B+ Tree wiki

이름: 권도현

학번: 2023065350
학과: 컴퓨터소프트웨어학부

실행 환경 / 언어: Window Powershell / Python

1. 기본 구조

가장 먼저, Tree를 저장할 **index 파일의 형식**은 다음과 같다.

1. 파일의 가장 첫 줄은 **b값이 적혀있다.**
2. 첫 줄을 제외한 각 줄은 **Tree에 들어있는 한 Node**를 나타낸다.
3. 한 줄, 각 Node의 형식은 다음과 같다.
    a. **node_id | is_leaf | keys | childs | next_leaf_id**
    b. `|` 를 기준으로 각 field가 구별되며 띄어쓰기는 없다.
    c. keys와 childs 속 각 key, child는 띄어쓰기 없이 `,` 로 구분되도록 한다.

Tree를 관리하기 위한 Tree Class와 각 node를 관리하기 위한 Node Class를 기본으로 한다.

Tree

`self.nodes`는 Tree내의 각 Node 객체를 담는 List이다.

`self.b`를 통해 각 Node의 Maximum을 지정할 수 있도록 한다.

`self.load_node()` 를 통해 index 파일에서 각 줄을 읽고, Node 구조체 생성자를 호출한 뒤 객체를 `self.nodes` 에 추가한다.

```
# 변수
self.idx_file = idx_file
# 바이너리 모드에서 텍스트로 읽기
first_line = idx_file.readline()
if isinstance(first_line, bytes):
    first_line = first_line.decode('utf-8')
self.b = int(first_line.rstrip('\n'))
self.nodes = []
self.load_node()
```

Node

먼저 Node는 **명세서에 지정되어 있는 필드**들과 Leaf인 경우에 Linked list를 구현하기 위한 `next_leaf_id` 와 `id` 로 구성되어져 있다.

Leaf인 경우와 Non-leaf인 경우에 `p_` 와 `p` 의 사용이 다르고, `r` 의 용도 역시 다르기에 leaf node 여부에 따라 다르게 정의되도록 하였다.

`is_full` 함수는 한 Node에 저장될 수 있는 최대 Key 개수를 넘겼는지 확인한다.

```
def __init__(self, b, node_id: int, is_leaf: int, keys: list, childs: list, next_leaf_id: int = None):
        self.b = b
        self.id = node_id
        self.is_leaf = bool(is_leaf)
        self.next_leaf_id = next_leaf_id

        if self.is_leaf:
            self.p_ = list(zip(keys, childs)) if keys else []
            self.p = []
            self.r = next_leaf_id
        else:
            self.p = list(zip(keys, childs[:-1])) if keys else []
            self.p_ = []
            self.r = childs[-1] if childs else None
```

```
        self.m = len(keys) # b-1개여야함

    def is_full(self): # 하나의 (Key, value)를 추가하면 넘치는지 확인한다.
        assert self.m <= self.b-1, "한 Node에 저장될 수 있는 최대 Key의 개수를 초과하였습니다."

        if (self.m == self.b - 1):
            return True
        else:
            return False
```

2. 파일 읽기 / 쓰기

우선 위의 Index file 형식에 맞춰 읽고, 각각에 맞는 Type에 맞는 변수로 생성하기 위해 `parse_line` 함수를 구현했다.

```
def parse_line(line):
    """
    node_id: int
    is_leaf: int (0: Not leaf, 1: Leaf)
    keys: List
    childs: List
    next_leaf_id: int
    """

    parsed_line = [l.strip() for l in line.split('|')]

    node_id = int(parsed_line[0])
    is_leaf = int(parsed_line[1])
    keys = list(map(int, parsed_line[2].split(','))) if parsed_line[2] else []
    childs = list(map(int, parsed_line[3].split(','))) if parsed_line[3] else []
    next_leaf_id = int(parsed_line[4]) if parsed_line[4] and parsed_line[4] != 'None' else None
```

```
    return node_id, is_leaf, keys, childs, next_leaf_id
```

이후, Tree.load_node()를 통해 Index file에서 각 줄을 읽고, Parsing하고 Node 객체를 생성한 후 self.nodes에 추가한다.

```
def load_node(self):
        # self.node initialize
        self.nodes.clear()

        for line in self.idx_file:
            # 바이너리 모드에서 텍스트로 읽기
            if isinstance(line, bytes):
                line = line.decode('utf-8')
            line = line.rstrip('\n')
            if not line:  # 빈 줄 건너뛰기
                continue

            node_id, is_leaf, keys, childs, next_leaf_id = parse_line(line)
            node = Node(self.b, node_id, is_leaf, keys, childs, next_leaf_id)
            self.nodes.append(node)
```

(1)에서 설명한 Node의 생성자에 따라 생성이 된다.

저장할 때는 우선 Node에 저장되어있는 field 값을 Index file 형태로 맞춰주기 위해 `Node.to_string` 함수를 구현했다.

```
def to_string(self):
        key = ','.join(map(str, [pair[0] for pair in (self.p_ if self.is_leaf else self.
p)]))

        if self.is_leaf:
            child = ','.join(map(str, [pair[1] for pair in self.p_]))
        else:
            child = ','.join(map(str, [pair[1] for pair in self.p]))
            if self.r is not None:
                child += f',{self.r}'

        next_leaf_str = str(self.next_leaf_id) if self.next_leaf_id is not None else
```

```
        '''

        return f"{self.id}|{int(self.is_leaf)}|{key}|{child}|{next_leaf_str}"
```

이후, Tree에서 self.nodes에 있는 Node 순서대로 저장하기 위해 `to_string`을 호출하고 파일에 Write한다.

- 이를 `save_file` 함수로 구현한다.

```
def save_file(self):
        self.idx_file.seek(0)
        self.idx_file.truncate()

        # 첫 줄에 b 값 저장
        self.idx_file.write(f'{self.b}\n'.encode('utf-8'))

        # node_id 순으로 정렬 후 저장
        sorted_nodes = sorted(self.nodes, key=lambda x: x.id)
        for node in sorted_nodes:
            self.idx_file.write((node.to_string() + '\n').encode('utf-8'))

        self.idx_file.flush()
```

3. 구현 알고리즘

a. Insertion

1. 우선 Insert의 대상이 되는 Leaf node를 search를 통해 찾는다.
2. 우선 해당 leaf node에 insert한다.
3. (2)의 node (2)의 insert 결과로 가득 차지 않았다면 그냥 추가한다.
4. 그렇지 않다면 split 해야한다.
    - 이 때, split은 오른쪽 child의 첫 번째 key값이 Internal key가 되도록 한다.
    - 이 때, **4가지 경우**가 있다.

    a. (4)번의 insert하는 **node가 root node이며 leaf node**이라면, root, right node를 새로 생성하고 root, left, right를 적절히 설정한다.

b. (4)번의 insert하는 **node가 root node이며 leaf node가 아니라면**, root, right node를 새로 생성하고 root, left, right를 적절히 설정한다.

c. (4)번의 **insert하는 node가 root node가 아니고 leaf node**이라면, right node를 새로 생성하고 기존 parent에 Internal key를 넘겨준다.

d. (4)번의 **insert하는 node가 root node가 아니고 leaf node가 아니라면**, right node를 새로 생성하고 grandparent에 Internal key를 넘겨준다.

