Algorithm analysis
Stack, Queue
Linked List

2021.01.17
Data eXperience Lab, Winter Seminar
2018312824 Ryu Caeeunun

OUTLINE

1. Algorithm analysis

2. Stack, Queue

3. Linked List

Chapter 3. Algorithm Analysis

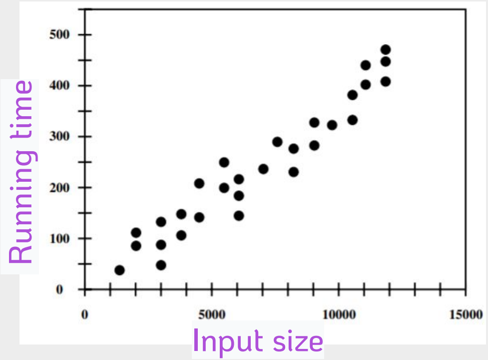
Algorithm analysis : Experimental Studies

- **Data structure**: a systematic way of organizing and accessing data.
- **Algorithm**: a step-by-step procedure for performing some task in a finite amount of time.

```
import time
def run_algorithm():
    print("algorithm running...")
    time.sleep(3)

start_time= time.time()
run_algorithm()
end_time = time.time()
elapsed = end_time-start_time
print("elapsed:{}".format(elapsed))
```

```
algorithm running...
elapsed:3.0700299739837646
```

Running time is a natural measure of "goodness" - computer solutions should run as fast as possible. In general, the running time increases with the input size.



We can study its running time by executing it on various test inputs and recording the time spent during each execution, but limitations to experimental study exist.

ex.) difficulty of comparison, limitation on the set of test inputs, necessity of full implementation

Beyond Experimental Analysis

Counting Primitive Operations

Primitive operations:

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function (excluding operations executed within the function)
- Returning from a function.

Measure of the running time: ***t***, count how many primitive operations are executed

Assumption: the running times of different primitive operations will be fairly similar, thus t will be proportional to the actual running time of that algorithm.

Beyond Experimental Analysis

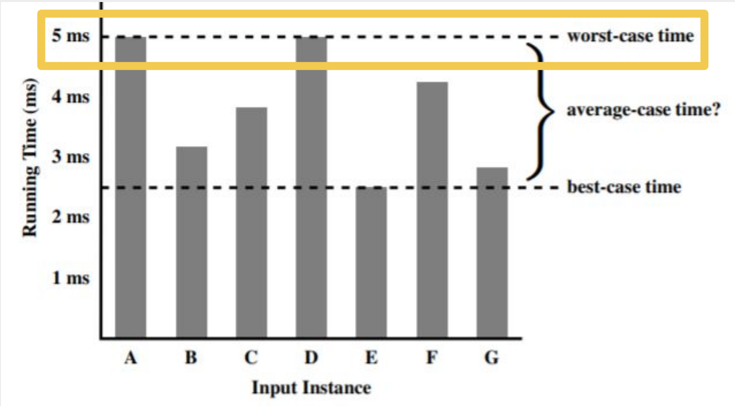
Measuring Operations as a Function of Input Size

Associate, with each algorithm, a function f(n) that characterizes the number of primitive operations that are performed as a function of the input size n.

Focusing on the Worst-Case Input

An algorithm may run faster on some inputs than it does on others of the same size.

Worst-case analysis is much easier(more simple) than average-case analysis



Success on this standard:

Guarantees that the algorithm will do well on every input.

7 Functions for the analysis of algorithms

The constant function

The Logarithm function

The Linear Function

The N-Log-N Function

The Quadratic Function

The Cubic Function and Other Polynomials

The Exponenti
al
Function

Measuring time complexity for each of the algorithms can help us decide which one is the most efficient and thus better in terms of performance.

The Constant & Linear Function

The Constant Function

***f*(*n*) = C**

- does <u>not depend on the input size(n)</u>.
- useful when we need to count the number of basic operations(ex. variable assignment, integer addition or subtraction) executed by an algorithm.
- g(n) = 1, f(n) = c x g(n)

```
#examples of basic operations
x = 10
name = 'Andrew'
is_verified = True
```

The Linear Function

***f*(*n*) = n**

- assigns the input(n) given to itself.
- useful when it comes to analysing an operation that needs to be performed over all n elements.

```
#linear function algorithm analysis
my_list = [1, 2, 3, 4, 5]
for i in my_list:
    print(i == 2)
```

```
False
True
False
False
False
```

The Logarithm Function

The Logarithm Function(The Log Function)

$$f(n) = log_b n$$ base

$$\text{where } b > 1 \text{ and } x = log_b n \text{ iff } n = b^x$$

- The most common base: 2 (computers store integers in binary and because the common operation in many algorithms is to repeatedly divide an input in half.)

$$\log n = \log_2 n.$$

- **Ceiling**: translates to the smallest integer greater than or equal to the logarithm.

when it reduces the size of the input data in each step (no need to look at all values of the input data)

```
def binary_search(data, value):
    n = len(data)
    left = 0
    right = n - 1
    while left <= right:
        middle = (left + right) // 2
        if value < data[middle]:
            right = middle - 1
        elif value > data[middle]:
            left = middle + 1
        else:
            return middle
    raise ValueError('Value is not in the list')
```

Floor function: $\lfloor x \rfloor$ = the largest integer less than or equal to x.
Ceiling function: $\lceil x \rceil$ = the smallest integer greater than or equal to x.

Log 18 ≅ 5
= (((((18/2)/2)/2)/2)/2)
= 0.5625 ≤ 1

The Quadratic & N-Log-N

The Quadratic Function

$$f(n) = n^2$$

- that assigns itself the square of the input ***n***.
- used to describe the complexity of **nested loops(**a sequence of n operations which are performed n times.)

Sum of number of operations in each iteration

$$1+2+3+\cdots+(n-2)+(n-1)+n.$$

$$\frac{n(n+1)}{2}$$

ex)

```
for x in data:
    for y in data:
        print(x, y)
```

The N-Log-N Function

f(*n*) **= n log n**

- assigns to an input n the value of n times the logarithm base-two of n.
- n grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function.
- ex) mergesort, heapsort

The Cubic Function & Other Polynomials

The Cubic Function

$$f(n) = n^3$$

- assigns to an input value n the product of n with itself three times.
- appears less frequently.
- poor performance

```
def disjoint1(A, B, C):
  """Return True if there
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False
  return True
```

The Quadratic Function

Sum of number of operations in each iteration

$$1+2+3+\cdots+(n-2)+(n-1)+n.$$

$$\frac{n(n+1)}{2}$$

Polynomials Function

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ \ldots \ + a_d n^d$$

examples

- N: input, a: coefficient, d: degree

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Summations

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

- $a \leq b$
- the running times of loops naturally give rise to summations.

examples

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

The Exponential F, Geometric Sums

The Exponential Function

$$f(n) = b^n$$

- b: base, n: exponent
- assigns to the input argument n the value obtained by multiplying the base b by itself n times.
- Default base: 2

ex)

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Geometric Sums

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

$$\frac{a^{n+1} - 1}{a - 1}$$

- For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a = 1$
- each term is geometrically larger than the previous one if $a > 1$

Asymptotic Analysis: The "Big-Oh"

