Heap size = N

For Parent i -

Left child = 2*i+1 and Right child = 2*i+2.

For Child i -

Parent = ceil(i/2)-1

Internal Node range = 0 to (N/2)-1 (Half Point)

Leaves range = (N/2) to (N-1) (Half Point)

Max-Heap = Root > Subtree Nodes

Min-Heap = Root < Subtree Nodes

Heapify Algorithm - (MAX HEAP) (LogN)

ith node to be rearranged to maintain heap property. (Except ith node, entire tree is a heap)

(Check with children and swap)

Heapify(ith node)

Check with children and swap till they reach stopping condition.

Stopping condition -

1. Reach Leaf.
2. Current Node > left child and right child.

```
Max_Heapify(A, i) {
      Left = 2*i+1;
      Right = 2*i+2;
      if(Left <= A.heap_size and A[Left] > A[i]) {
            Largest = Left;
      } else {
            Largest = i;
      }

      if(Right <= A.heap and A[Right] > A[Largest]) {
            Largest = Right;
      }

      if(Largest != i) {
            swap(A[Largest], A[i]);
            Max_Heapify(A, Largest);
      }
}
```

Build Heap - (MAX HEAP)   O(N)
Heap size = N
Internal node range = 0 to (N/2)-1
Leaves range = N/2 to N-1

NOTE: (Leaf Node will always follow Heap property and No need to heapify)
(For applying heapify on ith node, All it's subtrees should be heapified)

Build Heap - Heapify All the internal nodes in Bottom Up (Right Left) Approach.
(Traverse over range of internal nodes and heapify those)

```
Build_MAX_Heap(A) {
      for(i = ((A.length/2)-1) to 0) {
            Max_Heapify(A, i);
      }
}
```

Extract Max - (Remove max element)

```
Heap_Extract_MAX(A) {
      if(A.heap_size < 1) {
            print("Heap Underflow");
      }
      Max = A[0];
      A[0] = A[A.heap_size-1];
      A.heap_size--;
      Max_Heapify(A, 0);
      return max;
}
```

Increase Key for MAX_HEAP - Percolate up.
Decrease Key for MAX_HEAP - Heapify. (Percolate down)

Stopping condition -

1. Parent > Child.
2. I == Root (reached)

```
Max_Heap_Increase_Key(A, i, Key) {
      if(Key < A[i]) {
            print("Wrong operations");
      }
      A[i] = Key;
      while(i > 0 and A[i/2] < A[i]) {
          swap(A[i], A[i/2]);
          i = i/2;
     }
}
```

```
Max_Heap_Decrease_Key(A, i, Key) {
      if(Key > A[i]) {
            print("Wrong operations");
      }
      A[i] = Key;
      Max_Heapify(A, i);
}
```

Insert Element - In a complete binary tree, Insert elements at last level from left to right.

Steps -

A. Insert node at the last position (Left to Right order)
B. Percolate up till - (Same as Increase Key algorithm)
    a. Parent > Current_Node
    b. Currrent_Node is the root of the tree.

```
Heap_Insert_Element(A, val) {
      A.heap_size ++;
      A[A.heap_size-1] = val;
      i = A.heap_size-1;
      while(i > 0 and A[i/2] < A[i]) {
            swap(A[i/2], A[i]);
            i = i/2;
      }
}
```

Heap Sort - (Extract min/max) Heapify Algo.   O(NlogN)

(Given: Max Heap)

(Sort: Sort elements in ASC order)

```
Heap_Sort(A) {
      Build_Heap(A);           //O(N)
      for(i = (A.heap_size-1 to 1)) { //O(N)
            swap(A[i], A[0]);
            Heapify(A[0], i, (Size));     //O(logN)
      }
}
```

Heap Push / Pop   O(LogN)

```
Push(A, element, max_size) {
      if(current_size > max_size) {
            print("Overflow");
      } else {
            A[current_size] = element;
            i = current_size;
            while(i > 0 and A[i/2] < A[i]) {
                  swap(A[i], A[i/2]);
                  i = i/2;
            }
            current_size++;
      }
}
```

```
Pop(A) {
      if(A.heap_size < 0) {
            print("Underflow");
      }
      Int max = A[0];
      A[0] = A[A.heap_size-1];
      A.heap_size--;
      Max_Heapify(A);
      return max;
}
```