DATA STRUCTURE AND ALGORITHMS

UNIT-2

Sorting:

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being unique phone number can work as a key to locate any record in the list.

The complexity of sorting algorithm:

The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program

The Efficiency of sorting algorithm:

To get the amount of time required to sort an array of 'n' elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting techniques are data sensitive, and so the metrics for them depends on the order in which they appear in an input array.

Various sorting techniques are analyzed in various cases and named these cases as follows:

- Best case
- Worst case
- Average case

Categories of Sorting:

The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

Internal Sorting: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External Sorting: When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc., then external sorting methods are performed.

Types of sorting techniques:

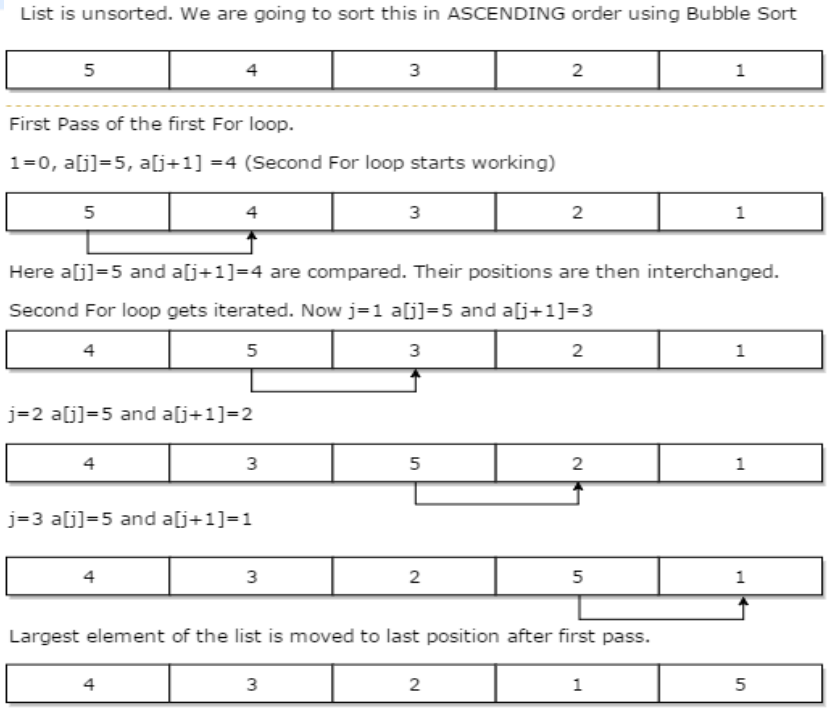
1. Bubble Sort

Bubble Sort is a comparison-based sorting algorithm. In this algorithm adjacent elements are compared and swapped to make the correct sequence. This algorithm is simpler than other algorithms, but it has some drawbacks also. This algorithm is not suitable for a large number of data set. It takes much time to solve the sorting tasks.

The complexity of the Bubble Sort Technique

- **Time Complexity:** O(n) for best case, O(n^2) for average and worst case
- **Space Complexity:** O(1)

Below given figure shows how Bubble Sort works:



Algorithm for Bubble Sort

1. Bubble_Sort(list)
2. Pre: list != fi
3. Post: list is sorted in ascending order for all values
4. for i <- 0 to list:Count - 1
5. for j <- 0 to list:Count - 1
6. if list[i] < list[j]
7. Swap(list[i]; list[j])
8. end if
9. end for
10. end for
11. return list
12. end Bubble_Sort

2. Selection Sort:

In the selection sort technique, the list is divided into two parts. In one part all elements are sorted and in another part the items are unsorted. At first, we take the maximum or minimum data from the array. After getting the data (say minimum) we place it at the beginning of the list by replacing the data of first place with the minimum data. After performing the array is getting smaller. Thus, this sorting technique is done.

The complexity of Selection Sort Technique:

- Time Complexity: O(n^2)
- Space Complexity: O(1)

The below-given figure shows how Selection Sort works:

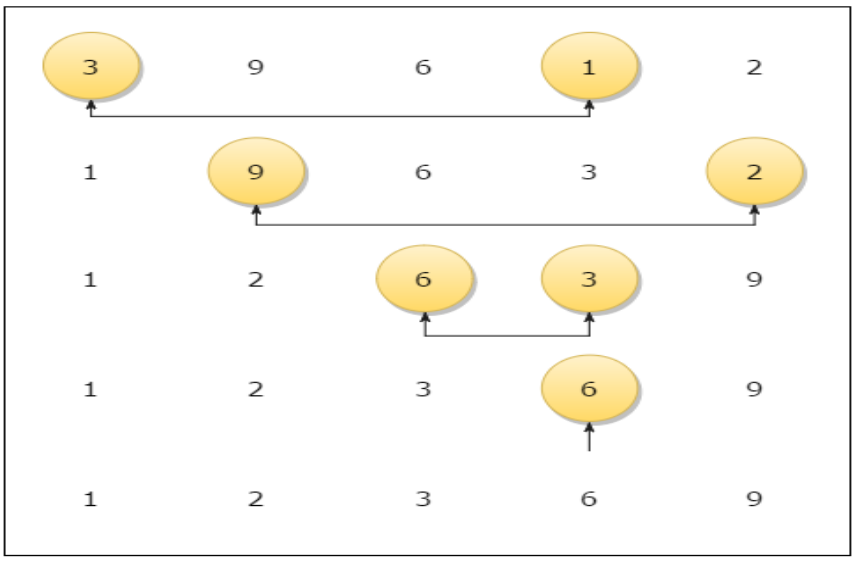


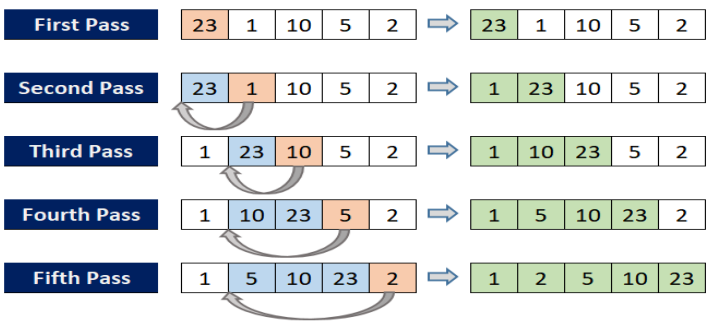
Fig. Selection Sort Technique

Algorithm:

1. Begin
2. for i := 0 to size-2 do
3. for j:= i+1 to size – 1 do
4. if array[j] < array[iMin] then
5. iMin :=j
6. Done
7. swap array[i] with array[iMin].
8. Done
9. End

3. Insertion Sort

This sorting technique is similar with the card sorting technique, in other words, we sort cards using insertion sort mechanism. For this technique, we pick up one element from the data set and shift the data elements to make a place to insert back the picked up an element into the data set.



The complexity of the Insertion Sort Technique

- Time Complexity: O(n) for best case, O(n^2) for average and worst case
- Space Complexity: O(1)

Algorithm

1. Begin
2. for i := 1 to size-1 do
3. key := array[i]
4. j := I
5. while j > 0 AND array[j-1] > key do
6. array[j] := array[j-1];
7. j := j – 1
8. Done
9. array[j] := key
10. Done
11. End

4. Merge Sort:

The merge sort technique is based on divide and conquers technique. We divide the whole dataset into smaller parts and merge them into a larger piece in sorted order. It is also very effective for worst cases because this algorithm has lower time complexity for the worst case also.

The complexity of Merge Sort Technique:

- **Time Complexity:** O(n log n) for all cases
- **Space Complexity:** O(n)

Algorithm:

```
Begin
  nLeft := m - left+1
  nRight := right – m
  define arrays leftArr and rightArr of size nLeft and nRight respectively

  for i := 0 to nLeft do
    leftArr[i] := array[left +1]
  done

  for j := 0 to nRight do
    rightArr[j] := array[middle + j +1]
  done
```

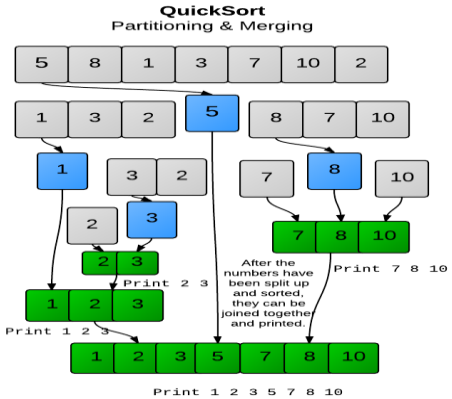
```
  i := 0, j := 0, k := left
  while i < nLeft AND j < nRight do
    if leftArr[i] <= rightArr[j] then
      array[k] = leftArr[i]
      i := i+1
    else
      array[k] = rightArr[j]
      j := j+1
    k := k+1
  done

  while i < nLeft do
    array[k] := leftArr[i]
    i := i+1
    k := k+1
  done

  while j < nRight do
    array[k] := rightArr[j]
    j := j+1
    k := k+1
  done
End
```


5.Quick Sort

The quicksort technique is done by separating the list into two parts. Initially, a pivot element is chosen by partitioning algorithm. The left part of the pivot holds the smaller values than the pivot, and right part holds the larger value. After partitioning, each separate lists are partitioned using the same procedure.



