
C
CodeIT

Lesson 2 Objectives:

To gain an understanding of:

1. What are **priority queues** and how we may efficiently implement them

2. What are and why we need the **heap (binary heap)** data structure

3. How to **analyse and implement** heaps

4. How to implement heaps with **changing keys**

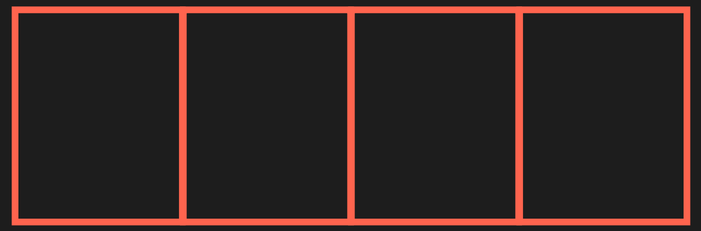
Recap: Queues

Queues

Queues are **FIRST IN FIRST OUT**

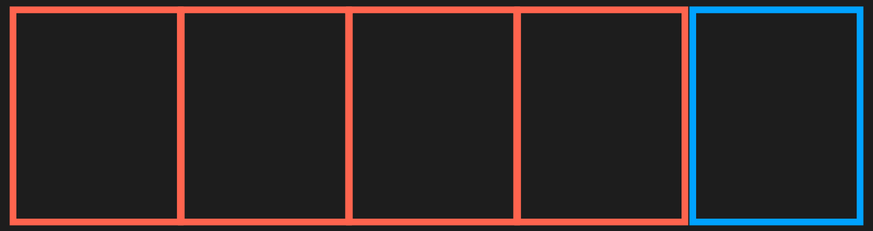
Queues

Queues are **FIRST IN FIRST OUT**



Queues

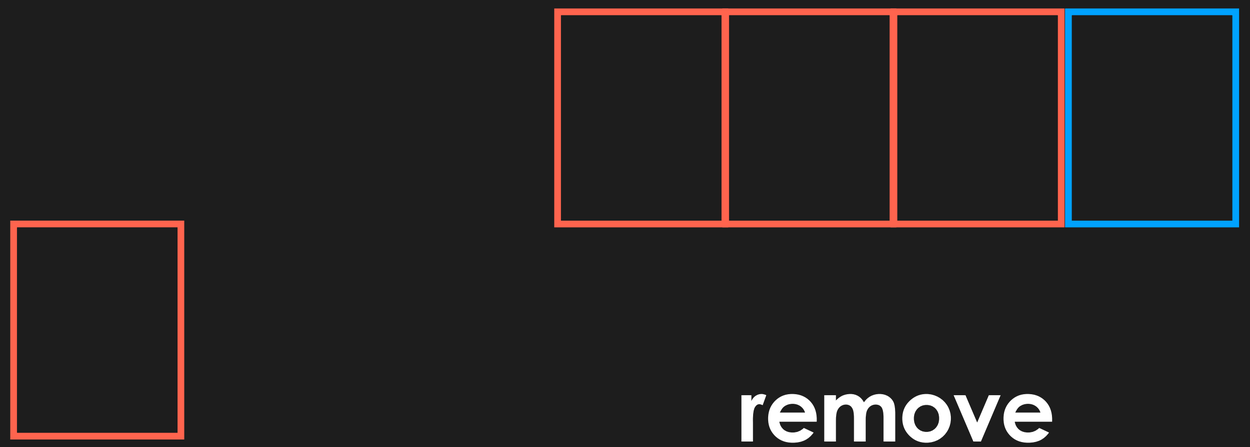
Queues are **FIRST IN FIRST OUT**



insert

Queues

Queues are **FIRST IN FIRST OUT**



Stock	Estimated Returns
Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

Imagine we had the following list of company stocks and their estimated returns on investment (higher is better)

Can we simulate the following behavior?

Front

Back

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

Can we simulate the following behavior?

Front

Google

Back

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

Can we simulate the following behavior?

Front

Google	Amazon

Back

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

Can we simulate the following behavior?

Front

Apple	Google	Amazon

Back

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

Can we simulate the following behavior?

Front

Apple	Microsoft	Google	Amazon

Back

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

Can we simulate the following behavior?

Front **Back**

Apple	Microsoft	Google	Netflix	Amazon

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The front of the queue should always be the stock with highest return

This is called a **Priority Queue**. We created a queue of stocks with the priority being its returns. How can we create such a structure?

Method 1: Linked Lists

Method 1: Linked Lists

Insertion: To insert into a PQ linked list, we can simply insert at the front. This takes **O(1)** time

Method 1: Linked Lists

Insertion: To insert into a PQ linked list, we can simply insert at the front. This takes **O(1)** time

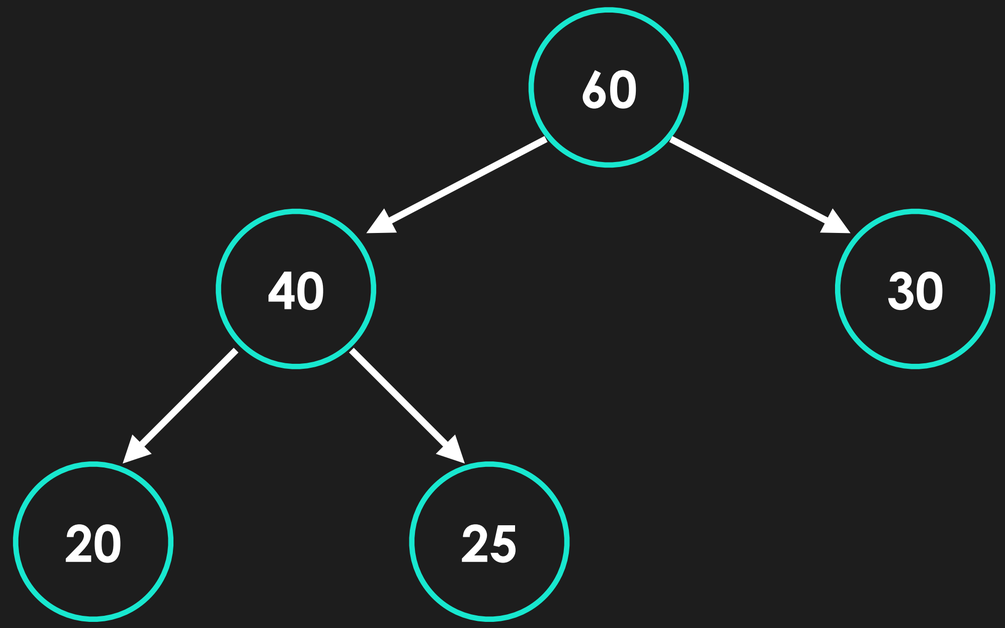
GetMax: To remove the item with highest priority at any time, traverse the list to find the max item using linear search, and remove. This takes **O(N)** time

Method 2: Binary Heaps

Method 2: Binary Heaps

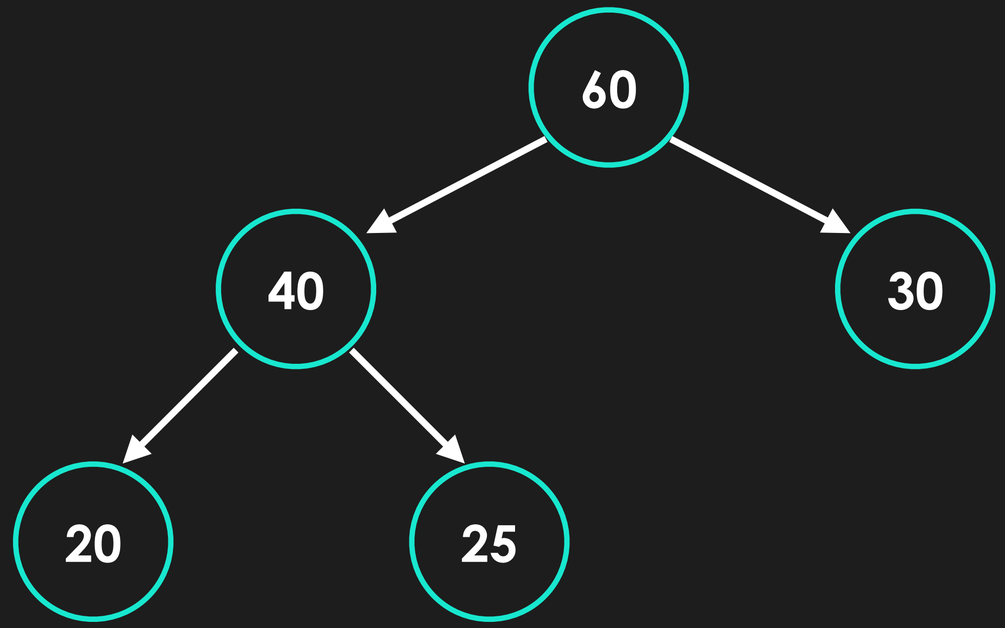
Binary Heaps make use of the binary tree structure to implement the priority queue logic

Binary Heap



Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

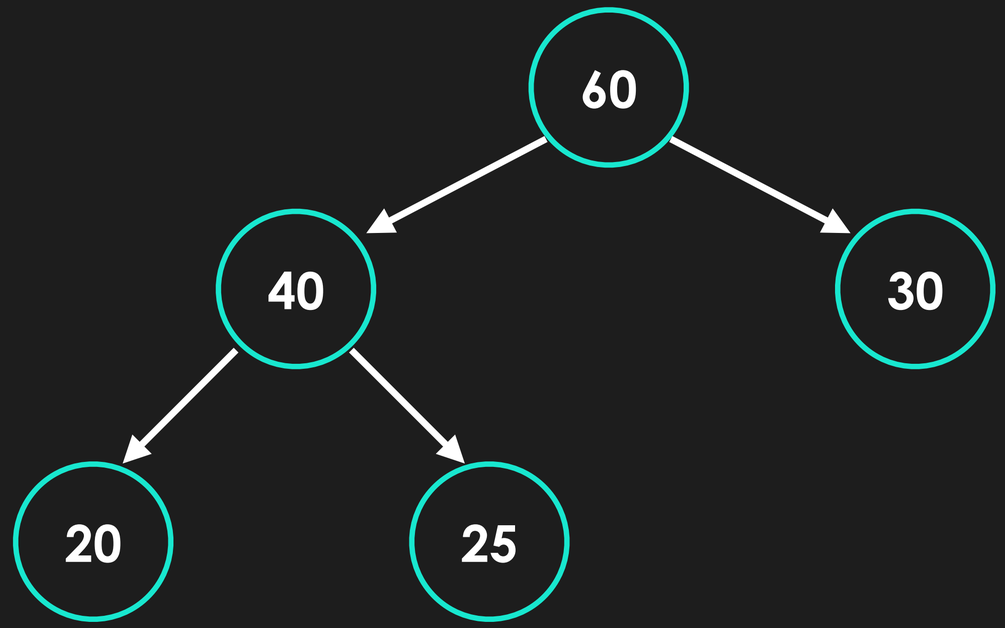
Binary Heap



Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

We will represent each node as it's **priority** (return)

Binary Heap

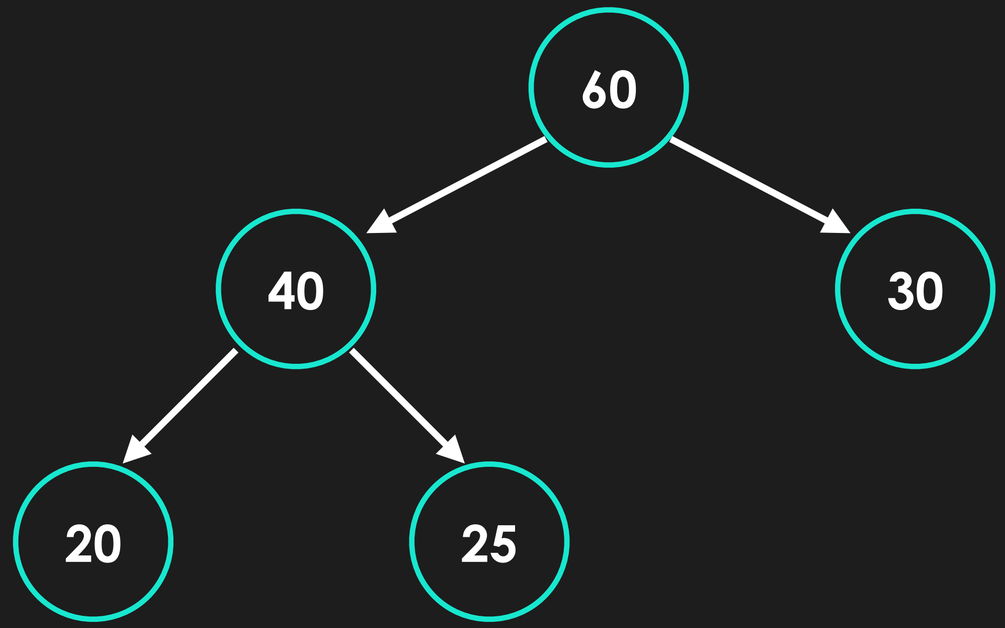


Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

Rules for Binary Heap:

1. Root node must be entry with max priority
2. For each node, every child node below must have lower priority

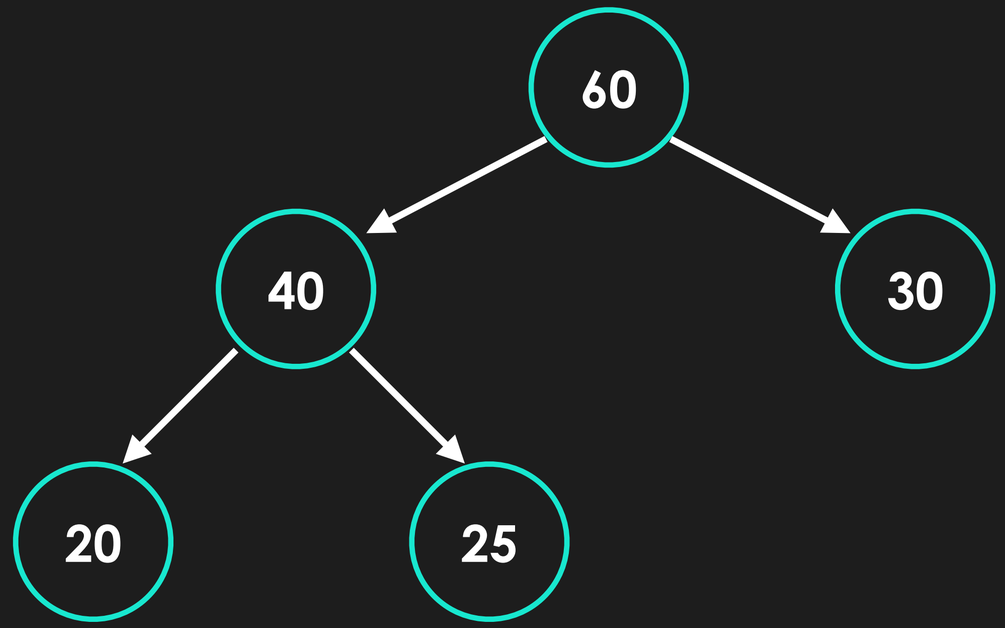
Binary Heap



Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

The above is also called a **max heap** because the priority is given to returns of **higher value**

Binary Heap



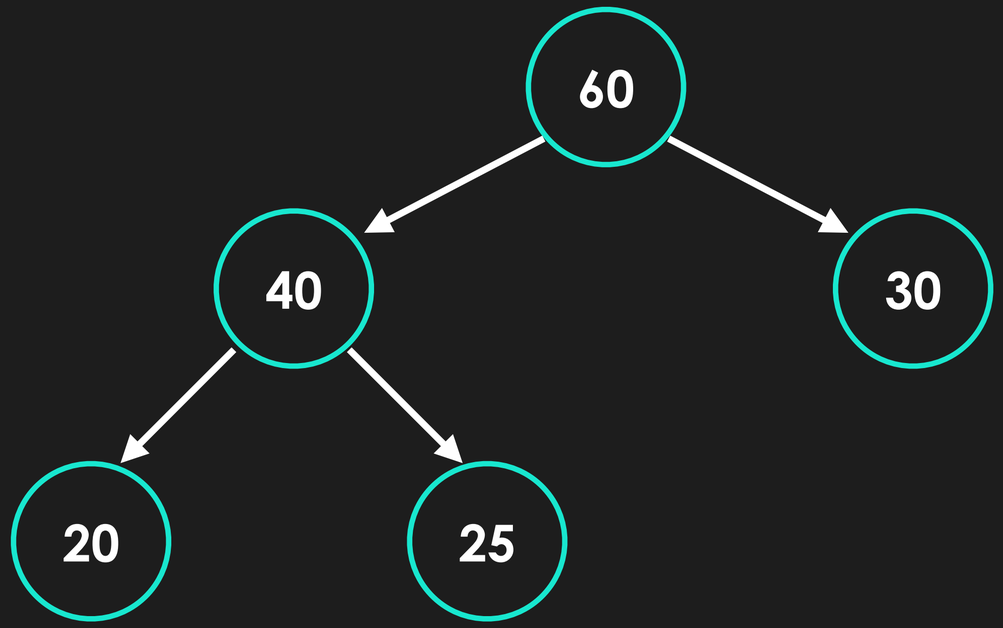
Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

Binary Heaps are a **more efficient** implementation of priority queues, and we will soon see why!

