4주차: 트리 재귀·BFS·BST

계층 기록의 깊이와 검색 순서

하위 작업이 최대 두 개씩 연결된 작업 트리에서 가장 깊은 단계는 어디일까요? 단계별 현황을 출력하고, 검색용 트리가 정렬 규칙을 지키는지도 확인합니다.

한 줄로 저장한 기록과 달리 각 노드에서 두 갈래를 따라가야 합니다. 재귀 반환값, 스택·큐, 조상의 범위로 필요한 정보를 전달합니다.

학습 목표

7차시: 재귀 반환값과 최대 깊이. 8차시: BFS·BST 검증·중위 순회.

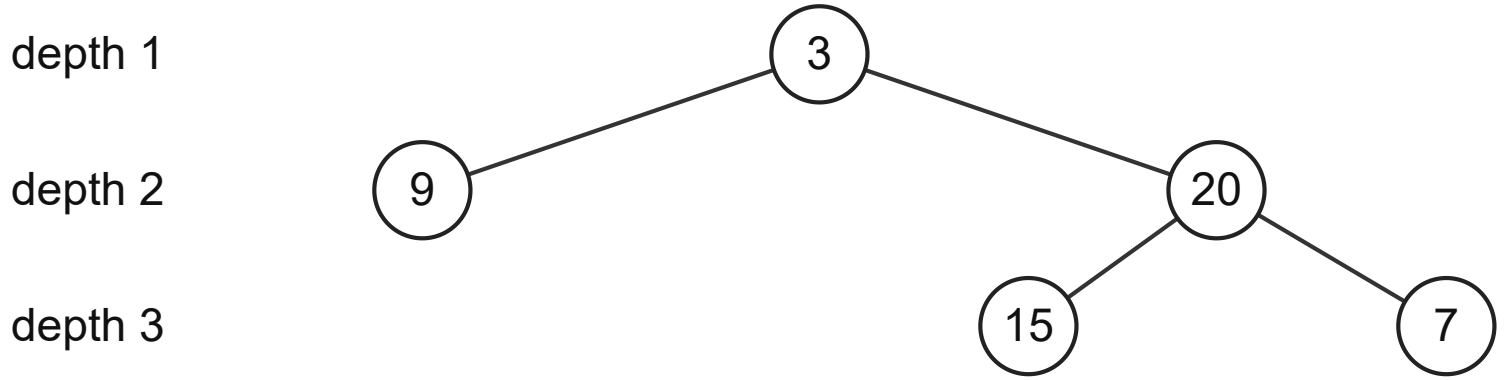
- 빈 트리와 자식 호출의 반환값 정의
- 한 레벨의 큐 경계와 조상 범위 추적
- 시간·보조 공간·출력 공간 구분
- 작은 예제로 재귀를 이해하고 반복 구현으로 긴 트리 처리

실습: LeetCode 104, 102, 98, 230.

트리의 높이·너비와 저장 공간

각 노드는 값과 두 자식 참조를 가집니다. 없는 자식은 None입니다. 깊이는 루트에서 현재 노드까지의 노드 수이며, 높이 h는 최대 깊이, 너비 w는 한 층의 최대 노드 수입니다.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val, self.left, self.right = val, left, right
```



이 트리는 n=5, h=3, w=2입니다. DFS는 아직 끝내지 않은 경로를 스택으로, BFS는 다음에 처리할 층을 큐로 보관합니다. 따라서 같은 노드 수라도 트리 모양에 따라 최대 저장량이 달라집니다.

리스트 표기에서 실제 트리로

`[4,2,7,None,3]`은 레벨 순서 표기입니다. 실제 노드의 자식을 왼쪽·오른쪽 순서로 읽으며 None은 그 자리가 비었다는 뜻입니다. None 자체의 자식은 읽지 않습니다.

읽은 값	연결	다음에 자식을 읽을 노드
4	루트 생성	[4]
2,7	4.left=2, 4.right=7	[2,7]
None,3	2.left=None, 2.right=3	[7,3]

남은 자식 표기는 생략된 None입니다. 따라서 경로 4→2→3의 깊이는 3이고, 레벨은 `[[4],[2,7],[3]]`입니다. 최대 깊이 함수는 이 목록을 직접 받는 것이 아니라 연결된 TreeNode의 root를 받습니다.

임의의 희소 트리에서 목록 위치 i의 자식이 항상 2i+1,2i+2인 것은 아닙니다. 예를 들어 `[1,None,2,3]`의 3은 노드 2의 왼쪽 자식입니다.

재귀 호출이 돌려주는 깊이

