Recursion, Stacks, Queues, Deques

Week 3

Week 3 Outline

- Logistics
- Algorithm complexity & Big-Oh notation
- Recursion, memoization, dynamic programming
- Array-based sequences:
  - Array class implementation
  - Stack
  - Queue
  - Deque

Logistics

- HW1:
  - save a copy of notebook, solve the assignment, download the notebook with solution to you computer, submit by uploading the downloaded file
- How to prep for the midterm:
  - Solve the Lab assignments without looking into the solution, understand Python related concepts
  - https://www.w3schools.com/python/default.asp
  - Weekly exercise assignments: not graded, prep for the midterm and the final, submission point with the deadline if you wish me to check your work
- Midterm announcement:
  - Week 5: 10 to 15 question for 15 points
  - Type of questions: T/F, multiple choice, fill in the answer e.g. calculate a value that function returns for the given value of arguments
  - essay (open) questions e.g write a function to calculate factorial; write out a comprehension to generate given list

Algorithm complexity & Big-Oh notation

- Count primitive operations, such as:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Asymptotic performance of the nonmutating and mutationg behaviors of the list class

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

Q1: O(n) = ?

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                          # create new list of n zeros
  for j in range(n):
    total = 0                          # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)               # record the average
  return A

def prefix_average2(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                          # create new list of n zeros
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1)       # record the average
  return A

def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                        # create new list of n zeros
  total = 0                          # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]                    # update prefix sum to include S[j]
    A[j] = total / (j+1)             # compute average based on current sum
  return A
```

A1: O(n) = ?

Q2: Time complexity

What is the time complexity expressed in Big-Oh notation?

- Code 1: document has n characters of which k are alphanumeric

```
letters = ''                    # start with empty string
for c in document:
  if c.isalpha():
    letters += c                # concatenate alphabetic character
```

- Code 2: document has n characters of which k are alphanumeric

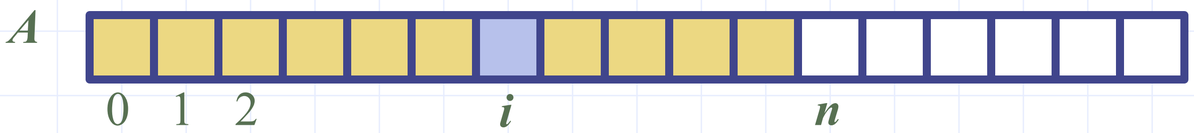
```
temp = []                       # start with empty list
for c in document:
  if c.isalpha():
    temp.append(c)              # append alphabetic character
letters = ''.join(temp)         # compose overall result
```

A2: Time complexity

What is the time complexity expressed in Big-O notation?

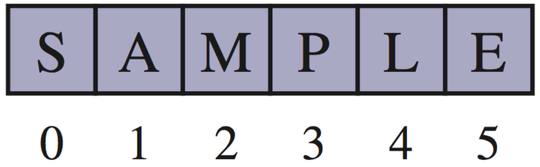
Python Sequence Classes

- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as A[i]
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.

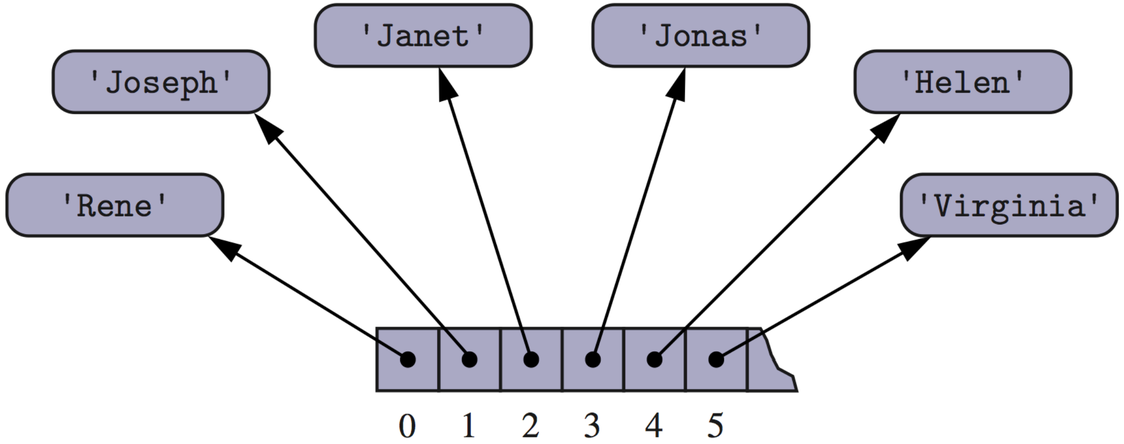


Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



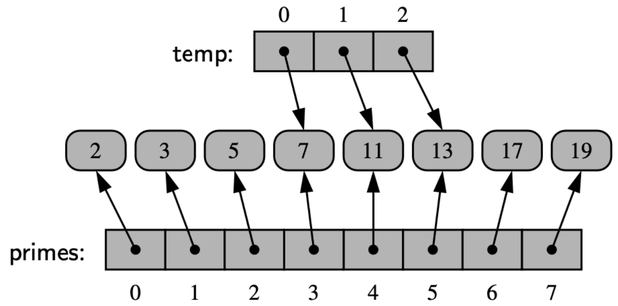


Figure 5.5: The result of the command temp = primes[3:6].

List slicing: the result is a new list instance, The new list has references to the same elements that are in the original list

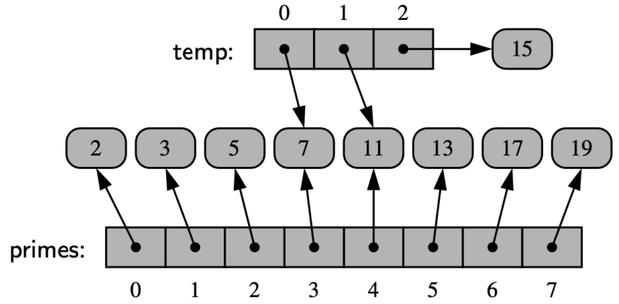


Figure 5.6: The result of the command temp[2] = 15 upon the configuration portrayed in Figure 5.5.

Reference temp[2] changes

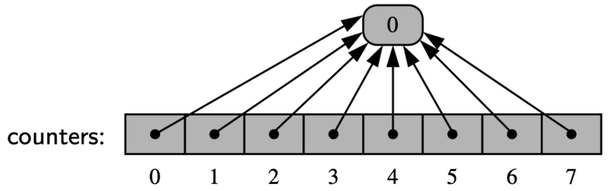


Figure 5.7: The result of the command data = [0] * 8.

All eight cells of the list reference the *same* object

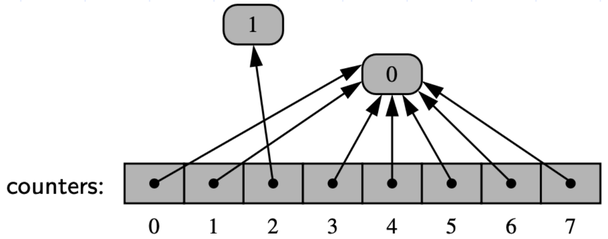


Figure 5.8: The result of command data[2] += 1 upon the list from Figure 5.7.

Q: What data types are mutable in Python?

Q3: A), B) or C), Why?

Question: How to instantiate multidimensional 3 x 6 array in Python?

A) data = ([0] ∗c) ∗r

B) data = [[0] ∗c] ∗r

C) data=[[0]*c **for** j **in** range(r)]

Suggestion:execute in Google Colab and check:

```
print(id(data[0][0]))
print(id(data[1][0]))
print(id(data[2][0]))

data[0][0] = 99
print(data)
```

A3:

Recursion

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a Python method:

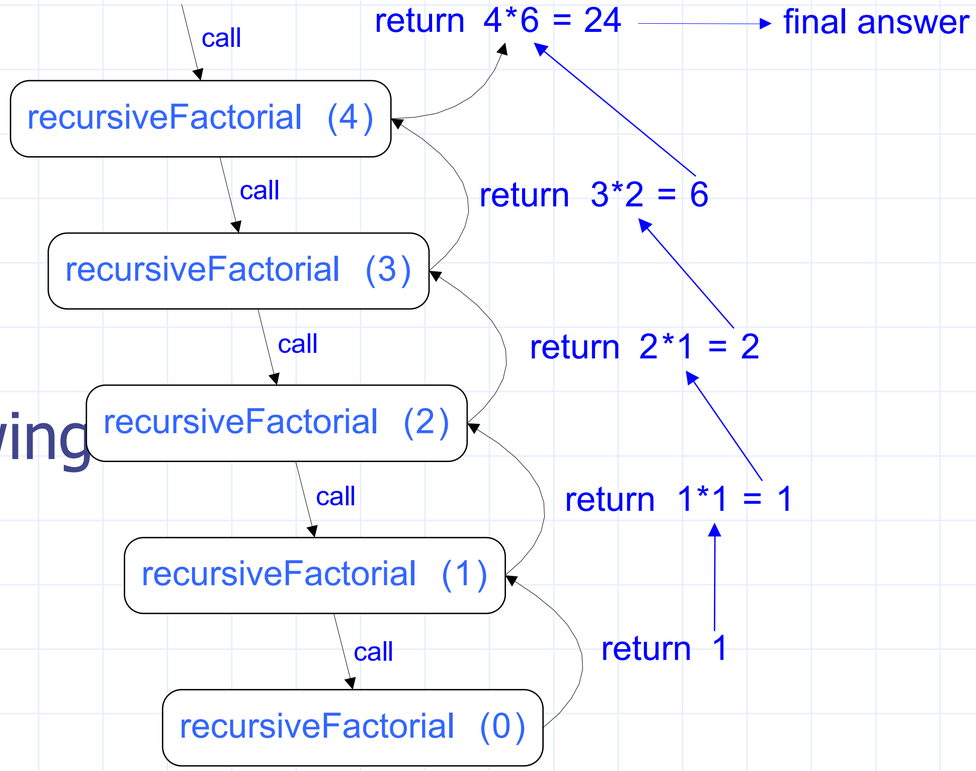
```
1  def factorial(n):
2    if n == 0:
3      return 1
4    else:
5      return n * factorial(n-1)
```

Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



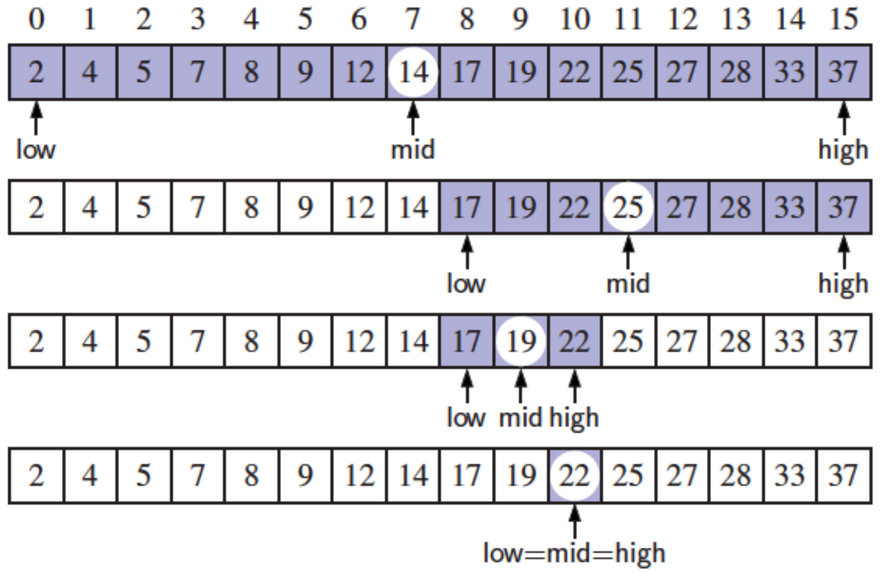
Binary Search

- Search for an integer, target, in an ordered list.

