DATA STRUCTURE
Graphs
SEARCHING & SORTING
UNIT III

 U3.‹#›

Graphs

 U3.‹#›

Learning Objectives

- Graphs
- Terminology
- Representation
- Basic operations
- DFS Traversal
- BFS Traversal
- Verify Graph for any Cycle
- Transitive Closure
  - Warshell's Algorithm
- Topological Sort
- Shortest Path

 U3.‹#›

Learning Objectives Cont..

- Minimum Spanning Tree
  - Prims' Algorithm
  - Kruskals' Algorithm
- Critical Path Analysis
- Graph Coloring

Graphs

Is a non-linear data structure

Used in order to represent a scenario where there can be a many to many relationship between parent to child nodes

Graphs serve as models for various processes and structures like:

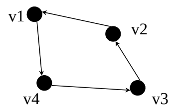
- Airline routes
- Flowchart of a program
- Message transmission in a network
- Cities and highways

Mathematical Definition

A **graph** G consists of two sets

- *V* whose members are **vertices** of *G*
- *E* whose members are pairs of vertices form *V* , and are termed as edges of G

Example



The graph in the figure given above is represented as

- V {v1, v2, v3, v4}
- E { (v1,v4), (v4,v3), (v3, v2), (v2,v1) }

Terminology

In the pair of vertices representing an edge are

- unordered => the graph is said to be **undirected graph**
- ordered => the graph is said to be **directed graph** or a **diagraph**

Weighted Graph: A graph having a weight / cost associated with each edge

Adjacent Vertices: Vertex v1 is said to be adjacent to v2 if there is an edge from v1 to v2

Path: A sequence of distinct vertices each adjacent to the next

Cycle: A path containing at least three vertices such that the last vertex is adjacent to first.

Computer Representation

While writing a program a graph can be implemented using:

- **Matrices**: Adjacency Matrix representation
- **Linked Lists**: Adjacency List representation

Adjacency Matrix Representation

Adjacency Matrix

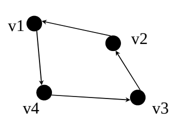
- A two-dimensional array of boolean values
- Assuming each vertex is represented by an index number
- An element **A[I][J]** is set to **true** if and only if the vertex **I** is **adjacent to** vertex **J**
- If the graph is undirected then **A[I][J]** = **A[J][I]**

Adjacency Matrix Representation

The structure:

```
struct Graph {
    int countOfVertices;
    char labels [MAX]
    Boolean adjMatrix [MAX][MAX];
};
```


Adjacency Matrix Representation



0	V1
1	V2
2	V3
3	V4

Destination Vertices →

Source Vertices ↓

0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0

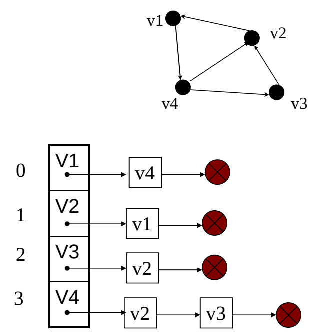
Linked Representation

Represented using two lists
- A list of vertices; and
- For each vertex, a list of adjacent vertices

The lists in turn can be represented as
- **contiguous** lists; or
- **linked** lists

A common approach is a **mixed representation** where:
- Vertices are represented by a contiguous list; and
- List of vertices adjacent to each vertex are stored as a linked list.

Linked Representation: Mixed Approach



Linked Representation: Structure

```
struct ENode {
       int index;
       struct ENode *next;
};

struct VNode {
   LabelType label; // can be used to name vertices
   struct ENode * first;
   struct Vnode *next;
};
```

Linked Representation: Structure

```
struct GraphMixedRepresentation {
  int count;       //number of vertices;
  struct VNode Vertices [MAX];
};

        struct GraphLinkedRepresentation {
          struct VNode *next;
        };
```

 U3.‹#›

Basic Operations on Graphs

Creation / Building
- Add a node
- Remove a node
- Add an edge
- Remove an edge

Simple Operations
- Generate Adjacency list of a node
- Calculate in-degree for a node
- Calculate out-degree for a node

Traversal
- Depth First
- Breadth First

 U3.‹#›

Adding / Removing a Node

Matrix Representation:
- Will require adjusting the size of adjacency matrix

Mixed Representation
- Will require adjusting the size of contiguous list that represents vertices

 U3.‹#›

Adding / Removing an Edge

Matrix Representation:

- Will require setting the corresponding element in the matrix to 0/1

Mixed Representation

- Will require adding / removing a link to the list representing adjacent nodes.

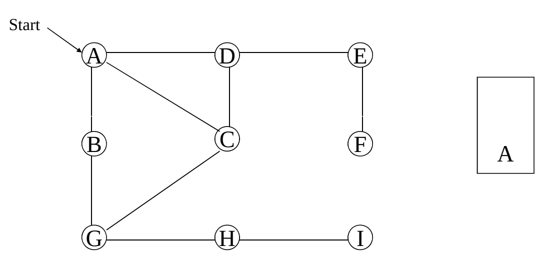
Graph Traversals

Depth First
Breadth First

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

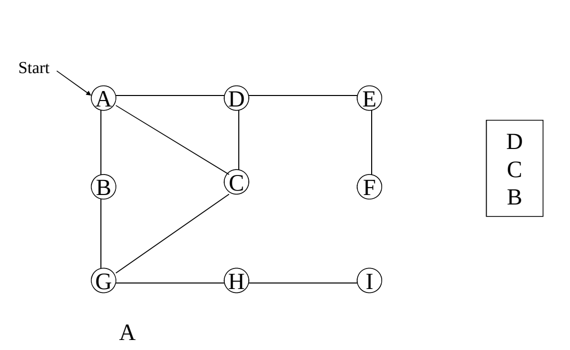
DFS Traversal

Roughly analogous to pre-order traversal of a tree

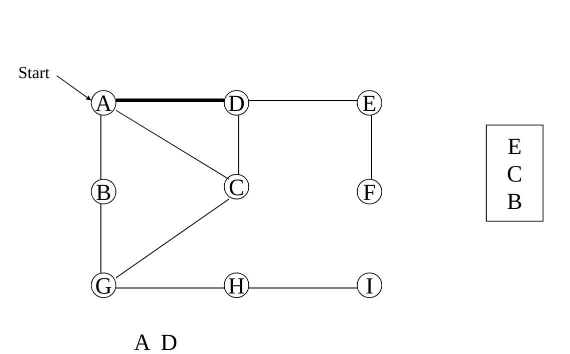


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

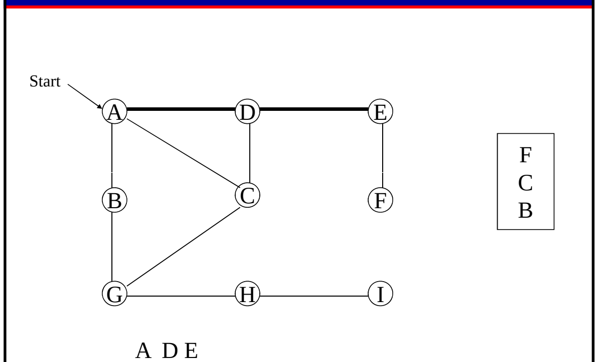
DFS Traversal

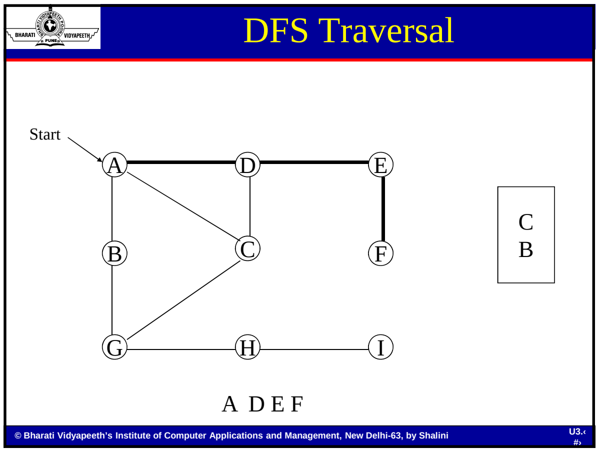


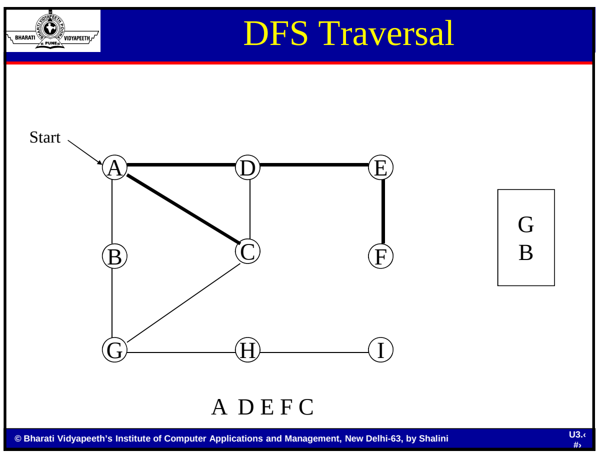
DFS Traversal

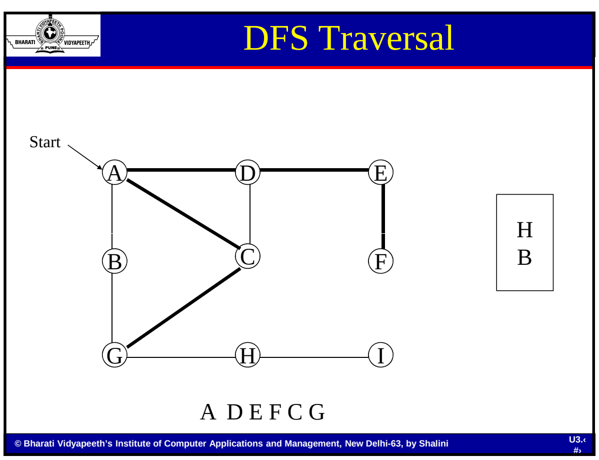


DFS Traversal

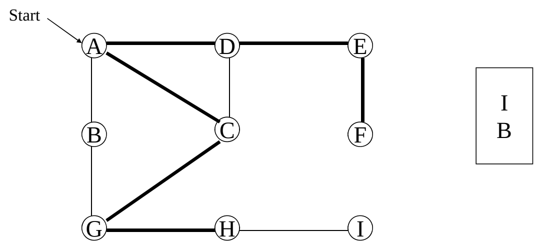



DFS Traversal
Start
A
D
E
B
C
F
G
H
I
C
B
A D E F
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini


