#### =

#### 

#### 

#### 

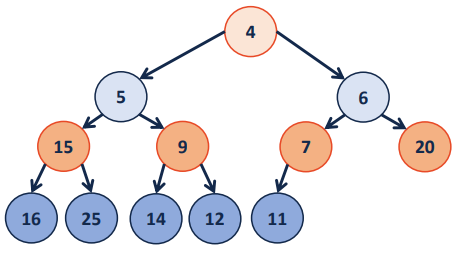
#### 

#### 

#### ...

#### **The minHeap opeartions**

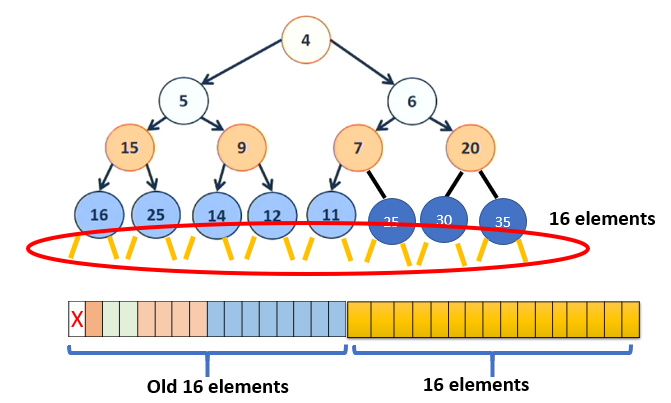
* ADT:
  + insert()
  + remove()
  + isEmpty()



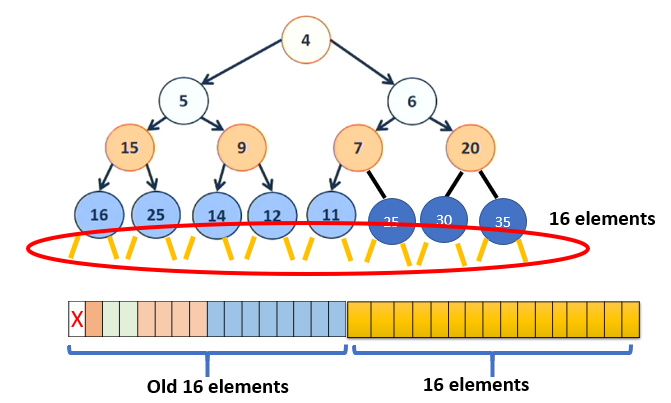
* The visualization is a tree, but the actual implementation will be an array (or vector)
  + Root is at index 1
  + For the node on index , its
    - Left child is at index
    - Right child is at index
    - Parent is at index

#### Insertion

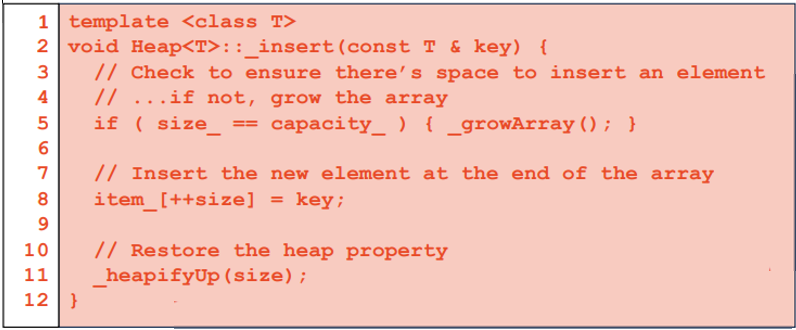
* Check if we still have the array capacityHeaps
  + If not, we double the size of the array



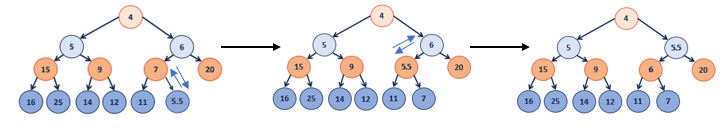
* + This is just adding a new layer to the tree

“

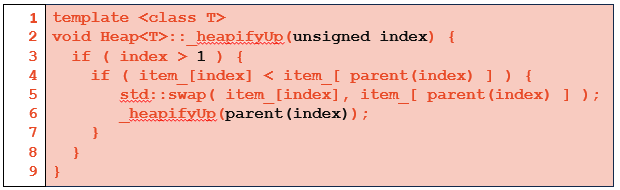
* Insert the element at the end of the array, and make sure the resulted heap is still a heap (applying heapify-up if needed)



