**DISJOINT SETS**

Consider a set \( S = \{1,2,3,4,5,6,7,8,9,10\} \). These elements can be partitioned into three disjoint sets.

\[
A = \{1,2,3,6\} \quad B = \{4,5,7\} \quad C = \{8,9,10\}
\]

Each set can be represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from parent to children.

These sets can be represented by storing every element of the set in the same array. Since the set elements are numbered \( 1 \) through \( n \), we represent the tree nodes using an array \( P[1:n] \), where \( n \) is the maximum number of elements. The \( i \)th element of this array represents the tree node that contains element \( i \). This array element gives the parent pointer of the corresponding tree node. Notice that the root nodes have parent as \(-1\).

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>-1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Array representation of sets \( A, B, C \)

**OPERATIONS ON SETS**

1) **MEMBER(\(a,S\))**
   
   Determine whether \( a \) is a member of set \( S \), if so print “yes” otherwise print “no”
   
   Example: \( S = \{1,2,3\} \)
   
   MEMBER(2,\(S\)) it means 2 belongs to \( S \) or not
   
   The function prints Yes

2) **INSERT(\(a,S\))**

   Replaces set \( S \) by \( S \cup \{a\} \)

   Example: INSERT(4,\(S\))
   
   \( S = S \cup \{4\} \)
   
   \( S = \{1,2,3,4\} \)

3) **DELETE(\(a,S\))**

   Replaces set \( S \) by \( S - \{a\} \)

   Example: DELETE(2,\(S\))
   
   \( S = S - \{2\} \)
   
   \( S = \{1,3,4\} \)

4) **UNION(\(S1,S2,S3\))**

   Calculates \( S3 = S1 \cup S2 \)

   We assume \( s1 \) and \( s2 \) are disjoint
   
   \( S1 = \{1,2\} \quad s2 = \{4,6\} \)
   
   \( S3 = S1 \cup S2 = \{1,2,4,6\} \)
5) FIND(a)  
Prints name of the set in which a is currently a member.  
Example : FIND(4) this returns S2 since 4 belongs to S2.

6) SPLIT(a,S)  
This operation partitions S into two sets S1 and S2 such that  
S1={b|b<=a and b ∈ S}  
S2={b|b>a and b ∈ S}  
Example : S = { 1,2,4,6,9,12}  
SPLIT(4,S) then S1={1,2,4} S2={6,9,12}

7) MIN(S)  
prints the smallest element of the set S.  
Example S={2,4,10}  
The function prints minimum element of set S i.e. 2.

UNION operation  
UNION(i,j) it means the elements of set i and elements of set j are combined. If we want to  
represent UNION operation in the form of a tree; the UNION(i,j) i is parent ; j is the child.  
UNION of S1 and S2 S1US2

To obtain the union of two sets, all that has to be done is to set the parent field of one of the  
roots to the other root. This can be accomplished easily if with each set name, we keep a  
pointer to the root of the tree representing the set.

Algorithm UNION(i,j)
{
   //replace the disjoint sets with roots i and j.
   integer i,j
   PARENT(j):=i;
}
initially assume that the parent array contains all zeroes.

P
0 0 0 0 0 0  
1 2 3 4 5 6

UNION(1,3)
P
0 0 1 0 0 0  
1 2 3 4 5 6

UNION(2,5)
P
0 0 1 0 2 0  
1 2 3 4 5 6

UNION(1,2)
P
0 1 1 0 2 0  
1 2 3 4 5 6

Since the time taken for a UNION is constant, all the n-1 unions can be processed in time  
O(n). However each find required following sequence of parent pointers from the element to  
be found to the root. Since the time required to process a find for an element at level i of a  
tree is O(i), the total time needed to process the n finds is O(∑ i) = O(n²).
**FIND Operation**
FIND(i) implies that it finds the root node of i\(^{th}\) node, in other words, it returns the name of the set.

![Disjoint Sets Diagram]

UNION(1,3)

FIND(1) --------1
FIND(3)--------1 since its parent is 1 (i.e. root node is 1)

Algorithm FIND(i)
{
    //find the root for the tree containing elements
    integer i,j

    j = i;

    while (parent[j]>0) do
        j:=parent[j];

    return j;
}

Let us explore this algorithm with previous example.

