Hashing Data Structure

Hash Table

The Hash table data structure **stores elements in key-value pairs where**

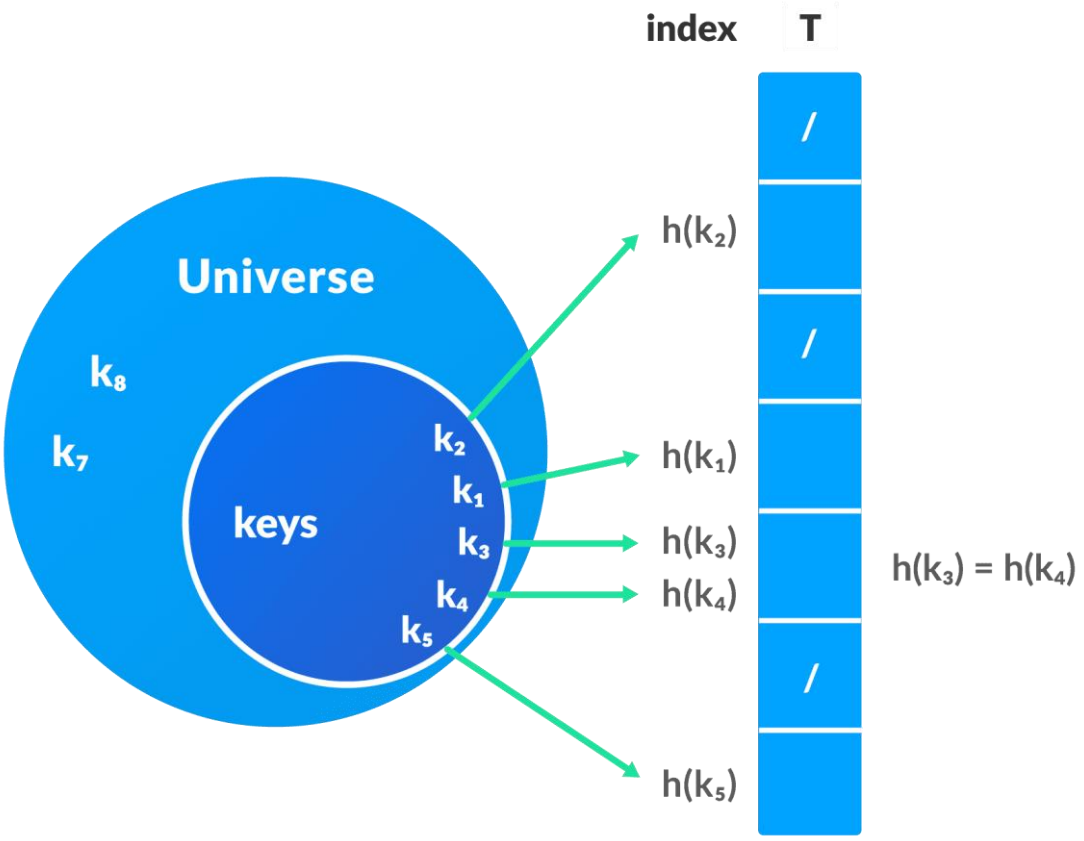
- **Key- unique integer that is used for indexing the values**
- **Value** - data that are associated with keys.

Key and Value in Hash table

Hashing (Hash Function)

In a hash table, **a new index is processed using the keys.** And, **the element corresponding to that key is stored in the index.** This process is called **hashing**.

Let **k** **be a key and** **h(x)** **be a hash function**. Here, **h(k)** will give us a new index to store the element linked with k.



Hash table Representation

To learn more, visit Hashing.

Hash Collision

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.**

We can resolve the hash collision using one of the following techniques.

