

Graphs

Module 5
MCA

Graph

- Array/Linked Lists & Stacks □ Linear Data Structure
- BST & Trees □ Non Linear Hierarchical Data Structure
- Graph is an example of Non Linear Data Structure
- A Graph is a collection of nodes connected through edges

Definition of Graph

- A graph $G=(V,E)$ is a collection of vertices and edges connecting these vertices
- Used to model Paths in a city, social networks, website, internal employee network, etc.
- A vertex or node is one fundamentals unity/entity of which graphs are formed
- An edge is uniquely defined by its 2 endpoints
- Directed Edge – One way connection
- Undirected Edge – Two way connection
- Directed Graph – All directed edges
- Undirected Graph – All undirected edges

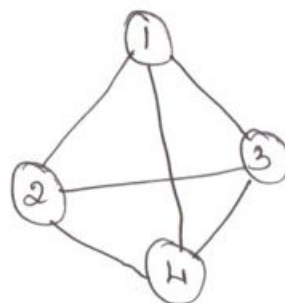
- 
- Indegree = No of edged going out of the node
 - Outdegree = No of edges coming into the node

Represent a graph

- Adjacency list Mark the nodes with the list of its neighbors
- Adjacency matrix $A_{ij} = 1$ for an edge between i and j , 0 otherwise

$$V(G) = \{1, 2, 3, 4\}$$

$$E(G) = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$$



Graph (G)

head nodes

$\{1\} \rightarrow 4 \rightarrow 2 \rightarrow 3$
 $\{2\} \rightarrow 3 \rightarrow 4 \rightarrow 1$
 $\{3\} \rightarrow 2 \rightarrow 4 \rightarrow 1$
 $\{4\} \rightarrow 1 \rightarrow 2 \rightarrow 3$


List (G)

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

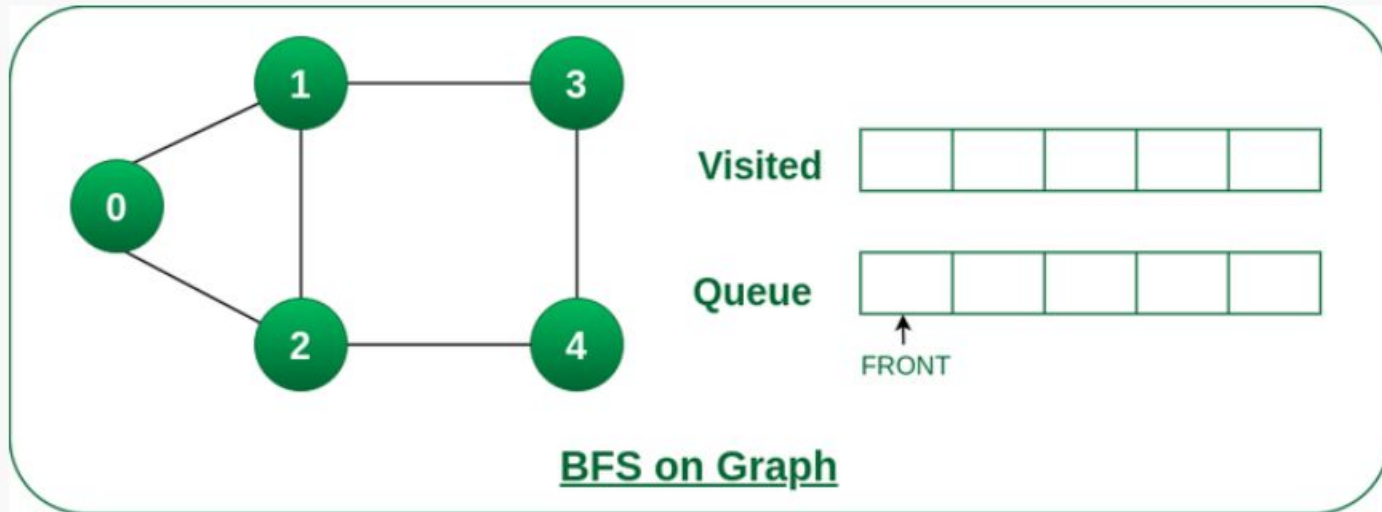
Matrix (G)

