SAMSUNG 삼성SDS

SW 아카데미 알고리즘 특강

Contents

오늘의 원리

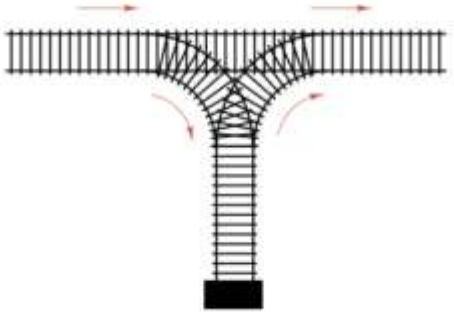
- 자료구조(Data Structure)란 효율적으로 접근하고 수정할 수 있도록 데이터를 구성하고 저장하는 방법을 이야기한다.
- Tree는 사이클이 없는 그래프로 루트노드를 시작으로 계층 구조를 이룬다.
- 자식이 최대 2개 까지 존재하는 Tree를 Binary Tree(이진 트리)라고 부른다.

학습목표

- 자주 사용하는 자료구조의 구조를 이해하고 구현 할 수 있다.
- 다양한 자료구조를 학습하고 API 사용 방법을 익혀 활용할 수 있다.

사전문제

1) 옆에 그림과 같은 구조의 선로가 있다고 하자. 기차들은 그림과 같이 왼쪽에서 오른쪽으로 이동할 수 있으며, 잠시 대기소로 들어가서 기차를 돌린 뒤 나올 수도 있다. 대기소 특성상 나중에 들어간 기차가 먼저 나오게 되어있다.



1,2,3,4,5 순서대로 기차가 지나간다고 할 때, 3,2,5,4,1 순서대로 기차가 나올 수 있는지 구하시오.

2) 다음 상황에서 크기가 10만인 배열의 구간(i~j) 합을 빠르게 구하는 방법에 대하여 논의하시오.

1. 구간 합을 한번만 구하는 경우
2. 구간 합을 여러 차례 구하는 경우
3. 구간 합을 여러 차례 구하는 중간에 배열의 값이 바뀌는 경우.

사전문제 접근법

1) 가장 먼저 나와야 하는 기차는 어떤 기차인지 고민해본다.

2) 각 경우 별로 미리 구해두어야 하는 값을 계산해본다.

문제풀이 방법

(1) 3 -> 2 -> 5 -> 4 -> 1 순서대로 지나가야 하므로 1, 2 번을 대기소에 넣고 3번을 보낸 뒤, 대기소서 2번을 뺀다. 4번을 대기소에 넣고 5번을 보낸 뒤 4번 1번 순서로 빼내면 위 순서대로 지나갈 수 있음을 알 수 있다.

(2) 단 한번만 구간합을 구하는 경우는 i~j까지 다 더하는 것이 최적이다. O(N)

여러 차례 구할 경우 누적 합을 이용하는 방법이 최적이다.
D[i] = D[i-1]+A[i] 라는 기조로 i~j까지 구간합을 구할 경우 D[j]-D[i-1] 로 한번에 구할 수 있다.

마지막 방법은 이후 설명할 인덱스 트리를 사용하여 해결한다.

자료구조의 정의 및 분류

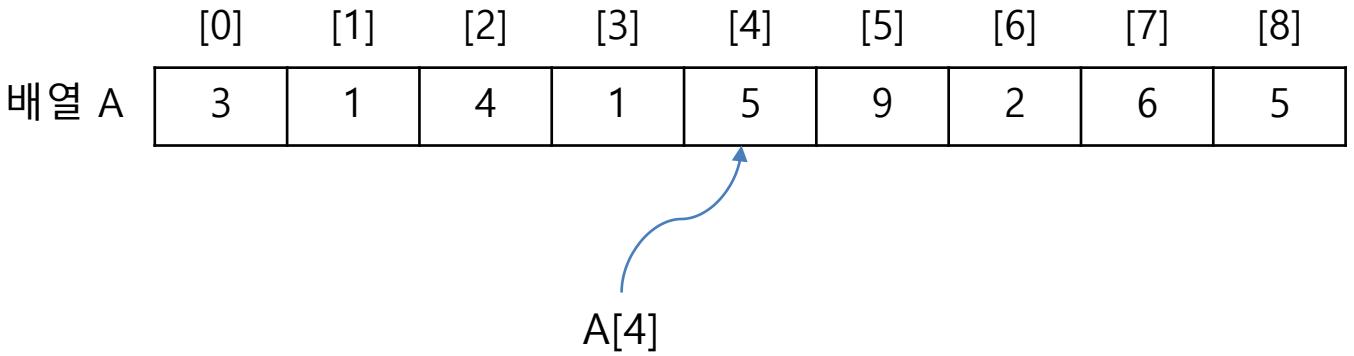
- **자료구조(Data Structure)란 효율적으로 접근하고 수정할 수 있도록 데이터를 구성하고 저장하는 방법을 이야기한다.**

- 자료구조는 저장되는 데이터의 형태에 따라 선형 자료구조와 비선형 자료구조로 구분된다.
  - 선형 자료구조 : 데이터가 일렬로 나열되어 있음
    - 배열 (Array)
    - 연결 리스트 (Linked List)
    - 스택 (Stack)
    - 큐 (Queue)
  - 비선형 자료구조 : 데이터가 특정한 형태를 띄고 있음
    - 트리 (Tree)
    - 그래프 (Graph)

배열 (Array)

가장 기본적인 자료구조로 동일한 자료형의 데이터를 일렬로 나열한 자료구조.

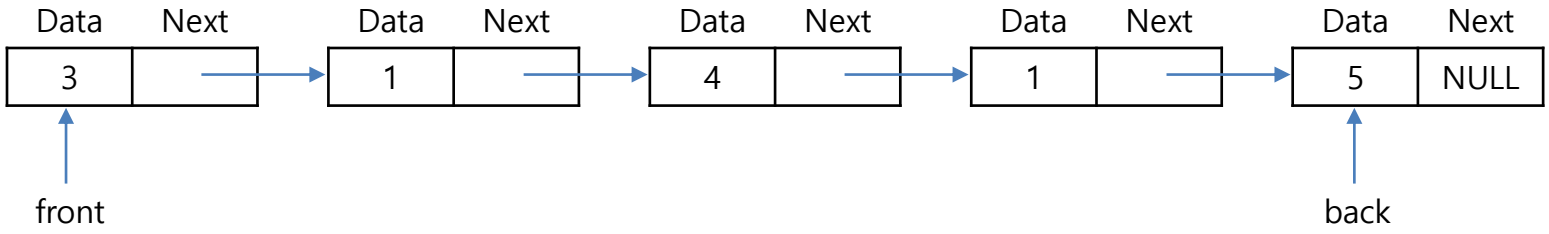
- 배열의 특징
  - 데이터 접근이 용이.
  - 데이터 삽입/삭제가 어려움.
  - 구조가 간단하여 프로그램 작성이 쉬움.



