9️⃣

009 - Heaps

Heaps are used to maintain a maximum or minimum value in a dataset.

Think of the max-heap as a binary tree with two qualities:

- The root is the *maximum value* of the dataset.
- Every parent's value is *greater than its children*.

These two properties are the defining characteristics of the max-heap. By maintaining these two properties, we can efficiently retrieve and update the maximum value.

- left child: `(index * 2) + 1`
- right child: `(index * 2) + 2`
- parent: `(index - 1) / 2` ***— not used on the root!***

Max heap implementation

```
class MaxHeap:
  def __init__(self):
    self.heap_list = [None]
    self.count = 0

  # HEAP HELPER METHODS
  # DO NOT CHANGE!
  def parent_idx(self, idx):
    return idx // 2

  def left_child_idx(self, idx):
    return idx * 2

  def right_child_idx(self, idx):
    return idx * 2 + 1

  # END OF HEAP HELPER METHODS
```

```
  def add(self, element):
    self.count += 1
    print("Adding: {0} to {1}".format(element, self.heap_list))
    self.heap_list.append(element)
    self.heapify_up()

  def heapify_up(self):
    print("Heapifying up")
    idx = self.count
    while self.parent_idx(idx) > 0:
      child = self.heap_list[idx]
      parent = self.heap_list[self.parent_idx(idx)]
      if parent < child:
        print("swapping {0} with {1}".format(parent, child))
        self.heap_list[idx] = parent
        self.heap_list[self.parent_idx(idx)] = child
      idx = self.parent_idx(idx)
    print("Heap Restored {0}".format(self.heap_list))
```

Heap Sort

Here's how we'll accomplish the implementation of heapsort:

1. Build a max-heap to store the data from an unsorted list.
2. Extract the largest value from the heap and place it into a sorted list.
3. Replace the root of the heap with the last element in the list. Then, rebalance the heap.
4. Once the max-heap is empty, return the sorted list.

Build a max-heap

For this algorithm, we'll want to build out a max-heap. As a reminder, in a max-heap, the root value is the largest value. Each parent node must have a larger value than its associated children.

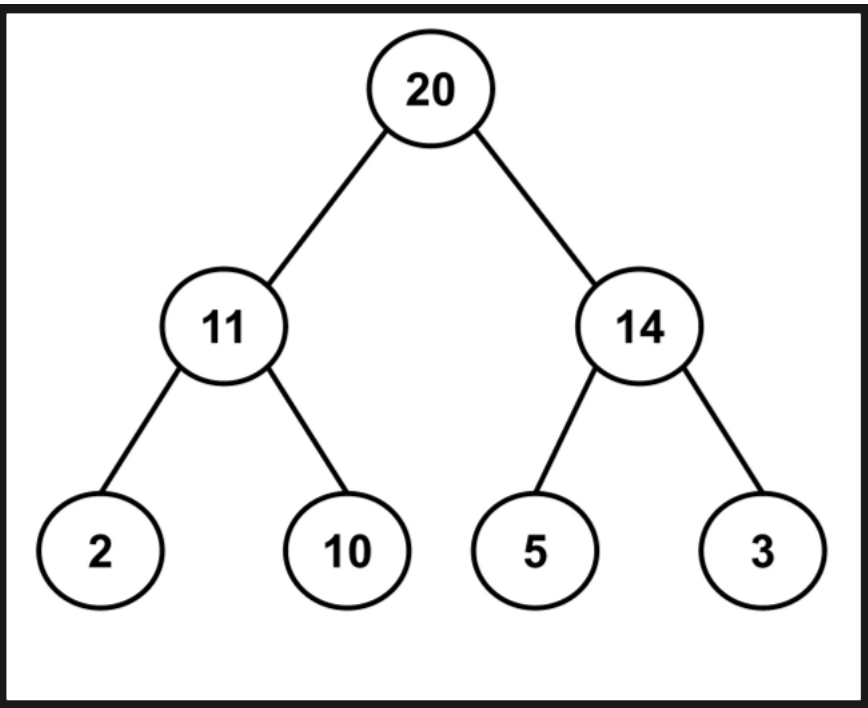
Imagine we had the following list of unsorted values:

```
[14, 11, 2, 20, 3, 10, 3]
```

By placing our values into a max-heap data structure, our list would look like this:

```
[20, 11, 14, 2, 10, 5, 3]
```

We can visualize the above max-heap like so:



Extract the root of the heap

In order to sort our data, we'll repeatedly extract and remove the largest value from the heap until it's empty. By following the rule of heaps, we can expect to find the largest value located at the root of the heap.

