R E P O R T

실습 및 과제 3

과목명	알고리즘및실습
분반	2 분 반
교수	한 연 희
학번	2020136129
이름	최 수 연
제출일	2022년 4월 21일 목요일

목차

서론

본 과제에서는 Binary Search Tree에 대한 노드 삽입 및 삭제에 대한 함수를 구현함으로써 BST 함수에 대한 삽입 및 삭제 과정에 대해 이해할 수 있고, BST를 중위 순회를 사용하여 삽입하는 과정을 직접 풀어봄으로써 BST의 중위 순회에 대해 학습할 수 있다. 또한 기존에 구성되어있는 트리에 추가 노드를 삽입하는 과정을 직접 계산하고 설명함으로써 BST에 대해 보다 정확한 이해가 가능하다.

B-Tree에 대한 노드 삽입 및 삭제 연산 과정을 설명하고 직접 그림을 제시해보고, B-Tree에 대한 블록과 키, 페이지 번호의 크기가 정해져 있을 때 임의의 노드가 가질 수 있는 최대 키와 최소 키의 개수를 구하는 과정에 대해 풀이 과정을 생각해봄으로써 B-Tree에 대한 이해를 높일 수 있다.

본론

[문제 1] BST에 대한 Python 프로그램 작성

[1-1] binary_search_tree.py 분석

- binary_search_tree 코드 파일에는 Node와 BST 두 클래스가 선언되어 있고, main에서 코드를 실행하고 출력한다.
- Node 클래스에서는 __init__ 메소드를 통해 tree에 필요한 객체를 생성하고, __str__ 메소드를 통해 문자열을 반환하도록 한다.
- BST 클래스에서는 __init__ 메소드를 통해 root와 tree의 크기를 생성한다.
- 그리고 중위 순회하는 함수를 선언하여 main에 입력된 트리를 중위 순회로 하나씩 출력하고, 해당 트리의 크기를 출력한다.
- 그 후 차례대로 노드를 검색하고, 삽입하고, 삭제하는 함수를 수행한다. 이때 해당 함수들은 비재귀적 버전으로 구현된다.

- 여기서 **tree_search** 함수와 **tree_delete** 함수에 대해서 간단히 분석해보겠다.

- 노드 검색 함수 **tree_search** 함수에서는 root가 만약 없으면 None을 반환하고, 그렇지 않으면, root를 현재 노드로 두고, 현재 노드가 None이 아닐 때 while문을 돌린다.
- 이때, 만약 현재 노드가 찾고자 하는 키 값이면 그 노드를 반환하고, 키 값이 현재 노드보다 작으면 왼쪽, 크면 오른쪽 자식 노드로 이동한다.
- while문이 끝나도 해당 key 값을 찾지 못했으면 None을 반환한다.

- 노드 삭제에서 **tree_delete** 함수는 삭제하고자 하는 key 값을 노드 검색 함수를 통해 찾고, 삭제할 노드에 넣는다.
- 삭제할 노드가 없으면 함수를 끝내고, 그렇지 않으면 if문을 실행한다.
- 만약 삭제할 노드가 root일 경우, 다음 _delete_node 함수에 해당 삭제할 노드 값을 넣어 root로 넣어준다.
- 그렇지 않고 만약, 삭제할 노드가 부모의 왼쪽 자식의 값과 같다면, 삭제할 노드가 왼쪽 자식을 가리키게 한다.
- 그렇지 않고 만약, 삭제할 노드가 부모의 오른쪽 자식의 값과 같다면, 삭제할 노드가 오른쪽 자식을 가리키게 한다.
- 그리고 마지막으로 size 값을 하나 줄인다.

- main에서는 트리에 노드를 삽입하고 삭제하는 함수를 직접 호출하여 결과를 출력한다.

[1-2] 노드 삽입 함수

```
def tree_insert(self, key=None):
        new_node = Node(key=key)

        if self.root is None:
            self.root = new_node
        else:
            # 새로운 노드를 위한 올바른 위치까지 순회하여 새로운 노드의 부모 설정
            current_node = self.root
            parent_node_for_new_node = None

            ###################################
            while current_node is not None:
              parent_node_for_new_node = current_node
              if key < current_node.key:
                current_node = current_node,left
              else:
                current_node = current_node.right
            if key < parent_node_for_new_node.key:
              parent_node_for_new_node.left = new_node
            else:
              parent_node_for_new_node.right = new_node

            ###################################

            # 새로운 노드의 부모 노드 설정
            new_node.parent = parent_node_for_new_node

        self.size += 1
```

