SE 2205a: Data Structures and Algorithm Design

Unit 6 – Part 2: Binary Search Tree & Heap Tree

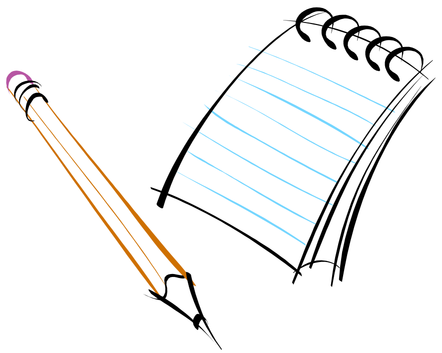
"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman, Ph.D, P.Eng, SMIEEE
Office Location: TEB 263
Email: QRAHMAN3@uwo.ca
Phone: 519-661-2111 x81399

"Genius is 1% talent and 99% percent hard work..." ~ Albert Einstein

Outline

- Application of Binary Tree
- Binary Search Trees
- Balanced Binary Trees
  - Heap Tree
    - Max-Heap
    - Min-Heap
    - Heap Sort
  - AVL, 2-3, 2-4, Red and Black Trees (Discussed in Part 3 of this Unit)

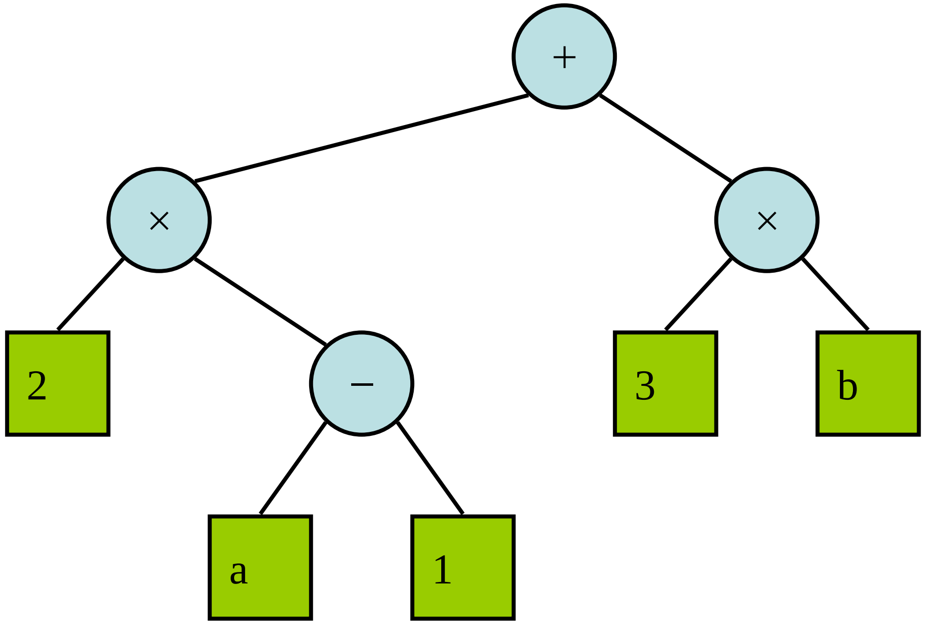


Applications of Binary Tree

- Applications:
  - Arithmetic expressions
  - Decision processes
  - Searching

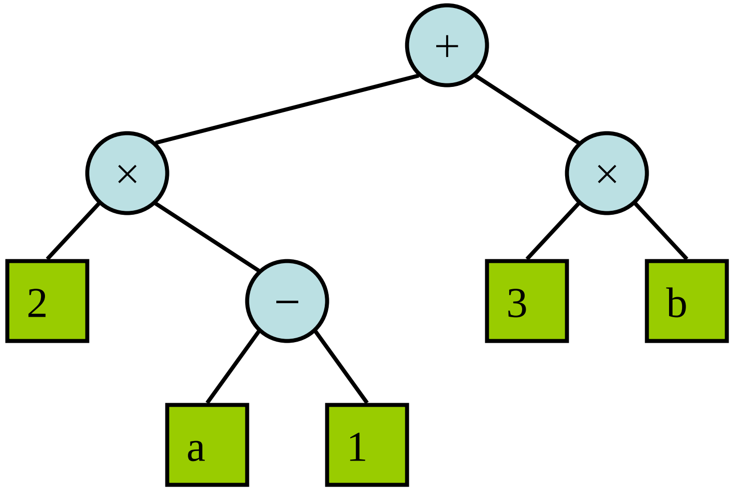
Application of a Binary Tree: Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes (nodes with at least one child): operators
  - Leaf nodes (A.K.A. external nodes): operands
  - To realize the expression, use the in-order traversal sequence
- Example: arithmetic expression tree for the expression: $(2 \times (a - 1) + (3 \times b))$



Print Algorithm - Arithmetic Expression Tree

- Specialization of an in-order traversal
  - print operand or operator when visiting the node/root with in-order traversal sequence
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

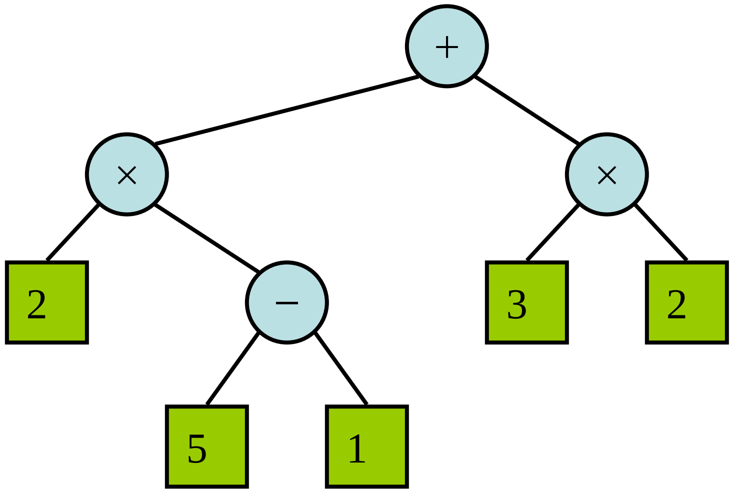


$((2 \times (a - 1)) + (3 \times b))$

Algorithm ***printExpression(tree(root))***
  if ***left subtree (root)*** $\neq$ **null**
    print("(")
    inOrder (left subtree (root))
  print(root.element ())
  if ***right subtree(root)*** $\neq$ **null**
    inOrder (rightsubtree(root))
    print (")")

Algorithm - Evaluate Arithmetic Expressions using Post-Order Traversal

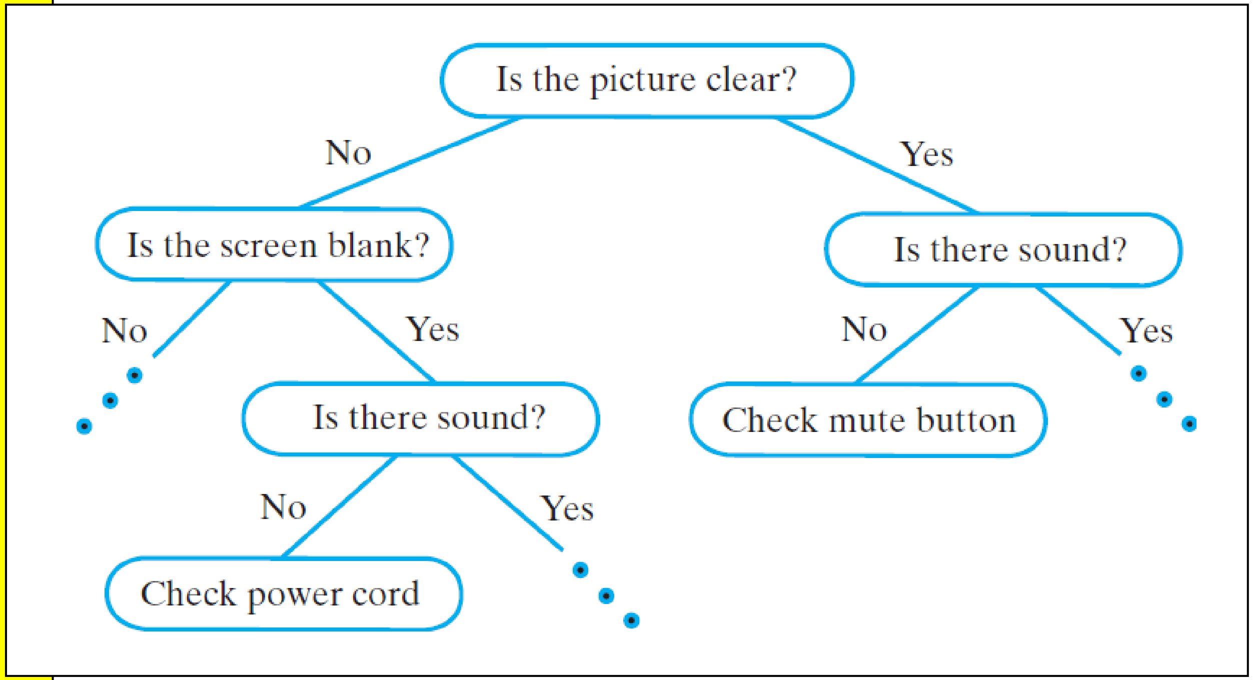
- Specialization of a post-order traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



Algorithm ***evalExpr***(***tree(root)***)
  if ***isExternal*** (***root***) {'external' is a leaf-node}
    return ***root.element*** ()
  else
    x ← ***evalExpr***(***left***(***root***))
    y ← ***evalExpr***(***right***(***root***))
    ◊ ← operator stored at ***root***
    return ***x*** ◊ ***y***

Application of Binary Tree: Decision Tree

- Used for expert systems
  - Helps users solve problems
  - Parent node asks question
  - Child nodes provide conclusion or further question

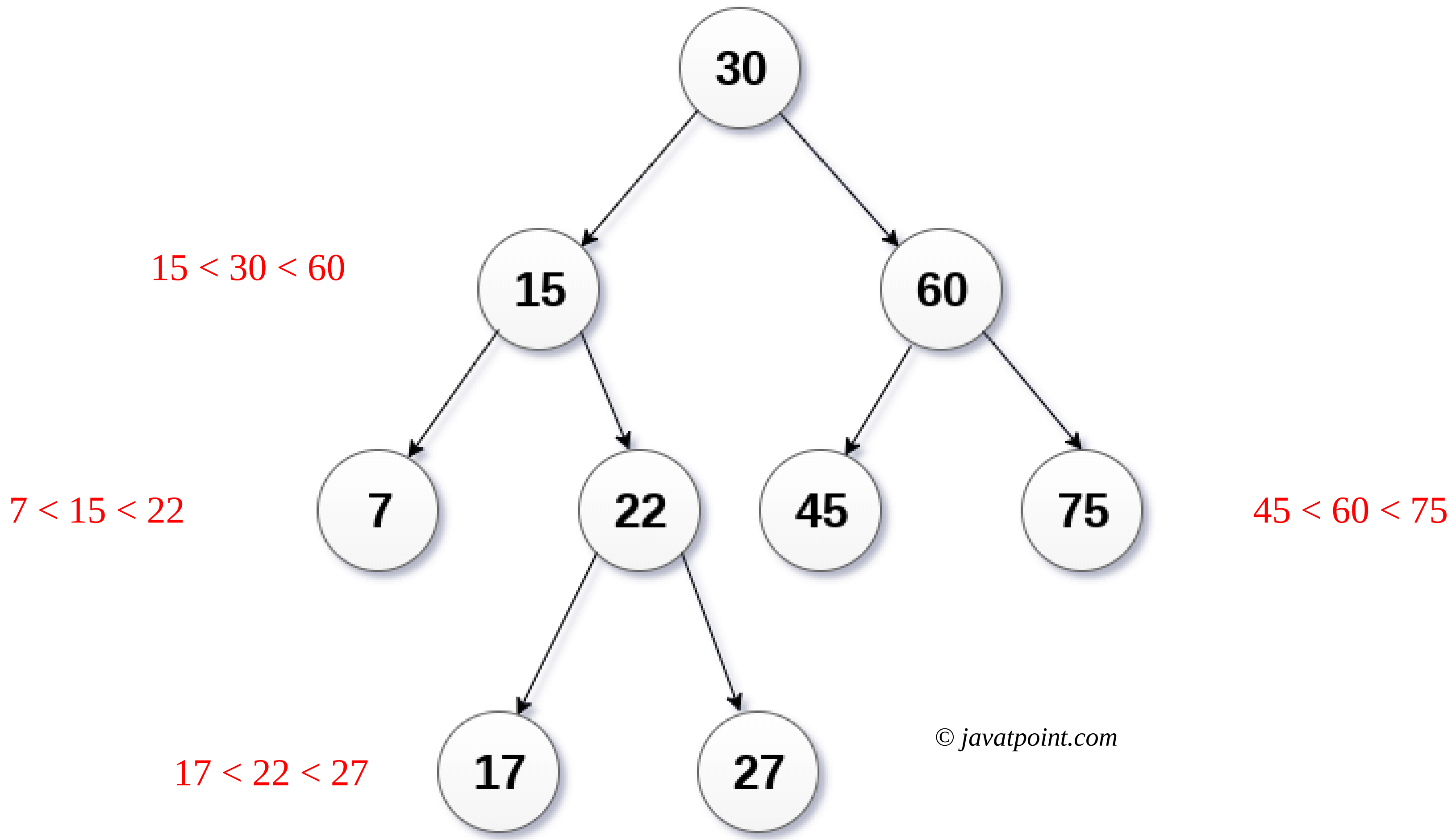


Binary Search Tree

- Binary Search Tree (BST) is a binary tree in which the nodes are arranged in a specific order to make the tree-search more efficient:
  - The value of all the nodes in the left sub-tree is less than the value of the root/parent.
  - The value of all the nodes in the right sub-tree is greater than or equal to the value of the root/parent.
- These rules are recursively applied to all the left and right sub-trees of the parent/root.
- The nodes in a binary tree contains Comparable Objects.

