CS 261-020 Data Structures

Lecture 12

AVL Trees (cont.)

Priority Queues and Heaps

2/27/24, Tuesday

Oregon State University

Odds and Ends

- Recitation 8 posted
- Assignment 4 posted!
  - Note: THIS IS A ONE-WEEK ASSIGNEMNT!!!
- Assignment 3 Due Extension → Monday 2/26 midnight

Lecture Topics:

- AVL Trees (cont.)

- Priority Queues & Heaps
- Array-based Heaps
- Build a heap from an arbitrary array
- Heapsort

Single vs. Double Rotation

<table>
<tr><td colspan="2" rowspan="2"></td><th colspan="2"><code>balanceFactor(N)</code></th></tr>
<tr><th>-2 (left-heavy)</th><th>2 (right-heavy)</th></tr>
<tr><td rowspan="3"><code>balanceFactor(C)</code></td><td>-1
(left-heavy)</td><td rowspan="2">Left-left imbalance
Single rotation:
right around N</td><td>Right-left imbalance
Double rotation:
1. right around C
2. left around N</td></tr>
<tr><td>0</td><td rowspan="2">Right-right imbalance
Single rotation:
left around N</td></tr>
<tr><td>1
(right-heavy)</td><td>Left-right imbalance
Double rotation:
1. left around C
2. right around N</td></tr>
</table>

AVL Tree operations

- Need a mechanism to retrace a path *upwards* from a given node back to the root

- How: by adding a pointer to the AVL tree node structure that points to the node's parent
  - Then, retracing the path upwards from a node to the tree's root is as simple as following these parent pointers up the tree

- Add an additional field that allows us to track the height of the subtree rooted at each node.

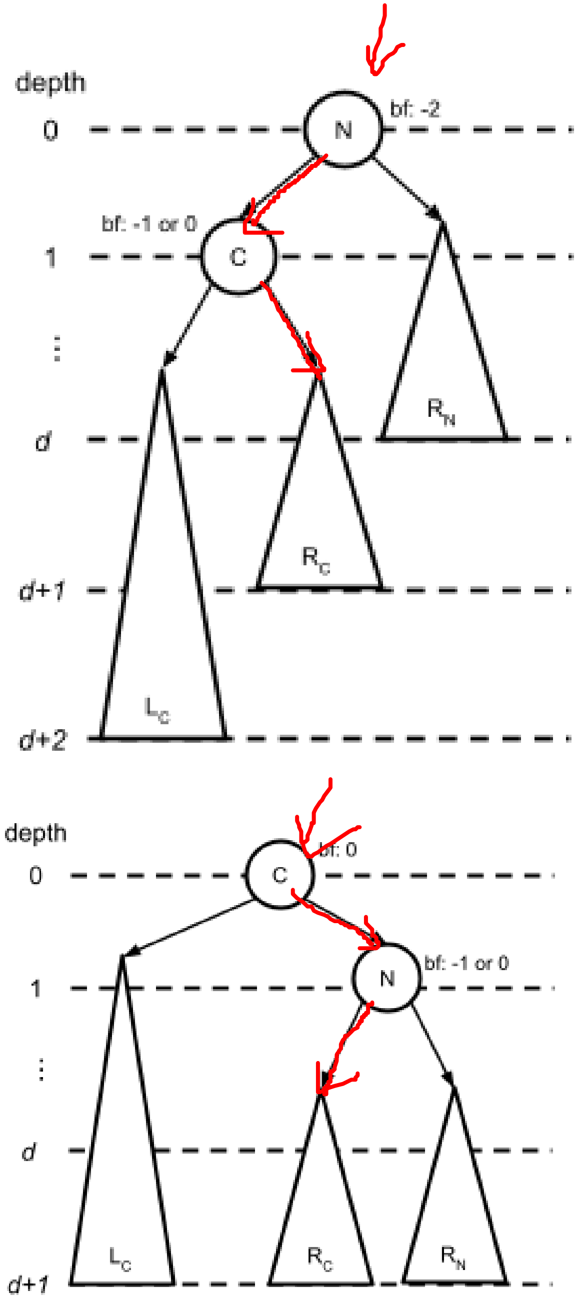
```
struct avl_node {
    int key;
    void* value;
    int height;
    struct avl_node* left;
    struct avl_node* right;
    struct avl_node* parent;
};
```

- When a node doesn't have a parent, parent = NULL.
- Specifically, the root node of the tree will always have a NULL parent pointer

AVL Tree operations

- Pseudocode for a right rotation:
- `rotateRight(N):`

```
C ← N.left
N.left ← C.right
if N.left is not NULL:
    N.left.parent ← N
C.right ← N
C.parent ← N.parent
N.parent ← C
updateHeight(N)
updateHeight(C)
return C
```



AVL Tree operations

- Pseudocode for a left rotation:
- `rotateLeft(N):`

```
rotateLeft(N):
    C ← N.right
    N.right ← C.left
    if N.right is not NULL:
        N.right.parent ← N
    C.left ← N
    C.parent ← N.parent
    N.parent ← C
    updateHeight(N)
    updateHeight(C)
    return C
```

```
updateHeight(N):
    N.height ← MAX(height(N.left), height(N.right)) + 1
```

AVL Tree operations

- How these pieces work:
  - Rotating left or right around a given node: simply involves trading a few pointers.
  - After every rotation, re-compute the subtree heights for both the node that moved downwards during the rotation (i.e. N) and the node that moved upwards during the rotation (i.e. C).

AVL Tree operations

- pseudocode for the insert operation:

```
avlInsert(tree, key, value):
      insert key, value into tree like normal BST insertion
      N ← newly inserted node
      P ← N.parent
      while P is not NULL:
            rebalance(P)
            P ← P.parent
```

- pseudocode for the remove operation:

```
avlRemove(tree, key):
      remove key from tree like normal BST removal
      P ← lowest modified node (e.g. parent of removed node)
      while P is not NULL:
            rebalance(P)
            P ← P.parent
```

The key piece: rebalance() function, which performs rebalancing at each node:

AVL Tree operations

- Pseudocode for rebalance():

```
• rebalance(N):
        if balanceFactor(N) < -1:
                if balanceFactor(N.left) > 0:
                        N.left ← rotateLeft(N.left)
                newSubtreeRoot ← rotateRight(N)
                if newSubtreeRoot.parent is not NULL
                        if newSubtreeRoot.parent.left is N:
                                newSubtreeRoot.parent.left ← newSubtreeRoot
                        else:
                                newSubtreeRoot.parent.right ← newSubtreeRoot
        else if balanceFactor(N) > 1:
                if balanceFactor(N.right) < 0:
                        N.right ← rotateRight(N.right)
                newSubtreeRoot ← rotateLeft(N)
                if newSubtreeRoot.parent is not NULL
                        if newSubtreeRoot.parent.left is N:
                                newSubtreeRoot.parent.left ← newSubtreeRoot
                        else:
                                newSubtreeRoot.parent.right ← newSubtreeRoot
  else:
        updateHeight(N)
```

Runtime Complexity of AVL Tree operations

- Single rotation (rotateLeft() and rotateRight()):
  - A limited number of pointers is updated
  - The height of two nodes is updated
  - Thus, O(1)
- rebalance() :
  - For each call, at most two rotations.
  - Thus, O(1)

- How many times will rebalance() be called?
  - once per node on a traversal upwards to the root of the tree
  - Thus, the maximum number of times rebalance() can be called is h (height of the tree)

- If a tree is height balanced, then h = log(n). Thus, the AVL tree's insert and remove operations each have overall complexity of O(h) = O(log n)

Lecture Topics:

- AVL Trees (cont.)

- Priority Queues & Heaps
- Array-based Heaps
- Build a heap from an arbitrary array
- Heapsort

Priority Queues

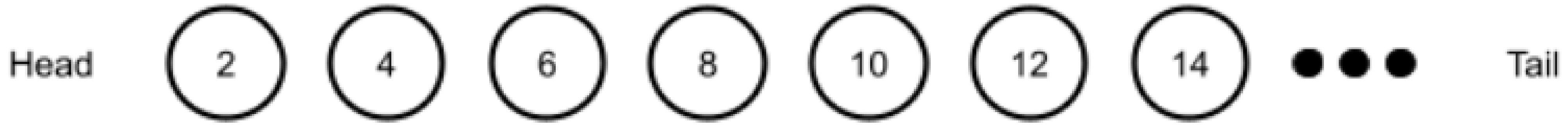
- *Priority Queue*: an ADT that associates a priority value with each element.

- The element with the highest priority is the first one dequeued.
  - highest priority – element with the lowest priority value

- Interface:
  - **`insert()`** – insert an element with a specified priority value
  - **`first()`** – return the element with the lowest priority value (the “first” element in the priority queue)
  - **`remove_first()`** – remove (and return) the element with the lowest priority value

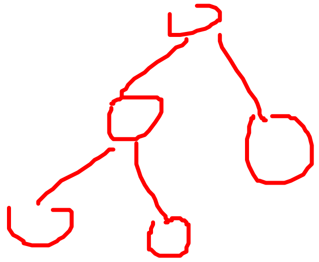
Priority Queues Visualization

- The user's view of a priority queue:



- A priority queue is typically implemented using a data structure called a ***heap***

