# Module 4: Sorting and Searching

---

**Syllabus**

**Sorting Techniques: - Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort.**

**Searching Techniques: - Linear Search, Binary Search, Hashing – Hashing functions: Mid square, Division, Folding, Digit Analysis; Collision Resolution: Linear probing, Quadratic Probing, Double hashing, Open hashing.**

---

## Sorting

Sorting algorithms are fundamental in computer science for arranging data in a particular order, typically either ascending or descending. Ascending order means arranging elements from the smallest to the largest value, while descending order involves arranging elements from the largest to the smallest value. These algorithms can be classified into two main categories:

- Internal Sorting:
  - Used when the elements to be sorted are in main memory.
  - Suitable for sorting small lists of elements that can fit entirely into main memory.
  - Examples of internal sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort.
- External Sorting:
  - Used when the elements to be sorted are in secondary memory, such as disks or tapes.
  - Necessary for sorting large files of records that cannot fit into main memory entirely.
  - External sorting algorithms are designed to efficiently manage the data transfer between main memory and secondary storage.
  - The efficiency of external sorting depends on factors such as the type of storage device (e.g., disks, tapes) and the number of devices that can be used simultaneously.

External sorting algorithms are typically more complex than internal sorting algorithms due to the additional considerations of managing data across multiple storage devices and minimizing input/output operations. Examples of external sorting algorithms include External Merge Sort and Polyphase Sort. These algorithms are essential for handling large datasets efficiently when the entire dataset cannot be accommodated in main memory.

**Internal sorting:**

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Merge sort
5. Quick sort
6. Heap sort

**Performance Of Sorting Algorithm**

Sorting algorithms can be evaluated based on two key factors:

1. Space Complexity:
   - Space complexity refers to the amount of memory required by an algorithm to perform its operation.
   - It quantifies the amount of memory consumed by the algorithm as a function of the input size.
   - Some sorting algorithms require additional memory for temporary storage or for auxiliary data structures.
   - Space complexity is typically denoted using Big O notation, similar to time complexity.
   - For example, an algorithm with space complexity O(n) indicates that the amount of memory required is proportional to the number of elements in the input list.
2. Time Complexity:
   - Time complexity measures the computational time required by an algorithm to execute, typically in terms of the number of basic operations performed.
   - It quantifies the efficiency of the algorithm in terms of its runtime performance.
   - Sorting algorithms can have different time complexities depending on their implementation and efficiency.
   - Time complexity is also expressed using Big O notation, which describes the upper bound on the growth rate of the algorithm's runtime as the input size increases.
   - For example, an algorithm with time complexity O(n^2) indicates that its runtime grows quadratically with the number of elements in the input list.

The choice of sorting algorithm depends on various factors, including the size of the input data, the available memory resources, and the desired performance characteristics. Some algorithms may have better space complexity but higher time complexity, while others may prioritize time efficiency at the expense of increased memory usage. Evaluating the trade-offs between space and time complexity is essential in selecting the most suitable sorting algorithm for a particular application or problem.

**Selection Sort**

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

**Algorithm: SELECTION_SORT(array)**

**Input:**

- array: An array of elements to be sorted.

**Output:**

- Sorted array in ascending order.

**Steps:**

1. Set n to the length of the array.
2. Repeat the following steps for i from 0 to n-1:
   a. Set minIndex to i.
   b. Iterate j from i+1 to n-1:
      - If array[j] is less than array[minIndex], update minIndex to j.
   c. Swap array[i] with array[minIndex].
3. End of loop.
4. Return the sorted array.

**Time complexity of selection sort**

Best case complexity→O(N)

Average case complexity→$O(N^2)$

Worst Case→ $O(N^2)$

**Insertion Sort**

Insertion Sort is a simple sorting algorithm that works by repeatedly inserting elements from an unsorted part of the array into their correct positions within a sorted sublist. It divides the array into two parts: a sorted sublist and an unsorted sublist. Initially, the sorted sublist contains only the first element, and the rest of the elements are in the unsorted sublist. The algorithm then iterates through the unsorted sublist, selecting one element at a time and inserting it into its proper position in the sorted sublist. This process continues until all elements are sorted. Insertion Sort is efficient for small datasets or nearly sorted arrays but becomes less practical for large datasets due to its quadratic time complexity. Despite its simplicity, Insertion Sort is often used in practice for sorting small arrays or as a building block for more advanced sorting algorithms.

**Algorithm: INSERTION_SORT(array)**

**Input:**

- array: An array of elements to be sorted.

**Output**:

- Sorted array in ascending order.

**Steps**:

1. Set n to the length of the array.
2. Iterate i from 1 to n-1:
   a. Set key to array[i].
   b. Set j to i-1.
   c. While j is greater than or equal to 0 and array[j] is greater than key:
      - Move array[j] one position ahead.
      - Decrement j.
   d. Place key at position j+1.
3. End of loop.
4. Return the sorted array.

**Time complexity of insertion sort**

Best case complexity→O(N)