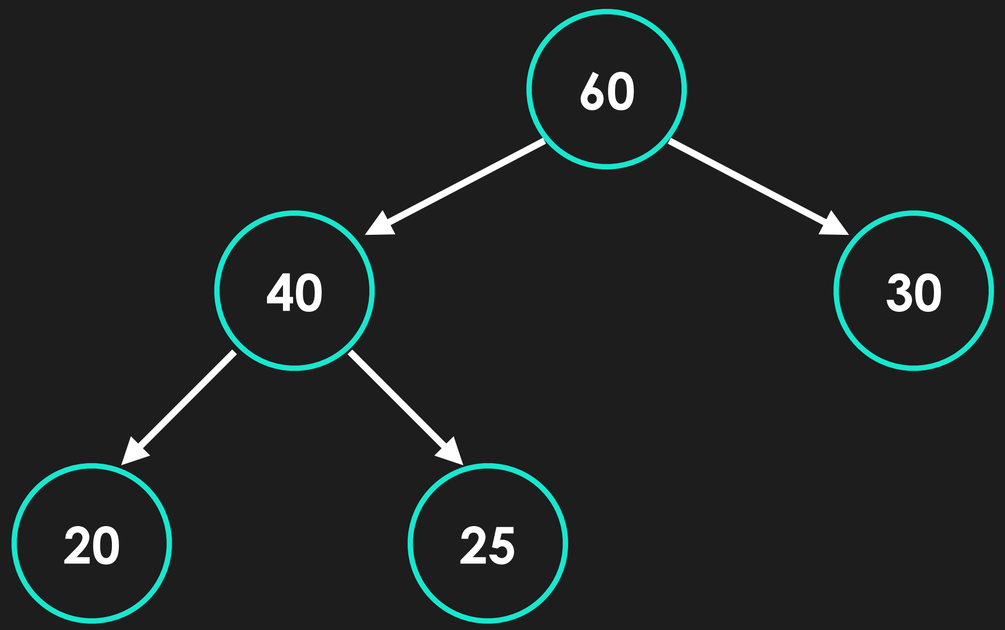
Binary Heap Representation

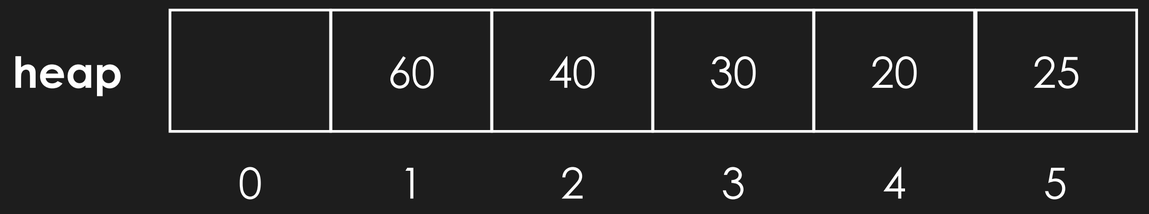
HeapItem Class

```
class HeapItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

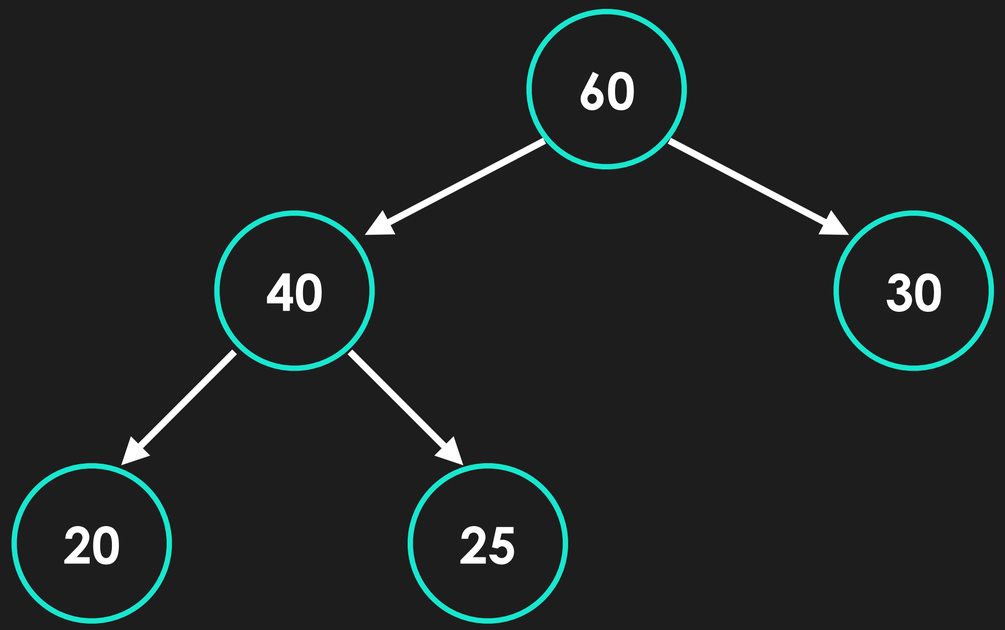


To represent binary heaps, we often use arrays (lists)





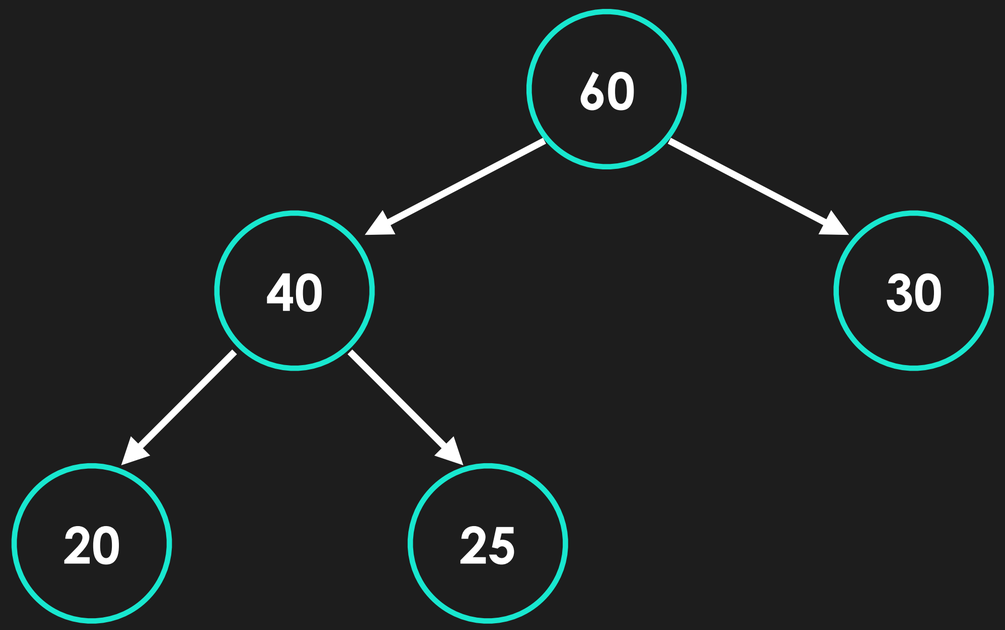
We only show the **value** of each HeapItem in the visualisation. However, each element in the heap is really a **HeapItem** class

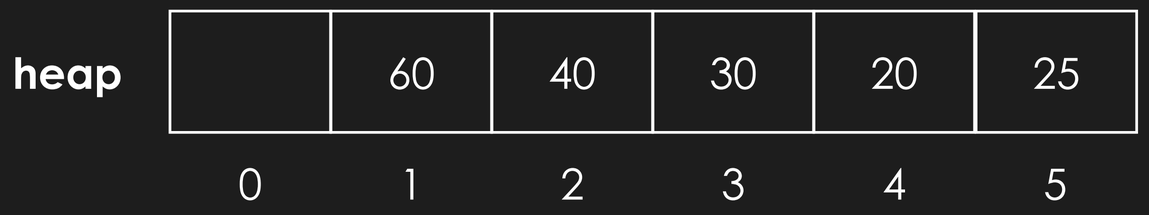


heap		60	40	30	20	25
	0	1	2	3	4	5

For a node at **index N:**

- It's **left child node** will be at index **2 * N**
- It's **right child node** will be at index **(2 * N + 1)**





For a node at **index N:**

- It's **left child node** will be at index **2 * N**

- It's **right child node** will be at index **(2 * N + 1)**

The **root node** will always hold **index 1** and we leave index 0 of our list empty to make things easier

Implementation of Max Heap

API

```
class MaxHeap:
    def __init__ (self, capacity):
        self.capacity = capacity
        self.heap = [None] * (capacity + 1)

    def sink(self, index):
        pass

    def swim (self, index):
        pass

    def insert(self, index):
        pass

    def getMax(self, index):
        pass
```

Helpers

```
# HELPER FUNCTIONS
def parent (index):
    return index // 2

def left (index):
    return index * 2

def right (index):
    return index * 2 + 1

def swap (array, index1, index2):
    array[index1], array[index2] = array[index2], array[index1]
```

Max Heap Construction

Initialisation

pq = MaxHeap(10)

```
def __init__ (self, maxsize):
    self.maxsize = maxsize
    self.size = 0
    self.heap = [None] * (maxsize + 1)
```

pq = MaxHeap(10)

```
def __init__ (self, maxsize):
    self.maxsize = maxsize
    self.size = 0
    self.heap = [None] * (maxsize + 1)
```

Insertion

STEP 1: Check if capacity is reached

pq.insert(30)

```
if self.size >= self.maxsize:
    print('size limit reached')
    return
```

STEP 2: Increment size and insert into heap at index size

pq.insert("Google", 30)

```
self.size += 1
self.heap[self.size] = HeapItem(newKey,
newValue)
```

STEP 2: Increment size and insert into heap at index size

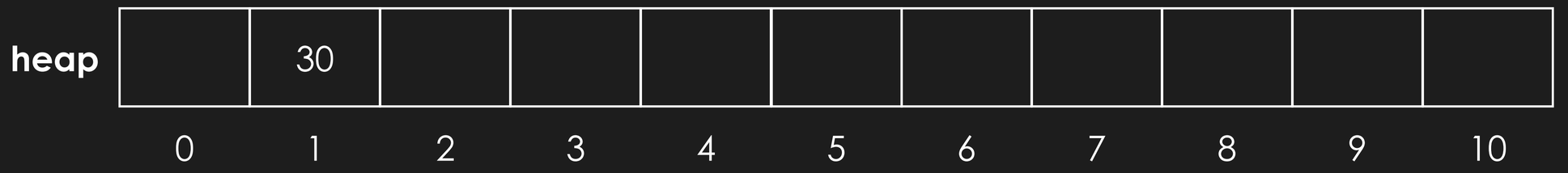
pq.insert("Google", 30)

```
self.size += 1
self.heap[self.size] = HeapItem(newKey,
newValue)
```

STEP 3: While parent node has higher priority, swap with parent node

pq.insert("Google", 30)

```
self.swim(self.size)
```



STEP 1: Check if capacity is reached

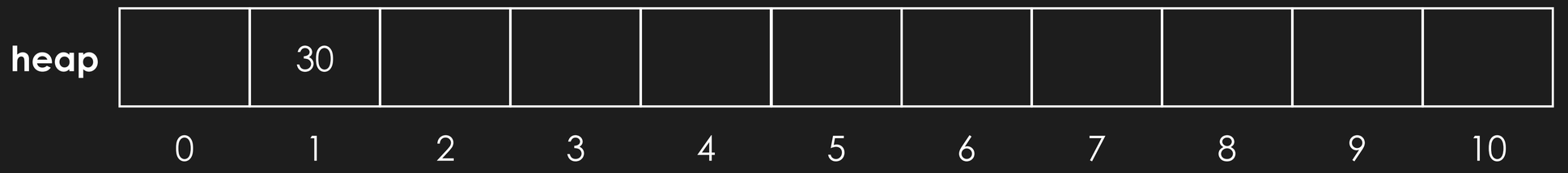
pq.insert("Amazon", 20)

```
if self.size >= self.maxsize:
    print('size limit reached')
    return
```

STEP 2: Increment size and insert into heap at index size

pq.insert("Amazon", 20)

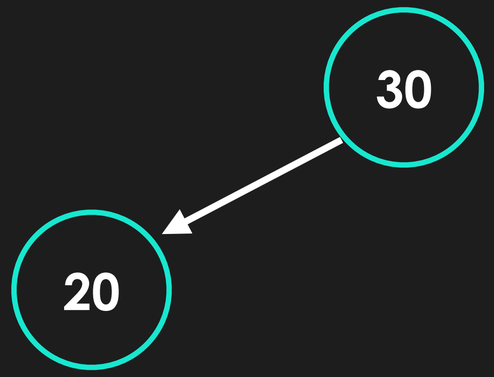
```
self.size += 1
self.heap[self.size] = HeapItem(newKey,
newValue)
```



STEP 2: Increment size and insert into heap at index size

pq.insert("Amazon", 20)

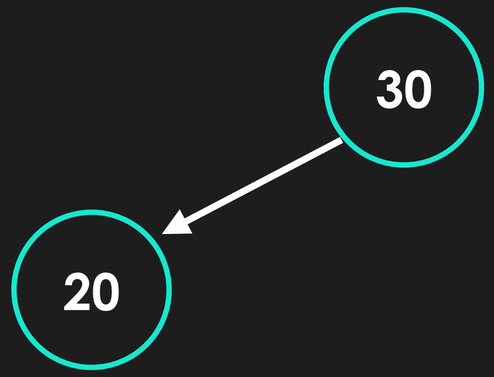
```
self.size += 1
self.heap[self.size] = HeapItem(newKey,
newValue)
```



STEP 3: While parent node has higher priority, swap with parent node

pq.insert("Amazon", 20)

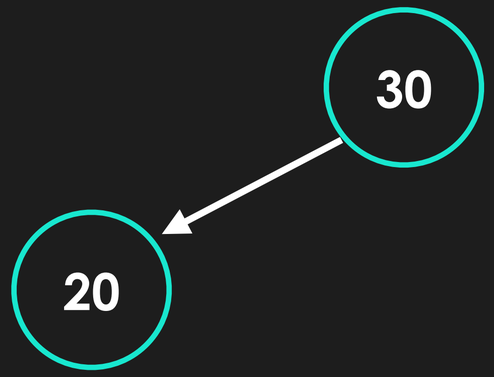
```
self.swim(self.size)
```



STEP 1: Check if capacity is reached

pq.insert("Apple", 60)

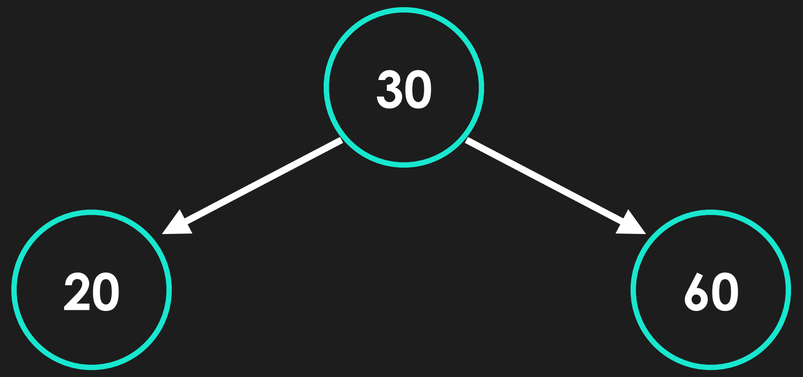
```
if self.size >= self.maxsize:
    print('size limit reached')
    return
```



STEP 2: Increment size and insert into heap at index size

pq.insert("Apple", 60)

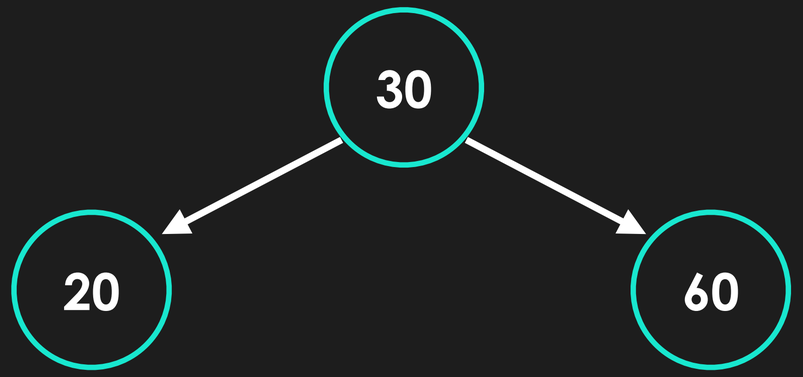
```
self.size += 1
self.heap[self.size] = HeapItem(newKey,
newValue)
```



STEP 3: While parent node has higher priority, swap with parent node

pq.insert("Apple", 60)

```
self.swim(self.size)
```



swim

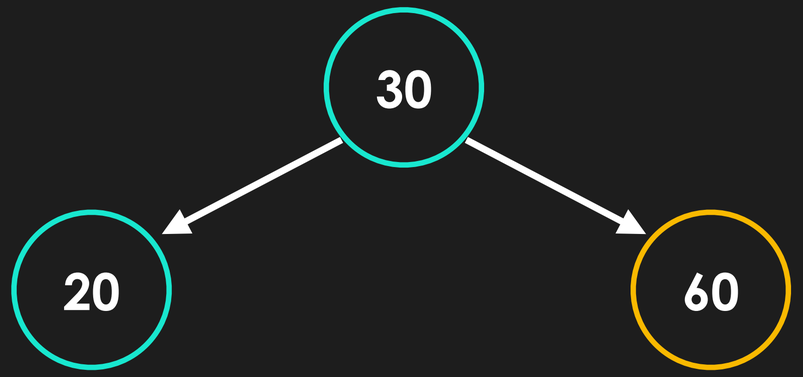
greater

```
def greater(self, index1, index2):
    if self.heap[index1].value > self.heap[index2].value:
        return True
    else:
        return False
```

We define a method **greater** which returns **True** if HeapItem at **index1** has a greater value (higher priority) then HeapItem at **index2**

swim(3)

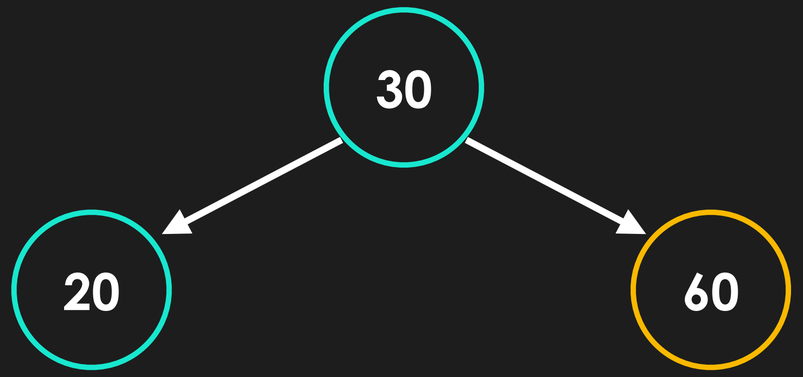
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 3

swim(3)

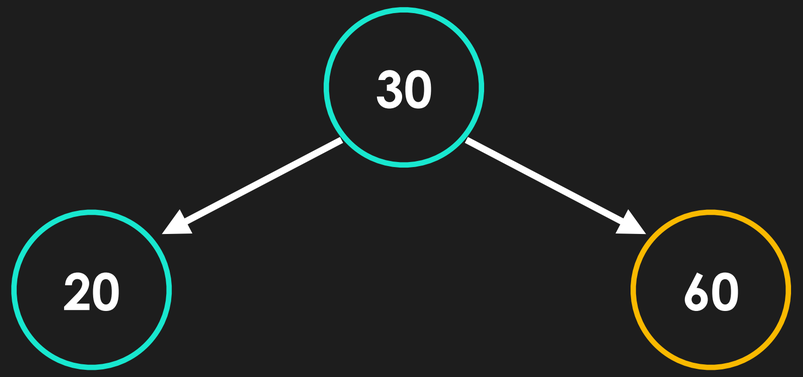
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 3

swim(3)

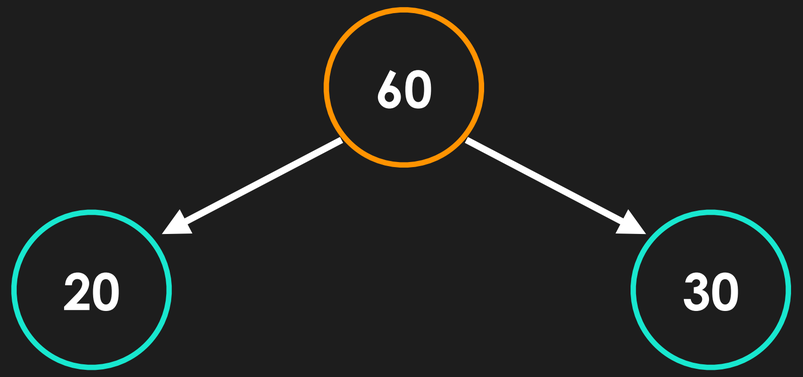
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 3

swim(3)

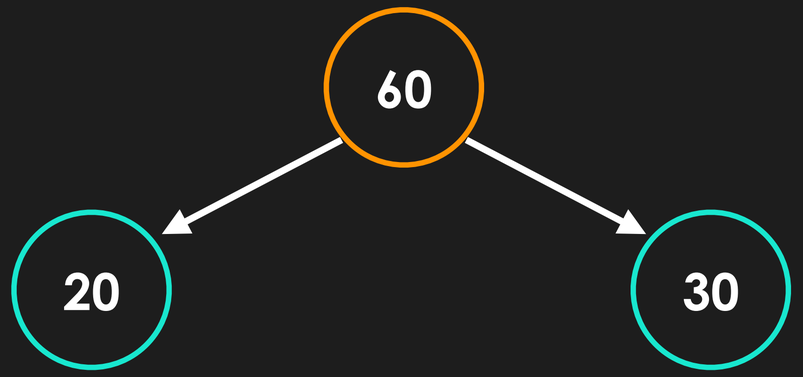
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 3

swim(3)

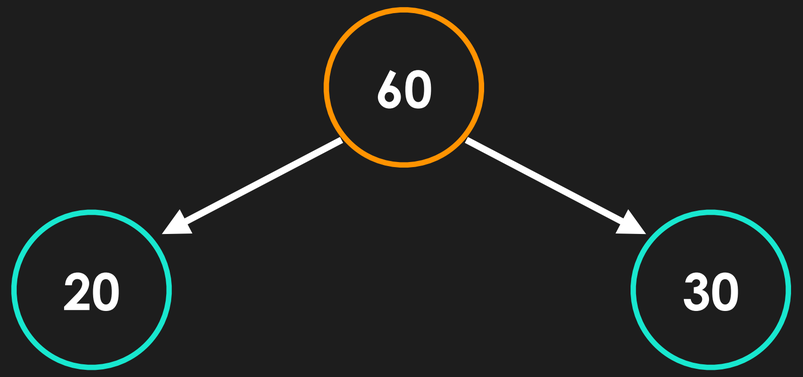
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 3

swim(3)

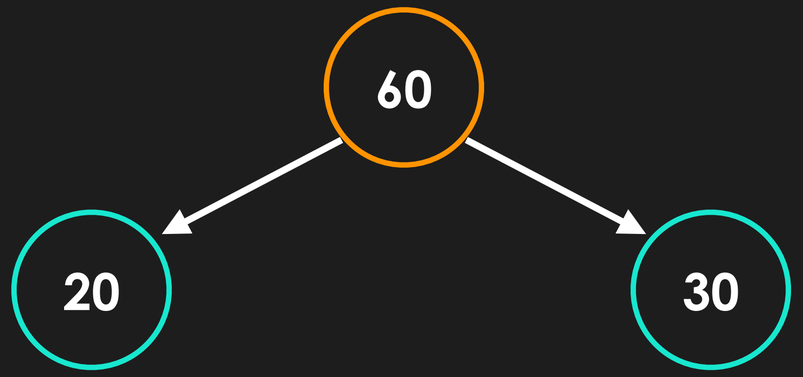
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 1

swim(3)

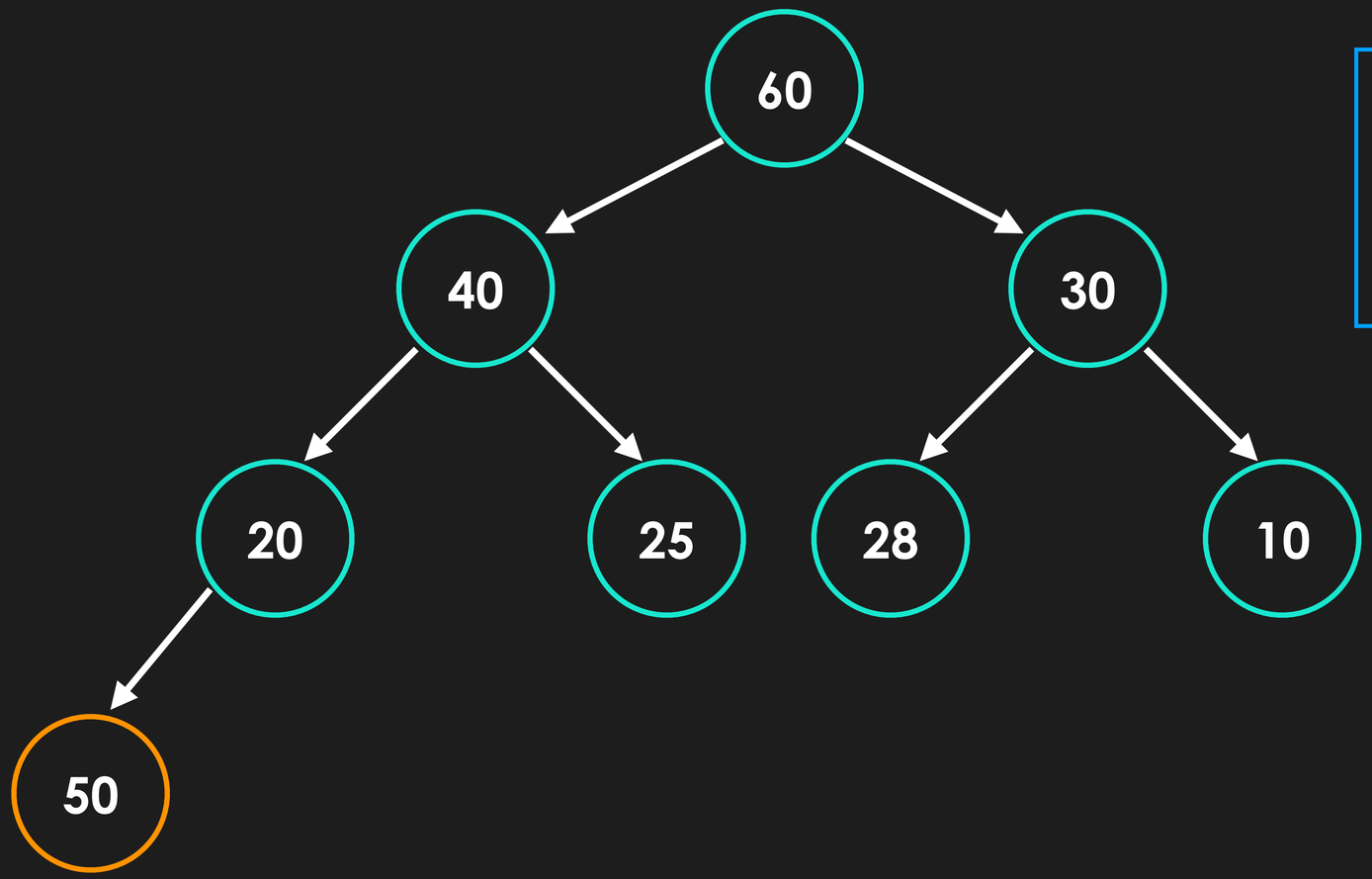
```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```