- 노드 삽입 함수에서는 인자로 받은 key 값을 새 노드로 선언하고, root가 없을 때 새 노드를 root 노드로 지정한다.
- 만약 그렇지 않으면, root를 현재 노드로 두고, 새 노드의 부모를 지정할 변수를 하나 선언 한다,
- 현재 노드가 None이 아닐 때 while문을 수행한다.
- 해당 while 문에서 현재 노드를 새 노드의 부모노드로 지정하고 if문을 실행한다.
- 만약 찾고자 하는 key 값 즉, 새 노드의 key 값이 현재 노드의 key 값보다 작으면, 현재 노드의 왼쪽으로 옮긴다.
- 그렇지 않고 찾고자 하는 key 값 즉, 새 노드의 key 값이 현재 노드의 key 값보다 크면, 현재 노드의 오른쪽으로 옮긴다.
- while문을 다 수행한 뒤 if문을 한 번 더 실행한다.
- 만약 key값이 새 노드의 부모 노드 key 값보다 작으면 왼쪽, 크면 오른쪽으로 새 노드 위치를 지정한다.
- 그리고 새 노드의 부모를 설정한 후, tree의 size를 하나 더한다.

[1-3] 노드 삭제 함수

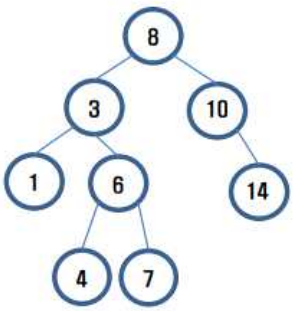
```
def _delete_node(self, node_to_be_deleted):
    if node_to_be_deleted.left is None and node_to_be_deleted.right is None:
        return None # Case 1
    elif node_to_be_deleted.left is None and node_to_be_deleted.right is not None:
        return node_to_be_deleted.right # Case 2-1
    elif node_to_be_deleted.left is not None and node_to_be_deleted.right is None:
        return node_to_be_deleted.left # Case 2-2
    else: # Case 3
        # 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
```

- tree_delete 함수는 [1-1]에서 설명하였으므로 생략한다.
- _delete_node 함수에서는 삭제할 노드를 인자로 받고, 만약 삭제할 노드의 자식이 모두 없으면 None, 삭제할 노드의 자식이 오른쪽만 있으면, 오른쪽 자식을, 삭제할 노드의 자식이 왼쪽만 있으면 왼쪽 자식을 반환한다.
- 위 경우가 모두 해당되지 않을 경우 다음을 수행한다.
- 먼저 삭제할 노드의 오른쪽 자식을 smallest_node로 설정한다.
- 그리고 smallest_node의 부모는 삭제할 노드 자신이 되도록 선언한다.
- smallest_node의 왼쪽 자식이 None이 아니면 while문을 실행한다.
- while 문 실행 시, smallest_node를 parent_of_smallest_node로 설정하고, smallest_node의 왼쪽 자식을 smallest_node로 변경한다.
- 그리고 while 문을 빠져나와 smallest_node의 key 값을 삭제할 노드의 key로 넣는다.
- 만약, smallest_node가 삭제할 노드의 오른쪽 자식과 같으면, smallest_node의 오른쪽 자식을 삭제할 노드의 오른쪽 자식으로 넣는다.
- 그렇지 않으면, smallest_node의 오른쪽 자식을 부모의 왼쪽 자식으로 넣는다.
- smallest_node를 삭제하고 삭제할 노드를 반환한다.

[1-4] 수행 결과

```
1 3 4 6 7 8 10 14 : SIZE = 8

Search 10: [NODE - key: 10, parent key: 8]

Search 100: None

Delete 10
1 3 4 6 7 8 14 : SIZE = 7

Delete 10
1 3 4 6 7 8 14 : SIZE = 7

Delete 3
1 4 6 7 8 14 : SIZE = 6

Delete 4
1 6 7 8 14 : SIZE = 5

Delete 8
1 6 7 14 : SIZE = 4
```

[1-5] BST에 대한 기술적 고찰



오른쪽 트리의 내부 경로 길이 IPL은 1+2+2+3+3+3+4+4= 22이다.

[정리 6-1]에 의해 키의 총 수가 n개인 모든 이진 검색 트리의 평균 IPL은 O(nlogn)이 된다.

이상적인 트리의 높이는 평균 키 검색 수 또는, IPL/키의 개수가 된다. 이때 IPL/키의 개수의 점근적 복잡도는 $O(nlogn)/n$이다.

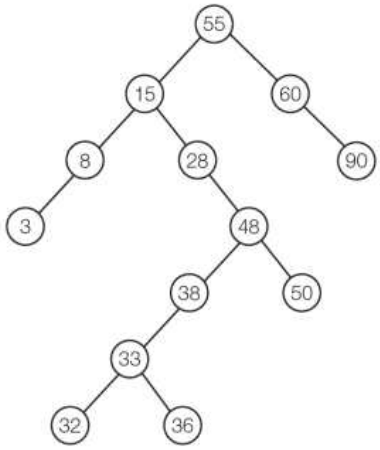
BST의 검색 알고리즘의 경우 트리 높이에 따라 점근적 복잡도가 $O(\log n) \sim O(n)$이 된다.

