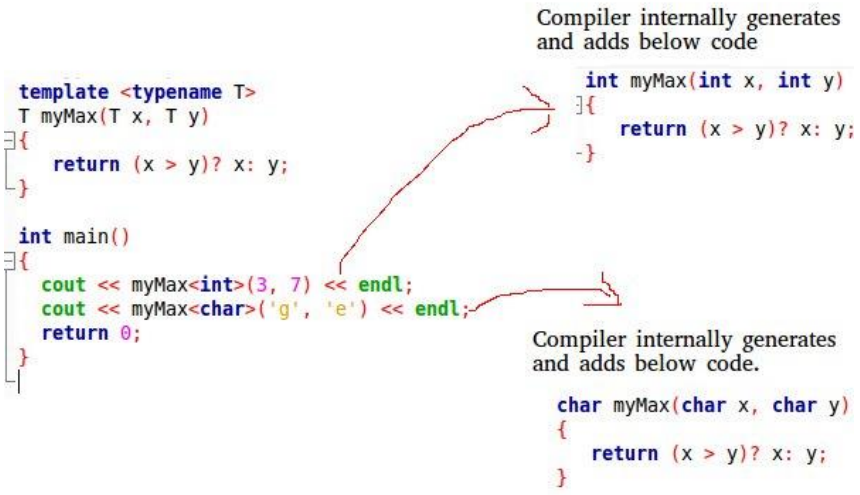
Data Types	Data Structures
Data Type is the kind or form of a variable that is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

Template:

A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types.



Polymorphism

The ability of a message to be displayed in more than one form.

Types

Compile time
Function overloading
Operator overloading

Run time
Virtual function

Operator Overloading

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator without changing its original meaning.

Code for operator overloading to add two complex number

```
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
```

```
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

There are some operators that cannot be overloaded. They are

- **Scope resolution operator ::**
- **Member selection operator .**
- **Member selection through .***
- **Scope resolution::**
- **sizeof**
- conditional or ternary operator(?:)

Binary Operator	Description
+	addition
–	subtraction
*	multiplication

/	division
%	remainder
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
&&	logical and
||	logical or
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
<<	bit-wise left-shift
>>	bit-wise right-shift
=	assignment
+=	addition assignment
-=	subtraction assignment
*=	multiplication assignment
/=	division assignment

%=	remainder assignment
&=	bit-wise AND assignment
|=	bit-wise OR assignment
^=	bit-wise XOR assignment
<<=	bit-wise left-shift assignment
>>=	bit-wise right-shift assignment

The unary operators are:

Unary Operator	Description
+	make positive
–	negate
!	not
~	bit-wise not
++	auto-increment
––	auto-decrement
*	indirection
&	address of

Inheritance:

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called the **“derived class” or “child class”** and the existing class is known as the **“base class”** or “parent class”. The derived class now is said to be inherited from the base class.

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
        //body
}
```

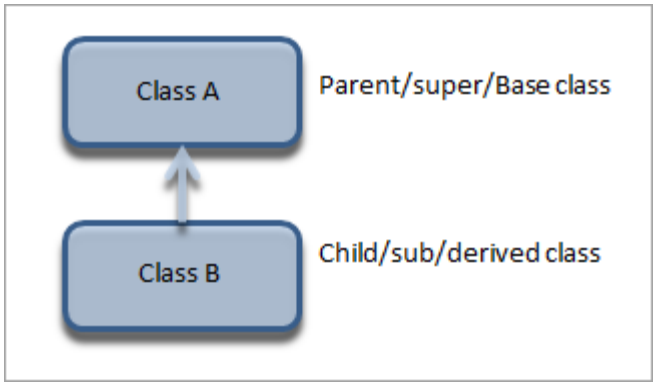
Why to use ..?

Code reusability: One of the main reasons to use inheritance is that you can reuse the code.

Transitive nature: Inheritance is also used because of its transitive nature. The derived class of the derived class will inherit the properties of the class base class as well.

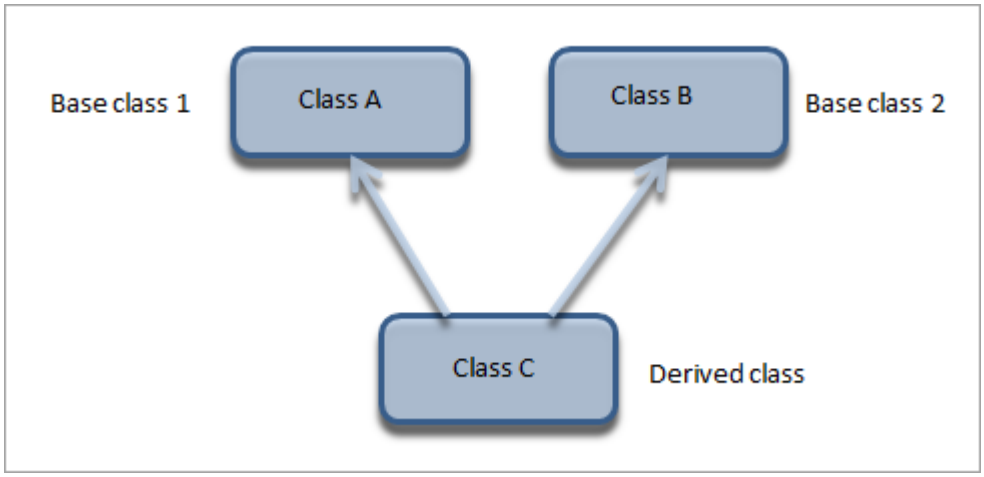
1) Single Inheritance

In single inheritance, a class derives from one base class only. This means that there is only one subclass that is derived from one superclass.



Multiple Inheritance

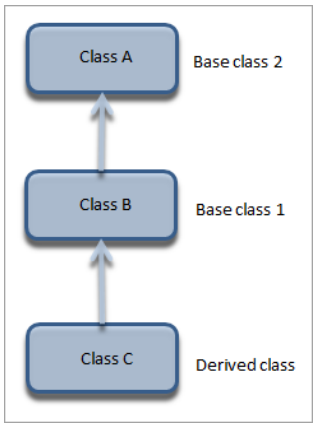
Multiple Inheritance is pictorially represented below.



Multiple inheritance is a type of inheritance in which a class derives from more than one class. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.

