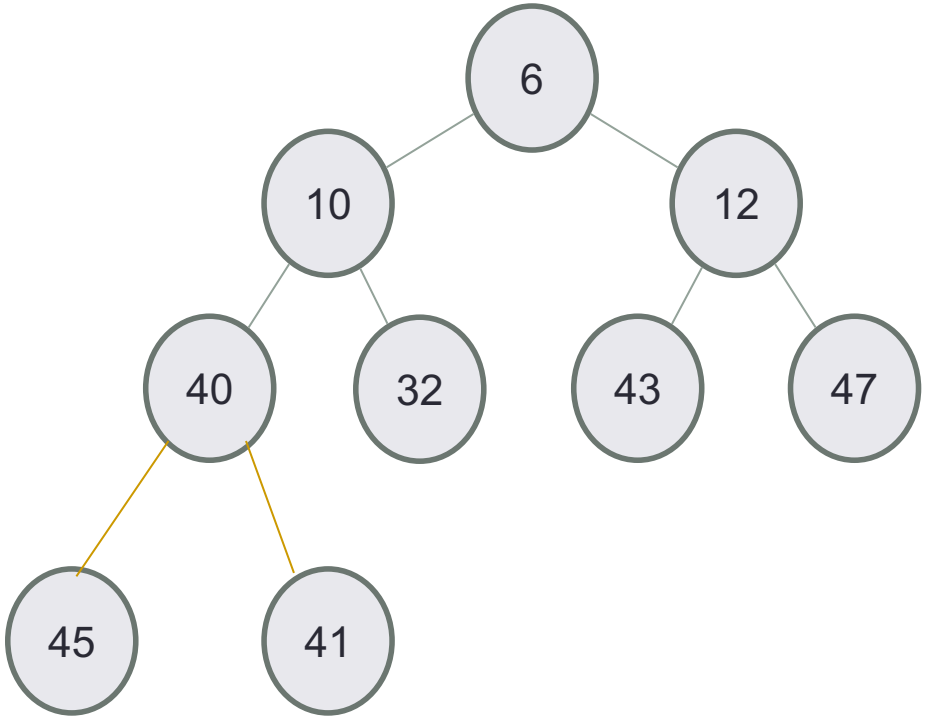
Max Heap with 9 nodes

Where is the maximum element?

Identifying heaps

Starting with the following min Heap which of the following operations will result in something that is NOT a min Heap

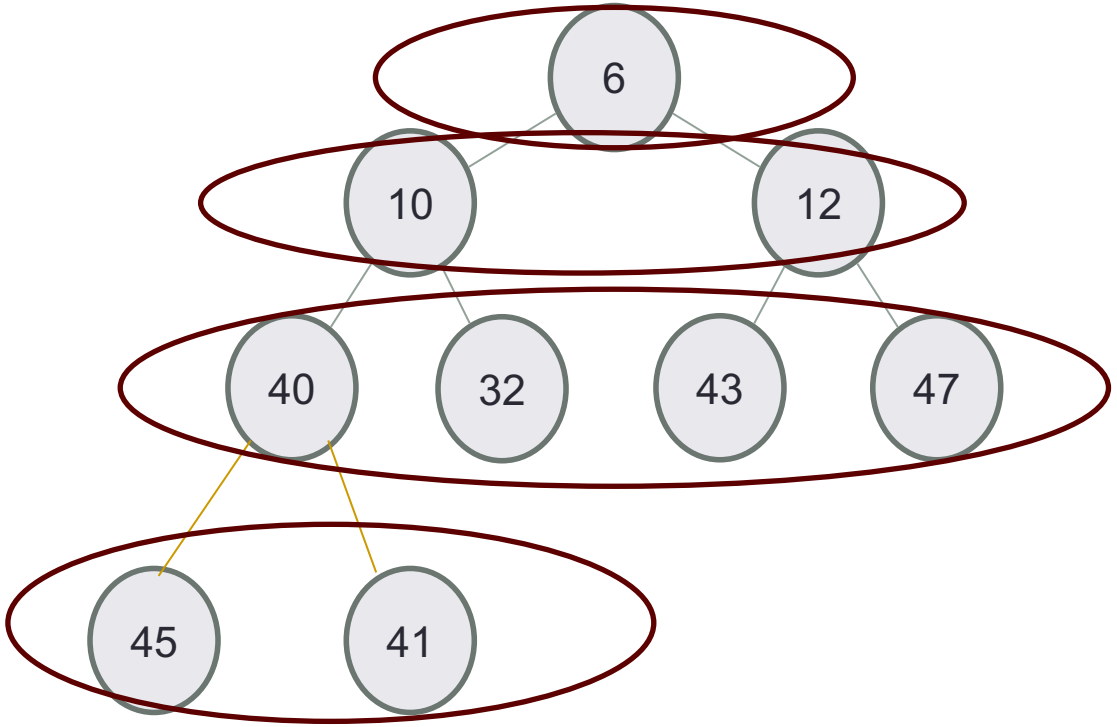
A. Swap the nodes 40 and 32
B. Swap the nodes 32 and 43
C. Swap the nodes 43 and 40
D. Insert 50 as the left child of 45
E. C&D



