Searching and Sorting

Linear Search,

Binary Search,

Selection Sort,

Merge sort,

Insertion sort,

Introduction to Hashing

Linear Search :

What Is a Linear Search Algorithm?

Linear search, often known as sequential search, is the most basic search technique. In this type of search, you go through the entire list and try to fetch a match for a single element. If you find a match, then the address of the matching target element is returned.

On the other hand, if the element is not found, then it returns a NULL value.

Following is a step-by-step approach employed to perform Linear Search Algorithm.

The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

Time Complexity

Best Case Complexity

•The element being searched could be found in the first position.
•In this case, the search ends with a single successful comparison.
•Thus, in the best-case scenario, the linear search algorithm performs O(1) operations.

Worst Case Complexity

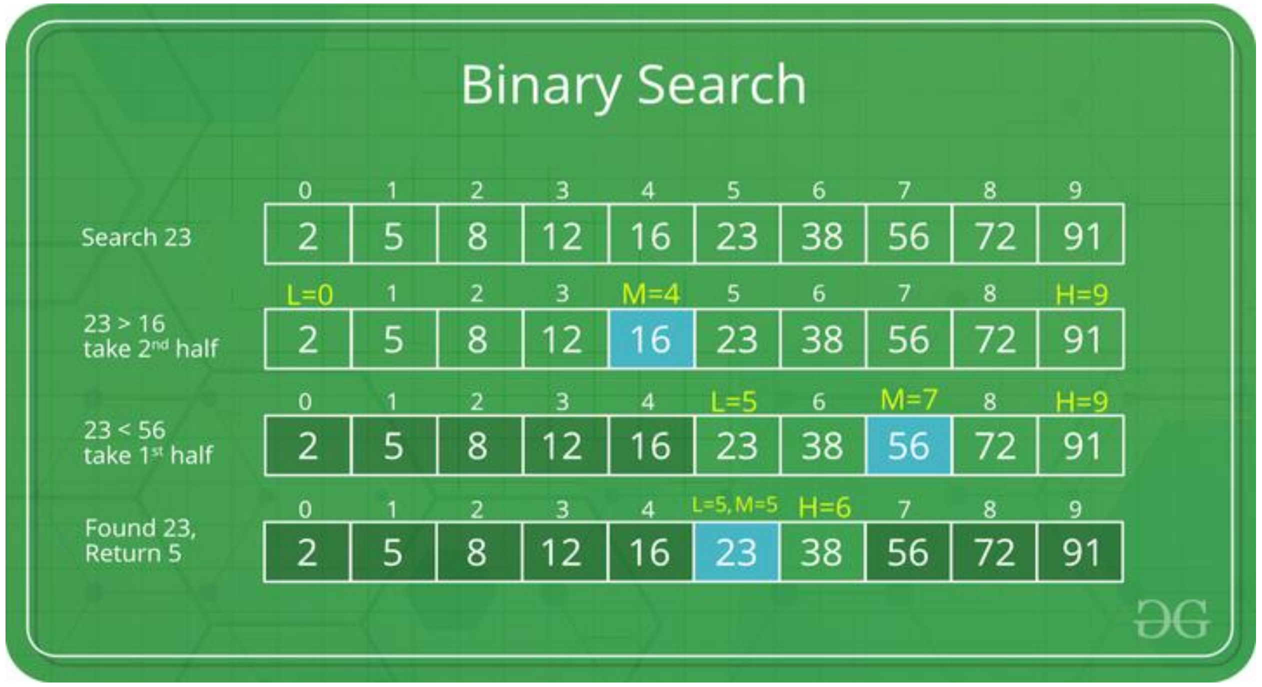
•The element being searched may be at the last position in the array or not at all.
•In the first case, the search succeeds in ‘n’ comparisons.
•In the next case, the search fails after ‘n’ comparisons.
•Thus, in the worst-case scenario, the linear search algorithm performs O(n) operations.

