Data Structure

Giftbot

Good or Bad Programmers

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.

Bad programmers worry about the code.

Good programmers worry about data structures and their relationships.

Abstract Data Type

추상 자료형 : 어떤 문제를 해결하기 위한 자료 형태와 그것을 다루는 연산을 수학적으로 정의한 모델

데이터와 연산 두 가지 부분에 대한 명세서(Specification)로 구성

- 인터페이스와 구현을 분리하여 추상화 계층(인터페이스)을 둔 것
- 내부 구현을 알지 못해도 ADT 명세만 알면 활용 가능

  예) TV 리모콘을 누를 때 내부 구현을 알지 못해도 버튼만 누르면 동작 - powerOn(), volumeUp()
- 관련 자료형의 기본 특성은 가지지만 정해진 표준은 없고 정의하는 회사나 사람에 따라 다를 수 있음
- 연산에 대한 복잡도나 구현 내용은 정의하지 않음

※ 추상 자료 구조 : 추상 자료형에 각 연산에 대한 복잡도까지 정의한 가상의 자료 구조

Data Structure

자료를 효율적으로 이용할 수 있도록 저장하는 방법을 의미

1차원 형태의 메모리 공간과 현실 세계의 다차원 데이터를 어떻게 변환할 것인지 다루는 일이기도 함

추상자료형에서 정의한 내용을 실제로 구체화한 형태

> 추상화 - 무엇(What) 을 할 것인가

> 구체화 - 어떻게(How) 할 것인가

잘 짜여진 자료구조는 적은 메모리 용량과 연산 시간을 갖게 되므로 효과적인 알고리즘 구현에 중요한 역할

주요 관점 : 검색, 삽입, 변경, 삭제

Data Structure

회사에서 각종 문서를 다루는 업무를 가진 A 와 B 라는 사원이 있다.

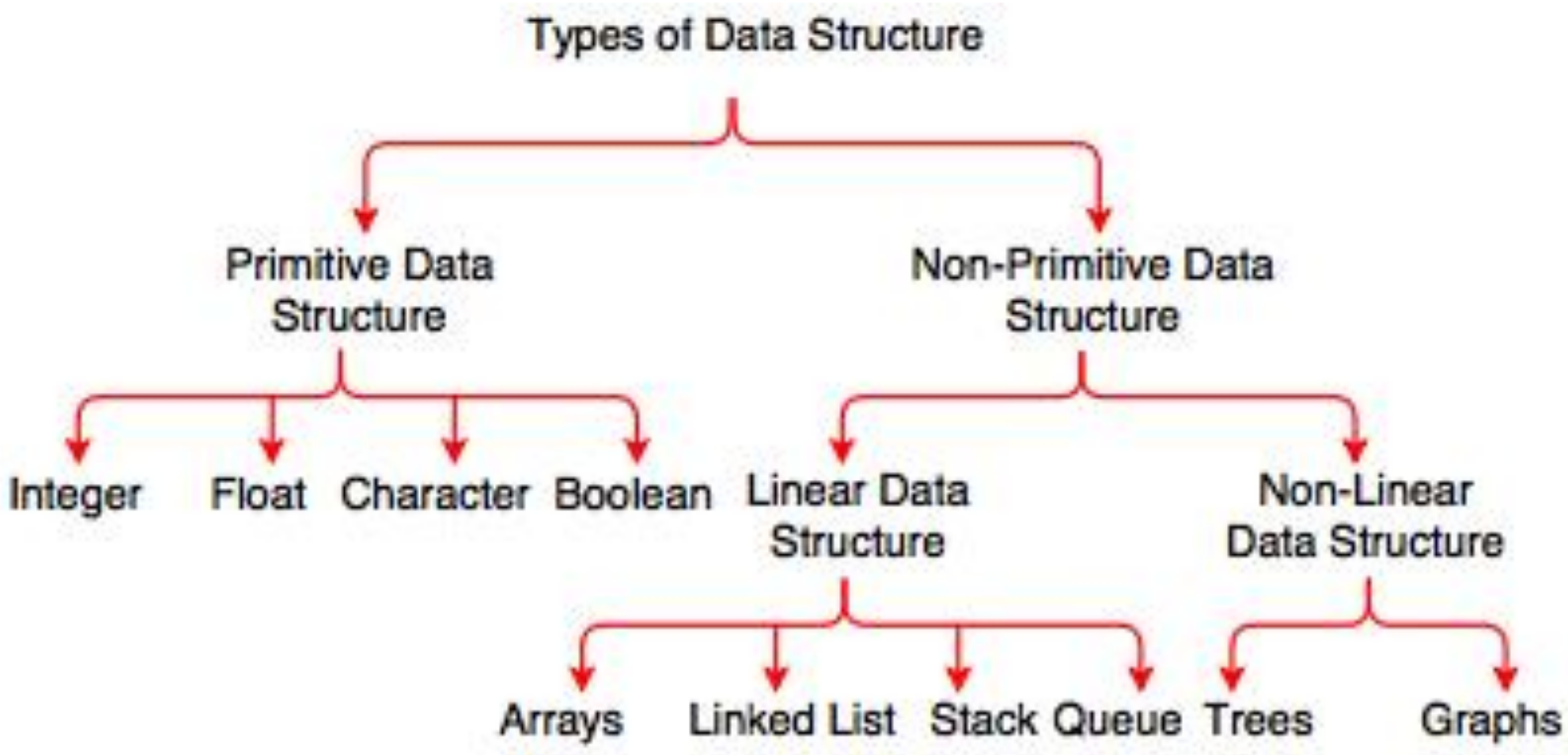
A 사원은 매번 문서가 들어올 때마다 여기저기 불려놓았던 까닭에 필요한 문서를 찾을 때마다 전체 문서를 다시 살펴보느라 오랜 시간을 헤매야만 했고 B 사원은 깔끔하게 카테고리별로 정리하고 일자를 구분하여 정리해 둔 까닭에 필요한 문서를 쉽게 정리하고 찾아 쓸 수 있었다.

- A 와 B 가 문서를 관리하고 검색하는 방식에 따른 효율이 바로 자료 구조가 필요한 이유
- 자료를 어떤 구조로 분류하고 정리해야 효율적인지 고민하고 최적의 방안을 선택
- 자료의 규모가 크면 클수록 자료 구조에 따른 구현 난이도나 연산 속도에 큰 영향

[좋은 자료 구조]

해결하고자 하는 문제와 관련된 자료의 추가, 삭제, 검색이 효율적으로 수행되고 복잡한 구조를 간결하게 표현할 수 있는 형태. 따라서 무조건 더 좋은 자료 구조는 없으며 구현 내용에 따라 적절한 데이터 구조가 다를 수 있다.