Multilevel Inheritance

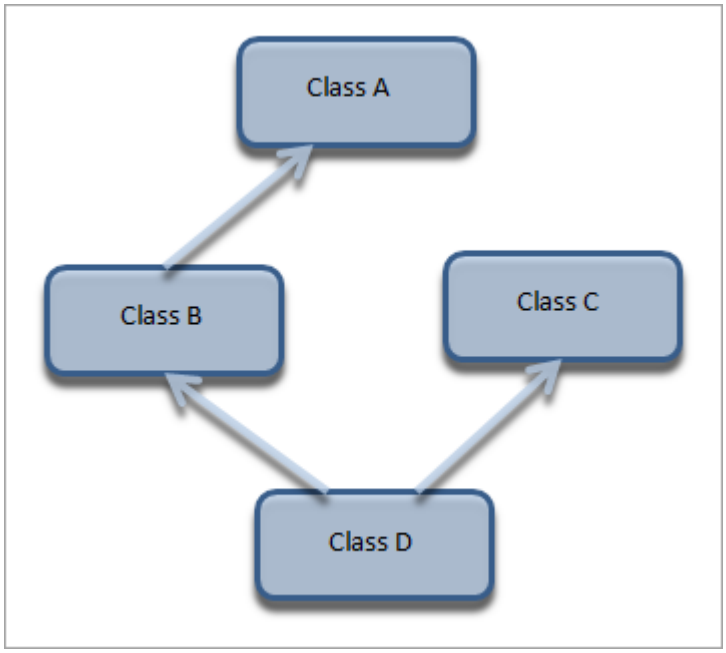
Multilevel inheritance is represented below.



In multilevel inheritance, a class is derived from another derived class. This inheritance can have as many levels as long as our implementation doesn't go wayward. In the above diagram, class C is derived from Class B. Class B is in turn derived from class A.

Hybrid Inheritance

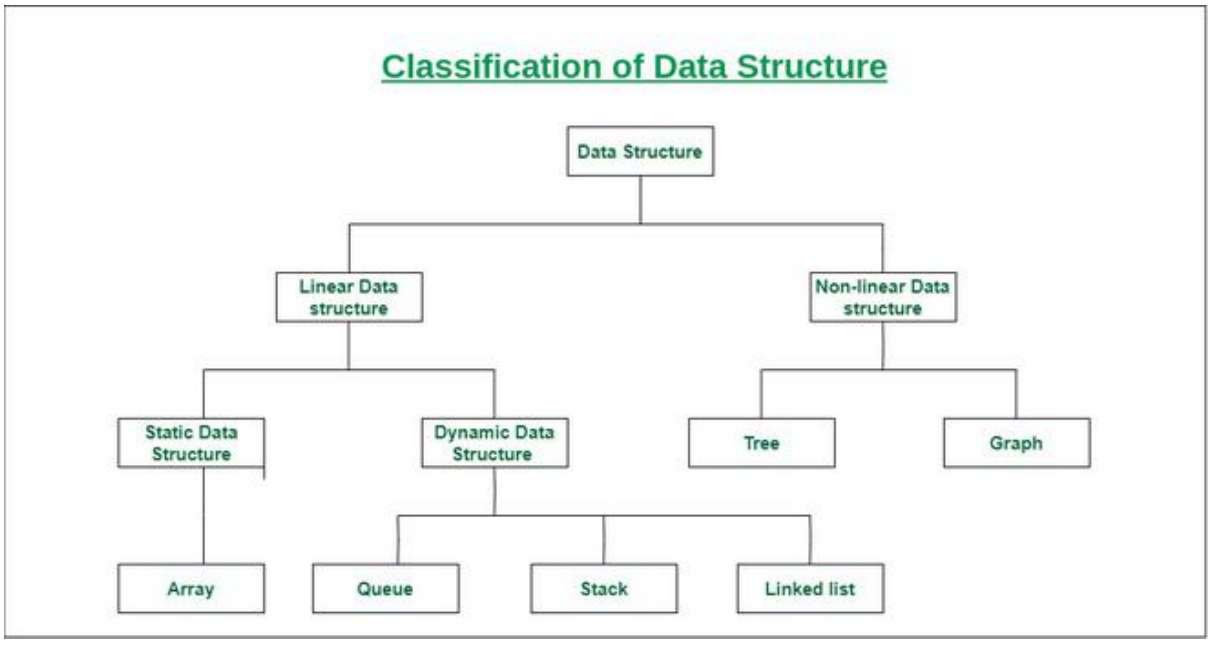
Hybrid inheritance is depicted below.



Hybrid inheritance is usually a combination of more than one type of inheritance. In the above representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B, and D) to get a hybrid inheritance.

Data Representation

The word data refers to constituting people, things, events, ideas. It can be a title, an integer, or any cast. After collecting data the investigator has to condense them in tabular form to study their salient features. Such an arrangement is known as the presentation of data. It refers to the process of condensing the collected data in a tabular form or graphically. This arrangement of data is known as Data Representation.



Basically, there are two **types of data structure**.

1. Primitive data structure
2. Non-primitive data structure

The primitive type of data structure includes predefined data structures such as char, float, int, and double.

The non-primitive data structures are used to store the collection of elements. This data structure can be further categorized into

1. Linear data structure
2. Non-Linear data structure.

Linear Data Structure

It is a type of data structure where the arrangement of the data follows a linear trend. The data elements are arranged linearly such that the element is directly linked to its previous and the next elements. As the elements are stored linearly, the structure supports single-level storage of data. And hence, traversal of the data is achieved through a single run only.

Characteristics

- It is a type of data structure where data is stored and managed in a linear sequence.
- Data elements in the sequence are linked to one after the other.
- Implementation of the linear structure of data in a computer's memory is easy as the data is organized sequentially.
- Array, queue. Stack, linked list, etc. are examples of this type of structure.
- The data elements stored in the data structure have only one relationship.
- Traversal of the data elements can be carried out in a single run as the data elements are stored in a single level.
- There is poor utilization of the computer memory if a structure storing data linearly is implemented.
- With the increase in the size of the data structure, the time complexity of the structure increases.

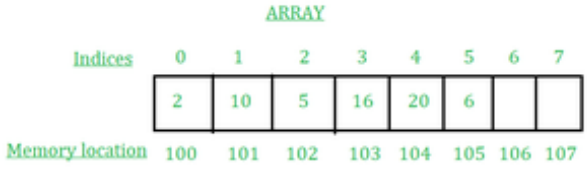
These structures can therefore be summarized as a type of data structure where the elements are stored sequentially and follow the order where:

- Only one *first element* is present which has one *next element.*
- Only one *last element* is present which has one *previous element.*
- All the other elements in the data structure have a *previous* and a *next* element

List of data structure in a linear type of data structure

1. Array (formula based representation)

is a linear data structure that is a collection of similar data types. Arrays are stored in contiguous memory locations. It is a static data structure with a fixed size. It combines data of similar types.




```c
#include <stdio.h>
#include <stdlib.h>

void main(){
  int array[5] = {4, 6, 1, 8, 5}; //Initialize array to hold 5 integers
  int *array_ptr; //Initialize array pointer
  array_ptr = array; //Set pointer to hold memory location of array

  //Print memory locations and values at that location
  for(int i = 0; i<sizeof(array)/sizeof(int); i++){
    printf("Memory Address: %p, Value at Address: %d \n", array_ptr + i, *(array_ptr + i));
  }
}
```

```
Memory Address: 000000000062FE20, Value at Address: 4
Memory Address: 000000000062FE24, Value at Address: 6
Memory Address: 000000000062FE28, Value at Address: 1
Memory Address: 000000000062FE2C, Value at Address: 8
Memory Address: 000000000062FE30, Value at Address: 5
```

Access the Elements of an Array

To access an array element, refer to its **index number**.