Average Case Complexity

When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is O(n). The values which are between 2 to n-1

Binary Search

***Binary Search** is defined as a **searching algorithm** used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).*



Repeat until the variables L and M meet each other.

```
M = (L + H)/2

if (x == arr[M]) return mid

else if (x > arr[M]) // x is on the right side

L = M + 1

else H = M - 1 // x is on the left side
```

Applied this algorithm on this example :
Where L = lowest index, H = highest Index and M is middle index.

•**Time Complexity:**

- Best Case: O(1) : if found at first instance of M
- Average Case: O(log N) : between best and worst case
- Worst Case: O(log N) : if found at last instance of M where L and H are meeting

Advantages of Binary Search:

•Binary search is faster than linear search, especially for large arrays.

•More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.

•Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

•The array should be sorted.

•Binary search requires that the data structure being searched be stored in contiguous memory locations.

•Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Applications of Binary Search:

•Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyper parameters for a model.

•It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.

•It can be used for searching a database.

Selection Sort

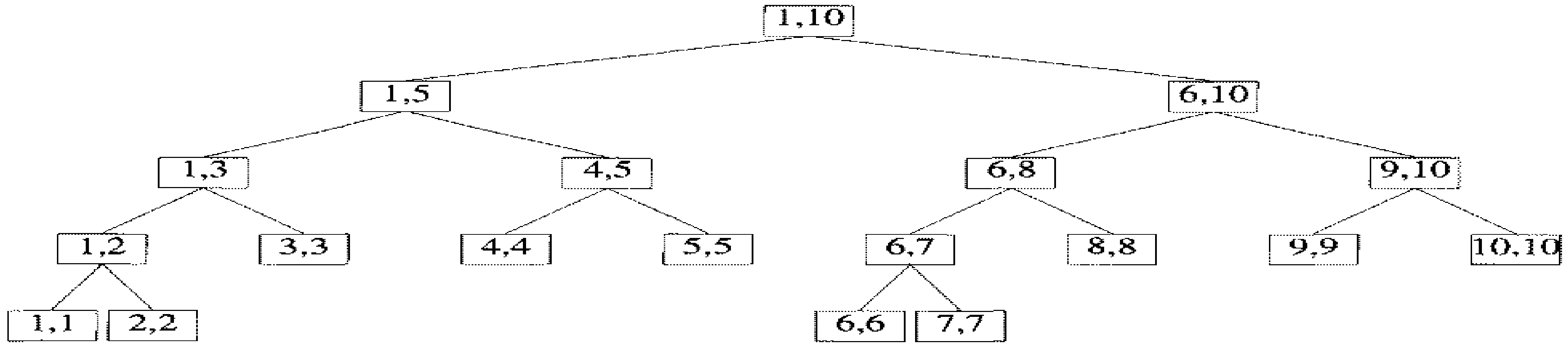
```
for (i = 0; i < n-1; i++)
    {
     int min = i;
        for (j = i+1; j < n; j++)
             {
                   if (arr[j] < arr[min])
                   min = j;
             }
             if(min != i)
             swap(arr[min],arr[i]);
     }
```

Example : Merge sort (Recursive)

```
**Algorithm** MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    **if** ($low < high$) **then**  // If there are more than one element
    {
        // Divide $P$ into subproblems.
          // Find where to split the set.
            $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
          MergeSort($low, mid$);
          MergeSort($mid + 1, high$);
        // Combine the solutions.
          Merge($low, mid, high$);
    }
}
```

Example : 310, 285, 179, 652, 351, 423, 861, 254, 450, 520

Solution : 179, 254, 285, 310, 351, 423, 450, 520, 652, 861



Merge algorithm

Insertion Sort

```
for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
```

Introduction to Hashing

- What is hashing?
- Advantages of hashing.
- Searching by hashing.
- Collision Handling
- Linear Probing: This method involves finding the next available slot in the hash table to place the key that collided with the original key.

SORTING

INTRODUCTION

- The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order. That is, if A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, A[0] < A[1] < A[2] < …… < A[N]
- For example, if we have an array that is declared and initialized as,
- int A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as, A[] = {0, 1, 9, 11, 21, 22, 34}
- A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order (that can either be numerical order, lexicographical order or any user-defined order). Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types of sorting:
- ***Internal sorting*** which deals with sorting the data stored in computer's memory
- ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in computer's memory.

