Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation of Graphs, Elementary Graph operations, Traversal methods: Breadth First Search and Depth First Search. **Sorting and Searching:** Insertion Sort, Radix sort, Address Calculation Sort. **Hashing:** Hash Table organizations, Hashing Functions, Static and Dynamic Hashing. **Files and Their Organization:** Data Hierarchy, File Attributes, Text Files and Binary Files, Basic File Operations, File Organizations and Indexing.

Graphs

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

A graph consists of two sets, V and E

V: finite nonempty set of vertices

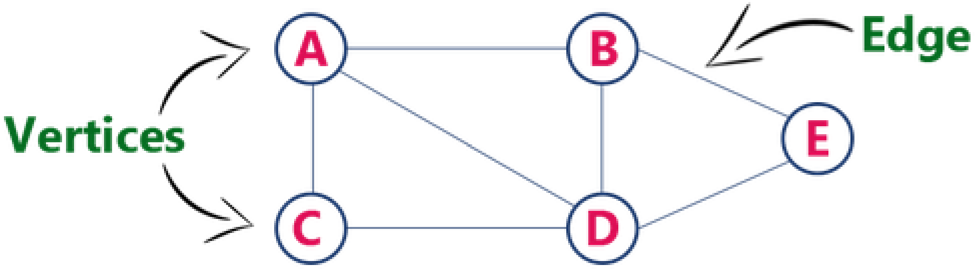
E: set of pairs of vertices

V(G) and E(G): represent set of vertices and edges

or G=(V,E)

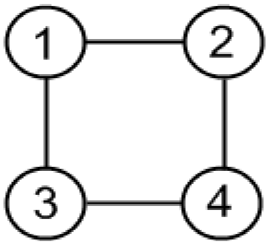
Example: Consider the following graph with 5 vertices and 6 edges, This graph G can be defined as, G=(V,E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

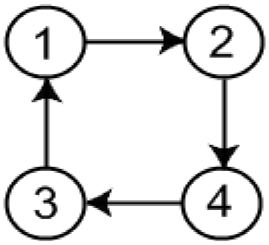


Undirected Graph: pair of vertices representing any edge is unordered, (u,v) & (v,u) represent same edge

Directed Graph: each edge is represented by a directed pair <u,v> u=tail ,v=head of the edge. Edges in directed graphs, called *arcs*, have a direction. (u,v) & (v,u) represent different edge.

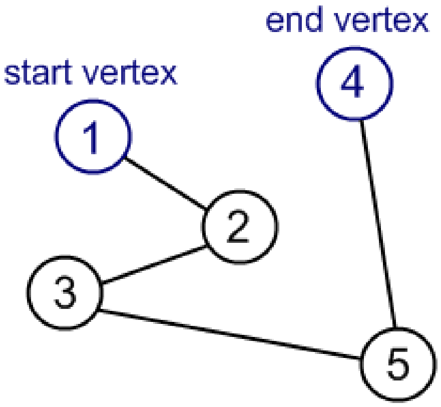


Undirected Graph

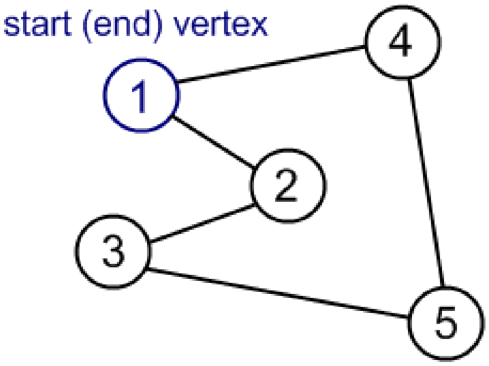


Directed Graph

Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called **path**. First vertex in the path is called the ***start vertex***; the last vertex in the path is called the ***end vertex*.** If start and end vertices are the same, path is called **cycle**. Path is called ***simple***, if it includes every vertex only once. Cycle is called ***simple***, if it includes every vertex, except start (end) one, only once.

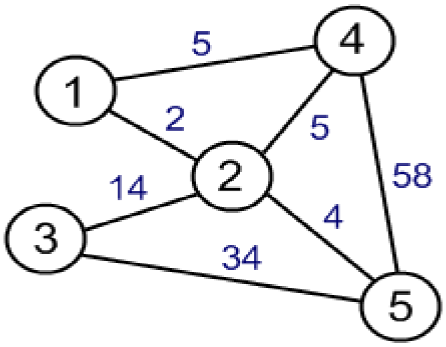


Path (Simple)



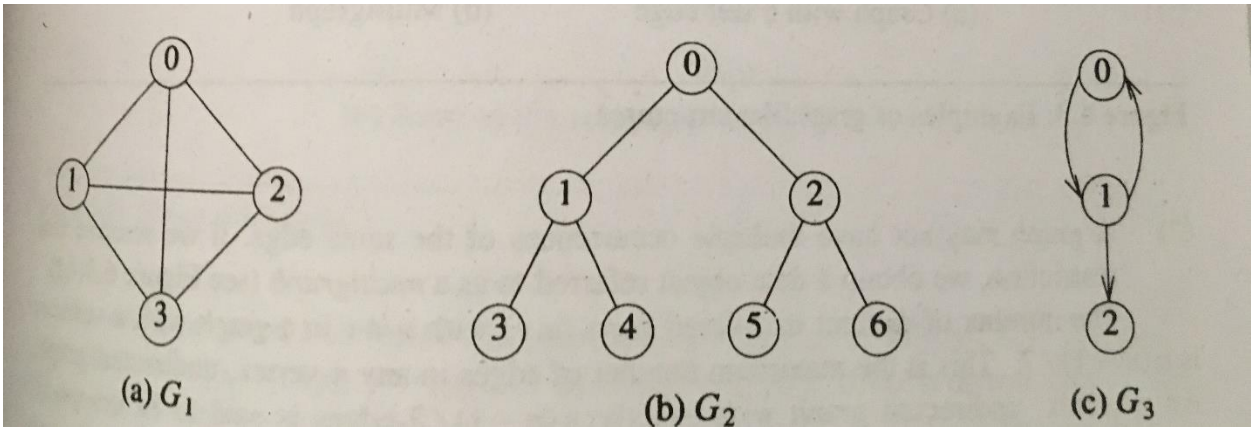
Cycle (Simple)

Graph is called ***weighted***, if every edge is associated with a real number, called edge weight.



Weighted Graphs

Consider the following Graphs,



The set representation of each of these graphs is,

$V(G_1)=\{0,1,2,3\}$; $E(G_1)=\{(0,1),(0,2) ,(0,3),(1,2),(1,3),(2,3)\}$

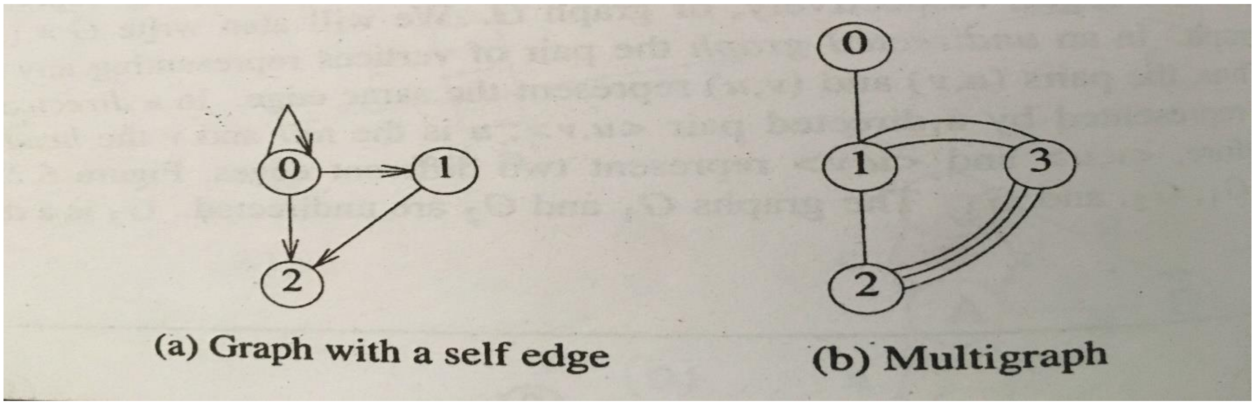
$V(G_2)=\{0,1,2,3,4,5,6\}$; $E(G_1)=\{(0,1),(0,2) ,(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$; $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

Note: The graph G_2 is a tree, the graphs G_1 and G_3 are not.

Restrictions on Graphs:

1) Self-Edges (Self Loops): A graph may not have an edge from a vertex v, back to itself. That is, edges of the form (v,v) & <v,v> are not legal.
2) A graph may not have multiple occurrences of the same edge(else it is multigraph)

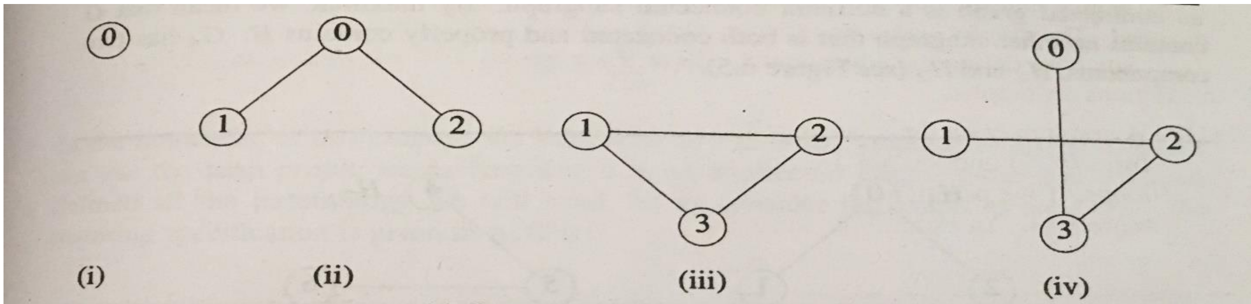


Note:

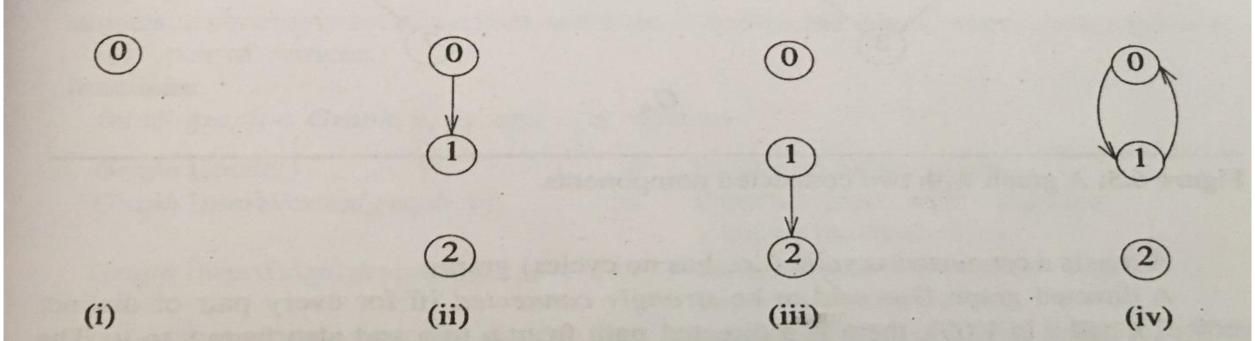
1) Number of distinct unordered pairs (u,v) with u≠v in a graph with n vertices is **n(n-1)/2**

2) G_1 is complete graph but G_2 and G_3 are not complete graphs

A subgraph of G is a graph G` such that V(G`)⊂V(G) and E(G`)⊂E(G). Below figure shows subgraphs of G_1 and G_3.