Array indexes start with **0**: [0] is the first element. [1] is the second element, etc.

```c
int myNumbers[] = {25, 50, 75, 100};
cout<< (myNumbers[0]);

// Outputs 25
```

Basic Operations:

- **Traversal - access each element of the linked list**
- **Insertion - adds a new element to the linked list**
- **Deletion - removes the existing elements**
- **Search - find a node in the linked list**
- **Sort - sort the nodes of the linked list**

Array Traversal :

Algorithm:

Step 1 start int arr[]

Step 2 initialize counter variable; set i=0

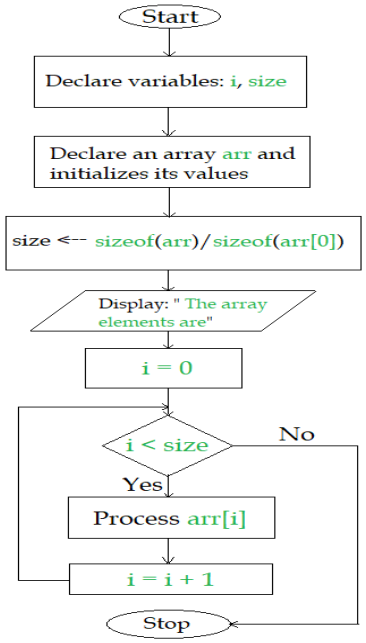
Step 3 repeat for lower counter value to higher counter value

Step 4 apply process to arr [i]

Step 5 end of loop

Step 6 stop

Flow chart:



// writing a program in C++ to perform traverse operation in array

```
#include <iostream>
using namespace std;

int main()
{
   int i, size;
   int arr[] = {53, 99, -11, 5, 102};    //declaring and initializing array "arr"

   size = sizeof(arr)/sizeof(arr[0]);
```

```
    cout << "The array elements are: ";

    for(i = 0; i < size; i++)
        cout << "\n" << "arr[" << i << "]= " << arr[i];

    return 0;
}
```

Insertion

Variables:

1. **arr :** Name of the array.
2. **size :** Size of the array (i.e., total number of elements in the array)
3. **i :** Loop counter or counter variable for the `for` loop.
4. **x :** The data element to be insert.
5. **pos :** The position where we wish to insert the element.

Algorithm to Insert an element in an Array:

- **Step 01:** Start

Step 02: [Reset size of the array.] set `size=size+1`

- **Step 03:** [Initialize counter variable.] `Set i = size - 1`
- **Step 04:** Repeat Step 05 and 06 for `i = size - 1` to `i >= pos - 1`
- **Step 05:** [Move i^{th} element forward.] set `arr[i+1] = arr[i]`
- **Step 06:** [Decrease counter.] Set `i = i - 1`
- **Step 07:** [End of step 04 loop.]
- **Step 08:** [Insert element.] Set `arr[pos-1] = x`
- **Step 09:** Stop

```
// writing a program in C++ to insert an element in an array

#include <iostream>
using namespace std;

int main()
{
     int i, size, x, pos;
     int arr[] = {2, 4, 6, 8, 12};
```

```
    size = sizeof(arr)/sizeof(arr[0]);

    cout << "The array elements before insertion operation:\n";

    for(i = 0; i < size; i++)
       cout << "arr[" << i << "] = " << arr[i] << "\n";

    cout << "\nEnter the element to be insert: ";
    cin >> x;

    cout << "\nEnter the position where you wish to insert the element: ";
    cin >> pos;

    size = size + 1;

    cout << "\nThe array elements after insertion operation are: ";

    for(i = size - 1; i >= pos - 1; i--)
       arr[i+1] = arr[i];

    arr[pos-1] = x;

    for(i = 0; i < size; i++)
       cout << "\narr[" << i << "] = " << arr[i];

    return 0;
 }
```

Output

```
The array elements before insertion operation:
arr[0] = 2
arr[1] = 4
arr[2] = 6
arr[3] = 8
arr[4] = 12

Enter the element to be insert: 786

Enter the position where you wish to insert the element: 1

The array elements after insertion operation are:
arr[0] = 786
arr[1] = 2
arr[2] = 4
arr[3] = 6
arr[4] = 8
arr[5] = 12
```

Variables we are using here:

1. **a :** Name of the array.
2. **size :** Size of the array (i.e., total number of elements in the array)
3. **i :** Loop counter or counter variable for the `for` loop.
4. **pos :** The position from where we wish to delete the element.

Algorithm to Delete an element from an Array:

- **Step 01:** Start
- **Step 02:** [Initialize counter variable.] `Set i = pos - 1`
- **Step 03:** Repeat Step 04 and 05 for `i = pos - 1` to `i < size`
- **Step 04:** [Move i^{th} element backward (left).] set `a[i] = a[i+1]`
- **Step 05:** [Increase counter.] Set `i = i + 1`
- **Step 06:** [End of step 03 loop.]
- **Step 07:** [Reset size of the array.] set `size = size - 1`
- **Step 08:** Stop

```
// writing a program in C++ to delete an element from an array

#include <iostream>
using namespace std;
 int main()
{
     int i, size, x, pos;
     int a[] = {-1, 87, -68, 10, 8};
      size = sizeof(a)/sizeof(a[0]);
      cout << "The array elements before deletion operation:\n";
     for(i = 0; i < size; i++)
        cout << "a[" << i << "] = " << a[i] << "\n";

     cout << "\nEnter the position from where you wish to delete the element: "
     cin >> pos;

     cout << "\nThe array elements after deletion operation are: ";
```

```
    for(i = pos - 1; i < size; i++)
       a[i] = a[i+1];

    size = size - 1;

    for(i = 0; i < size; i++)
       cout << "\na[" << i << "] = " << a[i];
    return 0;
```

- }

Output:

```
The array elements before deletion operation:
a[0] = -1
a[1] = 87
a[2] = -68
a[3] = 10
a[4] = 8

Enter the position from where you wish to delete the element: 2

The array elements after deletion operation are:
a[0] = -1
a[1] = -68
a[2] = 10
a[3] = 8
```

Search:

Variables we are using here:

1. **a :** Name of the array.
2. **n:** Size of the array (i.e., Total number of elements in the array).
3. **i :** Loop counter or counter variable for the `for` loop.
4. **x :** The data element to be search.

The following algorithm search a data element `x` in a linear array `a`.