INSERTION SORT

- Insertion sort is a very simple sorting algorithm, in which the sorted array (or list) is built one element at a time.
- Insertion sort works as follows.
- The array of values to be sorted is divided into two sets. One that stores sorted values and the other contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially the element with index 0 (assuming LB, Lower Bound = 0) is in the sorted set, rest all the elements are in the unsorted set
- The first element of the unsorted partition has array index 1 (if LB = 0)
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example: Consider an array of integers given below. Sort the values in the array using insertion sort.

39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36

9	39	45	63	18	81	108	54	72	36
9	39	45	63	18	81	108	54	72	36
9	39	45	63	18	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	54	63	81	108	72	36
9	18	39	45	54	63	72	81	108	36
9	18	36	39	45	54	63	72	81	108

Note : In Pass 1, A[0] is the only element in the sorted set.
In Pass 2, A[1] will be placed either before or after A[0], so that the array A is sorted
In Pass 3, A[2] will be placed either before A[0], in-between A[0] and A[1] or after A[1], so that the array is sorted.
In Pass 4, A[4] will be placed in its proper place so that the array A is sorted.

In Pass N, A[N-1] will be placed in its proper place so that the array A is sorted.
Therefore, we conclude to insert the element A[K] is in the sorted list A[0], A[1], …. A[K-1], we need to compare A[K] with A[K-1], then with A[K-2], then with A[K-3] until we meet an element A[J] such that A[J] <= A[K].
In order to insert A[K] in its correct position, we need to move each element A[K-1], A[K-2], …., A[J] by one position and then A[K] is inserted at the (J+1)th location.

```
Insertion sort (ARR, N) where ARR is an array of N elements

Step 1: Repeat Steps 2 to 5 for K = 1 to N
Step 2:    SET TEMP = ARR[K]
Step 3:    SET J = K - 1
Step 4:    Repeat while TEMP <= ARR[J]
                                 SET ARR[J + 1] = ARR[J]
                                 SET J = J - 1
                      [END OF INNER LOOP]
Step 5:    SET ARR[J + 1] = TEMP
           [END OF LOOP]
Step 6: EXIT
```

Complexity of Insertion Sort Algorithm

For an insertion sort, the best case occurs when the array is already sorted. In this case the running time of the algorithm has a linear running time (i.e., O(*n*)). This is because, during each iteration, the first element from unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(*n*2)).

Even in the average case, the insertion sort algorithm will have to make at least (K-1)/2 comparisons. Thus, the average case also has a quadratic running time.

SELECTION SORT

- Consider an array ARR with N elements. The selection sort takes N-1 passes to sort the entire array and works as follows. First find the smallest value in the array and place it in the first position. Then find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,
- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, A[0] and A[1] is sorted
- In pass 3, find the position POS of the smallest value in sub-array of N-2 elements. Swap ARR[POS] with ARR[2]. Now, ARR[0], ARR[1] and ARR[2] is sorted
-

  In pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1}. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], … , ARR[N-1] is sorted.

Example: Sort the array given below using selection sort

39	9	81	45	90	27	72	18

PASS	LOC	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90

```
SMALLEST (ARR, K, N, POS)

Step 1: [Initialize] SET SMALL = ARR[K]
Step 2: [Initialize] SET POS = K
Step 3: Repeat for J = K+1 to N
                    IF SMALL > ARR[J], then
                              SET SMALL = ARR[J]
                              SET POS = J
                    [END OF IF]
          [END OF LOOP]
Step 4: Exit
```

```
Selection Sort to sort an array ARR with N elements

Step 1: Repeat Steps 2 and 3 for K =1 to N-1
Step 2:    CALL SMALLEST(ARR, K, N, POS)
Step 3:    SWAP A[K] with ARR[POS]
           [END OF LOOP]
Step 4: Exit
```

