자료구조 & 알고리즘

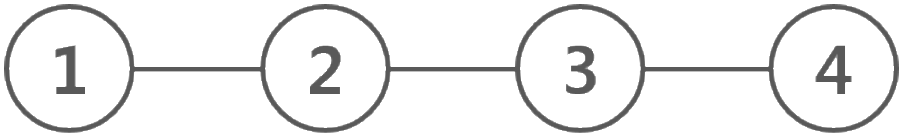
자료구조

- 선형 구조: 데이터의 연속적인 모임
  - ✓ Array(배열 – Contiguous List) – DenseList, ArrayList
  - ✓ LinkedList(연결 리스트)
  - ✓ Stack
  - ✓ Queue
  - ✓ Deque(Doubly Ended Queue): Stack 과 Queue의 결합으로 양쪽에서 삽입과 삭제가 가능한 자료구조로 입력은 한쪽에서만 가능하도록 제한하면 Scroll 이라고 하고 출력을 한쪽에서만 하도록 제한하면 Shelf 라고 함
- 비선형 구조: 데이터의 불연속적인 모임
  - ✓ Tree
  - ✓ Graph

선형 자료구조

❖ 선형 자료구조

✓ 자료를 구성하는 데이터를 순차적으로 1:1로 나열시킨 자료구조



✓ 종류

- List
- Stack
- Queue

List

- List
  - ✓ 동일한 자료형의 데이터를 순서대로 저장하는 자료구조
  - ✓ 순서대로 = 차례대로 = 한 줄로 = 선형 구조로
  - ✓ 다른 자료구조를 구현하는 수단으로도 사용
    - 스택(stack), 큐(queue), 트리(tree), 그래프(graph) 등에 활용
  - ✓ 종류
    - 연속 자료구조(Contiguous): Array(배열) & Vector(Array List)
    - 연결 자료구조(Link): Linked List

구분	순차 자료구조	연결 자료구조
메모리 저장 방식	메모리의 저장 시작 위치부터 빈자리 없이 자료를 순서대로 연속하여 저장한다. 논리적인 순서와 물리적인 순서가 일치하는 구현 방식이다.	메모리에 저장된 물리적 위치나 물리적 순서와 상관없이, 링크에 의해 논리적인 순서를 표현하는 구현 방식이다.
연산 특징	삽입·삭제 연산을 해도 빈자리 없이 자료가 순서대로 연속하여 저장된다. 변경된 논리적인 순서와 저장된 물리적인 순서가 일치한다.	삽입·삭제 연산을 하여 논리적인 순서가 변경되어도, 링크 정보만 변경되고 물리적 순서는 변경되지 않는다.

Array

- 배열
  - ✓ 크기가 고정 – 한 번 생성하면 크기를 변경할 수 없음
  - ✓ 빈 공간없이 데이터를 저장하기 때문에 메모리 효율이 좋음
  - ✓ 데이터를 중간에 삽입하거나 삭제할 수 없음
  - ✓ 삽입이나 삭제 작업을 해야 하는 경우에는 다른 배열을 생성하고 데이터를 복사하는 작업을 통해서 수행

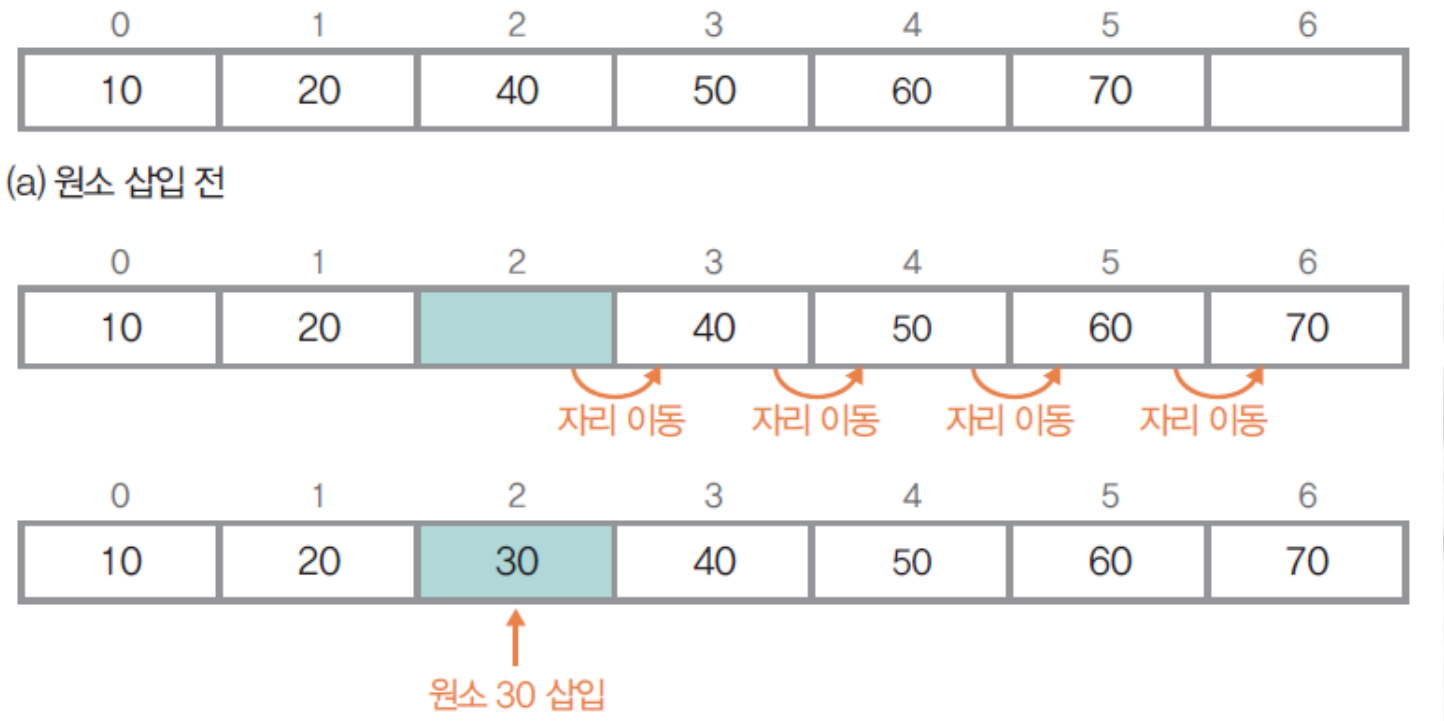
Array

- 배열
  - ✓ 배열의 마지막에 데이터 추가
    - 배열은 저장할 수 있는 데이터의 크기를 늘릴 수 없음
    - 데이터를 마지막에 추가하고자 하는 경우에는 기존 배열보다 데이터를 1개 더 저장할 수 있는 배열을 생성하고 데이터를 복사한 후 마지막 위치에 새로운 데이터를 추가
    - 10,20,30,40 이 저장된 배열의 마지막에 50 추가하기

	10	**20**	**30**	**40**	
1번 – 새로운 배열 생성					
2번 – 기존 데이터 복사	**10**	**20**	**30**	**40**	
3번 – 데이터를 추가	**10**	**20**	**30**	**40**	**50**

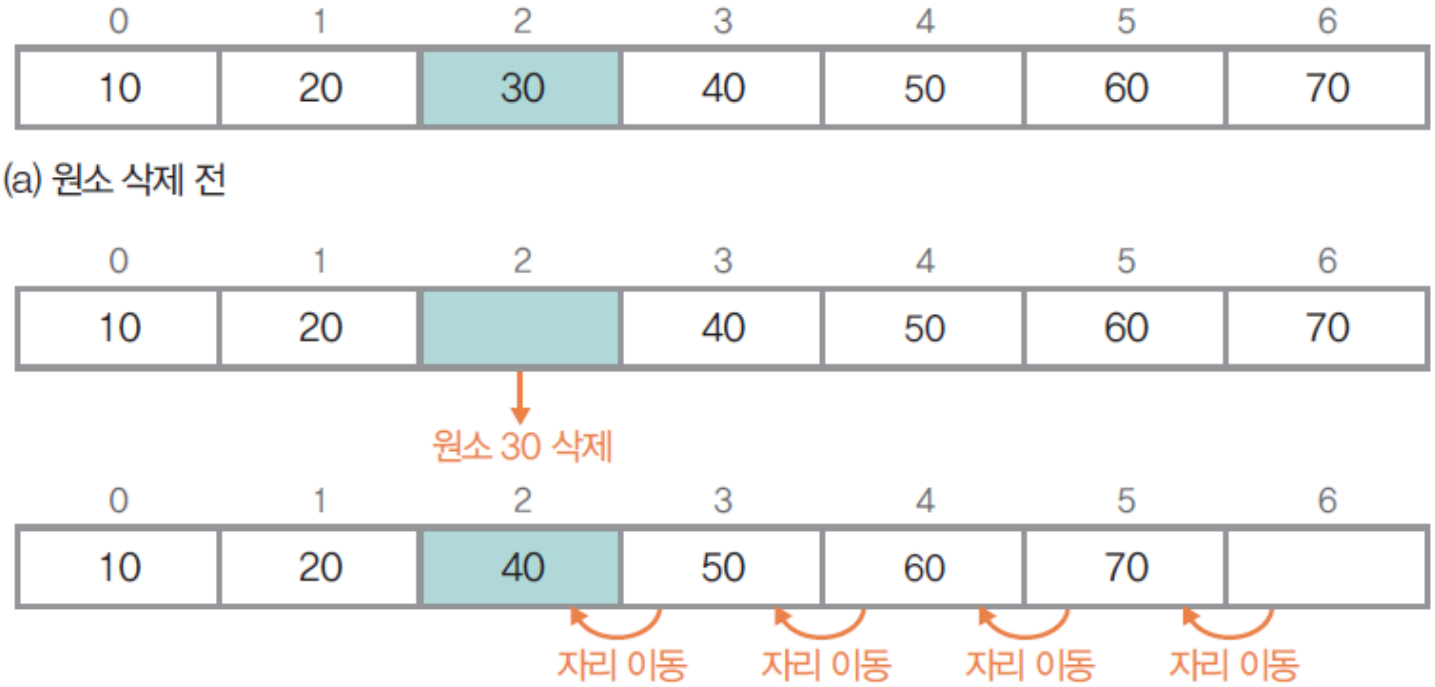
Array

- ❖ 배열
  - ✓ 배열의 중간 데이터 추가
    - 데이터를 중간에 추가하고자 하는 경우에는 기존 배열보다 데이터를 1개 더 저장할 수 있는 배열을 생성하고 삽입하고자 하는 위치 전까지의 데이터를 복사한 후 삽입할 데이터를 삽입한 후 나머지 데이터를 복사
    - 10,20,40,50,60,70 이 저장된 배열의 20 다음에 30을 삽입하기



Array

- 배열
  - ✓ 배열의 데이터 삭제
    - 중간에서 원소가 삭제되면, 그 이후의 원소들은 한 자리씩 자리를 앞으로 이동하여 물리적 순서를 논리적 순서와 일치시킴
    - 10,20,30,40,50,60,70 배열에서 데이터 삭제

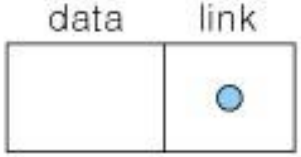


ArrayList

- ArrayList(Vector)
  - ✓ 자바에는 배열과 유사한 형태로 동작하지만 삽입과 삭제를 고려해서 여분의 공간을 확보하고 있는 java.util.ArrayList 나 Vector 라는 자료구조 클래스를 제공

Linked List

- Linked List
  - ✓ Data 와 Link로 구성된 노드들로 구성된 List
  - ✓ 배열이나 ArrayList는 Data 만으로 구성된 List
  - ✓ Linked List는 Data에 실제 자료를 저장하고 Link에 다음 데이터의 위치를 저장하는 구조로 데이터들이 물리적으로 연속된 공간에 저장될 필요가 없는 자료구조
  - ✓ 노드
    - 연결 자료구조에서 하나의 원소를 표현하기 위한 단위 구조

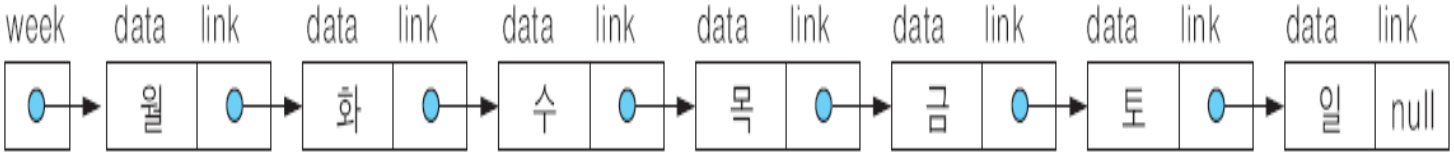


    - 데이터 필드(data field)
      - 원소의 값을 저장
      - 저장할 원소의 형태에 따라서 하나 이상의 필드로 구성
    - 링크 필드(link field)
      - 다음 노드의 주소를 저장
      - 포인터 변수를 사용하여 주소 값을 저장
      - 마지막 노드는 다음 데이터가 더 이상 없으므로 NULL

Linked List

❖ Linked List

✓ 저장 형태

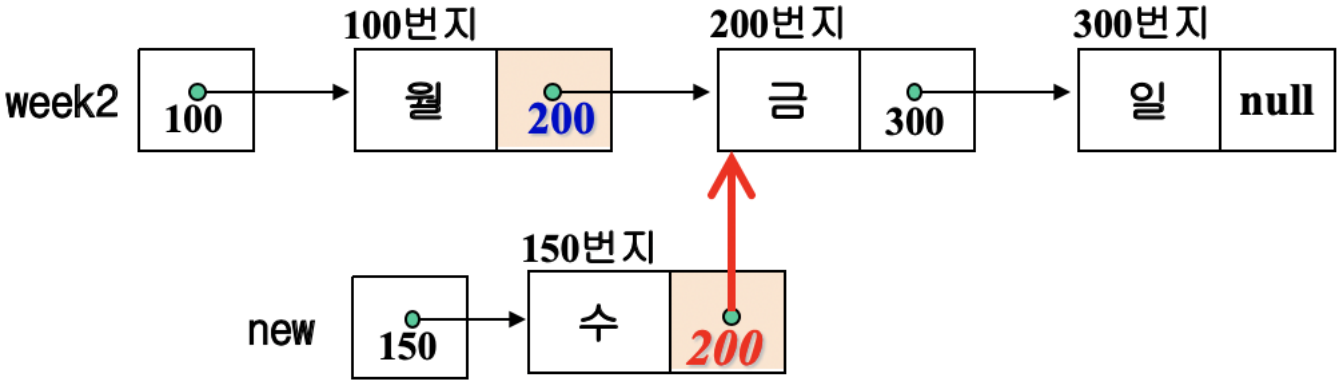


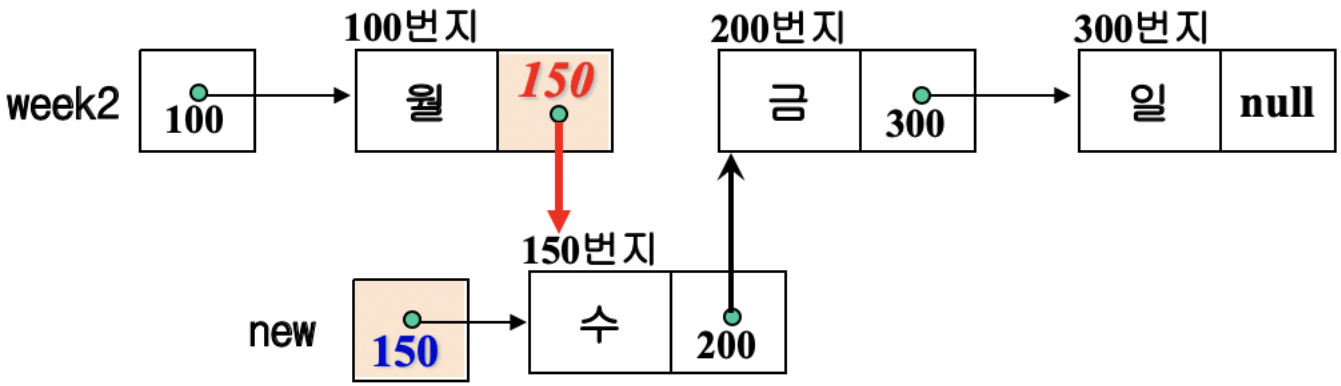
- 순서 상으로는 연결된 것처럼 보임
- 물리적으로는 떨어져 있을 수 있음
- Link에 의해 논리적으로 연결되어 있음
- 물리적으로 연속될 필요가 없기 때문에 새로운 원소를 추가할 경우 동적으로 원소를 생성하고 Link를 이용해서 연결만 해주면 됨

Linked List

❖ Linked List

✓ 데이터의 삽입 : 새로운 노드를 만들어서 Link만 수정

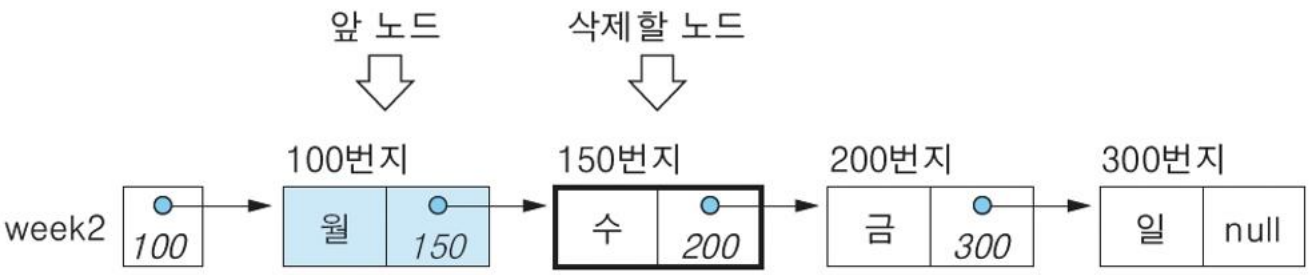


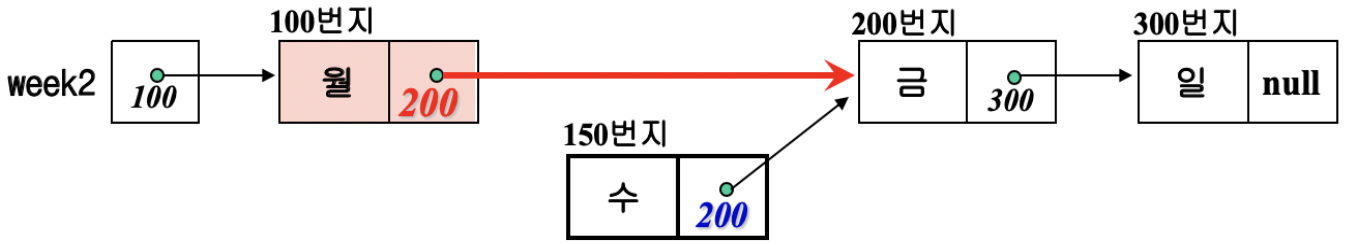


Linked List

❖ Linked List

✓ 데이터의 삭제 : 삭제할 데이터의 앞 요소를 찾아서 링크를 수정



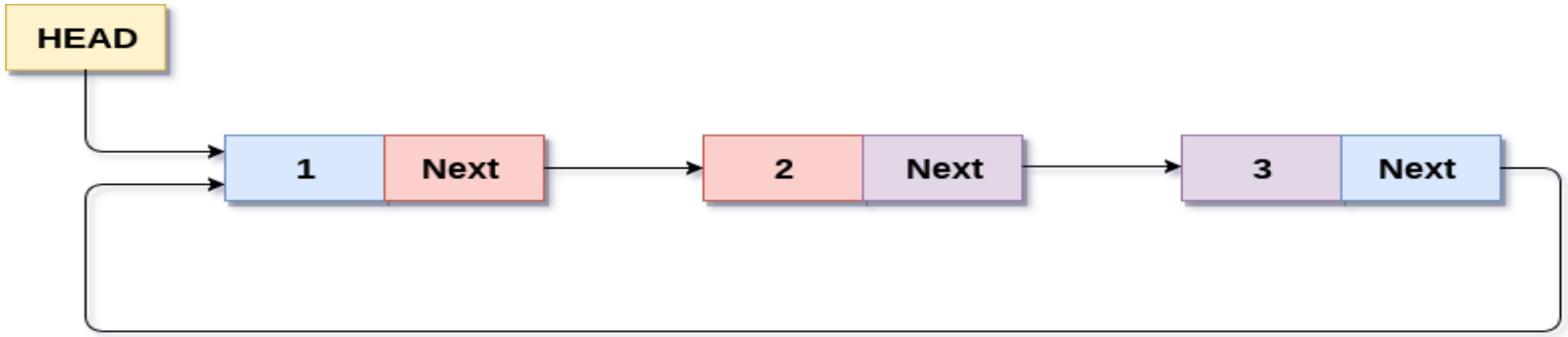


Linked List

- Linked List
  - ✓ 배열에 비교하여 가지는 장점
    - 새로운 원소를 중간에 삽입하거나 삭제하는 경우 원소의 이동 연산이 불필요하기 때문에 원소의 추가/삭제가 빈번히 발생한 경우라면 배열보다는 연결 리스트로 구현하는 것이 바람직
    - 배열은 연속적인 메모리 공간이 필요하므로 생성할 때 원소의 개수를 지정해야 하지만 연결 리스트는 원소의 개수 지정이 불필요
  - ✓ 배열과 비교하여 가지는 단점
    - 구현의 어려움 – 동적인 메모리 할당을 해야하기 때문에 구현의 비용이 높음
    - 메모리 관리와 관련하여 메모리 누수(Leak) 발생 가능성이 높음
    - 데이터를 탐색하는 비용이 높음 – 배열은 데이터를 연속적으로 저장했기 때문에 위치 인덱스를 이용해서 원하는 원소에 대한 접근이 한 번에 가능하지만 연결 리스트는 노드 사이에 연결된 Link를 통해 첫번째 원소부터 시작해서 위치-1 만큼 이동해야 하기 때문에 탐색 속도가 느림
  - ✓ 종류
    - Single Linked List: 하나의 link 만을 가지고 다음 데이터를 가리키는 형태
    - Circular Linked List: 마지막 노드의 link에 첫번째 데이터의 위치를 저장시키는 형태
    - Double Linked List: 2개의 link를 가지고 다음 데이터와 이전 데이터의 위치를 저장시키는 형태

Linked List

- Circular Linked List(환형 연결 리스트)
  - ✓ 리스트의 마지막 노드가 리스트의 첫번째 노드와 연결된 단순 연결 리스트
  - ✓ 단순 연결 리스트와 동일한 구조를 지니지만 리스트의 마지막 노드가 첫번째 노드와 연결되어 순환 구조를 지님
  - ✓ 단순 연결 리스트는 이전 노드에 접근하려면 HEAD로 이동한 후 접근해야 하지만 환형 연결 리스트는 계속 이동하다 보면 이전 노드에 접근이 가능
  - ✓ 이동하다가 다음 노드의 위치가 현재 노드이면 이전 노드가 됨

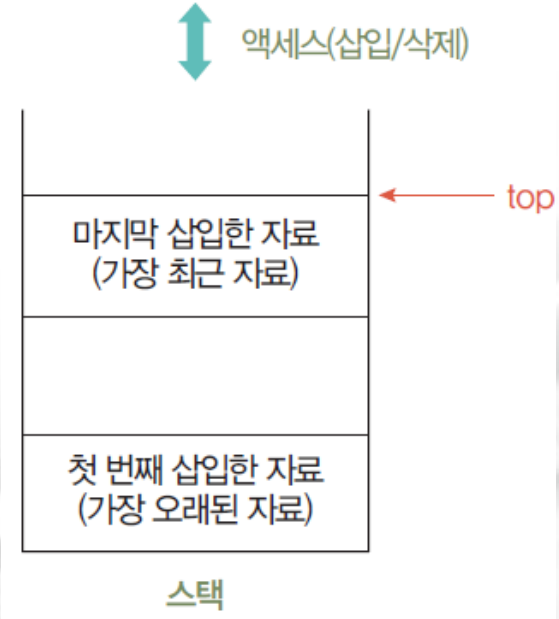


Linked List

- Double Linked List(이중 연결 리스트)
  - ✓ 하나의 노드가 이전 노드와 다음 노드를 가리키는 포인터를 2개 가지는 연결 리스트
  - ✓ 단순 연결 리스트와 환형 연결 리스트는 이전 노드를 접근하는 것이 어렵고 노드의 포인터가 소멸되면 리스트가 분할되는 현상이 발생
  - ✓ 양방향으로 접근이 가능해서 데이터의 접근은 편리하지만 추가적인 메모리 공간을 필요로 하는 단점이 있음

