1.

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Type | Name | Value |
| 0x000A | String[] | args | Received from caller |
| 0x000B | Point | s | 0x000H |
| 0x000C | Point | e | 0x000K |
| 0x000D | LineSegment | l | 0x000N |
| 0x000E |  |  |  |
| 0x000F |  |  |  |
| 0x000G |  |  |  |
| 0x000H | int | x | 0 |
| 0x000I | int | y | 0 |
| 0x000J |  |  |  |
| 0x000K | int | x | 5 |
| 0x000L | int | y | 10 |
| 0x000M |  |  |  |
| 0x000N | pointer | point a | 0x000H |
| 0x000O | pointer | point b | 0x000K |
| 0x000P |  |  |  |
| 0x000Q |  |  |  |
| 0x000R |  |  |  |
| 0x000S |  |  |  |
| 0x000T |  |  |  |
| 0x000U |  |  |  |
| 0x000V |  |  |  |
| 0x000W |  |  |  |
| 0x000X |  |  |  |
| 0x000Y |  |  |  |
| 0x000Z |  |  |  |

note: storage is allocated when you declare and then filled when you construct it

2.

A problem is the issue that a computer scientist has to solve, what they aim to achieve. An algorithm is like the method they use to do this, the specific “recipe” by which they get the final result. And a program is the final product “thing” that implements the algorithm in order to solve the problem.

3.

O(n^2) is the worst case time complexity so the actual amount of time it takes for each of the algorithms to complete sorting their list will depend on how sorted their initial list already is. Also, different types of sorting algorithms do better with differently arranged lists so this might also justify a difference in execution time between them even if the initial list given to them was exactly the same.

4.

No. The equals() method just compares the values of the things where as the “==” method compares the pointer objects first initialized in memory so they wouldn’t be equal unless they were exactly the same object (comparing point a to point a) OR equal primitive types (comparing 3 to 6/2). For example, in problem 1 above calling the equals() method on points s and e above would compare to see if (0,0) is the same as (5,10) where as the “==” method would compare values in the original Point objects “0x000H” and “0x000K.”

5.

Pre-Order Traversal: 14, 10, 6, 12, 15, 73, 21

In-Order Traversal: 6, 10, 12, 14, 15, 21, 73

Post-Order Traversal: 6, 12, 10, 12, 73, 15, 14

The binary tree is neither full nor complete. It’s not full because all nodes need to have either 0 or 2 children to be full and some nodes in this tree have only one child. It’s not complete because for it to be complete all layers of the tree must be filled except the leaf layer and in this tree both the leaf layer (with 21) and the layer above that (with 73) are not filled.

Yes, the tree seems like a binary search tree because: 1. the root has no parents, 2. every node has exactly 0, 1, or 2 children and 3. children to the left are smaller than their parent and children to the right are larger than their parent.

6.

(note I’m takeing the first node on a nodes list of neighbors to be the first one clockwise from 12:00 noon)

DepthFirstSearch: A->D, D->B, B->C, C->J, J->I, I->E

BreadthFirstSearch: G->D, D-B, B->E

OneCycle: A->G, G->A, etc. AnotherCycle: J->I, I->E, E->J, etc.

7.

Yes, I think Alex the bears solution will work. Because queues add onto the tail and take from the end the “enqueue(dequeue)” step will rebuild the list in the same order which is cool.

He forgot the step “size++” in the append() method, which will cause the size variable to inaccurately represent the number of items in the queue after each append(), but that’s the only mistake I see. The methods seem to work with all of the edge cases.

θ(n). Fetch() will have linear time complexity in relation to how many elements are in the queue, the size variable, and that is a tight bound.

θ(n). Remove() has exactly the same time complexity as fetch(). Both have this time complexity because the methods require us to sift through the entire list of items only one time.

θ(n+1). Insert() has the same linear time complexity as above plus constant time complexity to enqueue one more item. This is a tight bound.

It’s a good list implementation for problems for which you have to append a lot of new items. Other than that it is very similar to a linked list but it’s kind of nice that you don’t have to reset all the pointers and instead of keeping track of/updating a head and tail pointer, the enqueue() and dequeue() functions naturally do this for you.