1. (Solution and Grading by: Jayasi)

Map 1 - Input (a, b) where a follows b
   If (b == "@ChicagoBears")
      // If the user follows @ChicagoBears, add it as a value for the reduce call to be differentiated by a
      special prefix
      Emit (a, "@SpecialValue-ChicagoBears")
   Else:
      Emit (b, a)

Reduce 1 - Input (key, list)
   If (size(list) > 1M & list.contains("@SpecialValue-ChicagoBears")
      Emit(key)

2. (Solution and Grading by: Akshun)

Map 1 - Input (a, b) where a follows b
   If (b == "@packers"):
      Emit ("@packers", a)
   else:
      Emit(sort(a, b), "")
      # Meaning of sort: put a,b in a list and sort the list lexicographically. This ensures
      any ordering of a,b yields same key.

Reduce 1 - Input (key,list)
   If (key == "@packers"):
      # If key is packers, then the list contains users which follow the packers
      Emit (sort(u,v), "") for each unique (u,v) in list
      # Do not output (u,v) and (v,u). take only one of them.
   else:
      If len(list) == 2: # we know that a,b follow each other
      Emit(key, "") # key is a lexicographic sort of usernames.
Map 2 - Input (Key, Value) # Value is empty. Key is sorted tuple of usernames
   Emit (Key, Value)

Reduce 2 - input (a,b, list of empty strings “”)
If len(list) == 2: # we know that a,b follow each other and they follow packers because
   the pair was once emitted inside first if block in Reduce 1 and emitted once in the else block.
   Key = split(key)[0]
   Value = split(key)[1]
   Emit (Key, Value)

3. (Solution and Grading by: Xiaoyao)
Map1:
   // key: userId
   // value: userId
   void map(key, value) {
       emit(<key, 'follows', value>)
       emit(<value, 'is followed by', key>)
   }

Reduce1:
   // key: userId
   // values: list of <'follows', userId> and/or <'is followed by', userId>
   void reduce(key, values) {
       numFollowings := 0
       numFollowers := 0
       friendMap := map()
       friends := []

       for val in values {
           if val[0] == 'follows' {
               numFollowings ++
           } else {
               numFollowers ++
           }

           if friendMap.hasKey(val[1]) {
               ...
friendMap[val[1]] ++
} else {
    friendMap[val[1]] = 1
}

if numFollowers >= 1M {
    // find out friends, friends are who follows each others
    for k, v in friendMap {
        if v == 2 {
            friends.append(k)
        }
    }

    for friend in friends {
        emit(friend, 'has a friend with >=1M followers')  // satisfy condition #3
    }
}

if numFollowings >= 10 {
    emit(key, 'has >=1M followers and follows >=10 users')  // satisfy condition #1 and #2
}
}

Map2:
// key: userId
// value: either <userId, 'has a friend with >=1M followers'> or <userId, 'has >=1M followers and follows >=10 users'>
void map(key, value) {
    emit(key, value)  //noop
}

Reduce2:
// key: userId
// values: list of 'has a friend with >=1M followers' and/or 'has >=1M followers and follows >=10 users'
void reduce(key, values) {
    if values.contains('has a friend with >=1M followers') and values.contains('has >=1M followers and follows >=10 users') {
        emit(key)
    }
}

4. (Solution and Grading by: Mayank)

a. This approach does not provide any form of completeness. Consider a scenario where a certain process P is never selected by any of the other processes to send its heartbeats. The failure of P would not be detected by the other processes. Thus, this approach does not provide completeness.

b. This approach is not 100% accurate, since heartbeats may be lost or delayed, and thus a correct process may be perceived as faulty.

c.
1. Worst case load - (n - 1) heartbeats sent per second - a process is selected by all other processes to send it heartbeats
2. Best case load - 0 - a process is never selected by any other process to send it heartbeats
3. Average case - k heartbeats sent per second - sum (number of heartbeats a process P has to send) / n
   = nk/n (That sum equals the number of heartbeats each process asks for times the number of processes)
   = k

5. (Solution and Grading by: Akshun)
The probability that an arbitrary process gets pinged is \( P(k) = 1 - (1 - (1/n)^Q)^{(n - 1)} \).

\( Qf \) will be 0.99 (only 1 process fails which means there are 99 non-faulty processes)

\( T' = E[T'] * P(k) \)

\( T' \) will be independent of \( K \) as message loss rate is 0%. 
P(k) = 0.62
T' = 5*0.62 = 3.12 seconds

6. (Solution and Grading by: Faria)
The point guard is correct as the probabilities of picking a process to gossip to in either scenario are the same.
Probability of picking a process to gossip to with a full membership list = m/N
Probability of picking a process to gossip to with a partial membership list =
Probability of the process being in the membership list x Probability of choosing that process = k/N x m/k = m/N

7. (Solution and Grading by: Xiaoyao)
For the ith finger table entry at node n, instead of selecting the first peer beyond n+2^i, select among all peers between n+2^i and n+2^(i+1), that peer which is closest in round trip distance.

This ensures that as a query progresses, initially hops will be short in round trip distance (but long along the ring), and later they will be longer in round trip distance (and short along the ring).

Memory is clearly O(log(N)), since the number of finger table entries is the same as Chord.

Routing distance is no worse than Chord, since each finger table entry is at least as far (along the ring) as the corresponding Chord finger table entry (Chord selects the first beyond 2^i for fingers). Thus a query progresses along the ring no slower than a Chord route, thus it is O(log(N)).

8. (Solution and Grading by: Faria)
   a. The maximum capacity of each node is 4GB. A part of that space is used for the node’s own data, some replicated files and a metadata list of all files in the system. Thus, 4 GB = 500MB + 3*500MB + N * 400 * 100 bytes, where N is the number of nodes. N = 50,000
   b. This question flips the equation around:
      4 GB = 400 * S + 1000 * 400 * 100 bytes = 400*S + 0.04 GB
      S = 9.9 MB.
      Yes, this is realistic, given that 9.9 MB > 2.4 MB.
c. The highest possible churn rate can be found by:
100 KBps = number of nodes joining/leaving per second * 100 Bytes
Churn rate = 100 KBps/100bytes = 1000.
Note that the bandwidth incurred is 100 Bytes * churn rate, not 100 Bytes * churn rate/5 s. The timeout to disseminate the 100 B over x (=5 s) does not matter for bandwidth calculation, only the churn rate and 100 B does (because churn is happening all the time. think about it!).

9. (Solution and Grading by: Ashwini)
   Grading split: 4+3+3 points. (partial grades also given to reasonable answers)
   
a. 15 (all processes in the ring will receive the query message)
   b. 3
   c. 2 (messages can be sent from any process to any other process)

10. (Solution and Grading by: Ashwini)
    Grading split: 4+3+3 points. (partial grades also given to reasonable answers)
    
a. 2017, 2020, 2020, 1908, 1908, 1908, 1908, 1908, 1908, 1908, 1908