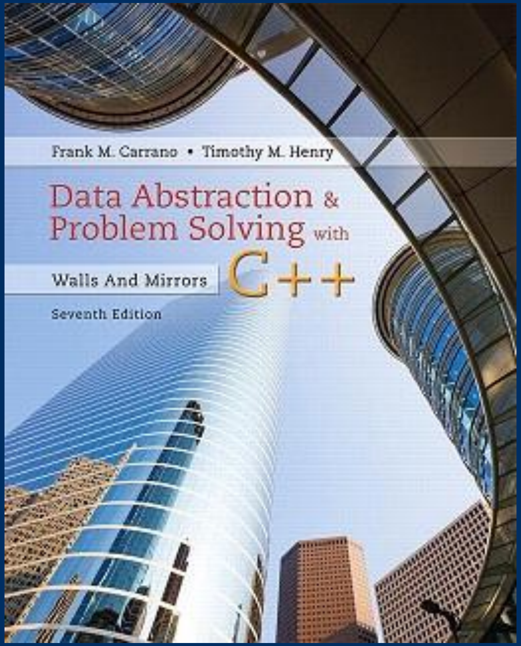
Chapter 17
Heaps

CS 302 - Data Structures

M. Abdullah Canbaz


Frank M. Carrano • Timothy M. Henry
Data Abstraction & Problem Solving with
Walls And Mirrors
C++
Seventh Edition

Modified from authors' slides

Reminders

- Assignment 5 is available
  - Due April 11th at 2pm
- TA
  - Athanasia Katsila,
    Email: akatsila [at] nevada {dot} unr {dot} edu,
    Office Hours: Tuesday, 10:30 am - 12:30 pm at SEM 211
- Unfortunately No!
  - Please take quizzes on time!
- No 70% rule!

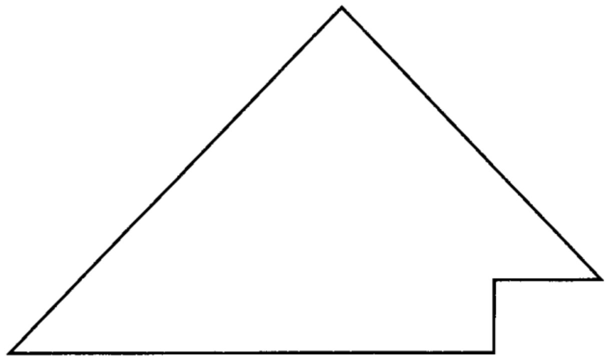
The ADT Heap

- A heap is a complete binary tree that either is
  - Empty or …
  - Whose root contains a value $\geq$ each of its children and has heaps as its subtrees
- It is a special binary tree … different in that
  - It is ordered in a weaker sense
  - it will always be a complete binary tree

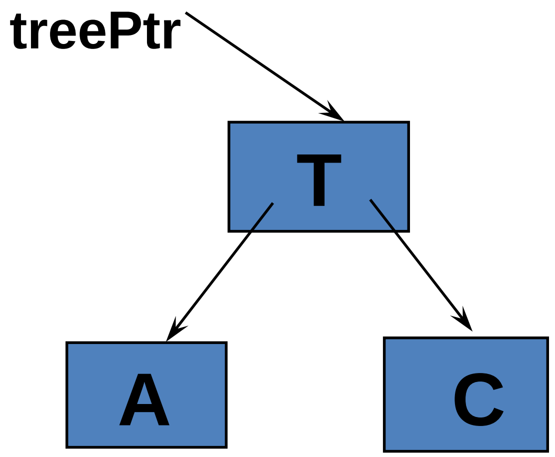
What is a Heap?

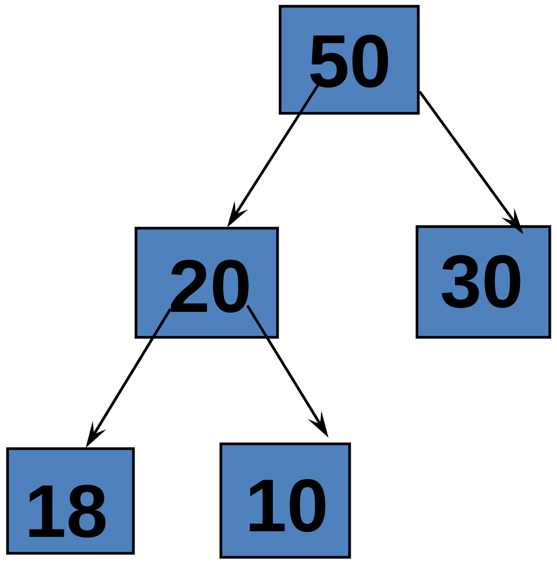
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- **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.**

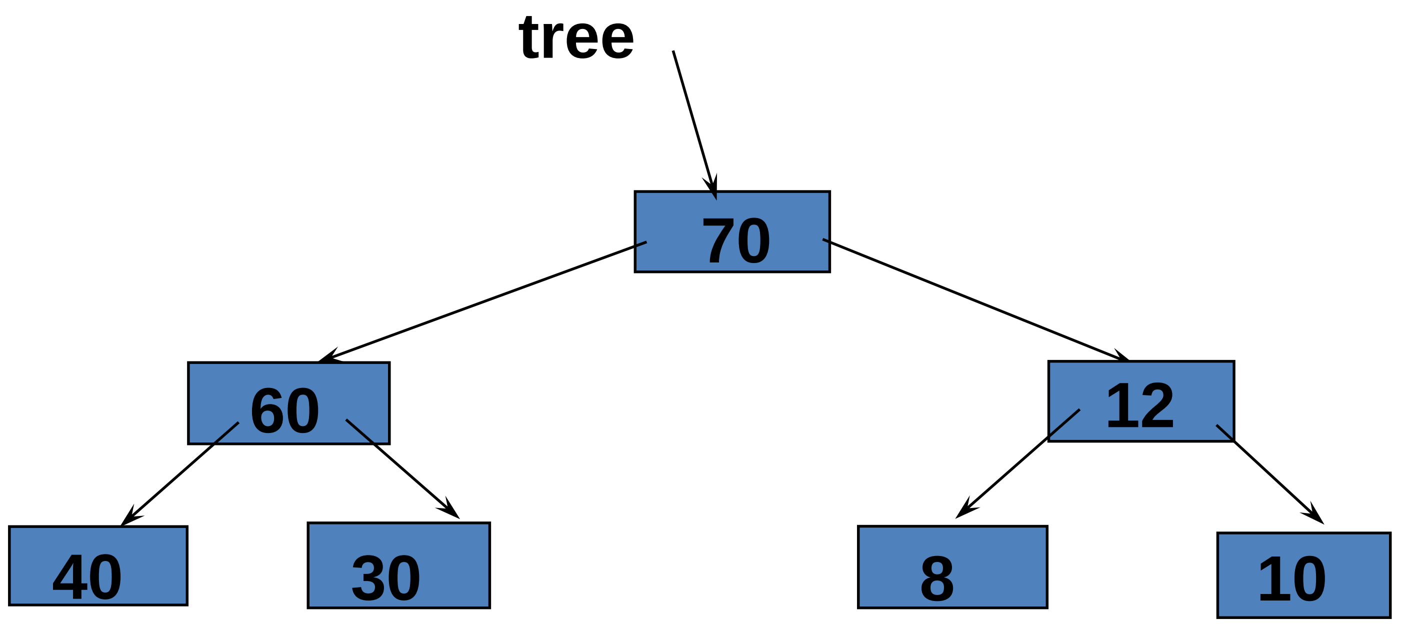


Are these Both Heaps?

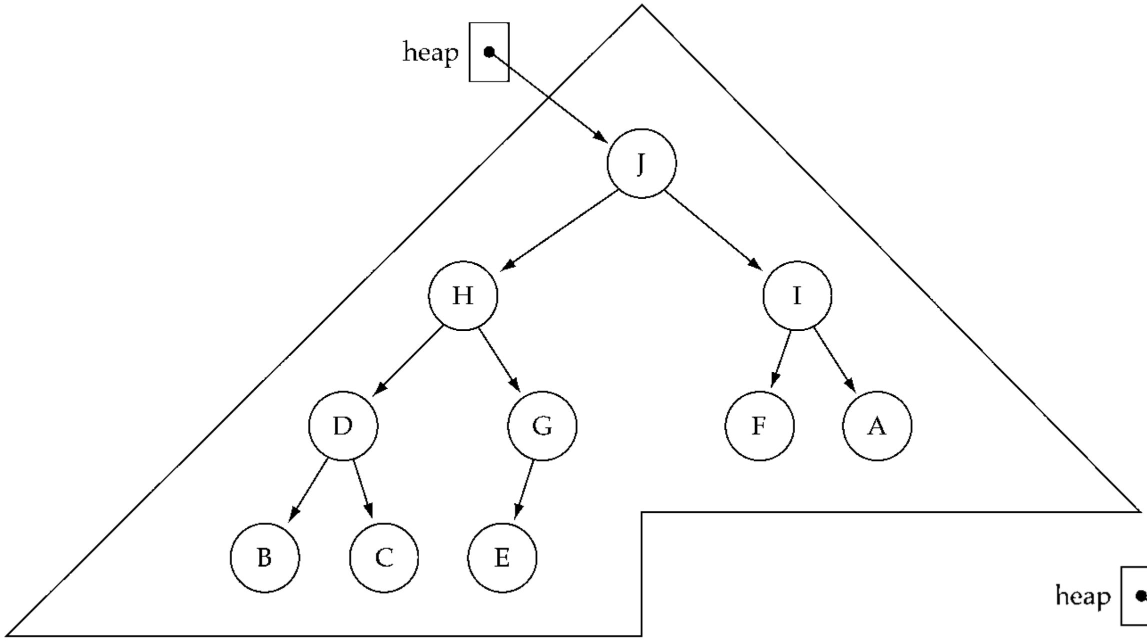


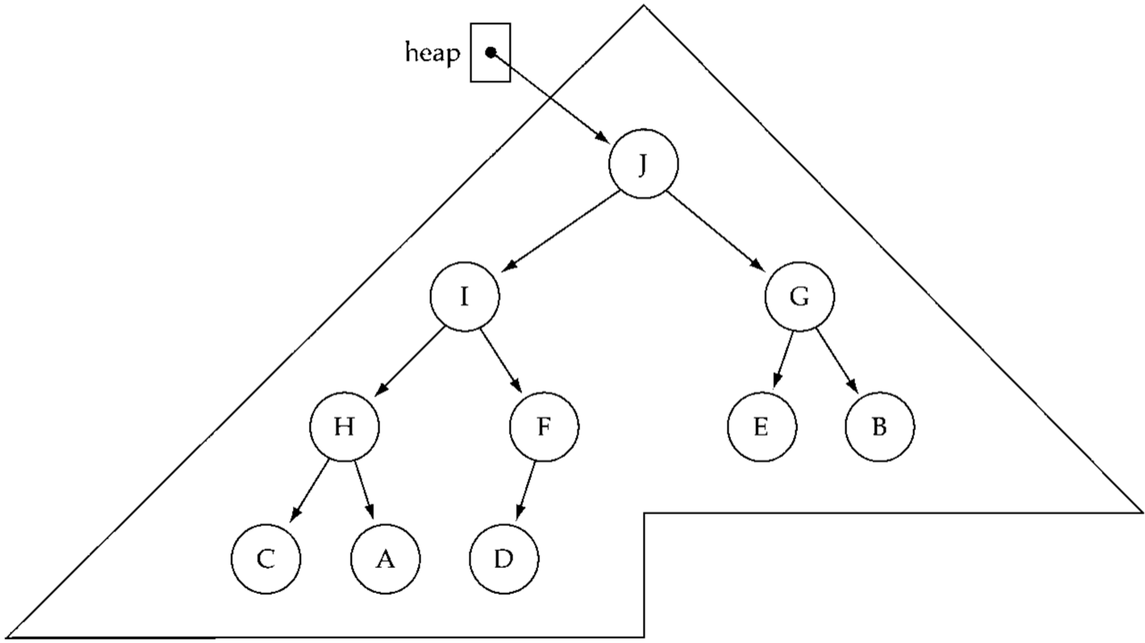


Is this a Heap?

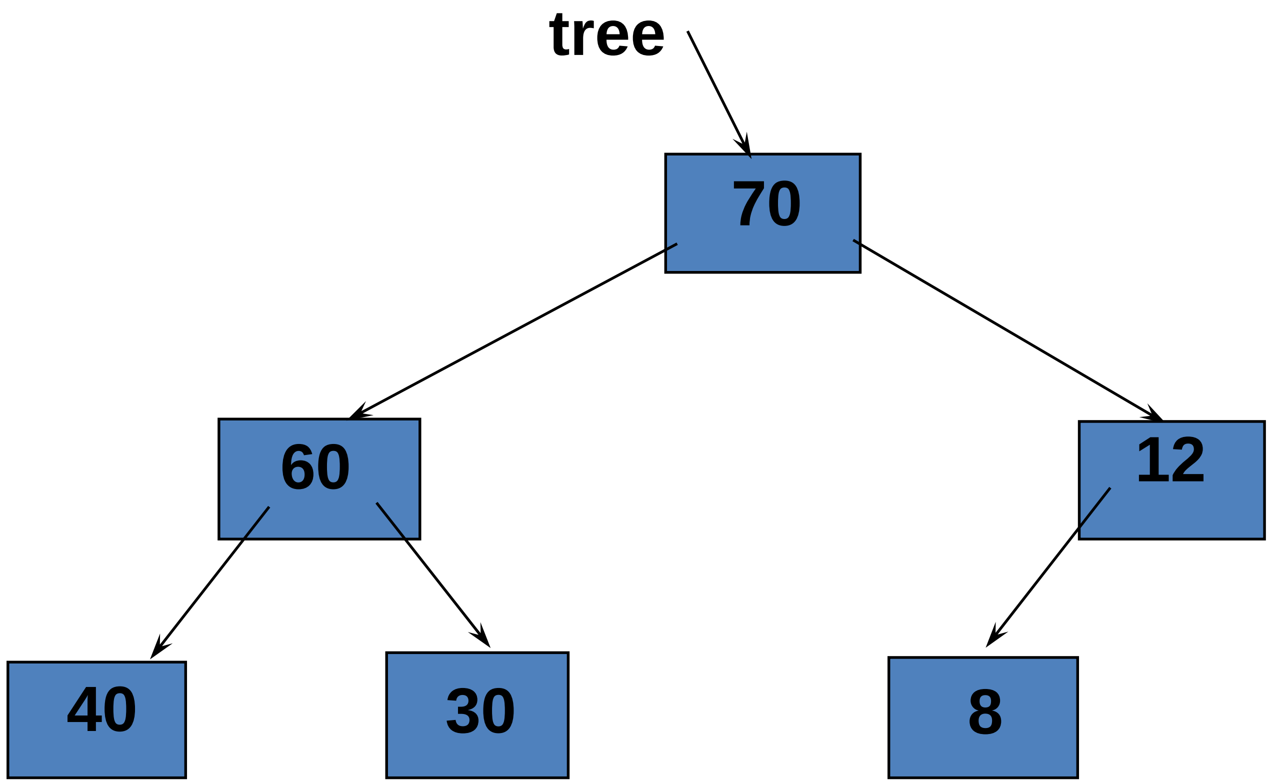


Unique?

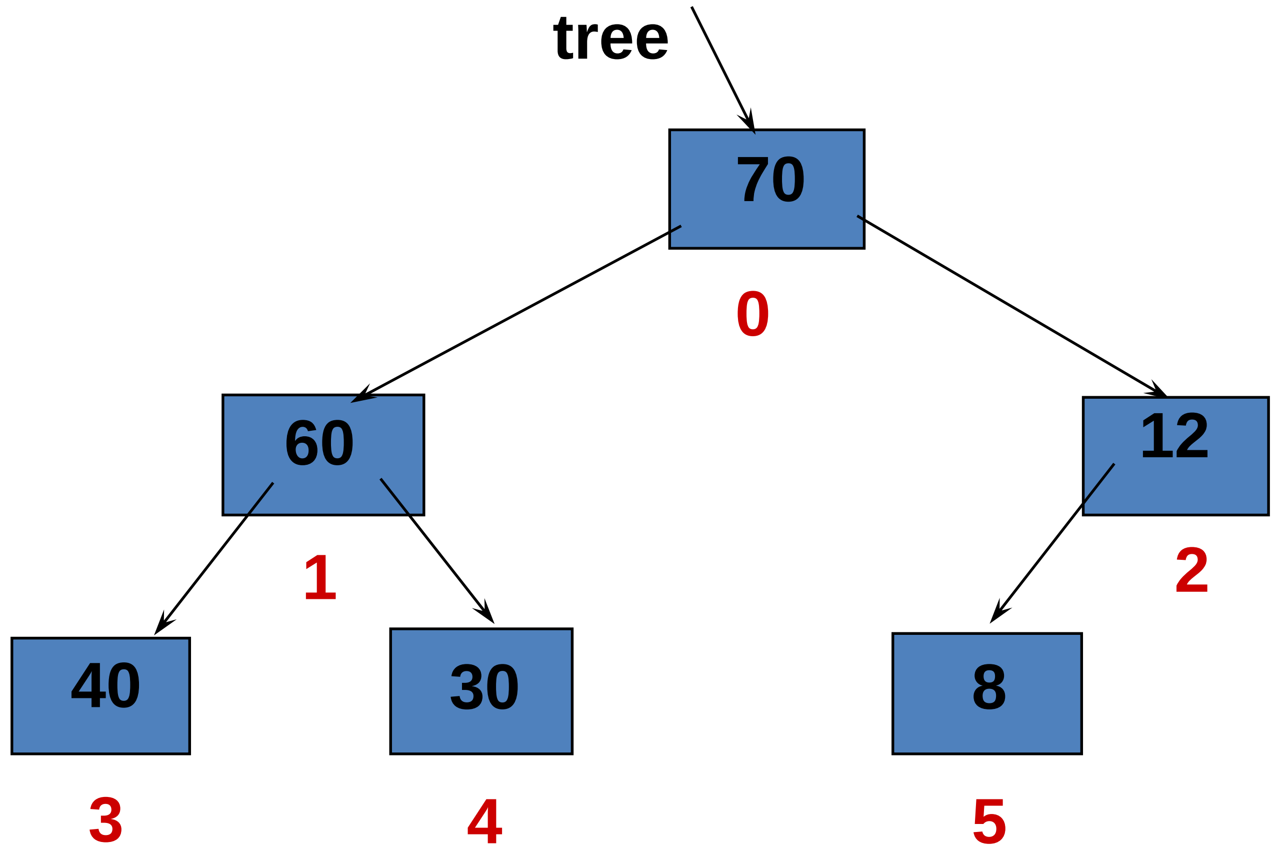




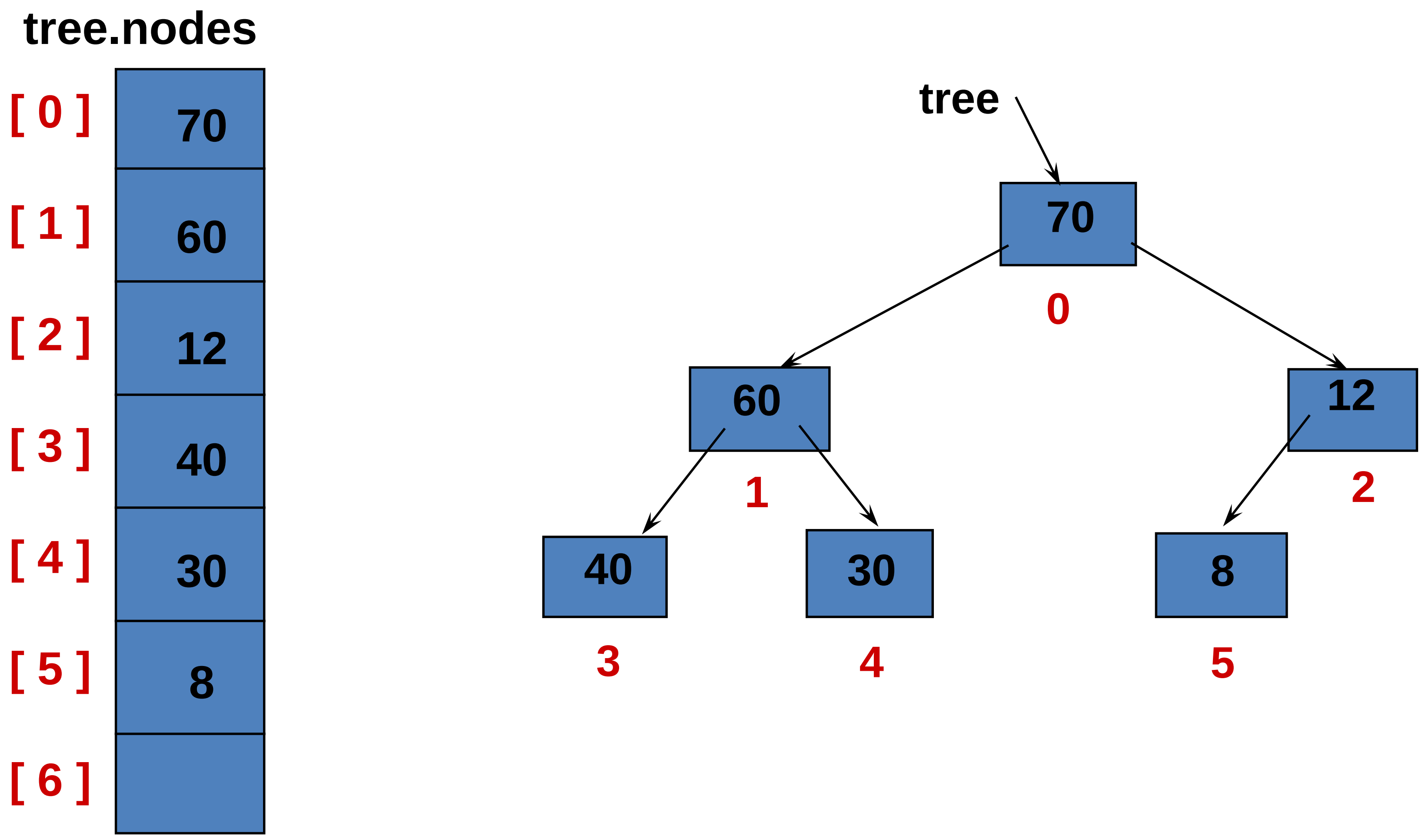
Where is the Largest Element in a Heap Always Found?



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

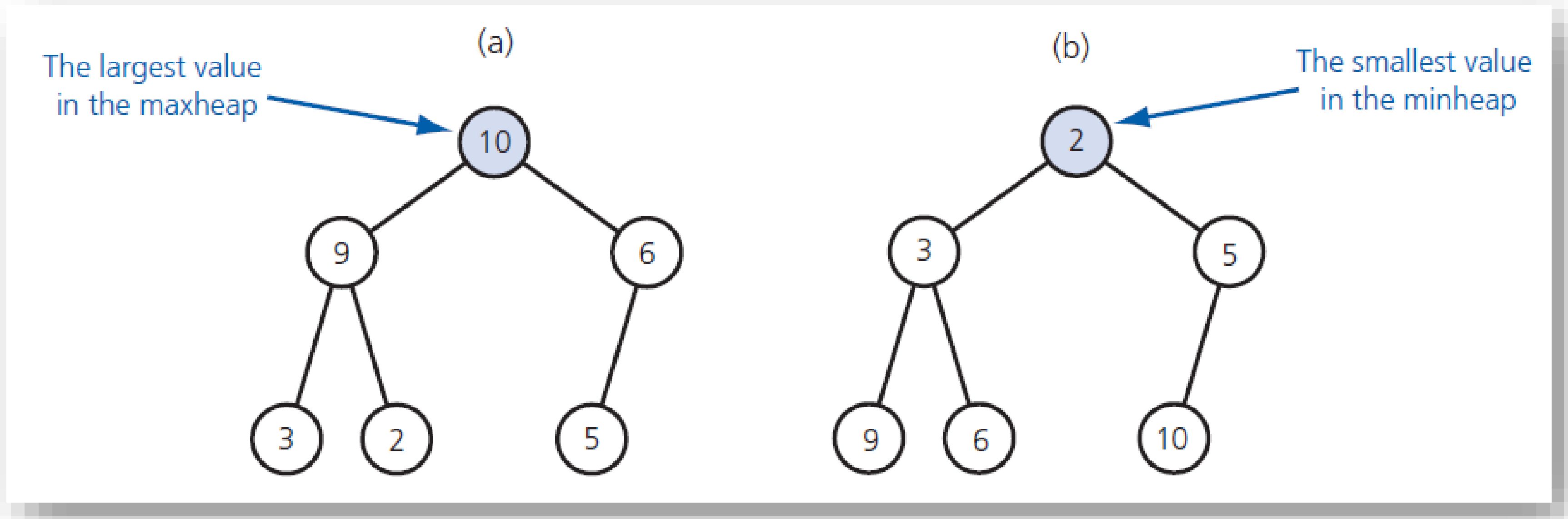


And use the Numbers as Array Indexes to Store the Heap



The ADT Heap

- (a) A maxheap and (b) a minheap


(a)
The largest value in the maxheap
10
9
6
3
2
5
(b)
The smallest value in the minheap
2
3
5
9
6
10

With Array Representation

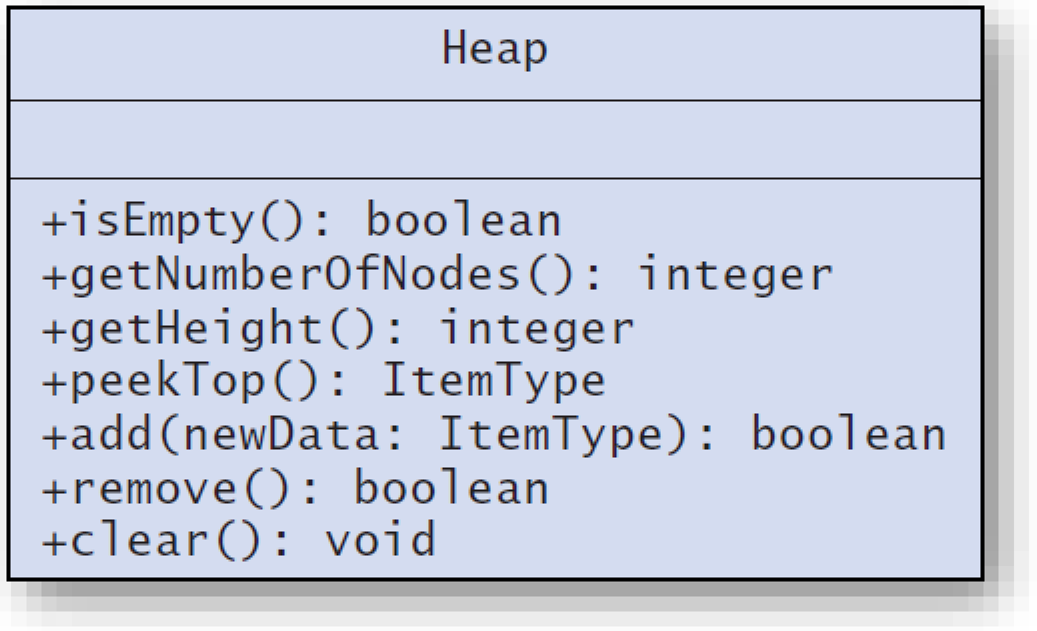
- For any node **`tree.nodes[index]`**
  - its left child is in
    - **`tree.nodes[index*2 + 1]`**
  - right child is in
    - **`tree.nodes[index*2 + 2]`**
  - its parent is in
    - **`tree.nodes[(index – 1)/2]`**

Leaf nodes:

```
    tree.nodes[numElements/2]     to
    tree.nodes[numElements - 1]
```

