CS302 - Data Structures
using C++

Topic: Heaps

Kostas Alexis

The ADT Heap

- A "**heap**" is a complete binary tree that is either:
  - Empty or
  - Whose root contains a value >= each of its children and has heaps as its subtrees

The ADT Heap

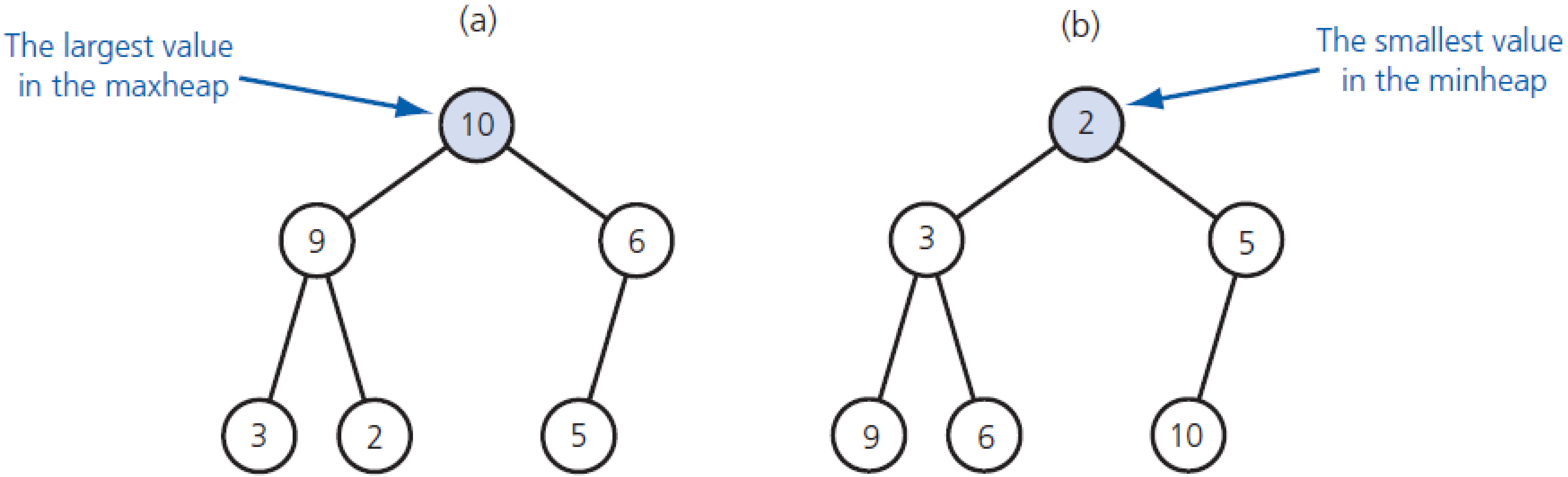
- A "**heap**" is a complete binary tree that is either:
  - Empty or
  - Whose root contains a value >= each of its children and has heaps as its subtrees
- It is a special binary tree – different from what we discussed so far in that:
  - It is ordered in a weaker sense
  - It will always be a complete binary tree

The ADT Heap

- **Maxheap:** A heap the root of which contains the item with the largest value.
- **Minheap:** a heap the root of which contains the item with the smallest value.

The ADT Heap

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



The ADT Heap

- **ADT heap** operations
  - Test whether a heap is empty
  - Get the number of nodes in a heap
  - Get the height of a heap
  - Get the data item in the heap’s root
  - Add a new data item to the heap
  - Remove the data item in the heap’s root
  - Remove all data from the heap

The ADT Heap

Abstract Data Type: Heap	
DATA	
• A finite number of objects in hierarchical order	
OPERATIONS	
PSEUDOCODE	**DESCRIPTION**
`isEmpty()`	Task: Sees whether this heap is empty. Input: None. Output: Tree if the heap is empty; otherwise false.
`getNumberOfNodes()`	Task: Gets the number of nodes in the heap. Input: None. Output: The number of nodes in the heap.
`getHeight()`	Task: Gets the height of this heap. Input: None. Output: The height of the heap.
`peekTop()`	Task: Gets the data that is in the root (top) of this heap. Input: None. Assumes the heap is not empty. Output: The data item in the root of the heap. If a maxheap, this data is the largest value. If a minheap, the data is the smallest value.
`add(newData)`	Task: Adds a new data item to this heap. Input: newData is the data item to be added. Output: True if the addition is successful, or false if not.
`remove()`	Task: Removes the data item in the root of this heap. Input: None. Output: True if the removal is successful, false otherwise
`clear()`	Task: Removes all data from this heap. Input: None. Output: The heap is empty.

The ADT Heap

- UML diagram for the class Heap

UML Notation
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

```
// Interface for the ADT heap

#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 getHeight() const = 0;
```

The ADT Heap

- An interface for the ADT heap

```
// 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.
// @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 otherwise
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;
```

