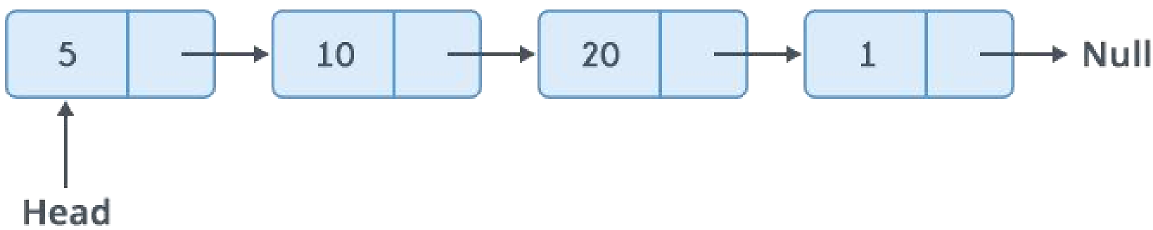
MODULE III

LINKED LISTS

A **linked list** is a way to store a collection of elements. Each element in a linked list is stored in the form of a **node**. A **data** part stores the element and a **next** part stores the link to the next node.

Linked List:



Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

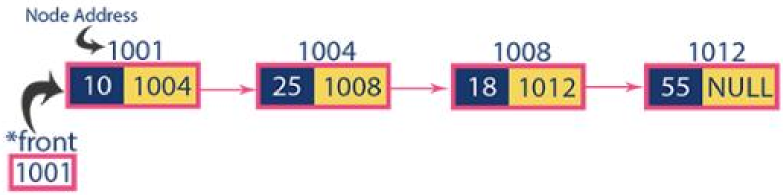
SINGLE LINKED LIST:

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...

Example



Operations on single linked list:
In a single linked list we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Traverse
5. Searching

Creation of a node:

- **Step 1:** Include all the **header files** and user defined functions.
- **Step 2:** Define a **Node** structure with two members **data** and **next**
- **Step 3:** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying operations menu

```
struct node
{
    int data;
    struct node *next;
};
```

Insertion:

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

- Step 1: Start
- Step 2: Create a newnode with given value.
- Step 3: Check whether list is Empty (head == NULL)
- Step 4: If it is **Empty**:
  set newNode→next = NULL
  set head = newNode.
- Step 5: If it is **Not Empty:**
  set newNode→next = head
  set head = newNode.
- Step 6: Stop

Inserting At End of the list

- Step 1: Start
- Step 2: Create a newnode with given value.
- Step 3: Check whether list is Empty (head == NULL)
- Step 4: If it is **Empty**:
  set newNode→next = NULL
  set head = newNode.
- **Step 5:** If it is **Not Empty** then define a node pointer **temp**
  temp = head (initialize temp with **head**).
- **Step 6:** move **temp** to its next node until it reaches to the last node in the list
  (until **temp → next** = **NULL**).
- **Step 7:** Set **temp → next** = **newNode**.
- **Step 8:** Stop

Inserting At Specific location in the list (After a Node)

- Step 1: Start
- Step 2: Create a newnode with given value.
- Step 3: Check whether list is Empty (head == NULL)
- Step 4: If it is **Empty**:
  set newNode→next = NULL
  set head = newNode.
- **Step 5:** If it is **Not Empty**, then define a node pointer **temp**
  temp = head (initialize temp with **head**).
- **Step 6:** move **temp** to its next node until it reaches specific location to insert
  (until **temp → data** = **location**).
- **Step 7:** Set **newNode → next** = **temp → next**
  Set temp → next = **newNode**
  Step 8: Stop

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

- **Step 1:** Start
- Step 2: Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty**:
  display **'List is Empty!!! Deletion is not possible'**.
- **S**tep 4: If it is Not Empty then define a Node pointer 'temp'
  Set temp = head (initialize temp with **head**).
- **Step 5: Set head** = **temp → next**
  delete **temp**.
  free (temp)

Deleting from End of the list

- **Step 1: start**
- Step 2: Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty**:
  display **'List is Empty!!! Deletion is not possible'**
- **Step 4:** If it is **Not Empty** then define two Node pointers **'temp1'** and '**temp2'**
  Set temp1 = head (initialize '**temp1'** with **head)**.
- **Step 5:** set '**temp2 = temp1** ' and move **temp1** to its next node.
- **Step 6:** Repeat the same until it reaches to the last node in the list.
  (until **temp1 → next** == **NULL**)
- **Step 7:** Finally, Set **temp2 → next** = **NULL**
  delete **temp1**.
  free (temp1).