5. (4)의 과정을 더 이상 추가할 Key가 없을 때까지 반복한다.

b. Delete

1. 우선 Delete의 대상이 되는 Leaf node를 search를 통해 찾는다.
2. **min_key**: 현재 Delete하고자 하는 Key가 right_child의 첫 번째 Key인 경우인지 확인한다.
    a. **현재 삭제하고자 하는 Key가 Internal node에 사용되고 있을 가능성**이 있는지 체크
3. min_key인 경우 현재 Key가 어떤 Internal node에 들어있는지 찾고, 그 Node를 **Target node**로 설정한다.
    a. 삭제 과정에서 Target node에서 **삭제하고자 하는 Key값을 삭제 후의 구조에 맞게 변경**해야 한다.
4. 이제 Key를 Leaf에서 delete한다. 이때, Min key 여부와 Delete 했을 때, leaf node가 유지될 수 있는지 여부에 따라 **4가지 케이스**로 나눈다.
    a. **Min key가 아니고 leaf node가 그대로 유지**될 수 있다면 그냥 삭제한다.
    b. **Min key이고 leaf node가 유지될 수 있다면** leaf node에서 삭제한 이후, target에서 일치하는 key값을 leaf node의 새로운 min key값으로 바꾼다.
    c. **Min key가 아니고, leaf node가 그대로 유지될 수 없다면,** 왼쪽 또는 오른쪽 노드에서 Key값을 빌려오거나 병합해야 한다.
        i. 이때, 병합 시에는 부모 Node에 대해 안정성 검사를 추가로 수행해야 한다.
    d. **Min key이고, leaf node가 그대로 유지될 수 없다면,** 우선 target에서 일치하는 Key값을 leaf node의 새로운 min_key값으로 바꾼다**.** 이후, 왼쪽 또는 오른쪽 노드에서 Key값을 빌려오거나 병합해야 한다.
        i. 이때, 병합 시에는 부모 Node에 대해 안정성 검사를 추가로 수행해야 한다.
5. (4)에서 Merge에 일어나는 경우도 **두 가지**로 나눌 수 있다.

a. **현재 Node를 왼쪽 Node에** 붙이는 경우

b. **오른쪽 Node를 현재 Node**에 붙이는 경우

6. Merge 이후에는 Parent가 유지될 수 있는지 확인해야 한다. 유지될 수 없다면 **Internal Key에서 왼쪽 또는 오른쪽 Node에서 Key 값을 빌려오거나 병합**한다.

   a. 이 작업은 모든 Node가 유지될 수 있을 때까지 반복된다.

c. Single Search

1. Root부터 시작해서 찾고자 하는 Key 값이 확인하고 있는 Node의 Key값보다 작다면 Left, 크거나 같다면 Right로 움직인다.
2. (1)의 작업을 Leaf node가 나올 때까지 반복한다.
3. (2)에서 찾은 Leaf node에 일치하는 key가 있다면 대응되는 Value를 출력하고, 없다면 ”NOT FOUND”를 출력한다.

d. Ranged Search

1. Root부터 시작해서 start_key 값이 확인하고 있는 Node의 Key값보다 작다면 Left, 크거나 같다면 Right로 움직인다.
2. (1)의 작업을 Leaf node가 나올 때까지 반복한다.
3. (2)에서 찾은 Leaf node에 start_key와 end_key 사이의 값을 가지는 Key가 있다면 해당 (Key, Value)를 출력한다.
4. (2)에서 찾은 Leaf node에서 end_key보다 큰 Key가 존재하지 않는다면 다음 Leaf node로 넘어가서 (3)의 작업을 반복한다.
5. end_key보다 큰 Key가 나올 때까지 (4)의 작업을 반복한다.

4. 자세한 구현

a. Insertion

아래 모든 함수는 tree.py에 구현되어 있습니다.

먼저 insertion 함수부터 살펴보자

- Tree에 아무 Node도 존재하지 않거나 중복 된 Key가 존재하는 등 예외 사항 먼저 처리한다.

- 이후, 현재 Key, value 삽입 이후, Node가 가득 차는지 확인하기 위해 `is_full` 함수를 호출한다.
- `is_full` = False라면 그냥 Leaf에 추가한다.
- `is_full` = True라면 알고리즘에서 설명한 4가지 경우를 `insert` 함수를 통해 처리한다.

```
def insertion(self, csv_reader):
        for row in csv_reader:
            assert len(row) >= 2, "Data file에는 key, vaule 쌍이 존재해야 합니다."
            key = int(row[0])
            value = int(row[1])
            raw_key = key # for debug

            print(f"입력할 Key, Value 쌍은 다음과 같습니다. {key}, {value}")

            # Tree에 아무 Node도 존재하지 않는 경우
            if len(self.nodes) == 0:
                node = Node(self.b, 0, 1, [key], [value], None)
                print(f"{key}: root 생성")
                self.nodes.append(node)
                continue

            # 위치를 찾아야함
            root = self.get_root()
            if root is None:
                # 빈 트리인 경우
                node = Node(self.b, 0, 1, [key], [value], None)
                self.nodes.append(node)
                continue

            node = self.search(root, key, False) # 일치하는 위치의 Leaf Node를 찾는다.

            if node is None or not node.is_leaf:
                print("일치하는 Node를 찾을 수 없거나 Leaf node가 아닙니다.")
                continue

            # node에 이미 같은 Key가 있는지 확인
```

```
        if node.searchLeaf(key) is not None:
            print("중복되는 Key는 Insert할 수 없습니다.")
            continue

        index = 0

        if node.is_full(): # 찾은 Leaf Node에 하나 추가하면 용량 초과
            # Split이 필요한 경우
            while (1):
                # Leaf node에 insert할 때, Search를 통해 중복 Key 검사를 하므로 여
기서 또 할 필요는 없을 듯

                # node가 None이면 추가할 게 없다는 뜻이므로 즉시 종료
                if node is None:
                    break

                # Node에 우선 삽입
                if node.is_leaf:
                    node.p_.append((key, value))
                    node.p_.sort(key=lambda k: k[0])
                else:
                    keys = [k for k, _ in node.p]
                    childs = [v for _, v in node.p] + [node.r]

                    if index == len(node.p):
                        keys.append(key)
                        childs.append(value) # 원래 왼쪽 Child가 위치한 위치 옆에 추
가
                    else:
                        keys.insert(index, key)
                        childs.insert(index+1, value) # 원래 왼쪽 Child가 위치한 위치
옆에 추가

                    node.p = list(zip(keys, childs[:-1]))
                    node.r = childs[-1]


                if not node.is_full(): # 이 경우 node에 추가해줘야 함 (key, value) /
```

추가하고 Loop를 종료하도록 한다.

```
                # Parent node에 공간이 남아있다면 추가한 후 업데이트 해줘야함
                # Parent node도 가득 찼다면 insert에서 update
                if node.is_leaf:
                    node.m = len(node.p_)
                else:
                    node.m = len(node.p)

                break

            # split은 우선 node.p 또는 p_에 key, value 쌍을 추가하는 작업부터 수행한다.
            left_key, left_value , right_key, right_value, m = self.split(node)



            # parent node도 꽉 찼다면 다시 Split 후 Insert 반복
            node, key, value, index = self.insert(node, left_key, left_value , right_key, right_value, m) # insert의 결과로 Parent node를 받음

            id = node.id if node else None

            self.set_id()

            self.leaf_node_check()

        else: # Leaf node에 그냥 추가해도 되는 경우
            node.p_.append((key, value))
            node.p_.sort(key=lambda k: k[0])
            node.m = len(node.p_)  # m 값 업데이트

        print(f"Insert {raw_key} 이후 최종 상태")
        print("-" * 50)
        for n in self.nodes:
            if n.is_leaf:
                print(n.id, int(n.is_leaf), n.p_, n.m, n.r, n.next_leaf_id)
            else:
                print(n.id, int(n.is_leaf), n.p, n.m, n.r, n.next_leaf_id)
        print("-" * 50, end="\n\n")
```

```
        # 변경사항을 파일에 저장
        self.save_file()
```

`insert` 함수를 살펴보기 전에, `split` 함수를 먼저 살펴보자.

