Heap Sort:

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[*n*] and A[1]: now max element is at the end of the array!

4. Discard node *n* from heap (by decrementing heap-size variable)

5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

6. Go to Step 2 unless heap is empty.

```
Void heap_sort(int A[], int n)
{
        for(int i =n/2; i>=1; i--)      /* Build heap (rearrange elements) */
        heapify(A, i);
        for(int i=n; i>=1; i--)      /* One by one extract an element from heap */
        {
                swap(&A[1], &A[i]);     /* Move current root to end */
                heapify(A, 1);
        }
}
```

Time Complexity:
Time complexity of Max-Heapify() is O(logn). Time complexity of Build-Max-Heap() is O(n) and overall time complexity of Heap Sort is O(nlogn).

the recurrence for heap sort is,

$T(n) = T(n-1) + \log_2(n), T(1) = 0$

$\Rightarrow T(n) = T(n-2) + \log_2(n-1) + \log_2(n)$

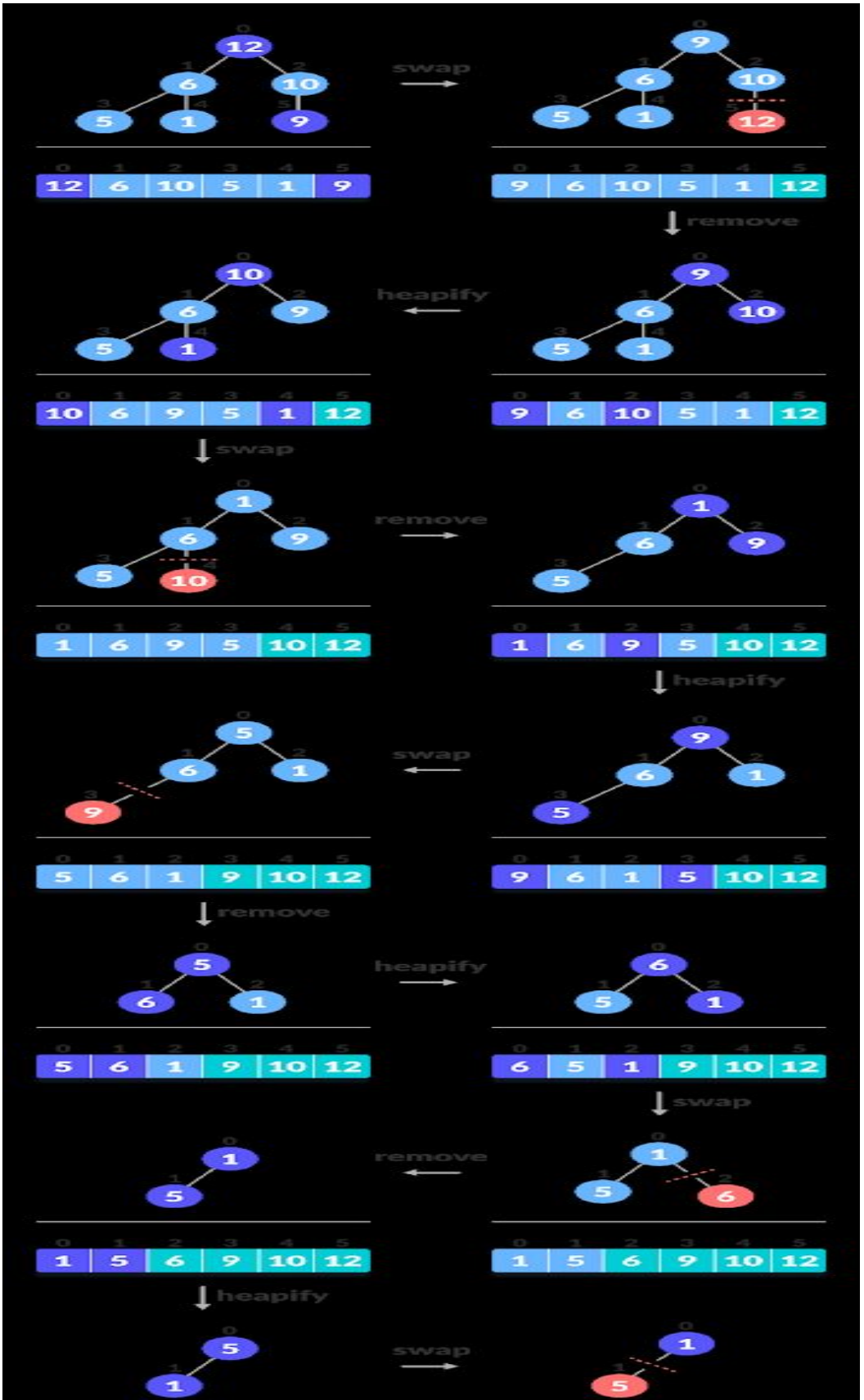
By substituting further, $\Rightarrow T(n) = T(1) + \log 2 + \log 3 + \log 4 + \ldots + \log n$

