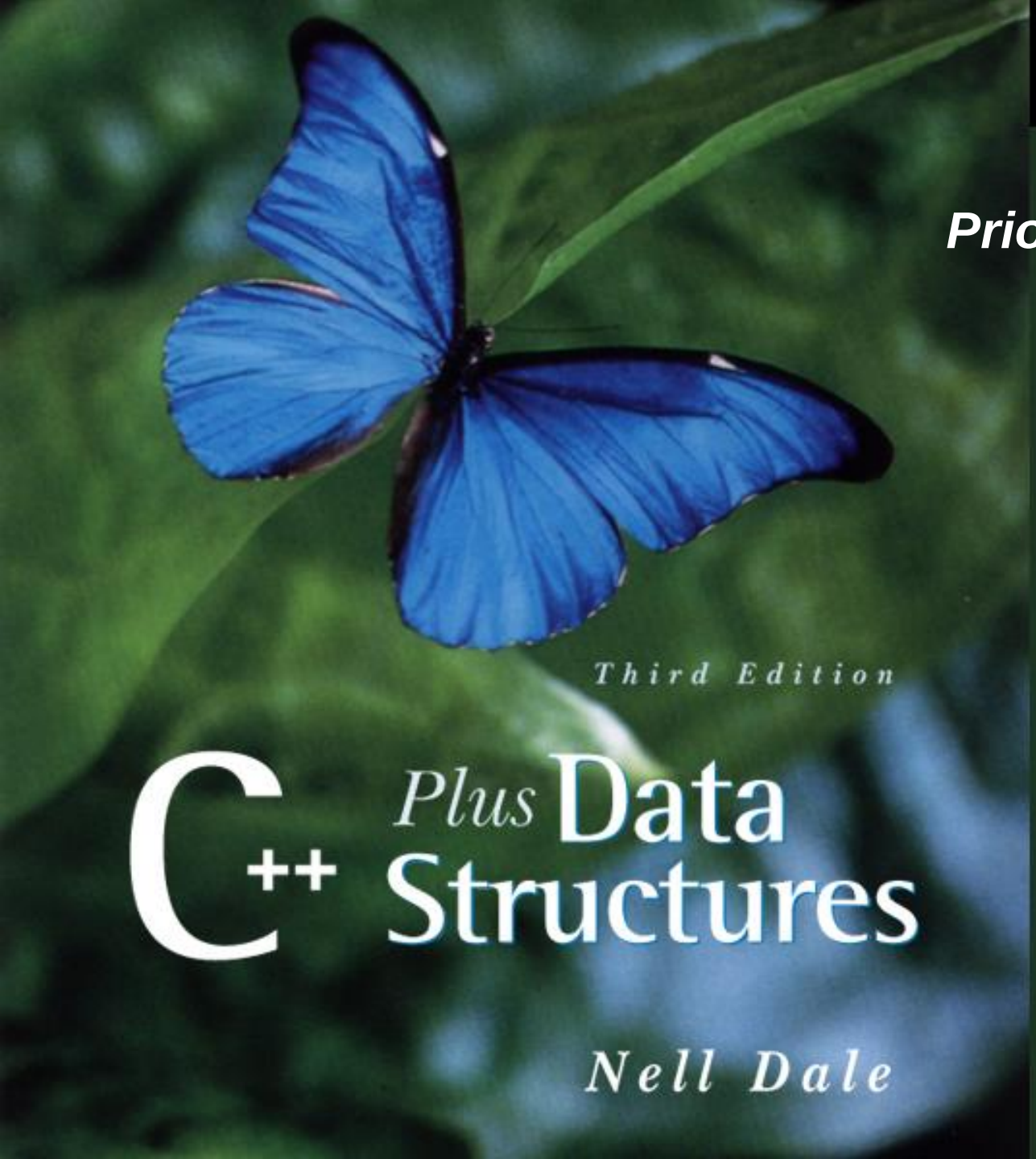


Chapter

9

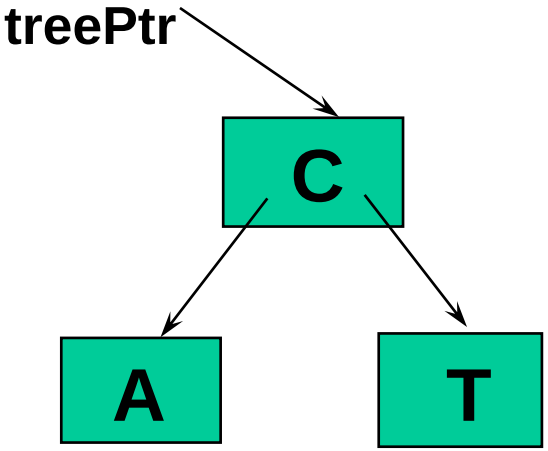
Priority Queues, Heaps, and Graphs

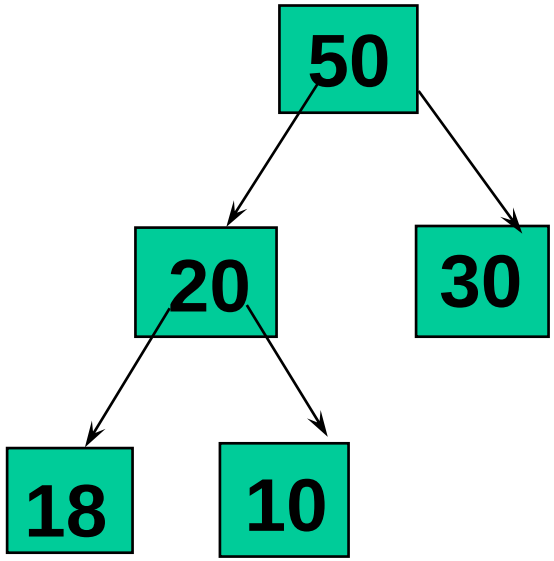
What is a Heap?

