Unit 8: Searching, Sorting and Hashing
8.1 Introduction
8.2 Sequential and Binary Search
8.3 Hashing: Hash function (truncation, division method, folding, midsquare)
8.4 Hash collision and resolution techniques
8.5 Sorting Algorithms: Bubble, Selection, Insertion, Merge, Quick and Heap Sort
8.6 Efficiency of Sorting Algorithms

<u>Practical Works</u>

8.1 Write program to implement:
 a) Bubble sort b) Selection sort c) Insertion sort
 d) Quick sort e) Merge sort f) Heap sort

8.2 Write program to implement searching algorithms: binary search and linear search

8.3 Write program to implement hash function

8.1 Introduction:

Searching is the process of finding an element within the list of elements stored in any order or randomly. Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue.

- **Types of Searching**

  1. Internal Searching:
  - ❖ Searching in primary memory i.e. retrieval of information from RAM with the help of key.
  - ❖ Used for less element (small volume of data).

  2. External Searching:
  - ❖ Searching in a secondary storage (disk) i.e. retrieval of information from secondary memory (disk) with the help of key.
  - ❖ Used for more elements (large volume of data).

8.2 Sequential and Binary Search

Sequential / Linear Search is one of the internal searching technique. In this type of searching, we access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found.

In brief, simply search for the given element left to right and return the index of the element if found. Otherwise return “Not Found”.

Algorithm

Step 1: Read the search element from the user.
Step 2: Compare the search element with the first element in the list.
Step 3: If both are matching, then display “Given Element Found” and terminate the function.
Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat Step 3 and Step 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list also doesn't match then display "Element not found" and terminate the function.

Example

Example:

Consider the following list of element and Search element:

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

Search element 12

Step 1:

Search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next elements.

Step 2:

Search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
		12						

Both are not matching. So move to next element.

Step 3: search element (12) is compared with (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
			12					

Both are not matching. So move to next element.

Step 4:

Search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
				12				

Both are not matching. So move to next element.

Step 5:

Search element(12) & compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
					12			

Both are not matching. So move to next element.

Step 6:

Search element(12) & compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
						12		

Both are matching. So we stop comparing and display element found at index 5.

- **Binary Search**

  Binary search, also known as half-interval search or logarithmic search is a search algorithm that finds the position of a target value within a sorted array.

  Binary search is an efficient algorithm for finding an item from a sorted list of items. It works repeatedly dividing in half the portion of the list that could contain the item, until you have narrowed down the possible locations to just one.

Algorithm

Step 1: Sort the elements in ascending order.

Step 2: Set **lower bound (LB) =0** and **upper bound (UB) =n-1**.

