

# Data Structure

# 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

자료를 효율적으로 이용할 수 있도록 저장하는 방법을 의미

1차원 형태로 사용되는 메모리 공간과 현실 세계의 다차원 데이터를 어떻게 변환할 것인지를 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

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.

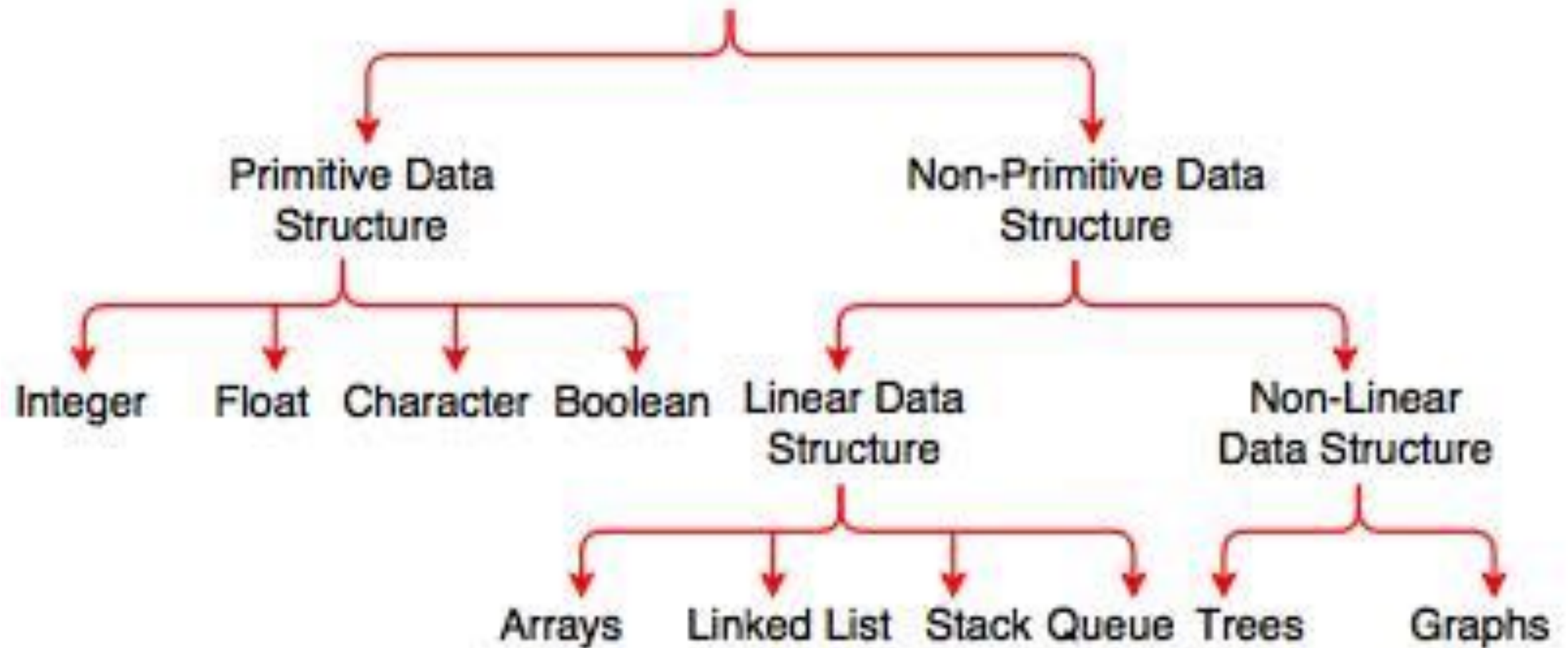
- A 와 B 가 문서를 관리하고 검색하는 방식에 따른 효율이 바로 자료 구조가 필요한 이유
- 자료를 어떤 구조로 분류하고 정리해야 효율적인지 고민하고 최적의 방안을 선택
- 자료의 규모가 크면 클수록 자료 구조에 따른 구현 난이도나 연산 속도에 큰 영향

## [ 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

## Types of Data Structure



## 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
<b>Array</b>	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
<b>Ordered Array</b>	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
<b>Stack</b>	Last-in, first-out acces	Slow access to other items
<b>Queue</b>	First-in, first-out access	Slow access to other items
<b>Linked List</b>	Quick inserts Quick deletes	Slow search
<b>Binary Tree</b>	Quick search Quick inserts Quick deletes <i>(If the tree remains balanced)</i>	Deletion algorithm is complex
<b>Red-Black Tree</b>	Quick search Quick inserts Quick deletes <i>(Tree always remains balanced)</i>	Complex to implement
<b>2-3-4 Tree</b>	Quick search Quick inserts Quick deletes <i>(Tree always remains balanced)</i> <i>(Similar trees good for disk storage)</i>	Complex to implement
<b>Hash Table</b>	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
<b>Heap</b>	Quick inserts Quick deletes Access to largest item	Slow access to other items
<b>Graph</b>	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

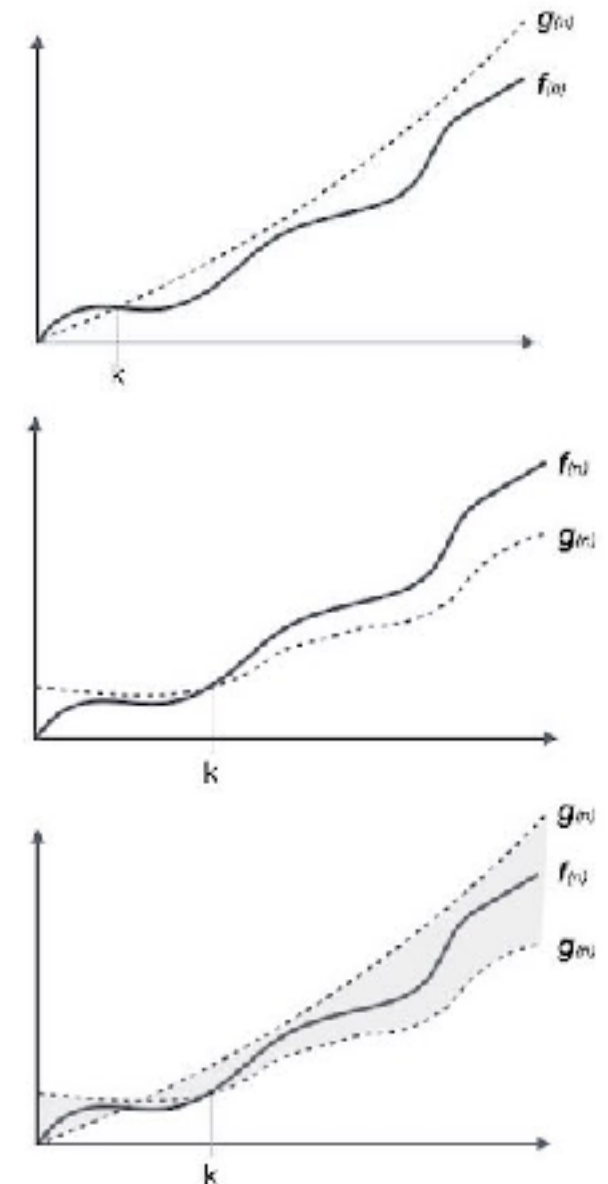
: 세타 ( $\Theta$ , Theta) 표기법

: 평균적인 경우를 가정

## Best Case

: 빅-오메가 ( $\Omega$ , Big-Omega) 표기법

: 최선의 경우를 가정

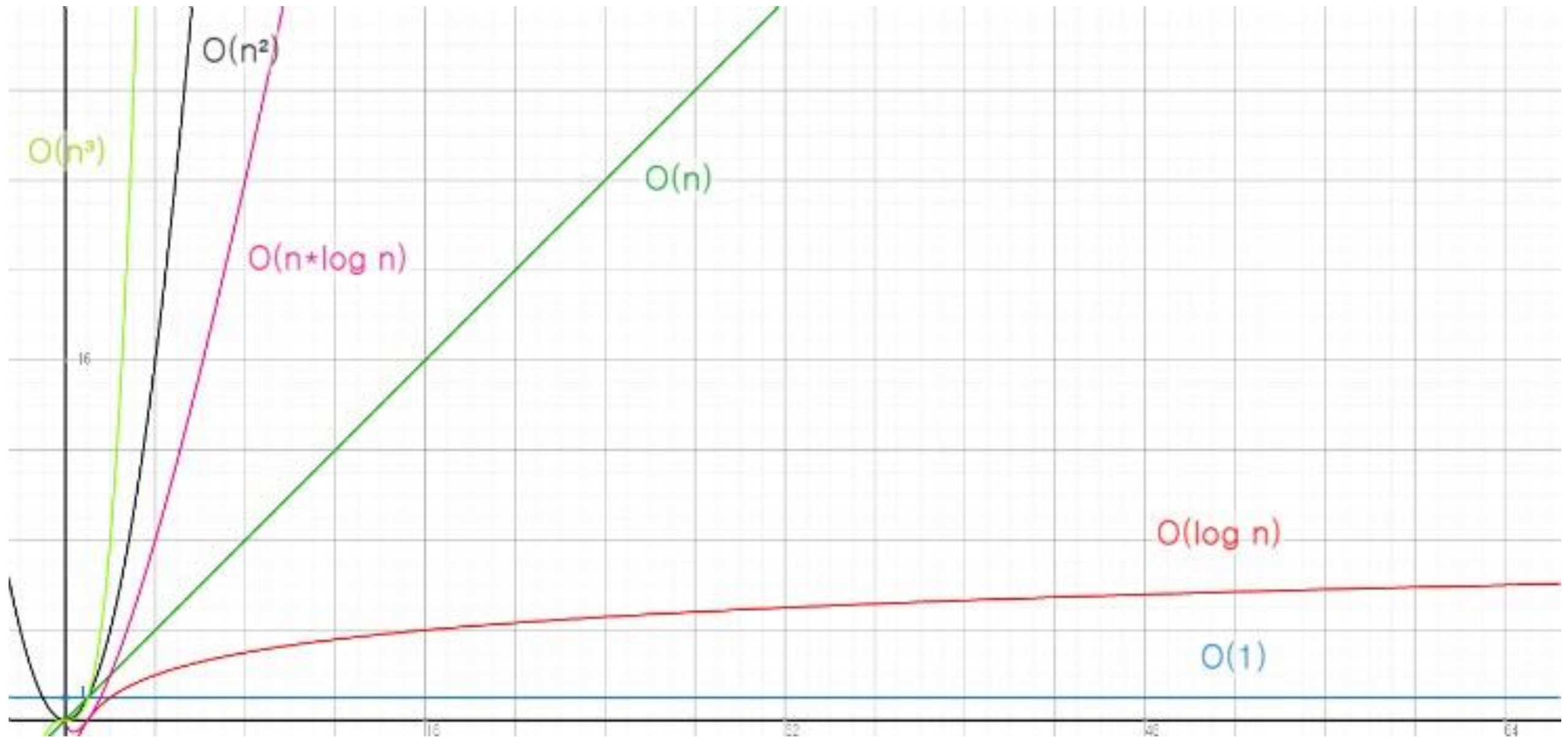




# Common Asymptotic Notations

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



# 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 액세스 성능이 낮고 불안정

## Double Linked List (이중 연결 리스트)

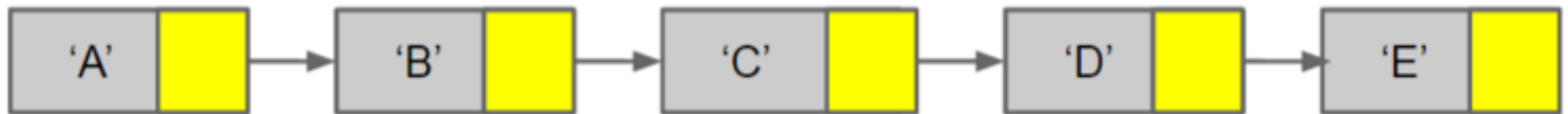
- : HEAD 가 이전, TAIL 이 이후 노드 데이터를 가르키는 것.
- : 끊어진 체인 복구 가능

