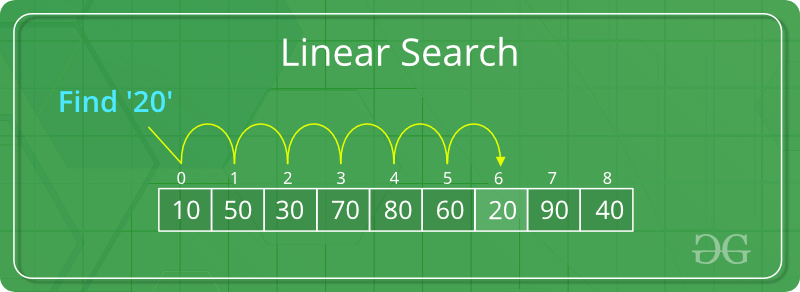
**What is Searching Algorithm?**

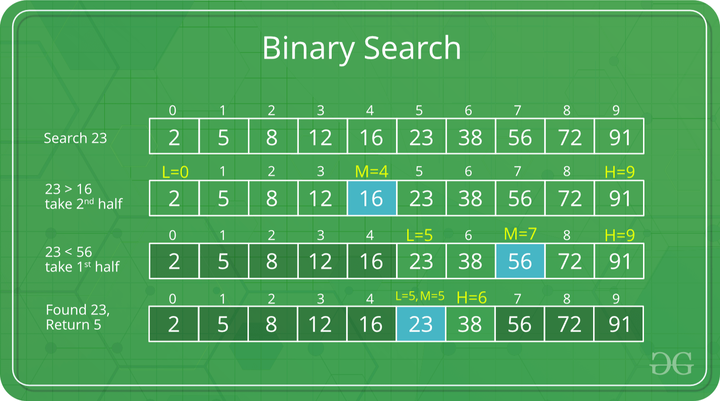
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

Based on the type of search operation, these algorithms are generally classified into two categories:

**Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: **Linear Search.**



**Interval Search:** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: **Binary Search**.



**How Linear Search Works?**

Step 1: First, read the search element (Target element) in the array.

Step 2: Set an integer i = 0 and repeat steps 3 to 4 till i reaches the end of the array.

Step 3: Match the key with arr[i].

Step 4: If the key matches, return the index. Otherwise, increment i by 1.

**Consider the array arr[] = {10, 50, 30, 70, 80, 20, 90, 40} and key = 20**

**Complexity Analysis of Linear Search:**

**Time Complexity:** Time complexity represents the number of times a statement is executed. The time complexity of an algorithm is NOT the actual time required to execute a particular code, since that depends on other factors like programming language, operating software, processing power, etc.

To express the time complexity of an algorithm, we use something called the **“Big O notation”**. The Big O notation is a language we use to describe the time complexity of an algorithm.

**Big O notation expresses the run time of an algorithm in terms of how quickly it grows relative to the input (this input is called “n”). This way, if we say for example that the run time of an algorithm grows “on the order of the size of the input”, we would state that as “O(n)”. If we say that the run time of an algorithm grows “on the order of the square of the size of the input”, we would express it as “O(n²)”.**

**There are different types of time complexities:**

**Constant Time Complexity: O(1)**

When time complexity is constant (notated as “O(1)”), the size of the input (n) doesn’t matter. Algorithms with Constant Time Complexity take a constant amount of time to run, independently of the size of n.



In the **best case**, the key might be present at the first index. So the best case complexity is O(1).

**Linear Time Complexity: O(n)**

****When time complexity grows in direct proportion to the size of the input. This means that as the input grows, the algorithm takes proportionally longer to complete.

**Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst case complexity is O(N) where N is the size of the list.

**Logarithmic Time Complexity: O(log n)**

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. This means that instead of increasing the time it takes to perform each subsequent step, the time is decreased at a magnitude that is inversely proportional to the input “n”.

This time complexity is generally associated with algorithms that divide problems in half every time, which is a concept known as “Divide and Conquer”.

1. They divide the given problem into sub-problems of the same type.

2. They recursively solve these sub-problems.

3. They appropriately combine the sub-answers to answer the given problem.

**Average Case: O(N)**

**Advantages of Linear Search:**

1. Linear search is simple to implement and easy to understand.
2. Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
3. Does not require any additional memory.
4. It is a well-suited algorithm for small datasets.

**Drawbacks of Linear Search:**

1. Linear search has a time complexity of O(n), which in turn makes it slow for large datasets.
2. Not suitable for large arrays.
3. Linear search can be less efficient than other algorithms, such as hash tables.

**Conditions for when to apply Binary Search in a Data Structure:**

To apply binary search in any data structure, the data structure must maintain the following properties:

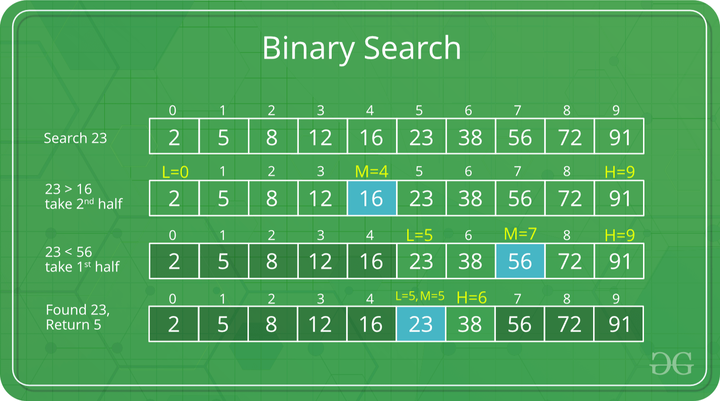
1. The data structure must be sorted.
2. Access to any element of the data structure takes constant time.

