Data structure

Data structure

A **Data structure**:

- Defines the way data are organized and stored in computers.
- Example:
  - Dictionary: Words and definitions are arranged for convenient lookup

- **Program = Algorithm + Data structure**
  - Algorithm: precise set of instructions to solve a particular task
  - Data structure: data type or representation and the associated operations

Data structure

Interest of Data structure:

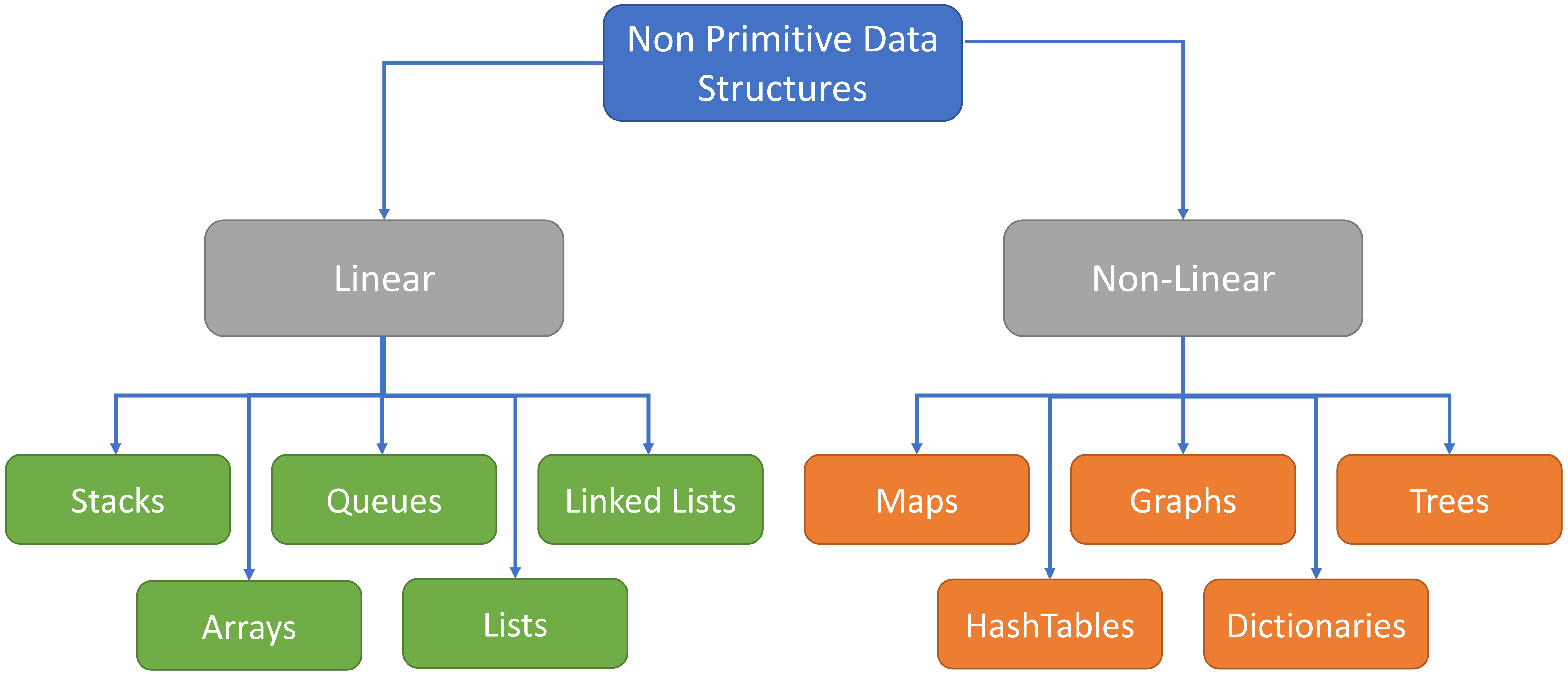
- Fundamental in Computer science and Software engineering
- Key topic in interviews
- Need to choose the most suitable data structure for a given problem

Major operations: Searching, Insertion, Update, Deletion

Data structures can also be classified as:

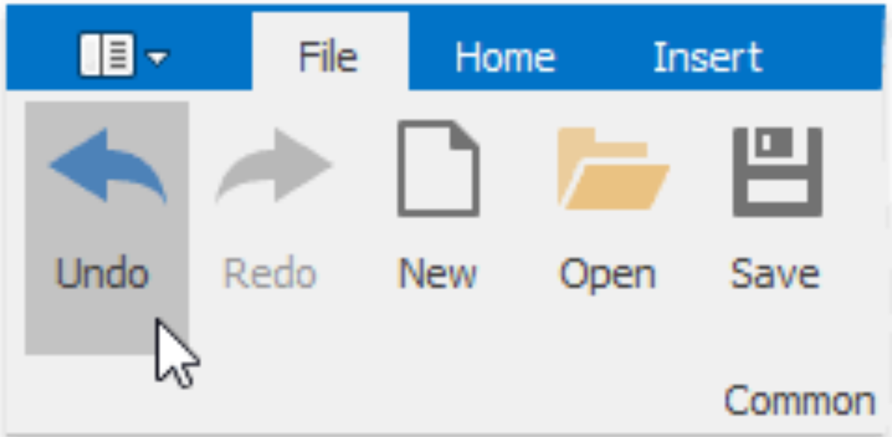
- **Static data structure**: The size is allocated at the compile time. The maximum size is fixed.
- **Dynamic data structure**: The size is allocated at the run time. The maximum size is flexible.

