EECE 2560:
Fundamentals of Engineering Algorithms

Heaps

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

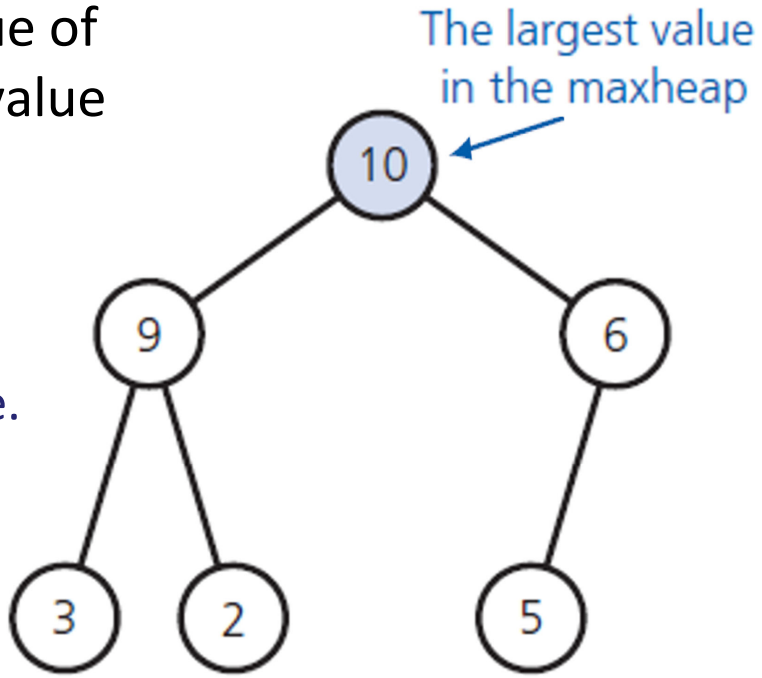
Sorting Algorithms Comparison

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Merge sort	$n \times \log n$	$n \times \log n$
Quick sort	n^2	$n \times \log n$
Radix sort	n	n
Tree sort	n^2	$n \times \log n$
Heap sort	$n \times \log n$	$n \times \log n$

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

The ADT Heap

- A heap is a **complete** binary tree.
  - A complete binary tree of height *h* is full down to level *h* – 1, with level *h* filled in from left to right
- A ***maxheap*** is a heap where the value of any node in the tree is at most the value of its parent.
- A ***minheap*** is a heap where the value of any node in the tree is at least the value of its parent.
- Heaps are special binary trees that are different in that:
  - They are ordered in a weaker sense.
  - They will always be complete binary trees.



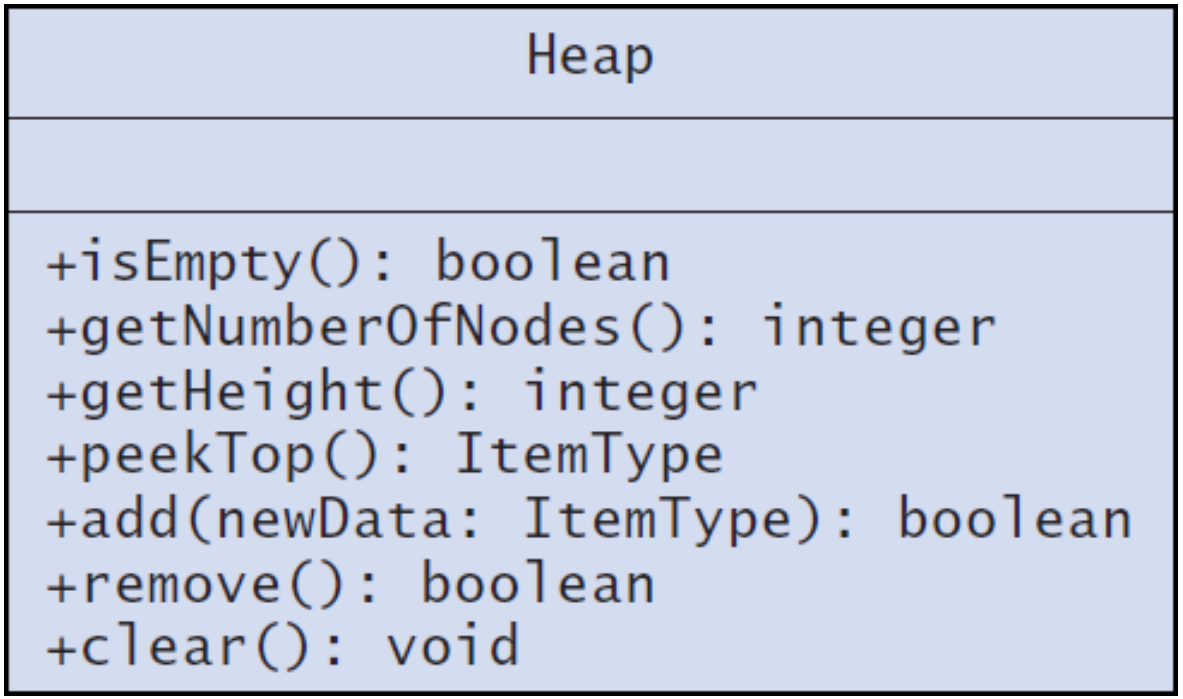
©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Heap Tree vs. Heap Memory Segment

