```
public void initializeCandidates(LinkedList<String> candidates){
   maxHeap = new PriorityQueue<>();
   nameToCandidate = new HashMap<>();
   this.candidates = new LinkedList<>(candidates);

   for(String candidateName: candidates){
       Candidate candidate = new Candidate(candidateName);
       nameToCandidate.put(candidateName, candidate);
       maxHeap.add(candidate);
   }
}
```

Time: O(NLogN) Space: O(N)
Time complexity is O(N) where N is the number of candidates in the Linked List. Taking a look at each loop, we see that the loop executes N times. Initializing a candidate is a constant operation, and adding to a hashmap is a constant operation. Inserting a new element into a max heap has a time complexity of logN, where N is the number of elements added into the heap. Therefore, for every N times the for each loop operates, maxHeap has to insert a new candidate and heapify. We know heapify is a logN operation so the time is NlogN.
The space complexity is O(N) where N is the number of candidates in the linked list. The maxHeap, HashMap, and candidates take O(N) space when initializing memory. Therefore, the time complexity is O(N) + O(N) + O(N) = O(N).

```
public void castVote(String candidate){
   if(nameToCandidate.get(candidate) == null ){
       System.out.println("Candidate " + candidate + " Does not Exist");

   }else {
       Candidate can = nameToCandidate.get(candidate);
       can.addVote();
       maxHeap.remove(nameToCandidate.get(candidate));
       maxHeap.offer(can);
   }
}
```

Time: O(N) Space: O(1)
The time complexity is O(N), where N is the number of elements/candidates in the maxHeap. The maxHeap takes O(N) time to remove. In the Java implementation, the maxHeap has to use an iterator to search the MaxHeap to find the element to be removed. It takes linear time to iterate the entire MaxHeap. The statements can.addVote() takes constant time. maxHeap.offer inserts a new element into the MaxHeap. The Java implementation says that it takes logN time to heapify. O(N) + O(logN) = O(N).

There is only one initialization and as the input increases the number of initializations remains constant. This is why space is O(1).

```java
public void castRandomVote(){
   Random ranGen = new Random();
   String ranCandidate = candidates.get(ranGen.nextInt(candidates.size()));
   castVote(ranCandidate);
}
```

Time: O(N)   Space: O(1)
The time complexity is O(N), where N is the number of Candidates in the Linked List. In Java, generating a random number takes O(1) time. The problem is that calling castVote() takes O(N) time. Because the amount of calls to O(N) doesn't increase with the size of N, the time complexity of castRandomVote is O(N)
The space complexity is O(1). Increasing the amount the input of candidates does not change the amount of initializations for renGen and castVote, also has space complexity of O(N).

```java
public void rigElection(String candidate, int numOfVotes){

   if(nameToCandidate.get(candidate) == null ){
       System.out.println("Candidate " + candidate + " Does not Exist");

   }else {
       int riggedVotes = numOfVotes/2 + 1;
       int diff = numOfVotes - riggedVotes;
       while(riggedVotes != 0){
           castVote(candidate);
           riggedVotes--;
       }
       while(diff != 0){
           castRandomVote();
           diff--;
       }
   }
}
```

Time: O(N*M)   Space: O(1)
The Time Complexity of the two while loops is O(N), where N is the numOfVotes. This is because both while loops are half of numOfVotes. This gives us O(n/2) + O(n/2) = O(N). As seen before, castVote() and castRandomVote() cost O(M) where M is the number of elements in the maxheap. Because we are calling castVote() and castRandomVote() N times, the time complexity is O(N*M).

Space Complexity is O(1) because the number of initializations made is constant as the input of candidates increases.

```
public void auditElection(){
   PriorityQueue<Candidate> temp = new PriorityQueue<>(maxHeap);
   int size = candidates.size();
   int votes;
   Candidate currCandidate;
   String name;
   int i = 0;
   while(i < size){
       currCandidate = temp.poll();
       name = currCandidate.getCandidateName();
       votes = currCandidate.getVotes();
       System.out.println(name + " - " + votes);
       i++;
   }
}
```

Time:O(N) Space: O(N)

The time complexity of the auditElection() is O(N), where N is the number of elements in the priority queue. poll() is a constant operation because it takes the first element from the queue. The while loop takes O(N) time to run due to the iteration through the inputs in the queue. Getname and getVotes are constant because it only returns a value that is stored in the class.

The space complexity is O(N), where N is the number of elements in the Priority Queue. As the number of Candidates increases, the larger the space needed is to initialize the temp priority queue.