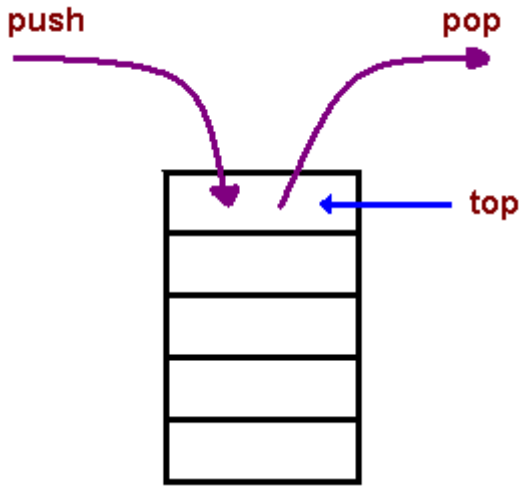
STACKS AND QUEUES

Stacks and Queues are both commonly used data structures that allow us to dynamically store and retrieve data items in two very different ways. We can use these in a variety of situations, however the choice depends on the problem that we are trying to solve.

The difference between stacks and queues is in removing. In a stack we remove the most recently added item; in queue, we remove the least recently added item.

STACKS	QUEUES
Stacks are based on the **LIFO** principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the **FIFO** principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

STACKS



A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a **recursive** data structure. Here is a structural definition of a Stack:

a stack is either empty or it consists of a top and the rest which is a stack;

BENEFITS

Stacks allow for constant-time adding and removing of an item. This is due to the fact that we don't need to shift items around to add and remove them from the stack.

CONSTRAINTS

Stacks, unfortunately, don't offer constant-time access to the nth item in the stack, unlike an array. This means it can possible take O(n) where n is the number of elements in the stack, time to retrieve an item.

METHODS

Stacks leverage the following methods:

- **pop():** Remove the top item from the stack
- **push(item):** Add an item to the top of the stack
- **peek():** Return the item at the top of the stack
- **isEmpty():** Returns true if the stack is empty

APPLICATIONS

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- **Backtracking**. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?
- Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.
- Language processing:

    - ✓ space for parameters and local variables is created internally using a stack.
    - ✓ compiler's syntax check for matching braces is implemented by using stack.
    - ✓ support for recursion

IMPLEMENTATION

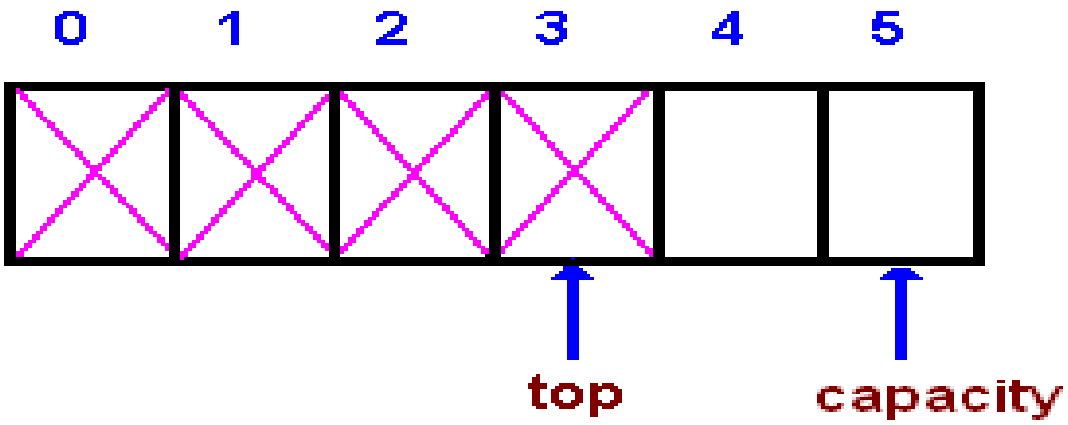
Array-based implementation

In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable top that refers to the top element in the stack and the capacity that refers to the array size. The variable top changes from -1

to capacity - 1. We say that a stack is empty when top = -1, and the stack is full when top = capacity-1.

