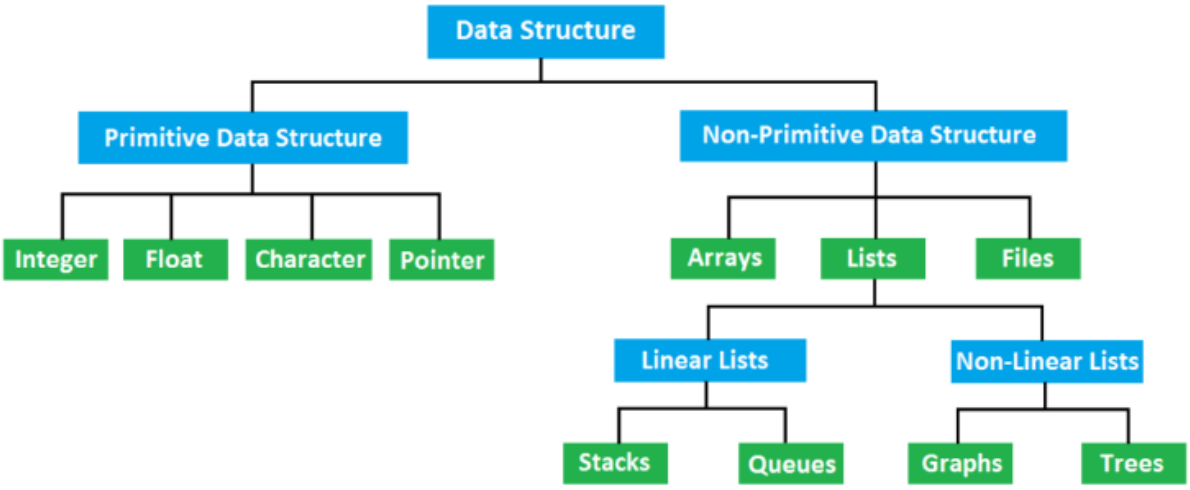
Data Structures Using C++

Unit I

Introduction

A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, Linked List, Stack, Queue, Tree, Graph etc. are all data structures that stores the data in a special way so that we can access and use the data efficiently. Each of these mentioned data structures has a different special way of organizing data so we choose the data structure based on the requirement. So, data structure is a collection of data elements and its manipulation functions.

Classification of data structures



Data structure is basically classified into two categories-

- Primitive Data type
- Non- primitive data type

But often Data structures in C++ are classified by their characteristics of storing data. Possible characteristics are:

1-**Linear or non-linear**: This characteristic describes whether the data items are arranged in **ordered sequence**, such as with an **array**, or in an **unordered sequence**, such as with a **graph**.

2-**Homogeneous or non-homogeneous**: This characteristic describes whether all data items in a given repository are of the same data type or of various types. for example- **An array has similar data type stored contiguously but a Structure has collection of different data types**.

3-**Static or dynamic**: This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time. Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

Types of data structures

Data structure types are determined by what types of operations can be performed on them and what kinds of algorithms are going to be used with. These types include:

1- Arrays

An array stores a collection of items at adjoining memory locations. Items that are the same data type get stored together so that the position of each element can be calculated or retrieved easily. Arrays can be fixed or flexible in length.

2- Linked lists

A linked list stores a collection of items in a linear order. Each element, or node, in a linked list contains a data item as well as a reference, or link, to the next item in the list.In simple word A linked list is a linear data structure where each element is a separate object. And unlike Array it is flexible in nature.

3- Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). So we can only follow these orders to get our data.

4- Queue

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for some resource where the consumer that came first is served first.

5-Trees

A tree stores a collection of items in an abstract, hierarchical way. Each node is linked to other nodes and can have multiple sub-values, also known as children.

6-Graphs

A graph stores a collection of items in a non-linear fashion. Graphs are made up of a finite set of nodes, also known as vertices, and lines that connect them, also known

as edges. These are useful for representing real-life systems such as computer networks.

7- Heap

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Now since we know what are data types and how values are stored and what are various type of storing data let get started with this series of learning about how to code different data structures in C. So let's get started from the next blog under this category. Till then stay tuned.

