Transform & Conquer

Definition

Transform & Conquer is a general algorithm design technique which works in two stages.

STAGE 1: (Transformation stage): The problem's instance is modified, more amenable to solution

STAGE 2: (Conquering stage): The transformed problem is solved

The three major variations of the transform & conquer differ by the way a given instance is transformed:

1. INSTANCE SIMPLIFICATION: Transformation of an instance of a problem to an instance of the same problem with some special property that makes the problem easier to solve.
   Example: list pre-sorting, AVL trees

2. REPRESENTATION CHANGE: Changing one representation of a problem's instance into another representation of the same instance.
   Example: 2-3 trees, heaps and heapsort

3. PROBLEM REDUCTION: Transforming a problem given to another problem that can be solved by a known algorithm.
   Example: reduction to graph problems, reduction to linear programming

Presorting

- Presorting is an example for instance simplification.
- Many questions about lists are easier to answer if the lists are sorted.
- Time efficiency of the problem's algorithm is dominated by the algorithm used for sorting the list.

NOTE:
No comparison based sorting algorithm can has a better efficiency than nlogn in the worst case

Example: Problem of checking element uniqueness in an array:

- **Limitation of brute-force algorithm:**
  Compares pairs of the array's elements until either two equal elements are found or no more pairs were left. Worst case efficiency = $\Theta(n^2)$
- **Using presorting:**
  Presorting helps to sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other

ALGORITHM PresortElementUniqueness(a[0…n-1])
//solves the element uniqueness problem by sorting the array first
//i/p: An array A of orderable elements
//o/p: Returns "true" if A has no equal elements, "false" otherwise

```
Sort the array A
for i ← 0 to n-2 do
        if A[i] = = A[i+1]
                return false
return true
```

Analysis:

- Input size: n – array size
- Basic operation: key comparison
- Running time = sum of the time spent on sorting AND time spent on checking consecutive elements.
  Therefore:
  $$\begin{aligned} T(n) &= T_{sort}(n) + T_{scan}(n) \\ &\in \Theta(n \log n) + \Theta(n) \\ &\in \Theta(n \log n) \end{aligned}$$

Example: Searching problem

- **Brute-force solution:**
  Sequential search using brute-force methods needs n comparisons in the worst case, Worst case efficiency = $\Theta(n)$
- **Using presorting:**
  Presorting sorts the array first and then even if binary search is applied to search the key, the running time is:
  Running time = time required to sort the array + time required to search the key using binary search
  $$\begin{aligned} T(n) &= T_{sort}(n) + T_{search}(n) \\ &\in \Theta(n \log n) + \Theta(\log n) \\ &\in \Theta(n \log n) \end{aligned}$$

NOTE:
Using presorting, the solution is inferior to sequential search using brute force. BUT if the list is searched many number of times, then presorting is advantageous.

Balanced search trees

Description:

The disadvantage of a binary search tree is that its height can be as large as N-1, which means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case. Balanced binary search tree is a tree with small height, O(log N). (A binary tree with N node has height at least $\Theta(\log N)$)

Example of balanced search trees:

- Instance simplification variety
  AVL trees, Red-black trees
- Representation change variety:
  2-3 trees, 2-3-4 trees, B-trees

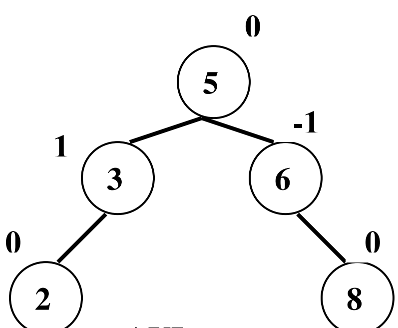
AVL trees

Definition:

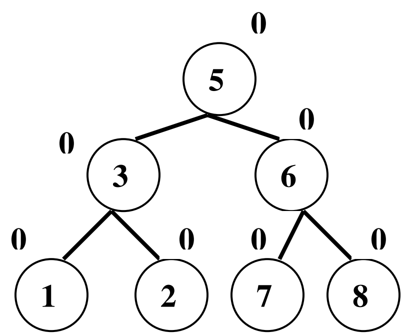
An AVL tree is a binary search tree in which the balance factor (**defined as**: difference between the heights of the node's left and right sub-trees) of every node is either 0 or +1 or -1.

NOTE:
The height of the empty tree is defined as -1
Basic operations like insertion, deletion, searching key, has efficiency as Θ(log n)

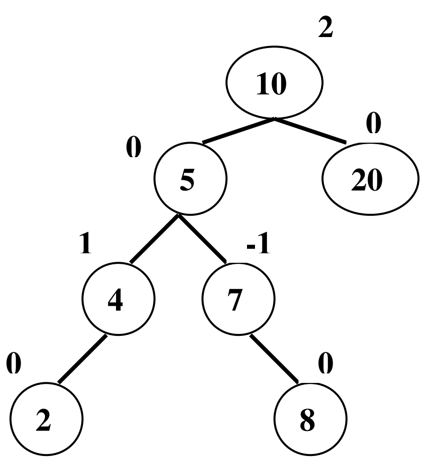
Example:



AVL tree



NOT AVL, because it is NOT Binary Search tree



NOT AVL tree, because balance factor of node (10) is 2

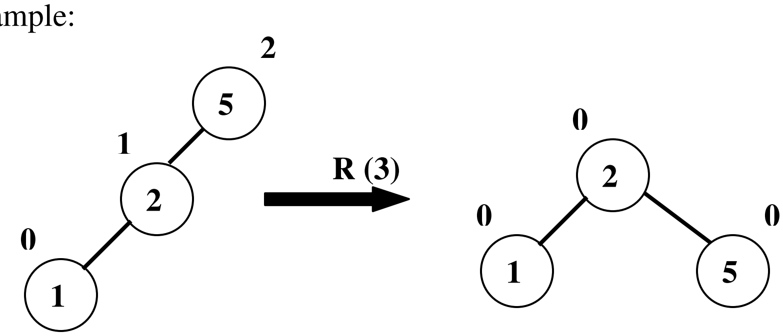
AVL tree rotation
If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.

