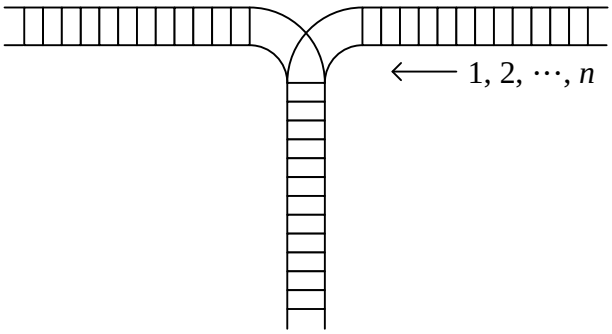
EECS2040 Data Structure Hw #2 (Chapter 3 Stack/Queue)
due date 4/11/2022
by 108011235 陳昭維

Part 1 (2% of final Grade)

1. (10%) consider the railroad switching network shown below. (textbook pp.138-139)



Railroad cars can be moved into the vertical track segment one at a time from either of the horizontal segments and then moved from the vertical segment to any one of the horizontal segments. The vertical segment operates as a **stack** as new cars enter at the top and cars depart the vertical segment from the top. Railroad cars numbered 1,2,3…, n is initially in the top right track segment. Answer the following questions for n=3 and 4 cases:

(a) What are the possible permutations of the cars that can be obtained?

<answer>
For n = 3, the possible permutations are (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,2,1)
For n = 4, the possible permutations are
(1,2,3,4), (2,1,3,4), (3,2,1,4), (4,3,2,1)
(1,2,4,3), (2,1,4,3), (3,2,4,1)
(1,3,2,4), (2,3,1,4), (3,4,2,1)
(1,3,4,2), (2,3,4,1),
(1,4,3,2), (2,4,3,1)

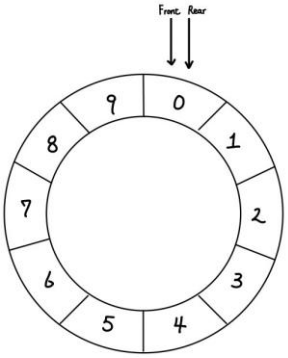
(b) Are any permutations not possible? If no, simply answer no. If yes, list them all.

<answer>
Yes, for both n = 3 and n = 4 there are impossible permutations
For n = 3, the impossible permutation is (3,1,2)
For n = 4, the impossible permutation is (1,4,2,3), (2,4,1,3), (3,1,2,4), (4,1,2,3)
(3,1,4,2),(4,1,3,2)
(3,4,1,2),(4,2,1,3)
,(4,2,3,1)
,(4,3,1,2)

2. (15%) A linear list of type T objects is being maintained circularly in an array with front and rear set up as for **circular queues**.

(a) Draw a diagram of the circular array showing the initial status (values of front, rear) when the linear list is created (constructed) with no elements stored yet (assume capacity = 10 is used for creating the array)

<answer>



(b) Obtain a formula in terms of the array capacity, front, and rear, for the number of elements in the list.

<answer>

$$\#\text{of elements} = \begin{cases} \text{rear} - \text{front}, & \text{if rear} > \text{front} \\ \text{rear} - \text{front} + \text{capacity}, & \text{if rear} < \text{front} \end{cases}$$

(c) Assume the kth element in the list is to be deleted, the elements after it should be moved up one position. Give a formula describing the positions of those elements to be moved up one position in terms of k, front, rear, capacity. Design an algorithm (pseudo code) for this Delete (int k) member function.

<answer>

$$\begin{cases} \text{no element needs to be moved}, & \text{if } (front + k) \text{ modulo } capacity = rear \\ from\big((front + k) \text{ modulo } capacity + 1\big) \text{ to } rear\,, & \text{if } (front + k) \text{ modulo } capacity < rear \\ from\ \big((front + k) \text{ modulo } capacity + 1\big) \text{ to } (capacity - 1) \text{ and } from\ 0 \text{ to } rear, & \\ & \text{if } (front + k) \text{ modulo } capacity > rear \end{cases}$$