![Array Representation]

Array representation is

<table>
<thead>
<tr>
<th>P</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

FIND(5) i=5, j=i , j=5

While(p[j]>0) do condition true

j←p[j] i.e., j=2

Again p[2]>0 do condition true

so j←p[2] i.e., j=1

While(P[1]>0) do condition false

Return(j). so 1 is root node of node 5. We can observe same thing from above FIND

**Time Complexity**: Each find requires following chain of parent links from node i to root. The time required to process a FIND for an element at level i of a tree is O(i). Hence the total time needed to process the n-2 finds is O(n^2).
**Weighting Rule** for UNION (i, j)
If the number of nodes in a tree i is less than the number in tree j, then make j the parent of i, otherwise make i the parent of j. Both arguments of UNION must be roots.

To implement the weighting rule, we need to know how many nodes are there in every tree. To do this easily, we maintain a count field in the root of every tree. If i is a root node then count[i] equals the number of nodes in that tree. Since all the nodes other than the roots of trees have a positive number in the P field, we can maintain the count in the P field of roots as a negative number.

Using this convention, the time required to perform a union has increased somewhat but still bounded by a constant. i.e O(1).

![Trees obtained using the weighting rule](image)
Algorithm UNION(i,j)
{
    // UNION sets with roots i and j, i ≠ j, using weighting rule
    // P[i] = -count[i] and P[j] = -count[j]

    temp:=p[i]+p[j];
    if (p[i]>p[j] then
    {
        // I has fewer nodes
        p[i]:=j;
        p[j]:=temp;
    }
    else
    {
        // j has fewer nodes
        p[j]:=i;
        p[i]:=temp;
    }
}

COLLAPSING RULE
If j is a node on the path from i to its root then set Parent(j)←root(i). A more sophisticated algorithm of FIND using collapsing rule is given below.

Algorithm Find(i)
{
    j←i;
    while (parent[j]>0)
    {
        j←parent[j];
    }
    k←i;
    while(k≠j)
    {
        t←parent[k];
        parent[k]←j;
        k←t;
    }
    return(j);
}
UNION and FIND Operations

Implement the following UNION and FIND Operations

<table>
<thead>
<tr>
<th>UNION(1,2)</th>
<th>UNION(3,4)</th>
<th>UNION(5,6)</th>
<th>UNION(7,8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION(1,3)</td>
<td>UNION(5,7)</td>
<td>UNION(1,5)</td>
<td></td>
</tr>
</tbody>
</table>

initially

UNION(1,2)
UNION(3,4)
UNION(5,6)
UNION(7,8)

Next we apply 8 FIND operations as follows

FIND(8)   FIND(8)   FIND(8)   FIND(8)
FIND(8)   FIND(8)   FIND(8)   FIND(8)

FIND(8) = root node i.e., By using first FIND algorithm it will require 3 moves
upword to reach the root node. Similarly for 8 FIND operations it will require 8*3=24 moves.
If we use FIND algorithm with collapsing rule, the first FIND(8) requires going up 3 link fields
(moves) and then resetting 3 links. Each of the remaining 7 FINDs requires going up
only 1 move (link field), the total cost is now only 13 moves.
DATA STRUCTURES
Data may be organised in many different ways. The logical or mathematical model of a particular organisation of data is called a data structure.

Examples) Arrays, Linked lists, Trees, Structures, Stacks, Queues, Graphs, Algebraic expressions

STACK
Def: A Stack is an ordered collection of data items, into which new items may be inserted and from which items may be deleted at one end called Top of the Stack.

Unlike an array stack provides the insertion and deletion of items. So a stack is a dynamic, constantly changing object.

New items may be put on top of the stack or items which are at the top of the stack may be removed.

The two changes which can be made to a stack are given special names, those are PUSH and POP. When an item is added to a stack it is pushed onto the stack and when an item is removed it is popped from the stack.

Although an array cannot be a stack it can be the home of the stack i.e., an array can be declared with a range that is large enough for the maximum size of the stack. During the course of program execution the stack will grow and shrink within the space reserved for it. One end of the array will be fixed bottom of the stack while the top of the stack will constantly shift as items are popped and pushed. Thus another field is needed to keep track of the current position of the top of the stack.