DFS Traversal
Start
A
D
E
B
C
F
G
H
I
G
B
A D E F C
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

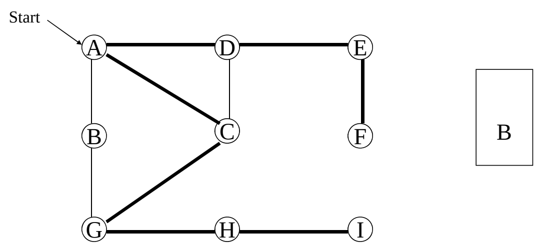

DFS Traversal
Start
A
D
E
B
C
F
G
H
I
H
B
A D E F C G
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

DFS Traversal



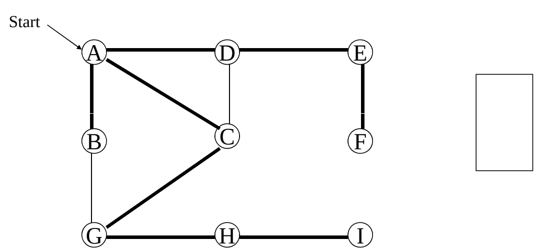
A D E F C G H

DFS Traversal



A D E F C G H I

DFS Traversal

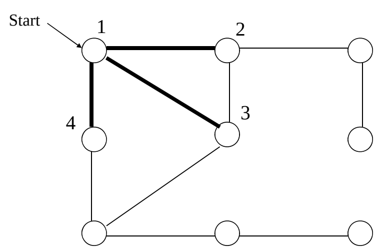


A D E F C G H I B

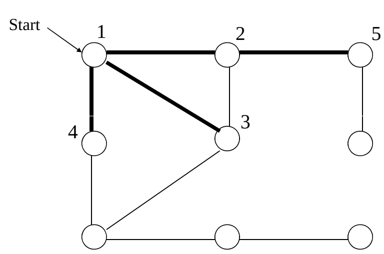
DFS Traversal

```
Push (Stk, VStart)
While Stk Is Not Empty
    Vtemp := Pop (Stk)
    If VTemp is Not Marked
        Process VTemp
        Mark VTemp
        For Each VAdj Adjacent to VTemp
            Push (Stk, VAdj)
        End For
    End IF
End While
```

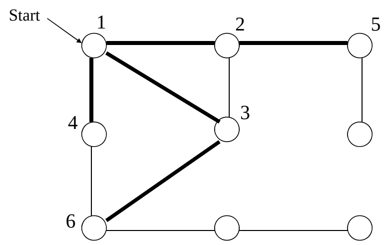

BFS Traversal



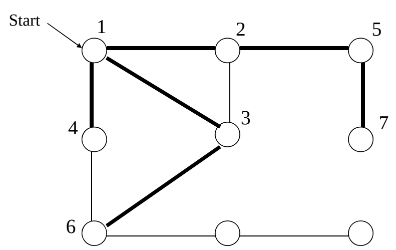
BFS Traversal



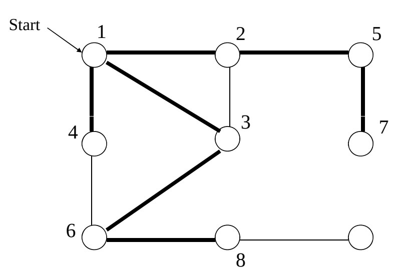
BFS Traversal



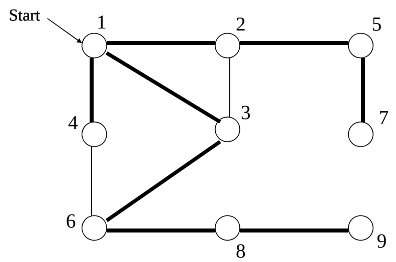
BFS Traversal



BFS Traversal



BFS Traversal



BFS Traversal

Enqueue **(Q, VStart)**
While Q Is Not *Empty*
    Dequeue **(Q, VTemp)**
    If VTemp is Not *Processed*
        Process **VTemp**
        Mark **VTemp**
        For Each VAdj Adjacent to VTemp
            Enqueue **(Q, VAdj)**
        End For
    End IF
End While

TRANSITIVE CLOSURE

Transitive Closure

The **transitive closure** of a graph G:

- A graph G* is the transitive closure of G if and only if
- For any two edges {a,b}, {b,c} there exists {a,c}
  - ✓i.e., {a,b} ∈ G*, if path exists from a to b in G.

Finding Transitive Closure

- Matrix Multiplication Method
- Warshall's Algorithm

Can be used to check reach ability, i.e. what other nodes can be reached from what node

Matrix Multiplication Method

Given the adjacency Matrix of a Graph G as A

Paths of length d: A^d = A x A x …. A (d times)

We can define the **path matrix** of ***order d*** as:

- A matrix where M[i][j]= Count of paths of length ***d or less*** between V_i and V_j
- **path^d** = $A + A^2 + A^3 + \ldots + A^d$

Transitive Closure: Substitute 1 for each non-zero entry in the path matrix of order N

Warhsall's Algorithm

Aim

- Calculate the transitive closure

Basic Working

- Initialize the Path Matrix using the Adjacency Matrix
  - ✓(i.e. P[I][J] is set if there is a direct path from Vi to Vj)
- During k^{th} iteration set V[I][J]
  - ✓if there is a path from Vi to Vj
  - ✓either directly or via Vk

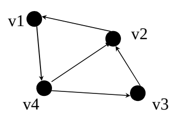
Result

- A matrix whose element P[I][J] is set if there is a path from Vi to Vj either directly or indirectly

Warhsall's Algorithm

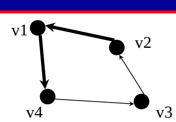
```
Copy A to P
For K=0 to N-1
   For I=0 to N-1
         For J=0 to N-1
            P[I][J]= P[I][J] | ( P[I][K] & P[K][J])
         End For
      End For
   End For
```


Example: Warshall's Algorithm



0	0	0	1
1	0	0	0
0	1	0	0
0	1	1	0

Example: Warshall's Algorithm

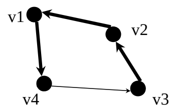


0	0	0	1
1	0	0	0
0	1	0	0
0	1	1	0

0	0	0	1
1	0	0	**1**
0	1	0	0
0	1	1	0

After K = 0

Example: Warshall's Algorithm



0	0	0	1
1	0	0	1
0	1	0	0
0	1	1	0

→

0	0	0	1
1	0	0	1
1	1	0	**1**
1	1	1	**1**

After K = 1

Example: Warshall's Algorithm

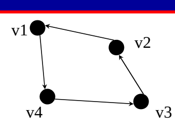
0	0	0	1
1	0	0	1
1	1	0	1
1	1	1	1

→

0	0	0	1
1	0	0	1
1	1	0	1
1	1	1	1

After K = 2

Example: Warshall's Algorithm



0	0	0	1
1	0	0	1
1	1	0	1
1	1	1	1

→

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

After K = 3

TOPOLOGICAL SORT

Topological Sort

Ordering of vertices in a directed acyclic graph, such that:
- if there is a path from V_i to V_j then Vi precedes before V_j in the sorting

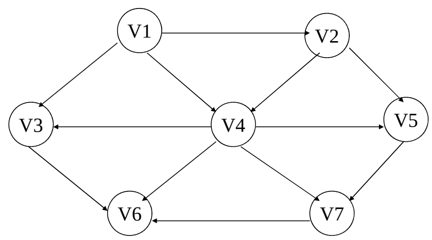
Topological Sorting is not possible for a graph containing cycles.
- As, for two vertices *v* and *w* in a cycle
- *V* precedes *w* and *w* precedes *v*

Applications & Strategy

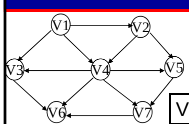
Topological Sorting can be used to:
- Cyclic dependencies in formulae
- Glossary of terms so that no term appears before it is defined

Strategy:
- Find a vertex with no incoming edges (in-degree zero)
- Process it
- Remove it from graph
- Continue with the same strategy till there are no more vertices in graph.