Step 3: Find the value of middle as ***mid***= $\frac{LB+UB}{2}$

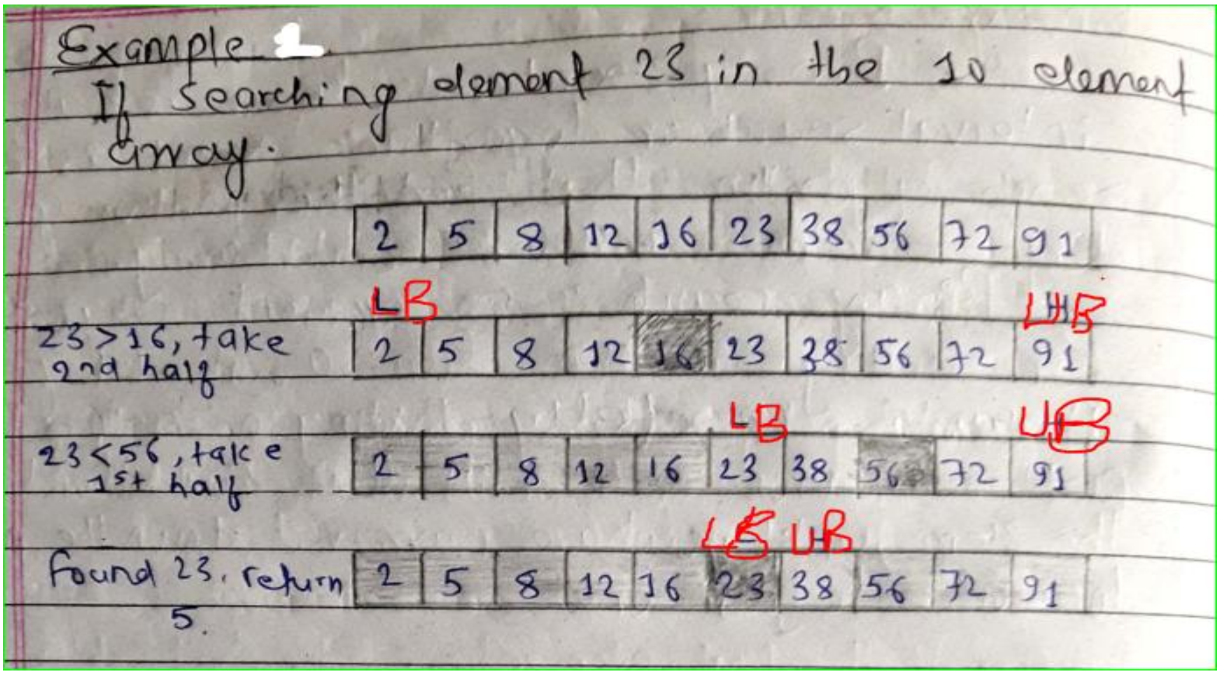
Step 4: if ***A[mid] > key***

Change ***UB*** to ***mid-1*** [UB = mid-1]

else

Change ***LB*** to ***mid+1*** [LB = mid+1]

Step 5: Repeat **Step 3** and Step 4 until the required element is found or ***LB>UB***

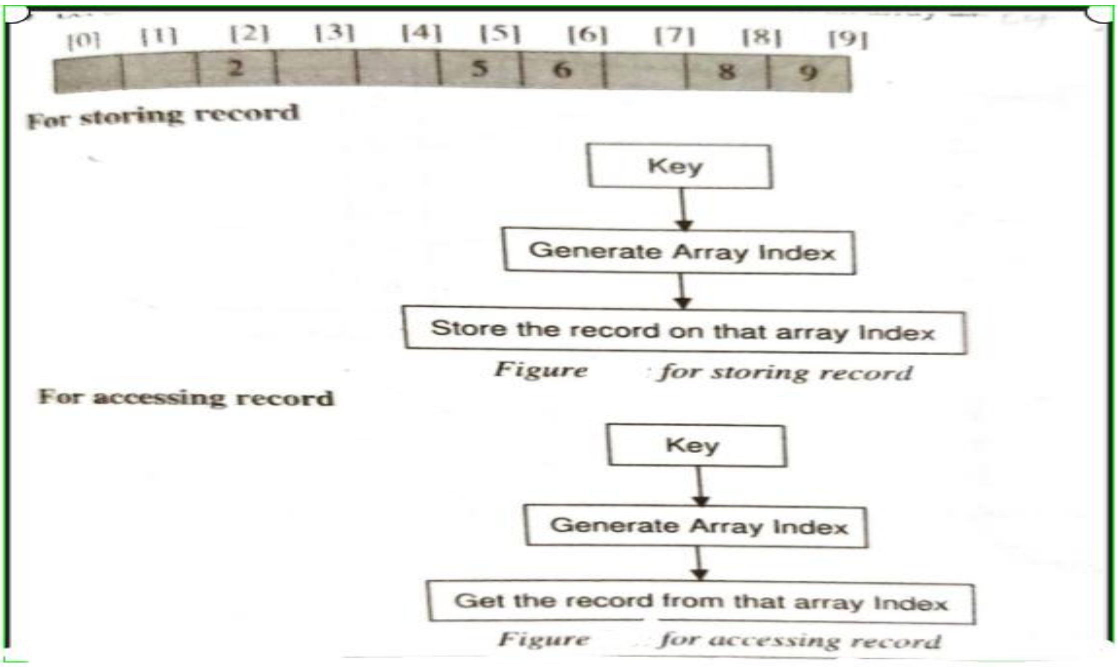


8.3 Hashing : Hash function (truncation, division method, folding, midsquare)

Hashing is the process of searching the desired element from the given set of data. In general, if the data set is huge then the high number of comparisons is required.

