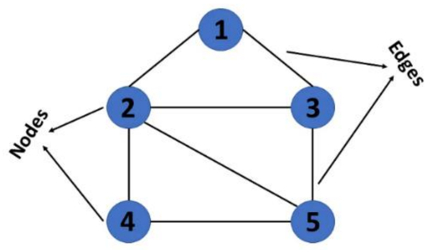
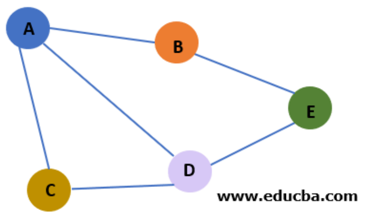
Graph

- A Graph is a non-linear data structure consisting of vertices and edges.
- The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E).**



- This graph has a set of vertices V= { 1,2,3,4,5} and
- a set of edges E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,5) }.

Graph Terminologies



Adjacent Nodes: Two nodes are called adjacent if they are connected through an edge. Node A is adjacent to nodes B, C, and D in the above example, but not to node E.

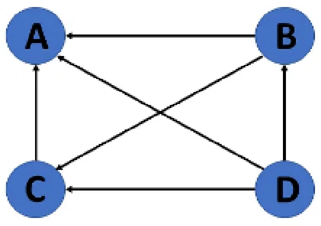
Directed Graph

A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.An edge of a directed graph is given as an ordered pair(u,v) of nodes in G.

For an edge(u,v), the edge begins at u and terminates at v.

U is know as the origin or initial point of e, and v is known as the destination or terminal point of e.

U is the predecessor of v and v is the successor of u.Nodes u and v ate adjacent to each other.



Out-degree of a node: The out – degree of a node u, written as outdeg(u), is the number of edges that originate at u.

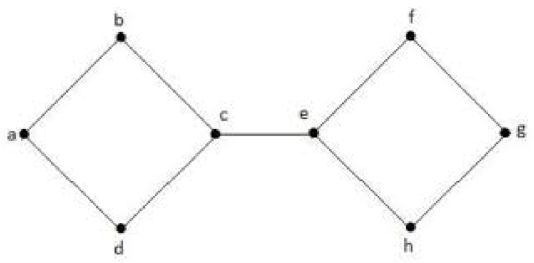
In-degree of a node: The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

Degree of a node: The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u)=indeg(u)+outdeg(u)

Isolated vertex: A vertex with degree zero such a vertex is not an end-point of any edge.

Pendant vertex: A vertex with degree one.

Cut vertex: A vertex which when deleted would disconnect the remaining graph.

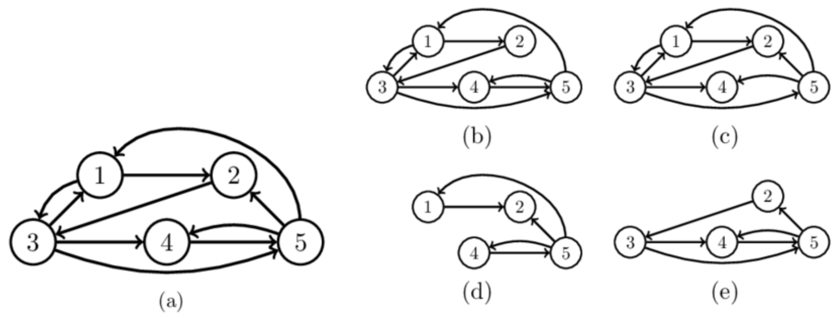


Source: A node u is known as source if it has a positive out-degree but a zero in-degree.

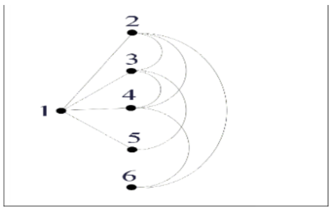
Sink: A node is known as a sink it it has a positive in-degree but a zero out-degree.

Reachability: A bode v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Strongly connected directed graph : A graph is said to be strongly connected it and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.



Unilaterally connected graph: A diagraph is said to be unilaterally connected if there exists a path between any pair of nodes u,v in G such that there is a path from u to v or path from v to u, but not both.

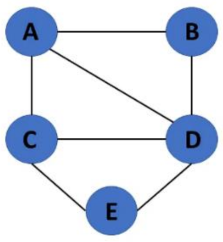


Weakly connected graph: A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. The nodes in a weakly connected graph must have either out-degree or in-degree of at least 1.

Simple directed graph: A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. It may contain cycles with an exception that it cannot have more than on loop at a given node.

Undirected Graph

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.

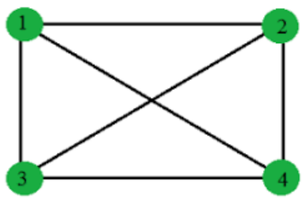


Degree of a node :

- Degree of a node u, deg(u), is the total number of edges containing the node u.
- If deg(u)=0, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular Graph

- It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree.
- A regular graph with vertices of degree k is called a k-regular graph or a regular graph of degree k.



3 Regular

Path: A finite sequence of edges that connects a series of vertices is called a path.

Closed Path: A path is said to be closed if it starts and ends at the same node.

Simple Path: A path is considered to be simple if all of its nodes are distinct, except the initial and last node.

Cycle: A path in which the first and last vertices are same.

A simple cycle: Has no repeated edges or vertices (except the first and last vertices)

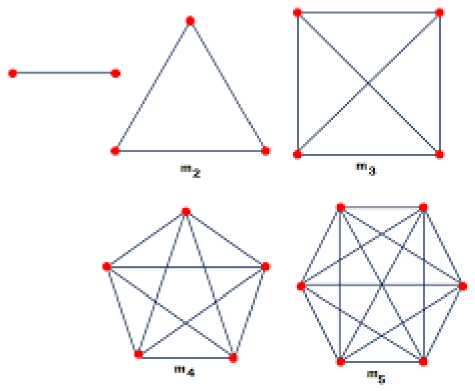
Connected graph: A graph is said to be connected if for any two vertices(u,v) in v there is a path from u to v.

There is no isolated nodes in a connected graph.

A connected graph that does not have any cycle is called a tree, Therefore, a tree is treated as a special graph.

Complete graph: A graph G is said to be complete if all its nodes are fully connected. That is , there is a path from one node to every other node in the graph.

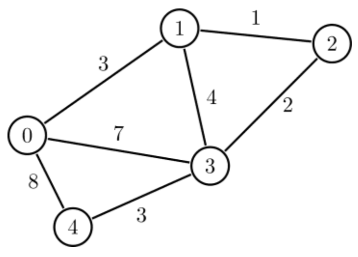
A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.



Labelled graph or Weighted graph:A graph is said to be labelled if every edges in the graph is assigned some data.

In a weighted graph, the edges of the graph are assigned some weight or length.

The weight of an edge denoted by w€ is a positive value which indicates the cost of traversing the edge.



Multiple edges: Distinct edges which connect the same end-points are called multiple edges or parallel edges.

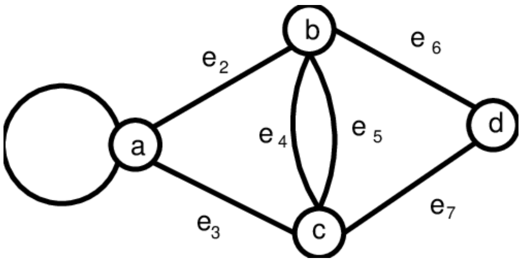
That is , e=(u,v) and e'=(u,v) are known as multiple edges of G.

Loop: An edge that has identical end-points is called a loop. That is e=(u,u).

It adds 2 to the degree of the node.

Multi-graph : A graph with multiple edges or loops is called a multi-graph.

Size of a graph: The size of a graph is the total number of edges in it.



Applications of Graph

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams.
- These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

REPRESENTATION OF GRAPH

- Three common ways of storing graphs in the computer’s memory.

Sequential representation by using an adjacency matrix.

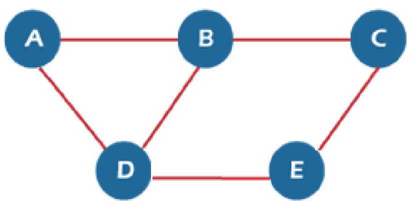
Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.

Adjacency multi-list which is an extension of linked representation.

Sequential representation (or, Adjacency matrix representation)

- An adjacency matrix is used to represent which nodes are adjacent to one another.
- By definition, two nodes are said to be adjacent if there is an edge connecting them.
- For any graph G having n nodes, the adjacency matrix will have the dimension of n*n.
- In adjacency matrix, the rows and columns are labelled by graph vertices.
- An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

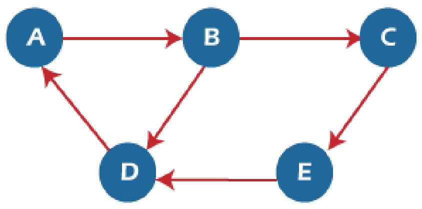
- aij = 1 {if there is a path exists from Vi to Vj}
- aij = 0 {Otherwise}
- The entries in the matrix depend on the ordering of the nodes in G. A change in the order of nodes will result in a different adjacency matrix.



