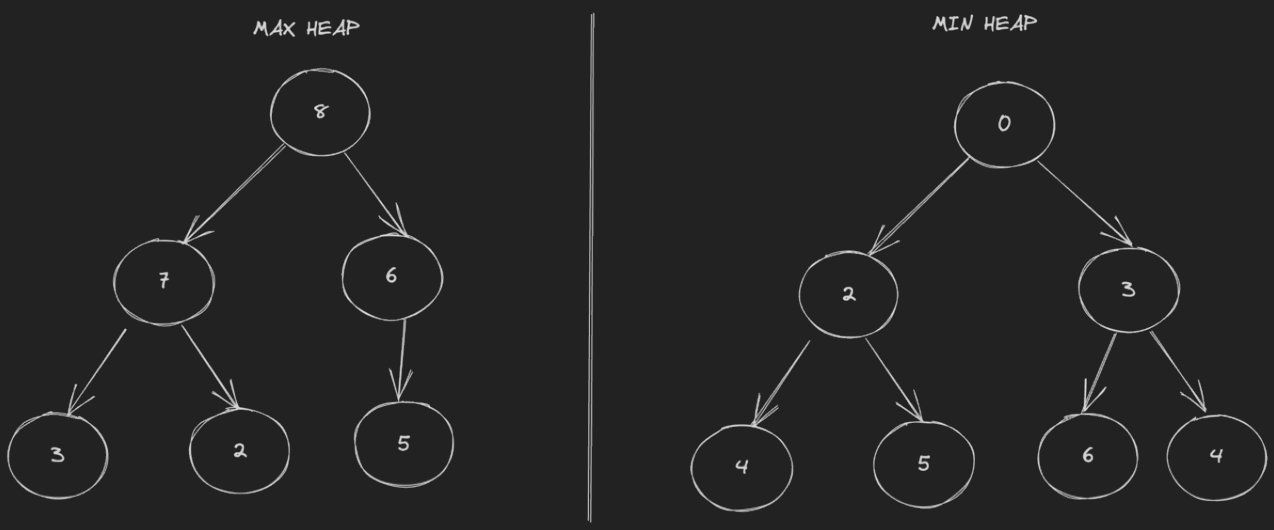
Operation	**Binary Heap (PQ)**
Adding	O(log n)
Peeking	O(1)
Remove	O(log n)
Binary Heap Construction	O(n)

Naive removing (Not the root)	O(n)
Advanced Removing from help using a hashtable	O(log(n))
Naive contains (Not the root)	O(n)
Contains check with help of a hash table	O(1)

Definition

A heap is a binary **tree-based** data structure that satisfies the **heap invariant (property).** Used to create Priority Queues



Heap Invariant

> "If A is a parent node of B then A is ordered with respect to B for all nodes A, B in the heap"

The heap property states that for every node in the heap, the value of that node is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the values of its children.

Max Heap

In a max heap, the value of each parent node is greater than or equal to the values of its children. The maximum value in the heap is stored at the root node.

Min Heap

In a min heap, the value of each parent node is less than or equal to the values of its children. The minimum value in the heap is stored at the root node. This is usually the heap most languages use and implement

Converting to a Max Heap

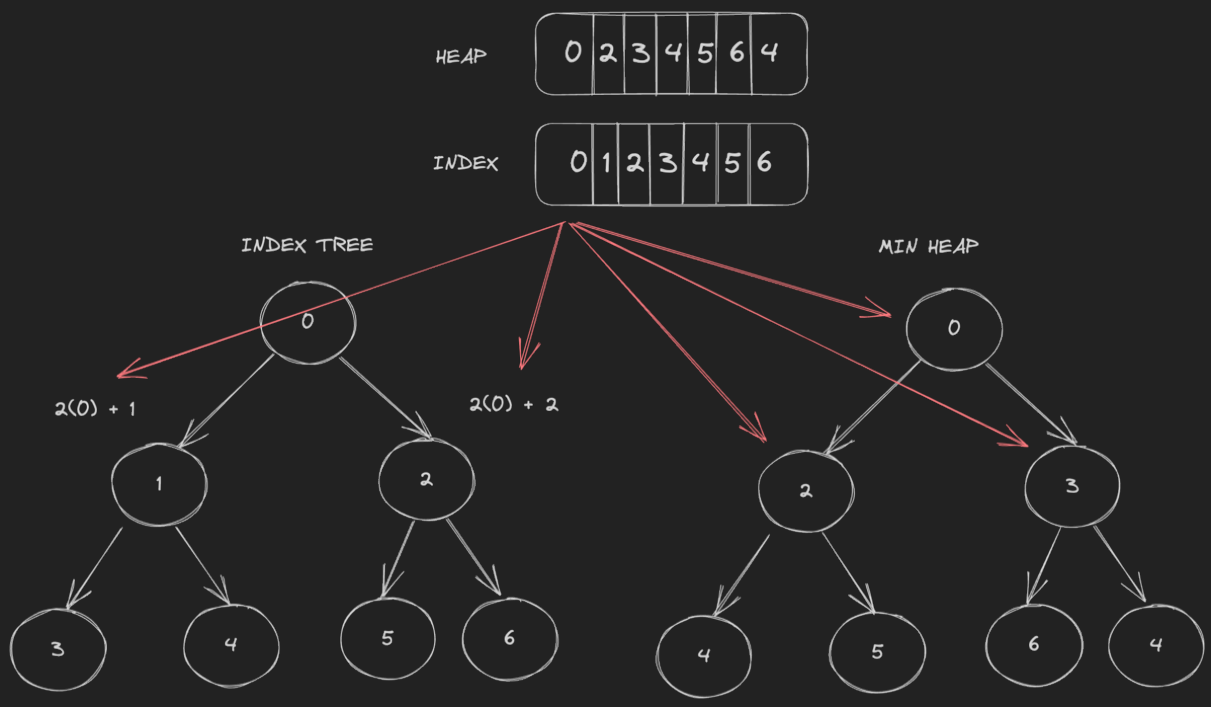
As most languages implement heaps as min heaps, if we want a max heap we can simply convert the values to negatives for integer/float values

```python
def convert_to_max_heap(arr: List[int]) -> List[int]:
    # Multiply each element by -1
    max_heap = [-1 * x for x in arr]
    return max_heap
```

