Linked List, Stack, and Queue

Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

Weekly Objectives

- This week, we learn the first set of data structures: linked list, stack, and queue.
- Objectives are
  - Understanding the definition of abstract data types
  - Firmly understanding how references work
  - Understanding various linked list, stack, and queue structures
    - Singly linked list, doubly linked list, circular linked list…
    - Able to implement a stack and a queue with a list
  - Understanding the procedures of linked list, stack, and queue management
    - Insert, delete, search…
    - Should be able to estimate the number of steps for inserts, deletes, and searches

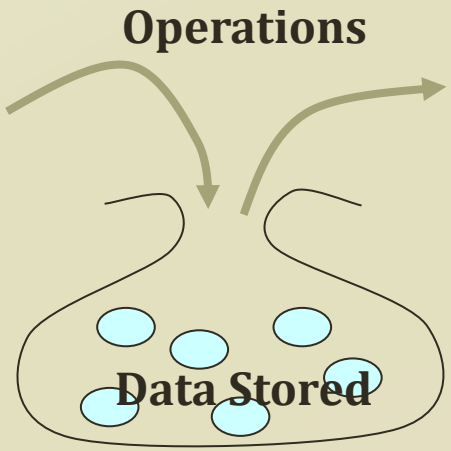
ARRAY FOR LIST

Scenario for List

- You are looking for Koh in the mass public
  - You are going to ask one by one “Are you Ms. Koh?”
  - Sometimes, you ask the question to a person multiple times
  - You realize that this is not going to work!
  - So, you line up the people and again ask the question one-by-one
- You are looking for a restroom on the floor
  - The floor has a long corridor, and every room has an entrance to the corridor
  - You only need to follow the corridor
- You have a dump of information of customers
  - How to store, search, and manipulate the information
- You line them up as a ***list***!

Abstract Data Types

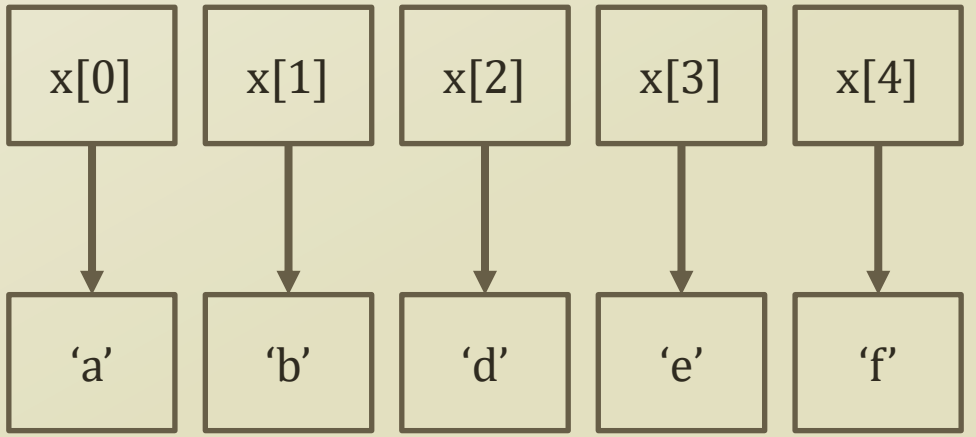
- An abstract data type (ADT) is an abstraction of a data structure
  - An ADT specifies:
    - Data stored
    - Operations on the data
    - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order



Creating a List by Array

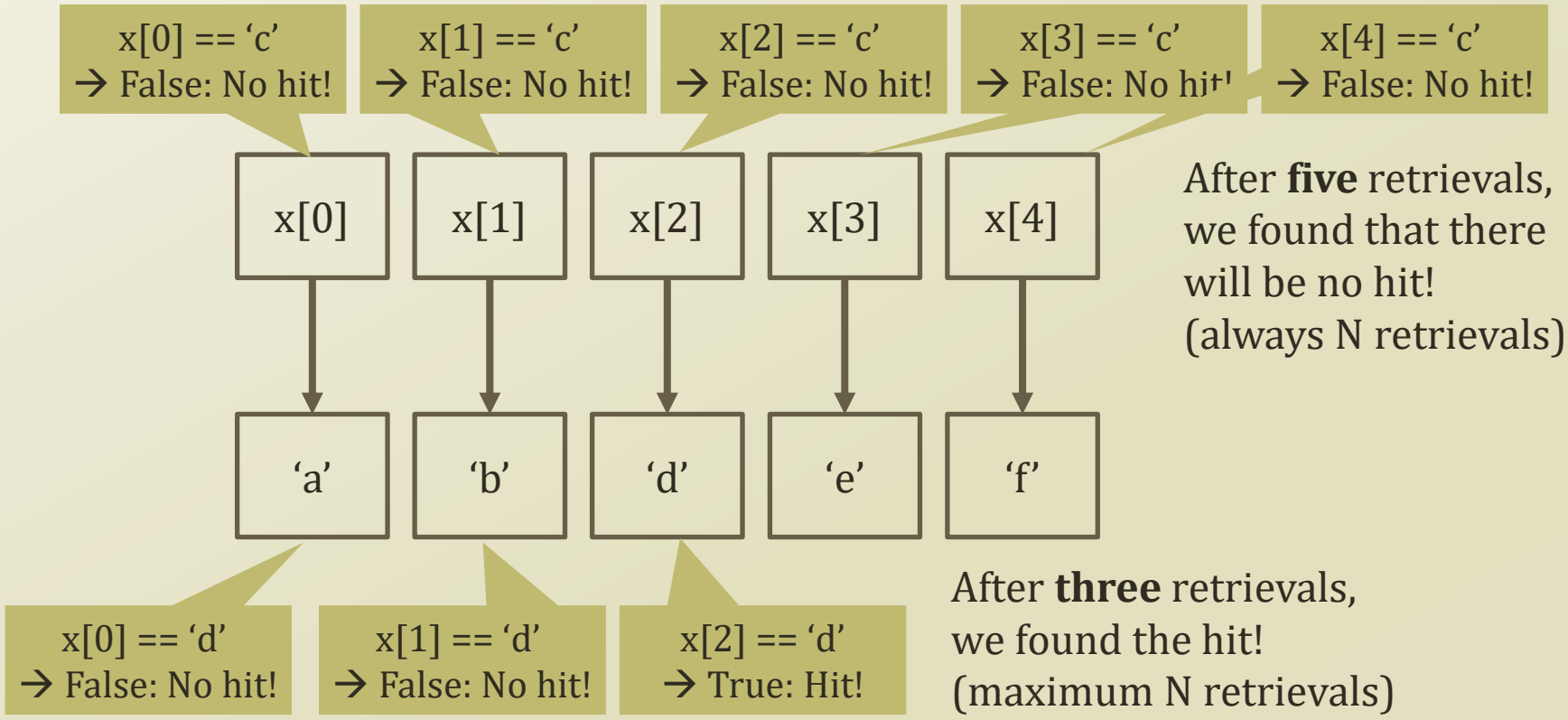
- Array (in our Python example, List, yet we will use only its index function)
  - Each element is accessible by index
  - Index is typically zero or a positive integer
  - Very simple creation
    - That's why people use it

```
x = ['a', 'b', 'd', 'e', 'f']
```



