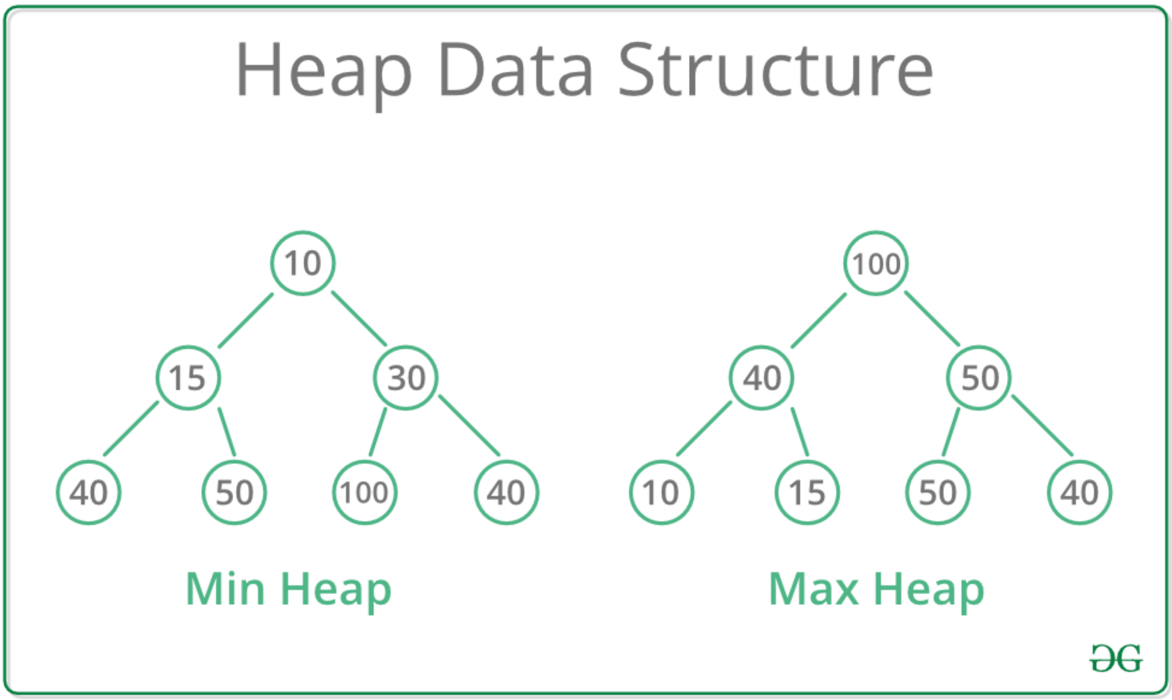
Heap Data Structure

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



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 $O(nlg(n))$, since each call to **Heapify** costs $O(lg(n))$ and **Build-Heap** makes $O(n)$ 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 $O(h)$.

$$
\begin{aligned}
&= O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) \\
&= O(n * 2) \\
&= O(n)
\end{aligned}
$$

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

$$
\begin{aligned}
T(n) &= \sum_{h=0}^{lg(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h) \\
&= O(n * \sum_{h=0}^{lg(n)} \frac{h}{2^h}) \\
&= O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})
\end{aligned}
\tag{1}
$$

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2($2^{h+1} = 2.2^h$). 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)

$$
\sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \tag{2}
$$

On differentiating both sides and multiplying by x, we get

$$
\sum_{n=0}^{\infty} n x^n = \frac{x}{(1-x)^2} \tag{3}
$$

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

$$
\begin{aligned}
&= O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) \\
&= O(n * 2) \\
&= O(n)
\end{aligned}
$$

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

class MinHeap {
    public:
        int *harr;
        int capacity;
        int heap_size;

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

        void linearSearch(int val) {

            for (int i=0; i<heap_size; i++) {
                if (harr[i]==val) {
                    cout<<"Value found"<<endl;
                    return;
                }
            }
            cout<<"Value not found!"<<endl;
        }

        PrintHeap() {
            for (int i=0; i<heap_size; i++) {
                cout<<harr[i]<<" ";
            }
            cout<<endl;
        }

        int getMin() {
            return harr[0];
        }

        int left(int i) {
            return 2*i+1;
        }

        int right(int i) {
            return 2*i+2;
        }

        int height() {
            return ceil(log2(heap_size+1))-1;
        }
```

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

void insertKey(int val) {
    if (heap_size==capacity) {
        cout<<"Memory Overflow!!"<<endl;
        return;
    }
    heap_size++;
    int i=heap_size-1;
    harr[i]=val;
    while (i and harr[parent(i)]>harr[i]) {
        swap(harr[i], harr[parent(i)]);
        i=parent(i);
    }
}

void MinHeapify(int i) {

    int l=left(i);
    int r=right(i);
    int smallest=i;
    if (l<heap_size and harr[l]<harr[smallest]) smallest=l;
    if (r<heap_size and harr[r]<harr[smallest]) smallest=r;
    if (smallest!=i) {
        swap(harr[i], harr[smallest]);
        MinHeapify(smallest);
    }
}

int extractMin() {
    if (heap_size<=0) return INT_MAX;
    if (heap_size==1) {
        heap_size--;
        return harr[0];
    }
    int root=harr[0];
    harr[0]=harr[heap_size-1];
    heap_size--;
    MinHeapify(0);
    return root;
}

void decreaseKey(int i, int new_val) {
    harr[i]=new_val;
    while (i and harr[parent(i)]>harr[i]) {
```

```
                swap(harr[i], harr[parent(i)]);
                i=parent(i);
            }
        }

        void deleteKey(int i) {
            decreaseKey(i, INT_MIN);
            extractMin();
        }
};

int main() {
    cout<<"Enter Size";
    int size, option;
    cin>>size;
    MinHeap obj(size);

    cout<<"\tPress the button below to call the following functions. Press 0 t
o exit."<<endl;
    cout<<"\t1: insertKey()"<<endl;
    cout<<"\t2: searchKey()"<<endl;
    cout<<"\t3: deleteKey()"<<endl;
    cout<<"\t4: getMin()"<<endl;
    cout<<"\t5: Extract Min "<<endl;
    cout<<"\t6: getHeight() "<<endl;
    cout<<"\t7: PrintHeap()"<<endl;
    do {
        cin>>option;

        switch(option) {

            case 0: break;
            case 1: {
                cout<<"insertKey() function called"<<endl;
                cout<<"Enter the value to be inserted : ";
                int val;
                cin>>val;
                obj.insertKey(val);
                cout<<val<<" is added successfully"<<endl;
                break;
            }
            case 2: {
                // We can just apply bia=nary search algorithm to search for a
 specific key
                break;
            }
            case 3: {
                cout<<"Enter index to be deleted :";
```

```
                int in;
                cin>>in;
                obj.deleteKey(in);
                break;
            }
            case 4: {
                cout<<"Minimum value : "<<obj.getMin()<<endl;
                break;
            }
            case 5: {
                cout<<"Extracted Min. Value : ";
                cout<<obj.extractMin()<<endl;
                break;
            }
            case 6 : {

                break;
            }
            case 7: {
                cout<<"Current Heap: ";
                obj.PrintHeap();
                break;
            }
            default : {
                cout<<"Enter proper option number\n\n";
                break;
            }
        }
    } while (option!=0);

    return 0;
}
```