Chapter 9

Heaps, Priority Queues, and Heap Sort

Priority Queue

Queue

- Enqueue an item
- Dequeue: Item returned has been in the queue the longest amount of time

Priority Queue

- Enqueue a pair <item, priority>
- Dequeue: Item returned has highest priority

Priority Queue: application layer

- A priority queue is an ADT with the property that only the highest-priority element can be accessed at any time.
- Server systems use priority queue to manage jobs/requests
  - priority: can be based upon users importance, or based upon deadline, …
- Some graph algorithms: Dijkstra algorithm, Spanning Tree algorithm use it too.

ADT Priority Queue Operations

Transformers

- MakeEmpty
- Enqueue
- Dequeue

change state

Observers

- IsEmpty
- IsFull

observe state

Implementation Level

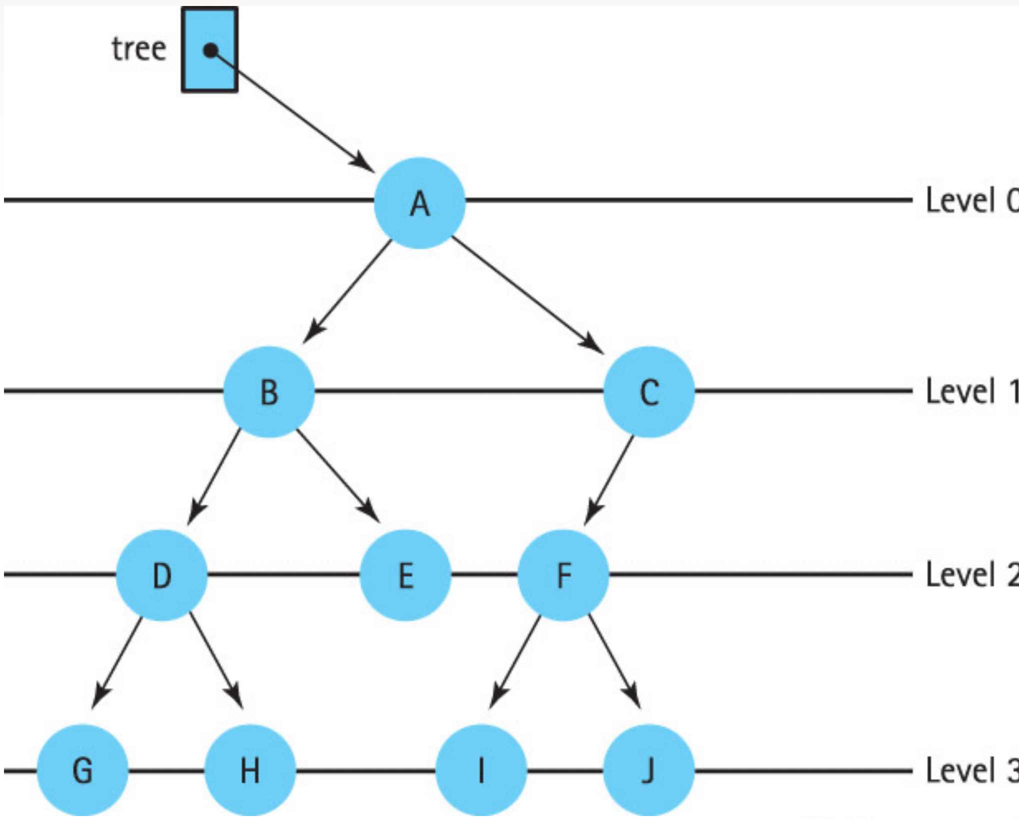
- There are many ways to implement a priority queue
  - An unsorted List-
    - dequeue: requires searching through entire list, O(N)
    - enqueue: constant time O(1)
  - An Array-Based Sorted List
    - Enqueue: O(N)
    - dequeue: constant time O(1)
  - A linked structure based Sorted List
    - enqueue: O(N)
    - dequeue: constant time O(1)
- N: The number of elements in the queue

Implementation Level (cont.)

- There are many ways to implement a priority queue
  - A Binary Search Tree-
    - enqueue?
    - dequeue?
  - A Heap:
    - enqueue and dequeue: both $O(\log_2 N)$ steps ,
    - even in the worst case!

A full tree: a binary tree in which each node has 0 or two children.

A complete binary tree: a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Filled? yes

filled? yes, at most 2 nodes at level 1

filled? no, maximally 4 nodes, only 3.

Full and Complete Trees (cont.)

