

MDOULE 1: DATA REPRESENTATION, SEARCHING AND ARRAYS

1.1 Data Representation

1.1.1 Introduction

A data structure is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations.

Data can be organized in different ways. The logical and mathematical model of a particular organization of data is called as a data structure.

The choice of a particular data model depends on two considerations.

- It must be rich enough in structure to mirror the actual relationships of the data in the real world.
- The structure should be simple enough that one can effectively process the data when necessary.

In programming, the term structure refers to scheme for organizing related pieces of information. The basic data structures include arrays, structures, stacks, queues, linked lists, trees and graphs etc.

The data structures can be categorized into

- linear and
- non-linear data structures.

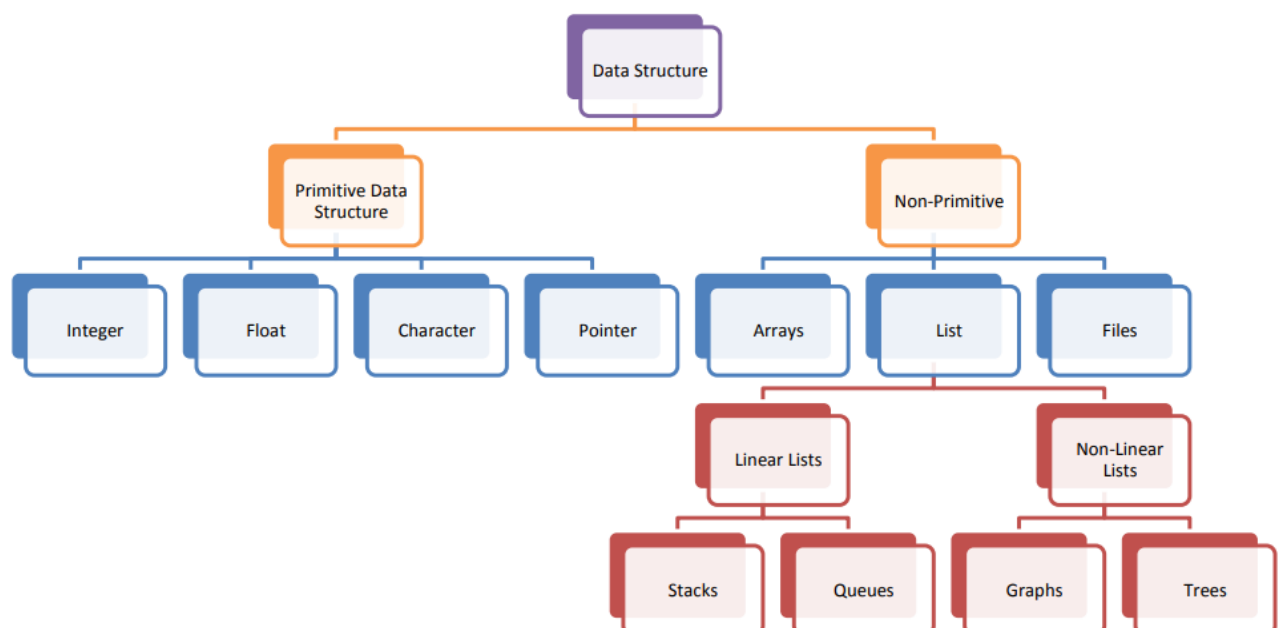


Fig 1.1: Types of Data Structures

Primitive Data Structure: - The data structure that are atomic or indivisible are called primitive. Example are integer, real, float, Boolean and characters.

Non-Primitive data structure: - The data structure that are not atomic are called non primitive or composite. Examples are records, arrays and strings. These are more sophisticated. The non-primitive data structures emphasize on structuring f a group of homogenous or heterogeneous data items.

Linear Data Structure:- In a linear data structure, the data items are arranged in a linear sequence. Example is array.

Non Linear data structure: - In a non-linear data structure, the data items are not in sequence. Example is tree.

1.1.2 Linear Lists

In general, a linear list is a data object whose values are of the form (e_1, e_2, \dots, e_n) , where e_i terms are the elements of the list, and n , a finite number, is its length.