Search procedure in array

- Let's find 'd' and 'c' from the list in an array
  - Of course, you can use '*in*', but we commit ourselves to use indexes only
- Then, navigating from the first to the last until hit is the only way

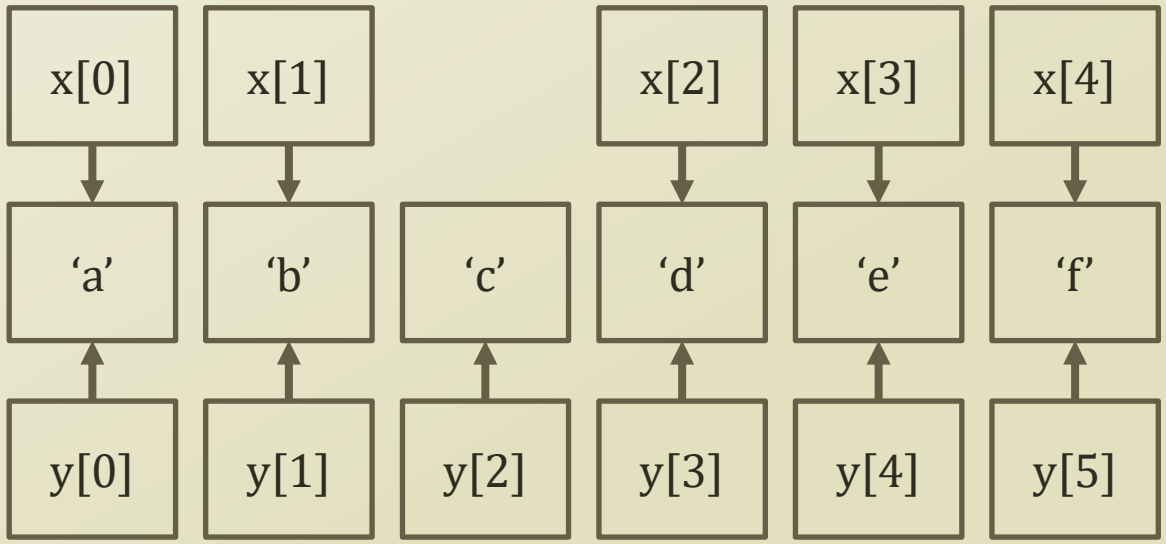


After **five** retrievals, we found that there will be no hit! (always N retrievals)

After **three** retrievals, we found the hit! (maximum N retrievals)

Insert procedure in array

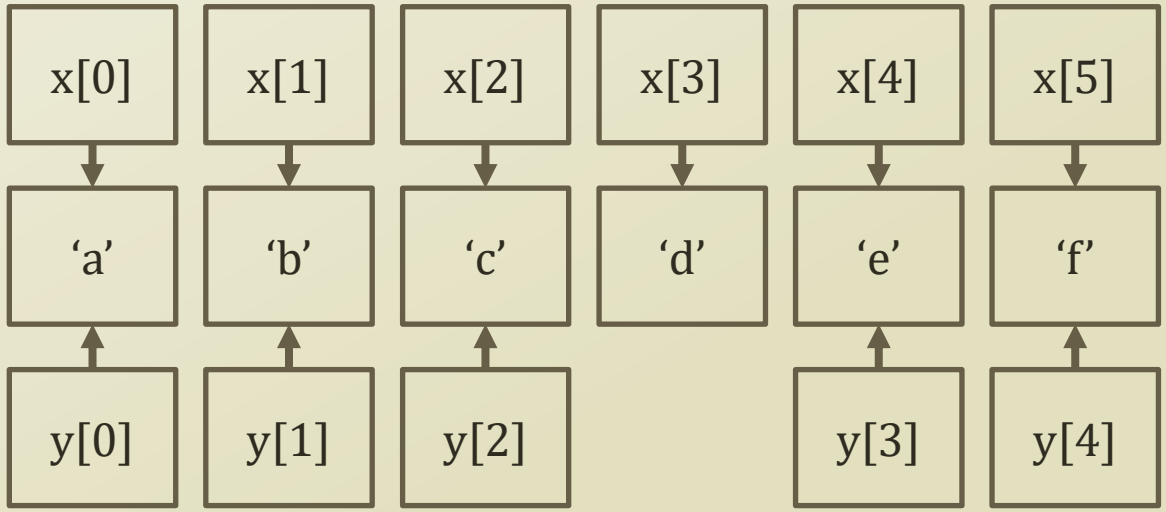
- Let's insert 'c' between 'b' and 'd' in the list *(a = insert position index)*
  - First, make new list, or y, with six cells
  - Second, copy the reference links of x[0:a-1] to y[0:a-1] *(retrieval cnt.: a)*
  - Third, put a reference link to 'c' in y[a] *(retrieval cnt.: 1)*
  - Fourth, copy the reference links of x[a:] to y[a+1:] *(retrieval cnt.: n-a-1)*
  - Fifth, change x's reference to y's reference
  - Total count of retrievals = a + 1 + n – a – 1 = n




```
x = ['a', 'b', 'd', 'e', 'f']
idxInsert = 2
valInsert = 'c'

y = list(range(6))

for itr in range(0, idxInsert):
    y[itr] = x[itr]

y[idxInsert] = valInsert

for itr in range(idxInsert, len(x)):
    y[itr+1] = x[itr]
```