Stack

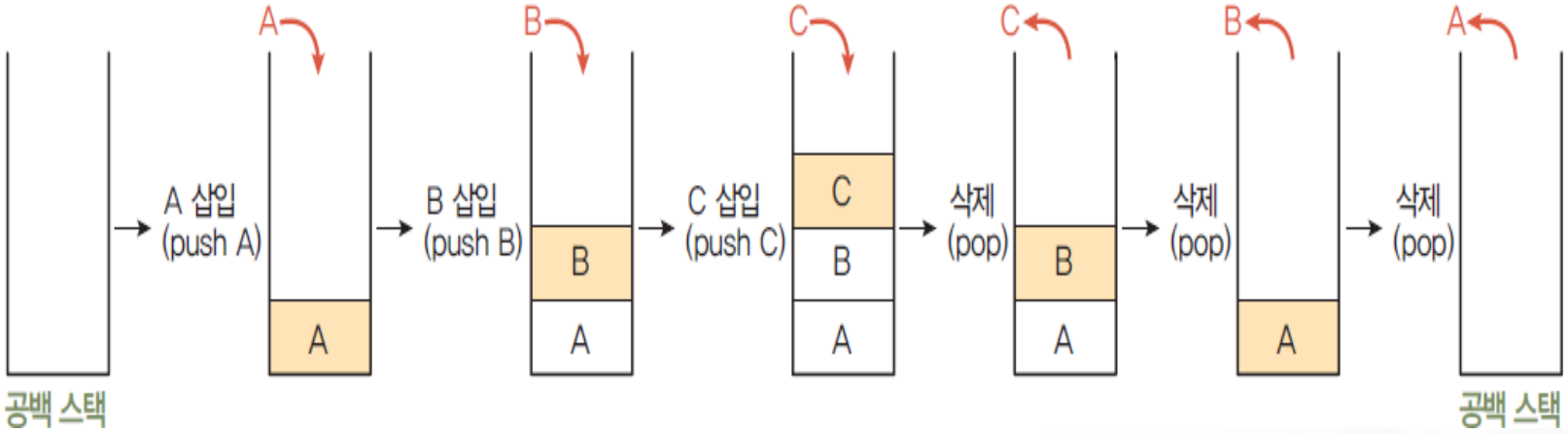
- Stack
  - ✓ 접시를 쌓듯이 자료를 차곡차곡 쌓아 올린 형태의 자료구조
  - ✓ 스택에 저장된 원소는 top 으로 정한 곳에서만 접근 가능
  - ✓ top의 위치에서만 원소를 삽입하므로 먼저 삽입한 원소는 밑에 쌓이고 나중에 삽입한 원소는 위에 쌓이는 구조
  - ✓ 마지막에 삽입(Last-In)한 원소는 맨 위에 쌓여 있다가 가장 먼저 삭제(First-Out)됨 ☞ 후입선출 구조 (LIFO, Last-In-First-Out)



Stack

❖ Stack

✓ 스택에서의 원소 삽입(push)/삭제(pop) 과정 - 스택에 원소 A, B, C를 순서대로 삽입하고 한번 삭제하는 연산 과정 동안의 스택 변화



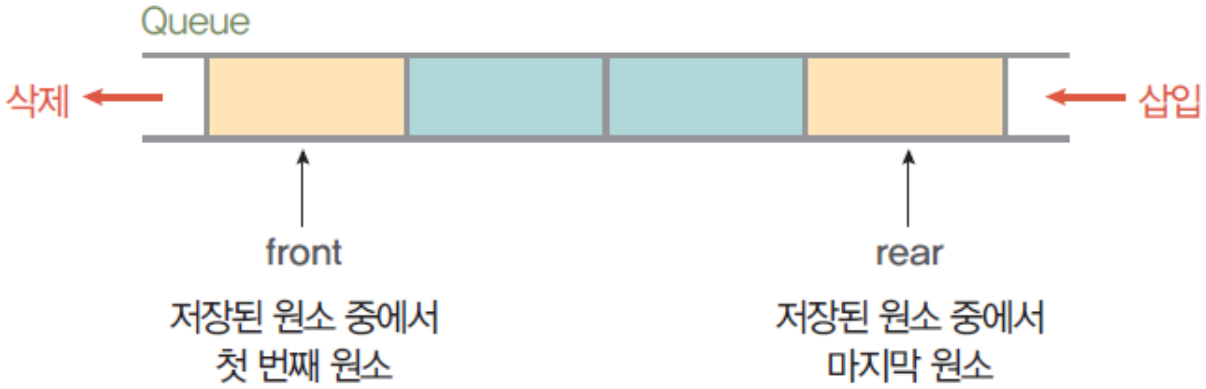
✓ 데이터가 없는 상태에서 pop을 하면 Underflow, 데이터가 전부 저장된 상태에서 데이터를 push 하면 Overflow 라는 예외가 발생

✓ Stack의 용도

- 운영체제(OS: Operating System): 프로그램에서 사용되는 함수들을 스택 자료형에 저장하여 사용
- 컴파일러(Compiler): 수학 기호들을 기계어로 변환 시 사용(괄호 등을 매칭하는 경우)
- 자바 가상 머신(JVM: Java Virtual Machine): JVM 내에서 메서드가 실행, 종료될 때 스택 프레임을 이용하여 관리

Queue

- Queue
  - ✓ 큐는 뒤(rear)에서는 삽입만 하고 앞(front)에서는 삭제만 할 수 있는 구조
  - ✓ 삽입한 순서대로 원소가 나열되어 가장 먼저 삽입(First-In)한 원소는 맨 앞에 있다가 가장 먼저 삭제(First-Out)

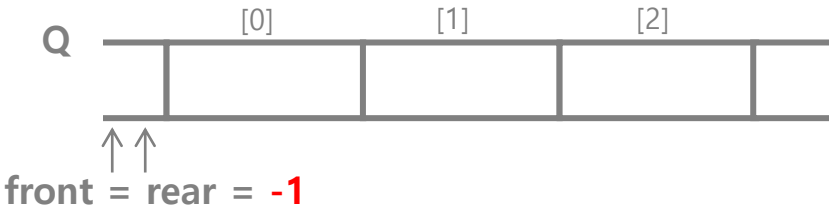


  - ✓ 큐의 연산
    - 삽입 : En-Queue
    - 삭제 : De-Queue

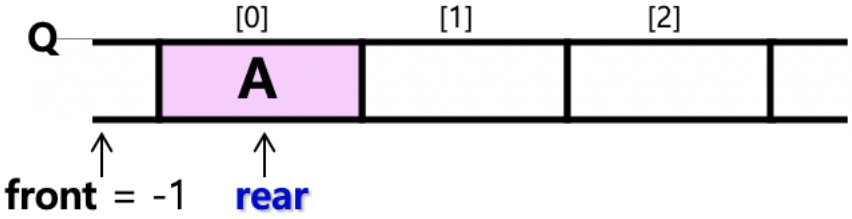
Queue

❖ Queue

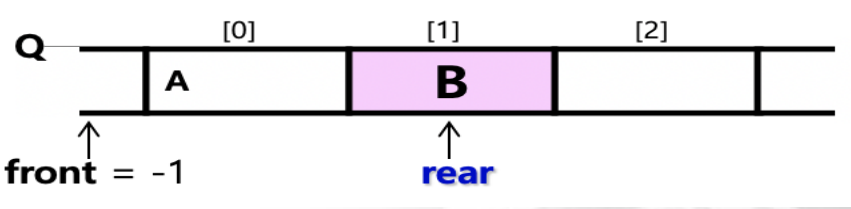
✓ 공백 큐 생성 : createQueue()



✓ En-Queue(Q, A)

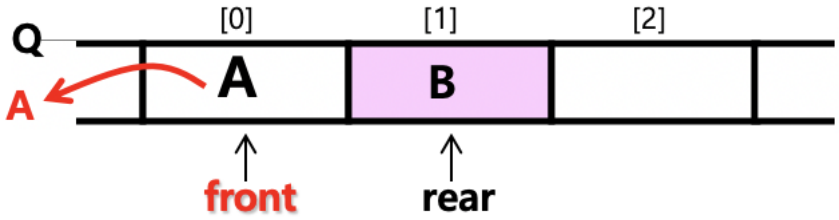


✓ En-Queue (Q, B)

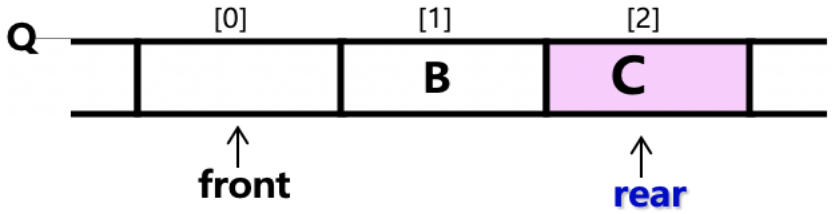


Queue

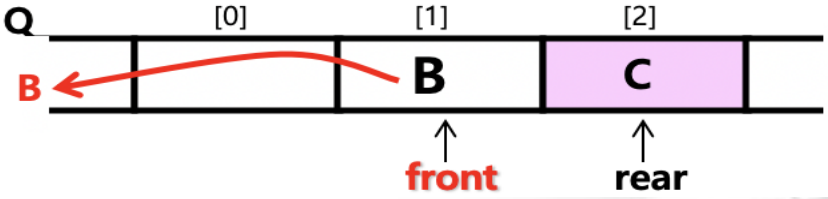
- Queue
  - ✓ De-Queue(Q)



  - ✓ En-Queue(Q, C);

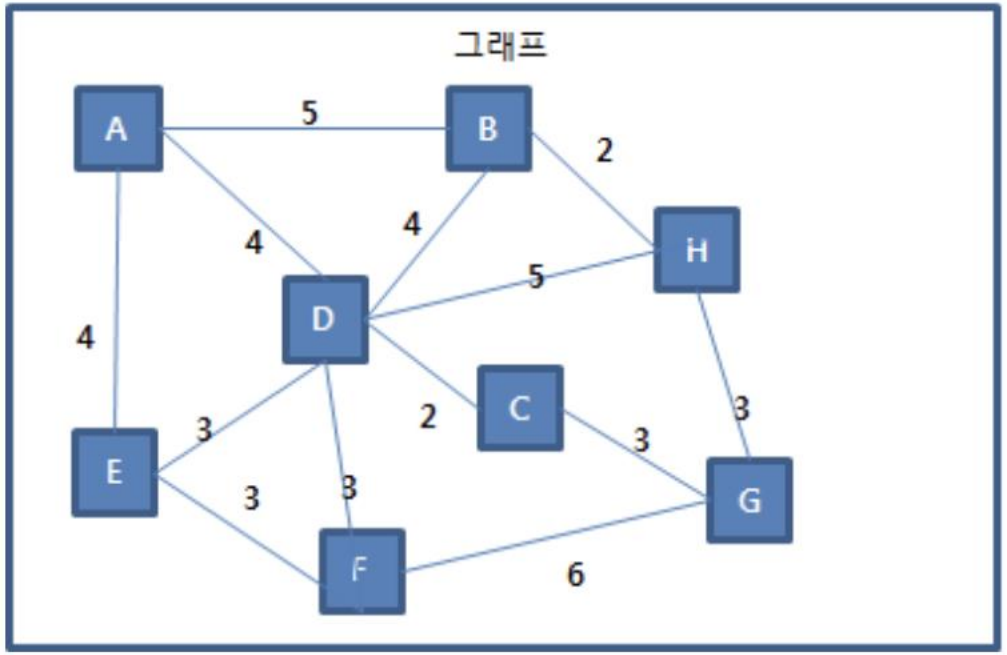


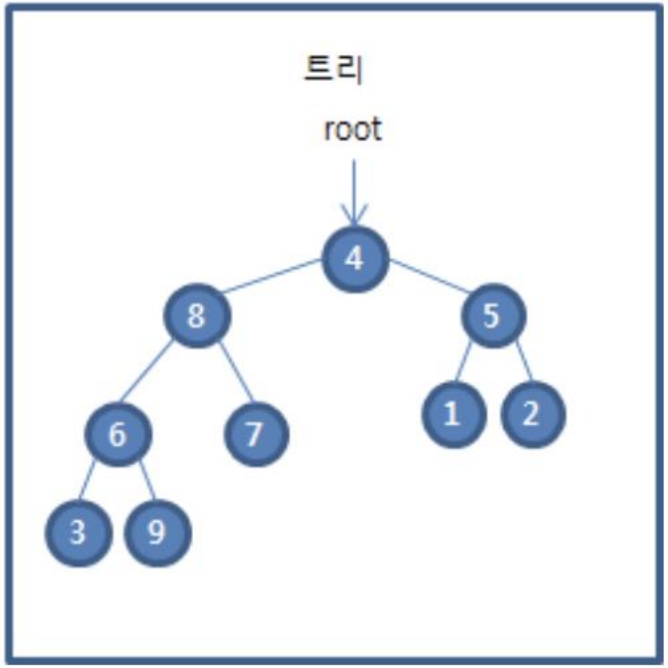
  - ✓ De-Queue(Q)



비선형 자료구조

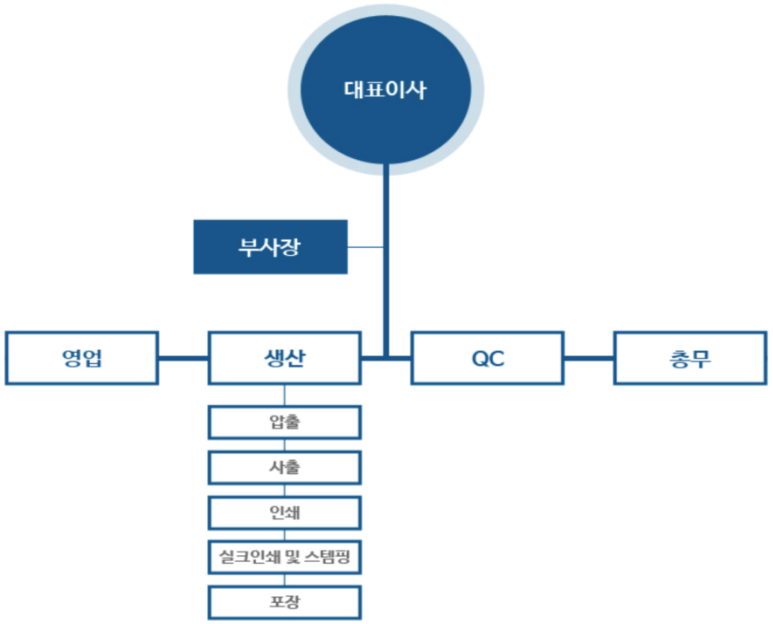
- 비선형 자료구조
  - ✓ 하나의 자료 뒤에 여러 개의 자료가 존재할 수 있는 구조
  - ✓ 비선형 자료구조의 종류
    - 트리: 방향성 있고 사이클이 존재하지 않으며 고립 영역이 없는 그래프
    - 그래프: 정점과 간선의 집합

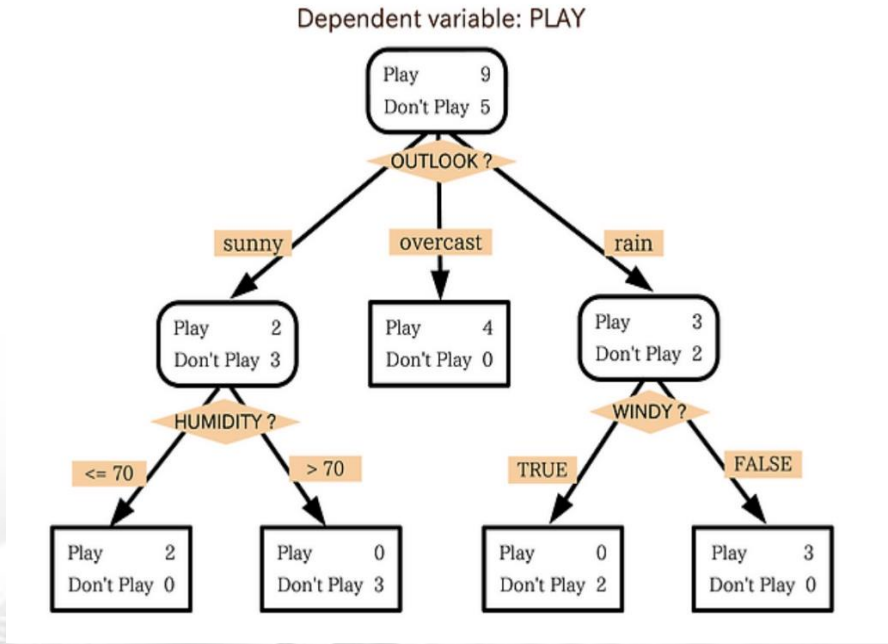




Tree

- Tree
  - ✓ 계층 구조로 자료를 저장하는 구조
  - ✓ 트리를 구성하는 노드가 부모-자식(Parent – Child) 관계라는 의미
  - ✓ 부모 노드 하나에 여러 개의 자식 노드들이 연결되는 구조
  - ✓ 회사의 조직도, 컴퓨터의 디렉토리 구조 등이 대표적이며 머신러닝에 사용되는 의사결정 트리 등이 있음



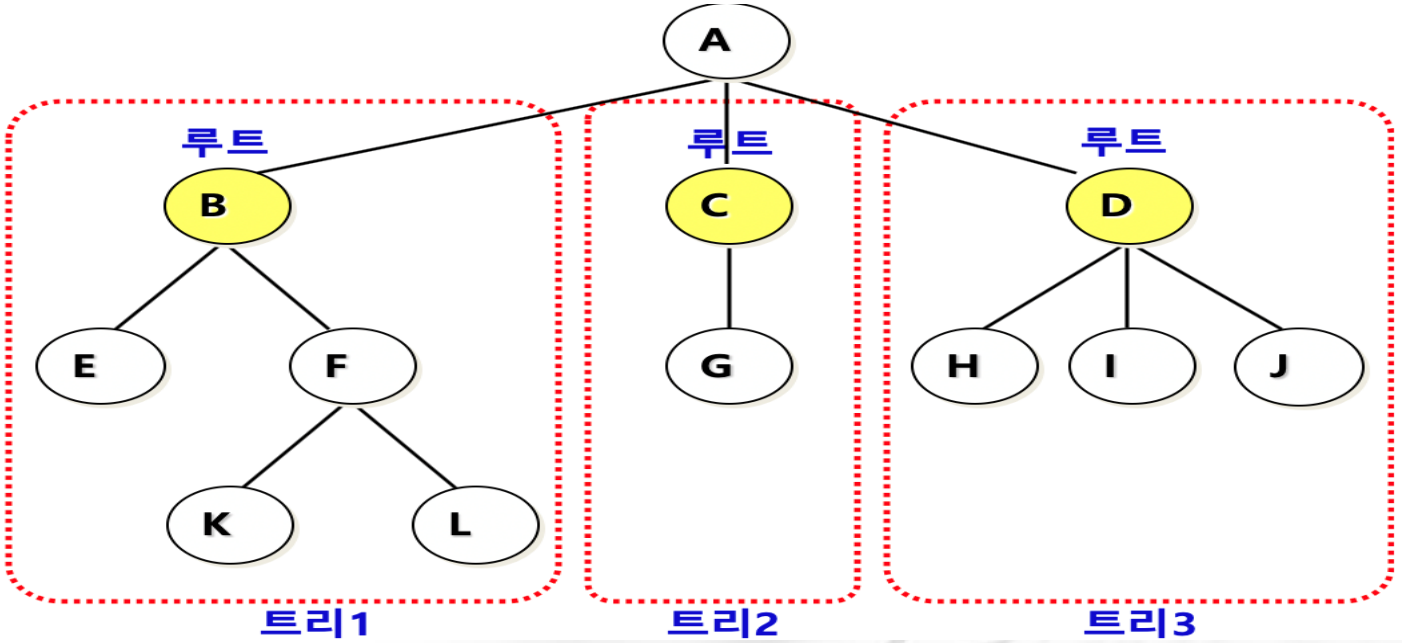


Tree

- Tree
  - ✓ 노드(Node)와 간선(Edge)의 집합
  - ✓ 조직도에서는 부서가 노드가 되며 의사결정 트리에서는 의사결정을 위한 조건이 노드
  - ✓ 간선은 이러한 노드들 사이의 부모-자식 관계를 정의
  - ✓ 노드의 종류
    - Root: 트리의 첫번째 노드
    - Leaf(Terminal): 자식 노드가 없는 노드
    - Internal: 자식 노드가 있는 노드
  - ✓ 노드 사이의 관계
    - Parent: 간선으로 연결된 상위 노드
    - Child: 간선으로 연결된 하위 노드
    - Ancestor(선조): root에서 parent까지의 경로 상에 있는 모든 노드
    - Descendant(후손): 특정 노드의 아래에 있는 모든 노드
    - Sibiling(형제): 같은 부모 노드의 자식 노드

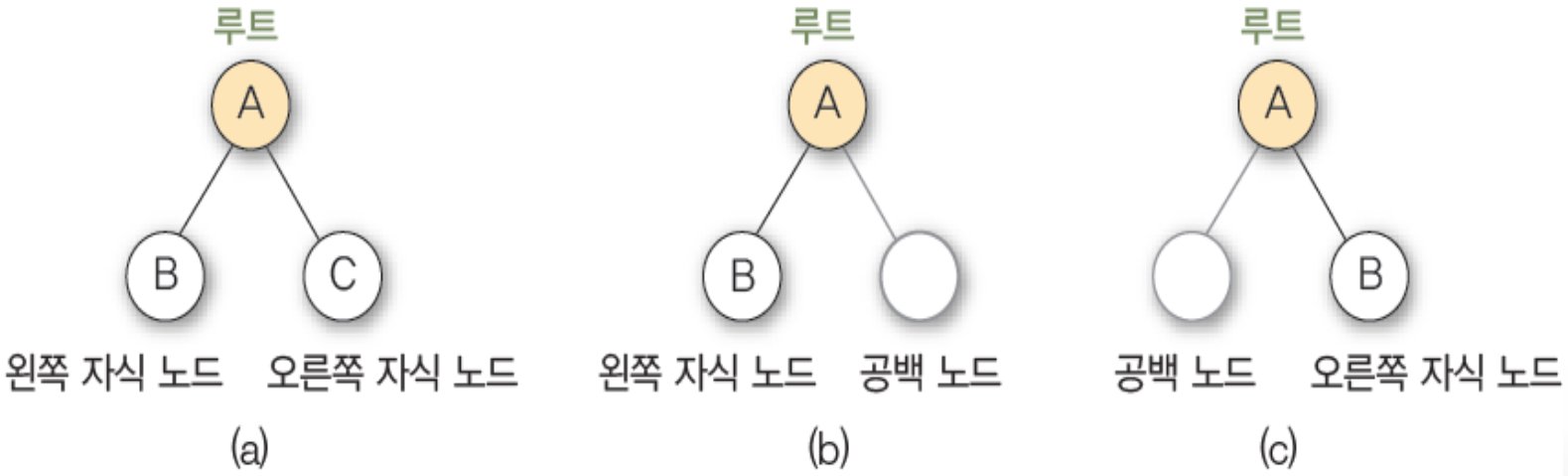
Tree

- Tree
  - ✓ 노드의 속성
    - Level: 루트 노드부터의 거리
    - Height: 루트 노드로부터 가장 먼 거리에 있는 자식 노드의 높이에 1을 더한 값
    - Degree: 한 노드가 가지는 자식 노드의 개수
  - ✓ Subtree: 트리에 속한 노드들의 부분 집합
  - ✓ Forest: 특정 노드가 제거되면서 만들어지는 Subtree 의 집합