The ADT Heap

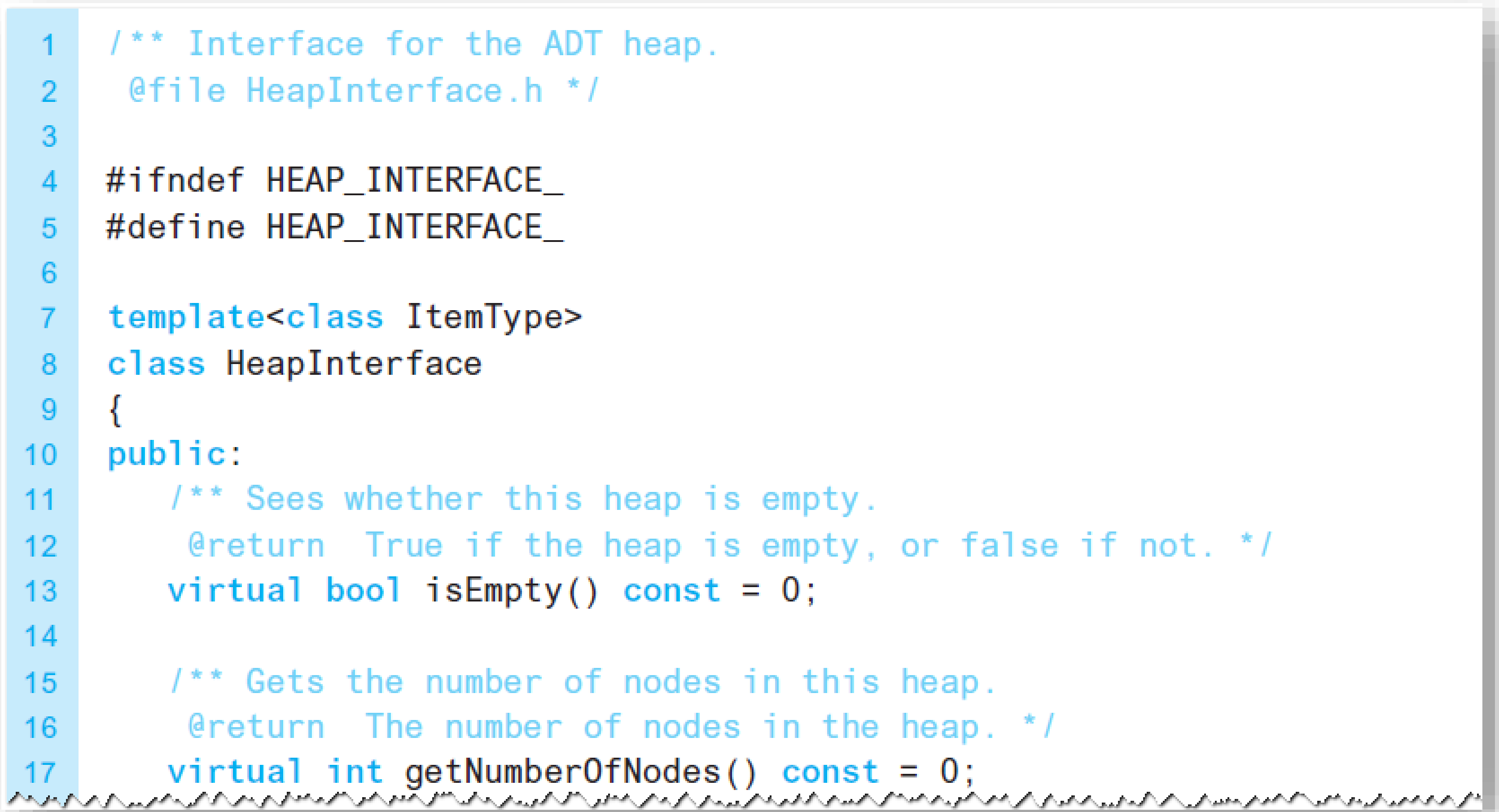
- UML diagram for the class Heap



```
Heap

+isEmpty(): boolean
+getNumberOfNodes(): integer
+getHeight(): integer
+peekTop(): ItemType
+add(newData: ItemType): boolean
+remove(): boolean
+clear(): void
```

The ADT Heap

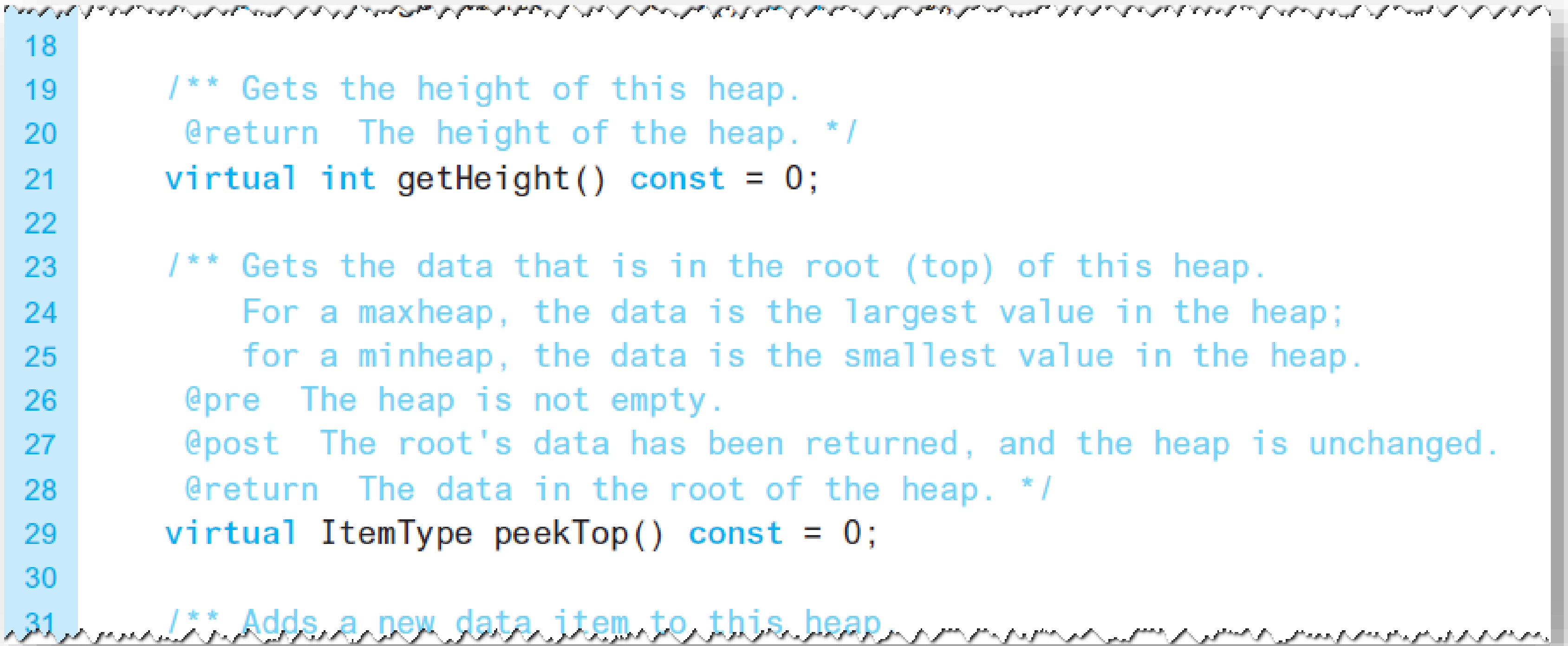
- An interface for the ADT heap



```cpp
/** Interface for the ADT heap.
 @file HeapInterface.h */

#ifndef HEAP_INTERFACE_
#define HEAP_INTERFACE_

template<class ItemType>
class HeapInterface
{
public:
   /** Sees whether this heap is empty.
    @return  True if the heap is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Gets the number of nodes in this heap.
    @return  The number of nodes in the heap. */
   virtual int getNumberOfNodes() const = 0;
```

The ADT Heap

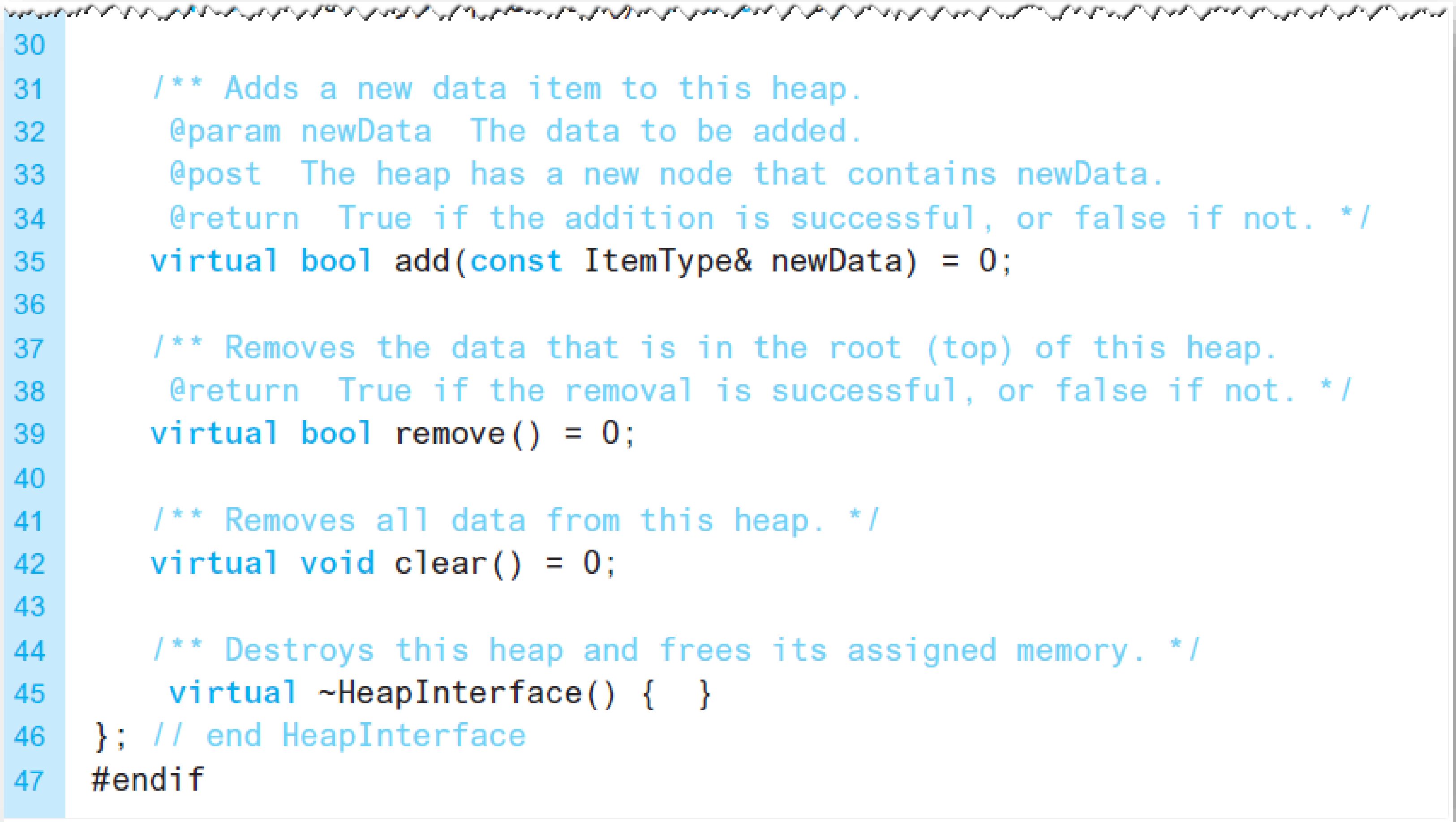
- An interface for the ADT heap



```

   /** Gets the height of this heap.
    @return  The height of the heap. */
   virtual int getHeight() const = 0;

   /** Gets the data that is in the root (top) of this heap.
       For a maxheap, the data is the largest value in the heap;
       for a minheap, the data is the smallest value in the heap.
    @pre  The heap is not empty.
    @post  The root's data has been returned, and the heap is unchanged.
    @return  The data in the root of the heap. */
   virtual ItemType peekTop() const = 0;

   /** Adds a new data item to this heap.
```

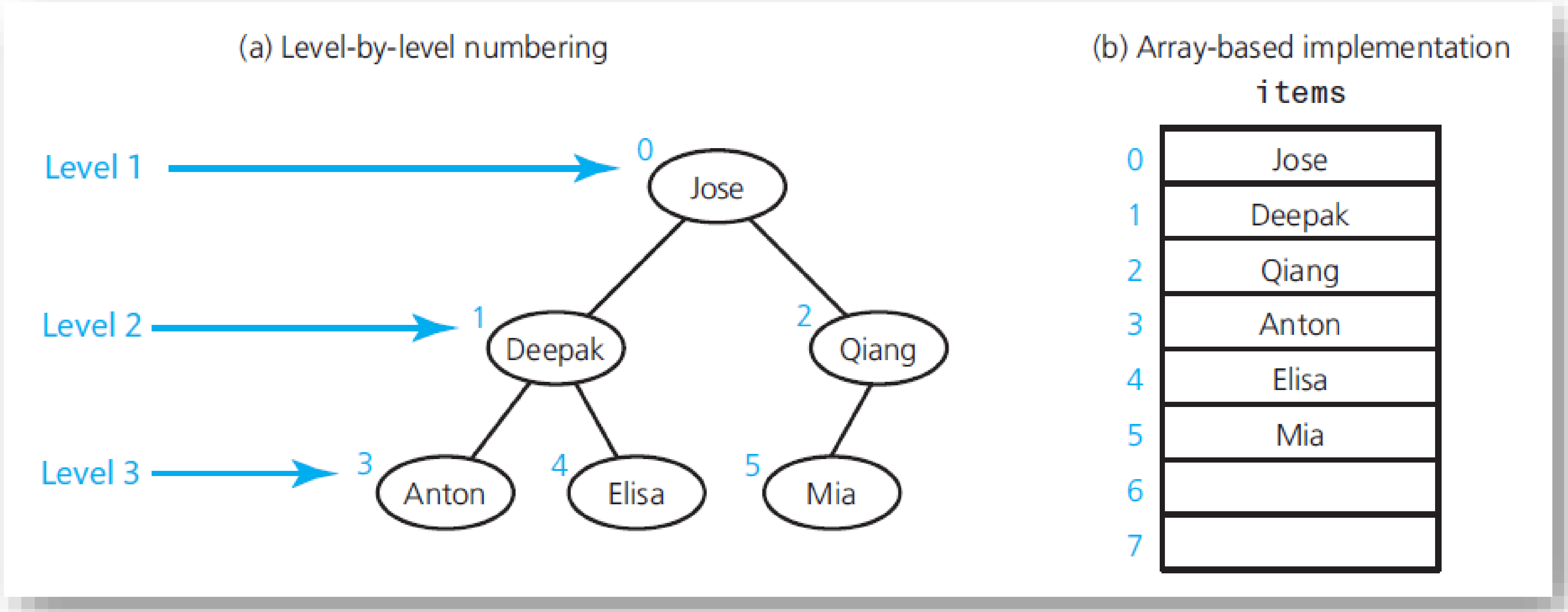
The ADT Heap

- An interface for the ADT heap



```

   /** Adds a new data item to this heap.
    @param newData  The data to be added.
    @post  The heap has a new node that contains newData.
    @return  True if the addition is successful, or false if not. */
   virtual bool add(const ItemType& newData) = 0;

   /** Removes the data that is in the root (top) of this heap.
    @return  True if the removal is successful, or false if not. */
   virtual bool remove() = 0;

   /** Removes all data from this heap. */
   virtual void clear() = 0;

   /** Destroys this heap and frees its assigned memory. */
    virtual ~HeapInterface() {  }
}; // end HeapInterface
#endif
```

Array-Based Implementation of a Heap



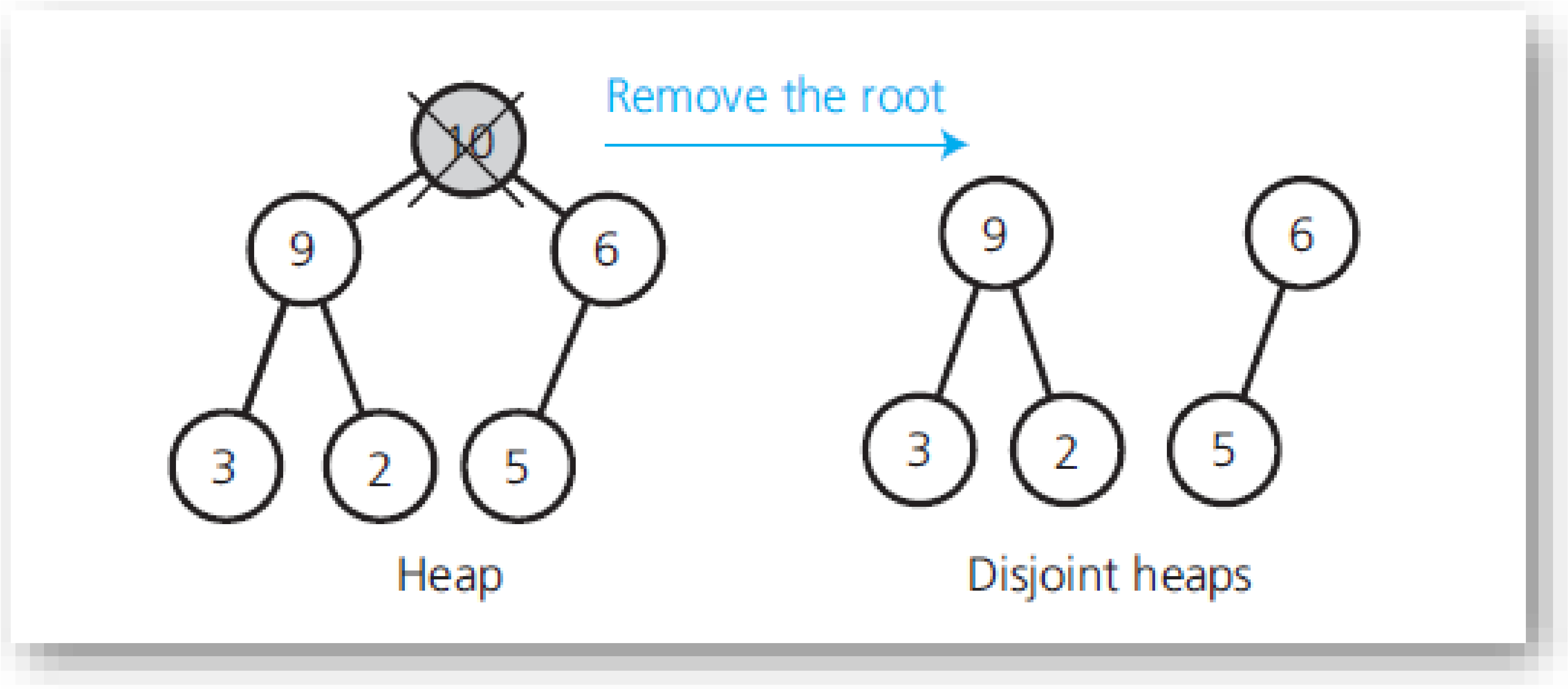
- A complete binary tree and its array-based implementation

Algorithms for Array-Based Heap Operations

- Assume following private data members
  - items: an array of heap items
  - itemCount: an integer equal to the number of items in the heap
  - maxItems: an integer equal to the maximum capacity of the heap

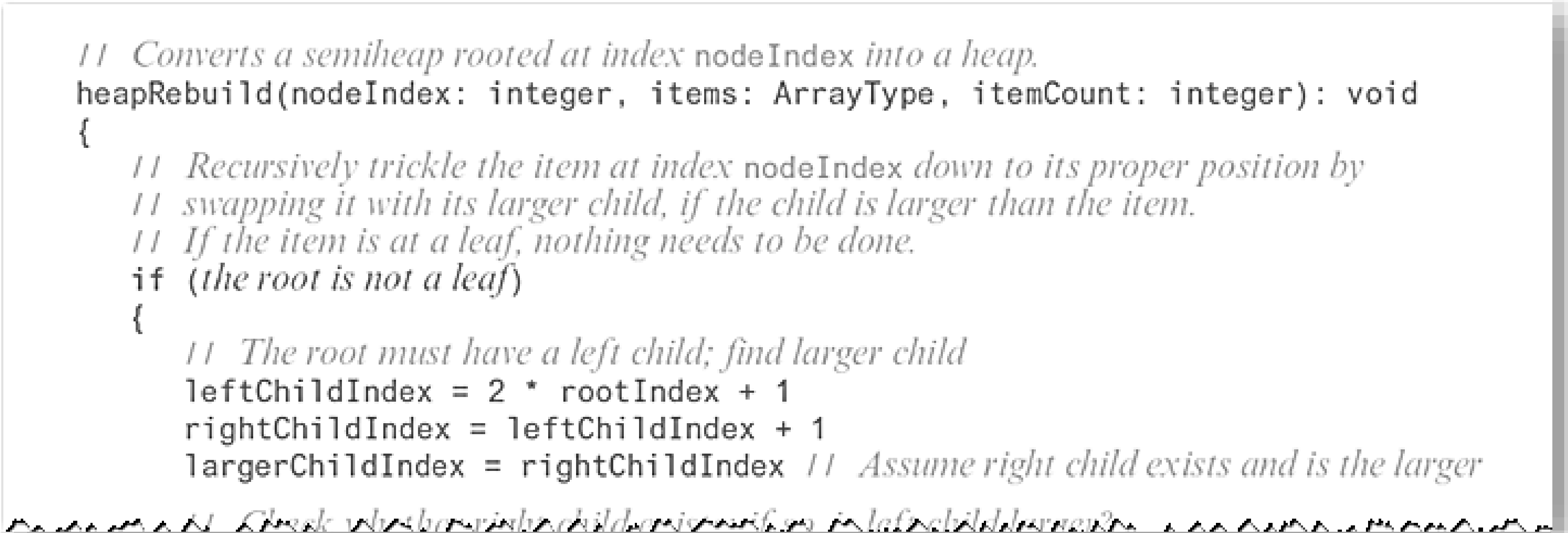
Algorithms for Array-Based Heap Operations

- Disjoint heaps after removing the heap's root



Algorithms for Array-Based Heap Operations

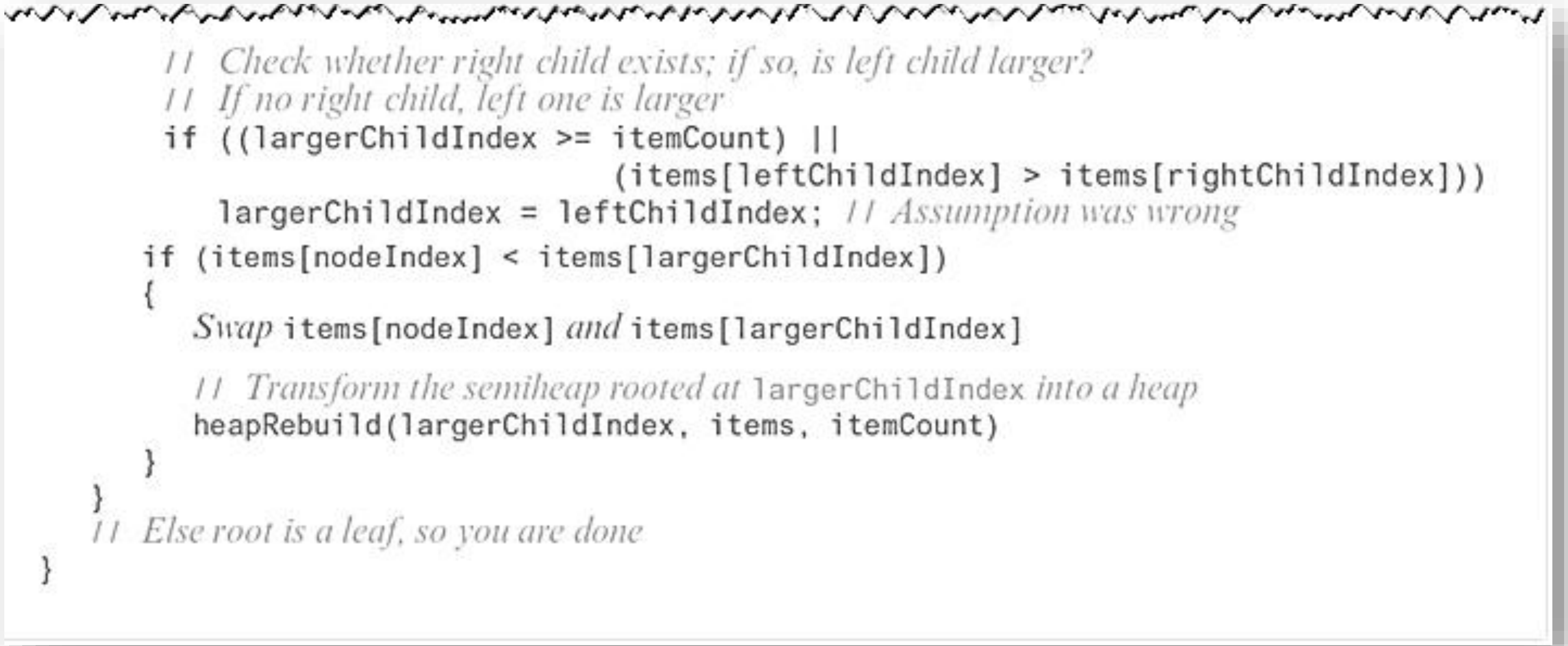
- Recursive algorithm to transform semiheap to heap.



```
// Converts a semiheap rooted at index nodeIndex into a heap.
heapRebuild(nodeIndex: integer, items: ArrayType, itemCount: integer): void
{
   // Recursively trickle the item at index nodeIndex down to its proper position by
   // swapping it with its larger child, if the child is larger than the item.
   // If the item is at a leaf, nothing needs to be done.
   if (the root is not a leaf)
   {
      // The root must have a left child; find larger child
      leftChildIndex = 2 * rootIndex + 1
      rightChildIndex = leftChildIndex + 1
      largerChildIndex = rightChildIndex // Assume right child exists and is the larger

      // Check whether right child exists; if so, is left child larger?
```

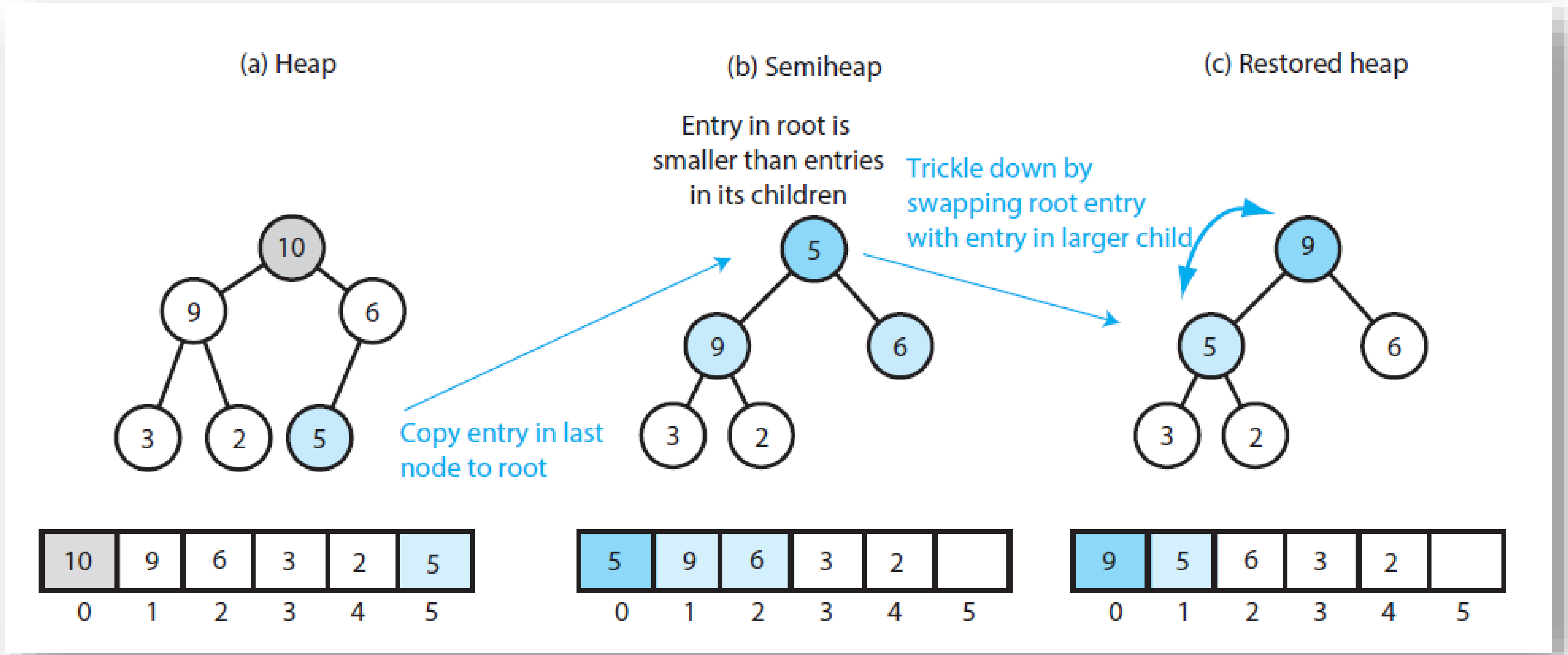
Algorithms for Array-Based Heap Operations

- Recursive algorithm to transform semiheap to heap.

