CE100 Algorithms and Programming II

Heap/Heap Sort

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-4 (Heap/Heap Sort)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-4-heap.md_doc.pdf), [SLIDE](ce100-week-4-heap.md_slide.pdf), [PPTX](ce100-week-4-heap.md_slide.pptx)

## Heap/Heap Sort

## Outline (1)

* Heaps
  + Max / Min Heap
* Heap Data Structure
  + Heapify
    - Iterative
    - Recursive

## Outline (2)

* Extract-Max
* Build Heap

## Outline (3)

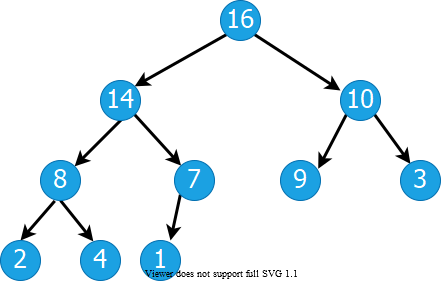
* Heap Sort
* Priority Queues
* Linked Lists
* Radix Sort
* Counting Sort

## Heapsort

* Worst-case runtime:
* Sorts in-place
* Uses a special data structure (heap) to manage information during execution of the algorithm
  + Another design paradigm

## Heap Data Structure (1)

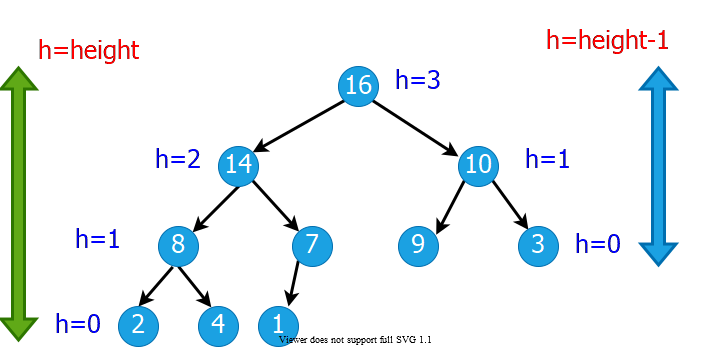
* Nearly complete binary tree
  + Completely filled on all levels except possibly the lowest level



alt:“alt” height:350px center

## Heap Data Structure (2)

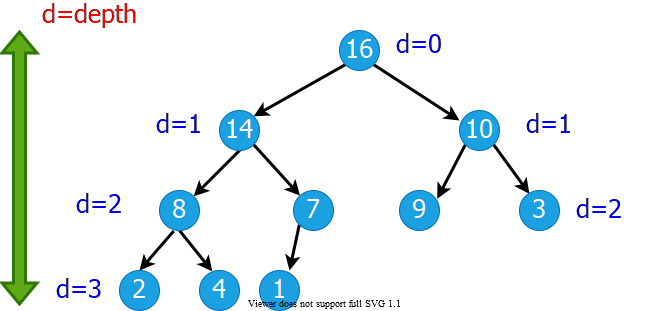
* **Height of node i:** Length of the longest simple downward path from **i** to a **leaf**
* **Height of the tree:** height of the **root**



alt:“alt” height:400px center

## Heap Data Structures (3)

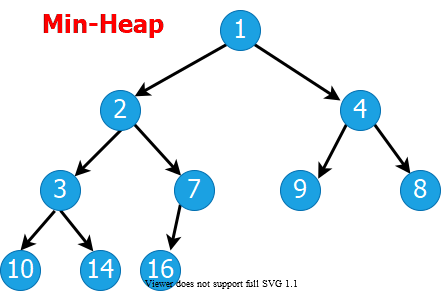
* **Depth of node i:** Length of the simple downward path from the **root** to node **i**



alt:“alt” height:350px center

## Heap Property: Min-Heap

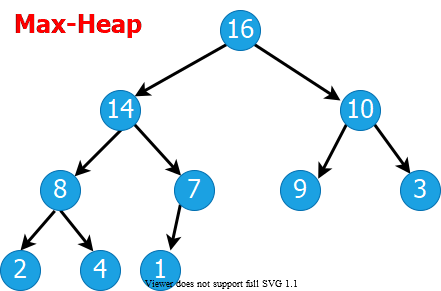
* The **smallest** element in any subtree is the **root** element in a **min-heap**
* **Min heap:** For every node **i** other than **root**,
  + Parent node is always smaller than the child nodes



bg right:40% w:480

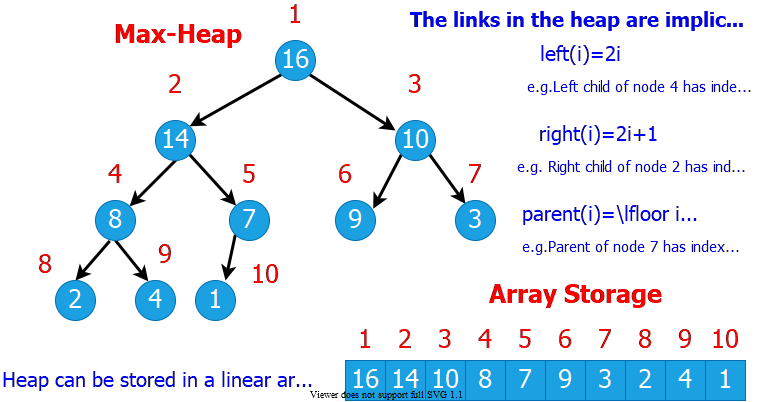
## Heap Property: Max-Heap

* The **largest** element in any subtree is the **root** element in a **max-heap**
  + We will focus on max-heaps
* **Max heap:** For every node **i** other than **root**,
  + Parent node is always larger than the child nodes



bg right:40% w:480

## Heap Data Structures (4)



alt:“alt” height:500px center

## Heap Data Structures (5)

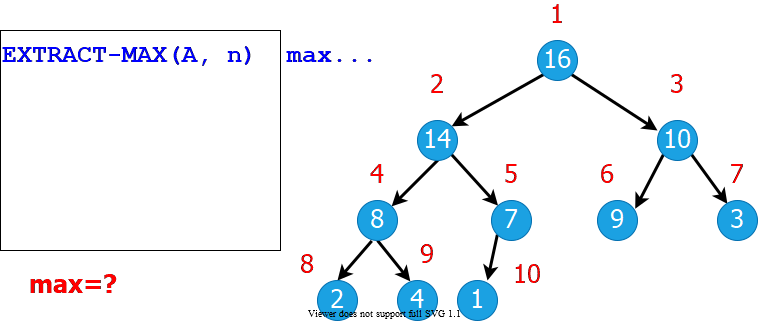
* Computing left child, right child, and parent indices very fast
  + **left(i) = 2i** binary left shift
  + **right(i) = 2i+1** binary left shift, then set the lowest bit to 1
  + **parent(i) = floor(i/2)** right shift in binary
* is always the **root** element
* Array has two attributes:
  + **length(A):** The number of elements in
  + **n = heap-size(A):** The number elements in

## Heap Operations : EXTRACT-MAX (1)

EXTRACT-MAX(A, n)  
 max = A[1]  
 A[1] = A[n]  
 n = n - 1  
 HEAPIFY(A, 1,n)  
 return max

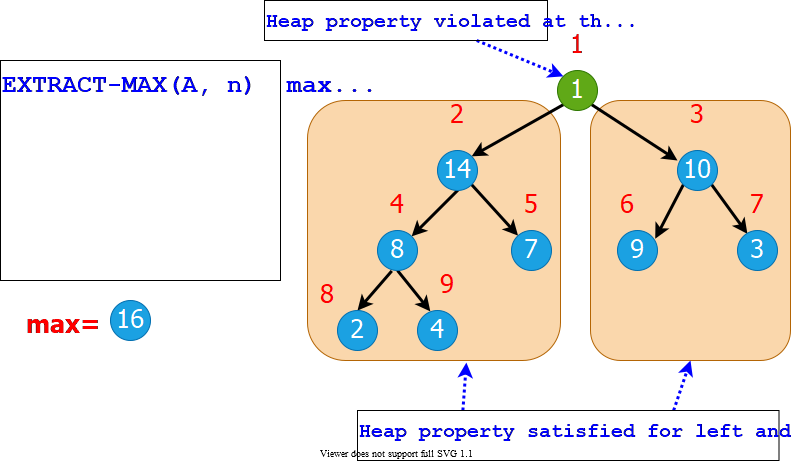
## Heap Operations : EXTRACT-MAX (2)

* Return the max element,and reorganize the heap to maintain heap property



alt:“alt” height:400px center

## Heap Operations: HEAPIFY (1)



alt:“alt” height:500px center

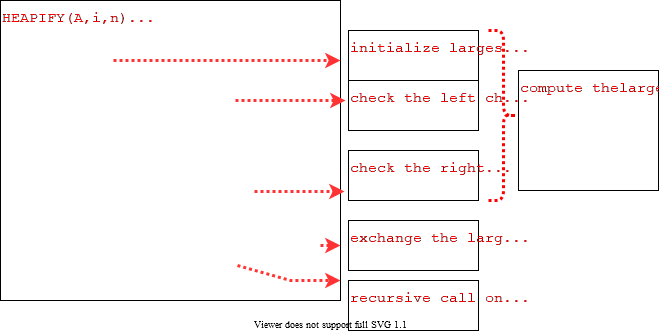
## Heap Operations: HEAPIFY (2)

* Maintaining heap property:
  + Subtrees rooted at and are already heaps.
  + But, may violate the heap property (i.e., may be smaller than its children)
* **Idea:** Float down the value at in the heap so that subtree rooted at becomes a heap.

