

Data Structure

자료구조



이진 탐색 트리



한국기술교육대학교
온라인평생교육원

학습내용

- 이진 탐색 트리의 개념
- 탐색 연산
- 삽입 연산
- 삭제 연산

학습목표

- 이진 탐색 트리의 개념을 설명할 수 있다.
- 탐색 연산의 개념을 설명할 수 있다.
- 삽입 연산의 개념을 설명할 수 있다.
- 삭제 연산의 개념을 설명할 수 있다.

이진 탐색 트리의 개념



이진 탐색 트리

이진 탐색 트리의 개념

1 이진 탐색 트리란?

이진 탐색 트리

- 효율적인 탐색을 위한 이진 트리 기반의 자료구조
- 삽입, 삭제, 탐색 $O(\log n)$

▶ 탐색 기법들의 성능 비교

탐색 방법		탐색	삽입	삭제
순차 탐색		$O(n)$	$O(1)$	$O(n)$
이진 탐색		$O(\log_2 n)$	$O(n)$	$O(n)$
이진 탐색 트리	균형 트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사 트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$

이진 탐색 트리

이진 탐색 트리의 개념

1 이진 탐색 트리란?

이진 탐색 트리

- 효율적인 탐색을 위한 이진 트리 기반의 자료구조
- 삽입, 삭제, 탐색 $O(\log n)$

▶ 탐색 기법들의 성능 비교

탐색 방법		탐색	삽입	삭제
순차 탐색		$O(n)$	$O(1)$	$O(n)$
이진 탐색		$O(\log_2 n)$	$O(n)$	$O(n)$
이진 탐색 트리	균형 트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사 트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$

이진 탐색 트리의 개념



이진 탐색 트리

이진 탐색 트리의 개념

1 이진 탐색 트리란?

이진 탐색 트리의 정의

- 모든 노드는 유일한 키를 갖는다.
- 왼쪽 서브 트리의 키들은 루트의 키보다 작다.
- 오른쪽 서브 트리의 키들은 루트의 키보다 크다.
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리이다.



이진 탐색 트리

이진 탐색 트리의 개념

1 이진 탐색 트리란?

◎ 이진 탐색 트리의 예



왼쪽 서브 트리 노드의 키값 < 루트 노드의 키값 < 오른쪽 서브 트리 노드의 키값

이진 탐색 트리의 개념



이진 탐색 트리

이진 탐색 트리의 개념

2 이진 탐색 트리의 연산

탐색 연산

- 키를 이용한 탐색 → 효율적인 탐색
- 값을 이용한 탐색 → 효율이 좋지 않을 수 있음
- 최대 노드와 최소 노드 탐색

삽입 / 삭제 연산

- 새로운 노드가 트리에 추가되거나 기존의 노드가 삭제
- 이진 탐색 트리의 특성이 그대로 유지되어야 함
 - 탐색 연산 ▶ 트리 구조 변경 시키지 않음
 - 삽입, 삭제 연산 ▶ 이진 탐색 트리의 특성이 유지되지 않을 수 있음

이진 탐색 트리

이진 탐색 트리의 개념

2 이진 탐색 트리의 연산

● 이진 탐색 트리의 구현 방법

연결된 구조가 유리

- 완전 이진 트리(예: 힙 트리) 구조를 보장하지 않음

이진 탐색 트리를 위한 노드의 구조

- 엔트리(탐색키, 키에 대한 값)의 형태
- 좌우 자식 노드에 대한 링크 필드

```
class TNode:
    def __init__(self, elem, left, right):
        self.data = elem
        self.left = left
        self.right = right
    def isLeaf(self):
        return self.left is None and self.right is None
```

