UNIT 3

Searching and Sorting

CONTENTS

PART-1

Searching : Concept of Searching, Sequential Search, IndexSequential Search, Binary Search.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.1. **What do you mean by searching ? Explain.**

Answer

1. Searching is the process of finding the location of given element in the linear array.
2. The search is said to be successful if the given element is found, *i.e.*, the element does exists in the array; otherwise unsuccessful.
3. There are two searching techniques :
   a. Linear search (sequential) b. Binary search
4. The algorithm which we choose depends on organization of the array elements.
5. If the elements are in random order, then we have to use linear search technique, and if the array elements are sorted, then it is preferable to use binary search.

Que 3.2. **Write a short note on sequential search and index sequential search.**

Answer

Sequential search :

1. In sequential (or linear) search, each element of an array is read one-by-one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is found.
2. Linear search is the least efficient search technique among other search techniques.
3. It is used when the records are stored without considering the order or when the storage medium lacks the direct access facility.
4. It is the simplest way for finding an element in a list.
5. It searches the elements sequentially in a list, no matter whether list is sorted or unsorted.
   a. In case of sorted list in ascending order, the search is started from 1st element and continued until desired element is found or the element whose value is greater than the value being searched.

b. In case of sorted list in descending order, the search is started from 1st element and continued until the desired element is found or the element whose value is smaller than the value being searched.
c. If the list is unsorted searching started from 1st location and continued until the element is found or the end of the list is reached.

Index sequential search :

1. In index sequential search, an index file is created, that contains some specific group or division of required record, once an index is obtained, then the partial searching of element is done which is located in a specified group.
2. In indexed sequential search, a sorted index is set aside in addition to the array.
3. Each element in the index points to a block of elements in the array or another expanded index.
4. First the index is searched that guides the search in the array.
5. Indexed sequential search does the indexing multiple times like creating the index of an index.
6. When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Que 3.3. Write down algorithm for linear/sequential search technique. Give its analysis.

Answer

LINEAR(DATA, *N*, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA] Set DATA[$N + 1$] := ITEM
2. [Initialize counter] Set LOC := 1
3. [Search for ITEM]
   Repeat while DATA[LOC] ≠ ITEM
   Set LOC := LOC + 1
   [End of loop]
4. [Successful?] If LOC = $N + 1$, then : Set LOC := 0
5. Exit

Analysis of linear search :

Best case : Element occur at first position. Time complexity is O(1).
Worst case : Element occur at last position. Time complexity is O(n).

Que 3.4. Write down the algorithm of binary search technique.Write down the complexity of algorithm.

Answer

Binary search (*A*, *n*, item, loc)

Let A is an array of 'n' number of items, item is value to be searched.

1. Set : beg = 0, Set : end = $n-1$, Set : mid = (beg + end) / 2
2. While ((beg ≤ end) and (a [mid] != item))
3. If (item < a[mid])
   then Set : end = mid – 1
   else
   Set : beg = mid + 1
   endif
4. Set : mid = (beg + end) / 2
   endwhile
5. If (beg > end) then
   Set : loc = – 1 // element not found
   else
   Set : loc = mid
   endif
6. Exit

Analysis of binary search :

The complexity of binary search is $O(\log_2 n)$.

Que 3.5. What is difference between sequential (linear) searchand binary search technique ?

Answer

S. No.	Sequential (linear) search	Binary search
1.	No elementary condition *i.e.*, array can be sorted or unsorted.	Elementary condition *i.e.*,array should be sorted.
2.	It takes long time to search an element.	It takes less time to search an element.
3.	Complexity is $O(n)$.	Complexity is $O(\log_2 n)$.
4.	It searches data linearly.	It is based on divide and conquer method.

PART-2

Concept of Hashing and Collision Resolution Techniques usedin Hashing.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.6. What do you mean by hashing ?

Answer

1. Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.
2. Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string.
3. In hashing, large keys are converted into small keys by using hash functions.
4. The values are then stored in a data structure called hash table.
5. The task of hashing is to distribute entries (key/value pairs) uniformly across an array.
6. Each element is assigned a key (converted key). By using that key we can access the element in O(1) time.
7. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.
8. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.
9. The element is stored in the hash table where it can be quickly retrieved using hashed key which is defined by

Hash Key = Key Value % Number of Slots in the Table

Que 3.7. Discuss types of hash functions.