Hashing is a technique of finding a desired number from a given set of data with a minimum number of comparisons

Let us take 5 numbers (9.6, 5, 8, 2) which is stored in an array of 10 elements as



- Hash Table:-

A hash table consists of an array in which data is accessed via a special index called a ***key***. The primary idea behind the hash table is to establish a mapping between the set of all possible ***keys*** and ***position*** in the array using ***hash function***. A hash function accepts a key and returns its ***hash coding***, ***hash value***. Keys vary in type, but hash coding are always integers. When hash functions guarantee that no two keys will generate the same hash coding, the resulting hash table is said to be ***direct addressed***. When two keys maps to the same position they ***collide***. A good hash function minimizes the ***collisions***.

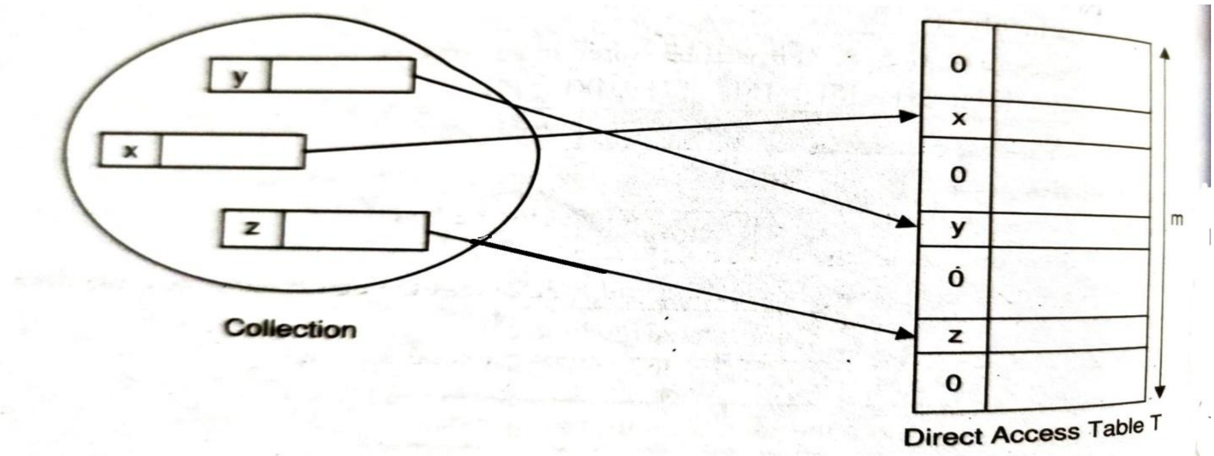


Figure **:** ***A Hash Table***

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values have its own unique index value. Access of data becomes very fast if we know the index of desired data.

- Hash Function:

Hash function is a function which takes a piece of data (i.e key) as an input and outputs an integer (i.e hash value) which maps the data to a particular index in the hash table. The desired property of hash function is those that compute and minimize the number of collisions.

Various types of Hash Functions:

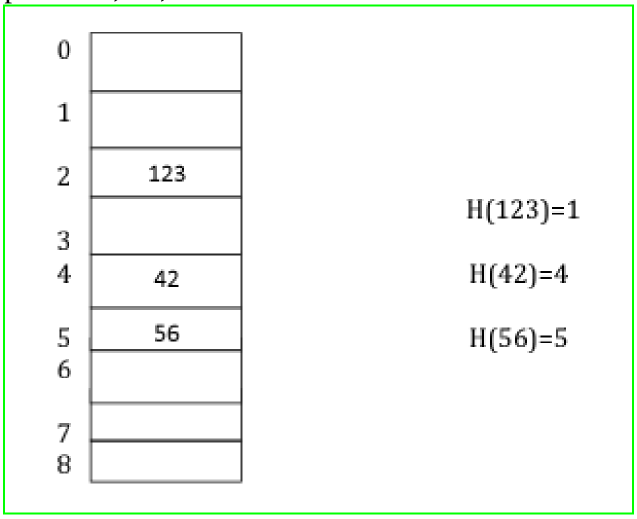
1. Truncation Method
2. Folding Method
3. Midsquare Method
4. Division Method (Modulo Division)

1. Truncation Method:

The Truncation Method truncates a part of the given keys, depending upon the size of the hash table.

