

Linked List, Stack, Queue

- What is the List
- Abstarct Data Type
- Array
- Array and List
- LinkedList
- Stack
- Queue

List

List는 왜 필요한가?
마구잡이로 엉켜있는 데이터를
선형적으로 정리하면 찾기가 쉬움

Scenario for List

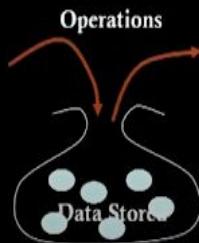
- ❖ You are looking for Koh in the mass public
 - ❖ You are going to ask one by one “Are you Ms. Koh?”
 - ❖ Sometimes, you ask the question to a person multiple times
 - ❖ You realize that this is not going to work!
 - ❖ So, you line up the people and again ask the question one-by-one
- ❖ You are looking 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 Type

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



Abstract Data Type 이란

데이터를 저장할 수 있고 데이터를 넣고 뺄 수 있으며
에러를 제어할 수 있는 자료구조를 뜻한다.

Array

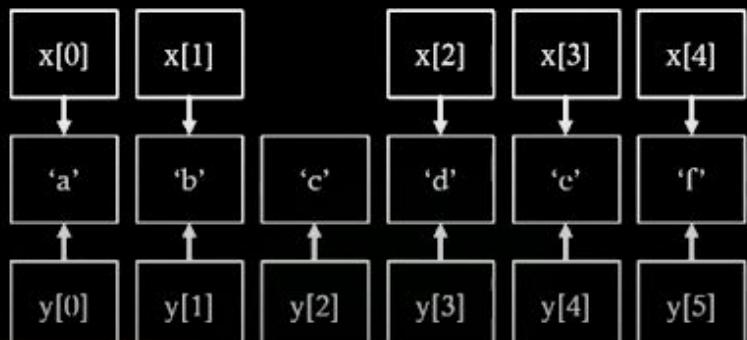
Array란?

동일한 데이터를 인덱스를 활용하여 저장할 수 있는것

다른 언어에서는 사이즈를 정하고 데이터 타입을 지정해줘야 하는데

파이썬에서는 List로 쉽게 구현할 수 있다.

