Searching

Introduction:

Searching is a process of finding an element within the list of elements stored in any order or randomly. Searching is divided into two categories Linear and Binary search.

Sequential Search:

In linear search, access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found.

In brief, Simply search for the given element left to right and return the index of the element, if found. Otherwise return “Not Found”.

Algorithm:

```
LinearSearch(A, n,key)
{
        for(i=0;i<n;i++)
        {
                if(A[i] == key)
                    return i;
        }
        return -1;//-1 indicates unsuccessful search
}
```

Analysis:

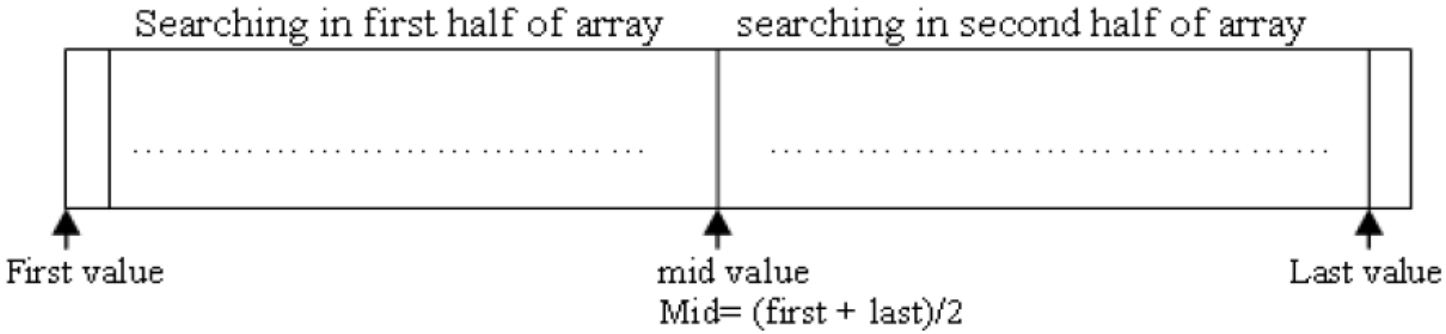
Time complexity = O(n)

Binary Search:

Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do this binary search, first we need to sort the array elements. The logic behind this technique is given below:

- ✔ First find the middle element of the array
- ✔ compare the middle element with an item.
- ✔ There are three cases:
  - ✗ If it is a desired element then search is successful
  - ✗ If it is less than desired item then search only the first half of the array.
  - ✗ If it is greater than the desired element, search in the second half of the array.

Repeat the same process until element is found or exhausts in the search area.
In this algorithm every time we are reducing the search area.



Running example:

Take input array a[] = {2 , 5 , 7, 9 ,18, 45 ,53, 59, 67, 72, 88, 95, 101, 104}

For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since A[mid] = = 2, return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition high = = low, since A[0] != 103, return 0(unsuccessful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since A[mid] = 67, return 9(successful).

Algorithm:

```
BinarySearch(A,l,r, key)
{
        if(l= = r)   //only one element
        {
                if(key = = A[l])
                        return l+1;       //index starts from 0
                else
                        return 0;
        }
        else
        {
                m = (l + r) /2 ;              //integer division
                if(key = = A[m]
                        return m+1;
                else if (key < A[m])
                        return BinarySearch(l, m-1, key) ;
                else
                        return BinarySearch(m+1, r, key) ;
        }
}
```

Efficiency:

From the above algorithm we can say that the running time of the algorithm is

$T(n) = T(n/2) + O(1)$

$= O(\log n)$ (verify).

In the best case output is obtained at one run i.e. $O(1)$ time if the key is at middle.

In the worst case the output is at the end of the array so running time is $O(\log n)$ time.ith

In the average case also running time is $O(\log n)$.

For unsuccessful search best, worst and average time complexity is $O(\log n)$.

Hashing:

It is an efficient searching technique in which key is placed in direct accessible address for rapid search.

Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

A function that transforms a key into a table index is called a hash function.

