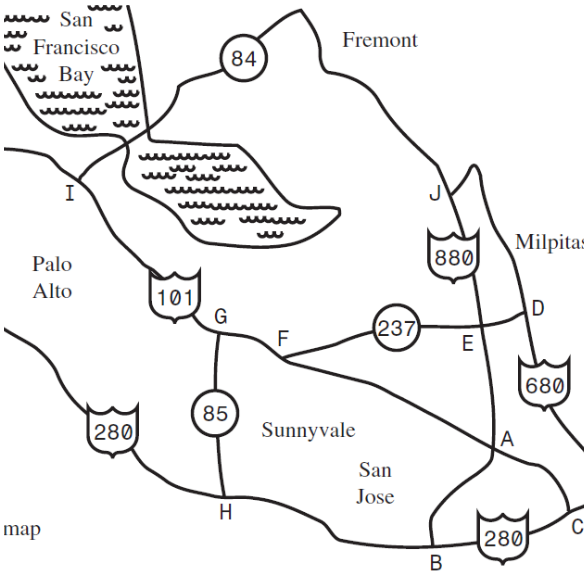
DATA STRUCTURES

WENYE LI
CUHK-SZ

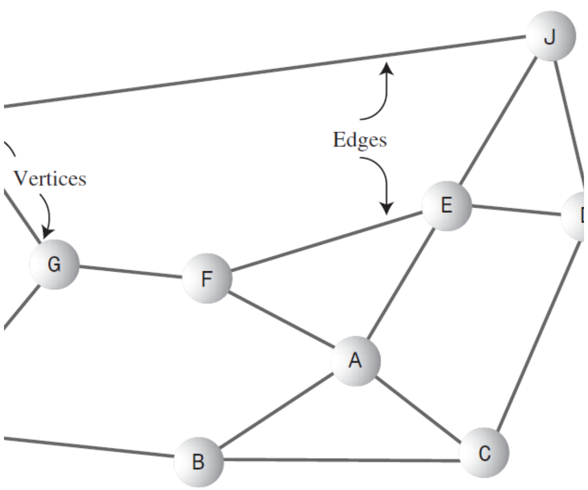
OUTLINE

- Concepts
- Implementations
- Examples

GRAPH



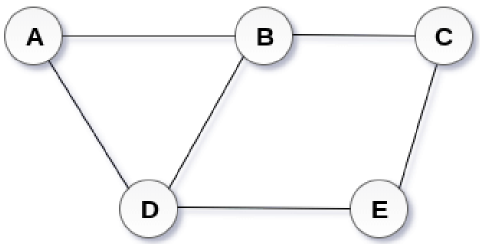
map



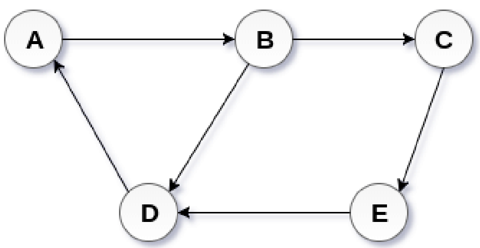
ph

- A graph: a group of vertices and edges that are used to connect these vertices.
- A graph can be seen as a cyclic tree, where the vertices (nodes) maintain any complex relationship among them instead of having parent child relationship.
- Definition: A graph G can be defined as an ordered set G(V, E)
  - V(G) represents the set of vertices
  - E(G) represents the set of edges which are used to connect these vertices

UNDIRECTED VS DIRECTED



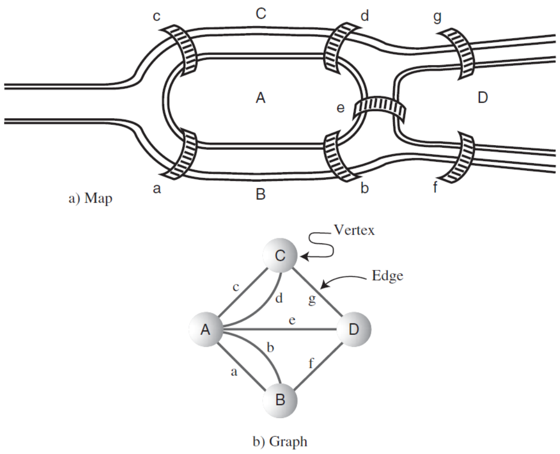
Undirected Graph



Directed Graph

- In an undirected graph, edges are not associated with the directions with them.
  - If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.
- In a directed graph, edges form an ordered pair.
  - Edges represent a specific path from some vertex A to another vertex B.
  - Node A is called initial node while node B is called terminal node.

TERMINOLOGY



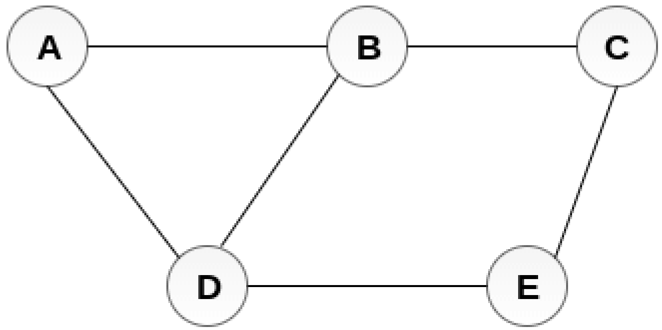
- Path: a sequence of edges connecting initial node V_0 to terminal node V_N.
- Closed Path: A path where the initial node is same as terminal node, ie, $V_0=V_N$.
- Simple Path: all the nodes of the path are distinct, with the exception $V_0=V_N$.
- Closed Simple Path: a simple path with $V_0=V_N$.
- Cycle: a path which has no repeated edges or vertices except the first and last vertices.
- Adjacent Nodes: two nodes u and v are connected via an edge e.
  - the nodes u and v are also called as neighbours.
- Degree of a Node: the number of edges that are connected with the node.
  - A node with degree 0 is called as isolated node.

