Algorithms
Chapter 5: Transform and conquer

GATE CS Lectures
by Monalisa

MonalisaCS

Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

MonalisaCS

- Chapter 5: <u>Transform and conquer</u>
- Heaps and Heap sort,
- Balanced Search Trees.
-

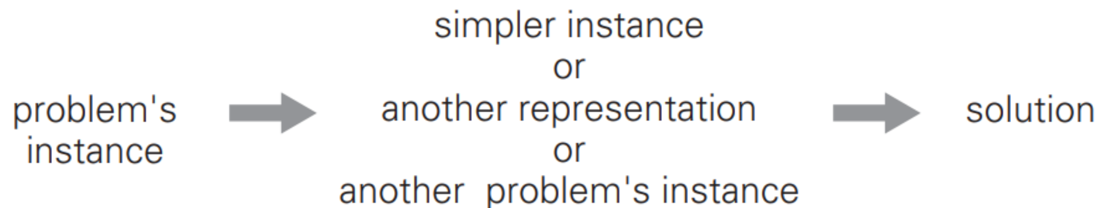
MonalisaCS

https://monalisacs.com/

Transform-and-Conquer

- This design methods based on the idea of transformation.
- These methods work as two-stage procedures.
- First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution.
- Then, in the second or conquering stage, it is solved.
- There are three major variations of this idea that differ by what we transform a given instance:
- Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
- Transformation to a different representation of the same instance—we call it ***representation change***.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

MonalisaCS

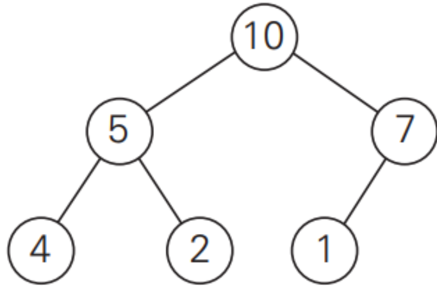


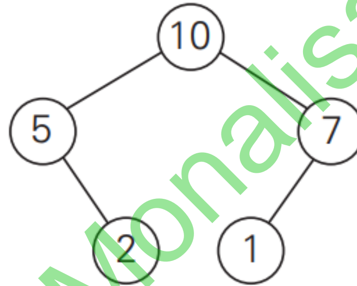
https://www.youtube.com/@MonalisaCS

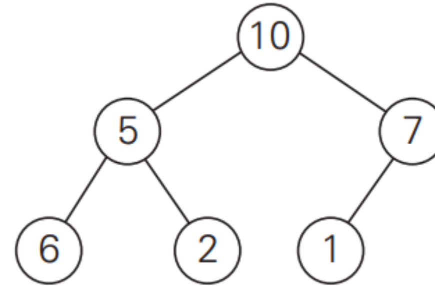
Heaps and Heapsort

https://monalisacs.com/

- **DEFINITION** A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:
- **1.** The ***shape property***—the binary tree is ***essentially complete*** , i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
- **2.** The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children.







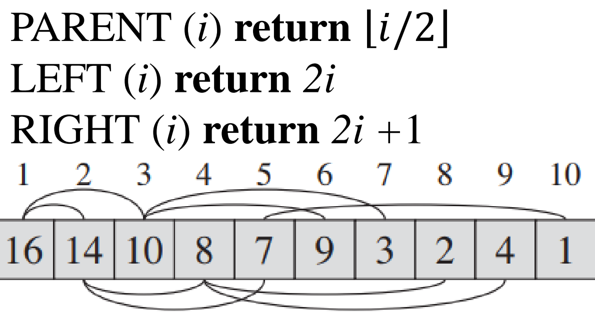
MonalisaCS

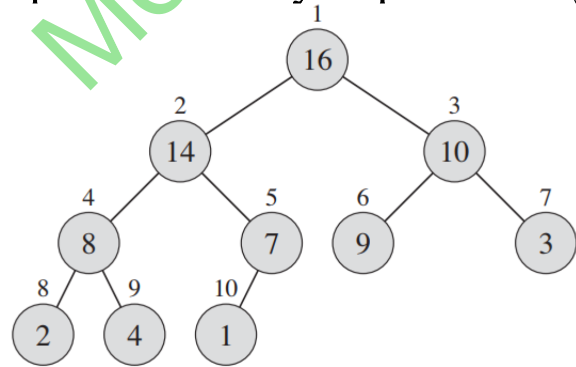
- The first tree is a heap.
- The second one is not a heap, because the tree's shape property is violated.
- The third one is not a heap, because the parental dominance fails for the node 5.
- Note that key values in a heap are ordered top down, i.e., a sequence of values on any path from the root to a leaf is decreasing.