Asymptotic Analysis →

O	Ω	Θ
Big Oh	Omega	Theta
Upper Bound	Lower Bound	Upper & Lower Bound
Worst Case	Best Case	Average 'ish'

- evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time)

The "Big-Oh" Notation

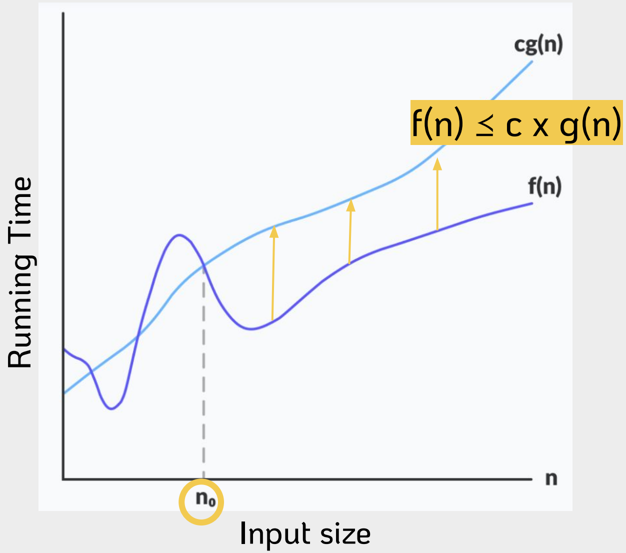
- **"Less than or equal to"**

$$f(n) \text{ is } O(g(n))$$ **Or "f(n) is order of g(n)"**

- If c > 0, n0 ≥ 1 such that,

$$f(n) \leq c \cdot g(n) \text{ when } n \geq n_0$$

- provides an upper bound on a function ensuring that the function never grows faster than the upper bound. -> measures the worst-case complexity of the algorithm.



Asymptotic Analysis: The “Big-Oh”

The “Big-Oh” Notation Properties and Examples

We should use the big-Oh notation to characterize a function as closely as possible.

```python
def find_max(data):
    """Return the maximum element from a nonempty Python list."""
    biggest = data[0]#start searching from the front
    for val in data:
        if val>biggest:
            biggest = val
    return biggest
```

O(n)

$$5n^4 + 3n^3 + 2n^2 + 4n + 1$$

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \le \overbrace{\boxed{(5+3+2+4+1)}}^{=15} n^4$$

C = 15, n0 = 1

$$O(n^4)$$

$$a_0 + a_1 n + \cdots + a_d n^d \longrightarrow O(n^d)$$

$$5n^2 + 3n\log n + 2n + 5 \longrightarrow O(n^2)$$

```python
if a > b:
    return True
else:
    return False
```

O(1)

best

poor

Name	Time Complexity
Constant Time	O(1)
Logarithmic Time	O(log n)
Linear Time	O(n)
Quasilinear Time	O(n log n)
Quadratic Time	O(n^2)
Exponential Time	O(2^n)
Factorial Time	O(n!)

Asymptotic Analysis: Big-Omega

O	Ω	Θ
Big Oh	Omega	Theta
Upper Bound	Lower Bound	Upper & Lower Bound
Worst Case	Best Case	Average 'ish'

Big-Omega

- **"Greater than or equal to"**

$f(n) \text{ is } \Omega(g(n))$ = " f(n) is big-Omega of g(n),"

- **if g(n) is O(f(n)), that is, there is a real constant c > 0 and an integer constant n0 ≥ 1 such that**

$$f(n) \geq cg(n), \quad \text{for} \quad n \geq n_0.$$

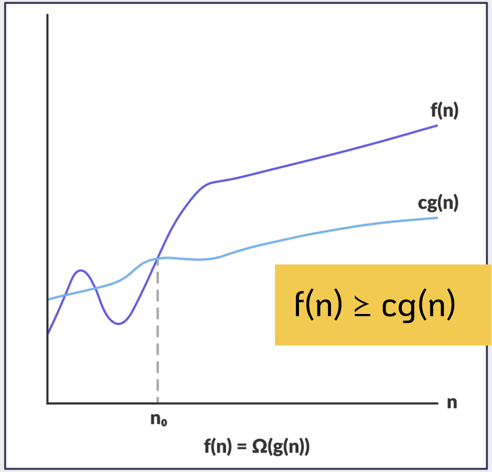
Example

$$3n\log n - 2n$$

$$3n\log n - 2n = n\log n + 2n(\log n - 1)$$

When $n \geq 2$:

$n\log n + 2n(\log n - 1) \geq n\log n$ → C = 1, n0 = 2 → $\Omega(n\log n)$



Asymptotic Analysis: Big-Theta

O	Ω	Θ
Big Oh	Omega	Theta
Upper Bound	Lower Bound	Upper & Lower Bound
Worst Case	Best Case	Average 'ish'

Big-Theta

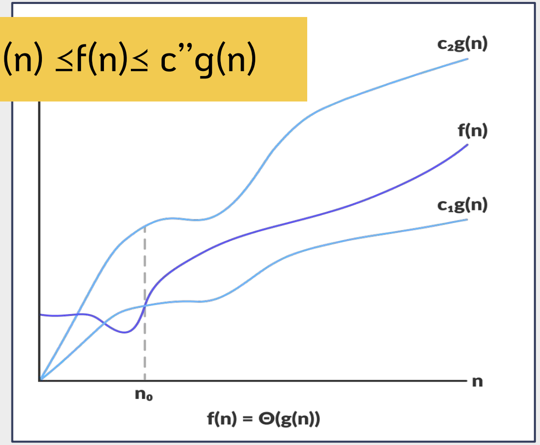
- **"Greater than or equal to"**

$f(n)$ is $\Theta(g(n))$ = " f(n) is big-Theta of g(n),"

- **if f(n) is O(g(n)) and f(n) is Ω(g(n)) , that is, there are real constants c > 0 and c > 0, and an integer constant n0 ≥ 1 such that**

$$c'g(n) \le f(n) \le c''g(n), \quad \text{for} \quad n \ge n_0.$$

c'g(n) ≤f(n)≤ c''g(n)



Example

→ $3n\log n + 4n + 5\log n$

→ When n≥2 :

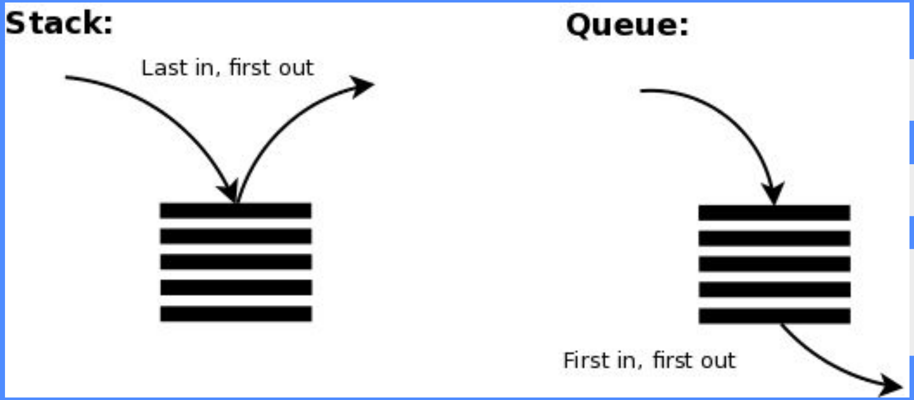
$$3n\log n \le 3n\log n + 4n + 5\log n \le (3+4+5)n\log n$$

→ C' = 3, c'' = 12 → $\Theta(n\log n)$

Time Complexity Summary

Average time complexity of different data structures covered today for different operations

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)



Stack & Queue

STACKS: Definition and Properties

Stacks

- last-in, first-out (LIFO) principle.