$\Rightarrow T(n) = \log(2 \cdot 3 \cdot 4 \cdots n)$

$\Rightarrow T(n) = \log(n!)$

$\Rightarrow \log(n!) \leq n\log n$ as $n! \leq n^n$ (Stirling's Approximation)

$\Rightarrow T(n) = O(n\log_2 n)$.


swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap

Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority.

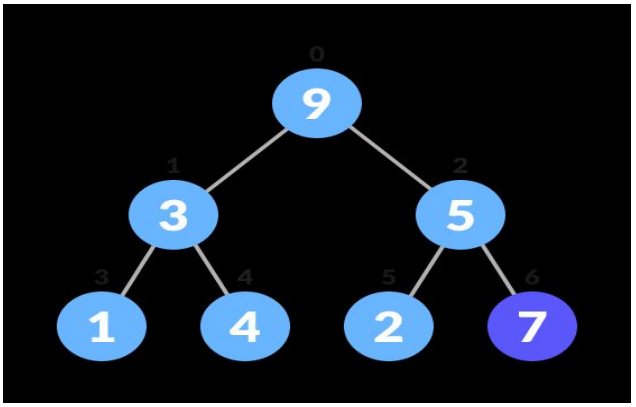
The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

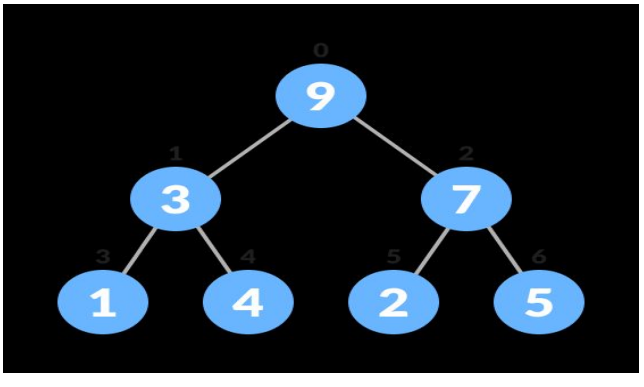
Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Binary Heap: O(1) for peek; O(log n) for insertion; O(log n) for deletion

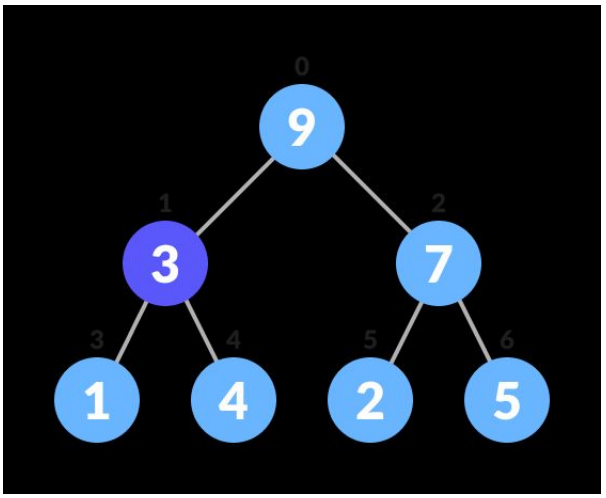
- Insert the new element at the end of the tree.



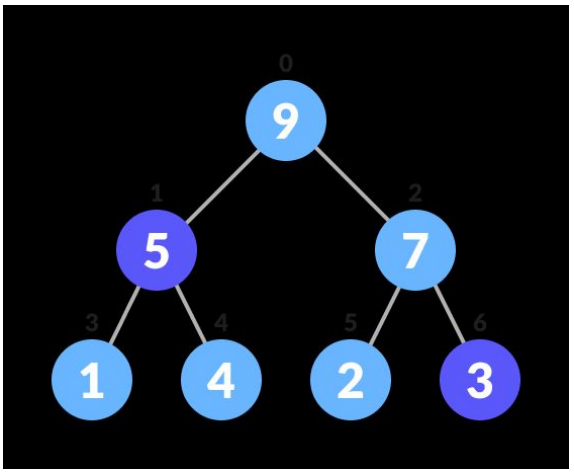
- Heapify the tree.