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **Heaps properties**
- **1.** There exists exactly one essentially complete binary tree with n nodes. height $=\lfloor \log_2 n \rfloor$.
- **2.** The root of a heap always contains its largest element.
- **3.** A node of a heap considered with all its descendants is also a heap.
- **4.** A heap can be implemented as an array by recording its elements in the topdown, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ unused. In such a representation,
- **a.** the parental node keys will be in the first $n/2$ positions of the array, while the leaf keys will occupy the last $n/2$ positions,
- **b.** the children of a key in the array's parental position i $(1 \le i \le n/2)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $i/2$.
- PARENT (i) **return** $\lfloor i/2 \rfloor$
- LEFT (i) **return** $2i$
- RIGHT (i) **return** $2i + 1$





MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- There are two kinds of binary heaps: max-heaps and min-heaps.
- In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root, $A[\textbf{Parent}\ (i)] \geq A[i]$
- Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
- A ***min-heap*** is organized in the opposite way, the ***min-heap property*** is that for every node i other than the root, $A[\textbf{Parent}\ (i)] \leq A[i]$
- The smallest element in a min-heap is at the root.
- For the heapsort algorithm , we use max-heaps , Min-heaps commonly implement priority queues.
- The MAX-HEAPIFY procedure, runs in O(lg n) time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, runs in linear time, produces a maxheap from an unordered input array.
- The HEAPSORT procedure, runs in O(n lg n) time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE -KEY, and HEAP-MAXIMUM procedures, run in O(lg n) time.

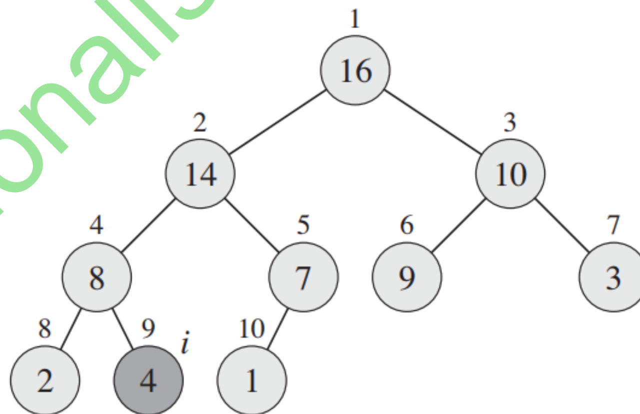
MonalisaCS

https://www.youtube.com/@MonalisaCS

- **Maintaining the heap property**
- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.
- Its inputs are an array A and an index i into the array.
- When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are maxheaps, but that A[i] might be smaller than its children, thus violating the max-heap property.
- MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.
- MAX-HEAPIFY(A, i)

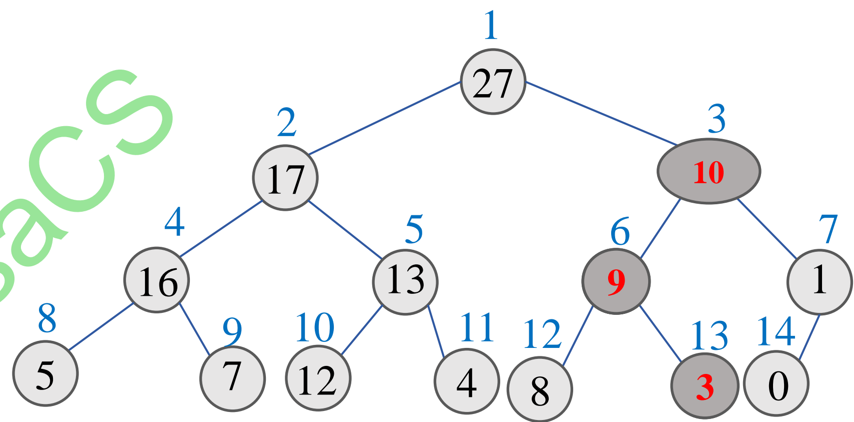
1. *l*=LEFT(i)
2. *r*=RIGHT(i)
3. **if** *l* ≤A.*heap-size* and A[*l*] > A[*i*]
4. *largest* = *l*
5. **else** *largest* =*i*
6. **if** *r*≤ A.*heap-size* and A[r] > A[*largest*]
7. *largest* = *r*
8. **if** *largest* ≠ *i*
9. exchange A[i] with A[*largest*]
10. MAX-HEAPIFY(A, *largest*)