TERMINOLOGY

- Connected Graph: a graph in which a path exists between every two vertices (u, v) in V.
  - There are no isolated nodes in connected graph.
- Complete Graph: a graph in which there is an edge between each pair of vertices.
  - A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.
- Weighted Graph: each edge is assigned with some data such as length or weight.
  - The weight of an edge e, w(e), must be positive indicating the cost of traversing the edge.
- Digraph: each edge of the graph is associated with some direction.
  - The traversing can be done only in the specified direction.

SEQUENTIAL REPRESENTATION

- Use adjacency matrix to store the mapping represented by vertices and edges.
- In adjacency matrix, the rows and columns are represented by the graph vertices.
- For a graph having n vertices, the adjacency matrix will have a dimension **n x n**.



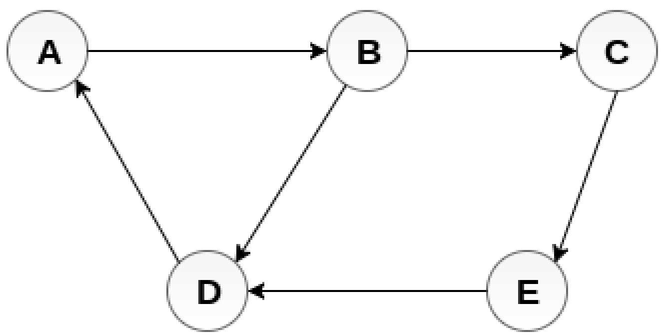
Undirected Graph

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

Adjacency Matrix

SEQUENTIAL REPRESENTATION

- Undirected: an entry M_{ij} in the adjacency matrix will be 1 if there exists an edge between V_i and V_j.



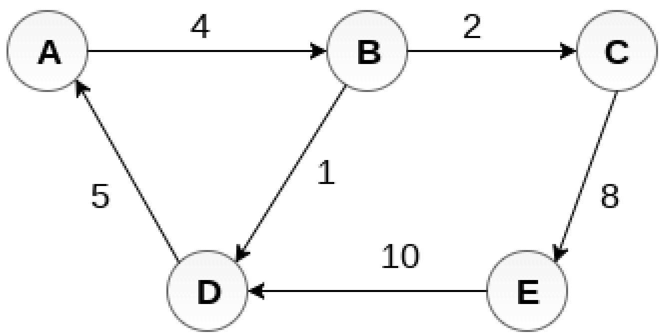
Directed Graph

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

Adjacency Matrix

SEQUENTIAL REPRESENTATION

- Directed: an entry A_{ij} in the adjacency matrix will be 1 if there exists an edge directly from V_i to V_j.



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

SEQUENTIAL REPRESENTATION

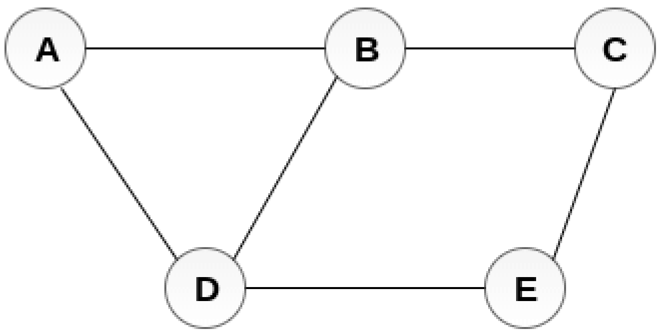
- Weighted Directed: non-zero entries of the adjacency matrix are represented by the weight of respective edges.

```java
class Vertex
{
    public char label; // label (e.g. 'A')
    public boolean wasVisited;
    public Vertex(char lab) // constructor
    {
        label = lab;
        wasVisited = false;
    }
} // end class Vertex
```

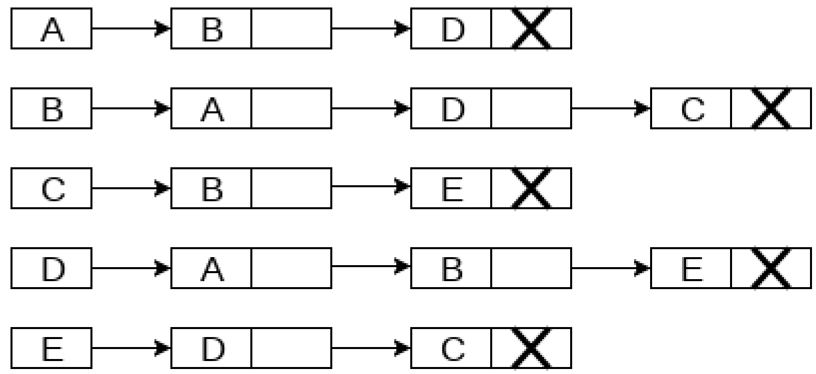
```java
class Graph
{
    private final int MAX_VERTS = 20;
    private Vertex vertexList[]; // array of vertices
    private int adjMat[][]; // adjacency matrix
    private int nVerts; // current number of vertices
    // -----------------------------------------------------------
    public Graph() // constructor
    {
        vertexList = new Vertex[MAX_VERTS];
        // adjacency matrix
        adjMat = new int[MAX_VERTS][MAX_VERTS];
        622 CHAPTER 13 Graphs
        nVerts = 0;
        for(int j = 0; j < MAX_VERTS; j++) // set adjacency
            for(int k = 0; k < MAX_VERTS; k++) // matrix to 0
                adjMat[j][k] = 0;
    } // end constructor
    // -----------------------------------------------------------
    public void addVertex(char lab) // argument is label
    {
        vertexList[nVerts++] = new Vertex(lab);
    }
    // -----------------------------------------------------------
    public void addEdge(int start, int end)
    {
        adjMat[start][end] = 1;
        adjMat[end][start] = 1;
    }
    // -----------------------------------------------------------
    public void displayVertex(int v)
    {
        System.out.print(vertexList[v].label);
    }
    // -----------------------------------------------------------
} // end class Graph
```