Types of Data Structure



Principles

자료 구조가 가져야 하는 특징

- **정확성(Correctness) - 필요한 자료에 필요한 연산을 정확히 적용 할 수 있어야 함**
- **효율성(Efficiency) - 상황에 맞는 구조를 사용하여 자료 처리의 효율성 상승**
- **추상화(Abstraction) - 복잡한 자료의 핵심 개념 또는 기능을 추상화하여 간단하고 쉽게 사용할 수 있도록 설계**
- **재사용성(Reusability) - 추상화된 개념을 모듈화하여 독립적이고 쉽게 재사용 가능하도록 함**

자료 구조 선택 기준

- **자료의 크기와 처리시간**
- **자료의 활용 및 갱신 빈도**
- **활용 용이성**

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out acces	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)*	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)* *(Similar trees good for disk storage)*	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex

Complexity

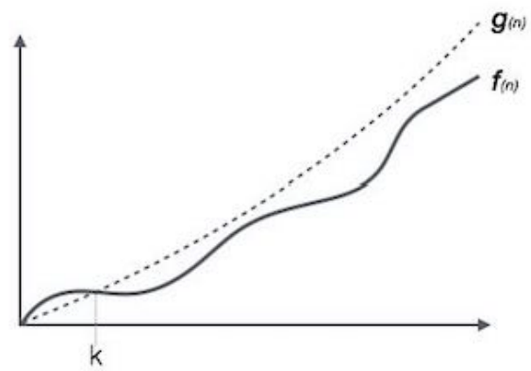
시간복잡성 - 데이터 연산 시간은 가능한 작아야 함

공간복잡성 - 데이터 연산 및 저장에 필요한 메모리 공간은 가능한 작아야 함

Worst Case

: 빅-오 (O, Big-Oh) 표기법.

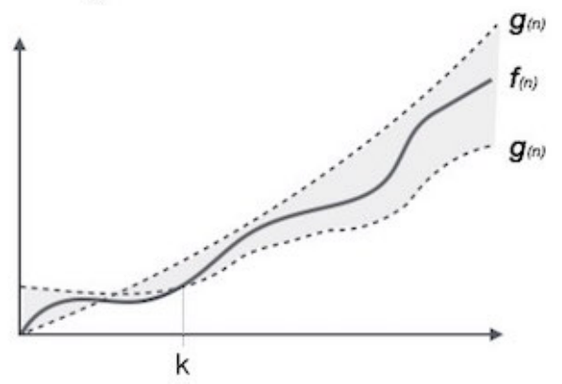
: 최악의 경우를 가정. 시간 복잡도를 말할 때 가장 일반적으로 사용



Average Case

: 세타 (Θ, Theta) 표기법

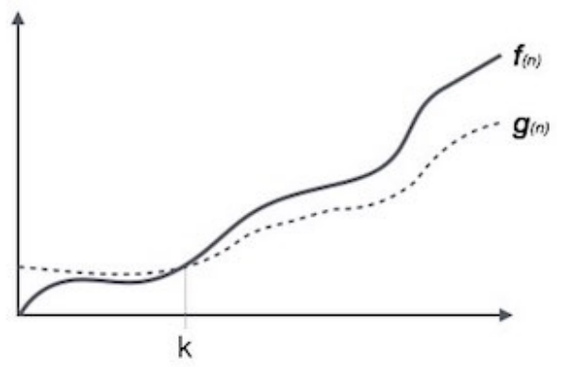
: 평균적인 경우를 가정



Best Case

: 빅-오메가 (Ω, Big-Omega) 표기법

: 최선의 경우를 가정



Common Asymptotic Notations

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Complexity

O(1)

```
[1,2,3].first
[1,2,3].last
[1].isEmpty
10 % 2
```

O(n)

```
for _ in 1...10 {
  count += 1
}
```

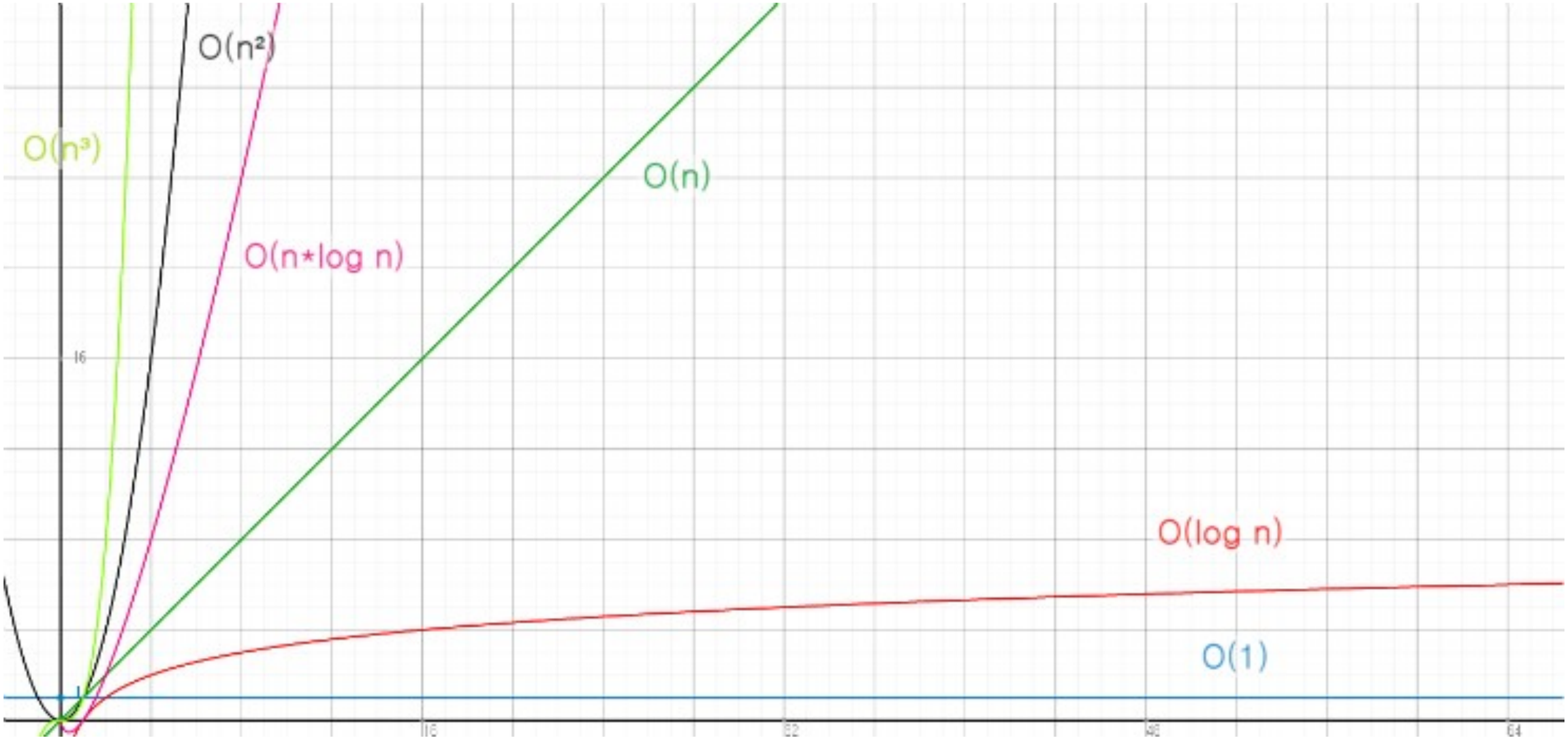
O(log n)

```
var num = 100
while num != 0 {
  num /= 2
  count += 1
}
```

