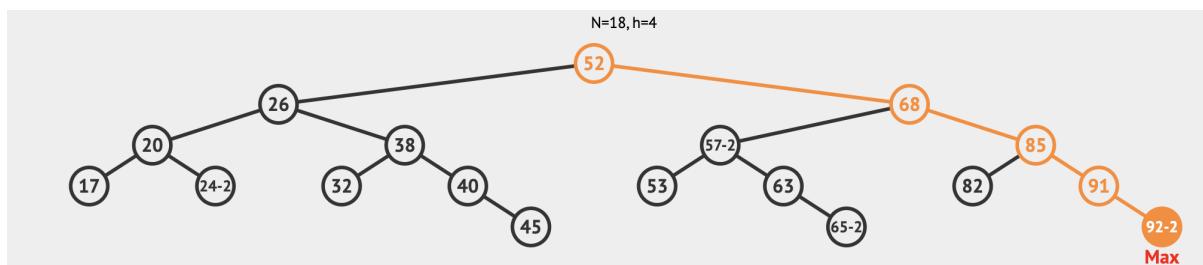


# 트리

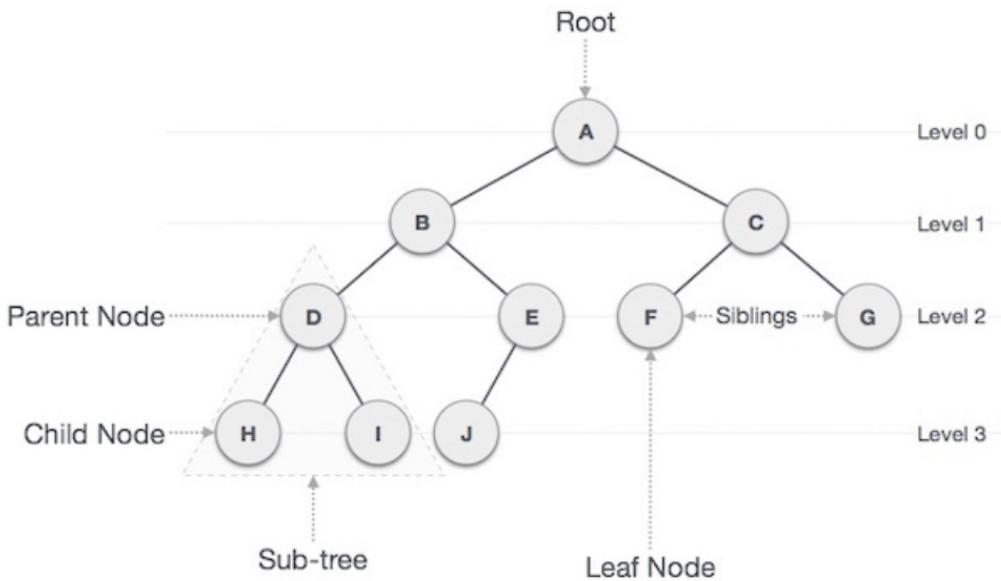
작성일시	@June 16, 2023 1:13 PM
자료	<a href="https://algodaily.com/challenges/binary-tree-inorder-traversal">https://algodaily.com/challenges/binary-tree-inorder-traversal</a> <a href="https://visualgo.net/en/bst">https://visualgo.net/en/bst</a> <a href="https://medium.com/swlh/binary-tree-level-order-traversal-a12df61a85d0">https://medium.com/swlh/binary-tree-level-order-traversal-a12df61a85d0</a>
#주차	4



트리(tree)는 **비순차적 자료 구조**이다. 트리는 정보를 쉽게 검색하기 위해 저장할 때 유용한 자료 구조이다.

트리는 **계층 구조(hierarchical structure)**를 추상화한 모델이다. 디렉토리나 조직도를 보면 사장님이 있고 각 부서장을 거쳐서 사원까지 나무 가지처럼 생겼다. 그런 정보를 컴퓨터 상의 데이터로 표현하는 방법을 트리 구조라고 한다. 주변에서 가장 흔한 예로 가계도가 있고, 회사 조직도 역시 트리 구조로 되어 있다.

## 트리 용어



트리는 부모-자식 관계를 가진 다수의 노드로 구성된다. 각 노드는 부모 노드를 가지며(최상위 노드를 제외하고), 다수의 자식 노드를 가질 수 있고 하나도 없을 수도 있다.

- 루트(root): 최상위 노드, 부모가 없는 노드. 예) A
- 트리의 원소를 노드라고 한다. 두 종류의 노드가 있다.
  - 내부노드: 1개 이상의 자식노드를 가진 노드
  - 외부노트(리프노드): 자식이 하나도 없는 노드. 예) F
- 노드는 조상(ancestor)과 후손(descendant)을 가질 수 있다.  
조상은(루트를 제외하고) 상위계층의 노드를, 후손은 하위 계층의 노드를 가리킨다.
  - 서브트리는 노드와 후손으로 구성된다. 예) 노드 D,H,I가 하나의 서브트리라고 할 수 있다.
- 노드의 깊이(depth): 조상의 개수  
위의 예시 그림의 깊이는 3이다.
- 트리의 높이(height): 깊이의 최대치. 트리는 레벨(level)로 구분하기도 한다. 루트는 0레벨, 그 아래 자식은 레벨 1. 위의 그림의 트리 높이는 3이다