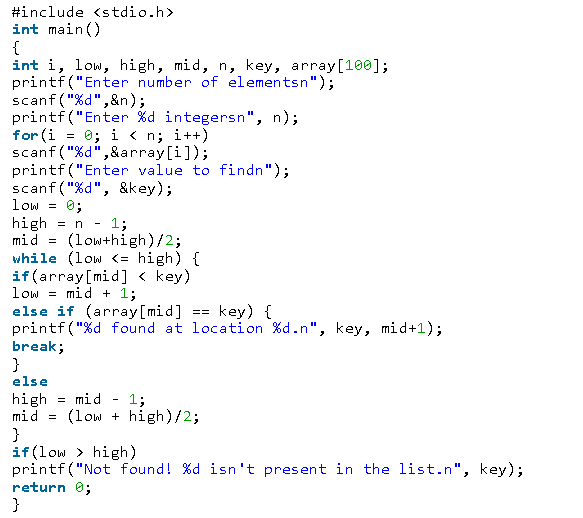
**How does Binary Search work?**

Consider an array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the target = 23.

* **First Step:**
* Initially the search space is from 0 to 9.
* Let’s denote the boundary by L and H where L = 0 and H = 9 initially.
* Now mid of this search space is M = 4.
* So compare target with **arr[M].**
* **Second Step:**
* As arr[4] is less than target, switch the search space to the right of 16, i.e., [5, 9].
* Now L = 5, H = 9 and M becomes 7.
* Compare target with **arr[M].**
* **Third Step:**
* arr[7] is greater than target.
* Shift the search space to the left of M, i.e., [5, 6].
* So, now L = 5, H = 6 and M = 6.
* Compare arr[M] with target.
* Here arr[M] and target are the same.

So, we have found the target.



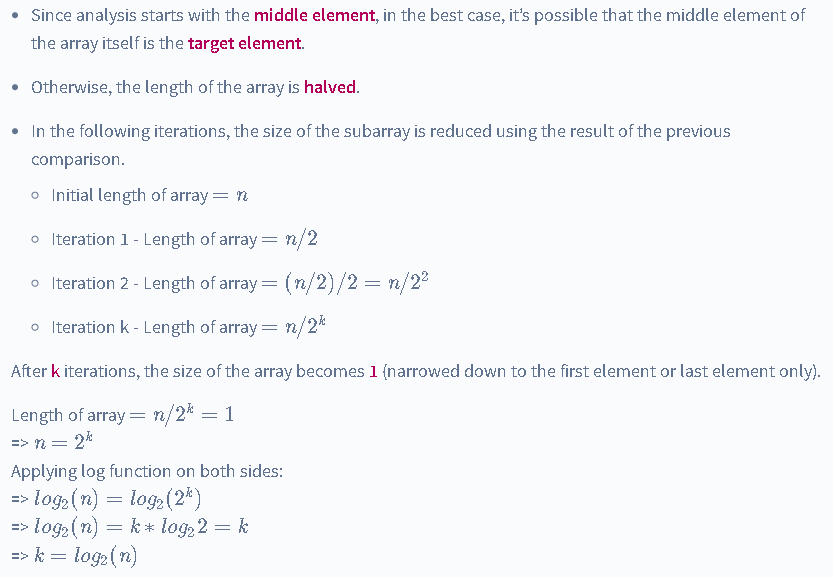


Best case is the function which performs the minimum number of steps on input data of n elements.

Worst case is the function which performs the maximum number of steps on input data of size n.

Average case is the function which performs an average number of steps on input data of n elements.

Complexity Analysis of Binary Search



**Analysis of Time Complexity of Binary Search**

**Best Case Time Complexity of Binary Search**

The best case scenario of Binary Search occurs when the target element is in the central index. In this situation, there is only one comparison. Therefore, the Best Case Time Complexity of Binary Search is O(1).

**Average Case Time Complexity of Binary Search**

The average case arises when the target element is present in some location other than the central index or extremities. The time complexity depends on the number of comparisons to reach the desired element.

Therefore, the overall Average Case Time Complexity of Binary Search is O(logn).

**Worst Case Time Complexity of Binary Search**

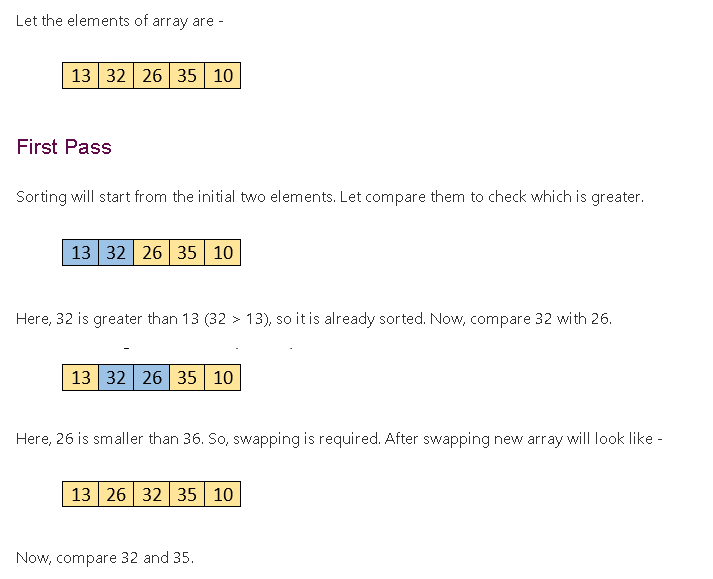
The worst-case scenario of Binary Search occurs when the target element is the smallest element or the largest element of the sorted array.