Average case complexity→$O(N^2)$

Worst Case→ $O(N^2)$

**Merge Sort**

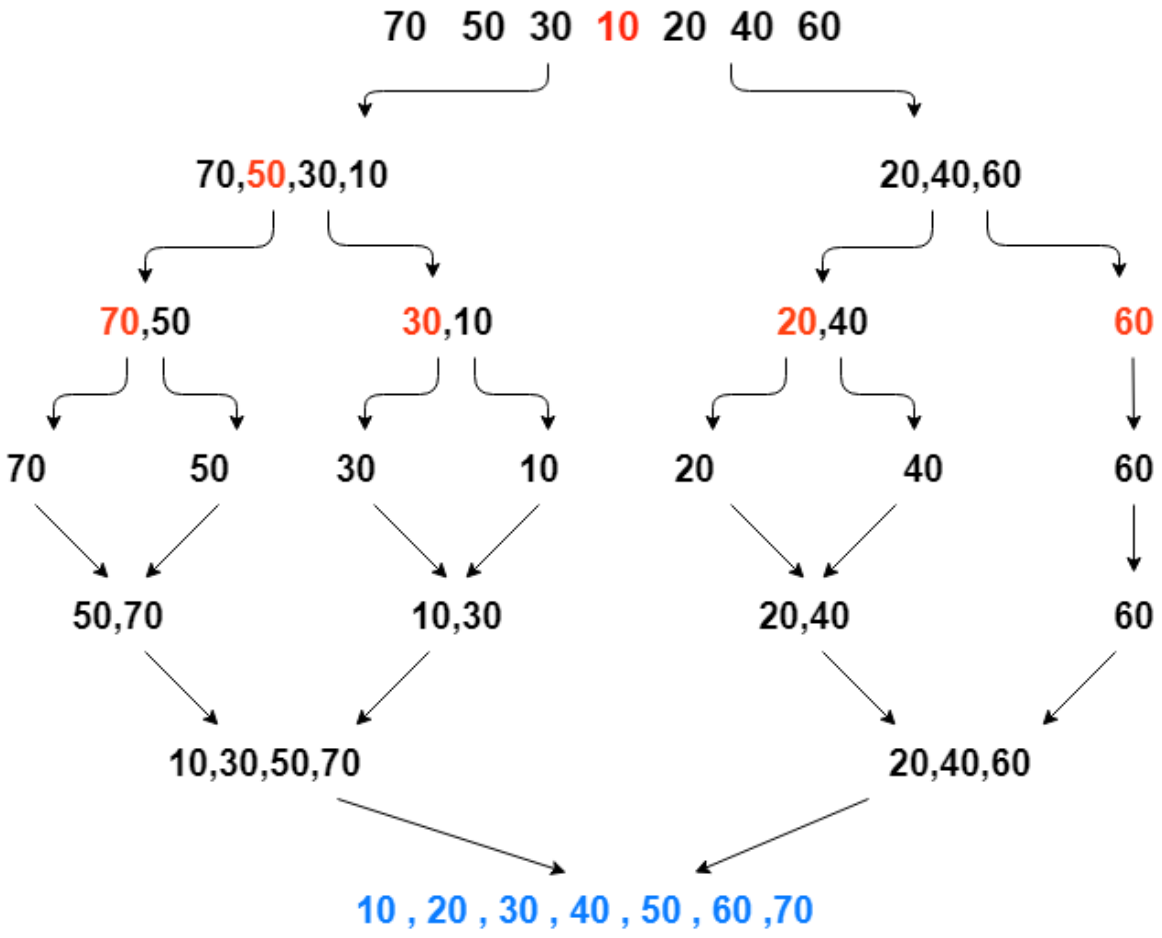
**Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

**The concept of Divide and Conquer involves three steps:**

- Divide the problem into multiple sub-problems.
- Solve the Sub Problems. The idea is to break down the problem into atomic sub-problems, where they are actually solved.
- Combine the solutions of the sub-problems to find the solution of the actual problem.

**So, the merge sort working rule involves the following steps:**

- Divide the unsorted array into sub-array, each containing a single element.
- Take adjacent pairs of two single-element array and merge them to form an array of 2 elements.
- Repeat the process till a single sorted array is obtained.



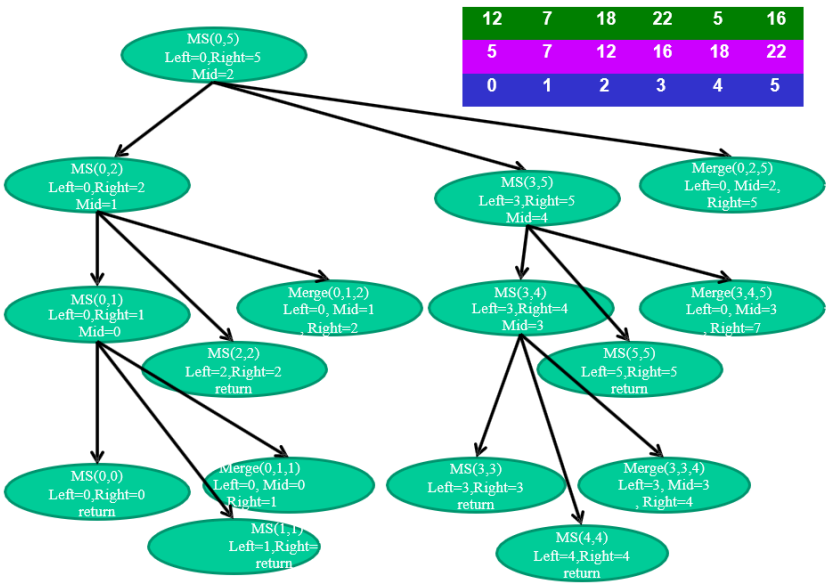

Algorithm

1. Divide: The array is divided into two halves by calculating the mid-index.

2. Conquer Part 1: The left half of the array is recursively sorted using the mergeSortAlgorithm function with the mid-index as the right end.
3. Conquer Part 2: Similarly, the right half of the array is recursively sorted using the mergeSortAlgorithm function with mid + 1 as the left end.
4. Combine: The sorted halves are merged together using the merge function to form the final sorted array.
5. Base Case: If the indices I and r become equal during recursive calls, it means the sub-array has only one element left, which is already sorted. In such cases, recursion stops, and the function returns.

This recursive approach continues until the base case is reached, ensuring that each sub-array is sorted individually before being merged with other sub-arrays.

```
void mergeSort(int a[], int beg, int end)
{
   if (beg < end)
   {
      int mid = (beg + end) / 2;
      mergeSort(a, beg, mid);
      mergeSort(a, mid + 1, end);
      merge(a, beg, mid, end);
   }
}
```



**HEAP SORT**

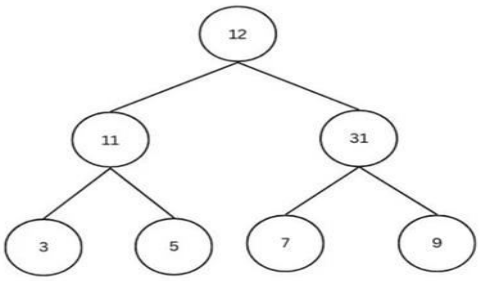
Heap sort is a sorting algorithm that organizes elements in an array into a binary heap, and then sorts that heap by moving the largest element in the array.