The ADT Heap

- An interface for the ADT heap

```
    // 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 heap is a binary tree.

Array-based Implementation of a Heap

- A heap is a binary tree.
- Any approach to implement a binary tree can be used to implement the heap.

Array-based Implementation of a Heap

- A heap is a binary tree.
- Any approach to implement a binary tree can be used to implement the heap.
- Among others one can use the array-based implementation of the binary tree if you know the maximum size of a heap.

Array-based Implementation of a Heap

- A heap is a binary tree.
- Any approach to implement a binary tree can be used to implement the heap.
- Among others one can use the array-based implementation of the binary tree if you know the maximum size of a heap.
- **However**
    - **Because a heap is a complete binary tree, we can use a simpler array-based implementation that saves memory.**

Array-based Implementation of a Heap

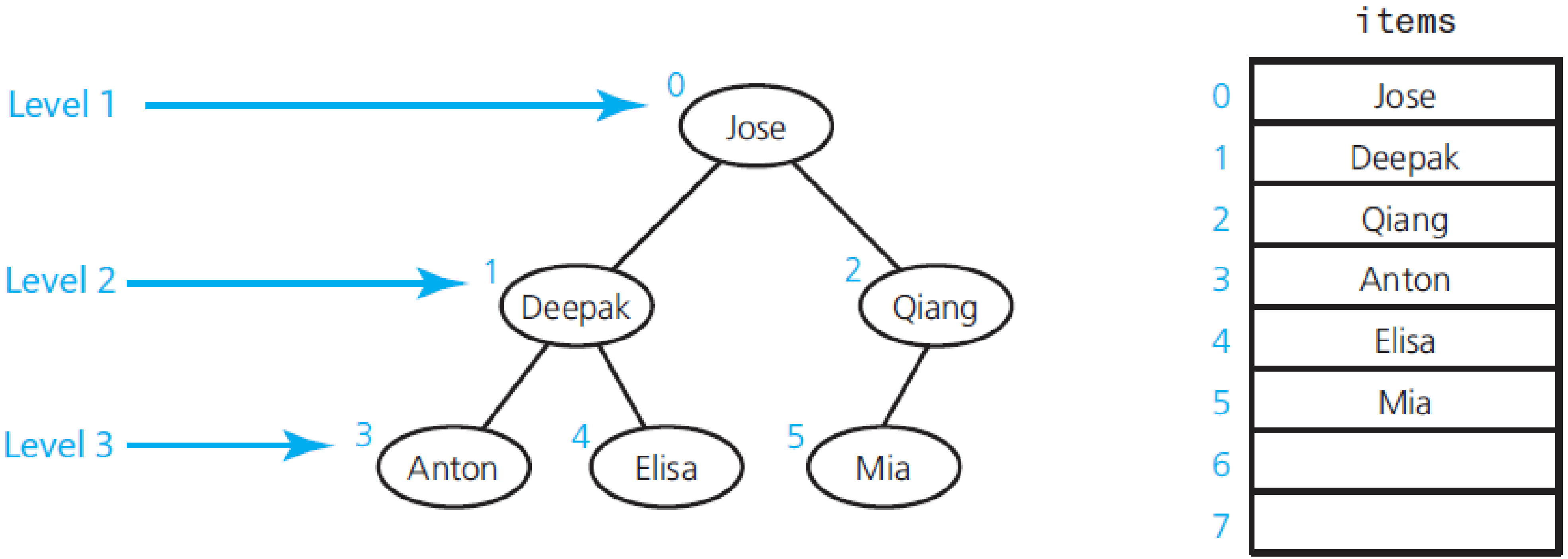
- A heap is a binary tree.
- Any approach to implement a binary tree can be used to implement the heap.
- Among others one can use the array-based implementation of the binary tree if you know the maximum size of a heap.
- **However**
  - **Because a heap is a complete binary tree, we can use a simpler array-based implementation that saves memory.**
- **Reminder**
  - **A complete tree of height h is full to level h-1 and has level h filled from left to right.**

Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation

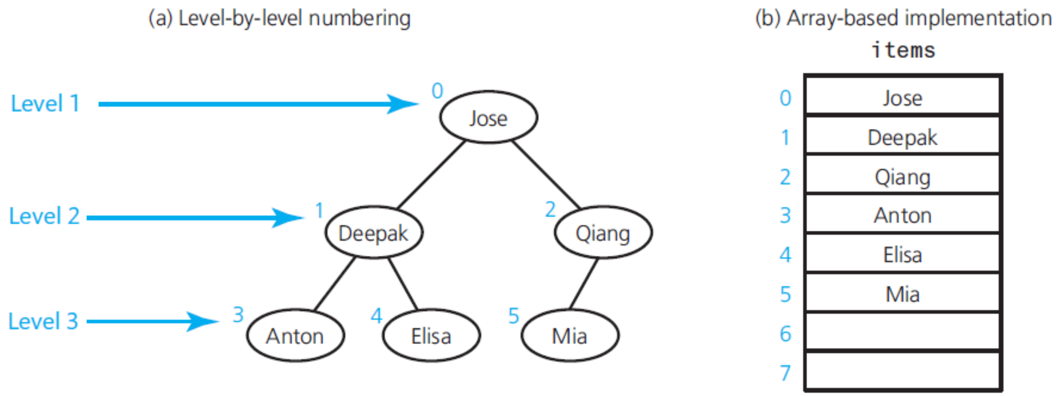
(a) Level-by-level numbering

(b) Array-based implementation



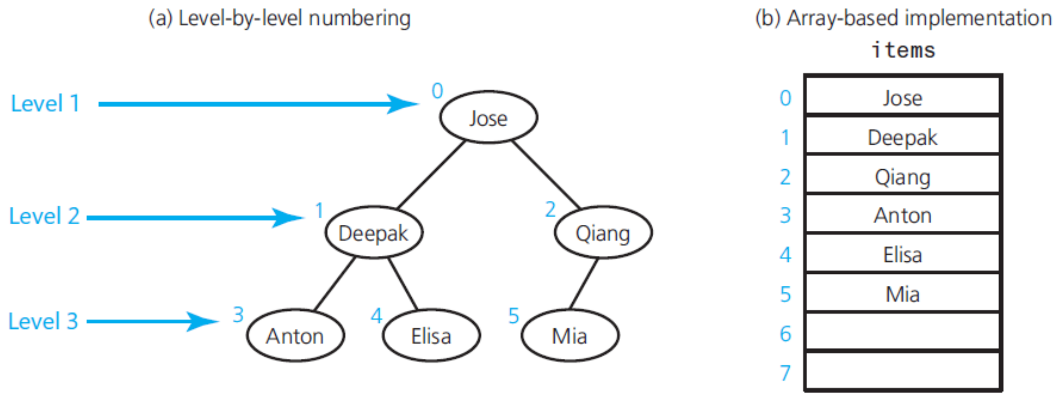
Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation
- Place nodes into array items in numeric order



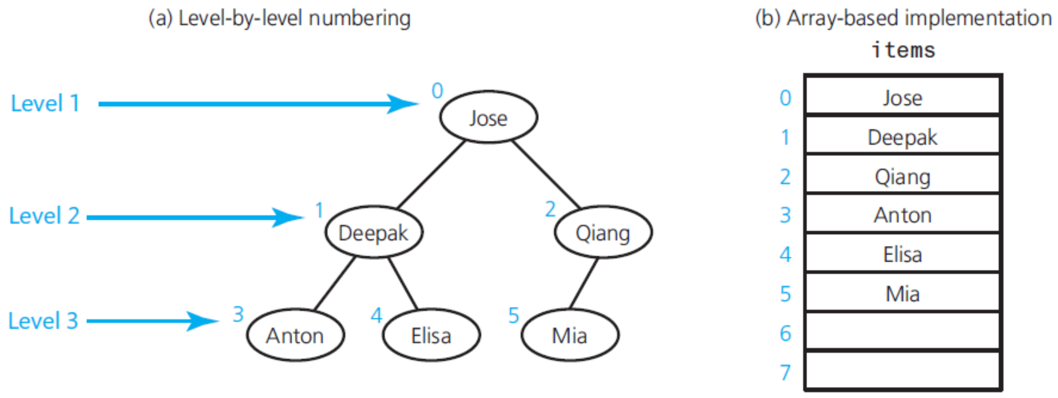
Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation
- Place nodes into array items in numeric order
  - `items[i]` contains the node numbered i



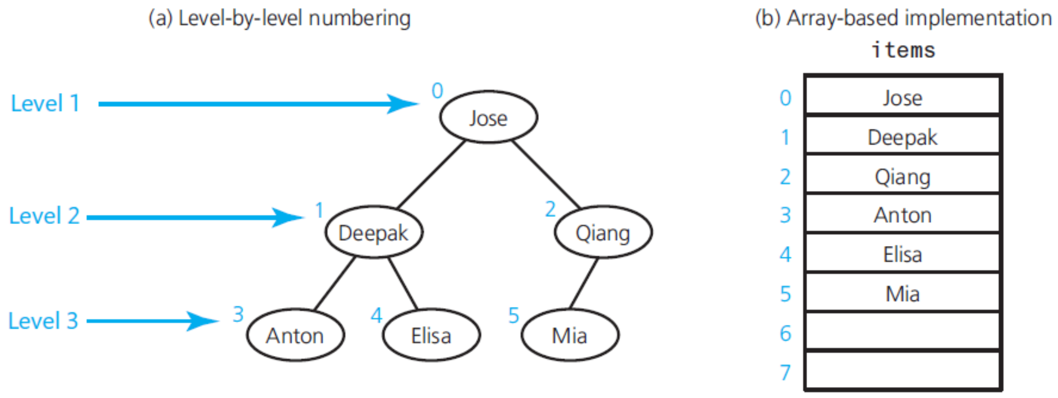
Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation
- Place nodes into array items in numeric order
  - `items[i]` contains the node numbered i
- **Localization of parent & children of `items[i]`**
  - Left child, if exists = `items[2 * i + 1]`
  - Right child, if exists = `items[2 * i +2]`
  - Parent, if exists = `items[(i-1) / 2]`



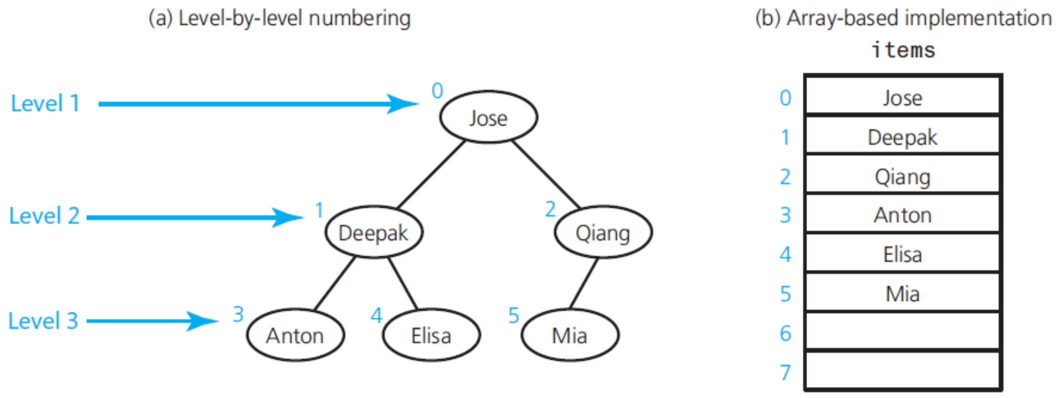
Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation
- Place nodes into array items in numeric order
  - `items[i]` contains the node numbered i
- **Localization of parent & children of `items[i]`**
  - Left child, if exists = `items[2 * i + 1]`
  - Right child, if exists = `items[2 * i +2]`