Implementing heaps as arrays

Conceptualize heaps as trees, implement as arrays

Value	6	10	12	40	32	43	47	45	41	
Index	0	1	2	3	4	5	6	7	8	



Level 0

Level 1

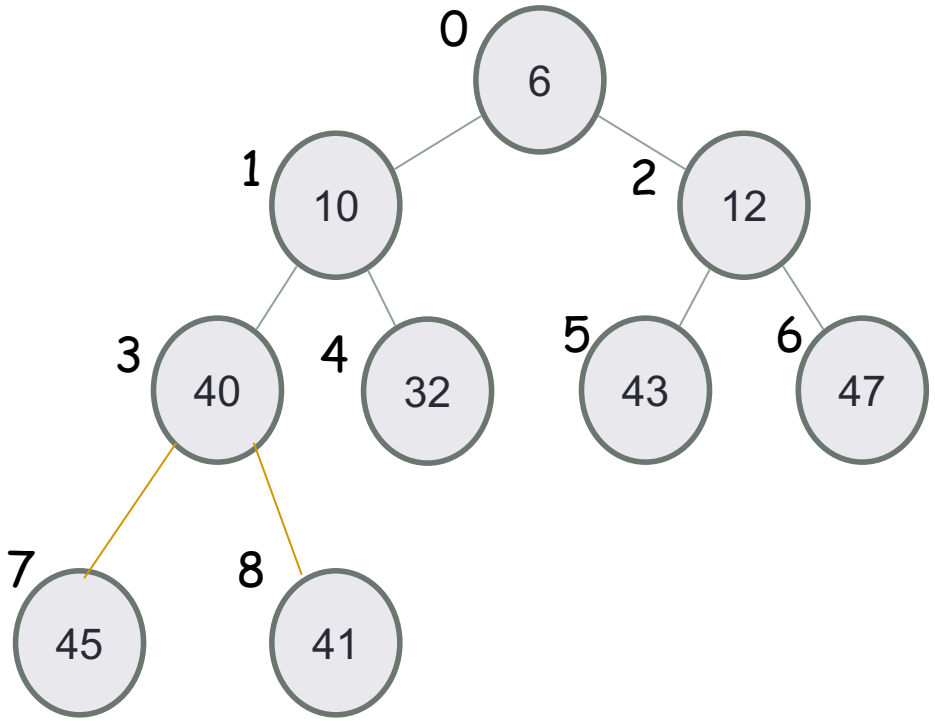
Level 2

Level 3

If the heap was implemented as an array:

- How can we access the parent of a node (x) located at index i of the array?
- How can we access the children of a node(x) located at index i?

Conceptualize heaps as trees, implement as arrays

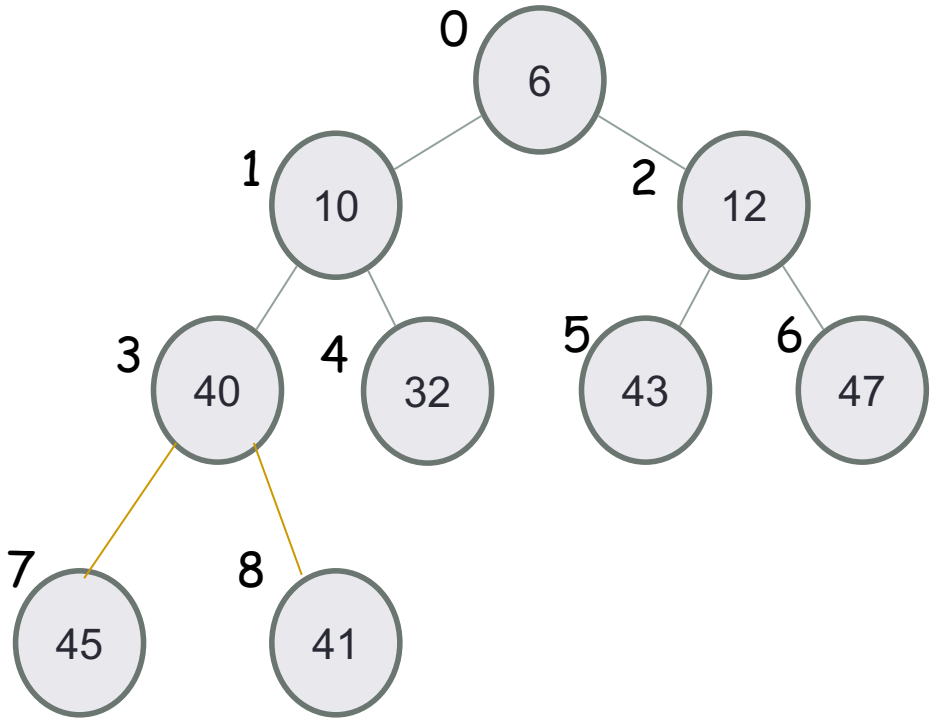


For a node at index i, what is the index of the left and right children?

A. (2*i, 2*i+1)
B. (2*i+1, 2*i+2)
C. (log(i), log(i)+1)
D. None of the above

Value	Index	Index of parent	Index of children
6	0	-	1, 2
10	1	0	3, 4
12	2	0	5, 6
40	3	1	7, 8
32	4	1	
43	5	2	
47	6	2	
45	7	3	
41	8	3	

Conceptualize heaps as trees, implement as arrays

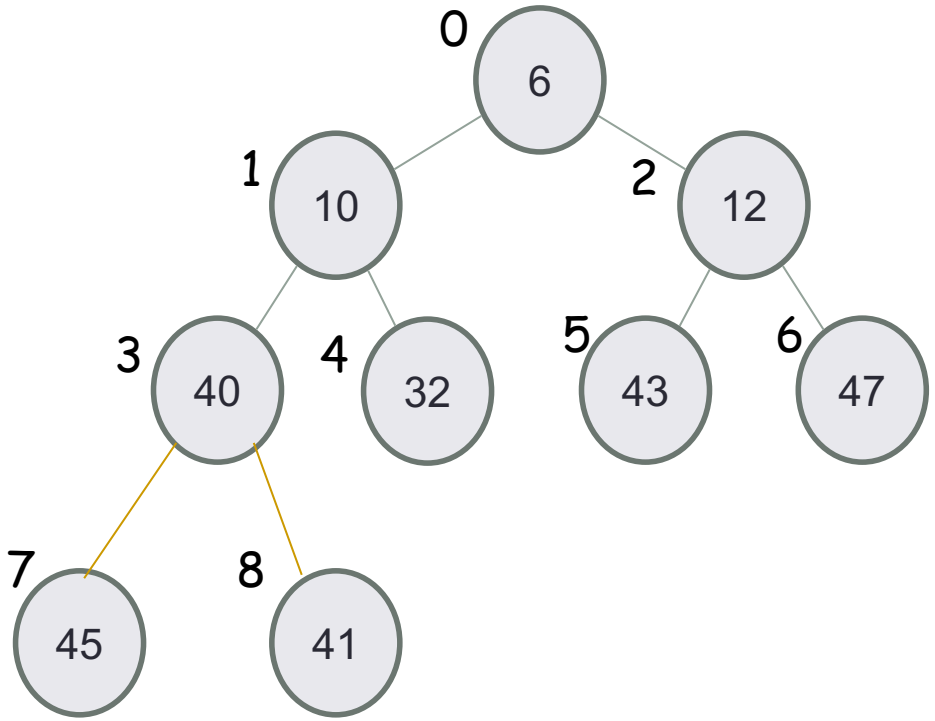


For a node at index i, what is the index of the left and right children?