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Advantages of DS

We need data structures because there are several advantages of using them, few of them are as follows:

1. **Data Organization**: We need a proper way of organizing the data so that it can access efficiently when we need that particular data. DS provides different ways of data organization so we have options to store the data in different data structures based on the requirement.

2. **Efficiency**: The main reason we organize the data is to improve the efficiency. We can store the data in arrays then why do we need linked lists and other data structures? Because when we need to perform several operations such as add, delete update and search on arrays, it takes more time in arrays than some of the other data structures. So the fact that we are interested in other data structures is because of the efficiency.

Arrays

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

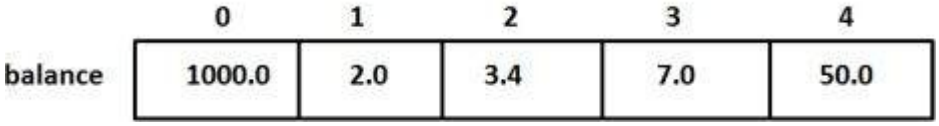
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance [4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

Advantages

It is better and convenient way of storing the data of same data type with same size.

It allows us to store known number of elements in it.

It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.

Iterating the array using their index is faster compared to any other methods like linked list etc.

It allows storing the elements in any dimensional array – supports multidimensional array.

Limitations

It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more number of records than declared then it is not possible. We should know array size at the compile time itself.

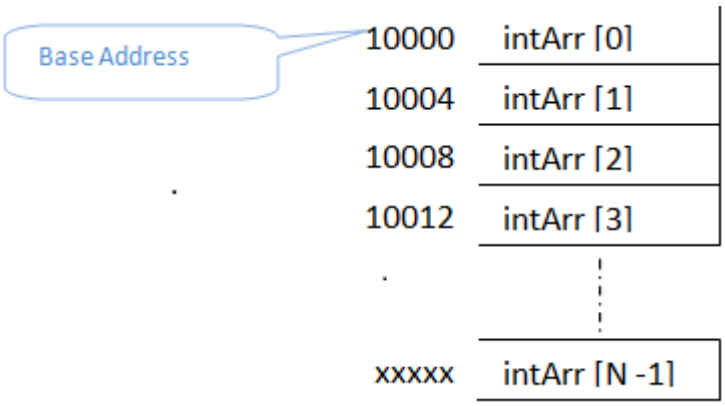
Inserting and deleting the records from the array would be costly since we add / delete the elements from the array; we need to manage memory space too.

It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.

Memory representation of arrays

One Dimensional Array

Below diagram shows how memory is allocated to a float array of N elements. Its base address – address of its first element is 10000. Since it is an integer array, each of its elements will occupy 4 bytes of space. Hence first element occupies memory from 10000 to 10003. Second element of the array occupies immediate next memory address in the memory, i.e.; 10004 which requires another 4 bytes of space. Hence it occupies from 10004 to 10007. In this way all the N elements of the array occupies the memory space.



If the array is a character array, then its elements will occupy 1 byte of memory each. If it is an integer array then its elements will occupy 2 bytes of memory each. But this is not the total size or memory allocated for the array. They are the sizes of individual elements in the array. If we need to know the total size of the array, then we need to multiply the number of elements with the size of individual element.

i.e.; Total memory allocated to an Array = Number of elements * size of one element

Address of intArr[i]=Base address of intArr[i] + i*sizeof the data element

Multidimensional Array

Two Dimensional Arrays

Two dimensional arrays are one dimensional array one dimensional arrays. In the case of multidimensional array, we have elements in the form of rows and columns. Here also memories allocated to the array are contiguous. But the elements assigned to the memory location depend on the two different methods:

Eg. a[0][0] a[0][1] a[0][2]