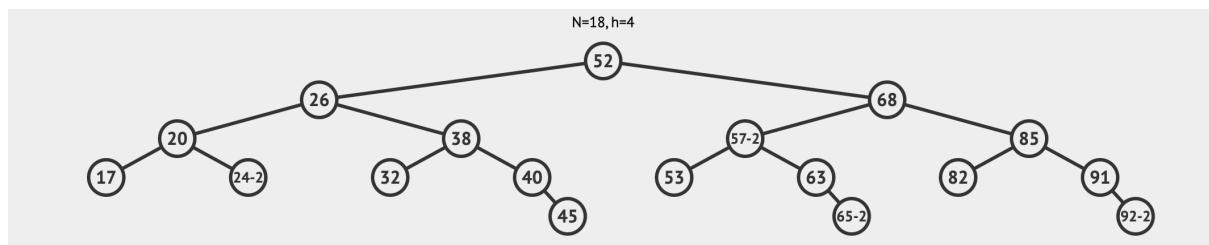
# 이진 트리와 이진 탐색 트리

## 이진 트리(binary tree)

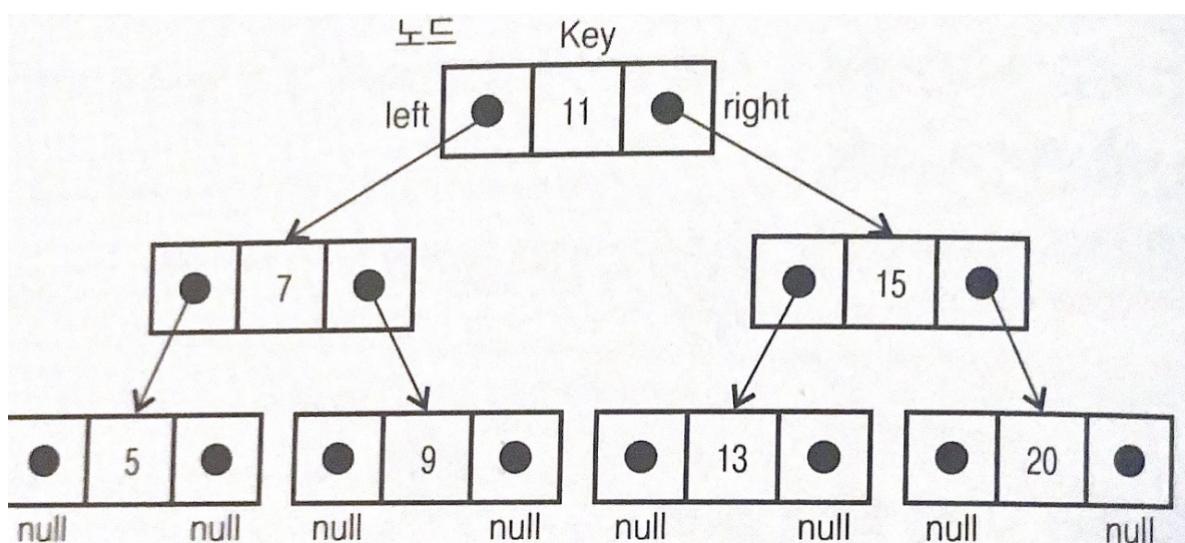
노드는 좌, 우측에 각각 하나씩, 최대 2개의 자식 노드를 갖는다.

## 이진 탐색 트리(Binary Search tree)

이진 트리의 변형으로, 좌측 자식 노드에는 더 작은 값을, 우측 자식 노드에는 더 큰 값을 들고 있다는 차이점이 있다.



## 이진 탐색 트리 구현(linked list)



```
// 뼈대
function BinaraySearchTree() {
  const Node = function(key) {
```

```

        this.key = key; // 노드 식별
        this.left = null; // 좌측 포인터
        this.right = null; // 우측 포인터
    };

    let root = null; // 자료구조의 첫 번째 노드
}

```

연결 리스트와 마찬가지로 노드(트리에서는 간선(edge)라고 표현) 간 연결은 포인터로 나타낸다. 트리는 2개의 포인터가 있는데 각각 좌측, 우측 자식 노드를 가리킨다.

트리에서는 키로 노드를 식별한다.

연결리스트에서 사용했던 패턴처럼 자료 구조의 첫 번째 노드를 변수로 선언해서 제어한다. 단, 트리에서는 헤드가 아닌 루트.

### 트리구현 시 필요한 메서드

- insert(키): 새로운 키 삽입
- search(키): 해당 키를 가진 노드가 존재하는지 여부를 true / false로 반환
- inOrderTraverse: 중위 순회(in-order traverse) 방식으로 전체 노드 방문
- preOrderTraverse: 전위 순회(pre-order traverse) 방식으로 트리의 전체 노드 방문
- postOrderTraverse: 후위 순회(post-order traverse) 방식으로 트리의 전체 노드 방문
- min: 트리의 최소 값/키를 반환
- max: 트리의 최대 값/키를 반환
- remove(키): 키를 삭제

### 트리에 키 삽입하기

```

this.insert = function(key) {

    const newNode = new Node(key);

    if(root === null){
        root = newNode;
    } else {
        insertNode(root, newNode);
    }
}

```

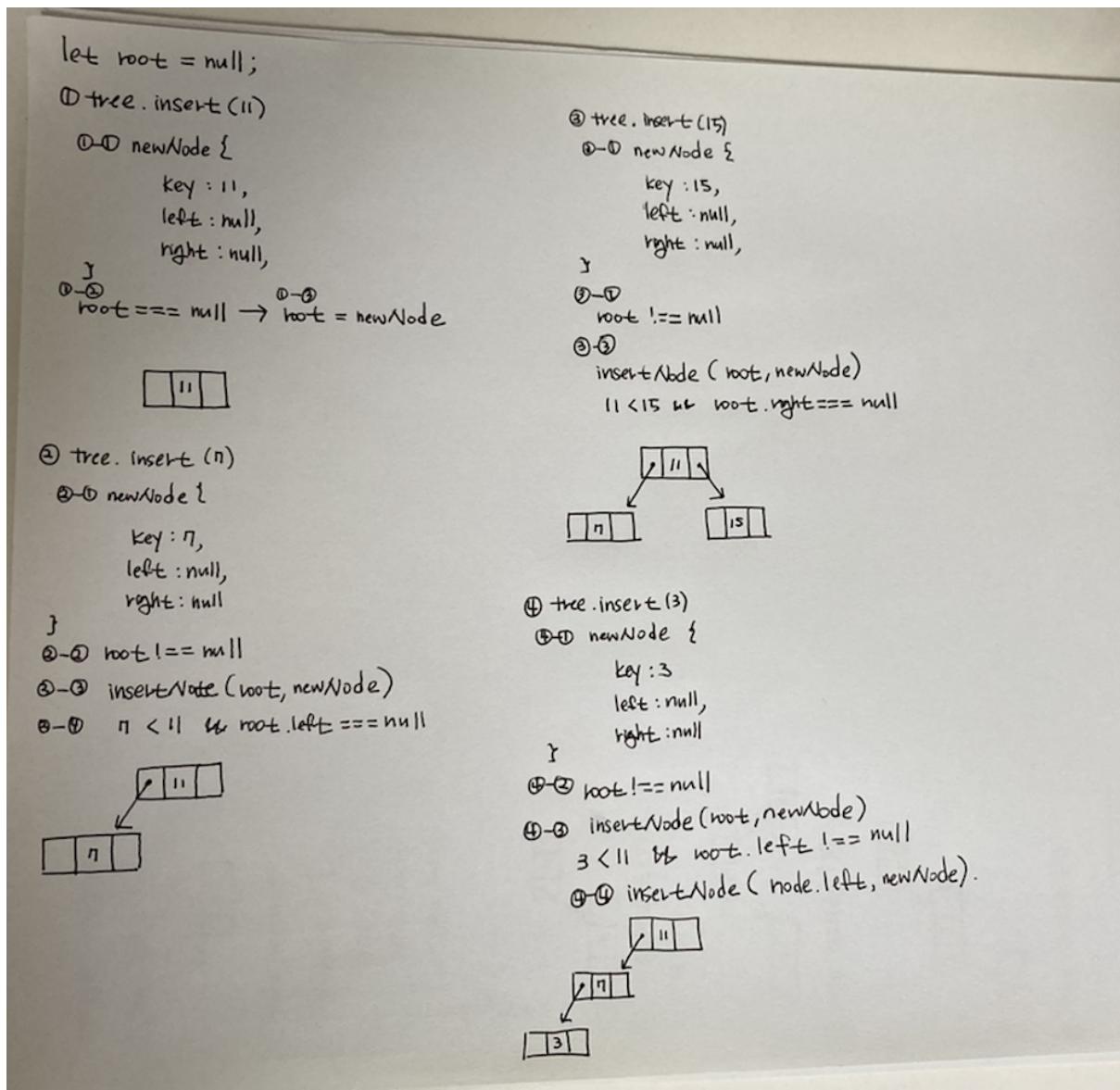
트리에 새 노드를 삽입할 때는 다음 세 단계 작업을 거친다.