(a) Some of the subgraphs of G_1



(b) Some of the subgraphs of G_3

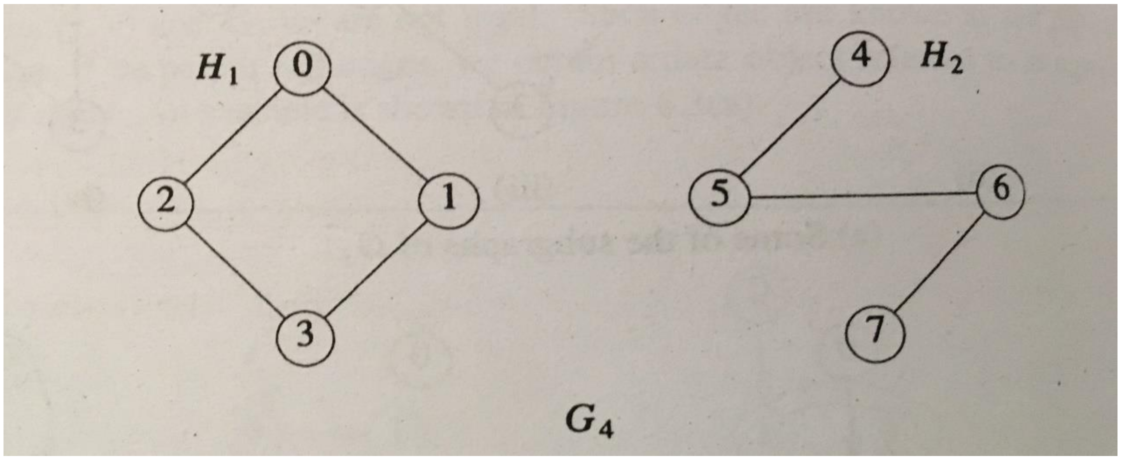
Complete Graph: Let **G** be simple graph on **n** vertices. If the degree of each vertex is **(n-1)** then the graph is called as complete graph. Complete graph on n vertices is denoted by **K_n**. In complete graph **K_n**, the number of edges are **n(n-1)/2.**

Regular Graph: If the degree of each vertex is same say **r** in any graph **G** then the graph is said to be a regular graph of degree **r**.

Bipartite Graph: A graph $G=(V(G),E(G))$ is said to be **Bipartite** if and only if there exists a partition $V(G)=A\cup B$ and $A\cap B=\emptyset$. Hence all edges share a vertex from both set A and B, and there are no edges formed between two vertices in the set A, and there are not edges formed between the two vertices in B.

An undirected graph is said to be **connected** if for every pair of distinct vertices **u** and **v** in **V(G)** there is a path from **u** to **v** in **G**. Graphs **G_1** and **G_2** are connected, whereas **G_4** is not connected.

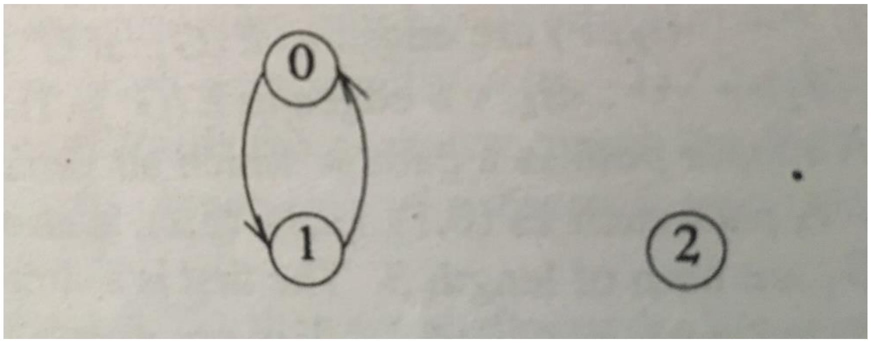
A connected component (or simply a component), H, of an undirected graph is a maximal connected subgraph. (Maximal means that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2.



A tree is a connected acyclic (i.e., has no cycles) graph.

A Directed graph G is said to be **strongly connected** if for every pair of distinct vertices **u** and **v** in **V(G)** ,there is a directed graph from **u** to **v** and also from **v** to **u**. The graph G_3 is not strongly connected, as there is no path from vertex 2 to 1.

A **strongly connected component** is a maximal subgraph that is strongly connected. G_3 has two strongly connected components shown below.



Adjacency Matrix

An adjacency matrix is a **VxV** binary matrix **A**. Element $\mathbf{A_{i,j}}$ is 1 if there is an edge from vertex i to vertex j else $\mathbf{A_{i,j}}$ is 0.

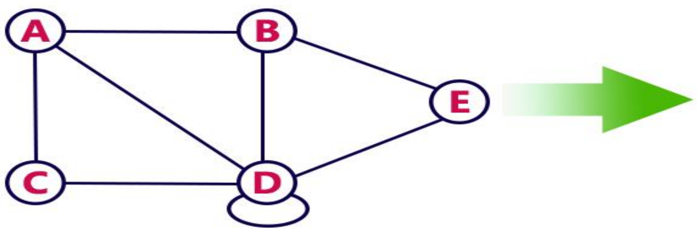
Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $\mathbf{A_{i,j}}$, the weight or cost of the edge will be stored.

In an undirected graph, if $\mathbf{A_{i,j}}$= 1, then $\mathbf{A_{j,i}}$ = 1. In a directed graph, if $\mathbf{A_{i,j}}$ = 1, then $\mathbf{A_{j,i}}$ may or may not be 1.

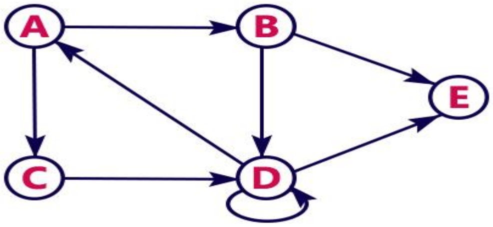
Adjacency matrix provides **constant time access (O(1))** to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is **O(V2)**.

Consider the following undirected graph representation,



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

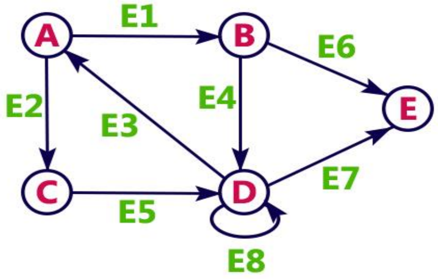
Directed graph representation,



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

Incidence Matrix

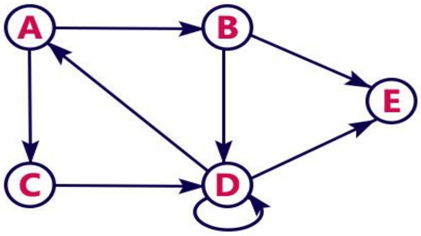
In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex. For example, consider the following directed graph representation,

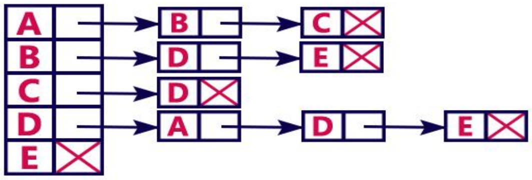


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

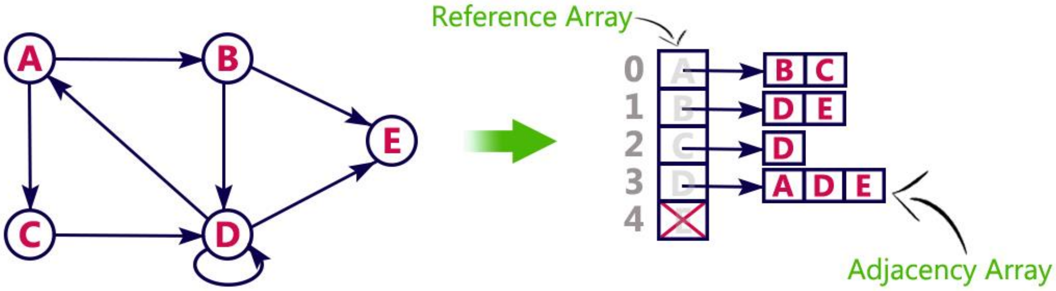
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list,





This representation can also be implemented using array as follows,



Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path. There are two graph traversal techniques,

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph. We use the following steps to implement DFS traversal,

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

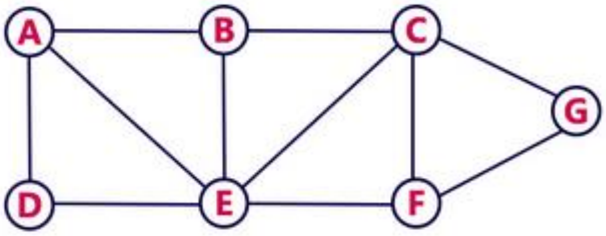
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

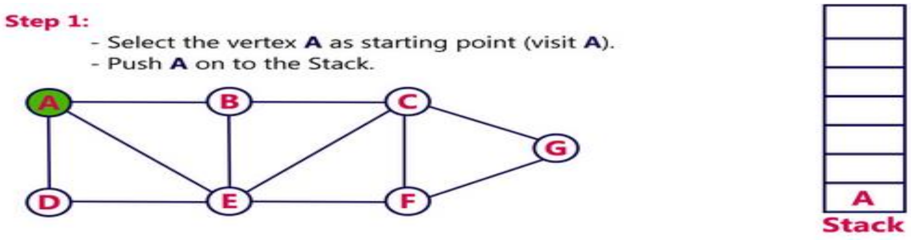
Back tracking is coming back to the vertex from which we came to current vertex.

