INTERNITY FOUNDATION

PRESENTATION ON BIT MASKING,RED BLACK TREE, HEAPS

Presented by:
Lipika Chugh
Aman Goyal
Akash Sharma
Harshit Garg

Bit Masking

- Bit masking is a method of manipulating bits from a binary sequence.
- Bit masking is basically done using Bitwise operators,like:
  - Bitwise AND Operator (&)
  - Bitwise OR Operator (|)
  - Bitwise NOT Operator (!)
  - Bitwise XOR Operator (^)
  - Bitwise Left Shift (<<)
  - Bitwise Right Shift (>>).
- Mask in Bitmask means hiding something. Bitmask is nothing but a binary number that represents something.

Some uses of bit masking like:

- Dividing a number by a power of 2
  Like for x=38 in binary:100110
  100110>>1=010011 which is 19 (38/2)
  100110>>2=001001 which is 9 (38/4) and so on.

- Multiplying a number by a power of 2
  For x=9,which is 1001
  1001<<1=10010 which is 18 (9*2)
  1001<<2=100100 which is 36 (9*4)

- Setting a particular bit to be as 1
  For x=10010101,if we want to set the 3rd bit as 1
  Then we OR it with 00100000.
  So we get 10110101

- Toggle the nth bit of a number
  For x=10101010110,if we want to change the 5th bit
  Then we XOR it with 00001000000
  So we get 10100010110

- Assign some bits as 0
  Foe x=1010101010,and we want to change the 5th,7th bit as 0
  Then we AND it with 111101011111
  So we get 1010000010.

Other applications of Bit masking

Uppercase English Alphabet to Lowercase:

ch |= ' ';

A -> 01000001 a -> 01100001
B -> 01000010 b -> 01100010
C -> 01000011 c -> 01100011
. .
. .
Z -> 01011010 z -> 01111010

ch = ‘A’ (01000001)

mask = ‘ ‘ (00100000)

ch | mask = ‘a’ (01100001)

Converting Lowercase to uppercase

ch &= '_' ;

A -> 01000001 a -> 01100001

B -> 01000010 b -> 01100010

C -> 01000011 c -> 01100011

. .

. .

Z -> 01011010 z -> 01111010

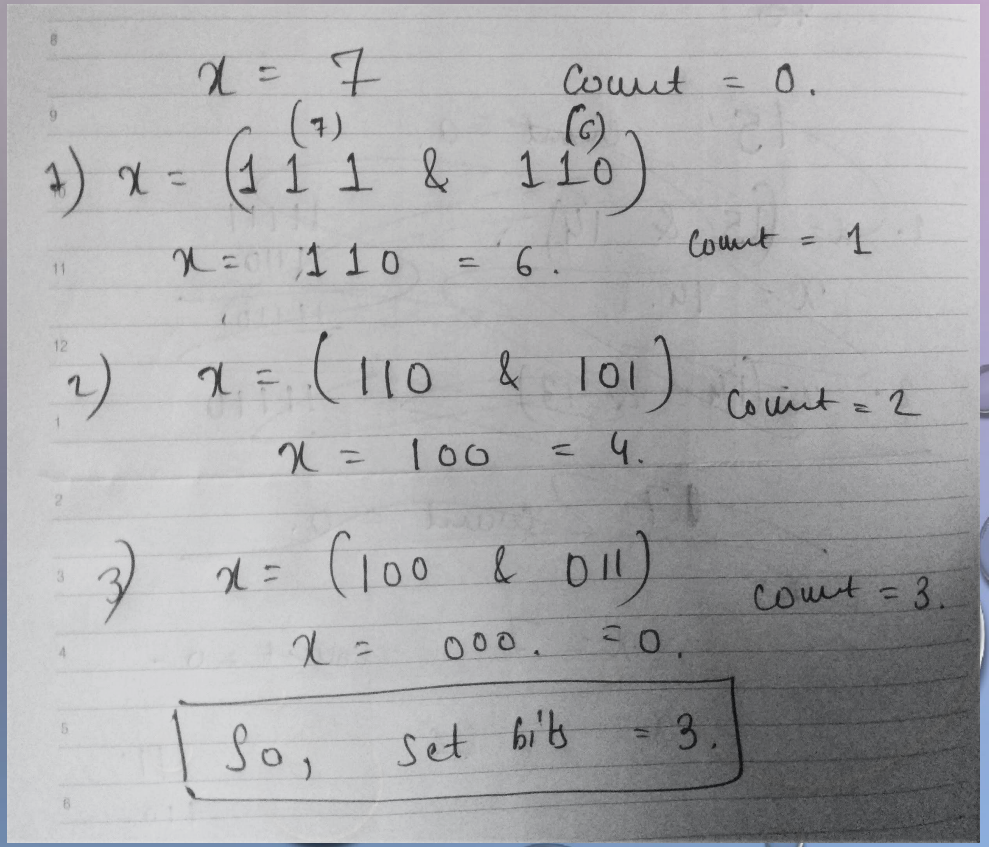
ch = ‘a’ (01100001)

mask = ‘_ ‘ (11011111) (95)

ch & mask = ‘A’ (01000001)

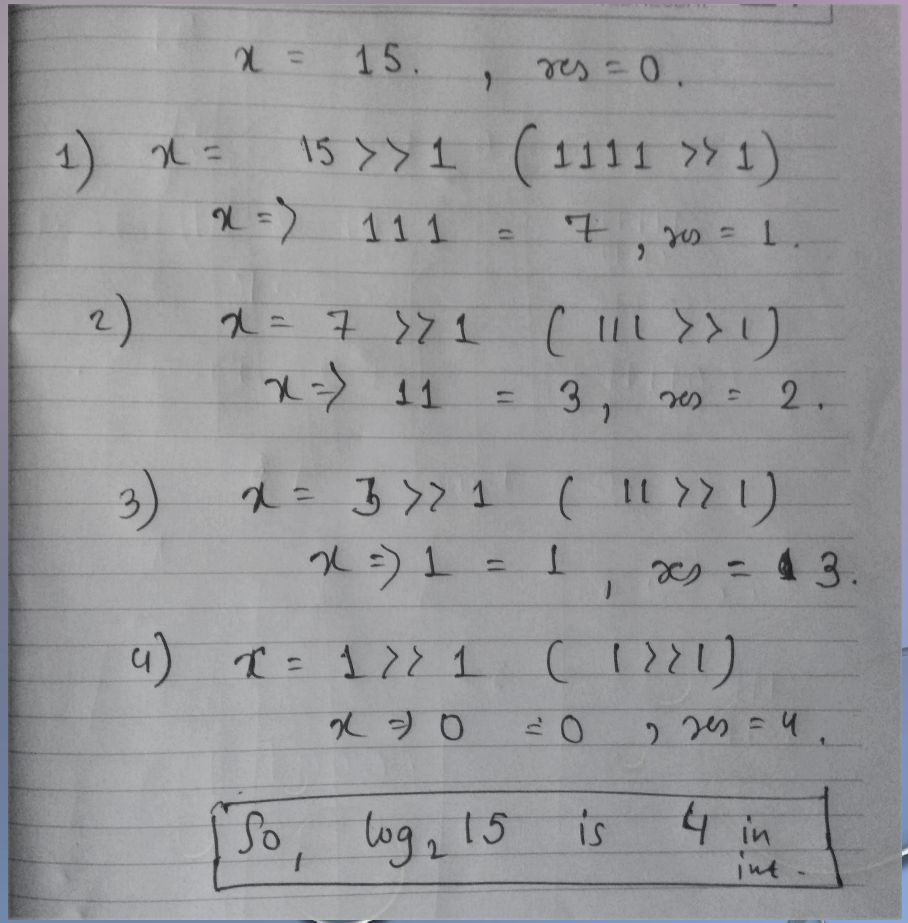
Count set bits in integer (Brian Kernighan's Algorithm)

```
int countSetBits(int x)
{
    int count = 0;
    while (x)
    {
        x &= (x-1);
        count++;
    }
    return count;
}
```



