

---

# 알고리즘및실습

[실습 및 과제 3]

---



과목명/분반	알고리즘및실습/03	담당교수	한연희
학 부 명	컴퓨터공학부	제 출 일	2022/ 04 / 21
학번	2018136121	이름	조원석

## INDEX

표지 및 차례-----	1
서론	
Homework의 내용 및 목적 -----	4
본론	
[문제1] BST에 대한 Python 프로그램 작성하기-----	5
> binary_search_tree.py 분석하기	
> 완성되지 않은 다음 두 함수 작성하기	
- 노드 삽입 : def tree_insert(self, key = None):	
- 노드 삭제 : def _delete_node(self, node_to_be_deleted):	
[문제2] BST에 대한 중위 순회-----	6
1 BST에 대한 중위 순회 결과를 한 줄로 제시하기	
2 비어있는 BST 완성하기	
3 [문제 1]를 통해 [문제2-1] 결과 검증하기	
[문제3] BST에 대한 키 삽입과 중위 순회 -----	7
1 [알고리즘 6-4]에 입각하여 [문제2]의 BST에서 키 40,39 80 삽입 결과 제시하기	
> 서술형으로 작성	
> 과정을 그리고 사진을 찍어 보고서에 첨부할 것	
2 위에서 제시된 키 노드들 삽입된 직후의 BST에 대한 중위 순회 결과 제시하기	
3 [문제2-3] 이후 코드를 이용해 [문제3-1, 3-2] 검증하기	
> 키 삽입될 때마다 출력된 중위 순회 결과 캡처화면 제시	
[문제4] BST에 대한 키 삭제와 중위 순회-----	8
1 [알고리즘 6-6]에 입각하여 [문제3]의 결과에 키 15 노드 삽입하기	
> 서술형으로 작성	
> 과정을 그리고 사진을 찍어 보고서에 첨부할 것	
2 [문제4-1]에 제시된 중위 순회 결과 제시하기	
3 [문제 3-3] 이후 코드를 이용해 [문제4-1, 4-2] 검증하기	
> 키 삽입될 때마다 출력된 중위 순회 결과 캡처화면 제시	

[문제5] B-Tree에 대한 문제 해결하기-----10

1 [알고리즘 6-7]에 입각하여 B-Tree[k=4]에서 키 45 노드 삽입하기

> 최종 결과 그림 제시

> 중간 과정 설명

2 [알고리즘 6-8]에 입각하여 B-Tree[k=4]에서, 언더플로우가 발생한 키 7 노드

삭제 과정 및 결과 그림 제시하기

> 최종 결과 그림 제시

> 중간 과정 설명

[문제6] B-Tree의 임의 노드가 가질 수 있는 최대/최소키 개수 제시하기---12

condition\_

1) 한 블록(페이지) 크기 : 35.448 byte

2) 키의 크기 : 32 byte

3) 페이지 번호 크기 : 4 byte

**결론**

고찰-----15

## 서론 : Homework의 내용 및 목적

### ▶ 내용

: binary\_search\_tree.py 코드상에 정의된 tree\_insert와 \_delete\_node 함수를 구현하여 BST에서 삽입/삭제 등의 연산과 중위 순회가 이루어지는 과정을 살펴보고 임의의 키 노드의 삽입/삭제하는 과정을 그려보기

: B-Tree에서 임의의 키 노드의 삽입/삭제하는 과정을 그려보고, 어떤 조건에서 임의의 노드가 가질 수 있는 최대 키와 최소 키의 개수를 확인해보기

### ▶ 목적

- : BST의 삽입/삭제 구현 및 동작 과정 이해
- : B-Tree의 삽입/삭제 동작 과정 이해
- : B-Tree의 최대 키, 최소 키 개수 도출 과정 학습
- : 트리를 이용한 선형 검색의 이해

## 본론 : 문제 풀이

[문제1] BST에 대한 Python 프로그램 작성하기

### > binary\_search\_tree.py 분석하기

#### ▶ 내용

: binary\_search\_tree.py는 연산을 진행합니다.

클래스를 기준으로 하나하나 살펴보면

먼저 Node에서 객체 인스턴스를 초기화하는 `__init__` 함수를 통해 `key`, `parent`, `left`, `right`를 정의하고 Node의 구조는 `__str__`라는 함수를 통해 출력합니다.

다음으로 BST에서 `__init__` 함수를 통해 Node 초기화, `_inorder` 함수를 통해 중위 순회, `print_bst` 함수를 통해 이진 탐색 트리를 출력하는 등 기본 작업을 해주고, `tree_search`, `tree_insert`, `tree_delete`, `_delete_node` 함수를 통해 이진 탐색 트리의 검색, 삽입, 삭제를 이루어내고 `main`에서 키를 넣고 빼면서 이진 탐색 트리의 동작을 관찰합니다

이 중에서 삽입과 삭제를 실질적으로 구현하는 부분인 `tree_insert` 함수와 `_delete_node` 함수를 구현해보려고 합니다.

### > 완성되지 않은 두 함수 완성하기

▶ *def tree\_insert(self, key = None):*

*# 노드 삽입*

▶ *def \_delete\_node(self, node\_to\_be\_deleted):*

*# 노드 삭제*

## ▶ def tree\_insert(self, key = None):

```
def tree_insert(self, key=None): # treeInsert(self, Key= None)
    # self.root : 트리의 루트 노드
    # key=None : 삽입하고자 하는 키
    new_node = Node(key=key)    #new_Node : 새 노드

    if self.root is None:      # if(트리의 루트 노드가 None일 때)
        self.root = new_node   # then 트리의 루트노드 <- 새 노드
    else:
        # 새로운 노드를 위한 올바른 위치까지 순회하여 새로운 노드의 부모 설정
        current_node = self.root    # 현재 노드 <- 트리의 루트노드
        parent_node_for_new_node = None # 부모 노드 <- None

        #####
        while current_node is not None: # while (현재 노드가 None이 아닐 때)
            parent_node_for_new_node = current_node # 부모 노드 <- 현재 노드
            if key < current_node.key: # if(삽입하고자 하는 키보다 현재 노드의 키가 클 때)
                current_node = current_node.left # 현재 노드 = 현재 노드의 왼쪽 자식
            else: #아닐 때
                current_node = current_node.right # 현재 노드 = 현재 노드의 오른쪽 자식
            if (key < parent_node_for_new_node.key):
                # if(삽입하고자 하는 키보다 부모 노드의 키가 더 클 때)
                parent_node_for_new_node.left = new_node# 부모노드의 왼쪽 자식 = 새 노드
            else: #아닐 때
                parent_node_for_new_node.right =new_node#부모노드의 오른쪽 자식 =새 노드

        #####

        # 새로운 노드의 부모 노드 설정
        new_node.parent = parent_node_for_new_node

    self.size += 1 # 트리의 사이즈를 하나 키워줌
```

def tree\_insert(self, key = None)는 트리에 삽입하는, 즉 비재귀적인 방법으로 이진 검색 트리에 노드를 추가하는 함수입니다. 이 함수는 Node 클래스를 이용해서 삽입하고자 하는 노드인 new\_node, 현재 노드를 나타내는 current\_node, 부모 노드를 나타내는 parent\_node\_for\_new\_node, 루트 노드를 나타내는 self.root로 이루어져 있는데 루트 노드가 없을 때는 새로운 노드를 루트 노드에 넣어줍니다. 루트 노드가 있을 때는 현재 노드를 루트 노드로 설정하고 현재 노드와 key 값을 비교하면서 작을 때는 왼쪽 자식 노드, 클 때는 오른쪽 자식 노드로 설정하고, 새로운 노드의 부모 노드를 설정하고 트리의 크기를 하나 키워줍니다.