Algorithm of Breadth-First Search:

- **Step 1:** Consider the graph you want to navigate.
- **Step 2:** Select any vertex in your graph (say v_1), from which you want to traverse the graph.
- **Step 3:** Utilize the following two data structures for traversing the graph.
 - Visited array(size of the graph)
 - Queue data structure
- **Step 4:** Add the starting vertex to the visited array, and afterward, you add v_1 's adjacent vertices to the queue data structure.
- **Step 5:** Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.
- **Step 6:** Repeat step 5 until the queue is not empty and no vertex is left to be visited.

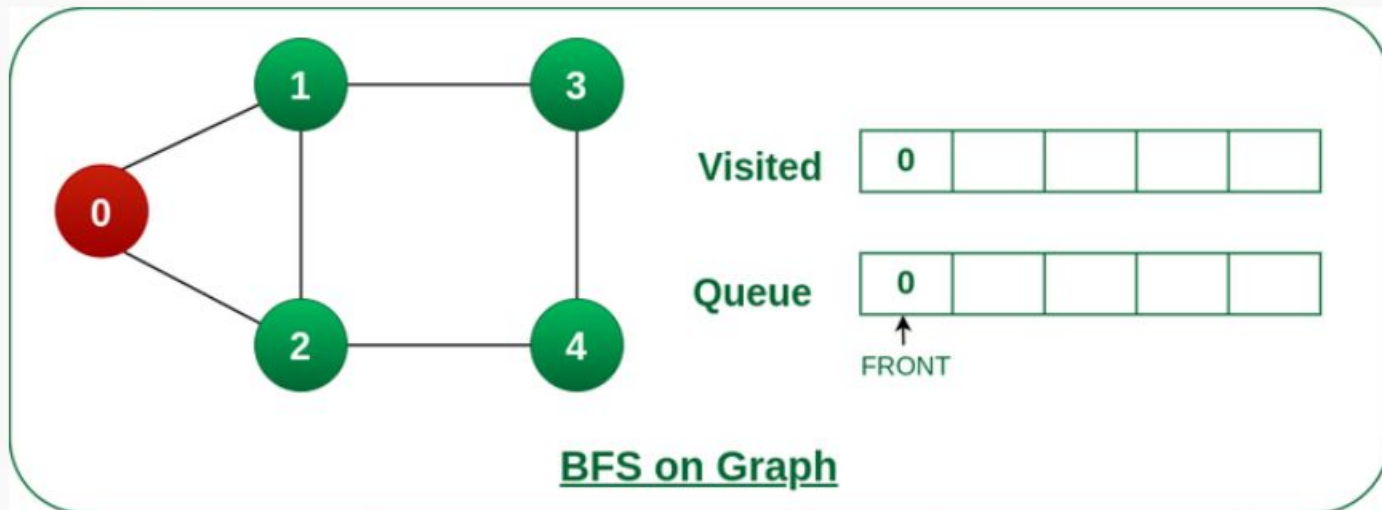
- 
- Declare a queue and insert the starting vertex.
 - Initialize a **visited** array and mark the starting vertex as visited.
 - Follow the below process till the queue becomes empty:
 - Remove the first vertex of the queue.
 - Mark that vertex as visited.
 - Insert all the unvisited neighbors of the vertex into the queue.

Step1: Initially queue and visited arrays are empty.