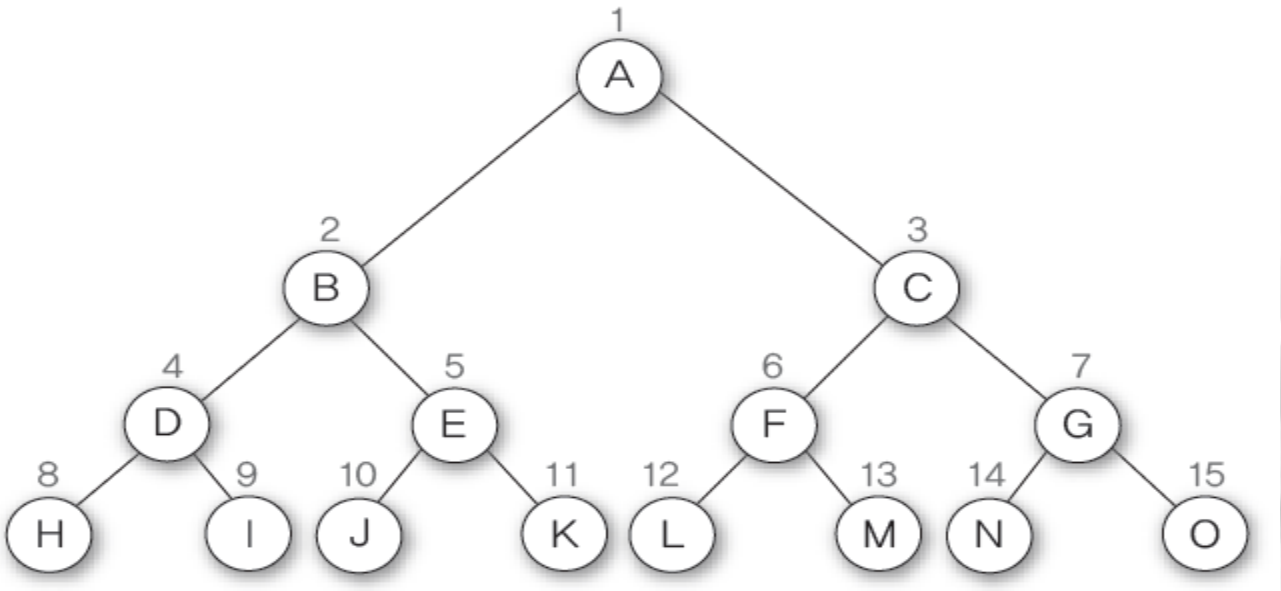
Tree

- Binary Tree
  - ✓ 모든 노드의 차수가 2 이하인 트리



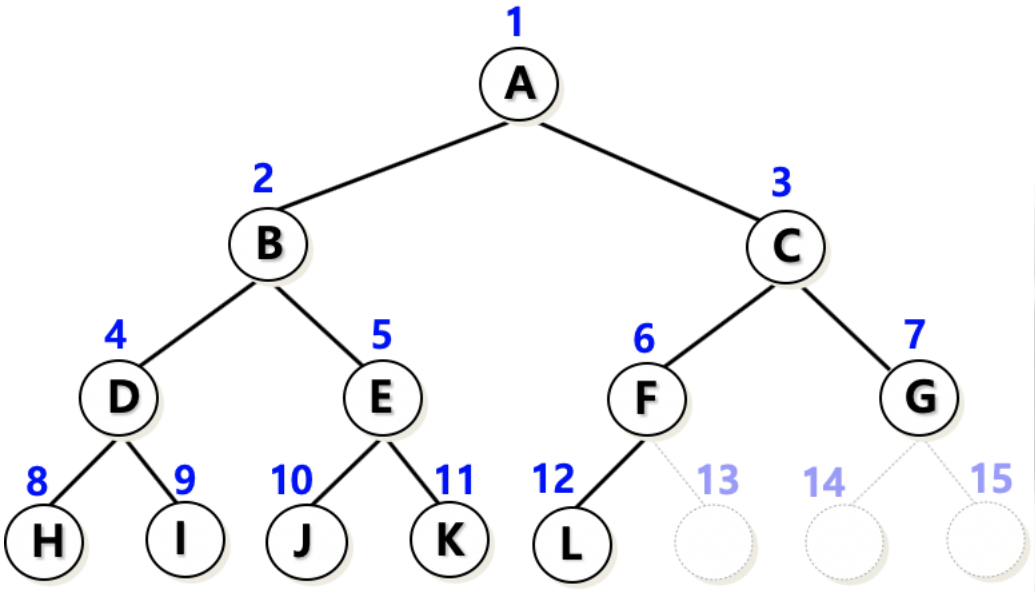
Tree

- ❖ Binary Tree
  - ✓ Binary Tree 의 종류
    - Full Binary Tree(포화 이진 트리)
      - 모든 레벨의 노드가 꽉 차있는 이진 트리
      - 높이가 h일 때 최대의 노드 개수인 $(2^{h+1}-1)$의 노드를 가진 이진 트리
      - 루트를 1번으로 하여 $2^{h+1}-1$까지 정해진 위치에 대한 노드 번호를 가짐



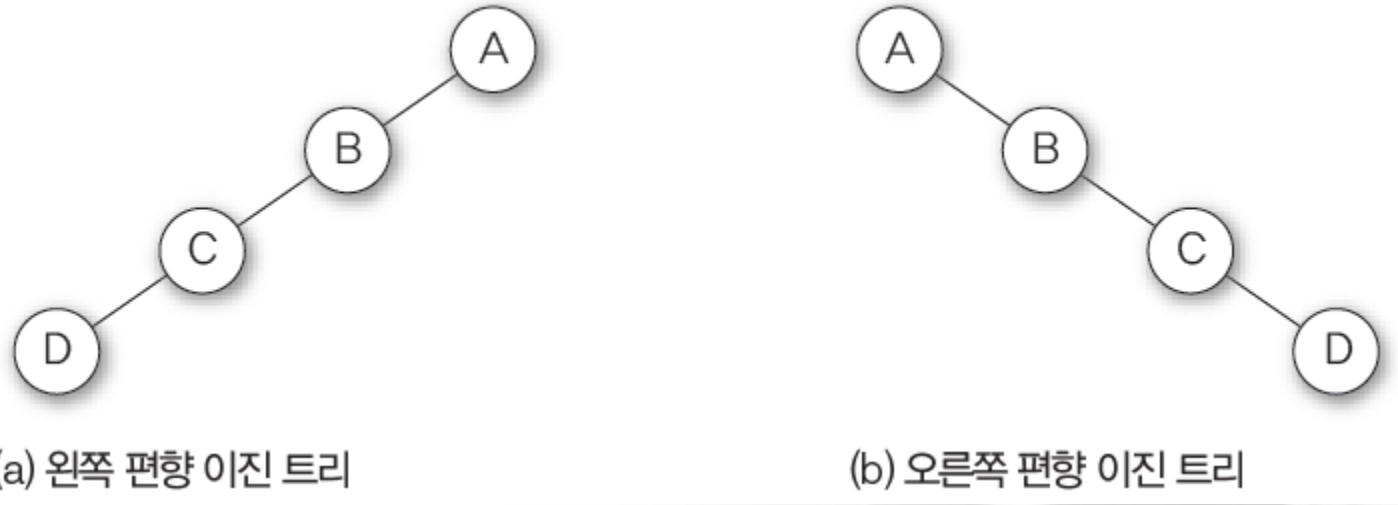
Tree

- Binary Tree
  - ✓ Binary Tree 의 종류
    - Complete Binary Tree(완전 이진 트리)
      - 높이가 h이고 노드 수가 n개일 때 (단, $n < 2^{h+1}-1$)
      - 노드 위치가 포화 이진 트리에서의 노드 1번부터 n번까지의 위치와 완전히 일치하는 이진 트리
      - 완전 이진 트리에서는 (n+1)번부터 ($2^{h+1}-1$)번까지 노드는 모두 공백 노드



Tree

- ❖ Binary Tree
  - ✓ Binary Tree 의 종류
    - ○ Skewed Binary Tree(편향 이진 트리)
      - ▪ 같은 높이의 이진 트리 중에서 최소 개수의 노드 개수를 가지면서 왼쪽 또는 오른쪽 서브트리만을 가진 이진 트리



(a) 왼쪽 편향 이진 트리 (b) 오른쪽 편향 이진 트리

Tree

- ❖ Binary Tree 순회
  - ✓ 전위 순회(preorder traversal)
    - ○ D(현재 노드) → L(왼쪽 서브 트리) → R(오른쪽 서브 트리) 순서로 처리

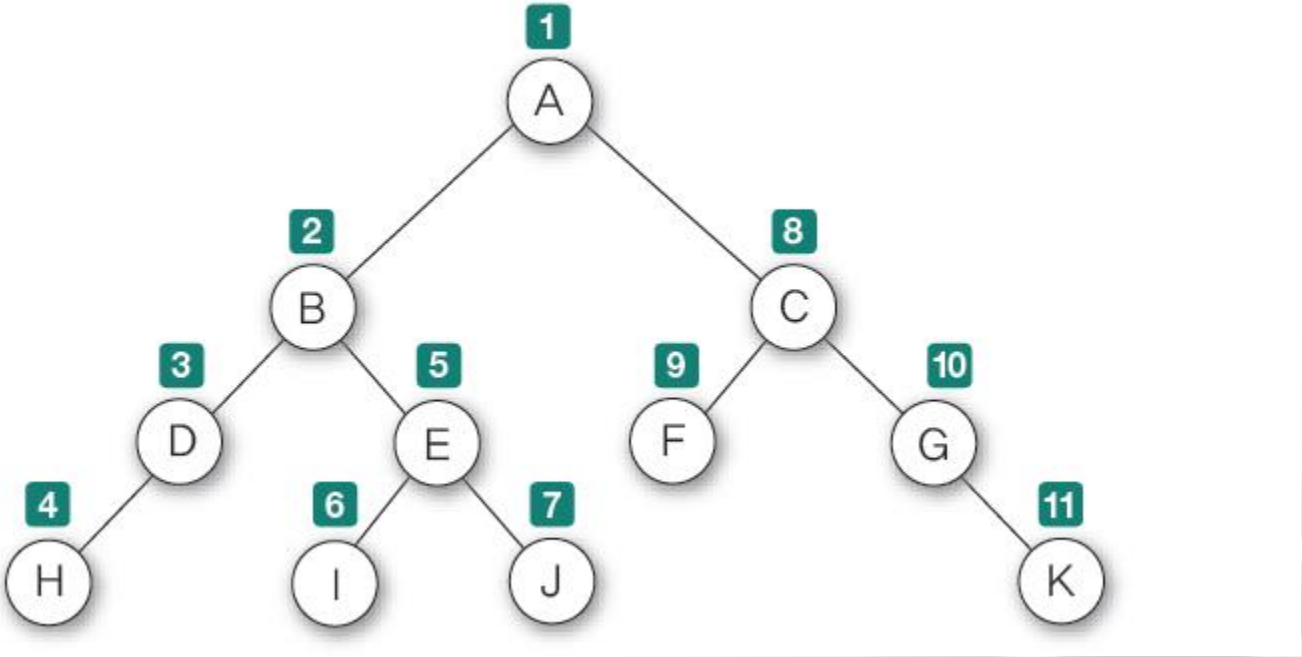
❶ 작업 D : 현재 노드 n을 처리한다.

❷ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.

❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

Tree

- Binary Tree 순회
  - ✓ 전위 순회(preorder traversal)



Tree

❖ Binary Tree 순회

✓ 중위 순회(Inorder traversal)

- L → D → R 순서로, 현재 노드를 방문하는 작업 D를 작업 L과 작업 R의 중간에 수행

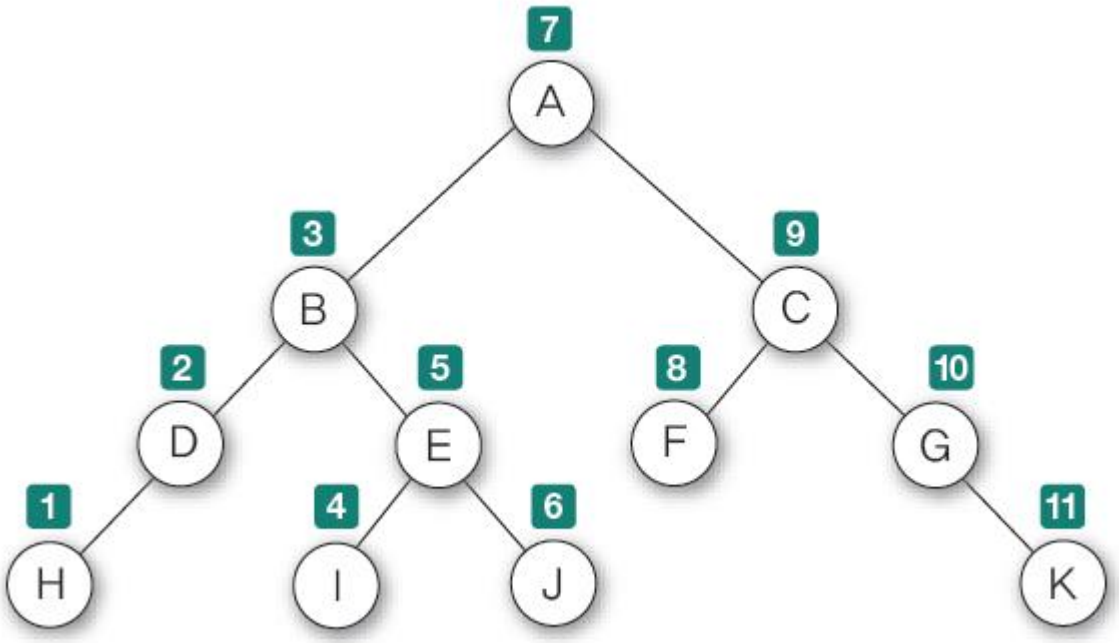
❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.

❷ 작업 D : 현재 노드 n을 처리한다.

❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

Tree

- Binary Tree 순회
  - ✓ 중위 순회(Inorder traversal)



Tree

❖ Binary Tree 순회

✓ 후위 순회(Postorder traversal)

- L-R-D 순서로 현재 노드를 방문하는 D 작업을 가장 나중에 수행

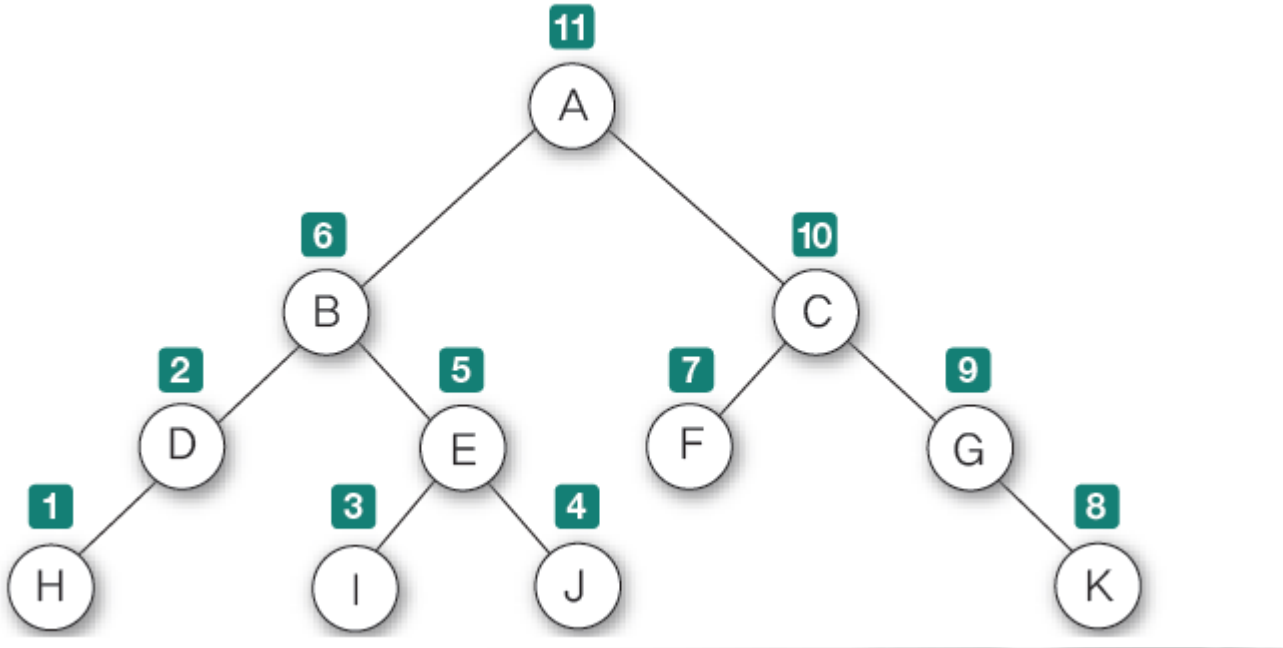
❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.

❷ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

❸ 작업 D : 현재 노드 n을 처리한다.

Tree

- Binary Tree 순회
  - ✓ 후위 순회(Postorder traversal)

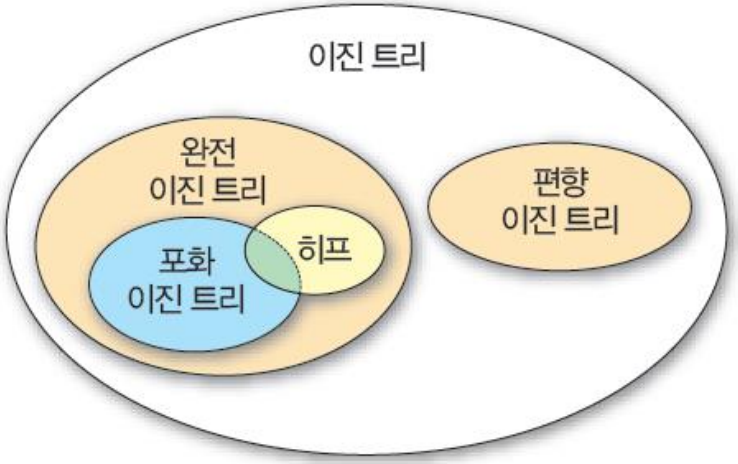


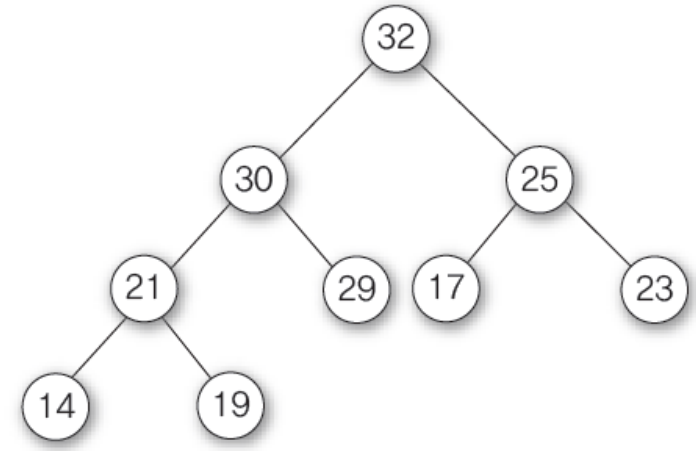
Tree

- Heap
  - ✓ 완전 이진 트리에 있는 노드 중에서 키값이 가장 큰 노드나 키값이 가장 작은 노드를 찾기 위해서 만든 자료구조
  - ✓ 루트 노드가 그 트리의 최댓값 혹은 최솟값을 갖는 트리
  - ✓ 최대 히프(max heap)
    - 키값이 가장 큰 노드를 찾기 위한 완전 이진 트리
    - {부모노드의 키값 ≥ 자식노드의 키값}
    - 루트 노드 : 키값이 가장 큰 노드
  - ✓ 최소 히프min heap
    - 키값이 가장 작은 노드를 찾기 위한 완전 이진 트리
    - {부모노드의 키값 ≤ 자식노드의 키값}
    - 루트 노드 : 키값이 가장 작은 노드

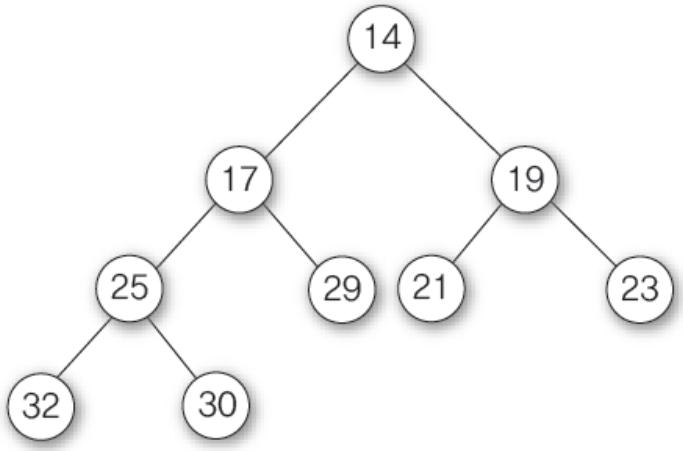
Tree

- Heap





(a) 최대 히프



(b) 최소 히프

Tree

- Heap
  - ✓ HEAP의 삽입 연산
    - ○ 트리의 마지막 자리에 임시 저장
    - ○ 부모 노드와 키 값 비교 및 이동
      - ▪ 새로 추가한 노드의 키 값이 부모 노드보다 크다면 서로 교환
      - ▪ 이 작업을 다시 수행 – 부모 노드보다 작을 때 까지 수행
  - ✓ HEAP의 삭제 연산 – 반드시 루트 노드만 삭제
    - ○ 루트 노드를 삭제
    - ○ 트리의 마지막 노드를 루트로 이동
    - ○ 루트 노드를 자식 노드들과 비교해서 자식 노드가 더 크다면 자식 노드와 위치 변경 – 이 때 2개의 자식 모두 루트보다 크다면 둘 중에 더 큰 노드와 교체
    - ○ 위의 작업을 반복

수식 표기법

- Infix
  - ✓ 일반적인 수식의 표기법은 infix
  - ✓ 두개의 피연산자 사이에 연산자가 존재하는 형태
  - ✓ 연산자의 우선 순위에 따라 수행되며 이해하기 쉬움
  - ✓ 무엇보다 일반적인 사용법이기 때문에 직관적으로 받아들일 수 있음

    (a + b) * c / d + e

수식 표기법

- postfix
  - ✓ 연산자를 피연산자의 뒷쪽에 표시하는 방법
  - ✓ Stack을 사용한 컴퓨터의 계산을 위해서 postfix와 prefix를 고안
  - ✓ 변형 방법은 괄호로 묶어서 괄호를 하나씩 지우면서 연산자를 뒤로 빼내면 된다.

```
(a + b) * c / d + e
> (((a + b) * c) / d) + e
> (((a + b) * c) / d) e  +
> ((a + b) * c) d / e  +
> (a + b) c * d / e  +
> a

 b + c * d / e  +
```

수식 표기법

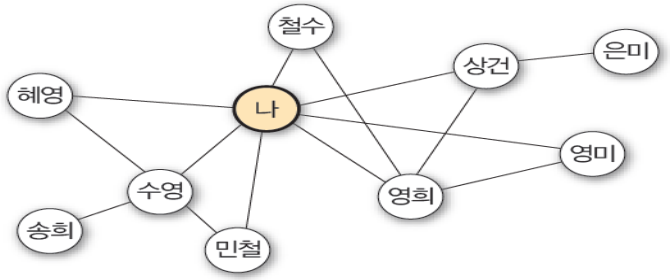
- prefix
  - ✓ prefix는 연산자를 앞으로 배치하는 방식이다.
  - ✓ 변환 방법은 postfix와 큰 차이 없이 괄호의 앞으로 연산자를 보내면 됨

(a + b) * c / d – e

> (((a + b) * c) / d) – e
> – (((a + b) * c) / d) e
> – / ((a + b) * c) d e
> – / * (a + b) c d e
> – / * + a b c d e