1. Choose the hash table size.

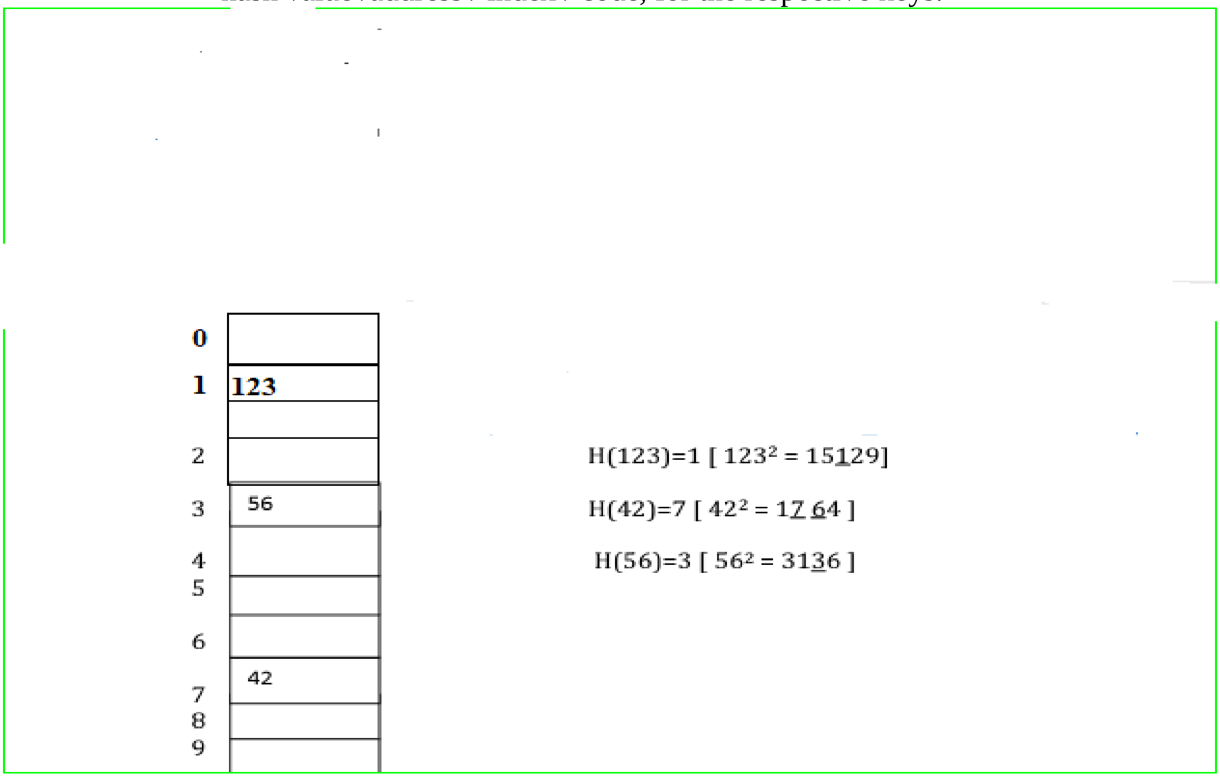
2. Then the respective right most or left most digits are truncated and used as hash code value.
Example : 123, 42, 56   Table size = 9



2. Midsquare Method:

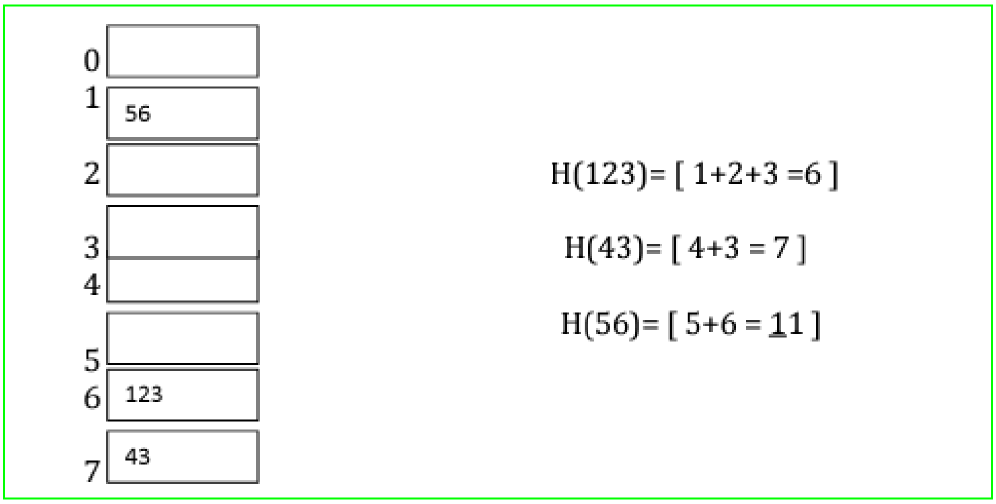
It is a hash function method.