Data structure

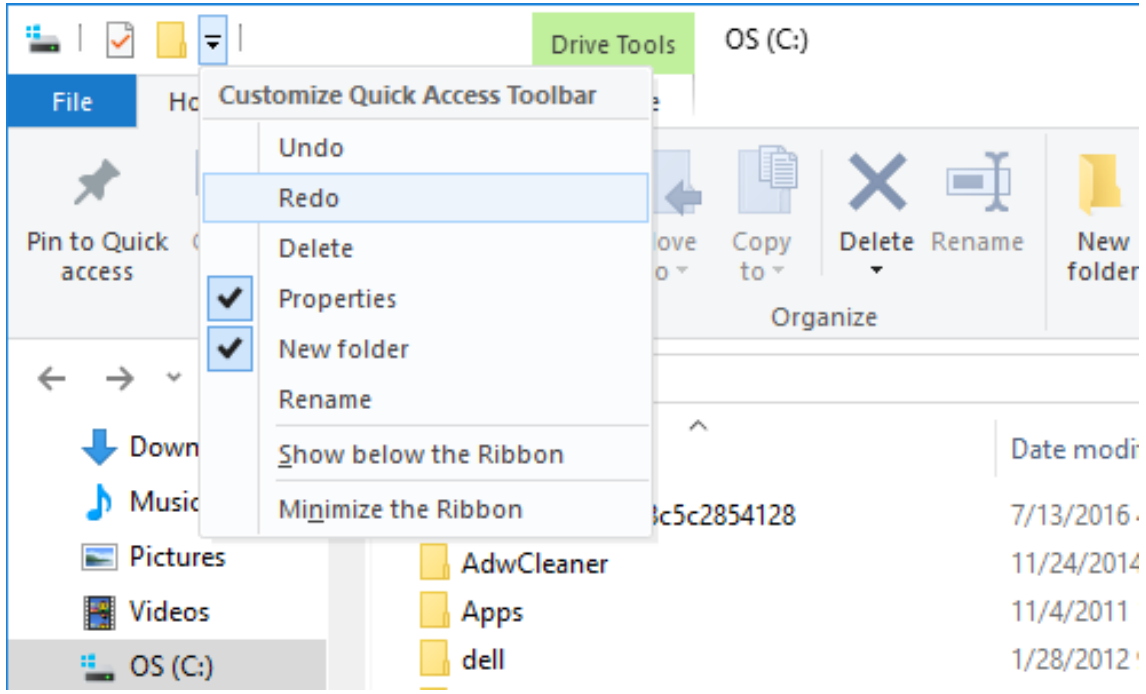


Stack

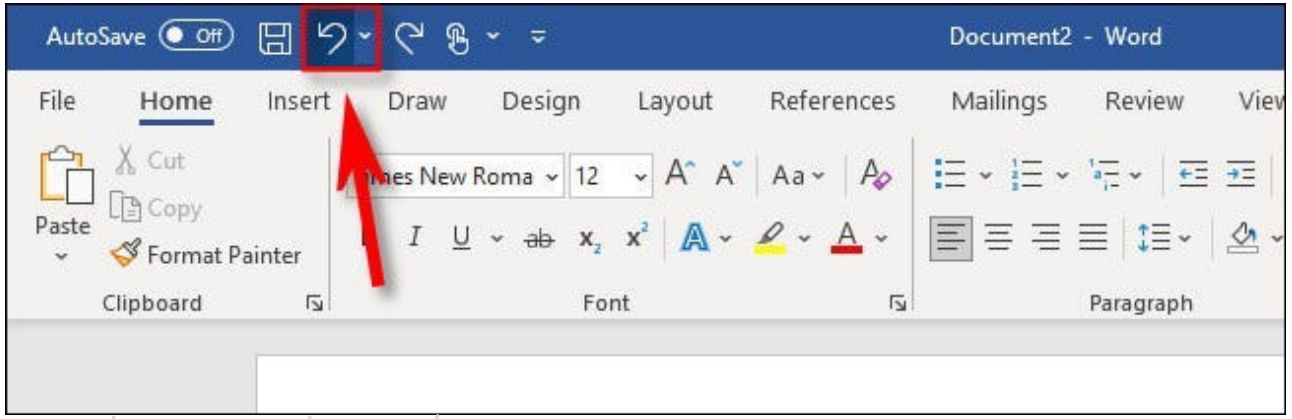
Undo operation



https://earthcape.com/docs/win-ui/images/img121308.png



https://support.moonpoint.com/os/windows/win10/Undo-Redo/Redo.png



https://www.howtogeek.com/wp-content/uploads/2021/11/word_undo_button.jpg?trim=1,1&bg-color=000&pad=1,1

Stack

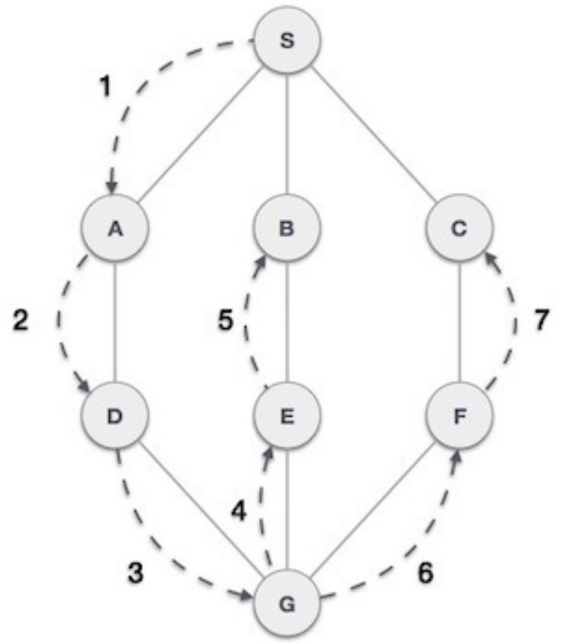
Follow the Last in First out (LIFO) ordering

- The last element added is the first element that comes out
- The last element added is the element on the top
- The first element added is at the bottom.

What are stacks used for?

Despite the simple implementation stacks can be used to solve very complex problems!

