Module 5:

Searching and sorting (5hrs)

CO5:List, explain and **examine** the concepts of sorting, searching techniques in real life problem solving

Subject In-charge

Ms. Pratibha Rane

Assistant Professor

Class : III SEM SEIT- B

Refer chapter 14: Reema Thareja

- Searching: Sequential Search, Binary Search.
- Hashing: Hash Functions: Truncation, Mid-square Method, Folding Method, Division Method.
- Collision Resolution:
- Open Addressing: Linear Probing, Quadratic Probing, Double Hashing,
- Closed addressing : Separate Chaining Bucket Hashing.
- Analysis of all searching techniques
- Sorting:

Insertion sort,
Selection sort,
Merge sort,
Quick sort and
Radix sort.

- Self-learning Topics: Implementation of different sorting techniques and searching.

Linear search :

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                  SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

value to be searched is VAL = 7

Linear search :

```
LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:            IF A[I] = VAL
                         SET POS = I
                         PRINT POS
                         Go to Step 6
                   [END OF IF]
                    SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

Analysis of Linear search

It executes in **O(n) time** where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array.

both the cases, n comparisons will have to be made.

However, the performance of the linear search algorithm can be improved by using a sorted array

binary search : to improve searching using sorted arrays

- int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; and the value to be searched is VAL = 9.

- The algorithm will proceed in the following manner.

- BEG = 0, END = 10, MID = (0 + 10)/2 = 5

- Now, VAL = 9 and A[MID] = A[5] = 5
- A[5] is less than VAL, therefore, we now search for the value in the second half of the array.

- So, we change the values of BEG and MID.
- Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8
- VAL = 9 and A[MID] = A[8] = 8
- A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.

- So, again we change the values of BEG and MID.
- Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
- Now, VAL = 9 and A[MID] = 9.

binary search algorithm:

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:         SET MID = (BEG + END)/2
Step 4:         IF A[MID] = VAL
                     SET POS = MID
                     PRINT POS
                     Go to Step 6
                ELSE IF A[MID] > VAL
                     SET END = MID - 1
                ELSE
                     SET BEG = MID + 1
                [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.2 Algorithm for binary search

The complexity in terms of f(n), where n is the number of elements in the array.

The complexity of the algorithm is calculated depending on the number of comparisons that are made.

With each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$$2^{f(n)} > n \text{ or } f(n) = \log_2 n$$

binary search algorithm:

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
           PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.2 Algorithm for binary search