(2*i+1, 2*i+2)

Value	Index	Index of parent	Index of children
6	0	-	1, 2
10	1	0	3, 4
12	2	0	5, 6
40	3	1	7, 8
32	4	1	
43	5	2	
47	6	2	
45	7	3	
41	8	3	

Conceptualize heaps as trees, implement as arrays



Value	Index	Index of parent	Index of children
6	0	-	1, 2
10	1	0	3, 4
12	2	0	5, 6
40	3	1	7, 8
32	4	1	
43	5	2	
47	6	2	
45	7	3	
41	8	3	

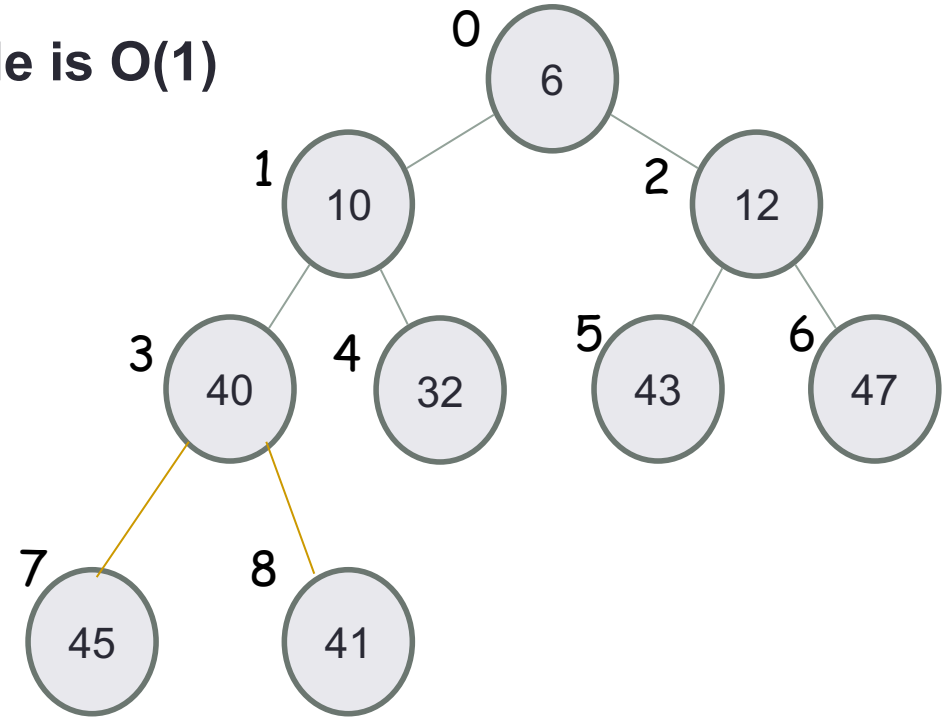
For a node at index i, index of the parent is:

i/2 - 1, if i is even

(i-1)/2, if I is odd

How is the array implementation of the heap useful?

- **More space efficient**
- **Accessing parent and children of a node is O(1)**
- **Easier to insert elements in the heap**



Value	Index
6	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8

Index of parent of node at index i is:

i/2 - 1, if i is even

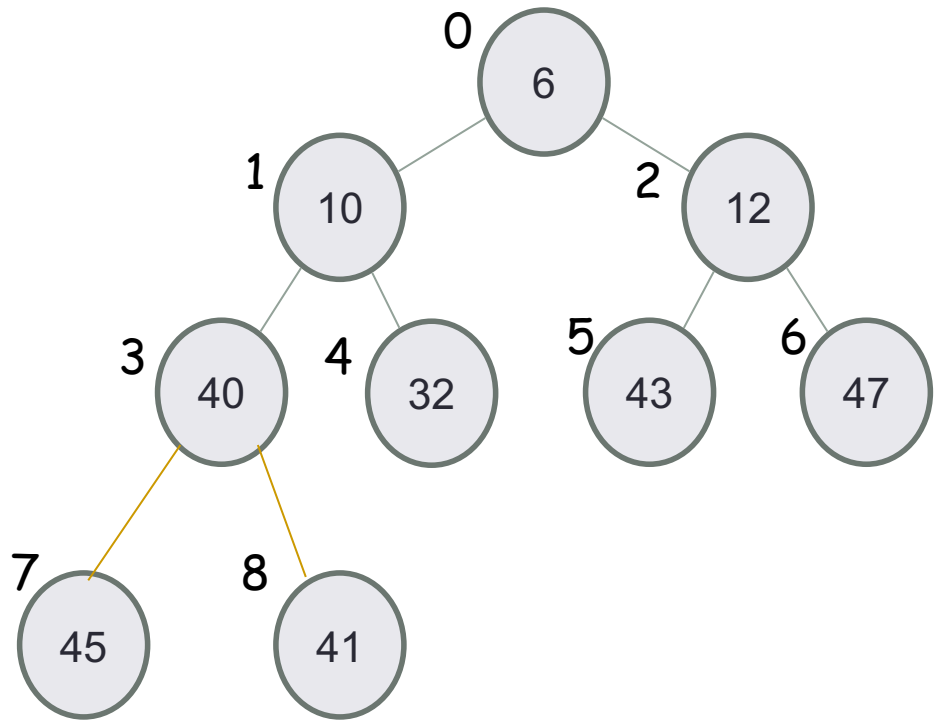
(i-1)/2, if i is odd

Index of children of i: 2*i+1, 2*i+2

Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**
- **For example : Insert 50**