Algorithm to push an element into a stack.
Algorithm push(s,x)
// s is stack and x is the element to be pushed
{
    if (top = MAXSTACK)
        print(“Error – Stack Overflow ”);
    else
    {
        top = top+1;
        s[top] = x;
    }
}
Algorithm to pop an element from a stack.
Algorithm pop(s)
{
    int x;
    if (top = 0) then print(“Error – Stack Underflow”);
    else
    {
        x = s[top];
        top = top –1;
        return (x);
    }
}

QUEUE
Def: A Queue is a ordered collection of items from which items may be deleted at one end called the front of the Queue and into which items may be inserted at the other end called the rear of the queue.

Function to insert an element into a queue.
Algorithm insert (q,x)
qtype *q;
int x;
{
    if (q-> rear = maxQ ) then
        printf(“Error – Q overflow
”);
    else
    {
        q-> rear = q-> rear +1;
        q-> entry[q->rear] = x;
    }
}

Function to remove / delete an item from a queue.
Algorithm delete(q)
qtype *q;
{
    int x;
    x = q-> entry[q-> front];
    q-> front = q-> front +1;
    return (x);
}

Insertion and deletion for queues can be carried out in a fixed amount of time or O(1).
BINARY TREES

Binary tree is a tree in which no node can have more than two children.
Binary tree is either empty, or it consists of a node called the root together with two binary
trees called the left sub-tree and the right sub-tree.

Traversal of binary tree: One of the most important operations on a binary tree is traversal,
moving through all the nodes of the binary tree, visiting each one in turn.
We can traverse the binary tree in many ways.

<table>
<thead>
<tr>
<th>Traversal</th>
<th>Preorder</th>
<th>Inorder</th>
<th>Postorder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VLR</td>
<td>LVR</td>
<td>LRV</td>
</tr>
</tbody>
</table>

With preorder traversal, the node is visited before the subtrees.
With inorder traversal, the node is visited between the subtrees.
With postorder traversal, the node is visited after the subtrees.
In all cases left is visited before right.

Consider the following binary tree.

![Binary Tree Diagram]

To traverse a non-empty binary tree in preorder, we perform the following three operations.
(i) Visit the root
(ii) Traverse the left subtree in preorder
(iii) Traverse the right subtree in preorder

To traverse a non-empty binary tree in inorder, we perform the following three operations.
(i) Traverse the left subtree in inorder.
(ii) Visit the root.
(iii) Traverse the right subtree in inorder.

To traverse a non-empty binary tree in postorder, we perform the following three operations.
(i) Traverse the left subtree in postorder.
(ii) Traverse the right subtree in postorder.
(iii) Visit the root.

preorder (p) /* Function to traverse a binary tree in preorder */
treenode *p
{
    if ( p != NULL)
    {
        printf ("%d", p->data);
        preorder (p->left);
        preorder ( p->right);
    }
}
inorder(p) /* Function to traverse a binary tree in inorder */
treenode *p;
{
    if (p != NULL)
    {
        inorder(p->left);
        printf("%d", p->data);
        inorder(p->right);
    }
}

postorder(p) /* Function to traverse a binary tree in postorder */
treenode *p;
{
    if (p != NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d", p->data);
    }
}

**Problem:**
The inorder and preorder traversals of a binary tree are

<table>
<thead>
<tr>
<th>Inorder</th>
<th>D B A C E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preorder</td>
<td>A B D C E</td>
</tr>
</tbody>
</table>

Give the postorder traversal for the above tree.

**Solution:**
The first letter in preorder must be the root. So ‘A’ is our root. By the definition of inorder, all nodes preceeding ‘A’ must occur in the left subtree and all nodes succeeding ‘A’ must occur in right subtree.

At this stage by seeing the preorder ‘B’ comes before ‘D’. So ‘B’ is the next root and from the inorder we see ‘B’ has an empty right subtree and ‘D’ is in it’s left subtree.

At this stage by seeing the preorder ‘C’ comes before ‘E’. So ‘C’ is the next root and from the inorder we see ‘C’ has an empty left subtree and ‘E’ is in its right subtree.