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

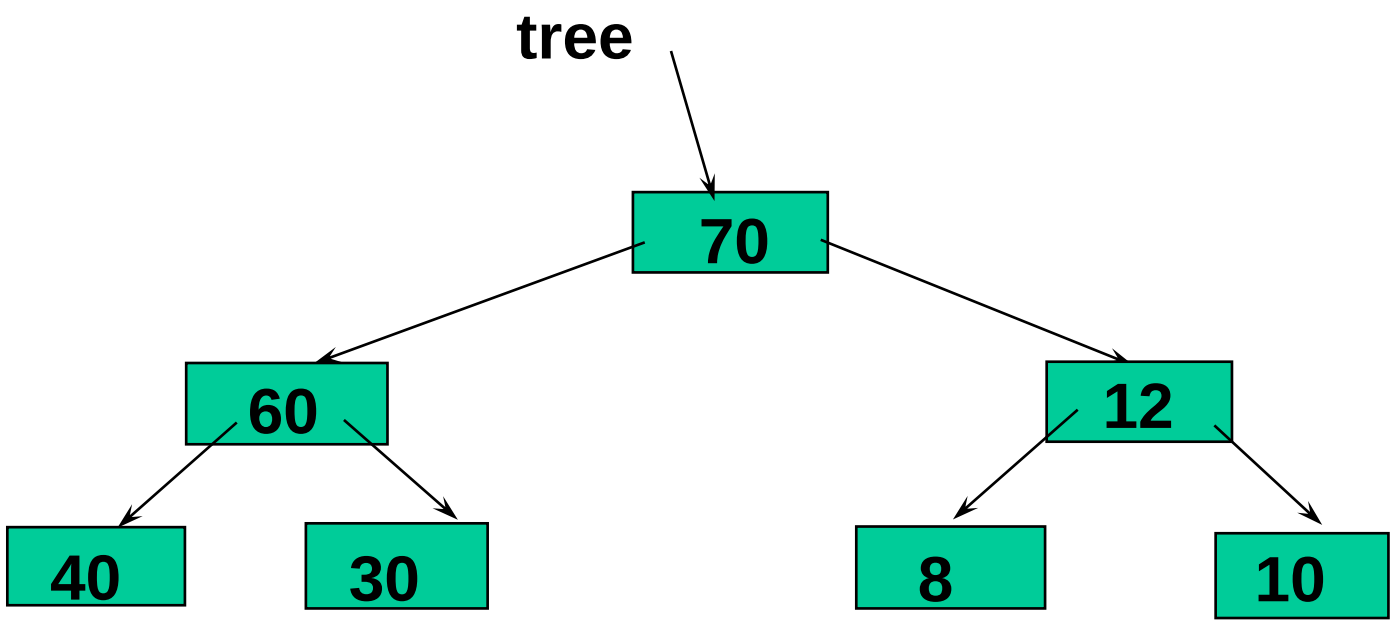
- **SHAPE property: Its shape must be a complete binary tree.**
- **ORDER property: For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

Are these Both Heaps?

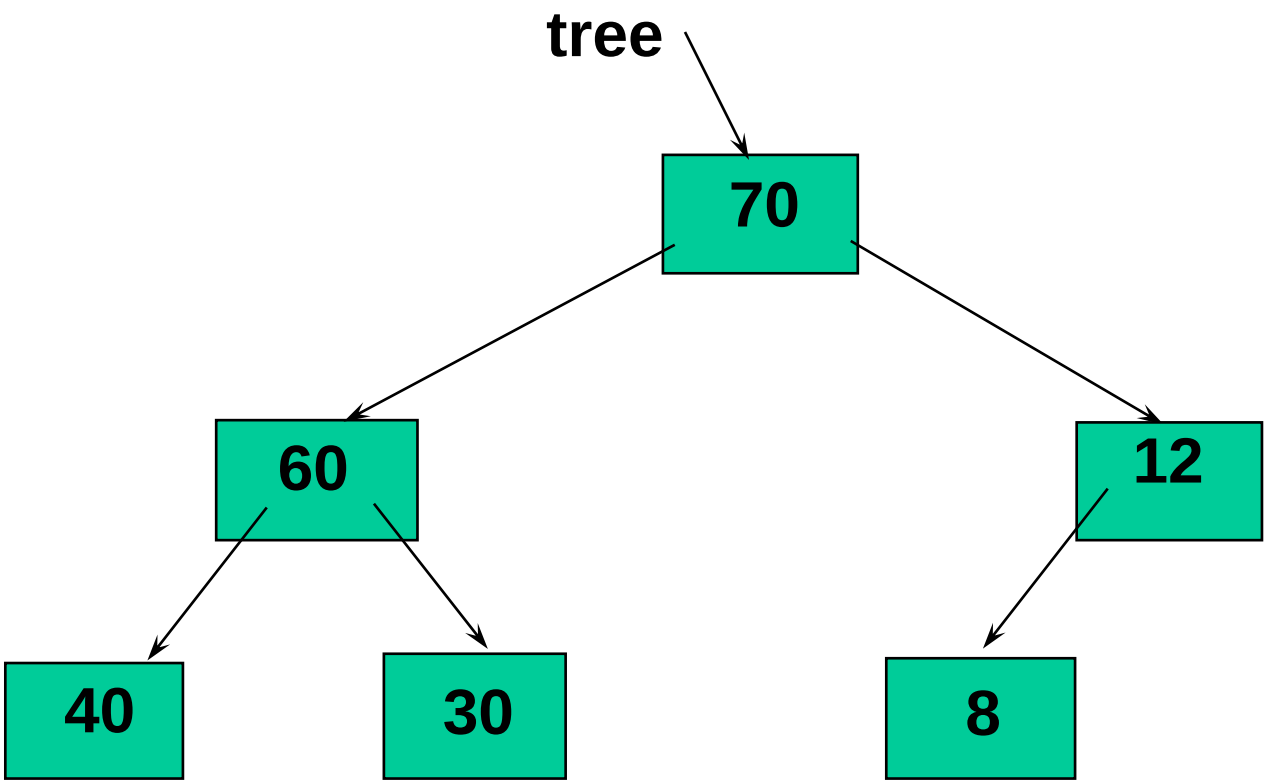




Is this a Heap?

