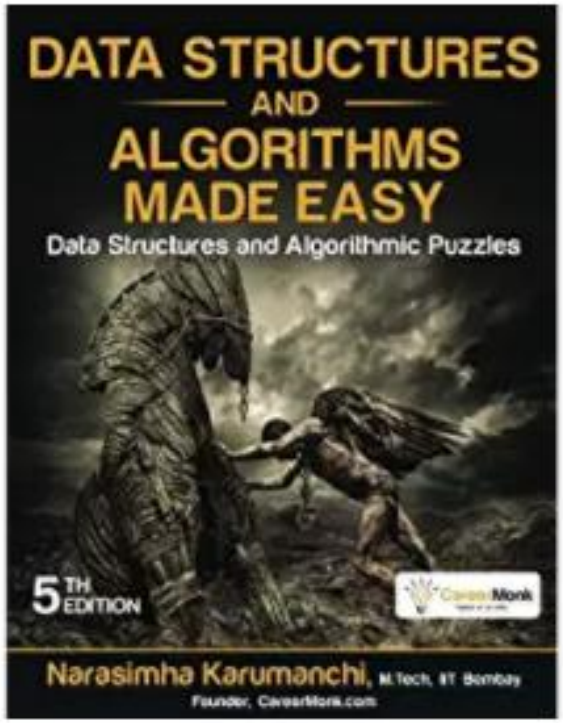
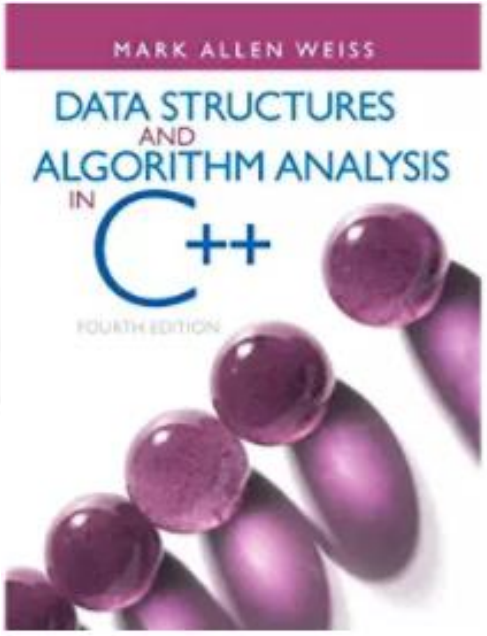
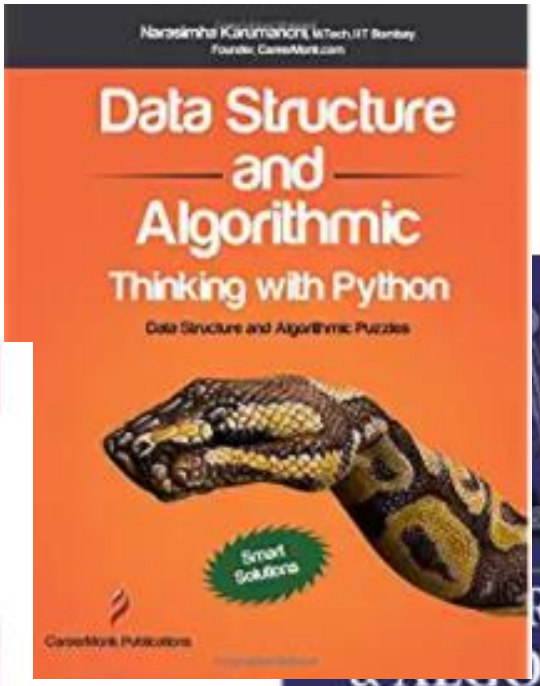
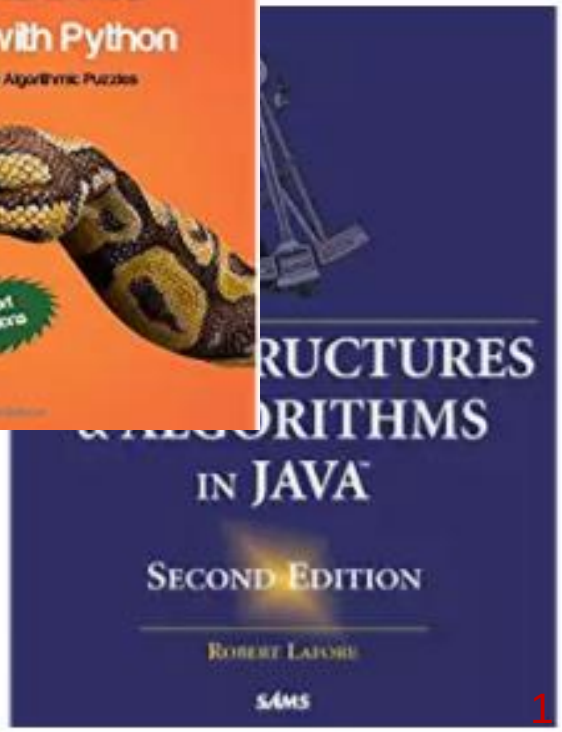
(Chapter 21) Data Structures in Python

Books for Data Structure Class

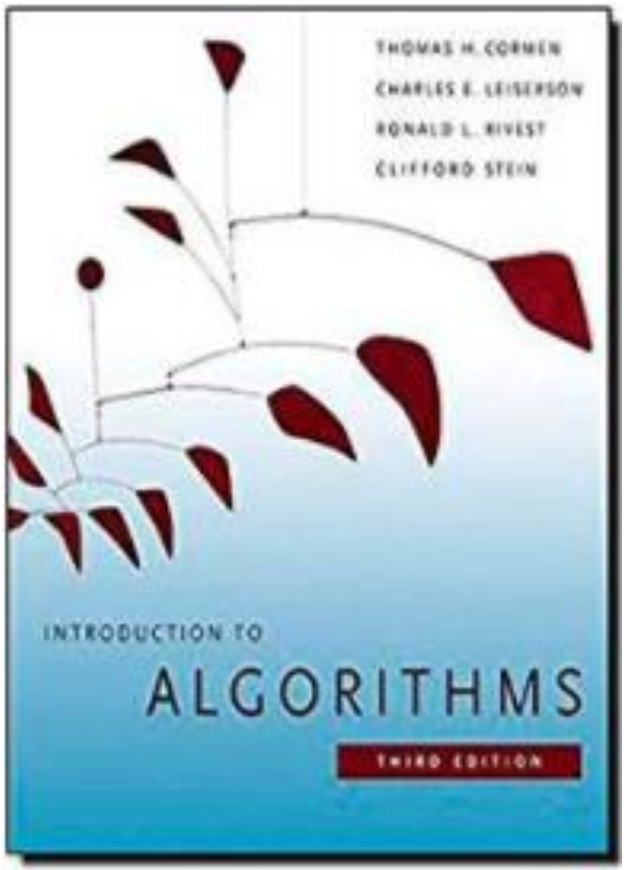


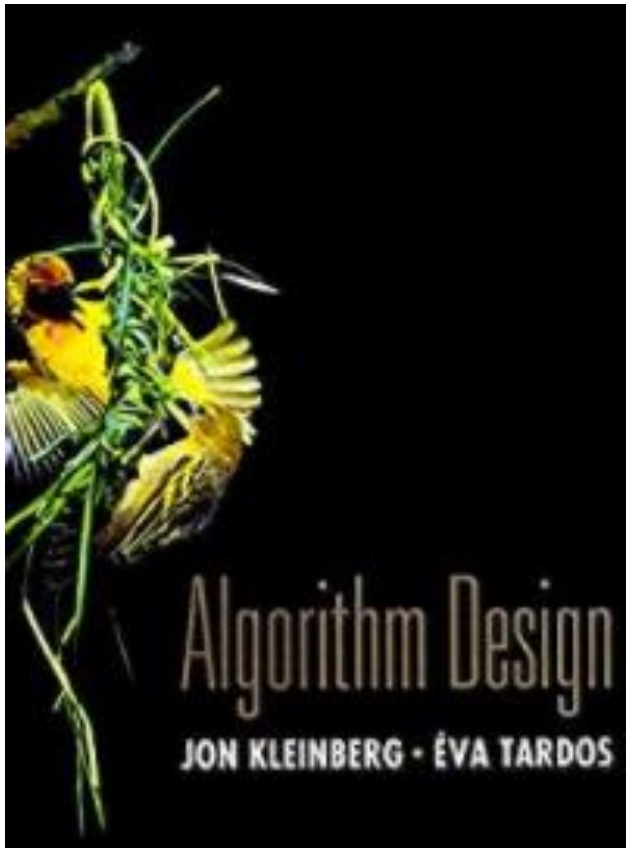






Books for Algorithm Class





Chapters of Data Structure Book

- Linear List (= 1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

Examples of Linear List

Students in COP3530: Jack, Jill, Abe, Henry, Mary, …, Judy

Exams in COP3530: exam1, exam2, exam3

Grades in COP3530: "Jack A+", "Jill B-", "Abe D", … "Judy F"

Days of Week: S, M, T, W, Th, F, Sa

Months: Jan, Feb, Mar, Apr, …, Nov, Dec

Functions of Linear List

- ***Suppose L = (a, b, c, d, e, f, g)***
- size() : Determine list size
  - *L.size() = 7*
- get(index) : Get element with given index
  - *get(0) = a* *get(2) = c* *get(4) = e* *get(-1) =* error *get(9) =* error
- indexOf(element) : Determine the index of an element
  - *indexOf(d) = 2* *indexOf(a) = 0* *indexOf(z) = -1*
- remove(index) : Remove and return element with given index.
  - *remove(2)* returns *c, L* becomes *(a,b,d,e,f,g),* indices of *d,e,f* and *g are decreased* by 1
  - *remove*(-1) → error *remove*(20) → error
- add(index, element) : Add an element so that the new element has a specified index
  - *add(0,h)* ➔ *L = (h,a,b,c,d,e,f,g) //* indices of *a,b,c,d,e,f,* and *g* are increased by *1*
  - *add(2,h)* ➔ *L = (a,b,h,c,d,e,f,g) //* indices of *c,d,e,f,* and *g* are increased by *1*
  - *add(10,h)* ➔ error *add(-6,h)* ➔ error

Python for Linear List?

- List (built-in data type in Python)

```
>>> L = [3, 7, 1]
>>> L.append(5)
>>> L
[3, 7, 1, 5]
```

- Array module in Python (not popular)

```
>>> from array import *
>>> A = array('i', [4, 3, 6])
>>> A
Array('i', [4,3,6])
>>> A.append(9)
>>>A
array('i', [4,3,6,9])
```

- Python에서 Vector는 List로 cover된다고 볼수 있음

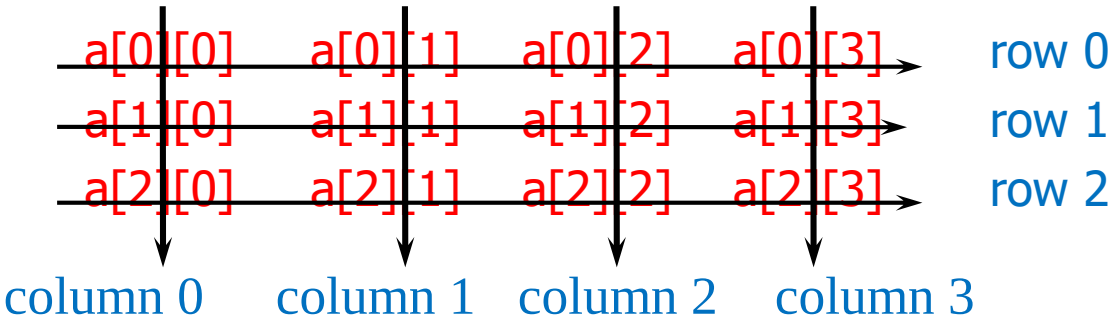
Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

2-D Array or Matrix

- 2D-Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]



- Matrix: Table of values
  - has as rows and columns like 2-D array
  - but numbering begins at 1 rather than 0

a	b	c	d	row 1
e	f	g	h	row 2
i	j	k	l	row 3

Matrix as Abstract Data Type

AbstractDataType *Matrix* {
 instances
  m1 = Matrix( )
  m2 = Matrix( )
 functions
  copy (Matrix m) : n = copy(m1)
  get (int i, int j) : return the value of the pair with this index
  Ex. m1.get(3, 4)
  set (int i, int j, newValue) : overwrite existing one (if any) with the same index
  Ex. m1.set(3, 4, "nnn")
  add(Matrix m) : Ex. m1.add(m2)
  multiply(Matrix m) : Ex. m1.multiply(m2)
}

Python for Array and Matrix?