Representation of Heaps

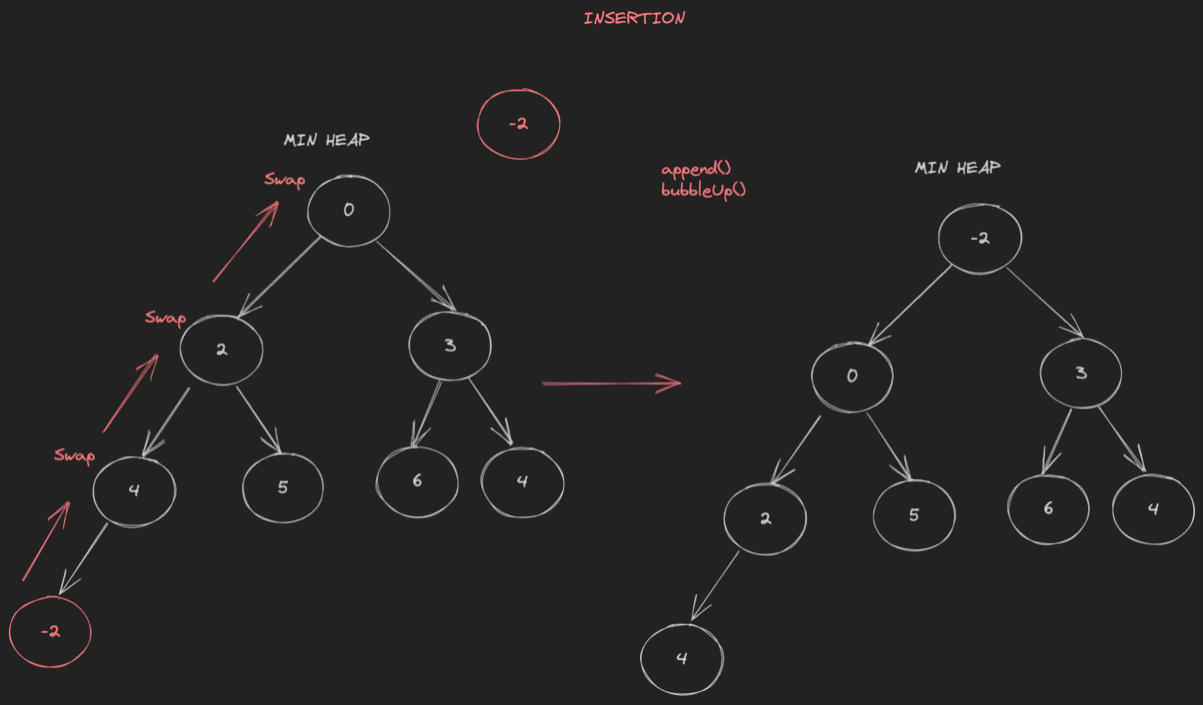
Can be represented as an array

> Access a parent i | Left child of that parent $2i + 1$ | Right child of that parent $2i + 2$