## 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.**

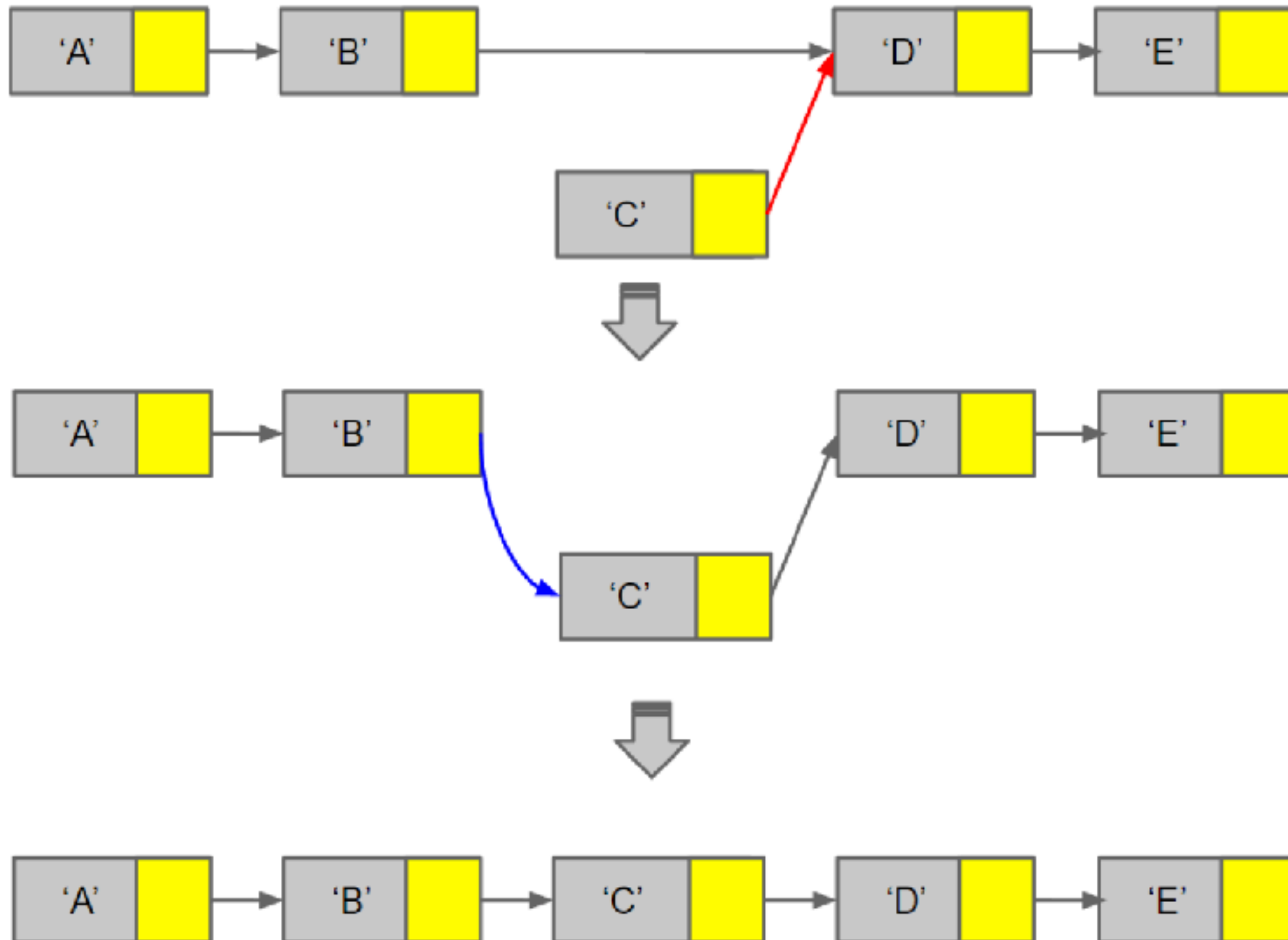
**Deletion – Deletes an element at the beginning of the list.**

**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

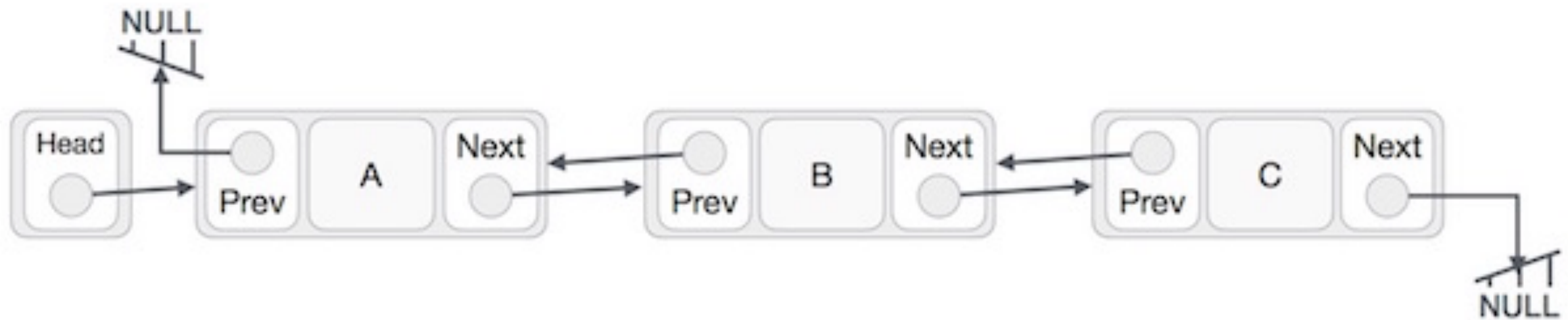


# Double Linked List





# 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.**

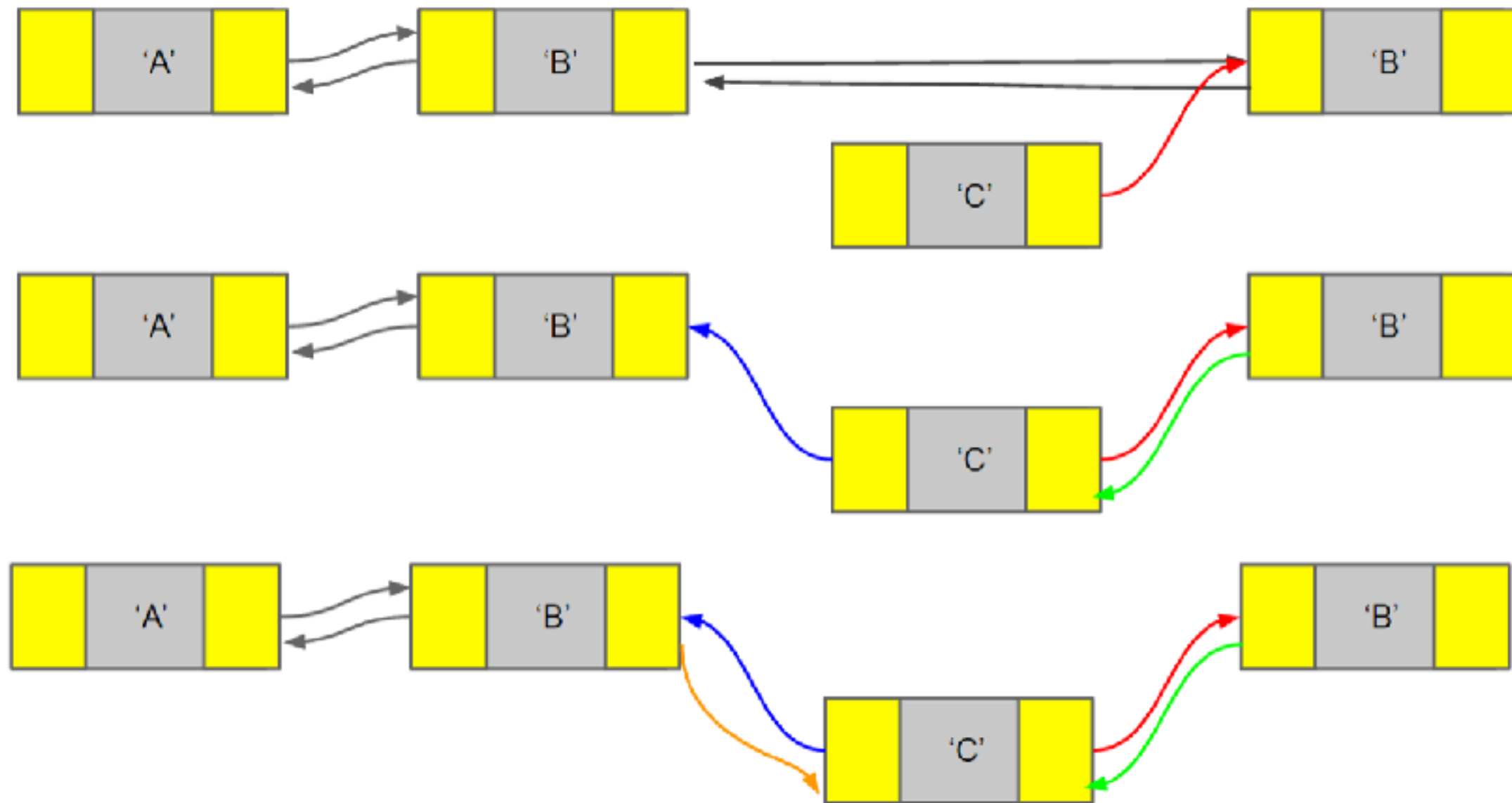
**Insert After – Adds an element after an item of the list.**

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

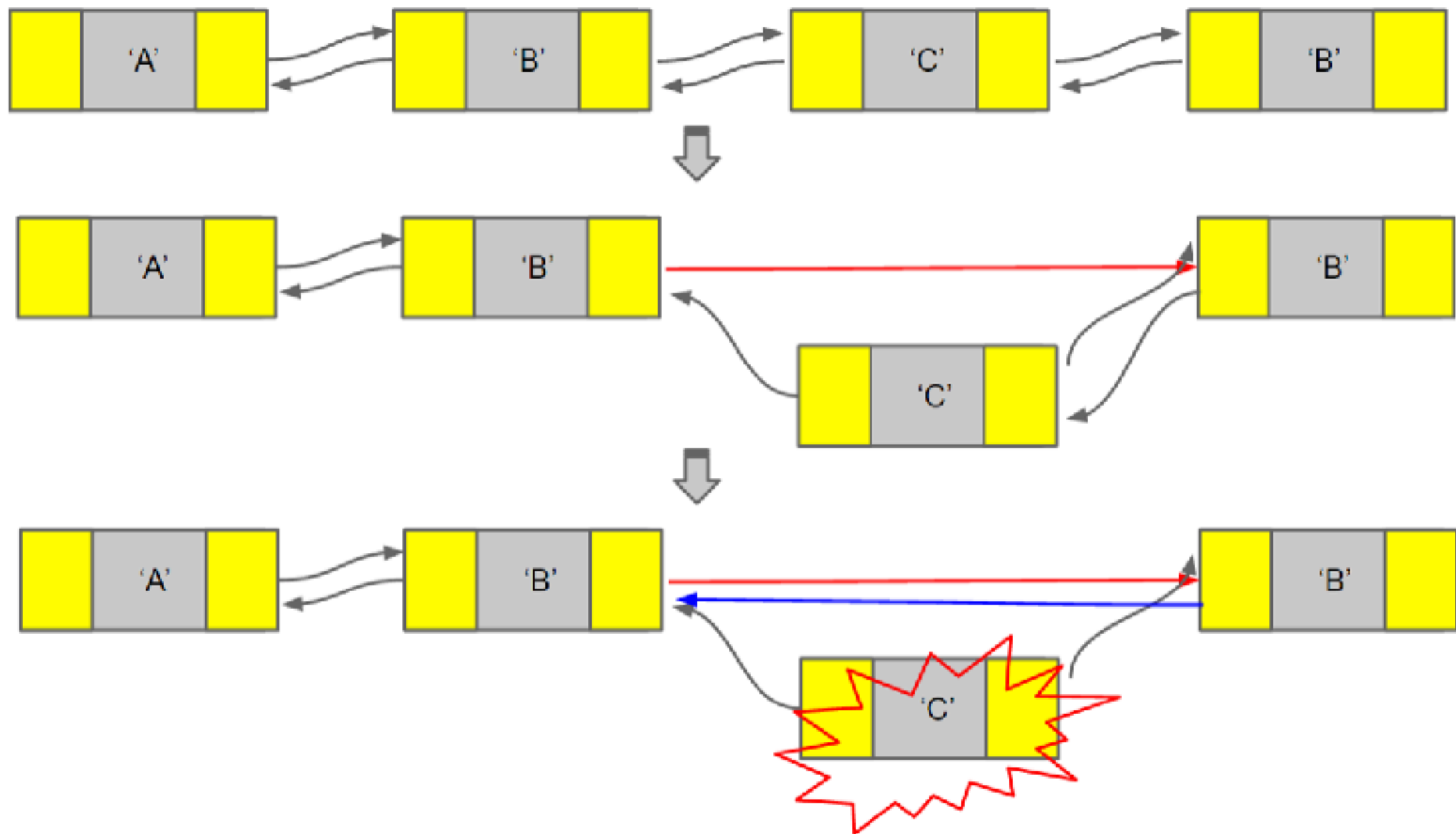
**Display forward – Displays the complete list in a forward manner.**

