Data Structures and Algorithms

HEAPS

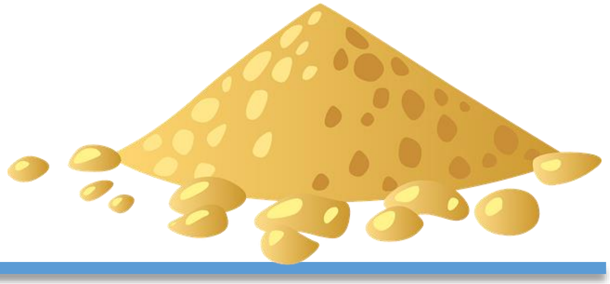
Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

Ho Chi Minh City, 05/2022

Outline

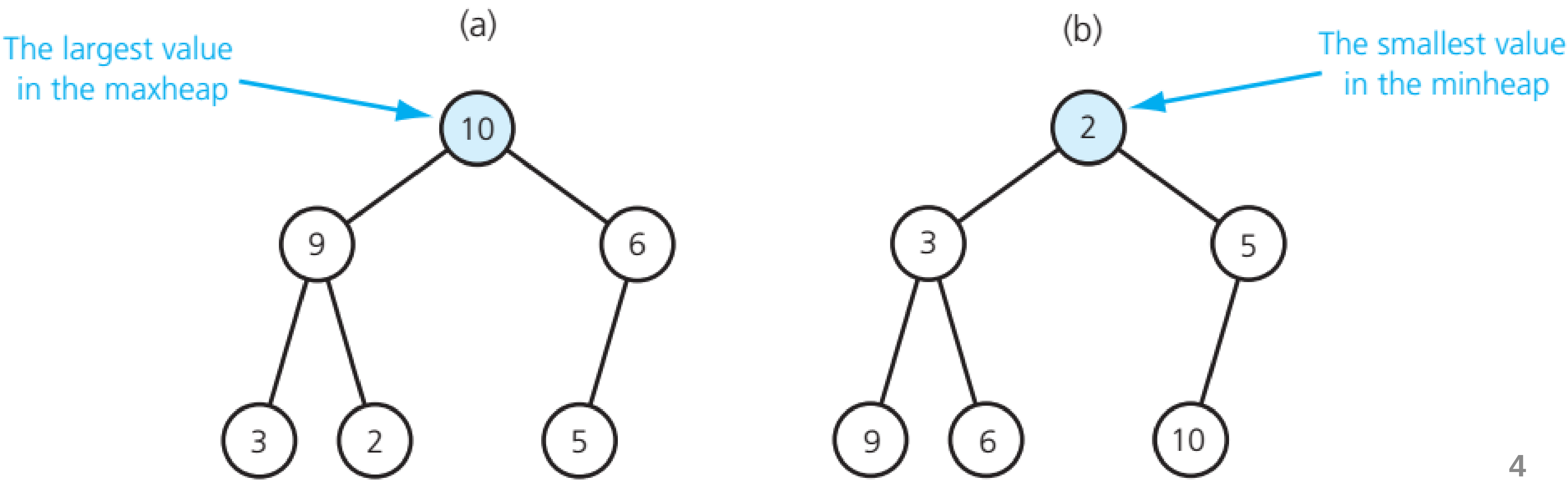
- The ADT Heap
- An array-based implementation of a heap
- A heap implementation of the ADT priority queue

The ADT Heap



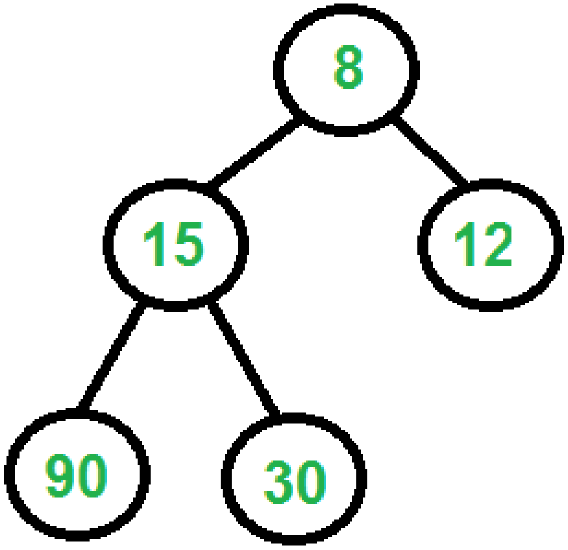
A definition of Heap

- A **maxheap** is a complete binary tree that either is empty or whose root
  - Contains a value greater than or equal to the value in each of its children and
  - Has heaps as its subtrees

