UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Announcements

- Exam Duration – 50 minutes
  - Total points: 50
  - Total 4 sections → 10 MCQs in each
  - All questions carry 1 point except Two, that carry 2 and 2.5 points
- No Labs this week
- Office hours for Thursday → 11:00 am to 12:00pm
- Assignment due date extended
- Career Fair briefing on October 04

Module 1 – Python Basics

- Mathematical Operators and precedence
  - +, - , *, /, //, %

Operator	Description	Explanation
( )	Items within parentheses are evaluated first.	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
exponent **	** used for exponent is next.	In x**y * 3, x to the power of y is computed first, with the results then multiplied by 3.
unary -	- used for negation (unary minus) is next.	In 2 * -x, the -x is computed first, with the result then multiplied by 2.
*** / %**	Next to be evaluated are *, /, and %, having equal precedence.	
+ -	Finally come + and - with equal precedence.	In y = 3 + 2 * x, the 2 * x is evaluated first, with the result then added to 3, because * has higher precedence than +. Spacing doesn't matter: y = 3+2 * x would still evaluate 2 * x first.

CSE-2050 – Data Structures and Object-Oriented Design
Review

Module 1 – Python Basics

- Mathematical Operators and precedence
  - +, - , *, /, //, %

Operator	Description	Explanation
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right. Note: The ** operator is evaluated from right-to-left.	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

Week 5 – 09/26 – 09/30 – Review

Module 1 – Python Basics

- Functions
  - Definition
  - Calls
- A ***parameter*** is a function input specified in a function definition.
  - Example: **def double(x)**
- An ***argument*** is a value provided to a function's parameter during a function call.
  - Example: double(3)
  - 3 is the argument
- Argument can be an **expression**, however parameter can NOT**.**
  - Example: double(3+5)

```
>>> def double(x):
...     return 2*x
...
>>> double(3)
6
>>> double(3.2)
6.4
>>> double("pika")
'pikapika'
```


Module 1 – Python Basics

- Variables and Scope

```
daily_cals = 2300 # Daily calories
soda_cals = 200

def drink_soda(cals_left):
        return cals_left - soda_cals

daily_cals = drink_soda(daily_cals)
```

Built-in namespace

int()	<funct>
…	…
input()	<funct>

Global namespace

daily_cals	2100
soda_cals	200
drink_soda	<funct>

Local namespace

cals_left	2300

Module 1 – Python Basics

- Branch statements and loops

`if` statement

```
if <condition>:
        ......
        ......
```

`if-else` statement

```
if <condition>
        ......
else:
        ......
```

`if-elif-else` statement

```
if <condition>:
        ......
elif <condition> :
        ......
        ......
elif <condition> :
        ......
....
else:
        ......
```

- Boolean Operators

  ==, >, <, >=, <=, !=

- Logical Operators

  and, or, not

a	b	a AND b
False	False	False
False	True	False
True	False	False
True	True	True

a	b	a OR b
False	False	False
False	True	True
True	False	True
True	True	True

a	NOT a
False	True
True	False

Module 2 – OOP and Testing

Class: template to create objects

Method: functions defined inside a Class

Attributes: Data/features of class

Instantiation: Creation of an object/instance of a class

Inheritance: Inheriting or extending the existing features of another class
- inherited class → Parent / Base / Super Class
- inheriting class → Child / Derived / Sub Class

Composition: Creating an instance of a class within another class
- Composite class → The class which contains an object of another class.
- Component class → The class being referenced in another class

Polymorphism: Use of single type entity to represent different types in different scenarios

Module 2 – OOP and Testing

- Testing
  - assert statements
    - Raise exception only if the Boolean expression returns False
    - Does not execute the remaining tests once any test in between fails
    - Does not specify where the error is occurred
  - Unittest
    - Runs all tests even if any test fails
    - Specify where the failure occurred
  - Test Driven Development
    - TDD is based on **Red-Greed-Refactor** phases
    - Red: The test fails
    - Green: You get the tests to pass by writing the correct code
    - Refactor: You clean up the code, removing clutter/duplication

Module 3 – Running Time Analysis

- Performance analysis via measuring execution time
  - Does not give us accurate performance because of different factors
- Asymptotic analysis

