Unit-II

Introduction to Data Structures

What is Data Structure?

Whenever we want to work with large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

A data structure can be defined as follows...

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy

NOTE

- Every data structure is used to organize the large amount of data
- Every data structure follows a particular principle
- The operations in a data structure should not violate the basic principle of that data structure.

Based on the organizing method of a data structure, data structures are divided into two types.

Linear Data Structures

Non - Linear Data Structures

Linear Data Structures

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure.

Example

Arrays

List (Linked List)

Stack

Queue

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

Tree

Graph

Dictionaries

Heaps

Tries, Etc.,

Single Linked List

What is Linked List?

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

What is Single Linked List?

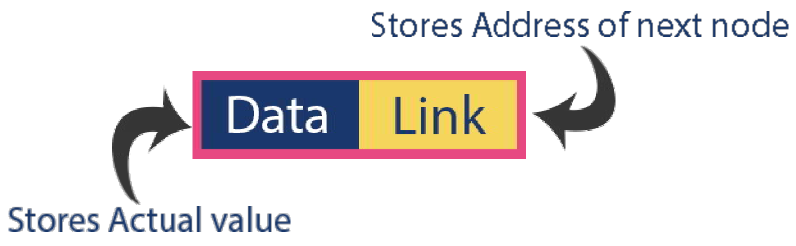
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

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.

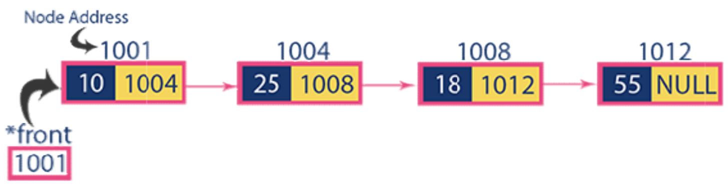
The graphical representation of a node in a single linked list is as follows...



NOTE

☀In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").
☀Always next part (reference part) of the last node must be NULL.

Example



Operations

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

Insertion

Deletion

Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.

Step 2: Declare all the user defined functions.

Step 3: Define a Node structure with two members data and next

Step 4: Define a Node pointer 'head' and set it to NULL.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

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

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, set newNode→next = NULL and head = newNode.

Step 4: If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.

Step 2: Check whether list is Empty (head == NULL).

Step 3: If it is Empty then, set head = newNode.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6: Set temp → next = newNode.

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

We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, set newNode → next = NULL and head = newNode.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7: Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion

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

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked 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' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → next == NULL)

Step 5: If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE then set head = temp → next, and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked 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' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Check whether list has only one Node (temp1 → next == NULL)

Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next== NULL)

Step 7: Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked 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' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8: If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9: If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11: If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Circular Linked List

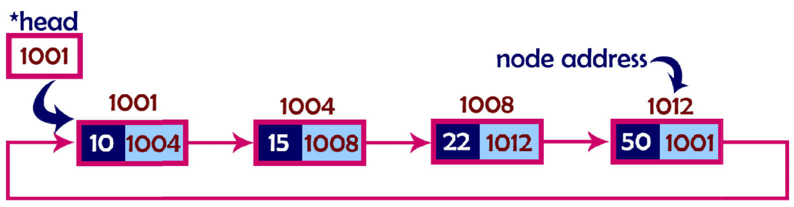
What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the 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.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations

In a circular linked list, we perform the following operations...

Insertion

Deletion

Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.

Step 2: Declare all the user defined functions.

Step 3: Define a Node structure with two members data and next

Step 4: Define a Node pointer 'head' and set it to NULL.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

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

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, set head = newNode and newNode→next = head .

Step 4: If it is Not Empty then, define a Node pointer 'temp' and initialize with 'head'.

Step 5: Keep moving the 'temp' to its next node until it reaches to the last node (until 'temp → next == head').

Step 6: Set 'newNode → next =head', 'head = newNode' and 'temp → next = head'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL).