Graph

- Graph
  - ✓ 연결되어 있는 원소 사이의 다:다 관계를 표현하는 자료구조
  - ✓ 표현능력이 우수하여 현실 세계의 다양한 문제를 효과적으로 모델링하기 위한 자료구조
  - ✓ 리스트, 스택, 큐 등의 선형자료구조는 데이터를 순차적으로 저장하여 효율적으로 데이터를 저장하는 것이 목적이고 트리는 계층 관계를 표현하기 위한 자료구조인데 트리는 계층구조가 아닌 일반적인 관계는 나타낼 수 없음
  - ✓ 트리는 순환 구조도 표현할 수 없음
  - ✓ 객체를 나타내는 정점(vertex)과 객체를 연결하는 간선(edge)의 집합
  - ✓ G = (V, E)
    - V는 그래프에 있는 정점들의 집합
    - E는 정점을 연결하는 간선들의 집합

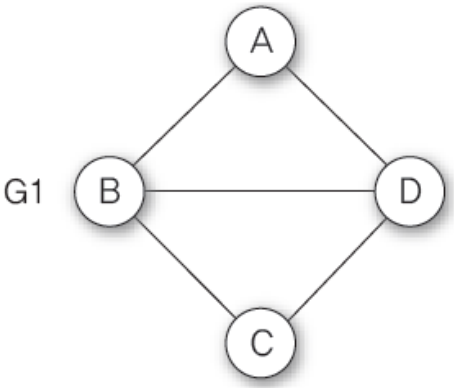


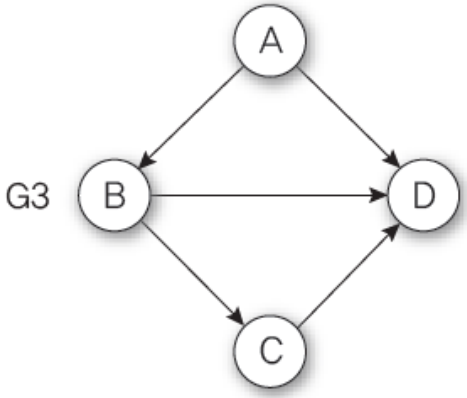
Graph

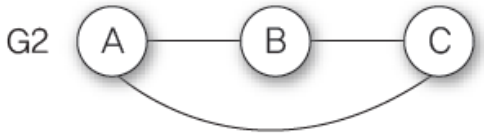
- 그래프(Graph)
  - ✓ 단순히 노드(N, node)와 그 노드를 연결하는 간선(E, edge)을 하나로 모아 놓은 자료 구조
  - ✓ 연결되어 있는 객체 간의 관계를 표현할 수 있는 자료 구조
  - ✓ 방향이 있는 방향성 그래프와 방향이 없는 무방향 그래프로 분류
  - ✓ 용어
    - 정점(vertex): 위치라는 개념. (node 라고도 부름)
    - 간선(edge): 위치 간의 관계. 즉, 노드를 연결하는 선 (link, branch 라고도 부름)
    - 인접 정점(adjacent vertex): 간선에 의 해 직접 연결된 정점
    - 정점의 차수(degree): 무방향 그래프에서 하나의 정점에 인접한 정점의 수
    - 무방향 그래프에 존재하는 정점의 모든 차수의 합 = 그래프의 간선 수의 2배
    - 진입 차수(in-degree): 방향 그래프에서 외부에서 오는 간선의 수 (내차수 라고도 부름)
    - 진출 차수(out-degree): 방향 그래픙에서 외부로 향하는 간선의 수 (외차수 라고도 부름)
    - 방향 그래프에 있는 정점의 진입 차수 또는 진출 차수의 합 = 방향 그래프의 간선의 수(내차수 + 외차수)
    - 경로 길이(path length): 경로를 구성하는 데 사용된 간선의 수
    - 단순 경로(simple path): 경로 중에서 반복되는 정점이 없는 경우
    - 사이클(cycle): 단순 경로의 시작 정점과 종료 정점이 동일한 경우

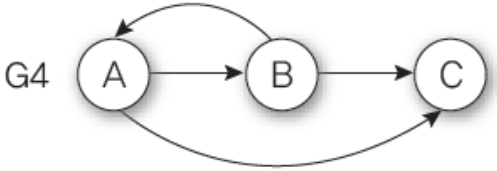
Graph

- Graph
  - ✓ 그래프의 종류
    - ○ 간선의 방향성
      - ▪ 무방향 그래프: 간선에 방향이 없는 그래프
      - ▪ 방향 그래프: 간선에 방향이 있는 그래프



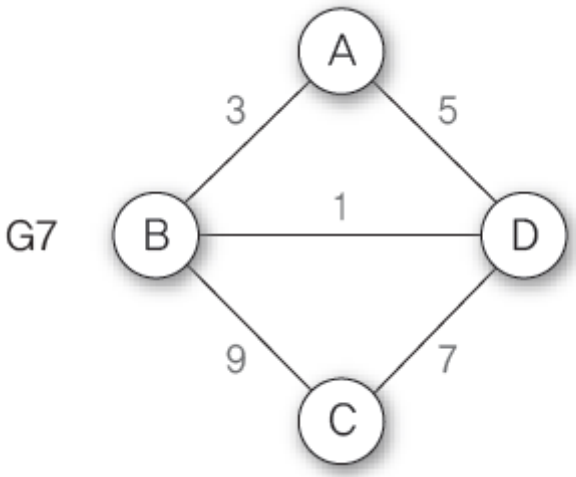


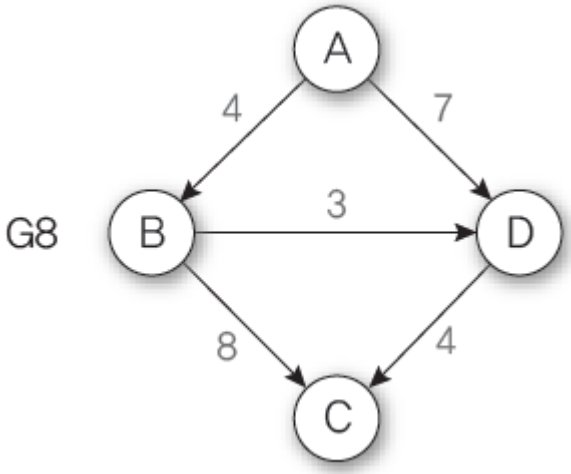




Graph

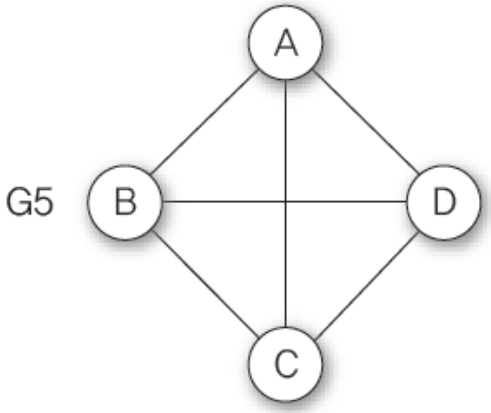
- Graph
  - ✓ 그래프의 종류
    - 가중 그래프: 간선에 가중치가 할당된 그래프

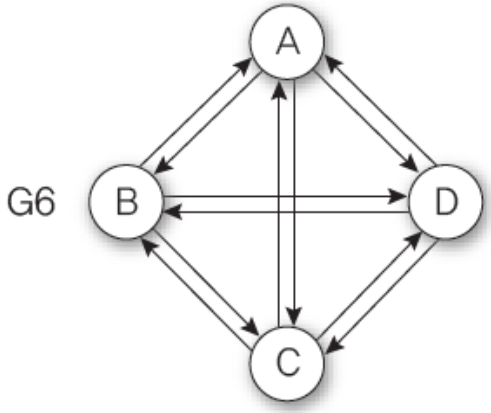




Graph

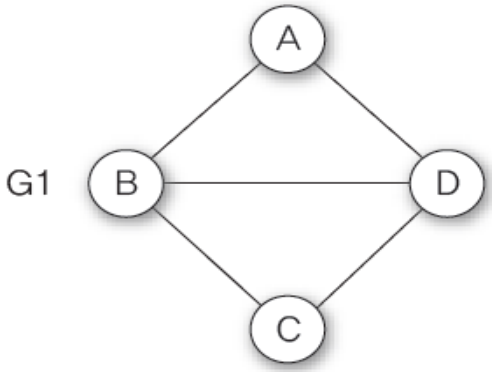
- Graph
  - ✓ 그래프의 종류
    - ○ 구조적 특징
      - ▪ 완전 그래프: 연결 가능한 최대 간선 수를 가진 그래프

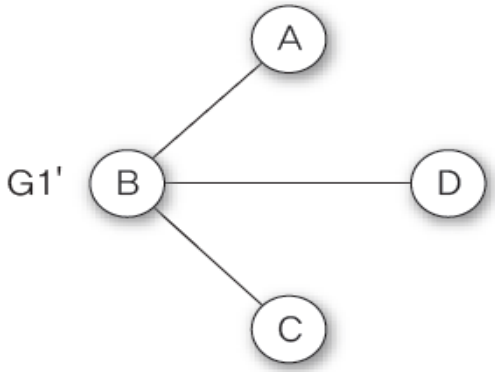


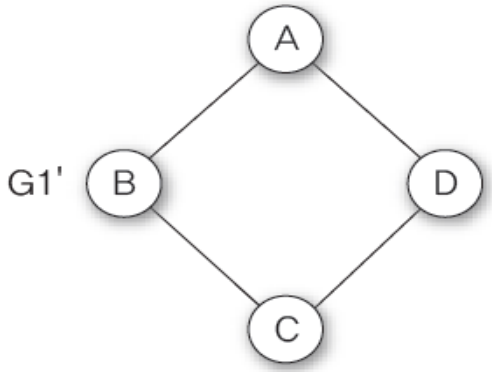


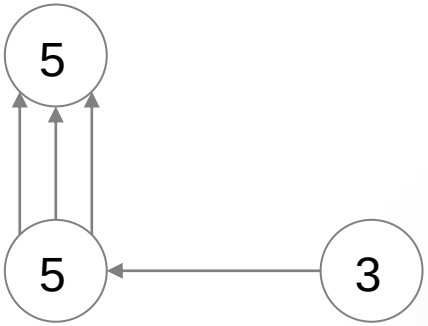
Graph

- Graph
  - ✓ 그래프의 종류
    - ○ 구조적 특징
      - 부분 그래프: 원래의 그래프에서 일부의 노드나 간선을 제외하여 만든 그래프
      - 다중 그래프: 중복된 간선을 포함하는 그래프









Graph

- Graph
  - ✓ 그래프의 용어
    - Adjacent(인접): 두 개의 노드를 연결하는 간선이 존재하는 경우
    - Incident(부속): 간선 (Vi, Vj)는 정점 Vi와 Vj에 부속(incident)되어 있다고 함
    - Degree(차수): 노드에 부속된 간선의 수
    - Path(경로): 그래프에서 간선을 따라 갈 수 있는 길을 순서대로 나열한 것으로 정점 Vi에서 Vj까지 간선으로 연결된 정점을 순서대로 나열한 리스트
    - 그래프의 동일성: 그래프의 모양은 다르더라도 노드와 간선의 집합이 같으면 동일한 그래프
    - Loop: 그래프의 임의의 노드에서 자기 자신으로 이어지는 간선

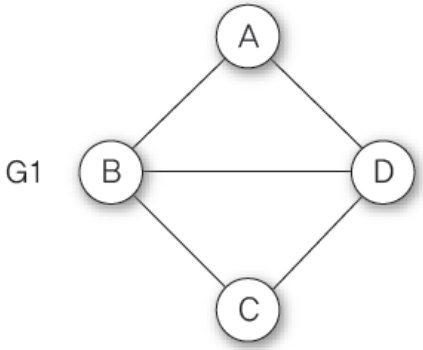
Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 : 인접 행렬
    - 행렬에 대한 2차원 배열을 사용하는 순차 자료구조 방법
    - 그래프의 두 정점을 연결한 간선의 유무를 행렬로 저장
    - n개의 정점을 가진 그래프 : n x n 정방행렬
    - 행렬의 행번호와 열번호 : 그래프의 정점
    - 행렬 값 : 두 정점이 인접되어 있으면 1, 인접되어 있지 않으면 0
    - 무방향 그래프의 인접 행렬
      - 행 i의 합 = 열 i의 합 = 정점 i의 차수
    - 방향 그래프의 인접 행렬
      - 행 i의 합 = 정점 i의 진출차수
      - 열 i의 합 = 정점 i의 진입차수

Graph

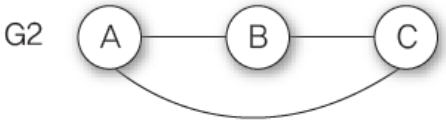
❖ Graph 구현

✓ 순차 자료구조를 이용한 그래프의 구현 : 인접 행렬



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

1+0+1+1=3 정점 B의 차수

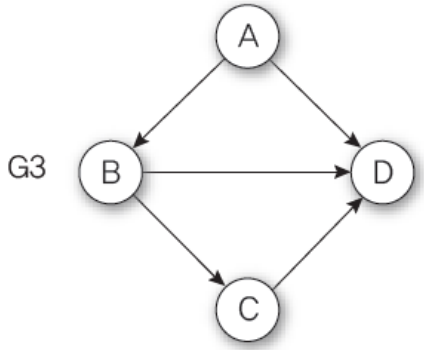


	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Graph

❖ Graph 구현

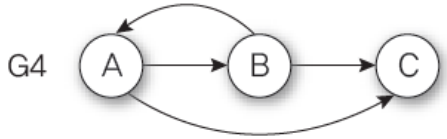
✓ 순차 자료구조를 이용한 그래프의 구현 : 인접 행렬



	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

0+0+1+1=2 정점 B의 진출 차수

1+0+0+0=1 정점 B의 진입 차수



	A	B	C
A	0	1	1
B	1	0	1
C	0	0	0

Graph

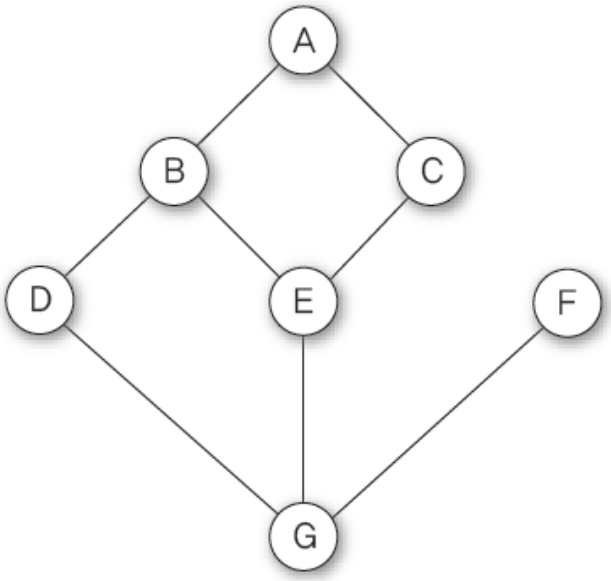
- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 : 인접 리스트
    - n개의 정점과 e개의 간선을 가진 무방향 그래프의 인접 리스트
      - 헤드 노드 배열의 크기 : n
      - 연결하는 노드의 수 : 2e
      - 각 정점의 헤드에 연결된 노드의 수 : 정점의 차수
    - n개의 정점과 e개의 간선을 가진 방향 그래프의 인접 리스트
      - 헤드 노드 배열의 크기 : n
      - 연결하는 노드의 수 : e
      - 각 정점의 헤드에 연결된 노드의 수 : 정점의 진출 차수

Graph

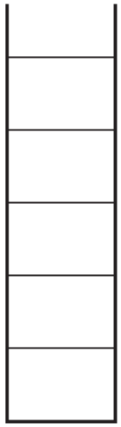
- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 시작 정점의 한 방향으로 갈 수 있는 경로가 있는 곳까지 깊이 탐색해 가다가 더 이상 갈 곳이 없으면, 가장 마지막에 만났던 갈림길 간선이 있는 정점으로 되돌아와 다른 방향의 간선으로 탐색을 계속 반복하여 결국 모든 정점을 방문하는 순회방법
    - 가장 마지막에 만났던 갈림길 간선의 정점으로 가장 먼저 되돌아가서 다시 깊이 우선 탐색을 반복해야 하므로 후입선출 구조의 스택 사용

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 그래프 G9의 깊이 우선 탐색을 위한 초기 상태 : 배열v isited를 false로 초기화하고 공백 스택 생성



G9



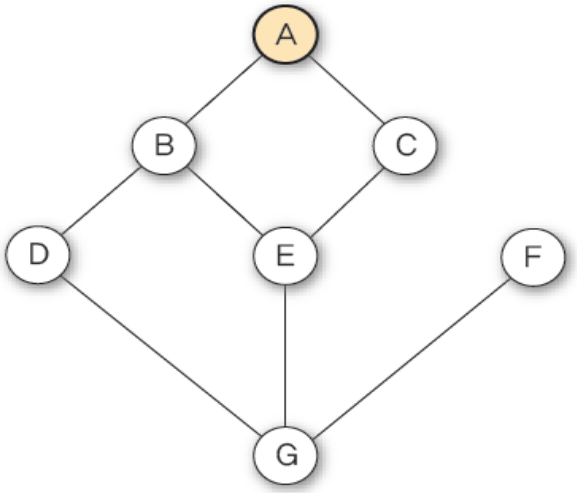
stack

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	F	F	F	F	F	F	F

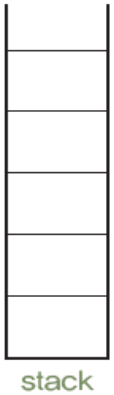
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 A를 시작으로 깊이 우선 탐색을 시작

```
visited[A] ← true;
A 방문;
```



G9



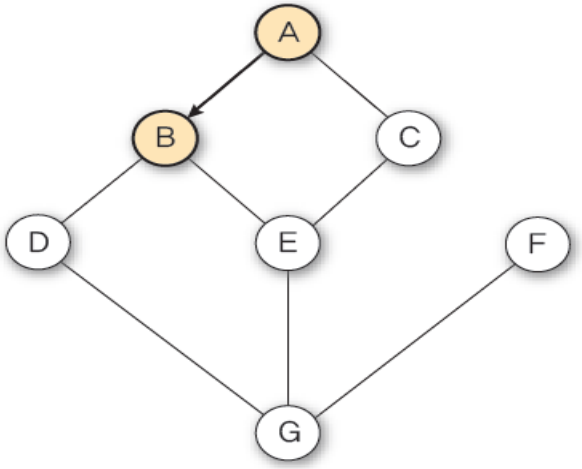
stack

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	F	F	F	F	F	F

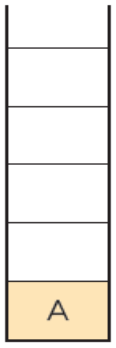
Graph

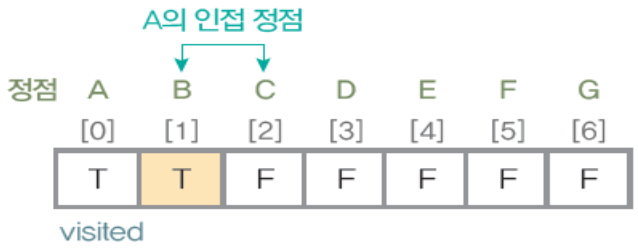
- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 A에 방문하지 않은 정점 B, C가 있으므로 A를 스택에 push하고, 인접 정점 B와 C 중에서 오름차순에 따라 B를 선택하여 탐색을 계속

```
push(stack, A);
visited[B] ← true;
B 방문;'
```



G9

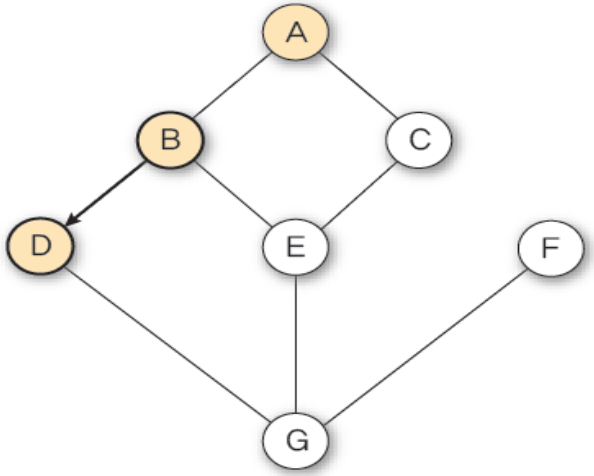


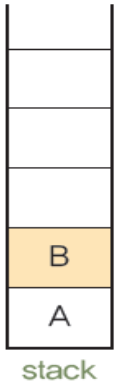


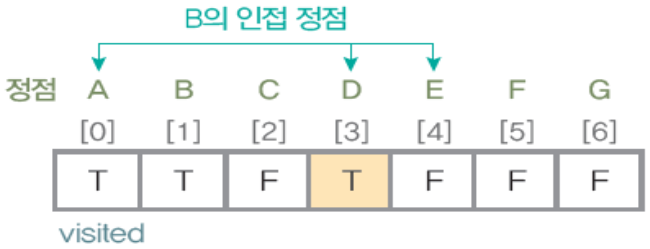
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 B에 방문하지 않은 정점 D, E가 있으므로 B를 스택에 push하고, 방문하지 않은 인접 정점 D와 E 중에서 오름차순에 따라 D를 선택하여 탐색을 계속

```
push(stack, B);
visited[D] ← true;
D 방문;
```



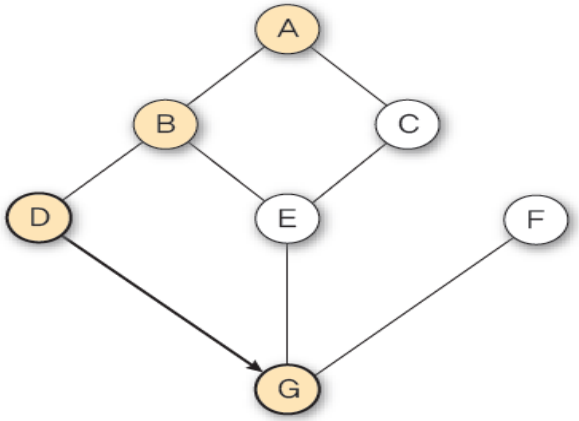


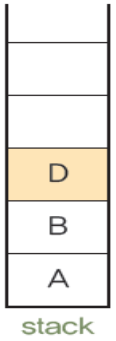


Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 D에 방문하지 않은 정점 G가 있으므로 D를 스택에 push하고, 인접 정점 G를 선택하여 탐색을 계속

```
push(stack, D);
visited[G] ← true;
G 방문;
```





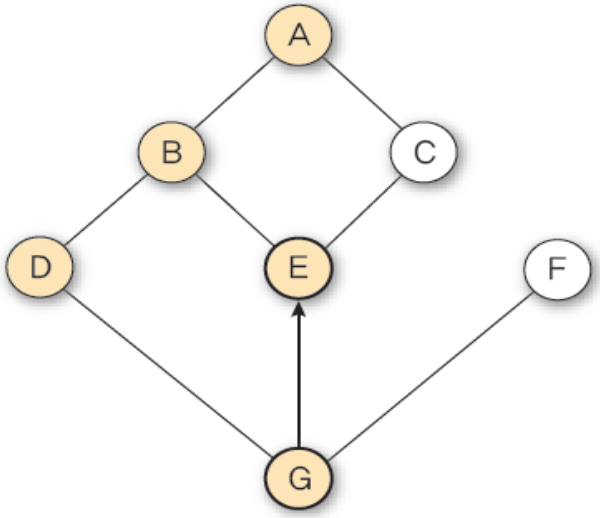
Graph

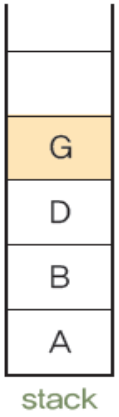
❖ Graph 탐색

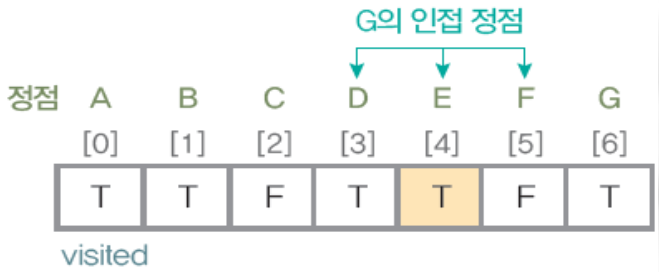
✓ 깊이 우선 탐색(DFS – Depth First Search)

