DoubleStack.java

- Read the first line, store it in an int variable, intInput.
- Create an array, doubleStack of size intInput.
- Read the next line as String, check if it is “—red”, “--blue”, set the color to red or blue respectively. If it is “?” or “-“, peek or pop from the stack respectively.
- If the input is not “—red”, “--blue”, “?” or “-“, push onto the respective stack(red or blue). The color is Red by default.
- In method push, if the stack is full, throw a RuntimeException. Else add the element onto the respective stack (red or blue) as specified in the parameters of push method.
- To add to Red stack, insert elements from the front of the doubleStack array while incrementing the variable, rFirst, which is the index of the red array, keeping track of the newest/latest element added.
- To add to Blue stack, insert elements from the back of the doubleStack array while decrementing the variable, bFirst, which is the index of the blue array, keeping track of the newest/latest element added.
- In method pop, print the latest element pushed, decrement the index and set that element to null. If the stack is empty
- In method peek, print the latest element pushed into the stack.

Infix2Prefix.java

- Extend class Infix2Prefix by DoubleStack
- Read the line and store it in an array called “in” using the split function.
- Start examining the array “in” from the back using a for loop until the end of the input is reached or until in[0] has been examined.
- If in[i] is an operand (i.e, not an operator(*,/,+ or -)) and is not a closed parenthesis or an open parenthesis then, push to the blue stack (for reference to blue stack or push method see: DoubleSack.java)
- If in[i] is a closed parenthesis, push to the red stack (for reference to red stack or push method see: DoubleSack.java)
- If in[i] is an operator (i.e *, /, + or -) then check if the red stack is empty, if it is empty, push the operator at in[i] onto the red stack. Else if peek(Color.RED) (i.e top of the stack) is a closed parenthesis, push the operator at in[i] onto the red stack. Else if the operator at in[i] has same/higher priority as compared to the operator at the top of the stack (returned by peek method from DoubleStack.java), push the operator at in[i] onto the red stack; else pop the operator at the top of the red stack and push it onto the blue stack and push the operator at in[i] ontp the red stack checking all the conditions again.
- If the operator at in[i] is an open parenthesis, pop elements of the red stack and push them onto the blue stack till a closed parenthesis is encountered. Pop the closed parenthesis making sure not to push it onto the blue stack.

- Once the entire input has been processed, pop the blue stack and save it in an "output" string.
- Pop the remaining operators, if any, and save them in a "temp" string.
- Reverse the "temp" string and add it in front of the "output" string.
- This final string is the final output in prefix form.

EvalPrefix.java

- Extend the EvalPrefix class by DoublStack.
- Read line from the input and store it in an array.
- Create a DoubleStack object of size of the array.
- Process each element in the array from reverse.
- Check if the element of the array is an operator (i.e, +, -, * or /), if not push the element onto the stack. (I am only using one stack (Red) for this part. I am not using the blue stack of the DoubleStack client).
- If the element is an operator (i.e, +, -, * or /), store it in a string "operator".
- Pop from the stack and parse it into an int "operand1".
- Pop from the stack again and parse it into an int "operand2"
- (Thus by popping twice, we have access the top two elements/numbers in the stack.)
- Call the "calculate" method and pass "operator", "operand1" and "operand2" as parameters of "calculate".

- In the method “calculate”, if the “operator is” *, /, - or +, perform multiplication, division, subtraction or addition of “operand1” and “operand2” respectively and store it in an int “result”.
- Parse “result” into string and push it onto the red stack.
- (This way the result pushed is the result of the two operands at the latest operator encountered. When the next operator is encountered, the new calculation will occur between the result (just pushed) and the next element. The result between the two will again be pushed onto the red stack. This process will continue till the last element of the input array and been examined. In this way the element at the top of the stack will be the final evaluation of the prefix notification inputted.)
- Once all the elements of the input array have been processed, pop the remaining element from the stack and print it, this is the result of the prefix notification given.

MyQueue.java

- Read String from the input given; check if it is not an operator (i.e, -, * or ?) enqueue it in the queue. The data structure implementation of the queue is a linked list.
- In the method enqueue, declare a node “oldlast” and set it equal to the node “last” and “oldlast.next” equal to “last”. Set the value of “last.next” to null and the value of the item in node “last” as that passed as the parameter of

enqueue. If the linked list was empty then set the “first” node equal to the “last” node.

- (This way each new enqueue operation occurs at the end of the linked list. As queues are FIFO so it is important to keep track of the first element that was enqueued.)
- Perform the next read string, if it is an operator (i.e, -, * or ?), perform dequeue, peek or lookup respectively.
- In the method dequeue, if the linked list is empty throw a RuntimeException. Else save the “first” node in a temporary node and set the value of “first” as “first.next”. Return the temporary node, which is the node that was at the beginning of the queue initially.
- In method peek, if the queue is empty throw a RuntimeException. Else return the item in the “first” node.
- The method lookup returns the item in the node at the position specified in the input after the “?” symbol.
- My lookup method has a constant time complexity as every time an enqueue or dequeue operation is performed, my algorithm stores a copy of the item in the node being enqueued in an array which is corresponding to the number of the node in the linkedlist/queue. Every time a dequeue operation is performed, my algorithm sets the value of corresponding element in the array to null as that node as been dequeued. Thus, when the lookup operation is performed, my algorithm returns the corresponding array element. Hence my lookup function has a constant time complexity as

opposed to incrementing through the entire queue till the elements to be lookedup is returned, that will be a O(n) time complexity.