- Do not confuse the ADT heap with the memory segment known as a heap.
- That memory is available for allocation to your C++ program when you use the **new** operator.
- The heap that contains this available memory is not an instance of the ADT heap.

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

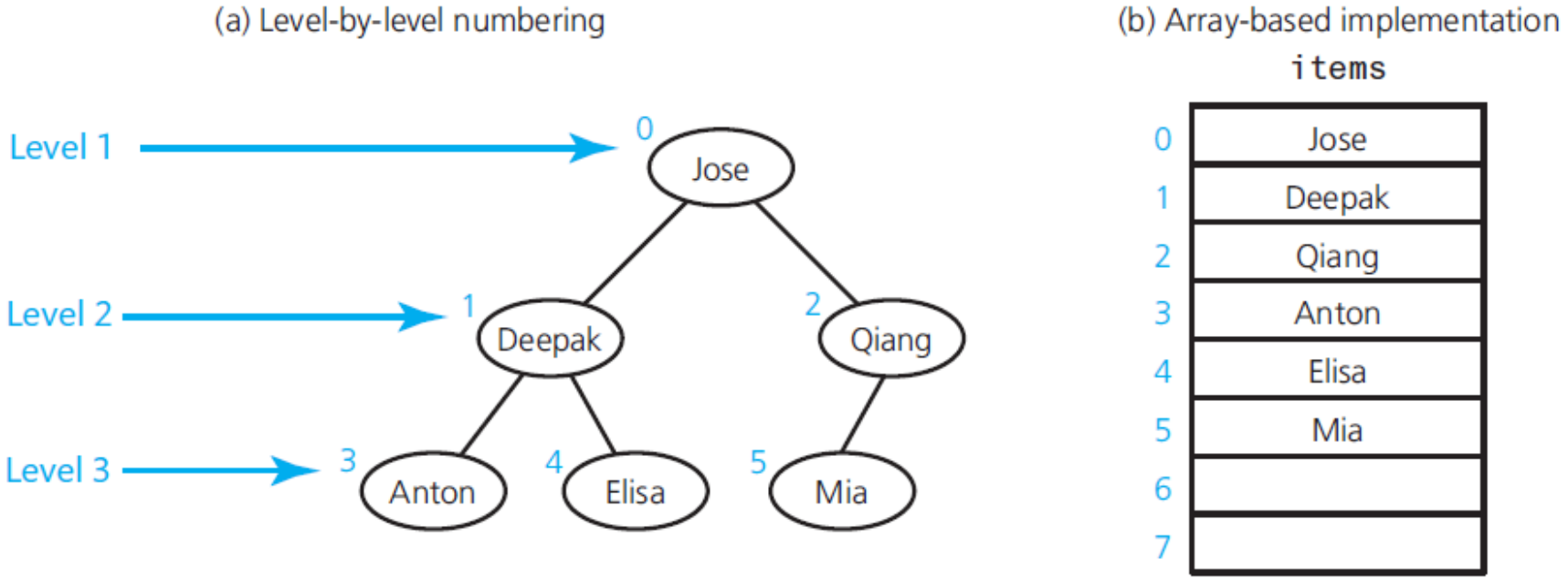
Class Heap UML Diagram



- **`peekTop`**: Get the item in the heap's root
- **`add`**: Insert a new item into the heap
- **`remove`**: Remove the item in the heap's root

