Heap Sort

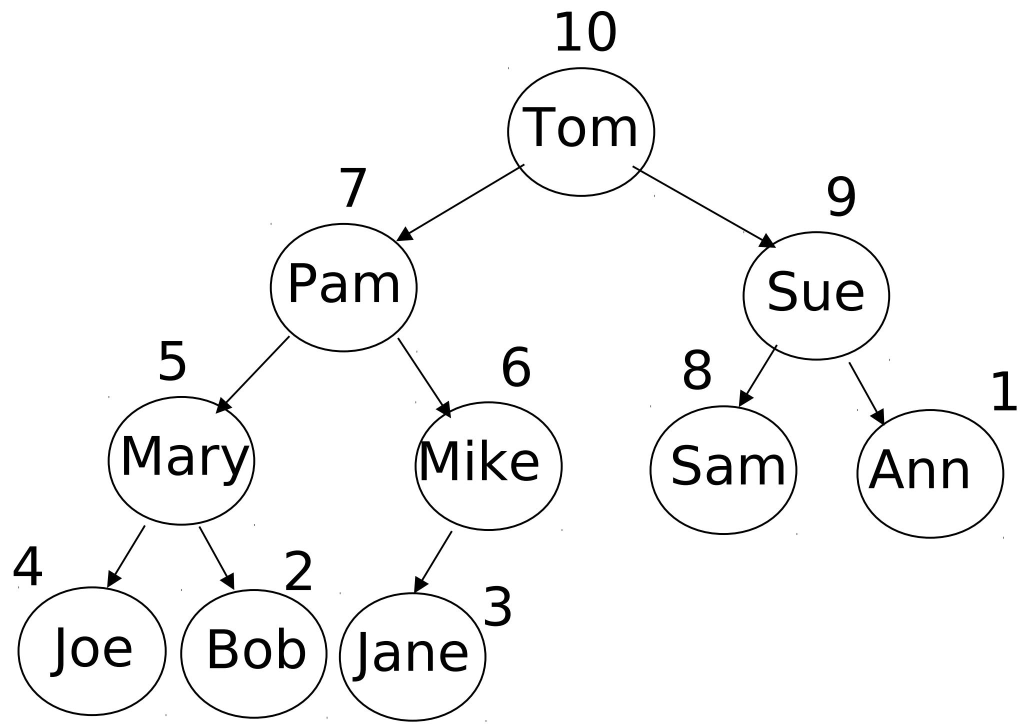
Heaps

A ***heap*** is a ***complete*** *binary tree* that is either:

- *empty*, or
- consists of a root and two subtrees, such that
  - both subtrees are *heaps*, and
  - the root contains a search key that is $\geq$ the search key of each of its children.

Array-Based Representation of a *Heap*

Search Key	Ite
10	To
7	Pa
9	Su
5	Ma
6	Mi
8	Sa
1	An
4	Joe
2	Bo
3	Jan



Array-Based Representation of a *Heap*

- Note that, for any node, the search key of its *left child* is not necessarily ≤ or ≥ the search key of its *right child*.

- The only constraint is that any *parent* node must have a search key that is ≥ the search key of *both of its children*.

- Note that this is sufficient to ensure that the item with the *greatest* search key in the heap is stored at the *root*.

The ADT Priority Queue

- A ***priority queue*** is an ADT in which items are ordered by a priority value. The item with the *highest priority* is always the *next* to be removed from the queue. (Highest Priority In, First Out: *HPIFO*)

- Supported operations include:
  - *Create* an empty priority queue
  - *Destroy* a priority queue
  - Determine whether a priority queue *is empty*
  - *Insert* a new item into a priority queue
  - *Retrieve,* and then *delete* from the priority queue the item with the *highest priority* value

PriorityQ: *Retrieve & Delete*

- Consider the operation, "*Retrieve,* and then *delete* from the priority queue the item with the *highest priority* value."

- In a heap where search keys represent the *priority* of the items, the item with the highest priority is stored at the *root*.

- Consequently, *retrieving* the item with the highest priority value is trivial.

- However, if the root of a heap is *deleted* we will be left with two separate heaps.

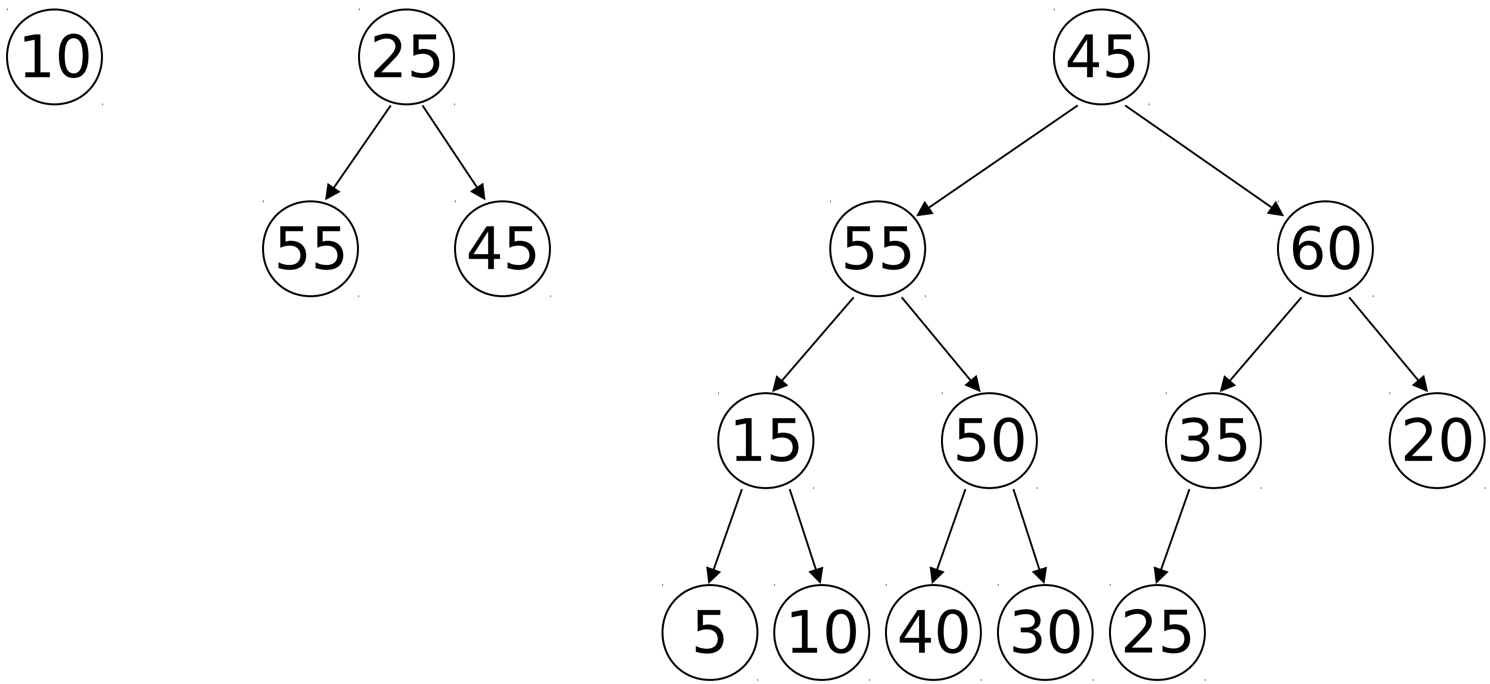
- We need a way to transform the remaining nodes back into a single heap.

PriorityQ: *Retrieve*

```
bool  PriorityQ::pqRetrieve( pq)
{
    if( pq IsEmpty( ) )  return  false;

    priorityItem  =  items[ 0 ];

    items[ 0 ]  =  items[ – – size ];

    heapRebuild( 0 );
    return  true;
}
```

Semiheap

A ***semiheap*** is a *complete* binary tree in which the root's left and right subtrees are both *heaps*.