index = 1

swim example 2

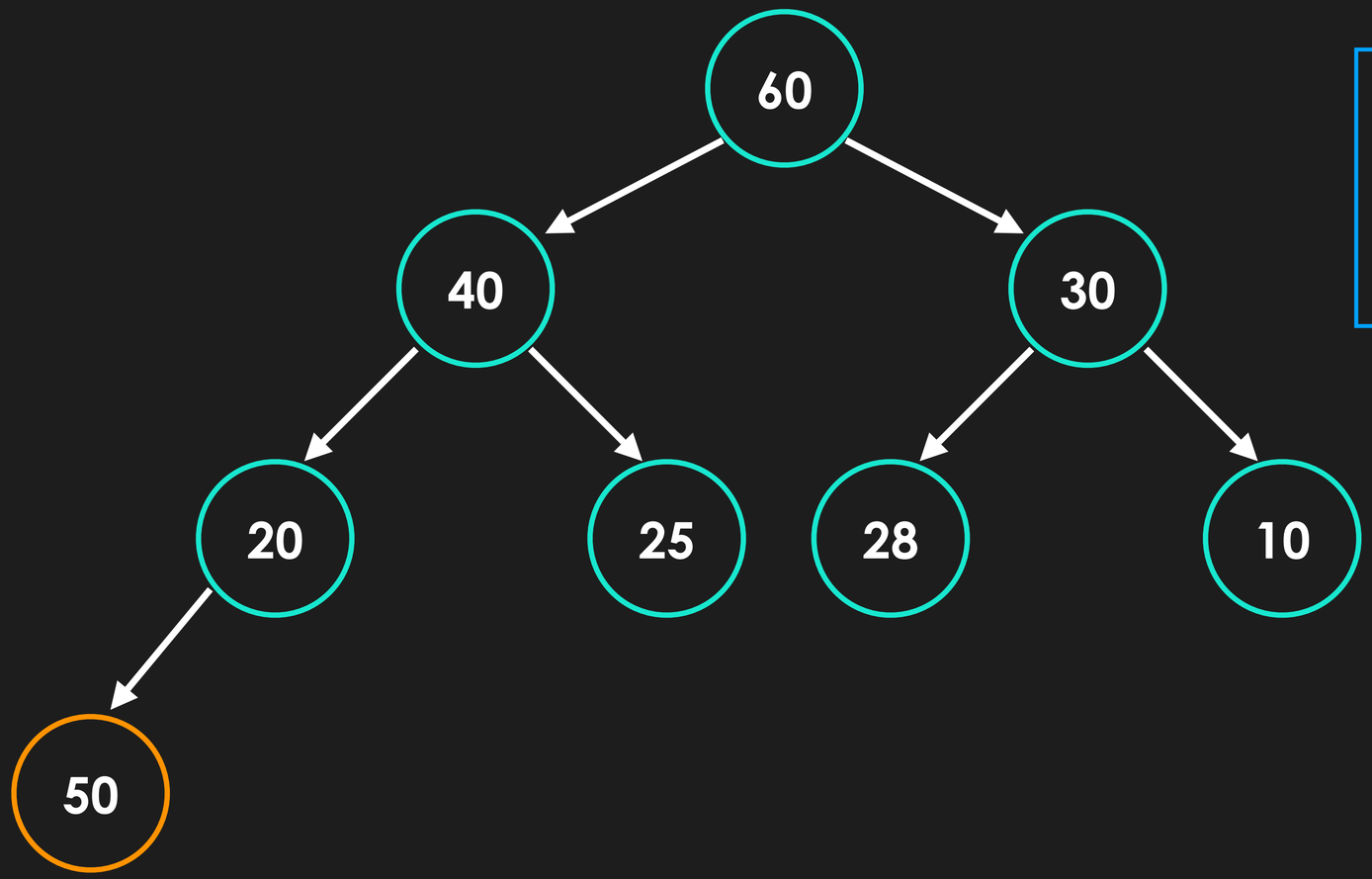
swim(8)




```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 8

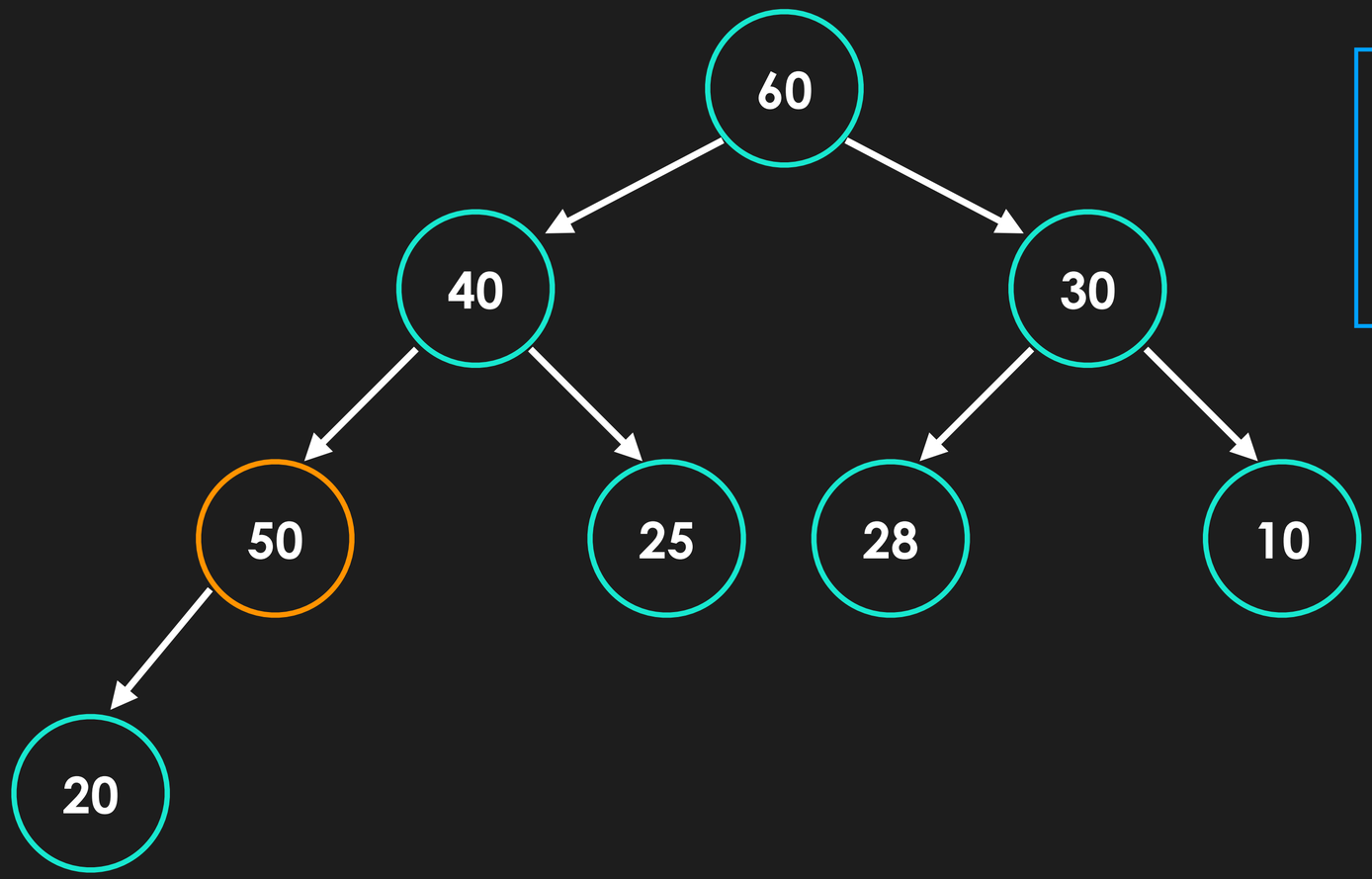
swim(8)




```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 8

swim(8)

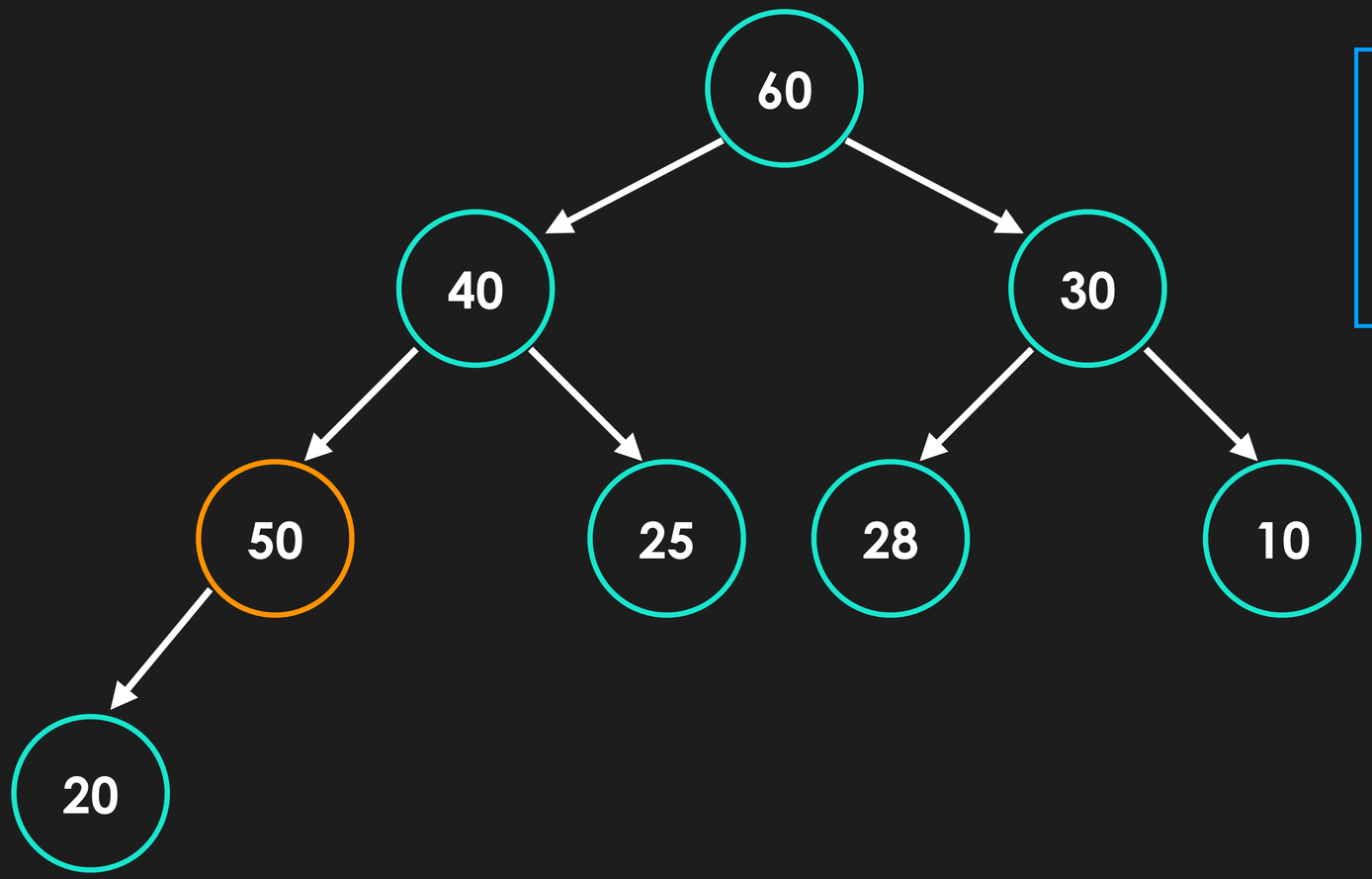



```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 8

heap		60	40	30	50	25	28	10	20		
	0	1	2	3	4	5	6	7	8	9	10

swim(8)

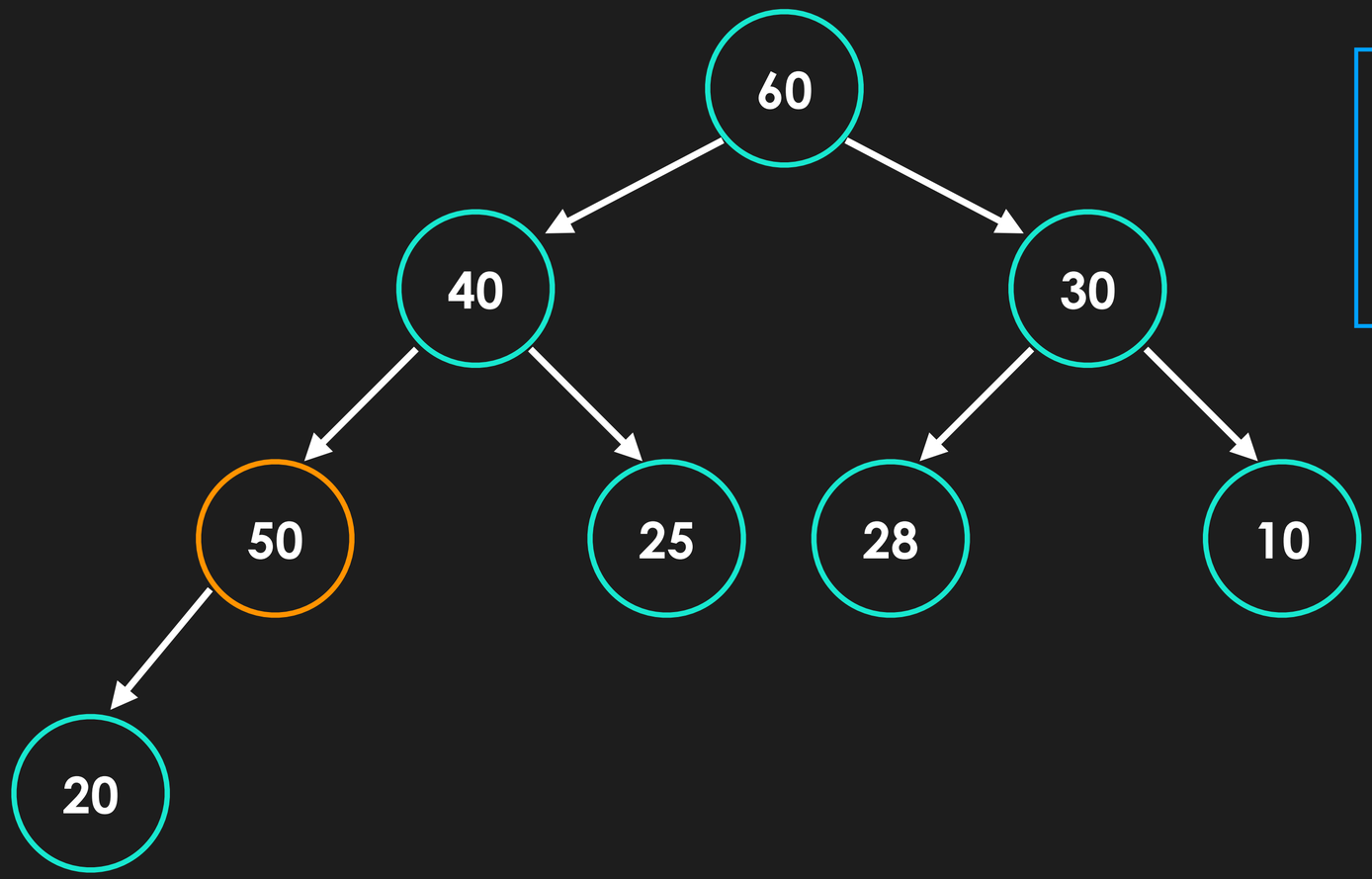



```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 4

heap		60	40	30	50	25	28	10	20		
	0	1	2	3	4	5	6	7	8	9	10

swim(8)

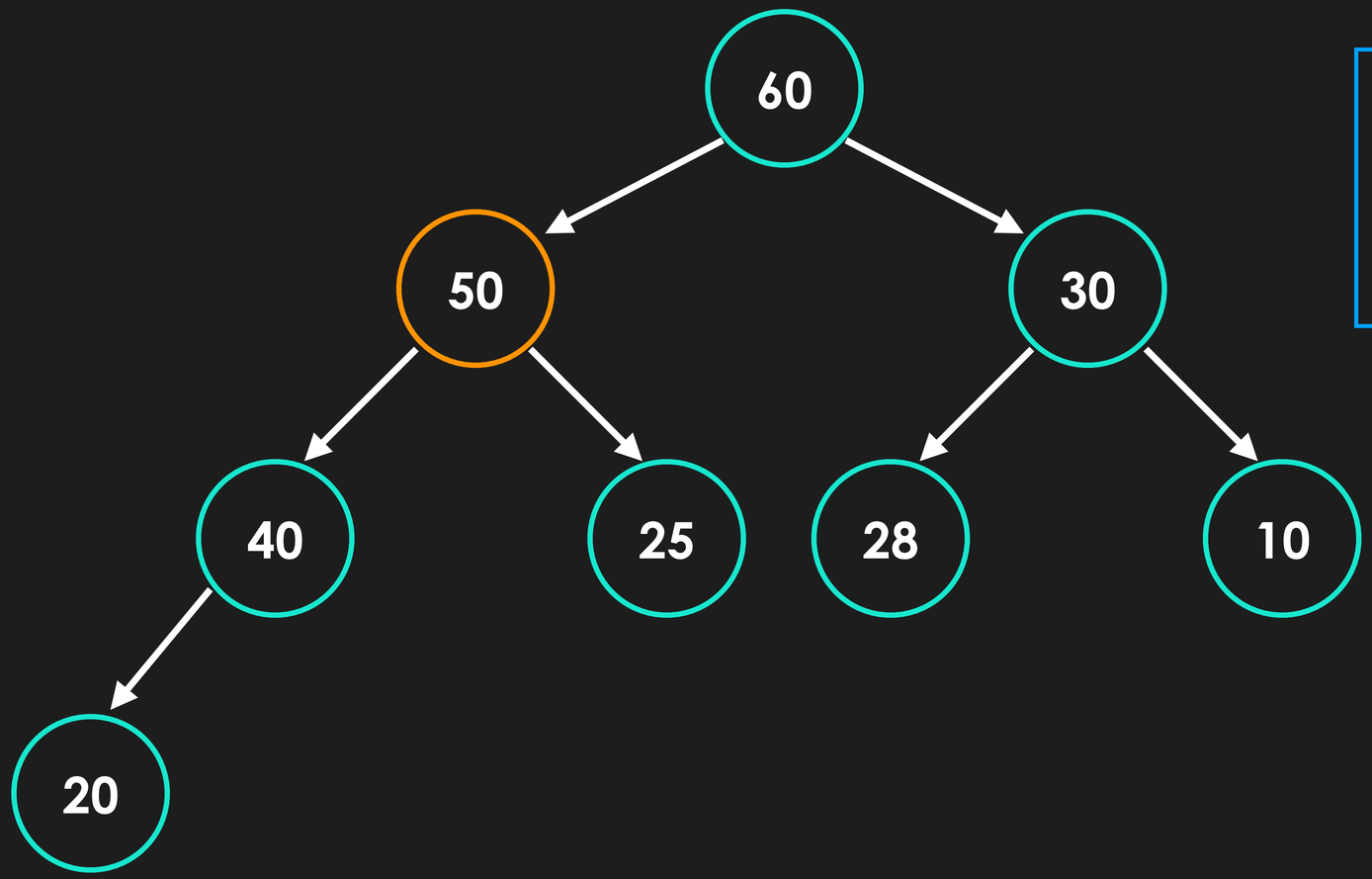



```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 4

heap		60	40	30	50	25	28	10	20		
	0	1	2	3	4	5	6	7	8	9	10

swim(8)

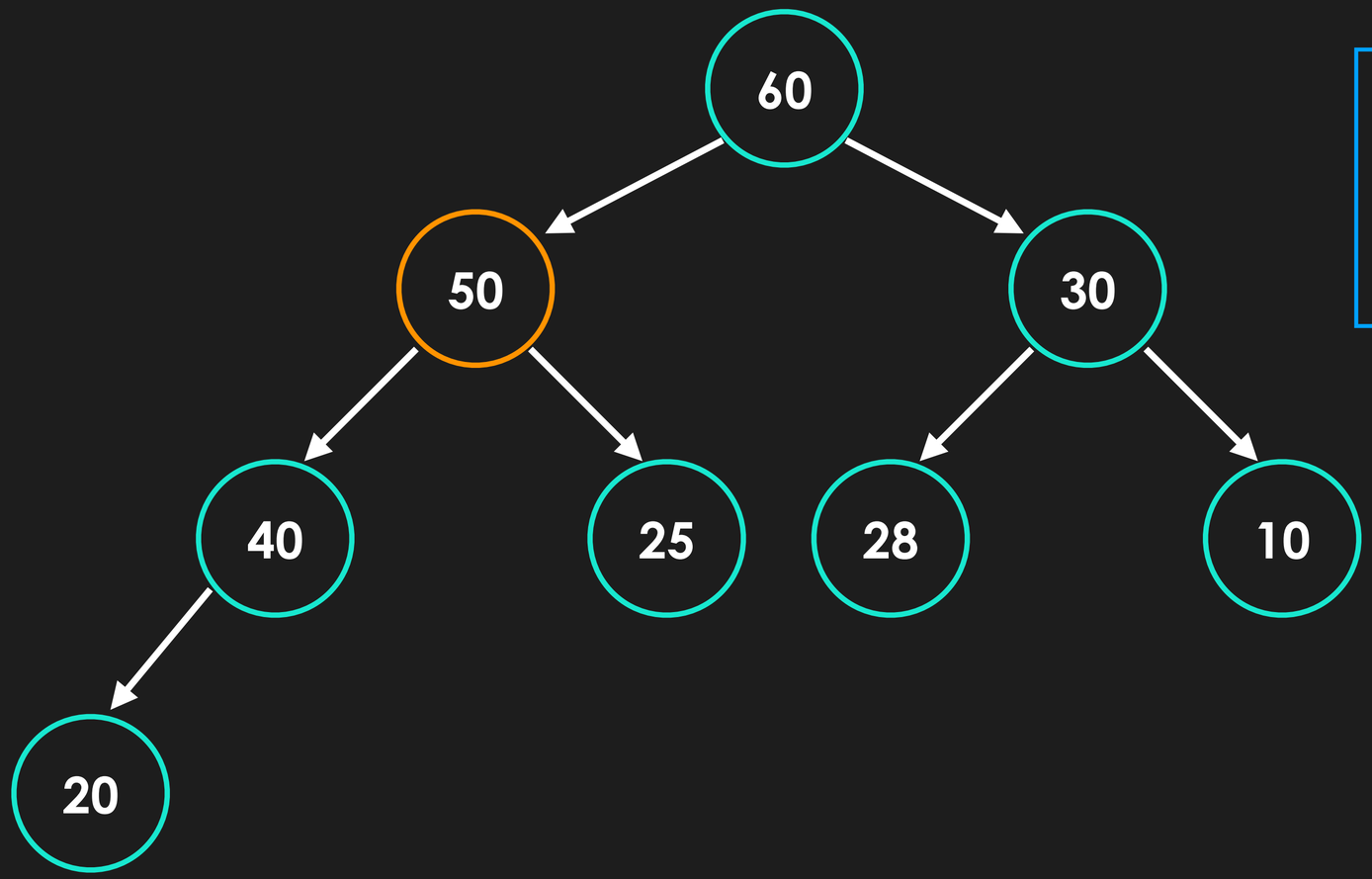