- 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
  - 정점 G에 방문하지 않은 정점 E, F가 있으므로 G를 스택에 push하고, 방문하지 않은 인접 정점 E와 F중에서 오름차순에 따라 E를 선택하여 탐색을 계속

```
push(stack, G);
visited[E] ← true;
E 방문;
```



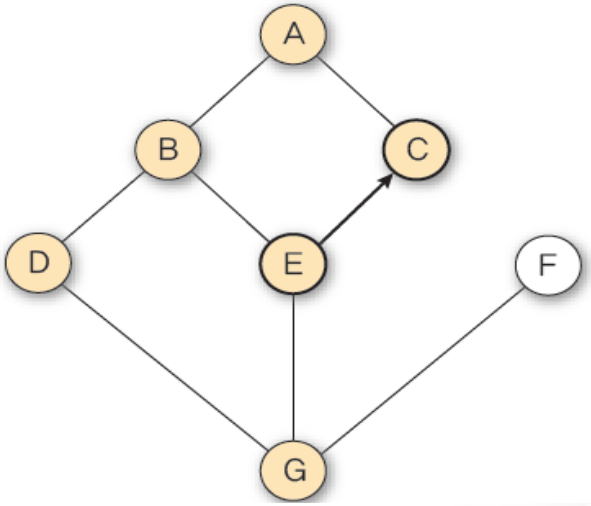


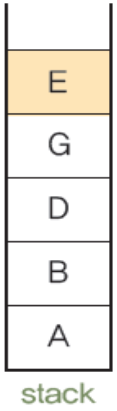


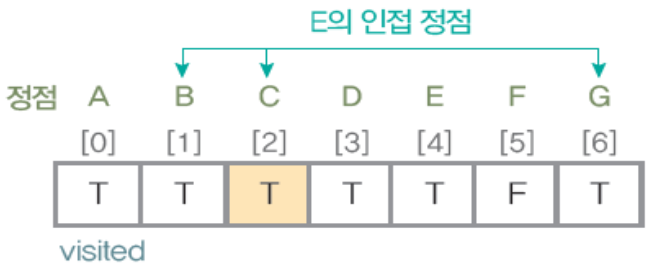
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 E에 방문하지 않은 정점 C가 있으므로 E를 스택에 push하고, 방문하지 않은 인접 정점 C를 선택하여 탐색을 계속

```
push(stack, E);
visited[C] ← true;
C 방문;
```



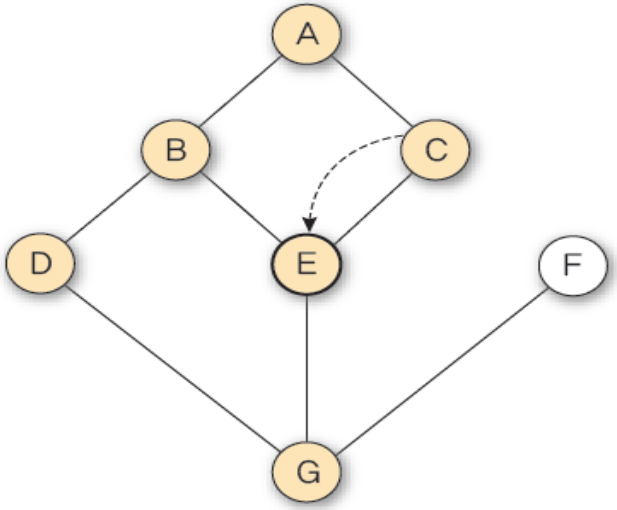


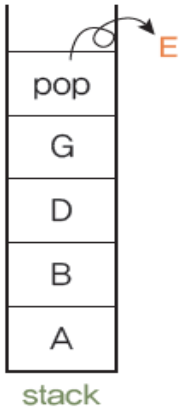


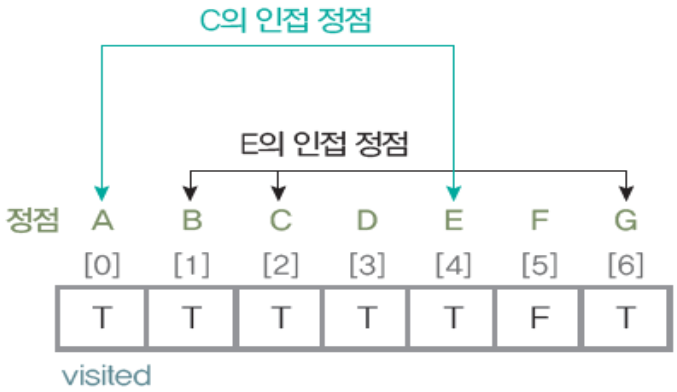
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 C에서 방문하지 않은 인접 정점이 없으므로 마지막 정점으로 돌아가기 위해 스택을 pop하여 받은 정점 E에 대해서 방문하지 않은 인접 정점이 있는지 확인
      - 정점 E는 방문하지 않은 인접 정점이 없음

```
pop(stack);
```



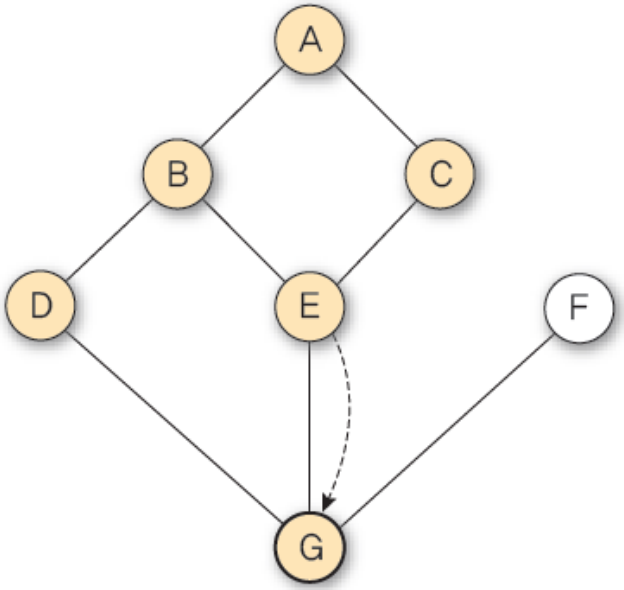


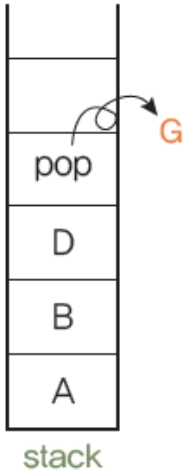


Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 현재 정점 E에서 방문할 수 있는 인접 정점이 없으므로, 다시 스택을 pop하여 받은 정점 G에 대해서 방문하지 않은 인접 정점이 있는지 확인

```
pop(stack);
```



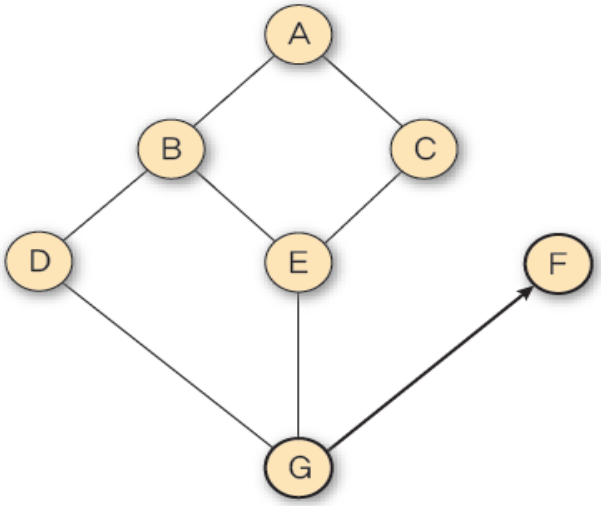


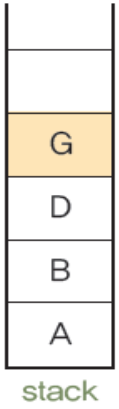
정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	F	T

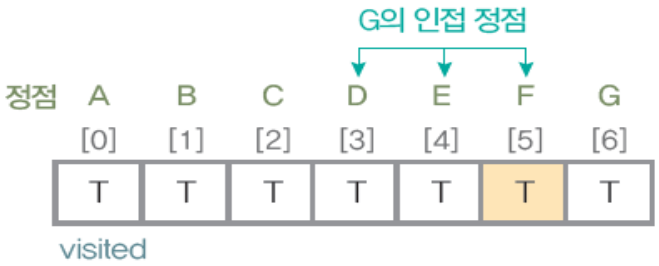
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 현재 정점 G에서 방문하지 않은 정점 F가 있으므로 G를 스택에 push하고, 인접 정점 F를 선택하여 탐색을 계속

```
push(stack, G);
visited[F] ← true;
F 방문;
```







Graph

❖ Graph 탐색

✓ 너비 우선 탐색(BFS – Breadth First Search)

❶ 시작 정점 v를 결정하여 방문한다.

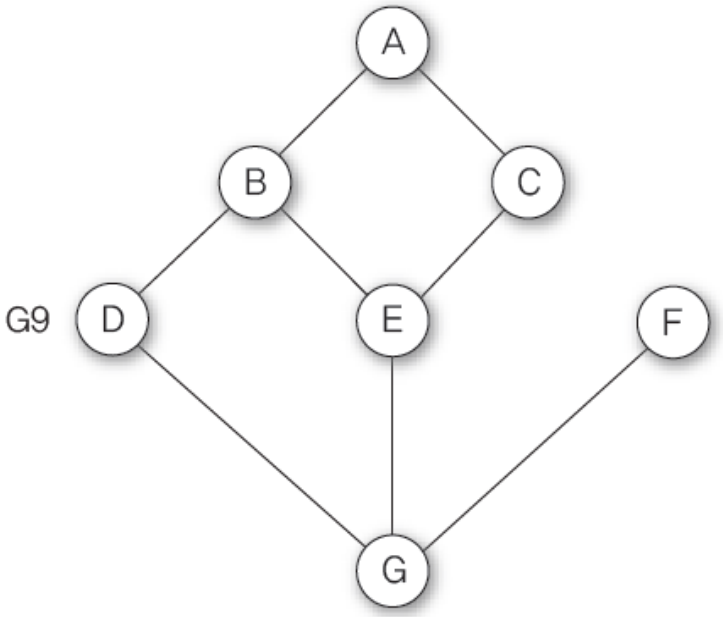
❷ 정점 v에 인접한 정점 중에서 방문하지 않은 정점을 차례로 방문하면서 큐에 enQueue한다.

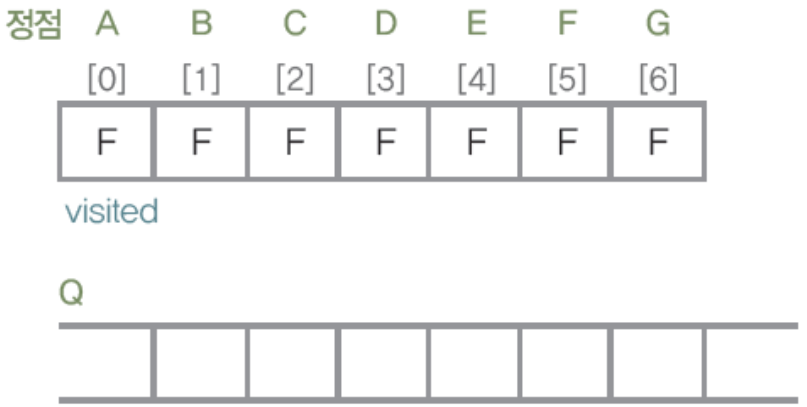
❸ 방문하지 않은 인접한 정점이 없으면, 방문했던 정점에서 인접한 정점을 다시 차례로 방문하기 위해 큐에서 deQueue 하여 받은 정점을 v로 설정하고 ❷를 반복한다.

❹ 큐가 공백이 될 때까지 ❷~❸을 반복한다.

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 초기상태 : 배열 visited를 False로 초기화, 공백 큐를 생성





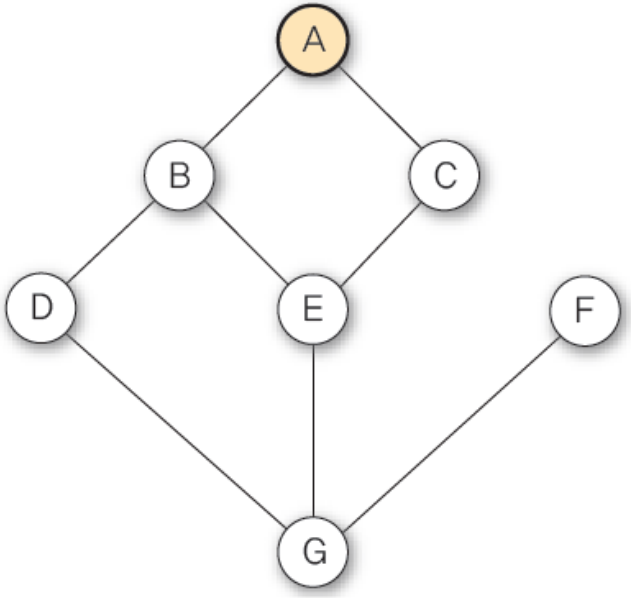
Graph

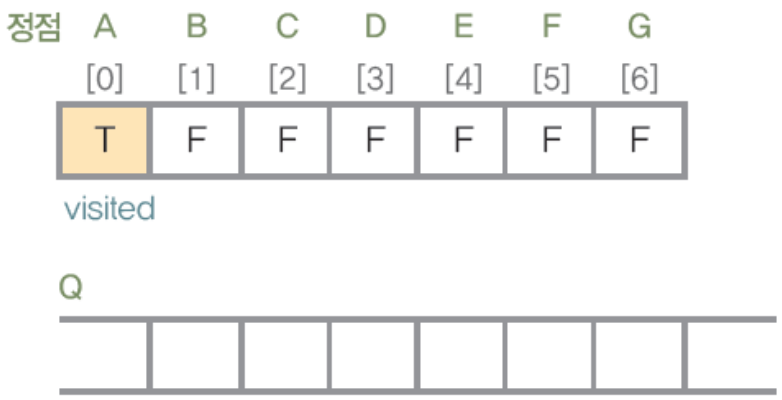
❖ Graph 탐색

✓ 너비 우선 탐색(BFS – Breadth First Search)

- 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
  - 정점 A를 시작으로 너비 우선 탐색을 시작

```
visited[A] ← true;
A 방문;
```

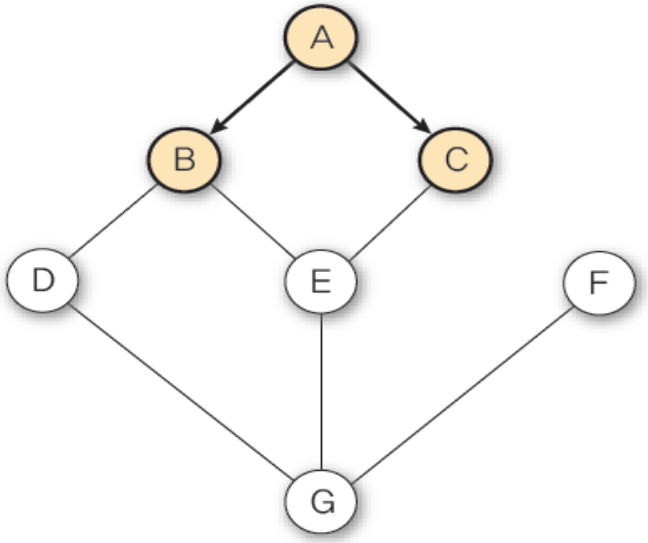


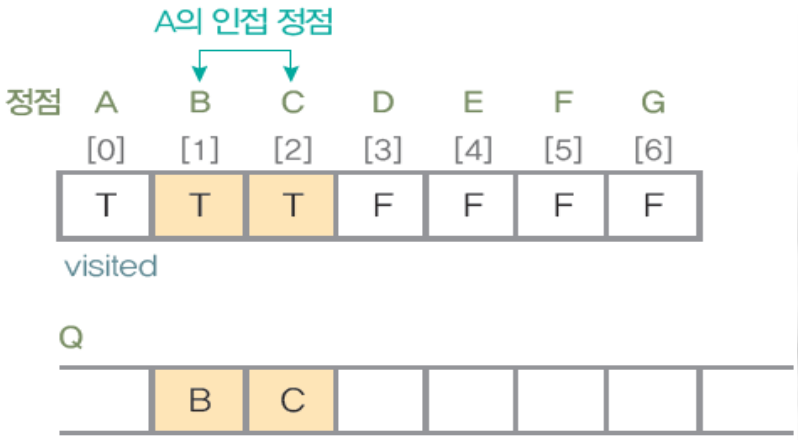


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 A에서 방문하지 않은 모든 인접 정점 B, C를 방문하고 큐에 enQueue

```
visited[(A가 방문하지 않은 인접 정점 B와 C)] ← true;
(A가 방문하지 않은 인접 정점 B와 C) 방문;
enQueue(Q, (A가 방문하지 않은 인접 정점 B와 C));
```

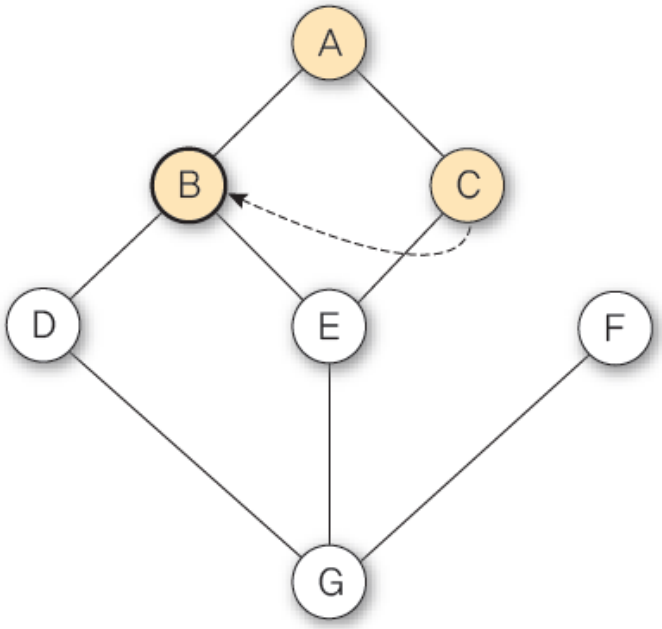


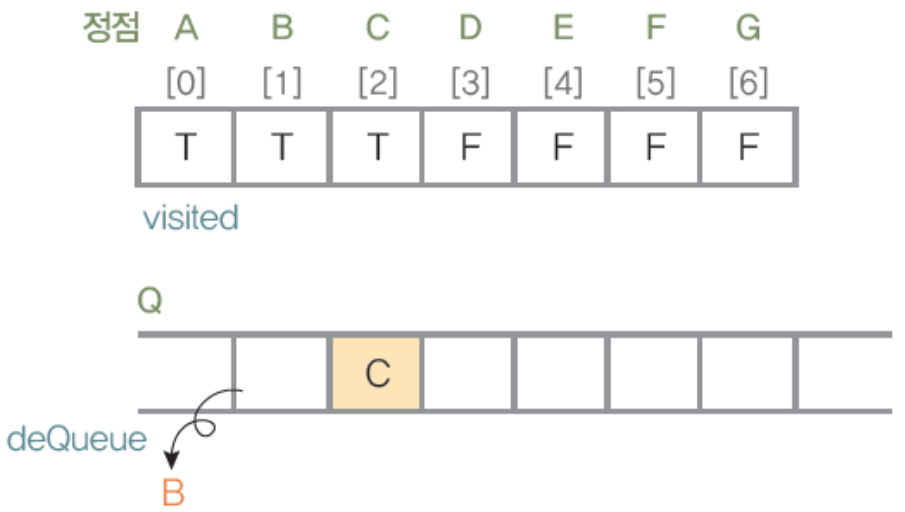


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 A에 대한 인접 정점들을 처리했으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 B를 받음

```
v ← deQueue(Q);
```

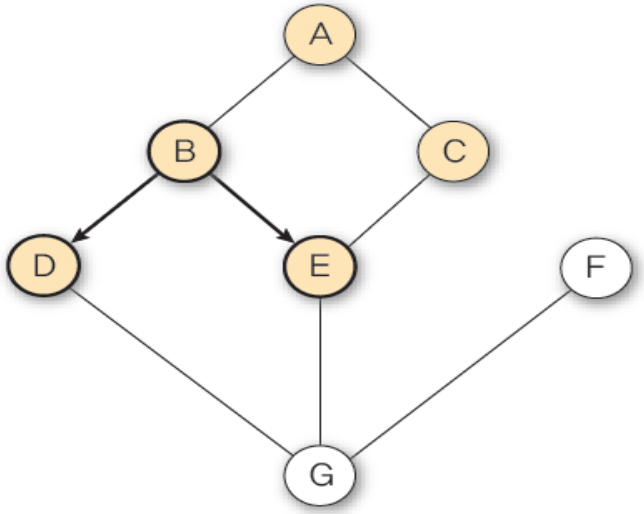


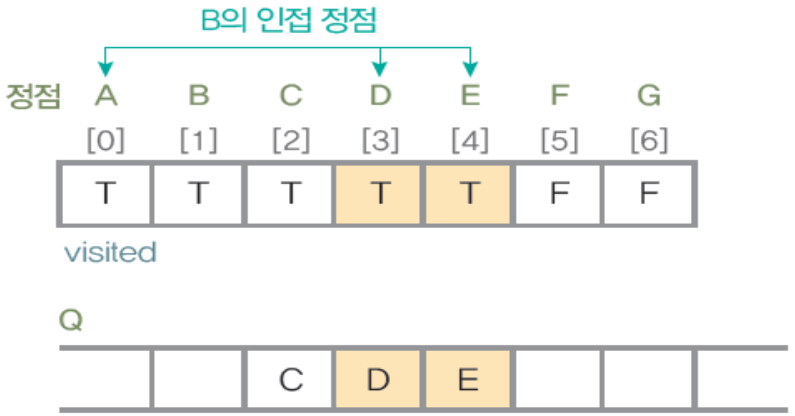


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 B에서 방문하지 않은 모든 인접 정점 D, E를 방문하고 큐에 enQueue

```
visited[(B가 방문하지 않은 인접 정점 D와 E)] ← true;
(B가 방문하지 않은 인접 정점 D와 E) 방문;
enQueue(Q, (B가 방문하지 않은 인접 정점 D와 E));
```