Algorithm:

```
//exceptions
if (stack is empty)
    Throw exception of empty
if ((capacity - front + rear) modulo capacity < k)
    Throw exception of element unattainable

// different case
if ((front + k) modulo capacity = rear)
    rear = ((rear - 1 + capacity) modulo capacity);
    return;

else if ((front + k) modulo capacity < rear)
    for element in range of (front + k) modulo capacity to (rear - 1):
        queue[element] = queue[element+1]
    rear = ((rear - 1 + capacity) modulo capacity) ;
    return;

else if ((front + k) modulo capacity > rear)
    for element in range of (front + k) modulo capacity to (capacity - 2):
        queue[element] = queue [element + 1];

    queue[capacity - 1] = queue[0];

    for element in range of 0 to (rear-1):
        queue[element] = queue [element + 1];

    rear = ((rear - 1 + capacity) modulo capacity);
    return;
```

(d) Assume that we want to insert an element y immediately after the kth element. So, the elements from the (k+1) th element on should be moved down one position in order to give space for y, which might cause insufficient capacity case in which array doubling will be needed. Describe the situation when array doubling is needed (in terms of k, front, rear, capacity). Design an algorithm (pseudo code) for this Insert (int k, T& y) member function.

<answer>

There will be insufficient capacity when the front pointer is at the same index as rear pointer, but it is identical to the empty case if there is no size variable to track the queue size. Thus we either can add a size variable in the class such that we double the capacity when (front == rear and !size), or double the capacity when the insertion makes the queue full ((rear + 1) modulo capacity == front)

(Consider the latter case mentioned where we double the capacity when the stack becomes full after insertion)

Algorithm:

```
if (stack is empty)
    throw exception;
if ((capacity + front -rear) modulo capacity < k)
    throw exception;

// case for the queue is becoming full
if ((rear + 1) modulo capacity == front)
    initialize a new array with doubled size
    for i from 1 to capacity - 1:
        if (i <= k):
            new array[i] = old array [(front + i) % capacity];
        if (i > k):
            new array [i + 1] = old array [(front + i) % capacity]
    new array[k+1] = element y;
    delete old array
    assign new array to the original pointer
    front = 0;
    rear = capacity;
    return;

// case for queue not becoming full
else
    if ((front + k) modulo capacity < rear)
        for i from rear to front + k+1 (backward):
            queue [i+1] = queue[i];
```

```
        queue [front + k+1] = element y
        rear = (rear + 1) modulo capacity;
        return;
else if ((front + k) modulo capacity > rear)
    for i from 0 to rear:
        queue[i+1] = queue[i];
    for i from front + k + 1 to capacity - 1:
        queue[(i+1) modulo capacity] = queue [i];
    queue [front + k+1] = element y
        rear = (rear + 1) modulo capacity;
        return;
```

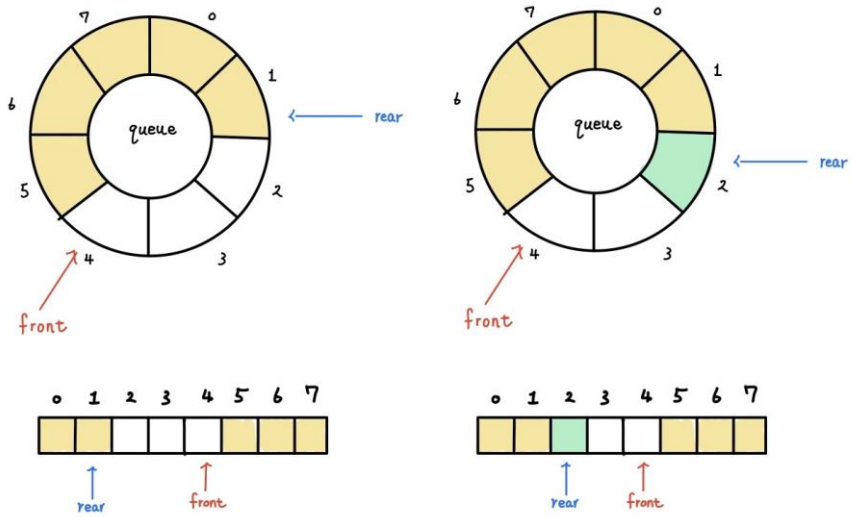
(e) The following code segment is used for Push member function in circular queue described in textbook. Please explain the code in detail. (using a graphical illustration and explanation)

```
template <class T>
void Queue<T>::Push(const T& x)
{// add x to queue
    if ((rear + 1) % capacity == front) //resize
    {   T* newQu = new T[2*capacity];
        int start = (front+1) % capacity;
        if(start<2)
           copy(queue +start, queue+start+capacity-1, newQu);
        else{
           copy(queue +start, queue+capacity, newQu);
           copy(queue, queue+rear+1,newQu+capacity-start);
        }
        front = 2*capacity - 1;
        rear = capacity -2;
        delete[] queue;
        queue = newQu; capacity *=2;
    }
    rear = (rear+1)%capacity;  queue[rear] = x;
}
```