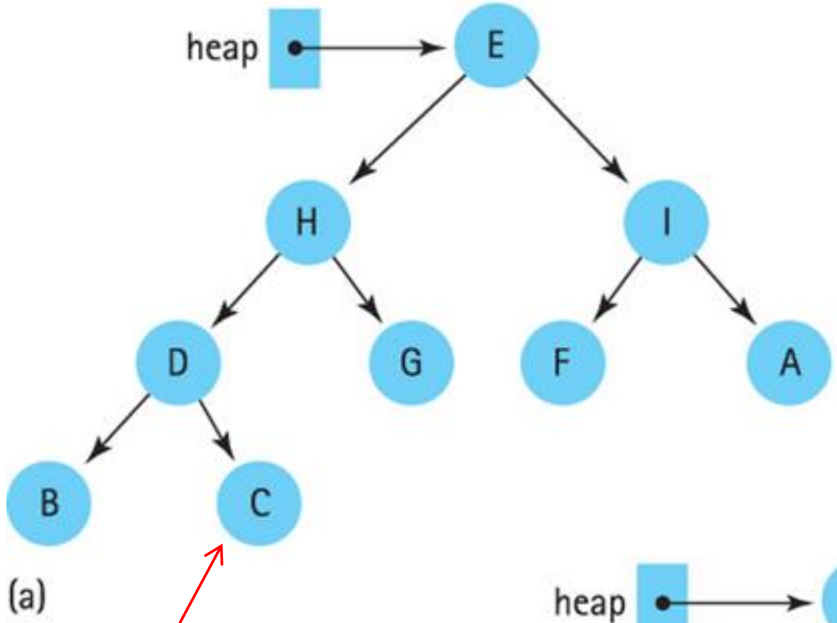


```
          // Check whether right child exists; if so, is left child larger?
          // If no right child, left one is larger
          if ((largerChildIndex >= itemCount) ||
                                   (items[leftChildIndex] > items[rightChildIndex]))
             largerChildIndex = leftChildIndex; // Assumption was wrong
         if (items[nodeIndex] < items[largerChildIndex])
         {
            Swap items[nodeIndex] and items[largerChildIndex]

            // Transform the semiheap rooted at largerChildIndex into a heap
            heapRebuild(largerChildIndex, items, itemCount)
         }
      }
      // Else root is a leaf, so you are done
   }
```

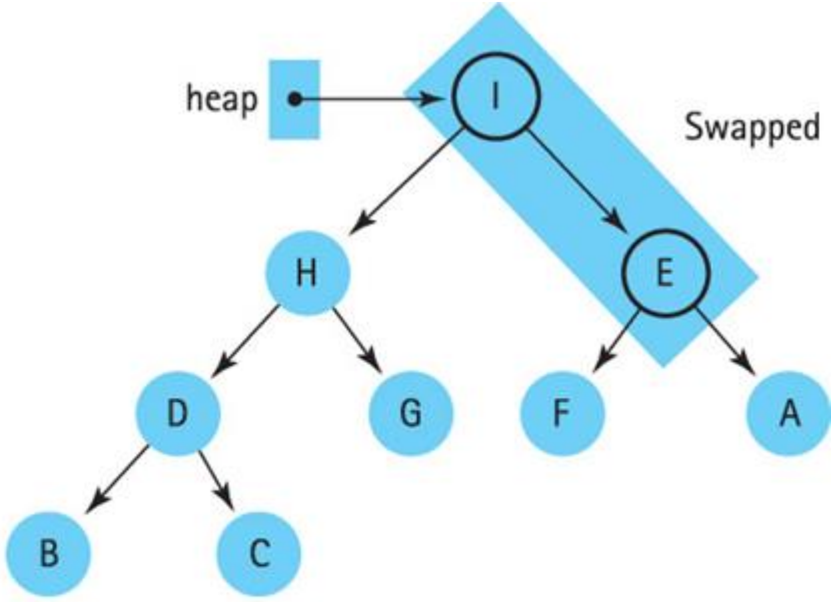
Algorithms for Array-Based Heap Operations

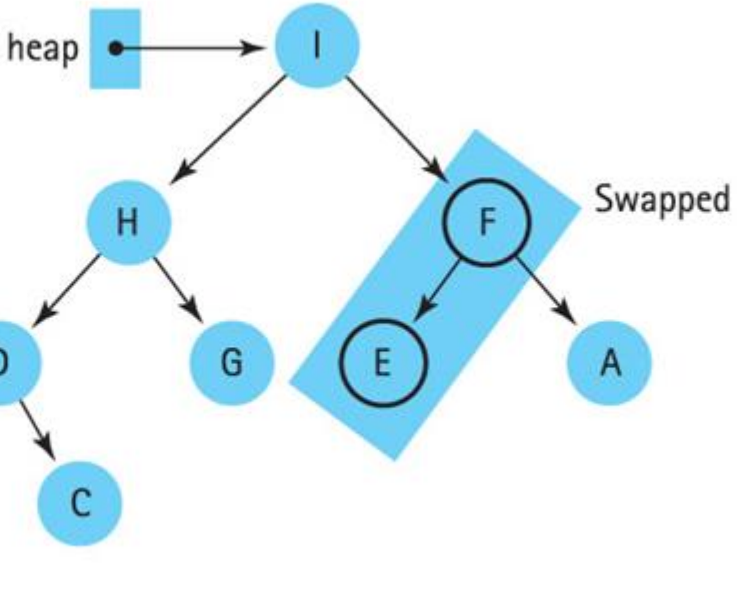

(a) Heap
(b) Semiheap
(c) Restored heap
Entry in root is smaller than entries in its children
Trickle down by swapping root entry with entry in larger child
Copy entry in last node to root
10
9
6
3
2
5
0
1
2
3
4
5

Reheap Down



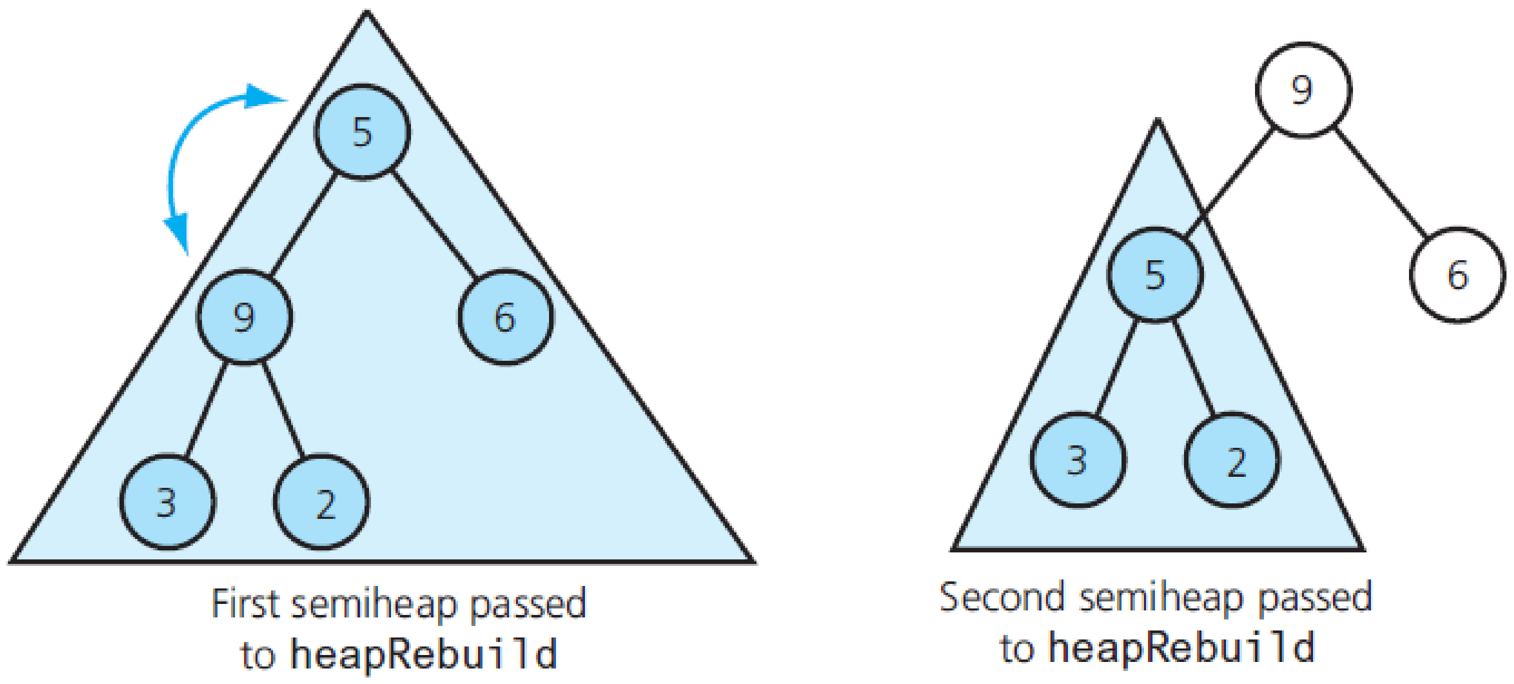
bottom





Algorithms for Array-Based Heap Operations

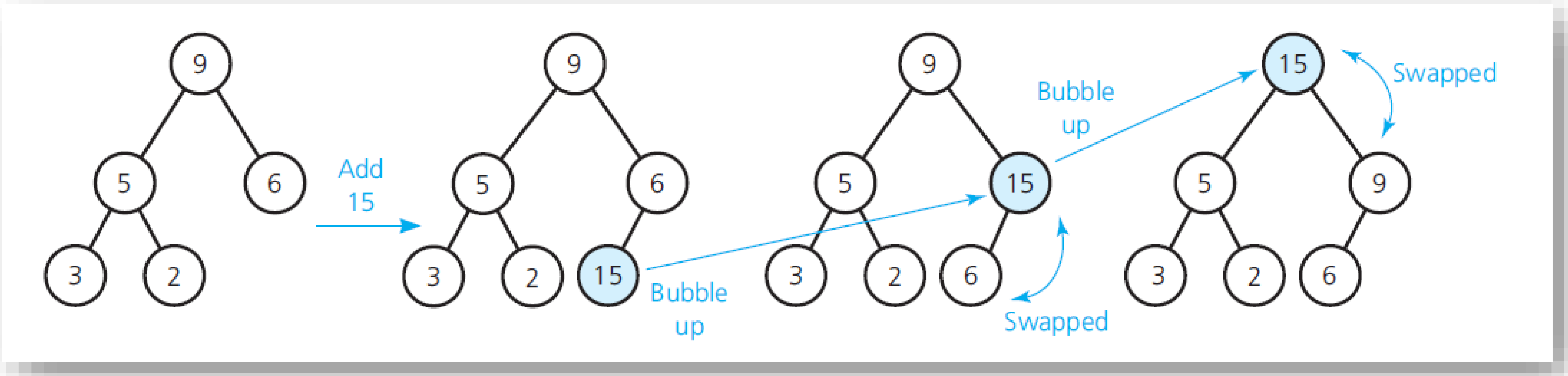
- Recursive calls to heapRebuild



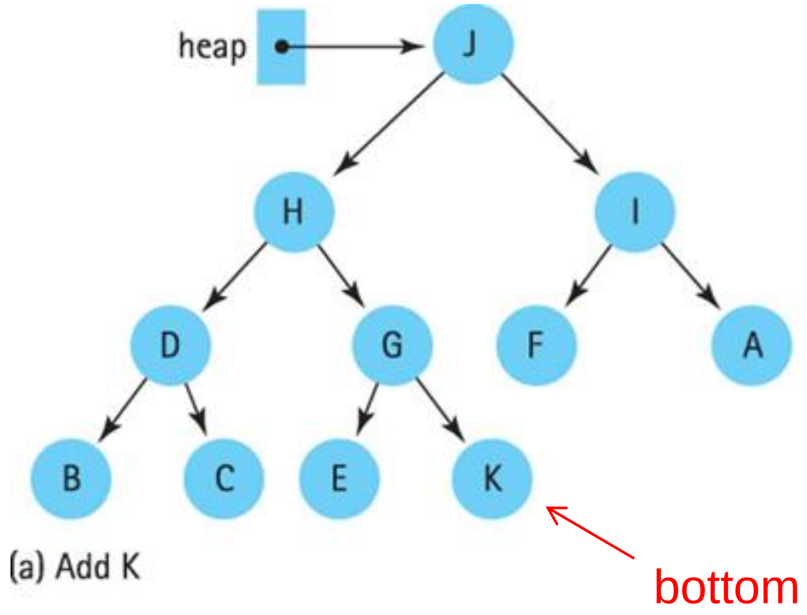
5
9
6
3
2
First semiheap passed to heapRebuild
9
5
6
3
2
Second semiheap passed to heapRebuild

Algorithms for Array-Based Heap Operations

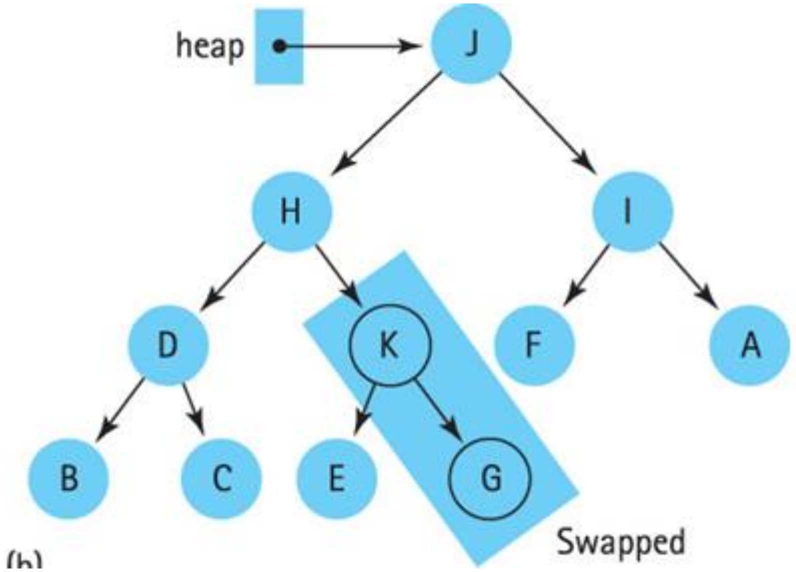
- Adding 15 to a heap



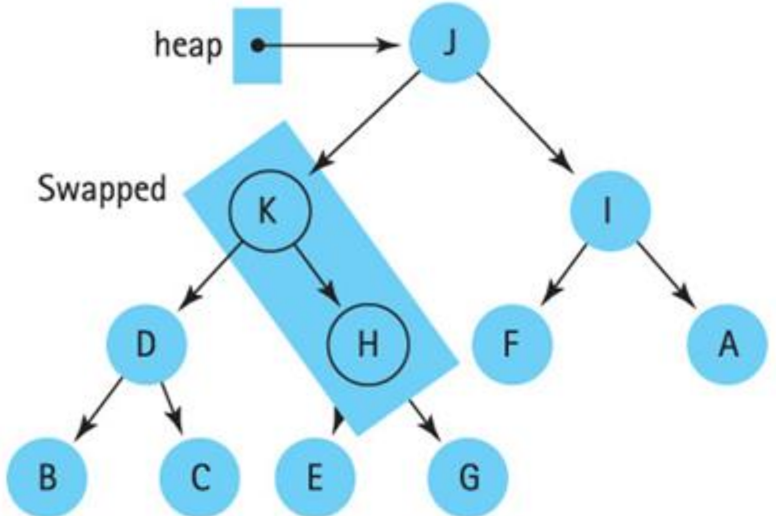
Reheap Up



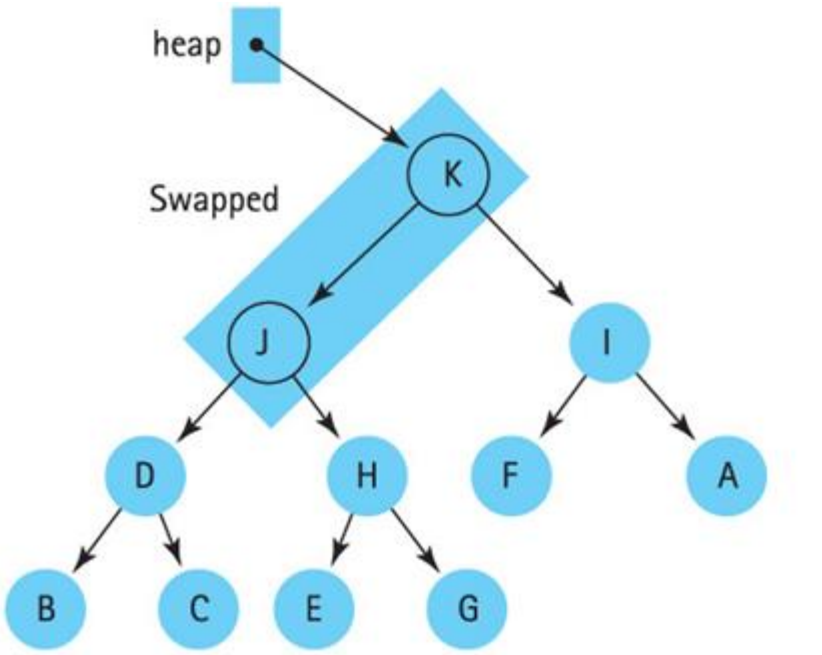
heap
J
H
I
D
G
F
A
B
C
E
K
(a) Add K
bottom



heap
J
H
I
D
K
F
A
B
C
E
G
Swapped

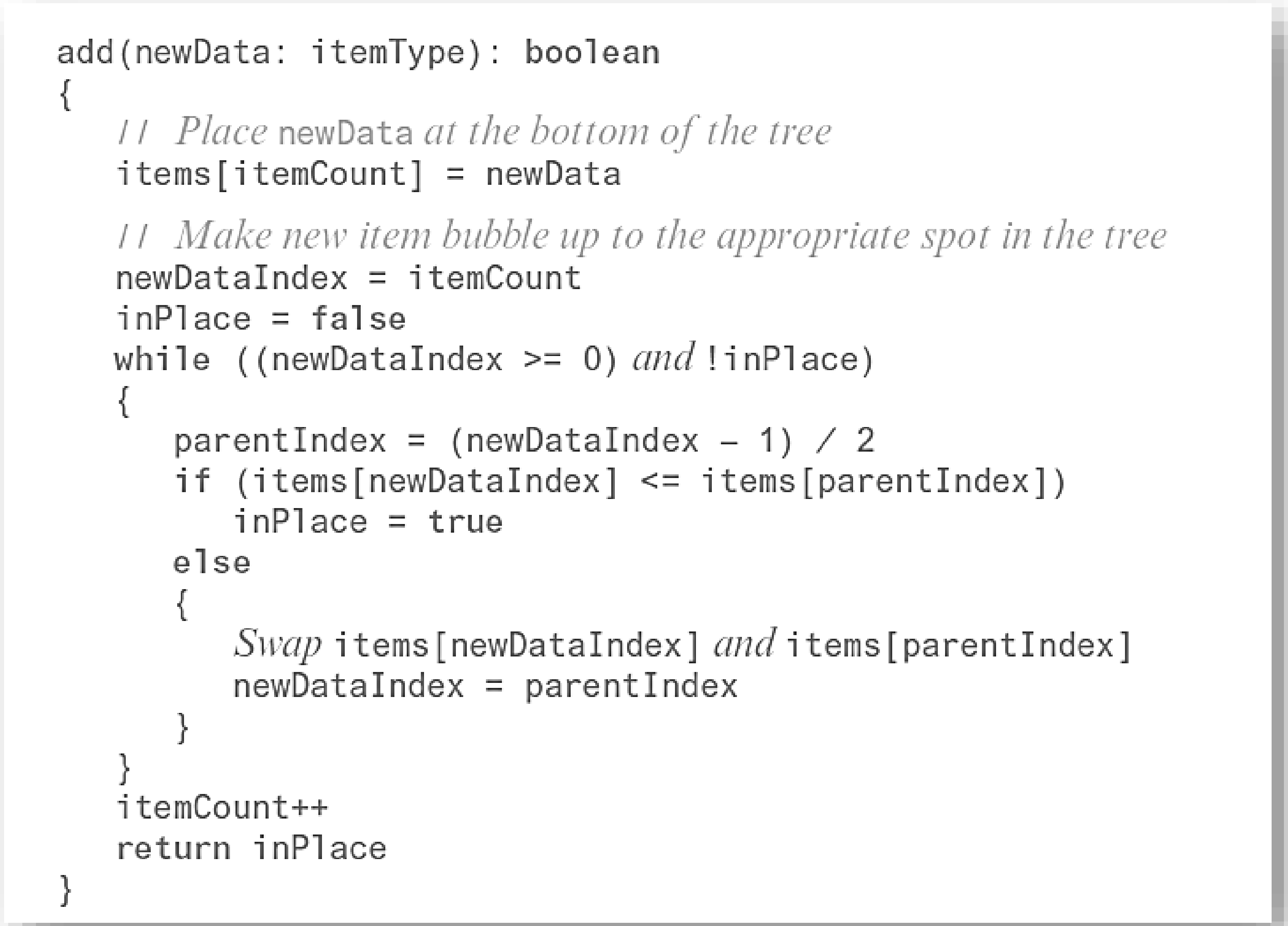


heap
J
Swapped
K
I
D
H
F
A
B
C
E
G



heap
K
Swapped
J
I
D
H
F
A
B
C
E
G

Algorithms for Array-Based Heap Operations

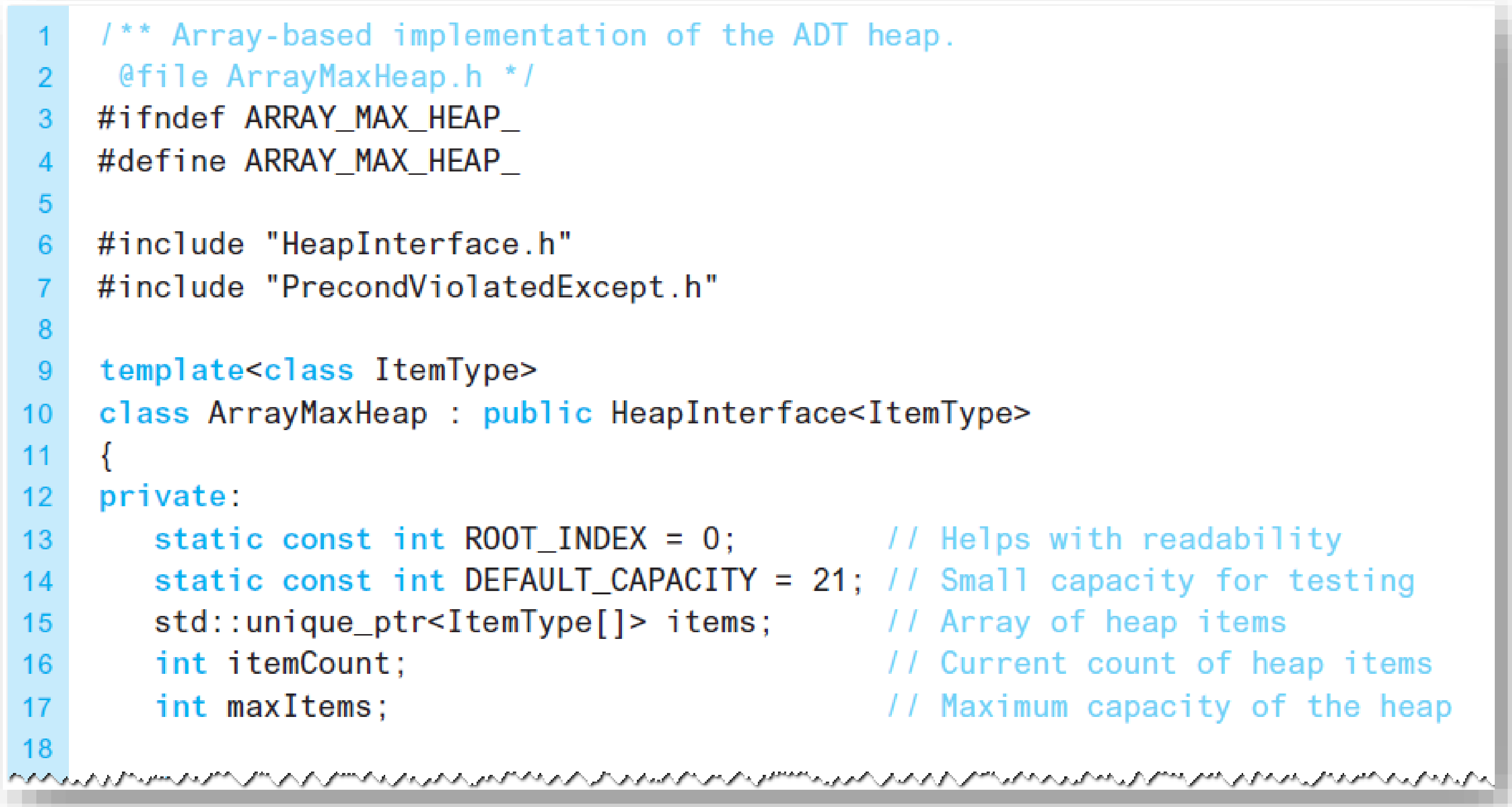


```
add(newData: itemType): boolean
{
   // Place newData at the bottom of the tree
   items[itemCount] = newData

   // Make new item bubble up to the appropriate spot in the tree
   newDataIndex = itemCount
   inPlace = false
   while ((newDataIndex >= 0) and !inPlace)
   {
      parentIndex = (newDataIndex - 1) / 2
      if (items[newDataIndex] <= items[parentIndex])
         inPlace = true
      else
      {
         Swap items[newDataIndex] and items[parentIndex]
         newDataIndex = parentIndex
      }
   }
   itemCount++
   return inPlace
}
```