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

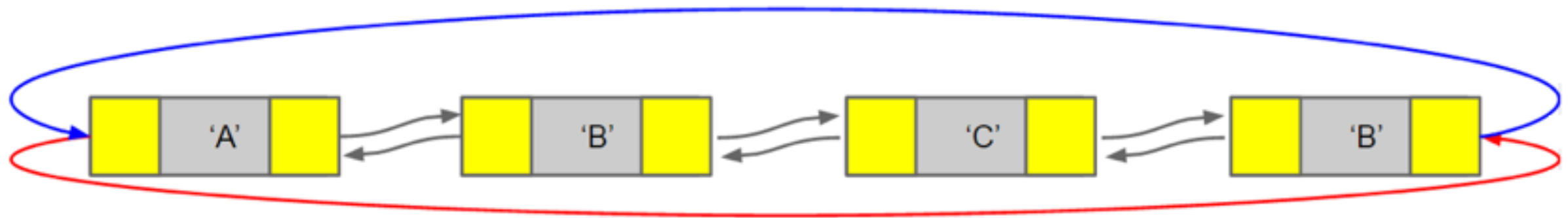
# Double Linked List - Insert Node



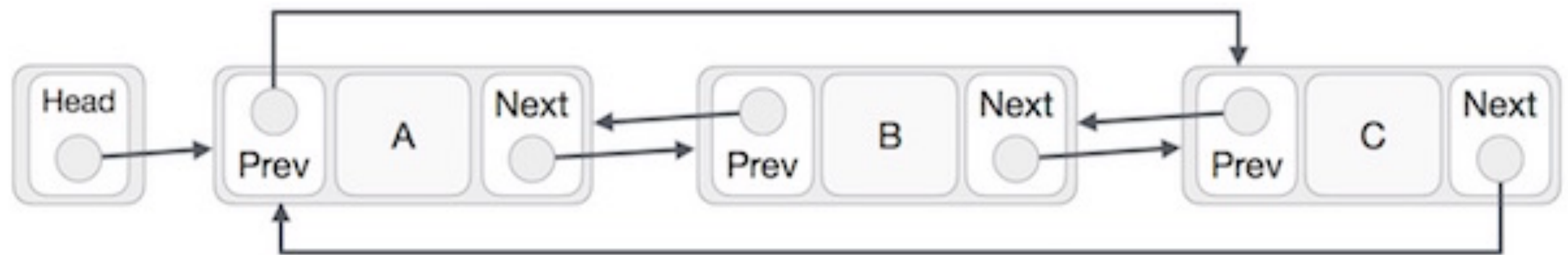
# Double Linked List - Delete Node



# Circular Linked List



# Circular Linked List



# Stack

# Stack

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

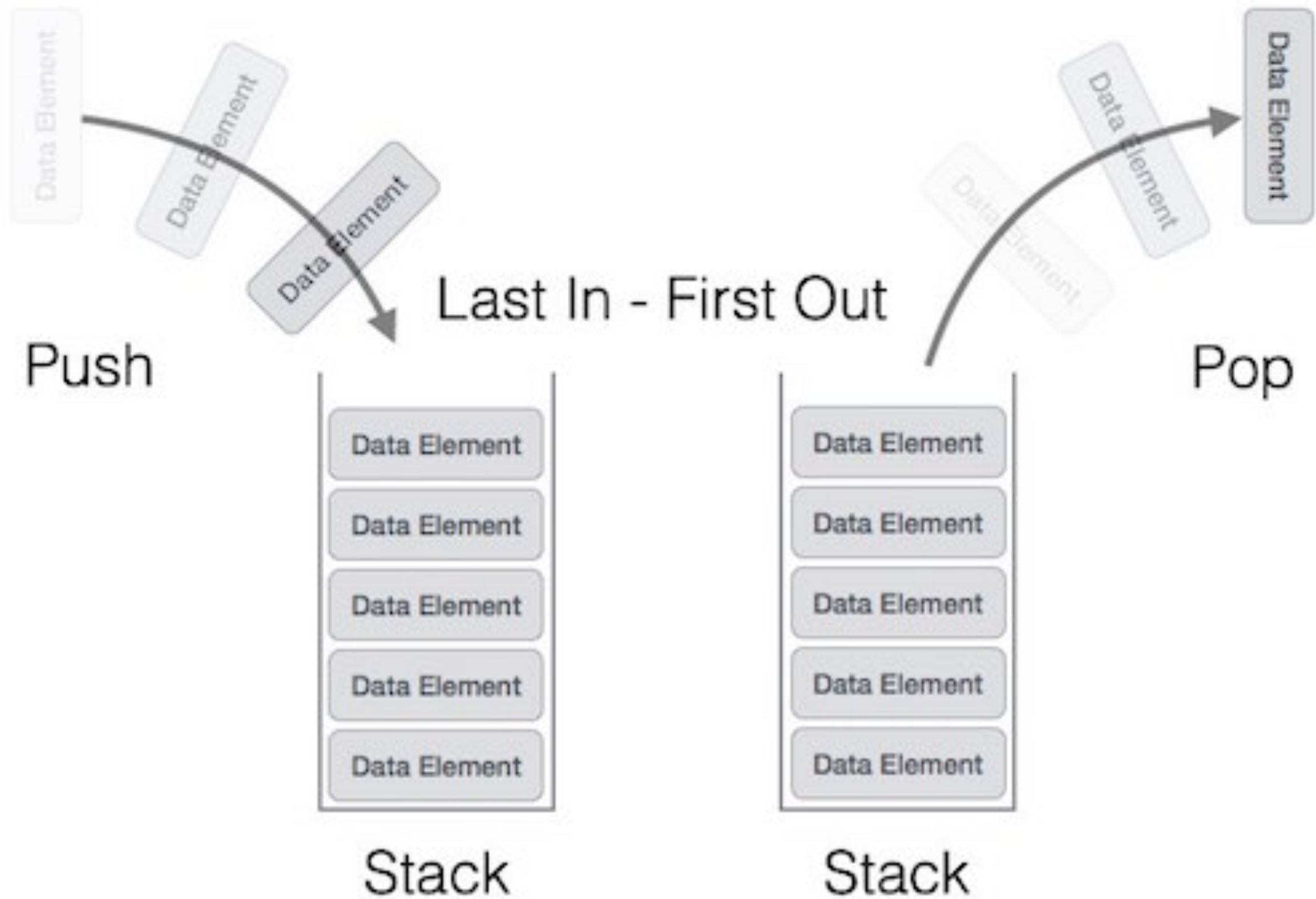
프링글스, 탄창, 대형마트 카트, 떡꼬치 등

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

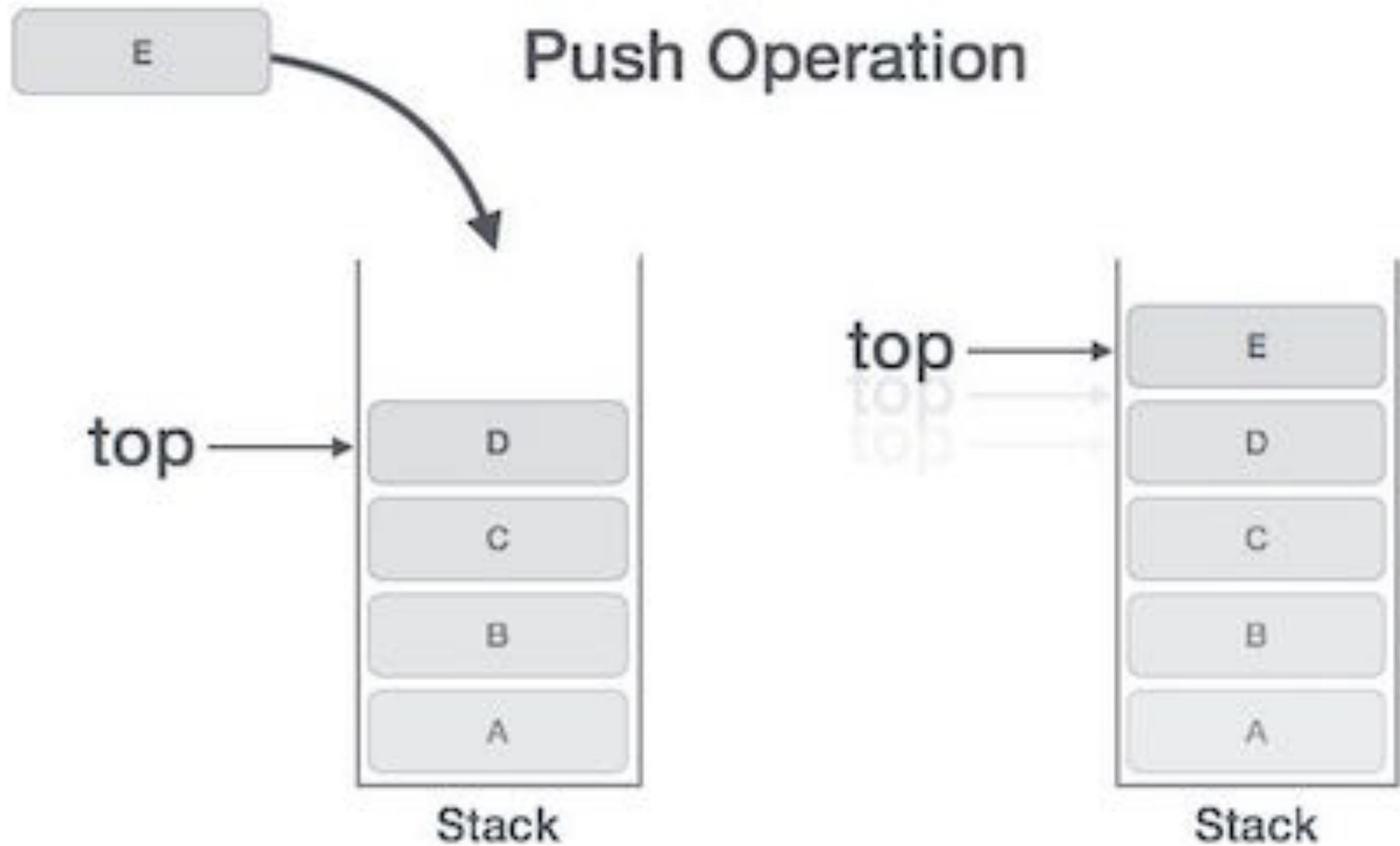




# Stack

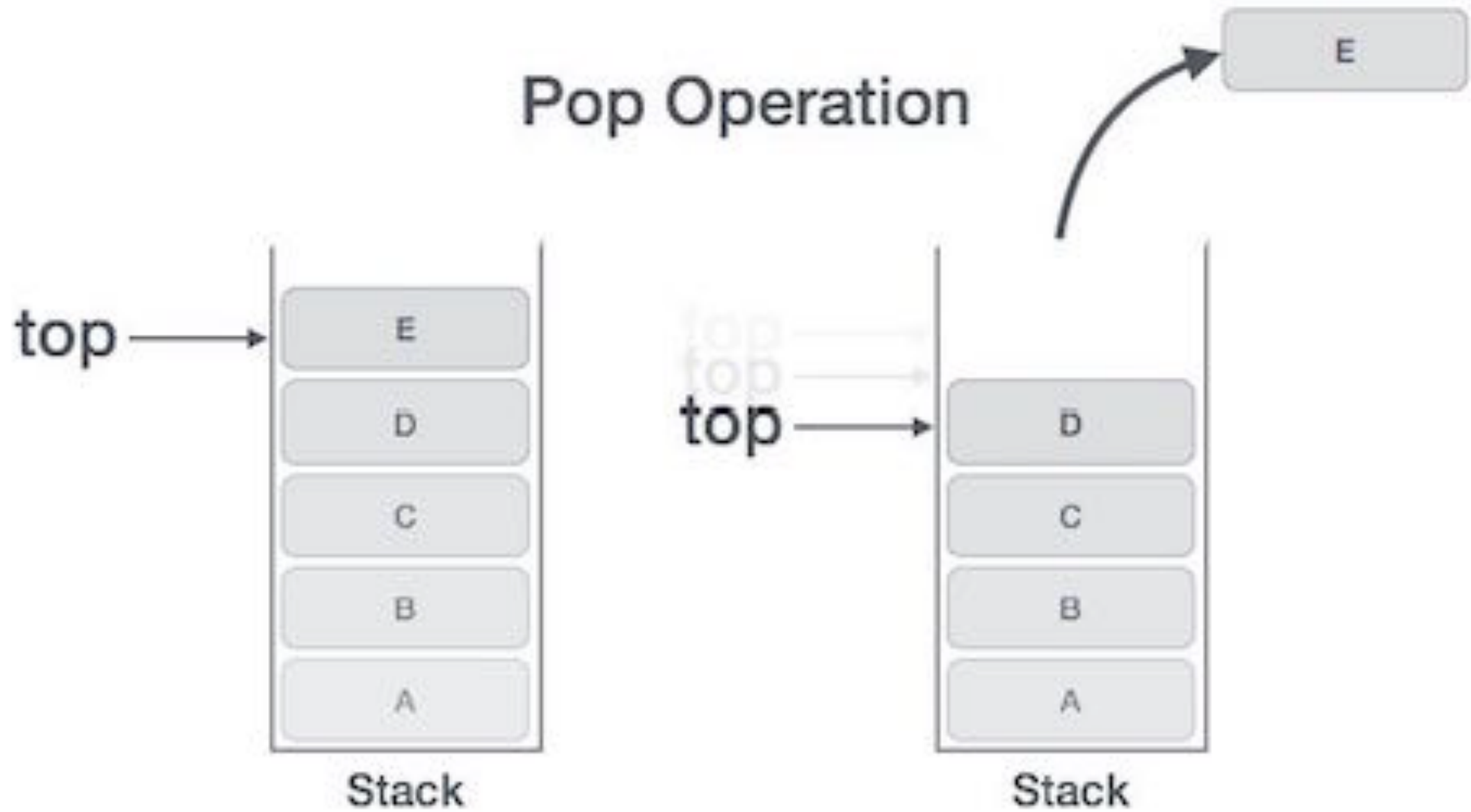


# Push



# Pop

## Pop Operation



# 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



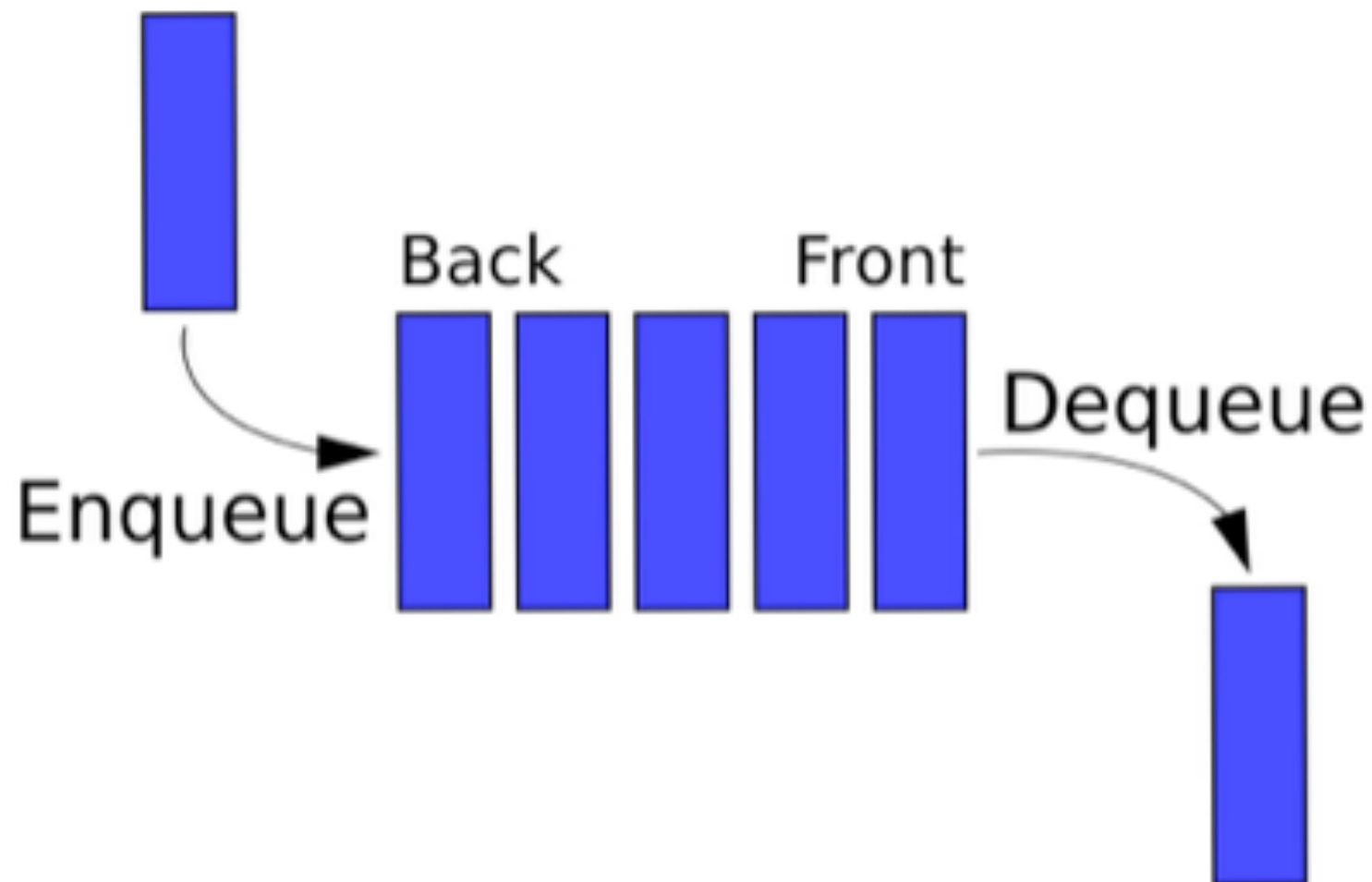
