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)

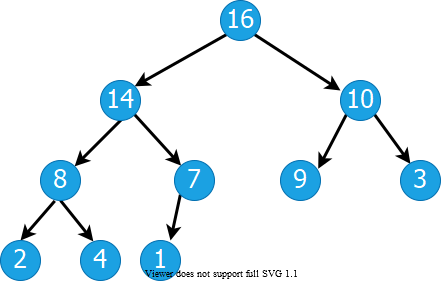
* Heap Sort
* Priority Queues
* Linked Lists
* Radix Sort
* Counting Sort

## 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)

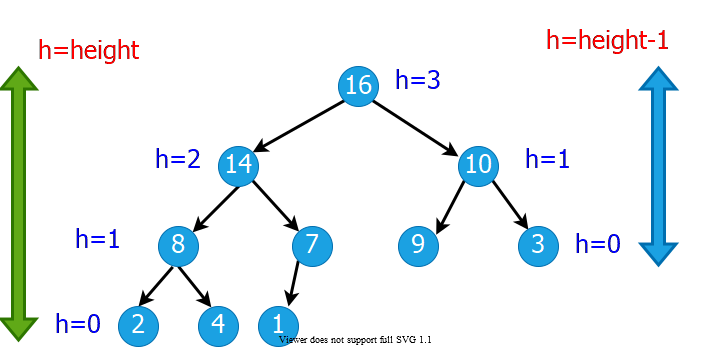
* Nearly complete binary tree
  + Completely filled on all levels except possibly the lowest level



alt:“alt” height:350px center

## Heap Data Structure (2)

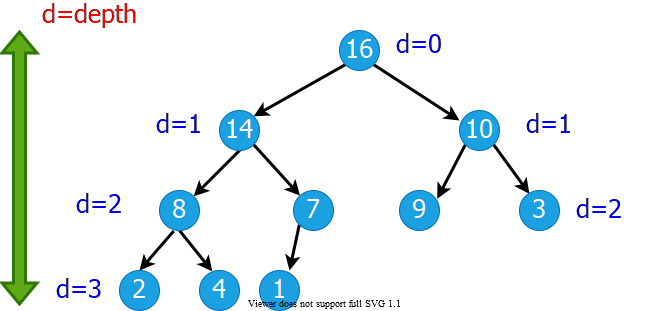
* **Height of node i:** Length of the longest simple downward path from **i** to a **leaf**
* **Height of the tree:** height of the **root**



alt:“alt” height:400px center

## Heap Data Structures (3)

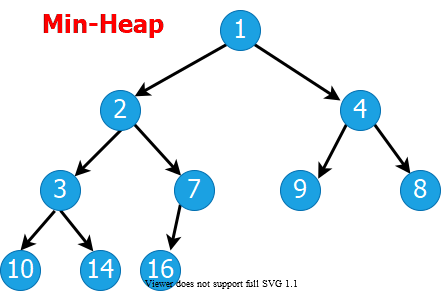
* **Depth of node i:** Length of the simple downward path from the **root** to node **i**



alt:“alt” height:350px center

## Heap Property: Min-Heap

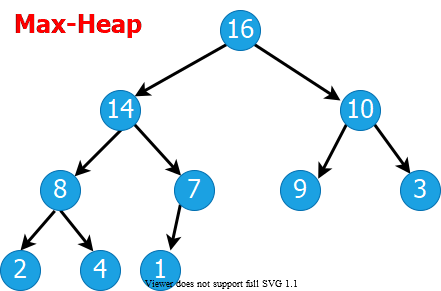
* The **smallest** element in any subtree is the **root** element in a **min-heap**
* **Min heap:** For every node **i** other than **root**,
  + Parent node is always smaller than the child nodes