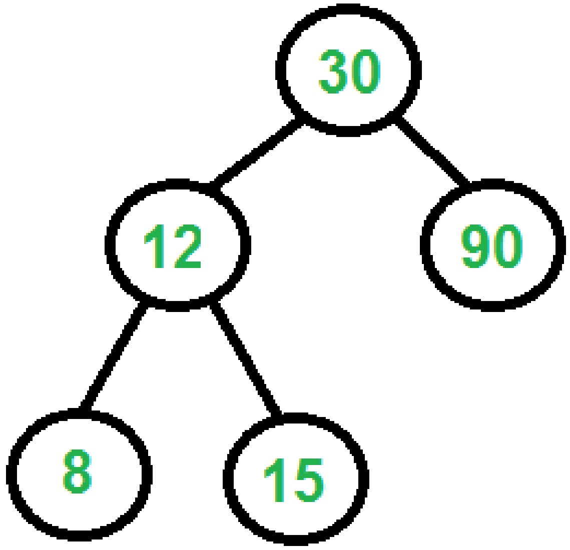


Heap vs. Binary search tree

- A heap is ordered in a much weaker sense than a BST.
- Heaps are always complete binary trees, while BST come in many different shapes.



Min Heap



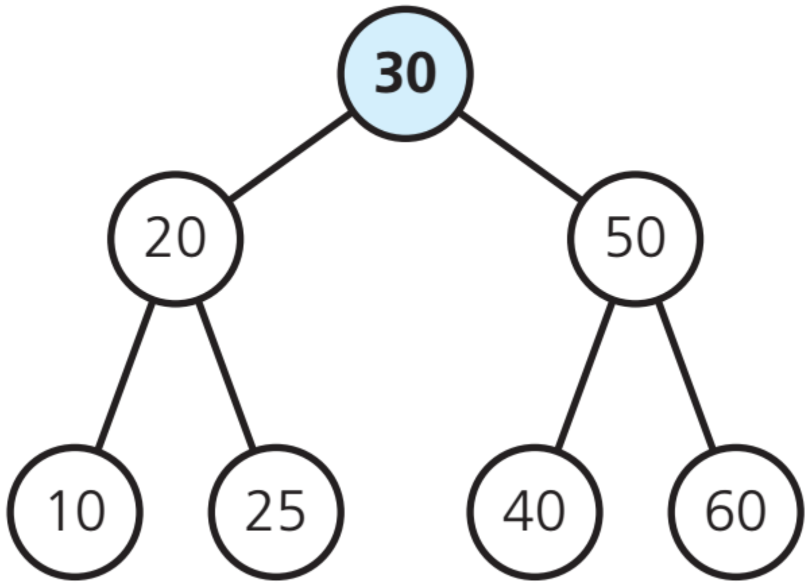
Binary Search Tree

ADT Heap operations

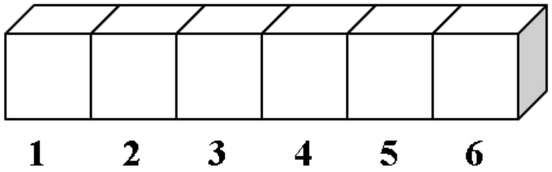
- `isEmpty(): Boolean`
  - Test whether a heap is empty
- `getNumberOfNodes(): integer`
  - Get the number of nodes in a heap
- `getHeight(): integer`
  - Get the height of a heap
- `peekTop(): ItemType`
  - Get the item in the heap’s root
- `add(newData: ItemType): Boolean`
  - Insert a new item into the heap
- `remove(): Boolean`
  - Remove the item in the heap’s root
- `clear(): void`
  - Remove all nodes from the heap

Quiz: Heaps

- Is this full binary tree a heap? Why?