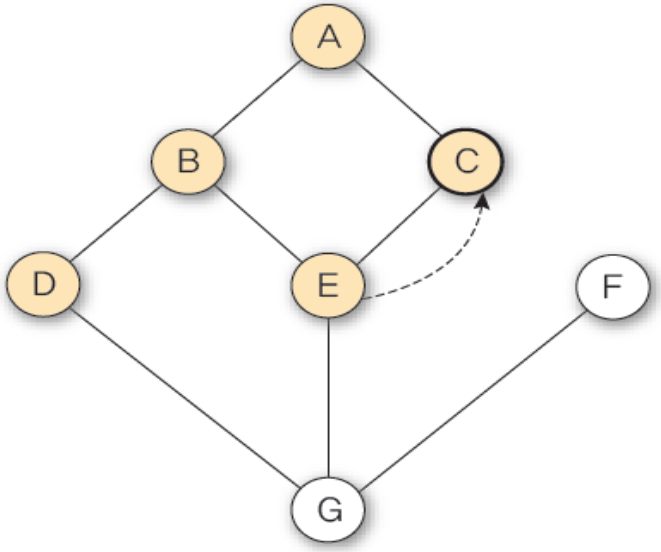


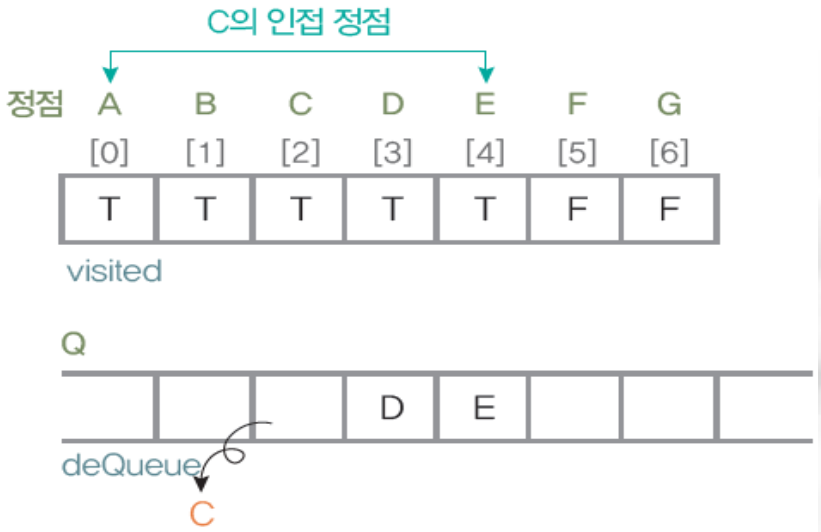


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 B에 대한 인접 정점들을 처리했으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 C를 받음

```
v ← deQueue(Q);
```

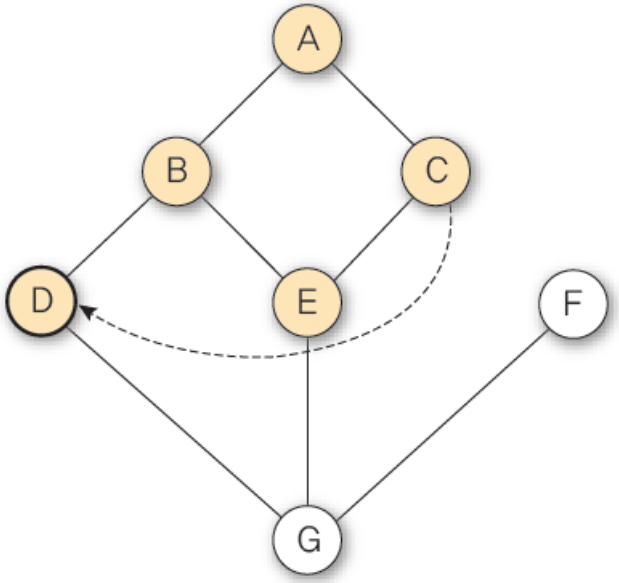


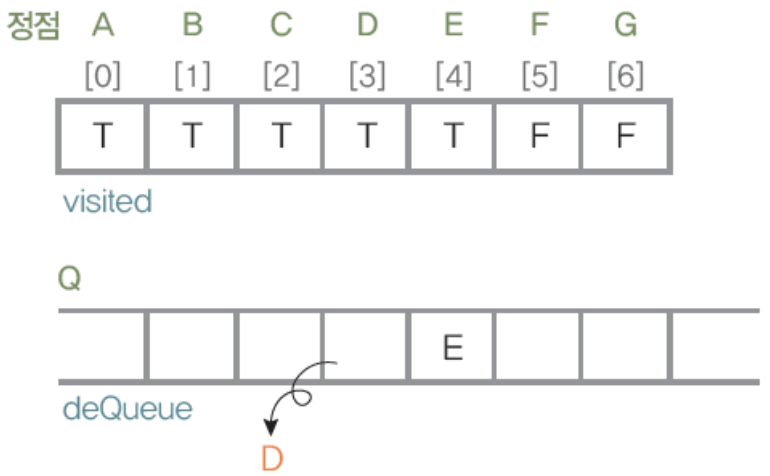


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 C에는 방문하지 않은 인접 정점이 없으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 D를 받음

```
v ← deQueue(Q);
```





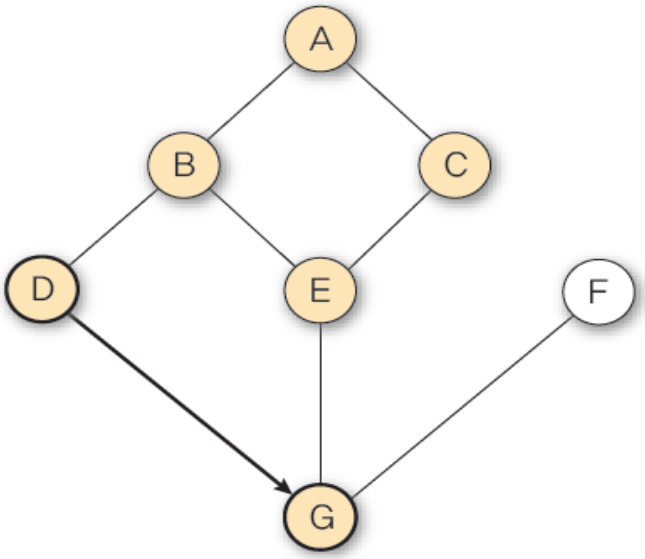
Graph

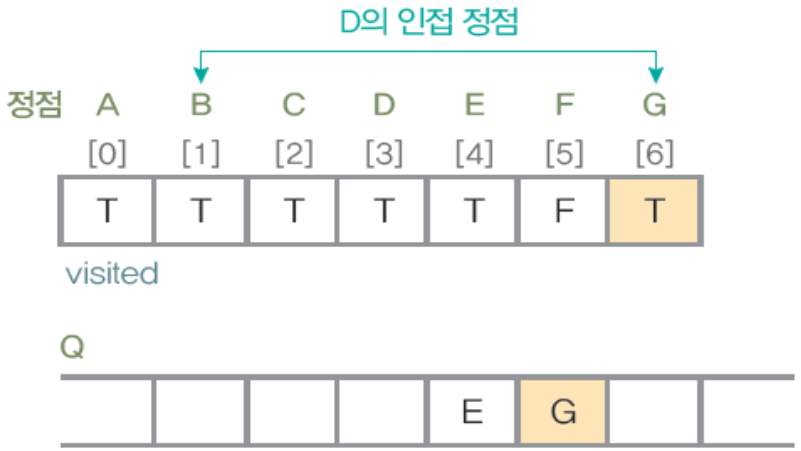
❖ Graph 탐색

✓ 너비 우선 탐색(BFS – Breadth First Search)

- 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
  - 정점 D에서 방문하지 않은 인접 정점 G를 방문하고 큐에 enQueue

```
visited[(D가 방문하지 않은 인접 정점 G)] ← true;
(D가 방문하지 않은 인접 정점 G) 방문;
enQueue(Q, (D가 방문하지 않은 인접 정점 G));
```

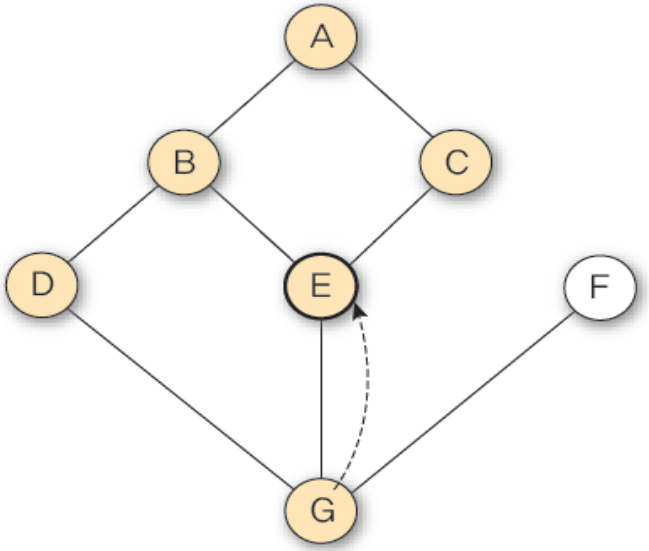


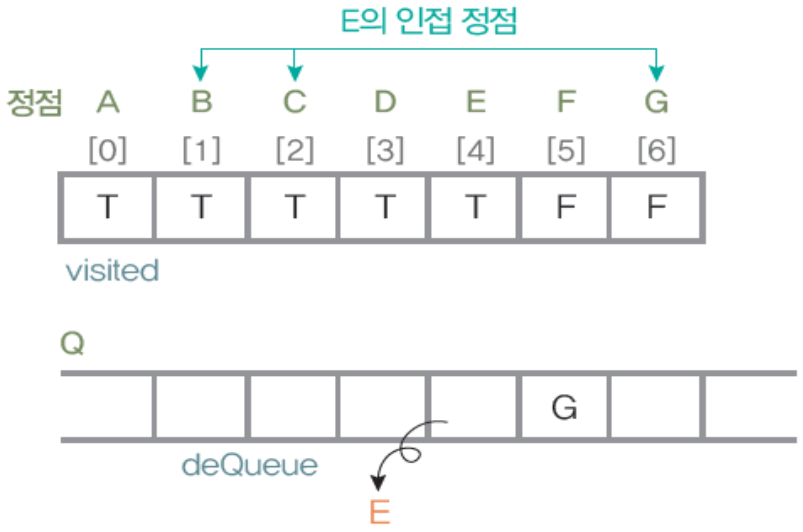


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 D에 대한 인접 정점들을 처리했으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 E를 받음

```
v ← deQueue(Q);
```

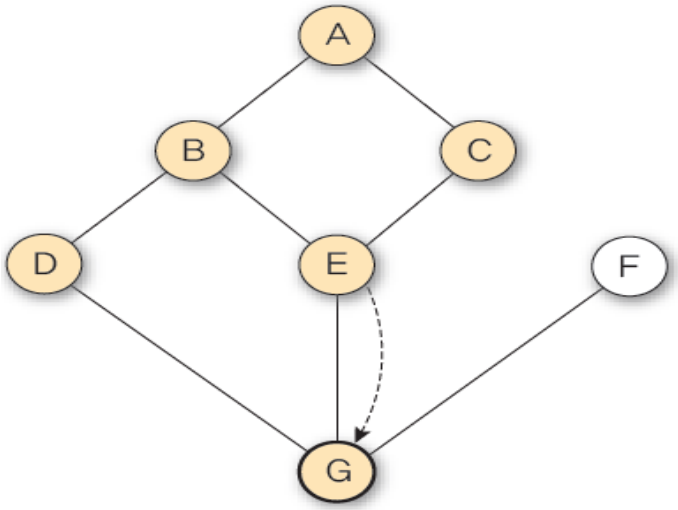


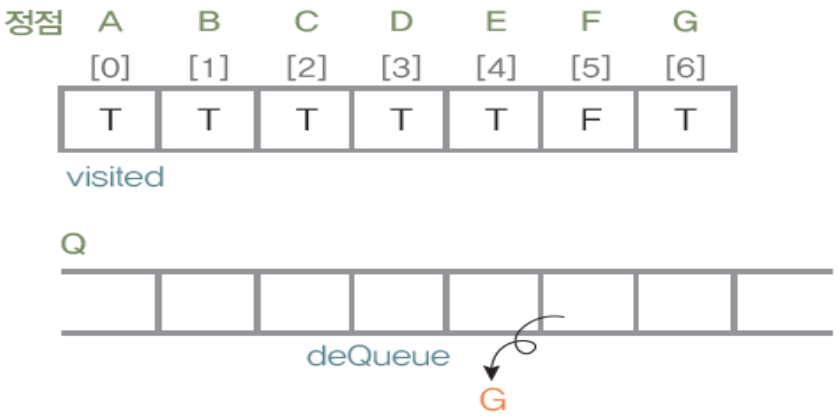


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 E에는 방문하지 않은 인접 정점이 없으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 G를 받음

```
v ← deQueue(Q);
```

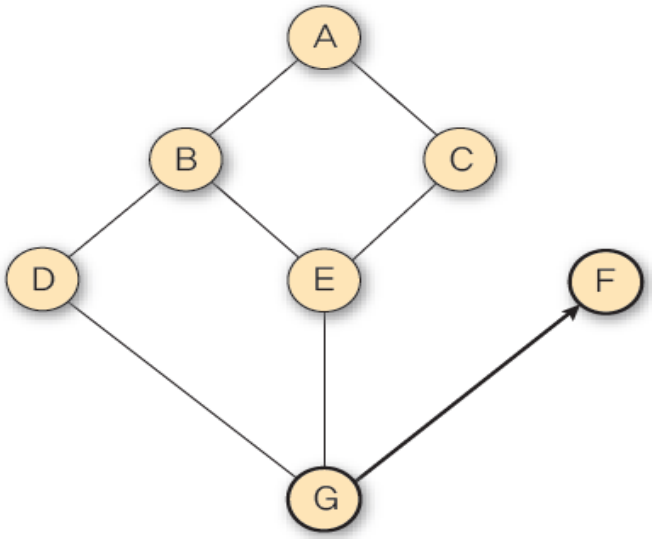




Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 G에서 방문하지 않은 인접 정점 F를 방문하고 큐에 enQueue

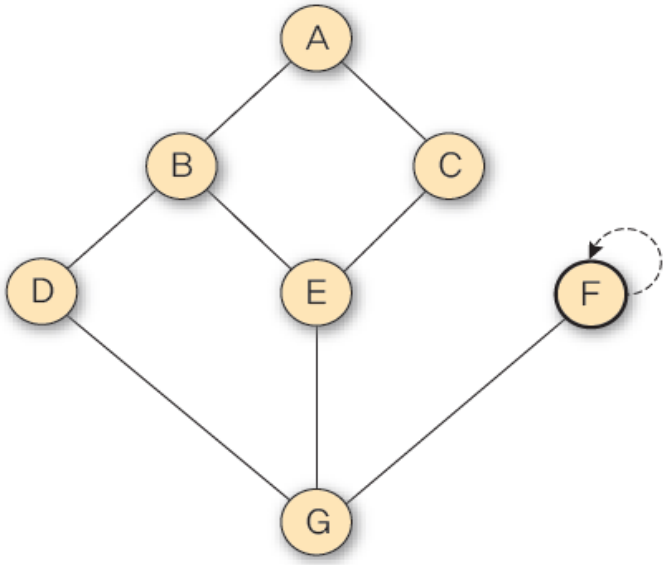
```
visited[(G가 방문하지 않은 인접 정점 F)] ← true;
(G가 방문하지 않은 인접 정점 F) 방문;
enQueue(Q, (G가 방문하지 않은 인접 정점 F));
```

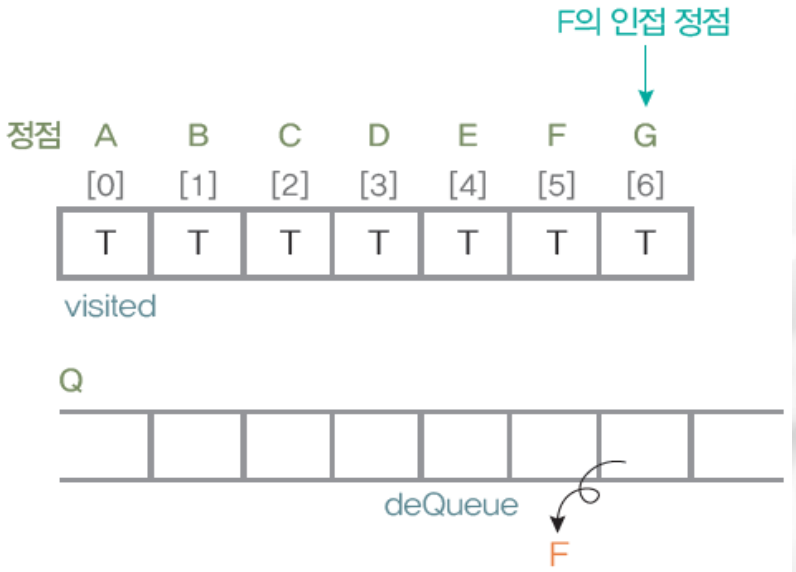


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 G에 대한 인접 정점들을 처리했으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 F를 받음

```
v ← deQueue(Q);
```





Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 F는 모든 인접 정점을 방문했으므로, 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue. 큐가 공백이므로 너비 우선 탐색을 종료

정렬

❖ 정렬(Sorting)
- ✓ 정렬은 데이터를 순서대로 나열하는 것
- ✓ 정렬 방식은 작은 것부터 큰 순서대로 나열하는 오름차순(Ascending) 정렬과 큰 것에서 작은 순서대로 나열하는 내림차순(Descending) 정렬이 있고 정렬을 하는 방법은 구현하는 알고리즘에 따라 여러 가지

기준	정렬 방식	설명
실행 방법	비교식 정렬 Comparative Sort	비교할 각 키값을 한 번에 두 개씩 비교하여 교환함으로써 정렬을 실행하는 방식
	분배식 정렬 Distribute Sort	키값을 기준으로 하여 자료를 여러 개의 부분집합으로 분해하고, 각 부분집합을 정렬함으로써 전체를 정렬하는 방식
정렬 장소	내부 정렬 Internal Sort	컴퓨터 메모리 내부에서 정렬
	외부 정렬 External Sort	메모리의 외부인 보조 기억 장치에서 정렬

정렬

❖ 정렬(Sorting)

✓ 내부 정렬(internal sort)

- 정렬할 자료를 메인 메모리에 올려서 정렬하는 방식
- 정렬 속도가 빠르지만 정렬할 수 있는 자료의 양이 메인 메모리의 용량에 따라 제한됨

구분	종류	설명
비교식	교환 방식	키를 비교하고 교환하여 정렬하는 방식(선택 정렬, 버블 정렬, 퀵 정렬)
	삽입 방식	키를 비교하고 삽입하여 정렬하는 방식(삽입 정렬, 셸 정렬)
	병합 방식	키를 비교하고 병합하여 정렬하는 방식(2-way 병합, n-way 병합)
	선택 방식	이진 트리를 사용하여 정렬하는 방식(히프 정렬, 트리 정렬)
분배식	분배 방식	키를 구성하는 값을 여러 개의 부분집합에 분배하여 정렬하는 방식(기수 정렬)

✓ 외부 정렬(external sort)

- 정렬할 자료를 보조 기억장치에서 정렬하는 방식
- 대용량의 보조 기억 장치를 사용하기 때문에 내부 정렬보다 속도는 떨어지지만 내부 정렬로 처리할 수 없는 대용량의 자료에 대한 정렬 가능
- 병합 방식 : 파일을 부분 파일로 분리하여 각각을 내부 정렬 방법으로 정렬하여 병합하는 정렬 방식 (2-way 병합, n-way 병합)

정렬

❖ Selection(선택) Sort

✓ 선택 정렬은 첫 번째 자리부터 마지막에서 두 번째 자리까지의 데이터를 기준으로 기준 뒤에 있는 모든 데이터들과 비교해서 기준 자리의 데이터가 크면 2개 요소의 자리를 변경

초기 상태 50 40 10 20 30

1Pass10 50 40 20 30 : 첫번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 10보다 작아서 교환

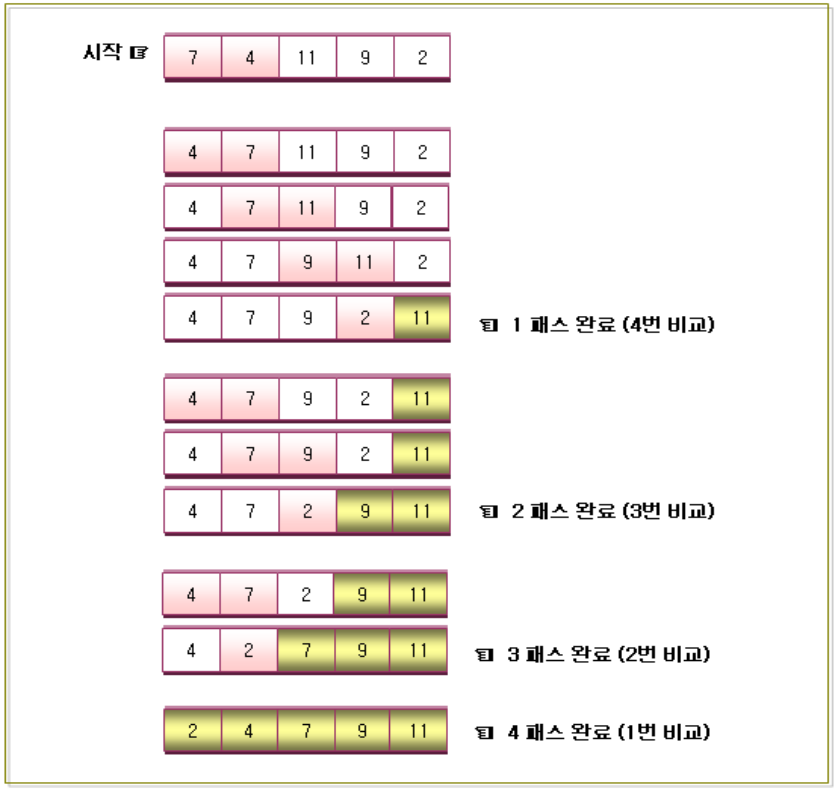
2Pass10 20 50 40 30 : 두번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 20보다 작아서 교환

3Pass10 20 30 50 40 : 세번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 30보다 작아서 교환

4Pass10 20 30 40 50 : 네번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환

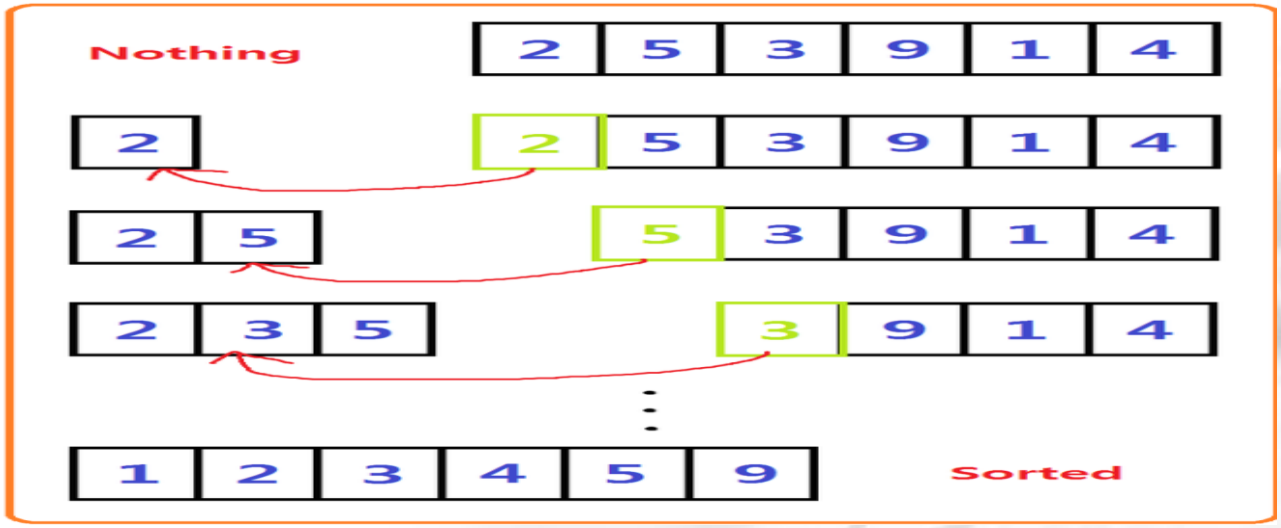
정렬

- Bubble Sort
  - ✓ 버블 정렬은 n개의 데이터가 있을 때 1부터 n-1 번째 자료까지 n-1번 동안 다음 자료와 비교해가면서 정렬하는 방법
  - ✓ 버블 정렬의 효과를 높이기 위해서는 비교 시 횟수만큼 빼면서 정렬하면 성능을 높일 수 있고 flag처리 등을 이용해서 자리 바꿈이 일어나지 않을 때 멈추게 하면 성능을 더욱 향상 시킬 수 있음



정렬

- Insertion(삽입) Sort
  - ✓ 정렬되어 있는 부분집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
    - 정렬할 자료를 두 개의 부분집합 SSorted Subset와 UUnsorted Subset로 가정
    - 부분집합 S : 정렬된 앞부분의 원소들
    - 부분집합 U : 아직 정렬되지 않은 나머지 원소들
    - 정렬되지 않은 부분집합 U의 원소를 하나씩 꺼내서 이미 정렬되어 있는 부분집합 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입
    - 삽입 정렬을 반복하면서 부분집합 S의 원소는 하나씩 늘리고 부분집합 U의 원소는 하나씩 감소하게 함. 부분집합 U가 공집합이 되면 삽입 정렬이 완성



정렬

❖ Quick Sort

❶ 왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피봇보다 크거나 같은 원소를 찾아 L로 표시한다. 단, L은 R과 만나면 더 이상 오른쪽으로 이동하지 못하고 멈춘다.