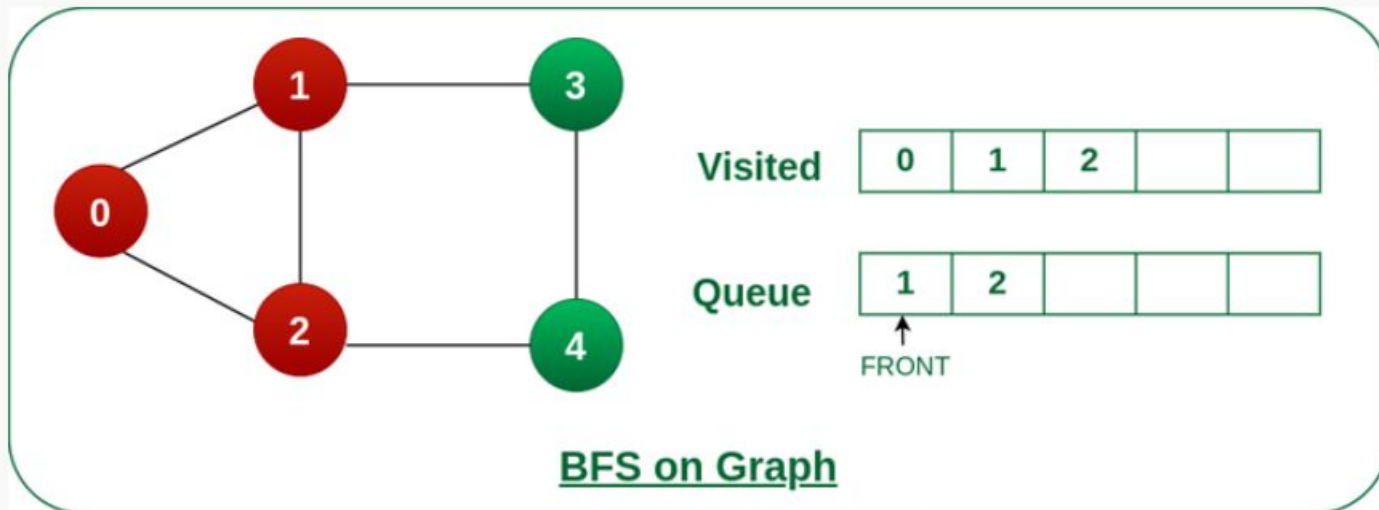


Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.

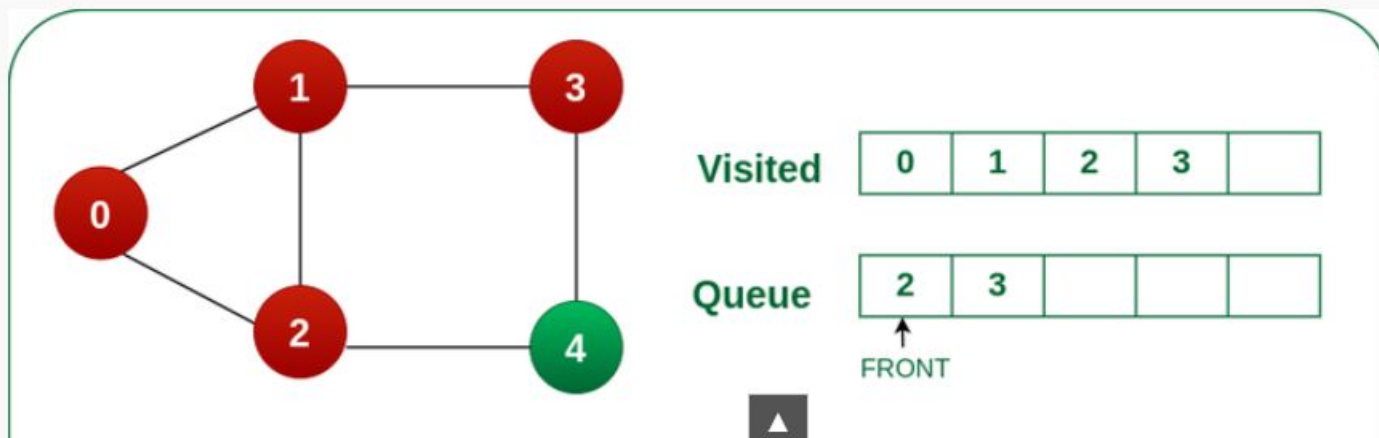


Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.