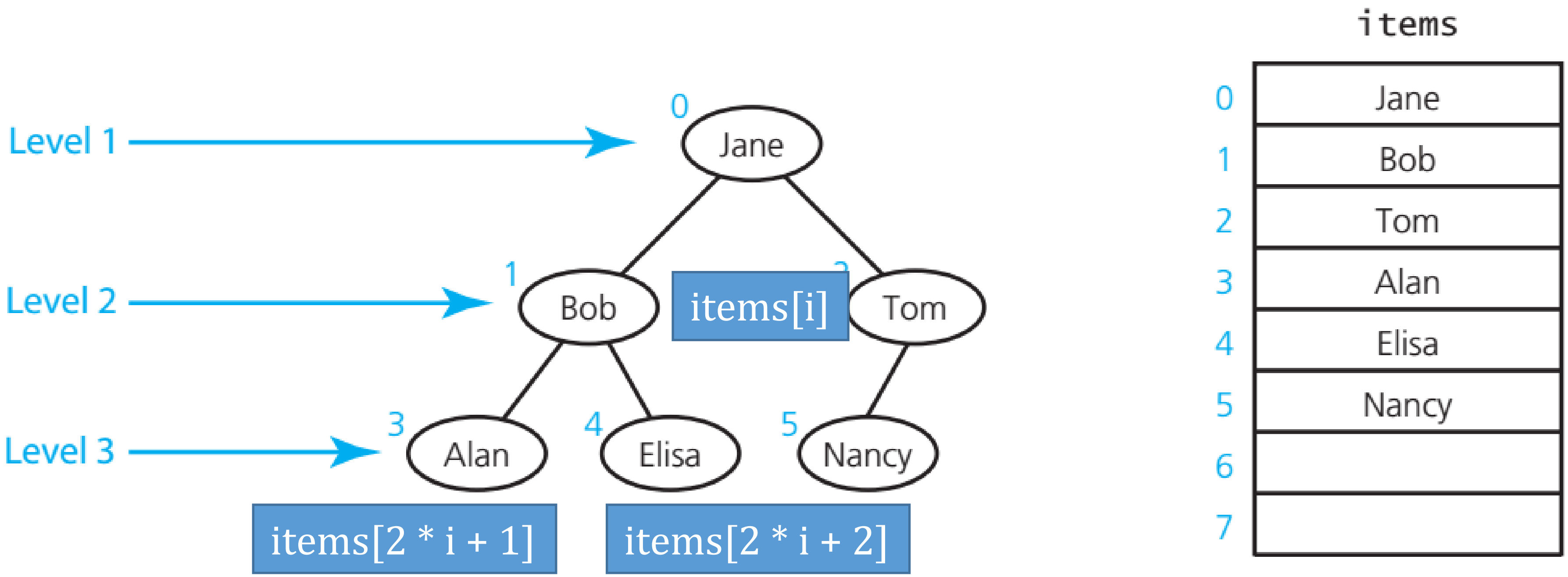
Array-based Heap



- *Algorithms for the Array-Based Heap Operations*
- *The Implementation*

Array-based implementation

- The array-based implementation is possible if the maximum size of the heap is available.



This array-based representation requires a complete binary tree.

Quiz: Array-based implementation

- What complete binary tree does the below array represent?
- Does the array represent a heap?

5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9

Array-based Heap operations

- Assume that we are considering a maxheap of integer.
- The class of heap has the following private data members
  - **items:** an array of heap items
  - **itemCount:** the number of items in the heap (integer)
  - **maxItems:** the maximum capacity of the heap (integer)