```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 4

CodeIT

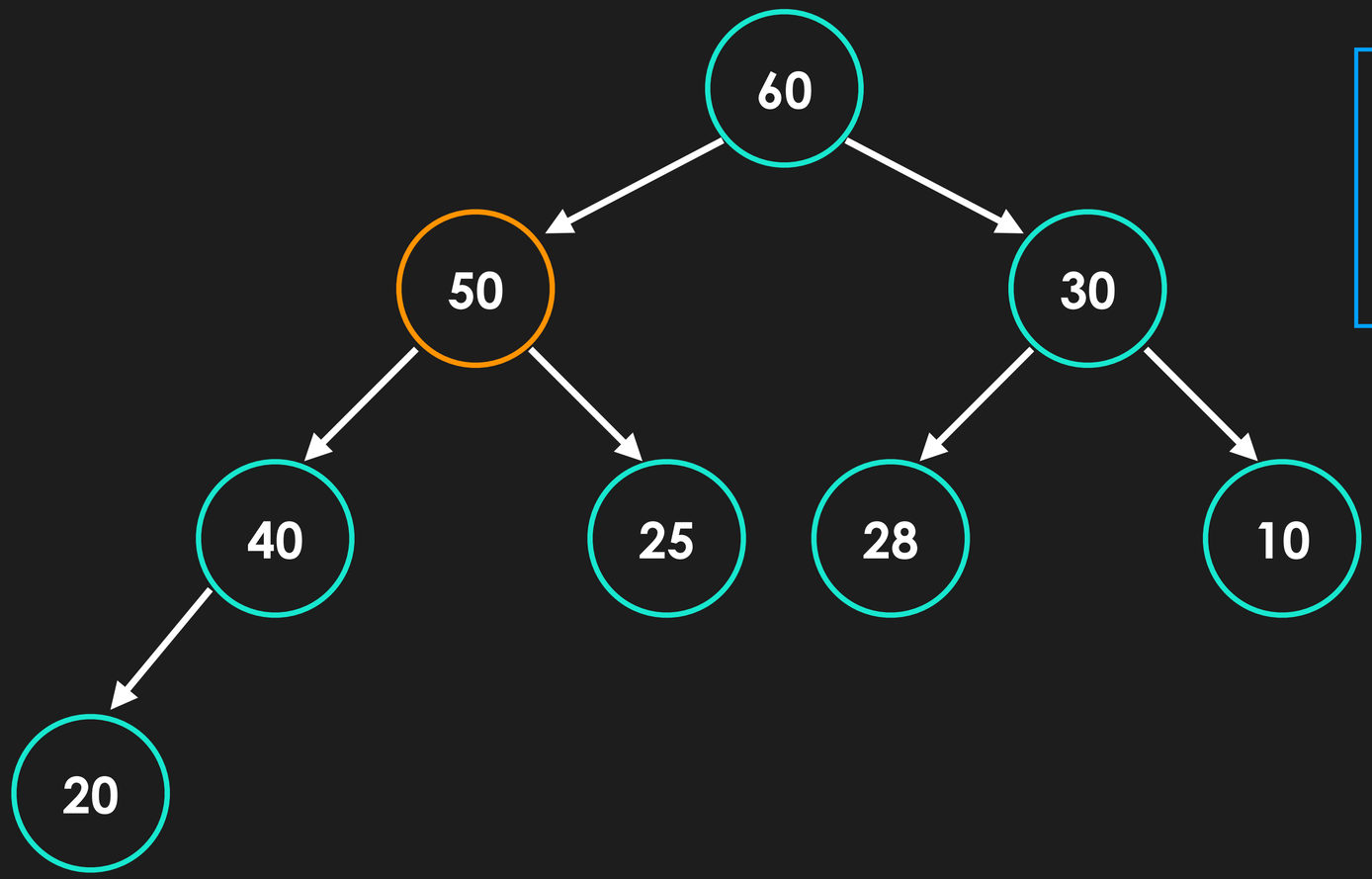
swim(8)




```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 2

swim(8)




```
while (index != 1 and self.greater(index,
parent(index))):
    swap(self.heap, index, parent(index))
    index = parent(index)
```

index = 2

MaxHeap Insertion

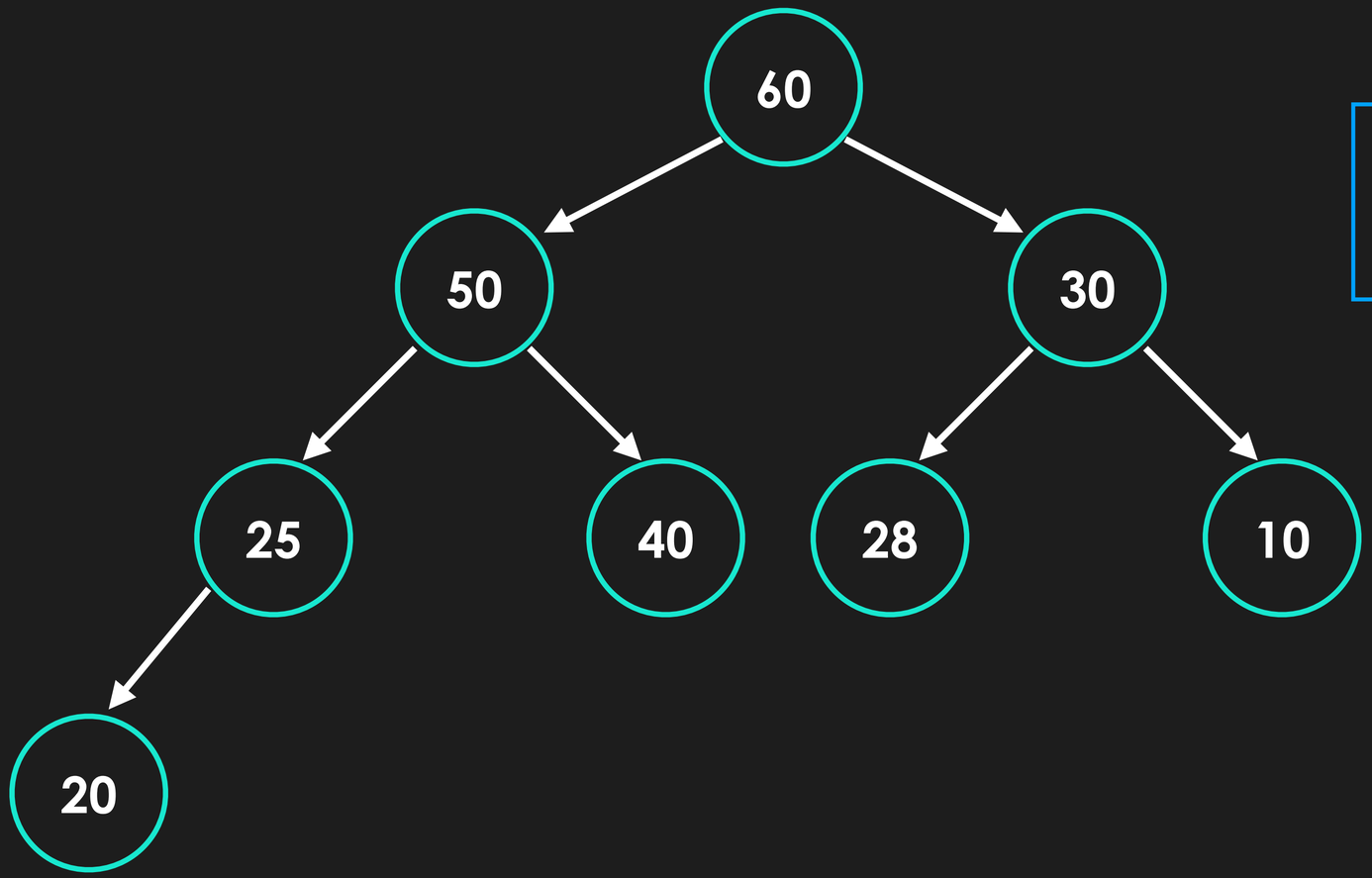
```
def swim (self, index):
    while (index != 1 and self.greater(index, parent(index))):
        swap(self.heap, index, parent(index))
        index = parent(index)
    return

def insert(self, newKey, newValue):
    if self.size >= self.maxsize:
        print('size limit reached')
        return

    self.size += 1
    self.heap[self.size] = HeapItem(newKey, newValue)
    self.swim(self.size)
```

Max Extraction

Max Extraction

- **Remove** max element
- **Ensure** Heap order is maintained

getMax()

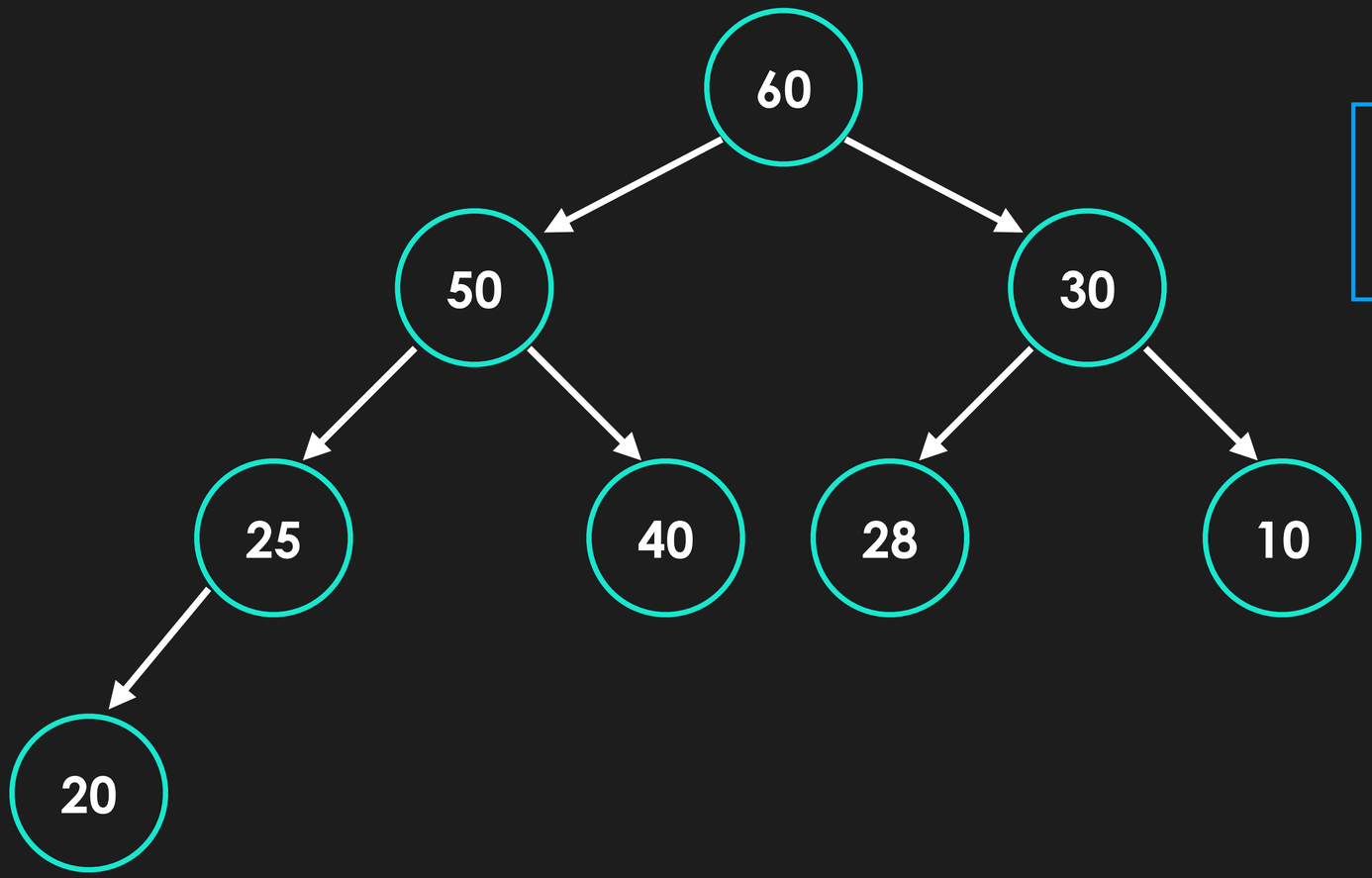



```
maxItem = self.heap[1]
```

STEP 1: Save max element into a variable

heap		60	50	30	25	40	28	10	20		
	0	1	2	3	4	5	6	7	8	9	10

getMax()

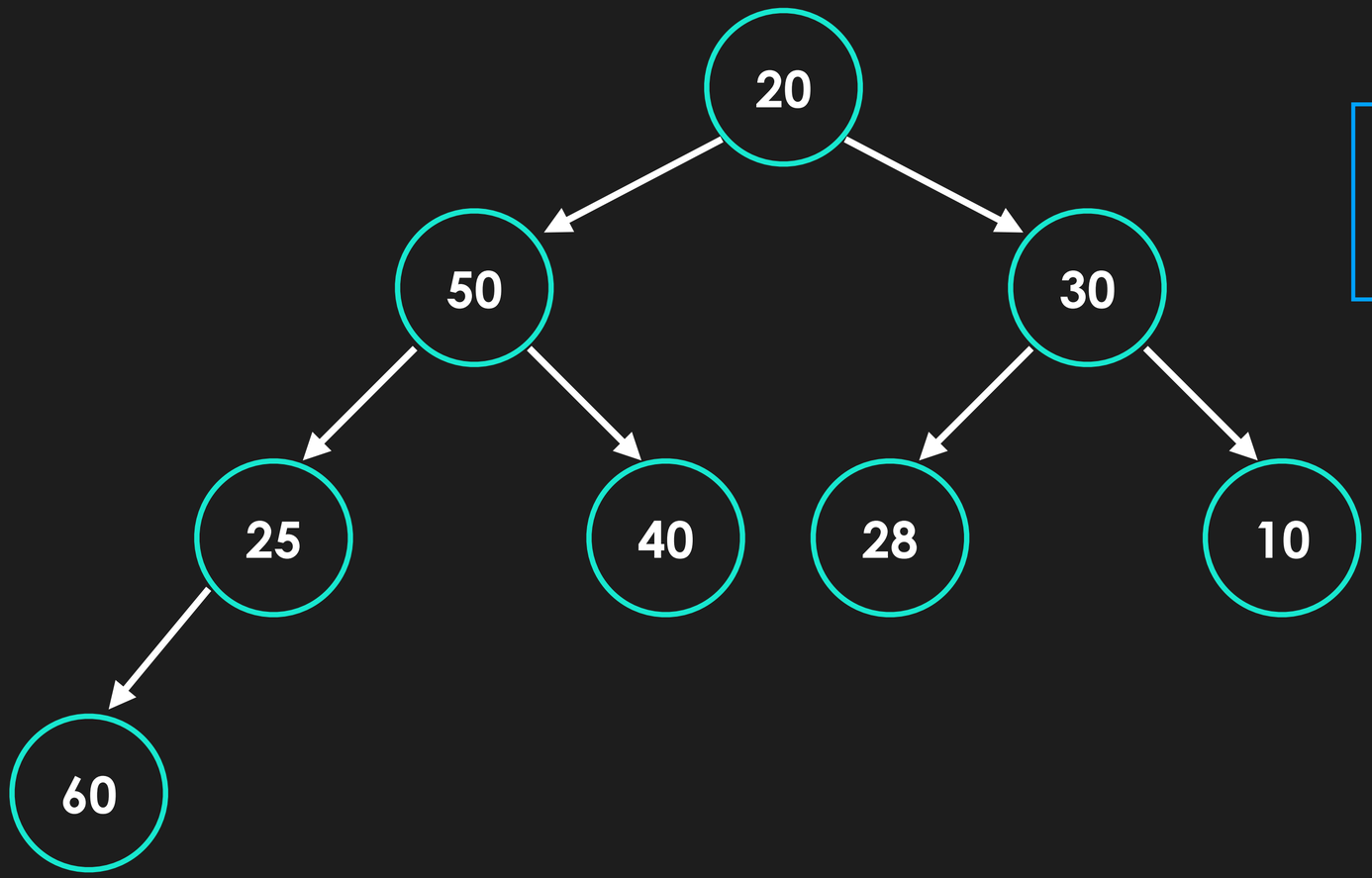



