



부산대학교

PUSAN NATIONAL UNIVERSITY

Binary Search Trees

Choose One Subtree

Ordering Invariant and Search Path

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

The Dictionary Problem

Map의 연산과 순서 순회를 함께 지원할 구현을 비교한다

정의

이진 탐색 트리(binary search tree, BST)

각 노드가 키를 가지며 왼쪽과 오른쪽 부분 트리의 키 순서를 이용해 Map을 구현하는 이진트리다. 정렬 연결리스트는 순서를 유지해도 탐색이 $O(n)$ 이고, 정렬 배열·해시테이블·BST는 같은 Map 연산을 서로 다른 비용으로 수행한다.

Map에 필요한 연산

연산	질문
탐색	주어진 키가 있는가? 값은 무엇인가?
삽입·갱신	새 키를 어디에 두고 기존 값은 어떻게 바꾸는가?
삭제	키를 없앤 뒤 구조를 어떻게 유지하는가?
순서 순회	키를 작은 순서로 읽을 수 있는가?

Adapted from UBC DSCI 221 (n.d.), The Dictionary Problem.

정렬하지 않은 배열

41	16	70	9	53	28
----	----	----	---	----	----

새 키임을 알면 끝에 두면 되지만, 같은 키를 확인하거나 키 53을 찾으려면 앞에서부터 비교해야 한다.

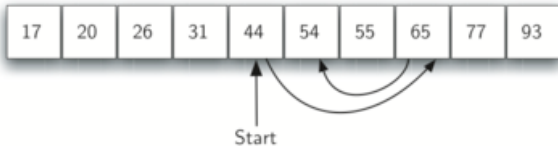
정렬하지 않은 배열은 삽입 위치를 쉽게 정하지만 탐색이 길어진다.

정렬 배열은 탐색 범위를 반씩 줄인다

정의

이진 탐색(binary search)

정렬 배열의 가운데 값과 비교해 남은 탐색 구간을 절반으로 줄이는 알고리즘이다. 탐색은 $O(\log n)$ 이지만 가운데에 삽입하면 뒤 항목을 옮겨야 하므로 $O(n)$ 이다.



54를 찾을 때 가운데 44에서 시작해 65를 거쳐 54에 닿는다. 비교할 때마다 남은 칸이 절반으로 준다.

Source: Miller and Ranum (2013), The Binary Search, Figure 3.

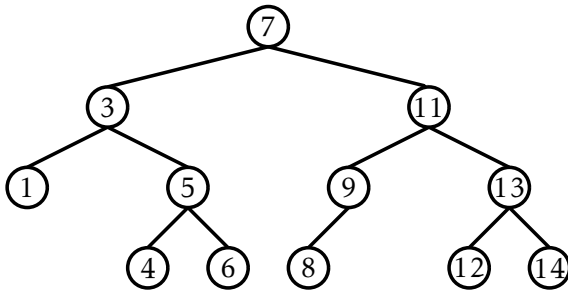
해시테이블은 순서를 저장하지 않는다

장점	제약
탐색·삽입·삭제 기대 $O(1)$	키의 대소 순서를 보존하지 않음
키로 버킷을 바로 계산	작은 키부터 읽으려면 별도 정렬 필요

빠른 키 조회만 필요하면 해시테이블이 알맞지만 정렬 순회는 기본 연산이 아니다.

BST는 비교 결과로 한쪽을 고른다

각 노드에서 찾는 키가 더 작은지 큰지만 확인하고 한쪽 부분 트리로 내려간다.



키의 순서를 구조에 담으면 탐색과 정렬 순회를 같은 트리에서 지원할 수 있다.

세 구현의 비용을 한 표로 본다

구현	탐색	삽입	삭제	순서 순회
정렬 배열	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
해시테이블	기대 $O(1)$	기대 $O(1)$	기대 $O(1)$	별도 정렬 필요
BST	경로 길이	경로 길이	경로 길이	$O(n)$

BST의 경로 길이는 루트에서 탐색이 멈춘 곳까지의 거리이며, 트리 모양에 따라 달라진다.