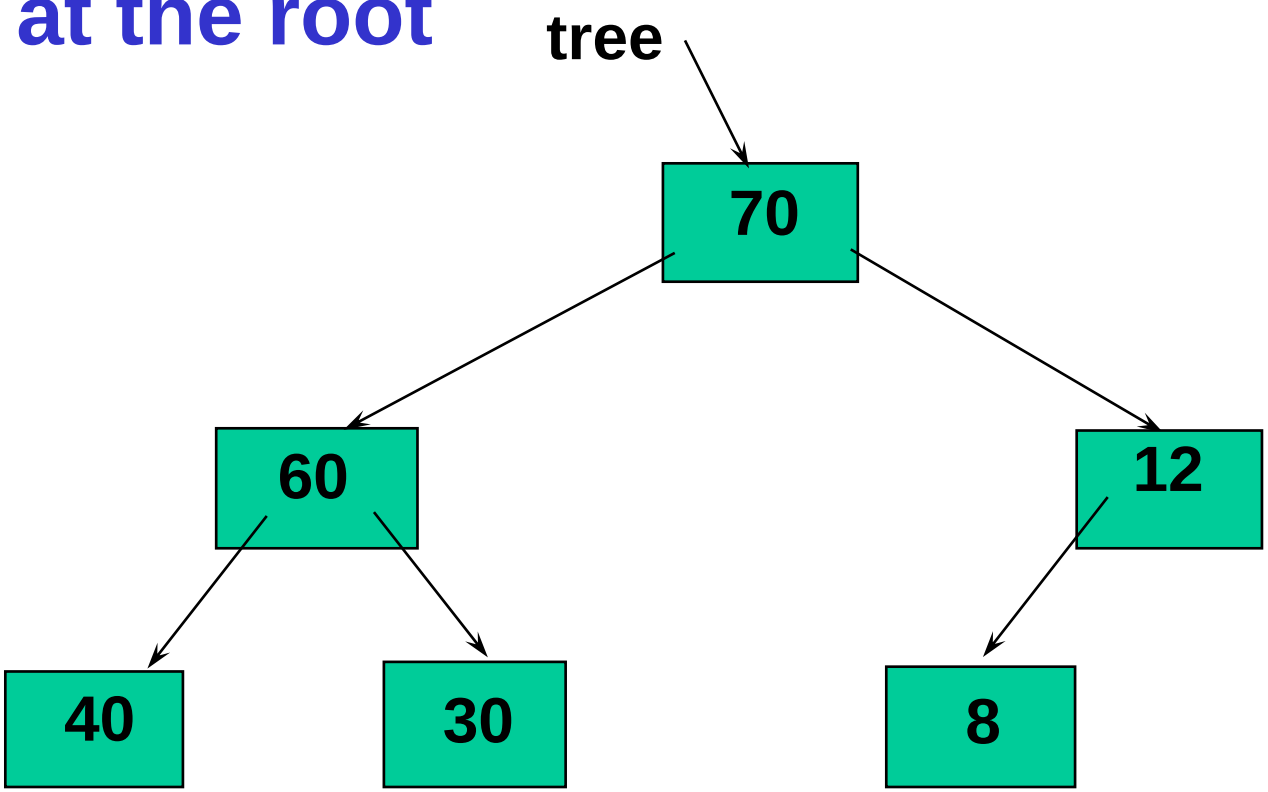


Where is the Largest Element in a Heap Always Found?



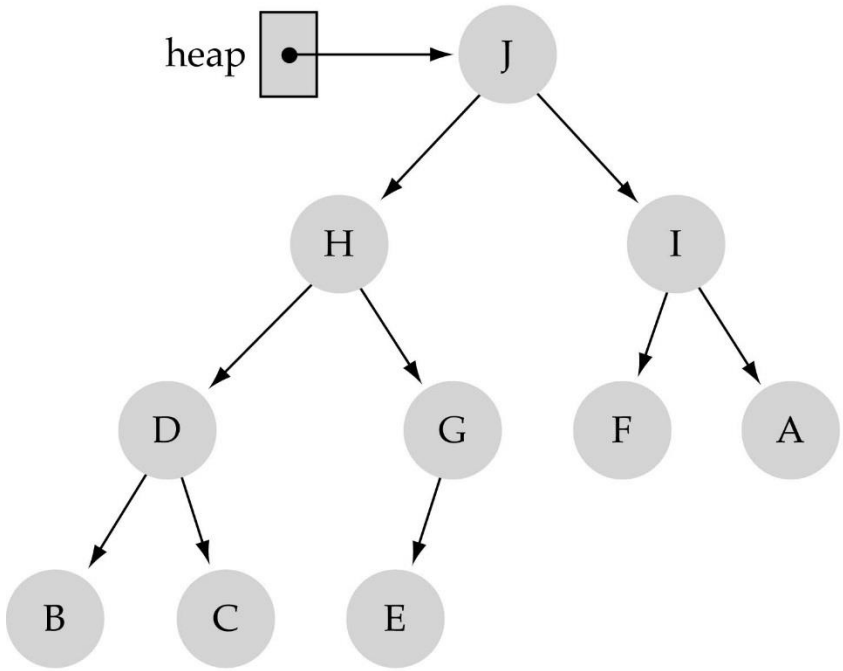
Where is the Largest Element in a Heap Always Found?

From *ORDER property*, the largest value of the heap is always stored at the root