After removing the largest value, we can't just leave our heap without a root because that would cause us to have two orphaned nodes. Instead, we can swap our root node with the last element in the heap. Since the last element has no children, we can easily remove the value from the heap.

This step does cause one major problem. By swapping the two elements, our root node isn't the largest value in the heap! We'll need to restructure the heap in order to ensure that it's balanced.

Restore the heap: heapify down

With the root value no longer holding the largest value, we've violated an important rule about heaps: the parent must contain a value that is larger than its children's values.

We can fix this though! In the previous lesson, we learned how to **heapify** up: adding a value to the end of a heap and working our way up the data structure to find its correct placement. Now we need to heapify *down*. To heapify down, we'll first compare our new root value to its children. Then, we'll select the child with the larger value and swap it with the root value. We'll continue working our way down the heap until it is balanced again:

In the example above, we swap the original root value `20` with the right-most child `3`. With `3` as the new root, we compare the value to its child value, `14`. Since `14` is greater than `3`, we will swap the two values and make `14` the new root. Next, we'll compare `3` to its new child value, `5`. Once again, the child value is greater than its parent, so we will swap `3` and `5`. With no more children to compare `3` to, our heap has been rebalanced.

Repeat

We'll repeat the process of swapping the root and the last element, extracting the largest value, and rebalancing the heap while the data structure has a size greater than `1`. Once we hit this condition, we will have an ordered list of values.

```
class MaxHeap:
  def __init__(self):
    self.heap_list = [None]
    self.count = 0

  # HEAP HELPER METHODS
  # DO NOT CHANGE!
  def parent_idx(self, idx):
    return idx // 2

  def left_child_idx(self, idx):
    return idx * 2

  def right_child_idx(self, idx):
    return idx * 2 + 1

  def child_present(self, idx):
    return self.left_child_idx(idx) <= self.count

  # END OF HEAP HELPER METHODS
```

```
def add(self, element):
  self.count += 1
  print("Adding: {0} to {1}".format(element, self.heap_list))
  self.heap_list.append(element)
  self.heapify_up()

def heapify_up(self):
  print("Heapifying up")
  idx = self.count
  while self.parent_idx(idx) > 0:
    child = self.heap_list[idx]
    parent = self.heap_list[self.parent_idx(idx)]
    if parent < child:
      print("swapping {0} with {1}".format(parent, child))
      self.heap_list[idx] = parent
      self.heap_list[self.parent_idx(idx)] = child
    idx = self.parent_idx(idx)
  print("Heap Restored {0}".format(self.heap_list))

def retrieve_max(self):
  if self.count == 0:
    print("No items in heap")
    return None
  max_value = self.heap_list[1]
  print("Removing: {0} from {1}".format(max_value, self.heap_list))
  self.heap_list[1] = self.heap_list[self.count]
  self.count -= 1
  self.heap_list.pop()
  print("Last element moved to first: {0}".format(self.heap_list))
  self.heapify_down()
  return max_value

def heapify_down(self):
  idx = 1
  while self.child_present(idx):
    print("Heapifying down!")
    larger_child_idx = self.get_larger_child_idx(idx)
    child = self.heap_list[larger_child_idx]
    parent = self.heap_list[idx]
    if parent < child:
      self.heap_list[idx] = child
      self.heap_list[larger_child_idx] = parent
    idx = larger_child_idx
  print("HEAP RESTORED! {0}".format(self.heap_list))
  print("")
```

```
    def get_larger_child_idx(self, idx):
      if self.right_child_idx(idx) > self.count:
        print("There is only a left child")
        return self.left_child_idx(idx)
      else:
        left_child = self.heap_list[self.left_child_idx(idx)]
        right_child = self.heap_list[self.right_child_idx(idx)]
        if left_child > right_child:
          print("Left child "+ str(left_child) + " is larger than right child " +
str(right_child))
          return self.left_child_idx(idx)
        else:
          print("Right child " + str(right_child) + " is larger than left child "
+ str(left_child))
          return self.right_child_idx(idx)

def heapsort(lst):
  sort = []
  max_heap = MaxHeap()
  for idx in lst:
    max_heap.add(idx)
  # add code below
  while max_heap.count > 0:
    max_value = max_heap.retrieve_max()
    sort.insert(0, max_value)
  return sort


my_list = [99, 22, 61, 10, 21, 13, 23]
sorted_list = heapsort(my_list)
# print the sorted list
print(sorted_list)
```

Heapify

Since our heap is actually implemented with an array, it would be good to have a way to actually create a heap in place starting with an array that isn't a heap and ending with an array that is heap. While it is possible to simply "insert" values into the heap repeatedly, the faster way to perform this task is an algorithm called Heapify.