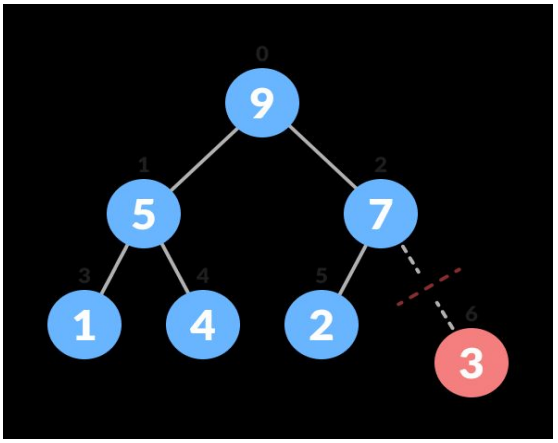
- Select the element to be deleted



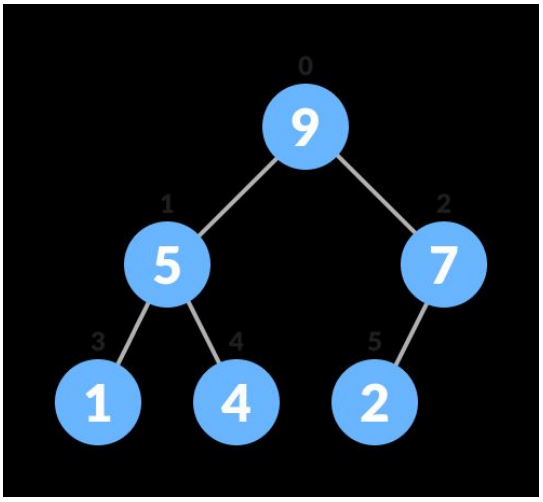
- Swap it with the last element.



- Remove the last element.



- Heapify the tree.



Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

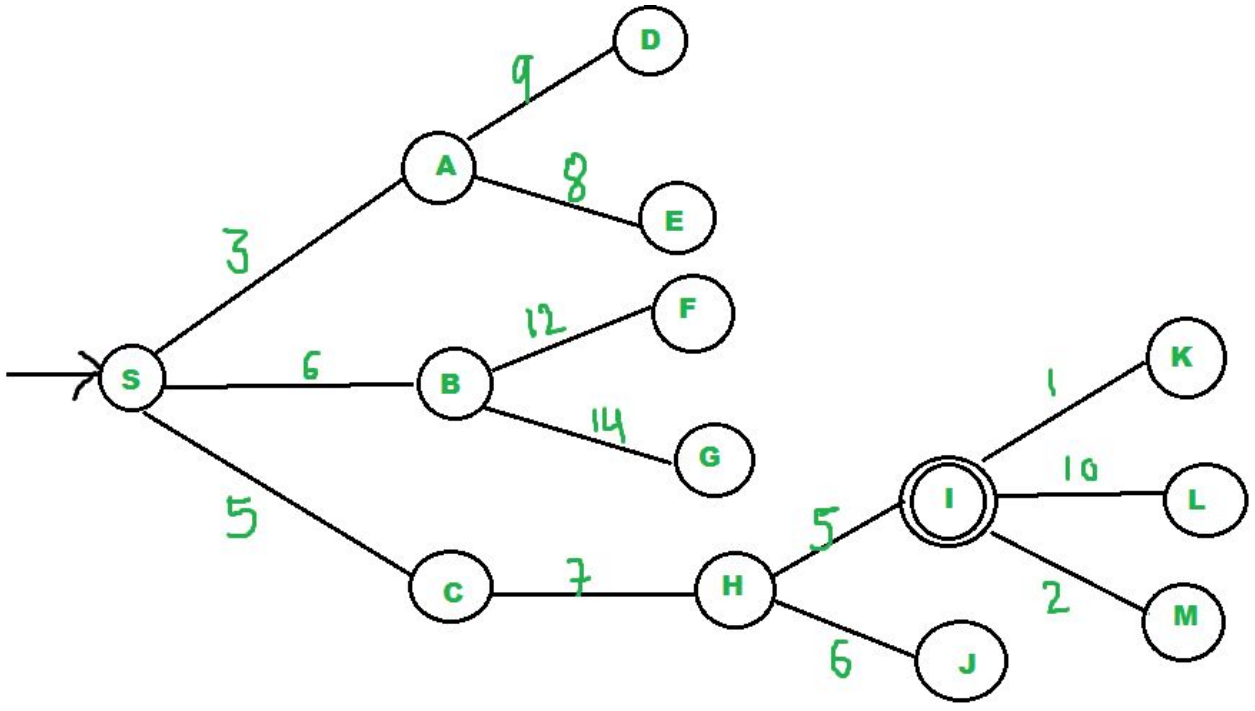
Time complexity:
We can use heaps to implement the priority queue. It will take O(logn) time to insert and delete each element in the priority queue.