1. 새 노드에 해당하는 Node 인스턴스를 생성한다. 생성자에는 트리에 주차할 값을 인자로 넘겨주며, left/right 포인터는 null로 초기화된다.
2. 추가할 노드가 트리의 최소 노드일 경우. 즉, 트리가 비어있을 때는 이 노드를 루트 노드로 세팅한다.
3. 루트를 제외한 다른 위치에 추가하는 일반적인 경우, 다음의 프라이빗 헬퍼 함수 insertNode를 호출한다.

```
const insertNode = (node, newNode) => {
  if(newNode.key < node.key) { // (4)
    if(node.left == null) { // (5)
      node.left = newNode; // (6)
    } else{
      insertNode(node.left, newNode); // (7)
    }
  } else {
    if(node.right == null) { // (8)
      node.right = newNode; // (9)
    } else{
      insertNode(node.right, newNode); // (10)
    }
  }
}
```

insertNode 함수의 기능은 새 노드를 추가할 위치를 정확히 찾는 것이다.

4. 새 노드의 키가 현재 노드의 키보다 작으면, 노드의 좌측 자식 노드를 확인해본다
5. 만약 좌측의 자식 노드가 없다면
6. 새 노드를 이 자리에 넣는다.
7. 그렇지 않으면 insertNode함수를 다시 재귀 호출해서 하위레벨로 다시 내려간다.
8. 새 노드의 키가 현재 노드의 키보다 크고 우측 자식 노드가 없다면
9. 새노드를 이 자리에 넣는다
10. 그렇지 않으면 insertNode함수를 다시 재귀 호출해서 하위레벨로 내려간다.



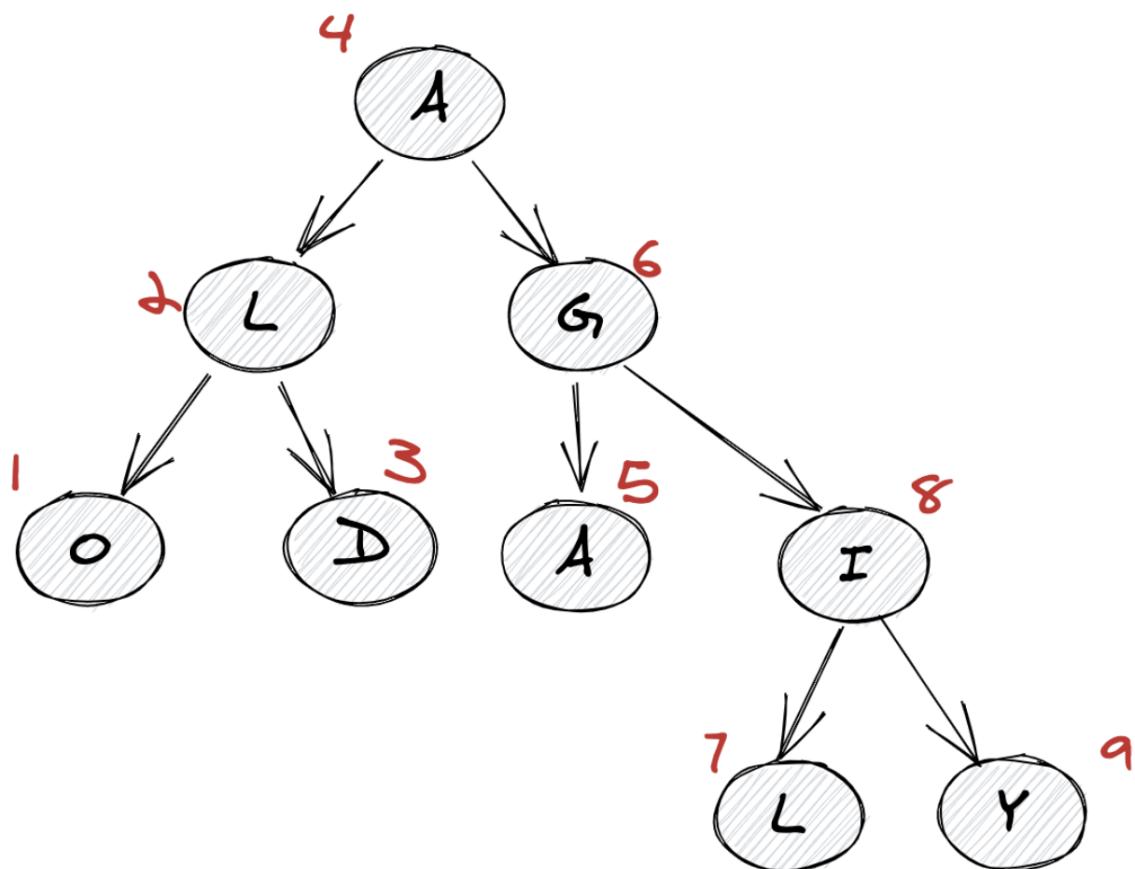
## 트리 순회

트리의 모든 노드를 방문해서 각 노드마다 어떤 작업을 수행하는 것을 트리 순회(traversal)라고 한다.

트리를 순회하는 방법 기준에 따라 세 가지로 분류된다.