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Array-Based Implementation

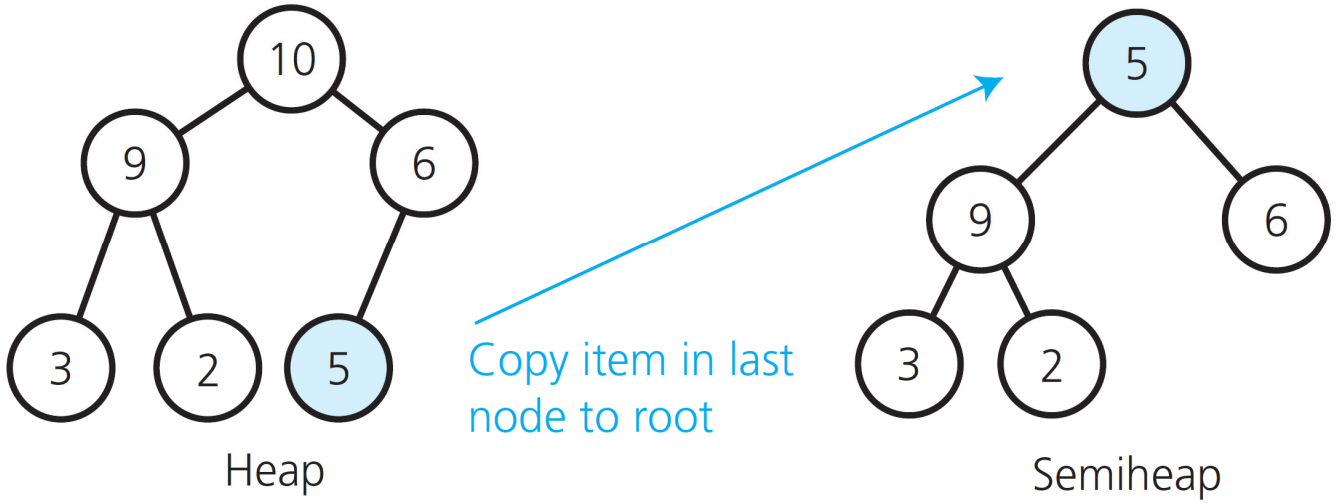


- The root of the tree is `items[0]`, and given the index *i* of a node, we can compute the indices of its parent, left child, and right child as:
  - Parent of `items[i]`: `items[|(i-1)/2|]`
  - Left child of `items[i]`: `items[2×i + 1]`
  - Right child of `items[i]`: `items[2×i + 2]`
- A heap with *n* nodes, the leaves are the nodes indexed by `⌊n/2⌋,…, n-1`. These nodes are the nodes with no children.

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Removing the Maximum Value

- While the peekTop operation returns the largest item in the maxheap – which is in its root- the heap operation remove must remove it.
- Removing the root of the maxheap leaves two disjoint heaps. Therefore, you do not want to actually remove the root. Instead, you remove the last node of the tree and place its item in the root, as shown.
- The only problem is that the item in the root usually is out of place (it is the only node with a value not larger than its children). Therefore, the resulting heap is not a ***maxheap*** but it is called a ***semiheap*** .



©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Algorithm to Remove the Root

- Assume the 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

```
// Copy the item from the last node and
// place it into the root
items[0] = items[itemCount - 1]

// Remove the last node
itemCount --

// Transform the semiheap back into a heap
heapRebuild(0, items, itemCount)
```