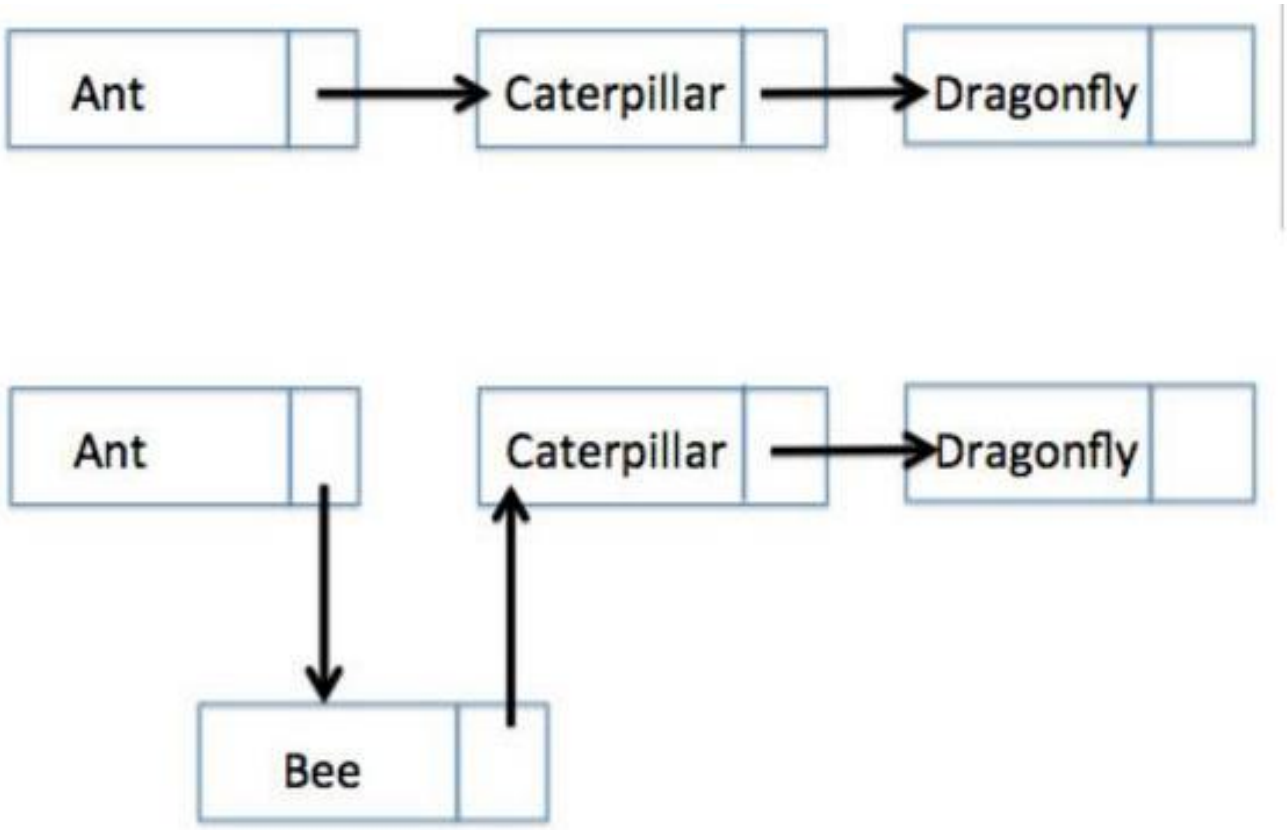
- List data type
- numpy module
  - np.array data type

Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

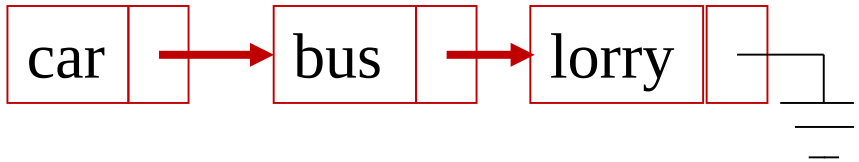
Concept of Linked Lists

- A data structure that uses pointers to point to the next item in the list
- Can be implemented using an array or a class



OO Implementation of Linked List

```
1    class Node:
2        def __init__(self, contents=None, next=None):
3            self.contents = contents
4            self.next  = next
5
6        def getContents(self):
7            return self.contents
8
9        def __str__(self):
10           return str(self.contents)
11
12   def print_list(node):
13       while node:
14           print(node.getContents())
15           node = node.next
16       print()
17
18   def testList():
19       node1 = Node("car")
20       node2 = Node("bus")
21       node3 = Node("lorry")
22       node1.next = node2
23       node2.next = node3
24       print_list(node1)
```

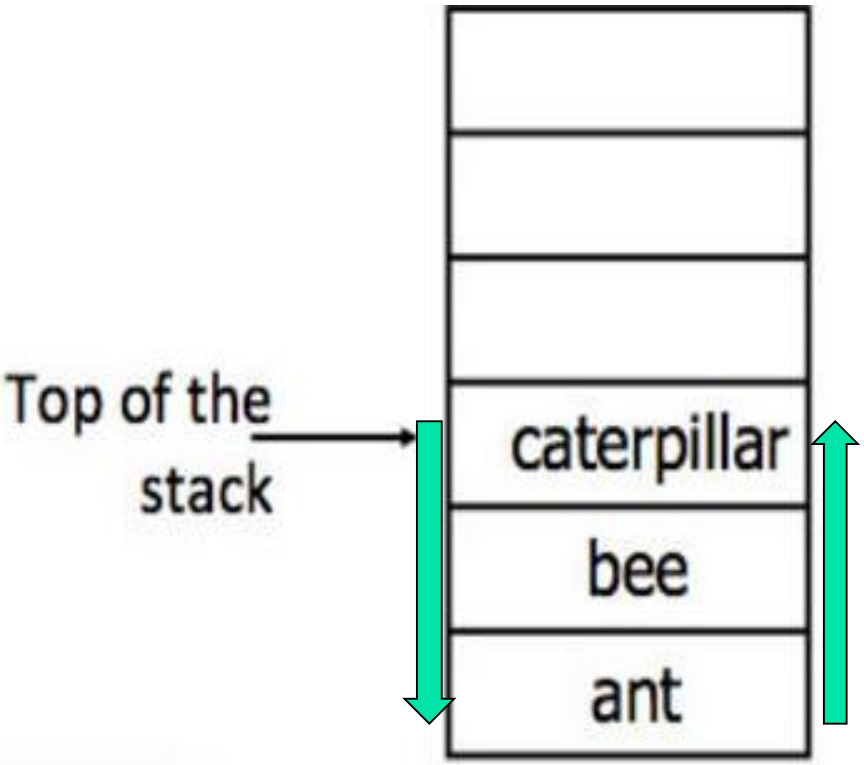


Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

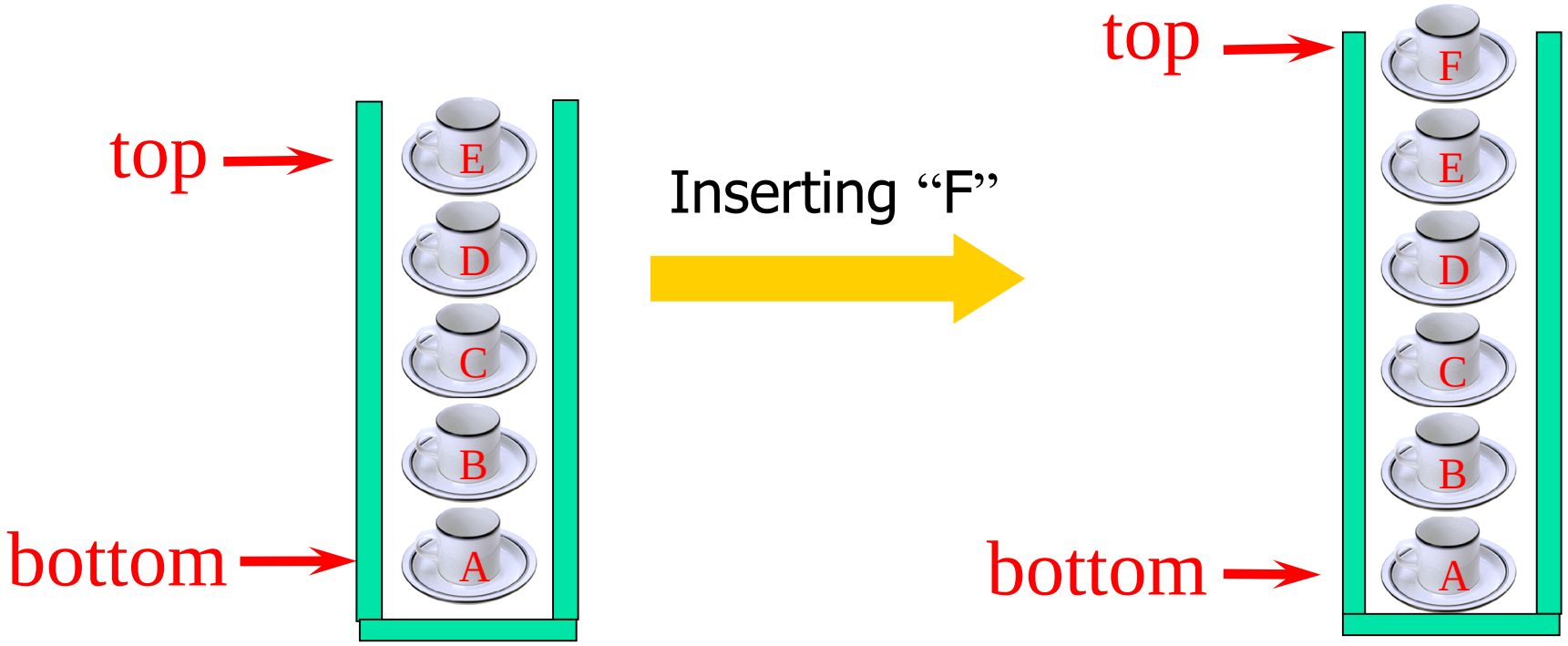
Concept of Stack

- A last-in, first-out (LIFO) structure where items are stored in the stack
- But if an item is taken from the stack, it is always the last one that was added
- Can be implemented using an array or a class



Stack as a Special Case of List

- A kind of Linear list
- One end is called "top" and the other end is called "bottom"
- Insertions and removals take place at the top
- A stack is a LIFO list (Last In First Out)
- Stack of Cups



Stack as Abstract Data Type

```
AbstactDataType Stack{
    instances
        linear list of elements;
        bottom;
        top;
    functions
        empty() : Return true if the stack is empty,
                  Return false otherwise;
          peek() :  Return the top element;
        push(x) :  Add element x at the top;
           pop() :  Remove the top element  and return it;
}
```

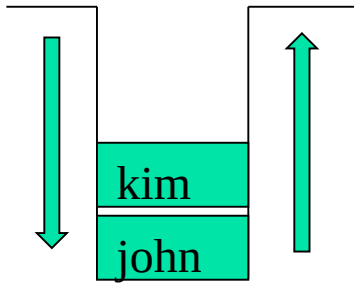
Python support for Stack?

