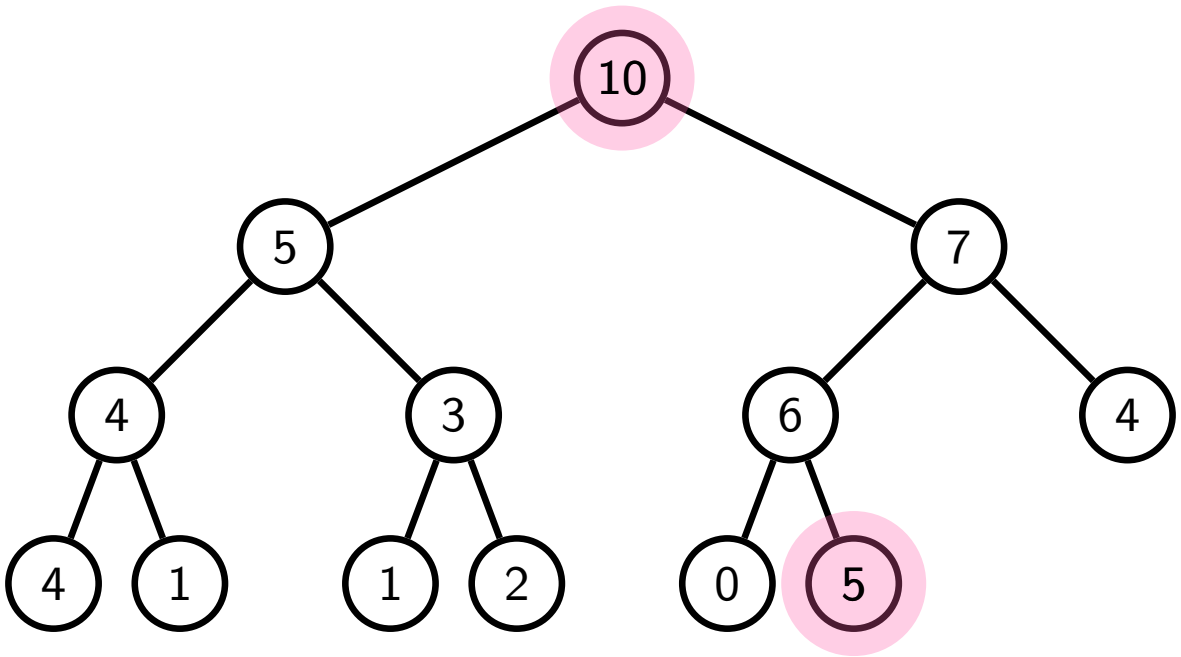
Binary Heap Tree Deletion

Delete Root

Idea:

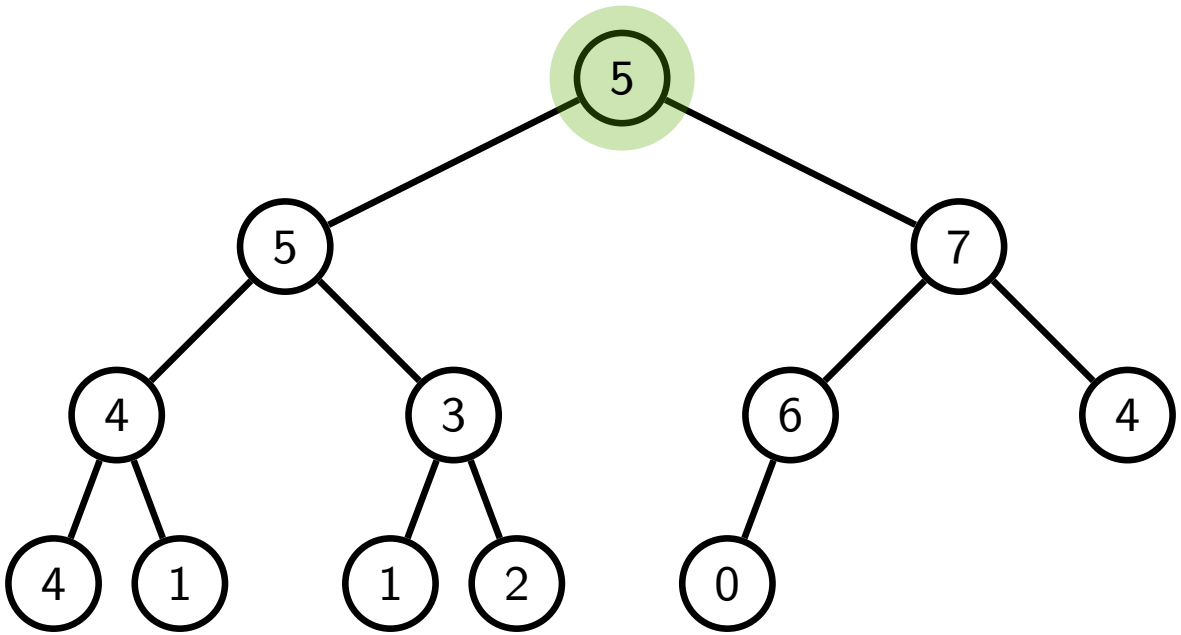
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