위 함수의 평균적인 경우의 점근적 복잡도는  $\theta(\log n)$  이라고 합니다.

## ▶ def tree\_delete(self, key):

```
def tree_delete(self, key):
    node_to_be_deleted = self.tree_search(key)
    if node_to_be_deleted is None:
        return
    else:
        # node_to_be_deleted가 루트인 경우
        if node_to_be_deleted == self.root:
            self.root = self._delete_node(node_to_be_deleted)
        elif node_to_be_deleted == node_to_be_deleted.parent.left:
            node_to_be_deleted.parent.left = self._delete_node(node_to_be_deleted)
        else:
            node_to_be_deleted.parent.right =
self._delete_node(node_to_be_deleted)

        self.size -= 1
```

def tree\_delete(self, key)는 노드를 제거하는 모습을 나타낸 함수입니다.

삭제할 노드가 리프 노드일 경우

삭제할 노드에 자식이 하나일 경우

삭제할 노드의 자식이 둘 일 경우를 나타낸 것으로

이번 코드에서는 직접적으로 동작하지는 않습니다.

## ▶ def \_delete\_node(self, node\_to\_be\_deleted):

```
def _delete_node(self, node_to_be_deleted):#_delete_node(self, key)
    # self.root 트리의 루트 노드
    # node_to_be_deleted: 삭제하고자 하는 노드
    if node_to_be_deleted.left is None and node_to_be_deleted.right is None:
# Case1 리프노드
        return None
    elif node_to_be_deleted.left is None and node_to_be_deleted.right is not None:
# Case2-1 오른쪽 자식만 있음
        return node_to_be_deleted.right
    elif node_to_be_deleted.left is not None and node_to_be_deleted.right is None:
# Case2-2 왼쪽 자식만 있음
        return node_to_be_deleted.left
    else: #Case3 남은 자식이 둘 이상
        # 삭제할 노드의 오른쪽 자식을 smallest_node로 설정
        smallest_node = node_to_be_deleted.right

        # 처음에는 smallest_node의 부모는 삭제할 노드 자신
        parent_of_smallest_node = None

        #####
        while smallest_node.left is not None:
            parent_of_smallest_node = smallest_node
            smallest_node = smallest_node.left
        node_to_be_deleted.key = smallest_node.key
        if smallest_node == node_to_be_deleted.right:
            node_to_be_deleted.right = smallest_node.right
        else:
            parent_of_smallest_node.left = smallest_node.right
        #####

        del smallest_node
        return node_to_be_deleted
```

def \_delete\_node(self, node\_to\_be\_deleted)는 방금 봤던 제거하는 모습을 실제 구현한 함수입니다. Case1 처럼 삭제할 노드가 리프 노드일 경우 삭제할 노드 node\_to\_be\_deleted를 그냥 버리고 Case2 처럼 노드의 자식 노드가 1개인 경우 노드 node\_to\_be\_deleted의 부모 노드가 노드 node\_to\_be\_deleted의 자식을 직접 가리키도록 하고 Case3 처럼 노드 node\_to\_be\_deleted의 자식 노드가 두 개인 경우에는 노드 node\_to\_be\_deleted의 오른쪽 서브 트리의 최소 원소 노드를 올려줍니다.

위 함수의 점근적 복잡도는 Case1과 Case2일 때는 평균적으로  $\Theta(1)$ , Case3일 때는 트리의 높이에 따라  $O(\log n) \sim O(n)$ 을 따른다고 합니다.