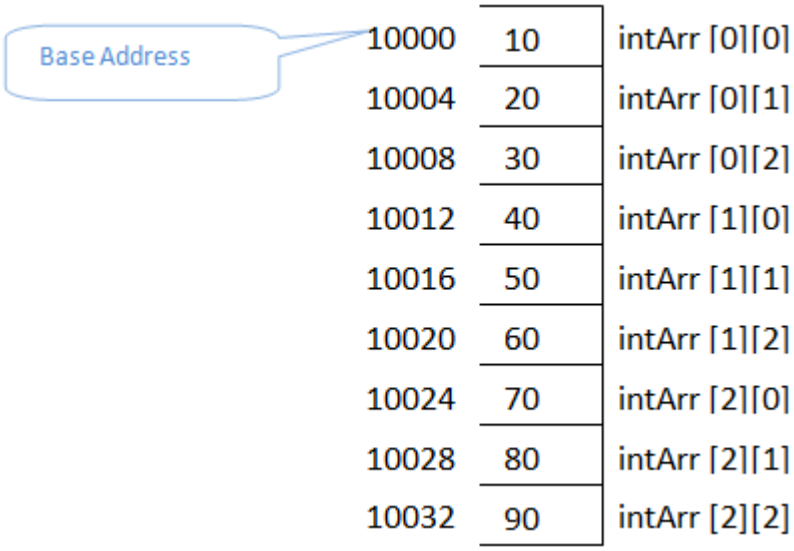
a[1][0] a[1][1] a[1][2]

a[2][0] a[2][1] a[2][2]

Row Major Order

Let us consider a two dimensional array to explain how row major order way of storing elements works. In the case of 2D array, its elements are considered as rows and columns of a table. When we represent an array as int Arr[i][j], the first index of it represents the row elements and the next index represents the column elements of each row. When we store the array elements in row major order, first we will store the elements of first row followed by second row and so on. Hence in the memory we can find the elements of first row followed by second row and so on. In memory

there will not be any separation between the rows. We have to code in such a way that we have to count the number of elements in each row depending on its column index. But in memory all the rows and their columns will be contiguous. Below diagram will illustrate the same for a 2D array of size 3X3 i.e.; 3 rows and 3 columns.



Total memory allocated to 2D Array = Number of elements * size of one element

= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes

= 10*10 * 4 bytes = 400 Bytes, where M =N = 10

= 500*5 *4 bytes= 10000 Bytes, where M=500 and N= 5

So, the address of a particular location is

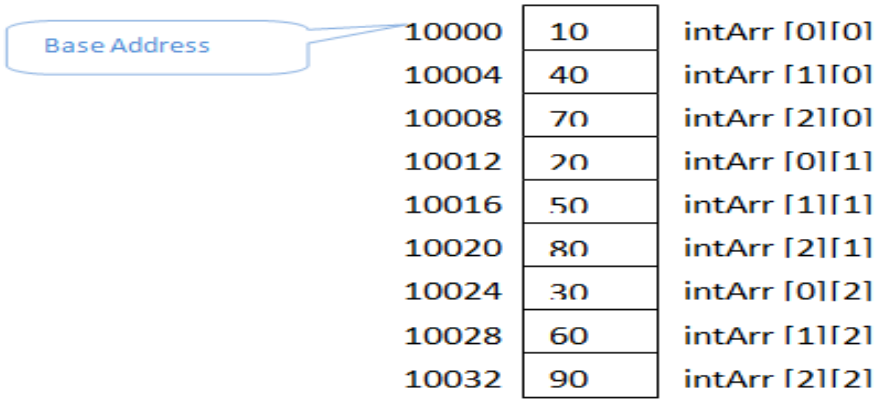
Address of intArr[i][j]=Base address of intArr + (i*N+j)*sizeof data element

Where i & J are corresponding row & column of 2D and M & N are the size of the 2D.

Column Major Order

This is the opposite method of row major order of storing the elements in the memory. In this method all the first column elements are stored first, followed by second column elements and so on.

