Ch.13 Priority Queues

BIRD'S-EYE VIEW (0)

- Chapter 12: Binary Tree
- Chapter 13: Priority Queue
  - Heap and Leftiest Tree
- Chapter 14: Tournament Trees
  - Winner Tree and Loser Tree

BIRD'S-EYE VIEW (1)

- A priority queue is efficiently implemented with the heap data structure
- Priority data structure
  - Heap
  - Leftist tree
- Priority Queue Applications
  - Heap sort
    - Use heap for an O(n*logn) sorting method
  - Machine scheduling
    - Use the heap data structure to obtain an efficient implementation
  - The generation of Huffman codes

Table of Contents

- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

Definition

- A priority queue is
  - Collection of zero or more elements with priority
- A min priority queue is
  - Find the element with minimum priority
  - Then, Remove the element
- A max priority queue is
  - Find the element with maximum priority
  - Then, Remove the element

- Priority queue is a conceptual queue where the output element has a certain property (i.e., priority)

15	35	65	20	17	80	12	45	2	4

Priority Queue Applications

- Priority queues in the machine shop simulation
  - Min priority queue
    - One machine, many jobs with priority (time requirement of each job), a fixed rate of payment
    - To maximize the earning from the machine
      - When a machine is ready for a new job, it selects the waiting job with minimum priority (time requirement)
    - getMin() & removeMin()
  - Max priority queue
    - Same duration jobs with priority (the amount of payment)
    - To maximize the earning from the machine
      - When a machine is ready for a new job, it selects the waiting job with maximum priority (the amount of payment)
    - getMax() & removeMax()

The ADT MaxPriorityQueue

```
AbstractDataType MaxPriorityQueue {
    instances
        finite collection of elements, each has a priority
    operations
        isEmpty()     : return true if the queue is empty
        size()        : return number of elements in the queue
        getMax()      : return element with maximum priority
        put(x)        : insert the element x into the queue
        removeMax()   : remove the element with largest priority
                        from the queue and return this element;
}
```

Table of Contents

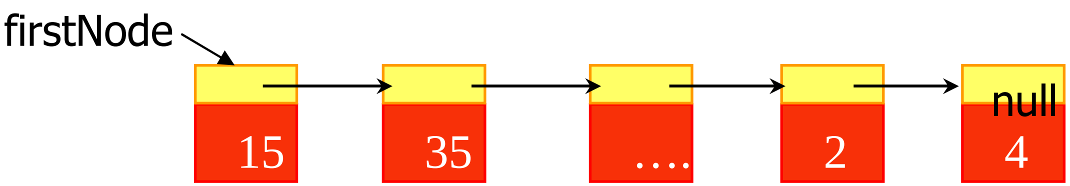
- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

Linear Lists for Priority Queue (1)

- Suppose Linear List for max priority queue with n elements
- Unordered linear list for a max queue
  - Array
    - Insert() or Put() : Θ(1) // put the new element the right end of the array
    - RemoveMax(): Θ(n) // find the max among n elements

15	35	65	20	17	80	12	45	2	4

  - Linked List
    - Insert() or Put() : Θ(1) // put the new element at the front of the chain
    - RemoveMax(): Θ(n) // find the max among n elements

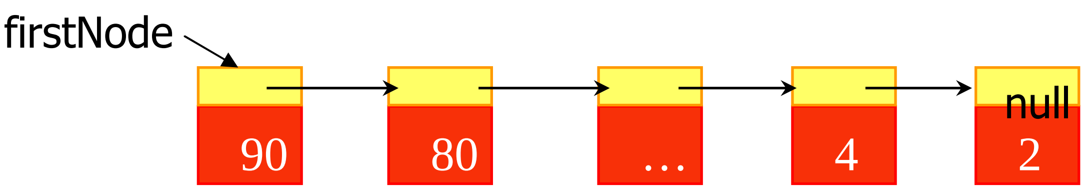


Linear Lists for Priority Queue (2)

- Ordered linear list for a max queue
  - Array
    - location(i) = i (i.e., array-based) where the max element is located in the last address (i.e., the nondecreasing order)
    - Insert() or Put() : Θ(n)
    - RemoveMax() : Θ(1)

2	4	13	20	....	80	90			

  - Linked List
    - chain (i.e., linked) where the max element is located in the head of chain (i.e., the nonincreasing order)
    - Insert() or Put() : Θ(n)
    - RemoveMax() : Θ(1)



Why HEAP?

- 0(logN) for Insert() or Put()
- 0(logN) for RemoveMax()

- Simple Array Implementation!

Table of Contents

- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

Max Tree & Max Heap

- A max tree is a tree in which the value in each node is greater than or equal to those in its children

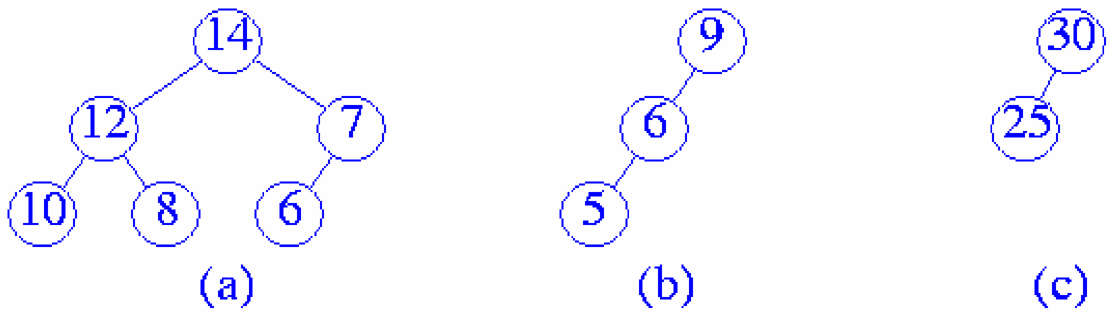


Figure 13.1 Max trees

- A max heap is
  - A max tree that is also a complete binary tree
  - Figure 13.1(b) : not CBT, so not max heap

Min Tree & Min Heap

- A min tree is a tree in which the value in each node is less than or equal to those in its children

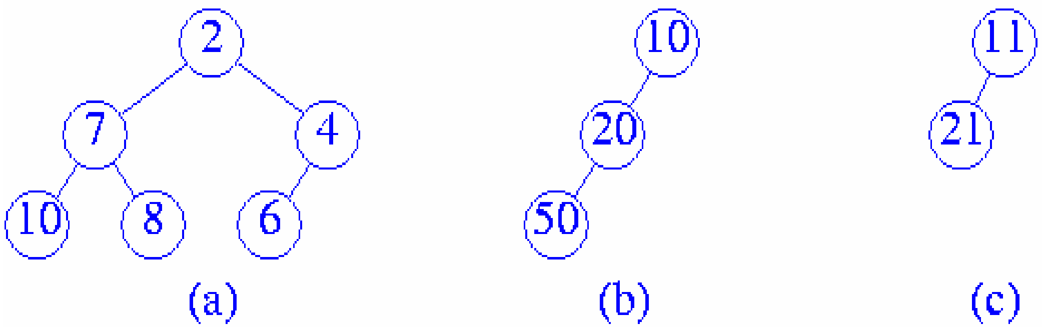


Figure 13.2 Min trees

- A min heap is
  - A min tree that is also a complete binary tree
  - Figure 13.2(b) : not CBT, so not min heap

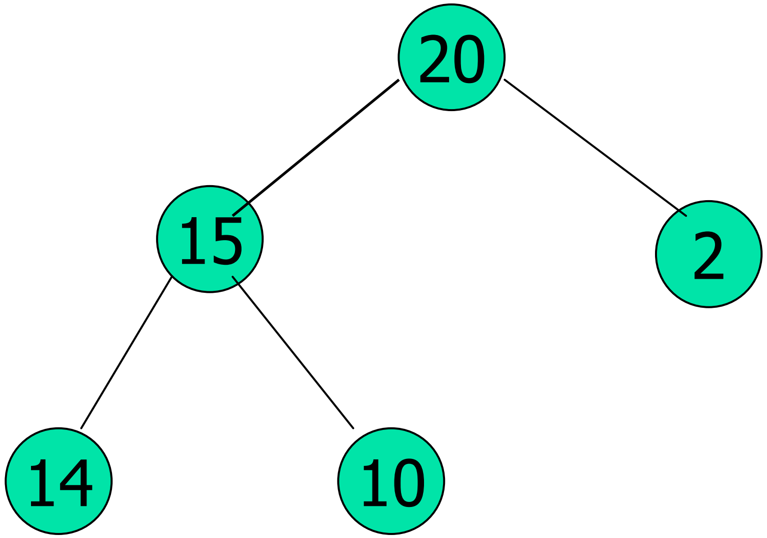
Heap Height

- Heap is a complete binary tree
  - **A heap with n elements has height $\lceil \log_2(n+1) \rceil$**

- put(): O(height) ➔ O(log n)
  - Increase array size if necessary
  - Find place for the new element
    - The new element is located as a leaf
    - Then moves up the tree for finding home
- removeMax(): O(height) ➔ O(log n)
  - Remove heap[1], so the root is empty
  - Move the last element in the heap to the root
  - Reheapify

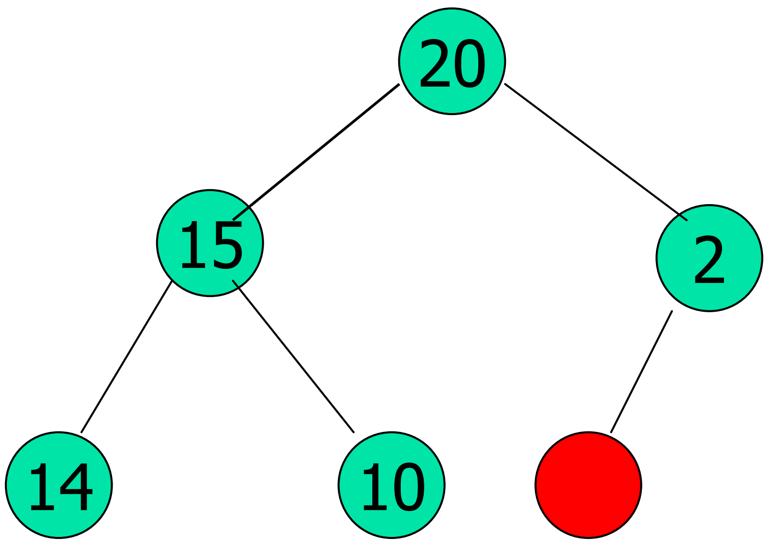
Put into Max Heap (1)

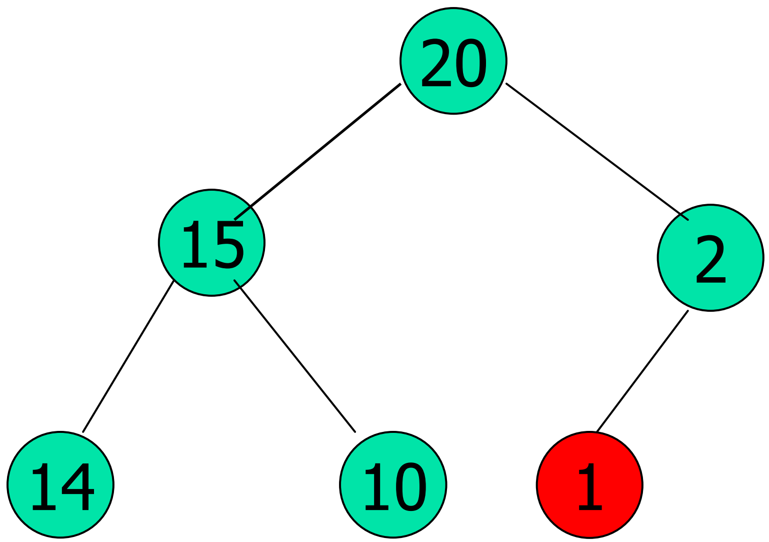
- Max heap with five elements



Put into Max Heap (2)

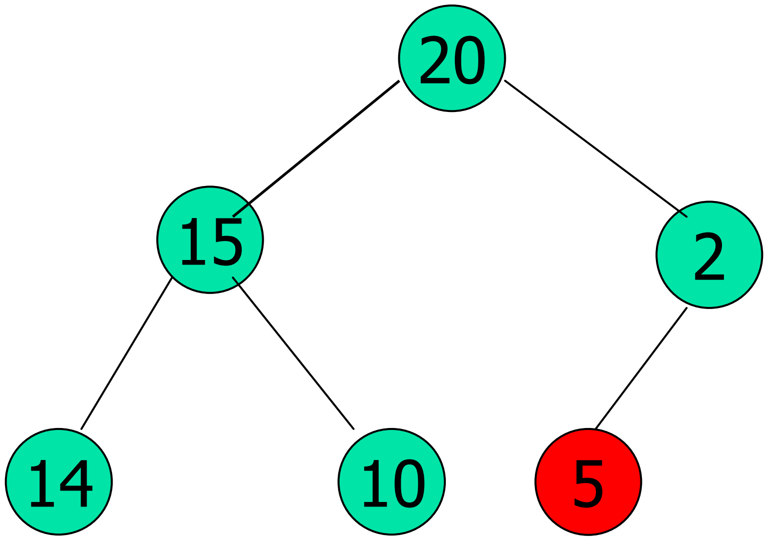
- When an element is added to this heap, the location for a new element is the red zone
- Suppose the element to be inserted has value 1, the following placement is fine



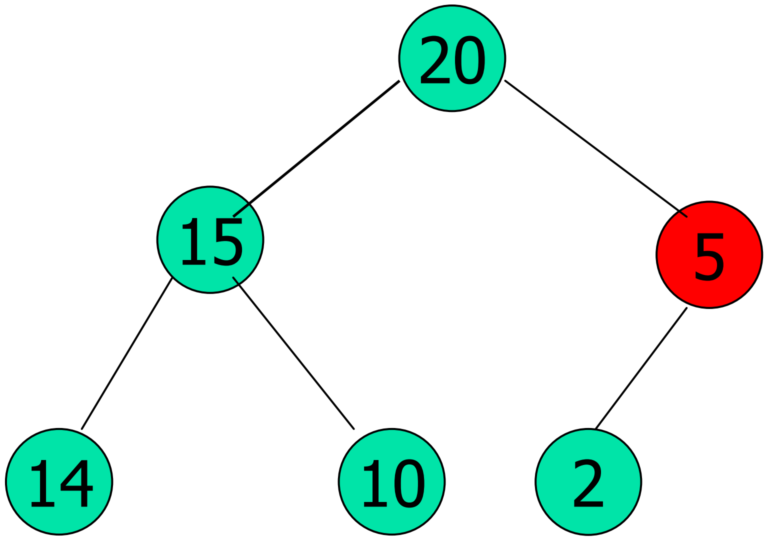


Put into Max Heap (3)

- Suppose the element to be inserted has value 5



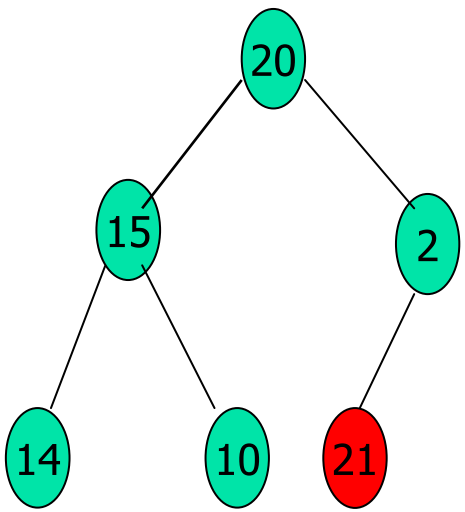
- The elements 2 and 5 must be swapped for maintaining the heap property

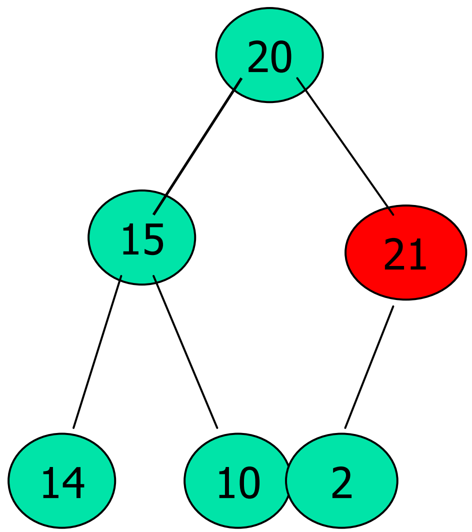


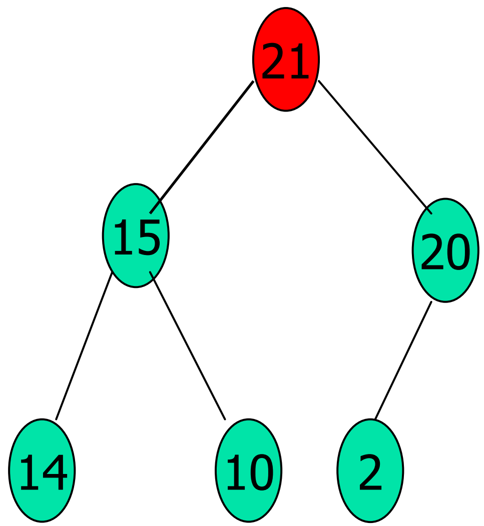
Put into Max Heap (4)

Suppose the element to be inserted has value 21

- The new element 21 will find its position by continuous swapping with the existing elements for maintaining the heap property
- Finally the new element 21 goes to the top







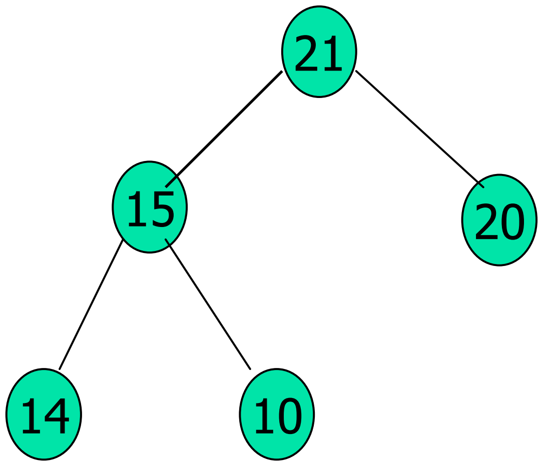
put() in MaxHeap

```
public void put ( Comparable theElement ) {
  if ( size == heap.length – 1 ) // increase array size if necessary
    heap=(Comparable []) ChangeArrayLength.changeLengthID (heap, 2 * heap.length);

  // find the place for theElement. currentNode starts at new leaf and moves up tree
  int currentNode = ++size;
  while (currentNode != 1 &&  heap[currentNode / 2].compareTo(theElement) < 0) {
  // cannot put theElement in heap[currentNode], So move element down
          heap[currentNode] = heap[currentNode / 2];
          currentNode /= 2;    // move to parent
     }

     heap[currentNode] = theElement;
}
```

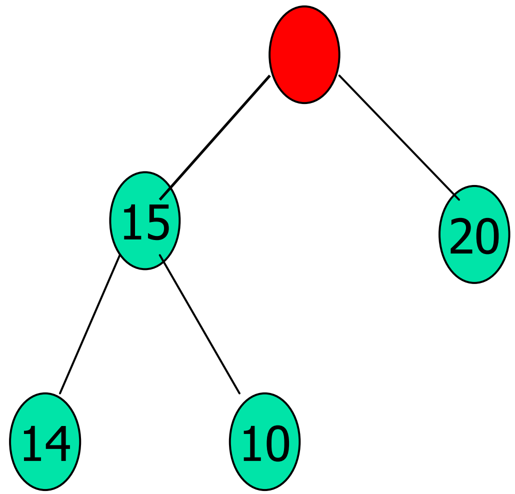
- At each level : ⊖(1)   So, Total complexity: O(height) = O(logn)

removeMax() from a MaxHeap (1)