When $n = 0$, the list is empty. Otherwise, e_1 is the first element, and e_n is the last one. For any other i , e_i precedes e_{i+1} . This is called the precedence relation for the linear list type.

The linear list type is used a lot in real life, e.g., an alphabetized list of students, a list of test scores, etc.

We immediately have the following list of functions, thinking about these applications: create a list; determine if a list is empty; find out the length of a list; find out the k th element in a list; Search for a given element; delete an element in a list; and insert another element in a list, etc.

Any data type can be specified as an Abstract Data Type, which is independent of any representation.

The List ADT Instances:

Instances: ordered finite collection of zero or more elements.

Operations: Create() constructs an empty list. Destroy() erases the list.

Is empty() returns true if the list is empty, otherwise, returns false.

Length() returns the size of the list.

Find(k, x) returns the k th element of the list, and put it in x ; returns false, if the size is smaller than k .
void Search(x) returns the position of x .

Delete(k, x) deletes the k th element of the list, and put it in x ; returns the modified list.

Insert(k, x) adds x just after k th element.

Output(out) puts the list into the output stream out.

Formula-based representation

A formula-based representation uses an array to represent the instances of an object. Each position of the array, called a cell or a node, holds one element that makes up an instance of that object. Individual elements of an instance are located in the array, based on a mathematical formula, e.g., a simple and often used formula is

$$\text{Location}(i) = i - 1,$$

which says the i th element of the list is in position $i - 1$. We also need two more variables, `length` and `MaxSize`, to completely characterize the list type.

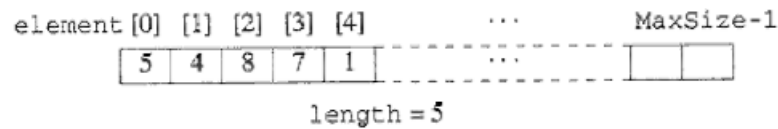


Fig 1.2: Linear List Representation

1.1.3 Array Based Representation & Operations

Array is a container which can hold a fix number of items and these items should be of the same type i.e., either *int* or *float* or *char*, etc. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of array.

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.



Fig 1.3: Array Representation

Let's consider array declaration in C. In the figure 1.3, an array of 10 integers can be declared by:



Fig 1.4: Array Declaration

As per the figure 1.3 and 1.4, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.

- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations:

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

In C, when an array is initialized with size, then it assigns default values to its elements in following order:

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array:

Algorithm

Let Array be a linear unordered array of MAX elements.

Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that $K \leq N$.

Following is the algorithm where ITEM is inserted into the K^{th} position of LA –

1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop

Following is the implementation of the above algorithm –

```
1. #include <stdio.h>
2. main() {
3.     int LA[] = {
4.         1, 3, 5, 7, 8
5.     };
6.     int item = 10, k = 3, n = 5;
7.     int i = 0, j = n;
8.     printf("The original array elements are :\n");
9.     for (i = 0; i < n; i++) {
10.        printf("LA[%d] = %d \n", i, LA[i]);
11.    }
12.    n = n + 1;
13.    while (j >= k) {
14.        LA[j + 1] = LA[j];
15.        j = j - 1;
16.    }
17.    LA[k] = item;
18.    printf("The array elements after insertion :\n");
19.    for (i = 0; i < n; i++) {
20.        printf("LA[%d] = %d \n", i, LA[i]);
21.    }
22. }
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$. Following is the algorithm to delete an element available at the K^{th} position of LA.

1. Start
2. Set $J = K$
3. Repeat steps 4 and 5 while $J < N$
4. Set $LA[J] = LA[J + 1]$
5. Set $J = J + 1$
6. Set $N = N - 1$
7. Stop

Following is the implementation of the above algorithm:

```
1. #include <stdio.h>
2. main() {
3.     int LA[] = {
4.         1, 3, 5, 7, 8
5.     };
6.     int k = 3, n = 5;
7.     int i, j;
8.     printf("The original array elements are :\n");
9.     for (i = 0; i < n; i++) {
10.        printf("LA[%d] = %d \n", i, LA[i]);
11.    }
12.    j = k;
13.    while (j < n) {
14.        LA[j - 1] = LA[j];
15.        j = j + 1;
16.    }
17.    n = n - 1;
18.    printf("The array elements after deletion :\n");
19.    for (i = 0; i < n; i++) {
20.        printf("LA[%d] = %d \n", i, LA[i]);
21.    }
22. }
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that $K \leq N$. Following is the algorithm to find an element with a value of **ITEM** using sequential search.