Complexity of Selection Sort Algorithm

Selection sort is a sorting algorithm that is independent of the original order of the elements in the array. In pass 1, selecting the element with smallest value calls for scanning all n elements; thus, n-1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In pass 2, selecting the second smallest value requires scanning the remaining $n - 1$ elements and so on. Therefore,

$(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1) / 2 = \Theta(n2)$ comparisons

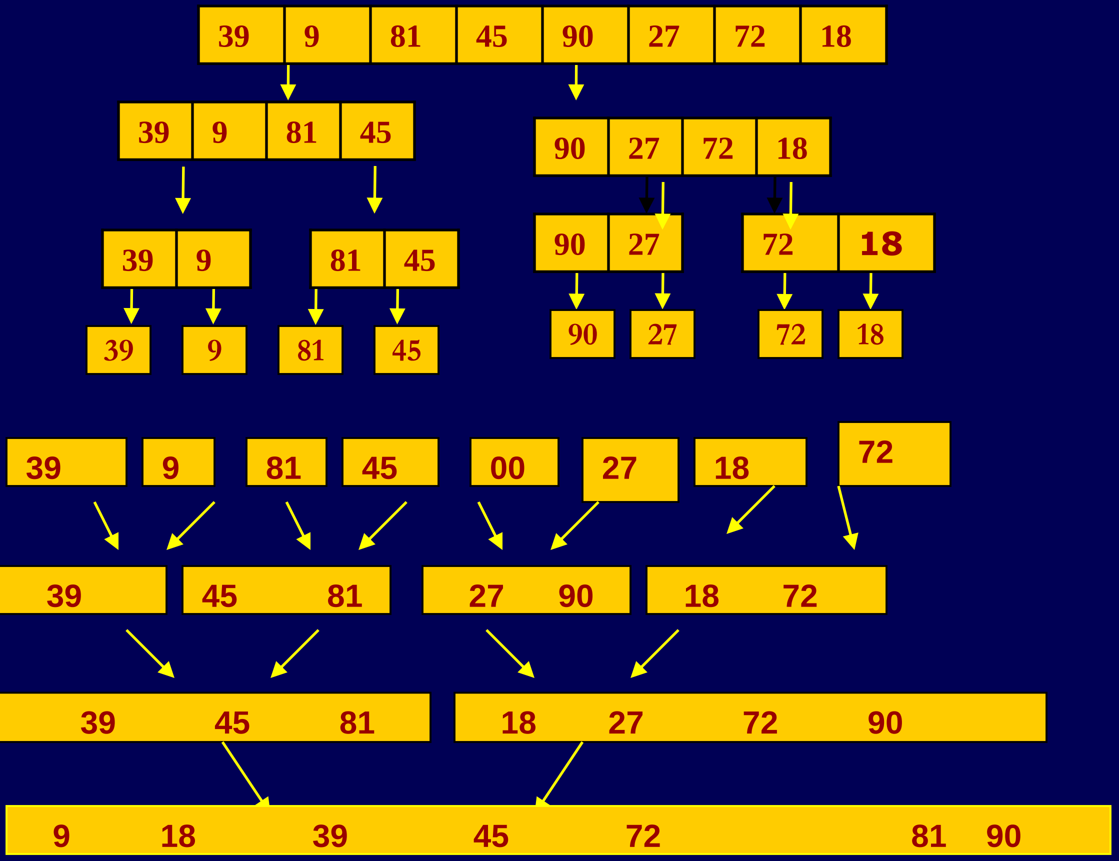
MERGE SORT

- Merge sort is a sorting algorithm that uses the divide, conquer and combine algorithmic paradigm. Where,
- ***Divide*** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements in each sub-array. (If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A).
- ***Conquer*** means sorting the two sub-arrays recursively using merge sort.
- ***Combine*** means merging the two sorted sub-arrays of size n/2 each to produce the sorted array of n elements.
- Merge sort algorithms focuses on two main concepts to improve its performance (running time):
- A smaller list takes few steps and thus less time to sort than a large list.
- Less steps, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.
- The basic steps of a merge sort algorithm are as follows:
- If the array is of length 0 or 1, then it is already sorted. Otherwise:
- (Conceptually) divide the unsorted array into two sub- arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array
- Merge the two sub-arrays to form a single sorted list