  - Parent, if exists = `items[(i-1) / 2]`
- Only node without a parent is the root (`items[0]`)



Array-based Implementation of a Heap

- A complete binary tree and its array-based implementation
- Place nodes into array items in numeric order
  - `items[i]` contains the node numbered i
- **Localization of parent & children of `items[i]`**
  - Left child, if exists = `items[2 * i + 1]`
  - Right child, if exists = `items[2 * i +2]`
  - Parent, if exists = `items[(i-1) / 2]`
- Only node without a parent is the root (`items[0]`)
- **This numbering requires a complete tree**



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

- 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
- The array **items** corresponds to the array-based representation of a complete binary tree

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
- The array **items** corresponds to the array-based representation of a complete binary tree
- We assume that we are working with a maximum heap of integers

Algorithms for Array-based Heap Operations

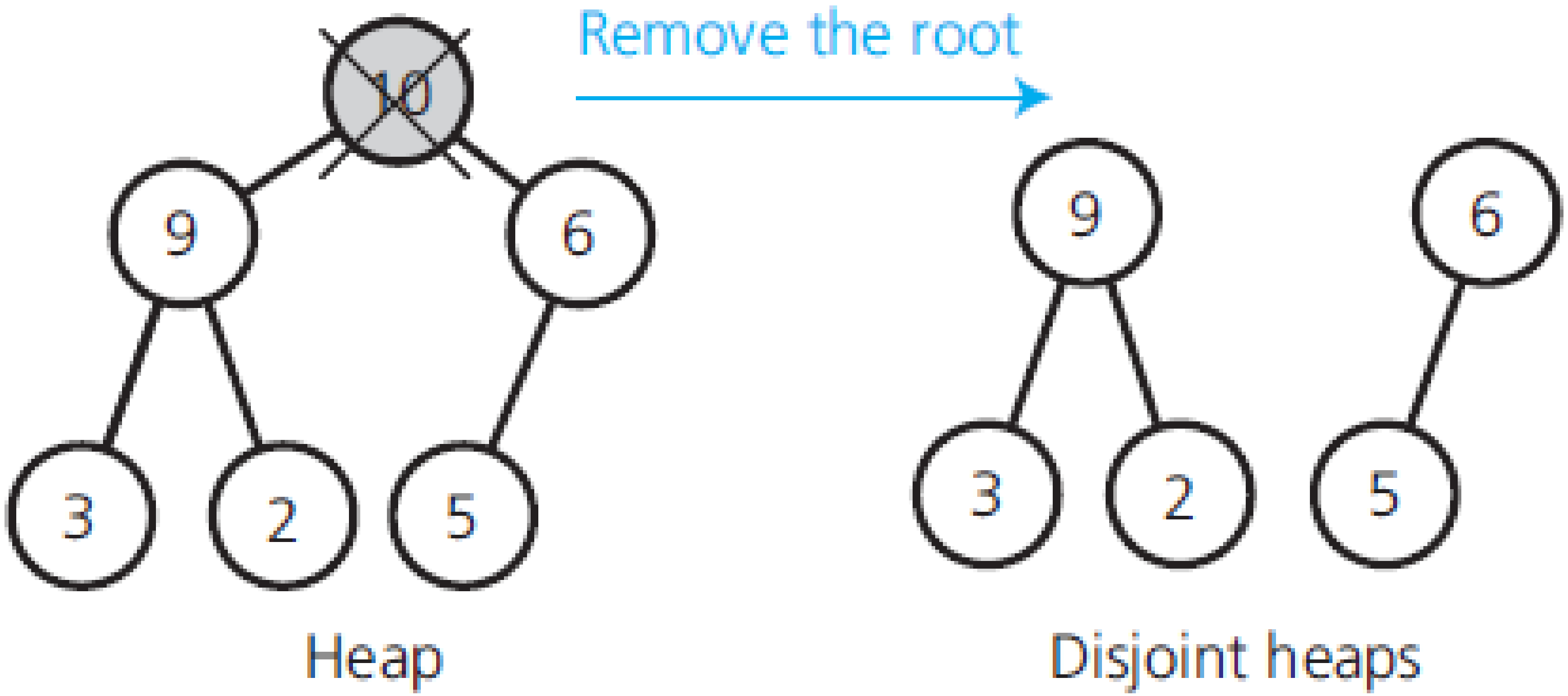
- Retrieving an item from the heap.
  - Consider peekTop operation
  - Largest value at the root
  - Therefore: **`return`** `items[0]`

Algorithms for Array-based Heap Operations

- Removing an item from a heap
  - Consider the case of root removal

Algorithms for Array-based Heap Operations

- Disjoint heaps after removing the heap's root



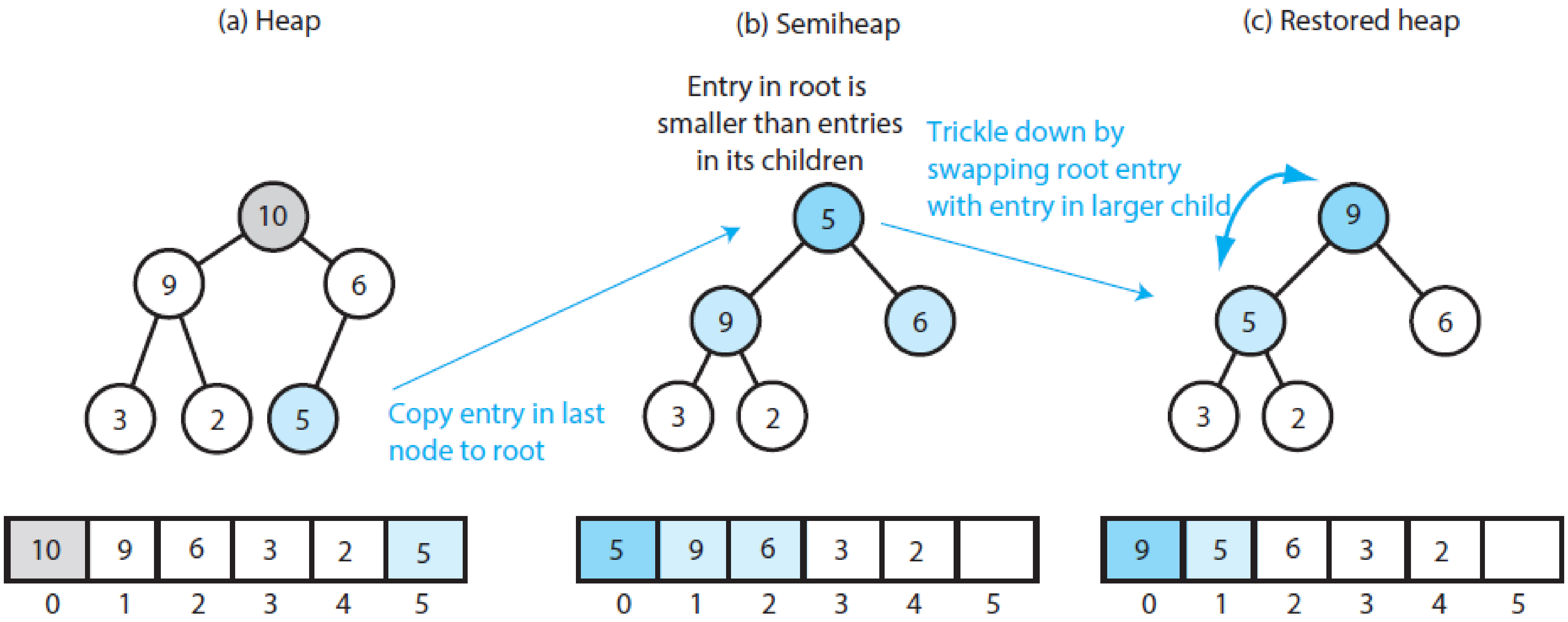
Algorithms for Array-based Heap Operations

- Removing an item from a heap
  - Consider the case of root removal
  - Naïve root removal leads to disjoint heaps

Algorithms for Array-based Heap Operations

- Removing an item from a heap
  - Consider the case of root removal
  - Naïve root removal leads to disjoint heaps
  - **Solution**
    - Remove the last node of the tree and place its item in the root
    - This is not necessarily a heap but it remains a complete binary tree whose left and right subtrees are both heaps
      - This is a **semiheap**
    - We need to transform a semiheap into a heap.
      - Allow the item in the root to **trickle down** (bubble down) the tree until it reaches a node in which it will not be out of place.
      - First compare the item in the root of the **semiheap** to the items in its children.
      - **Swap** the item in the root with that larger item if one is found.
      - Iterate this process.

Algorithms for Array-based Heap Operations



Algorithms for Array-based Heap Operations

- We need to implement two steps, namely root removal and semiheap to heap restoration

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

/ Remove the last node
itemCount--
```