Algorithm to Search an element in an Array:

- **Step 01:** Start
- **Step 02:** [Initialize counter variable.] `Set i = 0`
- **Step 03:** Repeat Step 04 and 05 for `i = 0` to `i < n`
- **Step 04:** `if a[i] = x`, then jump to step 07
- **Step 05:** [Increase counter.] Set `i = i + 1`
- **Step 06:** [End of step 03 loop.]
- **Step 07:** Print `x` found at `i + 1` position and go to step 09
- **Step 08:** Print `x` not found (`if a[i] != x`, after all the iteration of the above `for` loop.)
- **Step 09:** Stop

```
// writing a program in C++ to search an element in an array


#include <iostream>
#define MAX 50

using namespace std;

int main()
{
     int i, n, x;
     int a[MAX];

     cout << "Enter the size of the array: ";
     cin >> n;

     cout << "Enter " << n << " elements:\n";

     for(i = 0; i < n; i++)
        cin >> a[i];

     cout << "\nEnter the element to search: ";
```

```
        cin >> x;

        for(i = 0; i < n; i++)
        {

           if(a[i] == x)
           {
               cout << x << " found at " << i+1 << " position.";
               return 0;
           }
        }

        cout << x << " not found";
        return 0;

    }
```

output

```
Enter the size of the array: 5
Enter 5 elements:
11
22
33
-512
99

Enter the element to search: -512
-512 found at 4 position.
```

Binary Search

Problem: Given a sorted array **arr[]** of **n** elements, write a function to search a given element **x** in **arr[]** and return the index of x in the array. Consider array is 0 base index.

Binary Search Algorithm:

The basic steps to perform Binary Search are:

- Sort the array in ascending order.
- Set the low index to the first element of the array and the high index to the last element.
- Set the middle index to the average of the low and high indices.
- If the element at the middle index is the target element, return the middle index.

- If the target element is less than the element at the middle index, set the high index to the middle index – 1.
    - If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
    - Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Binary Search Algorithm can be implemented in the following two ways

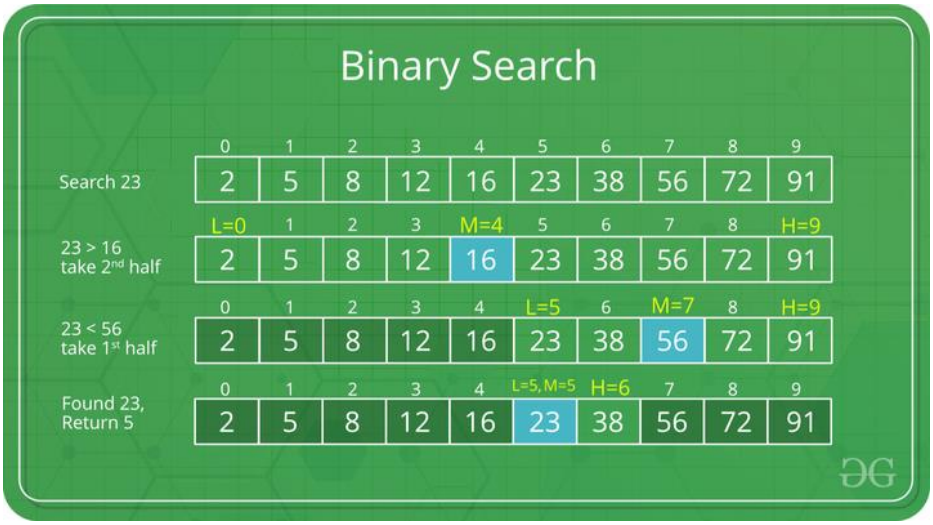
- Iterative Method
- Recursive Method

1. Iteration Method

```
binarySearch(arr, x, low, high)
        repeat till low = high
   mid = (low + high)/2
   if (x == arr[mid])
   return mid

    else if (x > arr[mid]) // x is on the right side
    low = mid + 1
    else          // x is on the left side
    high = mid - 1
```

2. Recursive Method (The recursive method follows the divide and conquer approach)

```
binarySearch(arr, x, low, high)
    if low > high
      return False
     else
      mid = (low + high) / 2
      if x == arr[mid]
      return mid
      else if x > arr[mid]    // x is on the right side
        return binarySearch(arr, x, mid + 1, high)

      else              // x is on the left side
        return binarySearch(arr, x, low, mid - 1)
```



What is sort operation in array?

Sort operation in Array or sorting an array simply means, arranging the elements of an array in some specific (definite) logical order.

How to sort the elements of an array in C/C++?

Variables we are using here:

1. **arr :** Name of the array.
2. **n:** Size of the array (i.e., Total number of elements in the array).
3. **i, j :** Loop counter or counter variable for the `for` loop.
4. **x :** A temporary variable for swapping two values.

The following algorithm sort the elements of an array in ascending order.

Algorithm to Sort the elements of an Array:

- **Step 01:** Start
- **Step 02:** Repeat Step 03 to 07 for i = 0 to i < n
- **Step 03:** Repeat Step 04 and 05 for j = i + 1 to j < n
- **Step 04:** Check a statement using the if keyword, If arr[i] > arr[j], Swap arr[i] and arr[j].
- **Step 05:** [Increase counter.] Set j = j + 1
- **Step 06:** [End of step 03 loop.]

- **Step 07:** [Increase counter.] Set i = i + 1
- **Step 08:** [End of step 02 loop.]
- **Step 09:** Stop

```
// writing a program to perform sort operation on an array in C++
#include <iostream>
#define MAX 50
using namespace std;
int main()
{
   int i, j, n, arr[MAX],x;
   cout << "Enter the size of the array: " ;
   cin >> n;

   cout << "Enter " << n << " elements:\n" ;
   for(i=0;i<n;i++)
     cin >> arr[i];

   for (i = 0; i < n; i++)
   {

     for (j = i + 1; j < n; j++)
     {

       if (arr[i] > arr[j])
       {

          x =  arr[i];
          arr[i] = arr[j];
          arr[j] = x;

       }
     }
   }
```

```
    cout << "\nThe numbers arranged in ascending order are given below: \n" ;
    for (i = 0; i < n; i++)
        cout << arr[i] << "\n" ;
    return 0;
}
```

output

```
Enter the size of the array: 4
Enter 4 elements:
7
10
9
2

The numbers arranged in ascending order are given below:
2
7
9
10
```

Applications, Advantages and Disadvantages of Array

Below are some applications of arrays.

Storing and accessing data: Arrays are used to store and retrieve data in a specific order. For example, an array can be used to store the scores of a group of students, or the temperatures recorded by a weather station.
Sorting: Arrays can be used to sort data in ascending or descending order. Sorting algorithms such as bubble sort, merge sort, and quicksort rely heavily on arrays.
Searching: Arrays can be searched for specific elements using algorithms such as linear search and binary search.
Matrices: Arrays are used to represent matrices in mathematical computations such as matrix multiplication, linear algebra, and image processing.
Stacks and queues: Arrays are used as the underlying data structure for implementing stacks and queues, which are commonly used in algorithms and data structures.
Graphs: Arrays can be used to represent graphs in computer science. Each element in the array represents a node in the graph, and the relationships between the nodes are represented by the values stored in the array.
Dynamic programming: Dynamic programming algorithms often use arrays to store intermediate results of subproblems in order to solve a larger problem.

Below are some real-time applications of arrays.

Signal Processing: Arrays are used in signal processing to represent a set of samples that are collected over time. This can be used in applications such as speech recognition, image processing, and radar systems.

Multimedia Applications: Arrays are used in multimedia applications such as video and audio processing, where they are used to store the pixel or audio samples. For example, an array can be used to store the RGB values of an image.

Data Mining: Arrays are used in data mining applications to represent large datasets. This allows for efficient data access and processing, which is important in real-time applications.