Answer

Types of hash functions :

a. Division method :

1. Choose a number m larger than the number n of key in K. (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.)
2. The hash function H is defined by :

$$H(k) = k \pmod m \quad \text{or} \quad H(k) = k \pmod m + 1$$

3. Here $k \pmod m$ denotes the remainder when k is divided by m.
4. The second formula is used when we want the hash addresses to range from 1 to m rather than from 0 to $m - 1$.

b. Midsquare method :

1. The key k is squared.
2. The hash function H is defined by : $H(k) = l$
   where l is obtained by deleting digits from both end of k^2.
3. We emphasize that the same positions of k^2 must be used for all of the keys.

c. Folding method :

1. The key k is partitioned into a number of parts, $k_1, \ldots\ldots, k_r$, where each part, except possibly the last, has the same number of digits as the required address.

2. Then the parts are added together, ignoring the last carry *i.e.*,
$$H(k) = k_1 + k_2 + \ldots\ldots + k_r$$
where the leading-digit carries, if any, are ignored.
4. Now truncate the address upto the digit based on the size of hash table.

Que 3.8. What is collision ? Discuss collision resolution techniques.

OR

Write a short note on hashing techniques.

AKTU 2017-18, Marks 3.5

Answer

Collision :

1. Collision is a situation which occur when we want to add a new record R with key k to our file F, but the memory location address $H(k)$ is already occupied.
2. A collision occurs when more than one keys map to same hash value in the hash table.

Collision resolution technique :

Hashing with open addressing :

1. In open addressing, all elements are stored in the hash table itself.
2. While searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table.
3. Thus, in open addressing, the load factor λ can never exceed 1.
4. The process of examining the locations in the hash table is called probing.
5. Following are techniques of collision resolution by open addressing :

a. Linear probing :

i. Given an ordinary hash function $h' : U\ [0, 1, \ldots\ldots, m-1]$, the method of linear probing uses the hash function.
$$h(k, i) = (h'(k) + i) \bmod m$$
where 'm' is the size of the hash table and $h'(k) = k \bmod m$ (basic hash function).

b. Quadratic probing :

i. Suppose a record R with key k has the address $H(k) = h$ then instead of searching the locations with address h, $h + 1$, $h + 2$, , we linearly search the locations with addresses h, $h + 1$, $h + 4$, $h + 9$, , $h + i^2$.

ii. Quadratic probing uses a hash function of the form
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$
where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots\ldots, m-1$.

c. Double hashing :

i. Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

ii. Double hashing uses a hash function of the form :

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

Hashing with separate chaining :

1. This method maintains the chain of elements which have same hash address.
2. We can take the hash table as an array of pointers.
3. Size of hash table can be number of records.
4. Here each pointer will point to one linked list and the elements which have same hash address will be maintained in the linked list.
5. We can maintain the linked list in sorted order and each elements of linked list will contain the whole record with key.
6. For inserting one element, first we have to get the hash value through hash function which will map in the hash table, then that element will be inserted in the linked list.
7. Searching a key is also same, first we will get the hash key value in hash table through hash function, then we will search the element in corresponding linked list.
8. Deletion of a key contains first search operation then same as delete operation of linked list.

Que 3.9. What do you mean by hashing and collision ? Discuss the advantages and disadvantages of hashing over other searching techniques.

AKTU 2014-15, Marks 10

Answer

Hashing : Refer Q. 3.6, Page 3–4A, Unit-3.

Collision : Refer Q. 3.8, Page 3–6A, Unit-3.

Advantages of hashing over other search techniques :

1. The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more).
2. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.
3. If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures.

Disadvantages of hashing over other search techniques :

1. Hash tables can be more difficult to implement than self-balancing binary search trees. Choosing an effective hash function for a specific application is more an art than a science. In open-addressed hash tables it is fairly easy to create a poor hash function.

2. The cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree.
3. Hash tables are not effective when the number of entries is very small. For certain string processing applications, such as spell-checking, hash tables may be less efficient than trees, finite automata, or arrays.
4. If each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values.

Que 3.10. **Write short notes on garbage collection.**

AKTU 2017-18, Marks 3.5

AKTU 2014-15, Marks 05

Answer

1. When some memory space becomes reusable due to the deletion of a node from a list or due to deletion of entire list from a program then we want the space to be available for future use.
2. One method to do this is to immediately reinsert the space into the free-storage list. This is implemented in the linked list.
3. This method may be too time consuming for the operating system of a computer.
4. In another method, the operating system of a computer may periodically collect all the deleted space onto the free storage list. This type of technique is called garbage collection.
5. Garbage collection usually takes place in two steps. First the computer runs through all lists, tagging those cells which are currently in use and then the computer runs through the memory, collecting all untagged space onto the free storage list.
6. The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free storage list or when the CPU is idle and has time to do the collection.

Que 3.11. Write the conditions when collision occurs in hashing.Describe any collision detection algorithm in brief.

Answer

Condition when collision occurs : Refer Q. 3.8, Page 3–6A, Unit-3.