Delete procedure in array

- Let's remove 'd' in the list *(a = delete position index)*
  - First, make new list, or y, with five cells
  - Second, copy the reference links of x[0:a-1] to y[0:a-1] *(retrieval cnt.: a)*
  - Third, copy the reference links of x[a+1:] to y[a:] *(retrieval cnt.: n-a-1)*
  - Fifth, change x's reference to y's reference
  - Total count of retrievals = a + n – a – 1 = n - 1




```
"""
delete example
"""
idxDelete = 3

y = list(range(5))

for itr in range(0, idxDelete):
    y[itr] = x[itr]

for itr in range(idxDelete+1, len(x)):
    y[itr-1] = x[itr]

x = y
```

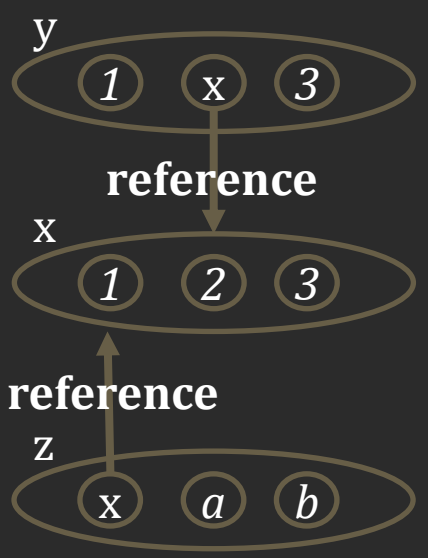
Problems in Array

- Whenever you put something in or get something out
  - You have to perform line-wise retrievals
    - Which is N retrievals
      (by assuming 100,000 – 1 ≒100,000)
  - This process is just like that
    - There is a line of airline passengers
    - You want to put a passenger in the middle of the line because his flight is about to leave
    - You are moving all the passengers one step back
    - Then, you put the customer in the line
- What-if we have a magic to create a space in the middle of the line?
  - Array → you are bounded to the 1-dimension that you have
  - Linked list → you are bounded no more!

LINKED LIST

Detour: Assignment and Equivalence

```
x = [1, 2, 3]
y = [100, x, 120]
z = [x, 'a', 'b']

print('x : ', x)
print('y : ', y)
print('z : ', z)

x[1] = 1717

print('₩nx : ', x)
print('y : ', y)
print('z : ', z)

x[1] = 2
x2 = [1, 2, 3]
if x == x2:
    print("Values are equivalent")
else:
    print("Values are not equivalent")

if x is x2:
    print("Values are stored at the same place")
else:
    print("Values are not stored at the same place")

if x[1] is y[1][1]:
    print("Values are stored at the same place")
else:
    print("Values are not stored at the same place")
```

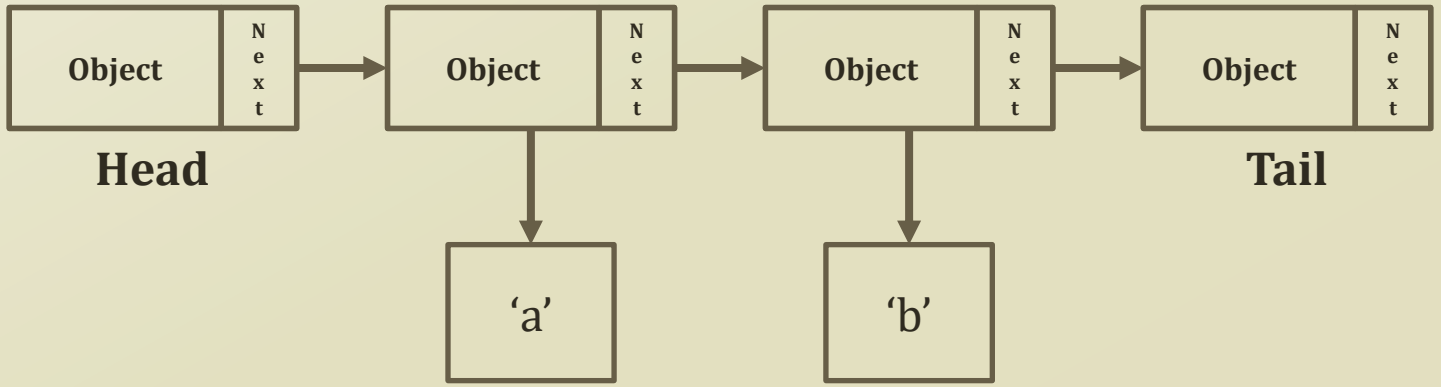



```
x :  [1, 2, 3]
y :  [100, [1, 2, 3], 120]
z :  [[1, 2, 3], 'a', 'b']

x :  [1, 1717, 3]
y :  [100, [1, 1717, 3], 120]
z :  [[1, 1717, 3], 'a', 'b']
Values are equivalent
Values are not stored at the same place
Values are stored at the same place
```

- One variable's value is changed
- But, you see three changes
- Why this happened?
  - Because of references
  - x has two references from y and z
  - The values of y and z are determined by x, and x is changed
    - See the ripple effects
- ***==***
  - Checks the equivalence of two referenced values
- ***is***
  - Checks the equivalence of two referenced objects' IDs

Basic Structure: Singly linked list

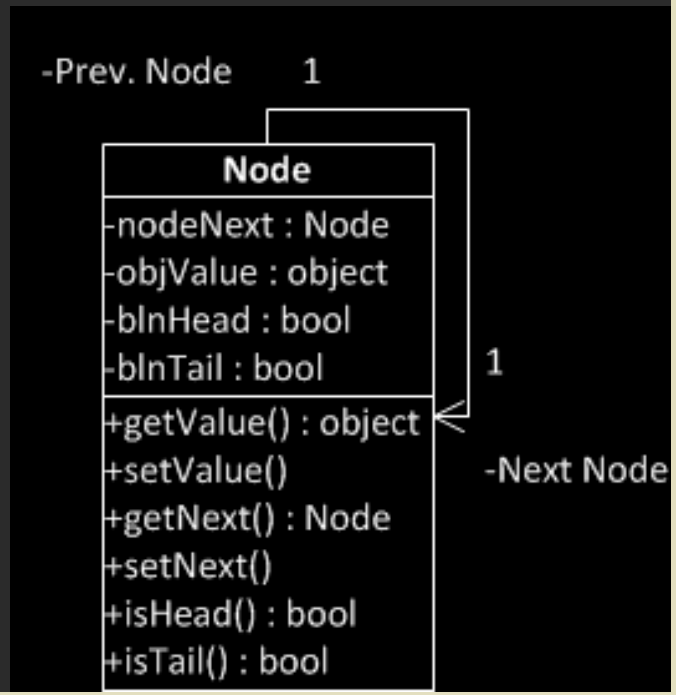
- Construct a singly linked list with nodes and references
  - A node consists of
    - A variable to hold a reference to its next node
    - A variable to hold a reference to its value object
  - Special nodes: Head and Tail
    - You can construct the singly linked list without them
    - But, using them makes search, insert and delete more convenient
  - Generally, requires more coding than array



