**Why Is heap Preferred Over BST As Priority Queue:**

A typical Priority Queue requires following operations to be efficient.

Get Top Priority Element (Get minimum or maximum)

Insert an element

Remove top priority element

Decrease Key

A Binary Heap supports above operations with following time complexities:

O(1)

O(Logn)

O(Logn)

O(Logn)

**So why is Binary Heap Preferred for Priority Queue?**

Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.

Although operations are of same time complexity, constants in Binary Search Tree are higher.

We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.

Binary Heap doesn’t require extra space for pointers.

Binary Heap is easier to implement.

There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in Θ(1) time

**Is Binary Heap always better?**

Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

Searching an element in self-balancing BST is O(Logn) which is O(n) in Binary Heap.

We can print all elements of BST in sorted order in O(n) time, but Binary Heap requires O(nLogn) time.

Floor and ceil can be found in O(Logn) time.

K’th largest/smallest element be found in O(Logn) time by augmenting tree with an additional field.

**Time Complexity For Building A Heap(this is important. Why is this O(n) instead of O(nlogn))**

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

Now, A quick look over the above algorithm suggests that the running time is O(nlogn)

Since, heapify takes O(logn) and build heap makes O(n) to heapify function.

But, we need deeper look. Since though correct, the upper bound 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 O(h)

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

Now, a heap of height h has atmost 20+21+22+…2(h-1) nodes=2h-1 nodes which is almost equal to 2h nodes.

That means a heap of size n, will have atmost  nodes at height h.

Now, 

Or, 

Or, 

(by applying big O notation rule)

Now, sum of infinite G.P



And, 

Apply these two and get the result.

**Sort An Array Which Was Nearly Sorted:**

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time.

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be O(nk)

We can sort such arrays more efficiently with the help of Heap data structure. Following is the detailed process that uses Heap.

1) Create a Min Heap of size k with first k elements. This will take O(k) time (See this Fact)

2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)\*logK)

O(k) is for building the initial heap. (so, looks like k element)

Now, after that, n-k elements are left. And, we have to remove one from min heap, add one in the min heap.

(initially first k+1 instead of k)

If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This approach is similar to Huffman Coding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes.

Let there be n ropes of lengths stored in an array len[0..n-1]

1) Create a min heap and insert all lengths into the min heap.

2) Do following while number of elements in min heap is not one.

……a) Extract the minimum and second minimum from min heap

……b) Add the above two extracted values and insert the added value to the min-heap.

1. Return the value of only left item in min heap.

**Kth Largest Element Of An Array:**

**(Use Max Heap)**

1) Build a Max Heap tree in O(n)

2) Use Extract Max k times to get k maximum elements from the Max Heap O(klogn)

Time complexity: O(n + klogn)

**(Use Order Statistics)**

1) Use order statistic algorithm to find the kth largest element. Please see the topic selection in worst-case linear time O(n)

2) Use QuickSort Partition algorithm to partition around the kth largest number O(n).

3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.

Time complexity: O(n) if we don’t need the sorted output, otherwise O(n+kLogk)

**Kth Largest Element Of An Already Sorted 2d array: (every row contains k elements)**

Create a binary heap of size k. Now, after creating that, insert the first row. (Now, first row contains smallest element of all columns) . Now, remove the minimum element. Insert an element of same column number but from next row. Do it for k-1 times. Kth time, just remove the element.

**Merge k Sorted Arrays:**

1. Create an output array of size n\*k.

2. Create a min heap of size k and insert 1st element in all the arrays into the heap

3. Repeat following steps n\*k times.

a) Get minimum element from heap (minimum is always at root) and store it in output array.

b) Replace heap root with next element from the array from which the element is extracted. If the array doesn’t have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

**Connect n ropes with minimum cost:**

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.

1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5.

2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.

3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.

**Heap Sort:  
  
// C++ program for implementation of Heap Sort**

**#include <iostream>**

**using namespace std;**

**// To heapify a subtree rooted with node i which is**

**// an index in arr[]. n is size of heap**

**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);**

This itself is build heap

**// 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);**

**}**

**}**

**/\* A utility function to print array of size n \*/**

**void printArray(int arr[], int n)**

**{**

**for (int i=0; i<n; ++i)**

**cout << arr[i] << " ";**

**cout << "\n";**

**}**

**// Driver program**

**int main()**

**{**

**int arr[] = {12, 11, 13, 5, 6, 7};**

**int n = sizeof(arr)/sizeof(arr[0]);**

**heapSort(arr, n);**

**cout << "Sorted array is \n";**

**printArray(arr, n);**

**}**

**Implement Priority Queue Using Heap:**