#### **Heapify-Up**



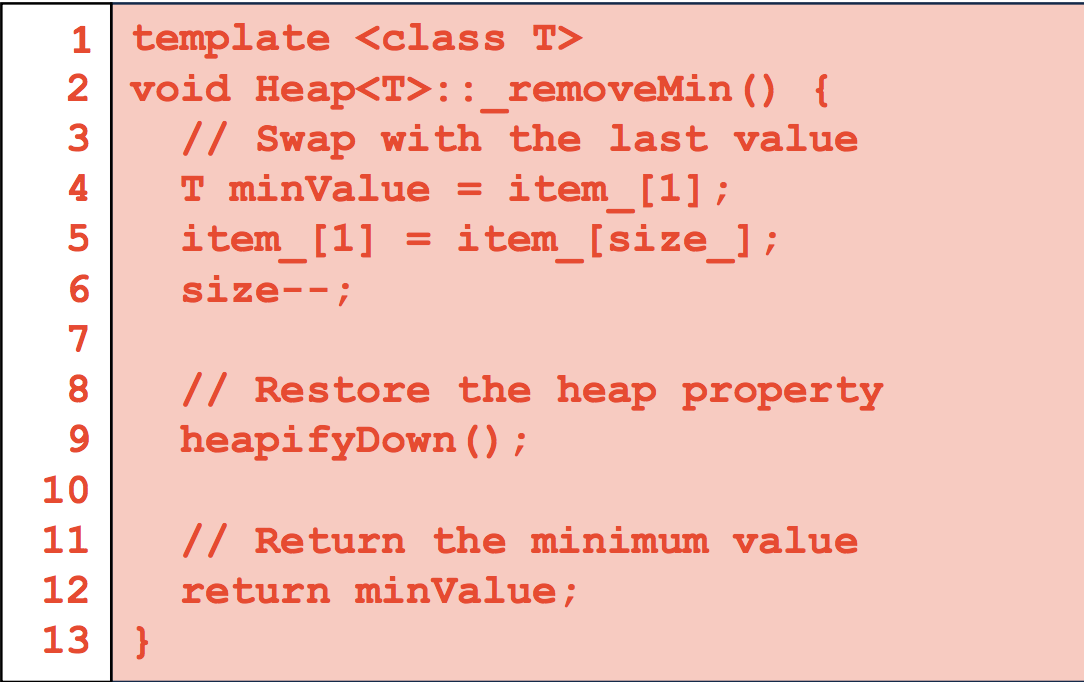
* Starting from the inserted node, and also assuming the heap is valid everywhere above that inserted node
* If the current element is not the root, and smaller than its parent:
  + Swap the current element with its parent node
  + Continue to applying heapifyUp on the parent node



* Runtime of Insertion operation
  + growArray() takes O(1) amortized
  + insertion takes O(1)
  + heapify-up takes O(h) = O(lg n) since the tree is complete
  + Total runtime: O(lg n)

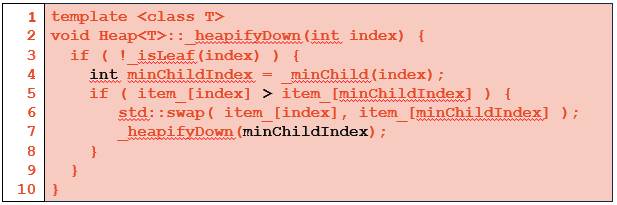
#### **Remove**

* Swap the root with the last element
* Then remove the last element
* Heapify-Down to ensure the heap property is preserved.

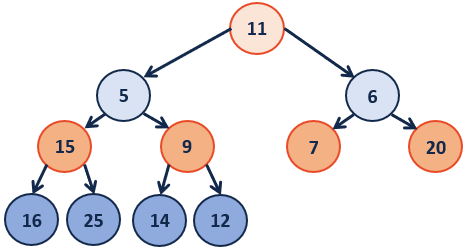


#### **Heapify-Down**

* Starting from the root node with the assumption that both subtrees are valid heaps
* If current is not leaf, find the minChild among the two children
  + Swap the value of minChild and subRoot if needed
  + Continue on the minChild node if swap happened



* Runtime of Remove()
  + swap takes O(1)
  + heapify-down takes O(h) = O(lg n) since the tree is complete
  + **Total runtime: O(log n)**
* **Example of HeapifyDown**