```
1  def binary_search(data, target, low, high):
2    """Return True if target is found in indicated portion of a Python list.
3
4    The search only considers the portion from data[low] to data[high] inclusive.
5    """
6    if low > high:
7      return False                          # interval is empty; no match
8    else:
9      mid = (low + high) // 2
10     if target == data[mid]:               # found a match
11       return True
12     elif target < data[mid]:
13       # recur on the portion left of the middle
14       return binary_search(data, target, low, mid - 1)
15     else:
16       # recur on the portion right of the middle
17       return binary_search(data, target, mid + 1, high)
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in O(log n) time.
  - The remaining portion of the list is of size high – low + 1.
  - After one comparison, this becomes one of the following:

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\leq\frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\leq\frac{\text{high}-\text{low}+1}{2}.$$

  - region in half; hence, there can be at most log n levels.

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).
- Python version:

```
1  def reverse(S, start, stop):
2    """Reverse elements in implicit slice S[start:stop]."""
3    if start < stop − 1:                          # if at least 2 elements:
4      S[start], S[stop−1] = S[stop−1], S[start]   # swap first and last
5      reverse(S, start+1, stop−1)                 # recur on rest
```

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):
    Input: An array A and nonnegative integer indices i and j
    Output: The reversal of the elements in A starting at index i and ending at j
    while i < j **do**
        Swap A[i] and A[j]
        i = i + 1
        j = j - 1
    return

Arrays

Compact Arrays

- Primary support for compact arrays is in a module named **array**.
  - That module defines a class, also named array, providing compact storage for arrays of primitive data types.
- The constructor for the array class requires a type code as a first parameter, which is a character that designates the type of data that will be stored in the array.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

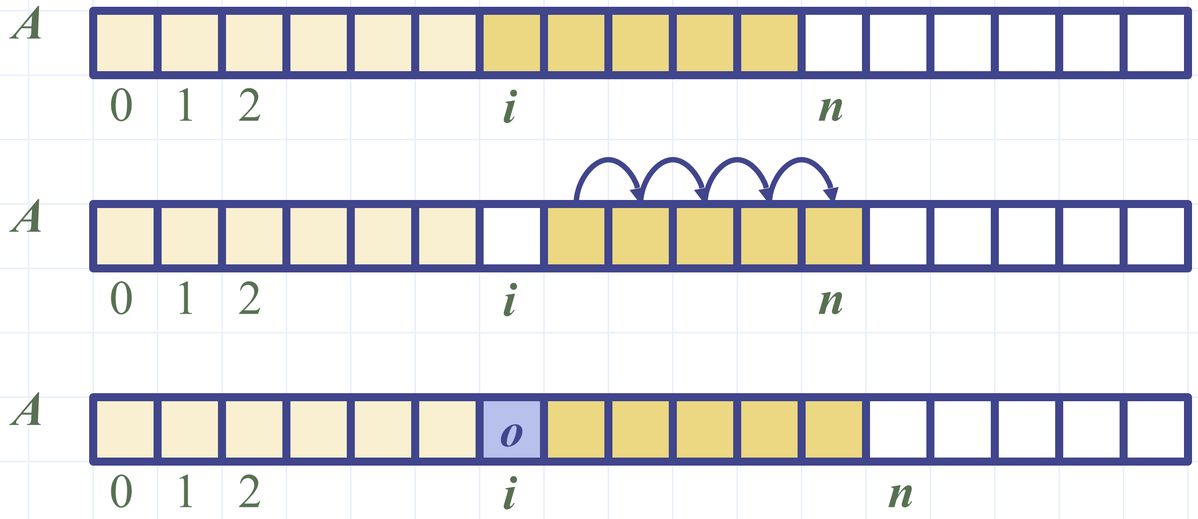
Type Codes in the array Class

- Python's array class has the following type codes:

Code	C Data Type	Typical Number of Bytes
`'b'`	signed char	1
`'B'`	unsigned char	1
`'u'`	Unicode char	2 or 4
`'h'`	signed short int	2
`'H'`	unsigned short int	2
`'i'`	signed int	2 or 4
`'I'`	unsigned int	2 or 4
`'l'`	signed long int	4
`'L'`	unsigned long int	4
`'f'`	float	4
`'d'`	float	8

