public void initializeCandidates(LinkedList<String> candidatesList, int p) {  
  
 voteTotal = p;  
 candidates = candidatesList;  
 maxHeap = new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());  
 candidates\_map = new HashMap<>();  
  
 for (int i = 0; i < candidates.size(); i++) {  
 String name = candidates.get(i);  
 candidates\_map.put(name, 0);  
 }  
  
 updateMaxHeap();  
  
}

**Time** **complexity**: O(n log n)

Reason: for (int i = 0; i < candidates.size(); i++) {  
 String name = candidates.get(i);  
 candidates\_map.put(name, 0);  
 }  
is O(n), where n is the size of the LinkedList of ‘candidates’

updateMaxHeap();

is O(n log n), because it is adding n elements to a PriorityQueue, which takes O(n log n) time in the worst case.

All other code in this snippet, aside from the code mentioned, is O(1).

Thus, O(n) + O(n log n). After chopping off the O(n), our worst case is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList.

**Space complexity**: O(n log n)

‘candidates\_map’ stores n elements, where n is the number of candidates, therefore it is O(n).

‘maxHeap’ stores n elements, where n is the number of candidates, therefore it is O(n).

‘candidates’ is a pointer to the LinkedList input, therefore it is O(1).

Thus, the space complexity is O(n), where n is the number of elements.

public void castVote(String candidate) {  
 if (candidates\_map.containsKey(candidate)) {  
 int currentVotes = candidates\_map.get(candidate);  
  
 candidates\_map.put(candidate, currentVotes + 1);  
 updateMaxHeap();  
 }  
}

**Time** **complexity**: O(n log n)

Reason: updateMaxHeap();  
 }  
is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList, because it is adding n elements to a PriorityQueue, which takes O(n log n) time in the worst case.

All other code in this snippet, aside from the code mentioned, is O(1).

Thus, Constants + O(n log n). After chopping off the constants, our worst case is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList.

**Space complexity**: O(1)

The purpose of this method is to modify data, now data is being introduced, nor is any data being cloned. Modifying data (in this case, incrementing a value by one) is O(1).

public void castRandomVote(){  
  
 Random gen = new Random();  
 String randomCandidate = candidates.get(gen.nextInt(candidates.size())); // Select random candidate  
 castVote(randomCandidate);  
  
 updateMaxHeap();  
  
}

**Time** **complexity**: O(n log n)

Reason: updateMaxHeap();  
 }  
is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList, because it is adding n elements to a PriorityQueue, which takes O(n log n) time in the worst case.

All other code in this snippet, aside from the code mentioned, is O(1).

Thus, Constants + O(n log n). After chopping off the constants, our worst case is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList.

**Space complexity**: O(1)

Variables are created, but there isn’t a big data structure being created not cloned.

public void rigElection(String candidate){  
  
 List<String> keys = new ArrayList<>(candidates\_map.keySet());  
  
 for (int i = 0; i < keys.size(); i++) {  
 candidates\_map.put(keys.get(i), 0);  
 }  
  
 int riggedVotes = ( (voteTotal / 2) + 1);  
 candidates\_map.put(candidate, riggedVotes);  
  
 int remainingVotes = voteTotal - riggedVotes;  
 int votesGiven = 0;  
  
 for (int i = 0; i < keys.size(); i++) {  
  
 String losing\_ccandidate = keys.get(i);  
  
 if (!losing\_ccandidate.equals(candidate) && votesGiven < remainingVotes) {  
 candidates\_map.put(losing\_ccandidate, 1);  
 votesGiven = votesGiven + 1;  
 }  
  
 }  
  
 updateMaxHeap();  
}

**Time** **complexity**: O(n log n)

Reason: for (int i = 0; i < keys.size(); i++) {  
  
 String losing\_ccandidate = keys.get(i);  
  
 if (!losing\_ccandidate.equals(candidate) && votesGiven < remainingVotes) {  
 candidates\_map.put(losing\_ccandidate, 1);  
 votesGiven = votesGiven + 1;  
 }  
  
 }  
is O(n), where n is the size of the LinkedList of ‘candidates’

for (int i = 0; i < keys.size(); i++) {  
 candidates\_map.put(keys.get(i), 0);  
 }

is O(n), where n is the size of the LinkedList of ‘candidates’

updateMaxHeap();

is O(n log n), because it is adding n elements to a PriorityQueue, which takes O(n log n) time in the worst case.

Thus, O(n) + O(n) + O(n log n). After chopping off the O(n)’s, our worst case is O(n log n), where n is the number of elements in the ‘candidates’ LinkedList.

**Space complexity**: O(n log n)

‘List<String> keys’ stores n elements, where n is the number of candidates, therefore it is O(n).

‘candidates\_map’ stores n elements, where n is the number of candidates, therefore it is O(n).

All other created variables are O(1).

Thus, the space complexity is O(n), where n is the number of elements.

public List<String> getTopKCandidates(int k){  
 List<String> topCandidates = new ArrayList<>();  
 PriorityQueue<Map.Entry<String, Integer>> tempHeap = new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());  
 tempHeap.addAll(candidates\_map.entrySet());  
  
 for (int i = 0; i < k && !tempHeap.isEmpty(); i++) {  
 topCandidates.add(tempHeap.poll().getKey());  
 }  
 return topCandidates;  
}

**Time** **complexity**: O(n log n)

Reason: for (int i = 0; i < k && !tempHeap.isEmpty(); i++) {  
 topCandidates.add(tempHeap.poll().getKey());  
 }

is O(n), where n is the size of the HashMap of ‘candidates’.

All other code in this snippet, aside from the code mentioned, is O(1).

Thus, O(n). Where n is the number of elements in the ‘candidates’ LinkedList.

**Space complexity**: O(n)

‘topCandidates‘ stores n elements, where n is the number of candidates, therefore it is O(n).