- leaf node 여부에 따라 `p_`, `p` 사용 여부가 다르므로 나누어 구현하였다.
- Internal node는 right child에 포함되도록 나눈다.

```
def split(self, node): # Node.p는 삽입할 (Key, Value)가 Append된 이후 정렬까지 된 상태이어야 한다.
        middle_idx = (node.m-1) // 2

        if node.is_leaf:
            left = node.p_[:middle_idx+1]
            right = node.p_[middle_idx+1:]

            left_key = [k for k, _ in left]
            left_value = [v for _, v in left]
            right_key = [k for k, _ in right]
            right_value = [v for _, v in right]

            m = right_key[0]

            return left_key, left_value , right_key, right_value, m
        else:
            middle_idx = middle_idx + 1 # //이 Floor 연산이라 1 추가해야됨
            m, v = node.p[middle_idx]
            # Non-leaf node split: middle key는 부모로 올리고 right에서 제거
            left = node.p[:middle_idx]
            right = node.p[middle_idx+1:]  # middle key는 부모로 올리므로 제외

            left_key = [k for k, _ in left]
            left_value = [v for _, v in left] # Child는 제거 없이 전부 사용
            left_value.append(v)
            right_key = [k for k, _ in right]
            right_value = [v for _, v in right] + [node.r]
```

```
        return left_key, left_value , right_key, right_value, m
```

`insert` 는 알고리즘에서 설명한 4가지 Case를 각각 처리한다.

- 내부적으로는 Insert 이후에 Split으로 인한 **Child pointer, node id, next_leaf_id, r 등을 업데이트** 하는 코드를 포함한다.
- Case를 4가지로 나누는 이유는 leaf_node인지에 따라 p, p_ 사용 방식이 다르며, node의 field update 방식이 다르기 때문이다.
- 각 Case를 주석에 자세히 설명하였다.

```
def insert(self, node, left_key, left_value , right_key, right_value, m): # node: 현재 insert를 하고자하는 Node
        parent = None
        key = None
        value = None
        index = None

        # 1. Root Node이면서 Leaf Node인 경우
        if node.id == 0 and node.is_leaf:
            print("Case 1")
            left_id = 1
            right_id = 2

            root = Node(self.b, 0, 0, [m], [left_id, right_id], None)
            right_child = Node(self.b, right_id, 1, right_key, right_value, None)

            # 기존 node의 Field를 업데이트
            node.id = left_id
            node.p_ = list(zip(left_key, left_value)) if left_key else []
            node.next_leaf_id = right_id
            node.r = right_id
            node.m = len(node.p_)

            # nodes 배열 업데이트
            self.nodes.insert(0, root)
            self.nodes.append(right_child)
```

```
            parent = None # 추가 삽입 없이 종료되도록 한다.

            return parent, key, value, index
        # 2. Root Node이고 Leaf Node가 아닌 경우
        elif node.id == 0 and not node.is_leaf:
            print("Case 2")
            left_id = 1
            right_id = 2

            root = Node(self.b, 0, 0, [m], [left_id, right_id], None)
            right_child = Node(self.b, right_id, 0, right_key, right_value, None)

            # 기존 node를 left child로 변경
            node.id = left_id
            node.p = list(zip(left_key, left_value[:-1])) if left_key else []
            node.r = left_value[-1]
            node.m = len(node.p)

            # nodes 배열 업데이트
            self.nodes.insert(0, root)
                    # Root 삽입의 결과로 Root를 제외한 모든 Non-leaf node의 child
pointer를 하나씩 증가
            for idx, n in enumerate(self.nodes):
                if idx == 0 or n.is_leaf:
                    continue
                for i, (k, v) in enumerate(n.p):
                    if v is not None and v >= 0:
                        n.p[i] = (k, v + 1)
                # r
                if n.r is not None and n.r >= 0:
                    n.r += 1

            if not right_child.is_leaf:
                for i, (k, v) in enumerate(right_child.p):
                    if v is not None:
                        right_child.p[i] = (k, v + 1)
                if right_child.r is not None:
                    right_child.r += 1
```

```
            if len(self.nodes) == right_id: # self.nodes의 가장 끝에 추가해야되는 경우
                self.nodes.append(right_child)
            else:
                self.nodes.insert(right_id, right_child)

            # right child가 새로 생겼기 때문에right 이후의 non-leaf node의 child pointer를 하나씩 증가
            for idx, n in enumerate(self.nodes):
                if idx == 0 or n.is_leaf:
                    continue
                for i, (k, v) in enumerate(n.p):
                    if v is not None and v >= right_id:
                        n.p[i] = (k, v + 1)
                if n.r is not None and n.r >= right_id:
                    n.r += 1

            # insertion에서 id 정렬 및 leaf node의 r, next_leaf_id 정렬한다.

            parent = None # 추가 삽입 없이 종료되도록 한다.

            return parent, key, value, index
        # 3. Root Node가 아니고 Leaf Node인 경우
        elif node.id != 0 and node.is_leaf:
            print("Case 3")
            right_id = node.id + 1  # 새로운 ID 할당

            right_child = Node(self.b, right_id, 1, right_key, right_value, node.next_leaf_id)

            # 기존 node 업데이트
            node.p_ = list(zip(left_key, left_value)) if left_key else []
            node.next_leaf_id = right_id
            node.r = right_id
            node.m = len(node.p_)
```

```
            # parent 찾는 logic 추가
            parent = None
            index = None

            parent, index = self.find_parent(node)

            assert parent is not None, "parent를 찾을 수 없습니다."

            key = m
            value = right_id

            # node.id 위치에 추가해야 맞는 위치에 추가된다.
            if len(self.nodes) == right_id: # self.nodes의 가장 끝에 추가해야되는 경우
                self.nodes.append(right_child)
            else:
                self.nodes.insert(right_id, right_child)

            # right child가 새로 생겼기 때문에right 이후의 non-leaf node의 child pointer를 하나씩 가
            for n in self.nodes:
                if n.is_leaf:
                    continue
                for i, (k, v) in enumerate(n.p):
                    if v is not None and v >= right_id:
                        n.p[i] = (k, v + 1)
                if n.r is not None and n.r >= right_id:
                    n.r += 1

            return parent, key, value, index
        # 4. Root도 아니고 Leaf도 아닌 경우
        elif node.id != 0 and not node.is_leaf:
            print("Case 4")
            right_id = node.id+1
            right_child = Node(self.b, right_id, 0, right_key, right_value, None)

            node.p = list(zip(left_key, left_value[:-1])) if left_key else []
            node.r = left_value[-1]
```

```
            node.m = len(node.p)

            # parent 찾는 logic 추가
            parent = None
            index = None

            parent, index = self.find_parent(node)

            assert parent is not None, "parent를 찾을 수 없습니다."

            key = m
            value = right_id

            # node.id 위치에 추가해야 맞는 위치에 추가된다.
            if len(self.nodes) == right_id: # self.nodes의 가장 끝에 추가해야되는 경우
                self.nodes.append(right_child)
            else:
                self.nodes.insert(right_id, right_child)
                    # right child가 새로 생겼기 때문에right 이후의 non-leaf node의 child pointer를 하나씩 가
            for n in self.nodes:
                if n.is_leaf:
                    continue
                for i, (k, v) in enumerate(n.p):
                    if v is not None and v >= right_id:
                        n.p[i] = (k, v + 1)
                if n.r is not None and n.r >= right_id:
                    n.r += 1

            return parent, key, value, index
        # 위 네 가지 경우 중 하나도 속하지 않는 경우
        else:
            print("잘못된 Insertion의 경우입니다.")
            return parent, key, value, index
```

b. Delete

우선 `deletion` 함수는 **(2)의 알고리즘에서 설명한 동일한 작업**을 수행한다.