A common hash function is

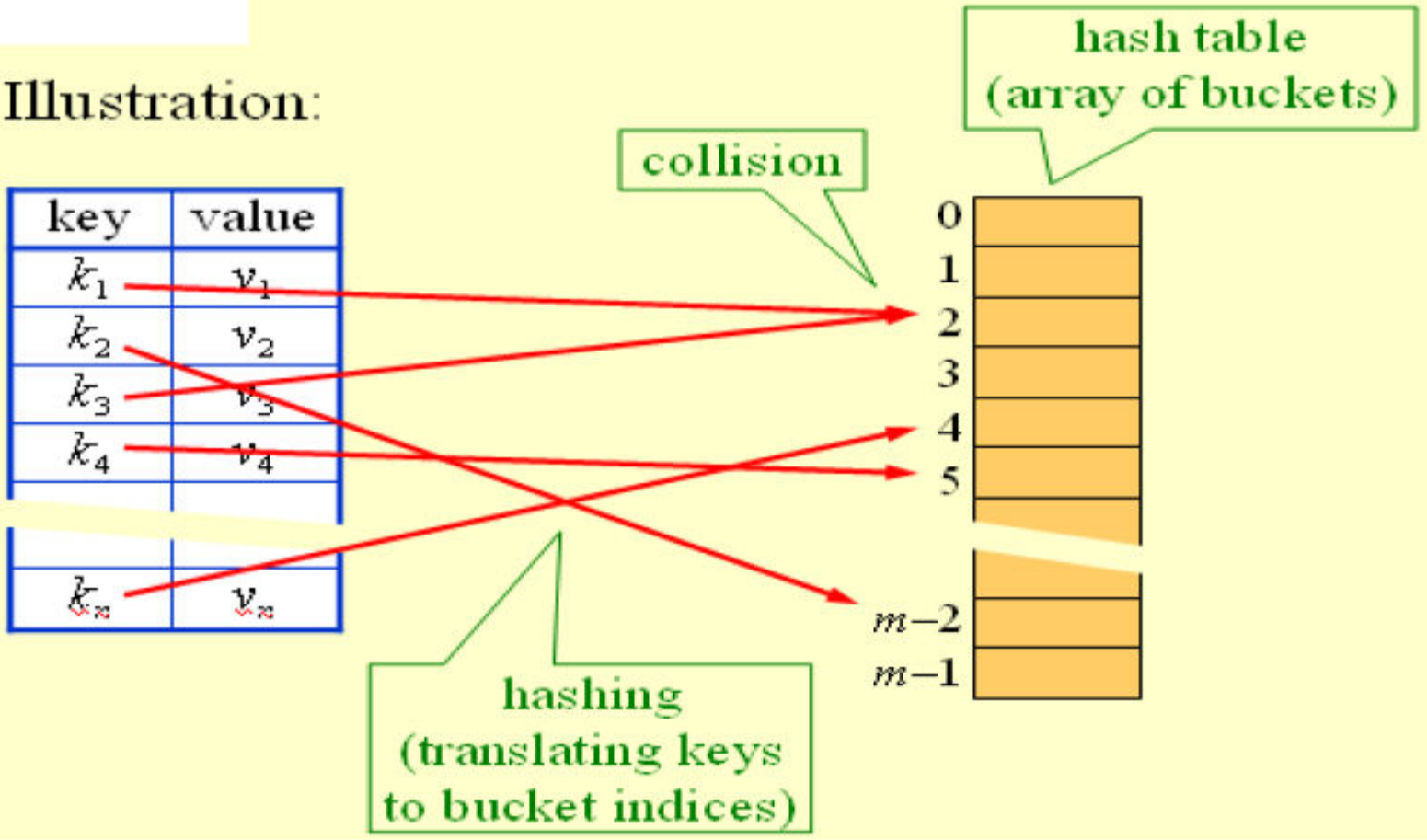
h(x)=x mod SIZE

if key=27 and SIZE=10 then

hash address=27%10=7

Hash-table principles:

- Illustration:



Hash collision:

If two or more than two records trying to insert in a single index of a hash table then such a situation is called hash collision.

Some popular methods for minimizing collision are:

- ✔ Linear probing
- ✔ Quadratic probing
- ✔ Rehashing
- ✔ Chaining
- ✔ Hashing using buckets etc

But here we need only first two methods for minimizing collision

Linear probing:

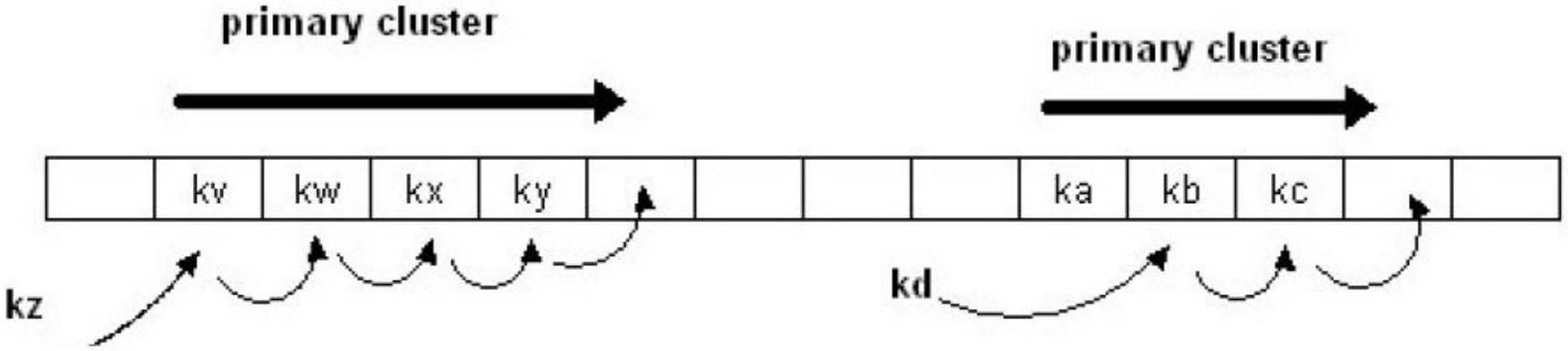
A hash-table in which a collision is resolved by putting the item in the next empty place within the occupied array space.

