The Brogrammers (Reagan Leonard, Jackson Lee, Jack Sparrow)

Test Plans

- Reagan
    - Queue_Template_Test_Plan (assuming Max_Length = 5)
        - Enqueue
            - Test 1 (lower boundary/typical case):
                - Input: E = 3, Q = < >
                - Expected Output: E = <3>, Q = <3>
            - Test 2 (upper boundary case):
                - Input: E = 5, Q = <1, 2, 3, 4>
                - Expected Output: E = <5>, Q = <1, 2, 3, 4, 5>
            - Test 3 (requires clause not met):
                - Input: E = 12, Q = <4, 6, 2, 9, 10>
                - Expected Output: Error, Max_Length exceeded!
        - Dequeue
            - Test 1 (requires clause not met):
                - Input: R = < >, Q = < >
                - Expected Output: Error, Q must contain an element!
            - Test 2 (lower boundary case):
                - Input: R = < >, Q = <2>
                - Expected Output: R = <2>, Q = < >
            - Test 3 (typical case):
                - Input: R = < >, Q = <5, 6, 8>
                - Expected Output: R = <5>, Q = <6, 8>
        - Swap_First_Entry
            - Test 1 (lower boundary)
                - Input: E = 3, Q = <1>
                - Expected Output: E = <1>, Q = <3>
            - Test 2 (upper boundary)
                - Input: E = 17, Q = <6, 9, 12, 18, 4>
                - Expected Output: E = <6>, Q = <17, 9, 12, 18, 4>
            - Test 3 (typical case)
                - Input: E = 5, Q = <3, 4, 5>
                - Expected Output: E = <3>, Q = <5, 4, 5>
        - Length
            - Test 1 (empty queue)
                - Input: Q = < >
                - Expected Output: Length = 0
            - Test 2 (lower boundary)
                - Input: Q = <9>
                - Expected Output: Length = 1
            - Test 3 (upper boundary)
                - Input: Q = <2, 4, 6, 8, 10>
                - Expected Output: Length = 5

    - Rem_Capacity
      - Test 1 (empty queue)
        - Input: Q = < >
        - Expected Output: Rem_Capacity = 5
      - Test 2 (typical case)
        - Input: Q = <2, 3, 4>
        - Expected Output: Rem_Capacity = 2
      - Test 3 (full queue!)
        - Input: Q = <6, 7, 8, 9, 10>
        - Expected Output: Rem_Capacity = 0
    - Clear
      - Test 1 (queue already empty)
        - Input: Q = < >
        - Expected Output: Q = < >
      - Test 2 (typical case)
        - Input: Q = <1, 4, 7>
        - Expected Output: Q = < >
      - Test 3 (upper boundary)
        - Input: Q = <10, 15, 20, 25, 30>
        - Expected Output: Q = < >
- Jackson
  - Search_Store_Template(assuming Max_Capacity = 5)
    - Add (restores k: Key; updates S: Store);
      - Lower Boundary
        - Input: k = 3, S = {};
        - Output: k = 3, S = {3};
      - Upper boundary
        - Input: k =7, S = {1,2,3,4}
        - Output: k = 7, S = {1,2,3,4,7}
      - Requires clause not met
        - Input: k = 3, S = {1,2,3,4}
        - Output: Error, k cannot already exist in S
    - Remove (restores k: Key; updates S: Store);
      - Lower boundary
        - Input: k =3, S = {3}
        - Output: k=3, S={}
      - Typical case
        - Input: k =2, S = {1,2,3,4}
        - Output: k = 2, S = {1,3,4}
      - Requires Clause not met
        - Input: k = 4, S = {1,2,3}
        - Output: Error. K must be within S
    - Remove_Any (replaces k: Key; updates S: Store);
      - Lower boundary
        - Input: k=3, S = {3}