Collision detection algorithm :

a. One of the collision detection algorithms is grid based algorithm.
b. In this algorithm, grids are space-filling.
c. Each cell or voxel (volume pixel) has a list of objects which intersects it.
d. The uniform grid is used to determine which objects are near to an object by examining object-lists of the cells the object overlaps.
e. Intersections for a given object are found by going through the object lists for all voxels containing the object, performing intersection tests against objects on those lists.

f. A grid based collision detection algorithm then works as follows :

```
1.  for i = 1 to n
2.      v_min = voxel (min (bbox (object (i))))
3.      v_max = voxel (max (bbox (object (i))))
4.      for x = v_min_x to x = v_max_x
5.          for y = v_min_y to y = v_max_y
6.              for z = v_min_z to z = v_max_z
7.                  for j = 1 to n_objects (voxel (x, y, z))
8.                      if (not tested (object (i), object (j)))
9.                          intersect (object (i), object (j))
```

PART-3

Sorting : Insertion Sort, Selection Sort, Bubble Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.12. Write a short note on insertion sort.

AKTU 2014-15, Marks 05

Answer

1. In insertion sort, we pick up a particular value and then insert it at the appropriate place in the sorted sublist, *i.e.*, during k^{th} iteration the element $a[k]$ is inserted in its proper place in the sorted sub-array $a[1], a[2], a[3] \ldots . a[k-1]$.
2. This task is accomplished by comparing $a[k]$ with $a[k-1]$, $a[k-2]$, $a[k-3]$ and so on until the first element $a[j]$ such that $a[j] \leq a[k]$ is found.
3. Then each of the elements $a[k-1]$, $a[k-2]$, $a[j+1]$ are moved one position up and then element $a[k]$ is inserted in $[j+1]^{st}$ position in the array.

Insertion-Sort (*A*)

```
1.  for j ← 2 to length[A]
2.  do key ← A[j]    /*Insert A[j] into the sorted sequence A[1 ..... j – 1].*/
3.  i ← j – 1
4.  while i > 0 and A[i] > key
5.      do A[i + 1] ← A[i]
6.          i ← i – 1
7.  A[i + 1] ← key
```

Analysis of insertion sort :

Complexity of best case is $O(n)$

Complexity of average case is $O(n^2)$

Complexity of worst case is $O(n^2)$

Que 3.13. Write a short note on selection sort.

Answer

1. In selection sort we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.
2. We begin by selecting the largest element and moving it to the highest index position.
3. We can do this by swapping the element at the highest index and the largest element.
4. We then reduce the effective size of the array by one element and repeat the process on the smaller sub-array.
5. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

Selection-Sort (A) :

1. $n \leftarrow$ length[A]
2. for $j \leftarrow 1$ to $n - 1$
3. smallest $\leftarrow j$
4. for $i \leftarrow j + 1$ to n
5. if $A[i] < A$[smallest]
6. then smallest $\leftarrow i$
7. exchange($A[j]$, A[smallest])

Analysis of selection sort : Complexity of best case is $O(n^2)$. Complexity of average case is $O(n^2)$.Complexity of worst case is $O(n^2)$.

Que 3.14. Discuss bubble sort.

Answer

1. Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent element if they are in wrong order.
2. Bubble sort procedure is based on following idea :
   a. Suppose if the array contains n elements, then $(n - 1)$ iterations are required to sort this array.
   b. The set of items in the array are scanned again and again and if any two adjacent items are found to be out of order, they are reversed.
   c. At the end of the first iteration, the lowest value is placed in the first position.
   d. At the end of the second iteration, the next lowest value is placed in the second position and so on.
3. It is very efficient in large sorting jobs. For n data items, this method requires $n(n - 1)/2$ comparisons.

Bubble-sort (A) :

1. for $i \leftarrow 1$ to length [A]
2. for $j \leftarrow$ length[A] down to $i + 1$

3. if $A[j] < A[j-1]$
4. exchange($A[j]$, $A[j-1]$)

Analysis of bubble sort : Complexity of best case is $O(n)$. Complexity of worst case is $O(n^2)$. Complexity of average case is $O(n^2)$.

PART-4

Quick Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.15. Explain and give quick sort algorithm. Determine its complexity.

Answer

1. Quick sort is a sorting algorithm that also uses the idea of divide and conquer.
2. This algorithm finds the elements, called pivot, that partitions the array into two halves in such a way that the elements in the left sub-array are less than and the elements in the right sub-array are greater than the partitioning element.
3. Then these two sub-arrays are sorted separately. This procedure is recursive in nature with the base criteria.