**Algorithm for Heap Sort**

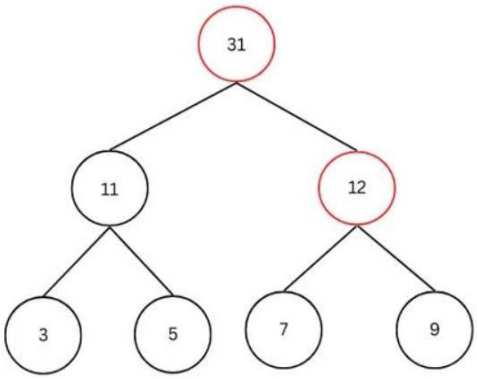
1. Transform the array into a binary tree by inserting each element as a node in a breadth-first manner.
2. Convert the binary tree into a max heap, ensuring that all parent nodes are greater than or equal to their child nodes.
3. Swap the root node — the largest element — with the last element in the heap.
4. Call the heapify() function to restore the max heap property.
5. Repeat steps 3 and 4 until the heap is sorted, and exclude the last element from the heap on each iteration.
6. After each swap and heapify() call, ensure that the max heap property is satisfied.

[12, 11, 31, 3, 5, 7, 9]

we can start by inserting the first element of the array as a node into the tree and continuing this process in a breadth-first manner until all elements of the array have been added.

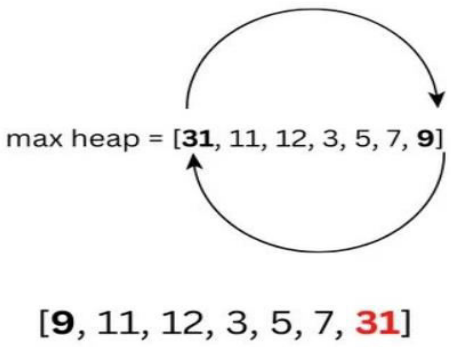



3. **Convert the Binary Tree Into a Max Heap:** all parent nodes must have values that are greater than or equal to the values of their children. this means swapping node 12 and node 31 positions in the tree to satisfy the requirements for a max-heap tree




max heap = [31, 11, 12, 3, 5, 7, 9]

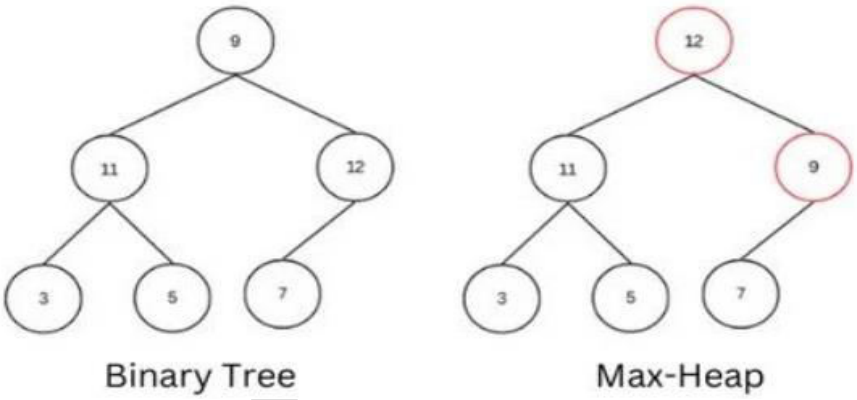
**3. Swap the Root Node With the Last Element in the Heap:** The next step in the sorting process is to swap the root node, which contains the largest element, with the last node in a heap. In other words, you're swapping the element in the first position of the max-heap array with the element in the last position of the max-heap array. Whatever method you choose to understand this step, 31 ends at the end of the array, and nine ends at the first position of the array




in future iterations, we will be omitting the last value because it's in a sorted position. Therefore, we move forward with the following array into the next step:

[9, 11, 12, 3, 5, 7]

Now, we will transform the array into a tree, then the tree into a max heap.




After swapping the root node with the last element in the heap, the visualized heap will now have one less element.

This is because the largest element has been placed at the end of the array and will be excluded from future iterations.

If it were not excluded, the sorting algorithm would never finish.

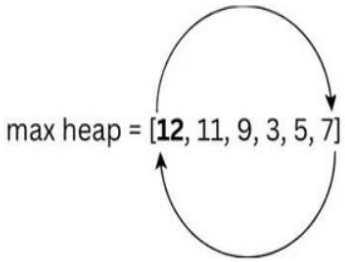
Therefore, every time the root node is swapped, the next iteration should exclude the last element in the heap.

**4. Call the heapify() Function**

Let's now refer to the process of converting the tree or array into a max heap as heapify

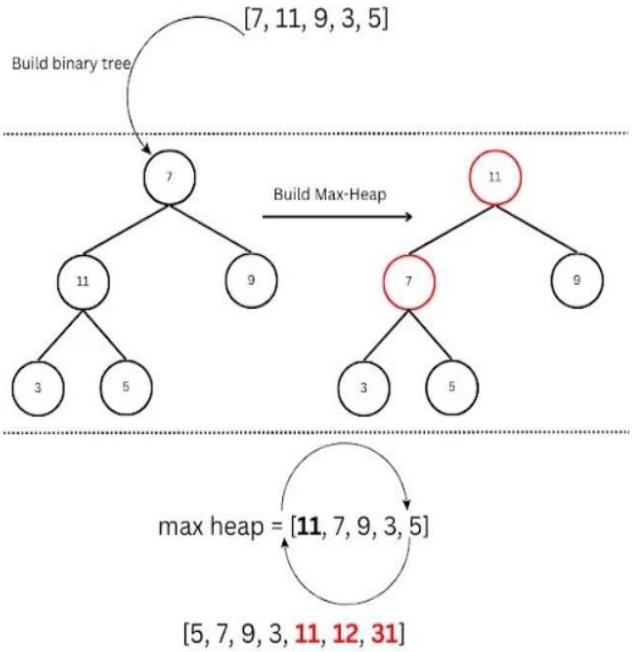
The heapify() function should be called again to restore the max heap property. This will result in the heap being rearranged as shown:

[12, 11, 9, 3, 5, 7]