Module 3 – Running Time Analysis

- Time complexity in list
  - Read
  - Pop
  - Insert

Operation Name	Average Code	Cost
Index access	`L[i]`	1
Index assignment	`L[i] = newvalue`	1
Append	`L.append(newitem)`	1
Pop (from end of list)	`L.pop()`	1
Pop (from index i)	`L.pop(i)`	n – i
Insert at index i	`insert(i, newitem)`	n – i
Delete an item (at index i)	`del(item)`	n – i
Membership testing	`item in L`	n
Slice	`L[a:b]`	b – a
Concatenate two lists	`L1 + L2`	$n_1 + n_2$
Sort	`L.sort()`	$nlog_2n$

Module 3 – Running Time Analysis

- Big O notation

The formal mathematical definition which allows us to ignore lower order terms and constants is called **Big-O** notation

Asymptotic Analysis: $5n^2 + 3n + 2$

Big O Notation: $O(n^2)$

Module 3 – Running Time Analysis

```
1  #Practice question 1
2  def func(L):
3      x = 0
4      for i in L:
5          for j in L:
6              x += i * j
7      return x
```

```
1  # Practice question 2
2  a = 0
3  b = 0
4  for i in range(N):
5      a = a + random()
6
7  for i in range(M):
8      b= b + random()
```

```
1  # Practice question 4
2  i = 1
3  while i <= n:
4      print(i)
5      i *= 2
```

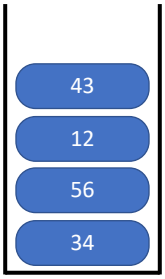
- Constant Functions → O(1)
- Logarithmic Functions → O(log n)
- Linear Functions → O(n)
- n Log n → O(nlogn)
- Quadratic Functions → $O(n^2)$

CSE-2050 – Data Structures and Object-Oriented Design

Review

Module 4 – Linear Data Structures

- Stacks - LIFO
  - Push
  - Pop
  - Peek



stack

- Time complexity for all operations using list implementation: O(1)

Week 5 – 09/26 – 09/30 – Review

Hasan Baig

Module 4 – Linear Data Structures

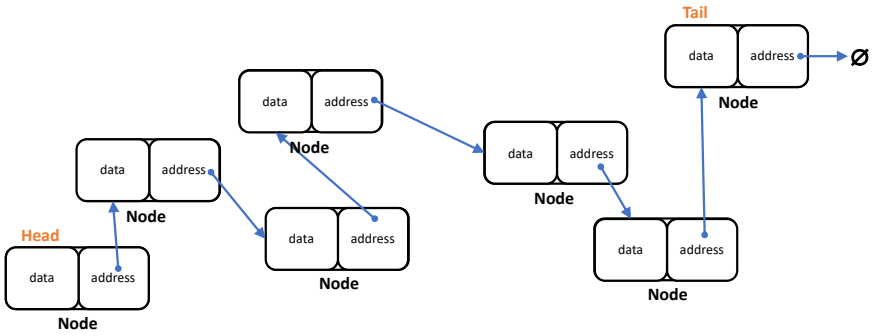
- Queues - FIFO
  - Enqueue
  - Dequeue
  - First/Peek

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

- Time Complexity: O(1) for all except dequeue → O(n)

Module 4 – Linear Data Structures

- Linked-Lists



- Data is arranged in distributed nodes
- Each node contains the element (data) + the address of next node

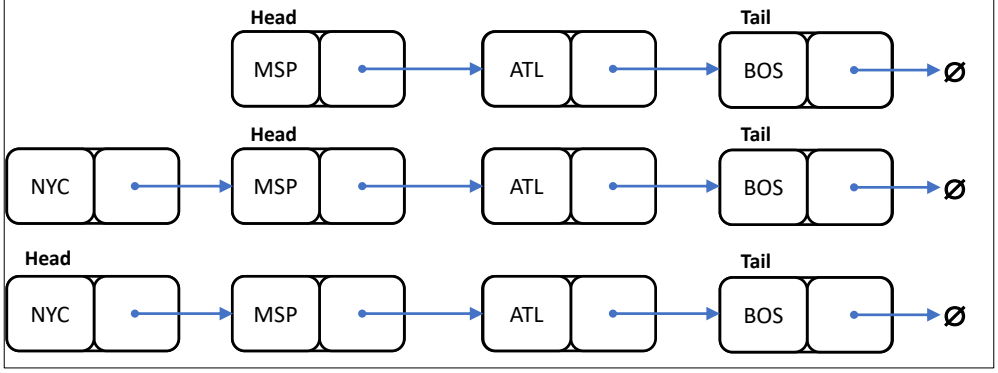
- Minimally, instance of a linked list must keep a reference of **head** node