Heap implementation using array representation

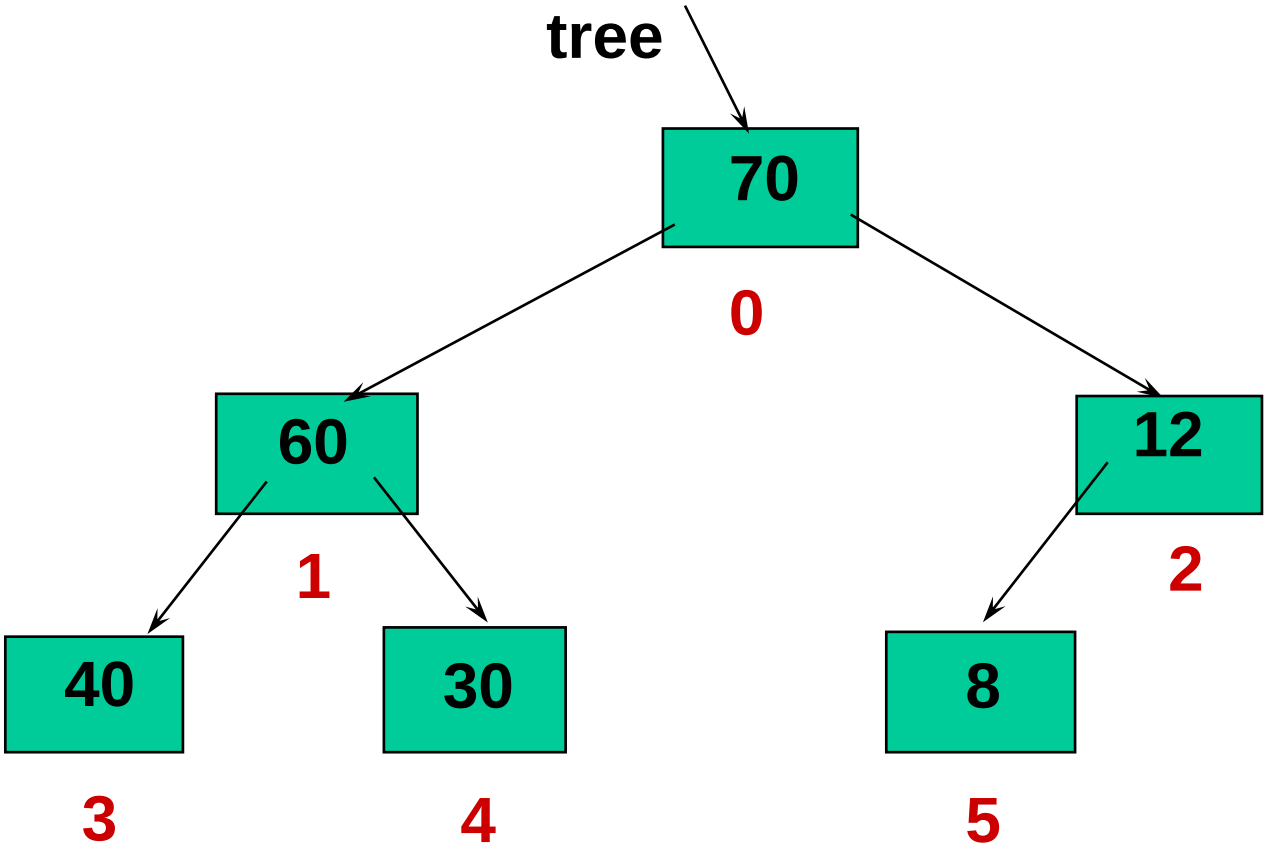
- A heap is a complete binary tree, so it is easy to be implemented using an array representation



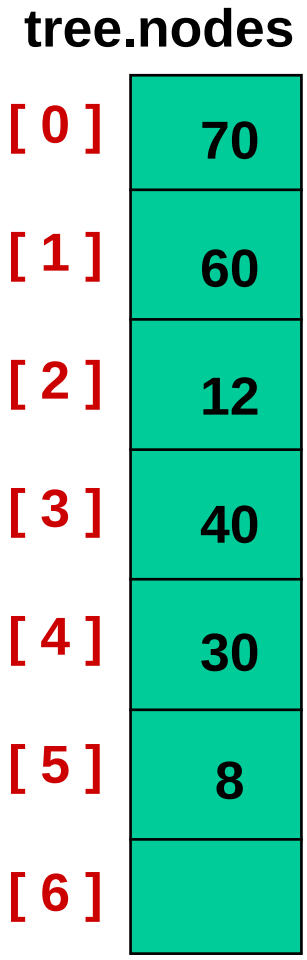
heap.elements

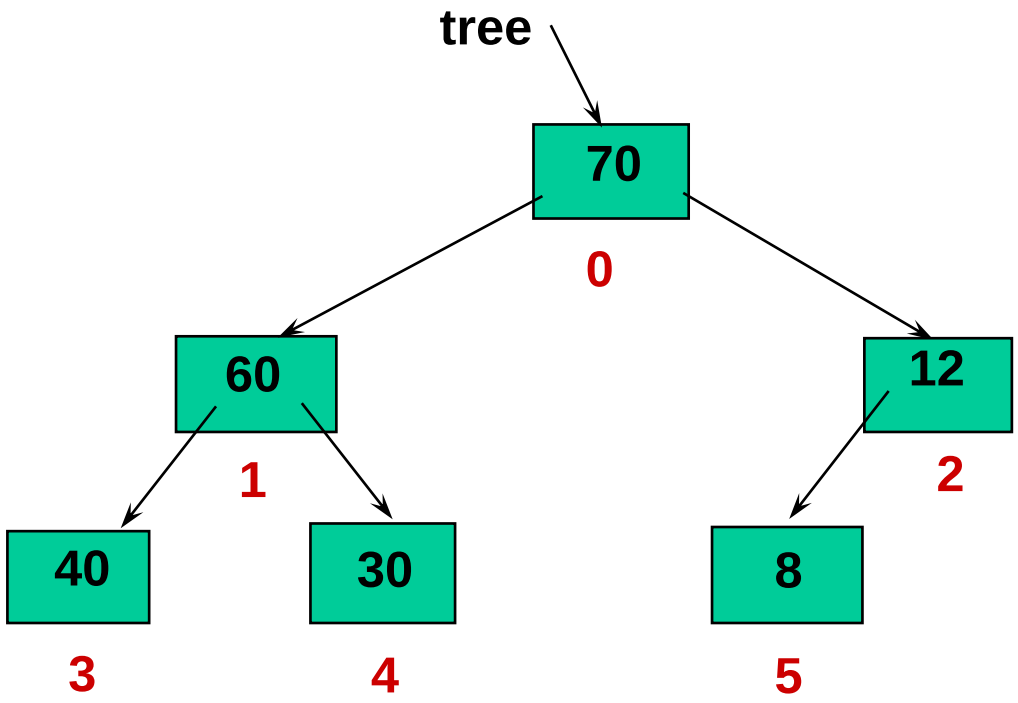
[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

