A Binary Heap is a data structure that takes the form of a binary tree. They are a useful way to implement priority queues, sorting algorithms and doing comparative operations, as their time complexity for inserting and deleting nodes is of O(log n). However, they will have a time complexity of O(n) when searching elements, because they will have to traverse through the tree to find said elements. On binary heaps, the parent node can be greater than or equal to the child nodes, and it is called a max-heap. Whereas if the parent node is less than or equal to the child nodes is called min-heap.

A max-heap is usually represented as an array, which we will be implementing below:

In [4]:

```
import sys

class MaxHeap():
    def __init__(self, maximum):
        self.maximum = maximum
        self.size = 0
        self.Heap = [0] * (self.maximum + 1)
        self.Heap[0] = sys.maxsize
        self.FRONT = 1
# The constructor will initialise the heap and give it a maximum size defined by the user
# Then, it sets the size as well as the rest of the heap components to 0.
# To simplify the calculations, we start indexing from 1 instead of 0,
# by filling the 0th element of the heap with a garbage value.

    def parent(self, position):
        return position // 2
# This method will return the position of a current node's parent node.
# The user only needs to provide the current node's position to obtain its parent's.
# This method is rather simple because of the way we are indexing our parent and children nodes.

    def left_child(self, position):
        return 2 * position
# This method returns the position of of a current node's left child.
# Alike parent(), the user provides the current node's position to obtain its left child's.

    def right_child(self, position):
        return (2 * position) + 1
# This method returns the position of of a current node's right child.

    def check_leaf(self, position):
```

```
        if position >= (self.size // 2) and position <= self.size:
            return True
        return False
# The method above checks whether a passed node is a leaf node(a childless node).
# When we look at the heap as an array, all the nodes in the second half are leaf nodes,
# which makes it easier to verify, because if the position passed is greater than or equal to half of the size
# of the heap and less than or equal to size of the heap, then it is a leaf node.

    def swap(self, fposition, sposition):
        self.Heap[fposition], self.Heap[sposition] = self.Heap[sposition], self.Heap[fposition]
# Swap() will make two nodes of the heap switch positions.

    def max_heapify(self, position):
        if not self.check_leaf(position): # If not a leaf node
            if (self.Heap[position] < self.Heap[self.left_child(position)] or
                    self.Heap[position] < self.Heap[self.right_child(position)]): # and smaller than any of the ch
                # Swap with the left child and heapify the left child
                if self.Heap[self.left_child(position)] > self.Heap[self.right_child(position)]:
                    self.swap(position, self.left_child(position))
                    self.max_heapify(self.left_child(position))

                # Swap with the right child and heapify the right child
                else:
                    self.swap(position, self.right_child(position))
                    self.max_heapify(self.right_child(position))
# max_heapify() will be called whenever the heap property is unbalanced, so that it can restore the heap's balance
# Firstly, the method checks whether the relevant node is a leaf node, and if so, it leaves it as is.
# Otherwise, if the node is smaller than its children, the method will check which child is the largest and make a
# As there is a chance the heap will become unbalanced after performing such operation, the method will be operati

    def insert(self, element):
        if self.size >= self.maximum:
            return
        self.size += 1
        self.Heap[self.size] = element
        current = self.size
        while self.Heap[current] > self.Heap[self.parent(current)]:
            self.swap(current, self.parent(current))
            current = self.parent(current)
# Now, we create insert().
# For starters, the method will increase the size of the heap by 1.
```

```
# Then, it will add the new element at the end of the heap, which may unbalance the heap.
# Because of that, the method will compare its value with its parent's and keep swapping it,
# while its parent node is smaller than it.

    def get_heap(self):
        for i in range(1, (self.size // 2) + 1):
            print(" PARENT: " + str(self.Heap[i]) +
                  " LEFT CHILD: " + str(self.Heap[2 *i]) +
                  " RIGHT CHILD: " + str(self.Heap[2 * i + 1]))
# The above method outputs the contents of the heap in a specific format

    def pop_max(self):
        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size -= 1
        self.max_heapify(self.FRONT)
        return popped
# The last method to be built is pop_max(), and it will remove and output the maximum element from the heap, which
# The method will copy the element at the heap's end into the root node and delete the last node, which will make
# In the end, the method will call max_heapify() on the root node to restore balance.
```

And now, we test our blueprint:

In [5]:

```
if __name__ == "__main__":

    new_heap = MaxHeap(15) # created a new obj 'new_heap' and defined its maximum size to 15

    new_heap.insert(5)
    new_heap.insert(3)
    new_heap.insert(17)
    new_heap.insert(10)
    new_heap.insert(84)
    new_heap.insert(19)
    new_heap.insert(6)
    new_heap.insert(22)
    new_heap.insert(9)

    new_heap.get_heap()
```

```
PARENT: 84 LEFT CHILD: 22 RIGHT CHILD: 19
PARENT: 22 LEFT CHILD: 17 RIGHT CHILD: 10
PARENT: 19 LEFT CHILD: 5 RIGHT CHILD: 6
PARENT: 17 LEFT CHILD: 3 RIGHT CHILD: 9
```

In [6]:

```
print('The maximum value of my new heap is ' + str(new_heap.pop_max()))
```

```
The maximum value of my new heap is 84
```

In [7]:

```
new_heap.get_heap() # will return how the heap looks like after popping 84(max value)
```

```
PARENT: 22 LEFT CHILD: 17 RIGHT CHILD: 19
PARENT: 17 LEFT CHILD: 9 RIGHT CHILD: 10
PARENT: 19 LEFT CHILD: 5 RIGHT CHILD: 6
PARENT: 9 LEFT CHILD: 3 RIGHT CHILD: 9
```

In [8]:

```
new_heap.insert(100)
new_heap.get_heap()
```

```
PARENT: 100 LEFT CHILD: 22 RIGHT CHILD: 19
PARENT: 22 LEFT CHILD: 17 RIGHT CHILD: 10
PARENT: 19 LEFT CHILD: 5 RIGHT CHILD: 6
PARENT: 17 LEFT CHILD: 3 RIGHT CHILD: 9
```

In [9]:

```
print(new_heap.Heap[0])
```

```
9223372036854775807
```