Heaps



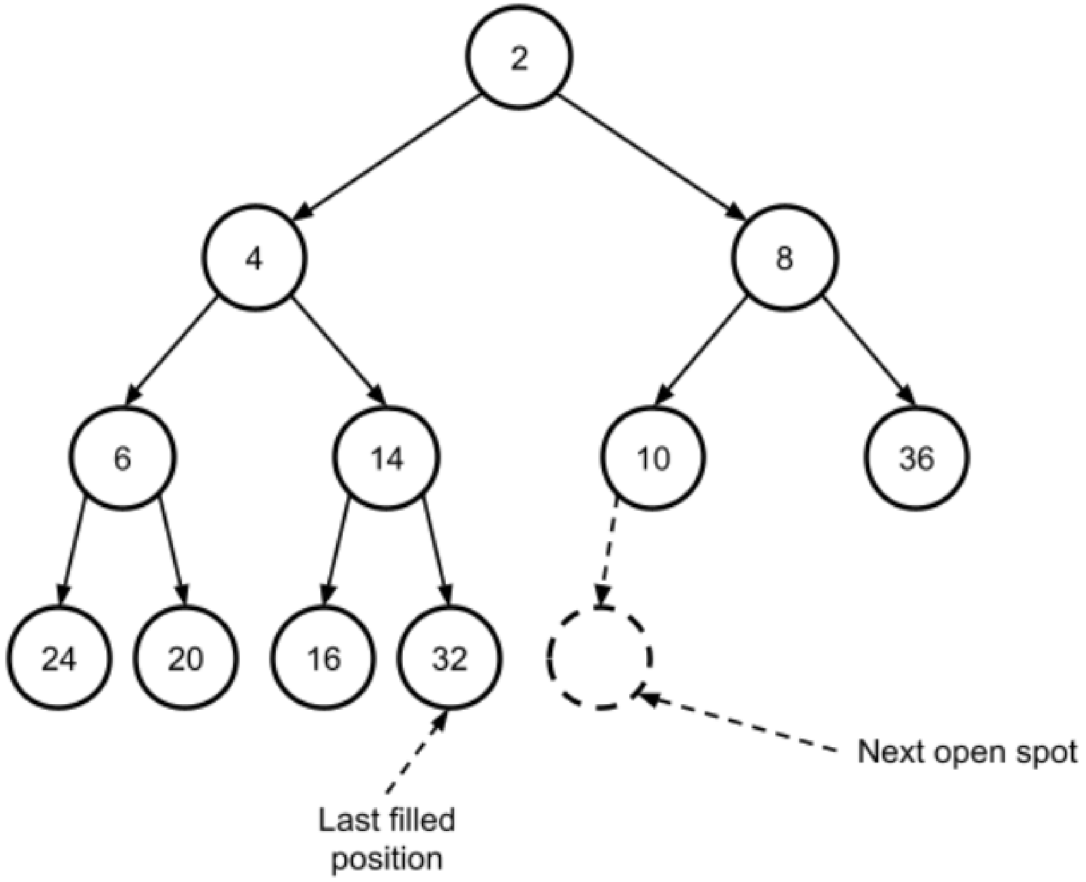
- Caveat: The heap data structure ≠ the dynamic memory space "heap"

- A heap data structure: a ***complete*** binary tree in which every node's value is less than or equal to the values of its children
  - This is called a minimizing binary heap, or just "min heap".
  - max heap: each node's value is greater than or equal to the values of its children

- Recall: a complete binary tree is one that is filled, except for the bottom level, which is filled from left to right
  - The longest path from root to leaf in such a tree is $O(\log n)$.

Min Heap Example

- With only priority values displayed:



Add a node to a Heap

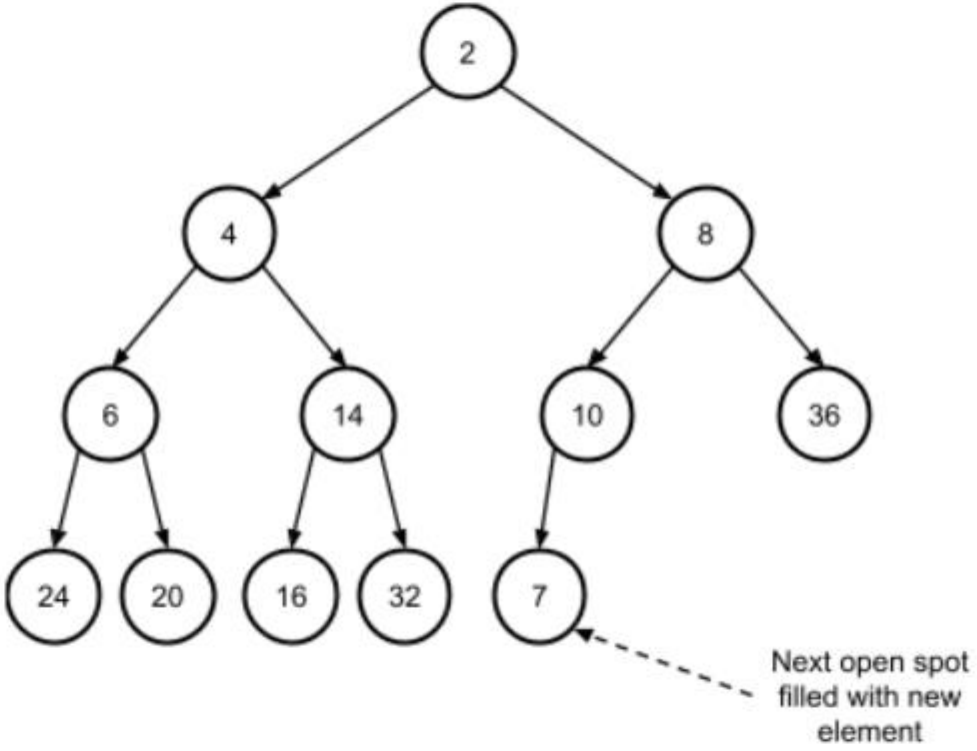
- A min (or max) heap is maintained through the addition and removal of nodes via **percolations**
  - **Percolation – move nodes up and down the tree according to their priority values.**

- When adding a value to a heap,
  - place it into the next open spot
  - percolate it up the heap until its priority value is less than both of its children

Add a node to a Heap

- Example: adding the value 7 to the min heap:

1. place it in the next open spot

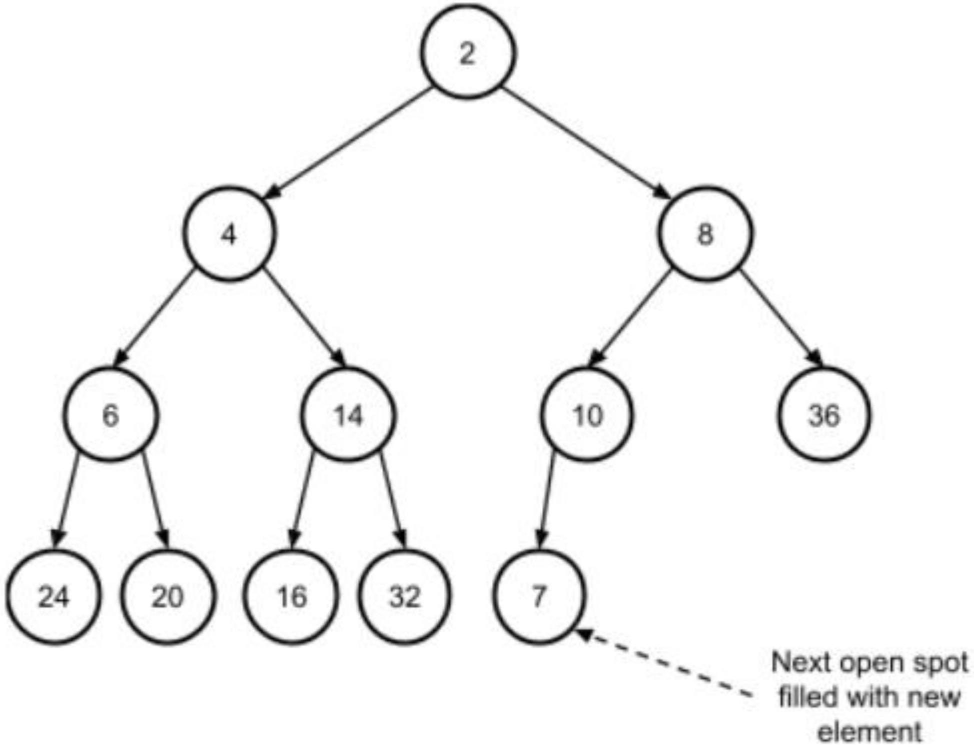


```
while new priority value < parent's priority value:
    swap new node with parent
```

Add a node to a Heap

- Example: adding the value 7 to the min heap:

2. percolate the new element up the tree

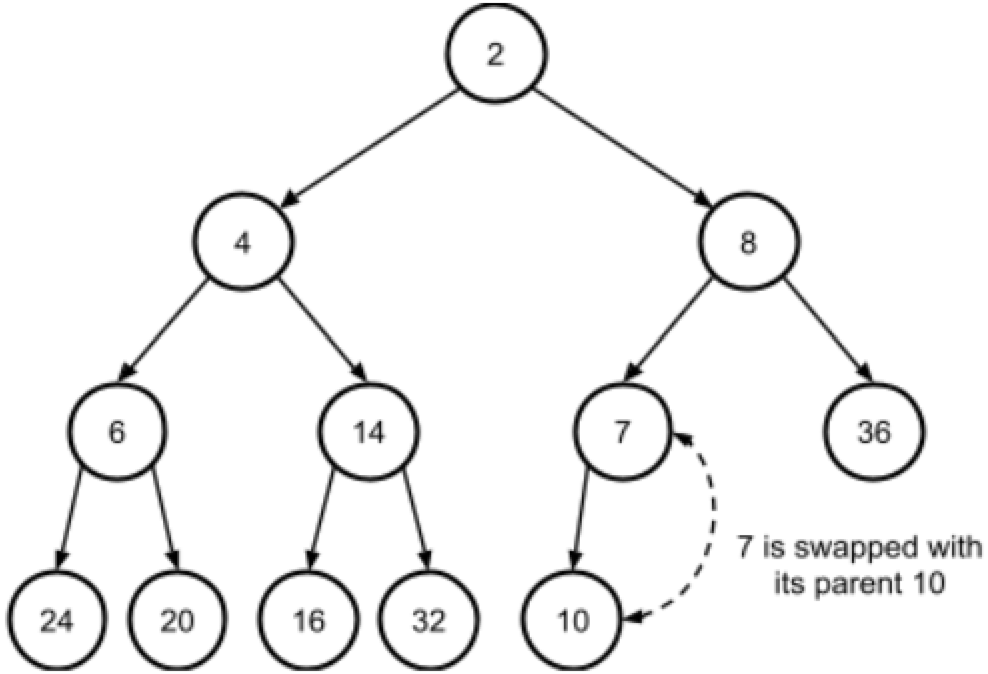


```
while new priority value < parent's priority value:
    swap new node with parent
```

Add a node to a Heap

- Example: adding the value 7 to the min heap:

2.1. compare the new node (7) with its parent (10) and see that they needed to be swapped to maintain the min heap property:

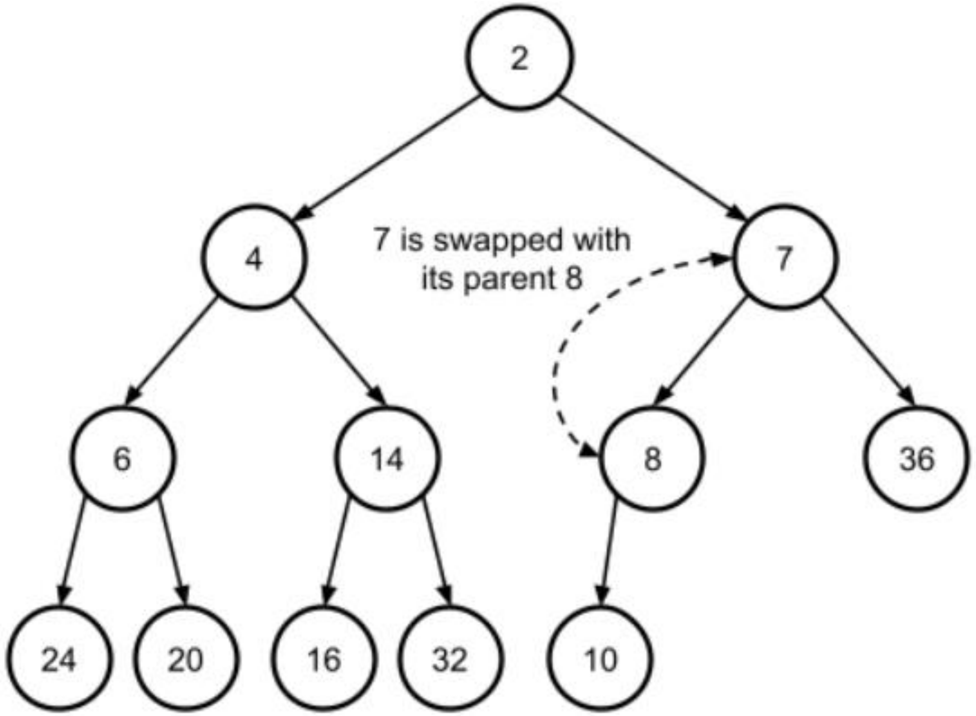


```
while new priority value < parent's priority value:
    swap new node with parent
```

Add a node to a Heap

- Example: adding the value 7 to the min heap:

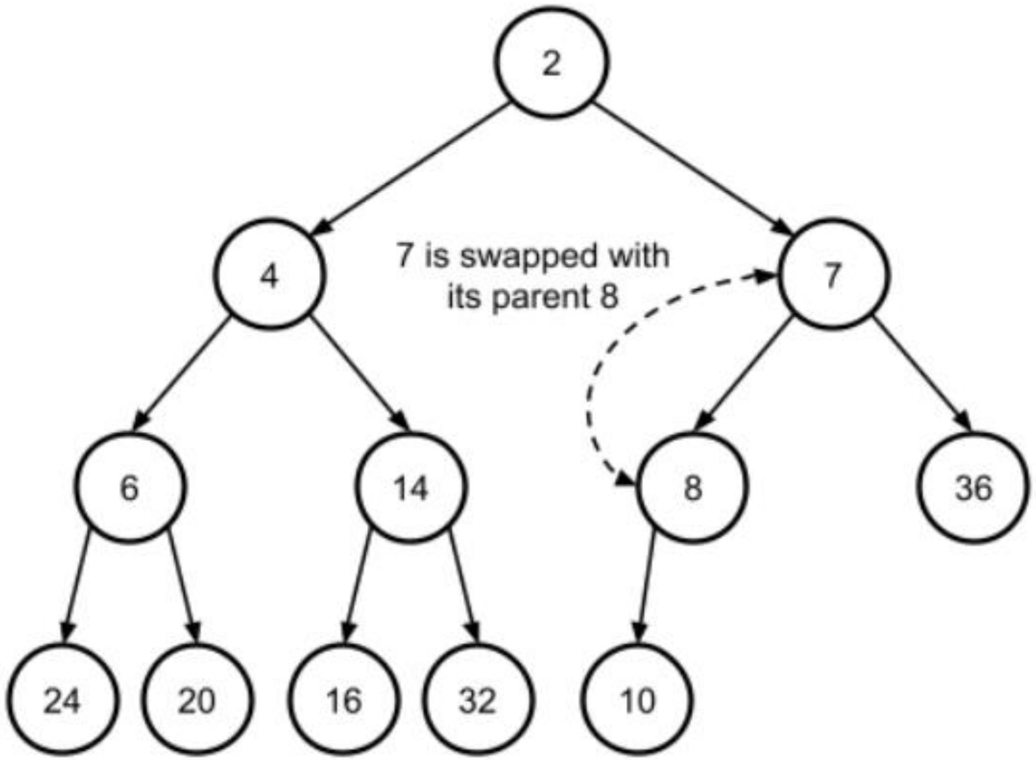
2.2. compare the new node (7) with its new parent (8) and see that they too needed to be swapped:



```
while new priority value < parent's priority value:
    swap new node with parent
```

Add a node to a Heap

- Runtime Complexity of percolation: O(log n)



Remove a node from a Heap

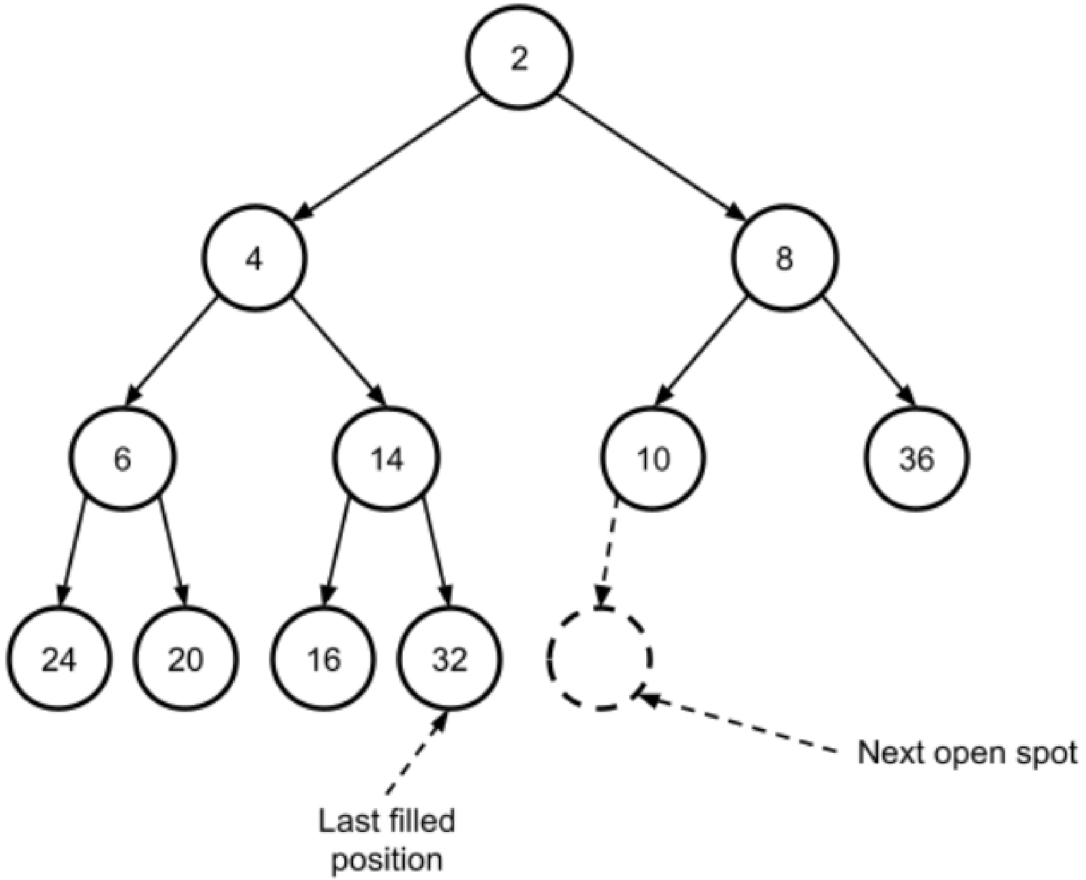
- In a min heap, the root node's priority value is always the lowest
  - the `first()` and `remove_first()` always access and remove the root node

- Question: If we always remove the root node, how do we replace it?
  - Remember, we need to maintain the completeness of the binary tree.

- Answer: replace it with the element last added to the heap and then fix the heap by percolating that node down

Remove a node from a Heap

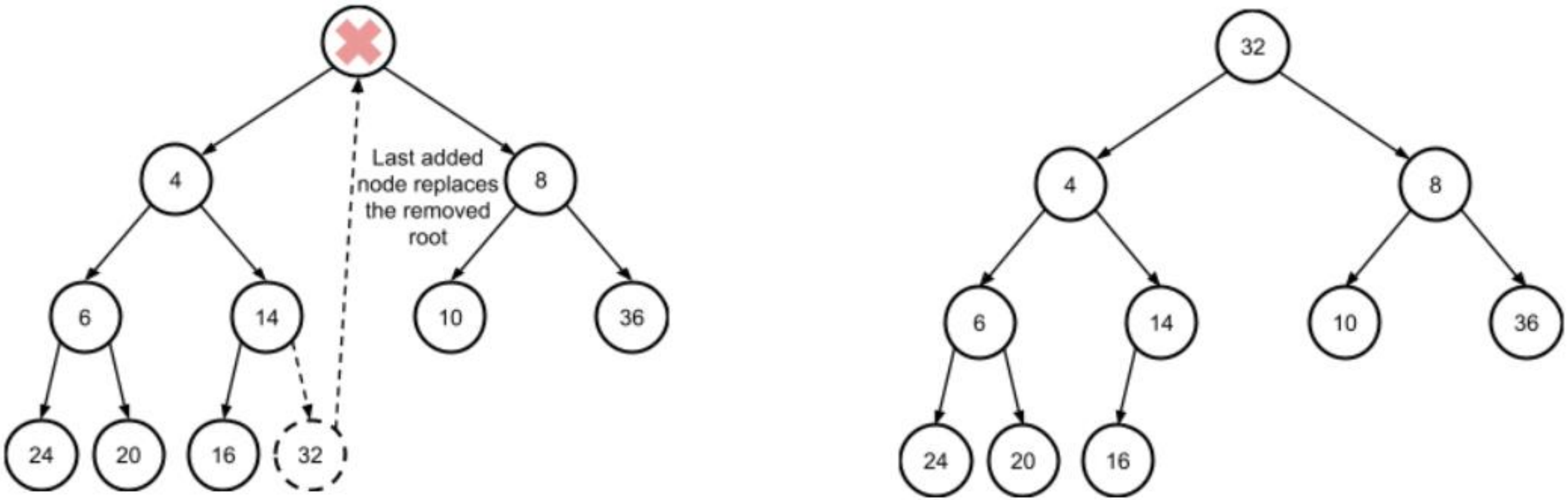
- Example: remove the root node (2) from that heap:



Remove a node from a Heap

- Example: remove the root node (2) from that heap:

1. replace it with the last added node (32)