- Pseudocode for add

The Implementation

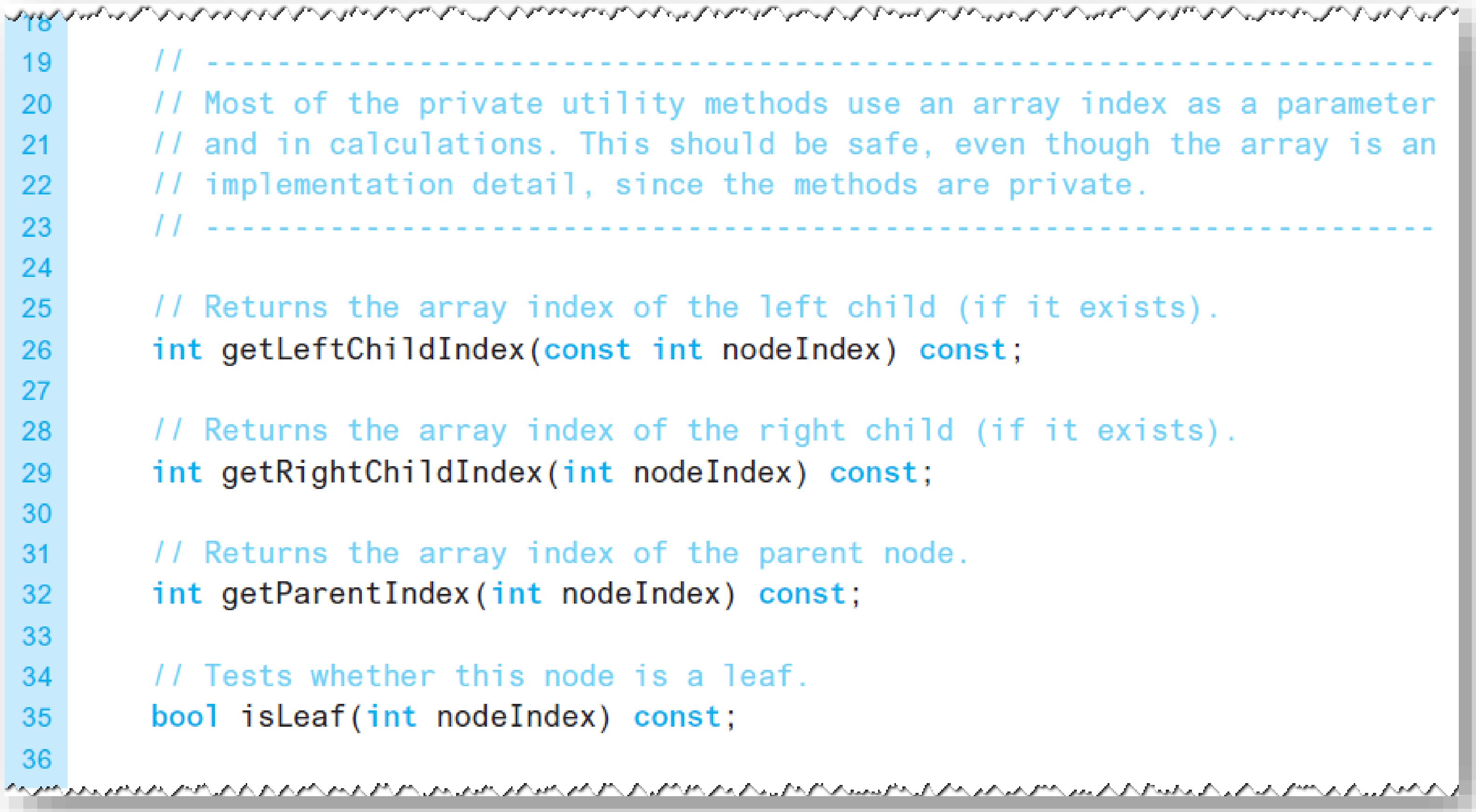
- The header file for the class ArrayMaxHeap



```
/** Array-based implementation of the ADT heap.
 @file ArrayMaxHeap.h */
#ifndef ARRAY_MAX_HEAP_
#define ARRAY_MAX_HEAP_

#include "HeapInterface.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class ArrayMaxHeap : public HeapInterface<ItemType>
{
private:
   static const int ROOT_INDEX = 0;          // Helps with readability
   static const int DEFAULT_CAPACITY = 21;   // Small capacity for testing
   std::unique_ptr<ItemType[]> items;        // Array of heap items
   int itemCount;                            // Current count of heap items
   int maxItems;                             // Maximum capacity of the heap

```

The Implementation

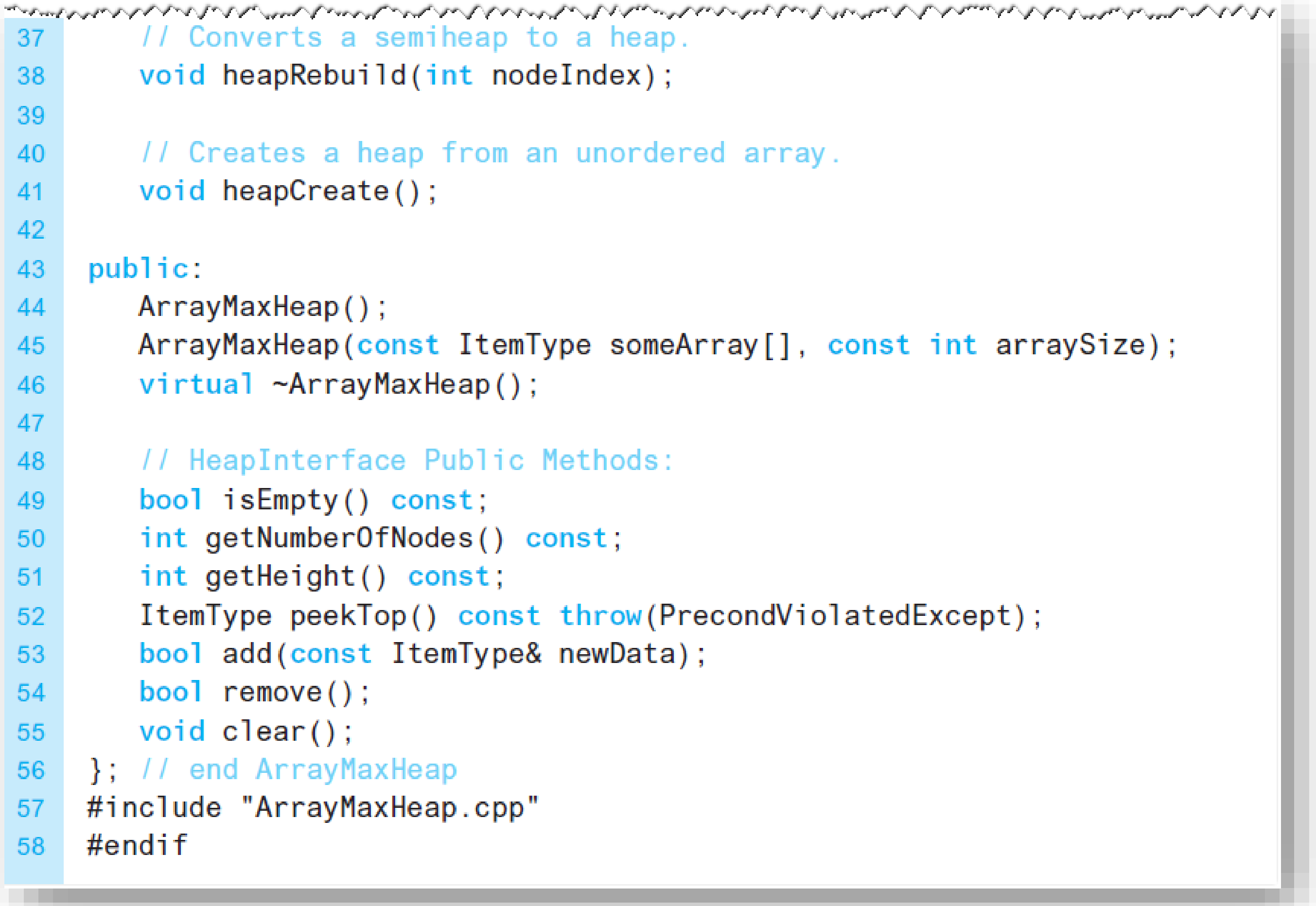
- The header file for the class ArrayMaxHeap



```
   // -------------------------------------------------------------------------
   // Most of the private utility methods use an array index as a parameter
   // and in calculations. This should be safe, even though the array is an
   // implementation detail, since the methods are private.
   // -------------------------------------------------------------------------

   // Returns the array index of the left child (if it exists).
   int getLeftChildIndex(const int nodeIndex) const;

   // Returns the array index of the right child (if it exists).
   int getRightChildIndex(int nodeIndex) const;

   // Returns the array index of the parent node.
   int getParentIndex(int nodeIndex) const;

   // Tests whether this node is a leaf.
   bool isLeaf(int nodeIndex) const;

```

The Implementation

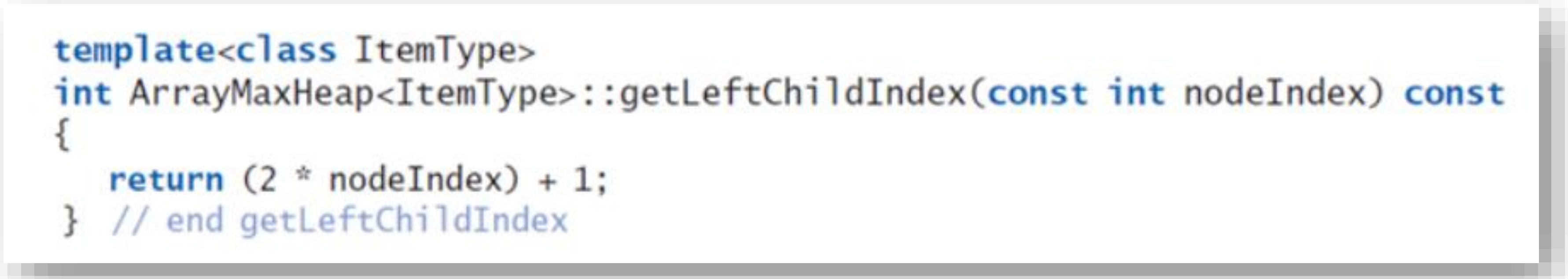
- The header file for the class ArrayMaxHeap



```
   // Converts a semiheap to a heap.
   void heapRebuild(int nodeIndex);

   // Creates a heap from an unordered array.
   void heapCreate();

public:
   ArrayMaxHeap();
   ArrayMaxHeap(const ItemType someArray[], const int arraySize);
   virtual ~ArrayMaxHeap();

   // HeapInterface Public Methods:
   bool isEmpty() const;
   int getNumberOfNodes() const;
   int getHeight() const;
   ItemType peekTop() const throw(PrecondViolatedExcept);
   bool add(const ItemType& newData);
   bool remove();
   void clear();
}; // end ArrayMaxHeap
#include "ArrayMaxHeap.cpp"
#endif
```

The Implementation

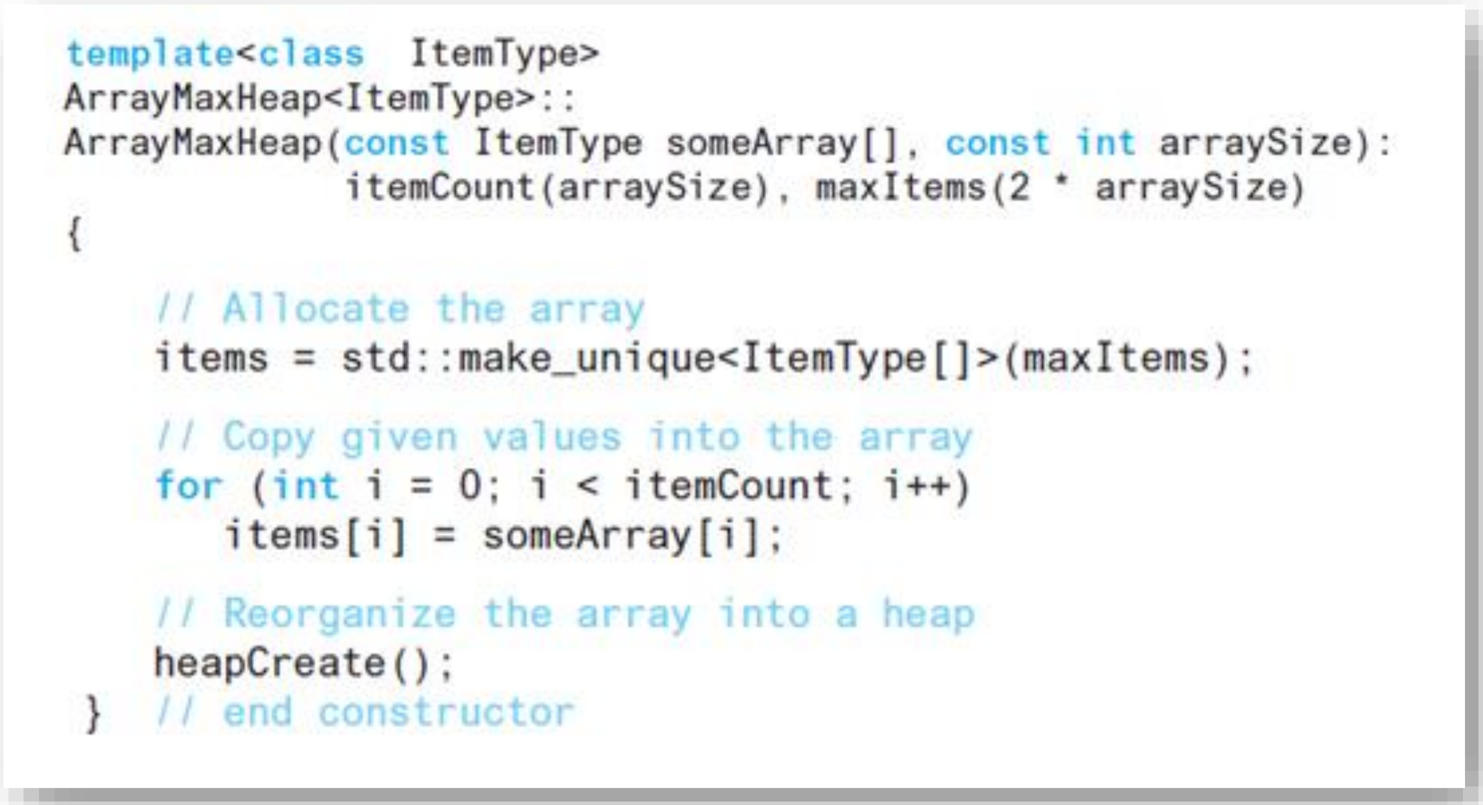
- Definition of method getLeftChildIndex



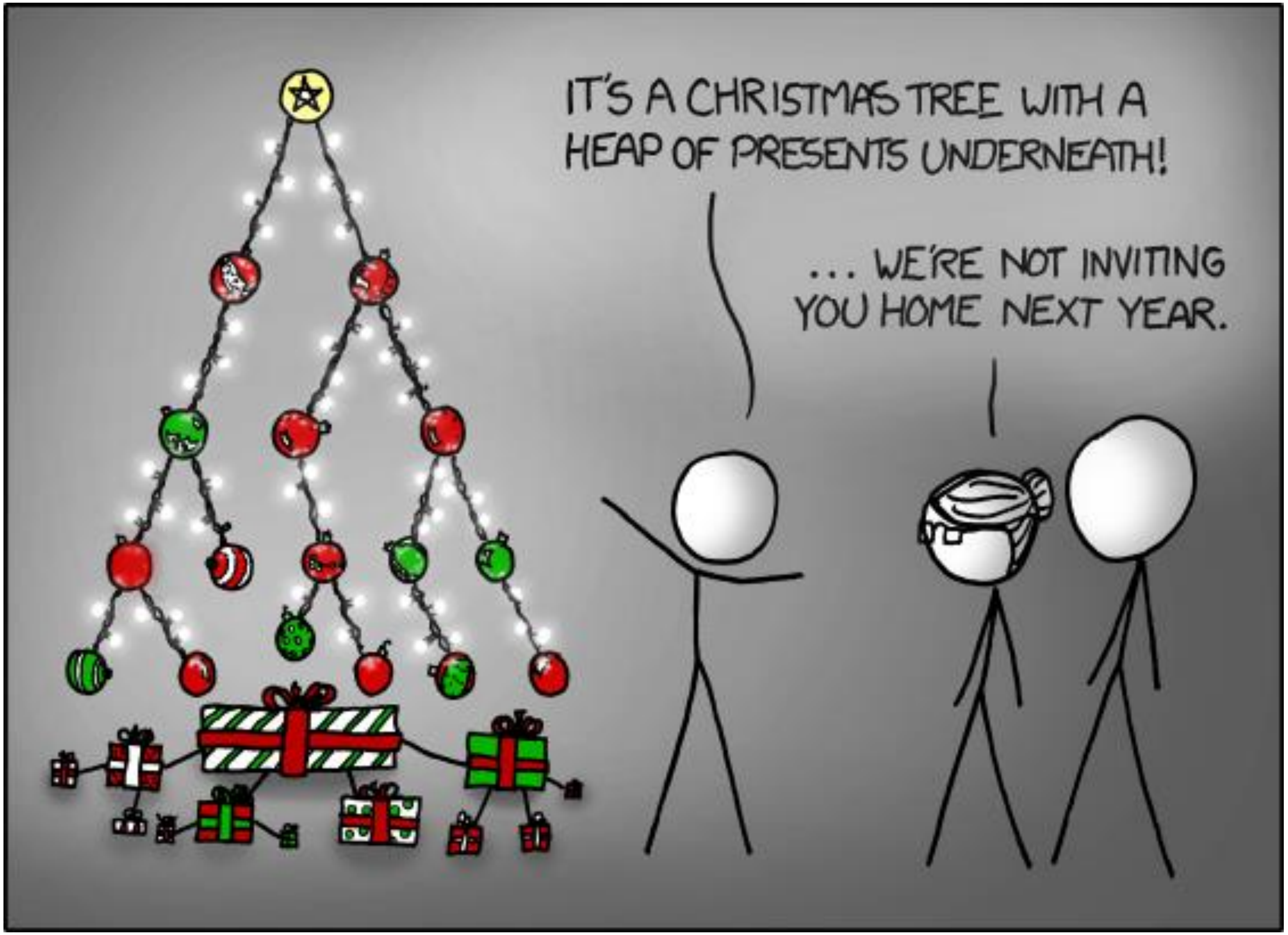
```
template<class ItemType>
int ArrayMaxHeap<ItemType>::getLeftChildIndex(const int nodeIndex) const
{
   return (2 * nodeIndex) + 1;
}  // end getLeftChildIndex
```

The Implementation

- Definition of the constructor

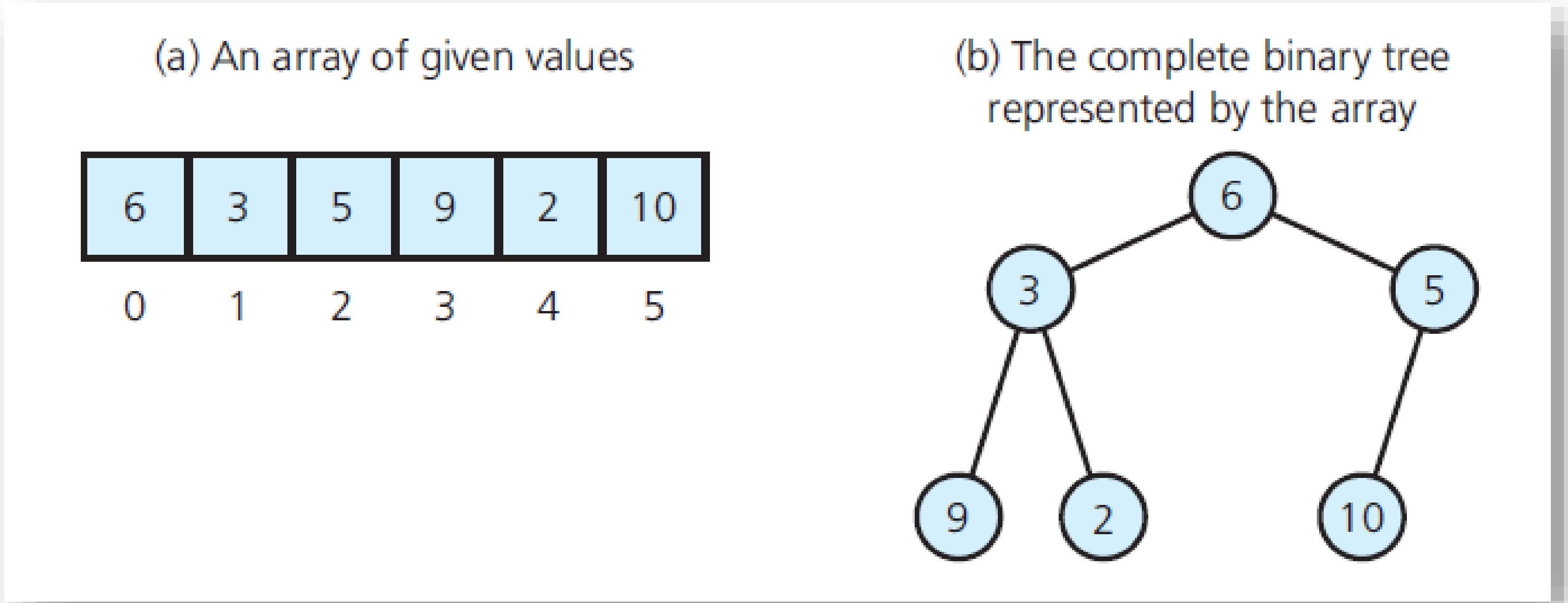


```
template<class  ItemType>
ArrayMaxHeap<ItemType>::
ArrayMaxHeap(const ItemType someArray[], const int arraySize):
             itemCount(arraySize), maxItems(2 * arraySize)
{

    // Allocate the array
    items = std::make_unique<ItemType[]>(maxItems);

    // Copy given values into the array
    for (int i = 0; i < itemCount; i++)
       items[i] = someArray[i];

    // Reorganize the array into a heap
    heapCreate();
 }  // end constructor
```


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

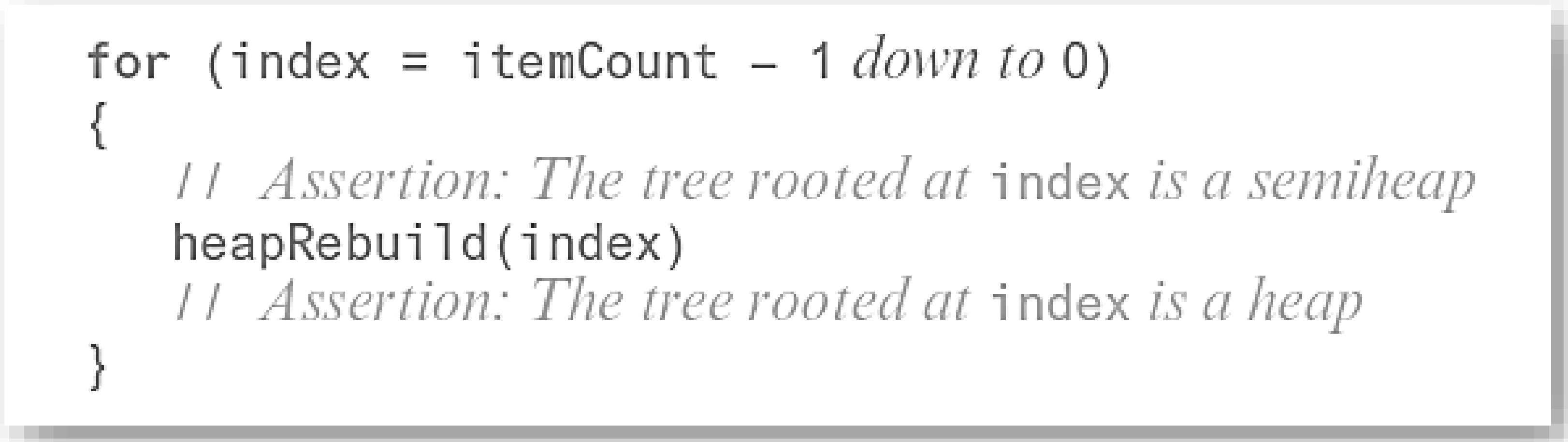
The Implementation

- Array and its corresponding complete binary tree


(a) An array of given values
6
3
5
9
2
10
0
1
2
3
4
5
(b) The complete binary tree represented by the array
6
3
5
9
2
10

The Implementation

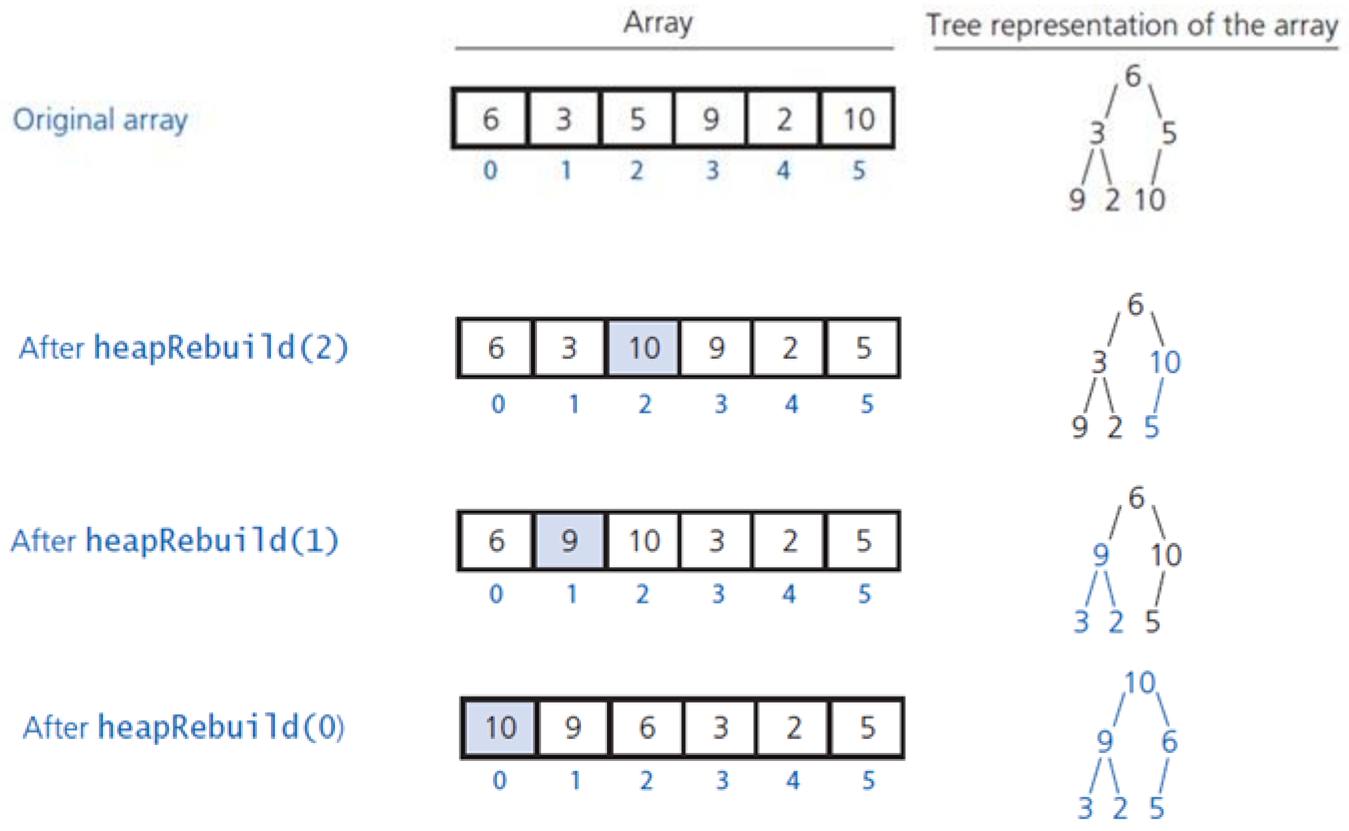
- Building a heap from an array of data



```
for (index = itemCount – 1 down to 0)
{
    // Assertion: The tree rooted at index is a semiheap
    heapRebuild(index)
    // Assertion: The tree rooted at index is a heap
}
```