Step 3: If it is Empty then, set head = newNode and newNode → next = head.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next == head).

Step 6: Set temp → next = newNode and newNode → next = head.

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

We can use the following steps to insert a new node after a node in the circular linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, set head = newNode and newNode → next = head.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp is reached to the last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7: If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8: If temp is last node then set temp → next = newNode and newNode → next = head.

Step 8: If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Deletion

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

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked 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' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head.

Step 4: Check whether list is having only one node (temp1 → next == head)

Step 5: If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)

Step 6: If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)

Step 7: Then set head = temp2 → next, temp1 → next = head and delete temp2.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked 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' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Check whether list has only one Node (temp1 → next == head)

Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)

Step 7: Set temp2 → next = head and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked 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' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node (temp1 → next == head)

Step 7: If list has only one node and that is the node to be deleted then set head = NULL and delete temp1 (free(temp1)).

Step 8: If list contains multiple nodes then check whether temp1 is the first node in the list (temp1 == head).

Step 9: If temp1 is the first node then set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.

Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).

Step 11: If temp1 is last node then set temp2 → next = head and delete temp1 (free(temp1)).

Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to head → data.

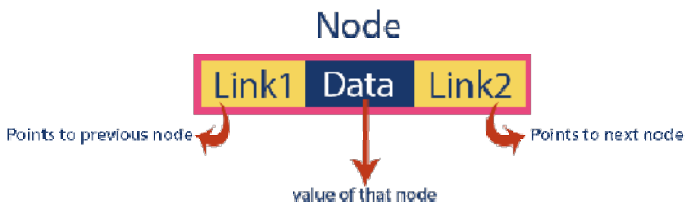
Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

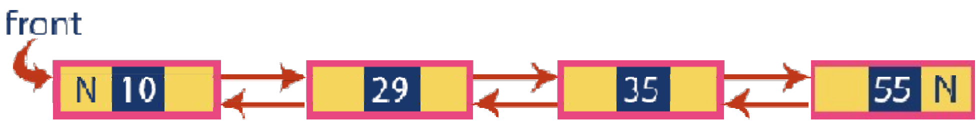
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

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...



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

Example



NOTE

☀ 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.

Operations

In a double linked list, we perform the following operations...

Insertion

Deletion

Display

Insertion

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

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1: Create a newNode with given value and newNode → previous as NULL.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4: If it is not Empty then, assign head to newNode → next and newNode to head.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty, then assign NULL to newNode → previous and newNode to head.

Step 4: If it is not Empty, then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6: Assign newNode to temp → next and temp to newNode → previous.

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

We can use the following steps to insert a new node after a node in the double linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → previous & newNode → next and newNode to head.

Step 4: If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.

Step 5: Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step 7: Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deletion

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

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked 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' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → previous is equal to temp → next)

Step 5: If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked 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' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5: If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7: Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked 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' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5: If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).

Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).

Step 9: If temp is the first node, then move the head to the next node (head = head → next),
set head of previous to NULL (head → previous = NULL) and delete temp.

Step 10: If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).

Step 11: If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).

Step 12: If temp is not the first node and not the last node, then
set temp of previous of next to temp of next (temp → previous → next = temp →

next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Display 'NULL <--- '.

Step 5: Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node

Step 6: Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Arrays

What is an Array?

Whenever we want to work with large number of data values, we need to use that much number of different variables. As the number of variables are increasing, complexity of the program also increases and programmers get confused with the variable names. There may be situations in which we need to work with large number of similar data values. To make this work more easy, C programming language provides a concept called "Array".

An array is a variable which can store multiple values of same data type at a time.

An array can also be defined as follows...

"Collection of similar data items stored in continuous memory locations with single name".