## Heap Operations: HEAPIFY (2)

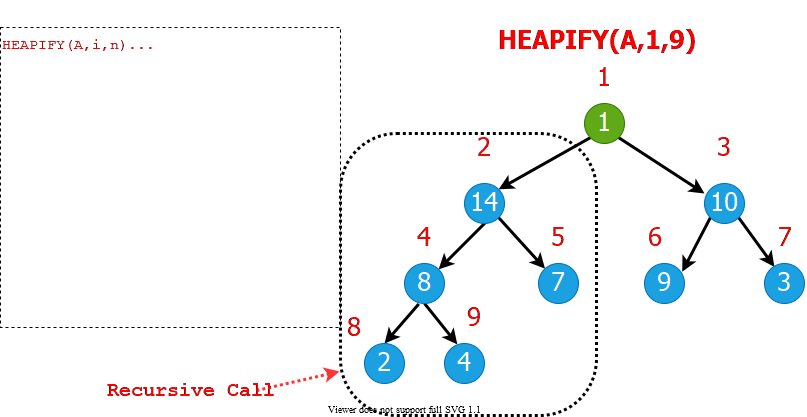
HEAPIFY(A, i, n)  
 largest = i   
  
 if 2i <= n and A[2i] > A[i] then   
 largest = 2i;  
 endif  
  
 if 2i+1 <= n and A[2i+1] > A[largest] then   
 largest = 2i+1;  
 endif  
  
 if largest != i then  
 exchange A[i] with A[largest];  
 HEAPIFY(A, largest, n);  
 endif

## Heap Operations: HEAPIFY (3)



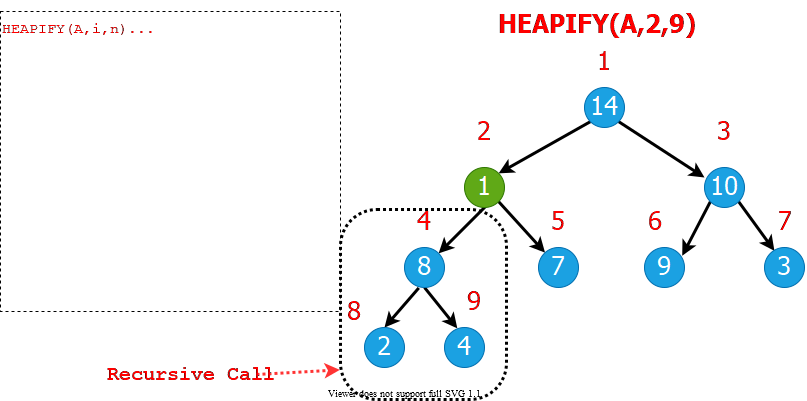
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (4)



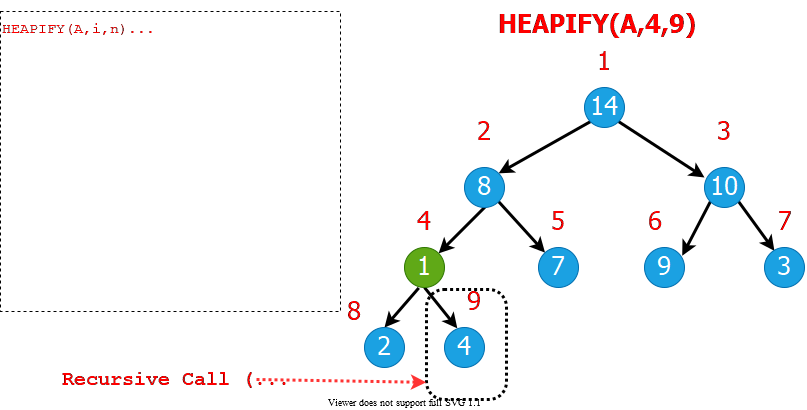
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (5)



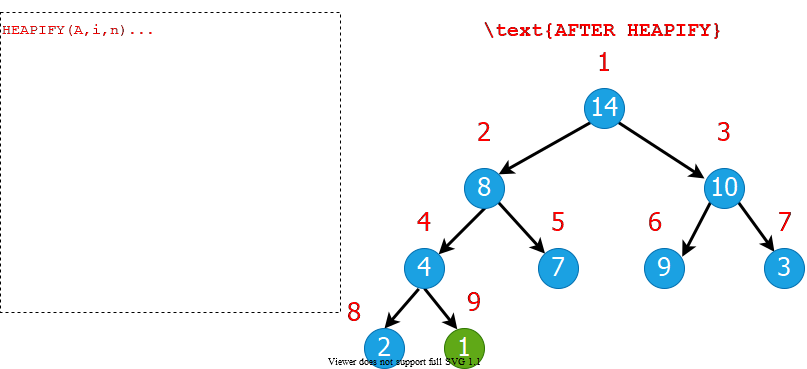
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (6)



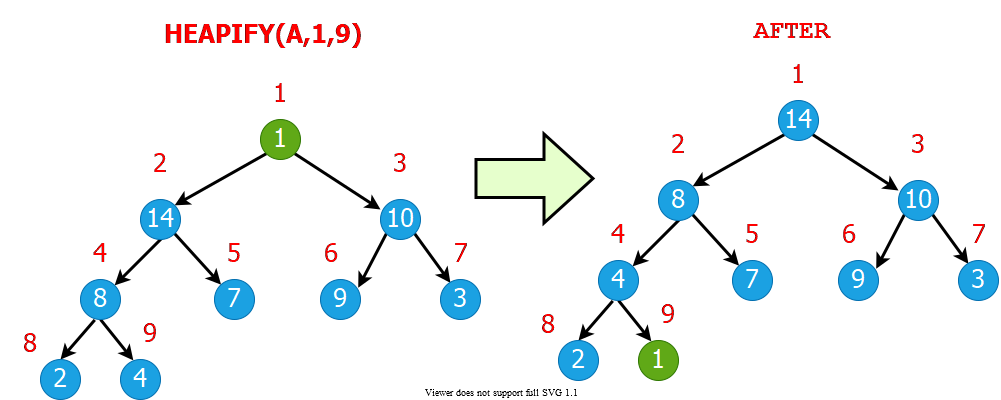
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (7)



alt:“alt” height:500px center

## Heap Operations: HEAPIFY (8)



alt:“alt” height:500px center

## Intuitive Analysis of HEAPIFY

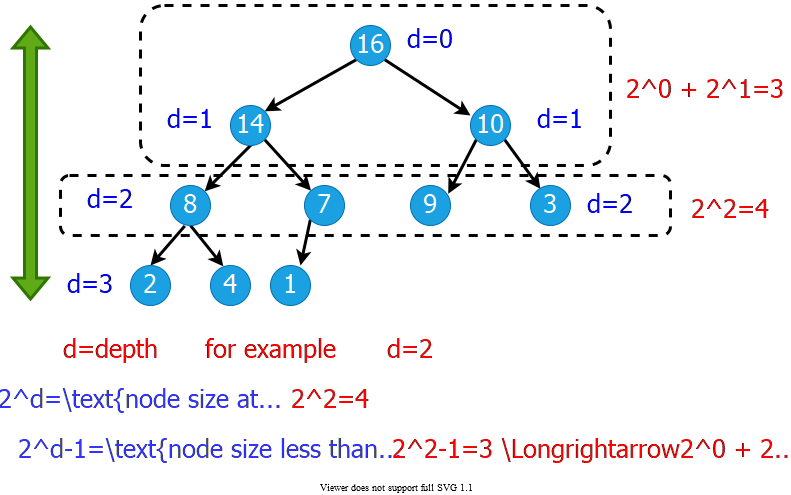
* Consider
  + let be the height of node
  + at most recursion levels
    - Constant work at each level:
  + Therefore
* Heap is almost-complete binary tree
* Thus

## Formal Analysis of HEAPIFY

* **What is the recurrence?**
  + Depends on the size of the **subtree** on which recursive call is made
    - In the next, we try to compute an **upper bound** for this **subtree**.

## Reminder: Binary trees

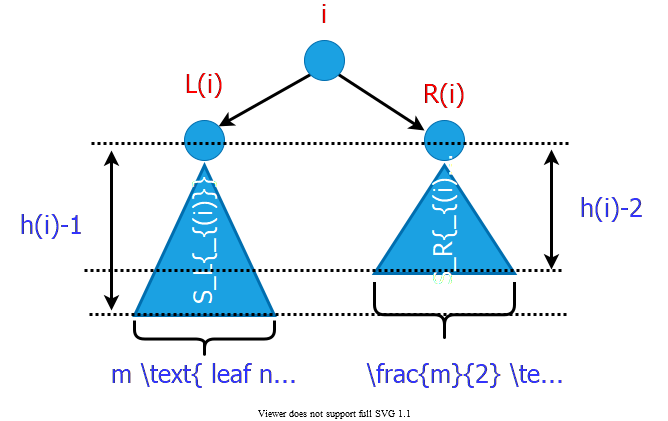
* For a complete binary tree:
  + of nodes at depth :
  + of nodes with depths less than :



alt:“alt” height:450px center

## Formal Analysis of HEAPIFY (1)

* Worst case occurs when last row of the subtree rooted at node is **half full**
* and are complete binary trees of heights and , respectively



bg right:50% w:600

## Formal Analysis of HEAPIFY (2)

* Let be the number of **leaf nodes** in

## Formal Analysis of HEAPIFY (2)

* **By CASE-2 of Master Theorem**

## Formal Analysis of HEAPIFY (2)

* Recurrence:
* *Case 2:*
* i.e., and grow at similar rates
* **Solution:**
  + (drop constants.)

## HEAPIFY: Efficiency Issues

* **Recursion vs Iteration:**
  + In the absence of tail recursion, **iterative version** is in general **more efficient** because of the **pop/push** operations **to/from** stack at each **level of recursion**.

## Heap Operations: HEAPIFY (1)

**Recursive**

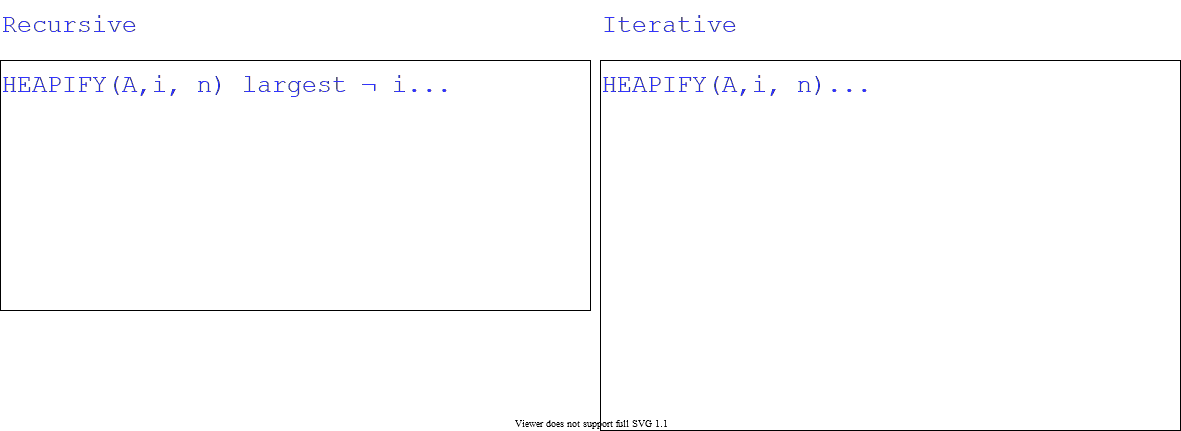
HEAPIFY(A, i, n)  
 largest = i   
  
 if 2i <= n and A[2i] > A[i] then   
 largest = 2i  
  
 if 2i+1 <= n and A[2i+1] > A[largest] then   
 largest = 2i+1  
  
 if largest != i then  
 exchange A[i] with A[largest]  
 HEAPIFY(A, largest, n)