Deleting a Specific Node from the list

- Step 1: start
- Step 2: Check whether list is **Empty** (**head** == **NULL**)
- Step 3: If it is **Empty**:
  display **'List is Empty!!! Deletion is not possible'**
- Step 4**:** If it is **Not Empty**, then define two Node pointers **'temp1'** and '**temp2'**
  Set temp1 = head (initialize '**temp1'** with **head)**.
- Step 5: set '**temp2 = temp1** ' and move **temp1** to its next node.
- Step 6: Repeat the same until it reaches specific node which we want to delete.
- Step 7: set **temp2 → next** = **temp1 → next**
  delete **temp1**
  free(temp1)

Traverse

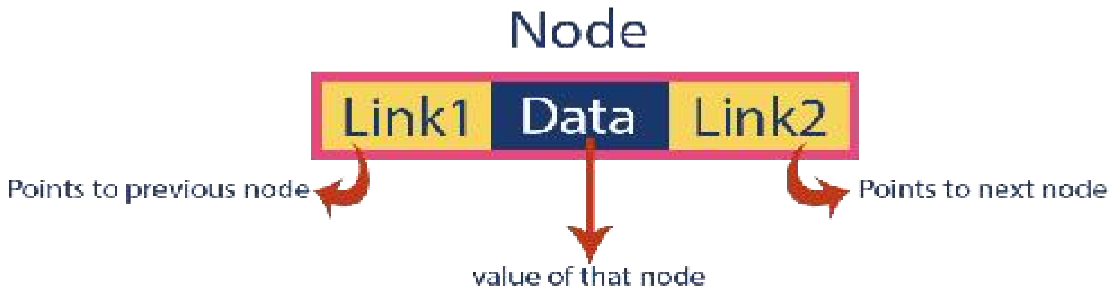
Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = head (initialize temp with head).
Step 5: Keep displaying temp→ data until temp reaches last node
temp = temp→next
Step 6: Stop

Searching

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty**:
display **List is Empty. Searching is not possible.**
Step 4: If it is **Not Empty:** define a Node pointer **'temp'**
Set temp = head (initialize temp with **head**).
Step 5: Enter item to search i.e., key
Step 6: move temp until it reaches key.
temp = temp →next
Step 7: if(temp →data = key) then
print "search is successful"
else
print "search is unsuccessful"

DOUBLE LINKED LIST:

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



☀ In double linked list, the first node must be always pointed by head.

☀ Always the previous field of the first node must be NULL.

☀ Always the next field of the last node must be NULL.

In a Double linked list we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Traverse
5. Searching

Creation of a node:

- Step 1: Include all the **header files** and user defined functions.
- Step 2: Define a **Node** structure with two members **data** and **next**
- Step 3: Define a Node pointer '**head**' and set it to **NULL**.
- Step 4: Implement the **main** method by displaying operations menu

```
struct node
{
    int data;
    struct node *prev, *next;
};
```

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then
Set newnode→prev =null

Set newNode → next = null
Set newnode→ data = value
Set head = newnode

Step 5: If it is **not Empty** then

Set newNode → next = head
Set head→prev = newNode
Set head = newnode

Inserting At Specific location in the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then

Set newnode→prev =null
Set newNode → next = null
Set newnode→ data = value
Set head = newnode

Step 5: If it is **not Empty**: (define a node pointer temp)
Set temp = head (initialize temp with head)

Step 6**:** move **temp** to its next node until it reaches specific location to insert
(until **temp** → **data** = **location**).

Step 7: Set newnode→next = temp → next
Set newnode→prev=temp
Set (temp→next)→prev = newnode
Set temp →next = newnode

Inserting At End of the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then

Set newnode→prev =null
Set newNode → next = null
Set newnode→ data = value
Set head = newnode

Step 5: If it is **not Empty**: (define a node pointer temp)
Set temp = head (initialize temp with head)

Step 6**:** move **temp** to its next node until it reaches last node to insert.
(until temp → next = null)

Step 7: Set temp→next = newnode
Set newnode→prev = temp
Set newnode → next =null

Deletion:
In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting at Specific location

Deleting from Beginning of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is not Empty: then define a Node pointer **'temp'**
Set temp = head (initialize temp with **head)**.
Step 4: Set head = temp→next
Set head→ prev = null
Set temp → next = null
delete temp
free (temp)

Deleting at Specific location

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is not Empty, then define a Node pointer **'temp'**
Set temp =head (initialize temp with head**)**.
Step 4: Keep moving the **temp** until it reaches specific node to delete.
Step 5: Set (temp→prev)→next = temp→next
Set (temp→next) →prev = temp→prev
delete temp
free(temp)

Deleting from End of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is **not Empty**, then define a Node pointer **'temp'**
Set temp =head (initialize temp with head).
Step 4: Keep moving the temp until it reaches last node to delete.
(until temp → next = NULL)
Step 5: set (temp → prev) → next = null
delete temp

free(temp)

Traverse (forward):

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = head (initialize temp with head).
Step 5: Keep moving temp forward
temp = temp→next
Step 6: Keep displaying temp→ data until temp reaches last node
(until temp→next=null)
Step 6: Stop

Traverse (backward):