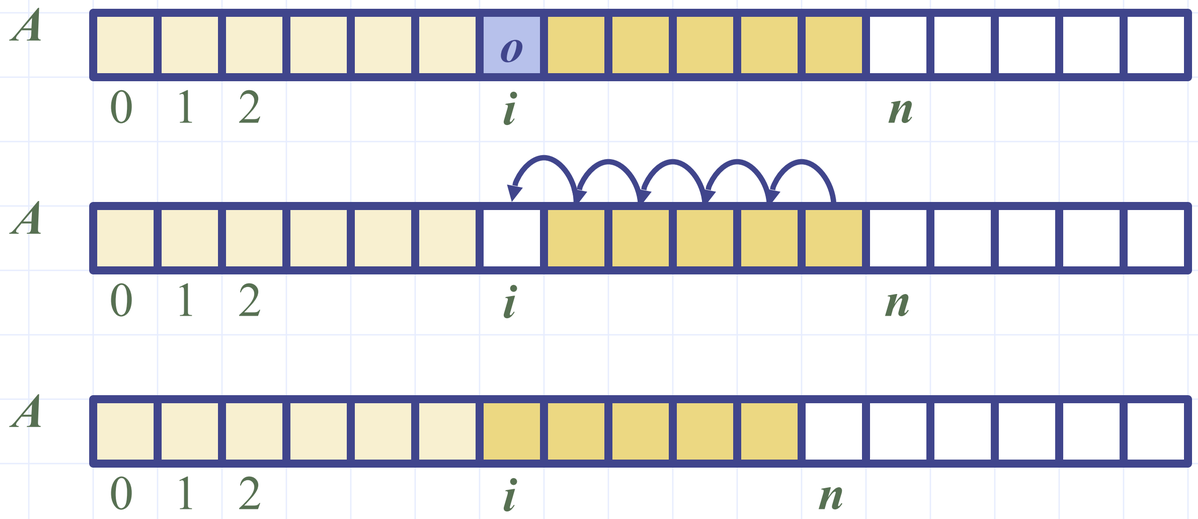
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In an array based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at I takes $O(1)$ time
  - *add* and *remove* run in $O(n)$ time in worst case
- In an *add* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *add*(o)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

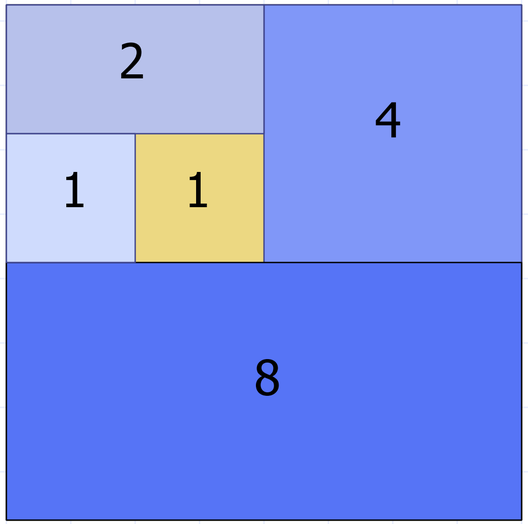
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Python Implementation

```
1  import ctypes                                  # provides low-level arrays
2
3  class DynamicArray:
4    """A dynamic array class akin to a simplified Python list."""
5
6    def __init__(self):
7      """Create an empty array."""
8      self._n = 0                                # count actual elements
9      self._capacity = 1                         # default array capacity
10     self._A = self._make_array(self._capacity) # low-level array
11
12   def __len__(self):
13     """Return number of elements stored in the array."""
14     return self._n
15
16   def __getitem__(self, k):
17     """Return element at index k."""
18     if not 0 <= k < self._n:
19       raise IndexError('invalid index')
20     return self._A[k]                          # retrieve from array
21
22   def append(self, obj):
23     """Add object to end of the array."""
24     if self._n == self._capacity:              # not enough room
25       self._resize(2 * self._capacity)         # so double capacity
26     self._A[self._n] = obj
27     self._n += 1
28
29   def _resize(self, c):                        # nonpublic utitity
30     """Resize internal array to capacity c."""
31     B = self._make_array(c)                    # new (bigger) array
32     for k in range(self._n):                   # for each existing value
33       B[k] = self._A[k]
34     self._A = B                                # use the bigger array
35     self._capacity = c
36
37   def _make_array(self, c):                    # nonpublic utitity
38     """Return new array with capacity c."""
39     return (c * ctypes.py_object)( )           # see ctypes documentation
```

Asymptotic performance of the nonmutating and mutating behaviors

Python lists of length *n* and with element *value* at index *k*

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

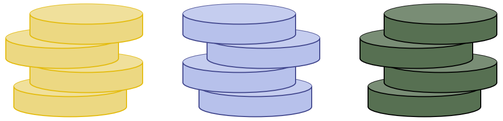
*amortized

Linear Data Structures

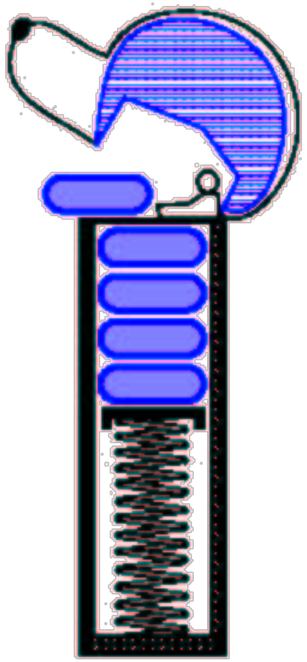
Linear data structures

- Stacks, queues, deques, linked lists
- Data collections
- Items ordered depending on how they are added or removed