Algorithm :

QUICK (*A, N,* BEG, END, LOC) :

1. [Initialize] Set LEFT := BEG, RIGHT := END and LOC := BEG
2. [Scan from RIGHT to LEFT]
   a. Repeat while *A* [LOC] ≤ *A* [RIGHT] and LOC ≠ RIGHT
      RIGHT := RIGHT – 1
      [End of Loop]
   b. If LOC = RIGHT, then : Return
   c. If *A* [LOC] > *A* [RIGHT], then :
      i. [Interchange *A* [LOC] and *A* [RIGHT]]
      TEMP := *A* [LOC], *A* [LOC] := *A* [RIGHT],
      A [RIGHT] = TEMP
      ii. Set LOC := RIGHT
      iii. Go to step 3
      [End of if structure]
3. [Scan from LEFT to RIGHT]
   a. Repeat while *A* [LEFT] ≤ *A* [LOC] and LEFT ≠ LOC :

LEFT := LEFT + 1
[End of Loop]

b. If LOC = LEFT, then : Return,

c. If A [LEFT] > A [LOC], then

i. [Interchange A [LEFT] and A [LOC]]
TEMP := A [LOC], A [LOC] := A [LEFT],
A [LEFT] := TEMP

ii. Set LOC := LEFT

iii. Go to step 2 [End of if structure]

Quick sort : This algorithm sorts an array A with N elements.

1. [Initialize] Top := NULL
2. [PUSH boundary values of A onto stack when A has 2 or more elements]
   If N > 1, then : TOP := TOP + 1, LOWER [1] := 1, UPPER [1] := N
3. Repeat steps 4 to 7 while TOP ≠ NULL
4. [POP sublist from stacks]
   Set BEG := LOWER [TOP], END := UPPER [TOP],
   TOP = TOP – 1
5. Call Quick (A, N, BEG, END, LOC)
6. [PUSH left sublist onto stack when it has 2 or more elements]
   If BEG < LOC – 1 then :
   TOP := TOP + 1, LOWER [TOP] := BEG,
   UPPER [TOP] = LOC – 1
   [End of if structure]
7. [PUSH right sublist onto stack when it has 2 or more elements]
   If LOC + 1 < END, then :
   TOP := TOP + 1, LOWER [TOP] := LOC + 1
   UPPER [TOP] := END
   [END of if structure]
   [END of step 3 loop]
8. Exit

Analysis of quick sort :

Complexity of worst case is $O(n^2)$.
Complexity of best case is $O(n \log n)$.
Complexity of average case is $O(n \log n)$.

Que 3.16. Write a recursive quick sort algorithm.

Answer

QUICK-SORT (A, p, r) :

1. If $p < r$ then
2. $q \leftarrow$ PARTITION (A, p, r)
3. QUICK-SORT (A, p, $q-1$)
4. QUICK-SORT (A, $q+1$, r)

PARTITION (A, p, r) :

