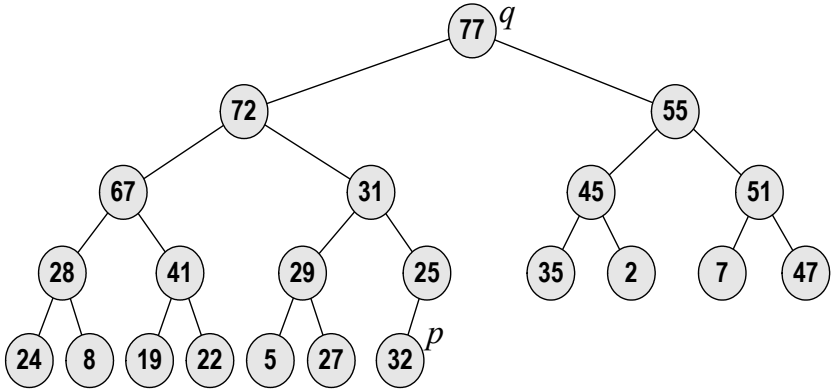
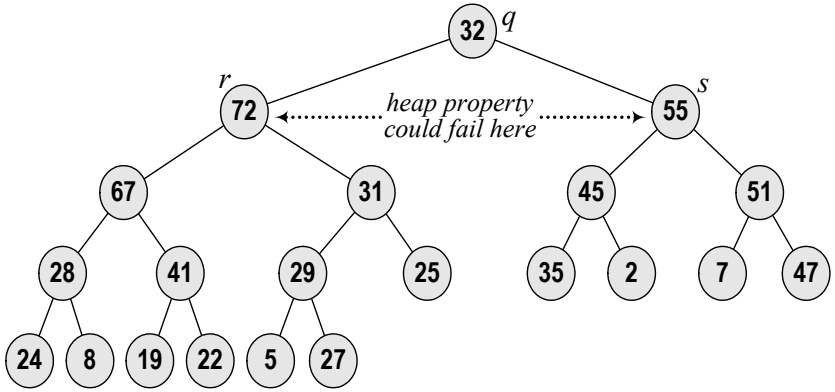
3) ***Remove the largest element***

- Let us remove the largest element from the heap

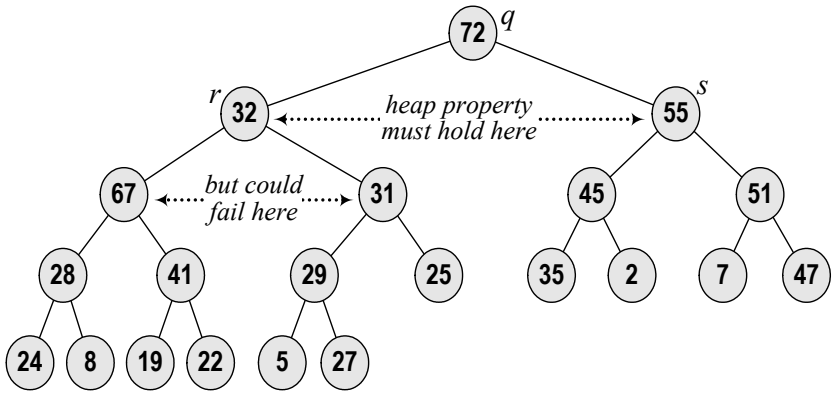


- The element to be removed (77) is in the root. Removing it leaves the root empty.
- The only node we can delete from the tree, and still have a nearly complete tree, is the last node (node *p*).
- So we move the element in node *p* (32) to the root (node *q*), and remove node *p* from the tree.
- We still have a nearly complete binary tree, and the heap property can fail only at the children of the root (nodes *r* and *s*).



- Given a nearly complete binary tree, in which the heap property can fail only at the children of the root, we can make the tree into a heap using a procedure called *max-heapify()*.
- Among the root and its two children (nodes *q*, *r*, *s*), we find the largest element. (Two comparisons will suffice.)
  - In this case, the largest (72) occurs in node *r*.
- If the largest of these three elements were to occur in the root (not the case here), we would be done.
- If the largest occurs in a child of the root (as happens here, node *r*), we exchange the element in the root with the element in this child.

- In our case, we exchange 32 and 72.



- This guarantees that the heap property holds at both children of the root, but may cause it to fail at the children of the node exchanged with the root (the children of node *r*, in our case).
- We apply the same process recursively to the subtree rooted at *r*, i.e., invoke *max-heapify*() recursively.
- The recursion terminates when we reach a leaf node, if not sooner.
  - The maximum number of calls to *max-heapify*() is $(\textit{height of heap}) = \lfloor \lg(n) \rfloor$

    and the maximum number of comparisons is $2\lfloor \lg(n) \rfloor$.

- With the array representation, we can write *max-heapify*() like this.

```
// A is an array of size at least n, which we think of
//   as a nearly complete binary tree.  In the subtree
//   of A[1..n] rooted at A[i] , the heap property
//   holds everywhere except possibly at the children
//   of A[i].  This function makes the subtree of
//    A[1..n] rooted at A[i] into a heap.
max-heapify( A, i, n)
      largest = i;
      if ( 2i ≤ n and A[2i] > A[i] )
            largest = 2i;
      if ( 2i+1 ≤ n and A[2i+1] > A[largest] )
            largest = 2i+1;
      if ( largest ≠ i )
            swap( A[i], A[largest] )
            max-heapify( A, largest, n);
```

- We can also write *max-heapify()* non-recursively like this:

```
max-heapify( A, i, n)
    while ( 2i ≤ n )
        largest = i;
        if (A[2i] > A[i] )
            largest = 2i;
        if ( 2i+1 ≤ n and A[2i+1] > A[largest] )
            largest = 2i+1;
        if ( largest ≠ i )
            swap( A[i], A[largest] )
            i = largest;
        else
            i = n+1;
```

Note: In addition to operations (1), (2), (3), we can perform several other operations efficiently ($\Theta(\lg(n))$ time).

- Increase or decrease the element in a known position.
- Remove the element in a known position.

However, we can ***not*** efficiently

- Given x, decide if the heap contains an element equal to x.
- Given k, find the k^{th} largest element in the heap (unless k is 1, or at least is very close to 1).
- Given x, remove x from the heap, if it is present.