We Can Number the Nodes Left to Right by Level This Way



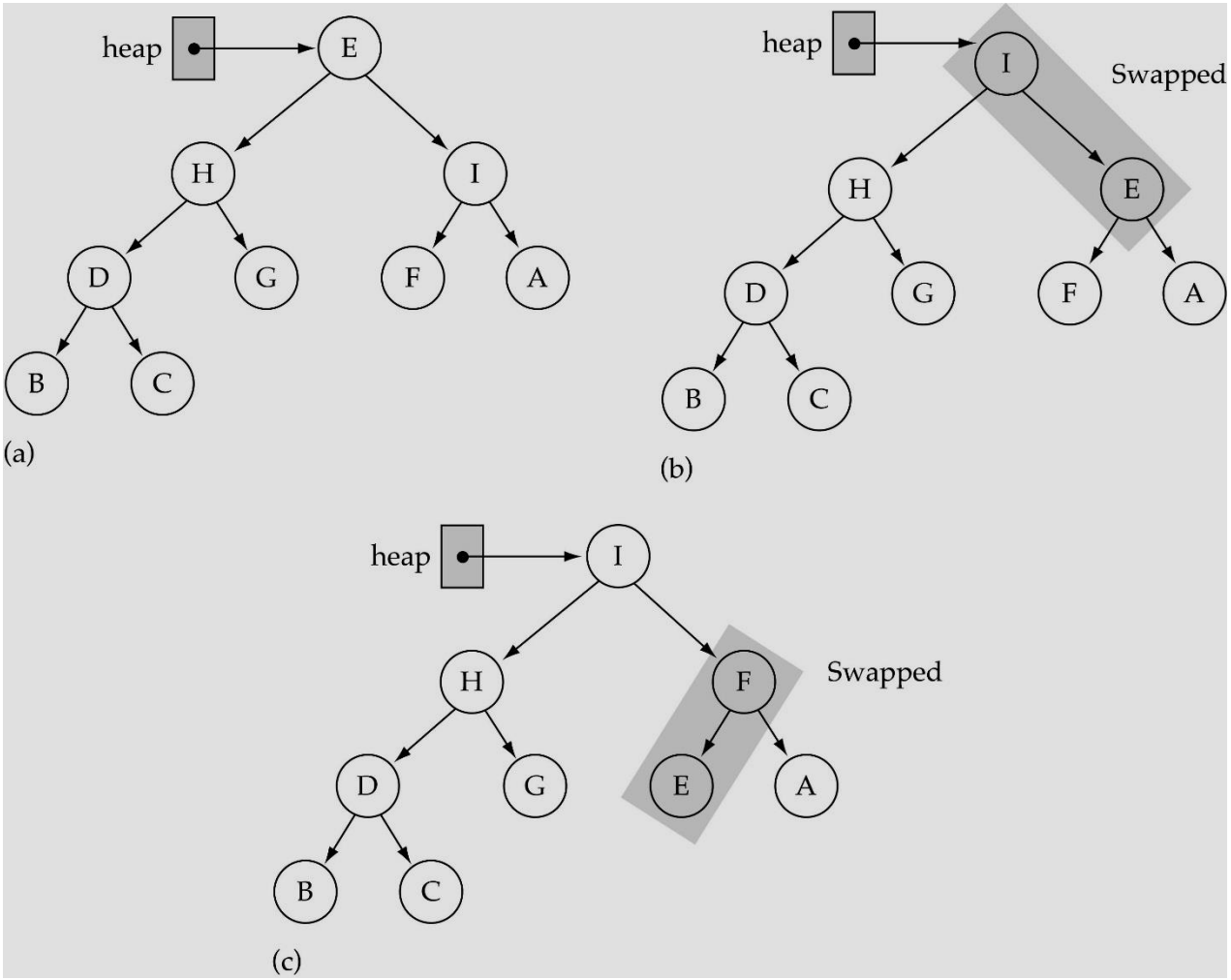
And use the Numbers as Array Indexes to Store the Trees





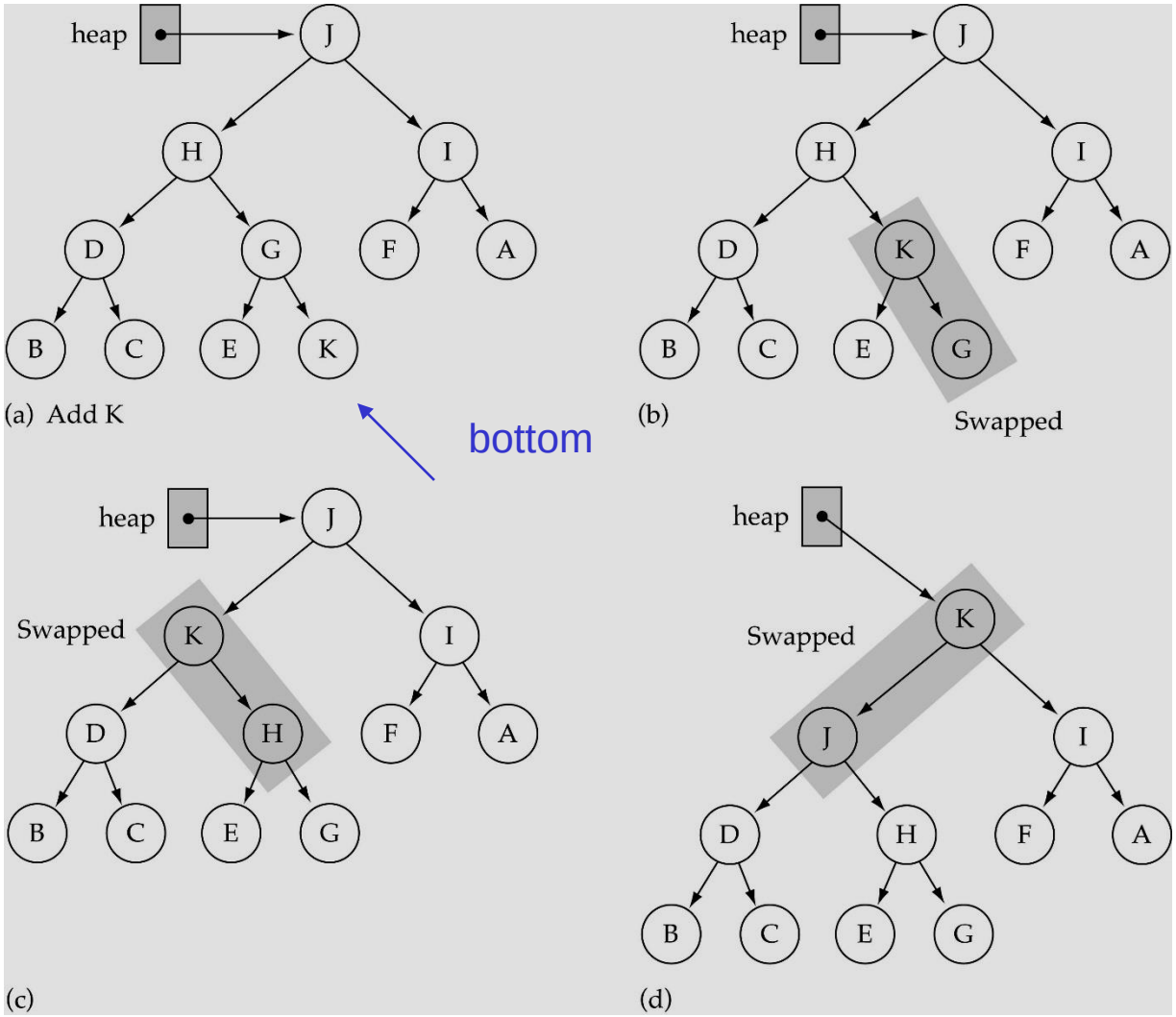
```
//  HEAP SPECIFICATION

//  Assumes  ItemType  is either a built-in simple data
// type or a class with overloaded relational operators.

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

  ItemType* elements; //ARRAY to be allocated dynamically

  int  numElements ;
};
```