alt:“alt” height:350px center

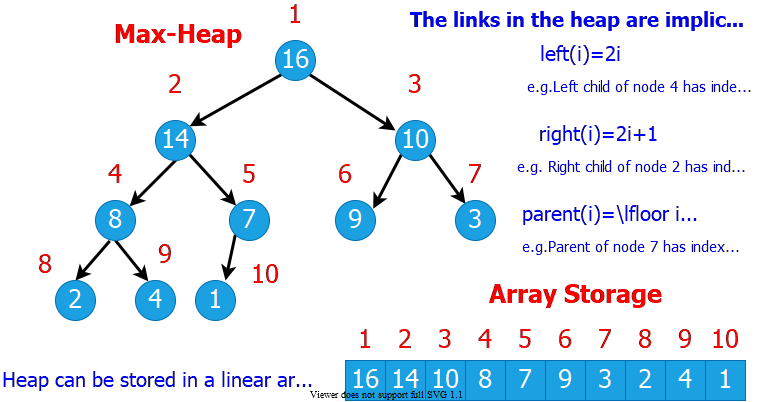
## Heap Property: Max-Heap

* 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



alt:“alt” height:350px center

## 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

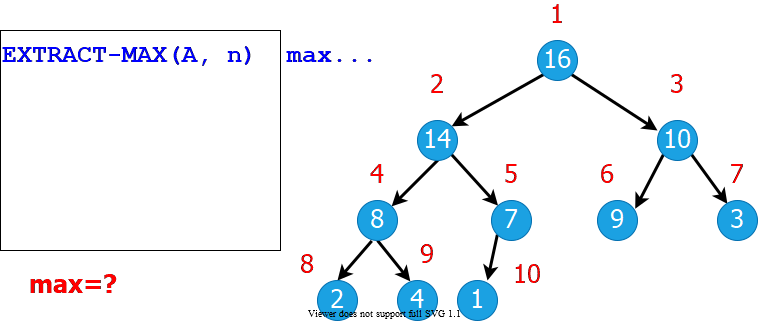
## 

## 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

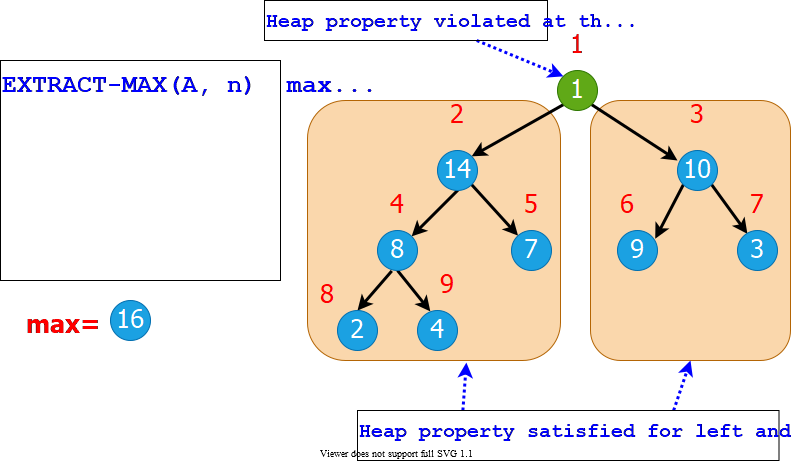
## Heap Operations : EXTRACT-MAX (2)

* 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

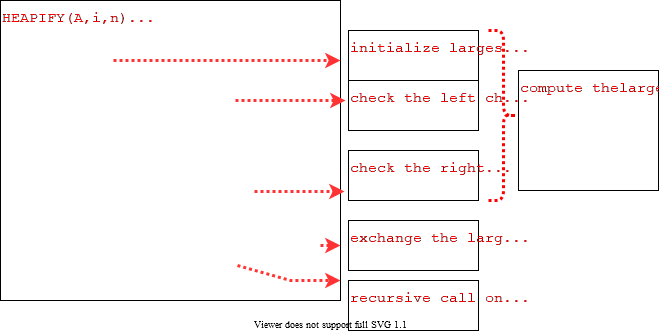
## Heap Operations: HEAPIFY (2)

* 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)

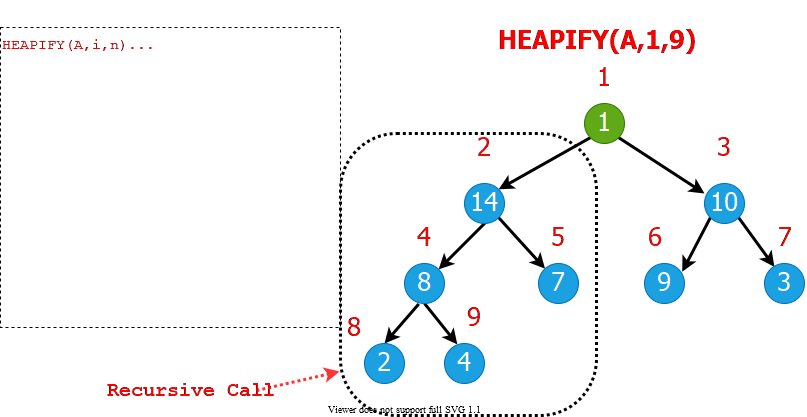
HEAPIFY(A, i, n)  
 largest = i   
   
 if 2i <= n and A[2i] > A[i] then   
 largest = 2i;  
 endif  
   
 if 2i+1 <= n and A[2i+1] > A[largest] then   
 largest = 2i+1;  
 endif  
  
 if largest != i then  
 exchange A[i] with A[largest];  
 HEAPIFY(A, largest, n);  
 endif