Depth First Search



Expression Evaluation Algorithm

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Stacks are used:

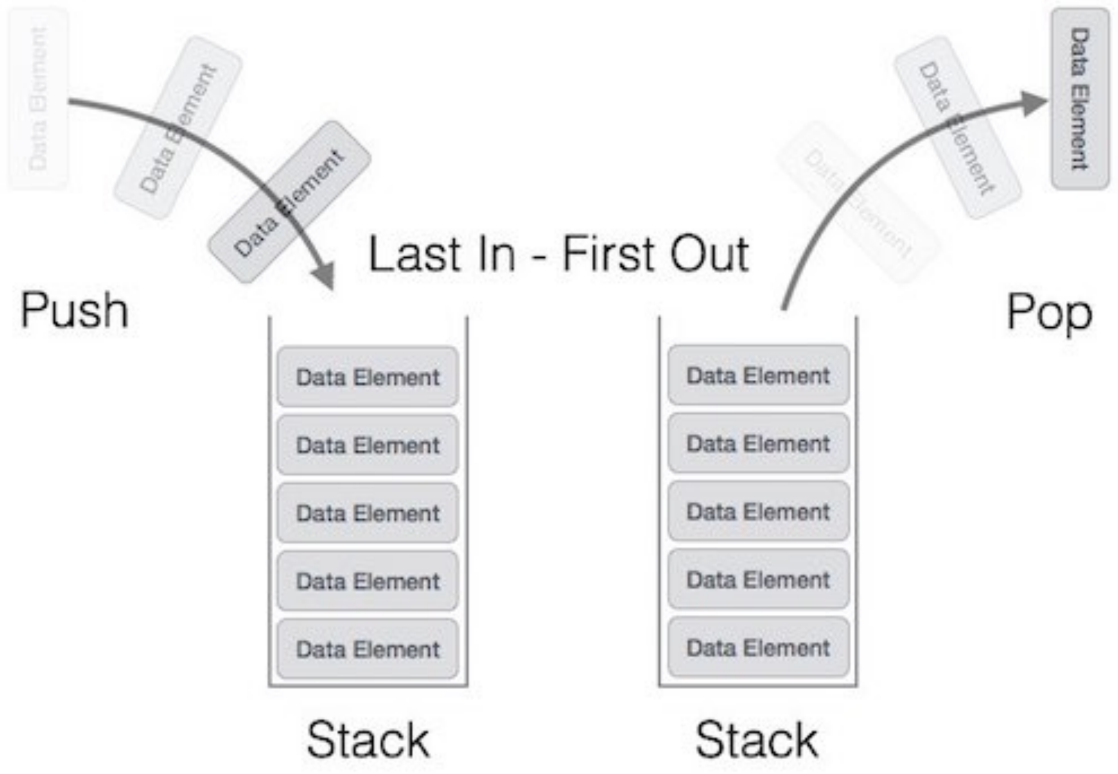
- To backtrack to the previous task/state.
- To store a partially completed task.

How do Stacks works?

A stack is an abstract data type that serves as a collection of elements, with two main principal operations:

- Push, which adds an element to the collection,

- Pop, which removes the most recently added element that was not yet removed.



Stack ADT

Basic Function	What does it do?
push(element)	Inserts an element at the top
pop()	Removes an element from the top and returns it

Additional Function	What does it do?
peek() / top()	Returns the top element of the stack, without removing it
is_empty()	Returns True if the stack is empty
size()	Returns the size of the stack

Stack implementation

- Stacks can can be implemented using different Python's data structures such as : **arrays, Lists** or **Linked Lists**.

- We will show an implementation of stacks using **lists**.

A typical Stack must contain the following functions:

push(element)
pop()
peek()
is_empty()
size()

Stack implementation using Python's list

Let us :

- Create a Stack.py file
- Construct a simple class MyStack
- Instatiate an object of that class

Stack.py

```
class MyStack:

        def __init__(self):
                self.stack_list = []

stack_obj = MyStack()
```

Stack implementation using Python’s list

Let us implement the following helpers:

- is_empty()
- peek()
- size()

```
def is_empty(self):
        return len(self.stack_list) == 0
```

```
def peek(self):
        if self.is_empty():
                return None
        return self.stack_list[-1]
```

```
def size(self):
        return len(self.stack_list)
```

Stack implementation using Python’s list

Let us implement the main methods:

- push(element)
- pop()

```
def push(self, element):
        self.stack_list.append(element)
```

```
def pop(self):
        if self.is_empty():
                return None
        return self.stack_list.pop()
```

Complexities of stack operations

Time complexity

Space complexity

O(n)

```
def is_empty(self):
        return len(self.stack_list) == 0
```

O(1)

```
def peek(self):
        if self.is_empty():
                return None
        return self.stack_list[-1]
```

O(1)

```
def size(self):
        return len(self.stack_list)
```

O(1)

```
def push(self, element):
        self.stack_list.append(element)
```

O(1)

```
def pop(self):
        if self.is_empty():
                return None
        return self.stack_list.pop()
```

O(1)

O(n) worst case, when an operation causes the list to resize its internal array.
(n: number of current elements in the list)

Improvements

1- Modify the MyStack class implementation in such a way that the stack_size is a data member of the class.

2- Modify the MyStack implementation so that the stack's capacity is limited to maxlen elements, where maxlen is an optional parameter to the constructor (that defaults to None). If push is called when the stack is at full capacity, print out an error message.

3- Re-write the MyStack class while pre-allocating an underlying list with length equal to the stack's maximum capacity. (Aim: Avoid amortization by reserving capacity)

Reversing Data using a Stack

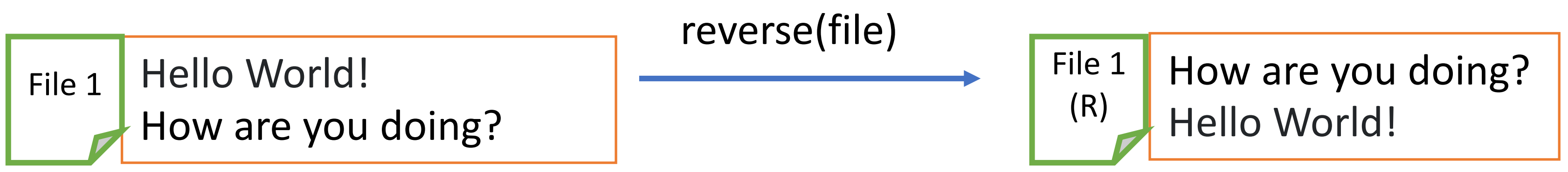
As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence.

$$\begin{tikzcd}\boxed{\text{"Hello World!"}} \arrow[r, "\text{reverse(text)}"] & \boxed{\text{"!dlroW olleH"}}\end{tikzcd}$$

1) Create an empty stack.
2) One by one push all characters of string to stack.
3) One by one pop all characters from stack and put them back to string.

Reversing Data using a Stack

As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence.