In the Heapify Algorithm, works like this:

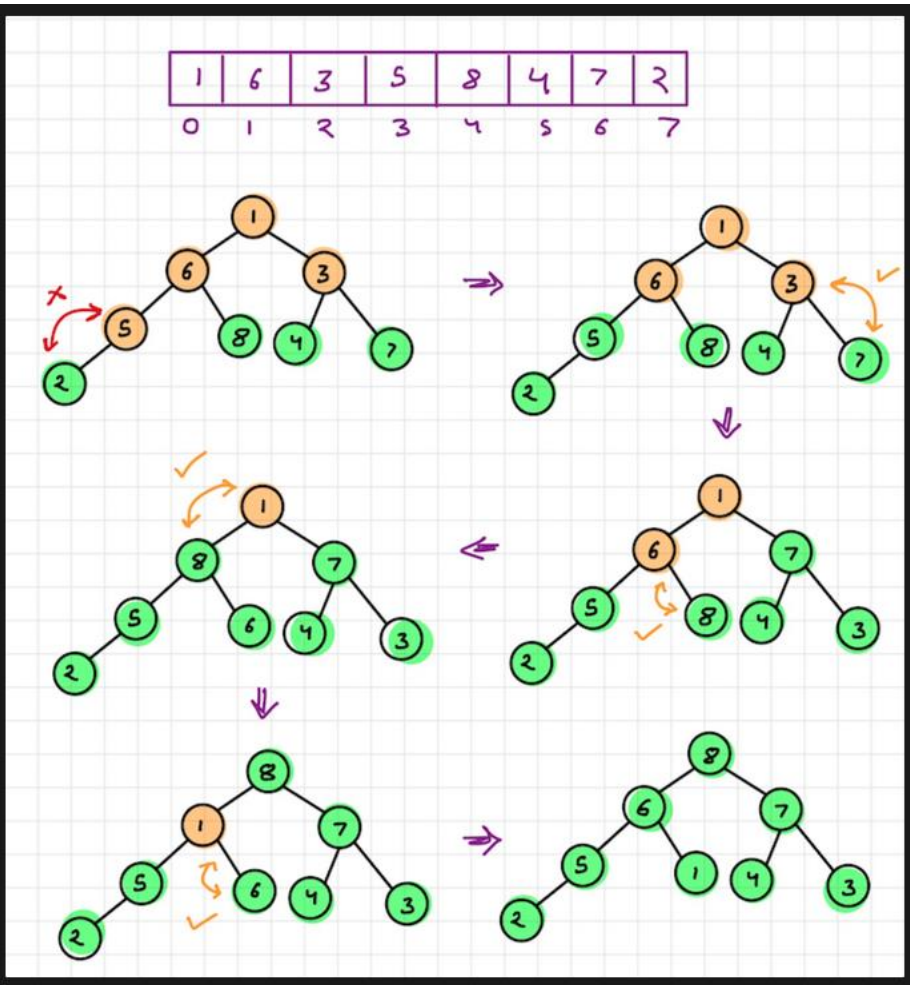
Given a node within the heap where both its left and right children are proper heaps (maintains proper heap order) , do the following:

- If the node has higher priority than both children, we are done, the entire heap is a proper heap
- Otherwise
  - swap current node with higher priority child
  - heapify() that subtree

Effectively what is happening is that we already know that both the left and right children are proper heaps. If the current node is higher priority than both children then the entire heap must be proper

However if its not then we do a swap, this means that the current node's value has now gone down into one of the subtrees. This could cause that subtree to no longer be a heap because the root of that subtree now has a value of lower priority than it use to have. so we heapify the subtree.

Our algorithm therefore starts at the first non-leaf node from the bottom. This node is at index (n-2)/2 where n is the total number of values in our heap.


1 6 3 5 8 4 7 2
0 1 2 3 4 5 6 7