LINKED REPRESENTATION

- An adjacency list is used to store the Graph into the computer's memory.
- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.
- If all the adjacent nodes are traversed, then store the NULL in the pointer field of last node of the list.



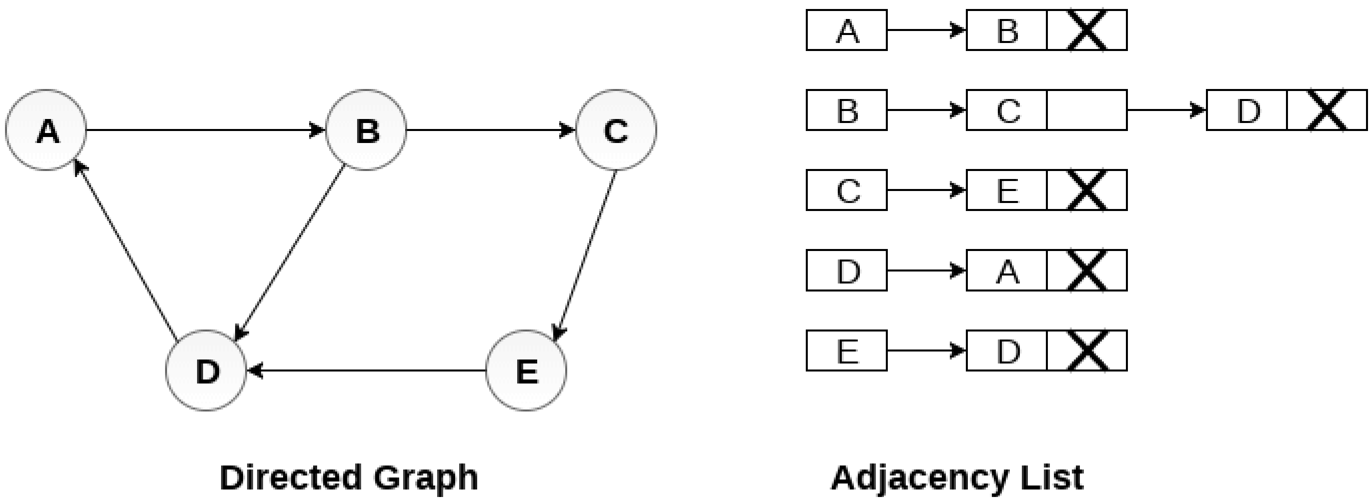
Undirected Graph



Adjacency List

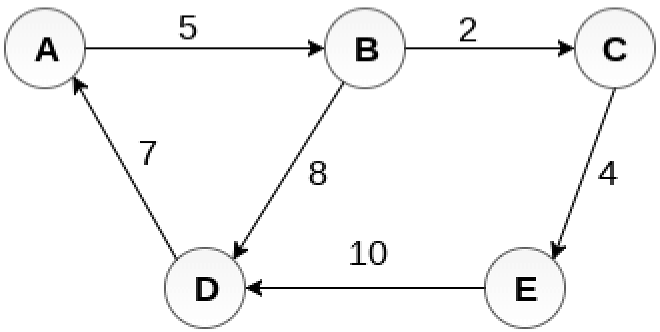
LINKED REPRESENTATION

- Undirected: The sum of the lengths of adjacency lists is equal to the twice of the number of edges.

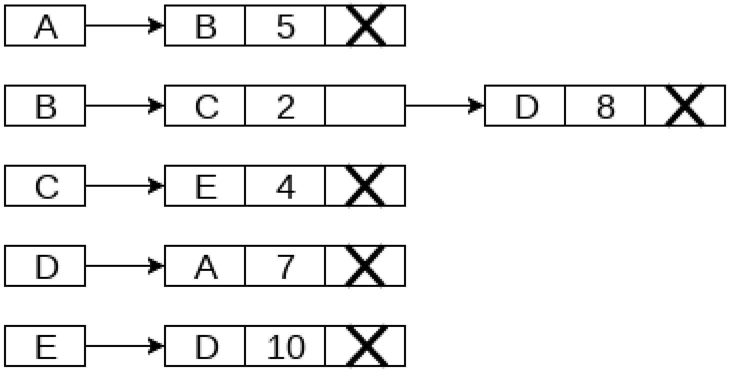


LINKED REPRESENTATION

- Directed: The sum of the lengths of adjacency lists is equal to the number of edges.