The ReheapDown function (used by deleteItem)



Assumption:
heap property is violated at the root of the tree

The ReheapUp function (used by insertItem)



Assumption:
heap property is violated at the rightmost node at the last level of the tree

ReheapDown

```
//   IMPLEMENTATION   OF RECURSIVE HEAP MEMBER FUNCTIONS
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 ;
```

rightmost node in the last level

ReheapDown (cont)

```
if  ( leftChild  <=  bottom ) // Is there leftChild?
{
   if ( leftChild  == bottom )  // only one child
       maxChild  =  leftChld ;
   else // two children
      {
        if (elements [ leftChild ] <= elements [ rightChild ] )
             maxChild  =  rightChild ;
        else
             maxChild  =  leftChild ;
      }
   if  ( elements [ root ] < elements [ maxChild ] )
      {
          Swap ( elements [ root ] , elements [ maxChild ] ) ;
          ReheapDown ( maxChild, bottom ) ;
      }
  }
}
```

ReheapUp

rightmost node
in the last level

```
//   IMPLEMENTATION                continued

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 ) // tree is not empty
    {
        parent = ( bottom - 1 ) / 2;
        if ( elements [ parent ]  <  elements [ bottom ] )
        {
            Swap ( elements [ parent ], elements [ bottom ] ) ;
            ReheapUp ( root, parent ) ;
        }
    }
}
```

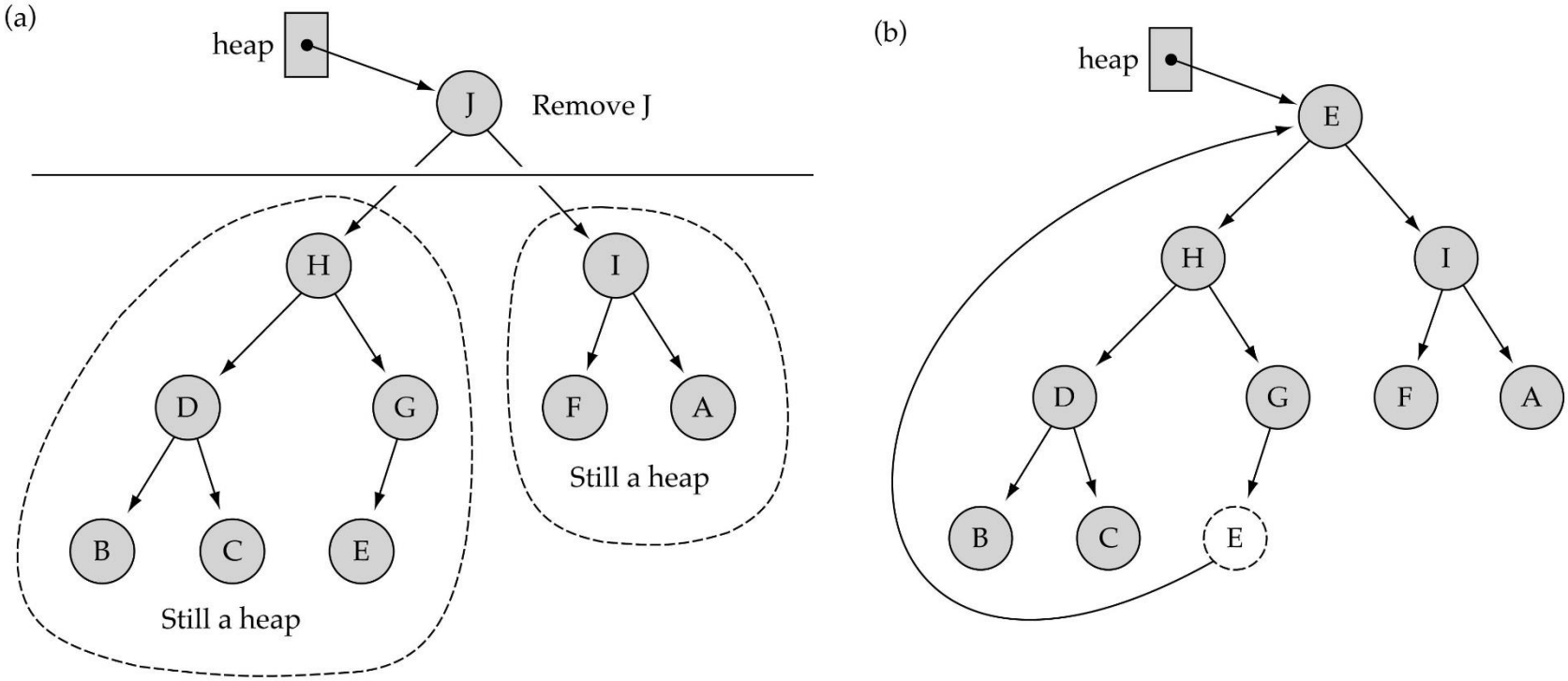
Removing the largest element from the heap