Stacks

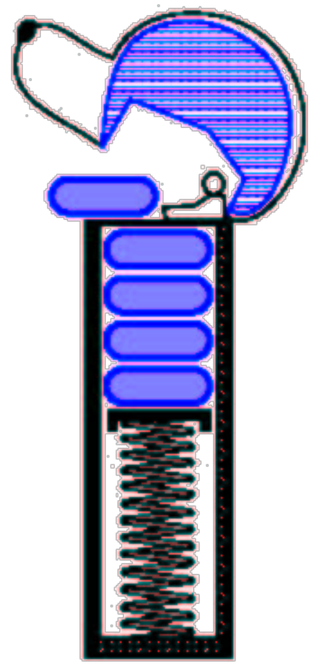


- An abstract data type (ADT) is an abstraction of a data structure
- Stack is ADT
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations



The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow **the last-in first-out scheme (LIFO)**
  - Think of a spring-loaded plate dispenser
- Main stack operations:
  - **push(object)**: inserts an element
  - **object pop()**: removes and returns the last inserted element
- Auxiliary stack operations:
  - object top() (peek()): returns the last inserted element without removing it
  - integer len()(size()): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored



Stack Example

TO DO: Fill in the columns Return Value and Stack Contents

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )		
S.is_empty( )		
S.pop( )		
S.is_empty( )		
S.pop( )		
S.push(7)		
S.push(9)		
S.top( )		
S.push(4)		
len(S)		
S.pop( )		
S.push(6)		
S.push(8)		
S.pop( )		

After performing all ops on stack as given in the table, answer:

len(S)=?
Last Return Value =?
Stack Contents =?

Stack Example

Applications of Stack

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language that supports recursion
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$ (amortized in the case of a push)

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

Array-based Stack in Python

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
```

```
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

Symbol Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

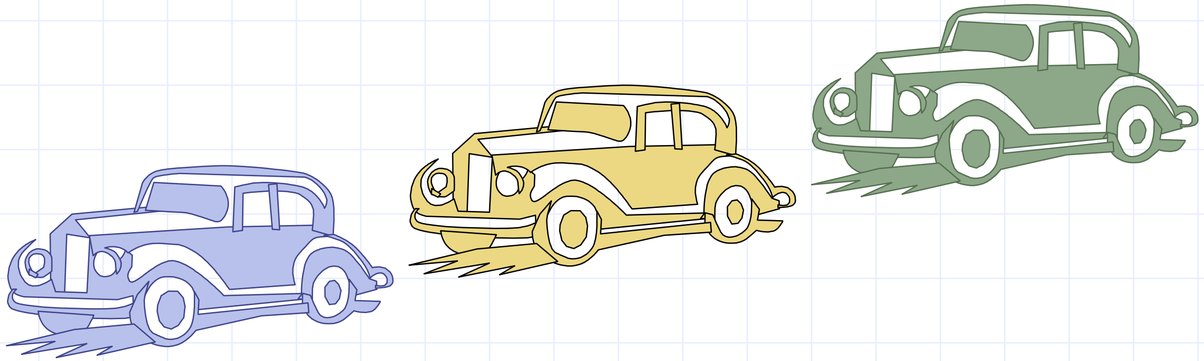
Parentheses Matching Algorithm Pseudocode

Algorithm ParenMatch(*X,n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
    if *X*[*i*] is an opening grouping symbol **then**
        S.push(*X*[*i*])
    else if *X*[*i*] is a closing grouping symbol **then**
        if *S*.is_empty() **then**
            return false {nothing to match with}
        if *S*.pop() does not match the type of *X*[*i*] **then**
            return false {wrong type}
if *S*.isEmpty() **then**
    return true {every symbol matched}
else return false {some symbols were never matched}

Parentheses Matching in Python

```
1  def is_matched(expr):
2    """Return True if all delimiters are properly match; False otherwise."""
3    lefty = '({['                                  # opening delimiters
4    righty = ')}]'                                 # respective closing delims
5    S = ArrayStack( )
6    for c in expr:
7      if c in lefty:
8        S.push(c)                                  # push left delimiter on stack
9      elif c in righty:
10       if S.is_empty( ):
11         return False                             # nothing to match with
12       if righty.index(c) != lefty.index(S.pop( )):
13         return False                             # mismatched
14   return S.is_empty( )                           # were all symbols matched?
```

Queues



Queues

- A **queue** is an ordered collection
- the addition of new items happens at one end, called the back, and the removal of existing items occurs at the other end, commonly called the front
- Insertions and deletions follow **the first-in first-out scheme (FIFO)**

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

The Queue Example

Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	
Q.is_empty( )	False	
Q.dequeue( )	3	
Q.is_empty( )	True	
Q.dequeue( )	"error"	
Q.enqueue(7)	–	
Q.enqueue(9)	–	
Q.first( )	7	
Q.enqueue(4)	–	
len(Q)	3	
Q.dequeue( )	7	

After performing all ops on dequeue as given in the table, answer:

len(Q)=?
Last Return Value =?
Q Content =?