Example



Results



In-degrees before removing V#

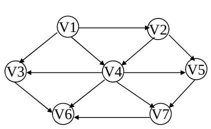
V1	0	-	-	-	-	-	-
V2	1	0	-	-	-	-	-
V3	2	1	1	0	-	-	-
V4	2	1	0	-	-	-	-
V5	2	2	1	0	0	-	-
V6	3	3	3	2	1	1	0
V7	2	3	2	1	1	0	-

O/P	V1	V2	V4	V3	V5	V7	V6

Algorithm: Topological Sort

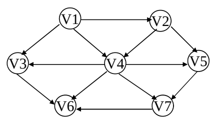
```
Calculate InDegree for all vertices -> IDeg [N]

For I = 1 to N
    If Ideg [I] == 0
       Enqueue (Q, V[I])
    End If
End For
```



Cont..

```
While Q Is Not Empty
   Dequeue (Q, VTemp)
   TopSort [Count++] <- VTemp
   For Each VAdj Adjacent to VTemp
      Decrement InDegree of VAdj
      If InDegree of VAdj == 0
         Enqueue (Q, VAdj)
      End IF
   End For
End While
```



O/P	V1	V2	V4	V3	V5	V7	V6

Cont..

```
If Count < N
   Graph has a Cycle
Else
   Return TopSort
End If
```


SHORTEST PATH

Shortest Path

Applications

- Finding shortest route from location X to location Y
- Finding communication paths with least cost in a computer network
- Finding transit paths with least cost in a railway network
- And many more…

Problems

- Finding shortest paths from a distinguished vertex **S** to every other vertex in a weighted graph **G**
- Finding shortest paths between *all possible pairs of vertices* in a weighted graph **G**

Dijkstra's Algorithm

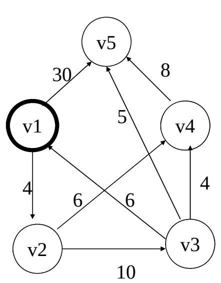
Used in order to find the shortest paths from a distinguished vertex **S** to every other vertex in a weighted graph **G**

Termed as ***Single Source Shortest Path*** algorithm

Technique:

- Select the vertex V having the least D_v among all the unknown vertices
- Declare the smallest path from S to V to be *known*
- For all unknown W adjacent to V, set $D_w = D_v + C_{v,w}$ if this value is lesser than the current D_w
- Set V to be the node that precedes W on the path from S to W

Dijkstra's SSSP



Initially

	Known	D	P
V1	1	0	0
V2	0	∞	0
V3	0	∞	0
V4	0	∞	0
V5	0	∞	0

Dijkstra's SSSP

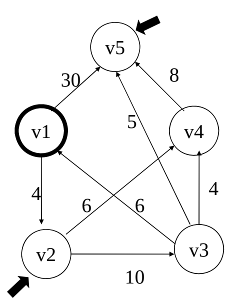
Start Point V1 =>

- D(V1)=0
- Unknown Vertex with Smallest Distance -> V1
- Declare V1 as ***Known***

Consider Unknown Vertices adjacent to V1 (V2, V5)

- D(V1) + D(V1, V2) < D(V2) => Update D(V2) & P(V2)
- D(V1) + D(V1, V5) < D(V5) => Update D(V5) & P(V5)

Dijkstra's SSSP: After V1 is *Known*



	Known	D	P
V1	1	0	0
V2	0	4	V1
V3	0	∞	0
V4	0	∞	0
V5	0	30	V1

Dijkstra's SSSP

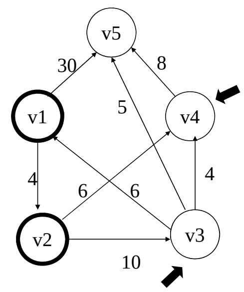
Unknown Vertex with Smallest Distance

- V2
- Declare V2 as ***Known***

Consider Unknown Vertices adjacent to V2 (V3, V4)

- D(V2) + D(V2, V3) < D(V3) => Update D(V3) & P(V3)
- D(V2) + D(V2, V4) < D(V4) => Update D(V4) & P(V4)

Dijkstra’s SSSP: After V2 is *Known*



	Known	D	P
V1	1	0	0
V2	1	4	V1
V3	0	14	V2
V4	0	10	V2
V5	0	30	V1

Dijkstra’s SSSP

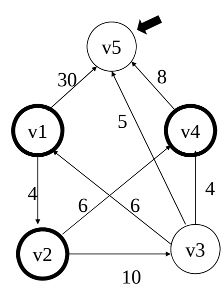
Unknown Vertex with Smallest Distance

- V4
- Declare V4 as ***Known***

Consider Unknown Vertices adjacent to V4 (V5)

- D(V4) + D(V4, V5) < D(V5) => Update D(V5) & P(V5)

Dijkstra’s SSSP: After V4 is *Known*



	Known	D	P
V1	1	0	0
V2	1	4	V1
V3	0	14	V2
V4	1	10	V2
V5	0	18	V4

Dijkstra's SSSP

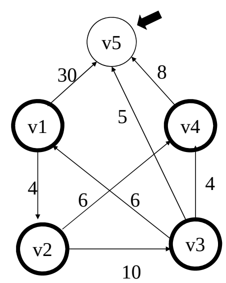
Unknown Vertex with Smallest Distance

- V3
- Declare V3 as ***Known***

Consider Unknown Vertices adjacent to V3 (V5)

- D(V3) + D(V3, V5) > D(V5) =>
  - ✓No Updation required for V5

Dijkstra's SSSP: After V3 is *Known*



	Known	D	P
V1	1	0	0
V2	1	4	V1
V3	1	14	V2
V4	1	10	V2
V5	0	18	V4

Dijkstra's SSSP

Unknown Vertex with Smallest Distance

- V5
- Declare V5 as ***Known***

Consider Unknown Vertices adjacent to V5

- None
- Shortest Paths to all nodes starting from V1 are known
- **Done**

Dijkstra's SSSP: Data

```
Struct NodeInfo
{
   LabelType Label; //(or better int vertexIndex)
   Boolean Known;
   DistanceType Distance;
   Vertex Previous; //(or better int prevVertexIndex)
};
```


Dijkstra's SSSP: Initialization

Create an array of type **NodeInfo**

For all elements

- Set Known to False
- Set Distance to Infinity
- Set Previous to NULL

For the *Start* node

- Set Known to True
- Set Distance to 0

Dijkstra's SSSP: Algorithm

```
Void Dijkstra (Graph G, Vertex Start)
Begin

  NodeInfo Nodes [COUNT];
  Initialize (Nodes, Start);

  While (TRUE)
     VLoc= SmallestDistanceUnknownVertex ();
     if ( VLoc == -1 )
           break;
     Nodes [VLoc]. Known = TRUE;
```

Dijkstra's SSSP: Algorithm

```
    For  Each VAdj AdjacentTo  VLoc
         If ( ! Nodes[VAdj].Known )
                Temp=Nodes[Vloc].Distance+Cost(Vloc, Vadj)
                IF ( Temp < Nodes[Vadj].Distance)
                         Nodes[Vadj].Distance= Temp;
                         Nodes[Vadj].Previous= Vloc;
                End IF
         End IF
    End For
End While
End
```


To Do

Use Recursion to print the actual path

Shortest Paths for All Pairs of Vertices: Floyd Warshall's

Aim:

- Find shortest weighted path between each pair of vertices in of directed graph

Algorithm:

- **Floyd-Warshall's**

Strategy:

- As an extension of Warshall's algorithm, apart from checking whether there is a path from V_i to V_j via V_k
- Calculate the new path length D[I][J] as D[I][K] + D[K][J]
- Use the shorter of the two and set *previous vertex* accordingly.

All Pair Shortest Path

D: Distance / Cost Matrix

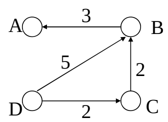
P: Intermediate Node

Initialization

- If there is no edge connecting Vi & Vj then set D[I][J] ->∞
- Else set D[I][J] to the weight of the edge (Vi, Vj)

- Set P[I][J] to NULL

All Pair Shortest Path: Illustration



Initialization

∞	∞	∞	∞
3	∞	∞	∞
∞	2	∞	∞
∞	5	2	∞

-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

All Pair Shortest Path: Illustration

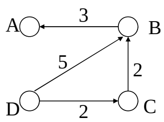
Update D[I][J] if

- D[I][0] + D[0][J] < D[I][J]

The above condition is false for all (Vi, Vj)

No Updations are made

All Pair Shortest Path: Illustration



After First Iteration

∞	∞	∞	∞
3	∞	∞	∞
∞	2	∞	∞
∞	5	2	∞

-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

All Pair Shortest Path: Illustration

Update D[I][J] if

- D[I][1] + D[1][J] < D[I][J]

D[2][1] + D[1][0] = 5 < D[2][0]

Thus;

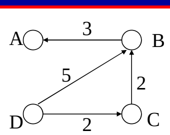
- Update D[2][0] to 5
- Set P[2][0] to 1

D[3][1] + D[1][0] = 8 < D[3][0]

Thus

- Update D[3][0] to 8, Set P[3][0] to 1

All Pair Shortest Path: Illustration



After Second Iteration

∞	∞	∞	∞
3	∞	∞	∞
5	2	∞	∞
8	5	2	∞

-	-	-	-
-	-	-	-
1	-	-	-
1	-	-	-

All Pair Shortest Path: Illustration

Update D[I][J] if

- D[I][2] + D[2][J] < D[I][J]

D[3][2] + D[2][0] = 7 < D[3][0]

Thus;

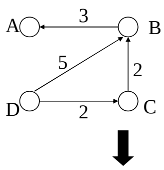
- Update D[3][0] to 7
- Set P[2][0] to 2

D[3][2] + D[2][1] = 4 < D[3][1]

Thus

- Update D[3][1] to 4, Set P[3][1] to 2

All Pair Shortest Path: Illustration



After Third Iteration

∞	∞	∞	∞
3	∞	∞	∞
5	2	∞	∞
7	4	2	∞

-	-	-	-
-	-	-	-
1	-	-	-
2	2	-	-

All Pair Shortest Path: Illustration

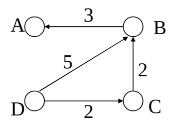
Update D[I][J] if

- D[I][3] + D[3][J] < D[I][J]

The above condition is false for all (Vi, Vj)

No Updations are made

All Pair Shortest Path: Illustration



After Fourth Iteration

∞	∞	∞	∞
3	∞	∞	∞
5	2	∞	∞
7	4	2	∞

-	-	-	-
-	-	-	-
1	-	-	-
2	2	-	-

All Pair Shortest Path: Illustration

Finding Cost of traveling from D to A

- D[3][0]= 7 => **Path Exists**

Finding Path from D to A

- P[3][0]= 2
- P[2][0]= 1
- P[1][0]= NULL but D[1][0] is not Infinity
  - ✓Thus, there is a direct edge from B to A

Path: 3, 2, 1, 0 or D, C, B, A

All Source Shortest Path: Algorithm

```
Void FloydWarshall
  ( TwoDimArray D, TwoDimArray P, Int N)
Begin
  For K=0 to N-1
      For I=0 to N-1
            For J=0 to N-1
                  If ( D[I][J] < D[I][K] + D[K][J] )
                  Begin
                        D[I][J] = D[I][K] + D[K][J]
                        P[I][J]= K
                  End
End
```

To Do

Use Recursion to print the actual path

Get Path: Algorithm

```
Void GetPath(i, j)
Begin
  If D[i][j] equals Infinity then
     return "No Path"
  intermediate = P[i][j]
  if intermediate equals Null then
     return " "
  else
     return GetPath(i, intermediate)+
            intermediate+
            GetPath( intermediate,j)
```


MINIMUM SPANNING TREES

Minimum Spanning Tree

Definition

- A tree formed from edges of graph G that connects all the the vertices of G at lowest total cost.