- **Example:** The action of MAX-HEAPIFY(A,2), where A.*heap-size*=10



MonalisaCS

- Running time of MAX-HEAPIFY by the recurrence : $T(n) \leq T(2n/3) + \Theta(1)$.
- The solution to this recurrence, by case 2 of the master theorem ,is $T(n)=O(\lg n)$.
- Alternatively, the running time of MAXHEAPIFY on a node of height h as $O(h)$. $h=lg\ n$
- **Exercises :**illustrate the operation of MAX-HEAPIFY(A,3) on the array A= {27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0}.
- MAX-HEAPIFY(A,3)
- *l*=LEFT(i)   //6
- *r*=RIGHT(i)   //7
- **if** *l* ≤A.*heap-size* and A[*l*] > A[*i*]   //10>3
- *largest* = *l* //6
- exchange A[i] with A[*largest*]//A[3] with A[6]
- MAX-HEAPIFY(A, *6*)
- *l*=LEFT(i)   //12
- *r*=RIGHT(i)   //13
- **if** *l* ≤A.*heap-size* and A[*l*] > A[*i*]   //8>3
- *largest* = *l* //12
- **if** *r*≤ A:*heap-size* and A[r] > A[*largest*]//9>8
- *largest* = *r*   //13
- exchange A[i] with A[*largest*]   //A[6] with A[13]

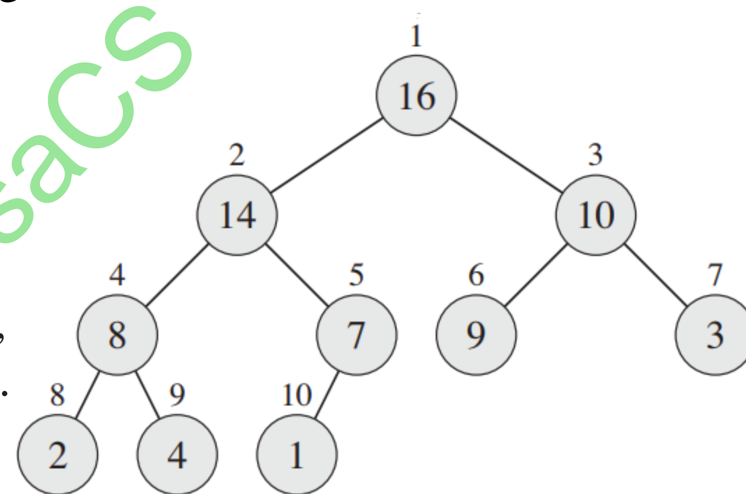


MonalisaCS

- **Building a heap**
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1…n], where n=A.*length*, into a max-heap.
- The elements in the subarray A[(⌊$n/2$⌋+1).. n] are all leaves of the tree , so the procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.
- BUILD-MAX-HEAP(*A*)

1. A.*heap-size* = A.*length*
2. **for** *i* = ⌊A.*length*/2⌋ **downto** 1
3. MAX-HEAPIFY(A,*i*)

- Each call to MAX-HEAPIFY costs O(lg *n*) time,
- and BUILD MAX-HEAP makes O(*n*) such calls.
- Thus, the running time is O(n lg n).
- Example of the action of BUILD-MAX-HEAP.



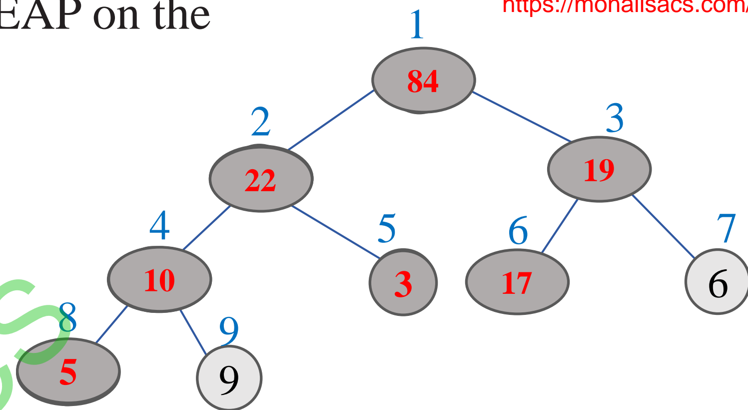
A	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7

MonalisaCS

https://monalisacs.com/

- **Exercises :** Illustrate the operation of BUILD-MAX-HEAP on the array A ={5, 3, 17, 10, 84, 19, 6, 22, 9}.
- BUILD-MAX-HEAP(*A*)