Rotation
Definition:
A rotation in an AVL tree is a local transformation of its sub-tree rooted at a node whose balance has become either +2 or -2.

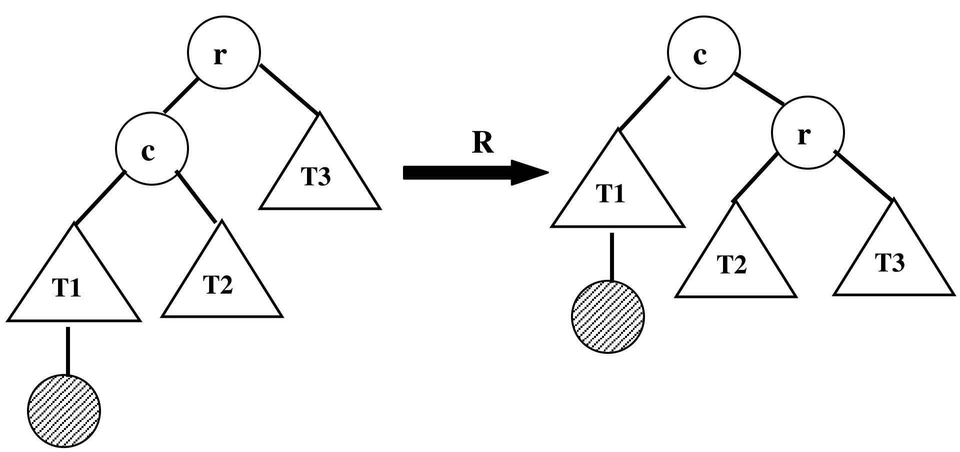
NOTE:
If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf

Types of rotations:
There are 4 types of rotations:
1. Single R – rotation
2. Single L – rotation
3. Double LR – rotation
4. Double RL – rotation

Single R – rotation
- Single right rotation
- Rotates the edge connecting the root and its left child in the binary tree
- Example:



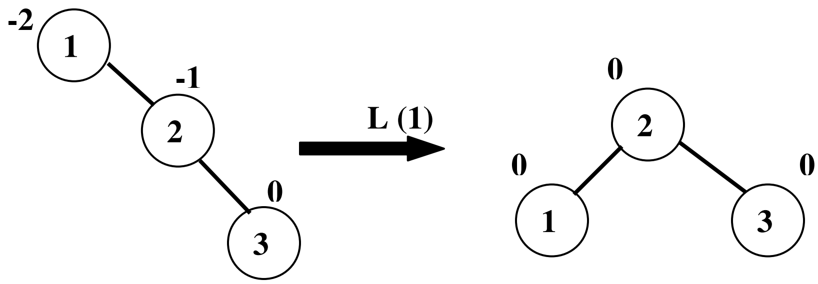
- General form: A shaded node is the last node inserted



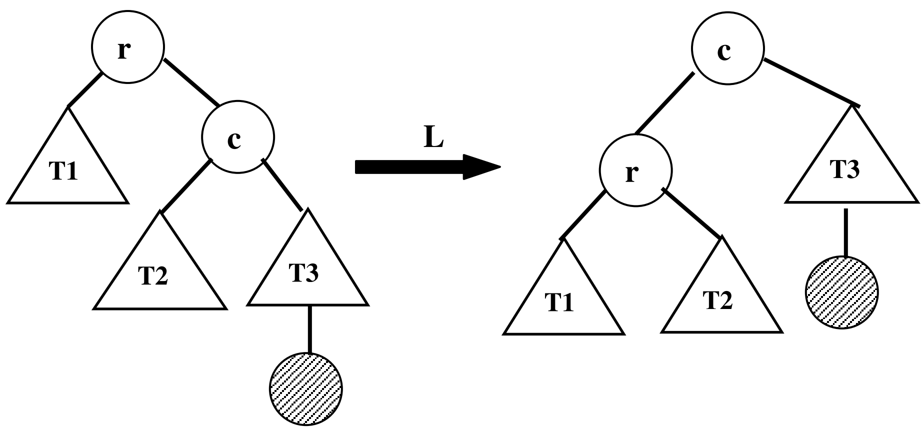
- Here rotation is performed after a new key is inserted into the left sub-tree of the left child of a tree whose root had the balance of +1 before the insertion

Single L – rotation

- Single left rotation
- Rotates the edge connecting the root and its right child in the binary tree
- Example:



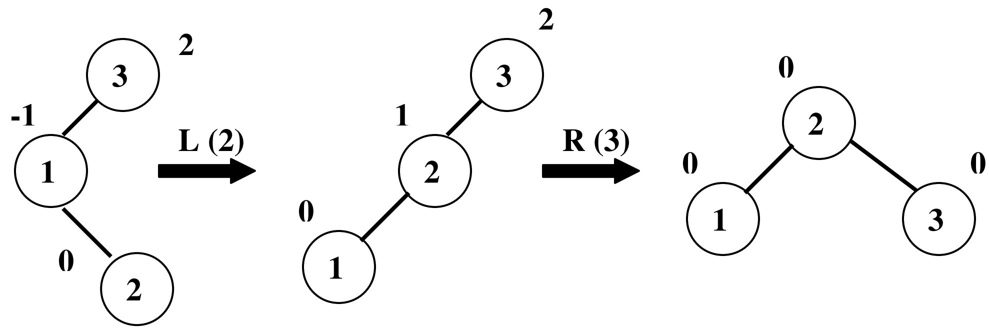
- General form: A shaded node is the last node inserted



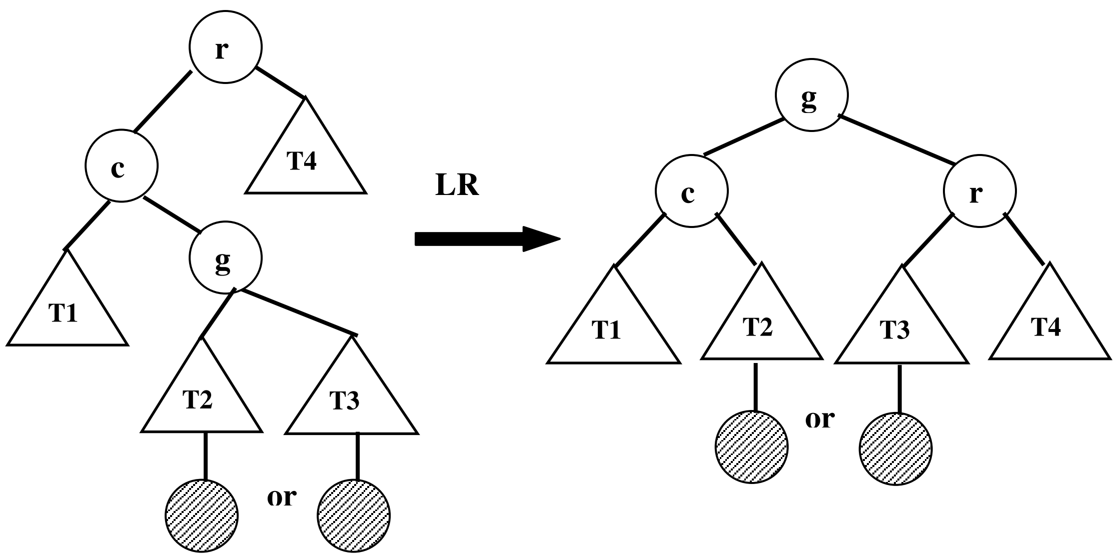
- Here rotation is performed after a new key is inserted into the right sub-tree of the right child of a tree whose root had the balance of -1 before the insertion

Double LR – rotation

- Double left-right rotation
- Combination of two rotations
    1. perform left rotation of the left sub-tree of root r
    2. perform right rotation of the new tree rooted at r
- It is performed after a new key is inserted into the right sub-tree of the left child of a tree whose root had the balance of +1 before the insertion
- Example:

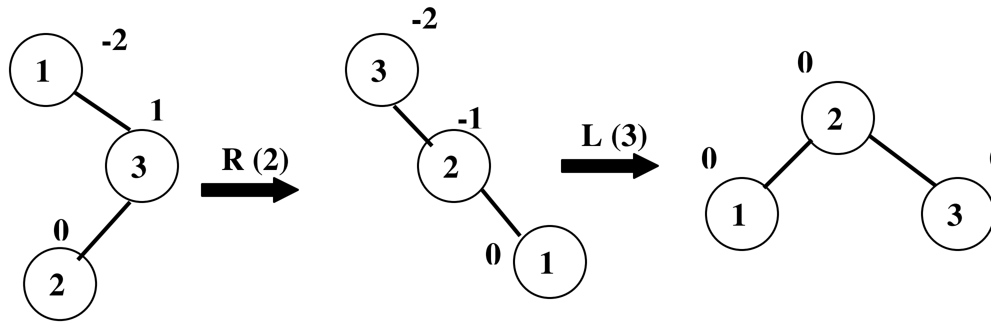


- General form: A shaded node is the last node inserted. It can be either in the left sub-tree or in the right sub-tree of the root's grandchild.