Algorithms for Array-based Heap Operations

- Recursive algorithm to transform semiheap to heap

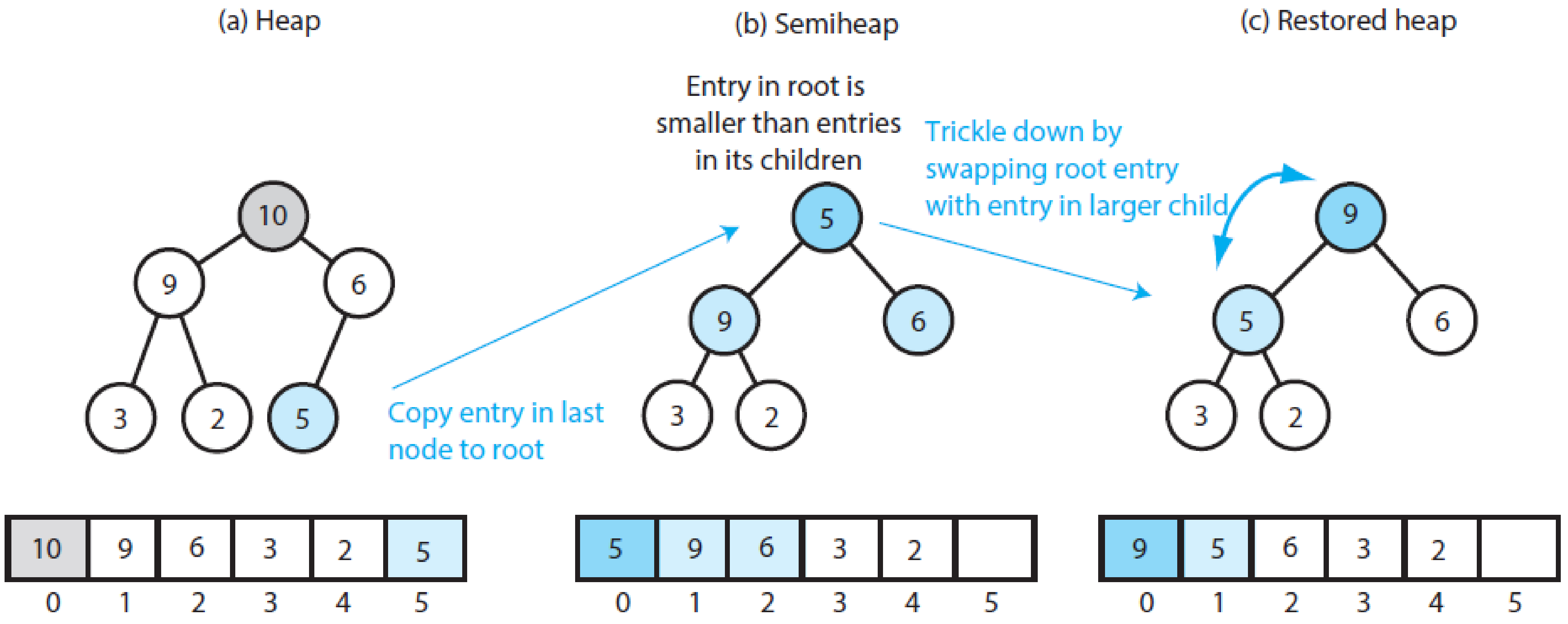
```
// Converts a semiheap rooted at index nodeIndex into a heap.
heapRebuild(nodeIndex: integer, items: ArrayType, itemCount: integer): void
{
    // Recursively trickle the item at index nodeIndex down to its proper position by
    // swapping it with its larger child, if the child is larger than the 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
```