### 중위 순회(in-order traversal)

## IN-ORDER



BST의 노드를 오름차순, 즉 작은 값에서 큰 값 방향으로 순회한다. 트리 정렬시 사용되는 방법이다.

```
this.inOrderTraverse = function(callback) {  
    inOrderTraverseNode(root, callback);  
}
```

inOrderTraverse 메서드가 인자로 취하는 콜백 함수에는 노드 방문시 수행할 작업을 기술 한다.(이런 식의 프로그래밍을 방문자 패턴visitor pattern이라한다. )

```
const inOrderTraverseNode = function(node, callback) {  
    if(node !== null) { // (2)  
        inOrderTraverseNode(node.left, callback); // (3)  
        callback(node.key); // (4)  
        inOrderTraverseNode(node.right, callback); // (5)
```

```
    }  
}
```

2. 먼저 인자 node 가 null인지 확인(재귀호출이 중단되는 시점을 알고리즘에서는 기본 상태 base case라고 한다)
3. 자기 자신을 재귀 호출해 좌측 자식 노드 방문
4. 방문한 노드에 어떤 작업 수행
5. 그리고 우측 자식노드 방문

```
function printNode(value) {  
  console.log(value);  
}  
  
tree.inOrderTraverse(printNode); // 오름차순으로 출력
```

## 전위 순회

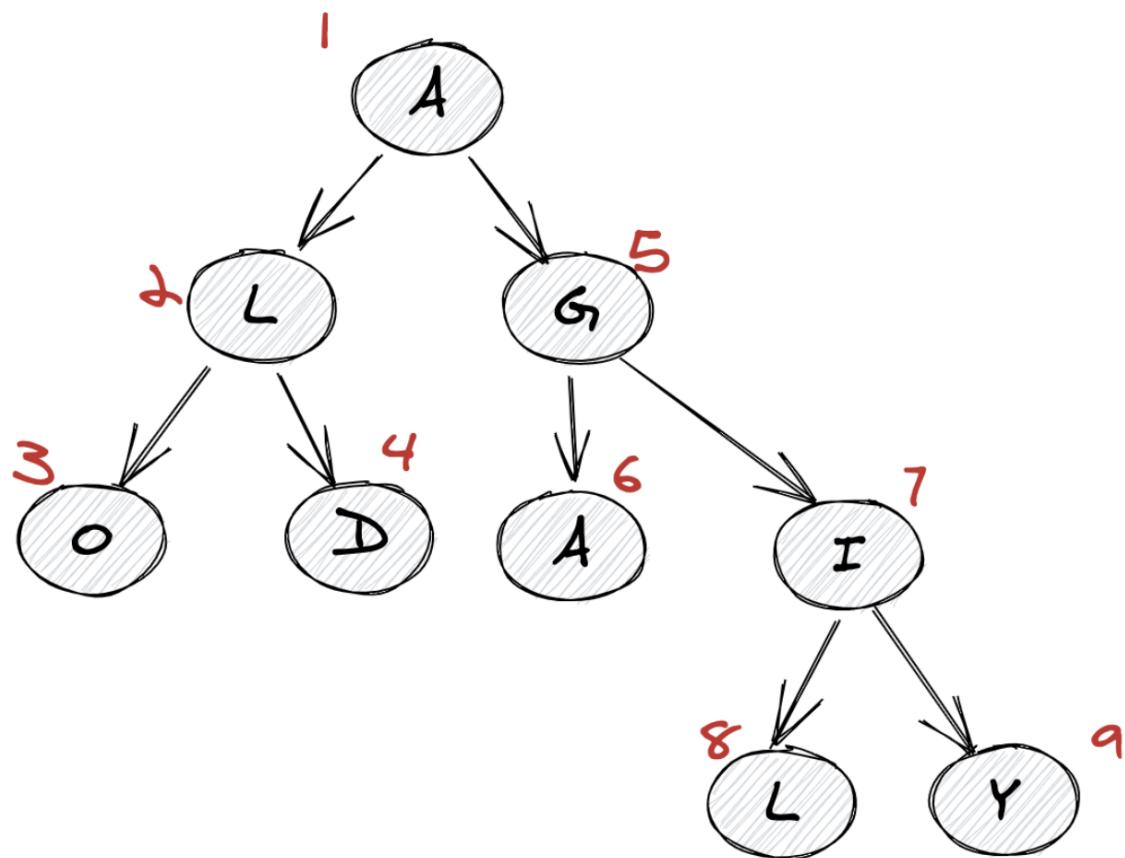
전위 순회(pre-order traverse) 에서는 자식 노드 보다 노드 자신을 먼저 방문한다.

구조화된 문서를 출력할 때 많이 이용하는 방법

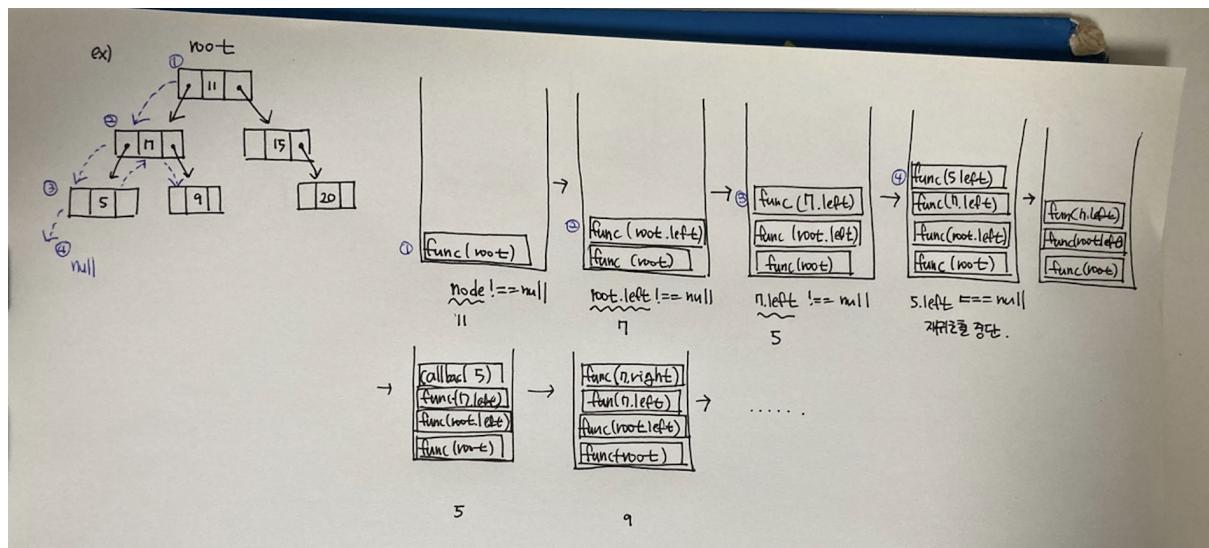
```
this.preOrderTraverse = function(callback) {  
  preOrderTraverseNode(root, callback);  
}  
  
const preOrderTraverseNode = function(node, callback) {  
  if(node !== null) {  
    callback(node.key); // (1)  
    preOrderTraverseNode(node.left, callback); // (2)  
    preOrderTraverseNode(node.right, callback); // (3)  
  }  
}
```

1. 전위 순회는 일단 노드를 먼저 방문후
2. 좌측 노드와
3. 우측 노드를 방문한다.