1. A.*heap-size* = A.*length* //9
2. **for** *i* = ⌊A.*length*/2⌋ **downto** 1 //4-1
3. MAX-HEAPIFY(A,*i*)



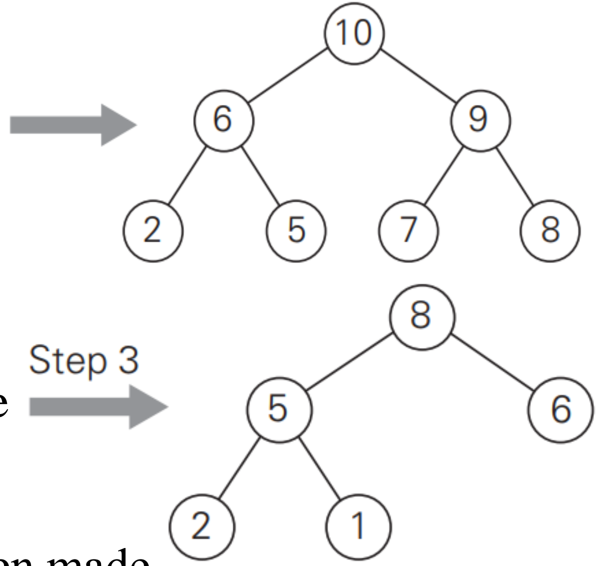
- There are two methods of constructing a heap.
- The first is the ***bottom-up heap construction*** algorithm.
- It initializes the essentially complete binary tree with *n* nodes by placing keys in the order given and then “heapifies” the tree as follows.
- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node.
- The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; the ***top-down heap construction*** algorithm.
- First, attach a new node with key *K* in it after the last leaf of the existing heap.
- Then sift *K* up to its appropriate place in the new heap as follows.
- Compare *K* with its parent’s key: if it is greater than or equal to *K,* stop ;
- Otherwise, swap these two keys and compare *K* with its new parent.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- This **insertion** operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is $\log_2 n$, The time efficiency of insertion is in $O(\log_2 n)$.
- **Example:** Inserting a key (10) into the heap constructed . The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).
- Deleting the root's key from a heap can be done with the following algorithm.
- **Maximum Key Deletion** from a heap
- **Step 1** Exchange the root's key with the last key of the heap.
- **Step 2** Decrease the heap's size by 1.
- **Step 3** "Heapify" the smaller tree by sifting root.
- **Example:** Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified"
- The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1.
- Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well .



MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **Heapsort**
- An interesting sorting algorithm discovered by J. W. J. Williams.
- This is a two-stage algorithm that works as follows.
- **Stage 1** (heap construction): Construct a heap for a given array.
- **Stage 2** (maximum deletions): Apply the root-deletion operation n - 1 times to the remaining heap.
- **The heapsort algorithm**
- The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A[1…n], where n =A.*length*.
- Since the maximum element of the array is stored at the root A[1], we can put it into its correct final position by exchanging it with A[n].
- Now discard node n from the heap—and we can do so by simply decrementing A.*heap-size*.
- All we need to do to restore the max-heap property, call MAX-HEAPIFY(A,1).
- The heapsort algorithm then repeats this process for the max-heap of size n-1 down to a heap of size 2.
- HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. **for** *i* = A.*length* **downto** 2
3. exchange A[1] with A[*i*]
4. A:*heap-size* = A:*heap-size*-1
5. MAX-HEAPIFY(A,1)

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- The HEAPSORT procedure takes time ***O(n* lg *n)***, since the call to BUILD-MAXHEAP takes time *O*(*n*) and each of the n- 1 calls to MAX-HEAPIFY takes time O(lg n).
- **Example:** Sorting the array 2, 9, 7, 6, 5, 8 by heapsort
- BUILD-MAX-HEAP(*A*)

1	2	3	4	5	6
2	9	7	6	5	8
2	9	**8**	6	5	**7**
9	**2**	8	6	5	7
9	**6**	8	**2**	5	7

- for *i* = 6
- exchange A[1] with A[6]

7	6	8	2	5	**9**

- A:*heap-size* = A:*heap-size*-1

7	6	8	2	5

- MAX-HEAPIFY(A,1)

8	6	**7**	2	5

- for *i* = 5
- exchange A[1] with A[5]

5	6	**7**	2	**8**

- A:*heap-size* = A:*heap-size*-1

5	6	7	2

- MAX-HEAPIFY(A,1)

7	6	**5**	2

- for *i* = 4
- exchange A[1] with A[4]

2	6	5	**7**

- A:*heap-size* = A:*heap-size*-1

2	6	5

- MAX-HEAPIFY(A,1)

6	**2**	5

- for *i* = 3
- exchange A[1] with A[3]

5	2	**6**

- A:*heap-size* = A:*heap-size*-1

5	2

- MAX-HEAPIFY(A,1)

5	2

- for *i* = 2
- exchange A[1] with A[2]