- 각 delete 이후, node의 field 정리를 해주는 부분이 있다.
- Internal Key를 다루기 위해 `find_target` 함수를 구현해, 찾고자 하는 Key값이 어떤 Internal node에 있는지 반환하도록 하였다.
- `self.nodes`에 **root만 존재하는 예외 케이스**를 따로 다룬다.
- 왼쪽 / 오른쪽 node를 찾고, **왼쪽 / 오른쪽 node가 존재하는 지에 따라 Case 3, 4에서는 세부적으로 경우의 수**를 나누었다.
    - Case 3 내부와 Case 4 내부에서 **왼쪽 Node가 있는지, 오른쪽 Node가 있는지에 따라 나뉘고 왼쪽 또는 오른쪽에서 하나를 빌려올 수 있는지와 그럴 수 없다면 병합하는 부분으로 세부적으로 나뉜다.**
    - 때문에 중복되는 코드가 많으니 해당 코드가 처음 등장하는 부분에만 자세한 주석을 작성했다.
- **왼쪽 / 오른쪽 node에서 Key 하나를 빌리는 부분**은 `deletion` 내부에 구현했다.
- `merge` 부분은 따로 함수를 구현해서 호출하여 사용한다.

```
def deletion(self, csv_reader):
        for row in csv_reader:
            if len(row) >= 1:  # key가 있는지 확인
                key = int(row[0])
                print(f"삭제할 Key: {key}")

                if len(self.nodes) == 0:
                    print("Tree 안에 Node가 없어 delete할 수 없습니다.")
                    continue

                # 위치를 찾아야함
                root = self.get_root()
                if root is None:
                    print("Root가 없습니다.")
                    continue
                            # Root node만 존재하는 Case
                if len(self.nodes) == 1:
                    print("Root만 존재하는 경우")
```

```
                for i, (k, v) in enumerate(root.p_):
                    if k == key:
                        leaf_index = i

                del root.p_[i]
                root.m = root.m - 1

                # Root node의 경우 is_half가 만족되지 않아도 괜찮다.

                # Root가 비면 모든 node 삭제
                if root.m == 0:
                    self.nodes = []
                continue

            leaf_node = self.search(root, key, False) # 일치하는 위치의 Leaf No
de를 찾는다.

            if leaf_node is None or not leaf_node.is_leaf:
                print("일치하는 Node를 찾을 수 없습니다.")
                continue

            # 리프 노드에서 키 찾기
            leaf_index = None
            for i, (k, v) in enumerate(leaf_node.p_):
                print(k, key)
                if k == key:
                    leaf_index = i
                    break

            if leaf_index is None:
                print("찾은 leaf node에서 key 값을 찾을 수 없습니다.")
                continue



            min_key = False
            if leaf_index == 0 and self.nodes[leaf_node.id-1].is_leaf: # 삭제할
키가 해당 노드의 첫 번째 키인 경우 + 가장 왼쪽 Leaf가 아닌 경우
                min_key = True # 최소키인지 여부를 체크
```

```
            target = None
            target_index = None

            if min_key:
                target, target_index = self.find_target(key)

            if min_key and target is None:
                print("Min Key이지만 상위 노드에 Key값이 없습니다.")

            # 우선 Leaf node에서 삭제
            del leaf_node.p_[leaf_index]
            leaf_node.m = leaf_node.m - 1

            # Case 1: Min_key도 아니고, leaf 삭제 문제 없음
            if not min_key and leaf_node.is_half_for_leaf():
                print("Case 1")
                self.print_tree(key)
                continue
            # Case 2: Min_key이고, leaf 삭제 문제 없음
            if min_key and leaf_node.is_half_for_leaf():
                print("Case 2")
                            # 삭제 후 첫 번째 Key 값으로 Target node에 있는 key
값을 대체
                            # key를 삭제하면 leaf_node의 삭제 후 첫 번째 값이 min
_key가 된다.
                if len(leaf_node.p_) > 0:
                    new_min_key, _ = leaf_node.p_[0]
                    _, v = target.p[target_index]
                    target.p[target_index] = (new_min_key, v)
                self.print_tree(key)
                continue
```

```
                    # Case 3: Min_key가 아니고, leaf 삭제 문제 있음
            if not min_key and not leaf_node.is_half_for_leaf():
                print("Case 3")
                # 빌려오기 또는 병합
                parent, parent_index = self.find_parent(leaf_node)
```

```
            if parent is None:
                print("Case 3: Parent를 찾을 수 없습니다.")
                return

            left = None
            right = None
            left, right = self.find_sibling(leaf_node, parent, parent_index)

            if left and right: # left, right 모두 존재하는 경우
                print("Case 3: left, right 모두 존재하는 경우")
                # 1. Left에서 빌려올 수 있는 경우
                if left.m > (self.b - 1) // 2:
                    print("Case 3: 왼쪽에서 빌려옴")
                    lk, lv = left.p_[-1]

                    # 현재 Node의 Internal key가 Left[-1]로 변함
                    if len(leaf_node.p_) > 0:
                        internal_key, _ = leaf_node.p_[0] # 원래 현재 Node의 min_
key
                        # 현재 Node의 가장 앞에 새로운 key가 추가
                        # 현재 Node의 Internal Key가 위 Key 값으로 변경되어야됨
                        new_target, new_target_index = self.find_target(internal_
key)
                        if new_target is not None:
                            _, v = new_target.p[new_target_index]
                            new_target.p[new_target_index] = (lk, v)
                                    # Left에서 삭제
                    del left.p_[-1]
                    left.m = left.m - 1

                    # Case 3에선 len(leaf_node.p_) == 0 / b=3인 경우가 나오면
안됨

                    leaf_node.p_.insert(0, (lk, lv))
                    leaf_node.m = leaf_node.m + 1


                # 2. Right에서 빌려올 수 있는 경우
```

```
                elif right.m > (self.b - 1) // 2:
                    print("Case 3: 오른쪽에서 빌려옴")
                    rk, rv = right.p_[0]

                    del right.p_[0]
                    right.m = right.m - 1

                    # Right의 min_key가 변경 -> Right의 target도 바뀌어야 함
                    # 기존 min_key인 rk로 찾고 새로운 key값으로 교체한다.
                    if len(right.p_) > 0:
                        new_min_key, _ = right.p_[0]
                        right_target, right_index = self.find_target(rk)
                        if right_target is not None: # 바뀌어야 하는 Target 대상이
있다면
                            _, v = right_target.p[right_index]
                            right_target.p[right_index] = (new_min_key, v)

                    leaf_node.p_.append((rk, rv))
                    leaf_node.m = leaf_node.m + 1

                # 3. Merge 필요
                else:
                    print("Case 3: Merge 필요")
                    # Left와 Right 모두 merge 가능
                    if left and right:
                        print("Case 3: Left와 Right 모두 merge 가능")
                        # Left와 merge
                        remove_index = parent_index - 1 # Internal Key를 제거해
야한다.
                        after_merge = self.merge_leaf_nodes(left, leaf_node, pa
rent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merge and not after_merge.is_half_for_internal():
                            self.check_parent(after_merge)
                    elif left:
                        print("Case 3: Left만 존재하는 경우")
                        # Left와 merge
                        remove_index = parent_index - 1 # Left의 Internal Key를
```

```
미
                        after_merge = self.merge_leaf_nodes(left, leaf_node, pa
rent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merge and not after_merge.is_half_for_internal():
                            self.check_parent(after_merge)
                    elif right:
                        print("Case 3: Right만 존재하는 경우")
                        # Right와 merge
                        remove_index = parent_index
                        after_merge = self.merge_leaf_nodes(leaf_node, right, p
arent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merge and not after_merge.is_half_for_internal():
                            self.check_parent(after_merge)
                elif left: # left만 존재하는 경우
                    print("Case 3: Left만 존재하는 경우")
                    # 1.Left에서 빌려올 수 있는 경우
                    if left.m > (self.b - 1) // 2:
                        print("Case 3: Left에서 빌려옴")
                        lk, lv = left.p_[-1]

                        # 현재 Node의 Internal key가 Left[-1]로 변함
                        if len(leaf_node.p_) > 0:
                            internal_key, _ = leaf_node.p_[0]
                            new_target, new_target_index = self.find_target(internal_
key)
                            if new_target is not None:
                                _, v = new_target.p[new_target_index]
                                new_target.p[new_target_index] = (lk, v)

                        del left.p_[-1]
                        left.m = left.m - 1

                        # Case 3에선 이 경우가 나오면 안됨 len(leaf_node.p_) == 0 /
b=3인 경우
                        leaf_node.p_.insert(0, (lk, lv))
                        leaf_node.m = leaf_node.m + 1
```

```
                # 2. Merge
                else:
                    print("Case 3: Merge 필요")
                    # Left와 merge
                    print("Case 3: Left와 merge")
                    remove_index = parent_index - 1
                    after_merged = self.merge_leaf_nodes(left, leaf_node, par
ent, remove_index)
                    # Parent가 underflow되는지 확인
                    if after_merged and not after_merged.is_half_for_internal():
                        self.check_parent(after_merged)
            elif right: # right만 존재하는 경우
                print("Case 3: Right만 존재하는 경우")
                # 1. Right에서 빌려올 수 있는 경우
                if right.m > (self.b - 1) // 2:
                    rk, rv = right.p_[0]

                    del right.p_[0]
                    right.m = right.m - 1

                    # Right의 min_key가 변경 -> Right의 target도 바뀌어야 함
                    # 기존 min_key인 rk로 찾고 새로운 key값으로 교체한다/
                    if len(right.p_) > 0:
                        new_min_key, _ = right.p_[0]
                        right_target, right_index = self.find_target(rk)
                        if right_target is not None: # 바뀌어야 하는 Target 대상이
있다면
                            _, v = right_target.p[right_index]
                            right_target.p[right_index] = (new_min_key, v)

