PART B: CHECKING FOR CONVEXITY

No, it does not. This is because, while it is true that it does correctly determine a boundary of a convex polygon, but it is not necessarily in counter-clockwise order. One example of this would be if P is not a simple polygon (i.e. it has self-intersections) like the (n,3) family of star polygons

PART D: DEQUE ANALYSIS

My deque implementation uses a doubly-linked list where the deque struct has a pointer to a head and a tail node as well as a size variable, where each node has a pointer to the previous node, the point itself, and a pointer to the next node. This allows most deque operations to run in constant time, O(1), like size operation which would just return the size variable of the deque or the new deque operation, as I would not need to iterate through every single node to get to the top or bottom node (i.e. there is no for loop or while loop in the functions). However, the free operation would have to be O(n) because it has to go through the whole deque and free every single node before freeing the deque.

The push operation just makes the current head of the deque point to the new node and turn that into the new head of the deque, and since we are not iterating through the list, the time complexity is O(1). The insert operation is essentially the same operation, flipped, and would therefore also have a time complexity of O(1).

The pop operation would take the store the point variable of the current head of the deque, temporarily store that node, make the previous node that the current head points to the new head, then free the old head. This also would not require any iteration of the list and would therefore have a time complexity of O(1), and just like how the insert operation is like the flipped version of the push operation, remove function is the flipped version of the pop, and would therefore have a time complexity of O(1).

PART E: INSIDE HULL TRACING

ACTION	BOTTOM						TOP
NEW DEQUE – C A B C	<	C,	A,	B,	C,		>
POP – C	<	C,	A,	B,			>
PUSH – D	<	C,	A,	B,	D,		>
REMOVE – C	<		A,	B,	D,		>
INSERT – D	<	D,	A,	B,	D,		>
PUSH – E	<	D,	A,	B,	D,	E	>
REMOVE – D	<		A,	B,	D,	E	>
INSERT – E	<	E,	A,	B,	D,	E	>
POP – E	<	E,	A,	B,	D,		>
PUSH – F	<	E,	A,	B,	D,	F	>
REMOVE – E	<		A,	B,	D,	F	>
INSERT – F	<	F,	A,	B,	D,	F	>

POP – F	<	F,	A,	B,	D,			>
PUSH – G	<	F,	A,	B,	D,	G		>
INSERT – G	< G,	F,	A,	B,	D,	G		>
PUSH – H	< G,	F,	A,	B,	D,	G,	H	>
REMOVE – G	<	F,	A,	B,	D,	G,	H	>
REMOVE – F	<		A,	B,	D,	G,	H	>
REMOVE – A	<			B,	D,	G,	H	>
INSERT – H	<		H,	B,	D,	G,	H	>

PART G: INSIDE HULL ANALYSIS

For my convex hull algorithm, all the operations that it uses such as checking the orientation, popping, or pushing, except for free, run in constant time, and would therefore result in an overall time complexity depending on how many times we check if a particular point is part of the convex hull.

The initial creation of the deque with the first three points will always be constant and only requires checking once to test which direction it is in. We then ensure that the most recently added point is both at the top and the bottom of the deque. This then creates a connected edge between the first point and the third point, thus creating a triangle.

Continuing, we get to the while loop where we check the orientation of the other unknown points with the partial convex hull. Now, because we are assuming that P, the set of points in the polygon, refers to a simple polygon, and assuming that we follow the counter-clockwise direction, we can be assured that the unknown point will never be to the left of the edge between the first and third point of the convex hull and to the right of edge not connected to the most recently added point in the partial convex hull (third point) at the same time, i.e. the edge will never intersect. Therefore, we need only check to see if the unknown point is to the left of both the edges connected to the most recently added point to know if the unknown point is within the triangle. If it is in the triangle then it would not change the convex hull and we can proceed to the next point.

We need only change the convex hull if it appears to the right of either one or both of the edges connected to the most recently added point. Checking for this then pushing/inserting or popping/removing would run in constant O(1) time, and since we can observe that each point is pushed/inserted and popped/removed at most once, iterating through a simple polygon array of n points would give us a resulting time complexity of O(n), thus contradicting the statement where the best algorithms are of order O(n log n).

Having said all that, this all assumes that the points given make up a simple polygon. If not, we would have to correct the order of the points fed into the algorithm and that would require iterating through all the edges and checking against every other edge which would have left with a time complexity of $O(n^2)$ or O(n log n) at best, which would then be in support the runtime statement given.