Find position of the element 88 using binary search method in an array given below:
A= { 77,33, 44, 11, 88, 22, 66, 55}

	beg			med.				end
element	11	22	33	44	55	66	77	88
Position	0	1	2	3	4	5	6	7

beg = 0
end = 7
med = (0 + 7)/2 = 3
∵ (beg + end)/2
Check if A[med] == val
i.e 44 == 88

beg = med + 1 = 3 + 1 = 4
end = 7
med = (beg + end)/2
= (4 + 7)/2
= 5

med = (beg + end)/2
= (6 + 7)/2
= 6

med = (7 + 7)/2
= 7

69, 88, 19, 58, 46, 12, 16, 4, 67

Search 67

Hashing :

What is hashing ?

It is a Process of indexing and retrieving element or data in a data structure to provide faster way of finding the element using the hash key

What is hashing function?

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.

What is hash key?

Hash(key) is the value which provides the index value where the actual data is likely to store in the data structure.

Hashing :

What is a hash table?

An array which maps a key into the data structure with the help of the hash functions such that insertion, deletion, search operations can be performed with constant time complexity

What are the applications of hash tables?

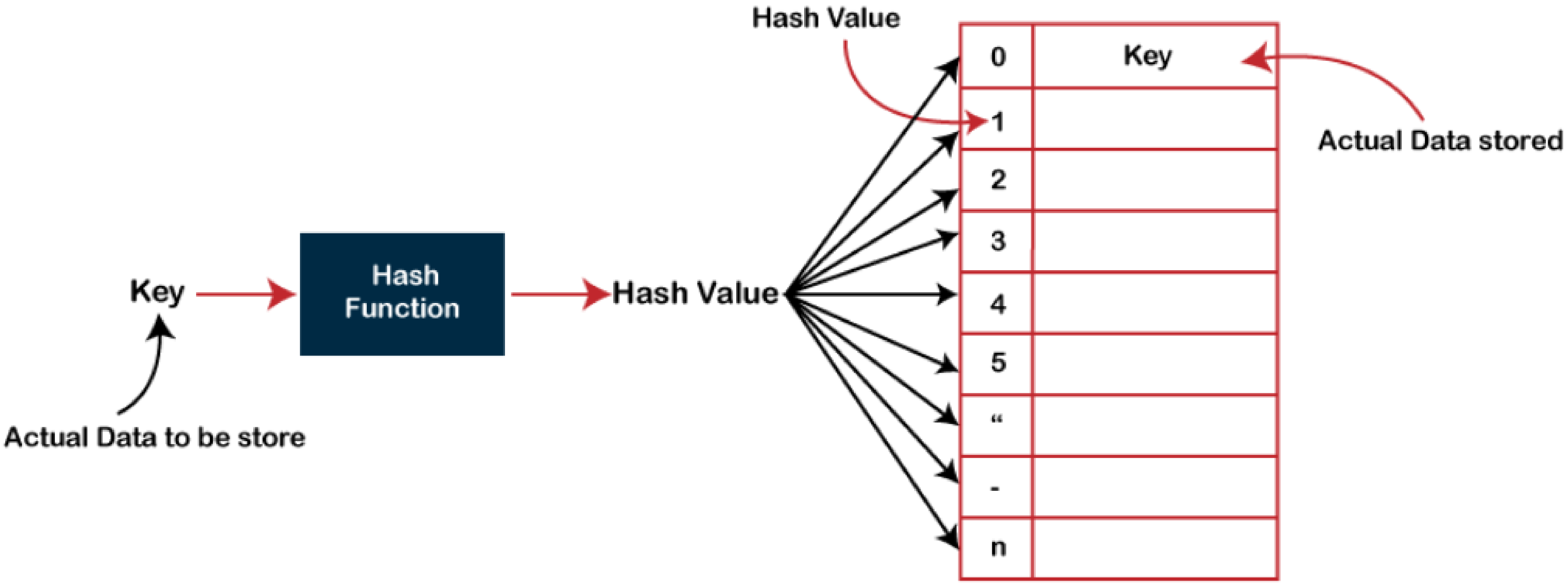
In database systems : random access to data in constant amount of time
For symbol tables : compilers uses has table to stores symbols
Used in data dictionary: for other data structures
Network processing algorithms : route lookup, packet classification, network monitoring

What is bucket?

It is used by hash tables to store entries (key, value) pairs in buckets

Hashing :

Index = hash(key)



Hashing :

There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

Hashing :

The values returned by hash functions can be called as hash code, hash sums or hash values

Mid square method : the key value is squared in this method and the middle part of the result is used as the index into the table

Eg key = 2199
Then square of the key is =4835601
thus the new index value where the key will be stored is the middle part of 4835601 i.e. 356

Advantage :
Not dominating the by the distribution of bottom or top digit
Disadvantage : calculation of the middle part

Hashing :

The values returned by hash functions can be called as hash code, hash sums or hash values

Folding method: partition the key into several pieces and then combine it in some or the other way

Eg – key = 356942781

P1 – 356

P2- 942

P3 – 781

Adding these values will yield 2079 thus new index =079

Hashing :

The values returned by hash functions can be called as hash code, hash sums or hash values

Division method:

Dividing the key with some value and use the remainder as an index

This gives the index in the range between 0 to m-1 so the hash table should be of size m.

Index := key MOD table_size
Eg Key Is 102 and table size is 10

Then index := 102 MOD 10
Index = 2

Collision :

When two different keys produce the same address in the hash table.

Eg. Stores key values in hash table of size 11.
23, 18, 29, 28, 39, 13, 16, 42, 17

Use
Index := key MOD 11

Thus
Index := 23 MOD 11

Index := 1

Key	Location
23	1
18	7
29	7
28	6
39	6
13	2
16	5
42	9
17	6

How to resolve Collision :

Collision resolution