                    leaf_node.p_.append((rk, rv))
                    leaf_node.m = leaf_node.m + 1

                # 2. Merge 필요
                else:
                    print("Case 3: Merge 필요")
                    print("Case 3: Right와 merge")
```

```
                    # Right와 merge
                    remove_index = parent_index
                    after_merged = self.merge_leaf_nodes(leaf_node, right, pa
rent, remove_index)
                    # Parent가 underflow되는지 확인
                    if after_merged and not after_merged.is_half_for_internal():
                        self.check_parent(after_merged)
```

```
                    # Case 4: Min_key이고, leaf 삭제 문제 있음
            if min_key and not leaf_node.is_half_for_leaf():
                print("Case 4")
                # 우선 Min key 처리
                # 삭제하고자 하는 Key가 Min key라는 것은 Case 2와 마찬가지로
                # Internal key를 바뀌야 한다는 의미이기 때문에
                # 우선 삭제 후 leaf node에서의 첫 번째 key값으로 변경한다.
                if len(leaf_node.p_) > 0:
                    new_min_key = leaf_node.p_[0][0]
                    _, v = target.p[target_index]
                    target.p[target_index] = (new_min_key, v)

                # 이후 빌려오기 또는 병합
                parent, parent_index = self.find_parent(leaf_node)
                if parent is None:
                    print("Case 4: Parent를 찾을 수 없습니다.")

                left = None
                right = None
                left, right = self.find_sibling(leaf_node, parent, parent_index)

                if left and right: # left, right 모두 존재하는 경우
                    print("Case 4: left, right 모두 존재하는 경우")
                    # 1. Left에서 빌려올 수 있는 경우
                    if left.m > (self.b - 1) // 2:
                        print("Case 4: 왼쪽에서 빌려옴")
                        lk, lv = left.p_[-1]

                        # target이 새로운 Internal key로 바뀌어야 한다.
                        # 현재 Node의 Internal Key가 변경
```

```
                    _, v = target.p[target_index]
                    target.p[target_index] = (lk, v)

                    del left.p_[-1]
                    left.m = left.m - 1

                    if len(leaf_node.p_) == 0: # b=3인 경우
                        leaf_node.p_.append((lk, lv)) # 이 경우에는 leaf_node.p_
가 비어있어 insert가 불가능
                        leaf_node.m = leaf_node.m + 1
                    else:
                        leaf_node.p_.insert(0, (lk, lv))
                        leaf_node.m = leaf_node.m + 1

                # 2. Right에서 빌려올 수 있는 경우
                elif right.m > (self.b - 1) // 2:
                    print("Case 4: 오른쪽에서 빌려옴")
                    rk, rv = right.p_[0]

                    del right.p_[0]
                    right.m = right.m - 1

                    # Right의 Internal key 변경
                    if len(right.p_) > 0:
                        new_right_min_key, _ = right.p_[0]
                        right_target, right_index = self.find_target(rk) # 기존 rk 값
으로 target 찾음
                        if right_target is not None: # 바뀌어야 하는 Target 대상이
있다면
                            _, v = right_target.p[right_index]
                            right_target.p[right_index] = (new_right_min_key, v)

                    if len(leaf_node.p_) == 0: # b=3인 경우
                        # 현재 node에 아무것도 없는 경우에는 new_min_key, _ = lea
f_node.p_[0]로 하면 index 에러
                        # right node가 삽입될 것이고 그때 right node를 새로운 min_
key로 사용해야 한다.
                        _, v = target.p[target_index]
```

```
                        target.p[target_index] = (rk, v)

                        leaf_node.p_.append((rk, rv)) # 이 경우에는 leaf_node.p_
가 비어있어 insert가 불가능
                        leaf_node.m = leaf_node.m + 1
                    else:
                        # target이 새로운 Internal key로 바뀌어야 한다.
                        # 현재 Node의 Internal key가 변경
                        if len(leaf_node.p_) > 0:
                            new_min_key, _ = leaf_node.p_[0] # 기존 min_key nod
e는 이미 삭제된 상태
                            _, v = target.p[target_index]
                            target.p[target_index] = (new_min_key, v)

                        leaf_node.p_.append((rk, rv))
                        leaf_node.m = leaf_node.m + 1

                    self.print_tree(key)
                    continue
                # 3. Merge 필요
                else:
                    # Left와 Right 모두 merge 가능
                    print("Case 4: Merge 필요")
                    print("Case 4: Left와 Right 모두 merge 가능")
                    if left and right:
                        # Left와 merge (left가 더 작은 ID를 가지므로)
                        remove_index = parent_index - 1
                        # left에 현재 node를 붙임
                        after_merged = self.merge_leaf_nodes(left, leaf_node, p
arent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merged and not after_merged.is_half_for_internal
():
                            self.check_parent(after_merged)
                    elif left:
                        # Left와 merge
                        remove_index = parent_index - 1
                        after_merged = self.merge_leaf_nodes(left, leaf_node, p
```

```
arent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merged and not after_merged.is_half_for_internal
():
                            self.check_parent(after_merged)
                    elif right:
                        # Right와 merge (현재 노드를 left로, right를 right로)
                        remove_index = parent_index
                        # 현재 Node에 right node를 붙임
                        after_merged = self.merge_leaf_nodes(leaf_node, right,
parent, remove_index)
                        # Parent가 underflow되는지 확인
                        if after_merged and not after_merged.is_half_for_internal
():
                            self.check_parent(after_merged)
                elif left: # left만 존재하는 경우
                    print("Case 4: Left만 존재하는 경우")
                    # 1.Left에서 빌려올 수 있는 경우
                    if left.m > (self.b - 1) // 2:
                        print("Case 4: Left에서 빌려옴")
                        lk, lv = left.p_[-1]

                        # target이 새로운 Internal node로 바뀌어야 한다.
                        # 현재 Node의 Internal node가 변경
                        _, v = target.p[target_index]
                        target.p[target_index] = (lk, v)

                        del left.p_[-1]
                        left.m = left.m - 1

                        if len(leaf_node.p_) == 0: # b=3인 경우
                            leaf_node.p_.append((lk, lv)) # 이 경우에는 leaf_node.p_
가 비어있어 insert가 불가능
                            leaf_node.m = leaf_node.m + 1
                        else:
                            leaf_node.p_.insert(0, (lk, lv))
                            leaf_node.m = leaf_node.m + 1
                    # 2. Merge
```

```
                else:
                    print("Case 4: Merge 필요")
                    print("Case 4: Left와 merge")
                    # Left와 merge
                    remove_index = parent_index - 1
                    after_merged = self.merge_leaf_nodes(left, leaf_node, par
ent, remove_index)
                    # Parent가 underflow되는지 확인
                    if after_merged and not after_merged.is_half_for_internal():
                        self.check_parent(after_merged)
            elif right: # right만 존재하는 경우
                print("Case 4: Right만 존재하는 경우")
                # 1. Right에서 빌려올 수 있는 경우
                if right.m > (self.b - 1) // 2:
                    print("Case 4: Right에서 빌려옴")
                    rk, rv = right.p_[0]

                    del right.p_[0]
                    right.m = right.m - 1

                    # Right의 Internal node도 변경
                    if len(right.p_) > 0:
                        new_right_min_key, _ = right.p_[0]
                        right_target, right_index = self.find_target(rk) # 기존 rk 값
으로 target 찾음
                        if right_target is not None: # 바뀌어야 하는 Target 대상이
있다면
                            _, v = right_target.p[right_index]
                            right_target.p[right_index] = (new_right_min_key, v)

                    if len(leaf_node.p_) == 0: # b=3인 경우
                        # 현재 node에 아무것도 없는 경우에는 new_min_key, _ = lea
f_node.p_[0]로 하면 index 에러
                        # right node가 삽입될 것이고 그때 right node를 새로운 min_
key로 사용해야 한다.
                        _, v = target.p[target_index]
                        target.p[target_index] = (rk, v)
```

```
                        leaf_node.p_.append((rk, rv)) # 이 경우에는 leaf_node.p_
가 비어있어 insert가 불가능
                        leaf_node.m = leaf_node.m + 1
                    else:
                        # target이 새로운 Internal node로 바뀌어야 한다.
                        # 현재 Node의 Internal node가 변경
                        if len(leaf_node.p_) > 0:
                            new_min_key, _ = leaf_node.p_[0] # 기존 min_key nod
e는 이미 삭제된 상태
                            _, v = target.p[target_index]
                            target.p[target_index] = (new_min_key, v)

                        leaf_node.p_.append((rk, rv))
                        leaf_node.m = leaf_node.m + 1

                # 2. Merge 필요
                else:
                    print("Case 4: Merge 필요")
                    print("Case 4: Right와 merge")
                    # Right와 merge
                    remove_index = parent_index
                    after_merged = self.merge_leaf_nodes(leaf_node, right, pa
rent, remove_index)
                    # Parent가 underflow되는지 확인
                    if after_merged and not after_merged.is_half_for_internal():
                        self.check_parent(after_merged)

            # 모든 변경사항 후 leaf node의 field 업데이트
            self.set_id()
            self.leaf_node_check()

            self.print_tree(key)

        # 변경사항을 파일에 저장
        self.save_file()
```