## Heap Operations: HEAPIFY (2)

**Iterative**

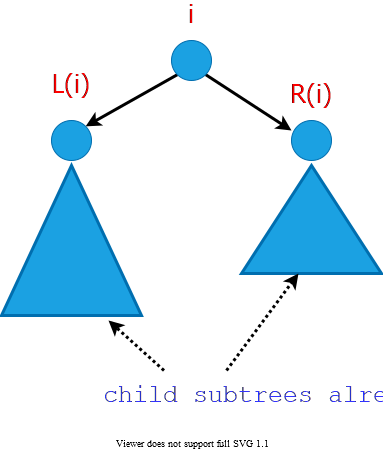
HEAPIFY(A, i, n)  
 j = i  
 while(true) do  
 largest = j   
  
 if 2j <= n and A[2j] > A[j] then   
 largest = 2j  
  
 if 2j+1 <= n and A[2j+1] > A[largest] then   
 largest = 2j+1  
  
 if largest != j then  
 exchange A[j] with A[largest]  
 j = largest  
 else return

## Heap Operations: HEAPIFY (3)



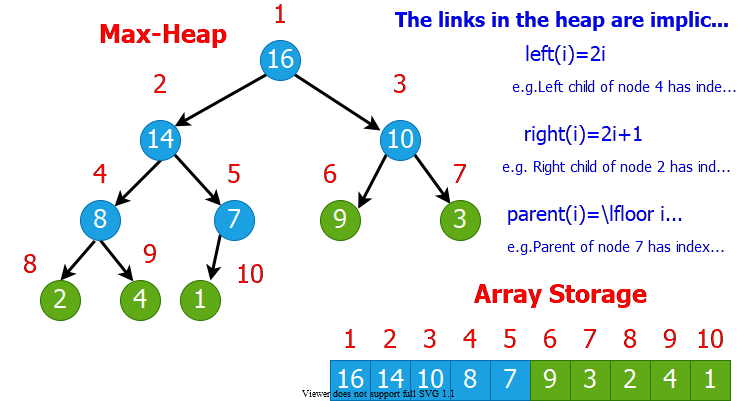
alt:“alt” height:500px center

## Heap Operations: Building Heap

* Given an arbitrary array, how to build a heap from scratch?
* **Basic idea:** Call on each node bottom up
  + Start from the leaves (which trivially satisfy the heap property)
  + Process nodes in bottom up order.
  + When is called on node , the subtrees connected to the and subtrees already satisfy the heap property.
* 
* bg right:25% w:300px

## Storage of the leaves (Lemma)

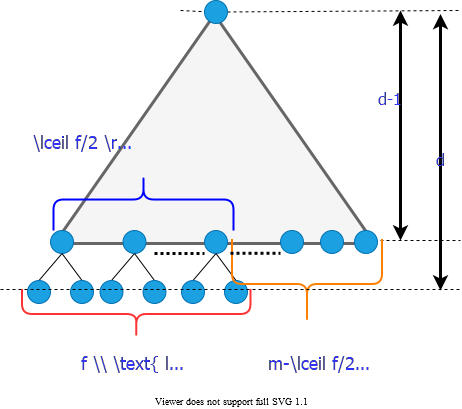
* **Lemma:** The last nodes of a heap are all leaves.



bg right:70% w:800

## Storage of the leaves (Proof of Lemma) (1)

* **Lemma:** last nodes of a heap are all leaves
* Proof :
  + : nodes at level
  + : nodes at level (last level)
* of nodes with depth :
* of nodes with depth :
* of nodes with depth :
* **Total** of nodes :

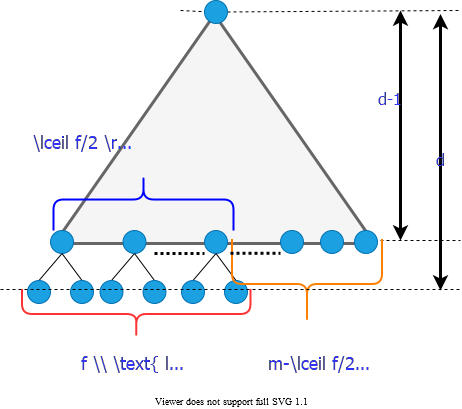


bg right:40% w:480

## Storage of the leaves (Proof of Lemma) (2)

* **Total** of nodes :

Proof is Completed



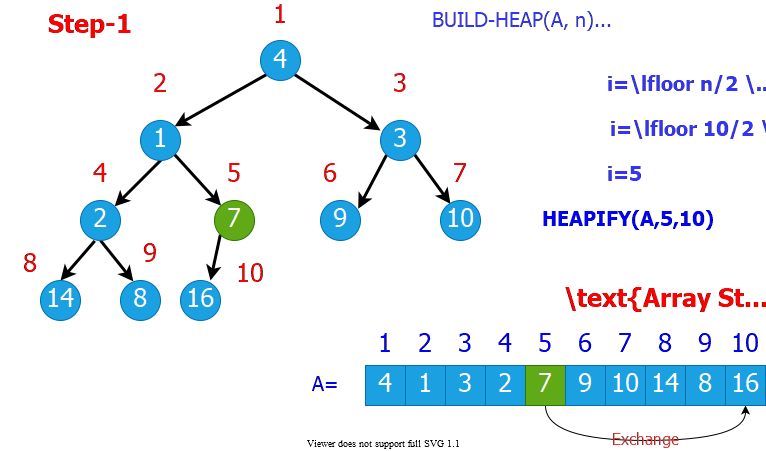
bg right:40% w:480

## Heap Operations: Building Heap

BUILD-HEAP (A, n)  
 for i = ceil(n/2) downto 1 do  
 HEAPIFY(A, i, n)

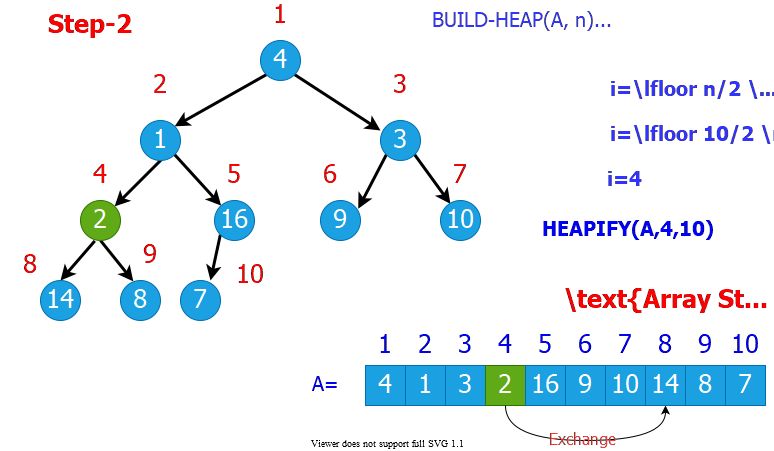
* **Reminder:** The last nodes of a heap are **all leaves**, which trivially satisfy the heap property