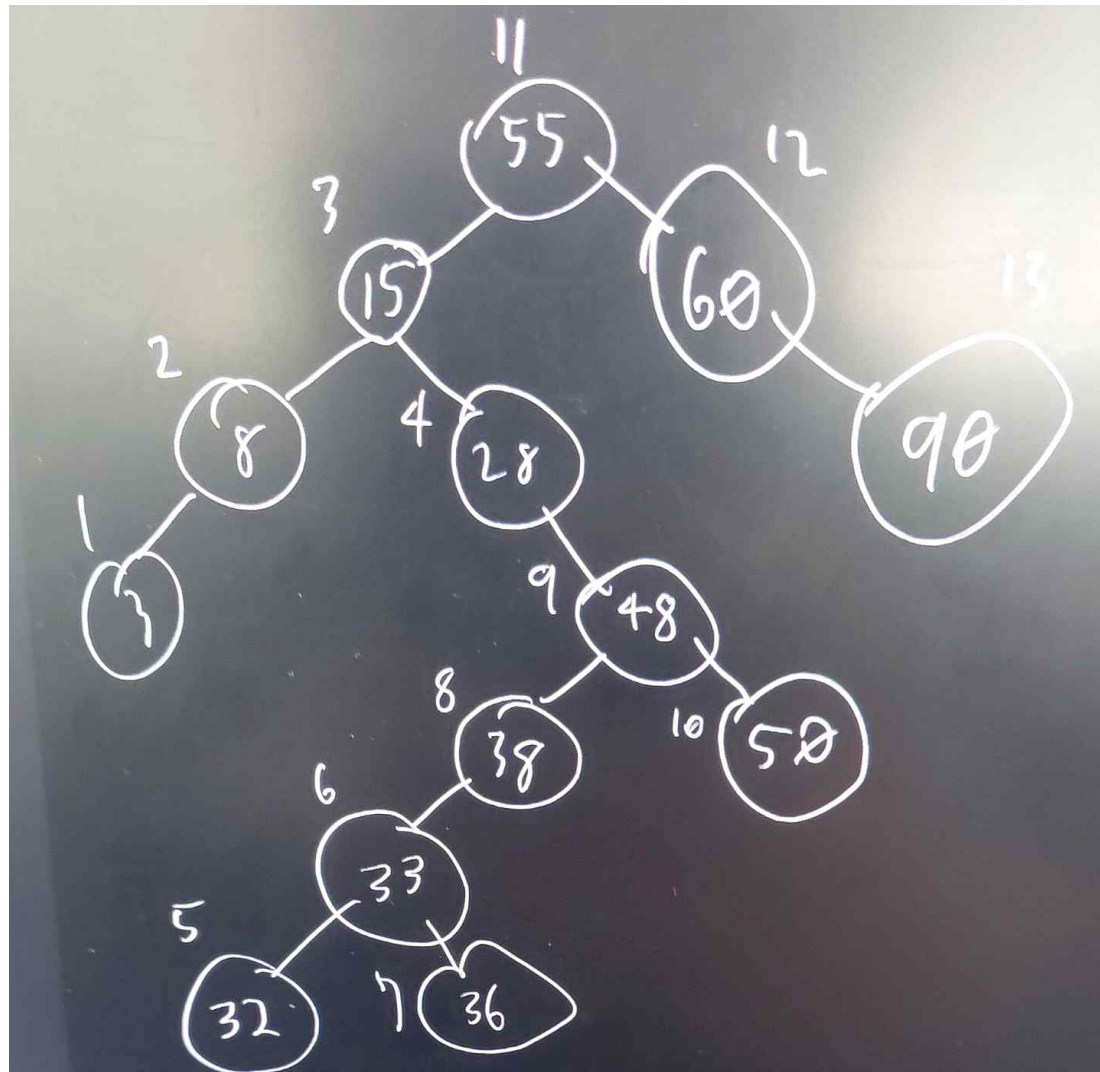
[문제2] BST에 대한 중위순회

## 1 BST에 대한 중위 순회 결과를 한 줄로 제시하기

3 → 8 → 15 → 28 → 32 → 33 → 36 → 38 → 48 → 50 → 55 → 60 → 90

중위 순회는 Left → Root → Right 순으로 이루어집니다.

그래서 순서대로 나타내면 위의 과정처럼 되고 이를 BST로 나타내면 아래와 같이 됩니다.



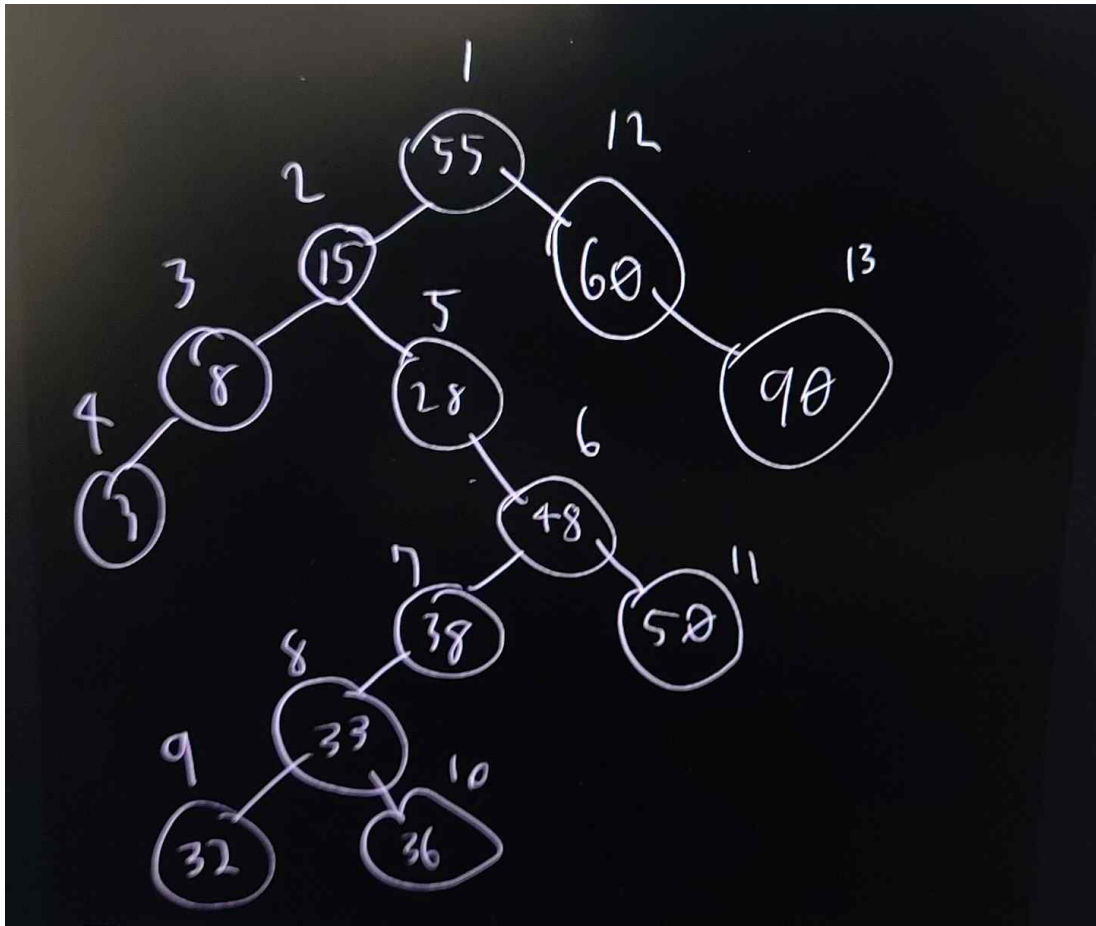
## 2 비어있는 BST 완성하기

55 → 15 → 8 → 3 → 28 → 48 → 38 → 33 → 32 → 36 → 50 → 60 → 90

BST를 완성하면 아래와 같은 형식을 취하게 됩니다.

노드 위에 있는 숫자는 삽입된 순서를 의미하며 이는 전위 순회와 같은 순서로 진행됩니다.

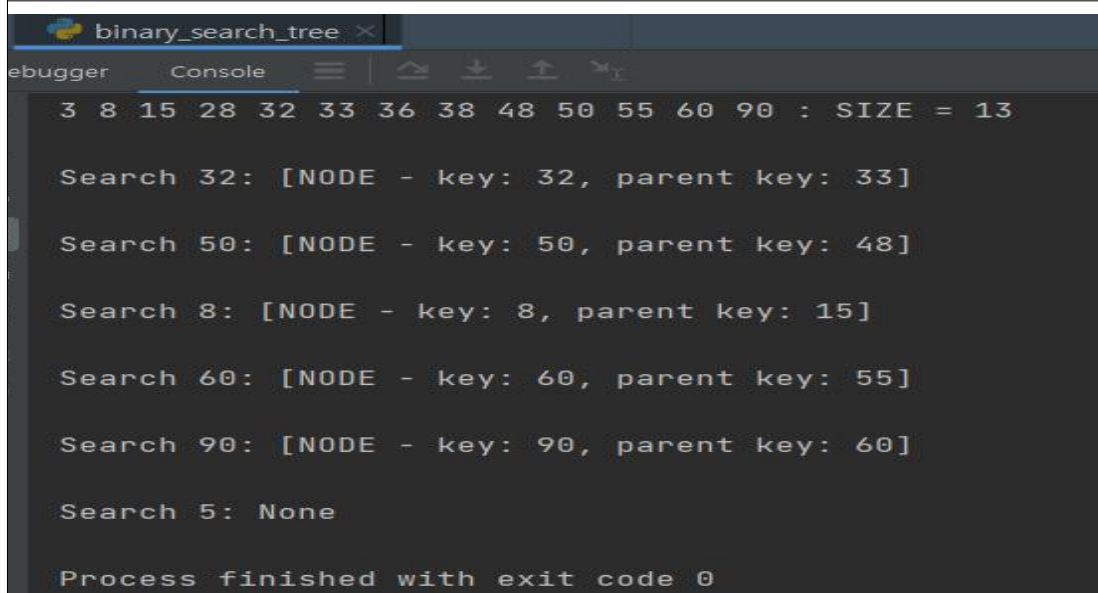
자세한 과정에 대한 그림은 아래에 첨부합니다.