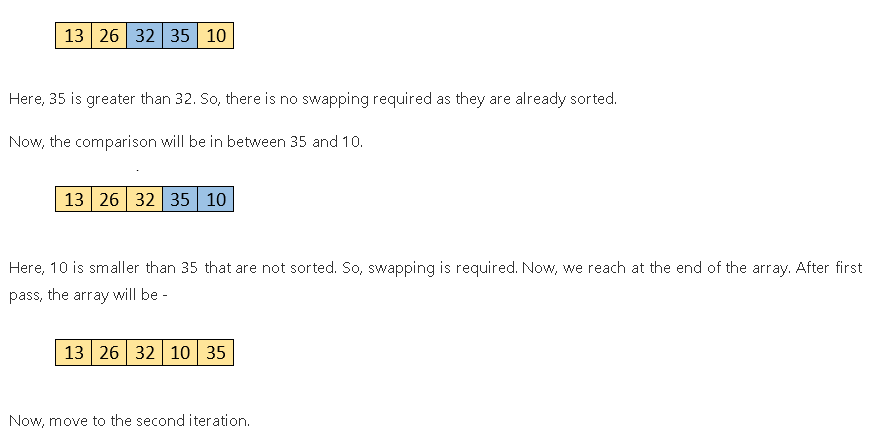
In each iteration or recursive call, the search gets reduced to half of the array.

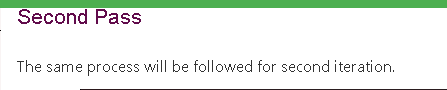
So for an array of size n, there are atmost log2n iterations or recursive calls.

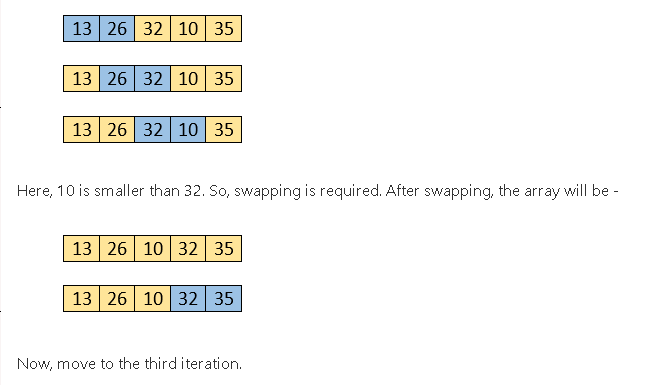
Since the target element is present in the extremitites (first or last index), there are logn comparisons in total. Therefore, the Worst Case Time Complexity of Binary Search is O(logn).

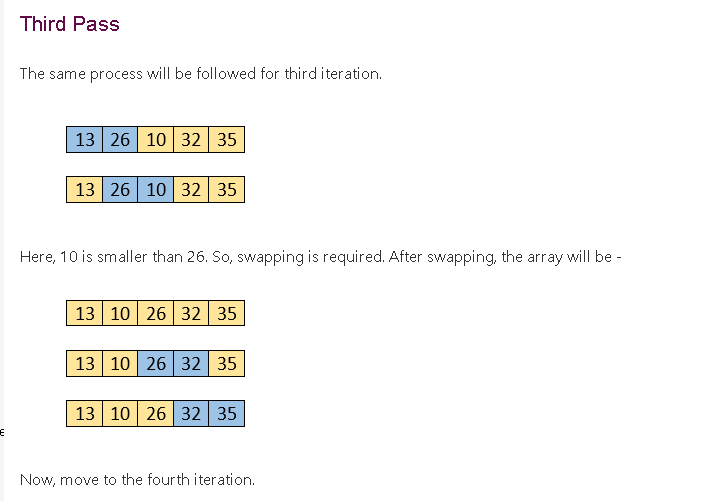
**Bubble Sort**

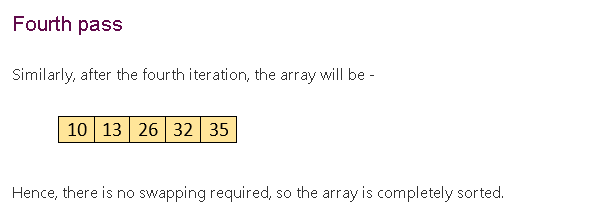










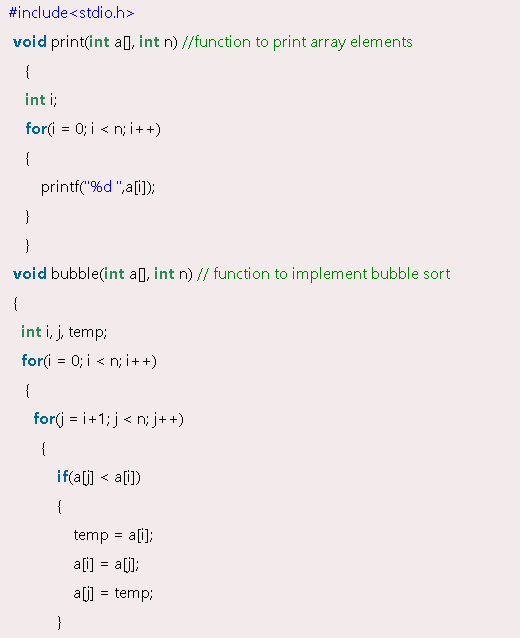
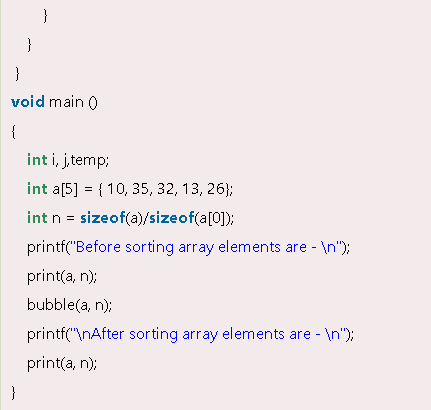