# PRE-ORDER



This algorithm is equivalent to the famous graph algorithm [Depth First Search \(DFS\)](#).



후위 순회

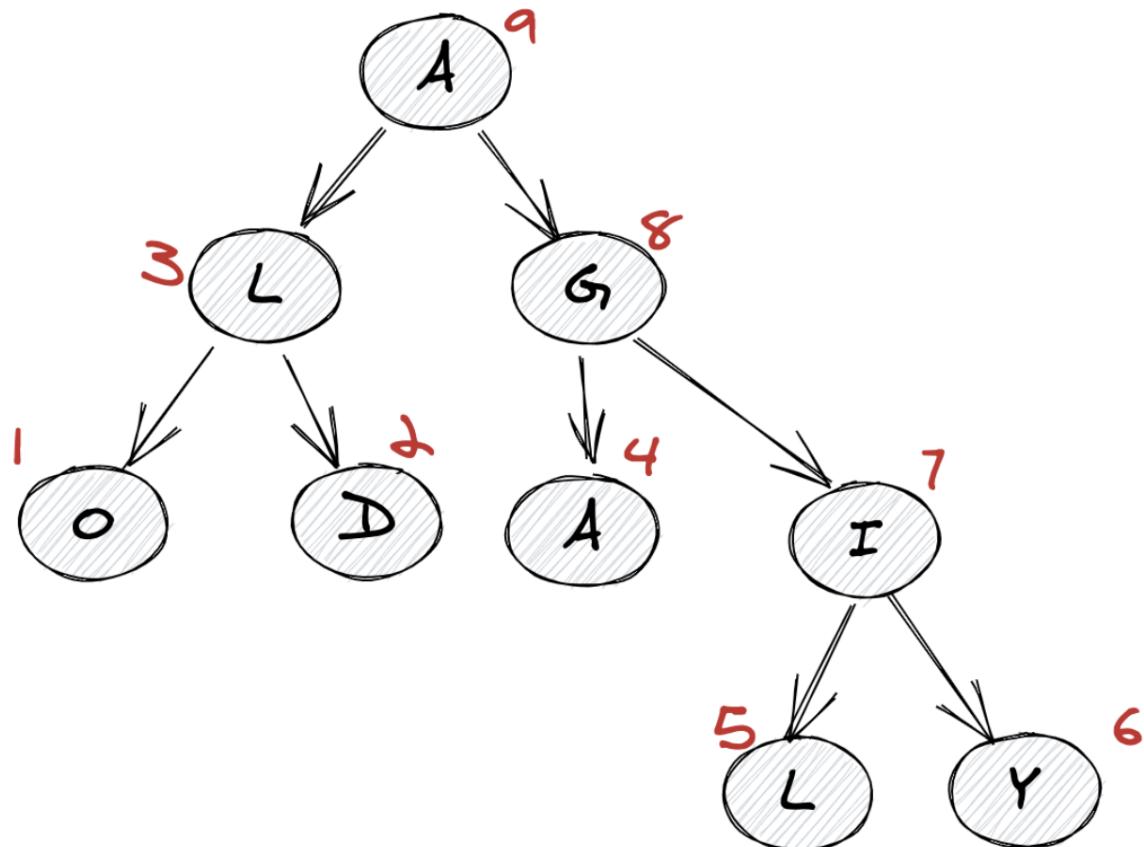
후위 순회(post-order traversal)는 자식노드를 노드 자신보다 먼저 방문한다.

디렉토리와 서브 디렉토리의 파일용량을 계산할 때 쓰는 방법이다.

```
this.postOrderTraverse = function(callback) {  
    postOrderTraverseNode(root, callback);  
}  
  
const postOrderTraverseNode = function(node, callback) {  
    if(node !== null) {  
        postOrderTraverseNode(node.left, callback); // (1)  
        postOrderTraverseNode(node.right, callback); // (2)  
        callback(node.key); // (3)  
    }  
}
```

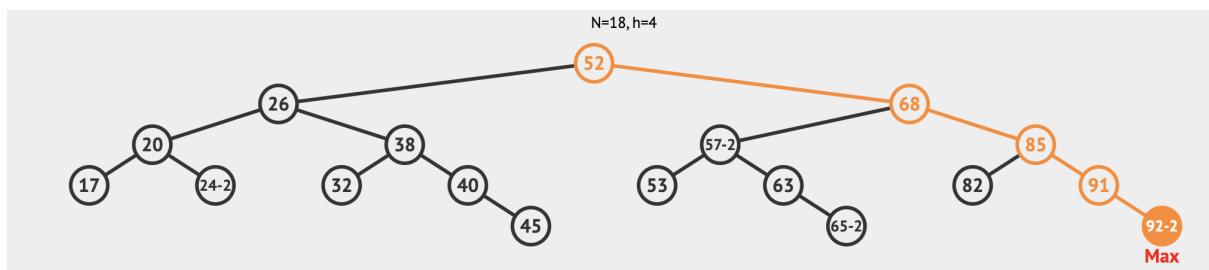
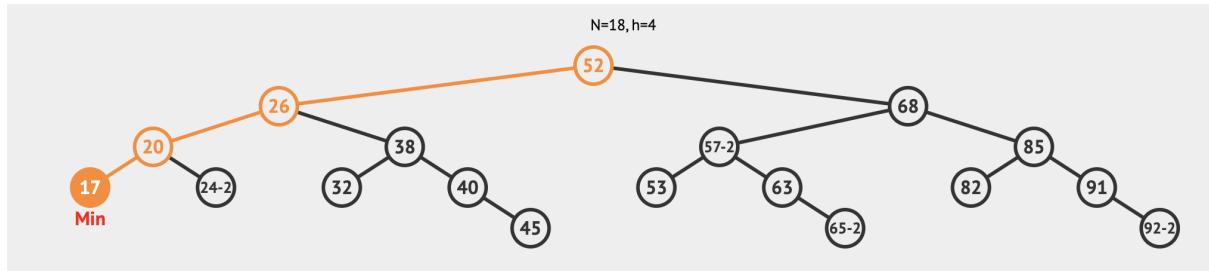
1. 좌측노드를 먼저 방문하고
2. 그 다음 우측 노드
3. 마지막으로 노드 자신을 방문