Methods

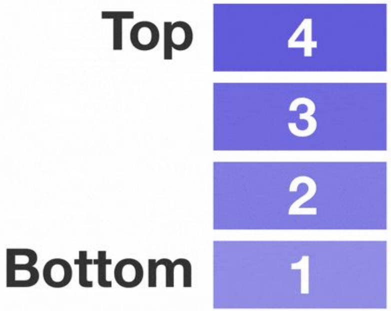
S.push(e): Add element e to the top of stack S.

S.pop(): Remove and return the top element from the stack S; an error occurs if the stack is empty.

S.top(): Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.

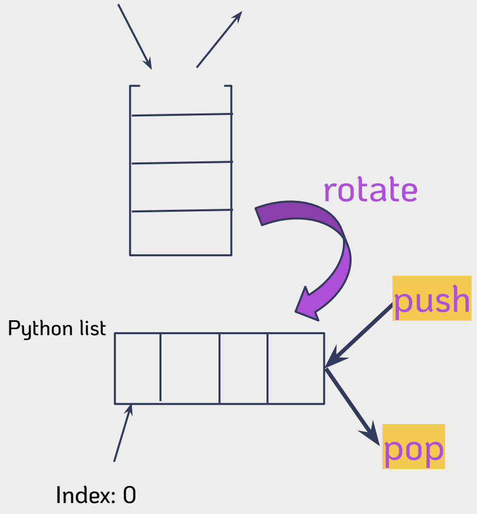
S.is_empty(): Return True if stack S does not contain any elements.

len(S): Return the number of elements in stack S; in Python, we implement this with the special method __len__.



STACKS: Implementation

Stacks in Array



Store data in python list

Append: insert at the end of array

Get last element

Remove last element

code

```
class Empty(Exception):
    """Error attempting to access an element from an empty container"""
    pass

class ArrayStack:
    def __init__(self):
        """create an empty stack."""
        self._data = []

    def __len__(self):
        """Return the number of elements in the stack."""
        return len(self._data)

    def is_empty(self):
        """return true if stack is empty"""
        return len(self._data) == 0

    def push(self, e):
        print("push:",e)
        """add element e to the top of the stack."""
        return self._data.append(e)

    def top(self):
        """
        return (but do not remove) the element at the top of the stack.
        raise empty function if the stack is empty.
        """
        if self.is_empty():
            raise Empty("Stack is Empty")
        return self._data[-1]

    def pop(self):
        """
        Remove and return the element from the top of the stack
        Raise Empty exception if the stack is empty
        """
        if self.is_empty():
            raise Empty("Stack is empty")
        print("popped:",self._data[-1])
        return self._data.pop()

    def visualize(self):
        print("₩n-DISPLAY STACK-")
        for i in range(len(self._data)):
            i+=1
            print("|",self._data[-1*i],"|")
            print("-----")
```

output

```
push: 5
push: 3

-DISPLAY STACK-
| 3 |
-----
| 5 |
-----
popped: 3

-DISPLAY STACK-
| 5 |
-----
is_empty(): False
popped: 5
push: 7
push: 9

-DISPLAY STACK-
| 9 |
-----
| 7 |
-----
```

STACKS: Analyzation

Analyzation

- Amortized bounds; occasionally an O(n)-time worst case, where n is the current number of elements in the stack.

Alternative

- Specify the maximum size of stack.

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

Queues: Definition and Properties

Queues

- first-in, first-out (FIFO) principle.

Methods

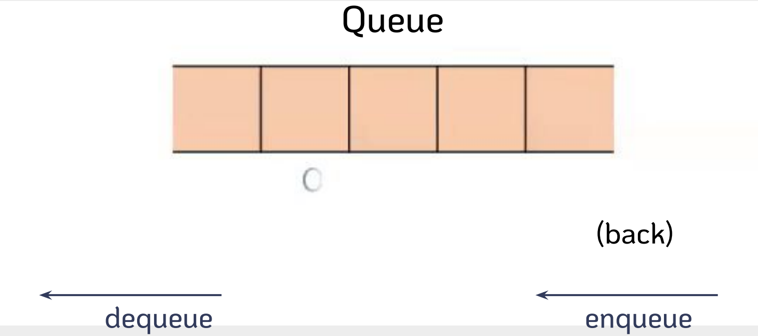
Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue( ): Remove and return the first element from queue Q; an error occurs if the queue is empty.

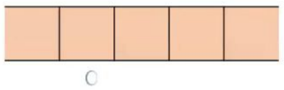
Q.first( ): Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

Q.is_empty( ): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method __len__.



QUEUES: Implementation(Code)



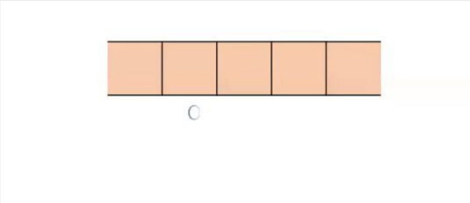
```
Display Queue
<-front --- back<-
====================
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
====================
enqueue: 11 at 10 current front: 0

Display Queue
<-front --- back<-
====================
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, None, None, None, None, None, None, None, None, None]
====================
dequeued: 1
dequeued: 2
enqueue: 13 at 11 current front: 2
```

```python
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")#Raise EmptyException if the queue is empty.
        print("first: {}".format(self._data[self._front]))
        return self._data[self._front]

    def dequeue(self):
        """Remove and return the first element of the queue(i.e., FIFO)."""
        if self.is_empty():
            raise Empty('Aueue is empty')#Raise Empty exception if the queue is empty.
        answer = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1)%len(self._data)
        self._size -=1
        print("dequeued: {}".format(answer))
        return answer

    def _resize(self, cap):
        """Resize to a new list of capacity >= len(self)."""
        old = self._data
        self._data = [None]*cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1+walk)%len(old)
        self._front = 0

    def enqueue(self, e):
        print("enqueue: {}".format(e))
        """Add an element to the back of queue"""
        if self._size == len(self._data):
            self._resize(2*len(self._data))
        avail = (self._front + self._size)%len(self._data)
        self._data[avail] = e
        self._size += 1
```

QUEUES:

Implementation(output)

```python
Q = ArrayQueue()
for i in range(10):
    Q.enqueue(i+1)
Q.displayQ()
Q.enqueue(11)
Q.displayQ()
Q.dequeue()
Q.dequeue()
Q.displayQ()
```




```
enqueue: 1
enqueue: 2
enqueue: 3
enqueue: 4
enqueue: 5
enqueue: 6
enqueue: 7
enqueue: 8
enqueue: 9
enqueue: 10

Display Queue
<-front --- back<-
====================
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
====================
enqueue: 11

Display Queue
<-front --- back<-
====================
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, None, None, None, None, None, None, None, None, None]
====================
dequeued: 1
dequeued: 2

Display Queue
<-front --- back<-
====================
[None, None, 3, 4, 5, 6, 7, 8, 9, 10, 11, None, None, None, None, None, None, None, None, None]
====================
```

Enqueue 10 elements

size : 10 -> size: 20

QUEUES: Analyzation

Analyzation

- Amortized bounds; occasionally an O(n)-time worst case, where n is the current number of elements in the stack.