1. Square the given keys.
2. Take the respective middle digits from each squared value and use that as the hash value /address / index / code, for the respective keys.



3. Folding Method:

Partition the key K into number of parts like K_1, K_2,…..Kn , then add the parts together and ignore the carry and use it as the hash value.



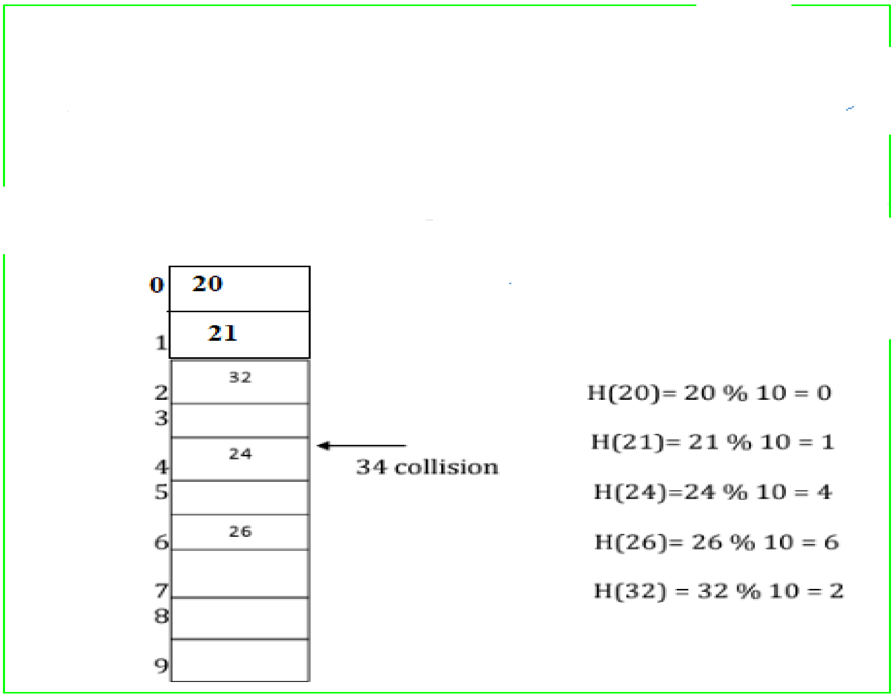
4. Division Method:

Choose a number ***m***, larger than the number of keys. The number ***m*** is usually chosen to be a prime number.

The formula for the division method:

Hash (key) = key % table size

Table size: 10   20, 21, 24, 26, 32, 34



8.4 Hash collision and resolution techniques.

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.**

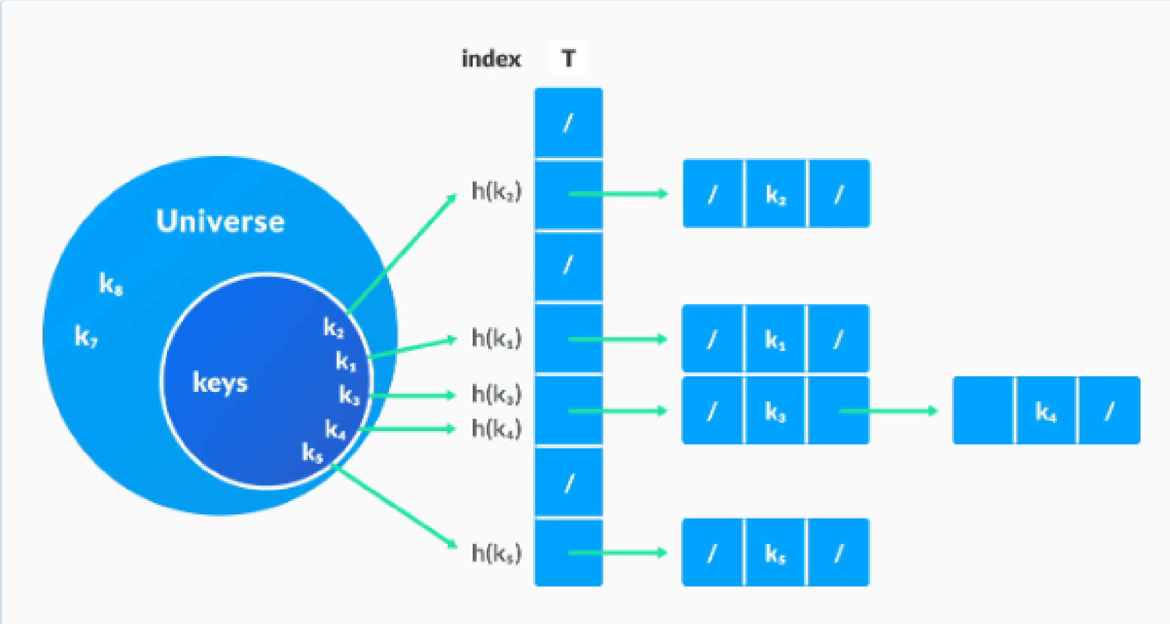
We can resolve the hash collision using one of the following techniques.