Robotics: Arrays are used in robotics to represent the position and orientation of objects in 3D space. This can be used in applications such as motion planning and object recognition.

Real-time Monitoring and Control Systems: Arrays are used in real-time monitoring and control systems to store sensor data and control signals. This allows for real-time processing and decision-making, which is important in applications such as industrial automation and aerospace systems.

Financial Analysis: Arrays are used in financial analysis to store historical stock prices and other financial data. This allows for efficient data access and analysis, which is important in real-time trading systems.

Scientific Computing: Arrays are used in scientific computing to represent numerical data, such as measurements from experiments and simulations. This allows for efficient data processing and visualization, which is important in real-time scientific analysis and experimentation.

Applications of Array in C#:

Implementing dynamic programming algorithms: Dynamic programming algorithms often use arrays to store intermediate results. For example, in the famous Fibonacci series algorithm, an array is used to store the values of Fibonacci series.

Database programming: Arrays can be used to store the results of database queries. For example, an array can be used to store the results of a SELECT query.

Parallel programming: Arrays are used in parallel programming to distribute workloads among multiple threads. For example, an array can be divided into multiple parts, and each part can be processed by a different thread.

Advantages of array data structure:

Efficient access to elements: Arrays provide direct and efficient access to any element in the collection. Accessing an element in an array is an O(1) operation, meaning that the time required to access an element is constant and does not depend on the size of the array.

Fast data retrieval: Arrays allow for fast data retrieval because the data is stored in contiguous memory locations. This means that the data can be accessed quickly and efficiently without the need for complex data structures or algorithms.

Memory efficiency: Arrays are a memory-efficient way of storing data. Because the elements of an array are stored in contiguous memory locations, the size of the array

is known at compile time. This means that memory can be allocated for the entire array in one block, reducing memory fragmentation.

Versatility: Arrays can be used to store a wide range of data types, including integers, floating-point numbers, characters, and even complex data structures such as objects and pointers.

Easy to implement: Arrays are easy to implement and understand, making them an ideal choice for beginners learning computer programming.

Compatibility with hardware: The array data structure is compatible with most hardware architectures, making it a versatile tool for programming in a wide range of environments.

Disadvantages of array data structure:

Fixed size:

Arrays have a fixed size that is determined at the time of creation. This means that if the size of the array needs to be increased, a new array must be created and the data must be copied from the old array to the new array, which can be time-consuming and memory-intensive.

Memory allocation issues:

Allocating a large array can be problematic, particularly in systems with limited memory. If the size of the array is too large, the system may run out of memory, which can cause the program to crash.

Insertion and deletion issues:

Inserting or deleting an element from an array can be inefficient and time-consuming because all the elements after the insertion or deletion point must be shifted to accommodate the change.

Wasted space:

If an array is not fully populated, there can be wasted space in the memory allocated for the array. This can be a concern if memory is limited.

Limited data type support:

Arrays have limited support for complex data types such as objects and structures, as the elements of an array must all be of the same data type.

Lack of flexibility:

The fixed size and limited support for complex data types can make arrays inflexible compared to other data structures such as linked lists and trees.

Advantages of Structure over Array:

- The structure can store different types of data whereas an array can only store similar data types.
- Structure does not have limited size like an array.
- Structure elements may or may not be stored in contiguous locations but array elements are stored in contiguous locations.
- In structures, object instantiation is possible whereas in arrays objects are not possible.

2. Linked list

- The linked list is that type of data structure where separate objects are stored sequentially.

- Every object stored in the data structure will have the data and a reference to the next object.
- The last node of the linked list has a reference to null.
- The first element of the linked list is known as the head of the list.

Dynamic memory allocation.

Disadvantage:

Getting to an element in a linked list is a slower process compared to the arrays as the indexing in an array helps in locating the element.

However, in the case of a linked list, the process has to start from the head and traverse through the whole structure until the desired element is reached.

Advantage:

In contrast to this, the advantage of using linked lists is that the addition or deletion of elements at the beginning can be done very quickly.

Types of Linked Lists:

- **Simple Linked List** – In this type of linked list, one can move or traverse the linked list in only one direction. where the next pointer of each node points to other nodes but the next pointer of the last node points to NULL. It is also called "**Singly Linked List**".


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

Data declaration

```
Node* one = new Node();
Node* two = new Node();
Node* three = new Node();
Node* four = new Node();
```

Memory allocation

```
one->data = 3;
two->data = 4;
three->data = 5;
four->data = 6;
```

Assign Data

```
one->next=two;
two->next=three;
three->next=four;
four->next=NULL;
```

Connect node

Doubly Linked List –

- In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)
- We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.
-

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

//Data declaration

```
struct node *head;
struct node *one=NULL;
struct node * two= NULL;
Struct node *three=NULL;
Struct node *four=NULL;
```

//Initialize nodes

```
Struct node *head;
```

```
Node* one = new Node();
Node* two = new Node();
Node* three = new Node();
Node* four = new Node();
```

//Allocate memory

```
one->data = 3;
two->data = 4;
three->data = 5;
four->data = 6;
```

//Assign data value

```
one->next=two;
one->prev=NULL;
two->next=three;
two->prev=one;
three->next=four;
three->prev=two;
four->next=NULL;
four->prev=three
```

//Node traversing

Circular Singly Linked List

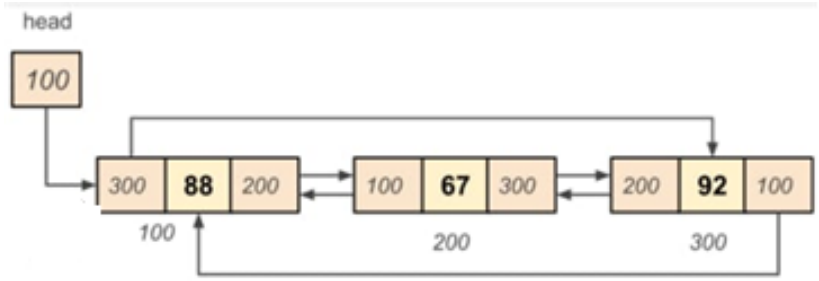
A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.


```
one->next=two;
two->next=three;
three->next=four;
three->prev=two;
four->next=head;
```

Connect node

Doubly Circular Linked List

A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.