Alternative

- Specify the maximum size of stack.

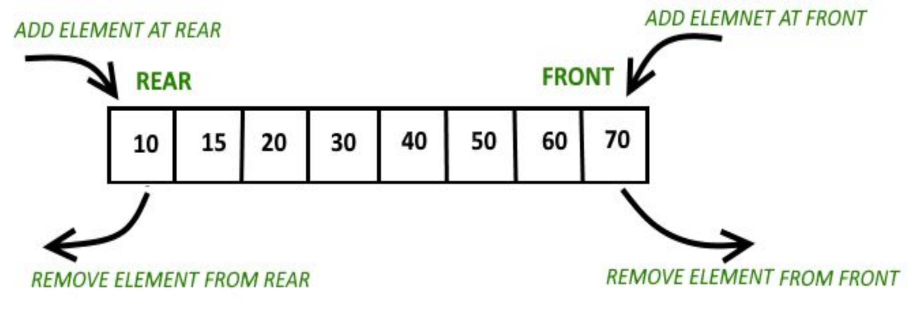
Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue()	$O(1)^*$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

*amortized

Double-Ended Queues

Double-Ended Queues (Deque)

- supports insertion and deletion at both the front and the back of the queue.



Methods

D.add_first(e): Add element e to the front of deque D.

D.add_last(e): Add element e to the back of deque D.

D.delete_first(): Remove and return the first element from deque D; an error occurs if the deque is empty.

D.delete_last(): Remove and return the last element from deque D; an error occurs if the deque is empty.

D.first(): Return (but do not remove) the first element of deque D; an error occurs if the deque is empty.

D.last(): Return (but do not remove) the last element of deque D; an error occurs if the deque is empty.

D.is_empty(): Return True if deque D does not contain any elements.

len(D): Return the number of elements in deque D; in Python, we implement this with the special method __len__.

Linked List

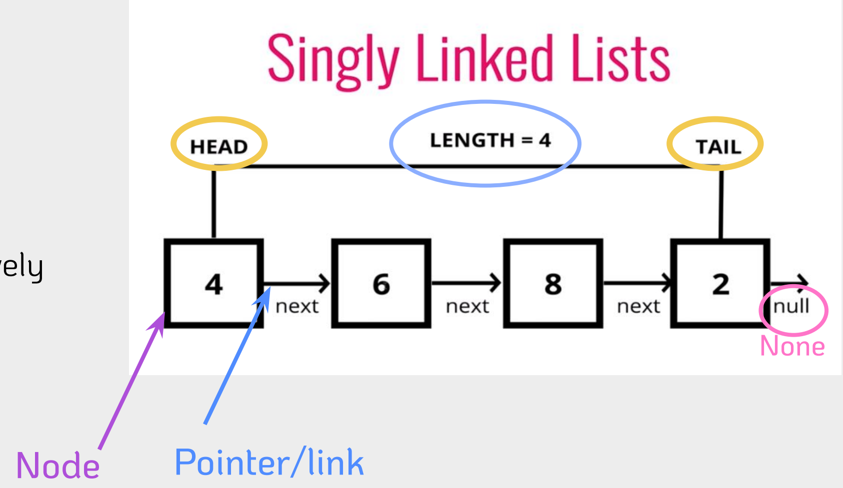
Recursion is the process of defining something in terms of itself.

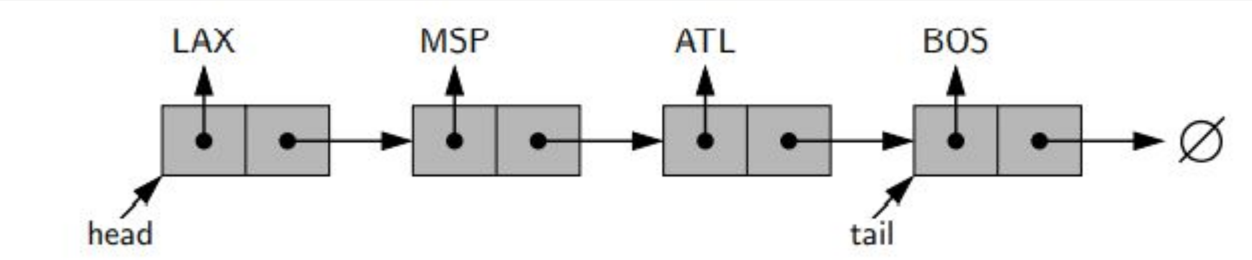
SINGLY LINKED LIST: Definition

Singly Linked List

Definition and properties:

- Node: a unique object in linked list.
- Singly Linked List: a collection of nodes that collectively form a linear sequence
- Link, pointer: next reference of a node
- Head: first node, tail: last node
- Traverse: Head->Tail
- size/length: Total number of nodes in a list





SINGLY LINKED LIST: Insertion

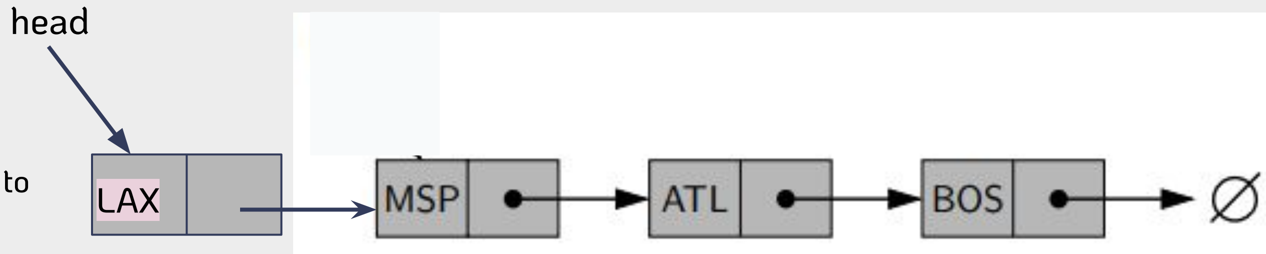
Singly Linked List: Insertion

- **At the head**
- **At the tail**

Inserting an Element at the Head of a Singly Linked List

```
Algorithm add_first(L, e):
  newest = Node(e)          {create new node instance storing reference to element e}
  newest.next = L.head      {set new node's next to reference the old head node}
  L.head = newest           {set variable head to reference the new node}
  L.size = L.size + 1       {increment the node count}
```

4. set the list's head to point to the new node



2. Set its element to the new element

1. Create a new node

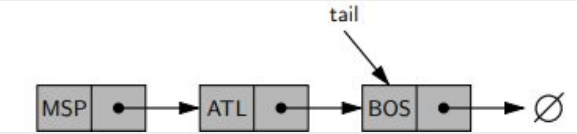
3. set its next link to refer to the current head