O(n^2)

```
for _ in 1...10 {
  for _ in 1...10 {
    count += 1
  }
}
```

Complexity



Linked List

Linked List

Single Linked List (단일 연결 리스트)

: 하나의 포인터 멤버로 다른 노드 데이터를 가르키는 것 (HEAD 없이 TAIL 만 존재)

: 앞으로 돌아갈 수 없으며, 중간 위치로 바로 접근하지 못 함. O(N)

: Head 노드 주소를 잃어버리면 데이터 전체 접근 불가. 중간이 유실되면 그 이후 노드들에 접근 불가

: Queue 구현에서 많이 사용. 파일 시스템 중 FAT 시스템이 이런 형태로 연결. 랜덤 엑세스 성능이 낮고 불안정

Double Linked List (이중 연결 리스트)

: HEAD 가 이전, TAIL 이 이후 노드 데이터를 가르키는 것

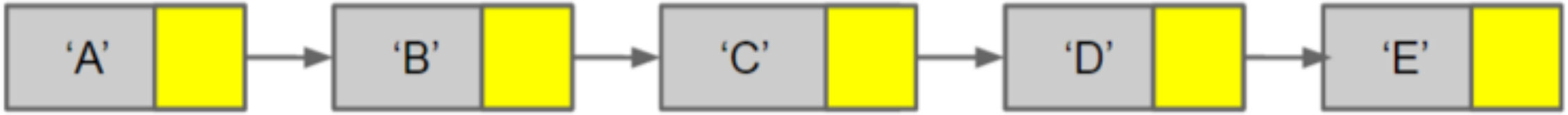
: 끊어진 체인 복구 가능

Circular Linked List (환형 연결 리스트)

: 처음 노드와 마지막 노드가 서로 연결된 구조

: 스트림 버퍼의 구현에 많이 사용되며 할당된 메모리 공간 삭제 및 재할당의 부담이 없어서 큐 구현에도 적합

Single Linked List



Single Linked List ADT Example

Insertion – Adds an element at the beginning of the list.

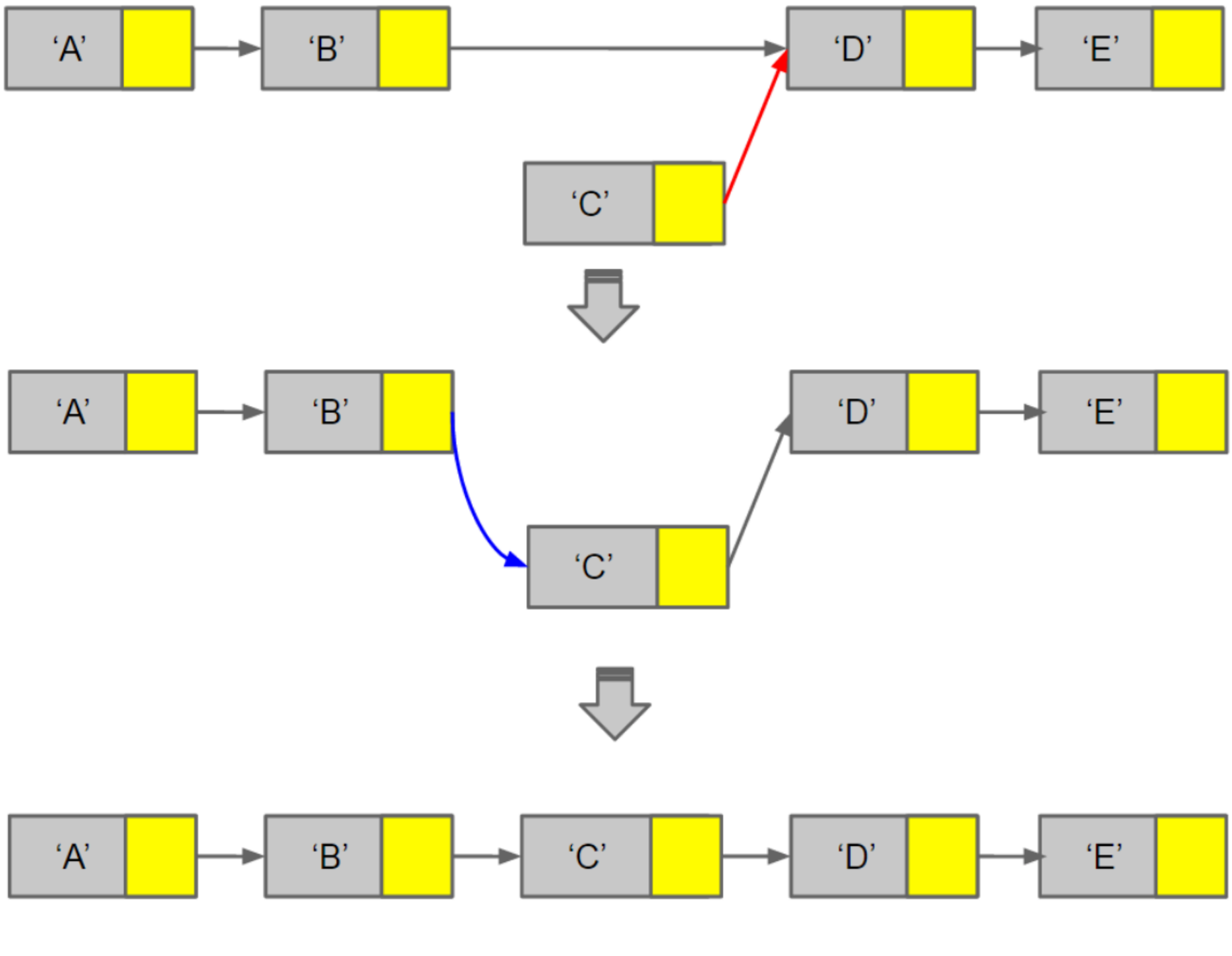
Deletion – Deletes an element at the beginning of the list.

Display – Displays the complete list.

Search – Searches an element using the given key.