(1) Copy the bottom rightmost element to the root

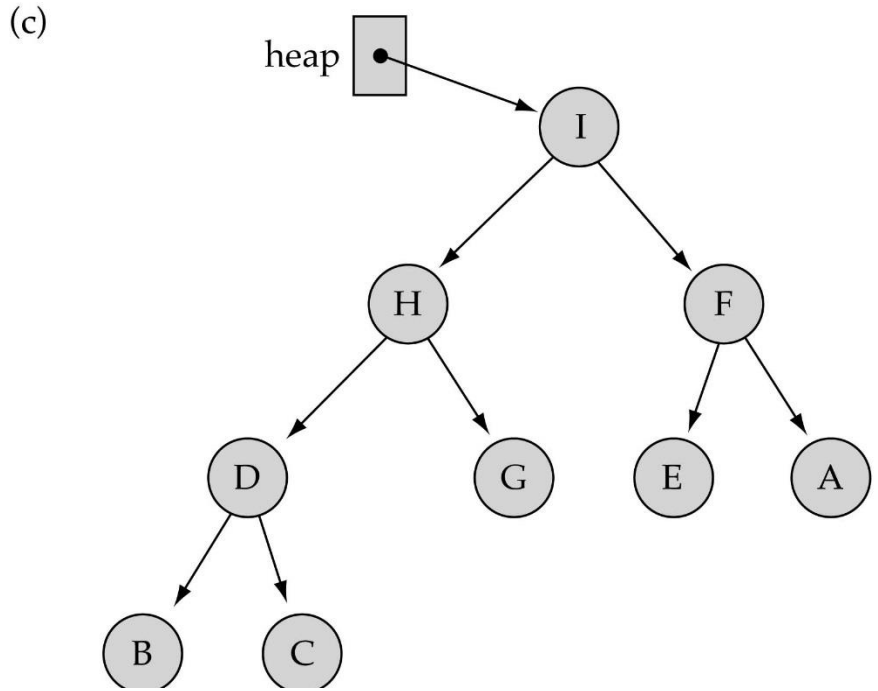
(2) Delete the bottom rightmost node

(3) Fix the heap property by calling *ReheapDown*

Removing the largest element from the heap (cont.)



Removing the largest element from the heap (cont.)

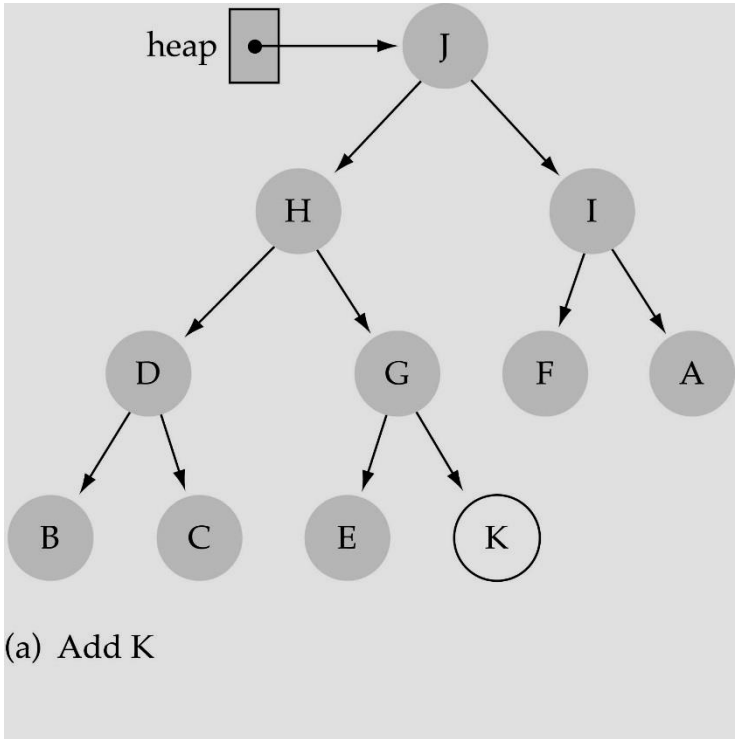


Inserting a new element into the heap

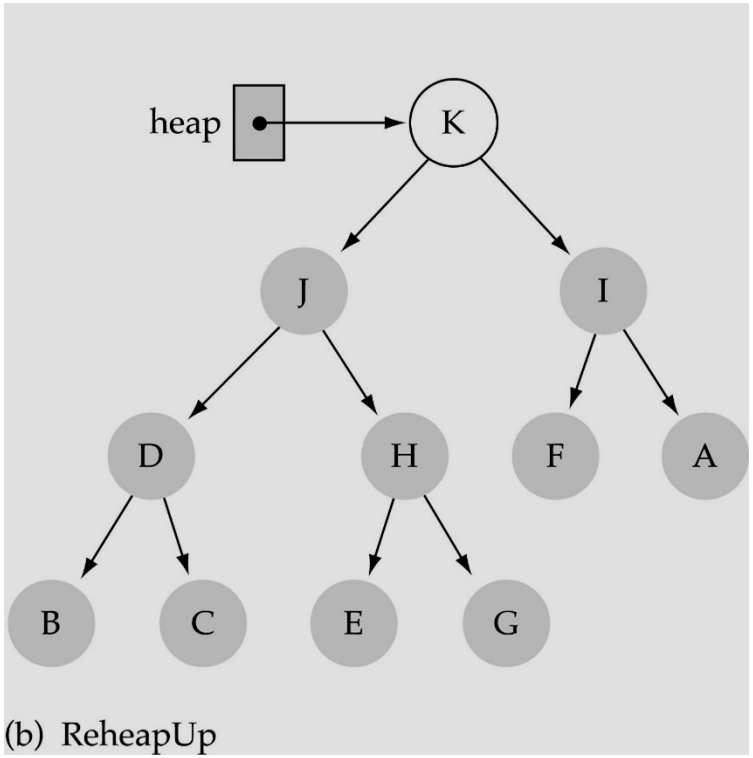
(1) Insert the new element in the next bottom leftmost place

(2) Fix the heap property by calling *ReheapUp*

Inserting a new element into the heap (cont.)