`merge` 는 아래와 같이 구현된다.

먼저 `leaf_node` 에 대해 먼저 `leaf_node` 끼리 merge하기 위한 코드가 필요하다.

- 이를 `merge_leaf_nodes` 라는 함수로 구현했다.
- 기본적으로 이 함수는 left에 right를 붙이고, right node를 제거한다.
- 이후, left에 right가 추가됨에 따라 생기는 field를 조정한다.

```
    # Leaf node 두 개를 Merge
  def merge_leaf_nodes(self, left, right, parent, remove_index):
    # right의 모든 키를 left에 추가한다.
    left.p_.extend(right.p_)
    left.p_.sort(key=lambda k: k[0])
    left.m = len(left.p_)

    # left의 next_leaf_id 업데이트
    left.next_leaf_id = right.next_leaf_id
    left.r = right.r

    # remove_index는 Internal node 위치
    if remove_index < len(parent.p): # Parent에서 Internal node를 제거
      del parent.p[remove_index]
    else: # 이 경우에는 parent.r을 left.r로 바꾸어야 제대로 동작한다.
      if remove_index > 0 and remove_index - 1 < len(parent.p):
        del parent.p[remove_index - 1]
      # parent.r은 left.id로 업데이트
      parent.r = left.id

    parent.m = len(parent.p)

    # right에 대한 pointer를 left.id로 바꾼다.
    new_p = []
    for (k, v) in parent.p:
      if v == right.id:
        new_p.append((k, left.id))
      else:
        new_p.append((k, v))
    parent.p = new_p

    print("[DEBUG]:", parent.p)
```

```
        if parent.r == right.id:
            parent.r = left.id

        # right를 nodes 배열에서 제거하고 ID 재정렬
        removed_id = right.id
        self.nodes.remove(right)

        # Right에 대응되는 Node가 삭제되었기 때문에 Field를 -1씩 수행한다.
        # Insert에서 Field를 조정한 방법과 동일하다.
        for n in self.nodes:
            # node id를 -1씩
            if n.id is not None and n.id > removed_id:
                n.id -= 1

            if n.is_leaf:
                # leaf인 경우에는 removed_id보다 크면 r, next_lead_id 조정
                if n.next_leaf_id is not None and n.next_leaf_id > removed_id:
                    n.next_leaf_id -= 1
                if n.r is not None and n.r > removed_id:
                    n.r -= 1
            else:
                # r만 조정
                n.p = [(k, (v - 1 if v is not None and v > removed_id else v)) for (k,
v) in n.p]
                if n.r is not None and n.r > removed_id:
                    n.r -= 1

        return parent
```

`Merge`가 발생한 이후에는 `Parent` 도 그대로 유지되어도 괜찮은지 확인한다.

- 이는 `check_parent` 함수를 구현해서 재귀적으로 확인할 수 있도록 구현했다.
- `check_parent` 가 호출되는 경우는 **leaf_node의 parent 혹은 다른 parent의 parent가 유지될 수 없는 경우**이다.
    - 때문에, `deletion` 에서 `is_half_for_leaf==False` 인 경우의 구현과 비슷하다.

- 따라서, leaf node가 안정적이지 않을 때 **좌우에서 빌려오거나 그럴 수 없다면 Merge 하는 방식을 그대로 사용한다.**
  - 이때, 빌려오는 과정에서는 기존과 다르게 Child pointer를 정확히 유지하기 위해서 **회전하는 방식**을 사용한다.
  - **가져오고자 하는 Key를 구분하던 상위 Node의 Key를 하위 level로 내리고 가져오고자 하는 Key를 상위 level**로 올린다.

```
# Merge 이후, parent가 underflow라면 처리
def check_parent(self, node):
    if node is None:
        return

    print("Parnet rebuilding")
    # Root node인 경우
    if node.id == 0:
        # Root가 비면 child를 위로 올려야 됨
        if node.m == 0 and node.r is not None:
            removed_id = node.id  # 0
            self.nodes.remove(node)
                        # Root 제거: 모든 Node의 각 Field를 -1씩
            for n in self.nodes:
                if n.id is not None and n.id > removed_id:
                    n.id -= 1

            for n in self.nodes:
                if n.is_leaf:
                    if n.next_leaf_id is not None and n.next_leaf_id > removed_id:
                        n.next_leaf_id -= 1
                    if n.r is not None and n.r > removed_id:
                        n.r -= 1
                else:
                    for i, (k, v) in enumerate(n.p):
                        if v is not None and v > removed_id:
                            n.p[i] = (k, v - 1)
                    if n.r is not None and n.r > removed_id:
                        n.r -= 1
```

```
            # id 정렬
            self.set_id()

        return

    # Parent 찾기
    parent, parent_index = self.find_parent(node)
    if parent is None:
        print("Merge에서 일치하는 Parent를 찾을 수 없습니다.")
        return

    # 현재 node의 왼쪽, 오른쪽 자식
    left, right = self.find_sibling(node, parent, parent_index)



    # 1. Left에서 가져오기
    if left is not None and left.m > (self.b - 1) // 2:
        print("Parent rebuilding: Left")
        lk, lv = left.p.pop(-1)
        old_left_r = left.r
        left.r = lv
        left.m = left.m - 1

        # parent의 key를 받아온다.
        # Left의 마지막 Key에 대응되는 Internal key
        if parent_index > 0 and parent_index - 1 < len(parent.p):
            parent_key, _ = parent.p[parent_index - 1]
            # parent_key를 현재 node로 내린다.
            # 부모 internal를 아래 노드로 내릴 때, 그 left_child는 원래 left의 r로 표현되는 자식이여야 한다.
            node.p.insert(0, (parent_key, old_left_r))
            # 부모의 internal를 왼쪽에서 가져온 키로 교체
                    # 오른쪽으로 회전한다고 생각하면 편하다.
            parent.p[parent_index - 1] = (lk, parent.p[parent_index - 1][1])
        else:
            # parent.r이 node를 가리키는 경우
            node.p.insert(0, (lk, old_left_r))
```

```
            node.m = node.m + 1
            return

        # 2. Right에서 가져오기
        elif right is not None and right.m > (self.b - 1) // 2:
            print("Parent rebuilding: Right")
            rk, rv = right.p[0]
            del right.p[0]
            right.m = right.m - 1

            # Right의 첫번째 key를 parent의 parent의 key로 바꾼다.
            if parent_index < len(parent.p):
                parent_key, _ = parent.p[parent_index]
                # 부모 node의 key를
                node.p.append((parent_key, node.r))
                node.r = rv
                # 부모의 internal를 오른쪽에서 가져온 키로 교체
                # 왼쪽으로 회전한다고 생각하면 편하다.
                parent.p[parent_index] = (rk, parent.p[parent_index][1])
            else:
                # parent.r이 right를 가리키는 경우
                node.p.append((rk, node.r))
                node.r = rv

            node.m = node.m + 1
            return

        # 3. Merge 필요
        if left is not None:
            print("Parent rebuilding: Merge left")
            remove_index = parent_index - 1
            merged_node = self.merge_internal_node(left, node, parent, remove
_index)
            if merged_node is not None and not merged_node.is_half_for_intern
al():
                self.check_parent(merged_node)
        elif right is not None:
            print("Parent rebuilding: Merge right")
```

```
            remove_index = parent_index
            merged_node = self.merge_internal_node(node, right, parent, remov
e_index)
            if merged_node is not None and not merged_node.is_half_for_intern
al():
                self.check_parent(merged_node)
```