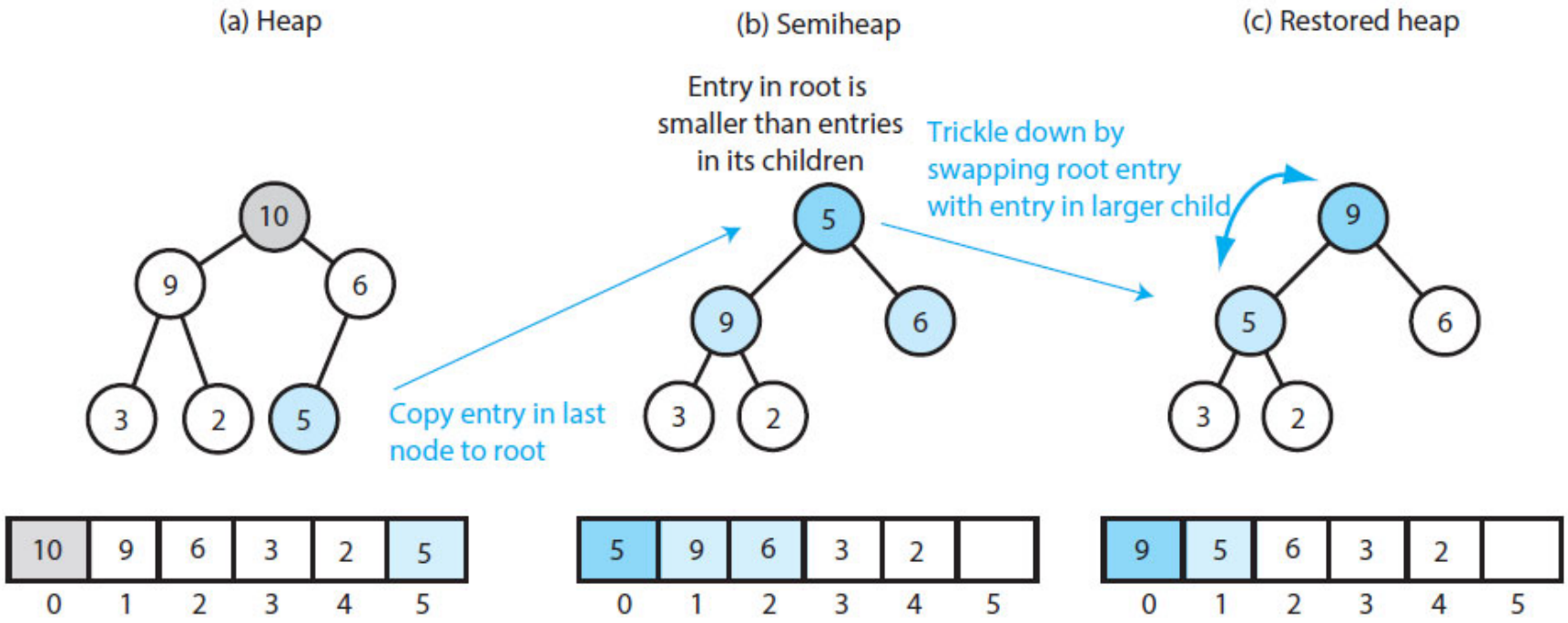
©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Transform Semiheap to Maxheap

```
// Converts a semiheap rooted at index root into a maxheap.
heapRebuild(root: integer, items: ArrayType, itemCount: integer)
// Recursively trickle the item at index root 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; assume it is the larger child
  largerChildIndex = 2 * rootIndex + 1 // Left child index
  if ( the root has a right child) {
    rightChildIndex = 2 * rootIndex + 2 // Right child index
    if (items[rightChildIndex] > items[largerChildIndex ])
      largerChildIndex = rightChildIndex // Larger child index
  }
// If the item in the root is smaller than the item in the larger child, swap items
  if (items[rootIndex] < items[largerChildIndex]) {
    Swap items[rootIndex] 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
```

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Semiheap to Maxheap Example



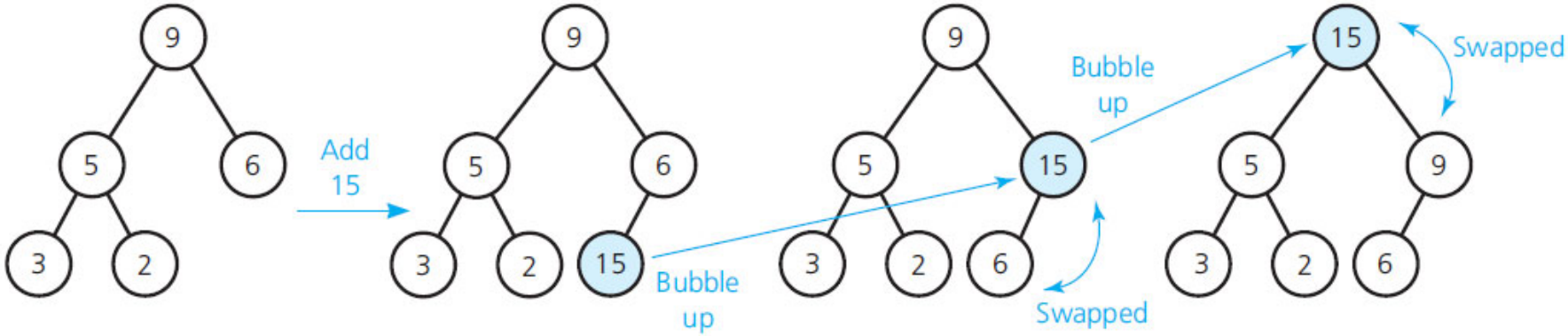
©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Algorithm to Add a New Item

```
add(newData: itemType) :boolean
{
  if (itemCount == maxCapacity) return false //no space available
   //Place newData at the bottom of the tree
  items[itemCount] = newData
  // Bubble new item 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 true
}
```