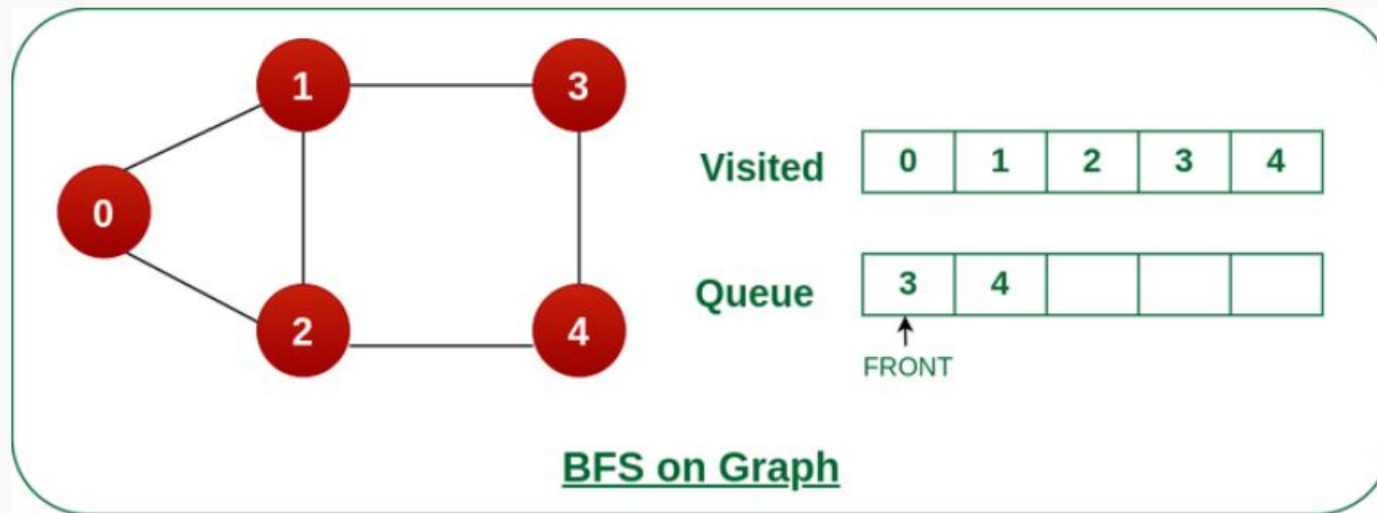


Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



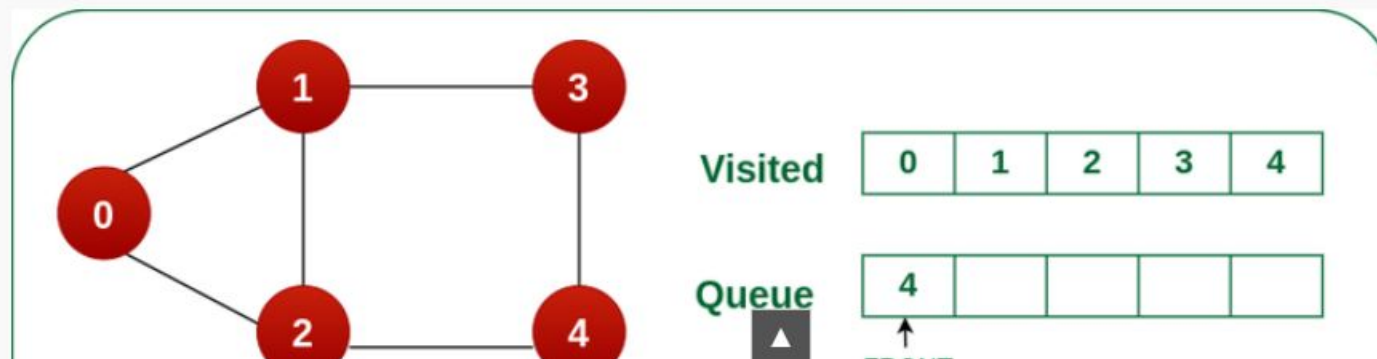
Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