Rebuilding a Heap: *Basic Idea*

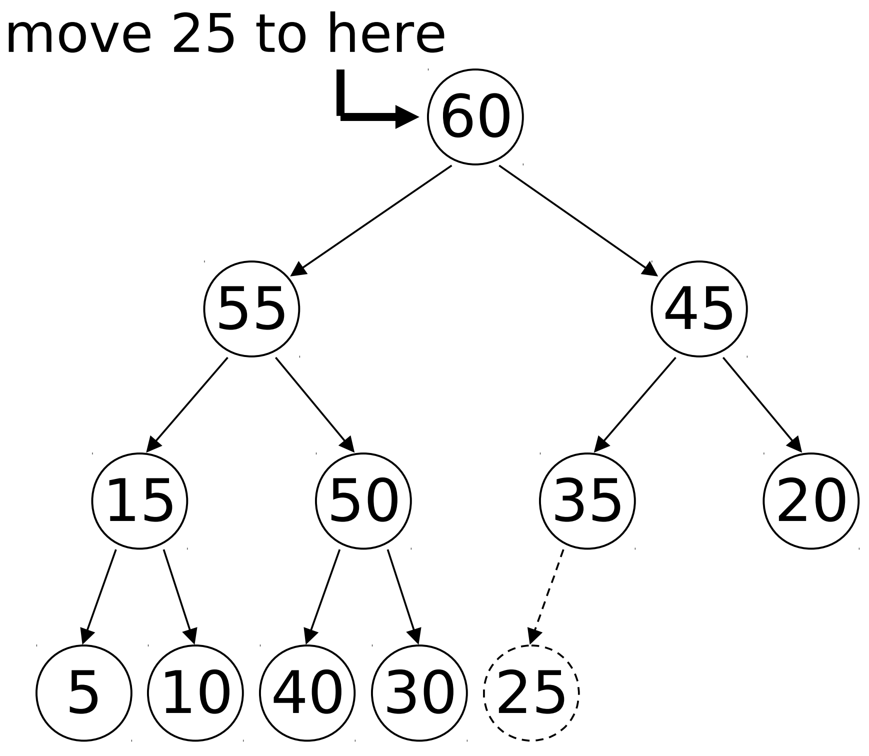
Problem: Transform a *semiheap* with given root into a *heap*.

Let *key(n)* represent the search key value of node *n*.

1) If the *root* of the *semiheap* is not a leaf, and key(*root*) < key(child of *root* with larger search key value) then swap the item in the root with the child containing the larger search key value.

2) If any items were swapped in step 1, then repeat step 1 with the subtree rooted at the node whose item was swapped with the root. If no items were swapped, then we are done: the resulting tree is a heap.

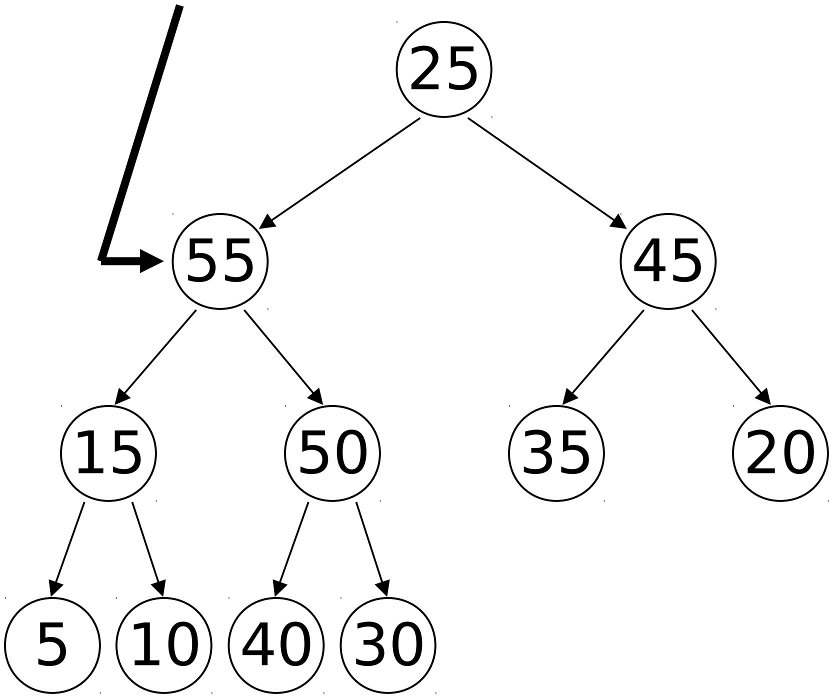
Retrieve & Delete: *Example*



- Retrieve the item with the *highest priority* value (= 60) from the *root*.
- Move the item from the *last* node in the heap (= 25) to the *root*, and delete the last node.

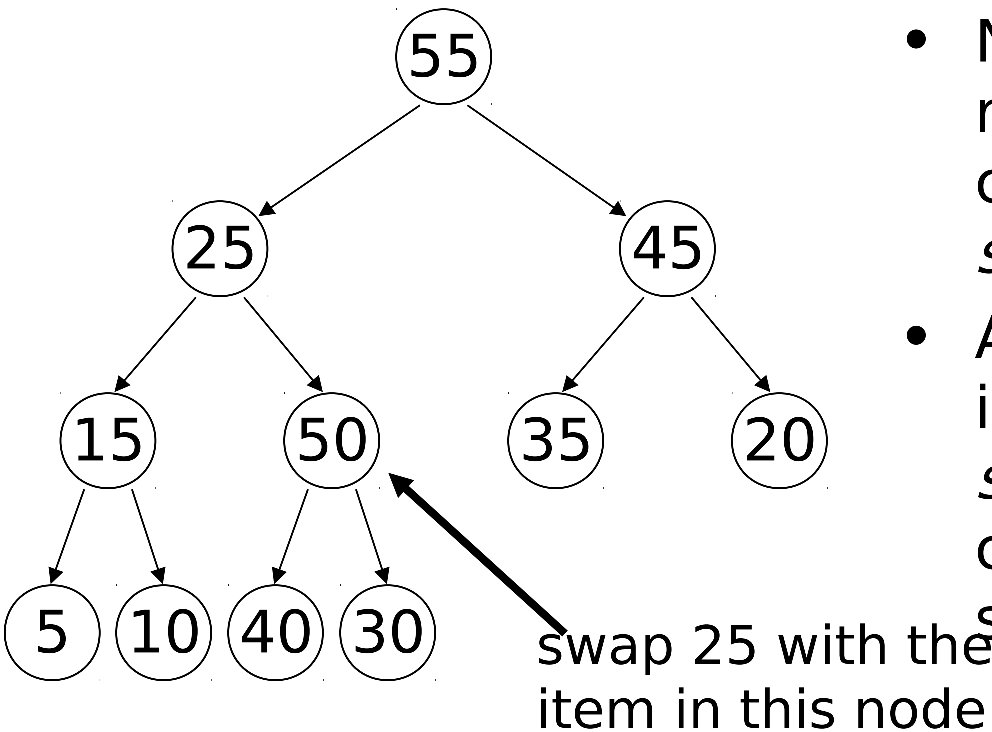
Rebuilding a Heap: *Example (Cont'd.)*

swap 25 with the item in this node



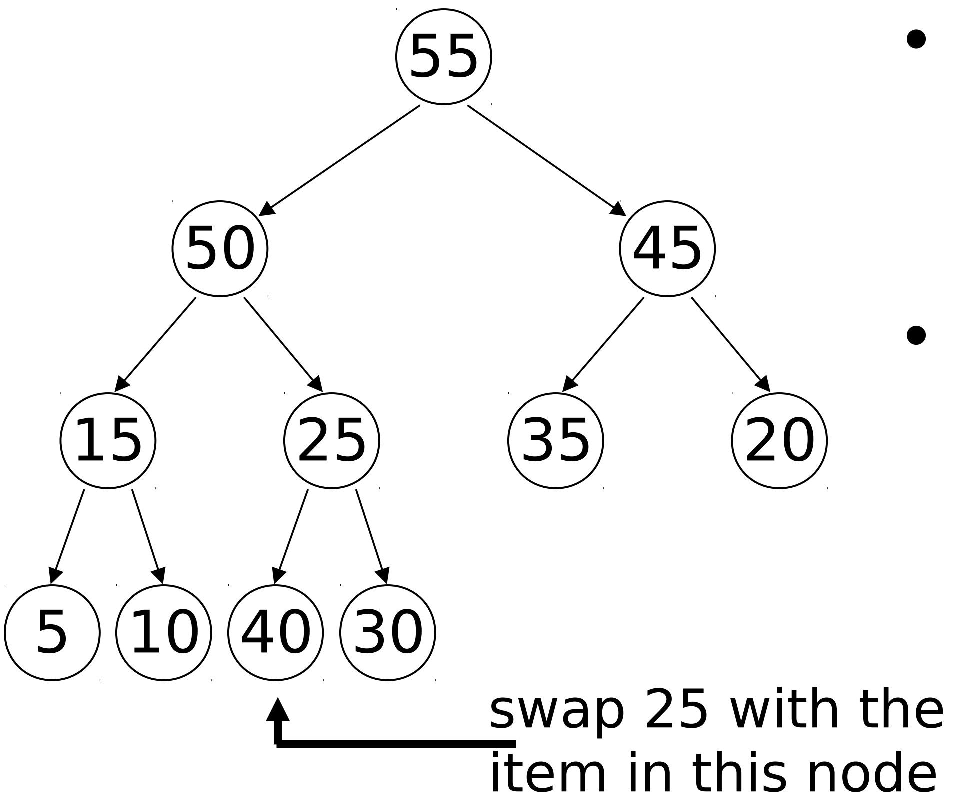
- The resulting data structure is a ***semiheap***, a *complete* binary tree in which the root's left and right subtrees are both *heaps*.
- To transform this *semiheap* into a *heap*, start by swapping the item in the root with its child containing the larger search key value.

