Victor Zhu

1.

Space for a singly linked list O(1), whereas for the circular queue it would be O(n), where n is the size of the array. Linked list would mean every structure would only need a pointer field, whereas a circular queue requires an array to hold everything and more fields. I think the time for both should be the same, since you would be traversing in order for the linked list so no need for random access. The time for both should be O(1).

2.

Worst case time for singly linked list hash table would be O(n-squared), and average time would be O(1). The worst case time for the binary search tree would be O(nlog(n)), since it would be the same as the worst time for a binary search tree if all values were put into the same bucket. Average case time would be O(1) since it has to be faster than the singly linked one. When it is set to 3, the worst case bucket size would be close to the average case bucket size.

3.

I would implement quadratic probing for closed hashing, and after the number of positions reaches around more than half the number of buckets, I would create a new hash table double the size and rehash to it, and continue hashing. Compared with opening hashing, the space needed would be O(n) , whereas the space needed for open hashing would be O(1). Speed would be O(1) for both of them.

4

In terms of time, I think the fastest one would be the third alternative presented in the question, since it would simply print from the stack formed. The second approach involves using a stack to keep track of the pointers, but this involves retraversing the stack afterwards. The first way, which is what we did, is the least efficient because we have to create more fields and operations to do the same things. The second approach has the pro of returning a stack with all the pointers, which could be used for other purposes. The third method would be simple and quick, but we wouldn't be able to make other use of the pointers. First version is convenient because we can now traverse in order the objects as we please.

5

When a position *X* is dequeued, the program could not generate the position Y that results moving back in the same direction from which X came, or it could generate all positions reachable from X and discover that Y was already in the hash table. Discuss the pros and cons of these alternatives.

For the first option, all that would happen is referencing the back pointer of X, and then using an if statement to check that Y won't be created. There would be no need to run the hash function and search through the hash table. The problem is every possibility you generate would have to be compared against the back pointer Y.

For the second alternative, the speed would be dependent on how quickly the hash table can check our Y, so it would potentially be slower, but you would only do a hash search once for Y.

6

You would have multiple characters be considered as a single object. So everytime any move or change is made to one index, you would have to find the index of all the other parts of the bigger object, then check to see if those indexes can all be moved into the new spots. It would involve more tedious coding, but the same idea applies for finding all possible movements.

7

For the straight one, instead of only checking for one move like in my original code, I would check for moving 2, 3, and up til wherever I can no longer move it to. It would generate more possibilities per loop at a time. For rectilinear, it would be an extension of the straight one, except I would also check for possibilities where after moving in one direction in any amount, I move in a different direction any amount. For example, if I move piece to all the north possibilities, for every north possibility I would check all the possible east and west possibilities. Unlike the original code where each step is only able to generate one piece movement, each step in the straight and rectilinear versions would generate far more possibilities, making those possibilities have a lower step count than for the single movement version.

8.

The most difficult part of this assignment was understanding C, since its my first time doing a major coding assignment in a language other than java. Multiple files would have helped with cleanliness, readability, and modularity, but it wouldn't have saved much time for me since my biggest problem was with pointers. If I could divide it up, everyone would be trying to code it at the same time, because many times people will get similar bugs. Some people will figure out how to solve some bugs, some will figure out other bugs, so they can all help each other debug each others code. Having everyone person individually trying to do it would allow for shared understanding of the project.