Binary Search Tree (BST)

Rule: Left < Parent <= Right, i.e., (Parent > Left && Parent<= Right)



In-Class Discussion

- Create a binary search tree using the following data elements (the first element forms the root of the tree)

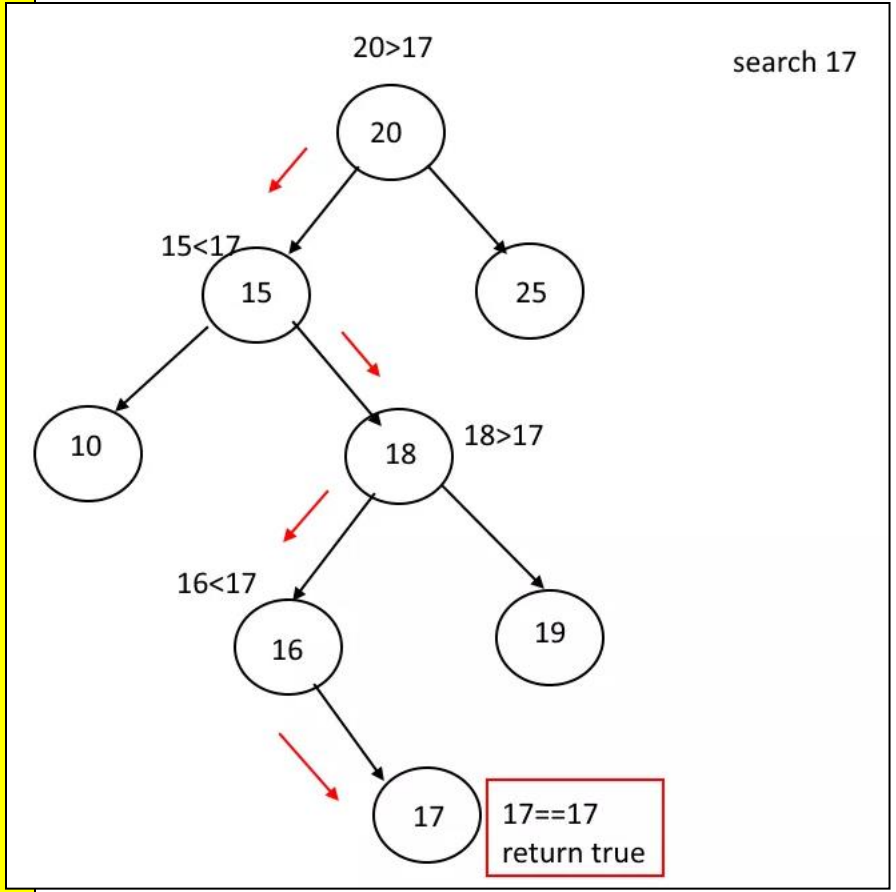
43, 10, 79, 90, 12, 54, 11, 9, 50

In-Class Discussion

How many structurally unique BSTs can be made using 3 nodes with corresponding weights of 1, 2, and 3?

Searching BSTs

- Searching a BST is quite efficient. Here is the algorithm:
  - Start from the root node.
  - Compare the search value with the root, if it is less than the root, then move to the left, else move to the right.
  - If the search value is found, return true, else return false.
- Note: <u>In-order traversal of a BST results in a sorted list of the nodes.</u>

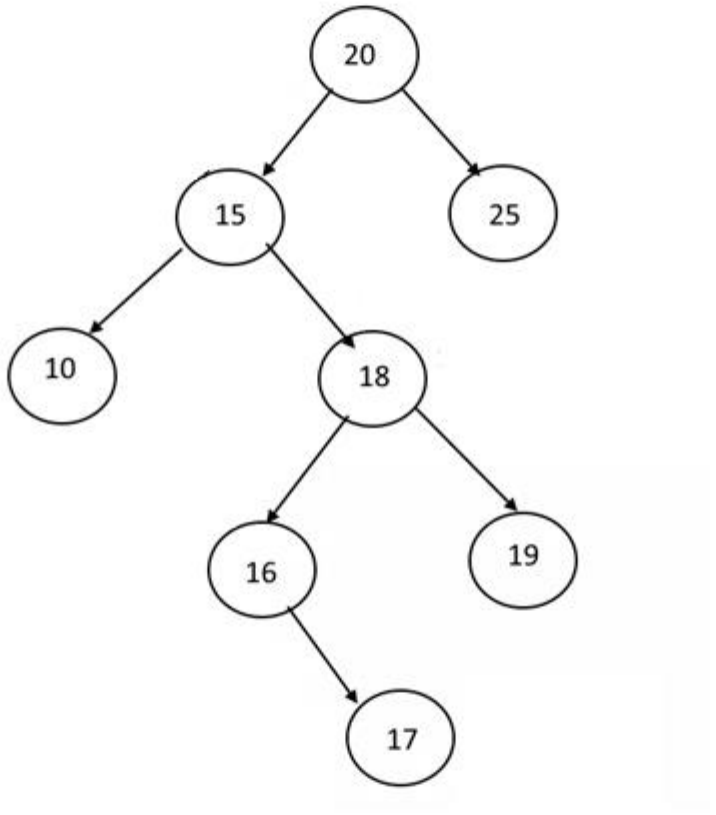


© tutorialhorizon.com

Check the search path with arrows (→).

In-Class Discussion

What is the In-order traversal sequence of the following tree?



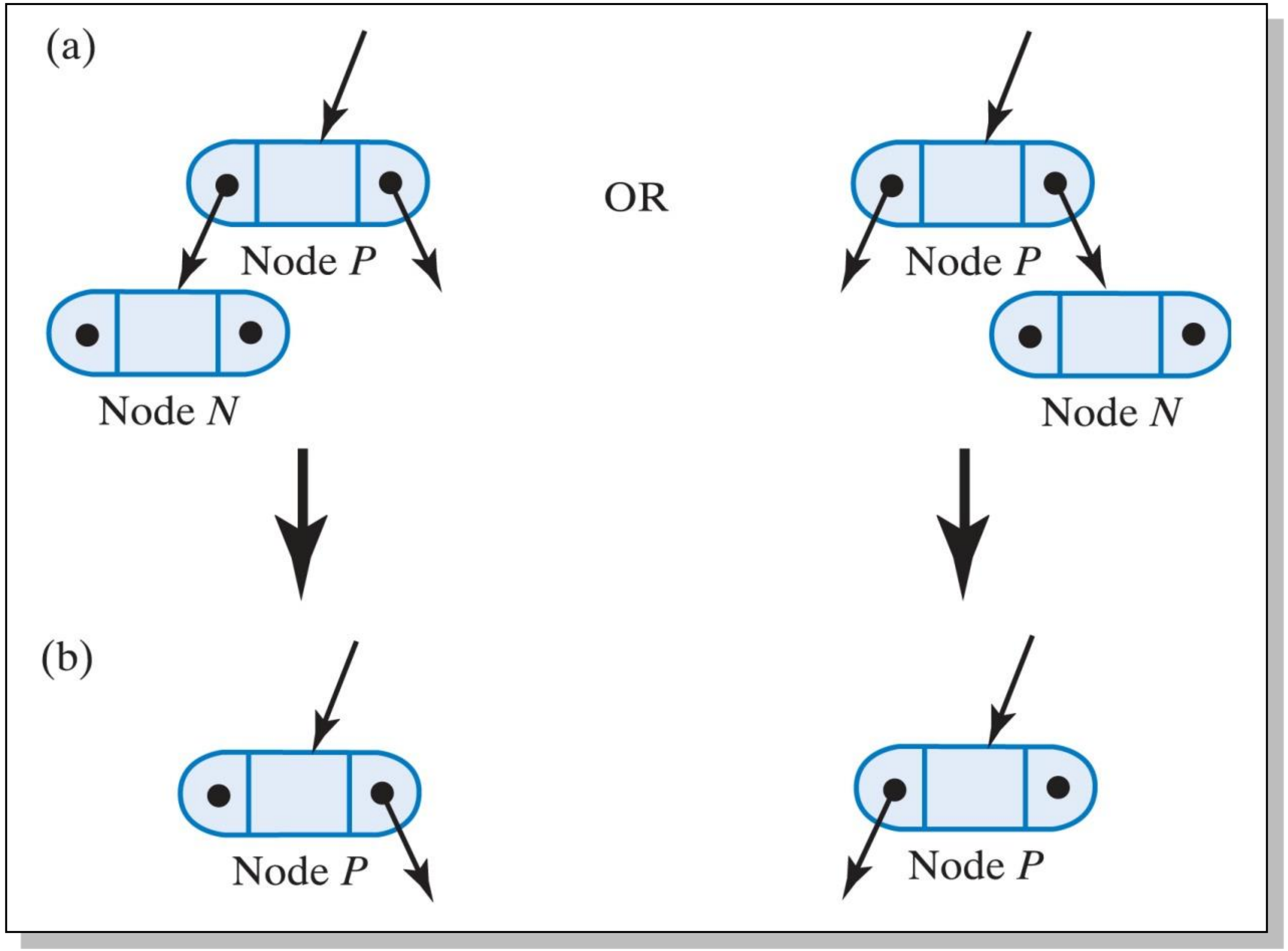
Operations on Binary Search Tree

- The basic operations on BST include:
  - Search (Binary tree can have duplicate elements. But, in case of BST: Although the definition permits duplicate keys, some BSTs don't permit duplicate keys.)
  - Retrieve
  - Add
  - Traverse
  - Remove
- We will discuss the remove() operation – rest are straight forward.

Removing an Entry

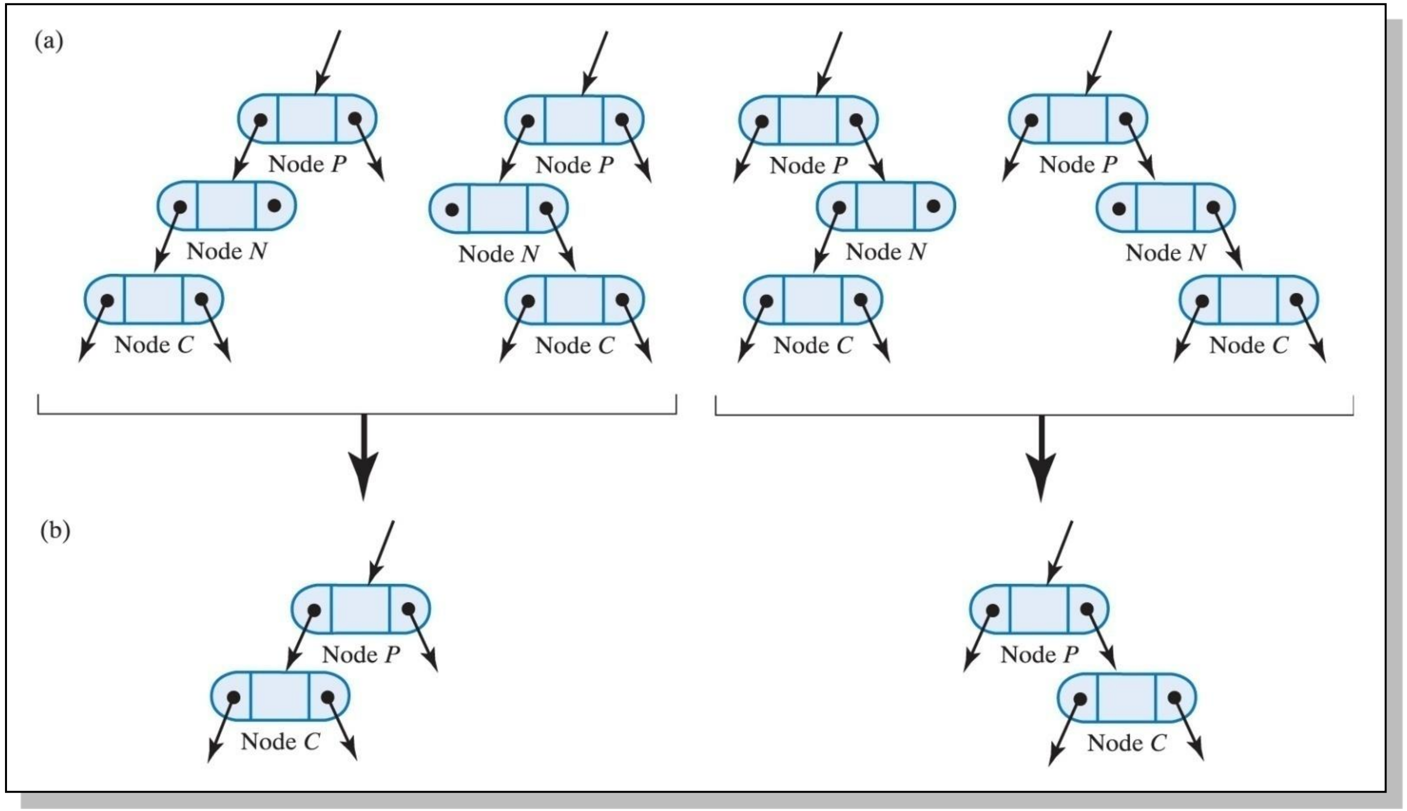
- The `remove()` method must receive an entry to be matched in the tree
  - If found, it is removed
  - Otherwise, the method returns null
- Three cases
  - The node has no children, it is a leaf (simplest case)
  - The node has one child (simple logic)
  - The node has two children

Case 1: Removing an Entry (Node N), Node is a Leaf



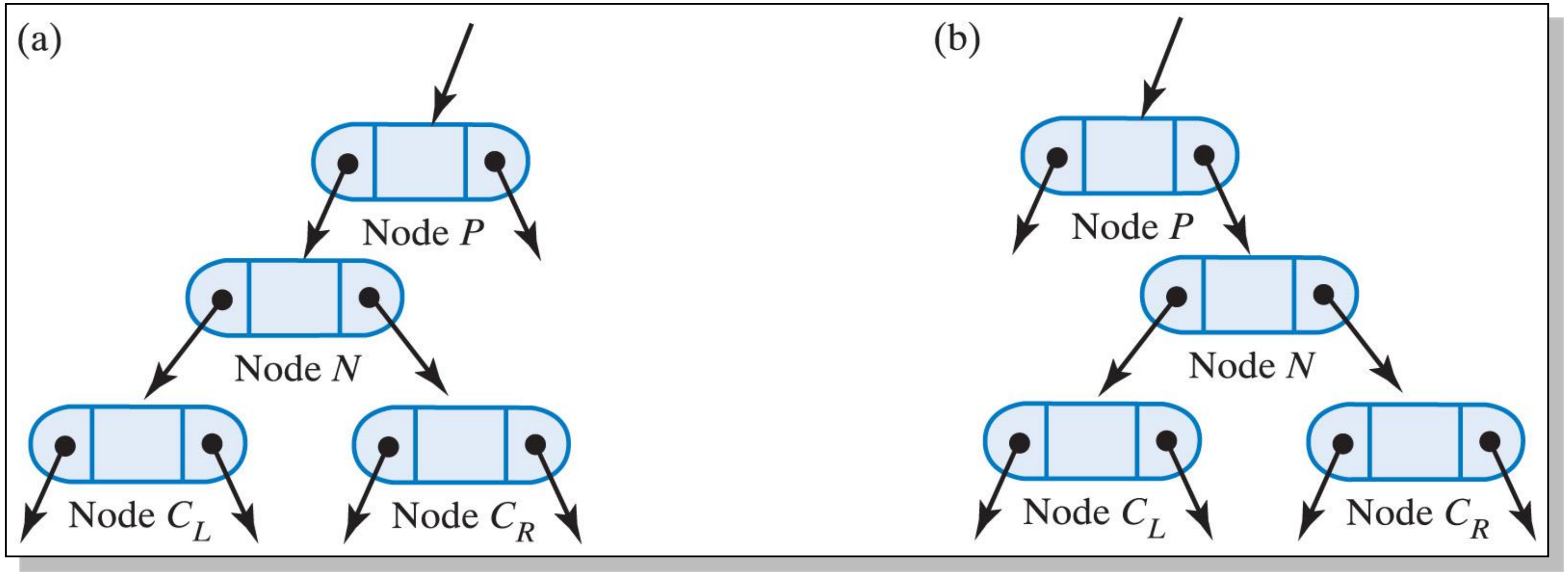
(a) Two possible configurations of leaf node *N*;
(b) the resulting two possible configurations after removing node *N*.

