Stack & Queue

Stack

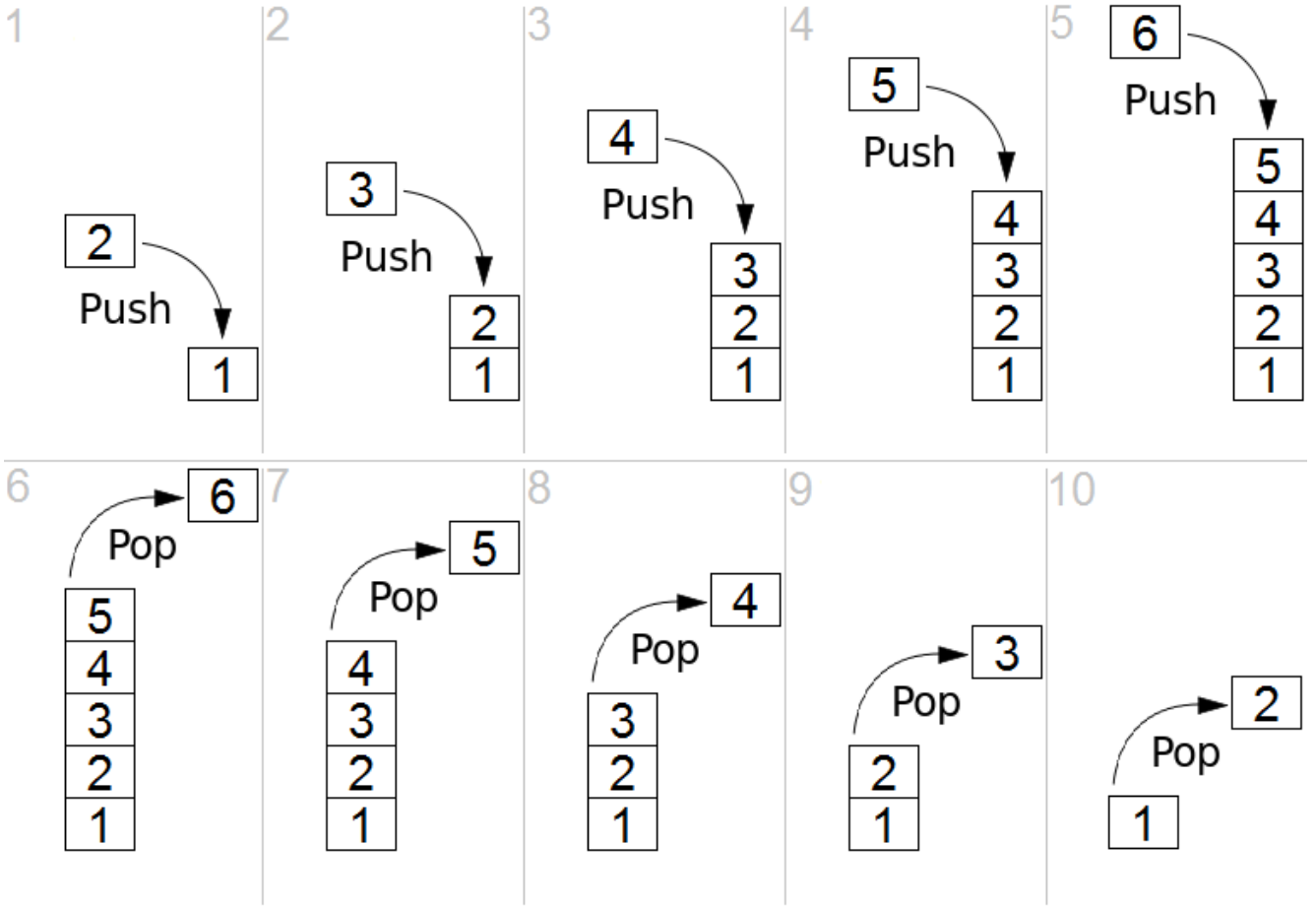
- A stack is a collection of elements where the added element will always go to the top of the collection.
- This also implies that when you take an element off the stack (remove from the collection), the last element to be added is the one taken off first.