The complexity of Quicksort Technique

- **Time Complexity:** O(n log n) for best case and average case, O(n^2) for the worst case.
- **Space Complexity**: O(log n)

Algorithm:

```
Begin
  pivot := array[lower]
  start := lower and end := upper
  while start < end do
    while array[start] <= pivot AND start < end do
      start := start +1
    done

    while array[end] > pivot do
      end := end – 1
    done
    if start < end then
      swap array[start] with array[end]
  done

  array[lower] := array[end]
  array[end] := pivot
  return end
End
```


6. Radix Sort

Radix sort is a non-comparative sorting algorithm. This sorting algorithm works on the integer keys by grouping digits which share the same position and value. The radix is the base of a number system. As we know that in the decimal system the radix or base is 10. So for sorting some decimal numbers, we need 10 positional boxes to store numbers.

The complexity of Radix Sort Technique

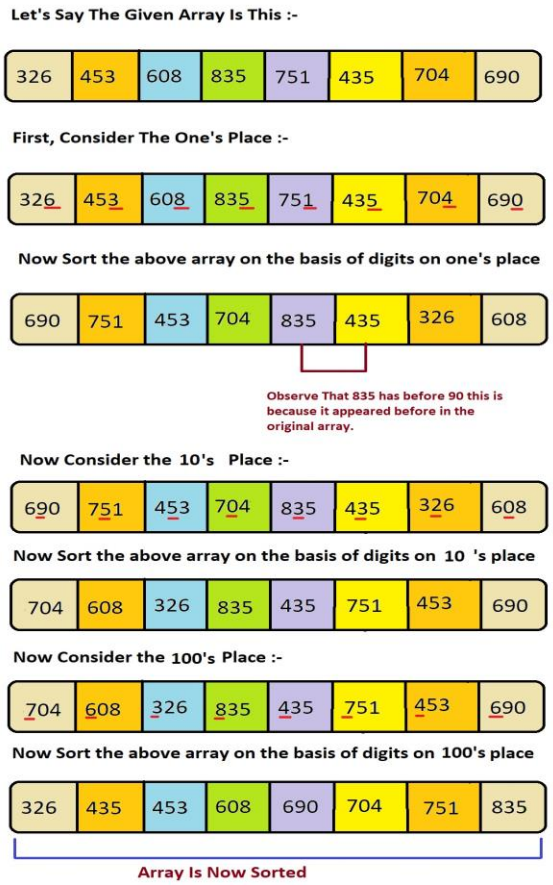
- **Time Complexity:** O(nk)
- **Space Complexity:** O(n+k)

Algorithm

```
Begin
  define 10 lists as pocket
  for i := 0 to max -1 do
    m = 10^i+1
    p := 10^i
    for j := 0 to n-1 do
```

```
      temp := array[j] mod m
      index := temp / p
      pocket[index]. append(array[j])
    done

    count := 0
    for j := 0 to radix do
      while pocket[j] is not empty
        array[count] := get first node of pocket[j] and delete it
        count := count +1
    done
  done
End
```

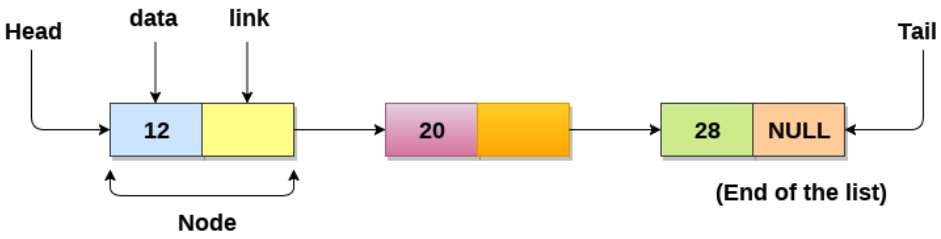


ARRAY VS LINKED LIST:

Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.

Linked List:

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.

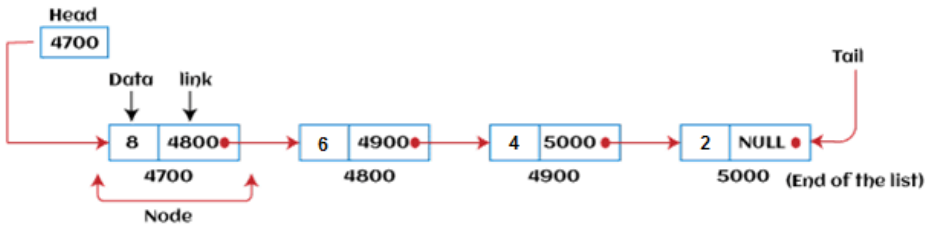


Uses of Linked List:

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Representation of a Linked list:

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory.