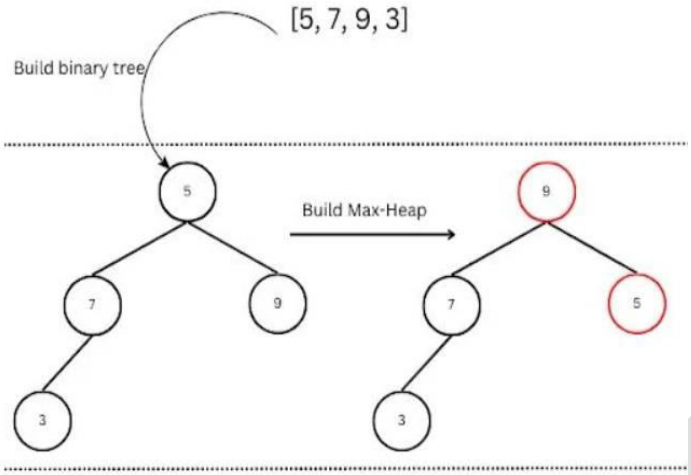



[7, 11, 9, 3, 5, 12, 31]

**5. Repeat Steps 3 and 4 Until the Heap Is Sorted**

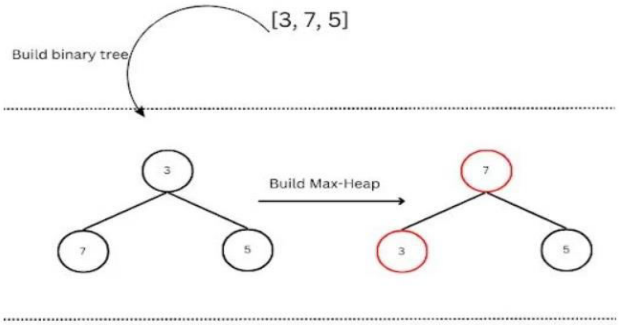



[5, 7, 9, 3, 11, 12, 31]



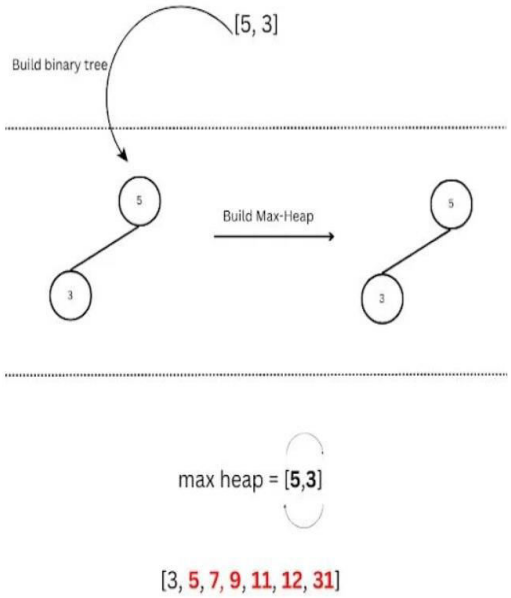

max heap = [**9**, 7, 5, **3**]

[3, 7, 5, **9**, **11**, **12**, **31**]




max heap = [**7**, 3, **5**]

[5, 3, **7**, **9**, **11**, **12**, **31**]




**The final sorted array:**

**[3, 5, 7, 9, 11, 12, 31]**

The heap sort algorithm's best, worst and average **time complexities** are all the same —**O(n*log(n))**.

Heap sort is called **an in-place algorithm** because it does not require extra memory space to sort. It uses the same array for both the elements' storage and the sorting process. This is done by rearranging the elements of the array in place to satisfy the max-heap property, which is used to sort the array.

**RADIX SORT**

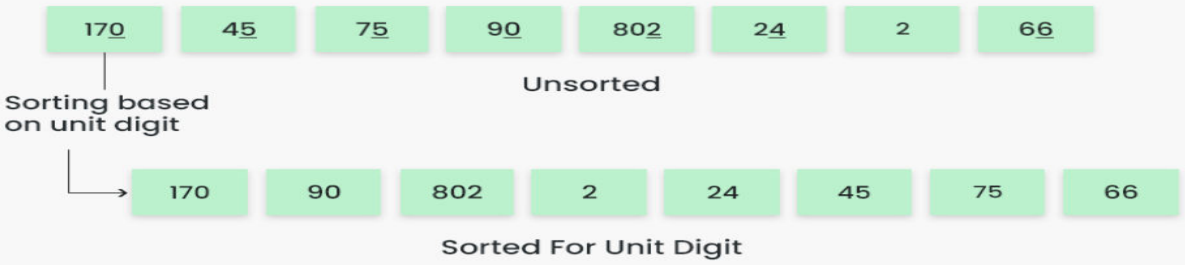
**Radix Sort** is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

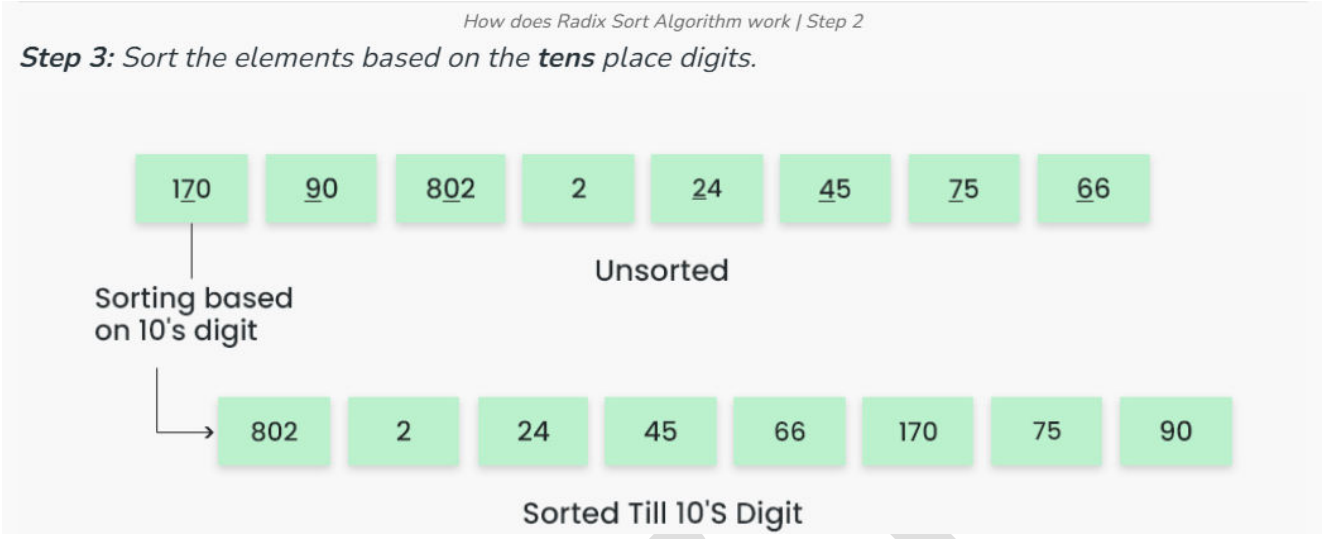
- It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.
- By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.
- We use a stable algorithm like Counting Sort to sort the individual digits so that the overall algorithm remains stable.