The Post order traversal is DBECA
/* Algorithm for preorder traversal using iterative method - Non-Recursive method */

Algorithm preorder(root)
{
    treenode *p,*q;
    
    top := 0;  /* empty stack */
    push(s,NULL);
    p := root;
    while (p != NULL)
    {
        q := p;
        while (q!=NULL)
        {
            print(q->data);
            if (q->right != NULL) then
                push(s,q->right);
                q = q->left;
            }
        p = pop(s);
    }
}

/*----------Algorithm for inorder traversal using iterative method - Non-Recursive method */

Algorithm inorder(root)
{
    treenode *p;
    top := 0;
    p:=root;
    do
    {
        while (p!=NULL)
        {
            push(s,p);
            p:=p->left;
        }
        if(!empty(s)) then
        {
            p := pop(s);
            print(p->data);
            p := p->right;
        }
    } while( (!empty(s)) || (p!=NULL) );
}

Time complexity = O(n)
/* Algorithm for postorder traversal using iterative method - Non-recursive method */

Algorithm postorder(root)
{
  treenode *p,*q,*a;
  a=(treenode *)malloc(sizeof(treenode));
  a->data := 1;
  a->left := NULL;
  a->right := NULL;
  top:=0;
  push(s,NULL);
  push(s,NULL);
  p:=root;
  while (p!=NULL)
  {
    q:=p;
    while (q!=NULL)
    {
      push(s,q);
      if (q->right != NULL) then
      {
        push(s,q->right);
        push(s,a);
      }
      q:=q->left;
    }
    while((q=pop(s))>0)
    {
      if (q=a) then
        break;
      else
        print(q->data);
    }
    p=pop(s);
  }
}
GRAPHS
A graph G consists of a set of V or vertices (nodes) and a set of edges (arcs).
We write G = (V, E).
V is a finite non empty set of vertices.
E is a set of pairs of vertices. These pairs are called edges.
An edge e=(v,w), is a pair of vertices v and w, and is said to be incident with v and w.

Graph Traversal
A graph traversal means visiting all the nodes of the graph. Two graph traversal methods are commonly used. These are,
Depth First Search (DFS)
Breadth First Search (BFS)

DEPTH FIRST SEARCH: In graphs, we do not have any start vertex or any special vertex signaled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with vertex v. An adjacent vertex is selected and a depth first search is initiated from it. i.e. V1,V2,....Vk are adjacent vertices to vertex v. We may select any vertex from this list. Say we select v1. Now all the adjacent vertices to v1 are identified and all of those are visited. Next v2 is selected and all its adjacent vertices visited and so on. This process continues till all the vertices are visited. Consider the following graph.

**Depth First Search:**
Let us start with V1. Visit V1.
Its adjacent vertices are V2, V8, and V3.
Let us pick on V2. Visit V2.
Its adjacent vertices are V1, V4, V5.
Its adjacent vertices are V2, V8.
V2 is already visited. Let us pick on V8. Visit V8.
Its adjacent vertices are V4, V5, V1, V6, V7.
V4 and V1 are already visited. Let us pick on V5. Visit V5.
Its adjacent vertices are V2, V8.
Both are already visited. Therefore we back track.
We have V6 and V7 unvisited in the list of V8. We may visit any. We visit V6.
Its adjacent are V8 and V3. Obviously the choice is V3. Visit V3.
Its adjacent vertices are V1, V7.
We visit V7.
All the adjacent vertices of V7 are already visited, we backtrack and find that we have visited all the vertices. Therefore the sequence of traversal is 
V1, V2, V4, V8, V5, V6, V3, V7.
We may implement the depth first search method by using a stack, pushing all unvisited vertices adjacent to the one just visited and popping the stack to find the next vertex to visit.
Algorithm for Depth First search:

Algorithm dfs (vertex V)
{
    visited [V] = true;
    for each w adjacent to V
        if (!visited [w])
            dfs(w);
}

**Complexity of DFS:** The procedure DFS is called once for each vertex of G. We can calculate the complexity of DFS by adding up the time taken by each of these calls to DFS. There are two parts to each call, the marking process and the for loop. The marking process takes a constant amount of time say C1. Each time through the loop, the for loop takes a constant amount of time to test the condition say C2. And a constant amount of time say c3 as an upper bound for the time to execute the body of the loop.

For each vertex, the loop will be calculated once for each entry on the vertex’s adjacency list. The total time to execute the loop for a single vertex V is bounded by C1+(C2+C3).

Time complexity  = \( O(|V|+|E|) \).

**BREADTH FIRST SEARCH**

In DFS we pick on one of the adjacent vertices; visit all of the adjacent vertices and back track to visit the unvisited adjacent vertices.

In BFS we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.

We start with \( V_1 \). Its adjacent vertices are \( V_2, V_8, V_3 \). We visit all one by one.