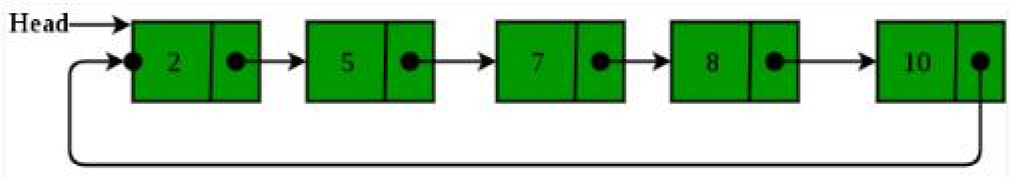
Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = tail (initialize temp with tail).
Step 5: Keep moving temp backward
temp = temp→prev
Step 6: Keep displaying temp→ data until temp reaches head node
(until temp→prev=null)
Step 7: Stop

Searching:

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty**:
display "**List is Empty. Searching is not possible".**
Step 4: If it is **Not Empty** then define a Node pointer **'temp'**
Set temp = head (initialize temp with **head**).
Step 5: Enter item to search i.e., key
Step 6: move temp until it reaches key.
temp = temp →next
Step 7: if(temp →data = key) then
print "search is successful"
else
print "search is unsuccessful"

CIRCULAR LINKED LIST:

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.



Operations:

- Insertion
- Deletion
- Traverse
- Searching

Insertion (begin)

Step 1: Start
Step 2: Create a new node with a given value
Step 3: Check whether list is Empty **(head == NULL)**
Step 4: If list is empty then

Set Newnode→data = value
Set Newnode→next=newnode
Set head = newnode
Set tail = newnode

Step 5: If list is non-empty then

Set newnode→next=head
Set head=newnode
Set tail→next=newnode

Step 6: Stop

Insertion (End)

Step 1: Start
Step 2: Create a new node with a given value
Step 3: Check whether list is Empty (**head == NULL**)
Step 4: If list is empty:

Set Newnode→data = value
Set Newnode→next=newnode
Set head = newnode
Set tail = newnode

Step 5: If list is non-empty:

Set tail→next=newnode
Set tail=newnode
Set tail→next=head

Step 6: Stop

Insertion (Specific location)

Step 1: Start
Step 2: Create a new node with a given value
Step 3: Check whether list is Empty (**head == NULL**)
Step 4: If list is empty:

Set Newnode→data = value
Set Newnode→next=newnode
Set head = newnode
Set tail = newnode

Step 5: If list is not empty: Define pointer temp.
Set **temp = head** (initialize temp with head)
Step 6: move **temp** to its next node until it reaches the location to insert new node

Set temp=temp→next
Set temp→data=location

Step 7: when location is reached

Set newnode→next=temp→next
Set temp→next=newnode

Step 8: Stop

Deletion (begin)

Step 1: Start
Step 2: Check whether list is Empty **(head == NULL)**
Step 3: If it is Empty:
Display "**List is Empty. Deletion is not possible**"
Step 4: If it is Not Empty: Define pointer temp.
Set **temp = head** (initialize temp with head)
Step 5**: Set head=temp→next**
Set tail→next=head
Step 6**: Delete temp**
free (temp)
Step 7: Stop

Deletion (end)

Step 1: Start
Step 2: Check whether list is Empty **(head == NULL)**
Step 3: If it is Empty:
Display "**List is Empty. Deletion is not possible**"
Step 4: If it is Not Empty: define pointers 'temp1' and 'temp2'
temp1 = head (initialize temp1 with head).
Step 5: set **temp2 = temp1** and move temp1 to its next node
Step 6: Repeat the same until **temp1 → next == head**

Step 7: **set temp2→next=head**
Step 8: **delete temp1**
free (temp1)

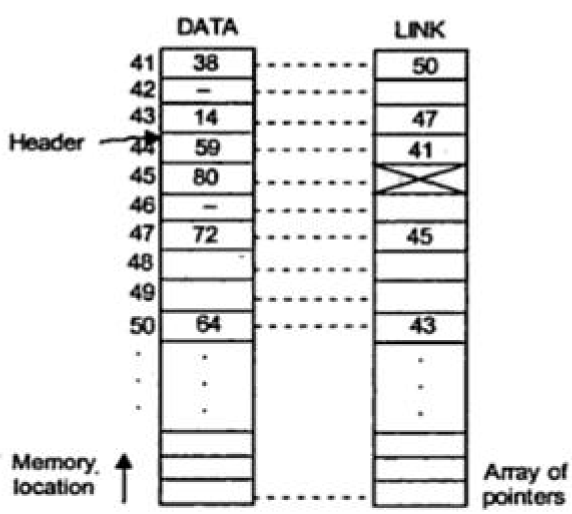
Deletion (specific location)

Step 1: Start
Step 2: Check whether list is Empty **(head == NULL)**
Step 3: If it is Empty:
Display "**List is Empty. Deletion is not possible**"
Step 4: If it is Not Empty: define pointers 'temp1' and 'temp2'
temp1 = head (initialize temp1 with head).
Step 5: set **temp2 = temp1** and move temp1 to its next node
Step 6: Repeat the same until temp1 reaches the node to delete at specific position in the list
Step 7: **Set temp2→next = temp1 → next**
Step 8: **delete temp1**
free (temp1)
Step 9: stop.