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Example of Adding an Item



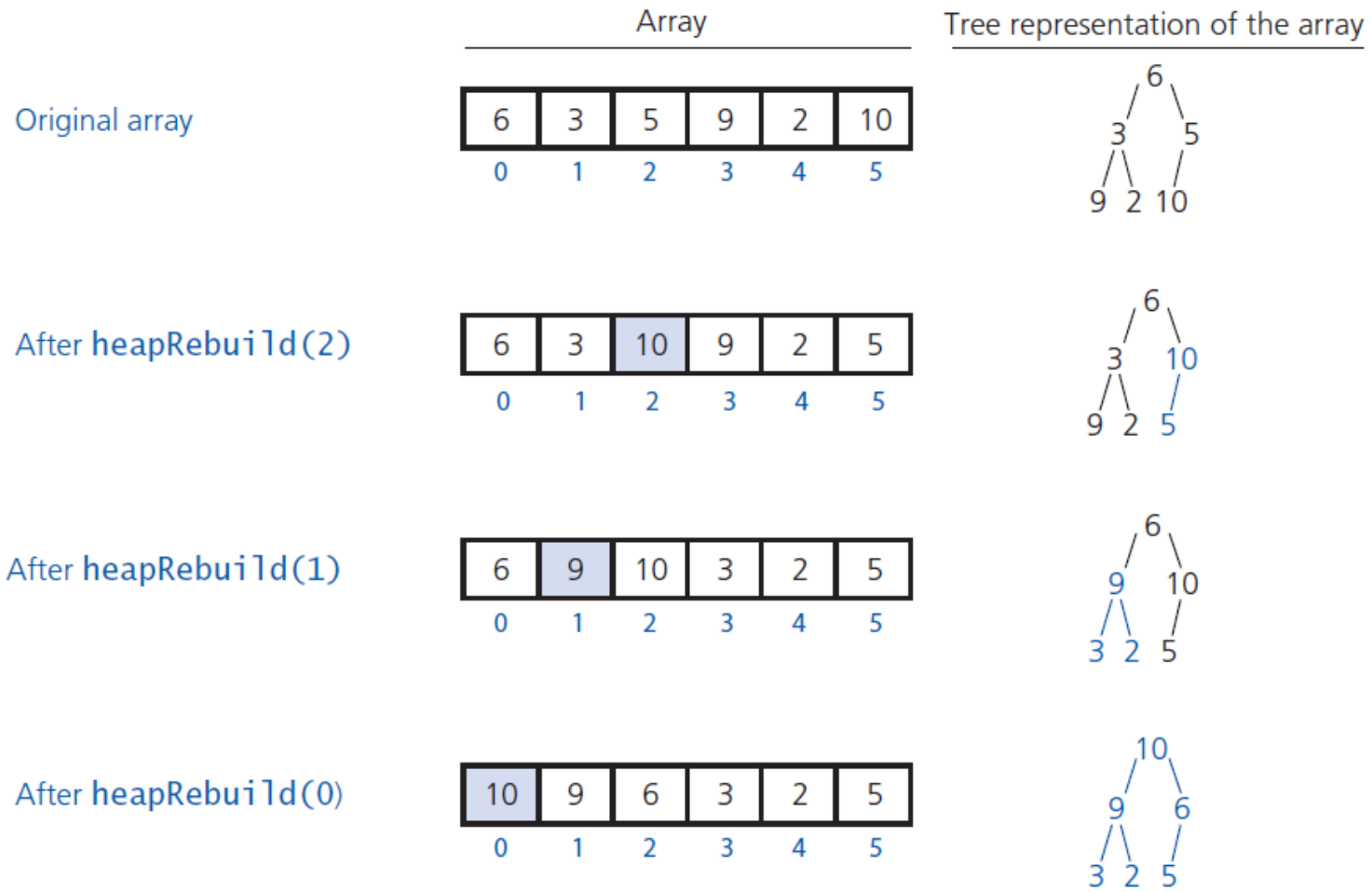
©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Transform An Array into a Heap

```
// Converts array content into a heap.
heapCreate(items: ArrayType, itemCount: integer)
{
  // Rebuild the heap starting from the last non-leaf node
 // Recall the index of the first leaf is ⌊n/2⌋
   for ( int index = (itemCount/2)-1; index >= 0; index--)
     heapRebuild(index, items, itemCount);
} // end heapCreate
```