SINGLY LINKED LIST: Insertion

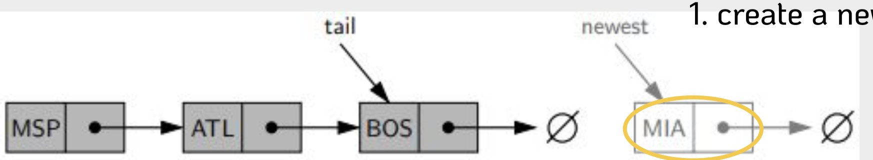
Singly Linked List: Insertion
- At the head
- At the tail

Inserting an Element at the Tail of a Singly Linked List

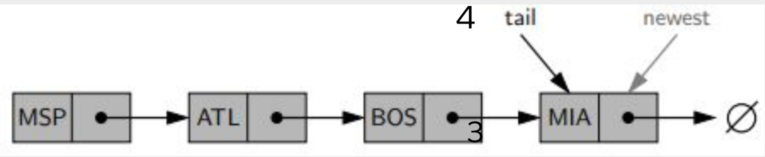
Algorithm add_last(L, e):
- newest = Node(e) {create new node instance storing reference to element e}
- newest.next = None {set new node's next to reference the None object}
- L.tail.next = newest {make old tail node point to new node}
- L.tail = newest {set variable tail to reference the new node}
- L.size = L.size + 1 {increment the node count}



1. create a new node



2. assign its next reference to None



3. set the next reference of the tail to point to this new node

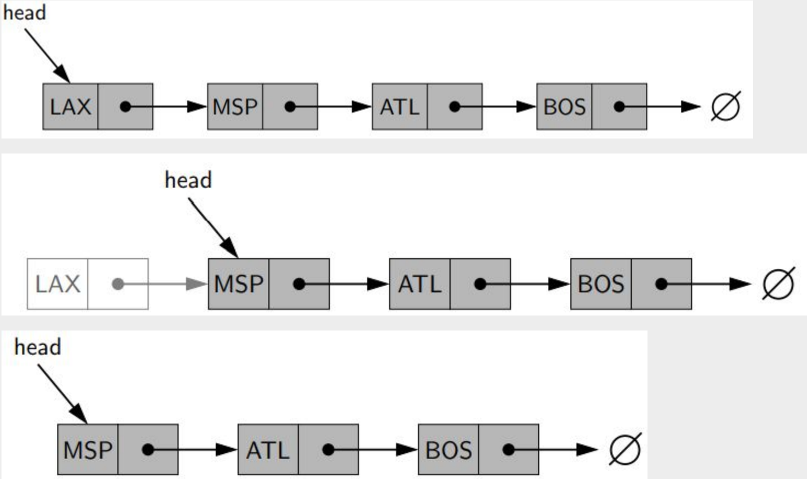
4. update the tail reference itself to this new node

SINGLY LINKED LIST: Removal

Algorithm remove_first(L):
  if L.head is None **then**
    Indicate an error: the list is empty.
  L.head = L.head.next   {make head point to next node (or None)}
  L.size = L.size − 1   {decrement the node count}

Singly Linked List: Removal

Removing an element from the Head of a Singly Linked List



Cf. Cost of removal from the tail is expensive
-> use doubly linked list instead

SINGLY LINKED LIST: Implementation(Stack)

Implementing a Stack with a Singly Linked List

- orient the top of the stack at the head of list.

```
LS = LinkedStack()
LS.push(3)
LS.push(2)
LS.push(1)
print("top:",LS.top())
LS.pop()
LS.pop()
LS.pop()
```

```
push:3
push:2
push:1
top: 1
popped:1
popped:2
popped:3
```

```
class Empty(Exception):
    pass

class LinkedStack:
    """LIFO Stack implementation using a singly linked list for storage."""

    class _Node:
        """Lightewight, nonpublic class for storing a singly linked node."""
        __slots__ = "_element","_next"

        def __init__(self, element, next):# initialize node' s fields
            self._element = element# reference to user' s element
            self._next = next# reference to next node

    def __init__(self):
        """Create an empty stack."""
        self._head = None# reference to the head node
        self._size = 0# number of stack elements

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

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

    def push(self, e):
        print("push:{}".format(e))
        """Add element e to the top of the stack"""
        self._head = self._Node(e, self._head)#create and link a new node
        self._size += 1

    def top(self):
        """Return (but do not remove) the element at the top of the stack.

        Raise Empty exception if the stack is empty."""
        if self.is_empty():
            raise Empty("Stack is Empty")#error message
        return self._head._element

    def pop(self):
        """Remove and return the element from the top of the stack (i.e., LIFO).
        Raise Empty exception if the stack is empty."""
        if self.is_empty():
            raise Empty("Stack is empty")
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        print("popped:{}".format(answer))
        return answer
```

SINGLY LINKED LIST: Implementation(Queue)

Implementing a Queue with a Singly Linked List

- Align the front of the queue with the head of the list, and the back of the queue with the tail of the list

```
LQ = LinkedQueue()
LQ.enqueue(1)
LQ.enqueue(2)
LQ.enqueue(3)
print("first:",LQ.first())
LQ.dequeue()
LQ.dequeue()
LQ.dequeue()
```

```
enqueue:1
enqueue:2
enqueue:3
first: 1
dequeued:1
dequeued:2
dequeued:3
```

LinkedStack vs. LinkedQueue

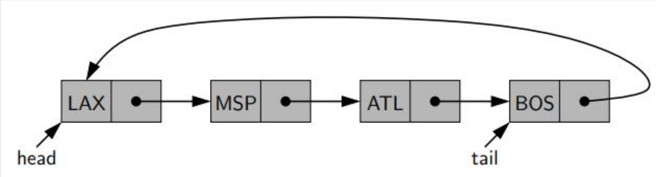
In terms of performance, the LinkedQueue is similar to the LinkedStack in that all operations run in worst-case constant time, and the space usage is linear in the current number of elements

```
class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage"""
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""

        __slots__ = "_element","_next"

        def __init__(self, element, next):# initialize node’ s fields
            self._element = element# reference to user’ s element
            self._next = next# reference to next node

    def __init__(self):
        """Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 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._head._element

    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._head._element
        self._head = self._head._next
        self._size -= 1
        if self.is_empty():
            self._tail = None
        print("dequeued:{}".format(answer))
        return answer

    def enqueue(self, e):
        """Add an element to the back of the queue."""
        print("enqueue:{}".format(e))
        newest = self._Node(e,None)
        if self.is_empty():
            self._head = newest
        else:
            self._tail._next = newest
        self._tail = newest
        self._size +=1
```

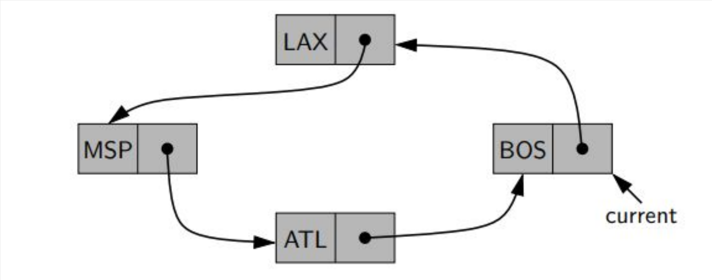
Circularly Linked Lists: Concept

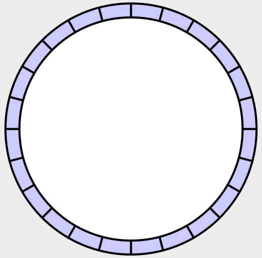
Circularly Linked List

- the tail of the list uses its next reference to point back to the head of the list