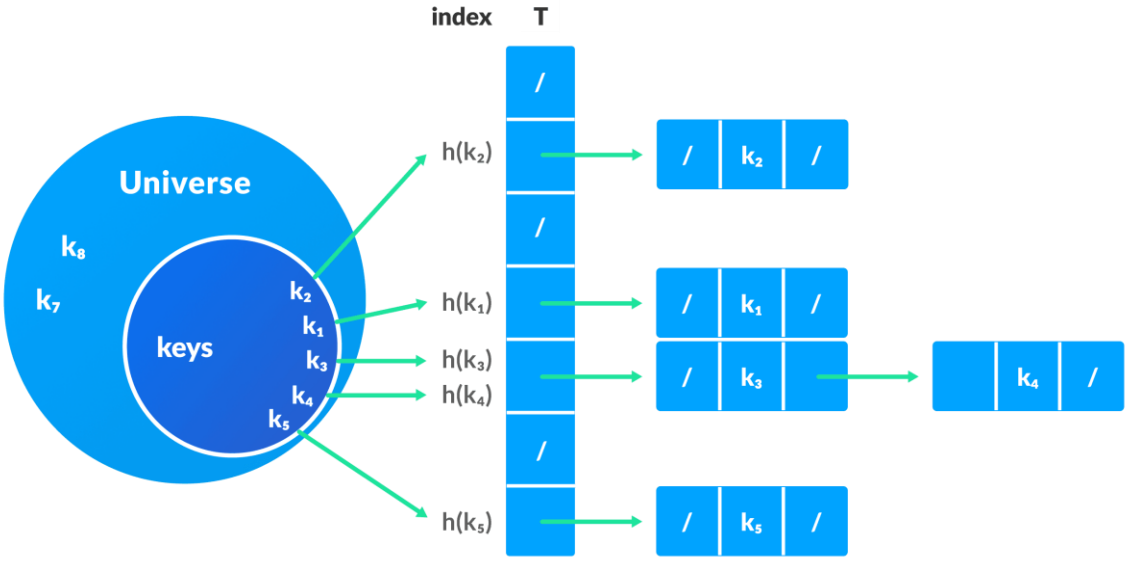
- Collision resolution by **chaining**
- **Open Addressing:** Linear/Quadratic Probing and Double Hashing

---

1. Collision resolution by chaining

In **chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.**

If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j contains NIL.



Collision Resolution using chaining

2. Open Addressing

Unlike chaining, **open addressing doesn't store multiple elements into the same slot.** Here, each slot is either **filled with a single key or left NIL.**

Different techniques used in open addressing are:

i. Linear Probing

In linear probing, **collision is resolved by checking the next slot.**

h (k, i) = (h′(k) + i) mod m
where,

- i = {0, 1, ...}
- h'(k) is a new hash function

If a collision occurs at h(k, 0), **then h(k, 1) is checked.** In this way, the value of i is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

ii. Quadratic Probing

It works similar to linear probing **but the spacing between the slots is increased (greater than one)** by using the following relation.

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
where,

- c_1 and c_2 are positive auxiliary constants,
- $i = \{0, 1, ...\}$

iii. Double hashing

If a collision occurs after applying a hash function $h(k)$, **then another hash function is calculated for finding the next slot.**
$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

Good Hash Functions

A good hash function **may not prevent the collisions completely however it can reduce the number of collisions.**

Here, we will look into different methods to find a good hash function

1. Division Method

If **k is a key and m is the size of the hash table**, the hash function h() is **calculated as:**

h(k) = k mod m

For example, **If the size of a hash table is 10 and k = 112 then h(k) = 112 mod 10 = 2.** The value of m must not be the powers of 2. This is because the powers of 2 in binary format are 10, 100, 1000, ….. When we find k mod m, we will always get the lower order p-bits.

2. Multiplication Method

3. Universal Hashing

In Universal hashing, the hash function is chosen at random independent of keys.

Double hashing is a collision resolution technique used in hash tables. It works **by using two hash functions to compute two different hash values for a given key.** The first hash function is used to **compute the initial hash value**, and the **second hash function is used to compute** the step size for the probing sequence.