A minimum spanning tree exists if and only if G is connected

The number of edges in a minimum spanning tree is Count(Vertices) –1

Applications

- Finding minimum cost of laying down cables
- Finding minimum cost of constructing roads

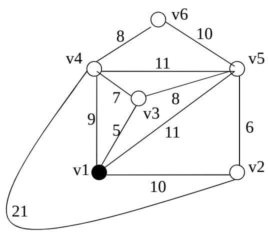
Common Algorithms

Prim's
Kruskal's

Prim's Algorithm

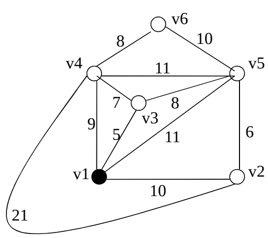
- The actual algorithm is a slight variation of Dijkstra's algorithm
- D_v is the weight of shortest arc connecting V to a known vertex.
- After V is selected for each unknown W adjacent to V
  - D_w= Min (D_w, $C_{w,v}$)
- The final tree is obtained from the collection of selected edges in the table

Prim's: Illustration



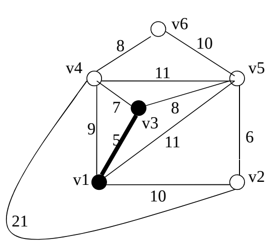
V1	1	0	-
V2	0	∞	-
V3	0	∞	-
V4	0	∞	-
V5	0	∞	-
V6	0	∞	-

Prim's: Illustration



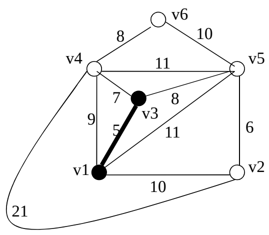
V1	1	0	-
V2	0	10	V1
V3	0	5	V1
V4	0	9	V1
V5	0	11	V1
V6	0	∞	-

Prim's: Illustration



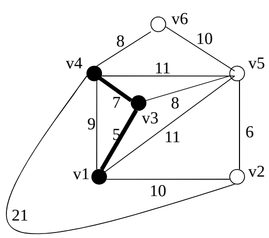
V1	1	0	-
V2	0	10	V1
V3	1	5	V1
V4	0	9	V1
V5	0	11	V1
V6	0	∞	-

Prim's: Illustration



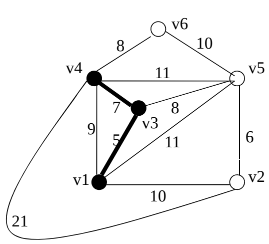
V1	1	0	-
V2	0	10	V1
V3	1	5	V1
V4	0	7	V3
V5	0	8	V3
V6	0	∞	-

Prim's: Illustration



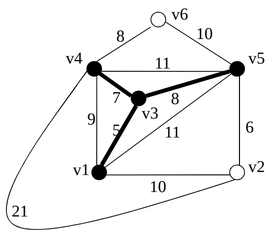
V1	1	0	-
V2	0	10	V1
V3	1	5	V1
V4	1	7	V3
V5	0	8	V3
V6	0	∞	-

Prim's: Illustration



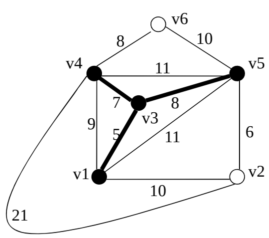
V1	1	0	-
V2	0	10	V1
V3	1	5	V1
V4	1	7	V3
V5	0	8	V3
V6	0	8	V4

Prim’s: Illustration



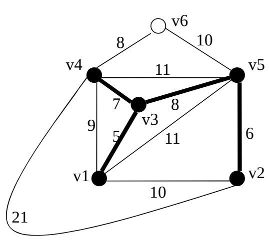
V1	1	0	-
V2	0	10	V1
V3	1	5	V1
V4	1	7	V3
V5	1	8	V3
V6	0	8	V4

Prim’s: Illustration



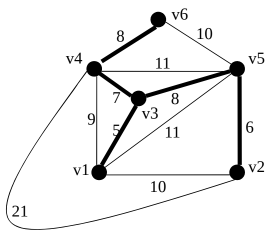
V1	1	0	-
V2	0	6	V5
V3	1	5	V1
V4	1	7	V3
V5	1	8	V3
V6	0	8	V4

Prim’s: Illustration



V1	1	0	-
V2	1	6	V5
V3	1	5	V1
V4	1	7	V3
V5	1	8	V3
V6	0	8	V4

Prim's: Illustration



V1	1	0	-
V2	1	6	V5
V3	1	5	V1
V4	1	7	V3
V5	1	8	V3
V6	1	8	V4

To Do

Modify Dijkstra's algorithm to get Prim's algorithm for Minimum Spanning Tree.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

Kruskal's Algorithm

Arrange all edges in increasing order of weight

Starting with the first edge (edge with minimum weight)

- Accept an edge into the tree if it doesn't lead to a cycle
- Else reject it
- Move on to the next edge
- Break when number of edges = Count(Vertices)-1

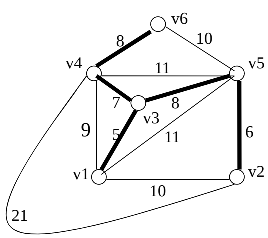
Note: Tree can be maintained as a graph in adjacency matrix form to check for cycles if required.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

Kruskal's Algorithm: Data

```
Struct EdgeInfo
{
    Int StartIndex;
    Int EndIndex;
    WeightType Weight;
    Int Status;
};
```

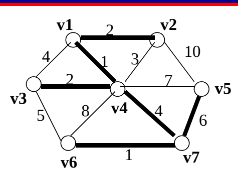

Kruskal's: Illustration



V1	V3	5	✓
V2	V5	6	✓
V3	V4	7	✓
V4	V6	8	✓
V3	V5	8	✓
V1	V4	9	
V1	V2	10	
V5	V6	10	
V1	V5	11	
V4	V5	11	
V2	V4	21	

Count of edges = 5
= Count (Vertices) –1
Hence: Done

Kruskal's: Illustration



V1	V4	1	✓
V6	V7	1	✓
V1	V2	2	✓
V3	V4	2	✓
V2	V4	3	✗
V1	V3	4	✗
V4	V7	4	✓
V3	V6	5	✗
V5	V7	6	✓
v4	V5	7	
v4	V6	8	
v2	V5	10	

Count of edges = 6
= Count (Vertices) –1
Hence: Done

CRITICAL PATH ANALYSIS

Critical Path Analysis

A common set of problems that are faced while planning allocating resources for projects:

- Estimating the earliest completion time for the project.
- Determining the activities that can be delayed (*and by how long*) without affecting the maximum completion time

The above said calculations can be made by modeling the project and its sub-activities as an acyclic graph

The technique is termed as ***critical path analysis.***

Activity-Node Graph (Activity Network)

A graph used to model various activities involved in a project.

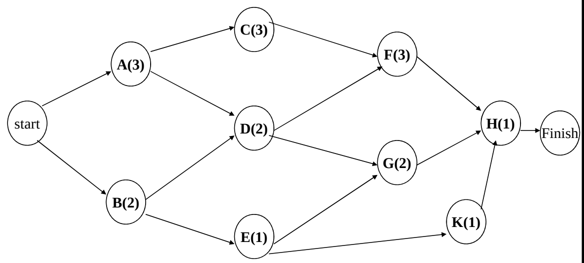
Each Node (Vertex) in the graph represents

- An **activity** that needs to be performed towards the completion of the project.
- The **time** required to complete the activity.

Each edge of the graph represents the relationship between two activities.

- An edge **(*v*, *w*)** means activity ***v*** must be **completed before** activity ***w*** begins

Illustration: AN Graph



An Activity-Node Graph

Event-Node Graph

To perform the calculations the ***Activity-Node*** graph is transformed to an ***Event-Node*** graph

In an Event-Node graph

- each node corresponds to completion of an activity and *all its dependent activities*.

As in activity-node graph, here also, the events reachable from a node ***v*** cannot commence until the even ***v*** is over.

Activity Node Graph to Event Node Graph

Rename Activity-Nodes to Event-Nodes.

Label Edges with the corresponding Event and the time required.

When an activity ***v*** depends on multiple activities,

- Introduce dummy edge/nodes that represent an intermediate state.
- This state signifies that all those activities whose completion is a pre-requisite for starting the activity ***v*** are over.