Insert Procedure in array



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

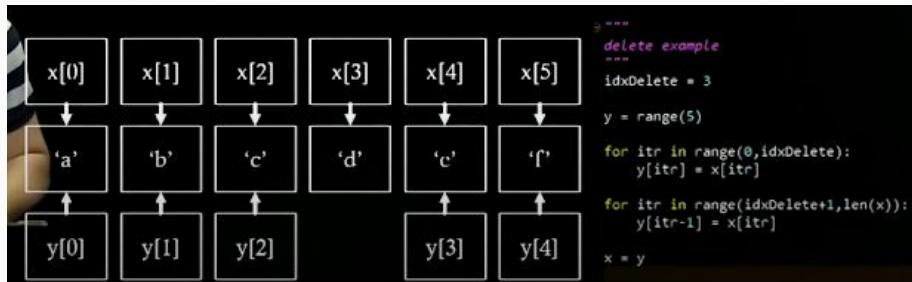
y = range(6)
for itr in range(0, idxInsert):
    y[itr] = x[itr]
y[idxInsert] = valInsert
for itr in range(idxInsert, len(x)):
    y[itr+1] = x[itr]
```

실행 수
 $a+1+n-a-1 = n$

총 n 번 만큼 실행

Array

Delete procedure in array



총 실행 수
 $a+n-a-1 = n-1$

총 $n-1$ 번 만큼 실행

문제점

N 개가 적으면 괜찮지만 숫자가 커질수록 10만, 100만이 되면 하나를 넣거나 뺄 때 그 전체를 다 실행해야 한다

해결

Linked List

Array and List

Array

고유한 식별자와 그 식별자에 대응하는 데이터의 묶음

인덱스 번호를 이용해서 빠르게 접근을 할 수 있지만 추가나 삭제행위가 실행되면 모든 데이터가 움직여야해서 용이하지 않다
(주민등록번호 처럼 앞 사람이 죽는다고 내 주민등록번호가 변하거나 사라지지 않음)

List

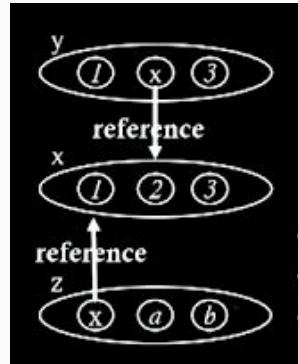
같은 값이 한 번 이상 존재할 수 있는 일련의 값이 모여있는 추상적 자료형

List는 pointer라는 개념이 있는데 하나의 메모리에는 pointer도 같이 존재하고 그 pointer는 다음 메모리의 위치를

가르키고 있다. 추가나 삭제할 경우 필요한 메모리만 건들기 때문에 동작속도가 빠름

	Read	Write/Update/Delete
Array :	빠름	느림
List :	느림	빠름

Linked List



Reference 를 이해하고 있어야지 Linked List를 사용할 수 있다.

Singly Linked List

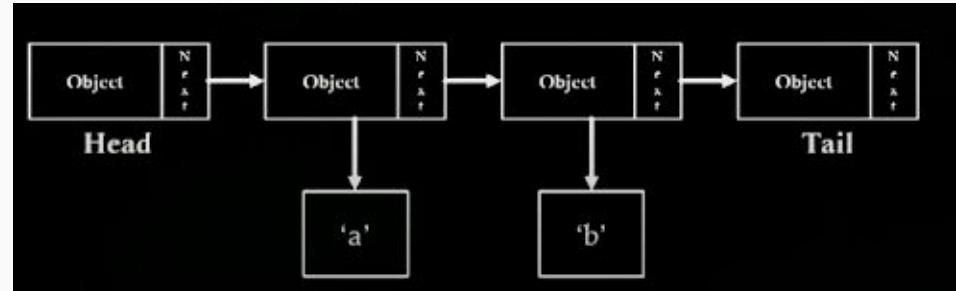
node:

두가지 Variable로 이루어져 있다.

1. next node
2. value

special node = head and tail

값은 없고 존재하기만 함

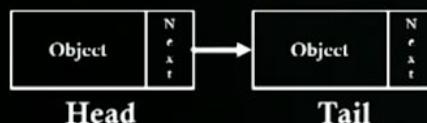


LinkedList

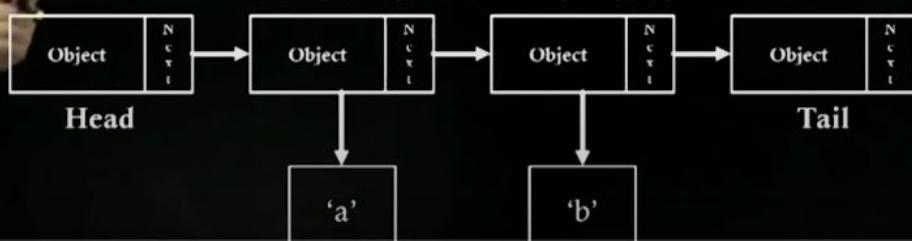
Head and Tail

- ❖ 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?

Empty Linked List



Linked List with Two Nodes



Head와 Tail의 Object부분은
비어있고 위치요소만
가지고있다

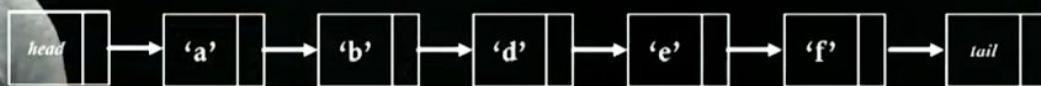
리스트의 시작과 끝만
알려주는 노드

Linked List

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

node.objValue == 'c' node.objValue == 'c' node.objValue == 'c' node.objValue == 'c' node.objValue == 'c'
→ False: No → False: No → False: No → False: No → False: No
hit! hit! hit! hit! hit!



node.objValue == 'd'
→ False: No
hit!
node.objValue == 'd'
→ False: No
hit!
node.objValue == 'd'
→ True: Hit!

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

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

1. find head from list

2. next node

