1. Pass half of the string into the queue. If it’s an odd number of characters, ignore the central character so if a string has 7 characters, pass first 3 into the queue. Going backwards from the end of the string, do comparisons with the front of the queue. If the characters match, dequeue. If it’s a palindrome, the queue should be empty.
2. It’s better to use an iterator that corresponds to the last item in the list. For dequeue(), using an iterator corresponding to the last item makes it constant time access because there’s constant access to the front of the list from the last node so removing it is easy. For enqueue(), the iterator is already at the last node of the linked list so adding something to the end is constant time access. Peeking() is constant time because it relies on moving from the last node to the first node in constant time access. The Length() method is always constant time because it’s a variable held inside the queue.
3. Using two queues, it’s possible to implement the stack ADT. Use two queues, one named storage and the other helper to hold all the values a stack normally would. These two queues would switch roles because when you pop, you take all the values from the storage but one into the helper. Remove the last value in the storage queue then switch the names of the queues. Assuming the queue is implemented in a way where enqueue and dequeue are constant time functions, push is constant time because it’s adding to the storage queue. Pop is O(n) because it relies on passing all the values but one from the storage queue to the helper queue.
4. Void copy(deque D, queue Q){

For(i<4){

Q.enqueue(D.removeFront())

}

D.addBack(D.removeFront());

Q.enqueue(D.removeFront());

Q.enqueue(D.removeBack());

For(i<3){

Q.enqueue(D.removeFront());

}

}

1. 1. A circular array would be the most appropriate for this task because you have constant access to all members of the array, adding a new caller is constant, and removing the longest-waiting caller.
2. addNewCaller(obj Caller){

if(back.index+1==front.index){

resize();

}

Back.index=(back.index++)%array.size();

Array[back.index]=Caller;

removeLongest(){

int oldIndex=front.Index;

front.setIndex(front.index++%array.size());

delete Array[oldIndex];

}

accessCaller(location){

if(front.index+location<array.size-1){

return array[front.index+location]

}

Else{

Return array[front.index+location-array.size()]

}

1. I would not choose a different type of data structure. My only other option in this case would be to use a linked list. Although removing a node in the middle of the linked list is constant time, the actual searching for such a node is O(n) because if you have to iterate through the linked list. For a circular array, finding the node is constant time but removing it is O(n) because every value in the array needs to be moved over by 1 to cover up the empty spot in the array. So in total both of these data structures have the same runtime as each other.