Case 2: Removing an Entry (Node N), Node has One Child



(a) Two possible configurations of node (to be removed) *N* that has one child;
(b) the resulting two possible configurations after removing node *N*.

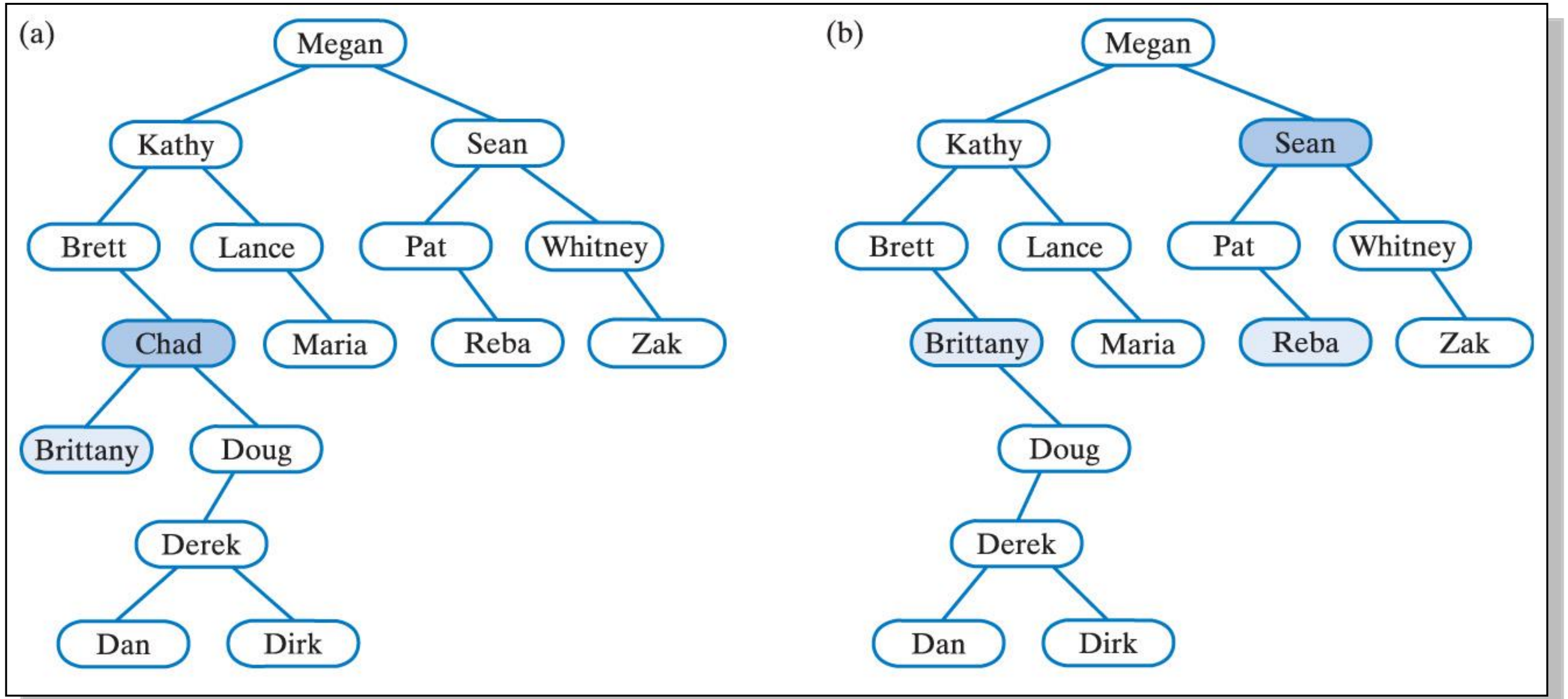
Case 3: Removing an Entry (Node N), Node has Two Children



Two possible configurations of node *N* that has two children.

High level description for removing an entry

- Algorithm to remove an entry (node N) that has two children:
- If the right or left subtree of N has a leaf-node, replace N with that leaf node, and then remove that leaf node. Else….
  - Find the maximum value, R, which is the rightmost node (R) in N's left sub-tree (R is the in-order successor of N), then replace N with R, and then delete R OR find the minimum value, L, which is the leftmost node (L) in N's right sub-tree (L is the in-order predecessor of N), then replace N with L, and then delete L.

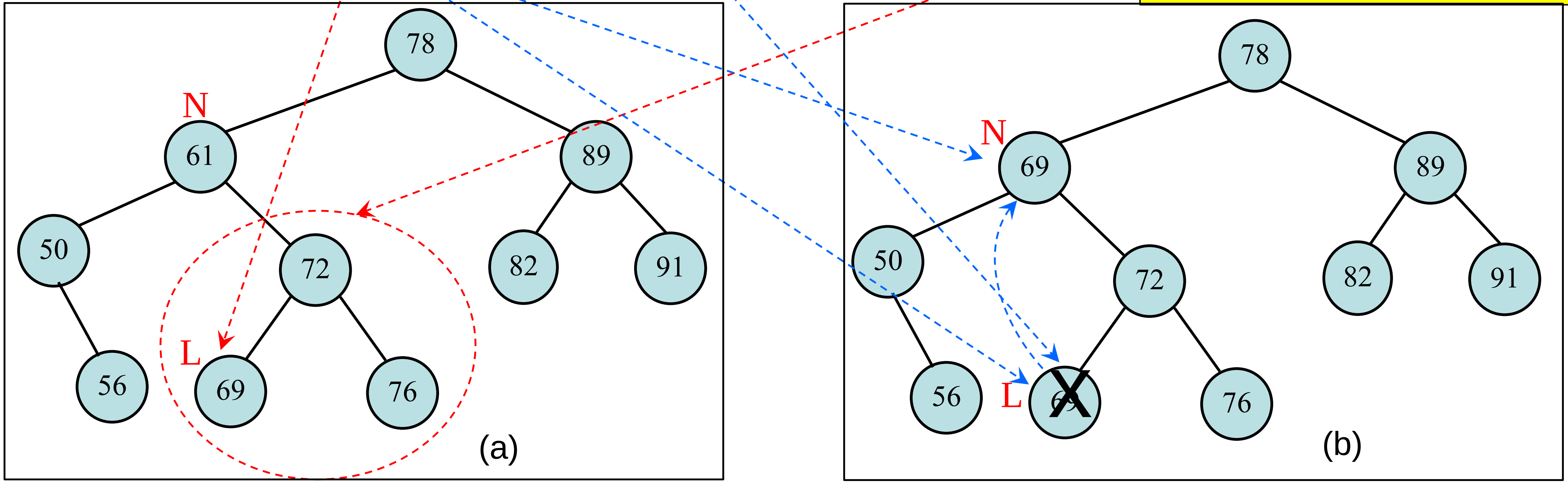


Removing an Entry (Chad), Node has Two Children: (a) Before removal (b) After removal.
Note: According to the algorithm - To remove Sean from (b), we can replace the value 'Sean' with 'Reba' and remove the node where 'Reba' was placed.

High level description for removing an entry: Example

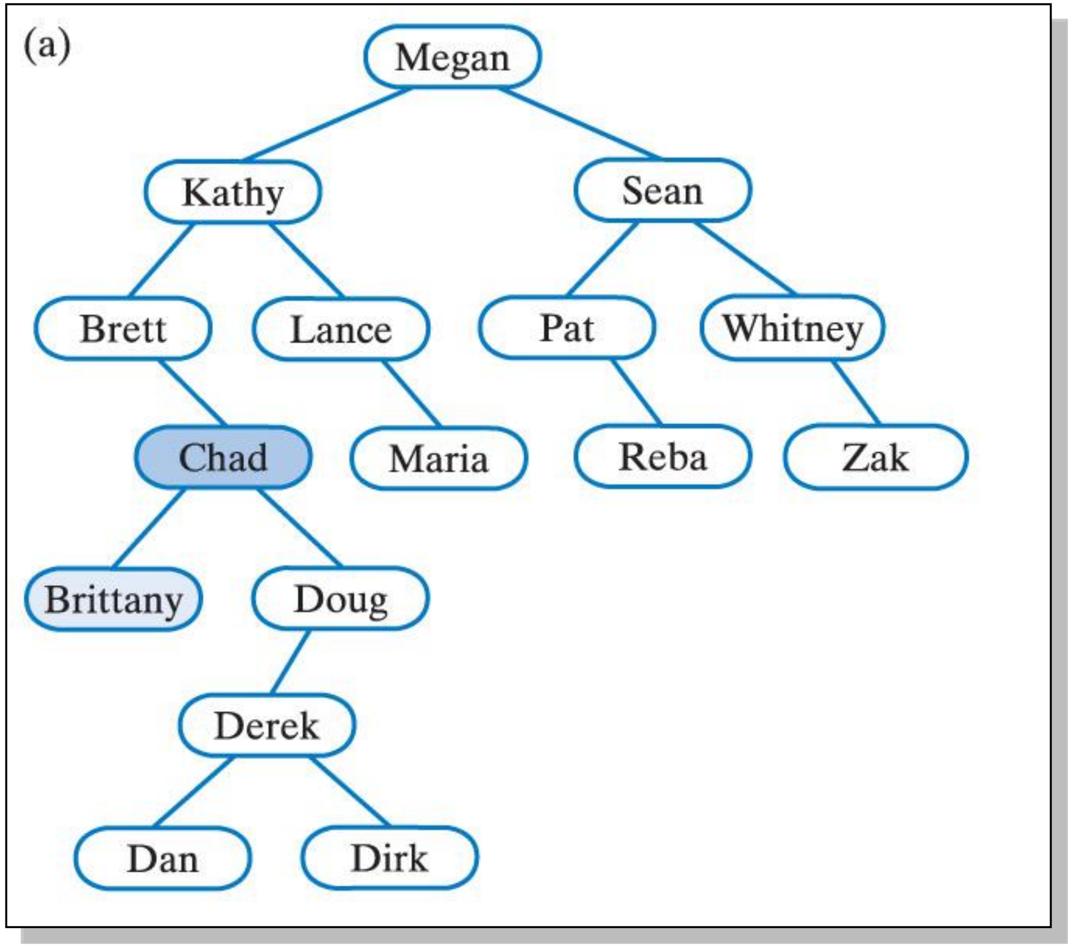
- Algorithm to delete an entry (node N which is 61 in the example) that has two children:
  - Find the leftmost node (the minimum value) (L = 69) in N's (61) right sub-tree
  - Replace node N with that entry
  - Delete node L

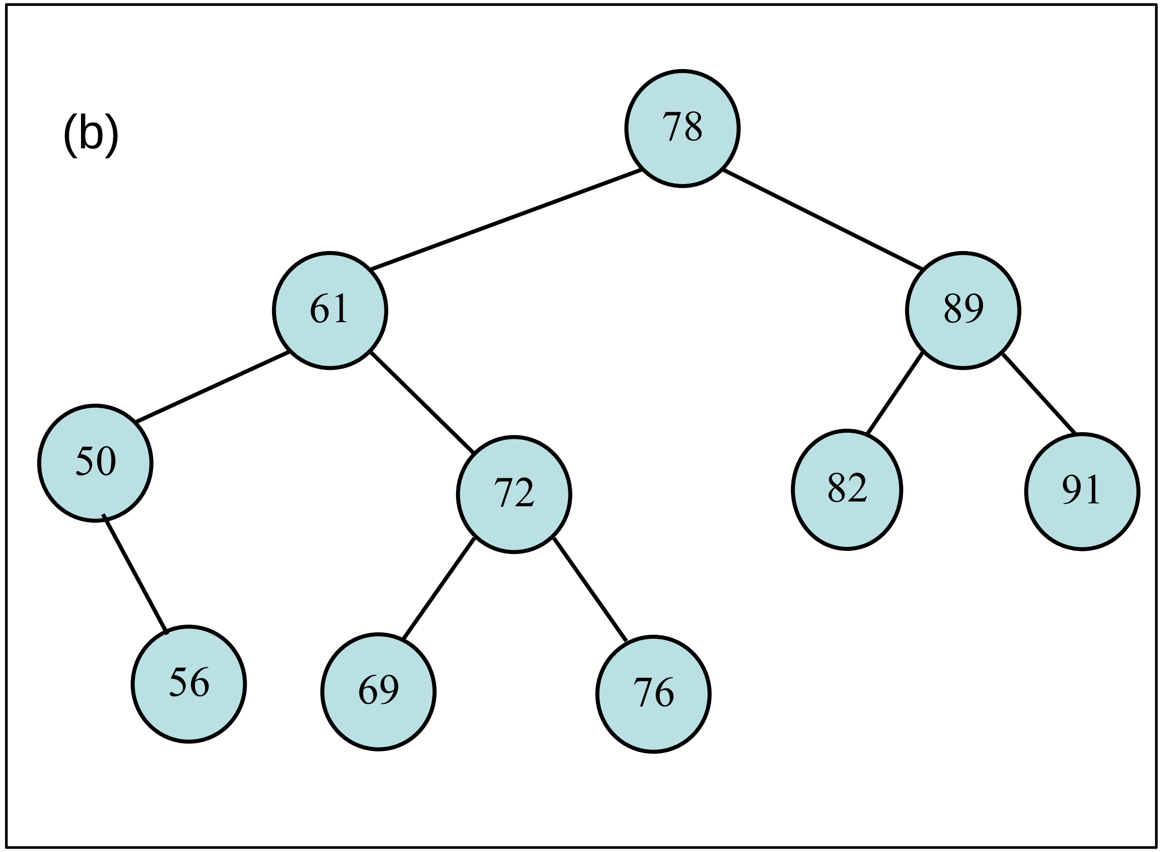
Removing an Entry (61), Node has Two Children: (a) Before removal (b) After removal