## Build-Heap Example (Step-1)



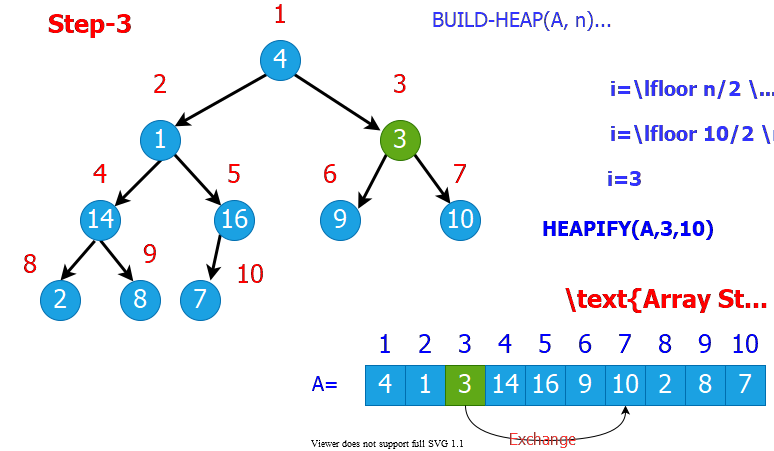
bg right:70% w:800px

## Build-Heap Example (Step-2)



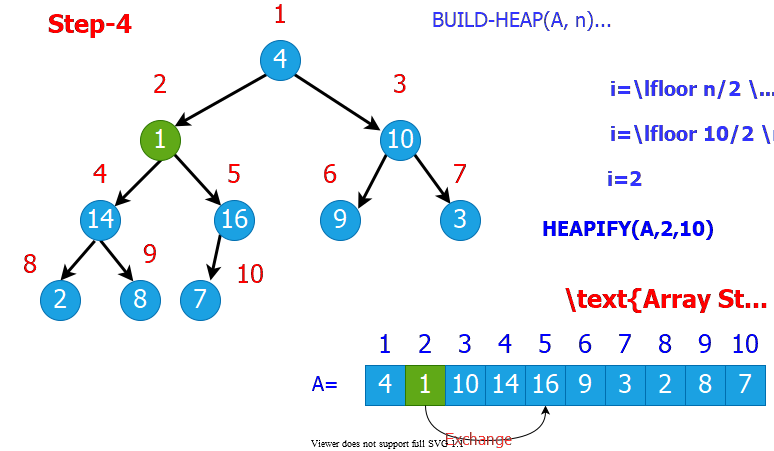
bg right:70% w:800px

## Build-Heap Example (Step-3)



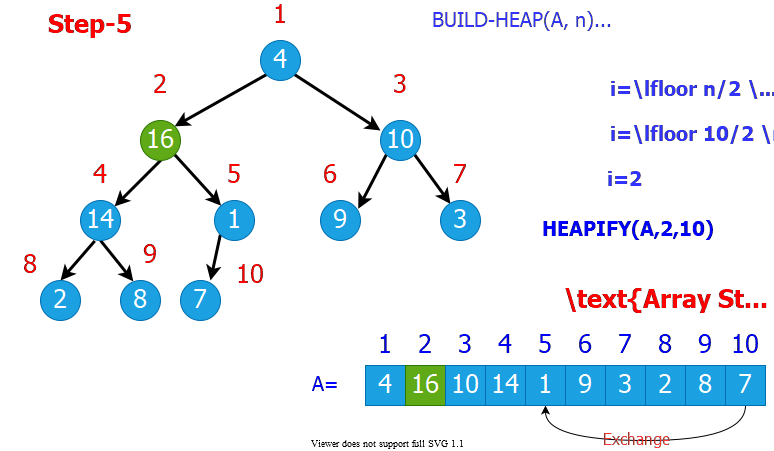
bg right:70% w:800px

## Build-Heap Example (Step-4)



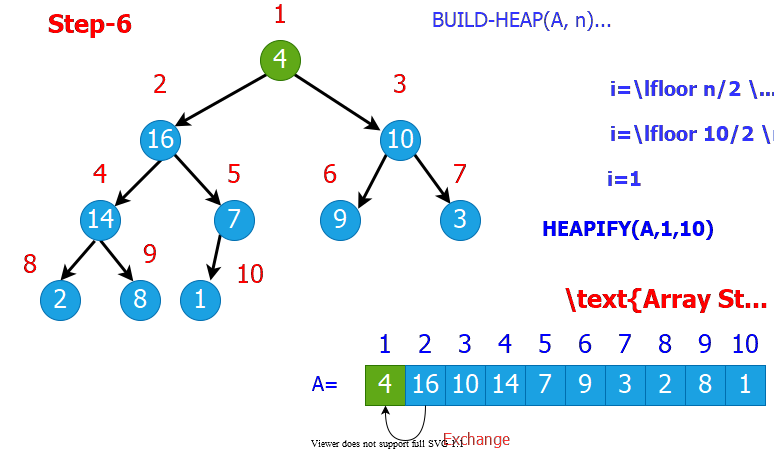
bg right:70% w:800px

## Build-Heap Example (Step-5)



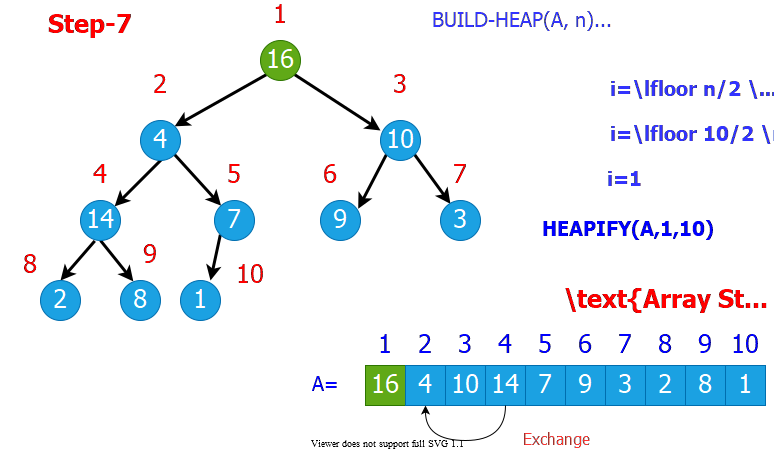
bg right:70% w:800px

## Build-Heap Example (Step-6)



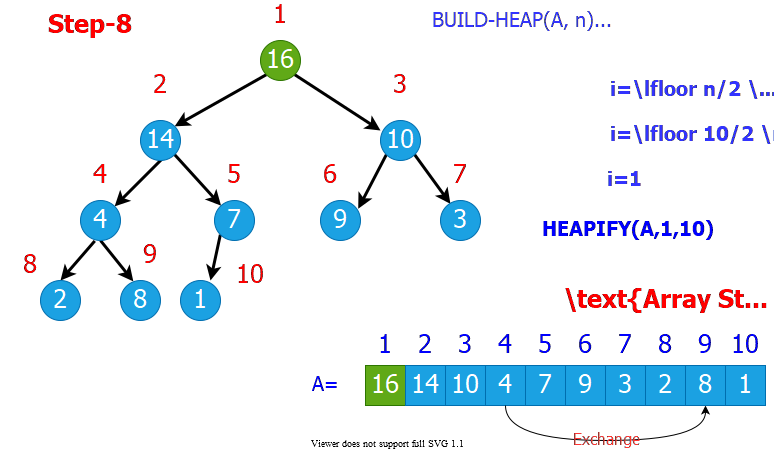
bg right:70% w:800px

## Build-Heap Example (Step-7)



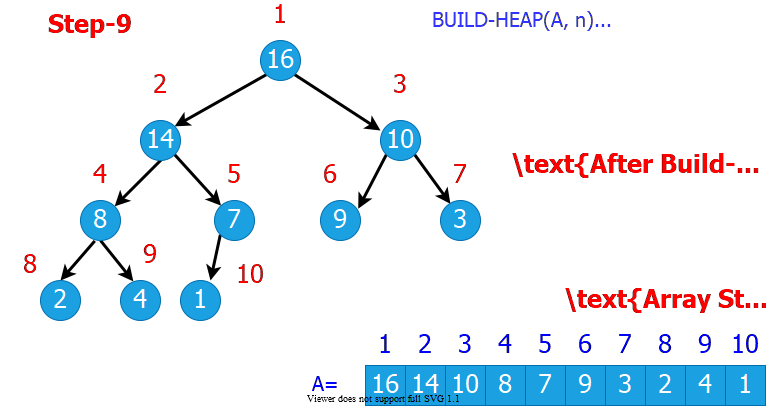
bg right:70% w:800px

## Build-Heap Example (Step-8)



bg right:70% w:800px

## Build-Heap Example (Step-9)



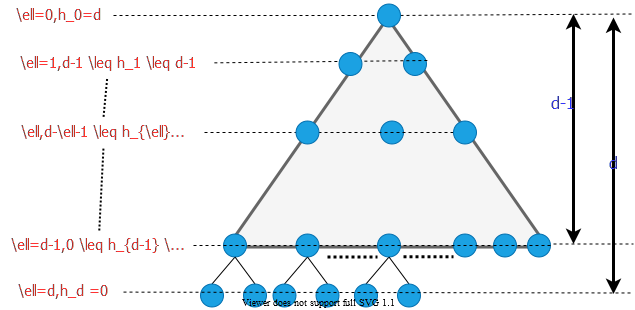
bg right:70% w:800px

## Build-Heap: Runtime Analysis

* Simple analysis:
  + calls to , each of which takes time
  + loose bound
* In general, a good approach:
  + Start by proving an easy bound
  + Then, try to tighten it
* Is there a tighter bound?

## Build-Heap: **Tighter** Running Time Analysis

* If the heap is complete binary tree then
* Otherwise, nodes at a given level do not all have the same height, But we have



alt:“alt” height:420px center

## Build-Heap: **Tighter** Running Time Analysis

* Assume that all nodes at level are processed

$$ $$

## Build-Heap: **Tighter** Running Time Analysis

* recall infinite decreasing geometric series
* differentiate both sides

## Build-Heap: **Tighter** Running Time Analysis

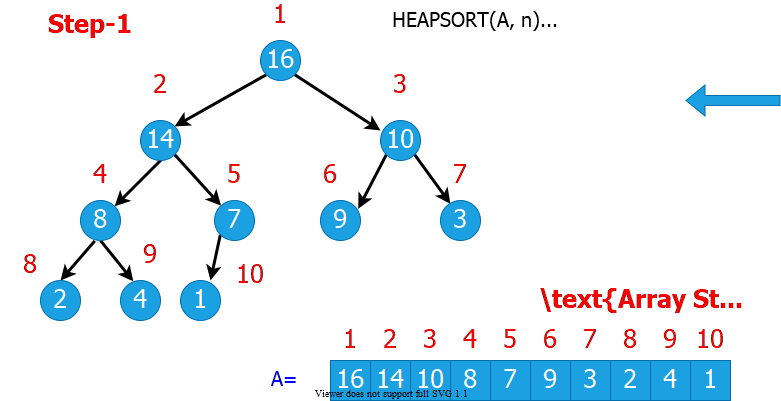
* then, multiply both sides by
* in our case: and

$$
\therefore \sum\_{h=0}^{\infty}h(1/2)^h = \frac{1/2}{(1-(1/2))^2}=2=O(1) \\
\therefore T(n)=O(n\sum\_{h=1}^{d}h(1/2)^h)=O(n)
$$

## Heapsort Algorithm Steps

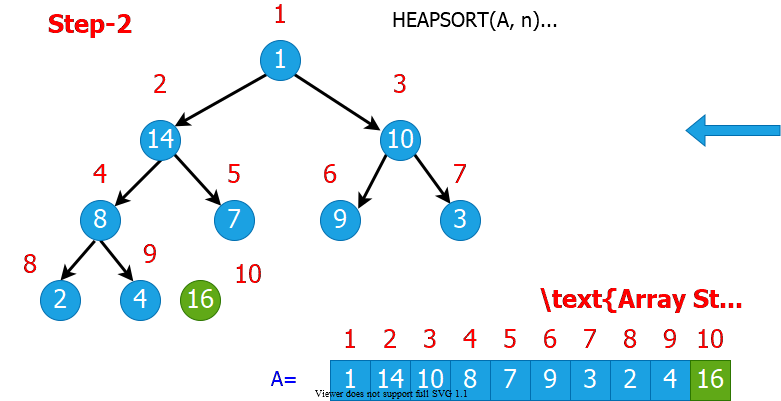
* **(1)** Build a heap on array by calling
* **(2)** The largest element is stored at the root
  + Put it into its correct final position by
* **(3)** Discard node from the heap
* **(4)** Subtrees rooted at children of root remain as heaps, but the new root element may violate the heap property.
  + Make a heap by calling
* **(5)**
* **(6)** Repeat steps **(2-4)** until