Find log base 2 of 32 bit integer

```
int log2(int x)
{
    int res = 0;
    while (x >>= 1)
        res++;
    return res;
}
```



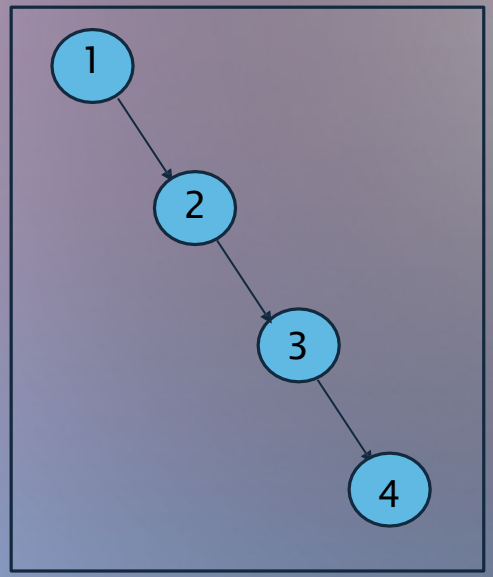
Red Black Trees

Data Structures

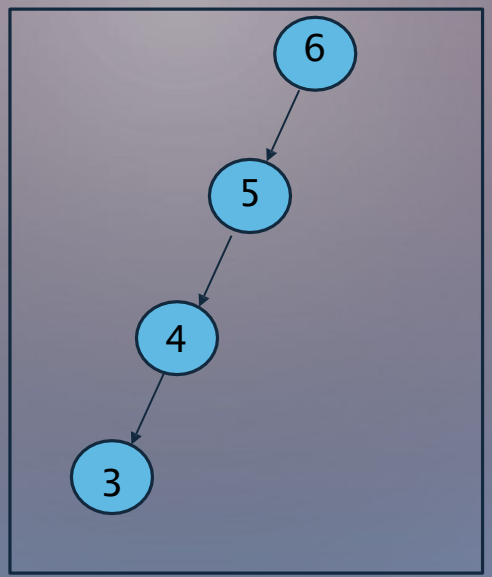
Problem with Binary Search Tree

- Binary Search Tree is fast in insertion and deletion etc. when balanced.
- Time Complexity of performing operations (e.g. searching , inserting , deletion etc) on binary search tree is O(logn) in best case i.e when tree is balanced.
- And on the other hand performance degrades from O(logn) to O(n) when tree is not balanced.
- Basic binary search trees have three very nasty degenerate cases where the structure
  stops being logarithmic and becomes a glorified linked list.
- The two most common of these degenerate cases is ascending or descending sorted order (the third is outside-in alternating order).

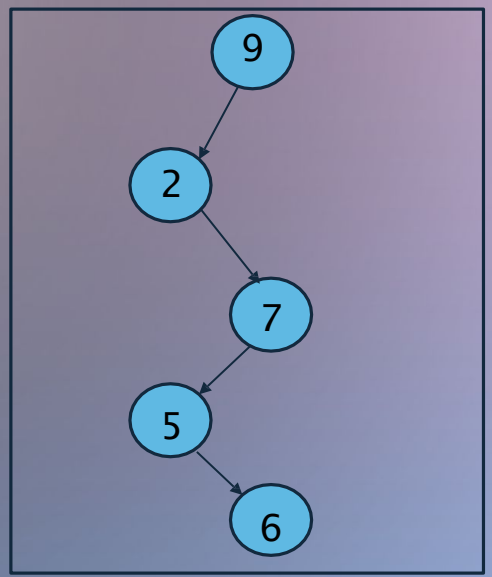
Problem with Binary Search Tree



Ascending order



Descending Order



Alternating Order

Red Black Tree

- A red-black tree is a balanced binary search tree with one extra bit of storage per node:
  its color, which can be either **Red** or **Black**.

- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

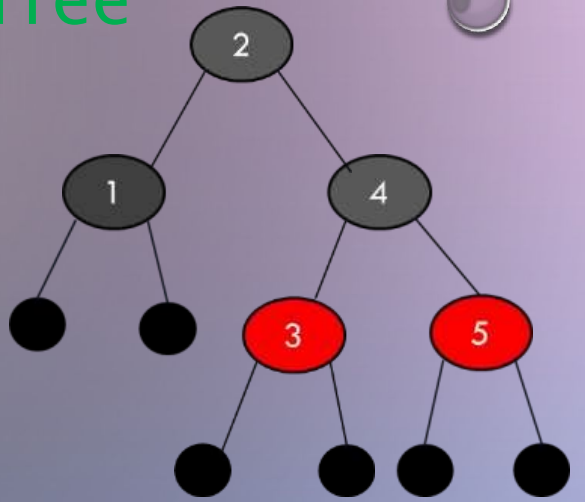
- Each node of the tree now contains the attributes color, key, left, right, and p.

- If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

Properties of Red Black Tree

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



Red Black Tree (Rotations)

- The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red black tree with n keys, take O(log n)time. Because they modify the tree, the result may violate the red-black properties.

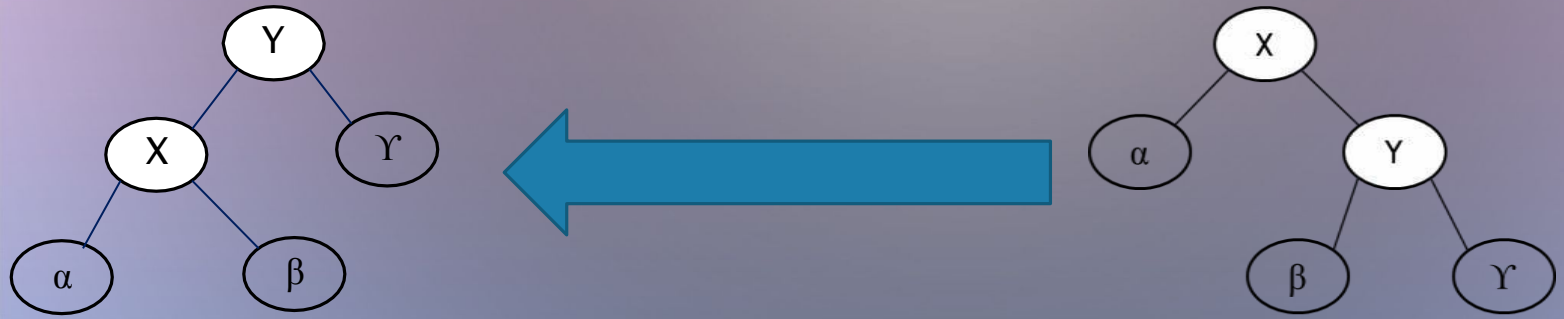
- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

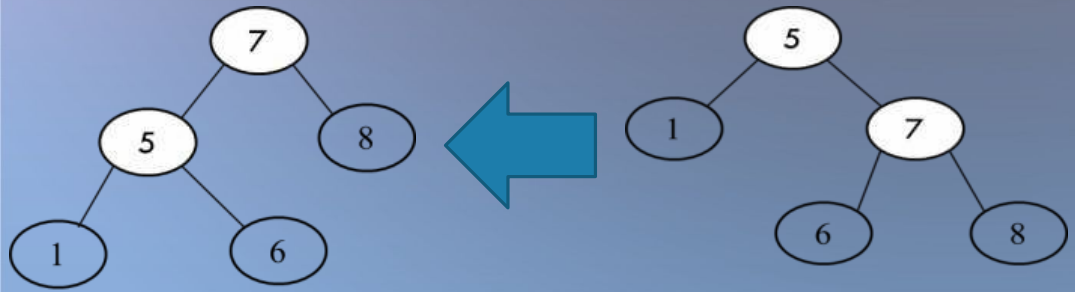
- We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property.