## POST-ORDER



# 트리 노드 검색

## 최솟값/ 최대값 찾기



```
// min
this.min = function() {
    return minNode(root);
}

const minNode = function(node) {
    if(node) {
        // 트리의 마지막 레벨에 위치한 노드에 도달할 때까지
        // 트리의 left 간선을 따라 순회
        while(node && node.left !== null) {
            node = node.left;
        }
        return node.key;
    }
    return null;
}
```

```
// max
this.max = function() {
    return maxNode(root);
}

const maxNode = function(node) {
```

```

if(node) {
    // 트리의 마지막 레벨에 위치한 노드에 도달할 때까지
    // 트리의 right 간선을 따라 순회
    while(node && node.right !== null) {
        node = node.right;
    }
    return node.key;
}
return null;
}

```

## 특정 값 찾기

```

this.search = function(key) {
    return searchNode(root, key); // (1)
}

const searchNode = function(node, key) {
    if(node == null) { // (2)
        return false;
    }
    if(key < node.key) { // (3)
        return searchNode(node.left, key); // (4)
    } else if(key > node.key) { // (5)
        return searchNode(node.right, key); // (6)
    } else {
        return true; // (7)
    }
}

```

1. search 메서드를 먼저 선언하고 내부적으로는 다른 BST의 메서드들처럼 헬퍼 함수를 호출
2. node인자가 null 이면 유효하지 않은 값이므로 false 반환
3. node가 null 아니면 이후로 또 다른 분기가 이루어진다. 찾는 키가 현재 노드의 키보다 작다면
4. 좌측 자식 노드쪽으로 서브트리를 따라서 검색을 계속하고
5. 반대로 더 트다면
6. 우측 자식 노드 쪽 서브트리를 따라서 검색을 계속한다.
7. 두 경우 모두 아니라면, 즉 현재 노드의 키가 바로 찾고자하는 키라면 검색이 완료된 것 이므로 true를 반환한다.

## 노드 삭제

```
this.remove = function(key) {  
    root = removeNode(root, key); // (1)  
}
```

- 삭제할 노드의 키를 인자로 받고 다시 root 와 key를 인자로 removeNode 메서드를 호출한다. 여기서 **removeNode**의 반환값을 root로 다시 셋팅하는 것이 중요하다.

```
const removeNode = function(node, key) {  
    if(node === null) { // (2)  
        return null;  
    }  
    if(key < node.key) { // (3)  
        node.left = removeNode(node.left, key); // (4)  
        return node; // (5)  
    } else if(key > node.key) { // (6)  
        node.right = removeNode(node.right, key); // (7)  
        return node; // (8)  
    } else {  
        // case1: 리프노드  
        if(node.left === null && node.right === null) { // (9)  
            node = null; // (10)  
            return node; // (11)  
        }  
        // case2: 자식이 하나뿐인 노드  
        if(node.left === null) { // (12)  
            node = node.right; // (13)  
            return node; // (14)  
        } else if(node.right === null) { // (15)  
            node = node.left; // (16)  
            return node; // (17)  
        }  
        // case3: 자식이 둘인 노드  
        const aux = findMinode(node.right); // (18)  
        node.key = aux.key; // (19)  
        node.right = removeNode(node.right, aux.key); // (20)  
        return node; // (21)  
    }  
}
```

- 노드가 null 이면 이 트리에 해당 키는 없다는 뜻이므로 null 반환

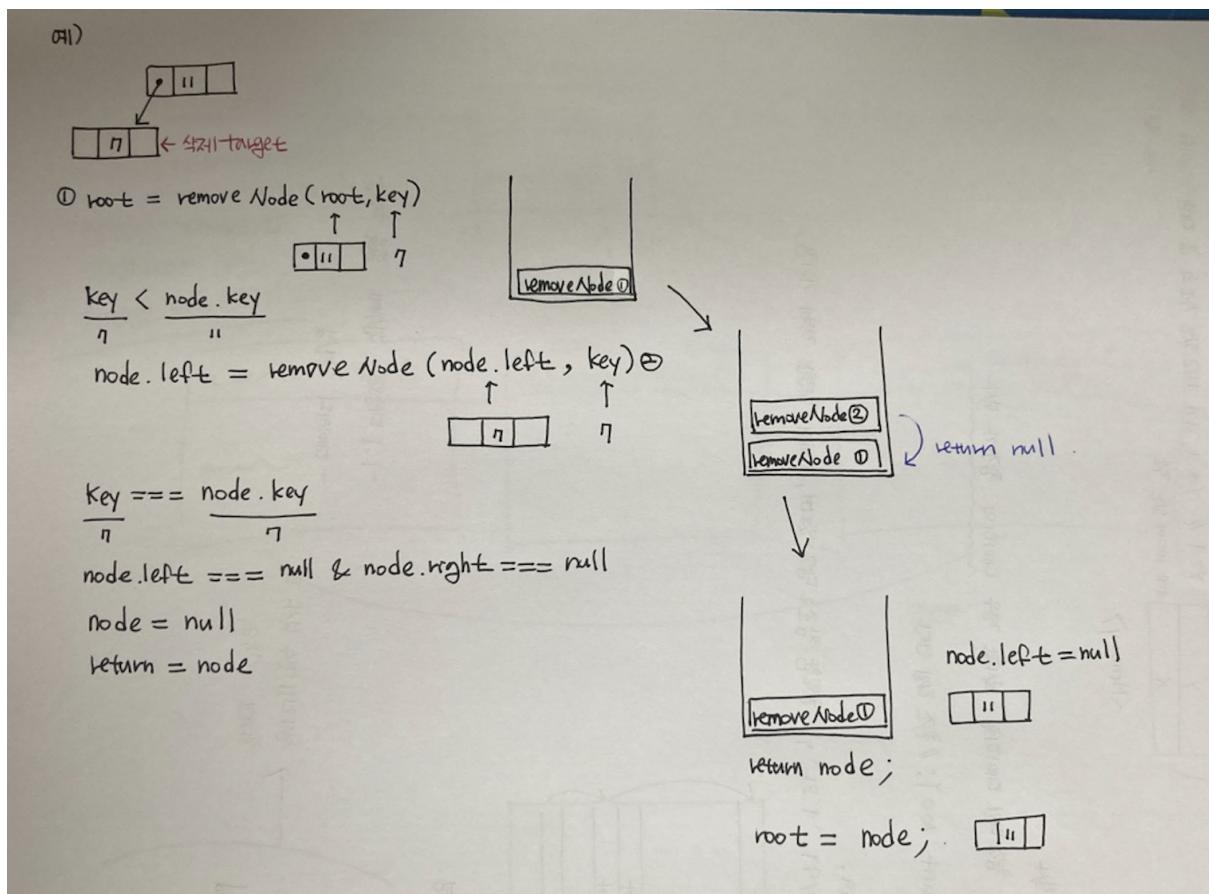
null 이 아니라면 다음 코드로 넘어가 트리에서 노드를 찾아야 한다.