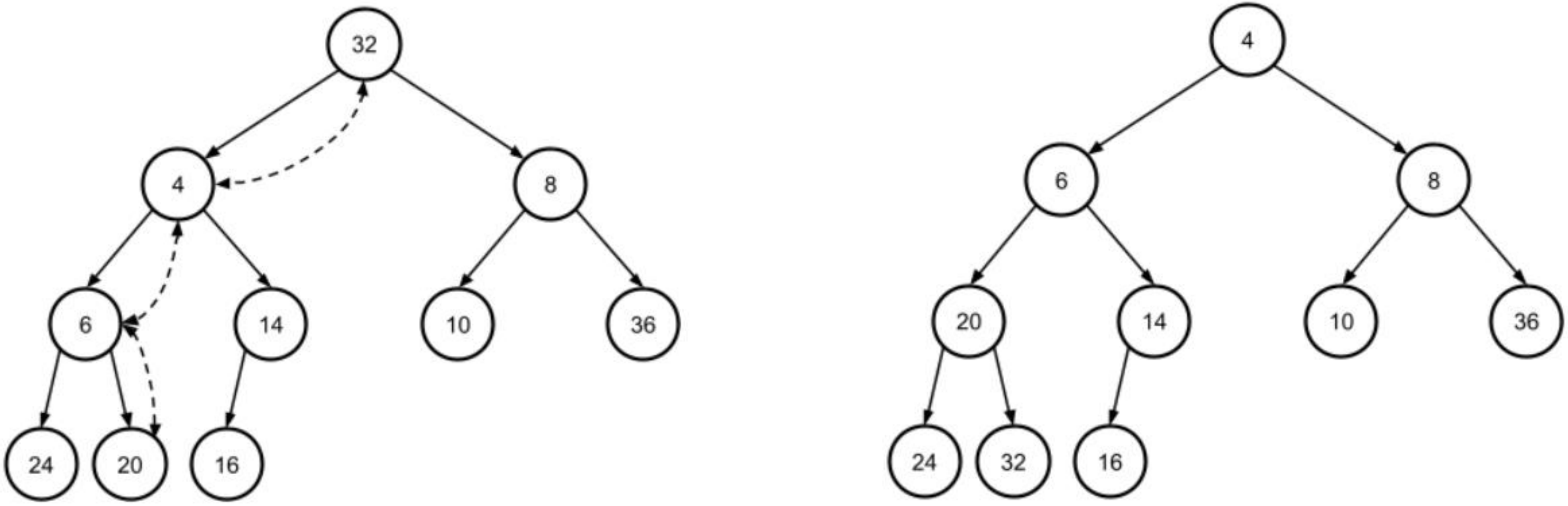


```
while priority > smallest child priority:
        swap with smallest child
```

Remove a node from a Heap

- Example: remove the root node (2) from that heap:

2. percolate the replacement node down the tree



Lecture Topics:

- Priority Queues & Heaps
- Array-based Heaps
- Build a heap from an arbitrary array
- Heapsort

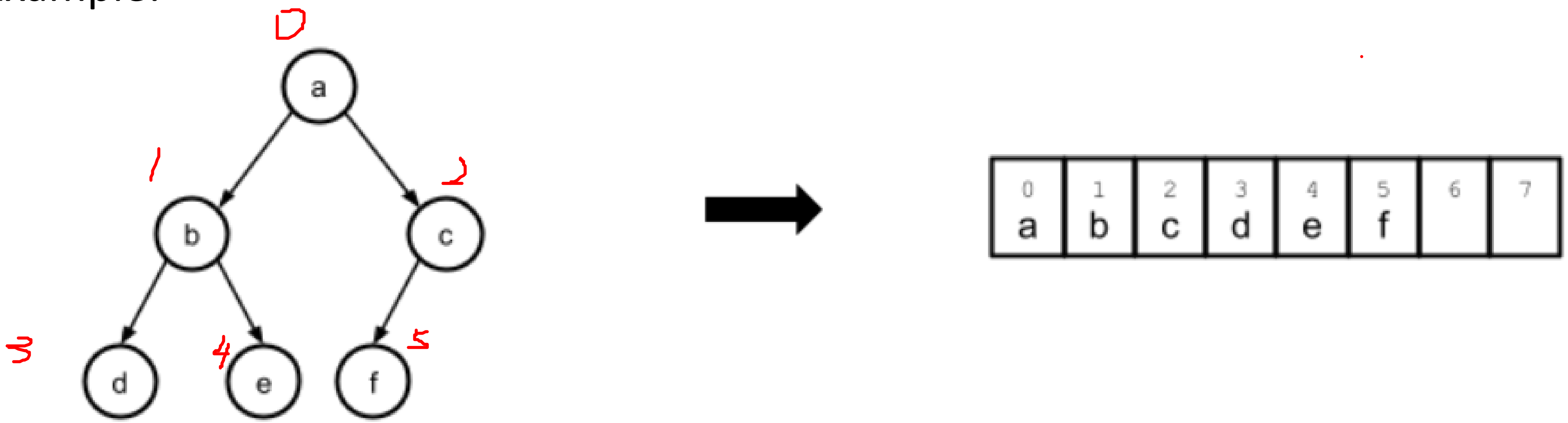
Implement a Heap

- Many ways to implement a heap…
- Recall: a heap data structure contains a complete binary tree
- Then…

Implement a Heap

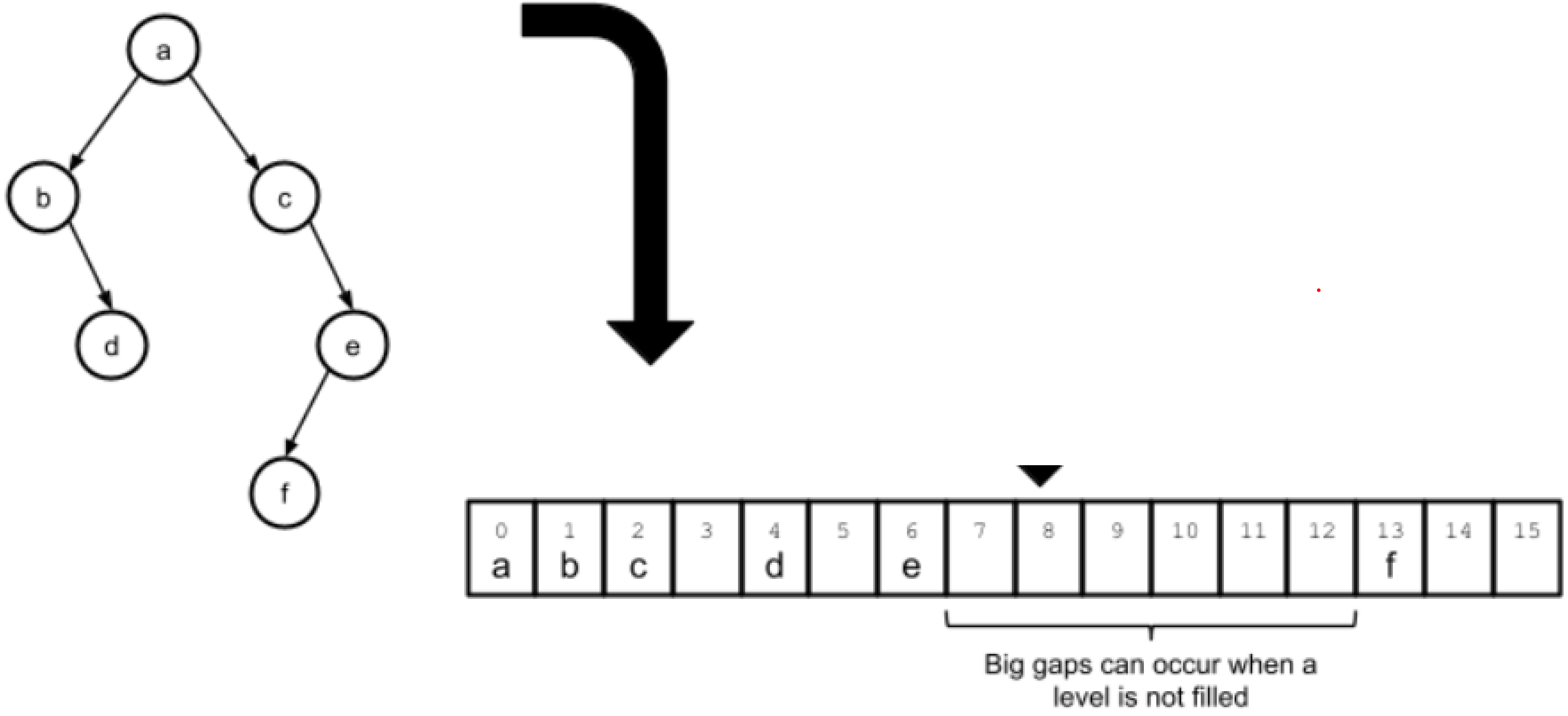
- Implement the complete binary tree representation of a heap using an array:
  - root node of the heap is stored at index 0
  - The left and right children of a node at index i are stored respectively at indices 2 * i + 1 and 2 * i + 2
  - The parent of a node at index i is at (i - 1) / 2 (using the floor that results from integer division).