❷ 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾아 R로 표시한다. 단, R은 L과 만나면 더 이상 왼쪽으로 이동하지 못하고 멈춘다.

❸-ⓐ ❶과 ❷에서 찾은 L 원소와 R 원소가 있는 경우, 서로 교환하고 L과 R의 현재 위치에서 ❶과 ❷ 작업을 다시 수행한다.

❸-ⓑ ❶~❷를 수행하면서 L과 R이 같은 원소에서 만나 멈춘 경우, 피봇과 R의 원소를 서로 교환한다. 교환된 자리를 피봇 위치로 확정하고 현재 단계의 퀵 정렬을 끝낸다.

❹ 피봇의 확정된 위치를 기준으로 만들어진 새로운 왼쪽 부분집합과 오른쪽 부분집합에 대해서 ❶~❸의 퀵 정렬을 순환적으로 반복 수행하는데, 모든 부분집합의 크기가 1 이하가 되면 전체 퀵 정렬을 종료한다.

정렬

❖ Quick Sort

✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)

○ 1단계

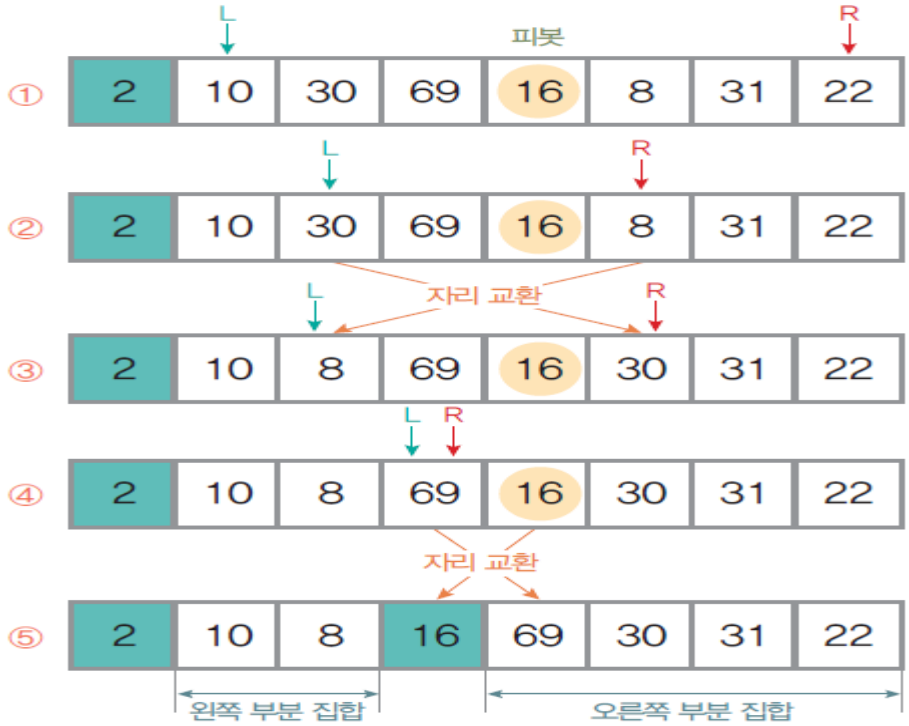
- 원소 2를 피봇으로 선택하고 퀵 정렬을 시작
- L은 정렬 범위의 왼쪽 끝에서 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 찾고, R은 정렬 범위의 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음. L은 원소 69를 찾았지만, R은 피봇보다 작은 원소를 찾지 못한 상태로 원소 69에서 L과 만남. L과 R이 만나 더 이상 진행할 수 없는 상태가 되었으므로
- 원소 69를 피봇과 자리를 교환하고 피봇 원소 2의 위치를 확정

정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 2단계: 위치가 확정된 피봇 2의 왼쪽 부분집합은 공집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합에 대해서 퀵 정렬을 수행
      - 오른쪽 부분집합의 원소가 일곱 개이므로 가운데 있는 원소 16을 피봇으로 선택하고 퀵 정렬을 시작.
      - L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소인 30을 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소인 8을 찾음.
      - L이 찾은 30과 R이 찾은 8을 서로 자리를 교환한 후 현재 위치에서 L은 다시 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소69를 찾고 R은 피봇보다 작은 원소를 찾음
      - R이 원소 69에서 L과 만나 더 이상 진행할 수 없는 상태가 되었으므로,
      - 원소 69를 피봇과 교환하고 피봇 원소 16의 위치를 확정

정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 2단계: 위치가 확정된 피봇 2의 왼쪽 부분집합은 공집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합에 대해서 퀵 정렬을 수행

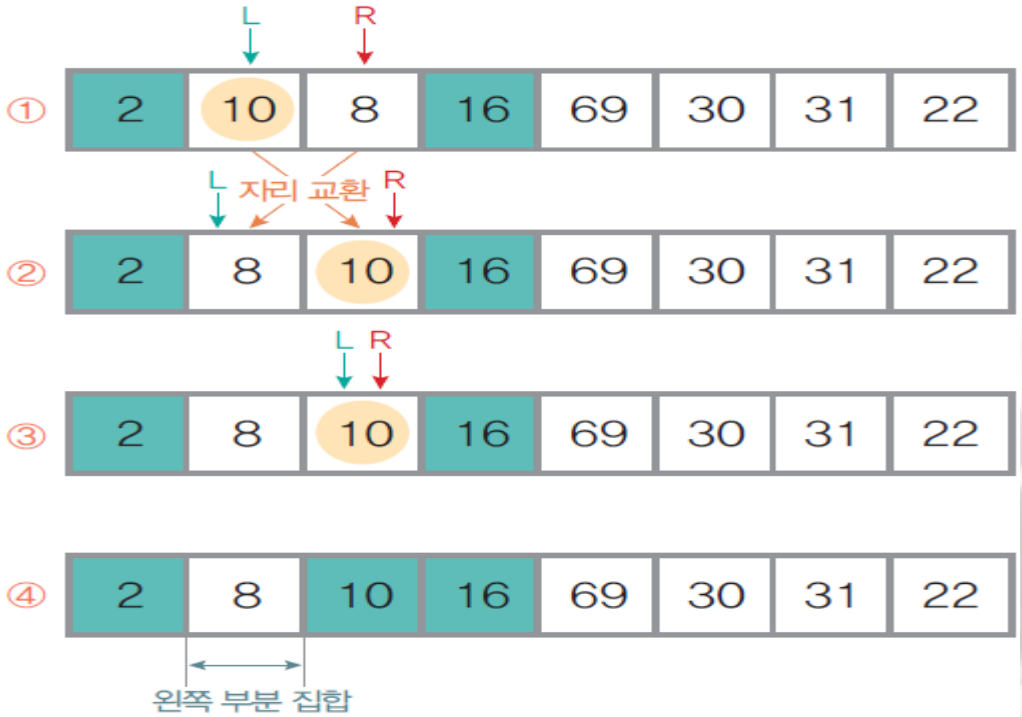


정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 3단계: 위치가 확정된 피봇 16을 기준으로 새로 생긴 왼쪽 부분집합에서 퀵 정렬을 수행
      - 원소 10을 피봇으로 선택하고 L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를, R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L이 찾은 10과 R이 찾은 8을 서로 교환
      - 현재 위치에서 L은 다시 오른쪽으로 움직이다가 원소 10에서 R과 만나서 멈추게 됨. 더 이상 진행할 수 없는 상태가 되었으므로,
      - R의 원소와 피봇을 교환하고 피봇 원소 10의 위치를 확정한다(이 경우에는 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)

정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 3단계: 위치가 확정된 피봇 16을 기준으로 새로 생긴 왼쪽 부분집합에서 퀵 정렬을 수행

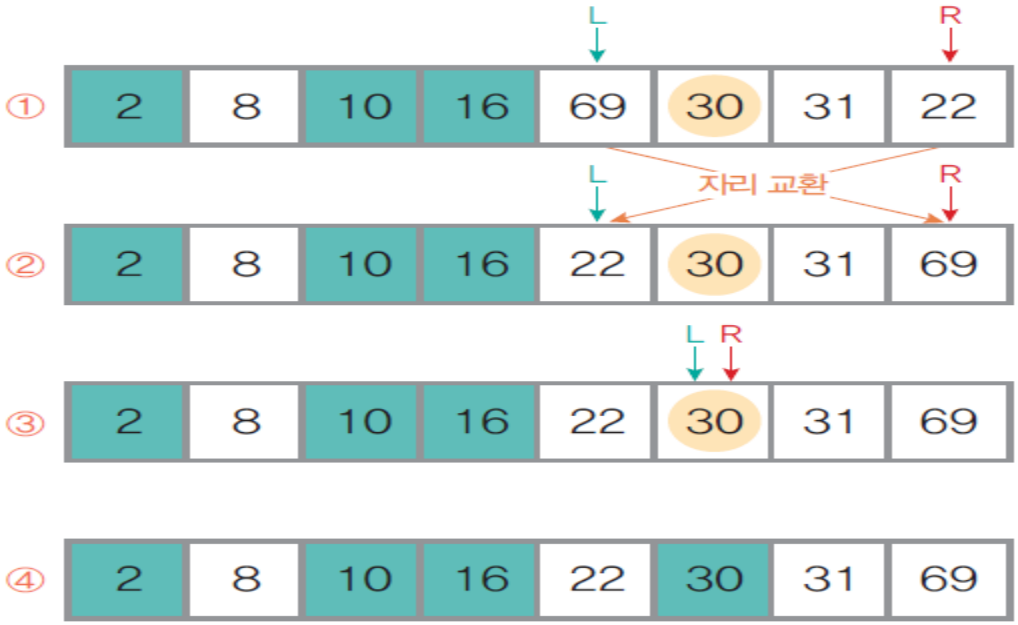


정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 4단계: 피봇 10의 왼쪽 부분집합은 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합은 공집합이므로 역시 퀵 정렬을 수행하지 않고 [2]에서 피봇이었던 원소 16의 오른쪽 부분집합에 대해서 퀵 정렬을 수행
      - 오른쪽 부분집합의 원소가 네 개이므로 가운데 있는 원소 30을 피봇으로 선택. L은 오른쪽으로 이동하면서 피봇보다 크거나 같은 원소를 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L이 찾은 69와 R이 찾은 22를 서로 교환. 현재 위치에서 다시 L은 피봇보다 크거나 같은 원소를 찾아 오른쪽으로 움직이고 R은 피봇보다 작은 원소를 찾아 왼쪽으로 움직이다가
      - 원소 30에서 L과 R이 만나 멈춤. 더 이상 진행할 수 없음
      - R의 원소와 피봇을 교환하고 피봇 원소30의 위치가 확정 (이 경우에 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)

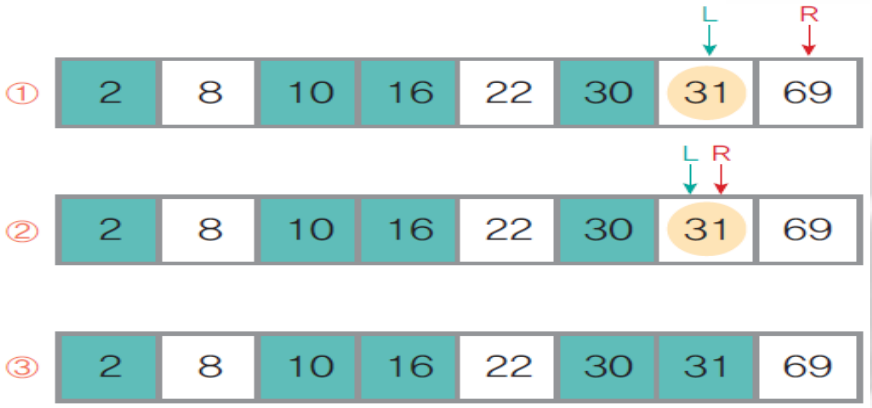
정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 4단계: 피봇 10의 왼쪽 부분집합은 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합은 공집합이므로 역시 퀵 정렬을 수행하지 않고 [2]에서 피봇이었던 원소 16의 오른쪽 부분집합에 대해서 퀵 정렬을 수행



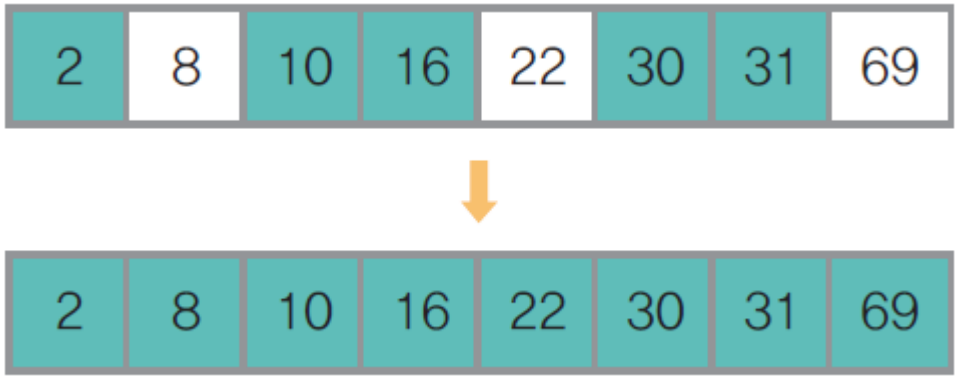
정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 5단계: 피봇 30의 왼쪽 부분집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합에 대해서 퀵 정렬을 수행
      - 피봇은 31을 선택하고 L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L과 R이 원소 31에서 만나 더 이상 진행하지 못하는 상태가 되어
      - 원소 31을 피봇과 교환하여 위치를 확정(이 경우에는 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)



정렬

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)
    - 6단계: 피봇 31의 오른쪽 부분집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않지 않고 모든 부분 집합의 크기가 1 이하이므로 전체 퀵 정렬을 종료



Shell Sort

❖Shell Sort

- ✓ 일정한 간격(interval)으로 떨어져있는 자료들끼리 부분집합을 구성하고 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행하는 작업을 반복하면서 전체 원소들을 정렬하는 방법
- ✓ 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하게 되면 비교연산과 교환연산 감소

- ✓ 셸 정렬의 부분집합
  - ○ 부분집합의 기준이 되는 간격을 매개변수 h에 저장
  - ○ 한 단계가 수행될 때마다 h의 값을 감소시키고 셸 정렬을 순환 호출
  - ○ h가 1이 될 때까지 반복

- ✓ 셸 정렬의 성능은 매개변수 h의 값에 따라 달라짐
  - ○ 정렬할 자료의 특성에 따라 매개변수 생성 함수를 사용
  - ○ 일반적으로 사용하는 h의 값은 원소 개수의 1/2을 사용하고 한 단계 수행될 때마다 h의 값을 반으로 감소시키면서 반복 수행

정렬

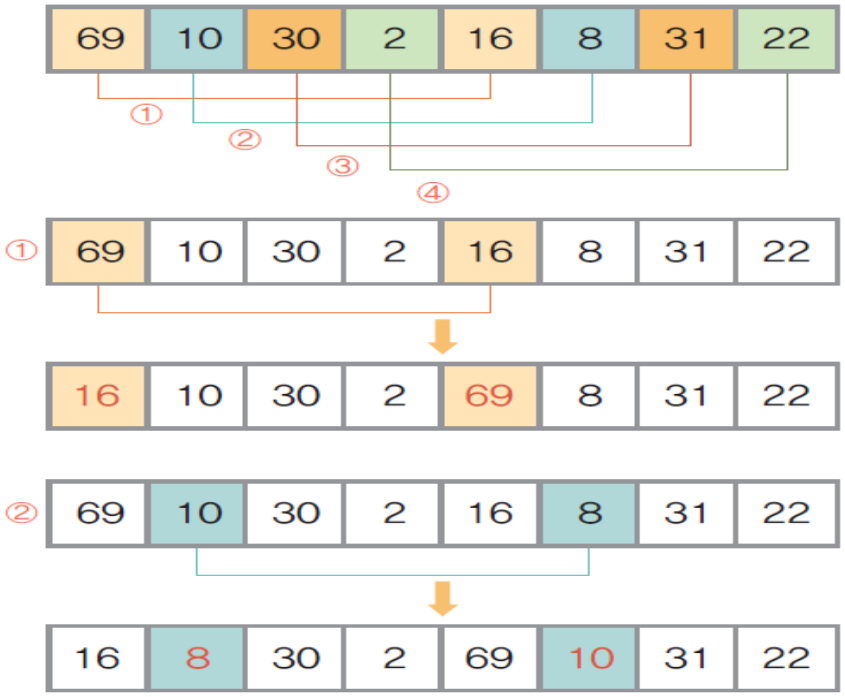
- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 원소 개수가 여덟 개이므로 매개변수 h는 4에서 시작한다. h=4이므로 간격이 4만큼 떨어져 있는 원소들을 같은 부분집합으로 만들면 네 개의 부분집합이 만들어짐(같은 부분집합은 동일한 색으로 표시)
      - 첫 번째 부분집합 {69, 16}에 대해서 삽입 정렬을 수행하여 정렬
      - 두 번째 부분집합{10, 8}에 대해서 삽입 정렬을 수행
      - 세 번째 부분집합 {30, 31}에 대해서 삽입 정렬을 수행. 30<31이므로 자리 이동은 이루어지지 않음.
      - 네 번째 부분집합 {2, 22}에 대해서 삽입 정렬을 수행. 2<22이므로 자리 이동은 이루어지지 않음

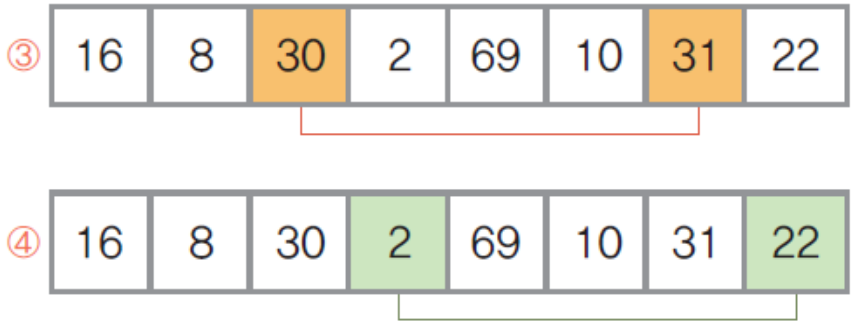
정렬

❖ Shell Sort

✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정

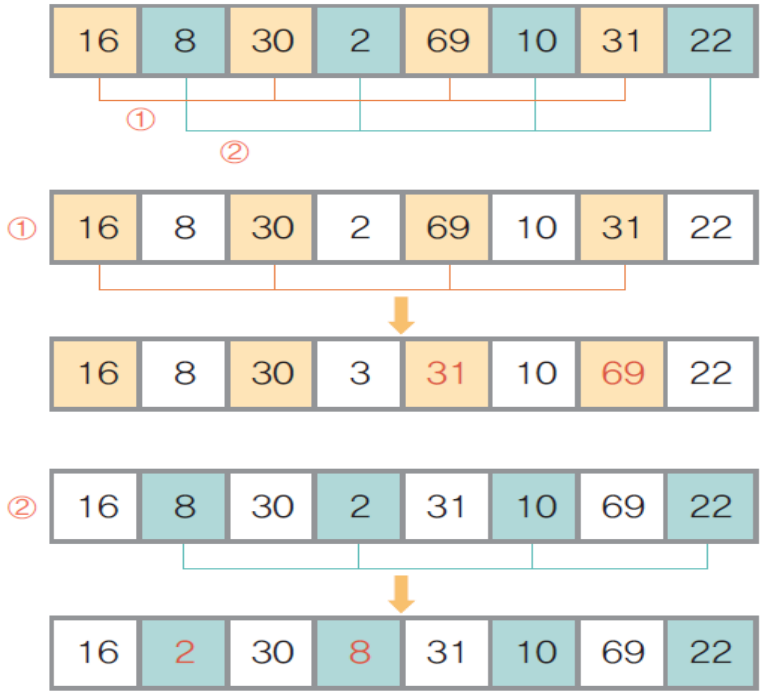
- 원소 개수가 여덟 개이므로 매개변수 h는 4에서 시작한다. h=4이므로 간격이 4만큼 떨어져 있는 원소들을 같은 부분집합으로 만들면 네 개의 부분집합이 만들어짐(같은 부분집합은 동일한 색으로 표시)





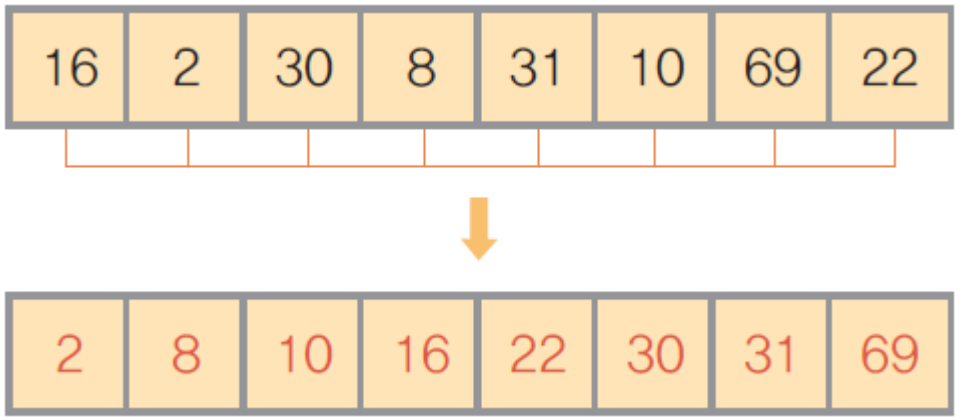
정렬

- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 이제 h를 2로 변경하고 다시 셸 정렬을 시작. h=2이므로 간격이 2만큼 떨어진 원소들을 같은 부분집합으로 만들면 두 개의 부분집합이 만들어짐
      - 첫 번째 부분집합 {16, 30, 69, 31}에 대해 삽입 정렬을 수행하여 정렬
      - 두 번째 부분집합 {8, 2, 10, 22}에 대해 삽입 정렬을 수행하여 정렬



정렬

- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - ○ 이제 h를 1로 변경하고 다시 셸 정렬을 시작. h=1이므로 간격이 1만큼 떨어져 있는 원소들을 같은 부분집합으로 만들면 한 개의 부분집합이 만들어짐 - 전체 원소에 대해서 삽입 정렬을 수행



정렬

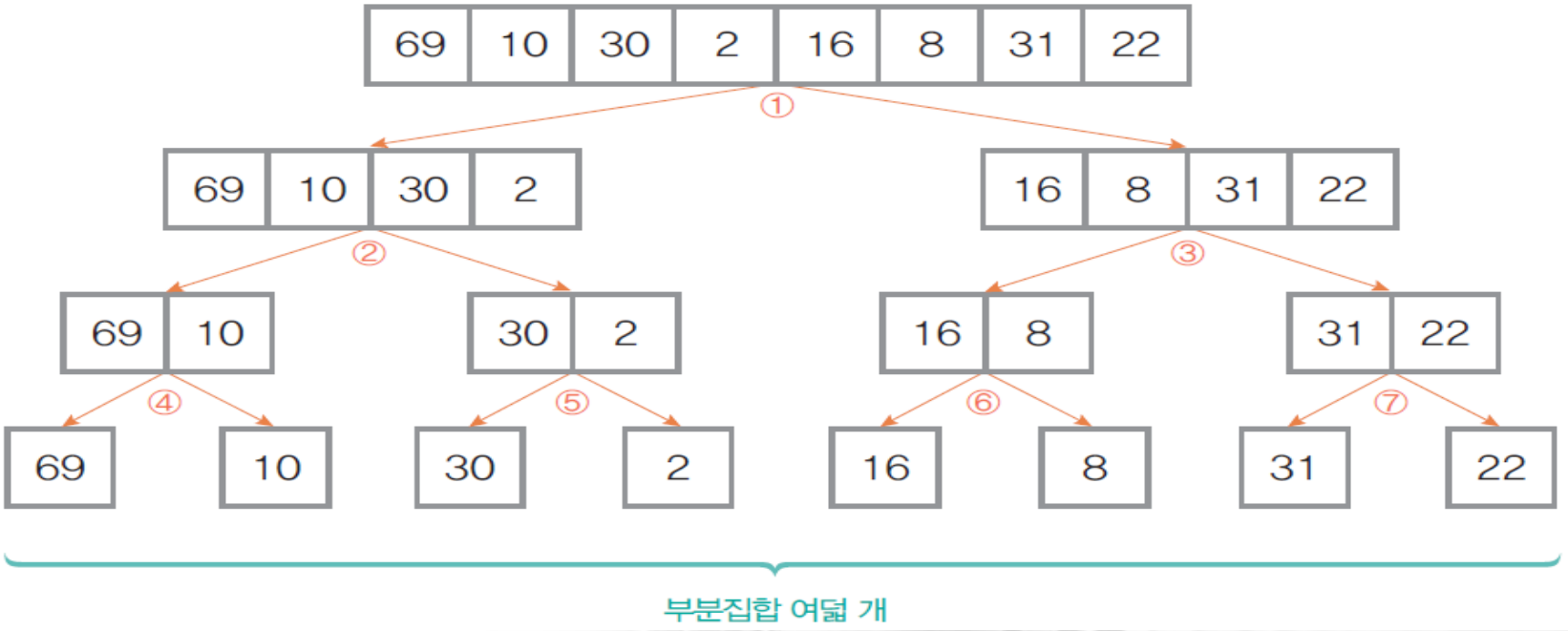
❖ Merge Sort

- ✓ 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
- ✓ 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법 사용
- ✓ 병합 정렬 방법의 종류
  - 2-way 병합 : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
  - n-way 병합 : n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
- ✓ 2-way 병합 정렬

- 분할Divide : 자료들을 두 개의 부분집합으로 분할한다.
- 정복Conquer : 부분집합에 있는 원소를 정렬한다.
- 결합Combine : 정렬된 부분집합들을 하나의 집합으로 정렬하여 결합한다.