1. Collision resolution by chaining
2. Open Addressing: Linear/Quadratic Probing and Double Hashing

1. Collision resolution using 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.

$h(k, i) = (h'(k) + i) \bmod m$

where

$i = \{0, 1, \ldots.\}$

$h'(k)$ 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,
$i = \{0, 1, \ldots.\}$

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.
$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$

8.5 Sorting Algorithms: Bubble, Selection, Insertion, Merge, Quick and Heap Sort

- Sorting :

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types.

1. Internal Sorting:

This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required in this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sort, they can only process relatively small lists due to memory constraints.

Types of internal sorting are
a) Selection Sort: Eg. Selection sort, Heap sort
b) Insertion Sort: Eg. Insertion sort, Shell sort
c) Exchange Sort: Eg. Bubble sort, Quick sort

2. External Sorting:

Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together. Eg. Merge Sort

[redacted]

1. Bubble Sort:

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(5 1 4 2 8) -> (1 5 4 2 8), Here, algorithm compares first two elements, and swaps since 5>1.

(1 5 4 2 8) ->(1 4 5 2 8), Swap since 5>4

(1 4 5 2 8) ->(1 4 2 5 8), Swap since 5>2

(1 4 2 5 8) -> (1 4 2 5 8), Now, since these elements are already in order (8>5), algorit-hm does not swap them.

Second Pass:

(1 4 2 5 8) ->(1 4 2 5 8)

(1 4 2 5 8) ->(1 2 4 5 8), Swap since 4>2

(1 2 4 5 8) -> (1 2 4 5 8)

(~~1 2~~

Now, the array is already sorted, but our ~~order~~ algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third pass:
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)

Algorithm ->
begin bubble Sort (list)
for all elements of list
if list [i] > list [i+1]
swap (list [i], list [i+1])
end if
end for
return list
end BubbleSort

Bubble Sort Example 2:

Initially

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 2

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

2 Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

i) The sub-array which is already sorted
ii) Remaining sub-array which is unsorted.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$; where n is the number of items.

Example

Assume that the array A = [7, 5, 4, 2] needs to be sorted in ascending order.

Step 1. | 7 | 5 | 4 | 2 | → | 2 | | 5 | 4 | 7 |
↑min element — Sorted — Unsorted

Step 2. | 2 | | 5 | 4 | 7 | → | 2 | 4 | | 5 | 7 |
↑min element — Sorted — Unsorted

Step 3. | 2 | 4 | | 5 | 7 | → | 2 | 4 | 5 | | 7 |
↑min element — Sorted — Unsorted

Step 4. | 2 | 4 | 5 | | 7 | → | 2 | 4 | 5 | 7 |
↑min element — Sorted Array

fig: Selection Sort

Algorithm

Step 1: Set MIN to location 0.
Step 2: Search the minimum element in the list.
Step 3: Swap with value at location MIN.
Step 4: Increment MIN to point to next element.
Step 5: Repeat until list is sorted.

[redacted]

3. Insertion Sort:

Insertion sort is a simple sorting algorithm that works the way we sort ~~sor~~ playing cards in our hands.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as ~~the~~ its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Example:

5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Algorithm for insertion Sort.

Step 1 - If it is the first element it is already sorted ~~ret~~ return 1;

Step 2 - Pick the next

Step 3 - Compare with all elements in the sorted sub-list.