**Bubble sort complexity**

**Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).

**Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is O(n2).

**Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is O(n2).



**Insertion Sort**

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

Insertion sort has various advantages such as –

1. Simple implementation
2. Efficient for small data sets
3. Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

**Algorithm**

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

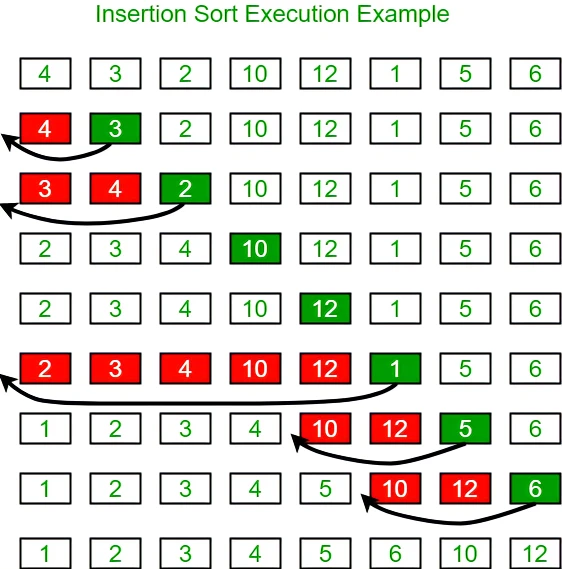
Step2 - Pick the next element, and store it separately in a key.

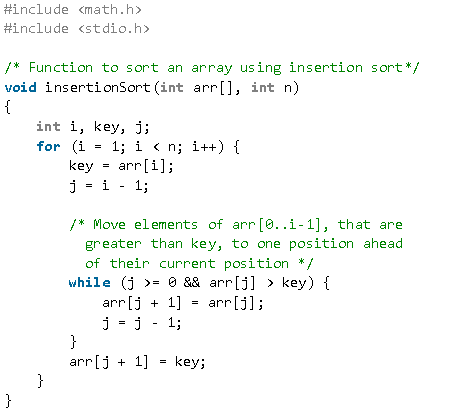
Step3 - Now, compare the key with all elements in the sorted array.

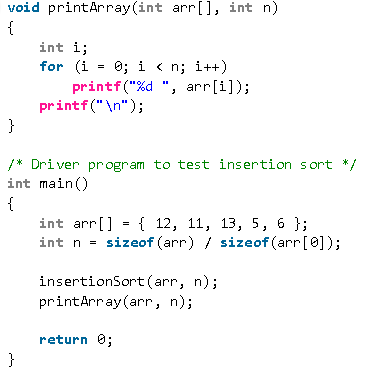
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.







**Time Complexities**

1. **Worst Case Complexity:** **O(n2)**

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n\*(n-1) ~ n2

1. **Best Case Complexity: O(n)**

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

1. **Average Case Complexity: O(n2)**

It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

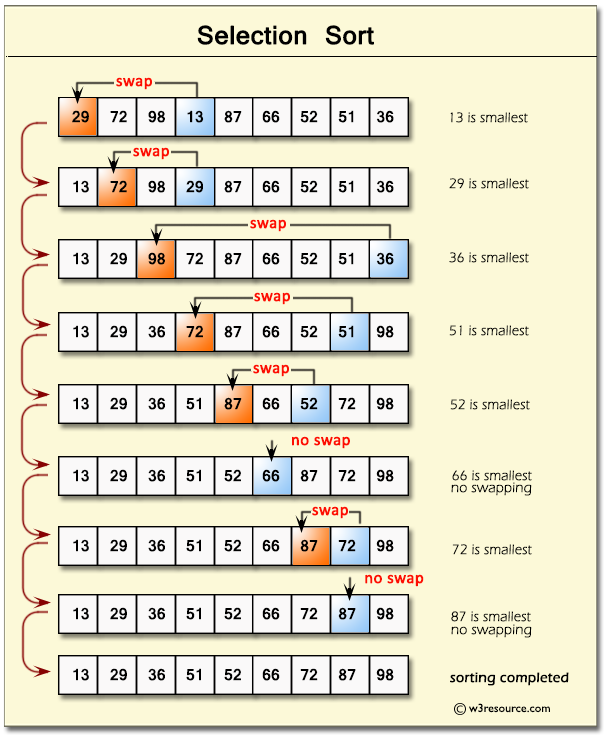
**Insertion Sort Applications**

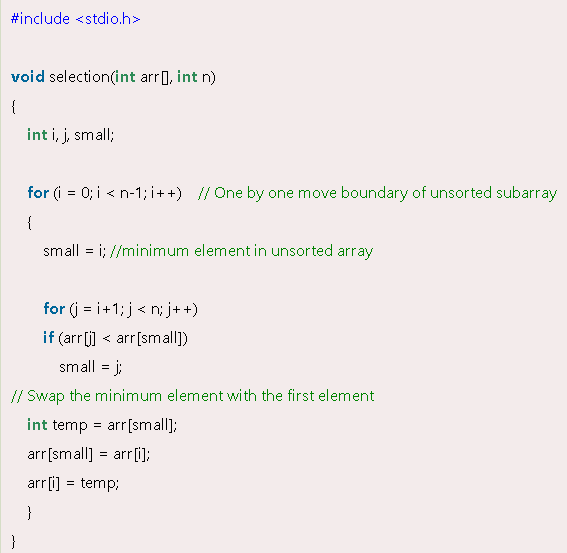
The insertion sort is used when:

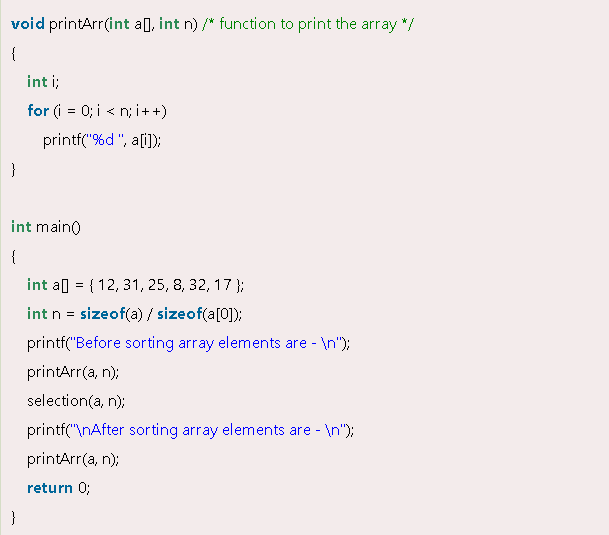
1. the array has a small number of elements
2. there are only a few elements left to be sorted

**Selection Sort**

* It works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
* The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion.
* This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.







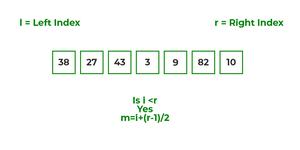
* The time complexity of the selection sort depends upon the iteration and then finding the minimum element in each iteration. In each iteration, we find the minimum element from the sub-array and bring it to the right position using swapping.
* The time complexity of finding the minimum element in a sub-array takes O(N) time where N is the number of elements in the unsorted sub-array. As we will need to traverse the entire unsorted sub-array of size N for finding the minimum element so the time complexity of searching comes out to be O(N). Now, after the searching, we will be performing the swapping (if needed) but the swapping is a constant time operation i.e. it takes only O(1) time hence it will not affect the overall time complexity of the selection sort.
* Now, for each of the N iterations, we perform the searching and swapping so the time complexity of the selection sort becomes N\*N i.e. O(N2). Let us look at the best, average, and worst-case time complexity of selection sort.
* The best case is when the array is already sorted. So even if the array is already sorted, we will traverse the entire array for checking, and in each iteration or traversal, we will perform the searching operation. Only the swapping will not be performed as the elements will be at the correct position. Since the swapping only takes a constant amount of time i.e. O(1)the best time complexity of selection sort comes out to be O(N2).
* The worst case is when the array is completely unsorted or sorted in descending order. So, we will traverse the entire array for checking, and in each iteration, we will perform the searching operation. After searching, we will swap the element at its correct position. As we know that the swapping only takes a constant amount of time i.e. O(1) so the worst time complexity of selection sort also comes out to be O(N2).
* The average case is when some part of the array is sorted. So, as we have seen earlier in the best and worst cases, we will need to perform the searching, and swapping in each iteration. So, the average time complexity of the selection sort is O(N2) as well.

**Merge Sort Algorithm**

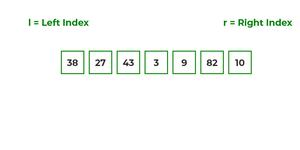
* Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
* Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement.
* One of the main advantages of merge sort is that it has a time complexity of **O(n log n),** which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

To know the functioning of merge sort lets consider an array **arr[] = {38, 27, 43, 3, 9, 82, 10}**

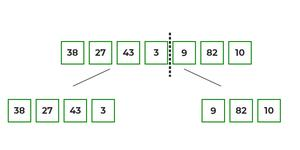
1. At first, check if the left index of array is less than the right index, if yes then calculate its mid point.



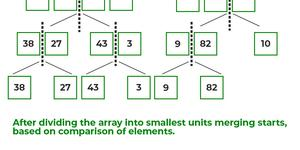
1. Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
2. Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



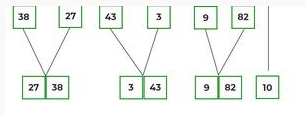
1. Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



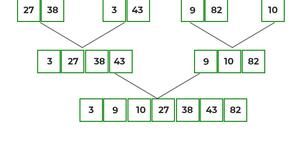
1. Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

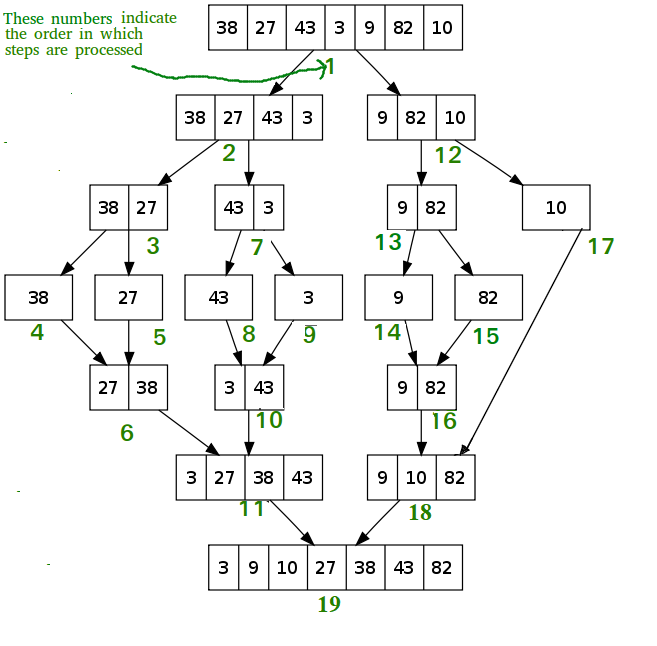


1. After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
2. Firstly, compare the element for each list and then combine them into another list in a sorted manner.