- "queue" standard library
  - LifoQueue class

OO Implementation of Stack using Python List

```
>>> class Stack:
        # the stack class
        def __init__(self):
                self.items = []
        def push(self, item):
                self.items.append(item)
        def pop(self):
                return self.items.pop()
        def isEmpty(self):
            if self.items == []:
                    return True
            else:
                    return False
        def peek(self):
                return self.items[len(self.items)-1]
```

```
>>> myStack = Stack()
>>> myStack.push("john")
>>> myStack.push("kim")
>>> myStack.peek()
'kim'
>>> myStack.pop()
'kim'
>>> myStack.items
['john']
>>>
```



Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

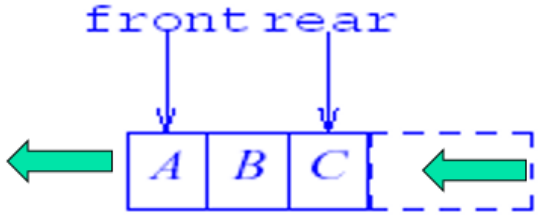
Concept of Queue

- A first in, first out (FIFO) structure where the first item to join the queues is the first to leave
- Can be implemented using a linked list or a class

A list implementation for a linear queue will use an append method to add to the queue and a delete method to remove from the queue.

Queue as a Special Case of List

- The index i for the front element is 0
  - front_ptr = location(front element)
  - rear_ptr = location(last element)

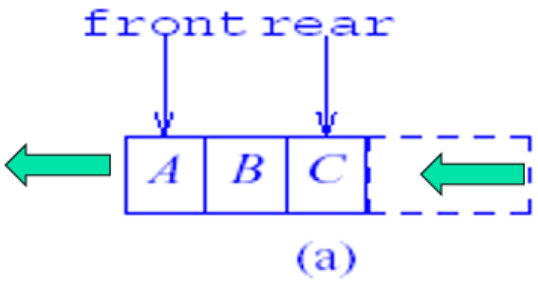


- Empty queue has the condition: rear_ptr < front_ptr

- Insert an element: The worst-case time from **O**(1) to **O**(queue.length)
  - Move rear_ptr to right by 1

- Delete an element: **O**(1) time
  - Move front_ptr to right by 1

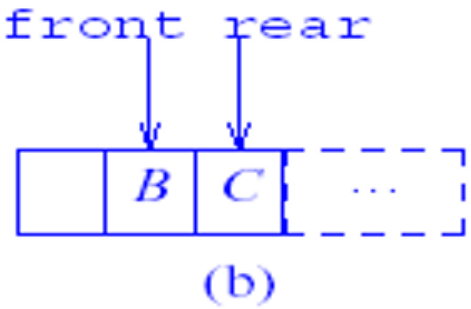
Operations of Queue



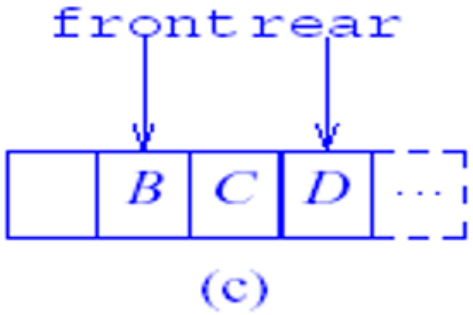
Insert “A”

Insert “B”

Insert “C”

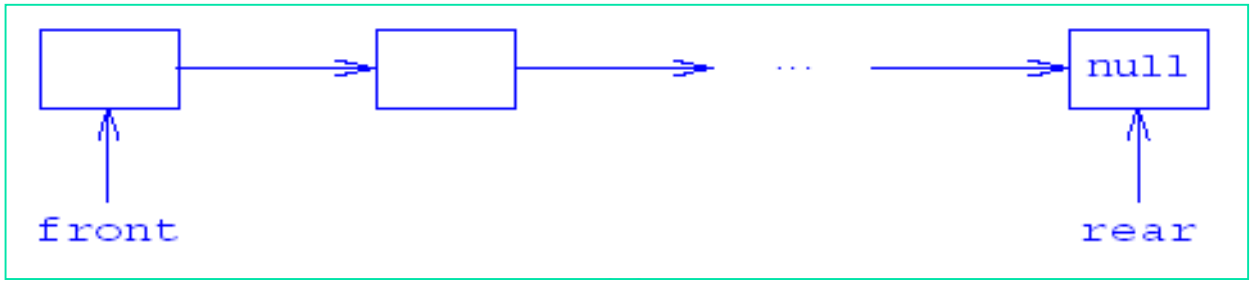


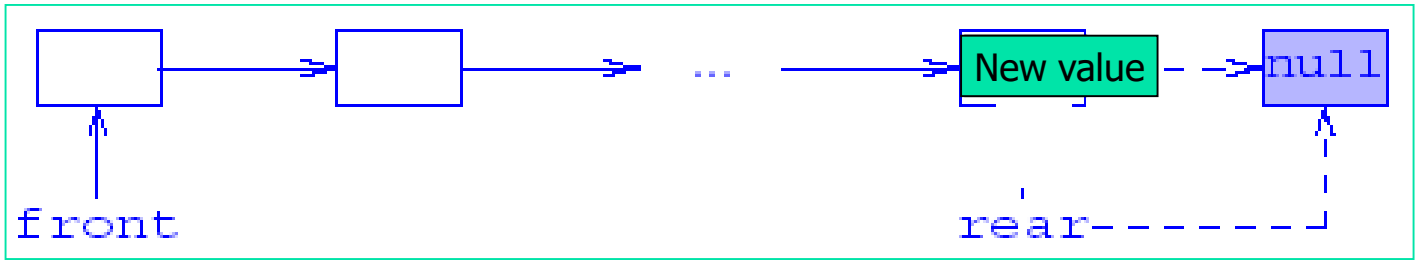
Deleting “A”



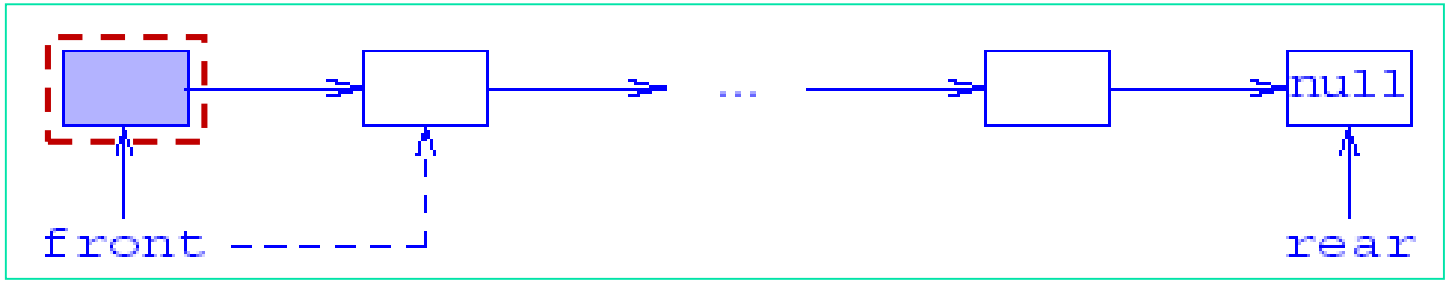
Inserting “D”

Queue in Linked-List Structure





Insert a new item



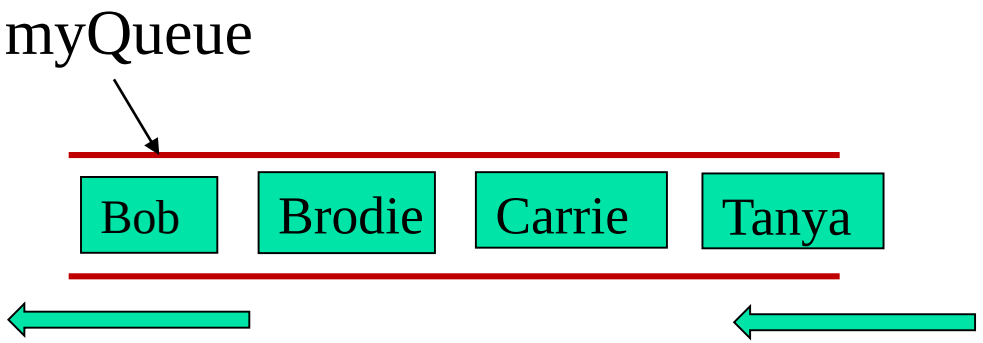
Delete an existing item

Python for Queue?

- "queue" standard library
  - Queue class, LifoQueue class, PriorityQueue class

```
# OOP Implementation of a queue using Python list
class Queue:
    def  __init__(self):
         self.items = [ ]
    def  add(self, item):
          self.items.append(item)
    def  delete(self):
          itemToDelete = self.items[0]
          del self.items[0]
          return itemToDelete
     def size(self):
          return len(self.items)
     def report(self):
         return self.items
```

```
myQueue=Queue()
myQueue.add("Bob")
myQueue.add("Brodie")
myQueue.add("Carrie")
myQueue.add("Tanya")
print(myQueue.size())
print(myQueue.report())
print(myQueue.delete())
print(myQueue.report())
```



“queue” Python Standard Library

List, Set, Dict같은것으로 Queue를 지원하는것은 불편하고, 세가지 대표 queue종류을 한번에 지원!!

- The queue module provides a safe implementation of FIFO structure
  - Queue class implemented all the required locking semantics

- There are 3 types of Queue, which differ in the order of the entities retrieved
  - FIFO queue ➔ Queue Class
  - LIFO queue (Works like a stack) ➔ LifoQueue Class
  - Priority queue ➔ PriorityQueue Class

```
import queue

a = queue.Queue(5)
b = queue.LifoQueue(5)
c = queue.PriorityQueue(5)

print("Successfully created 3 queues")
```

Result

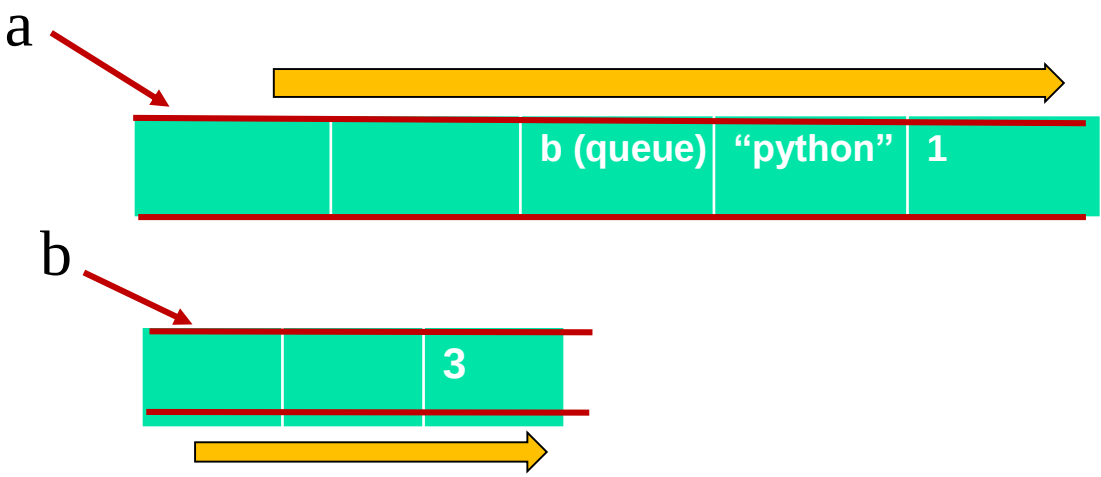
```
>>>
Successfully created 3 queues
```

"queue" Module: Queue Class for Queue [1/2]