In Class: Removing a Node that has two Children

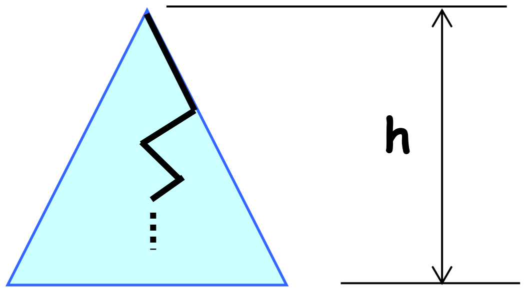
- In-Class Discussion: Let's remove some nodes from the following trees:





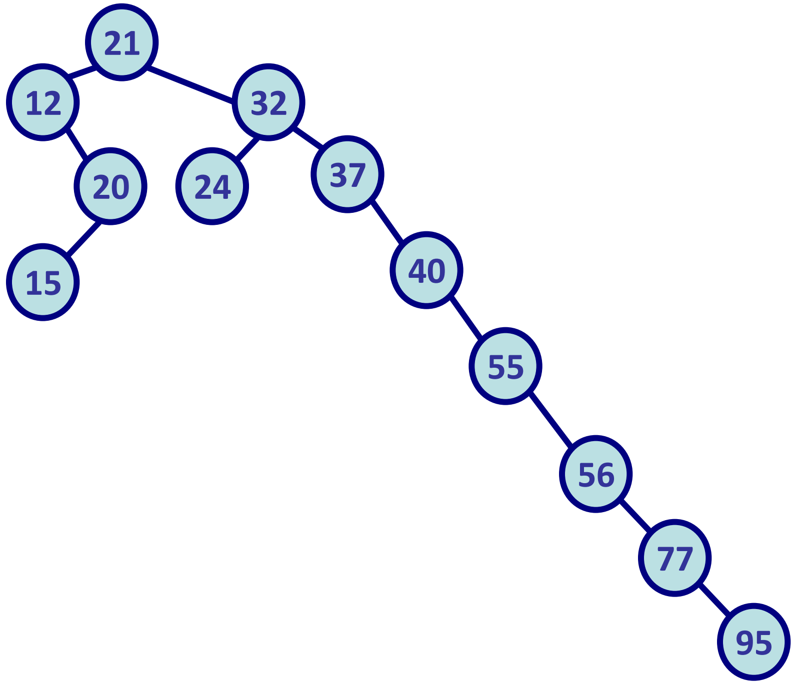
Efficiency of Operations in BST

- Operations add(), remove(), getEntry() require a search that begins at the root of a BST.
- Maximum number of comparisons is directly proportional to the height, *h* of the tree.
- Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height short.
- These operations are O(*h*).
- In-Class Discussion: For an n-node BST what will be the worst-case time complexity?
- What would be the highest and lowest possible heights of a BST in terms of n number of nodes?



Review: Efficiency of Operations in BST

- Since the shape of a BST is determined by the order that data is inserted, we run the risk of trees that are essentially a long chain or list.

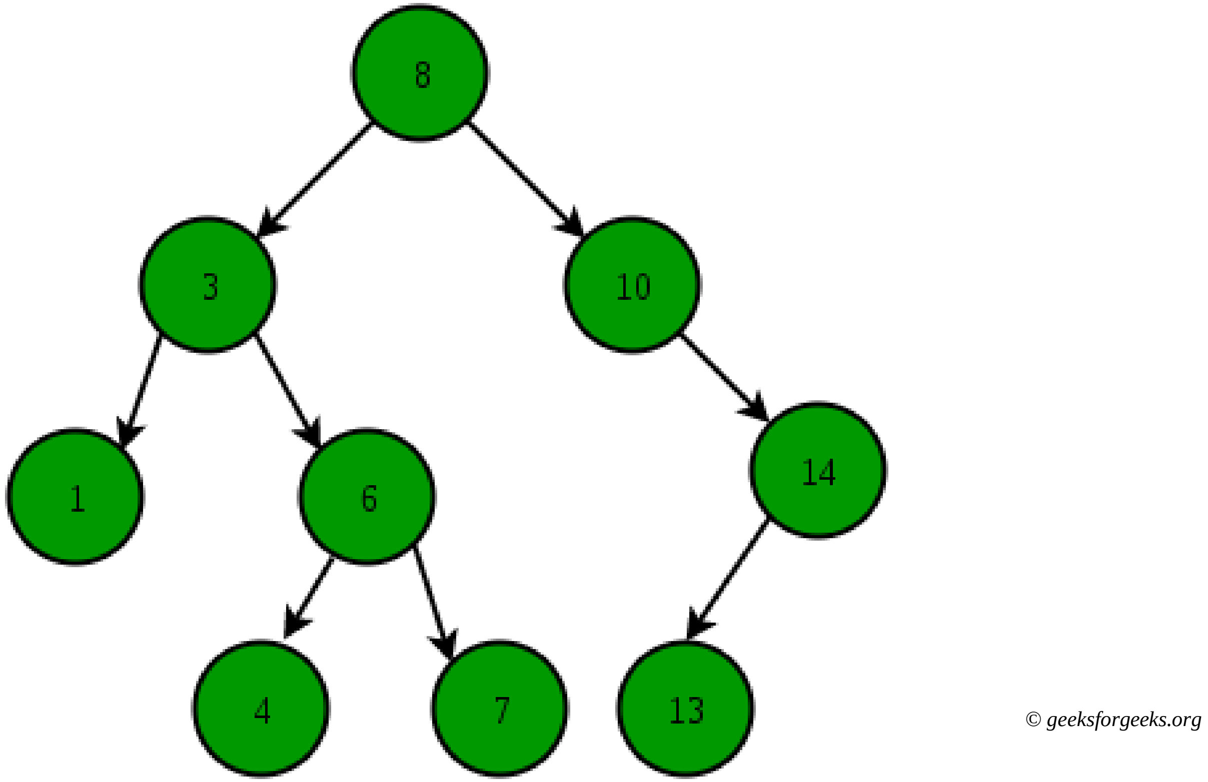


- So, the worst case for a single BST operation can be O(n), and for m operations can be O(m*n)

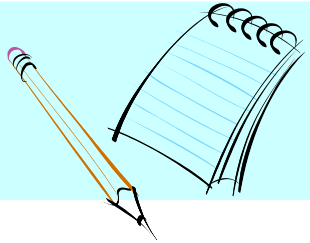
- In balanced (discussed next) BST single operation can be done in O(log n), and for m operations, O(m log n)

In-Class Discussion

Insert “15” in the tree below.



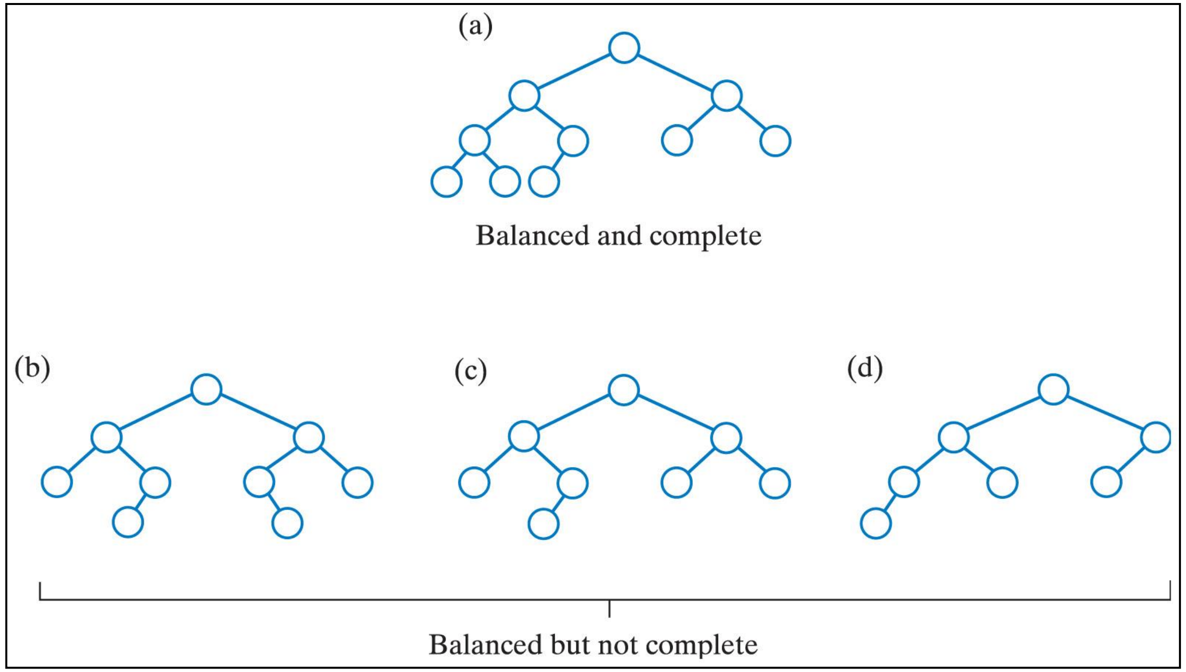
Take away on BST



- When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree. However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n).
- To avoid O(n) complexity and get the advantage of O(log n) complexity, we aim for full or complete binary tree.
- The question is how can we make that possible?

A Balanced Tree

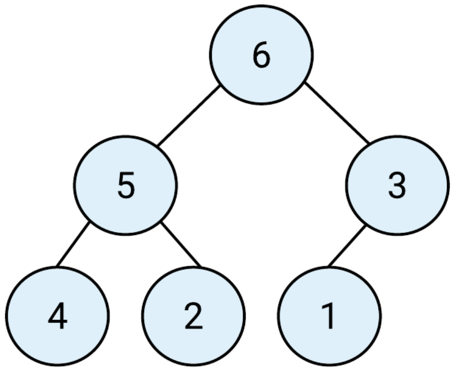
- Fully balanced Tree:
  - Subtrees of each node have exactly same height (e.g., Full / perfect BT)
- Height balanced Tree:
  - Subtrees of each node in the tree differ in height by no more than 1
- Fully balanced or height balanced trees are balanced Trees.
- A balanced tree automatically keeps its height small.
- A Binary Tree (BT) or a Generic Tree can be a Balanced tree.
- A Balanced BT's height is guaranteed to be logarithmic (In-Class discussion).
- A Full BT and Complete BT are always balanced.



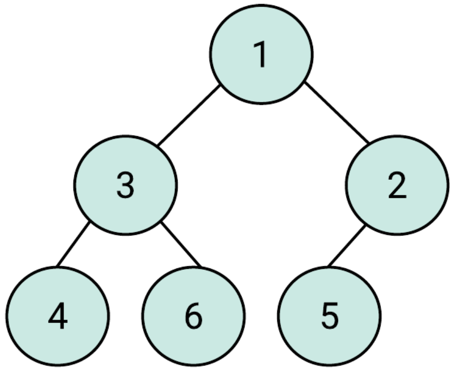
Some binary trees that are height balanced.

Heap Trees

- Heap Tree (AKA Heap) is a <u>complete binary tree</u> (so it is naturally balanced) whose nodes are ordered in two different configurations – Maxheap and Minheap.
- Max-Heap: The root node is greater than its children. This property is recursively true for all the sub-trees.
- Min-Heap: The root node is smaller than its children. This property is recursively true for all the sub-trees.
- The Nodes in a heap contain Comparable objects.
- The priority queues are often referred to as "heaps".
- Note: Heap trees are NOT BSTs.