- do not have any particular notion of a beginning and end
- we must maintain a reference to a particular node, using the identifier ***current*** in order to make use of the list.
- How to traverse: current = current.next





Circularly Linked Lists: Implementation

Implementation of Circularly Linked List

- rely on the intuition , in which the queue has a head and a tail, but with the next reference of the tail linked to the head.
-
- The only two instance variables:
    - _tail: reference to the tail node (or None when empty)
    - _size: the current number of elements in the queue
- How to reach front:
    - self._tail._next

```
class CircularQueue:
    """queue implementation using circularly linked list for storage"""
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""

        __slots__ = "_element","_next"

        def __init__(self, element, next):# initialize node’ s fields
            self._element = element# reference to user’ s element
            self._next = next# reference to next node

    def __init__(self):
        """Create an empty queue."""
        self._tail = None# will represent tail of queue
        self._size = 0# number of queue elements

    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):#get the element next of tail
    """Return (but do not remove) the element at the front of the queue.
    Raise Empty exception if the queue is empty."""
    if self.is_empty():
        raise Empty('Queue is empty')
    head = self._tail._next
    return head._element

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")

    oldhead = self._tail._next
    if self._size == 1:
        self._tail = None#empty the queue
    else:
        self._tail._next = oldhead._next
    self._size -= 1
    return oldhead._element

def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e,None)
    if self.is_empty():#if queue is empty
        newest._next = newest#initialize circularly
    else:
        newest._next = self._tail._next#connect to head
        self._tail._next = newest#old tail points to new node
    self._tail = newest#new node becomes the tail
    self._size +=1

def rotate(self):
    print("rotate")
    """Rotate front element to the back of the queue"""
    if self._size >0:
        self._tail = self._tail._next

def traverse(self):
    print("display circularly linked queue")
    cur = self._tail._next#from head
    for i in range(self._size):
        print(cur._element, end = " ")
        cur = cur._next
    print("₩n")
```

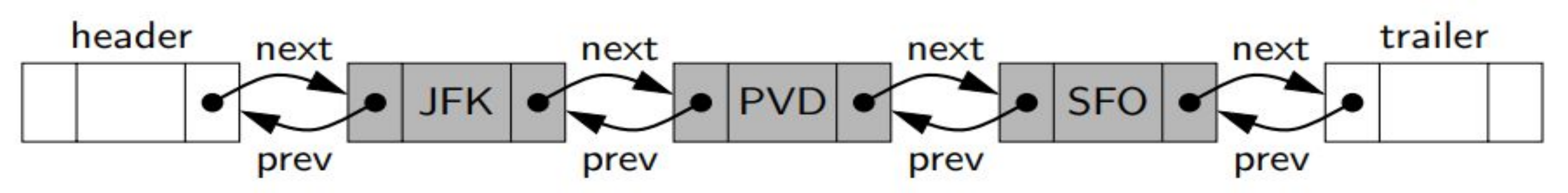
Circularly Linked Lists: Example of running

Running Circularly Linked List

```python
CQ = CircularQueue()
for j in range(10):
    CQ.enqueue(j+1)
CQ.traverse()
print("dequeued:",CQ.dequeue())
CQ.traverse()
print("dequeued:",CQ.dequeue())
CQ.traverse()
print("first:",CQ.first())
CQ.rotate()
CQ.traverse()
```

```
-----display circularly linked queue----
1 2 3 4 5 6 7 8 9 10

dequeued: 1

-----display circularly linked queue----
2 3 4 5 6 7 8 9 10

dequeued: 2

-----display circularly linked queue----
3 4 5 6 7 8 9 10

first: 3

====rotate=====

-----display circularly linked queue----
4 5 6 7 8 9 10 3
```

Doubly Linked List: Concept

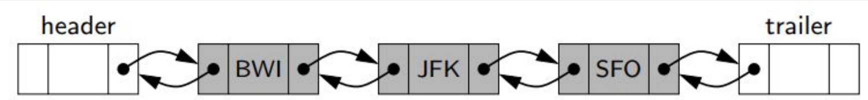


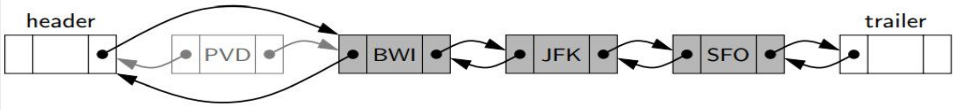
Doubly Linked List

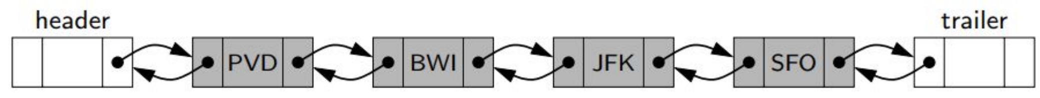
- add two special nodes(**sentinels**; nodes that do not store elements of the primary sequence) at both ends of the list:
    - header node at the beginning of the list
    - a trailer node at the end of the list
    - -> the header and trailer nodes never change—**<u>only the nodes between them change.</u>**
    - -> sentinels greatly simplify operations

Doubly Linked List: Insertion

Insertion of Doubly Linked List <u>at front</u>



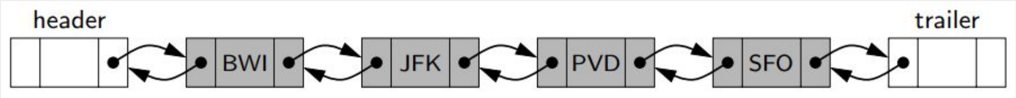


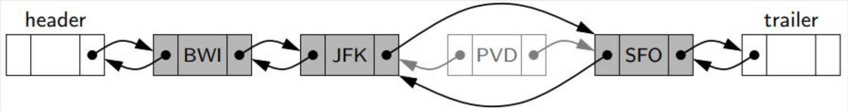


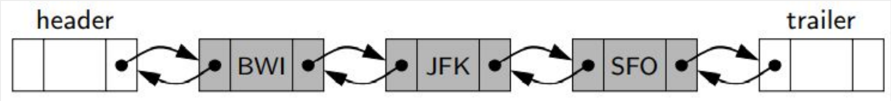
All operations are made between the header and trailer