Implementation of Node class

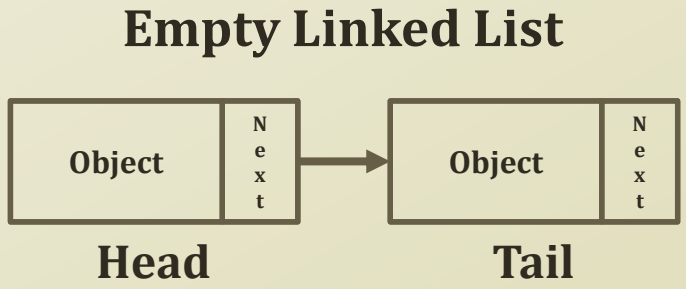
- Member variables
  - Variable to reference the next node
  - Variable to hold a value object
  - (Optional) Variable to check whether it is a head or not
  - (Optional) Variable to check whether it is a tail or not
- Member functions
  - Various set/get methods

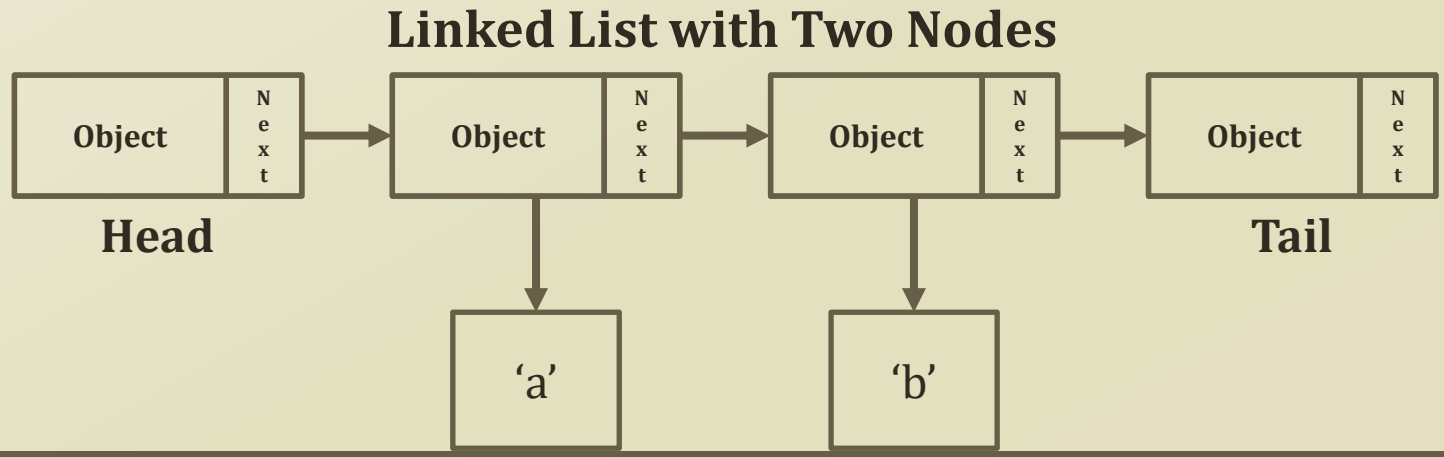
```
class Node:
    nodeNext = None
    nodePrev = ''
    objValue = ''
    blnHead = False
    blnTail = False
    def __init__(self, objValue = '', nodeNext = None, blnHead = False, blnTail = False):
        self.nodeNext = nodeNext
        self.objValue = objValue
        self.blnHead = blnHead
        self.blnTail = blnTail
    def getValue(self):
        return self.objValue
    def setValue(self, objValue):
        self.objValue = objValue
    def getNext(self):
        return self.nodeNext
    def setNext(self, nodeNext):
        self.nodeNext = nodeNext
    def isHead(self):
        return self.blnHead
    def isTail(self):
        return self.blnTail

node1 = Node(objValue = 'a')
nodeTail = Node(blnTail = True)
nodeHead = Node(blnHead = True, nodeNext = node1)
```



Head and Tail

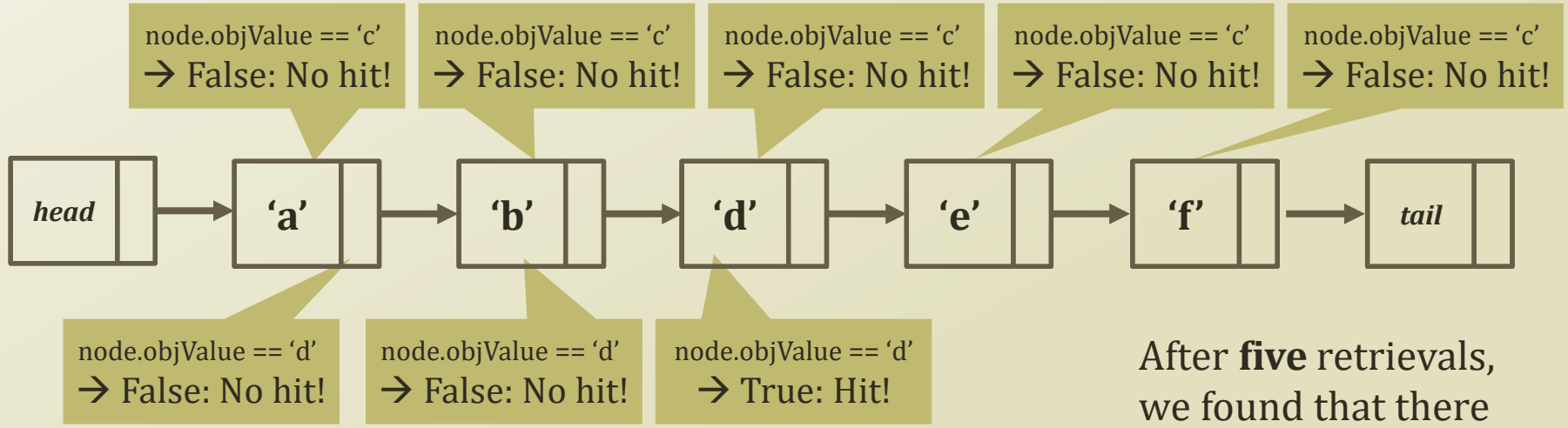
- Specialized node
  - Head : Always at the first of the list
  - Tail : Always at the last of the list
  - These are the two corner stone showing the start and the end of the list
- These are optional nodes.
  - Linked list works okay without these
  - However, having these makes implementation very convenient
  - Any example?





