heap_introduction

May 4, 2020

1 Priority Queues - Intuition

Consider the following scenario -

A doctor is working in an emergency wing at a hospital. When patients come in, a nurse checks their symptoms and based on the severity of the illness, sends them to the doctor. For e.g. a guy who has had an accident is sent before someone who has come with a runny nose. But there is a slight problem. There is only one nurse and only one doctor. In the amount of time nurse takes to check the symptoms, the doctor has to work alone with the patients, hurting their overall productivity.

You are a ninja programmer. The doctor comes to you for help. Your job is to write a small software in which patients will enter their symptoms and will receive a priority number based on their illness. The doctor has given you a list of common ailments, and the priority in which he would prefer seeing them. How would you solve the priority problem?

1.1 Priority Queues

Like the name suggests, a **priority queue** is similar to a regular queue, except that each element in the queue has a priority associated with it. A regular queue is a FIFO data structure, meaning that the first element to be added to the queue is also the first to be removed.

With a priority queue, this order of removal is instead based on the priority. Depending on how we choose to set up the priority queue, we either remove the element with the most priority, or an element of the least priority.

For the sake of discussion, let's focus on removing the element of least priority for now.

1.2 Functionality

If we were to create a `PriorityQueue` class, what methods would it need to have?

Here are the two key methods: * `insert` - insert an element * `remove` - remove an element

And we can also add the same utility methods that we had in our regular `Queue` class: * `front` - returns the element at the front of the queue * `size` - returns the number of elements present in the queue * `is_empty` - returns `True` if there are no elements in the queue, and `False` otherwise

As part of this functionality, we will need a way of assigning priorities to the items.

A very common way to solve the patient-doctor problem mentioned above would be to assign each ailment a priority. For e.g.

```
* A running nose may be assigned priority 1
* Fever may be assigned 2
* Accident may get a priority 10
```

You will find this theme recurring in all of programming. We use numbers to effectively represent data.

For the sake of simplicity, let's only consider integers here. Let us assume a scenario where we get integers as input and we assign a priority on how large / small they are. Let us say the smaller the number, the smaller its priority. So, in our simplified version of the problem statement the value of the integer serves as a priority.

Our goal is to create a queue where the element with the lowest priority is removed first. Therefore, the `remove` method will remove the smallest number from the priority queue. Thus, the largest number will be the last to be removed from the priority queue and the smallest number will be the first to be removed.

1.3 How should we implement it?

What we've described above is just the abstract characteristics that we expect from this data structure. As with stacks and queues (and other abstract data types), there is more than one way that we could implement our priority queue such that it would exhibit the above behaviors.

However, not all implementations are ideal. When we implemented a regular queue earlier, you may remember the `enqueue` and `dequeue` methods had a time complexity of $O(1)$. Similarly, we would like the `insert` and `remove` methods on our priority queue to be fast.

So, what underlying structure should we use to implement the priority queue such that it will be as efficient as possible? Let's look at some different structures and consider the pros and cons.

1.3.1 Arrays

Earlier, we saw that one way to implement a queue was by using an array. We could do a similar thing for priority queues. We could use the array to store our data.

Insertion in an array is very fast. Unless the array is full, we can do it in `O(1)` time.

Note: When the array is full, we will simply create a new array and copy all the elements from our old array to new array. It's exactly similar to what we do for our queue's implementation using arrays.

What about removal? We always want to remove the smallest or highest priority data from the array, depending on if this is a max-heap or min-heap. In the worst case, we will have to search the entire array, which will take `O(n)` time. Thus, to remove the element, the time complexity would be `O(n)`.

This also creates an additional problem for us. The index from which we removed the element is now empty. We cannot leave empty indices in our array. Over the course of operations, we will be wasting a lot of space if we did that.

Therefore, insertion no longer happens in `O(1)` time. Rather, every time we insert, we will have to look for these empty indices and put our new element in the first empty index we find. In the worst case, this also takes `O(n)` time. Therefore, our time complexity with arrays (for both insertion and removal) would be `O(n)`.

1.3.2 LinkedList

Insertion is very easy in a linked list. If we maintain a variable to keep track of the `tail` of the linked list, then we can simply add a new node at this location. Thus, insertion takes `O(1)` time.

For removal, we will have to traverse the entire list and find the smallest element, which will require `O(n)` time.