Max Heap



Min heap

Example code: MaxHeap Interface

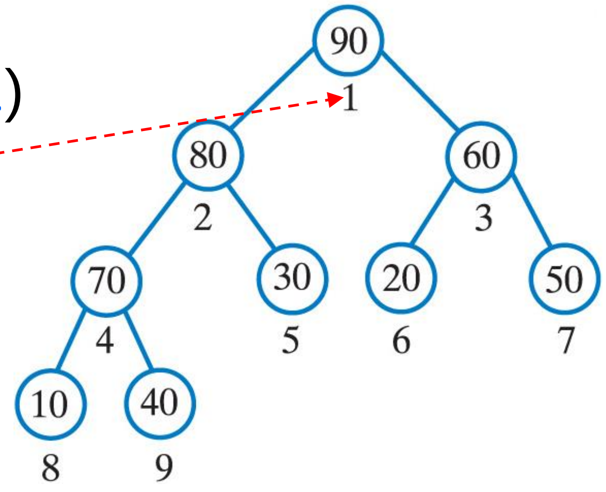
```
public interface MaxHeapInterface < T extends Comparable < ? super T >>
{
    /** Task: Adds a new entry to the heap.
    * @param newEntry an object to be added */
    public void add (T newEntry);

    /** Task: Removes and returns the largest item in the heap.
    * @return either the largest object in the heap or if the heap is empty before the operation, null */
    public T removeMax ();

    /** Task: Retrieves the largest item in the heap.
    * @return either the largest object in the heap or, if the heap is empty, null */
    public T getMax ();

    /** Task: Detects whether the heap is empty.
    * @return true if the heap is empty, else returns false */
    public boolean isEmpty ();

    /** Task: Gets the size of the heap.
    * @return the number of entries currently in the heap */
    public int getSize ();

    /** Task: Removes all entries from the heap. */
    public void clear ();
} // end MaxHeapInterface
```

Using an Array to Represent a Heap

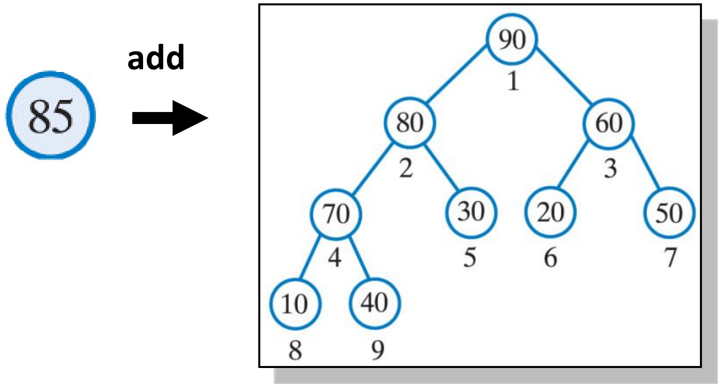
- Heap tree is a complete binary tree, and hence one can use level-order traversal to store data in consecutive locations of an array.
- It enables easy location of the data in a node's parent or children
- In this case, two approaches can be used to devise the algorithm.
- Approach 1 (Approach 2 is in blue): The cell at index 0 is not used (index 0 is used)
  – Parent of a node at i is found at $i/2$ when $i \neq 1$ (parent of $i = (i - 1)/2$, when $i \neq 0$)
  – the left child is at index $2i$ (the left child is at index $2i + 1$)
  – the right child is at index $2i + 1$ (the right child is at index $2i + 2$)

  (Approach 1 is used in the discussion here)

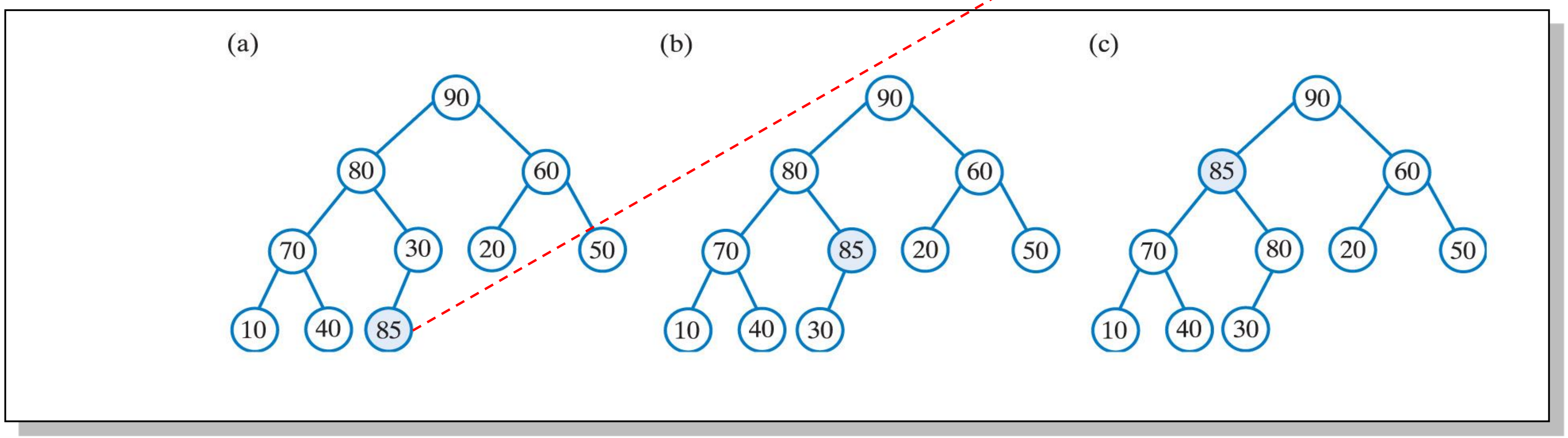
	90	80	60	70	30	20	50	10	40			
0	1	2	3	4	5	6	7	8	9	10	11	12



Data Insertion into the Max Heap



90	**80**	**60**	**70**	**30**	**20**	**50**	**10**	**40**	**85**



The steps in adding 85 to the maxheap

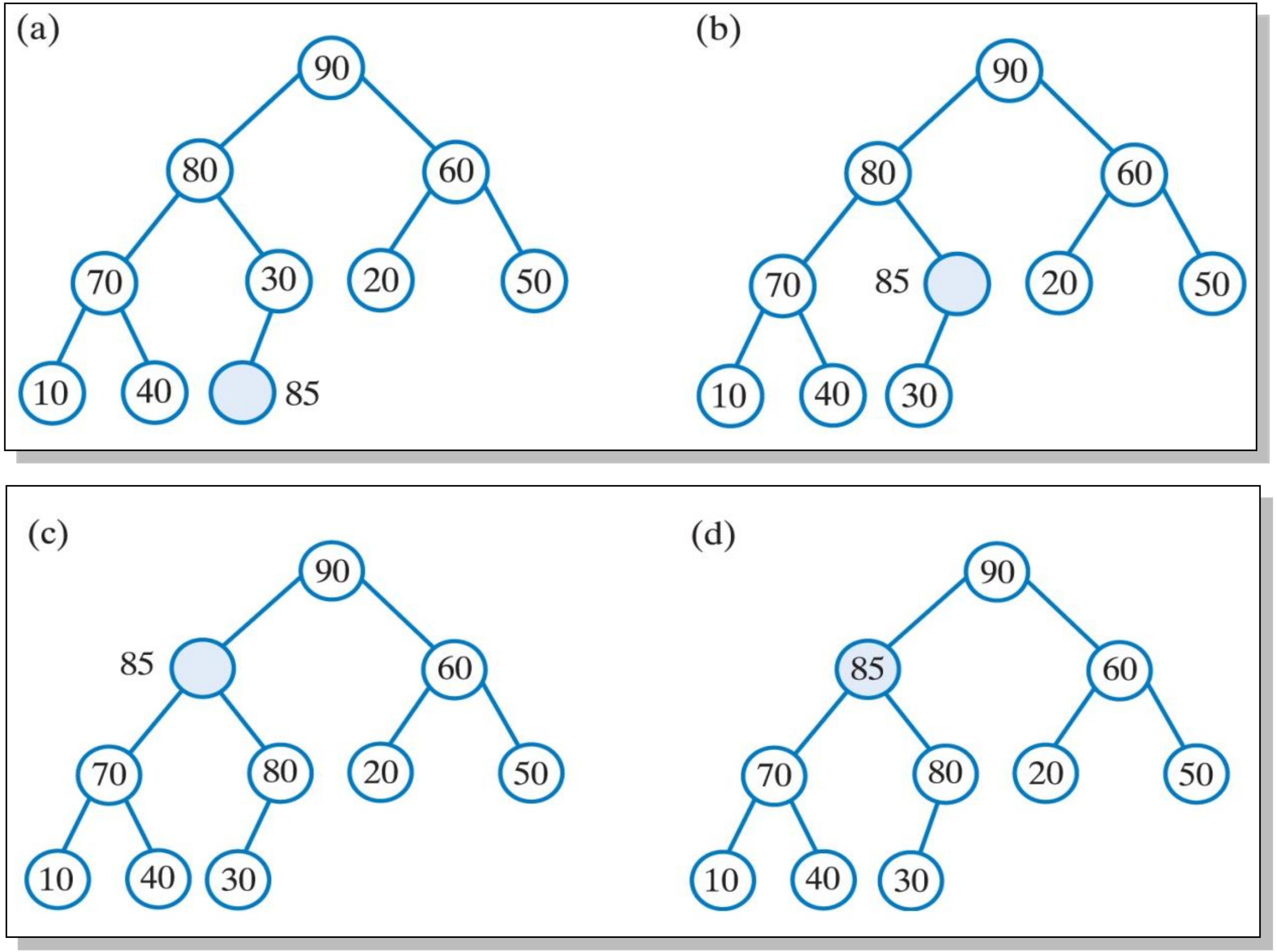
Upheap

- Insert the new key *k* at the first available leaf position on the far left (the first leftmost empty position).
- After the insertion of this new key *k*, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping *k* along an upward path from the insertion node.
- Upheap terminates when the key *k* reaches the root or a node whose parent has a key greater (smaller for minheap) than or equal to *k*

Save Time - Revised Upheap Insertion

- Hang on to the key to be inserted.
- Locate the first available leaf position on the far left (the first leftmost empty position).
- Follow path from this leaf toward root just by comparing the key with the parent-node value (without swapping) until correct position for key is found.
- As this is done
  - Move the values (the entries) from parent to child
  - Make room for the new entry
  - Insert the key (very similar idea as ‘Insertion sort’)

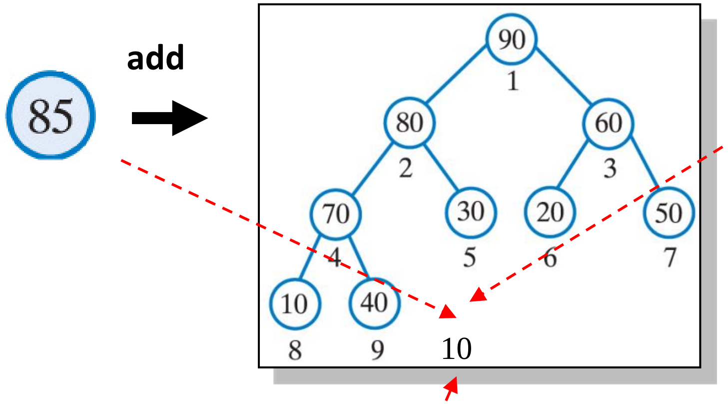
Upheap – Avoid Swapping



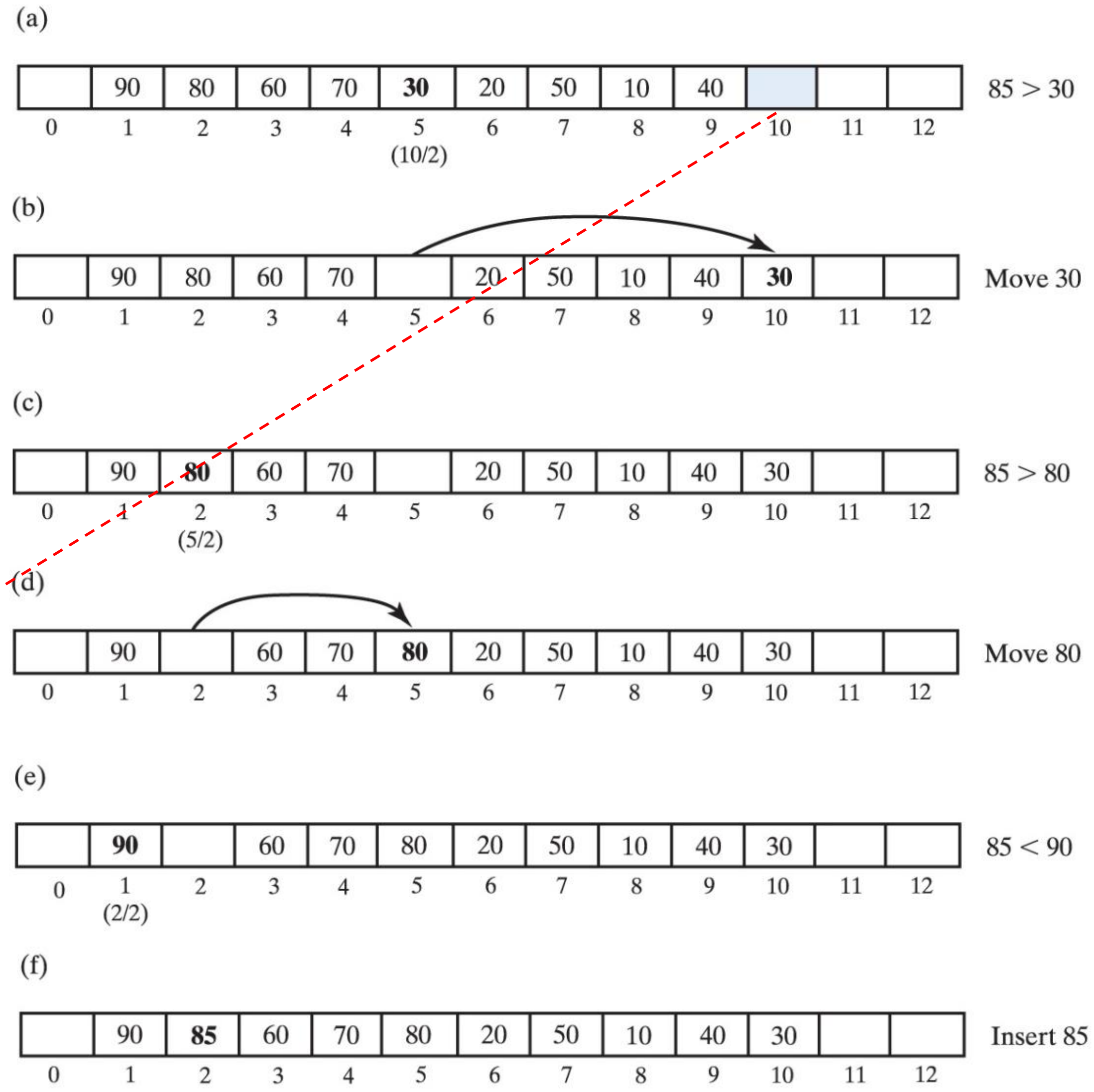
A revision of steps shown in the previous example.

Adding an Entry to a Max Heap

An array representation of the revised upheap insertion.