Total memory allocated to 2D Array = Number of elements * size of one element



= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes

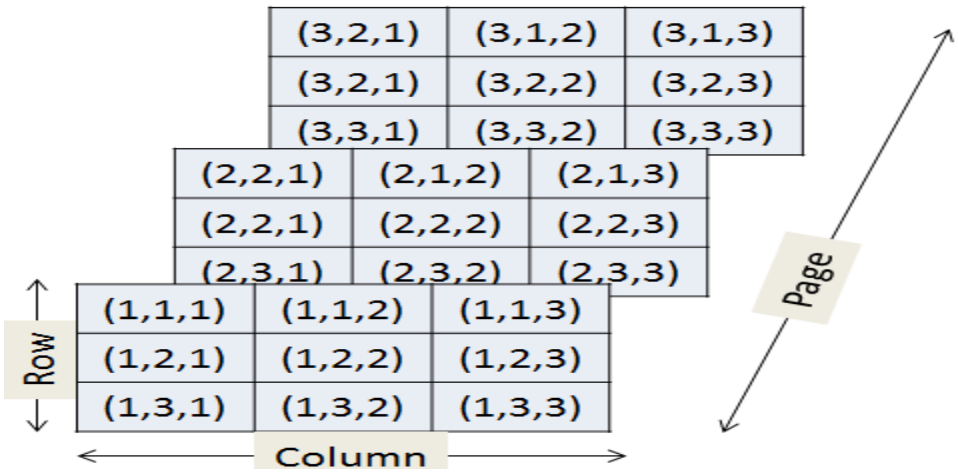
= 10*10 * 4 bytes = 400 Bytes, where M =N = 10

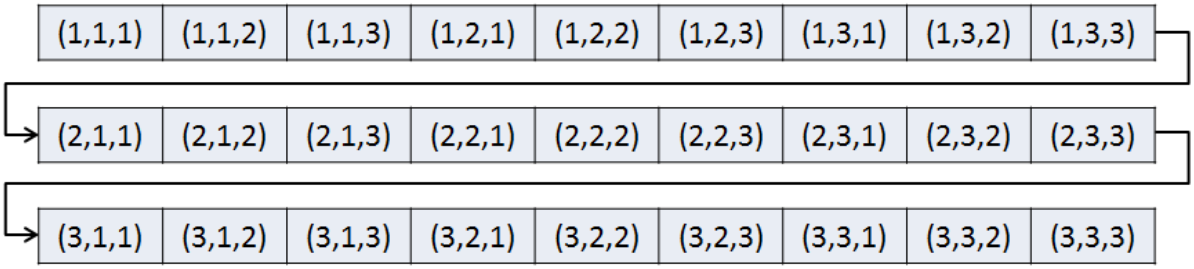
= 500*5 *4 bytes= 10000 Bytes, where M=500 and N= 5

So, Address of intArr[i][j]=Base address of intArr[i][j]+(i+j*M)*size of the data type

Three Dimensional Arrays

The three dimensional arrays are one dimensional array of two dimensional arrays.





So, a location address of a Three dimensional array A with size MxNxP is A[i][j][k]=Base Address of A + (i*N*P + j*P + k)*size of the data type

where I,j & k are the corresponding subscripts

N-Dimensional Arrays

Let A be an array with size, $M_1XM_2XM_3x\ldots\ldots\ldots\ldots\ldots M_N$ with subscripts

$i_1,i_2,i_3,\ldots\ldots\ldots i_N.$

So, $A[i_1,i_2,\ldots\ldots\ldots\ldots i_N]$=Base address of A+ $((M_2xM_3x\ldots\ldots\ldots\ldots xM_N)x\ i_1$ +

$(M_3xM_4x\ldots\ldots M_N)x\ i_2 +\ldots\ldots\ldots\ldots\ldots\ldots M_N\ x\ i_{N-1} + i_N))$x sizeof the data type.

Array Operations

- Creation
- Insertion
- Deletion
- Traverse
- Merge

Applications of Arrays

Sparse Matrices

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 values, then it is called a sparse matrix.

