vii) This algorithm work efficiently for already sorted input sequence by design and hence running time of an algorithm is O(n) and it is considered as a best case.

Algorithm

Step 1:
If it is the first element, it is already sorted return 1.

Step 2:
pick next element.

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

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

Step 5:
Insert the value.

Step 6:
Repeat until list is sorted.

Pseudocode.

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

* Key points about insertion sort.

1) Best case insertion sort $\Omega(n)$
2) Worst case insertion sort $O(n^2)$
3) Average case insertion sort $\Theta(n^2)$
4) insertion sort algorithm work efficiently for already sorted input array by design.
5) It is efficient for smaller data sets, but very inefficient for larger list.
6) insertion sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
7) Space complexity is less.

8) The best time complexity. $O(n)$
9) The worst time complexity $O(n^2)$

4] Merge Sort.

i) merge sort is a sorting technique based on divide and conquer technique.

ii) There are 2 steps in this algorithm.
Step 1 :-

a) In each iteration by means of calculating mid pos, we are dividing big size array logically into two subarray's left subarray will be from left to mid, and right subarray will be from mid + 1 to right.

b) For left subarray, value of left remains as it is and right = mid

For right subarray, value of right remains as it is and left = mid + 1.

Step 2 :

a) merge two already sorted subarray's into a single array in a sorted manner.

b) If subarray contains only 1 ele ⇒ subarray is already sorted.

* Algorithm

Step 1:
If it is only one element in the list it is already sorted, return.

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

Step 3:
Merge the smaller lists into new list in sorted manner.

* Pseudocode.

```
Procedure mergsort (var a as array)

if (n==1) return a

var l1 as array = a[0] ... a[n/2]
var l2 as array = a[n/2+1] ... a[n]

l1 = mergsort (l1)
l2 = mergesort (l2)

return merge (l1, l2)
end procedure
```

```
procedure merge (var a as array, var b as array

var c as array
while (a and b have elements)
   if (a[0] > b[0])
      add b[0] to the end of c
      remove b[0] from b.
else
      add a[0] to the end of c
      remove a[0] from a
end if
 end while

while (a has elements)
      add a[0] to the end of c
      remove a[0] from a
      end while

while (b has elements)
   add b[0] to the end of c
      remove b[0] from b
   end while

return c
end procedure.
```

* Key points about merge sort.

1) Merge sort is useful for sorting linked list.
2) merge sort is a stable sort which means that the same element is an array maintain their original position with respect to each other.
3) Overall time complexity of merge sort is O(nlogn)
4) space complexity of merge sort is O(n).
5) merge sort best case $\Omega(n \log n)$
6) merge sort worst case $O(n \log n)$
7) merge sort average case $\Theta(n \log n)$

5) Quick Sort.

i) Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data, into smaller array.

ii) A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds value greater than the pivot value.

iii) Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

iv) This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

v) This algorithm follows divide and conquer approch.

vi) In this algorithm the basic logic is a partitioning.

vii) partitioning:
In partitioning, pivot element gets selected first (it may be either leftmost or rightmost or middle most element in an array).

viii) After selection of pivot element all the elements which are smaller than pivot gets arranged towards as its left as possible & elements which are greater than pivot gets arranged as its right as possible, and big size array is divided into two subarray's. so after first pass pivot element gets settled as its appropriate position, elements which are at left of pivot is referred as left partition and elements which are at its right reffered as a right partition.

* Key points about Quick sort.

1) Quick sort is useful for sorting long arrays.
2) In efficient implementation Quick sort is not a stable sort, meaning that the relative order of equal sort items is not preserved.
3) Overall time complexity of Quick sort is $O(n \log n)$
4) The space complexity of Quick sort is $o(n \log n)$
5) Best case for Quick sort $\Omega(n \log n)$
6) Worst case for Quick sort $O(n^2)$
7) Average case for Quick sort $\theta(n \log n)$
8) Quick sort algorithm is considered the most efficient Sorting algorithm for ~~smaller input~~ larger input size array.

* Array Data Structure.

i) Array is a container which can hold a fix number of items and these items should be of the same type.

ii) Most of the data structure make use of arrays to implement their algorithms follow are the important terms to understand th concept of array.

Element: Each item is stored in an array called an element.

Index - Each location of an element in an array has a numerical index, which is used to identify index.

* Basic Operations

Following are basic operations supported by an array.

1) Traverse
print all elements one by one

2) Insertion.
Adds an element at the given index.

3) Deletion.
Deletes an element at the given index.

4) Search.
Searches an element at the given index.