(a) Add K



(b) ReheapUp

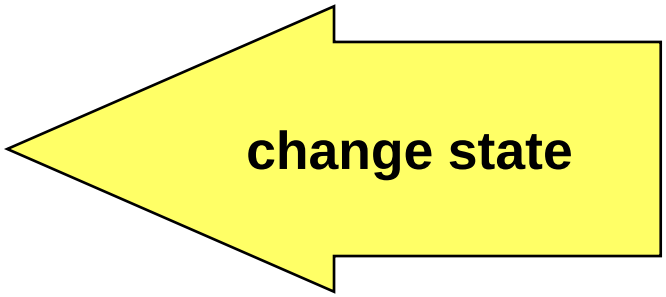
Priority Queue

A priority queue is an ADT with the property that only the highest-priority element can be accessed at any time.

ADT Priority Queue Operations

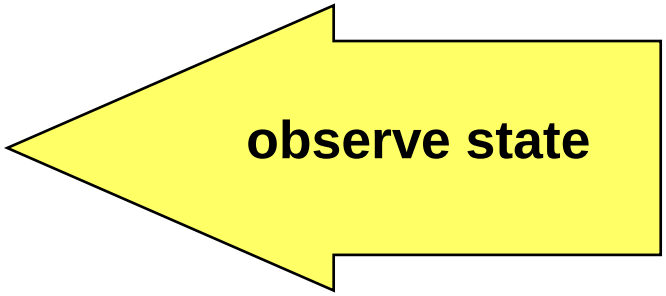
Transformers

- MakeEmpty
- Enqueue
- Dequeue



Observers

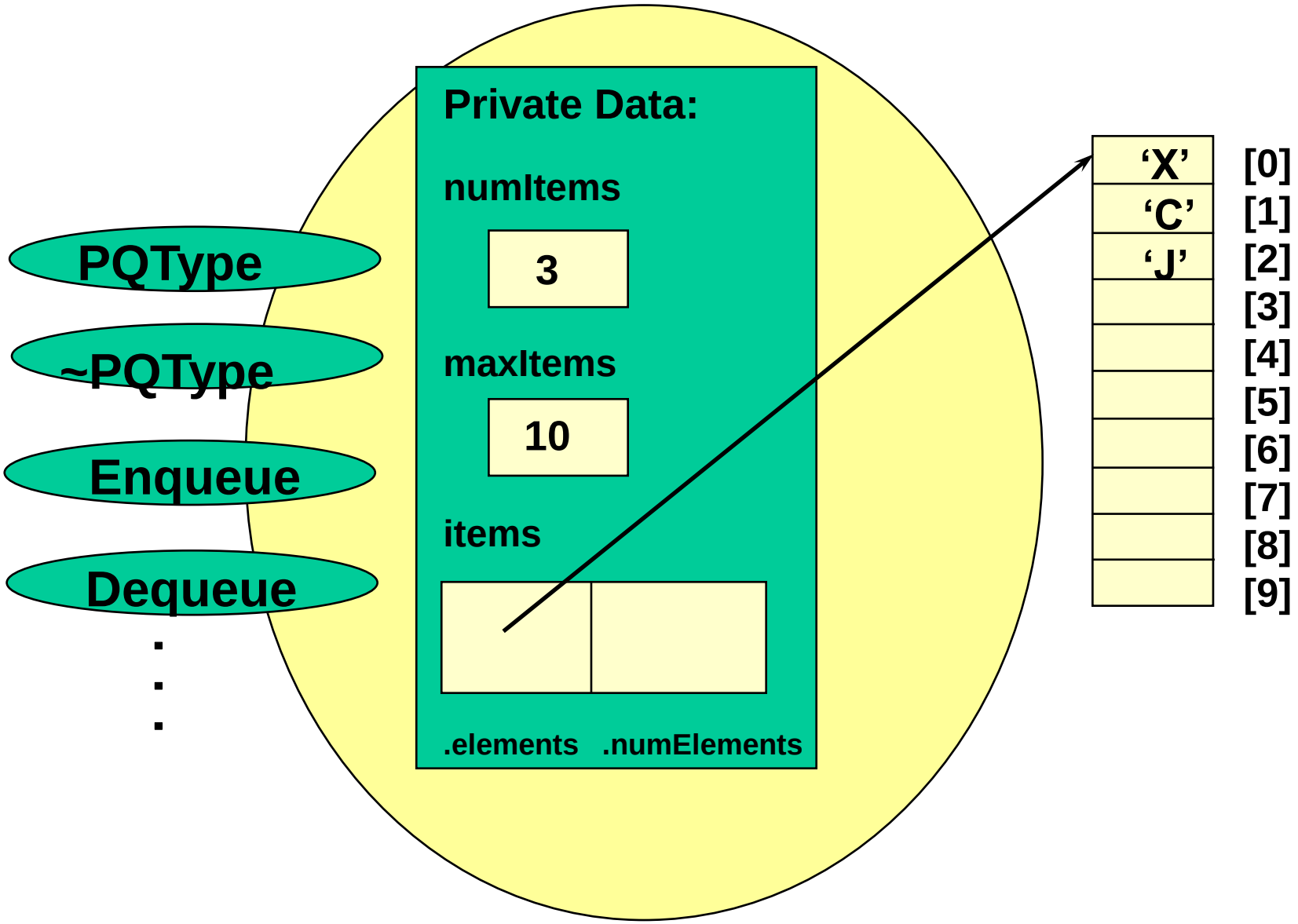
- IsEmpty
- IsFull



Implementation Level

- There are many ways to implement a priority queue
  - **An unsorted List-** dequeuing would require searching through the entire list
  - **An Array-Based Sorted List-** Enqueuing is expensive
  - **A Linked Sorted List-** Enqueuing again is 0(N)
  - **A Binary Search Tree-** On average, $0(\log_2 N)$ steps for both enqueue and dequeue
  - **A Heap**- guarantees $0(\log_2 N)$ steps, even in the worst case

class PQType<char>



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($\log_2 N$)**	**O($\log_2 N$)**
Linked List	**O(*N*)**	**O(*N*)**
Binary Search Tree		
Balanced	**O($\log_2 N$)**	**O($\log_2 N$)**
Skewed	**O(*N*)**	**O(*N*)**