## Heap Operations: HEAPIFY (3)



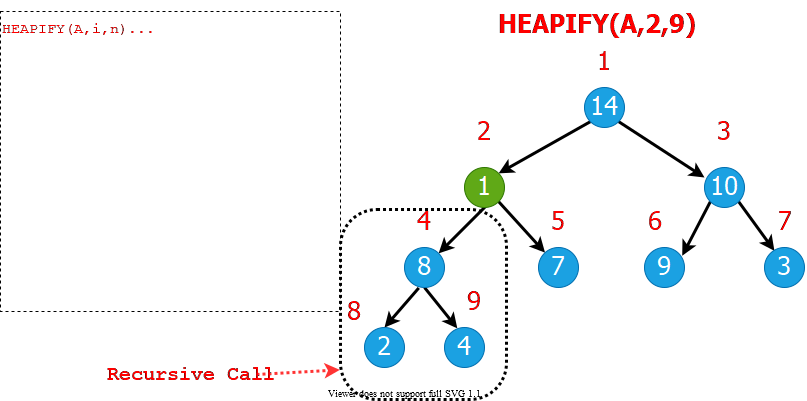
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (4)



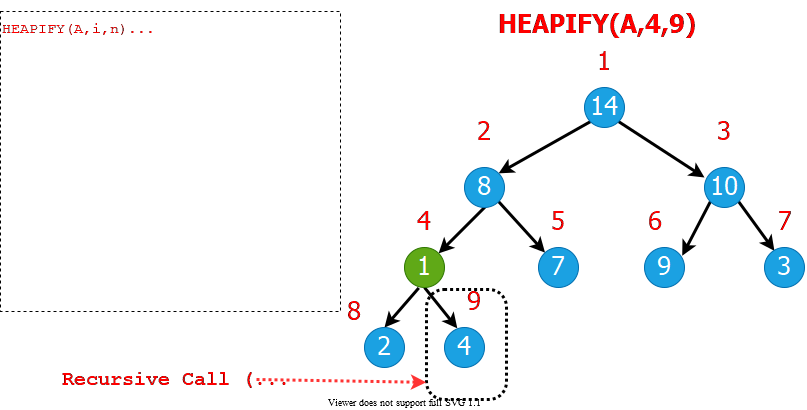
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (5)



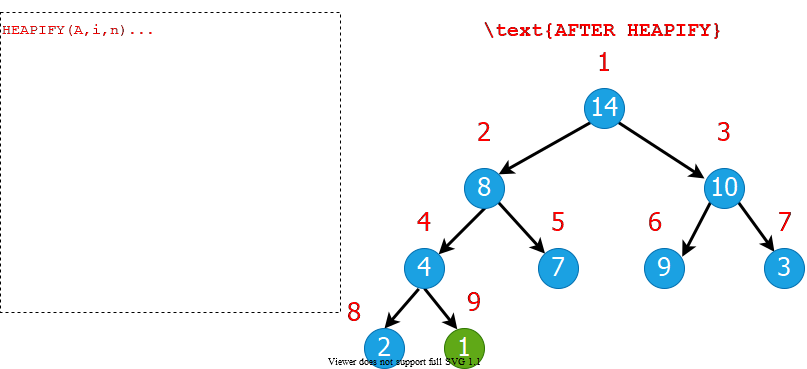
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (6)