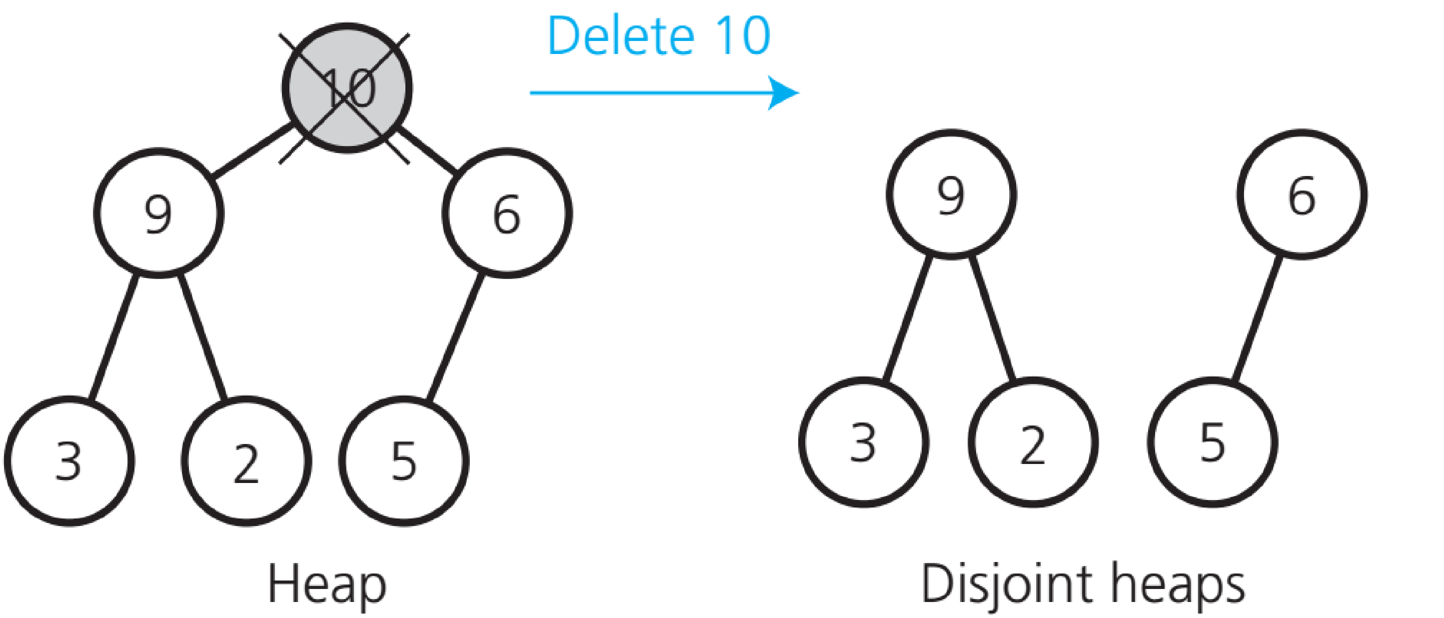
Retrieving an item from a heap

- The largest item must be in the root of the tree, i.e. at the top of the heap

```
// Get the extremal item in the heap's root
ItemType peakTop( )
   return items[0]            // Return the item in the root
```

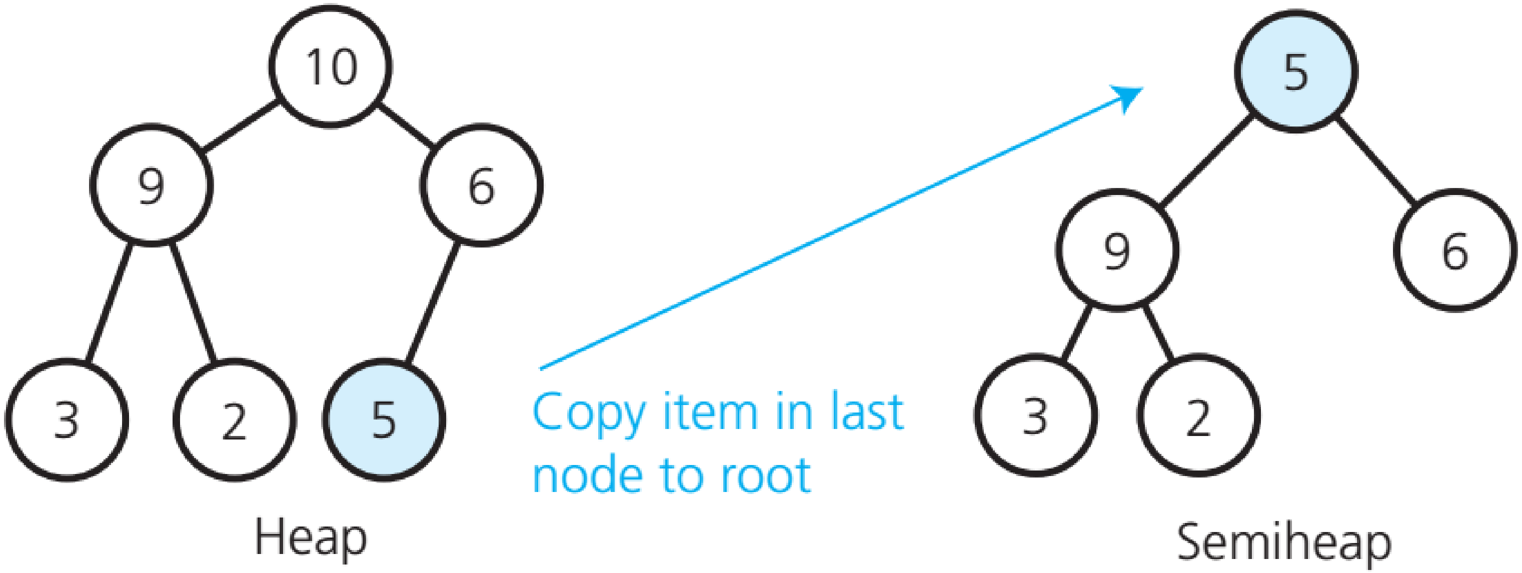
Removing an item from a heap

- Removing the root of the heap leaves two disjoint heaps.