-If element to be inserted is mapped to the same location, where an element is already Inserted then we have a collision and must Be resolved.

Open addressing – (closed hashing)

1. Linear probing
2. Quadratic probing
3. Double hashing

Closed addressing – open hashing

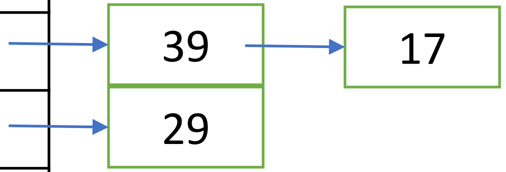
1. separate chaining

Key	Location
23	1
18	7
29	7
28	6
39	6
13	2
16	5
42	9
17	6

Collision Resolution Strategies

Separate chaining method

0	
1	23
2	13
3	
4	
5	16
6	28
7	18
8	
9	42
10	



- Closed address means separate chaining method
- Create link list nodes to store new key

Key	Location
23	1
18	7
29	7
28	6
39	6
13	2
16	5
42	9
17	6

Collision Resolution Strategy:
Use divisive method and open addressing : linear probing

0	17
1	23
2	13
3	
4	
5	16
6	28
7	18
8	29
9	39
10	42

Open addressing: by default use linear probing if not mentioned
(u+i) % m

Key	Location	Probes
23	1	1
18	7	1
29	7	2
28	6	1
39	6	4
13	2	1
16	5	1
42	9	2
17	6	6

Simple process but leads to some clustering when keys are computed to closer values

Collision Resolution-Use divisive method and double hashing(Two hashing Function)

0	
1	23
2	18
3	
4	
5	
6	29
7	
8	
9	
10	

Can be given as
h(k,i) = [h1(k) + i * h2(k)] mod m
where m is size of the table

h1(k) and h2(k) are two hash functions
Where
h1(k) = k mod m
And
h2(k) = k mod m'

Key	Location (u)	Probes
23	1	
18	7	
29	7	
28	6	
39	6	
13	2	
16	5	
42	9	
17	6	

Collision Resolution-Use divisive method and double hashing(Two hashing Function)

0	-1
1	23
2	-1
3	-1
4	-1
5	-1
6	29
7	18
8	-1
9	-1
10	-1

Input: 23, 18, 29, 28, 39, 13, 16, 42, 17

h1(k) = k mod m

h1(23) = 23 mod 11 = 1
h1(18) = 18 mod 11 = 7
h1(29) = 29 mod 11 = 7 (collision)
Thus use double hashing
[29 mod 11 + (0 * (29 mod 8)] Mod 11 = 7 + 0 = 7
[29 mod 11 + (1 * (29 mod 8)] mod 11
= (7 + 5) mod 11 = 1

[29 mod 11 + (2 * (29 mod 8)] mod 11
= (7 + 10) mod 11 = 6

Key	Location (u)	Probes
23	1	
18	7	
29	7	3
28	6	
39	6	
13	2	
16	5	
42	9	
17	6	

No. of searches

Homework:Complete table with key insertion and Probes

Collision Resolution-Use divisive method and quadratic probing method to store the vaulues

0	-1
1	81
2	72
3	63
4	24
5	101
6	36
7	27
8	-1
9	-1

h(i,k) = [h'(k) + c1* i + c2* i^2] mod m

Example: Insert keys 72, 27, 36, 24 63, 81, and 101 into table of size 10 , take c1= 1 and c2= 3

h(k,i) = [h'k + c1 * i + c2 * i^2] mod m

h(72,0) = [72 mod 10 + 1 * 0 + 3 * 0] mod 10 = 2

h(27, 0) = [27 mod 10 + 1 * 0 + 3 * 0] mod 10 = 7

h(36,0) = 6

h(24, 0) = 4

h(63,0) = 3

h(81,0) = 1

h(101,0) = 1 (collision) thus

h(101,1) = [101 mod 10 + 1 * 1 + 3 * 1] mod 10 = 5

Homework:

1.Insert the keys 7,24,18,52,36,54,11,23 in a chained hash table of 9 memory locations. Use h(k) = k mod m

2.Using double hashing Insert keys 72, 27, 36, 24 63, 81, and 101 into table of size 10 , take c1= 1 and c2= 3 , where h1(k) = k mod 10 and h2(k) = k mod 8

3.Calculate hash values of keys 1892, 1921, 2007 an 3456 using different hashing methods

Sorting techniques:

Insertion sort,

Selection sort,

Merge sort,

Quick sort and

Radix sort.

Insertion Sort:

Places an unsorted element at its suitable place in each iteration.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
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
```