- There are two kinds of rotations : left rotations and right rotations.

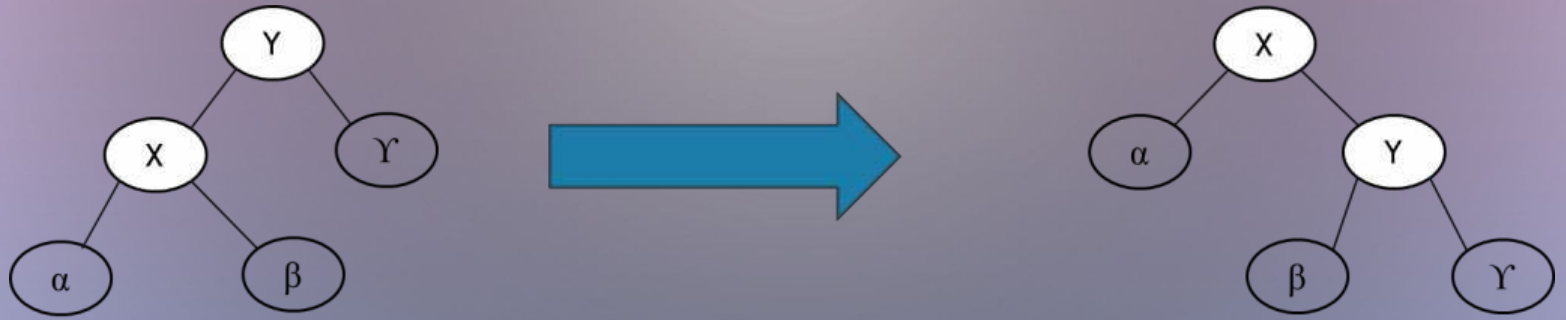
LEFT ROTATION

- When we do a left rotation on a node x, we assume that its right child y is not null ;x may be any node in the tree whose right child is not null.





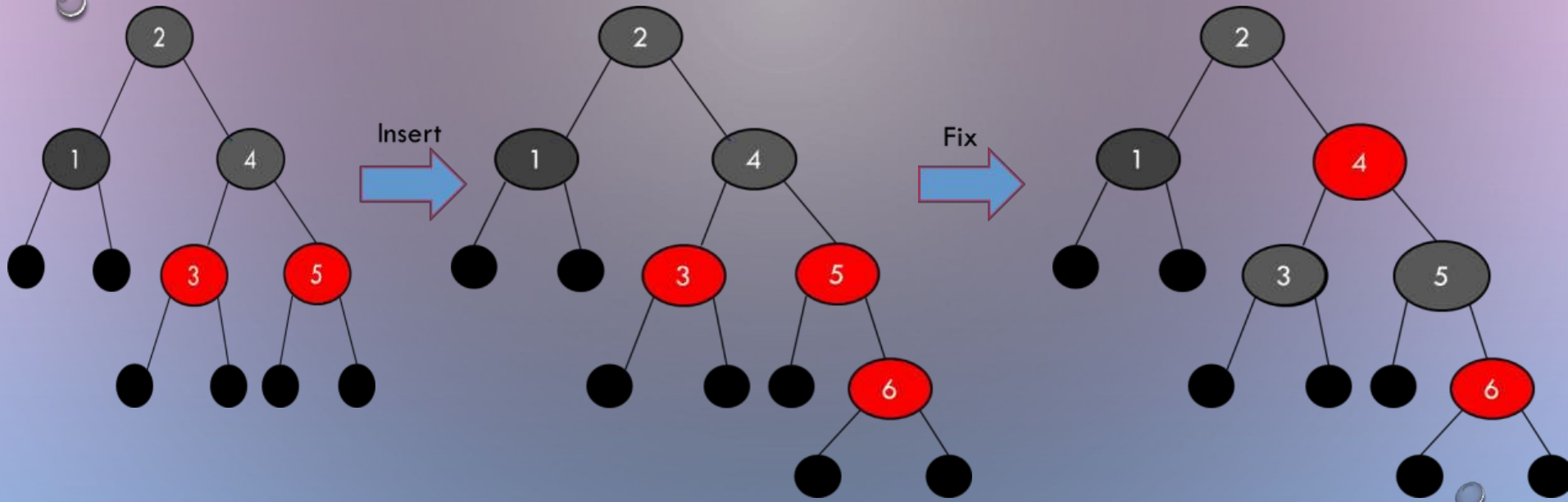
RIGHT ROTATION



Insertion

- We can insert a node into an n-node red-black tree in O(logn) time.
- To do so, we use a slightly modified version of the TREE-INSERT procedure to insert node into the tree T as if it were an ordinary binary search tree.
- Then we color it to red.
- To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT FIXUP to recolor nodes and perform rotations.

Insertion



Insertion

```
RB-INSERT(T, z)
 1.   y ← T.nil
 2.   x ← T.root
 3.   while x ≠ T.nil
 4.         y = x
 5.         if z.key < x.key
 6.             then x ← x.left
 7.         else x ← x.right
 8.   z.p = y
 9.   if y = T.nil
10.         then T.root ← z
11.   else if z.key< y.key
12.         then y.left ← z
13.   else y.right ← z
14.         z.left ← T.nil
15.         z.right ← T.nil
16.         z.color← RED
17.   RB-INSERT-FIXUP(T, z)
```

Insertion

- The procedures TREE-INSERT and RB-INSERT differ in four ways..
  - First, all instances of NIL in TREE-INSERT are replaced by T.nil.
  - Second, we set z.left and z.right to T. nil in lines 14–15 of RB-INSERT, in order to
    maintain the proper tree structure.
  - Third, we color z red in line 16.
  - Fourth, because coloring z.red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T,z) in line 17 of RB-INSERT to restore the red-black properties

Insertion

- To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps.
- First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red.
- Second, we shall examine the overall goal of the while loop in lines 1–15.
- Finally, we shall explore each of the three cases within the while loop's body and see how they accomplish the goal.

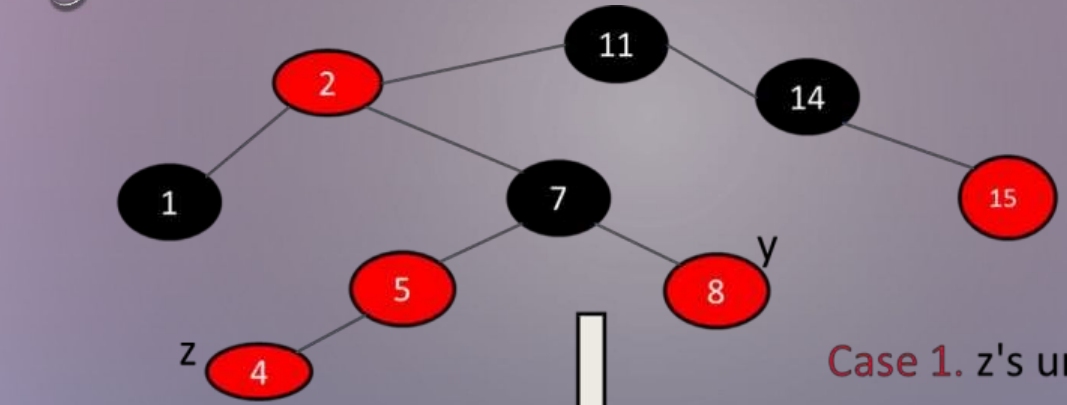
Insertion

- The while loop in lines 1–15 maintains the following three-part invariant.
- At the start of each iteration of the loop,
  a) Node z is red.