Insert 50



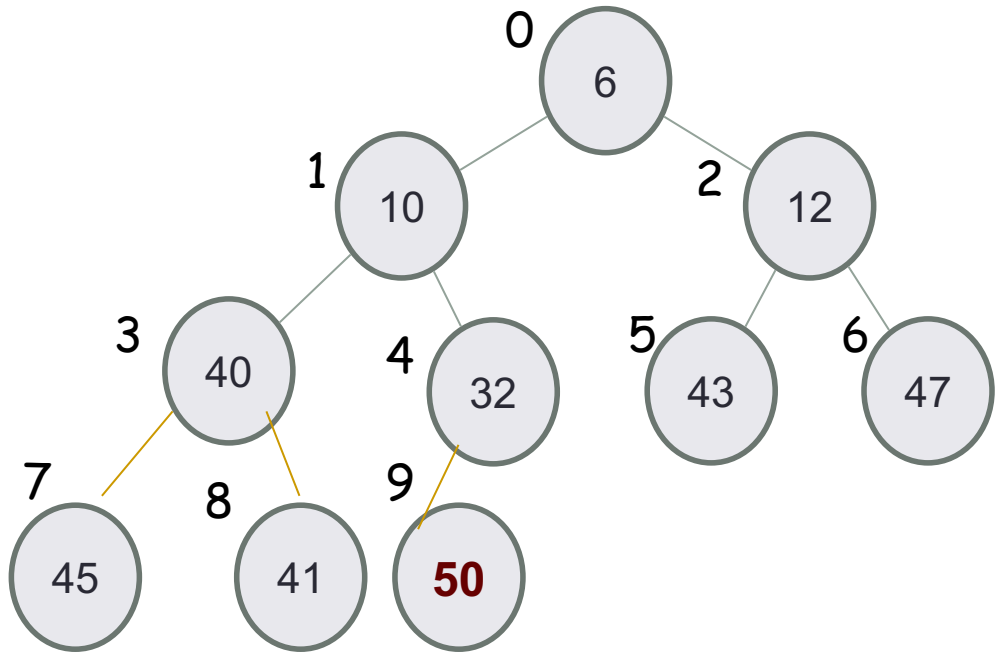
Value	Index
6	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8

Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**
- **For example :**

  Insert 50, then 35

Insert 35

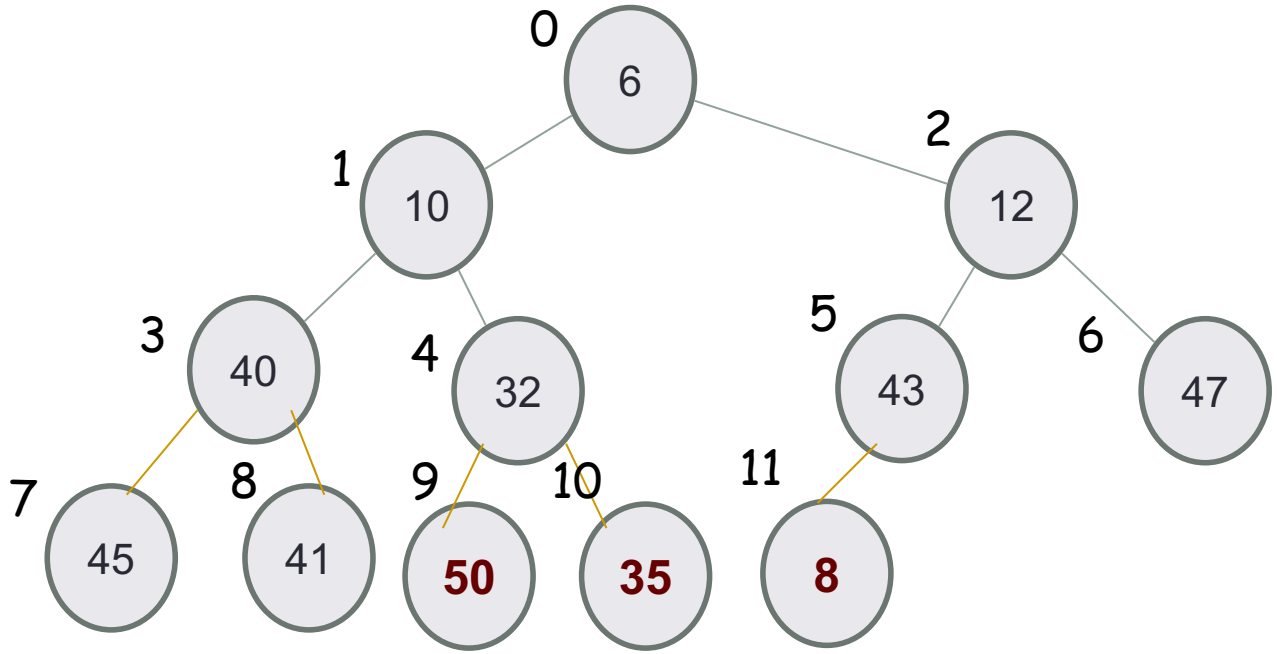


Value	Index
6	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8
50	9

Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**

Insert 8



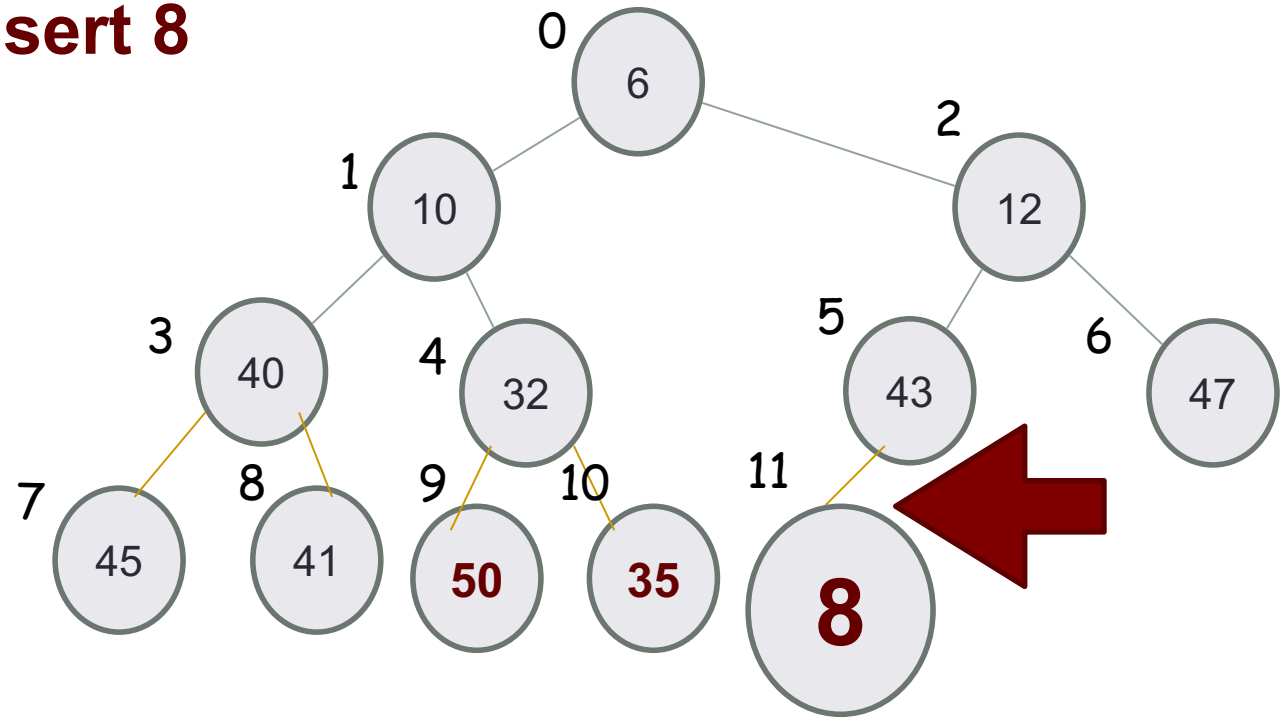
Is the heap property violated when we insert 8?
A. Yes. B No

Value	Index
6	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8
50	9
35	10
8	**11**

Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**
- **Else: while(key(parent(x))>key(x)) swap the values of x and parent(x)**

Insert 8



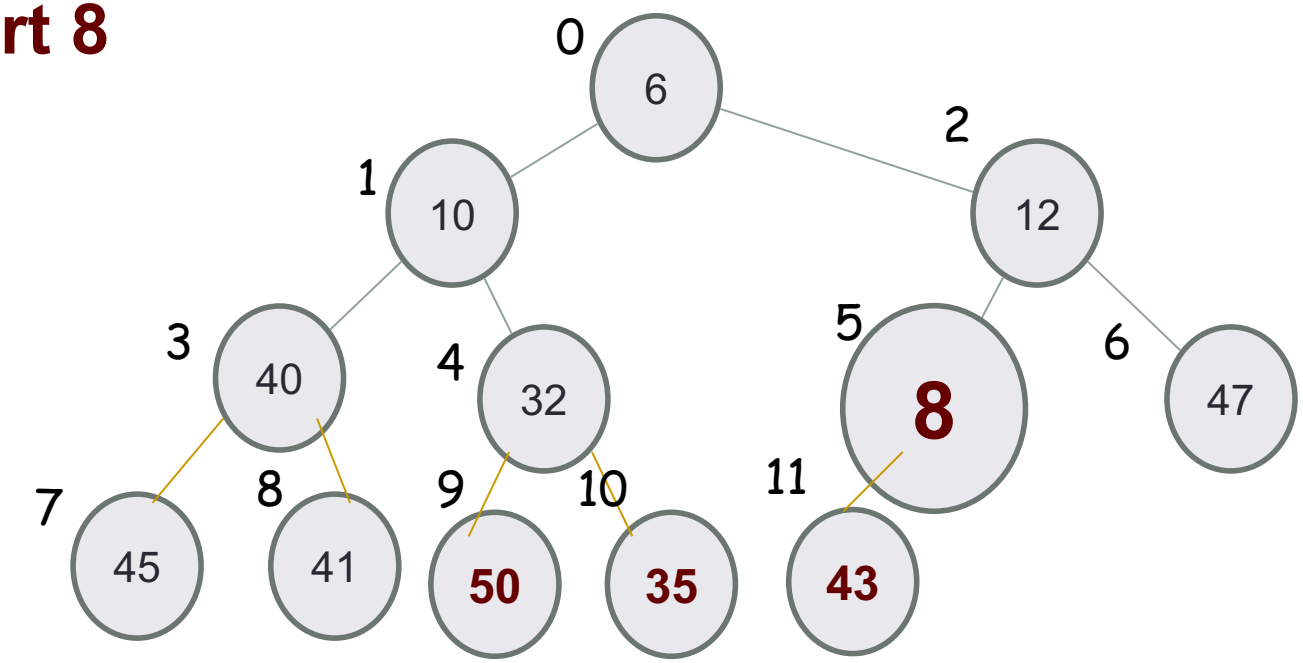
Heap property violated between 43 and 8!!!
Swap 43 and 8

Value	Index
6	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8
50	9
35	10
8	**11**

Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**
- **Else: while(key(parent(x))>key(x)) swap the values of x and parent(x)**