Note that with linked lists, unlike arrays, we do not have to worry about empty indices.

A linked linked certainly seems to be a better option than an array. Although they have the same time complexity for removal, the time complexity for insertion is better.

1.3.3 HashMap

The same problem lies in HashMap as well. We can insert in `O(1)` time. Although, we can remove an element from a HashMap in `O(1)` time but we have to first search for the smallest element in the map. This will again take `O(n)` time. Therefore, the time complexity of `remove` is `O(n)` for hashmaps.

1.3.4 Binary Search Trees

Binary Search Trees are laid out according to the value of the node that we want to insert. All elements greater than the `root` go to the right of the `root`, and all elements smaller than the `root` go to the left of the `root`.

If we assume that our Binary Search tree is balanced, insertion would require `O(h)` time in the worst case. Similarly, removal would also require `O(h)` time. Here `h` is the height of the binary search tree.

A Binary Tree is called a `Balanced Binary Tree` when the difference between the heights of it's left subtree and right subtree do not differ by more than one. Additionally, to be balanced, all the subtrees of the binary tree must also be balanced.

For a balanced tree, we can safely approximate the height of the tree `h` to `log(n)`. Thus, both insertion and removal require `O(log(n))` time in a binary search tree.

However, in the worst case, our binary search tree might just be a sequential list of nodes (stretching to the right or to the left). Consider the following tree:

In such a scenario the binary search tree effectively turns into a linked list. In this case, the time complexity would be `O(n)`

To avoid this situation, we would need a self-balancing tree which incure additional complexity.

We could use any of the above data structures to implement our priority queue—and they would work, in the sense that they would exhibit the outward behavior we expect in a priority queue.

However, none of them acheived our goal of having $O(1)$ time complexity for both `insert` and `remove`. To do that, we will need to explore something new: A *heap*.

2 Heaps

A heap is a data structure with the following two main properties:

1. Complete Binary Tree
2. Heap Order Property

1. **Complete Binary Tree** - Like the name suggests we use a binary tree to create heaps. A complete binary tree is a special type of binary tree in which all levels must be filled except for the last level. Moreover, in the last level, the elements must be filled from left to right.

A. is a complete binary tree. Notice how every level except the last level is filled. Also notice how the last level is filled from left to right.

B. is not a complete binary tree. Although evey level is filled except for the last level. Notice how the last level is not filled from left to right. `25` does not have any right node and yet there is one more node (`9`) in the same level towards the right of it. It is mandatory for a complete binary tree to be filled from left to right.

C. is also not a binary tree. Notice how the second level is not completely filled and yet we have elements in the third level. The right node of `10` is empty and yet we have nodes in the next level.

- **Heap Order Property** - Heaps come in two flavors
  - Min Heap
  - Max Heap
- Min Heap - In the case of min heaps, for each node, the parent node must be smaller than both the child nodes. It's okay even if one or both of the child nodes do not exists. However if they do exist, the value of the parent node must be smaller. Also note that it does not matter if the left node is greater than the right node or vice versa. The only important condition is that the root node must be smaller than both it's child nodes
- Max Heap - For max heaps, this condition is exactly reversed. For each node, the value of the parent node must be larger than both the child nodes.

Thus, for a data structure to be called a Heap, it must satisfy both of the above properties. 1. It must be a complete binary tree 2. It must satisfy the heap order property. If it's a min heap, it must satisfy the heap order property for min heaps. If it's a max heap, it should satisfy the heap order property for max heaps.

2.1 Complete Binary Tree

Let's go back to our complete binary tree `A`.

If we have to insert one more node, where should the next node go? Because `A.` is a complete binary tree, the next node can only go as the `left` node of `15`.

Similarly, let's look back `A.` again. If we have to delete a node from `A.`, which node should we delete? Again, to ensure that our tree remains a complete binary tree even after deleting a node, we can only remove `9`.

Thus, we know which node to remove and where to insert a new node. Notice that both of these operations do not depend upon values of other nodes. Rather, both `insert` and `remove` operations on a complete binary tree depend upon the position of the last inserted node.

This cell may require some visualization due to the mathematics involved

Now that we know about a complete binary, let's think about it in terms of Priority Queues. We talked about binary search trees where the complexity for `insert` and `remove` operation would be `O(log(n))` if the BST is balanced.