- queue.**Queue**(x) : Construct a FIFO queue of size 'x'
- queue.**Queue**() : Construct a FIFO queue of infinite size
- queue.**put**(x) : Put item into the queue. Item can be anything
- queue.**get**(x) : Delete the item and return that item

```
import queue

a = queue.Queue(5)
b = queue.Queue(3)

a.put(1)
a.put("python")
a.put(b)

b.put(3)

print(a.get())
print(a.get())
print(a.get().get())
```

Return the queue 'b'



Result

```
>>>
1
python
3
```

“queue” Module: Queue Class for Queue [2/2]

- queue.**qsize**() : Return the number of items in the queue
- queue.**empty()** : Return True if the queue is empty, False otherwise
- queue.**full**() : Return True if the queue is full, False otherwise

```
import queue

a = queue.Queue(3)
b = queue.Queue()

a.put(1)
a.put(2)
a.put(3)
```

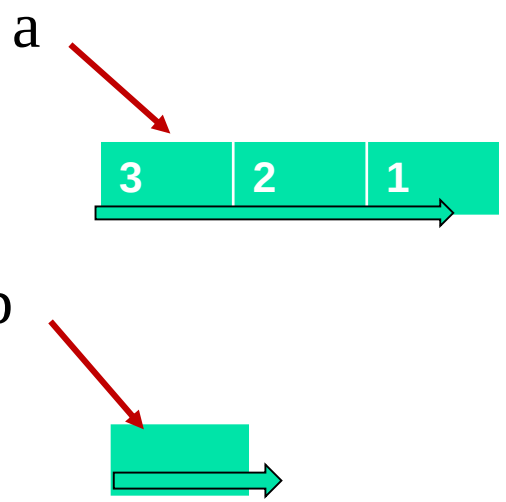
```
print("qsize : ")
print(a.qsize())
print(b.qsize())
print()
```

```
print("Empty?")
print(a.empty())
print(b.empty())
print()
```

```
print("Full?")
print(a.full())
print(b.full())
print()
```

Result

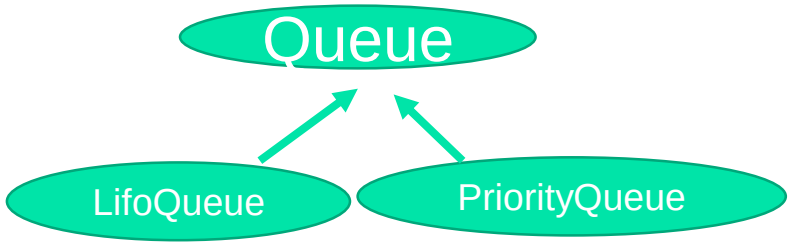
```
>>>
qsize :
3
0

Empty?
False
True

Full?
True
False
```

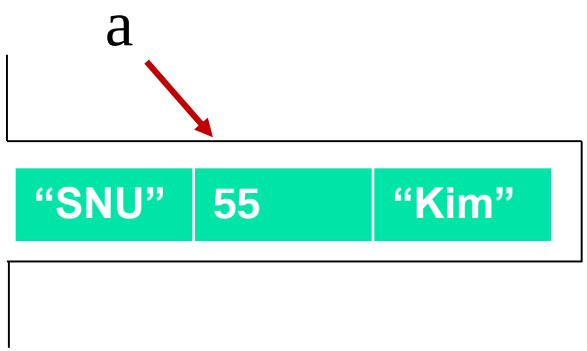


"queue" Module: LiFoQueue Class for Stack

- Subclass of Queue class
- put(x), get(x), qsize(), empty(), full() are all similar with that of Queue class

```
>>> import queue
>>> a = queue.LifoQueue(3)
>>> a.put("Kim")
>>> a.put(55)
>>> a.put("SNU")
>>> a.qsize()
3
>>> a.get()
'SNU'
>>> a.qsize()
2
```

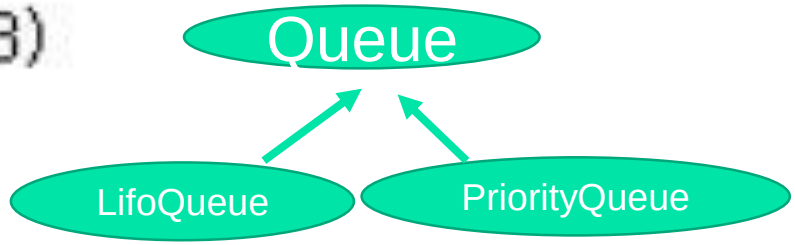


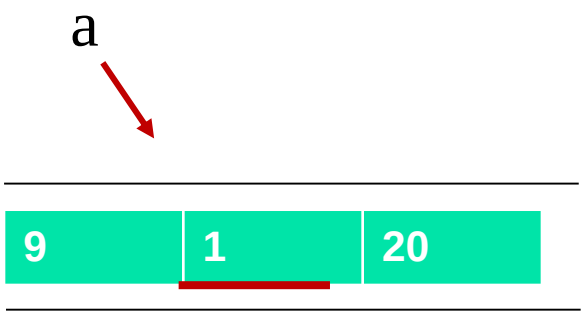


“queue” Module : PriorityQueue Class

- A subclass of Queue class, retrieves entries in priority order (lowest first)
- put(x), get(x), qsize(), empty(), full() are all similar with that of Queue class

```
>>> import queue
>>> a = queue.PriorityQueue(3)
>>> a.put(20)
>>> a.put(1)
>>> a.put(9)
>>> a.qsize()
3
>>> a.get()
1
>>> a.qsize()
2
```





Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linear List
- Stack
- Queue
- Dictionary and Hashing
  - Python dictionary data type
  - Python hashlib standard library
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph

Dictionary in Python [1/4]

Dictionaries in Python are implemented using hash tables. It is an array whose indexes are obtained using a hash function on the keys.

We declare an empty dictionary like this:

```
>>> D = {}
```

Then, we can add its elements:

```
>>> D['a'] = 1
>>> D['b'] = 2
>>> D['c'] = 3
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

Dictionary in Python [2/4]

It's a structure with (key, value) pair:

```
D[key] = value
```

The string used to "index" the hash table D is called the "key". To access the data stored in the table, we need to know the key:

```
>>> D['b']
2
```

Dictionary in Python [3/4]

How we loop through the hash table?

```
>>> for k in D.keys():
...     print D[k]
...
1
3
2
```

If we want to print the (key, value) pair:

```
>>> for k,v in D.items():
...     print k,':',v
...
a : 1
c : 3
b : 2
```

Dictionary in Python [4/4]

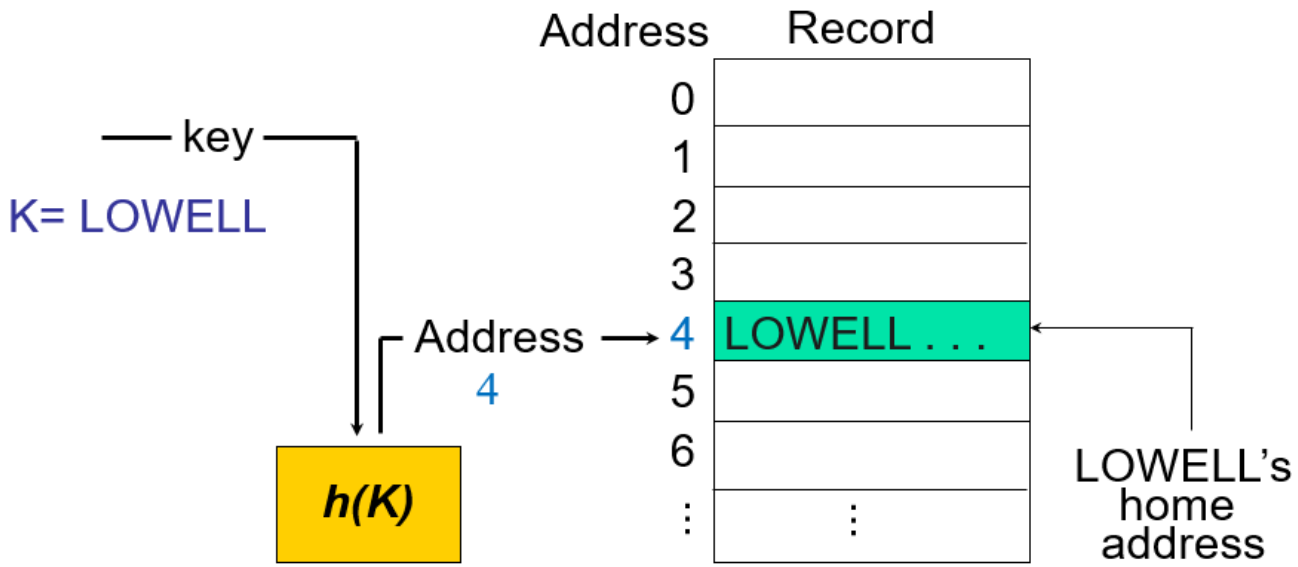
Create a dictionary from two arrays

Using two Arrays of equal length, create a Hash object where the elements from one array (the keys) are associated with the elements of the other (the values):

```
>>> keys = ['a', 'b', 'c']
>>> values = [1, 2, 3]
>>> hash = {k:v for k, v in zip(keys, values)}
>>> hash
{'a': 1, 'c': 3, 'b': 2}
```

A Simple Hashing Scheme

Name	ASCII for 2 Letters	Multiplication	Modula (% 100)	Home Address
BALL	66 65	66 X 65 = 4,290	4290 % 100	90
LOWELL	76 96	76 X 96 = 6,004	6004 % 100	4
TREE	84 82	84 X 82 = 6,888	6888 % 100	88



Hashing in Python [1/2]

- Built-In hash()

```
>>> map(hash, [0, 1, 2, 3])
[0, 1, 2, 3]
>>> map(hash, ['0','1','2','3'])
[6144018481, 6272018864, 6400019251, 6528019634]
>>> hash('0')
6144018481
```

- 암호화를 위한 hashlib module (Secure Hashes and Message Digests)
  - Two famous cryptographic hash functions: MD5, SHA1
  - **MD5(Message-Digest algorithm 5)**
    - 128비트 암호화 해시 함수
    - 파일이 원본인지를 확인하는데 사용
    - 1991, by R. Rivest
  - **SHA-1 (Secure Hash Algorithm 1)**
    - 160비트 암호화 해시 함수
    - designed by the United States National Security Agency
    - U.S. Federal Information Processing Standard

Example of Hashing: Hashing based on Dept_name

- hash("Music") = 1 hash("History") = 2
- hash("Physics") = 3 hash("Elec. Eng.") = 3

Id	name	Dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hashing in Python [2/2]

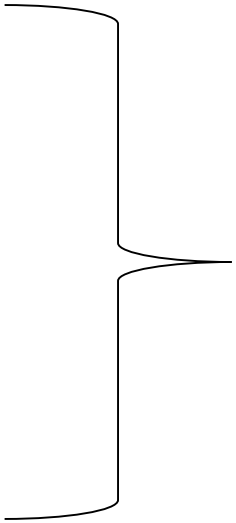
```
>>> import hashlib

>>> hashlib.md5('a')

>>> hashlib.md5('a').digest()
'\x0c\xc1u\xb9\xc0\xf1\xb6\xa81\xc3\x99\xe2iw&a;'
>>> hashlib.md5('a').hexdigest()
'0cc175b9c0f1b6a831c399e269772661'

>>> hashlib.sha512('a')