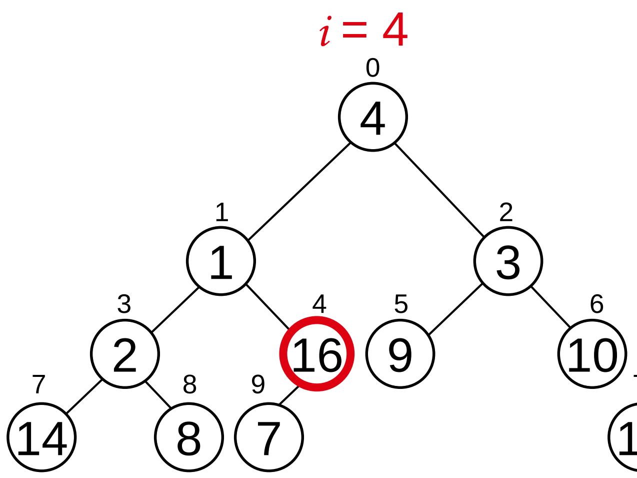
The Implementation

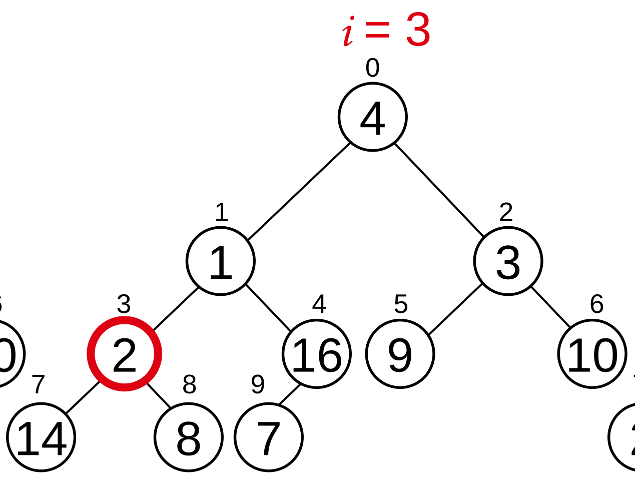
- Transforming an array into a heap

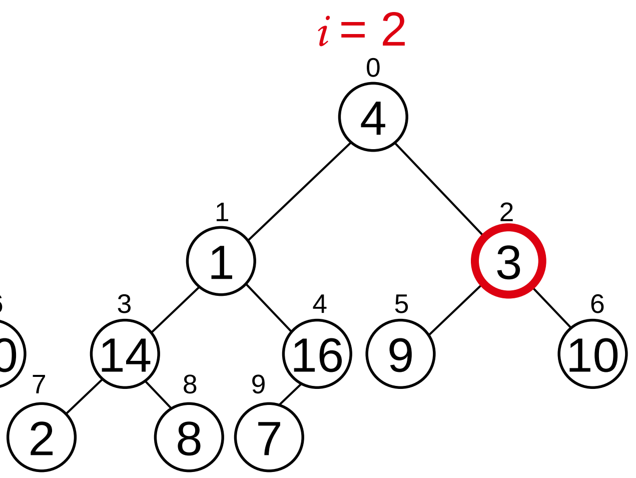


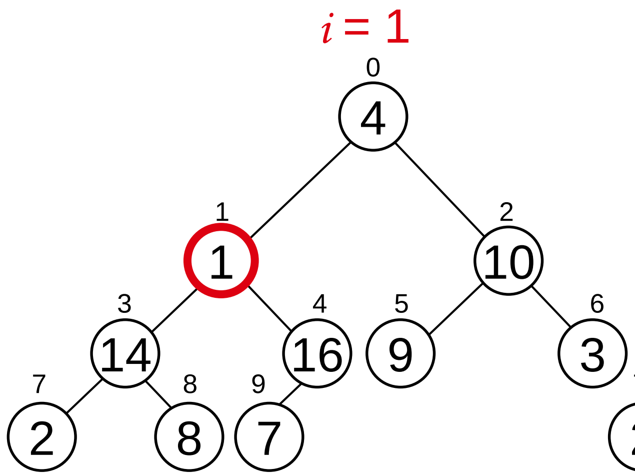
Building a Heap:

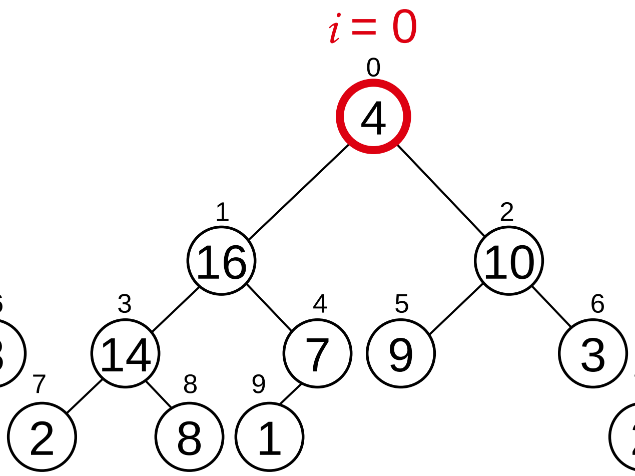
4	1	3	2	16	9	10	14	8	7

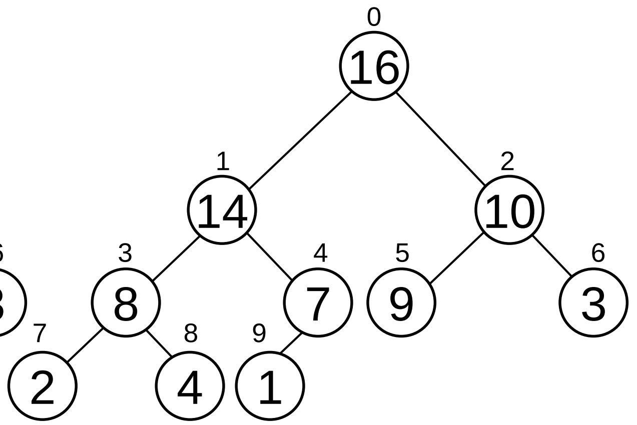






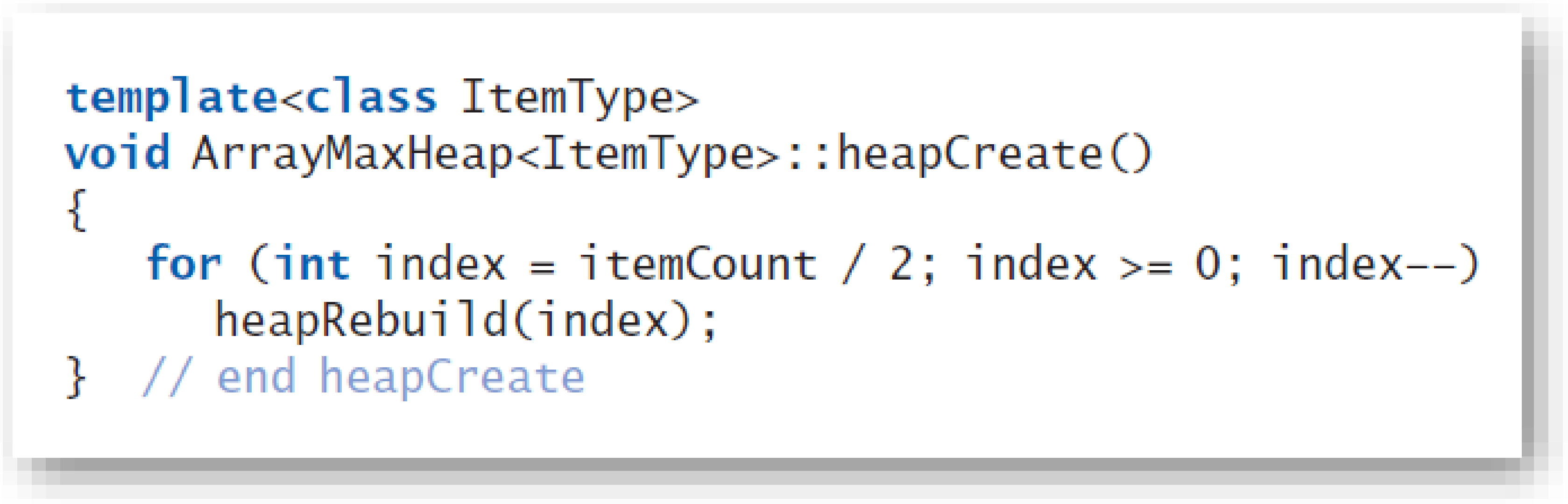






The Implementation

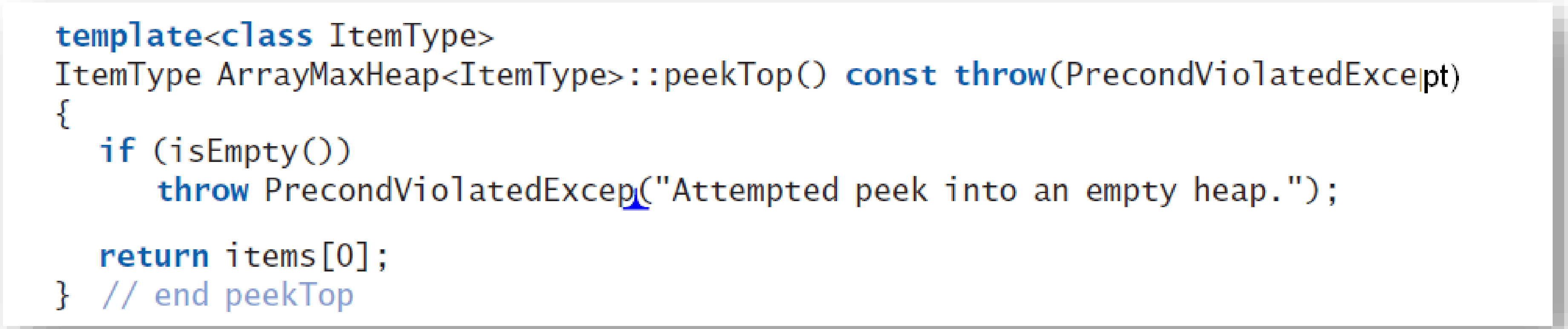
- C++ method heapCreate



```
template<class ItemType>
void ArrayMaxHeap<ItemType>::heapCreate()
{
   for (int index = itemCount / 2; index >= 0; index--)
      heapRebuild(index);
}  // end heapCreate
```

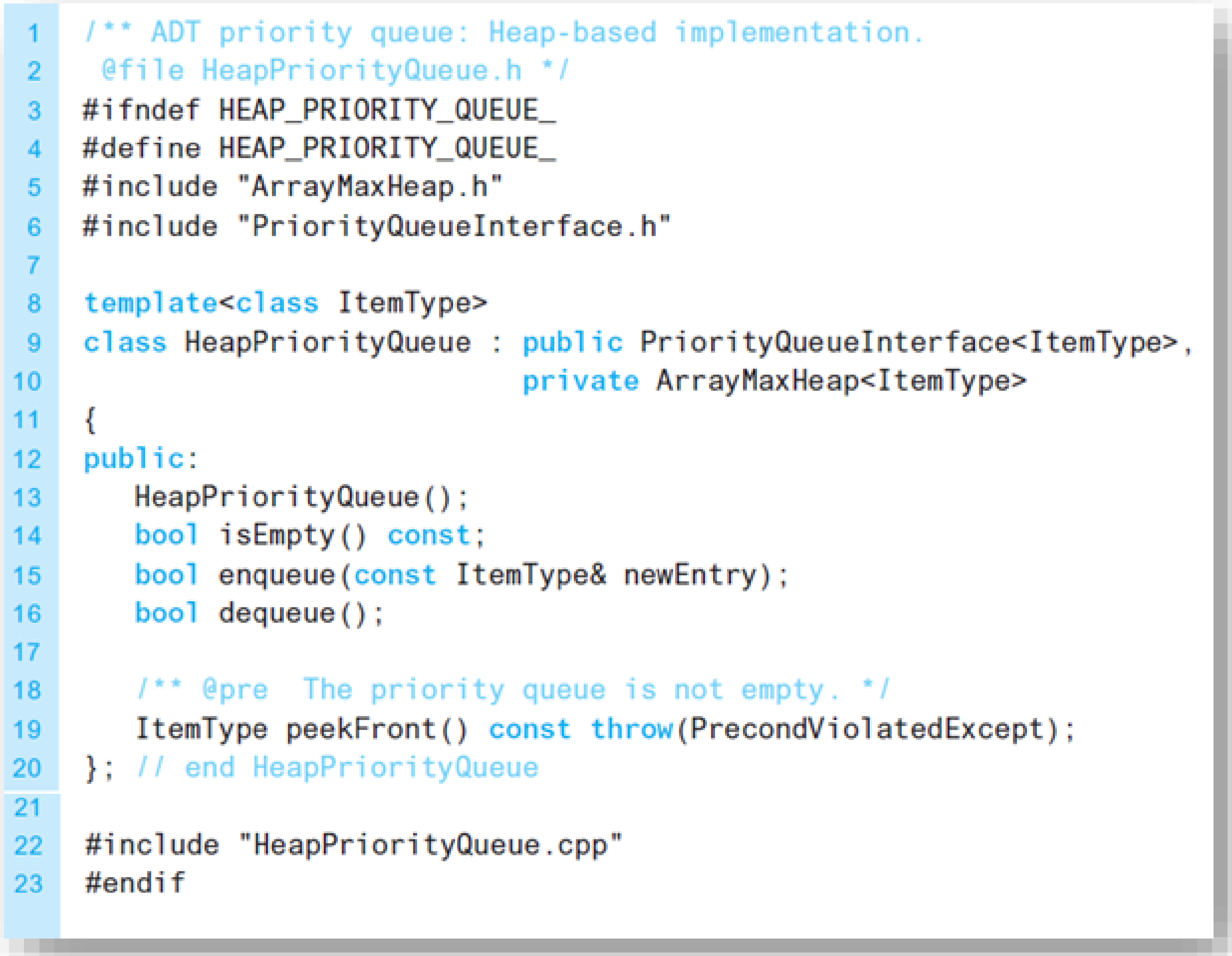
The Implementation

- C++ method peekTop which tests for an empty heap



```
template<class ItemType>
ItemType ArrayMaxHeap<ItemType>::peekTop() const throw(PrecondViolatedExcept)
{
   if (isEmpty())
      throw PrecondViolatedExcep("Attempted peek into an empty heap.");

   return items[0];
}  // end peekTop
```

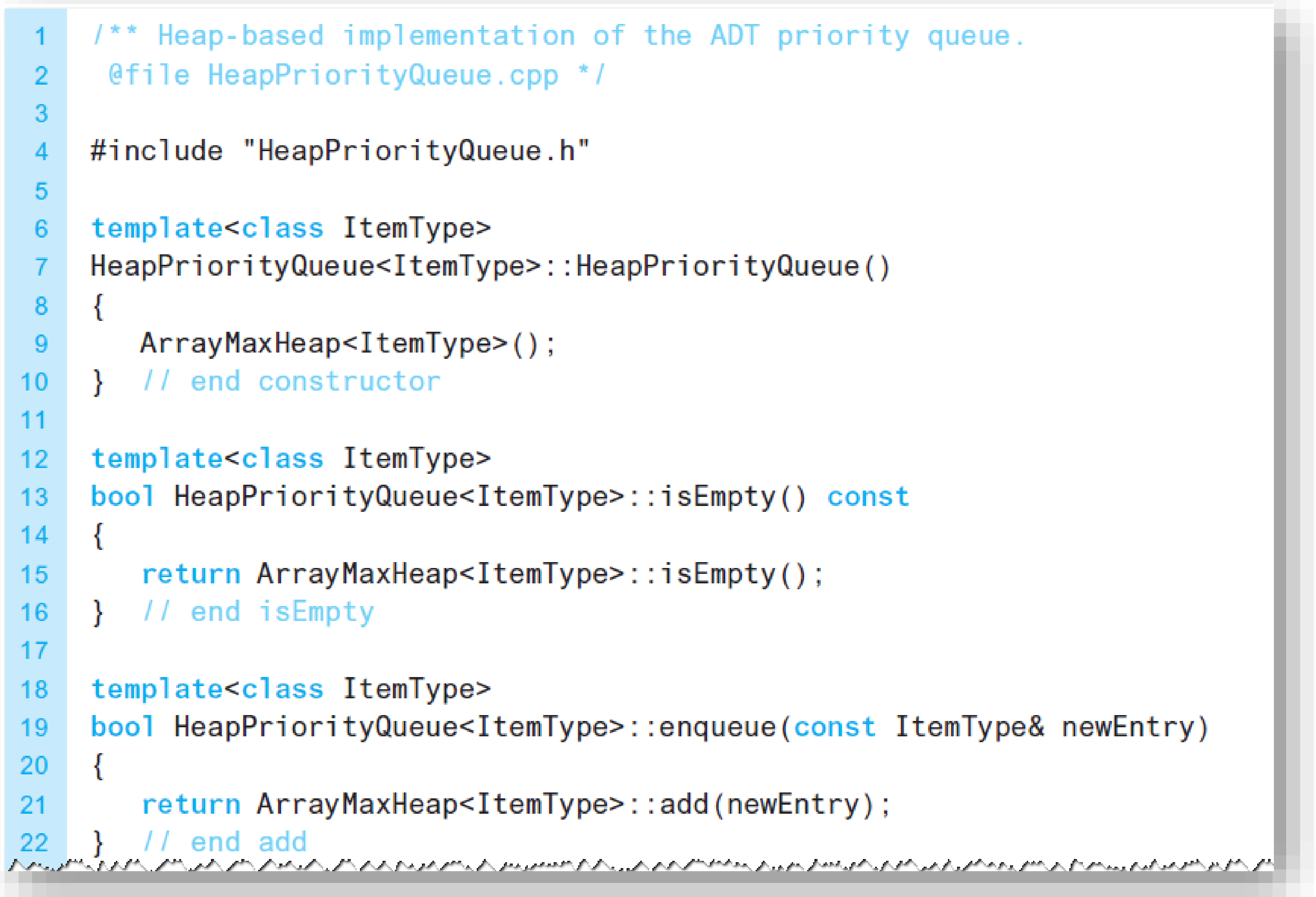
Heap Implementation of the ADT Priority Queue



```
/** ADT priority queue: Heap-based implementation.
 @file HeapPriorityQueue.h */
#ifndef HEAP_PRIORITY_QUEUE_
#define HEAP_PRIORITY_QUEUE_
#include "ArrayMaxHeap.h"
#include "PriorityQueueInterface.h"

template<class ItemType>
class HeapPriorityQueue : public PriorityQueueInterface<ItemType>,
                          private ArrayMaxHeap<ItemType>
{
public:
   HeapPriorityQueue();
   bool isEmpty() const;
   bool enqueue(const ItemType& newEntry);
   bool dequeue();

   /** @pre  The priority queue is not empty. */
   ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end HeapPriorityQueue

#include "HeapPriorityQueue.cpp"
#endif
```

- A header file for the class HeapPriorityQueue

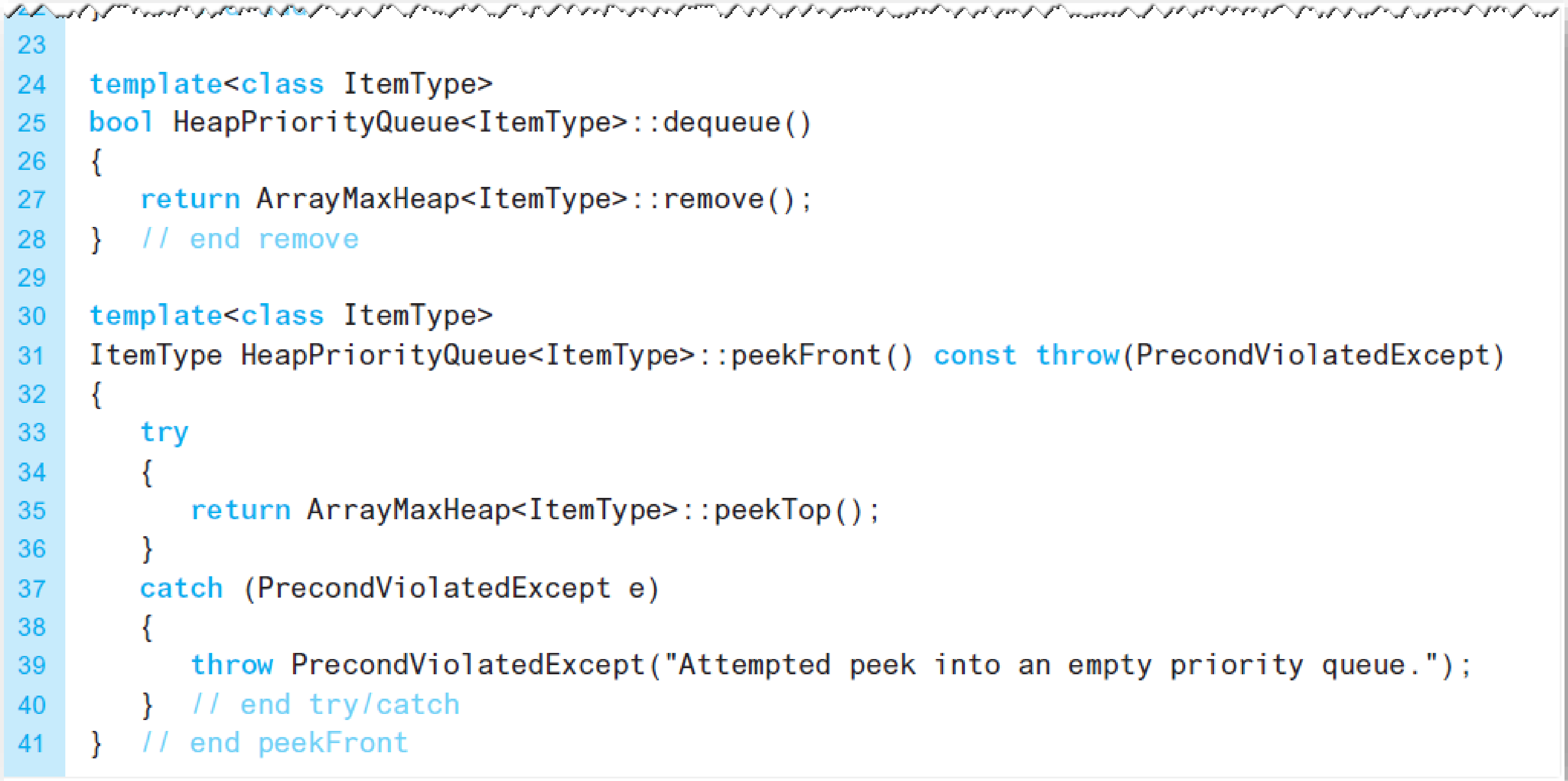
Heap Implementation of the ADT Priority Queue



```
/** Heap-based implementation of the ADT priority queue.
 @file HeapPriorityQueue.cpp */

#include "HeapPriorityQueue.h"

template<class ItemType>
HeapPriorityQueue<ItemType>::HeapPriorityQueue()
{
   ArrayMaxHeap<ItemType>();
}  // end constructor

template<class ItemType>
bool HeapPriorityQueue<ItemType>::isEmpty() const
{
   return ArrayMaxHeap<ItemType>::isEmpty();
}  // end isEmpty

template<class ItemType>
bool HeapPriorityQueue<ItemType>::enqueue(const ItemType& newEntry)
{
   return ArrayMaxHeap<ItemType>::add(newEntry);
}  // end add
```

- An implementation of the class HeapPriorityQueue

Heap Implementation of the ADT Priority Queue



```
template<class ItemType>
bool HeapPriorityQueue<ItemType>::dequeue()
{
   return ArrayMaxHeap<ItemType>::remove();
}  // end remove

template<class ItemType>
ItemType HeapPriorityQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
{
   try
   {
      return ArrayMaxHeap<ItemType>::peekTop();
   }
   catch (PrecondViolatedExcept e)
   {
      throw PrecondViolatedExcept("Attempted peek into an empty priority queue.");
   }  // end try/catch
}  // end peekFront
```

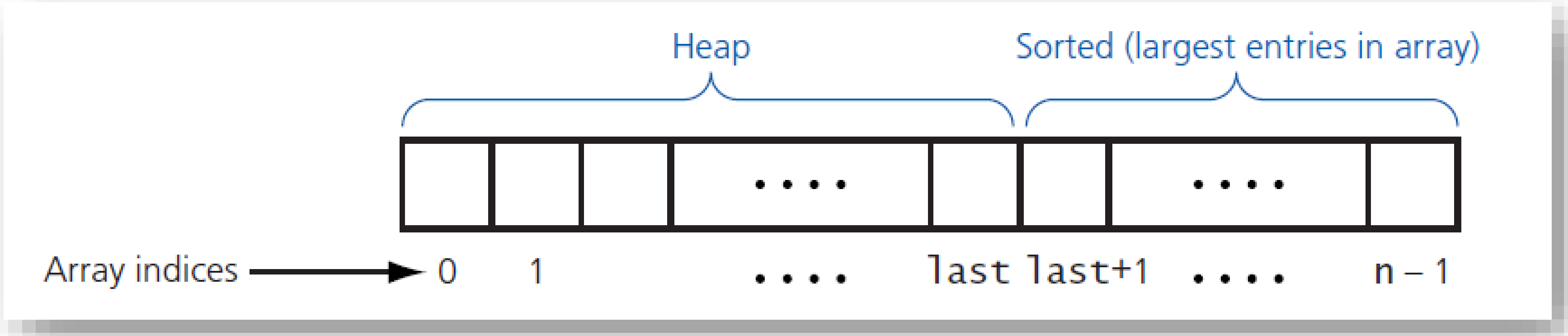
- An implementation of the class HeapPriorityQueue

Heap Implementation of the ADT Priority Queue

- Heap versus a binary search tree
  - If you know maximum number of items in the priority queue, heap is the better implementation
- Finite, distinct priority values
  - Many items likely have same priority value
  - Place in same order as encountered

Heap Sort

- Heap sort partitions an array into two regions



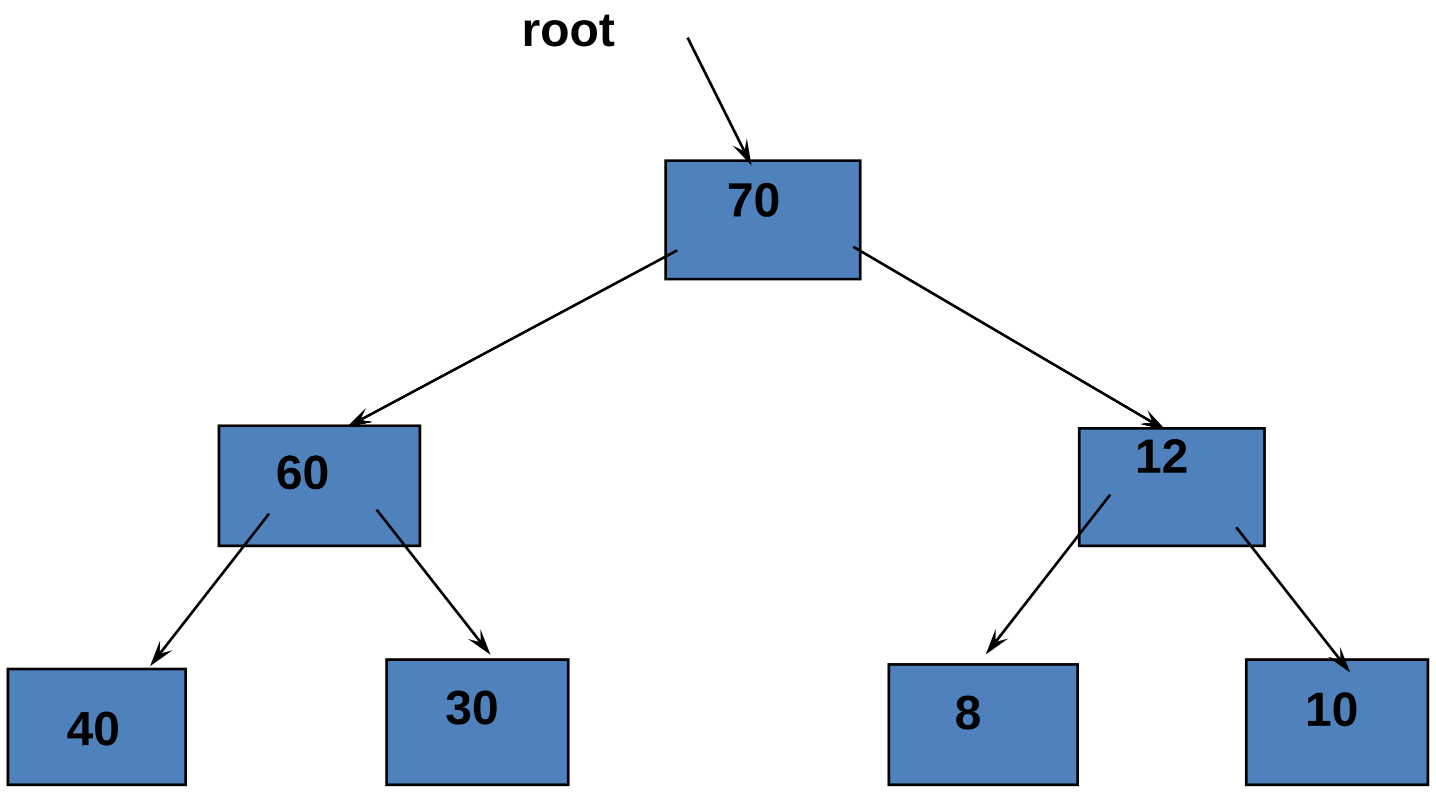
Heap Sort: Recall that . . .