```
swap(self.heap, 1, self.size)
```

STEP 2: Swap first and last element

heap		60	50	30	25	40	28	10	20		
	0	1	2	3	4	5	6	7	8	9	10

getMax()

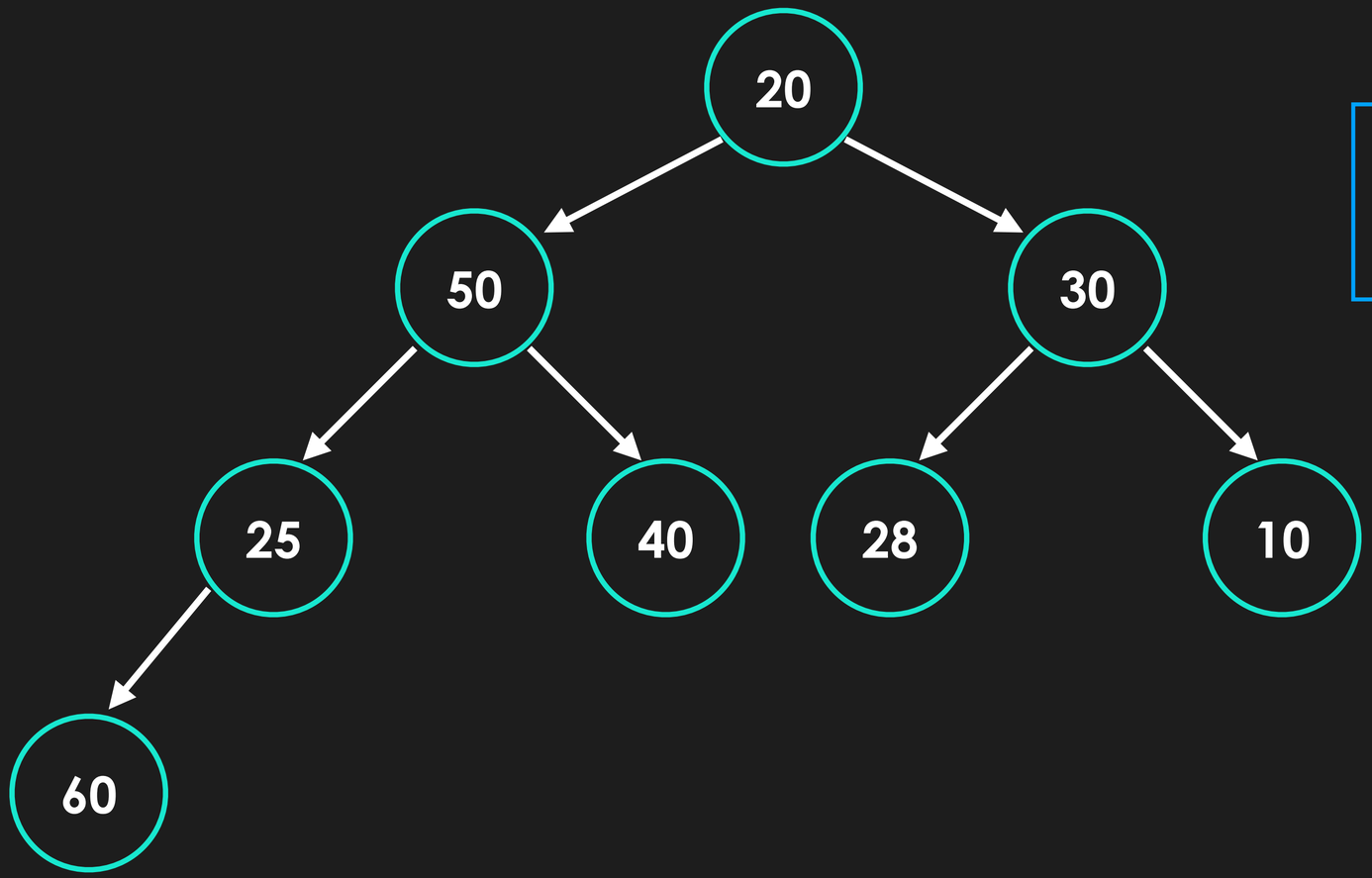



```
swap(self.heap, 1, self.size)
```

STEP 2: Swap first and last element

heap		20	50	30	25	40	28	10	60		
	0	1	2	3	4	5	6	7	8	9	10

getMax()

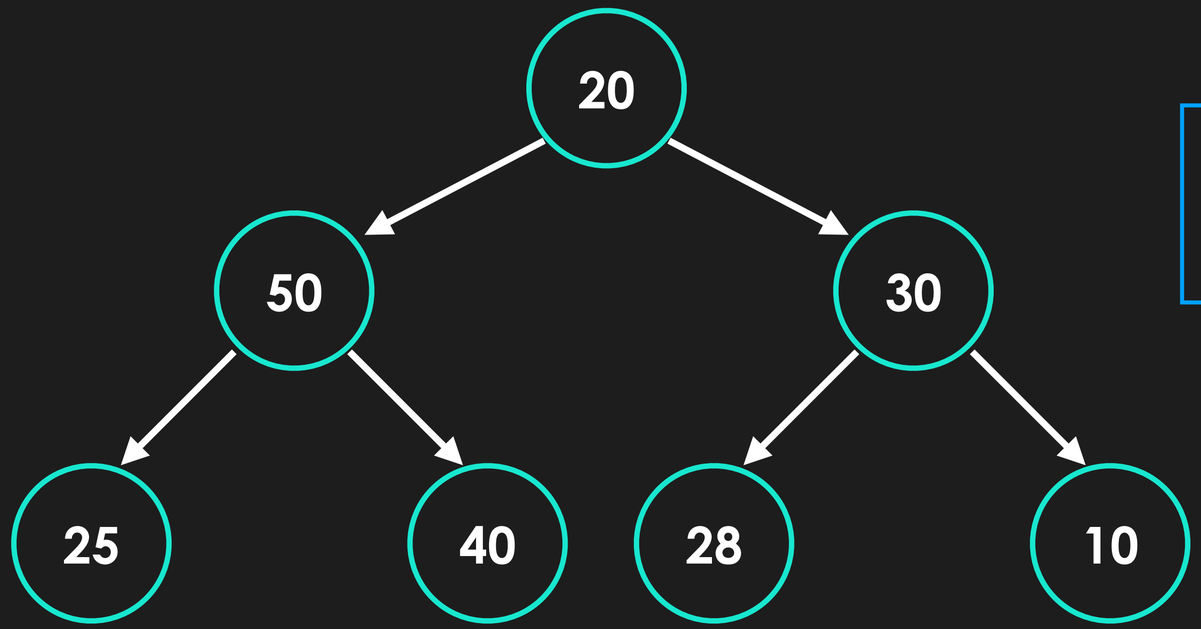



```
self.heap[self.size] = None
self.size -= 1
```

STEP 3: Remove last element

heap		20	50	30	25	40	28	10	60		
	0	1	2	3	4	5	6	7	8	9	10

getMax()

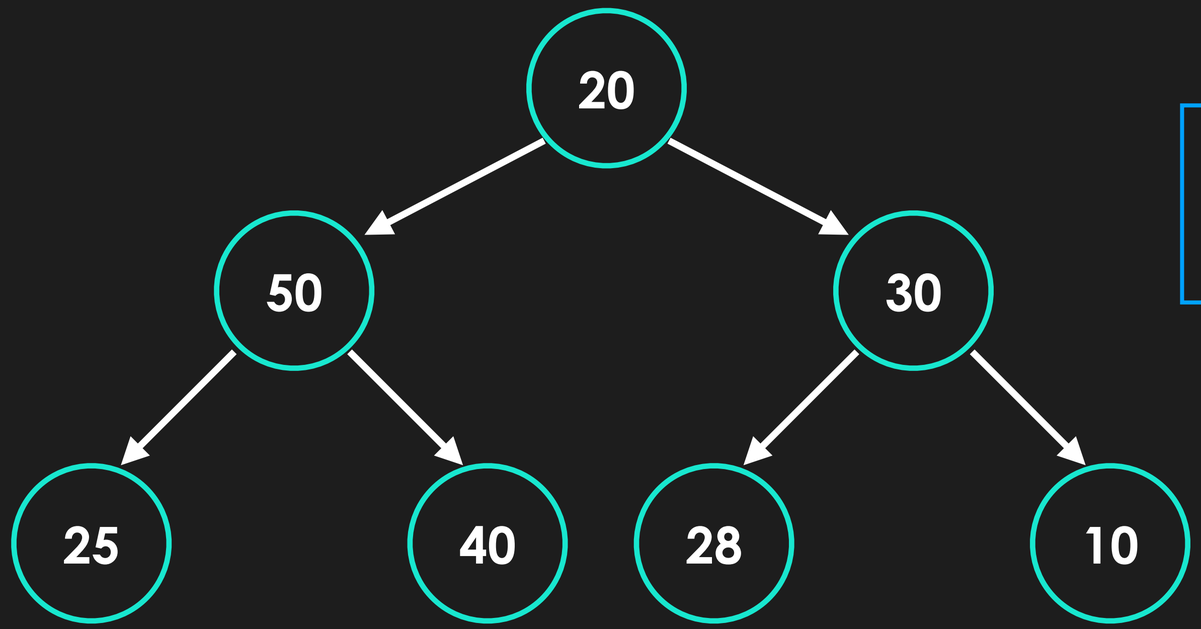



```
self.heap[self.size] = None
self.size -= 1
```

STEP 3: Remove last element

heap		20	50	30	25	40	28	10			
	0	1	2	3	4	5	6	7	8	9	10

getMax()



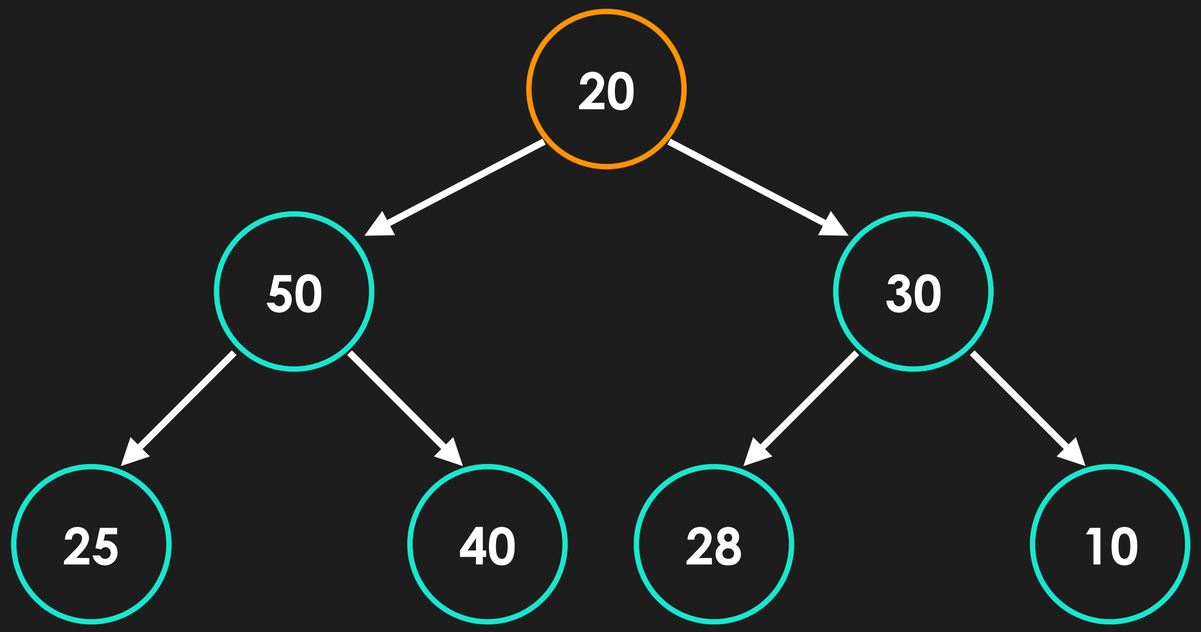