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

An Array to Heap Example

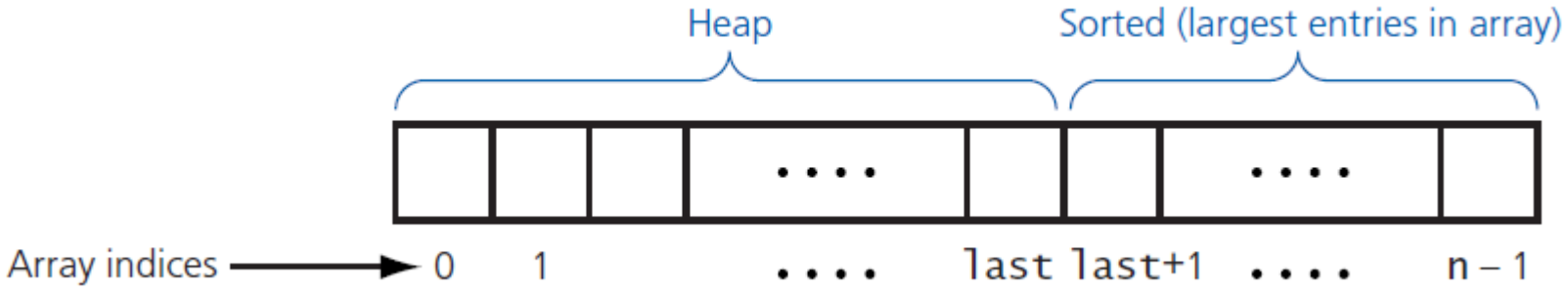


The last array is a heap but it is not sorted !!

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

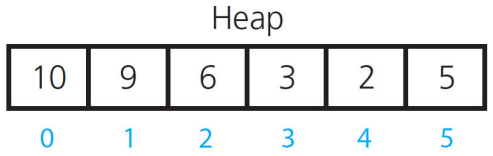
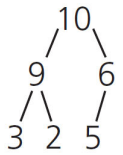
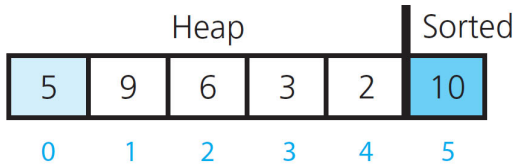
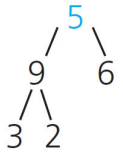
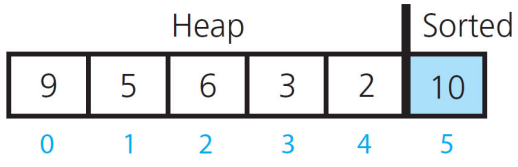
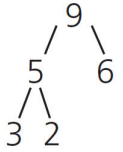
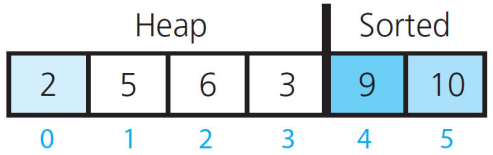
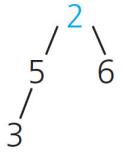
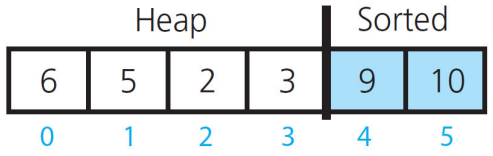
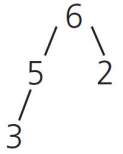
Heap Sort

