HEAP
SESSION - 1

Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

Examples of Min Heap:

```
       10                      10
     /      \                /       \
   20        100          15         30
  /                       / \         /  \
30                     40    50    100    40
```

How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

The traversal method use to achieve Array representation is **Level Order**

Operations on Min Heap:

1) **getMini()** : It returns the root element of Min Heap. Time Complexity of this operation is O(1).

2) **extractMin**(): Removes the minimum element from MinHeap. Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.

3) **decreaseKey():** Decreases value of key. The time complexity of this operation is O(Logn). If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) **insert():** Inserting a new key takes O(Logn) time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) **delete():** Deleting a key also takes O(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After

decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.

Time Complexity of building a heap

Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1
    do HEAPIFY(A, i);
    end for
END
```

A quick look over the above algorithm suggests that the running time is , since each call to **Heapify** costs and **Build-Heap** makes such calls.

This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the running time of **Heapify** depends on the height of the tree 'h' (which is equal to lg(n), where n is number of nodes) and the heights of most sub-trees are small.

The height 'h' increases as we move upwards along the tree. Line-3 of **Build-Heap** runs a loop from the index of the last internal node (heapsize/2) with height=1, to the index of root(1) with height = lg(n). Hence, **Heapify** takes different time for each node, which is .

For finding the Time Complexity of building a heap, we must know the number of nodes having height h.

For this we use the fact that, A heap of size n has at most nodes with height h.

Now to derive the time complexity, we express the total cost of **Build-Heap** as-

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2(). Similarly in Step three, the upper limit of the summation can be increased to infinity since we are using Big-Oh notation.

Sum of infinite G.P. ($x < 1$)

On differentiating both sides and multiplying by x, we get

Putting the result obtained in (3) back in our derivation (1), we get

HeapSort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

```
         4

       /      \

   10          3

   /   \

 5      1
```

The numbers in bracket represent the indices in the array

representation of data.

Applying heapify procedure to index 1:

```
      4
    /   \
   10    3
  /  \
 5    1
```

Applying heapify procedure to index 0:

```
        10
        / \
       5    3
      /  \
     4    1
```

The heapify procedure calls itself recursively to build heap in top down manner.

```
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Heap Sort for decreasing order using min heap

Given an array of elements, sort the array in decreasing order using min heap.

Examples:
Input : arr[] = {5, 3, 10, 1}

Output : arr[] = {10, 5, 3, 1}

Input : arr[] = {1, 50, 100, 25}

Output : arr[] = {100, 50, 25, 1}

Algorithm :

1. Build a min heap from the input data.

2. At this point, the smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

```
void heapify(int arr[], int n, int i)
{
    int smallest = i; // Initialize smalles as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is smaller than root
    if (l < n && arr[l] < arr[smallest])
```

```
      smallest = l;

    // If right child is smaller than smallest so far
    if (r < n && arr[r] < arr[smallest])
        smallest = r;

    // If smallest is not root
    if (smallest != i) {
        swap(arr[i], arr[smallest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, smallest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
```

```
                    heapify(arr, i, 0);
          }
  }

```

Time complexity: It takes **O(logn)** for **heapify** and **O(n)** for **constructing a heap**. Hence, the overall time complexity of **heap sort** using **min heap** or **max heap** is **O(nlogn).**

Iterative HeapSort

HeapSort is a comparison based sorting technique where we first build Max Heap and then swaps the root element with last element (size times) and maintains the heap property each time to finally make it sorted.

Examples:
Input : 10 20 15 17 9 21

Output : 9 10 15 17 20 21

Input: 12 11 13 5 6 7 15 5 19

Output: 5 5 6 7 11 12 13 15 19

In first Example, first we have to build Max Heap.
So, we will start from 20 as child and check for its parent. Here 10 is smaller, so we will swap these two.
Now, 20 10 15 17 9 21

Now, child 17 is greater than its parent 10. So, both will be swapped and order will be
20 17 15 10 9 21

Now, child 21 is greater than parent 15. So, both will be swapped.
20 17 21 10 9 15

Now, again 21 is bigger than parent 20. So,
21 17 20 10 9 15
This is Max Heap.

Now, we have to apply sorting. Here, we have to swap first element with last one and we have to maintain Max Heap property.

So, after first swapping : 15 17 20 10 9 21
It clearly violates Max Heap property. So, we have to maintain it. So, order will be
20 17 15 10 9 21
17 10 15 9 20 21
15 10 9 17 20 21
10 9 15 17 20 21
9 10 15 17 20 21
Here, underlined part is sorted part.

```
void buildMaxHeap(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
            // if child is bigger than parent
            if (arr[i] > arr[(i - 1) / 2])
            {
                int j = i;
              // swap child and parent until
                // parent is smaller
              while (arr[j] > arr[(j - 1) / 2])
              {
                swap(arr[j], arr[(j - 1) / 2]);
                       j = (j - 1) / 2;
                   }
                }
        }
}

void heapSort(int arr[], int n)
{
        buildMaxHeap(arr, n);
        for (int i = n - 1; i > 0; i--)
        {
                // swap value of first indexed
                // with last indexed
                swap(arr[0], arr[i]);
                // maintaining heap property
                // after each swapping
                int j = 0, index;
                do
                {
                    index = (2 * j + 1);
                // if left child is smaller than
            // right child point index variable
                    // to right child
                    if (arr[index] < arr[index + 1] && index < (i - 1))
                        index++;

                    // if parent is smaller than child
                    // then swapping parent with child
                    // having higher value
                    if (arr[j] < arr[index] && index < i)
```

```
                        swap(arr[j], arr[index]);

                    j = index;

                } while (index < i);
            }
        }

```

Here, both function buildMaxHeap and heapSort runs in O(nlogn) time. So, overall time complexity is O(nlogn).