Algorithms for Array-based Heap Operations

- Recursive algorithm to transform semiheap to heap

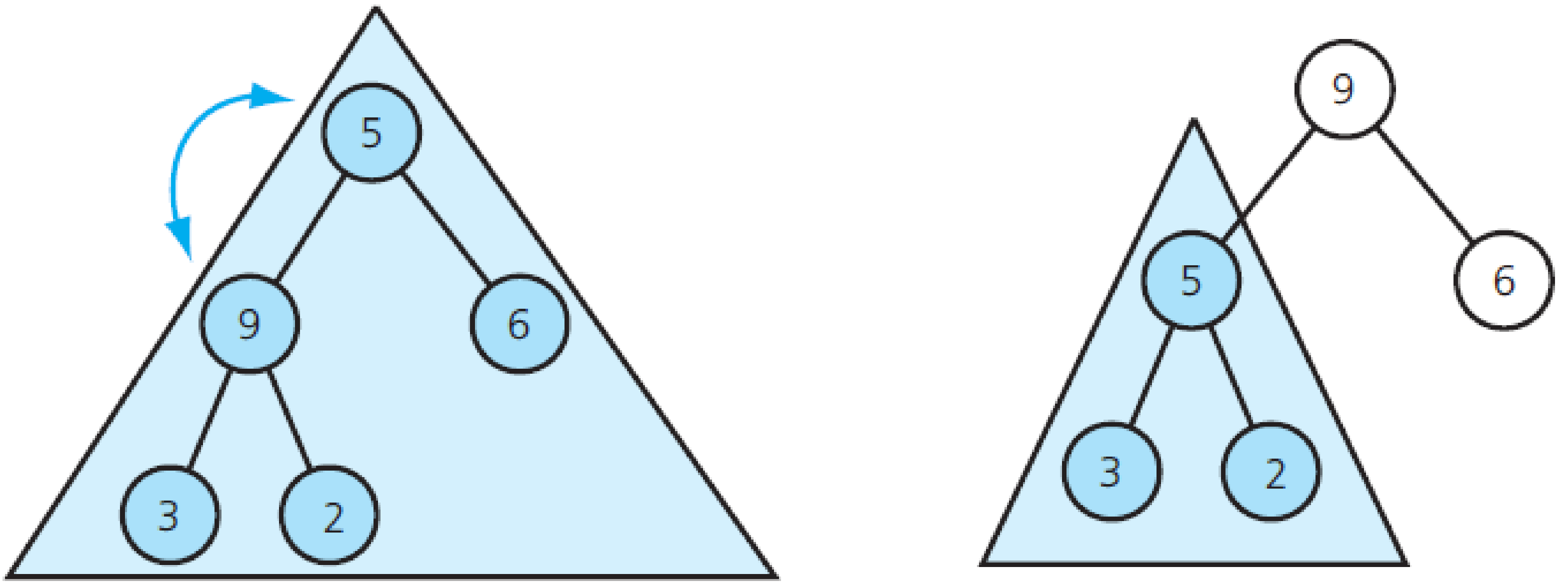
```
        // Check whether right child exists; if so, is left child larger?
        // If no right child, left one is larger
        if ((largerChildIndex >= itemCount) || (items[leftChildIndex] > items[rightChildIndex]))
                largerChildIndex = leftChildIndex; // Assumption was wrong
 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



Algorithms for Array-based Heap Operations

- Recursive calls to **heapRebuild**



First semiheap passed to `heapRebuild`

Second semiheap passed to `heapRebuild`

Algorithms for Array-based Heap Operations

- Finally, heap's remove operation using heapRebuild is as follows

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

// Remove the last node
itemCount--

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

Algorithms for Array-based Heap Operations

- Remove is O(logn)

Algorithms for Array-based Heap Operations

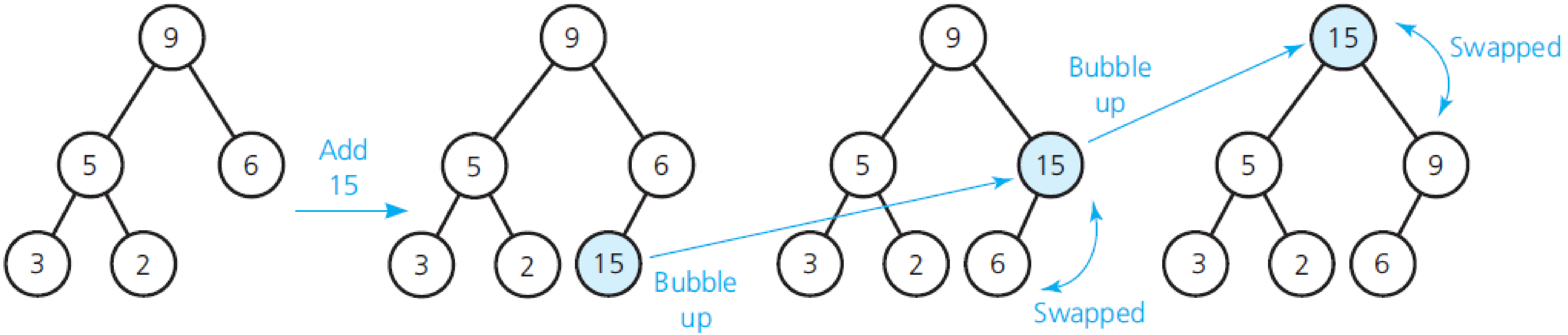
- Adding a data item to a heap

Algorithms for Array-based Heap Operations

- Adding a data item to a heap
  - A new data item is placed at the bottom of the tree
  - It then **bubbles up** to its proper place

Algorithms for Array-based Heap Operations

- Adding 15 to a heap



Algorithms for Array-based Heap Operations

- Pseudocode for add

```
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
}
```

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 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 method **getLeftChildIndex**

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

Outcome of how the heap is built

The Implementation

- Definition of the **constructor** (must use **heapRebuild**)

```
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 = 0; i < itemCount; i++)
        items[i] = someArray[i];

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

The Implementation

- Definition of the **constructor** (must use **heapRebuild**)

```
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 = 0; i < itemCount; i++)
        items[i] = someArray[i];

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

The Implementation

- Definition of the **constructor** (must use **heapRebuild**)

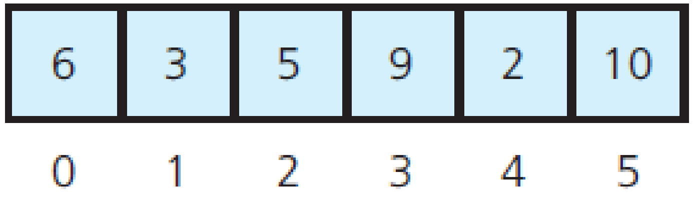
```
template<class ItemType>
ArrayMaxHeap<ItemType>::
ArrayMaxHeap(const ItemType someArray[], const int arraySize):
    itemCount(arraySize), maxItems(2 * arraySize)
{
    // Allocate the array
    items = std::make_unique<ItemType[]>(maxItems);

