

WIKI for
B+ tree implementation assignment

컴퓨터소프트웨어학부

2022085069 손주은

Design

0. Node construction

- order: order of b+ tree
- keys[]: an array of keys
- pointers[]: an array of pointers
 - internal node: children pointer
 - leaf node: value pointer
- nextKey: leaf node에서 오른쪽 노드를 가리킴
- isLeaf: leaf node인지 아닌지를 나타내는 flag

1. Insertion

- 키를 삽입할 위치를 찾고, key, value 값을 삽입한다.
- 삽입 후 overflow가 난 경우 node를 split한다.
- Overflow 발생 조건은 key의 개수가 $\lfloor m/2 \rfloor - 1$ 이상인 경우다.

2. Deletion

- 삭제할 키의 위치를 찾고, 삭제한다.
- 삭제 후 underflow가 난 경우,
 - 형제로부터 빌려올 수 있다면 빌려오고, 그럴 수 없을 때는 merge한다.
 - 형제로부터 빌려올 수 있는 조건은 형제 노드의 키의 개수가 $\lfloor m/2 \rfloor$ 이상인 경우이다.
- 왼쪽 형제로부터 확인한다음, 오른쪽 형제를 확인한다.
- 삭제할 키가 internal node에도 있다면, internal node에서도 삭제한다.
- 단, internal node에도 있는 경우에는 flag를 남겨뒀다가 leaf node에서 먼저 삭제하고 internal node에서 삭제한다.
- Internal node의 키 삭제는 그 위치에 후임자 키를 찾아서 복사해준다.

3. Search

- 1) Search
 - Insertion, deletion할 때 key의 위치를 찾기 위한 함수로, leaf node를 return한다.
- 2) Single-key-search
 - Key가 tree 내에 있는지 없는지 판단하는 함수로, 찾아가는 경로를 출력한다.
- 3) Range search
 - 범위 내에 있는 key, value 쌍을 출력하는 함수이다.

Implementation

Basic Functions

In class Node:

```
def insert_at_leaf(self, key, pointer):  
    if self.keys:  
        for i in range(len(self.keys)):  
            if key < self.keys[i]:  
                self.keys = self.keys[:i] + [key] + self.keys[i:]  
                self.pointers = self.pointers[:i] + [pointer] + self.pointers[i:]  
                return  
        self.keys.append(key)  
        self.pointers.append(pointer)  
    else:  
        self.keys = [key]  
        self.pointers = [pointer]
```

- Leaf node에 insert하는 함수로, 오름차순 정렬로 삽입한다.
- 삽입할 노드의 키와 하나씩 비교해가면서 값을 집어넣는다. 노드의 키들보다 삽입할 키의 값이 더 크면 맨 뒤에 append한다. 배열에 아무것도 없는 상태면 그냥 집어넣는다.

In class bptree:

```
def __init__(self, order):  
    self.root = Node(order)  
    self.root.isLeaf = True  
    self.order = order
```

- B+ tree를 초기화하는 함수이다.
- root 노드를 만들고, isLeaf를 True로 바꾼다. order에도 입력 받은 값을 넣어준다.

```
def search(self, key):  
    current_node = self.root  
  
    flagged_node = None # internal node flag  
  
    while not current_node.isLeaf: # insert or delete 할 '리프노드' 찾기  
        for i in range(len(current_node.keys)):  
            if key < current_node.keys[i]:  
                if key in current_node.keys:  
                    flagged_node = current_node # 삭제할 때 삭제할 key를 갖고있는 internal node를 저장  
                    current_node = current_node.pointers[i]  
                    break  
        else: # 현재 노드의 모든 키 보다 큰 경우  
            if key in current_node.keys:  
                flagged_node = current_node  
                current_node = current_node.pointers[-1]  
  
    return current_node, flagged_node # return leaf node, internal node
```

- Insertion, deletion할 때 위치를 찾는 함수로, leaf node를 return한다.
- 리프노드가 될 때까지 반복문을 돌면서 삽입할 혹은 삭제할 위치를 찾는다. 그 과정에서 flagged_node를 기록하는데, 이건 delete할 때 사용할 값으로, internal node에 삭제할 키 값이 있는지 없는지를 저장한다.
- 따라서 current_node 즉, 원래 찾고자 했던 위치와 flagged_node를 같이 return한다.

```
def insert(self, key, pointer):

    old_node, _ = self.search(key)
    old_node.insert_at_leaf(key, pointer)

    # overflow가 발생한 경우
    if len(old_node.keys) > old_node.order - 1:
        self.split_node(old_node)
```