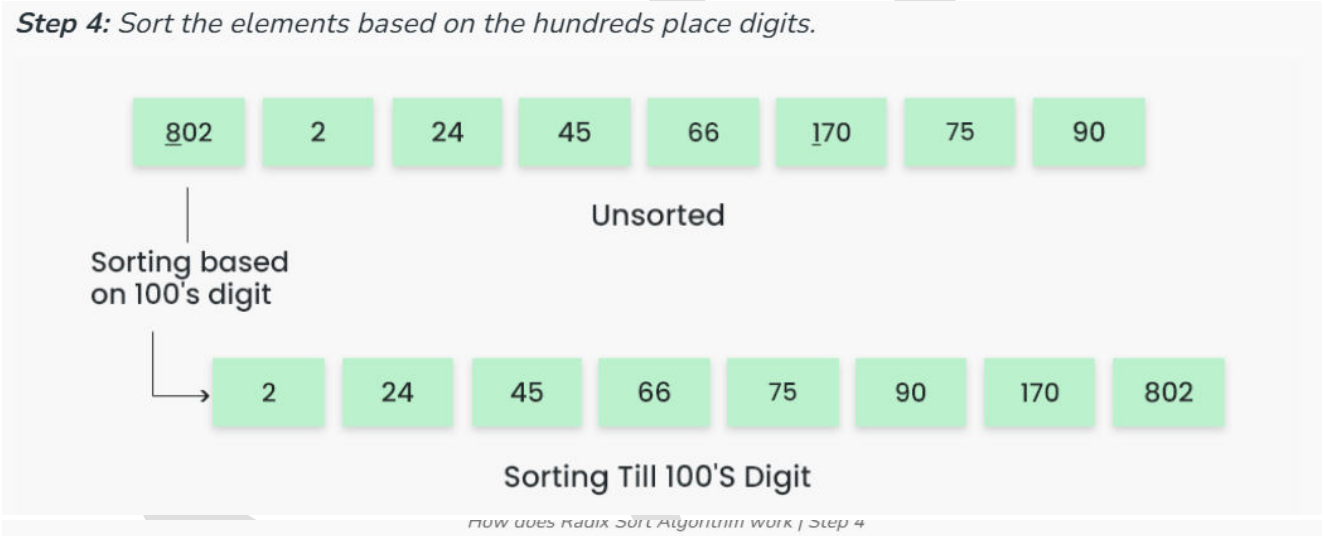
*To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:*

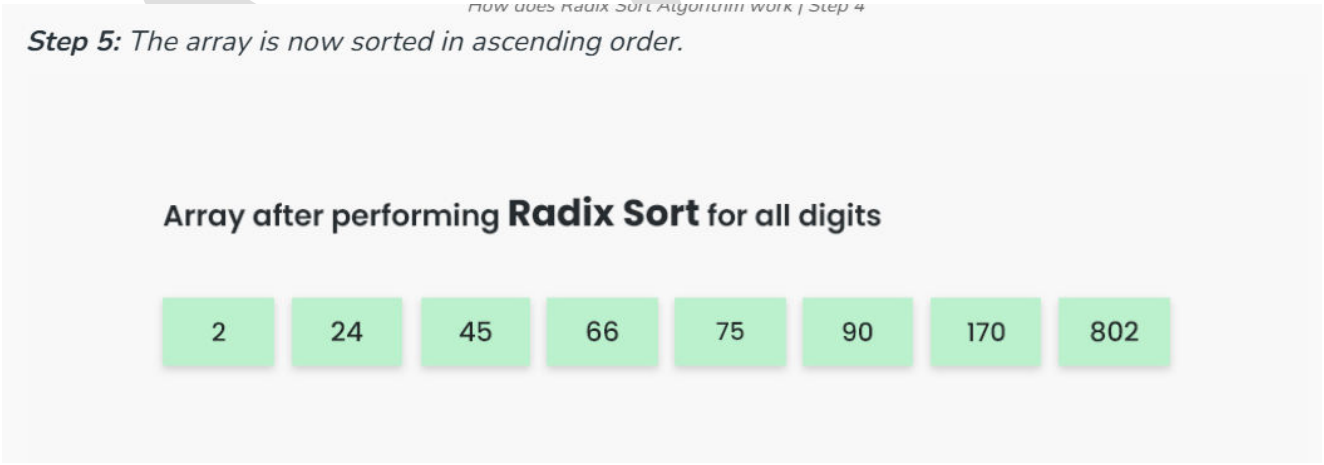
***Step 1:*** *Find the largest element, which is 802. It has three digits, so we will iterate three times.*

***Step 2:*** *Sort the elements based on the* ***unit*** *place digits (X=0).*






**Time Complexity:**

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value.

It has a time complexity of **O(d * (n + b))**, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.

In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

**Hashing**

- Hashing in the data structure is used to quickly identify a specific value within a given array.
- It creates a unique hash code for each element in the array and then stores the hash code instead of the actual element.
- This allows for quick lookup when searching for a specific value, as well as easy identification of any duplicates. Hashing in the data structure is a technique that is used to quickly identify a specific value within a given array.
- It works by creating a unique hash code for each element in the array and then stores the hash code in lieu of the actual element.
- This allows for quick look-up when searching for a specific value, as well as easy identification of any duplicates.