- The Max element “21” is in the root

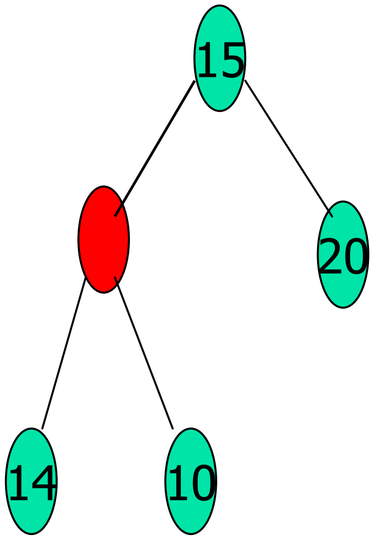


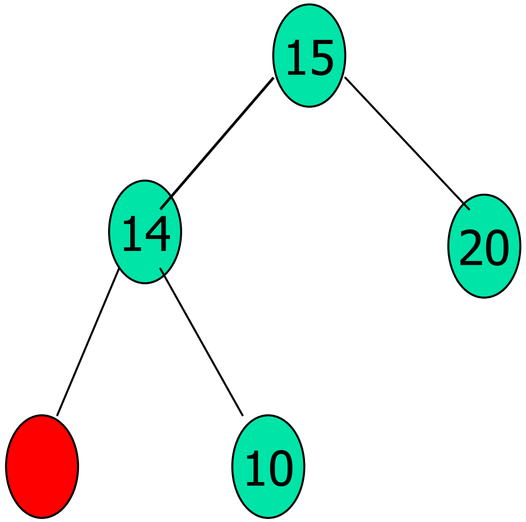
- After the max element “21” is removed

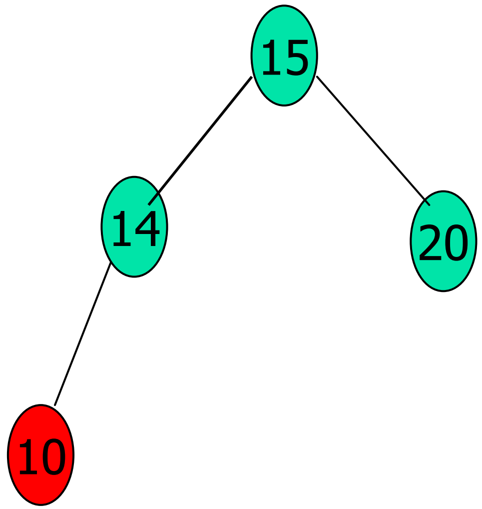


removeMax() from a MaxHeap (2)

- The element 15 will go to the top by swapping
- The element 14 is also swapped to one level up
- Even the element 10 needs to relocated for maintaining the complete binary tree property







removeMax() in MaxHeap

```
public Comparable removeMax()  {
    if (size == 0) return null; // if heap is empty return null
    Comparable maxElement = heap[1];  // max element
    Comparable lastElement = heap[size--]; // reheapify
    // find place for lastElement starting at root
    int currentNode = 1,  child = 2;     // child of currentNode
    while (child <= size) {  // heap[child] should be larger child of currentNode
        if (child < size && heap[child].compareTo(heap[child + 1]) < 0)  child++;
        // can we put lastElement in heap[currentNode]?
        if (lastElement.compareTo(heap[child]) >= 0)   break;   // yes
        heap[currentNode] = heap[child]; // no   // move child up
        currentNode = child;            // move down a level
        child *= 2; }
     heap[currentNode] = lastElement;
     return maxElement;
}
```

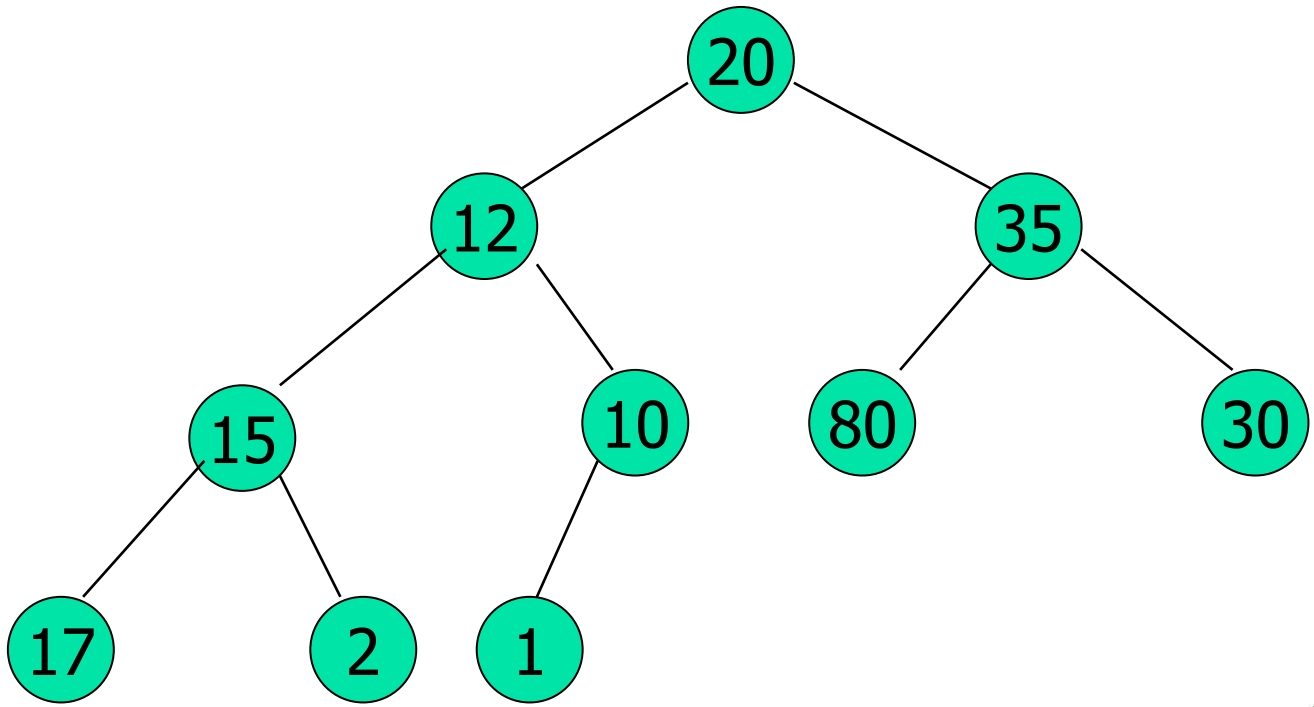
** At each level ⊖(1), So complexity: O(height) = O(logn)

MaxHeap Initialization

- **Steps**
  - **Allocate the elements in an array**
  - **Form a complete binary tree**
  - **In the array, start with the rightmost node having a child**
    - **node number → n/2**
  - **Fix the heap in the node**
  - **Reverse back to the first node in the array**

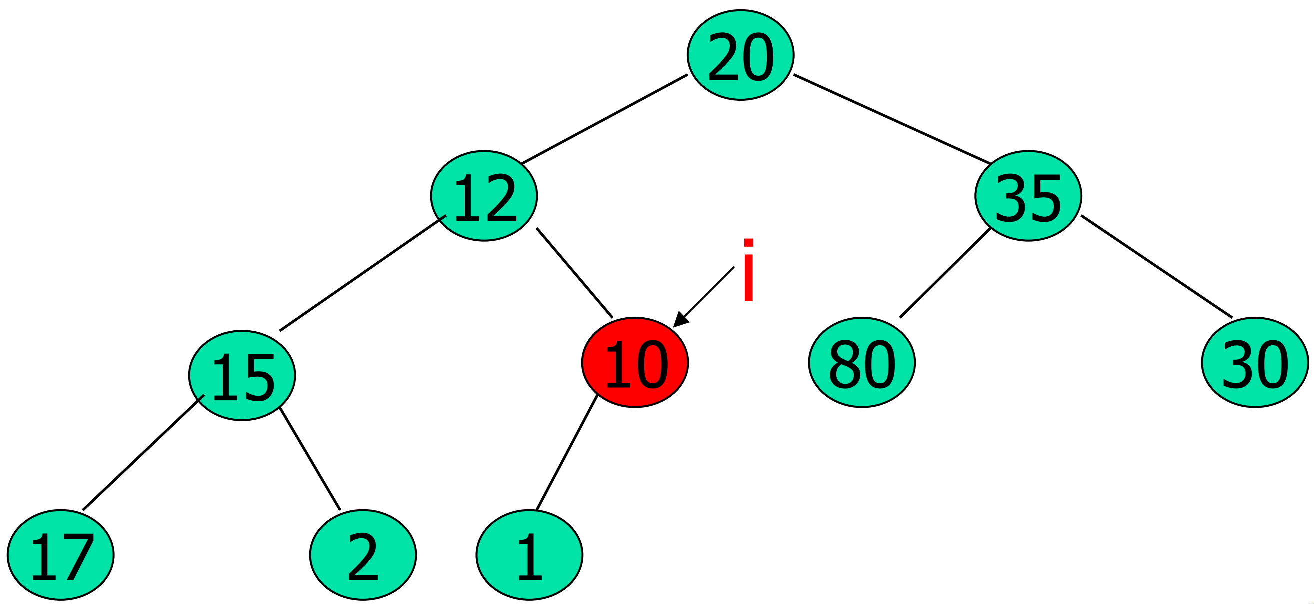
MaxHeap Initialization (1)

- Input array = [20, 12, 35, 15, 10, 80, 30, 17, 2, 1]
- Just make a complete binary tree



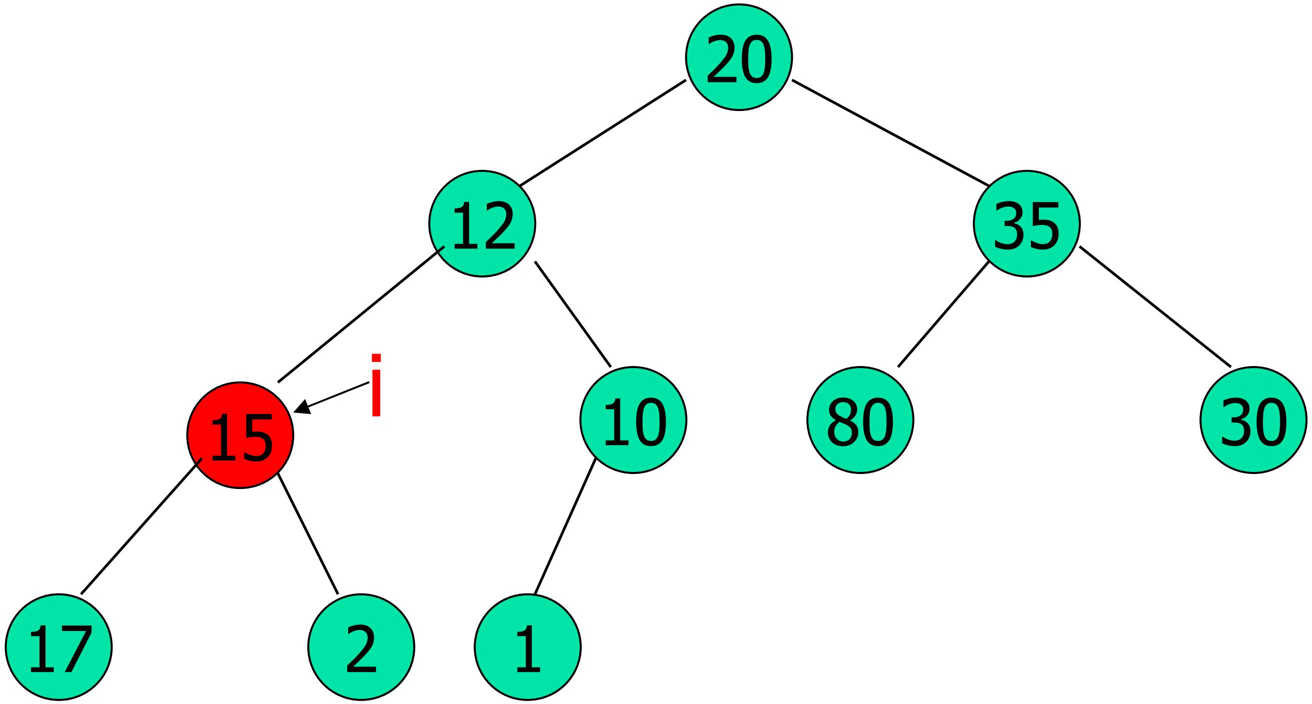
MaxHeap Initialization (2)

- Start at rightmost array position that has a child.
- Index i is $(n/2)^{th}$ of the array.



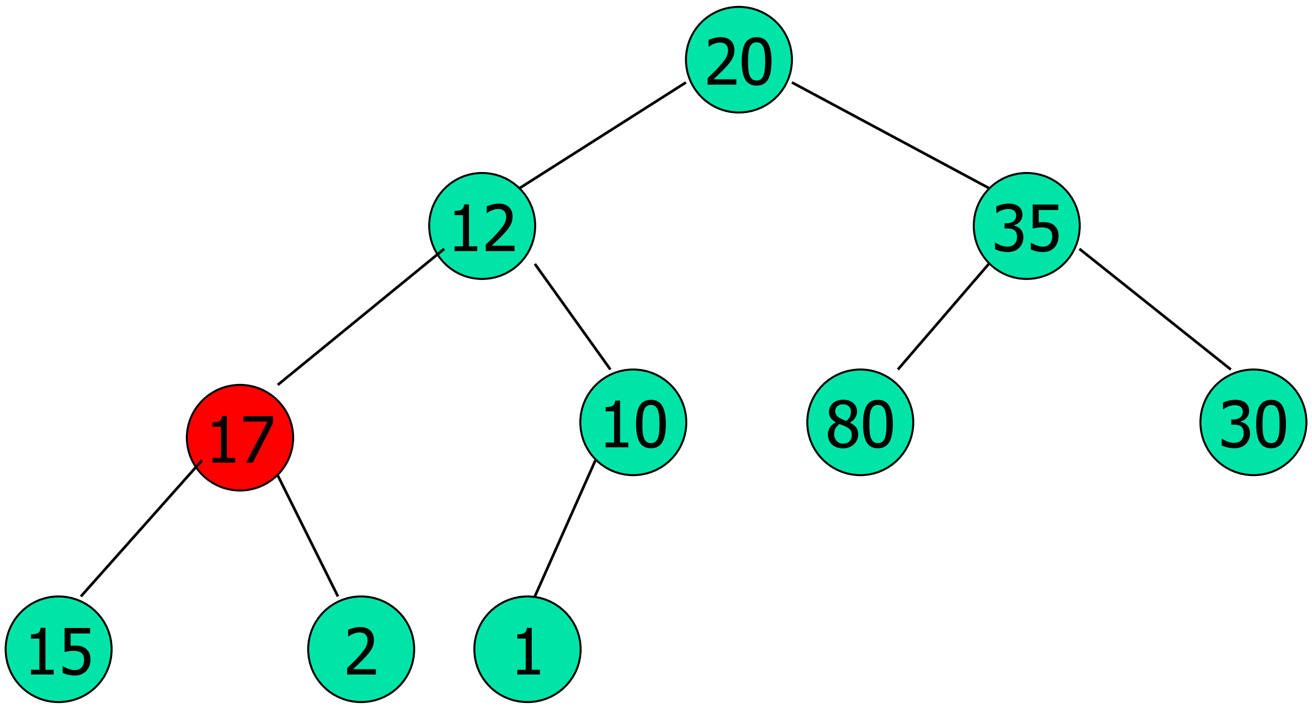
MaxHeap Initialization (3)

- Move to next lower array position.



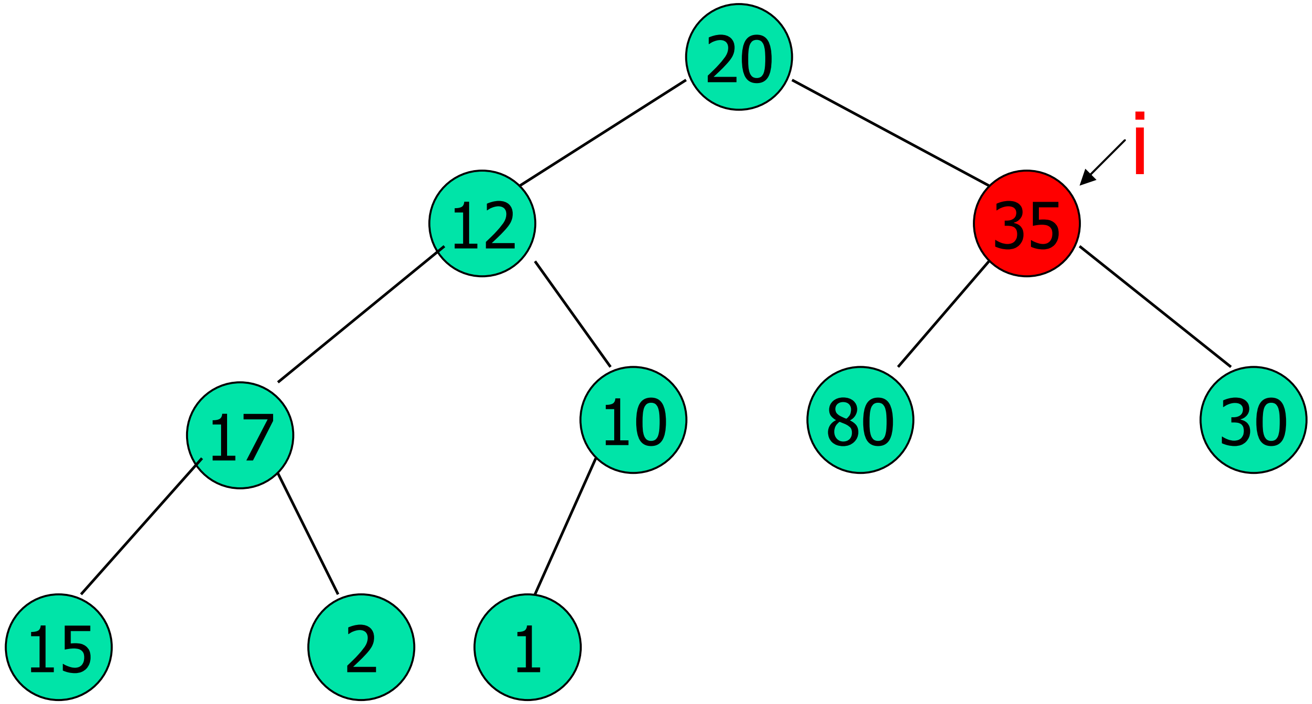
MaxHeap Initialization (4)

- Find a home for 15



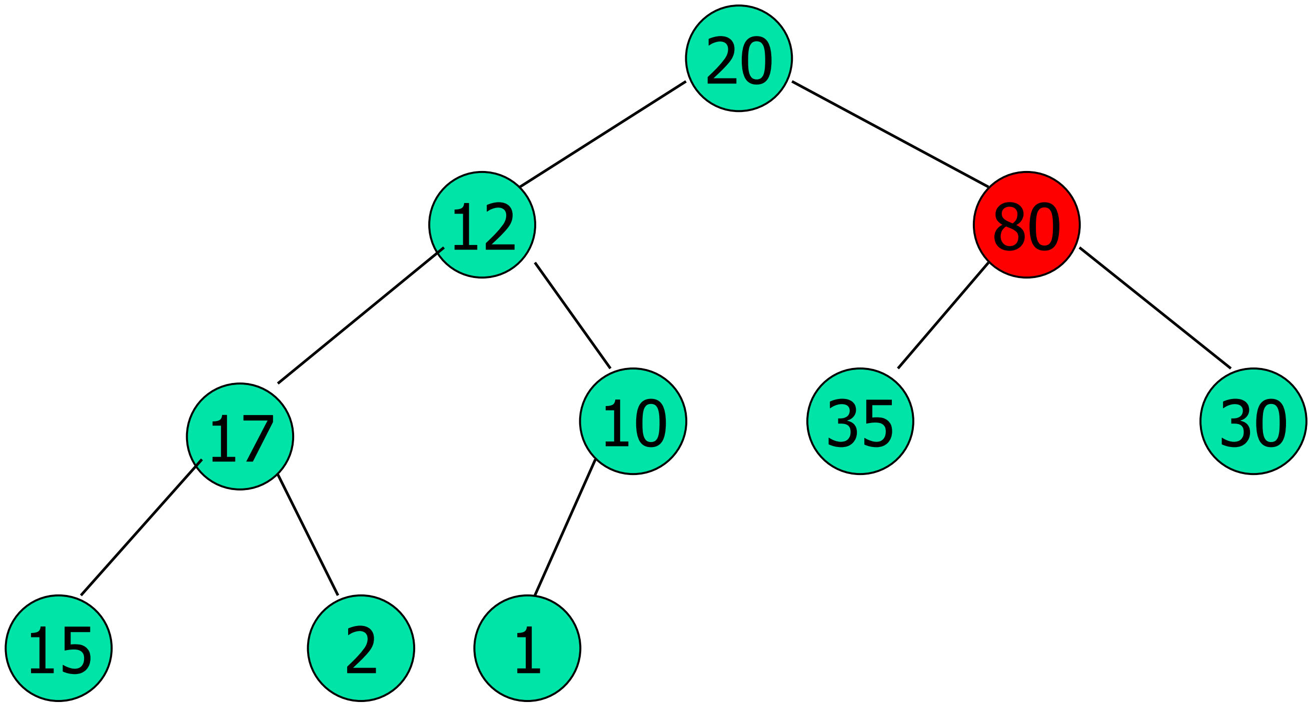
MaxHeap Initialization (5)

- Move to next lower array position.



