



부산대학교

PUSAN NATIONAL UNIVERSITY

BST Insertion and Deletion

Keeping the Order

Inserting, Deleting, and What It Costs

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Insertion by Search

탐색이 멈춘 자리가 곧 넣을 자리다

탐색이 멈춘 자리에 새 노드를 넣는다

ordering invariant(순서 불변식)를 지키면서 넣을 수 있는 자리는 하나뿐이다

	하는 일	어디에서
1	넣을 값을 탐색한다	루트에서 시작
2	탐색이 멈춘 자리를 확인한다	비어 있는 자식 자리
3	그 자리에 단말 노드로 붙인다	탐색이 멈춘 자리

멈춘 자리에 넣으면 ordering invariant가 유지된다

근거	내용
지나온 노드	새 값과 이미 대소를 비교했다
비교에 따른 방향	모두 ordering invariant에 맞는 쪽으로 내려왔다
결론	멈춘 자리에 놓아도 조건이 깨지지 않는다

이미 있는 키를 다시 넣을 때의 규칙

정할 것	보통 하는 선택
중복 허용 여부	허용하지 않는다
중복을 허용하지 않을 때	아무것도 하지 않거나 값을 덮어쓴다

Caution 중복을 허용하려면 어느 쪽 부분 트리에 돌지 미리 정해야 한다

일곱 개를 차례로 넣으면

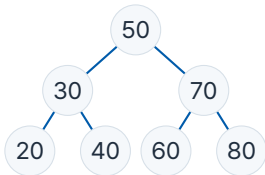
50, 30, 70, 20, 40, 60, 80을 순서대로 넣어 본다

넣은 값	지나온 길	붙은 자리
50	없음	루트
30	50에서 왼쪽	50의 왼쪽
70	50에서 오른쪽	50의 오른쪽
20	50, 30에서 왼쪽	30의 왼쪽
40	50에서 왼쪽, 30에서 오른쪽	30의 오른쪽
60	50에서 오른쪽, 70에서 왼쪽	70의 왼쪽
80	50, 70에서 오른쪽	70의 오른쪽

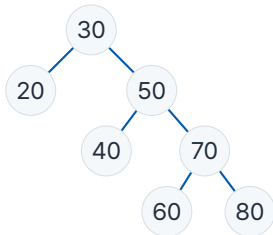
지나온 노드마다 키를 한두 번 비교하므로 비교 횟수는 지나온 길의 길이에 비례한다

같은 키도 넣는 순서에 따라 모양이 달라진다

50, 30, 70, 20, 40, 60, 80



30, 20, 50, 40, 70, 60, 80



두 트리의 중위 순회는 모두 20부터 80까지 같은 순서지만, 높이는 왼쪽이 2, 오른쪽이 3이다.

키 집합이 같아도 삽입 순서가 트리의 모양과 높이를 정한다

재귀로 적으면

경우	하는 일
빈 자리에 닿으면	새 노드를 만들어 돌려준다
값이 더 작으면	왼쪽에 넣고 그 결과를 왼쪽에 잇는다
값이 더 크면	오른쪽에 같은 일을 한다

`node.left = insert(node.left, key, value)`처럼 반환된 부분 트리 루트를 다시 저장한다.

돌려받은 루트를 부모의 자식 참조에 대입해야 새 노드까지 연결된다

Checkpoint

Insertion by Search

- 새 값을 넣을 자리를 어떻게 찾는가?
- 새 노드는 트리의 어디에 붙는가?
- 같은 값들을 다른 순서로 넣으면 무엇이 달라지는가?