2	**5**

- A:*heap-size* = A:*heap-size*-1

2

MonalisaCS

https://www.youtube.com/@MonalisaCS

- Time complexity to find the minmum element in max-heap is $\Theta(n)$.
- We can't perform binary search ,because its not BST and heaps need not be in sorted order.
- So we need to perform linear search on leaves($\lceil n/2 \rceil$ elements) to find minimum element.
- In linear search $\lceil n/2 \rceil$-1 comparisons so time complexity $\Theta(n)$.
- Similar for finding maximum element in min-heap, $\Theta(n)$.
- The minimum number of comparisons required to find the maximum in the min-heap and minimum in max-heap is $\lceil n/2 \rceil$-1 .
- kth smallest element in a min-heap cannot go deeper than level k because the path from root to that level goes through k-1 smaller elements.
- kth largest element in a max-heap cannot go deeper than level k because the path from root to that level goes through k-1 larger elements.
- Applications of Heaps:
- Heap Sort: Heap Sort is one of the best sorting algorithms that use Binary Heap to sort an array in O(N log N) time.
- Priority Queue: A priority queue can be implemented by using a heap .
- Graph Algorithms: The heaps are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

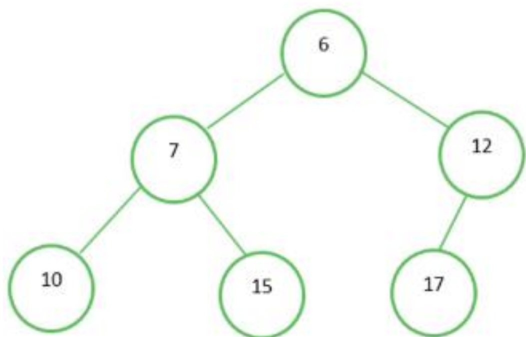
MonalisaCS

https://monalisacs.com/

Min Heap

- In a Min-Heap the key present at the parent node must be less than or equal to the keys present at all of its children.
- In a Min-Heap the minimum key element present at the root.
- A Min-Heap uses the ascending priority
- Find minimum O(1),maximum O(n)
- Insertion ,Deletion O(log n)

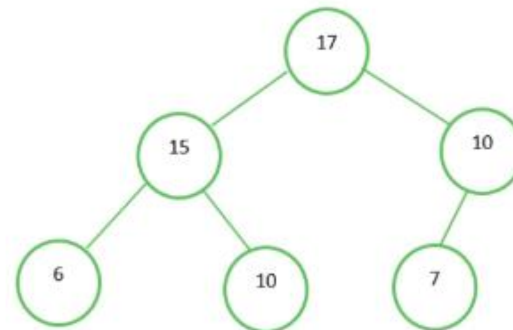
Min-Heap



Max Heap

- In a Max-Heap the key present at the parent node must be greater than or equal to the keys present at all of its children.
- In a Max-Heap the maximum key element present at the root.
- A Max-Heap uses the descending priority.
- Find maximum O(1),minimum O(n)
- Insertion ,Deletion O(log n)

Max-Heap



MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Balanced Search Trees

- It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.
- This transformation is an example of the ***representation-change technique***.
- What do we gain by such transformation compared to the straightforward array?
- We gain in the time efficiency of searching, insertion, and deletion, which are all in $\Theta(\log n)$, but only in the average case.
- In the worst case, these operations are in $\Theta(n)$ because the tree can degenerate into a severely unbalanced one with its height equal to n - 1.
- To avoids its worst-case degeneracy. They have come up with two approaches.
- The first approach is of the ***instance-simplification* variety**: an unbalanced binary search tree is transformed into a balanced one.
- Such trees are called ***self-balancing***. An ***AVL tree*** requires the difference between the heights of the left and right subtrees of every node never exceed 1.
- A ***red-black tree*** tolerates the height of one subtree being twice as large as the other subtree of the same node.
- If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by special transformations called ***rotations*** that restore the balance required.
- In this section, we will discuss only AVL trees.
- Information about other types of binary search trees that rebalance via rotations, including red-black trees and ***splay trees***, can be found in Data Structure.

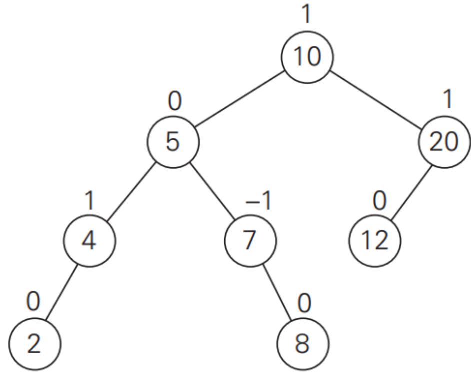
MonalisaCS

https://www.youtube.com/@MonalisaCS

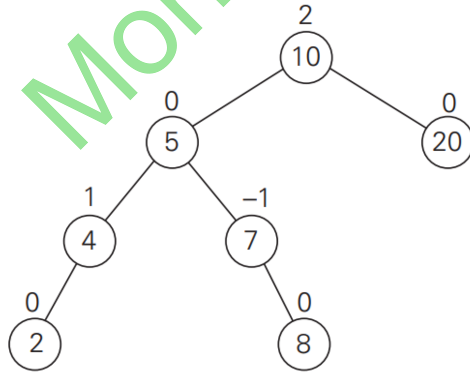
https://monalisacs.com/

- The second approach is of the ***representation-change variety***: allow more than one element in a node of a search tree.
- Such trees are ***2-3 trees***, ***2-3-4 trees***, and more general and important ***B-trees***.
- They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced.
- **AVL Trees**
- AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis , after whom this data structure is named.
- **DEFINITION** An ***AVL tree*** is a binary search tree in which the ***balance factor*** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1.
- Example : (a) AVL tree.(b) Binary search tree that is not an AVL tree.