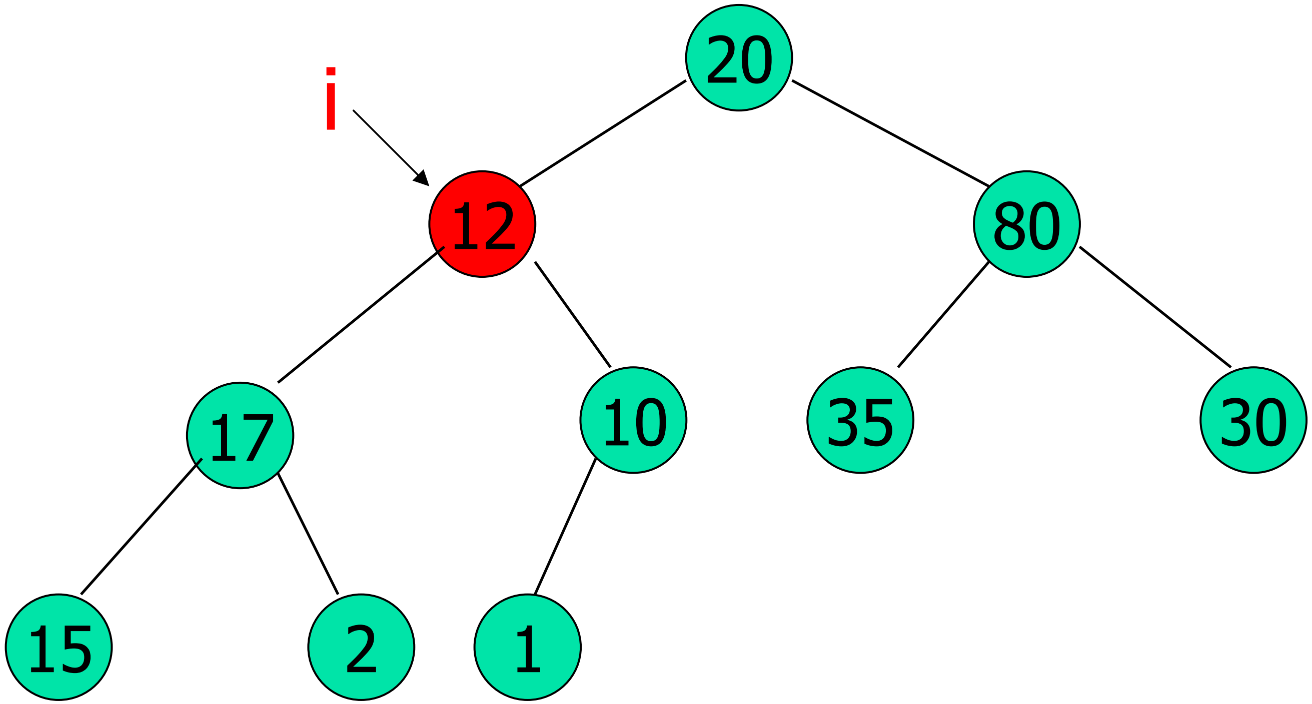
MaxHeap Initialization (6)

- Find a home for 35



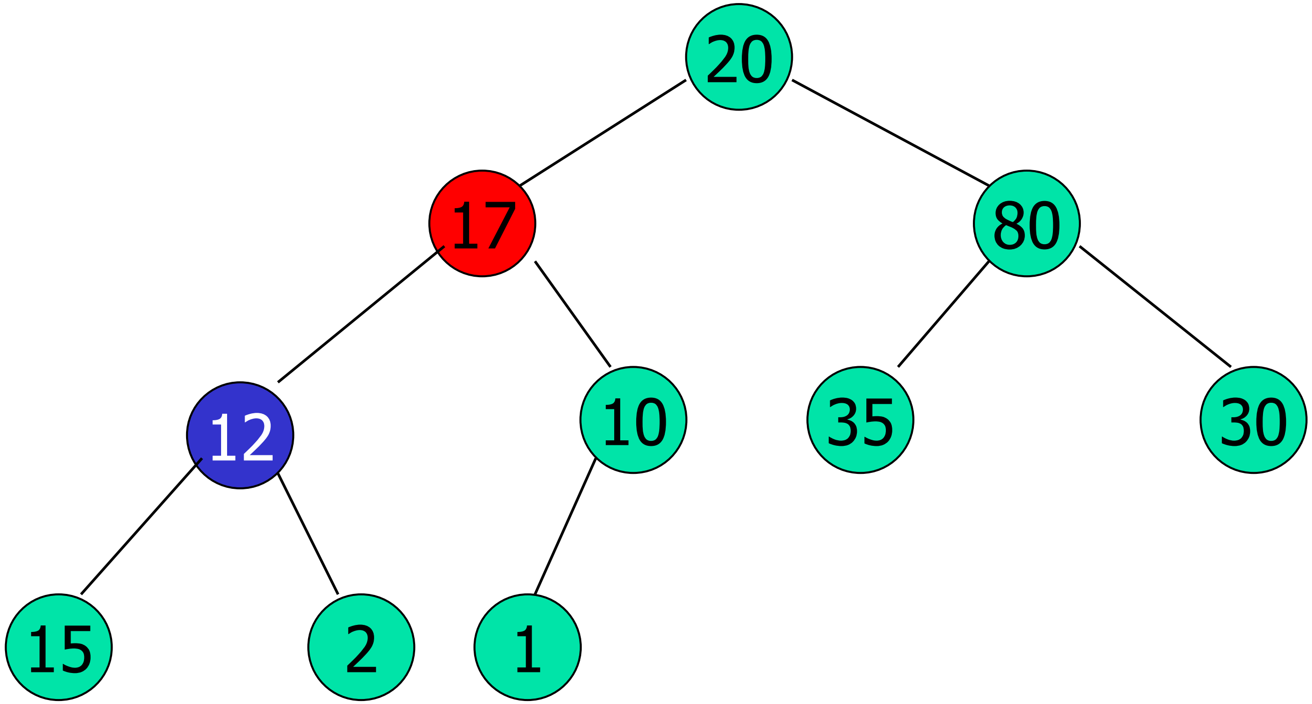
MaxHeap Initialization (7)

- Move to next lower array position.



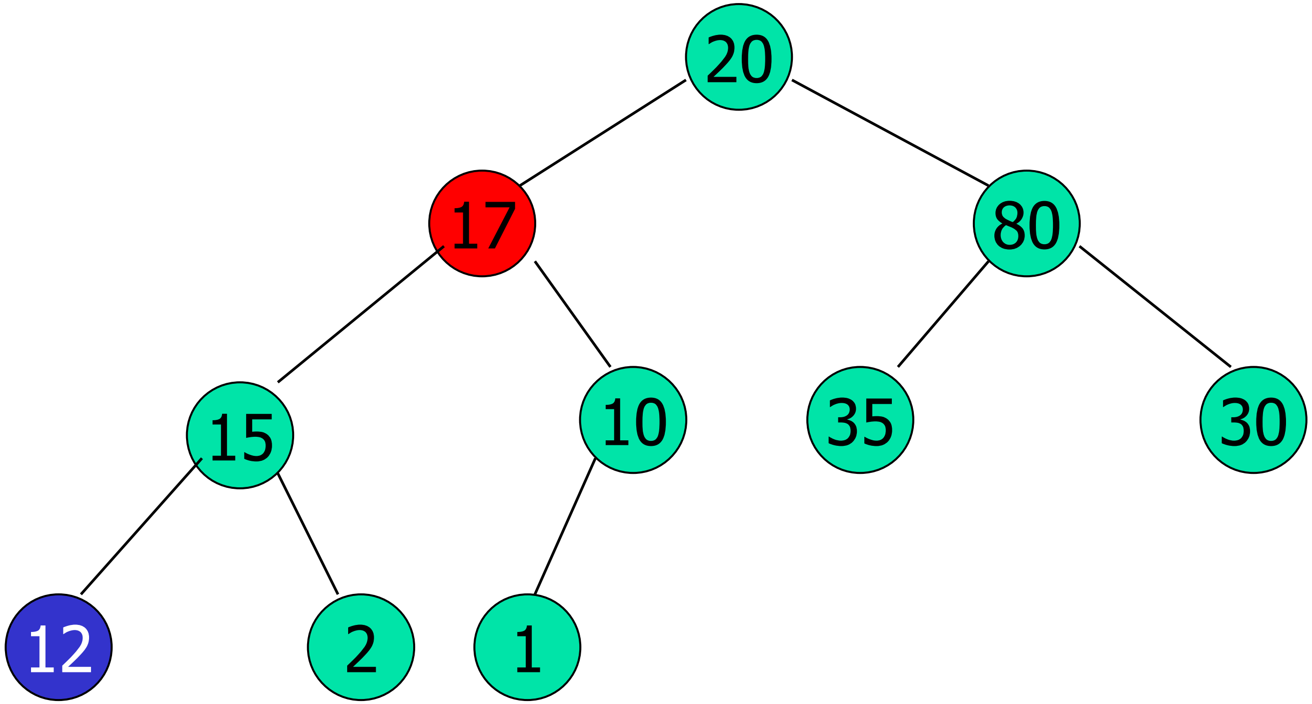
MaxHeap Initialization (8)

- Find a home for 12



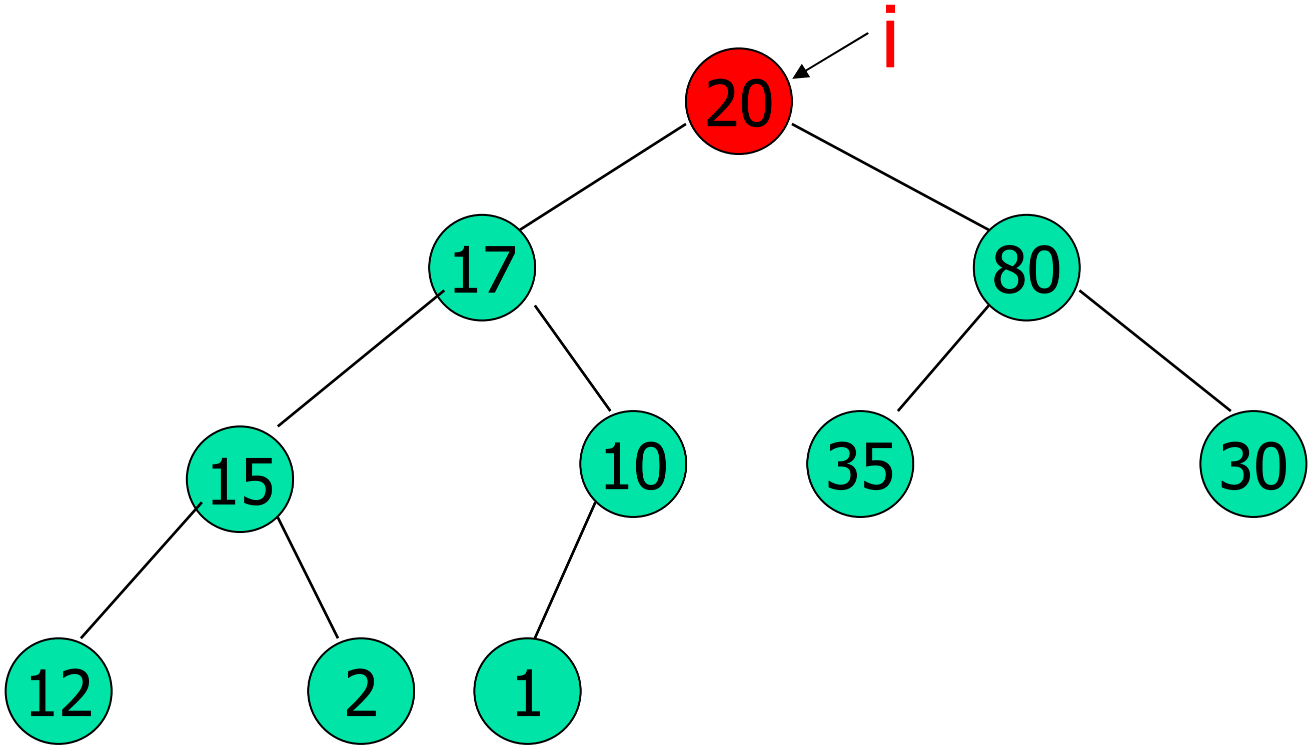
MaxHeap Initialization (9)

- Find a home for 12



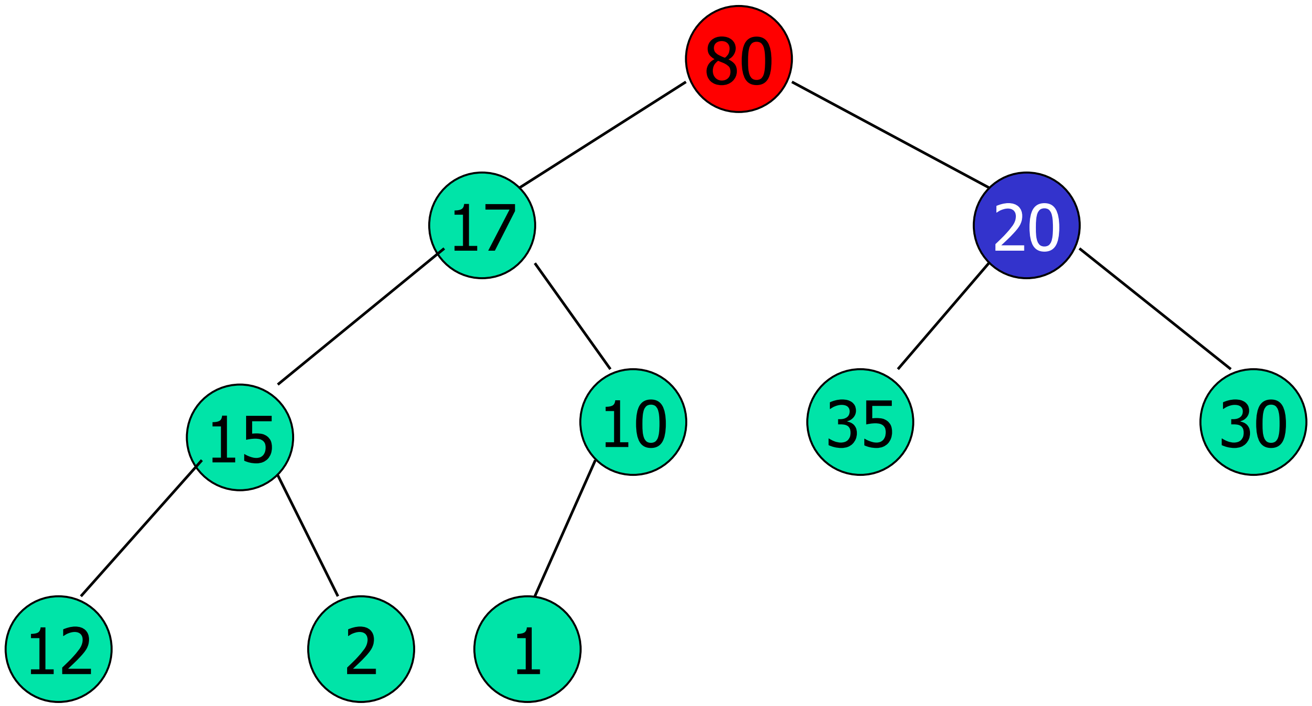
MaxHeap Initialization (10)

- Move to next lower array position.