```
def depth_recursive(root):
    if root is None:
        return 0
    return 1 + max(depth_recursive(root.left),
                   depth_recursive(root.right))
```

현재 노드	왼쪽 반환값	오른쪽 반환값	현재 호출의 반환값
잎 9,15,7 각각	0	0	1
20	1 (15에서)	1 (7에서)	2
3	1 (9에서)	2 (20에서)	3

각 호출의 root는 서로 다른 지역 변수입니다. 부모 호출은 자식의 결과를 기다리며 남아 있습니다. 따라서 방문 작업은 노드당 한 번으로 O(n)이지만 동시에 기다리는 호출은 높이 h개여서 보조 공간 O(h)입니다.

Python의 재귀 깊이는 제한되어 있습니다. 다음 구현은 같은 깊이 정보를 명시적 스택에 넣어 긴 사슬도 처리합니다.

104 · 풀이의 출발점과 실행 추적

LeetCode 104: Maximum Depth of Binary Tree · Easy

root에서 잎까지 가장 긴 경로의 노드 수를 반환합니다.

깊이는 루트에서 현재 노드까지 지나온 노드 수입니다. 노드만 저장하면 깊이를 다시 계산해야 하므로 `(node, depth)`를 함께 스택에 넣습니다. 자식은 부모보다 한 단계 깊고, 가장 큰 depth가 답입니다.

트리: 루트 3, 왼쪽 9, 오른쪽 20, 20의 자식 15·7. 스택 오른쪽이 top입니다.

pop한 (값,깊이)	자식 삽입 후 stack	best
(3,1)	[(9,2),(20,2)]	1
(20,2)	[(9,2),(15,3),(7,3)]	2
(7,3)	[(9,2),(15,3)]	3
(15,3)	[(9,2)]	3
(9,2)	[]	3

104 · 아이디어를 코드로 옮기기

```
def max_depth(root):
    if root is None:
        return 0
    stack, best = [(root, 1)], 0
    while stack:
        node, depth = stack.pop()
        best = max(best, depth)
        if node.left:
            stack.append((node.left, depth + 1))
        if node.right:
            stack.append((node.right, depth + 1))
    return best
```

루트 깊이를 1로 넣고, 꺼낸 깊이로 best를 갱신한 뒤 자식 깊이 `depth+1`을 저장합니다. 왼쪽을 먼저 넣고 오른쪽을 나중에 넣으므로 실제 방문은 오른쪽부터입니다. 최댓값은 방문 순서와 무관합니다.

빈 트리는 깊이 0입니다. 재귀식 `1 + max(왼쪽 깊이, 오른쪽 깊이)`도 같은 의미지만 긴 사슬에는 명시적 스택을 사용합니다.

104 · 정당성·시간·메모리의 근거

재귀는 자식 높이가 맞다고 가정하면 현재 높이를 정확히 결합합니다. 명시적 스택은 각 노드의 실제 root 깊이를 전달하며 최대를 기록하므로 같은 답입니다.

트리에서는 각 노드가 부모 하나에서 한 번만 스택에 들어갑니다. n번의 push·pop과 노드당 두 자식 검사이므로 총 시간은 O(n)입니다. 루프 안의 if 두 개는 n에 따라 늘어나지 않습니다.

깊이 h인 이진 트리에서 스택에는 현재 경로의 각 층에 남긴 형제 가지가 최대 하나씩 쌓여 O(h) 공간입니다. 최악에는 h=O(n)입니다. 이 반복 구현에서 자식이 하나뿐인 사슬은 스택 원소가 한 개뿐이지만 재귀 호출은 h개 쌓입니다.

102 · 풀이의 출발점과 실행 추적

LeetCode 102: Binary Tree Level Order Traversal · Medium

트리 값을 깊이별 왼쪽부터 묶어 반환합니다.

같은 깊이끼리 묶으려면 먼저 들어온 노드를 먼저 꺼내는 큐를 사용합니다. 한 층을 시작할 때 큐 길이를 고정하면, 처리 중 추가되는 자식은 다음 층으로 남습니다.

앞의 3–9–20–15–7 트리를 사용합니다. 큐 왼쪽이 다음에 꺼낼 위치입니다.

층 시작 queue	고정한 길이	이번 level	다음 queue
[3]	1	[3]	[9,20]
[9,20]	2	[9,20]	[15,7]
[15,7]	2	[15,7]	[]

결과는 `[[3],[9,20],[15,7]]`입니다.

두 번째 층에서 처음 queue=[9,20]이면 반복 횟수는 2로 정해집니다. 9를 꺼내 [20], 20을 꺼내 자식들을 넣으면 [15,7]입니다. 아직 큐가 비지 않아도 두 번을 마쳤으므로 level=[9,20]을 저장하고 새 층을 시작합니다.

102 · 아이디어를 코드로 옮기기