/* Write a program to check whether a given graph is connected or not using DFS method */

```
void dfs(int source)
{
        int v, top = -1;
        s[++top] = 1;
        b[source] = 1;
        for(v=1; v<=n; v++)
        {
                if(a[source][v] == 1 && b[v] == 0)
                {
                        printf("\n%d -> %d", source, v);    dfs(v);
                }
        }
}
void main( )
{
        int ch;
        printf("\nEnter the source vertex to find the connectivity:");
        scanf("%d",&source);
        m=1;
        dfs(source);
        for(i=1;i<=n;i++)
        {
                if(b[i]==0)
                m=0;
        }
        if(m==1)
                printf("\nGraph is Connected");
          else
                printf("\nGraph is not Connected");
}
```

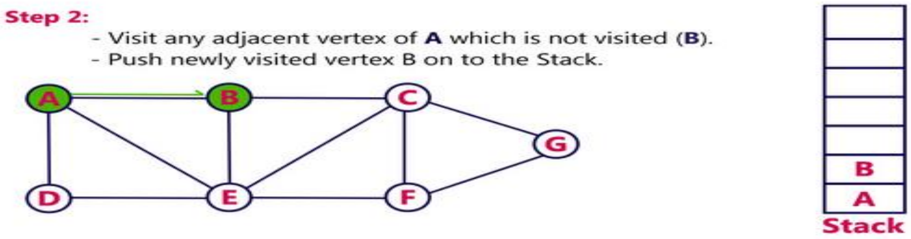
Consider the following graph to perform DFS,





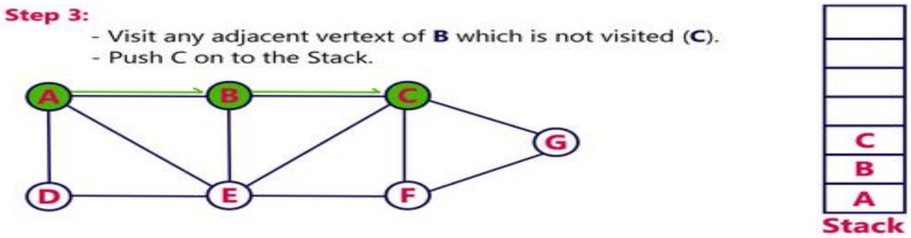
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



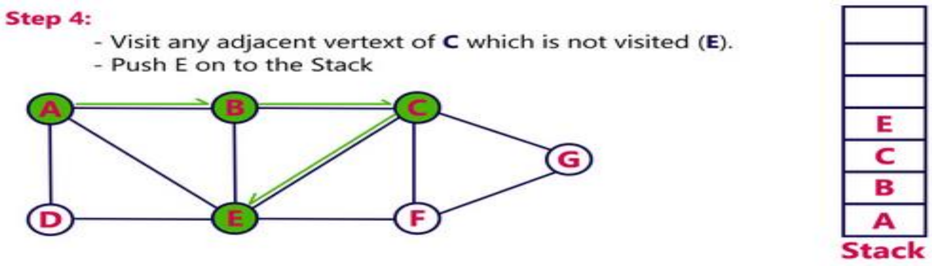
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



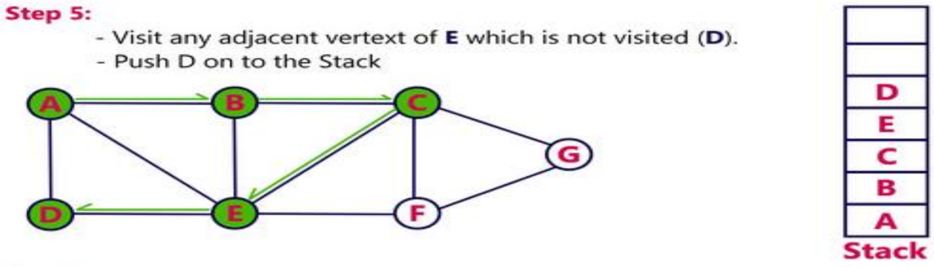
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



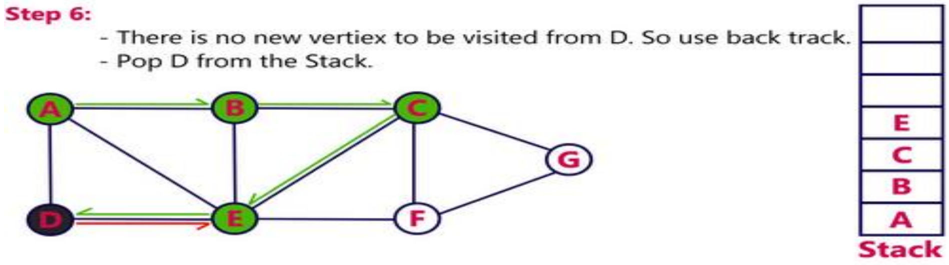
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

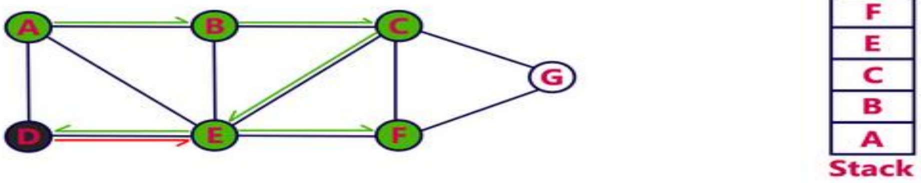


Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

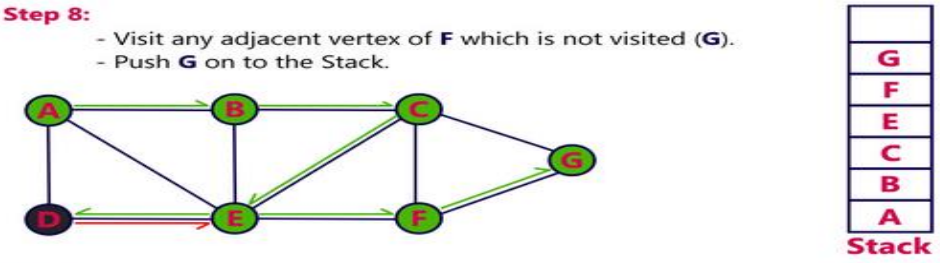
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



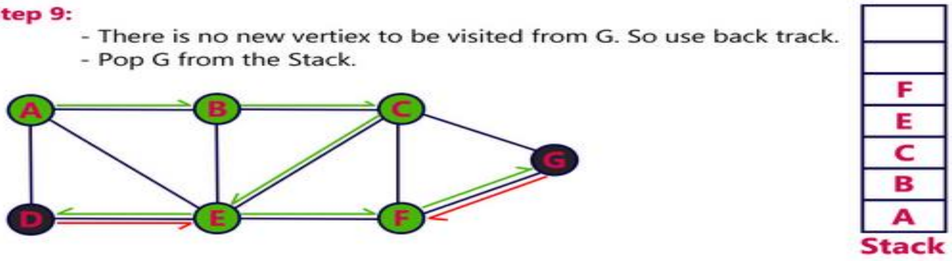
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



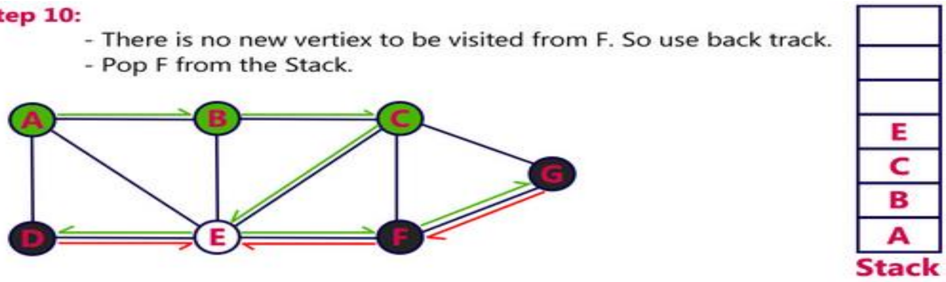
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



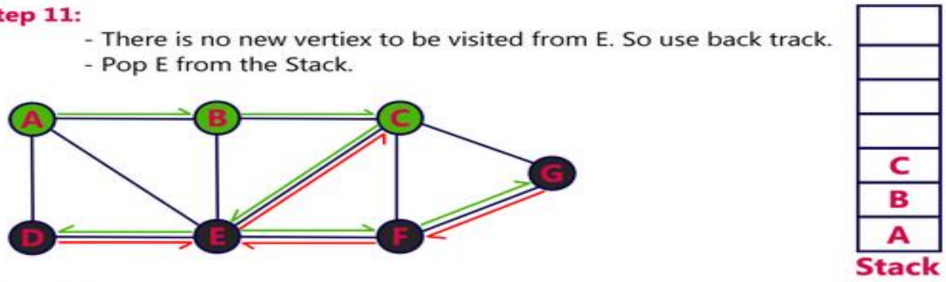
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



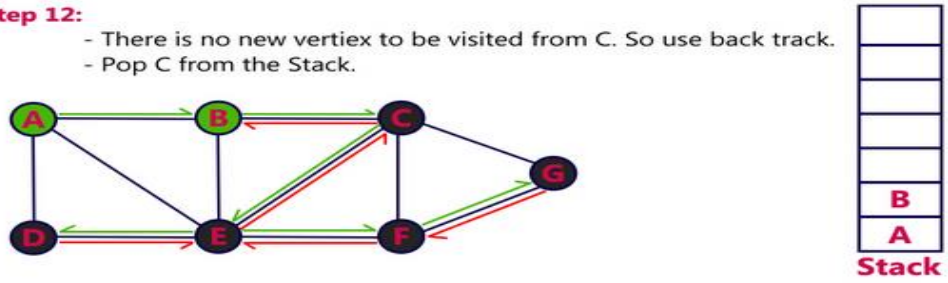
Step 11:

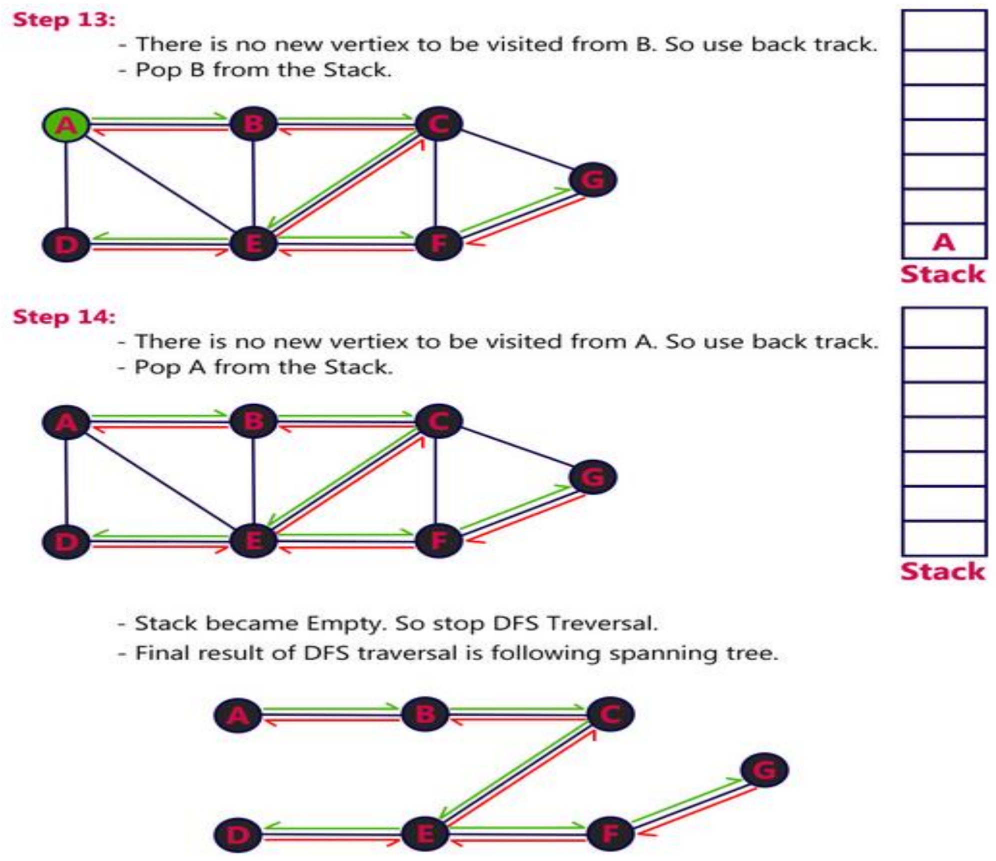
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

BFS (Breadth First Search)

BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph. Steps to implement BFS traversal,

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

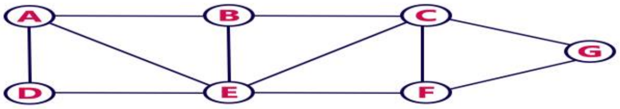
Step 3: Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