Removing an item from a heap

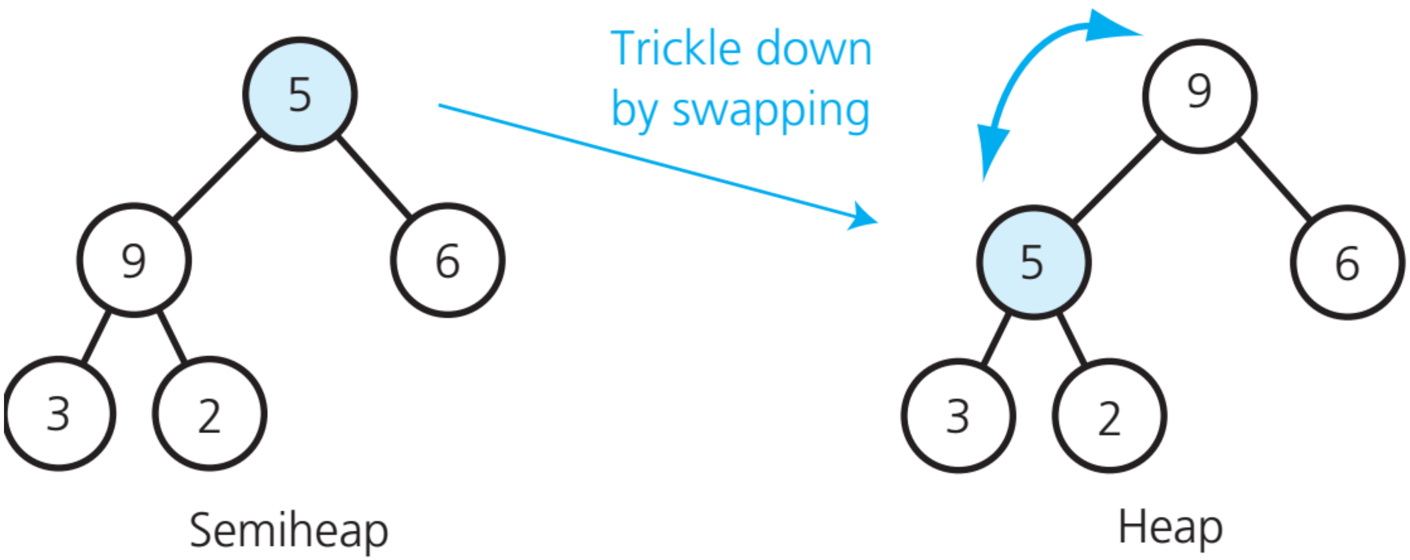
- Instead, remove the last node of the tree and place its item in the root




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

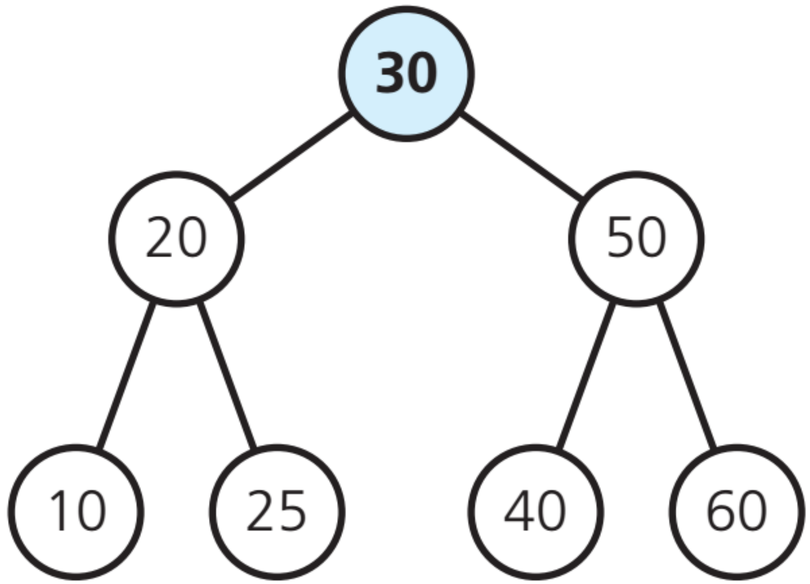
From semiheap to heap

- **Semiheap:** a complete binary tree whose left and right subtrees are both heaps but the root is out of place
- Transform a semiheap into a heap: Trickle down the tree until it reaches a node in which it will not be out of place



Quiz: Semiheap

- Is this full binary tree a semiheap? Why?