- Example:



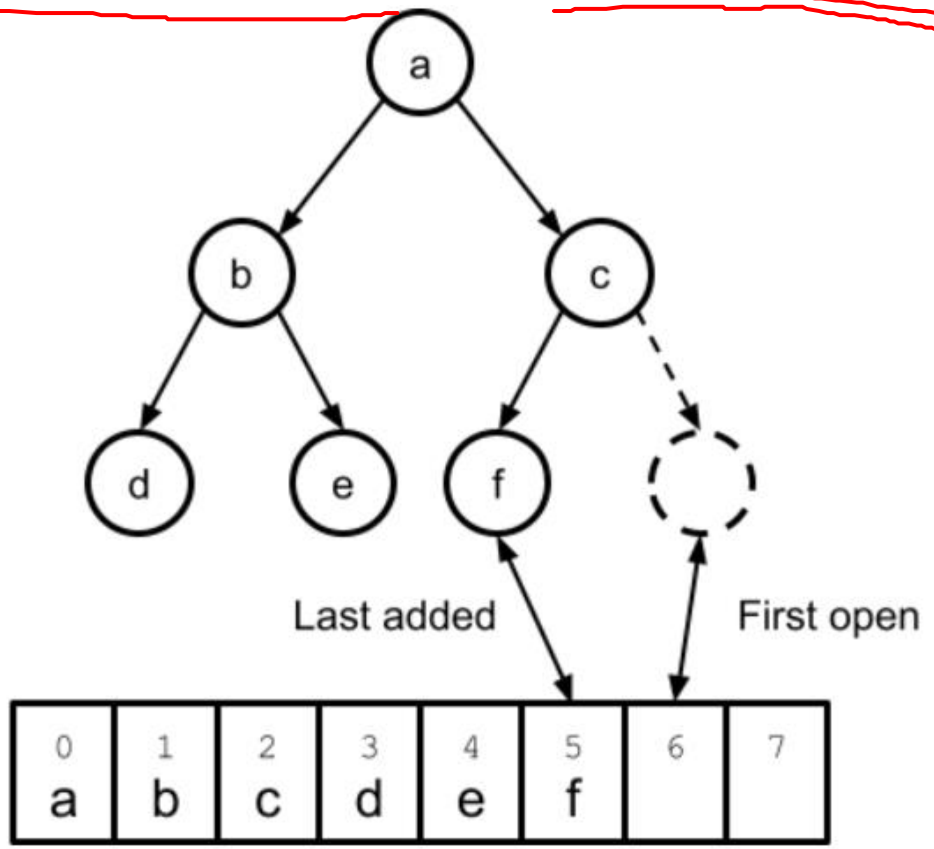
Implement a Heap

- Q: Can you implement a binary tree that was not complete using an array?
- A: No!
- Example:



Implement a Heap

- Keeping track of the last added element and the first open spot in the array representation of the heap is simple
  - simply the last element in the array and the following empty spot
- Example:

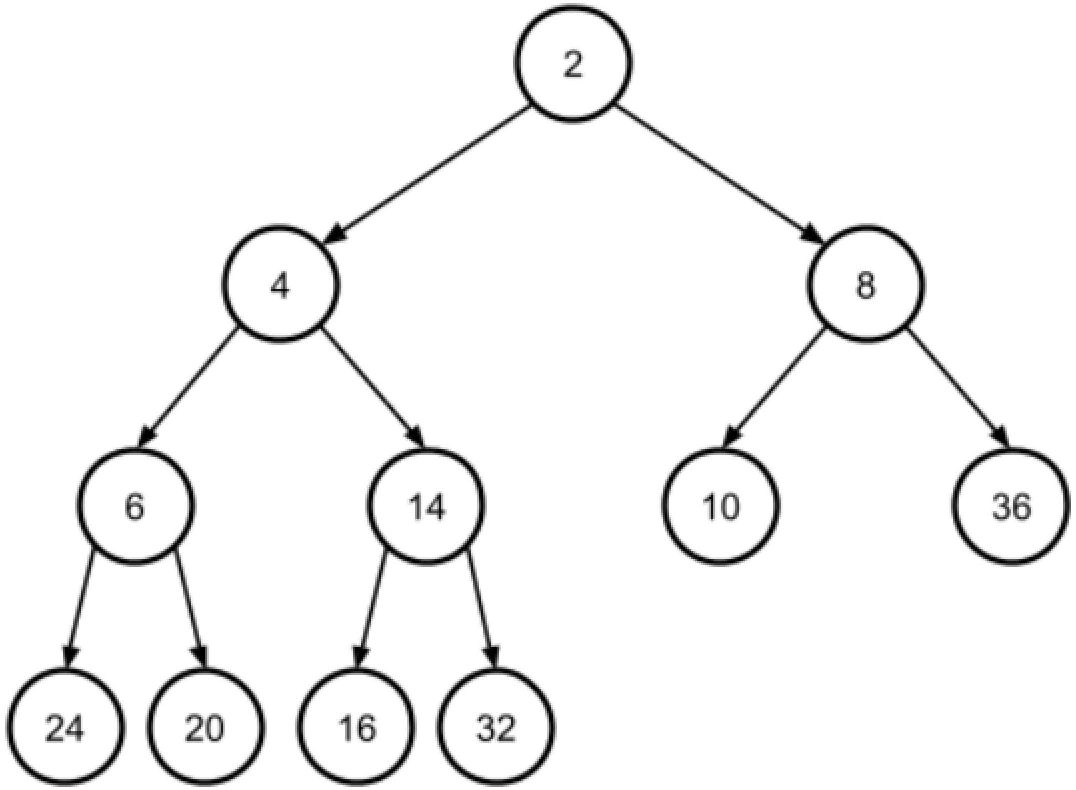


Inserting into an array-based Heap

- Inserting an element into the array representation of the heap follows this procedure:
  1. Put new element at the end of the array. at idx size
  2. Compute the inserted element's parent index `((i - 1) / 2)`.
  3. Compare the value of the inserted element with the value of its parent.
  4. If the value of the parent is greater than the value of the inserted element, swap the elements in the array and repeat from step 2.
     - Do not repeat if the element has reached the beginning of the array.

Inserting into an array-based Heap

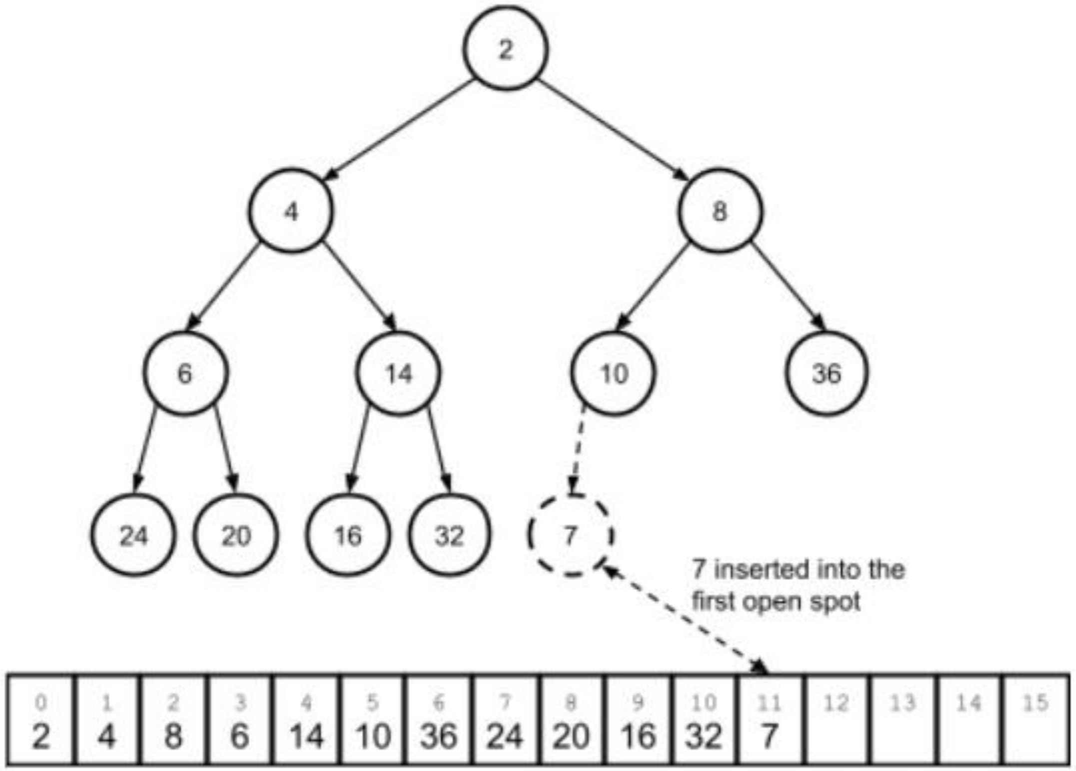
- Example: added 7 to the following heap



Inserting into an array-based Heap

- Example: added 7 to the following heap

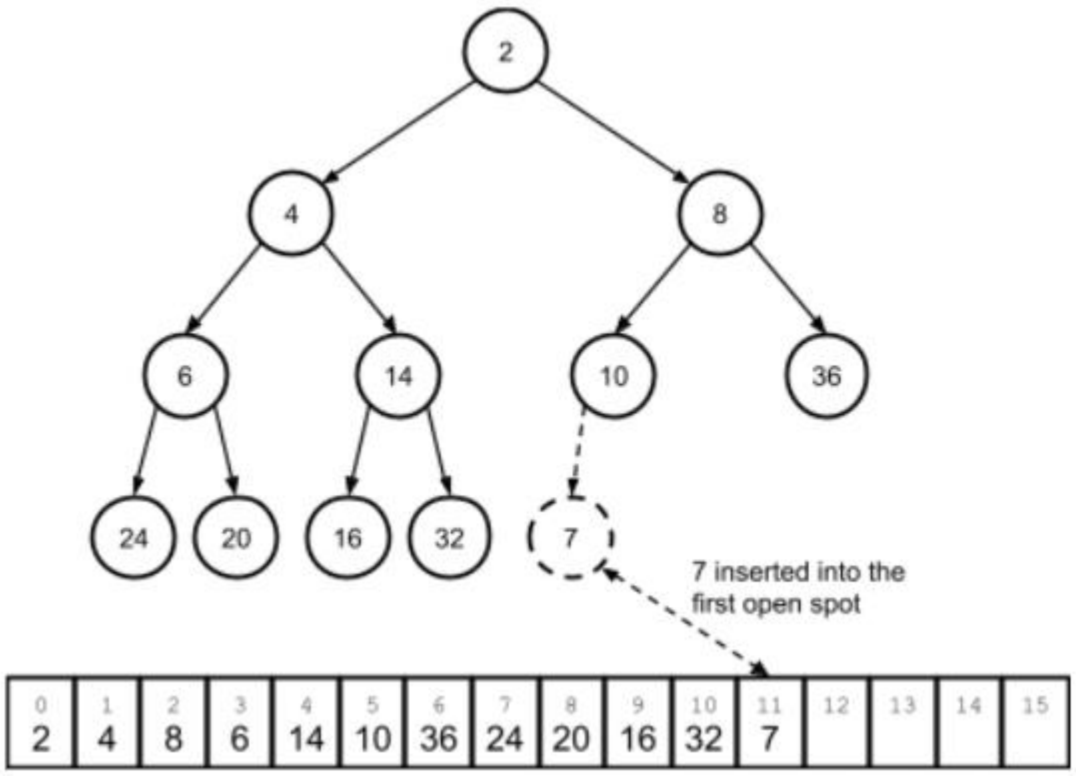
1. insert the new element into the end of the array