We pick on one of these, say \( V_2 \). The unvisited adjacent vertices to \( V_2 \) are \( V_4, V_5 \). We visit both. We go back to the remaining visited vertices of \( V_1 \) and pick on one of those say \( V_3 \). The unvisited adjacent vertices are \( V_6, V_7 \). There are no more unvisited adjacent vertices of \( V_8, V_4, V_5, V_6 \) and \( V_7 \). Thus the sequence so generated is \( V_1, V_2, V_8, V_3, V_4, V_5, V_6, V_7 \).

Here we need a queue instead of a stack to implement it. We add unvisited vertices adjacent to the one just visited at the rear and read at front to find the next vertex to visit.
Algorithm bfs (vertex v)
{
    vertex w;
    queue q;
    visited [v] = true;
    initialise (q);
    addqueue (q,v)
    while (! Emptyqueue(q))
    {
        deletequeue (q,v);
        for all vertices w adjacent to v
        if (!visited [w])
        {
            addqueue (q,w);
            visited[w] = true;
        }
    }
}

The sequence generated is V₁ V₂ V₈ V₃ V₄ V₅ V₆ V₇.

bfs(v₁)
visit v₁
add v₁ to queue
delete queue now v is v₁
adjacent vertices of V₁ are v₂, v₈, v₃
unvisited vertices are visited and added to queue. V₂ v₈ v₃ are visited and added to queue
delete queue. Now v is v₂
adjacent vertices of V₂ are v₁, v₄, v₅
v₁ is already visited. unvisited vertices v₄, v₅ are visited and added to queue
delete queue. Now V is V₈
adjacent vertices of V₈ are V₄, V₅, V₁, V₆, V₇
V₄, V₅, V₁ are already visited. unvisited vertices V₆, V₇ are visited and added to queue
delete queue. Now V is V₃
adjacent vertices of V₃ are V₁, V₆, V₇
V₁, V₆, V₇ are already visited. There are no unvisited vertices of V₃
Delete queue Now V is V₄
Adjacent Vertices of V₄ are V₂ V₈ both are already visited
And so on….

Complexity of BFS: The Operations of enqueueing and dequeueing take O(1) time. So the
total time devoted to queue operations is O(V). The total time spent in scanning
adjacency list is O(E). The total run time for BFS is O(V+E).
DFS Vs BFS

<table>
<thead>
<tr>
<th>Method</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The DFS algorithm explores each possible path to its conclusion before another path is tried.</td>
<td>BFS is exactly the opposite of DFS. In this method, each node on the same level is checked before the search proceeds to the next level.</td>
</tr>
<tr>
<td>Data structure used</td>
<td>Stack (LIFO list)</td>
<td>QUEUE(FIFO list)</td>
</tr>
<tr>
<td>Type of edges</td>
<td>Back edges</td>
<td>Cross edges</td>
</tr>
<tr>
<td>Complexity</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>Applications</td>
<td>Spanning forest, connected components path and cycles, Biconnected components, articulation point</td>
<td>Spanning forest, connected components path and cycles, shortest path, to produce a reverse topological ordering of nodes.</td>
</tr>
</tbody>
</table>

**Graph Representation**

Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus this mathematical structure must be represented as some kind of data structures. Two such representations are commonly used. These are,

**Adjacent matrix Representation and Adjacency list representation**

The choice of representation depends on the application and function to be performed on the graph.

**Adjacency Matrix:**

The adjacency matrix A for graph G = (V,E) with n vertices, is an nxn matrix of d bits, such that,

\[ A_{ij} = 1 \text{ if there is an edge from } v_i \text{ to } v_j \text{ and } A_{ij} = 0 \text{ if there is no such edge.} \]

```
Vertice 1 2 3 4 5
1 0 1 1 0 1
2 1 0 1 0 0
3 1 1 0 1 1
4 0 0 1 0 1
5 1 0 1 1 0
```

You may observe that the adjacency matrix for an undirected graph is symmetric, as the lower and upper triangles are same. Also all the diagonal elements are zero.
The total number of 1’s account for the number of edges in the digraph. The number of 1’s in each row tells the out degree of the corresponding vertex.

**Adjacency List Representation:** In this representation, we store a graph as a linked structure. We store all the vertices in a list and then for each vertex, we have a linked list of its adjacent vertices.

The adjacency list representation needs a list of all of its nodes. i.e.