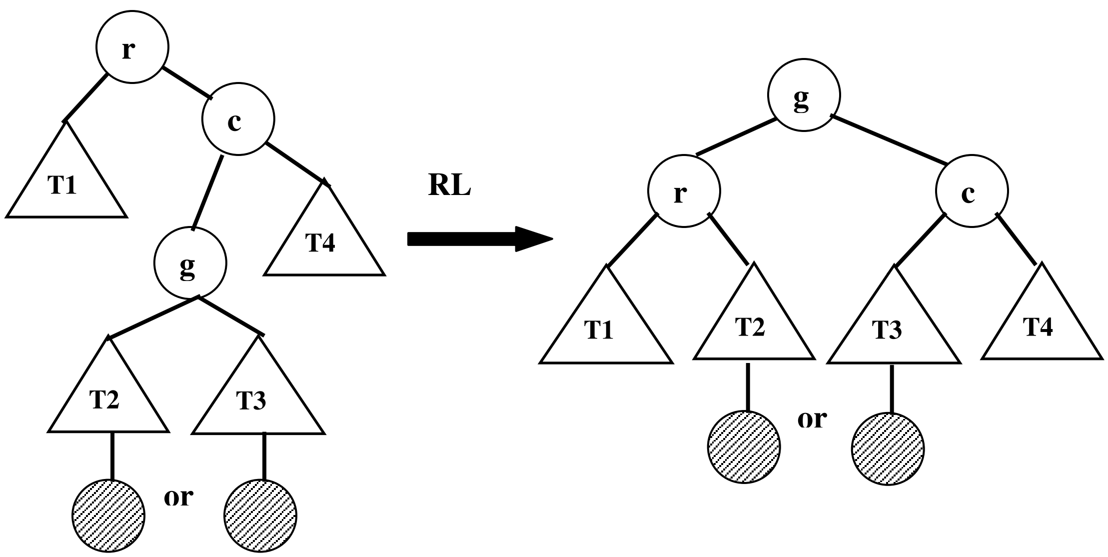


Double RL – rotation

- Double right-left rotation
- Combination of two rotations
    1. perform right rotation of the right sub-tree of root r
    2. perform left rotation of the new tree rooted at r
- It is performed after a new key is inserted into the left sub-tree of the right child of a tree whose root had the balance of -1 before the insertion
- Example:



- General form: A shaded node is the last node inserted. It can be either in the left sub-tree or in the right sub-tree of the root's grandchild.



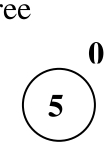
Construction of AVL tree

Question: Construct AVL tree for the list by successive insertion

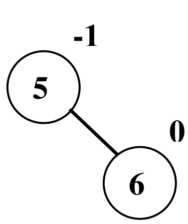
5, 6, 8, 3, 2, 4, 7

Solution:

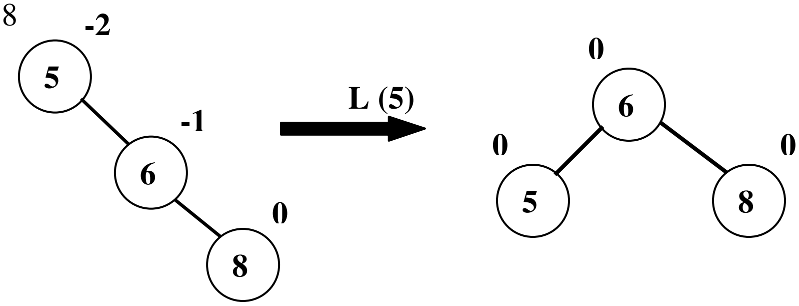
i. Insert 5 into empty tree



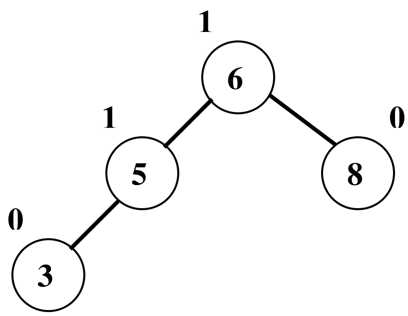
ii. Insert 6



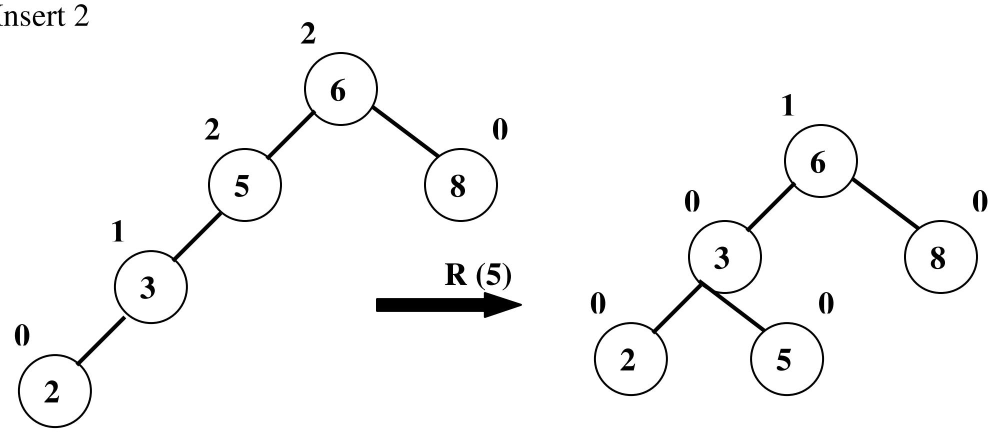
iii. Insert 8



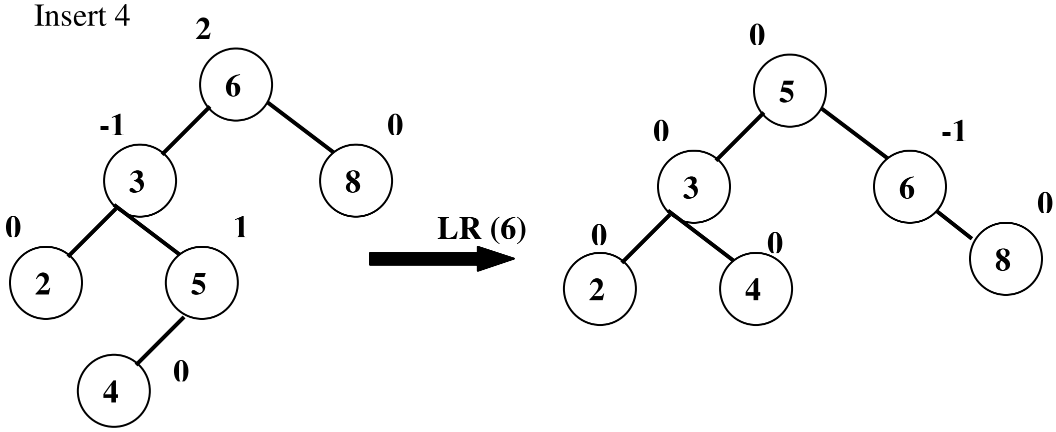
iv. Insert 3



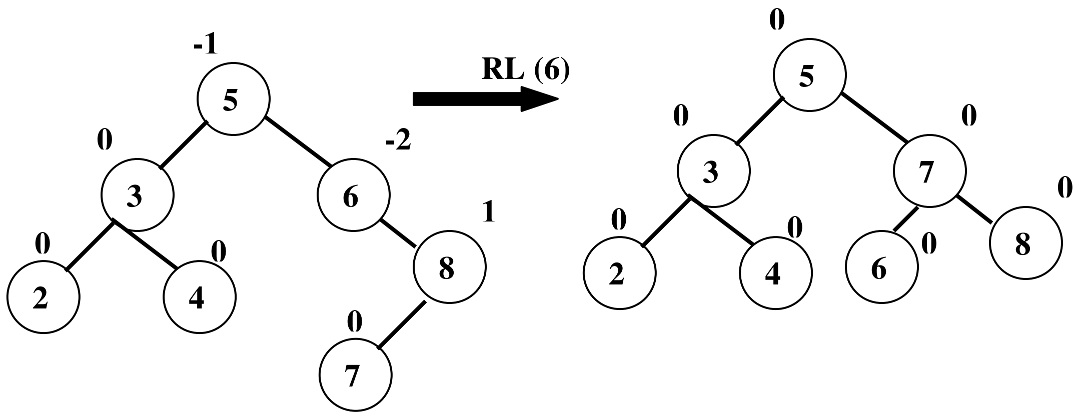
v. Insert 2



vi. Insert 4



vii. Insert 7



Limitations of AVL trees

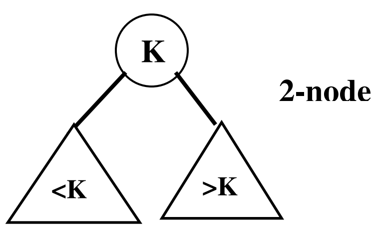
- Requires frequent rotations to maintain balances for the tree's nodes
- Even though the deletion operation efficiency is Θ(log n), it is considerably more difficult than insertion

2-3 trees

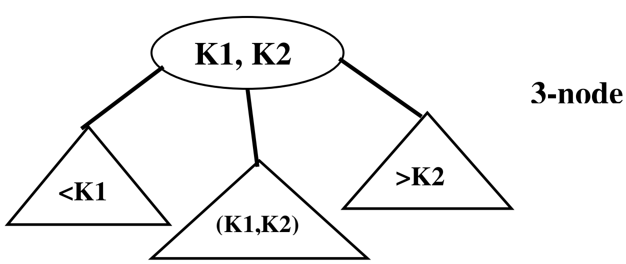
Definition:

2-3 tree is a height balanced search tree, that has all its leaves on the same level and can have nodes of two kinds:

- **2-node**: contains a single key K and has two children- the left child serves as the root of a sub-tree whose keys are less than K, and the right child serves as the root of a sub-tree whose keys are greater than K



- **3-node**: contains two ordered keys K1 and K2, (K1 < K2) and has three children. The leftmost child serves as the root of a sub-tree with keys less than K1, middle child serves as the root of a sub-tree with keys between K1 and K2, and the rightmost child serves as the root of a sub-tree with keys greater than K2



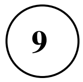
Construction of 2-3 tree:

Question: Construct a 2-3 tree by successive insertion for the following list

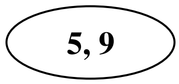
9, 5, 8, 3, 2, 4, 7

Solution:

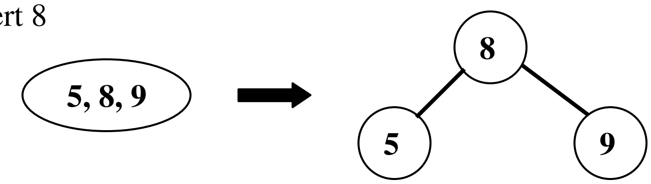
i. Insert 9 into empty tree



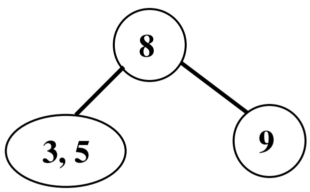
ii. Insert 5



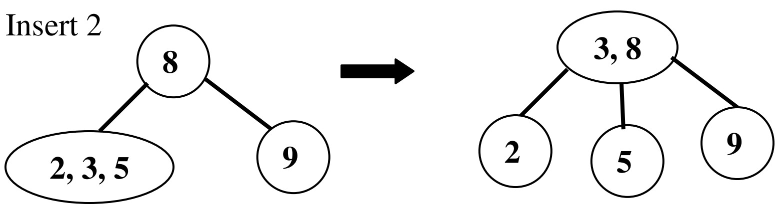
iii. Insert 8



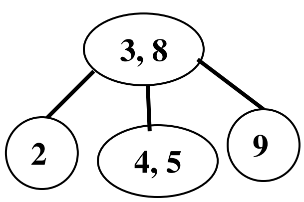
iv. Insert 3



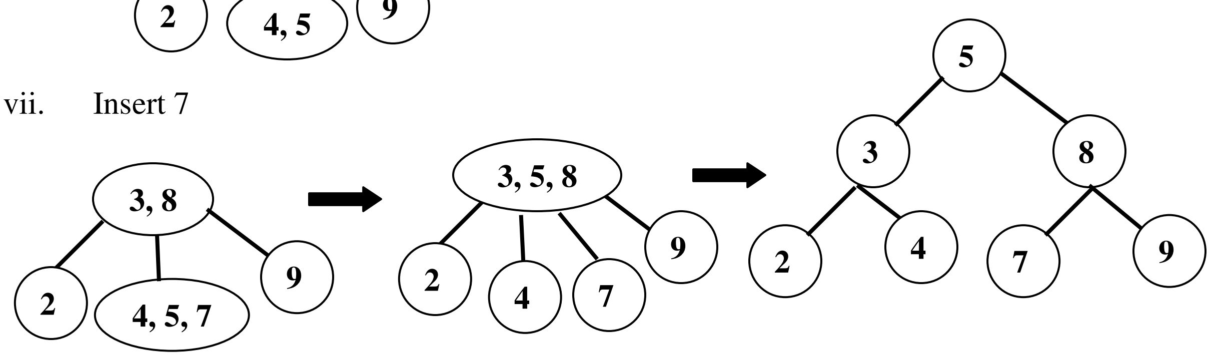
v. Insert 2



vi. Insert 4



vii. Insert 7



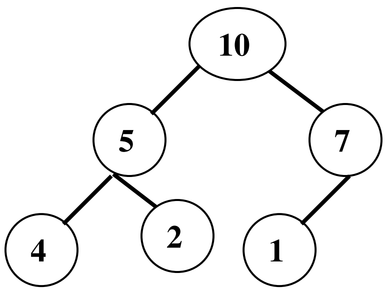
Heaps

Definition:

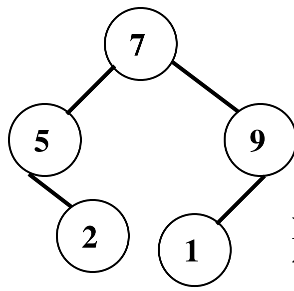
A heap is a binary tree with keys assigned to its nodes, provided the following two conditions are met:

i. **Tree's shape requirement**: The binary tree is essentially complete, all its level are full except possibly the last level, where only some rightmost leaves may be missing.
ii. **Parental dominance requirement**: The key at each node is greater than or equal to the keys at its children.

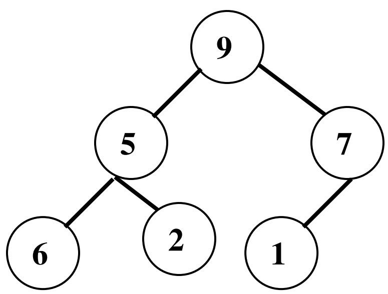
Example:



Heap



NOT a heap, as tree's shape requirement fails

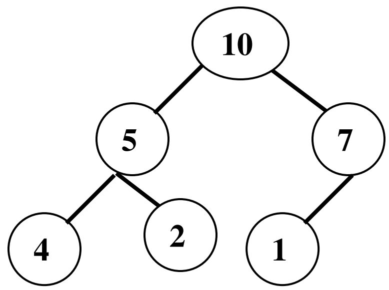


NOT a heap, as parental dominance requirement fails at node 5

Important properties of heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$
2. The root of a heap always contains its largest element
3. A node of a heap considered with all its descendents is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. Heap elements are stored in positions 1 through n of an array. In such representation:
    a. The parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.
    b. The children of a key in the array's parental position i ($1 \leq i \leq \lfloor n/2 \rfloor$) will be in position 2i and 2i +1, and, correspondingly the parent of a key in position i ($2 \leq i \leq n$) will be in position $\lfloor i/2 \rfloor$

Example:



Index	0	1	2	3	4	5	6
value		10	5	7	4	2	1
			Parents			leaves	

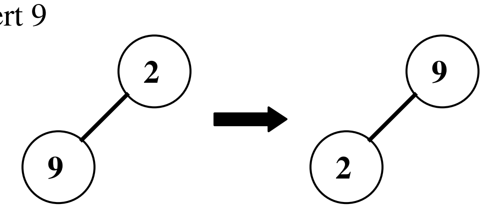
Heap construction

Construct heap for the list: 2, 9, 7, 6, 5, 8

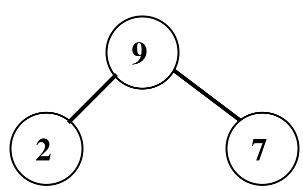
There are two methods for heap construction:

1. **Top-down construction**: Constructs a heap by successive insertions of a new key into a previously constructed heap.
    a. Insert 2 into empty tree