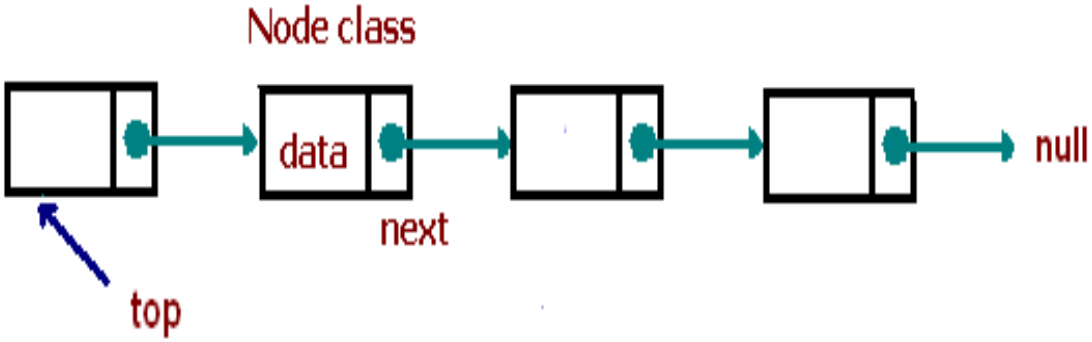
In a fixed-size stack abstraction, the capacity stays unchanged, therefore when top reaches capacity, the stack object throws an exception.

In a dynamic stack abstraction when top reaches capacity, we double up the stack size.



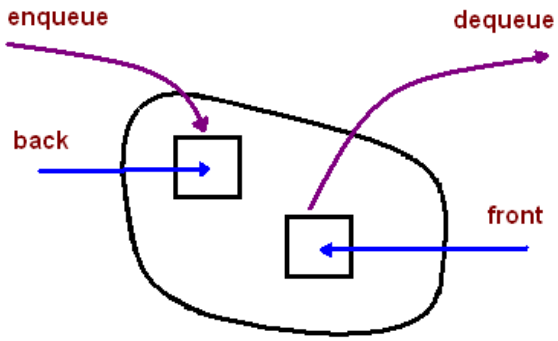
Linked List-based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



See the code on my GitHub ->

QUEUES



A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

USE CASES

Queues are very similar to linked lists and are typically used in breadth-first searches or when implementing a cache.

CONSTRAINTS

Queues are much harder to update when adding and removing nodes.

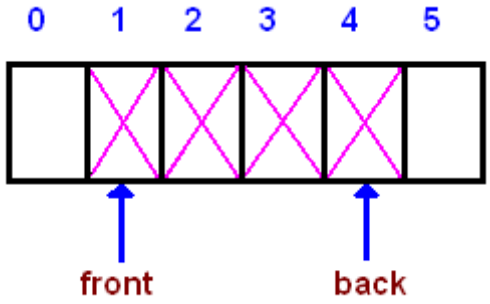
METHODS

Queues leverage the following methods:

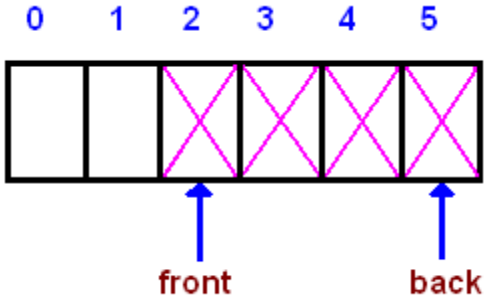
- **enqueue(item):** Remove the top item from the queue
- **dequeue():** Add an item to the top of the queue
- **peek():** Return the item at the top of the queue
- **isEmpty():** Returns true if the queue is empty