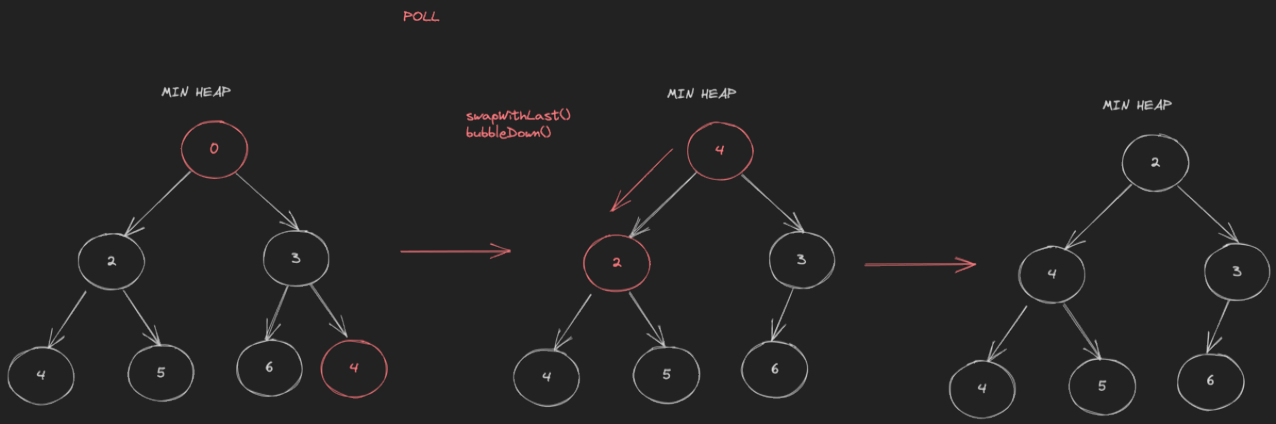
Insertion

Adding a new element to the heap. The element is placed at the next available position according to the heap structure, generally in a complete binary tree fashion—where every level is filled before going to the next, and then it is recursively moved up or down to maintain the heap property.



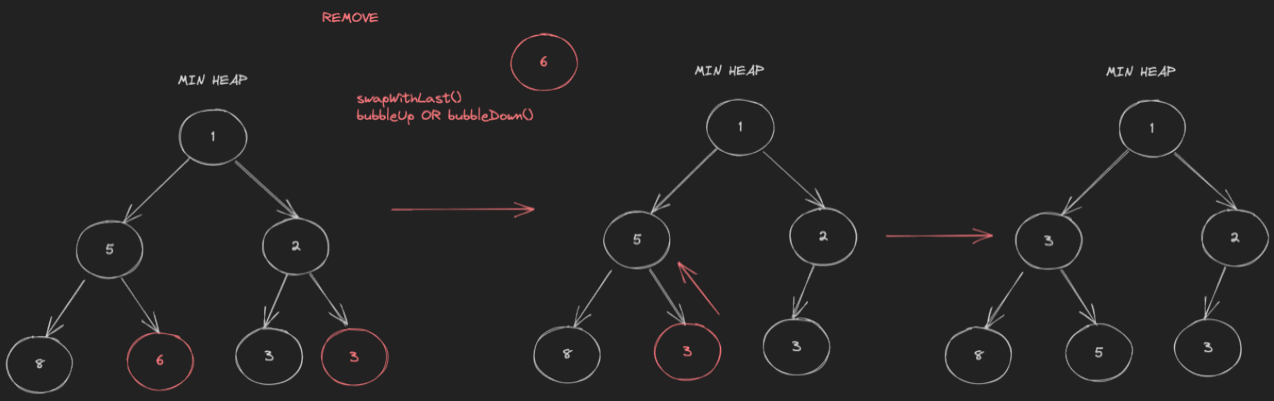
Polling

Removing the root element from the heap. The last element in the heap is moved to the root position, and then it is recursively moved down to its appropriate position to maintain the heap property.