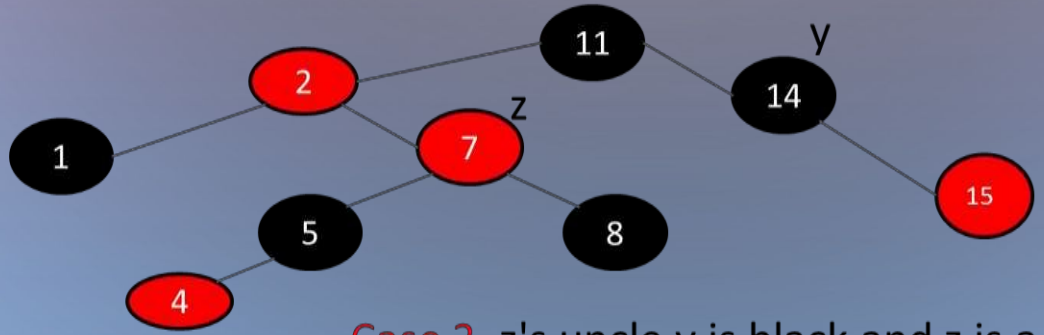
  b) If z.p is the root, then z.p is black.

  c) If there is a violation of the red-black properties , there is at most one violation, and it is of either property 2 or property 4. If there is a violation of property 2, it occurs because z is the root and is red. If there is a violation of property 4, it occurs because both z and z.p are red.

Insertion

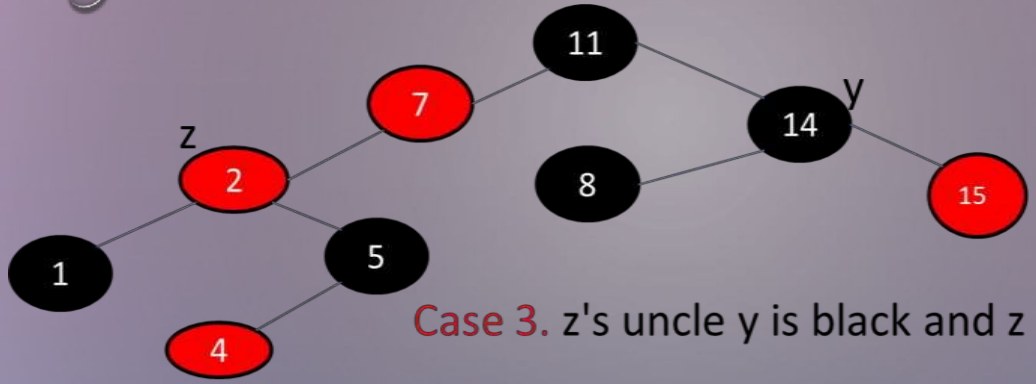


Case 1. z's uncle y is red

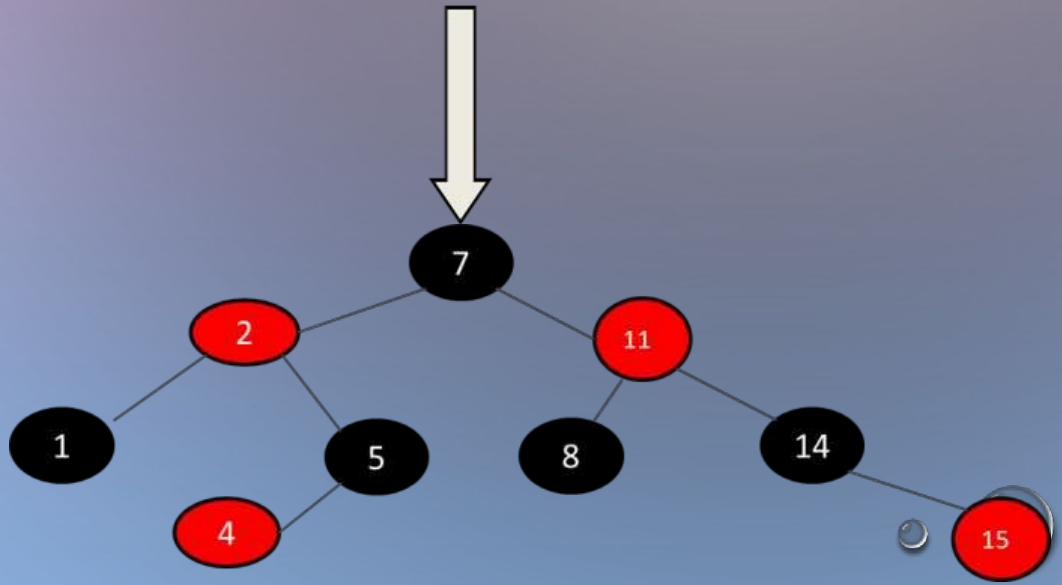


Case 2. z's uncle y is black and z is a right child

Insertion



Case 3. z's uncle y is black and z is a left child



Heaps

Heap is a data structure which is similar to binary tree with the two properties:

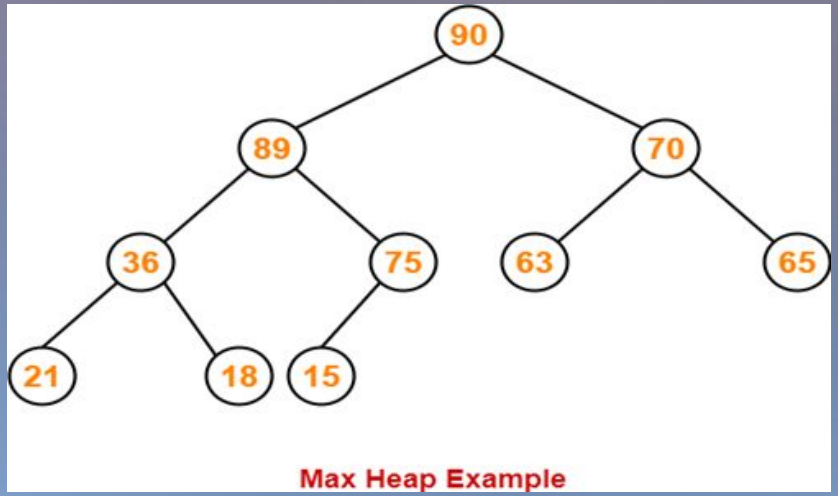
1. Complete Binary Tree
2. Heap Order Property

We use heaps in order to resolve the issues with a balanced BST, i.e., maintenance of balancing factor after every insertion or deletion and storing the many child pointers.

•A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

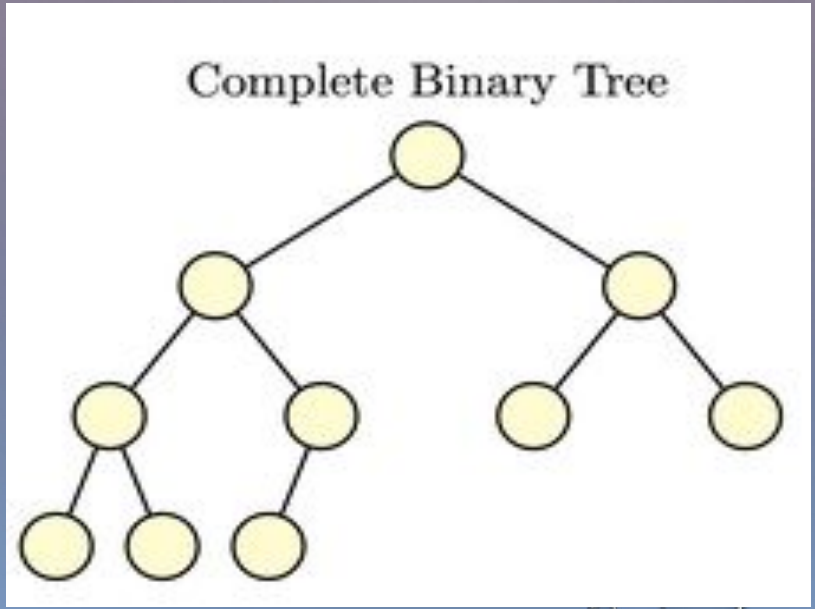
•the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

•the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value **element at the root.**



Max Heap Example

- Complete binary tree– if not full, then the only unfilled level is filled in from left to right.