연결 리스트 (Linked List)

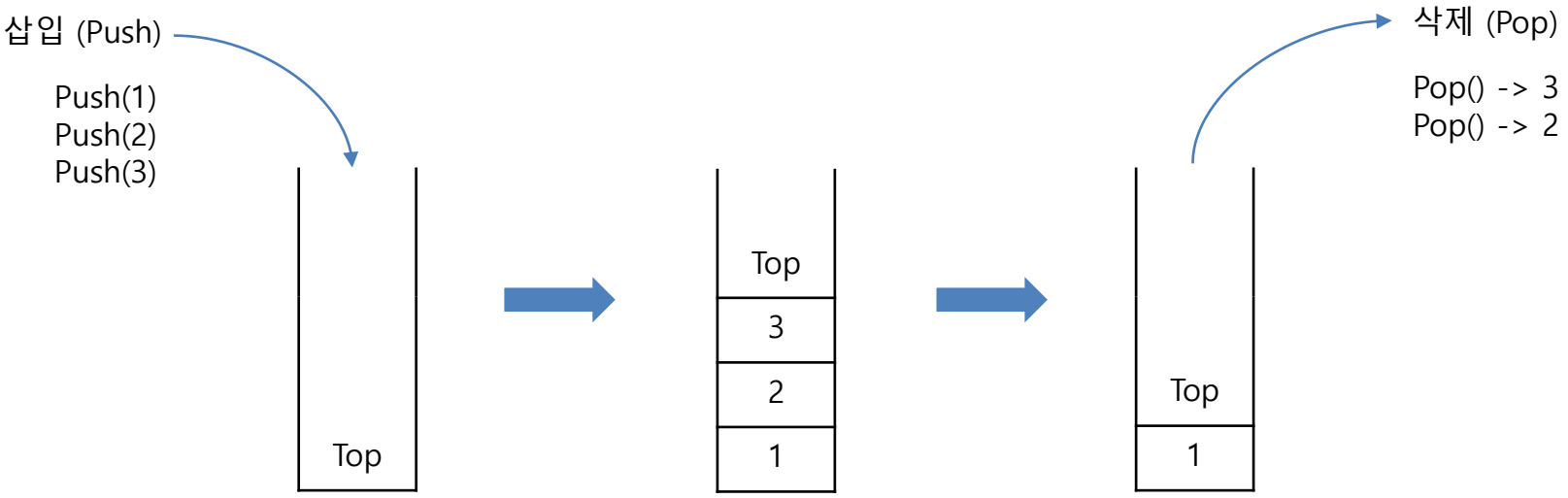
각 노드가 데이터와 포인터를 가지고 일렬로 연결되어있는 방식으로 데이터를 저장하는 자료구조.

- 연결 리스트의 특징
  - 배열과 반대되는 특징을 가짐.
  - 데이터의 접근이 느림
  - 데이터의 삽입/삭제 연산이 용이
  - 포인터를 위한 추가 공간 필요, 프로그램 작성이 어려움



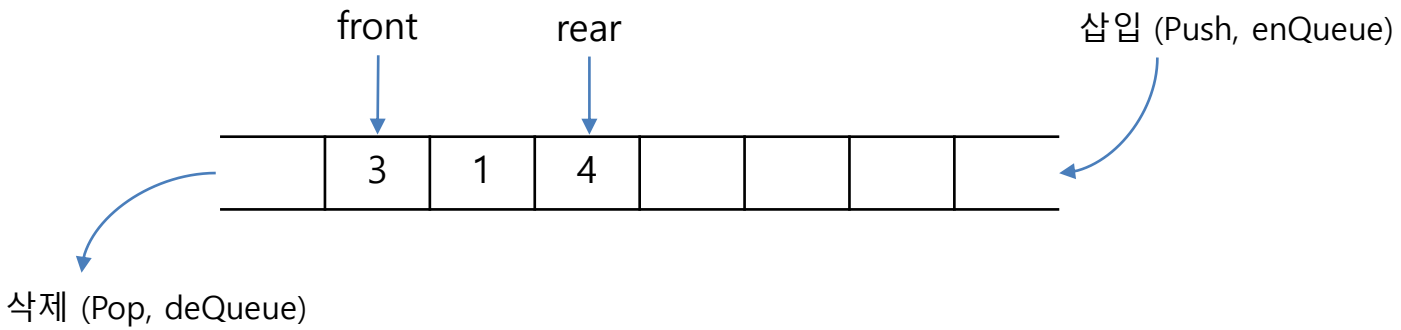
스택 (Stack)

- 스택 (Stack)
  - 삽입연산, 삭제연산이 한 방향에서 이루어지는 선형 자료구조.
  - 한 방향에서 삽입과 삭제가 이루어지기 때문에 나중에 삽입된 데이터가 먼저 삭제되는 후입선출(LIFO, Last In First Out) 구조를 가진다.
  - 스택에 데이터가 삽입될 위치를 Top이라고 한다.



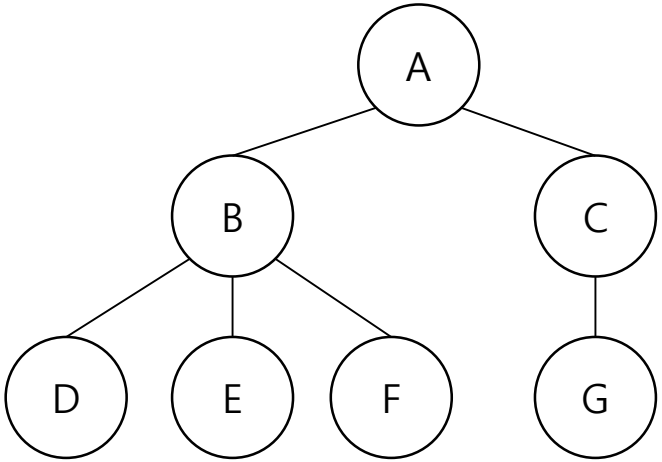
큐 (Queue)

- 큐 (Queue)
  - 한 방향에서는 삽입연산이, 반대편에서는 삭제연산이 이루어지는 자료구조.
  - 스택과 마찬가지로 선형 자료구조이다.
  - 한쪽 방향에서 삽입이, 반대편에서 삭제가 이루어지기 때문에 먼저 삽입된 데이터가 먼저 삭제되는 선입선출(FIFO, First In First Out) 구조를 가진다.
  - 데이터가 삭제될 위치를 Front / Head, 마지막 데이터가 삽입된 위치를 Rear / Tail 이라 부른다.
  - 양 방향에서 삽입연산과 삭제연산이 모두 이루어지는 큐를 덱(Deque)이라고 한다.



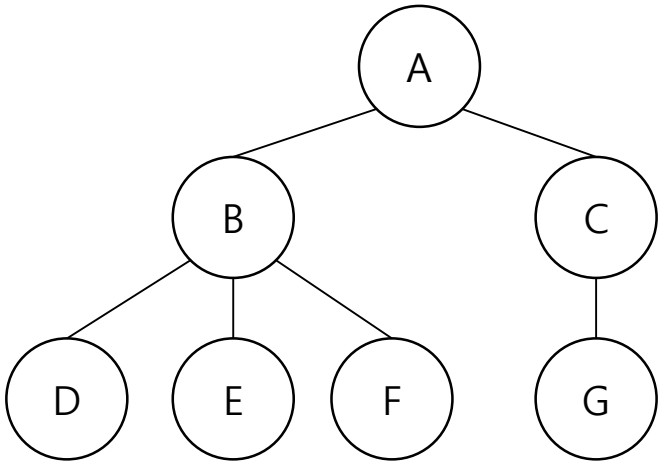
트리 (Tree)

- 트리 (Tree)
  - 자료들 사이의 계층적 관계를 나타내는데 사용하는 자료구조로 부모-자식 관계로 표현된다.
  - 트리는 1개 이상의 노드를 갖는 집합으로 다음의 조건을 만족한다.
    - 루트(root) 노드가 존재한다.
    - 트리의 부분트리(Sub Tree) 또한 트리 구조를 따른다.