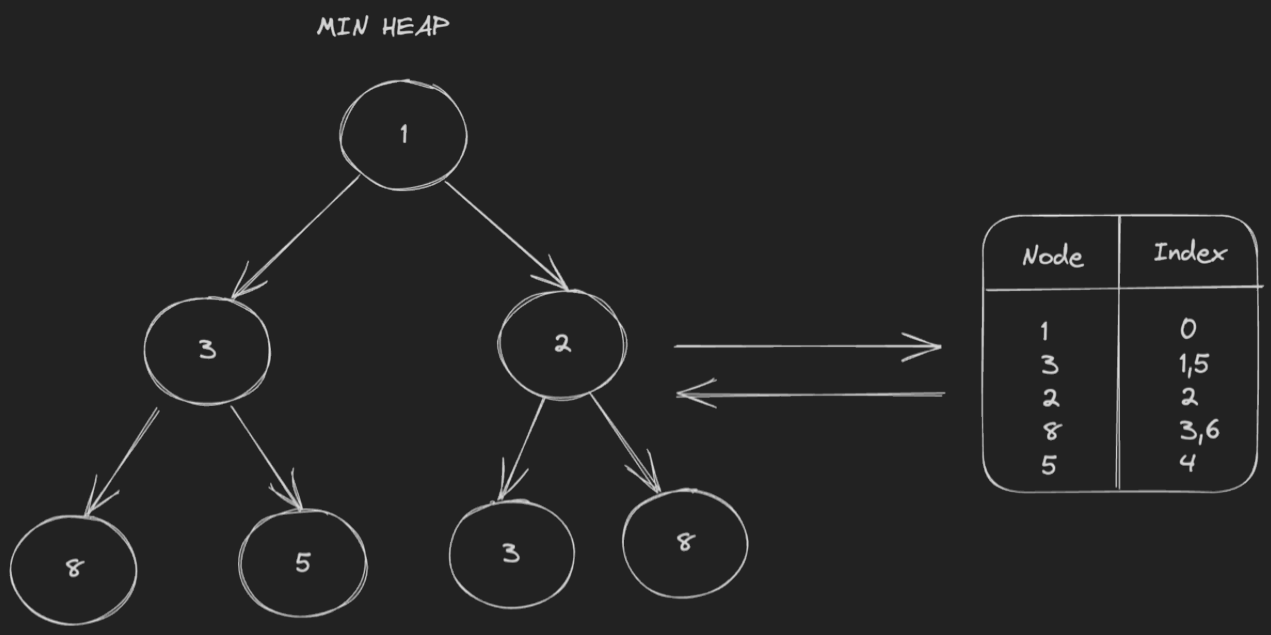
Remove

Removing a non-root element from the heap. Linear search to find the node, then last element in the heap is moved to that position, and then it is recursively moved up or down to maintain the heap property.



Hashtables

Stores the heap into a hashable, it is a key-set pair where the key is the value of the node and the set/list is all the indexes that share the same value, this makes it really efficient to find and remove or determine if a heap contains a value since you no longer have to do a linear search.

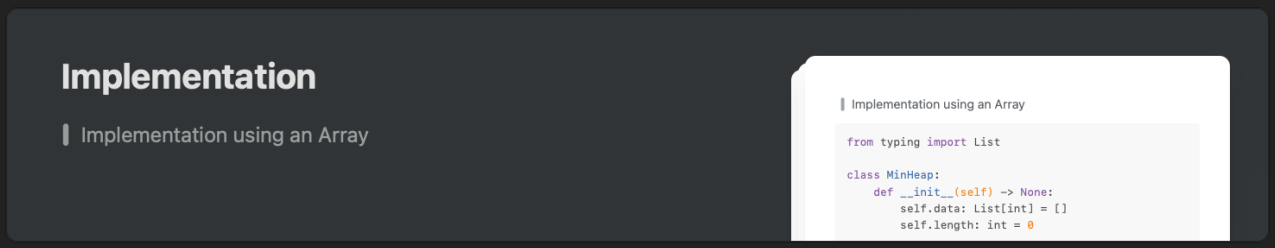


Peek

Retrieving the value of the root element without removing it from the heap.

Use Cases

- Used to implement Dijkstra's shortest path algorithm
- Anytime you need to dynamically fetch the ‘next best or worse’ element
- Used in Huffman coding, (used for lossless data compression)
- Best First Search (BFS) algorithms such as A*
- Used by Minimum Spanning Tree (MST) algorithms
- Priority queues
- Heapsort, a sorting algorithm using heaps



↑ Heap

Implementation

Implementation using an Array

```python
from typing import List

class MinHeap:
    def __init__(self) -> None:
        self.data: List[int] = []
        self.length: int = 0

    def insert(self, value: int) -> None:
        self.data.append(value)
        self.heapify_up(self.length)
        self.length += 1

    def delete(self) -> int:
        if self.length == 0:
            return -1

        out = self.data[0]
        self.length -= 1

        if self.length == 0:
            self.data = []
            return out

        self.data[0] = self.data[self.length]
        self.heapify_down(0)

        return out

    def heapify_down(self, idx: int) -> None:
        left_idx = self.left_child(idx)
        right_idx = self.right_child(idx)

        if idx >= self.length or left_idx >= self.length:
            return

        left_value = self.data[left_idx]
        right_value = self.data[right_idx]
        value = self.data[idx]

        if left_value < right_value and value > left_value:
            self.data[idx] = left_value
            self.data[left_idx] = value
            self.heapify_down(left_idx)
        elif right_value < left_value and value > right_value:
            self.data[idx] = right_value
            self.data[right_idx] = value
            self.heapify_down(right_idx)

    def heapify_up(self, idx: int) -> None:
        if idx == 0:
            return

        parent = self.parent(idx)
        parent_value = self.data[parent]
        value = self.data[idx]

        if parent_value > value:
            self.data[idx] = parent_value
            self.data[parent] = value
            self.heapify_up(parent)
```