- The heap property of a tree is a condition that must be true for the tree to be considered a heap.

  Min-heap property: for min-heaps, requires

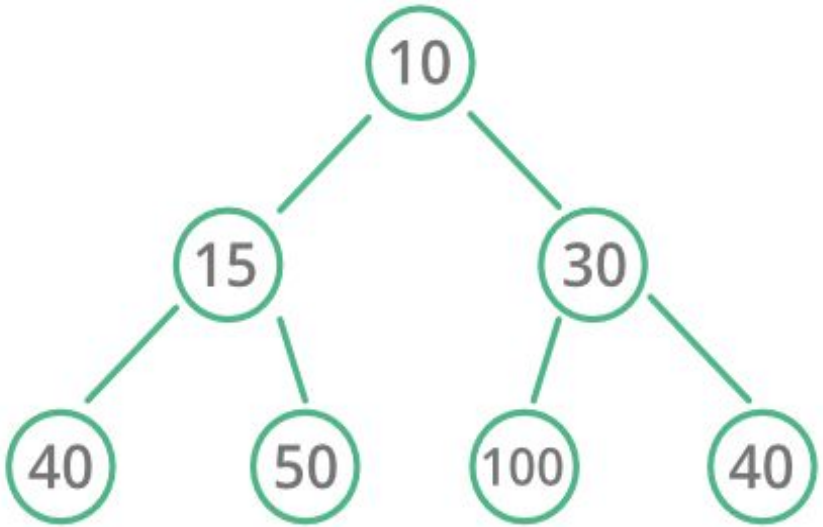
  A[parent(i)] <= A[i]

  *So, the root of any sub-tree holds the **least** value in that sub-tree.*

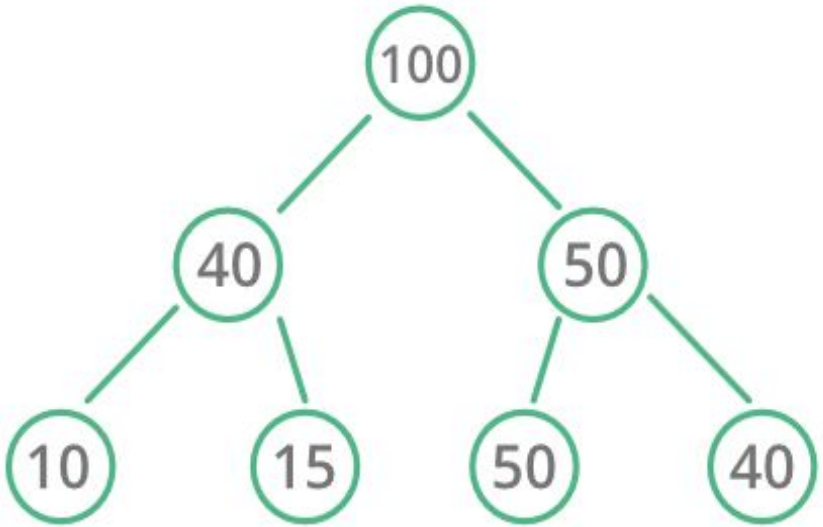
  Max-heap property: for max-heaps, requires

  A[parent(i)] >= A[i]

  *The root of any sub-tree holds the **greatest** value in the sub-tree.*



Min Heap



Max Heap

Max Heap Construction Algorithm

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Operations on Max Heap

- Finding Maximum
- Insertion
- Deletion of maximum element

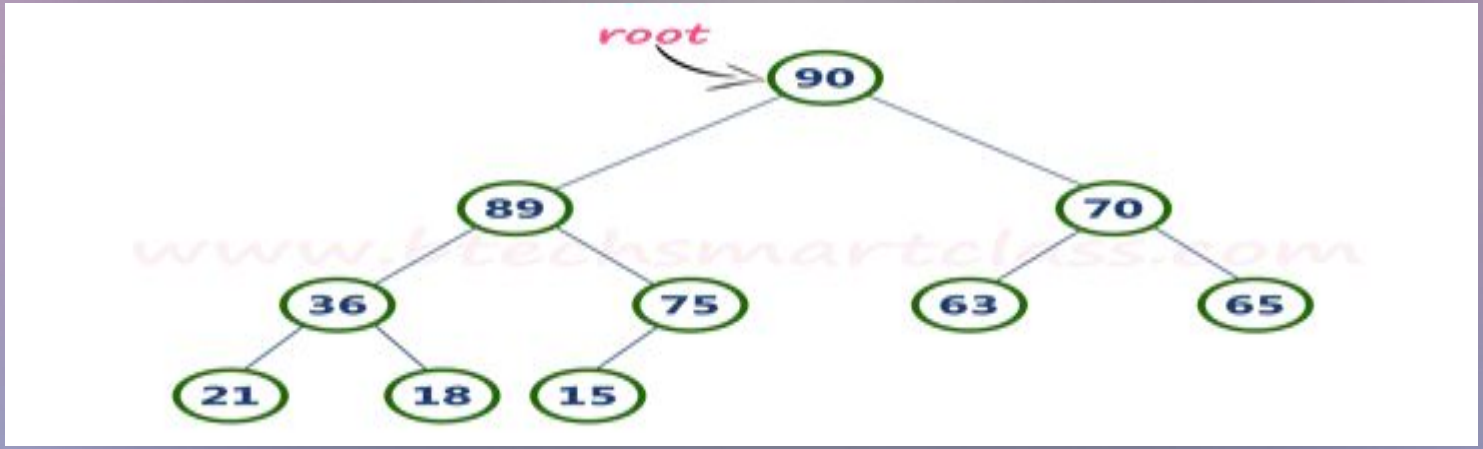
Finding Maximum Value Operation in Max Heap

- Finding the node which has maximum value in a max heap is very simple.
- In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

- Step 1 - Insert the **newNode** as **last leaf** from left to right.
- Step 2 - Compare **newNode value** with its **Parent node**.
- Step 3 - If **newNode value is greater** than its parent, then **swap** both of them.
- Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