Undirected Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

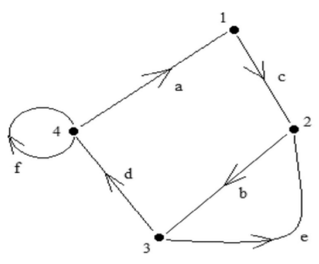
Adjacency Matrix



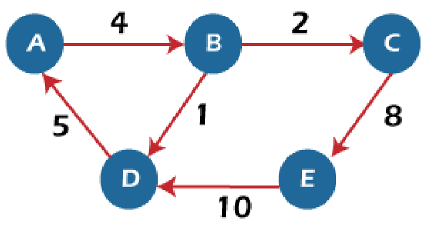
Directed Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix



$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$



weighted Directed Graph

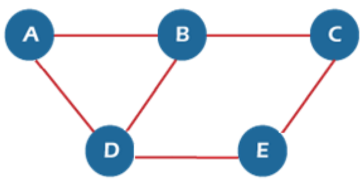
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Adjacency Matrix

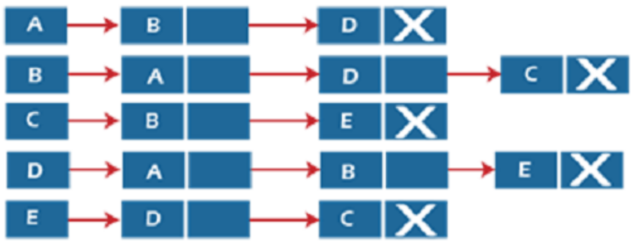
- For a simple graph(that has no loops) , the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of adjacency matrix O(n^2) , where n is the number of nodes in the graph.
- Number of 1s in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Linked list representation Adjacency List

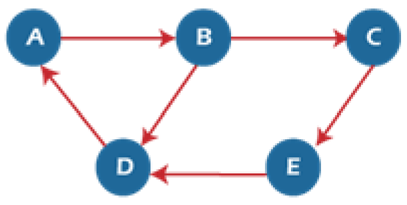
- An adjacency list consists of a list of all nodes in G.
- Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.
- The key advantages of using an adjacency list are:
- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graph that have a small-to-moderate number of edges [sparse graphs].
- Otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list.
- Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.



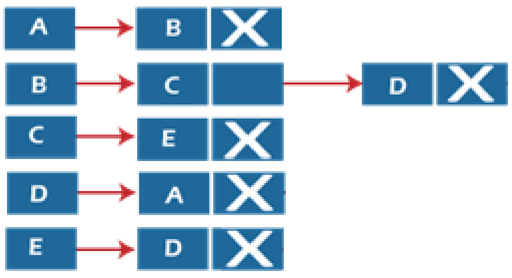
Undirected Graph



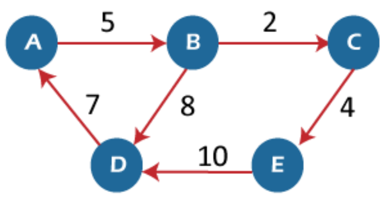
Adjacency List



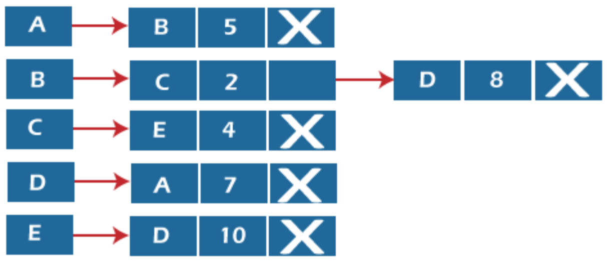
Directed Graph



Adjacency List



Weighted Directed Graph



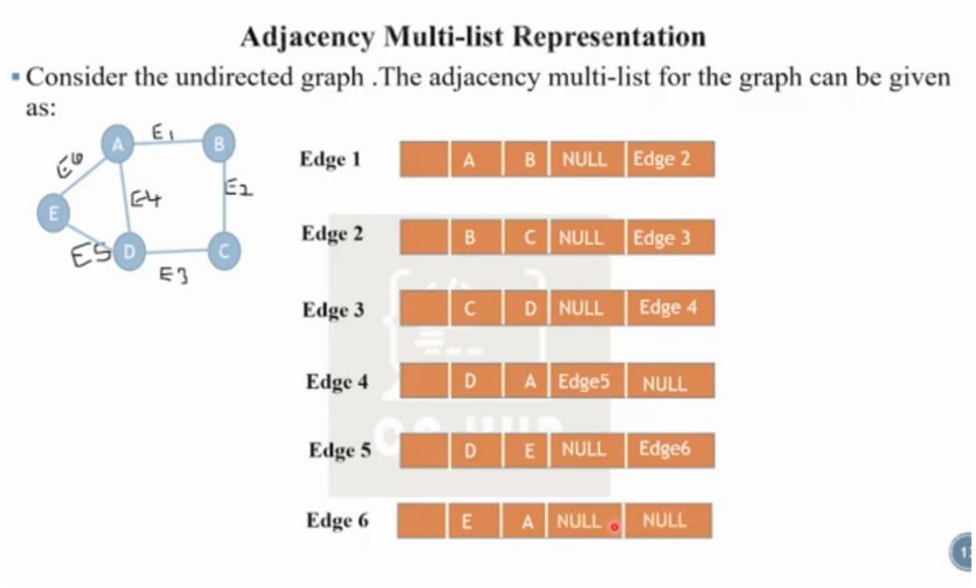
Adjacency List

Adjacency Multi-list Representation

- Multi-list representation is the modified version of adjacency lists.
- Adjacency multi-list is an edge-based rather than a vertex-based representation of graph.
- A multi-list representation basically consists of two parts – a dictionary of nodes information and a set of linked lists storing information about edges.

- In a multi-list representation, the information about an edge(vi,vj) of an undirected graph can be stored using the following attributes:
- M : A single bit field to indicate whether the edge has been examined or not.
- Vi: A vertex in the graph that is connected to vertex Vj by an edge.
- Vj: A vertex in the graph that is connected to vertex Vi by an edge.
- Link i for vi: A link that points to another node that has an edge incident on vi
- Link j for vi: A link that points to another node that has an edge incident on vj

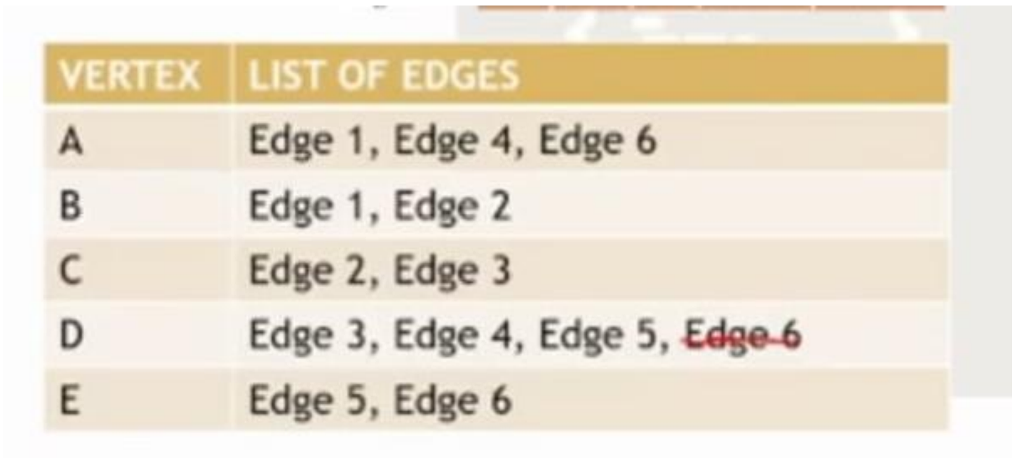
Consider the undirected graph. The adjacency multi-list for the graph can be given as:



Adjacency Multi-list Representation

- Consider the undirected graph .The adjacency multi-list for the graph can be given as:

Edge 1		A	B	NULL	Edge 2
Edge 2		B	C	NULL	Edge 3
Edge 3		C	D	NULL	Edge 4
Edge 4		D	A	Edge5	NULL
Edge 5		D	E	NULL	Edge6
Edge 6		E	A	NULL	NULL



VERTEX	LIST OF EDGES
A	Edge 1, Edge 4, Edge 6
B	Edge 1, Edge 2
C	Edge 2, Edge 3
D	Edge 3, Edge 4, Edge 5, ~~Edge 6~~
E	Edge 5, Edge 6

Graph Traversal

Depth-First and Breadth-First Traversal

Graph Traversal

- Graph traversal is a technique used for a searching vertex in a graph.
- Graph traversal is a process of visiting each node in a graph, usually from a starting node, and keeping track of which nodes have been visited.
- There are two graph traversal techniques

- DFS (Depth First Search)
- BFS (Breadth First Search)

Breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing.

Depth-first search schemes uses a stack.