1. Start
2. Set $J = 0$
3. Repeat steps 4 and 5 while $J < N$
4. IF $LA[J]$ is equal **ITEM** THEN GOTO STEP 6
5. Set $J = J + 1$
6. PRINT J , **ITEM**
7. Stop

Following is the implementation of the above algorithm –

```
1. #include <stdio.h>
2. main() {
3.     int LA[] = {
4.         1, 3, 5, 7, 8
5.     };
6.     int item = 5, n = 5;
7.     int i = 0, j = 0;
8.     printf("The original array elements are :\n");
9.     for (i = 0; i < n; i++) {
10.        printf("LA[%d] = %d \n", i, LA[i]);
11.    }
12.    while (j < n) {
13.        if (LA[j] == item) {
14.            break;
15.        }
16.        j = j + 1;
17.    }
18.    printf("Found element %d at position %d\n", item, j + 1);
19. }
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that $K \leq N$. Following is the algorithm to update an element available at the K^{th} position of **LA**.

1. Start
2. Set $LA[K-1] = \text{ITEM}$

3. Stop

Following is the implementation of the above algorithm –

```

1. #include <stdio.h>
2. main() {
3.     int LA[] = {
4.         1, 3, 5, 7, 8
5.     };
6.     int k = 3, n = 5, item = 10;
7.     int i, j;
8.     printf("The original array elements are :\n");
9.     for (i = 0; i < n; i++) {
10.        printf("LA[%d] = %d \n", i, LA[i]);
11.    }
12.    LA[k - 1] = item;
13.    printf("The array elements after updation :\n");
14.    for (i = 0; i < n; i++) {
15.        printf("LA[%d] = %d \n", i, LA[i]);
16.    }
17. }

```

1.1.4 Indirect Addressing

This approach combines the formula-based approach and that of the linked representation. As a result, we can not only get access to elements in more than 1 times, but also have the storage flexibility, elements will not be physically moved during insertion and/or deletion. In indirect addressing, we use a table of pointers to get access to a list of elements, as shown in the figure 1.5

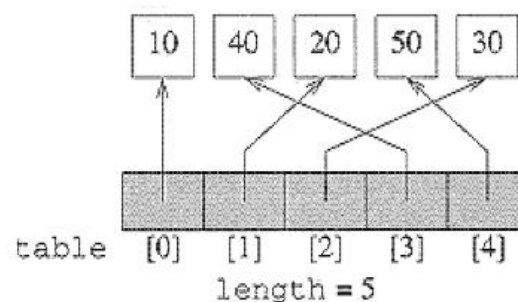


Fig 1.5: Indirect Addressing

1.1.5 Linked Representation

The array-based representation of list has following limitations:

1. size of the list must be known in advance.
2. We may come across situations when an attempt to add an element cause overflow. However, list, as an abstract data structure cannot be full. Hence abstractly, it is always possible to add an element onto the list. Therefore, representing list as an array prohibits the growth of list beyond the finite number of elements.

Therefore, some dynamic structure is required to represent list.

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List:

- Link – Each link of a linked list can store a data called an element.
- Next – Each link of a linked list contains a link to the next link called Next.
- LinkedList – A Linked List contains the connection link to the first link called First.

Linked list can be visualized as a chain of nodes, where every node points to the next node.



Fig 1.5: Linked List Representation

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Some of the basic operations supported by linked lists are:

- Insertion – Adds an element at the beginning of the list.
- Deletion – Deletes an element at the beginning of the list.
- Display – Displays the complete list.
- Search – Searches an element using the given key.
- Delete – Deletes an element using the given key.

1.1.6 Comparing Linear and Linked Representations

When we compare two implementations of list array and Linked, we consider certain factors. So which implementation is better? The answer, as usual, is: It depends. The linked implementation certainly gives more flexibility, and in applications where the number of list items can vary greatly, it wastes less space when the list is small. In situations where the list size is totally unpredictable, the linked implementation is preferable, because size is largely irrelevant. Why, then, would we ever want to use an array-based implementation? Because it is short, simple, and efficient. If insertion and deletion occur frequently, the array-based implementation executes more quickly because it does not incur the run-time overhead of the new and delete operations.

Overall, the three list implementations (Static Array, Dynamic array and Linked representation) are roughly equivalent in terms of the amount of work they do, differing in only one of the five operations and the class destructor (which is used in linked representation for destroying the pointers and nodes after use).

1.2 Searching

1.2.1 Linear Search or Sequential Search

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear search is used on a collection of items. It relies on the technique of traversing a list from start to end by exploring properties of all the elements that are found on the way. Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

A linear search scans one item at a time, without jumping to any item.

- The worst-case complexity is $O(n)$, sometimes known as $O(n)$ search.
- Time taken to search elements keep increasing as the number of elements are increased.

Linear search is implemented using following steps:

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and terminate the function

Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Implementation of Linear Search

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main() {
4.     int list[20], size, i, sElement;
5.     printf("Enter size of the list: ");
6.     scanf("%d", &size);
7.     printf("Enter any %d integer values: ", size);
8.     for (i = 0; i < size; i++) scanf("%d", &list[i]);
9.     printf("Enter the element to be Search: ");
10.    scanf("%d", &sElement); // Linear Search Logic
11.    for (i = 0; i < size; i++) {
12.        if (sElement == list[i]) {
13.            printf("Element is found at %d index", i);
14.            break;
15.        }
16.    }
17.    if (i == size) printf("Given element is not found in the list!!!");
18.    getch();
19. }
```