First available leaf position (10)



Adding an Entry to a Max Heap

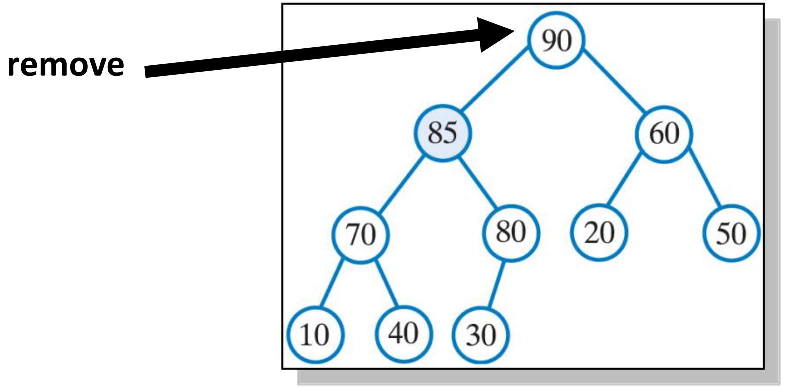
```
public void add (T newEntry)
{
    lastIndex++;
    if (lastIndex >= heap.length)
        doubleArray (); // expand array
    int newIndex = lastIndex;
    int parentIndex = newIndex / 2;
    while ((parentIndex > 0) &&
            newEntry.compareTo (heap [parentIndex]) > 0)
    {
        heap [newIndex] = heap [parentIndex];
        newIndex = parentIndex;
        parentIndex = newIndex / 2;
    } // end while
    heap [newIndex] = newEntry;
} // end add
```

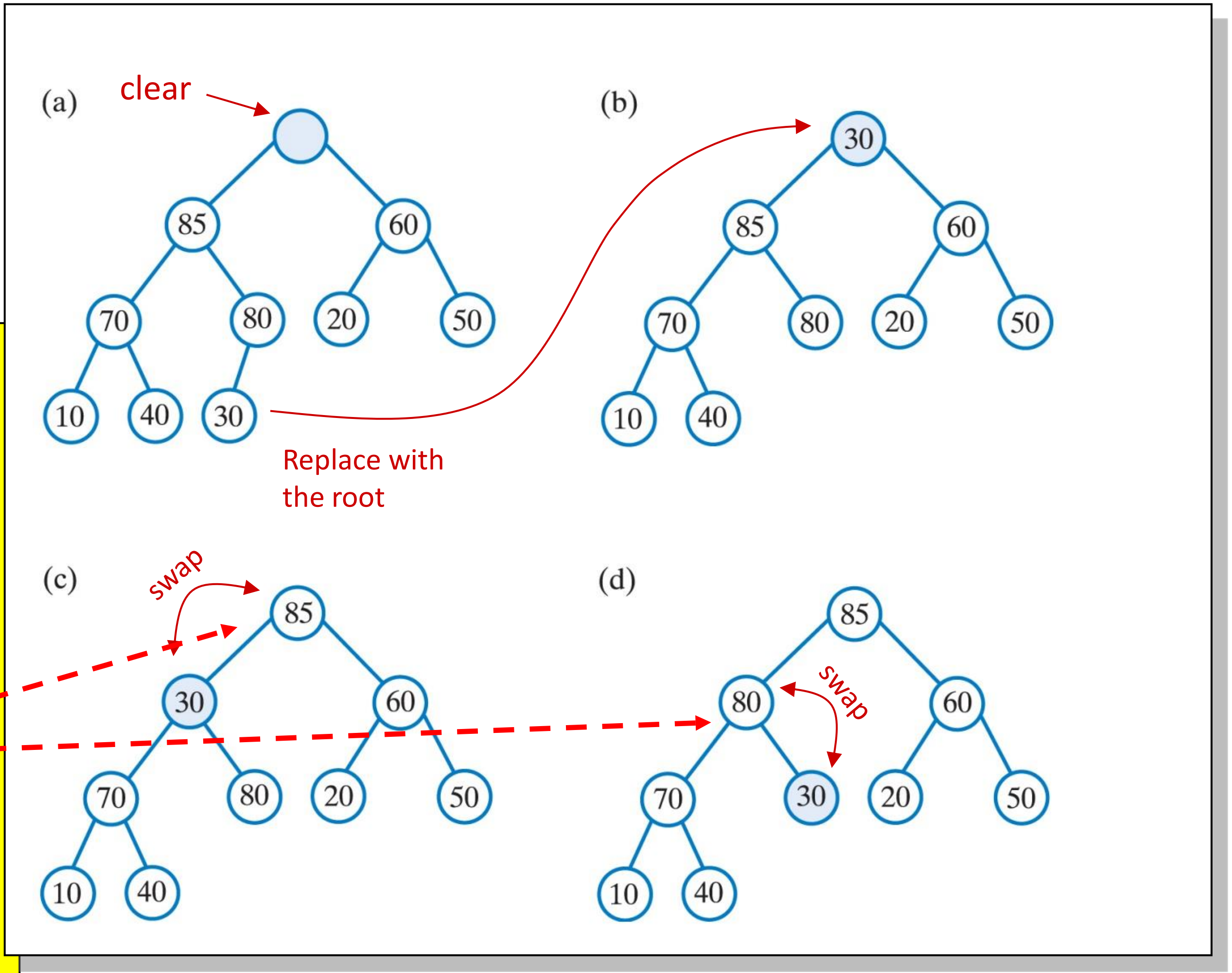
Note: The method add() is an O(log n) operation; adding n different entries in an array will make the operation to be O(n log n)

Note: if index 0 is used for the root.

```
parentIndex =  (newIndex-1)/2;
while(prentIndex >=0 && ….){
    heap[newIndex] = heap[parentIndex];
    if(prentIndex == 0) break;
    newIndex = parentIndex;
    parentIndex =  (newIndex-1)/2;
}
…..
}
```

Removing the Root of a Max-Heap





- To remove a heap's root; View method removeMax (later)
  - Replace the root with heap's last leaf
- This forms a semiheap
- Then use the method reheap () [check the algorithm on the next slide; it's a down-heap process that works exactly like upheap. Down-heap process <u>starts from the root and goes all the way down to the last level</u> using the height, and so it uses O(log n) time]
  - Move the elder (larger) child up to the root of the tree

reheap() Algorithm (Pseudo-Code) for Max Heap

```
Algorithm reheap(rootIndex) // It is a down-heap process same as upheap
// Transforms the semi-heap, rooted at root-Index, into a heap
done = false
orphan = heap[rootIndex]
while (! done and heap [rootIndex] has a child) {
      largerChildIndex = index of the larger child of heap [rootIndex]
      if (orphan < heap[largerChildIndex]) {
         heap[rootIndex] = heap[largerChildIndex]
         rootIndex = largerChildIndex
      } else
         done = true
}
heap [rootIndex] = orphan
```

Down-heap process <u>starts from the root of a heap (which is a heap-tree except for the root), and goes all the way down to the last level</u> using the height, and so it uses O(log n) time.

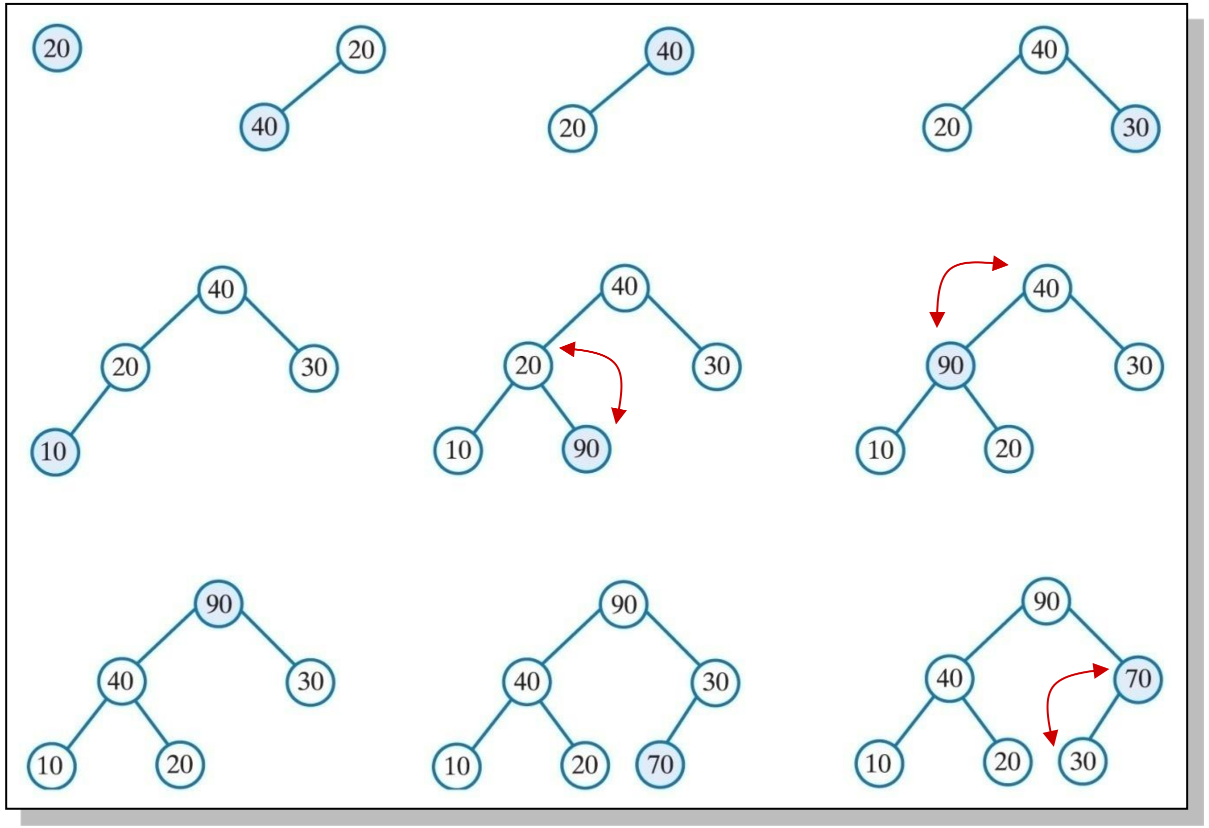
```java
//Implementation of reheap() Method, which is a down-heap process same as upheap
private static <T extends Comparable<? Super T> void reheap(T[] heap, int
rootIndex, int lastIndex) {
  boolean done = false;
  T orphan = heap[rootIndex];
  int leftChildIndex = 2 * rootIndex;
  while (!done && (leftChildIndex <= lastIndex) ) {
    int largerChildIndex = leftChildIndex; // assume larger
    int rightChildIndex = leftChildIndex + 1;
    if ( (rightChildIndex <= lastIndex) &&
          heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0) {
      largerChildIndex = rightChildIndex;
    } // end if
    if (orphan.compareTo(heap[largerChildIndex]) < 0) {
      heap[rootIndex] = heap[largerChildIndex];
      rootIndex = largerChildIndex;
      leftChildIndex = 2 * rootIndex;
    }
    else
      done = true;
  } // end while
  heap[rootIndex] = orphan;
} // end reheap
```

removeMax() method

```
public T removeMax (T[] heap, int lastIndex)
{
    T root = null;
    if (!isEmpty ())
    {//in this implementation, the root is in index 1
        root = heap [1]; // return value;
        heap [1] = heap [lastIndex]; // form a semiheap
        lastIndex--; // decrease size
        reheap (1); // transform to a heap
    } // end if
    return root;
} // end removeMax
```

Creating a Heap (Max Heap)

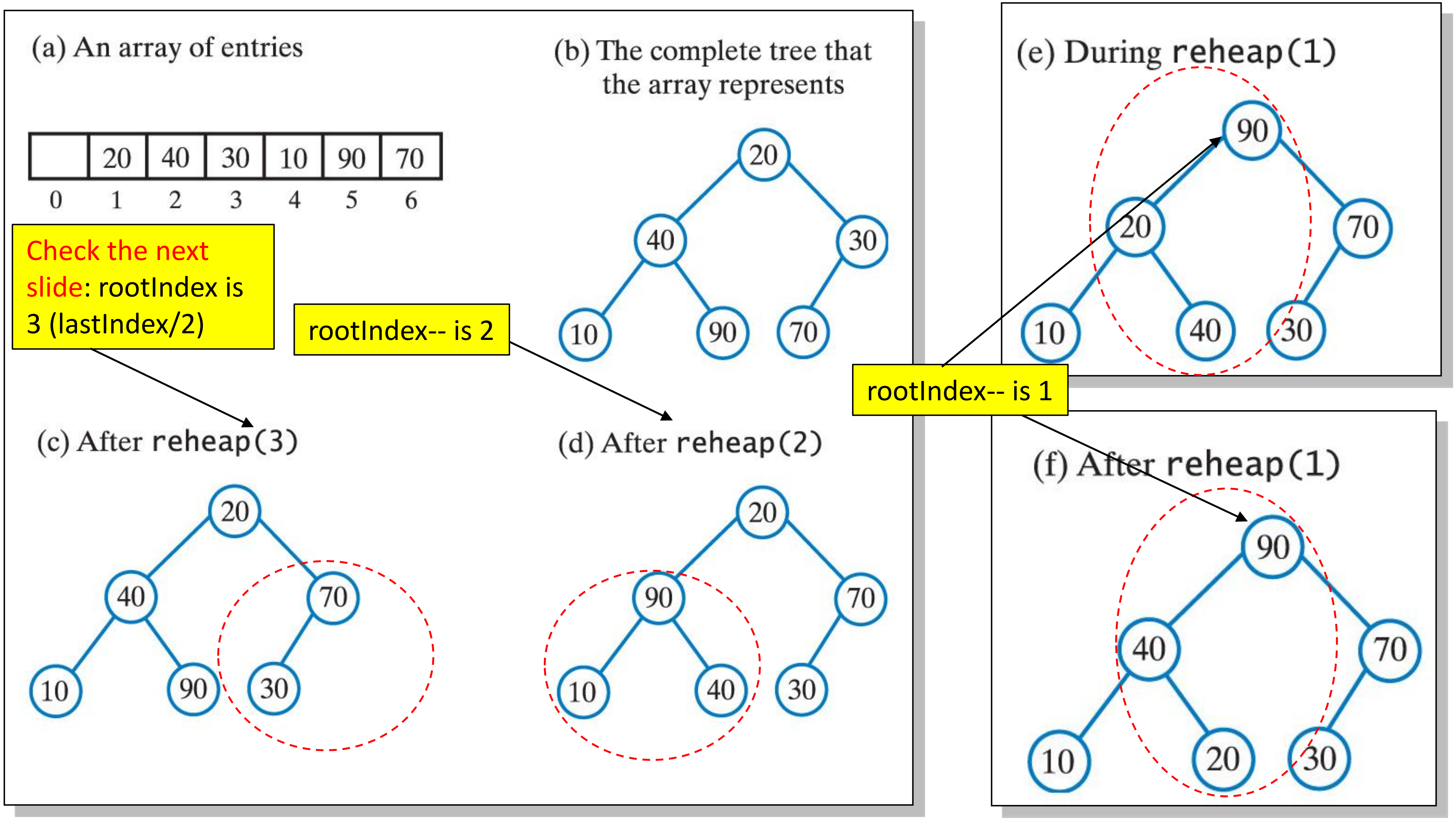
Build a Heap tree by adding each object to an initially empty heap from the following numbers: 20, 40, 30, 10, 90, 70



Note: adding n different entries in an array will make the operation to be $O(n \log n)$

Creating a Heap from a given binary tree (*Heapify*)

The steps in creating a heap by using **`reheap (index)`** takes O(n/2) = O(n) operations. In this case, the leaves are already in a heap with no-children, and the comparison starts from the last leaf's parent.