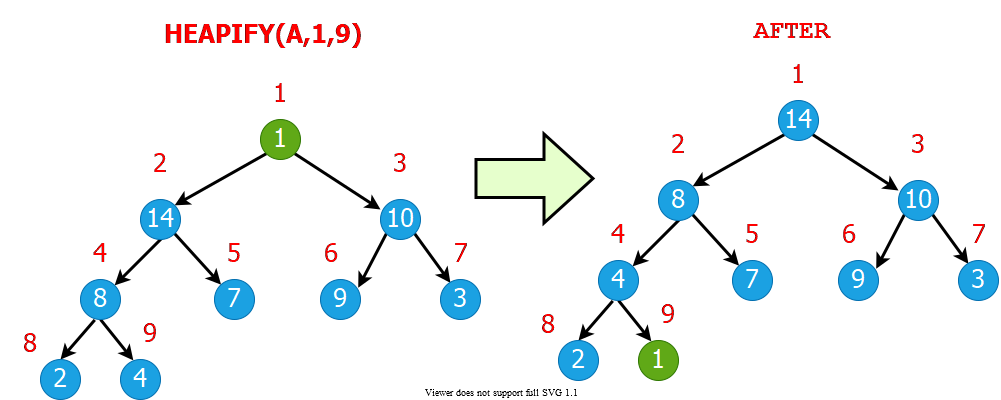
alt:“alt” height:500px center

## Heap Operations: HEAPIFY (7)



alt:“alt” height:500px center

## Heap Operations: HEAPIFY (8)



alt:“alt” height:500px center

## Intuitive Analysis of HEAPIFY

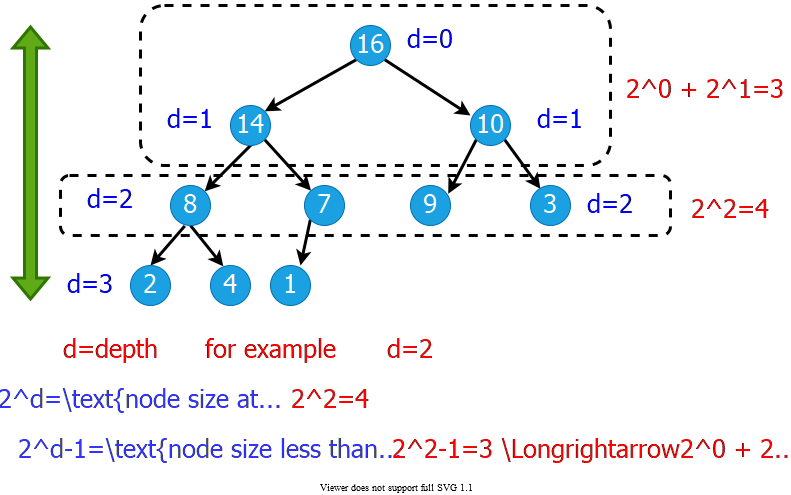
* Consider
  + let be the height of node
  + at most recursion levels
    - Constant work at each level:
  + Therefore
* Heap is almost-complete binary tree
* Thus

## 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

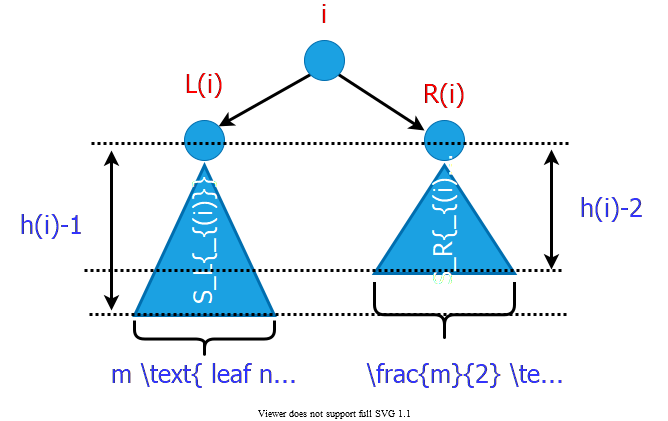
* For a complete binary tree:
  + of nodes at depth :
  + of nodes with depths less than :