### 3 [문제 1]를 통해 [문제2-1] 결과 검증하기

<실행과정>

```
if __name__ == "__main__":
    bst = BST()
# 2-1의 트리
# 아래 실행 결과는 2-1의 중위 순회 과정과 일치한다.
    bst.tree_insert(55)
    bst.tree_insert(15)
    bst.tree_insert(8)
    bst.tree_insert(3)
    bst.tree_insert(28)
    bst.tree_insert(48)
    bst.tree_insert(38)
    bst.tree_insert(33)
    bst.tree_insert(32)
    bst.tree_insert(36)
    bst.tree_insert(50)
    bst.tree_insert(60)
    bst.tree_insert(90)
    bst.print_bst()
# 32, 50, 8, 60, 90, 5를 검색했을 때 32, 50, 8, 60, 90이 각각 검색되고 5는 실패한다.
    print("\nSearch 32:", bst.tree_search(32))
    print("\nSearch 50:", bst.tree_search(50))
    print("\nSearch 8:", bst.tree_search(8))
    print("\nSearch 60:", bst.tree_search(60))
    print("\nSearch 90:", bst.tree_search(90))
    print("\nSearch 5:", bst.tree_search(5))
```



```
binary_search_tree x
ebugger Console
3 8 15 28 32 33 36 38 48 50 55 60 90 : SIZE = 13
Search 32: [NODE - key: 32, parent key: 33]
Search 50: [NODE - key: 50, parent key: 48]
Search 8: [NODE - key: 8, parent key: 15]
Search 60: [NODE - key: 60, parent key: 55]
Search 90: [NODE - key: 90, parent key: 60]
Search 5: None
Process finished with exit code 0
```

[문제3] BST에 대한 키 삽입

1 [알고리즘 6-4]에 입각하여 [문제2]의 BST에서 키 40,39,80 삽입 결과 제시하기

> 서술형으로 작성

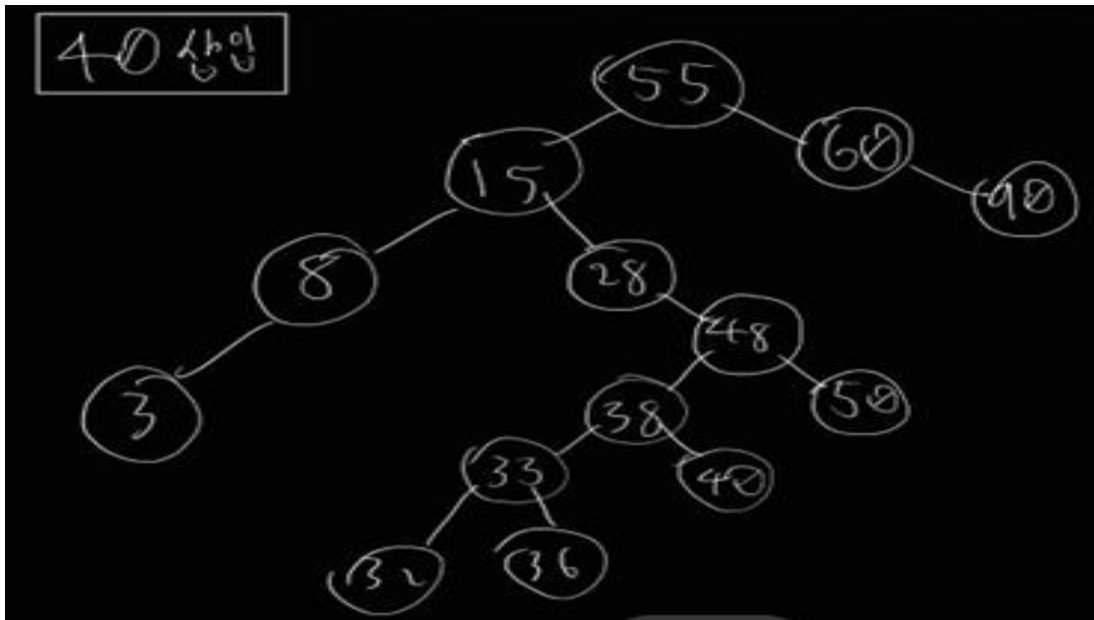
> 과정을 그리고 사진을 찍어 보고서에 첨부할 것

알고리즘 6-4	이진 검색 트리에서 삽입(비재귀 버전)
<p><i># 비재귀</i></p> <pre> treeInsert(<i>t</i>, <i>x</i>) ▷ <i>t</i> : 트리의 루트 노드 ▷ <i>x</i> : 삽입하고자 하는 키 {     <i>key</i>[<i>r</i>] ← <i>x</i>; <i>left</i>[<i>r</i>] ← NIL; <i>right</i>[<i>r</i>] ← NIL;     if (<i>t</i> = NIL) then <i>root</i> ← <i>r</i>;     else {         <i>p</i> ← NIL; <i>tmp</i> ← <i>t</i>;         while (<i>tmp</i> ≠ NIL) {             <i>p</i> ← <i>tmp</i>;             if (<i>x</i> &lt; <i>key</i>[<i>tmp</i>]) then <i>tmp</i> ← <i>left</i>[<i>tmp</i>];             else <i>tmp</i> ← <i>right</i>[<i>tmp</i>];         }         if (<i>x</i> &lt; <i>key</i>[<i>p</i>]) then <i>left</i>[<i>p</i>] ← <i>r</i>;         else <i>right</i>[<i>p</i>] ← <i>r</i>;     } }                 </pre> <p><i>루트노드</i> ▷ <i>r</i> : 새 노드</p> <p><i>parent ← tmp (current)</i></p>	