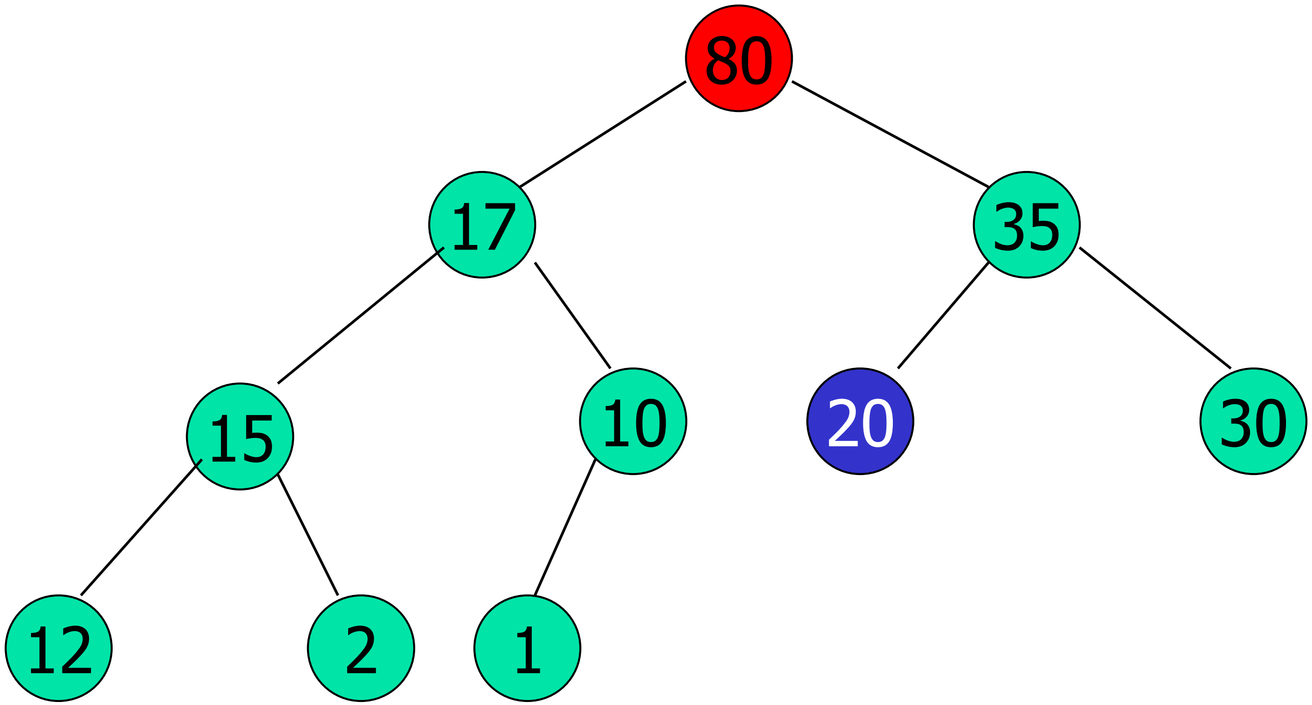
MaxHeap Initialization (11)

- Find a home for 20



MaxHeap Initialization (12)

- Result the max heap



initialize() in MaxHeap

```
public void initialize(Comparable [] theHeap, int theSize) {
heap = theHeap;
size = theSize;
for (int root = size / 2; root >= 1; root--) { // heapify
      Comparable rootElement = heap[root];
      // find place to put rootElement
      int child = 2 * root; // parent of child is target location for rootElement
     while (child <= size) { // heap[child] should be larger sibling
     if (child < size && heap[child].compareTo(heap[child + 1]) < 0) child++;
        // can we put rootElement in heap[child/2]?
     if (rootElement.compareTo(heap[child]) >= 0)  break;  // yes
     heap[ child / 2 ] = heap[ child ]; // no // move child up
     child *= 2;                        // move down a level
     }
      heap[ child / 2 ] = rootElement;
   }
}
```

Complexity of Heap Initialization

- Rough Analysis
  - for each element n/2, for-loop 0(log n) ➔ 0(n * log n)
- Careful Analysis
  - Height of heap = h
  - Height of each subtree at level j = $h' = h - j + 1$
  - Num of nodes at level j $\leq 2^{j-1}$
  - Time for each subtree at level $j = O(h') = O(h-j+1)$
  - Time for all nodes at level j $\leq 2^{j-1} * (h-j+1) = t(j)$
  - Total time for all level is $t(1) + t(2) + \ldots + t(h-1) = O(n)$

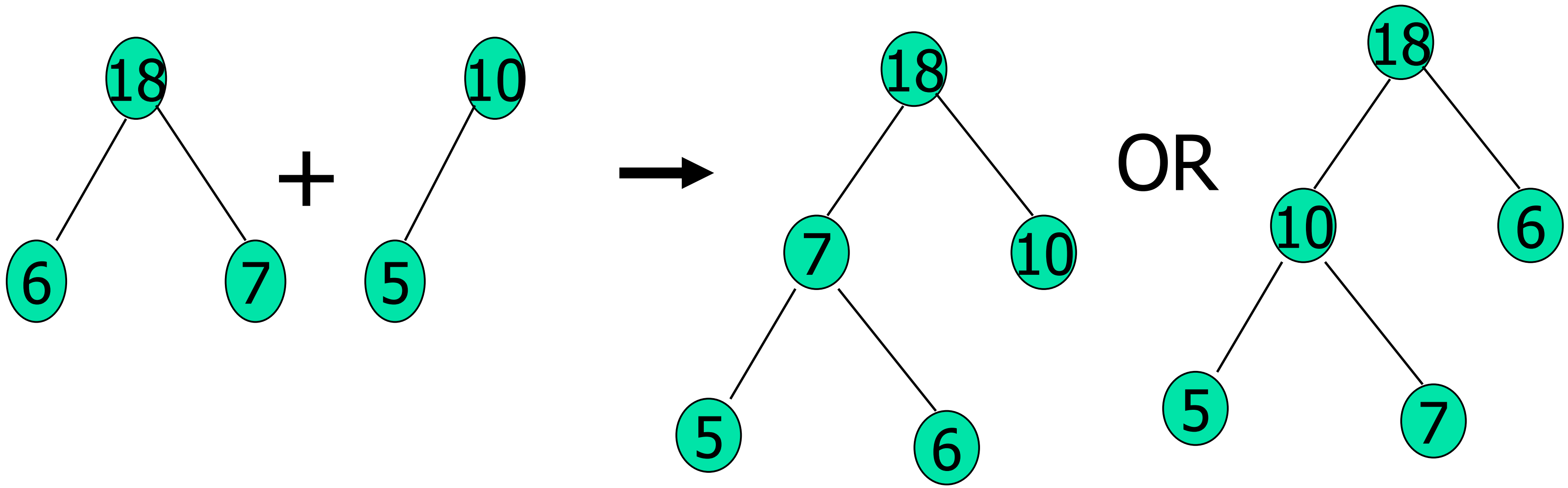
- No more than n swappings!

Table of Contents

Jump To HeapSort

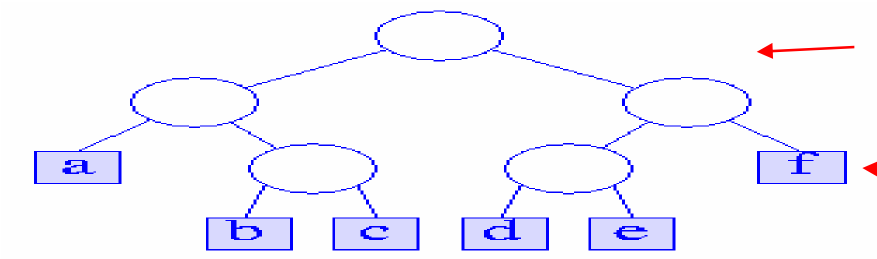
Merging Two Priority Queues

- Heap is efficient for priority queue
- Some applications require merging two or more priority queues
- Heap is not suitable for merging two or more priority queues
- Leftiest tree is powerful in merging two or more priority queues



Height-Biased Leftist Tree (HBLT)

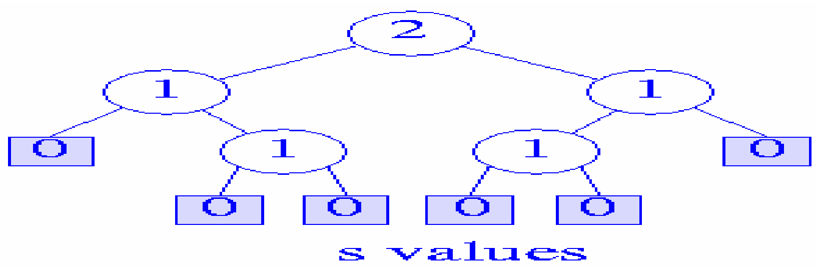
- Extended Binary Tree: Add an external node replaces each empty subtree.



Internal node

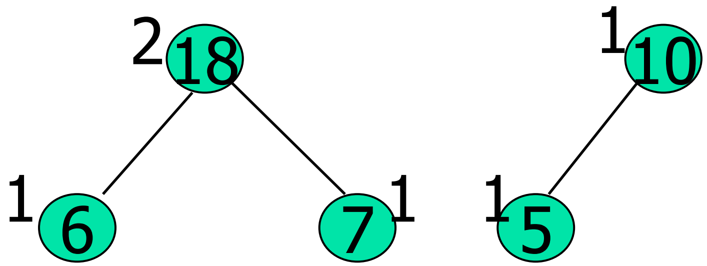
External node

- Let s(x) be the length of a shortest path from node x to an external node in its subtree.



Height-Biased Leftist Tree (HBLT)

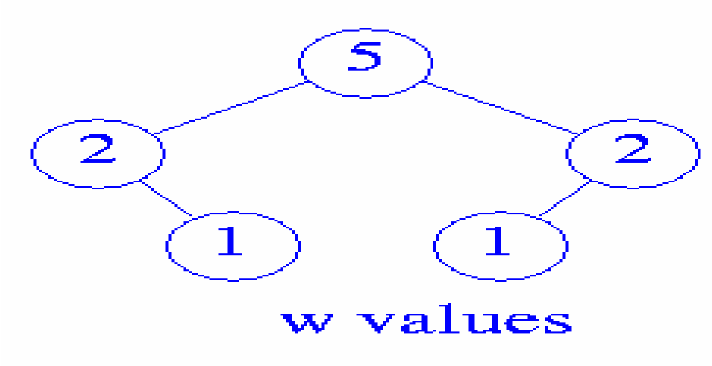
- A binary tree is a height-biased leftist tree (HBLT)
  iff at every internal node, the s value of the left child is greater than or equal to the s value of the right child.
  - A max HBLT is an HBLT that is also a max tree.
  - A min HBLT is an HBLT that is also a min tree.



- S values in HBLT contributes to make complete binary tree!!!!!!
- [Theorem] Let x be any internal node of an HBLT
  - The number of nodes in the subtree with root x is at least $2^{s(x)} - 1$
  - If the subtree with root x has m nodes, s(x) is at most $\log_2(m+1)$
  - The length of the right-most path from x to an external node is s(x)

Weight Biased Leftist Tree (WBLT)

- Let w(x) be the weight from node x to be the number of internal nodes in the subtree with root x



- A binary tree is weight-biased leftist tree (WBLT)

  iff at every internal node the w value of the left child is greater than or equal to the w value of the right child

  - A max WBLT is a max tree that is also a WBLT
  - A min WBLT is a min tree that is also a WBLT

Put a Max Element into a HBLT

- Create a new max HBLT
- Meld this max HBLT and the original

```
public void put (Comparable theElement) {
    HbltNode q = new HbltNode (theElement, 1);
    // meld q and original tree
    root = meld (root, q);
    size++;
}
```

Remove a Max element from a HBLT

- Delete the root
- Meld its two subtrees

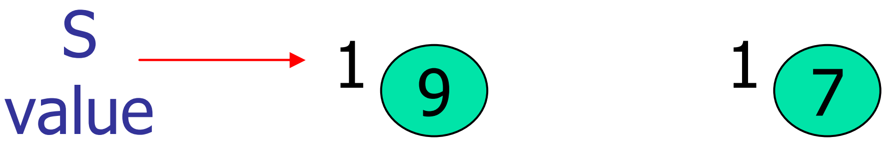
```
public Comparable removeMax() {
    if (size == 0) return null;   // tree is empty

    // tree not empty
    Comparable x = root.element;  // save max element
    root = meld (root.leftChild, root.rightChild);
    size--;
    return x;
}
```

Meld Two HBLTs

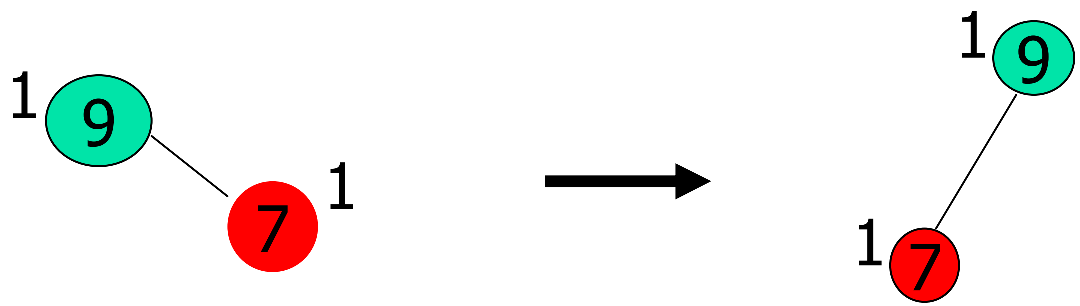
- Let A & B be the two HBLTs
- Compare the root of A & B
- The bigger value is the new root for the melded tree
  - Assume the root of A is bigger & A has left subtree L
- Meld the right subtree and B ➔ result C
- A has the left subtree L and the right subtree C
- Compare the S values of L & C
- Swap if necessary

Melding 2 HBLTs: Ex 1

- Consider the two max HBLTs

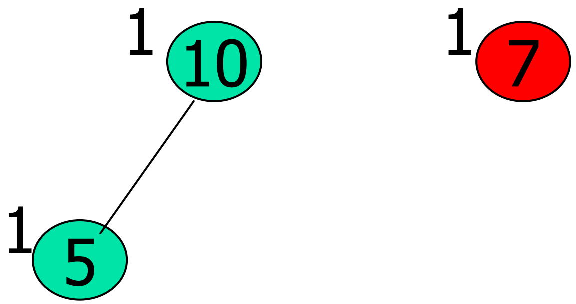


- 9 > 7, so 9 is root.
- The s value of the left subtree of 9 is 0 while the s value of the right subtree is 1 ➔ Swap the left subtree and the right subtree

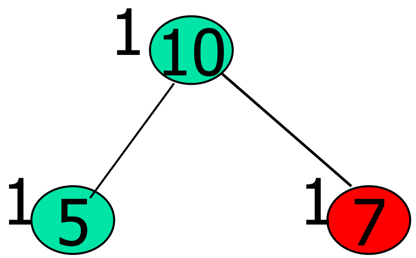


Melding 2 HBLTs: Ex 2

- Consider the two max HBLTs



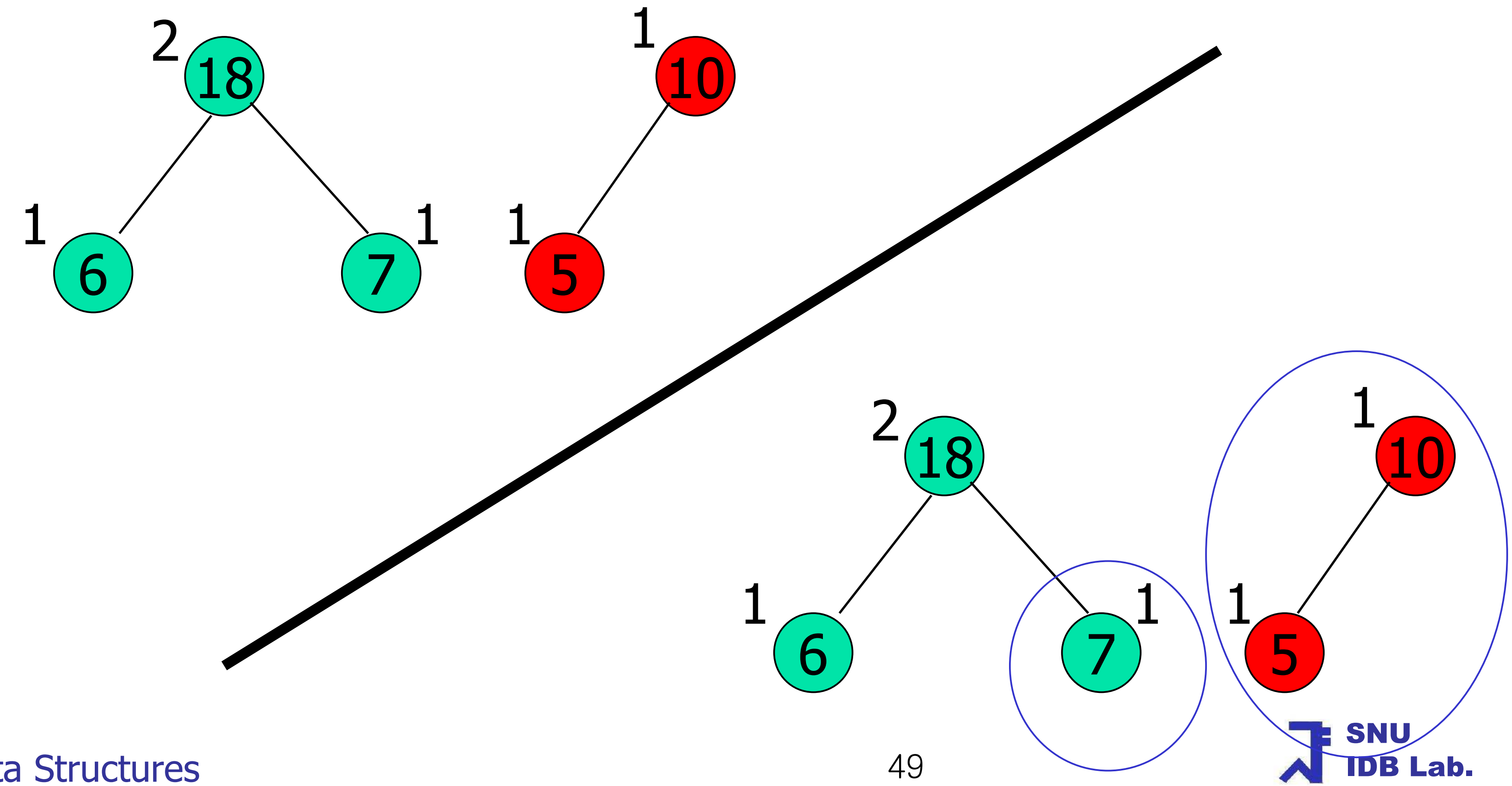
- 10 > 7, so root is 10



- Comparing the s values of the left and right children of 10, a swap is not necessary

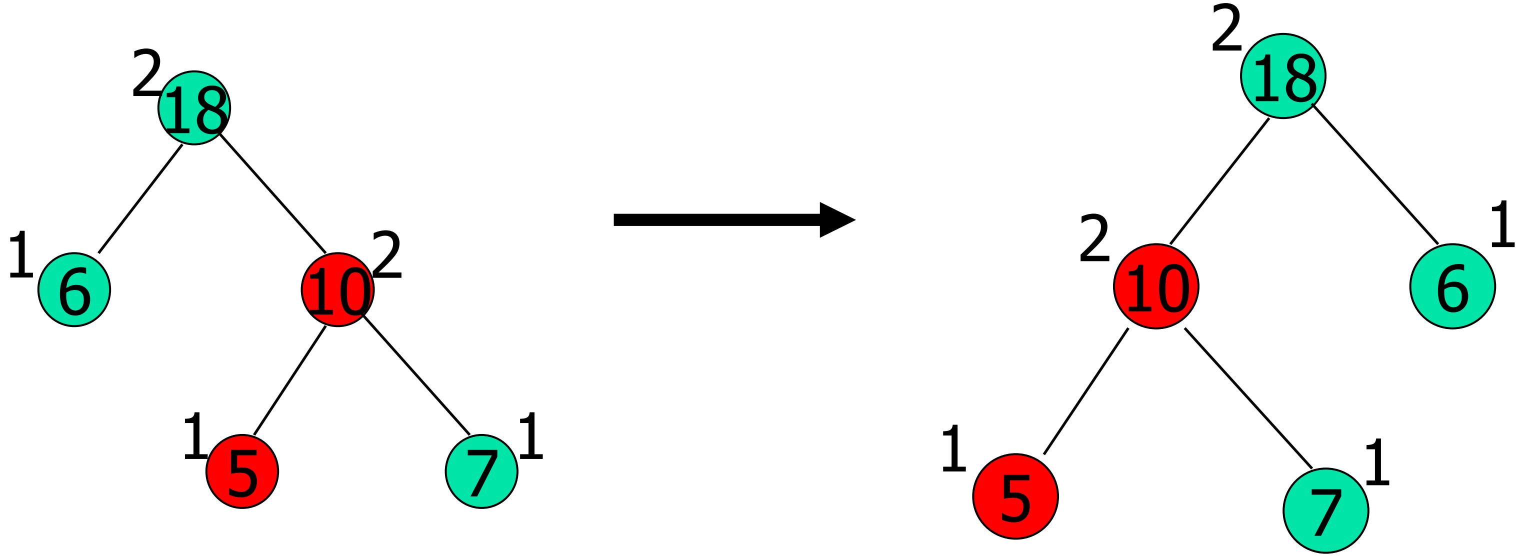
Melding 2 HBLTs: Ex 3 (1)