AN Graph to EN Graph

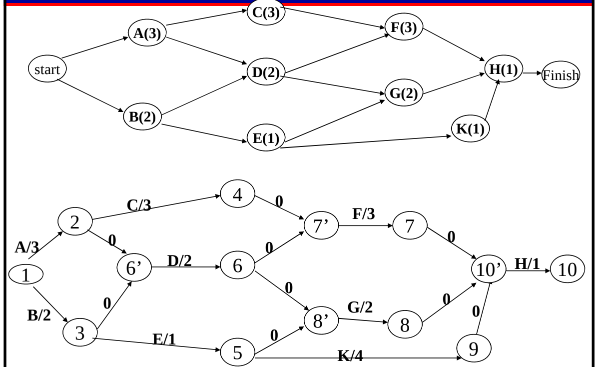
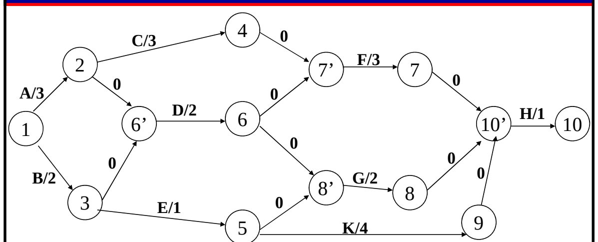


Illustration: EN Graph



- Here, n' is an event introduced prior to an event n that, in turn, depends on multiple events
- Time to reach n' from all the activities on which it depends is zero.

Earliest Completion Time

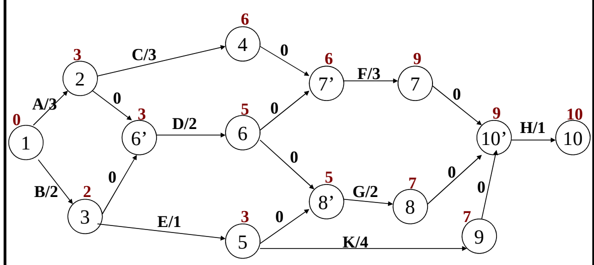
The ***earliest completion time of the project*** is given by

- the length of the longest path
- from the first event
- to the last event.

The ***earliest completion time for an activity*** is given by the relation:

- $EC_1 = 0$
- $EC_v = \max(EC_u + C_{u,v})$ (*For all the edges u,v existing in the graph*)

Illustration: EC Time



Latest Completion Time

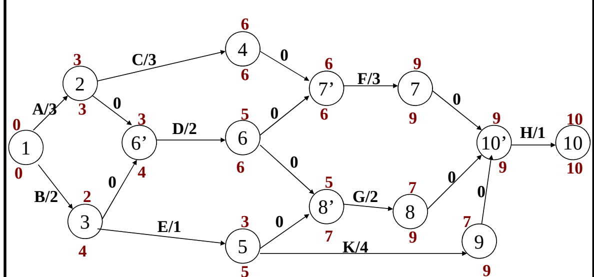
The late completion time for all the events is also an important factor that decides whether a project can be completed in time or not.

The late completion time ***for an activity*** is:

- The latest time that an event E_i can take to complete **without affecting the final completion** time
- The value is given by the relation:
  - ✓ $LC_n = Ec_n^*$
  - ✓ $LC_v = \min(LC_w - C_{v,w})$ (*For all the edges v,w existing in the graph*)

We want to finish the project earliest possible

Illustration: LC Time



Slack Time

Slack time for each edge in an event node graph implies

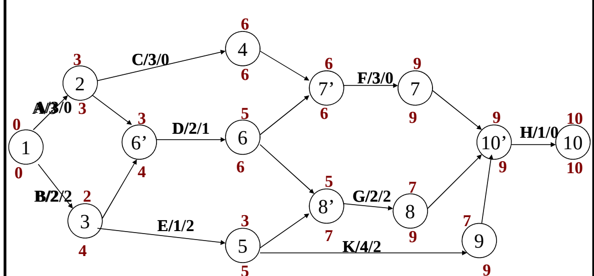
The amount of time

That the completion of the corresponding activity can be delayed **Without delaying the overall completion.**

Slack time for an activity is given by:

- $Slack_{(v,w)} = LC_w - (EC_v + C_{v,w})$

Illustration: Slack Time



Critical Path

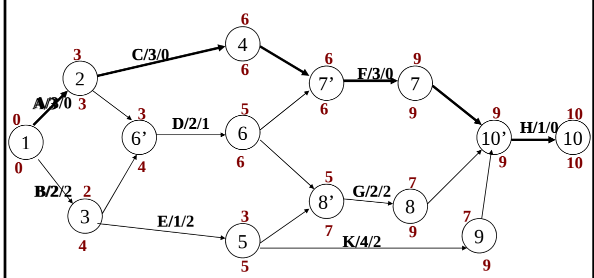
Critical Activities

- Activities with zero slack
- Must complete on time to ensure timely completion of project

Critical Path

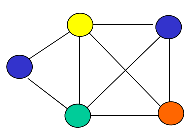
- Path on which all the activities are *zero-slack* activities.
- Thus implying that all the activities on the path must complete on their scheduled time
- As, there is no buffer period
- ***There will be at-least on such path on the graph***

Illustration: Critical Path



Coloring Graphs

Definition: A graph has been colored if a color has been assigned to each vertex in such a way that adjacent vertices have different colors.



Definition: The chromatic number of a graph is the smallest number of colors with which it can be colored.

In the example above, the chromatic number is 4.

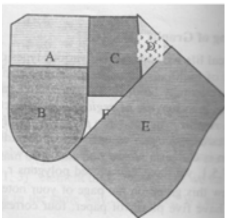
Coloring maps

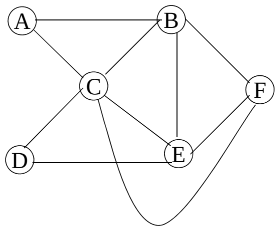
Color a map such that two regions with a common border are assigned different colors.

Each map can be represented by a graph:

- Each region of the map is represented by a vertex;
- Edges connect two vertices if the regions represented by these vertices have a common border.

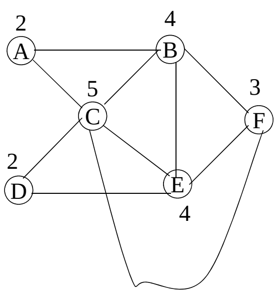
Example Map to Graph





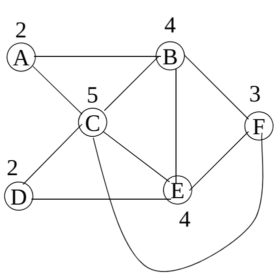
Illustration

Compute Degree



Illustration

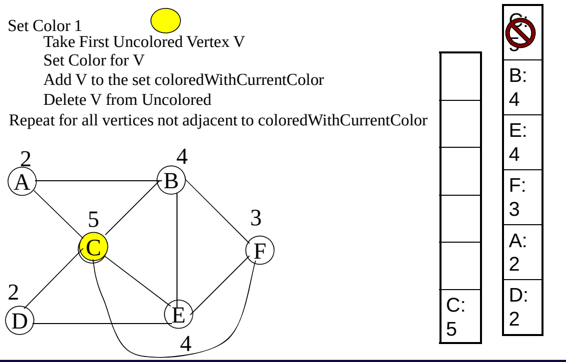
Set uncolored = V sorted in decreasing order of Degree(v).



C:5
B:4
E:4
F:3
A:2
D:2

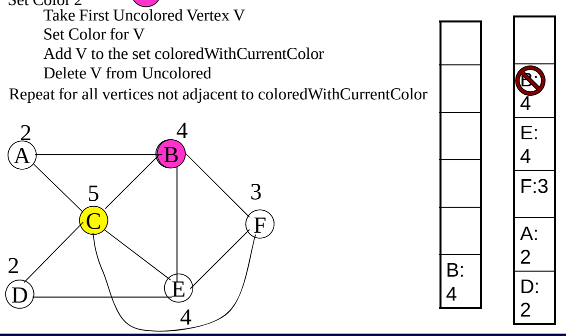
Illustration

Set Color 1
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor



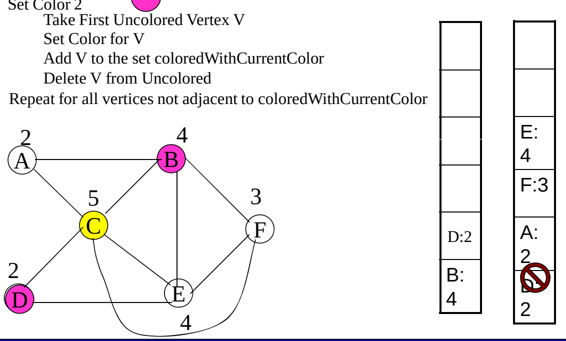
Illustration

Set Color 2
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor



Illustration

Set Color 2
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor



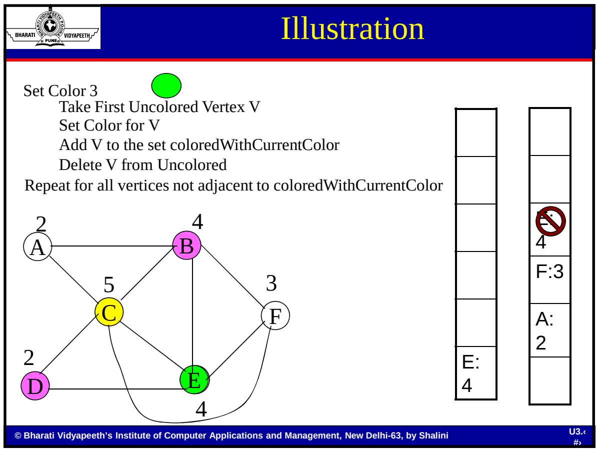


Illustration
Set Color 3
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor
2
A
4
B
5
C
3
F
2
D
E
4
E:
4
4
F:3
A:
2
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