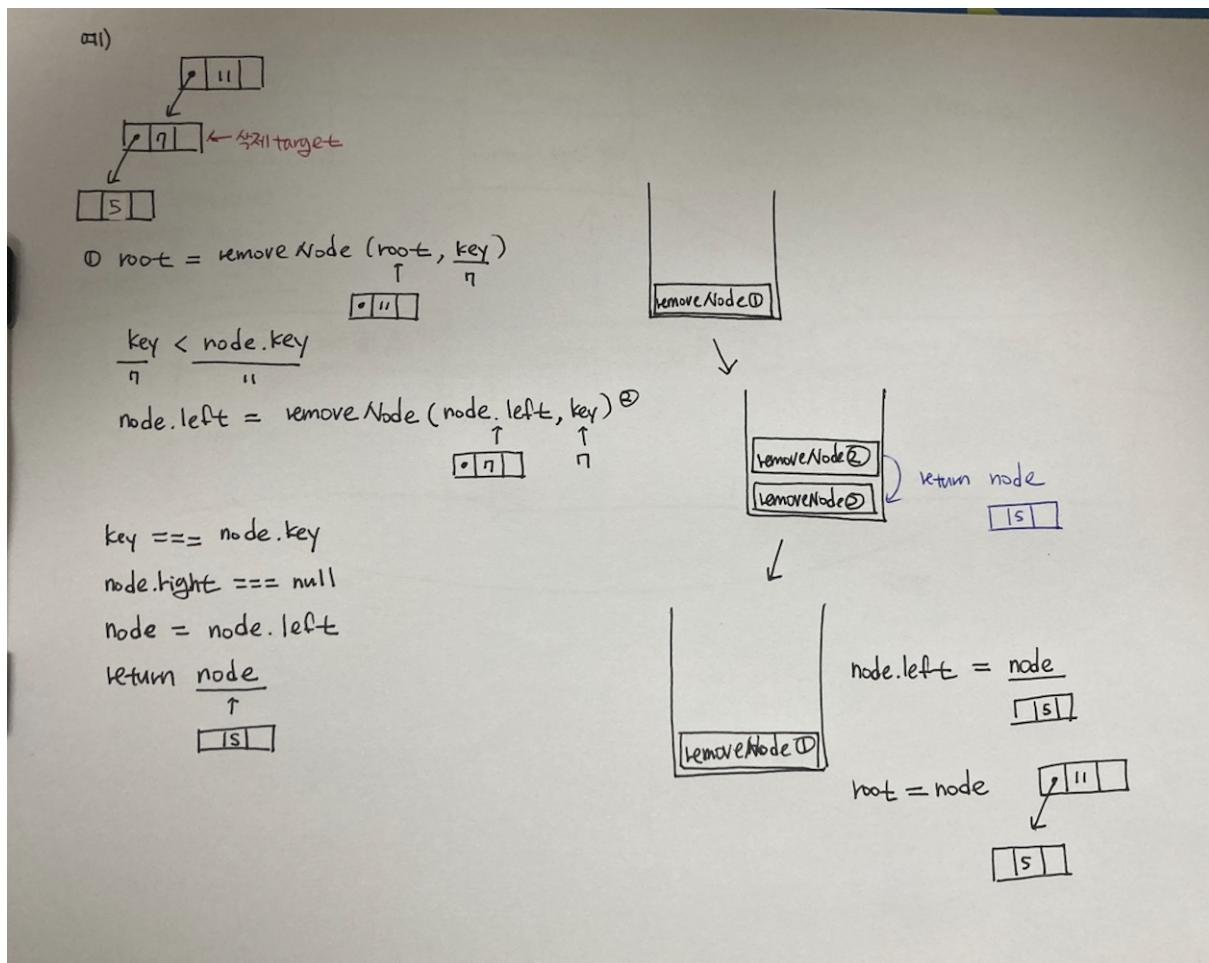
3. 찾는 키가 현재 노드의 키보다 작으면

4. 트리의 좌측 간선을 따라 이동하고

6. 그 반대라면

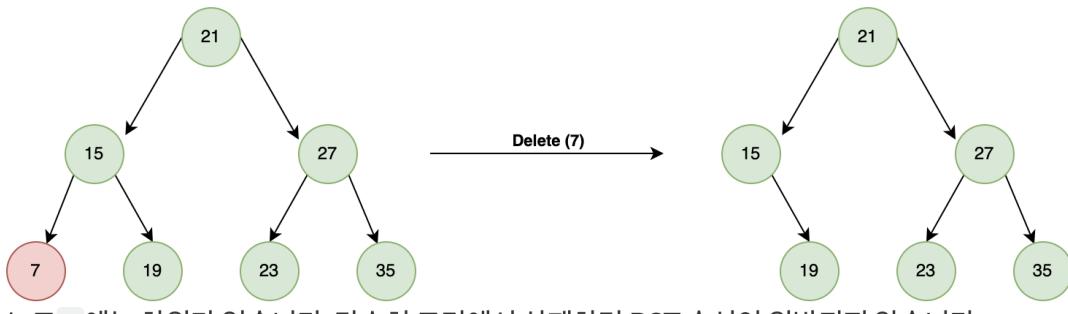
7. 트리의 우측 간선을 따라 이동한다.





## 리프노드인 경우

01. 삭제할 노드에 자식이 없습니다-리프입니다.