```
if next == tail  
    break  
elif next != tail  
    find objValue
```

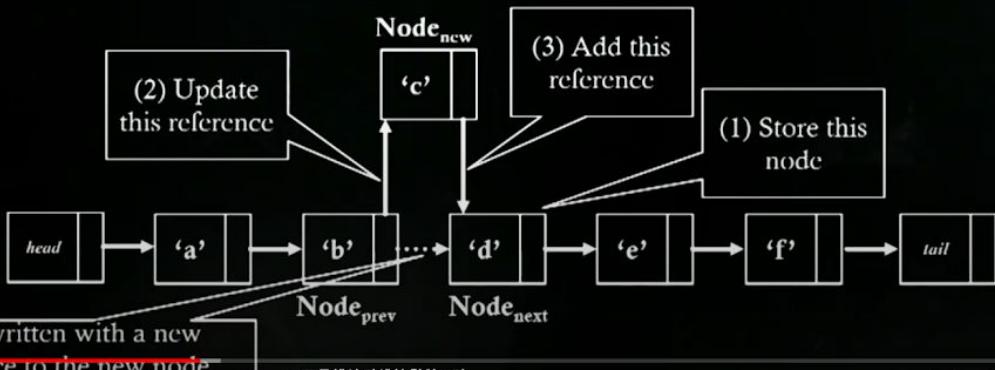
n번 검사를 해야 알 수 있음

Linked List

2 | Lecture 21 | 데이터 구조, 알고리즘

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, $\text{Node}_{\text{prev}}$ that you want to put your new node next
 - ◊ First, you store a Node, or a $\text{Node}_{\text{next}}$, pointed by a reference from $\text{Node}_{\text{prev}}$'s nodeNext member variable
 - ◊ Second, you change a reference from $\text{Node}_{\text{prev}}$'s nodeNext to Node_{new}
 - ◊ Third, you change a reference from Node_{new} 's nodeNext to $\text{Node}_{\text{next}}$



1. $\text{Node}_{\text{prev}}$ 의 next 를 끊어버리고 Node_{new} 를 가르키게 만든다

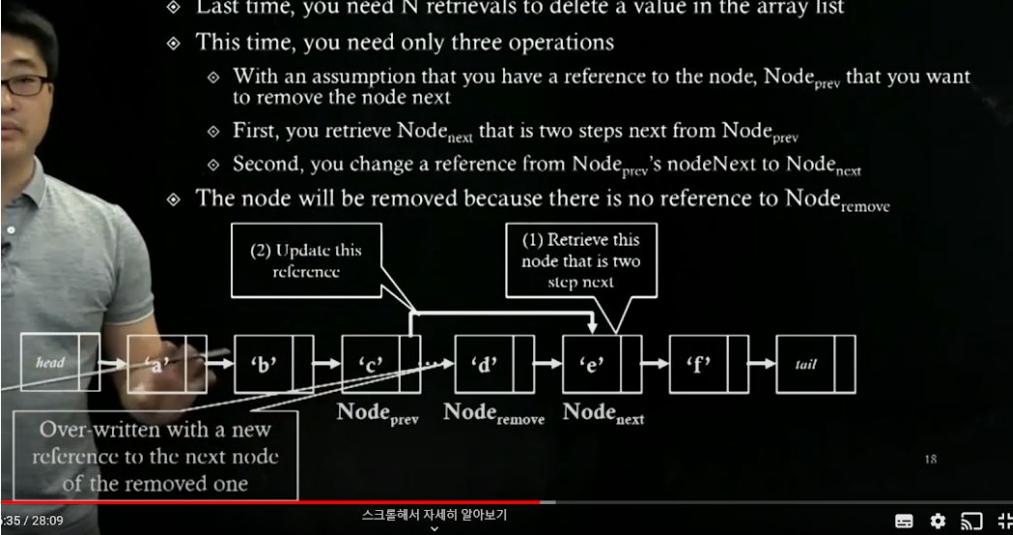
2. $\text{Node}_{\text{prev}}$ 가 들고 있던 next 를 Node_{new} 의 next 에 집어 넣는다

움직인건 3개의 노드뿐

Linked List

Delete procedure in singly linked list

- ◇ This is 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, $\text{Node}_{\text{prev}}$ that you want to remove the node next
 - ◇ First, you retrieve $\text{Node}_{\text{next}}$ that is two steps next from $\text{Node}_{\text{prev}}$
 - ◇ Second, you change a reference from $\text{Node}_{\text{prev}}$'s `nodeNext` to $\text{Node}_{\text{next}}$
- ◇ The node will be removed because there is no reference to $\text{Node}_{\text{remove}}$



1. Node Remove = $\text{Node}_{\text{prev}}.\text{next}$
2. Node next = $\text{Node}_{\text{prev}}.\text{next}.\text{next}$

Node remove를 가르키는 노드가
없어지면서 사라진것처럼 보이게 되고
Garbage collector가 이런 메모리구조를
찾아서 메모리를 지움

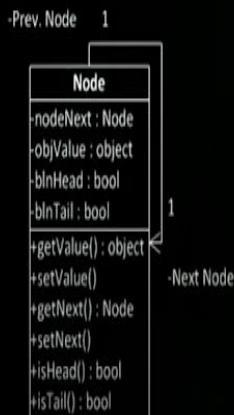
사용된 노드는 3개의 노드 뿐

LinkedList Class

LinkedList Class->

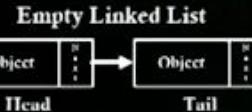
node 클래스