And for each node a linked list of its adjacent node. Therefore we shall have
Note that adjacent vertices may appear in the adjacency list in arbitrary order. Also an arrow from $v_2$ to $v_3$ in the list linked to $v_1$ does not mean that $v_2$ and $v_3$ are adjacent.

**Connected Graph:** In an undirected graph $G$, two vertices are said to be connected iff there is a path from $u$ to $v$ (since $G$ is an undirected graph this means there must also be a path from $v$ to $u$). An undirected graph is said to be connected iff for every pair of distinct vertices $u$ and $v$ in $V(G)$, there is a path from $u$ to $v$ in $G$.

![Diagram of graphs G1 and G2](attachment://graphs.png)

**G1 and G2 are Connected graphs**

![Diagram of graph G3](attachment://graphG3.png)

**G3**

Graph $G_3$ is not connected. It is one graph having two unconnected components. Since there are unconnected components, it is an unconnected graph.
STRONGLY CONNECTED
A tree is a connected acyclic graph (which has no cycles). A directed graph is said to be strongly connected iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u.

A digraph is called strongly connected if there is a directed path from any vertex to any other vertex.

There does not exist a directed path from vertex 1 to vertex 4 also from vertex 5 to other vertices and so on. Therefore it is a weakly connected graph.

Let us make the above graph strongly connected as

Strongly Connected Graph
BICONNECTED COMPONENTS

A connected undirected graph G is said to be biconnected if it remains connected after removal of any one Vertex and the edges that are incident upon that vertex.

A biconnected component of an undirected graph is a maximal biconnected subgraph that is a biconnected sub graph not contained in any larger biconnected sub graph.

Articulation point or cut point

A Vertex v is an articulation point or cut point for an undirected graph G if there are distinct vertices w and x, distinct from v also such that v is in every path from w to x.

[Diagram of an un-directed graph]

An Un-Directed Graph

[Diagram of bi-connected components of the above graph]

Bi-Connected Components of Above Graph

Articulation Point:

- After deleting vertex B and its incident edges the given graph is divided into two non empty components.

[Diagram of graph obtained after deleting B from the graph]

Graph Obtained after Deleting B from the Graph
After deleting vertex E and its incident edges the resulting non empty components are shown below.

Graph Obtained after Deleting E from the Graph

After deleting vertex F and its incident edges the resulting non empty components are shown below.

Graph Obtained after Deleting F from the Graph

From the above graphs we can say that B, E and F are the articulation points of the given graph.

**Bi-Connected Graph**

An Un-Directed Graph

After vertex B, if the Graph structure still remains connected, it is said to be a bi-connected graph.

Graph Obtained after Deleting B from the above Graph
Construction of **Bi-Connected Graph:::**

i) Check whether the given graph is bi-connected or not

ii) If the given graph is not bi-connected then identify all the Articulation points.

iii) If Articulation points exist, determine a set of edges whose inclusion makes the graph bi-connected

An Un-Directed Graph

- Given graph is not a bi-connected graph.
- The Articulation Points are 2,3,5
- To Transform the given graph into bi-connected graph, the new edges are included corresponding to the Articulation point.
- Edges corresponding to the Articulation point 3-(4, 10) (10, 9)
- Edges corresponding to the Articulation point 2-(1, 5) (3, 8)
- Edges corresponding to the Articulation point 5-(6, 7).
Single Source Shortest Path Problem

There are many paths from A to H.

For example length of path \( A \ F \ D \ E \ H = 1 + 3 + 4 + 6 = 14 \)
\( A \ B \ C \ E \ H = 2 + 2 + 3 + 6 = 13 \)

We may further look for a path with length shorter than 13 if exists.

**Algorithm:**

1. We start with source vertex A.
2. We locate the vertex closest to it. B, F are adjacent vertices. Length of AF < length of AB so we choose F.
3. Now we look for all the adjacent vertices excluding the just earlier vertex of newly added vertex and the remaining adjacent vertices of earlier vertices, i.e., we have D, E and G (as adjacent vertices of F) and B (as remaining adjacent vertex of A).

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>AFD</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>AFE</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>AB</td>
<td>2</td>
</tr>
</tbody>
</table>

We choose vertex B.

4) We go back to step 3 and continue till we exhaust all the vertices.

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>ABD</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>ABC</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>ABE</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>AFE</td>
<td>4</td>
</tr>
</tbody>
</table>

We may choose D, C or E.

We choose say D through B.