Search procedure in singly linked list

- Again, let's find 'd' and 'c' from the list
- Just like an array, navigating from the first to the last until hit is the only way
- No difference in the search pattern, though you cannot use index any further!
  - Your list implementation may include the index function, but it is not required in the linked list

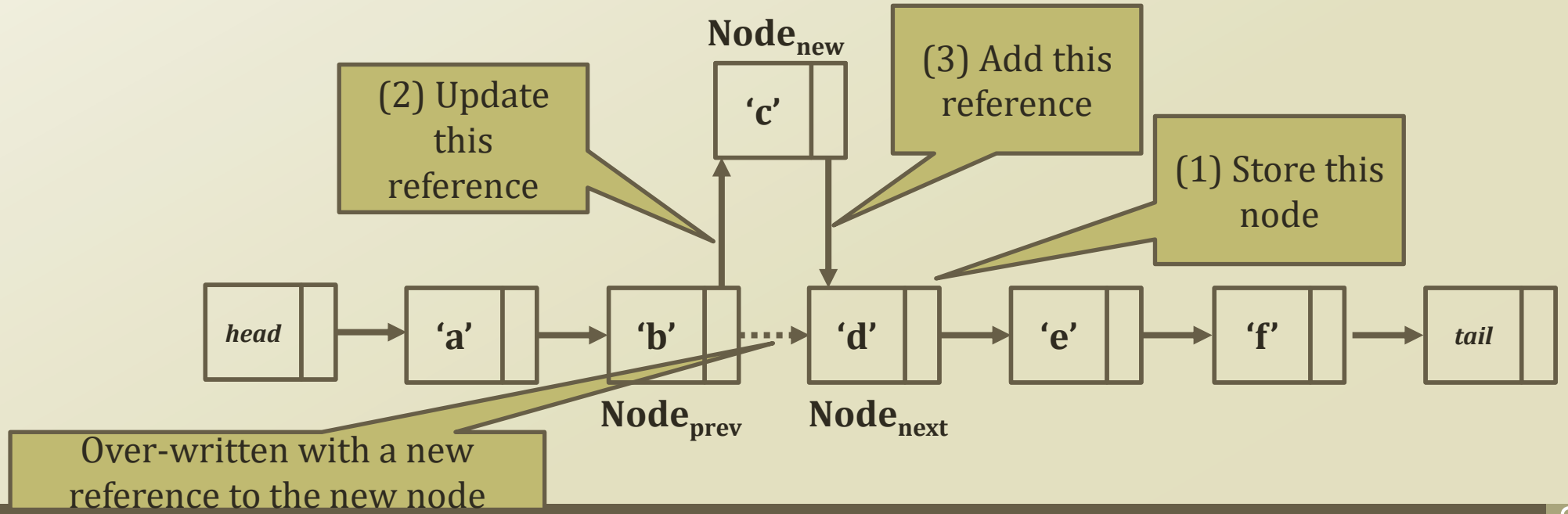


After **three** retrievals,
we found the hit!
(maximum N retrievals)

After **five** retrievals,
we found that there
will be no hit!
(always N retrievals)

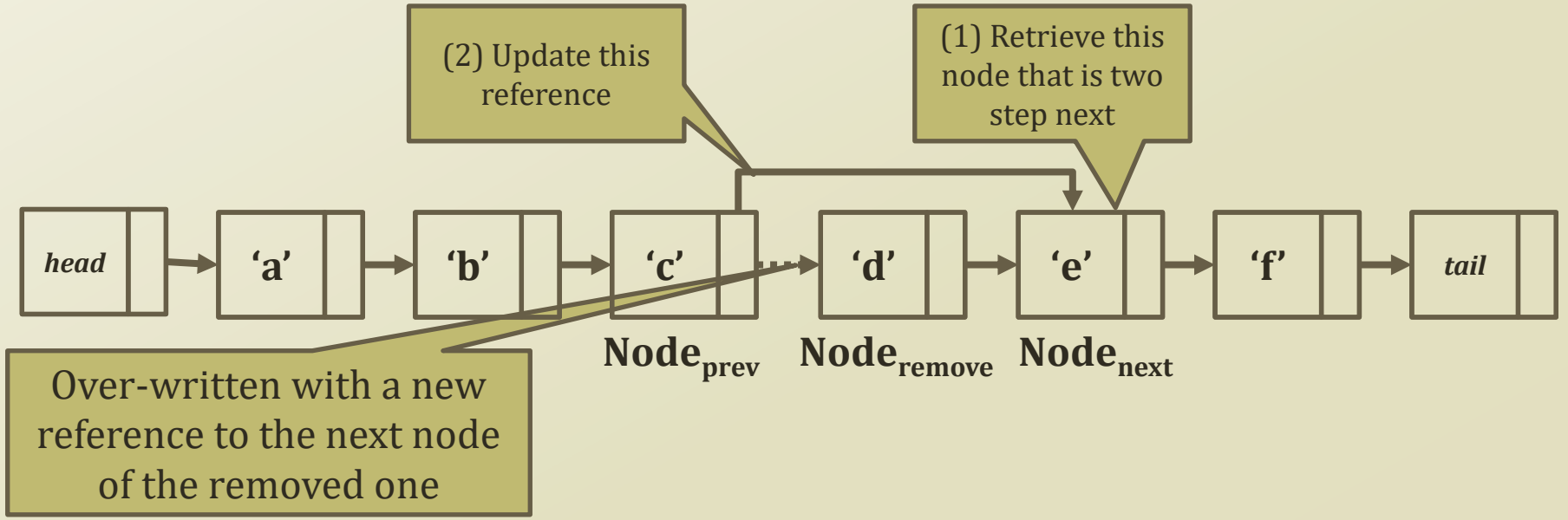
Insert procedure in singly linked list

- This is the moment that you see the power of a linked list
- Last time, you need N retrievals to insert a value in the array list
- This time, you need only three operations
  - With an assumption that you have a reference to the node, $Node_{prev}$ that you want to put your new node next
  - First, you store a Node, or a $Node_{next}$, pointed by a reference from $Node_{prev}$'s nodeNext member variable
  - Second, you change a reference from $Node_{prev}$'s nodeNext to $Node_{new}$
  - Third, you change a reference from $Node_{new}$'s nodeNext to $Node_{next}$