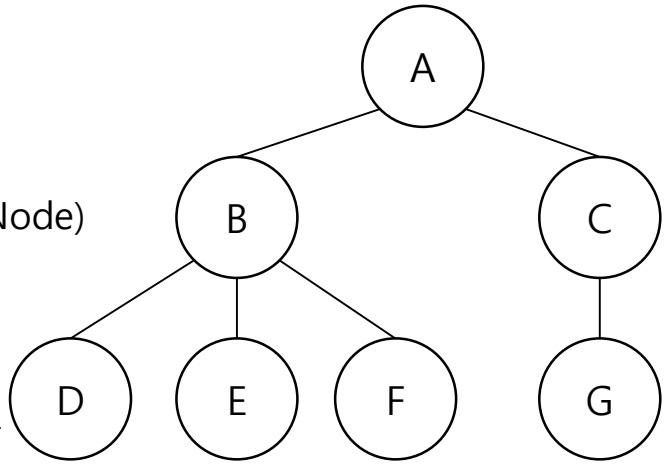
트리 (Tree)의 용어

- 루트 노드 (Root Node)
  - 트리의 최상위 노드. 유일하다.
- 깊이 (Depth)
  - 루트 노드에서 해당 노드까지 도달하는데 사용하는 간선의 개수
  - 루트 노드는 자기 자신까지 도달하는 간선의 개수가 0이므로 루트 노드의 깊이는 0이다.
- 레벨 (Level)
  - 노드의 깊이 +1
- 부모 노드 (Parent Node)
  - 부모 자식 관계에서 상위 계층에 있는 노드.
  - 상위 계층에 있을수록 노드의 깊이와 레벨이 낮다.
- 자식 노드 (Child Node)
  - 부모 자식 관계에서 하위 계층에 있는 노드.
- 형제 노드
  - 부모가 동일한 노드.



트리 (Tree)의 용어

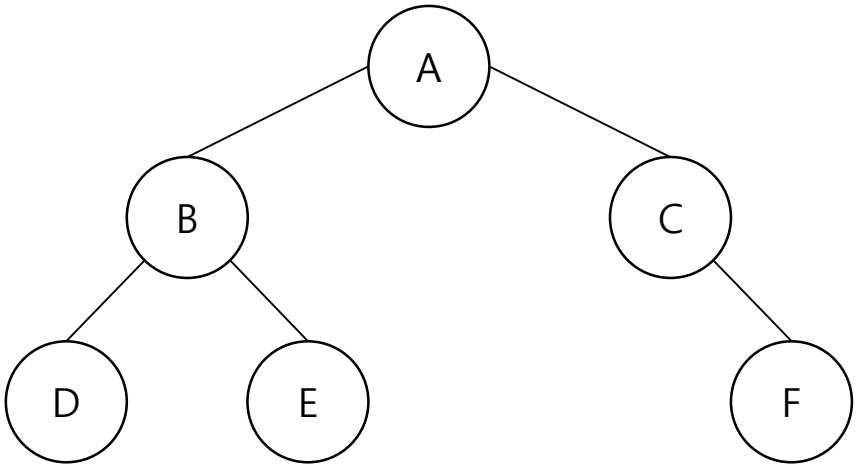
- 조상 노드
  - 해당 노드의 부모노드부터 루트노드까지 가는 경로에 존재하는 노드들을 그 노드의 조상 노드라고 한다.
- 후손 노드
  - 해당 노드를 루트로 하는 서브트리에 있는 모든 노드를 그 노드의 후손 노드라고 한다.
- 노드의 분지 수 (노드의 차수, Degree)
  - 노드의 자식 수
- 트리의 분지 수 (트리의 차수)
  - 해당 트리 내 모든 노드의 분지 수 중 최댓값
- 내부 노드 (internal/nonterminal Node)
  - 자식이 있는 노드
- 외부 노드 (단말 노드, 잎새 노드, leaf/external/terminal Node)
  - 자식이 없는 노드
- 높이 (Height)
  - 루트 노드에서 해당 노드까지 도달하는데 지나간 정점의 개수
  - 노드 중 가장 높이가 높은 노드의 높이를 트리의 높이라고 한다.



이진 트리 (Binary Tree)

트리의 분지 수가 2 이하인 트리. 대부분의 트리 자료구조는 이진 트리 형태에서 나온다.

- 이진 트리의 특징
  - 자식이 최대 2개이기 때문에 자식을 왼쪽 자식과 오른쪽 자식으로 구분한다.
  - 높이가 N인 이진 트리의 최대 노드 개수는 $2^N - 1$ 개이다.



이진 트리 (Binary Tree)의 종류

- 정 이진 트리 (Full Binary Tree)
  - 모든 노드의 차수가 0 또는 2인 이진 트리

- 포화 이진 트리 (Perfect Binary Tree)
  - 정 이진 트리에서 모든 단말 노드의 깊이가 같은 이진 트리
  - 높이가 H인 포화 이진 트리의 노드 개수는 2^H-1개이다.
  - 반대로 노드가 N개인 포화 이진 트리의 높이는 $\log_2(N+1)$이다.
  - 깊이가 D인 포화 이진 트리의 단말 노드 개수는 2^D개이다.

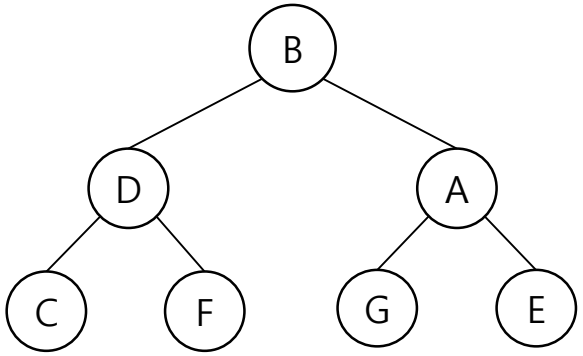
- 완전 이진 트리 (Complete Binary Tree)
  - 마지막 레벨은 노드가 왼쪽에 몰려있고 마지막 레벨을 제외하면 포화 이진 트리 구조를 띄고 있는 이진 트리.