    // 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
```

- **heapCreate** must form a heap from the values in the array items.
- One way to do so is to use the heap's add method to add the data items to the heap one by one. But a more efficient way is possible.
- Image the array as a complete binary tree
- Transform this tree into a heap by calling **heapRebuild** repeatedly
- Observation: every leaf is a **semiheap**

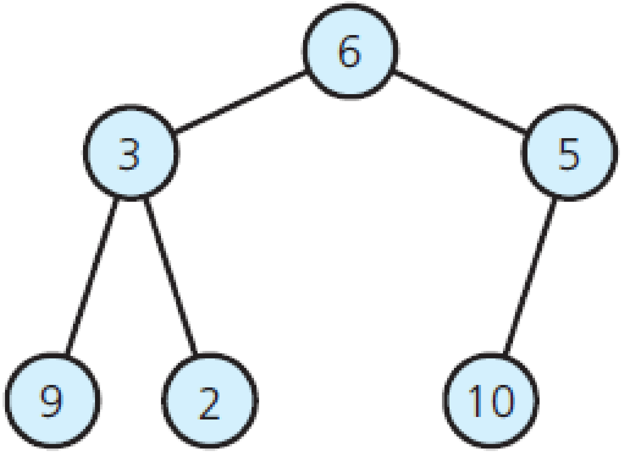
The Implementation

- Array and its corresponding complete binary tree

(a) An array of given values



(b) The complete binary tree represented by the array



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

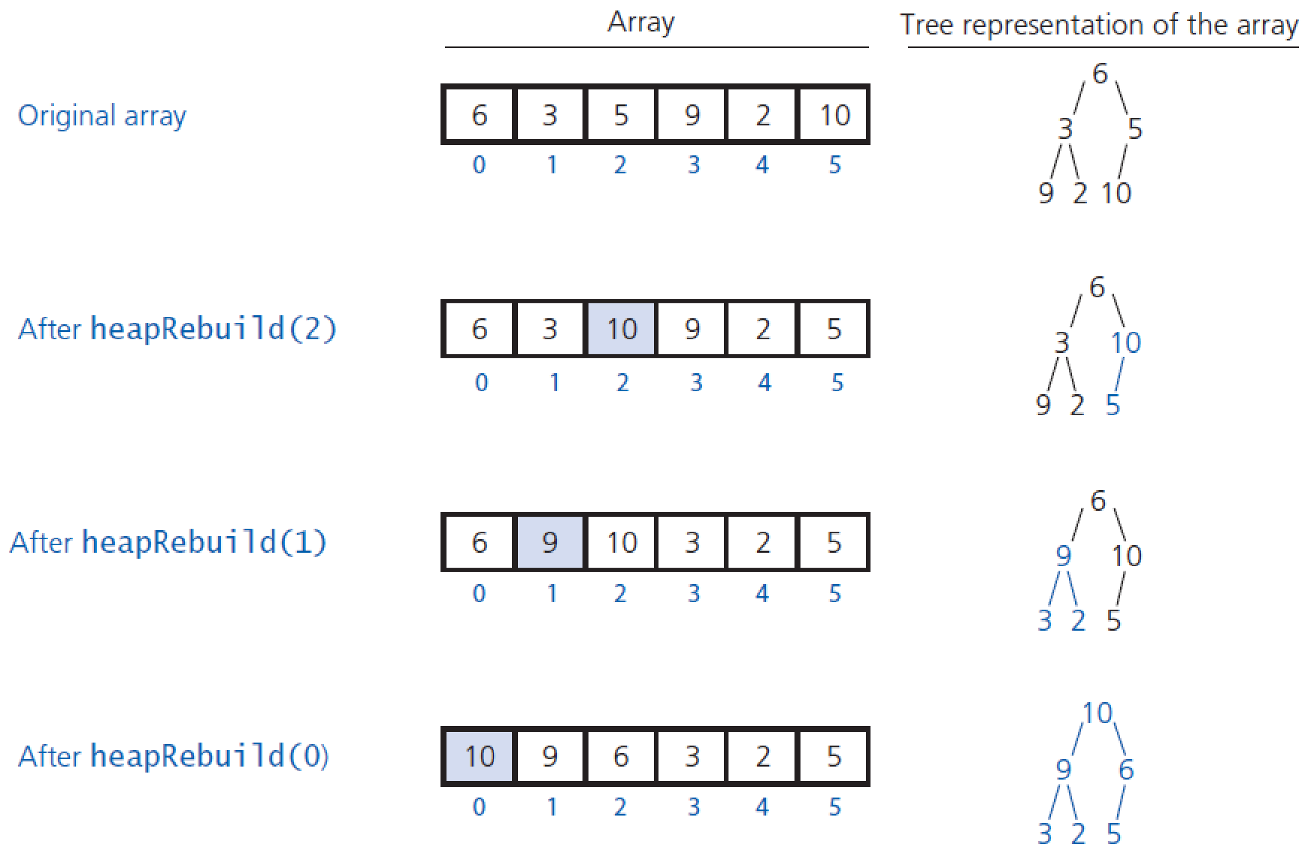
- Building a heap from an array of data: possible improvement.

```
for (index = itemCount / 2 - 1 down to 0)
{
    // Assertion: The tree rooted at index is a semiheap
    heapRebuild(index)
    // Assertion: The tree rooted at index is a heap
}
```

Why?

The Implementation

- Transforming an array into a heap



The Implementation

- Method **heapCreate**

```
template<class ItemType>
void ArrayMaxHeap<ItemType>::heapCreate()
{
    for (int index = itemCount / 2 - 1; index >= 0; index--)
        heapRebuild(index)
} // end heapCreate
```

The Implementation

- Method **peekTop** which tests for an empty heap

```
template<class ItemType>
ItemType ArrayMaxHeap<ItemType>::peekTop() const throw(PrecondViolatedExcept)
{
    if (isEmpty())
        throw PrecondViolatedExcept("Attempted peek into an empty heap.");
    return items[0];
} // end peekTop
```

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**
  - The implementation of the priority queue can reuse ArrayMaxHeap

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**
  - The implementation of the priority queue can reuse ArrayMaxHeap
  - We could use an instance of ArrayMaxHeap as a data member of the class of priority queues, or we can consider inheritance.

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**
  - The implementation of the priority queue can reuse ArrayMaxHeap
  - We could use an instance of ArrayMaxHeap as a data member of the class of priority queues, or we can consider inheritance.
  - Although a heap provides an implementation for a priority queue, a priority queue is not a heap.

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**
  - The implementation of the priority queue can reuse ArrayMaxHeap
  - We could use an instance of ArrayMaxHeap as a data member of the class of priority queues, or we can consider inheritance.
  - Although a heap provides an implementation for a priority queue, a priority queue is not a heap.
  - Since an is-a relationshops does not exist between ArrayMaxHeap and the class of priority queues, public inheritance is not appropriate.

Heap Implementation of the ADT Priority Queue

- Provided ADT heap, the implementation of the ADT Priority Queue is straightforward
- **The priority value in a priority queue item corresponds to an item in a heap.**
  - The implementation of the priority queue can reuse ArrayMaxHeap
  - We could use an instance of ArrayMaxHeap as a data member of the class of priority queues, or we can consider inheritance.
  - Although a heap provides an implementation for a priority queue, a priority queue is not a heap.
  - Since an is-a relationshops does not exist between ArrayMaxHeap and the class of priority queues, public inheritance is not appropriate.
  - **We can use private inheritance instead.**

Heap Implementation of the ADT Priority Queue

- A header file for the class **HeapPriorityQueue**

```
// ADT Priority Queue: Heap-based Implementation
#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(PrecondViolatexExcept);
}; // end HeapPriorityQueue
#endif
```

Heap Implementation of the ADT Priority Queue

- A header file for the class **HeapPriorityQueue**

```
// Heap-based implementation of the ADT priority queue.

#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
```

Heap Implementation of the ADT Priority Queue

- A header file for the class HeapPriorityQueue

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

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
```

Heap Implementation of the ADT Priority Queue

- Heap versus a Binary Search Tree
  - **If you know the maximum number of items in the priority queue, heap is the better implementation**

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 uses a heap to sort an array of items that are in no particular order.