Example: Sort the array given below using merge sort

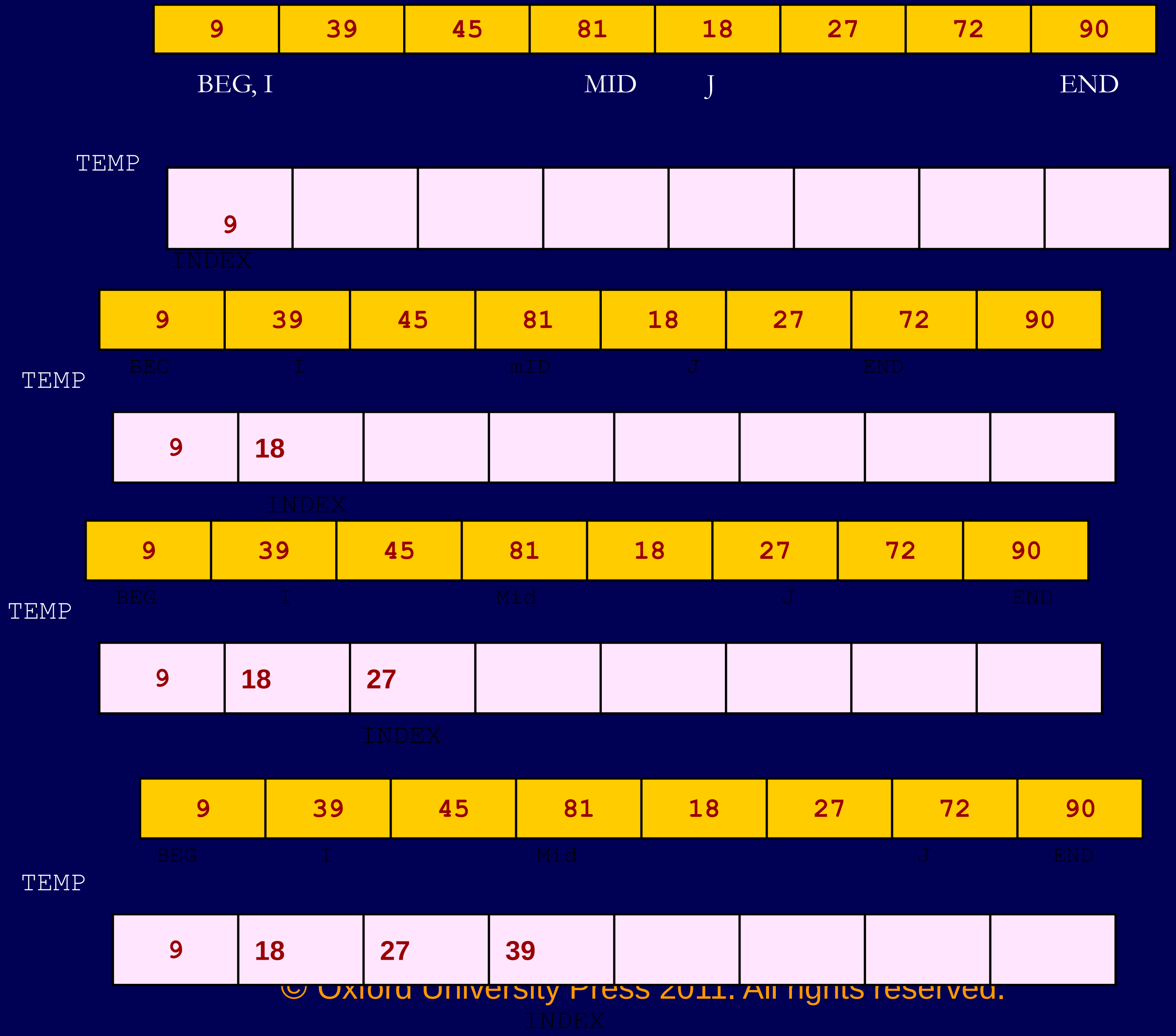
9	9	81	45	90	27	72	18

Divide and conquer the array



Combine the elements to form a sorted array

To understand the merge algorithm, consider figure 14.4 which shows how we merge two lists to form one list. For the sake of understanding we have taken two sub-lists each containing four elements. The same concept can be utilized to merge 4 sub-lists containing two elements, and eight sub-lists having just one element.