It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location.

Hence this method searches in straight line, and it is therefore called linear probing.

Disadvantage:

Clustering problem



Example:

Insert keys {89, 18, 49, 58, 69} with the hash function

h(x)=x mod 10 using linear probing.

solution:

when x=89:
h(89)=89%10=9
insert key 89 in hash-table in location 9

when x=18:
h(18)=18%10=8
insert key 18 in hash-table in location 8

when x=49:
h(49)=49%10=9 (Collision occur)
so insert key 49 in hash-table in next possible vacant location of 9 is 0

when x=58:
h(58)=58%10=8 (Collision occur)
insert key 58 in hash-table in next possible vacant location of 8 is 1
(since 9, 0 already contains values).

when x=69:
h(89)=69%10=9 (Collision occur)
insert key 69 in hash-table in next possible vacant location of 9 is 2
(since 0, 1 already contains values).

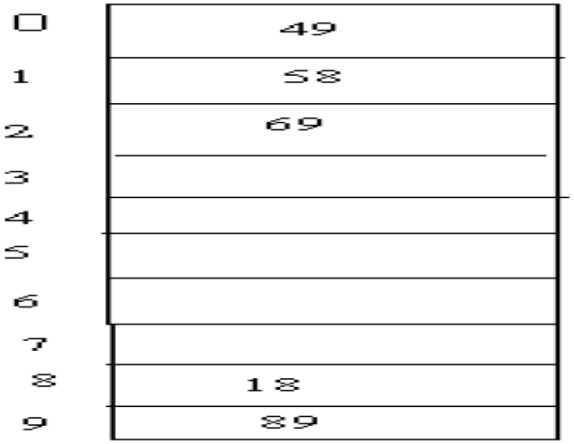


Fig Hash-table for above keys using linear probing

Quadratic Probing:

Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing.

When collision occur then the quadratic probing works as follows:

(Hash value + 1^2)% table size

if there is again collision occur then there exist rehashing.

(hash value + 2^2)%table size

if there is again collision occur then there exist rehashing.

(hash value = 3^2)% table size

in general in ith collision

hi(x)=(hash value +i^2)%size

Example:

Insert keys {89, 18, 49, 58, 69} with the hash-table size 10 using quadratic probing.

solution:

when x=89:

h(89)=89%10=9

insert key 89 in hash-table in location 9

when x=18:

h(18)=18%10=8

insert key 18 in hash-table in location 8

when x=49:

h(49)=49%10=9 (Collision occur)

so use following hash function,

h1(49)=(49 + 1)%10=0

hence insert key 49 in hash-table in location 0

when x=58:

h(58)=58%10=8 (Collision occur)

so use following hash function,

h1(58)=(58 + 1)%10=9

again collision occur use again the following hash function ,

h2(58)=(58+ 22)%10=2

insert key 58 in hash-table in location 2

when x=69:

h(89)=69%10=9 (Collision occur)

so use following hash function,

h1(69)=(69 + 1)%10=0

again collision occur use again the following hash function ,

h2(69)=(69+ 22)%10=3

insert key 69 in hash-table in location 3

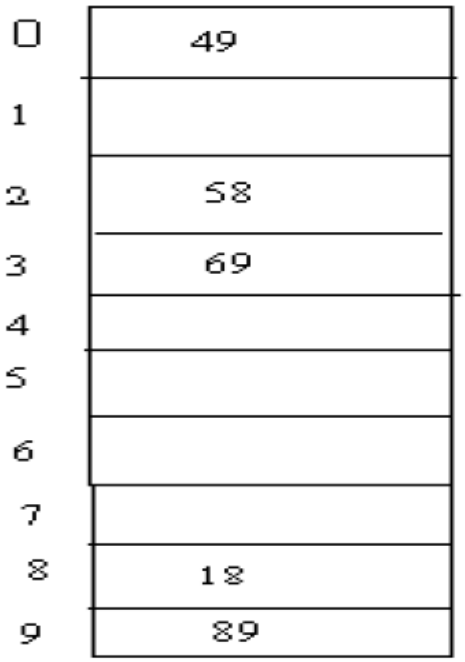
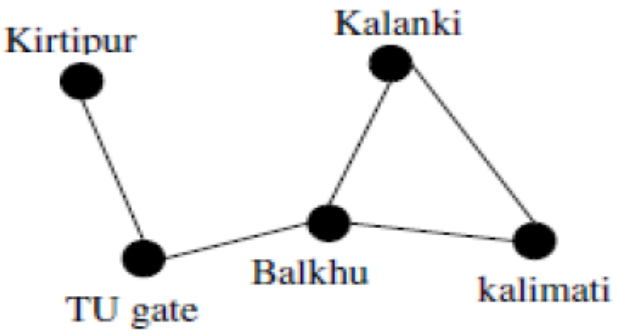


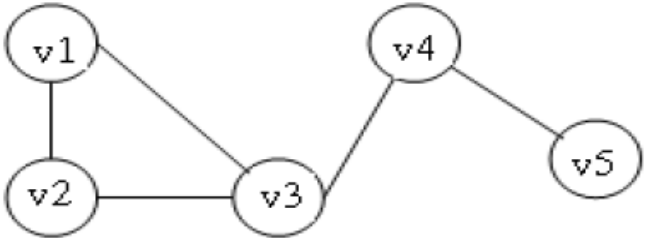
fig:Hash table for above keys using quadratic probing

Graph:

A Graph is a pair G = (V,E) where V denotes a set of vertices and E denotes the set of edges connecting two vertices. Many natural problems can be explained using graph for example modeling road network, electronic circuits, etc. The example below shows the road network.