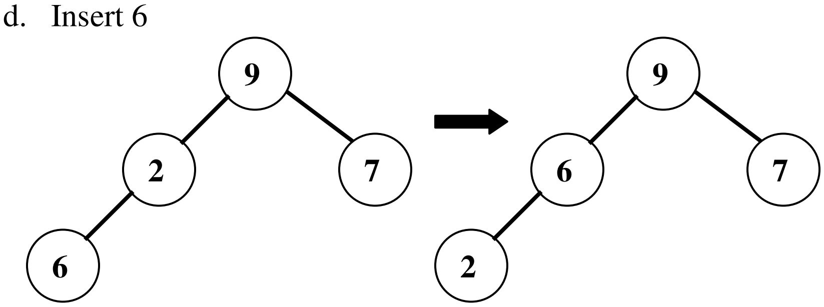
    b. Insert 9



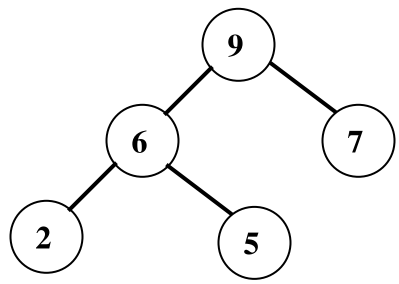
    c. Insert 7



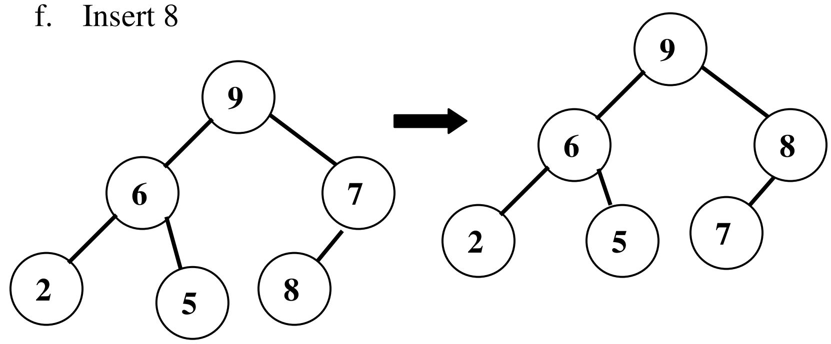
    d. Insert 6



e. Insert 5



f. Insert 8

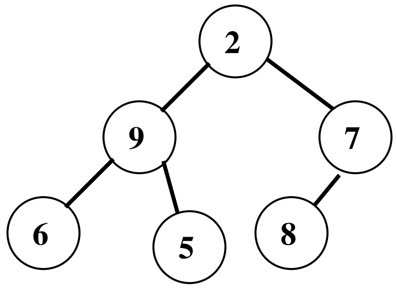


2. **Bottom - up construction**:

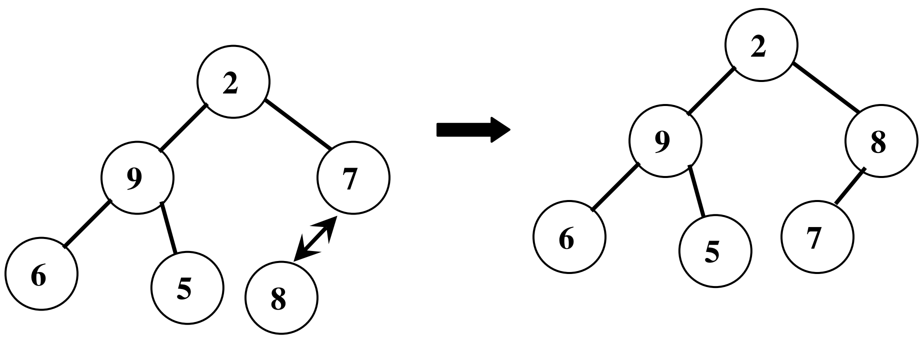
ALGORITHM HeapBottomUp(H[1…n])
//constructs a heap from the elements of a given array by bottom-up algorithm
//i/p: An array H[1…n] of orderable items
//o/p: A heap H[1…n]

```
for i ← ⌊n/2⌋ down to 1 do
        k ← i
        v ← H[k]
        heap ← false
        while NOT heap AND 2*k ≤ n do
                j ← 2 * k
                if j < n
                        if H[j] < H[j+1]
                                j ← j + 1
                if v ≥ H[j]
                        heap ← true
                else
                        H[k] ← H[j]
                        K ← j
        H[k] ← v
```

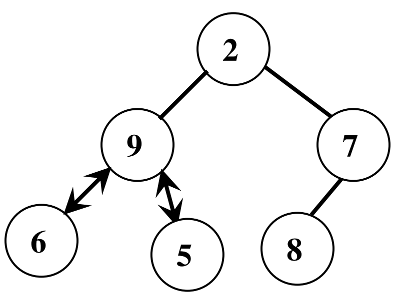
1. Initialize the essentially complete binary tree with n nodes by placing keys in the order given and then heapify the tree.



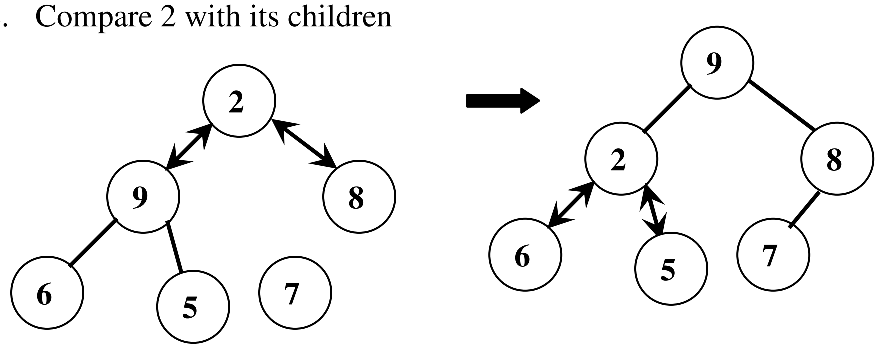
2. Heapify
    a. Compare 7 with its child