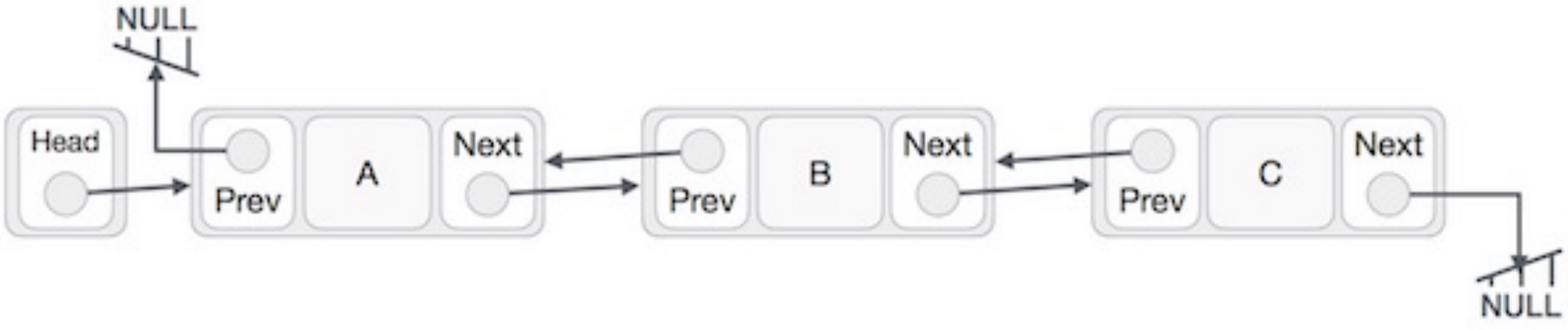
Delete – Deletes an element using the given key.

Single Linked List - Insert Node



Double Linked List

Double Linked List



Double Linked List ADT Example

Insertion – Adds an element at the beginning of the list.

Deletion – Deletes an element at the beginning of the list.

Search – Searches an element using the given key.

Insert Last – Adds an element at the end of the list.

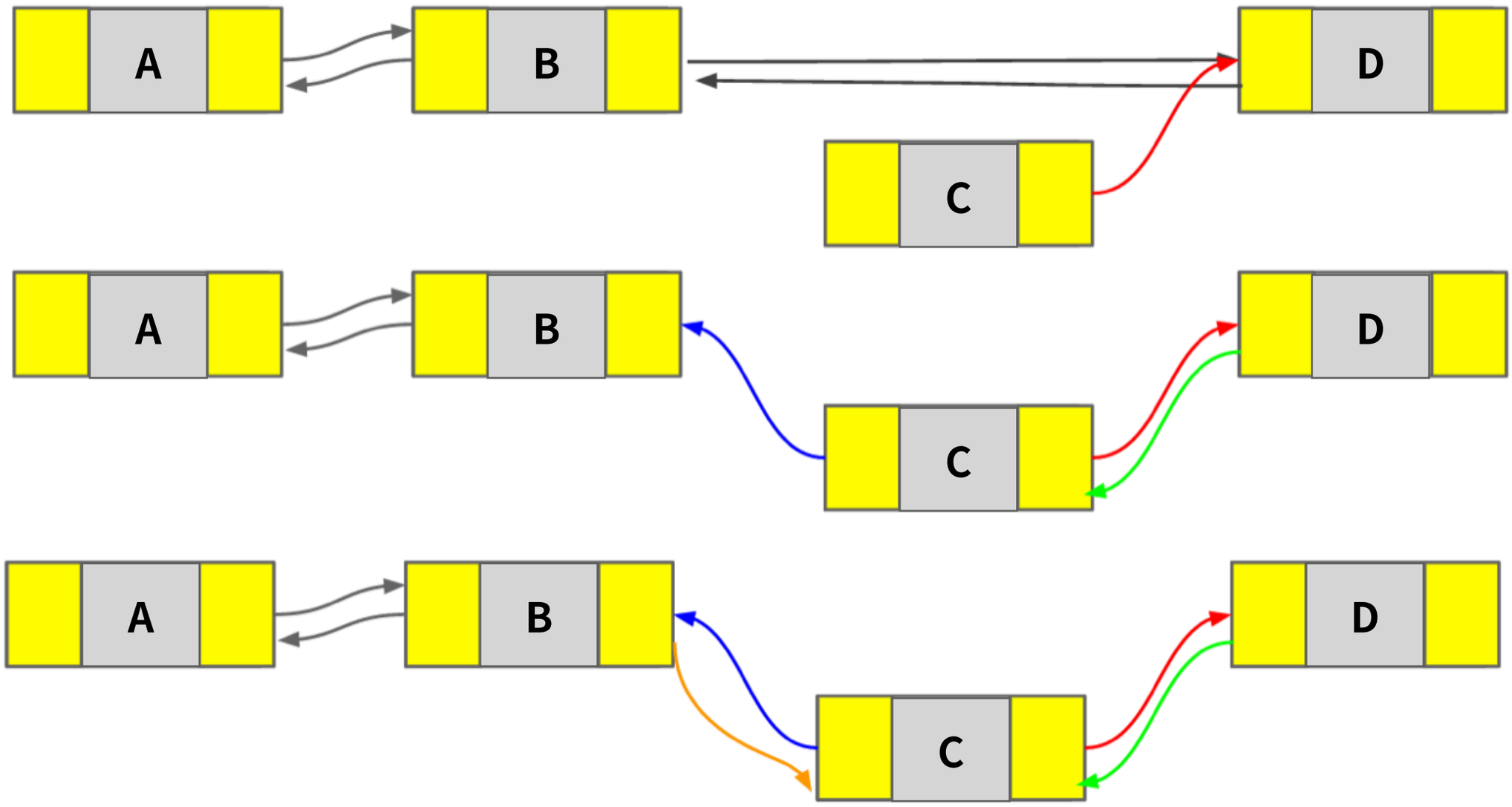
Insert After – Adds an element after an item of the list.

Delete Last – Deletes an element from the end of the list.