BST는 세 연산을 한 경로로 처리하면서 순서 순회도 $O(n)$ 에 제공한다.

Checkpoint

The Dictionary Problem

- 정렬 배열은 탐색과 삽입 가운데 어느 연산이 더 비싼가?
- 해시테이블에서 키를 작은 순서로 읽으려면 무엇이 더 필요한가?
- 갱신과 정렬 순회를 함께 원할 때 BST가 적합한 이유는 무엇인가?

The Ordering Invariant

BST를 만드는 키 순서 조건을 부분 트리 전체에 적용한다

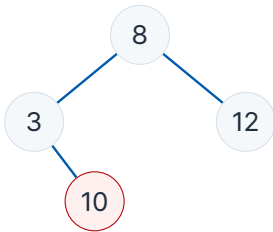
BST를 만드는 한 가지 조건

정의

ordering invariant(순서 불변식)

모든 노드가 지키는 키 순서 조건이다. 노드 u 의 왼쪽 부분 트리에 있는 모든 키는 $u.key$ 보다 작고 오른쪽 부분 트리의 모든 키는 더 크며, 이 조건은 루트뿐 아니라 모든 노드에서 성립해야 한다.

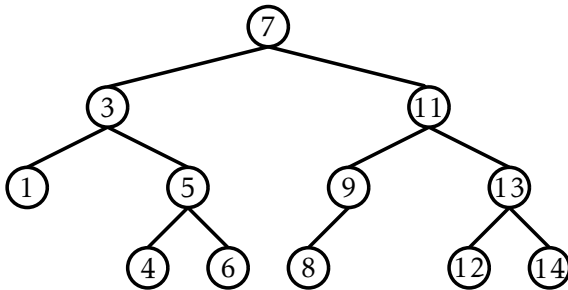
조건은 자식 노드 하나가 아니라 부분 트리 전체에 걸린다



3 아래의 10이 루트 8보다 커서 왼쪽 부분 트리의 ordering invariant를 어기므로 BST가 아니다.

올바른 BST를 읽어 본다

각 노드를 기준으로 왼쪽의 모든 키는 작고 오른쪽의 모든 키는 크다.



한 노드에서 조건이 맞는지만 보지 말고 그 아래 부분 트리의 모든 키를 함께 확인한다.

허용 범위를 내려보내면 조건을 검사할 수 있다

```
def is_bst(node, low, high):  
    if node is None:  
        return True  
    if not (low < node.key < high):  
        return False  
    return (is_bst(node.left, low, node.key)  
            and is_bst(node.right, node.key, high))  
  
is_bst(root, float("-inf"), float("inf"))
```

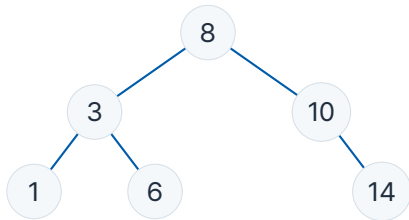
왼쪽으로 갈 때는 상한을, 오른쪽으로 갈 때는 하한을 현재 키로 좁힌다.

중복 키 정책을 먼저 정한다

정책	같은 키를 다시 넣으면
Map 방식	기존 키의 값을 갱신
중복 허용 방식	정한 규칙에 따라 한쪽에 배치
BinarySearchTree	기존 값을 갱신

중복 처리 규칙이 없으면 같은 입력으로도 서로 다른 구조가 만들어질 수 있다.

중위 순회로 읽으면 키가 정렬된다



중위 순회의 출력은 1, 3, 6, 8, 10, 14로 정렬된다.

정렬 결과는 부분 트리에서 이어진다

중위 순회 단계	ordering invariant가 보장하는 것
왼쪽 부분 트리	현재 키보다 작은 값들
현재 노드	두 범위의 경계
오른쪽 부분 트리	현재 키보다 큰 값들

각 부분 트리의 중위 순회도 정렬되어 있으므로 세 부분을 이으면 전체가 정렬된다.