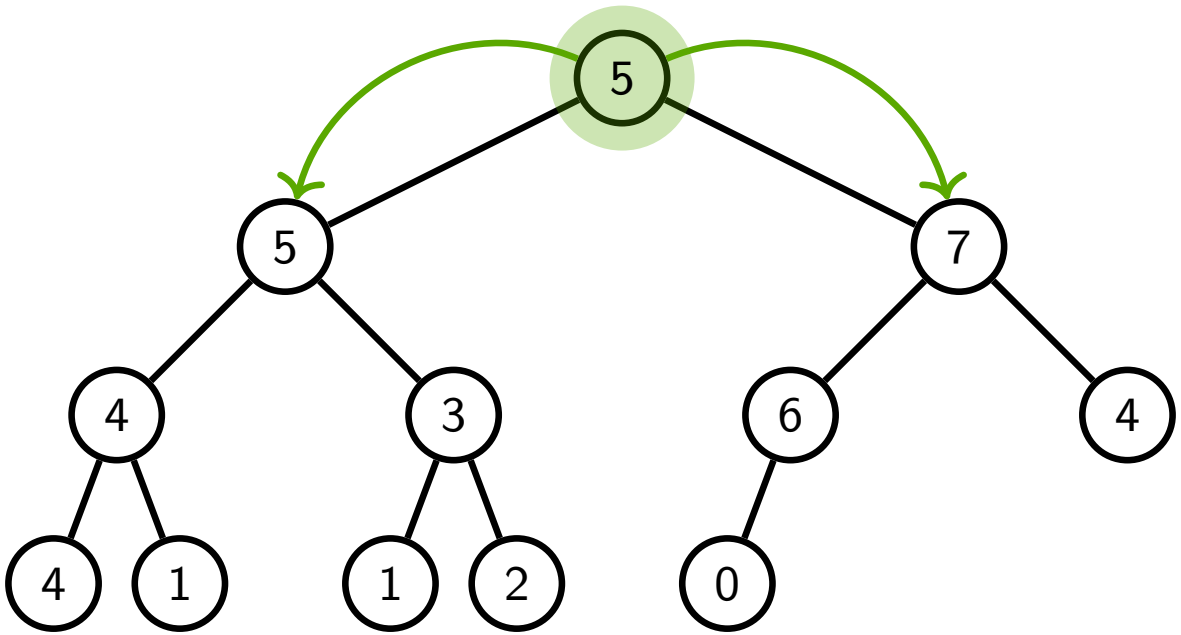
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

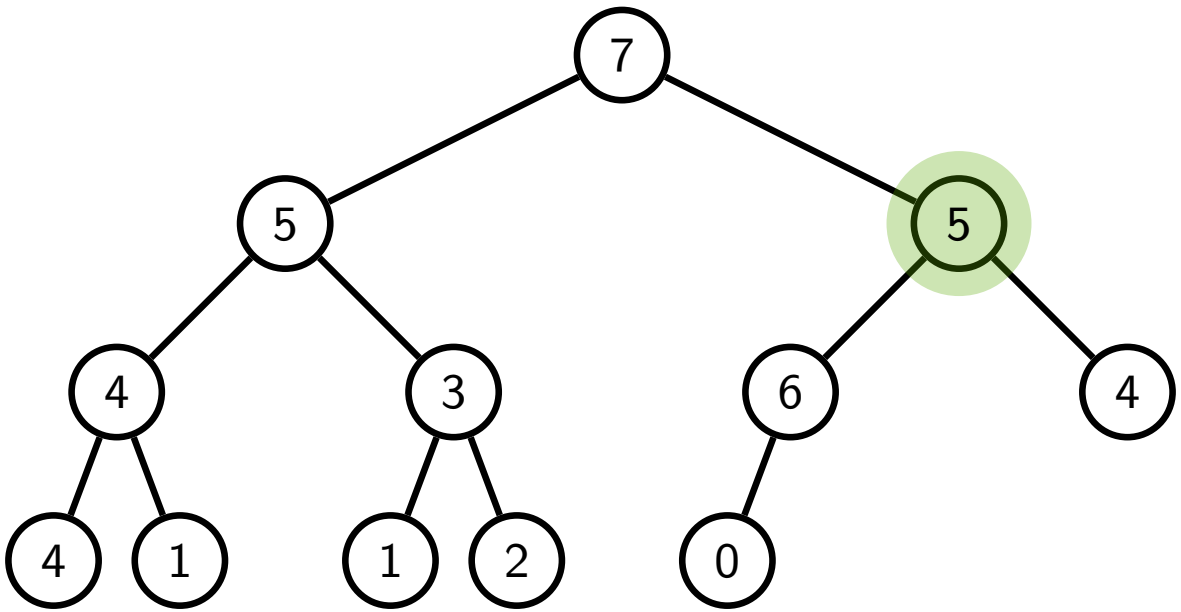
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