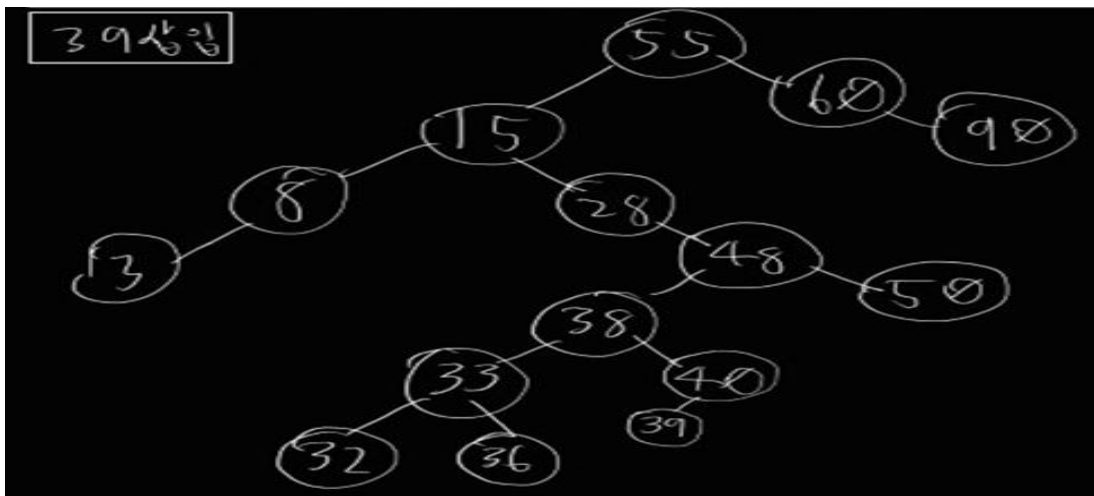
알고리즘 6-4은 키를 가진 노드가 트리를 순회하면서 삽입되는 자리를 잡는 과정을 나타낸 의사코드입니다. 만약 루트 노드가 비어있다면 키를 가진 노드가 루트에 들어가고 아닐 땐 빈 노드를 찾을 때까지 작은 경우 왼쪽 자식, 클 경우에는 오른쪽 자식으로 들어가는 식으로 트리 속을 순회하면서 자리를 잡아갑니다.

40을 가진 노드는 38을 가진 노드의 왼쪽 자식, 39를 가진 노드는 40을 가진 노드의 왼쪽 자식, 80을 가진 노드는 90을 가진 노드의 왼쪽 자식에 들어가게 되는데 자세한 과정은 다음의 그림과 같습니다.

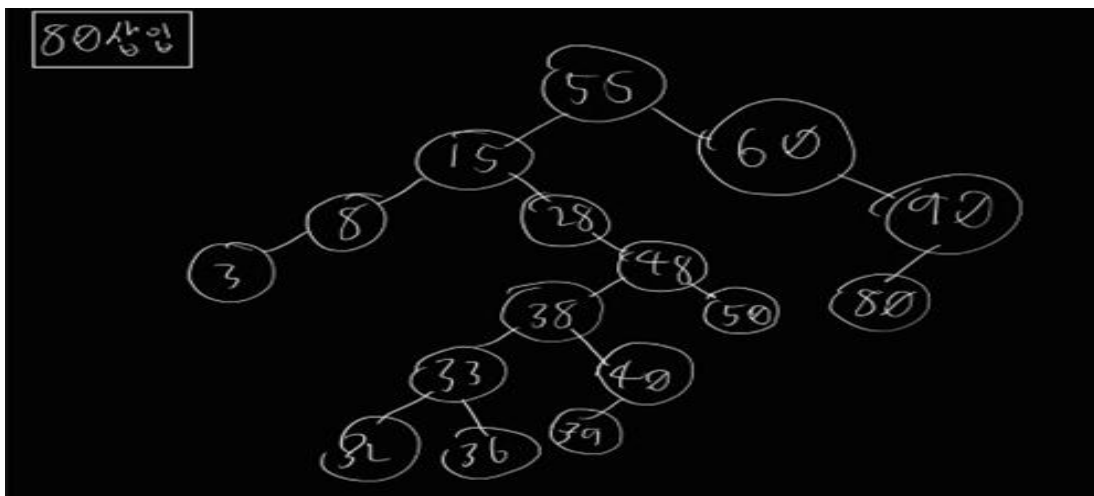
40삽입



39삽입



80삽입



## 2 위에서 제시된 키 노드들이 삽입된 직후의 BST에 대한 중위 순회 결과 제시하기

중위 순회를 한 결과는

40을 넣었을 때는

3 → 8 → 15 → 28 → 32 → 33 → 36 → 38 → 40 → 48 → 50 → 55 → 60 → 90

39를 넣었을 때는

3 → 8 → 15 → 28 → 32 → 33 → 36 → 38 → 39 → 40 → 48 → 50 → 55 → 60 → 90

80을 넣었을 때는

3 → 8 → 15 → 28 → 32 → 33 → 36 → 38 → 39 → 40 → 48 → 50 → 55 → 60 → 80 → 90

순과 같습니다.

## 3 [문제2-3] 이후 코드를 이용해 [문제3-1, 3-2] 검증하기

> 키 삽입될 때마다 출력된 중위 순회 결과 캡처화면 제시

40을 넣었을 때



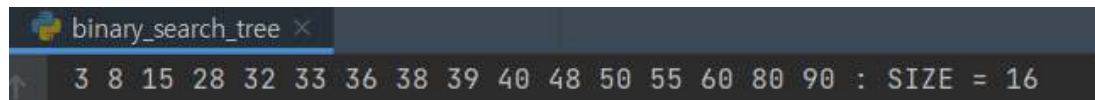
```
binary_search_tree ×  
↑ 3 8 15 28 32 33 36 38 40 48 50 55 60 90 : SIZE = 14
```

39를 넣었을 때



```
binary_search_tree ×  
↑ 3 8 15 28 32 33 36 38 39 40 48 50 55 60 90 : SIZE = 15
```

80를 넣었을 때



```
binary_search_tree ×  
↑ 3 8 15 28 32 33 36 38 39 40 48 50 55 60 80 90 : SIZE = 16
```