9	39	45	81	18	27	72	90
BEG		I	Mid			J	END

9	18	27	39	45			
				INDEX			

9	39	45	81	18	27	72	90
BEG			I,Mid			J	END

9	18	27	39	45	72		
					INDEX		

When I is greater than MID copy the remaining elements of the right sub-array in TEMP

9	18	27	39	45	72	72	81	90
								INDEX

```
MERGE (ARR, BEG, MID, END)

Step 1: [Initialize] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
     IF ARR[I] < ARR[J], then
                              SET TEMP[INDEX] = ARR[I]
          SET I = I + 1
                    ELSE
                              SET TEMP[INDEX] = ARR[J]
                              SET J = J + 1
                    [END OF IF]
          SET INDEX = INDEX + 1
          [END OF LOOP]
Step 3: [ Copy the remaining elements of right sub-array, if any] IF I > MID, then
                    Repeat while J <= END
                              SET TEMP[INDEX] = ARR[J]
                              SET INDEX = INDEX + 1, SET J = J + 1
                    [END OF LOOP]
          [Copy the remaining elements of left sub-array, if any] Else
                    Repeat while I <= MID
                              SET TEMP[INDEX] = ARR[I]
                              SET INDEX = INDEX + 1, SET I = I + 1
                    [END OF LOOP]
          [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
                    a. SET ARR[K] = TEMP[K]
                    b. SET K = K + 1
          [END OF LOOP]
Step 6: END
```

```
MERGE_SORT( ARR, BEG, END)

Step 1: IF BEG < END, then
                    SET MID = (BEG + END)/2
                    CALL MERGE_SORT( ARR, BEG, MID)
                    CALL MERGE_SORT (ARR, MID + 1, END)
                    MERGE (ARR, BEG, MID, END)
          [END OF IF]
Step 2: END
```

Complexity of Merge Sort Algorithm

The running time of the merge sort algorithm in average case and worst case can be given as O(n logn). Although algorithm merge sort has an optimal time complexity but a major drawback of this algorithm is that it needs an additional space of O(*n*) for the temporary array TEMP

HASHING AND COLLISION

INTRODUCTION

- In chapter 12, we have studied about two search algorithms- linear search and binary search. While linear search algorithm has running time proportional to O(n), binary search takes time proportional to O(log n), where n is the number of elements in the array.
- is there any way in which searching can be done in constant time, irrespective of the number of elements in the array?
- There are two solutions to this problem. To analyze the first solution let us take an example. In a small company of 100 employees, each employee is assigned an Emp_ID number in the range 0 – 99. To store the employee's records in an array, each employee's Emp_ID number act as an index in to the array where this employee's record will be stored as shown in figure

KEY	ARRAY OF EMPLOYEE'S RECORD
Key 0 [0]	Record of employee having Emp_ID 0
Key 1 [1]	Record of employee having Emp_ID 1
Key 2 [2]	Record of employee having Emp_ID 2
……………………………	……………………………………………
……………………………	……………………………………………..
Key 98 [98]	Record of employee having Emp_ID 98
Key 99 [99]	Record of employee having Emp_ID 99

- In this case we can directly access the record of any employee, once we know his Emp_ID, because array index is same as that of Emp_ID number. But practically, this implementation is hardly feasible.
- Let us assume that the same company use a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we will need an array of size 100,000, of which only 100 elements will be used. This is illustrated in figure