In case of a complete binary tree we do not have to worry about whether the tree is balanced or not.

- `Max number of nodes in 1st level = 1`
- `Max number of nodes in 2nd level = 2`
- `Max number of nodes in 3rd level = 4`
- `Max number of nodes in 4th level = 8`

We see that there is a clear patter here.

- `Max number of nodes in hth level` = $2^{(h-1)}$

Also, we can calculate the `max number of nodes from 1st level to hth level` = $2^h - 1$
Similarly, we can calculate the `min number of nodes from 1st level to hth level` =$2^{(h-1)}$
Note: the minimum number of nodes from 1st level to hth level = max number of nodes from 1st level to (h-1)th level + 1

Thus, in a complete binary tree of height `h`, we can be assured that the number of elements `n` would be between these two numbers i.e.

$$2^{(h-1)} <= n <= 2^h - 1$$

- If we write the first inequality in base-2 logarithmic format we would have

$$\log_2 (2^{(h-1)}) <= \log_2 n$$

or

$$h <= \log_2 n + 1$$

- Similarly, if we write the second equality in base-2 logarithmic format

$$\log_2(n+1) <= \log_2\ 2^h$$

or

$$\log_2(n+1) <= h$$

Thus the value of our height `h` is always

$$\log_2(n+1) <= h <= \log_2 n + 1$$

We can see that the height of our complete binary tree will always be in the order of `O(h)` or `O(log(n))`

So, if instead of using a binary search tree, we use a complete binary tree, both `insert` and `remove` operation will have the time complexity $\log_2 n$

2.1.1 Heaps for Priority Queues

Let's take a step back and reflect on what we have done.

1. We have examined popular data structures and observed their time complexities.
2. We have looked at a new data structure called Heap
3. We know that Heaps have two properties - i. CBT ii. Heap Order Property
4. We have looked at what CBT is and what Heap Order Property is

By now, it must have been clear to you that we are going to use Heaps to create our Priority Queues. But are you convinced that heaps are a good structure to create Priority Queues?

Ans.

1. Other than Binary Search trees, all other popular data structures seemed to have a time complexity of `O(n)` for both `insertion and removal`.
2. Binary Search Trees seemed like an effective data structure with average case time complexity of `O(log(n)` (or `O(h)`) for both the operations. However, in the worst case, a Binary Search Tree may not be balanced and instead behave like a linked list. In such a case, the time complexity in terms of height would still be `O(h)` but because the height of the binary search tree will be equal to the number of elements in the tree, the actual time complexity in terms of `number of elements n` would be `O(n)`.
3. The CBT property of Heaps ensures that the tree is always balanced. Therefore, the `height h` of the tree will always be equal to `log(n)`.
4. The Heap Order Property ensures that there is some definite structure to our Complete Binary Tree with respect to the value of the elements. In case of a min-heap, the minimum element will always lie at the root node. Similarly, in case of a maxp-heap, the maximum element will always lie at the root node. In both the cases, every time we `insert or remove` an element, the time complexity remains `O(log(n))`.

Therefore, because of the time complexity being `O(log(n))`, we prefer heaps over other popular data structures to create our Priority Queues.

2.2 Complete Binary Trees using Arrays

Although we call them complete binary trees, and we will always visualize them as binary trees, we never use binary trees to create them. Instead, we actually use arrays to create our complete binary trees.

Let's us see how.

An array is a contiguous blocks of memory with individual "blocks" are laid out one after the other in memory. We are used to visualizing arrays as sequential blocks of memory.

However, if we visualize them in the following way, can we find some similarities between arrays and complete binary trees?

Let's think about it.

- **In a complete binary tree, it is mandatory for all levels before the last level to be completely filled.**

If we visualize our array in this manner, do we satisfy this property of a CBT? All we have to ensure is that we put elements in array indices sequenially i.e. the smaller index first and the larger index next. If we do that, we can be assured that all levels before the last level will be completely filled.

- **In a CBT, if the last level is not completely filled, the nodes must be filled from left to right.**

Again, if we put elements in the array indices sequentially, from smaller index to larger index, we can be assured that if the last level is not filled, it will certainly be filled from left to right.

Thus we can use an array to create our Completer Binary Tree. Although it's an array, we will always visualize it as complete binary tree when talking about heaps.