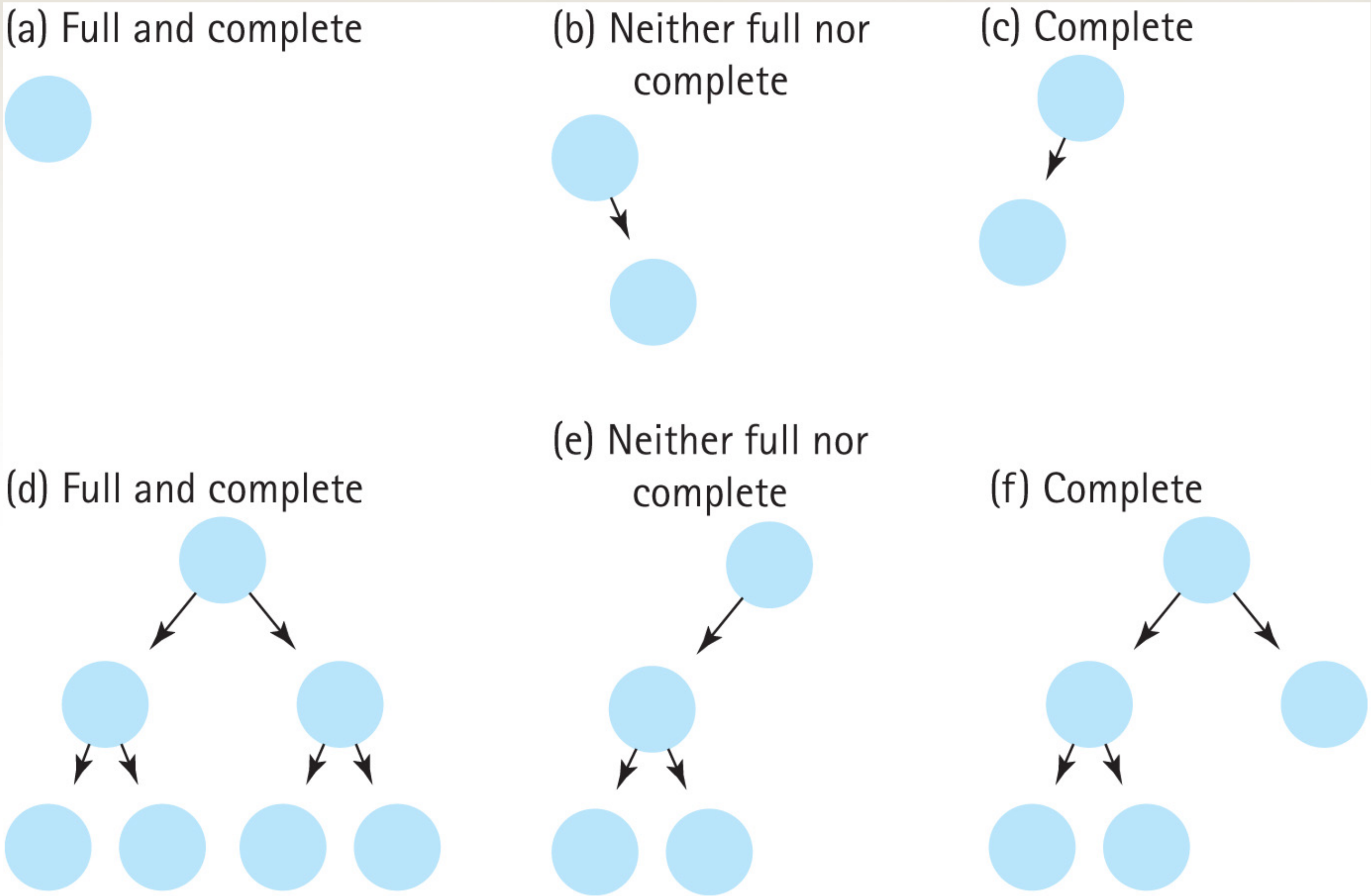


Figure 9.3 Examples of binary trees (a) Full and complete (b) Neither full nor complete (c) Complete (d) Full and complete (e) Neither full nor complete (f) Complete

A complete binary tree: a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Can you draw a complete binary tree with 5 nodes?

with 10 nodes?

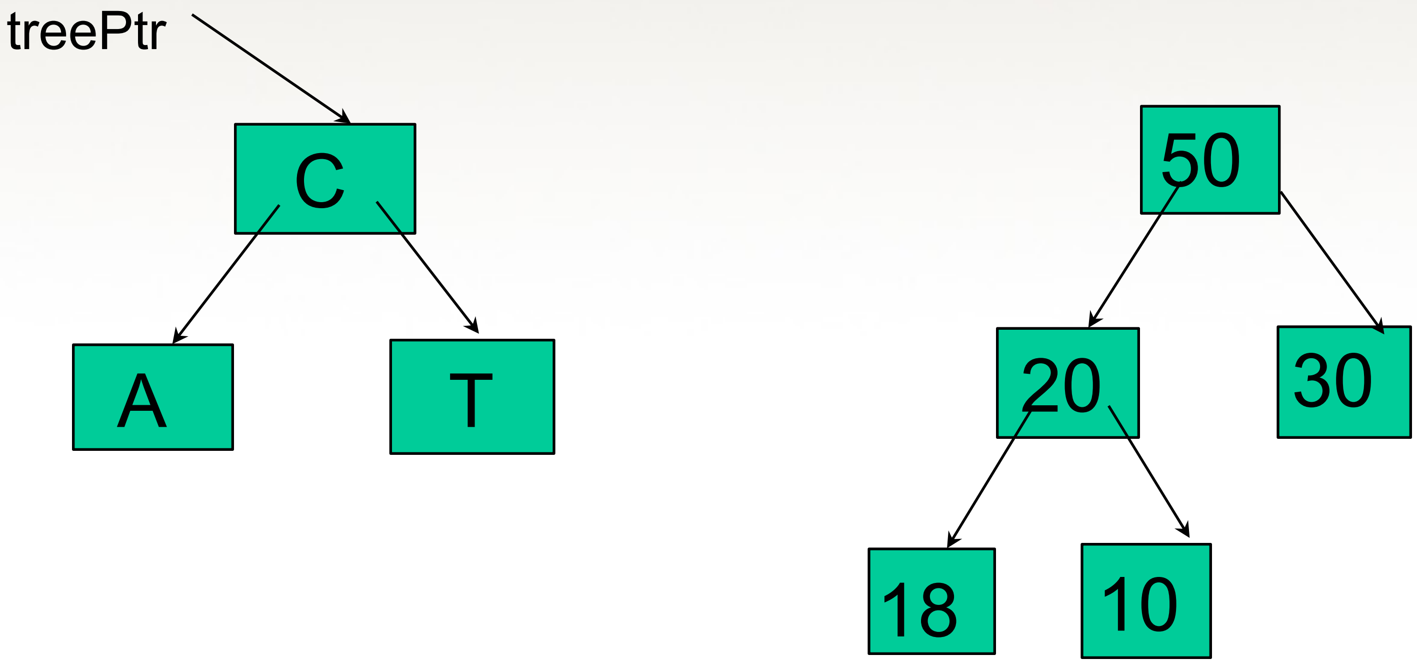
Note that the shape of the tree is completely decided!

What is a Heap?