1. After the final merging, the list looks like this:



****

**Algorithm:**

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

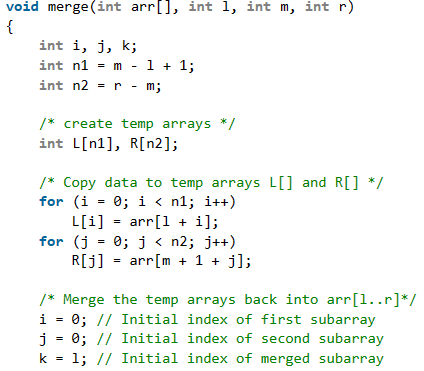
mid= (left+right)/2

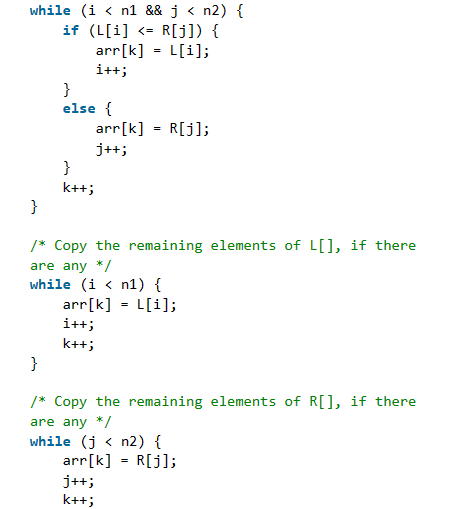
mergesort(array, left, mid)

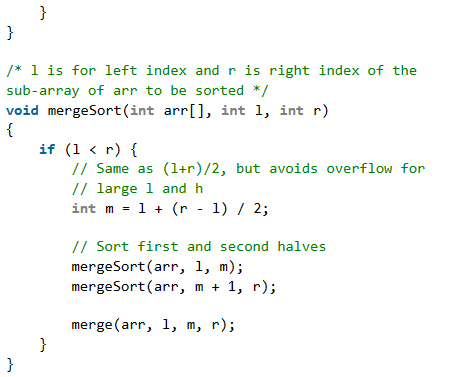
mergesort(array, mid+1, right)

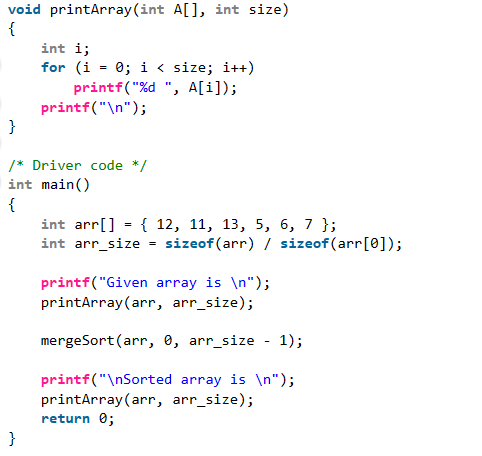
merge(array, left, mid, right)

step 4: Stop

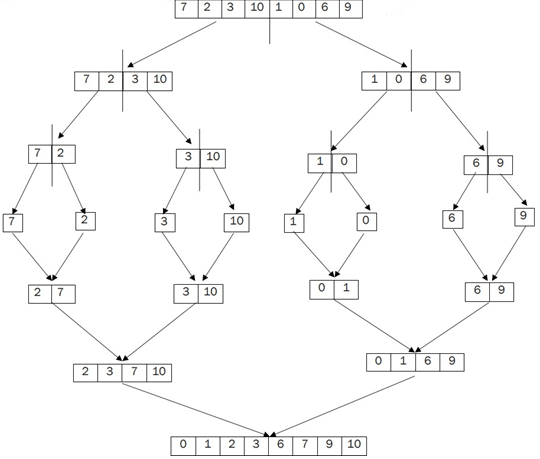








**Example 2:**

****

**Runtime Test Cases**

**Test Case 1 – Average Case: Here, the elements are entered in random order.**

**/\* Average case \*/**

**Enter the size: 8**

**Enter the elements of array: 7 2 3 10 1 0 6 9**

**The sorted array is: 0 1 2 3 6 7 9 10**

**Test Case 2 – Best Case: Here, the elements are already sorted.**

**/\* Best case \*/**

**Enter the size: 5**

**Enter the elements of array: -7 -3 8 9 12**

**The sorted array is: -7 -3 8 9 12**

**Test Case 3 – Worst Case: Here, the elements are reverse sorted.**

**/\* Worst case \*/**

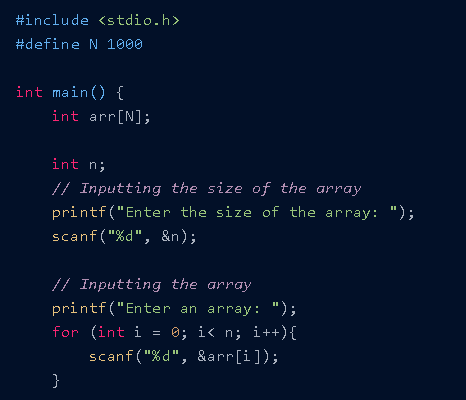
**Enter the size: 4**

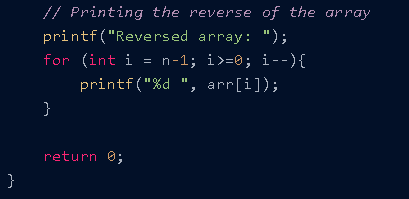
**Enter the elements of array: 84 32 3 -9**