But both these algorithms make use of a variable STATUS

Status	**State of the node**	**Description**
1	Ready	The initial state of the node N
2	Waiting node	N is placed on the queue or stack and waiting to be processed
3	Processes Node	N has been completely processed

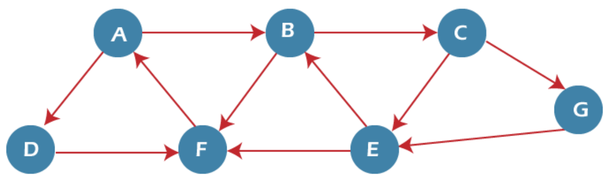
Breadth-First Search Algorithm

- Breadth-First Search(BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, the algorithm explored neighbor nodes and so on ,until it finds the goal.
- This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once.
- This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state if the node.
- BFS traversal of a graph produces a spanning tree as final result.

Algorithm

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set
their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

Example:



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Complexity of BFS algorithm

- Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is O(V+E), since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).
- The space complexity of BFS can be expressed as O(V), where V is the number of vertices.

Applications of Breadth-First Search

- Breadth-first search can be used to solve many problems such as:
- Finding all connected components in a graph G.

- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v of an unweighted graph.

Depth-First Search Algorithm

- Depth-first search algorithm processes by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node.
- Then , it examines each node N along a path P which begins at A.
- The algorithm proceeds like this until we reach a dead-end(end of path P)
- On reaching the dead end, we backtrack to find another path P.
- The algorithm terminates when backtracking leads back to the starting nodeA.
- In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges.
- This algorithm is similar to the in-order traversal of a binary tree.
- For it’s a stack is used and a variable STATUS represents the current state .

Algorithm

Step1**:** Set STATUS=1(ready state) for each node in G

Step2: Push the starting node A on the stack and set its STATUS=2(waiting state)

Step3: Repeat steps 4 and 5 until Stack is empty.

Step4: Pop the node N. Process it and set its STATUS=3(Processed state)

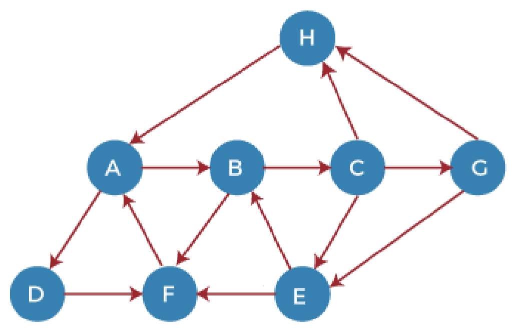
Step5: Push on the stack all the neighbours of N that are in the ready state

(whose STATUS=1) and their STATUS=2(waiting state)

[END OF LOOP]

Step6: EXIT

Example of DFS algorithm



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H   STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

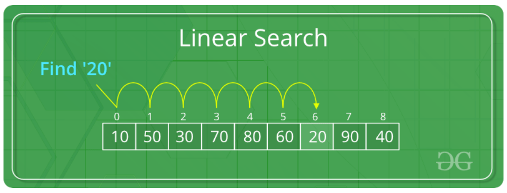
The space complexity of the DFS algorithm is O(V).

Applications of Depth First Search:

- Finding a path between two specified nodes, u and v of an unweighted graph.
- Finding a path between two specified nodes u and v of weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Searching

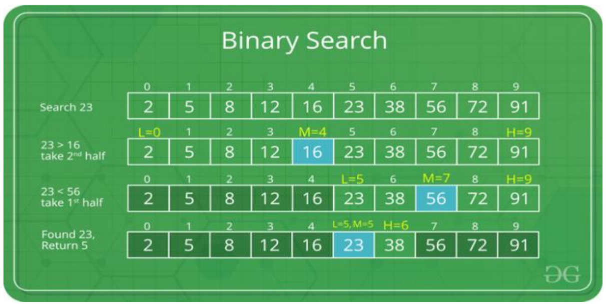
- Searching in data search refers to the process of looking for specific information or content within a dataset or a collection of data.
- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.
- Based on the type of search operation, 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.
- **Linear Search to find the element "20" in a given list of numbers**



-
- **Binary 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.

Binary Search to find the element "23" in a given list of numbers

-

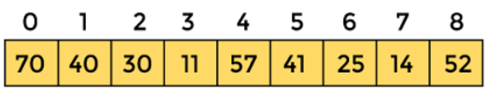


Linear Search Algorithm

- Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is O(n).

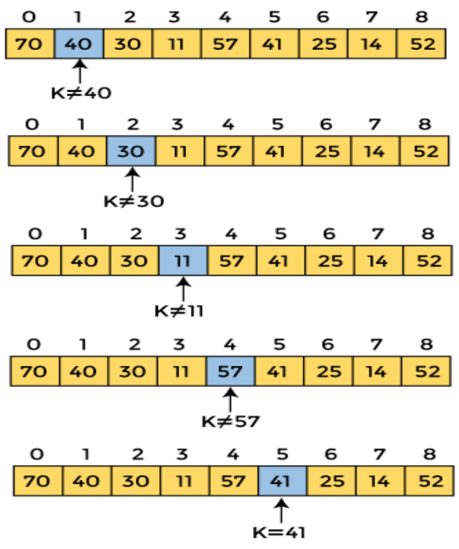
Algorithm

- Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
- Step 1: set pos = -1
- Step 2: set i = 1
- Step 3: repeat step 4 while i <= n
- Step 4: if a[i] == val
- set pos = i
- print pos
- go to step 6
- [end of if]
- set i = i + 1
- [end of loop]
- Step 5: if pos = -1
- print "value is not present in the array "
- [end of if]
- Step 6: exit



Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

Program

```
#include <stdio.h>

int linearSearch(int a[], int n, int val) {

  // Going through array sequencially

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

    {  if (a[i] == val)

        return i+1;  }

  return -1;

}

int main() {

  int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array

  int val = 41; // value to be searched

  int n = sizeof(a) / sizeof(a[0]); // size of array

  int res = linearSearch(a, n, val); // Store result

  printf("The elements of the array are - ");

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

  printf("%d ", a[i]);
```

```
 printf("\nElement to be searched is - %d", val);

 if (res == -1)

 printf("\nElement is not present in the array");

 else

 printf("\nElement is present at %d position of array", res);

 return 0;

}
```

Binary Search Algorithm

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match

Algorithm

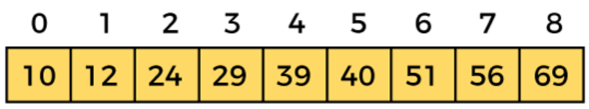
```
Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1
```

print "value is not present in the array"

[end of if]

Step 6: exit

Let the elements of array are –



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

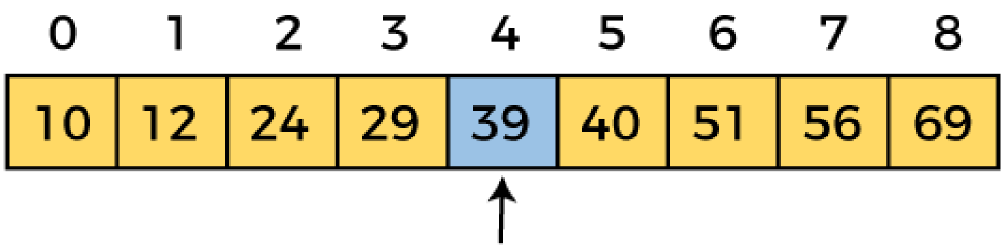
1. mid = (beg + end)/2

So, in the given array -

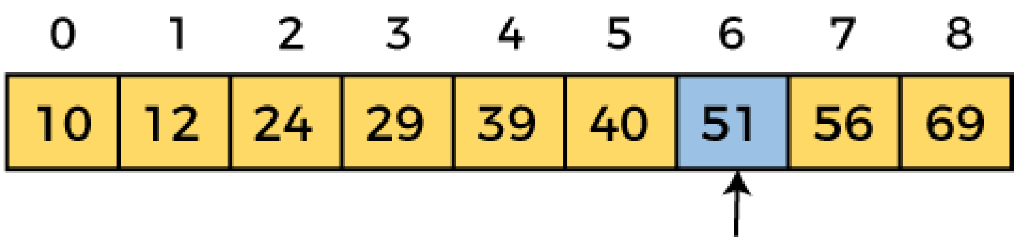
beg = 0

end = 8

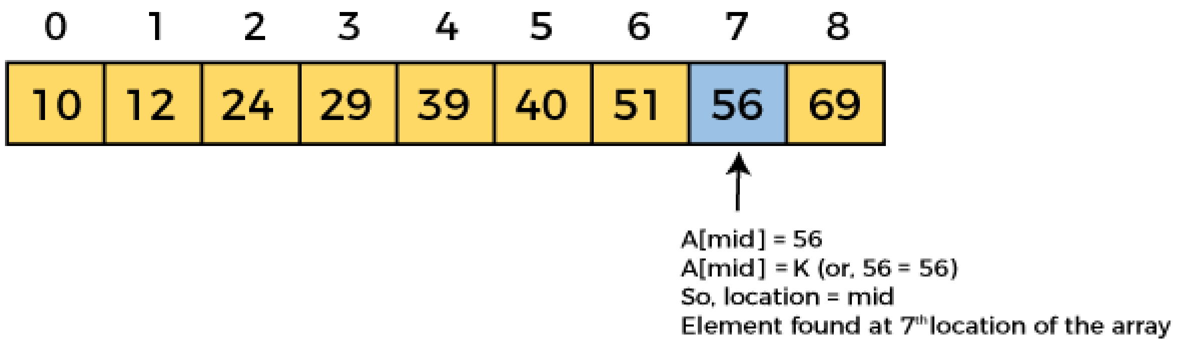
mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.