평균의 경우 BST 삽입 알고리즘의 점근적 복잡도는 $\theta(\log n)$이고, 최악의 경우 균형이 맞지 않는 BST가 만들어지므로 $\theta(n)$이 된다.

BST의 삭제 알고리즘의 경우 [1-3]의 Case 1은 $\theta(1)$, Case 2도 $\theta(1)$이고, Case 3에서는 최악의 경우 트리 전체 높이에 대해 while 루프문을 수행하기 때문에, 트리의 높이에 따라 $O(\log n) \sim O(n)$이 된다.

[문제 2]

[2-1] 오른쪽 BST에 대한 중위 순회 결과를 한 줄로 제시하시오



답) 3 -> 8 -> 15 -> 28 -> 32 -> 33 -> 36 -> 38 -> 48 -> 50 -> 55 -> 60 -> 90

[2-2] 아무런 노드도 존재하지 않는 비어있는 BST에서 오른쪽 BST 를 만들기 위하여 처음에 키 55 노드를 삽입하면서 구성한다고 가정하고, 순차적으로 삽입되는 노드의 키값을 순서대로 나열하시오.

답) 55 -> 15 -> 60 -> 8 -> 28 -> 90 -> 3 -> 48 -> 38 -> 50 -> 33 -> 32 -> 36

[2-3] [문제 1]에서 작성한 코드의 메인파트를 처음부터 다시 작성하여 (즉, bst.tree_insert(55) 등을 순차적으로 다시 작성) 위와 같은 BST를 구성하고 bst.print_bst()를 통해 BST를 출력한 결과가 [2-1]에서 제시한 중위순회 결과와 동일한지 비교하시오.

메인파트 재작성 코드

```
if __name__ == "__main__":
    bst = BST()

    bst.tree_insert(55)
    bst.tree_insert(15)
    bst.tree_insert(60)
    bst.tree_insert(8)
    bst.tree_insert(28)
    bst.tree_insert(90)
    bst.tree_insert(3)
    bst.tree_insert(48)
    bst.tree_insert(38)
    bst.tree_insert(50)
    bst.tree_insert(33)
    bst.tree_insert(32)
    bst.tree_insert(36)

    bst.print_bst()
```

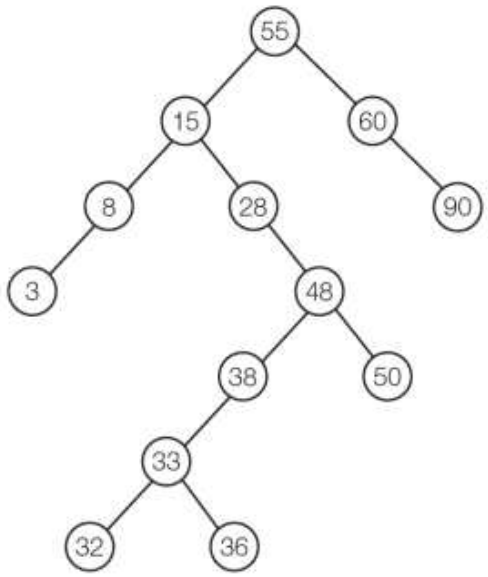
수행 결과

```
3 8 15 28 32 33 36 38 48 50 55 60 90 : SIZE = 13
```

수행 결과, [2-1]에서 제시한 중위순회 결과와 동일하게 나왔다.

[문제 3]

[3-1] 앞선 [문제 2]에서 구성한 아래의 BST에서 키 40, 키 39, 키 80을 삽입하는 과정을 [알고리즘 6-4]에 입각하여 제시하시오.