Weighted Directed Graph



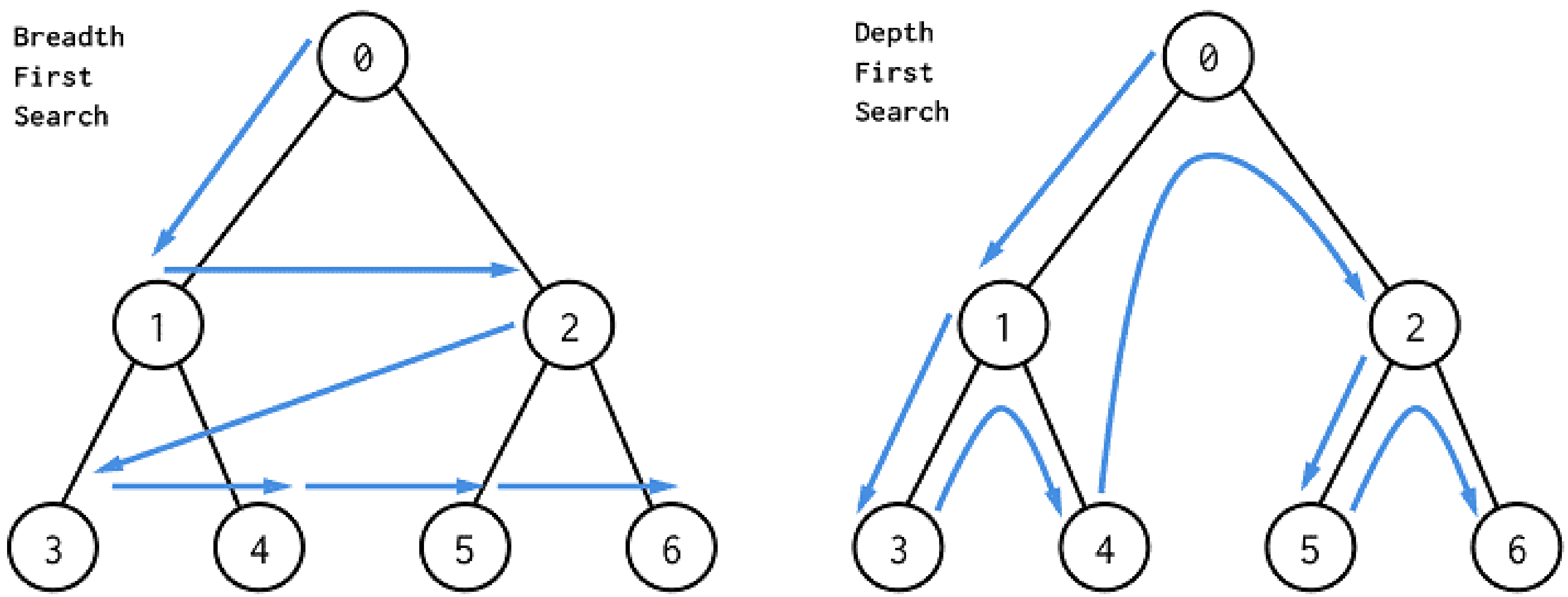
Adjacency List

LINKED REPRESENTATION

- Weighted Directed: each node contains an extra field, called the weight of the node.

OUTLINE

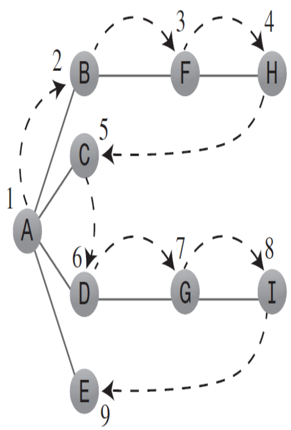
- Concepts
- Implementations
- Examples



GRAPH TRAVERSAL

- Traversing the graph means examining all the nodes and vertices of the graph
- Two standard methods to traverse graphs
  - Breadth First Search
  - Depth First Search

DEPTH FIRST SEARCH



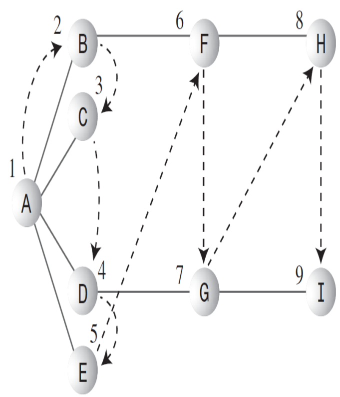
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

- Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.
- Rule2 : If you can't follow Rule 1, then, if possible, pop a vertex off the stack.
- Rule 3: If you can't follow Rule 1 or Rule 2, you're done.

```java
// returns an unvisited vertex adjacent to v
public int getAdjUnvisitedVertex(int v)
{
    for(int j = 0; j < nVerts; j++)
        if(adjMat[v][j] == 1 && vertexList[j].wasVisited == false)
            return j; // return first such vertex
    return -1; // no such vertices
} // end getAdjUnvisitedVertex()

public void dfs() // depth-first search
{
    // begin at vertex 0
    vertexList[0].wasVisited = true; // mark it
    displayVertex(0); // display it
    theStack.push(0); // push it
    while( !theStack.isEmpty() ) // until stack empty,
    {
        // get an unvisited vertex adjacent to stack top
        int v = getAdjUnvisitedVertex( theStack.peek() );
        if(v == -1) // if no such vertex,
            theStack.pop(); // pop a new one
        Searches 629
        else // if it exists,
        {
            vertexList[v].wasVisited = true; // mark it
            displayVertex(v); // display it
            theStack.push(v); // push it
        }
    } // end while
    // stack is empty, so we’re done
    for(int j = 0; j < nVerts; j++) // reset flags
        vertexList[j].wasVisited = false;
} // end dfs
```