# Queue

값을 추가하는 것 : Enqueue 또는 Put

값을 꺼내는 것 : Dequeue 또는 Get

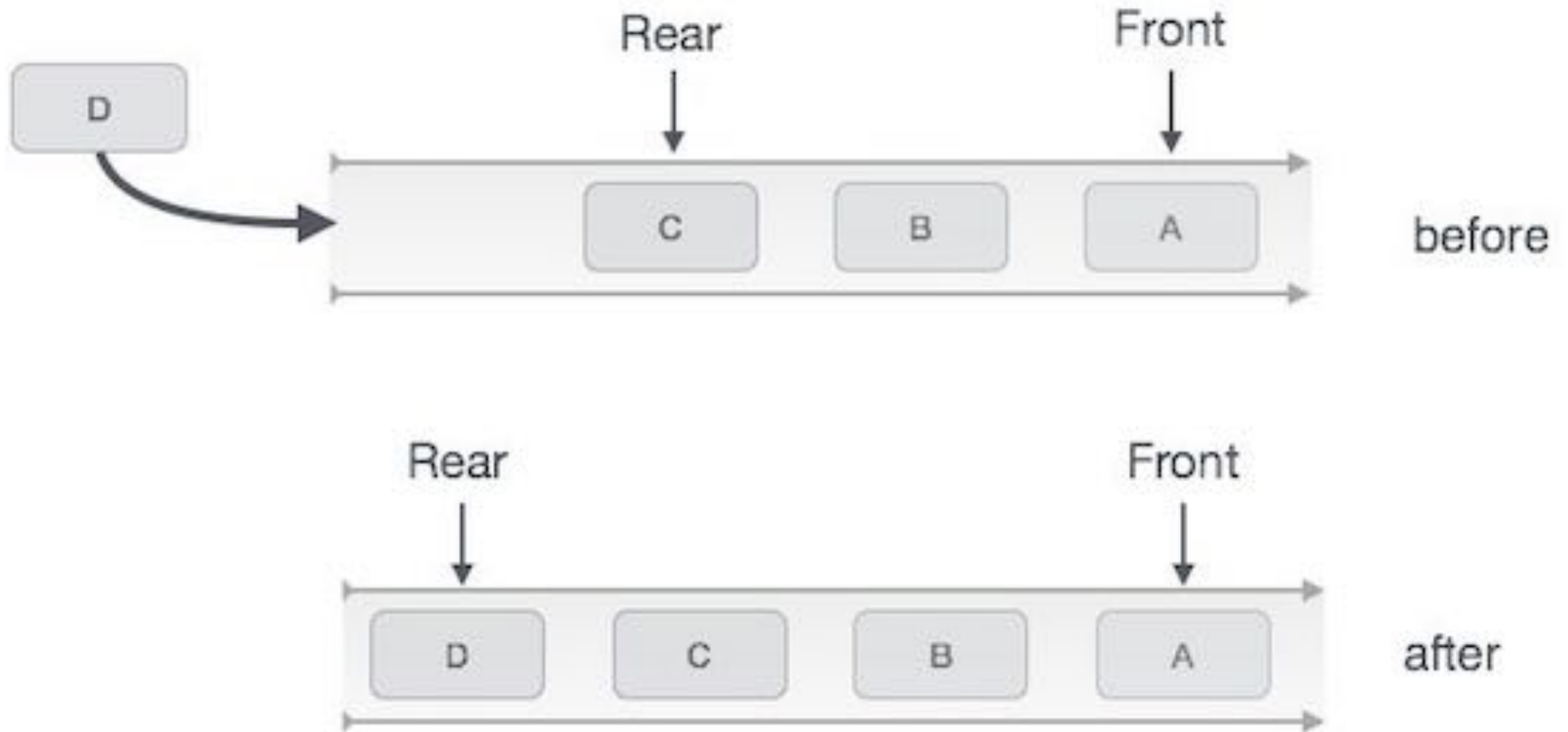
Front : Dequeue 에 사용될 인덱스

Rear (Back) : Enqueue 에 사용될 인덱스

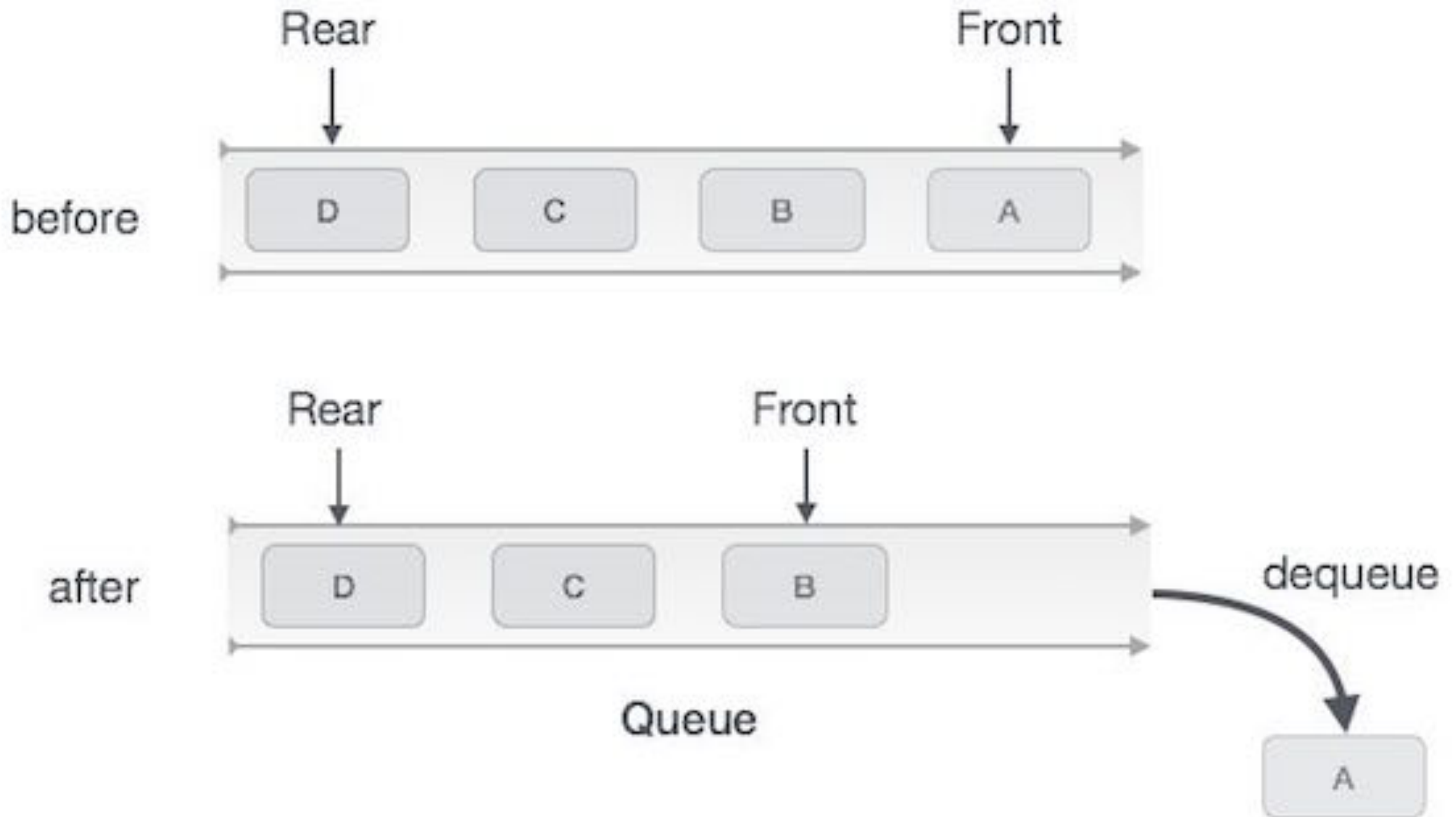




# Enqueue



# Dequeue



# Queue ADT Example

---

**front(head)** - Dequeue 에 사용될 위치를 가리키는 인덱스

**rear(tail)** - Enqueue 에 사용될 위치를 가리키는 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

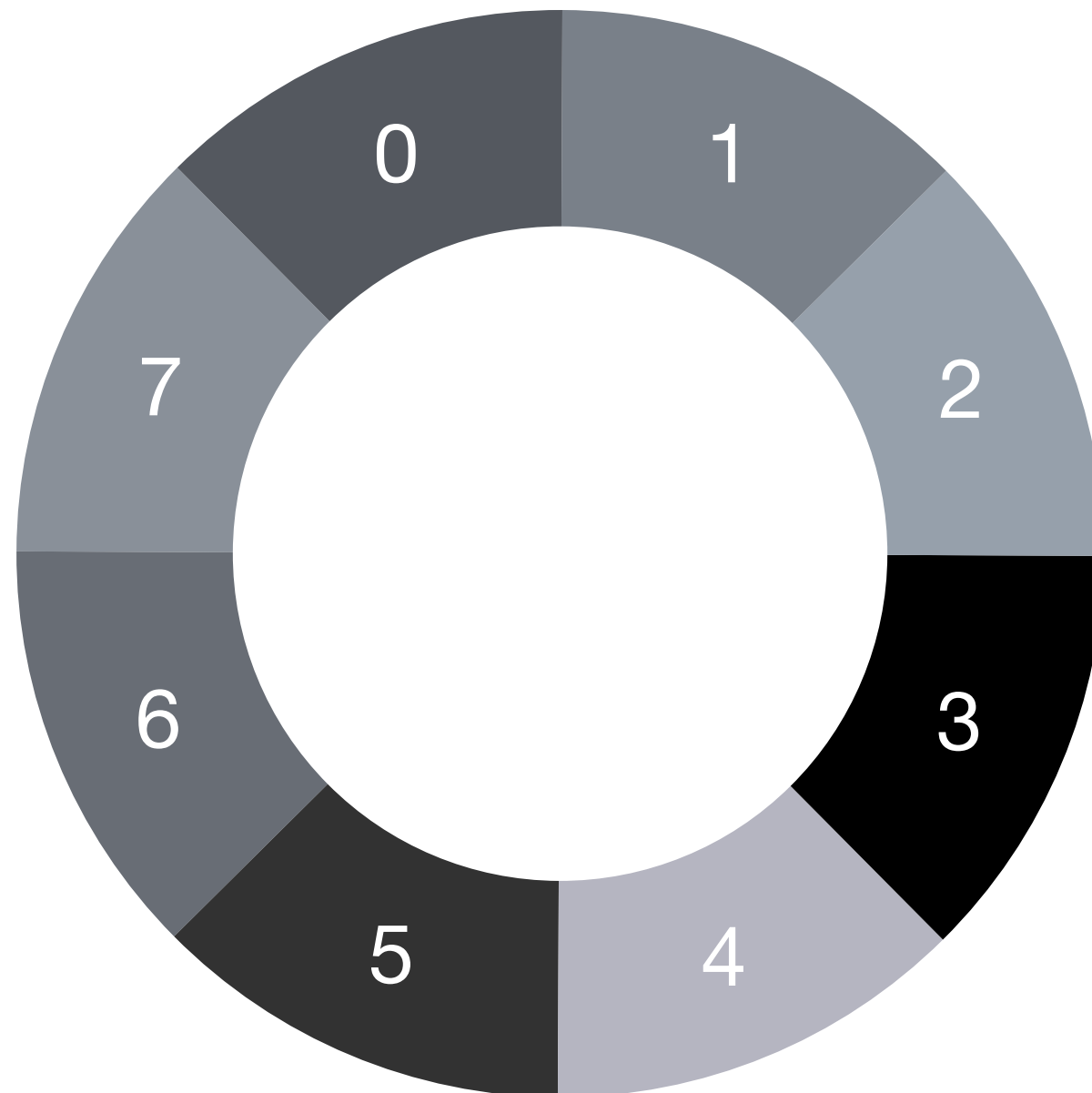
**isEmpty** - Queue 에 데이터가 있는지 확인

**size** - Queue 에 들어가 있는 데이터 개수 확인

# Circular Queue

배열로 Queue 구현 시 발생하는 문제점을 해결하기 위해 원형 큐 이용

비었을 때와 가득 찼을 때를 구별하기 위해 하나의 요소를 빈 노드로 사용 (따라서  $\text{maxSize} + 1$ 의 크기 필요)



# Deque (Double Ended Queue)

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