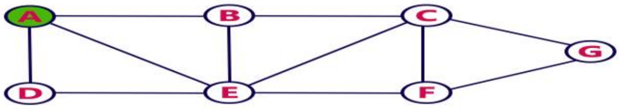
Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

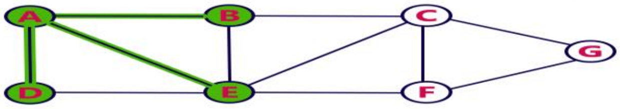


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

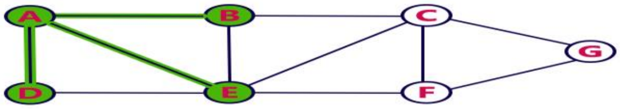


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

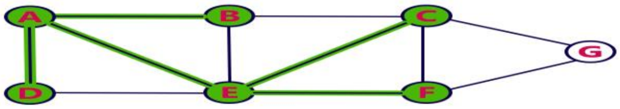


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

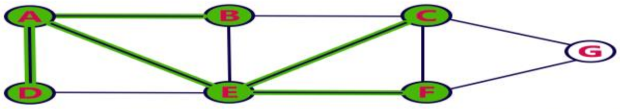


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

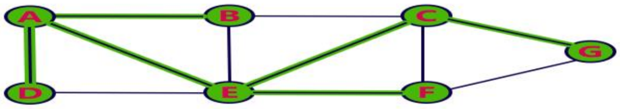


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

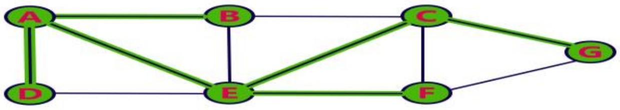


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

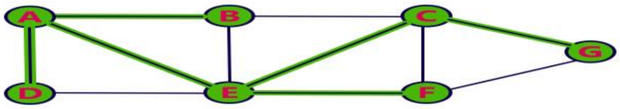


Queue

						G

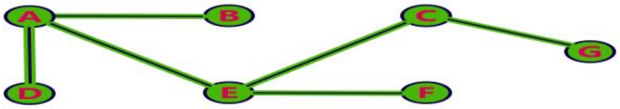
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



/* Program to traverse a graph and print all the nodes reachable from a given starting node in a digraph using BFS method */

```
void bfs( )
{
   int q[10], u, front=0, rear=-1;
   printf("\nEnter the source vertex to find other nodes reachable or not: ");
   scanf("%d", &source);
   q[++rear] = source;
   visited[source] = 1;
   printf("\nThe reachable vertices are: ");
   while(front<=rear)
   {
        u = q[front++];
        for(i=1; i<=n; i++)
        {
             if(a[u][i] == 1 && visited[i] == 0)
             {
                  q[++rear] = i;
                  visited[i] = 1;
                  printf("\n%d", i);
             }
        }
   }
}

void main( )
{
   int ch;
   bfs( );
   for(i=1;i<=n;i++)
   if(visited[i]==0)
   printf("\nThe vertex that is not rechable %d", i);
}
```

Insertion Sort

The basic step in this method is to insert a new record into a sorted sequence of i records in such a way that the resulting sequence of size i + 1 is also ordered.

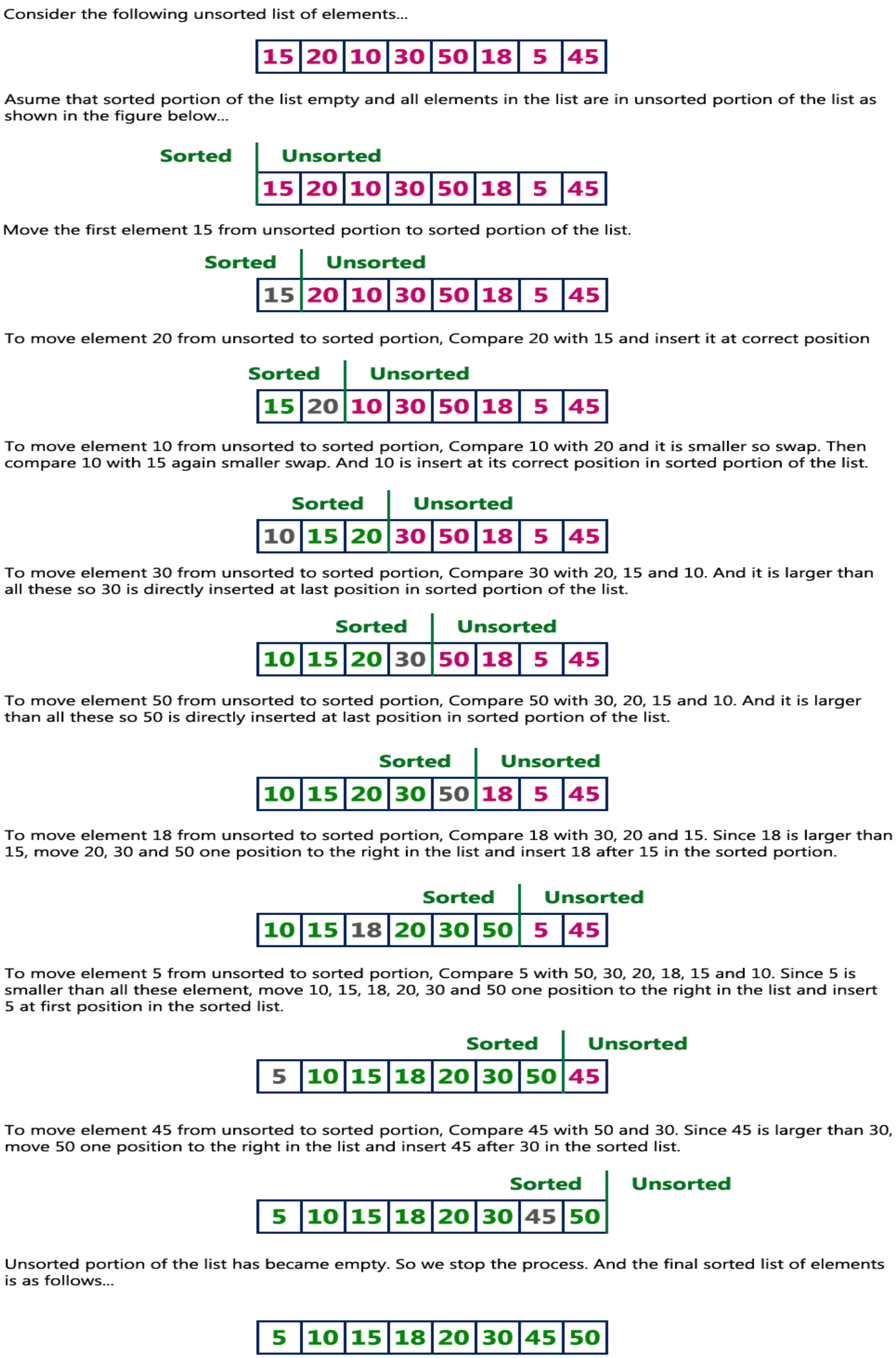
```
/* Insertion into a sorted list */
void insert(int e , int a[ ], int i)
{ /* insert e into ordered list  a[1:i] such that the resulting list a[1:i+1] is also ordered ,the array a
must have space allocated for at least i+2 elements */
    a[0]=e;
    while(e < a[i])
    {
           a[i+1]=a[i];
           i--;
     }
     a[i+1]=e;
}

/* Insertion Sort */
void insertionsort(int a[ ],int n)
{ /* sort a[1:n] into nondecreasing order */
       int j,temp;
       for(j=2;j<=n;j++)
       {
              temp=a[j];
              insert(temp,a,j-1);
       }
}
```

Example 1: Assume that n=5 and the input key sequence is 5, 4, 3, 2, 1.

j	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Example 2: Assume that n=8 and the input key sequence is 15, 20, 10, 30, 50, 18, 5, 45



Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted | Unsorted

15	20	10	30	50	18	5	45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted | Unsorted

10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	45	50

Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50

Radix Sort

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.

- Linear sorting algorithm for integers & uses the concept of sorting names in alphabetical order(radix is 26)
- Radix sort is also known as bucket sort
- After every pass, all the names are collected in order of buckets
  - ➢ In first pass, pick up the names in the first bucket that contains names beginning with A
  - ➢ In second pass, collect names from the second bucket and so on
- When radix sort is used on integers , sorting is done on each of the digits in the number
- Sorting procedure proceeds by sorting the least significant to the most significant
- Number of passes will depend on the length of the number having maximum number of digits