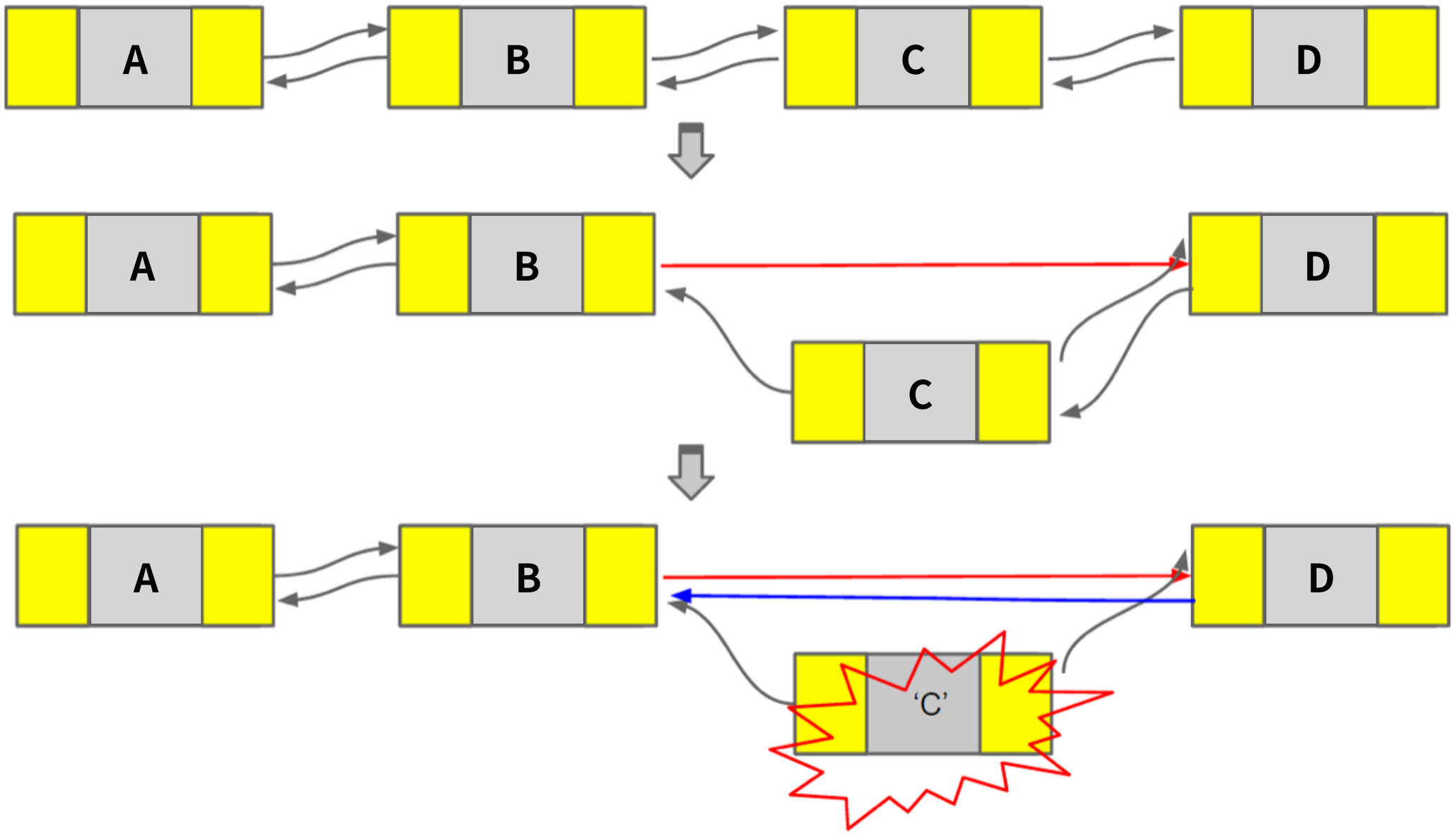
Display forward – Displays the complete list in a forward manner.

Display backward – Displays the complete list in a backward manner.