```
self.sink(1)
```

STEP 4: Sink first element, (swap with child node if priority is lower)

heap		20	50	30	25	40	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink

sink(1)

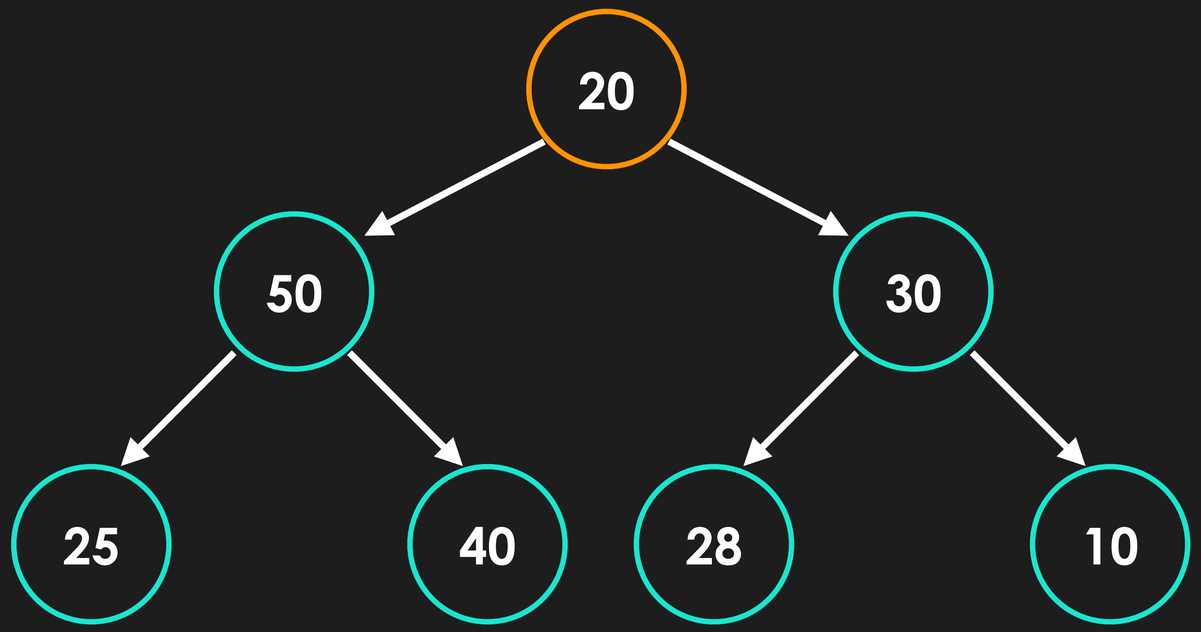



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		20	50	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

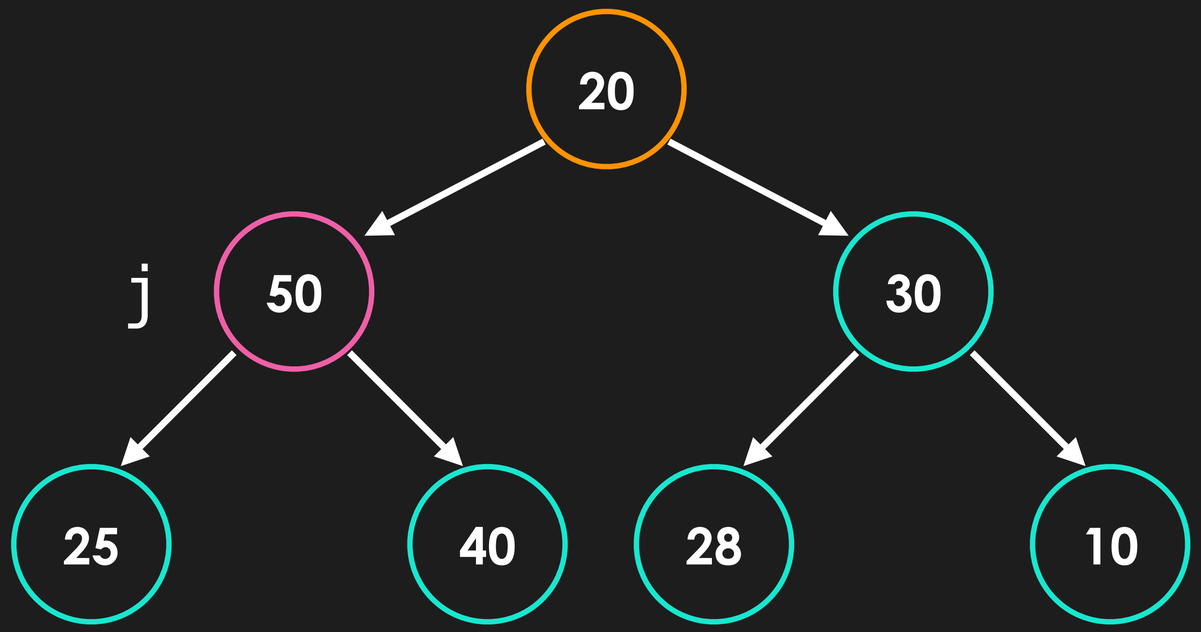



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		20	50	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

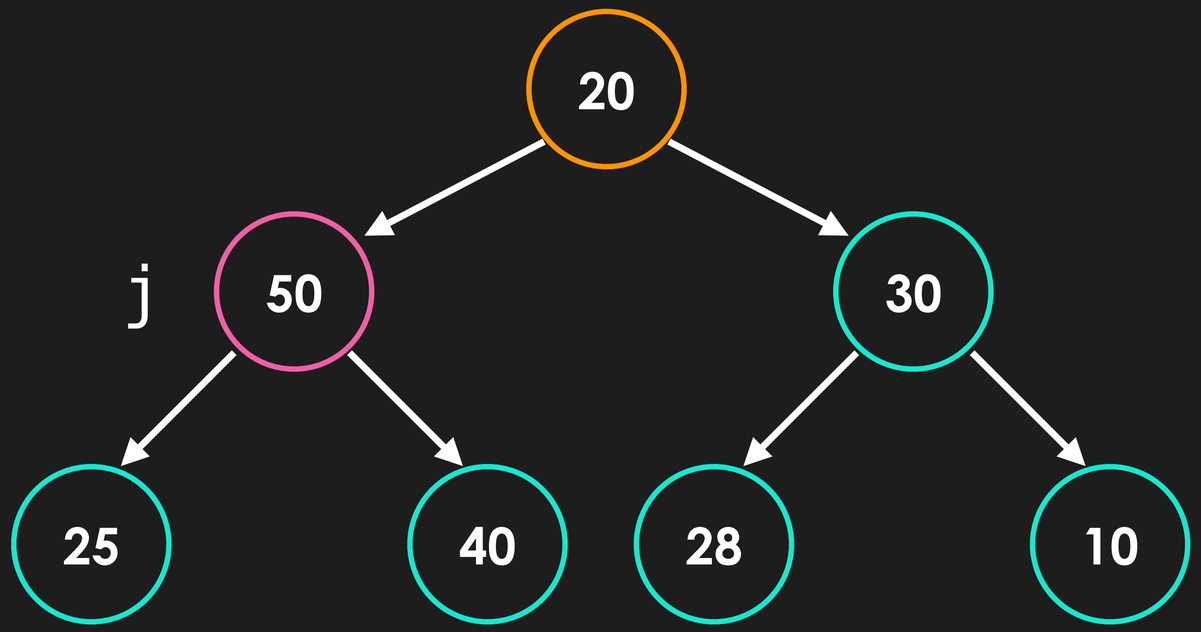



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		20	50	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

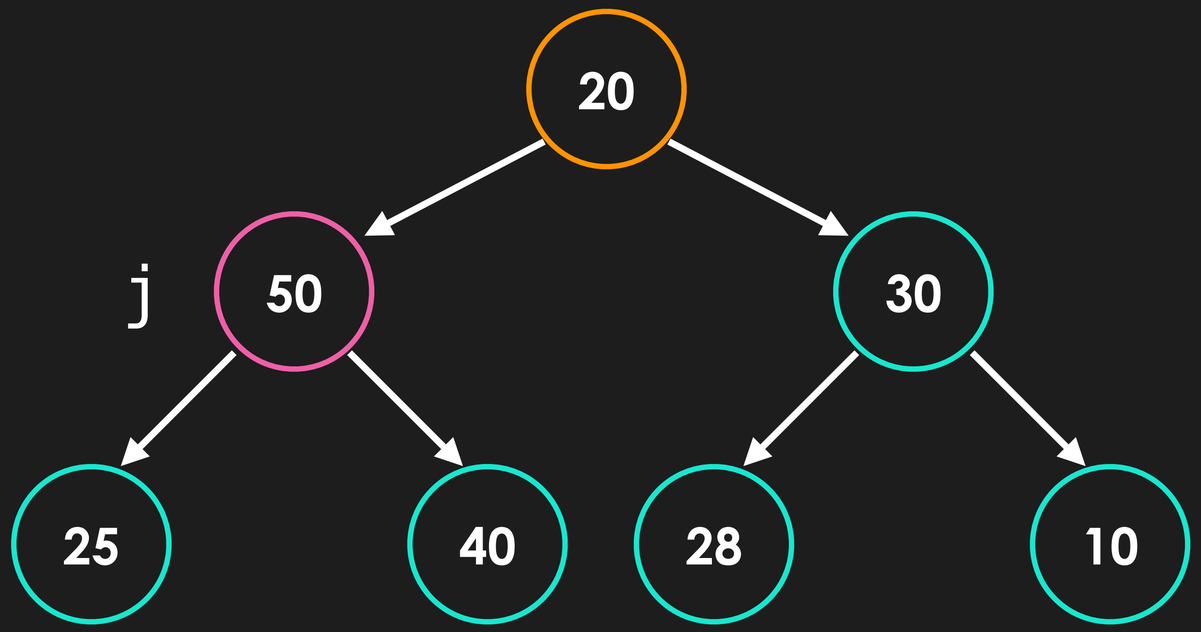



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		20	50	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

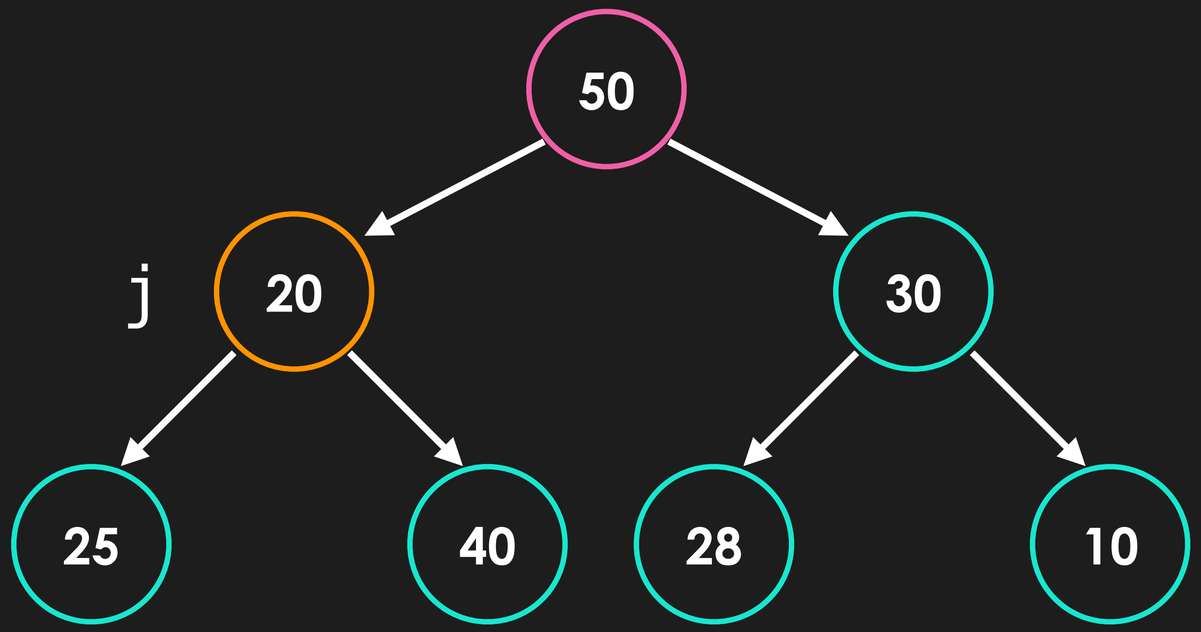



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		20	50	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

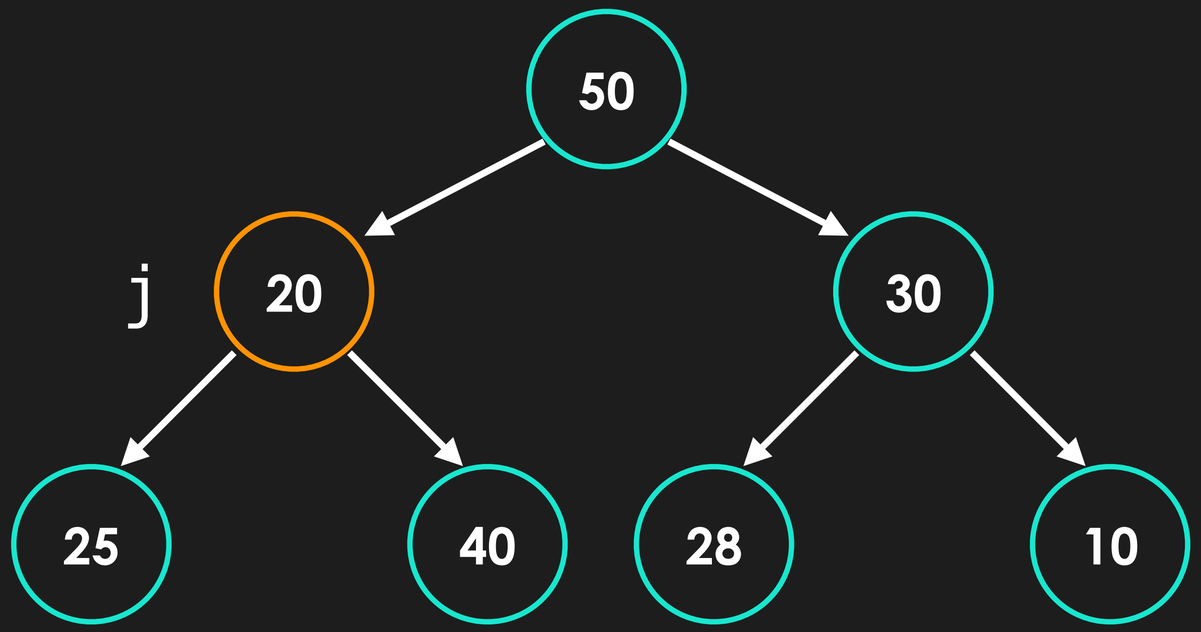



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 1

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

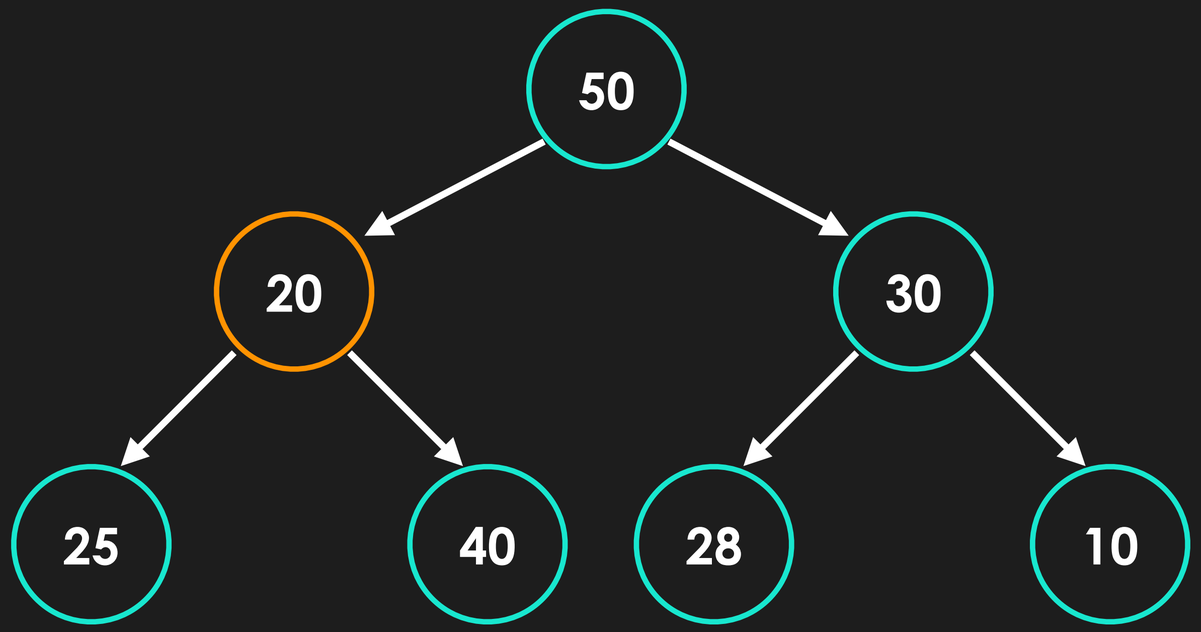



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

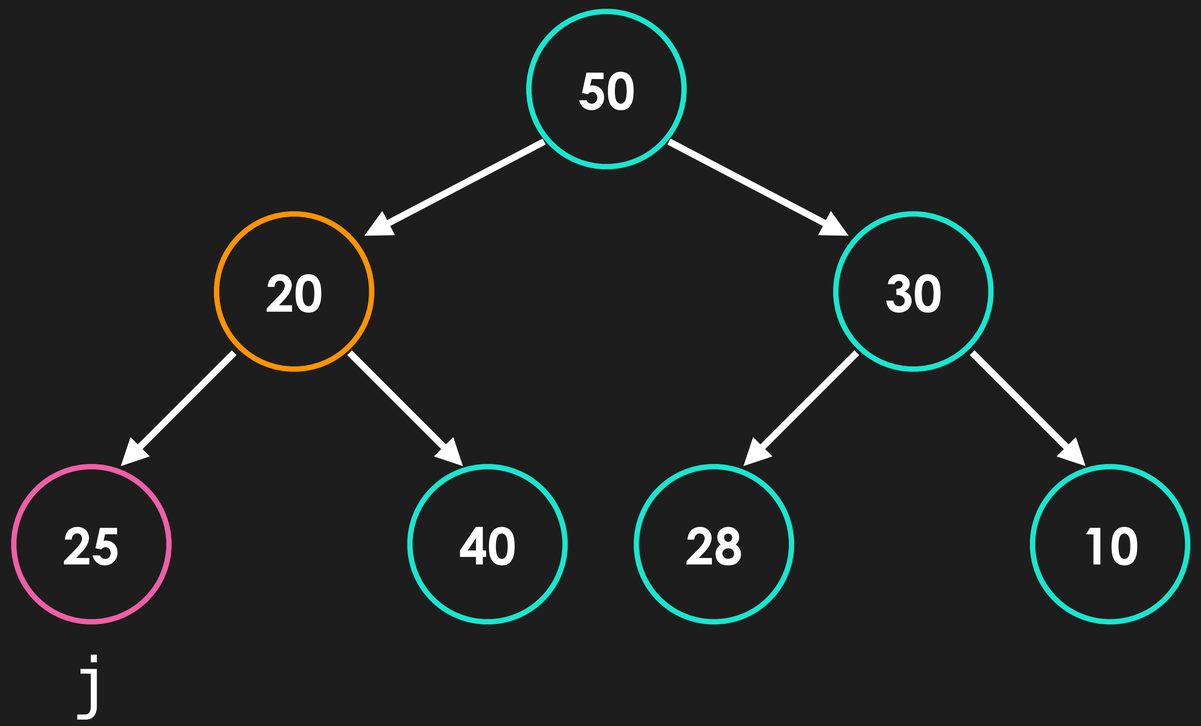



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

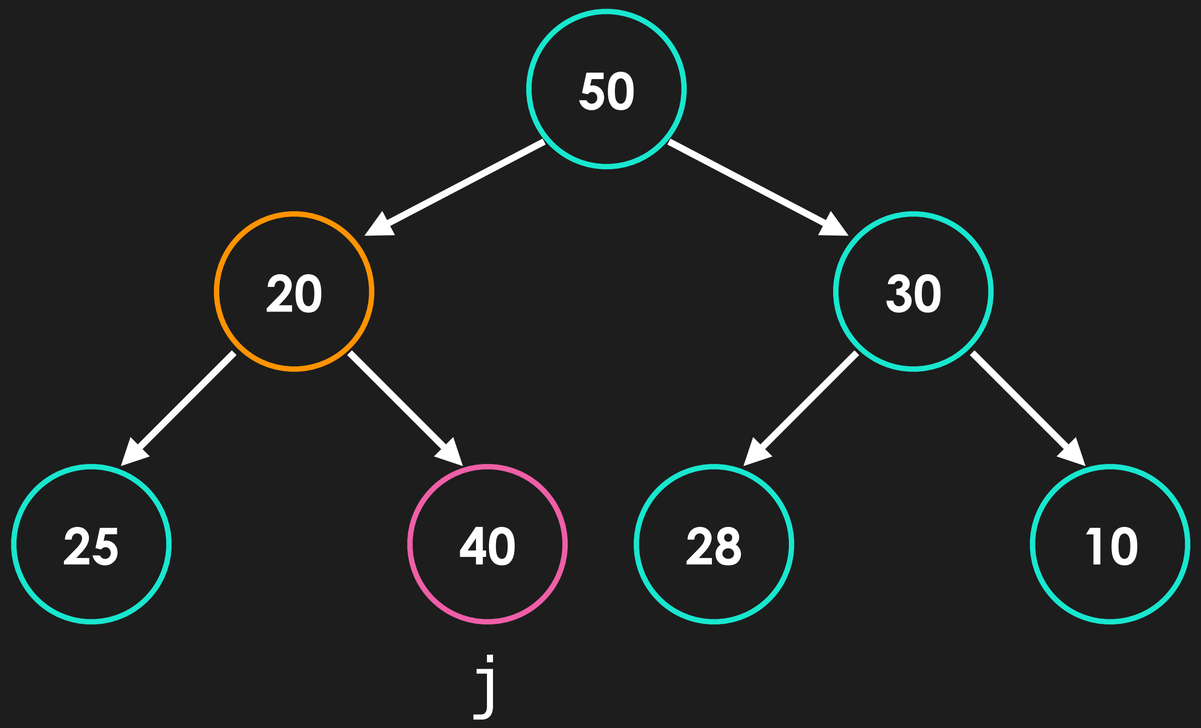



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

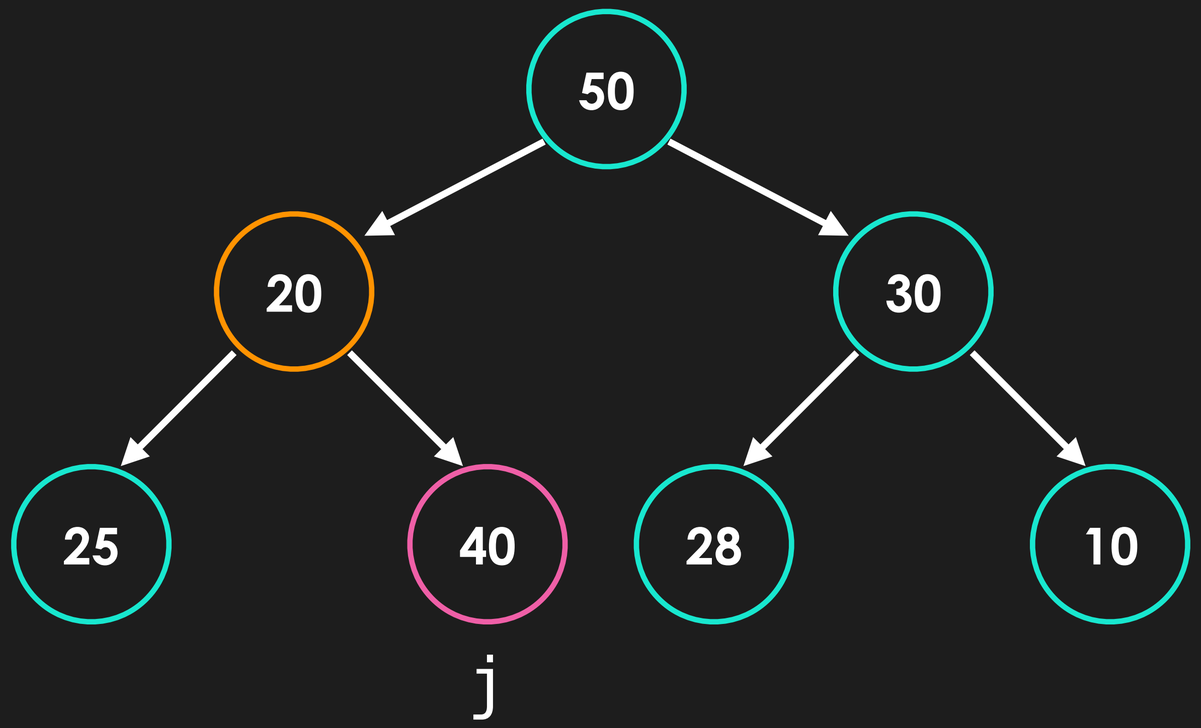



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

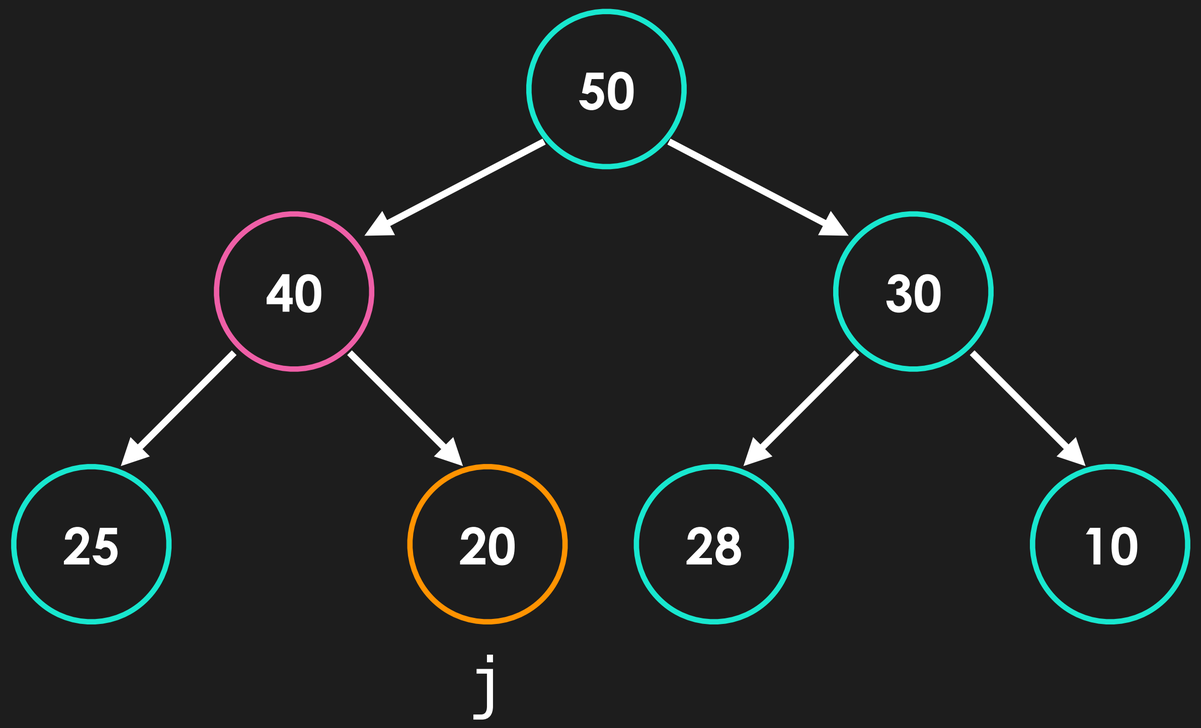



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	20	30	40	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

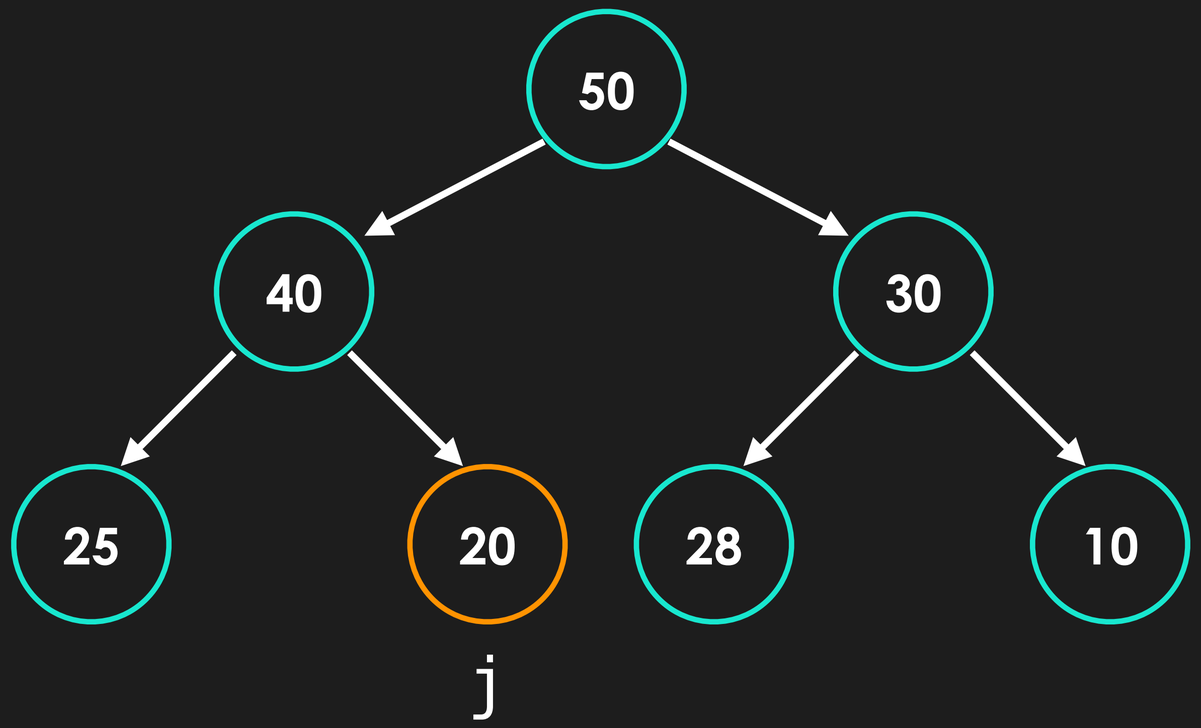



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 2

heap		50	40	30	20	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)

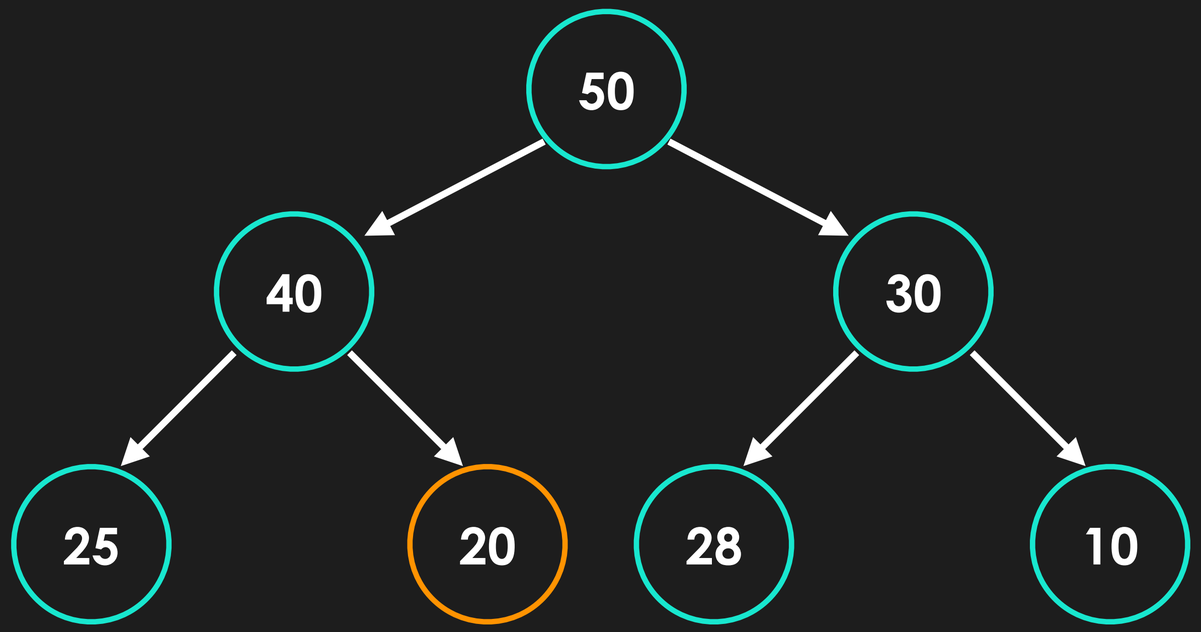



```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 4

heap		50	40	30	20	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink(1)




```
while (2 * index <= self.size):

    j = left(index)

    if (j < self.size and
    self.greater(right(index), j)):
        j = right(index)

    if self.greater(index, j):
        break

    else:
        swap(self.heap, j, index)
        index = j
```

index = 4

heap		50	40	30	20	25	28	10			
	0	1	2	3	4	5	6	7	8	9	10

sink

```
def sink(self, index):
    while (2 * index <= self.size):

    # initialize j to be left child
        j = left(index)

        # compare left and right
        if (j < self.size and self.greater(right(index), j)):
            j = right(index)

        if self.greater(index, j):
            break

        else:
            swap(self.heap, j, index)
            index = j

    return
```

getMax

```
def getMax(self):

    if self.size == 0:
        return None

    # SAVE MAX ITEM FOR REMOVAL
    maxItem = self.heap[1]

    # SWAP MAX ITEM
    swap(self.heap, self.positions, 1, self.size)

    # REMOVE ITEM
    self.heap[self.size] = None
    self.size -= 1

    # SINK FIRST ITEM
    self.sink(1)
    return maxItem
```

Putting it all together

```
pq = MaxHeap(10)
pq.insert(30, "Google")
pq.insert(20, "Amazon")
pq.insert(60, "Apple")
pq.insert(40, "Microsoft")
pq.insert(25, "Netflix")

while pq.size != 0:
    maxItem = pq.getMax()
    print(maxItem.key, maxItem.value)
```

```
pq = MaxHeap(10)
pq.insert(30, "Google")
pq.insert(20, "Amazon")
pq.insert(60, "Apple")
pq.insert(40, "Microsoft")
pq.insert(25, "Netflix")

while pq.size != 0:
    maxItem = pq.getMax()
    print(maxItem.key, maxItem.value)
```

```
Apple 60
Microsoft 40
Google 30
Netflix 25
Amazon 20
```

Note: A heap doesn't **necessarily have to contain key-value pairs**. It can contain, for instance, just a **simple array of numbers**. The logic of a heap to extract min / max is essential to many logical processes!