```
from collections import deque

def level_order(root):
    if root is None:
        return []
    queue, result = deque([root]), []
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

`range(len(queue))`는 for를 시작할 때 횟수를 결정합니다. 자식을 append해도 이 횟수는 늘지 않습니다. `level`은 층마다 새 리스트로 만들고 층 처리가 끝나면 result에 넣습니다.

`deque.popleft()`로 앞을 제거합니다. 리스트의 `pop(0)`을 쓰면 나머지 원소를 앞으로 옮기는 비용이 발생합니다.

102 · 정당성·시간·메모리의 근거

각 외부 반복 시작 때 큐에는 현재 레벨만 있습니다. 그 수만 꺼내면 새 자식은 다음 레벨 큐로 남습니다. 왼쪽 자식부터 추가하면 순서도 유지됩니다.

바깥 while의 반복 수는 층 수지만, 안쪽 for의 횟수를 모든 층에 걸쳐 합하면 정확히 n개 노드입니다. 각 노드의 큐 삽입·제거가 O(1)이므로 시간은 O(n)입니다. 넓은 층을 list.pop(0)으로 처리하면 총 이동량이 O(n^2)까지 커질 수 있습니다.

최대 층 너비를 w라 하면 큐에는 현재 층의 나머지와 다음 층 일부가 함께 있어도 O(w)개만 저장됩니다. 보조 큐는 O(w), 반환하는 모든 층의 값은 O(n)입니다. 결과를 포함한 전체 추가 공간은 O(n)입니다.

98 · 풀이의 출발점과 실행 추적

LeetCode 98: Validate Binary Search Tree · Medium

모든 왼쪽 후손이 작고 오른쪽 후손이 큰 BST인지 판정합니다.

부모와 자식만 비교하면 조상에게서 내려온 조건을 놓칩니다. 루트가 5이면 오른쪽 부분 트리의 모든 값이 5보다 커야 합니다. 각 노드에 허용 구간 `(lower, upper)`을 전달하면 모든 조상의 제약을 두 경계로 요약할 수 있습니다.

루트 5, 오른쪽 자식 8, 그 왼쪽 자식 4인 트리를 검사합니다.

노드 값	허용 구간	검사 결과
5	(-∞,∞)	통과
8	(5,∞)	통과
4	(5,8)	5 < 4가 거짓 → False

4는 부모 8보다 작지만 루트 5의 오른쪽에 올 수 없습니다.

98 · 아이디어를 코드로 옮기기

```
def valid_bst(root):
    stack = [(root, float("-inf"), float("inf"))]
    while stack:
        node, lower, upper = stack.pop()
        if node is None:
            continue
        if not lower < node.val < upper:
            return False
        stack.append((node.right, node.val, upper))
        stack.append((node.left, lower, node.val))
    return True
```

오른쪽 자식에는 lower를 현재 값으로 올리고 upper는 유지합니다. 왼쪽 자식에는 upper를 현재 값으로 내리고 lower는 유지합니다.

`lower < node.val < upper`의 엄격한 부등식은 중복 값을 허용하지 않는 BST 조건입니다. None은 실제 노드가 아니므로 건너뜁니다.