# Graph

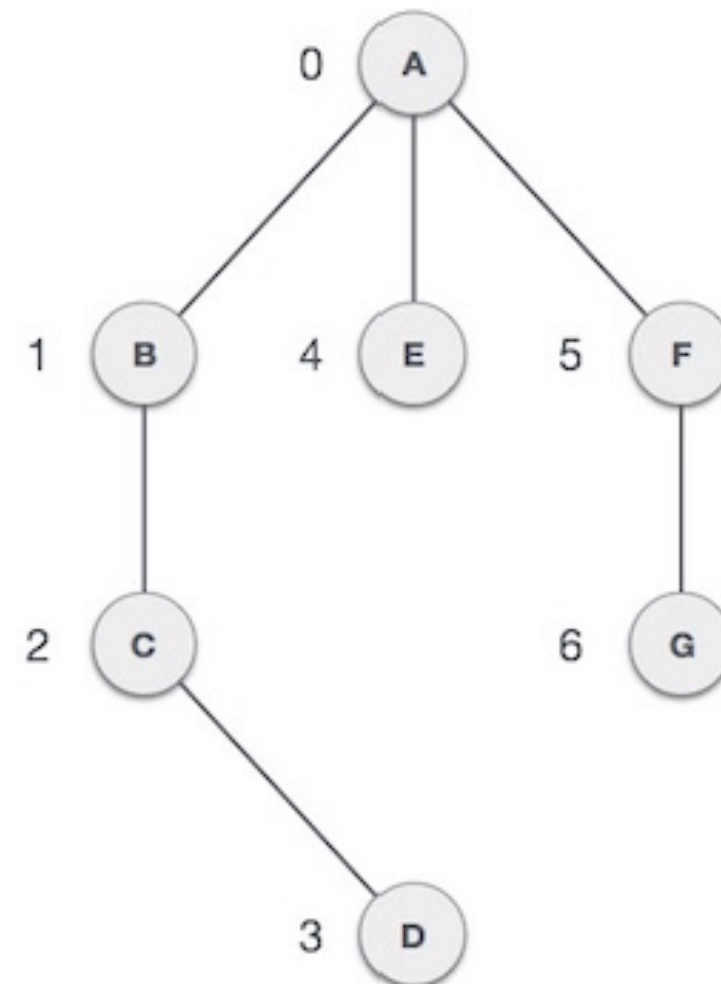
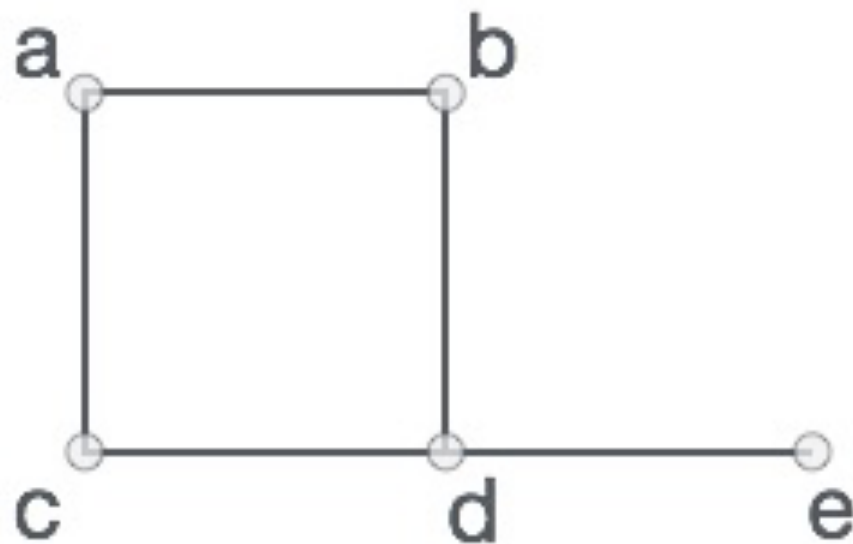
# Graph

그래프  $G$  는  $(V, E)$  의 쌍

$V$  는 정점(vertex) 의 집합(set) 이고  $E$  는 간선(edge) 의 집합

$V$  (정점) - 독립된 개체(Stand-alone object)로 동그라미로 표현

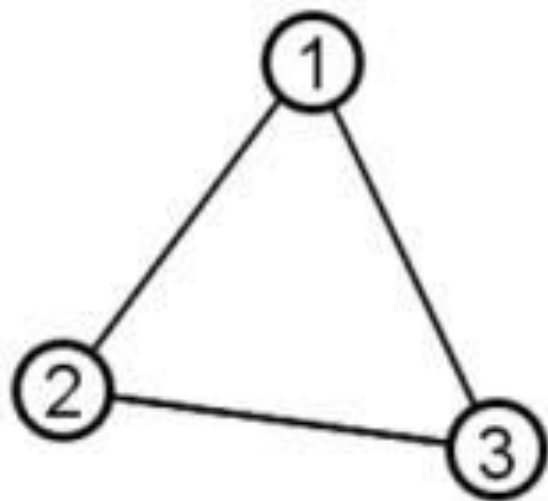
$E$  (간선) - 두 정점을 잇는 개체로 화살표가 있는 선 또는 단순한 연결선으로 표현



# (Un)Directed Graph

무방향성(Undirected) - 무방향성 간선을 가진 그래프. 직선(line) 사용

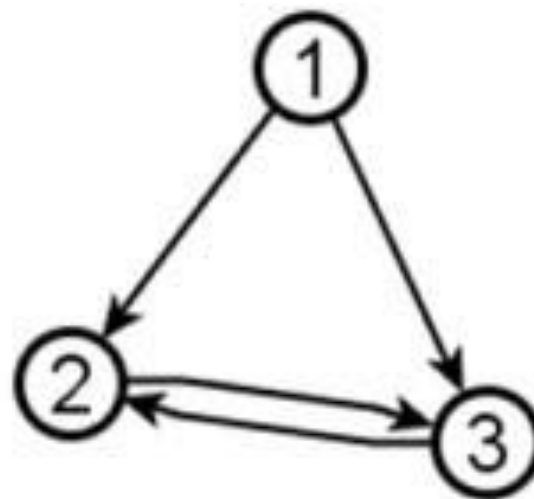
방향성(Directed) - 방향성이 있는 간선을 가지고 있는 그래프. 화살표가 있는 선 사용



Undirected graph  $(V_1, E_1)$

$$V_1 = \{1, 2, 3\}$$

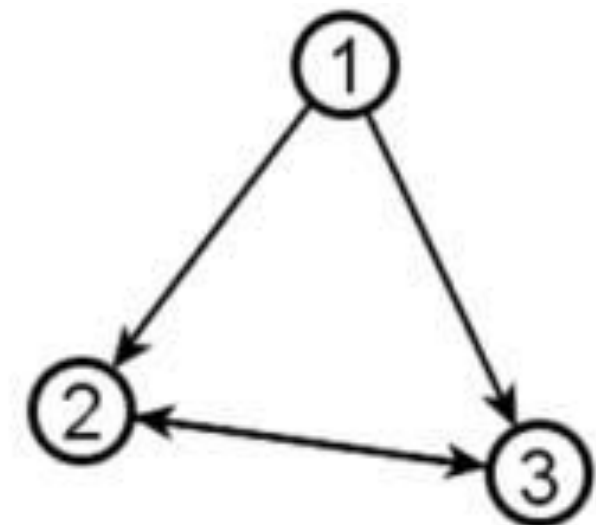
$$E_1 = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$$