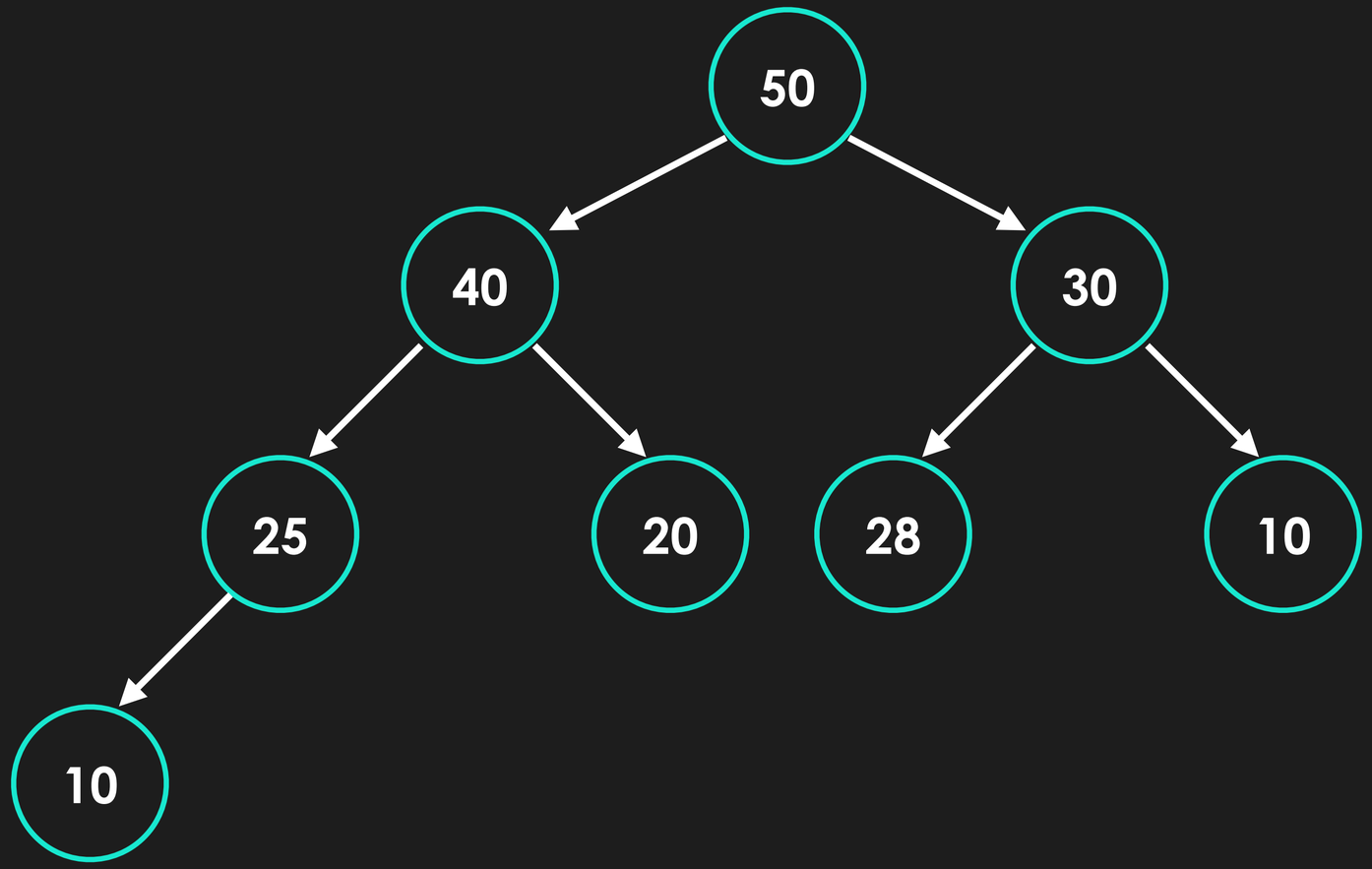
Lab Session 1

- Implement **min_heap1.py**
- To test, run `**python utils/mh1_test.py**`

Analysis of Heap Operations

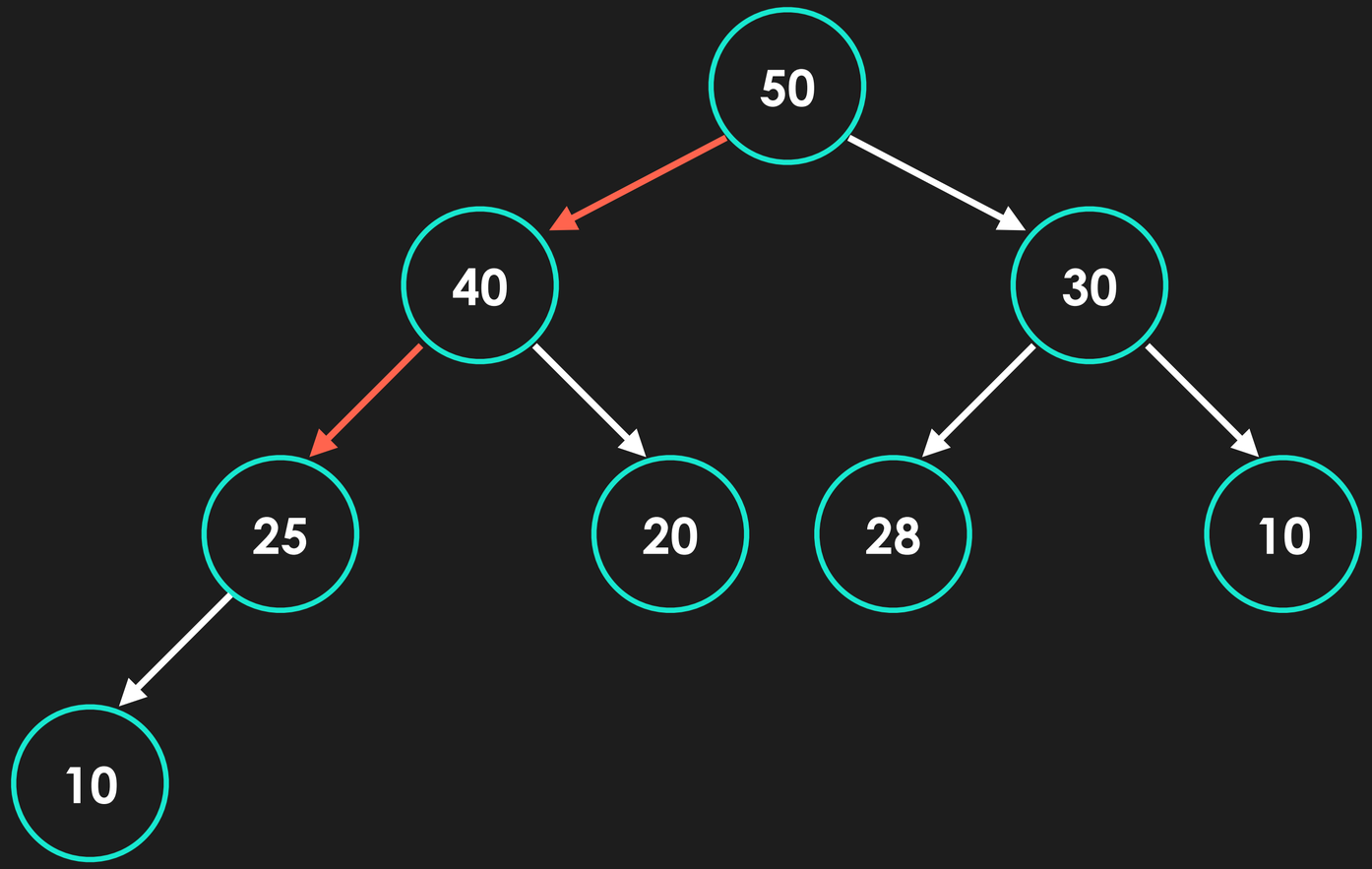
Tree Height

Maximum Tree Height



Maximum Tree Height

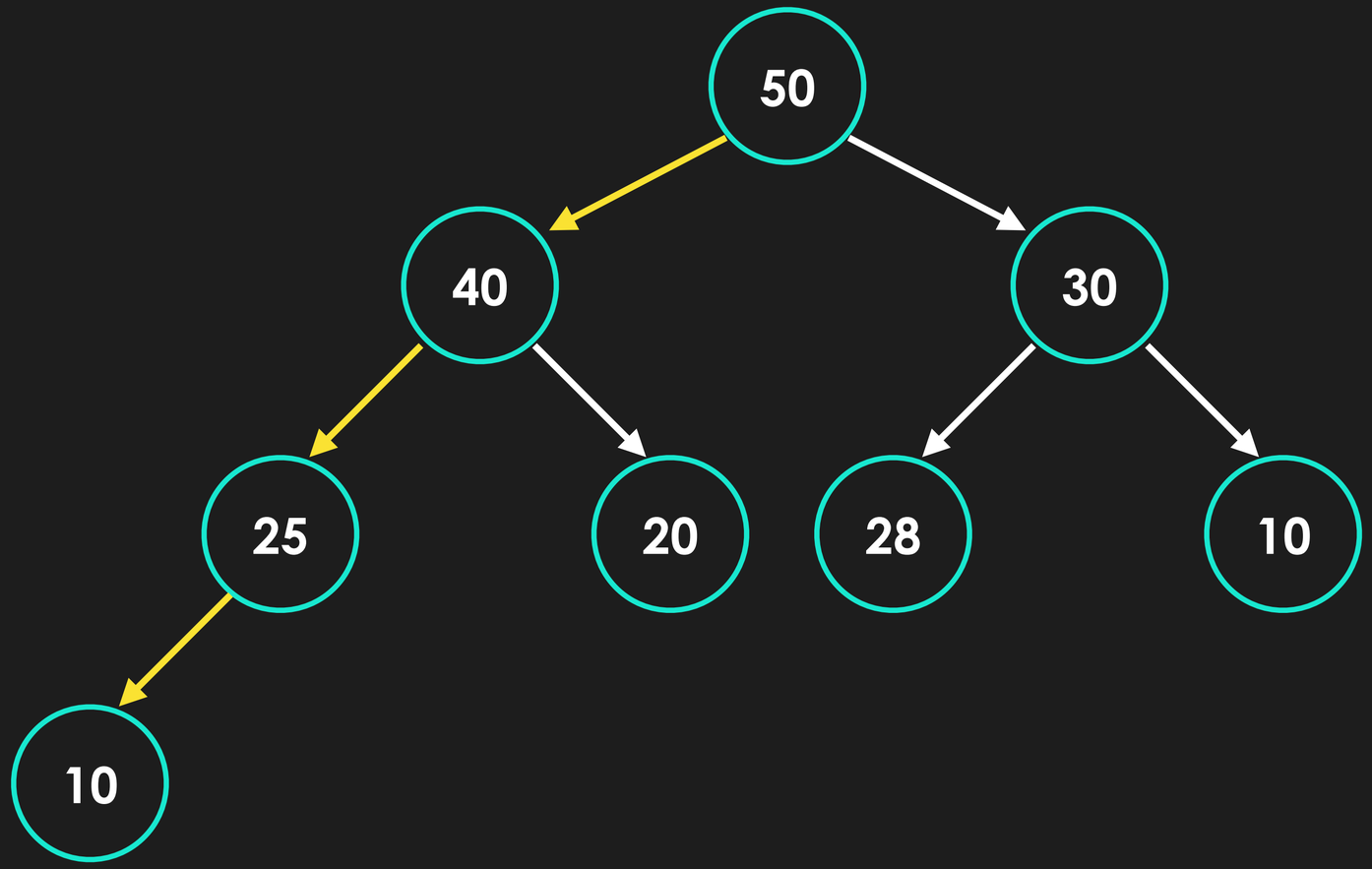
4 nodes: 2



Maximum Tree Height

4 nodes: 2

8 nodes: 3

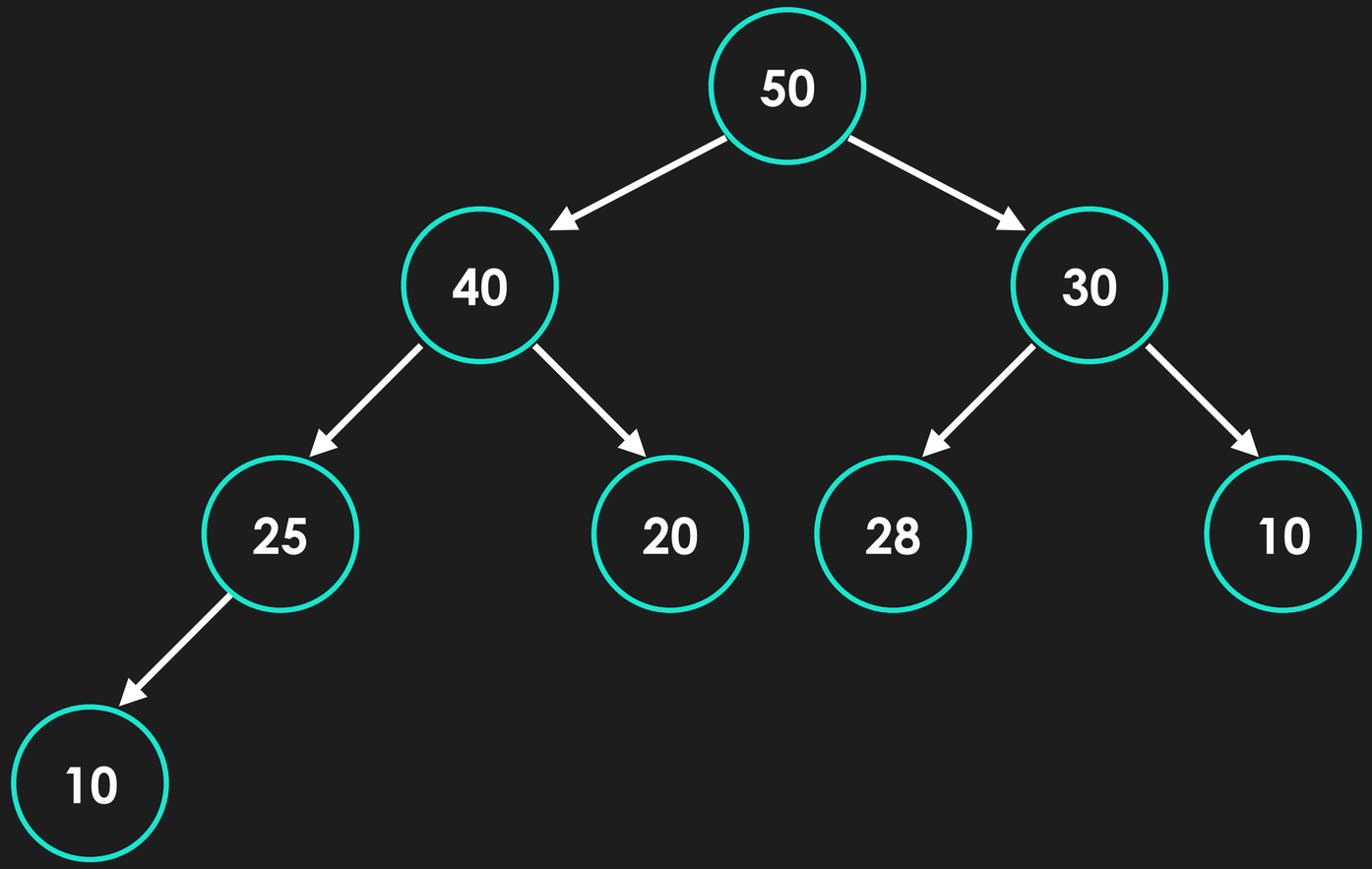


Maximum Tree Height

4 nodes: 2

8 nodes: 3

N nodes: logN (base 2)



Time complexity for sink & swim

- Both sink and swim have worst case time complexity of **O (logN)**
- This is because at most **logN** nodes are visited!

Time complexity for heap insertion

- Assuming no **resize**, adding an extra item will take **constant time**, and calling **swim** on the new item will take **logN**. Therefore, heap insertion has **logN** worst case time complexity!

Time complexity for heap extraction

- For extraction, swapping the first and last element takes **constant time**. Performing **sink** will take **logN** time. Therefore, heap extraction has worst case time complexity of **logN**

Comparison of Priority Queue Data Structures

	Worst Case		Average Case	
	Binary Heap	**List**	**Binary Heap**	**List**
Insert	logN	1	logN	1
Extract	logN	N	logN	N

Note: If your application will have a huge number of inserts but very few removals of highest priority, lists / arrays would be the way to go because you can insert in **constant time**

MinHeap

Imagine if now instead of tracking returns, we want to track risk, and therefore keep track of investments with the lowest risk

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

We can use a min heap, and to construct one, all we have to make is one small change:

```
def greater(self, index1, index2):
    if self.heap[index1].value > self.heap[index2].value:
        return True
    else:
        return False
```

We can use a min heap, and to construct one, all we have to make is one small change:

```
def greater(self, index1, index2):
    if self.heap[index1].value < self.heap[index2].value:
        return True
    else:
        return False
```

Our comparisons now treat lower values with higher priority!

```
pq = MaxHeap(10)
pq.insert(30, "Google")
pq.insert(20, "Amazon")
pq.insert(60, "Apple")
pq.insert(40, "Microsoft")
pq.insert(25, "Netflix")

while pq.size != 0:
    maxItem = pq.getMax()
    print(maxItem.key, maxItem.value)
```

```
pq = MaxHeap(10)
pq.insert(30, "Google")
pq.insert(20, "Amazon")
pq.insert(60, "Apple")
pq.insert(40, "Microsoft")
pq.insert(25, "Netflix")

while pq.size != 0:
    maxItem = pq.getMax()
    print(maxItem.key, maxItem.value)
```

```
Amazon 20
Netflix 25
Google 30
Microsoft 40
Apple 60
```

Applications of Priority Queues

Applications of Priority Queues

1. **Rank** (Get kth largest element in a list)
2. **Greedy Algorithms** (E.g. Dijkstra's Shortest Path)
3. **Huffman Compression**
4. **Stock Exchange Matching** (match buyers to sellers, highest bid)

Priority Queues with changing values

Priority Queues with changing values

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

Google	30%
Amazon	20%
Apple	60%
Microsoft	10%
Netflix	90%

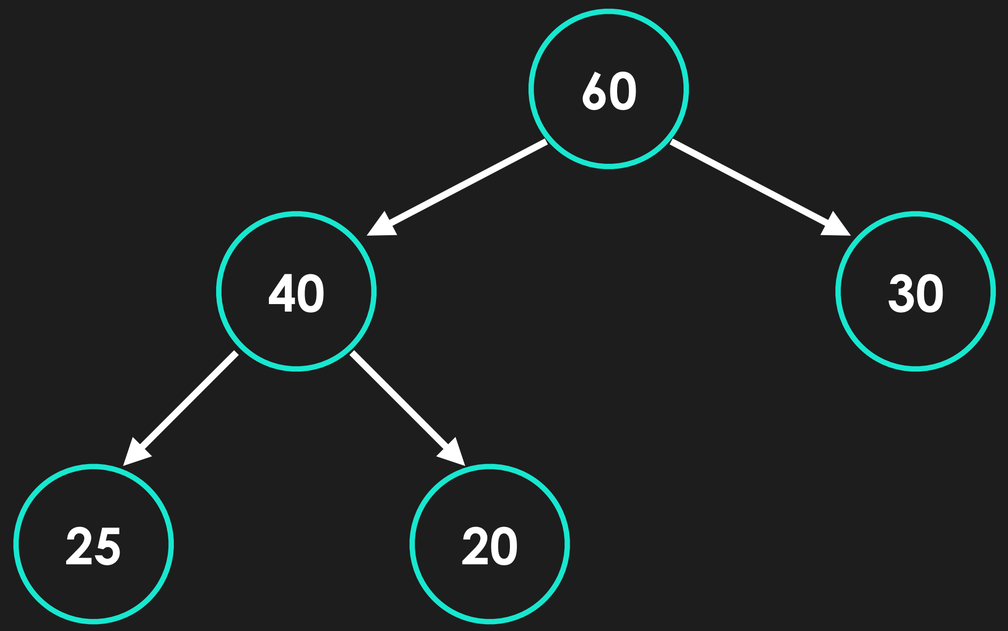
Priority Queues with changing values

Google	30%
Amazon	20%
Apple	60%
Microsoft	40%
Netflix	25%

Google	30%
Amazon	20%
Apple	60%
Microsoft	10%
Netflix	90%

If our keys' values change, how can we support this in a Heap structure?

