**1)**

I) An element is being added to the list.

II) In programming, an element is added to a linked list by setting the “next” pointer in the previous element to the new node, and the “next” pointer in the new node is set to the next element in the linked list. Before the “next” pointer is changed in the previous node, though, the current “next” pointer of the previous node must be saved. In the diagram, the value “nodePtr” would save this value. Thus, the C++esque pseudocode would be:

nodePtr = previousNode->next;

previousNode->next = newNode; //newNode is a pointer

newNode->next = nodePtr;

**2)** Linked Lists are different from stacks because adding and removing elements in the middle (rather than end or “top”) is still possible. There is also usually an implementation difference, in that stacks can be implemented with contiguous memory but Linked Lists never are as doing so would rid them of all their benefits. To give an example, in deprecated immediate mode OpenGL there is a “transformation matrix stack” of matrix transforms which are all applied to vertices before rendering to the screen. After you have pushed a matrix onto the stack, it is impossible to change anything “underneath” it until it has been popped off. This means that for every object, if you wish to render every object from multiple camera angles, you must first push on the cameraspace to screenspace transform, then the worldspace to cameraspace transform, then the modelspace to worldspace transform and then render the first camera angle. Then, you must pop all these transforms off again and repeat the process for the cameraspace→screenspace transform and worldspace→cameraspace transform for the second camera angle, and render the mesh, so on and so forth. This means that the modelspace→worldspace transform is being pushed on multiple times when it doesn’t need to be changed.

With a linked list or other storage method allowing for alterations in the middle, redundant changes of the Modelspace→worldspace transform are no longer necessary. Changing the camera transforms without changing the model transform is entirely possible.

**3)**

a) This is my attempt to interpret the algorithm as written:

Pre-loop:

Stack: (

infix String: (6+2) \* 5-8/4)

Postfix String:

Iteration 1)

Stack: ( (

Infix String (LEFT TO PROCESS): 6+2)\*5-8/4)

Postfix String:

Iter 2)

Stack: ( (

IF: +2)\*5-8/4)

PF: 6

Iter 3)

Stack: ((+

IF: 2)\*5-8/4)

PF: 6

Iter 4)

Stack: ((+

IF: )\*5-8/4

PF: 6\_2 (NOTE: underscores are used to seperate numbers for clarity, spaces would be the most reasonable character to use)

Iter 5)

Stack: (

IF: \*5-8/4)

PF: 6\_2+

Iter 6)

Stack: (\*

IF: 5-8/4)

PF: 6\_2+

Iter 7)

Stack:(\*

IF: -8/4)

PF: 6\_2+5

Iter 8)

Stack: (-

IF: 8/4)

PF: 6\_2+5\*

(Multiplication is higher precedence than subtraction)

Iter 9)

Stack: (-

IF: /4)

PF: 6\_2+5\*8

Iter 10)

Stack: (-/

IF: 4)

PF: 6\_2+5\*8

Iter 11)

Stack: (-/

IF: )

PF: 6\_2+5\*8\_4

Iter 12)

Stack:

IF:

PF:6\_2+5\*8\_4/-

b) Here is my attempt to parse the postfix string generated so that it generates a correct answer.

The postfix string is 6\_2+5\*8\_4/-

VALUE STACK:

OPERATOR:

Iter 1)

VALUE STACK: 6

OPERATOR:

Iter 2)

VALUE STACK: 6 2

OPERATOR:

Iter 3)

VALUE STACK: 6 2

OPERATOR: +

Iter 3 part 2, Evaluate)

VALUE STACK: 8

OPERATOR:

Iter 4)

VALSTACK: 8 5

OPS:

Iter 5)

VALSTACK: 8 5

OPS: \*

Iter 5 part 2, Evaluate)

VALSTACK: 40

OPS:

Iter 6)

VALSTACK: 40 8

OPS:

Iter 7)

VALSTACK: 40 8 4

OPS:

Iter 8)

VALSTACK: 40 8 4

OPS: /

Iter 8 part 2, Evaluate)

VALSTACK: 40 2

OPS:

Iter 9)

VALSTACK: 40 2

OPS: -

Iter 9 part 2, Evaluate)

VALSTACK: 38

OPS:

Return 38;

4)

a)

Elements are added to a Queue at the end, and come off the beginning. First in, First out. It took me quite a few minutes to figure out what you meant by “positions where elements are removed can be used to add elements to the queue [Take into consideration insertions past the last array index]” and I am still unsure if my interpretation is correct, however I believe I have a proposed system that fits this.

One could construct a contiguous queue that operates in a “circular” manner by deciding beforehand how long the maximum size of this queue can be. The queue itself would store 3 Unsigned Integer values, Size, NextAccess and NextWrite.

NextIndex and NextWrite would start equal to zero, and size would be given by the constructor.

When pushing, Array[NextWrite] is written to. NextWrite is incremented, and NextWrite %= Size;

When popping, Array[NextAccess] is returned. NextAccess is incremented and NextAccess %= Size;

This would result in NextAccess “following” NextWrite, circularly, going across the length of the array. Insertions past the last array index simply wrap around. Places where elements were removed previously can be written to again because NextWrite will eventually reach it once again.

This also means that attempting to pop non-existent values from the queue will never return an error, although it may return unexpected values unless elements are set to some default value initially and when popped.

b) Using Circular Queues and clearing default and popped values to 0:

[5][0][0][0]

[5][7][0][0]

[0][7][0][0]

[0][7][9][0]

[0][7][9][12]

[0][0][9][12]

[10][0][9][12]

Using traditional “infinite” Queue structure:

[5]

[5][7]

[7]

[7][9]

[7][9][12]

[9][12]

[9][12][10]