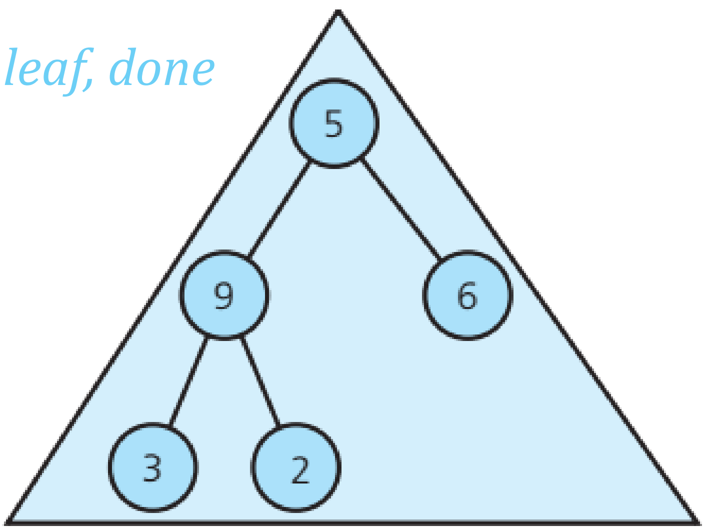


From semiheap to heap

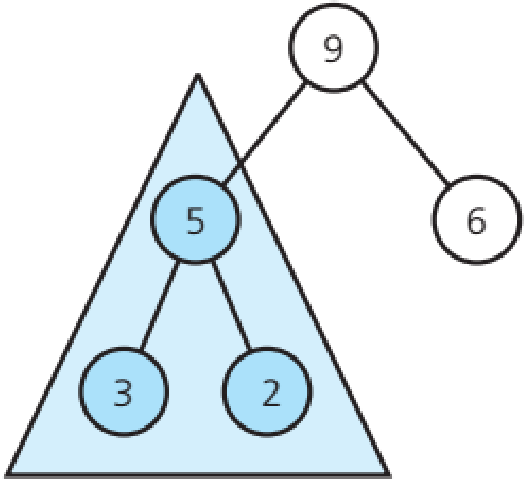
```
// Converts a semiheap rooted at index root into a heap.
heapRebuild(root: integer, items: ArrayType, itemCount: integer)
// Recursively trickle the item at index root down to its proper position by
// swapping it with its larger child, if the child is larger than the item.
// If the item is at a 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 = largerChildIndex + 1 // Right child index
        if (items[rightChildIndex] > items[largerChildIndex ])
            largerChildIndex = rightChildIndex // Larger child index
    }
    .....
```

From semiheap to heap

```
.....
// If the item in the root is smaller than the item in the larger child, swap
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, done
```

