Max Heap (Implementation)

Let's implement a max Heap!

We'll cover the following

Max-heap Implementation

Let's start with some function declarations for the heap class. The `__percolateUp()` function is meant to restore the heap property going up from a node to the root. The `__maxHeapify()` function restores the heap property starting from a given node down to the leaves. The two underscores before the `__percolateUp()` and `__maxHeapify()` functions imply that these functions should be treated as private functions although there is no actual way to *enforce* class function privacy in Python. You can still call these functions by prepending `_className` like so, `heap._maxHeap__percolateUp(index)`.

```python
class MaxHeap:
    def __init__(self):
        pass

    def insert(self, val):
        pass

    def getMax(self):
        pass

    def removeMax(self):
        pass

    def __percolateUp(self, index):
        pass

    def __maxHeapify(self, index):
        pass


heap = MaxHeap()

```

Implementing the constructor

The constructor will initialize a list that will contain the values of the heap.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        pass

    def getMax(self):
        pass

    def removeMax(self):
        pass

    def __percolateUp(self, index):
        pass

    def __maxHeapify(self, index):
        pass


heap = MaxHeap()

```

Implementing the `insert()` function

This function appends the given value to the `heap` list and calls the `__percolateUp()` function on it. This function will swap the values at parent-child nodes until the heap property is restored. The time complexity of this function is in $O(log(n))$ because that is the maximum number of nodes that would have to be traversed and/or swapped.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        pass

    def removeMax(self):
        pass

    def __percolateUp(self, index):
        pass

    def __maxHeapify(self, index):
        pass


heap = MaxHeap()
```

Implementing the `getMax()` function

This function returns the maximum value in the heap which is the root, i.e., the first value in the list. It does not modify the heap itself. The time complexity of this function is in $O(1)$ constant time which is what makes heaps so special!

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        pass

    def __percolateUp(self, index):
        pass

    def __maxHeapify(self, index):
        pass


heap = MaxHeap()
```

Implementing the `removeMax()` function

This function removes and returns the maximum value in the heap. It first checks if the length of the heap is greater than 1, if it is, it saves the maximum value in a variable, swaps the maximum value with the last leaf, deletes it, and restores the max heap property on the rest of the tree by calling the `__maxHeapify()` function on it. The function then checks if the heap is of size 1, if it is, it saves the maximum value in the tree (the only value really) in a variable, deletes it, and returns it. Then it checks if the heap is empty and returns `None` if it is. The time complexity of this function is in $O(log(n))$ because that is the maximum number of nodes that would have to be traversed and/or swapped.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
```

```
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        if len(self.heap) > 1:
            max = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
        elif len(self.heap) == 1:
            max = self.heap[0]
            del self.heap[0]
            return max
        else:
            return None

    def __percolateUp(self, index):
        pass

    def __maxHeapify(self, index):
```

Implementing the `__percolateUp()` function

This function restores the heap property by swapping the value at a parent node if it is less than the value at a child node. After swapping, the function is called recursively on each parent node until the root is reached. The time complexity of this function is in $O(log(n))$ because that is the maximum number of nodes that would have to be traversed and/or swapped.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        if len(self.heap) > 1:
            max = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
        elif len(self.heap) == 1:
            max = self.heap[0]
            del self.heap[0]
            return max
        else:
            return None

    def __percolateUp(self, index):
        parent = (index-1)//2
        if index <= 0:
            return
```

Implementing the `__maxHeapify()` function

This function restores the heap property after a node is removed. It swaps the values of the parent nodes with the values of their largest child nodes until the heap property is restored. The time complexity of this function is in $O(log(n))$ because that is the maximum number of nodes that would have to be traversed and/or swapped.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        if len(self.heap) > 1:
            max = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
        elif len(self.heap) == 1:
            max = self.heap[0]
            del self.heap[0]
            return max
        else:
            return None

    def __percolateUp(self, index):
        parent = (index-1)//2
        if index <= 0:
            return
```