Let us take a graph:



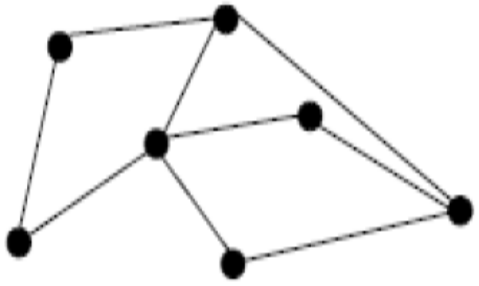
V(G)={v1, v2, v3, v4, v5}
E(G)={(v1,v2),(v2,v3),(v1,v3),(v3,v4),(v4,v5)}

Types of Graph:

Simple Graph:

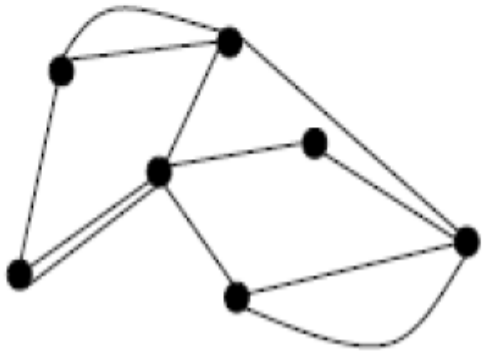
We define a simple graph as 2 – tuple consists of a non empty set of vertices V and a set of unordered pairs of distinct elements of vertices called edges. We can represent graph as G = (V, E). This kind of graph has no loops and can be used for modeling networks that do not have connection to themselves but have both ways connection when two vertices are connected

but no two vertices have more than one connection. The figure below is an example of simple graph.



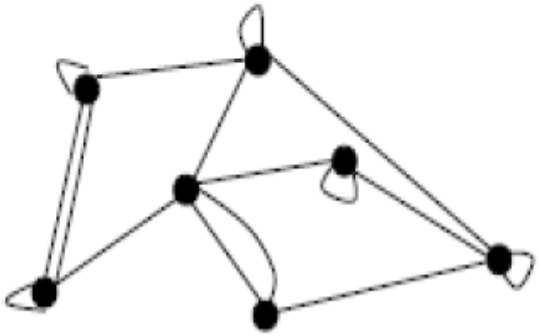
Multigraph:

A multigraph G =(V, E) consists of a set of vertices V, a set of edges E, and a function *f* from E to {{u, v}|u, v Î V, u ¹ v}. The edges e_1 and e_2 are called multiple or parallel edges if $f(e_1) = f(e_2)$. In this representation of graph also loops are not allowed. Since simple graph has single edges every simple graph is a multigraph. The figure below is an example of a multigraph.



Pseudograph:

A pseudograph G =(V, E) consists of a set of vertices V, a set of edges E, and a function *f* from E to {{u, v}|u, v ∈ V}. An edge is a loop if *f(e)* = {u, u} = {u} for some u ∈ V. The figure below is an example of a multigraph



Directed Graph:

A directed graph (V, E) consists of a set V of vertices, a set E of edges that are ordered pairs of elements of V. The below figure is a directed graph. In this graph loop is allowed but

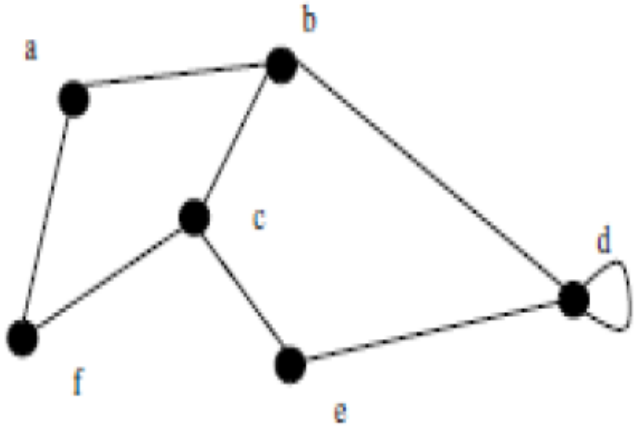
Terminologies:

Two vertices u, v are adjacent vertices of a graph if {u, v} is an edge.

The edge e is called incident with the vertices u and v if e = {u, v}. This edge is also said to connect u and v. where u and v are end points of the edge.

Degree of a vertex in an undirected graph is the number of edges incident with it, except a loop at a vertex. Loop in a vertex counts twice to the degree. Degree of a vertex v is denoted by deg (v).A vertex of degree zero is called isolated vertex and a vertex with degree one is called pendant vertex.