- A stack works exactly like a pile of dishes.
  - When you push a value onto the stack, it goes on top of the stack.
  - When you pop a value from the stack, it removes the most recently pushed value (the top plate).

- Or another analogy is like wearing layers of clothes.
- Shirt goes first, then a sweater, then a jacket.
  - Jacket is put on last, but will come off first.

- Stack is a LIFO structure… Last In First Out

Pushing and Popping from a stack



Visualization of stack

- https://www.cs.usfca.edu/~galles/visualization/StackArray.html

Stack operations

- Typical operations in a stack are:
  - *Push*: adds an element to the stack (at the top).
  - *Pop*: removes the top element from the stack(if stack is not empty)
  - *Top* or *Peek*: returns the top element in the stack (if stack is not empty)
  - *IsEmpty*: returns true if stack is empty, else false.

- A stack can be implemented with an array or with a list.
  - If implemented with an array, then:
    - It will have a fixed size because arrays are usually fixed in size (unless using dynamic arrays).
    - Better locality of reference, because array is a contiguous chunk of memory.
  - If implemented with a linked list, then:
    - It does not have a fixed size, as linked lists grow a node at a time. (of course, there will be an upper limit based on the memory available, but usually that's quite high).

- Push will increase the size of a stack by 1, and Pop will decrease by 1

Big O (Time)

- Stack implemented by an array (dynamic array)
    - *Push*: Adds an element to the end of the array.
        - Time complexity?
    - *Pop*: Takes off the last element.
        - Time complexity?
    - *Top* or *Peek:* Returns the element on top (assume stack is not empty)
        - Time complexity?
    - *IsEmpty*: returns true if stack is empty, else false.
        - Time complexity?

Big O(Space)

- Array implementation:
  - O(N)

- Linked list implementation:
  - O(N)
  - This implementation does require additional space for the pointer to next node:
    - N pointers in singly linked.
    - 2*N pointers in doubly linked list.
    - However, Big O will still be O(N).

Example

- What is the output here:

```
Stack s1 = new Stack<Integer>();
s1.push( 10 );
s1.push( 20 );
int i1 = s1.peek();
int i2 = s1.peek();
s1.pop();
int i3 = s1.peek();

print i1, i2, i3;
print s1.size();     ← Assume size method exists.
s1.push( 50 );
print s1.peek();
```

LAB 1 of 2 – Balanced parenthesis

- Write a function that takes as input a string containing some characters that include open and close parenthesis.
  - return true if the parenthesis are well formed, false otherwise.
  - Parenthesis can be any of '(' or ')' or '{' or '}' or '[' or ']'

Some examples of well-formed parenthesis:

- ( )[ ]
- ([{ ( ) }])
- xyz ([ ])
- xyz (fff [fff])

Some examples of not well-formed parenthesis:

- ( ]
- ([ )]
- ( ) [ ] (
- { ()

Queue

- A queue is a collection of elements where the added element will always go to the end of the collection.
- When you take an element off the queue, the first element to be added is the one taken off first.
- So, it works just like you would expect a queue of people to work in a line. The first person in the queue is served first.

- Adding to a queue is also known as *enqueue*.
- Removing from a queue is also known as *dequeue*.

- Queue is a FIFO structure… First In First Out

Queue operations

- Typical operations in a queue are:
  - *Enqueue* or *Push*: adds an element to the queue (at the end).
  - *Dequeue* or *Pop*: removes the first element from the queue(if queue is not empty)
  - *Top* or *Peek*: returns the first element in the queue (if queue is not empty)
  - *IsEmpty*: returns true if queue is empty, else false.