- 사향 이진 트리 (Skewed Binary Tree)
  - 연결리스트처럼 한 줄로 연결 되어 있는 형태의 이진 트리

이진 트리 (Binary Tree)의 표현

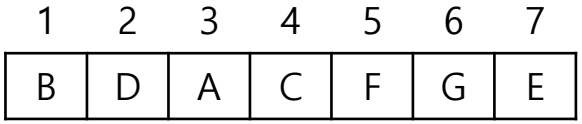
- 연결 자료 구조 - 연결리스트 형태의 자료구조

```
class Node
    Object data
    Node left_child, right_child
```



- 연속 구조 - 일차원 배열을 이용한 구현
  - 배열에 트리 노드를 레벨 순서대로 왼쪽에서 오른쪽으로 저장한다. 완전 이진 트리로 가정하고 배열의 1번 위치부터 저장을 시작한다고 가정했을 때 i번 노드의 부모, 왼쪽 자식, 오른쪽 자식을 구하는 방법은 다음과 같다.

```
function Parent( i )
    return i/2
```




```
function Left_Child( i )
    return i*2
```

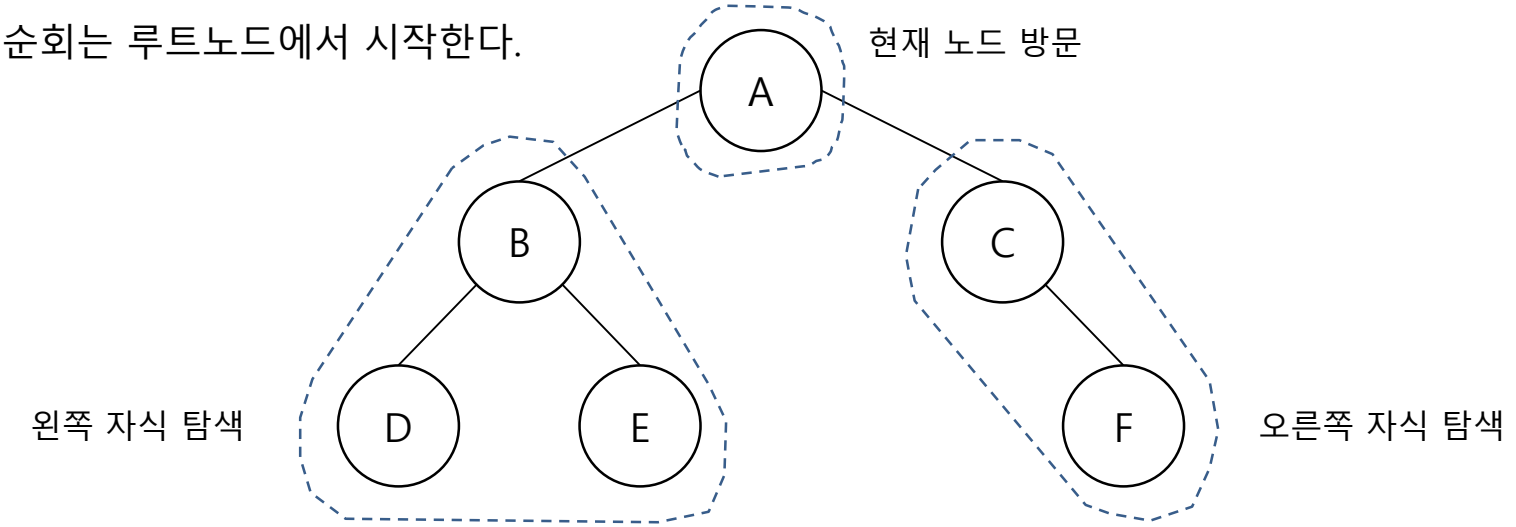
```
function Right_Child( i )
    return i*2+1
```

이진 트리 (Binary Tree)의 순회

- 트리는 비 선형 자료구조이기 때문에 모든 노드를 for문 한번으로 방문이 불가능하다. 이런 트리 구조에서 모든 노드를 방문하기 위한 과정을 트리의 순회라고 한다.

- 특히 이진 트리의 순회는 왼쪽 자식 탐색, 오른쪽 자식 탐색, 현재 노드 방문의 세 가지 주요 과정을 통해 진행 되며, 노드를 방문하는 순서에 따라 전위 순회(pre-order), 중위 순회(in-order), 후위 순회(post-order)로 나뉜다.

- 모든 순회는 루트노드에서 시작한다.



이진 트리 (Binary Tree)의 순회

- **전위 순회(pre-order)**

  1. 현재 노드 방문
  2. 왼쪽 자식 탐색
  3. 오른쪽 자식 탐색

  방문 순서 : A – B – D – E – H - C – F – G

- **중위 순회(in-order)**

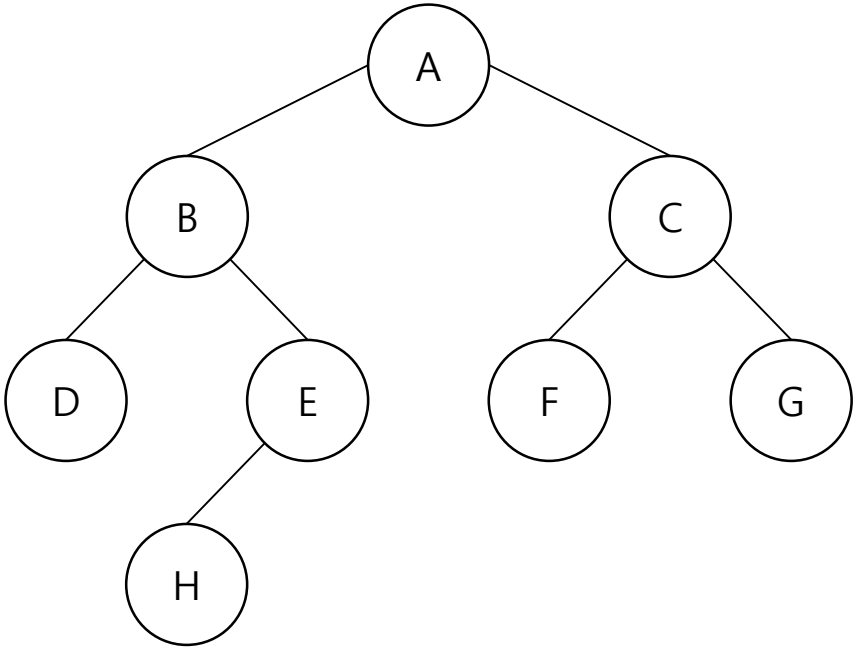
  1. 왼쪽 자식 탐색
  2. 현재 노드 방문
  3. 오른쪽 자식 탐색

  방문 순서 : D – B – H – E – A – F – C – G

- **후위 순회(post-order)**

  1. 왼쪽 자식 탐색
  2. 오른쪽 자식 탐색
  3. 현재 노드 방문

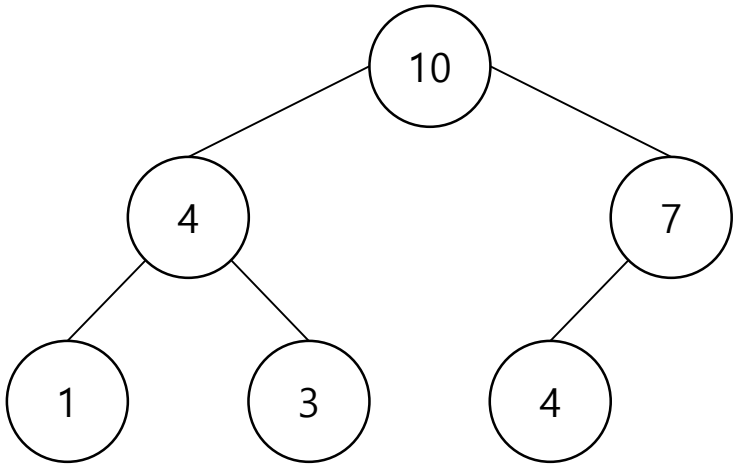
  방문 순서 : D – H – E – B – F – G – C - A



이진 트리 (Binary Tree)의 응용 – 힙 (Heap)

- 힙 (Heap)
  - 완전 이진트리 형태의 자료구조
  - 힙 조건 (Heap Condition)을 만족한다.
    - 힙 조건 예시 : 각 노드의 키 값은 자식 노드의 키 값보다 더 크다. (그림 참고)
  - 일차원 배열로 구현한다.
  - 일반적으로 그룹을 정렬(Heap Sort)하거나 입력된 데이터 안에서 최소/최대 값을 찾을 때 사용한다. 데이터의 삽입과 삭제가 빠르며 각각의 수행시간이 O(logN)이다.

