Binary Heap

*A **Binary Heap** is a **[complete Binary Tree](#)** which is used to store data efficiently to get the max or min element based on its structure.*

A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at the 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
```

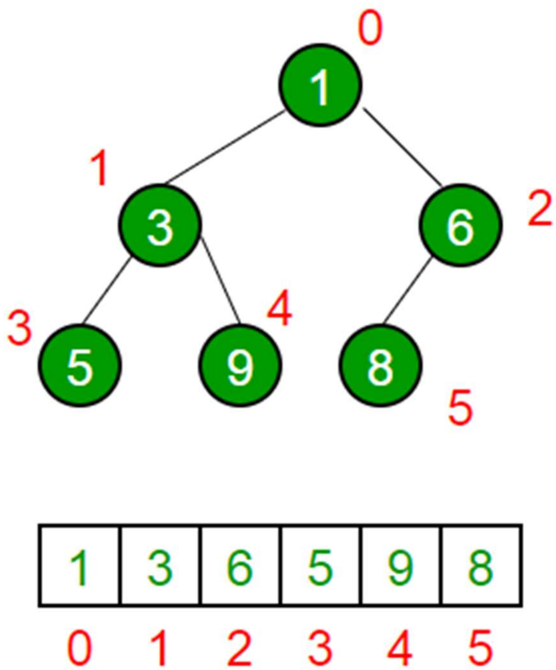
How is Binary Heap represented?

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].
- The below table shows indices 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 Traversal**. Please refer to Array Representation Of Binary Heap for details.



Operations on Heap:

Below are some standard operations on min heap:

- **getMin():** It returns the root element of Min Heap. The time Complexity of this operation is **O(1)**. In case of a maxheap it would be **getMax()**.
- **extractMin():** Removes the minimum element from MinHeap. The time Complexity of this Operation is **O(log N)** as this operation needs to maintain the heap property (by calling **heapify()**) after removing the root.
- **decreaseKey():** Decreases the value of the key. The time complexity of this operation is **O(log N)**. If the decreased 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.
- **insert():** Inserting a new key takes **O(log N)** time. We add a new key at the end of the tree. If the 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.

- **delete():** Deleting a key also takes **O(log N)** time. We replace the key to be deleted with the minimum infinite by calling **decreaseKey()**. After decreaseKey(), the minus infinite value must reach root, so we call **extractMin()** to remove the key.

Below is the implementation of basic heap operations.

- C++
- Java
- Python
- C#
- Javascript

```
// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    MinHeap(int capacity);

    // to heapify a subtree with the root at given index
```

```
    void MinHeapify(int i);

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to extract the root which is the minimum element
    int extractMin();

    // Decreases key value of key at index i to new_val
    void decreaseKey(int i, int new_val);

    // Returns the minimum key (key at root) from min heap
    int getMin() { return harr[0]; }

    // Deletes a key stored at index i
    void deleteKey(int i);

    // Inserts a new key 'k'
    void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
```

```
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
```

```
}

// Decreases value of key at index 'i' to new_val.  It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
```

```
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}



// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}

// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
```

```
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Driver program to test above functions
int main()
{
    MinHeap h(11);
    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    cout << h.extractMin() << " ";
```

```
    cout << h.getMin() << " ";
    h.decreaseKey(2, 1);
    cout << h.getMin();
    return 0;
}
```

Learn Data Structures & Algorithms with GeeksforGeeks

Output

```
2 4 1
```

Applications of Heaps:

- Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
- Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
- Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.
- Many problems can be efficiently solved using Heaps. See following for example. a) K'th Largest Element in an array. b) Sort an almost sorted array/ c) Merge K Sorted Arrays.