Java statement in Transforming a Complete Tree into a maxheap – The high-Level Description

- Note: The array entries are located from the index 1 to the last index. (Note: you can start from index zero too, as discussed before)
- In applying reheap() we begin at the first non-leaf (at level h-1) closest to the end of the array.
  - This non-leaf is at index `lastIndex/2`.
- Then we work toward heap[1]
  - The statement:

```
for (int rootIndex = lastIndex/2; rootIndex>0; rootIndex--)
        reheap(rootIndex);
```

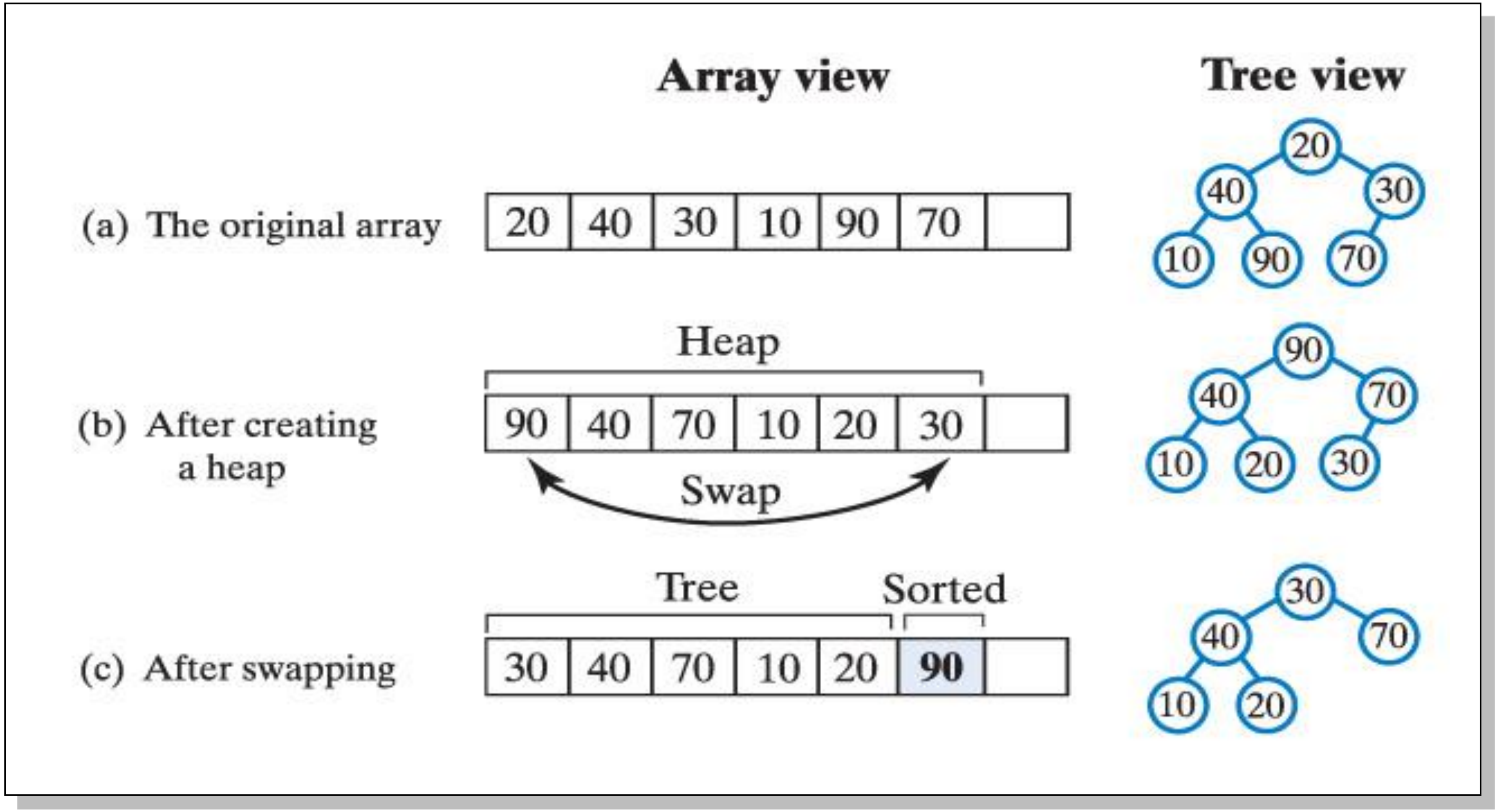
Heapsort

- The High-level Description of the Heapsort Algorithm:
  - Step 1: Place array items into a maxheap [O(n)]
  - Step 2: Swap the root with the content of the last index.
  - Step 3: Decrement the last-index of the array.
  - Step 4: Re-heap (we do down-heap here, same as upheap) [O(logn)]
  - Repeat steps 2 to 4 till the last index is greater than or equal to zero [O(n logn)]
  - The resultant array will be sorted in ascending order.
  - Note: For descending order, we can use minheap.
  - [(1) HEAP-sort with Hungarian (MEZŐSÉGI) folk dance - YouTube]

Generic Implementation of heapSort

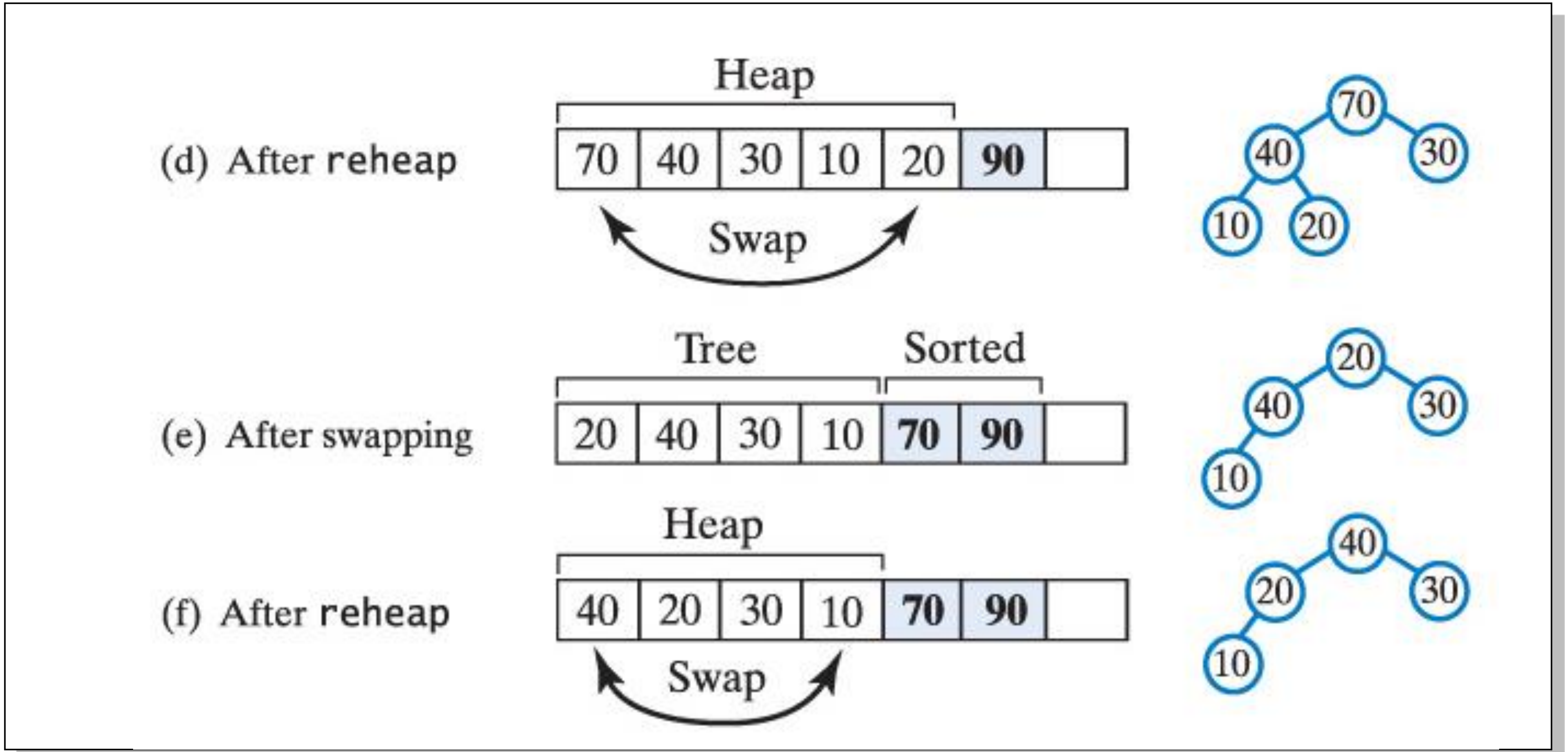
```
public static < T extends Comparable < ? super T >>
     void heapSort (T [] array, int n) {
   // create the heap first; root is in index 0 of the array
   for (int rootIndex = n / 2 - 1 ; rootIndex >= 0 ; rootIndex--)
        reheap (array, rootIndex, n - 1); // This is O(n/2) = O(n)

   swap (array, 0, n - 1);
   for (int lastIndex = n - 2 ; lastIndex > 0 ; lastIndex--) {
        reheap (array, 0, lastIndex); // down-heaping from index 0
        //This downheap like upheap takes )(log n)
        swap (array, 0, lastIndex);
   } // end for
} // end heapSort
```

Tracing Heapsort (descending order)



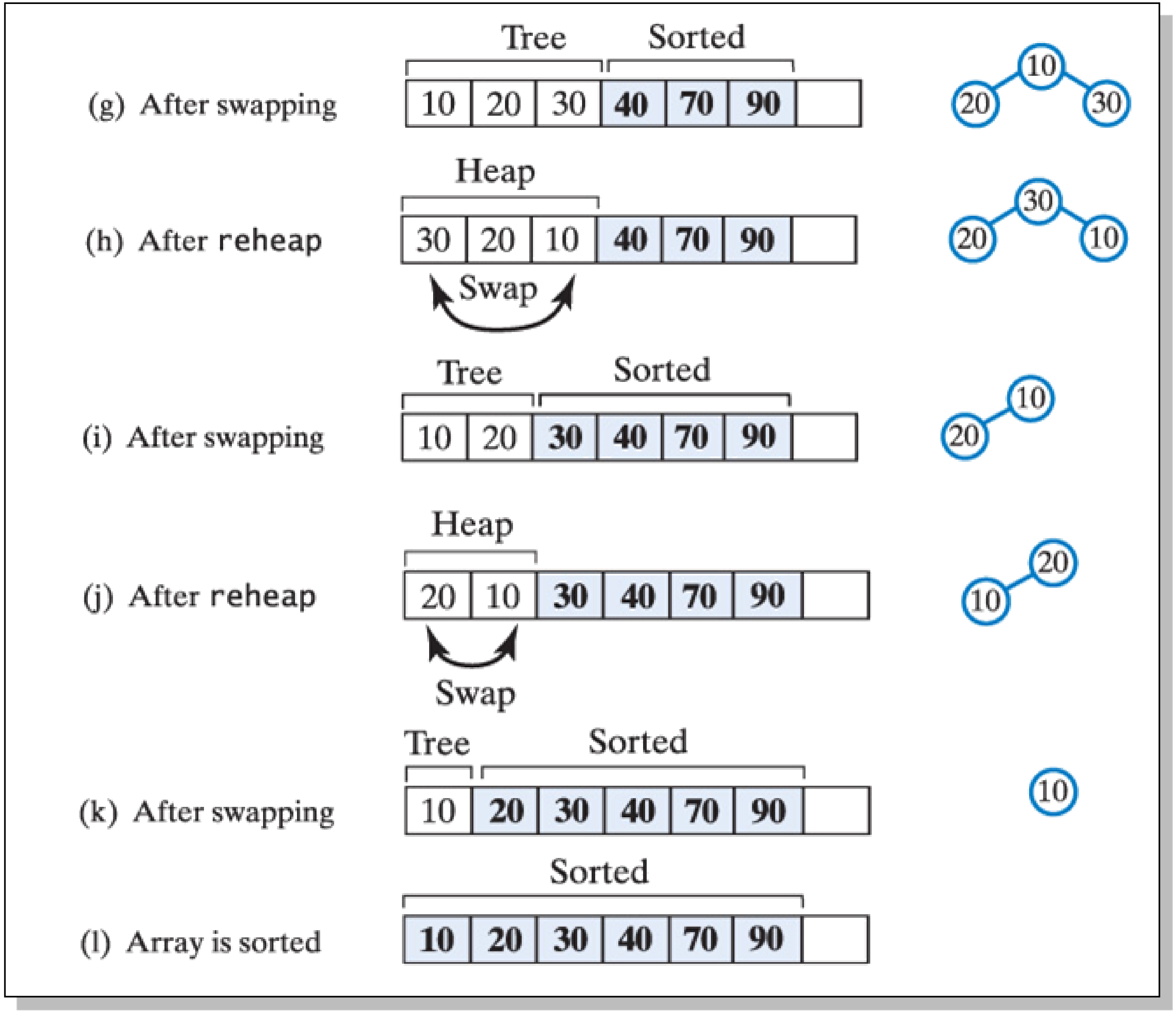
A trace of heapsort (a – c)