## Heapsort Algorithm Example (Step-1)



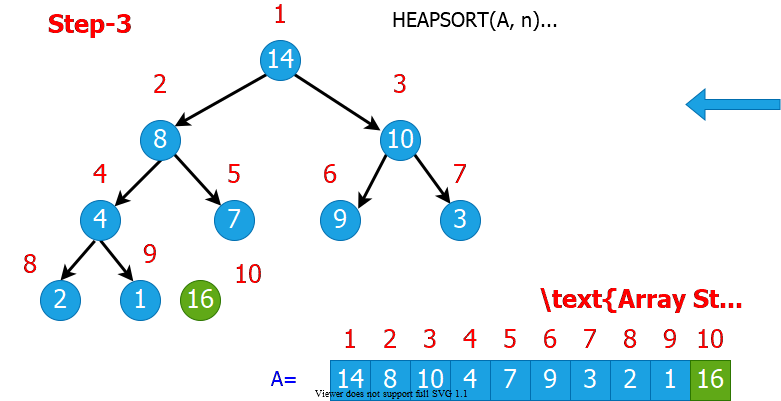
bg right:70% w:800px

## Heapsort Algorithm Example (Step-2)



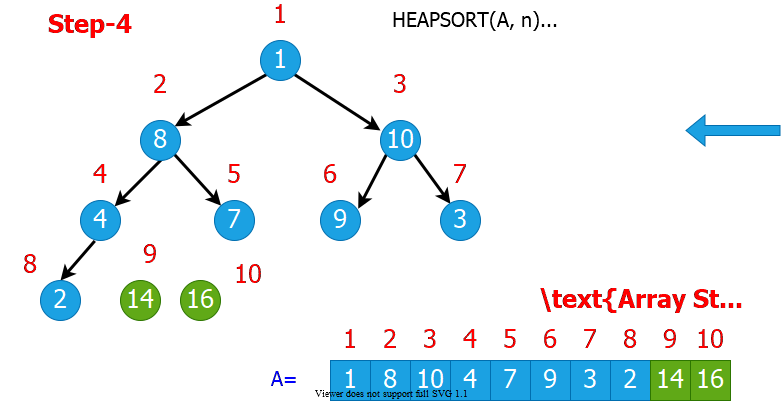
bg right:70% w:800px

## Heapsort Algorithm Example (Step-3)



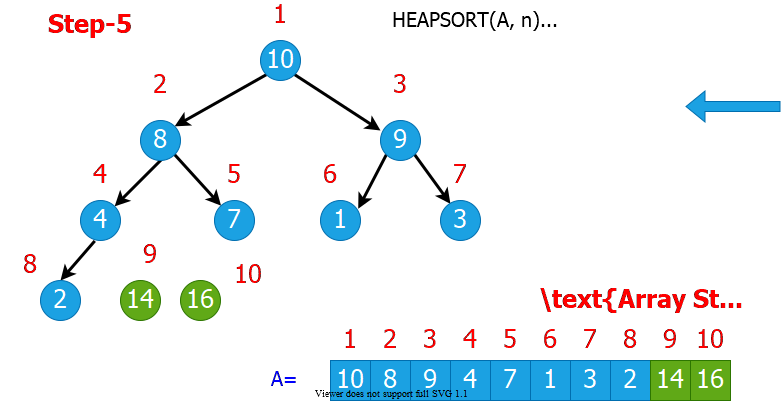
bg right:70% w:800px

## Heapsort Algorithm Example (Step-4)



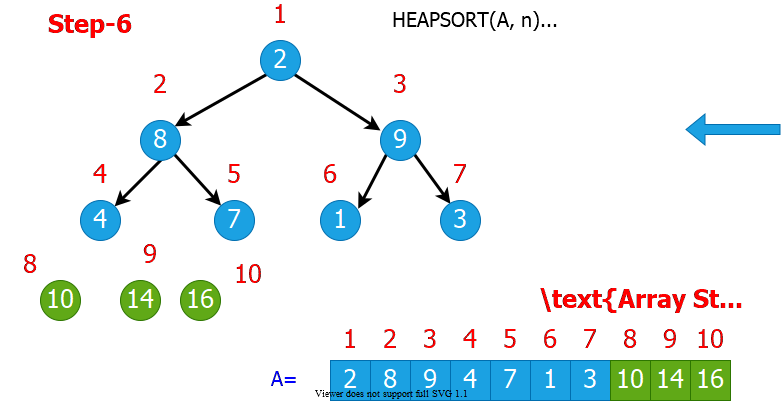
bg right:70% w:800px

## Heapsort Algorithm Example (Step-5)



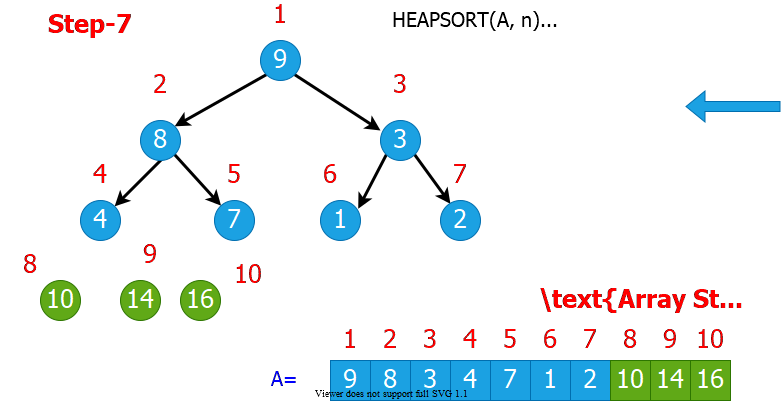
bg right:70% w:800px

## Heapsort Algorithm Example (Step-6)



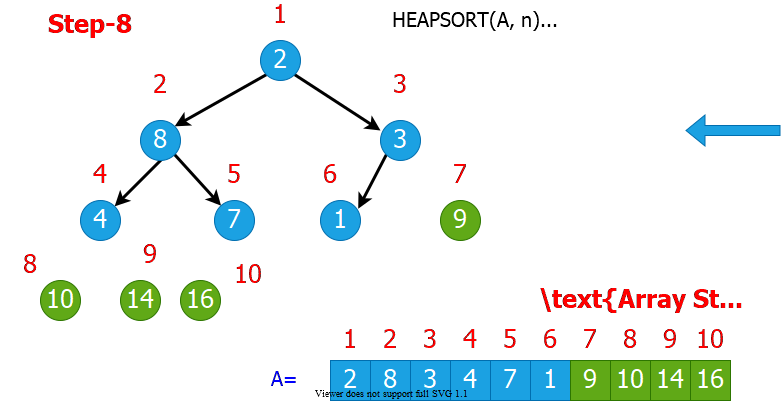
bg right:70% w:800px

## Heapsort Algorithm Example (Step-7)



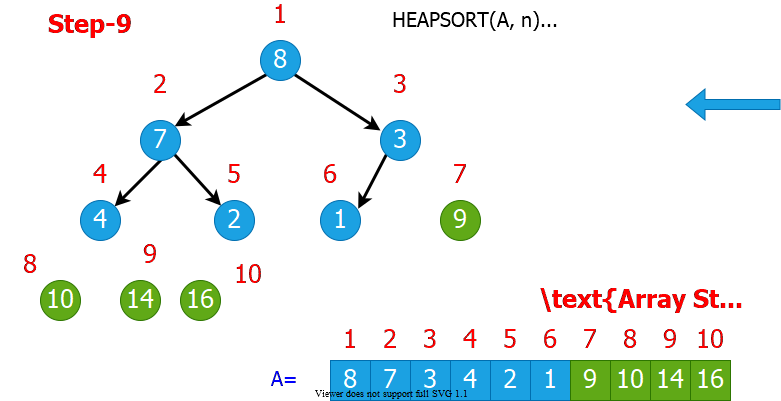
bg right:70% w:800px

## Heapsort Algorithm Example (Step-8)



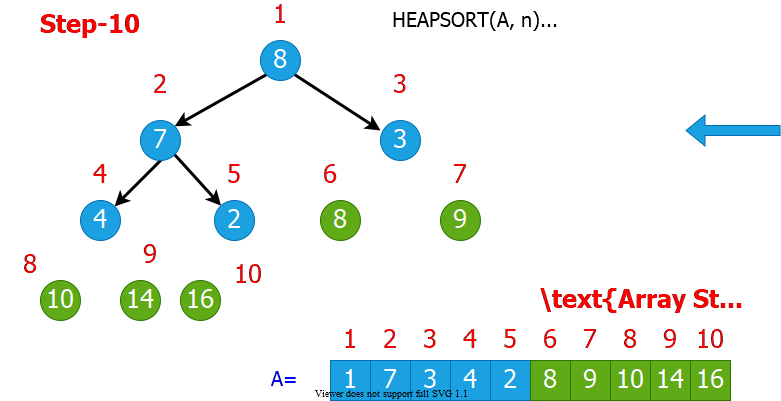
bg right:70% w:800px

## Heapsort Algorithm Example (Step-9)



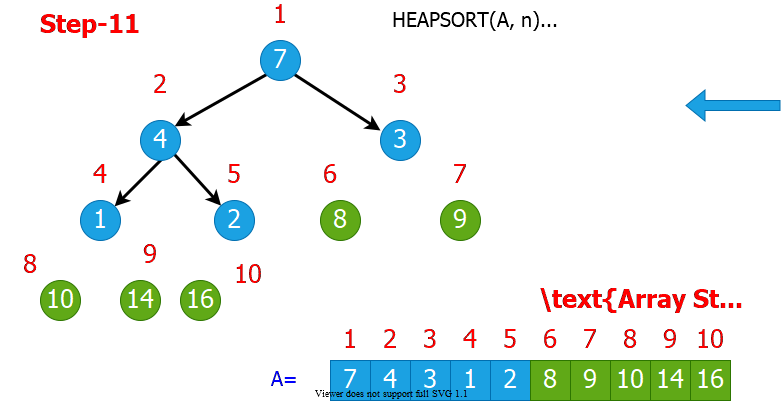
bg right:70% w:800px

## Heapsort Algorithm Example (Step-10)



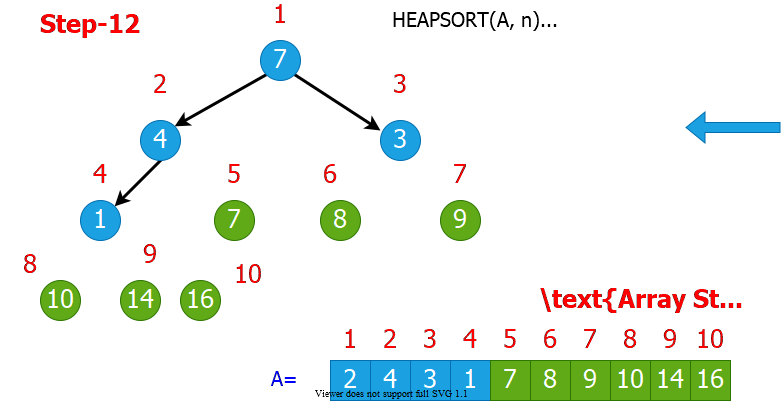
bg right:70% w:800px

## Heapsort Algorithm Example (Step-11)



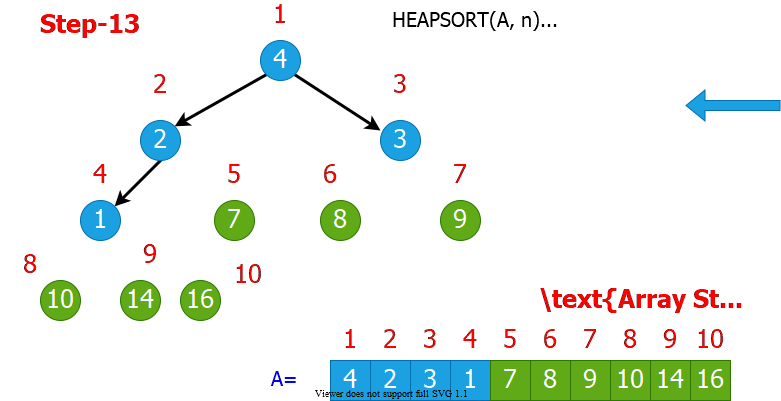
bg right:70% w:800px

## Heapsort Algorithm Example (Step-12)



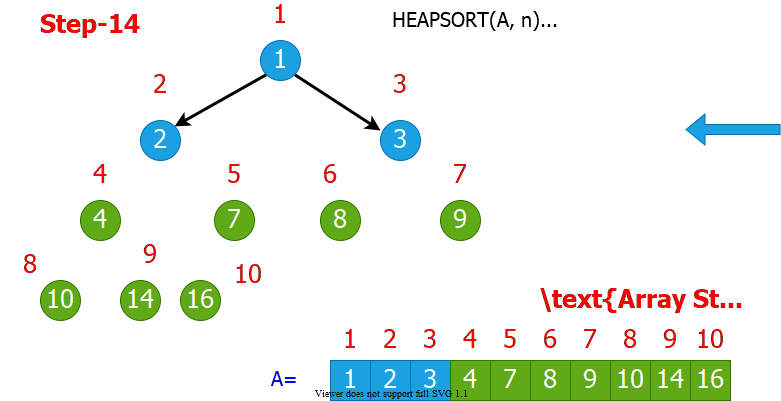
bg right:70% w:800px

## Heapsort Algorithm Example (Step-13)



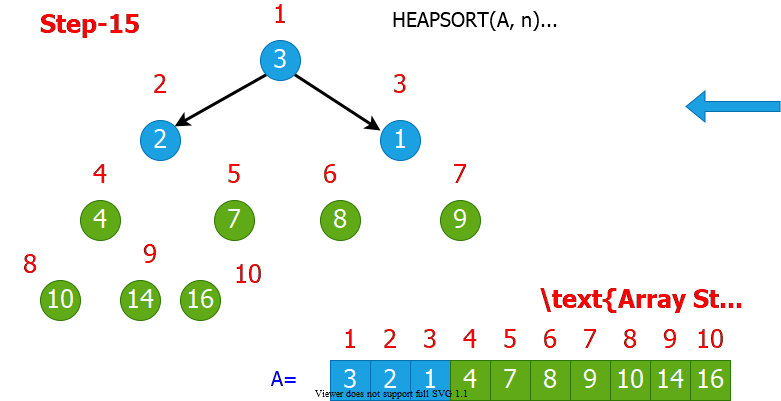
bg right:70% w:800px

## Heapsort Algorithm Example (Step-14)



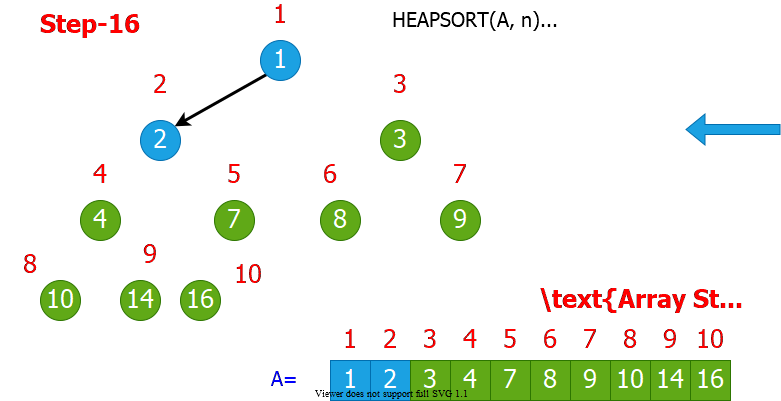
bg right:70% w:800px

## Heapsort Algorithm Example (Step-15)



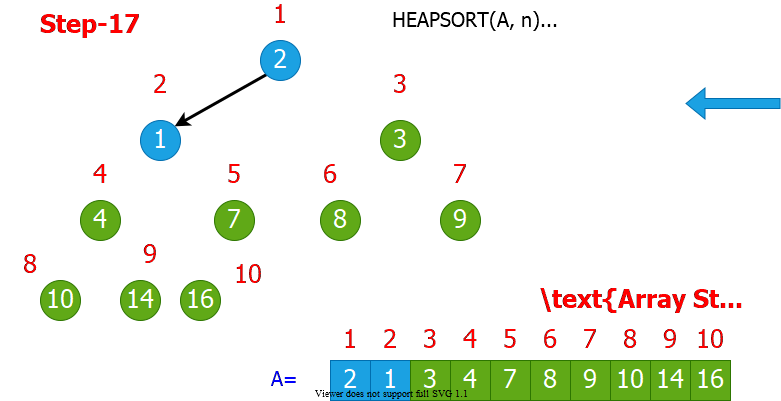
bg right:70% w:800px

## Heapsort Algorithm Example (Step-16)



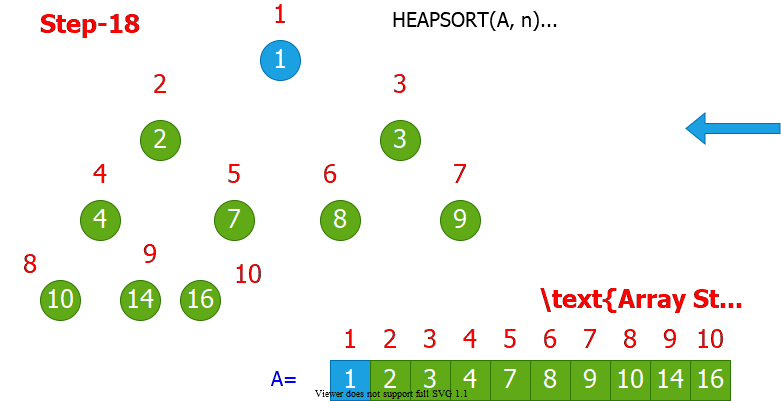
bg right:70% w:800px

## Heapsort Algorithm Example (Step-17)



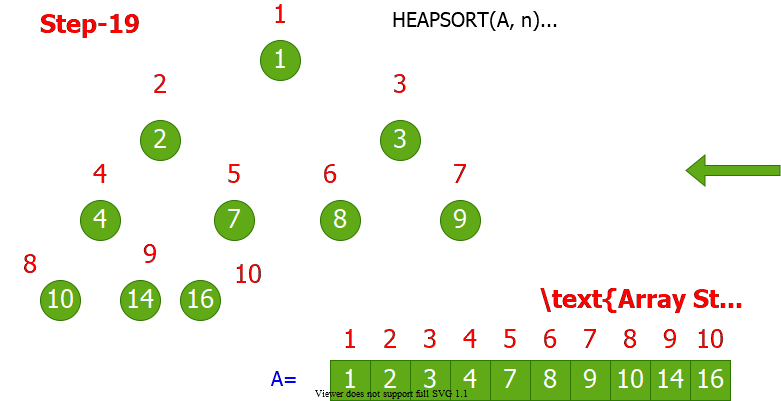
bg right:70% w:800px

## Heapsort Algorithm Example (Step-18)



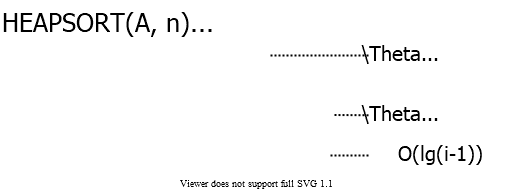
bg right:70% w:800px

## Heapsort Algorithm Example (Step-19)



bg right:70% w:800px

## Heapsort Algorithm: Runtime Analysis



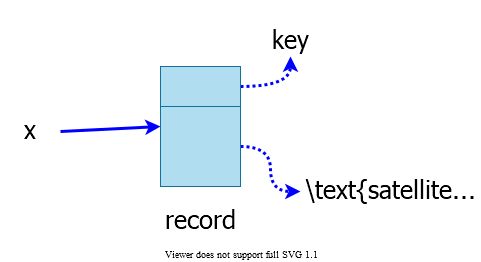
center height:250px

## Heapsort - Notes

* **Heapsort** is a very good algorithm but, a good implementation of **quicksort** always **beats** heapsort **in practice**