Example

Applications of Queues

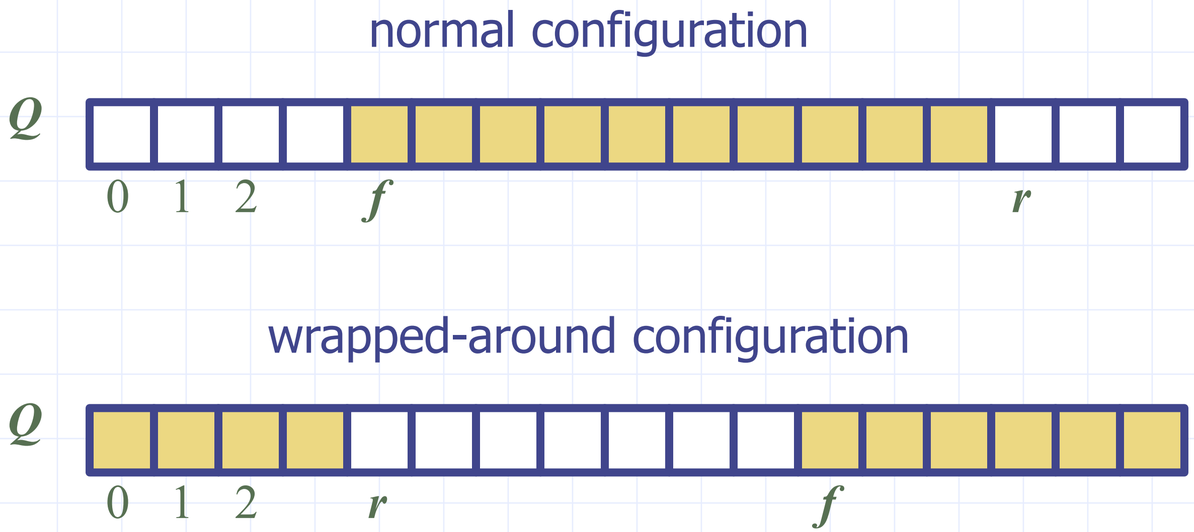
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear

  f index of the front element

  r index immediately past the rear element
- Array location r is kept empty

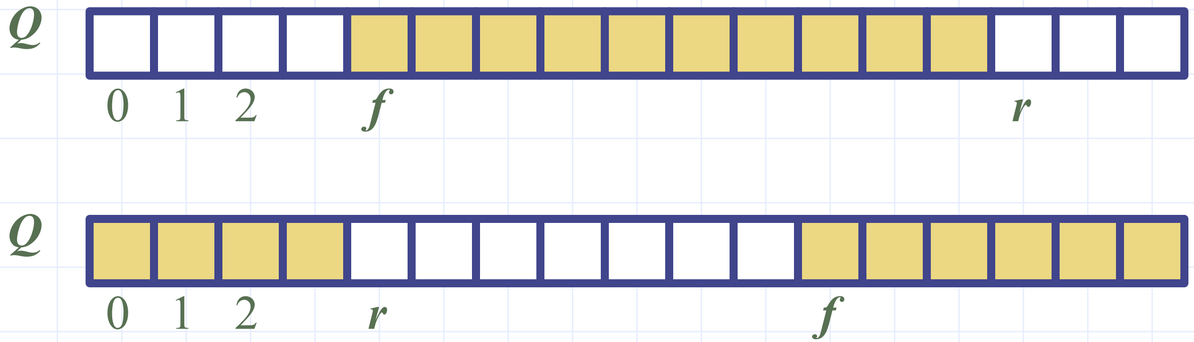


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

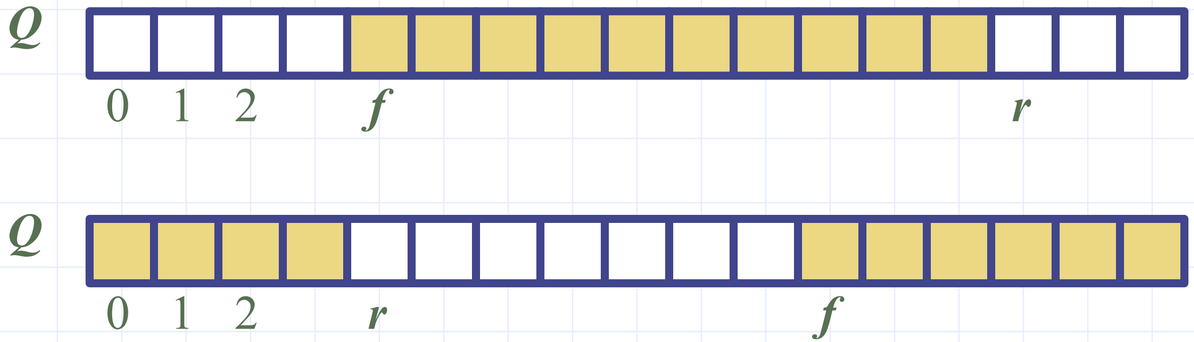
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