MonalisaCS



(a)



(b)

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

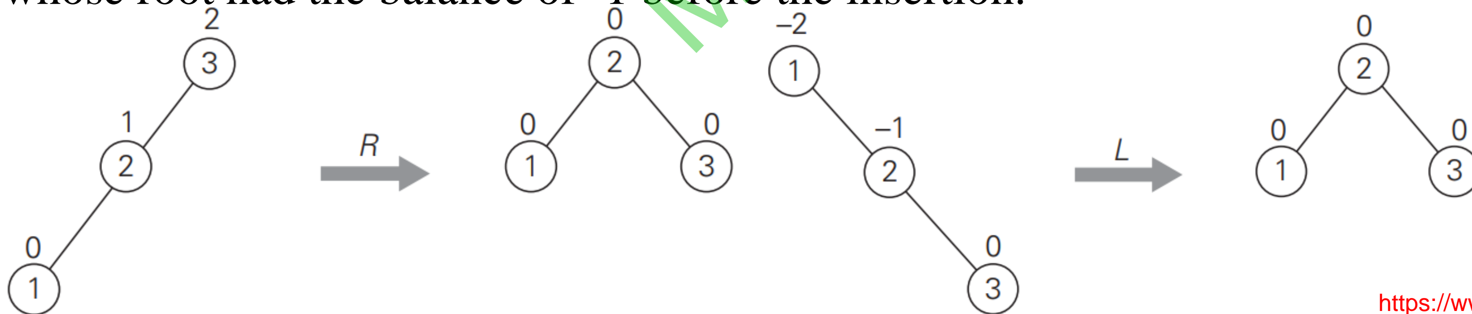
- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- A ***rotation*** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2.
- If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only four types of rotations; in fact, two of them are mirror images of the other two.

➢ The first rotation type is called the ***single right rotation***, or ***R-rotation***.

- This rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

➢ The symmetric ***single left rotation***, or ***L-rotation***, is the mirror image of the single *R*-rotation.

- It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion.

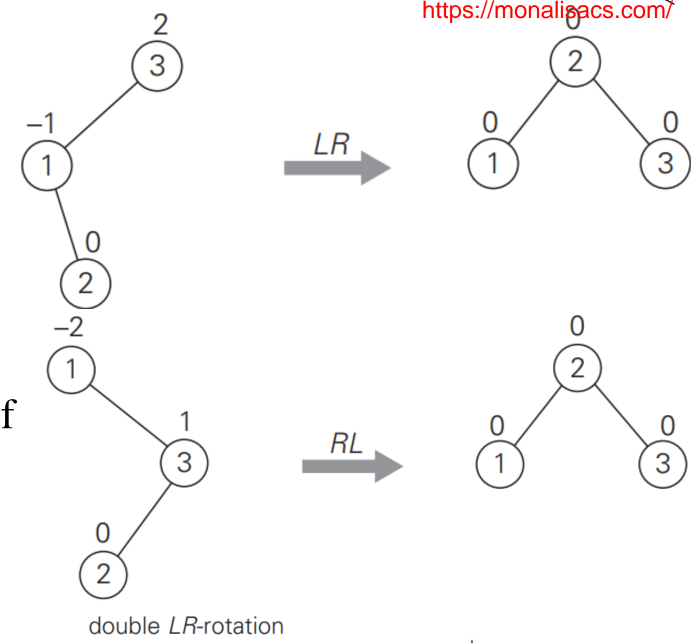


MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- The second rotation type is called the ***double left-right rotation*** (***LR-rotation***).
- It is a combination of two rotations: we perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r* .
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.
- The ***double right-left rotation*** (***RL-rotation***) is the mirror image of the double *LR*-rotation and is left for the exercises.
- General form of the *R-rotation* and *LR*-rotation in the AVL tree.
- A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.



single *R*-rotation

double *LR*-rotation

MonalisaCS

https://www.youtube.com/@MonalisaCS

- Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

L(5)

R(5)

LR(6)

RL(6)