엔트리(key,value 2개의 필드로 이루어진 형태의 데이터)
left
right
단말 노드인지 찾는 연산

탐색 연산



이진 탐색 트리 탐색 연산

1 키를 이용한 탐색

아이디어

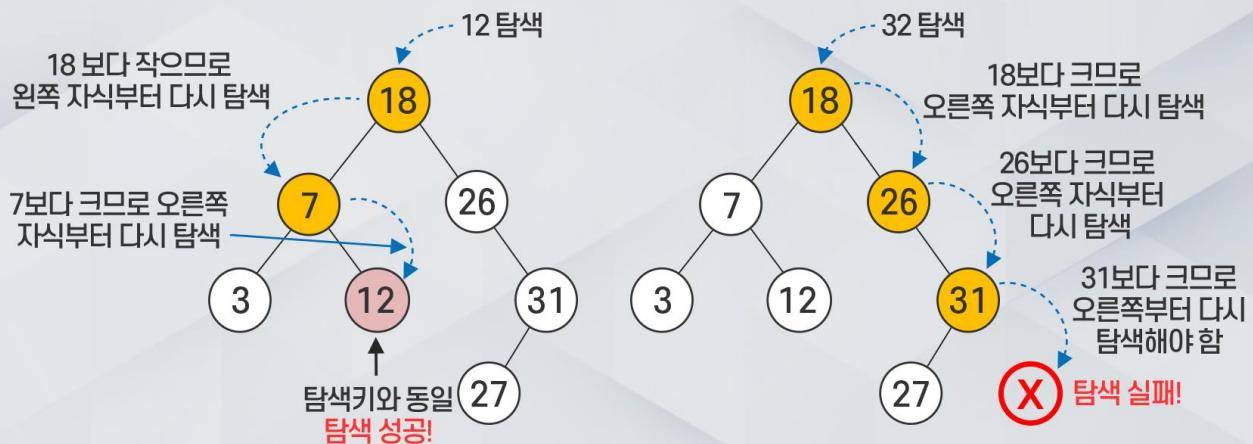
- 키를 기준으로 이진 탐색 트리가 구성
- 루트 노드부터 탐색 시작
 - 찾는 키값이면 → 탐색 성공
 - 찾는 키값이 더 작으면 → 왼쪽 서브 트리 탐색
 - 찾는 키값이 더 크면 → 오른쪽 서브 트리 탐색
 - 서브 트리가 공백 노드이면 → 탐색 실패

이진 탐색 트리 탐색 연산

1 키를 이용한 탐색

● 키를 이용한 탐색 예

▶ 키값이 12와 32인 노드의 탐색 과정



탐색 연산



이진 탐색 트리

탐색 연산

1 키를 이용한 탐색

● 탐색 알고리즘(순환 구조)

순환 구조와 반복 구조로 기술할 수 있음

▶ 순환 구조 알고리즘

```
def search_bst(n, key) :
    if n == None :
        return None
    elif key == n.key:
        return n
    elif key < n.key:
        return search_bst(n.left, key)
    else:
        return search_bst(n.right, key)
```

키를 이용한 탐색
공백 → 탐색 실패
같으면 → 탐색 성공
키가 작으면 → 왼쪽 서브 트리
키가 크면 → 오른쪽 서브 트리

이진 탐색 트리

탐색 연산

1 키를 이용한 탐색

● 탐색 알고리즘(반복 구조)

▶ 반복 구조 알고리즘

```
def search_bst_iter(n, key) :
    while n != None :
        if key == n.key:
            return n
        elif key < n.key:
            n = n.left
        else:
            n = n.right
    return None
```

키를 이용한 탐색(반복 구조)
공백이 아닐 때 까지
같으면 → 탐색 성공
키가 작으면 → 왼쪽 서브 트리
키가 크면 → 오른쪽 서브 트리



탐색 연산

이진 탐색 트리 탐색 연산

2 값 이용한 탐색

이진 탐색 트리는 값에 의해 정렬되어 있지 않음

- 효율적인 탐색이 불가능
- 트리의 순회 알고리즘을 이용해 모든 노드를 검사해야 함
 - 전위, 중위, 후위, 레벨 순회 등을 사용할 수 있음
- 시간 복잡도가 키를 이용한 탐색 보다 더 높음