Now let's talk about `insert` and `remove` operation in a heap. We will create our heap class which with these two operations. We also add a few utility methods for our convenience. Finally, because we know we are going to use arrays to create our heaps, we will also initialize an array.

Note that we are creating min heaps for now. The max heap will follow the exact some process. The only difference arises in the Heap Order Property.

As always we will use Python lists like C-style arrays to make the implementation as language agnostic as possible.

```
In [ ]: class Heap:
            def __init__(self, initial_size):
                self.cbt = [None for _ in range(initial_size)]        # initialize arrays
                self.next_index = 0                                   # denotes next index where

            def insert(self, data):
                pass

            def remove(self):
                pass
```

2.2.1 Insert

Insertion operation in a CBT is quite simple. Because we are using arrays to implement a CBT, we will always insert at the `next_index`. Also, after inserting, we will increment the value of `next_index`.

However, this isn't enough. We also have to maintain the heap order property. We know that for min-heaps, the parent node is supposed to be smaller than both the child nodes.

Counting indices, we know that our next element should go at index 8. Let's say we want to insert `15` as our next element in the heap. In that case, we start off by inserting `15` at index 8.

Remember, although we are using arrays to implement a CBT, we will always visualize it as a binary tree. We will only consider them as arrays while implementing them.

So, we went ahead and insert `15` at index 8. But this violates our heap order property. We are considering min-heap and the parent node of `15` is larger.

In such a case, we `heapify`. We consider the parent node of the node we inserted and compare their values. In case of min-heaps, if the parent node is larger than the child node (the one we just inserted), we swap the nodes.

Now the complete binary tree looks something like

Is the problem solved?

Swapping the nodes for `15` and `50` certainly solved our problem. But it also introduced a new problem. Notice `15` and `20`. We are again in the same spot. The parent node is larger than the child node. And in a min-heap we cannot allow that. So, what do we do? We heapify. We swap these two nodes just as we swapped our previous two nodes.

After swapping, our CBT looks like

Does everything seem fine now?

We only have to consider the nodes that we swapped. And looks like we are fine.

Now let's take a step back and see what we did.

- We first inserted our element at the possible index.
- Then we compared this element with the parent element and swapped them after finding that our child node was smaller than our parent node. And we did this process again. While writing code, we will continue this process until we find a parent which is smaller than the child node. Because we are travering the tree upwards while heapifying, this particular process is more accurately called `up-heapify`.

Thus our `insert` method is actually done in two steps: * insert * up-heapify

2.2.2 Time Complexity

Before talking about the implementation of `insert`, let's talk about the time complexity of the `insert` method.

- Putting an element at a particular index in an array takes `O(1)` time.
- However, in case of heapify, in the worst case we may have to travel from the node that we inserted right to the root node (placed at 0th index in the array). This would take `O(h)` time. In other words, this would be an `O(log(n))` operation.

Thus the time complexity of `insert` would be `O(log(n))`.

2.2.3 Insert - implementation

Although we are using arrays for our CBT, we are visualizing it as a binary tree for understanding the idea. But when it comes to the implementation, we will have to think about it as an array. It is an array, after all.

In the above image, we can safely assume that

- index 0 is the root node of the binary tree
- index 0 is the parent node for indices 1 and 2 i.e. 1 is the left node of index 0, and 2 is the right node
- Similarly, 3 and 4 are the child nodes of index 1.
- And 5 and 6 are the child nodes of index 2

Can we deduce any pattern from this?

- If you notice carefully, the child nodes of 0 are ---> 1 and 2
- The child nodes of 1 are ---> 3 and 4
- The child nodes of 2 are ---> 5 and 6

The child nodes of p are ---> `2 * (p + 1)` and `2 * (p + 2)`
i.e. `the child nodes of a parent index p are placed at indices 2 * (p + 1) and 2 * (p + 2)`
Similarly, can you deduce parent indices from a child index c?
Ans. `for a child node at index c, the parent node will be located at (p - 1)//2`
Note the integer division
Using these ideas, implement the `insert` method.

```
In [ ]: class Heap:
            def __init__(self, initial_size):
                self.cbt = [None for _ in range(initial_size)]        # initialize arrays
                self.next_index = 0                                   # denotes next index where

            def insert(self, data):
                """
                Insert `data` into the heap
                """
                pass
```