- Consider the two max HBLTs



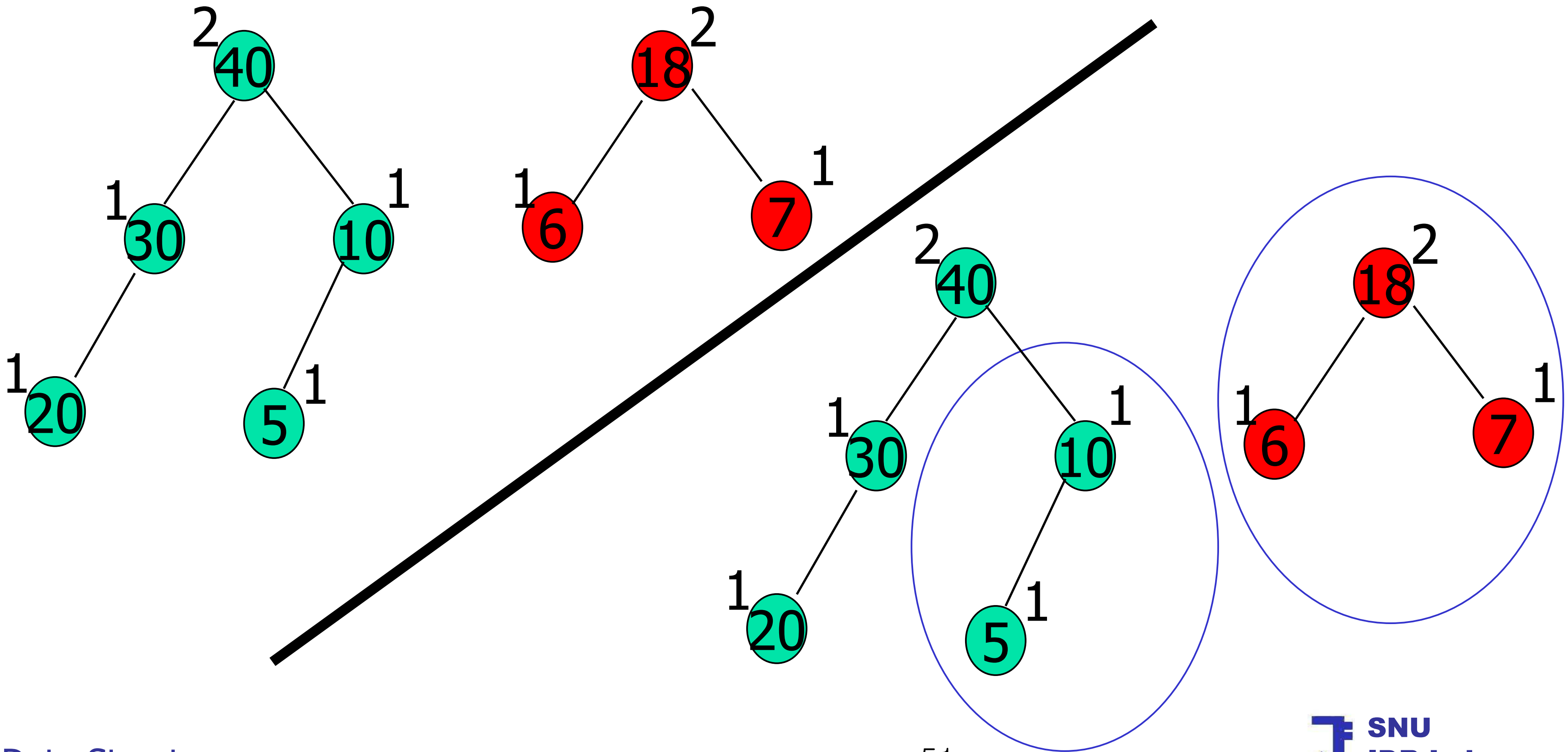
Melding 2 HBLTs: Ex 3 (2)

- 18 > 10, root is 18
- Meld the right subtree of 18
- s(left) < s(right), swap left and right subtree



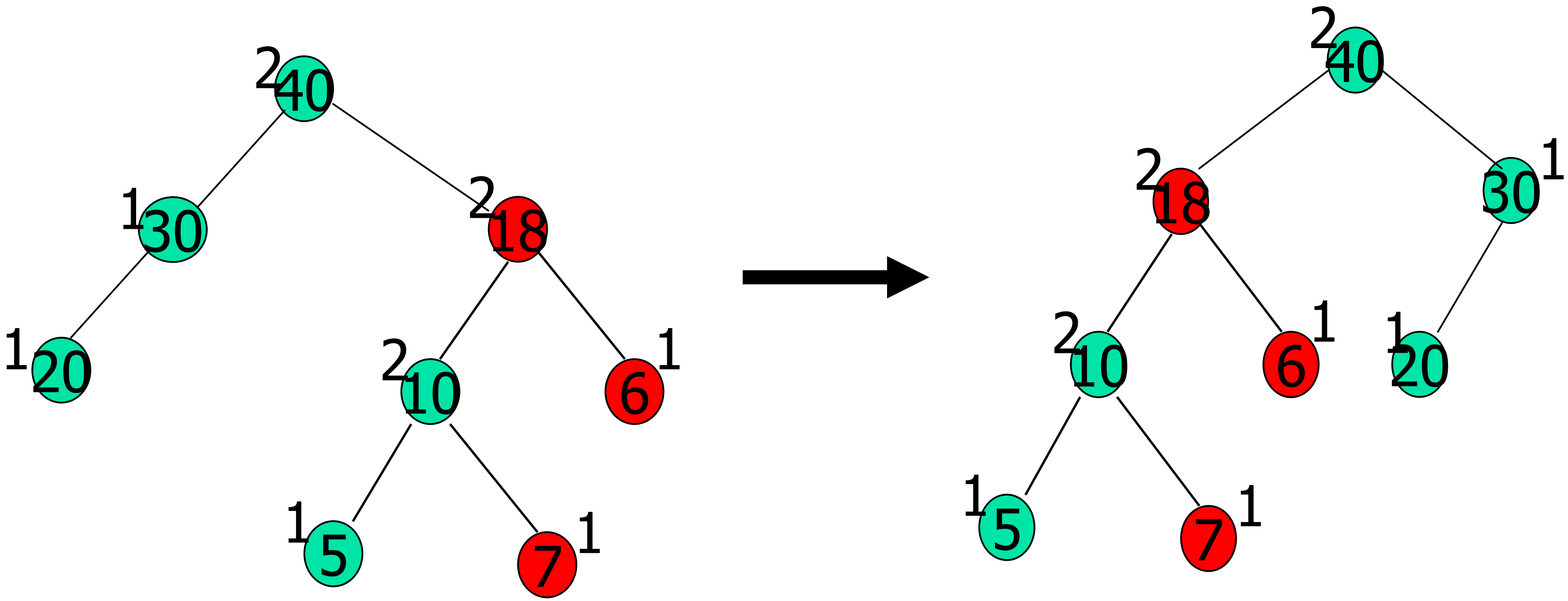
Melding 2 HBLTs: Ex 4 (1)

- Consider the two max HBLTs



Melding 2 HBLTs: Ex 4 (2)

- 40 > 18, root is 40
- Meld the right subtree of 40
- s(left) < s(right), swap left and right subtree



meld() in HBLT

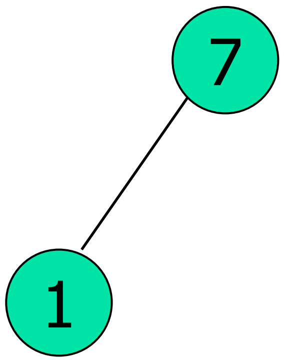
```
private static HbltNode meld (HbltNode x, HbltNode y) {
    if (y == null)    return x;   // y is empty
    if (x == null)    return y;   // x is empty
    // neither is empty, swap x and y if necessary
    if (x.element.compareTo(y.element) < 0)  { // swap x and y
            HbltNode t = x;    x = y;  y = t; } // now x.element >= y.element
    x.rightChild = meld (x.rightChild, y);
    if (x.leftChild == null) { // left subtree is empty, swap the subtrees
            x.leftChild = x.rightChild;  x.rightChild = null; x.s = 1; }
    else { // swap only if left subtree has a smaller s value
                if (x.leftChild.s < x.rightChild.s) { // swap subtrees
                  HbltNode t = x.leftChild; x.leftChild = x.rightChild;  x.rightChild = t; }
                x.s = x.rightChild.s + 1; // update s value
     }
    return x;
}
```

Initializing a Max HBLT (1)

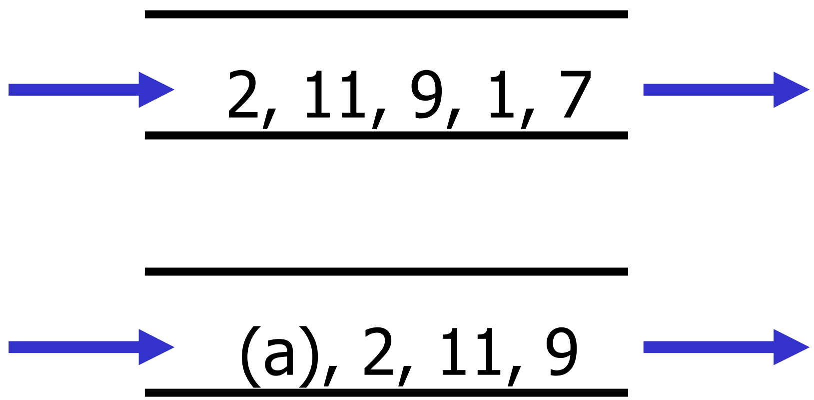
- Create a max HBLT with the five elements 7, 1, 9, 11, and 2
- Five single-element max HBLTs are created and placed in a FIFO queue

- The max HBLTs 7 and 1 are deleted from the queue and melded into (a)

7 & 1 →



(a)

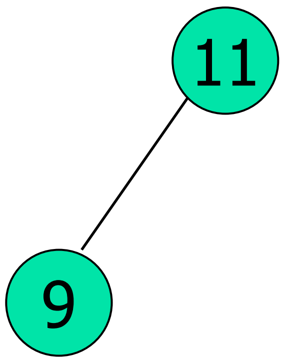


- The result (a) is added to the queue

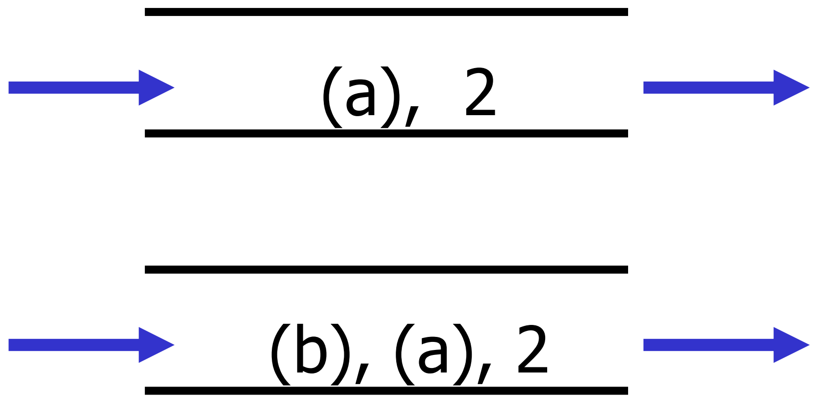
Initializing a Max HBLT (2)

- The max HBLTs 9 and 11 are deleted from the queue and melded into (b)

9 & 11 ➔



(b)

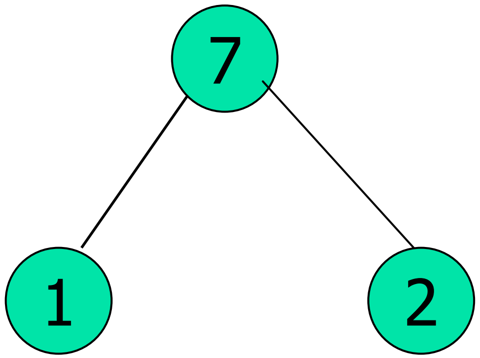


- The result (b) is added to the queue

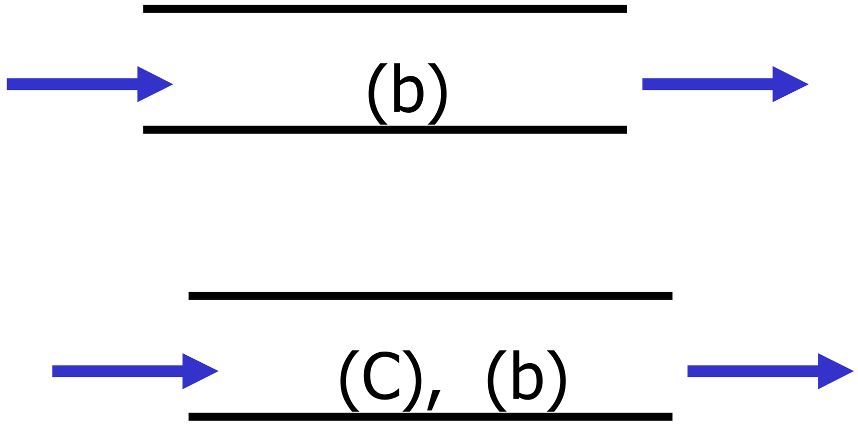
Initializing a Max HBLT (3)

- The max HBLTs 2 and (a) are deleted from the queue and melded into (c)

2 & (a) ➔



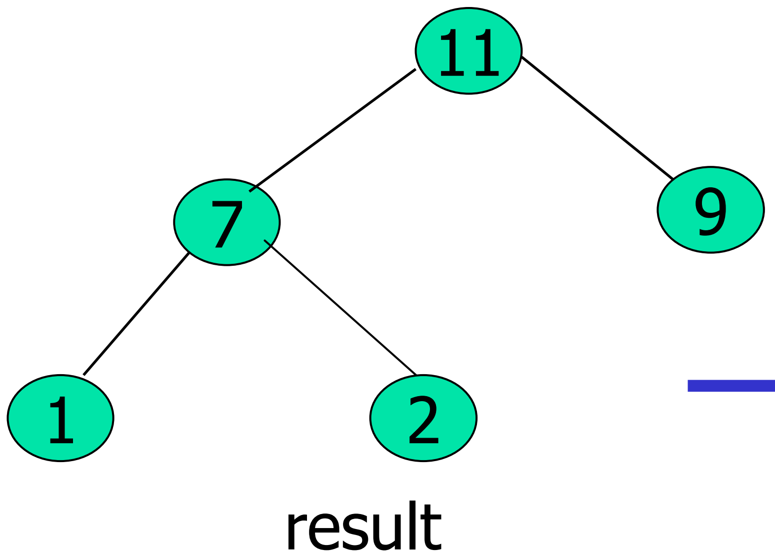
(c)



- The result (c) is added to the queue

Initializing a Max HBLT (4)

- The max HBLTs (b) and (c) are deleted from the queue and melded into the result



- The result is added to the queue
- The queue now has just one max HBLT, and we are done with the initialization

initialize() in HBLT

```
public void initialize(Comparable [] theElements, int theSize) {
    size = theSize;
    ArrayQueue q = new ArrayQueue(size);
    // initialize queue of trees
    for (int i = 1; i <= size; i++) // create trees with one node each
        q.put(new HbltNode(theElements[i], 1));
   // repeatedly meld from queue q
    for (int i = 1; i <= size - 1; i++) { // remove and meld two trees from the queue
            HbltNode b = (HbltNode) q.remove();
            HbltNode c = (HbltNode) q.remove();
            b = meld(b, c);
            // put melded tree on queue
            q.put(b);
    }
    if (size > 0)  root = (HbltNode) q.remove();
}
```

Complexity Analysis of HBLT

- getMax
  - $\Theta(1)$
- The complexity of put() and removeMax() is the same as that of = meld()
  - put() and removeMax() are used for meld()
- meld()
  - Root x and y
  - O(s(x) + s(y)) where s(x) and s(y) are at most log(m+1) and log(n+1)
    - m and n are the number of elements in the max HBLTs with root x and y
  - O(log(m) + log(n)) = O(log(m*n))

Complexity of Initialize HBLT

- n = size of a power of 2
- The first n/2 melds involve max HBLTs with one element each
- The next n/4 melds involve max HBLTs with two elements each
- The next n/8 melds involve max HBLTs with four elements each
- And so on

- Meld two trees with 2^i elements each
  - O(i+1)
- Total time
  - O(n/2 + 2*(n/4) + 3*(n/8) + ···) = O(n)

Table of Contents

- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

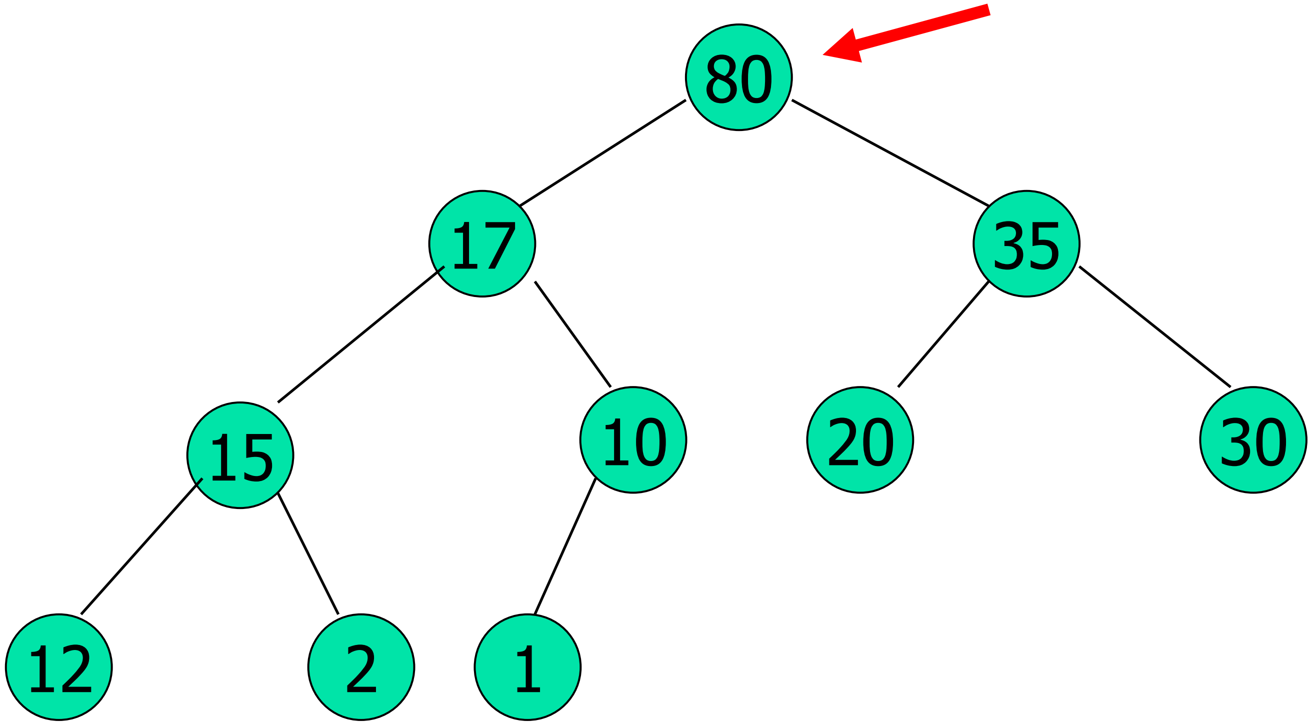
Heap Sort

```
public static void heapSort (Comparable [] a)     {

    //create a max heap of the elements
    MaxHeap h = new MaxHeap();
    h.initialize(a, a.length – 1);

    //extract one by one from the max heap
    for (int i = a.length – 2; i >= 1; i--)
            a[i + 1] = h.removeMax() ;
}
```

MaxHeap class: initialize(), removeMax()