**Components of Hashing**

There are majorly three components of hashing:

- **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
- **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
- **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

**How does Hashing work?**

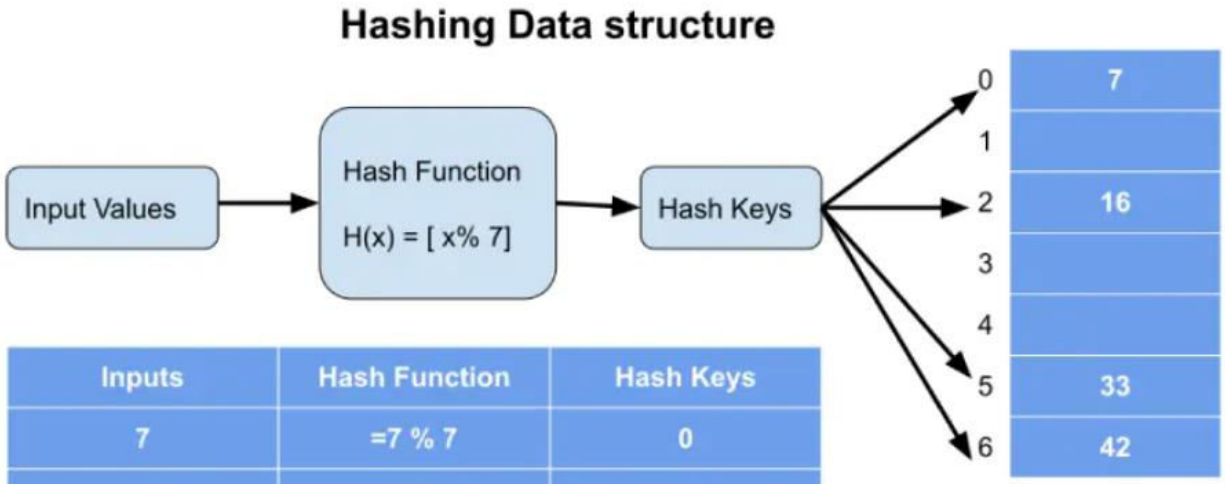
Eg 1:

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
  - "a" = 1,
  - "b"=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
  - *"ab" = 1 + 2 = 3,*
  - *"cd" = 3 + 4 = 7 ,*
  - *"efg" = 5 + 6 + 7 = 18*
- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.
- **Step 5:** So we will then store
  - "ab" in 3 mod 7 = 3,
  - "cd" in 7 mod 7 = 0, and
  - "efg" in 18 mod 7 = 4.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| cd | | | ab | efg | | |

Figure 6:Mapping key with indices of array




| Inputs | Hash Function | Hash Keys |
|---|---|---|
| 7 | =7 % 7 | 0 |
| 16 | =16 % 7 | 2 |
| 33 | =33 % 7 | 5 |
| 42 | =42 % 7 | 6 |

- Consider a hash table of size 10 whose indices are 0,1,….8,9.
- Let the set of key values :10,19,35,43,62,59,31,49,77,33
- The hash function:
  - Add two digits in the key
  - Take the digit at unit place of the result as index. Ignore the digit at decimal place, if any.
- key values :10,19,35,43,62,59,31,49,77,33

Hash table

| key | Hash | Index |
|---|---|---|
| 10 | 1+0 | 1 |
| 19 | 1+9 | 0 |

| Index |
|---|
| 0 19 |
| 1 10 |

**What is a Hash function?**

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

**Types of Hash functions:**

1. Division Method

2. Mid Square Method

3. Folding Method

4. Multiplication Method

**Division Method**

The division method is the simplest and easiest method used to generate a hash value. In this hash function, the value of k is divided by M and uses the remainder as obtained.

***Formula - h(K) = k mod M***

(where k = key value and M = the size of the hash table)

**Example –**

k = 1320

M = 11

h (1320) = 1320 mod 11 = 0

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

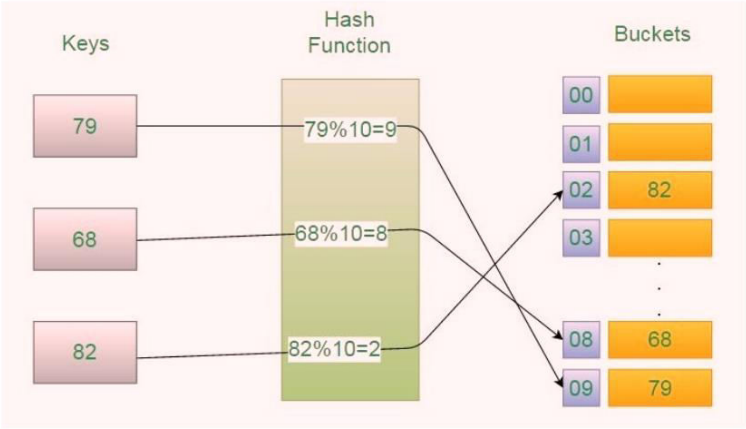
2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8

| | |
|---|---|
| [0] | 72 |
| [1] | |
| [2] | 18 |
| [3] | 43 |
| [4] | 36 |
| [5] | |
| [6] | 6 |
| [7] | |




**Mid Square Method**

The steps involved in computing this hash method include the following -

Squaring the value of k ( like k*k)

Extract the hash value from the middle r digits.

***Formula - h(K) = h(k x k)***

(where k = key value )

Suppose the size of the Hash Table (m) = 10 (0 - 9) *maximum digits required for the index is 1*
Element (x) = 12 $\Rightarrow x^2 = 144$
Mid 1 digit of 1**4**4 is 4, so the element x=12 will be stored at the index=4 in the hash table with the size of 10 slots.

**Another Example:**
Suppose the size of the Hash Table (m) = 1000 (0 - 999) maximum digits required for the index is 3
Element (x) = 87431 $\Rightarrow x^2 = 7644179761$
The possible 3 digit mids of 764***4179***761 are 417 or 179, we can pick any of those mids. If we pick 419 then the element x=87431 will be stored at the index=419 in the hash table with the size of 1000 slots.