Algorithm: **RadixSort(Arr,N)**

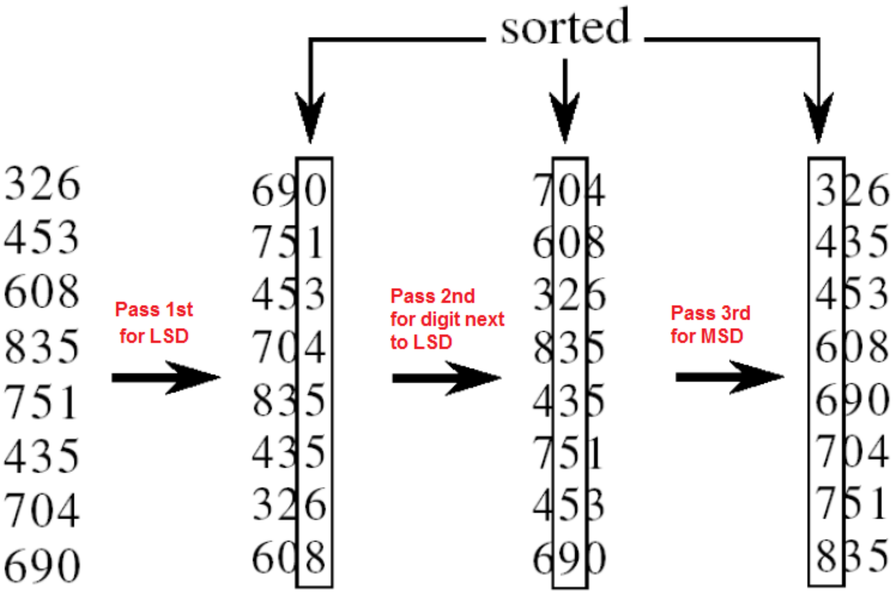
```
Step 1: Find the largest number in Arr as LARGE
Step 2: [INITIALIZE] SET NOP=Number of digits in LARGE
Step 3: SET Pass=0
Step 4: Repeat Step 5 while Pass<=NOP-1
Step 5:         Set I=0 and INITIALIZE buckets
Step 6:         Repeat Step 7 to 9 while I<N-1
Step 7:             Set DIGIT=digit to Pass^th place in A[I]
Step 8:             Add A[I] to the bucket numbered DIGIT
Step 9:             Increment bucket count for bucket numbered DIGIT
                [END OF LOOP]
Step 10:        Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Example 1: Consider how Radix sort operates on seven 3-digits number.

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

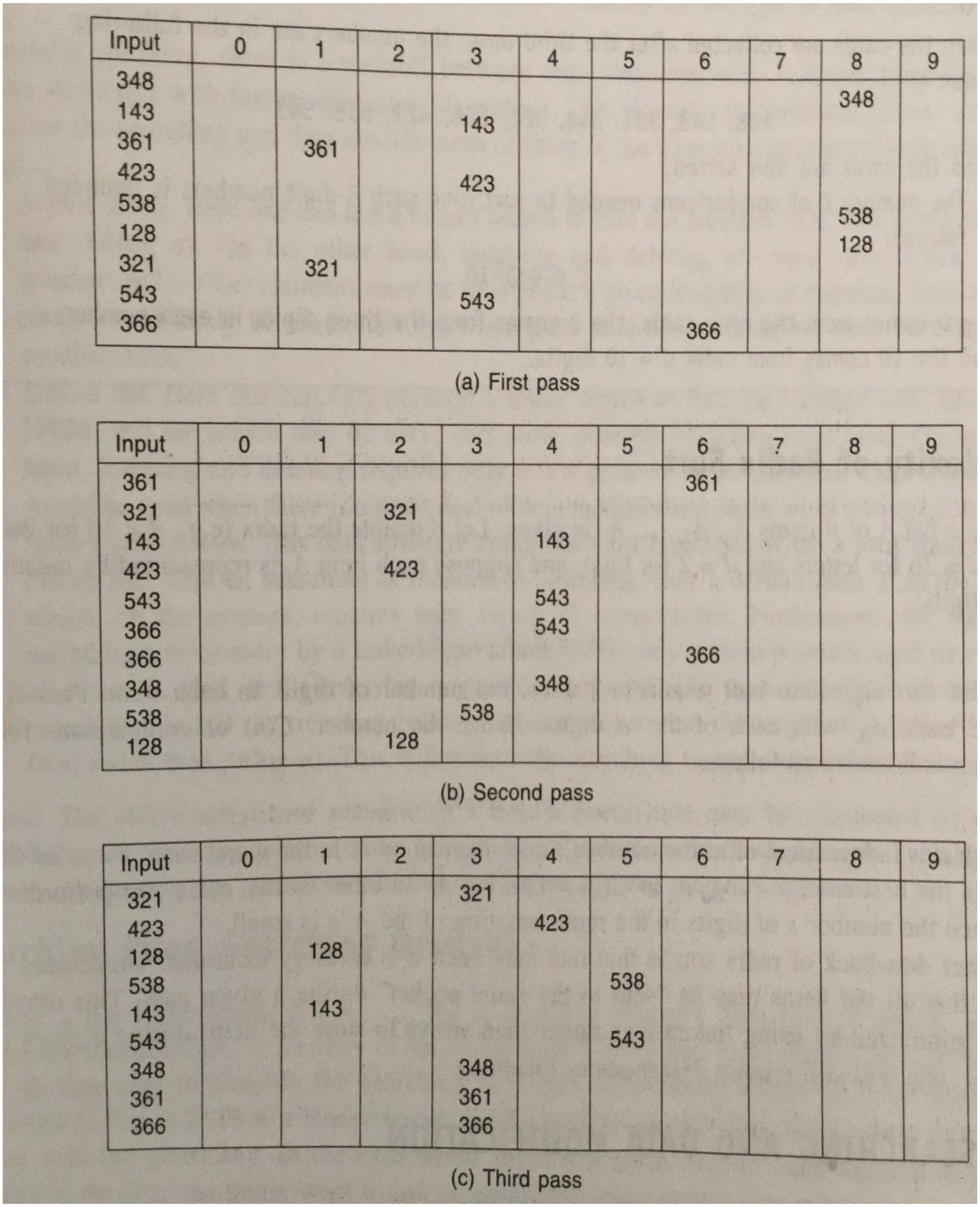
In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n*** elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.

Example 2: Consider a group of numbers. It is given by the list,
326, 453, 608, 835, 751, 435, 704, 690



Example 3: Suppose 9 cards are punched as follows,348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases as follows,



Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass

Algorithm: **Radix Sort**

```
Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A[1..n] is a d-digit integer.
//(Digits are numbered 1 to d from right to left.)
   for j = 1 to d do
      //A[ ]-- Initial Array to Sort
      int count[10] = {0};
      //Store the count of "keys" in count[ ]
      //key- it is number at digit place j
      for i = 0 to n do
       count[key of(A[i]) in pass j]++

      for k = 1 to 10 do
       count[k] = count[k] + count[k-1]

      //Build the resulting array by checking
      //new position of A[i] from count[k]
      for i = n-1 downto 0 do
       result[ count[key of(A[i])] ] = A[j]
       count[key of(A[i])]--

      //Now main array A[] contains sorted numbers
      //according to current digit place
      for i=0 to n do
        A[i] = result[i]