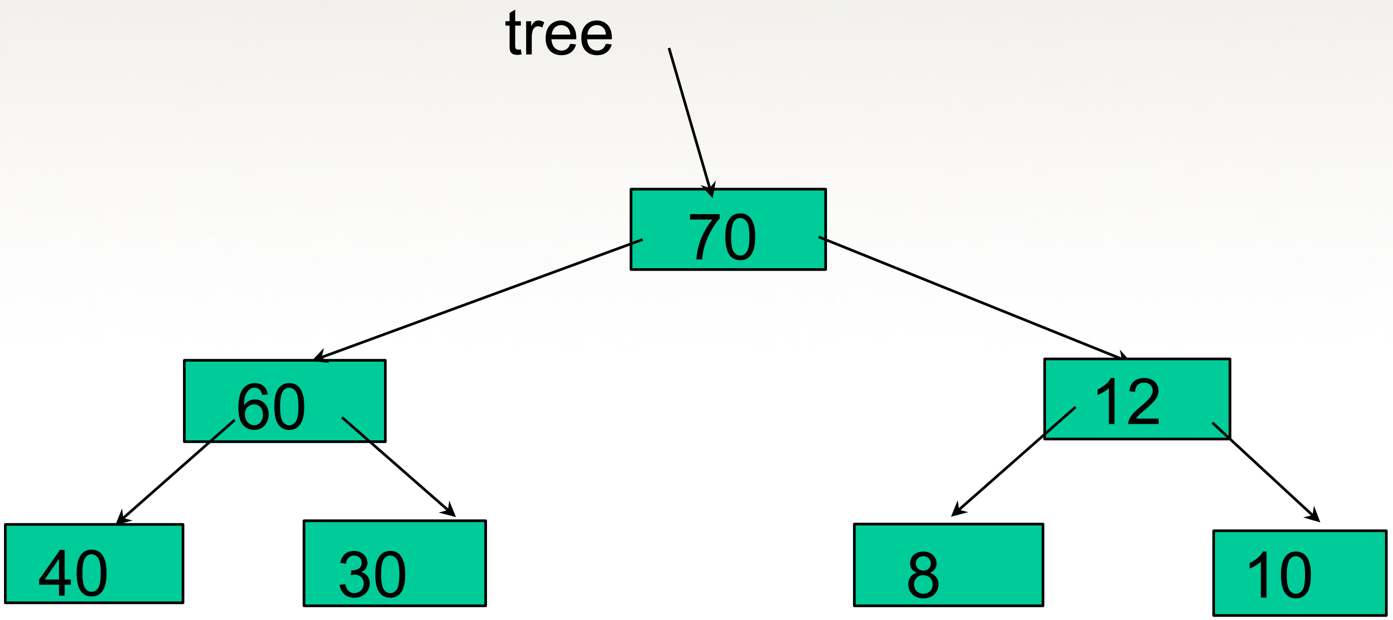
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- Its shape must be a complete binary tree.
- For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

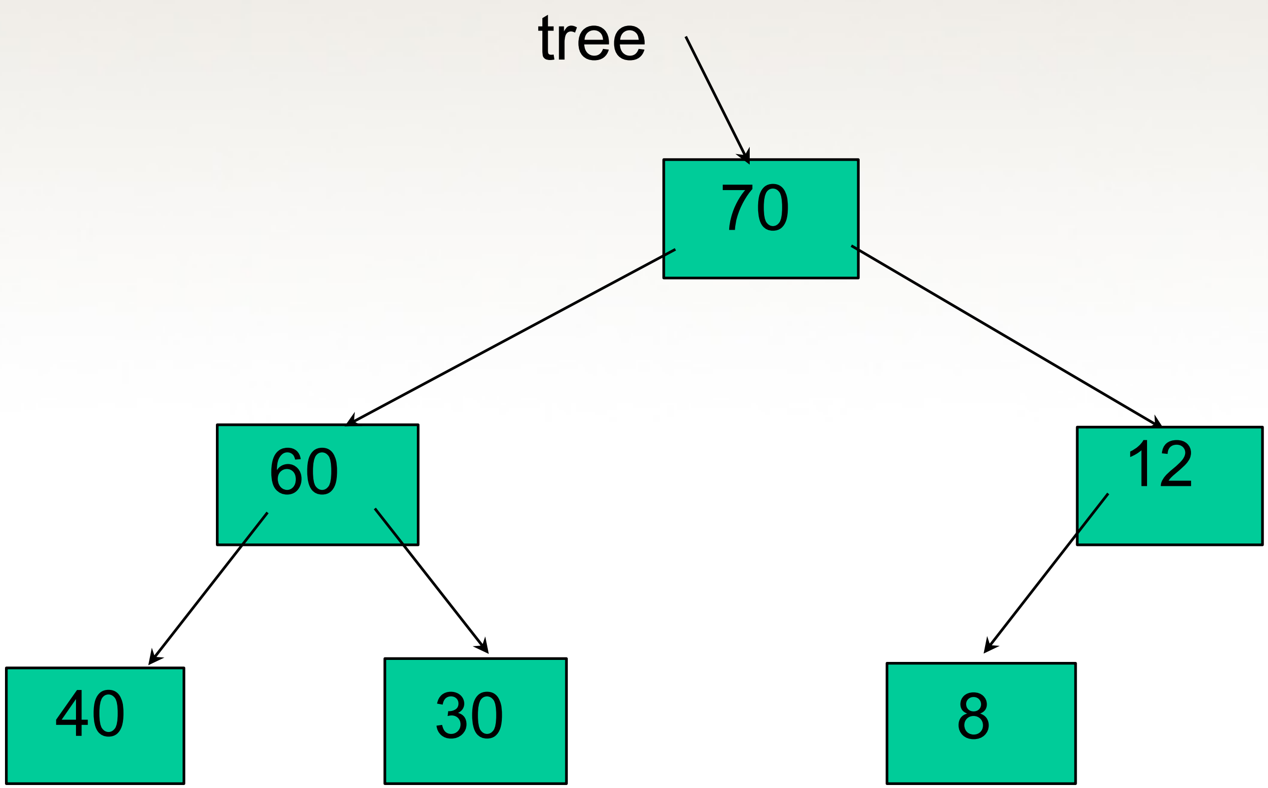
Are these Both Heaps?