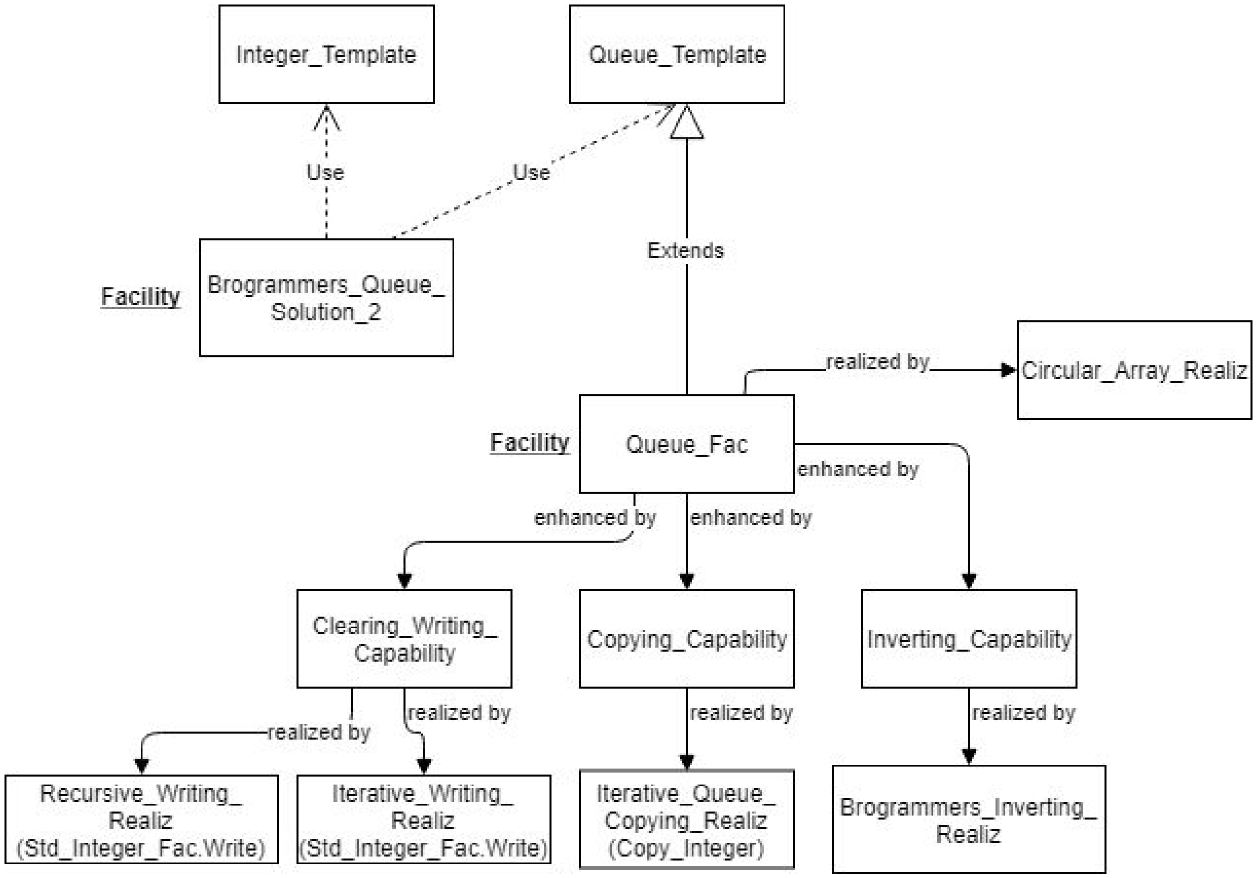
        - Output: k = 3, S = {}
    - Upper Boundary
        - Input: k = 5, S = {1,2, 3,4,5}
        - Output: k=5, S= {1,2,3,4}
    - Difficult case
        - Input: k = 3, S = {1,2}
        - Output: k = 3, S ={1,2}

- Is_Present(restores k: Key; restores S: Store): Boolean;
    - Lower Boundary
        - Input: k = 3, S={3}
        - Output: True.
    - Typical Case
        - Input: k =4, S = {1,2,3}
        - Output: False.
    - Upper Boundary
        - Input: k =2, S = {1,2,3,2,4}
        - Output: True
- Key_Count (restores S: Store): Integer;
    - Lower Boundary
        - Input: S= {}
        - Output: 0
    - Upper Boundary
        - Input: S = {5,5,5,5,5}
        - Output: 5
    - Typical Case
        - Input: S= (2,2,2)
        - Output: 3
- Rem_Capacity (restores S: Store): Integer;
    - Lower Boundary
        - Input: S= {}
        - Output: 5
    - Upper Boundary
        - Input: S = {5,5,5,5,5}
        - Output: 0
    - Typical Case
        - Input: S= (2,2,2)
        - Output: 2
- Clear (clears S: Store);
    - Lower Boundary
        - Input: S= {}
        - Output: S = {}
    - Upper Boundary
        - Input: S = {5,5,5,5,5}
        - Output: S = {}