- bptree에 insert하는 함수이다.
- search함수로 삽입할 위치를 찾아서 old_node에 저장한다.
- old_node에 key,pointer 값을 insert 한다.
- 만약 insert한 후 overflow가 발생했을 경우, node를 split한다.

```
def split_node(self, old_node):
    new_node = Node(old_node.order)
    new_node.isLeaf = old_node.isLeaf

    mid = len(old_node.keys) // 2
    if old_node.isLeaf:
        # leaf node일 때
        # 부모로 올릴 키 값을 오른쪽 자식의 첫번째 키에 저장해야함.
        # split 한 후, nextKey 조정
        new_node.keys = old_node.keys[mid:]
        new_node.pointers = old_node.pointers[mid:]
        old_node.keys = old_node.keys[:mid]
        old_node.pointers = old_node.pointers[:mid]

        new_node.nextKey = old_node.nextKey
        old_node.nextKey = new_node

        parent_key = new_node.keys[0]
    else:
        # internal node일 때
        # 부모로 올릴 키 값을 오른쪽 자식에 저장할 필요 없음.
        new_node.keys = old_node.keys[mid + 1:]
        new_node.pointers = old_node.pointers[mid + 1:]
        parent_key = old_node.keys[mid]
        old_node.keys = old_node.keys[:mid]
        old_node.pointers = old_node.pointers[:mid + 1]

        for pointer in new_node.pointers:
            if pointer is not None:
                pointer.parent = new_node

    # 부모로 insert
    self.insert_into_parent(old_node, new_node, parent_key)
```

- Overflow가 발생했을 경우, 해결해주는 함수이다.
- 기존의 노드를 쪼개서 새로운 노드에 나눈다.
- Leaf node일 경우, 부모로 올릴 키 값을 오른쪽 자식의 첫번째 키에 저장한다. 따라서 new_node에 old_node의 mid 번째 키부터 집어넣는다. 포인터도 마찬가지로 nextKey도 적절히 조정해준다.
- Internal node는 부모로 올릴 키 값을 저장할 필요가 없기 때문에 mid값은 old_node, new_node 어디에도 저장되지 않는다. 따라서 new_node에 mid+1 번째 키부터 집어넣는다. 포인터도 마찬가지로.
- 노드를 쪼개 후엔 parent노드를 삽입해준다.

```
def insert_into_parent(self, old_node, new_node, key):
    parent = old_node.parent
    if parent is None:
        new_root = Node(old_node.order)
        new_root.keys = [key]
        new_root.pointers = [old_node, new_node]
        old_node.parent = new_root
        new_node.parent = new_root
        self.root = new_root
        return

    for i in range(len(parent.keys)):
        if key < parent.keys[i]:
            parent.keys = parent.keys[:i] + [key] + parent.keys[i:]
            parent.pointers = parent.pointers[:i + 1] + [new_node] + parent.pointers[i + 1:]
            new_node.parent = parent
            break
    else:
        parent.keys.append(key)
        parent.pointers.append(new_node)
        new_node.parent = parent

    # 부모로 insert한 후에도 overflow가 나는지 확인
    if len(parent.keys) > parent.order - 1:
        self.split_node(parent)
```

- split_node를 수행한 후에 노드들의 관계를 조정해주는 함수다.
- Key는 parent에 삽입할 키로, 자식들로부터 승격된 key를 의미한다.
- 이때도 오름차순으로 정렬하면서 삽입한다.
- 만약 부모 노드가 없는 경우 새로운 루트 노드를 만들어서 key, pointer값을 집어넣고 자식 노드들의 parent를 새로운 노드로 지정한다.
- 부모를 insert한 후에도 overflow가 발생하는지 확인하고, 만약 발생한다면 노드를 split한다.

```

def delete(self, key):
    # leaf node에서 먼저 삭제
    leaf_node, flagged_node= self.search(key)

    if key not in leaf_node.keys:
        print(f"Key {key} not found for deletion.")
        return False

    index = leaf_node.keys.index(key)
    leaf_node.keys.pop(index)
    leaf_node.pointers.pop(index)

    #underflow 발생 시 처리
    if len(leaf_node.keys) < math.ceil(self.root.order / 2) - 1:
        #print(f"Underflow after delete {key}")
        self.handle_underflow(leaf_node, key)

    #internal node에 key가 있는지 확인하고 삭제
    if flagged_node is not None:
        self.delete_from_internal_nodes(flagged_node, key)
    return True

```

- Key를 받아서 해당 키를 tree에서 삭제하는 함수다.
- Leaf node와 Internal node 모두에 삭제할 키가 있는지 확인하고, leaf node부터 삭제한다.
- Key를 삭제한 후에 underflow가 발생했는지 검사한다.
- Underflow를 해결한 후에, internal node의 키를 삭제해야 한다. (internal node에 삭제할 키가 있는 경우)

```

def delete_from_internal_nodes(self, node, key):
    if key in node.keys:
        #print(f"Deleting key {key} from internal node: {node.keys}")
        # 후임자랑 위치를 바꾸고
        # 삭제
        index = node.keys.index(key)

        successor= self.find_min(node.pointers[index+1])

        node.keys[index] = successor

        # underflow 발생 시 처리
        if len(node.keys) < math.ceil(self.root.order/2)-1:
            self.handle_underflow(node, key)