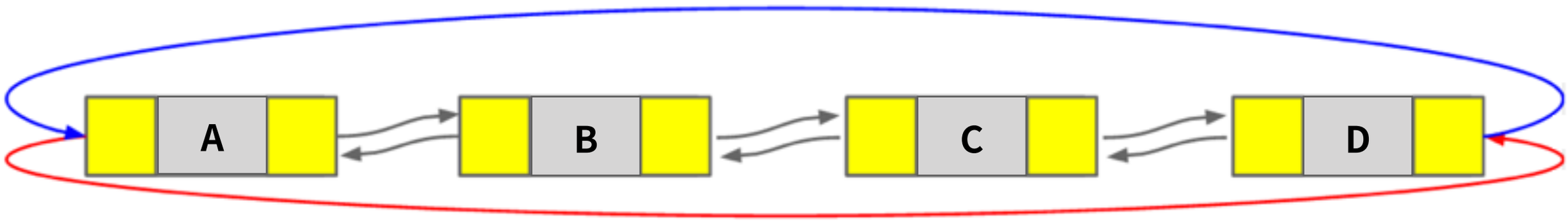
Double Linked List - Insert Node



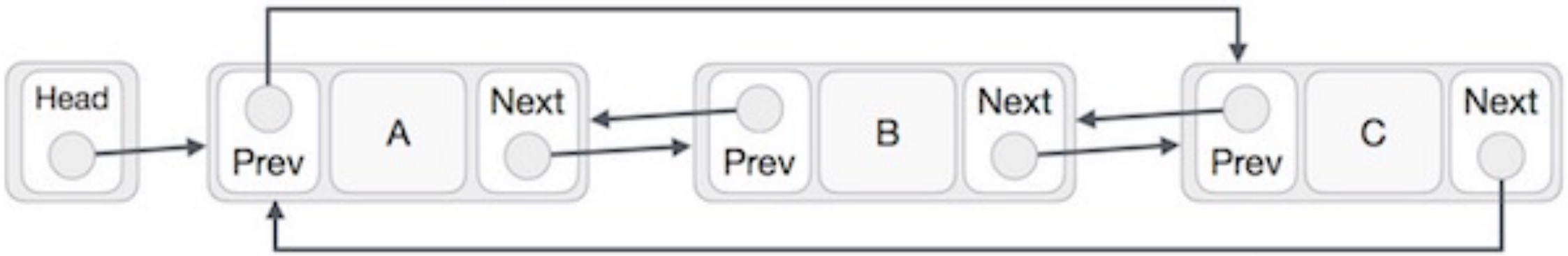
Double Linked List - Delete Node



Circular Linked List



Circular Linked List



Stack

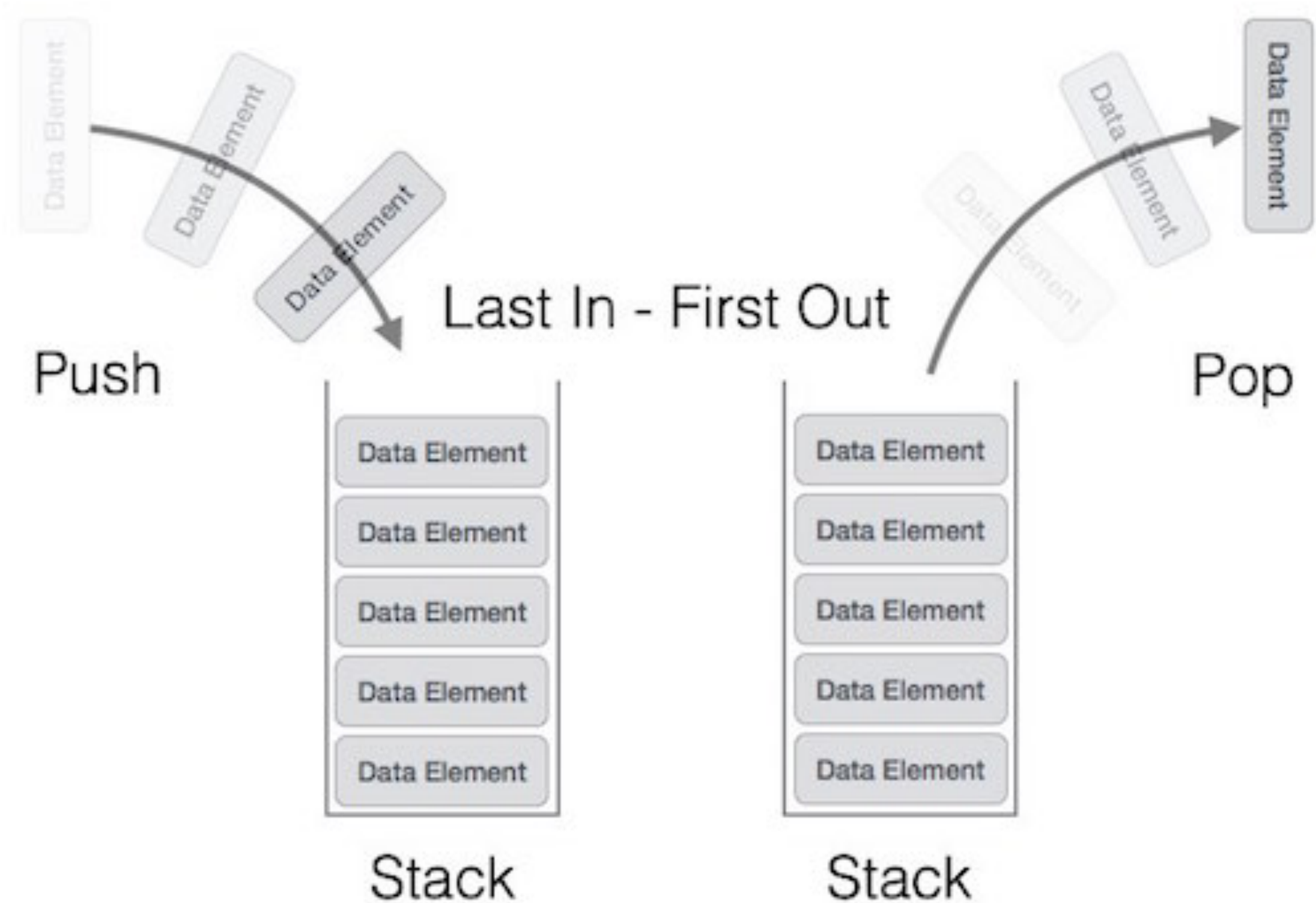
Stack

LIFO(Last In First Out) , FILO(First In Last Out)