Example: Find the degrees of the vertices in the following graph.



Solution:

deg(a) = deg(f) = deg(e) = 2 ; deg(b) = deg(c) = 3; deg(d) = 4

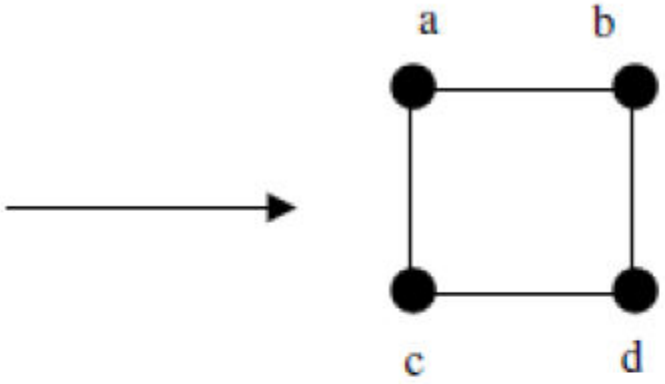
Representation of Graph

Generally graph can be represented in two ways namely adjacency lists(Linked list representation) and adjacency matrix(matrix).

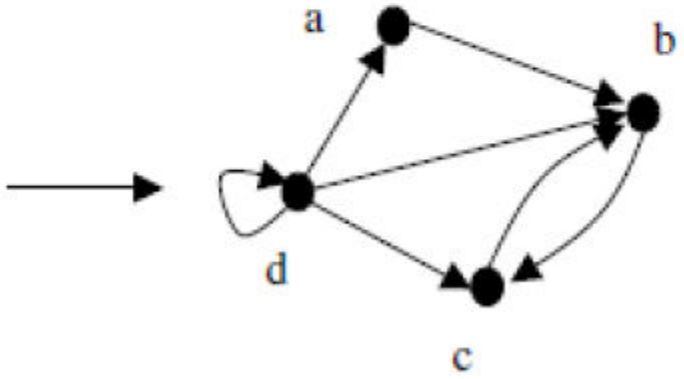
Adjacency List:

This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

Edge List for Simple Graph	
Vertex	Adjacent Vertices
a	b, c
b	a, d
c	a, d
d	b, c



Edge List for Directed Graph	
Initial Vertex	End Vertices
a	b
b	c
c	b
d	a, b, c, d



If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high number of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. In this class we discuss two ways of representing graphs in form of matrix.

Adjacency Matrix:

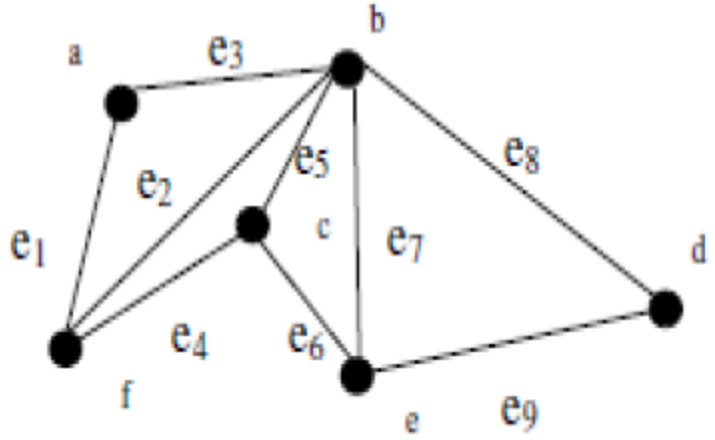
Given a simple graph G =(V, E) with |V| = n. assume that the vertices of the graph are listed in some arbitrary order like $v_1, v_2, \ldots, v_n$. The adjacency matrix A of G, with respect to the order of the vertices is n-by-n zero-one matrix ($A = [a_{ij}]$) with the condition,

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

Since there are n vertices and we may order vertices in any order there are n! possible order of the vertices. The adjacency matrix depends on the order of the vertices, hence there are n! possible adjacency matrices for a graph with n vertices.

In case of the directed graph we can extend the same concept as in undirected graph as dictated by the relation

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

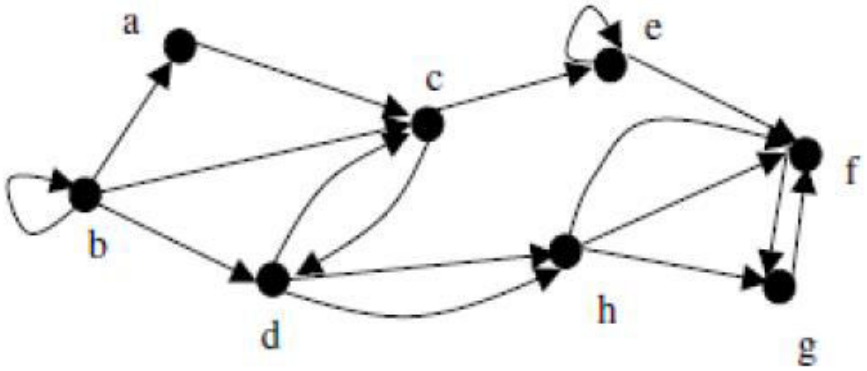


Solution:Let the order of the vertices be a, b, c, d, e, f

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix

Let us take a directed graph



Solution:

Let the order of the vertices be a, b, c, d, e, f, g

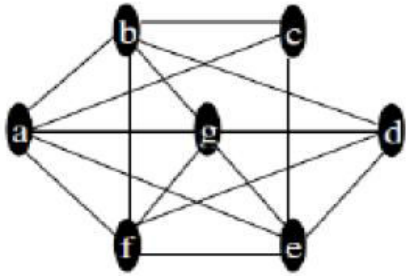
$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \end{bmatrix}$$

Graph Traversals

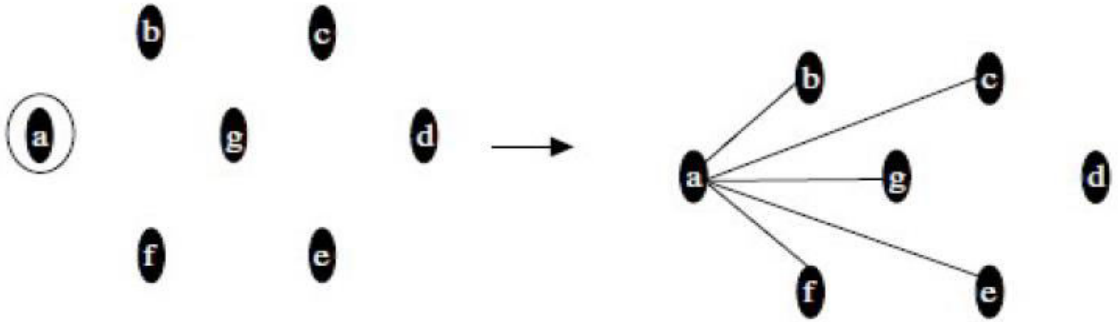
Breadth-first search:

This is one of the simplest methods of graph searching. Choose some vertex arbitrarily as a root. Add all the vertices and edges that are incident in the root. The new vertices added will become the vertices at the level 1 of the BFS tree. Form the set of the added vertices of level 1, find other vertices, such that they are connected by edges at level 1 vertices. Follow the above step until all the vertices are added.

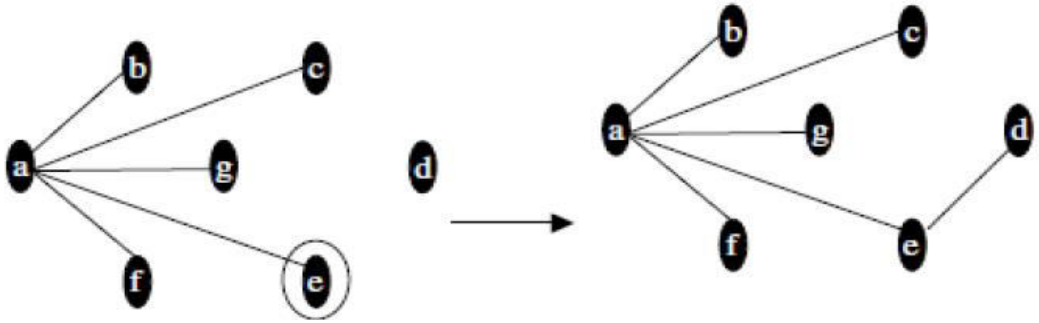
Example:
Use breadth first search to find a BFS tree of the following graph



Solution:
Choose a as initial vertex then we have



Order the vertices of level 1 i.e. {b, c, g, e, f}. Say order be {e, f, g, b, c}.



Algorithm:

```
BFS(G,s) //s is start vertex
{
      T = {s};
      L =Φ; //an empty queue
      Enqueue(L,s);
      while (L != Φ )
      {
            v = dequeue(L);
            for each neighbor w to v
            if ( w∉ L and w ∉ T )
            {
                  enqueue( L,w);
                  T = T U {w}; //put edge {v,w} also
            }
      }
}
```

Analysis

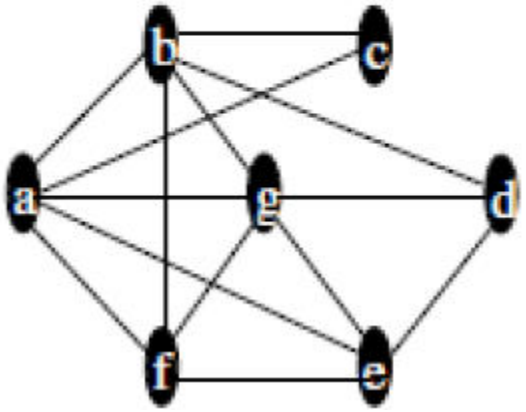
From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in O(n) time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as O(n2).

Depth First Search:

This is another technique that can be used to search the graph. Choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met. If there is such a path continue the process above. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called **backtracking**.