Advantages of Linked list:

- **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list:

- **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.

- **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list:

The basic operations that are supported by a list are mentioned as follows -

- **Insertion -** This operation is performed to add an element into the list.
- **Deletion -** It is performed to delete an operation from the list.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Complexity of Linked list:

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. Time Complexity

Operations	Average case time complexity	Worst-case time complexity

Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

Types of linked List:

1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list
4. Doubly Circular Linked list

1. Single Linked List:

the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.

2. Doubly linked list:

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

3. Circular singly linked list:

In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

4. Circular doubly linked list:

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Node Creation:

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *) malloc(sizeof(struct node *));
```

Operations on Single Linked List:

1. Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

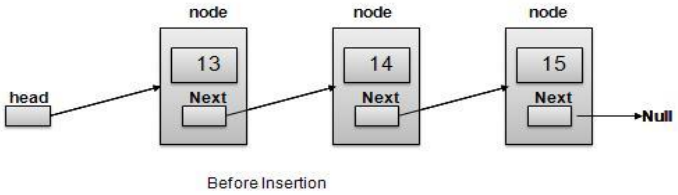
1. Insertion at beginning: It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.

2. Insertion at end of the list: It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.

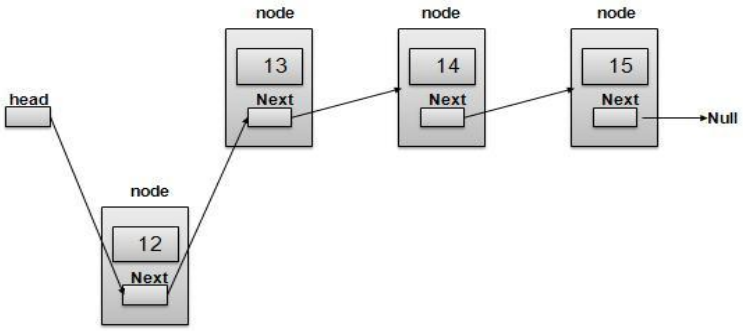
3. Insertion after specified node: It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Insertion is a three-step process −

- Create a new Link with provided data.
- Point New Link to old First Link.
- Point First Link to this New Link.



Before Insertion



After Insertion

```
//insert link at the first location
void insertFirst(int key, int data){
  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;

  //point it to old first node
  link->next = head;

  //point first to new first node
  head = link;
}
```


2. Deletion:

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

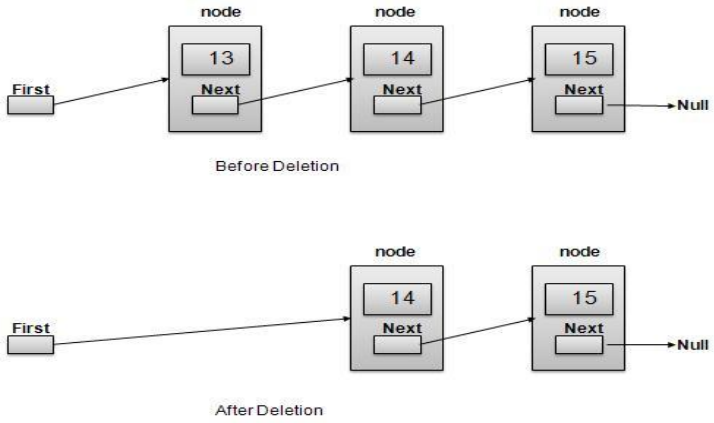
1. Deletion at beginning: It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just needs a few adjustments in the node pointers.

2. Deletion at the end of the list: It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.

3. Deletion after specified node: It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted.

Deletion is a two-step process –

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.




```
//delete first item
struct node* deleteFirst(){
  //save reference to first link
  struct node *tempLink = head;

  //mark next to first link as first
  head = head->next;

  //return the deleted link
  return tempLink;
}
```


3. Traversing

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
ptr = head;
      while (ptr!=NULL)
        {
           ptr = ptr -> next;
        }
```

Algorithm:

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

STEP 3: WRITE "EMPTY LIST"
GOTO STEP 7
END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL

STEP 5: PRINT PTR→ DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: EXIT

4. Searching

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element, then the location of the element is returned from the function.

Algorithm:

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL
- WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
- **STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL
- **STEP 5:** if ptr → data = item
- write i+1
  End of IF
- **STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT
- [END OF LOOP]
- **STEP 8:** EXIT