**The sorted array is: -9 3 32 84**

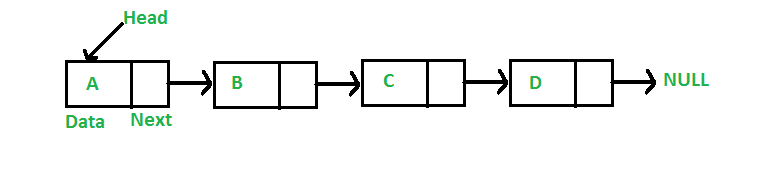
**Interview questions on Arrays:**

1. **Mention some advantages and disadvantages of Arrays.**
2. **What will happen if you do not initialize an Array?**
3. **Can a Negative number be passed in Array size?**
4. **When will we get ArrayStoreException?**
5. **When will we get ArrayIndexOutOfBounds Exception?**
6. **Find the minimum and maximum element in an array**
7. **C program to find occurrence of an element in one dimensional array**
8. **Reverse an array in C.**





**What is Linked List**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

* **A Node Creation:**

class Node {

public:

int data;

Node\* next;

};

void push(Node\*\* head\_ref, int new\_data)

{

// 1. allocate node

Node\* new\_node = new Node();

// 2. put in the data

new\_node->data = new\_data;

// 3. Make next of new node as head

new\_node->next = (\*head\_ref);

// 4. Move the head to point to

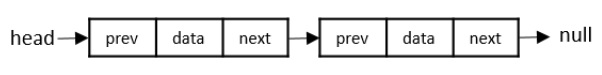
// the new node

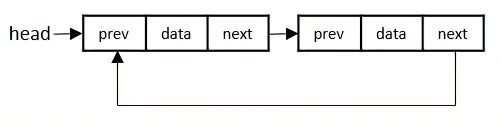
(\*head\_ref) = new\_node;

}

* If arrays accommodate similar types of data types, linked lists consist of elements with different data types that are also arranged sequentially.
* In the case of arrays, the size is limited to the definition, but in linked lists, there is no defined size. Any amount of data can be stored in it and can be deleted from it.
* There are three types of linked lists −

1. Singly Linked List − The nodes only point to the address of the next node in the list.
2. Doubly Linked List − The nodes point to the addresses of both previous and next nodes.
3. Circular Linked List − The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

**Doubly Linked List**:

**Circular Linked List**:

These operations are performed on Singly Linked Lists as given below −

Insertion − Adds an element at the beginning of the list.

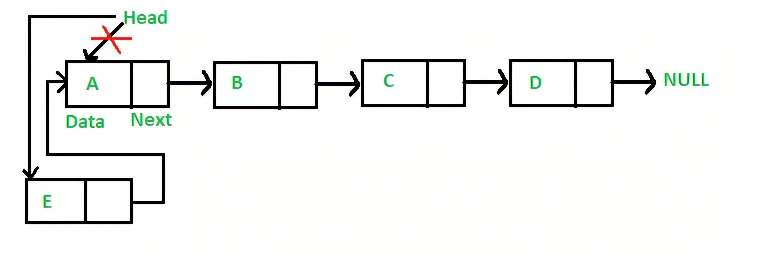
Deletion − Deletes an element at the beginning of the list.

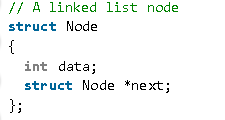
Display − Displays the complete list.

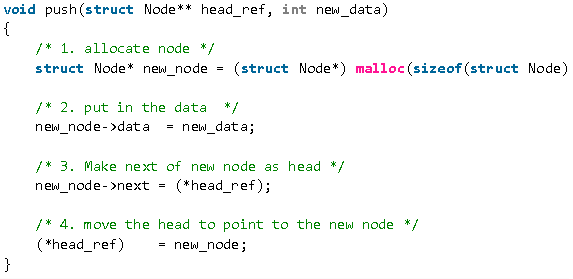
Search − Searches an element using the given key.

Delete − Deletes an element using the given key.