**Folding Method**

- There are two steps in this method -
- The key-value k should be divided into a specific number of parts, such as k1, k2, k3,..., kn, each having the very same number of digits aside from the final component, which may have fewer digits than the remaining parts.
- Add each component separately. The last carry, if any, is disregarded to determine the hash value.

***Formula - k = k1, k2, k3, k4, ….., kn***

***s = k1+ k2 + k3 + k4 +….+ kn***

***h(K)= s***

(Where, s = addition of the parts of key k)

Example -

k = 54321

k1 = 54 ; k2 = 32 ; k3 = 1

Therefore,

s = k1 + k2 + k3

= 54 + 32+ 1

= 87

Thus,

h (k) = 87

Ex. (SSN) 123-45-6789

1. Divide into 3 parts: **123, 456 and 789.**
2. Add them. **123+456+789=1368**
3. h(k)=k mod M
   where M = 1000
   h(1368) = 1368 mod 1000
   = **368**

1. Divide into five parts: **12, 34, 56, 78 and 9.**
2. Add them. **12 + 34 + 56 + 78 + 9 = 189**
3. h(k)=k mod M
   where M = 1000
   h(189) = 189 mod 1000
   = **189**

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|
| 123 | 203 | 241 | 112 | 20 |

| | |
|---|---|
| $P_1$ | 123 |
| $P_2$ | 203 |
| $P_3$ | 241 |
| $P_4$ | 112 |
| $P_5$ | 20 |
| | 699 |

Shift Folding

| | |
|---|---|
| $P_1$ | 123 |
| $P_2^r$ | 302 |
| $P_3$ | 241 |
| $P_4^r$ | 211 |
| $P_5$ | 20 |
| | 897 |

Folding at boundaries

**Multiplication Method**

Steps to follow -

- Pick up a constant value A (where $0 < A < 1$)
- Multiply A with the key value
- Take the fractional part of kA
- Take the result of the previous step and multiply it by the size of the hash table, M.

***Formula - h(K) = floor (M (kA mod 1))***

(Where, M = size of the hash table, k = key value and A = constant value)

**Example -**

k = 1234

A = 0.35784

M = 100

So,

h (1234) = floor [ 100(1234 x 0.35784 mod 1)]

= floor [ 100 ( 441.57456 mod 1)]

= floor [100 ( 0. 57456)]

= floor [ 57.456]

= 57

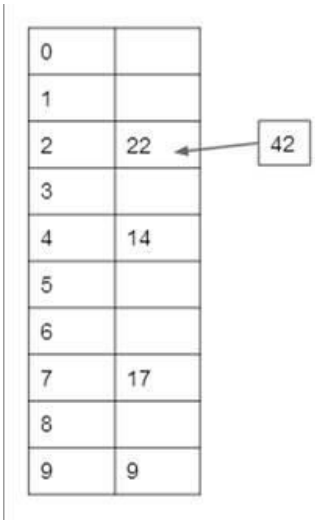
Thus,

h (1234) = 57

**Hash Collision**

- A hash collision happens when the same hash value is produced for two different input values by a hash algorithm.
- Collisions occur because different hashing techniques in data structure convert every input into a fixed-length code, regardless of its length.
- Since there are an endless number of inputs and a limited number of outputs, the hashing algorithms will eventually produce repeating hashes.

- Let's take the exact same hash function from before: take the value to be hashed mod 10, and place it in that slot in the hash table.
- Numbers to hash: 22, 9, 14, 17, 42
- As before, the hash table is shown to the right.
- As before, we hash each value as it appears in the string of values to hash, starting with the first value. The first four values can be entered into the hash table without any issues. It is the last value, 42, however, that causes a problem. 42 mod 10 = 2, but there is *already* a value in slot 2 of the hash table, namely 22. This is a collision.




**Collision resolution techniques**

Collision resolution techniques are strategies used to handle collisions that occur when two or more keys hash to the same index in a hash table. Here's a brief note on some common collision resolution techniques:

**Open Hashing (Chaining):**

- Open hashing, also known as chaining, is a collision resolution technique where each bucket in the hash table maintains a linked list (or another data structure) of key-value pairs that hash to the same index.
- When a collision occurs, the collided key-value pair is inserted into the linked list associated with the corresponding bucket.
- Chaining allows multiple keys to hash to the same index, and collisions are resolved by appending elements to the linked list.

**Closed Hashing (Open Addressing):**

- Closed hashing, or open addressing, is a collision resolution technique where collisions are resolved by finding an alternative location within the hash table itself.

- Unlike chaining, closed hashing stores collided key-value pairs directly in the hash table rather than using auxiliary data structures.
- Various probe sequences are used to locate an empty slot for insertion or to find a key during lookup.

**a. Linear Probing:**

- In linear probing, if a collision occurs at index i, the algorithm probes the next index i+1, then i+2, and so on until an empty slot is found.
- The probe sequence is linear, which means that collisions are resolved by incrementing the index linearly until an empty slot is found.
- Linear probing tends to cause clustering, where consecutive occupied slots create long chains of contiguous filled slots.

**b. Quadratic Probing:**

- Quadratic probing uses a quadratic probe sequence to resolve collisions.
- Instead of probing linearly, the algorithm uses a quadratic function to determine the next probe index.
- The probe sequence follows the formula h(k) ± c^2 (where c is a constant), which spreads out the search and reduces clustering compared to linear probing.

**c. Double Hashing:**

- Double hashing resolves collisions using two hash functions: the primary hash function determines the initial index, and the secondary hash function determines the step size (stride) for probing.
- If a collision occurs at index i, the probe sequence follows the formula i = (hash1(key) + j * hash2(key)) % tableSize.
- Double hashing offers good performance and avoids clustering by varying the probe sequence based on both the key and the table size.

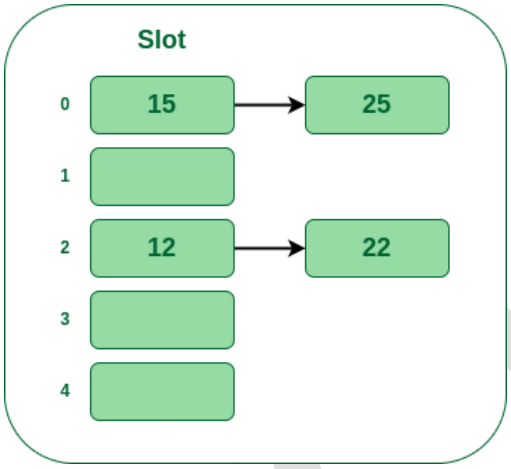
**Separate Chaining**

- The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

- **Hash function = key % 5,**
  **Elements = 12, 15, 22, 25 and 37.**

- **Step 1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.