Implementing the `buildHeap()` function

This function restores creates a heap from a list passed as an argument. It calls `_maxHeapify` method at every index starting from the last index of the list building a heap.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        if len(self.heap) > 1:
            max = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
```

```
20              return max
21          elif len(self.heap) == 1:
22              max = self.heap[0]
23              del self.heap[0]
24              return max
25          else:
26              return None
27  
28      def __percolateUp(self, index):
29          parent = (index-1)//2
30          if index <= 0:
31              return
```

Let's derive a tight bound for the complexity of building a heap.

Notice that we start from the bottom of the heap, i.e., `range(len(arr)-1,-1,-1)` (**line 54**). The number of comparisons for a particular node at height h is $O(h)$. Also, the number of nodes at height 0 is at most $\lceil \frac{n}{2} \rceil$, that at height 1 is $\lceil \frac{n}{4} \rceil$ and so on. In general, the number of nodes at height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.

Thus, for a heap with n nodes, that has a height of $log(n)$, the running time of bottom-up heap construction is:

$$T(h) = \sum_{i=0}^{log(n)} \lceil \frac{n}{2^{i+1}} \rceil O(i)$$

Now, $\lceil \frac{n}{2^{i+1}} \rceil < \frac{n}{2^i}$ (reducing the denominator increases the value). Thus, we can write:

$$T(h) \leq \sum_{i=0}^{log(n)} \frac{n}{2^i} O(i)$$

Or, $T(h) = O(\sum_{i=0}^{log(n)} \frac{i \times n}{2^i}) = O(n \sum_{i=0}^{log(n)} \frac{i}{2^i})$

The above summation is upper bounded by the corresponding infinite series, thus:

$$T(h) = O(n \sum_{i=0}^{\infty} \frac{i}{2^i})$$

The sum of the above infinie series is known to be approximately 2. Thus:

$$T(h) = O(2n) = O(n)$$

A complete implementation of `MaxHeap`:

MaxHeap.py

```
1  class MaxHeap:
2      def __init__(self):
3          self.heap = []
4  
5      def insert(self, val):
6          self.heap.append(val)
7          self.__percolateUp(len(self.heap)-1)
8  
9      def getMax(self):
10         if self.heap:
11             return self.heap[0]
12         return None
13 
14     def removeMax(self):
15         if len(self.heap) > 1:
16             max = self.heap[0]
17             self.heap[0] = self.heap[-1]
```

```
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
        elif len(self.heap) == 1:
            max = self.heap[0]
            del self.heap[0]
            return max
        else:
            return None

    def __percolateUp(self, index):
        parent = (index-1)//2
        if index <= 0:
            return
        elif self.heap[parent] < self.heap[index]:
            tmp = self.heap[parent]
            self.heap[parent] = self.heap[index]
            self.heap[index] = tmp
            self.__percolateUp(parent)

    def __maxHeapify(self, index):
        left = (index * 2) + 1
        right = (index * 2) + 2
        largest = index
        if len(self.heap) > left and self.heap[largest] < self.heap[left]:
            largest = left
        if len(self.heap) > right and self.heap[largest] < self.heap[right]:
            largest = right
        if largest != index:
            tmp = self.heap[largest]
            self.heap[largest] = self.heap[index]
            self.heap[index] = tmp
            self.__maxHeapify(largest)

    def buildHeap(self, arr):
        self.heap = arr
        for i in range(len(arr)-1, -1, -1):
            self.__maxHeapify(i)


heap = MaxHeap()
heap.insert(12)
heap.insert(10)
heap.insert(-10)
heap.insert(100)

print(heap.getMax())

```

Now that we have studied the implementation of Max-Heaps in depth, implementing a Min-Heap will not be a problem and that's what we are going to study in the next lesson.

Back
Max Heap: Introduction

Next
Min Heap: Introduction

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/max-heap-implementation__introduction-to-heap__data-structures-for-coding-interviews-in-python)