A[mid] = 39
A[mid] < K (or,39 < 56)
So, beg = mid + 1 = 5, end = 8
Now, mid =(beg + end)/2 = 13/2 = 6



A[mid] = 51
A[mid] < K (or, 51 < 56)
So, beg = mid + 1 = 7, end = 8
Now, mid =(beg + end)/2 = 15/2 = 7



Now, the element to search is found. So algorithm will return the index of the element matched.

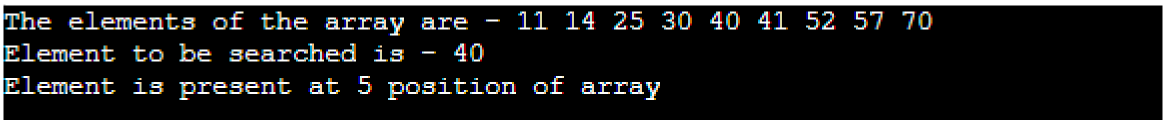
Binary Search complexity

Time Complexity

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

```
#include <stdio.h>
int binarySearch(int a[], int beg, int end, int val)
{
   int mid;
   if(end >= beg)
   {      mid = (beg + end)/2;
/* if the item to be searched is present at middle */
      if(a[mid] == val)
      {
         return mid+1;
      }
         /* if the item to be searched is smaller than middle, then it can only be in left subarray */
      else if(a[mid] < val)
      {
            return binarySearch(a, mid+1, end, val);
        }
            /* if the item to be searched is greater than middle, then it can only be in right subarray */
        else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
int main() {
  int a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
  int val = 40; // value to be searched
  int n = sizeof(a) / sizeof(a[0]); // size of array
  int res = binarySearch(a, 0, n-1, val); // Store result
  printf("The elements of the array are - ");
  for (int i = 0; i < n; i++)
  printf("%d ", a[i]);
  printf("\nElement to be searched is - %d", val);
  if (res == -1)
  printf("\nElement is not present in the array");
  else
  printf("\nElement is present at %d position of array", res);
  return 0;
}
```

Output



```
The elements of the array are - 11 14 25 30 40 41 52 57 70
Element to be searched is - 40
Element is present at 5 position of array
```

Hashing

Importance of Hashing

- The time complexity of search algorithms depends upon the number of elements in the list.
- Linear search and binary search algorithms have a time complexity of O(n) and O(log n) respectively.
- **Hashing is a search technique which is independent of the number of elements in the list.**
- It uniquely identify a specific item from a group of similar items.
- Therefore, hashing allows searching, updating and retrieval of data in a constant time, that is O(1).

Hashing

- Hashing is a technique to make things more efficient by effectively narrowing down the search at the outset.
- Hashing is a technique or process of mapping keys and values into the hash table by using a hash function.
- It is done for faster access to elements.
- The efficiency of mapping depends on the efficiency of the hash function used.
- The major 2 components of hashing are
  - Hash table
  - Hash function

Hash Table

- A hash table is a data structure that stores data as key-value pairs.
- Each key is matched to a value in the hash table by a hash function.
- Keys are used to index the data.
- Values specifies the data associated with the keys.
- A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).

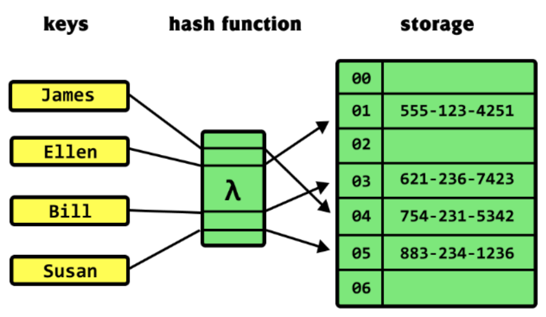
Hash function

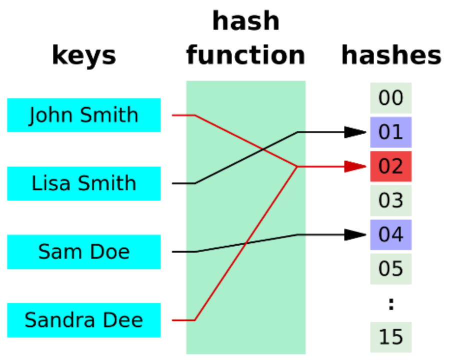
- A hash function is a mathematical formula, used for mapping keys into table indices.
- The process of mapping the keys to corresponding indices in a hash table is called hashing.
- Assume k is a key and h(x) is a hash function.
- In a hash table , an element linked with key k will be stored at the index h(k).
- If two keys point to the same memory location it is known as collision.

Characteristics of a good hash function

- It should be very simple and quick to compute.

- The cost of execution must be low.
- It should distribute the hash addresses as evenly as possible within L, so there are fewer collisions.
- The same hash value must be generated for a given input value.





- Generally hash function use numeric keys.
- In real-world applications where we can have alphanumeric keys rather than simple numeric keys.
- In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric keys.

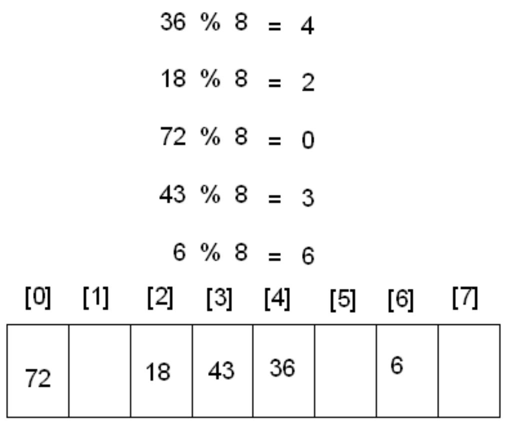
Types of Hash function

- Different types of hash functions include
- Division method
- Multiplication method
- Mid square method
- Folding method

Division method

- It is the most important method of hashing an integer x.
- This method divides x by M and then uses the remainder obtained.
- In this case, the hash function can be given as
- **$h(x) = x \bmod M$**
- The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast.
- However, extra care should be taken to select a suitable value for M.

For example: Suppose K=8, the hash values can be calculated as :



36 % 8 = 4

18 % 8 = 2

72 % 8 = 0

43 % 8 = 3

6 % 8 = 6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36		6	

Multiplication method

The following steps are involved in multiplication method.

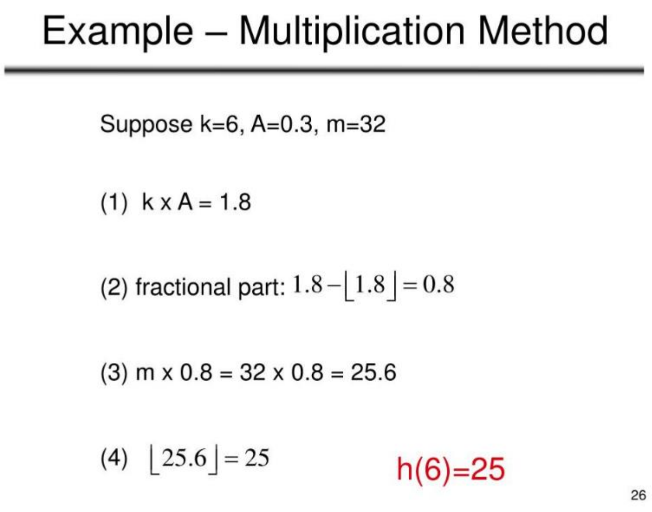
- Choose a constant A such that 0<A<1
- Multiply K by A
- Extract the fractional part
- Multiply the result by the size of hash table (m).

The hash function can be given as :

H(k) = [M (KA mod 1)]

Eg: Given a hash table of size 1000 , map the key 12345 to an appropriate location in the hash table.

- Use A=0.618033, m=1000, and k=12345
- H(12345) = [1000 (12345*0.618033 mod 1)]
- H(12345) = [1000 (7629.617385 mod 1)]
- H(12345) = [1000 (0.617385)]
- H(12345) = [.617.385]
- H(12345) = 617



Example – Multiplication Method

Suppose k=6, A=0.3, m=32

(1) k x A = 1.8

(2) fractional part: $1.8 - \lfloor 1.8 \rfloor = 0.8$

(3) m x 0.8 = 32 x 0.8 = 25.6

(4) $\lfloor 25.6 \rfloor = 25$ h(6)=25

26

Mid square method

- The mid square method is a good hash function which works in two steps:
- Step1: Square the value of the key. That is , find k^2
- Step2: Extract the middle r digits of the result obtained in step1.

