Search

**Sequential Seacrch** -> In this, the list or array is traversed sequentially and every element is checked.

**Interval Search** -> These algorithms are specifically designed for searching in sorted data-structures.

**Linear Search** -> a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

**Pseudocode**:

procedure linear\_search (list, value)

for each item in the list

if match item == value

return the item's location

end if

end for

end procedure

Examples:

Input : arr[] = {10, 20, 80, 30, 60, 50,

110, 100, 130, 170}

x = 110;

Output : 6

Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,

110, 100, 130, 170}

x = 175;

Output : -1

Element x is not present in arr[].

**Binary Search**

fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. Our array needs to be sorted already.

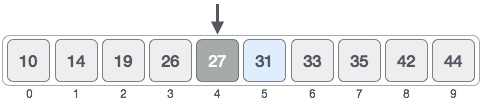


First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

mid = (low+high)/2

Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

**(NOTE)**

**low = mid + 1**

**high = mid - 1**

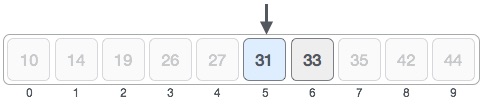
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



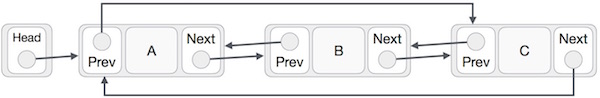
We conclude that the target value 31 is stored at location 5.

**linked list**

 sequence of data structures, which are connected together via links.

**Types**

* **Simple Linked List** − Item navigation is forward only. 
* **Doubly Linked List** − Items can be navigated forward and backward. 
* **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.



**Stack**

A stack is an Abstract Data Type (ADT), This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



[empty()](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/) – Returns whether the stack is empty – Time Complexity : O(1)   
[size()](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/) – Returns the size of the stack – Time Complexity : O(1)   
[top()](https://www.geeksforgeeks.org/stack-top-c-stl/) – Returns a reference to the top most element of the stack – Time Complexity : O(1)   
[push(g)](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)   
[pop()](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) – Deletes the top most element of the stack – Time Complexity : O(1)

**Queue**

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first. 

**Enqueue,dequeuer,front,rear**

**Sorting**

Sorting algorithm specifies the way to arrange data in a particular order. Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

**Bubble Sort ( check in pair)**

This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

**Insertion Sort ( start from first element after swap also check sorted array again.)**

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

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

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

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.

Insertion Sort

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

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

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

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Selection Sort( select a smallest element and compare from unsorted array.)**

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



**Quick Sort( select pivot and divide data into 2 arrays.)**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

**For reference**

<https://www.tutorialspoint.com/data_structures_algorithms/images/quick_sort_partition_animation.gif>

<https://www.youtube.com/watch?v=tWCaFVJMUi8>

**for reference.**

**https://www.tutorialspoint.com/data\_structures\_algorithms**

**Graphs**

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

**Depth First Search**

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

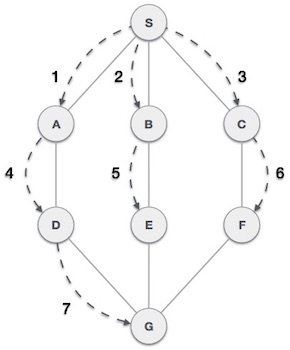


|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Depth First Search Step One | Initialize the stack. |
| 2 | Depth First Search Step Two | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 | Depth First Search Step Three | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 | Depth First Search Step Four | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 | Depth First Search Step Five | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |
| 6 | Depth First Search Step Six | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node

**Breadth First Search**

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Breadth First Search Step One | Initialize the queue. |
| 2 | Breadth First Search Step Two | We start from visiting **S** (starting node), and mark it as visited. |
| 3 | Breadth First Search Step Three | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4 | Breadth First Search Step Four | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5 | Breadth First Search Step Five | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6 | Breadth First Search Step Six | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7 | Breadth First Search Step Seven | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

**Linked List**

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

* The size of the array must be known in advance before using it in the program.
* Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
* All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

* It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
* In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

* **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
* **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
* **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
* **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

* **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
* **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
* **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

|  |  |
| --- | --- |
| **Array** | **Linked list** |
| An array is a collection of elements of a similar data type. | A linked list is a collection of objects known  as a node where node consists of two parts,  i.e., data and address. |
| Array elements store in a contiguous memory location. | Linked list elements can be stored anywhere  in the memory or randomly stored. |
| Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time. | The Linked list works with dynamic memory  . Here, dynamic memory means that the memory  size can be changed at the run time  according to our requirements. |
| Array elements are independent of each other. | Linked list elements are dependent on each other.  As each node contains the address of the next node  so to access the next node, we need to access its  previous node. |
| Array takes more time while performing any operation like insertion, deletion, etc. | Linked list takes less time while performing any  operation like insertion, deletion, etc. |
| Accessing any element in an array is faster as the element in an array can be directly accessed through the index. | Accessing an element in a linked list is slower as it starts  traversing from the first element of the linked list. |
| In the case of an array, memory is allocated at compile-time. | In the case of a linked list, memory is allocated at  run time. |
| Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused. | Memory utilization is efficient in the case of a linked  list as the memory can be allocated or deallocated at the  run time according to our requirement. |

|  |  |  |
| --- | --- | --- |
| **Basis for comparison** | **Stack** | **Queue** |
| **Principle** | It follows the principle LIFO (Last In- First Out), which implies that the element which is inserted last would be the first one to be deleted. | It follows the principle FIFO (First In -First Out), which implies that the element which is added first would be the first element to be removed from the list. |
| **Structure** | It has only one end from which both the insertion and deletion take place, and that end is known as a top. | It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion. |
| **Number of pointers used** | It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack. | It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue. |
| **Operations performed** | It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list. | It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue. |
| **Examination of the empty condition** | If top==-1, which means that the stack is empty. | If front== -1 or front = rear+1, which means that the queue is empty. |
| **Examination of full condition** | If top== max-1, this condition implies that the stack is full. | If rear==max-1, this condition implies that the stack is full. |
| **Variants** | It does not have any types. | It is of three types like priority queue, circular queue and double ended queue. |
| **Implementation** | It has a simpler implementation. | It has a comparatively complex implementation than a stack. |
| **Visualization** | A Stack is visualized as a vertical collection. | A Queue is visualized as a horizontal collection. |