KEY		ARRAY OF EMPLOYEE'S RECORD
Key 00000	[0]	Record of employee having Emp_ID 00000
…………………………		………………………………………….
Key n	[n]	Record of employee having Emp_ID n
…………………………		…………………………………………..
Key 99998	[99998]	Record of employee having Emp_ID 99998
Key 99999	[99999]	Record of employee having Emp_ID 99999

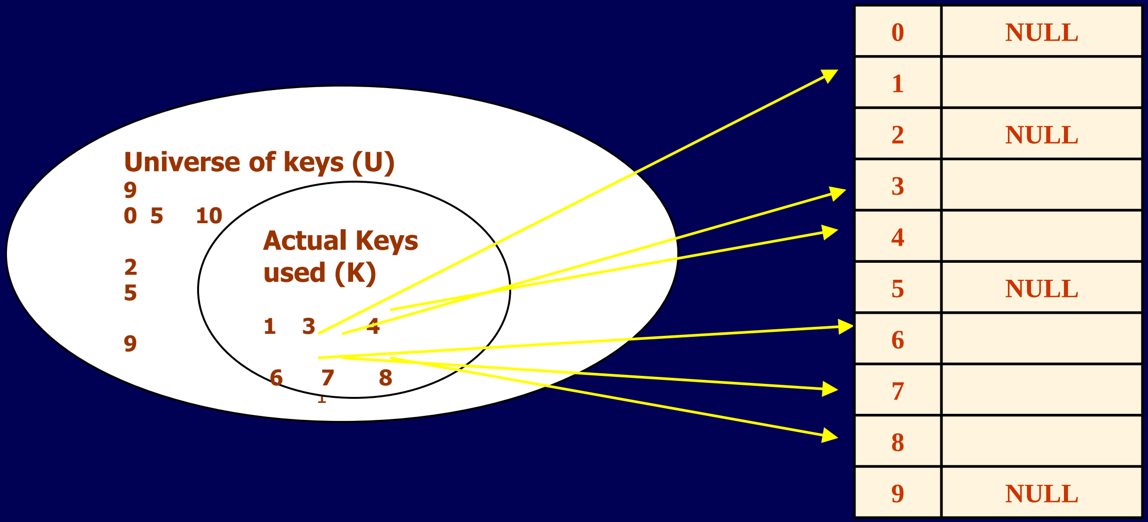
It is impractical it is to waste that much storage just to ensure that each employee' record is in a unique and predictable location.

Whether we use a two digit primary key (Emp_ID) or a five digit key, there are just 100 employees in the company. Thus, we will be using only 100 locations in the array. Therefore, in order to keep the array size down to the size that we will actually be using (100 elements), another good option is to use just the last two digits of key to identify each employee. For example, the employee with Emp_ID number 79439 will be stored in the element of the array with index 39. Similarly, employee with Emp_ID 12345 will have its record stored in the array at the 45th location.

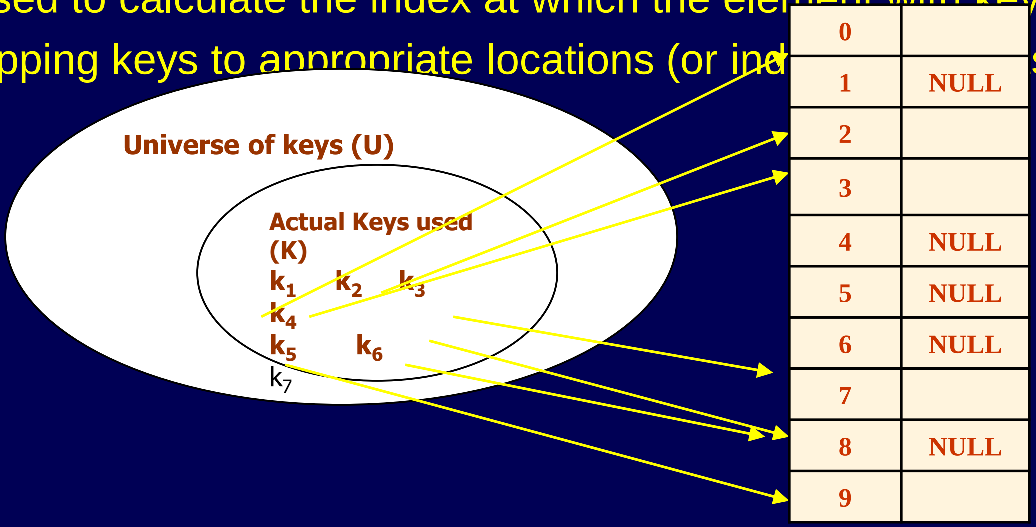
So, in the second solution we see that the elements are not stored according to the *value* of the **key.** So in this situation, we need a way to convert a five-digit key number to two-digit array index. We need some *function* that will do the transformation. In this case, we will use the term Hash Table for an array and the function that will carry out the transformation will be called a Hash Function.