>>> hashlib.sha512('a').digest()
'\x1f@\xfc\x92\xda$\x16\x94u\ty\xeel\xf5\x82\xf2\xd5\xd7\xd2\x8e\x183]\xe0Z\xbcT\xd0\
>>> hashlib.sha512('a').hexdigest()
'1f40fc92da241694750979ee6cf582f2d5d7d28e18335de05abc54d0560e0f5302860c652bf08d560252
>>>
```

- Methods in hash Objects
- hash.update(data) : Update the hash object with the bytes-like object
- hash.digest() : Return the digest of data passed to the update( ) method
- hash.hexdigest( ) : Like digest( ), except the digest is returned as a string object of double length, containing only hexadecimal digits

Chapters of Data Structure Book

- Linear List (1D Array)
- 2D Array and Matrix
- Linked List
- Stack
- Queue
- Dictionary and Hashing

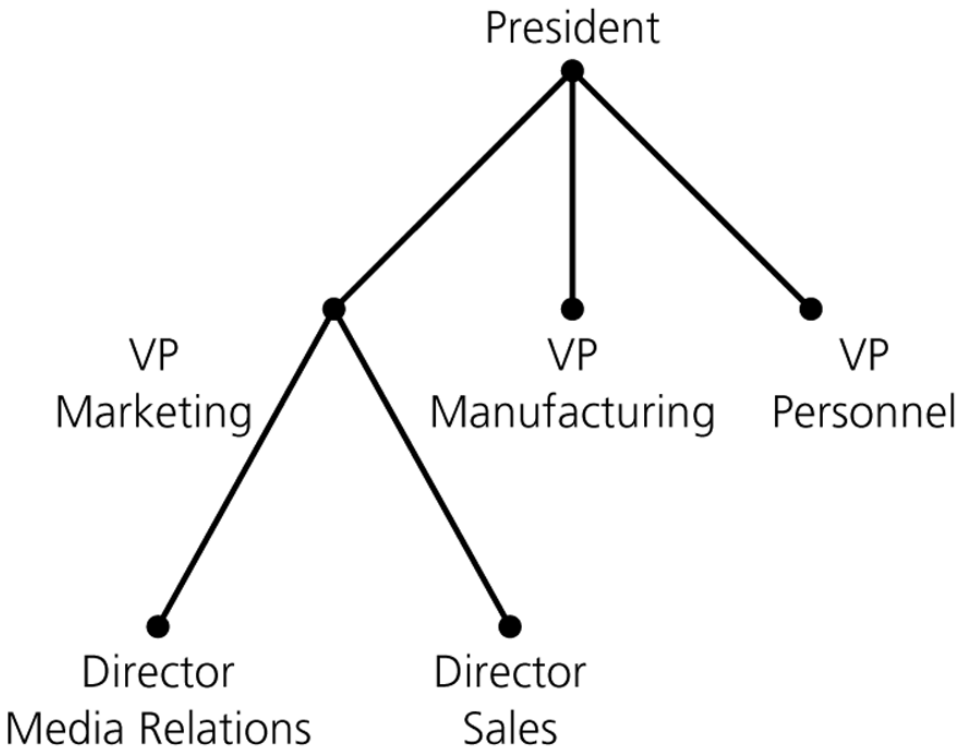
- Tree
  - Binary Tree
  - Priority Queue
  - Binary Search Tree
  - Balanced Tree
- Graph
  - Directed Graph
  - Undirected Graph



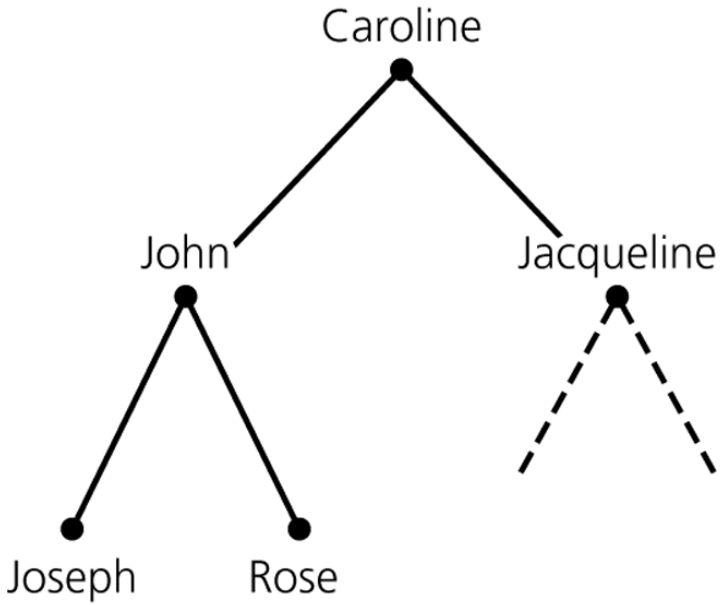
Open source code를 이용하는 경우도 있고, 본인이 개발을 해도 되고!

Real-World Tree Structures

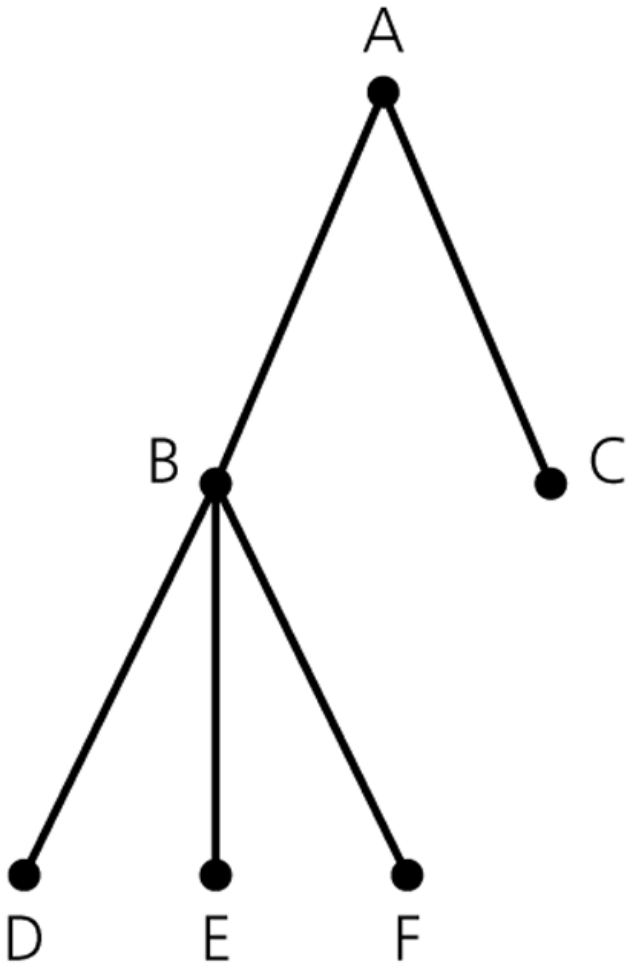
An organization chart



A family tree



A general tree

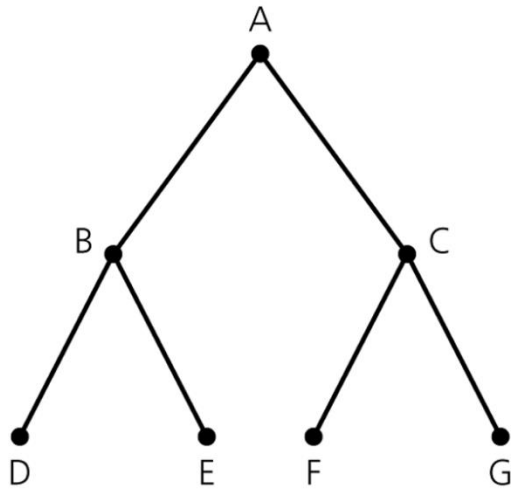


Terminology

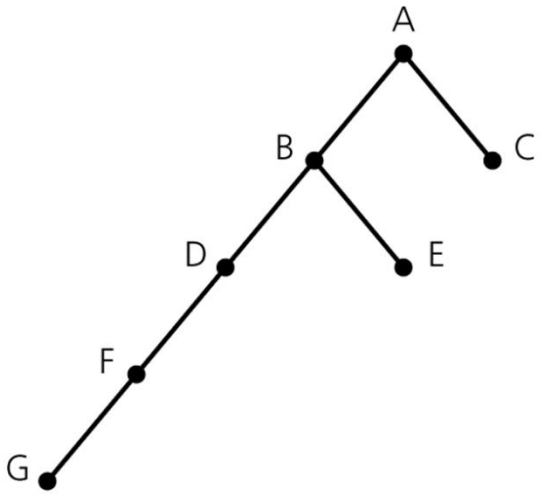
node or vertex
edge
parent
child
siblings
root
leaf
ancestor
descendant
subtree

Height of a Tree

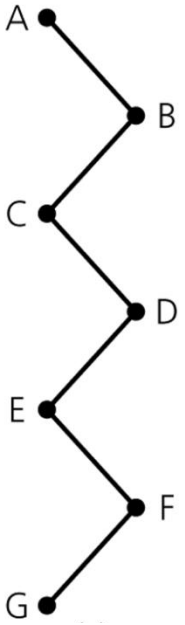
- The number of nodes on the longest path from the root to a leaf



Height 3



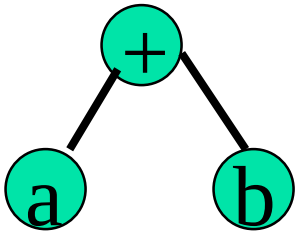
Height 5



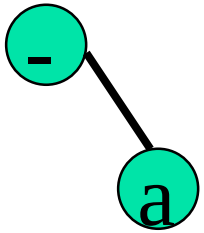
Height 7