Rebuilding a Heap: *Example (Cont’d.)*



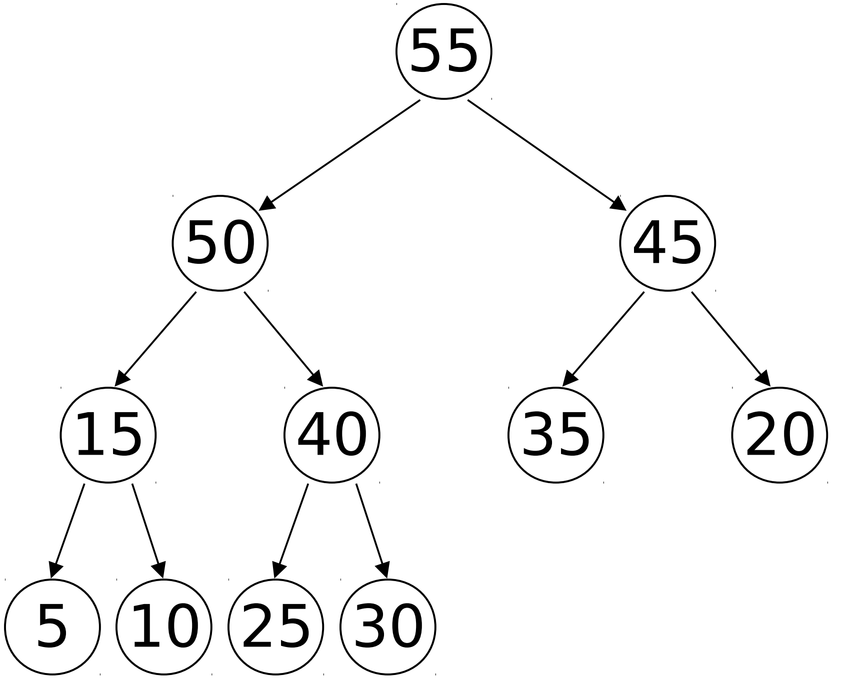
- Note that the subtree rooted at the node containing 25 is a *semiheap*.
- As before, swap the item in the root of this *semiheap* with its child containing the larger search key value.

Rebuilding a Heap: *Example (Cont'd.)*



- Note that the subtree rooted at the node containing 25 is a *semiheap*.
- As before, swap the item in the root of this *semiheap* with its child containing the larger search key value.

Rebuilding a Heap: *Example (Cont'd.)*



- Note that the subtree rooted at the node containing 25 is a *semiheap* with two empty subtrees.
- Since the root of this *semiheap* is also a *leaf*, we are done.
- The resulting tree rooted at the node containing 55 is a *heap*.

PriorityQ: *Private Member Function Definition*

```
void  PriorityQ::heapRebuild( int  root )
{
    int  child  =  2 * root + 1;
    if( child  <  size )
    {
      int  rightChild  =  child + 1;
      if( rightChild  <  size  && getKey( items[ rightChild ] )  >  getKey( items[ child ] ) )
                child = rightChild;    // child has the larger search key
            if( getKey( items[ root ] )  <  getKey( items[ child ] ) )
                {
                swap( items[ root ],  items[ child ] );
                heapRebuild( child );
                }
        }
}
```

PriorityQ **Insert**: *Basic Idea*

Problem: Insert a new item into a *priority queue*, where the priority queue is implemented as a *heap*.

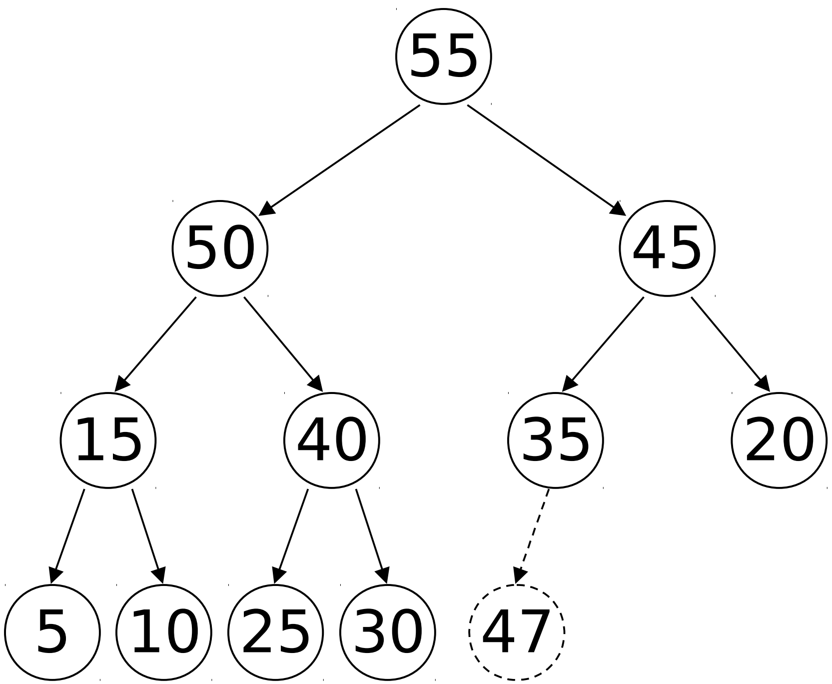
Let *key(n)* represent the search key value of node *n*.

1) Store the new item in a new node at the end of the heap.

2) If the node containing the new item has a parent, and key(node containing new item) > key(node's parent) then swap the new item with the item in its parent node.

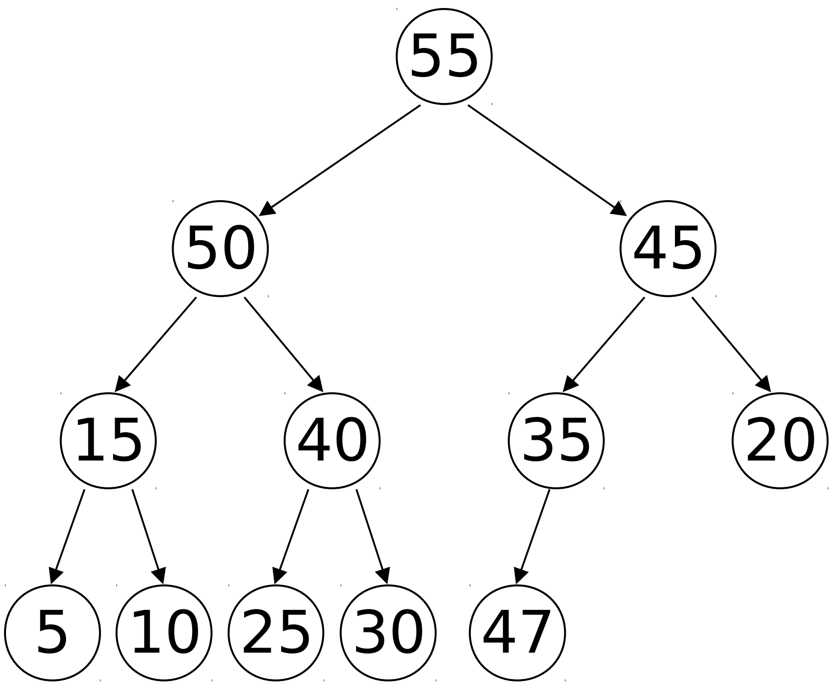
3) If the new item was swapped with its parent in step 2, then repeat step 2 with the new item in the parent node. If no items were swapped, then we are done: the resulting tree is a heap containing the new item.