1) Create an empty stack.
2) Read each line and push it onto the stack.
3) Write the lines in the order they are poped.

Reversing Data using a Stack

```
def reverse_file(filename):
        S = MyStack()
        original = open(filename)

        for line in original:
                S.push(line.rstrip(‘\n’))
        original.close()

        output = open(filename, ‘w’)
        while not S.is_empty():
                output.write(S.pop() + ’\n’)
        output.close()
```

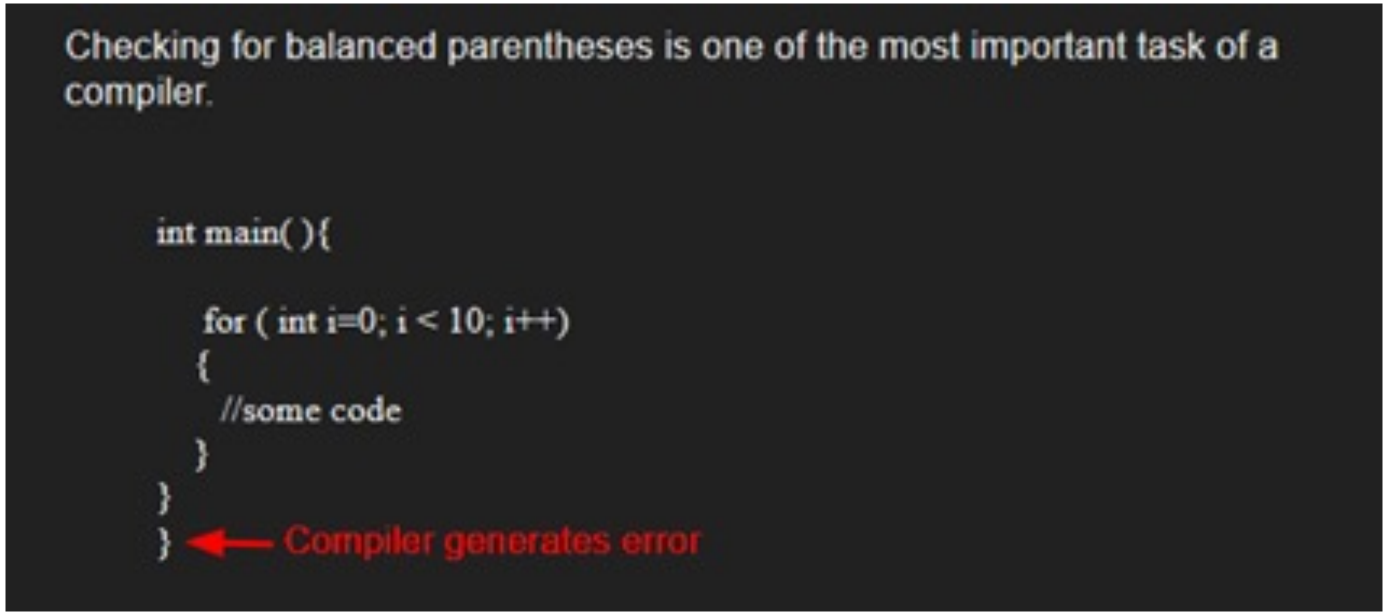
Matching Parentheses

We consider arithmetic expressions that may contain various pairs of grouping symbols, such as:

- Parentheses: “(” and “)”
- Braces: “{” and “}”
- Brackets: “[” and “]”

Each opening symbol must match its corresponding closing symbol.

- Correct: ( )(( )){([( )])}
- Correct: ((( )(( )){([( )])}))
- Incorrect:)(( )){([( )])}
- Incorrect: ({[ ])}
- Incorrect: (



Checking for balanced parentheses is one of the most important task of a compiler.

```
int main( ){

    for ( int i=0; i < 10; i++)
    {
      //some code
    }
}
} ← Compiler generates error
```

Matching Parentheses

1- Read one character at a time
2- If it is an opening delimiter, push it onto the stack
3- If it is a closing delimiter, the stack should not be empty, and pop an element
4- The poped character should match the closing delimiter
5- When no input character is left, check if the stack is empty

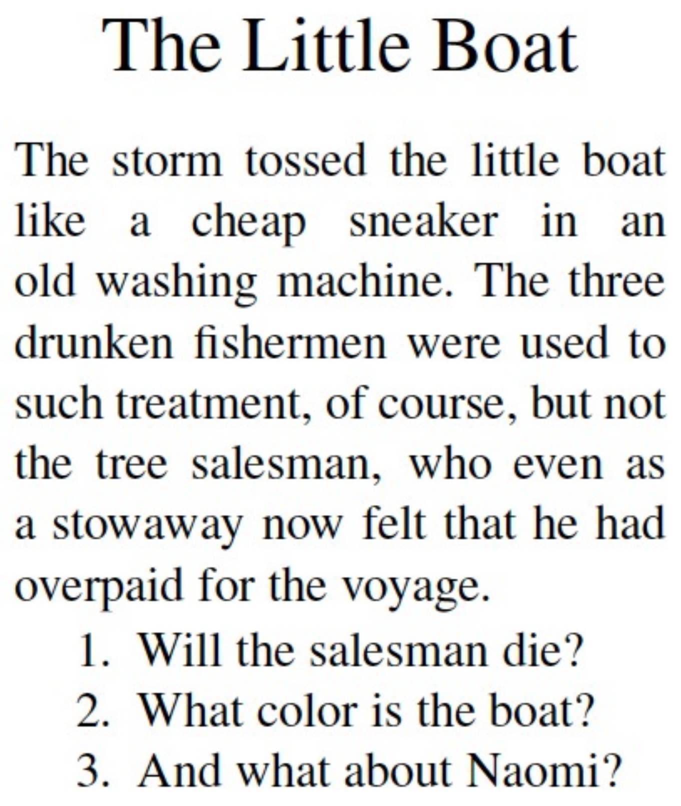
Matching Parentheses

```
def is_matched(expr):
      lefty = ’({[‘
      righty = ‘)}]’
      S = MyStack()
      for c in expr:
            if c in lefty:
                  S.push(c)
            elif c in righty:
                  if S.is_empty():
                        return False
                  if righty.index(c) != lefty.index(S.pop()):
                        return False
      return S.is_empty()
```