Inserting into an array-based Heap

- Example: added 7 to the following heap

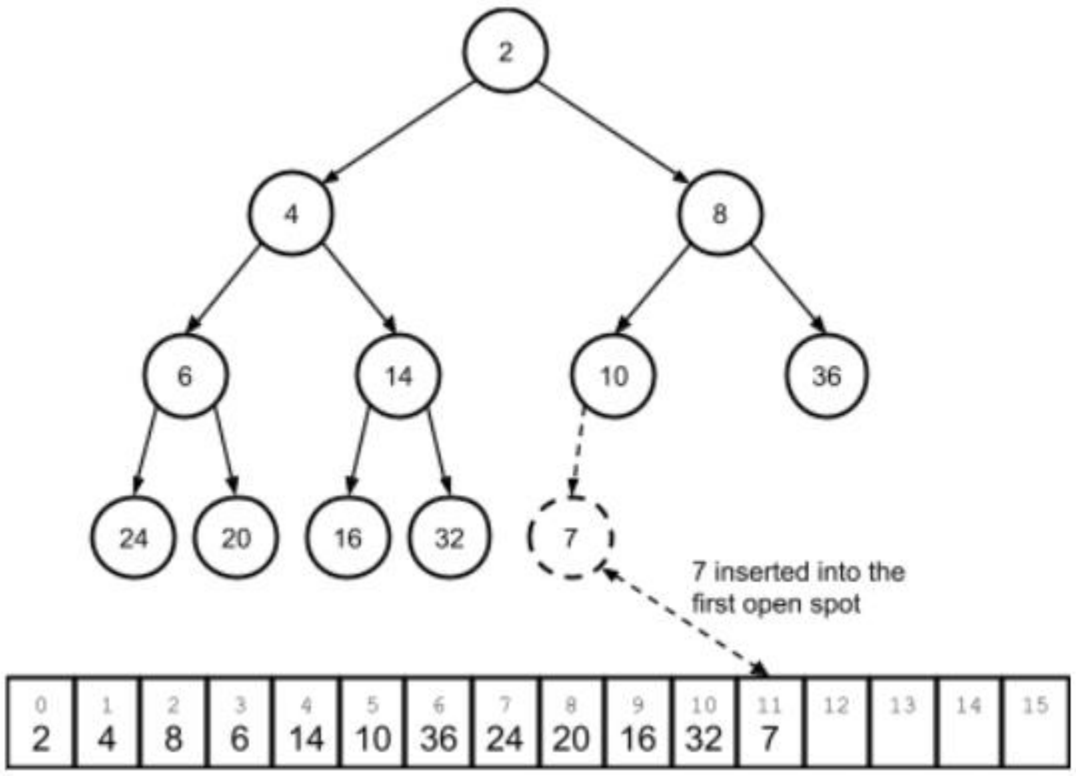
2. compute the index of 7's parent node ((11 - 1) / 2 → 5)



Inserting into an array-based Heap

- Example: added 7 to the following heap

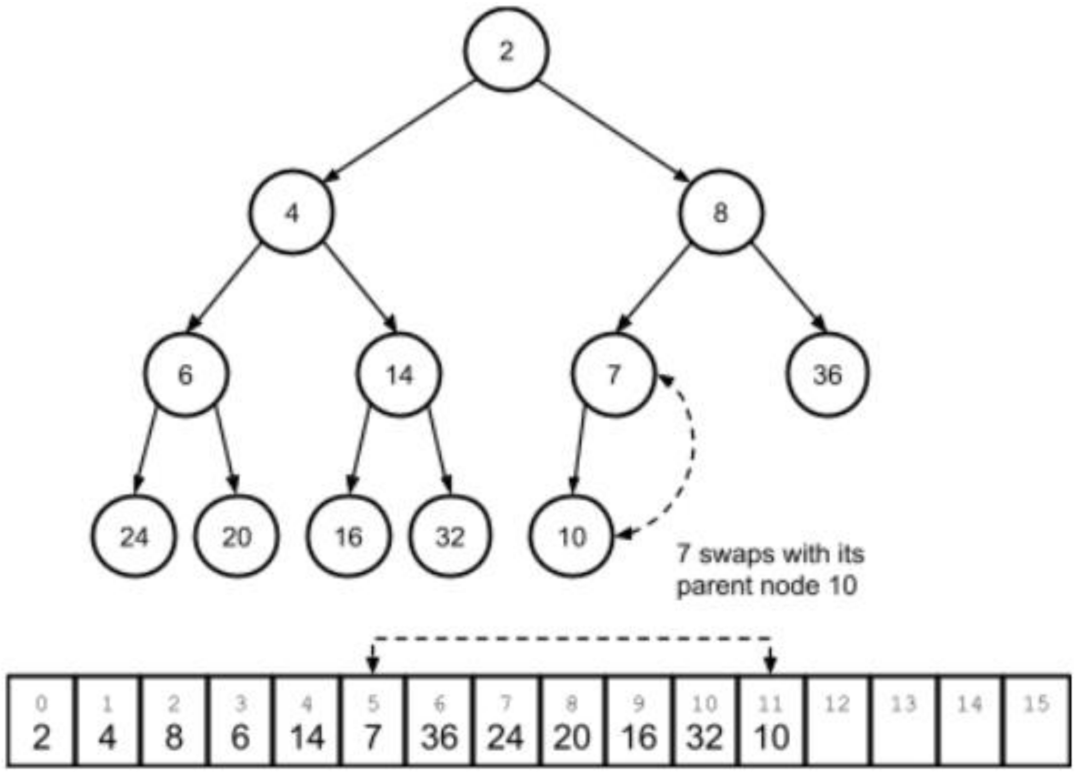
3. compare 7 with the value we found there (at index 5 → 10)



Inserting into an array-based Heap

- Example: added 7 to the following heap

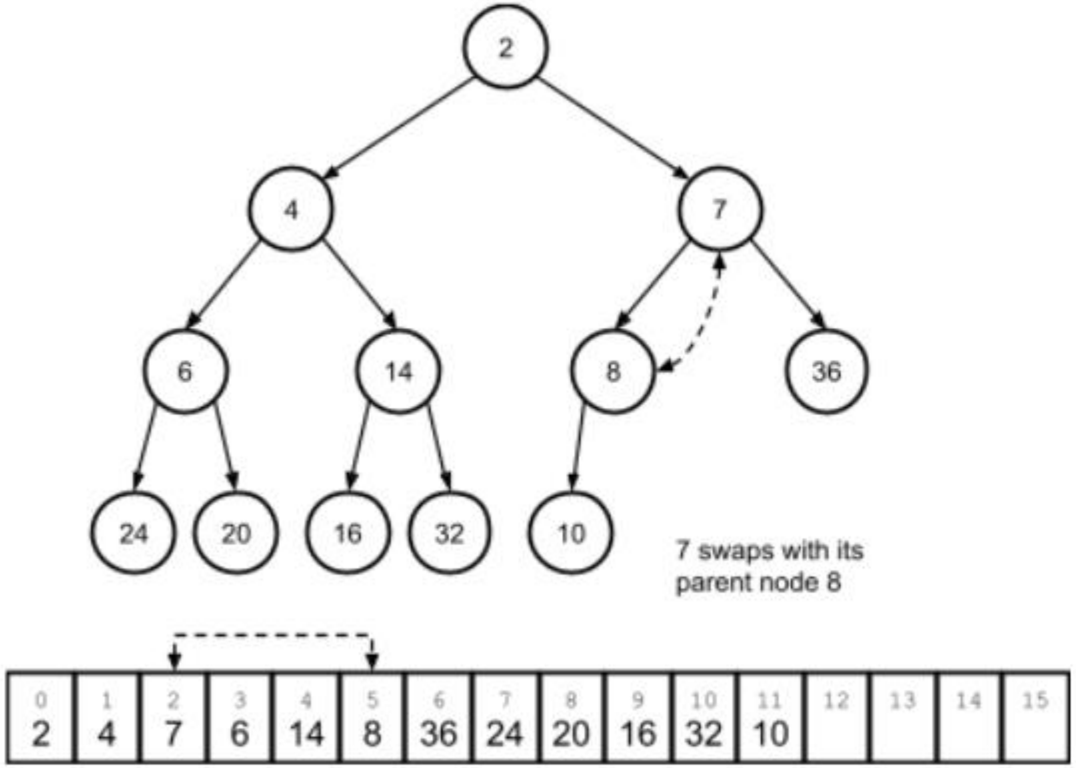
4. Since 7 is less than 10, swap them



Inserting into an array-based Heap

- Example: added 7 to the following heap

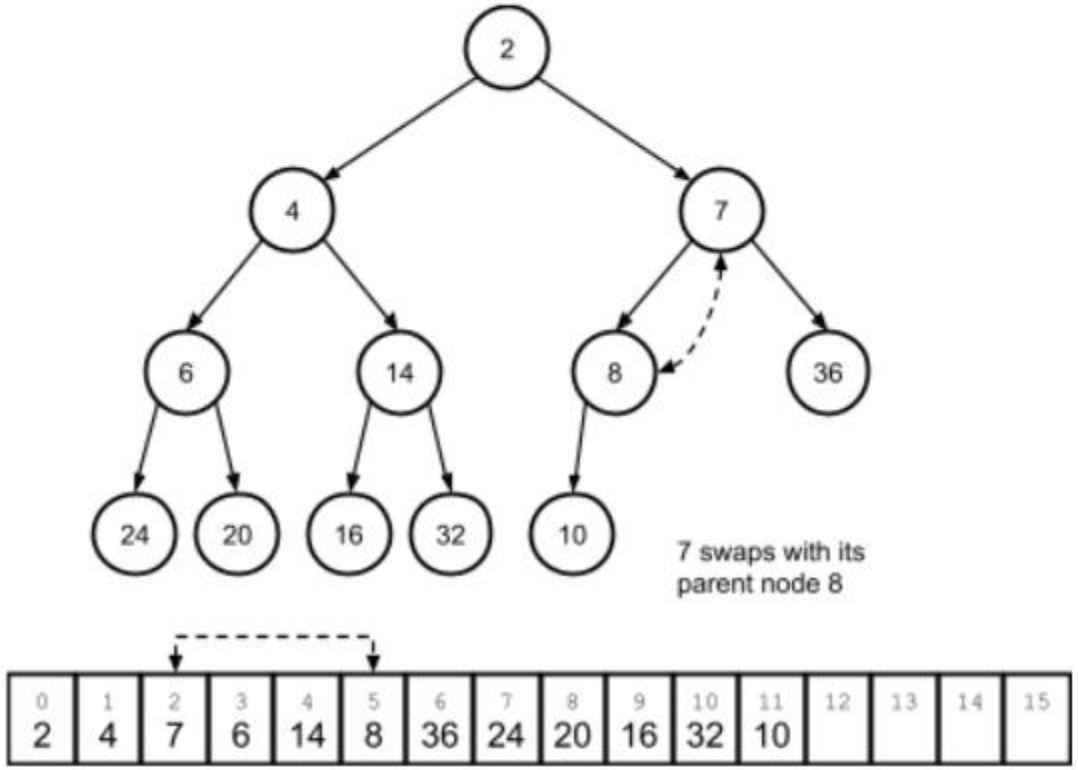
5. Repeat, comparing 7 to its new parent 8 at index (5 - 1) / 2 → 2, and swap again