Example: Insert 85

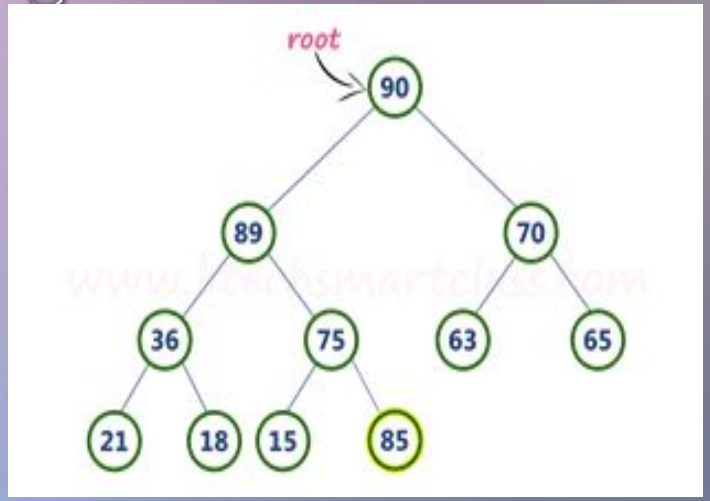


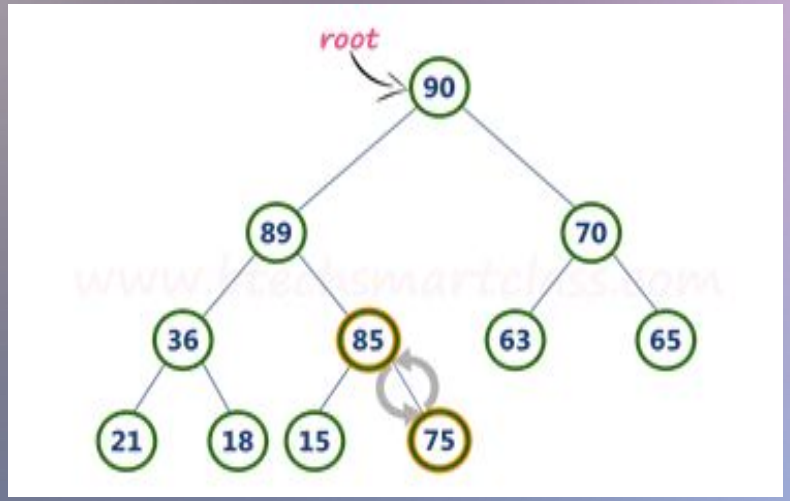
•Step 1: Insert the **newNode** with value 85 as **last leaf** from left to right. newNode is added as a right child of node with value 75.

•Step 2 - Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**

•Step 3 - Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them.

•Step 4 - Now, again compare newNode value (85) with its parent node value (89). Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process.


root
90
89
70
36
75
63
65
21
18
15
85


root
90
89
70
36
85
63
65
21
18
15
75

Deletion of max in Max Heap

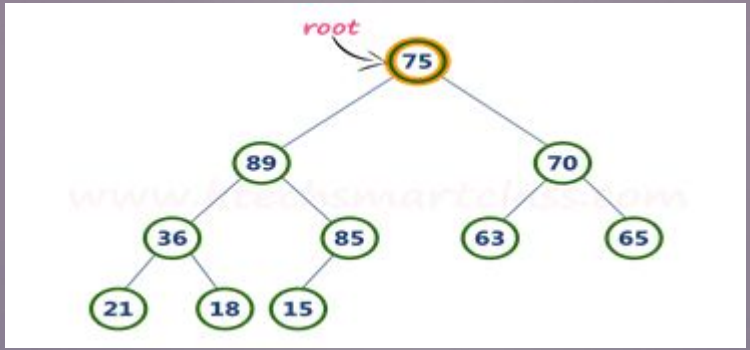
Step 1 - Swap the root node with last node in max heap

Step 2 - Delete last node.

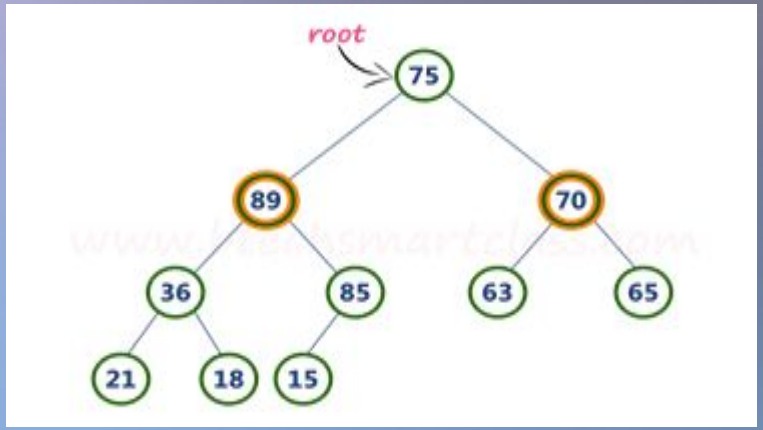
Step 3 - Now, compare root value with its left child value and right child value

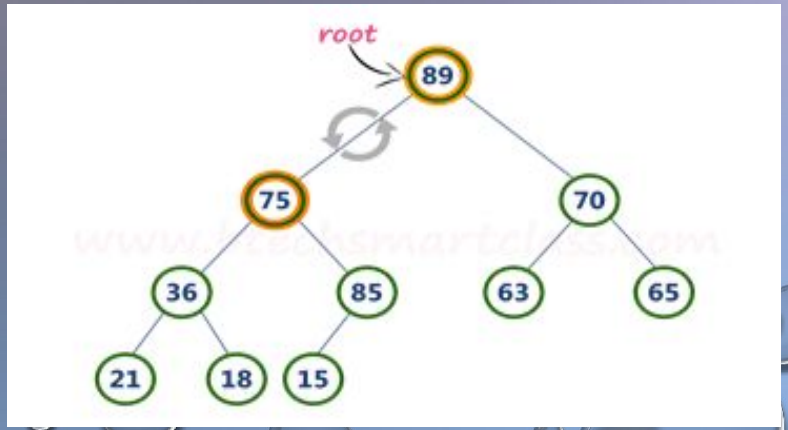
Step 4 - Copy larger child up and go down one level.

Step 5- Done if both children are <= item or reached a leaf node

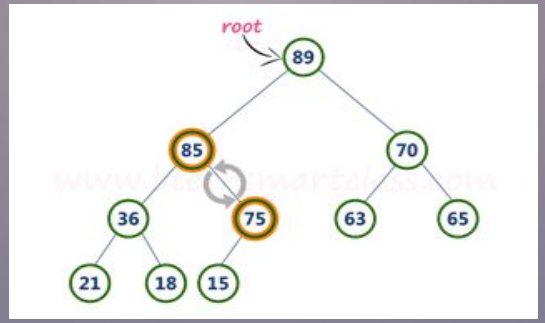


•Step 3 - Compare **root node (75)** with its **left child (89) and right chile(70).** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.

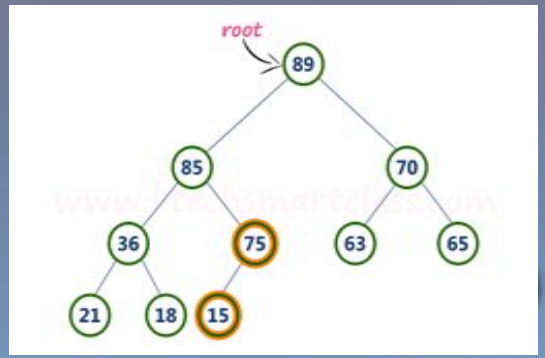




•Step 4 - Now, again compare **75** with its **left child (36) and right child(85.** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them.**)**



•Step 5 - Now, compare node with value **75** with its left child (**15**). Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.



Heap Sort

- Heaps can be used in sorting an array.
- In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.
- Working of Heap Sort:

  Consider an array Arr which is to be sorted using Heap Sort.