Program

```
for ( I = 1 ; I < n; i++)
{
        temp = a[i];
        j = k-1;
        while(j>=0 && a[j] > temp)
        {
                a[j+1] = a[j];
                J --;
        }
        a[j+1] = temp;
}
```

Avg,Worst case:O(n^2),Best case O(n) as array already sorted

Application case:

1. **array is has a small number of elements**
2. **there are only a few elements left to be sorted**

Sort using Insertion Sort : 5,4,10,1,6,2

Selection sort :It selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

1. Set the first element as minimum
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum. Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.
3. After each iteration, minimum is placed in the front of the unsorted list.
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

Pseudocode:

```
selectionSort(array, size)
repeat (size - 1) times
set the first unsorted element as the minimum
 for each of the unsorted elements
  if element < currentMinimum
      set element as new minimum
swap minimum with first unsorted position
end selectionSort
```

Sort using Selection sort :20,12,10,15,2

20, 12,10, 15,2
↑
min

20, 12,10, 15,2
↑
min

20, 12,10, 15,2
↑
min

20, 12,10, 15,2
↑
min

20, 12,10, 15,2
↑
min

20, 12,10, 15,2
↑
min

Swap 20 and 2

After first iteration: 2, 12,10, 15,20

Example-31,55,27,16,49

Selection sort :Complexity Calculation-

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons: (n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2 nearly equals to n^2. i.e. $O(n^2)$

1. Selection sort is a sorting algorithm that has a quadratic running time complexity of $O(n^2)$, thereby making it inefficient to be used on large lists.
2. performs worse than insertion sort algorithm,
3. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Sort using Selection sort :31,55,27,16,49

Merge Sort:

1. uses the divide, conquer, and combine algorithmic paradigm.

1. **Divide** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. 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.
2. **Conquer** means sorting the two sub-arrays recursively using merge sort.
3. **Combine** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge Sort:

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

1. A smaller list takes fewer steps and thus less time to sort than a large list.
2. As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