1	10
2	4
3	7
4	1
5	3
6	4



이진 트리 (Binary Tree)의 응용 – 힙 (Heap)

- 최소 힙 (Min Heap)과 최대 힙(Max Hap)

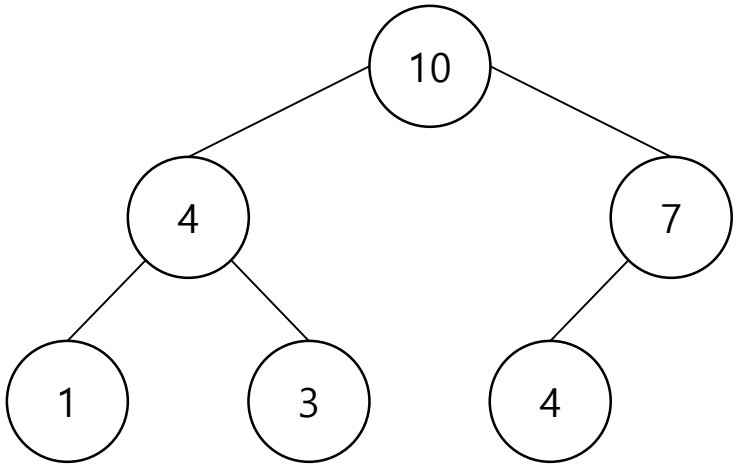
  최소 힙과 최대 힙은 각각 다음 Heap Condition을 가지는 대표적인 유형의 힙이다.

  – 최소 힙 : 부모노드의 키 값이 자식노드의 키 값보다 항상 작다.

  – 최대 힙 : 부모노드의 키 값이 자식노드의 키 값보다 항상 크다.

  부모노드의 키 값이 자식노드의 키 값보다 작은(큰) 관계가 유지되므로 트리 내에서 가장 작은(큰) 키 값을 가진 노드는 항상 루트 노드가 된다. 아래는 Max Heap의 예시이다.

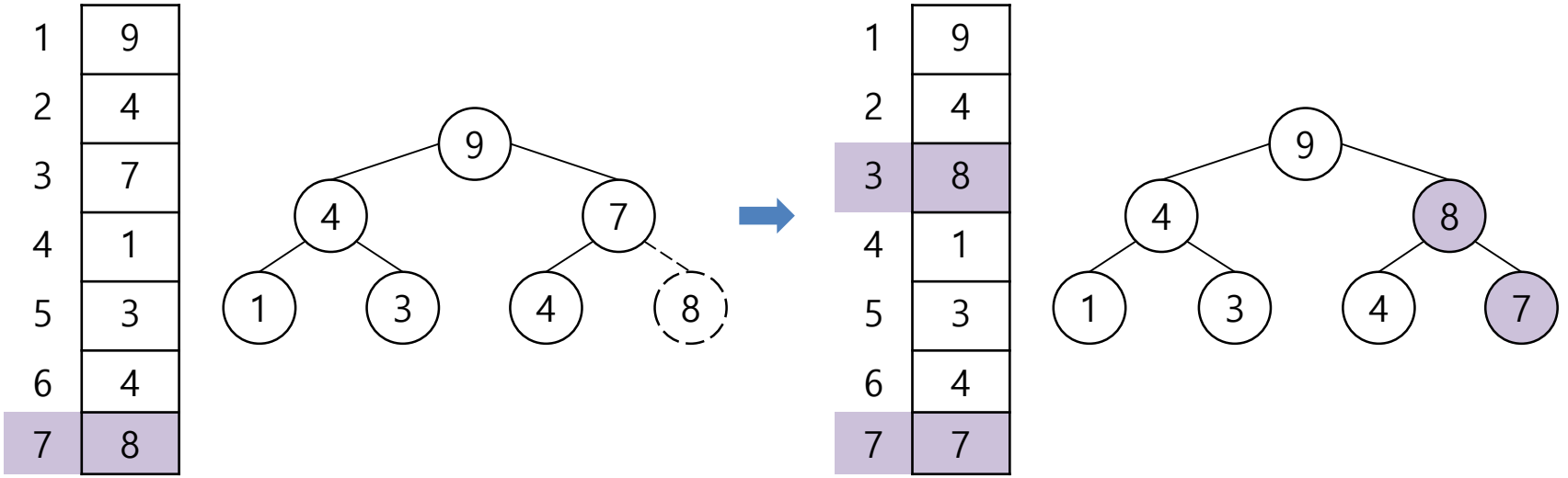
1	10
2	4
3	7
4	1
5	3
6	4



힙 (Heap)의 구현

- 힙 (Heap) 의 삽입 연산
  1. 트리의 가장 마지막 위치에 노드를 삽입한다.
  2. 추가된 노드와 그 부모 노드가 힙 조건을 만족하는지 확인한다.
  3. 만족하지 않는다면 부모와 자식의 키 값을 바꾼다.
  4. 조건에 만족하거나 추가된 노드가 루트에 도달할 때까지 2 ~ 3 을 반복한다.

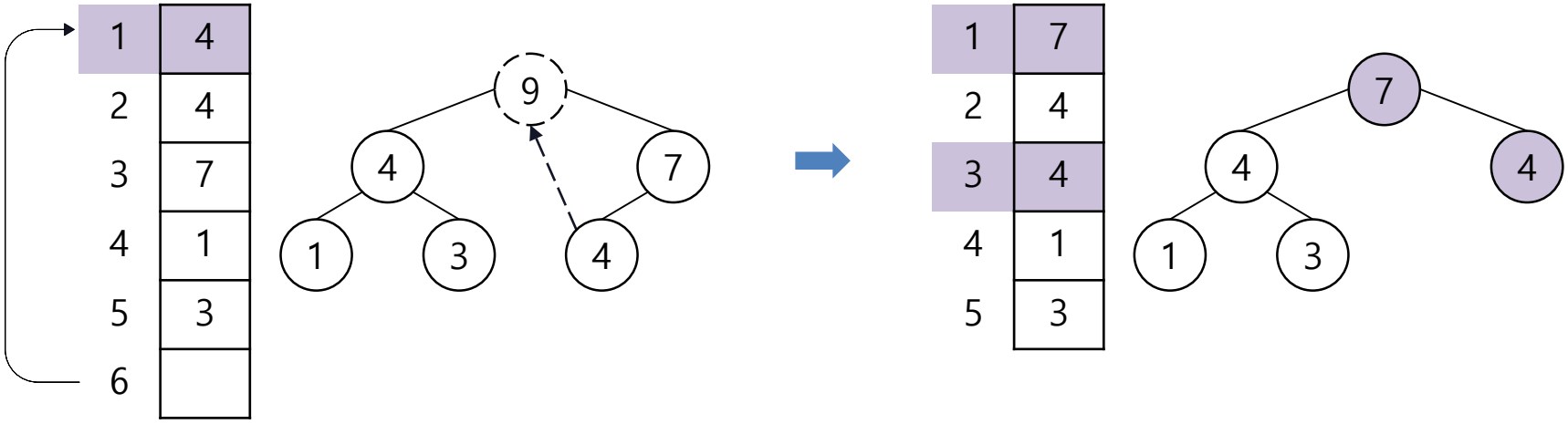
❖ 완전 이진 트리의 리프 노드부터 루트 노드까지 연산을 함. 노드가 N개일 때 수행 시간 O(logN)