<answer>

Case 1 (The queue is not full)

then we simply had the queue [rear + 1] equals the new element and assign the rear becomes rear +1

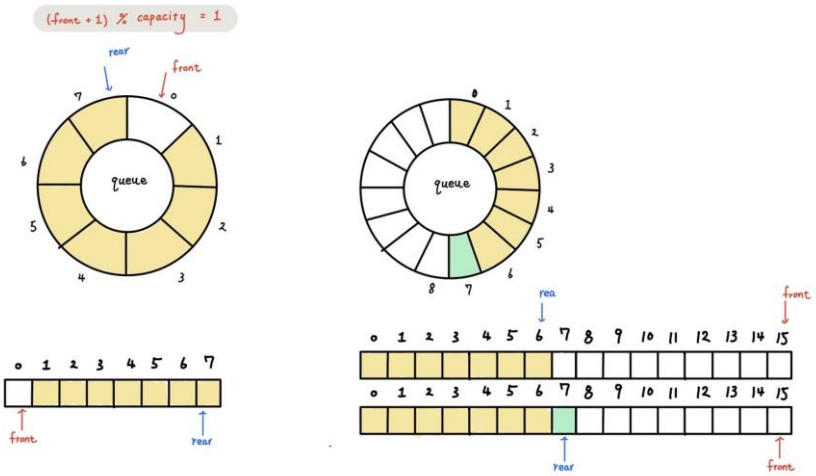


Case 2 (The queue is becoming full)

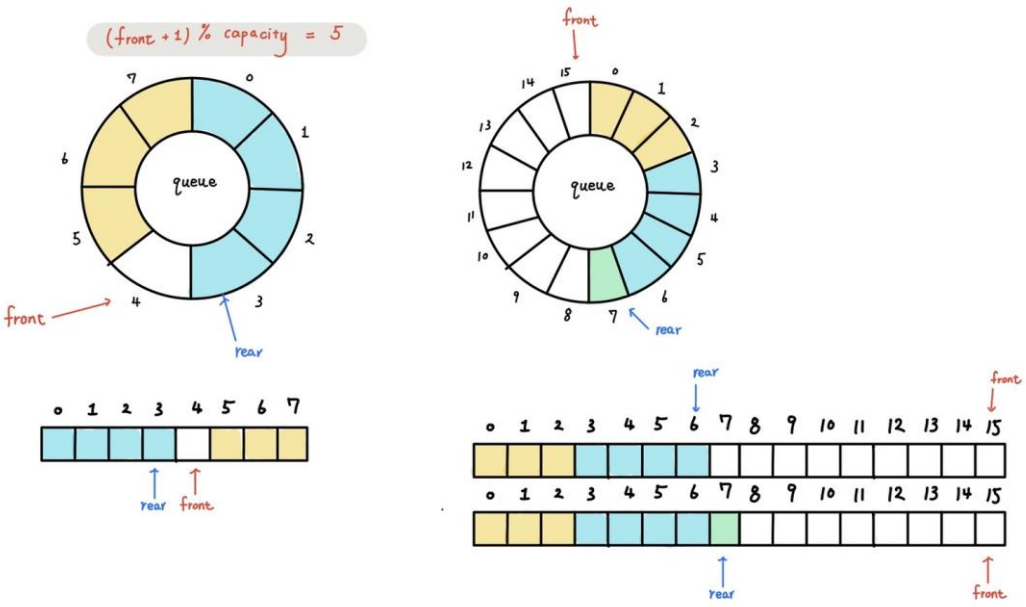
if ((rear + 1) % capacity == front) this means that the queue is becoming full, then we resize the queue capacity by below operation
T newQu = new T[2*capacity];*

if ((front + 1) % capacity) < 2)

Then we can directly copy the original element sequentially into the doubled array without worrying the index wrap around back to 0.



else ((front + 1) % capacity) > 2, then partially copy the original queue with two segments from (front + 1) to (capacity-1) and from 0 to rear element to the new array.