PriorityQ Insert: *Example*



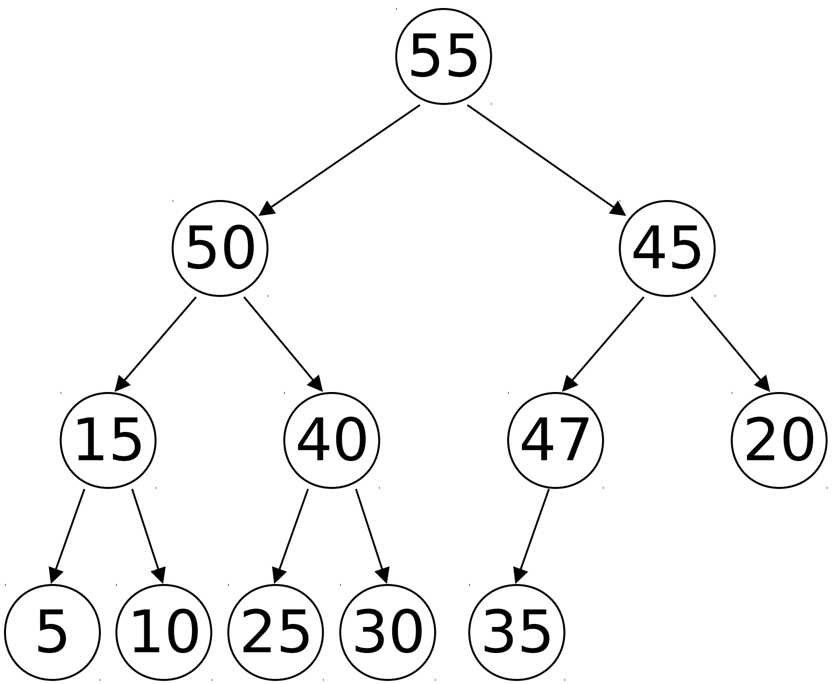
- Suppose that we wish to insert an item with search key = 47.
- First, we store the new item in a new node at the end of the heap.

PriorityQ Insert: *Example (Cont'd.)*



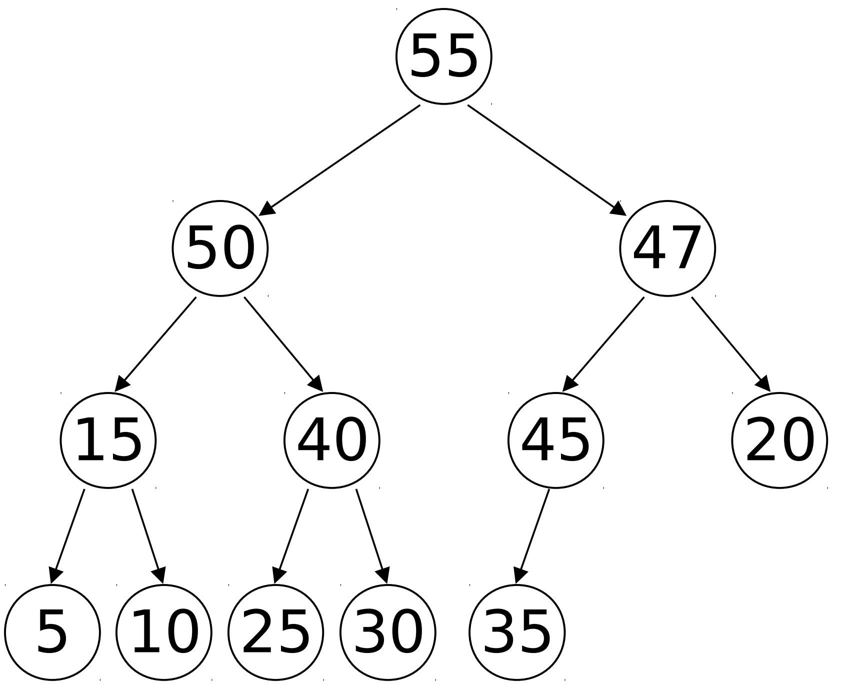
- Since the search key of the new item (= 47) > the search key of its parent (= 35), swap the new item with its parent.

PriorityQ Insert: *Example (Cont'd.)*



- Since the search key of the new item (= 47) > the search key of its parent (= 45), swap the new item with its parent.

PriorityQ Insert: *Example (Cont'd.)*



- Since the search key of the new item (= 47) $\leq$ the search key of its parent (= 55), we are done.
- The resulting tree is a *heap* containing the new item.

PriorityQ: *Public Member Function Definition*

```
bool  PriorityQ::pqInsert( const  PQItemType  &newItem )
    {
    if( size  >  MaxItems )  return  false;
    items[ size ] = newItem;
    int  newPos = size,  parent = (newPos – 1) / 2;
    while( parent >= 0  &&
       getKey( items[ newPos ] )  >  getKey( items[ parent ] ) )
         {
         swap( items[ newPos ],  items[ parent ] );
         newPos = parent;
         parent = (newPos – 1) / 2;
         }
    size++;    return  true;
    }
```

Heap-Based PriorityQ: *Efficiency*

- In the *best case*, no swaps are needed after an item is inserted at the end of the heap. In this case, ***insertion*** requires constant time, which is O(1).
- In the *worst case*, an item inserted at the end of a heap will be swapped until it reaches the root, requiring O(height of tree) swaps. Since heaps are *complete binary trees*, and hence, *balanced*, the height of a heap with n nodes is $\lceil \log_2 (n + 1) \rceil$. Therefore, in this case, ***insertion*** is O(log n).
- In the *average case*, the inserted item will travel halfway to the root, which makes ***insertion*** in this case also O(log n).
- The "***retrieve & delete***" operation spends most of its time rebuilding a heap. A similar analysis shows that this is O(log n) in the *best*, *average*, and *worst cases*.

Heapsort: *Basic Idea*

Problem: Arrange an array of items into sorted order.

1) Transform the array of items into a *heap*.

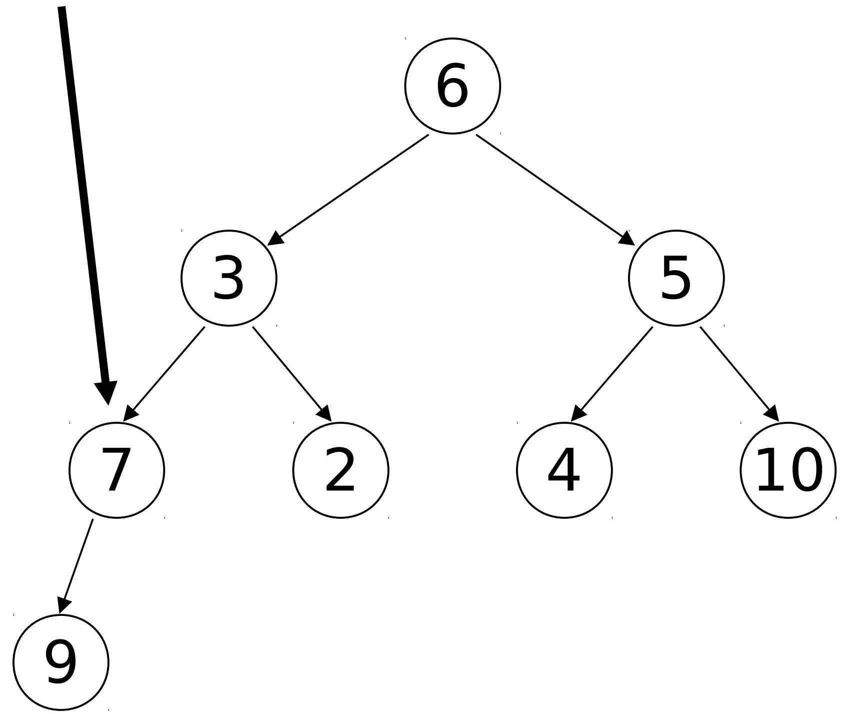
2) Invoke the "*retrieve & delete*" operation repeatedly, to extract the largest item remaining in the heap, until the heap is empty. Store each item retrieved from the heap into the array from back to front.

Note: We will refer to the version of *heapRebuild* used by *Heapsort* as *rebuildHeap*, to distinguish it from the version implemented for the class *PriorityQ*.

Transform an Array Into a Heap: *Example*

6	3	5	7	2	4	10	9
0	1	2	3	4	5	6	7

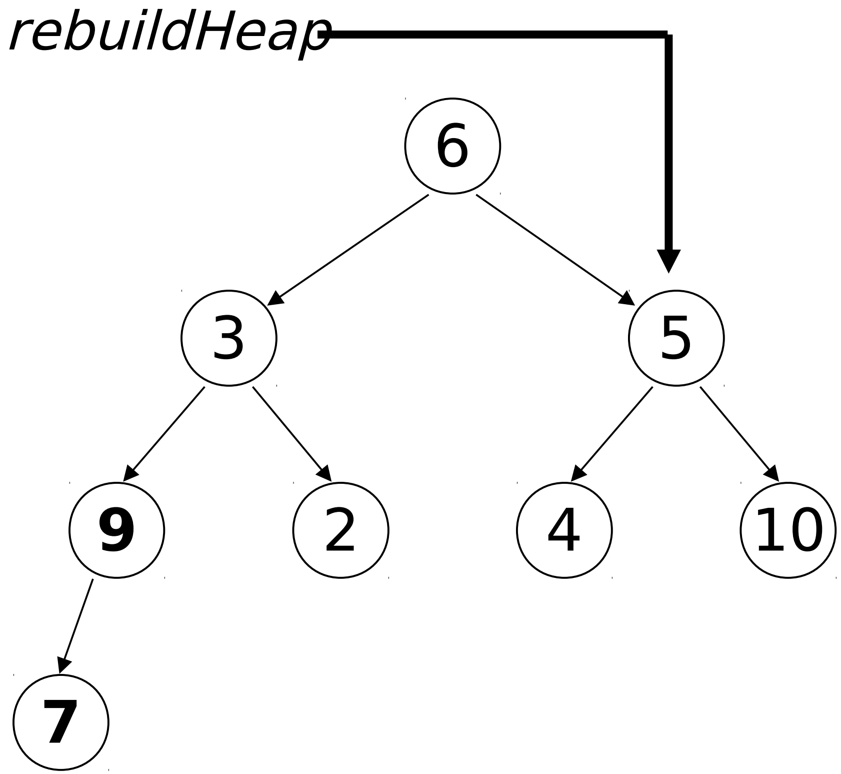
rebuildHeap



- The items in the array, above, can be considered to be stored in the complete binary tree shown at right.
- Note that leaves 2, 4, 9 & 10 are *heaps;* nodes 5 & 7 are roots of *semiheaps*.
- *rebuildHeap* is invoked on the *parent* of the last node in the array (= 9).

