# Exercise 9.1

The left child of the element at index ii is located at index $2i + 1$.

The right child of the element at index ii is located at index $2i + 2$.

The parent of the element at index ii is located at index $(i - 1) / 2$

Here are all possible arrays that represent a binary max-heap with the keys 1, 2, 3, 4, 5:

**Array 1: [5, 4, 3, 1, 2]**

```
    5
   / \
  4   3
 / \
1   2
```

**Array 2: [5, 4, 3, 2, 1]**

```
    5
   / \
  4   3
 / \
2   1
```

**Array 3: [5, 3, 4, 1, 2]**

**Array 4: [5, 3, 4, 2, 1]**

# Exercise 9.2

**Algorithm for Extract-Max:**

1- **Identify the Largest and Second Largest Elements:**

- The largest element in the max-heap (root) is Q[0].
- The second largest element must be one of the children of the root, i.e., Q[1] or Q[2] (if both exist).

2- **Remove the Second Largest Element:**

- Compare the values of Q[1] and Q[2] (if both exist) and identify the second largest element.
- Swap this second largest element with the last element in the heap.
- Reduce the heap size by one (effectively removing the last element which now holds the second largest value).
- Restore the max-heap property by heapifying down the swapped element.

# Exercise 9.3

In a binary max-heap, each parent node is greater than or equal to its child nodes. The largest element is always at the root (index 0), and the second-largest element can only be one of its child nodes (indices 1 or 2).

For the third-largest element, it must be one of the children of the second-largest element. So we need to consider all possible scenarios:

1. If the second-largest element is at index 1, its children are at indices 3 and 4.
2. If the second-largest element is at index 2, its children are at indices 5 and 6.
3. If the second-largest element is at index 1, the third-largest element could also be 2.
4. If the second-largest element is at index 2, the third-largest element could also be 1.

Therefore, the third-largest element could be at any of the following indices: 1, 2, 3, 4, 5, 6.

# Exercise 9.4

Let's define the hash function h(k) as follows:

h(k)= 1

And for the type of probing, we'll use **linear probing**. In linear probing, if a collision occurs, we linearly probe the next bucket by incrementing the index by 1 until we find an empty bucket.

**1-** Insert 1:

h(1) = 1

It goes to bucket 1 directly.

**2-** Insert 3:

h(3) = 1

It goes to bucket 1 but it’s full, so it goes to 2

**3-** Insert 3:

h(3) = 1

It goes to bucket 1 but it’s full, so it goes to 2, but it’s full, so it goes to 3.…

…

# Exercise 9.5

Once we remove the maximum element, we need to traverse the entire list to find the new maximum, which means **ExtractMax** is O(n) in the worst case. This traversal offsets the claimed O(1) time complexity for **ExtractMax** after deletion.

In summary:

- **Insert**: O(1)
- **Max**: O(1)
- **ExtractMax**: O(n) due to the need to find the new max after deletion

# Exercise 9.6

- **Combine Arrays**:

Merge the two arrays A and B to create a new array C that contains all the elements of A and B. This step takes O(n + m) time, where n is the size of A and m is the size of B.

- **Build a Max-Heap**:

Use the array C to build a new max-heap. This can be done in O(k) time, where k is the size of C. The build-heap operation involves heapifying elements from the bottom up.

- Complexity of building max-heap:

$$\sum_{h=0}^{\log(n)} 2^h \cdot (\log(n) - h) = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots = \mathrm{n}$$

$$a + ar + ar^2 + ar^3 + \cdots = \frac{a}{1-r}$$