Then we assign the front equals to the original capacity $\times$ 2 $-$ 1 and assign the rear to original capacity $-$ 2, then we free the original space for the old array and assign the pointer to the new one also make the capacity value becomes double, then we make the similar push operation for case 1.

3. (10%) Design an algorithm, reverseQueue, that takes as a parameter a queue object and uses a stack object to reverse the elements of the queue. The operations on queue and stack should strictly follow the ADT 3.2 Queue ADT and ADT 3.1 Stack ADT.

   Prerequisite: I create my queue class with *size* parameter, and I make the reverseQueue function without return type and doesn't take parameter type, simply a class method. Where the operation

```
queue1.reverseQueue();
```

   simply reverses the queue.

   Algorithm:

```
First initialize a stack<T> with same size as the queue

while (the queue is not empty) {
    push the front element of the queue into the stack;
    pop the queue;
}

while (the stack is not empty) {
    push the top element of the stack into the queue
    pop the stack
 }
```

4. (5%) Given an integer k and a queue of integers, how do you reverse the order of the first k elements of the queue, leaving the other elements in the same relative order? For example, if k=4 and queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90]; the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90]. Design your algorithm for this task.

   Prerequisite: Consider we have size variable in the queue class.

   Algorithm

```
First initialize a temp queue<T> with the size of (size – k)
Then initialize a temp stack<T> with the size of k
Initialize OrigSize = size  // the OrigSize carries the value of the
queue size

while (size > OrigSize - k) {
    push the queue value into the temp stack;
    pop the original queue;
}

while (size > 0) {
    push the queue value into the temp queue;
    pop the original queue;
}

while (temp stack is not empty) {
    push the top element of the temp stack back into the original
queue;
    pop the temp stack;
}

while (temp queue is not empty) {
    push the front element of the temp queue back into the original
queue;
    pop the temp queue;
}
```

5. (10%) Suppose that you are a financier and purchase 100 shares of stock in Company *X* in each of January, April, and September and sell 100 shares in each of June and November. The prices per share in these months were

Jan	Apr	Jun	Sep	Nov
$10	$30	$20	$50	$30

Determine the total amount of your capital gain or loss using (a) FIFO (first-in first-out) accounting and (b) LIFO (last-in, first-out) accounting [that is, assuming that you keep your stock certificates in (a) a queue or (b) a stack.]
The 100 shares you still own at the end of the year do not enter the calculation.

(a)
Algorithms

```
initialize annual earn = 0;
while (Same year) {
    if (Purchasing) {
        push the negative value of price times the market price;
   }
    else if (Selling) {
        annual earn += queue.front();
        pop the queue.
        annual earn += price times the market price;
}
return annual earn;
```

Trace code:
-100 * 10 + -100 * 30 +100 * 20 + 100 * 30 = $ 1000

(b)

Algorithms

```
initialize annual earn = 0;
while (Same year) {
    if (Purchasing) {
        push the negative value of price times the market price;
   }
    else {
        annual earn += stack.top();
        pop the stack;
        annual earn += price times the market price;
}
return annual earn;
```

Trace code:

30 * 100 + 50 * -100 + 20 * 100 + 30 * -100 = $ -3000

6. (15%) For the maze problem,

(a) What is the maximum path length from start to finish for any maze of dimensions *m* x *p*?

<answer>

For maze that every grid is accessible, which means all grid has value of 1.

Consider either m or p is odd then we can zig-zag through the path to the end, thus creating a maximum path length of $m \times p$.

(b) Design a recursive version of algorithm for Path ().