* However, **heap data structure** has many popular applications, and it can be efficiently used for implementing **priority queues**

## Data structures for **Dynamic Sets**

* Consider sets of records having **key** and **satellite** data



center height:300px

## Operations on **Dynamic Sets**

* **Queries:** Simply return info;
  + (Query) return with the **largest/smallest**
  + (Query) return with
  + (Query) return which is the next **larger/smaller** element after
* **Modifying operations:** Change the set
  + (Modifying)
  + (Modifying)
  + (Modifying) return and delete with the largest/smallest
* Different data structures support/optimize different operations

## Priority Queues (PQ)

* Supports

## Priority Queues (PQ)

* **One application:** Schedule jobs on a shared resource
  + **PQ** keeps track of jobs and their relative priorities
  + When a job is finished or interrupted, highest priority job is selected from those pending using
  + A new job can be added at any time using

## Priority Queues (PQ)

* **Another application:** Event-driven simulation
  + Events to be simulated are the items in the **PQ**
  + Each event is associated with a time of occurrence which serves as a
  + Simulation of an event can cause other events to be simulated in the future
  + Use at each step to choose the next event to simulate
  + As new events are produced insert them into the **PQ** using

## Implementation of **Priority Queue**

* **Sorted linked list:** Simplest implementation
  + - time
    - Scan the list to find place and splice in the new item
    - time
    - Take the first element
  + **Fast** extraction but **slow** insertion.

## Implementation of **Priority Queue**

* **Unsorted linked list:** Simplest implementation
  + - time
    - Put the new item at front
    - time
    - Scan the whole list
  + **Fast** insertion but **slow** extraction.
* Sorted linked list is better on the average
  + **Sorted list:** on the average, scans element per insertion
  + **Unsorted list:** always scans element at each extraction

## Heap Implementation of **PQ**

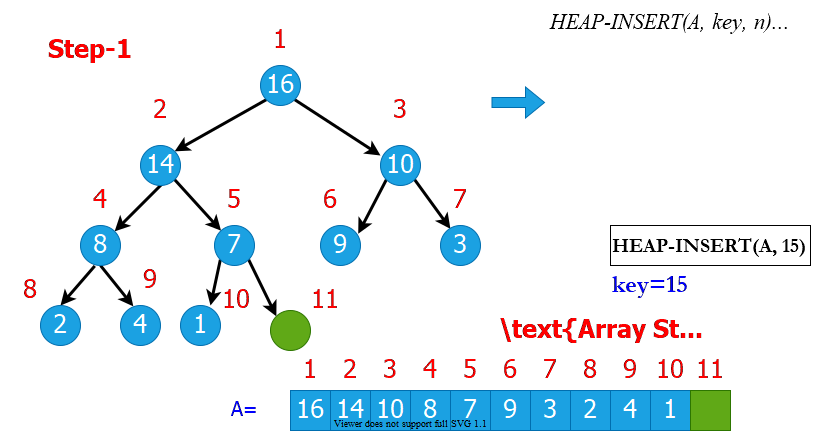
* and are both
  + good compromise between fast insertion but slow extraction and vice versa
* : already discussed
* : Insertion is like that of Insertion-Sort.