1. $x \leftarrow A[r]$
2. $i \leftarrow p-1$

```
3.  for j → p to r − 1
4.      do if A[j] ≤ x
5.          then i → i + 1
6.               exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Que 3.17. What is quick sort ? Sort the given values using quicksort; present all steps/iterations :

38, 81, 22, 48, 13, 69, 93, 14, 45, 58, 79, 72 **AKTU 2016-17, Marks 10**

Answer

Quick sort : Refer Q. 3.15, Page 3–11A, Unit-3.

Numerical : *A* = 38, 81, 22, 48, 13, 69, 93, 14, 45, 58, 79, 72. Choose the pivot element to be the element in position (left + right)/2.

During the partitioning process,

1. Elements strictly to the left of position *lo* are less than or equivalent to the pivot element (69).
2. Elements strictly to the right of position *hi* are greater than the pivot element. When *lo* and *hi* cross, we are done. The final value of *hi* is the position in which the partitioning element ends up.

   Swap pivot element with leftmost element *lo* = left + 1; *hi* = right;

left	left+1										right
38	81	22	48	13	69	93	14	45	58	79	72

Move *hi* left and *lo* right as far as we can; then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.

	lo								*hi*←*hi*←*hi*		
69	81*	22	48	13	38	93	14	45	58*	79*	72*

Repeat above

		lo →	*lo* →	*lo* →	*lo* →	*lo*		*hi*			
69	58	22*	48*	13*	38*	93*	14	45*	81	79	72

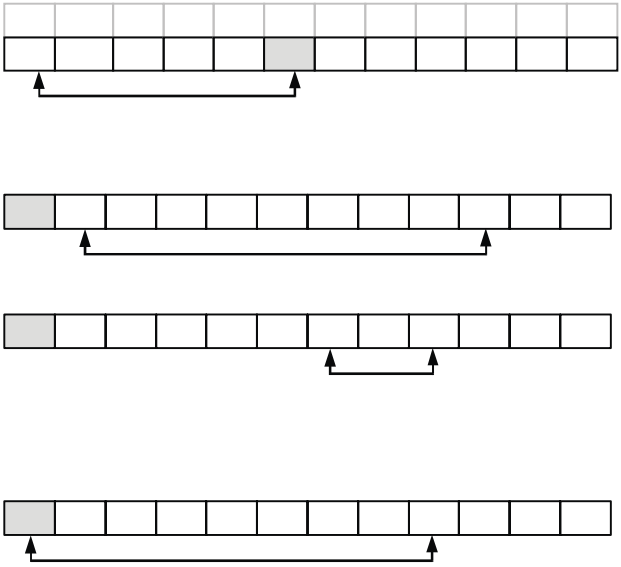
Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].

69	58

l
o
→
81 79 72
o

22 48 13 38
45 14** 93*

Partitioning complete; return value of *hi*.



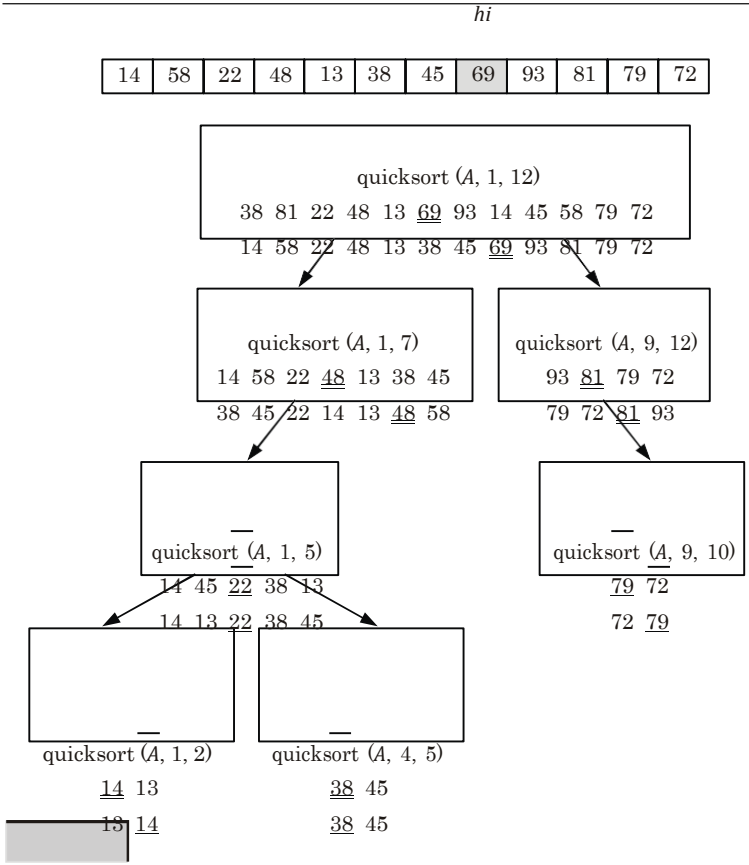


Fig. 3.17.1.

Que 3.18. Write algorithm for quick sort. Trace your algorithm on the following data to sort the list: 2, 13, 4, 21, 7, 56, 51, 85, 59, 1, 9, 10. How the choice of pivot elements affects the efficiency of algorithm.

AKTU 2018-19, Marks 07

Answer

Quick sort algorithm : Refer Q. 3.16, Page 3–12A, Unit-3.

Numerical :

1	2	3	4	5	6	7	8	9	10	11	12
2	13	4	21	7	56	51	85	59	1	9	10

Here $p = 1$, $r = 12$

$x = A[12] = 10$
$i = p - 1 = 0$
$j = 1 + 0 = 1$

Now, $j = 1$ and $i = 0$

$A[1] = 2 \leq 10$ (True)

then $i = 0 + 1 = 1$ and $A[1] \leftrightarrow A[1]$

Now, $j = 2$ and $i = 1$

$A[2] = 13$ and $13 \leq 10$ (False)

So, $j = 3$ $i = 1$

$A[3] = 4$ and $4 \leq 10$ (True)

then, $i = 1 + 1 = 2$ and $A[2] \leftrightarrow A[3]$

	1	2	3	4	5	6	7	8	9	10	11	12
i.e.,	2	4	13	21	7	56	51	85	59	1	9	10

Now, $j = 4$ and $i = 2$

$A[4] = 21$ and $21 \nleq 10$ (False)

$j = 5$ and $i = 2$

$A[5] = 7 \leq 10$ (True)

then, $i = 2 + 1 = 3$ and $A[3] \leftrightarrow A[5]$

	1	2	3	4	5	6	7	8	9	10	11	12
i.e.,	2	4	7	21	13	56	51	85	59	1	9	10

Now, $j = 6$ and $i = 3$

$A[6] = 56$ and $56 \nleq 10$

So, $j = 7$ and $i = 3$

$A[7] = 51$ and $51 \nleq 10$

$j = 8$ and $i = 3$

$A[8] = 85$ and $85 \nleq 10$

$j = 9$ and $i = 3$

$A[9] = 59$ and $59 \nleq 10$

$j = 10$ and $i = 3$

$A[10] = 1 \leq 10$ (True)

then, $i = 3 + 1 = 4$ and $A[4] \leftrightarrow A[10]$