Insert 8



Parent(8)

key:8

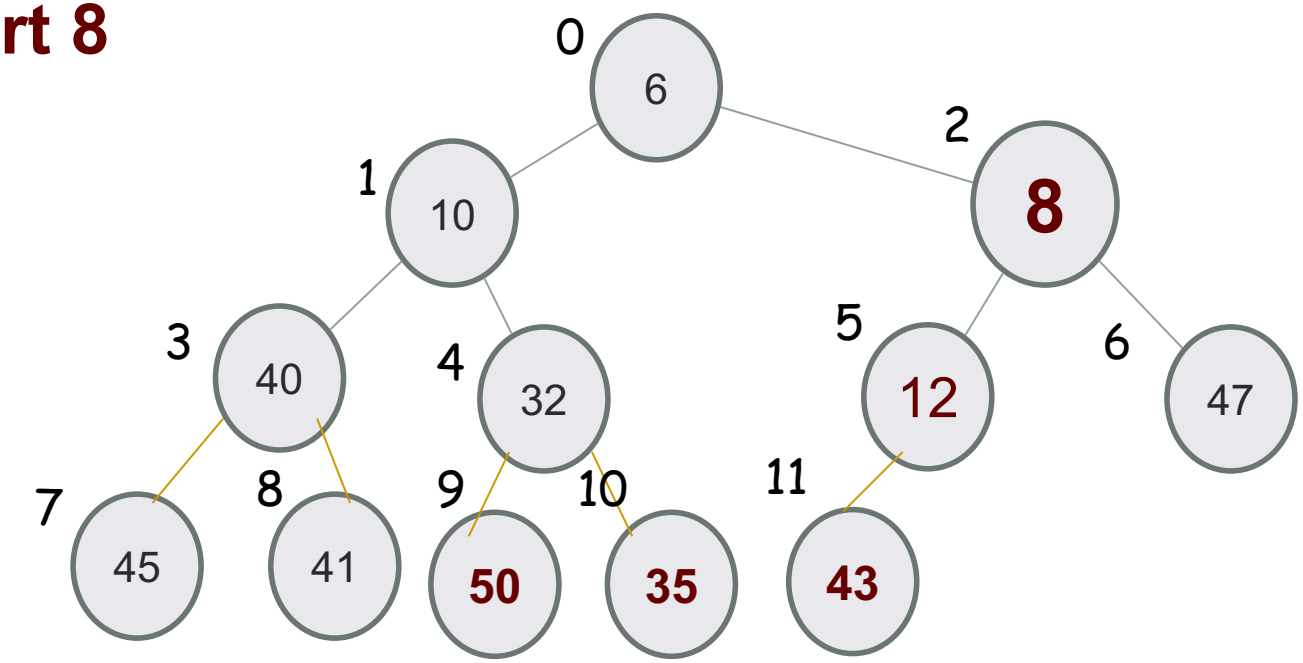
Value	Index
6	0
10	1
12	2
40	3
32	4
8	5
47	6
45	7
41	8
50	9
35	10
43	**11**

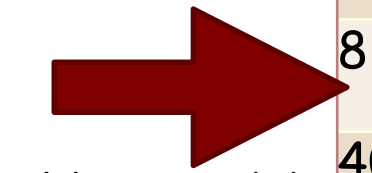
Heap property violated between 8 and 12!!!
Swap 8 and 12

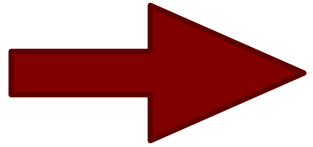
Insert into a heap

- **Insert key(x) at the last level of tree in the first open spot (going from left to right)**
- **If the heap property is not violated - Done**
- **Else: while(key(parent(x))>key(x)) swap the values of x and parent(x)**

Insert 8





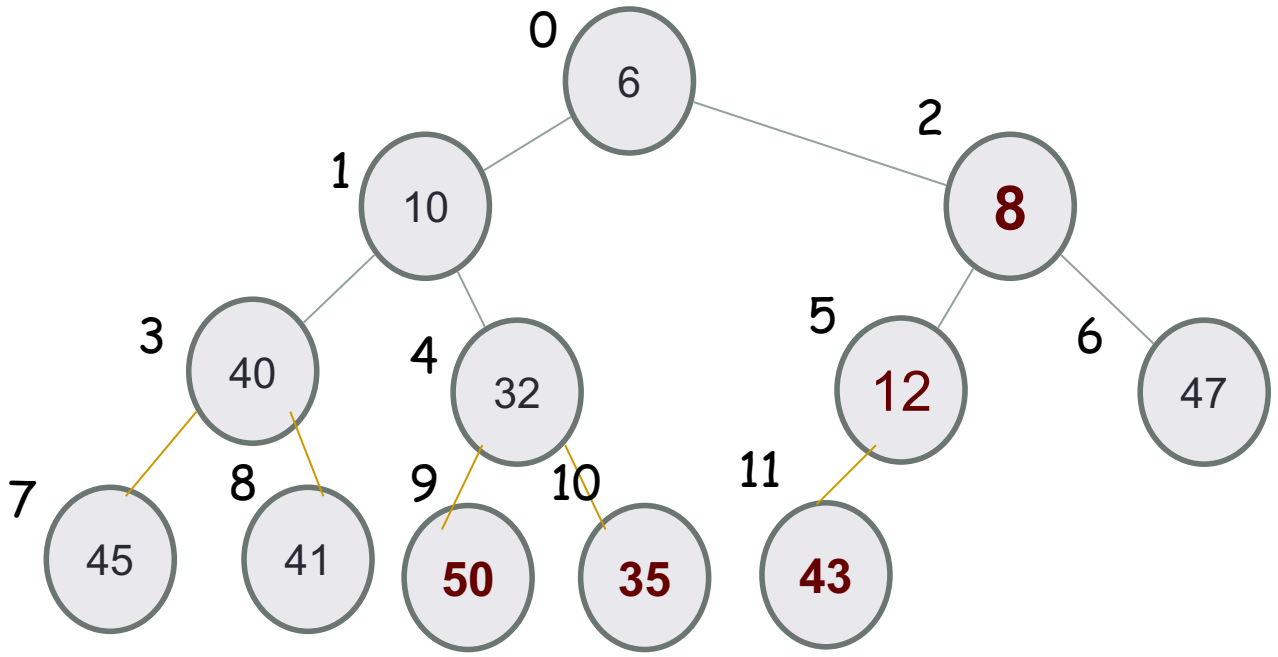


Value	Index
6	0
10	1
8	2
40	3
32	4
12	5
47	6
45	7
41	8
50	9
35	10
43	**11**

To fix the heap property on an insert BUBBLE UP!
Worst case: O(logN)