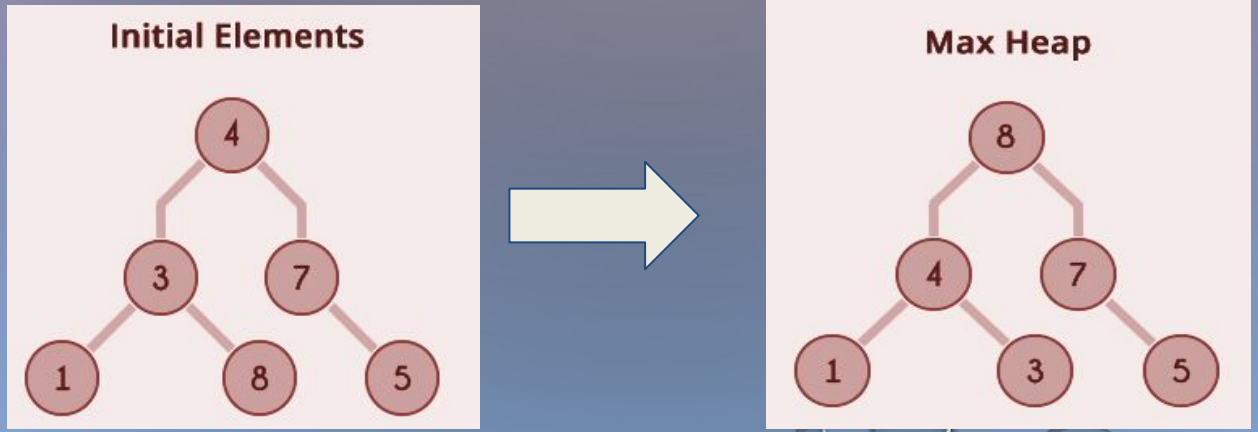
1. Initially build a max heap of elements in Arr.
2. The root element that is Arr[1], will contain maximum element of Arr. After that swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and the decrease the length of heap by one.
3. Repeat step 2, until all the elements are in their correct position.

Example for Heap Sort

- Consider the following unsorted array Arr which is having 6 elements.

Arr		4	3	7	1	8	5
	0	1	2	3	4	5	6

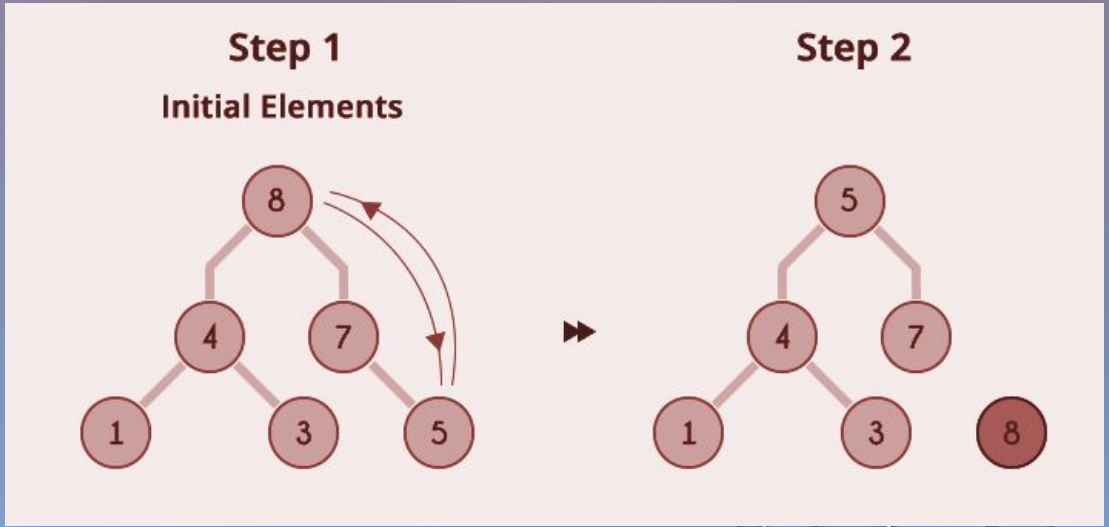
- Now max heap is built.



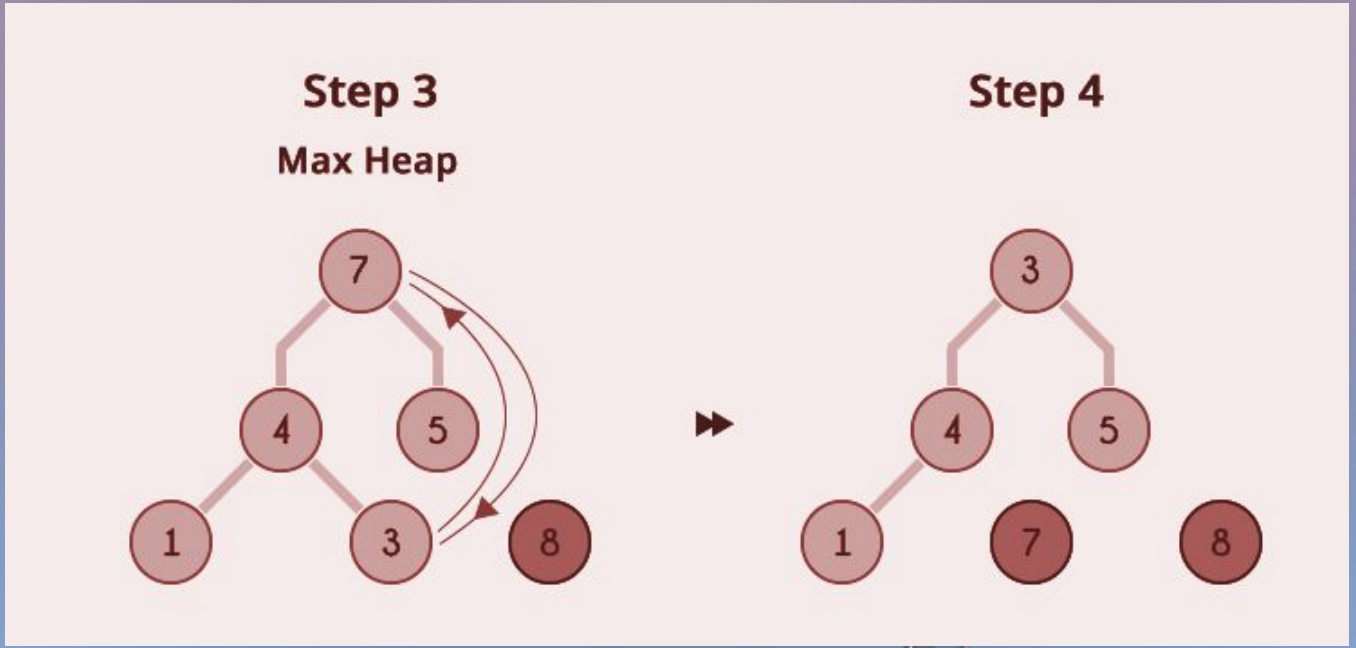
- After building max heap the elements in the Arr will be:

Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

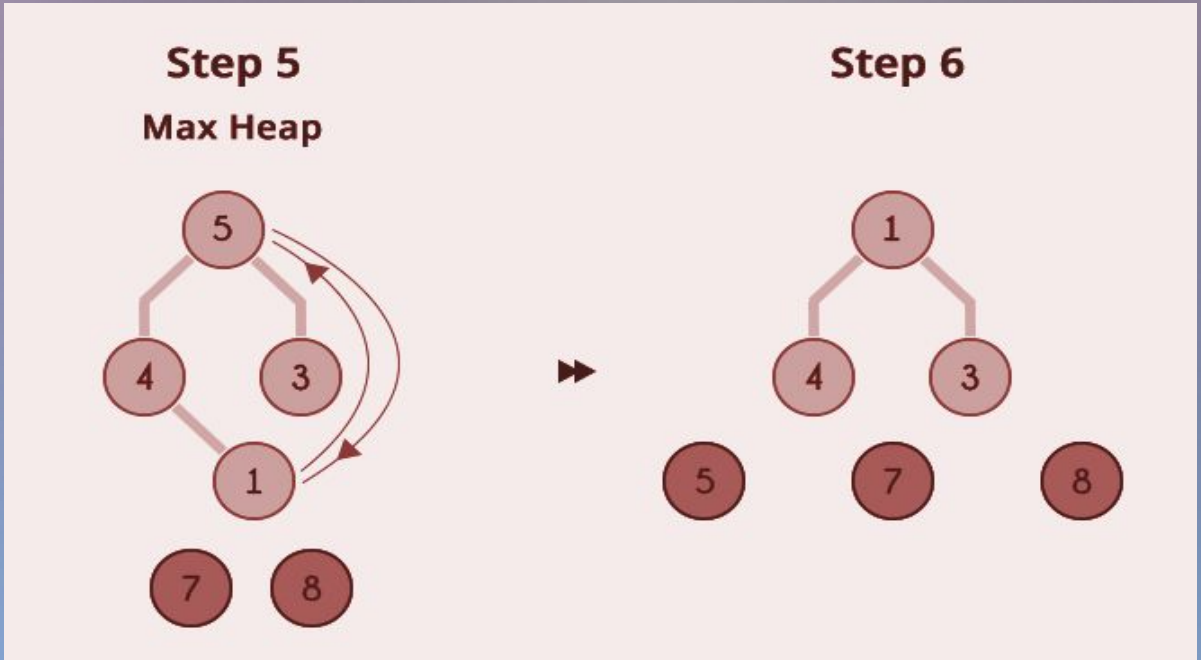
- Step 1: 8 is swapped with 5.
- Step 2: 8 is disconnected from heap as 8 is in correct position now.