Delete procedure in singly linked list

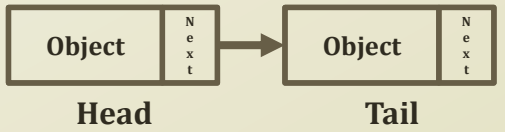
- This is the another moment that you see the power of a linked list
- Last time, you need N retrievals to delete a value in the array list
- This time, you need only three operations
  - With an assumption that you have a reference to the node, $Node_{prev}$ that you want to remove the node next
  - First, you retrieve $Node_{next}$ that is two steps next from $Node_{prev}$
  - Second, you change a reference from $Node_{prev}$'s nodeNext to $Node_{next}$
- The node will be removed because there is no reference to $Node_{remove}$



Implementation of Singly linked list

```python
class SinglyLinkedList:
    nodeHead = ''
    nodeTail = ''
    size = 0
    def __init__(self):
        self.nodeTail = Node(blnTail=True)
        self.nodeHead = Node(blnHead=True, nodeNext=self.nodeTail)

    def insertAt(self, objInsert, idxInsert):
        nodeNew = Node(objValue=objInsert)
        nodePrev = self.get(idxInsert - 1)
        nodeNext = nodePrev.getNext()
        nodePrev.setNext(nodeNew)
        nodeNew.setNext(nodeNext)
        self.size = self.size + 1

    def removeAt(self, idxRemove):
        nodePrev = self.get(idxRemove - 1)
        nodeRemove = nodePrev.getNext()
        nodeNext = nodeRemove.getNext()
        nodePrev.setNext(nodeNext)
        self.size = self.size - 1
        return nodeRemove.getValue()

    def get(self, idxRetrieve):
        nodeReturn = self.nodeHead
        for itr in range(idxRetrieve + 1):
            nodeReturn = nodeReturn.getNext()
        return nodeReturn

    def printStatus(self):
        nodeCurrent = self.nodeHead
        while nodeCurrent.getNext().isTail() == False:
            nodeCurrent = nodeCurrent.getNext()
            print(nodeCurrent.getValue(), end=" ")
        print("")

    def getSize(self):
        return self.size
```

Empty Linked List




```python
list1 = SinglyLinkedList()
list1.insertAt('a', 0)
list1.insertAt('b', 1)
list1.insertAt('d', 2)
list1.insertAt('e', 3)
list1.insertAt('f', 4)
list1.printStatus()

list1.insertAt('c', 2)
list1.printStatus()

list1.removeAt(3)
list1.printStatus()
```

```
a b d e f
a b c d e f
a b c e f
```

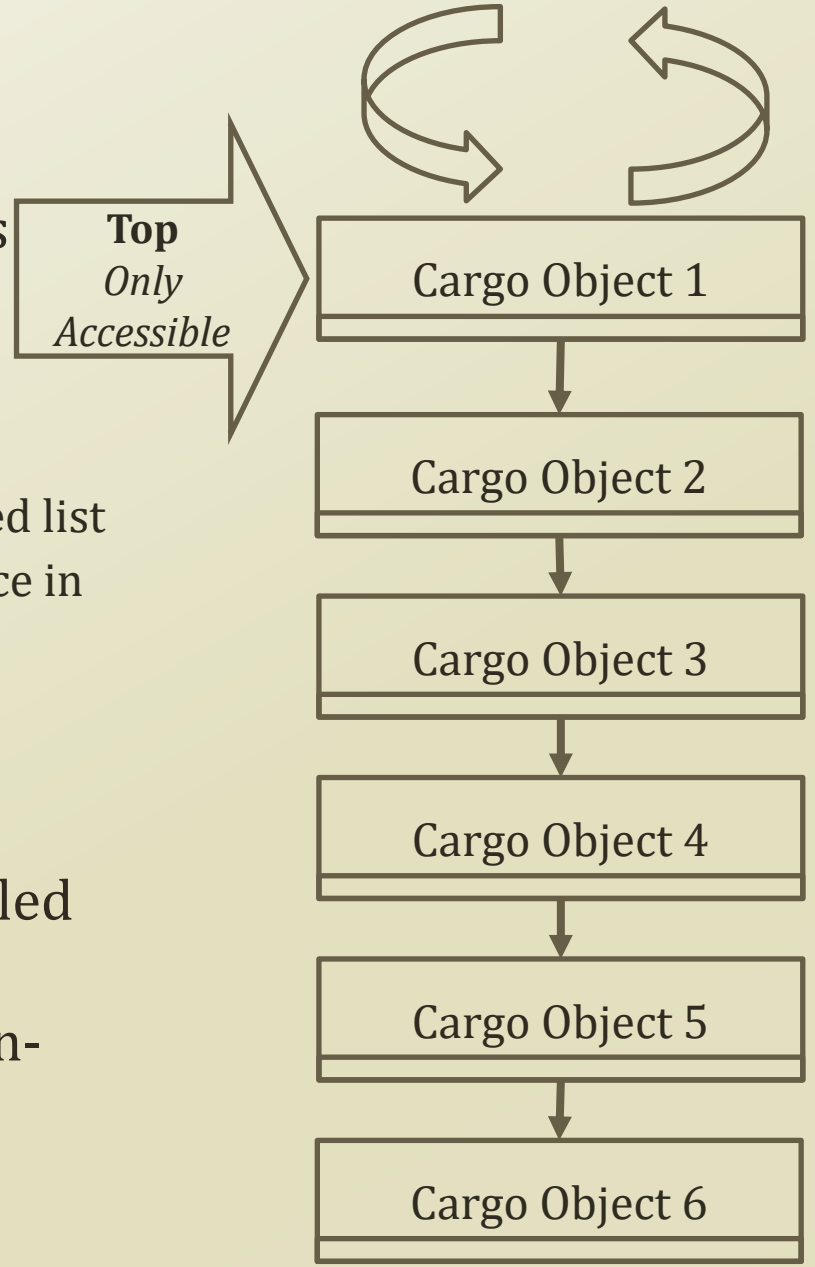
STACK AND QUEUE

Scenario for Stack

- Back seat of a taxi
  - One way in and out
  - We are traveling from the source to our destinations
  - Who should seat first and last?
- A scenario for a taxi
  = A scenario for SCM
  - A cargo plane with one way in and out
  - How to organize the cargo loading to the plane?