The matching algorithm on a sequence of length n runs in O(n) time

Matching Tags in a Markup Language

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Matching Tags in a Markup Language

```
def is_matched_html(raw):

    S = MyStack()
    j = raw.find(‘<‘)
    while j != -1:
        k = raw.find(‘>’, j+1)
        if k == -1:
            return False

        tag = raw[j+1:k]
        if not tag.startswith(‘/’):
            S.push(tag)
        else:
            if S.is_empty():
                return False
        if tag[1:] != S.pop():
            return False
        j = raw.find(‘<‘, k+1)
    return S.is_empty( )
```

Exercices:

4 - we assume that opening tags in HTML have form <name>, as with <li>. More generally, HTML allows optional attributes to be expressed as part of an opening tag.
The general form used is <name attribute1="value1" attribute2="value2">;
for example, a table can be given a border and additional padding by using an opening tag of <table border="3" cellpadding="5">.
Modify the is_matched_html(raw) so that it can properly match tags, even when an opening tag may include one or more such attributes.

Expression Evaluation

An arithmetic expression can be written in three different but equivalent notations
These notations are named as how they use operator in expression.

These notations are :

- Infix Notation:
    - Operators are used **in-between** operands. e.g.: a + b
    - Easy for humans to read, write and speak in infix notation
    - Not that easy for computing devices, (Algorithms processing infix notation could be costly)

- Prefix Notation:
    - Operator is **prefixed** to the operands. e.g.: + a b

- Postfix Notation:
    - Operator is **postfixed** to the operands. e.g.: a b +

Expression Evaluation

Infix Notation	Prefix Notation	Postfix Notation
a + b	+ a b	a b +
(a + b) * c		
a * (b + c)		
a / b + c / d		
(a + b) * (c + d)		
((a + b) * c) - d		

Expression Evaluation

Infix Notation	Prefix Notation	Postfix Notation
a + b	+ a b	a b +
(a + b) * c	* + a b c	a b + c *
a * (b + c)	* a + b c	a b c + *
a / b + c / d	+ / a b / c d	a b / c d / +
(a + b) * (c + d)	* + a b + c d	a b + c d + *
((a + b) * c) - d	- * + a b c d	a b + c * d -

Postfix Expression Evaluation

1 – Scan the expression from left to right

2 – If it is an operand:

- Push it to the stack

3 – If it is an operator:

- Pop the two operands from the stack
- Perform the operation
- Store the output back to the stack

4 – Scan the expression until all operands are consumed

5 – Pop the stack

Postfix Expression Evaluation

$$\mathbf{2\ 3\ 4\ *\ +9\ -}$$

1 – Scan the expression from left to right

2 – If it is an operand:

- Push it to the stack

3 – If it is an operator:

- Pop the two operands from the stack
- Perform the operation
- Store the output back to the stack

4 – Scan the expression until all operands are consumed

5 – Pop the stack

Character	Stack
2	2
3	2 3
4	2 3 4
*	2 12
+	14
9	14 9
-	5

Postfix Expression Evaluation

```
def evaluatePostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:

            # If the scanned character is an operand
            # (number here) push it to the stack
            if i.isdigit():
                self.push(i)

            # If the scanned character is an operator,
            # pop two elements from stack and apply it.
            else:
                val1 = self.pop()
                val2 = self.pop()
                self.push(str(eval(val2 + i + val1)))

        return int(self.pop())
```

Infix to Postix Conversion

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    1. If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a ‘(‘), push it.
    2. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Character	Stack	Output
A		A
*	*	A
(	* (	A
B	* (	A B
/	* (/	A B
C	* (/	A B C
+	* (+	A B C /
D	* (+	A B C / D
*	* (+ *	A B C / D
E	* (+ *	A B C / D E
)	*	A B C / D E * +
-	-	A B C / D E * + *
F	-	A B C / D E * + * F
		A B C / D E * + * F -

$$A * (B \; / \; C + D * E) - F$$

Rules:

- Priorities :
  - *, /
  - +, -
- No 2 operators of the same priority can stay next to each other in the stack.
- Lowest priority operators cannot be placed on top of a higher priority operator.

- If operand, output
- If operator, check rules

```
def infixToPostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack
            elif i  == '(':
                self.push(i)

            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and
                                self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()
```

Infix to Postix Conversion

```
            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

        print "".join(self.output)
```

Queue

Queue

A Linear Data Structure

Follow the First in First out (FIFO) principle

The first element added is the first element that comes out

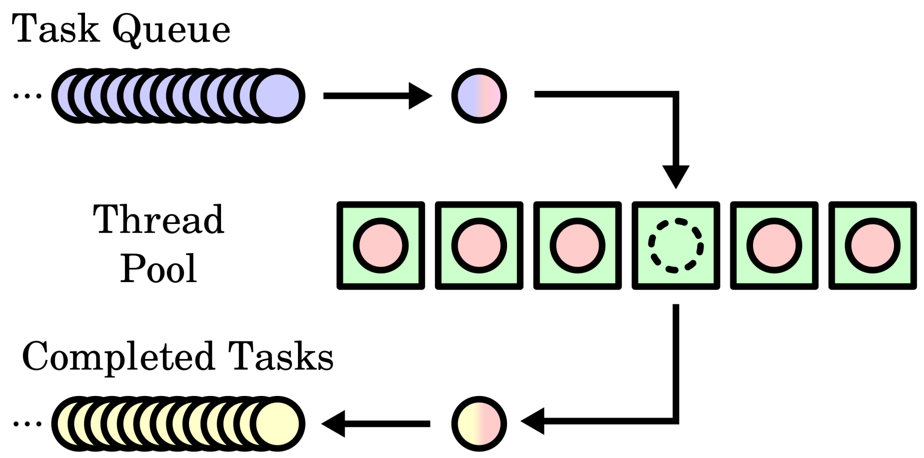
Queue

We usually say that elements enter a queue at the back and are removed from the front