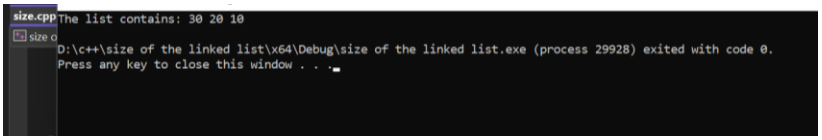

```
one->next=two;
one->prev=four;
two->next=three;
two->prev=one;
three->next=four;
three->prev=two;
four->next=one;
four->prev=three
```

Connect node

- **Traversal - access each element of the linked list**
- **Insertion - adds a new element to the linked list(front, end, at specified position)**
- **Deletion - removes the existing elements**
- **Search - find a node in the linked list**
- **Length:** find the length of the linked list
- **Sort - sort the nodes of the linked list**

Linked List operations:

```
#include <iostream>
using namespace std;

//node structure
struct Node {
    int data;
    Node* next;
};

class LinkedList {
private:
    Node* head;
public:
    LinkedList() {
        head = NULL;
    }

    //Add new element at the start of the list
    void push_front(int newElement) {
        Node* newNode = new Node();
        newNode->data = newElement;
        newNode->next = head;
        head = newNode;
    }

    //display the content of the list
    void PrintList() {
        Node* temp = head;
        if (temp != NULL) {
            cout << "The list contains: ";
            while (temp != NULL) {
                cout << temp->data << " ";
                temp = temp->next;
            }
            cout << endl;
        }
        else {
            cout << "The list is empty.\n";
        }
    }
};
// test the code
int main() {
    LinkedList MyList;
    Node* head = NULL;
    //Add three elements at the start of the list.
    MyList.push_front(10);
    MyList.push_front(20);
    MyList.push_front(30);
    MyList.PrintList();
   return 0;
      }
```



Insert node at back

```
void push_back(int num)// insert data to the back
{
	struct node* new_node = new node();
	new_node->data = num;
	new_node->next = NULL;

		if (head == NULL)
	{

		head = new_node;

	}
		else
		{
			node* temp = head;

			while (temp->next != NULL)
			temp = temp->next;
			temp->next = new_node;



		}
		}
```

Insert node at given position

```
void insert_pos(int n, int pos)    // insert at given position
	{
	node* temp = new node;
	temp = head;
	node* new_node = new node;
	new_node->data = n;
	new_node->next = NULL;
	if (pos == 1)
	{
		new_node->next=head;
		head = new_node;
		//new_node->next = NULL;
	}
	else
	{
		for (int i = 0; i = pos - 1; i++)
		{
			if (temp != NULL)
			{
				new_node->next = temp->next;
				temp->next = new_node;
			  }
			else
			    {
				cout << "\nthe previous node is null;" << endl;

			     }
		}
	}
	 }
```

Delete a node

```
void pop_at(int position) {
	if (position < 1) {
		cout << "\nposition should be >= 1.";
	}
	else if (position == 1 && head != NULL) {
		node* temp = head;
		head = temp->next;
		free(temp);
	}
	else {
		node* temp = head;
		for (int i = 1; i < position - 1; i++) {
			if (temp != NULL) {
				temp = temp->next;
			}
		}
		if (temp != NULL&& temp->next!=NULL) {
			node* temp2 = temp->next;
			temp->next = temp->next->next;
			free(temp2);
		}
		else {
			cout << "\nThe node is already null.";
		}
	}
	}
```

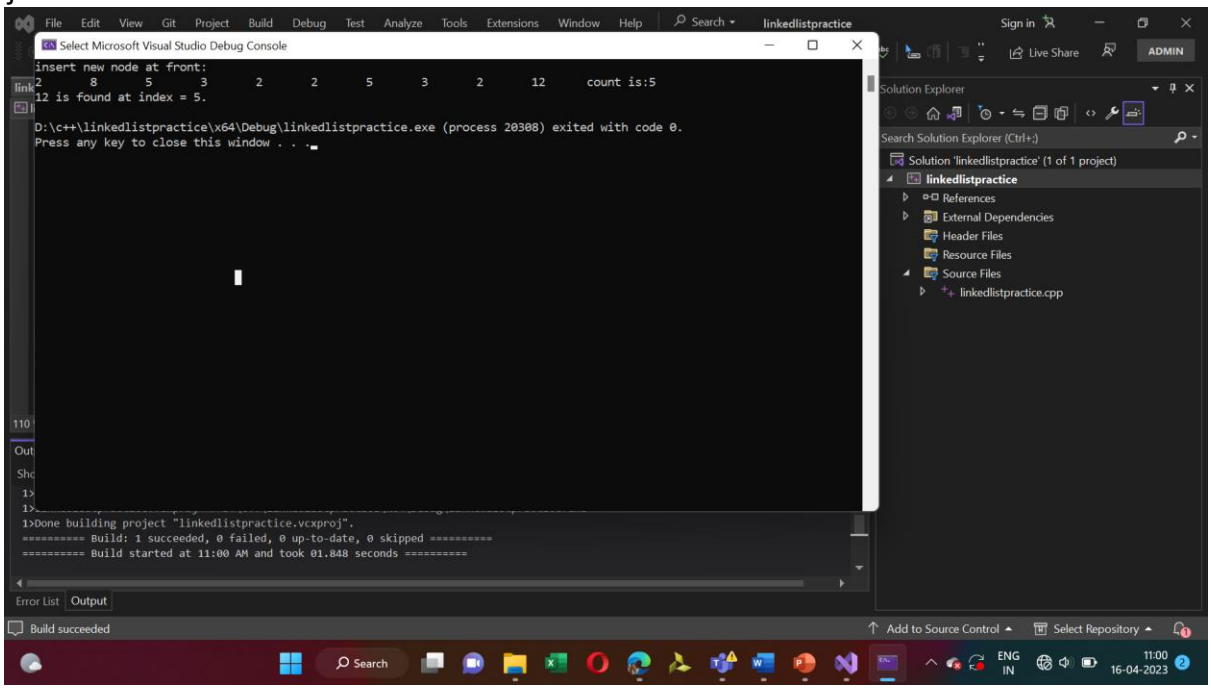
Search an element

```
void SearchElement(int searchValue)
{

	node* temp = head;
	int found = 0;
	int i = 0;
	if (temp != NULL) {
		while (temp != NULL) {
			i++;
			if (temp->data == searchValue) {
				found++;
				break;
			}
			else {
				temp = temp->next;

			}
			}
		if (found == 1) {
			cout << searchValue << " is found at index = " << i << ".\n";
		}
		else {
			cout << searchValue << " is not found in the list.\n";
		}
	}
	else {

		cout << "The list is empty.\n";
	}
	}
```

Count no. of node

```
void count()
```

```
 {
        int count = 0;
        temp = head;
        while (temp != 0)
        {
               temp = temp->next;
               count++;
        }
        cout << "count is:" << count << endl;

        }


       //driver code

int main()
{
       int k;
       cout << "insert new node at front:" << endl;
       push_front(2);
       push_front(3);
       push_front(5);
       push_front(8);
       push_front(2);
       display();
       push_back(12);
       pop_at(2);
       display();
       count();
       return 0;

}
```