CIRCULAR QUEUE

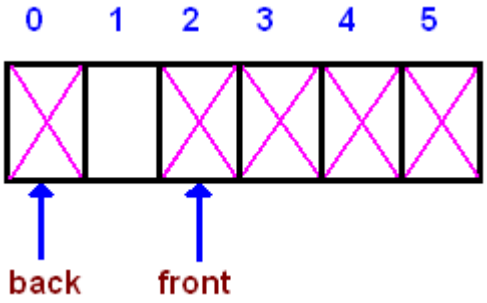
Given an array A of a default size (≥ 1) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves *back* reaches the end, leaving no space for adding new elements



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**



Finally, when *back* reaches *front*, the queue is full. There are two choices to handle a full queue:

a) throw an exception;

b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: *cur* - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

APPLICATIONS

There are many situations where a queue can be useful, a few implementations of it's use could be found in:

- **First Come First Serve** - If your application makes use of a queueing system for requests, then quite obviously the queue structure becomes applicable for use.
- **Breadth First Search** - During search, a queue can be used to store nodes that are to be processed. During processing, the adjacent node is added to a queue to allow processing in the same order that they are viewed.
- **Cache** - The LRU cache implementation is essentially a queue structure, whilst a queue is not efficient enough on it's own, one can be used to keep the order of cache items.

IMPLEMENTATION

See the code on my GitHub ->

USAGE IN REAL-WORLD

The simplest two search techniques are known as Depth-First Search(DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.

Depth-First Search with a Stack

In depth-first search we go down a path until we get to a dead end; then we *backtrack* or back up (by popping a stack) to get an alternative path.

- Create a stack
- Create a new choice point
- Push the choice point onto the stack
- while (not found and stack is not empty)
    - ✓ Pop the stack
    - ✓ Find all possible choices after the last one tried
    - ✓ Push these choices onto the stack
- Return

Breadth-First Search with a Queue

In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

- Create a queue
- Create a new choice point
- Enqueue the choice point onto the queue
- while (not found and queue is not empty)
    - Dequeue the queue
    - Find all possible choices after the last one tried
    - Enqueue these choices onto the queue
- Return

Arithmetic Expression Evaluation

An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:

```
1 + ((2 + 3) * 4 + 5)*6
```

We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

Converting from Infix to Postfix. Typically, we deal with expressions in infix notation

```
2 + 5
```

where the operators (e.g. +, *) are written between the operands (e.q, 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression

```
2 5 +
```

Writing the operators before the operands gives a prefix expression

```
+2 5
```

Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25%):

```
70 + 150 * 1.0725
```

Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation

```
70  150 + 1.0725 *
```

Postfix has the nice property that parentheses are unnecessary.

Now, we describe how to convert from infix to postfix.

1. Read in the tokens one at a time
2. If a token is an integer, write it into the output
3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
4. If a token is a left parentheses '(', push it to the stack
5. If a token is a right parentheses ')', you pop entries until you meet '('.
6. When you finish reading the string, you pop up all tokens which are left there.
7. Arithmetic precedence is in increasing order: '+', '-', '*', '/';

Example. Suppose we have an infix expression:2+(4+3*2+1)/3. We read the string by characters.

```
'2' - send to the output.
'+' - push on the stack.
'(' - push on the stack.
'4' - send to the output.
'+' - push on the stack.
'3' - send to the output.
'*' - push on the stack.
'2' - send to the output.
```

Evaluating a Postfix Expression. We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

Consider the following postfix expression

```
5 9 3 + 4 2 * * 7 + *
```

Here is a chain of operations

```
Stack Operations          Output
-------------------------------------
push(5);                  5
push(9);                  5 9
push(3);                  5 9 3
push(pop() + pop())           5 12
push(4);                  5 12 4
push(2);                  5 12 4 2
push(pop() * pop())           5 12 8
push(pop() * pop())           5 96
push(7)                   5 96 7
push(pop() + pop())           5 103
push(pop() * pop())           515
```

Note, that division is not a commutative operation, so 2/3 is not the same as 3/2.