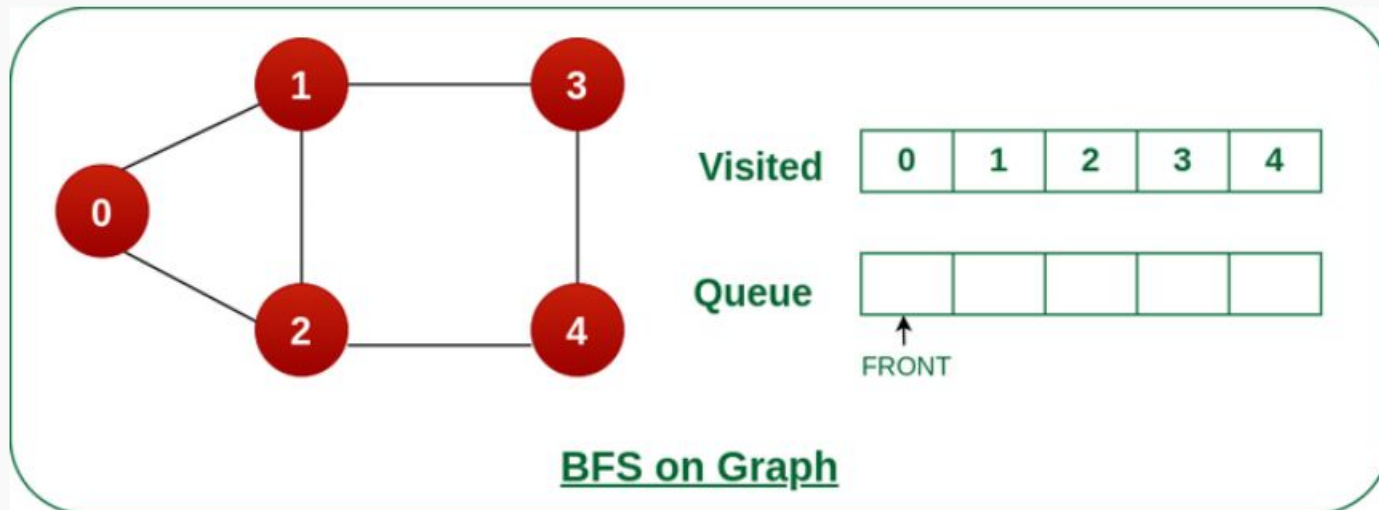
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



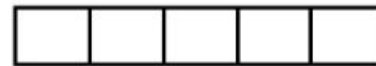
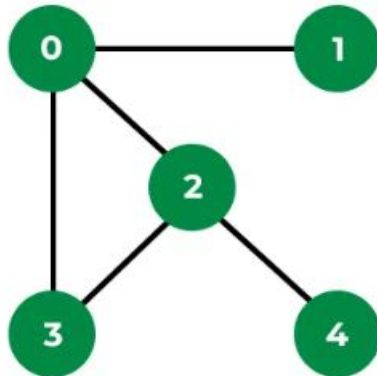
Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

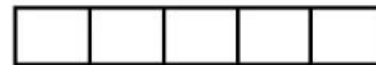
Depth-first search

- Depth-first search is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node.
- Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Step1: Initially queue and visited arrays are empty.

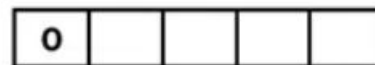
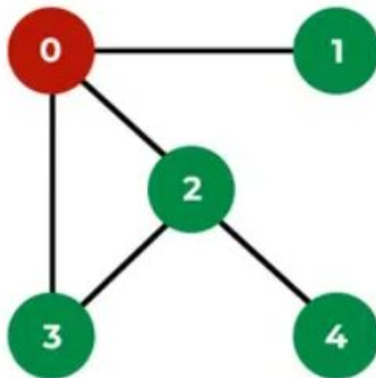