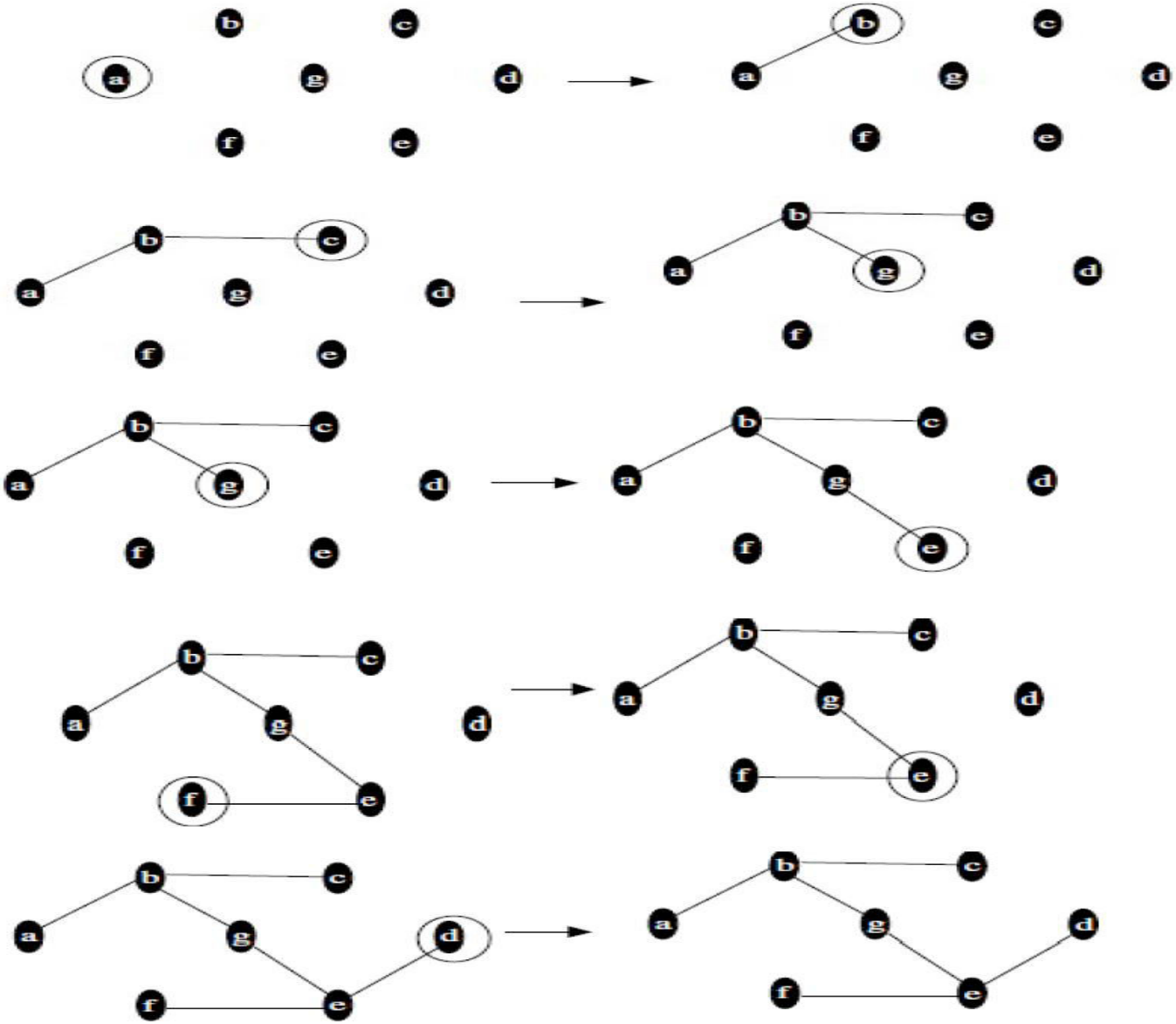
Example:

Use depth first search to find a spanning tree of the following graph.



Solution:

Choose a as initial vertex then we have



Algorithm:

```
DFS(G,s)
{
        T = {s};
        Traverse(s);
}
Traverse(v)
{
        for each w adjacent to v and not yet in T
        {
                T = T U {w}; //put edge {v,w} also
                Traverse (w);
        }
}
```

Analysis:

The complexity of the algorithm is greatly affected by **Traverse** function we can write its running time in terms of the relation $T(n) = T(n-1) + O(n)$, here $O(n)$ is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is $O(n_2)$.

Minimum Spanning Trees

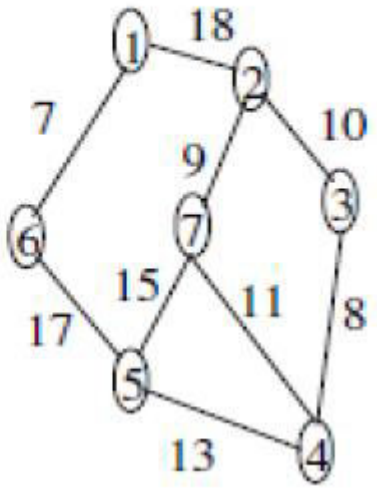
A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges. In this part we study one algorithm that is used to construct the minimum spanning tree from the given connected weighted graph.