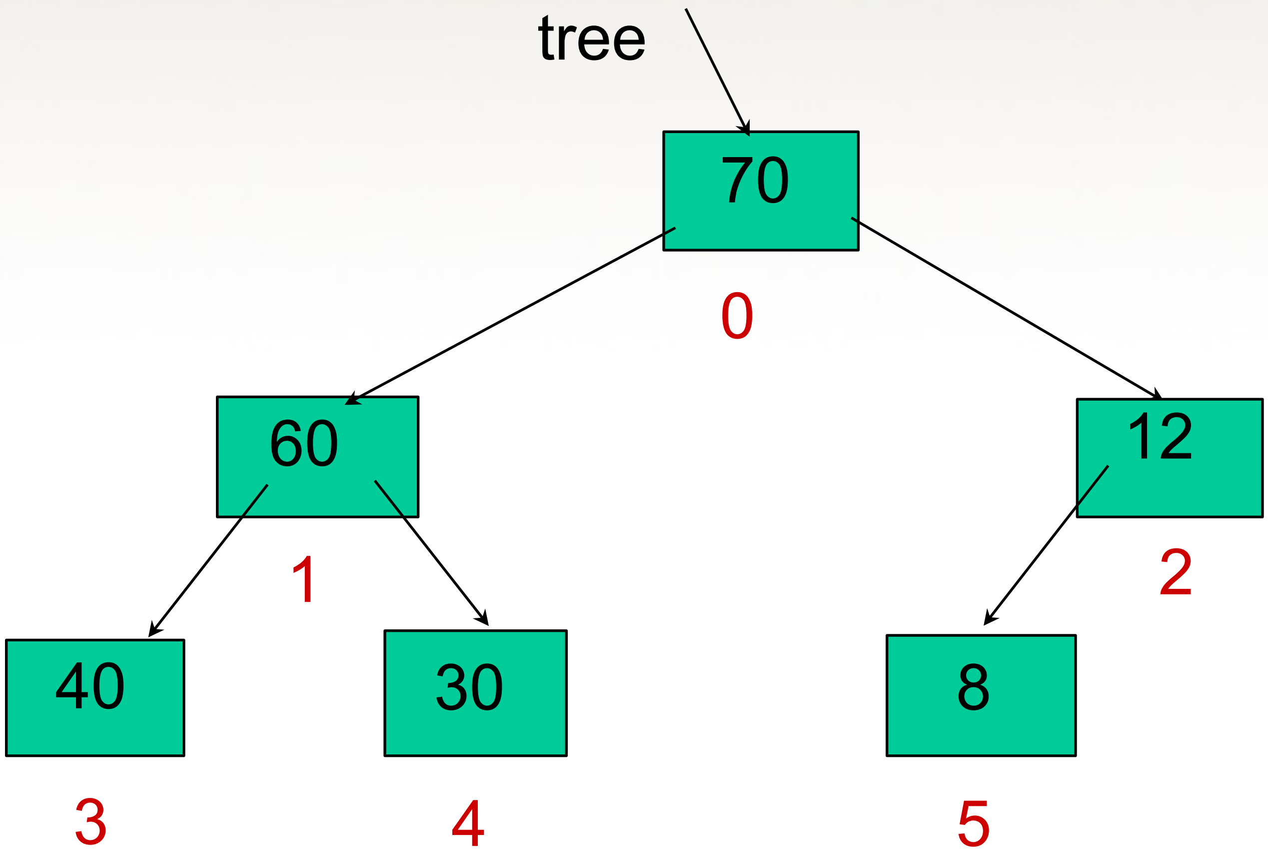
Is this a Heap?



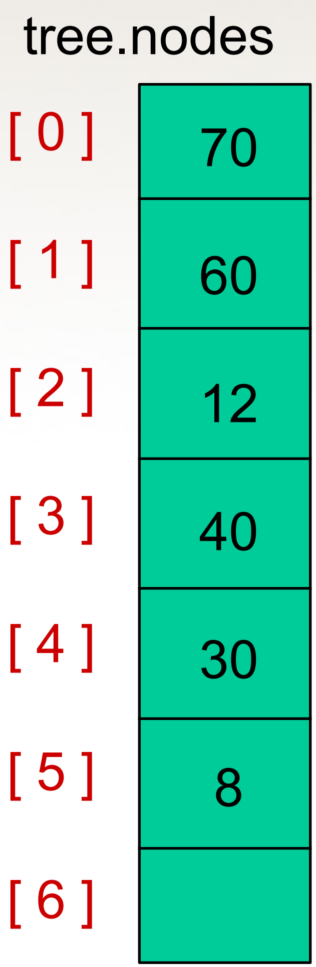
Where is the Largest Element in a Heap Always Found?

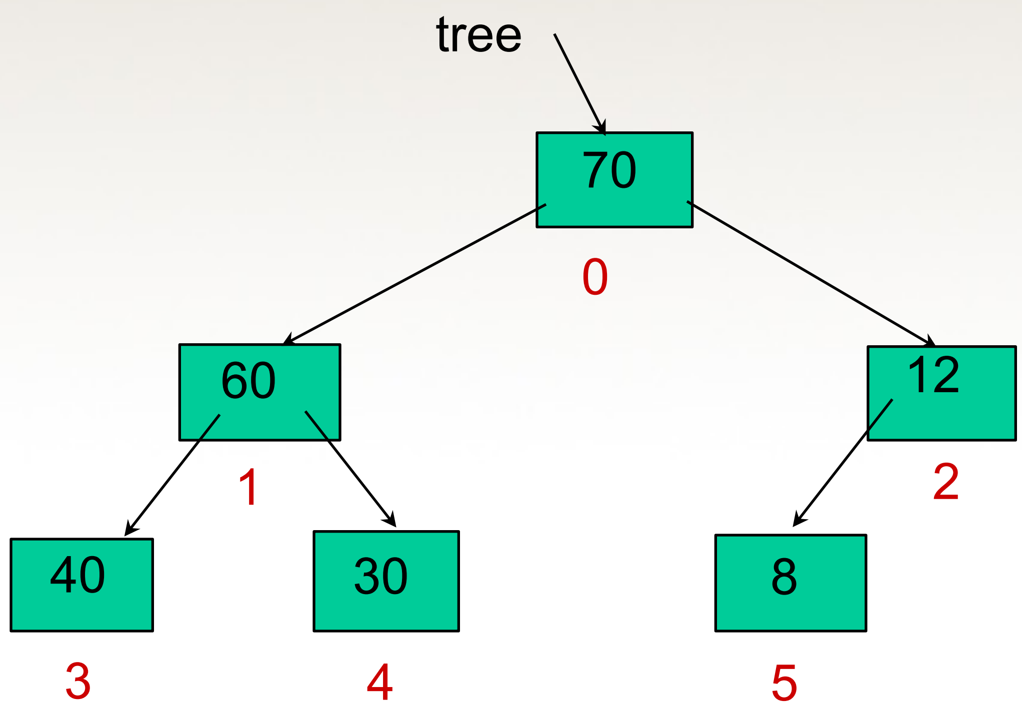


Numbering Nodes Left to Right by Level:



And store tree nodes in array, using the numbering as array Indexes





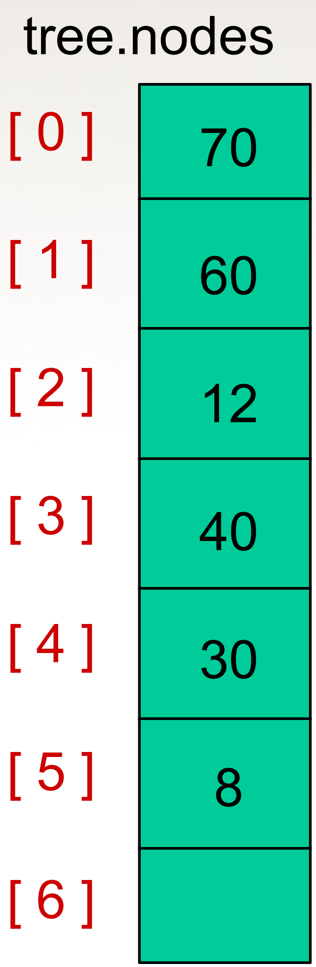
Notice: the relation between a node's numbering with that of its parent:

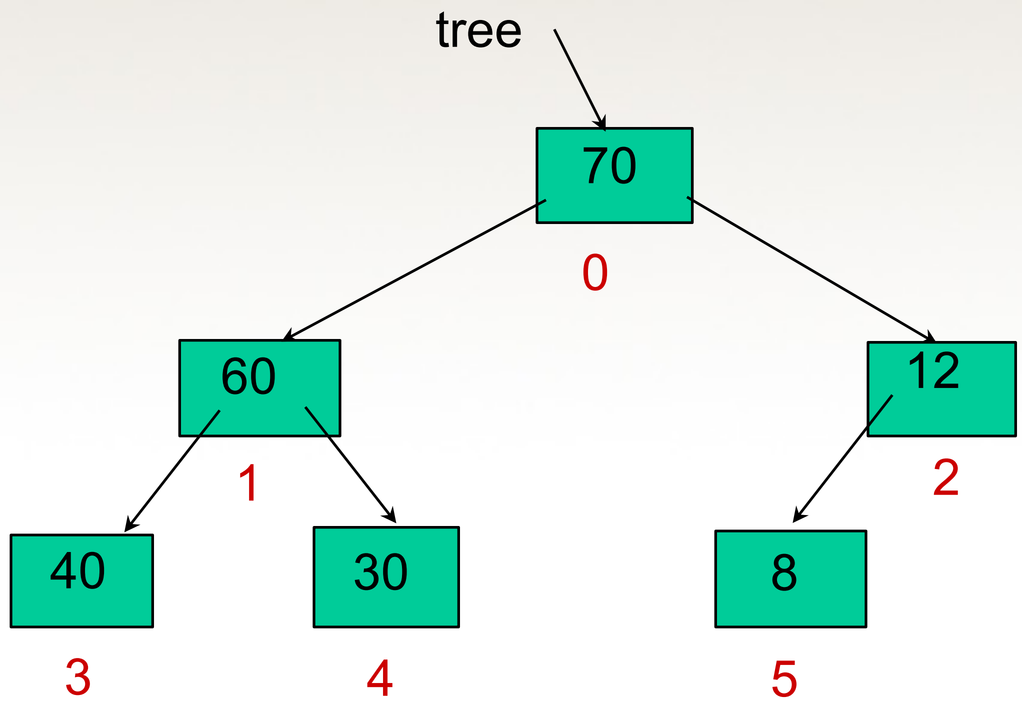
leftChild = ??

rightChild = ??

parent = ??

And store tree nodes in array, using the numbering as array Indexes





Notice: the relation between a node's numbering with that of its parent:

leftChild = (root * 2) + 1 ;

rightChild = (root * 2) + 2 ;

parent = [(child/2)-1]

Use an array to store a complete binary tree

tree elements stored in by level, from left to right:

13	3	4	10	23	31	100	32	
0	1	2	3	4	5	6	7	

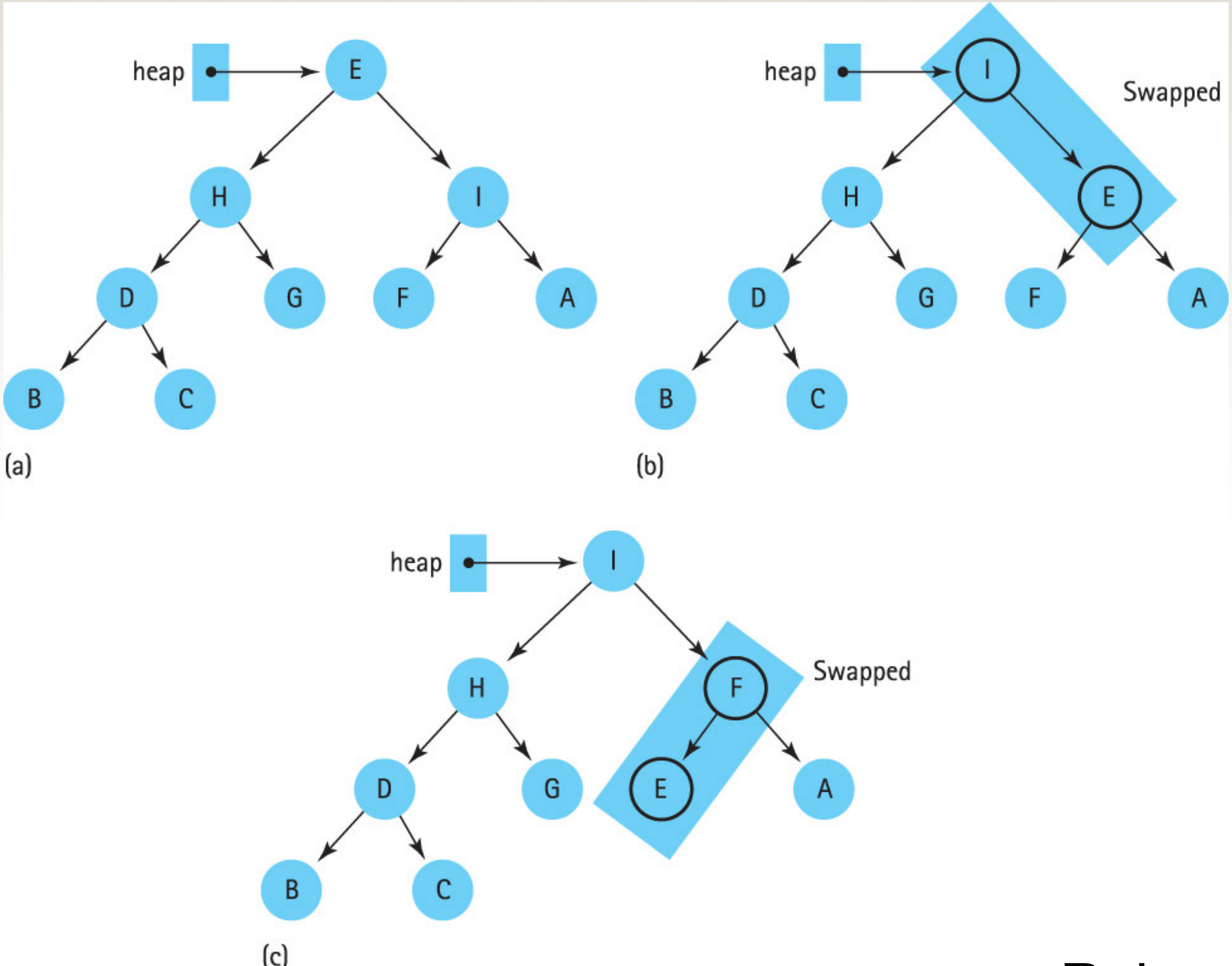
Can you draw the complete binary tree?