5) Update.
Updates an element at the given index.

* Difference between array and linked list.

Comparison chart.

Basis For comparison	Array	Linked List.
i) Basic	It is a consistent set of a data fixed numbers.	It is an ordered set comprising a variable number of data items.
ii) Size	specified during declaration.	No need to specify grow and shrink during execution.
iii) Storage allocation.	Element location is allocated during compile time	Element position is assigned during run time.
iv) Order of element.	Stored consecutively.	stored randomly.

v)	Acessing the element.	Direct or randomly accessed i.e specify the array inder or subscript.	Sequentially accessed i.e traverse starting from the first node in the list by the pointe
vi)	Insertion & deletion of element.	slow relatively as shifting is required.	Easier fast and efficient.
vii)	Searching	binary search & linear search.	linear search.
viii)	memory required	less	more.
ix)	memory utilization	Ineffective	Efficient.

* Key differences between array and linked list.

1) An array is a data structure contains a collection of similar type data elements whereas the linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.

2) In the array the element belongs to indexes i.e if you want to get into the fourth element you have to write the variable name with its index or location with the square bracket.

In a linked list through you have to start from the head and work your way through until you get to the fourth element.

3) While accessing an element is faster in array while linked list takes linear time so, it is quite bit slower.

4) Operation like insertion and deletion in array consume a lot of time. On the other hand, the performance of these operations in linked lists is fast.

5) Arrays are of fixed size. In contrast, linked list are dynamic and flexible and can expand and contract its size.

6) In an array memory is assigned during compile time while in a linked list it is allocating during execution or runtime.

7) Elements are stored consecutively in array whereas it is stored randomly in linked lists.

8) The requirement of memory is less due to actual data being stored within the index in the array. As against there is need for more memory in linked list due to storage of additional next and previous referceing elements.

9) In addition memory utilization is inefficient in the array, conversly memory utilization is efficient in array.

* Linked List

i) It is a basic/linear data structure, which is a collection/list of logically related similar type of data element in which an address of first element always gets stored into a pointer variable referred as head and each element contains actual data and an address of its next element.

ii) Elements in the collection/list/data structure are explicitly linked with each other and hence it is called linked list.

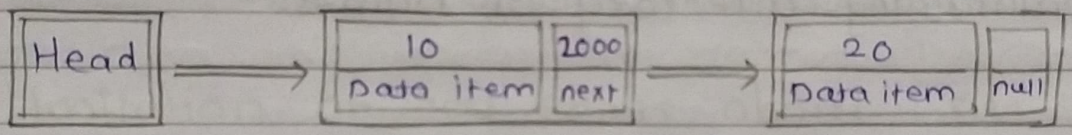
iii) A linked list is a sequence of data structure which are connected together via links.

iv) Linked list is a sequence of links which contains items, each link contain connection to another link.

i) Link
Each link of a linked list can store a data called an element.

ii) Next.
Each link of a linked list contains a link to the next called next.

* Linked List Representation.



As per above figure.

i) Linked list contains a link element called first.
ii) Each link carries a data fields and a link field called next.
iii) Each link is linked with its next link using its next link.
iv) Last link carries a link as null mark the end of the list.

NULL
NULL is a predefined macro whose value is 0 which is typecasted into a void*

```
# define NULL ((void *)0).
```

* Key points about linked list.

i) Used for implementation of stacks and queues.
ii) Implementation of graph : Adjacency list representation of graph is most popular which is uses linked list to store adjacent vertices.
iii) Dynamic memory allocation: we use linked list of free blocks.
iv) maintaining directory of names.

* Linked list divided into two types.
1) singly linked list
2) doubly linked list.

1) Singly linked list.
i) In singly linked list is a type of linked list in which each element / node contains actual data and an address of its next node i.e each node contains only one / single link.

ii) Singly linked list divided into two types.
a) singly linear linked list.
b) Singly circular linked list.

a) Singly linear linked list.
It is a type of linked list in which, head always contains an address of first node, if

list is not empty each node has two parts.
1. Data part : contains actual data.
2. pointer part : contains an address of its next last node points to null i.e next part of last node contains NULL.