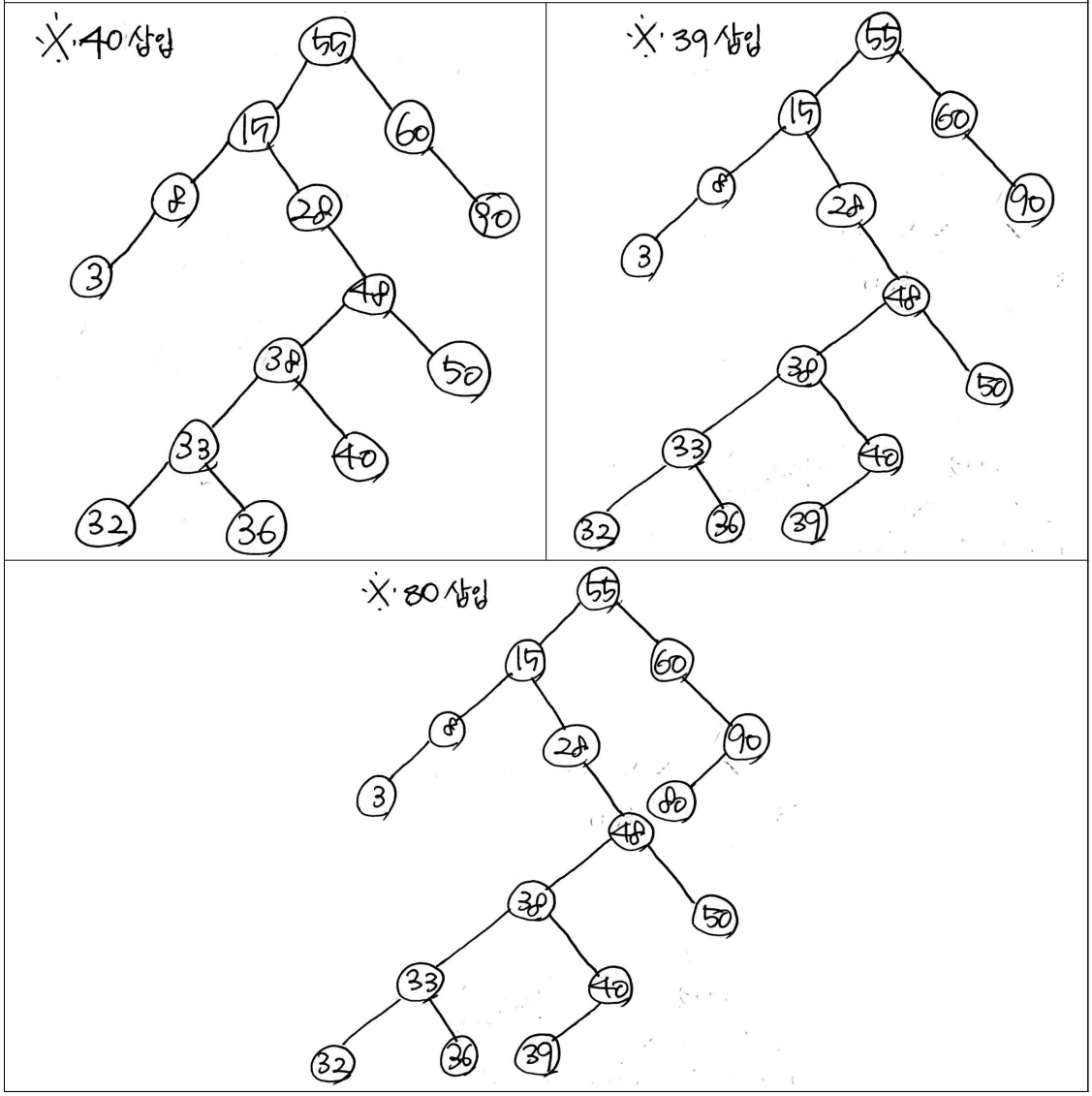
[알고리즘 6-4]에서는 노드를 삽입할 때, root가 없으면 노드를 root로 설정하고, 아니면 노드를 하나씩 비교해가면서 작으면 왼쪽, 크면 오른쪽으로 순회하며 새로운 노드의 부모를 설정한다. 이 알고리즘에 입각하여 40, 39, 80을 삽입하면 다음과 같다.

40 삽입) 40을 삽입할 때에는 root부터 시작하여, 55보다 작으므로 왼쪽, 15보다 크므로 오른쪽, 28보다 크므로 오른쪽, 48보다 작으므로 왼쪽, 38보다 크므로 오른쪽인데 더 이상 노드가 없으므로 해당 부분에 40을 삽입한다.

39 삽입) 39를 삽입할 때에는 root부터 시작하여, 55보다 작으므로 왼쪽, 15보다 크므로 오른쪽, 28보다 크므로 오른쪽, 48보다 작으므로 왼쪽, 38보다 크므로 오른쪽, 40보다 작으므로 왼쪽으로 삽입된다.

80 삽입) 80을 삽입할 때에는 root부터 시작하여, 55보다 크므로 오른쪽, 60보다 크므로 오른쪽, 90보다는 작으므로 왼쪽으로 가서 삽입된다.

삽입 과정 그림



[3-2] [문제 3-1]에서 제시된 3개의 키 노드에 대하여 순차적으로 삽입된 직후의 BST에 대한 중위 순회 결과를 한 줄씩 차례로 제시하시오.