LINKED STACKS AND QUEUES

Static Representation:

Static representation maintains two arrays: one for data and other for links. Two parallel arrays of equal size are allocated which would be sufficient to store the entire linked list.
Static representation of linked lists is shown below:



Static representation of a single linked list using arrays.

Dynamic representation

The efficient way of representing a linked list is using free pool of storage. In this method, there is a memory bank and memory manager. During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the node requested and if found grants a desired block to the caller. Again, there is also another program called garbage collector, it plays whenever a node is no more in use; it returns unused node to the memory bank. Such a memory management is called Dynamic memory management.

APPLICATIONS OF LINKED LISTS:

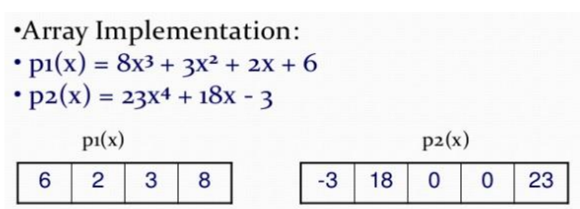
1. Sparse matrix Representation

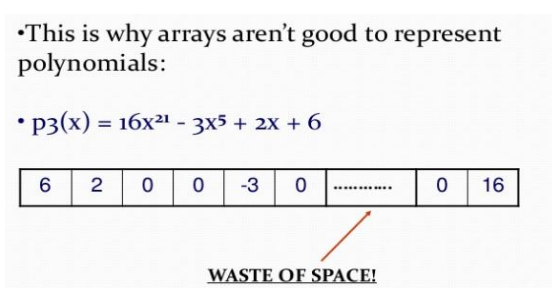
2. Polynomial representation

3. Dynamic storage management

Polynomial Representation

<u>Array implementation:</u>

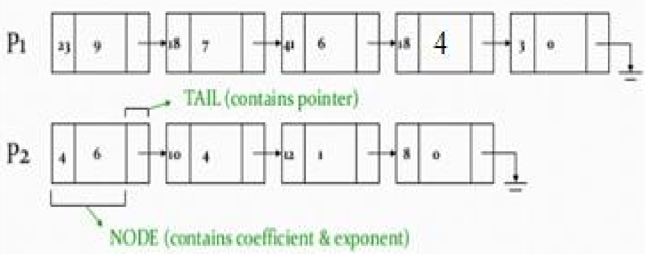




Linked list implementation:

- Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 18x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



Procedure to add polynomials using linked list

- Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

- Case 1: exponent of p1 > exponent of p2
  - Copy node of p1 to end of p3.

[go to next node]

- Case 2: exponent of p1 < exponent of p2
  - Copy node of p2 to end of p3.

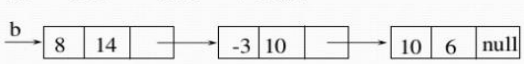
[go to next node]

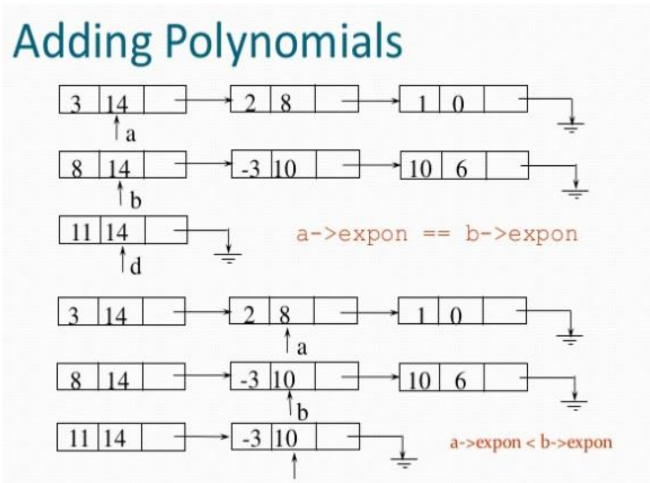
- Case 3: exponent of p1 = exponent of p2
  - Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

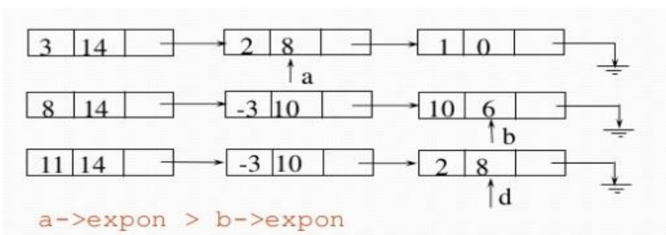
Example

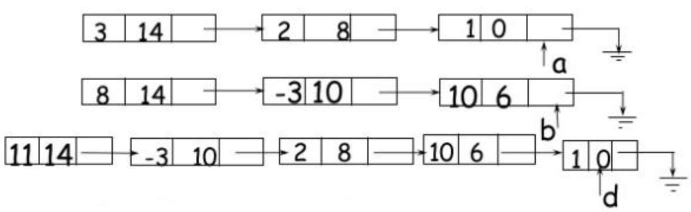
$$a = 3x^{14} + 2x^8 + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^6$$









Sparse matrix Representation

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

A sparse matrix can be represented by using TWO representations, those are as follows...