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



alt:“alt” height:350px center

## 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**

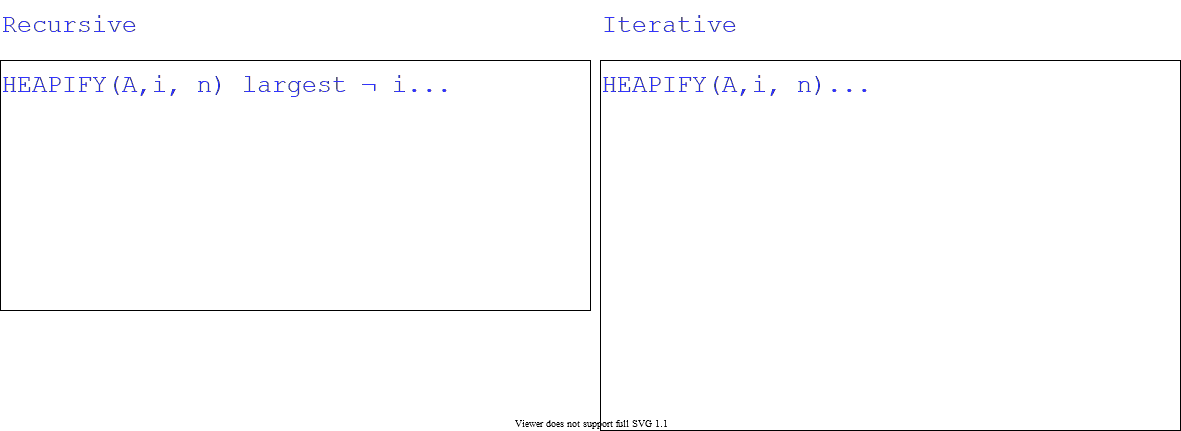
HEAPIFY(A, i, n)  
 largest = i   
  
 if 2i <= n and A[2i] > A[i] then   
 largest = 2i  
  
 if 2i+1 <= n and A[2i+1] > A[largest] then   
 largest = 2i+1  
  
 if largest != i then  
 exchange A[i] with A[largest]  
 HEAPIFY(A, largest, n)

## Heap Operations: HEAPIFY (2)

**Iterative**

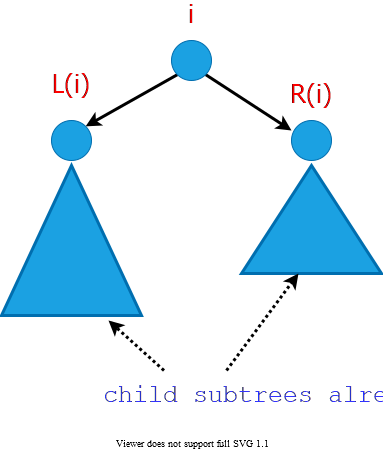
HEAPIFY(A, i, n)  
 j = i  
 while(true) do  
 largest = j   
  
 if 2j <= n and A[2j] > A[j] then   
 largest = 2j  
  
 if 2j+1 <= n and A[2j+1] > A[largest] then   
 largest = 2j+1  
  
 if largest != j then  
 exchange A[j] with A[largest]  
 j = largest  
 else return

## Heap Operations: HEAPIFY (3)



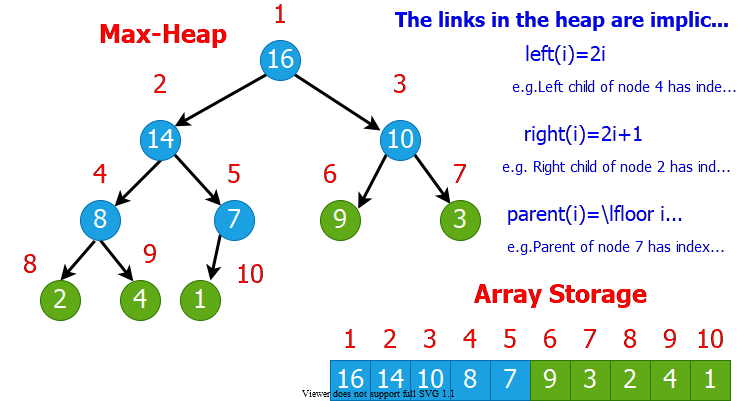
alt:“alt” height:500px center

## Heap Operations: Building Heap

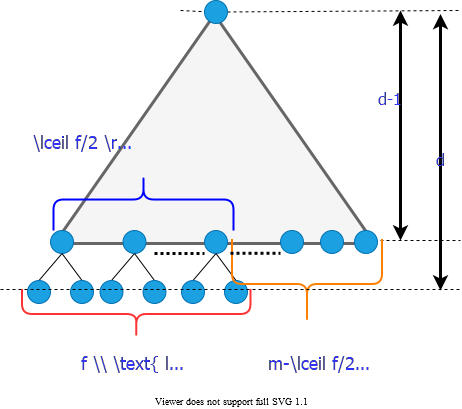
* Given an arbitrary array, how to build a heap from scratch?
* **Basic idea:** Call on each node bottom up
  + Start from the leaves (which trivially satisfy the heap property)
  + Process nodes in bottom up order.
  + When is called on node , the subtrees connected to the and subtrees already satisfy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

## Storage of the leaves

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



bg right w:600



bg right w:600

* asdasdsdas
* asdsadsa
* asdas

## 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/)
* [NIST Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)

TODO