      - Typical Case
        - Input: S= (2,2,2)
        - Output: S = {}
- Jack
  - Globally_Bounded_List_Template
    - Advance (updates P: List);
      - Test 1: Requires Clause Not Met
        - Input: P = <3, 4, 5>; P.Prec = <3, 4, 5>; P.Rem = < >
        - Output: Error! P.Rem must contain an entry!
      - Test 2: P.Rem Lower Boundary
        - Input: P = <8, 9,10, 7>; P.Prec = <8, 9, 10>; P.Rem = <7>
        - Output: P = <8, 9, 10, 7>; P.Prec = <8, 9, 10, 7>; P.Rem = < >
      - Test 3: P.Prec Lower Boundary
        - Input: P = <1, 2, 3, 4, 5>; P.Prec = < >; P.Rem = <1, 2, 3, 4, 5>
        - Output: P = <1, 2, 3, 4, 5>; P.Prec = <1>; P.Rem = <2, 3, 4, 5>
    - Reset (updates P: List);
      - Test 1: P.Prec Is Empty (lower boundary)
        - Input: P = <3, 4, 5>; P.Prec = < >; P.Rem = <3, 4, 5>
        - Output: P = <3, 4, 5>; P.Prec = < >; P.Rem = <3, 4, 5>
      - Test 2: P.Prec Upper Boundary
        - Input: P = <6, 7, 8, 9>; P.Prec = <6, 7, 8, 9>; P.Rem = < >
        - Output: P = <6, 7, 8, 9>; P.Prec = < >; P.Rem = <6, 7, 8, 9>
      - Test 3: Typical Case
        - Input: P = <7, 7, 7, 7, 7>; P.Prec = <7, 7, 7>; P.Rem = <7, 7>
        - Output: P = <7, 7, 7, 7, 7>; P.Prec = < >; P.Rem = <7, 7, 7, 7, 7>
    - Is_Rem_Empty (restore P: List): Boolean;
      - Test 1: P.Rem Lower Boundary
        - Input: P = <7, 5, 7>; P.Prec = <7, 5, 7>; P.Rem = < >
        - Output: True
      - Test 2: P.Rem Upper Boundary
        - Input: P = <8, 8>; P.Prec = < >; P.Rem = <8, 8>
        - Output: False
      - Test 3: Typical Case
        - Input: P = <5, 4, 3, 2>; P.Prec = <5, 4>; P.Rem = <3, 2>
        - Output: False
    - Insert (alters New_Entry: Entry; updates P: List);
      - Test 1: Empty List
        - Input: New_Entry = <1>; P = < >; P.Prec = < >; P.Rem = < >

        - Output: New_Entry = <0>; P = <1>; P.Prec = < >; P.Rem = <1>
    - Test 2: P.Rem Lower Boundary
        - Input: New_Entry: <9>; P = <3, 3, 3>; P.Prec = <3, 3, 3>; P.Rem = < >
        - Output: New_Entry: <0>; P = <3, 3, 3, 9>; P.Prec = <3, 3, 3>; P.Rem = <9>
    - Test 3: Typical Case
        - Input: New_Entry = <4>; P = <1, 2, 3, 5, 6>; P.Prec = <1, 2, 3>; P.Rem = <5, 6>
        - Output: New_Entry = <0>; P = <1, 2, 3, 4, 5, 6>; P.Prec = <1, 2, 3>; P.Rem = <4, 5, 6>
- Remove (replaces Entry_Removed: Entry; updates P: List);
    - Test 1: Requires Clause Not Met
        - Input: P = <1, 2, 3>; P.Prec = <1, 2, 3>; P.Rem = < >; Entry_Removed = < >
        - Output: Error! P.Rem must contain an entry!
    - Test 2: Typical
        - Input: P = <18, 9, 3, 1>; P.Prec = <18>; P.Rem = <9, 3, 1>;
        - Output: P = <18, 3, 1>; P.Prec = <18>; P.Rem = <3, 1>; Entry_Removed = <9>
    - Test 3: P.Rem Lower Boundary
        - Input: P = <7, 7, 7, 7, 7>; P.Prec = <7, 7, 7, 7>; P.Rem = <7>
        - Output: P = <7, 7, 7, 7>; P.Prec = <7, 7, 7, 7>; P.Rem = < >; Entry_Removed = <7>
- Advance_to_End (updates P: List);
    - Test 1: P.Prec Lower Boundary
        - Input: P = <6, 6, 6>; P.Prec = < >; P.Rem = <6, 6, 6>
        - Output: P = <6, 6, 6>; P.Prec = <6, 6, 6>; P.Rem = < >
    - Test 2: P.Prec Upper Boundary
        - Input: P = <8, 8, 8, 9>; P.Prec = <8, 8, 8, 9>; P.Rem = < >
        - Output: P = <8, 8, 8, 9>; P.Prec = <8, 8, 8, 8>; P.Rem = < >
    - Test 3: Typical
        - Input: P = <3, 4, 5>; P.Prec = <3>; P.Rem = <4, 5>
        - Output: P = <3, 4, 5>; P.Prec = <3, 4, 5>; P.Rem = < >
- Swap_Remainders (updates P, Q: List);
    - Test 1: P.Rem & Q.Rem Lower Boundaries
        - Input: P = <2, 2, 2>; P.Prec = <2, 2, 2>; P.Rem = < >; Q = <1, 1, 1>; Q.Prec = <1, 1, 1>; Q.Rem = < >
        - Output: P = <2, 2, 2>; P.Prec = <2, 2, 2>; P.Rem = < >; Q = <1, 1, 1>; Q.Prec = <1, 1, 1>; Q.Rem = < >
    - Test 2: P.Rem & Q.Rem Upper Boundaries