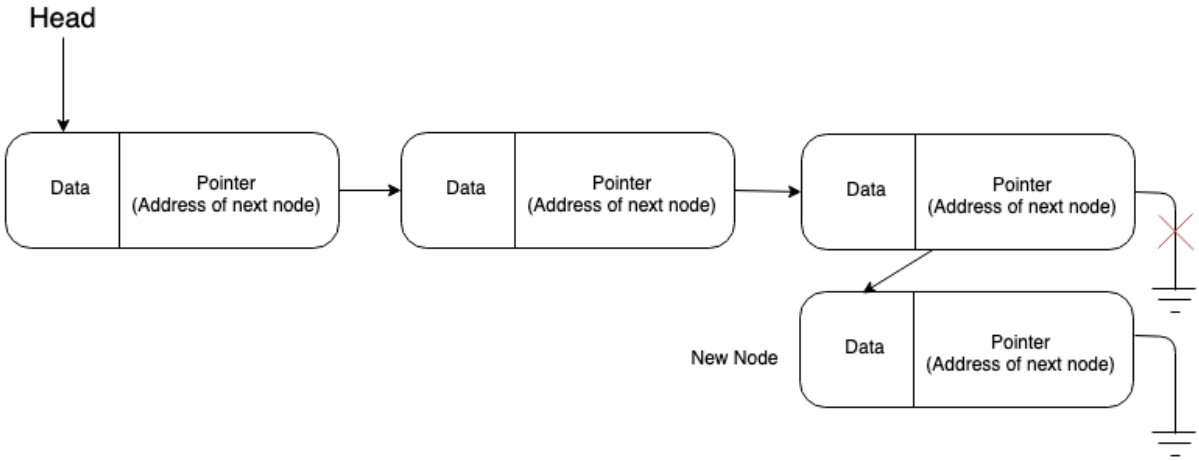


Diagram of appending a node at back

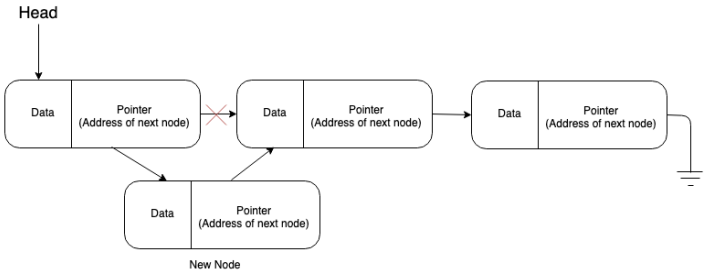


Diagram of inserting a node in the middle of a linked list

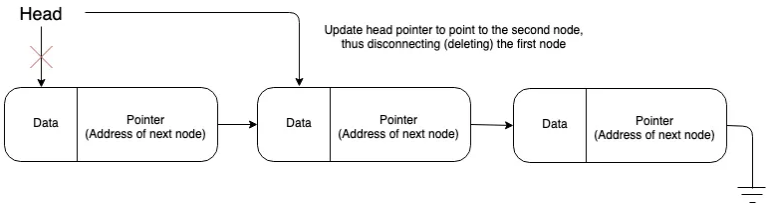


Diagram of deleting first node from linked list

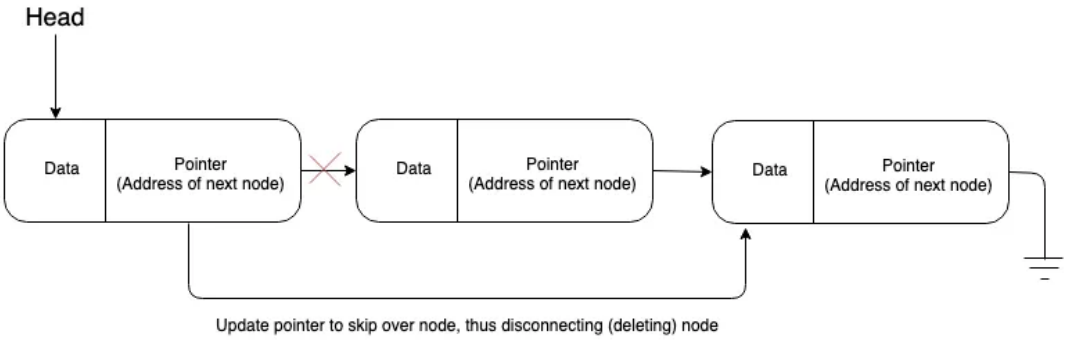


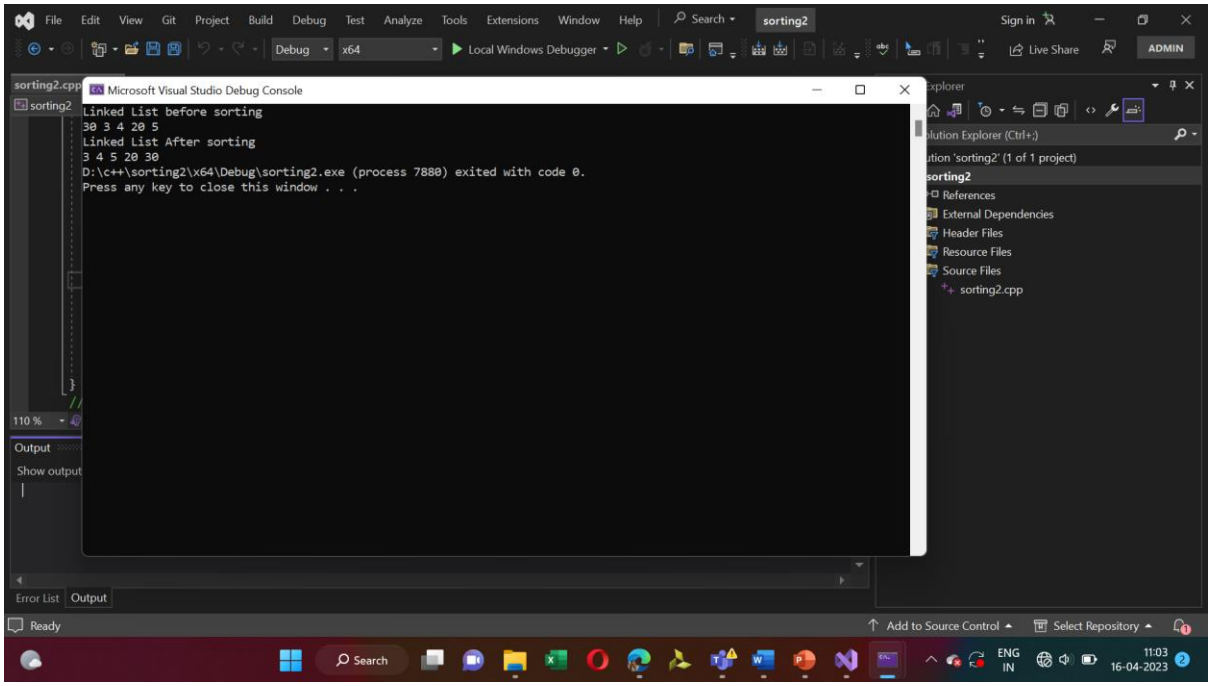
Diagram of deleting a node from the middle of linked list