HEAP-INSERT(A, key, n)  
 n = n+1  
 i=n   
 while i>1 and A[floor(i/2)] < key do  
 A[i]=A[floor(i/2)]   
 i= floor(i/2)  
 A[i]=key

## Heap Implementation of **PQ**

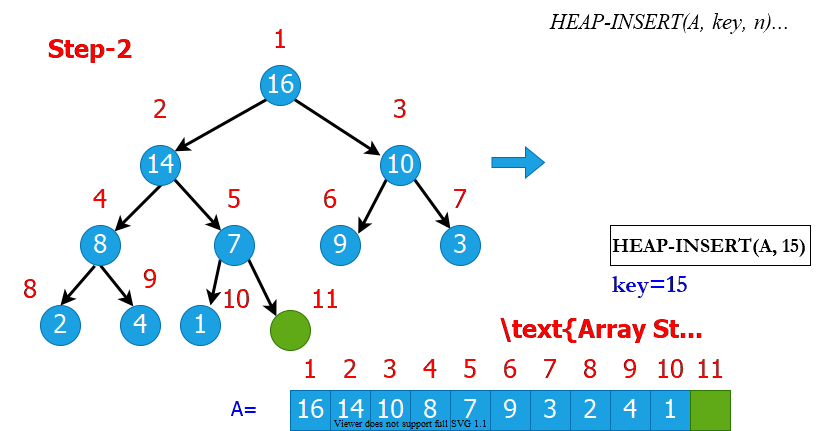
* Traverses nodes, as does but makes fewer comparisons and assignments
  + : compares parent with both children
  + : with only one

## HEAP-INSERT Example (Step-1)



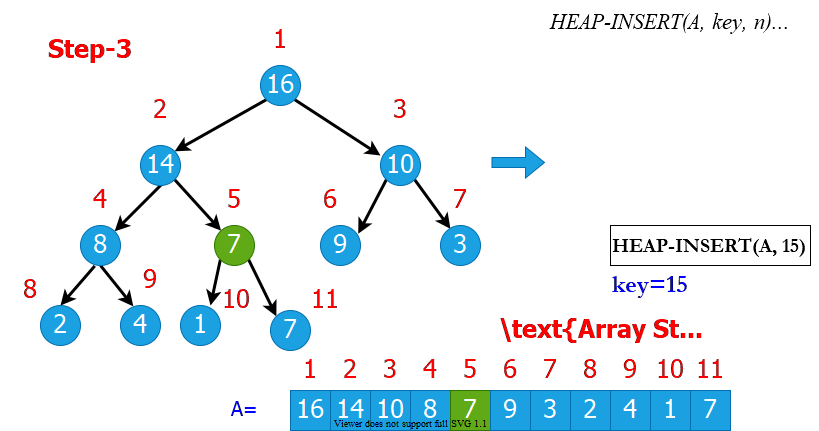
bg right:70% w:800px

## HEAP-INSERT Example (Step-2)



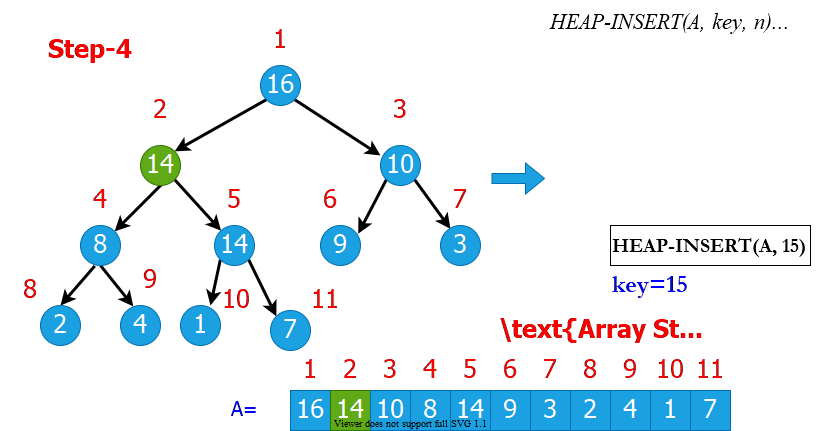
bg right:70% w:800px

## HEAP-INSERT Example (Step-3)



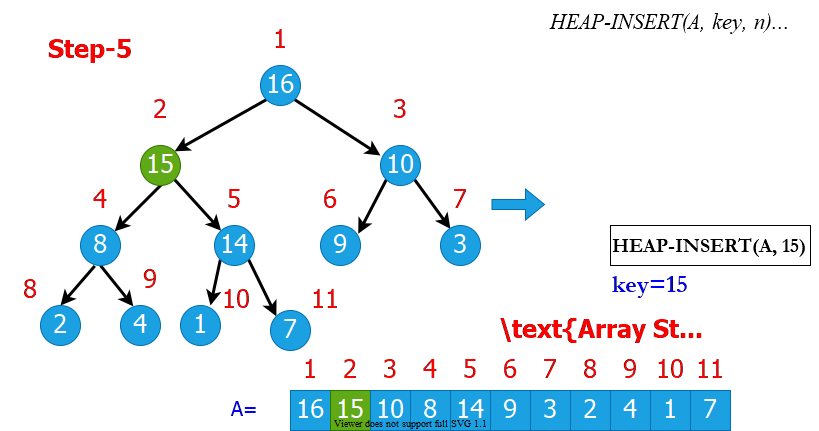
bg right:70% w:800px

## HEAP-INSERT Example (Step-4)



bg right:70% w:800px

## HEAP-INSERT Example (Step-5)



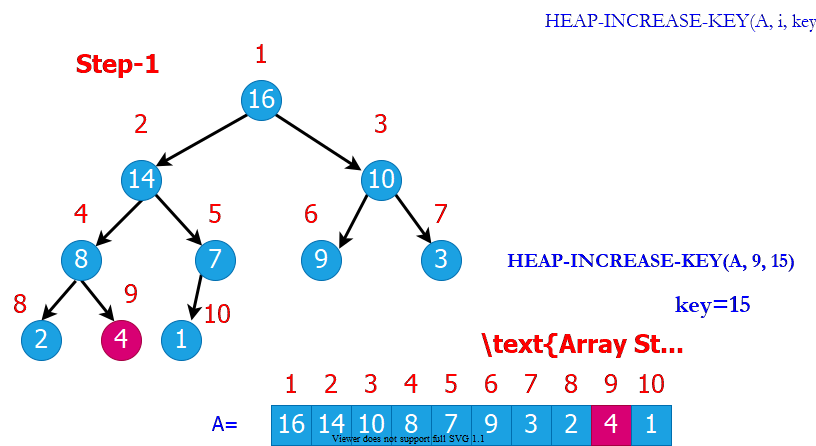
bg right:70% w:800px

## Heap Increase Key

* Key value of element of heap is increased from to

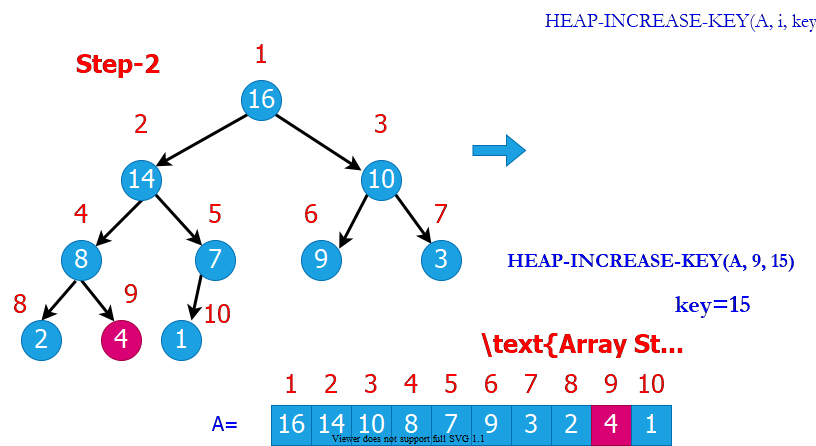
HEAP-INCREASE-KEY(A, i, key)  
  
 if key < A[i] then  
 return error  
  
 while i > 1 and A[floor(i/2)] < key do  
 A[i] = A[floor(i/2)]   
 i = floor(i/2)  
  