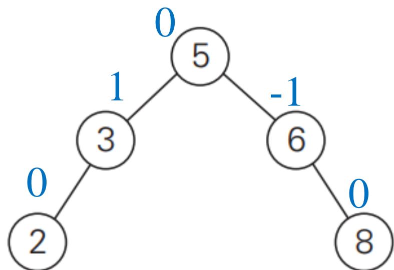
https://monalisacs.com/

MonalisaCS

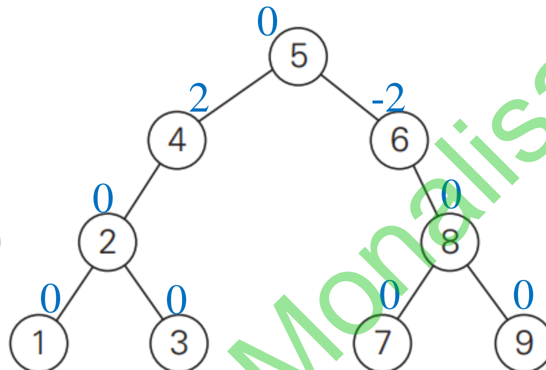
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

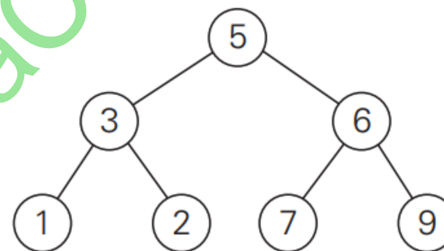
- Searching and insertion are Θ(log *n*) in the worst case.
- The operation of key deletion in an AVL tree is considerably more difficult than insertion, but fortunately it turns out to be in the same efficiency class as insertion, i.e., logarithmic.
- The drawbacks of AVL trees are frequent rotations and the need to maintain balances for its nodes.
- Q1. Which of the following binary trees are AVL trees?



(a)



(b)



(c)

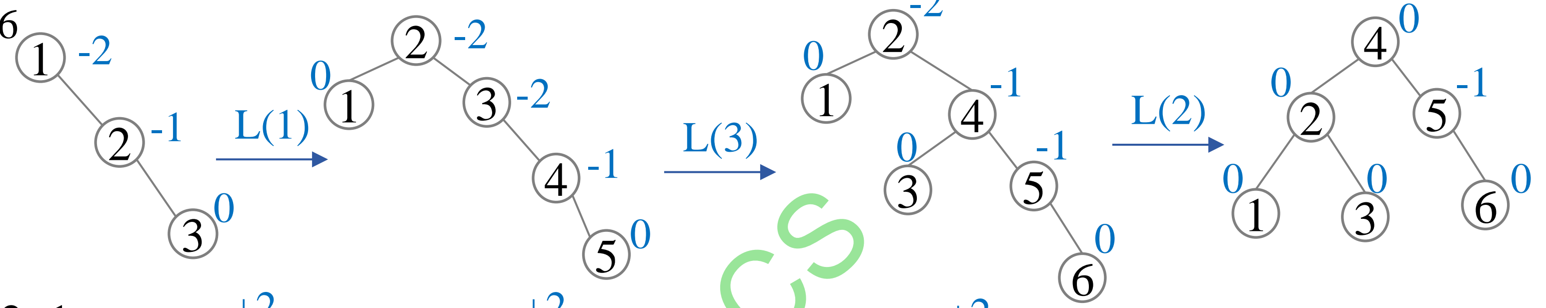
- a: AVL Tree
- b: Not AVL Tree due to balance factor 2,-2
- c: Not AVL Tree as not BST

MonalisaCS

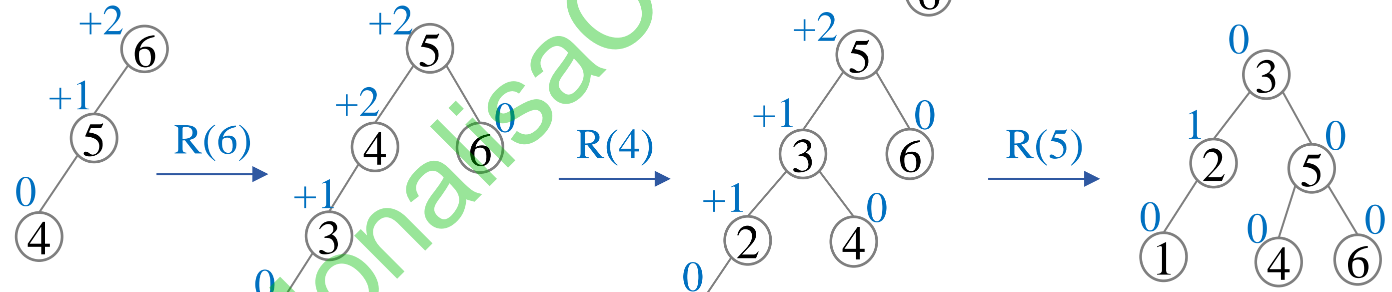
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