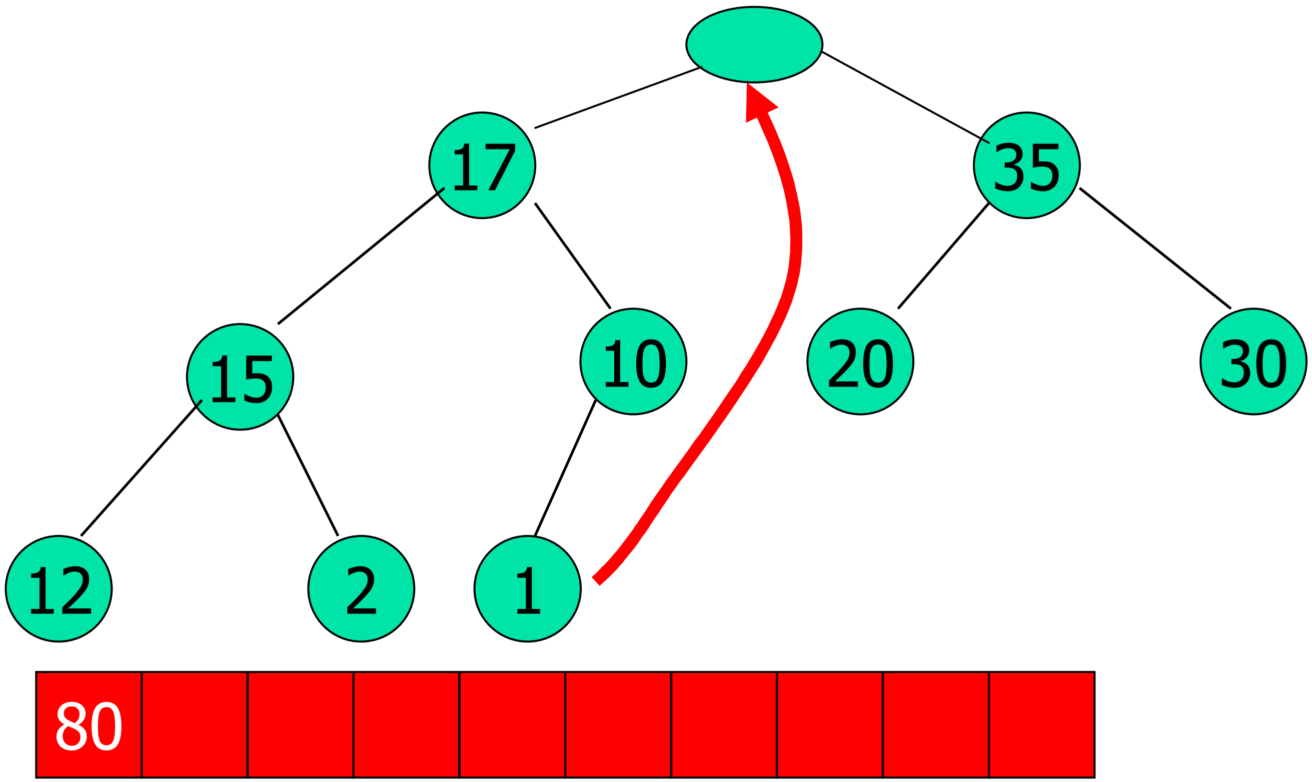
Heap Sort Ex (1)

- This sorting loop begins with the max heap



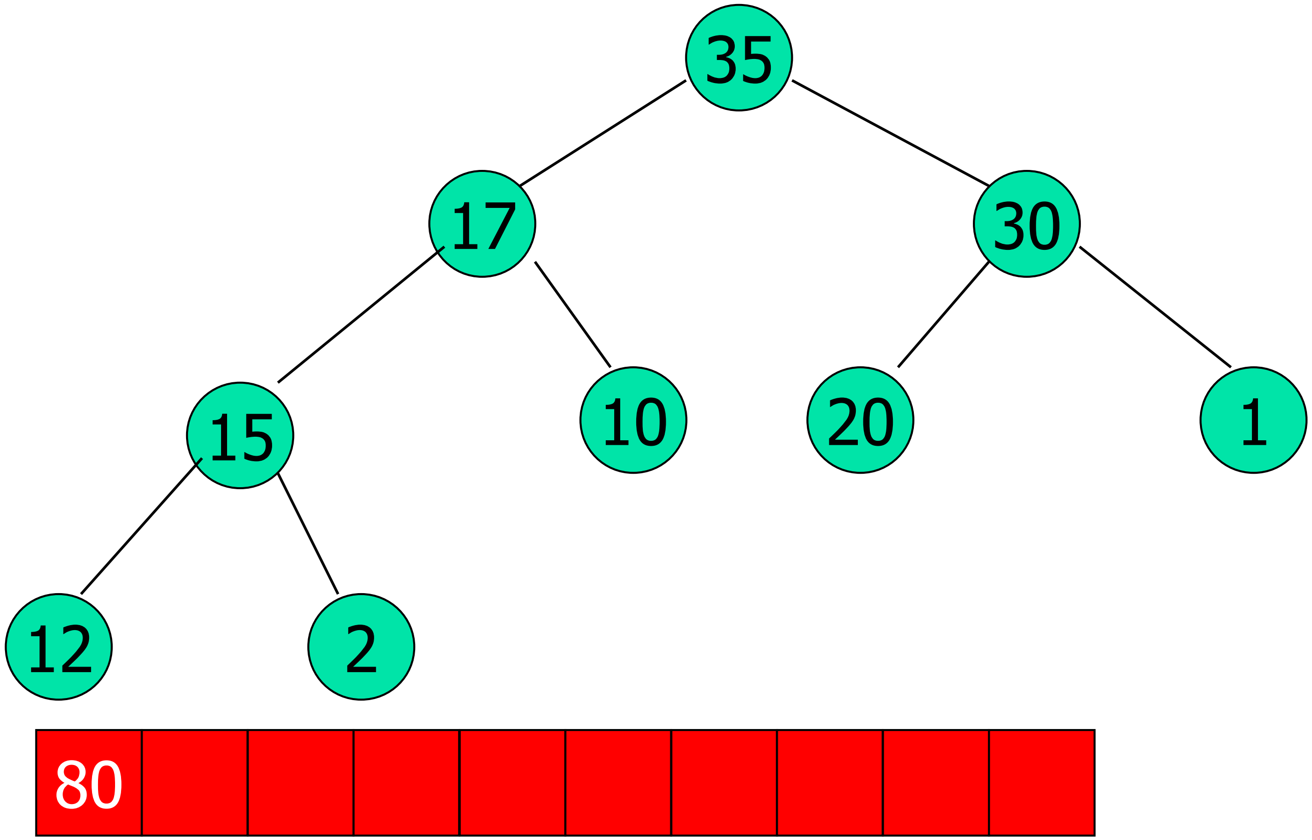
Heap Sort Ex (2)

- Remove Max & move the last element "1" to the root



Heap Sort Ex (3)

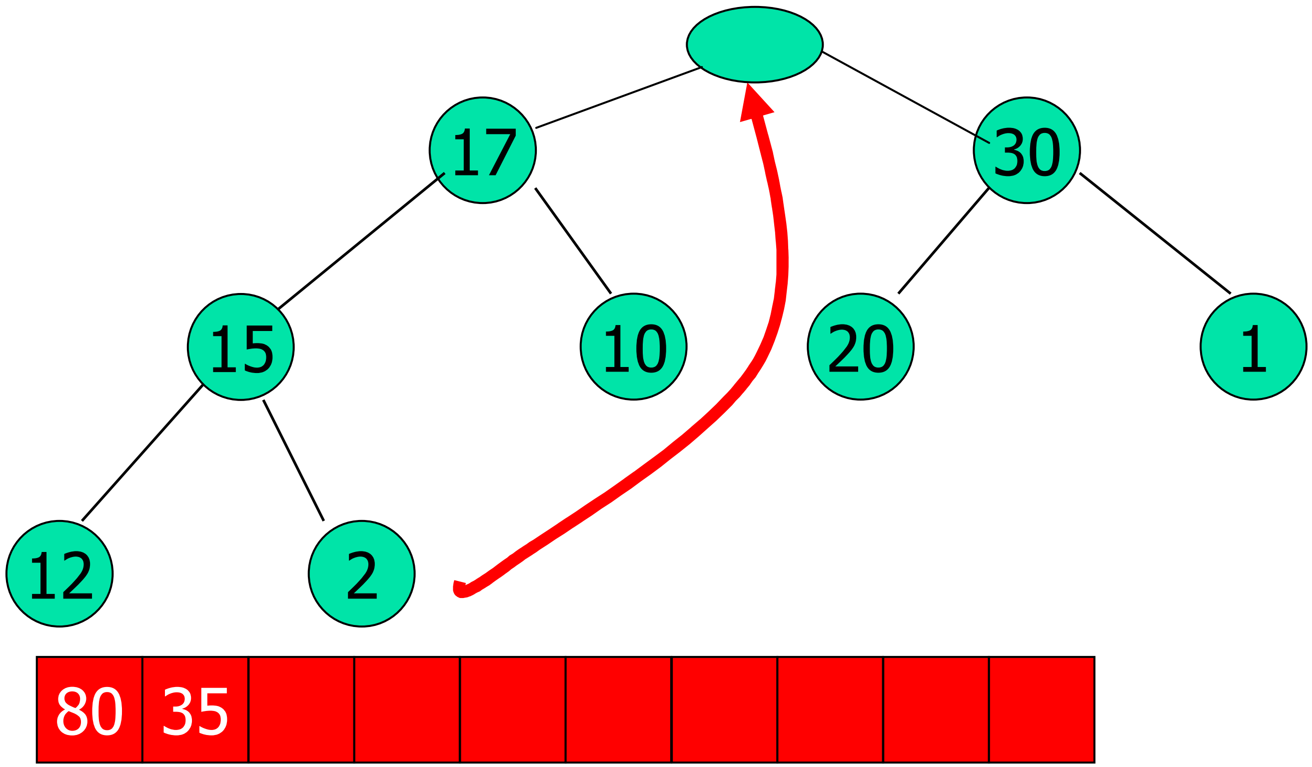
- Reheapify: Meld root.leftChild and root.rightChild



SNU IDB Lab.

Heap Sort Ex (4)

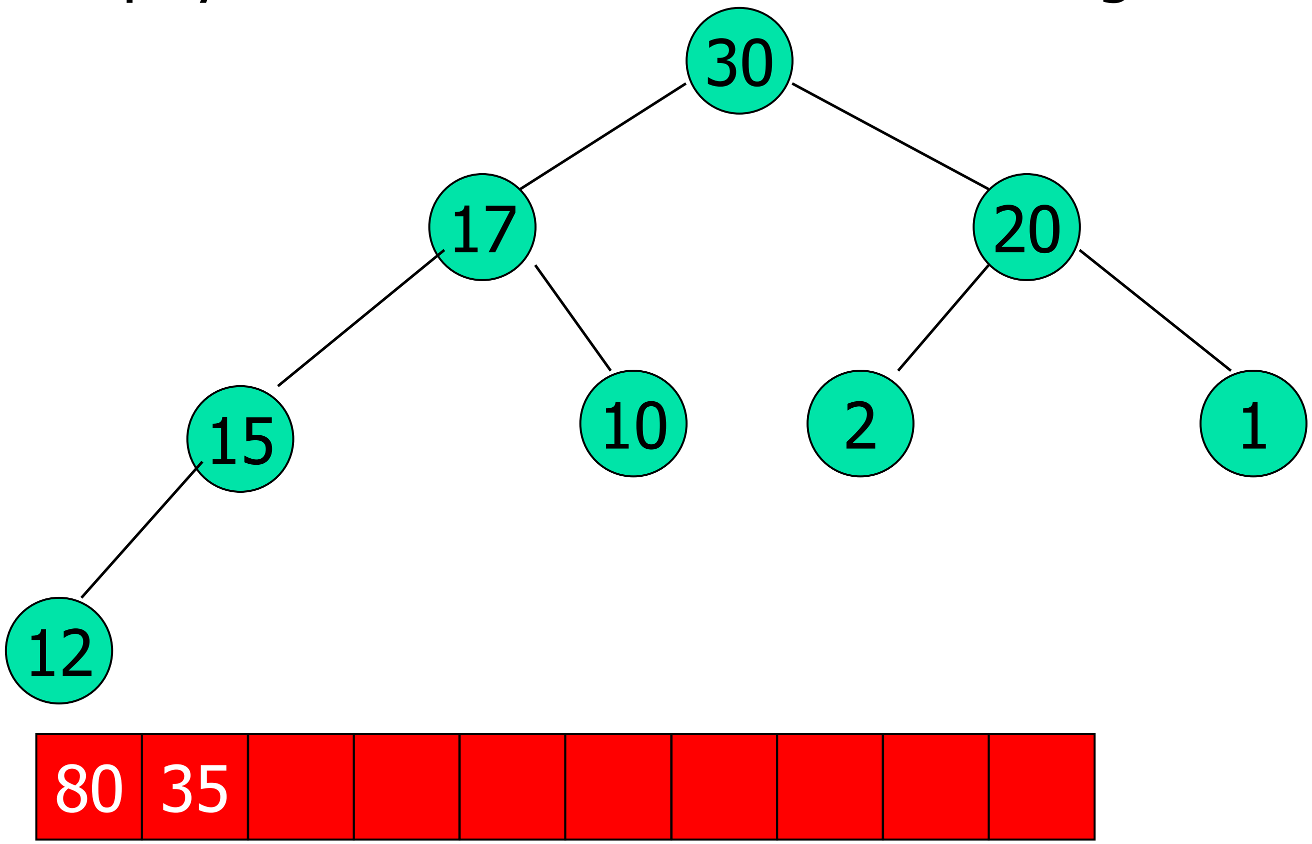
- Remove Max & move the last element "2" to the root



SNU
IDB Lab.

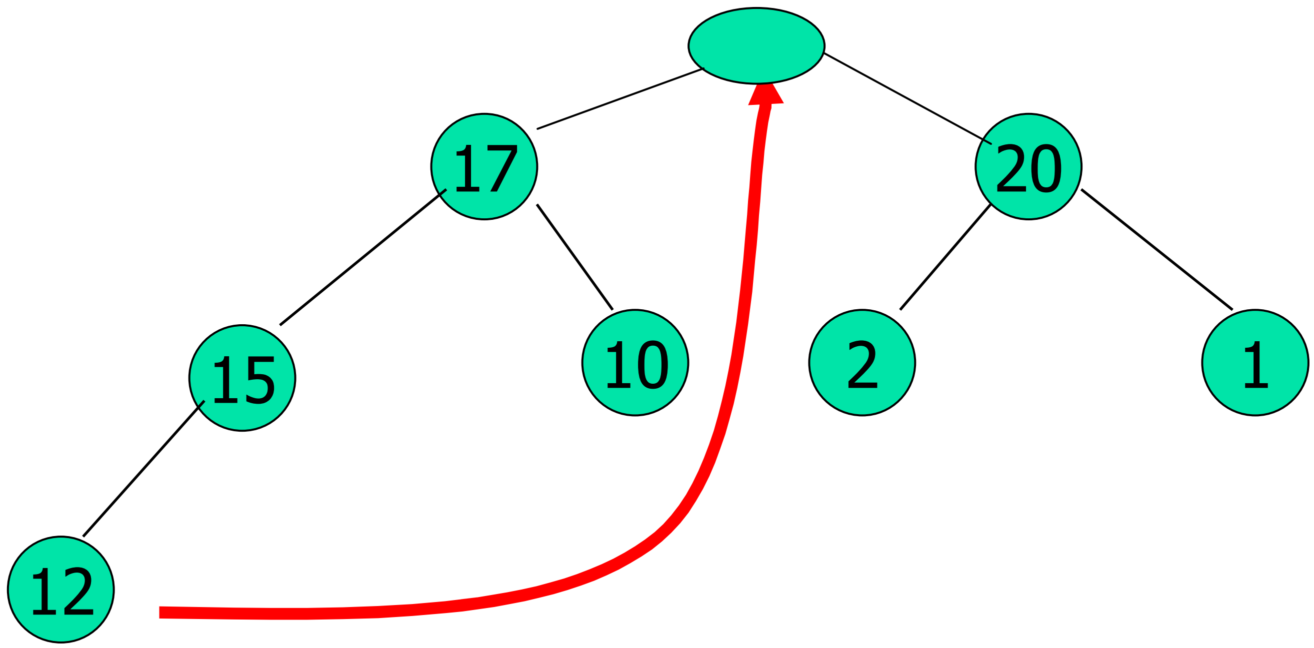
Heap Sort Ex (5)

- Reheapify: Meld root.leftChild and root.rightChild



Heap Sort (6)

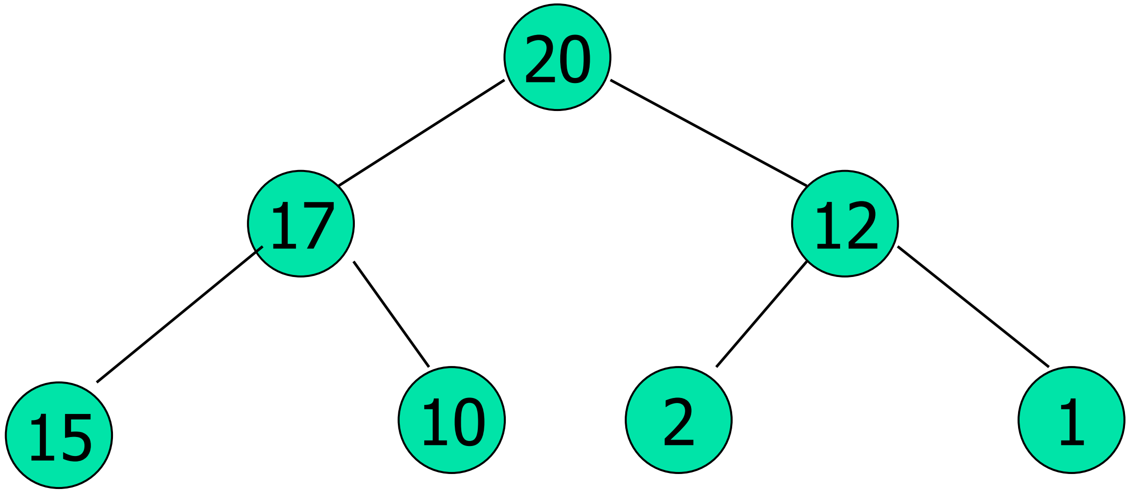
- Remove Max & move the last element “12” to the root



80	35	30							

Heap Sort Ex (7)

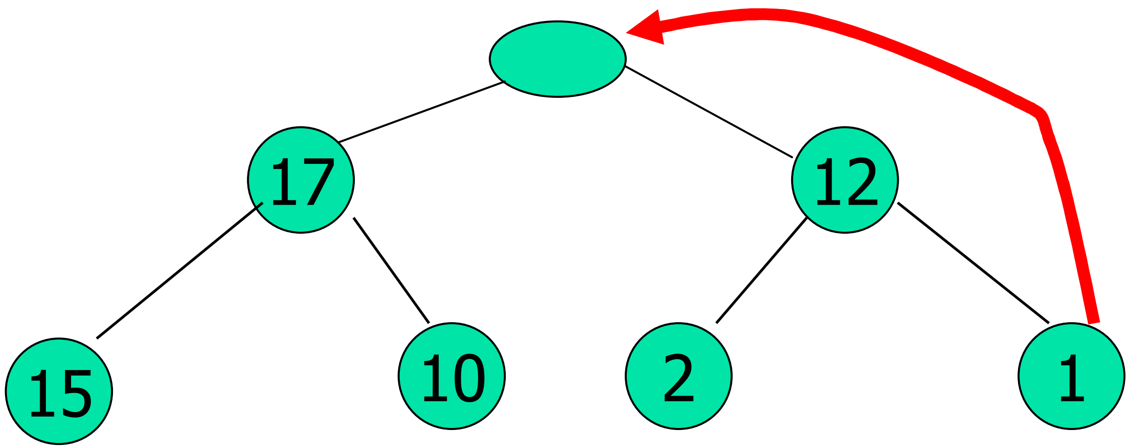
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30							

Heap Sort Ex (8)