The idea of Heap sort is to partition an array into two regions as shown:



©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

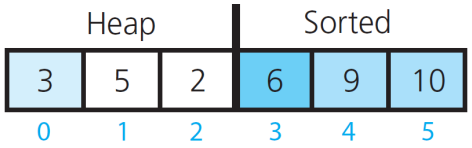
Heap Sort Example (1 of 2)

	Array `anArray`	Tree representation of Heap region
After making `anArray` a heap		
After swapping `anArray[0]` with `anArray[5]` and decreasing the size of the Heap region		
After `heapRebuild(0, anArray, 4)`		
After swapping `anArray[0]` with `anArray[4]` and decreasing the size of the Heap region		
After `heapRebuild(0, anArray, 3)`		

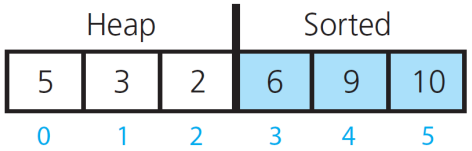
©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Heap Sort Example (2 of 2)

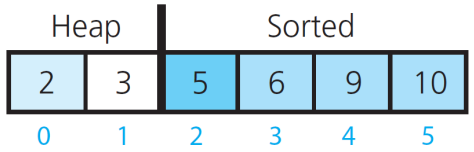
After swapping `anArray[0]` with `anArray[3]` and decreasing the size of the Heap region



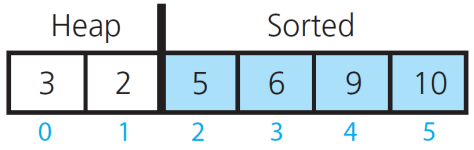
After `rebuildHeap(0, anArray, 2)`



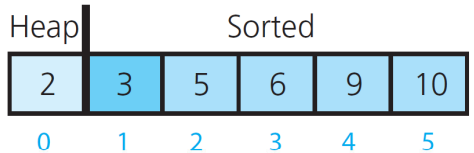
After swapping `anArray[0]` with `anArray[2]` and decreasing the size of the Heap region



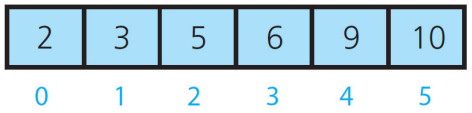
After `heapRebuild(0, anArray, 1)`



After swapping `anArray[0]` with `anArray[1]` and decreasing the size of the Heap region



Array is sorted



©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Heap Sort Algorithm

```
// Sorts anArray[0..n-1].
heapSort(anArray: ArrayType, n: integer)
 heapSize = n;

 // Converts array content into a heap.
 heapCreate(anArray, heapSize)

 // anArray[0] is the largest item in heap anArray[0.. heapSize-1]
 // Swap it with the end of the heap and decrease the heap size
 // Repeat the same process until the heap size is 1
 while (heapSize > 1) {
  Swap anArray[0] and anArray[heapSize - 1]
  heapSize--
  heapRebuild(0, anArray, heapSize)
}
```

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Heap Sort Analysis

- `heapRebuild()` is O(log *n*) and it is repeated *n* times.
- The efficiency of Heap sort is similar to that of Merge sort as both algorithms are O(*n* × log *n*) in both the worst and average cases.

- The Heap sort algorithm performs its operations in place.
  - It does not require a temp array like the Merge sort algorithm.

©2017 Pearson, F. Carrano & T. Henry: "Data Abstraction & Problem Solving with C++"

Implementing Priority Queue with Heap

- Recall that a *Priority Queue* can be implemented as a *sorted list* where we need the sort to be based on the priorities of the entries.
- Using a heap to implement a priority queue results in a more time-efficient implementation.
  - *Note*: as heap is implemented with an array, you need to know, in advance, the maximum number of items in the priority queue
- The priority queues operations can be done by heap operations as follows:

Priority Queue	Heap
`isEmpty`	`isEmpty`
`enqueue`	`add`
`dequeue`	`remove`
`peekFront`	`peekTop`