Doubly Linked List: Insertion

Insertion of Doubly Linked List <u>in the middle</u>



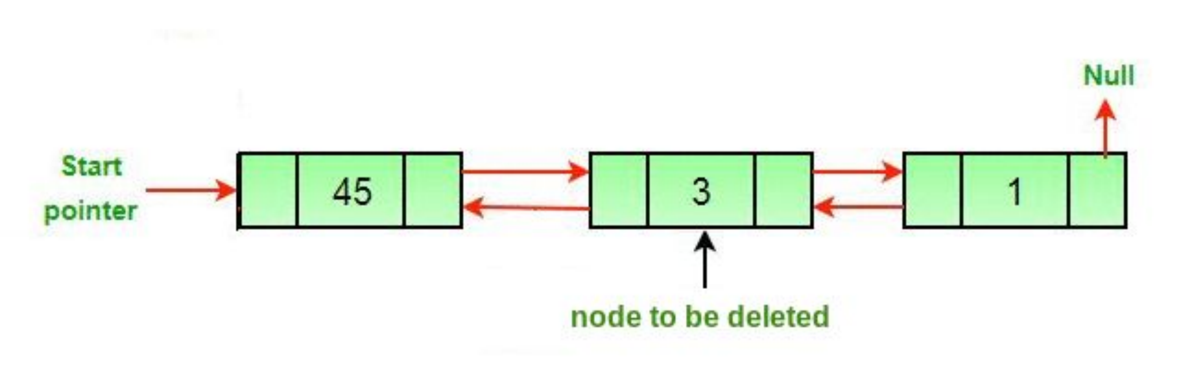




All operations are made between the header and trailer

Doubly Linked List: Deletion

Deletion of designated node in Doubly Linked List



All operations are made between the header and trailer

The neighbors of the node to be deleted are linked directly to each other, thereby bypassing the deleted node from the list.

Doubly Linked Lists: Implementation

Implementation of Doubly Linked List (Code)

- _insert_between: creates a new node, with that node's fields initialized to link to the specified neighboring nodes. Then the fields of the neighboring nodes are updated to include the newest node in the list.
- _delete_node: The neighbors of the node to be deleted are linked directly to each other, thereby bypassing the deleted node from the list.

```
class _DoublyLinkedBase:
    """ A base class providing a doubly linked list representation"""
    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        __slots__ = "element", "_prev", "_next"#streamline memory

        def __init__(self, element, prev, next):
            self._element = element
            self._prev = prev
            self._next = next

    def __init__(self):
        """Create an empty list."""
        self._header = self._Node(None, None, None)#element, prev, next
        self._trailer = self._Node(None, None, None)
        self._header._next = self._trailer#trailer is after header
        self._traler._prev = self._header#header is before trailer
        self._size = 0#number of elements

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

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

```
def _insert_between(self, e, predecessor, successor):
    """Add element e between two exiting nodes and return new node."""
    newest = self._Node(e,predecessor,successor)#link to neighbors
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest

def _delete_node(self, node):
    """Delete nonsentinel node from the list and return its element."""
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element
    node._prev = node._next = node._element = None#deprecate node
    return element#return element of the deleted node
```

Doubly Linked Lists: Implementation

Implementation of Doubly Linked List (Run)

```
DList = _DoublyLinkedBase()
first_elem = DList._insert_between(1,DList.header(),DList.trailer())
second_elem = DList._insert_between(2,first_elem,DList.trailer())
third_elem = DList._insert_between(4,second_elem,DList.trailer())
DList.traverse()
deleted = DList._delete_node(second_elem)
print("deleted:",deleted)
DList.traverse()
```

→

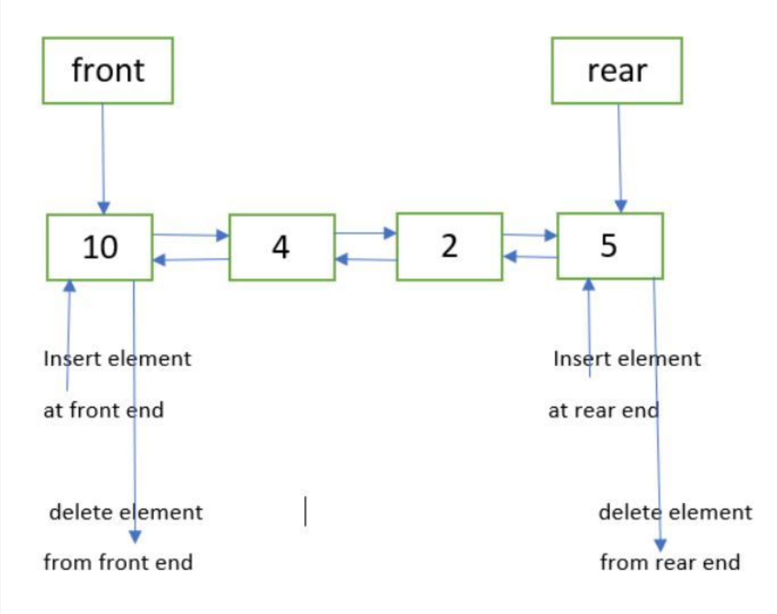
```
header -> | 1 || 2 || 4 |-> trailer
deleted: 2
header -> | 1 || 4 |-> trailer
```

```
def header(self):
    return self._header

def trailer(self):
    return self._trailer

def traverse(self):
    h = self._header
    print("header ->", end = " ")
    cur = h._next
    for i in range(self._size):
        print("|",cur._element,"|", end = "")
        try:
            cur = cur._next
        except:
            pass
    print("-> trailer")
```

Deque with a Doubly Linked List

Implementing a Deque with a Doubly Linked List(Concept)

- It is the node just after the header that stores the first element (assuming the deque is nonempty). Similarly, the node just before the trailer stores the last element of the deque.



Linked Deque: Implementation

Implementing a Deque with a Doubly Linked List(Code)

- Inherits Doubly Linked List class

```
class LinkedDeque(_DoublyLinkedBase):#use of inheritance
    """Double-ended queue implementation based on a doubly linked list."""
    def first(self):
        """Return (but do not remove) the element at the front of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._header._next._element #real item just after the header

    def last(self):
        """Return (but do not remove) the element at the back of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._trailer._prev._element

    def insert_first(self, e):
        """Add adn element to the front of the deque."""
        self._insert_between(e,self._header,self._header._next)

    def insert_last(self,e):
        """Add an element to the back of the deque."""
        self._insert_between(e,self._trailer._prev, self._trailer)

    def delete_first(self):
        """Remove and return the element from the front of the deque.
        Raise Empty exception if the deque is empty."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        return self._delete_node(self._header._next)

    def delete_last(self):
        """Remove and return the element from the back of the queue."""
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._delete_node(self._trailer._prev)
```

Linked Deque: Implementation

Implementing a Deque with a Doubly Linked List(Run)

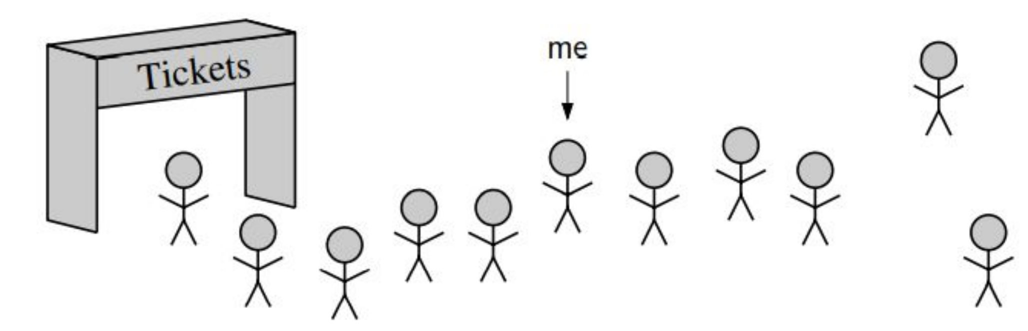
- Inherits Doubly Linked List class