Sorting operation

```
// C++ program to sort link list
// using insertion sort
#include <iostream>
using namespace std;
```

```
struct Node
{
    int val;
    struct Node* next;
    Node(int x)
    {
        val = x;
        next = NULL;
    }
};

class LinkedlistIS
{
public:
    Node* head;
    Node* sorted;

    void push(int val)
    {
        // Allocate node
        Node* newnode = new Node(val);

        // Link the old list of the
        // new node
        newnode->next = head;
        // Move the head to point to the
        // new node
        head = newnode;
    }

    // Function to sort a singly linked list
    // using insertion sort
    void insertionSort(Node* headref)
    {
        // Initialize sorted linked list
        sorted = NULL;
        Node* current = headref;

        // Traverse the given linked list
        // and insert every node to sorted
        while (current != NULL)
        {
            // Store next for next iteration
            Node* next = current->next;

            // Insert current in sorted
            // linked list
            sortedInsert(current);

            // Update current
            current = next;
        }

        // Update head_ref to point to
        // sorted linked list
        head = sorted;
    }

    /* Function to insert a new_node in a list.
       Note that this function expects a pointer
       to head_ref as this can modify the head of
       the input linked list (similar to push()) */
```

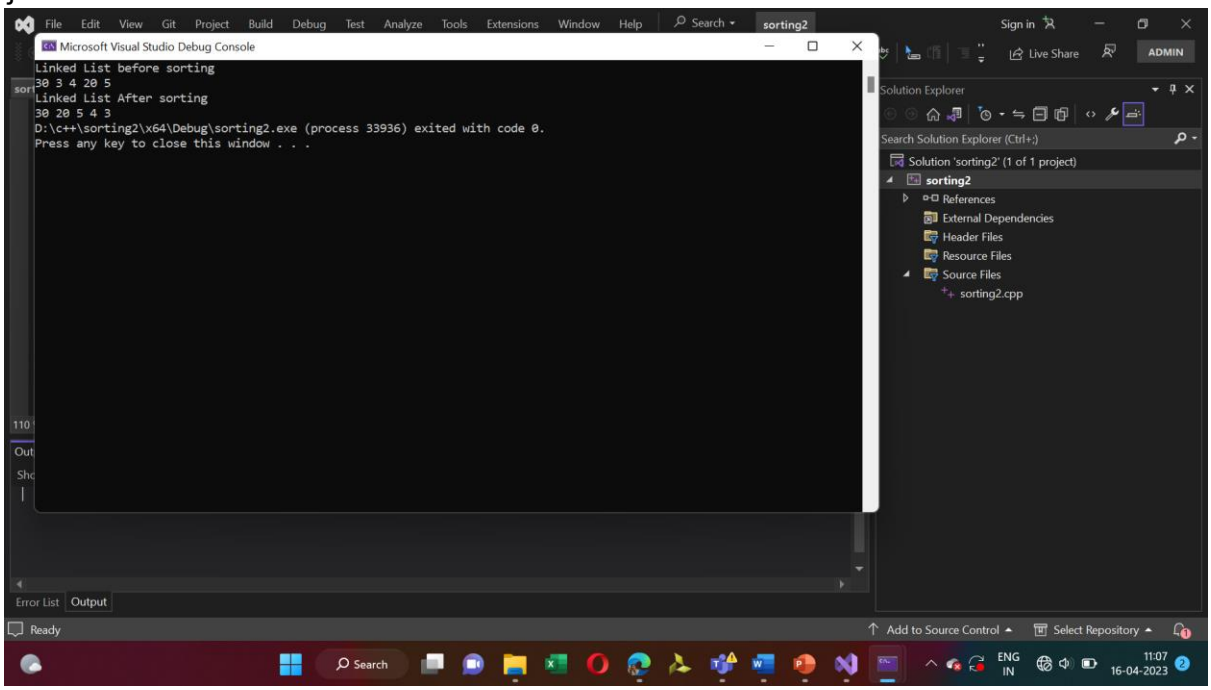
```
    void sortedInsert(Node* newnode)
    {
        // Special case for the head end
        if (sorted == NULL ||
            sorted->val >= newnode->val)
        {
            newnode->next = sorted;
            sorted = newnode;
        }
        else
        {
            Node* current = sorted;

            /* Locate the node before the
               point of insertion */
            while (current->next != NULL &&
                current->next->val < newnode->val)
            {
                current = current->next;
            }
            newnode->next = current->next;
            current->next = newnode;
        }
    }

    // Function to print linked list
    void printlist(Node* head)
    {
        while (head != NULL)
        {
            cout << head->val << " ";
            head = head->next;
        }
    }
};

// Driver code
int main()
{
    LinkedlistIS list;
    list.head = NULL;
    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);
    cout << "Linked List before sorting" <<
        endl;
    list.printlist(list.head);
    cout << endl;
    list.insertionSort(list.head);
    cout << "Linked List After sorting" <<
        endl;
    list.printlist(list.head);
}
```




```cpp
// C++ program to sort link list descending order
// using insertion sort
#include <iostream>
using namespace std;

struct Node
{
    int val;
    struct Node* next;
    Node(int x)
    {
        val = x;
        next = NULL;
    }
};

class LinkedlistIS
{
public:
    Node* head;
    Node* sorted;

    void push(int val)
    {
        // Allocate node
        Node* newnode = new Node(val);

        // Link the old list of the
        // new node
        newnode->next = head;
        // Move the head to point to the
        // new node
        head = newnode;
    }

    // Function to sort a singly linked list
    // using insertion sort
    void insertionSort(Node* headref)
    {
```

```
        // Initialize sorted linked list
        sorted = NULL;
        Node* current = headref;

        // Traverse the given linked list
        // and insert every node to sorted
        while (current != NULL)
        {
            // Store next for next iteration
            Node* next = current->next;

            // Insert current in sorted
            // linked list
            sortedInsert(current);

            // Update current
            current = next;
        }

        // Update head_ref to point to
        // sorted linked list
        head = sorted;
    }

    /* Function to insert a new_node in a list.
       Note that this function expects a pointer
       to head_ref as this can modify the head of
       the input linked list (similar to push()) */
    void sortedInsert(Node* newnode)
    {
        // Special case for the head end
        if (sorted == NULL ||
            sorted->val <= newnode->val)
        {
            newnode->next = sorted;
            sorted = newnode;
        }
        else
        {
            Node* current = sorted;

            /* Locate the node before the
               point of insertion */
            while (current->next != NULL &&
                current->next->val > newnode->val)
            {
                current = current->next;
            }
            newnode->next = current->next;
            current->next = newnode;
        }
    }

    // Function to print linked list
    void printlist(Node* head)
    {
        while (head != NULL)
        {
            cout << head->val << " ";
            head = head->next;
        }
    }
};
```

```
// Driver code
int main()
{
    LinkedlistIS list;
    list.head = NULL;
    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);
    cout << "Linked List before sorting" <<
        endl;
    list.printlist(list.head);
    cout << endl;
    list.insertionSort(list.head);
    cout << "Linked List After sorting" <<
        endl;
    list.printlist(list.head);
}
```



Disadvantages of Linked List

Slower Search Time:

Linked list have slower search times than arrays as random access is not allowed.

	Array	Linked List
Strength	• Random Access (Fast Search Time) • Less memory needed per element • Better cache locality	• Fast Insertion/Deletion Time • Dynamic Size • Efficient memory allocation/utilization
Weakness	• Slow Insertion/Deletion Time • Fixed Size • Inefficient memory allocation/utilization	• Slow Search Time • More memory needed per node as additional storage required for pointers