Can you find the parent of node 5? (without drawing the tree?)

Where are the left child of node 2, right child?

```
leftChild  =  root * 2 + 1 ;
rightChild  =  root * 2 + 2 ;
```

```
//  HEAP SPECIFICATION


template< class  ItemType >
class  HeapType
{
  void   ReheapDown (int  root ,  int  bottom ) ;
  void   ReheapUp (int  root,  int  bottom ) ;

  ItemType* elements; //ARRAY to be allocated dynamically

  int  numElements ;
};
```


heap
E
H
I
D
G
F
A
B
C
(a)
heap
I
Swapped
H
E
D
G
F
A
B
C
(b)
heap
I
H
F
Swapped
D
G
E
A
B
C
(c)

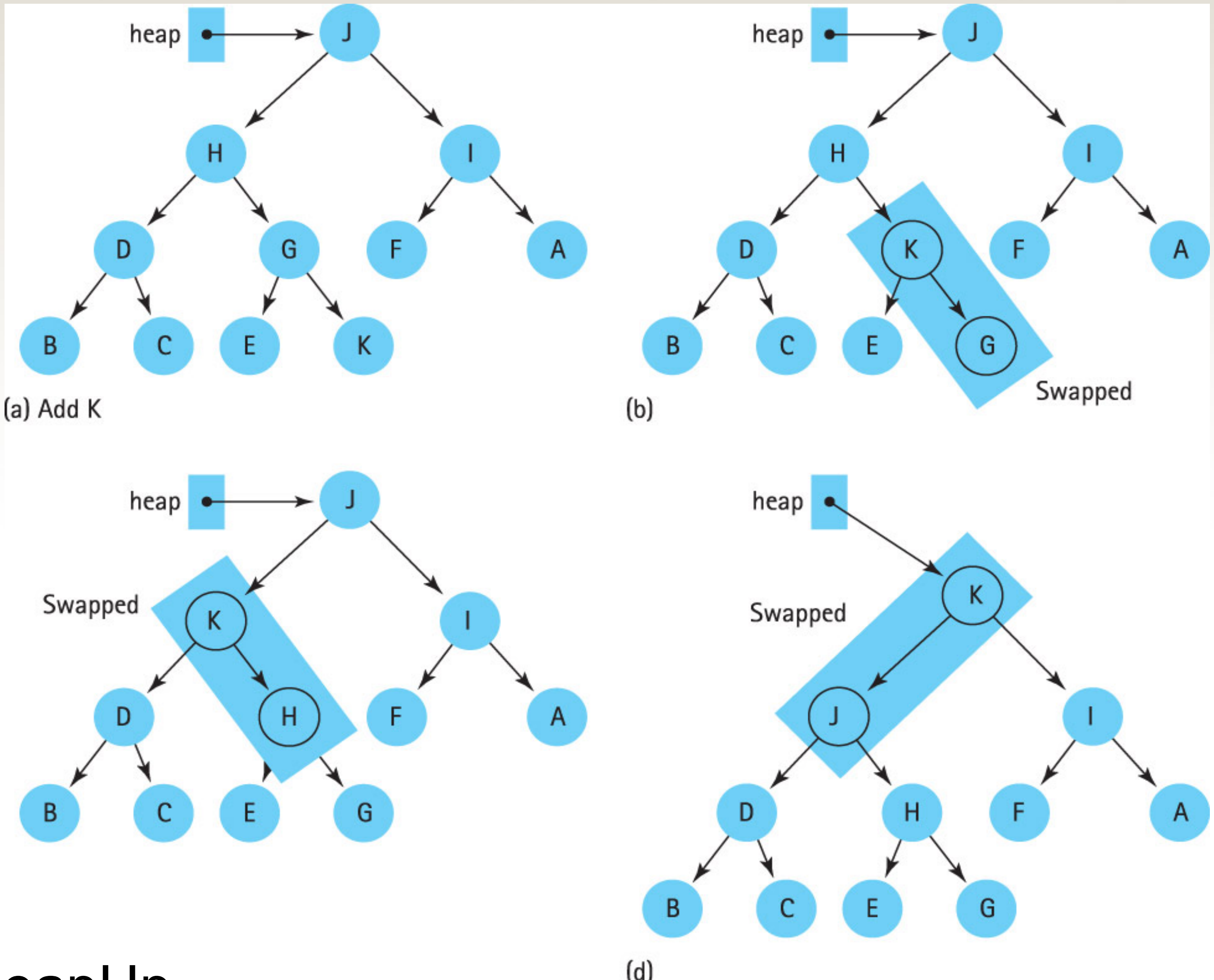
ReheapDown

ReheapDown

```
//  IMPLEMENTATION  OF RECURSIVE HEAP MEMBER FUNCTION

template< class  ItemType >
void   HeapType<ItemType>::ReheapDown ( int root, int  bottom )

// Pre:  root is the index of the node that may violate the
// heap order property
// Post: Heap order property is restored between root and bottom

{
      int  maxChild ;
      int  rightChild ;
      int  leftChild ;

      leftChild  =  root * 2 + 1 ;
      rightChild  =  root * 2 + 2 ;
```

ReheapDown (cont.)

```
  if  ( leftChild  <=  bottom ) // ReheapDown continued
  {
     if ( leftChild  == bottom )
         maxChild  =  leftChild;
     else
        {
          if (elements [ leftChild ] <= elements [ rightChild ] )
               maxChild  =  rightChild;
          else
               maxChild  =  leftChild;
        }
     if ( elements [ root ] < elements [ maxChild ] )
     {
         Swap ( elements [root] , elements [maxChild] );
         ReheapDown ( maxChild, bottom );
        }
   }
}
```


heap
J
H
I
D
G
F
A
B
C
E
K
(a) Add K
heap
J
H
I
D
K
F
A
B
C
E
G
Swapped
(b)
heap
J
Swapped
K
I
D
H
F
A
B
C
E
G
heap
K
Swapped
J
I
D
H
F
A
B
C
E
G
(d)