이진 탐색 트리 탐색 연산

2 값을 이용한 탐색

● 전위 순회 예

```
def search_value_bst(n, value) :          # 값을 이용한 탐색
    if n == None :
        return None
    elif value == n.value:                 # 루트를 검사
        return n
    res = search_value_bst(n.left, value)   # 왼쪽 서브 트리를 먼저 탐색
    if res is not None:                   # 왼쪽에서 찾았으면 → 탐색 성공
        return res
    return search_value_bst(n.right, value) # 오른쪽 서브 트리를 다음에 탐색
```

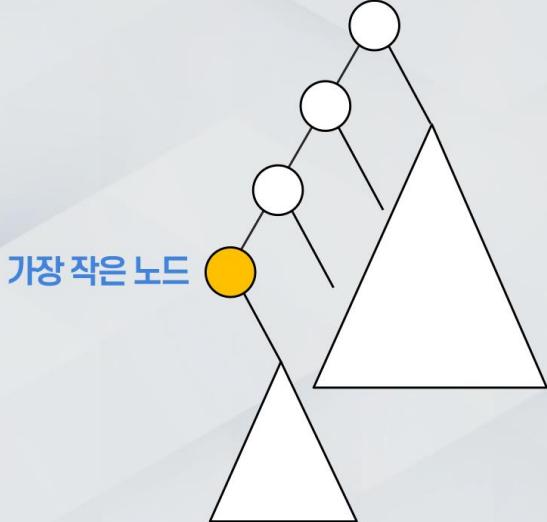


탐색 연산

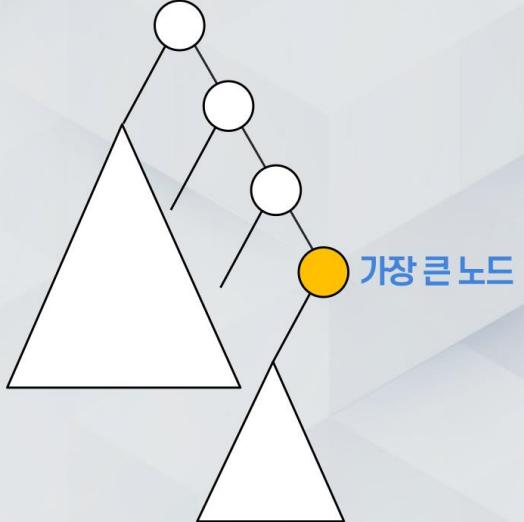
이진 탐색 트리

탐색 연산

3 최대와 최소 노드 탐색



가장 작은 노드 → 트리의 가장 왼쪽 노드



가장 큰 노드 → 트리의 가장 오른쪽 노드

이진 탐색 트리

탐색 연산

3 최대와 최소 노드 탐색

● 탐색 알고리즘

▶ 최소 노드 탐색

```
def search_min_bst(n) :
    while n != None and n.left != None: # 왼쪽 서브 트리가 공백이 아닐때까지
        n = n.left
    return n
```

▶ 최대 노드 탐색

```
def search_max_bst(n) :
    while n != None and n.right != None: # 오른쪽 서브 트리가 공백이 아닐때까지
        n = n.right
    return n
```