힙 (Heap)의 구현

- 힙 (Heap) 의 삭제 연산
  1. 힙의 삭제연산은 항상 루트 노드를 삭제한다.
  2. 트리의 가장 마지막 노드를 루트 자리로 삽입한다.
  3. 바꾼 위치의 노드가 힙 조건을 만족하는지 확인한다.
  4. 만족하지 않는다면 왼쪽 자식과 오른쪽 자식 중 적합한 노드와 키 값을 바꾼다.
  5. 조건을 만족하거나 리프 노드에 도달할 때까지 3 ~ 4를 반복한다.

❖ 완전 이진 트리의 루트 노드부터 리프 노드까지 연산을 함. 노드가 N개일 때 수행 시간 O(logN)



힙 (Heap)의 활용 – 우선순위 큐(Priority Queue)

- 우선순위 큐 (Priority Queue)
  - 각 데이터에 우선순위를 부여하여 큐에 넣은 데이터를 꺼낼 때 우선순위가 높은 데이터를 먼저 꺼내는 자료구조.
  - 주로 정의된 우선순위를 Heap Condition으로 가지는 Heap 자료구조를 통해 구현된다.
  - 우선순위 큐는 기본적으로 다음 두 가지 연산을 지원한다.
    1. 큐에 데이터를 우선순위를 지정하여 추가한다.
    2. 큐에서 가장 우선순위가 높은 데이터를 제거하고 반환한다.
  - Heap을 통해 우선순위 큐를 구현하면 위 연산이 아래와 같이 정의된다.
    1. Heap에 데이터를 우선순위를 지정하여 삽입연산을 수행한다.
    2. Heap의 삭제연산을 수행하고 삭제된 데이터를 반환한다.

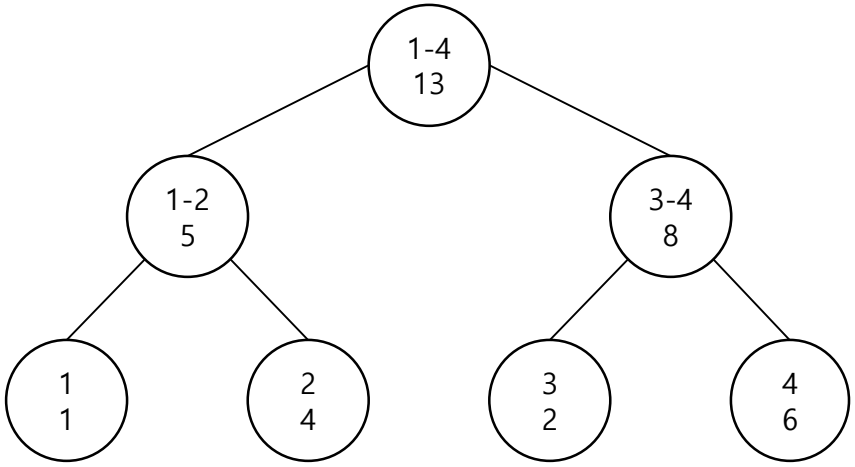
이진 트리 (Binary Tree)의 응용 – 인덱스 트리(Indexed Tree)

- 인덱스 트리(Indexed Tree)
  - 배열 A가 있고 다음 두 연산을 M번 수행해야 하는 문제가 있다고 한다.
    1. 구간 l, r (l<=r)이 주어졌을 때 A[l] + A[l+1] + A[l+2] … … +A[r-1] + A[r]을 구하여라.
    2. i번째 수 A[i]를 V로 바꾸어라.
  - 일반적인 방법으로는 1번 연산을 수행하는데 O(N), 2번 연산을 수행하는데 O(1)의 시간복잡도가 소요되므로 최종 시간복잡도는 O(NM)
  - 2번 연산이 없다고 가정하면 누적합 배열을 이용해 1번 연산을 O(1)에 수행할 수 있다.
  - 인덱스 트리를 사용하면 1번 연산, 2번 연산을 모두 O(logN)에 수행할 수 있다. 최종 시간복잡도 O(MlogN)

	1	2	3	4	5	6	7	8
data	3	2	4	5	1	6	2	7

이진 트리 (Binary Tree)의 응용 – 인덱스 트리(Indexed Tree)

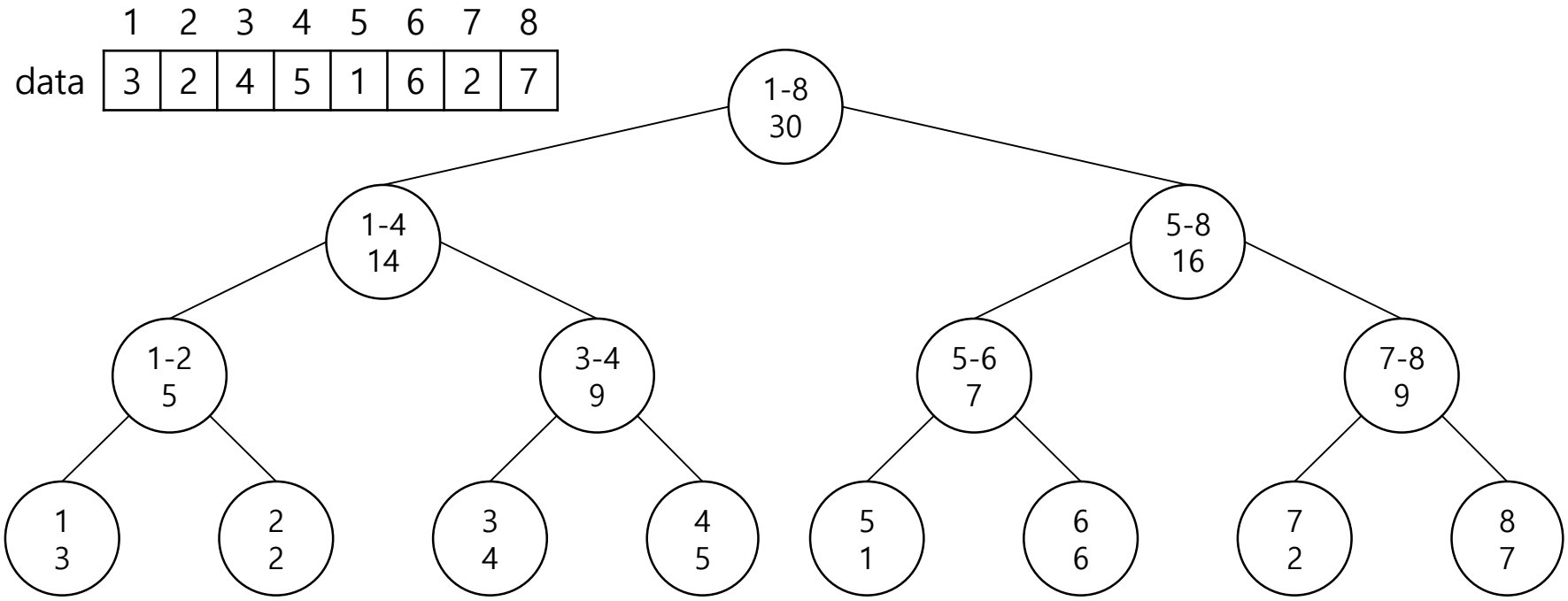
- 인덱스 트리(Indexed Tree)
  - 포화 이진트리 형태의 자료구조
  - 각 노드가 다음과 같은 의미를 가짐
    1. 리프 노드 : 배열에 적혀있는 수
    2. 내부 노드 : 왼쪽 자식과 오른쪽 자식의 합
  - 리프 노드의 개수가 N개 이상인 포화 이진트리는 높이가 최소 logN이다.
  - 리프 노드의 개수가 N개 보다 많아 비어있는 공간이 발생할 경우 구조에 지장이 가지 않도록 초기값을 설정한다.