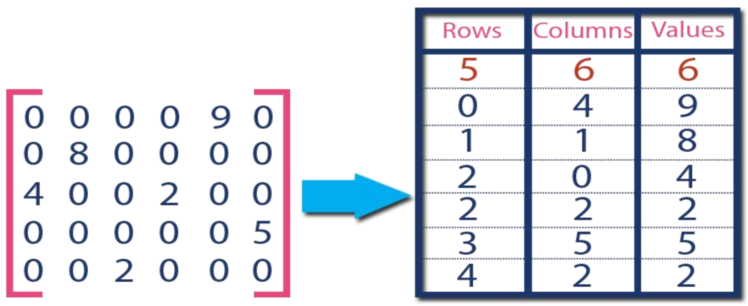
Triplet Representation

Linked Representation

Triplet Representation

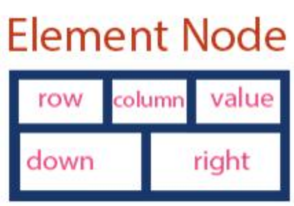
In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

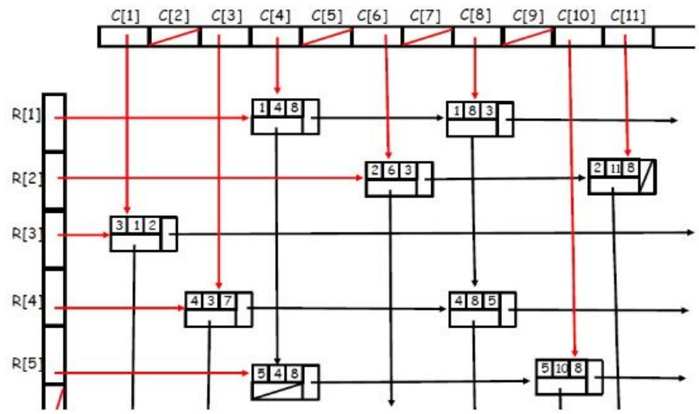


Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the image...



Example: There are two arrays **of pointers** that are the row array and column array. Each cell of the array is pointing to the respective line/column. It is as in the picture below:



Being a sparse matrix, where there are zeroes, you see those arrows as those nodes that have zeroes with them do not exist.

Dynamic memory management:

Dynamic memory management scheme is based on these principles:

- Allocation schemes
- Deallocation schemes

Allocation schemes: how request for a node will be serviced:

- Fixed block allocation
- Variable block allocation
    - First fit
    - Next fit
    - Best fit
    - Worst fit

Deallocation schemes: how to return a node to memory bank whenever it is no more required.

- Random deallocation
- Ordered deallocation

SORTING

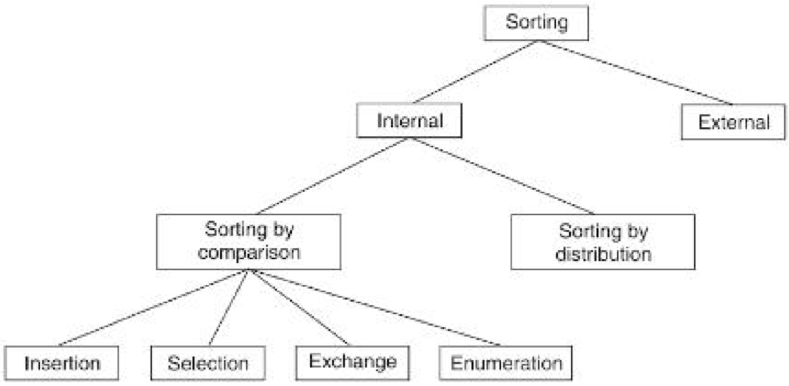
Sorting: Sorting means arranging data in a particular format (ascending or descending).

Sorting Algorithm: Sorting algorithm specifies the way to arrange data in a particular order.

Sorting techniques:

There are two types of sorting techniques. They are:

- Internal sorting
- External sorting



Internal sorting: This sorting is performed in computer main memory that is restricted to sort small set of data items.