삽입 연산

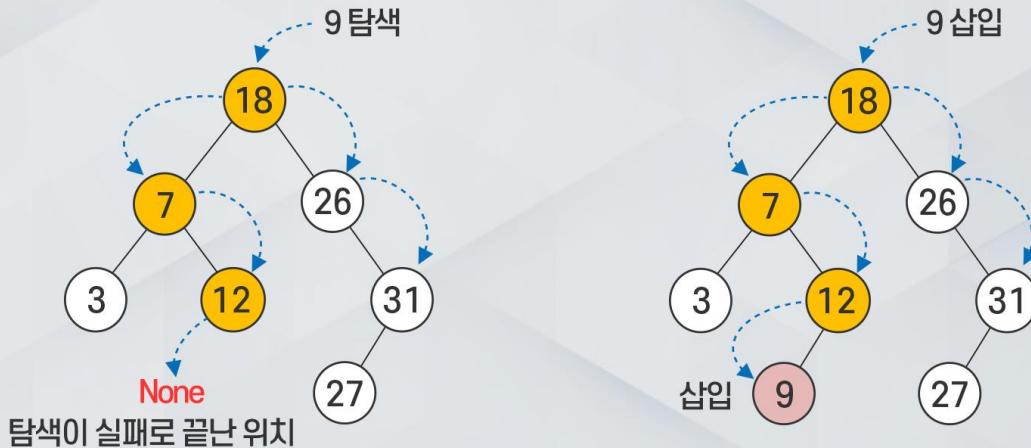
이진 탐색 트리

삽입 연산

1 이진 탐색 트리의 삽입 연산

먼저 키값을 탐색

- 탐색에 실패한 위치 → 노드를 삽입해야 하는 위치



이진 탐색 트리

삽입 연산

2 삽입 연산 알고리즘

아이디어

- 루트에서부터 탐색 연산을 진행
- 탐색이 성공하면?
 - 삽입하지 않음 → 중복을 허용하지 않음
- 다음 탐색 위치가 None이면?
 - 해당 위치에 삽입

삽입 연산의 구현 방법

순환 구조 또는 반복 구조



삽입 연산

이진 탐색 트리 삽입 연산

2 삽입 연산 알고리즘

● 순환 구조의 삽입 연산 알고리즘

```

def insert_bst(r, n) :
    if n.key < r.key:
        if r.left is None :
            r.left = n
            return True
        else :
            return insert_bst(r.left, n)
    elif n.key > r.key :
        if r.right is None :
            r.right = n
            return True
        else :
            return insert_bst(r.right, n)
    else :
        return False
    
```

삽입 연산
 # 왼쪽 서브 트리로 진행
 # 왼쪽 서브 트리가 공백이면
 # 왼쪽 자식으로 삽입

 # 공백이 아니면 순환호출
 # 오른쪽 서브 트리로 진행
 # 오른쪽 서브 트리가 공백이면
 # 오른쪽 자식으로 삽입

삭제 연산



이진 탐색 트리 삭제 연산

1 이진 탐색 트리의 삭제 연산

3가지 경우로 나누어 처리

- 삭제하려는 노드가 단말 노드일 경우
- 삭제하려는 노드가 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
- 삭제하려는 노드가 두 개의 서브 트리 모두 가지고 있는 경우