Inserting into an array-based Heap

- Example: added 7 to the following heap

6. Repeat, compare to 7’s new parent node 2 at index (2 - 1) / 2 → 0, and we’d stop, since 2 is less than 7

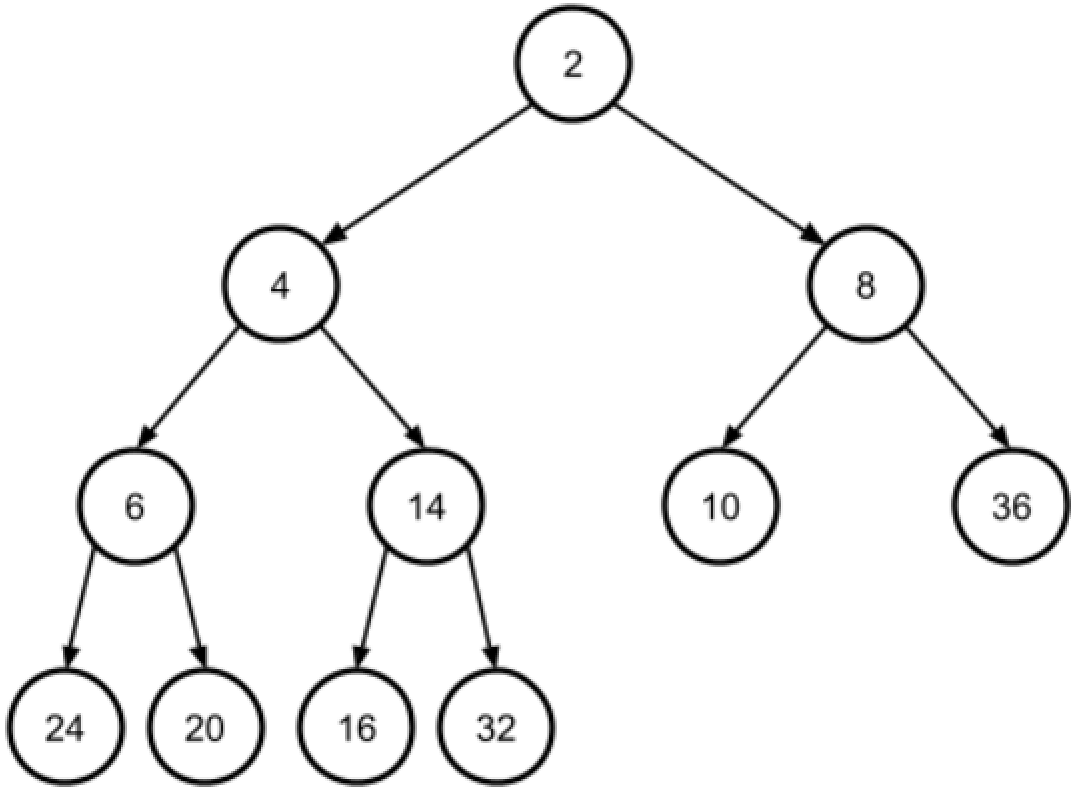


Removing from an array-based Heap

- Recall: in min heap, always remove the node with the lowest priority value (i.e., root)
- Remove an element from the array representation of the heap follows this procedure:
  1. Remember the value of the first element in the array (to be returned later).
  2. Replace the value of the first element in the array with the value of the last element and remove the last element.
  3. If the array is not empty (i.e. it started with more than one element), compute the indices of the children of the replacement element (2 * i + 1 and 2 * i + 2).
     - If both of these elements fall beyond the bounds of the array, stop here.
  4. Compare the value of the replacement element with the minimum value of its two children (or possibly one child).
  5. If the replacement element's value is greater than its minimum child's value, swap those two elements in the array and repeat from step 3

Removing from an array-based Heap

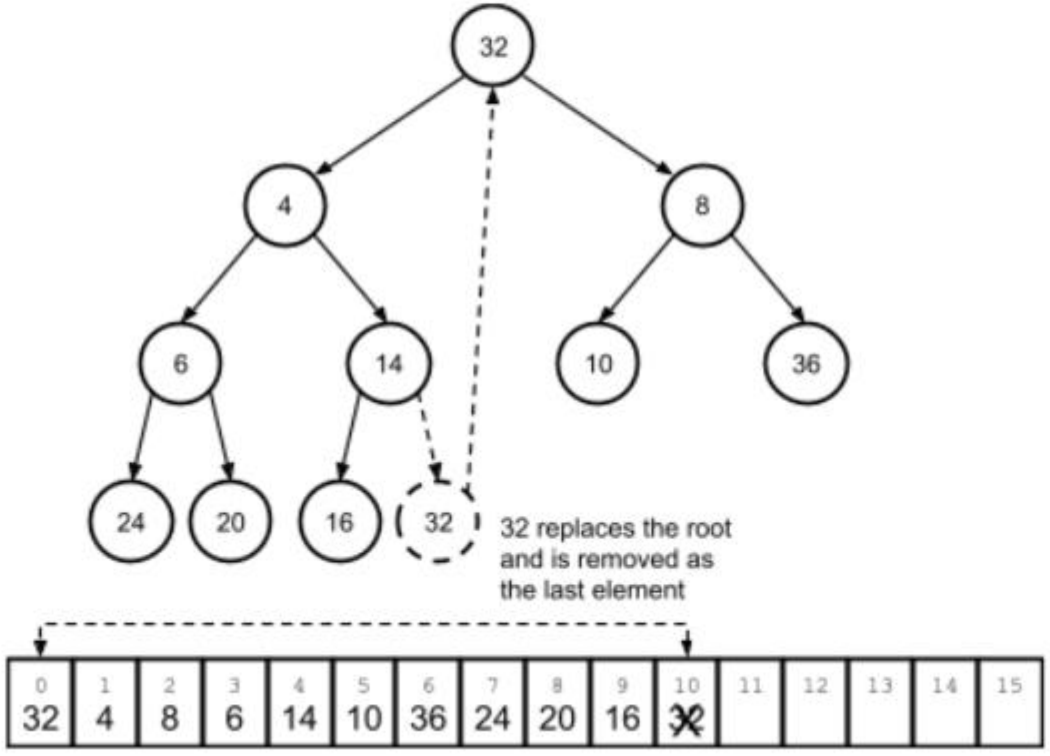
- Example: removing the root (2) from the following heap



Removing from an array-based Heap

- Example: removing the root (2) from the following heap

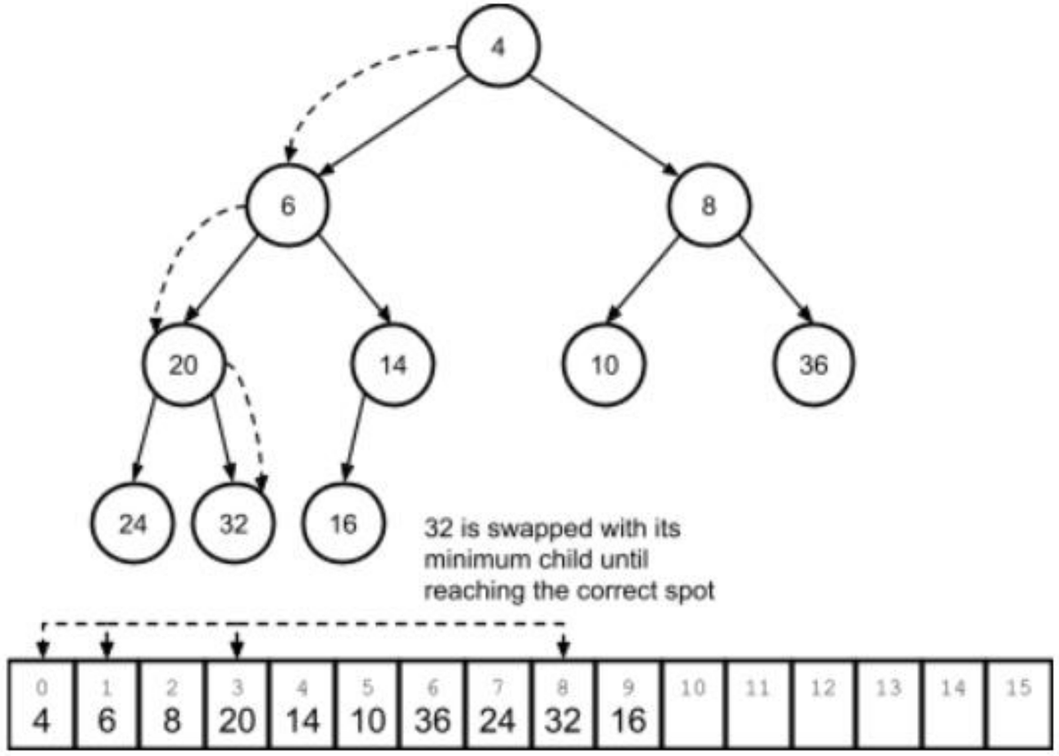
1. replacing the root (the first element in the array) with the last element and then removing the last element



Removing from an array-based Heap

- Example: removing the root (2) from the following heap

2. percolate 32 down the array, comparing it to its minimum-value child and swapping values in the array until 32 reached its correct place



Lecture Topics:

- Priority Queues & Heaps
- Array-based Heaps
- Build a heap from an arbitrary array
- Heapsort

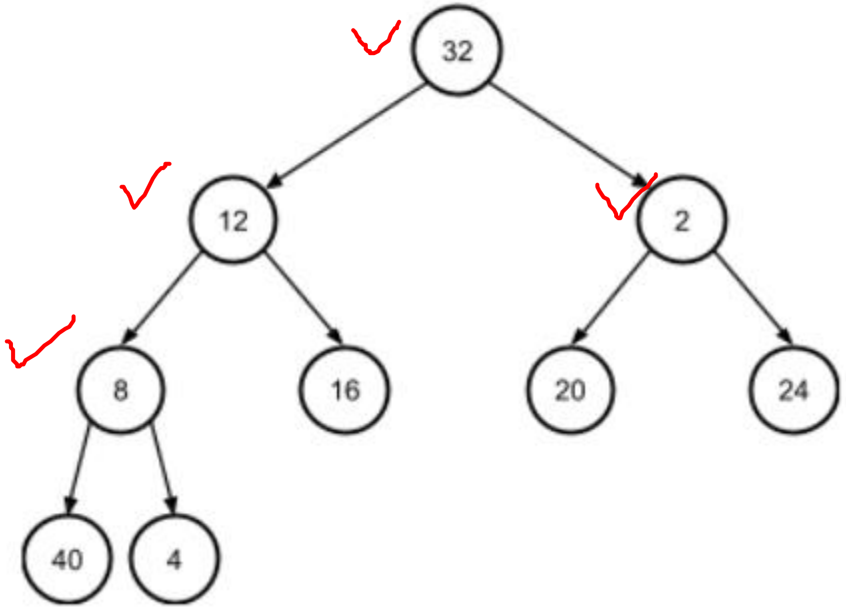
Building a heap from an arbitrary array

- Example: Convert the following arbitrary array to a heap:

0	1	2	3	4	5	6	7	8
32	12	2	8	16	20	24	40	4

complete binary tree.