Deletion: Leaf or One Child

빈자리가 생기지 않는 두 경우를 먼저 본다

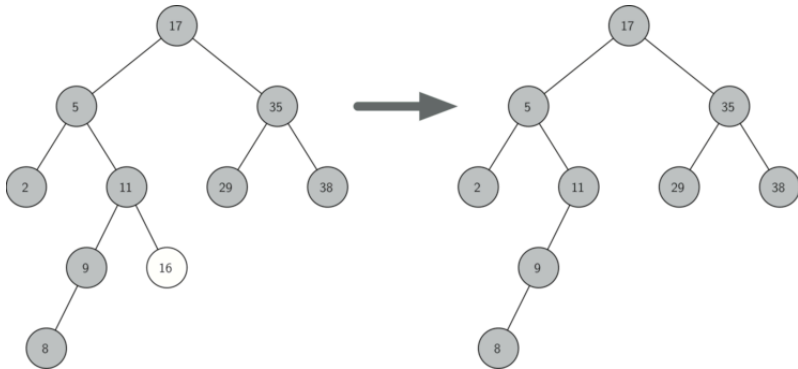
지우는 일이 어려운 이유

노드를 빼면 그 자리가 비고 아래에 매달린 부분 트리가 갈 곳을 잃는다

경우	자식 노드 수	어려움
1	없음	없다
2	하나	잇기만 하면 된다
3	둘	올릴 노드를 골라야 한다

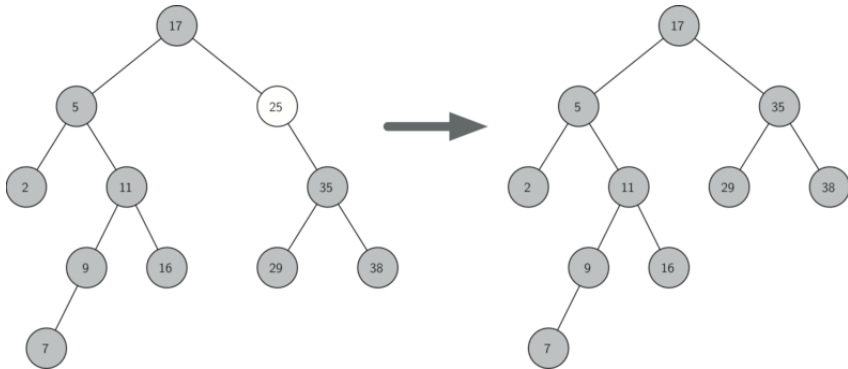
경우를 나누는 기준은 지우려는 노드의 자식 노드 수다

경우 1: 단말 노드



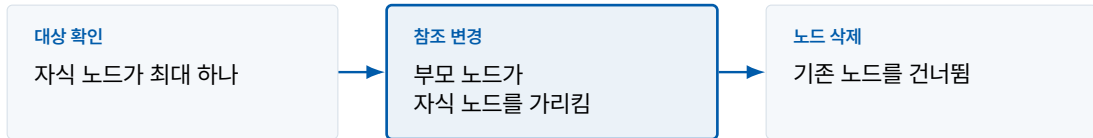
아래에 아무것도 없으므로 부모 노드의 참조만 비우면 끝난다.

경우 2: 자식 노드가 하나



지우는 노드의 자리에 그 하나뿐인 자식 노드를 그대로 끌어올린다.

노드 이어 붙이기



splice(노드를 떼어내고 부모와 자식을 직접 잇기)는 참조 하나를 바꾸므로 $O(1)$ 이다

자식 부분 트리를 올려도 ordering invariant가 유지된다

자식 부분 트리의 모든 값은 삭제한 노드에 허용됐던 값이므로 부모 노드와의 대소 관계도 유지된다

근거	내용
올라온 부분 트리의 값	삭제한 노드의 허용 범위 안에 있었다
부모 노드와의 관계	그 범위 안이므로 대소 관계가 그대로다

20을 지워 보면

단말 노드이므로 부모 노드의 참조만 비우면 끝난다

모양	
삭제 전	$50(30(20, 40), 70(60, 80))$
20을 지운 뒤	$50(30(., 40), 70(60, 80))$

$a(b, c)$ 는 a 의 왼쪽 자식이 b , 오른쪽 자식이 c 라는 뜻이고, 점(.)은 빈 자리다.

나머지 노드는 하나도 움직이지 않았다

자식 노드가 하나인 30을 지우면

앞 예제와 별도로, 30의 자식 노드가 20 하나뿐인 트리에서 30을 지운다

	모양
삭제 전	$50(30(20, .), 70)$
30을 지운 뒤	$50(20, 70)$

Checkpoint

Deletion: Leaf or One Child

- 삭제의 세 경우를 나누는 기준은 무엇인가?
- 자식 노드가 하나인 노드는 어떻게 지우는가?
- 노드를 이어 붙일 수 없는 경우는 언제인가?