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

Example: 0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representation of sparse in memory

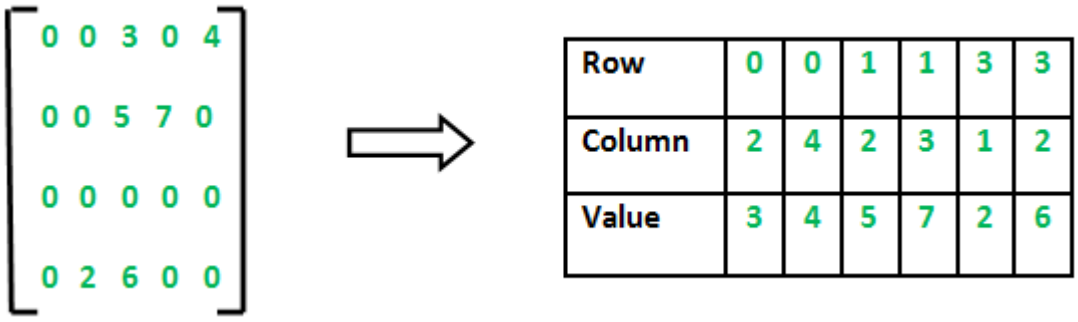
Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

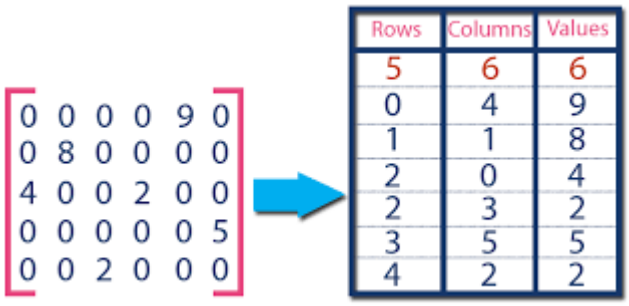
Method 1: Using Arrays(Triplet Representation)

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non-zero element located at index – (row, column)





Polynomial Representation and Addition Using Arrays

The general form of the polynomial is

$A_nX^n+A_{n-1}X^{n-1}+----------A_1X+A_0=0$

Eg. $5x^4+6x^2-7x+4=0$

To represent a polynomial in memory by using the following array data structure

```
struct poly
{  Int coe,exp;
};
struct poly p1[10];
```

Using this variable we can store a polynomial with 10 terms.

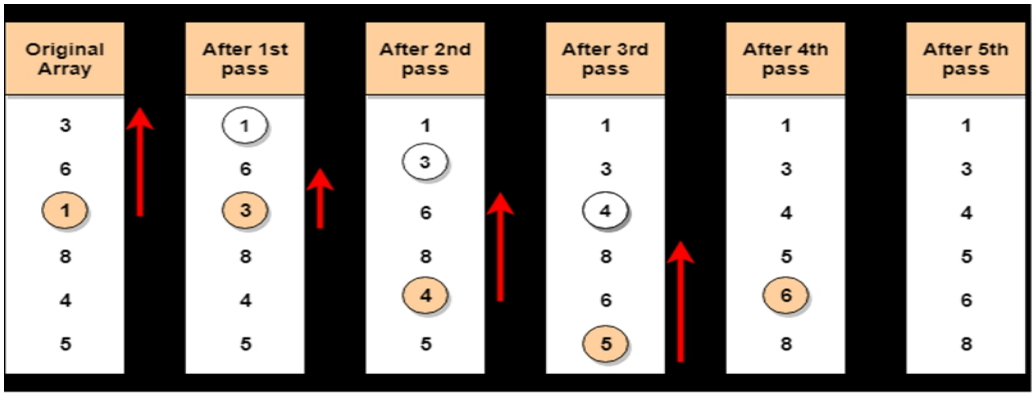
Sort Procedures

How Selection Sort Works?

Following are the steps involved in selection sort (for sorting a given array in ascending order):

1. **Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.**
2. **We then move on to the second position, and look for smallest element present in the sub array, starting from index 1, till the last index.**
3. **We replace the element at the second position in the original array, or we can say at the first position in the sub array, with the second smallest element.**
4. **This is repeated, until the array is completely sorted.**

Let's consider an array with values {3, 6, 1, 8, 4, 5}



Algorithm

1. **Repeat step-2 to 4 for k=0 to n-1**
2. **Set min=a[k] and loc=k // Usually first element in the list**
3. **Repeat for j=K+1 to n-1**

   If min>a[j] then

   Set min=a[j]

   Set loc=j;
4. **Interchange a[k] and a[loc]**
5. **End.**

Insertion Sort

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-array.

```
INSERTION-SORT(A,n)

  for i = 1 to n

        key ← A [i]

        j ← i – 1

         while j > = 0 and A[j] > key

                A[j+1] ← A[j]

                j ← j – 1

        End while

        A[j+1] ← key

  End for
```

Quick Sort

In Quick sort algorithm, partitioning of the list is performed using following steps...