Step 4 - Shift all the elements in the sorted sub-list that is greater than the value to be sorted.

Step 5 - Insert the value.

Step 6 - Repeat until list is sorted.

Divide and Conquer Sorting : Merge, Quick and Heap Sort

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps:

i.) Divide: Break the given problem into subproblems of same type.

ii) Conquer: Recursively solve these subproblems.

iii) Combine: Appropriately combine the answers.

1. Merge Sort:

Merge sort is a sorting algorithm based on divide and conquer technique. The algorithm divides the array into two equal halves, recursively sorts them and finally merges/combines them in a sorted manner.

Algorithm

Step1- if it is only one element in the list it is already sorted, return.

Step 2:- divide the list recursively into two halves until it can no more be divided.

Step3- Merge the smaller lists into new list in sorted order

Sample:

14 | 33 | 27 | 10 | 35 | 19 | 42 | 44

14 | 33 | 27 | 10   35 | 19 | 42 | 44

14 | 33   27 | 10   35 | 19   42 | 44

14   33   27   10   35   19   42   44

14 | 33   10 | 27   19 | 35   42 | 44

10 | 14 | 27 | 33   19 | 35 | 42 | 44

10 | 14 | 19 | 27 | 33 | 35 | 42 | 44

2. Quick Sort:

Quick sort is based on the divide and conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: left side of pivot contains all the elements that are less than the pivot element. Right side contains all elements greater than the pivot.

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in array results in improved time complexity in most of the cases.

~~Example~~: There are different versions of quicksort that pick pivot in different ways:

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

Algorithm.

Step 1 - Make the right-most index value pivot.
Step 2 - Partition the array using pivot value.
Step 3 - quicksort left partition recursively.
Step 4 - quicksort right partition recursively.

Example

p 0	1	2	3	4	5	6	7	8	r 9
9	7	5	11	12	2	14	3	10	6

↓

p 0	1	q 2	3	4	5	6	7	8	r 9
5	2	3	6	12	7	14	9	10	11

↓

p 0	q 1	r 2		p 4	5	6	q 7	8	r 9
2	3	5	6	7	9	10	11	14	12

↓

pr 0		pr 2		p 4	5	qr 6		pq 8	r 9
2	3	5	6	7	9	10	11	12	14

↓

				p 4	qr 5				pr 9
2	3	5	6	7	9	10	11	12	14

↓

				pr 4					
2	3	5	6	7	9	10	11	12	14

3. Heap Sort:

Heap Sort is a comparison based sorting technique based on Binary heap data structure. It is similar to selection sort where we find ~~first~~ the maximum element and place the maximum ~~value~~ element at the end. We repeat the same process for remaining element.

Binary Heap:

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that values in a parent node is greater (or smaller) than the values in two children nodes.

Algorithm:

1- Creating a Heap of the unsorted list / array.
2- Then a sorted ~~list is~~ array is created by repeatedly removing the largest / smallest element from the heap and inserting it into the array.
3. Heap is reconstructed after each removal.