	1	2	3	4	5	6	7	8	9	10	11	12
i.e.,	2	4	7	1	13	56	51	85	59	21	9	10

$j = 11$ and $i = 4$

$A[11] = 9 \leq 10$ (True)

$i = 4 + 1 = 5$ and $A[5] \leftrightarrow A[11]$

	1	2	3	4	5	6	7	8	9	10	11	12
i.e.,	2	4	7	1	9	56	51	85	59	21	13	10

$A[6] \leftrightarrow A[12]$

Partitioning complete, return value of q :

1	2	3	4	5	6	7	8	9	10	11	12
2	4	7	1	9	10	56	51	85	59	21	13

Quicksort (*A*, 1, 12)

2, 4, 7, 1, 9, 10, 56, 51, 85, 59, 21, 13

Quicksort (*A*, 1, 5)

2, 4, 7, 1, 9

Quicksort (*A*, 7, 12)

56, 51, 85, 59, 21, 13

13, 56, 51, 85, 59, 21

Quicksort (*A*, 1, 4)

2, 4, 7, 1

1, 2, 4, 7

Quicksort (*A*, 7, 11)

56, 51, 85, 59, 21

21, 56, 51, 85, 59

Quicksort (*A*, 2, 4)

2, 4, 7

Quicksort (*A*, 8, 11)

56, 51, 85, 59

56, 51, 59, 85

Quicksort (*A*, 2, 3)

2, 4

Quicksort (*A*, 8, 9)

56, 51

51, 56

Choice of pivot element affects the efficiency of algorithm :

If we choose the last or first element of an array as pivot element then it results in worst case scenario with $O(n^2)$ time complexity. If we choose the median as pivot element then it divides the array into two halves every time and results in best or average case scenario with time complexity $O(n \log n)$. Thus, the efficiency of quick sort algorithm depends on the choice of pivot element.

Que 3.19. Use quick sort algorithm to sort 15, 22, 30, 10, 15, 64, 1, 3,9, 2. Is it a stable sorting algorithm? Justify.

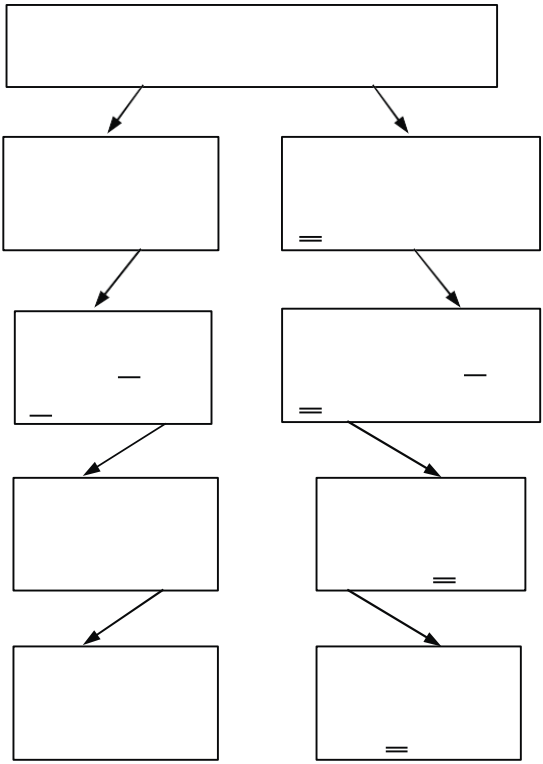
AKTU 2017-18, Marks 07

Answer

Let *A* [ ] =

1	2	3	4	5	6	7	8	9	10
15	22	30	10	15	64	1	3	9	2

Here $p = 1$, $r = 10$



$x = A[10]$ *i.e.*, $x = 2$
$i = p - 1$ *i.e.*, $i = 0$
$j = 1$ to 9

Now, $j = 1$ and $i = 0$
$A[j] = A[1] = 15$ and $15 \le 2$

So, $j = 2$ and $i = 0$
$A[2] = 22 \le 2$ (False)

Now, $j = 3$ and $i = 0$
$A[3] = 30 \le 2$ (False)
$j = 4$ and $i = 0$
$A[4] = 10 \le 2$ (False)
$j = 5$
$A[5] = 15 \le 2$ (False)
$j = 6$
$A[6] = 64 \le 2$ (False)
$j = 7$
$A[7] = 1 \le 2$ (True)
$i = 0 + 1 = 1$
$A[1] \leftrightarrow A[7]$

i.e.,

1	2	3	4	5	6	7	8	9	10
1	22	30	10	15	64	15	3	9	2

$j = 8$ and $i = 1$
$A[8] = 3 \le 2$ (False)
$j = 9$ and $i = 1$
$A[9] = 9 \le 2$ (False)

then, $A[1 + 1] \leftrightarrow A[r]$
$A[2] \leftrightarrow A[10]$
$q \leftarrow 2$

i.e.,

1	2	3	4	5	6	7	8	9	10
1	2	30	10	15	64	15	3	9	22

QUICK SORT (A, 1, 1)

1	2
1	2

QUICK SORT (A, 3, 10)

3	4	5	6	7	8	9	10
30	10	15	64	15	3	9	22

Here $p = 3, r = 10$
$x = A[10] = 22$
$i = 3 - 1 = 2$
$j = 3$ to 9; $j = 3$ and $i = 2$
$A[3] = 30 \le 22$ (False)
$j = 4$ and $i = 2$
$A[4] = 10 \le 22$ (True)
$i = 2 + 1 = 3$ and $A[3] \leftrightarrow A[4]$

i.e.,

3	4	5	6	7	8	9	10
10	30	15	64	15	3	9	22

$j = 5$ and $i = 3$

$A[5] = 15 \leq 22$ (True)

$i = 3 + 1 = 4$ and $A[4] \leftrightarrow A[5]$

3	4	5	6	7	8	9	10
10	15	30	64	15	3	9	22

Similarly

$j = 7$, $i = 4$

$A[7] = 15 \leq 22$ (True)

$i = 4 + 1 = 5$ and $A[5] \leftrightarrow A[7]$

i.e.,

3	4	5	6	7	8	9	10
10	15	15	64	15	3	9	22

Similarly, we get another pivot element

10	15	15	3	9	22	64	30