Delete min

- **Delete the root:**
  - **Replace the root with the last node of the lowest level (43)**



Value	Index
6	0
10	1
8	2
40	3
32	4
12	5
47	6
45	7
41	8
50	9
35	10
43	**11**

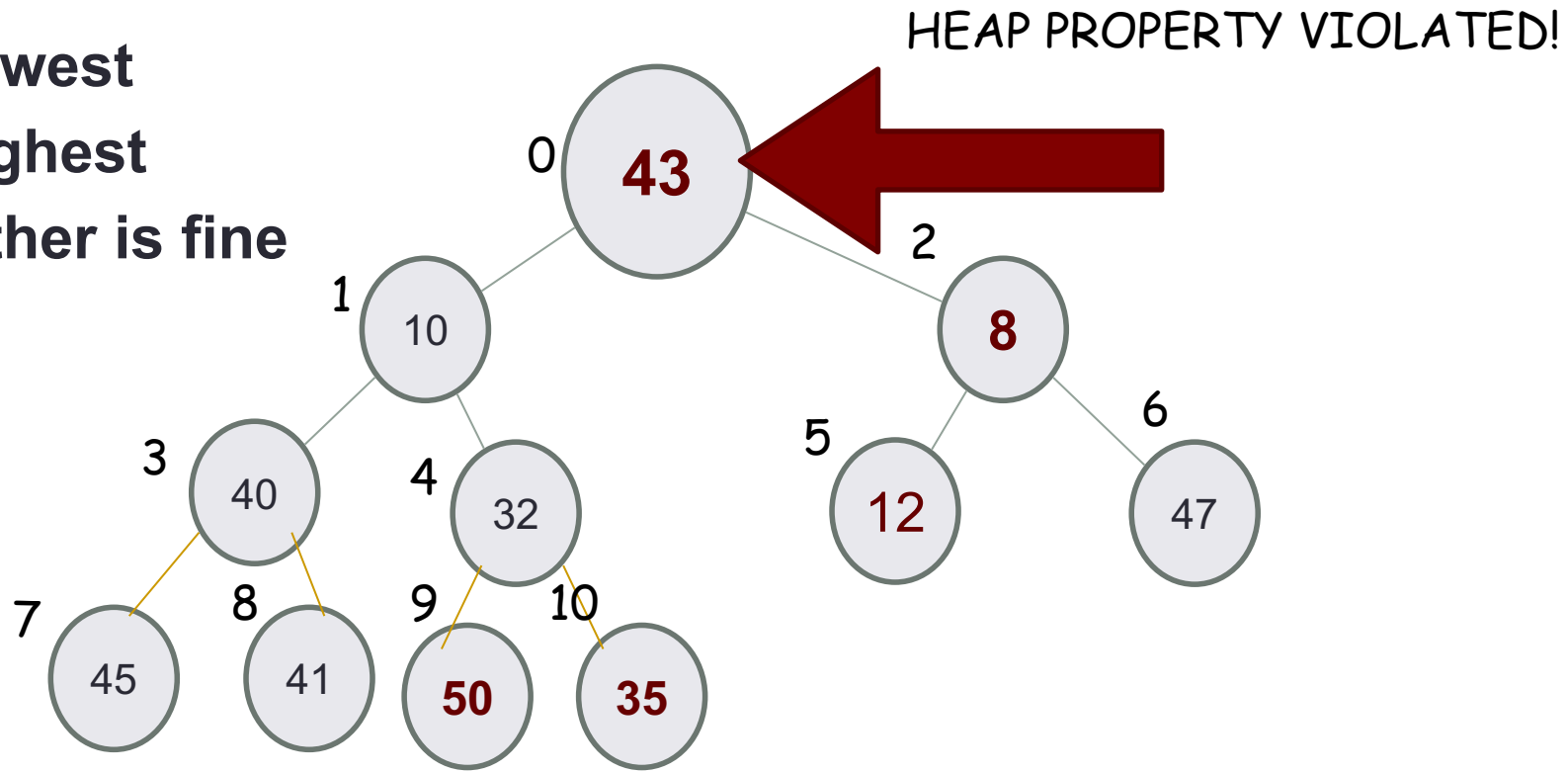
Delete min

- **Delete the root:**
  - **Replace the root with the last node in the array**
  - **If heap property is violated - swap with the child that has the ____ key value**