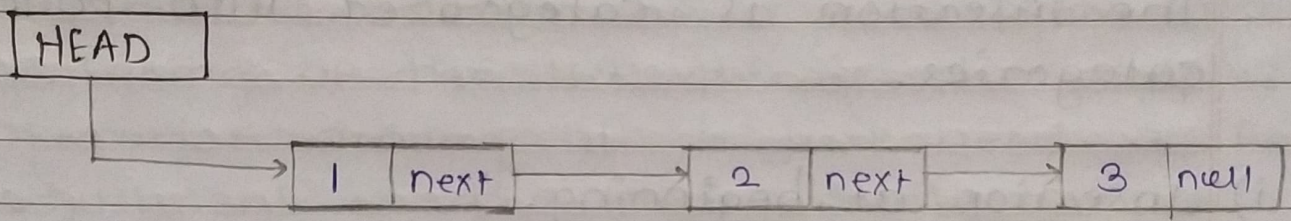


Fig: Singly linear linked list.

* Limitation or key points about SLLL.

i) We can traverse singly linear linked list only in a forward direction.
ii) previous node of any node cannot be accessed from it.
iii) add_last and delete_last() function are not efficient as it takes O(n) time.
iv) any node cannot be revisited.

* Complexity of Linked list.

Data Structure	Time Complexity (Average)				Worst time complexity			
	Access	search	insertion	Deletion	Access	search	deletion	insertion
Singly linked list	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Worst space complexity :— $O(n)$.

* Insertion into linked list.

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

i) 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.

ii) 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.

iii) Insertion after specified node.
It involves insertion after the specified node of the linked list. We need to skip desired no. of nodes in order to reach the node after which the new node will be inserted.

* Deletion and Traversing.
The deletion of a node from a singly linked list can be performed at different

position, based on the position of the node being delered, the operation is categorized into the following categories.

i) Deletion at beginning.
It involves deletion of a node from the beginning of the list this is the simplest operation among all. it just need a few adjustment in the node pointers.

ii) Deletion at the end of the list.
It involves delering the last node of the list can either be empty or full. different logic is implemented for the different scenarios.

iii) 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 delered. This requires traversing through the list.

iv) Traversing.
In traversing we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

v) Searching
In Searching, we match each element of the

given element. if the element is found on any of the location then location of that element is returned otherwise null is returned

b) Singly circular linked list.

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

ii) We traverse a circular singly linked list until we reach the same node where we started.

iii) The circular singly linked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

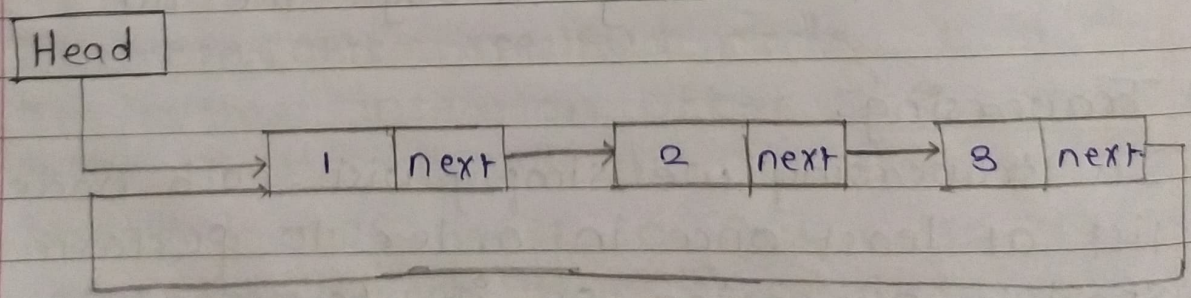


Fig: circular linked list.

* Limitation or key points of SCLL.

i) We can traverse singly circular linked list only in a forward direction.
ii) Previous node of any node cannot accessed from it.
iii) As even while adding or delering node at first pos. we need to traverse list till last node to maintain next part of last node.

* Operations on circular singly linked list.

i) Insertion at beginning.
Adding a node into circular singly linked list at the beginning.

ii) Insertion at the end.
Adding a node into circular singly linked list at the end.

iii) Deletion at beginning.
Removing the node from circular singly linked list at the beginning.

iv) Deletion at the end.
Removing the node from circular singly linked list at the end.

v) Searching
Compare each element of the node with the

given item and return the location at which the item is present in the list otherwise return null.

vi) Traversing.
visiting each element of the list at least once in order to perform some specific operation.

2] Doubly linked List.

i) 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.

ii) Therefore, in doubly linked list, a node consist of three parts:
i) node data
ii) pointer to the next node in sequence
iii) pointer to previous node.

