**Case 1: Simplify the Problem (Not considering millions of people)**

|  |
| --- |
| Linkedlist<Person> findPathBiBFS(HashMap<Integer, Person> people,                                      int source, int destination)  {      BFSData sourceData = new BFSData(people.get(source));      BFSData destData = new BFSData(people.get(destination));        while (!sourceData.isFinished() && !destData.isFinished())      {            /\* Search out from source. \*/          Person collision = searchlevel(people, sourceData, destData);          if (collision != null)              return mergePaths(sourceData, destData, collision.getID());            /\* Search out from destination. \*/          collision = searchlevel(people, destData, sourceData);          if (collision != null)              return mergePaths(sourceData, destData, collision.getID());      }        return null;  }      /\* Search one level and return collision, if any.\*/  Person searchLevel(HashMap<Integer, Person> people,                  BFSData primary, BFSData secondary)  {        /\* We only want to search one level at a time. Count         how many nodes are currently         in the primary's level and only do that many nodes.         We continue to add nodes to the end. \*/        int count = primary.toVisit.size();      for (int i= 0; i < count; i++)      {          /\* Pull out first node. \*/          PathNode pathNode = primary.toVisit.poll();          int personld = pathNode.getPerson().getID();            /\* Check if it's already been visited. \*/          if (secondary.visited.containsKey(personid))              return pathNode.getPerson();            /\* Add friends to queue. \*/          Person person = pathNode. getPerson();          Arraylist<Integer> friends = person.getFriends();          for (int friendid : friends)          {              if (!primary.visited.containsKey(friendid))              {                  Person friend= people.get(friendld);                  PathNode next = new PathNode(friend, pathNode);                  primary.visited.put(friendld, next);                  primary.toVisit.add(next);              }          }      }      return null;  }      /\* Merge paths where searches met at the connection. \*/  Linkedlist<Person> mergePaths(BFSData bfsl, BFSData bfs2,                                            int connection)  {      // endl -> source, end2 -> dest      PathNode endl = bfsl.visited.get(connection);      PathNode end2 = bfs2.visited.get(connection);        Linkedlist<Person> pathOne = endl.collapse(false);      Linkedlist<Person> pathTwo = end2.collapse(true);        pathTwo.removeFirst(); // remove connection      pathOne.addAll(pathTwo); // add second path        return pathOne;  }    class PathNode  {      private Person person = null;      private PathNode previousNode = null;      public PathNode(Person p, PathNode previous)      {          person = p;          previousNode = previous;      }        public Person getPerson()      {          return person;      }        public Linkedlist<Person> collapse(boolean startsWithRoot)      {          Linkedlist<Person> path= new Linkedlist<Person>();          PathNode node = this;          while (node != null)          {              if (startsWithRoot)                  path.addlast(node.person);              else                  path.addFirst(node.person);              node = node.previousNode;          }            return path;      }  }    class BFSData  {      public Queue<PathNode> toVisit = new Linkedlist<PathNode>();      public HashMap<Integer, PathNode> visited =                                   new HashMap<Integer, PathNode>();        public BFSData(Person root)      {          PathNode sourcePath = new PathNode(root, null);          toVisit.add(sourcePath);          visited.put(root.getID(), sourcePath);      }      public boolean isFinished()      {          return toVisit.isEmpty();      }  } |

**Case 2: Handle Millions of Users**

|  |
| --- |
| // A server that holds list of all machines  class Server  {      HashMap<Integer, Machine> machines =                         new HashMap<Integer, Machine>();      HashMap<Integer, Integer> personToMachineMap =                          new HashMap<Integer, Integer>();        public Machine getMachineWithid(int machineID)      {          return machines.get(machineID);      }        public int getMachineIDForUser(int personID)      {          Integer machineID = personToMachineMap.get(personID);          return machineID == null ? -1 : machineID;      }        public Person getPersonWithID(int personID)      {          Integer machineID = personToMachineMap.get(personID);          if (machineID == null) return null;            Machine machine = getMachineWithid(machineID);          if (machine == null) return null;            return machine.getPersonWithID(personID);      }  }    // A person on social network has id, friends and other info  class Person  {      private Arraylist<Integer> friends =                                 new Arraylist<Integer>();      private int personID;      private String info;        public Person(int id)      {          this.personID =id;      }      public String getinfo()      {          return info;      }      public void setinfo(String info)      {          this.info = info;      }      public Arraylist<Integer> getFriends()      {          return friends;      }      public int getID()      {          return personID;      }      public void addFriend(int id)      {          friends.add(id);      }  } |