Binary Tree Form of Arithmetic Expression

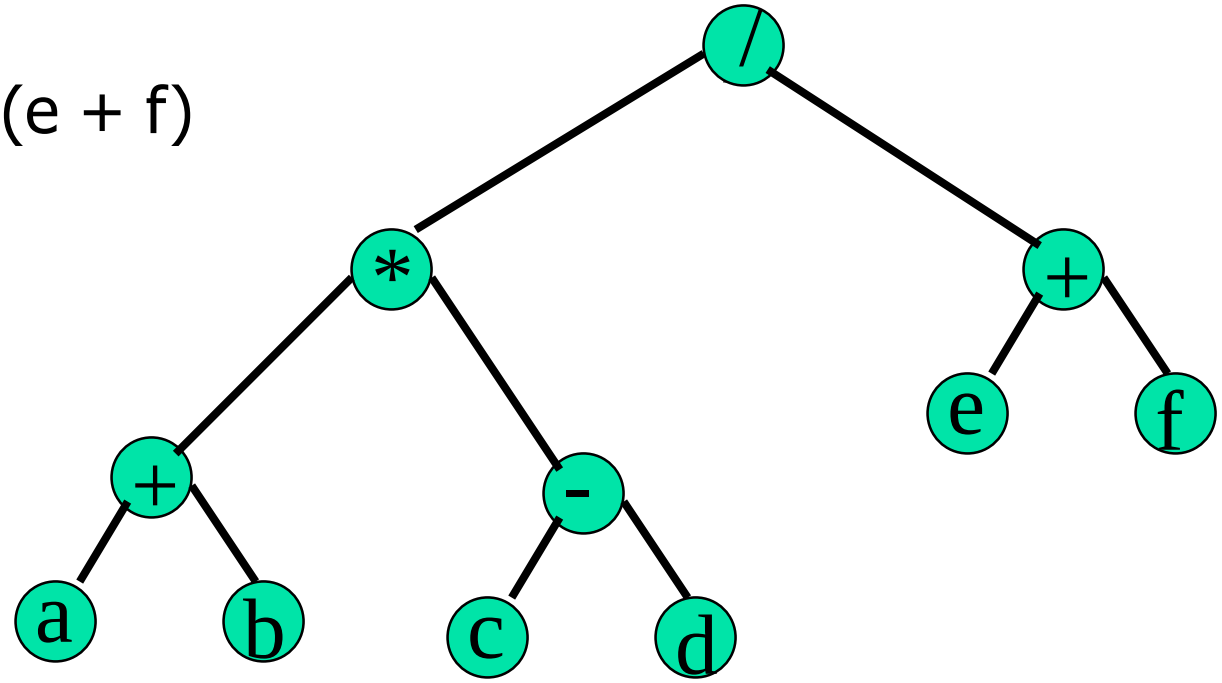
- a + b



- - a



- (a + b) * (c – d) / (e + f)

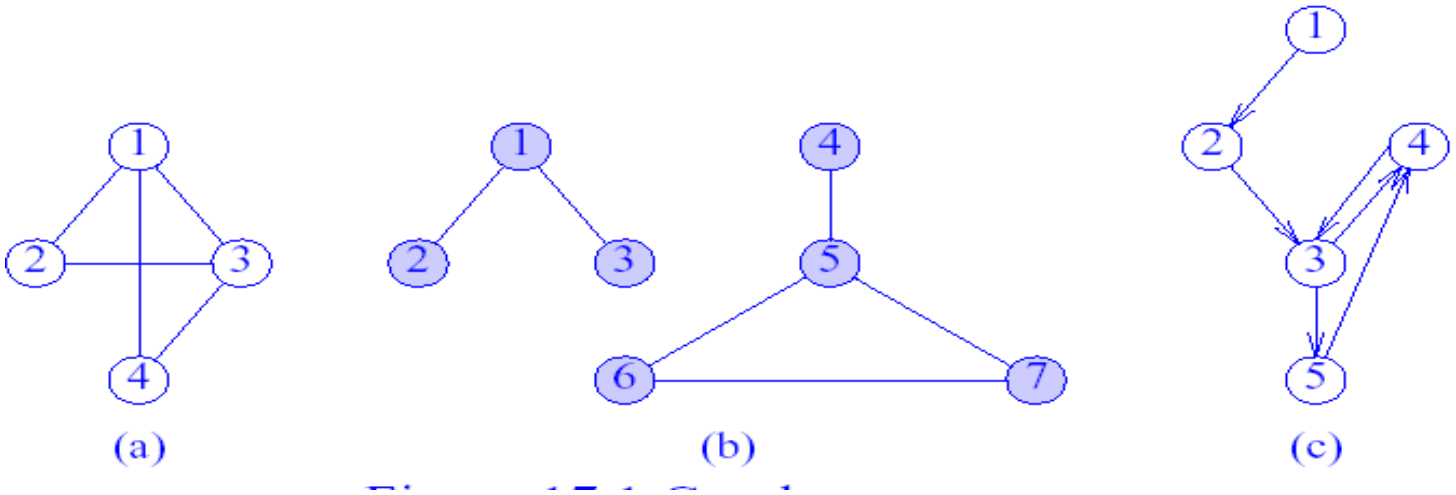


Graph Definition [1/2]

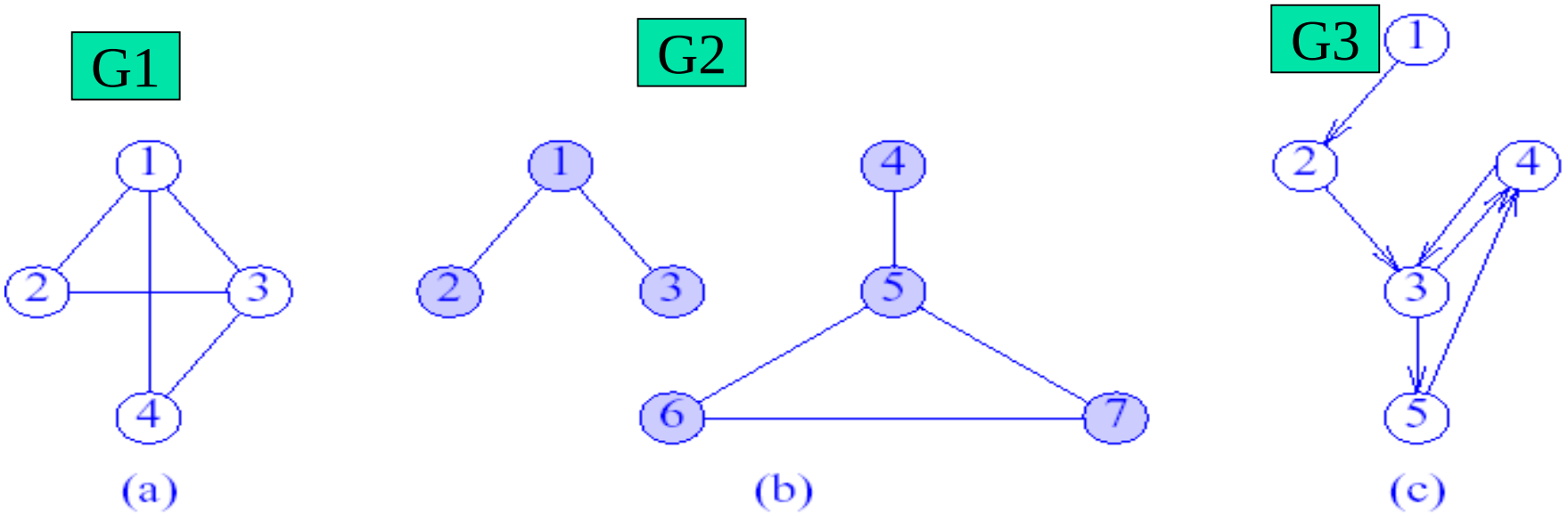
- Graph G = (V, E)
  - Finite set V (=vertices, nodes, points)
  - Finite set E (=edges, arcs, lines)

- Directed edge: orientation
- Undirected edge: no orientation

- Vertices i and j are **adjacent** vertices iff (i,j) is an edge in the graph
- Edge(i,j) is **incident** on the vertices i and j



Graph Definition [2/2]

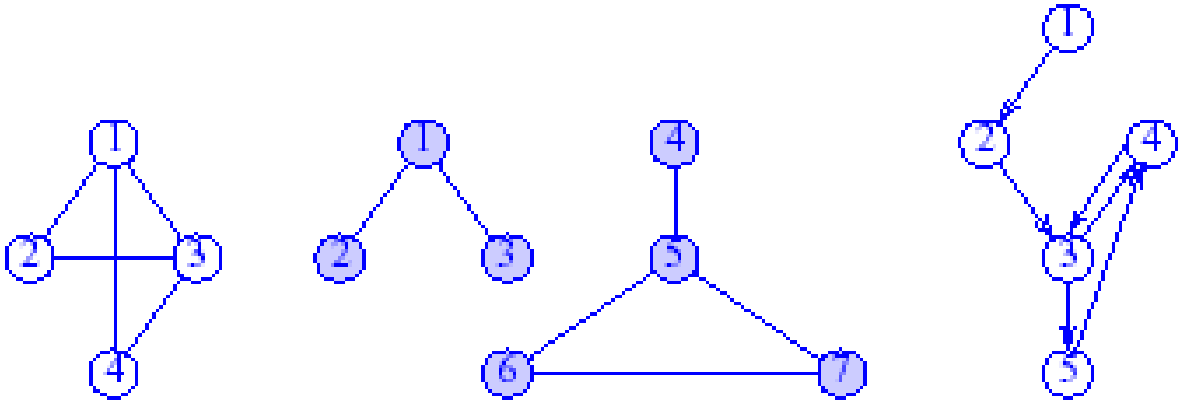


- $G_1 = (V_1, E_1)$:
  $V_1 = \{1, 2, 3, 4\}$, $E_1 = \{(1,2), (1,3), (2,3), (1,4), (3,4)\}$

- $G_2 = (V_2, E_2)$:
  $V_2 = \{1, 2, 3, 4, 5, 6, 7\}$, $E_2 = \{(1,2), (1,3), (4,5), (5,6), (5,7),(6,7)\}$

- $G_3 = (V_3, E_3)$:
  $V_3 = \{1, 2, 3, 4, 5\}$, $E_3 = \{(1,2), (2,3), (3,4), (4,3), (3,5), (5,4)\}$

Matrix Representation for Adjacency of Graph



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

(a)

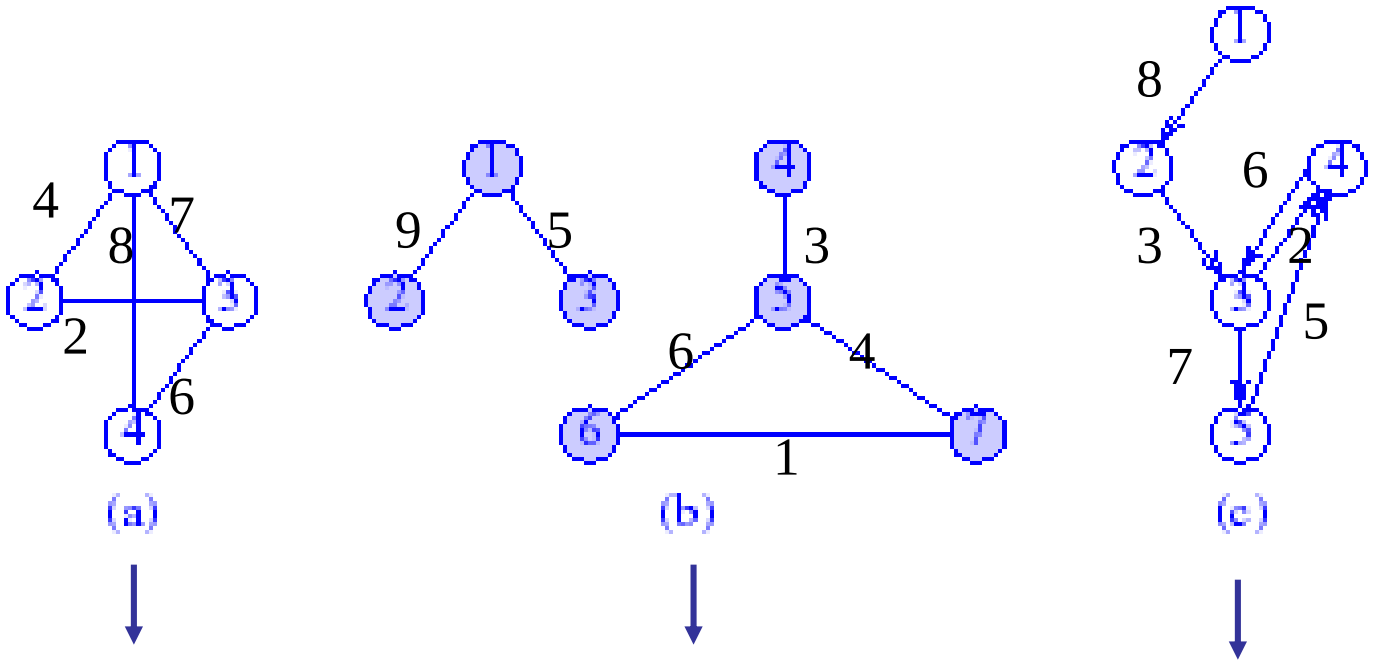
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0
5	0	0	0	1	0	1	1
6	0	0	0	0	1	0	1
7	0	0	0	0	1	1	0

(b)

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	0	1	0

(c)

Matrix Representation for Weighted Graph



	1	2	3	4
1	-	4	7	8
2	4	-	2	-
3	7	2	-	6
4	8	-	6	-

(a)

	1	2	3	4	5	6	7
1	-	9	5	-	-	-	-
2	9	-	-	-	-	-	-
3	5	-	-	-	-	-	-
4	-	-	-	-	3	-	-
5	-	-	-	3	-	6	4
6	-	-	-	-	6	-	1
7	-	-	-	-	4	1	-

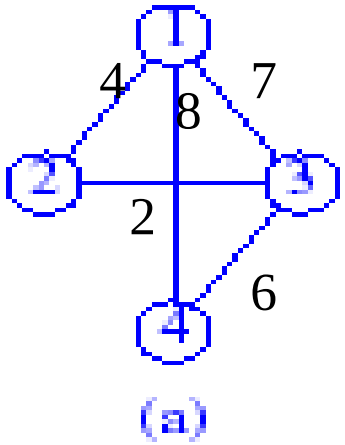