Transform an Array Into a Heap: *Example*

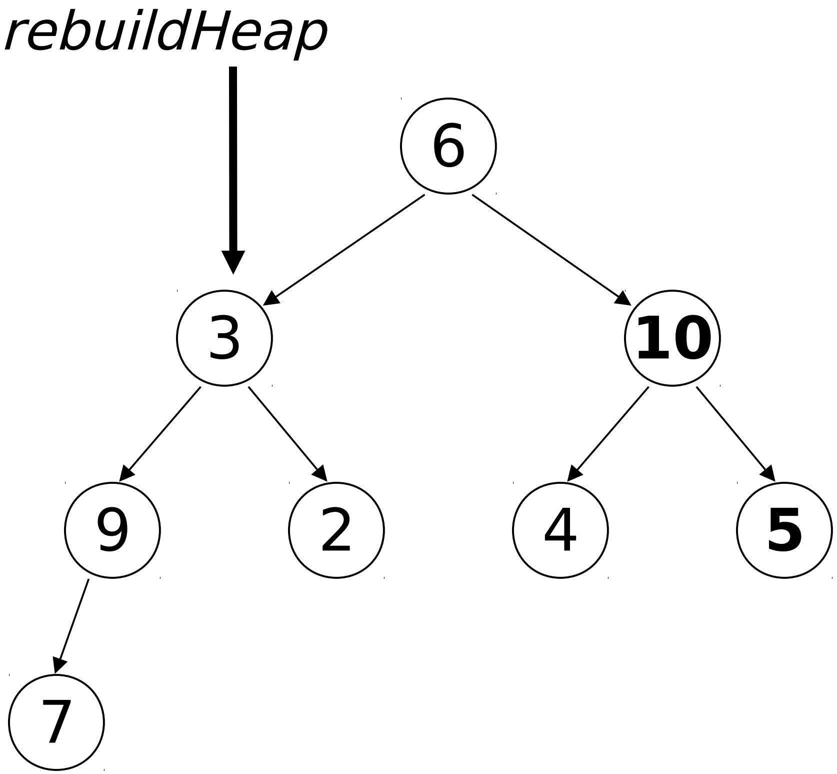
6	3	5	**9**	2	4	10	**7**
0	1	2	3	4	5	6	7



- Note that nodes 2, 4, 7, 9 & 10 are roots of *heaps;* nodes 3 & 5 are roots of *semiheaps*.
- *rebuildHeap* is invoked on the node in the array *preceding* node 9.

Transform an Array Into a Heap: *Example*

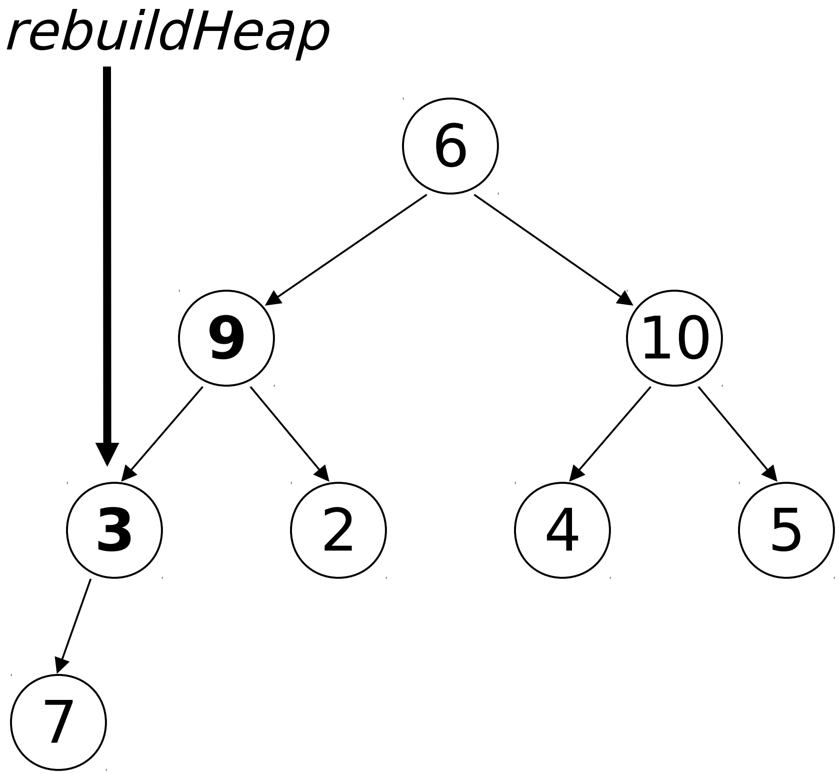
6	3	**10**	9	2	4	**5**	7
0	1	2	3	4	5	6	7



- Note that nodes 2, 4, 5, 7, 9 & 10 are roots of *heaps;* node 3 is the root of a *semiheap*.
- *rebuildHeap* is invoked on the node in the array *preceding* node 10.

Transform an Array Into a Heap: *Example*

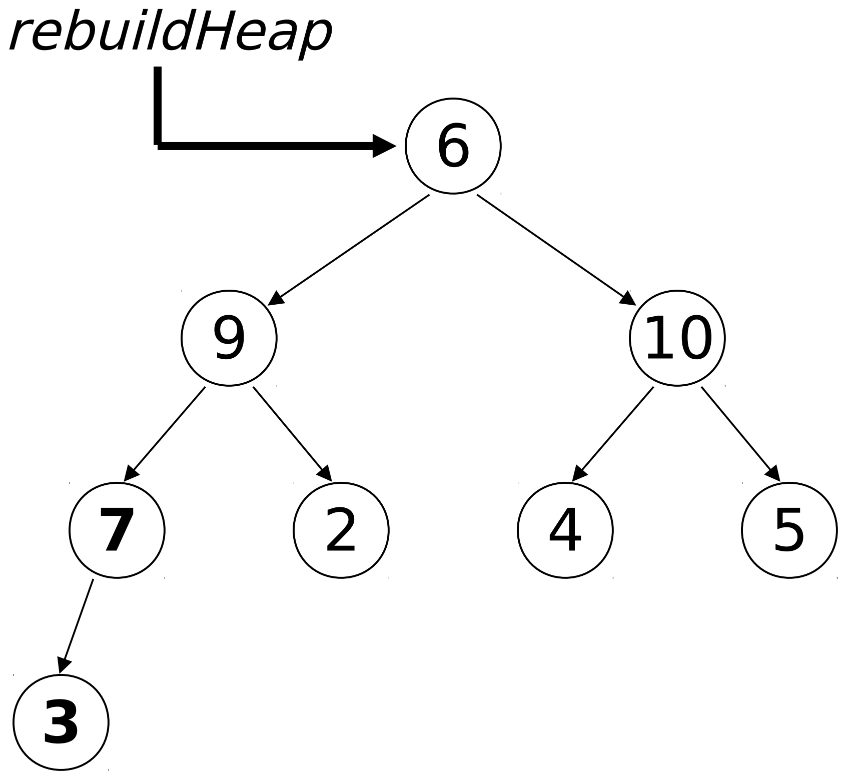
6	**9**	10	**3**	2	4	5	7
0	1	2	3	4	5	6	7



- Note that nodes 2, 4, 5, 7 & 10 are roots of *heaps;* node 3 is the root of a *semiheap*.
- *rebuildHeap* is invoked recursively on node 3 to complete the transformation of the *semiheap* rooted at 9 into a *heap*.