```

- Internal node에 삭제할 키가 존재하는 경우, 이 함수를 통해서 삭제한다.
- 후임자와 위치를 바꾸고 삭제한다. 사실상, 후임자를 찾아서 그 값을 삭제할 key의 위치

에 복사해주는 것과 동일하다.

```
def find_min(self, node):

    # to find sucesor
    # 인자를 받을 때 후임자가 필요한 노드의 오른쪽 자식을 넣어줘야 함.
    current_node= node
    while not current_node.isLeaf:
        if not current_node.pointers:
            raise IndexError("Node pointeres are empty in find_min")
        current_node= current_node.pointers[0] # 계속 왼쪽 노드로

    if not current_node.keys:
        raise IndexError("Leaf node keys are empty in find_min")
    return current_node.keys[0] # 현재 키 값들보다 큰 수 중에서 가장 작은 수
```

- 후임자를 찾는 함수다.
- 인자를 받을 때 후임자가 필요한 노드의 오른쪽 자식을 넣어줘야 한다 !!
- 리프 노드에 도달할 때까지 반복문을 수행하며 계속 왼쪽 자식으로 내려간다.
- 리프 노드에 도달했다면 그 노드의 가장 작은 키 값을 return 한다.
- 이것은 현재 키보다 큰 수들 중에서 가장 작은 수를 찾는 것을 의미한다.

```
def handle_underflow(self, node, key=None):
    parent = node.parent
    if not parent:
        if len(node.keys) == 0:
            if node.isLeaf:
                self.root = None
            else:
                # 왼쪽, 오른쪽 자식이 모두 살아있는 경우
                if len(node.pointers) == 2:
                    smallest_key = self.find_min(node.pointers[1])
                    #print(f"smallest key: {smallest_key}")
                    self.root = node.pointers[0]
                    self.root.parent = None
                    self.insert_into_parent(self.root, node.pointers[1], smallest_key)
                    #node.keys.append(smallest_key)
                elif node.pointers[0]:
                    self.root = node.pointers[0]
                    self.root.parent = None
                elif node.pointers[1]:
                    self.root = node.pointers[1]
                    self.root.parent = None
                else:
                    raise IndexError("Invalid node structure, no child pointers")
        return

    try:
        index = parent.pointers.index(node)
    except ValueError:
        return # 포인터 리스트에서 찾지 못한 경우 중단
```

- 이 함수는 길어서 나누어서 설명하겠다.
- Delete를 한 후에 underflow가 발생했을 경우 처리해주는 함수다.
- 부모의 정보가 필요하기 때문에 부모를 저장해주고, 부모가 없다면 경우에 따라 다르게 처리해줘야 한다.
 - 부모도 없고, node가 leaf node라면 트리가 비어 있는 상태이다.
 - 만약 왼쪽, 오른쪽 자식 포인터가 모두 살아있는 경우, 후임자를 찾아서 루트 노드에 넣어준다.
 - 왼쪽만 살아있거나 오른쪽만 살아있는 경우, 그 살아있는 자식을 루트로 승격시키고, parent는 None으로 설정해준다.
 - 나머지는 Error이다.
- Index를 저장한다.

```
# 왼쪽 형제에서 빌려오기 시도
if index > 0:
    left_sibling = parent.pointers[index - 1]
    if len(left_sibling.keys) > math.ceil(self.root.order / 2) - 1:

        if node.isLeaf:

            #node.keys.insert(0, parent.keys[index-1])
            node.keys.insert(0, left_sibling.keys[-1])
            node.pointers.insert(0, left_sibling.pointers[-1])

            breakpoint
            parent.keys[index-1] = node.keys[0]
            left_sibling.keys.pop(-1)
            left_sibling.pointers.pop(-1)

        else:
            node.keys.insert(0, parent.keys[index-1])
            node.pointers.insert(0, left_sibling.pointers[-1])

            left_sibling.pointers.pop(-1)
            parent.keys[index-1] = left_sibling.keys.pop(-1)

    if len(parent.keys) < math.ceil(self.root.order / 2) - 1:

        self.handle_underflow(parent)

    return
```