Tracing Heapsort



A trace of heapsort (d – f)

Tracing Heapsort



A trace of heapsort (g – l)

Efficiency of Heapsort and more on Sorting

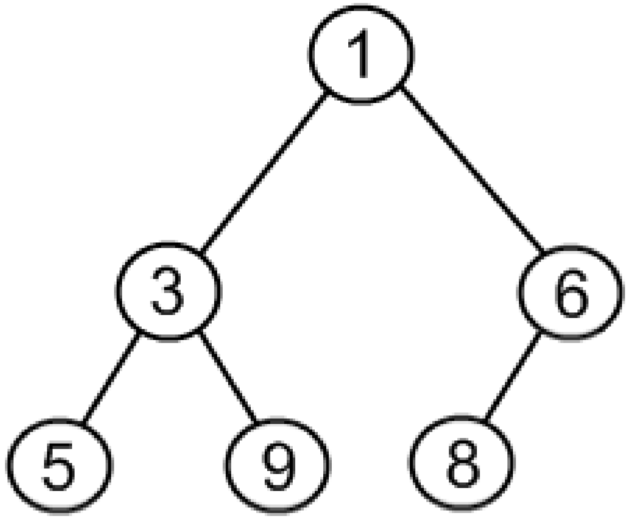
- Heapsort is an efficient, unstable sorting algorithm with an average, best-case, and worst-case time complexity of *O(n log n)*.
- Heapsort is significantly slower than Quicksort and Merge Sort, so Heapsort is less commonly encountered in practice.
  - Note: Merge and quick sort algorithms have the best and average case efficiency of O(n log n); worst case for merge sort is O(n log n) and for quick sort is $O(n^2)$
  - Unlike merge sort that requires O(n) space, Heap-sort does not require a second array
  - Since we can avoid quick sort's worst case by choosing appropriate pivots, it is generally the preferred sorting method.
  - In real-time scenarios, where we have a fixed amount of time to perform a sorting operation and the input data can fit into main memory, the heap-sort algorithm is probably the best choice. It can be made to execute in-place.

In-Class Discussion

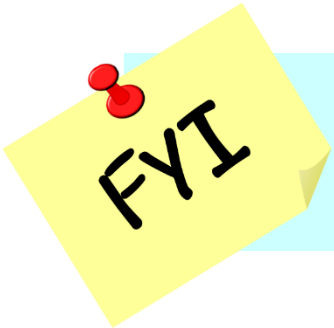
Build a Max-Heap and a Min-Heap from [3,1,6, 5, 2, 4].

In-Class Discussion

- Insert “10” to the heap below.
- Insert “0” to the heap below.
- What is the time complexity of the insertion operation?



End of this Unit

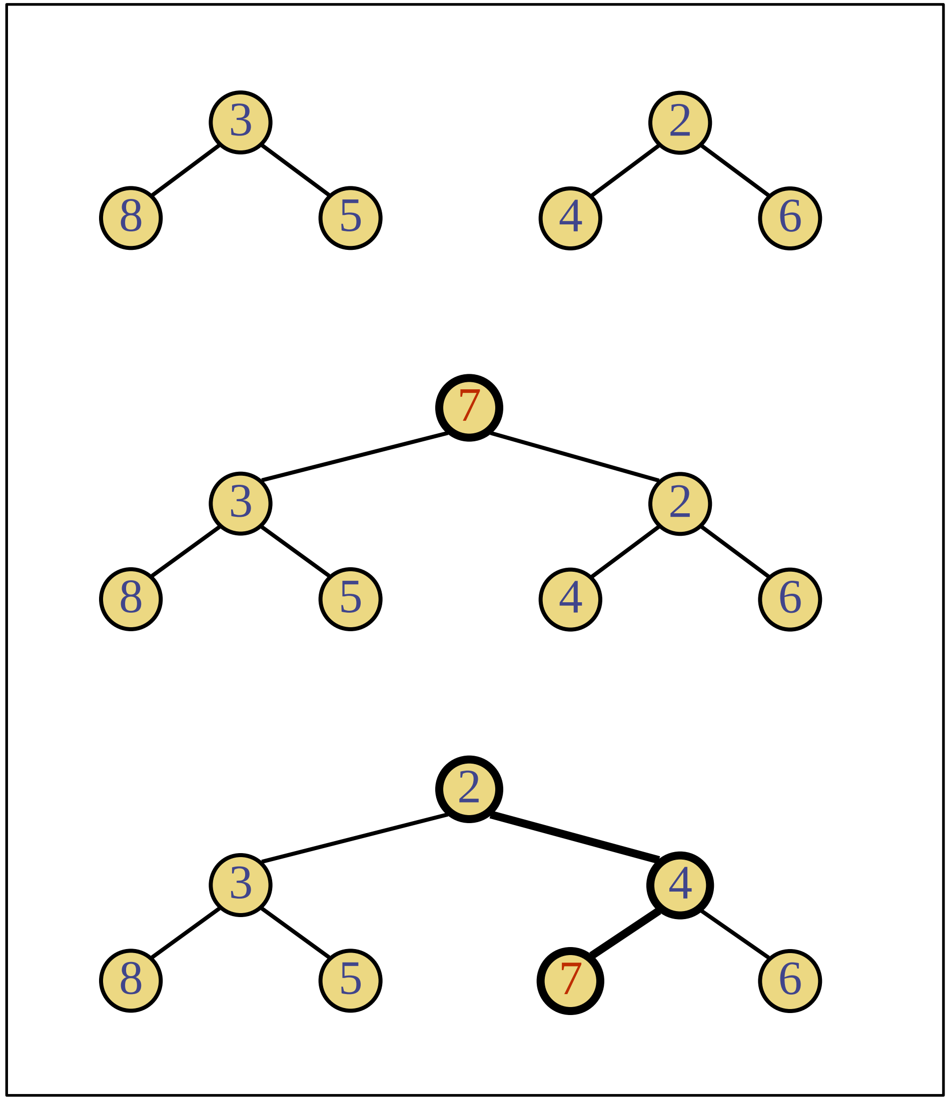


Appendix: Merging Heaps

- This section has been added so that the interested readers can read this section and get ready for any future interview with any software industry.

Merging Two Min-heaps using a key

- A heap can be created from two given heaps and a key *k*
- We create the new heap with the root node storing the key *k* and with the two given heaps as subtrees
- We perform downheap to restore the heap-order property
- The principle involved in downheap operation for minheap structure: the younger (smaller in weight) child is moved up to the root of the subtree
- We can follow the same procedure for max heap.



Example: Creating a minHeap by Merging – Bottom-up Solution

- For simplicity of the explanation let's assume we are given an *n* ($n = 2^{h+1}-1$) elements for a full tree (perfect binary tree) of height of *h*. For a height of 3, we have 15 elements with keys: 16, 15, 4, 12, 6, 7, 23, 30, 25, 5, 11, 27, 7, 8, 10
- First, we construct (n+1)/2 (integer division) element as:

- Add one more key (for each pairs) from ((n+1)/2)/2 more keys and do the merge process (this procedure continues till no element is left):

Example: Creating a minHeap by Merging *contd..*

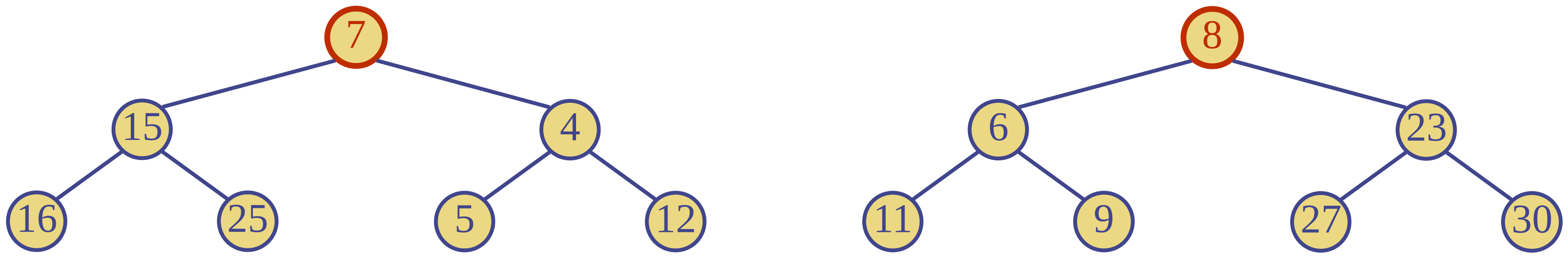
16, 15, 4, 12, 6, 7, 23, 30, 25, 5, 11, 27, 7, 8, 10

- Downheap process:

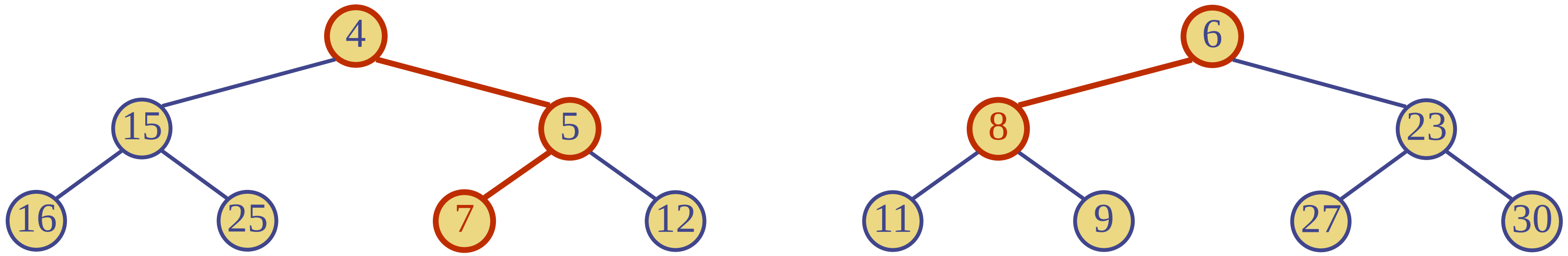
Example: Creating a minHeap by Merging *contd..*

16, 15, 4, 12, 6, 7, 23, 30, 25, 5, 11, 27, 7, 8, 10

- Add one more key for each two subtrees and do the merge process:



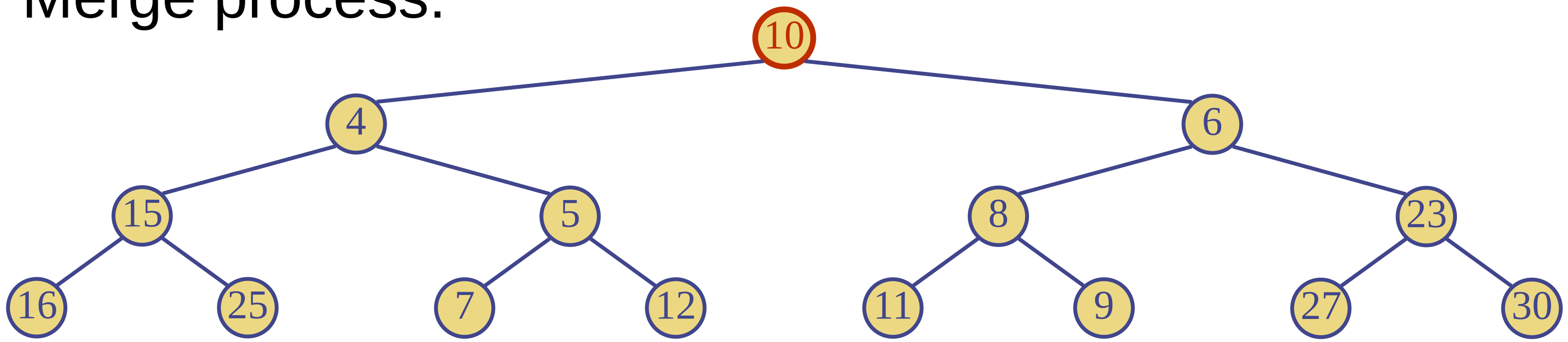
- Downheap process:



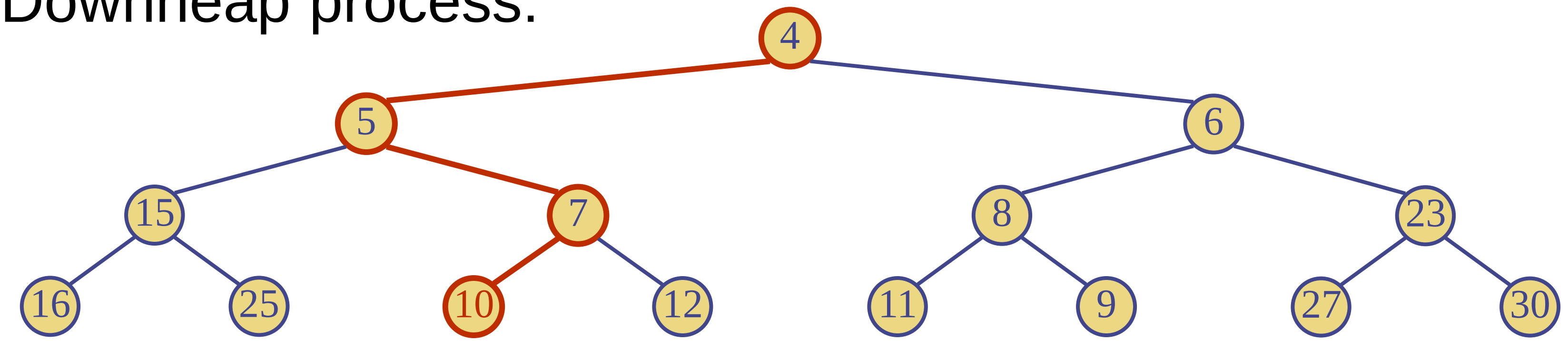
Example: Creating a minHeap by Merging *contd..*

16, 15, 4, 12, 6, 7, 23, 30, 25, 5, 11, 27, 7, 8, 10

- Merge process:



- Downheap process:



Note: It can be shown that Bottom-up heap construction through merging is faster (O(n)) than ***n*** successive insertions (nlogn).