98 · 정당성·시간·메모리의 근거

범위는 모든 조상에서 온 제약의 교집합입니다. 범위를 만족하면 조상 관계를 모두 만족하고, 자식 범위를 줄여 같은 성질을 유지합니다. 한 위반이라도 전체가 False입니다.

유효한 트리에서는 n개 노드와 각 노드의 고정 개수 빈 자식을 처리합니다. 노드당 두 경계 비교와 스택 조작만 하므로 시간은 O(n)입니다. 잘못된 노드를 빨리 만나면 조기 종료할 수 있지만 최악 비용은 그대로입니다.

각 스택 항목은 노드 참조와 숫자 두 개입니다. 항목 크기는 O(1)이고 동시에 남는 항목 수가 O(h)이므로 보조 공간은 O(h), 최악 O(n)입니다. 경계값을 추가해도 항목당 고정 크기라는 점은 바뀌지 않습니다.

230 · 풀이의 출발점과 실행 추적

LeetCode 230: Kth Smallest Element in a BST · Medium

BST에서 작은 순서로 k번째 값을 반환합니다.

BST에서는 왼쪽 부분 트리, 현재 노드, 오른쪽 부분 트리 순으로 읽으면 값이 오름차순입니다. 모든 값을 배열에 저장한 뒤 정렬할 필요 없이, 이 순서대로 k번째 값을 만나는 순간 멈춥니다.

루트 3, 왼쪽 1(오른쪽 자식 2), 오른쪽 4. k=2이며 스택 오른쪽이 top입니다.

동작	stack	남은 k	방문 값
왼쪽으로 3,1 저장	[3,1]	2	없음
1 pop, 오른쪽 2로	[3]	1	1
2 저장	[3,2]	1	1
2 pop	[3]	0	1,2 → 2 반환

230 · 아이디어를 코드로 옮기기

```
def kth_smallest(root, k):
    stack, node = [], root
    while stack or node:
        while node:
            stack.append(node)
            node = node.left
        node = stack.pop()
        k -= 1
        if k == 0:
            return node.val
        node = node.right
    raise ValueError("k exceeds tree size")
```

안쪽 while은 아직 방문하지 않은 왼쪽 경로를 저장합니다. pop할 때 비로소 현재 노드를 방문하므로 그때 k를 1 줄입니다. 그 뒤 오른쪽 부분 트리에서 다시 왼쪽 끝을 찾습니다.

스택에는 오른쪽 부분을 나중에 처리해야 하는 조상이 남습니다. 입력 조건은 유효한 BST와 `1 <= k <=` 노드 수입니다.

230 · 정당성·시간·메모리의 근거

BST의 왼쪽 전체는 현재보다 작고 오른쪽 전체는 큽니다. 귀납적으로 inorder는 정렬 순서이므로 k번째 방문 값이 답입니다.

초기 왼쪽 경로를 내려가는 데 최대 h번 push하고 k개 노드를 pop합니다. 방문 중의 push는 이미 pop한 노드들과 아직 남은 최대 h개 경로 항목으로 계산할 수 있어 시간은 O(h+k)입니다. k가 n이면 전체 O(n)입니다.

동시에 저장하는 것은 조상 경로 최대 h개이므로 보조 공간은 O(h)입니다. 전체 순회 결과 배열을 만들지 않습니다. 균형 트리는 h=O(log n)이지만 편향 트리는 h=O(n)이므로 항상 로그 공간이라고 말할 수는 없습니다.

중간 점검과 제출물

1. `[4,2,7,None,3]`의 최대 깊이와 레벨별 값을 실행 전에 적습니다.
2. `[8,3,10,None,None,6,12]`가 BST인지 근거와 함께 판정합니다.
3. BST `[5,2,8,1,3]`에서 k=1, 3, 5의 반환값을 구합니다.

코드·상태 정의·정당성·비용·경계 입력 제출. 학생 실습을 먼저 풀고 해설과 비교합니다.

실습과 참고자료

LeetCode · 104, 102, 98, 230 · 풀이 · Python ZIP

```
python demo_07.py
python demo_08.py
python verify.py
```

학생 파일을 구현한 뒤 `python verify.py --student`로 검사합니다.

Erik Demaine, Srini Devadas. MIT 6.006, Introduction to Algorithms (2011).