```
def traverse(self):
    print("₩n-----Deque------")
    h = self._header
    c = h._next
    for i in range(self._size):
        print("|",c._element,"|", end = "")
        c = c._next
    print("₩n---------------₩n")
```

```
DQ = LinkedDeque()
DQ.insert_first("2")
DQ.insert_first("1")
DQ.insert_last("3")
DQ.traverse()
print(">>first element",DQ.first())
print(">>last element",DQ.last())
d = DQ.delete_last()
print(">>Delete last element:",d)
DQ.traverse()
d = DQ.delete_first()
print(">>Delete first element",d)
DQ.traverse()
```

```
-----Deque------
| 1 || 2 || 3 |
----------------

>>first element 1
>>last element 3
>>Delete last element: 3

-----Deque------
| 1 || 2 |
----------------

>>Delete first element 1

-----Deque------
| 2 |
----------------
```

The Positional List: Concept



The Positional List

- an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.
- instead of relying directly on nodes, an independent position abstraction to denote the location of an element within a list.
- positions serve as parameters to some methods and as return values from other methods.
- the only way in which a position becomes invalid is if an explicit command is issued to delete it.

The Positional List: ADT-(1)

The Positional List Abstract Data Types

- an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.
- instead of relying directly on nodes, an independent position abstraction to denote the location of an element within a list.
- the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- positions serve as parameters to some methods and as return values from other methods.

P: position, L: list

p.element(): Return the element stored at position p.

L.first(): Return the position of the first element of L, or None if L is empty.

L.last(): Return the position of the last element of L, or None if L is empty.

return the associated positions, not the elements.

L.before(p): Return the position of L immediately before position p, or None if p is the first position.

L.after(p): Return the position of L immediately after position p, or None if p is the last position.

L.is_empty(): Return True if list L does not contain any elements.

len(L): Return the number of elements in the list.

iter(L): Return a forward iterator for the *elements* of the list. See Sec-

The Positional List: ADT-(2)

The Positional List Abstract Data Types

Update functions: the functions that **modify** the list

P: position, L: list, e: element

L.add_first(e):	Insert a new element e at the front of L, returning the position of the new element.
L.add_last(e):	Insert a new element e at the back of L, returning the position of the new element.
L.add_before(p, e):	Insert a new element e just before position p in L, returning the position of the new element.
L.add_after(p, e):	Insert a new element e just after position p in L, returning the position of the new element.
L.replace(p, e):	Replace the element at position p with element e, returning the element formerly at position p.
L.delete(p):	Remove and return the element at position p in L, invalidating the position.

Implementation of a Positional List-(1)

implementation of a PositionalList class using a <u>doubly linked list</u>

A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.

```python
class PositionalList(_DoublyLinkedBase):

    """A sequential container of elements allowing positional access."""

    #----------nested Position class-----------
    class Position:
        """An abstraction representing **the location** of a single element."""
        def __init__(self, container, node):
            """Constructor should not be invoked by user."""
            self._container = container
            self._node = node

        def element(self):
            """Return the element stored at this Position."""
            return self._node._element

        def __eq__(self, other):#equal
            """Return True if other is a Position representing the same position."""
            return type(other) is type(self) and other._node is self._node

        def __ne__(self, other):#not equal
            """Return True if other does not represent the same location"""
            return not(self == other)

    #----------utility method-----------
    def _validate(self,p):
        """Return position's node, or raise appropriate error if invalid."""
        if not isinstance(p, self.Position):
            raise TypeError("p must be proper position type")
        if p._container is not self:
            raise ValueError("p does not belong to this container")
        if p._node._next is None:
            raise ValueError("p is no longer valid")
        return p._node

    def _make_position(self, node):
        """Return Position instance for given node(or None if sentinel)."""
        if node is self._header or node is self._trailer:
            return None
        else:
            return self.Position(self, node)
```

Returns position

```python
    #---------accessors------------------
    def first(self):
        """Return the first Position in the list(or None if list is empty.)"""
        return self._make_position(self._header._next)

    def last(self):
        """Return the last Position in the list(or None if list is empty.)"""
        return self._make_position(self._trailer._prev)

    def before(self, p):
        """Return the Position just before Position p(or None if p is first)."""
        node = self._validate(p)
        return self._make_position(node._prev)

    def after(self, p):
        """Return the Position just after Position p (or None if p is last)."""
        node = self._validate(p)
        return self._make_position(node._next)

    def __iter__(self):
        """Generate a forward iteration of the elements of the list."""
        cursor = self.first()
        while cursor is not None:
            yield cursor.element()#returns generator
            cursor = self.after(cursor)
```

Implementation of a Positional List-(2)

implementation of a PositionalList class using a doubly linked list

A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.

Functions allowed for users to access

```python
#----------mutators-----------------
#override inherited version to return Position rather than Node
def _insert_between(self, e, predecessor, successor):
    """Add element between existing nodes and return new Position."""
    node = super()._insert_between(e, predecessor, successor)
    return self._make_position(node)

def add_first(self,e):
    """Insert element e at the front of the list, and return new Position."""
    return self._insert_between(e,self._header, self._header._next)

def add_last(self, e):
    """Insert element e at the back of the list, and return new position"""
    return self._insert_between(e,self._trailer._prev, self._trailer)

def add_before(self,p,e):
    """Insert element e into list before position p and return new position"""
    original = self._validate(p)
    return self._insert_between(e, original._prev, original)

def add_after(self, p,e):
    """Insert element e into list after position p and return new position."""
    original = self._validate(p)
    return self._insert_between(e, original, original._next)

def delete(self, p):
    """Remove and return the element at position p."""
    original = self._validate(p)
    return self._delete_node(original)#inherited method returns element

def replace(self,p,e):
    """Replace the element at position p with e.
    Return the element formerly at Position p."""
    original = self._validate(p)
    old_value = original._element#temporarily store old element
    original._element = e
    return old_value#return the old element value

def traverse(self):
    print("₩n======================List===========================")
    cur = self.first()
    while cur is not None:
        print(" |{}|".format(cur.element()), end = "")
        cur = self.after(cur)
    print("₩n=====================================================₩n")
```

```python
PL = PositionalList()
#add elements to list
PL.add_first(1)
for i in range(9):
    PL.add_last(i+2)
PL.traverse()
print("first position in list:",PL.first().element())
h = PL.add_first("h")
o = PL.add_first("o")
p = PL.add_first("p")
PL.traverse()
print("first position in list:",PL.first().element())

d = PL.delete(o)
print("Delete:",d)
PL.traverse()

PL.replace(h,"i")
print("replace h with i ")
PL.traverse()
```

```
======================List===========================
 |1| |2| |3| |4| |5| |6| |7| |8| |9| |10|
=====================================================

first position in list: 1

======================List===========================
 |p| |o| |h| |1| |2| |3| |4| |5| |6| |7| |8| |9| |10|
=====================================================

first position in list: p
Delete: o

======================List===========================
 |p| |h| |1| |2| |3| |4| |5| |6| |7| |8| |9| |10|
=====================================================

replace h with i

======================List===========================
 |p| |i| |1| |2| |3| |4| |5| |6| |7| |8| |9| |10|
=====================================================
```