<answer>

Algorithm:

```
Bool Path (x, y, map length, map width) {
 record the current position(y,x);
 mark[y][x] = 1;
 initialize current direction to east;
 initialize count = 8;

     while (count --) {
         assign variable newWay to store a new direction tuple (n_y, n_x);
         (n_y, n_x) = (y + direction, x+ direction);
         direction++; // changes to other direction in enumeration
         if((n_y,n_x) == (map width, map length))
             return true;
         if (!maze[n_y][n_x] && !mark[n_y][n_x]) {
             if (Path (n_x, n_y, map length, map_width))
                 return true;
         }
    }
    return false;
}
```

(c) What is the time complexity of your recursive version?

<answer>

Clearly the function is bounded by $8 \times (m \times p - 1)$, thus it is O(mp), considered we traverses all the point the function has visited, thus creating a connected graph. The time complexity is thus the cardinality of the edge plus the cardinality of the vertex = O (|V| + |E|), where E stands for possible connection with every visited point and V stands for every visited point.

7. (10%) Using the operator priorities of Figure 3.15 (shown below) together with those for '(' and '#' to answer the following:

priority	operator
1	Unary minus, !
2	*, /, %
3	+, -
4	<, <=, >, >=
5	==, !=
6	&&
7	||

(a) In function Postfix (Program 3.19, pptx **Infix to Postfix Algorithm**), what is the maximum number of elements that can be on the stack at any time if the input expression has n operators and delimiters?

<answer>

If n is odd, suppose we have the infix presentation (A/(B/(C/(…/(M/N)…)))), which has $\frac{n+1}{2}$ operator "/" , $\frac{n-1}{2}$ left parentheses "(" and 1 "#" in the stack before occurring the first right parentheses ")" pops all element from the stack, the maximum number of elements on the stack is $\frac{n+1+n-1}{2} + 1 = n + 1$

Now consider n is even, suppose the infix presentation (A/(B/(C/(…/(M/N)…)))), which has $\frac{n}{2}$ operator "/", $\frac{n}{2}$ left parentheses "(" and 1 "#" in the stack before occurring the first right parentheses ")" pops all element from the stack, the maximum number of the elements on the stack is $\frac{n}{2} + \frac{n}{2} + 1 = n + 1$

From above results, the maximum number of elements that is on the stack is n + 1

(b) What is the answer to (a) if the input expression e has n operators and the depth of nesting of parentheses is at most 6?

<answer>

Consider the term X_i, where it equals $-A_i*B_i+C_i>D_i==E_i\&\&F_i \|\|$ for i = 1, 2, 3, …
We then can have a postfix representation of $X_1(X_2(X_3(X_4(X_5(X_6(X_7Y_1))))))) + Y_2 + Y_3 + \ldots$
where Y_i is single operand.

Then we calculate the total operands before occurring right parentheses ")". for $X_1(X_2(X_3(X_4(X_5(X_6(X_7$ there are 7 * 7 operands contributed by X_i and 6 * 1 left parentheses "("and 1 "#" thus total of $7 \times 7 + 6 + 1 = 56$ elements on the stack, we have 55 operands before the left parentheses.

Therefore, we can obtain the formula, the maximum number of the element on the stack is 56

8. (20%) Write the postfix form and prefix form of the following infix expressions:

(a) –A + B – C + D*A/B

<answer>

Postfix form: 0A- B+C-DA*B/+

Prefix form: +-+-0ABC/*DAB

(b) A * -B + C/D – A*C

<answer>

Postfix form: A0B-*CD/+AC*-+

Prefix form: -+*A-0B/CD*AC

(c) (A + B) /D + E / (F + A * D) + C

<answer>

Postfix form: AB+D/EFAD*+/+C+

Prefix form: ++/+ABD/E+F*ADC

(d) A && B || C ||! (E > F)

<answer>

Postfix form: AB&&C || EF>! ||

Prefix form: || || && ABC! > EF

(e) !(A && !((B < C) || (C > E))) || (C < D)

<answer>

Postfix form: ABC<CE>||! &&! CD<||

Prefix form: || !&&A!|| < BC>CE<CD

9. (10%) Evaluate the following postfix expressions:

(a) 8 2 + 3 * 16 4 / - =

<answer>

(8+2) * 3 – (16/4) = 26

(b) 12 25 5 1 / / * 8 7 + - =

<answer>

12*(25/(5/1)) – (8+7) = 45