Stacks and Queues

1 Syntax Checking

How can we check whether the syntax is correct?

```
public void add( int idx, AnyType x) { if( theItems.length == size( ) ) ensureCapacity( size ( ) * 2
+ 1); for( int i=theSize; i > idx; i- ) theItems[ i ] = theItems[ i - 1 ]; theItems[ idx ] = x; theSize++;
}
```

Check that all opened brackets are closed as part of checking that the syntax is correct

A stack is good for the problem of bracket checking as the bracket closed should match with the last opened bracket.

```
input B, a sequence of brackets
output accept or reject

stack S
for b in B
    if b is an open bracket then
        S.push(b)
    end if

    if b is a close bracket then
        b'=S.pop()
        if b≠b' don't match then
            reject
if S.isEmpty is false then
    reject
accept
```

Requirements

- Every open bracket must be closed
- Each close bracket matches the last open bracket that has not been closed

2 Stacks

- A stack is a collection of objects that are inserted and removed according to the **last-in-first-out** (LIFO) principle
- Objects can be inserted into a stack at any time, but only the most recently inserted object (the **last**) can be removed at any time

2.1 Methods

A stack supports the following methods

- **push(e)**: Insert element e at the top of the stack
- **pop**: Remove and return the top element of the stack; an error occurs if the stack is empty

And possibly also:

- **size**: Return the number of elements in the stack
- **isEmpty**: Return a Boolean indicating if the stack is empty
- **top**: Return the top element in the stack, without removing it; an error occurs if the stack is empty

2.2 Example

What are the effects of the following on an initially empty stack? What is the output of each and what are the contents of the stack?

- push(5) - 5
- push(3) - 3,5
- pop - 5
- push(7) - 7,5
- pop - 5
- top -5
- pop -
- pop - ERROR
- isEmpty - True

2.3 Implementation using arrays

- In an array based implementation, the stack consists of an N-element array S, and an integer variable t that gives the top element of the stack
- We initialise t to -1, and we use this value for t to identify an empty stack. The size of the stack is t+1

2.4 Methods

```
# How do you find the find the size of the stack
size
    return t+1


# How do you determine if the stack is empty
isEmpty
    if t<0
        return True
    else
        return False

# Return the top element of the stack
top
    if isEmpty then
        throw a EmptyStackException
    end if
    return S[i]

# Add an element to the stack
push(e)
    # If size = N then the stack is full
    if size = N then
        throw a FullStackException
    end if
    t=t+1
    S[t]=e
```

```
# Remove an element from the stack
pop()
if isEmpty then
    throw a EmptyStackException
end if
e = S[t]
S[t] = NULL
t = t - 1
return e
```

2.5 Implementation using arrays

- The array based stack implementation is time efficient. The time taken by all methods does not depend on the size of the stack
- However, the fixed size N of the array can be a serious limitation:
  - If the size of the stack is much less than the size of the array, we waste memory
  - If the size of the stack exceeds the size of the array, the implementation will generate an exception
- The array based implementation of the stack has fixed capacity

How could you implement the stack using a linked list?
The easiest way is for the top of the stack to be at the head of the list, as it is better to insert and remove data from here

3 Queues

- A **queue** is a collection of objects that are inserted and removed according to the **first-in-first-out** principle
- Element access and deletion are restricted to the first element in the sequence, which is called the **front** of the queue
- Element insertion is restricted to the end of the sequence, which is called the **rear** of the queue

3.1 Methods

A queue supports the following methods

- **enqueue(e)**: Insert element e at the rear of the queue
- **dequeue**: Remove and return from the queue the element at the front; an error occurs if the queue is empty

And possibly also:

- **size**: Return the number of elements in the queue
- **isEmpty**: Return a boolean indicating if the queue is empty
- **front**: Return the front element of the queue, without removing it; an error occurs if the queue is empty

3.2 Example

What are the effects of the following on an initially empty queue? What is the output of each and what are the contents of the queue?

- enqueue(5) - [5]
- enqueue(3) - [3,5]
- dequeue - [3]
- enqueue(7) - [7,3]
- dequeue - [7]

- front - Output 7
- dequeue - []
- dequeue - ERROR
- isEmpty - True

3.3 Implementation using arrays

How can we implement a queue using an array Q of size N?

- We could put the front of the queue at Q[0] and let the queue grow from there
- This is not efficient. It requires moving all the elements forward one array cell each time we perform a dequeue operation .
- Instead, we use two variables f and r which have the following meaning:
  - f is an index to the cell of Q storing the front of the queue, unless the queue is empty, in which case f=r
  - r is an index to the next available cell in Q, that is, the cell after the rear of Q, if Q is not empty
- Initially we assign f=r=0, indicating that the queue is empty
- After each enqueue operation we increment r. After each dequeue operation we increment f
- r is incremented after each enqueue operation and never decremented. After N enqueue operations we would get an array-out-of-bounds error. So something is done when r=0
- To avoid this problem, we let r and f **wrap around** the end of Q, by using modulo N arithmetic on them

3.4 Methods

3.4.1 Size

```
return (N-f+r) mod N
# This allows for loop around as when the number gets bigger than N
# it will return the number into the LHS the array goes
```

3.4.2 isEmpty

```
return (f=r)
```

3.4.3 Front

```
if isEmpty then
    throw a EmptyQueueException
end if
```

3.4.4 enqueue(e)

```
if size=N-1 then
    throw a FullQueueException
end if
Q[r]=e
r=r+1 mod N
```

Plus 1 so that don't use last cell as f=r and the queue would be empty, even though it is full

3.4.5 Dequeue

```
if isEmpty then
    throw a EmptyQueueException
end if
temp=Q[f]
Q[f]=NULL
f=f+1 mod N
return temp
```

3.5 Implementation using arrays

- If the size of the queue is N, then f=r and the isEmpty method returns true, even though the queue is not empty
- We avoid this problem by keeping the maximum number of elements that can be stored in the queue to N-1. See the FullQueueException in the enqueue algorithm
- The array based implementation of the queue is time efficient. All methods run in constant time
- Similarly to the array based implementation of the stack, the capacity of array based implementation of array based implementation of the queue is fixed
- What if we used linked list implementation instead?
  - Put in data with each element linked together, front of queue at head of list, rear of queue at tail of list
  - To take something from the front of the queue, update the head to the next element in the queue
  - To add new data to the queue, create a new node, and point the tail at that, and point the new node to the previously added node

4 More

```
input integer k
output binary representation of k
stack S
while k>0
    S.push (remainder of k/2)
    k=k/2
while S isEmpty=False
    S.pop()
```