Hide Solution

```
In [ ]: class Heap:
            def __init__(self, initial_size):
                self.cbt = [None for _ in range(initial_size)]  # initialize arrays
                self.next_index = 0  # denotes next index where new element should go

            def _up_heapify(self):
                child_index = self.next_index

                while child_index >= 1:
                    parent_index = (child_index - 1) // 2
                    parent_element = self.cbt[parent_index]
                    child_element = self.cbt[child_index]

                    if parent_element > child_element:
                        self.cbt[parent_index] = child_element
                        self.cbt[child_index] = parent_element

                        child_index = parent_index
                    else:
                        break

            def insert(self, data):
                # insert element at the next index
                self.cbt[self.next_index] = data

                # heapify
                self._up_heapify()
```

```
        # increase index by 1
        self.next_index += 1

        # double the array and copy elements if next_index goes out of array bounds
        if self.next_index >= len(self.cbt):
            temp = self.cbt
            self.cbt = [None for _ in range(2 * len(self.cbt))]

            for index in range(self.next_index):
                self.cbt[index] = temp[index]
```

2.2.4 Remove

For min-heaps, we remove the smallest element from our heaps. For max-heaps, we remove the largest element from the heap.

By now, you must have realized that in case of min-heaps, the minimum element is stored at the root node of the complete binary tree. *Again, we are emphasizing the fact that we will always visualize a complete binary tree as a binary tree and not an array.*

Consider this CBT. Our remove operation should remove `10` from the tree. But if we remove `10`, we need to put the next smaller element at the root node. But that will again leave one node empty. So, we will again have to go to our next smaller element and place it at the node that is empty. This sounds tedious.

Rather, we use a very simple yet efficient trick to remove the element. We swap the first node of the tree (which is the minimum element for a min-heap) with the last node of the tree.

If we think about the implementation of our complete binary tree, we know that `10` will now be present at the last index of the array. So, removing `10` is a `O(1)` operation.

However, you might have noticed that our complete binary tree does not follow the heap order property which means that it's no longer a heap. So, just like last time, we `heapify`. This time however, we start at the top and heapify in downward direction. Therefore, this is also called as `down-heapify`.

We look at `50` which is present at the root node, and compare it with both it's children. We take the minimum of the three nodes i.e. `50, 15, and 40`, and place this minimum at the root node. At the same time, we place `50` at the node which we placed at the root node.

Following this operation, our CBT looks like

Even now the CBT does not follow the heap order property. So, we again compare `50` with it's child nodes and swap `50` with the minimum of the three nodes.

At this point we stop because our CBT follows the heap order property.

Can you code the `remove` method?

```
In [ ]: class Heap:
            def __init__(self, initial_size=10):
                self.cbt = [None for _ in range(initial_size)]  # initialize arrays
                self.next_index = 0  # denotes next index where new element should go

            def size(self):
                return self.next_index
```

```
        def remove(self):
            """
            Remove and return the element at the top of the heap
            """
            pass
```

Hide Solution

```
In [ ]: class Heap:
        def __init__(self, initial_size=10):
            self.cbt = [None for _ in range(initial_size)]  # initialize arrays
            self.next_index = 0  # denotes next index where new element should go

        def _down_heapify(self):
            parent_index = 0

            while parent_index < self.next_index:
                left_child_index = 2 * parent_index + 1
                right_child_index = 2 * parent_index + 2

                parent = self.cbt[parent_index]
                left_child = None
                right_child = None

                min_element = parent

                # check if left child exists
                if left_child_index < self.next_index:
                    left_child = self.cbt[left_child_index]

                # check if right child exists
                if right_child_index < self.next_index:
                    right_child = self.cbt[right_child_index]

                # compare with left child
                if left_child is not None:
                    min_element = min(parent, left_child)

                # compare with right child
                if right_child is not None:
                    min_element = min(right_child, min_element)

                # check if parent is rightly placed
                if min_element == parent:
                    return

                if min_element == left_child:
                    self.cbt[left_child_index] = parent
```

```
                self.cbt[parent_index] = min_element
                parent = left_child_index

            elif min_element == right_child:
                self.cbt[right_child_index] = parent
                self.cbt[parent_index] = min_element
                parent = right_child_index

    def size(self):
        return self.next_index

    def remove(self):
        """
        Remove and return the element at the top of the heap
        """
        if self.size() == 0:
            return None
        self.next_index -= 1

        to_remove = self.cbt[0]
        last_element = self.cbt[self.next_index]

        # place last element of the cbt at the root
        self.cbt[0] = last_element

        # we do not remove the elementm, rather we allow next `insert` operation to over
        self.cbt[self.next_index] = to_remove
        self._down_heapify()

        return to_remove
```