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>ABC</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>AFE</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>ABE</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>BDE</td>
<td>8</td>
</tr>
</tbody>
</table>

We may choose C or E, choose C.
### Vertices that may be attached

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>AFE</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>ABE</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>ABDE</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>ABCE</td>
<td>7</td>
</tr>
<tr>
<td>H</td>
<td>ABCH</td>
<td>5</td>
</tr>
</tbody>
</table>

We choose E via AFE.

### Vertices that may be attached

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>AFEG</td>
<td>11</td>
</tr>
<tr>
<td>H</td>
<td>ABCH</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>AFEH</td>
<td>10</td>
</tr>
</tbody>
</table>

We choose H via ABCH.

### Vertices that may be attached

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>Path from A</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>AFG</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>AFEG</td>
<td>11</td>
</tr>
</tbody>
</table>

We choose path AFG.

Therefore the shortest paths from source vertex A to all the other vertices are:

- AB
- ABC
- ABD
- ABCH
- AF
- AFE
- AFG.
SPANNING TREE
A spanning tree for a connected undirected graph $G=(V,E)$ is a sub group of $G$ that is an undirected tree and contains all the vertices of $g$. A spanning tree of a graph should include all the vertices and a subset of edges.

Consider the following graph and some of the tree structures for the graph.

SPANNING TREES
You may notice that all the spanning trees differ from each other significantly, however for each structure
i) The vertex set is same as that of graph $G$
ii) The edge set is a subset of $G(E)$ and
iii) There is no cycle.
Such a structure is called spanning tree of graph. Take any vertex $V$ as an initial partial tree and add edges one by one so that each edge joins a new vertex to the partial tree.
MINIMAL SPANNING TREE

Frequently we encounter weighted graphs and we need to build a sub-graph that must include every vertex in the graph. To construct such a graph with least weight or least cost we must not have cycles in it.

Consider the above graph. The MST for this graph could be building a least cost communication network. We begin by first selecting an edge with least cost, it can between any two vertices of graph G. Subsequently from the set of remaining edges, we can select another least cost edge and so on. Each time an edge is picked, we determine whether or not the inclusion of this edge into the spanning tree being constructed creates a cycle. If it does this edge is discarded. If no cycle is created, this edge is included in the spanning tree being constructed. The minimum cost is BA.

Now we have

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>EDGE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>BF</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>AF</td>
<td>7</td>
</tr>
<tr>
<td>G</td>
<td>BG</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>AC</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>AD</td>
<td>5</td>
</tr>
</tbody>
</table>

The least cost is AC. Therefore we choose AC. Now we have

<table>
<thead>
<tr>
<th>Vertices that may be attached</th>
<th>EDGE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>BF</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>AF</td>
<td>7</td>
</tr>
<tr>
<td>G</td>
<td>BG</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>CG</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>AD</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>CE</td>
<td>8</td>
</tr>
</tbody>
</table>

The least cost is AD. Therefore we choose AD now we have
Vertices that may be attached | EDGE | COST
--- | --- | ---
F | BF | 8
   | AF | 8
   | DF | 10
G | BG | 9
   | CG | 10
E | CE | 8
   | DE | 9
   | FE | 11

AF is the minimum cost edge. Therefore we add it to the partial tree. Now we have
Vertices that may be attached | EDGE | COST
--- | --- | ---
G | BG | 9
   | CG | 10
E | CE | 8
   | DE | 9
   | FE | 11

Obvious choice is CE.
The only vertex left is G and we have the minimum cost edge that connects it to the tree is BG. Therefore we add it and the minimal spanning tree constructed would be of costs $9 + 3 + 7 + 5 + 4 + 8 = 36$.

**POSSIBLE QUESTIONS:**

1) Write a non recursive algorithm of inorder traversal of a tree and also analyse its time complexity.
2) Write a non recursive algorithm of preorder traversal of a tree.
3) Write a non recursive algorithm of postorder traversal of a tree.
4) The inorder and preorder traversals of a binary tree are
   - Inorder – D B A C E
   - Preorder – A B D C E
   Give the postorder traversal for the above tree.
5) Show that the inorder and preorder sequences of a binary tree uniquely define binary tree.
   - Preorder : ABDIEHJCFKLGM
   - Inorder : DIBJHEAFKLCGM
6) Differentiate between BFS and DFS.
7) Explain properties of DFS.
8) Explain game tree with an example.