In the mid-square method, the same r digits must be chosen from all the keys.

Therefore, the hash function can be given as

H(k) = s

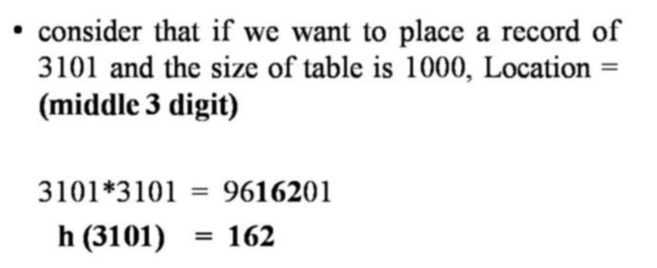
Where s is obtained by selecting r digits from K^2

Suppose k=1234.Then let's find the hash value for a hash table of size 100.

Since the index of hash table varies between 0 and 99, we can choose r=2.

Then K=1234, K^2= 1522756, h(1234) = 27

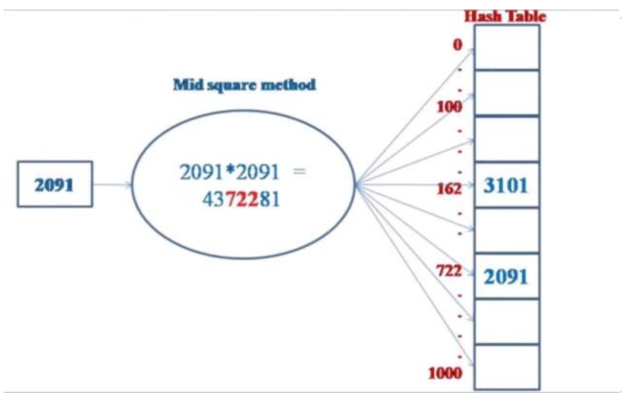
Exaples:

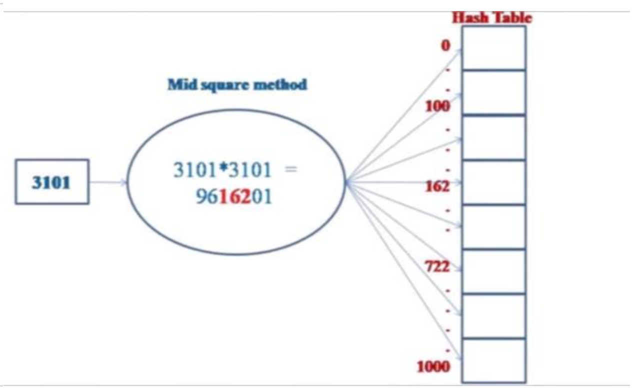


- consider that if we want to place a record of 3101 and the size of table is 1000, Location = **(middle 3 digit)**

3101*3101 = 96**162**01

h (3101) = 162

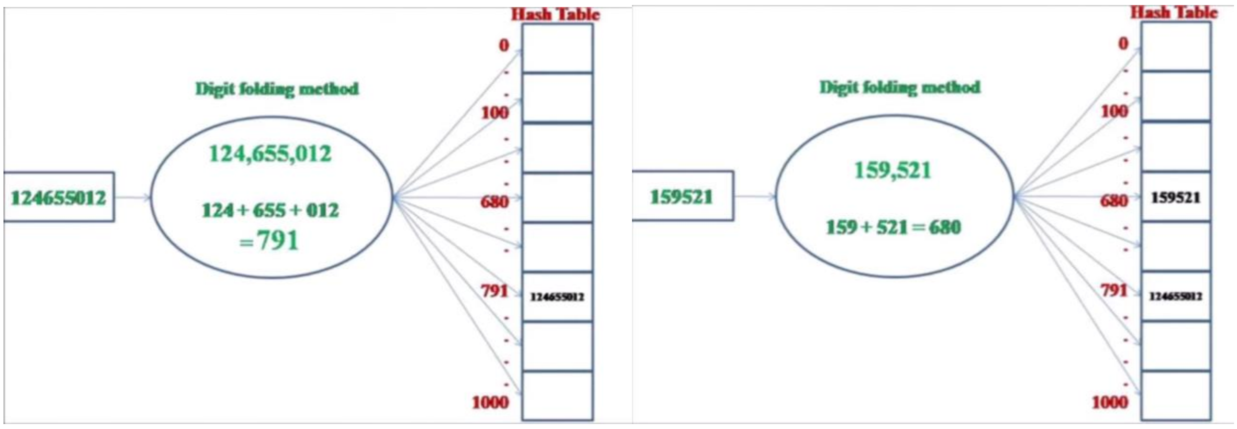




Folding method

- The key k is divided into a number of parts of same length k1,k2,…..kr.
- And they are added together ignoring last carry if any.
- The hash function can be given by:
- H(k) = k1+k2+………..+kr

- For example: consider a record of 124655012 then it will be divided into parts i.e. 124, 655, 012. After dividing the parts combine these parts by adding it.
- H(key)=124+655+012 =791



Collisions

- Collisions occur when the hash function maps two different keys to the same location.
- Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called collision resolution technique, is applied.

There are two types of collision resolution techniques.

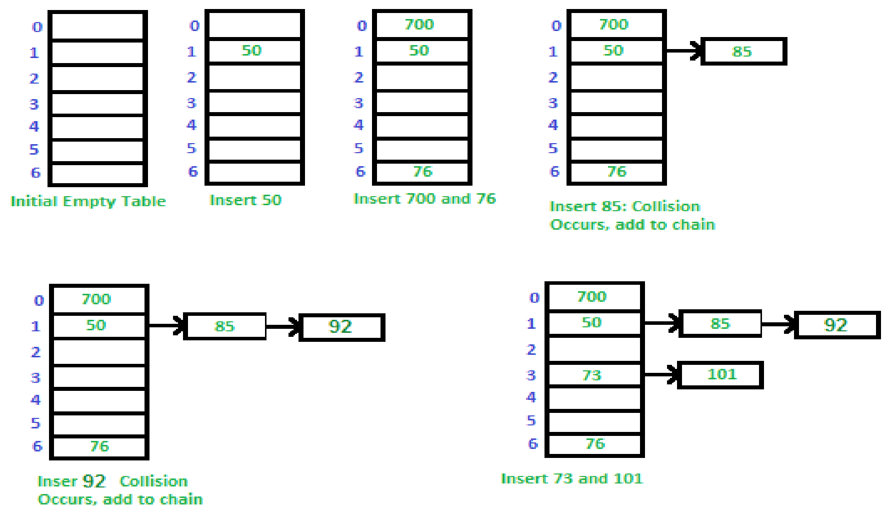
- Separate chaining (open hashing)
- Open addressing (closed hashing)

Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

- Here, all those elements that hash into the same slot index are inserted into a linked list.
- In separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, add to chain
Inser 92 Collision Occurs, add to chain
Insert 73 and 101

Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself.

The following techniques are used in open addressing:

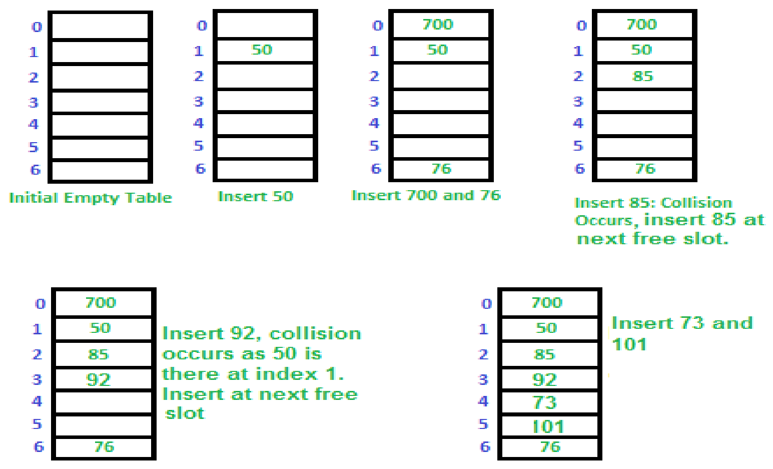
- Linear probing
- Quadratic probing
- Double hashing

1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

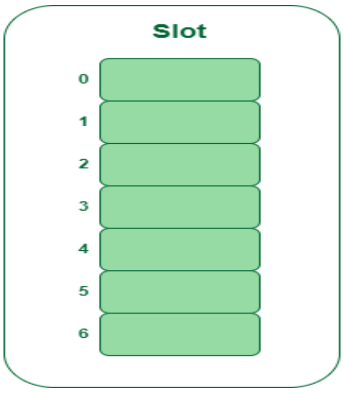
2. Quadratic Probing

This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

let hash(x) be the slot index computed using hash function.