Transform an Array Into a Heap: *Example*

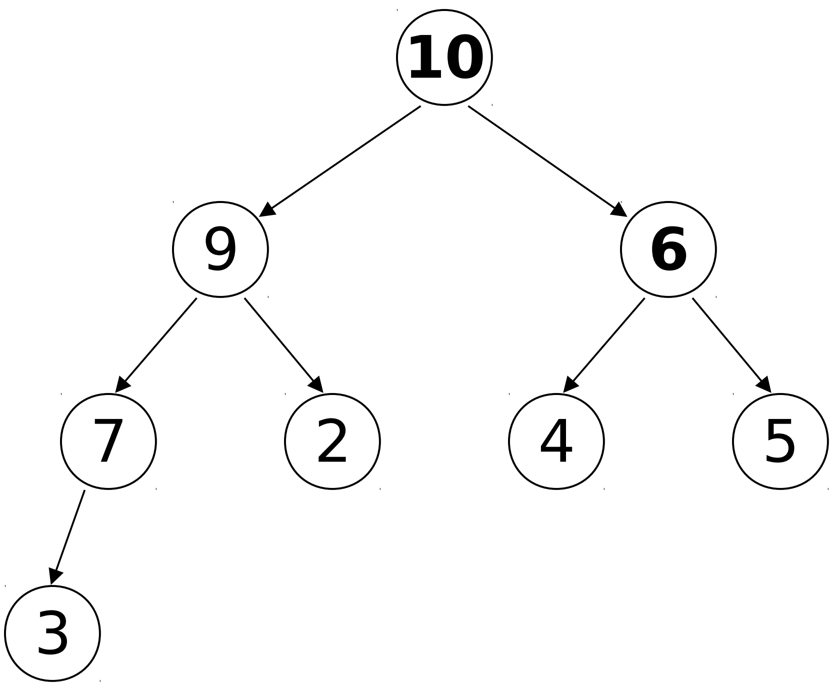
6	9	10	**7**	2	4	5	**3**
0	1	2	3	4	5	6	7



- Note that nodes 2, 3, 4, 5, 7, 9 & 10 are roots of *heaps;* node 6 is the root of a *semiheap*.
- The recursive call to *rebuildHeap* returns to node 9.
- *rebuildHeap* is invoked on the node in the array *preceding* node 9.

Transform an Array Into a Heap: *Example*

10	9	**6**	7	2	4	5	3
0	1	2	3	4	5	6	7



- Note that node 10 is now the root of a *heap*.
- The transformation of the *array* into a *heap* is complete.

Transform an Array Into a Heap (Cont'd.)

- Transforming an array into a heap begins by invoking *rebuildHeap* on the *parent of the last node* in the array.
- Recall that in an array-based representation of a complete binary tree, the *parent* of any node at array position, *i*, is

$\lfloor (i - 1) / 2 \rfloor$

- Since the last node in the array is at position *n – 1*, it follows that transforming an array into a heap begins with the node at position

$\lfloor (n - 2) / 2 \rfloor = \lfloor n / 2 \rfloor - 1$

and continues with each preceding node in the array.

Transform an Array Into a Heap: *C++*

```
for( int  root  =  n/2 – 1;   root  >=  0;   root – – )
   {
   rebuildHeap( a,  root,  n );
   }
```

Transform a Heap Into a Sorted Array: *Basic Idea*

Problem: Transform array a[ ] from a heap of *n* items into a sequence of *n* items in sorted order.

Let *last* represent the position of the last node in the heap. Initially, the heap is in a[0 .. *last*], where *last* = *n* – 1.

1) Move the largest item in the heap to the beginning of an (initially empty) sorted region of a[ ] by swapping a[0] with a[*last*].
2) Decrement *last*. a[0] now represents the root of a semiheap in a[0 .. *last*], and the sorted region is in a[*last* + 1 .. *n* – 1].
3) Invoke *rebuildHeap* on the semiheap rooted at a[0] to transform the semiheap into a heap.
4) Repeat steps 1 - 3 until *last* = -1. When done, the items in array a[ ] will be arranged in sorted order.

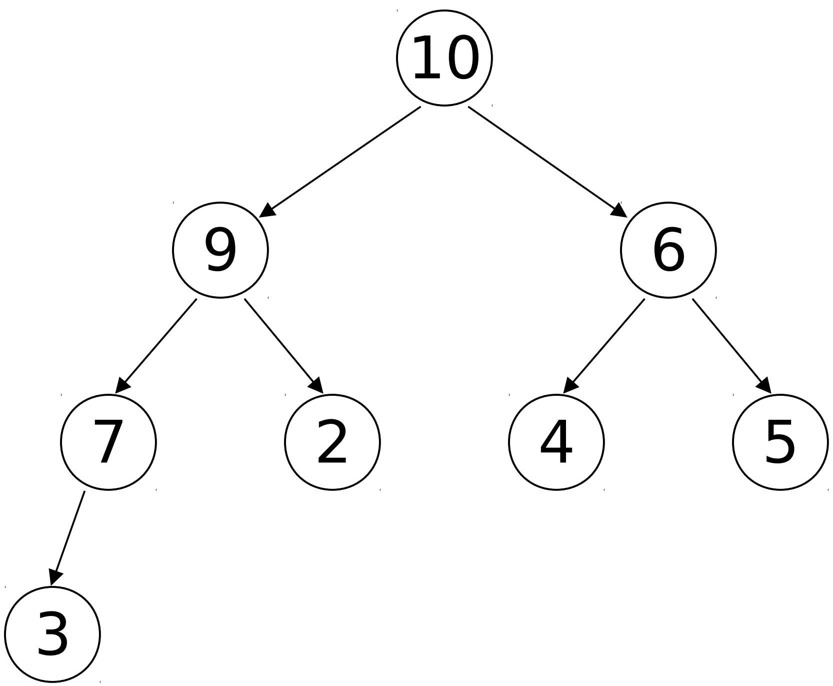
Heapsort: *C++*

```
void  heapsort( ItemType  a[ ],  int  n )
    {
     for( int  root  =  n/2 – 1;   root  >=  0;   root – – )
           rebuildHeap( a,  root,  n );
     for( int  last = n – 1;  last  >   0;   )
          {
          swap( a[0],  a[ last ] );    last – – ;
          rebuildHeap( a,  0,  last );
          }
    }
```

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	10	9	6	7	2	4	5	3

Heap

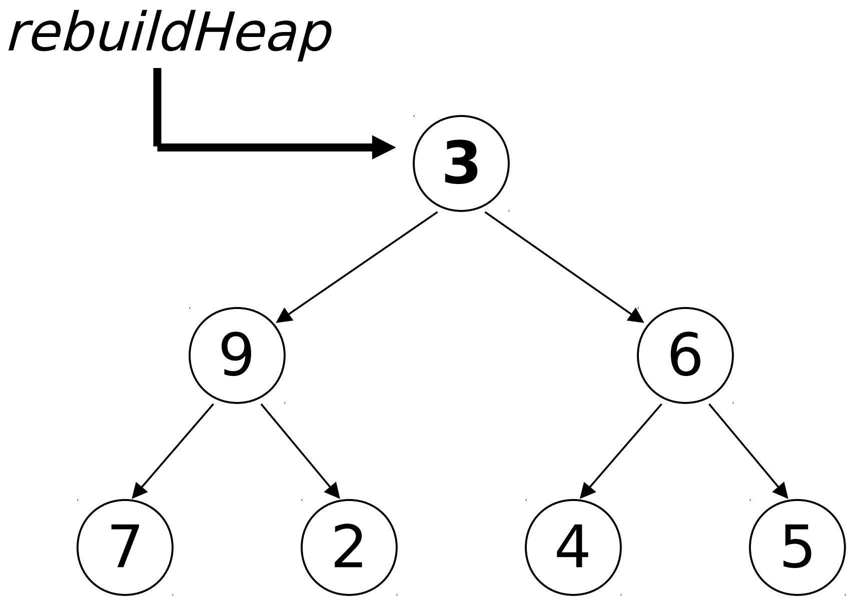


- We start with the heap that we formed from an unsorted array.
- The heap is in a[0..7] and the sorted region is empty.
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[7].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**3**	9	6	7	2	4	5	**10**

Semiheap (a[0..6]) | Sorted (a[7])