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

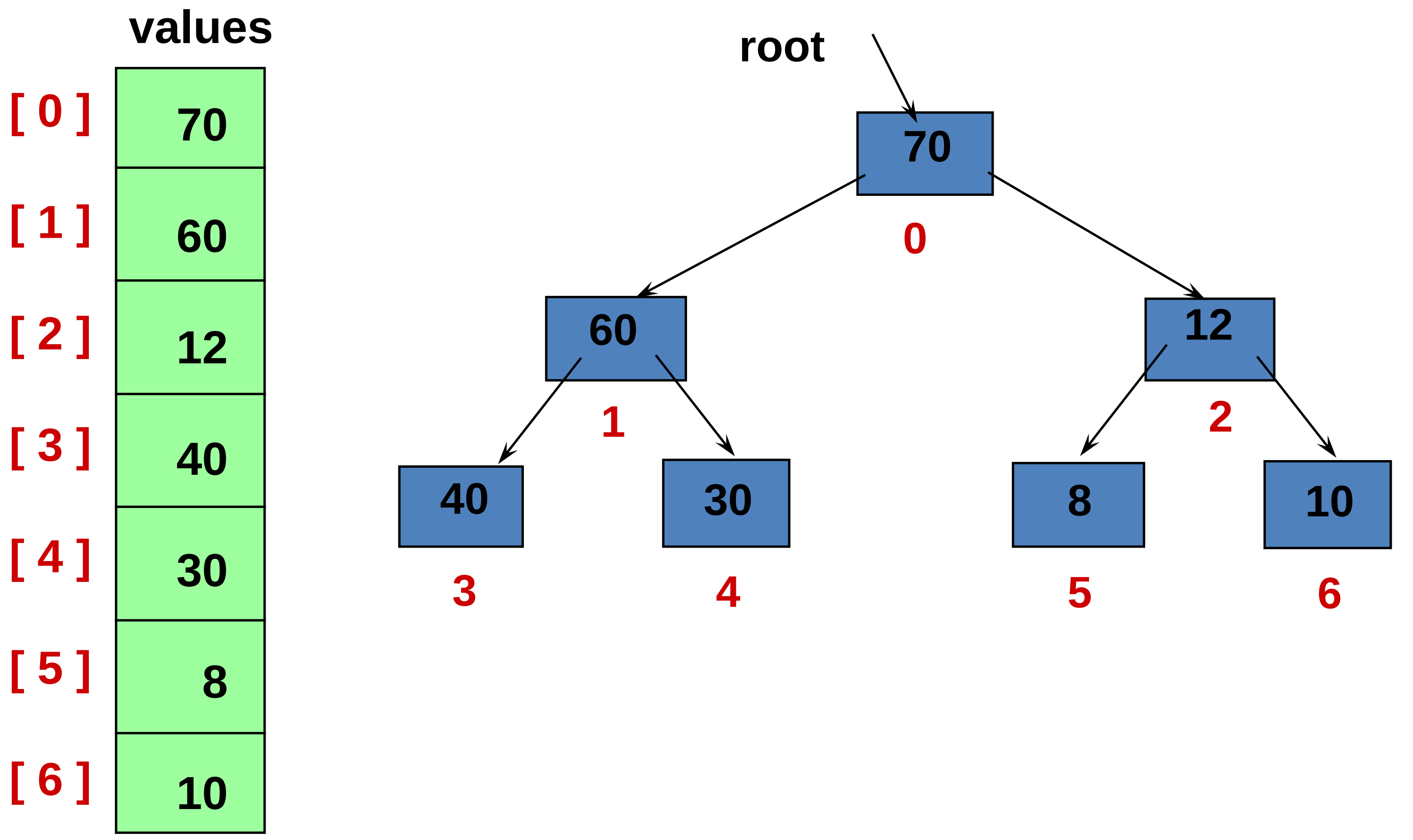
- Its shape must be a complete binary tree.
- For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

The largest element in a heap

is always found in the root node



The heap can be stored



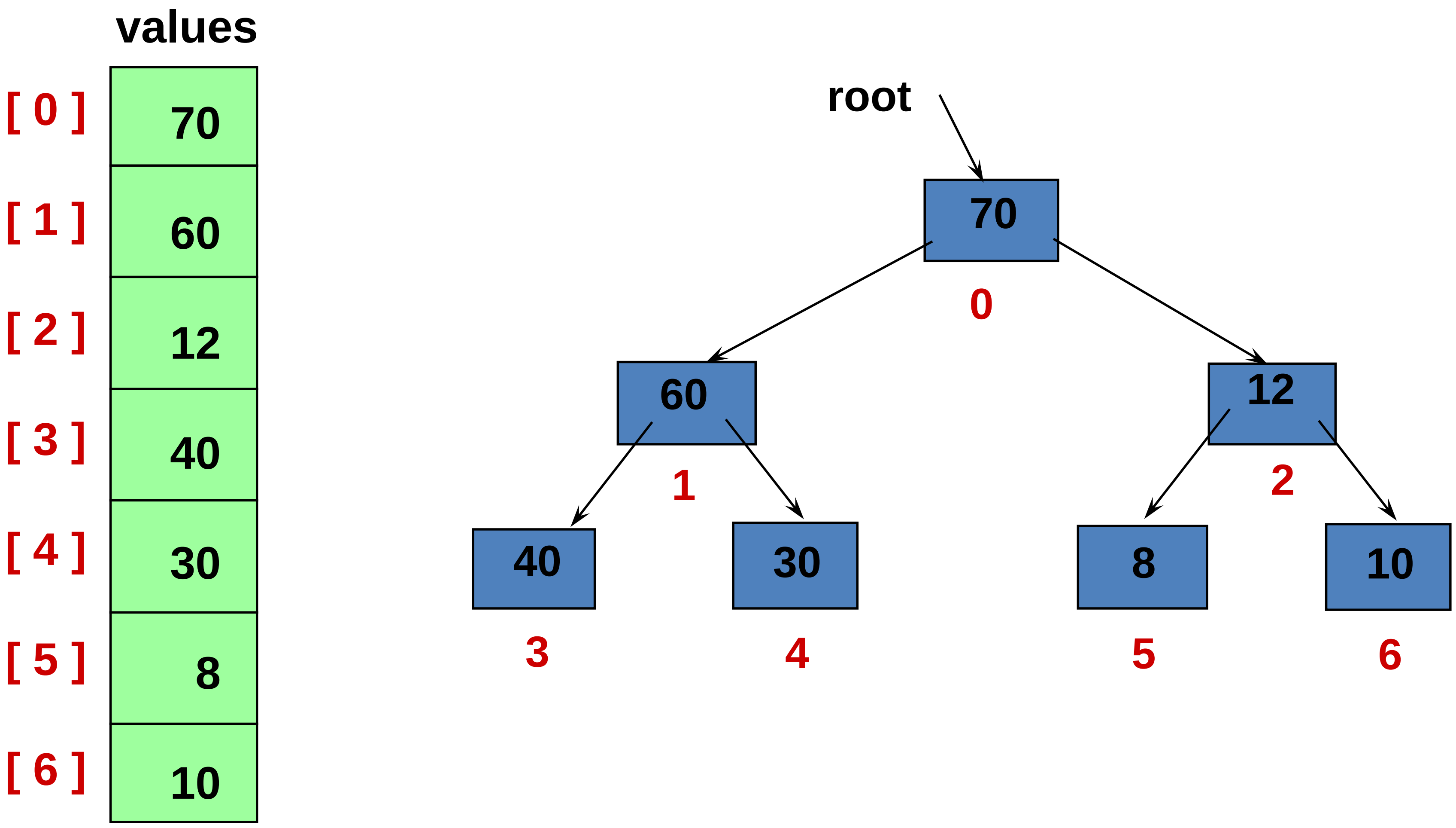
Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property.

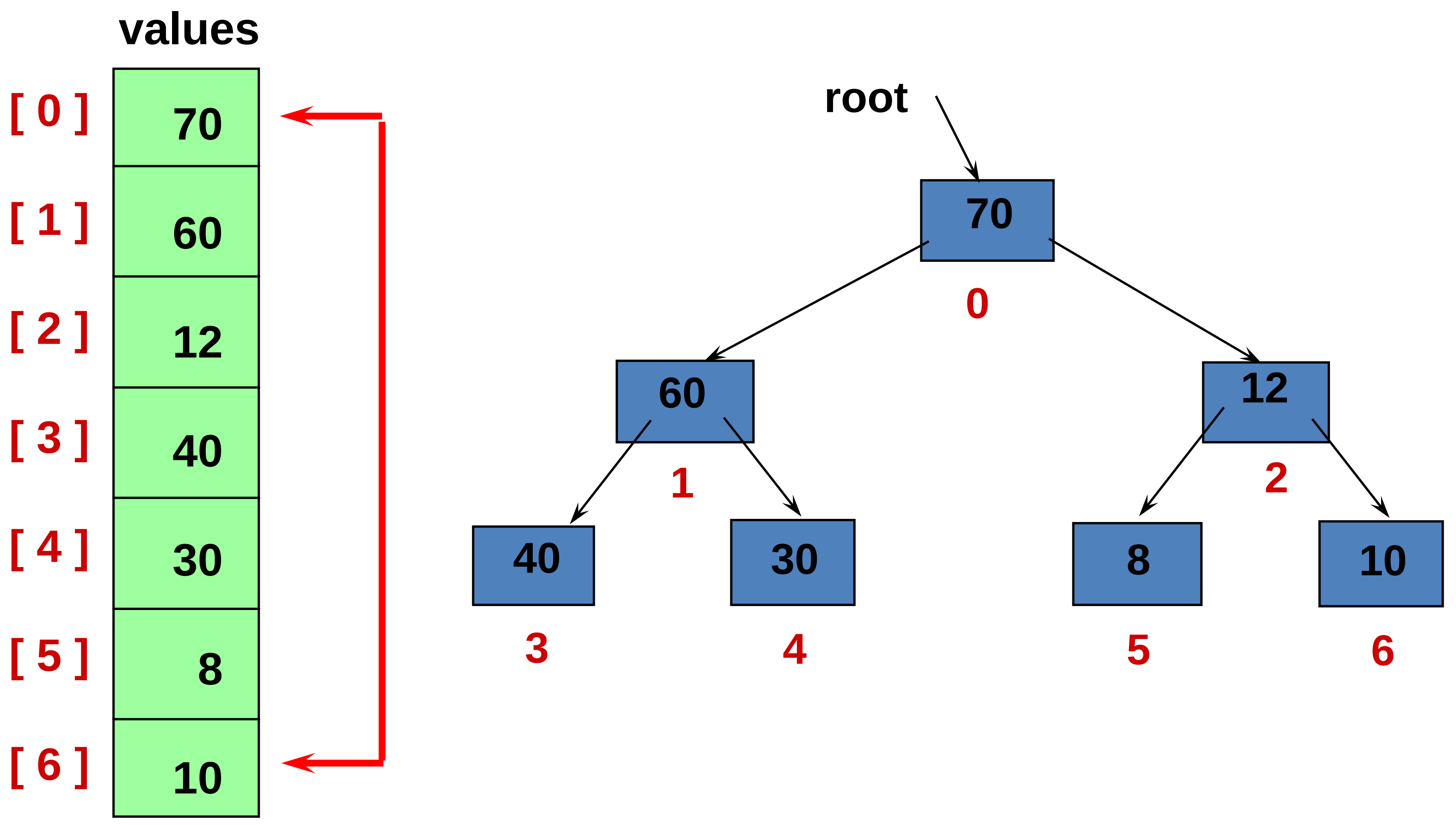
Then repeat the steps below until there are no more unsorted elements.

- Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.
- Reheap the remaining unsorted elements.
    - This puts the next-largest element into the root position.

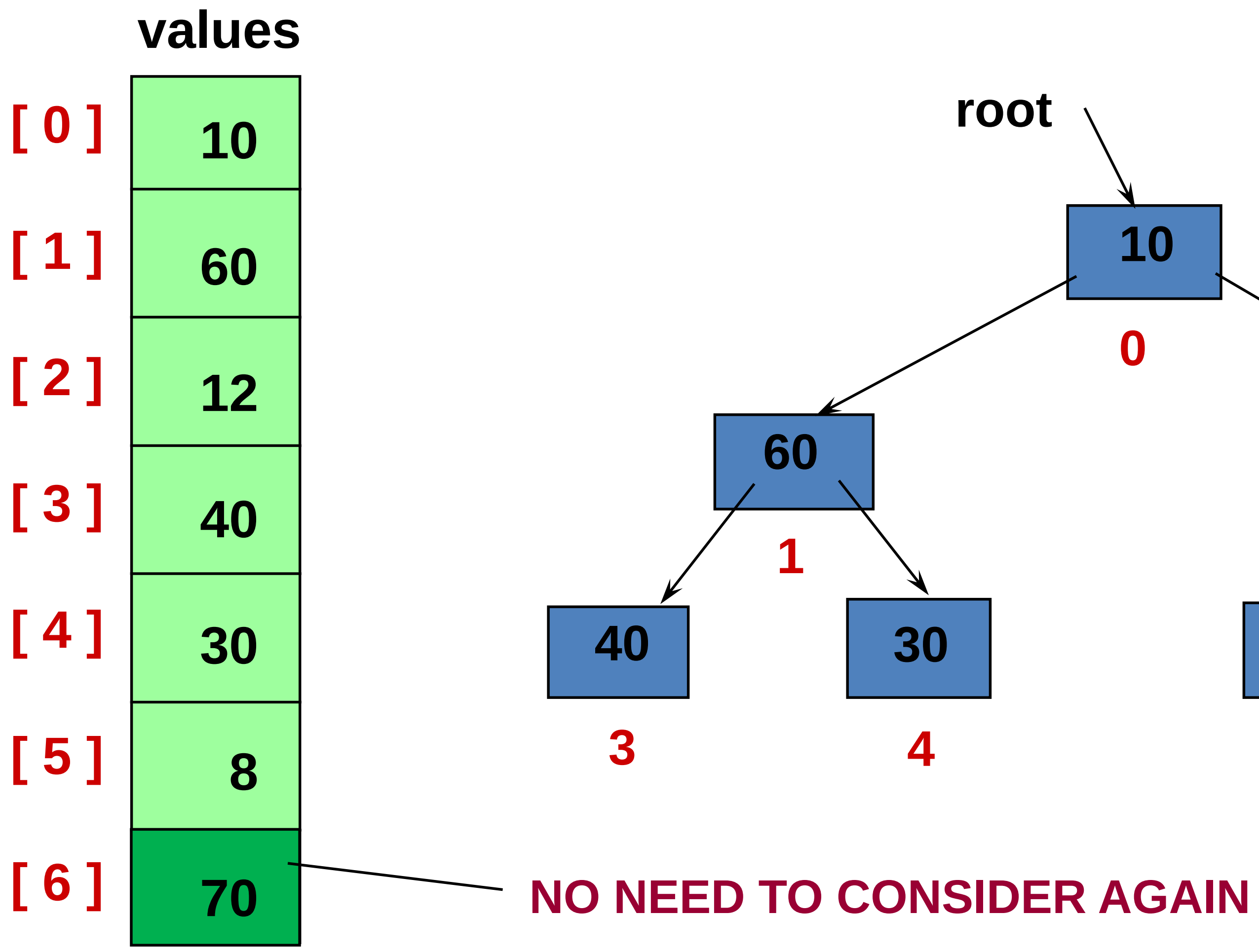
After creating the original heap



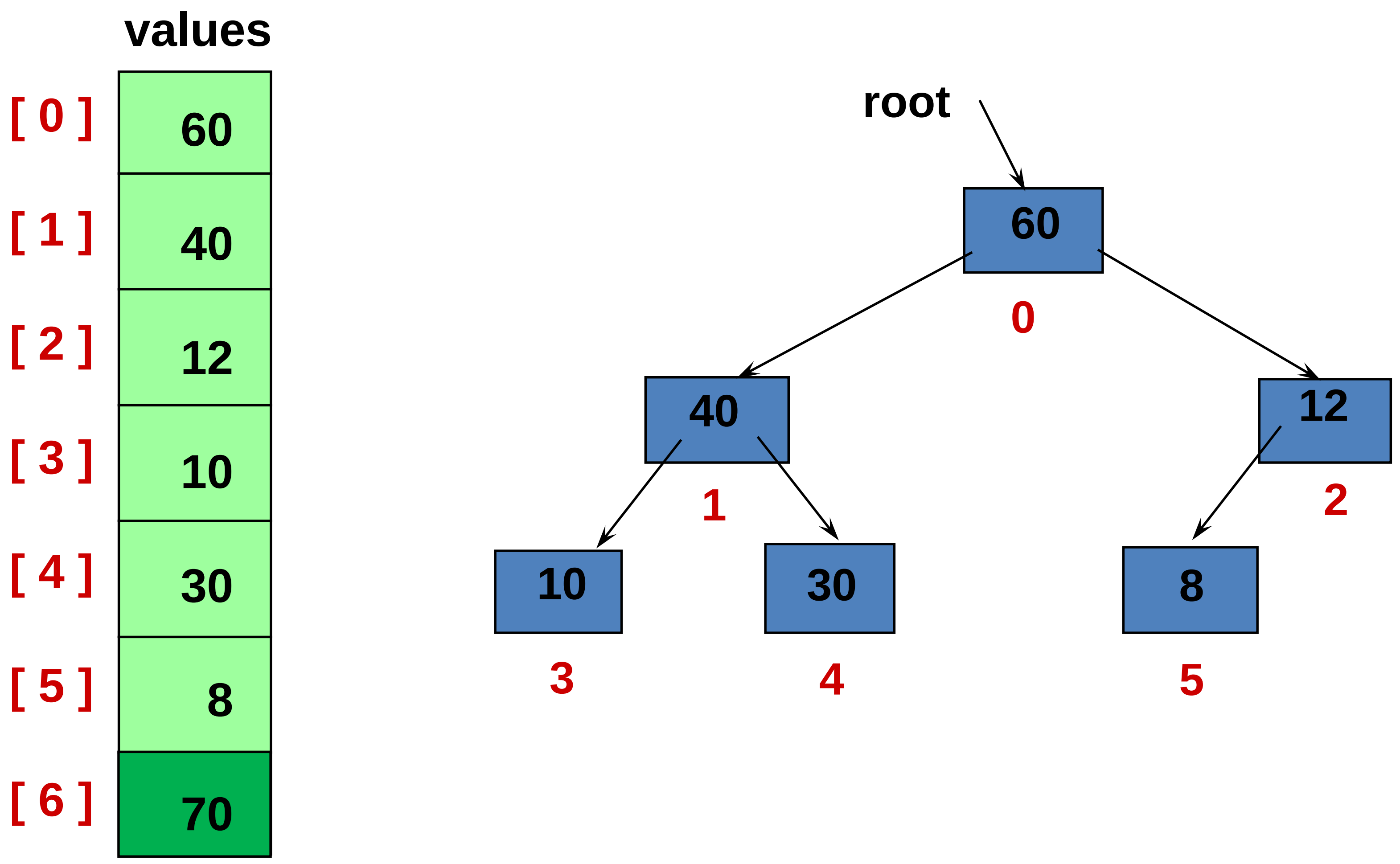
Swap root element into last place in unsorted array



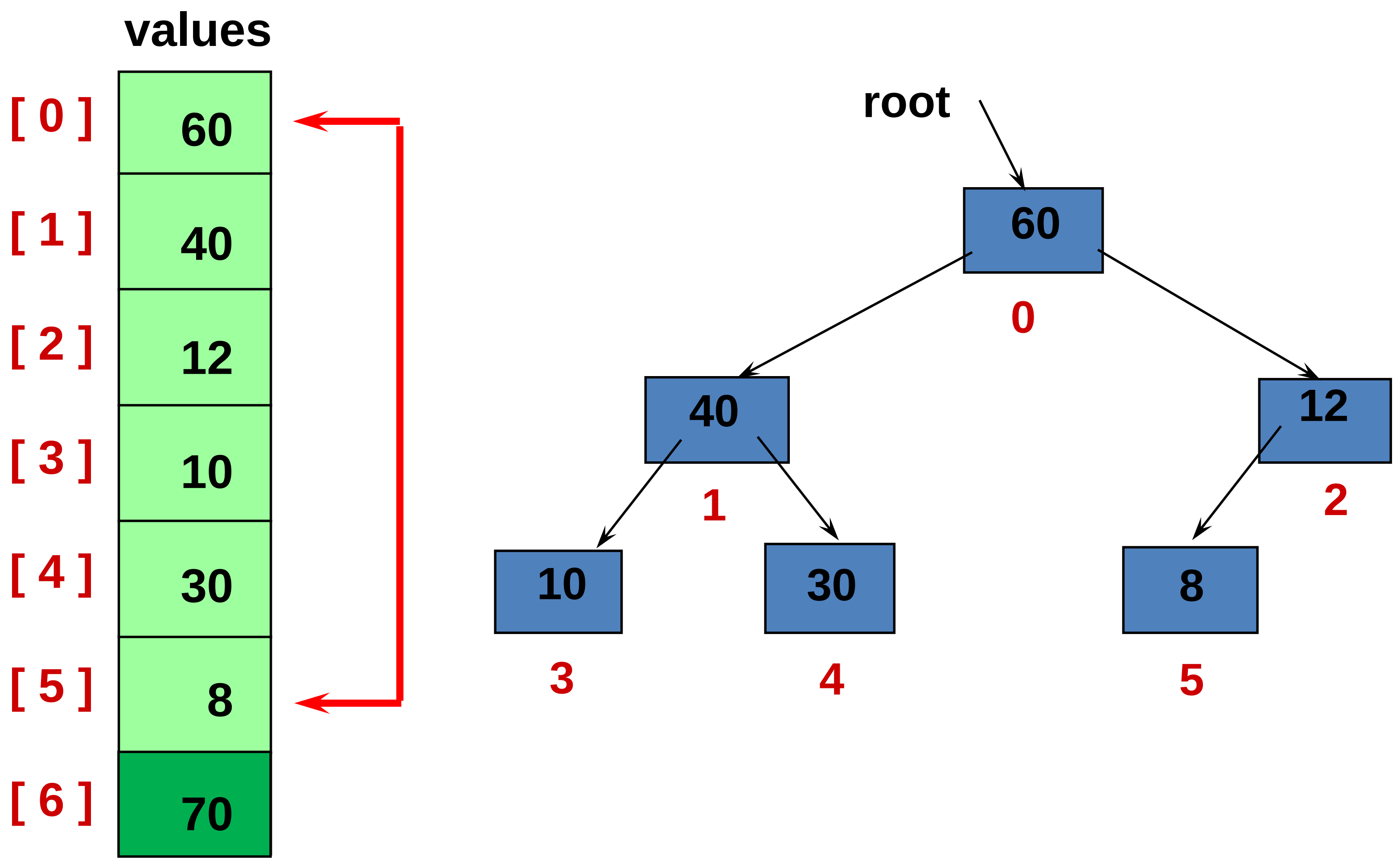
After swapping root element into it place



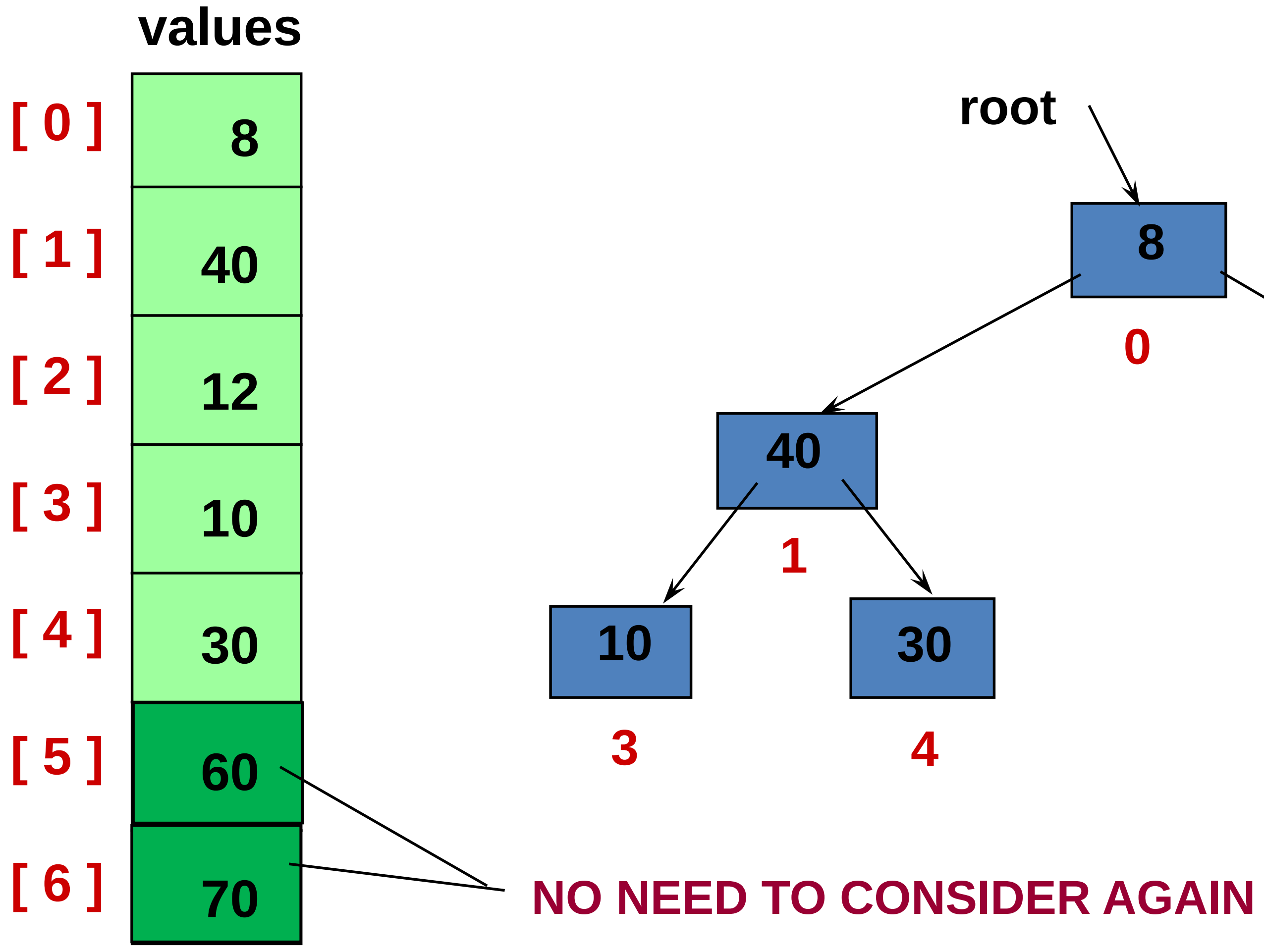
After reheaping remaining unsorted elements



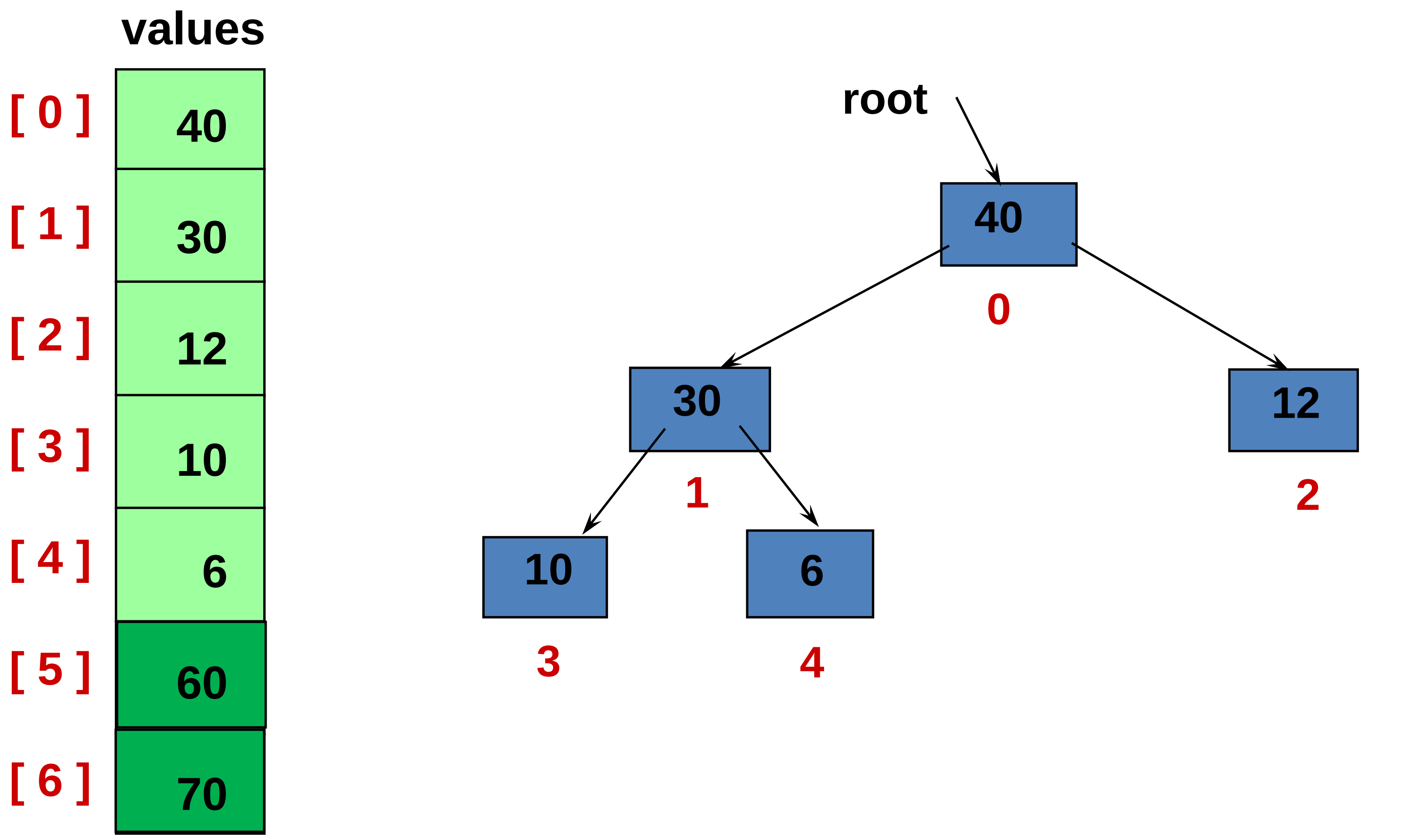
Swap root element into last place in unsorted array