Thus, this is a stable algorithm.

PART-5

Merge Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.20. Describe two way merge sort method. Explain the complexities of merge sort method.

Answer

Merge sort :

a. Merge sort is a sorting algorithm that uses the idea of divide and conquer.

b. This algorithm divides the array into two halves, sorts them separately and then merges them.

c. This procedure is recursive, with the base criteria that the number of elements in the array is not more than 1.

MERGE_SORT (a, p, r) :

1. if $p < r$
2. then $q \leftarrow \lfloor (p + r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, $q + 1$, r)

5. MERGE (A, p, q, r)

MERGE (A, p, q, r) :

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays $L[1.....n_1 + 1]$ and $R[1..... n_2 + 1]$
4. for $i = 1$ to n_1
   do
   $L[i] = A[p + i - 1]$
   endfor
5. for $j = 1$ to n_2
   do
   $R[j] = A[q + j]$
   endfor
6. $L[n_1 + 1] = \infty$, $R[n_2 + 1] = \infty$
7. $i = 1, j = 1$
8. for $k = p$ to r
   do
   if $L[i] \le R[j]$
   then $A[k] \leftarrow L[i]$
   $i = i + 1$
   else $A[k] = R[j]j$
   $= j + 1$
   endif
   endfor
9. exit

Complexity of merge sort algorithm :

1. Let $f(n)$ denote the number of comparisons needed to sort an n-element array A using the merge sort algorithm.
2. The algorithm requires at most log n passes.
3. Moreover, each pass merges a total of n elements, and by the discussion on the complexity of merging, each pass will require at most n comparisons.
4. Accordingly, for both the worst case and average case,
   $$f(n) \le n \log n$$
5. This algorithm has the same order as heap sort and the same average order as quick sort.
6. The main drawback of merge sort is that it requires an auxiliary array with n elements.
7. Each of the other sorting algorithms requires only a finite number of extra locations, which is independent of n.
8. The results are summarized in the following table :

Algorithm	Worst case	Average case	Extra memory
Merge sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

Que 3.21. **Write an algorithm for merge sorting. Using the algorithm sort in ascending order : 10, 25, 16, 5, 35, 48, 8**

AKTU 2014-15, Marks 10

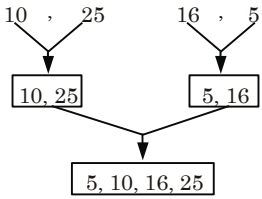
Answer

Merge sorting : Refer Q. 3.20, Page 3–18A, Unit-3.

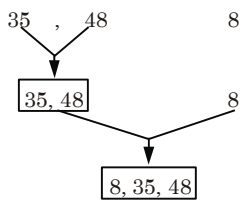
Numerical :

10, 25, 16, 5, 35, 48, 8

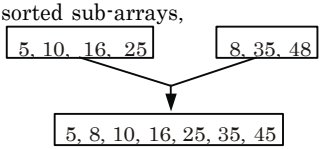
1. Divide first half [10, 25, 16, 5] [35, 48, 8]
2. **Consider the first half :** 10, 25, 16, 5 again divide into two sub- arrays



3. **Consider the second half :** 35, 48, 5 again divide into two sub-arrays



4. Merge these two sorted sub-arrays,



This is the sorted array.

Que 3.22. **How do you calculate the complexity of sorting algorithms ? Also, write a recursive function in 'C' to implementthe merge sort on given set of integers. AKTU 2015-16, Marks 10**

Answer

Complexity : Refer Q. 3.20, Page 3–18A, Unit-3.

Function :