Directed graph  $(V_2, E_2)$

$$V_2 = \{1, 2, 3\}$$

$$E_2 = \{(1, 2), (2, 3), (3, 2), (1, 3)\}$$



Easier way to draw  
directed graph  $(V_2, E_2)$



인접 (Adjacency) -  $(u,v)$  라는 간선이 있으면, 정점  $v$  와 정점  $u$ 는 인접한 관계

차수 (Degree) - 진입차수와 진출차수의 합

진출차수 (Out-Degree) - 정점을 나가는 간선의 수

진입차수 (In-Degree) - 정점을 나가는 간선의 수

경로 (Path) - 경로의 길이(length)는 경로에 있는 간선의 수

단순 경로(Simple Path) - 경로에 있는 모든 정점들이 서로 다른 경우

순환 (Cycle) - 경로의 시작과 끝이 같으면 순환

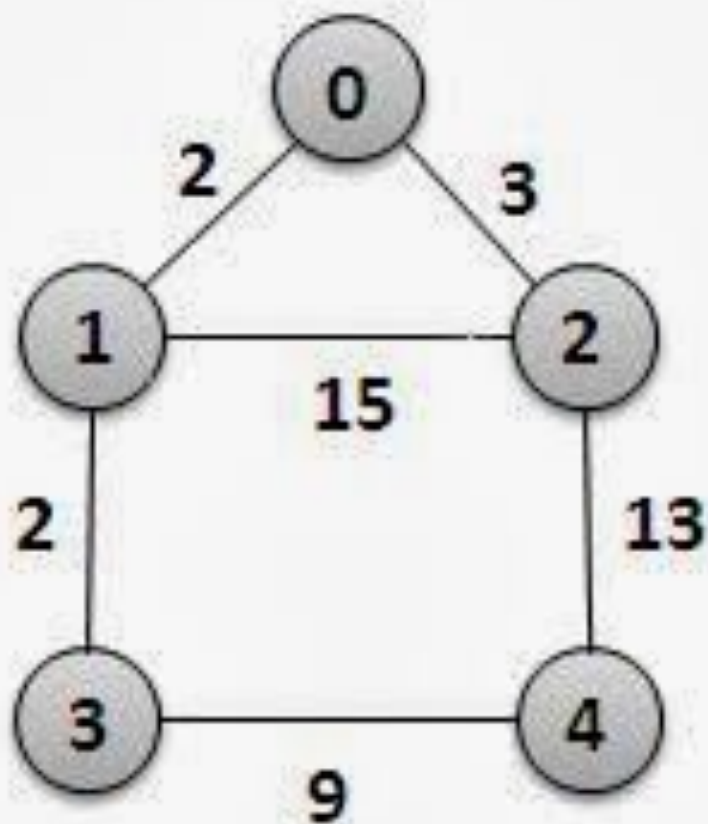
비순환 (Acyclic) - 순환이 없으면 비순환 그래프

연결 그래프 (A connected graph) - 정점의 모든 쌍이 경로를 가지는 무방향성 그래프

완전 그래프 (A complete graph) - 무방향성 그래프에서 모든 정점의 쌍이 서로 인접

포레스트 (Forest) - 순환하지 않는 무방향성 그래프

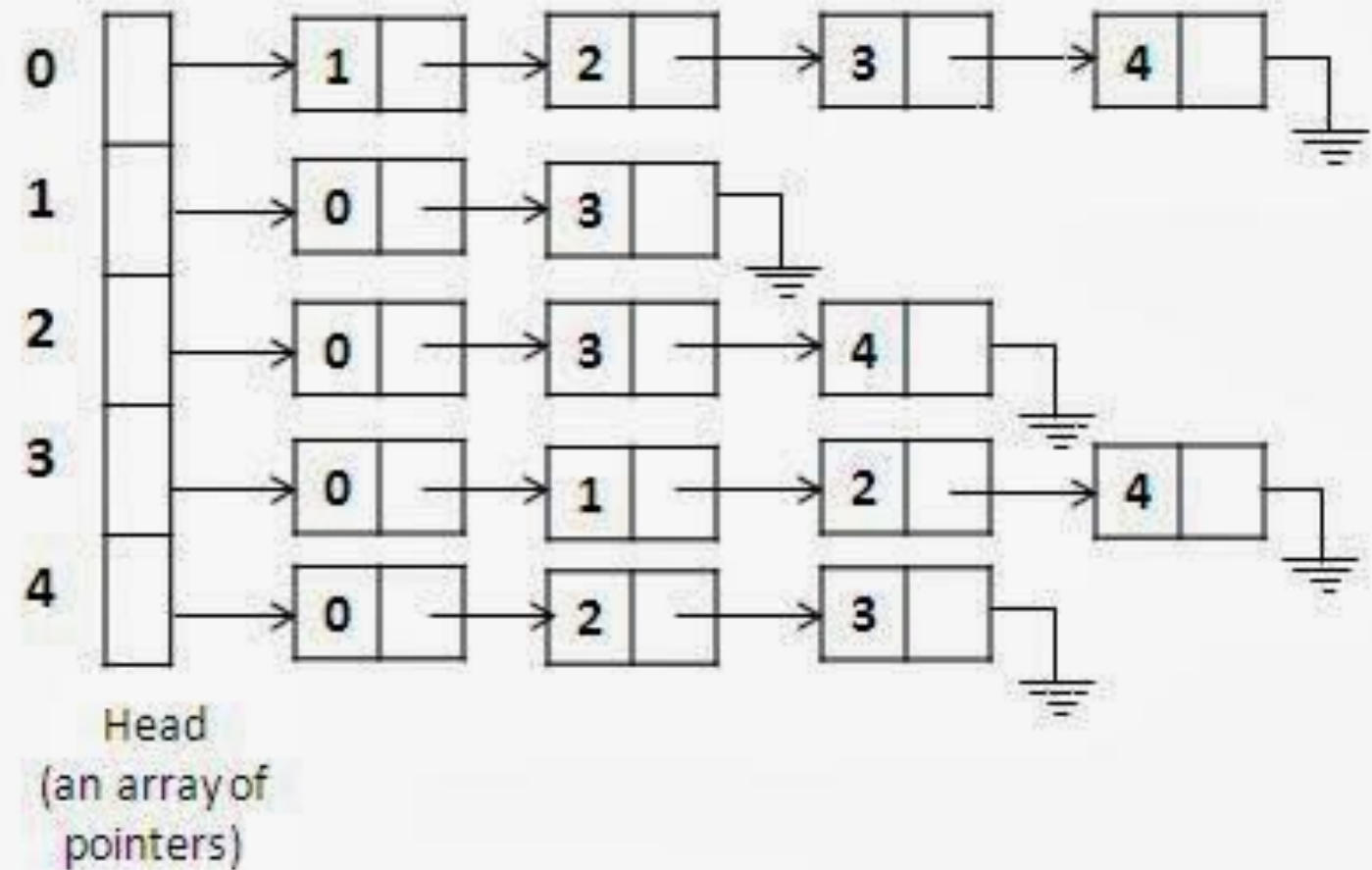
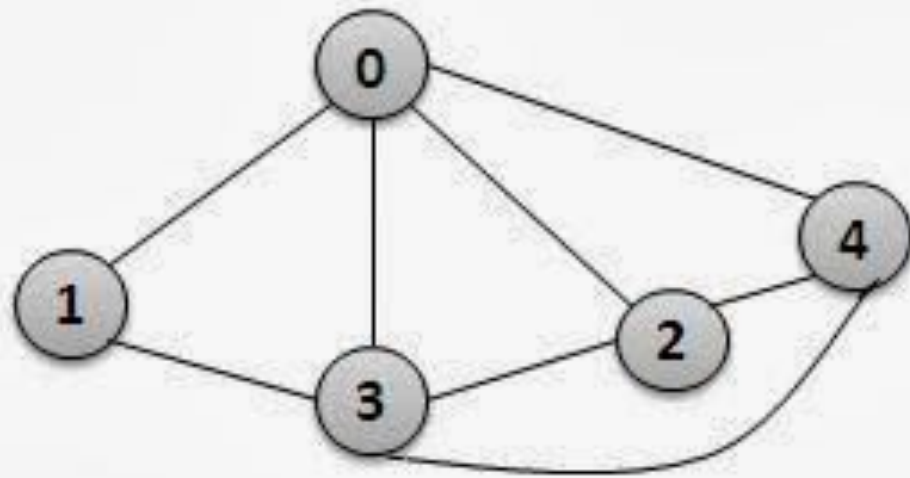
# Adjacency Representation (Matrix)