Rehashing:

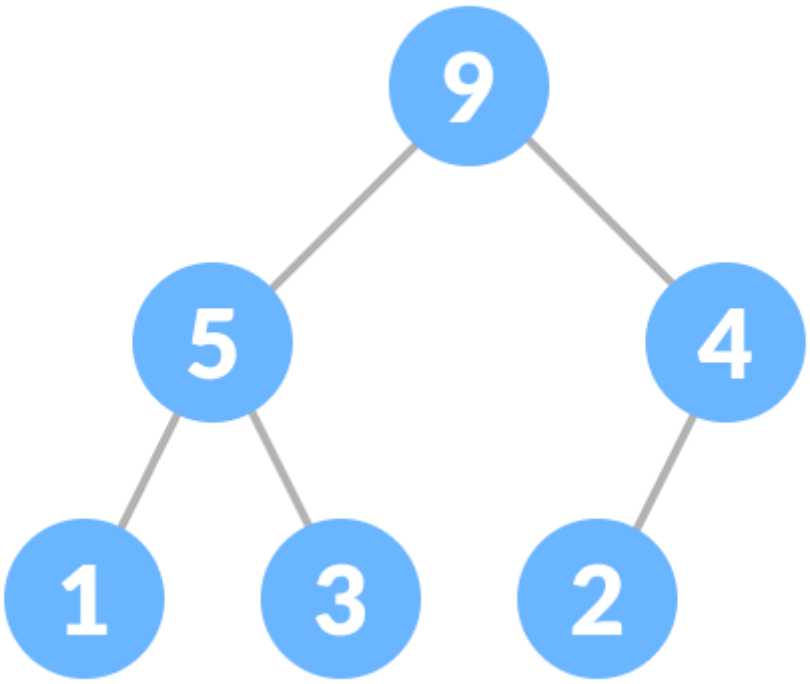
Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

When a hashmap becomes full, **the load factor (i.e., the ratio of the number of elements to the number of buckets) increases.** As the load factor increases, the number of collisions also increases, which can lead to poor performance. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions.

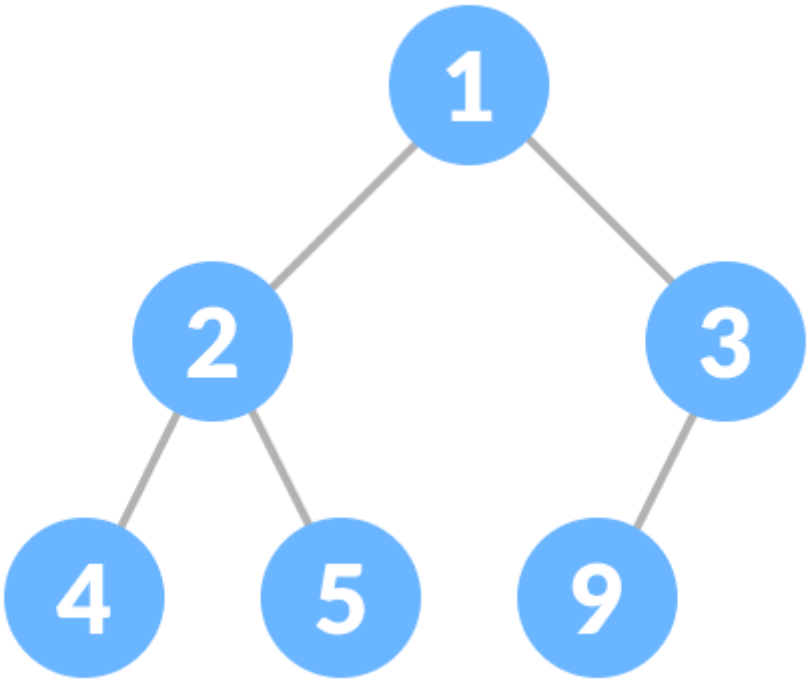
Heap Data Structure

Heap data structure is a complete binary tree that satisfies **the heap property**, where any given node is

- **always greater than its child node/s and the key of the root node is the largest among all other nodes.** This property is also called **max heap property**.
- **always smaller than the child node/s and the key of the root node is the smallest among all other nodes.** This property is also called **min heap property**.