A. Lowest

B. Highest

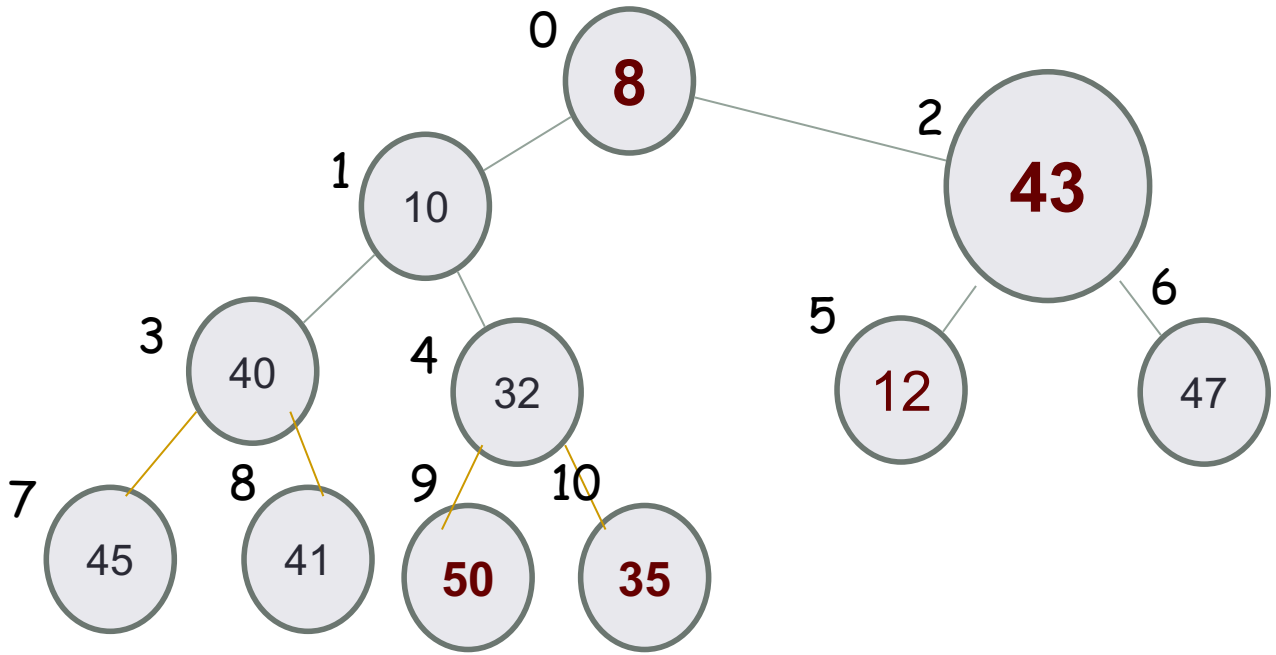
C. Either is fine



Value	Index
43	0
10	1
8	2
40	3
32	4
12	5
47	6
45	7
41	8
50	9
35	10

Delete min

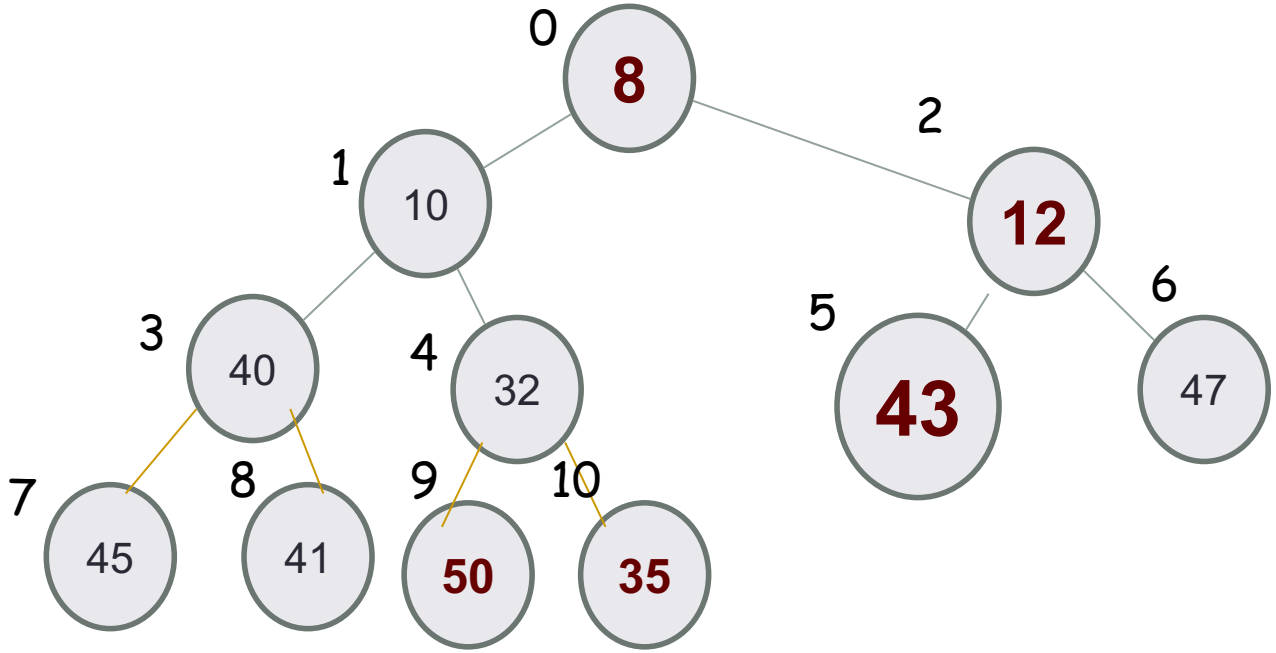
- **Delete the root:**
  - **Replace the root with the last node in the array**
  - **If heap property is violated - swap with the child that has the LOWEST key value**



Value	Index
8	0
10	1
43	2
40	3
32	4
12	5
47	6
45	7
41	8
50	9
35	10

Delete min

- **Delete the root:**
  - **Replace the root with the last node in the array**
  - **If heap property is violated - swap with the child that has the LOWEST key value - Repeat until heap property is fixed**



To fix the heap property on an insert BUBBLE DOWN!
Worst case: O(logN)

Value	Index
8	0
10	1
12	2
40	3
32	4
43	5
47	6
45	7
41	8
50	9
35	10

std::priority_queue template arguments

The template for priority_queue takes 3 arguments:

```
1 template < class T, class Container = vector<T>,
2            class Compare = less<typename Container::value_type> > class priority_queue;
```

- The first is the type of the elements contained in the queue.
- If it is the only template argument used, the remaining 2 get their default values:
  - a **vector<T>** is used as the internal store for the queue,
  - **less** a class that provides priority comparisons

priority_queue<int> pq; //Elements with highest priority is on the top
Methods:
* push() //insert
* pop() //delete min
* top() //get min

Applications

- Fast sort
- Finding the median of a sequence of numbers
- Building a Huffman-code tree efficiently

Announcements

- Lab05 due this Friday at midnight because of outages in submit
- Pa03 due 03/18
- Final exam 03/19 (Next Monday)