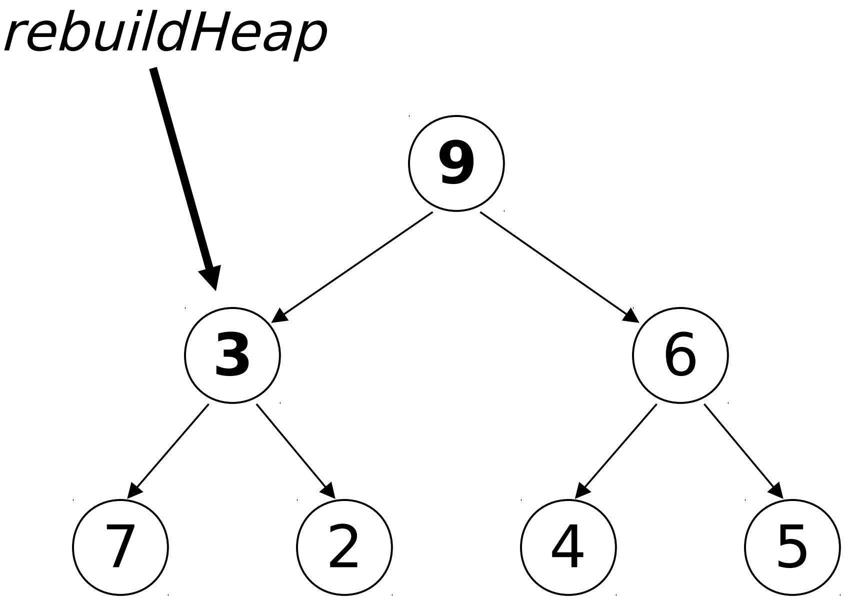


- a[0..6] now represents a semiheap.
- a[7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**9**	**3**	6	7	2	4	5	10

Becoming a Heap | Sorted

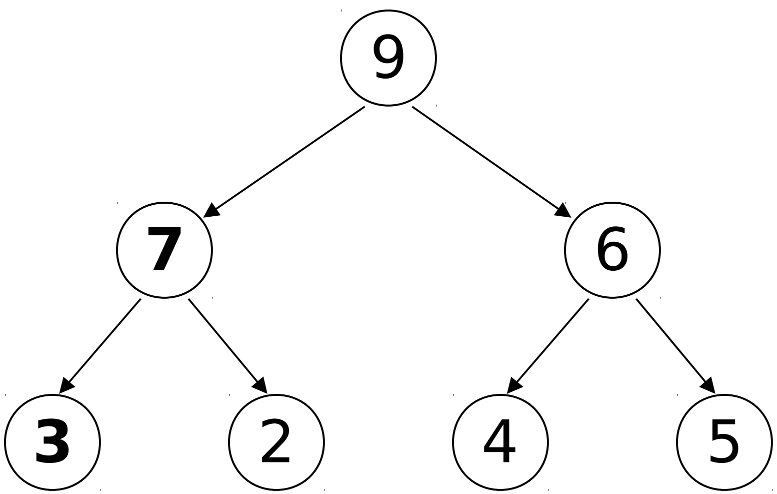


- *rebuildHeap* is invoked recursively on a[1] to complete the transformation of the semiheap rooted at a[0] into a heap.

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	9	**7**	6	**3**	2	4	5	10

Heap | Sorted

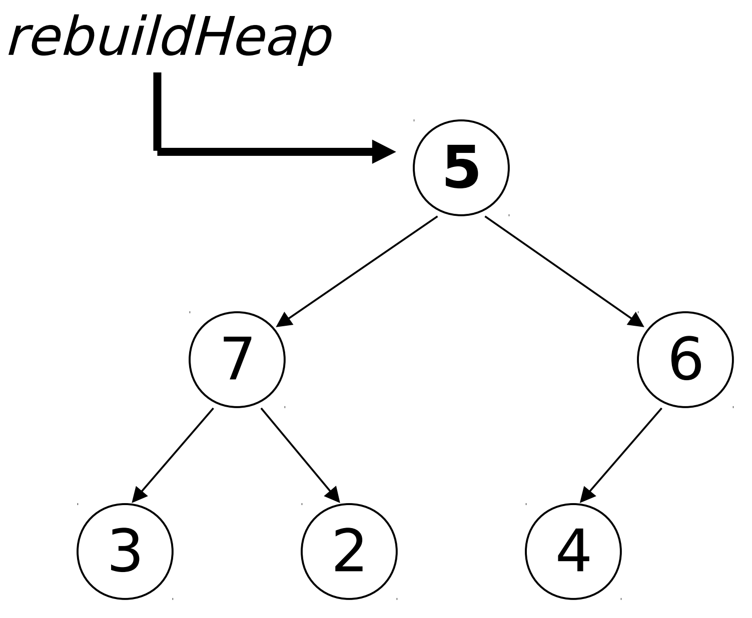


- a[0] is now the root of a heap in a[0..6].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[6].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**5**	7	6	3	2	4	**9**	10

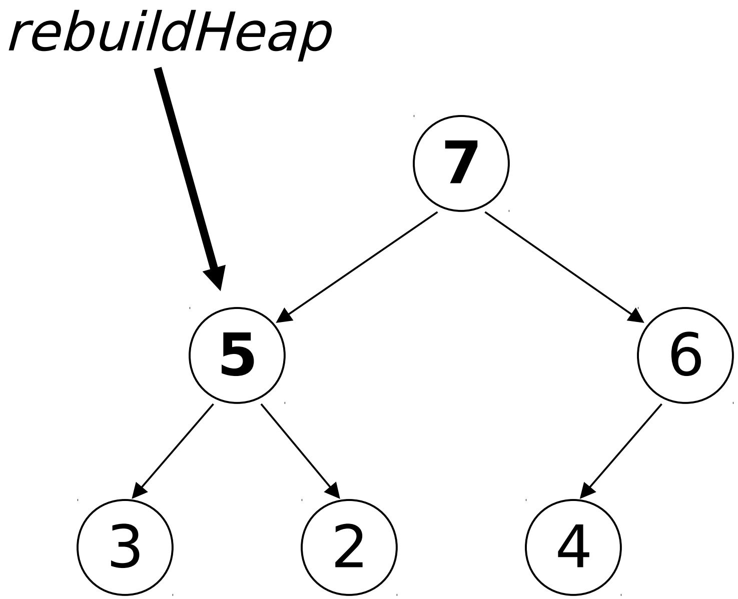
Semiheap (a[0..5]) — Sorted (a[6..7])



- a[0..5] now represents a semiheap.
- a[6..7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**7**	**5**	6	3	2	4	9	10
	Heap						Sorted	

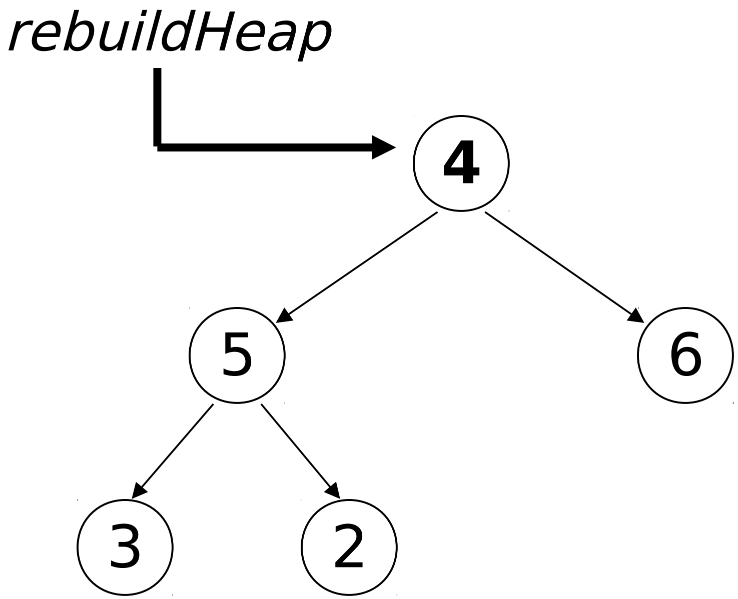


- Since a[1] is the root of a heap, a recursive call to *rebuildHeap* does nothing.
- a[0] is now the root of a heap in a[0..5].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[5].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**4**	5	6	3	2	**7**	9	10

Semiheap — Sorted

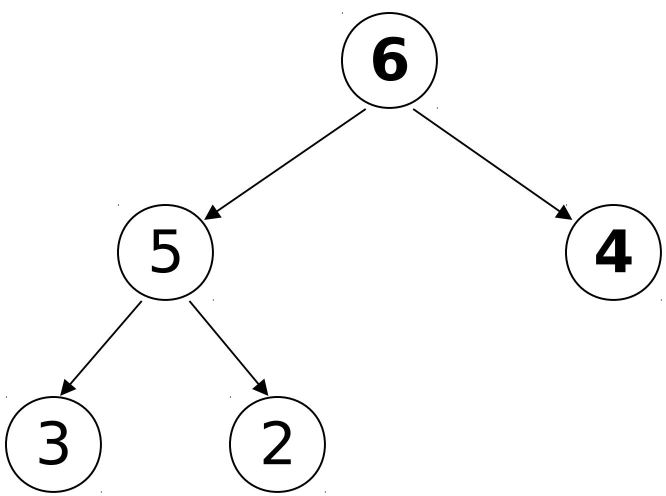


- a[0..4] now represents a semiheap.
- a[5..7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**6**	5	**4**	3	2	7	9	10