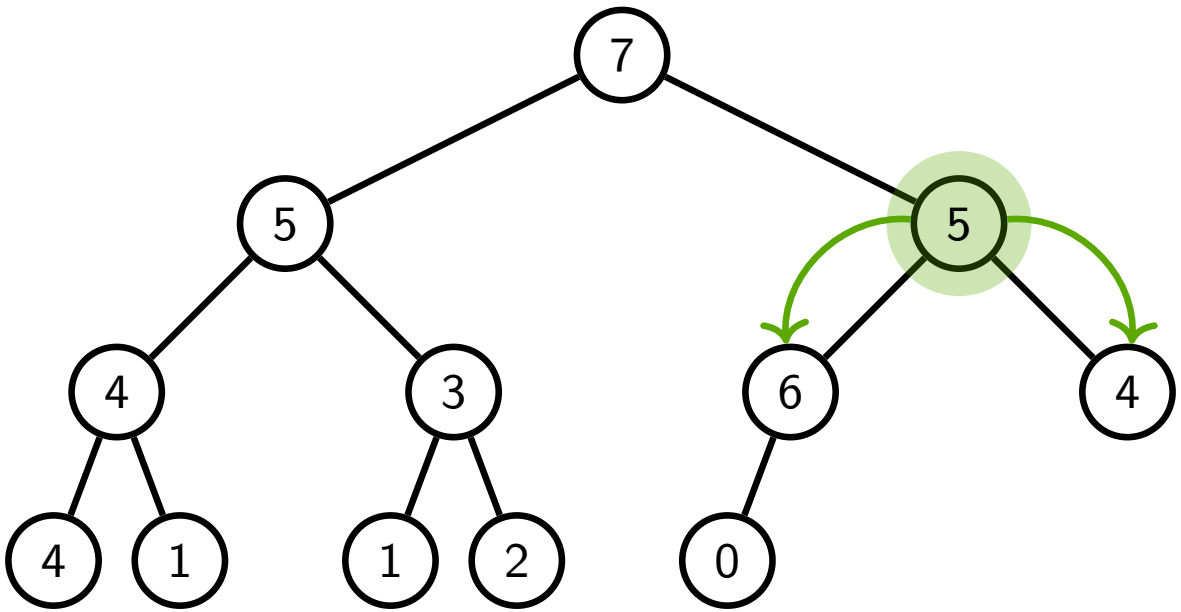
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