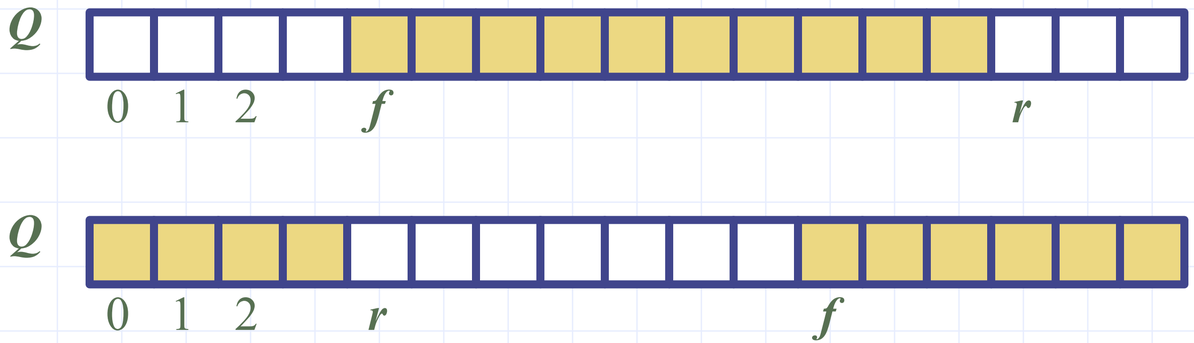
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *FullQueueException*
  else
    Q[*r*] ← *o*
    r ← (*r* + 1) mod *N*



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
    if isEmpty() then
        throw EmptyQueueException
    else
        o ← Q[f]
        f ← (f + 1) mod N
        return o
```



Queue in Python

- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - _front: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).

Queue in Python, Beginning

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
```

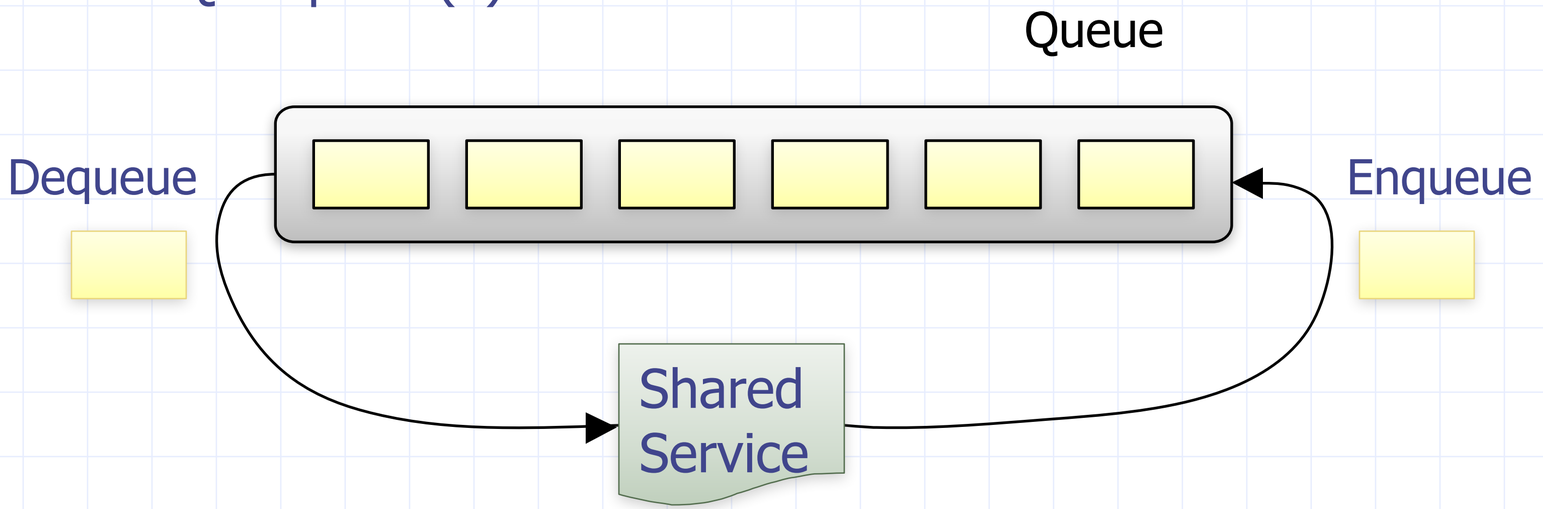
```
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
27
28   def dequeue(self):
29     """Remove and return the first element of the queue (i.e., FIFO).
30
31     Raise Empty exception if the queue is empty.
32     """
33     if self.is_empty():
34       raise Empty('Queue is empty')
35     answer = self._data[self._front]
36     self._data[self._front] = None          # help garbage collection
37     self._front = (self._front + 1) % len(self._data)
38     self._size -= 1
39     return answer
```