Deletion: Two Children

자식 노드가 둘일 때 빈자리를 어느 값으로 메울지 정한다

후계 노드의 값으로 빈자리를 채운다

두 자식 노드 가운데 하나를 올리면 다른 쪽 부분 트리가 갈 곳을 잃는다

선택	결과
왼쪽 자식 노드를 올림 값 하나만 옮김	오른쪽 부분 트리를 다시 매달아야 한다 그 자리에 와도 되는 값이 필요하다

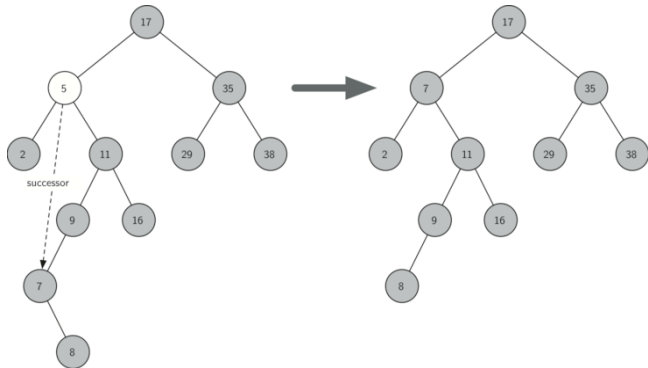
노드를 옮기는 대신 값 하나만 가져오면 모양을 크게 건드리지 않는다

정의

후계 노드(successor)

중위 순회에서 바로 다음에 오는 노드다. 오른쪽 자식 노드가 있으면 오른쪽 부분 트리의 최솟값을 가진 노드이며, 오른쪽 부분 트리에서 왼쪽으로만 내려가면 만난다. 자식 노드가 둘인 노드의 삭제는 항상 이 경우다. 이 값을 지울 자리에 옮겨도 왼쪽 부분 트리의 모든 값보다 크고 오른쪽 부분 트리의 나머지 값보다 작다.

경우 3: 자식 노드가 둘



후계 노드의 값을 지우려는 자리에 옮겨 적고, 원래 후계 노드는 splice로 떼어 낸다.

후계 노드에는 왼쪽 자식 노드가 없다

오른쪽 부분 트리의 최솟값 노드에는 왼쪽 자식 노드가 있을 수 없다

근거	내용
왼쪽 자식 노드가 있다고 가정	그 노드가 더 작으므로 최솟값이 아니다
후계 노드의 자식 노드	없거나 오른쪽 하나뿐이다
삭제 방법	경우 1이나 경우 2로 지운다

세 번째 경우는 결국 앞의 두 경우로 되돌아온다

50을 지워 보면

단계	모양
시작	50(30(20, 40), 70(60, 80))
후계 노드 찾기	오른쪽으로 간 뒤 왼쪽 끝, 60
값 옮기고 splice	60(30(20, 40), 70(., 80))

루트를 지웠는데도 나머지 노드는 제자리에 있다

순서가 지켜졌는지 확인하기

중위 순회	
삭제 전	20, 30, 40, 50, 60, 70, 80
50을 지운 뒤	20, 30, 40, 60, 70, 80

지운 값 하나만 빠지고 나머지 순서는 그대로다

선행 노드를 써도 순서가 유지된다

항목	내용
선행 노드	왼쪽 부분 트리의 최댓값
쓸 수 있는 근거	그 값도 양쪽 부분 트리 사이에 놓인다
후계·선행 노드 선택	구현에서 정하기 나름이다

어느 쪽을 골라도 BST의 키 순서가 유지된다

삭제의 비용

후계 노드를 찾는 일도 한 갈래를 내려가는 일이다

단계	비용	
지울 노드 찾기	높이만큼	탐색
후계 노드 찾기	높이만큼	왼쪽으로만
값 옮기고 잇기	$O(1)$	

두 번 내려가도 여전히 높이에 비례한다

Checkpoint

Deletion: Two Children

- 후계 노드는 어디에서 찾는가?
- 후계 노드에 왼쪽 자식 노드가 없는 이유는 무엇인가?
- 자식 노드가 둘인 노드를 지우는 절차를 순서대로 말해 보라