최솟값과 최댓값은 끝까지 한쪽으로 간다

원하는 값	이동
최솟값	왼쪽 자식 노드가 없을 때까지 왼쪽으로
최댓값	오른쪽 자식 노드가 없을 때까지 오른쪽으로

ordering invariant가 방향을 정해 주므로 다른 부분 트리는 볼 필요가 없다.

ordering invariant가 제공하는 연산

연산	ordering invariant의 역할
키 탐색	비교 결과로 한쪽 부분 트리 제외
최솟값·최댓값	한 방향의 끝 노드 선택
정렬 순회	중위 순회 결과를 오름차순으로 만들

BST의 여러 연산은 모두 같은 ordering invariant를 사용한다.

Checkpoint

The Ordering Invariant

- BST의 왼쪽 부분 트리에는 어떤 키가 올 수 있는가?
- 자식 노드와의 대소 관계만 맞아도 BST라고 할 수 없는 이유는 무엇인가?
- BST를 중위 순회로 순회하면 키가 어떤 순서로 출력되는가?

The Search Path

비교 결과로 한쪽 부분 트리를 고르며 탐색 경로를 계산한다

탐색은 루트에서 시작한다

정의

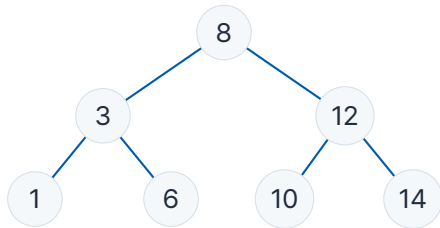
탐색 경로(search path)

루트에서 시작해 비교 결과에 따라 내려가며 방문한 노드의 나열이다. 현재 키가 찾는 키와 같으면 성공하고 내려갈 자식 노드가 없으면 실패한다.

비교 결과는 세 가지다

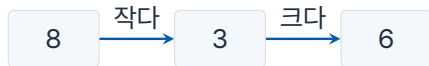
비교	다음 동작
$x < u.key$	왼쪽 자식 노드로 이동
$x > u.key$	오른쪽 자식 노드로 이동
$x = u.key$	현재 노드에서 성공

탐색 예제에 사용할 BST



같은 트리에서 6을 찾는 성공 경로와 11을 찾는 실패 경로를 비교한다.

6을 찾는 경로



$6 < 8$ 이므로 왼쪽, $6 > 3$ 이므로 오른쪽으로 간 뒤 키 6을 찾는다.

성공 탐색 경로는 $8 \rightarrow 3 \rightarrow 6$ 이다.

성공 탐색을 표로 추적한다

단계	현재 키	판정
1	8	6이 작으므로 왼쪽
2	3	6이 크므로 오른쪽
3	6	같으므로 성공

11을 찾으면 비어 있는 자리에서 멈춘다



11 > 8, 11 < 12, 11 > 10이므로 오른쪽 자식 노드를 확인하지만 그 자리가 비어 있어 탐색이 실패한다.

실패 경로도 정보를 남긴다

단계	현재 키	판정
1	8	11이 크므로 오른쪽
2	12	11이 작으므로 왼쪽
3	10	11이 크므로 오른쪽
4	None	실패

마지막 노드와 비어 있는 방향은 새 키가 들어갈 위치를 알려 준다.

한 번의 비교가 부분 트리 하나를 버린다

현재 상황	보지 않아도 되는 곳
찾는 키가 더 작음	현재 노드와 오른쪽 부분 트리
찾는 키가 더 큼	현재 노드와 왼쪽 부분 트리
키가 같음	두 부분 트리 모두

```
def find(root, key):  
    node = root  
    while node is not None:  
        if key < node.key: node = node.left  
        elif key > node.key: node = node.right  
        else: return node  
    return None
```

비용은 탐색 경로 길이다

측정	탐색에서의 의미
방문 노드 수	노드마다 키 비교 한두 번
탐색 경로 길이	루트에서 멈춘 곳까지의 거리
최대 경로	트리의 높이 이하

탐색 시간은 $O(\text{높이})$ 이며 트리 모양에 따라 탐색 경로 길이가 달라진다.