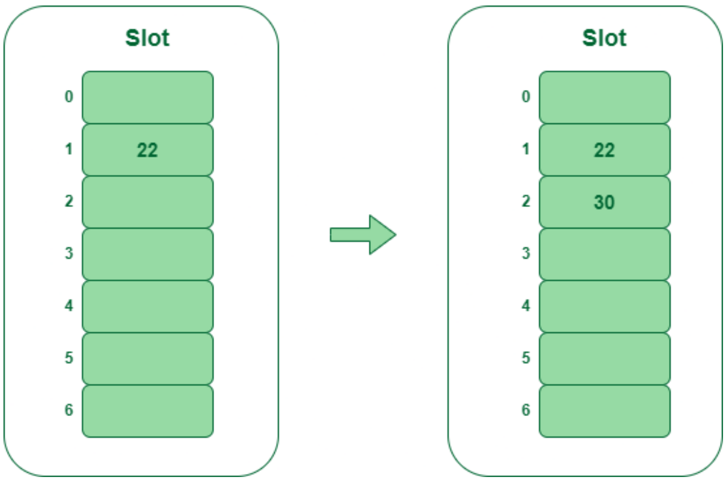
*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*
…………………………………………

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2. Insert = 22, 30, and 50.
Step 1: Create a table of size 7.



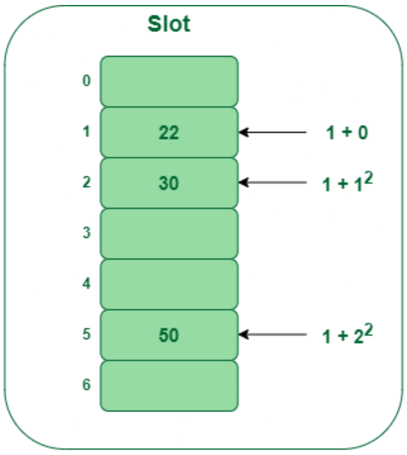
Step 2 – Insert 22 and 30

- Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Step 3: Inserting 50

- Hash(50) = 50 % 7 = 1
- In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
- Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
- Now, cell 5 is not occupied so we will place 50 in slot 5.



3. Double Hashing

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i**th rotation.

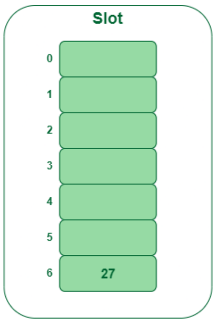
let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*
....

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

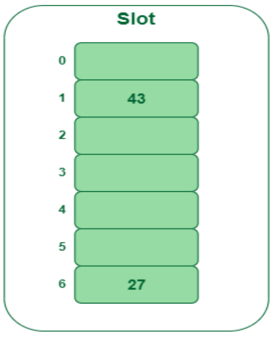
Step 1: Insert 27

- 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Step 2: Insert 43

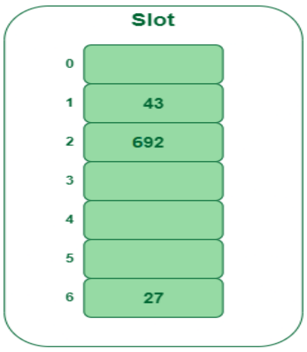
- 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



Step 3: Insert 692

- 692 % 7 = 6, but location 6 is already being occupied and this is a collision
- So we need to resolve this collision using double hashing.

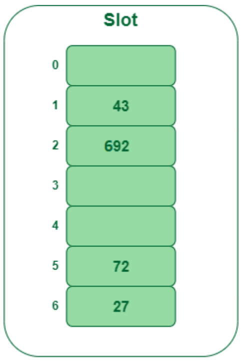
*h_{new} = [h1(692) + i * (h2(692)] % 7*
*= [6 + 1 * (1 + 692 % 5)] % 7*
= 9% 7
= 2
Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.



Step 4: Insert 72

- 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
- So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
= 5 % 7
= 5,
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Sorting

- It is a technique of arranging data in a particular order.
- Usually based on some linear relationship among the data tem
- Numerical sorting – ascending or descending
- Sorting makes the data easier to search or analyse.

Sorting Algorithms

- Bubble sort
- Insertion sort
- Selection sort
- Quick sort
- Merge sort

Terminologies in Sorting

- Internal Sorting :- When all data is placed in memory
- External Sorting:- When all data to be sorted cannot be placed in memory at a time (eg.merge sort).
- Inplace :- Sort the list only by modifying the order of elements within the list.It uses constant extra space for producing output(Eg:Bubble sort,insertion sort)
- Out-of-place/non -in-place:-It uses additional space for sorting.(E:Merge sort)
- Stable:-Relative order of equal keys do not change.(Eg:Bubble Sort)

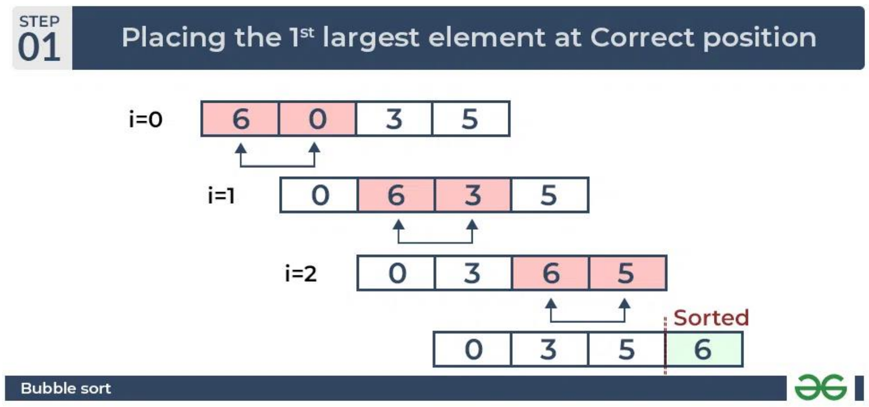
Bubble sort

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

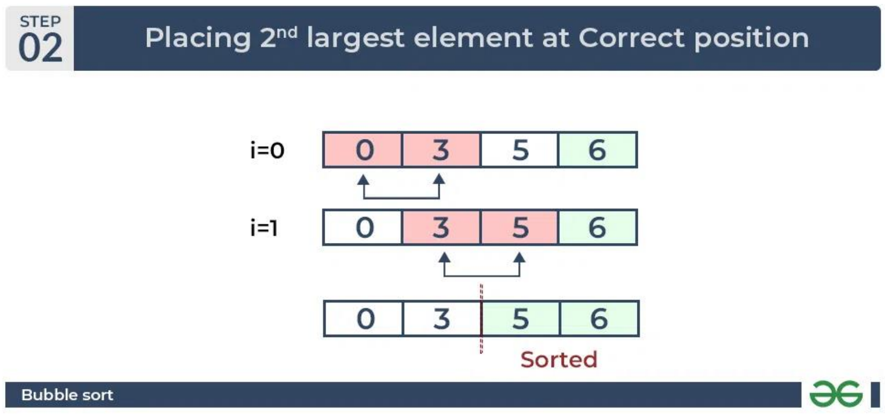
Example:

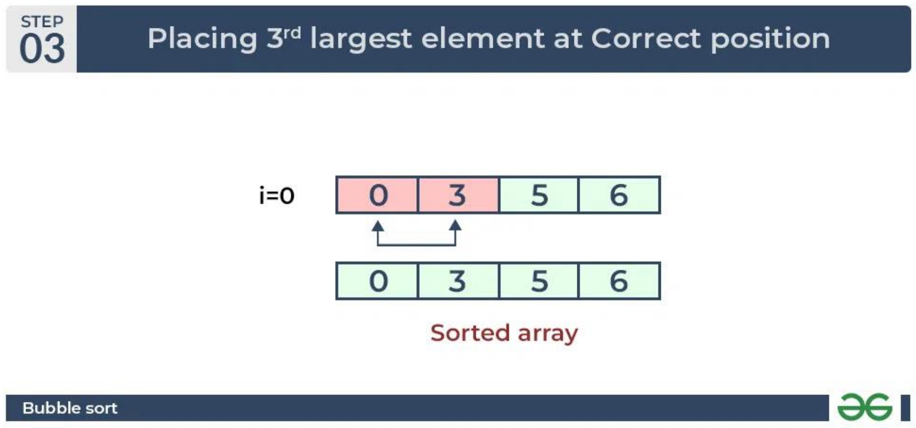
Input: *arr[] = {6, 3, 0, 5}*

First Pass:
The largest element is placed in its correct position, i.e., the end of the array.