40 삽입) 3 -> 8 -> 15 -> 28 -> 32 -> 33 -> 36 -> 38 -> **40** -> 48 -> 50 -> 55 -> 60 -> 90

39 삽입) 3 -> 8 -> 15 -> 28 -> 32 -> 33 -> 36 -> 38 -> **39** -> 40 -> 48 -> 50 -> 55 -> 60 -> 90

80 삽입) 3 -> 8 -> 15 -> 28 -> 32 -> 33 -> 36 -> 38 -> 39 -> 40 -> 48 -> 50 -> 55 -> 60 -> **80** -> 90

[3-3] 앞선 [문제 1]에서 작성한 코드 및 [문제 2-3]에서 작성한 메인파트 이후에, 직접 추가적으로 [문제 3-1]에서 제시된 3개의 키 노드에 대한 삽입 코드를 넣어서 위 [문제 3-1] 및 [문제 3-2]에서 제시한 답안과 일치하는지 조사하시오.

메인파트 재작성 코드

```
if __name__ == "__main__":
    bst = BST()

    bst.tree_insert(55)
    bst.tree_insert(15)
    bst.tree_insert(60)
    bst.tree_insert(8)
    bst.tree_insert(28)
    bst.tree_insert(90)
    bst.tree_insert(3)
    bst.tree_insert(48)
    bst.tree_insert(38)
    bst.tree_insert(50)
    bst.tree_insert(33)
    bst.tree_insert(32)
    bst.tree_insert(36)

    bst.print_bst()

    print("\nInsert 40")
    bst.tree_insert(40)
    bst.print_bst()

    print("\nInsert 39")
    bst.tree_insert(39)
    bst.print_bst()

    print("\nInsert 80")
    bst.tree_insert(80)
    bst.print_bst()
```