Height Decides the Cost

세 연산의 비용이 모두 높이에 달려 있음을 본다

세 연산이 같은 것을 한다

탐색도 삽입도 삭제도 루트에서 한 갈래를 따라 내려간다

연산	내려간 뒤 하는 일	비용
탐색	찾았는지 답한다	높이
삽입	단말 노드로 붙인다	높이
삭제	경우에 따라 잇는다	높이

세 연산의 비용은 모두 트리의 높이가 정한다

삽입 순서가 BST의 높이를 결정한다

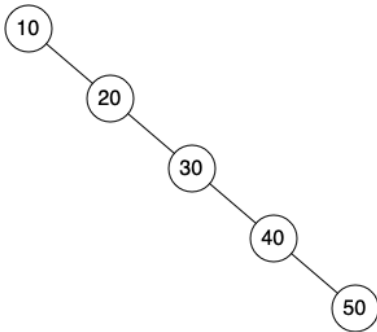
삽입 순서	높이	비용
50, 30, 70, 20, 40, 60, 80	2	$O(\log n)$
10, 20, 30, 40, 50, 60, 70	6	$O(n)$

정의

편향 트리(skewed tree)

한쪽으로만 자라 사실상 한 줄이 된 BST다. 정렬된 순서로 값을 넣으면 이런 모양이 되며, 탐색 경로가 길어져 BST 연산은 연결리스트와 같은 $O(n)$ 비용이 된다.

편향 트리의 모습



10부터 50까지 정렬된 순서로 넣은 모양이다. 한쪽 자식 노드만 계속 이어져 있어 어느 노드에서도 갈림길이 없다.

평균 비용은 최악 비용을 보장하지 않는다

상황	결과
무작위 순서로 삽입	대개 $O(\log n)$
입력 순서	사용하는 쪽이 정하지 못한다
정렬된 입력	실제로 드물지 않다

Caution 평균 비용이 $O(\log n)$ 이어도 최악의 경우에는 $O(n)$ 이 될 수 있다

순서가 필요한 연산에는 BST를 쓴다

해시테이블이 기대 $O(1)$ 인데도 BST를 쓰는 자리가 있다

필요한 연산	해시테이블	BST
키 하나 찾기	기대 $O(1)$	높이만큼
순서대로 출력	별도 정렬 필요	중위 순회
범위로 찾기	모든 키 확인	경계까지 내려가며 범위 안 키만 방문

순서가 필요하면 BST를 써야 하므로 높이가 낮게 유지된다는 보장이 중요해진다

Checkpoint

Height Decides the Cost

- 세 연산의 비용이 모두 높이인 이유는 무엇인가?
- 편향 트리는 어떤 입력에서 생기는가?
- 평균이 $O(\log n)$ 인데도 문제가 되는 이유는 무엇인가?

Section 5 • BST Insertion and Deletion

A BST Map in Python

삽입·삭제의 세 경우가 Python 메서드에서 나뉘는 조건을 확인한다

BST Map 연산을 한 클래스에 모은다

메서드	역할
<code>put, get</code>	키로 값을 저장하고 찾는다
<code>delete</code>	키를 지우고 ordering invariant를 유지한다
<code>_put, _get</code>	한 부분 트리에서 재귀적으로 작업한다
<code>__contains__</code>	<code>in</code> 으로 키가 있는지 답한다
<code>__iter__</code>	키를 중위 순회로 내보낸다

`put`은 해시테이블에서 본 Map의 `set`과 같은 저장 연산이다.

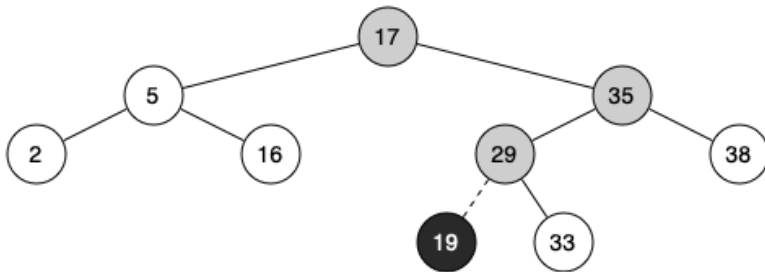
put은 빈 트리와 일반 경우를 나눈다

```
def put(self, key, value):  
    if self.root is None:  
        self.root = TreeNode(key, value)  
    else:  
        self._put(key, value, self.root)
```

루트가 있으면 현재 부분 트리를 인자로 받는 보조 메서드가 삽입 위치를 찾는다.