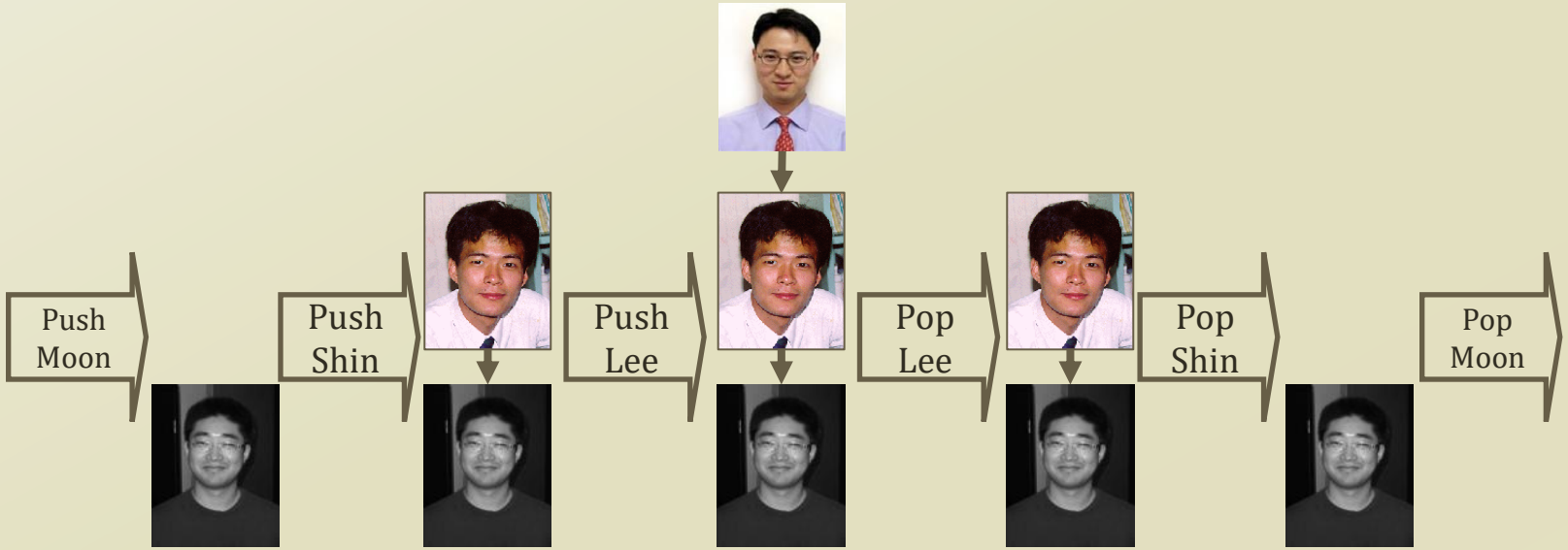
Structure of Stack

- Stacks are linear like linked lists
  - A variation of a singly linked list
- Difference
  - Voluntarily giving up
    - Access to the middle in the linked list
    - Only accesses to the first instance in the list
  - The first instance in the list
    = The top instance in the stack
- An item is inserted or removed from the stack from one end called the “top” of the stack.
- This mechanism is called Last-In-First-Out (LIFO).



Operation of Stack

- Stack operation
  - Push
    - = Insert an instance at the first in the linked list
    - = Put an instance at the top in the stack
  - Pop
    - = Remove and return an instance at the first in the linked list
    - = Remove and return an instance at the top in the stack



Implementation of Stack

- Python code of a stack
  - Utilizing a singly linked list
  - To pop an instance
    - 1 retrieval count
  - To push an instance
    - 1 retrieval count

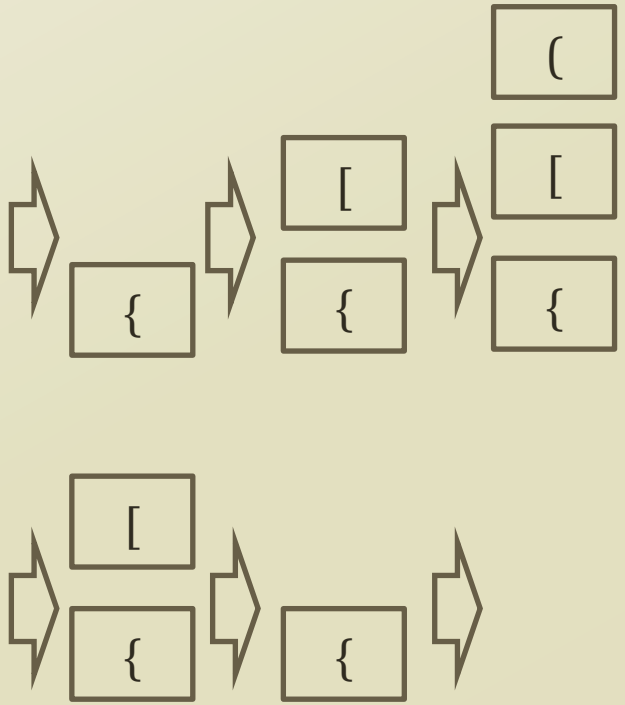
```
from src.edu.kaist.seslab.ie362.week3.SinglyLinkedList import SinglyLinkedList

class Stack(object):
    lstInstance = SinglyLinkedList()
    def pop(self):
        return self.lstInstance.removeAt(0)
    def push(self, value):
        self.lstInstance.insertAt(value, 0)


stack = Stack()
stack.push("a")
stack.push("b")
stack.push("c")

print(stack.pop())
print(stack.pop())
print(stack.pop())
```

```
c
b
a
```