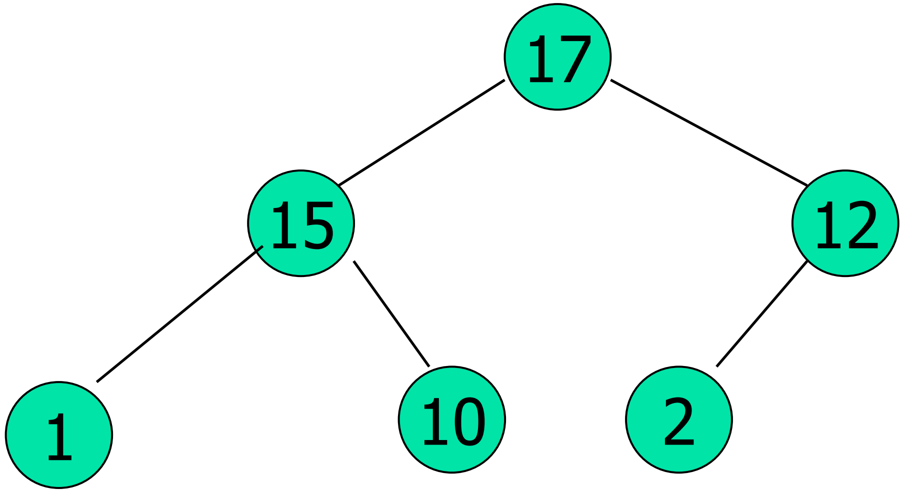
- Remove Max & move the last element "1" to the root



80	35	30	20						

Heap Sort Ex (9)

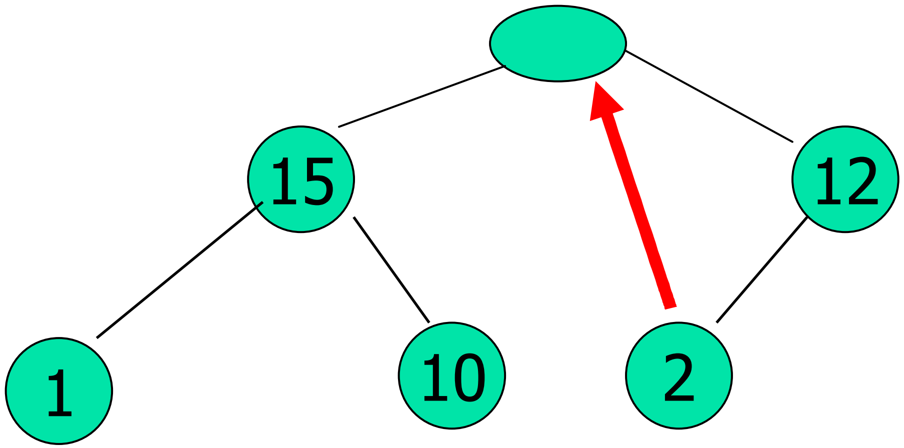
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20						

Heap Sort Ex (10)

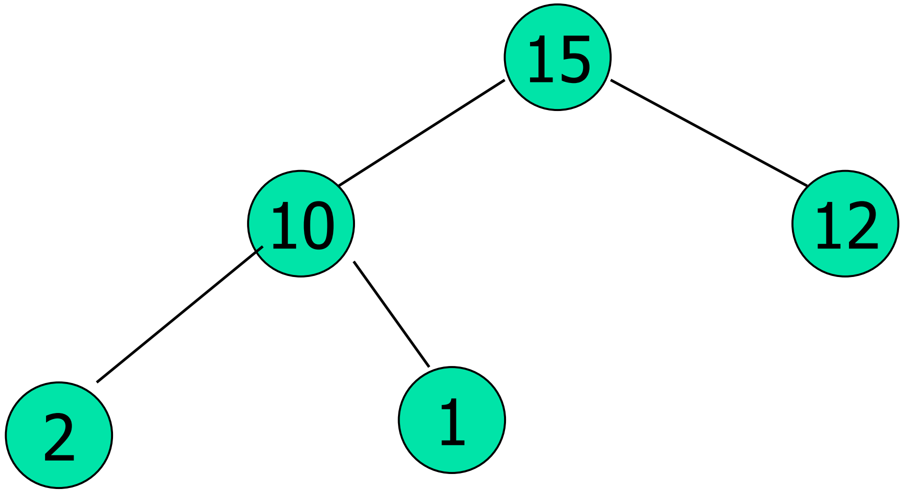
- Remove Max & move the last element "2" to the root



80	35	30	20	17					

Heap Sort Ex (11)

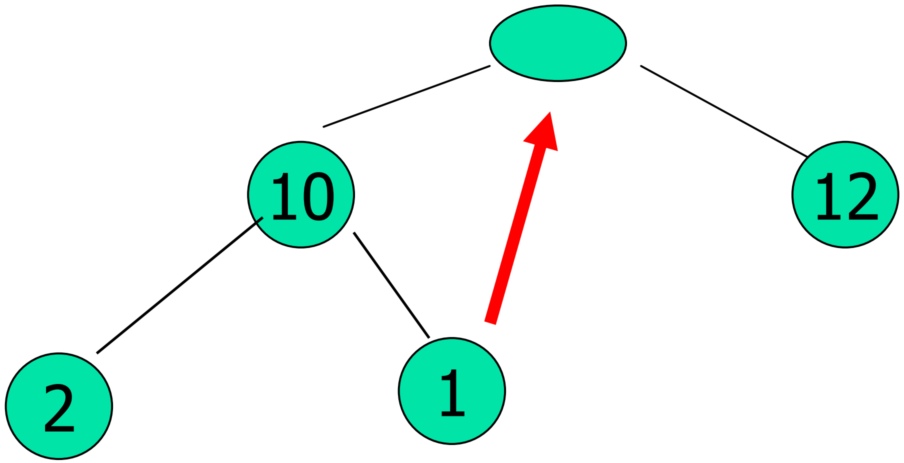
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20	17					

Heap Sort Ex (12)

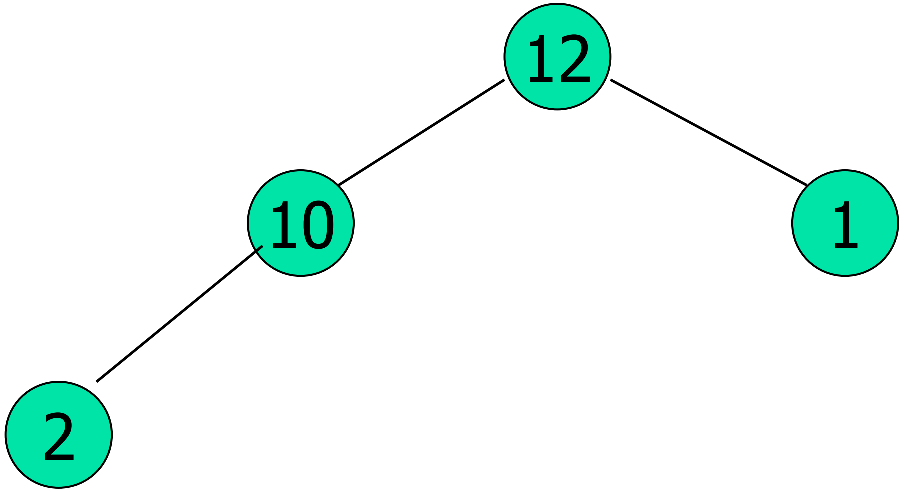
- Remove Max & move the last element "1" to the root



80	35	30	20	17	15				

Heap Sort Ex (13)

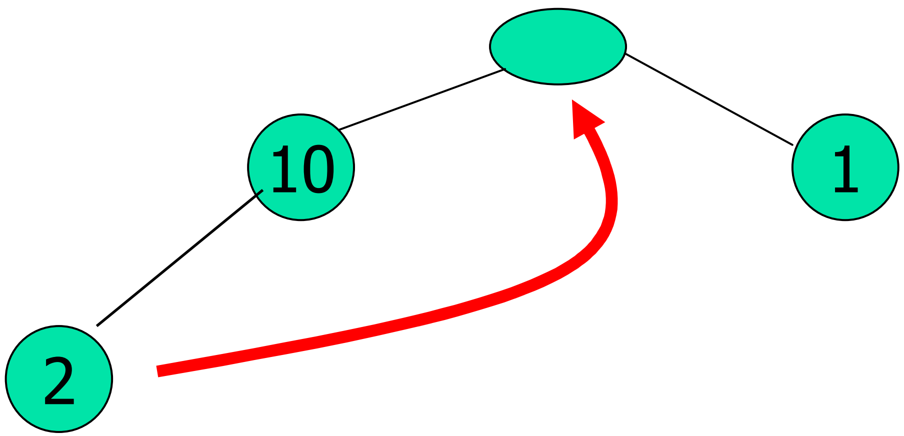
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20	17	15				

Heap Sort Ex (14)

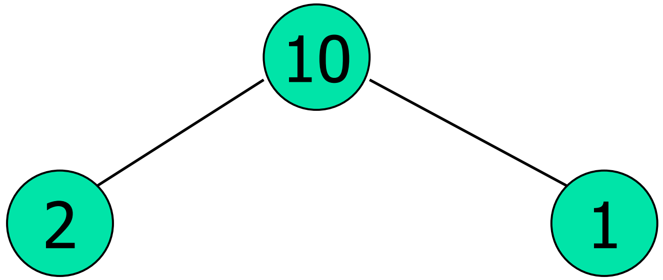
- Remove Max & move the last element “2” to the root



80	35	30	20	17	15	12			

Heap Sort Ex (15)

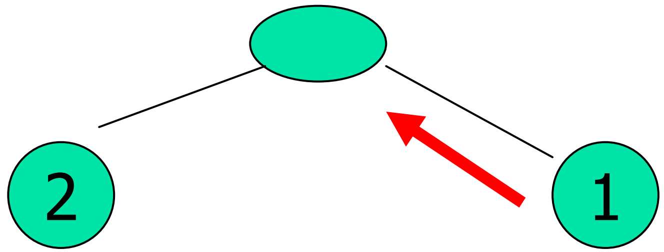
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20	17	15	12			

Heap Sort Ex (16)

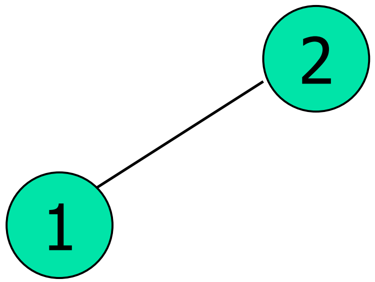
- Remove Max & move the last element “1” to the root



80	35	30	20	17	15	12	10		

Heap Sort Ex (17)

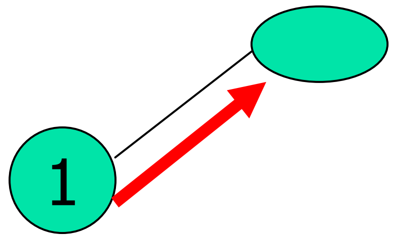
- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20	17	15	12	10		

Heap Sort Ex (18)

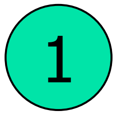
- Remove Max & move the last element "1" to the root



80	35	30	20	17	15	12	10	2	

Heap Sort Ex (19)

- Reheapify: Meld root.leftChild and root.rightChild



80	35	30	20	17	15	12	10	2	

Heap Sort Ex (20)

- Remove Max & we are done!

80	35	30	20	17	15	12	10	2	1

- Complexity of Heap Sort : O(n * logn)
  - Initialization : O(n)
  - Deletion : O(logn)
  - Sort → deletion n times → o(n * logn)

Table of Contents

- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

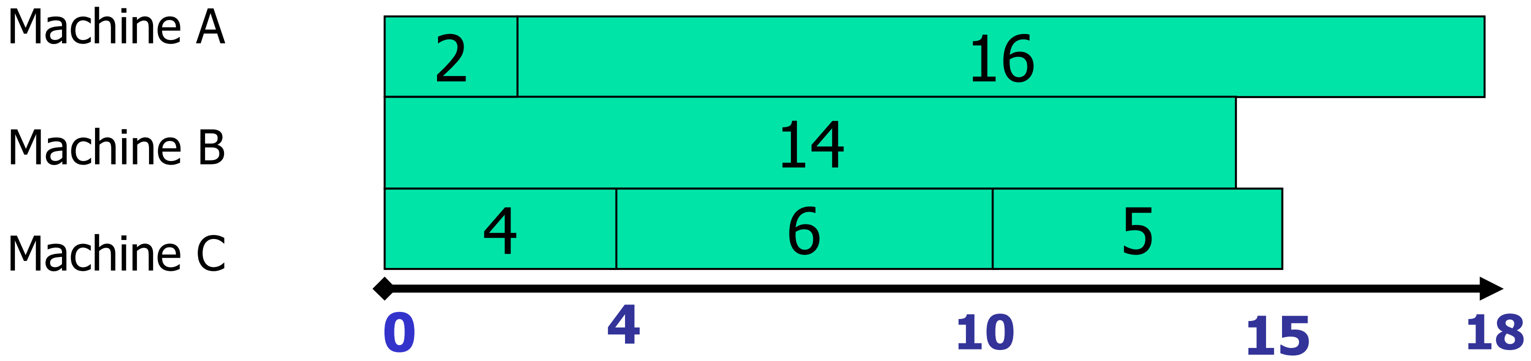
Machine Scheduling

- A schedule is an assignment of jobs to time intervals on machines
  - No machine processes more than one job at any time.
  - No job is processed by more than one machine at any time.
  - Each job i is assigned for a total of t_i units of processing.

- The finish time or length is
  - The time at which all jobs have completed
  - Strat time : s_i
  - Completion time : $s_i + t_i$

Three-machine schedule

- Seven jobs with processing requirements (2, 14, 4, 16, 6, 5, 3)

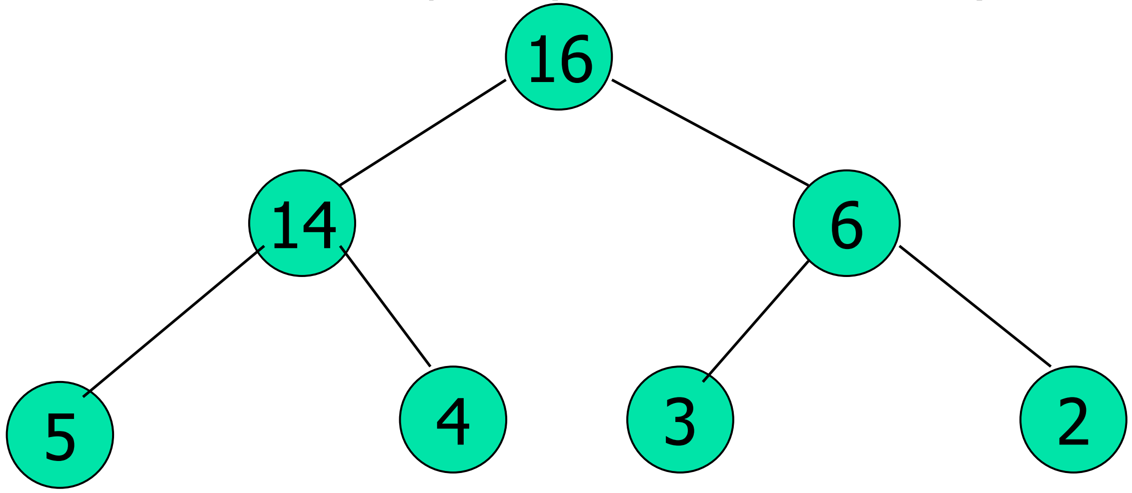


- Finish time : 18
- Objective: Find schedules with minimum finish time

LPT Schedule (1)

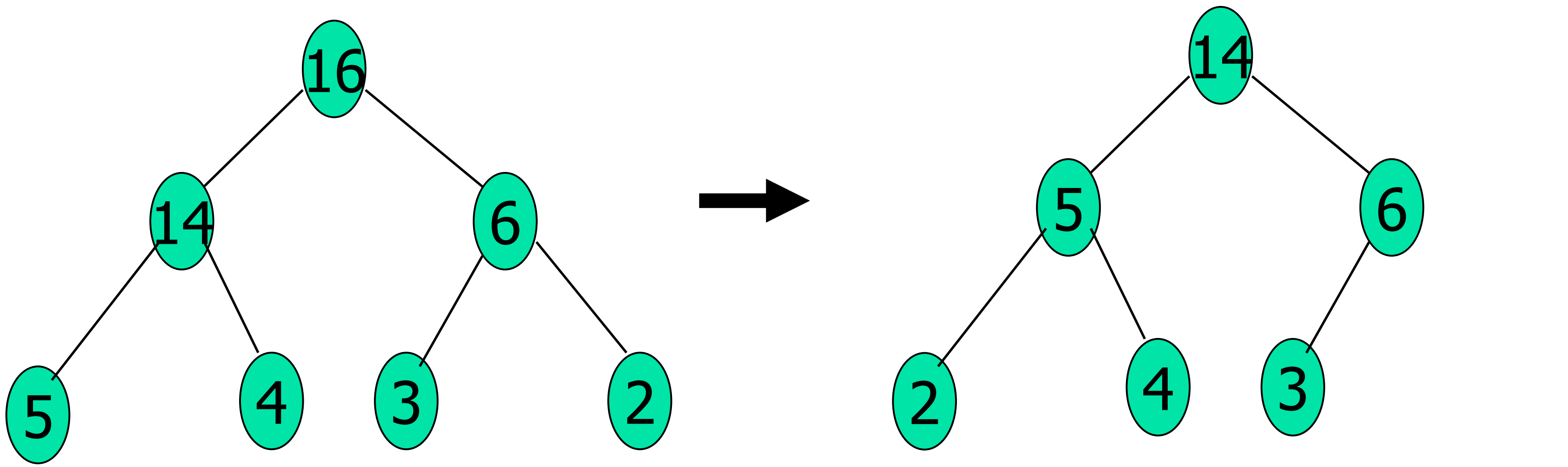
- Longest Processing Time first.
  - Jobs are scheduled in the order: 16, 14, 6, 5, 4, 3, 2
  - Each job is scheduled on the machine on which it finishes earliest.