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

**Adjacency Matrix Representation of  
Weighted Graph**

# Adjacency Representation (Linked List)

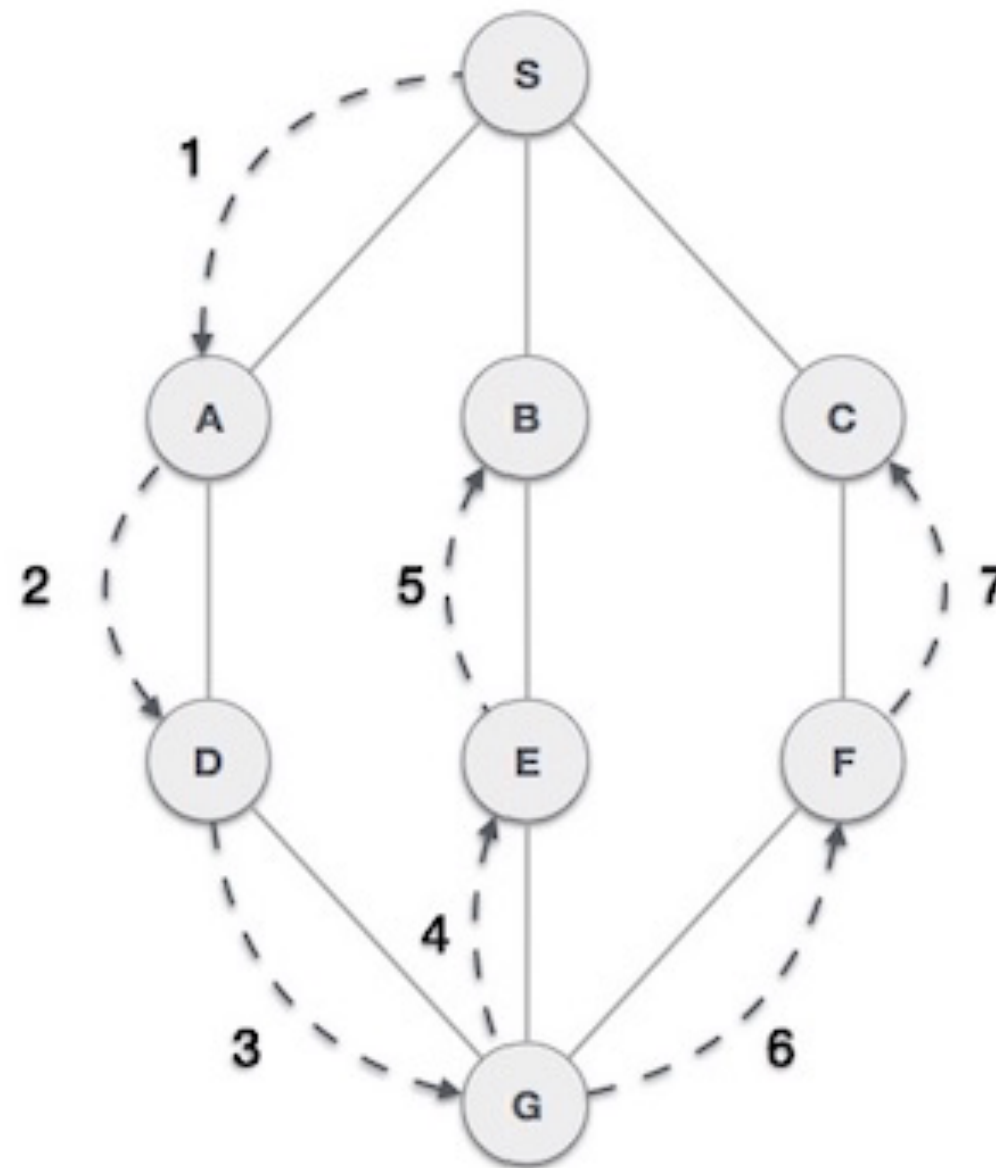


Adjacency List Representation of Graph

# Depth-First Search (DFS)

깊이 우선 탐색 - 탐색된 첫 번째 노드를 먼저 자세히 탐색하는 방법

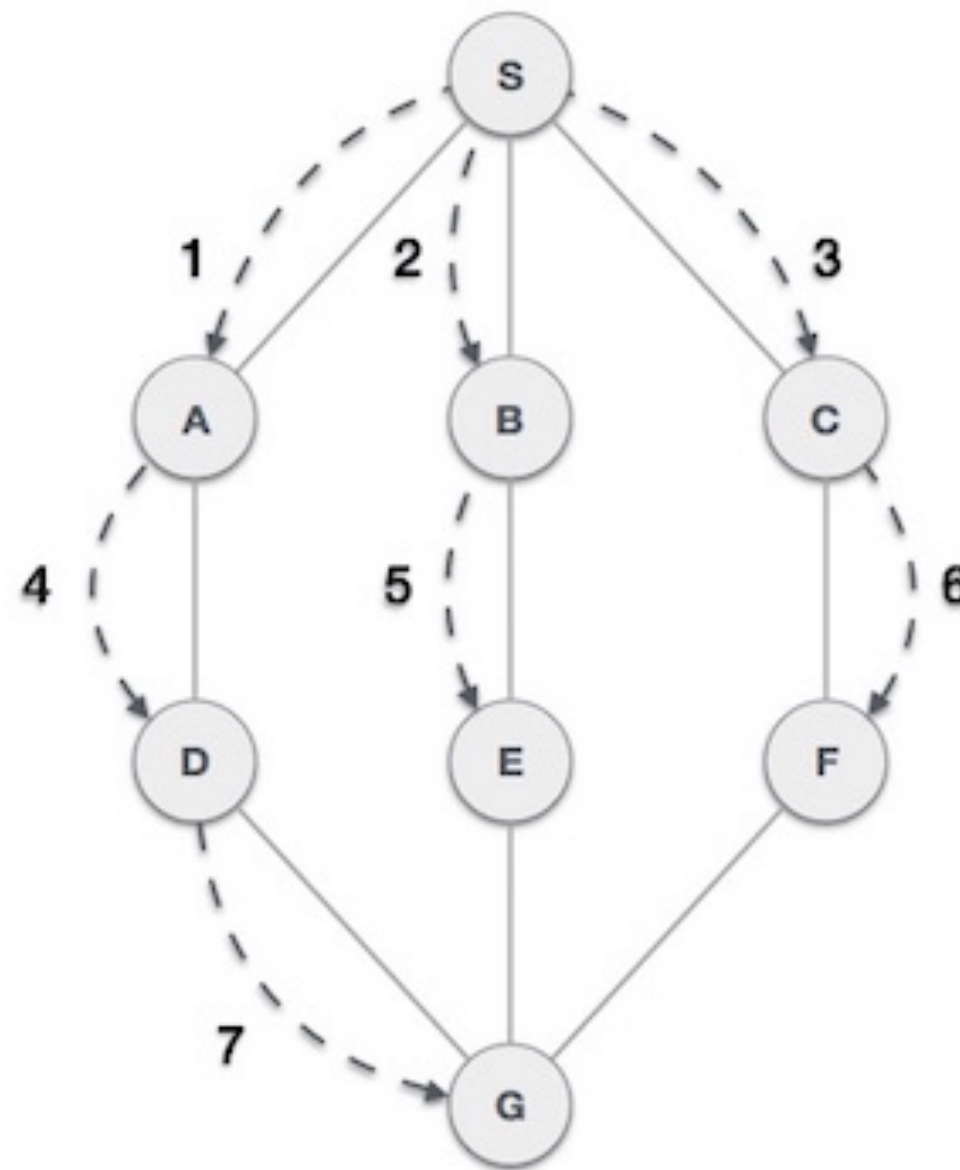
Stack 을 이용해서 탐색 구현



# Breadth-First Search (BFS)

깊이 우선 탐색 - 인접 노드들을 먼저 차례대로 탐색하는 방법

Queue 를 이용해서 탐색 구현



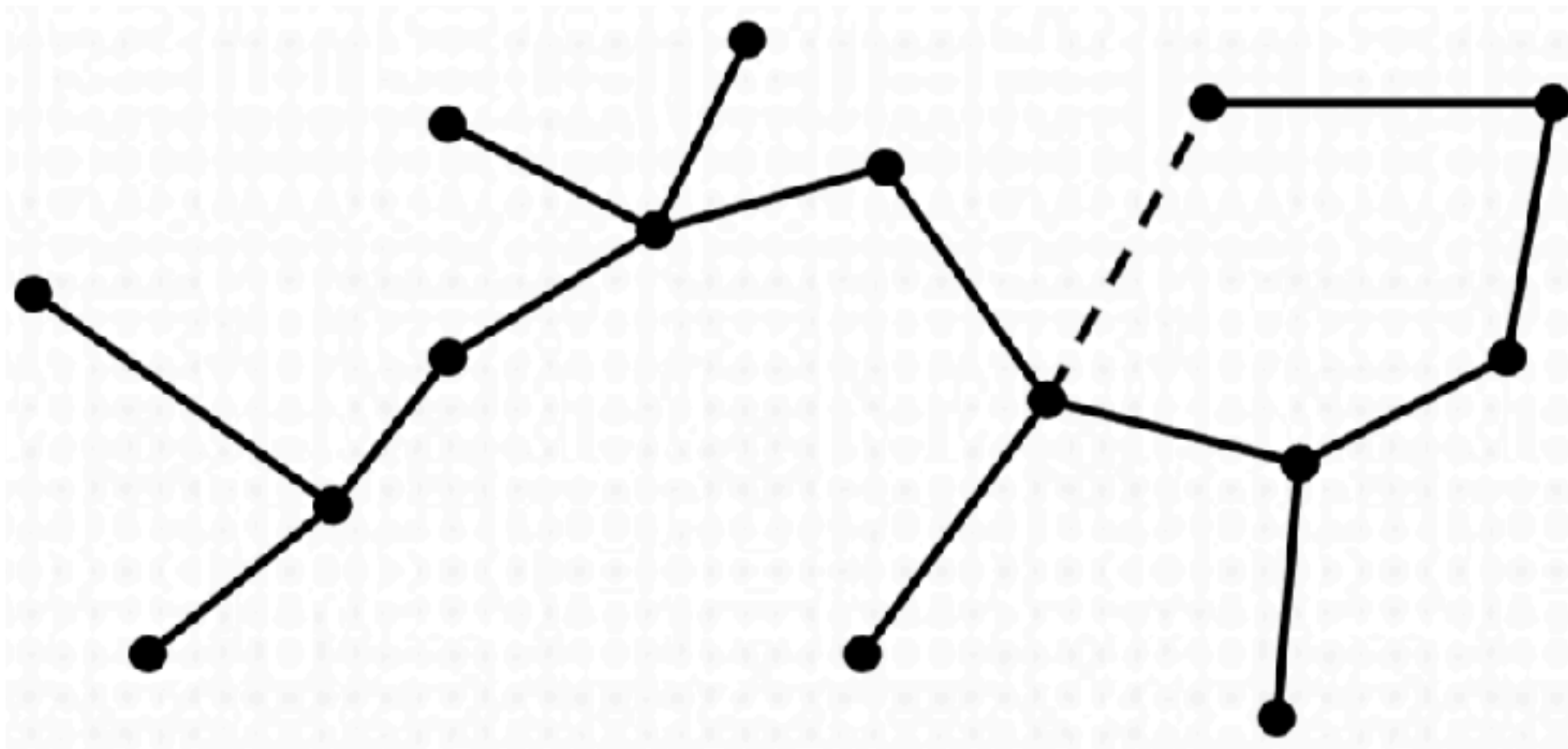
# Tree

# Tree

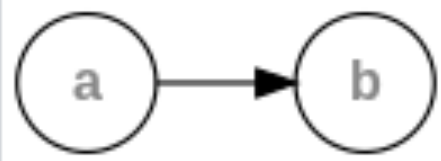
연결된(connected), 비순환(acyclic), 무방향성(undirected) 그래프

Tree 인 경우 간선을 하나 추가하면 순환이 발생하고(cycle) 제거하면 연결이 끊김(disconnected)

$|E| = |V| - 1$  (간선의 개수 = 정점의 개수 - 1)



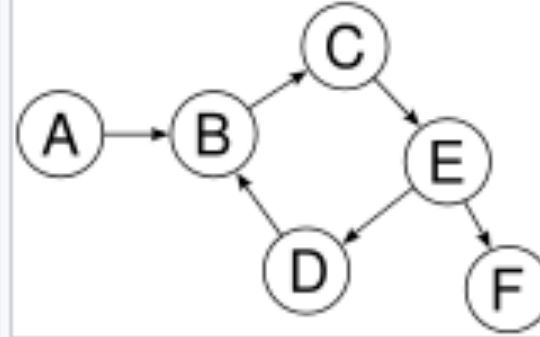
# Not a Tree



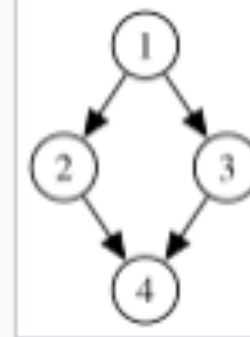
Each linear list is trivially a tree



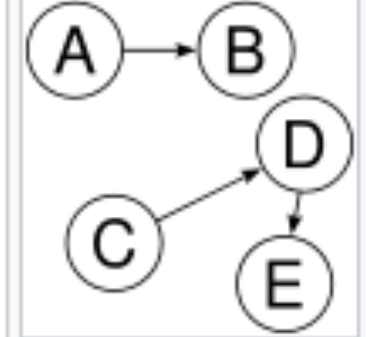
Not a tree: cycle  $A \rightarrow A$ . A is the root but it also has a parent.



Not a tree: cycle  $B \rightarrow C \rightarrow E \rightarrow D \rightarrow B$ . B has more than one parent (inbound edge).



Not a tree: undirected cycle 1-2-4-3. 4 has more than one parent (inbound edge).



Not a tree: two non-connected parts,  $A \rightarrow B$  and  $C \rightarrow D \rightarrow E$ . There is more than one root.