Second Pass:
Place the second largest element at correct position





- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2

Algorithm(Bubble Sort)

1. Set I=1 (Set Iteration Count)
2. Repeat steps 3 to 7 while I<N
3. Set J=0 [set array index for comparison]
4. Repeat steps 5 to while J< N -I
5. If A[J] > A[J+1] then Swap A[j] and A[J+1]
6. Set J=J+1
7. Set I=I+1
8. Exit

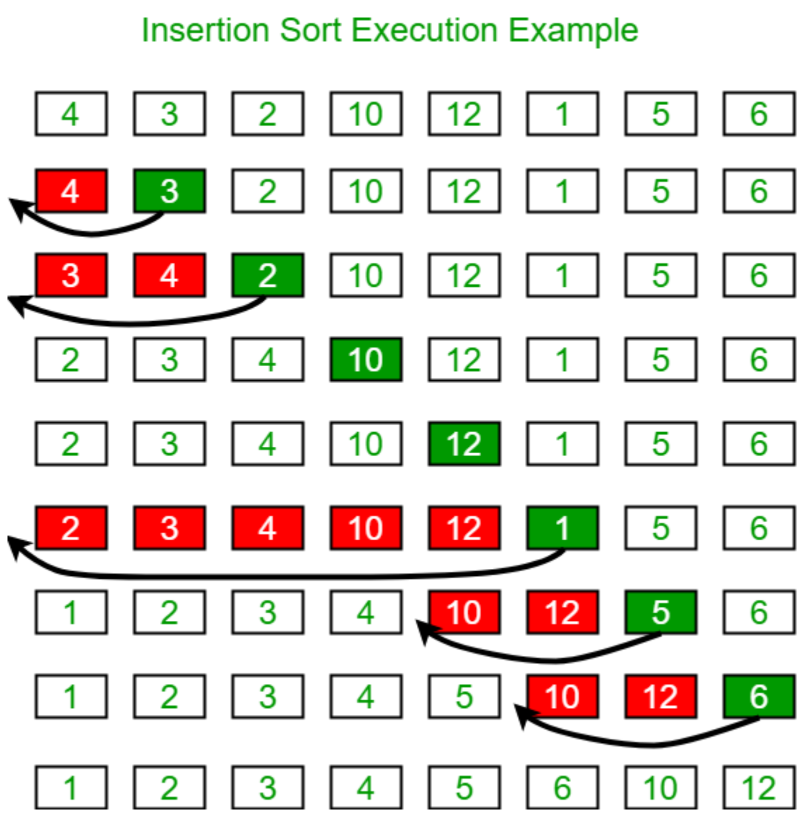
C Program

```
// Optimized implementation of Bubble sort

#include <stdbool.h>

#include <stdio.h>

void swap(int* xp, int* yp)

{

        int temp = *xp;

        *xp = *yp;

        *yp = temp;

}

// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

        int i, j;

        bool swapped;

        for (i = 0; i < n - 1; i++) {

                swapped = false;

                for (j = 0; j < n - i - 1; j++) {

                        if (arr[j] > arr[j + 1]) {

                                swap(&arr[j], &arr[j + 1]);

                                swapped = true;

                        }

                }
        // If no two elements were swapped by inner loop,
        // then break
        if (swapped == false)
            break;
    }
}
// Function to print an array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}
// Driver program to test above functions
int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Insertion Sort

Basic operation – insertion

Builds the sorted array by inserting one element at a time. Insert an element to a sorted array at its correct position.



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.

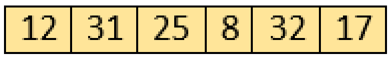
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

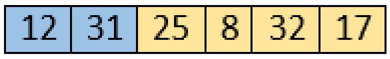
To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

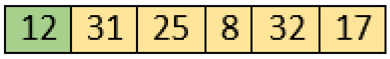


12	31	25	8	32	17

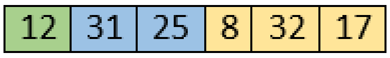
Initially, the first two elements are compared in insertion sort.

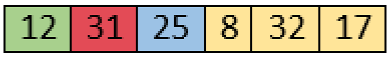


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



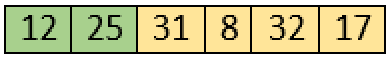
Now, move to the next two elements and compare them.



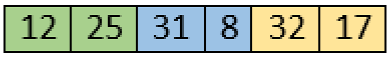


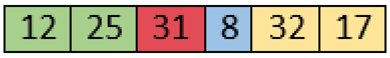
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

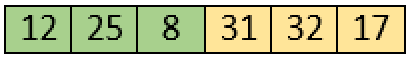


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

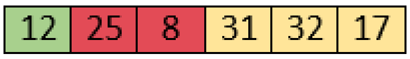




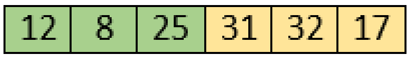
Both 31 and 8 are not sorted. So, swap them.



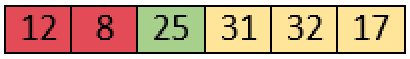
After swapping, elements 25 and 8 are unsorted.



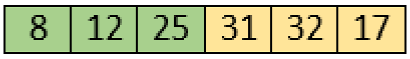
So, swap them.



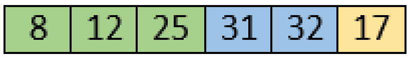
Now, elements 12 and 8 are unsorted.



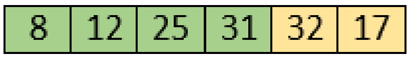
So, swap them too.



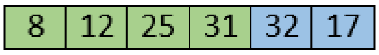
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



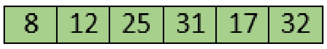
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

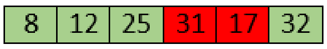


Move to the next elements that are 32 and 17.



17 is smaller than 32. So, swap them.





Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32

8	12	25	17	31	32

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32

Now, the array is completely sorted.

Algorithm(Insertion Sort)

Array A with N elements

```
1.  Set I=1[Iteration Counter]
2.  Repeat steps3 to   9   while I<N
3.  Set Val= A[I]
4.  Set J=I-1
5.  Repeat Steps 6 to  7   while J>=0 and A[J]>Val
6.  Set A[J+1] = A[J]
7.  Set J=J-1
8.  Set A[J+1]=Val
9.  I=I+1
10. Exit
```

C Program(Insertion Sort)

```
#include <stdio.h>
 void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
   int i, j, temp;
   for (i = 1; i < n; i++) {
       temp = a[i];
       j = i - 1;
         while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one positi
on ahead from their current position*/
       {
           a[j+1] = a[j];
           j = j-1;
```

```
            }
            a[j+1] = temp;
        }
    }
     void printArr(int a[], int n) /* function to print the array */
    {
        int i;
        for (i = 0; i < n; i++)
            printf("%d ", a[i]);
    }
     int main()
    {
        int a[] = { 12, 31, 25, 8, 32, 17 };
        int n = sizeof(a) / sizeof(a[0]);
        printf("Before sorting array elements are - \n");
        printArr(a, n);
        insert(a, n);
        printf("\nAfter sorting array elements are - \n");
        printArr(a, n);
          return 0;
    }
```

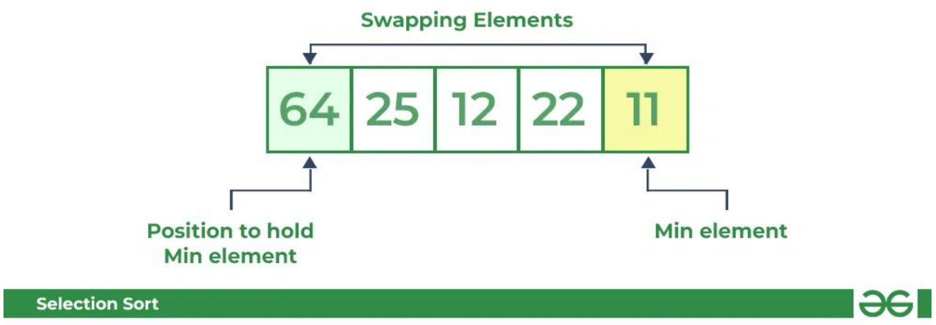
Selection Sort

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 part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Lets consider the following array as an example: ***arr[] = {64, 25, 12, 22, 11}***

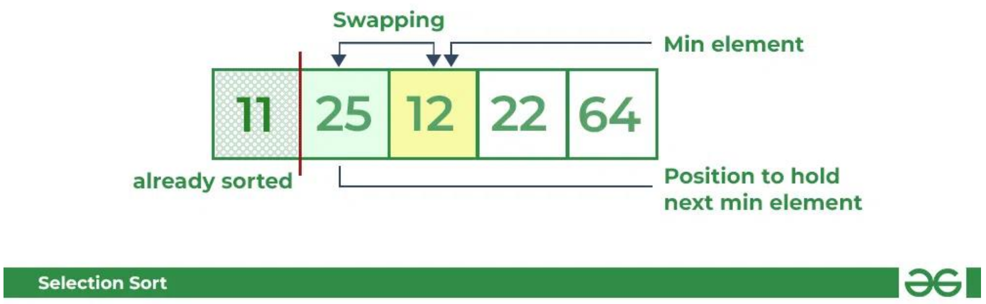
First pass:

- *For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where* ***64*** *is stored presently, after traversing whole array it is clear that* ***11*** *is the lowest value.*
- *Thus, replace 64 with 11. After one iteration* ***11****, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*



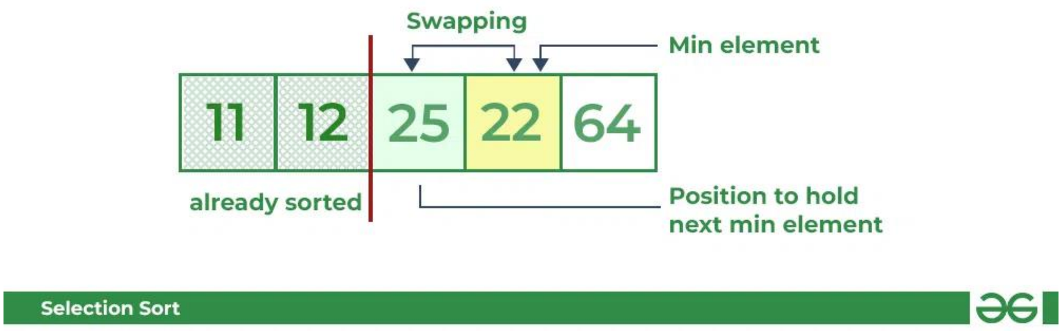
Second Pass:

- *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*
- *After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*



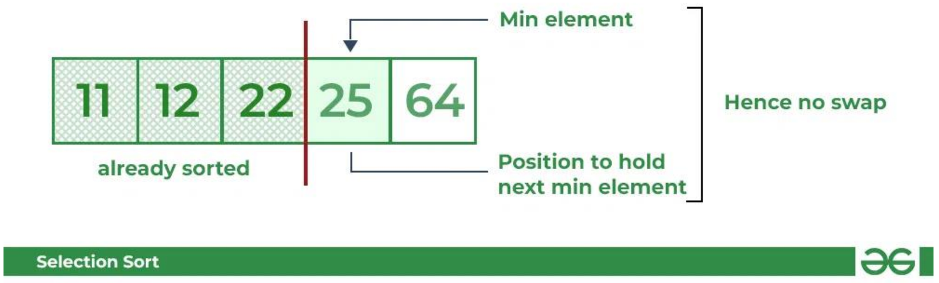
Third Pass:

- *Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.*
- *While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.*



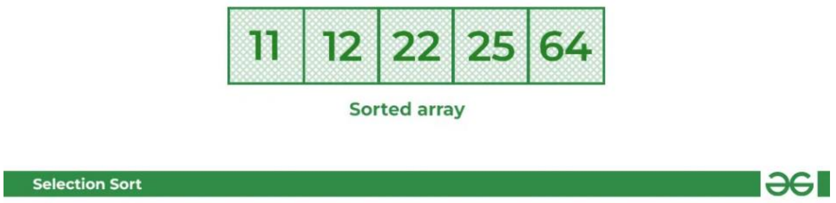
Fourth pass:

- *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
- *As **25** is the 4th lowest value hence, it will place at the fourth position.*



Fifth Pass:

- *At last the largest value present in the array automatically get placed at the last position in the array*
- *The resulted array is the sorted array.*