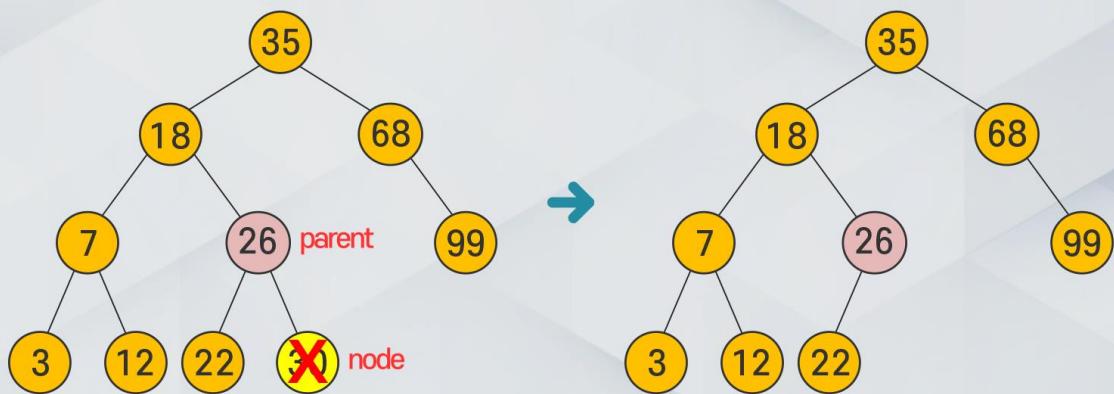
이진 탐색 트리 삭제 연산

1 이진 탐색 트리의 삭제 연산

● 삭제 연산 Case 1

Case 1: 단말 노드 삭제

부모의 해당 자식을 None으로 처리



삭제 연산

이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

● Case 1 단말 노드 삭제 알고리즘

```
def delete_bst_case1 (parent, node, root) :
    if parent is None:
        root = None
    else :
        if parent.left == node :
            parent.left = None
        else :
            parent.right = None
    return root
```

삭제할 단말 노드가 루트이면
공백 트리가 됨

삭제할 노드가 부모의 왼쪽 자식이면
부모의 왼쪽 링크를 None
오른쪽 자식이면
부모의 오른쪽 링크를 None

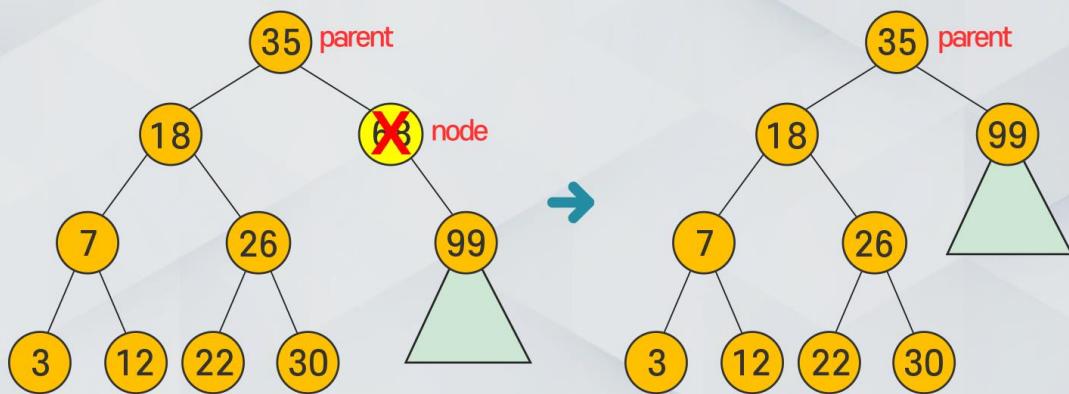
이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

● 삭제 연산 Case 2

Case 2: 자식이 하나인 노드의 삭제

자신의 유일 자식을 부모의 자식으로 연결



삭제 연산



이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

Case 2 자식이 하나인 노드의 삭제 알고리즘

```
def delete_bst_case2 (parent, node, root) :
    if node.left is not None :
        child = node.left
        # 삭제할 노드가 왼쪽 자식만 가짐
        # child는 왼쪽 자식
    else :
        child = node.right
        # 삭제할 노드가 오른쪽 자식만 가짐
        # child는 오른쪽 자식

    if node == root :
        root = child
        # 없애려는 노드가 루트이면
        # 이제 child가 새로운 루트가 됨

    else :
        if node is parent.left :
            parent.left = child
            # 삭제할 노드가 부모의 왼쪽 자식
            # 부모의 왼쪽 링크를 변경
        else :
            parent.right = child
            # 삭제할 노드가 부모의 오른쪽 자식
            # 부모의 오른쪽 링크를 변경

    return root
```

