Heap Data Structure

Heap

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, if X is the parent node of Y then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.

The maximum number of children of a node in a heap depends on the type of heap.

There can be two types of heap:

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Min Heap: In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.

Binary Heap

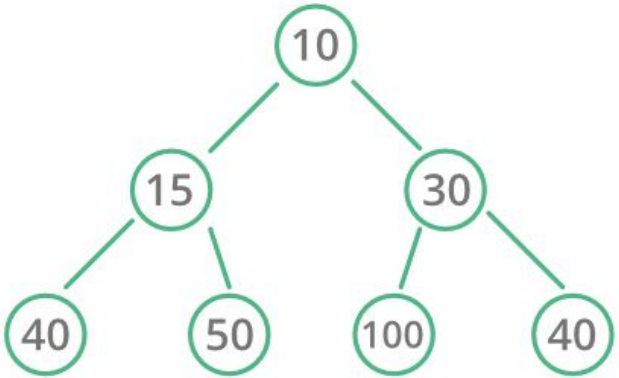
A binary heap is defined as a binary tree with two additional constraints:

- Shape property: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.

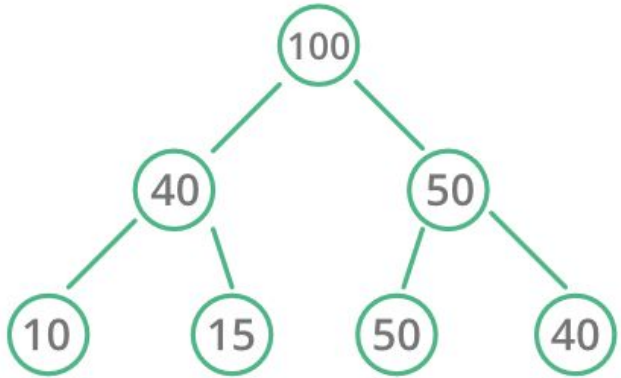
Heaps where the parent key is greater than or equal to (≥) the child keys are called *max-heaps*; those where it is less than or equal to (≤) are called *min-heaps*. Efficient (logarithmic time) algorithms are known for the two operations needed to implement a priority queue on a binary heap: inserting an element, and removing the smallest or largest element from a min-heap or max-heap, respectively. Binary heaps are also commonly employed in the heapsort sorting algorithm, which is an in-place algorithm because binary heaps can be implemented as an implicit data structure, storing keys in an array and using their relative positions within that array to represent child-parent relationships.

.

Heap Data Structure



Min Heap



Max Heap

Binary Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take O(log *n*) time.

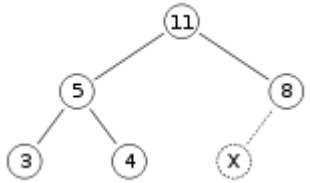
Insert:

To add an element to a heap, we can perform this algorithm:

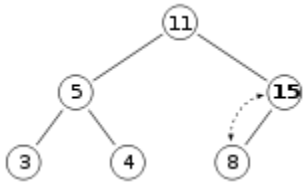
1. Add the element to the bottom level of the heap at the leftmost open space.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

Steps 2 and 3, which restore the heap property by comparing and possibly swapping a node with its parent, are called *the* ***up-heap*** operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *swim-up*, *heapify-up*, or *cascade-up*).

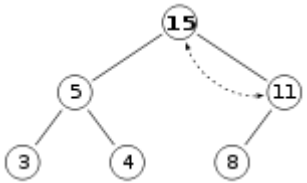
As an example of binary heap insertion, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



However the heap property is still violated since 15 > 11, so we need to swap again:



which is a valid max-heap.

Extract

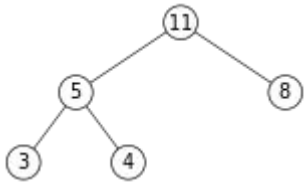
The procedure for **deleting the root from the heap** (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) while retaining the heap property is as follows:

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

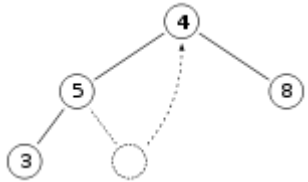
Steps 2 and 3, which restore the heap property by comparing and possibly swapping a node with one of its children, are called the ***down-heap*** (also known as *bubble-down*, *percolate-down*, *sift-down*, *sink-down*, *trickle down*, *heapify-down*, *cascade-down*) operation.

- **Extract_Max performs on Max heap**
- **Extract_Min performs on Min heap**

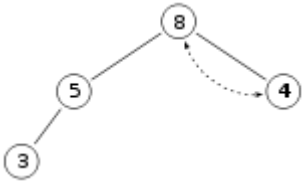
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position.

Decrease key or increase key

The decrease key operation replaces the value of a node with a given lower value, and the increase key operation does the same but with a higher value. This involves finding the node with the given value, changing the value, and then down-heapifying or up-heapifying to restore the heap property. Takes O(log n) time.

Decrease key can be done as follows:

1. Find the index of the element we want to modify
2. Decrease the value of the node
3. Down-heapify (assuming a max heap) to restore the heap property

Increase key can be done as follows:

1. Find the index of the element we want to modify
2. Increase the value of the node
3. Up-heapify (assuming a max heap) to restore the heap property

Search

Finding an arbitrary element takes O(n) time.

Delete

Deleting a key also takes O(Log n) time. In the case of a min heap,we replace the key to be deleted with minus infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.

Likewise, In the case of a max heap,we replace the key to be deleted with plus infinite by calling increaseKey(). After increaseKey(), the plus infinite value must reach root, so we call extractMax() to remove the key.

getMax or getMin

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

- getMax performs on Max heap
- getMin performs on Min heap

APPLICATIONS:

1) **Heap Sort:**

We can use heaps in sorting the elements in a specific order in efficient time.

Let’s say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.

Idea: We build the max heap of elements stored in Arr and the maximum element of Arr will always be at the root of the heap.

Leveraging this idea we can sort an array in the following manner.

Processing:

- Initially we will build a max heap of elements in Arr.
- Now the root element that is Arr[1] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will decrease the length of heap by one.
- We will repeat the step 2, until we get all the elements in their correct position.
- We will get a sorted array.

Implementation: Suppose there are N elements stored in array Arr

```
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);
     }
    }
```

2) **Priority Queue:**

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like **Dijkstra's Shortest Path** and **Prim's Minimum Spanning Tree.**

4) Many problems can be efficiently solved using Heaps. example.

a) **K'th Largest Element in an array.**

b) **Sort an almost sorted array**.

c) **Merge K Sorted Arrays.**