   end for(j)
end func
```

Address Calculation Sort (Hashing)

- In this method a function f is applied to each key.
- The result of this function determines into which of the several subfiles the record is to be placed.
- The function should have the property that: if x <= y , f (x) <= f (y), Such a function is called order preserving.

- An item is placed into a subfile in correct sequence by placing sorting method – simple insertion is often used.

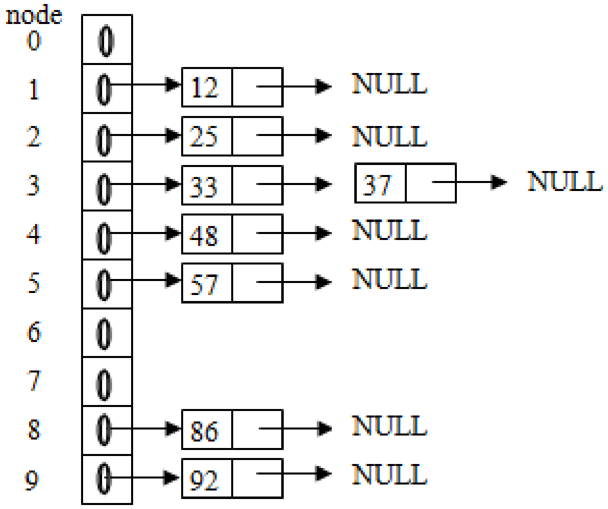
Example: Consider the list of numbers,

25 57 48 37 12 92 86 33

Let us create 10 subfiles. Initially each of these subfiles is empty. An array of pointer f(10) is declared, where f(i) refers to the first element in the file, whose first digit is i. The number is passed to hash function, which returns its last digit (ten's place digit), which is placed at that position only, in the array of pointers.

num = 25	–	f(25) gives 2
57	–	f(57) gives 5
48	–	f(48) gives 4
37	–	f(37) gives 3
12	–	f(12) gives 1
92	–	f(92) gives 9
86	–	f(86) gives 8
33	–	f(33) gives 3 which is repeated.

Thus it is inserted in 3rd subfile (4th) only, but must be checked with the existing elements for its proper position in this subfile.



Hash Tables

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Hash Table is a data structure in which key elements are mapped to array locations based on hash function.

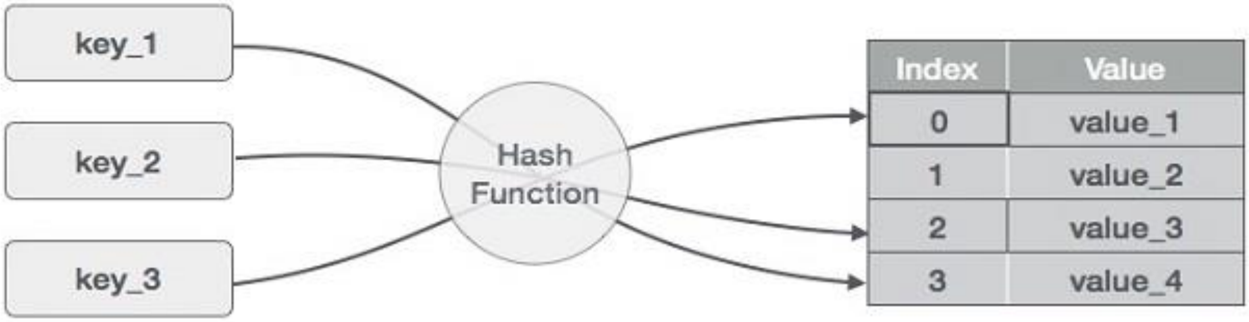
Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hash Function is a mathematical formulae which when applied to key elements produces integer values which serve as array indices in hash table for that respective element.

Hashing is a technique to convert a range of key values into range of indexes of an array

Hash table is a data structure in which keys are mapped to array positions by a hash function

If 2 or more keys map to the same memory location, this is known as collision.



Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

Hashing: It is a searching technique, called hashing or hash addressing which is essentially independent of number **n**. Hash function **H** will be applicable on a key **K** to generate memory address **L**, i.e.

H: K → L

Goal of Hash Function

1. To reduce the range of array indices that have to be handled
2. Reduce the amount of storage space required

Hash functions

Properties of Good hash function

1. Low Cost: cost of executing hash function must be small
2. Determinism and Uniformity(Map Evenly)

Different Hash Functions

1. **Division Method:** Choose a number **m** larger than the number **n** of keys in K. (The number m is usually chosen to be prime number or a number without small divisors, since this frequently minimizes the collision.) The hash function **H** is defined by,

   H(k) = k mod m Or H(k) = (k mod m) + 1

   Here **k mod m** denotes the remainder when **k** is divided by **m**. The second formula is used when we want the hash addresses to range from 1 to m rather than from 0 to m – 1.

   Note: It is best to choose m as prime number (for uniformity)

   Code segment

   ```
   int const m=97 ;
   int h(int x)
   {
           return (x%m);
   }
   ```

   Example: Calculate the hash values of keys 1234 and 5462

   h(1234) = 70

   h(5462) = 30

2. **Multiplication Method**

   Step 1: Choose a constant **A** such that 0<A<1

   Step 2: Multiply the key **k** by **A**

   Step 3: Extract the fractional part of **kA**

   Step 4: Multiply the result of Step 3 by the size of hash table (m)

   H(k) = floor(m(kA mod 1))

Example: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table (Given A=0.618033)

A= 0.618033

A * k i.e. 0.618033*12345=7629.61738

frac(7629.61738) = 0.61738

0.61738 * 1000 = 617.38

floor(617.38) = 617, Hence, H(12345) = 617

3. **Mid-Square Method**

Step 1: Square the value of the key. That is find k^2

Step 2: Extract the middle r digits of the result obtained in step 1

Example: Calculate the hash value for keys 3205, 7148 and 2345 using the mid-square method. Hash table has 100 memory location,

(0-99, 2 digits are needed) so r=2

When k = 3205, k^2 = 102**72**025 H(1234) = 72

When k = 7148, k^2 = 510**93**904 H(5462) = 93

When k = 2345, k^2 = 54**99**025 H(5462) = 99

Note: Same digits must be chosen from all keys

4. **Folding Method**

Step 1: Divide the key value into number of parts (Except last part which may have lesser digits than the other parts)

Step 2: Add the individual parts .Obtain the sum of $k_1+k_2+\ldots k_n$. The hash value is produced by **ignoring the last carry**, if any.

Key	**5678**	**321**	**34567**
Parts	56 and 78	32 and 1	34,56 and 7
Sum	134	33	97
Hash value	34	33	97

For extra “milling”, the even-numbered parts, k_2, k_4,…, are each reversed before addition.

H(5678) = 56 + 87 = 143 so H(5678) = 43 (After ignoring last carry)

H(321) = 32 + 1 = 33 so H(321) = 33

H(34567) = 34 + 65 + 7 = 106 so H(34567) = 06

Collisions

Suppose if we want to add a new record R with key k tour file F, but suppose the memory location address H(k) is already occupied. This situation is called collision. Two records cannot be stored in the same location.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

Collision Resolution Technique

- **Open Addressing / Closed Hashing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.
- **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

Open Addressing (Closed Hashing)

In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

Delete(k): If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”. Insert can insert an item in a deleted slot, but search doesn’t stop at a deleted slot.

Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

In this technique if a value is stored at a location generated by **H(k)** then the following hash function is used to reverse the collision.

$$H(k,i) = [\, H'(k) + i] \bmod m$$

where, **m** is size of the hash table

H'(k) = k mod m

i = probe number varies from 0 to m-1

Let **H(x)** be the slot index computed using hash function and **m** be the table size,

If slot H(x) % m is full, then we try (H(x) + 1) % m

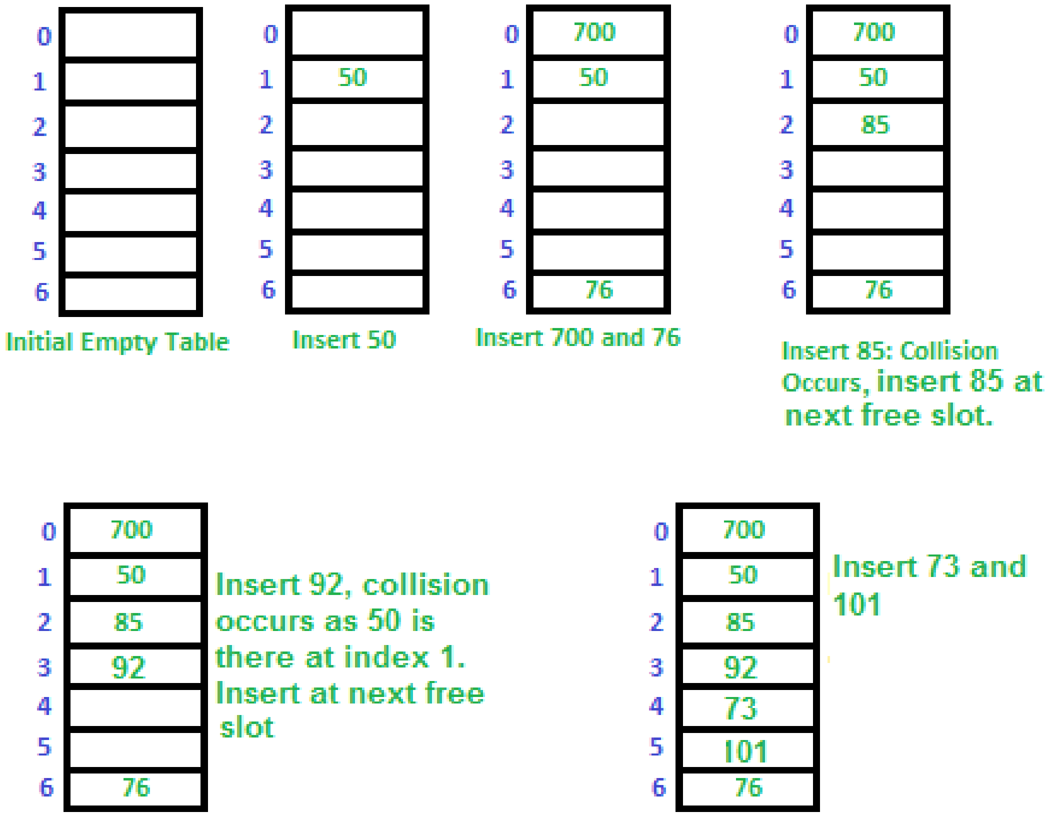
If (H(x) + 1) % m is also full, then we try (H(x) + 2) % m

If (H(x) + 2) % m is also full, then we try (H(x) + 3) % m

....

....

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



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element. The below techniques minimize clustering,

b) Quadratic Probing: Suppose a record R with key k has the hash address H(k) = h. Then, instead of searching the locations with addresses h, h + 1, h + 2, …, we linearly search the locations with addresses h, h + 1, h + 4, h + 9, h + 16, …, h + i^2, …

If the number **m** of locations in the table **T** is a prime number, then the above sequence will access half of the locations in **T**.

Let **H(x)** be the slot index computed using hash function and **m** be the table size,

If slot H(x) % m is full, then we try (H(x) + 1*1) % m

If (H(x) + 1*1) % m is also full, then we try (H(x) + 2*2) % m

If (H(x) + 2*2) % m is also full, then we try (H(x) + 3*3) % m

....

....

c) Double Hashing: Here a second hash function **H**' is used for resolving a collision, as follows. Suppose a record **R** with key **k** has the hash addresses H(k) = h and H' (k) = h' ≠ m.