BREADTH FIRST SEARCH

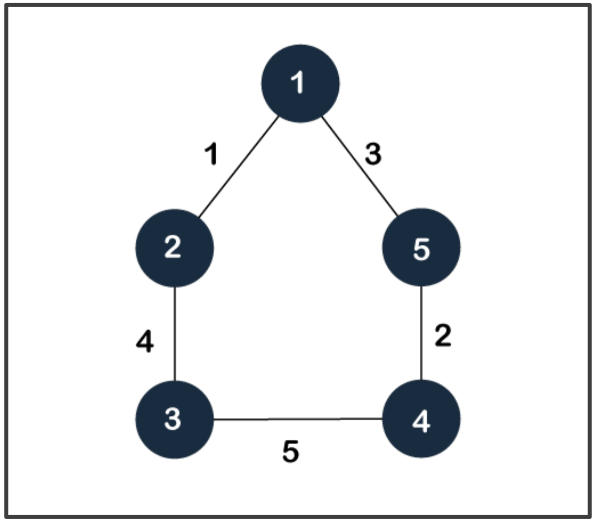


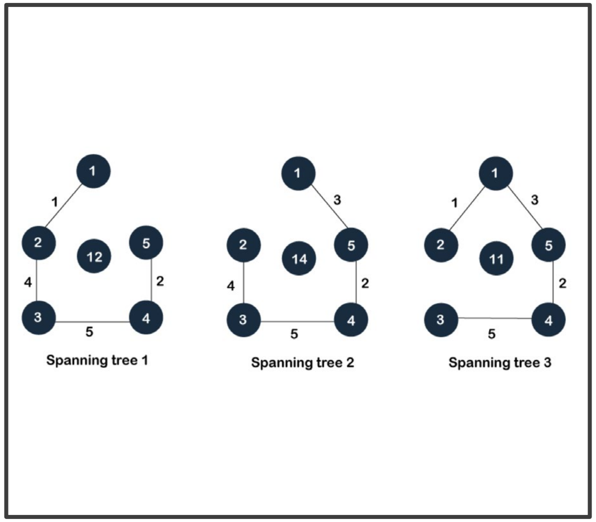
Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

- Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.
- Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.
- Rule 3: If you can't carry out Rule 2 because the queue is empty, you're done.

```java
public void bfs() // breadth-first search
{
    // begin at vertex 0
    vertexList[0].wasVisited = true; // mark it
    displayVertex(0); // display it
    theQueue.insert(0); // insert at tail
    int v2;
    while( !theQueue.isEmpty() ) // until queue empty,
    {
        int v1 = theQueue.remove(); // remove vertex at head
        // until it has no unvisited neighbors
        while( (v2 = getAdjUnvisitedVertex(v1)) != -1 )
        {
            // get one,
            vertexList[v2].wasVisited = true; // mark it
            displayVertex(v2); // display it
            theQueue.insert(v2); // insert it
        } // end while(unvisited neighbors)
    } // end while(queue not empty)
    // queue is empty, so we're done
    for(int j = 0; j < nVerts; j++) // reset flags
        vertexList[j].wasVisited = false;
} // end bfs()
```

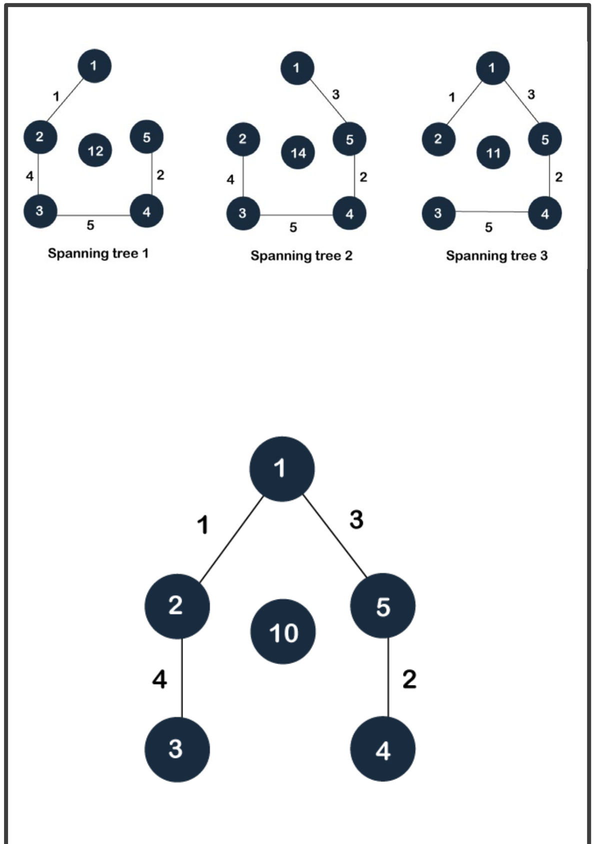
SPANNING TREE





- Idea: for any connected set of vertices and edges, remove any extra edges
- For a graph G=(V, E), its spanning tree
  - have the same number of vertices as the graph.
  - the edges = the number of vertices in the graph - 1.
- Represent the spanning tree by $G' = (V', E')$
  - $V' = V$
  - $E' \subseteq E$
  - $|E'| = |V| - 1$

MINIMUM SPANNING TREE



- the total edge weight of the spanning tree 1 is 12
- the total edge weight of the spanning tree 2 is 14
- the total edge weight of the spanning tree 3 is 11

- Minimum spanning tree: the spanning tree whose sum of edge weights is minimum.
  - The minimum spanning tree has a weight of 10.