이진 트리 (Binary Tree)의 응용 – 인덱스 트리(Indexed Tree)

- 인덱스 트리(Indexed Tree) – 예시 – 트리 구성 방법
  1. 리프 노드의 개수가 N개 이상이 되도록 높이가 T인 트리 배열을 만든다. (배열 크기 : 2^T)
  2. 리프 노드에 데이터를 입력한다. (2^{T-1}번 방 ~ 2^{T-1} + N-1번 방)
  3. 내부 노드인 2^{T-1}-1번 방 부터 1번 방 순서로 양쪽 자식 값을 참조하여 데이터를 입력한다.

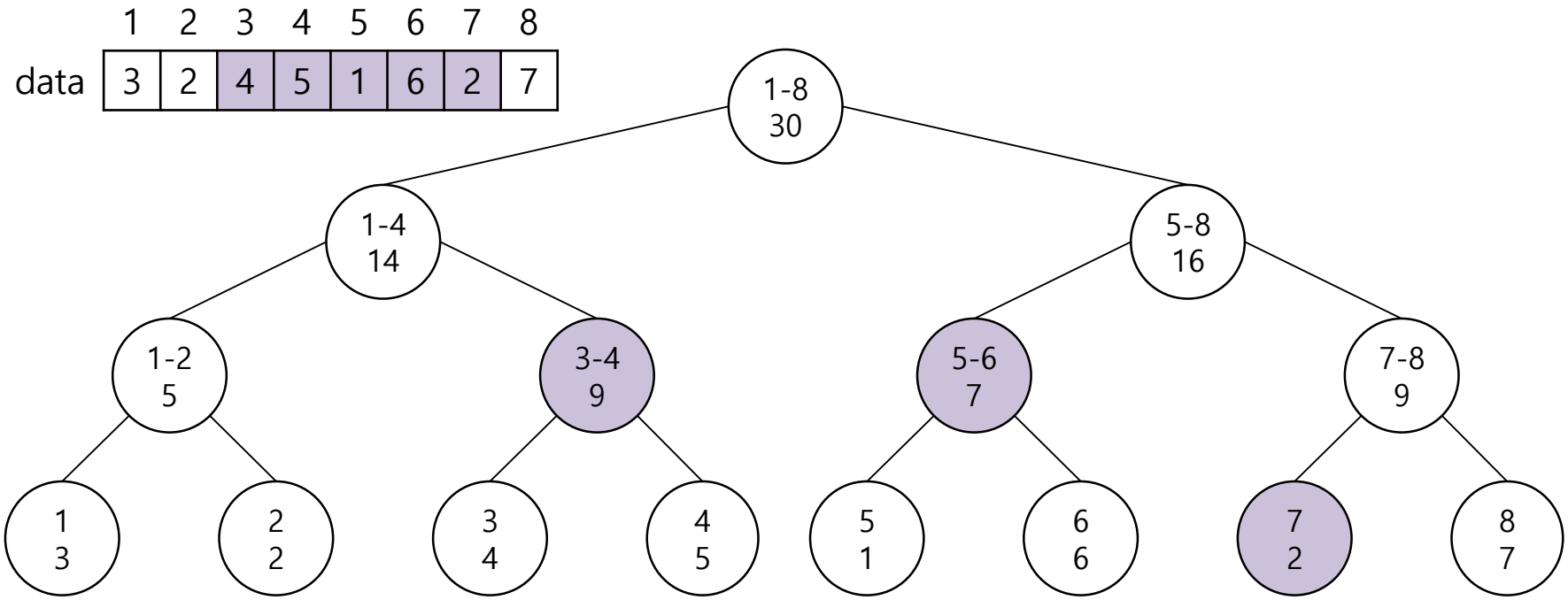
❖ 트리의 노드 개수는 꽉 찬 트리에서 약 2N개 이다. 따라서 수행 시간은 O(N) 이다.



이진 트리 (Binary Tree)의 응용 – 인덱스 트리(Indexed Tree)

- 인덱스 트리(Indexed Tree) – 예시 – 3 – 7 구간의 합을 구하는 쿼리 수행 방법
  1. 루트 노드에서 시작해서 트리를 전위 순회 하는 방식으로 탐색한다.
  2. 현재 노드가 구하고자 하는 구간에 완전히 속해있으면 그 노드의 결과값을 반환한다.
  3. 왼쪽 자식에서 반환된 값과 오른쪽 자식에서 반환된 값을 더해서 반환한다.

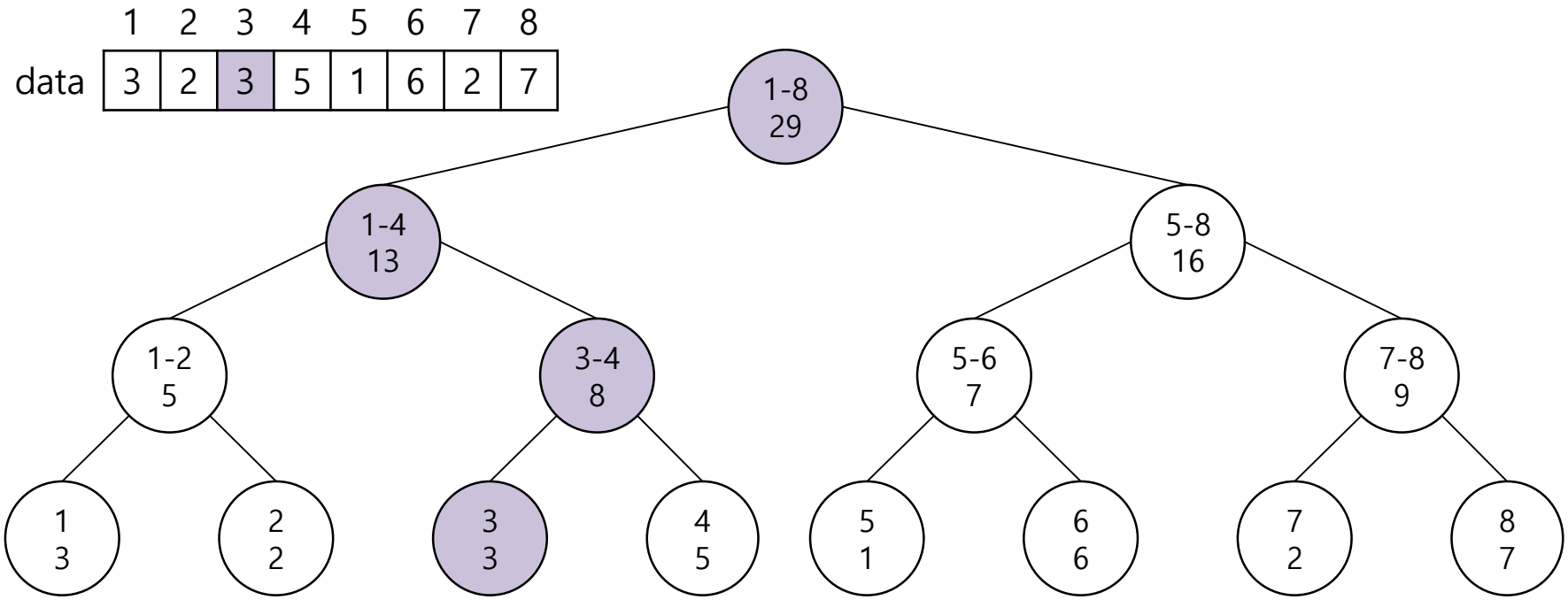
❖ 색칠된 노드를 찾기 위해 루트 노드에서 리프 노드까지 탐색하므로 수행 시간은 O(logN)