- Use MaxHeap for LPT Schedule
  - Construct a MaxHeap for (2, 14, 4, 16, 6, 5, 3)



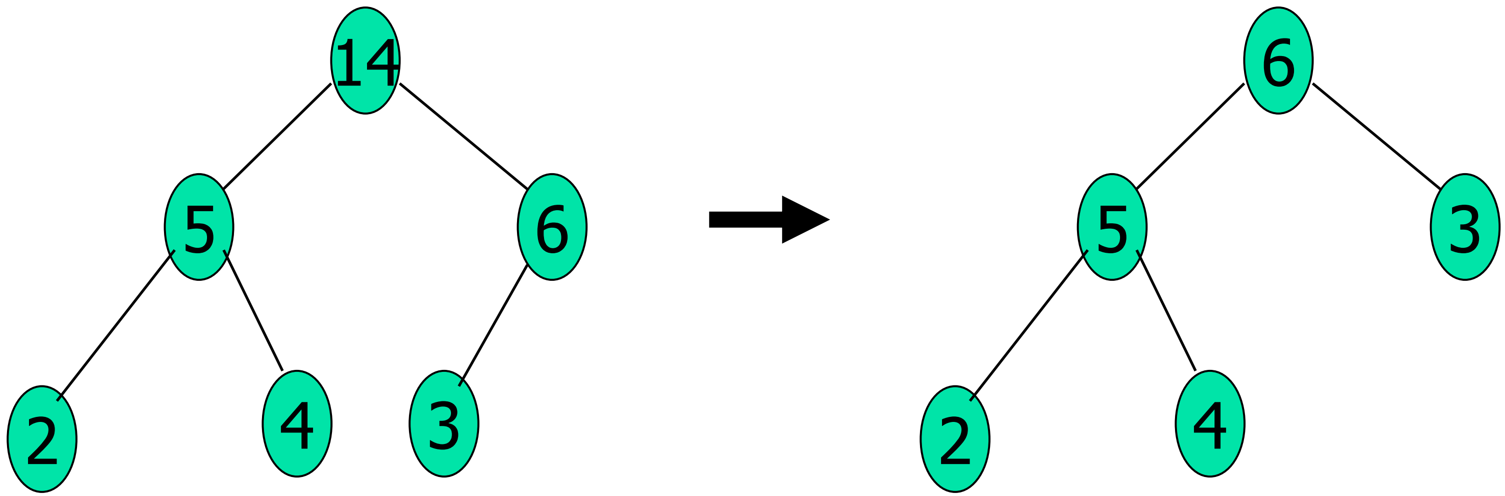
LPT Schedule (2)

- MaxHeap for LPT Schedule
  - First, place a job with priority 16 in Machine A which is free
  - ReHeapify



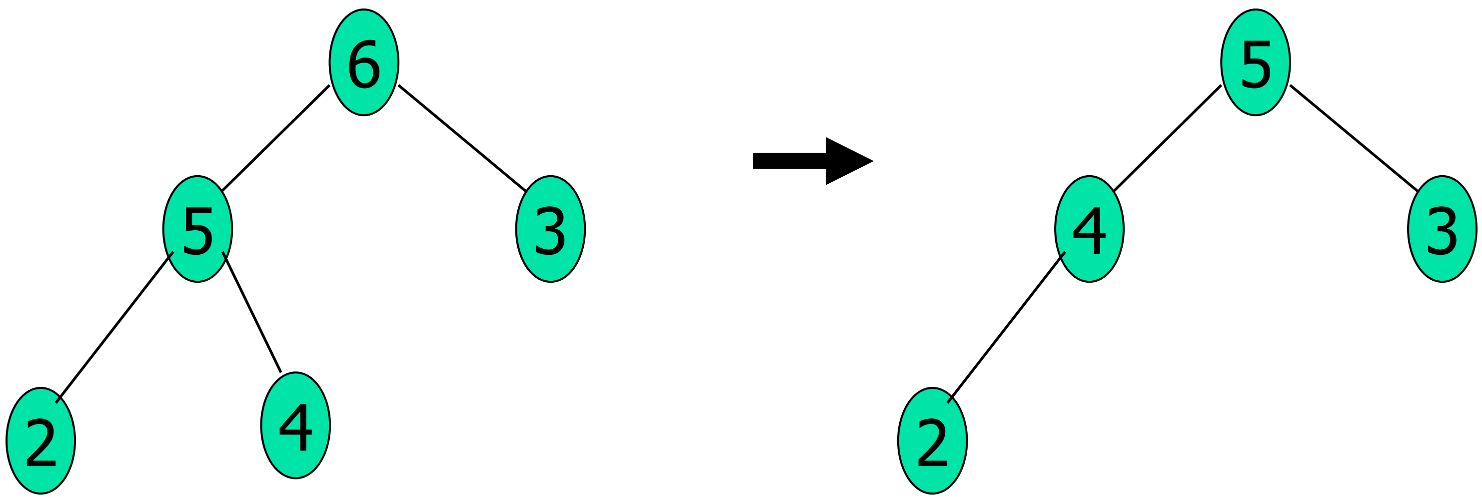
LPT Schedule (3)

- MaxHeap for LPT Schedule
  - Place a job with priority 14 in Machine B which is free
  - ReHeapify



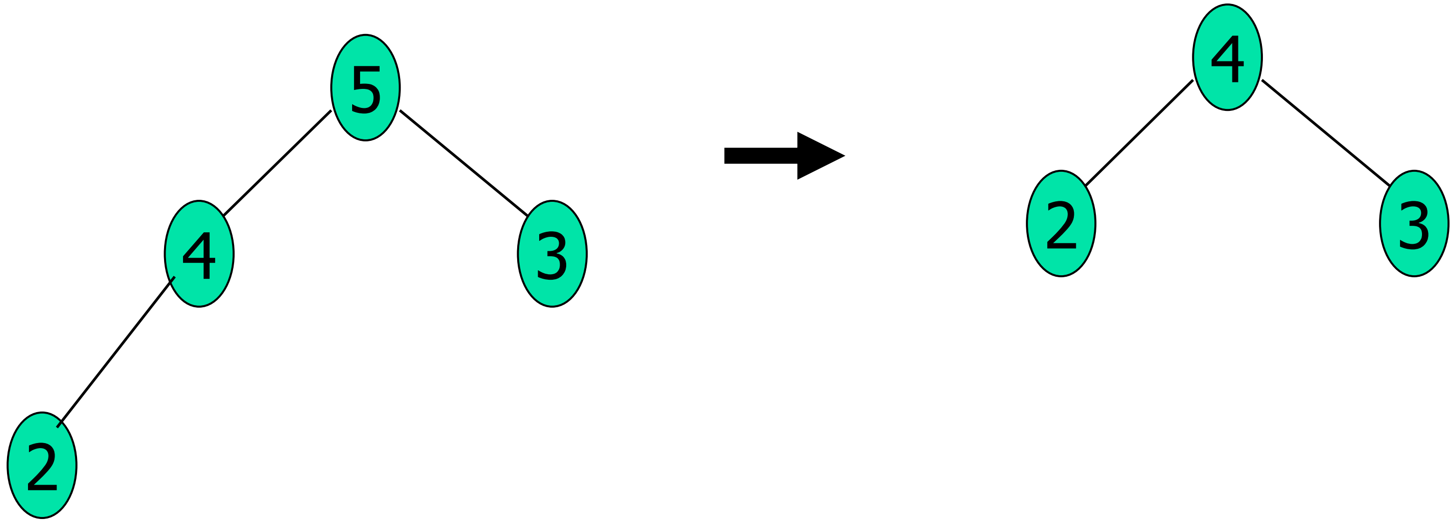
LPT Schedule (4)

- MaxHeap for LPT Schedule
  - Place a job with priority 6 in Machine C which will finish this job in the earliest time
  - ReHeapify



LPT Schedule (5)

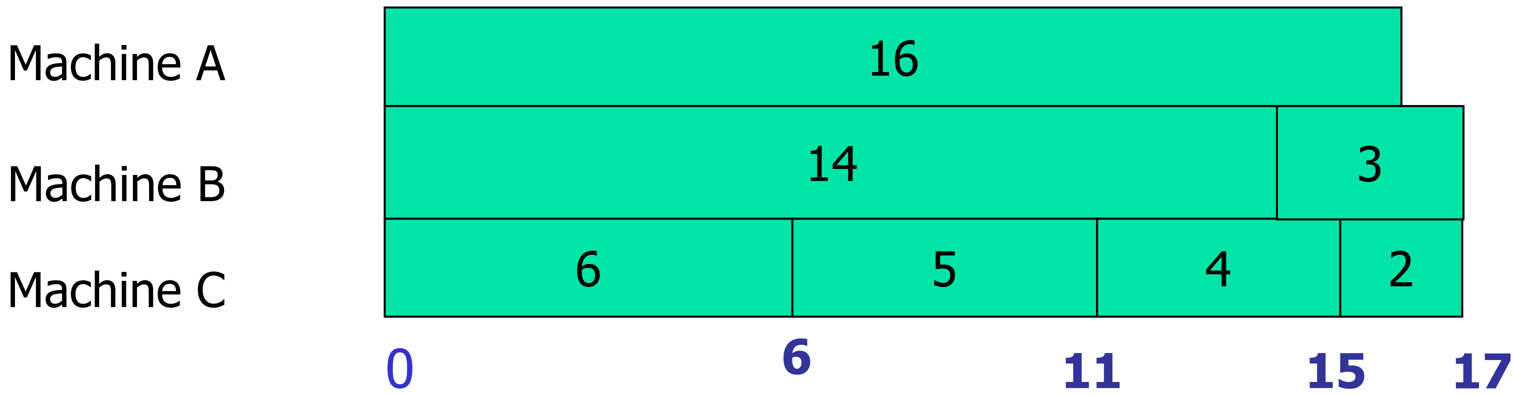
- MaxHeap for LPT Schedule
  - Place a job with priority 5 in Machine C which will finish this job in the earliest time
  - ReHeapify



** Keep Going until No element is left in the Heap

LPT Schedule (6)

- The generated schedule by LPT algorithm



- Finish time : 17

Analysis on LPT

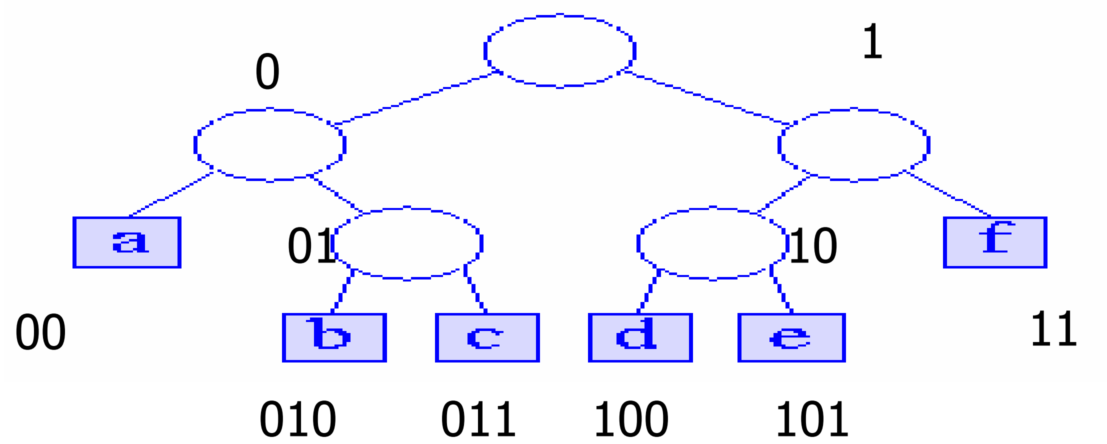
- Minimum finish time scheduling is NP-hard
  - Finding optimal solutions are generally NP
- LPT is an Approximation Algorithm
  - much closer to minimum finish time
- Proved By Graham
  - (LPT Finish Time) / (Minimum Finish Time) <= 4/3 - 1/(3m) where m is number of machines.
- Sort jobs into decreasing order of task time
  - O(n*logn) time (n is number of jobs)
- Schedule jobs in this order
  - assign job to machine that becomes available first
  - must find minimum of m (m is number of machines) finish times

Table of Contents

- Definition
- Linear Lists for Priority Queue
- Heaps for Priority Queue
- Leftist Trees for Priority Queue
- Priority Queue Applications
  - Heap Sort
  - Machine Scheduling
  - Huffman code

Huffman code

- Text compression: Suppose the codes for the paths to the nodes (a, b, c, d, e, f) are (00, 010, 011, 100, 101, 11)
  - Use extended binary trees to derive a special class of variable-length codes



- Let F(x) be the frequency of the symbol x ∊ {a, b, c, d, e, f}
- Length of the original string (by the number of bytes): 4 X number of chars
- Length of encoded string (by the number of bits)
  - 2*F(a)+3*F(b)+3*F(c)+3*F(d)+3*F(e)+2*F(f)

Huffman code encoding

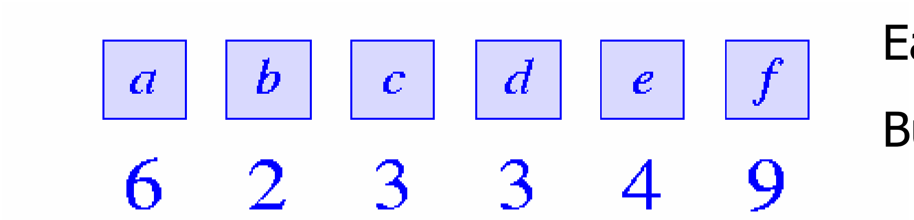
- To Encode a string using Huffman codes,
  - 1. Determine the different symbols in the string and their frequencies
  - 2. Construct a binary tree with minimum WEP (Weighted External Path length)
    - The external nodes of this tree are labeled by the symbols in the string
    - The weight of each external node is the frequency of the symbol that is its label
  - 3. Traverse the root-to-external-node paths and obtain the codes
  - 4. Replace the symbols in the string by their codes

  - 각 심볼을 bit code로 변환할 때, 빈번히 출현하는 심볼일수록 최대한 짧은 bit code를 가져야 한다.

Constructing a Huffman tree (1)

Extended Binary Tree: A binary tree with external nodes added

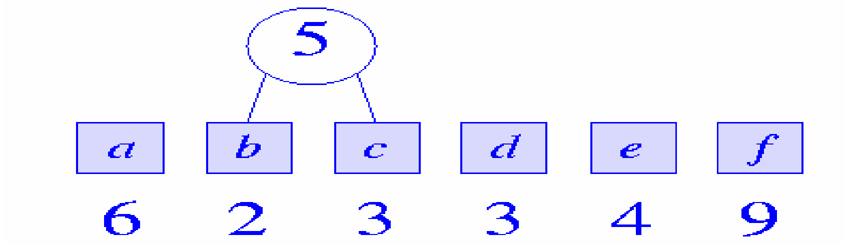
MinHeap: complete binary tree & the value in each node is less than or equal to those in its children



(a) Initial collection of trees

Each element has weight which is frequency

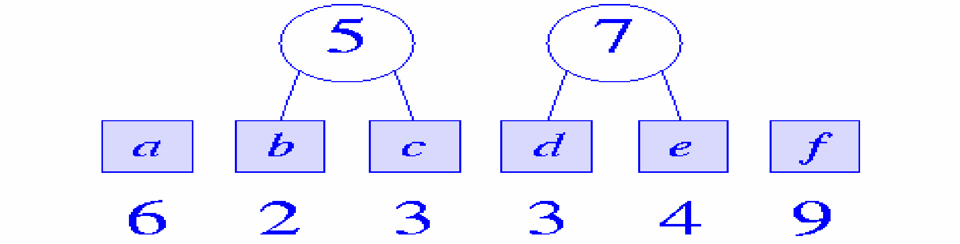
Build a MinHeap (6,2,3,3,4,9)



(b) After first combining

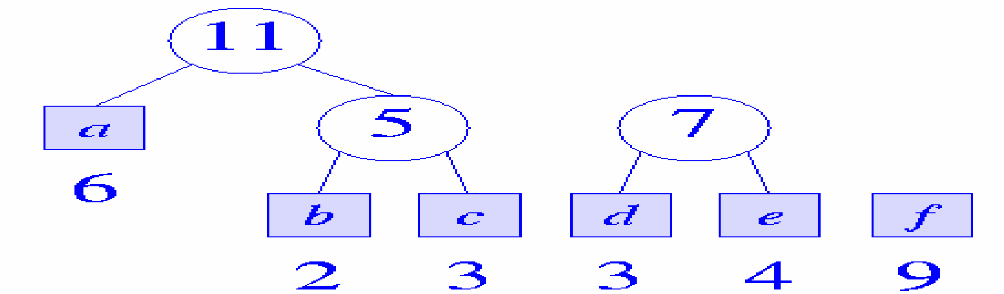
1. Remove two elements of lowest weight from a MinHeap (6,2,3,3,4,9)
2. Insert 5 & Reheapify (6,5,3,4,9)
3. Build a tree in the left

Constructing a Huffman tree (2)



(c) After second combining

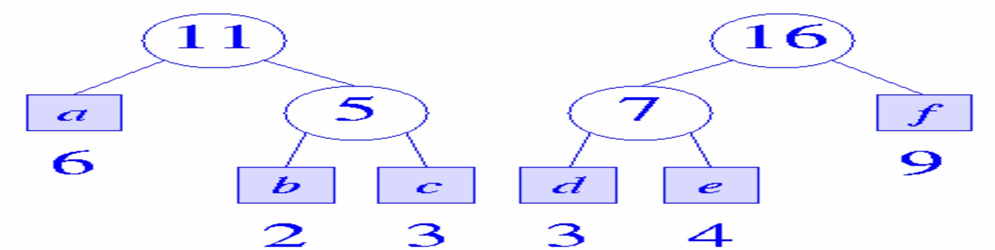
1. Remove two elements of lowest weight from a MinHeap (6,5,3,4,9)
2. Insert 7 & Reheapify (6,5,7,9)
3. Build a tree in the left



(d) After third combining

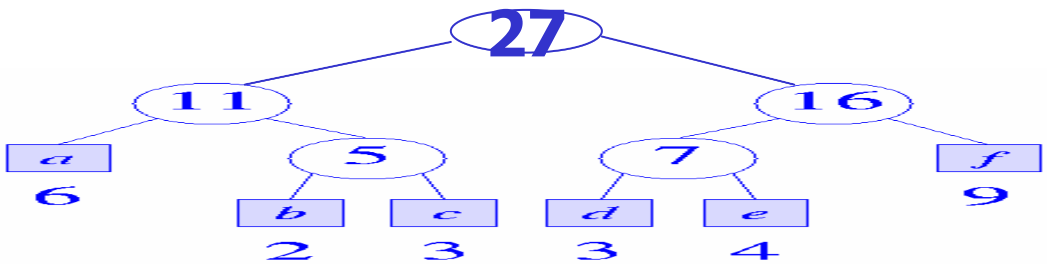
1. Remove two subelements of lowest weight from a MinHeap (6,5,7,9)
2. Insert 11 & Reheapify (11,7,9)
3. Build a tree in the left

Constructing a Huffman tree (3)



(e) After fourth combining

1. Remove two elements of lowest weight from a MinHeap (11,7,9)
2. Insert 16 & Reheap (11,16)
3. Build a tree in the left



(f) After fifth combining

1. Remove two elements of lowest weight from a MinHeap (11,16)
2. Insert 27 & Reheapify (27)
3. Build a tree in the left

Then, Assign huffman code from root to leaf nodes

huffmanTree ()

```
/** @return Huffman tree with weights w[0:a.length-1] */
public static LinkedBinaryTree huffmanTree(Operable [] w) { // create an array of single-node trees
    HuffmanNode[] hNode = new HuffmanNode[w.length+1];
    LinkedBinaryTree emptyTree = new LinkedBinaryTree();
    for (int i = 0; i < w.length; i++) {
            LinkedBinaryTree x = new LinkedBinaryTree();
            x.makeTree(new MyInteger(i), emptyTree, emptyTree);
            hNode[i + 1] = new HuffmanNode(x, w[i]); }
    MinHeap h = new MinHeap(); // make node array into a min heap
    h.initialize(hNode, w.length);
    // repeatedly combine pairs of trees from min heap until only one tree remains
    for (int i = 1; i < w.length; i++) {  // remove two lightest trees from the min heap
        HuffmanNode x = (HuffmanNode) h.removeMin();
        HuffmanNode y = (HuffmanNode) h.removeMin();
        LinkedBinaryTree z = new LinkedBinaryTree(); //combine them into a single tree t
        z.makeTree(null, x.tree, y.tree);
        HuffmanNode t = new HuffmanNode(z, (Operable) x.weight.add(y.weight));
        h.put(t); // put new tree into the min heap  }
    return ((HuffmanNode) h.removeMin()).tree;  // final tree
}
```

Summary

- A priority queue is efficiently implemented with the heap data structure.
- Priority data structure
  - Heap
  - Leftist tree: HBLT, WBLT

- Priority Queue Applications
  - Heap sort: Use heap to develop an O(nlogn) sort
  - Machine scheduling
    - Use the heap data structure to obtain an efficient implementation
  - The generation of Huffman codes

Sahni class: dataStructures.MaxPriorityQueue (p.504)

public interface ***MaxPriorityQueue*** {

methods

boolean isEmpty(): Returns *true* if empty, *false* otherwise

public int size(): Returns the number of elements in the queue

public Comparable getMax(): Returns element with maximum priority

public void put(Comparable obj): Inserts obj into the queue

public Comparable removeMax(): Removes and returns element with maximum priority

}

Sahni class: dataStructures.MinPriorityQueue (p.503)

public interface ***MinPriorityQueue*** {

methods

boolean isEmpty(): Returns *true* if empty, *false* otherwise

public int size(): Returns the number of elements in the queue

public Comparable getMin(): Returns element with minimum priority

public void put(Comparable obj): Inserts obj into the queue

public Comparable removeMin(): Removes and returns element with minimum priority

}