To understand the concept of arrays, consider the following example declaration.

int a, b, c;

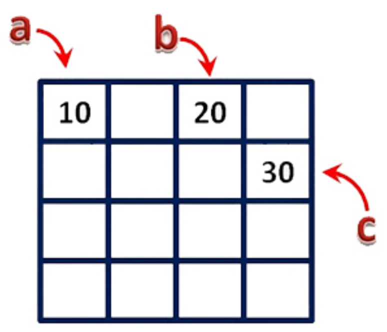
Here, the compiler allocates 2 bytes of memory with name 'a', another 2 bytes of memory with name 'b' and more 2 bytes with name 'c'. These three memory locations are may be in sequence or may not be in sequence. Here these individual variables can store only one value at a time.

In computer memory is organized as shown in figure. Here assume that each box is of 2 bytes of memory.

2 byte for ‘a’, another 2 bytes for ‘b’ and 2 more bytes for ‘c’.

If we assign following values they will inserted into that memory locations.

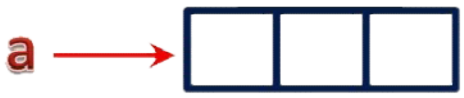
```
a = 10;
b = 20;
c = 30;
```



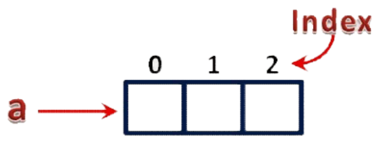
Now consider the following declaration...

int a[3];

Here, the compiler allocates total 6 bytes of continuous memory locations with single name 'a'. But allows to store three different integer values (each in 2 bytes of memory) at a time. And memory is organized as follows...



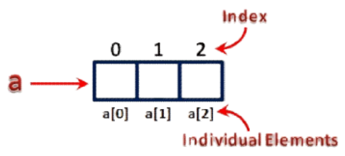
That means all these three memory locations are named as 'a'. But "how can we refer individual elements?" is the big question. Answer for this question is, compiler not only allocates memory, but also assigns a numerical value to each individual element of an array. This numerical value is called as "Index". Index values for the above example are as follows...



The individual elements of an array are identified using the combination of 'name' and 'index' as follows...

arrayName[indexValue]

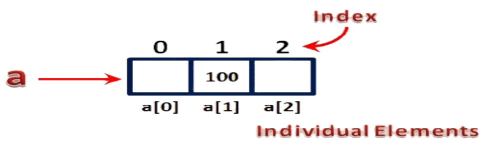
For the above example the individual elements can be referred to as follows...



If I want to assign a value to any of these memory locations (array elements), we can assign as follows...

a[1] = 100;

The result will be as follows...



There are different types of arrays in c programming language. Click on the below link to read TYPES OF ARRAYS

Sparse Matrix

What is Sparse Matrix?

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.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

Sparse Matrix Representations

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

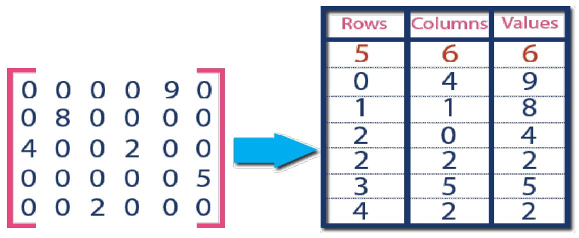
Triplet Representation

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...



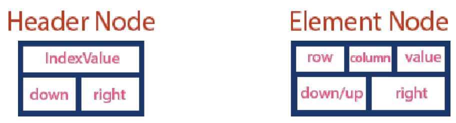
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

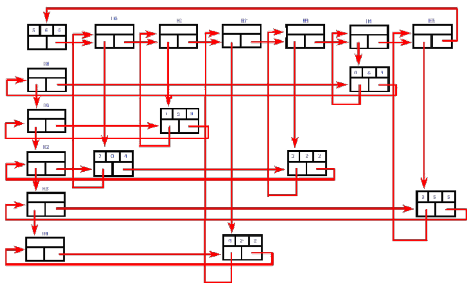
In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

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...



Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.