`check_parent` 함수에서 Merge가 필요한 경우, `Deletion` 과 동일하게 처리하지만, Leaf node가 아니고 internal node일 것이기 때문에 `merge_internal_node` 함수를 구현한다.

- `p` 와 `p_` 사용에 차이가 있고 `r` 을 다뤄야 한다는 점이 `merge_leaf_node` 와 다르다.

```
# 두 Internal node끼리 Merge
    def merge_internal_node(self, left, right, parent, remove_index):
        if remove_index < len(parent.p):
            internal_key, _ = parent.p[remove_index]

            left.p.append((internal_key, left.r))
        else:
            # parent.r이 right를 가리키는 경우
            internal_key = None

            left.p.append((internal_key, left.r))

        # right의 모든 키를 left에 추가
        left.p.extend(right.p)
        left.r = right.r
        left.m = len(left.p)

        # parent에서 separator와 right를 가리키는 포인터 제거
        if remove_index < len(parent.p):
            del parent.p[remove_index]
        else:
            # parent.r이 right를 가리키는 경우
            if remove_index > 0 and remove_index - 1 < len(parent.p):
                del parent.p[remove_index - 1]
            # parent.r은 left.id로 업데이트
```

```
            parent.r = left.id

        parent.m = len(parent.p)

        new_p = []
        for (k, v) in parent.p:
            if v == right.id:
                new_p.append((k, left.id))
            else:
                new_p.append((k, v))
        parent.p = new_p

        if parent.r == right.id:
            parent.r = left.id

        # right를 nodes 배열에서 제거하고 ID 재정렬
        removed_id = right.id
        self.nodes.remove(right)

        # Right에 대응되는 node 제거되었기 때문에 Field를 -1씩
        for n in self.nodes:
            if n.id is not None and n.id > removed_id:
                n.id -= 1

            if n.is_leaf:
                if n.next_leaf_id is not None and n.next_leaf_id > removed_id:
                    n.next_leaf_id -= 1
                if n.r is not None and n.r > removed_id:
                    n.r -= 1
            else:
                n.p = [(k, (v - 1 if v is not None and v > removed_id else v)) for (k,
v) in n.p]
                if n.r is not None and n.r > removed_id:
                    n.r -= 1

        return parent
```

c. Single Search

현재 Key를 담고 있을 법한 Leaf node를 찾는 코드는 아래와 같다.

- `is_search` 는 `search` 함수를 Insert나 Delete에서 재사용할 때, Internal node path가 출력되지 않도록 하기 위함이다.
- Tree의 search는 하위 Level의 node로 옮겨가도록 해주고, Node의 search는 지금 해당 node에 key가 포함되는 지를 확인한다.

```
# Tree의 search
def search(self, root, key, is_search):
    node = root

    while(not node.is_leaf):
        i = node.search(key, is_search) # 다음 Node의 Index
        if i is not None and i < len(self.nodes):
            node = self.nodes[i]
        else:
            break

    return node

# Node의 search
def search(self, key, is_search):
    assert self.is_leaf == 0, "Node.search는 None leaf node에서만 호출되어야 합니다."

    # Single Search인 경우에만 Key 전부 출력
    if is_search:
        keys = [k for k, v in self.p]
        sys.stdout.write(",".join(map(str, keys)) + "\n")

    for k, v in self.p:
        if key < k:
            return v
```

이후, 찾은 Leaf node에서 일치하는 key가 있는지 찾는다.

`single_search` 는 `searchLeaf` 에서 찾은 값이 있으면 그 값을, 없다면 NOT FOUND를 출력한다.

```
# Tree의 함수
def single_search(self, key):
        if len(self.nodes) == 0:
            print("No nodes in the tree")
            return None

        root = self.get_root()

        node = self.search(root, key, True)

        value = node.searchLeaf(key)

        if value == None:
            sys.stdout.write("NOT FOUND" + "\n")
        else:
            sys.stdout.write(str(value) + "\n")

 # Node의 함수
 def searchLeaf(self, key):
        assert self.is_leaf == 1, "Node.search_in_leaf는 leaf node에서만 호출되
어야 합니다."

        for k, v in self.p_:
            if key == k:
                return v

        return None
```

d. Ranged Search

Leaf node를 찾는 과정까진 Single search와 동일하다.

`searchLeaf_for_ranged` 는 범위에 속하는 모든 (Key, Value)를 출력하고 End_key보다 큰 값을 찾지 못 했다면 next_leaf를 반환한다.

leaf node가 더 이상 없거나, end_key보다 큰 값을 찾은 경우 전 까지 `ranged_search` 함수를 통해 `searchLeaf_for_ranged` 반복해서 함수를 호출한다.

```
# Tree의 함수
def ranged_search(self, start_key, end_key):
        if len(self.nodes) == 0:
            print("No nodes in the tree")
            return None

        root = self.get_root()

        node = self.search(root, start_key, False)

        # Linked list로 Leaf node를 탐색
        while(1):
            next_id = node.searchLeaf_for_ranged(start_key, end_key)

            if next_id == None:
                break
            else:
                node = self.nodes[next_id]

  # Node의 함수
  def searchLeaf_for_ranged(self, start_key, end_key):
        assert self.is_leaf == 1, "Node.search_in_leaf는 leaf node에서만 호출되
어야 합니다."

        for k, v in self.p_:
            if start_key <= k <= end_key: # Including endpoint
                sys.stdout.write(str(k) + "," + str(v) + "\n")
            elif end_key < k: # 해당 Node에서 end_key보다 큰 값이 나왔으면 다음 N
ode에서 end_key보다 작은 Key 값이 나올 수 없다.
                return None

        return self.r # 다음 Leaf Node의 Id 반환
```

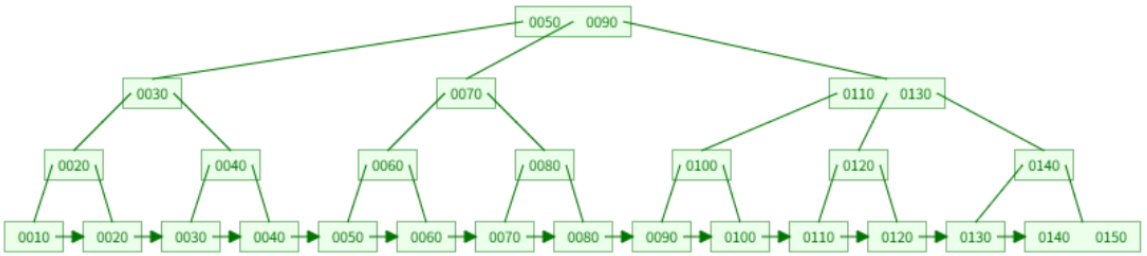
5. 테스트 결과

제공된 Data를 이용하여 **b=3~7인 경우에 모든 동작이 B+ tree simulator와 동일하게 동작하였다.**

추가적인 테스트로 b=3일 때, 10, 20, 30, ..., 150의 숫자를 순서대로 삽입하는 경우를 기준으로 살펴보자.

a. Insetion

B+ tree Simulator에 따르면 아래 사진과 같이 결과가 나와야 한다.