Analysis of Linear Search:

Whether the linear search is carried out on lists implemented as arrays or linked lists or on files, the performance will be affected by the comparison statement: $\text{if}(a[i]==k)$. Obviously, the fewer the number of comparisons, the sooner the algorithm will terminate. Now, look at the following:

- The number of minimum possible comparisons is 1 when the required item is the first item in the list.
- The number of maximum comparisons is N when the required item is the last item in the list.

Thus, if the required item is in position 1 in the list, then 1 comparison is required.

Hence, the average number of comparisons required by linear search is:

$$(1+2+3+\dots+N)/N = [N(N+1)]/(2 * N) = (N+1)/2$$

Time Complexity

Worst Case : $O(n)$

Best Case : $\Omega(1)$

Average Case : $\Theta(n)$

Advantages:

- This is very easy to implement.
- Well suited for short lists as the complexity is of $O(N)$.

Disadvantage:

- This will become disastrous if your list is very long. This is because the time required to find out an element using linear search is directly proportional to the size of list (i.e., the number of elements in the list).

1.2.2 Binary Search:

Binary search algorithm finds given element in a list of elements with $O(\log n)$ time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of elements. That means, binary search can be used only with list of elements which are already arranged in an order. The binary search cannot be used for list of elements which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element. If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps:

Step 1: Read the search element from the user

Step 2: Find the middle element in the sorted list

Step 3: Compare, the search element with the middle element in the sorted list.

Step 4: If both are matching, then display "Given element found!!!" and terminate the function

Step 5: If both are not matching, then check whether the search element is smaller or larger than middle element.

Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.

Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until sub list contains only one element.

Step 9: If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Implementation of Binary Search

```

1. #include <stdio.h>
2. #include <conio.h>
3. void main() {
4.     int first, last, middle, size, i, sElement, list[100];
5.     clrscr();
6.     printf("Enter the size of the list: ");
7.     scanf("%d", &size);
8.     printf("Enter %d integer values in Assending order\n", size);
9.     for (i = 0; i < size; i++) scanf("%d", &list[i]);
10.    printf("Enter value to be search: ");
11.    scanf("%d", &sElement);
12.    first = 0;
13.    last = size - 1;
14.    middle = (first + last) / 2;
15.    while (first <= last) {
16.        if (list[middle] < sElement) first = middle + 1;
17.        else if (list[middle] == sElement) {
18.            printf("Element found at index %d.\n", middle);
19.            break;
20.        } else last = middle - 1;
21.        middle = (first + last) / 2;
22.    }
23.    if (first > last) printf("Element Not found in the list.");
24.    getch();
25. }
```