**Insertion at Beginning:**

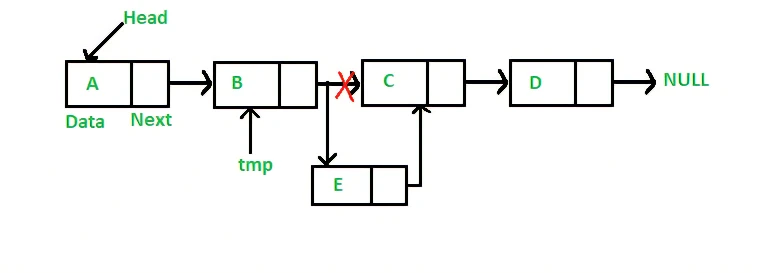
****

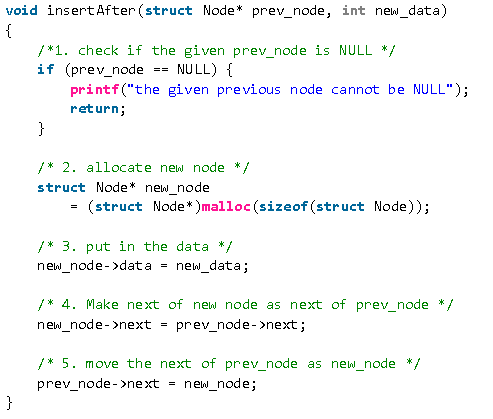




**Complexity Analysis:**

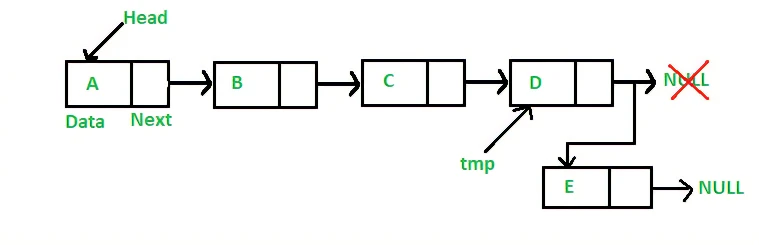
Time Complexity: O(1), We have a pointer to the head and we can directly attach a node and change the pointer. So the Time complexity of inserting a node at the head position is O(1) as it does a constant amount of work.

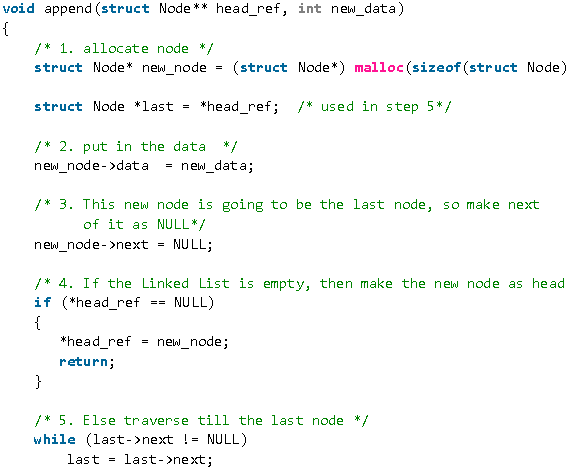
**Insertion after a given node:**

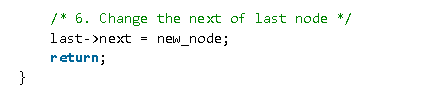


**Complexity Analysis:**

Time complexity: O(1), since prev\_node is already given as argument in a method, no need to iterate over list to find prev\_node.

**Insertion at the end:**

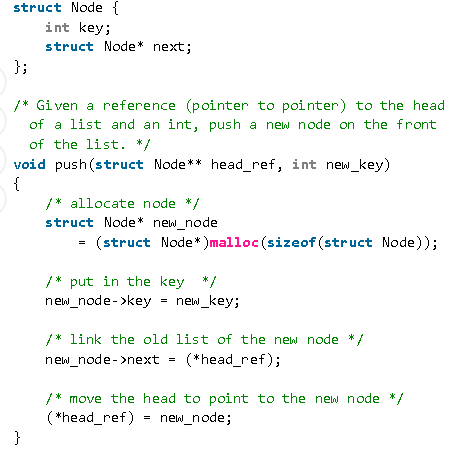


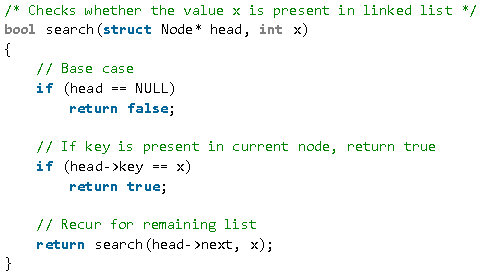


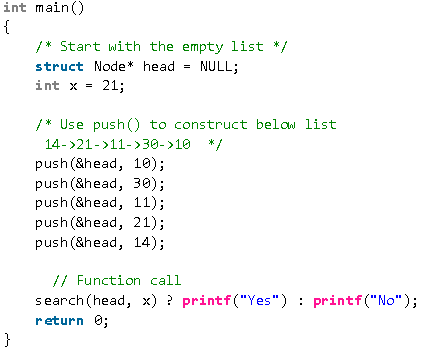
**Search an element in a Linked List**

Follow the below steps to solve the problem:

* If the head is NULL, return false.
* If the head’s key is the same as X, return true;
* Else recursively search in the next node.







**Why Linked List?**

Arrays can be used to store linear data of similar types, but arrays have the following limitations:

* **The size of the arrays is fixed**: So, we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
* **Insertion of a new element / Deletion of an existing element in an array of elements is expensive:** The room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.

**Advantages of Linked Lists over arrays:**

* Dynamic Array.
* Ease of Insertion/Deletion.
* Insertion at the beginning is a constant time operation and takes O(1) time, as compared to arrays where inserting an element at the beginning takes O(n) time, where n is the number of elements in the array.

**Drawbacks of Linked Lists:**

* Random access is not allowed. We have to access elements sequentially starting from the first node (head node). So, we cannot do a binary search with linked lists efficiently with its default implementation.
* Extra memory space for a pointer is required with each element of the list.
* Searching for an element is costly and requires O(n) time complexity.
* Sorting of linked lists is very complex and costly.
* Appending an element to a linked list is a costly operation, and takes O(n) time, where n is the number of elements in the linked list, as compared to arrays that take O(1) time.