```
class Node:  
    def __init__(self, objValue = "", nodeNext = "", binHead = False, binTail = False):  
        self.nodeNext = nodeNext  
        self.objValue = objValue  
        self.binHead = binHead  
        self.binTail = binTail  
  
    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.binHead  
  
    def isTail(self):  
        return self.binTail  
  
node1 = Node(objValue = 'a')  
nodeTail = Node(binTail = True)  
nodeHead = Node(binHead = True, nodeNext = node1)
```



```
class SinglyLinkedList:  
    nodeHead = ""  
    nodeTail = ""  
    size = 0  
  
    def __init__(self):  
        self.nodeTail = Node(binTail = True)  
        self.nodeHead = Node(binHead = 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(),  
            print  
  
    def getSize(self):  
        return self.size
```

Implementation of Singly linked list



```
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

Scenario for Stack

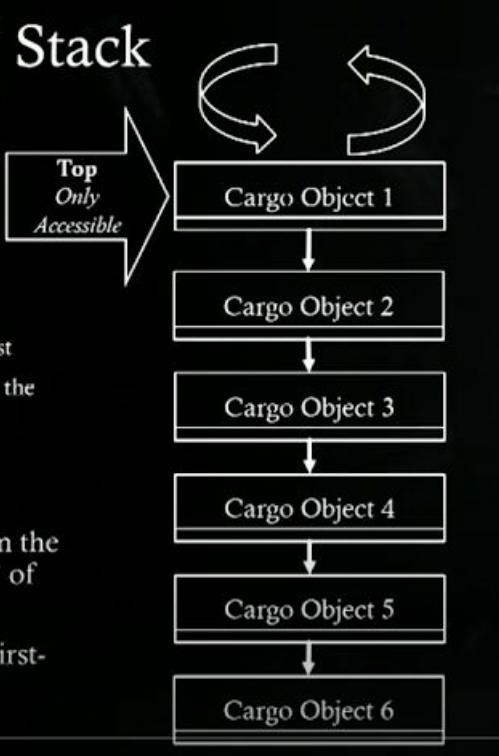
- ◊ Back seat of a taxi
 - ◊ One way in and out
 - ◊ We are traveling from the source to our destinations
 - ◊ Who should seat first and last?
- ◊ A scenario for a taxi
= A scenario for SCM
 - ◊ A cargo plane with one way in and out
 - ◊ How to organize the cargo loading to the plane?



Stack

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).



선형적이다

오직 첫 노드에 대해서만 관여함

LIFO
(Last in First out)

Stack

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



Top만 알 수 있다

Push
Top에 넣기

Pop
Top 빼기

Stack Class

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 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

SinglyLinkedList를 활용함

0번째만 건드림

Example

Example: Balancing Symbols

- ◊ Balancing symbols?
 - ◊ $[2+(1+2)]-3 \rightarrow$ Symbols are balanced
 - ◊ $[2+(1+2]-3 \rightarrow$ 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

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



Queue

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?



FIFO
(First In First Out)
터널

Queue Structure

Structure of Queue

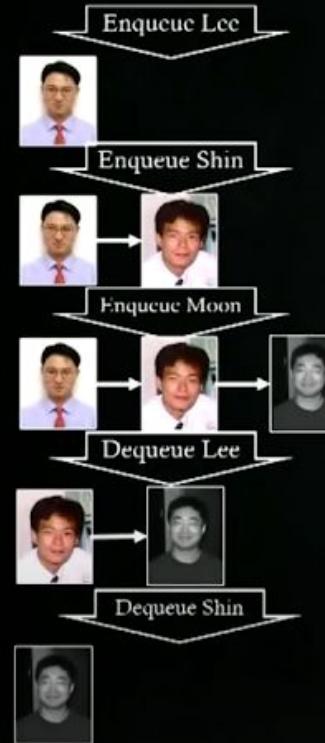
- ◊ Queues are linear like linked lists
 - ◊ A variation of a singly linked list
- ◊ Difference
 - ◊ Voluntarily giving up
 - ◊ Access to the middle in the linked list == Same to the stacks
 - ◊ Only accesses to the first and the last instances in the list
 - ◊ The first instance in the list
 - = The 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 Fast-In-First-Out (FIFO)



Operation of Queue

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



Queue Class

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 edu.kaist.seslab.ie362.week3.SinglyLinkedList import SinglyLinkedList

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

queue = Queue()
queue.enqueue("a")
queue.enqueue("b")
queue.enqueue("c")                                || a
                                                || b
                                                || c
print queue.dequeue()
print queue.dequeue()
print queue.dequeue()
```