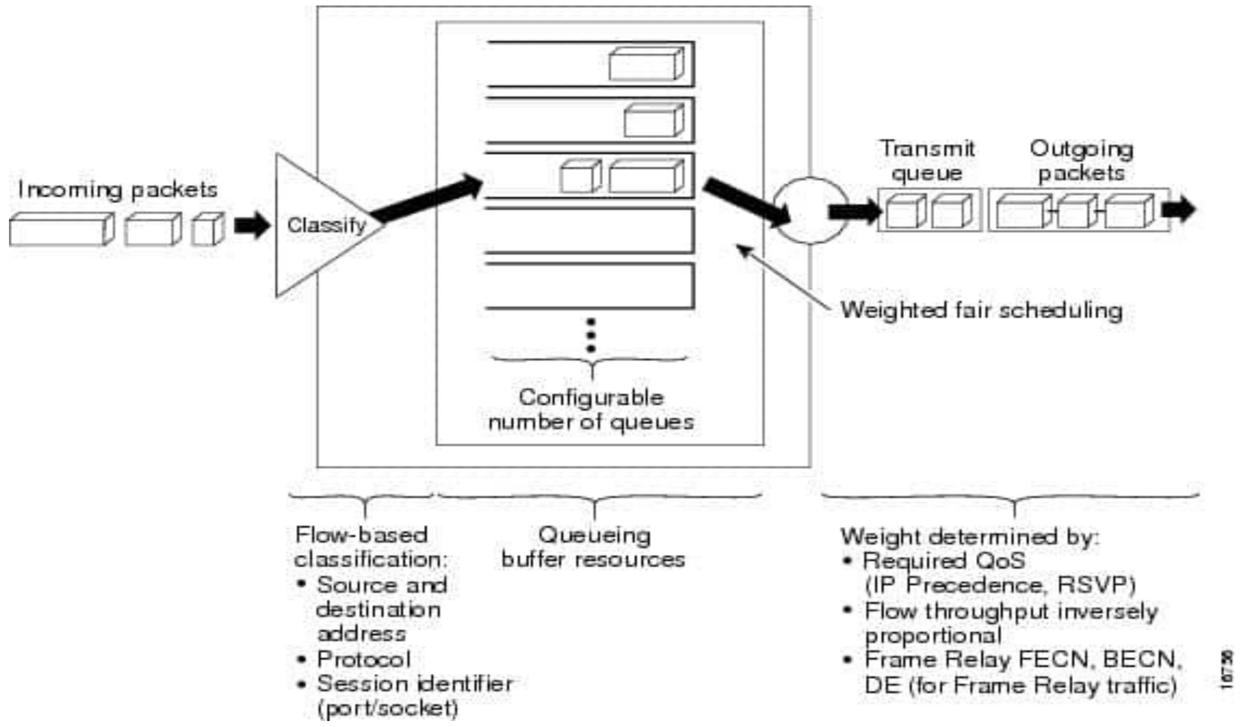
https://www.bbc.com/news/magazine-28850379

What are Queues used for?

FIFO queues are used by many computing devices, such as a networked printer, or a Web server responding to requests



https://en.wikipedia.org/wiki/Scheduling_(computing)


Incoming packets
Classify
Transmit queue
Outgoing packets
Weighted fair scheduling
Configurable number of queues
Flow-based classification:
• Source and destination address
• Protocol
• Session identifier (port/socket)
Queueing buffer resources
Weight determined by:
• Required QoS (IP Precedence, RSVP)
• Flow throughput inversely proportional
• Frame Relay FECN, BECN, DE (for Frame Relay traffic)

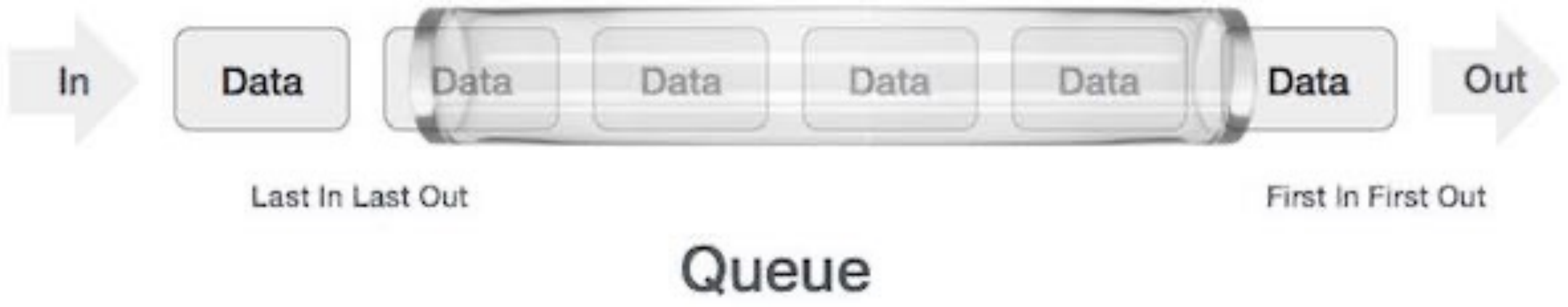
https://www.cisco.com/c/en/us/td/docs/ios/qos/configuration/guide/12_2sr/qos_12_2sr_book/congstion_mgmt_oview.html

How do Queues works?

A stack is an abstract data type that defines a collection that keeps objects in a sequence following the FIFO principle.
The main principal operations:

- Enqueue, which adds an element to the back of the queue,

- Dequeue, which removes and return the first added element.



Queues ADT

Basic Function	What does it do?
enqueue(element)	Inserts an element at the back of the queue
dequeue()	Removes and returns an element from the front

Additional Function	What does it do?
front()	Returns the element at the front of the queue
is_empty()	Checks is the queue is empty
size()	Returns the size of the queue

Queue implementation

- Queues can be implemented using different Python’s data structures such as : **arrays, Lists** or **Linked Lists**.

- We will show an implementation of queues using **lists**.