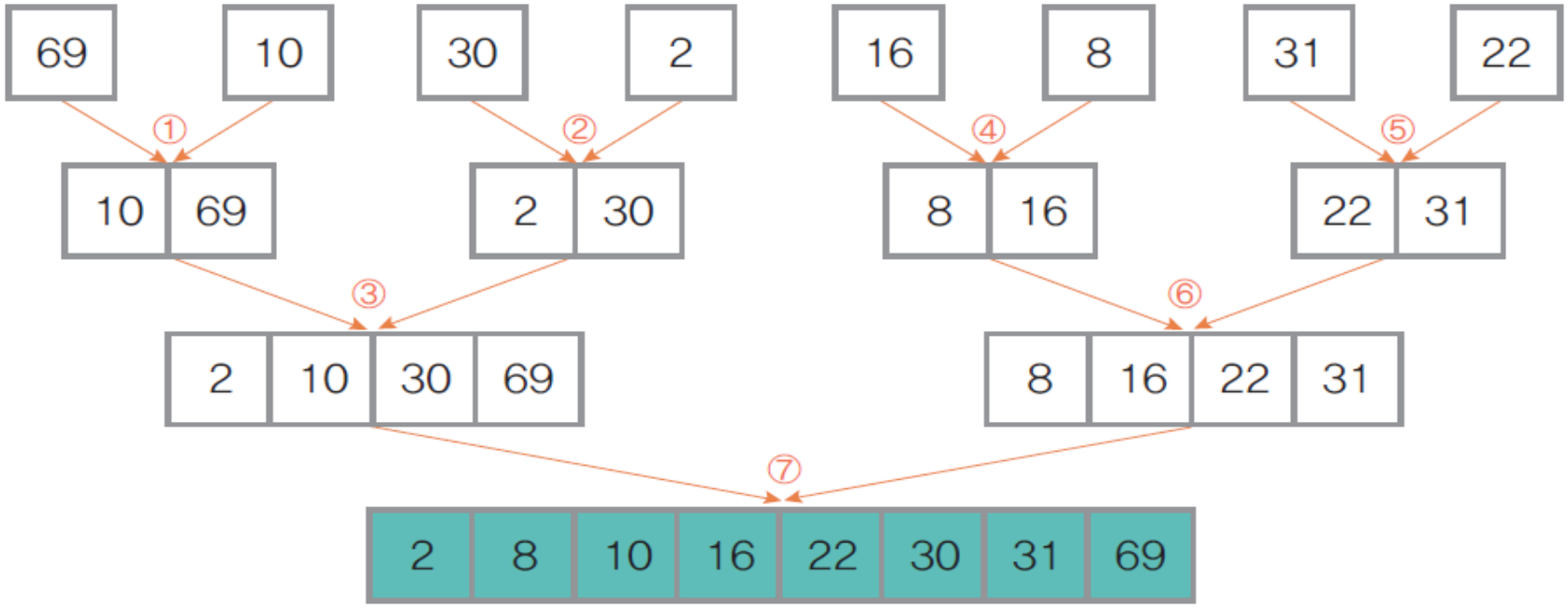
Merge Sort

- Merge Sort
  - ✓ 2-way 병합 정렬
    - 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정
      - 분할 단계 : 정렬할 전체 자료의 집합에 대해서 최소 원소의 부분집합이 될 때까지 분할 작업을 반복하여 한 개의 원소를 가진 부분집합 8개 만듦



Merge Sort

- Merge Sort
  - ✓ 2-way 병합 정렬
    - 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정
      - 정복과 결합 단계 : 부분집합 두 개를 정렬하여 하나로 결합. 전체 원소가 집합 하나로 묶일 때까지 반복

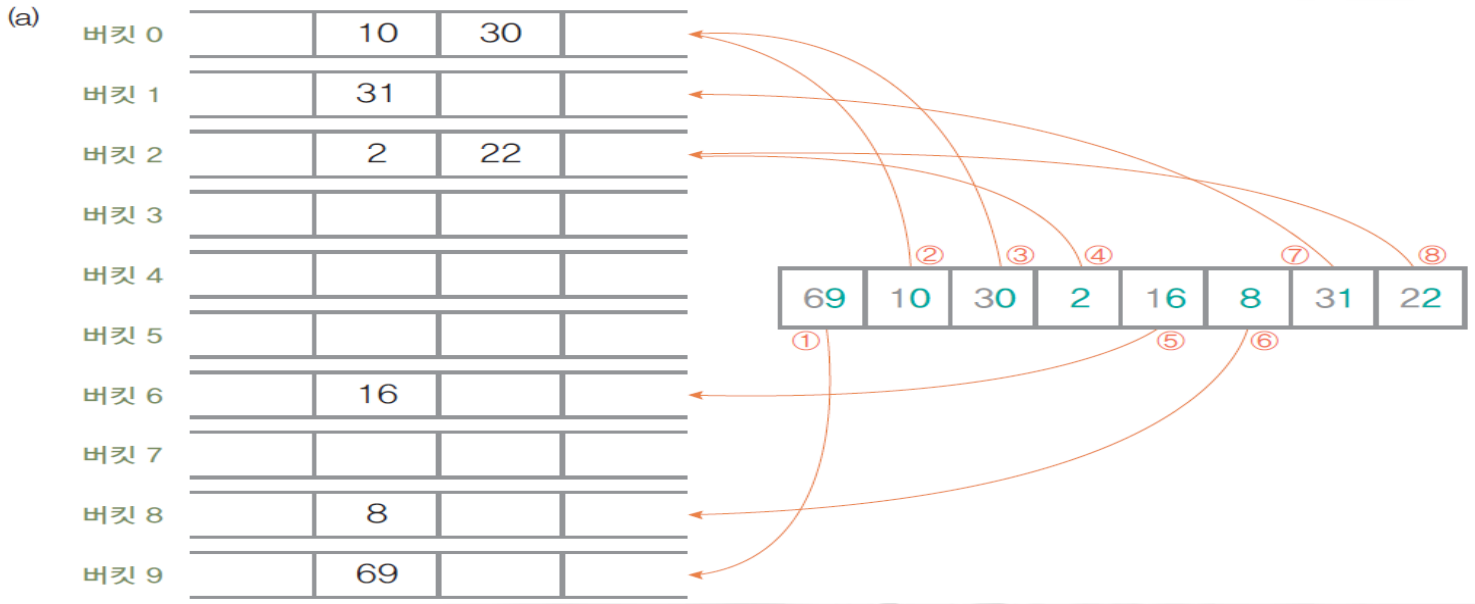


정렬

- Radix Sort
  - ✓ 원소의 키값을 나타내는 기수를 이용한 정렬 방법
  - ✓ 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
    - 원소의 키를 표현하는 기수만큼의 버킷 사용
    - 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0부터 9까지 10개의 버킷 사용
  - ✓ 키 값의 자리 수 만큼 기수 정렬을 반복
    - 키 값의 일의 자리에 대해서 기수 정렬을 수행
    - 다음 단계에서는 키 값의 십의 자리에 대해서
    - 다음 단계에서는 백의 자리에 대해서 기수 정렬 수행
  - ✓ 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷의 순서대로 꺼내서 다음 단계의 기수 정렬을 수행해야 하므로 큐를 사용하여 버킷을 만듦

정렬

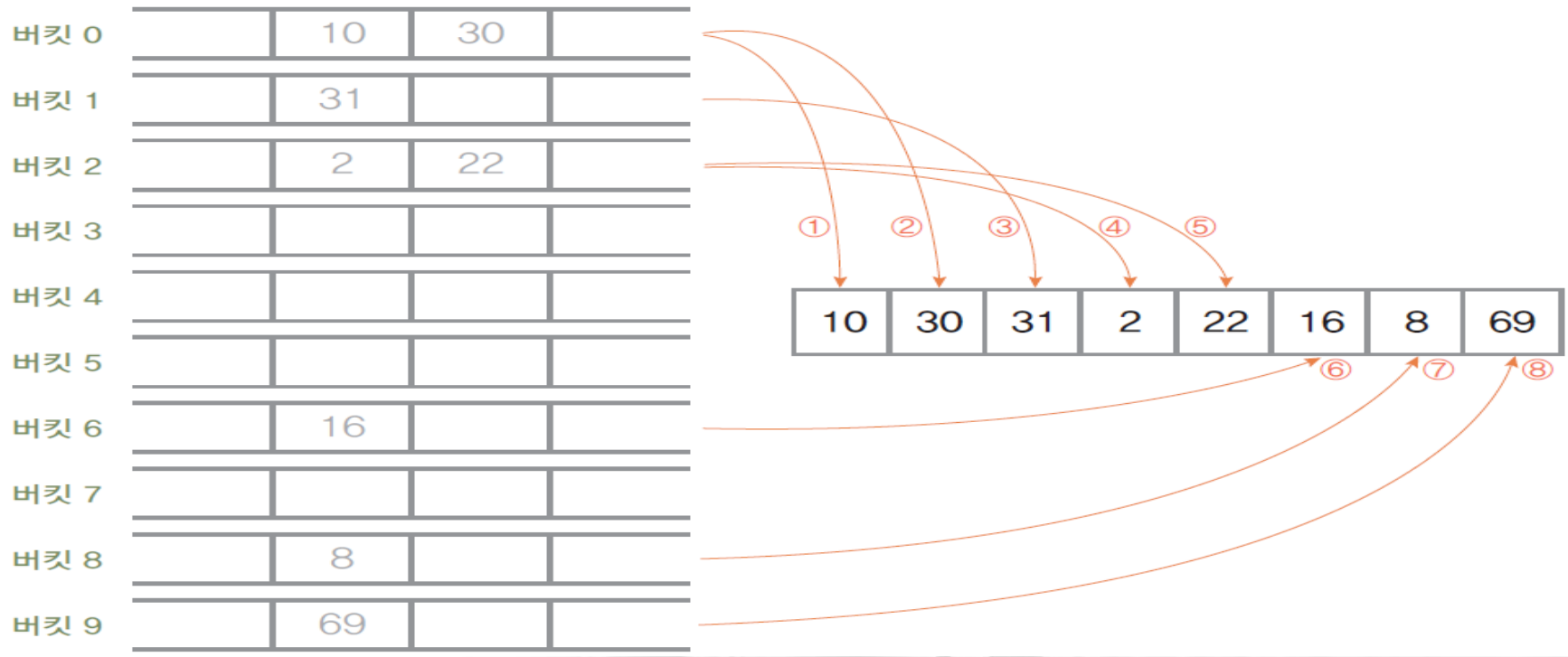
- Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - ○ 키값이 10진수이므로 0부터 9까지 열 개의 버킷을 사용하고, 키값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - ▪ 키값의 1의 자리에 대해서 기수 정렬을 수행
        - a) 정렬할 원소의 키값의 1의 자리에 맞춰 버킷에 분배
        - b) 버킷에 분배되어 있는 원소들을 버킷 0부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장



정렬

- Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - 키값이 10진수이므로 0부터 9까지 열 개의 버킷을 사용하고, 키값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - 키값의 1의 자리에 대해서 기수 정렬을 수행

(b)

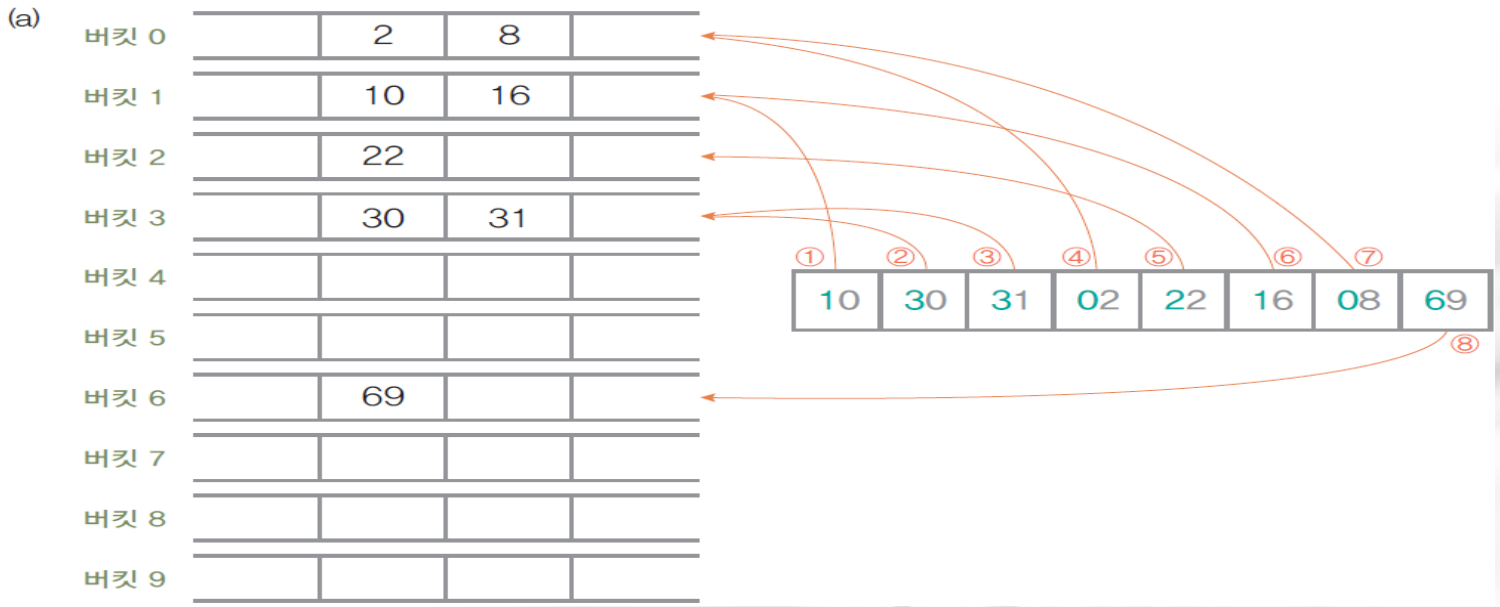


정렬

❖ Radix Sort

✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬

- 키값이 10진수이므로 0부터 9까지 열 개의 버킷을 사용하고, 키값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
  - 키값의 10의 자리에 대해서 기수 정렬을 수행.
    a) 정렬할 원소의 10의 자리에 대해서 버킷에 분배.
    b) 버킷에 분배되어 있는 원소들을 버킷 0부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장하면 전체 원소에 대한 정렬이 완성



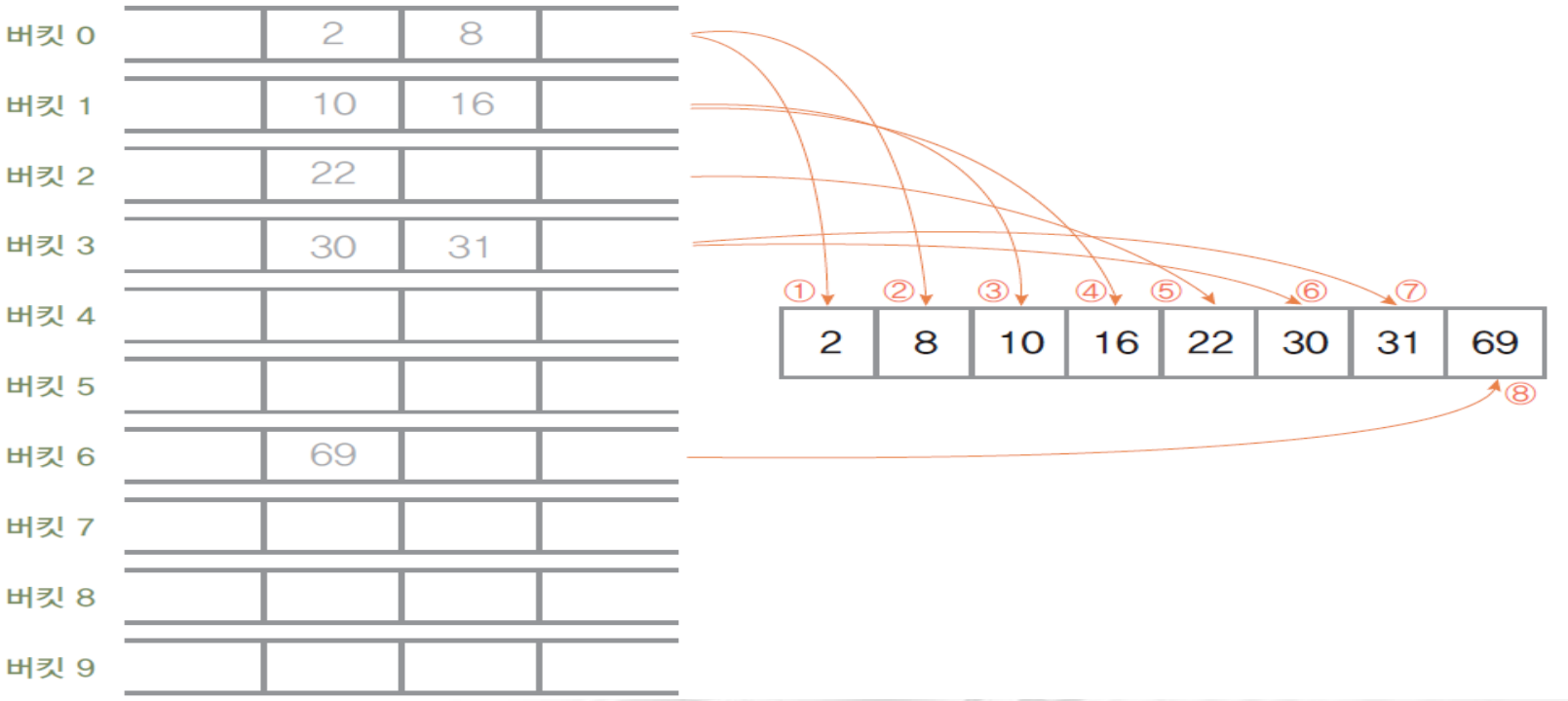
정렬

❖ Radix Sort

✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬

- 키값이 10진수이므로 0부터 9까지 열 개의 버킷을 사용하고, 키값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
  - 키값의 10의 자리에 대해서 기수 정렬을 수행

(b)



정렬

- Heap Sort
  - ✓ 히프 자료구조를 이용한 정렬 방법
  - ✓ 히프에서는 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
  - ✓ 최대 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
  - ✓ 최소 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행
  - ✓ 오름차순 정렬
    - ○ 최대 heap을 구성
    - ○ 원소 개수 만큼의 배열을 생성
    - ○ heap의 삭제 연산을 데이터의 개수만큼 반복수행하는데 이 때 리턴되는 데이터를 배열의 마지막에서부터 저장
    - ○ 삭제 연산이 종료되면 배열의 데이터도 오름차순으로 정렬됨

Search

- 순차 검색
  - ✓ 데이터가 정렬되어 있지 않은 경우 첫 번째 데이터부터 마지막 데이터까지 모두 확인해서 찾는 방법
  - ✓ 정리되지 않은 서랍에서 물건을 찾는 경우와 동일
  - ✓ 평균 비교 횟수가 데이터가 있을 확률을 0.5라고 한다면

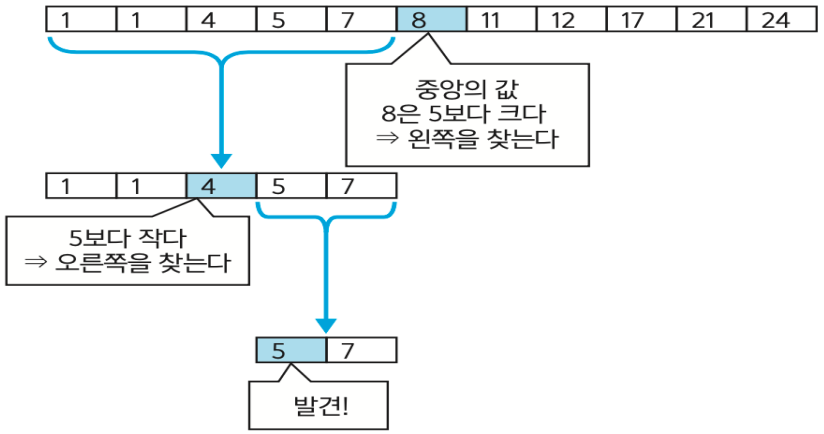
    0.5 * n + 0.5 * n / 2
  - ✓ 여러 번 검색해야 하는 경우 데이터가 많거나 찾고자 하는 데이터가 배열에 없는 경우가 많을 경우 비 효율적인 방법

검색

❖ 이분 검색

✓ 데이터가 정렬되어 있는 경우 중앙 값과 비교해서 작으면 왼쪽으로 크면 오른쪽으로 이동해서 검색하는 방법

✓ 배열에서 5를 찾는 경우



1)low = 0(데이터의 시작위치), high = n(데이터의 개수)-1로 초기화

2)무한 반복 문에서

low > high 이면 데이터가 없는 것이므로 break;

low > high 가 아니면 middle = (low + high)/2를 해서 middle값을 찾는다.

검색 값과 data[middle]과 비교해서 같으면 찾은 것이므로 출력하고 break;

검색 값이 중앙값보다 크다면 low = middle +1

작다면 high = middle -1을 해서 반복

검색

- 해시(Hash) 알고리즘
  - ✓ 해시 테이블(Hash Table)에 해시 함수(Hash Function)을 이용하여 자료를 저장하거나 검색하는 자료구조 및 알고리즘
  - ✓ 해시 함수에 의해 자료를 저장할 위치나 저정한 위치를 계산하는 것을 해싱(Hashing)이라 함
  - ✓ 해싱(Hashing)은 해시 함수에 위해 결정한 위치에 직접 접근(Direct Access Method)이 가능
  - ✓ 검색 속도가 가장 빠름
  - ✓ 키를 해시 함수에 의해 보관할 혹은 보관한 주소로 변환하는 방식