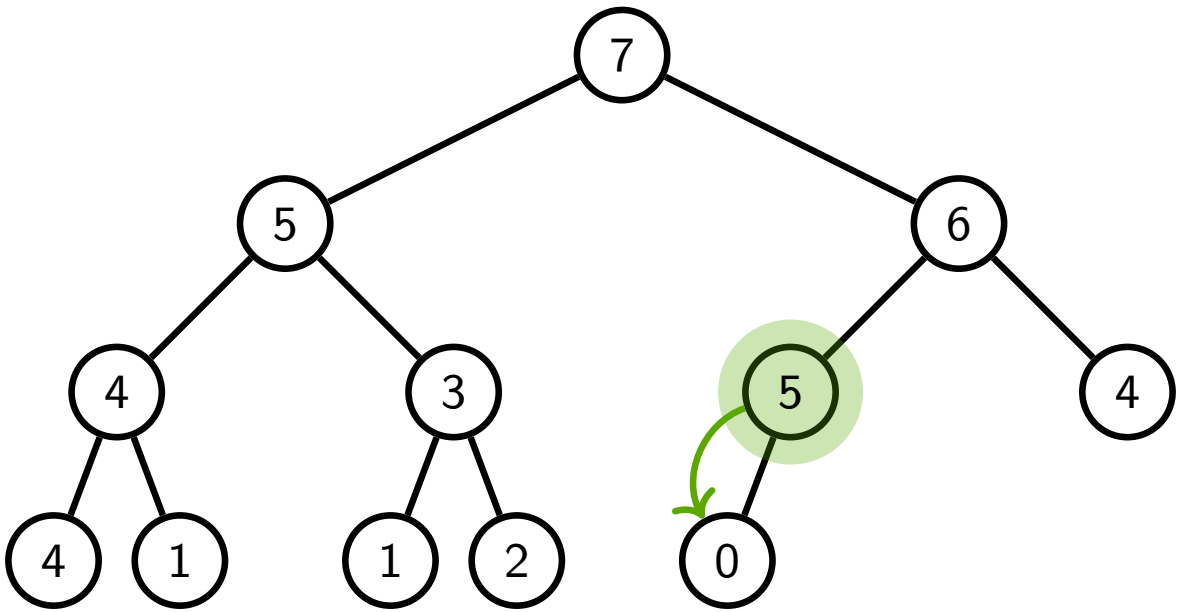
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

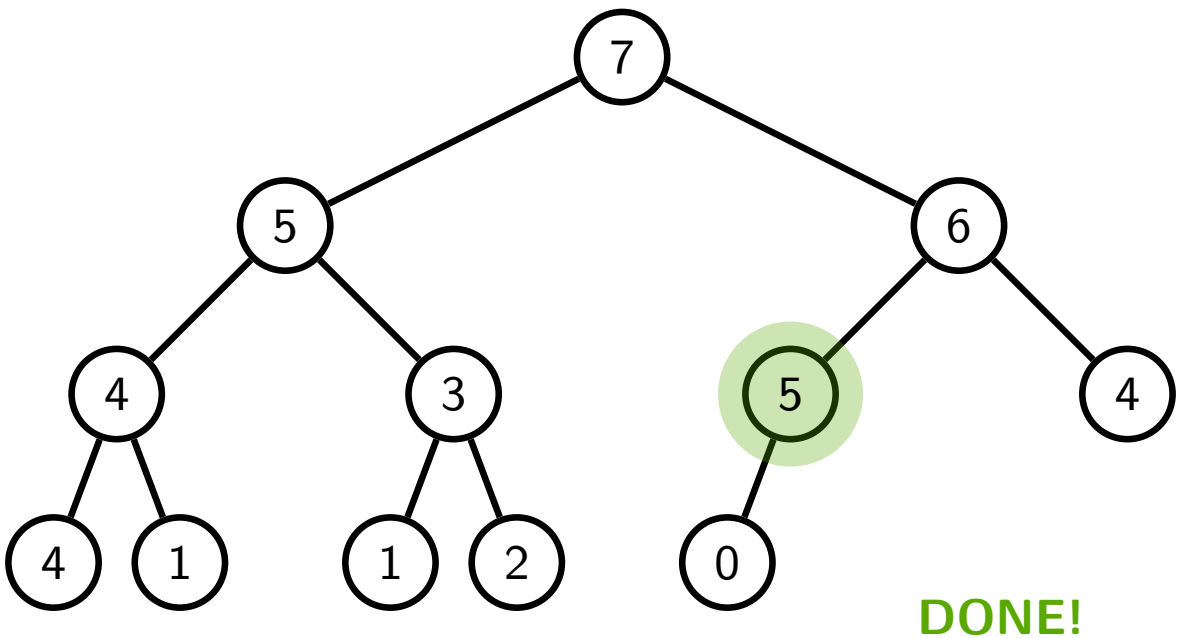
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root

Idea:

1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



Delete Root (pseudocode)

```
1 public void deleteRoot() {
2   if (n < 1)
3     throw EmptyHeapException;
4
5   heap[1] = heap[n];
6   n = n-1;
7   bubbleDown(1);
8 }
```

bubbleDown

The `bubbleDown` method needs to deal with 5 cases, depending on the current node that is being bubbled down:

1. it is a leaf node: nothing to do, `bubbledown` is complete
2. it has a left child, but no right child: if left child is greater than it, swap left child with current and `bubbleDown` is complete (no right child and this is a complete tree means the left child cannot have any children)
3. it has two children, the left child is greater of the two and is greater than the current node: swap left child and current and continue recursively bubbling down the left child
4. as the previous case but the right child is the greater: continue as above but on the right child
5. it has two children and neither is greater than the current node: nothing to do, `bubbledown` is complete