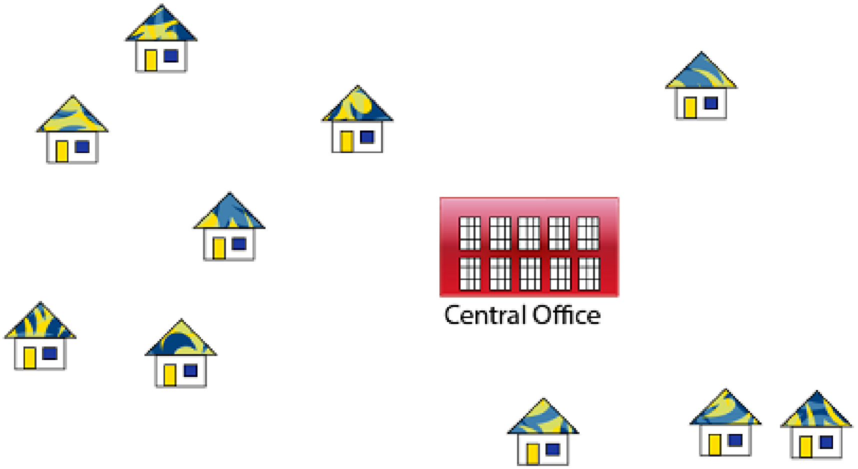
MINIMUM SPANNING TREE

- MST can be obtained with a slight modification of depth-first search, or breadth-first search.

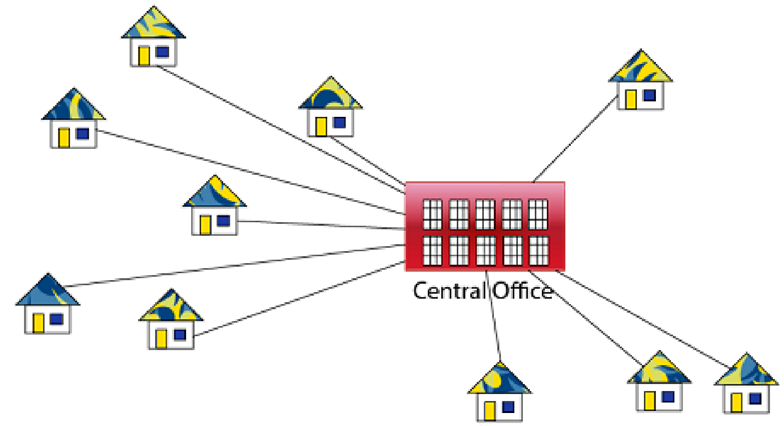
```java
public void mst() // minimum spanning tree (depth first)
{
    // start at 0
    vertexList[0].wasVisited = true; // mark it
    theStack.push(0); // push it
    while( !theStack.isEmpty() ) // until stack empty
    {
        // get stack top
        int currentVertex = theStack.peek();
        // get next unvisited neighbor
        int v = getAdjUnvisitedVertex(currentVertex);
        if(v == -1) // if no more neighbors
            theStack.pop(); // pop it away
        else // got a neighbor
        {
            vertexList[v].wasVisited = true; // mark it
            theStack.push(v); // push it
            // display edge
            displayVertex(currentVertex); // from currentV
            displayVertex(v); // to v
            System.out.print(“ “);
        }
    } // end while(stack not empty)
    // stack is empty, so we’re done
    for(int j = 0; j < nVerts; j++) // reset flags
        vertexList[j].wasVisited = false;
} // end tree
```

PROPERTIES OF SPANNING TREE

- A connected graph can contain more than one spanning tree.
- All the possible spanning trees that can be created from the given graph G have the same number of vertices.
- The number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle.
- The spanning tree cannot be disconnected.
- If two or three edges have the same edge weight, then there would be more than two minimum spanning trees.
- If each edge has a distinct weight, then there will be only one or unique minimum spanning tree.
- A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where n is the number of vertices.
  - If n=5, the number of spanning trees would be equal to 125.
- Each connected and undirected graph contains at least one spanning tree.
- The disconnected graph does not contain any spanning tree.



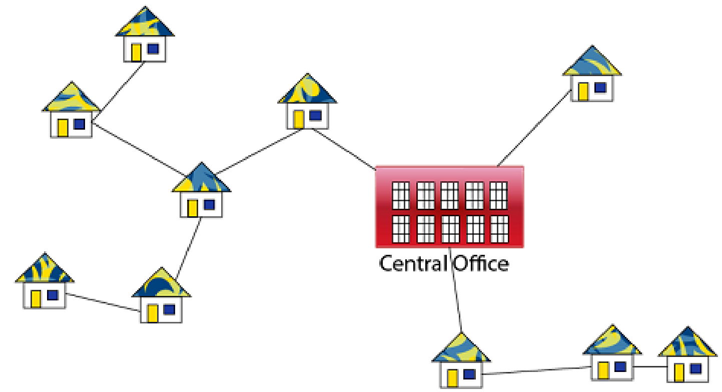
Wiring : Naive Approach



Expensive!