We can create a dictionaries (positions), to store the index of each key in the heap



heap

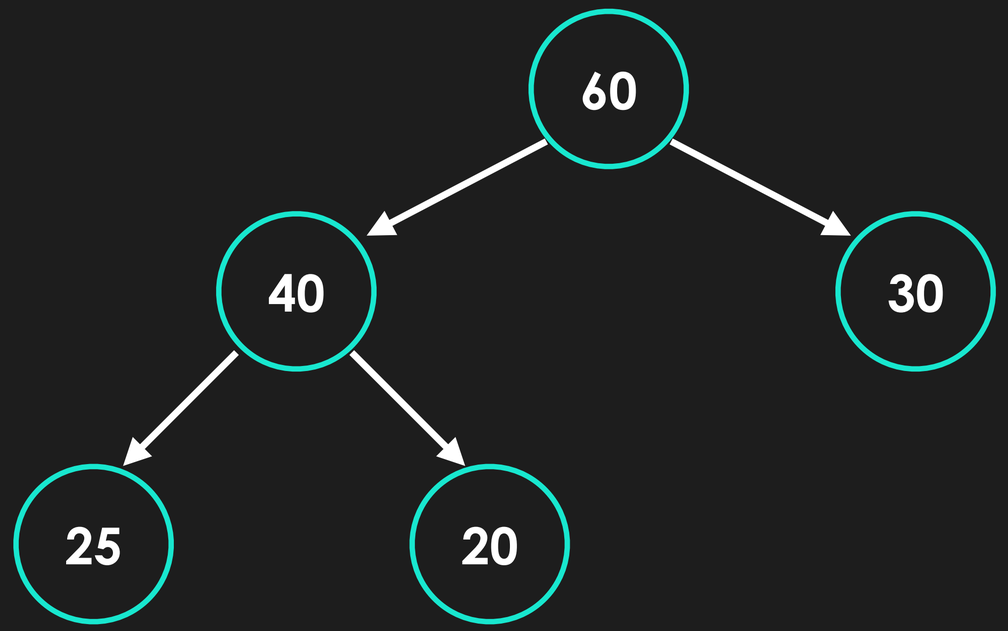
	60	40	30	20	25
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	5
"Amazon"	4

STEP 1: Check if key exists

pq.insert("Microsoft", 10)



heap

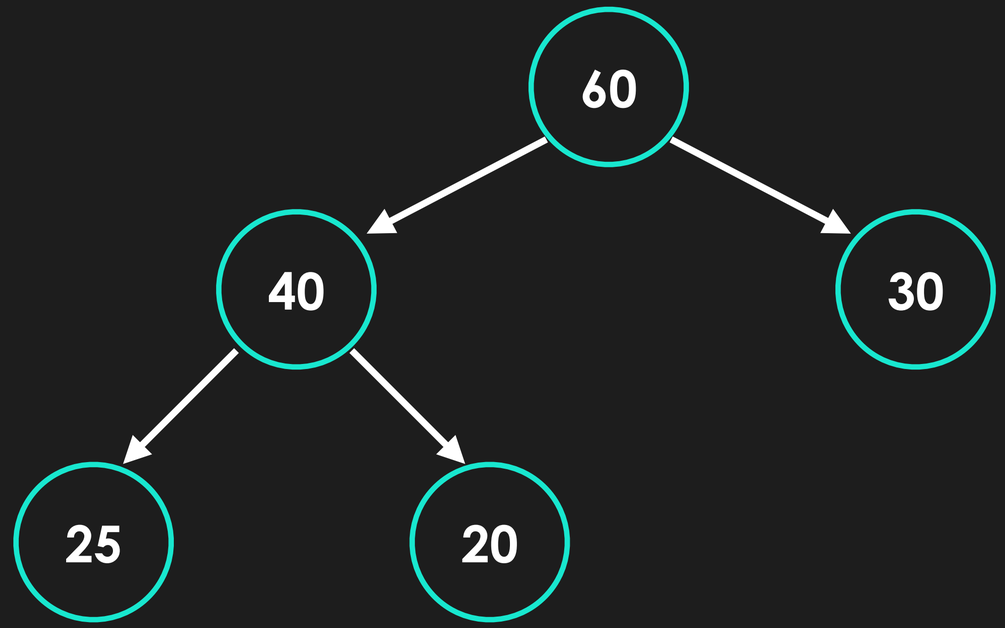
	60	40	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	5
"Amazon"	4

STEP 1: Check if key exists

pq.insert("Microsoft", 10)



heap

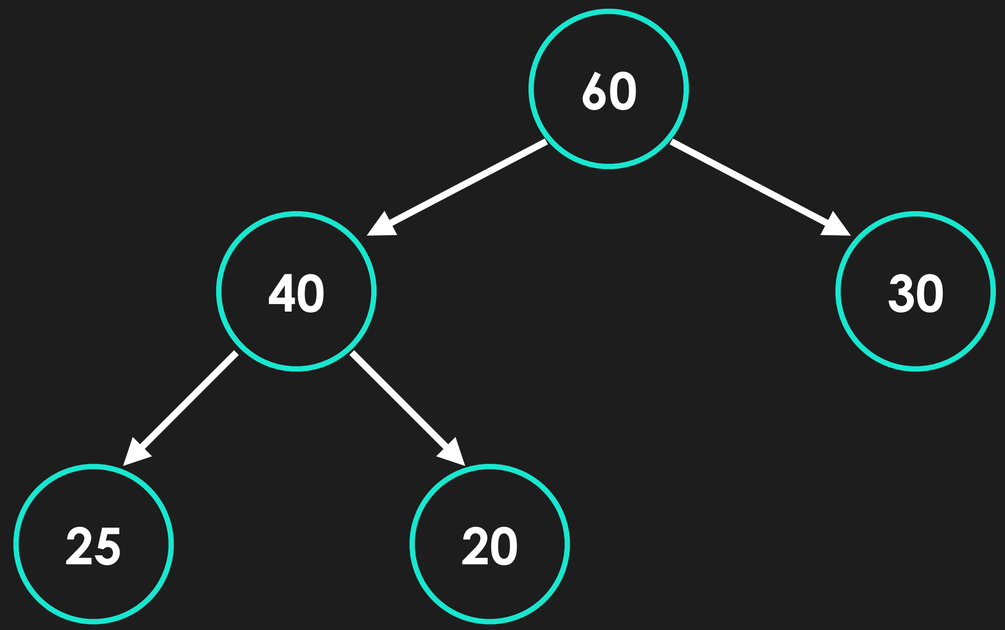
	60	40	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 2: Get index based on positions dict

pq.insert("Microsoft", 10)



heap

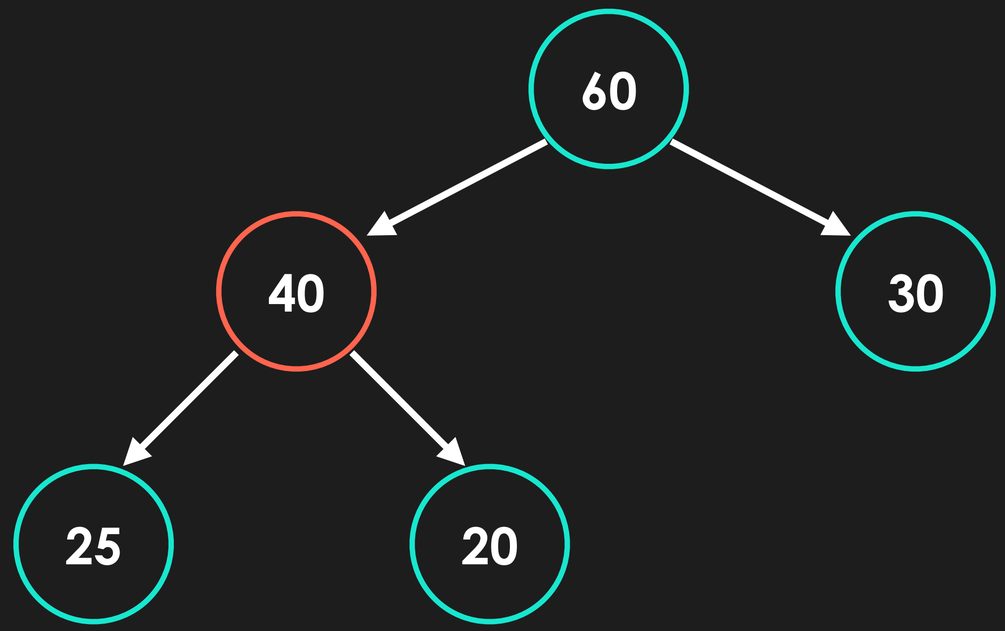
	60	40	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 2: Get index based on positions dict

pq.insert("Microsoft", 10)



heap

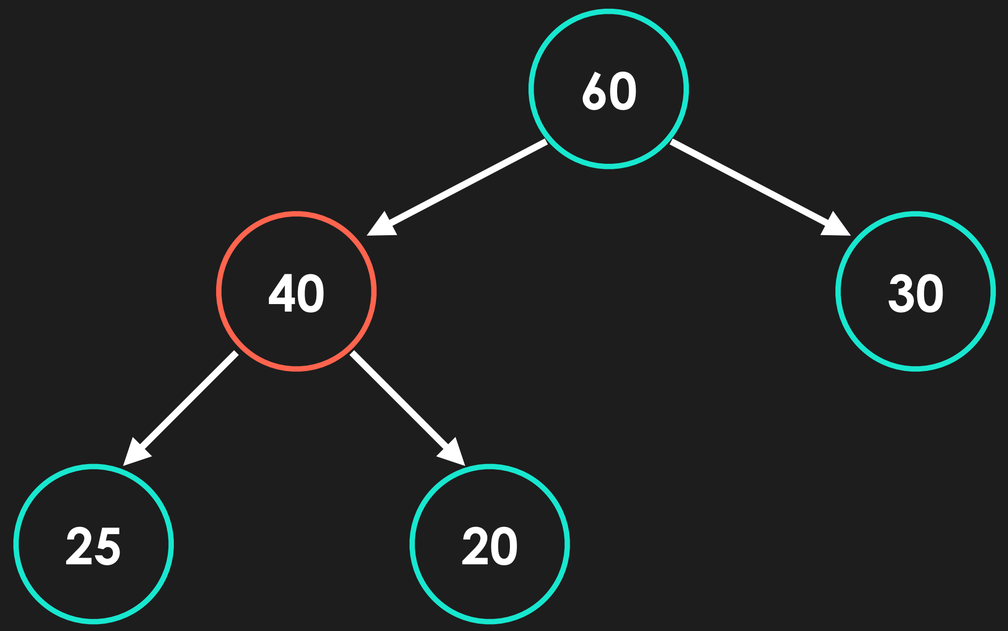
	60	40	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 3: Assign new value to item

pq.insert("Microsoft", 10)



heap

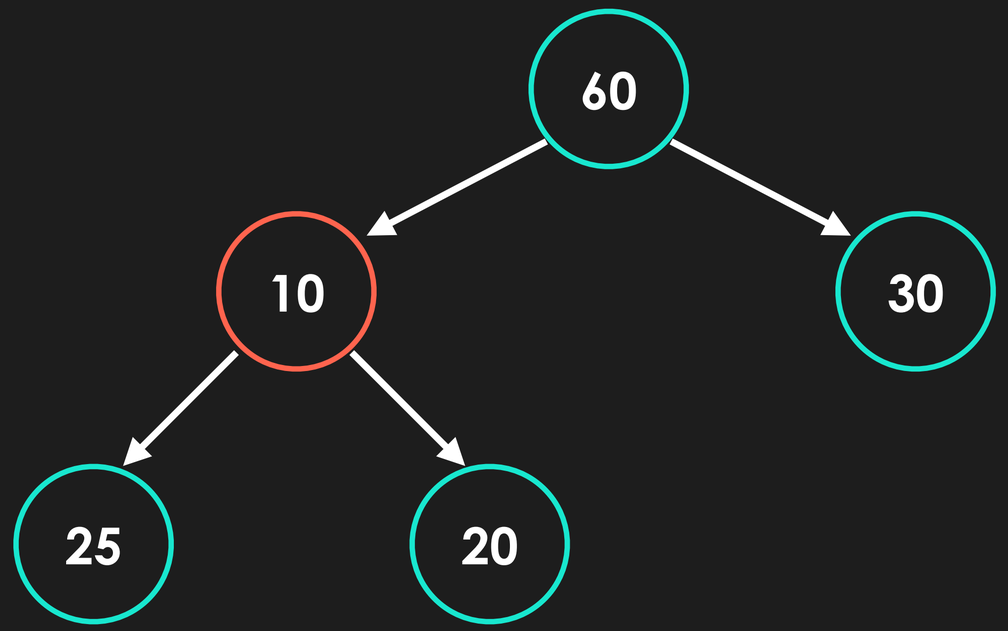
	60	40	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 3: Assign new value to item

pq.insert("Microsoft", 10)



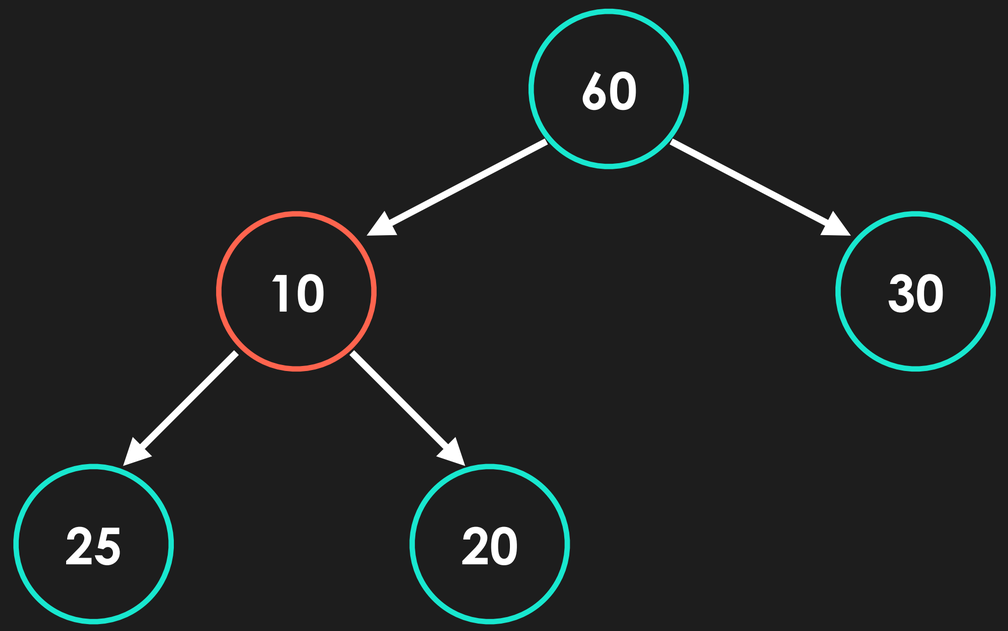
heap

	60	10	30	25	20
0	1	2	3	4	5

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 4: If parent value is lower priority, perform swim, else, perform sink



heap

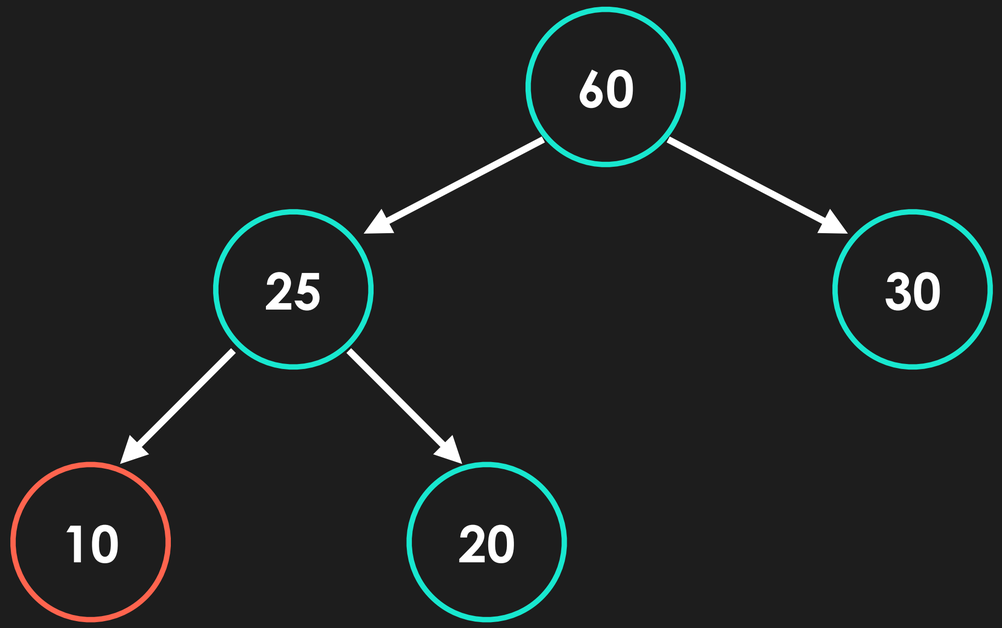
	60	10	30	25	20
0	1	2	3	4	5

pq.insert("Microsoft", 10)

positions

key	value
"Apple"	1
"Microsoft"	2
"Google"	3
"Netflix"	4
"Amazon"	5

STEP 4: If parent value is lower priority, perform swim, else, perform sink



heap

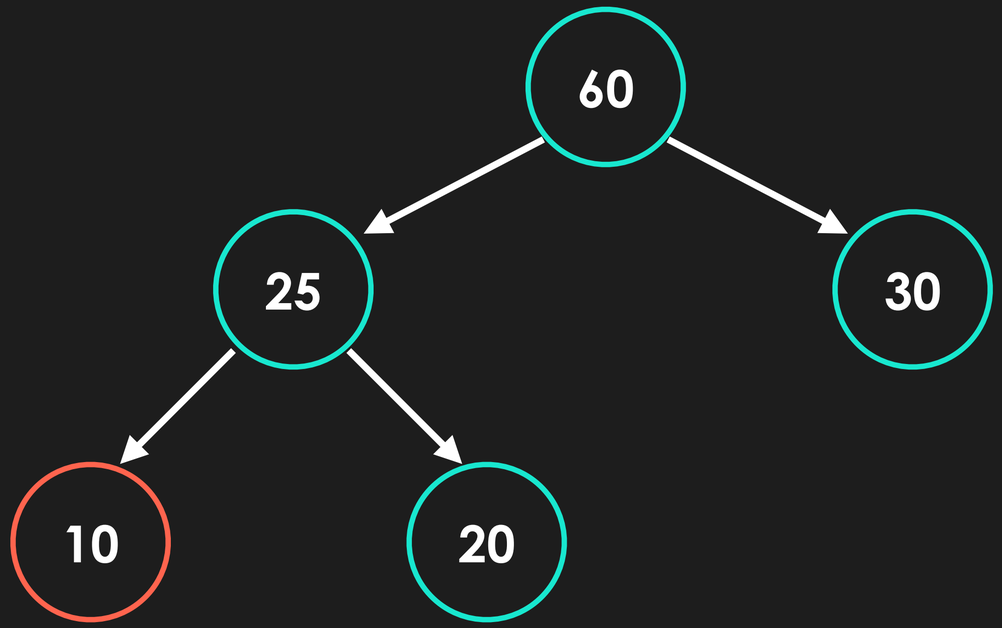
	60	25	30	10	20
0	1	2	3	4	5

pq.insert("Microsoft", 10)

positions

key	value
"Apple"	1
"Amazon"	2
"Google"	3
"Microsoft"	4
"Amazon"	5

Note: When swapping, you must now also swap in positions dictionary!



heap		60	25	30	10	20
	0	1	2	3	4	5

pq.insert("Microsoft", 10)

positions

key	value
"Apple"	1
"Amazon"	2
"Google"	3
"Microsoft"	4
"Amazon"	5

Implementation

constructor

```
def __init__ (self, maxsize):
    self.maxsize = maxsize
    self.size = 0
    self.heap = [None] * (maxsize + 1)
    self.positions = {}
```

swap

```
def swap (array, positions, index1, index2):
    key1 = array[index1].key
    key2 = array[index2].key
    positions[key1], positions[key2] = positions[key2], positions[key1]
    array[index1], array[index2] = array[index2], array[index1]
```

insert

```
def insert(self, newKey, newValue):
    if self.size >= self.maxsize:
        print('size limit reached')
        return

    # HANDLE EXISTING KEY, NEW VALUE
    if newKey in self.positions:

        currentIndex = self.positions[newKey]
        self.heap[currentIndex].value = newValue

        if self.greater(currentIndex, parent(currentIndex)):
            self.swim(currentIndex)
        else:
            self.sink(currentIndex)

        return


    self.size += 1
    self.heap[self.size] = HeapItem(newKey, newValue)
    self.positions[newKey] = self.size

    self.swim(self.size)
```

When would we need changing values?

When would we need changing values?

In Dijkstra's algorithm, if we have a heap that handles changing values, we can achieve a faster time complexity for the algorithm!

Lab Session 1

- Implement **min_heap2.py**
- To test, run `**python utils/mh2_test.py**`