Visited

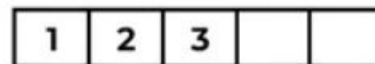


Stack

DFS on Graph



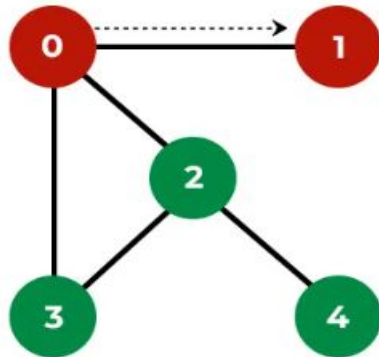
Visited



Stack

DFS on Graph

Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



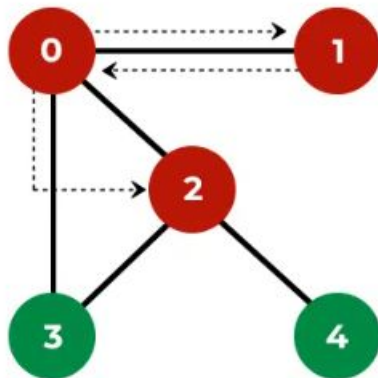
0	1			
---	---	--	--	--

Visited

2	3			
---	---	--	--	--

Stack

DFS on Graph



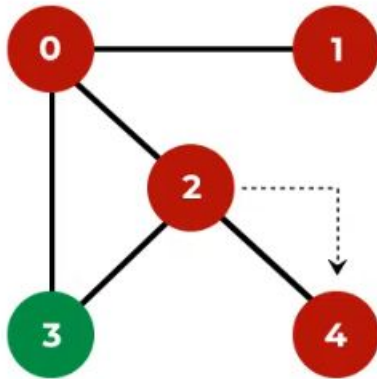
0	1	2		
---	---	---	--	--

Visited

4	3			
---	---	--	--	--

Stack

DFS on Graph



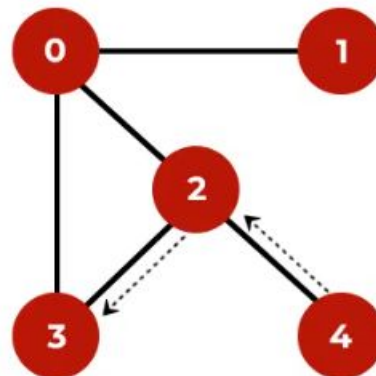
0	1	2	4	
---	---	---	---	--

Visited

3				
---	--	--	--	--

Stack

DFS on Graph



0	1	2	4	3
---	---	---	---	---

Visited

--	--	--	--	--

Stack

DFS on Graph

1.	Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
4.	Technique	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.

5.	Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
6.	Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
7.	Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
8.	Suitability for Decision-Trees	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and then explore all paths through this decision. And if this decision leads to win situation, we stop.

INSERTION SORT


- This sorting technique involves inserting a particular element in proper position.
 - In the first iteration, the second element is compared with the first.
 - In second iteration, the third element is compared with second and then the first
 - The element is compared with all the elements before it.
 - If the element is found to be greater than any of its previous elements, Swap the elements

Consider an example

25 12 30 8 7 43 32

I iteration

25 12 30 8 7 43 32 (12 < 25, so insert 12 at first position)



12 25 30 8 7 43 32

II iteration

12 25 30 8 7 43 32 (30 > 25, so don't compare 30 with 12)



12 25 30 8 7 43 32


III iteration

12 25 30 8 7 43 32 (8 < 30, 25, 12 so insert 8 at 1st position)



IV iteration

8 12 25 30 7 43 32 (7 < 30, 25, 12, 8. So insert 7 at 1st pos)




V iteration

7 8 12 25 30 43 32 (43 > 30. So, don't compare 43 with other)



VI iteration

7 8 12 25 30 43 32 (32 < 43 but 32 > 30. So, insert in-between)



Sorted list:

7 8 12 25 30 32 43

```
for(i=1;i<n;i++)  
{  
    item=a[i];  
    for(j=i-1;j>=0 && item<a[j];j--)  
        a[j+1]=a[j];  
    a[j+1]=item;  
}
```

RADIX SORT

- The radix sort is based on the idea that the number which is having the highest significant digit greater than the corresponding digit of another number will be larger.