HASH TABLE

- Hash Table is a data structure in which keys are mapped to array positions by a hash function.
- A value stored in the Hash Table can be searched in O(1) time using a hash function to generate an address from the key (by producing the index of the array where the value is stored).
- Look at the figure which shows a direct correspondence between the key and the index of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.



- However, when the set K of keys that are actually used is much smaller than that of U, a hash table consumes much less storage space. The storage requirement for a hash table is just O(k), where k is the number of keys actually used.
- In a hash table, an element with key k is stored at index h(k) not k. This means, a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or ind sh table is called ***hashing***.



That is, when two or more keys maps to the same memory location, a collision is said to occur.

HASH FUNCTION

- Hash Function, h is simply a mathematical formula which when applied to the key, produces an integer which can be used as an index for the key in the hash table. The main aim of a hash function is that elements should be relatively randomly and uniformly distributed. Hash function produces a unique set of integers within some suitable range. Such function produces no collisions. But practically speaking, there is no hash function that eliminates collision completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Division Method

- Division method is the most simple method of hashing an integer *x.* The method divides *x* by *M* and then use the remainder thus obtained. In this case, the hash function can be given as

$$h(x) = x \bmod M$$

- The division method is quite good for just about any value of *M* and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for *M*.
- For example, *M* is an even number, then *h*(*x*) is even if *x* is even; and *h*(*x*) is odd if *x* is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.
- Generally, it is best to choose *M* to be a prime number because making *M* a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values. Then M should also be not too close to exact powers of 2. if we have,

  $h(k) = x \bmod 2k$

- then the function will simply extract the lowest *k* bits of the binary representation of *x*

COLLISIONS

- Collision occurs when the hash function maps two different keys to same location. Obviously, two records can not be stored in the same location. Therefore, a method used to solve the problem of collision also called collision resolution technique is applied. The two most popular method of resolving collision are:
- Collision resolution by open addressing
- Collision resolution by chaining

- **Collision Resolution by Open Addressing**
- Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values- either sentinel value (for example, -1) or a data value. The presence of sentinel value indicates that the location contains no data value at present but can be used to hold a value.
- The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using- linear probing, quadratic probing and double hashing. We will discuss all these techniques in this section.

- ***Linear Probing***
- The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at location generated by *h(k),* then the following hash function is used to resolve the collision.
- $$h(k, i) = [h'(k) + i] \bmod m$$
- where, *m* is the size of the hash table, *h'(k) = k mod m* and *i* is the probe number and varies from *0* to *m-1*.

Example: Consider a hash table with size = 10. Using linear probing insert the keys 72, 27, 36, 24, 63, 81 and 92 into the table.

Let h'(k) = k mod m, m = 10

Initially the hash table can be given as,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step1: Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10
= (2) mod 10
= 2

Since, T[2] is vacant, insert key 72 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step2: Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10
= (7) mod 10
= 7

Since, T[7] is vacant, insert key 27 at this location

Step3: Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6

Since, T[6] is vacant, insert key 36 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step4: Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since, T[4] is vacant, insert key 24 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since, T[3] is vacant, insert key 63 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step6: Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since, T[1] is vacant, insert key 81 at this location

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for next location. Thus probe, i = 1, this time. Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (2 + 1) mod 10

= 3