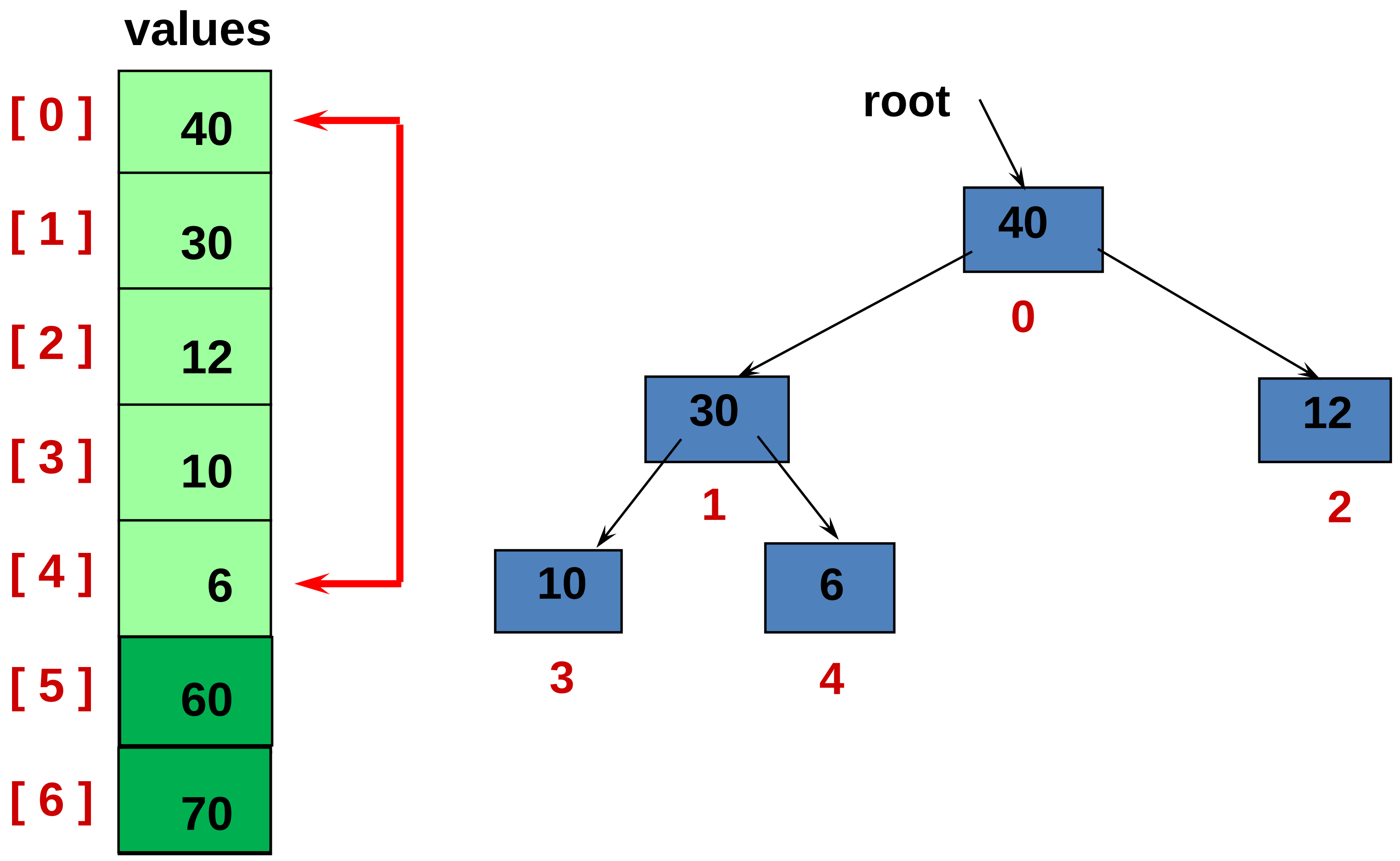
After swapping root element into its place



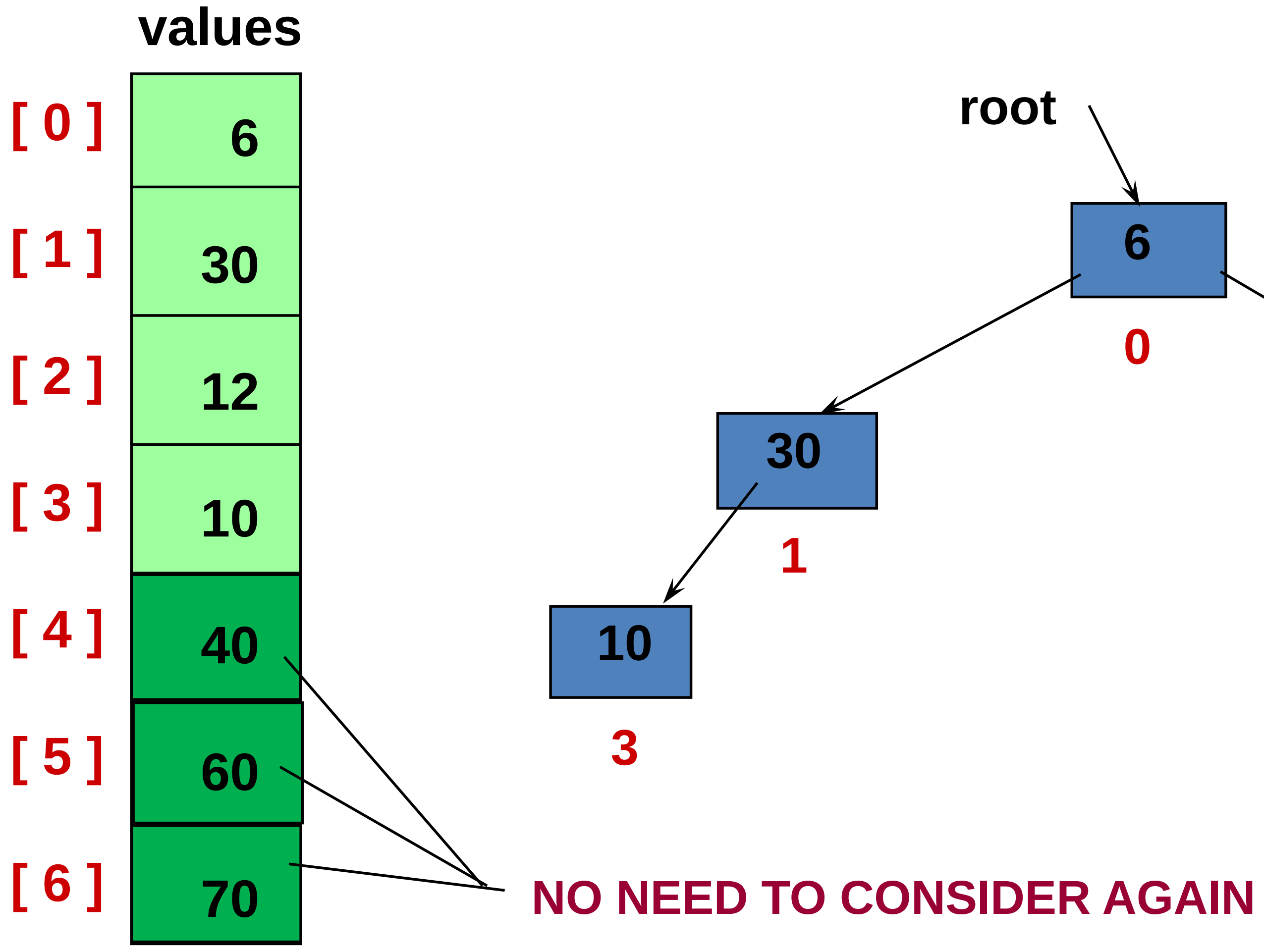
After reheaping remaining unsorted elements



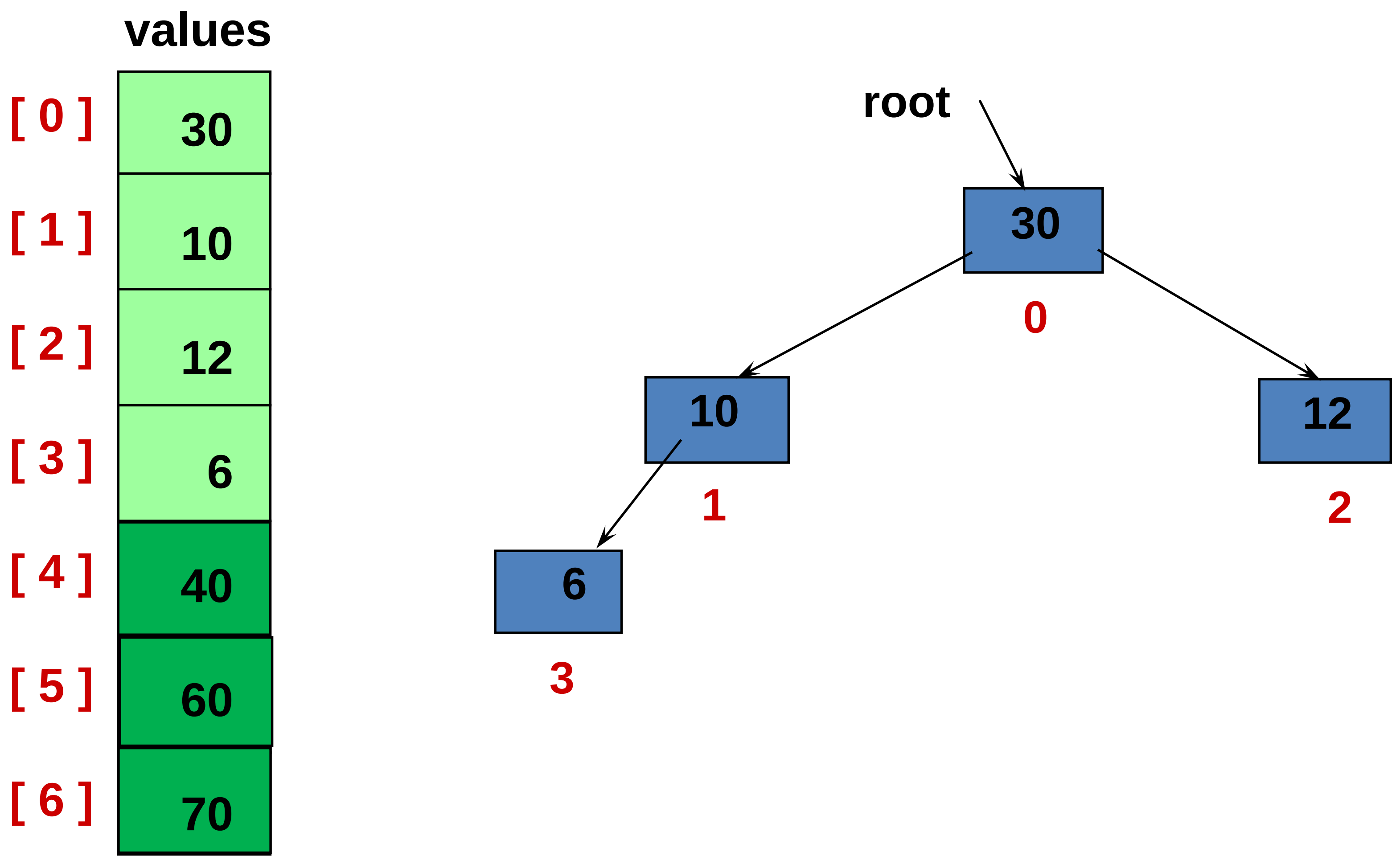
Swap root element into last place in unsorted array



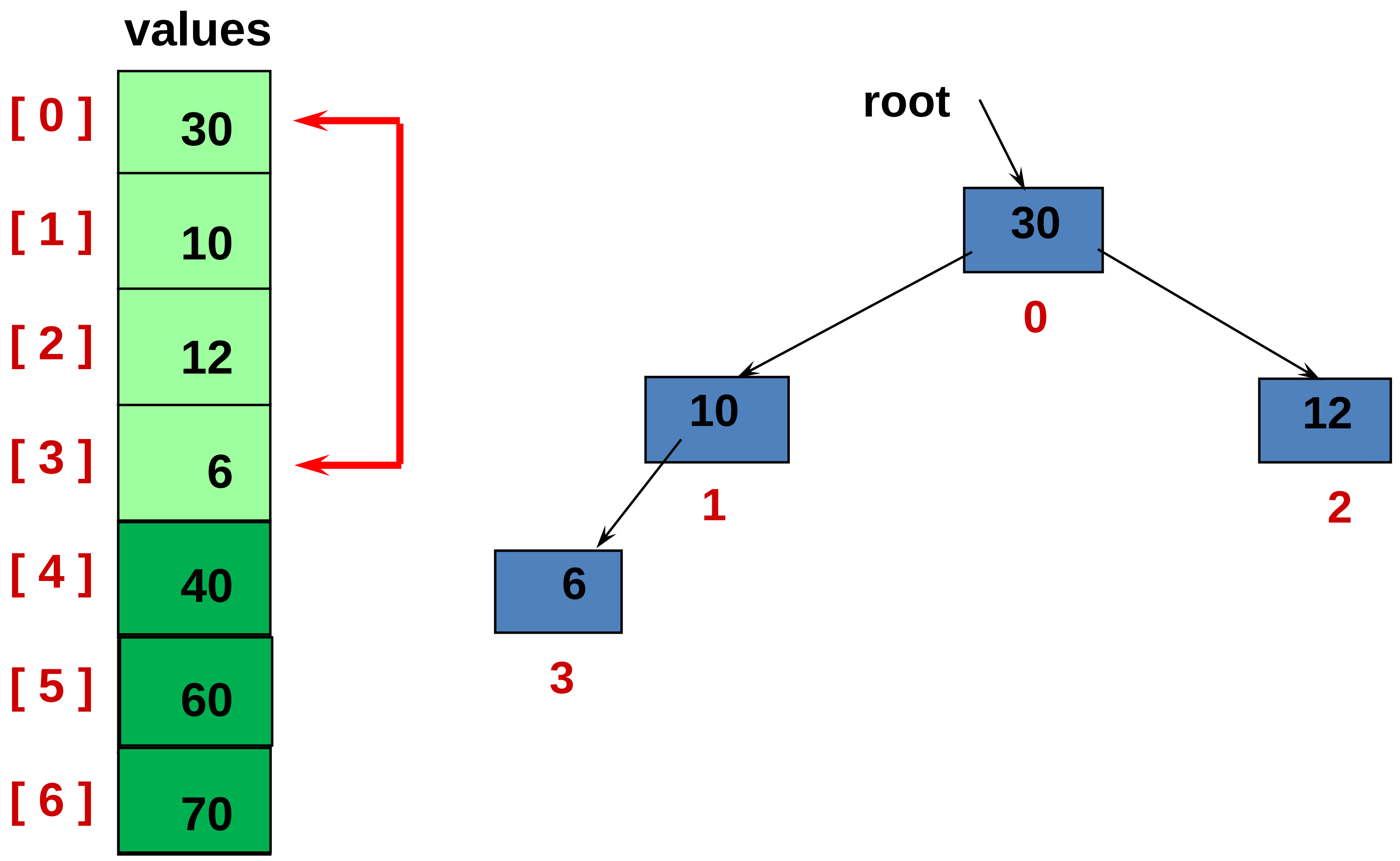
After swapping root element into its place



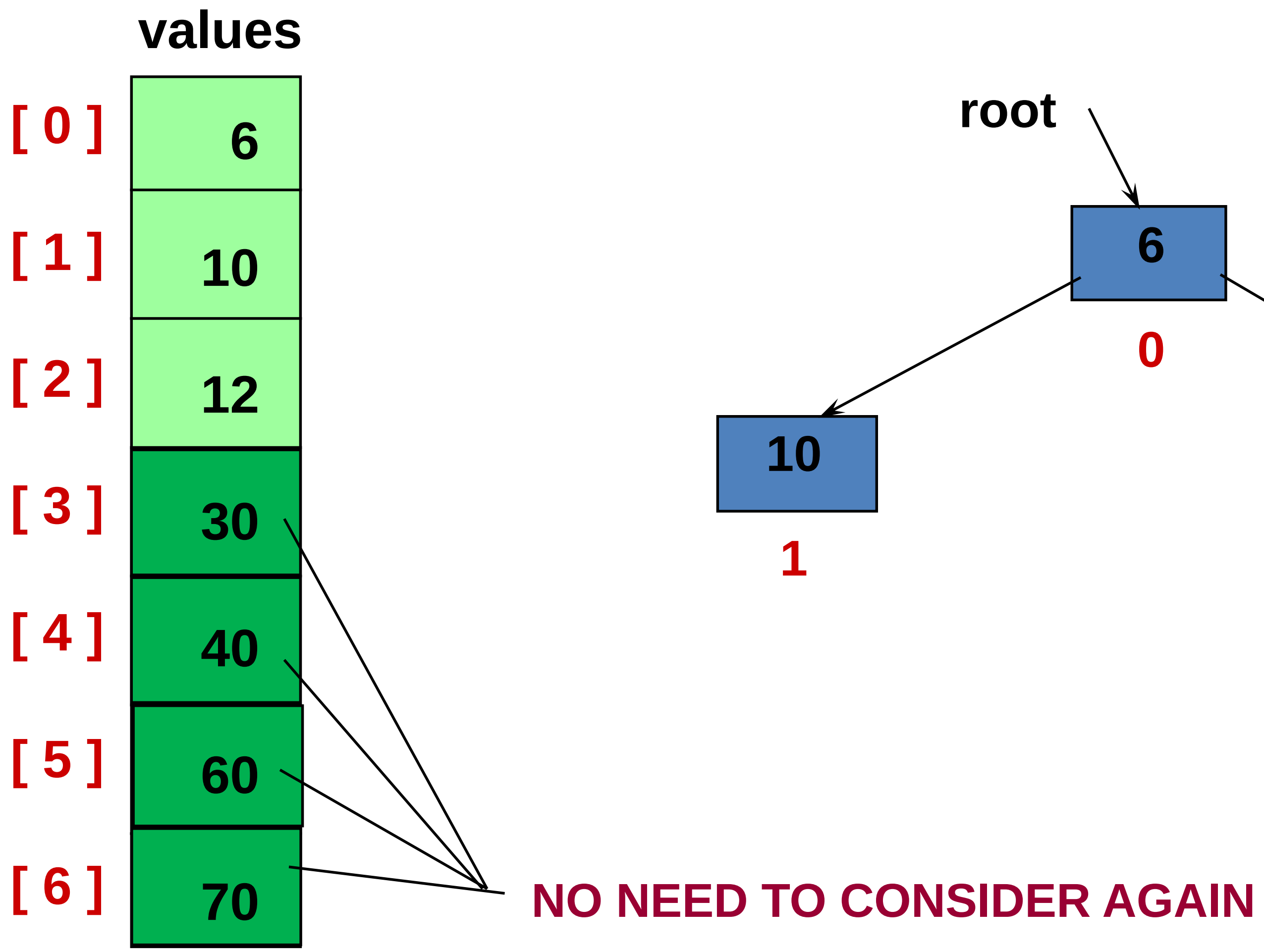
After reheaping remaining unsorted elements



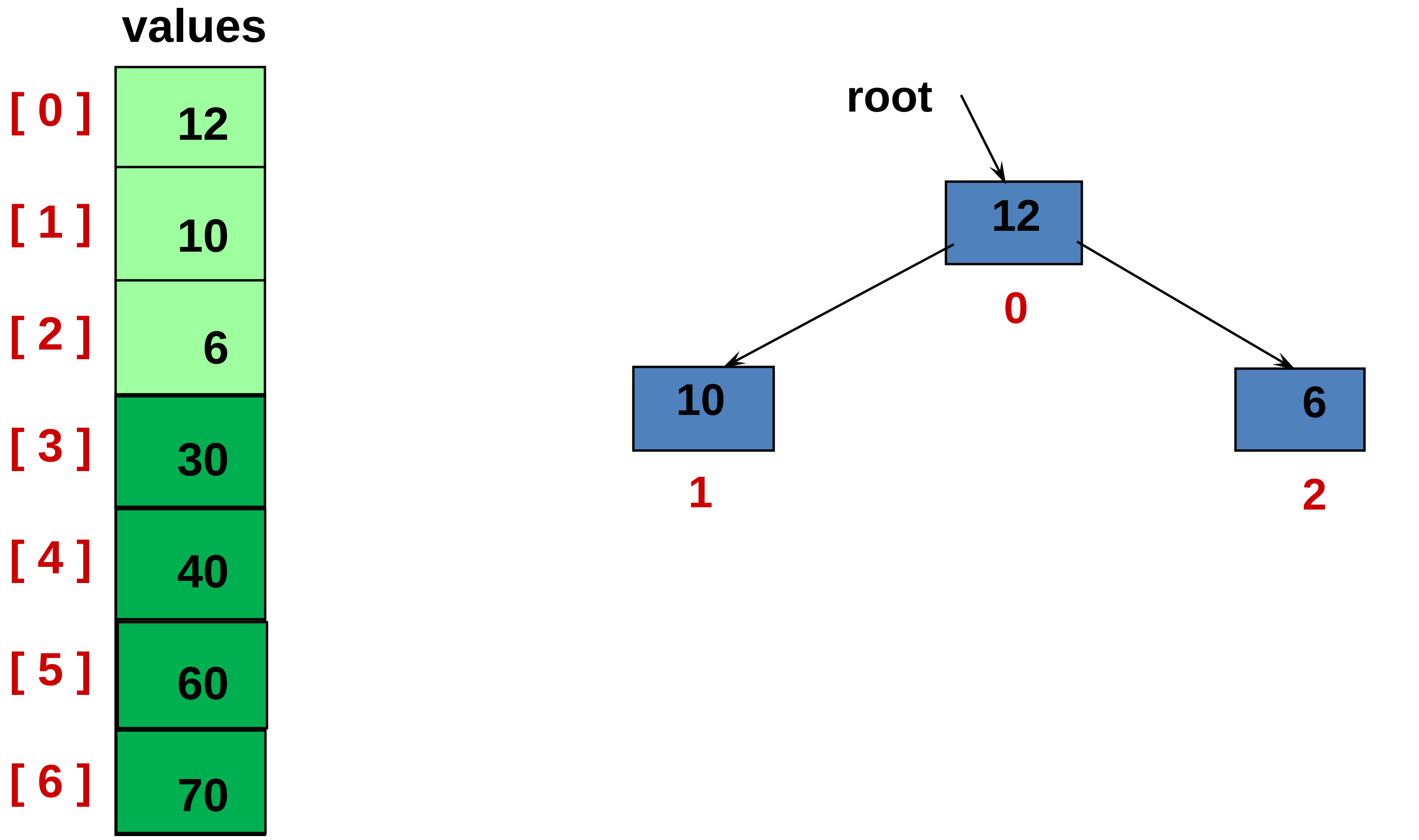
Swap root element into last place in unsorted array



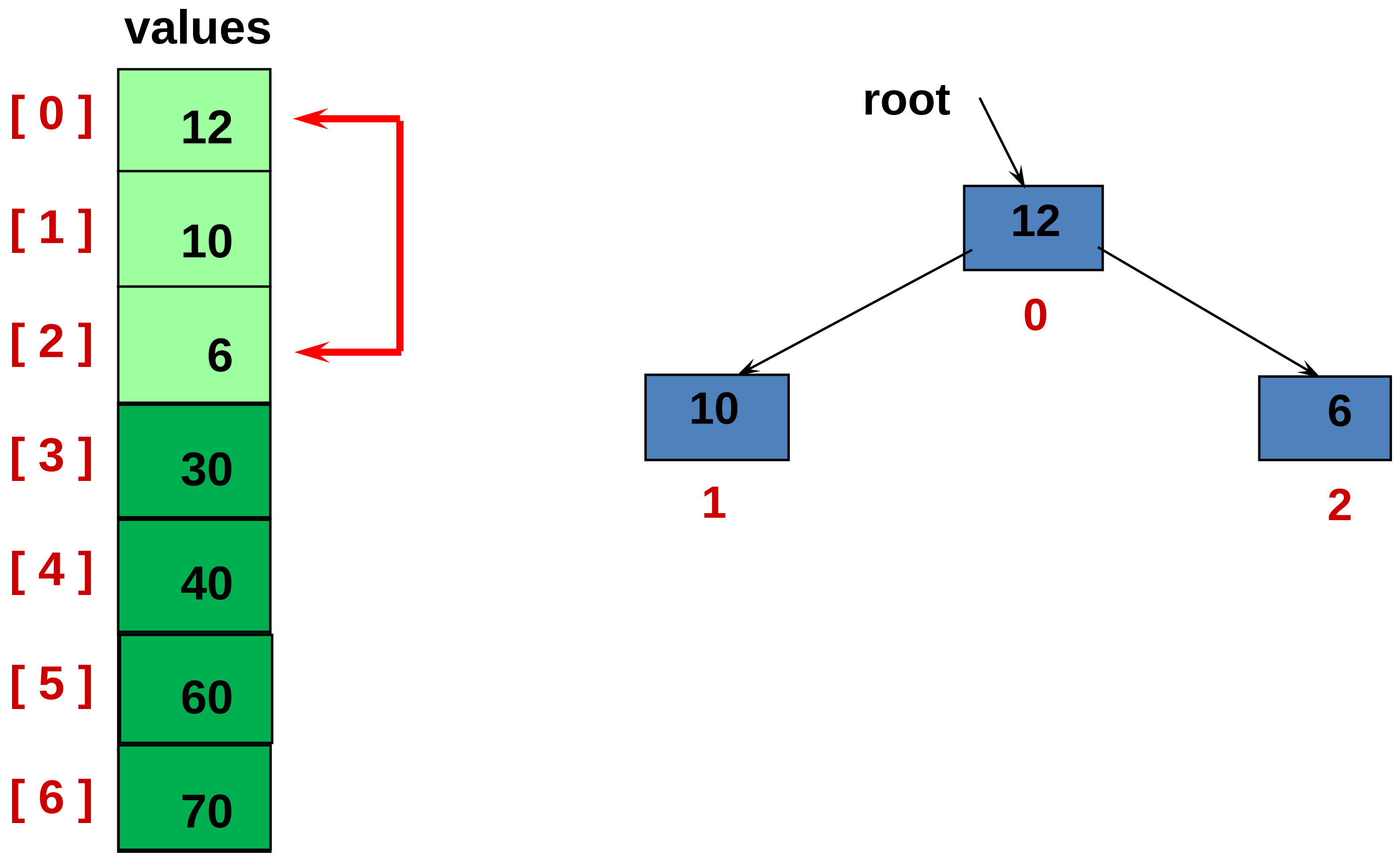
After swapping root element into its place