```
// C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}

void selectionSort(int arr[], int n)
```

```
{
        int i, j, min_idx;

        // One by one move boundary of unsorted subarray
        for (i = 0; i < n-1; i++)
        {
                // Find the minimum element in unsorted array
                min_idx = i;
                for (j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                        min_idx = j;

                // Swap the found minimum element with the first element
                if(min_idx != i)
                        swap(&arr[min_idx], &arr[i]);
        }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
        int i;
        for (i=0; i < size; i++)
                printf("%d ", arr[i]);
        printf("\n");
}
```

```
// Driver program to test above functions
int main()
{
        int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr)/sizeof(arr[0]);
        selectionSort(arr, n);
        printf("Sorted array: \n");
        printArray(arr, n);
        return 0;
}
```

Quick Sort

Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Algorithm:

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
```

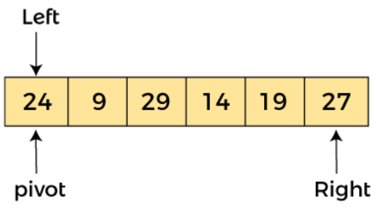
```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ? i + 1
 6 swap A[i] with A[j]
 7  }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

Let the elements of array are –

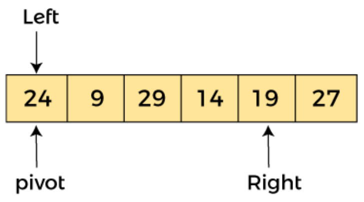
24	9	29	14	19	27

In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

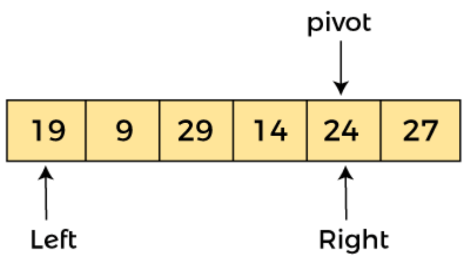


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



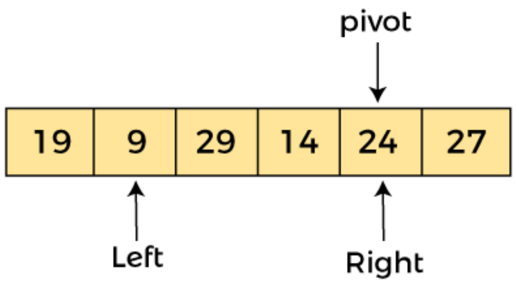
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as

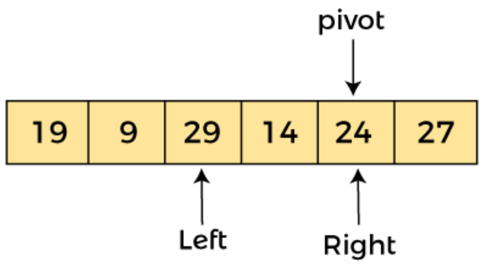


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

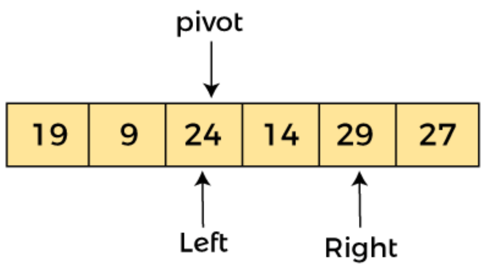
As a[pivot] > a[left], so algorithm moves one position to right as –



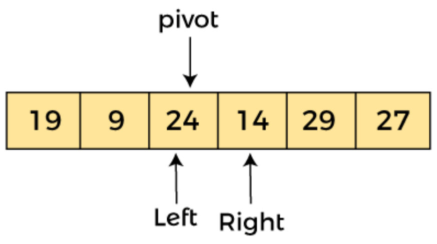
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as –



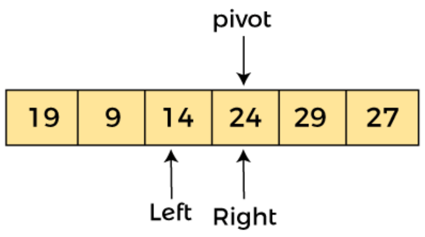
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. –



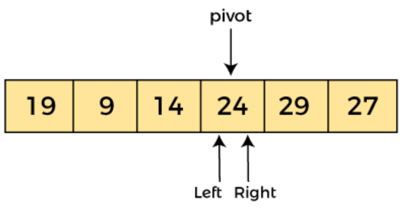
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as –



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. –



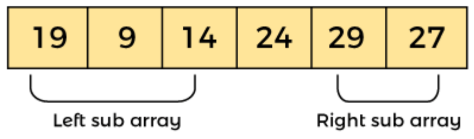
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



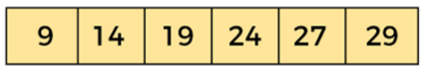
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be –



Program: Write a program to implement quicksort in C language.

```
#include <stdio.h>
/* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot.  */
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}

/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end =
Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}

/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main()
{
    int a[] = { 24, 9, 29, 14, 19, 27 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    quick(a, 0, n - 1);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

Exchange Sort

*Exchange sort is an algorithm used to sort in **ascending** as well as **descending** order. It compares the **first** element with every element if any element seems out of order it **swaps**.*

Example:
Input: arr[] = {5, 1, 4, 2, 8}
Output: {1, 2, 4, 5, 8}
Explanation: Working of exchange sort:

- **1st Pass**:
  Exchange sort starts with the very first elements, comparing with other elements to check which one is greater.
  (**5 1 4 2 8**) –> (**1 5 4 2 8**).
  Here, the algorithm compares the first two elements and swaps since **5 > 1**.
  No swap since none of the elements is smaller than 1 so after 1st iteration (**1 5 4 2 8**)
- **2nd Pass**:
  (**1 5 4 2 8**) –> (**1 4 5 2 8**), since **4 < 5**

(**1 4 5 2 8**) –> (**1 2 5 4 8**), since **2 < 4**
(**1 2 5 4 8**) No change since in this there is no other element smaller than 2

- **3rd Pass:**
  (1 2 5 4 8) -> (**1 2 4 5 8**), since **4 < 5**
  after completion of the iteration, we found array is sorted
- After completing the iteration it will come out of the loop, Therefore array is sorted.

```
• procedure ExchangeSort(num: list of sortable items)
    n = length(A)
•   // outer loop
    for i = 1 to n – 2 do
•   // inner loop
•       for j = i + 1 to n-1 do
•            if num[i] > num[j] do
•                swap(num[i], num[j])
             end if
        end for
     end for
  end procedure
```