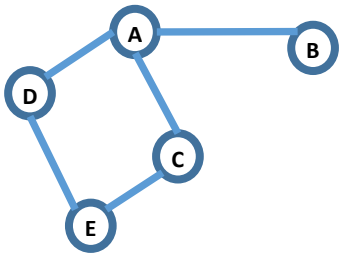
Activity 9-1

All possible paths with BSF (manual)

1. Trace through the steps using BSF (using queue and without coding) to find out all the possible paths from one vertex to another (say Vertex B to Vertex E) for the following graph (the adjacent list is also shown for each vertex). Assuming all edges are bidirectional.



A	B C D
B	A
C	A E
D	A E
E	C D

Evaluate source: B
Add all adjacent of B to a queue (so only A in queue)
Loop to poll from queue and evaluate further (only 1 loop for A)

Evaluate: A
Add all adjacent of B to a queue (so D,C,B in queue)
Loop to poll from queue and evaluate further (3 loops)

Loop 1 from A

Evaluate: D
(E,A in queue)
Loop (2 loops)

Evaluate: E
Destination found!

Evaluate: A
Skip as A has been visited

Path found!
Return.

Return.

Loop 2 from A

Evaluate: C
(A,E in queue)
Loop (2 loops)

Evaluate: E
Destination found!

Evaluate: A
Skip as A has been visited

Path found!
Return.

Return.

Finish all 3 loops.
Return.

Loop 3 from A

Evaluate: B
Skip as B has been visited

Return.

2. Consider the following data structure
    - a HashSet structure to record the visited vertex.
    - an ArrayQueue to holds the adjacent vertices of the vertex under evaluation
    - an ArrayList to record the path
    - another ArrayList of ArrayList to hold the multiple paths

3. This problem needs a recursive program as the number of vertices and edges in the graph is not known at compile time. Consider carefully:
    - What are the input arguments for the recursive method
    - How to managed the status of visited vertex and path after each recursion

4. Use the skeleton in Activity 9-2 as a guide.
5. Try to get a better understanding of the nature of the problem by repeat the manual tracing process for a different graph.

Activity 9-2

All possible paths with BSF (Coding with Recursive Call)

1. Code the algorithm in Activity 9-1.
2. The skeleton is given below.
3. A HashSet ***isVisited*** is being used to track vertices being visited instead of ArrayList (to take care of duplication)

```
public class TestBSF {
   public static ArrayList<ArrayList<Vertex>> overall = new ArrayList<ArrayList<Vertex>>();
   public static void main(String[] args) {
      Vertex a = new Vertex("A");
      Vertex b = new Vertex("B");
      Vertex c = new Vertex("C");
      Vertex d = new Vertex("D");
      Vertex e = new Vertex("E");

      a.adjList.add(b);
      a.adjList.add(c);
      a.adjList.add(d);
      b.adjList.add(a);
      c.adjList.add(a);
      c.adjList.add(e);
      d.adjList.add(a);
      d.adjList.add(e);
      e.adjList.add(c);
      e.adjList.add(d);

       Set<Vertex> isVisited = new HashSet<>();
      ArrayList<Vertex> pathList = new ArrayList<>();

     // arbitrary source and destination
      Vertex source = b;
      Vertex dest = e;
      isVisited.add(source);
      pathList.add(source);
      bfs(source,dest,isVisited,pathList);

      if (overall.size()==0)
         System.out.println("No path found");
      else {
         System.out.println(
               "Following are all different paths from "   + source.label + " to " + dest.label);
        // Code to print out all paths from overall

      }
   }

  // continue next page
```

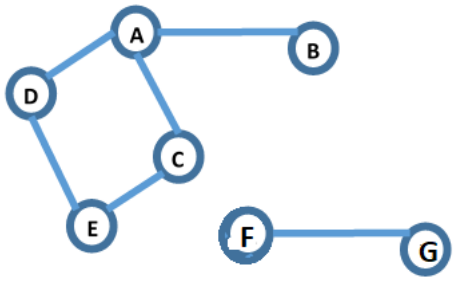
private static void bfs(Vertex u, Vertex d, Set<Vertex> isVisited, ArrayList<Vertex> localPathList)

```
{
   if (u.label.equals(d.label)){
      // Code to add localPathList into overall

      return;
   }

    Vertex temp=null;
   ArrayDeque<Vertex> localQ = new ArrayDeque<>();
   u.adjList.forEach(eee->localQ.addFirst(eee));
   while (!localQ.isEmpty()){

      //  Code to manage localPathList, ArrayDeque, isVisited and recursive call of bfs



   return;
  }
}
```

```
class Vertex{
    String label;
    ArrayList<Vertex> adjList = new ArrayList<>();
    Vertex (String label) {
        this.label = label;
    }
}
```

Test your program with different sources and destinations.

Test your program with different graphs, such as:



Activity 9-3

All possible paths with DSF (Coding with Recursive Call)

1. Implement Activity 9-2 using DSF.
2. DSF uses stack instead of queue. Take advantage of the stack nature of recursive calls.
3. Do a manual tracing to better understand the algorithm needed.
4. The skeleton is given below.

```
public class TestDSF {
   public static ArrayList<ArrayList<Vertex>> overall = new ArrayList<ArrayList<Vertex>>();
   public static void main(String[] args) {
      Vertex a = new Vertex("A");
      Vertex b = new Vertex("B");
      Vertex c = new Vertex("C");
      Vertex d = new Vertex("D");
      Vertex e = new Vertex("E");

      a.adjList.add(b);
      a.adjList.add(c);
      a.adjList.add(d);
      b.adjList.add(a);
      c.adjList.add(a);
      c.adjList.add(e);
      d.adjList.add(a);
      d.adjList.add(e);
      e.adjList.add(c);
      e.adjList.add(d);

       Set<Vertex> isVisited = new HashSet<>();
      ArrayList<Vertex> pathList = new ArrayList<>();

     // arbitrary source and destination
      Vertex source = b;
      Vertex dest = e;
      isVisited.add(source);
      pathList.add(source);

      dfs(source,dest,isVisited,pathList);

      if (overall.size()==0)
         System.out.println("No path found");
      else {
         System.out.println(
               "Following are all different paths from "   + source.label + " to " + dest.label);
          // Code to print out all paths from overall
      }
   }

  // continue next page
```

private static void dfs(Vertex u, Vertex d, Set<Vertex> isVisited, ArrayList<Vertex> localPathList)

```
  {
     if (u.label.equals(d.label)){
        // Code to add localPathList into overall

        return;
     }

    isVisited.add(u);
     for (Vertex i : u.adjList) {
        //  Code to manage localPathList, isVisited and recursive call of dfs



     return;
  }
}

class Vertex{
    String label;
    ArrayList<Vertex> adjList = new ArrayList<>();
    Vertex (String label) {
        this.label = label;
    }
}
```