APPLICATIONS OF MINIMUM SPANNING TREE

Wiring : Better Approach



Minimize the total length of wire connecting the customers

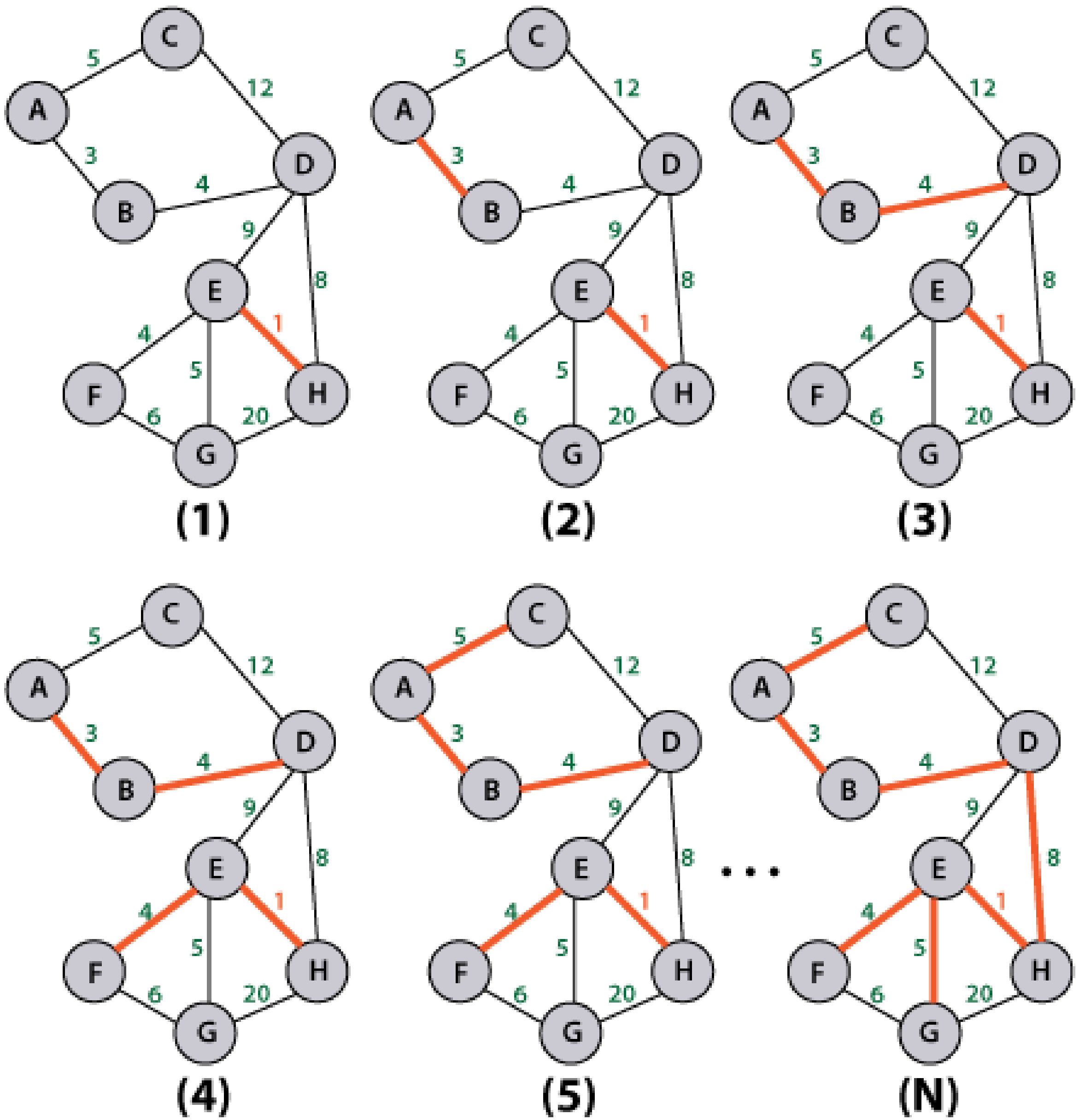
KRUSKAL'S ALGORITHM

- A greedy algorithm to construct an MST for a connected weighted graph
  - put the smallest weight edge without forming a cycle in the MST constructed so far

- Algorithm
  - Take connected and undirected graph from the user.
  - We then sort all the edges from low weight to high weight.
  - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
  - Keep adding edges until we reach all vertices.

- The complexity of the algorithm is $O(|E| \log |E|) = O(|E| \log |V|)$.