A typical Queue must contain the following functions:

enqueue(element)
dequeue()
front()
is_empty()
size()

Queue implementation using Python’s list

Let us :
- Create a Queue.py file
- Construct a simple class MyQueue
- Instatiate an object of that class

Queue.py

```
class MyQueue:

      def __init__(self):
            self.queue_list = []

queue_obj = MyQueue()
```

Queue implementation using Python’s list

Let us implement the following helpers:

- is_empty()
- front()
- size()

```
def is_empty(self):
        return len(self.queue_list) == 0
```

```
def front(self):
        if self.is_empty():
                return None
        return self.queue_list[0]
```

```
def size(self):
        return len(self.queue_list)
```

Queue implementation using Python’s list

Let us implement the main methods:

- enqueue(element)
- dequeue()

```
def enqueue(self, element):
        self.queue_list.append(element)
```

```
def dequeue(self):
        if self.is_empty():
                return None
        return self.queue_list.pop(0)
```

Complexities of queue operations

Time complexity

Space complexity

O(n)

```
def is_empty(self):
        return len(self.queue_list) == 0
```

O(1)

```
def front(self):
        if self.is_empty():
                return None
        return self.queue_list[0]
```

O(1)

```
def size(self):
        return len(self.queue_list)
```

O(1)

```
def enqueue(self, element):
        self.queue_list.append(element)
```

O(1)

```
def dequeue(self):
        if self.is_empty():
                return None
        return self.queue_list.pop(0)
```

O(n)

When pop is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the pop

Improvements

1- Modify the MyQueue class implementation by:

- Avoid the call to pop(0)
- Replace the dequeued element with a reference to None, and keep track of the index of the element at the front.

Aim: Dequeue() in O(1) time

2- Update the size() method and add a size variable to keep track of the current queue length.

From Linear Queue to Circular Queue

If we repeatedly enqueue a new element and then dequeue another

Possible Situation: Few elements stored in an arbitrarily large list

The size of the underlying list would grow to O(m)

(m is the total number of enqueue operations since the creation of the Queue)

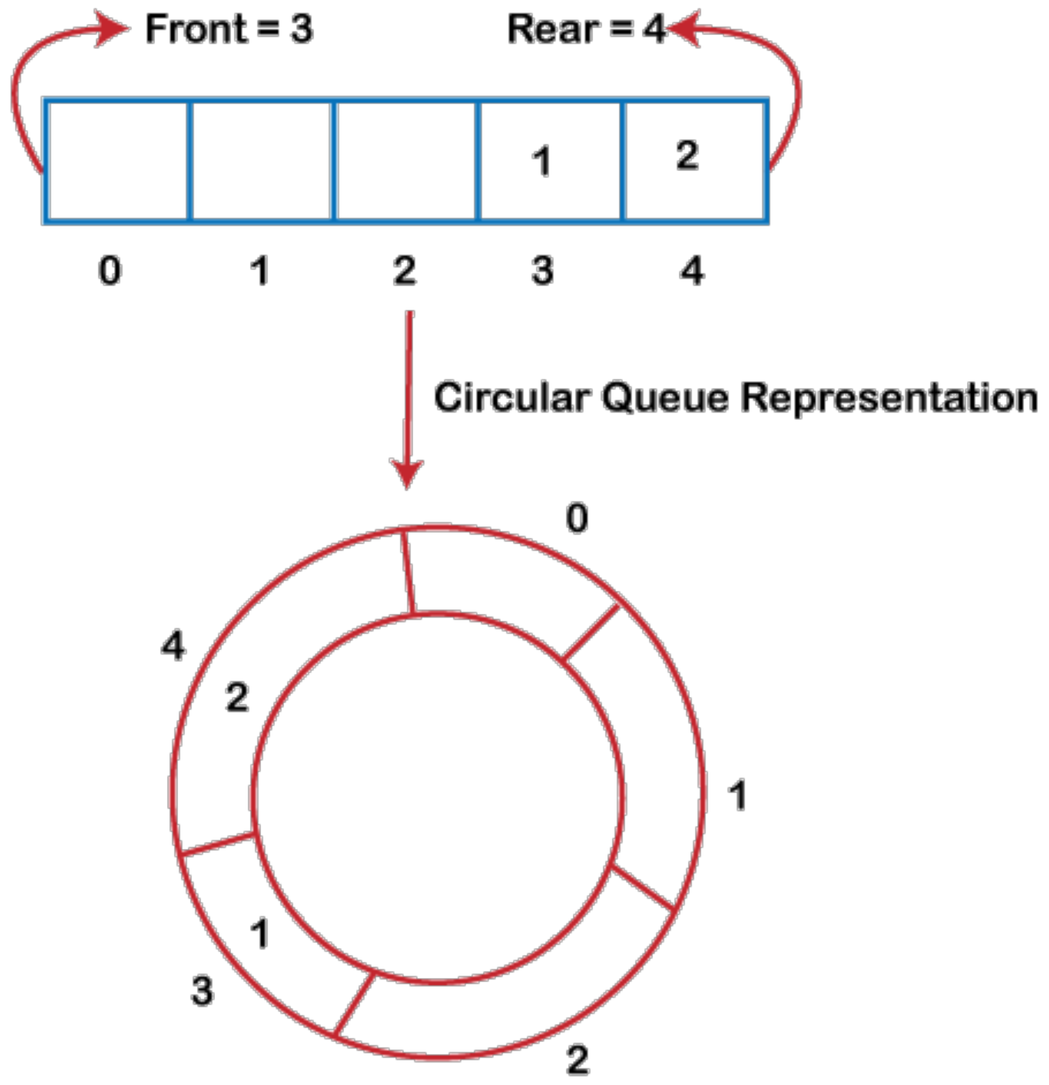
Solution: Allow the contents of the queue to "wrap around" the end of an underlying array

Circular Queue

Circular queues are almost similar to linear queues with only one exception.

Both ends are connected to form a circle.

Initially, the front and rear part of the queue point to the same location. Eventually they move apart as more elements are inserted into the queue.



Implementation

- Initially reserve a list of moderate size for storing data, although the queue formally has size zero.

- Modify the enqueue and the dequeue methods, while taking into account the index of the element in the front.