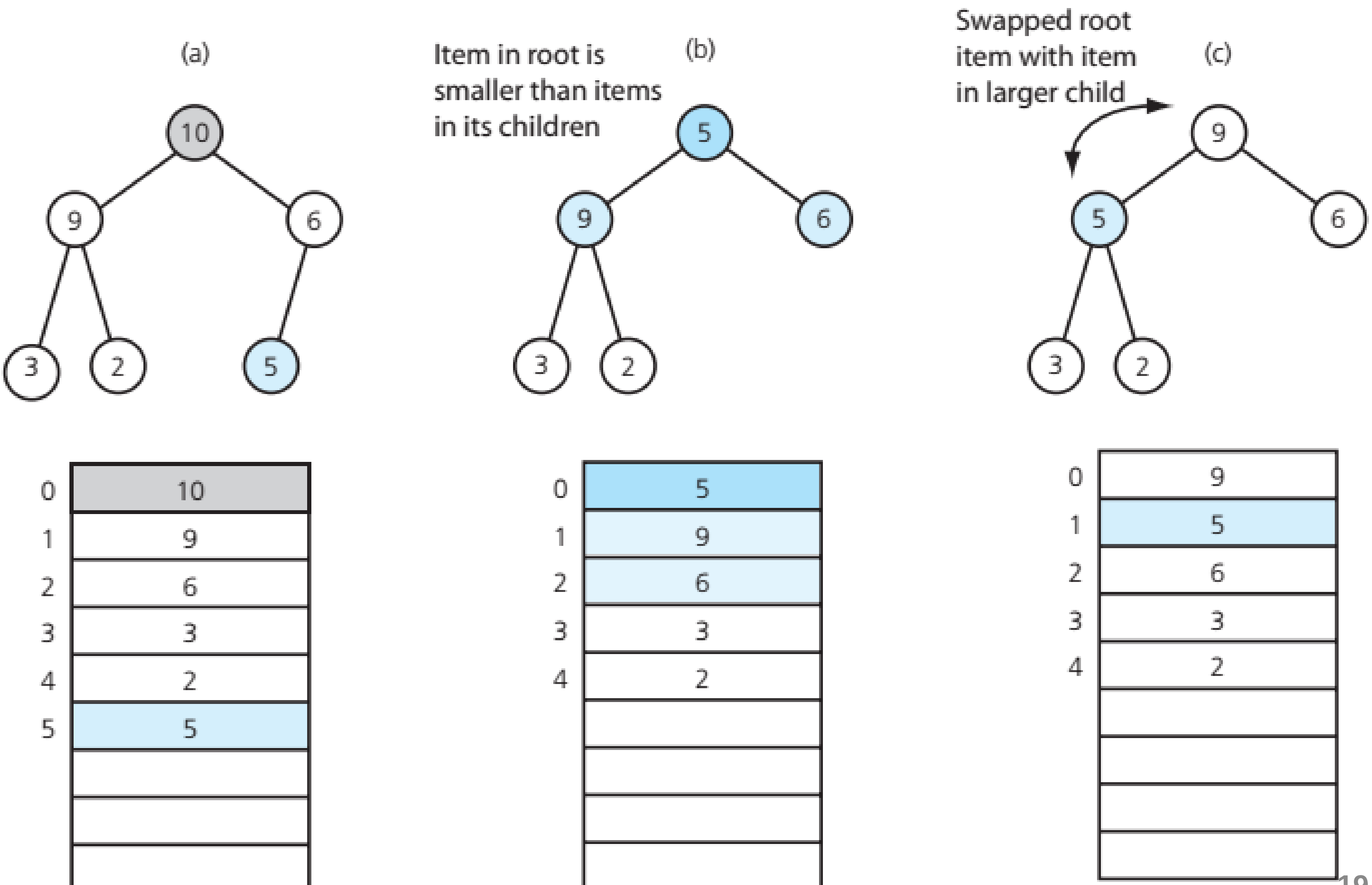


First semiheap passed to `heapRebuild`



Second semiheap passed to `heapRebuild`

From semiheap to heap: An example



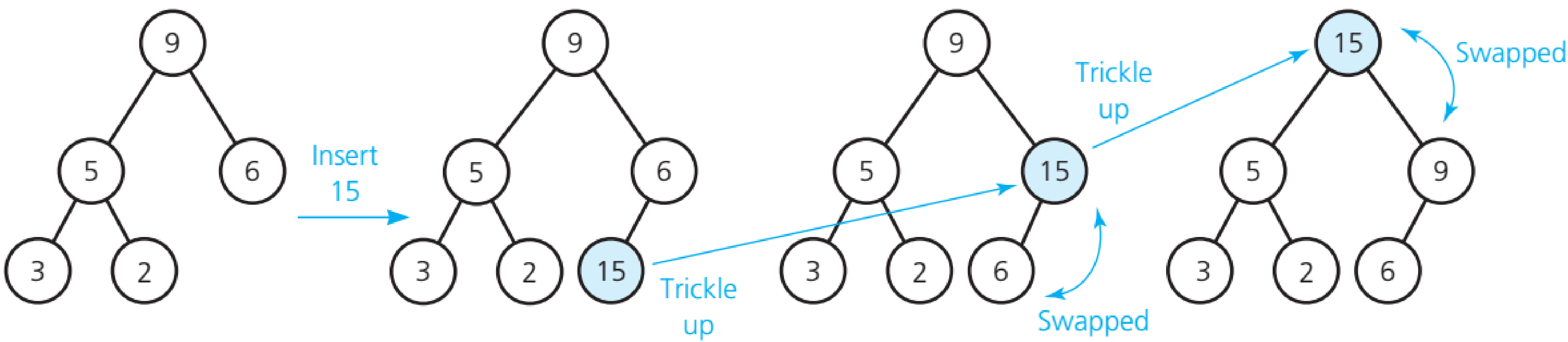
Removing an item from a heap

- The number of array items that **heapRebuild** must swap is no greater than the height of the tree.
- The height of a complete binary tree is always $\lceil \log_2(n+1) \rceil$
- Each swap requires three data moves
- Thus, **remove** requires $3 \times \lceil \log_2(n+1) \rceil + 1$ data moves