        - Input: P = <7, 7, 7, 7>; P.Prec = < >; P.Rem = <7, 7, 7, 7>; Q = <6, 6, 6>; Q.Prec = < >; Q.Rem = <6, 6, 6>
        - Output: P = <6, 6, 6>; P.Prec = < >; P.Rem = <6, 6, 6>; Q = <7, 7, 7, 7>; Q.Prec = < >; Q.Rem = <7, 7, 7, 7>;
    - Test 3: Typical
        - Input: P = <2, 4, 6, 8>; P.Prec = <2, 4>; P.Rem = <6, 8>; Q = <1, 3, 5>; Q.Prec = <1, 3>; Q.Rem = <5>
        - Output: P = <2, 4, 5>; P.Prec = <2, 4>; P.Rem = <5>; Q = <1, 3, 6, 8>; Q.Prec = <1, 3>; Q.Rem = <6, 8>
- Is_Prec_Empty (restores P: List): Boolean;
    - Test 1: P.Prec Lower Boundary
        - Input:
        - Output:
    - Test 2: P.Prec Upper Boundary
        - Input:
        - Output:
    - Test 3: Typical
        - Input:
        - Output:
- Clear (clears P: List)
    - Test 1: Empty List
        - Input: P = < >; P.Prec = < >; P.Rem = < >
        - Output: P = < >; P.Prec = < >; P.Rem = < >
    - Test 2: One Entry List
        - Input: P = <2>; P.Prec = < >; P.Rem = <2>
        - Output: P = < >; P.Prec = < >; P.Rem = < >
    - Test 3: Typical
        - Input: P = <6, 8, 0>; P.Prec = <6, 8>; P.Rem = <0>
        - Output: P = < >; P.Prec = < >; P.Rem = < >



Integer_Template
Queue_Template
Use
Use
Extends
Facility
Brogrammers_Queue_ Solution_2
realized by
Circular_Array_Realiz
Facility
Queue_Fac
enhanced by
enhanced by
enhanced by
Clearing_Writing_ Capability
Copying_Capability
Inverting_Capability
realized by
realized by
realized by
realized by
Recursive_Writing_ Realiz (Std_Integer_Fac.Write)
Iterative_Writing_ Realiz (Std_Integer_Fac.Write)
Iterative_Queue_ Copying_Realiz (Copy_Integer)
Brogrammers_Inverting_ Realiz