내 코드로 실행하고 생긴 Index.dat file의 결과를 보면 동일한 것을 확인할 수 있다.

data > index.dat

```
3
0|0|50,90|1,2,3|
1|0|30|4,5|
2|0|70|6,7|
3|0|110,130|8,9,10|
4|0|20|11,12|
5|0|40|13,14|
6|0|60|15,16|
7|0|80|17,18|
8|0|100|19,20|
9|0|120|21,22|
10|0|140|23,24|
11|1|10|1|12
12|1|20|2|13
13|1|30|3|14
14|1|40|4|15
15|1|50|5|16
16|1|60|6|17
17|1|70|7|18
18|1|80|8|19
19|1|90|9|20
20|1|100|10|21
21|1|110|11|22
22|1|120|12|23
23|1|130|13|24
24|1|140,150|14,15|
```

b. Deletion

150, 10, 60, 20, 30, 50, 80, 70, 90, 130, 120, 110, 100, 140, 40 의 순서로 delete한다.

출력이 길어 **마지막 4개 Key**의 삭제에 대한 출력만 확인해보자.

- Delete가 정상적으로 동작하는 것을 확인할 수 있다.
- Node가 root뿐인 경우에는 남은 key가 없다면 Tree에서 Node가 삭제되어 어떤 Node도 존재하지 않는 것을 확인할 수 있다.

```
------------------------------------------------

삭제할 Key: 110
Case 4
Case 4: Right만 존재하는 경우
Case 4: Merge 필요
Case 4: Right와 merge
Parnet rebuilding
Parent rebuilding: Merge left
Parnet rebuilding
Delete 110 이후 최종 상태
------------------------------------------------
0 0 [(100, 1), (140, 2)] 2 3 None
1 1 [(40, 4)] 1 2 2
2 1 [(100, 10)] 1 3 3
3 1 [(140, 14)] 1 None None
------------------------------------------------

삭제할 Key: 100
Case 4
Case 4: left, right 모두 존재하는 경우
Case 4: Merge 필요
Case 4: Left와 Right 모두 merge 가능
Delete 100 이후 최종 상태
------------------------------------------------
0 0 [(140, 1)] 1 2 None
1 1 [(40, 4)] 1 2 2
2 1 [(140, 14)] 1 None None
------------------------------------------------

삭제할 Key: 140
Case 4
Case 4: Left만 존재하는 경우
Case 4: Merge 필요
Case 4: Left와 merge
Parnet rebuilding
Delete 140 이후 최종 상태
------------------------------------------------
0 1 [(40, 4)] 1 None None
------------------------------------------------

삭제할 Key: 40
Root만 존재하는 경우
```

c. Single Search

존재하지 않는 Key인 **200**을 검색해보자.

```
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -s index.dat 200
50,90
110,130
140
NOT FOUND
```

- Leaf 까지의 Search 경로가 출력된 후, **NOT FOUND**가 출력된다.

존재하는 Key인 **100, 110**을 각각 검색해보자.

```
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -s index.dat 100
50,90
110,130
100
10
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -s index.dat 110
50,90
110,130
120
11
```

- Leaf 까지의 Search 경로가 출력된 후, 각 Key의 Value인 **10, 11**이 출력된다.

d. Ranged Search

먼저 **50~150** 사이의 Key와 **60~80** 사이의 Key를 탐색하는 것을 확인해보자.

```
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -r index.dat 50 150
50,5
60,6
70,7
80,8
90,9
100,10
110,11
120,12
130,13
140,14
150,15
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -r index.dat 60 80
60,6
70,7
80,8
```

- **양 끝 값을 포함**하여 정확히 반환하는 것을 확인할 수 있다.

이제 End의 범위가 초과되었을 때, 현재 Tree에 존재하는 Key의 범위까지의 Key만 반환하는지 확인해보자.

- 이를 위해서 **60 ~ 1000 사이 Key를 사용한다.**

```
PS C:\Users\kwond\Database_system\BPTREE\Source> python bptree.py -r index.dat 60 1000
60,6
70,7
80,8
90,9
100,10
110,11
120,12
130,13
140,14
150,15
```

- 1000까지 검색해도 **150까지만 반환**하는 것을 확인할 수 있다.

6. Compile 방법

```
PS C:\Users\kwond\Database_system\B+tree_Assignment> cd Source
PS C:\Users\kwond\Database_system\B+tree_Assignment\Source> python bptree.py -c index.dat 3
PS C:\Users\kwond\Database_system\B+tree_Assignment\Source>
```

- Insert에 사용하는 `input.csv` 파일과 Delete에 사용하는 `delete.csv` 파일은 `B+tree_Assignment/Data` 안에 위치해야 합니다.
  - **Data: 대소문자 구분 필수입니다.**
- 아래 설명은 이미 `B+tree_Assignment` 폴더에 위치한다고 가정한 설명입니다.
- 메인 소스 코드는 `B+tree_Assignment/Source` 에 위치하며 먼저 Terminal에서 `cd Source` 를 통해 해당 폴더로 이동해야 합니다.
- 이후, **별다른 컴파일 과정 없이** 명세서의 명령어 **(-c, -i, -d, -s, -r)**를 실행할 수 있습니다.
- `-c` 명령의 결과로 생기는 Index file은 `B+tree_Assignment/Source` 에 위치합니다.

7. Trouble Shooting

기본 로직보다 내가 설정한 Tree의 각 **Node field를 Insert / Delete 이후에 정확히 유지하도록 하는 작업**이 가장 어려웠다.

- 기본 로직 자체가 케이스가 다 구별되어 있었기 때문에, 디버깅을 통해 어느 부분에서 문제가 발생하는 알 수 있었다.

- 특히 문제가 발생하는 케이스에서 오른쪽 Child는 따로 다루는 if 문 등의 케이스를 처리하면서 해결했다.

8. 느낀점 및 아쉬운 점

Insert / Delete를 위해 Search가 필요하다고 생각해서 Search를 가장 먼저 구현하였다.

Search 구현은 비교적 쉽게 끝났다.

이후, 유튜브나 구글 등에서 B+ Tree algorithm 관련 자료를 많이 찾아보고, 의사 코드 기반으로 Insert / Delete 를 구현해보고자 하였다.

의사 코드 기반으로 Insert / Delete를 구현해보니 예외적인 상황에 걸리는 경우가 구분이 되지 않았고, 특히 어떤 케이스에서 문제가 발생하는지 확인할 수 없었다.

때문에, Insert / Delete를 나올 수 있는 모든 경우를 각각 구현하게 되었다.

이로 인하여 아쉬웠던 점이 있었다.

- 이 과정에서 불필요한 코드의 재사용이 많아 코드가 길어졌다.
- Node 내부를 다루는 부분은 Node class의 함수로 구현했어야 했는데 그렇지 못 했다.
- 자주 재사용되는 코드는 최적화해야 하지만, 시간적인 여유가 없어 최적화하지 못 하였다.
- 디버깅 중에 나오는 에러를 기반으로 억지로 해결한 느낌이 있다.

이후에는 중복되는 코드를 함수화하고 의사 코드 기반으로 케이스를 나누지 않고 깔끔하게 구현해보고 싶다는 생각이 들었다.

본 과제의 Insert / Delete는 케이스별로 분리하여 명확히 확인·검증한 후, 최적화 단계를 거쳐 제출할 계획이었습니다. 다만 제출 기한이 임박해 최적화 작업을 충분히 수행하지 못했습니다. 현재 구현은 모든 제공 테스트 케이스와 개인적으로 확인한 테스트 케이스를 통과하였

으나, 케이스 분리 구조와 동일한 코드의 재사용으로 인해 코드의 가독성이 다소 저하된 점 양해 부탁드립니다.