수행 결과

```
3 8 15 28 32 33 36 38 48 50 55 60 90 : SIZE = 13

Insert 40
3 8 15 28 32 33 36 38 40 48 50 55 60 90 : SIZE = 14

Insert 39
3 8 15 28 32 33 36 38 39 40 48 50 55 60 90 : SIZE = 15

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

수행 결과, [3-1]과 [3-2]에서 제시한 답안과 동일하게 나왔다.

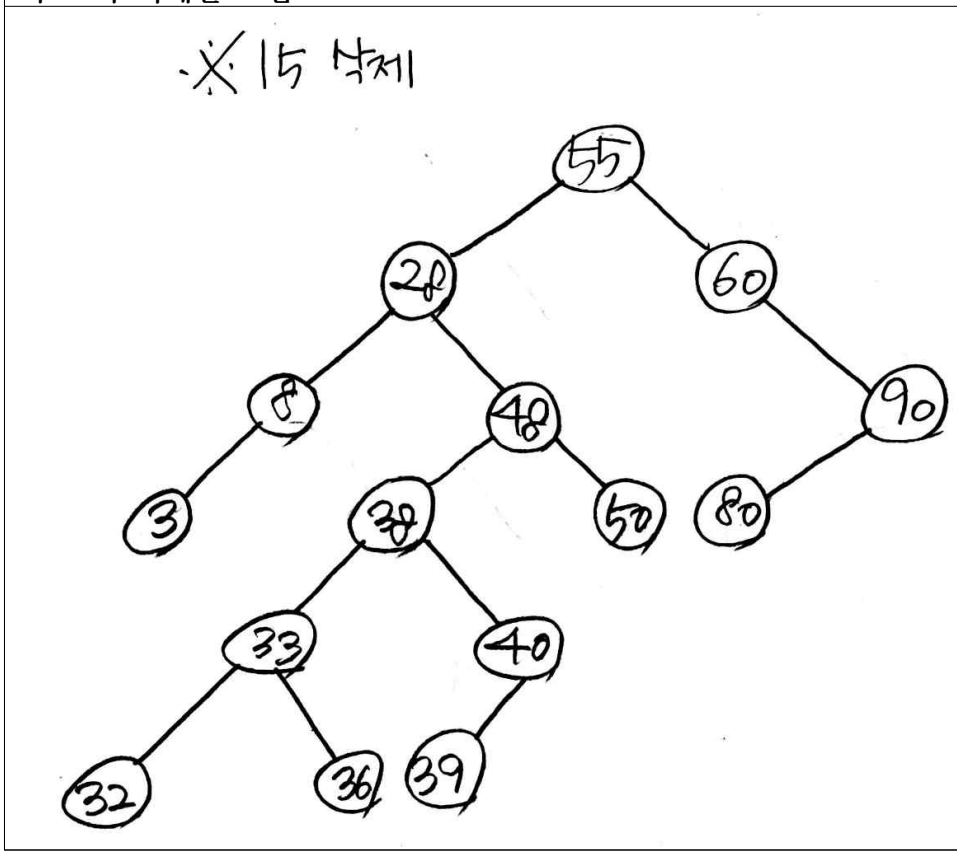
[문제 4]

[4-1] 앞선 [문제 3]에서 구성한 오른쪽 BST에서 키 40, 키 39, 키 80을 삽입한 이후의 BST에 대해 키 15 노드를 [알고리즘 6-6]에 입각하여 제시하시오.

[알고리즘 6-6]에서는 노드를 삭제할 때, 삭제할 노드가 root 노드인 경우 해당 노드를 삭제하고, root 노드가 아닐 경우, 노드가 왼쪽 자식이랑 같으면 왼쪽 자식, 오른쪽 자식이랑 같으면 오른쪽 자식을 가리키고 deleteNode 함수를 실행한다. 해당 함수는 if문을 사용하여 Case 1, Case 2, Case 3에 따라 수행 방법이 달라진다. 만약 자식이 모두 없으면 None을 반환하고, 오른쪽 자식만 있을 경우 오른쪽 자식, 왼쪽 자식만 있을 경우 왼쪽 자식을 반환한다. 만약 모두 아닐 경우, 삭제할 노드의 자식들 중 가장 작은 값을 구해 교환하는 방식으로 진행하는데, 먼저 오른쪽 자식을 가장 작은 값이라고 정해두고 while 문을 실행하여 왼쪽 자식이 없을 때까지 왼쪽으로 계속 내려간다. 그리고 해당 값을 삭제할 노드와 바꾼 후, if문을 실행하여 가장 작은 값으로 지정했던 노드의 자식들을 이진 검색 트리 특성에 맞게 다시 정리한다. 따라서 아래 그림에서 키 15노드를 삭제하는 과정은 다음과 같다.

키 15는 양쪽 자식이 모두 있으므로 Case 3을 수행한다. 이때 15의 오른쪽 자식노드의 자식노드 중에 가장 작은 값을 찾아서 교환해야 하는데, 15의 오른쪽 자식노드는 28이고, 28의 왼쪽 자식이 없기 때문에 28이 가장 작은 값이 된다. 따라서 키 28을 키 15 노드의 자리에 넣고, 28은 오른쪽 자식들만 있으므로 28의 바로 오른쪽 자식인 키 48 노드를 기존 15가 있던 부분의 오른쪽 자식 노드로 삽입한다.