[문제4] BST에 대한 키 삽입과 중위 순회

1 [알고리즘 6-6]에 입각하여 [문제3]의 결과에 키 15 노드 삭제하기

> 서술형으로 작성

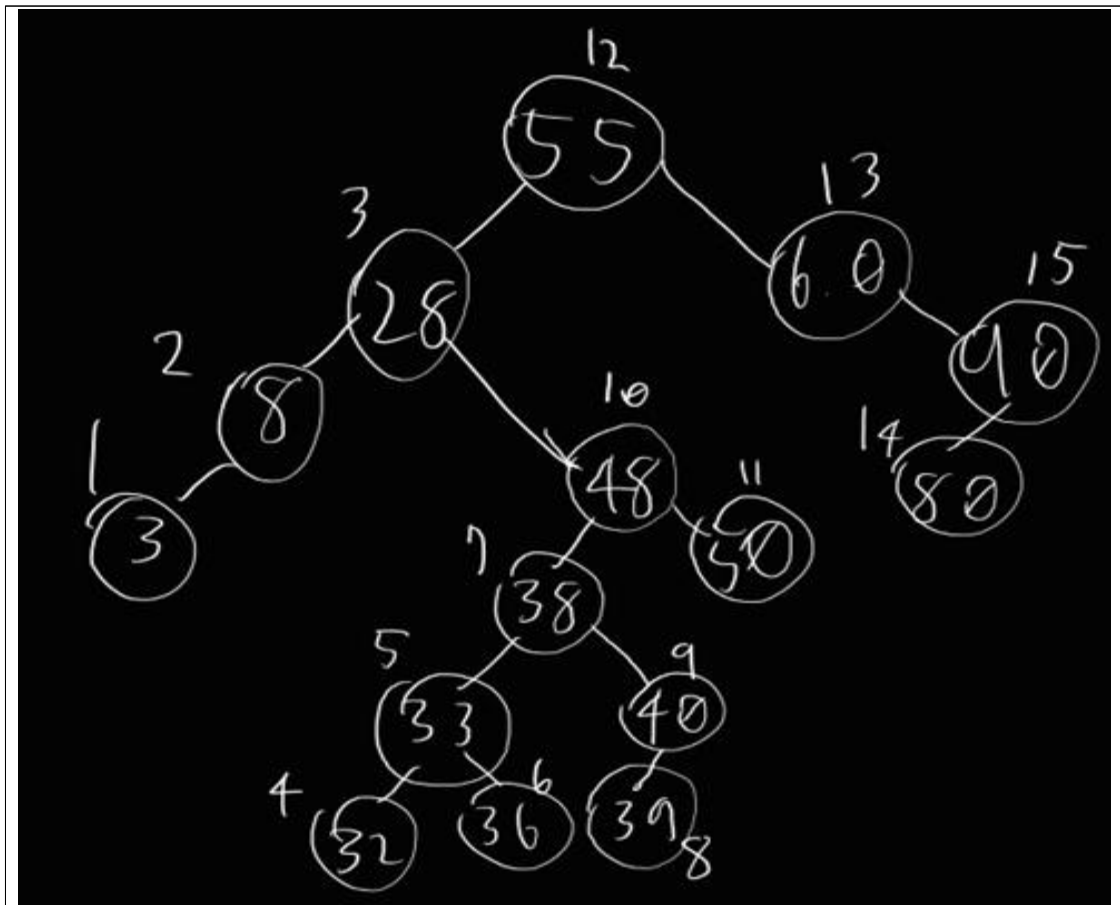
> 과정을 그리고 사진을 찍어 보고서에 첨부할 것

알고리즘 6-6	이진 검색 트리에서 삭제
<pre>treeDelete(<i>t</i>, <i>r</i>, <i>p</i>) ▷ <i>t</i>: 트리의 루트 노드 ▷ <i>r</i>: 삭제하고자 하는 노드, <i>p</i>: <i>r</i>의 부모 노드 {     ▷ <i>r</i>이 루트 노드인 경우     if (<i>r</i>=<i>t</i>) then <i>root</i> ← deleteNode(<i>t</i>);     ▷ <i>r</i>이 루트 노드가 아닌 경우     else if (<i>r</i>=left[<i>p</i>])         then left[<i>p</i>] ← deleteNode(<i>r</i>);      ▷ <i>r</i>이 <i>p</i>의 왼쪽 자식         else right[<i>p</i>] ← deleteNode(<i>r</i>);      ▷ <i>r</i>이 <i>p</i>의 오른쪽 자식     }     deleteNode(<i>r</i>)     {         if (left[<i>r</i>] = right[<i>r</i>] = NIL) then return NIL;      ▷ Case 1         else if (left[<i>r</i>] = NIL and right[<i>r</i>] ≠ NIL) then return right[<i>r</i>];      ▷ Case 2-1         else if (left[<i>r</i>] ≠ NIL and right[<i>r</i>] = NIL) then return left[<i>r</i>];      ▷ Case 2-2         else {      ▷ Case 3             <i>s</i> ← right[<i>r</i>];             while (left[<i>s</i>] ≠ NIL)                 {parent ← <i>s</i>; <i>s</i> ← left[<i>s</i>];}             key[<i>r</i>] ← key[<i>s</i>];             if (<i>s</i> = right[<i>r</i>]) then right[<i>r</i>] ← right[<i>s</i>];                 else left[parent] ← right[<i>s</i>];             return <i>r</i>;         }     } }</pre>	

알고리즘 6-6은 키를 가진 노드를 삭제한 후의 자리를 잡는 과정을 나타낸 의사코드입니다. Case1는 리프노드일 때 Case2는 자식노드가 하나일 때 Case3는 자식노드가 두 개일 때를 나타냅니다.

15를 가진 노드는 8과 28이라는 두 개의 자식노드를 갖는 Case3에 해당합니다. 여기서는 15 노드가 삭제될 때 15 노드의 오른쪽에 있는 노드중 최소원소노드에 해당되는 28 노드가 *s*로 설정됩니다.

자세한 중위 순회된 모습은 아래 사진과 같습니다.



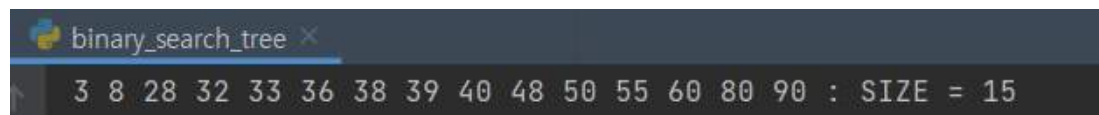
## 2 [문제4-1]에 제시된 중위 순회 결과 제시하기

3 → 8 → 28 → 32 → 33 → 36 → 38 → 39 → 40 → 48 → 50 → 55 → 60 → 80 → 90

## 3 [문제 3-3] 이후 코드를 이용해 [문제4-1, 4-2] 검증하기

> 키 삽입될 때마다 출력된 중위 순회 결과 캡처화면 제시

15 삭제 후의 캡처 화면





[문제5] B-Tree에 대한 문제 해결하기

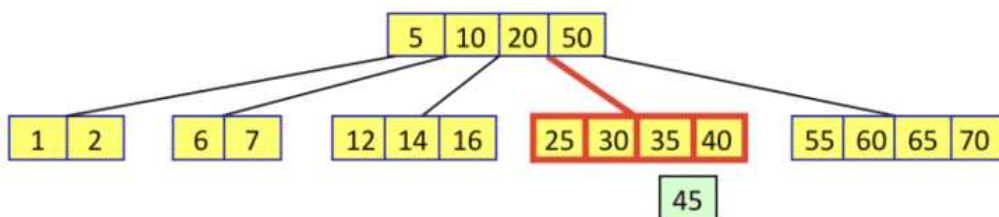
- 1 [알고리즘 6-7]에 입각하여 B-Tree[k=4]에서 키 45 노드 삽입하기
  - > 최종 결과 그림 제시
  - > 중간 과정 설명