(b)

	1	2	3	4	5
1	-	8	-	-	-
2	-	-	3	-	-
3	-	-	-	2	7
4	-	-	6	-	-
5	-	-	-	5	-

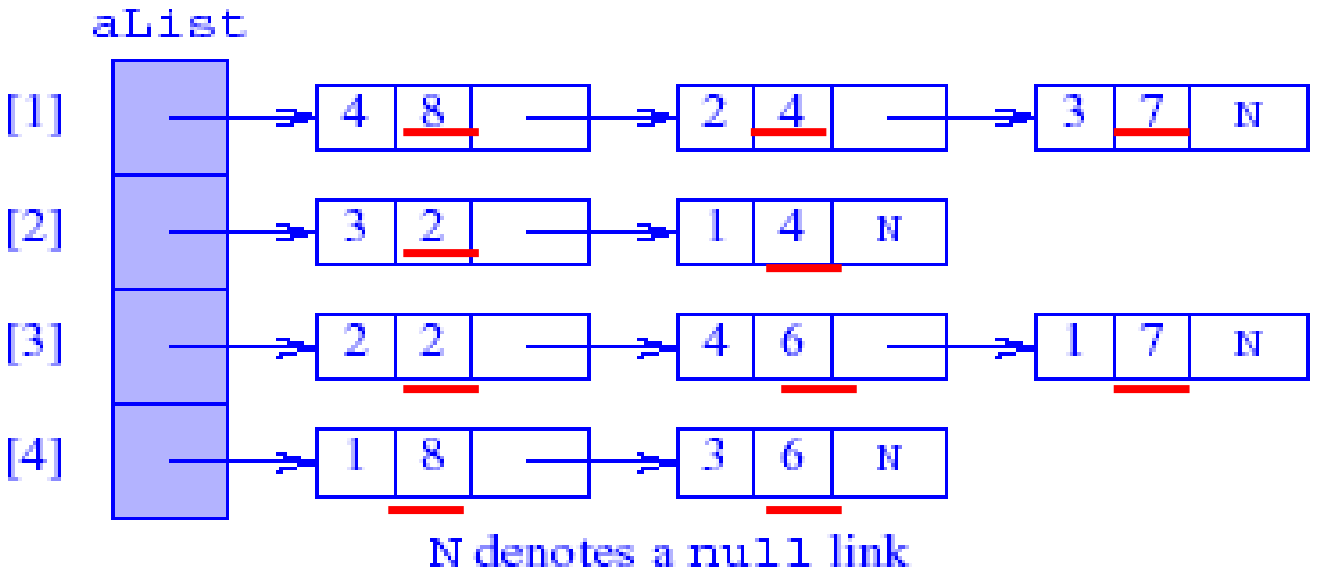
(c)

- denotes a `null` value

Linked-List Representation for Weighted Graph



(a)

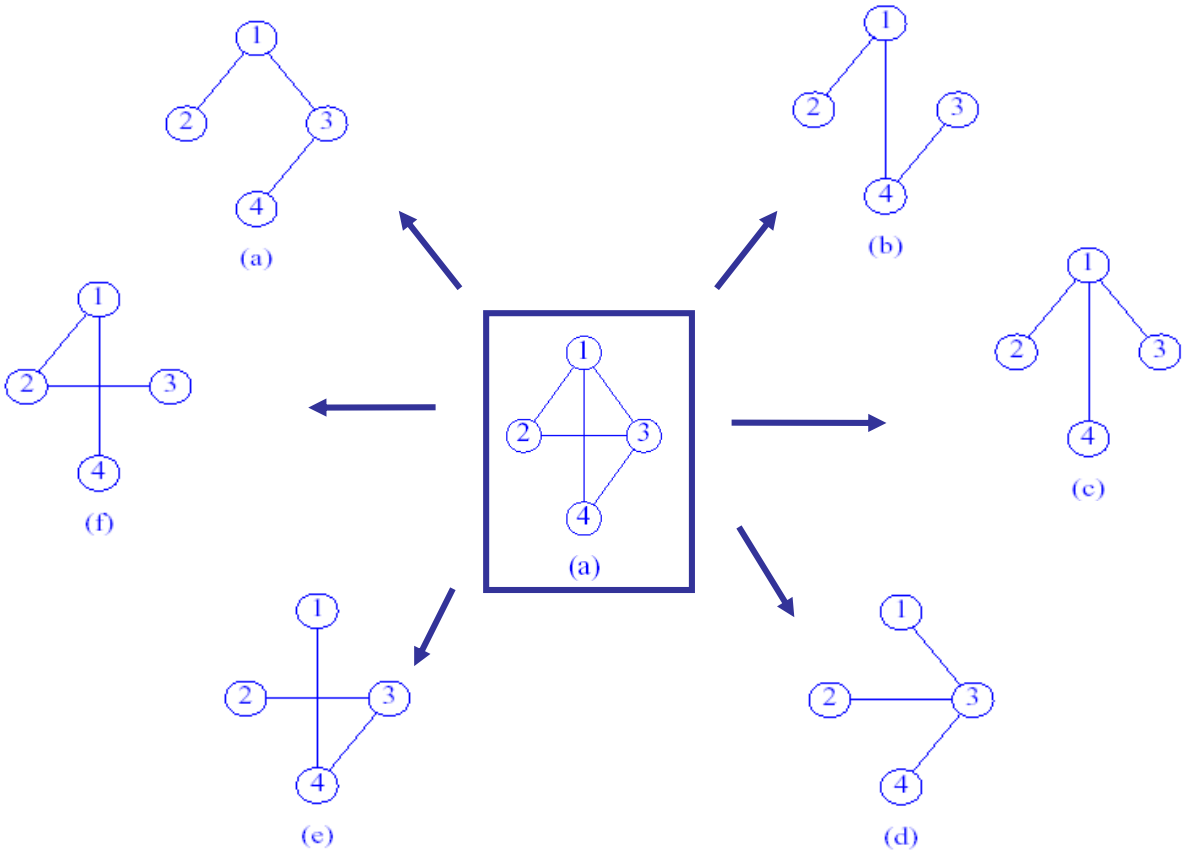


	1	2	3	4
1	-	4	7	8
2	4	-	2	-
3	7	2	-	6
4	8	-	6	-

(a)

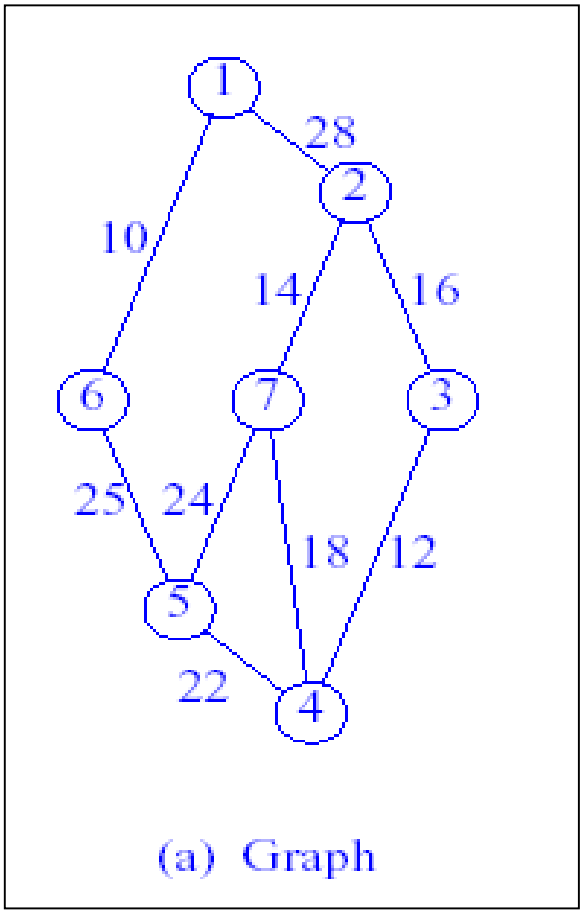
Graph Application: Spanning Trees

- A spanning tree is a tree and a subgraph of G that contains all the vertices of G

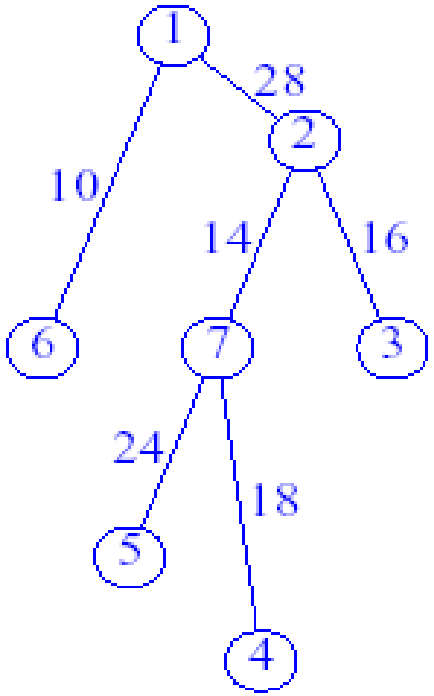


Graph Application: Weighted Graph and its Spanning Trees

** Minimum Cost Spanning Tree **

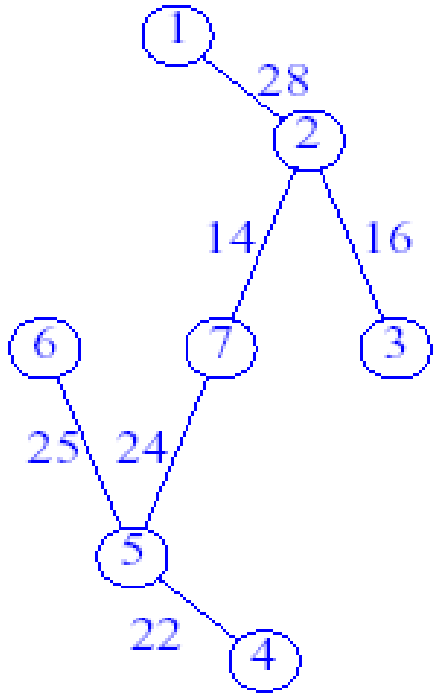


(a) Graph



cost = 110

(b) Spanning tree

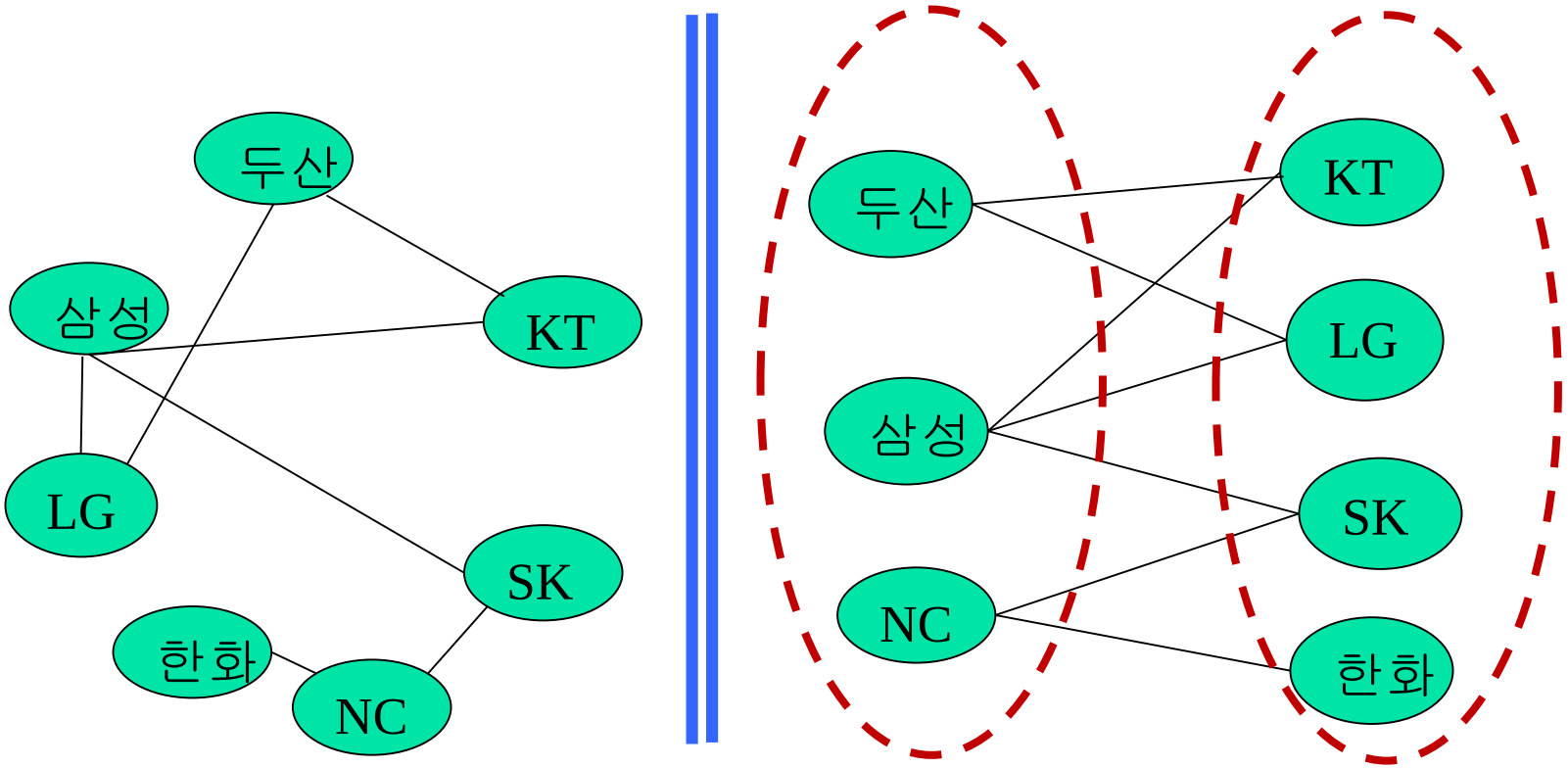


cost = 129

(c) Spanning tree

Graph Application: Bipartite Graphs

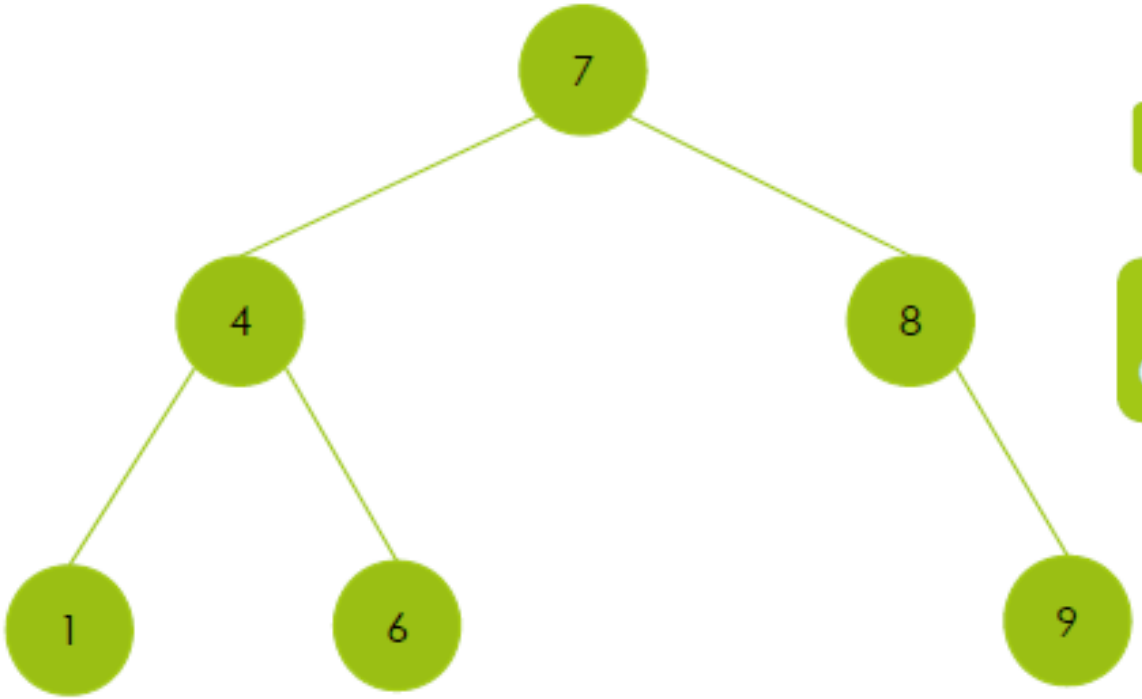
- Partition the vertex set into two subsets A and B so that every edge has one endpoint in A and the other in B
  - Example: 프로야구 개막후 3일동안 (두산, LG), (삼성, 한화), (LG, 삼성), (두산, KT), (NC,SK), (NC, 한화), (SK, 삼성) 의 경기가 있었다. 아직 한번도 서로 경기를 안한 팀들을 2 group을 묶어라? (즉, 이 경기Graph에서 Bipartite Graph를 찾아라?)



OOP Implementation of Binary Search Tree

Binary Search Tree

BST ordering invariant: At any node with key *k*, all keys of elements in the left subtree are strictly less than *k* and all keys of elements in the rightsubtree are strictly greater than *k* (assume that there are no duplicates in the tree)



Binary tree

Satisfies the ordering invariant

BST TreeNode Class [1/3]