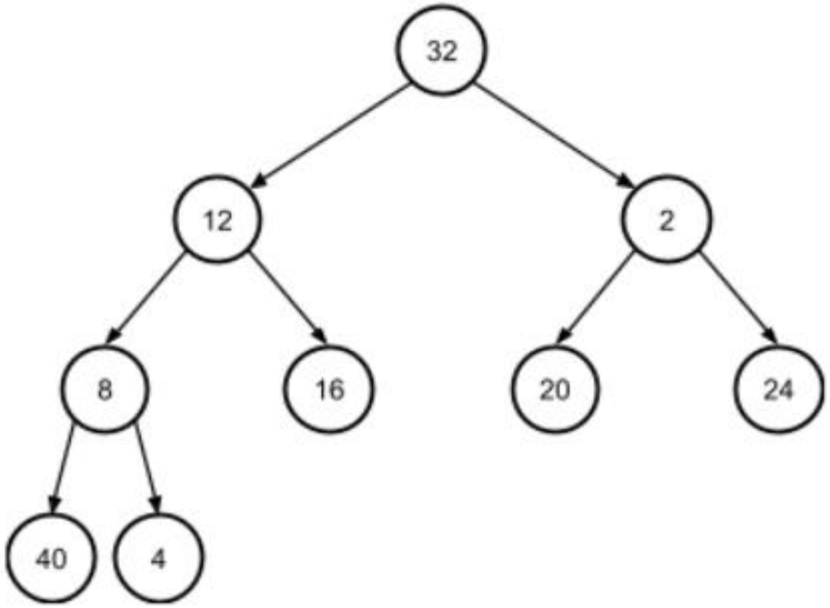
  - First, consider this arbitrary array as a ~~heap~~:



Building a heap from an arbitrary array

- Percolate down the first non-leaf element, then the subtree rooted at that element's original position will be a proper heap
  - first non-leaf element (from the back of the array) is at n / 2 - 1

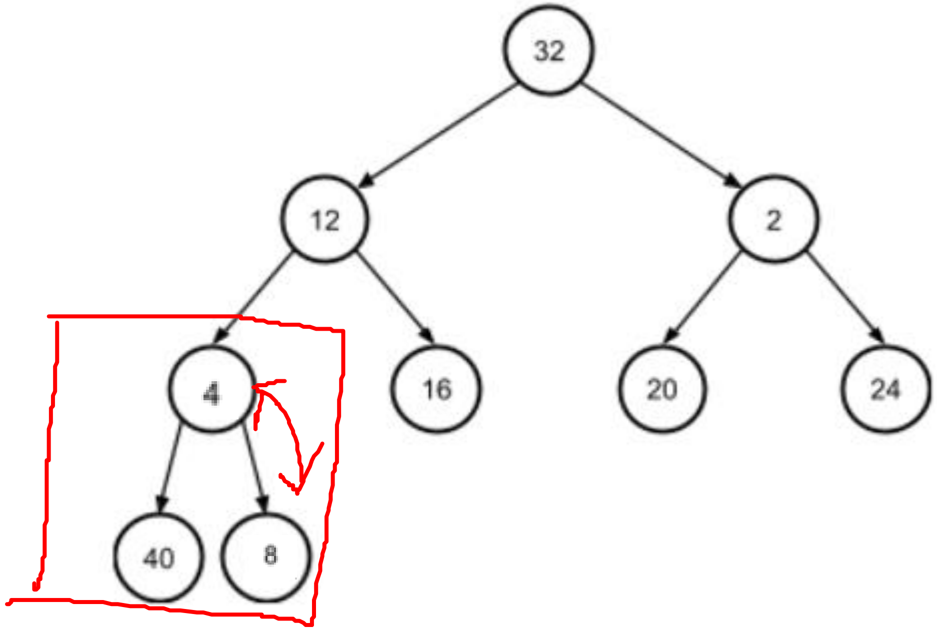
0	1	2	3	4	5	6	7	8
32	12	2	8	16	20	24	40	4



Building a heap from an arbitrary array

- Percolate down the first non-leaf element, then the subtree rooted at that element's original position will be a proper heap
  - first non-leaf element (from the back of the array) is at n / 2 - 1

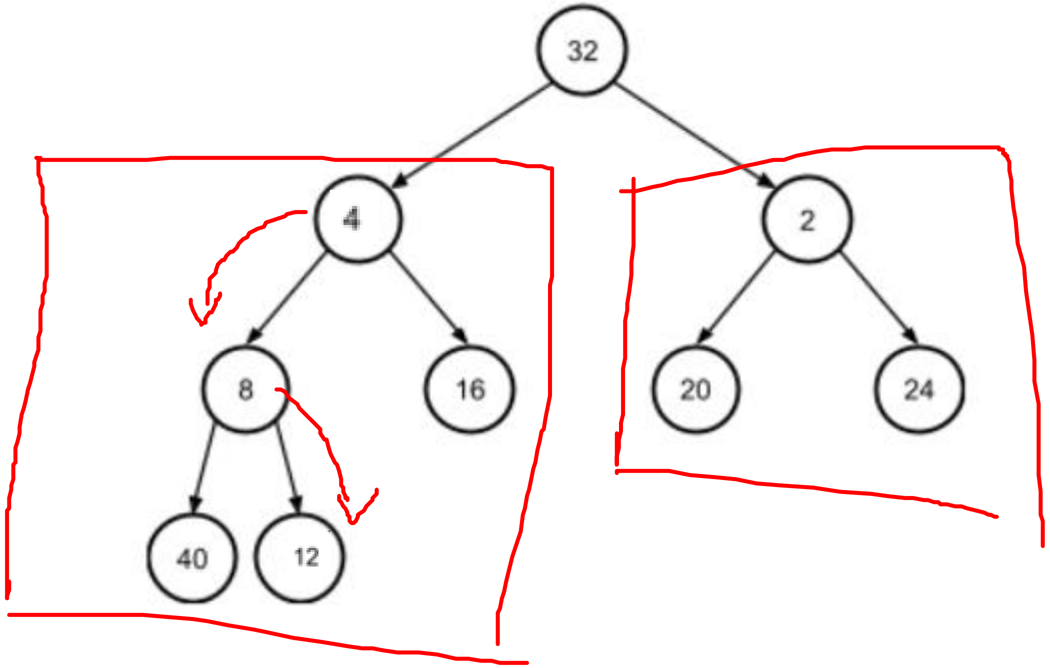
0	1	2	3	4	5	6	7	8
32	12	2	4	16	20	24	40	8



Building a heap from an arbitrary array

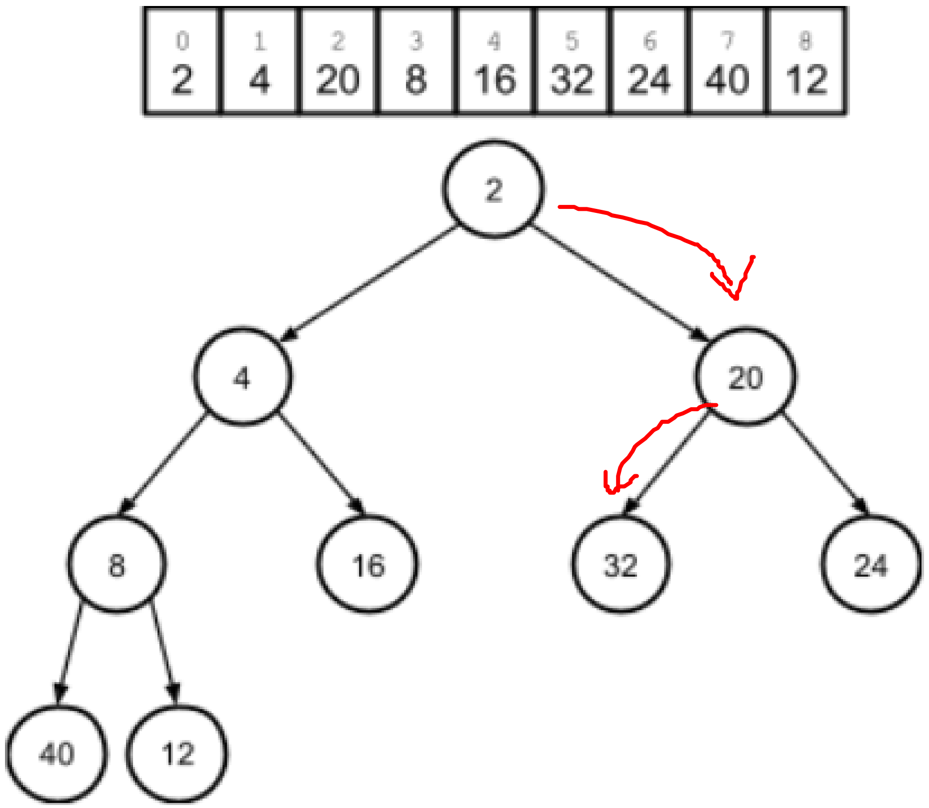
- Percolate down the first non-leaf element, then the subtree rooted at that element's original position will be a proper heap
  - first non-leaf element (from the back of the array) is at n / 2 - 1

0	1	2	3	4	5	6	7	8
32	4	2	8	16	20	24	40	12



Building a heap from an arbitrary array

- Once we percolate down the root element, the entire array will represent a proper heap



Building a heap from an arbitrary array

- Time Complexity:
  - perform n / 2 downward percolation operations.
  - Each of these operations is O(log n).
  - This means the total complexity is O(n log n).

- Space Complexity:
  - No additional space needed and no recursive calls: O(1)