Then we linearly search the locations with addresses,

h, h + h', h + 2h', h + 3h', …

If **m** is a prime number, then the above sequence will access all the locations in the table **T**.

Let **H(x)** be the slot index using hash function and **m** be the table size.

If slot H(x) % m is full, then we try (H(x) + 1*H'(x)) % m

If (H(x) + 1*H'(x)) % m is also full, then we try (H(x) + 2*H'(x)) % m

If (H(x) + 2*H'(x)) % m is also full, then we try (H(x) + 3*H'(x)) % m

....

....

Note: The disadvantage of open addressing procedure is the implementation of deletion. Suppose a record **R** is deleted from the location **T[r]**. Afterwards suppose if we get **T[r]** while searching for another record **R'**. This does not mean that the search is unsuccessful. Thus, when deleting the record **R**, we must label the location T[r] to indicate that it previously did contain a record. Accordingly, open addressing may seldom be used when a file F is constantly changing.

Collision Resolution by Chaining

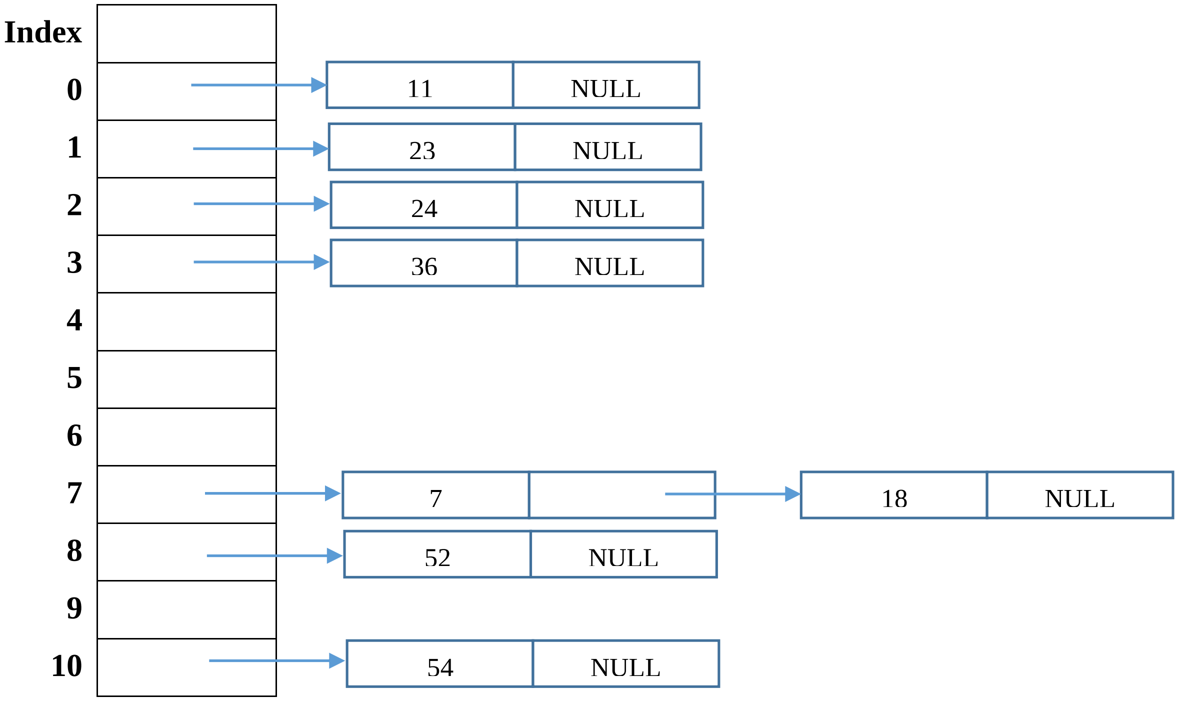
- Each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location
- The key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location

Chaining involves maintaining two tables in memory. Table **T** in memory contains the records in **F**, except that **T** now has an additional field **LINK** which is used so that all records in **T** with the same hash address **h** may be linked together to form a linked list in **T**.

Suppose a new record **R** with key **k** is added to the file **F**. We place **R** in the first available location in the table **T** and then add **R** to the linked list with pointer **LIST[H(k)]**. If the linked list of records are not sorted, then **R** is simply inserted at the beginning of its linked list. Searching for a record or deleting a record is nothing more than searching for a node or deleting a node from a linked list.

Example 1: Insert the keys 7, 24, 18, 52, 36, 54, 11 and 23 in a chained hash table of 11 memory locations, Use H(k) = k mod m

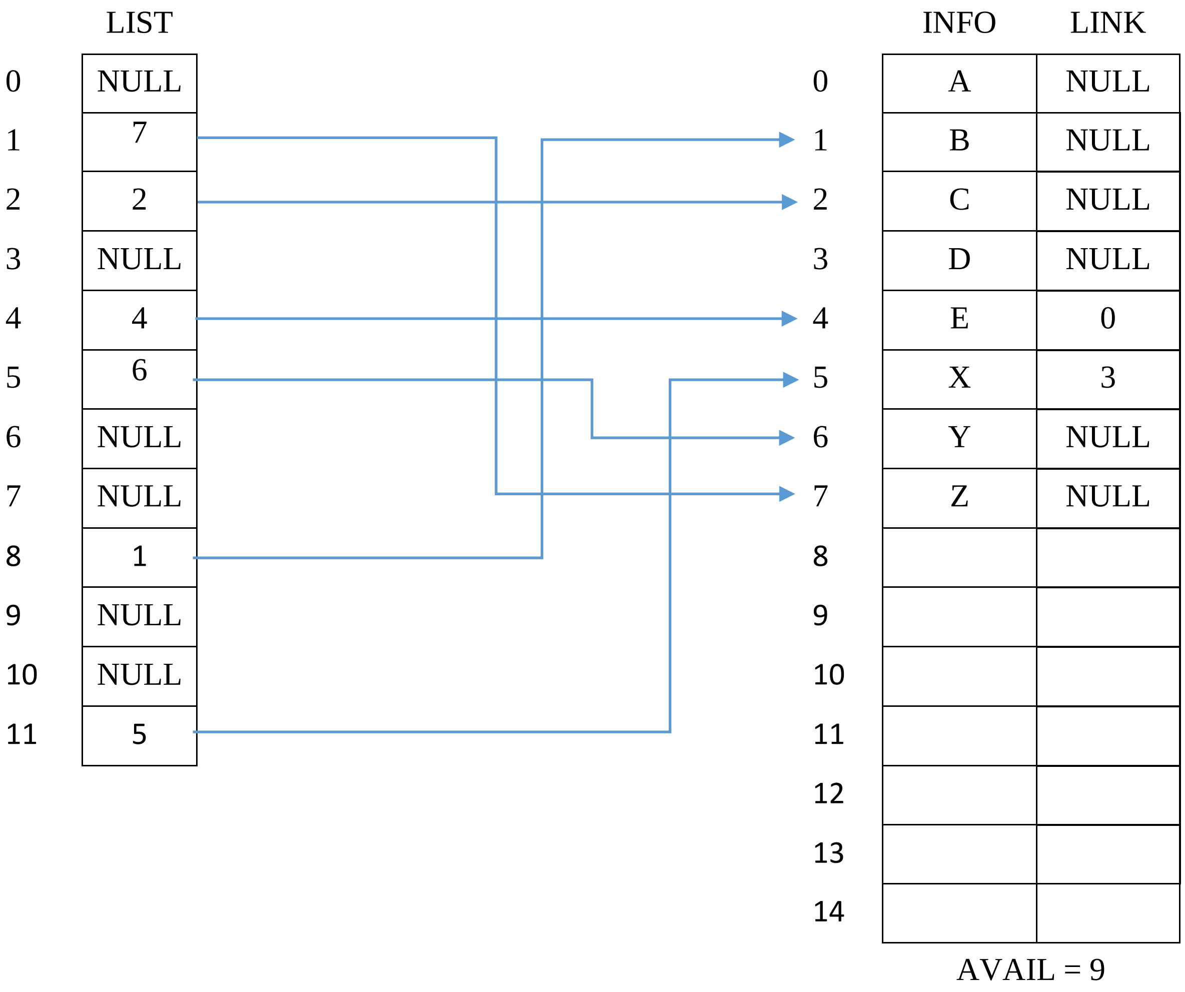
Keys	7	24	18	52	36	54	11	23
H(K)	7	2	7	8	3	10	0	1



Example 2: Consider the below data, where the 8 records have the following hash addresses:

Keys	A	B	C	D	E	X	Y	Z
H(K)	4	8	2	11	4	11	5	1

Using chaining, the records will appear in memory as shown below, the location of a record R in table T is not related to its hash address. A record is simply put in the first node in the AVAIL list of table T. In fact, table T need not have the same number of elements as the hash address table.



	LIST
0	NULL
1	7
2	2
3	NULL
4	4
5	6
6	NULL
7	NULL
8	1
9	NULL
10	NULL
11	5

	INFO	LINK
0	A	NULL
1	B	NULL
2	C	NULL
3	D	NULL
4	E	0
5	X	3
6	Y	NULL
7	Z	NULL
8		
9		
10		
11		
12		
13		
14		

AVAIL = 9

Note: The disadvantage of chaining is that one needs **3m** memory cells for the data. If there are **m** cells for the information field INFO, there are **m** cells for the link field LINK, and there are **m** cells for the pointer array LIST. Suppose each record requires only 1 word for its information field. Hence open addressing is preferable with table of **3m** locations.

Dynamic Hashing (Extendible hashing)

- To ensure good performance, it is necessary to increase the size of a hash table whenever, its loading density exceeds a pre-specified threshold
- We use array doubling to increase the number of buckets , at same time hash function divisor changes

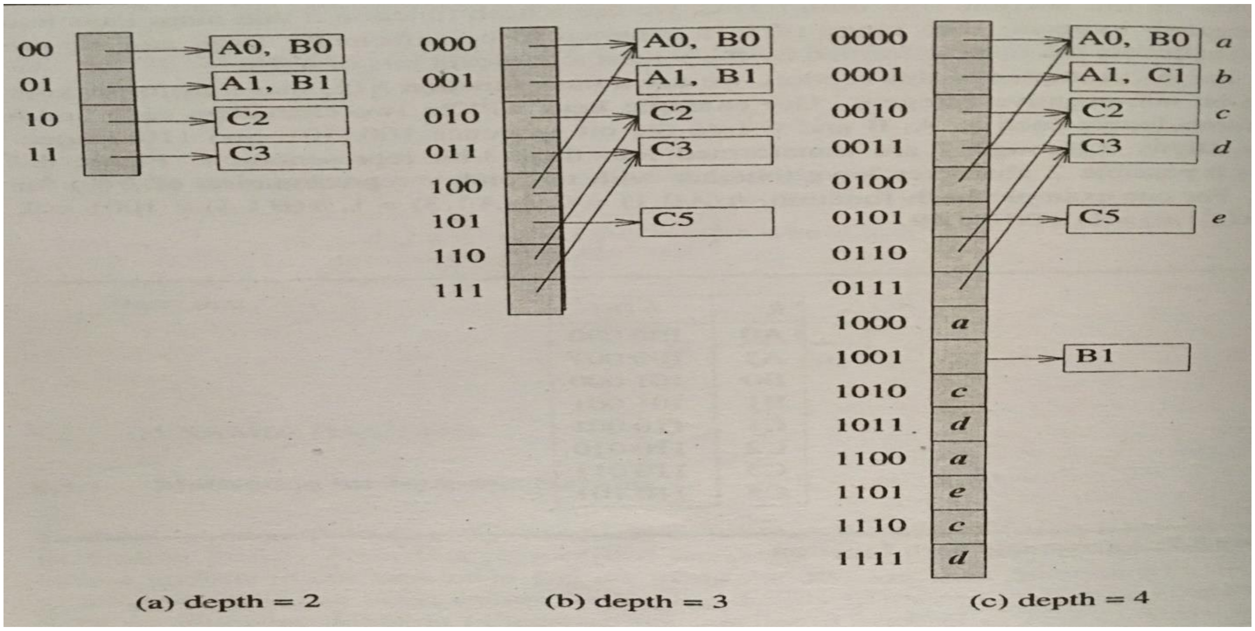
Two forms

1. **Using Directories**
2. **Directory Less**

- For both forms, we use a hash function h that maps keys into non negative integers
- h: Assume, sufficiently large
- **h(k , p):** integer formed by the p least significant bits of **h(k)**
- Hash function **h(k)** transforms keys into 6-bit non-negative integers

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Dynamic Hashing Using Directories



(a) depth = 2 (b) depth = 3 (c) depth = 4

- The size of the directory depends on the number of bits of **h(k)** used to index into the directory
- The number of bits of **h(k)** used to index the directory is called directory depth
- The size of the directory is $\mathbf{2^t}$, where t is the directory depth and the number of buckets is at most equal to the directory size

Suppose we insert, C5, H(C5,2)=01. This gets us to the bucket with A1 and B1 (Bucket full and overflow)

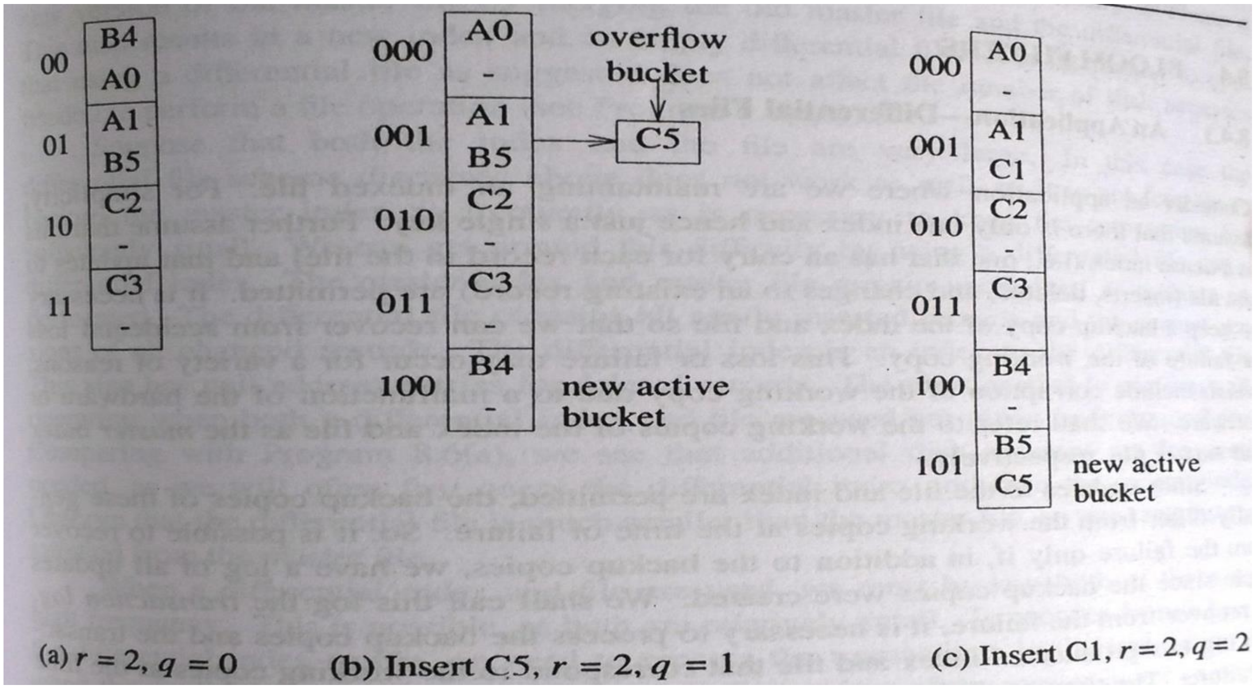
Note: If least **u** is greater than the directory depth, we increase the directory depth to this least u value

Directory-Less Dynamic Hashing

- Also known as linear dynamic hashing
- An array, **ht**, of buckets is used
- We assume that this array is as large as possible and so there is no possibility of increasing its size dynamically
- To avoid initializing such a large array ,we use two variables **q** and **r**, $\mathbf{0 \leq q < 2^r}$,to keep track of active buckets
- At any time ,only **0** through $\mathbf{2^r + q - 1}$ are active

r: number of bits of **h(k)** used to index into hash table

q: bucket that will split next



(a) $r = 2, q = 0$ (b) Insert C5, $r = 2, q = 1$ (c) Insert C1, $r = 2, q = 2$

- Buckets **0** through **q-1** as well as buckets $\mathbf{2^r}$ through $\mathbf{2^r + q - 1}$ are indexed using **h(k, r + 1)** while the remaining buckets are indexed using **h(k, r)**.

Searching a Directory-Less Hash Table

if (h(k,r) < q) search the chain that begins at bucket h(k,r+1);

else search the chain that begins at bucket h(k,r);

- An overflow is handled by activating bucket $\mathbf{2^r + q}$; reallocating the entries in the chain **q** between **q** and the newly activated bucket (or chain) $\mathbf{2^r + q}$, and incrementing **q** by **1**. In case q now becomes $\mathbf{2^r}$, we increment **r** by **1** and reset **q** to **0**. The reallocation is done using **h(k,r+1)**.

Files

A file is a collection of data stored on a secondary memory device such as hard disks. Every file has a file name and its extension.

Example: student name.txt

Here student name is a file name and txt is file extension. Extension identifies the type of file.

A file is a place on the disk where a group of related data is stored.

Files can be classified into ASCII text files and binary files.

ASCII text files are stream of characters which are processed sequentially.

Binary files are those which contain data encoded in binary form.

Advantages of using Files

1. Value can be stored and used whenever input is needed.
2. Large data sets can be stored and used.
3. Files can be used for reading and storing data.

Defining and Opening a File

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system such as,

a) Filename

b) Data Structure

c) Purpose

Filename

Filename is a string of characters that make up a valid filename for the operating system. Filename includes two parts namely a primary name, an optional period with the extension.

Example1: input.txt

Example 2: program1.c

Data Structure

Data structure of a file is defined as FILE in the library of standard I/O function definitions. FILE is a defined datatype.

When a file has to be opened its mandatory to specify the operation to be performed on the file.

Syntax for Declaring and Opening a File:

```
FILE *fp;

fp=fopen(“filename”,”mode”);
```

The first statement declares the variable fp as a pointer to the data type FILE. A file pointer is used to address the file from within a c program. A file pointer has data type FILE*.

The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp, it also specifies the purpose of opening this file through mode. Various modes of file operation are as follows,

Mode	Description
r	Opens an existing file for reading only
w	Opens a file for writing, if file doesn’t exist then a new file is created
a	Opens a file for appending (writing at the end of existing file) and creates the file if it doesn’t exist
r+	Opens a file for reading and writing
w+	Opens a file for reading and writing and creates a file in case if the file doesn’t exists
a+	Opens a file for reading and appending. Creates a file if doesn’t exist. Reading will start from beginning but adding content from end of file.

Closing a File

fclose( ) library function is used for closing the file.

Syntax:

```
int fclose(FILE *fp);
```

fp is the file pointer pointing to file which has to be closed. fclose( ) closes an opened file and flushes all the buffers that are maintained for that file.

fclose( ) returns zero on successful closing of the file and a non-zero value is returned if an error has occurred.

Input and Output Operations of File

fscanf( )

Syntax:

```
int fscanf(FILE *stream,”control string”,argument list);
```

Example: fscanf(fp,”%s%d”,name,&age);

fscanf( ) is used to read formatted data from input file stream. The file must be opened in read mode to scan its contents.

Example:

```
#include <stdio.h>
main( )
{
char str[25];
FILE *fp;
fp=fopen(“sample.txt”,“r”);
fscanf(fp,“%s”,str);
fclose(fp);
printf(“contens of file is%s”,str);
}
```

fprintf( )

Syntax:

```
int fprintf(FILE *stream,”control string”,variable
```

Example: fprintf(fp,”%s%d”,name,age);

fprintf( ) is used to write formatted output to file stream. The file must be opened in write mode to add contents to it.

Example:

```
#include<stdio.h>
main( )
{
char str[10]=“HELLO”;
FILE*fp;
fp=fopen(“example.txt”,“w”);
fprintf(“fp,“%s”,str);
fclose(fp);
}
```

fgetc( )

getc( ) function is used to read a character from file. The function returns the character read in integer form. File must be opened in read mode

Syntax:

```
int getc (FILE *stream);
```

Example:

```
#include<stdio.h>
main( )
{
  FILE *fp;
  fp=fopen(“example.txt”,”r”);
  int ch=getc(fp);
  printf(“%c”,ch);
  ch=getc(fp);
  fclose(fp);
  }
```

fputc( )

fputc( ) is used to write a character into the file, the file must be opened in write mode to add contents to it.

Syntax:

```
int fputc(character,FILE *stream);
```

First argument is character in integer form to be inserted to the file.

Second argument is file pointer returned by fopen( ).

Example:

```
#include<stdio.h>
main( )
{
```

```
FILE *fp;
char ch;
fp=fopen(“example.txt”,”w”);
ch=’v’;
putc(ch,fp);
fclose(fp);
}
```

fgets( )

fgets( ) reads string of character from file. The file must be opened in read mode to scan its contents.

Syntax:

```
int fgets(char *str,int n,FILE *stream);
```

First argument indicates where read strings have to be stored.

Second argument is number of bytes to be read.

Third argument is the file pointer to the file.

Example:

```
#include<stdio.h>
main ()
{
FILE*fp;
char str [100];
fp=fopen (“example.txt”,”r”);
fgets (str,100, fp);
printf(“%s\n”,str);
fclose (fp);
 };
```

fputs( )

fputs( ) writes the string into file. The file must be opened in write mode to add contents to it.

Syntax:

```
int fputs (const  char *str,FILE *stream);
```

First argument indicates data to be written.

Second argument indicate FILE pointer given by fopen().

Example:

```
#include<stdio.h>
main ()
{
FILE *fp;
char str[25]=”Akshay”;
fp=fopen (“example.txt”,”w”);
fputs (str,fp);
fclose (fp);}
```

getw( )

getw( ) is an integer oriented function. It is used to read integer values. This function would be useful when we deal with only interger data. The file must be open in read mode to scan its contents.

Syntax:

```
int getw(FILE * stream);
```

Example:

```
#include<stdio.h>
void main()
{
FILE * fp;
int number;
fp=fopen(“filename.txt”,”r”);
number=getw(fp);
printf(“%d”,number);
fclose(fp);
}
```

It reads a number stored in a file and printed using printf(note:number must be present in a file).

putw( )

putw( ) is an integer function. It is used to write integer values. This function would be useful when we deal with only integer data. The file must be opened in write mode to add contents to it.

Syntax:

```
int putw(integer, fp);
```

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
int number=9;
fp=fopen(“filename.txt”, “w”);
putw(number,fp);
fclose(fp);
}
```

Output: writes number value 10 to file filename.txt

fread( )

fread( ) is used to read data from file.The file must be opened in read mode to scan it contents.

Syntax:

```
int fread(void *str,size_t size_t num,FILE *stream);
```

Where,

str: pointer to a memory block with minimum size of size*num bytes.

size: size in bytes of each element to be read.

num: number of elements of size “size”.

stream: file pointer.

Example:

```
#include<stdio.h>
void main()
```

```
{
FILE *fp;
char str[20];
fp=fopen(“example.txt”,”r”);
fread(str,1,10,fp);
printf(“First 10 characters of file are %s”,str);
fclose(fp);
}
```

It opens file example.txt and reads first 10 characters of file which are of 1 byte size.

fwrite( )

fwrite( ) function is used to write data to file. The file must be opened in write mode to add contents to it.

Syntax:

```
int fwrite(void *str,size_t size,size_t.num,FILE *stream);
```

Where,

str: pointer to array of elements to be written

size: size in bytes of each element to be written

num: number of elements which of size one byte each

stream: file pointer

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
char str[20]=”Ruthvik”;
fp=fopen(“filename.txt”, “w”);
fwrite(str,1,sizeof(str), fp);
fclose(fp);
}
```

Output: it excludes null character while printing.

sizeof(str) is used because it has to write all the characters in string.

Functions for Selecting Records Randomly in a File

C provides following functions for selecting records randomly in a file

fseek( )

ftell( )

fgetpos( )

rewind( )

fsetpos( )

feof( )

fseek( )

fseek() is used to set the file position pointer for the given stream to given offset.

Syntax:

```
int fseek(FILE *stream,long int offset, int position);
```

Where,

stream: File pointer

offset: Number of bytes to offset from position

- ➢ positive means move forward
- ➢ negative means move backward

position: place from where offset is added

- ➢ 0: indicates beginning position
- ➢ 1: indicates current position
- ➢ 2: indicates end position

Consider the following,

fseek(fp,0,0); - Go to the beginning of file.

fseek(fp,0,1); - Stay at current position.

fseek(fp,0,2); - Go to the end of file.

fseek(fp,m,0); - Move to (m+1)th byte in the file.

fseek(fp,m,1); - Go forward by m bytes.

fseek(fp,-m,2); - Go backwards by m bytes from the end of file.

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
fp=fopen (“filename.txt”, “w”);
fputs (“Vicky is nick name of Prakhyath”,fp);
fseek(fp,9,1);
fclose (fp);
}
```

ftell( )

ftell( ) returns the current file pointer position from which next I/O operation is performed.

Syntax:

```
long into ftell (FILE *stream);
```

ftell returns current value of position indicator if successful or else -1.

Example: len=ftell(fp);

Here,

fp is file pointer name,

len gives the size (in bytes) of file pointed by fp if fp is placed at end of file.

Example:

```
#include<stdio.h>
void main()
{
FILE*fp;
intlen ;
fp=fopen(“filename.txt”, “r”);
fseek(fp,0,2);                    /*place the file pointer at end of file */
len=ftell(fp);
printf(“Total size of file =%d bytes\n” ,len);
getch();
}
```

Output: if file is having word Prakhyath in it then it gives output as

Total size of file=9 bytes (since 9 characters each of 1 byte is counted)

rewind( )

rewind( ) is used to adjust the position of file pointer so that next I/O operation take place at the beginning of file.

Syntax:

```
void rewind(FILE *stream);
```

It is similar to fseek(FILE *stream,0,0);

Example:

```
#include<stdio.h>
void main()
{
FILE*fp :
fp=fopen(“file name.txt”, “w”0;
fputs(“ait is in ckm”, fp);
rewind(fp);
fputs(“bit is in dvg”,fp);
fclose(fp);
}
```

fgetpos( )

fgetpos( ) is used to determine the current position of the stream. Returns zero on successful execution and non zero if unsuccessful.

Syntax:

```
int fgetpos(FILE *stream,fpos_t*pos);
```

Where,

stream is file pointer

fpose_t is datatype defined in stdio.h header file

pos is variable declared with fpos_t stores the position information

fsetpos( )

fsetpos( ) is used to move the file pointer to the position indicated by position argument

Syntax:

```
int fsetpos(FILE *stream,fpos_t *pos);
```

Example:

```
 /* Program to illustrate fgetpos( ) and fsetpos( ) */
#include<stdio.h>
void main()
{
  FILE * fp;
  fpos_t position;
  fp=fopen (“file name .txt”,“w”);
  fgetpos(fp,&position );
  fputs(“Prakhyath”,fp);
  fsetpos(fp,&position);
  fputs (“Mangaluru is the Beach City of Karnataka”,fp);
  fclose(fp);
}
```

Error Handling in Files

It is possible that an error may occur during i/o operation on a file, typical error situations include,

- Trying to read beyond the end of file mark.
- Device overflow.

- Trying to use a file that has not been opened.
- Trying to perform an operation on a file, when the file is opened for another type of operation.
- Opening a file with an invalid file name.
- Attempting to manipulate protected file.

The two status inquiry library functions feof( ) and ferror( ) that can help us detect i/o errors in the files.

feof( )

feof( ) is used to test for an end of file condition. This function returns a non zero value when EOF is reached or else it returns zero. It takes a FILE pointer as its only argument and returns a non-zero integer value if all of the data from the specified file has been read, and returns zero otherwise.

Syntax:

```
int feof(FILE *stream);
```

Example:

```
#include <stdio.h>
main( )
{
char str[25];
FILE *fp;
fp=fopen(“sample.txt”,“r”);
if(feof(fp))
printf(“End of file \n”);
fclose(fp);
}
```

ferror( )

The ferror( ) reports the status of the file indicated. It takes a FILE pointer as its argument and returns a non zero integer if an error has been detected up to that point during processing, It returns zero otherwise.

Syntax:

```
int ferror(FILE *stream);
```

Example:

```
#include <stdio.h>
main( )
{
  char str[25];
  FILE *fp;
  fp=fopen("sample.txt","r");
  if(ferror(fp)!=0)
  {
  printf("An error has occurred\n");
  }
  fclose(fp);
}
```

clearerr( )

clearerr( ) is function used to clear the EOF and error indicators for stream

Syntax:

```
int clearerr(FILE *stream);
```

Example: clearer(fp);

perorr( )

perorr( ) is function used to print error message.

Syntax:

```
void perorr(char *message);
```

Example: perorr("file not found");

Note: Whenever a file is opened using fopen( ), a file pointer is returned. If the file cannot be opened for some reasons then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not.

Example:

```
#include <stdio.h>
main( )
{
  char str[25];
  FILE *fp;
  fp=fopen(“sample.txt”,“r”);
  if(fp==NULL)
  {
  printf(“File sample.txt cannot be opened\n”);
  }
  fclose(fp);
}
```