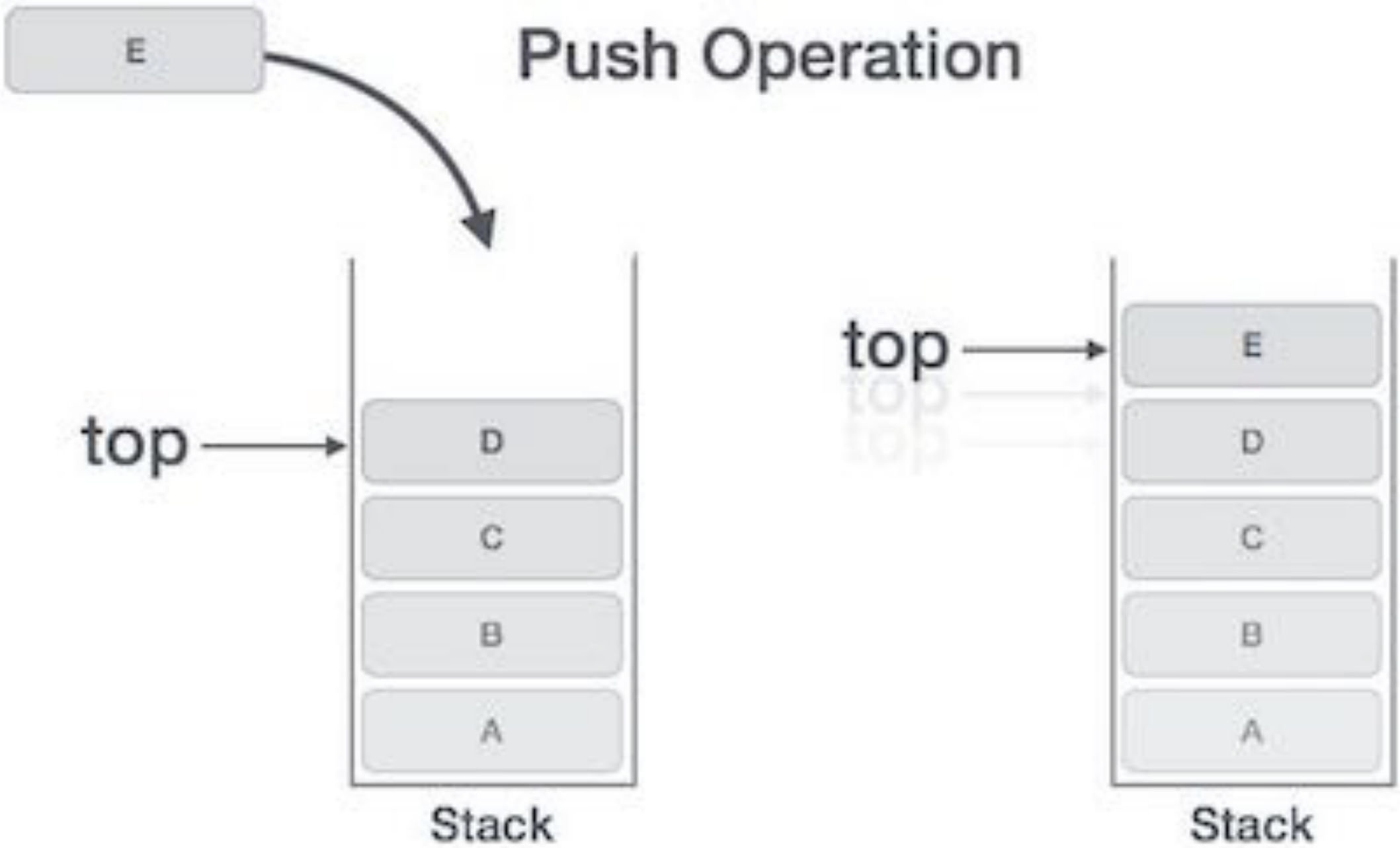
프링글스, 탄창, 대형마트 카트, 떡꼬치 등

웹 브라우저 히스토리 (뒤로 가기), 실행 취소 (Undo/Redo) 등

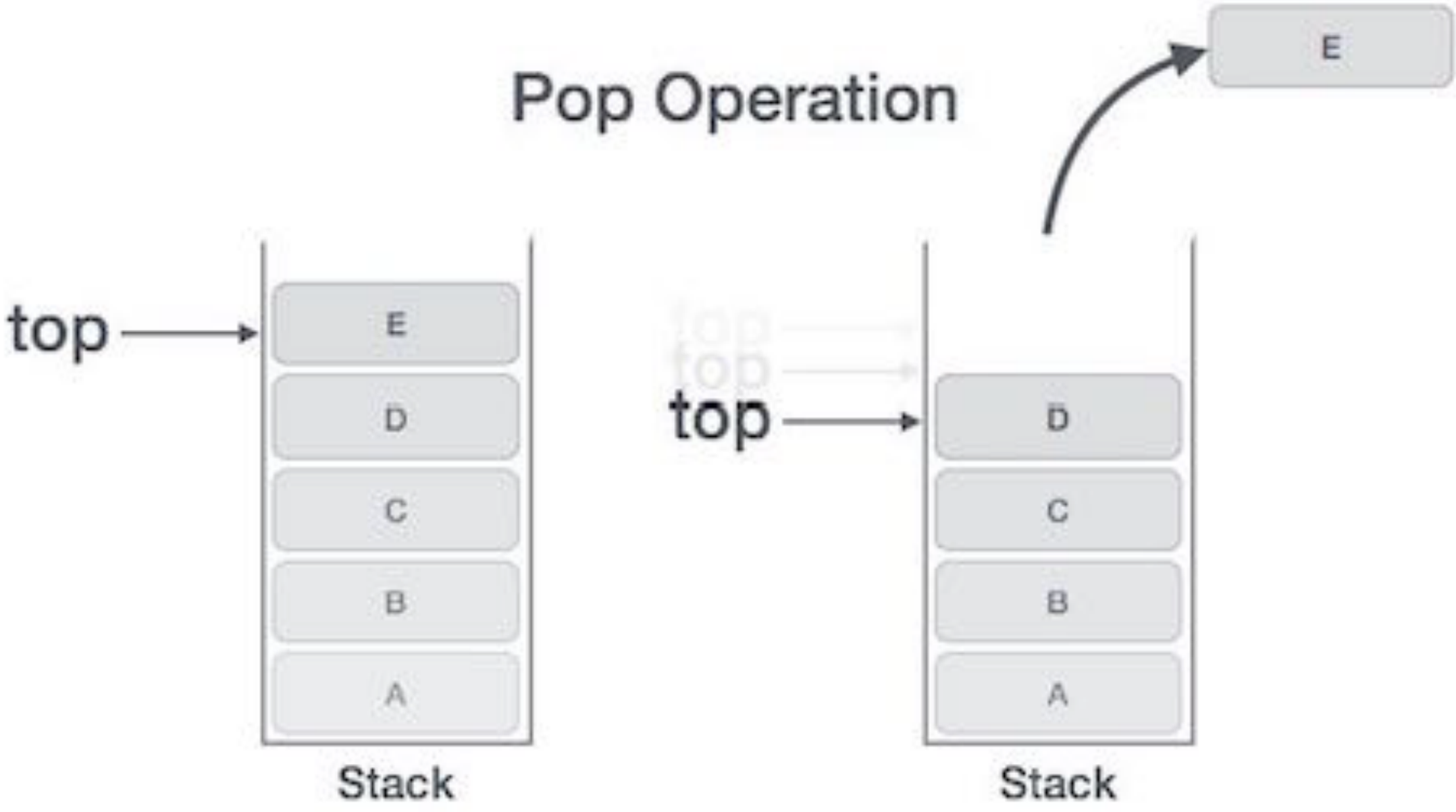
Stack


Data Element
Data Element
Data Element
Last In - First Out
Push
Pop
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Stack
Stack

Push



Pop



Array vs Linked List

배열을 이용한 구현 : 구조가 간단하고 빠름

링크드 리스트를 이용한 구현 : 구조 및 사이즈 유연성 , 데이터 타입 변형 가능

```
var stack: [Int] = []
stack.append(1)
stack.append(2)
stack.append(3)
stack.popLast()
stack.popLast()
stack.popLast()
```

```
var stack: [Any] = []
stack.append(1)
stack.append("2")
stack.append(3.0)
stack.popLast()
stack.popLast()
stack.popLast()
```

Stack ADT Example

push - Stack 에 데이터 추가

pop - Stack 에 마지막으로 추가된 데이터를 꺼내는 동시에 Stack 에서 삭제

peek - Stack 에 마지막으로 추가된 데이터를 확인만 하고 삭제는 하지 않음

isEmpty - Stack 에 데이터가 있는지 확인

size - Stack 에 들어가 있는 데이터 개수 확인

Queue

Queue

FIFO (First In First Out) , LILO (Last In Last Out)