Heap Sort

- Heap sort uses a heap to sort an array of items that are in no particular order.
- Transform the array into a heap.

```
for (int index = itemCount / 2 -1; index >=0; index--)
    heapRebuild(index)
```

- As members of the class **ArrayMaxHeap**, both **heapCreate** and **heapRebuild** have access to the class' data members, including the array items and its number of entries.
- To use **heapRebuild** in a heap sort, we must revise it so that it has the array and its size as parameters.

```
void heapRebuild(int startIndex, ItemType& anArray[], int n)
```

Heap Sort

- Heap sort uses a heap to sort an array of items that are in no particular order.
- Transform the array into a heap.

```
for (int index = itemCount / 2 -1; index >=0; index--)
    heapRebuild(index)
```

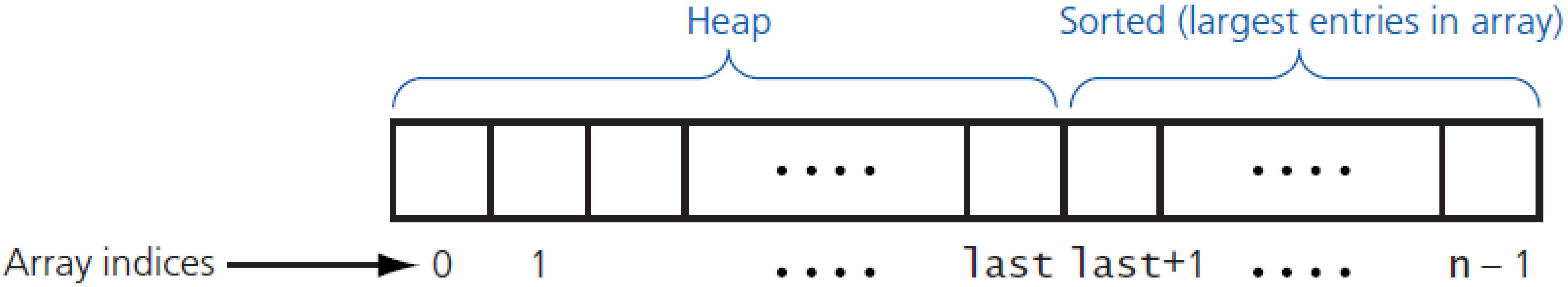
- As members of the class **ArrayMaxHeap**, both **heapCreate** and **heapRebuild** have access to the class' data members, including the array items and its number of entries.
- To use **heapRebuild** in a heap sort, we must revise it so that it has the array and its size as parameters.

```
void heapRebuild(int startIndex, ItemType& anArray[], int n)
```

- After transforming the array into a heap, **heap sort** partitions the array into two regions – the **Heap region** and the **sorted region**. Initially the heap region `anArray[0..last]` and the Sorted region is in `anArray[last+1..n-1]`
- Initially the **Heap region** is all of `anArray` and the **Sorted region** is empty.

Heap Sort

- Heap sort partitions an array into two regions



Heap Sort

- Each step of the algorithm moves an item I from the **Heap region** to the **Sorted region**. During this process, the following statements are true
  - The Sorted region contains the largest values in `anArray`, and they are in sorted order – that is, `anArray[n-1]` is the largest item, `anArray[n-2]` is the second largest, and so on.
  - The items in the heap region from a heap.
- So that the invariant holds, I must be the item that has the largest value in the **Heap region**, and therefore I must be in the root of the heap.

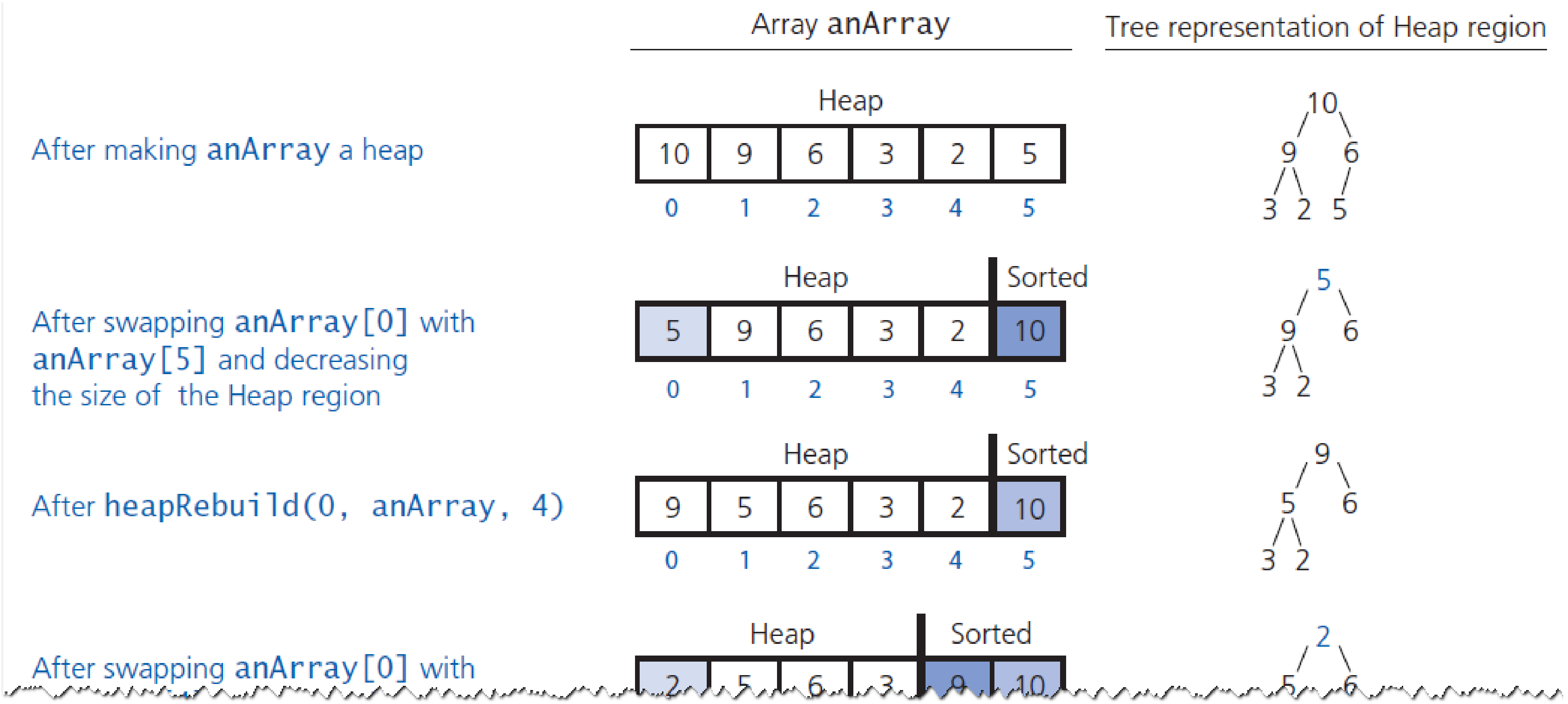
Heap sort

- Heap sort key steps

```
// Sorts anArray[0..n-1]
heapsort(anArray: ArrayType, n: integer)
{
     // Build initial heap
     for (index = n / 2 - 1 down to 0)
     {
          // Assertion: The tree rooted at index is a semiheap
          heapRebuild(index, anArray, n)
          // Assertion: The tree rooted at index is a heap
     }
     // Assertion: anArray[0] is the largest item in heap anArray[0..n-1]

     // Move the largest item in the Heap region - the root anArray[0] - to the beginning of the Sorted region
     // by swapping and then adjusting the size of the regions
     Swap anArray[0] and anArray[n-1]
     heapSize = n-1 // Decrease the size of the Heap region, expand the Sorted region
     while (heapSize > 1)
     {
          // Make the Heap region a heap again
          heapRebuild(0, anArray, heapSize)
          // Move the largest item in the Heap region - the root anArray[0] - to the beginning of the Sorted
          // region by swapping items and then adjusting the size of the regions
          Swap anArray[0] and anArray[heapSize-1]
          heapSize-- // Decrease the size of the Heap region, expand the Sorted region
     }
}
```