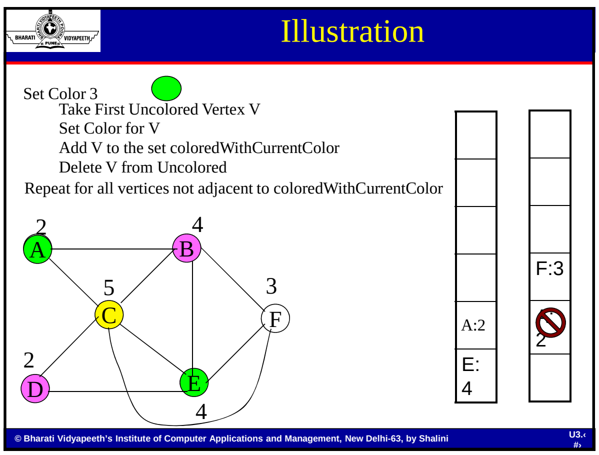


Illustration
Set Color 3
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor
2
A
4
B
5
C
3
F
2
D
E
4
A:2
E:
4
F:3
2
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

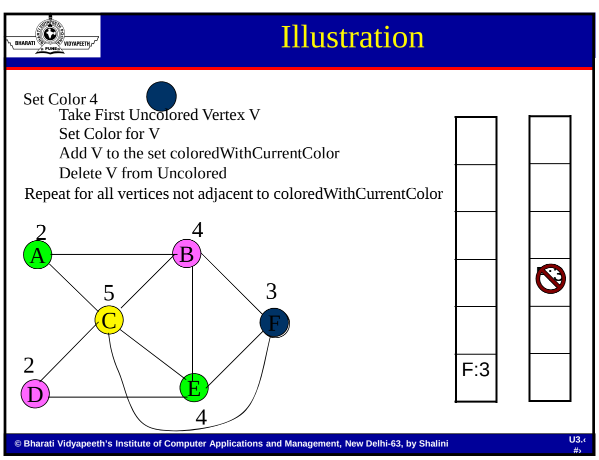


Illustration
Set Color 4
Take First Uncolored Vertex V
Set Color for V
Add V to the set coloredWithCurrentColor
Delete V from Uncolored
Repeat for all vertices not adjacent to coloredWithCurrentColor
2
A
4
B
5
C
3
F
2
D
E
4
F:3
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

Algorithm

Given G=(V,E)
Compute Degree(v) for all v in V
Set uncolored = V sorted in decreasing order of Degree(v).
set currentColor = 0.
while there are uncolored nodes
  set A=first element of uncolored
  remove A from uncolored
  set Color(A) = currentColor
  set coloredWithCurrentColor = {A}

Contd..

for each v in uncolored:
    if v is not adjacent to anything in coloredWithCurrentColor:
      set Color(v)=currentColor.
      add v to coloredWithCurrentColor.
      remove v from uncolored.
    end if
  end for
  currentColor = currentColor + 1.
end while

What we Studied

- ✓ Graphs
- ✓ Terminology
- ✓ Representation
- ✓ Basic operations
- ✓ DFS Traversal
- ✓ BFS Traversal
- ✓ Transitive Closure
- ✓ Warshell's Algorithm
- ✓ Topological Sort
- ✓ Shortest Path
- ✓ Critical Path Analysis
- ✓ Coloring Graph

What we Studied Cont..

- ✓Dijkstra's Algorithm
- ✓Minimum Spanning Tree
- ✓Prims' Algorithm
- ✓Kruskals' Algorithm
- ✓Critical Path Analysis
- ✓Graph Coloring

Learning Objectives

- **Internal Sorting Techniques**
- **External Sorting Techniques**

INTERNAL SORTING TECHNIQUES

Learning Objectives

- Sorting Techniques & Algorithm Analysis
  - Exchange sort
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
  - Heap Sort
  - Radixsort,

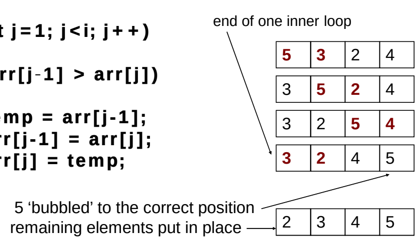
Sorting

- The objective is to take an unordered set of comparable data items and arrange them in order.
- We will usually sort the data into ascending order — sorting into descending order is very similar.
- Data can be sorted in various ADTs, such as arrays and trees.

Bubble sort

A pretty dreadful type of sort! However, the code is small:

```
for (int i=N; i>0; i--)
{
  for (int j=1; j<i; j++)
  {
    if (arr[j-1] > arr[j])
    {
      temp = arr[j-1];
      arr[j-1] = arr[j];
      arr[j] = temp;
    }
  }
}
```



Selection Sort

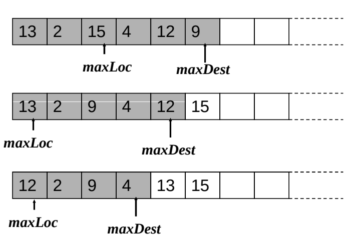
- Find the largest element in the array [0:N-1], place it at the location N-1
- Find the largest element in the array [0:N-2], place it at the location N-2
- And so on…
- The major disadvantage is the performance overhead of finding the largest element at each step

Selection Sort: algorithm

- Initialise ***maxDest*** to the last index of the Array
- Search from the start of the array to ***maxDest*** for the largest element: call its position ***maxLoc***
- Swap element indexed by ***maxLoc*** with element indexed by ***maxDest***
- Decrement ***maxDest*** by one
- Repeat steps 2 – 4

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

Selection Sort



etc.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini

Time complexity of Selection Sort

The main operations being performed in the algorithm are:

1. Comparisons to find the largest element in the subarray (to find ***maxLoc*** index): there are '$n + (n-1) + \ldots + 2 + 1$' comparisons, i.e., $n/2\ (n+1) = (n^2 + n) / 2$ so this is an $O(n^2)$ operation
2. Swapping elements between ***maxLoc*** and ***maxDest***: n-1 exchanges are performed so this is an O(n) operation

- The dominant operation (time-wise) gives the overall time complexity, i.e., $O(n^2)$
- Although this is an $O(n^2)$ algorithm, its advantage over $O(n \log n)$ sorts is its simplicity

Time complexity of Selection Sort

- For very small sets of data, SelectionSort may actually be more efficient than $O(n \log n)$ algorithms
- This is because many of the more complex sorts have a relatively high level of overhead associated with them, e.g., recursion is expensive compared with simple loop iteration
- This overhead might outweigh the gains provided by a more complex algorithm where a small number of data elements is being sorted
- SelectionSort does better than BubbleSort as fewer swaps are required, although the same number of comparison operations are performed (each swap puts an element in its correct place)

Insertion sort

Tree Insertion Sort

- This is inserting into a normal tree structure:
- i.e. data are put into the correct position when they are inserted.
- Requires a find and an insert.
- The time complexity for one insert is O(*logN*) + O(1) = O(*logN*);
- therefore to insert *N* items will have a complexity of O(*NlogN*).

Insertion sort

Array Insertion Sort

The array must be sorted; insertion requires a find + insert

.

To insert *N* items will have a complexity of **O(*N*²)**.

Array Insertion Sort

10	5	4	7	3	*Temp=5*
10	10	4	7	3	*A[0]= temp*

Array Insertion Sort

5	10	4	7	3	*Temp=4*
5	10	10	7	3	
5	5	10	7	3	*A[0]= temp*
4	5	10	7	3	

Array Insertion Sort

4	5	10	7	3	*Temp=7*

4	5	10	10	3	*A[2]= temp*

4	5	7	10	3	*Temp=3*

4	5	7	10	10

Array Insertion Sort

4	5	7	7	10

4	5	5	7	10

4	4	5	7	10	*A[0]= temp*

3	4	5	7	10

Array Insertion Sort: ALGO

```
FOR (P=1; P<N; P++)
   BEGIN
        temp= A[P];
        FOR (j=P; j>0 && A[j-1]>temp; j--)
                A[j]= A[j-1]
        A[j]= temp;
   END
END
```

Shell sort [diminishing increment sorting]

Named after D.L. Shell! But it is also rather like shrinking shells of sorting, for Insertion Sort.

Shell sort aims to reduce the work done by insertion sort (i.e. scanning a list and inserting into the right position).

Do the following:

- Begin by looking at the lists of elements x_1 elements apart and sort those elements by insertion sort
- Reduce the number x_1 to x_2

Shell Sort: Illustration

Gap=3

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

4	3	5	10	7	6	21	2	43	54

Shell Sort: Illustration

Gap =3

4	3	5	10	7	6	21	2	43	54

4	2	5	10	3	6	21	7	43	54

4	2	5	10	3	6	21	7	43	54

Continue for other values of *gap*…

Shell Sort: Algorithm

```
For Each Gap in GapValues
   For I=Gap to N-1
        Temp= A[I]
        For (J=I; J>=Gap && A[J-Gap]>Temp; J=J-Gap)
                A[J]= A[J-Gap]
        End For
        A[J]= Temp
   End For
End Form
```


How do you choose the gap size?

- The idea of the decreasing gap size is that the list becomes more and more sorted each time the gap size is reduced,
- Therefore (for example) having a gap size of 4 followed by a gap size of 2 is not a good idea, because you'll be sorting half the numbers a second time.
- There is no formal proof of a good initial gap size, but about a 10th the size of N is considered to be a reasonable start.
- Try to use prime numbers as gap size, or odd numbers *if a list of primes is not feasible to generate* (though note gaps of 9, 7, 5, 3, 1 will be doing less work when gap=3).

Shell Sort

Running time of Shell sort depends upon the gap sequence chosen.