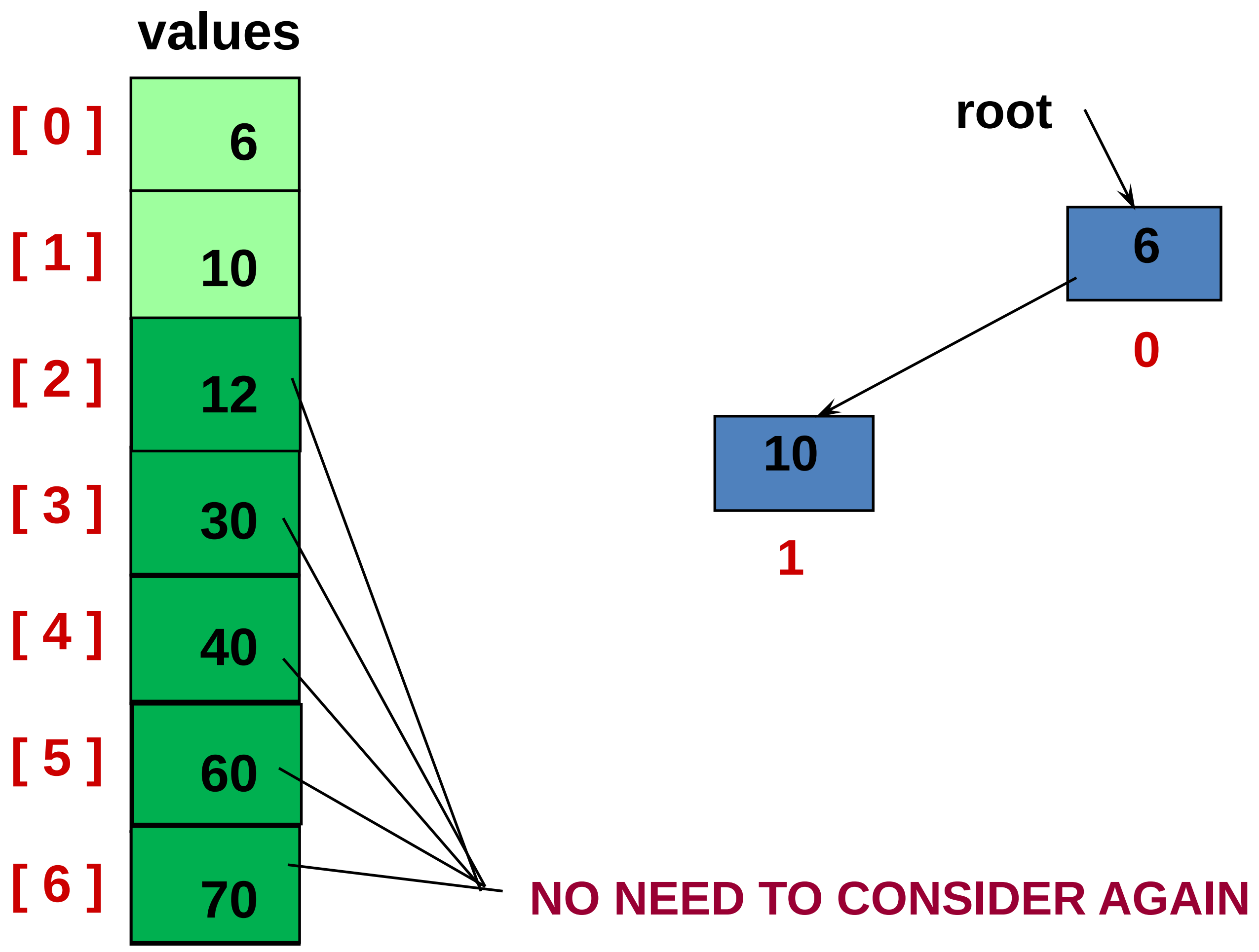
After reheaping remaining unsorted elements



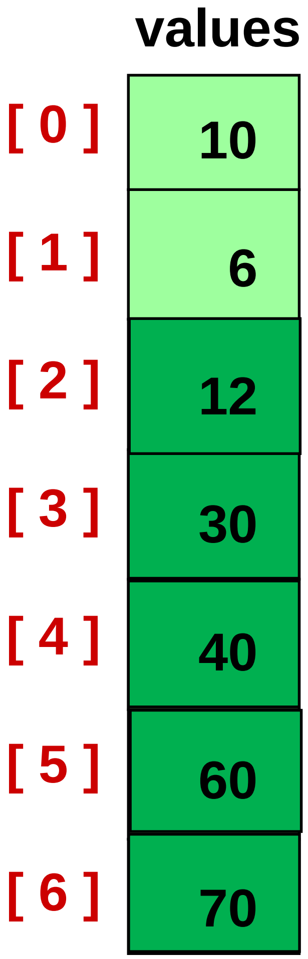
Swap root element into last place in unsorted array

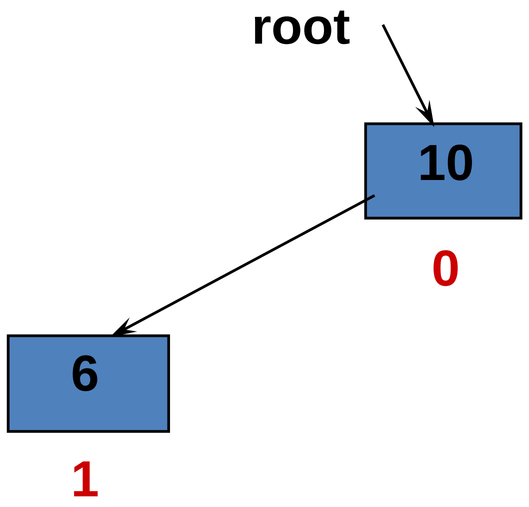


After swapping root element into its place

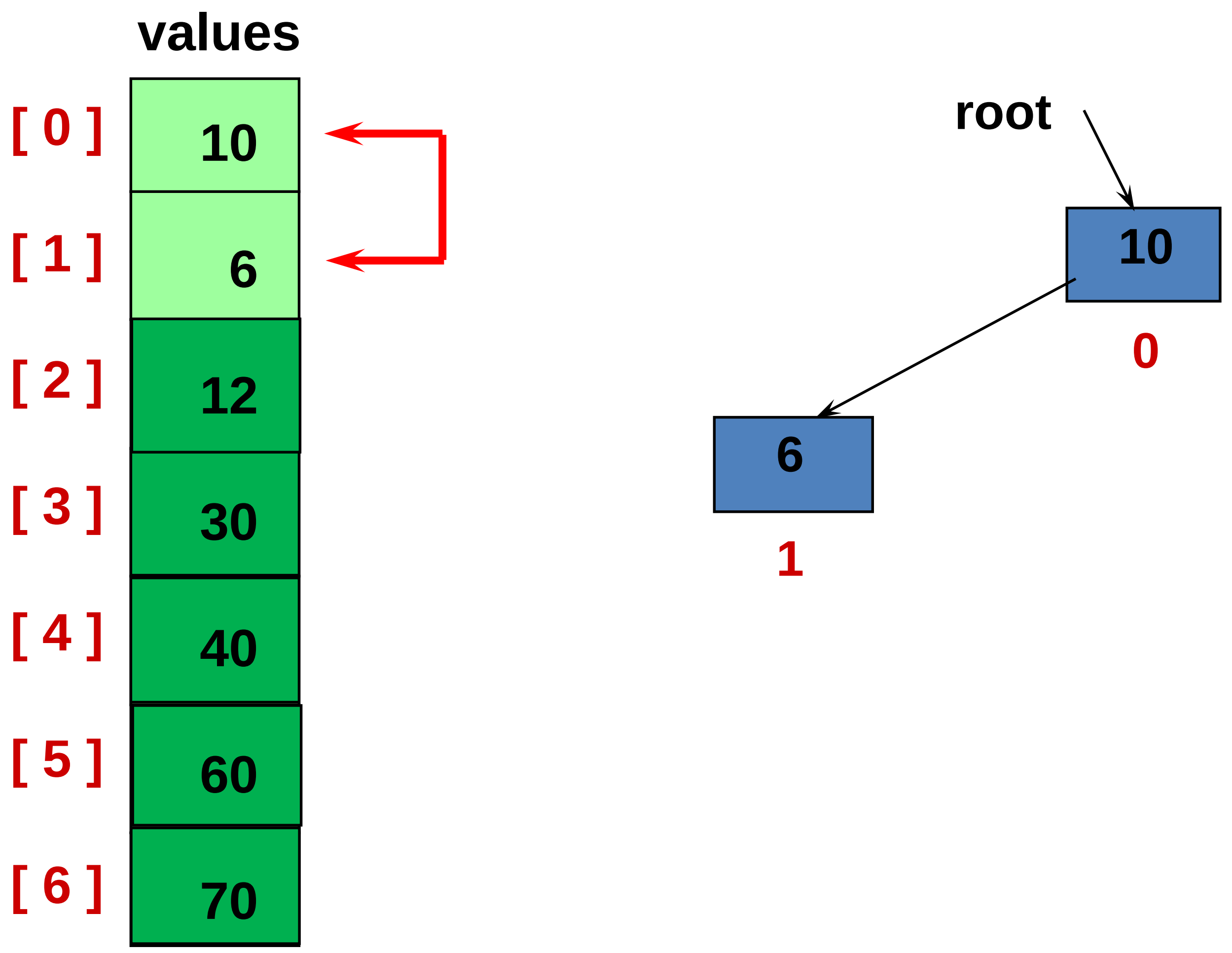


After reheaping remaining unsorted elements

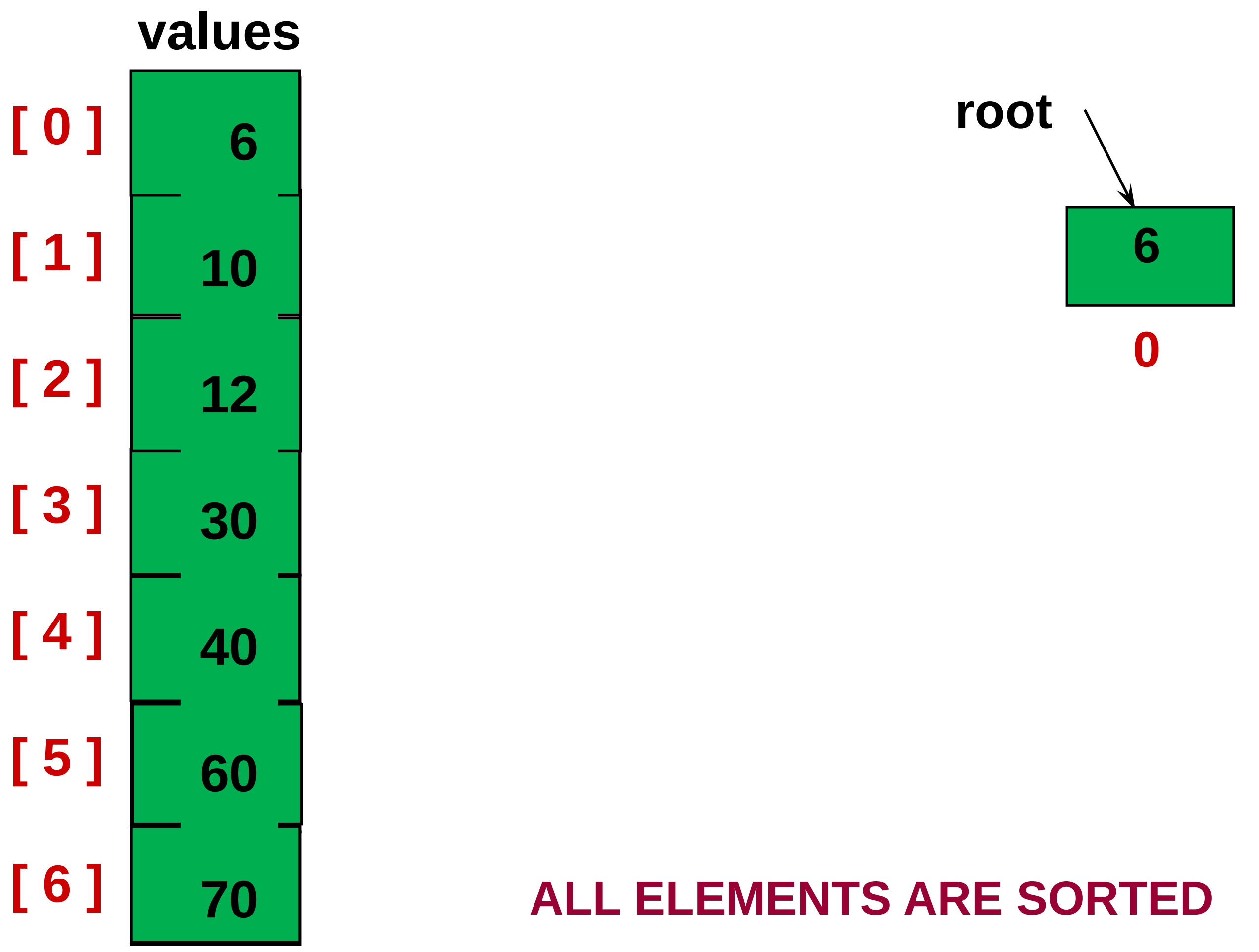




Swap root element into last place in unsorted array



After swapping root element into its place



ALL ELEMENTS ARE SORTED

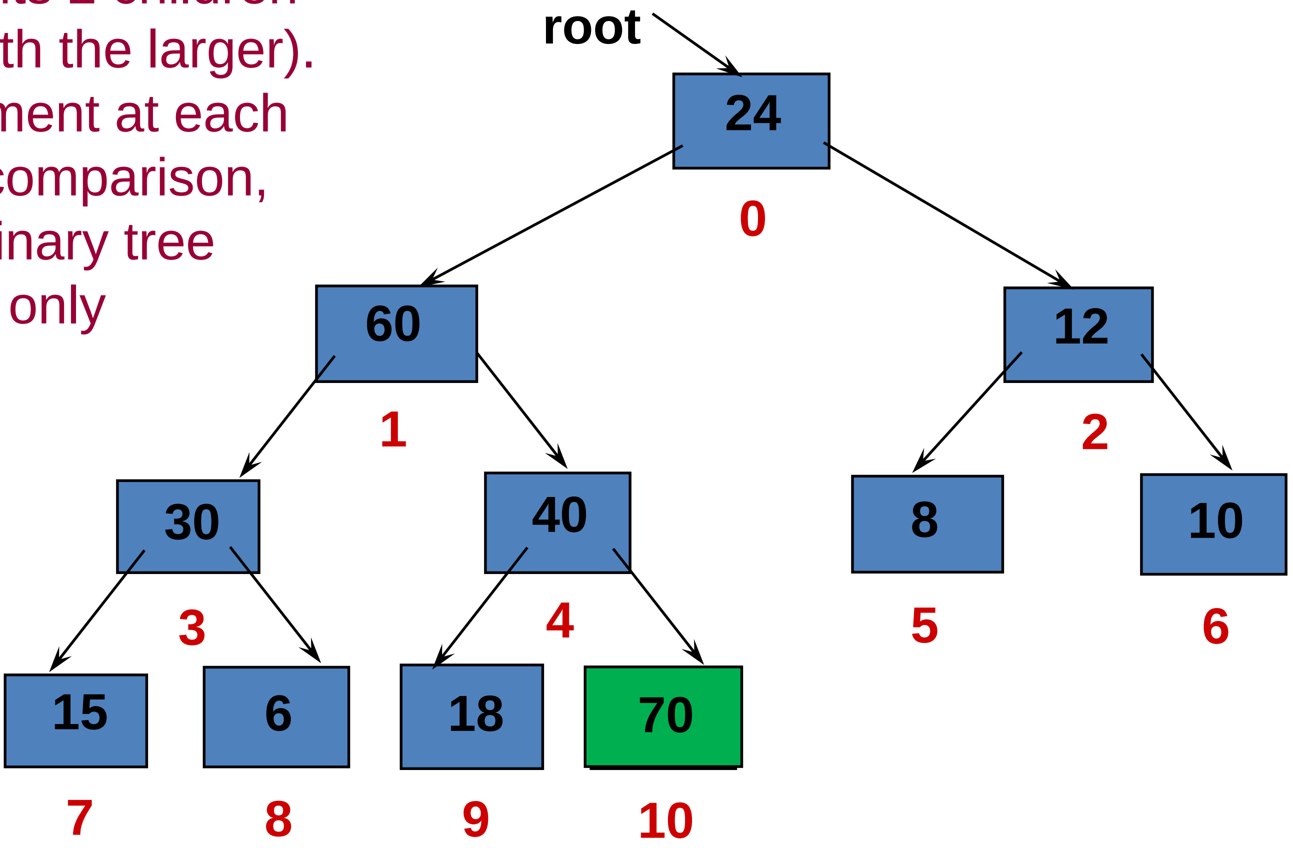
```
template <class  ItemType >
void  HeapSort ( ItemType  values[], int  numValues )

//  Post: Sorts array values[ 0 . . numValues-1 ] into
//   ascending order by key
{
   int  index ;

   // Convert array  values[0..numValues-1] 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);
   }
}
```

ReheapDown

```
template< class  ItemType >
void  ReheapDown ( ItemType  values[], int root, int bottom )

//  Pre:  root is the index of a 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 ;
    if (leftChild  <=  bottom)  // ReheapDown continued
     {
       if  (leftChild  ==  bottom)
        maxChild = leftChild;
       else
       {
        if (values[leftChild] <=  values [rightChild])
          maxChild  =  rightChild ;
        else
           maxChild  =  leftChild ;
       }
       if  (values[ root ] < values[maxChild])
       {
        Swap (values[root], values[maxChild]);
        ReheapDown ( maxChild, bottom )    ;
       }
      }
}
```

Heap Sort: How many comparisons?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only $O(\log_2 N)$ levels.



Heap Sort of N elements: How many comparisons?

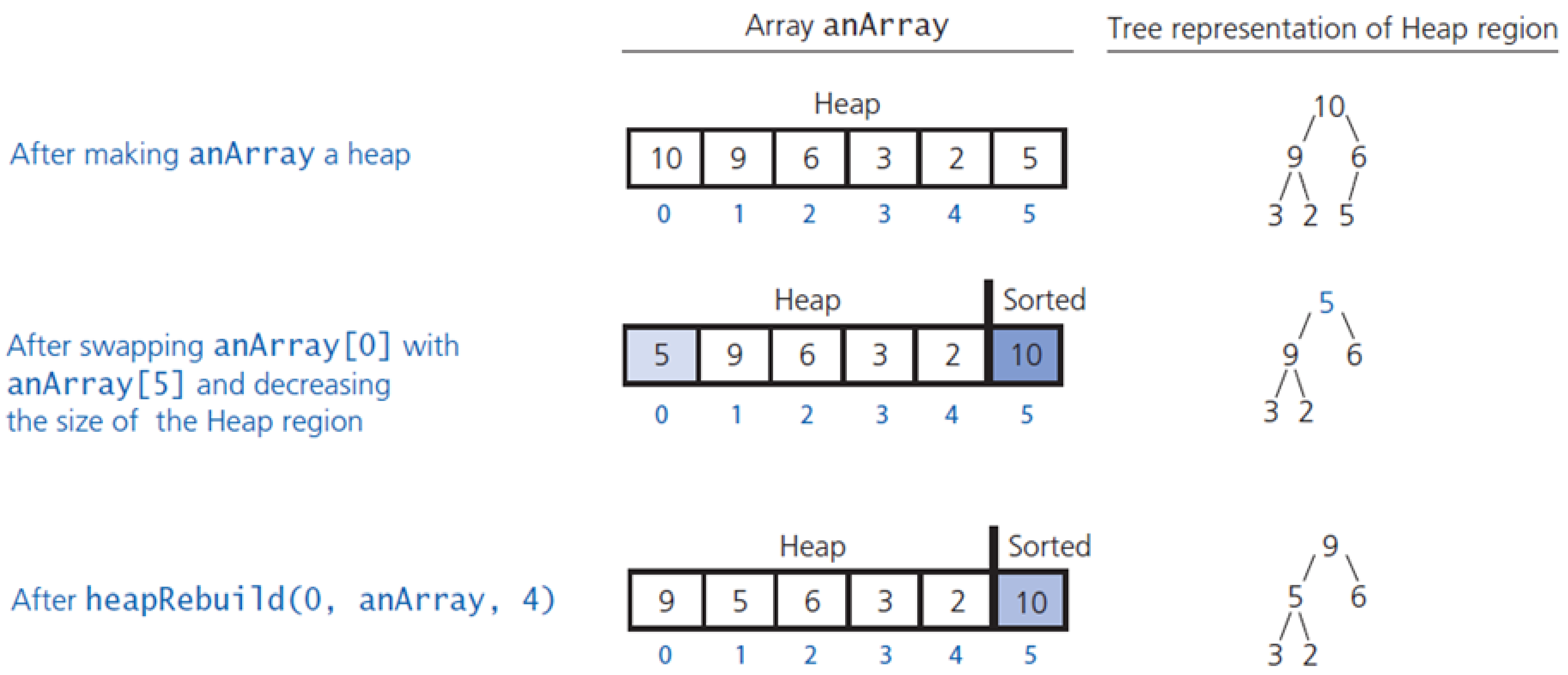
(N/2) * O(log N) compares to create original heap

(N-1) * O(log N) compares for the sorting loop

= O (N * log N) compares total

(reheap takes O(log N))

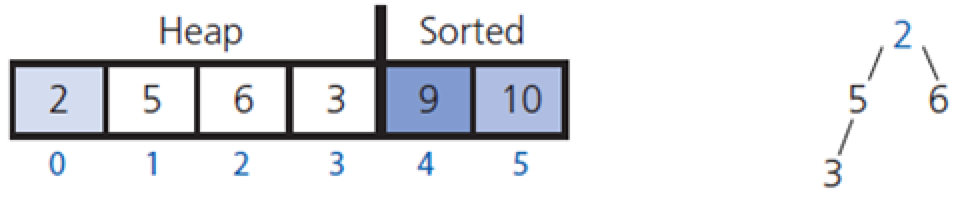
Heap Sort


Array anArray
Tree representation of Heap region
After making anArray a heap
Heap
10 9 6 3 2 5
0 1 2 3 4 5
After swapping anArray[0] with anArray[5] and decreasing the size of the Heap region
Heap
Sorted
5 9 6 3 2 10
After heapRebuild(0, anArray, 4)
9 5 6 3 2 10

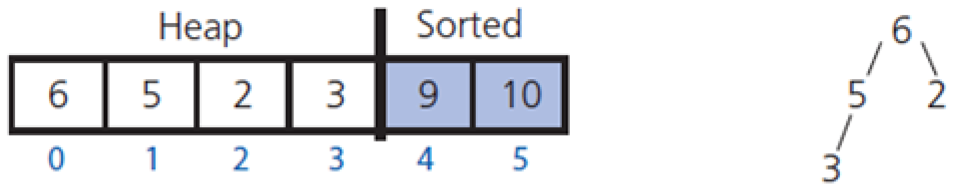
- A trace of heap sort, beginning with the heap

Heap Sort

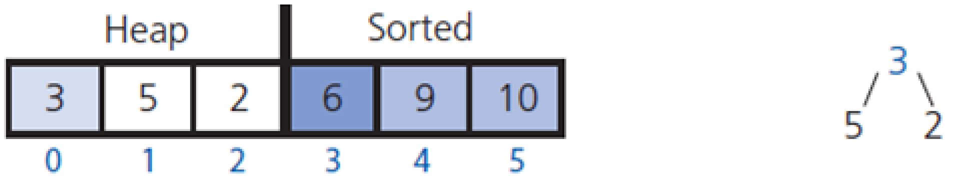
After swapping `anArray[0]` with `anArray[4]` and decreasing the size of the Heap region



After `heapRebuild(0, anArray, 3)`

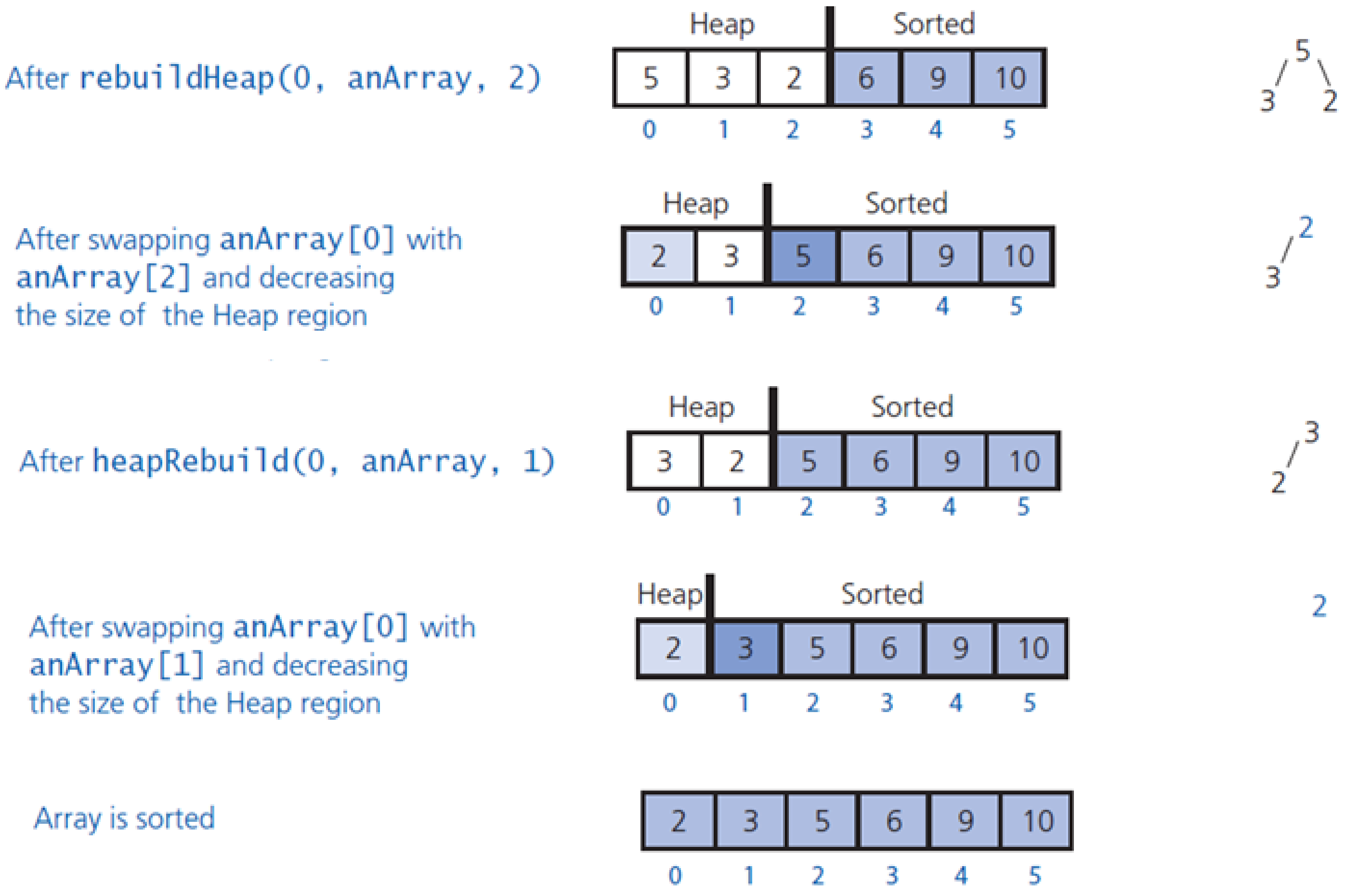


After swapping `anArray[0]` with `anArray[3]` and decreasing the size of the Heap region



- A trace of heap sort, beginning with the heap

Heap Sort



After rebuildHeap(0, anArray, 2)
Heap
Sorted
5 3 2 6 9 10
0 1 2 3 4 5
5
3 2
After swapping anArray[0] with anArray[2] and decreasing the size of the Heap region
Heap
Sorted
2 3 5 6 9 10
0 1 2 3 4 5
2
3
After heapRebuild(0, anArray, 1)
Heap
Sorted
3 2 5 6 9 10
0 1 2 3 4 5
3
2
After swapping anArray[0] with anArray[1] and decreasing the size of the Heap region
Heap
Sorted
2 3 5 6 9 10
0 1 2 3 4 5
2
Array is sorted
2 3 5 6 9 10
0 1 2 3 4 5

- A trace of heap sort, beginning with the heap