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).

Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.

Step 4 - Decrement j until list[j] < pivot then stop.

Step 5 - If i < j then exchange list[i] and list[j].

Step 6 - Repeat steps 3,4 & 5 until i > j.

Step 7 - Exchange the pivot element with list[j] element.

Step 8 – Apply the same procedure in both the sub arrays (left and right sub arrays of

pivot element

Step 9 - Stop

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

```
// Quick Sort Program
#include<iostream.h>
#include<conio.h>
void quickSort(int [10],int,int);
void main()
{
  int list[20],size,i;
  cout<<"Enter size of the list: ";
  cin>>size;
  cout<<"Enter integer values: ",size);
  for(i = 0; i < size; i++)
  cout<<list[i];
  quickSort(list,0,size-1);
  cout<<"List after sorting is: ";
  for(i = 0; i < size; i++)
  cout<<list[i];
  getch();
}
void quickSort(int list[10],int first,int last)
{
    int pivot,i,j,temp;
     if(first < last)
        {
         pivot = first;
         i = first;
         j = last;
         while(i < j)
           {
             while(list[i] <= list[pivot] && i < last)
                 i++;
             while(list[j] > list[pivot])
                 j--;
             if(i <j){
                  temp = list[i];
                  list[i] = list[j];
                  list[j] = temp;
```

```
            }
        }
        temp = list[pivot];
        list[pivot] = list[j];
        list[j] = temp;
        quickSort(list,first,j-1);
        quickSort(list,j+1,last);
    }
}
```

Linear Search Procedure

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

Implementing Linear Search

Following are the steps of implementation that we will be following:

1. Traverse the array using a for loop.
2. In every iteration, compare the target value with the current value of the array.
3. If the values match, return the current index of the array.
4. If the values do not match, move on to the next array element.
5. If no match is found, return -1.

Given a array, Search a given element in array.

Case 1:

Input: Search 20.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: True (20 is present in array)

Case 2:

Input: Search 26.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: False (26 is not present in array)

Binary Search Procedure

Binary search is the most popular Search algorithm.It is efficient and also one of the most commonly used techniques that is used to solve problems.

Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.

When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Binary search Algorithm

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 matched, then display "Given element is found!!!" and terminate the function.

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

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

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

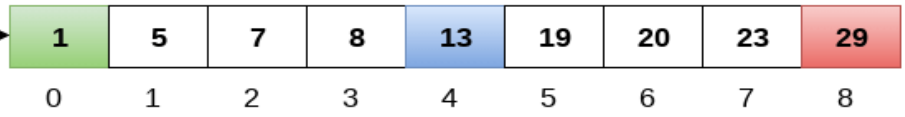
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Let us consider the following array:

Array A=1, 5, 7, 8, 13, 19, 20, 23, 29

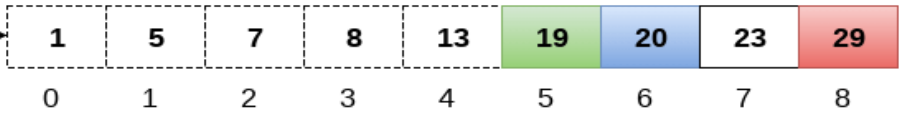
Item to be searched = 23

Step 1



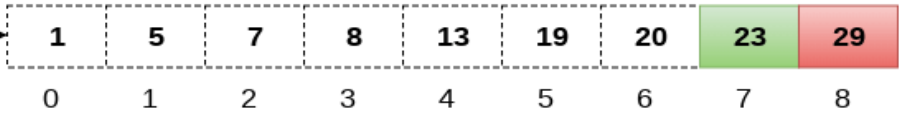
a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6

Step 2



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7

Step 3



a [mid] = 23
23 = 23
loc = mid

Return location 7