ReheapUp

ReheapUp

```
template< class  ItemType >
void   HeapType<ItemType>::ReheapUp ( int  root,  int  bottom )

//  Pre:  bottom is the index of the node that may violate the heap
//  order property.  The order property is satisfied from root to
//  next-to-last node.
//  Post:  Heap order property is restored between root and bottom

{
    int  parent ;

    if  ( bottom  > root )
    {
        parent = ( bottom - 1 ) / 2;
        if ( elements [ parent ]  <  elements [ bottom ] )
        {
            Swap ( elements [ parent ], elements [ bottom ] );
            ReheapUp ( root, parent );
        }
    }
}
```

Heaps and Priority Queues (PQ)

- Dequeue returns the highest priority item, which is the root of the heap
- This leaves a hole at the top of the heap; like with UnsortedType, this hole can be filled with the bottom element of the heap
- But now the order property may be violated by the root
- Call ?? to fix it

Heaps and Priority Queues (cont.)

- Enqueue puts an element in the appropriate place in the queue by priority. How?
- Start by putting it as the bottom element, thus preserving the shape property.
- Now the bottom element of the heap may be violating the order property.
- Call ?? to fix it

Class PQType Declaration

```
class FullPQ(){};
class EmptyPQ(){};
template<class ItemType>
class PQType
{
public:
  PQType(int);
  ~PQType();
  void MakeEmpty();
  bool IsEmpty() const;
  bool IsFull() const;
  void Enqueue(ItemType newItem);
  void Dequeue(ItemType& item);
private:
  int length;
  HeapType<ItemType> items;
  int maxItems;
};
```

Class PQType Function Definitions

```
template<class ItemType>
PQType<ItemType>::PQType(int max)
{
  maxItems = max;
  items.elements = new ItemType[max];
  length = 0;
}
template<class ItemType>
void PQType<ItemType>::MakeEmpty()
{
  length = 0;
}
template<class ItemType>
PQType<ItemType>::~PQType()
{
  delete [] items.elements;
}
```

Class PQType Function Definitions

Dequeue

- Set item to root element from queue
- Move last leaf element into root position
- Decrement length
- items.ReheapDown(0, length-1)

Enqueue

- Increment length
- Put newItem in next available position
  - items.ReheapUp(0, length-1)

Code for Dequeue

```
template<class ItemType>
void PQType<ItemType>::Dequeue(ItemType& item)
{
  if (length == 0)
    throw EmptyPQ();
  else
  {
    item = items.elements[0];
    items.elements[0] = items.elements[length-1];
    length--;
    items.ReheapDown(0, length-1);
  }
}
```

Code for Enqueue

```
template<class ItemType>
void PQType<ItemType>::Enqueue(ItemType newItem)
{
  if (length == maxItems)
    throw FullPQ();
  else
  {
    length++;
    items.elements[length-1] = newItem;
    items.ReheapUp(0, length-1);
  }
}
```

Comparison of Priority Queue Implementations

	Enqueue	***Dequeue***
Heap	$O(\log_2 \boldsymbol{N})$	$O(\log_2 \boldsymbol{N})$
Linked List	$O(\boldsymbol{N})$	$O(\boldsymbol{N})$
Binary Search Tree Balanced	 $O(\log_2 \boldsymbol{N})$	 $O(\log_2 \boldsymbol{N})$

Heap Sort

- A simple sort algorithm is to search for the highest value, insert it at the last position, repeat for the next highest value, and so on
- Searching for the next highest value in the list makes this **selection sort** inefficient
- By using a heap, there is no need to search for the next largest element
- Remove the root, ReheapDown, repeat

Building a Heap

- The unsorted list needs to be turned into a heap
- It already satisfies the shape property: there are no holes in the array
- ReheapUp and ReheapDown can reshape a heap, but only if their preconditions are met
- Does the array meet the preconditions?

Example Array

Here's an unsorted array and the tree it forms:

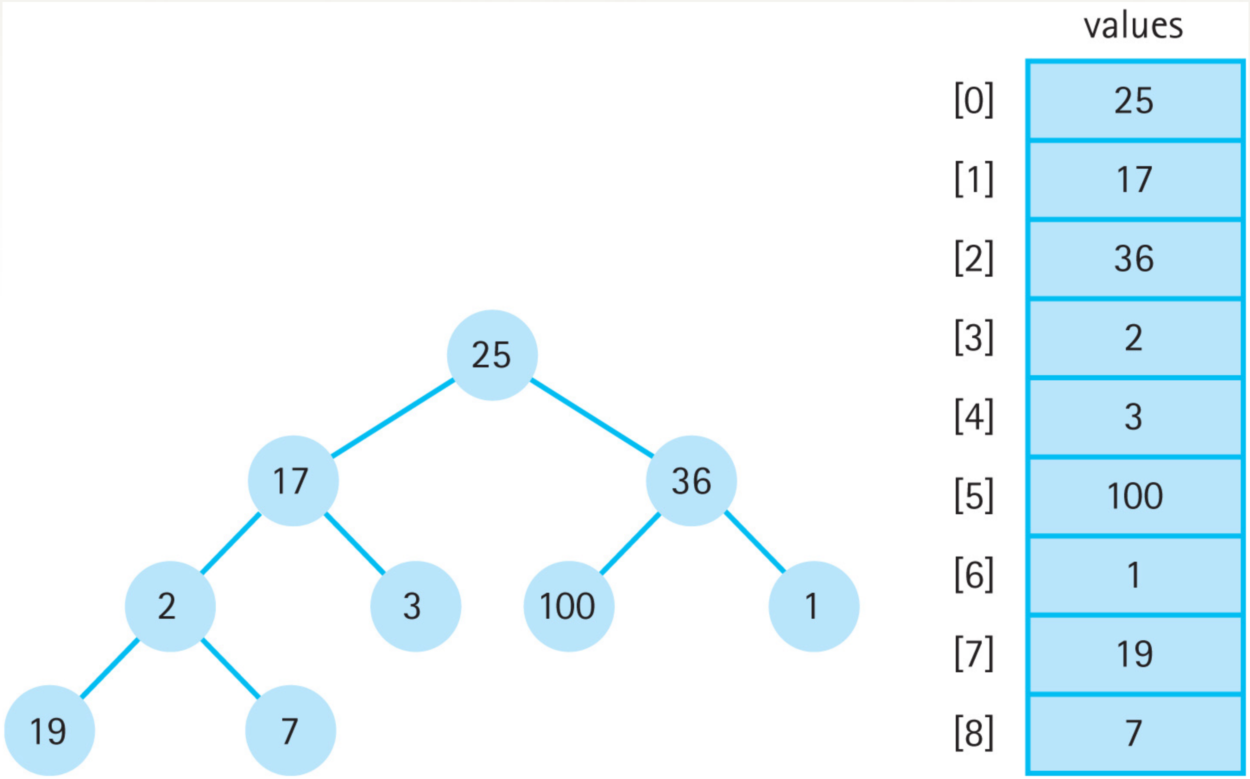


Figure 9.11 An unsorted array and its tree

Building a Heap

- All the leaf nodes are heaps already
- The heap rooted at 2 is almost a heap except for the root; this is perfect for ReheapDown
- Calling ReheapDown on each non-leaf node from the bottom up turns the array into a heap
- For convenience, ReheapDown is written as a global function that takes the heap as an additional parameter

Building a Heap (cont.)

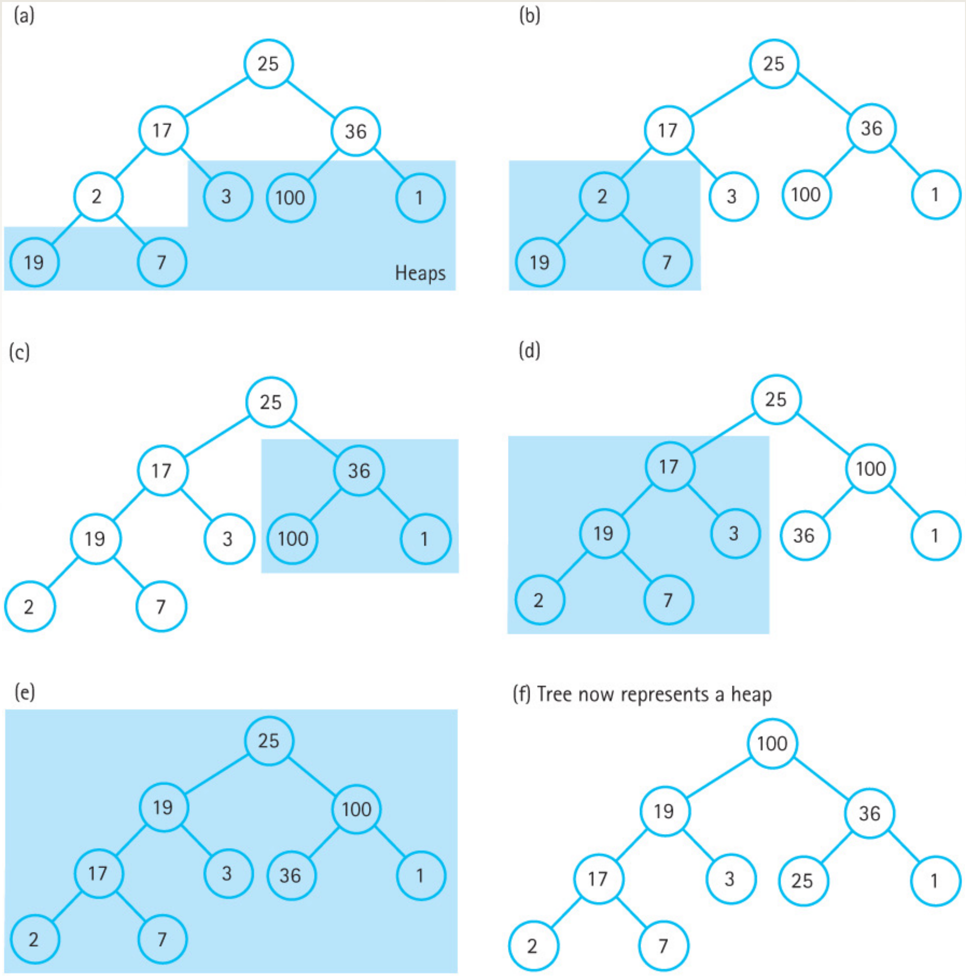


Figure 9.12 The heap-building process (f) Tree now represents a heap

Sorting with the Heap

- Recall that Priority Queue’s Dequeue removes the root and replaces it with the bottom value, reducing the size of the heap by 1
- Heap Sort swaps the root and the bottom value, then calls ReheapDown on the heap
- The root, now at the end of the array, is in the correct position in the sorted list
- The sorted list and the heap coexist in the array

Sorting with the Heap (cont.)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
values	100	19	36	17	3	25	1	2	7
Swap	7	19	36	17	3	25	1	2	100
ReheapDown	36	19	25	17	3	7	1	2	100
Swap	2	19	25	17	3	7	1	36	100
ReheapDown	25	19	7	17	3	2	1	36	100
Swap	1	19	7	17	3	2	25	36	100
ReheapDown	19	17	7	1	3	2	25	36	100
Swap	2	17	7	1	3	19	25	36	100
ReheapDown	17	3	7	1	2	19	25	36	100
Swap	2	3	7	1	17	19	25	36	100
ReheapDown	7	3	2	1	17	19	25	36	100
Swap	1	3	2	7	17	19	25	36	100
ReheapDown	3	1	2	7	17	19	25	36	100
Swap	2	1	3	7	17	19	25	36	100
ReheapDown	2	1	3	7	17	19	25	36	100
Swap	1	2	3	7	17	19	25	36	100
ReheapDown	1	2	3	7	17	19	25	36	100
Exit from sorting loop	1	2	3	7	17	19	25	36	100

Figure 9.14 Effect of HeapSort on the array

Heap Sort

- The heap was only a temporary structure, used internally by the sorting algorithm
- ReheapDown is $O(\log_2 N)$ and was called each time an element was removed, so Heap Sort is an $O(N \log_2 N)$ sort
- Using an external heap would have cost twice as much memory

Heap Sort Code

```
template<class ItemType>
void HeapSort(ItemType values[], int numValues)
// Assumption: Function ReheapDown is available.
// Post: The elements in the array values are sorted by key.
{
  int index;
  // Convert the array of values into a heap.
  for (index = numValues/2 - 1; index >= 0; index--)
    ReheapDown(values, index, numValues-1);
  // Sort the array.
  for (index = numValues-1; index >= 1; index--)
  {
    Swap(values[0], values[index]);
    ReheapDown(values, 0, index-1);
  }
}
```