키 15가 삭제된 그림



[4-2] [문제 4-1]에서 제시된 BST에 대한 중위 순회 결과를 제시하시오.

15 삭제) 3 -> 8 -> 28 -> 32 -> 33 -> 36 -> 38 -> 39 -> 40 -> 48 -> 50 -> 55 -> 60 -> 80 -> 90

[4-3] 앞선 [문제 1]에서 작성한 코드 및 [문제 3-3]에서 작성한 메인파트 이후에, 직접 추가적으로 [문제 4-1]에서 제시된 1개의 키 노드에 대한 삭제 코드를 넣어서 위 [문제 4-1] 및 [문제 4-2]에서 제시한 답안과 일치하는지 조사하시오.

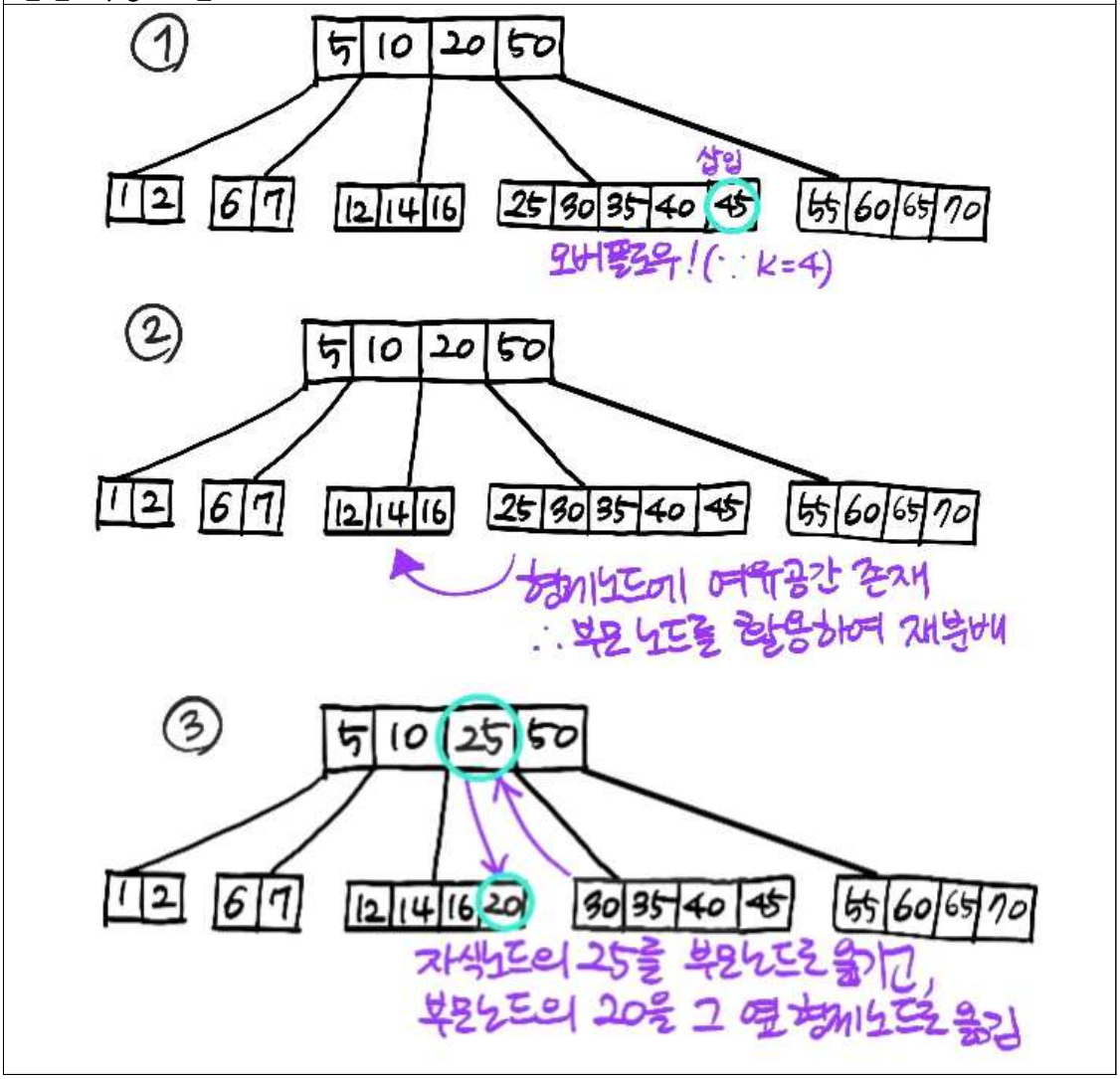
수행 결과	
3 8 15 28 32 33 36 38 48 50 55 60 90 : SIZE = 13 Insert 40 3 8 15 28 32 33 36 38 40 48 50 55 60 90 : SIZE = 14 Insert 39 3 8 15 28 32 33 36 38 39 40 48 50 55 60 90 : SIZE = 15 Insert 80 3 8 15 28 32 33 36 38 39 40 48 50 55 60 80 90 : SIZE = 16 Delete 15 3 8 28 32 33 36 38 39 40 48 50 55 60 80 90 : SIZE = 15	수행 결과, [4-1]과 [4-2]에서 제시한 답안과 동일하게 나왔다.

[문제 5] B-Tree에 대한 다음 문제를 해결하시오.

[5-1] 다음 그림에 주어진 B-Tree(k=4)에서 새롭게 키 45를 지닌 노드를 삽입하려고 한다. 교재에서 주어진 [알고리즘 6-7]에 입각하여 삽입하는 과정을 설명하고, 최종 결과 그림을 제시하시오.

[알고리즘 6-7]에 입각하여 먼저 삽입하고자 하는 키 45를 삽입할 리프노드를 찾는다. 그리고 세 번째 자식노드의 마지막 부분에 45가 삽입한다. 그렇게 되면 k는 4인데, 해당 노드가 총 5개의 키를 갖게 되므로 오버플로우가 발생한다. 따라서 형제노드에서 먼저 여유공간이 있는지 체크해야 한다. 왼쪽의 12, 14, 16 키를 가지고 있는 노드는 여유 공간이 아직 하나 존재한다. 따라서 부모노드를 활용해 재분배한다. 이때 25를 부모노드로 옮기고 부모노드의 20을 옆 형제노드로 내린다. 그렇게 되면 삽입이 정상적으로 완료된다.