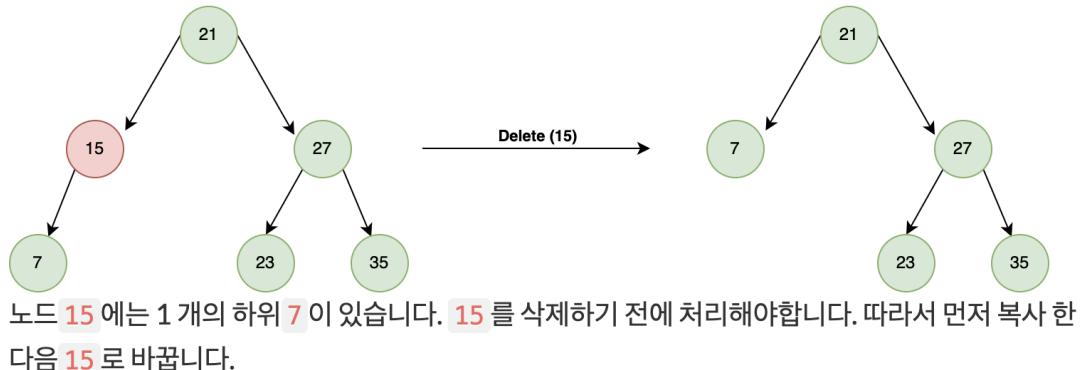


<https://www.delftstack.com/ko/tutorial/data-structure/binary-search-tree-delete/>

포인터를 신경써서 처리해야 한다. 리프 노드는 자식 노드는 없지만 부모 노드는 있으므로, 이 노드를 가리키는 부모 노드의 포인터 역시 null 이 되어야 한다. 바로 이런 이유로 노드의 값을 함수에서 반환하는 것이며, 부모 노드는 이 반환값을 건네받는다.

## 좌/우측 어느 한 쪽에만 자식노드가 있는 경우

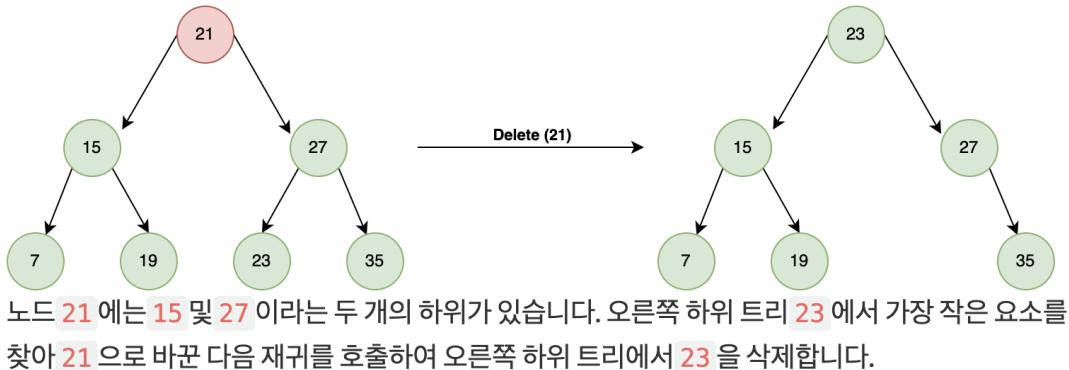
02. 삭제할 노드에는 자식이 하나만 있습니다.



이런 경우 해당노드를 건너뛰어 할아버지가 손주의 손을 직접 잡게하면 된다.

## 두 자식을 모두 가진 노드일 경우

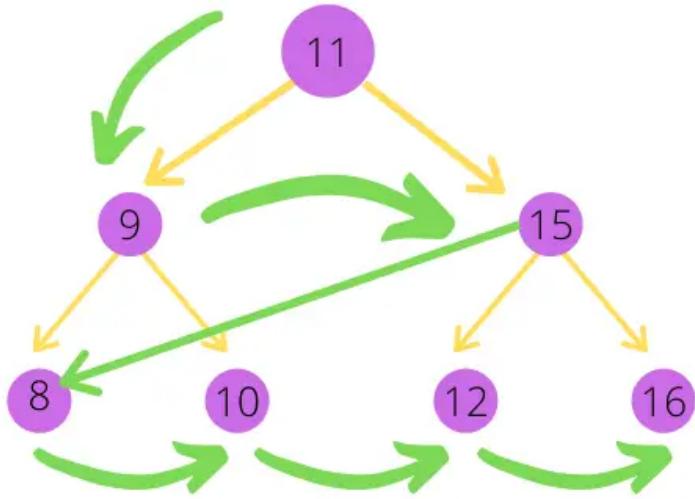
03. 삭제할 노드에는 두 하위 항목이 있습니다.



1. 우측 서브트리로부터 최소 노드를 찾는다. (18)
2. 이렇게 찾은 최소 노드의 값으로 수정(19) 삭제할 노드의 키 자체가 교체되므로 결국 삭제되는 것이다.
3. 하지만 그결과 한 트리에 동일한 키를 가진 노드가 중복되어 생기는 꼴이므로 있을 수 없는 일이다. 그래서 우측 서브 트리의 최소 노드는 삭제해야 하는데, 이 노드를 2에서 삭제한 노드 위치로 옮기면 된다.(20)
4. 마지막으로 수정된 노드를 반환한다.(21)

# 이진트리 Level Order Traversal

learnersbucket.com



Level order traversal of binary tree

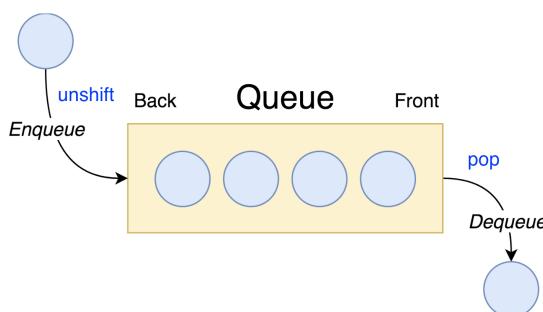
<https://learnersbucket.com/examples/algorithms/tree-traversal-in-javascript/>

다음 레벨로 내려가기 전 노드를 왼쪽에서 오른쪽으로 순회하는 것은 **breadth-first traversal** 또는 **level order tree traversal** 이다.

## Breath-First-Search(BFS) Algorithm:

다음은 queue를 활용한 BRS 알고리즘이다.

\* queue는 FIFO(First in First out)으로 동작한다.



<https://www.newline.co/books/javascript-algorithms/queue>

1. 배열로 만들어진 새로운 큐를 만들고 큐에 root노드를 삽입한다.
2. 큐가 노드 요소를 가지고 있는한 while loop를 돌린다.
3. 먼저, 큐에서 가장 위에있는 첫번째 원소를 제거한다. 이것은 현재 노드를 의미한다.
4. 이 경우, 현재 노드의 값을 콘솔에 출력한다.
5. 마지막으로 노드에 왼쪽 또는 오른쪽 자식이 있는지 확인하고 이것들을 큐에 삽입한다.

```
/* Level Order Traversal: Given a binary tree, print its nodes in level order.*/
function levelOrder(root) {
    let queue = [root]
    while (queue.length != 0) {
        let node = queue.pop()
        console.log(node.val)
        if (node.left) queue.unshift(node.left)
        if (node.right) queue.unshift(node.right)
    }
}
```

## 배열을 이용한 구현