2.2.5 Time Complexity

Can you determine the time complexity for `remove` using the same process that we followed for `insert`?

Ans: the time complexity for `remove` is also O(log(n))

2.2.6 Final Heap

Using the `insert` and `remove` functions, let's run the heap.

```
In [ ]: class Heap:
            def __init__(self, initial_size=10):
                self.cbt = [None for _ in range(initial_size)]  # initialize arrays
                self.next_index = 0  # denotes next index where new element should go

            def insert(self, data):
                # insert element at the next index
```

```
        self.cbt[self.next_index] = data

        # heapify
        self._up_heapify()

        # increase index by 1
        self.next_index += 1

        # double the array and copy elements if next_index goes out of array bounds
        if self.next_index >= len(self.cbt):
            temp = self.cbt
            self.cbt = [None for _ in range(2 * len(self.cbt))]

            for index in range(self.next_index):
                self.cbt[index] = temp[index]

    def remove(self):
        if self.size() == 0:
            return None
        self.next_index -= 1

        to_remove = self.cbt[0]
        last_element = self.cbt[self.next_index]

        # place last element of the cbt at the root
        self.cbt[0] = last_element

        # we do not remove the elementm, rather we allow next `insert` operation to over
        self.cbt[self.next_index] = to_remove
        self._down_heapify()
        return to_remove

    def size(self):
        return self.next_index

    def is_empty(self):
        return self.size() == 0

    def _up_heapify(self):
        # print("inside heapify")
        child_index = self.next_index

        while child_index >= 1:
            parent_index = (child_index - 1) // 2
            parent_element = self.cbt[parent_index]
            child_element = self.cbt[child_index]

            if parent_element > child_element:
```

```
            self.cbt[parent_index] = child_element
            self.cbt[child_index] = parent_element

            child_index = parent_index
        else:
            break

def _down_heapify(self):
    parent_index = 0

    while parent_index < self.next_index:
        left_child_index = 2 * parent_index + 1
        right_child_index = 2 * parent_index + 2

        parent = self.cbt[parent_index]
        left_child = None
        right_child = None

        min_element = parent

        # check if left child exists
        if left_child_index < self.next_index:
            left_child = self.cbt[left_child_index]

        # check if right child exists
        if right_child_index < self.next_index:
            right_child = self.cbt[right_child_index]

        # compare with left child
        if left_child is not None:
            min_element = min(parent, left_child)

        # compare with right child
        if right_child is not None:
            min_element = min(right_child, min_element)

        # check if parent is rightly placed
        if min_element == parent:
            return

        if min_element == left_child:
            self.cbt[left_child_index] = parent
            self.cbt[parent_index] = min_element
            parent = left_child_index

        elif min_element == right_child:
            self.cbt[right_child_index] = parent
            self.cbt[parent_index] = min_element
```

```
                    parent = right_child_index

        def get_minimum(self):
            # Returns the minimum element present in the heap
            if self.size() == 0:
                return None
            return self.cbt[0]
```

```
In [ ]: heap_size = 5
        heap = Heap(heap_size)

        elements = [1, 2, 3, 4, 1, 2]
        for element in elements:
            heap.insert(element)
        print('Inserted elements: {}'.format(elements))

        print('size of heap: {}'.format(heap.size()))

        for _ in range(4):
            print('Call remove: {}'.format(heap.remove()))

        print('Call get_minimum: {}'.format(heap.get_minimum()))

        for _ in range(2):
            print('Call remove: {}'.format(heap.remove()))

        print('size of heap: {}'.format(heap.size()))
        print('Call remove: {}'.format(heap.remove()))
        print('Call is_empty: {}'.format(heap.is_empty()))
```

That's it for heaps! Now it's time for the next topic, self-balancing trees.