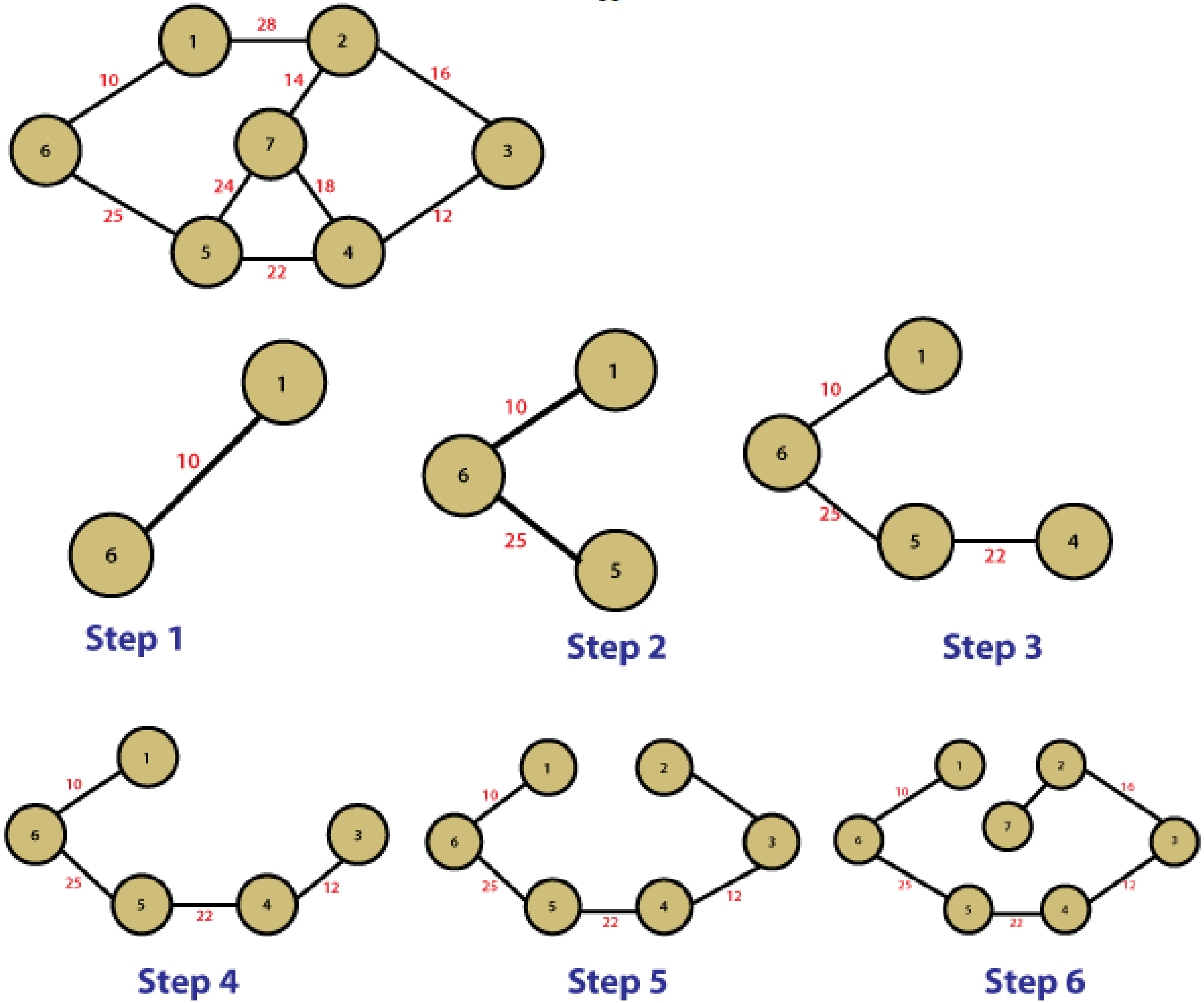
Kruskal Algorithm



PRIM'S ALGORITHM

- A greedy algorithm: starts with an empty spanning tree, and maintains two vertices sets:
  - vertices already included in MST; vertices not yet included.
  - At every step, it considers all edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.
- Algorithm
  - For all vertices, set its key-value to infinite. Set the first vertex's key value to 0.
  - Repeat the following steps until setOfMST contains all vertices.
    - Select vertex u that is not in setOfMST and having a minimum key-value. Add vertex u to setOfMST.
    - Change the key-value of all adjacent vertices of u: for an adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, change the key value as the weight of u-v.

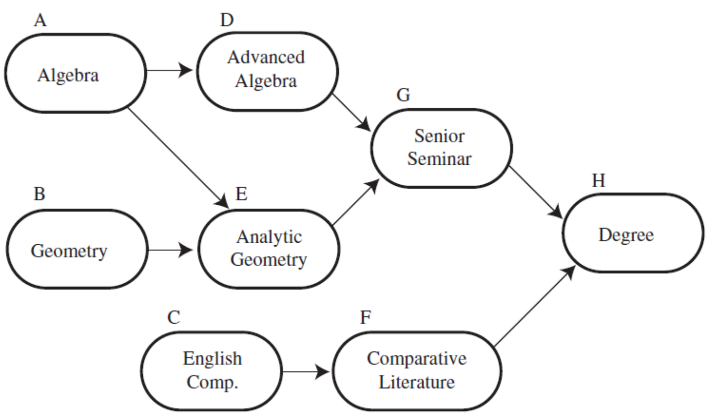
Prim's Algorithm



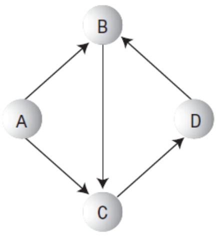
OUTLINE

- Concepts
- Implementations
- Examples

TOPOLOGICAL SORTING WITH DIRECTED ACYCLIC GRAPHS



- Arrange events in a specific order
- Repeat until all vertices are gone
  - Find a vertex that has no successors.
  - Delete this vertex from the graph, and insert its label at the beginning of a list.
- Topological-sort cannot handle a graph with cycles.



```
public int noSuccessors() // returns vert with no successors
{
    // (or -1 if no such verts)
    boolean isEdge; // edge from row to column in adjMat
    for(int row = 0; row < nVerts; row++) // for each vertex,
    {
        isEdge = false; // check edges
        for(int col = 0; col < nVerts; col++)
        {
            if( adjMat[row][col] > 0 ) // if edge to
            {
                // another,
                isEdge = true;
                break; // this vertex
            } // has a successor
        } // try another
        if( !isEdge ) // if no edges,
            return row; // has no successors
    }
    return -1; // no such vertex
} // end noSuccessors()

public void topo() // topological sort
{
    int orig_nVerts = nVerts; // remember how many verts
    while(nVerts > 0) // while vertices remain,
    {
        // get a vertex with no successors, or -1
        int currentVertex = noSuccessors();
        if(currentVertex == -1) // must be a cycle
        {
            System.out.println("ERROR: Graph has cycles");
            return;
        }
        // insert vertex label in sorted array (start at end)
        sortedArray[nVerts - 1] = vertexList[currentVertex].label;
        deleteVertex(currentVertex); // delete vertex
    } // end while
    // vertices all gone; display sortedArray
    System.out.print("Topologically sorted order: ");
    for(int j = 0; j < orig_nVerts; j++)
        System.out.print( sortedArray[j] );
    System.out.println("");
} // end topo
```

SUMMARY

- Graphs consist of vertices connected by edges.
- Graphs can represent many real-world entities.
- Search algorithms allow to visit each vertex in a graph in a systematic way.
- Two main search algorithms: depth-first search (DFS) and breadth-first search (BFS).
- Depth-first search can be based on a stack; breadth-first search can be based on a queue.
- An MST consists of the minimum number of edges necessary to connect all vertices.
- For unweighted graph, depth-first search algorithm yields its MST.
- For weighted graph, Kruskal's algorithm and Prim's algorithm yield its MST.