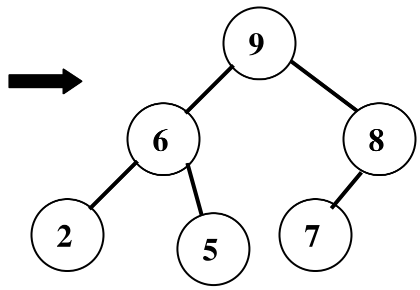


    b. Compare 9 with its children



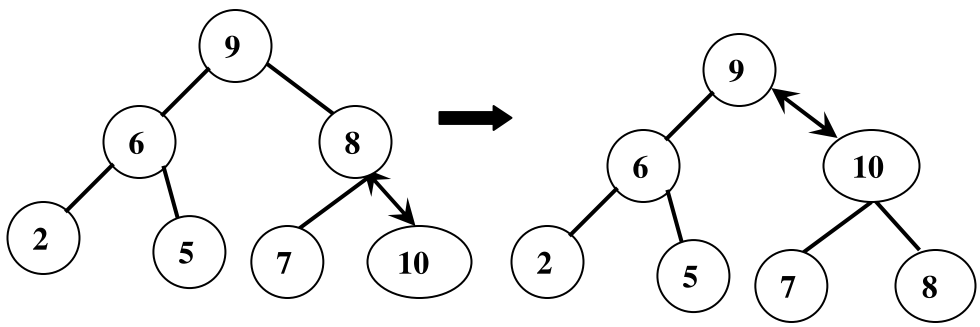
    c. Compare 2 with its children

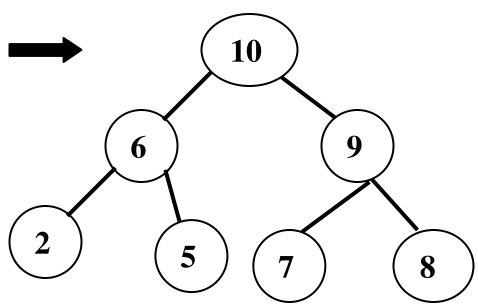




Inserting a key into heap

Example: Insert a key 10 into the heap (9, 6, 8, 2, 6, 7)
Insert the new key as the last node in the heap, then heapify.

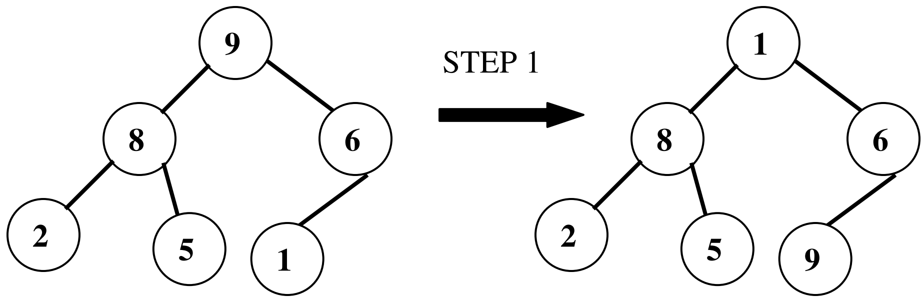


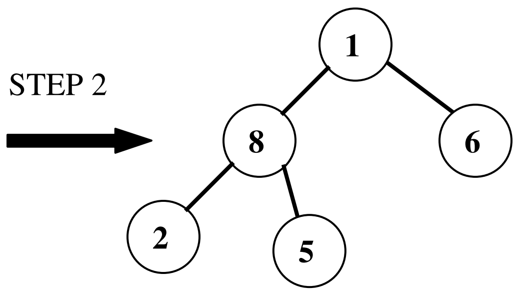


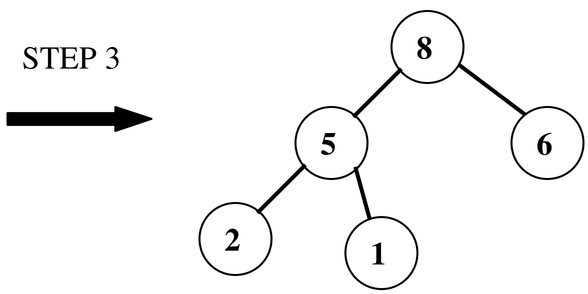
Deleting a key from heap

Example: Delete the root's key from the heap (9, 8, 6, 2, 5, 1)
Solution:

MAXIMUM Key Deletion algorithm
Step 1: Exchange the root's key with the last key K of the heap.
Step 2: Decrease the heap's size by 1
Step 3: "Heapify" the smaller tree.







Heap sort

Description:

- Invented by J.W.J Williams
- It 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

Example:

Sort the following lists by heap sort by using the array representation of heaps.

2, 9, 7, 6, 5, 8

STAGE 1: Heap construction

2 9 **7** 6 5 8
2 **9** 8 6 5 7
2 9 8 6 5 7
9 **2** 8 6 5 7
9 **6** 8 2 5 7

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

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

2 6 5
6 2 5

5 2
5 2

2
2

STAGE 2: Maximum deletion

9 6 8 2 5 7
7 6 8 2 5 | **9**

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

7 6 5 2
2 6 5 | **7**

6 2 5
5 2 | **6**

5 2
2 | **5**

| **2**

Efficiency of heap sort:

Heap sort is an in place algorithm. Time required for heap construction + time required for maximum deletion

= O(n) + O(n log n)
= O(n log n)