은행 대기열, 컨테이너 벨트 등

DispatchQueue, NotificationQueue 등

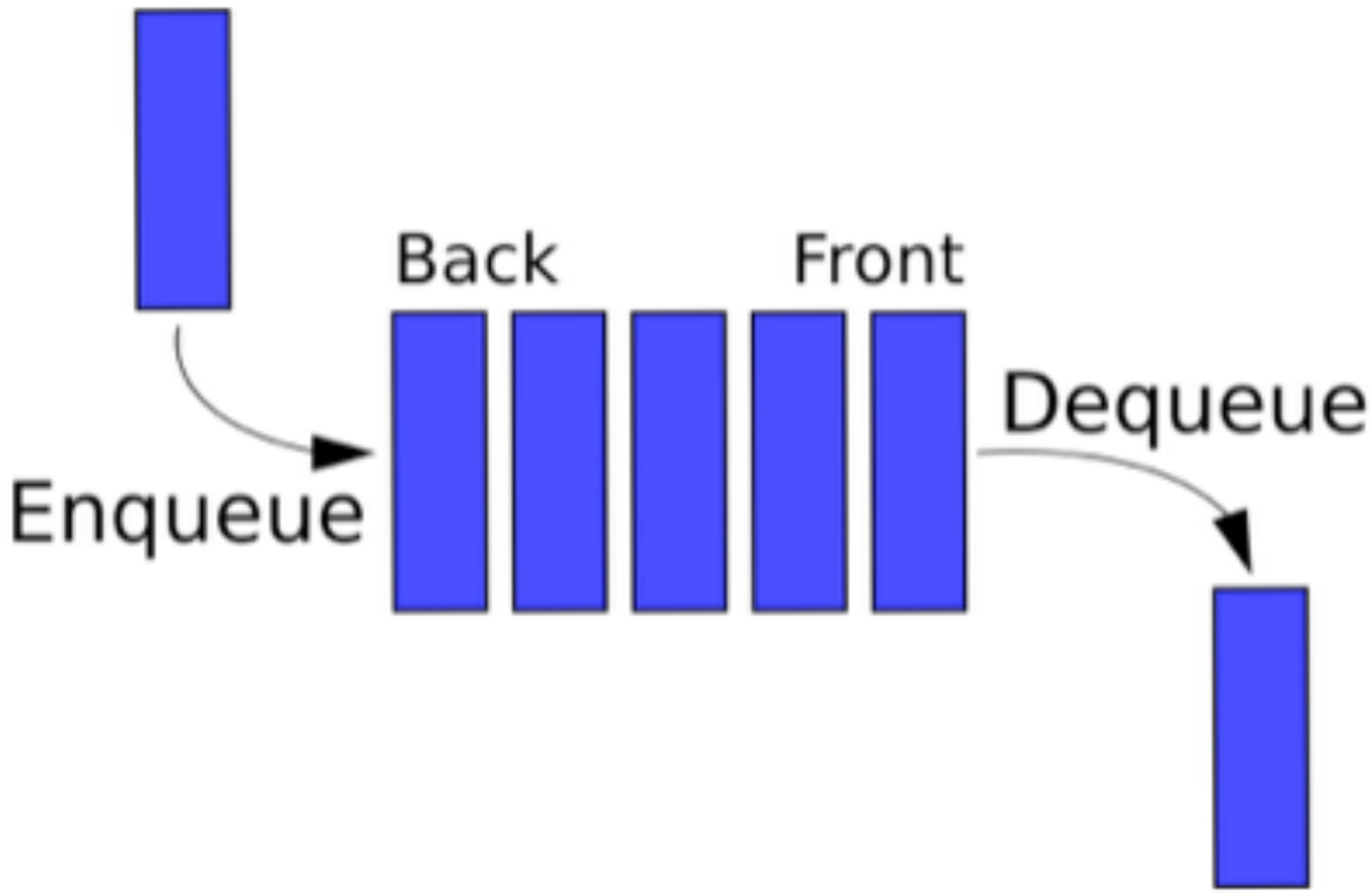
Queue

값을 추가하는 것 : Enqueue 또는 Put

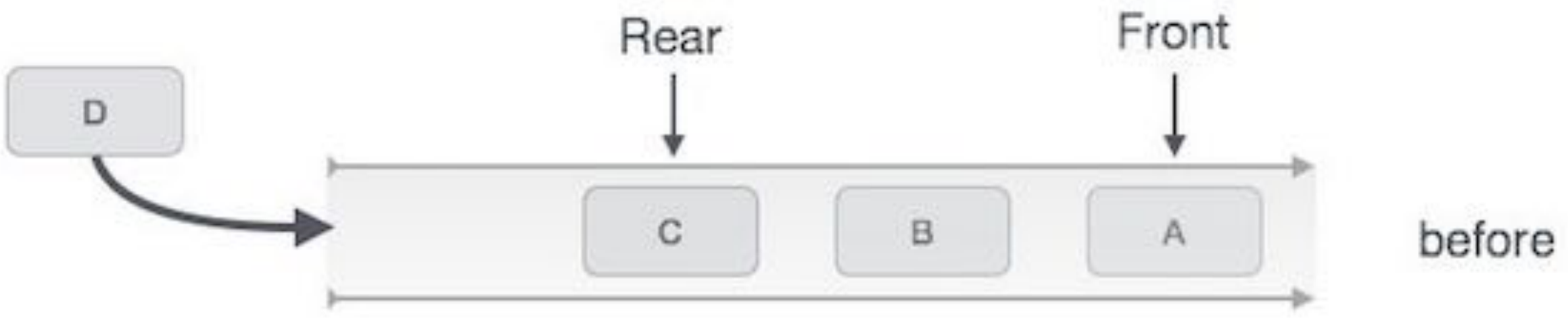
값을 꺼내오는 것 : Dequeue 또는 Get

Front : Dequeue 에 사용될 인덱스

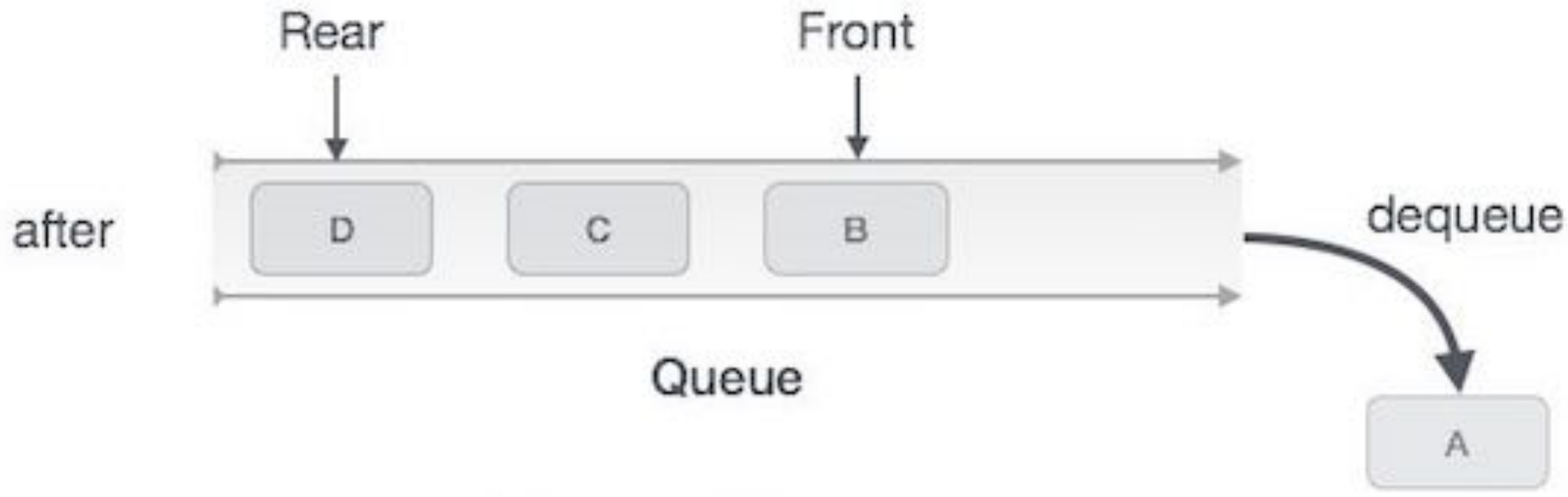
Rear (Back) : Enqueue 에 사용될 인덱스



Enqueue



Dequeue



Queue ADT Example

front(head) - Dequeue 에 사용될 위치를 가르키는 인덱스

rear(tail) - Enqueue 에 사용될 위치를 가르키는 인덱스

enqueue - Rear 위치에 새로운 데이터를 삽입

dequeue - Front 에서 데이터를 꺼내는 동시에 Queue 에서 삭제

peek - Front 에 위치한 데이터를 확인만 하고 삭제는 하지 않음

isEmpty - Queue 에 데이터가 있는지 확인

size - Queue 에 들어가 있는 데이터 개수 확인

Deque (Double Ended Queue)

front 와 rear 구분 없이 양 쪽 모두에서 Enqueue / Dequeue 를 가능하도록 한 구조