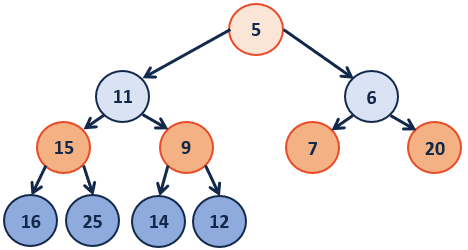
HeapifyDown(1):

minChildIndex = 2;

if(11 > 5) 😊 (true)

swap elements;

heapifyDown(2);

；



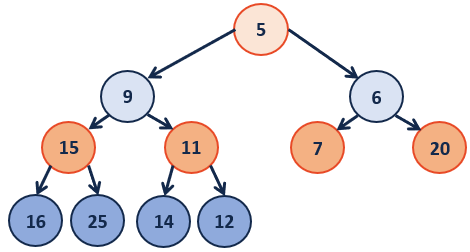
HeapifyDown(2):

minChildIndex = 5;

if(11 > 9) 😊

swap elements;

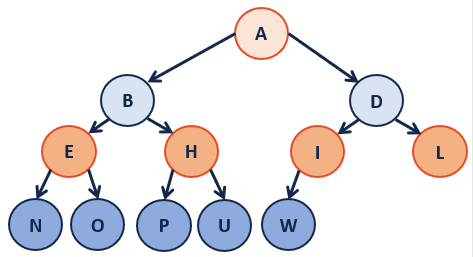
heapifyDown(5);





HeapifyDown(5):

minChildIndex = 11;

if(11 > 12) ☹

Done: Heap property restored!

*Everything so far can be done using an AVL tree under the same runtime*

*But the below function, buildHeap, gives Heaps the edge over AVLs*

#### **BuildHeap**

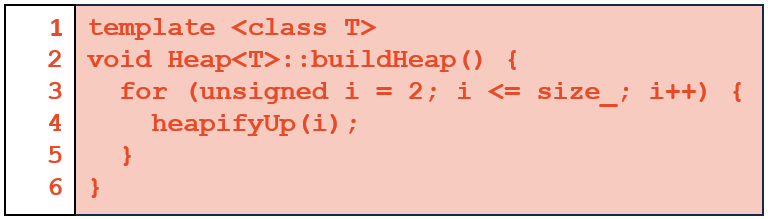
* We want to build a heap using a given array:
* **Method 1: sorting**



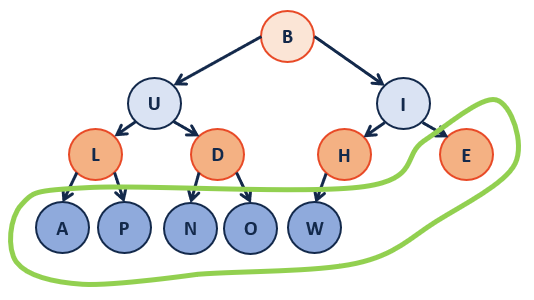
* + A sorted array is always a heap

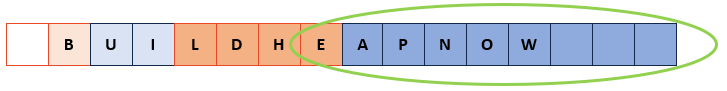
\*\*

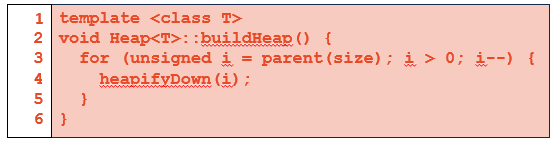
* + Runtime:
* **Method 2: heapify-up**
  + Call Heapify-Up on every element from the root



* + Takes for every element, so in total
* **Method 3: heapify-down**
  + Call Heapify-Down on every element from the end of the array
    - Notice that the last level already has the heap property!
    - We can start from the second-last level
    - In the case below, “H” is the first element that is not a heap







* + Since heapify-down runs in O(h) time:
    - heapify on “H” takes 1 unit of work
    - heapify on “I” takes 2 units of work
    - heapify on “B” takes 3 units of work
    - In total, we have 1+1+1+2+2 + 3 = 10 units of work, this is just linear to the number of elements
  + Then we have the runtime:
    - Proof in next lecture