1. Random pick users:
   We will pick users randomly from the team of which number of length is the largest. And pick another one from other users. TC is O(n)
   from random import *

```
from collections import defaultdict
def matchUsers(users):
    memo = defaultdict(set)
    result = []
    for user in users:
        memo[user['team']].add(user['id'])
    while True:
        user_pair = randomPickUser(memo)
        if not user_pair:
            return result
        else:
            result.append(user_pair)


def randomPickUser(user_memo):
    if len(user_memo.keys()) < 2:
        return False

    v, k = max(map(lambda a: (len(a[1]), a[0]), user_memo.items()))
    user1 = pickOne(user_memo, k)
    user2_key = k
    keys = list(user_memo.keys())
    while user2_key == k:
        user2_key = choice(keys)
    user2 = pickOne(user_memo, user2_key)
    return (user1, user2)

def pickOne(user_memo, key):
    user = choice(list(user_memo[key]))
    user_memo[key].remove(user)
    if len(user_memo[key]) == 0:
        del user_memo[key]
    return user
```

2. Waiting List
   We will use a single linked list to record our waiting list. Every time we will traverse from head and find qualified node in our linked list. Then we will delete that node. And we will add new party to the end of waiting list. TC is O(n) and O(1).
   class WaitingList:

```
def __init__(self, parties):
    self.dummy = Node(0, 0)
    self.tail = self.dummy
    for party in parties:
        self.addpartyToWaitingList(party)

def addpartyToWaitingList(self, party):
    node = Node(party['label'], party['number'])
    self.tail.next = node
    self.tail = node

def moveWaitingList(self, capacity):
    node = self.dummy
    while node.next and node.next.val != capacity:
        node = node.next
    ret = node.next
    if ret:
        node.next = ret.next
        if self.tail == ret:
            self.tail = node
        return ret.key
    else:
        return None
```

3. Find Destination
   We will use BFS to find all unvisited nodes until we get node whose children is an empty list. TC is O(n)

```
def findDest(pairs, startPoint):
    children = defaultdict(list)
    for pair in pairs:
        children[pair[0]].append(pair[1])
    cur = set([startPoint])
    visited = set([startPoint])
    while cur:
        next_ite = set()
        for node in cur:
            if len(children[node]) == 0:
                return node
            for n in children[node]:
                if n not in visited:
                    next_ite.add(n)
                    visited.add(n)
        cur = next_ite
    return None
```

4. Waiting List Small Party Version
   It's very similar to question 2 except that we need to check their arriving time. TC is O(n)

```
class Node:
    def __init__(self, time, value, key):
        self.time = time
        self.value = value
        self.key = key
        self.next = None

class WaitingList:
    def __init__(self):
        self.dummy = Node(0, 0, 0)
        self.tail = self.dummy

    def addPartyTOWL(self, party, time):
        value, key = party['number'], party['label']
        node = Node(time, value, key)
        self.tail.next = node
        self.tail = node

    def moveWL(self, time, capacity):
        node = self.dummy
        ret = None
        while node.next:
            if node.next.value == capacity or (node.next.value < capacity and time -
node.next.time >= 30):
                ret = node.next
                node.next = ret.next
                if self.tail == ret:
                    self.tail = node
                break
            node = node.next
        return ret.key if ret else None
```

5. Design twitter
   We will use two hashmap to store our follow infos and tweets. For getNewsFeed, we will fetch all posts from user itself and it followeers and then get top 10. TC is O(n)

```
from collections import defaultdict
from heapq import *
class Twitter:

    def __init__(self):
        """
```

```
        Initialize your data structure here.
        """
        self.count = 0
        self.tweets = defaultdict(list)
        self.follows = defaultdict(set)

    def postTweet(self, userId: int, tweetId: int) -> None:
        """
        Compose a new tweet.
        """
        self.tweets[userId].append((self.count, tweetId))
        self.count += 1

    def getNewsFeed(self, userId: int) -> List[int]:
        """
        Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
        """
        tweets = self.tweets[userId][:]
        topKtweets = []
        for followUserId in self.follows[userId]:
          tweets += self.tweets[followUserId]
        for tweet in tweets:
          heappush(topKtweets, tweet)
          if len(topKtweets) > 10:
            heappop(topKtweets)
        topKtweets.sort(reverse=True)
        return list(map(lambda a: a[1], topKtweets))


    def follow(self, followerId: int, followeeId: int) -> None:
        """
        Follower follows a followee. If the operation is invalid, it should be a no-op.
        """
        if followerId != followeeId:
          self.follows[followerId].add(followeeId)


    def unfollow(self, followerId: int, followeeId: int) -> None:
        """
        Follower unfollows a followee. If the operation is invalid, it should be a no-op.
        """
```

```
        self.follows[followerId].discard(followeeId)
```