Internal sorting techniques are based on two principles:

- Sorting by comparison
- Sorting by distribution

Sorting by comparison: A data item is compared with other items in the list of items in order to find its place in the sorted list. In this, there are four types:

- Insertion sort
- Exchange sort
- Selection sort
- Merge sort

Sorting by distribution: All items under sorting are distributed over an auxiliary storage space and then grouped together to get the sorted list. In this, there are three types:

- Radix
- Counting
- Hashing

Radix: Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

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

Counting: Items are sorted based on their relative counts

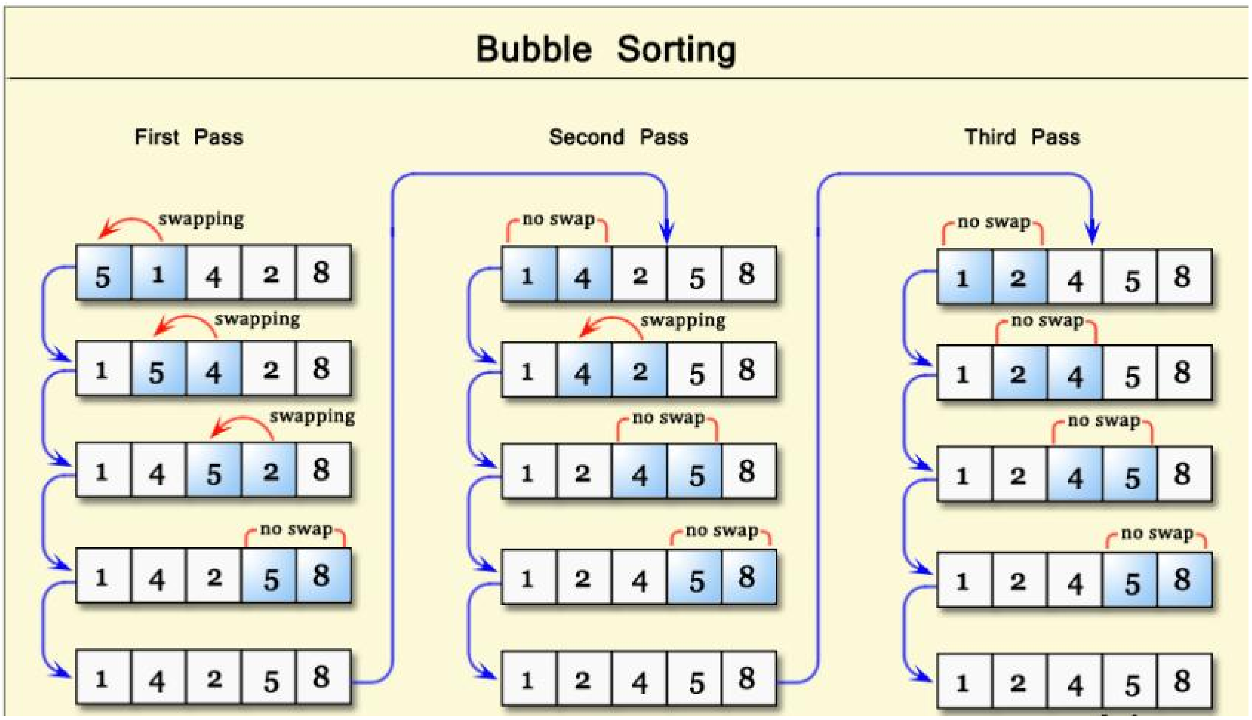
Hashing: In this method, Items are hashed into a list based on hash function.

BUBBLE SORT:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Example:

Consider the elements 5,1,4,2,8.



SELECTION SORT:

Selection sort requires n-1 pass to sort an array of n elements. In each pass we search for the smallest element from the search range and swap it with appropriate place.

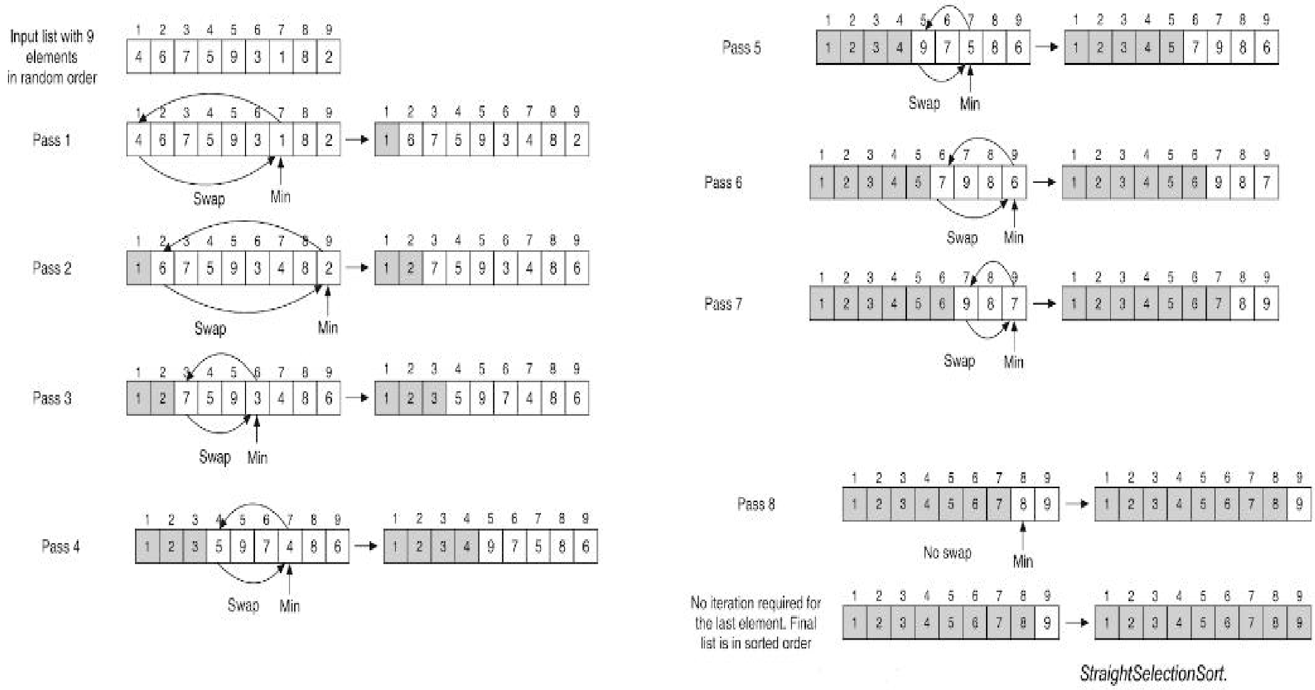
Straight selection sort:

The following are the two steps to be followed in straight selection sort:

Select: select the smallest key in the list of remaining key values say, $k_i, k_{i+1}, \ldots\ldots\ldots\ldots k_n$.

Let the smallest key value be k_j.

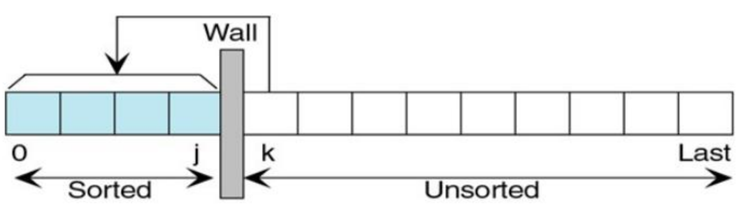
Swap: Swap the two key values k_i and k_j.



INSERTION SORT:

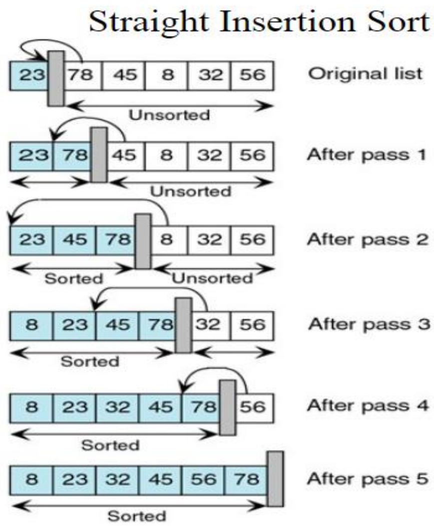
It uses two pieces of data to sort: sorted and unsorted.

In each pass, the first element of the unsorted sub-list is transferred to the sorted sub-list by insertion at appropriate place.



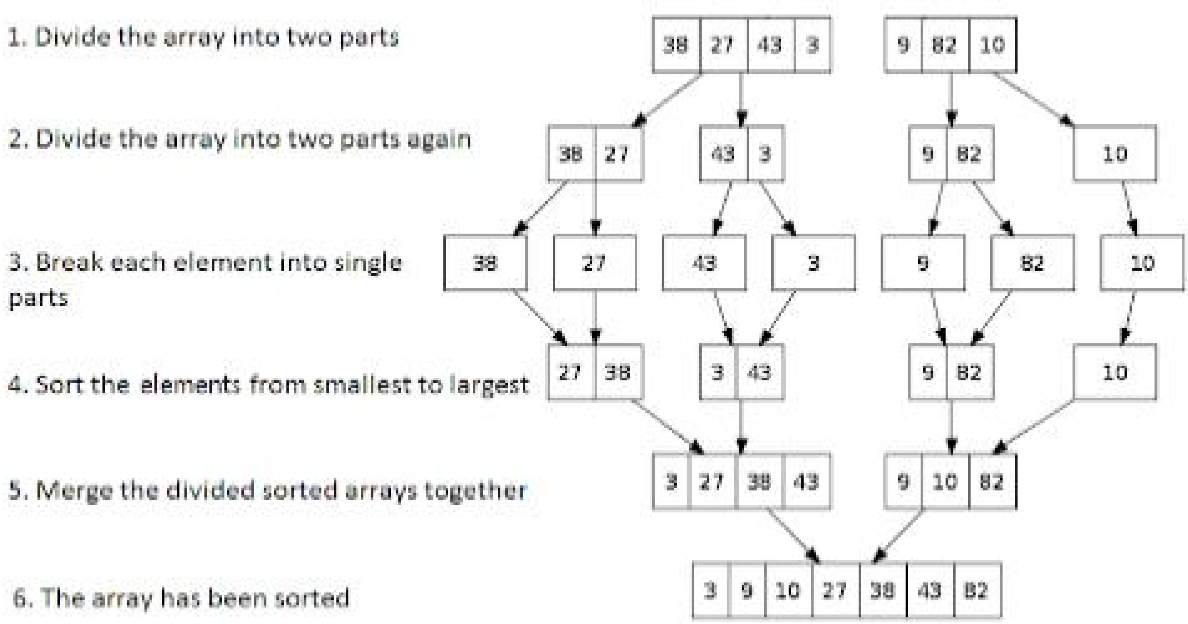
It will take at most n-1 passes to sort the data.