```
void merge (int low, int mid, int high)
{
int temp [MAX] ;
int i = low;
int j = mid + 1;
int k = low;
while ((i <= mid) && (j <= high))
{
if (array [i] <= array [j ] )
      temp [k++] = array [i++];
else
      temp [k++] = array [j++] ;
}
      while (i <= mid)

temp [k++] = array [i++];
while (j <= high)
temp [k++] = array [j++] ;
for (i = low; i <= high; i++)
array [i] = temp [i];
}
void merge_sort (int low, int high)
{
int mid;
if (low != high)
{
mid = (low + high) / 2;
merge_sort (low, mid);
merge_sort (mid + 1, high);
merge (low, mid, high);
}
}
```

PART-6

Heap Sort and Radix Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.23. **Write a short note on heap sor05**

OR

Explain heap sort. **AKTU 2017-18, Marks 3.5**

Answer

1. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.
2. The general approach of heap sort is as follows :
   a. From the given array, build the initial max heap.
   b. Interchange the root (maximum) element with the last element.
   c. Use repetitive downward operation from root node to rebuild the heap of size one less than the starting.
   d. Repeat step (a) and (b) until there are no more elements.

Analysis of heap sort :

Complexity of heap sort for all cases is $O(n \log_2 n)$.

MAX-HEAPIFY (A, i) :

1. $i \leftarrow$ left [i]
2. $r \leftarrow$ right [i]
3. if $l \leq$ heap-size [A] and $A[l] > A[i]$
4. then largest $\leftarrow l$
5. else largest $\leftarrow i$
6. if $r \leq$ heap-size [A] and $A[r] > A$ [largest]
7. then largest $\leftarrow r$
8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A$[largest]
10. MAX-HEAPIFY [A, largest]

HEAP-SORT(A) :

1. BUILD-MAX-HEAP (A)
2. for $i \leftarrow$ length [A] down to 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. heap-size [A] $\leftarrow$ heap-size [A] – 1
5. MAX-HEAPIFY (A, 1)

Que 3.24. Write a short note on radix sort.

OR

Explain radix sort. **AKTU 2017-18, Marks 3.5**

Answer

1. Radix sort is a small method that many people uses when alphabetizing a large list of names (here Radix is 26, 26 letters of alphabet).
2. Specifically, the list of name is first sorted according to the first letter of each name, *i.e.*, the names are arranged in 26 classes.
3. Intuitively, one might want to sort numbers on their most significant digit.

4. But radix sort do counter-intuitively by sorting on the least significant digits first.
5. On the first pass entire numbers sort on the least significant digit and combine in an array.
6. Then on the second pass, the entire numbers are sorted again on the second least-significant digits and combine in an array and so on.
7. Following example shows how radix sort operates on seven 3-digit number.

Table 3.24.1.

Input	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

8. In the above example, the first column is the input.
9. The remaining shows the list after successive sorts on increasingly significant digits position.
10. The code for radix sort assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

RADIX_SORT (A, d)

for $i \leftarrow 1$ to d do
use a stable sort to sort array A on digit i
// counting sort will do the job

Analysis :

1. The running time depends on the table used as an intermediate sorting algorithm.
2. When each digit is in the range 1 to k, and k is not too large, COUNTING_SORT is the obvious choice.
3. In case of counting sort, each pass over n d-digit numbers takes $\Theta(n + k)$ time.
4. There are d passes, so the total time for radix sort is $\Theta(n + k)$ time. There are d passes, so the total time for radix sort is $\Theta(dn + kd)$. When d is constant and $k = \Theta(n)$, the radix sort runs in linear time.