(Here's all of our final code, *juuust* in case!)

```
Facility Brogrammers_Queue_Solution_2;
    uses Integer_Template, Queue_Template;

    Operation Copy_Integer(replaces C: Integer; restores Orig: Integer);
                    ensures C = Orig;
            Procedure
                    C := Orig;
            end Copy_Integer;

    Facility Queue_Fac is Queue_Template(Integer, 3)
            realized by Circular_Array_Realiz
        enhanced by Inverting_Capability
            realized by Brogrammers_Inverting_Capability
        enhanced by Copying_Capability
            realized by Iterative_Queue_Copying_Realiz(Copy_Integer)
        (*enhanced by Clearing_Writing_Capability
            realized by Iterative_Writing_Realiz(Std_Integer_Fac.Write)*)
        (*enhanced by Clearing_Writing_Capability
            realized by Recursive_Writing_Realiz(Std_Integer_Fac.Write)*);


--This operation was completed by Reagan
    Operation Rotate (updates Q: Queue);
        requires 1 <= |Q|;
        ensures Q = Prt_Btwn(1, |#Q|, #Q) o Prt_Btwn(0, 1, #Q) and 1 <= |Q|;

        Procedure
        Var R: Integer;

        Dequeue(R, Q);
        Enqueue(R, Q);

    end Rotate;
(*I had several repeated errors on line 19 and 23. I was originally
overthinking this code and trying to put a loop in here until I
realized that was completely unnecessary.*)

--This operation was completed by Jackson
    Operation Split_Into(clears P: Queue; replaces E: Integer; replaces Q: Queue);
        requires 1 <= |P|;
        ensures #P = Q o <E>;
    Procedure
        Dequeue(E,P);
```

```
    Clear(Q);
      While ( 1 <= Length(P) )
            maintaining #P = Q o <E> o P;
            decreasing |P|;
      do
    Enqueue(E,Q);
        Dequeue(E,P);

            end;
  end Split_Into;
(*my difficulty with this part was largely in figuring out where to place
the E and Q values, I at first had the idea to place the Dequeue below the
while statement and change the while statement to 1 < Length(P) but i
quickly realized this cause problems for the body of the while loop and was
overall unnecessary due to the E value being set to the last value in P
automatically.*)

--This operation was completed by Jack
  Operation Combine (updates P: Queue; alters E: Integer; clears Q: Queue);
    --Combines the contents of two Queues with a new entry between them
    requires |P| + 1 + |Q| <= 3;
    ensures P = #P o <#E> o #Q;

    Procedure
    --Place E onto P
    Enqueue(E, P);

    --This while loop goes through Q taking its entries and
    --putting them onto the end of P
      While(1 <= Length(Q))
        maintaining #P o <#E> o #Q = P o Q;
        decreasing |Q|;
      do
        Dequeue(E, Q);
        Enqueue(E, P);
      end;
  --I had an issue concerning the maintaining specification I was
  --using for my While loop. I had originally tried using temporary
  --variables and kept running into problems with that particular line.
  --So I simplified my code and then Enqueued E onto P earlier so that
  --I would be able to use E to move entries from Q onto P. My
  --maintaining statement was simplified and proved immediately.

    end Combine;
```

```
--This operation was completed by Reagan
   Operation Read_Queue(replaces Q: Queue);
   --informally ensures that Q has inputs in reverse order
      requires |Q| <= 3;
      ensures 1 <= |Q|;
         Procedure
                  Var E: Integer;

      Clear(Q);
                  While ( 1 <= Rem_Capacity(Q) )
                           decreasing 3 - |Q|;
                  do
         Read(E);
                           Enqueue(E, Q);
                  end;

         end Read_Queue;
```

(*This was very difficult for me. At first, I didn't know how to complete this using only Enqueue and Dequeue. Then I realized it would work if I used the Inject_at_Front enhancement. I also struggled with what the invariant should be until I realized that I couldn't really be more specific than a vague one (using length). Then I had to change the loop condition to make sure I was checking that I had enough room in T and that Q wasn't getting too small. Then it worked!*)

```
--This operation was completed by Jackson
 Operation Write_Queue(clears Q:Queue);
      -- informally ensures output has the contents of #Q in order
      requires 1 <= |Q|;
   Procedure
                  Var T: Queue;
      Var A: Integer;
      Dequeue(A,Q);
                  While ( 1 <= Length(Q) )
         maintaining #Q = T o <A> o Q;
                           decreasing |Q|;
                  do
         Enqueue(A, T);
                           Dequeue(A, Q);

                  end;

         end Write_Queue;
```

```
(*
   Operation Copy_Queue(restores Q : Queue; replaces Q_Copy:  Queue);
   ensures Q_Copy = Q;
   Procedure
   Var E: Entry;
   Var T: Queue;
   Clear(Q_Copy);
   Dequeue(E, Q);
      While ( 1 <= Depth(Q) )
                     maintaining #Q = T o Q;
                     decreasing |Q|;
               do
         Enqueue(E, T);
         Enqueue(E, Q_Copy);
                     Dequeue(E, Q);
               end;
   Q :=: T;

   end Copy_Queue;
*)

--This operation was completed by Jack
   Operation Main();
   --Main program that calls all five operations
   Procedure
      Var P: Queue;
      Var E: Integer;
      Var Q: Queue;

      Read_Queue(P);

      --Rotates P so its first Integer is at the end
      Rotate(P);

      --Takes P's last Integer and puts it in E while
      --Taking the rest of P's contents and puts them into Q
      Split_Into(P, E, Q);

      --Combines contents of the empty queue P, element E, and Q
      --This should restore the contents of P to #P
      Combine(P, E, Q);

      Invert(P);
```

```
        Copy_Queue(P, Q);

        --Makes sure output for P is in correct order
        Write_Queue(P);

    end Main;

end Brogrammers_Queue_Solution_2;
```