- Consider an example for radix sort. We have to sort the numbers-
212, 310, 451, 117, 256, 813, 514, 315, 789, 618, 912, 513

Step (i): Scan each number and put it into the appropriate packets based on last digit (least significant digit) of a number as below

0	1	2	3	4	5	6	7	8	9
310	451	212	813	514	315	256	117	618	789
		912	513						

Now, remove elements from the packets-

310, 451, 212, 912, 813, 513, 514, 315, 256, 117, 618, 789

Step (ii) : Put these elements into packets based on 2nd digit as follows –

0	1	2	3	4	5	6	7	8	9
	310				451			789	
	212				256				
	912								
	813								
	513								
	315								
	117								
	618								

- Step (iii) : Put these elements into packets based on 1st digit as follows –

0	1	2	3	4	5	6	7	8	9
	117	212	313	451	513	618	789	813	912
		256	315		514				

- Remove from the packets

117, 212, 256, 313, 315, 451, 513, 514, 618, 789, 819, 912

This is a sorted list.

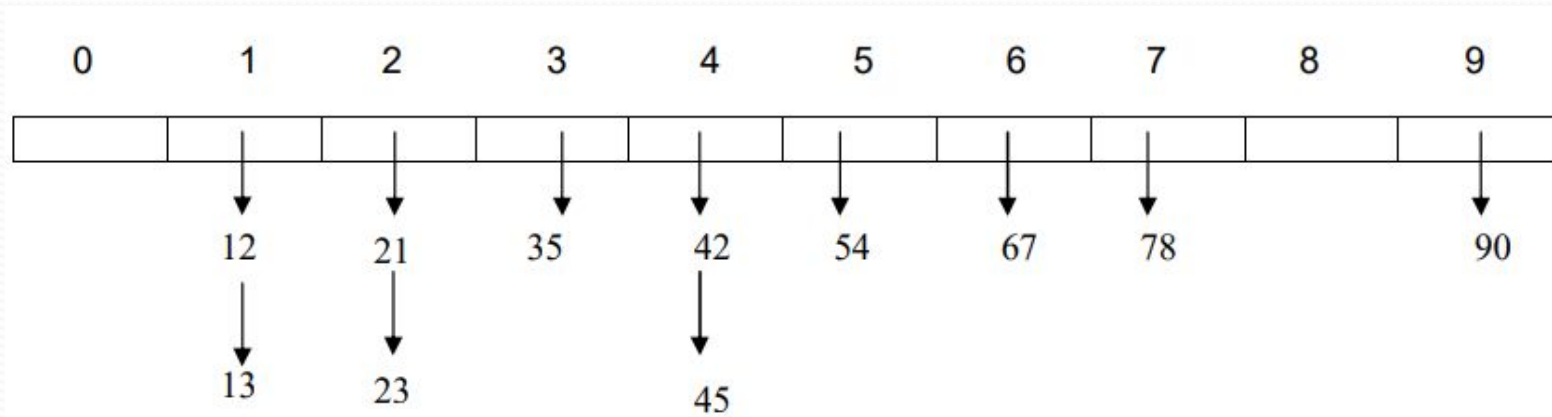

```

void radix_sort(int a[], int n)
{
    int i, j, k, m, big, digit;
    NODE p[10], temp;
    big=largest(a,n);
    m=log 10(big)+1;
    For(j=1;j<=m;j++)
    {
        for(i=0;i<=9;i++)        //0-9 packets
            p[i]=NULL;
        for(i=0;i<n;i++)
        {
            digit=separate (a[i],j);
            p[digit]=insert_rear(a[i], p[digit]);
        }
        k=0;
        for(i=0;i<=9;i++)
        {
            temp=p[i];
            while(temp!=NULL)
            {
                a[k++]=temp->data;
                temp=temp->link;
            }
        }
    }
}

```

return item/(int) pow(10, j-1)%10;

ADDRESS CALCULATION SORT



Cont..

- This algorithm uses an **address table** to store the values which is simply a list (or array) of Linked lists.
- The Hash function is applied to each value in the array to find its corresponding address in the address table
- Then the values are inserted at their corresponding addresses in a sorted manner by comparing them to the values which are already present at that address.

Input : arr = [29, 23, 14, 5, 15, 10, 3, 18, 1]

Output:

After inserting all the values in the address table, the address table looks like this:

ADDRESS 0: 1 --> 3

ADDRESS 1: 5

ADDRESS 2:

ADDRESS 3: 10

ADDRESS 4: 14 --> 15


ADDRESS 5: 18

ADDRESS 6:

ADDRESS 7: 23

ADDRESS 8:

ADDRESS 9: 29

- 
- The time complexity of this algorithm is in the best case. This happens when the values in the array are uniformly distributed within a specific range.
 - Whereas the worst-case time complexity is $O(n^2)$. This happens when most of the values occupy 1 or 2 addresses because then significant work is required to insert each value at its proper place.