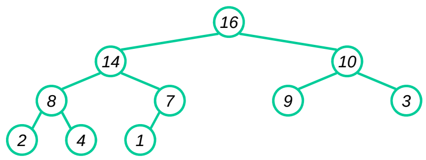
The set of gap values suggested by Shell, (N/2, N/4, …, 1) give a worst case running time of $O(N^2)$

Consider set

$\mathbf{1}_{\mathbf{0}}$, 9_1, $\mathbf{2}_{\mathbf{2}}$, 10_3, $\mathbf{3}_{\mathbf{4}}$, 11_5, $\mathbf{4}_{\mathbf{6}}$, 12_7,
$\mathbf{5}_{\mathbf{8}}$, 13_9, $\mathbf{6}_{\mathbf{10}}$, 14_{11}, $\mathbf{7}_{\mathbf{12}}$, 15_{13}, $\mathbf{8}_{\mathbf{14}}$, 16_{15}

Review: Heaps

A *heap* is a “complete” binary tree, usually represented as an array:



A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

Review: The Heap Property

Heaps also satisfy the *heap property*:

A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1

- In other words, the value of a node is at most (in max heap) the value of its parent
- The largest value is thus stored at the root (A[1])

Because the heap is a binary tree, the height of any node is at most Θ(lg *n*)

Review: Heapify()

`Heapify()`: maintain the heap property

- Given: a node *i* in the heap with children *l* and *r*
- Given: two subtrees rooted at *l* and *r*, assumed to be heaps
- Action: let the value of the parent node “float down” so subtree at *i* satisfies the heap property
  - ✓If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
  - ✓Recurse on that subtree
- Running time: O(*h*), h = height of heap = O(lg *n*)

Review: BuildHeap()

We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays

- Fact: for array of length *n*, all elements in range A[⌊n/2⌋ + 1 .. n] are heaps (*Why?*)
- So:
  - ✓Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
  - ✓Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Review: Heaps

To represent a heap as an array (base 1):

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```


Algorithm HeapSort

```
void HeapSort (int A[], int N)
{
   int i;

   for (i=N/2; i>=0; i--)
        ShiftDown (A, i, N);

   for (i=N-1; i>0; i--)
   {
        swap (&A[0], &A[i]);
        ShiftDown (A, 0, i);
   }
}
```

Algorithm ShiftDown

```
#define LeftChild(i) (2 * (i) +1)

void ShiftDown (int A[], int i, int N)
{
   int Child;
   int temp;

   for (temp=A[i]; LeftChild (i)<N; i=Child)
   {
        Child= LeftChild(i);
        if ( Child!=N-1 && A[Child+1]>A[Child] )
                Child++;
        if (temp < A[Child])
                A[i]= A[Child];
        else
                break;
   }
   A[i]= temp;
}
```

Radix Sort

Radix sort:

- Assumption: input has *d* digits ranging from 0 to *k*
- Basic idea:
  - ✓Sort elements by digit starting with *least* significant
- Each pass over *n* numbers with *d* digits takes time O(*n*+*k*), so total time O(*dn*+*dk*)
  - ✓When *d* is constant and *k*=O(*n*), takes O(*n*) time
- **Fast! Stable! Simple!**
- Doesn't sort in place

Example

Queue X ={25, 29, 38, 36, 32, 54, 44, 40}

	FRONT	REAR
Q[0]	40	
Q[1]		
Q[2]	32	
Q[3]		
Q[4]	54	44
Q[5]	25	
Q[6]	36	
Q[7]		
Q[8]	38	
Q[9]	29	

AFTER FIRST PASS

Queue X ={40, 32, 54, 44, 25, 36, 38, 29}

Cont..

Queue X ={40, 32, 54, 44, 25, 36, 38, 29}

	FRONT		REAR
Q[0]			
Q[1]			
Q[2]	25		29
Q[3]	32	36	38
Q[4]	40		44
Q[5]	54		
Q[6]			
Q[7]			
Q[8]			
Q[9]			

AFTER SECOND PASS

Queue X ={25, 29, 32, 36, 38, 40, 44, 54}

Algorithm Radix Sort

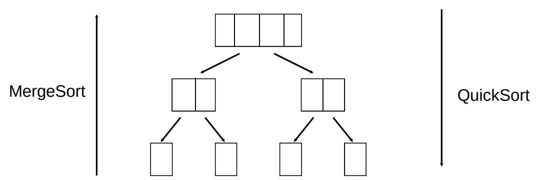
```
For d = 1 to number of digits do
    For i =0 to n-1 do
        d = Kth digit of x[i]
        Place x[i] at rear of Q[d]
    For k = 0 to 9 do
        Delete all elements of Q[k] and place in Queue X
end
```


Review: Radix Sort

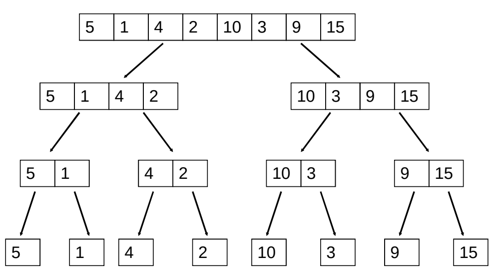
Radix sort:

- Assumption: input has d digits ranging from 0 to k
- Basic idea:
  - ✓Sort elements by digit starting with *least* significant
  - ✓Use a stable sort (like counting sort) for each stage
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - ✓When d is constant and $k=O(n)$, takes $O(n)$ time
- **Fast! Stable! Simple!**
- Doesn't sort in place

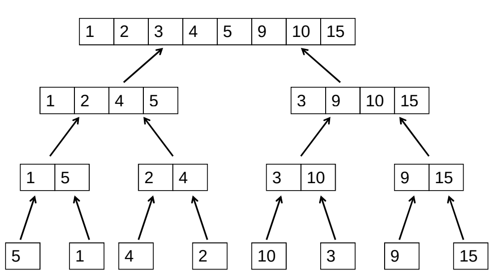
Divide and conquer sorting



Divide ...



and conquer



Merge Sort

For MergeSort an initial array is repeatedly divided into halves, until arrays of just one or zero elements remain.

At each level of recombination, two sorted arrays are merged into one.

This is done by copying the smaller of the two elements from the sorted arrays into the new array, and then moving along the arrays.

1	13	24	26	2	15	27	38						.....

Aptr Bptr Cptr

Merging

1	13	24	26	2	15	27	38	1					.....

Aptr Bptr Cptr

1	13	24	26	2	15	27	38	1	2				.....

Aptr Bptr Cptr

1	13	24	26	2	15	27	38	1	2	13			.....

Aptr Bptr Cptr

etc.

Merge Sort

```
MergeSort(A, tmpA, left, right) {
  if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, tmpA, left, mid);
      MergeSort(A, tmpA, mid+1, right);
      Merge(A, tmpA, left, mid+1, right);
  }
}
// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// It requires O(n)
// time, and *does* require allocating O(n) space
```

Merge Function

```
void Merge(A, tmpA,  LeftPos, RightPos, RightEnd)
{
LeftEnd = RightPos-1;
TmpPos=LeftPos;
NoElements= RightEnd-LeftPos+1;

While(LeftPos<= LeftEnd && RightPos <=RightEnd)
   If(A[LeftPos] <= A[RightPos])
        tmpA[TmpPos++] = A[LeftPos++];
   else
        tmpA[TmpPos++] = A[RightPos++];
```


Merge Function Contd..

```
    While(LeftPos <= LeftEnd)
      tmpA[TmpPos++] = A[LeftPos++];

    While(RightPos <= RightEnd)
      tmpA[TmpPos++] = A[RightPos++];

    For(i=0; i<NoElements; i++, RightEnd--)
      A[RightEnd]=tempA[RightEnd]
 }
```


QuickSort

As its name implies, QuickSort is the fastest known sorting algorithm *in practice*

It was devised by C.A.R. Hoare in 1962

Its average running time is **O(*n* log *n*)** and it is very fast

It has worst-case performance of **O(n^2)** but this can be made very unlikely with little effort

QuickSort

The idea is as follows:

1. **If the number of elements to be sorted is 0 or 1, then return**
2. **Pick any element, *v* (this is called the *pivot*)**
3. **Partition the other elements into two disjoint sets, S_1 of elements $\leq v$, and S_2 of elements $> v$**
4. **Return QuickSort (S_1) followed by *v* followed by QuickSort (S_2)**

Review: Quicksort

Another divide-and-conquer algorithm

- **The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]**
  - ✓**Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]**
- **The subarrays are recursively sorted by calls to quicksort**
- **Unlike merge sort, no combining step: two subarrays form an already-sorted array**

Review: Partition

Clearly, all the action takes place in the **`partition()`** function

- Rearranges the subarray **in place**
- End result:
  - ✓Two subarrays
  - ✓All values in first subarray ≤ all values in second
- Returns the index of the "pivot" element separating the two subarrays

QuickSort example

5	1	4	2	10	3	9	15	12

Pick the middle element as the pivot, i.e., 10

Partition into the two subsets below

5	1	4	2	3	9

15	12

Sort the subsets

1	2	3	4	5	9

12	15

Recombine with the pivot

1	2	3	4	5	9	10	12	15

Partitioning example

5	11	4	25	10	3	9	15	12

Pick the middle element as the pivot, i.e., 10

Move the pivot out of the way by swapping it with the first element

10	11	**4**	25	5	3	9	15	12

swapPos

Step along the array, swapping small elements into swapPos

10	**4**	**11**	25	**5**	3	9	15	12

swapPos

Partitioning example (2)

10	4	**5**	25	**11**	**3**	9	15	12

swapPos

10	4	5	**3**	11	**25**	9	15	12

swapPos

10	4	5	3	**9**	25	11	15	12

swapPos

9	4	5	3	**10**	25	11	15	12

Partition

Quicksort Code

```
Quicksort(A, first, last)
{
    if (first < last)
    {
        p = Partition(A, first, last);
        Quicksort(A, first, p-1);
        Quicksort(A, p+1, last);
    }
}
```