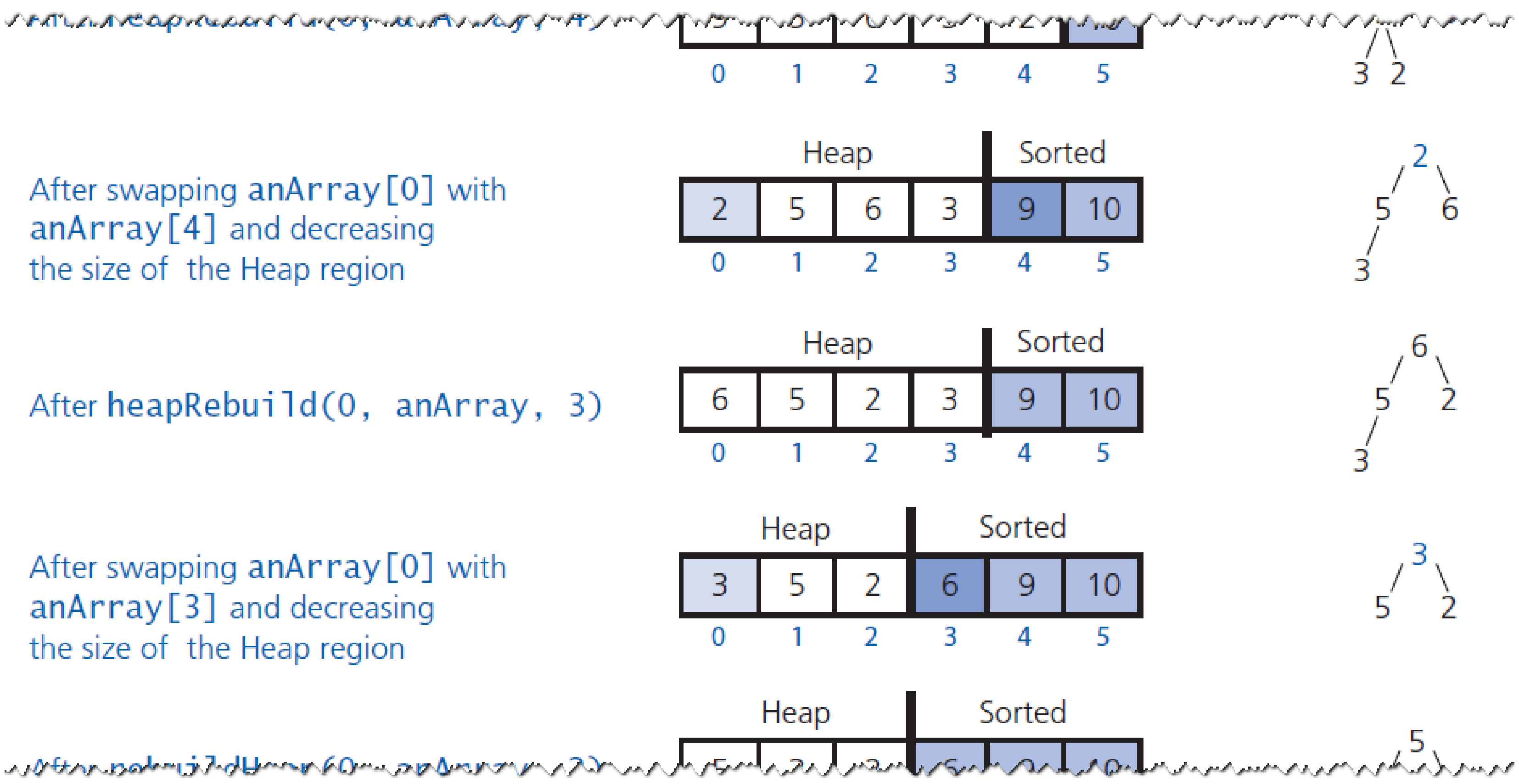
Heap Sort

- A trace of heap sort



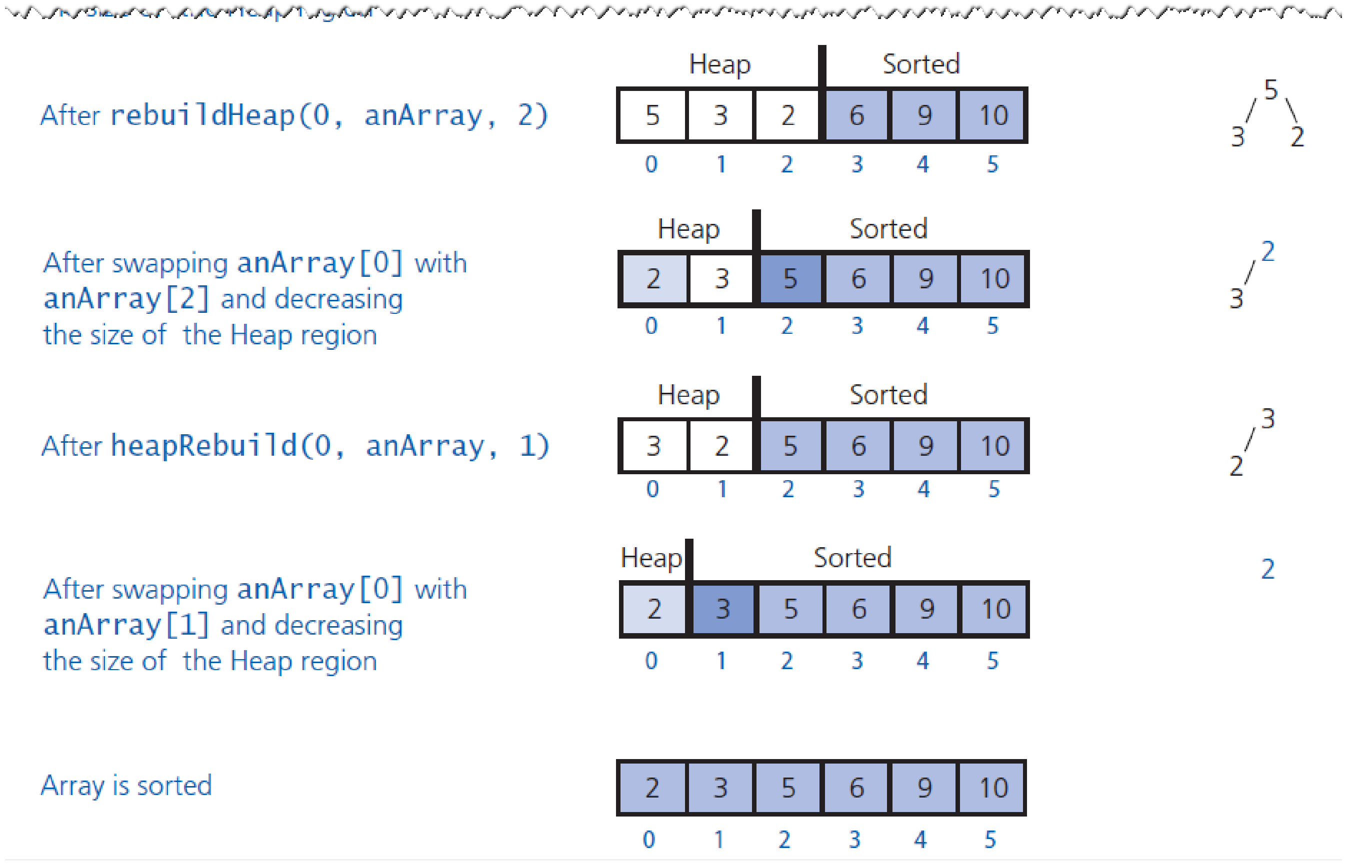
Heap Sort

- A trace of heap sort (cont)



Heap Sort

- A trace of heap sort (cont)



Thank you