Max Heap



Min-heap

This type of data structure is also called a **binary heap**.

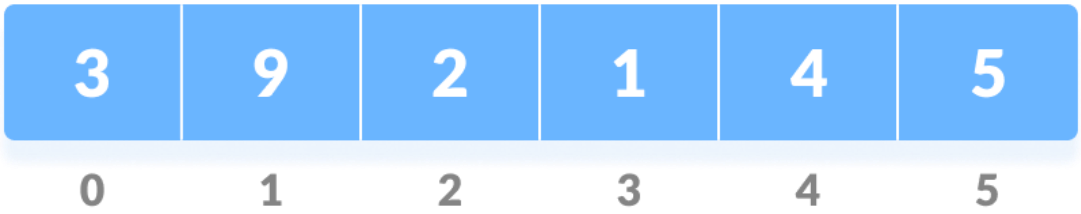
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

Heapify

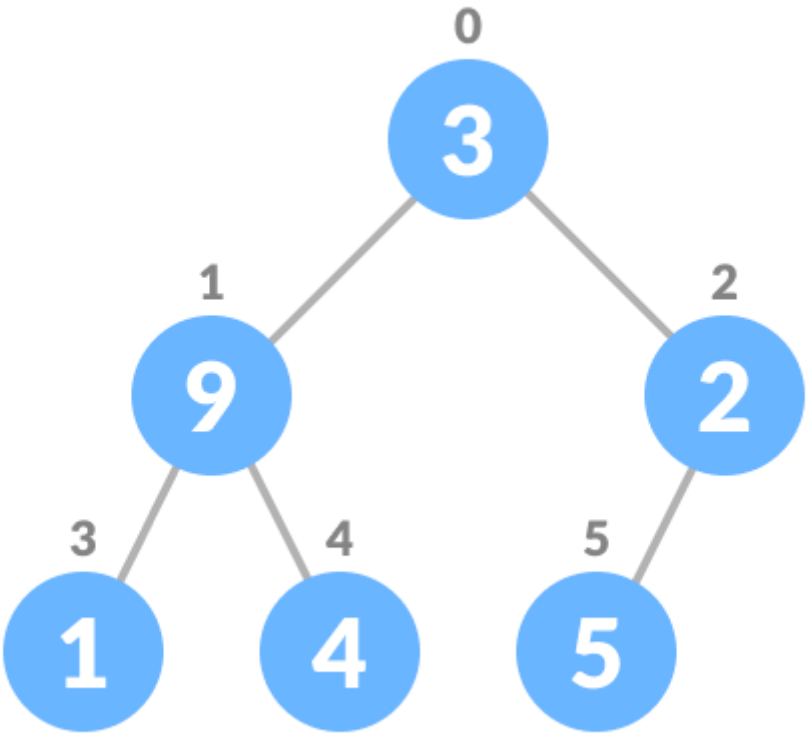
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