이진 탐색 트리 삭제 연산

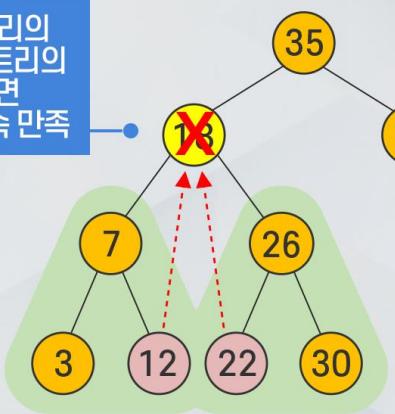
2 삭제 연산 알고리즘

삭제 연산 Case 3

Case 3: 두 개의 자식을
가진 노드 삭제

- 가장 비슷한 값을 가진 노드를 삭제 위치로 가져옴
- 후계 노드의 선택

삭제할 위치에 왼쪽 서브 트리의
가장 큰 노드나 오른쪽 서브 트리의
가장 작은 노드가 들어가면
이진 탐색 트리의 조건을 계속 만족



삭제 연산



이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

● 삭제 연산 Case 3



이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

● Case 3 두 개의 자식을 가진 노드 삭제 알고리즘

```

def delete_bst_case3 (parent, node, root) :
    succp = node
    succ = node.right
    while (succ.left != None) :
        succp = succ
        succ = succ.left

        if (succp.left == succ) :
            succp.left = succ.right
        else :
            succp.right = succ.right

        node.key = succ.key
        node.value= succ.value
    #   node = succ;

    return root
    
```

후계자의 부모 노드
후계자 노드
후계자와 부모노드 탐색

후계자가 왼쪽 자식이면
후계자의 오른쪽 자식 연결
후계자가 오른쪽 자식이면
후계자의 왼쪽 자식 연결

후계자의 키와 값을
삭제할 노드에 복사
실제로 삭제하는 것은 후계자 노드

삭제 연산



이진 탐색 트리 삭제 연산

2 삭제 연산 알고리즘

● Case 4 전체 삭제 알고리즘

```

def delete_bst (root, key) :
    if root == None : return None
    parent = None
    node = root
    while node != None and node.key != key :
        parent = node
        if key < node.key : node = node.left
        else : node = node.right;
        if node == None : return None           # 삭제할 노드가 없음
    if node.left == None and node.right == None :   # case 1
        root = delete_bst_case1 (parent, node, root)
    elif node.left==None or node.right==None :      # case 2
        root = delete_bst_case2 (parent, node, root)
    elif :                                         # case 3
        root = delete_bst_case3 (parent, node, root)
    return root

```

삭제 연산

The screenshot shows the PyCharm IDE interface with two open files: `BSTMap.py` and `MaxHeap.py`. The `BSTMap.py` file contains the following code:

```

48
49     =====
50     class BSTNode:
51         def __init__(self, key, value):
52             self.key = key
53             self.value = value
54             self.left = None
55             self.right = None
56
57         def isLeaf(self):
58             return self.left is None and self.right is None
59
60
61     # 이진탐색트리 탐색연산 (키를 탐색)
62     def search_bst(n, key):
63         if n == None:
64             return None
65         elif key == n.key:
66             return n
67         elif key < n.key:
68             return search_bst(n.left, key)
69         else:
70             return search_bst(n.right, key)

```

The `MaxHeap.py` file is partially visible at the top. The left sidebar shows a tree structure of files, including `BSTMap.py` and `AVLMap.py`.

실습 단계

- 최대의 노드와 최소 노드를 찾는 알고리즘도 동일하게 처리
- 삽입 연산도 마찬가지로 순환을 이용하여 같은 코드를 넣어서 처리
- 루트를 넣고 키값을 넣어서 삭제를 하도록 하는 것
- 이진 탐색 트리에서 가장 큰 노드를 찾는 것
- search는 기본적으로 키값을 이용해서 찾는 것
- 삭제를 하고 난 다음의 루트가 변경 될 수도 있음
- 루트를 간신
- 여러가지 방법 중 레벨 모드를 이용하여 출력
- 이진 탐색 트리를 테스트하는 코드는 다음과 같음
- 지금은 키만 넣고 value는 없음
- 데이터에 있는 항목들을 하나씩 키에 삽입