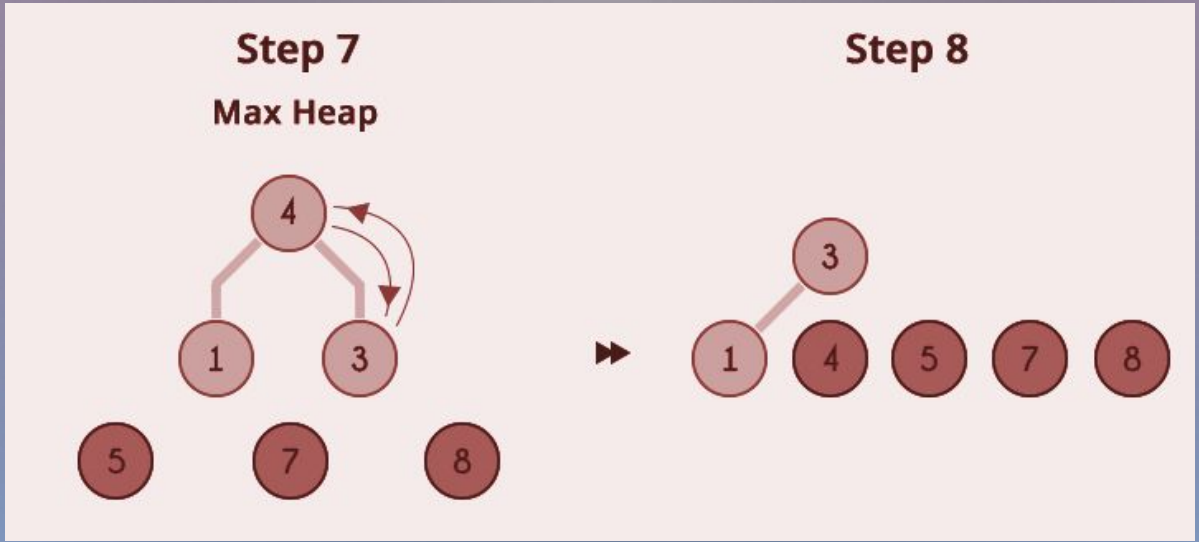
- Step 3: Max-heap is created and 7 is swapped with 3.
- Step 4: 7 is disconnected from heap.



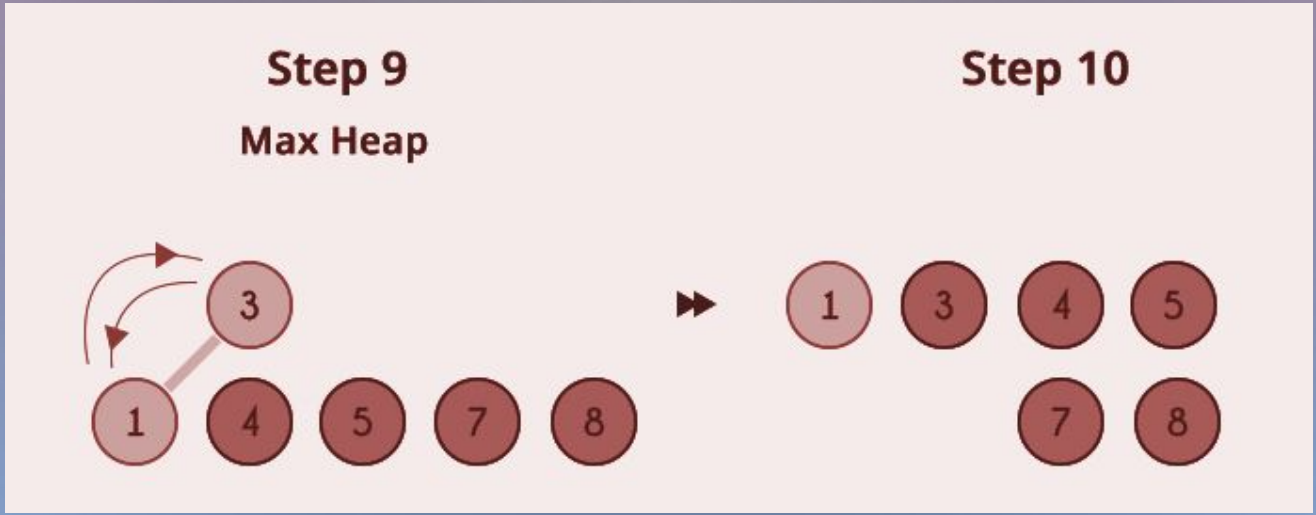
- Step 5: Max heap is created and 5 is swapped with 1.
- Step 6: 5 is disconnected from heap.



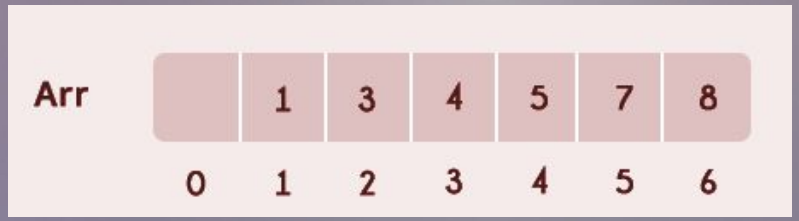
- Step 7: Max heap is created and 4 is swapped with 3.
- Step 8: 4 is disconnected from heap.



- Step 9: Max heap is created and 3 is swapped with 1.
- Step 10: 3 is disconnected.



- After all the steps, we will get a sorted array as shown in the figure below.



Implementation & Time Complexity of Heap Sort

- Here's the pseudo code for the implementation of heap sort:

```
void heap_sort(int Arr[ ])

{
     int heap_size = N;

     build_maxheap(Arr);
     for(int i = N; i >= 2 ; i-- )
     {
          swap|(Arr[ 1 ], Arr[ i ]);
          heap_size = heap_size - 1;
          max_heapify(Arr, 1, heap_size);
     }
}
```

- Complexity:
  Heap Sort has ***O(nlog n)*** time complexities for all the cases (best case, average case, and worst case).

- max_heapify has complexity O(logN) , build_maxheap has complexity O(N) and we run max_heapify N−1 times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).

Heap Sort Applications

- Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort because of the `o(n log n)` upper bound on Heapsort's running time and constant `o(1)` upper bound on its auxiliary storage.
- Although Heap Sort has `o(n log n)` time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be efficiently used if we want to extract the smallest (or largest) from the list of items without the overhead of keeping the remaining items in the sorted order. For e.g Priority Queues.