Best First Search

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes implementation of best-first search is same as that of breadth First search. We will use the priority queue just like we use a queue for BFS.



Source: S Target: M

step1:

S

step2: output- S

A C B

(S, A)3 < (S, C)5 < (S, B)6

step3: output- S, A

C B **E** D

(S, C)5 < (S, B)6 < (A, E)8 < (A, D)9

step4: output- S, A, C

B **H** E D

(S, B)6 < (C, H)7 < (A, E)8 < (A, D)9

step5: output- S, A, C, B

H E D F G

(C, H)7 < (A, E)8 < (A, D)9 < (B, F)12 < (B, G)14

step6: output- S, A, C, B, H

I **J** E D F G

(H, I)5 < (H, J)6 < (A, E)8 < (A, D)9 < (B, F)12 < (B, G)14

step7: output- S, A, C, B, H, I

K **M** J E D **L** F G

(I, K)1 < (I, M)2 < (H, J)6 < (A, E)8 < (A, D)9 < (I, L)10 < (B, F)12 < (B, G)14

step8: output- S, A, C, B, H, I, K

M J E D **L** F G

step9: output- S, A, C, B, H, I, K, M

J E D **L** F G

step10: STOP

Time Complexity:

The worst case time complexity for Best First Search is O(nlogn) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.

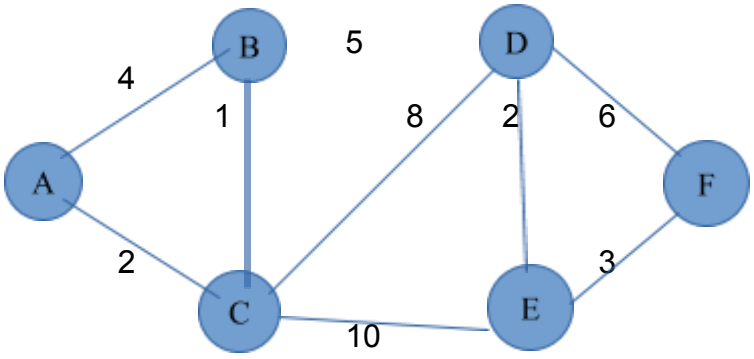
Performance of the algorithm depends on how well the cost or evaluation function is designed.

Bidirectional Search:

Bidirectional search is a graph search algorithm which find smallest path form source to goal vertex. It runs two simultaneous search –

1. Forward search form source/initial vertex toward goal vertex
2. Backward search form goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. The search terminates when two graphs intersect.



Source: A
Target: F

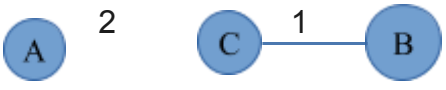
Step1: from **source** from **target**

A **C = 2***
A **B = 4**

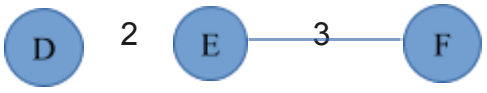
F **E = 3***
F **D = 6**

Step2: from **source** from **target**

C B = 3*
C E = 12
C D = 10



E D = 5*
E C = 13



Step3: from **source** from **target**

B ——— D = 8* D ——— B = 10*
D C = 13

D is vertex of sub graph from **target** B is vertex of sub graph from **source**

two sub graph are merged (vertex B, D are common in two sub graph)

Shortest Path cost = 13

if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity would be $O(b^d)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$for each search and total complexity would be $O(b^{d/2} + b^{d/2})$which is far less than $O(b^d)$.

- Completeness: Bidirectional search is complete if BFS is used in both searches.
- Optimality: It is optimal if BFS is used for search and paths have uniform cost.
- Time and Space Complexity: Time and space complexity is $O(b^{d/2})$.