Module 4 – Linear Data Structures

- Linked-Lists

Inserting element at:

The HEAD of linked list



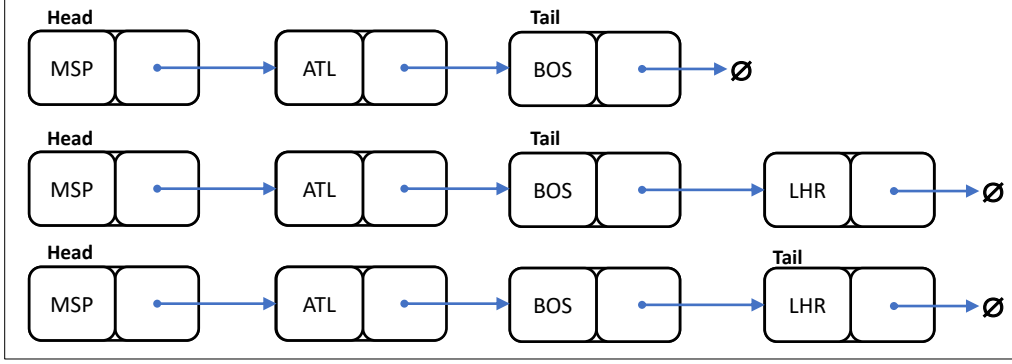
Hasan Baig

Module 4 – Linear Data Structures

- Linked-Lists

Inserting element at:

The TAIL of linked list



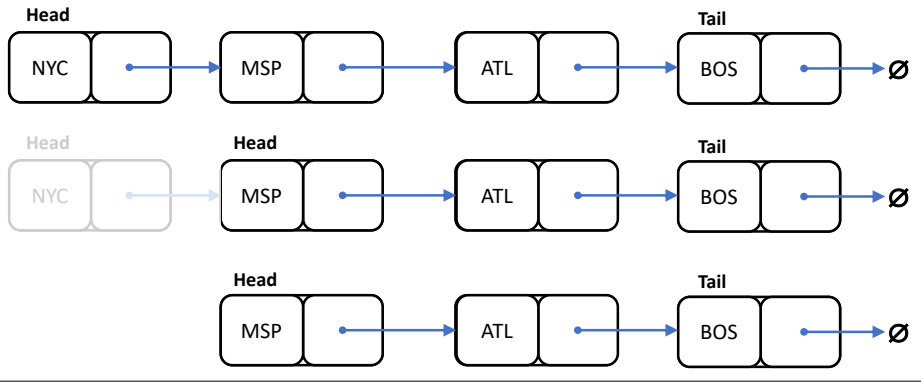
Hasan Baig

Module 4 – Linear Data Structures

- Linked-Lists

Deleting element from:

The HEAD of linked list

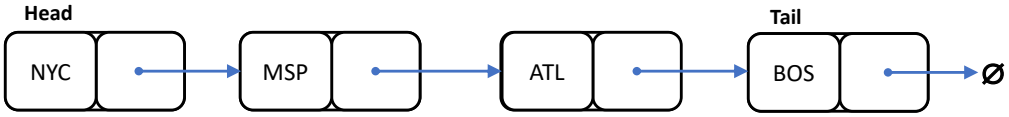


Module 4 – Linear Data Structures

- Linked-Lists

Deleting element from:

The TAIL of linked list



- We cannot easily delete the last node of a singly linked list
  - Though, we can hold a reference to the Tail node
  - There is no way to determine the node preceding to the Tail node
- List traversal may help, but takes O(n) time