**알고리즘 6-7** B-트리에서 삽입 스케치

```

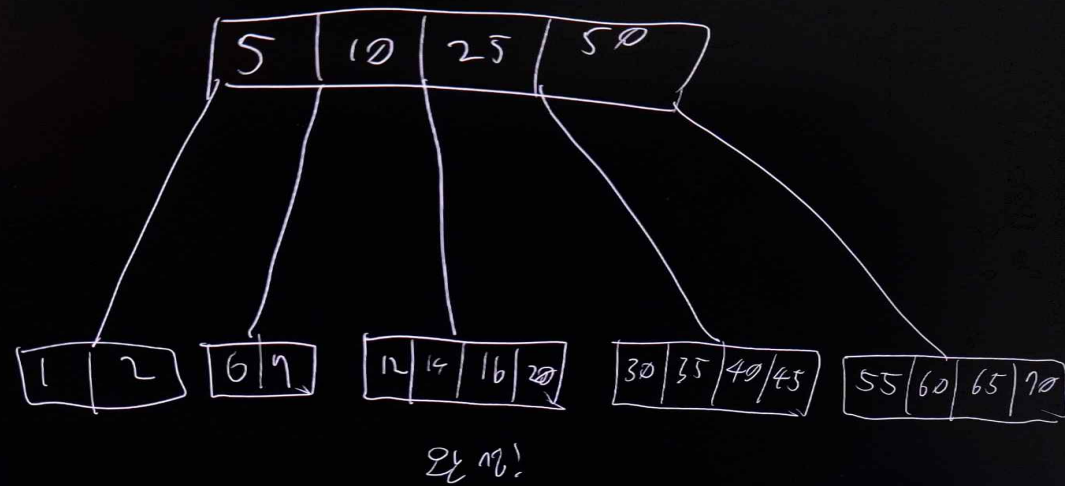
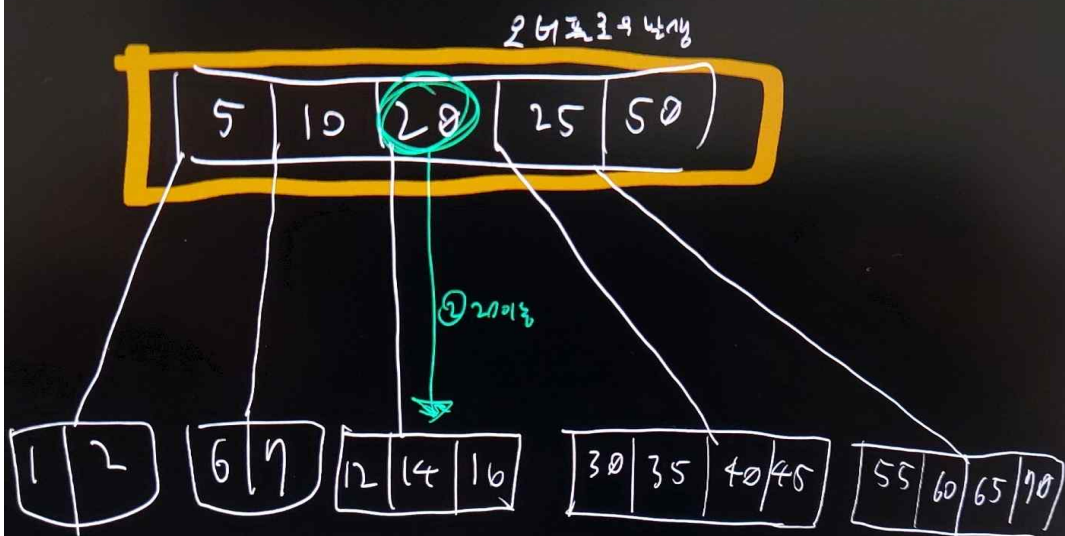
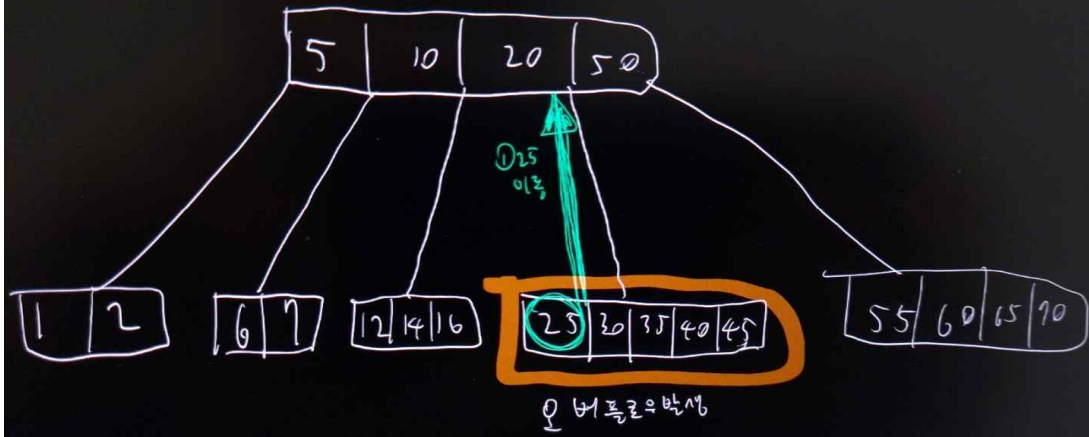
Sketch-BTreeInsert( $t, x$ )
▷  $t$ : 트리의 루트 노드
▷  $x$ : 삽입하고자 하는 키
{
     $x$ 를 삽입할 리프 노드  $r$ 을 찾는다;
     $x$ 를  $r$ 에 삽입한다;
    if ( $r$ 에 오버플로 발생) then clearOverflow( $r$ );
}
clearOverflow( $r$ )
{
    if ( $r$ 의 형제 노드 중 공간 여유가 있는 노드가 있음) then  $r$ 의 남은 키를 넘긴다;
    else {
         $r$ 을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
        if (부모 노드  $p$ 에 오버플로 발생) then clearOverflow( $p$ );
    }
}
    
```

*Handwritten notes:*  
 - In the first if statement: "1c4" (likely 1c4) and an arrow pointing to the condition.  
 - In the second if statement: "1c4" and "1c4" with arrows pointing to the condition and the recursive call respectively.  
 - At the bottom right: "1c4" and "1c4" with arrows pointing to the recursive call and the final closing brace respectively.



다음과 같은 상황일 때 B-Tree [k=4] 에서 45가 삽입되면 Sketch-BTreeInsert( $t, x$ )에서의 if문에서 k+1, 즉 크기가 5가 되어 오버플로우가 발생하게 되고 clearOverflow( $r$ )가 호출되어서 25, 30, 35, 40, 45 중 25가 부모 노드로 올라갑니다. 그런데 여기서 부모노드에서 또 오버플로우가 발생하며, 5, 10, 20, 25, 50 중 20이 자식 노드로 내려가 병합되는 방식으로 B-Tree가 만들어집니다.

자세한 과정은 아래 그림과 같습니다.