- Pay attention to how the back of the queue is computed.

```
class ArrayQueue:
      """FIFO queue implementation using a Python list as underlying storage."""
      DEFAULT_CAPACITY = 10 # moderate capacity for all new queues

      def __init__(self):
            """Create an empty queue."""
            self._data = [None] ∗ ArrayQueue.DEFAULT_CAPACITY
            self._size = 0
            self._front = 0
```

```
      def __len__(self):
            """Return the number of elements in the queue."""
            return self._size

      def is_empty(self):
            """Return True if the queue is empty."""
            return self._size == 0
```

```
      def first(self):
      """Return (but do not remove) the element at the front of the queue."""
            if self.is_empty():
                  raise Empty("Queue is empty")
            return self._data[self._front]
```

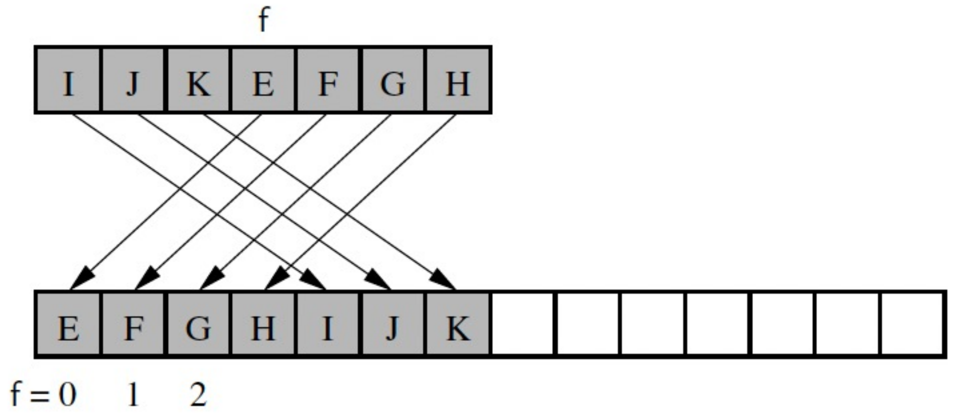
```
def dequeue(self):
        ”””Remove and return the first element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty.
        ”””
        if self.is_empty( ):
                raise Empty(“Queue is empty”)

        answer = self._data[self._front]
        self._data[self._front] = None                    # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size −= 1
        return answer
```

```
def enqueue(self, e):
        ”””Add an element to the back of queue.”””
        if self._size == len(self._data):
                self._resize(2 ∗ len(self._data))         # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
```

Resizing the underlying Array

Once the queue is full, double the size of the underlying list

Realign the front of the queue with index 0 in the new array



Resizing the underlying Array

```
def _resize(self, cap):                         # we assume cap >= len(self)
    """Resize to a new list of capacity >= len(self)."""

    old = self._data                    # keep track of existing list
    self._data = [None] * cap           # allocate list with new capacity
    walk = self._front

    for k in range(self._size):         # only consider existing elements
        self._data[k] = old[walk]           # intentionally shift indices
        walk = (1 + walk) % len(old)        # use old size as modulus
    self._front = 0                         # front has been realigned
```

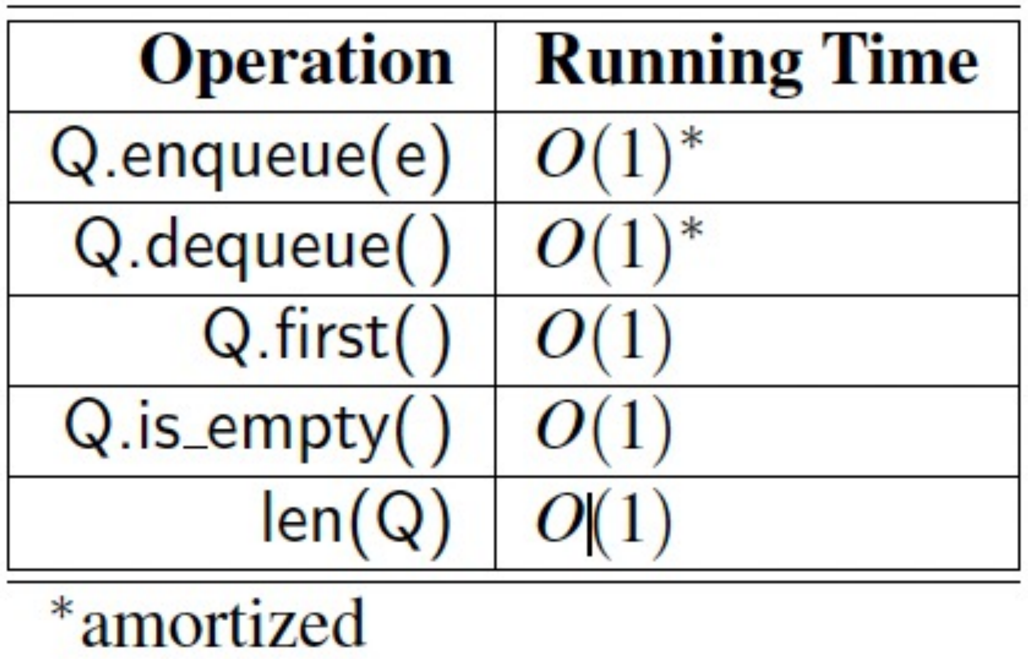
Shrinking the underlying Array

Shrinking the array when needed:

- Reduce the array to half of its current size, whenever the number of elements stored in it falls below one fourth of its capacity

```
if 0 < self._size < len(self._data) // 4:

       self._resize(len(self._data) // 2)
```

Complexities of circular queue operations



Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue()	$O(1)^*$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

*amortized

Types of Queues

- **Linear Queue**
- **Circular Queue**
- **Priority Queue**: all elements have a priority associated with them and are sorted such that the most prioritized object appears at the front and the least prioritized object appears at the end of the queue.
- **Double-ended Queue**: acts as a queue from both ends(back and front).

Deque

- Called Double-ended Queue
- Pronounced “Deck”
- A queue-like data structure that supports insertion and deletion at both the front and the back of the queue.

Deques ADT

Basic Function	What does it do?
add_first(element)	Inserts an element to the front of the deque
add_last(element)	Inserts an element to the back of the deque
delete_first()	Remove and return the first element from the deque
delete_last()	Remove and return the last element from the deque

Deque ADT

Additional Function	What does it do?
first()	Return the first element of the deque
last()	Return the last element of the deque
is_empty()	Checks is the deque is empty
size()	Returns the size of the deque

Deque implementation using Python's list

- Create a Deque.py file
- Construct a simple class MyDeque
- Instantiate an object of that class
- Use three instance variables: deque_list, size, front
- When needed, the back is computed
- Implement the **main** function for the Deque