이진 트리 (Binary Tree)의 응용 – 인덱스 트리(Indexed Tree)

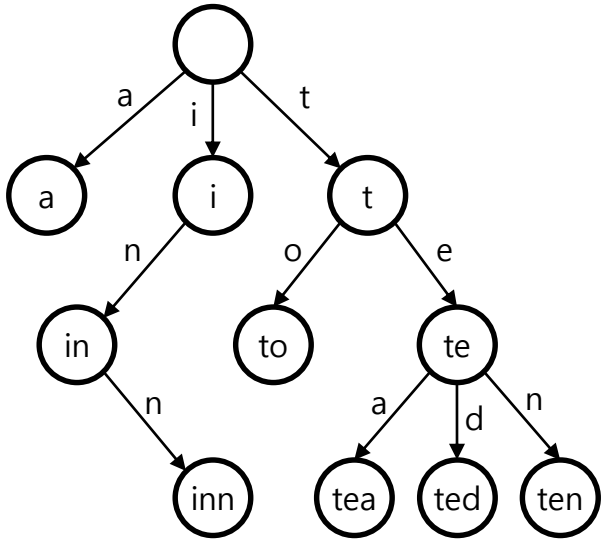
- 인덱스 트리(Indexed Tree) – 예시 – 데이터 갱신 방법
  1. 해당 노드 위치의 값을 수정한다.
  2. 부모를 따라 올라가면서 결과값을 수정한다.
  3. 3번 위치의 데이터가 3으로 바뀐다면 다음 색칠한 노드들이 수정된다.

❖ 리프 노드부터 루트 노드까지 올라가므로 수행 시간은 O(logN) 이다.



트리(Tree)의 응용 – 트라이(Trie)

- 트라이(Trie)
  - Prefix Tree, Digital Search Tree, Retrieval Tree
  - 문자열을 빠르게 검색할 수 있는 자료 구조
  - K진 트리 구조
  - 단어 사전을 트라이로 생성 후 찾을 단어를 트라이를 사용하여 검색
  - 트라이의 Root 노드는 항상 공백문자열(빈 문자열) 상태를 의미함



트리(Tree)의 응용 – 트라이(Trie)

- 트라이 구축하기
  - 트라이 노드 설계

```
class Node
    Object data
    Node child[ ]
```

  - 단어 사전의 입력할 단어를 트라이에 삽입
  - Root 노드부터 시작하여 단어의 첫 글자부터 트라이를 탐색
  - 만약 현재 노드의 자식 노드 중 현재 입력 중인 철자에 해당하는 자식이 있다면, 현재 노드를 해당하는 자식 노드로 이동
  - 만약 현재 노드의 자식 노드 중 현재 입력 중인 철자에 해당하는 자식이 없다면, 새로운 자식을 추가
  - 단어를 삽입 후, 탐색된 마지막 노드에 현재 입력된 단어의 정보를 추가

트리(Tree)의 응용 – 트라이(Trie)

- 트라이를 이용하여 검색하기
  - Root 노드부터 시작하여 검색할 단어의 첫 글자부터 트라이를 탐색
  - 만약 현재 노드의 자식 노드 중 현재 입력 중인 철자에 해당하는 자식이 있다면, 현재 노드를 해당하는 자식 노드로 이동
  - 만약 현재 노드의 자식 노드 중 현재 입력 중인 철자에 해당하는 자식이 없다면, 검색할 단어는 단어사전에 존재하지 않는 단어로 처리
  - 탐색이 완료되었다면, 탐색된 마지막 노드의 정보를 이용

해싱 (Hashing)

- 해싱 (Hashing)
  - 해싱이란 입력된 데이터(Key)를 해시 함수(Hash Function)를 통해 얻은 주소로부터 그 위치를 직접 참조하는 방법이다.
  - 탐색, 삽입, 삭제 연산 모두 해시 함수(Hash Function)를 거치는 시간만 소요되기 때문에 일반적으로 O(1)의 시간복잡도를 가진다.

- 해시 표(Hash Table)
  - 버킷(Bucket)과 슬롯(Slot)으로 구성된 이차원 배열
  - 버킷 개수 : 해시 함수를 통해 나오는 주소 값의 범위
  - 슬롯 개수 : 각 버킷에 저장할 수 있는 데이터(Key)의 개수

셋 (Set)

- 셋 (Set)
  - 집합을 정의하는 자료구조로 동일한 자료형을 모아놓은 것.
  - 집합의 모든 원소(Key)는 유일하다. 즉 중복이 허용되지 않는다.
  - 삽입, 삭제, 탐색의 세 가지 연산을 지원한다.
  - 구현 방식에 따라 해시 셋(Hash Set), 트리 셋(Tree Set)등이 존재한다.
- 해시 셋(Hash Set)
  - 해싱을 이용하여 데이터를 저장하는 방법.
  - 모든 연산이 O(1)에 수행되기 때문에 가장 빠르다.
  - Key값을 나열했을 때 순서를 예측할 수 없다.
- 트리 셋(Tree Set)
  - 일반적으로 균형 이진 검색 트리(Balanced Binary Search Tree) 중 레드 블랙 트리(Red-black Tree)로 구현되어 있다.
  - 모든 연산이 O(logN)에 수행된다.
  - Key값을 나열했을 때 정렬된 순서로 불러온다. 정렬 방식을 지정할 수 있다.

맵 (Map)

- 맵 (Map)
  - Key와 Value로 이루어진 객체를 저장하는 자료구조
  - Set과 같이 Key값의 중복을 허용하지 않으며, value값은 제약이 없다.
  - 삽입, 삭제, 탐색의 세 가지 연산을 지원한다.
  - 구현 방식에 따라 해시 맵(Hash Map), 트리 맵(Tree Map)등이 존재한다.
    각각의 특징은 Hash Set, Tree Set과 동일하다.