Analysis of Binary Search:

The binary search method needs no more than $\lceil \log_2 n \rceil + 1$ comparisons. Contrast this with the case of linear search which on the average will need $(n+1)/2$ comparison. This implies that

for an array of million entries, only about 20 comparisons needed for binary search while a million comparisons are required for linear search.

Time Complexity

Worst Case : $O(\log n)$

Best Case : $\Omega(1)$

Average Case : $\Theta(\log n)$

1.3 Arrays and Matrices

1.3.1 Arrays, Matrices

In C, we can define multidimensional matrices in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:

data_type array_name[size1][size2]....[sizeN];

data_type: Type of data to be stored in the array.

Here data_type is valid C/C++ data type

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions

Size of multidimensional arrays:

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array `int x[10][20]` can store total $(10*20) = 200$ elements.

Similarly array `int x[5][10][20]` can store total $(5*10*20) = 1000$ elements.

Two – dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one – dimensional array for easier understanding.

- The basic form of declaring a two-dimensional array of size x, y:
Syntax:

data_type array_name[x][y];

data_type: Type of data to be stored. Valid C data type.

- We can declare a two-dimensional integer array say 'x' of size 10,20 as:

`int x[10][20];`

- Elements in two-dimensional arrays are commonly referred by $x[i][j]$ where i is the row number and ' j ' is the column number.
- A two – dimensional array can be seen as a table with ' x ' rows and ' y ' columns where the row number ranges from 0 to $(x-1)$ and column number ranges from 0 to $(y-1)$. A two – dimensional array ' x ' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	$x[0][0]$	$x[0][1]$	$x[0][2]$
Row 1	$x[1][0]$	$x[1][1]$	$x[1][2]$
Row 2	$x[2][0]$	$x[2][1]$	$x[2][2]$

Initializing Two – Dimensional Arrays:

There are two ways in which a Two-Dimensional array can be initialized.

- First Method:

```
int x[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

The above array have 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on.

- Better Method:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays:

Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

```
int x[2][1];
```

The above example represents the element present in third row and second column.

Note:

In arrays if size of array is N . Its index will be from 0 to $N-1$. Therefore, for row index 2 row number is $2+1 = 3$.

1.3.2 Sparse Matrices

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with ' m ' columns and ' n ' rows represents a $m \times n$ 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.

Why to use Sparse Matrix instead of simple matrix?

Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements.

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 $100 \times 100 \times 2 = 20000$ bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.


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

- Triplet(Array) Representation.
- Linked Representation.

Triplet(Array) 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.



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

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:

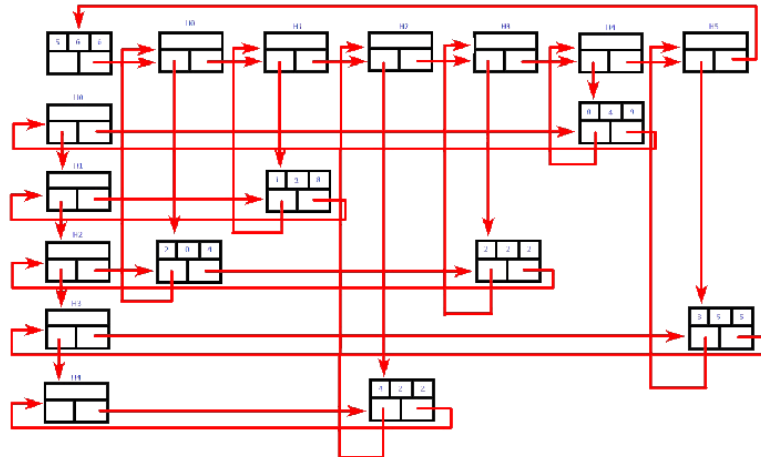
Header Node



Element Node



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.