실패한 탐색 위치에 단말 노드를 붙인다

키 19의 탐색이 비어 있는 자리에서 끝나면 그 자리에 새 노드를 연결한다.



새 노드를 붙인 뒤에도 왼쪽 부분 트리는 작고 오른쪽 부분 트리는 크다.

`_put`은 비교한 방향으로 내려간다

```
def _put(self, key, value, node):  
    if key < node.key:  
        if node.left is None: node.left = TreeNode(key, value)  
        else: self._put(key, value, node.left)  
    elif key > node.key:  
        if node.right is None: node.right = TreeNode(key, value)  
        else: self._put(key, value, node.right)  
    else:  
        node.value = value
```

`_put`은 빈 자식을 찾은 부모의 `left`나 `right`를 직접 바꾸므로 반환값으로 재연결하지 않는다.

같은 키는 값을 갱신한다

비교 결과	동작
키가 작음	왼쪽 부분 트리로 이동
키가 큼	오른쪽 부분 트리로 이동
키가 같음	<code>node.value = value</code>

Map에서는 같은 키의 노드를 하나만 두고 값을 새 값으로 바꾼다.

delete는 자식 노드 수로 나눈다

자식 노드 수	삭제 동작
0	부모 노드의 참조를 비운다
1	부모 노드를 남은 자식 노드에 직접 잇는다
2	후계 노드 값을 복사한 뒤 후계 노드를 splice한다

공개 메서드는 `_delete`만 부르고, 세 경우는 그 안에서 나눈다.

`_delete`는 세 경우를 한 재귀에서 나눈다

```
def _delete(self, key, node):
    if node is None: return None
    if key < node.key: node.left = self._delete(key, node.left)
    elif key > node.key: node.right = self._delete(key, node.right)
    elif node.left is None: return node.right
    elif node.right is None: return node.left
    else:
        s = node.right
        while s.left is not None: s = s.left
        node.key, node.value = s.key, s.value
        node.right = self._delete(s.key, node.right)
    return node
```

앞서 본 반환·재연결 방식으로 공개 `delete`는 `self.root = self._delete(key, self.root)` 한 줄이다.
자식이 둘이면 후계 노드의 키와 값을 옮긴 뒤 오른쪽 부분 트리에서 그 노드를 지운다.

in은 탐색을 재사용하고 순회는 중위 순회로 만든다

```
def __contains__(self, key):  
    return self._get(key, self.root) is not None  
  
def __iter__(self):  
    yield from self._inorder(self.root)  
  
def _inorder(self, node):  
    if node is not None:  
        yield from self._inorder(node.left)  
        yield node.key  
        yield from self._inorder(node.right)
```

존재 확인은 `_get`을 재사용하고, 정렬된 키 순회는 중위 순회 생성기 `_inorder`로 만든다.

구현 뒤 확인할 조건

확인	기대 결과
삽입 뒤 중위 순회	키가 오름차순
삭제 세 경우 뒤 중위 순회	남은 키가 오름차순
없는 키 탐색·삭제	정한 실패 규칙을 따름
같은 키 재삽입	노드 수는 같고 값만 갱신

각 메서드의 결과뿐 아니라 ordering invariant가 계속 유지되는지도 검사한다.

Checkpoint

A BST Map in Python

- 공개 put과 보조 메서드 _put은 역할이 어떻게 다른가?
- 같은 키를 다시 넣을 때 노드를 새로 만들지 않는 이유는 무엇인가?
- __iter__가 중위 순회를 사용하면 어떤 순서가 보장되는가?

자료구조

What If Order Is Only Partial?

Next Lecture: Heaps and Priority Queues

- 우선순위 큐가 답하는 것
- 완전 이진트리를 배열로 담는 법
- 힙 성질