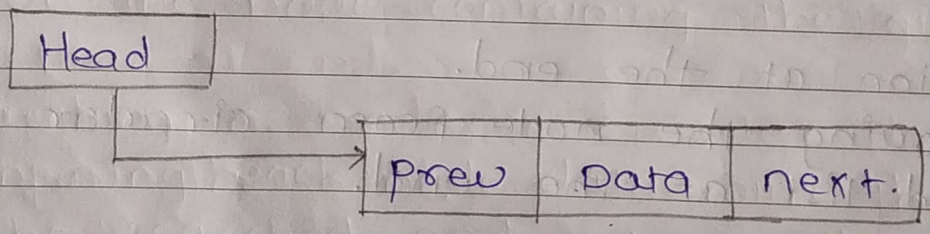


Fig: Doubly linked list.

iii) Singly linked list divided into two parts.
a) Doubly linear linked list.
b) Doubly circular linked list.

a) Doubly linear linked list.

i) A doubly linked list containing three node having numbers from 1 to 3 in their data part, is shown in the following.

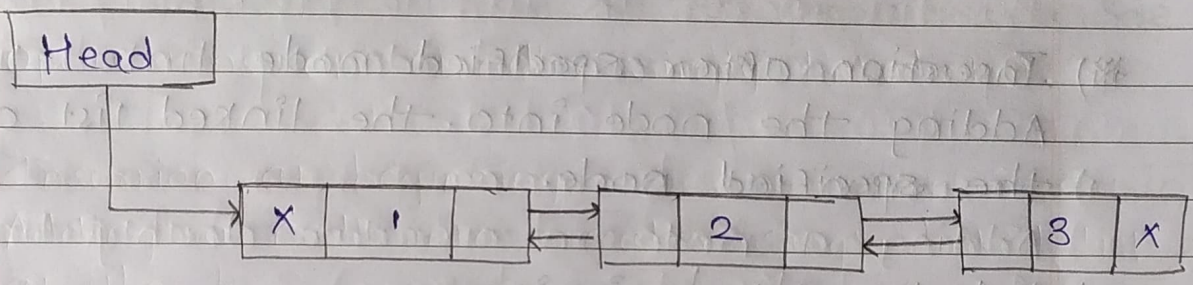


Fig: ~~St~~ Doubly linear linked list.

ii) The previous part of the first node and the next part of the last node will always contain null indicating end in each direction.

iii) In a singly linked list, we could travers only in one direction, because each node contains address of the next node and it doesn't have any previous nodes.

iv) However, doubly linked list overcome this limitation of singly linked list. Due to the fact that each node of the list contains the address of its previous node.

* Operation on doubly linked list.

i) Insertion at beginning.
Adding the node into the linked list at beginning.

ii) Insertion at end.
Adding the node into the linked list to the end.

iii) Insertion after specified node
Adding the node into the linked list after the specified node.

iv) Deletion at beginning.
Removing the node from beginning of the list.

v) Deletion at the end.
Removing the node from end of the list.

vi) Deletion of the node having given data.
Removing the node which is present just after the node containing the given data.

vii) Searching
comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

viii) Traversing.
visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

b) Doubly circular linked list.

i) circular doubly linked list is a more complexed type of data structure in which a node contain pointer to its previous node as well as the next node.

ii) circular doubly linked list doesn't contain NULL in any of the node.

iii) The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

iv) A circular doubly linked list is shown in following figure.

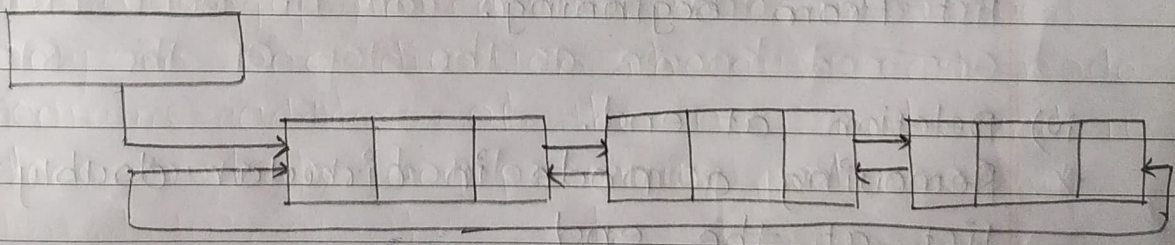


Fig: Circular doubly linked list.

v) Due to the fact that a circular doubly

linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operation.

vi) However, a circular doubly linked list provides easy manipulation of the pointer and the searching become twice as efficient.

* Operation on circular doubly linked list.

i) Insertion at beginning.
Adding a node in circular doubly linked list at the beginning.

ii) Insertion at end.
Adding a node in circular doubly linked list at the end.

iii) Deletion at beginning.
Removing a node in circular doubly linked list from beginning.

iv) Deletion at end.
Removing a node in circular doubly linked list at the end.