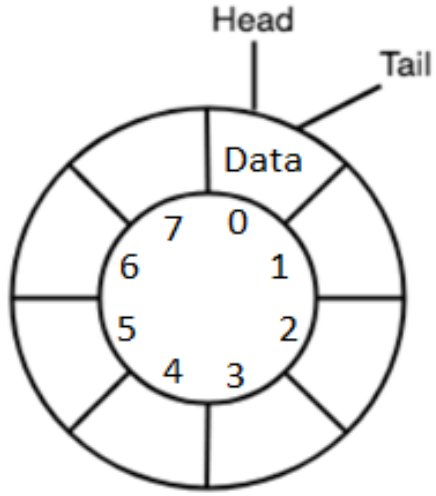
- A queue can be implemented with an array or with a list.
  - If implemented with an array, then:
    - It would typically be like a circular array.
      - Since in a queue, you remove from the front, you don't want to shift all remaining elements to the left one spot… that would be expensive.
      - To avoid this shifting, use a circular array buffer.
    - Better locality of reference, because array is a contiguous chunk of memory.
  - If implemented with a linked list, then:
    - It does not have a fixed size, as linked lists grow a node at a time. (of course, there will be an upper limit based on the memory available, but usually that's quite high).
- Push/Enqueue will increase the size of a queue by 1, and Pop/Dequeue will decrease by 1

Circular Queue

- When u create a circular queue, *head* and *tail* are initialized to -1

- When u **Enqueue**:
  - *tail* = (*tail* + 1) % SIZE
    - This ensures *tail* circles around the end of the array (How ?)
    - If queue is full
      - return error
    - array[*tail*] = dataToBeAdded;
  - If *head* is -1, we set *head* to 0

- Queue full check: *tail == head*



Circular queue of size 8.
Head and tail after the very first Enqueue

- When u **Dequeue**:
    - if queue is not empty, u return the element at *head*.
    - If *head* == *tail*, then queue is now empty (we just returned the last element).
        - Set *head* and *tail* to -1
    - else
        - *head* = (*head* + 1) % SIZE

Big O (Time)

- Queue implemented by a circular array:
    - *Push*: Adds an element to the end of the array.
        - Time complexity?
    - *Pop*: Takes off the first element
        - Time complexity?
    - *Top* or *Peek:* Returns the first element (assume queue is not empty)
        - Time complexity?
    - *IsEmpty*: returns true if queue is empty, else false.
        - Time complexity?

Big O(Space)

- Array implementation:
  - O(N)

- Linked list implementation:
  - O(N)
  - This implementation does require additional space for the pointer to next node:
    - N pointers in singly linked.
    - 2*N pointers in doubly linked list.
    - However, Big O will still be O(N).

Example

- What is the output here:

```
Queue q1 = new Queue <Integer>();
q1.enqueue( 10 );
q1.enqueue( 20 );
int i1 = q1.peek();
int i2 = q1.peek();
q1.dequeue();
int i3 = q1.peek();

print i1, i2, i3;
print q1.size();     ← Assume size method exists.
q1.enqueue( 50 );
print q1.peek();
```

LAB 2 of 2

Write a method **RemoveMax**, given the following **specifications**:

- Takes a stack of integer as a parameter
- Removes the maximum value from the stack, and returns that value.
- The order of other values in the stack should NOT be changed.

Example:

- A stack s1 contains: bottomOfStack { 7, 77, 88, 2, 97, 5, 117, 107, 61, 107, 52 } topOfStack
- Call **RemoveMax** on this stack, and it should remove 117 from the stack and return it
  - int maxValue = RemoveMax(s1); // maxValue will contain 117
  - Resultant stack:
    - bottomOfStack { 7, 77, 88, 2, 97, 5, 107, 61, 107, 52 } topOfStack
  - All instances of the maximum value should be removed.
    - Calling RemoveMax again should give:
    - bottomOfStack { 7, 77, 88, 2, 97, 5, 61, 52 } topOfStack // all instances of 107 removed

- You can use one (only 1) queue as auxiliary storage.
- You cannot use any other data structure in your code.
- Of course, using built in primitive type variables is ok (like, integer, etc.)