Kruskal's Algorithm:

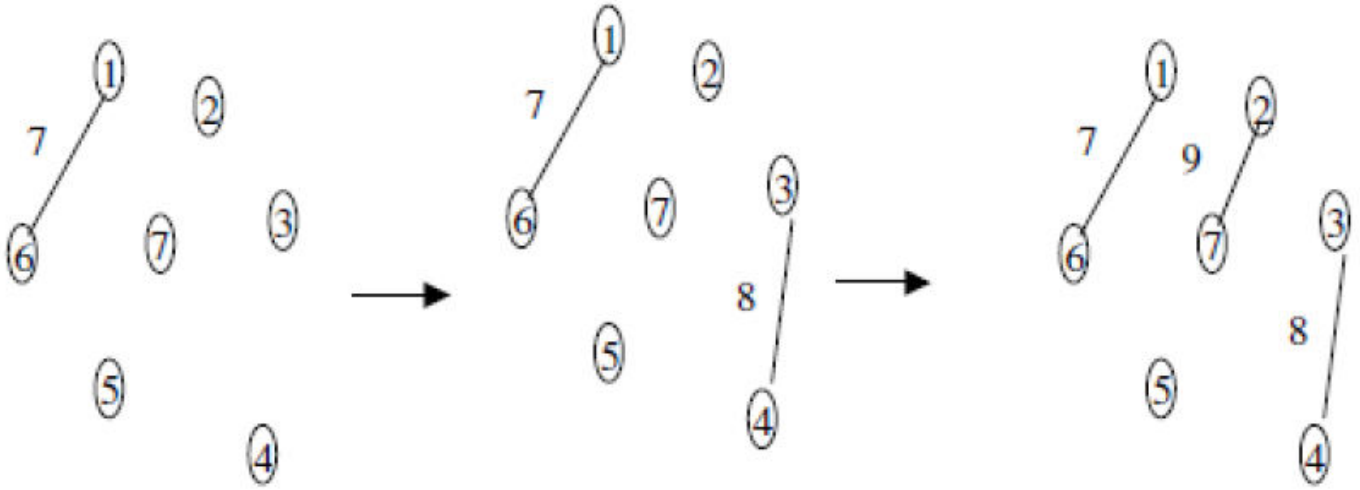
The problem of finding MST can be solved by using Kruskal's algorithm. The idea behind this algorithm is that you put the set of edges form the given graph G = (V,E) in nondecreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will be the minimum. Note that we have G as a graph, V as a set of n vertices and E as set of edges of graph G.

Example:

Find the MST and its weight of the graph.

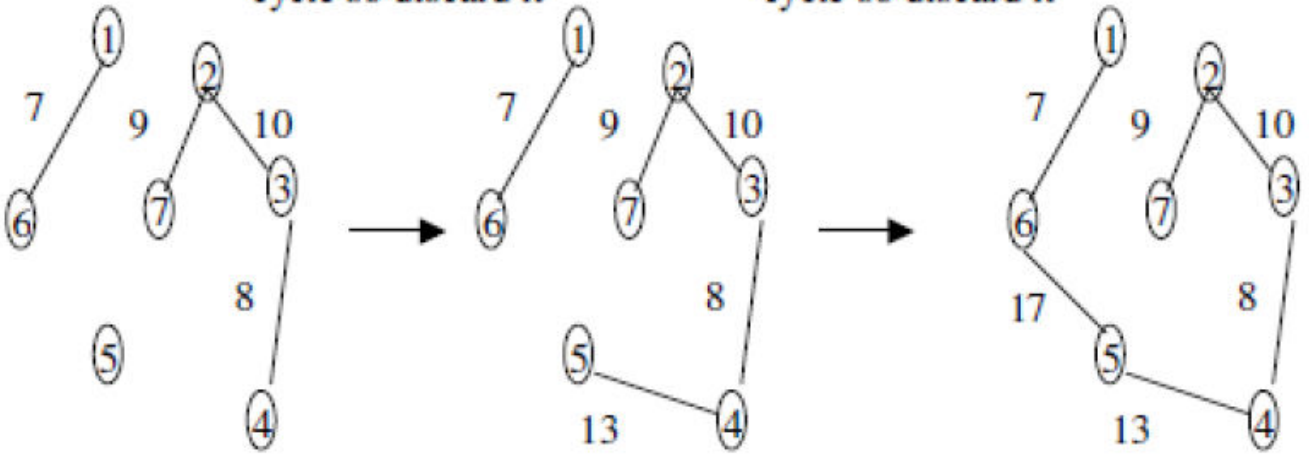


Solution:



Edge with weight 11 forms cycle so discard it

Edge with weight 15 forms cycle so discard it



The total weight of MST is 64.

Algorithm:

```
KruskalMST(G)
{
        T = {V} // forest of n nodes
        S = set of edges sorted in nondecreasing order of weight
        while(|T| < n-1 and E !=Æ)
        {
                Select (u,v) from S in order
                Remove (u,v) from E
                if((u,v) doesnot create a cycle in T))
                T = T È {(u,v)}
        }
}
```

Analysis:

In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE)

Thanks You

Any Queries?