1. **Let the input array be**



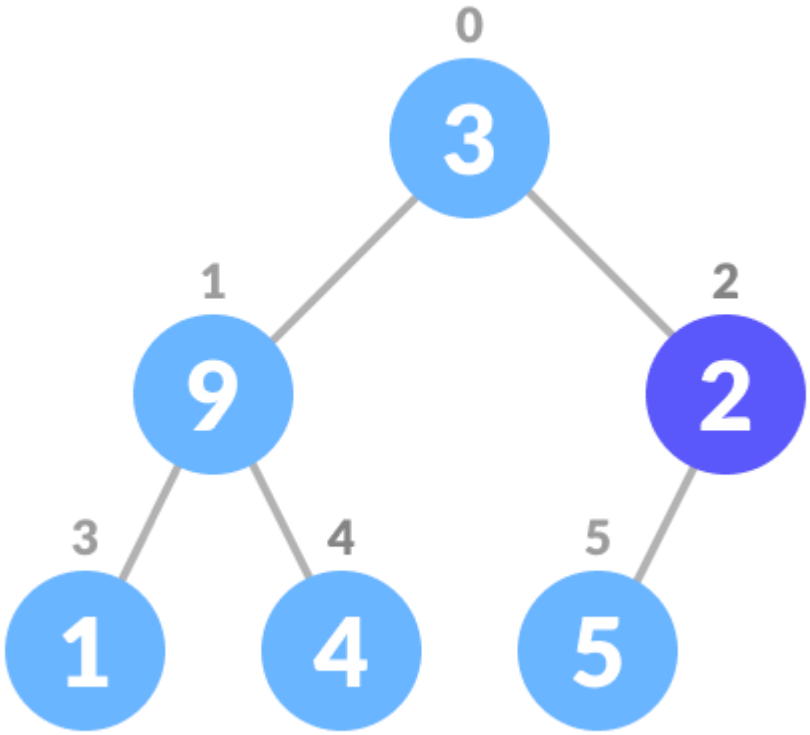
Initial Array

2. Create a complete binary tree from the array



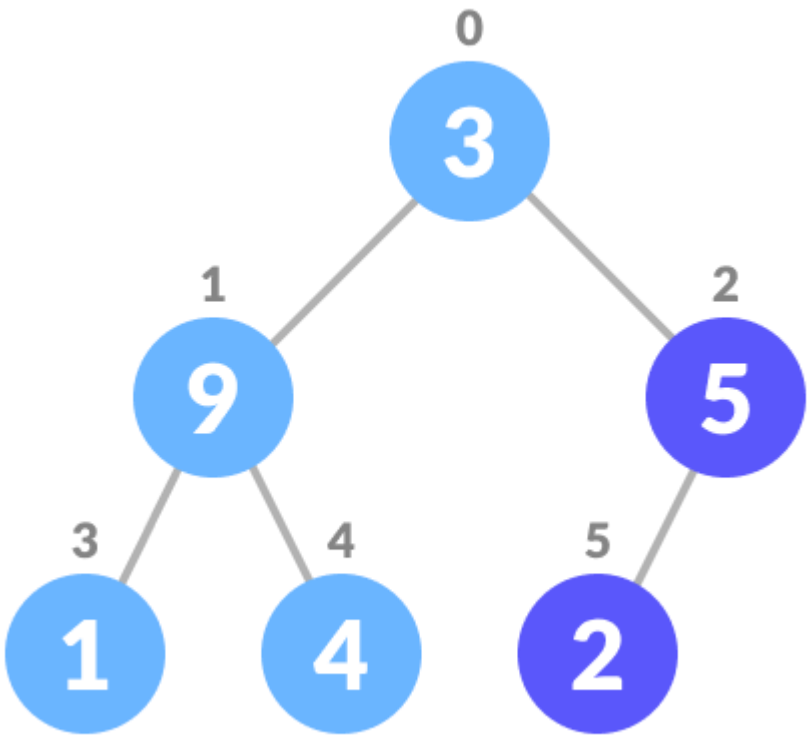
Complete binary tree

3. **Start from the first index of non-leaf node whose index is given by n/2 - 1.**



4. Set current element **i** as largest.
5. The **index of left child is given by 2i + 1** and the **right child is given by 2i + 2.**
   If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.
   If rightChild is greater than element in largest, set rightChildIndex as largest.

6. **Swap `largest` with `currentElement`**



Swap if necessary

7. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest

  swap array[i] and array[largest]
```

To create a Max-Heap:

```
MaxHeap(array, size)
  loop from the first index of non-leaf node down to zero
    call heapify
```

For Min-Heap,
both `leftChild` and `rightChild` must be larger than the parent for all nodes.

Insert Element into Heap

Algorithm for insertion in Max Heap

```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to
right.)

heapify the array
```

Delete Element from Heap

Algorithm for deletion in Max Heap

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```