Pseudo code for partitioning

```
Partition(A, first, last) {
    pivotPos = (first + last) /2;
    swap a[pivotPos] with a[first];   // Move pivot out of the way
    swapPos = curr= first + 1;
    while a[curr] >= a[first] curr++;
    while curr < last
       // If the a[curr] < pivot we move it towards start of array
       if (a[curr] < a[first]):
          swap a[swapPos++] with a[curr];
       curr++;
    // Now move the pivot back to its rightful place
    if swapPos>first+1 swap a[first] with a[swapPos-1];
    return swapPos-1;   // Pivot position
}
```

Some observations about QuickSort

- A consistently poor choice of pivot can lead to $O(n^2)$ time performance
- A good strategy is to pick the middle value of the left, centre, and right elements
- For small arrays, with n less than (say) 20, QuickSort does not perform as well as simpler sorts such as SelectionSort
- Because QuickSort is recursive, these small cases will occur frequently
- A common solution is to stop the recursion at $n = 10$, say, and use a different, non-recursive sort
- This also avoids nasty special cases, e.g., trying to take the middle of three elements when n is one or two

What we studied

- ✓ Bubble Sort
- ✓ Selection Sort
- ✓ Insertion Sort
- ✓ Shell Sort
- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Radix Sort
- ✓ Heap Sort

Hashing

Hashing

Technique for performing Insertion, Deletion, Search in constant average time

Ordering of elements is not supported efficiently

Keys are mapped onto a number between 0 & TableSize-1

Mapping is done on basis of a function called *Hash Function*

Hash Function

Transforms a key into a cell/bucket address

Must be simple to compute

Should ensure that distinct keys get distinct cells
- Not possible in all cases as number of keys increases
- Leads to collisions (multiple keys map to the same hash value)

So choose a function that leads to even distribution of keys

Considerations

Which hash function to use

How to respond to collisions

Common hash functions

Mod
Mid Square
Folding
Digit Analysis

Division Method

H(K) = K mod M or
H(K) = (K mod M)+1

M = 10
K = 12, 30, 11, 12, 34

0	30
1	11
2	12
3	
4	34
5	
6	
7	
8	
9	

Mid Square Method

- Square Key
- Consider some mid value

K	: 3205	7148	2345
K^2	: 102**72**025	510**93**904	54**99**025
H(K)	: 72	93	99

Folding Method

- When Key is Large
- Partition Key into K_1, K_2……K_n
  - Shift Folding
  - Boundary Folding

Shift Folding Method

K = 768932834567

K_1 = 768

K_2 = 932

K_3 = 834

K_4 = 567

3101

$H(K) = K_1 + K_2 + \ldots + K_n$

Reduce the result into two digits if require

Boundary Folding Method

K = 768932834567

K_1 = 768

K_2 = **239 <- Reversed**

K_3 = 834

K_4 = 567

2408

$H(K) = K_1 + K_2 + \ldots + K_n$

Reduce the result into two digits if require

Digital Analysis

Select and shift digits

K = 75**4612**3 transformed to

2164

OR

K = 7**54612**39 transformed to

9265

Reduce the result into two digits if require

Collision Resolution

Open address hashing
- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining

Rehashing

Open Addressing

In case of a collision alternate cells are tried till an empty cell is not found.

Cell $h_i(X)$= Hash(X) + F(i)
- Given F(0)=0

For Linear probing F(i) is a linear function of *i* ;

For Quadratic probing F(i) is a function of i^2;

For Double Hashing F(i) is some function of I other than the one chosen originally.

Linear Probing

For a table large enough in size to hold all the keys; free space will always be found
- Though the time required will be large

Drawback
- Blocks of occupied cells might get formed: PRIMARY CLUSTERING
- i.e a key that hashes into a cluster will require several attempts to resolve collision

Linear Probing

Consider a hash table with 10 slots.

Say,

- The keys to be inserted are 12, 30, 11, 32, 34, 54, 50
- The hash function is ***mod 10***
- This divisor is chosen just for illustration and is not a good choice
  - ✓as a maximum of 10 resultant cells get generated, thus collisions will be frequent.
  - ✓The divisor should preferably be a prime number

Stages of insertion are illustrated on following slides

Linear Probing: Illustration

Add 30 on Cell 30%10= 0

Add 11 on Cell 11%10= 1

Add 12 on Cell 12%10= 2

Try to Add 32 on Cell 32%10= 2; Not available; Try Next

Add 34 on Cell 34%10= 4

Try to Add 54 on Cell 54%10= 4; Not available; Try Next

Try to Add 50 on Cell 50%10= 0; Not available; Try Next… Till an empty cell isn't found

30	0
11	1
12	2
32	3
34	4
54	5
50	6
	7
	8
	9

Quadratic Probing

Similar treatment can be given when collisions occur in case of Quadratic probing;

Here,

- instead of choosing the next cell that lies after the ideal cell ***i*** (or a cell given by a linear function of ***i***)
- A new cell number given by some quadratic function of ***i*** is chosen

Separate Chaining

Maintains a list of all the keys that hash to the same value

To insert:

- Calculate the hash function
- Access the corresponding list
- Add a link to the list

i.e. A link is added in case of a collision

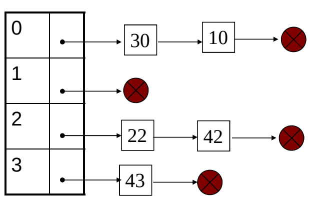
The new key might be added at either end of the list

Better for large sized records, handles collisions & overflow efficiently.

Not as efficient when record size is small or domain of keys values is limited to a small number of entries

Separate Chaining: Illustration

Insert Sequence: 22, 42, 30, 43, 10



Rehashing

When table gets *Too full,*

- number of collisions increase;
- thus, resulting in a degradation in performance while inserting as well as searching

Build another hash table with size ~ 2*OldSize

Scan the original table; for each entry

- Compute the new hash value
- Insert in the new hash table

Rehashing is costly, thus, should not be done very frequently.

Rehashing: Illustration

Consider the hash table as given in the figure:

–The keys to be inserted are 12, 30, 11, 32, 34, 54, 50

–The hash function is ***mod 10***

30	0
11	1
12	2
32	3
34	4
54	5
50	6
	7
	8
	9

Rehashing

- New table size 19
- The hash function is ***mod 23***

						50	30	54	32		11	12	34					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Conclusion

✓ Searching Techniques

✓ Internal Sorting Techniques

✓ Hashing

Review Questions (Objective)

1. A list is ordered from smaller to largest when a sort is called. Which sort would take the longest time to execute?
2. A list is ordered from smaller to largest when a sort is called. Which sort would take the shortest time to execute?
3. When will you sort an array of pointers to list elements, rather than sorting the elements themselves?
4. The element being searched for is not found in an array of 100 elements. What is the average number of comparisons needed in a sequential search to determine that the element is not there, if the elements are completely unordered?
5. What is the average number of comparisons needed in a sequential search to determine the position of an element in an array of 100 elements, if the elements are ordered from largest to smallest?
6. Which sort show the best average behavior?
7. What is the average number of comparisons in a sequential search?
8. Which one is faster? A binary search of an orderd set of elements in an array or a sequential search of the elements.
9. Under what circumstances would you not use a quick sort.
10. Define Heap sort with example.
11. Running time of merge sort algrothim is...........
12. Define radix sort.

Review Questions (Short Type)

1. Compare the time complexity of various sorting algrothims (Best and Worst case).
2. Write an algrothim (non-recursive) to implement quick sort.
3. What do u mean by insertion sort.Which data structure is best suited for inserted sort and why.Explain the algorithm for insertion sort.
4. What do u mean by searching.What are the condition for binary search.Explain the algorithm for the binary search.
5. Write a Binary Search program.
6. Give the difference between linear and binary search.
7. Define merge sort .give its algorithm.
8. Write programs for Bubble Sort, Quick sort .
9. List out few of the Application of tree data structure.
10. Define selection sort with example.
11. What is heap sort? Explain with example.
12. Explain binary search and linear search.
13. Explain merge sort with example.
14. Give the complexity of all sorting algorithms.
15. Give the complexity of binary search algorithm .explain it
16. Give the limitations of binary search algorithm.
17. Under what circumstances would you not use a quick sort.
18. Sort the given values using Quick Sort?
    65 70 75 80 85 60 55 50 45

Review Questions (Long Type)

1. Explain quick sort and merge sort algorithms and derive the time-constraint relation for these.
2. Write algorithm for any of the following sorting methods: -
   - Merge Sort
   - Quick Sort
   - Radix Sort
3. Compare above three methods of sorting for ideal, worst and average cases.
4. Define bubble sort. Give the algorithm and explain it with example. Suppose the following numbers are stored in an array A.
   32, 51, 27, 85 , 66, 23, 13, 57.
   Sort the array using bubble sort.
5. What do u mean by hashing? What are various hash function? Explain three hash function.
6. What do u mean by searching? What are various searching techniques? Which searching technique u like the most and why?.Give an algorithm for the searching technique.
7. Write an algorithm for radix sort.
8. Define linked list..Give the algorithms when list is sorted and unsorted.
9. Describe K-Way Merge External Sort .

References

- E. Horowitz and S. Sahani, “Fundamentals of Data Structures in C”, 2nd Edition, Universities Press, 2008.
- Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition Addison-Wesley, 1997.
- Schaum’s Outline Series, “Data Structure”, TMH, Special Indian Ed., Seventeenth Reprint, 2009.
- Y. Langsam et. al., “Data Structures using C and C++”, PHI, 1999.
- N. Dale and S.C. Lilly, D.C. Heath and Co., “Data Structures”, 1995.
- R. S. Salaria, Khanna, “Data Structure & Algorithms”, Book Publishing Co. (P) Ltd., 2002.
- Richard F. Gilberg and Behrouz A. Forouzan, “Data Structure A Pseudocode Approach with C”, Cengage Learning, 2nd Ed., 2005.
- Mary E. S. Loomes, “Data Management and File Structure”, PHI, 2nd Ed., 1989.
- http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/radixSort/radix.html

 U3.‹#›