 A[i] = key

## HEAP-INCREASE-KEY Example (Step-1)



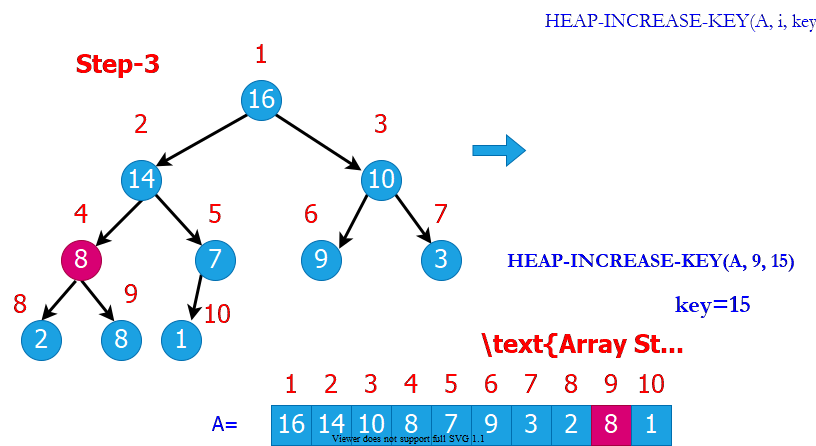
bg right:70% w:800px

## HEAP-INCREASE-KEY Example (Step-2)



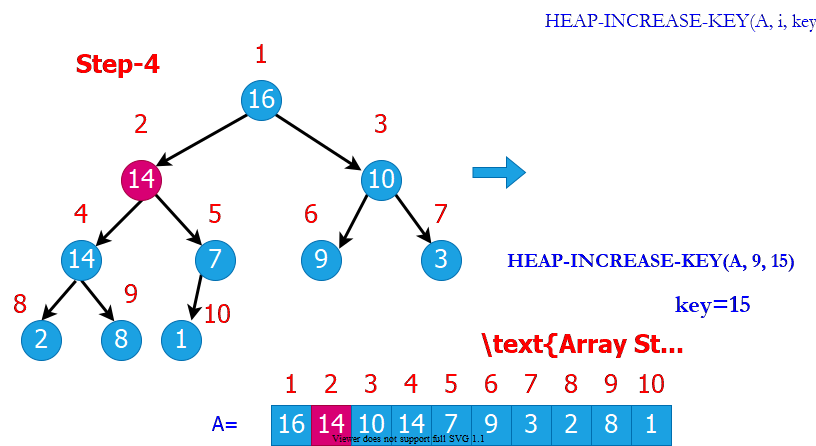
bg right:70% w:800px

## HEAP-INCREASE-KEY Example (Step-3)



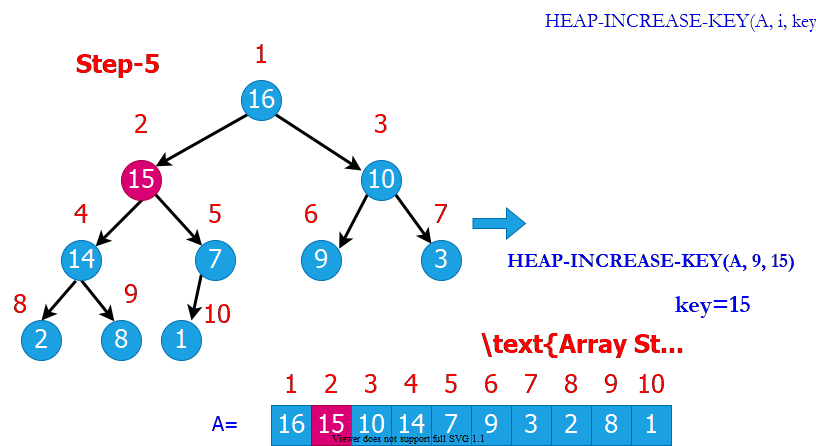
bg right:70% w:800px

## HEAP-INCREASE-KEY Example (Step-4)



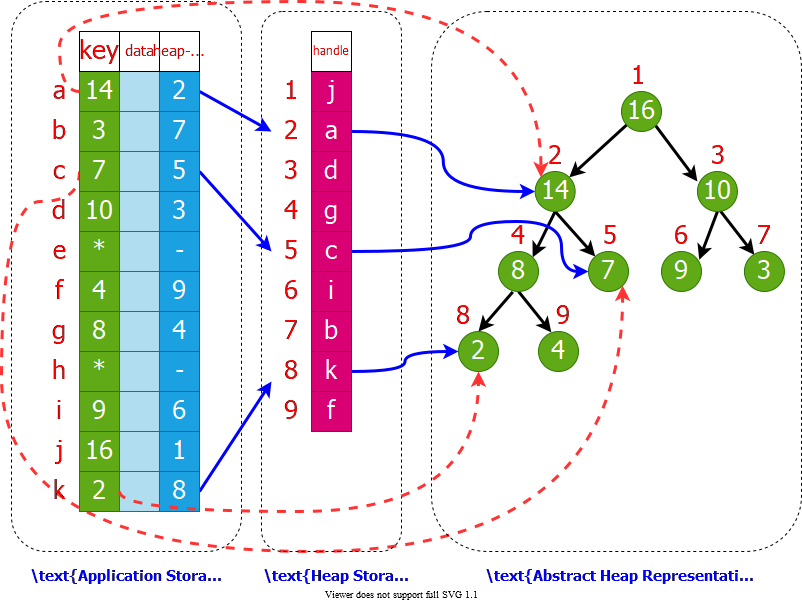
bg right:70% w:800px

## HEAP-INCREASE-KEY Example (Step-5)



bg right:70% w:800px

## Heap Implementation of Priority Queue (PQ)



bg right:70% w:800px

## Summary: **Max Heap**

* **Heapify(A, i)**
  + Works when both child subtrees of node i are heaps
  + “*Floats down*” node i to satisfy the heap property
  + Runtime:
* **Max(A, n)**
  + Returns the max element of the heap (no modification)
  + Runtime:
* **Extract-Max(A, n)**
  + Returns and removes the max element of the heap
  + Fills the gap in with , then calls **Heapify(A,1)**
  + Runtime:

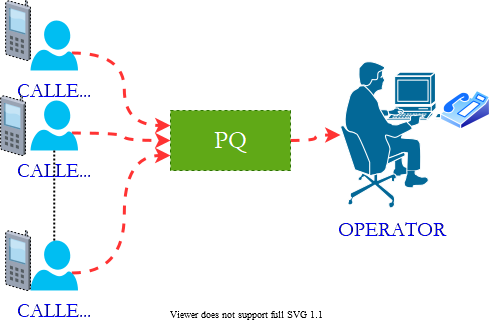
## Summary: **Max Heap**

* **Build-Heap(A, n)**
  + Given an arbitrary array, builds a heap from scratch
  + Runtime:
* **Min(A, n)**
  + How to return the min element in a max-heap?
  + Worst case runtime:
    - because ~half of the heap elements are leaf nodes
  + Instead, use a min-heap for efficient min operations
* **Search(A, x)**
  + For an arbitrary value, the worst-case runtime:
  + Use a sorted array instead for efficient search operations

## Summary: **Max Heap**

* **Increase-Key(A, i, x)**
  + Increase the key of node (from to )
  + “*Float up*” until heap property is satisfied
  + Runtime:
* **Decrease-Key(A, i, x)**
  + Decrease the key of node (from to )
  + Call **Heapify(A, i)**
  + Runtime:

## **Phone Operator** Problem

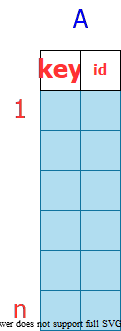


bg right:40% w:500px

* A phone operator answering  **phones**
* Each phone has  **people waiting** in line for their calls to be answered.
* Phone operator needs to answer the phone with the largest number of people waiting in line.
* New calls come continuously, and some people hang up after waiting.

## **Phone Operator** Solution

* **Step 1:** Define the following array:
* : the ith element in heap
* : the index of the corresponding phone
* : of people waiting in line for phone with index



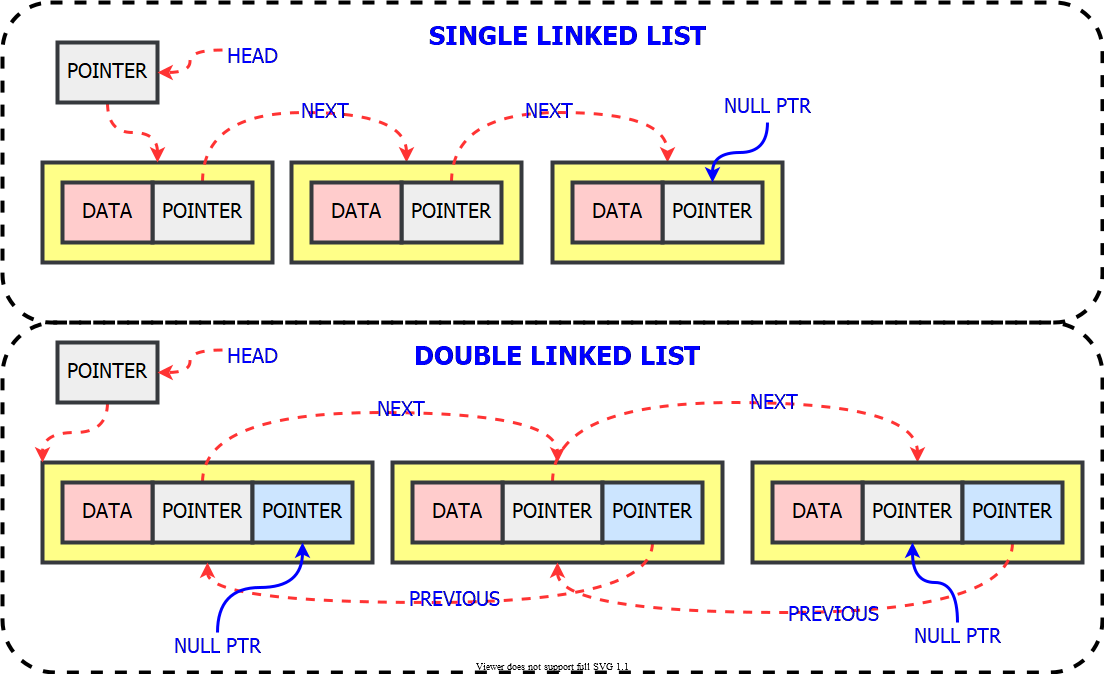
bg right:30% w:200px

## **Phone Operator** Solution

* **Step 2:**
  + **Execution:**
    - When the operator wants to answer a phone:
      * + answer phone with index
      * When a new call comes in to phone i:
      * When a call drops from phone i:

## Linked Lists

* Like arrays, Linked List is a linear data structure.
* Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



bg right:60% w:730px

## Linked Lists - C Definition

* C
* // A linked list node  
  struct Node {  
   int data;  
   struct Node\* next;  
  };

## Linked Lists - Cpp Definition

* Cpp
* class Node {  
  public:  
   int data;  
   Node\* next;  
  };

## Linked Lists - Java Definition

* Java
* class LinkedList {  
   Node head; // head of the list  
    
   /\* Linked list Node\*/  
   class Node {  
   int data;  
   Node next;  
    
   // Constructor to create a new node  
   // Next is by default initialized  
   // as null  
   Node(int d) { data = d; }  
   }  
  }

## Linked Lists - Csharp Definition

* Csharp
* class LinkedList {  
   // The first node(head) of the linked list  
   // Will be an object of type Node (null by default)  
   Node head;  
    
   class Node {  
   int data;  
   Node next;  
    
   // Constructor to create a new node  
   Node(int d) { data = d; }  
   }  
  }

## Priority Queue using **Linked List** Methods

* Implement Priority Queue using Linked Lists.
  + **push():** This function is used to insert a new data into the queue.
  + **pop():** This function removes the element with the highest priority from the queue.
  + **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

## Priority Queue using **Linked List** Algorithm

PUSH(HEAD, DATA, PRIORITY)  
 Create NEW.Data = DATA & NEW.Priority = PRIORITY  
 If HEAD.priority < NEW.Priority   
 NEW -> NEXT = HEAD  
 HEAD = NEW   
 Else  
 Set TEMP to head of the list   
 Endif  
  
 WHILE TEMP -> NEXT != NULL and TEMP -> NEXT ->PRIORITY > PRIORITY THEN  
 TEMP = TEMP -> NEXT   
 ENDWHILE  
  
 NEW -> NEXT = TEMP -> NEXT   
 TEMP -> NEXT = NEW

## Priority Queue using **Linked List** Algorithm

POP(HEAD)  
//Set the head of the list to the next node in the list.  
HEAD = HEAD -> NEXT.  
Free the node at the head of the list

PEEK(HEAD):   
Return HEAD -> DATA

## Priority Queue using **Linked List** Notes

* LinkedList is already sorted.
* Time Complexities and Comparison with Binary Heap

|  | peek() | push() | pop() |
| --- | --- | --- | --- |
| Linked List |  |  |  |
| Binary Heap |  |  |  |

## Sorting in Linear Time

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)
* [Insertion Sort - GeeksforGeeks](https://www.geeksforgeeks.org/insertion-sort/)
* [Priority Queue Using Linked List - GeeksforGeeks](https://www.geeksforgeeks.org/priority-queue-using-linked-list/)
* [Priority Queue Using Linked List - JavatPoint](https://www.javatpoint.com/priority-queue-using-linked-list)
* [NIST Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)