  $\rightarrow O(\log_2 n)$, quite efficient

Adding an item to a heap

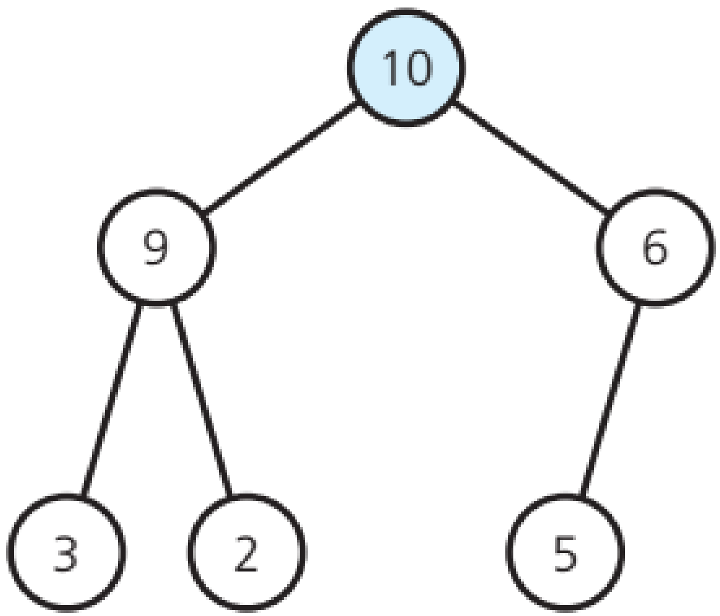
- The strategy for **add** is the opposite of that for **remove**.
- A new item is inserted at the bottom of the tree, and it trickles up to its proper place.



- The efficiency of **add** is like that of **remove**, also $O(\log_2 n)$.

Quiz: Add and Remove

- Consider the maxheap below.
- Draw the heap after you insert 12 and then remove 12.



Creating a Heap

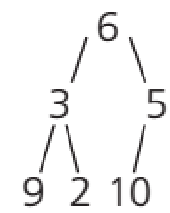
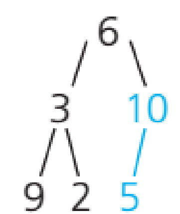
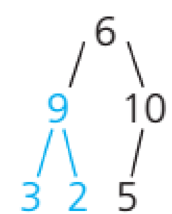
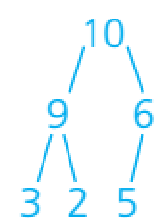
- Building a heap with the items in an array

```
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
}
```

or

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

Creating a Heap

	Array						Tree representation of the array
Original array	6	3	5	9	2	10	
	0	1	2	3	4	5	
After **heapRebuild(2)**	6	3	10	9	2	5	
	0	1	2	3	4	5	
After **heapRebuild(1)**	6	9	10	3	2	5	
	0	1	2	3	4	5	
After **heapRebuild(0)**	10	9	6	3	2	5	
	0	1	2	3	4	5	

Quiz: Creating a Heap

- Create a heap with the following array.

5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9

Heap Implementation of the ADT Priority Queue

Heap implementation

- Priority queue operations are exactly analogous to heap operations.
- Defining a priority queue with a heap results in a more time-efficient implementation.
  - Use an instance of ArrayMaxHeap as a data member of the class of priority queues, or use inheritance

Heap vs. Binary search tree

- If the maximum number of items in the priority queue is known, the heap is superior.
- A heap is complete and balanced $\rightarrow$ major advantage
- A search tree can be made balanced with operations that are far more complex than the heap operations.

Quiz: Heap-based implementation

- Consider a heap-based implementation of the ADT PQ.
- What does the underlying heap contain after the following sequence of pseudocode operations, assuming that pQueue is an initially empty priority queue?

```
pQueue.add(5)
pQueue.add(9)
pQueue.add(6)
pQueue.add(7)
pQueue.add(3)
pQueue.add(4)
pQueue.remove()
pQueue.add(9)
pQueue.add(2)
pQueue.remove()
```

THE END