HASHING

- Hashing is a way of representing dictionaries
- Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements
- The elements of dictionary can be numeric or characters or most of the times, records.
- For example, student record may contain student id, name, gender, marks etc.
- Every record is usually identified by some key.

- Here we will consider the implementation of a dictionary of n records with keys $k_1, k_2 \dots k_n$.
- Hashing is based on the idea of distributing keys among a one-dimensional array $H[0 \dots m-1]$, called hash table.
- 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.
- A value is computed using a predefined function called hash function
- This function assigns an integer, called hash address
- if the keys k_1, k_2, \dots, k_n are integers, then a hash function can be $h(K) = K \bmod m$

Let us take keys as 65, 78, 22, 30, 47, 89

$$h(k) = k \% 10.$$

$$h(65) = 65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(47) = 47 \% 10 = 7$$

$$h(89) = 89 \% 10 = 9$$

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

Hash Collisions

- Each key will have an address with any value between 0 to $m-1$,
 - it is obvious that more than one key will have same hash address.
 - That is, two or more keys need to be hashed into the same cell of hash table.
 - This situation is called as hash collision.
 - There are two methods for hash collision resolution
 - Open hashing and closed hashing

Open Hashing (or Separate Chaining)

- In open hashing, keys are stored in linked lists attached to cells of a hash table.
- Each list contains all the keys hashed to its cell

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as $h(k) = k \% 10$, then the hash addresses will be – $h(65) =$

$$65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(47) = 47 \% 10 = 7$$

$$h(89) = 89 \% 10 = 9$$

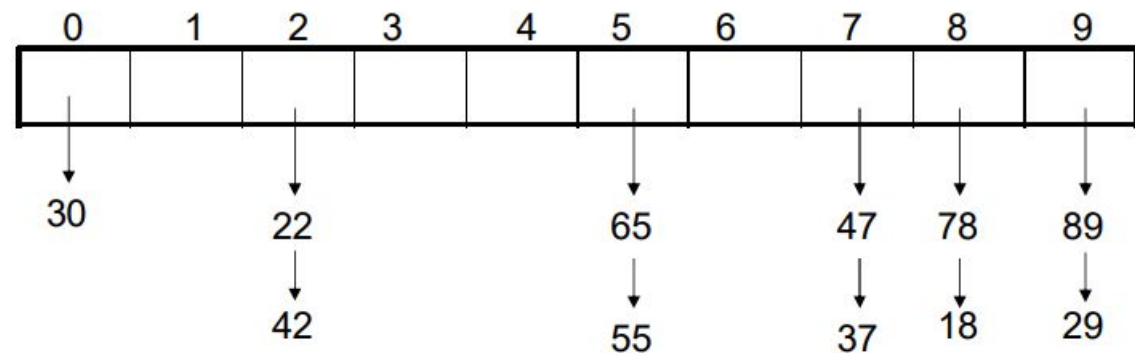
$$h(55) = 55 \% 10 = 5$$

$$h(42) = 42 \% 10 = 2$$

$$h(18) = 18 \% 10 = 8$$

$$h(29) = 29 \% 10 = 9$$

$$h(37) = 37 \% 10 = 7$$



Closed Hashing (or Open Addressing)

- All keys are stored in the hash table itself without using linked lists
- To check the next cell from where the collision occurs. If that cell is empty, the key is hashed there

65, 78, 18, 22, 30, 89, 37, 55, 42

Let us take the hash function as $h(k) = k \% 10$, then the hash addresses will be –

$$h(65) = 65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(18) = 18 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(89) = 89 \% 10 = 9$$

$$h(37) = 37 \% 10 = 7$$

$$h(55) = 55 \% 10 = 5$$

$$h(42) = 42 \% 10 = 2$$

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18