- **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.
- **Step 3:** Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12.
- **Step 4:** The next key is 15. It will map to slot number 0 because 15%5=0.
- **Step 5:** Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.




**Open Addressing**

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

**2.a) Linear Probing**

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

**2.b) Quadratic Probing**

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

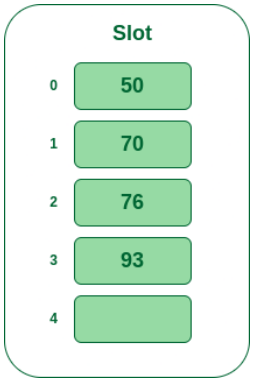
**2.c) Double Hashing**

The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.

### 2.a) Linear Probing

**Example:** Let us consider a simple hash function as ***"key mod 5"*** and a sequence of keys that are to be inserted are ***50, 70, 76, 85, 93***. insert all the keys in the hash table one by one.

- The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.
- The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.
- The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.
- The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.

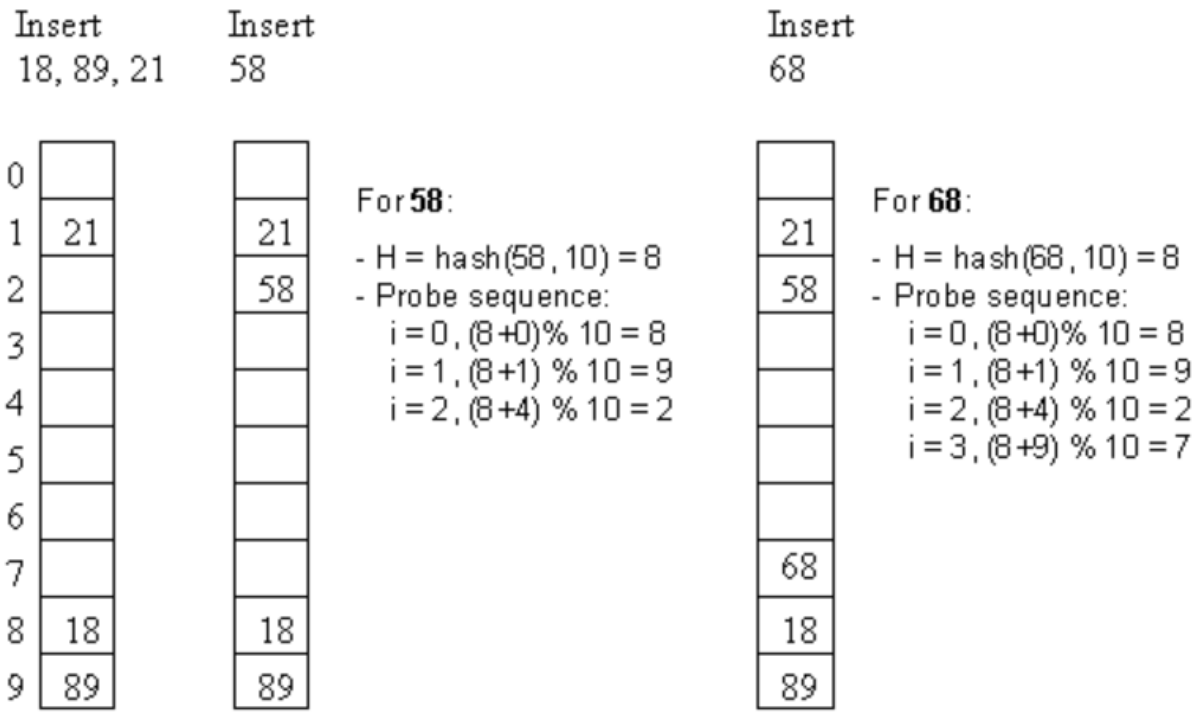



### Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

***H*** + *12*, ***H*** + *22*, ***H*** + *32*, ***H*** + *42*…… ***H*** + *k2*




**2.c) Double Hashing**

The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.

The formula for the double hashing technique is as follows:

**(firstHash(key) + i * secondHash(key)) % sizeOfTable**

Where i is the offset value. This offset value keeps incremented until it finds an empty slot.

**Example:** Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
- 7 % 7 = 6, location 6 is empty so insert 27 into 6 slot.
- **Step 2:** Insert 43
- 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.
- **Step 3:** Insert 692
- 692 % 7 = 6, but location 6 is already being occupied and this is a collision

- So we need to resolve this collision using double hashing.

hnew = [h1(692) + i * (h2(692)] % 7
= [6 + 1 * (1 + 692 % 5)] % 7
= 9 % 7
= 2
Now, as 2 is an empty slot, so we can insert 692 into 2nd slot.

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

hnew = [h1(72) + i * (h2(72)] % 7
= [2 + 1 * (1 + 72 % 5)] % 7
= 5 % 7
= 5,

(***suppose 5 is not an empty sort we do double hashing again with i=2)***
Now, as 5 is an empty slot, so we can insert 72 into 5th slot.

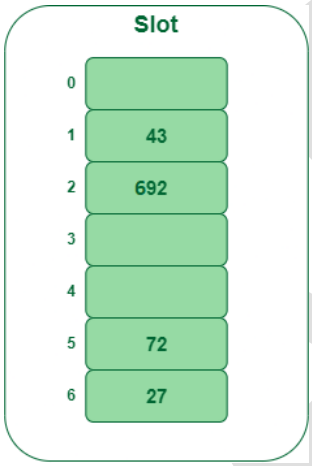


Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys: 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

| | |
|---|---|
| [0] | 49 |
| [1] | |
| [2] | |
| [3] | 69 |
| [4] | |
| [5] | |
| [6] | |
| [7] | 58 |
| [8] | 18 |
| [9] | 89 |