Heap (indices 0–4) | Sorted (indices 5–7)

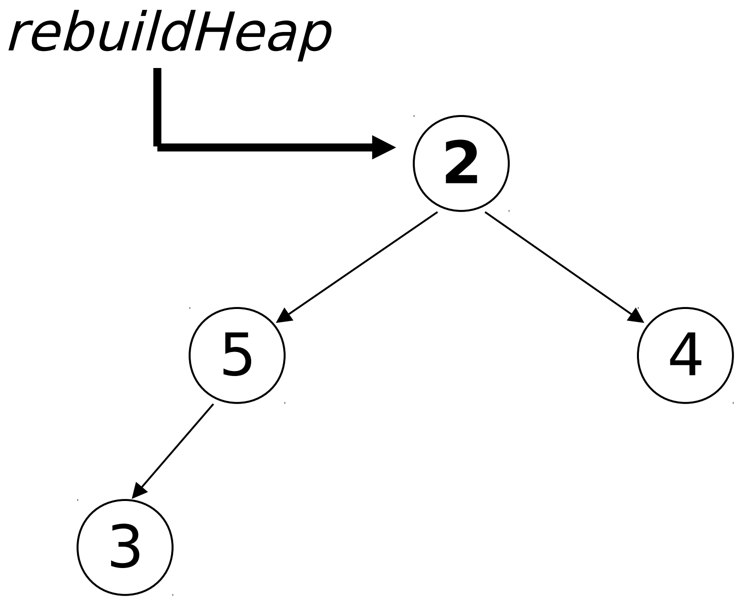


- a[0] is now the root of a heap in a[0..4].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[4].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**2**	5	4	3	**6**	7	9	10

Semiheap (a[0..3]) | Sorted (a[4..7])

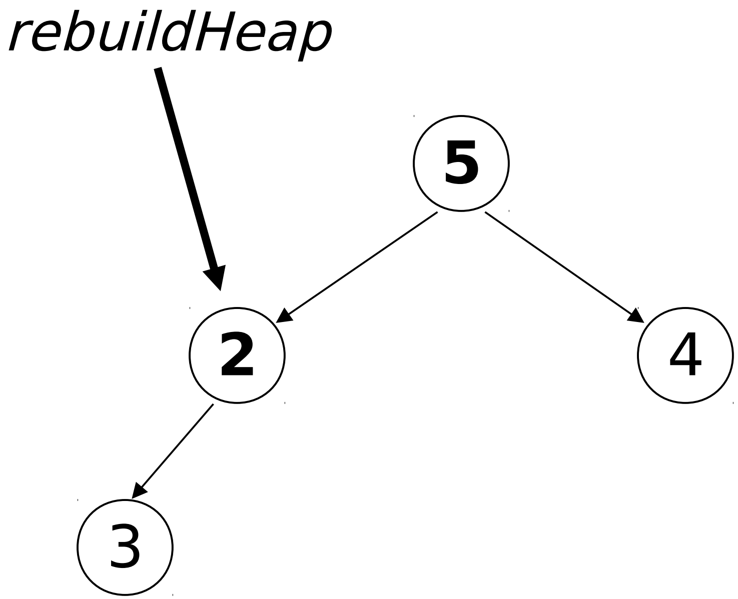


- a[0..3] now represents a semiheap.
- a[4..7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**5**	**2**	4	3	6	7	9	10

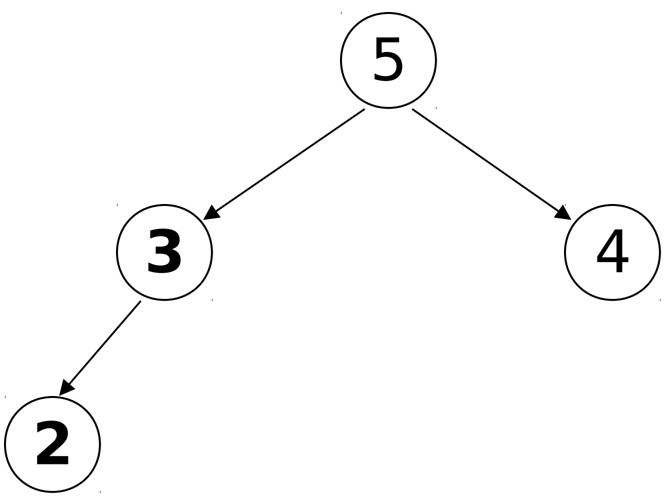
Becoming a Heap | Sorted



- *rebuildHeap* is invoked recursively on a[1] to complete the transformation of the semiheap rooted at a[0] into a heap.

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	5	**3**	4	**2**	6	7	9	10
	Heap				Sorted			

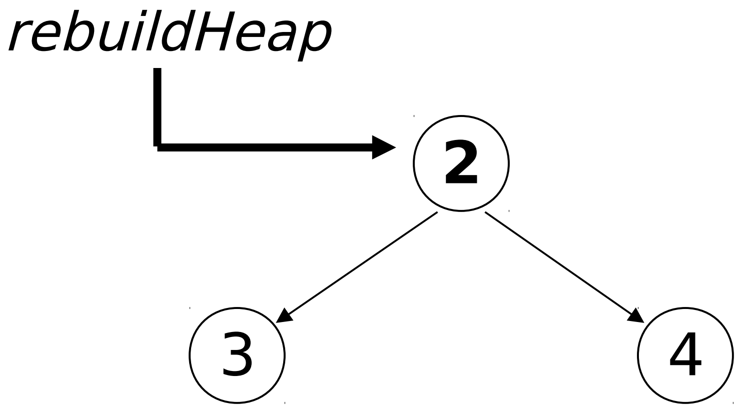


- a[0] is now the root of a heap in a[0..3].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[3].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**2**	3	4	**5**	6	7	9	10

Semiheap (a[0..2]) | Sorted (a[3..7])

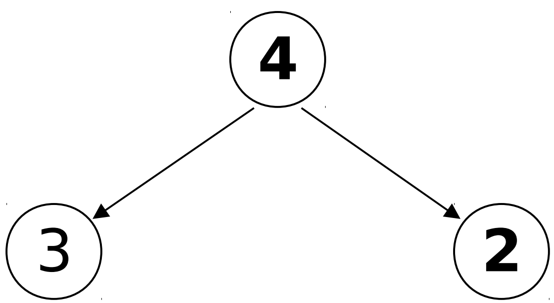


- a[0..2] now represents a semiheap.
- a[3..7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**4**	3	**2**	5	6	7	9	10

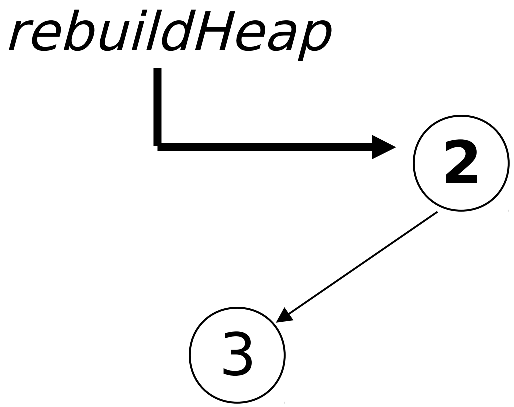
Heap (indices 0–2) | Sorted (indices 3–7)



- a[0] is now the root of a heap in a[0..2].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[2].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**2**	3	**4**	5	6	7	9	10
	Semiheap		Sorted					

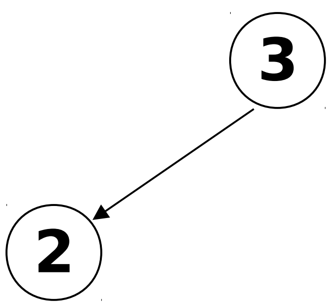


- a[0..1] now represents a semiheap.
- a[2..7] is the sorted region.
- Invoke *rebuildHeap* on the semiheap rooted at a[0].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**3**	**2**	4	5	6	7	9	10

Heap | Sorted



- a[0] is now the root of a heap in a[0..1].
- We move the largest item in the heap to the beginning of the sorted region by swapping a[0] with a[1].

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**2**	**3**	4	5	6	7	9	10

Heap | Sorted

- a[1..7] is the sorted region.
- Since a[0] is a heap, a recursive call to *rebuildHeap* does nothing.
- We move the only item in the heap to the beginning of the sorted region.

Transform a Heap Into a Sorted Array: *Example*

	0	1	2	3	4	5	6	7
a[ ]:	**2**	3	4	5	6	7	9	10

Sorted

- Since the sorted region contains all the items in the array, we are done.

Thank You