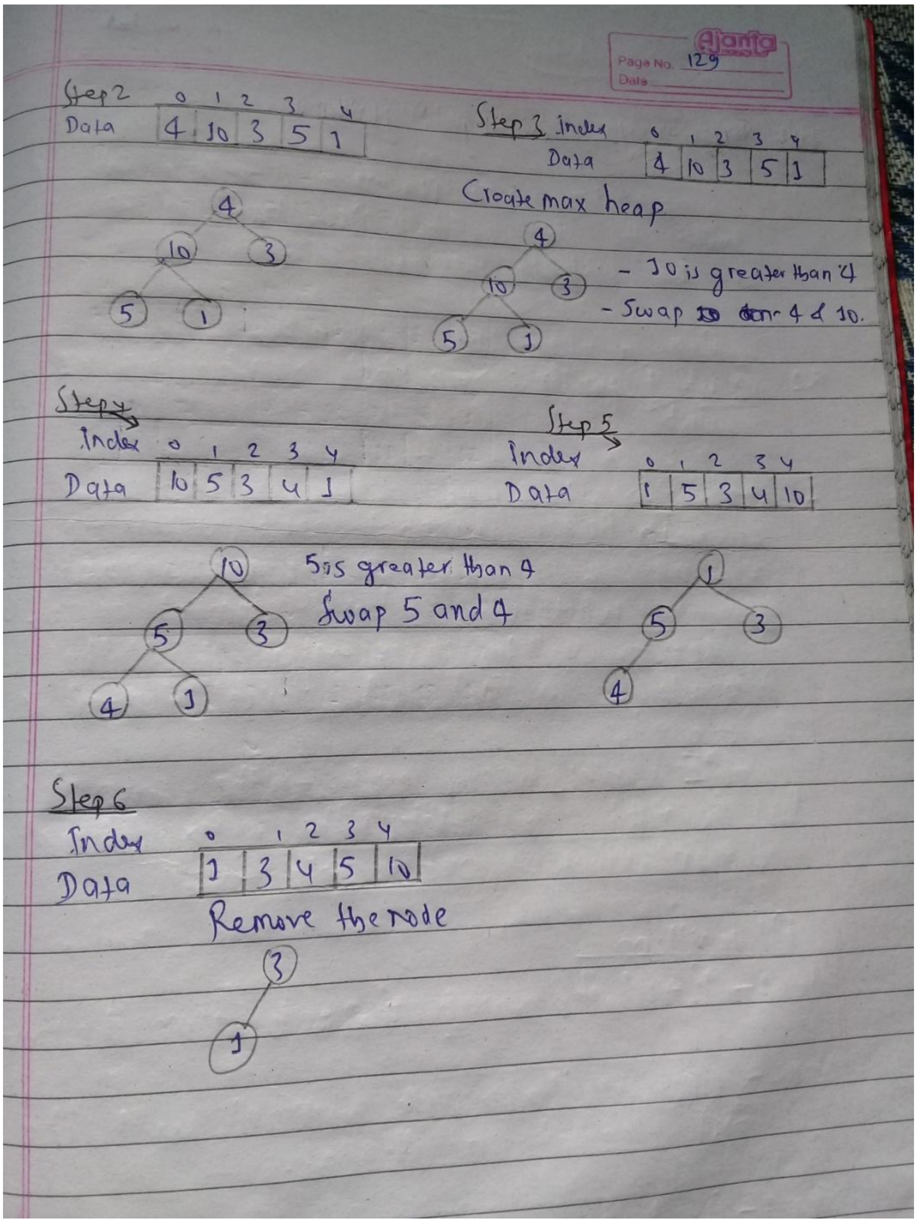
Example

Index	0	1	2	3	4
Input Data	4	10	3	5	1

↑

Build Heap

4



8.6 Efficiency of Sorting Algorithms

Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large chunk of data (for instance, a struct containing a name and address) based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key.

Sorting algorithms are usually judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Most of the algorithms in use have an algorithmic efficiency of either O(n^2) or O(n*log(n)). A few special case algorithms can sort certain data sets faster than O(n*log(n)). These algorithms are not based on comparing the items being sorted and rely on tricks. It has been shown that no key-comparison algorithm can perform better than O(n*log(n)).

Many algorithms that have the same efficiency do not have the same speed on the same input. First, algorithms must be judged based on their average case, best case, and worst case efficiency. Some algorithms, such as quick sort, perform exceptionally well for some inputs, but horribly for others. Other algorithms, such as merge sort, are unaffected by the order of input data. Even a modified version of bubble sort can finish in O(n) for the most favorable inputs.

A second factor is the "constant term". As big-O notation abstracts away many of the details of a process, it is quite useful for looking at the big picture. But one thing that gets dropped out is the constant in front of the expression: for instance, O(c*n) is just O(n). In the real world, the constant, c, will vary across different algorithms. A well-implemented quick sort should have a much smaller constant multiplier than heap sort.

A second criterion for judging algorithms is their space requirement -- do they require scratch space or can the array be sorted in place (without additional memory beyond a few variables)? Some algorithms never require extra space, whereas some are most easily understood when implemented with extra space (heap sort, for instance, can be done in place, but conceptually it is much easier to think of a separate heap). Space requirements may even depend on the data structure used (merge sort on arrays versus merge sort on linked lists, for instance).

A third criterion is stability -- does the sort preserve the order of keys with equal values? Most simple sorts do just this, but some sorts, such as heap sort, do not.

The following chart compares sorting algorithms on the various criteria outlined above; the algorithms with higher constant terms appear first, though this is clearly an implementation-dependent concept and should only be taken as a rough guide when picking between sorts of the same big-O efficiency.

	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	Stops after reaching a sorted array
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.