Example: Balancing Symbols

- Balancing symbols?
  - [2+(1+2)]-3 → Symbols are balanced
  - [2+(1+2]-3 → Symbols are not balanced
  - Then, just counting opening and closing symbols?
    - What if? [2+(1]+2)-3
- Algorithm for the balanced symbol checking
  - Make an empty stack
  - read symbols until end of formula
    - if the symbol is an opening symbol push it onto the stack
    - if it is a closing symbol do the following
      - If the stack is empty report an error
      - Otherwise pop the stack.
        - If the symbol popped does not match the closing symbol report an error
  - At the end of the of formula if the stack is not empty report an error

7+{[2+(1+2)]*3}

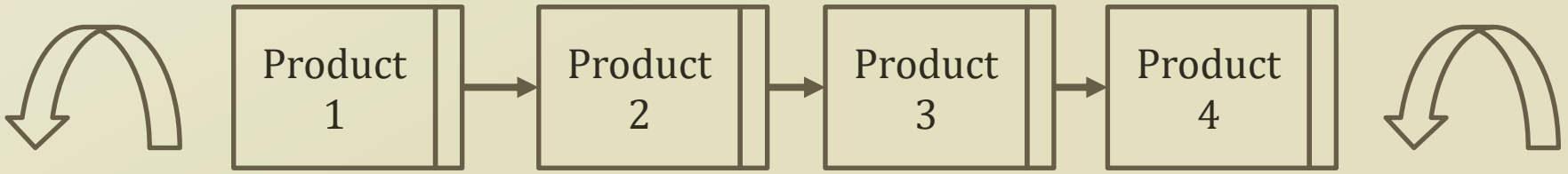


Scenario for Queue

- Line at an airport
  - A way for going in
    - At the end of the line
  - Another way for going out
    - At the first of the line
  - No one gets in the middle of the line
- A scenario of a line at the airport
  = A scenario of a production line at a factory
  - The first product out is the first product in
  - How to track the production line?

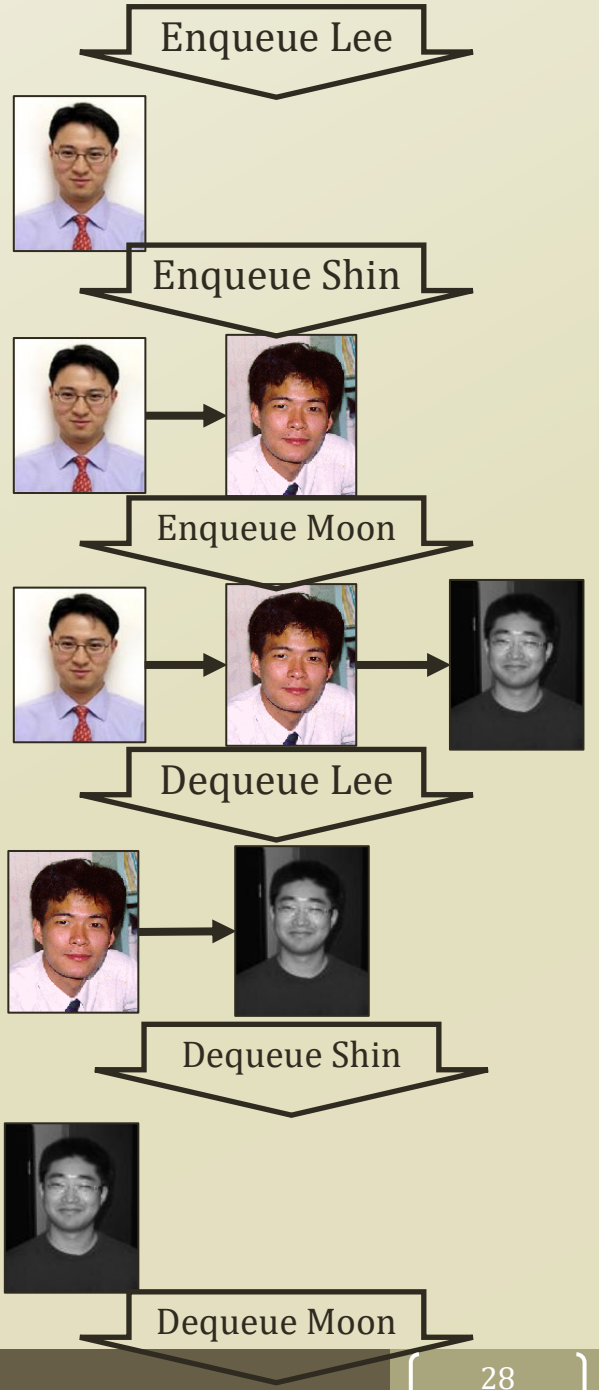
Structure of Queue

- Queues are linear like linked lists
  - A variation of a singly linked list
- Difference
  - Voluntarily giving up
    - Access to the middle in the linked list == Same to the stacks
    - Only accesses to the first and the last instances in the list
  - The first instance in the list
    = The front instance in the queue
  - The last instance in the list
    = The rear instance in the queue
- An item is inserted at the last
- An item is removed at the front
- This mechanism is called First-In-First-Out (FIFO)



Operation of Queue

- Queue operation
  - Enqueue
    - = Insert an instance at the last in the linked list
    - = Put an instance at the rear in the queue
  - Dequeue
    - = Remove and return an instance at the first in the linked list
    - = Remove and return an instance at the front in the queue



Implementation of Queue

- Python code of a queue
  - Utilizing a singly linked list
  - To enqueue an instance
    - 1 retrieval count
  - To dequeue an instance
    - 1 retrieval count

```
from src.edu.kaist.seslab.ie362.week3.SinglyLinkedList import SinglyLinkedList

class Queue(object):
    lstInstance = SinglyLinkedList()
    def dequeue(self):
        return self.lstInstance.removeAt(0)
    def enqueue(self, value):
        self.lstInstance.insertAt(value,self.lstInstance.getSize())


queue = Queue()
queue.enqueue("a")
queue.enqueue("b")
queue.enqueue("c")

print(queue.dequeue())
print(queue.dequeue())
print(queue.dequeue())
```

```
a
b
c
```