2 [알고리즘 6-8]에 입각하여 B-Tree[k=4]에서, 언더플로우가 발생한 키 7 노드 삭제 과정 및 결과 그림 제시하기

> 최종 결과 그림 제시

> 중간 과정 설명

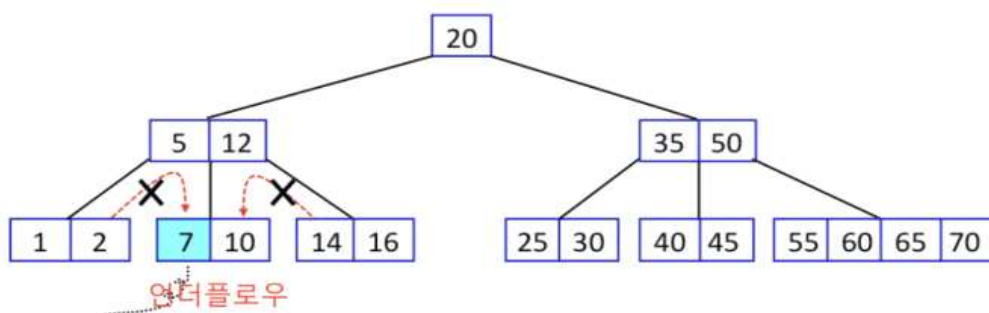
**알고리즘 6-8**      B-트리에서 삭제 스케치

```

Sketch-BTreeDelete( $t, x, v$ )
▷  $t$ : 트리의 루트 노드
▷  $x$ : 삭제하고자 하는 키,  $v$ :  $x$ 를 갖고 있는 노드
{
    if ( $v$ 가 리프 노드 아님) then {
         $x$ 의 직후 원소  $y$ 를 가진 리프 노드를 찾는다;
         $x$ 와  $y$ 를 맞바꾼다;
    }
    리프 노드에서  $x$ 를 제거하고 이 리프 노드를  $r$ 이라 한다;
    if ( $r$ 에서 언더플로 발생) then clearUnderflow( $r$ );
}
clearUnderflow( $r$ )
{
    if ( $r$ 의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
    then  $r$ 이 키를 넘겨받는다;
    else {
         $r$ 의 형제 노드와  $r$ 을 병합하고 부모 노드에서 키를 하나 받는다;
        if (부모 노드  $p$ 에 언더플로 발생) then clearUnderflow( $p$ );
    }
}

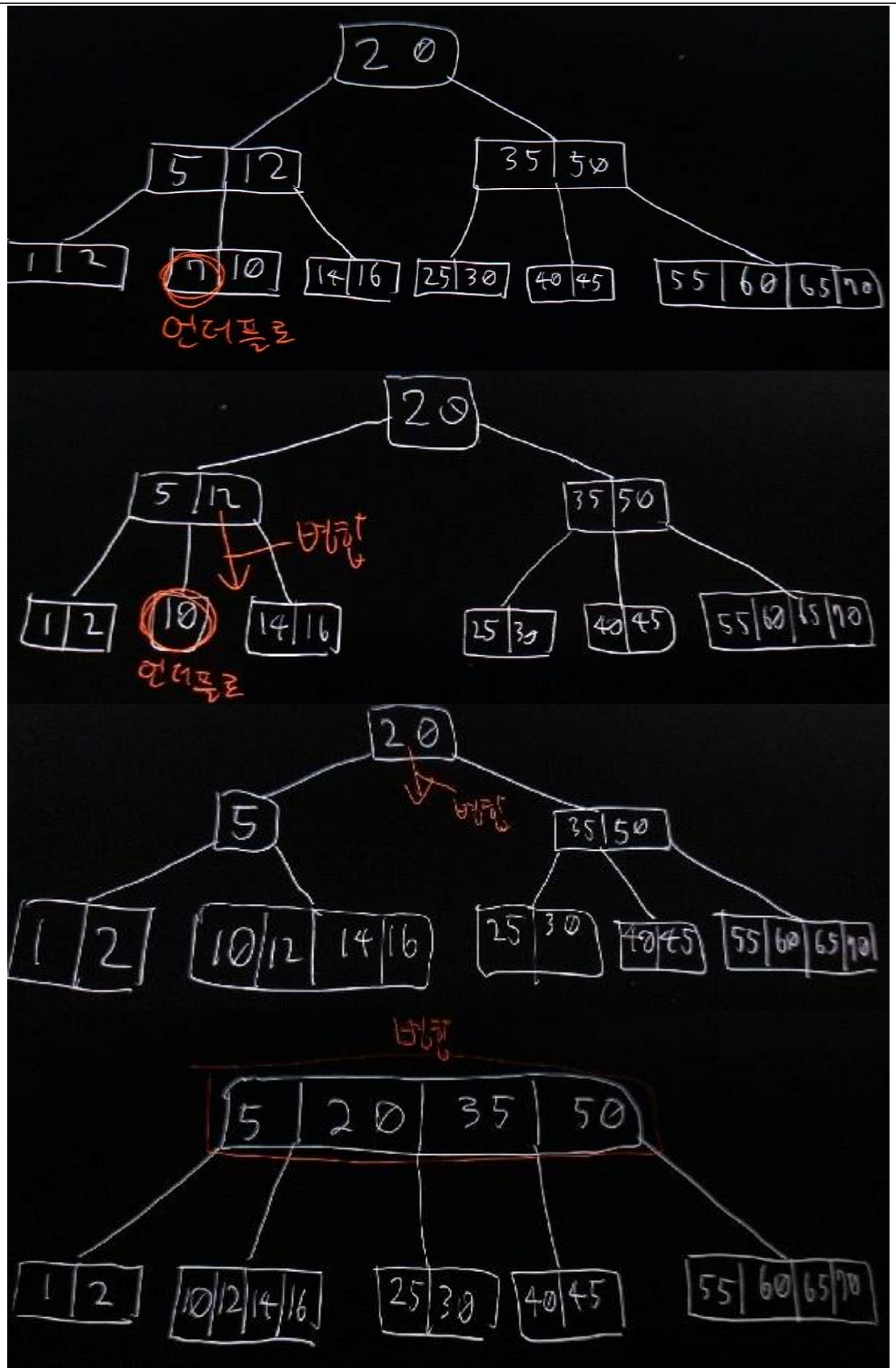
```

LINK@KOREATECH



7을 제거했을 때 형제 노드들에 여분의 노드가 존재하지 않아 부모 노드에서 키 12를 받아 병합을 먼저 수행하고, 그로 인해 생긴 부모 노드의 부족한 노드는 다시 부모 노드의 부모노드 20에서 가져와 병합하면서 구현합니다.

자세한 과정은 아래 그림과 같습니다.



[문제6]B-Tree의 임의 노드가 가질 수 있는 최대/최소키 개수 제시하기

▶ 조건 명세

- 1) 한 블록[페이지] 크기 : 35,448 byte
- 2) 키의 크기 32 byte
- 3) 페이지 번호 크기 : 4 byte

▶ 최대 키 개수 구하기

$4 + (4 + (32 + 4)) * n + 4 = 35448$   
 $40 * n + 8 = 35448$   
 $40 * n = 35440$   
 $n = 35440 / 40 = 886$   
 $\therefore$  최대 키 개수 : 886

▶ 최소 키 개수 구하기

Key의 범위는  $\lfloor k/2 \rfloor \sim k$  이므로  
 $\therefore$  최소 키 개수 :  $\lfloor 886/2 \rfloor = 443$

## 결론 : 고찰

이진 탐색 트리와 B-Tree를 알아보는 시간을 가질 수 있어서 좋았지만 시험기간이라는 한계 때문에 과제의 완성도를 높이지 못해서 개인적으로 아쉬웠습니다. 다음엔 더 많은 시간을 들여 질적인 완성도가 높은 과제를 하고 싶은 마음이 들었습니다.

감사합니다.