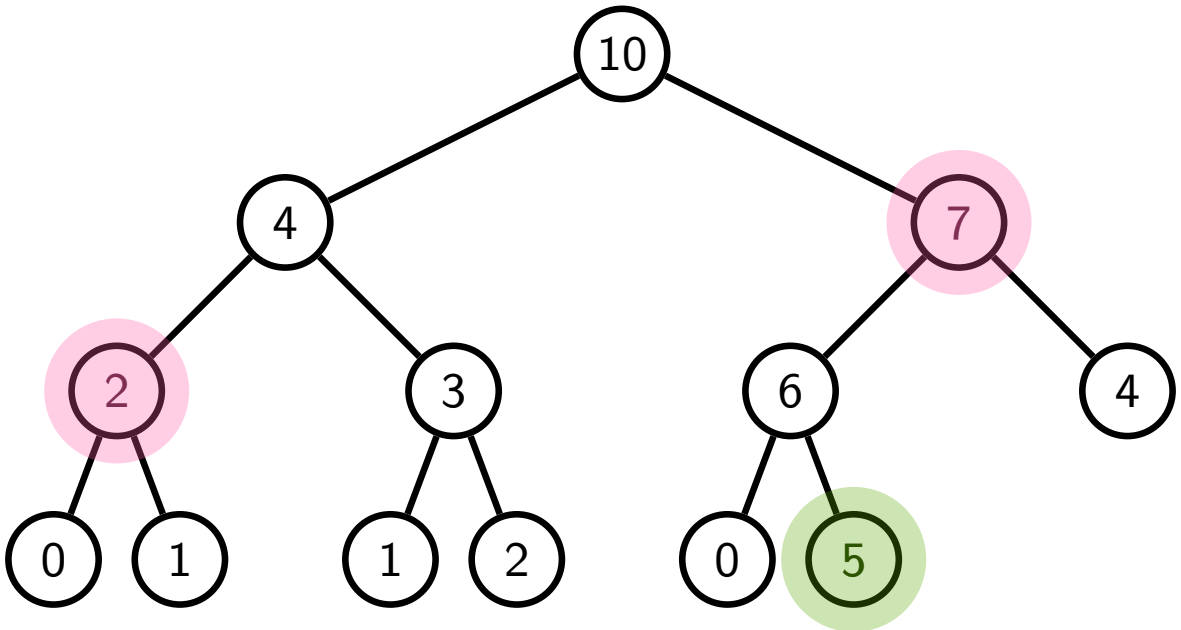
bubbleDown (pseudocode)

```
1  private void bubbleDown(int i) {
2     if ( left(i) > n )                    // no children
3        return;
4     else if ( right(i) > n )              // only left child
5     {  if ( heap[i] < heap[left(i)] )
6           swap heap[i] and heap[left(i)]
7     }
8     // else two children
9     else if ( heap[left(i)] > heap[right(i)])
10    {  // left child has higher priority than right
11       if (heap[i] < heap[left(i)] )
12       {   swap heap[i] and heap[left(i)]
13           bubbleDown(left(i))
14       }
15    }
16    else // right child has higher priority than left
17       if ( heap[i] < heap[right(i)] )
18       {  swap heap[i] and heap[right(i)]
19          bubbleDown(right(i))
20       }
21    }
22 }
```

Delete

Deleting an arbitrary node means the node might be anywhere in the tree.

1. Remove the last node and use it to replace the node to be deleted
2. This node may be smaller than its children or larger than its parent, so bubble up or bubble down as necessary



Delete (pseudocode)

```
1  public void delete(int i) {
2    if (n < 1)
3      throw EmptyHeapException;
4    if (i < 1 or i > n)
5      throw IndexOutOfBoundsException;
6
7    heap[i] = heap[n];
8    n = n-1;
9    bubbleUp(i)
10   bubbleDown(i);
11 }
```

Note that only one of `bubbleUp` or `bubbleDown` is necessary, but since the one which turns out to be unnecessary will not do anything, it is safe to call both, one after the other, rather than test to check which one should be invoked.

Update

Update means modifying the priority of a element in the tree. This works like `delete` except that we set the priority of the target node from a parameter, rather than from the last element in the tree, and we don't reduce the size of the tree.

```
1  public void update(int i, int priority) {
2    if (n < 1)
3      throw EmptyHeapException;
4    if (i < 1 or i > n)
5      throw IndexOutOfBoundsException;
6
7    heap[i] = priority;
8    bubbleUp(i)
9    bubbleDown(i);
10 }
```