Queue in Python, Continued

```
40    def enqueue(self, e):
41      """Add an element to the back of queue."""
42      if self._size == len(self._data):
43        self._resize(2 * len(self.data))          # double the array size
44      avail = (self._front + self._size) % len(self._data)
45      self._data[avail] = e
46      self._size += 1
47
48    def _resize(self, cap):                       # we assume cap >= len(self)
49      """Resize to a new list of capacity >= len(self)."""
50      old = self._data                            # keep track of existing list
51      self._data = [None] * cap                   # allocate list with new capacity
52      walk = self._front
53      for k in range(self._size):                 # only consider existing elements
54        self._data[k] = old[walk]                 # intentionally shift indices
55        walk = (1 + walk) % len(old)              # use old size as modulus
56      self._front = 0                             # front has been realigned
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Double-Ended Queues

Deques

- Double-ended queue ==> deque (pronounced “deck”)
- Unrestrictive nature of adding and removing items

The Deque Abstract Data Type

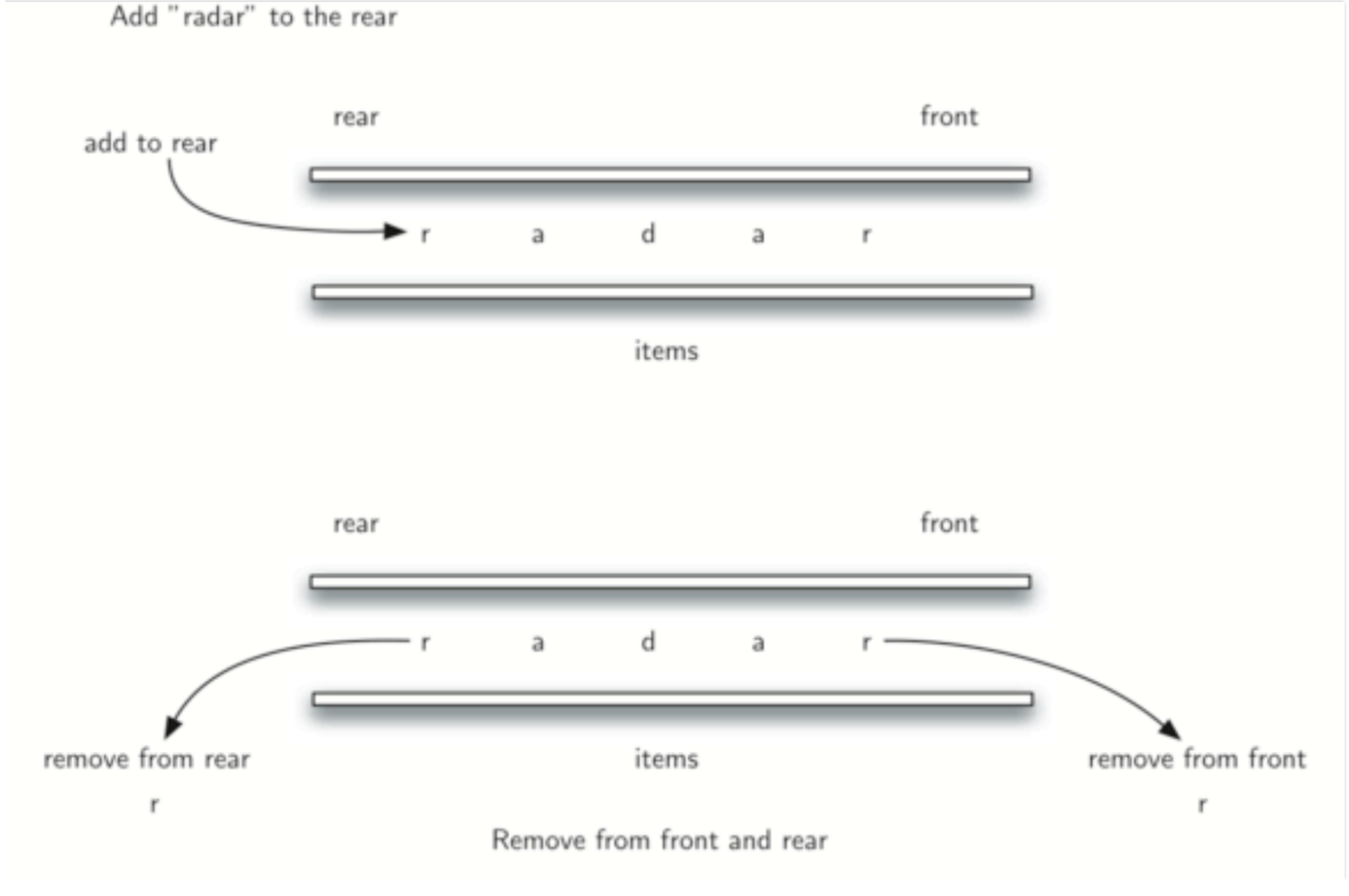
`Deque()` creates a new deque that is empty. It needs no parameters and returns an empty deque.
`add_front(item)` adds a new item to the front of the deque. It needs the item and returns nothing.
`add_rear(item)` adds a new item to the rear of the deque. It needs the item and returns nothing.
`remove_front()` removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
`remove_rear()` removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
`is_empty()` tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
`size()` returns the number of items in the deque. It needs no parameters and returns an integer.

Deque Example

See Tutorial: Palindrome checker

https://runestone.academy/ns/books/published/pythonds3/BasicDS/PalindromeChecker.html

https://github.com/psads/pythonds3/blob/master/pythonds3/basic/deque.py



The Deque Python Implementation

```python
class Deque:
    """Deque implementation as a list"""

    def __init__(self):
        """Create new deque"""
        self._items = []

    def is_empty(self):
        """Check if the deque is empty"""
        return not bool(self._items)

    def add_front(self, item):
        """Add an item to the front of the deque"""
        self._items.append(item)

    def add_rear(self, item):
        """Add an item to the rear of the deque"""
        self._items.insert(0, item)

    def remove_front(self):
        """Remove an item from the front of the deque"""
        return self._items.pop()

    def remove_rear(self):
        """Remove an item from the rear of the deque"""
        return self._items.pop(0)

    def size(self):
        """Get the number of items in the deque"""
        return len(self._items)
```

Week 3 Reading

Problem Solving with Algorithms and Data Structures using Python, by B. Miller and D. Ranum

Chapter 3. Basic Data Structures

Data Structures and Algorithms in Python, by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser
Chapter 5 Array-Based Sequences
Chapter 6 Stacks,Queues, Deques
Chapter 7 Linked Lists (Week 4)