```
class TreeNode:

    def __init__(self, key, val, left=None,
                 right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):          # return True or False
        

    def hasRightChild(self):         # return True or False
        

    def isLeftChild(self):           # return True or False
        

    def isRightChild(self):          # return True or False
        
```

```
def isRoot(self):          # return True or False
    

def isLeaf(self):          # return True or False
    

def hasAnyChildren(self): # return True or False
    

def hasBothChildren(self): # return True or False
    

def replaceNodeData(self, key, value, lc, rc):
    
```

BST TreeNode Class [2/3]

```
def findSuccessor(self):
    # self node의 next key value를 가진 node를 찾아서 self node가 delete 되면
    #  그자리에 넣기 위한 작업
    self가 leaf node이면 return no-successor

    self가 right child를 가지고 있으면 return right child side's minimum value

    self가 parent가 있고 left child를 가지고 있으면 return parent
```

```
def findMin(self):
    if self.hasLeftChild():
        return self.leftChild.findMin()
    else:
        return self
```

BST Tree Node [3/3]

```
def sliceOut(self):
    """ move node's child to its own position """
    if self.parent and self.hasRightChild():
        if self.isLeftChild():
            self.parent.leftChild = self.rightChild
        else:
            self.parent.rightChild = self.rightChild
    # !!! the successor node never has a left child.
```

```
def inorder_traverse(self):
    # ! in-order traverse prints out an sorted list.
    if self.hasLeftChild():
        self.leftChild.inorder_traverse()
    print(self.payload)
    if self.hasRightChild():
        self.rightChild.inorder_traverse()
```

BinarySearchTree Class [1/7]

```
class BinarySearchTree:

    def __init__(self):
            self.root = None
            self.size = 0


    def length(self):


    def __len__(self):


    def __iter__(self):
```

BinarySearchTree Class [2/7]

```
def put(self, key, val):
        if self.root:
        self._put(key, val, self.root)
        else:
        self.root = TreeNode(key, val)
        self.size += 1
```

```
def _put(self, key, val, currentNode):
    key 값이 currentNode.key 보다 작으면
        currentNode에 left child가 있으면 _put() recursion with the left child
        currentNode에 left child가 없으면 key값을 가진 node를 생성하여 currentNode의 left child로 만듬

    key 값이 currentNode.key 보다 크면  (key이므로 값이 같을수는 없다)
      currentNode에 right child가 있으면 _put() recursion with the right child
      currentNode에 right child가 없으면 key값을 가진 node를 생성하여 currentNode에 right child로 만듬
```

BinarySearchTree Class [3/7]

```
def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    return None
```

```
def _get(self, key, currentNode):
    current node가 없으면 return None
    current node의 key가 원하는 key면 return current node
    current node의 key가 원하는 key보다 크면
        current node의 left child를 가지고 _get() recursion
    current node의 key가 원하는 key보다 작으면
        current node의 right child를 가지고 _get() recursion
```

```
def __setitem__(self, k, v):
    self.put(k, v)
```

```
def __getitem__(self, key):
    [redacted]

def __contains__(self, key):
    [redacted]
```

BinarySearchTree Class [4/7]

```
def delete(self, key):
    if self.size > 1:
        nodeToRemove = self._get(key, self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size -= 1
        else:   raise KeyError('Error, key is not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = 0
    else:   raise KeyError('Error, key not in tree')
```

BinarySearchTree Class [5/7]

def remove(self, currentNode):

current node가 leaf node이면

current node가 parent node의 left child이면 parent node의 left child part를 none으로 변경

current node가 parent node의 right child이면 parent node의 right child part를 none으로 변경

current node가 leaf node가 아니면 child가 1개 or 2개가 있는 경우

(A) left child와 right child를 다 가지고 있는 경우

replace current node with next largest node (only key and payload)

successor's right child move to its parent's position. This is done with 'node.sliceOut()'

(B) left child를 가지고 있는데

(B-1: LL type) current node가 parent node의 left child이면

parent node의 left child part를 current node의 left child로 변경

(B-2: RL type) current node가 parent node의 right child이면

parent node의 right child part를 current node의 left child로 변경

(C) right child를 가지고 있는데

(C-1: LR type) current node가 parent node의 left child이면

parent node의 right child part를 current node의 right child로 변경

(C-2: RR type) current node가 parent node의 right child이면

parent node의 right child part를 current node의 left child로 변경

Case Analysis of Remove [6/7]

A

```
#          parent
#            |
#        current node << REMOVE
#       /             \
#  left-child      right-child
#-------------------------------------------------------
# REPLACE CURRENT NODE with NEXT LARGEST NODE (only key and payload)
#
#        current node << REMOVE
#       /             \
#  left-child      right-child << NEXT?
#                 /
#           next-left-child
#           /
#    next-next-left-child  <<< NEXT!!! = (SUCCESSOR)
#                     \
#                       Successor's rightChild
#-------------------------------------------------------
# Successor's right child move to its parent's position.
# This is done with 'node.sliceOut()'
#
#         (SUCCESSOR)
#       /             \
#  left-child      right-child << NEXT?
#                 /
#           next-left-child
#           /
#        Successor's right child
```

B-1: LL case

```
# current node is left child of its parent node.
#                  parent
#                 /
#           currentnode << remove
#          /
#    childnode
```

B-2: RL case

```
# current node is right child of its parent node.
#                 parent
#                     \
#                   currentNode <<< REMOVE
#                    /
#                childnode
```

C-1: LR case

```
# current node is left child of its parent node.
#             parent
#           /
#       currentNode <<< REMOVE
#                  \
#                   childnode
```

C-2: RR case

```
# current node is right child of its parent node.
#        parent
#              \
#              currentNode <<< REMOVE
#                         \
#                          childnode
```

BinarySearchTree Class [7/7]

```
def main():
    bst = BinarySearchTree()
    input_data = (17, 5, 25, 2, 11, 29, 38, 9, 16, 7, 8)
    for i in input_data:
        bst.put(i, i)
    bst.root.inorder_traverse()
    #
    print('remove 5')
    bst.delete(5)
    bst.root.inorder_traverse()
    #
    print(‘put 39')
    bst.put(39, 39)
    bst.root.inorder_traverse()
```