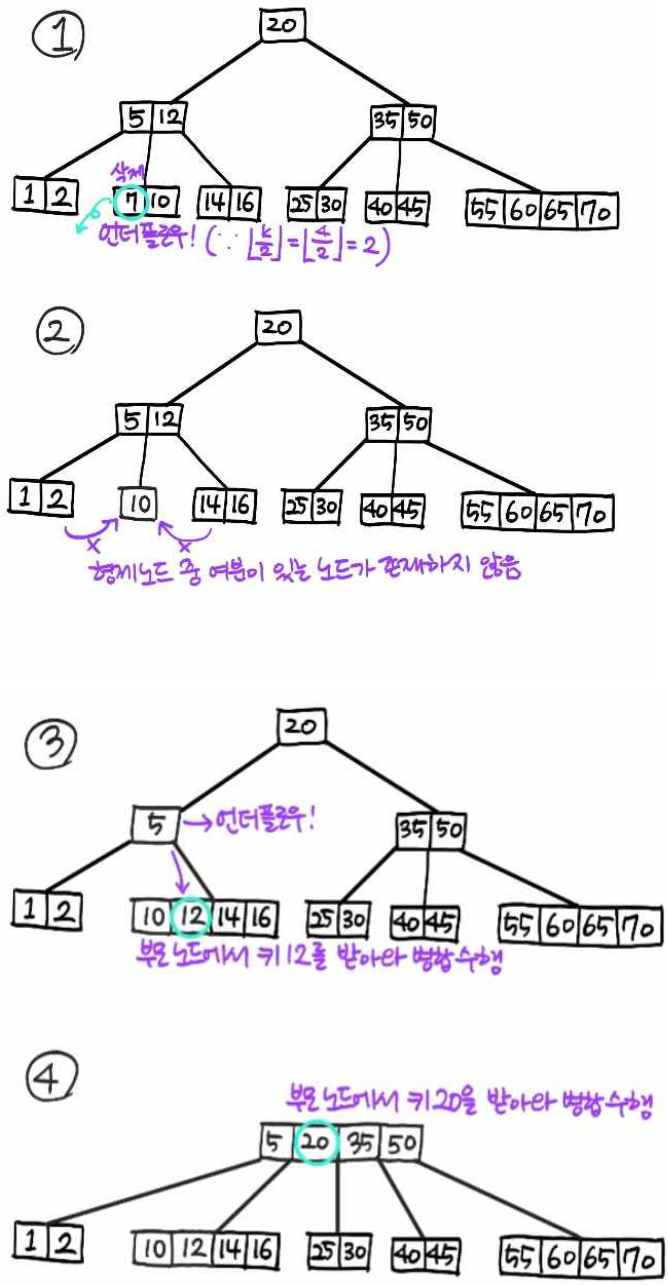
삽입 과정 그림



[5-2] 다음 그림에 주어진 B-Tree(k=4)에서 키 7을 지닌 노드를 삭제하려고 한다. 교재에서 주어진 [알고리즘 6-8]에 입각하여 키 7을 지닌 노드가 Underflow가 발생하고 있다. 이후부터 삭제하는 과정을 설명하고, 최종적으로 키 7을 지닌 노드를 삭제한 결과 그림을 제시하시오.

[알고리즘 6-8]에 입각하여, 먼저 삭제할 키를 가진 노드가 리프노드인지 확인한다. 키 7을 지닌 노드는 리프노드이므로 해당 노드를 삭제한다. 이때 ⌊k/2⌋=⌊4/2⌋= 2이므로 7가 삭제되면 10 하나만 남아 언더플로우가 발생하는데, 이때 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 존재하지 않는다. 따라서 부모노드에서 키를 하나 받아 옆 형제노드와 병합한다. 그런데 이때 부모노드에서 키 하나 받아오는 과정에서 부모노드의 12가 내려오면서 키 5 하나만 남아 부모노드에서도 언더플로우가 발생했다. 따라서 키 5를 지닌 노드의 부모노드 키 하나 20을 받아와 다시 병합수행한다.

삭제 과정 그림



[문제 6]

B-트리의

1) 한 블록(페이지) 크기가 35,448바이트이고,

2) 키의 크기가 32바이트이고,

3) 페이지 번호 크기가 4바이트일 때,

임의의 노드가 가질 수 있는 최대 키 개수와 최소 키 개수는

어떻게 되는지 풀이 과정과 함께 제시하시오.

최대 키 개수
최대 키 개수를 x라고 하면, 4 + (4 + (32 + 4)) * x + 4 = 35,448 x = 886 따라서, **최대 키 개수**는 **886개**이다.

최소 키 개수
최대 키 개수가 886개 이고, 만약 최대 키 개수가 k라고 했을 때, 최소 키 개수는 ⌊k/2⌋이므로, 최대 키 개수를 대입하면 다음과 같다. 최소 키 개수가 y이면, y=⌊886/2⌋= 443 따라서, **최소 키 개수**는 **443개**이다.

결론

이번 과제 덕분에 시험공부 하는 데 도움이 많이 되었던 것 같다. 이진 검색 트리와 B-트리가 어려운 내용은 아닌 것 같은데 생각보다 삽입, 삭제 과정이 체계적이어서 생각할게 많았다. 계속 트리에 대한 알고리즘을 외워놓고 나중에 필요할 때 써먹을 수 있었으면 좋겠다. 그리고 교수님께서 과제 기간 하루 늘려주신 덕분에 교양 수업 잘 끝내고 과제할 수 있어서 너무 감사했다.