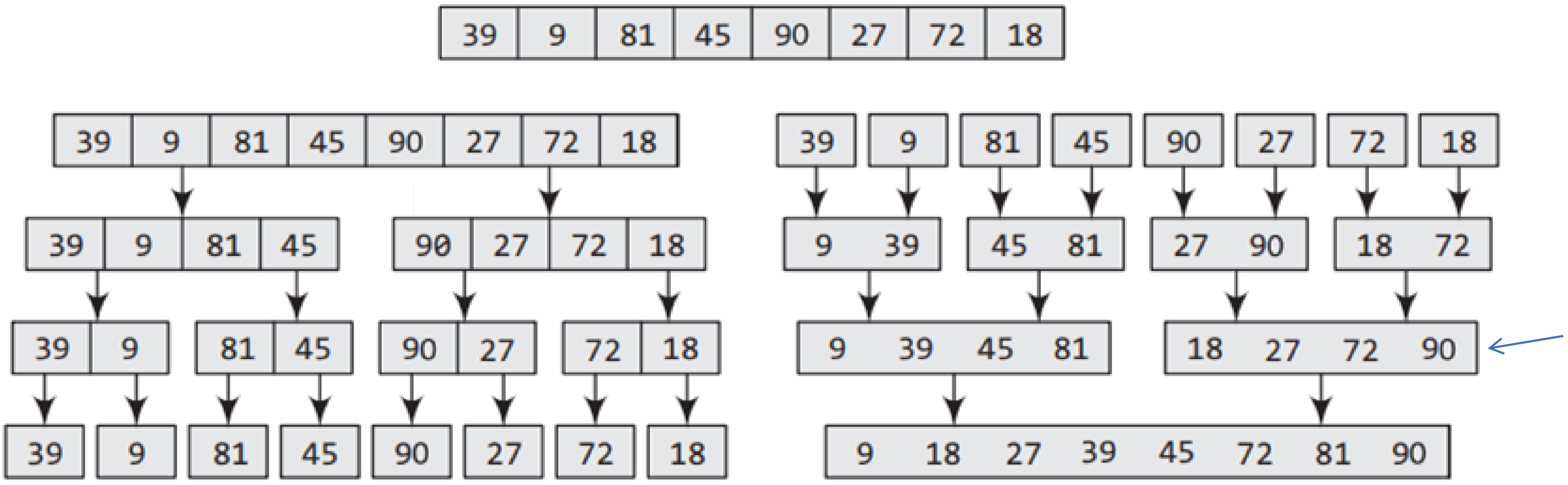
The basic steps of a merge sort algorithm are as follows:

1. If the array is of length 0 or 1, then it is already sorted.
2. Otherwise, divide the unsorted array into two sub-arrays of about half the size.
3. Use merge sort algorithm recursively to sort each sub-array.
4. Merge the two sub-arrays to form a single sorted list.

Merge Sort:

Example 14.5 Sort the array given below using merge sort.

Solution



(Divide and Conquer the array) (Combine the elements to form a sorted array)

Merge Sort:

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

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = BEG
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  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
               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
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            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
```

The running time of merge sort in the average case and the worst case can be given as O(n log n). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Merge Sort:

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

TEMP

9							
INDEX							

Compare ARR[I] and ARR[J], the smaller of th two is placed in TEMP a the location specified by INDEX and subsequentl the value I or J is incremented.

TEMP

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

9	18						
	INDEX						

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

9	18	27					
		INDEX					

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

9	18	27	39				
			INDEX				

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		

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

9	18	27	39	45	72	81	
						INDEX	

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

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

9	18	27	39	45	72	81	90
							INDEX

Quick Sort:

makes O(n log n) comparisons in the average case to sort an array of n elements.

In the worst case, it has a quadratic running time given as $O(n^2)$.

Basically, the quick sort algorithm is faster than other O(n log n) algorithms

Quick sort is also known as partition exchange sort.

Quick Sort:

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way).
3. After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
4. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Quick Sort:

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
               SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
               SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
               SWAP ARR[LOC] with  ARR[RIGHT]
               SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
               Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
               [END OF LOOP]
Step 6:        IF LOC = LEFT
                       SET FLAG = 1
               ELSE IF ARR[LOC] < ARR[LEFT]
                       SWAP ARR[LOC] with  ARR[LEFT]
                       SET LOC = LEFT
               [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
           CALL PARTITION (ARR, BEG, END, LOC)
           CALL QUICKSORT(ARR, BEG, LOC - 1)
           CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Example 14.6 Sort the elements given in the following array using quick sort algorithm

27	10	36	18	25	45

We choose the first element as the pivot. Set `loc` = 0, `left` = 0, and `right` = 5.

27	10	36	18	25	45
loc left					right

Scan from right to left. Since `a[loc]` < `a[right]`, decrease the value of `right`.

27	10	36	18	25	45
loc left				right	

Since `a[loc]` > `a[right]`, interchange the two values and set `loc` = `right`.

25	10	36	18	27	45
left				right loc	

Start scanning from left to right. Since `a[loc]` > `a[left]`, increment the value of `left`.

25	10	36	18	27	45
		left		right loc	

Since `a[loc]` < `a[left]`, interchange the values and set `loc` = `left`.

25	10	27	18	36	45
		left loc		right	

Scan from right to left. Since `a[loc]` < `a[right]`, decrement the value of `right`.

25	10	27	18	36	45
		left loc	right		

Since `a[loc]` > `a[right]`, interchange the two values and set `loc` = `right`.

25	10	18	27	36	45
		left	right loc		

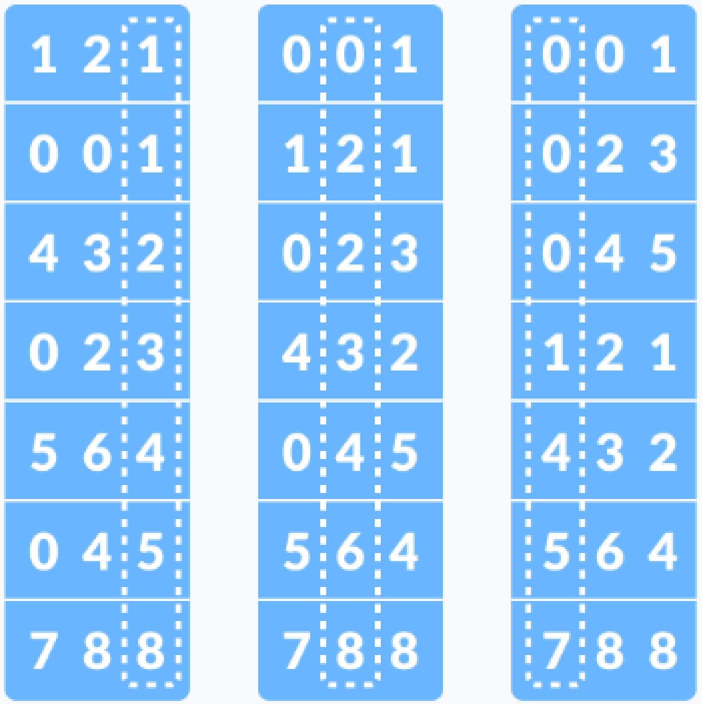
Start scanning from left to right. Since `a[loc]` > `a[left]`, increment the value of `left`.

25	10	18	27	36	45
			right loc left		

pivot

RADIX SORT

- Also knows as bucket sort
- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order



sorting the integers according to units, tens and hundreds place digits

RADIX SORT

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:           SET I = 0 and INITIALIZE buckets
Step 6:           Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT  = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                  [END OF LOOP]
Step 10:          Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Figure 14.11 Algorithm for radix sort

RADIX SORT

Example 14.7 Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Analysis of Radix Sort algorithm

- To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number.
- In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute.
- When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Analysis of Sorting algorithm:

Algorithm	Time Complexity - Best	Time Complexity - Worst	Time Complexity - Average
Bubble Sort	n	n^2	n^2
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Merge Sort	nlog n	nlog n	nlog n
Quicksort	nlog n	n^2	nlog n
Radix Sort	n+k	n+k	n+k
Bucket Sort	n+k	n^2	n
Heap Sort	nlog n	nlog n	nlog n

- Important questions:

1) Explain linear search
2) Explain binary search
3) How does binary search is different from linear search
4) short notes on hashing techniques
5) Describe hash functions
6) Explain bucket hashing
7) What are the methods to resolve collision ?
8) Explain linear probing using an example
9) Explain sorting algorithms(all in the syllabus)
10) Comparisons of all sorting techniques
11) Compare the running time complexity of different sorting algorithms.
12) Implementation of any sorting technique

Thank You