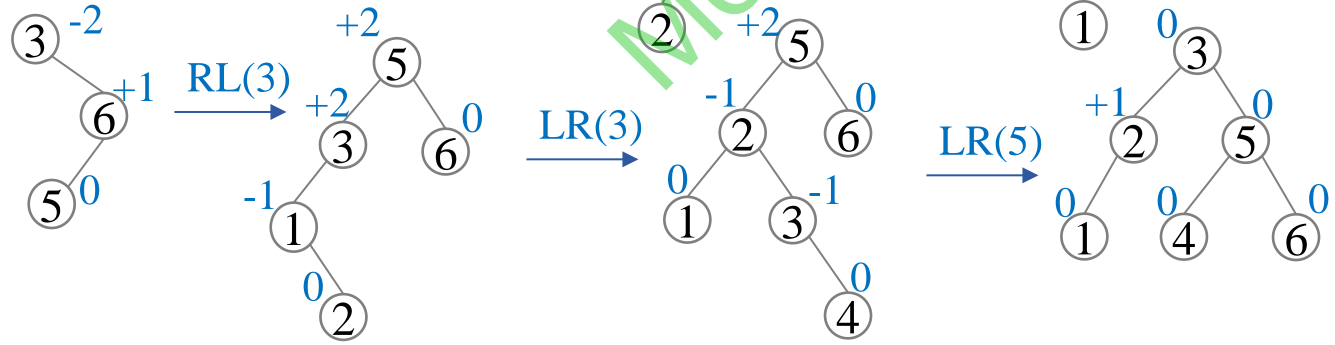
- Q2. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree. **a.** 1, 2, 3, 4, 5, 6 **b.** 6, 5, 4, 3, 2, 1 **c.** 3, 6, 5, 1, 2, 4
- **a.** 1, 2, 3, 4, 5, 6



- **b.** 6, 5, 4, 3, 2, 1



- **c.** 3, 6, 5, 1, 2, 4

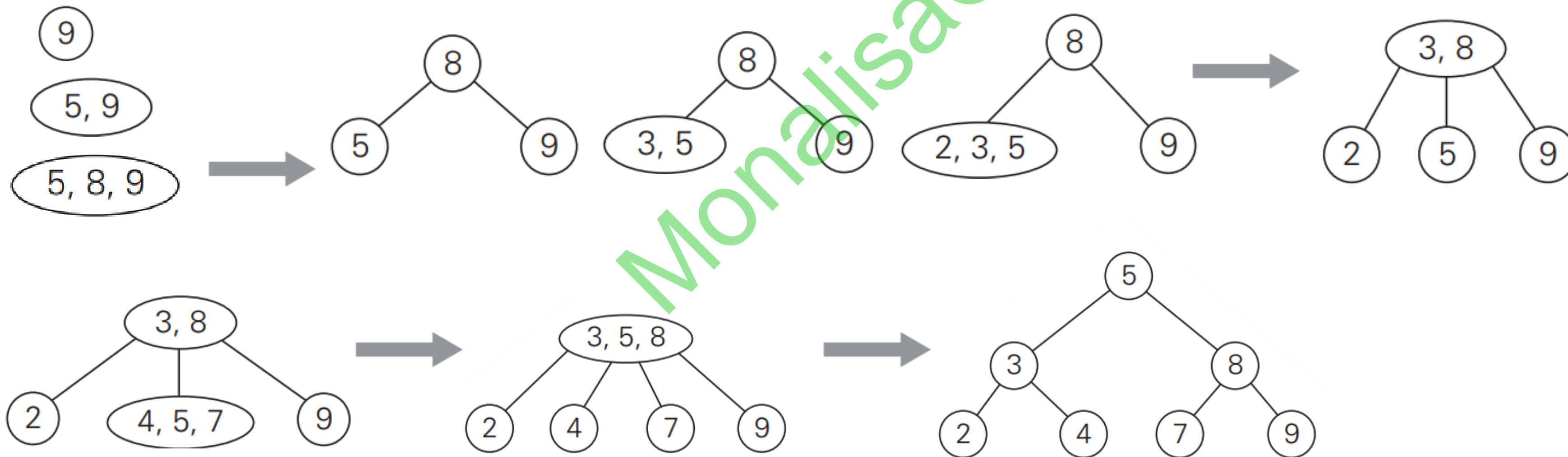


MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- A ***2-3 tree*** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.
- All its leaves must be on the same level.
- In other words, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf.
- Searching , insertion, and deletion are all in $\Theta(\log n)$ in both the worst and average case.
- We consider a very important generalization of 2-3 trees, called *B*-trees .
- Ex : Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7



MonalisaCS

https://www.youtube.com/@MonalisaCS