Checkpoint

The Search Path

- 찾는 키가 현재 키보다 작을 때 어느 자식 노드로 이동하는가?
- 실패 탐색은 어떤 조건에서 끝나는가?
- BST 탐색 비용을 전체 노드 수보다 탐색 경로 길이로 설명하는 이유는 무엇인가?

Section 4 • Binary Search Trees

Search in Python

BST 탐색 절차를 Python 클래스의 메서드로 구현한다

구조와 연산을 나눈다

클래스	책임
TreeNode	키, 값, 왼쪽, 오른쪽 저장
BinarySearchTree	루트 관리와 Map 연산 제공

노드의 연결 정보와 트리 전체 연산을 분리하면 탐색 보조 메서드가 단순해진다.

TreeNode는 네 필드를 가진다

```
class TreeNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
```

ordering invariant는 key에 적용되고 value는 키로 찾은 뒤 반환하거나 갱신한다. 앞에서 본 BinaryTree의 left_child, right_child와 같은 역할을 이 클래스는 left, right로 부른다.

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def get(self, key):
        node = self._get(key, self.root)
        if node is None:
            return None
        return node.value
```

get은 탐색을 _get에 맡기고 결과 노드에서 값만 꺼낸다.

재귀 보조 메서드 _get

```
def _get(self, key, node):  
    if node is None: return None  
    if key == node.key: return node  
    if key < node.key:  
        return self._get(key, node.left)  
    return self._get(key, node.right)
```

반복문으로도 같은 경로를 만든다

```
def search(self, key):  
    node = self.root  
    while node is not None and node.key != key:  
        if key < node.key: node = node.left  
        else: node = node.right  
    return node
```

재귀와 반복문은 표현만 다르고 같은 탐색 경로를 따라간다.

in 연산자는 탐색 결과만 확인한다

```
def __contains__(self, key):  
    return self._get(key, self.root) is not None
```

```
tree = BinarySearchTree()  
tree.root = TreeNode(8, "a")  
tree.root.left = TreeNode(3, "b")  
tree.root.left.right = TreeNode(6, "c")  
print(6 in tree, 11 in tree)
```

노드를 손으로 이어 만든 트리에서 출력은 True False다. 같은 `_get` 결과를 값 조회와 키 존재 확인에 함께 쓴다.

반환 규칙은 값의 범위와 함께 정한다

상황	선택 가능한 규칙
키가 없음	None 반환 또는 예외 발생
값으로 None 허용	노드 존재 여부를 따로 확인
이번 예제	_get은 노드 또는 None 반환

키가 없는 경우와 값이 None인 경우를 구분하도록 보조 탐색은 노드를 반환한다.

탐색·검사를 구현한 뒤 삽입과 삭제를 다룬다

탐색·검사	삽입·삭제·비용
노드와 루트 구조	삽입 경로와 새 단말 노드 연결
get, _get, in	삭제 세 경우와 빈자리를 채울 노드
ordering invariant와 중위 순회 검사	높이에 따른 비용 비교

먼저 ordering invariant와 탐색을 정확히 구현한 뒤 구조를 바꾸는 연산을 더한다.

Checkpoint

Search in Python

- `TreeNode`와 `BinarySearchTree`는 각각 무엇을 저장하고 관리하는가?
- `_get`이 현재 노드를 인자로 받는 이유는 무엇인가?
- 재귀 구현과 반복문 구현이 같은 결과를 내는 이유는 무엇인가?

How Does a Comparison Pick a Subtree?

Next Lecture: BST Insertion and Deletion

- 새 키를 넣는 자리
- 자식이 둘인 노드의 삭제
- 후계 노드를 찾는 일