Example:



MERGE SORT:

Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner.



Divide and conquer:

Using the Divide and Conquer technique, we divide a problem into sub-problems. When the solution to each sub-problem is ready, we 'combine' the results from the sub-problems to solve the main problem.

Algorithm

Step 1: divide the list recursively into two halves until it can no more be divided

```
Step 2: sort the two subsequences using the algorithm
Step 3: Merge two sorted subsequences to form the output sequence.

Mergesort (list, first, last)
if(first < last)
        middle = (first + last)/2
        mergesort (list,first,middle)    // split left sublist
        mergesort (list, middle + 1, last)      // split right sublist
        merge (list, first, middle, last)
endif
```

QUICK SORT:

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- 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 values greater than the pivot value.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Algorithm

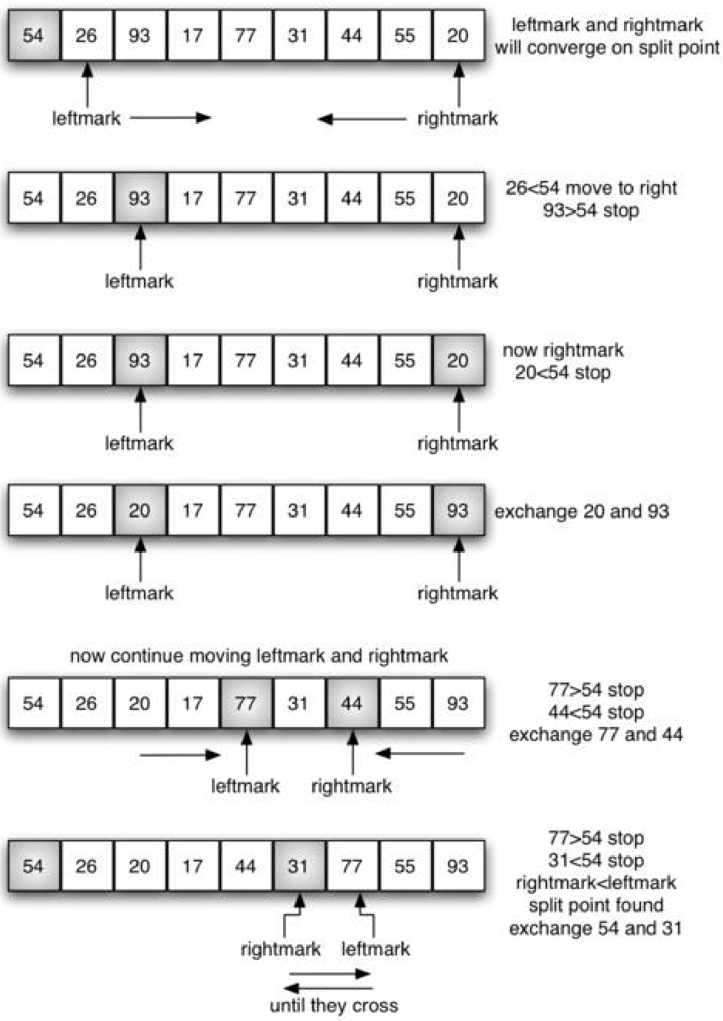
Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

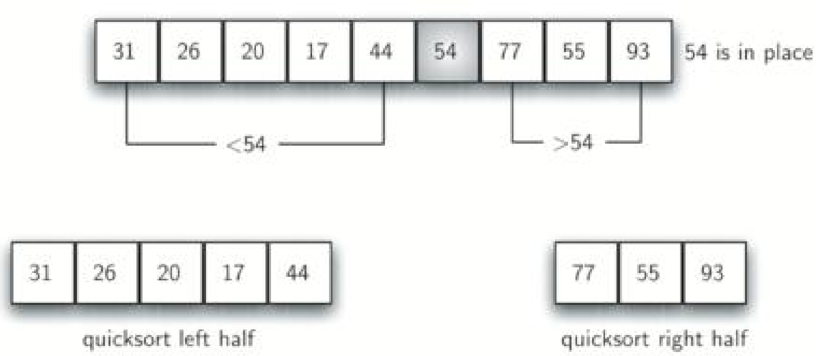
Example:

Let's consider an array with values 54, 26, 93, 17, 77, 31, 44, 55, 20

Below, we have a pictorial representation of how quick sort will sort the given array.



Now swap pivot and right mark values



Recursively Quick sort both left and right half. Final sorted list: 17, 20, 26, 31, 44, 55, 77, 93.