day 8 theory

1A. A queue can only be added to on one end and accessed from the other.

1B. A stack can only be appended and accessed from one end — so only one item is visible.

② All insert and remove functions take at least O(n) time in an array list because the entire array must be re-written.
When implementing stacks or queues a link-backed list is preferable because pointers to the first and last links in the list make insertions and removals at either end take only constant time.

③ (a) O(n) time because a new array must be created and use a for loop to copy every value from the old list.

(b) O(n) time, maybe slightly slower than append because the for loop must also check each value it touches in addition to re-writing a new array.

(c) O(1) time because only an array[i]; call is required — the computer immediately knows where the ith value of the array lives in memory.

④ (a) O(1) time because the tail pointer immediately knows the location of the append.

(b) O(n) time if the value was not fetched by fetch current (like 4c)
O(1) time if the current pointer is still on the fetched value (like 4a)

(c) O(n) time because a for loop must start from an end and jump through links i times.

⑤ (a) O(n) because a loop must be used to retrieve every value and check it against the given value

(b) still O(n) because a loop must jump from link to link checking each value

day 8 theory

1A. A queue can only be added to on one end and accessed from the other.

1B. A stack can only be appended and accessed from one end — so only one item is visible.

② All insert and remove functions take at least O(n) time in an Array list because the entire array must be re-written.
When implementing stacks or queues a link-backed list is preferable because pointers to the first and last links in the list make insertions and removals at either end take only constant time.

③ (a) O(n) time because a new array must be created and use a for loop to copy every value from the old list.

(b) O(n) time, maybe slightly slower than append because the for loop must also check each value it touches in addition to re-writing a new array.

(c) O(1) time because only an array[i]; call is required — the computer immediately knows where the ith value of the array lives in memory.

④ (a) O(1) time because the tail pointer immediately knows the location of the append.

(b) O(n) time if the value was not fetched by fetch current (like 4c)
O(1) time if the current pointer is still on the fetched value (like 4a)

(c) O(n) time because a for loop must start from an end and jump through links i times.

⑤ (a) O(n) because a loop must be used to retrieve every value and check it against the given value

(b) still O(n) because a loop must jump from link to link checking each value

(5c) The upper bound for a BST is O(n) because it could be organized as one long branch — so one operation (constant time) only eliminates one value.

As the BST is organized in a more wide/complete manner (in which every level is filled), the relationship between ~~height~~ depth and width of a level looks like this:

depth ~~height~~	0 (root level)	1	2	3	4	5	6	7	...
width	1	2	4	8	16	32	64	128	...

The number of constant time operations required to sort a tree of depth ~~6~~ is ~~6~~ (the first op. moves the pointer to the children of the root) — but because each progressive level hold 2^{depth} values ~~then~~ each operation eliminates more and increases the "speed" of the search.

$$\text{time of 1 operation} = 2^{\text{depth}} \text{ values}$$

$$\text{ops. performed} = 2^{\text{ops}} \text{ values searched}$$

So as the size (and depth) increases, the average number of values searched by one operation increases — so the lower bound for a BST that is near complete approaches $\Omega(\log(n))$.

(5d) The more complete the tree, the closer it will be to logarithmic time because of the relationship between width of a level and values ~~searched~~ eliminated by each progressive operation.

A tree that is a long branch (giving O(n) time) is not complete — so a complete tree cannot have O(n) time, it will always be logarithmic. ~~pics~~

(6) Unless the BST has been loaded in some inauspicious way that makes it a single long branch, it will always have better than linear time (logarithmic plus some).

A sorted list using a binary sort could give similar results because like the search tree, it necessarily eliminates values without directly examining them by splitting the list and searching the half with larger or smaller numbers than the middle depending on the target.