레벨 (Level) - 루트에서 자신에게 걸리는 거리를 레벨(Level), 루트를 Level 1로 출발

트리 높이 (Height) - 가장 높은 레벨(Level), 깊이(Depth)라고 부름

부모 (Parent) - Level N인 노드로 가기 위한 Level N-1에 있는 노드

자식 (Son) - Level N인 노드와 연결 상태인 Level N+1에 있는 노드

형제 (Sibling) - 같은 부모를 갖는 노드

조상 (Ancestors) - 자신에게 오기 위한 경로에 있는 모든 노드

후손 (Descendant) - 자신을 출발로 갈 수 있는 모든 노드

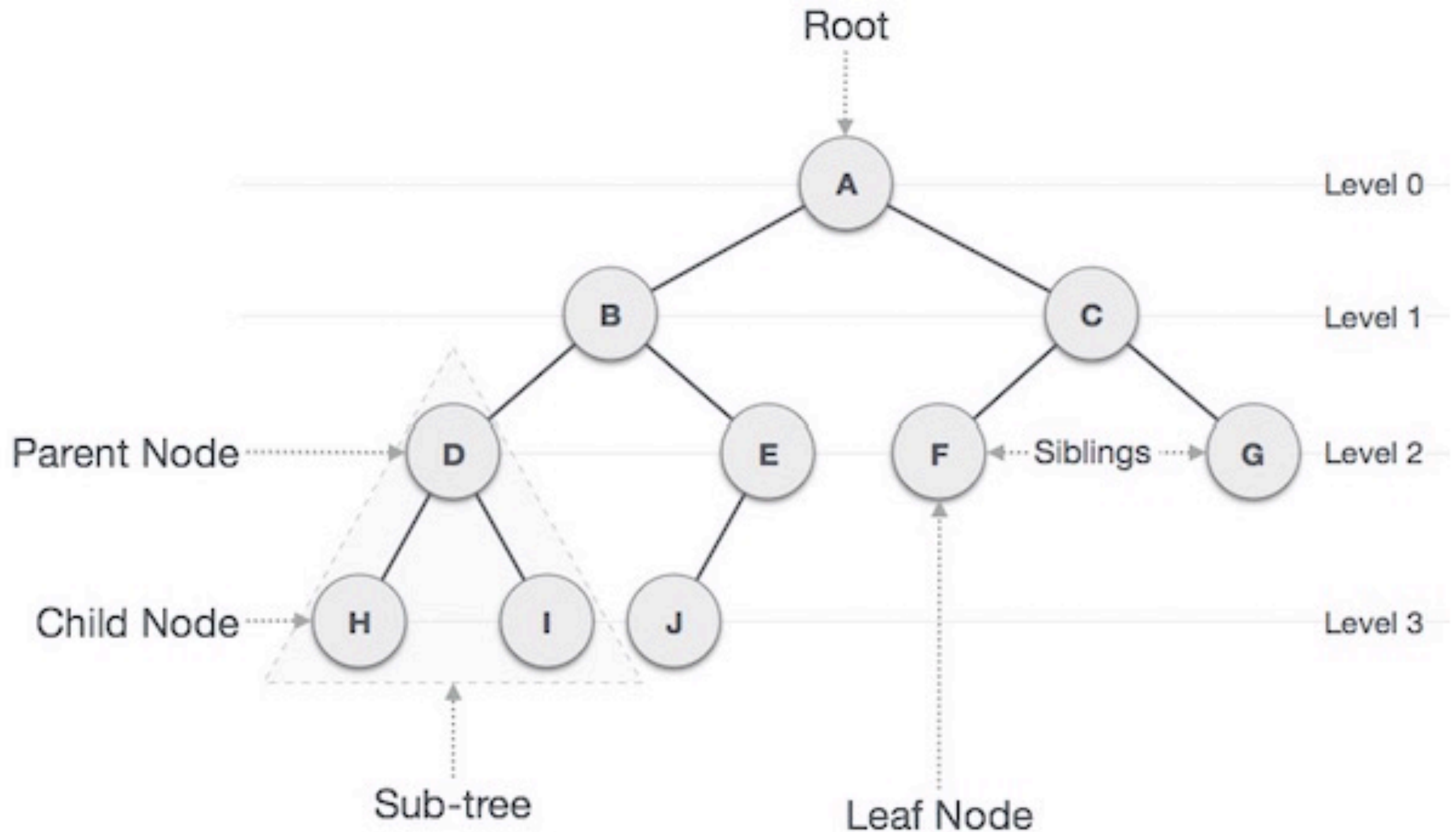
노드의 차수 (Degree) - 특정 노드의 자식 수

트리의 차수 - 트리의 모든 노드 중에 가장 높은 차수

단말 (Terminal) - 차수가 0인(자식이 없는) 노드, 단말(Terminal) 혹은 잎(Leaf)이라고 부름

서브 트리 (Sub Tree) - 자신의 자식을 루트로 하는 트리, 차수와 서브 트리의 개수는 같음

# Basic Terms



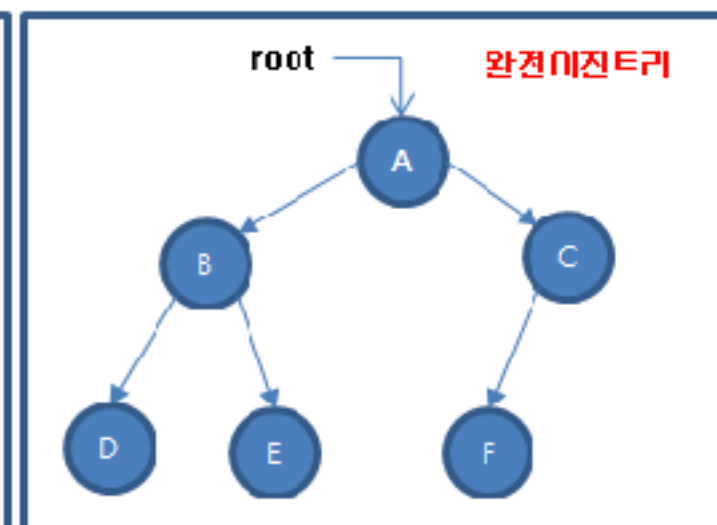
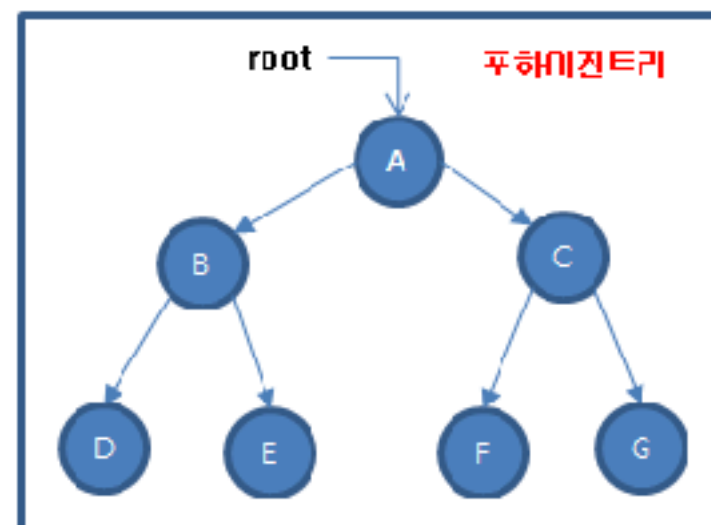
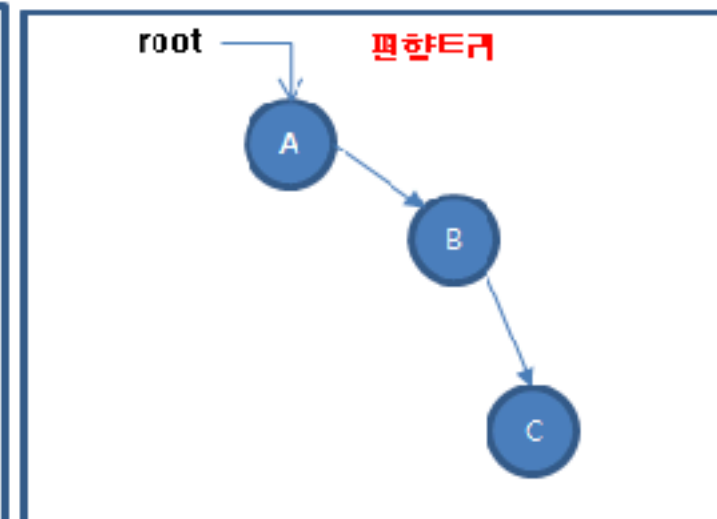
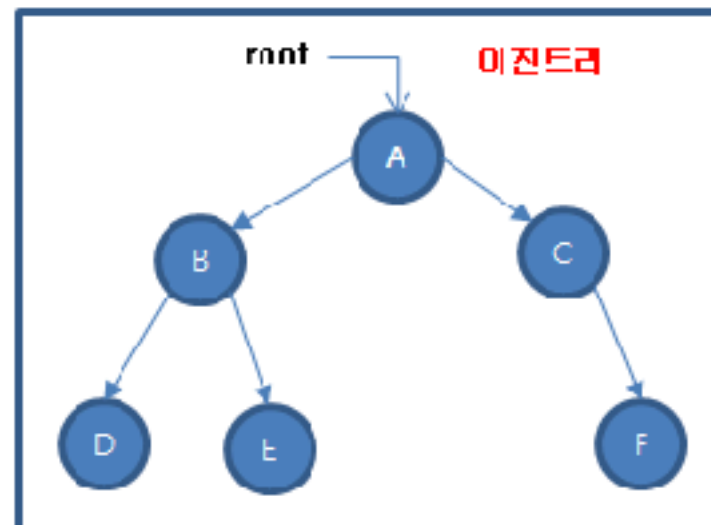
# Tree Types

이진 트리 (Binary Tree) : 모든 노드의 차수가 2 이하

편향 트리 또는 사향 트리 (Skewed Binary Tree) : 한쪽으로 치우쳐진 트리

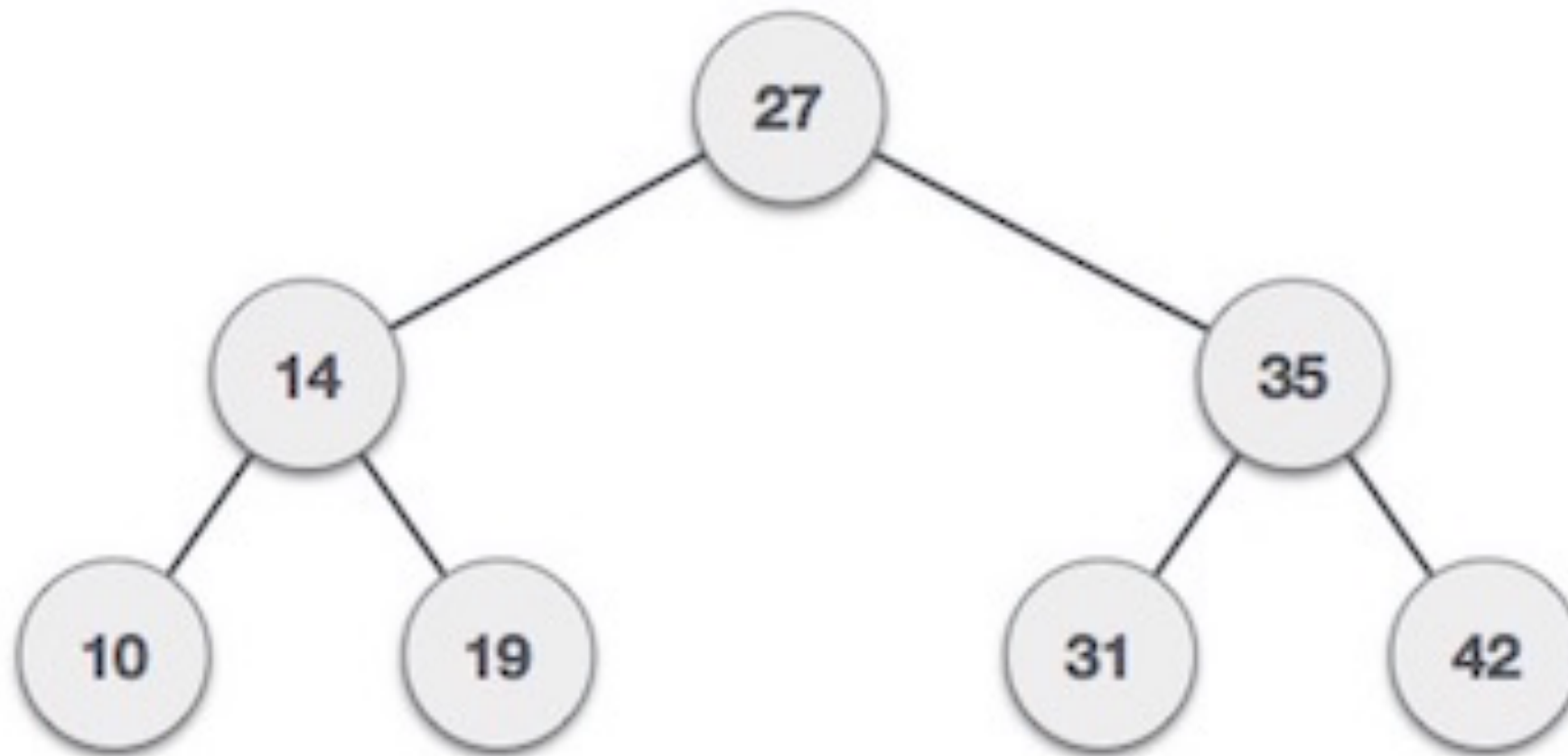
포화 이진 트리 (Full Binary Tree) : 마지막 레벨까지 모든 노드가 있는 이진 트리

완전 이진 트리 (Complete Binary Tree) : 노드가 왼쪽부터 차례대로 추가되는 이진 트리



# Binary Search Tree

노드의 좌측 서브트리에는 그 노드의 값보다 작은 값을 가진 노드, 우측에는 큰 값을 가진 노드들로 구성  
각 노드의 키 값은 모두 달라야 하며 중복된 노드가 없어야 함  
좌우 하위 트리는 각각이 다시 이진 탐색 트리여야 함

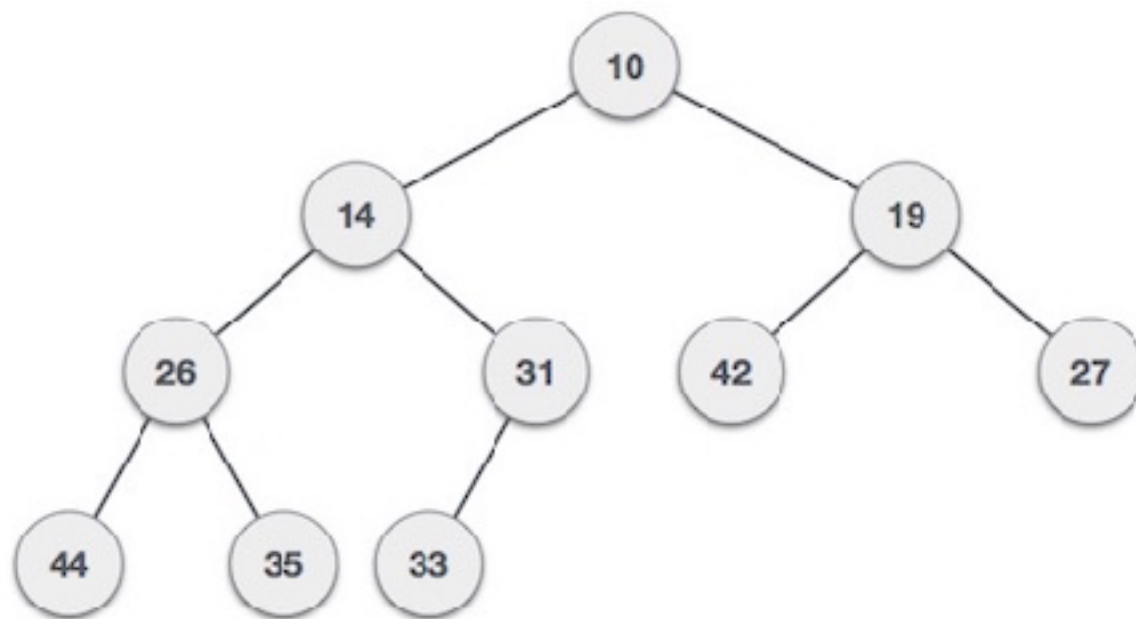


평형(균형) 이진 트리 (Balanced Binary Tree, BBT) 의 특수 케이스

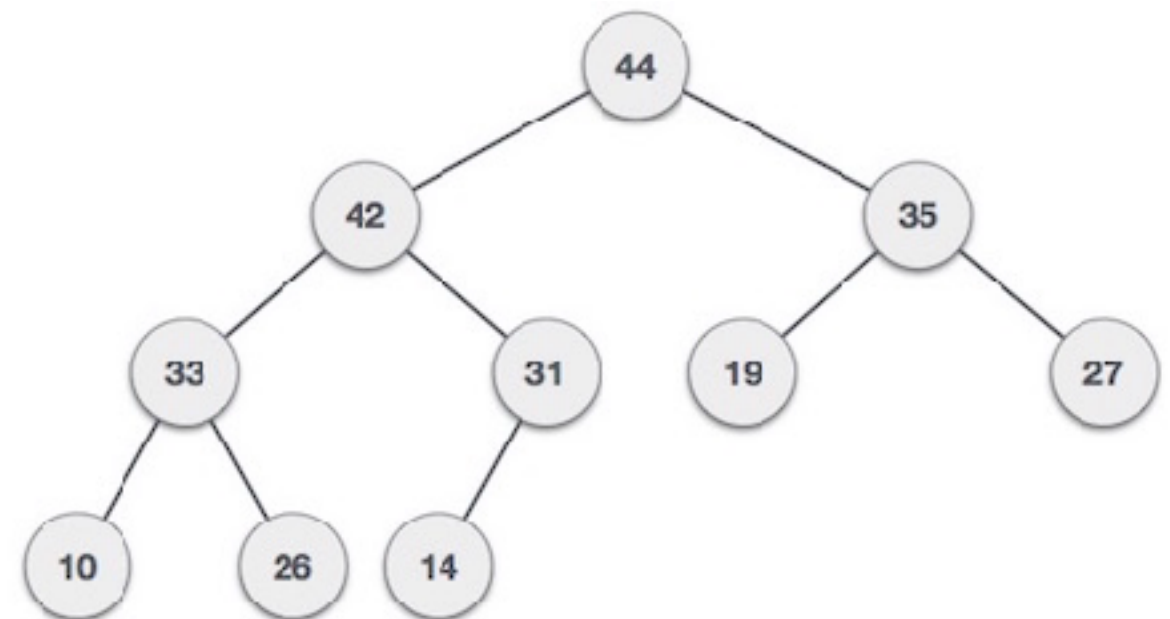
※ 평형 이진 트리 - Subtree 간의 Max Height 차이가 1 이하인 경우

평형 이진 트리면서 부모 노드의 값이 자식 노드의 값보다 항상 크거나 같은, 또는 작거나 같은 트리

최소힙 (Min-Heap) 또는 최대힙 (Max-Heap) 2가지



[ Min-Heap ]



[ Max-Heap ]