- 왼쪽 형제에서부터 빌려 오기를 시도한다.
- Index가 0보다 크다는 건 가장 왼쪽 노드가 아님을 의미한다.
- left_sibling의 key개수를 확인해서 빌려줄 수 있다면 빌려준다.
- Leaf node라면, node에 left_sibling의 마지막 key와 pointer를 삽입 → parent key를 node.keys[0]으로 업데이트 → left_sibling에서 key와 pointer pop
- Internal node라면, node에 부모의 키와 , left_sibling의 마지막 pointer를 받아옴 → left_sibling의 마지막 포인터 pop → parent의 key를 left_sibling의 마지막 key로 업데이트, 동시에 left_sibling에서 pop


```

# 오른쪽 형제에서 빌려오기 시도
if index < len(parent.pointers) - 1:
    right_sibling = parent.pointers[index + 1]
    if len(right_sibling.keys) > math.ceil(self.root.order / 2) - 1:

        if node.isLeaf:
            node.keys.append(parent.keys[index])
            node.pointers.append(right_sibling.pointers[0])
            right_sibling.pointers.pop(0)
            right_sibling.keys.pop(0)

            parent.keys[index] = right_sibling.keys[0]
        else:
            node.keys.append(parent.keys[index])
            node.pointers.append(right_sibling.pointers[0])
            right_sibling.pointers.pop(0)
            parent.keys[index] = right_sibling.keys.pop(0)

    if len(parent.keys) < math.ceil(self.root.order / 2) - 1:
        self.handle_underflow(parent)
    return

self.merge(node, index, key)
# 부모 노드에서도 underflow 발생 시 처리
if len(parent.keys) < math.ceil(self.root.order / 2) - 1:
    self.handle_underflow(parent)

```

- 오른쪽 형제에서 빌리는 경우도 개념은 동일하다.
- Leaf node인 경우, 부모의 키와 오른쪽 형제의 첫번째 포인터를 받아옴 → right_sibling에서 key와 pointer pop → parent key를 오른쪽 형제의 첫번째 key로 업데이트
- Internal node인 경우, 나머지는 동일한데 부모의 key를 업데이트 해주는 부분만 차이가 있다. 이때는 부모의 key가 오른쪽 자식의 첫번째 key에 저장될 필요가 없으므로 부모를 업데이트 함과 동시에 right_sibling 에서 pop 해버리면 된다.
- 못 빌릴 경우에는 merge한다.
- Underflow를 해결한 후, 부모 노드에서도 underflow가 발생했는지 확인하고, 처리한다.

```

def merge(self, node, index, key=None):
    parent = node.parent
    if index > 0:
        # left_sibling 에 node를 합치는 경우
        left_sibling = parent.pointers[index - 1]

        if parent.keys[index-1] not in left_sibling.keys:
            # parent의 키가 left_sibling에 없을 때만
            if key is not None and parent.keys[index-1] == key:
                # parent의 키가 삭제할 key와 같으면 부모로부터 빌려오지 않음
                #print('skip') # 말에서 삭제해줌
                pass
            else:
                left_sibling.keys.append(parent.keys[index-1])

        for key in node.keys:
            if key not in left_sibling.keys:
                left_sibling.keys.append(key)

        left_sibling.pointers.extend(node.pointers)

        del parent.keys[index - 1]
        del parent.pointers[index]

        if not node.isLeaf:
            for child in node.pointers:
                if isinstance(child, Node):
                    child.parent = left_sibling
        else:
            left_sibling.nextKey = node.nextKey

```

- merge함수도 길어서 나누어 설명하겠다.
- Left_sibling에 node를 합치는 경우와 node에 right_sibling을 합치는 경우로 나눌 수 있다.
- 기본적으로 merge할 때는 부모의 키를 빌려온 후, 형제 노드와 합친다. 다만, 부모 노드의 키가 삭제할 키와 같은 경우에는 부모로부터 빌리지 않는다.
- 형제와 키, 포인터를 합친 후에 부모에서 key와 pointer를 없애준다.
- 합쳐진 노드가 리프 노드였을 경우, nextKey를 조정하고 중간 노드였을 경우, parent를 조정해준다.

```

else:
    right_sibling = parent.pointers[index + 1]

    if parent.keys[index] not in node.keys:
        if parent.keys[index] == key:
            #print('skip') # 밑에서 삭제해줌
            pass
        else:
            node.keys.append(parent.keys[index])

    for key in right_sibling.keys:
        if key not in node.keys:
            node.keys.append(key)

    node.pointers.extend(right_sibling.pointers)

    del parent.keys[index]
    del parent.pointers[index + 1]

    if not node.isLeaf:
        for child in node.pointers:
            if isinstance(child, Node):
                # 부모 설정
                child.parent = node
    else:
        # nextKey 설정
        node.nextKey = right_sibling.nextKey

```

- Node에 right_sibling을 합치는 경우이다.
- 위의 코드와 동일한 기능을 수행한다.

```

def print_tree(self, node, level=0):
    indent = "  " * level
    print(f"{indent}Level {level} Keys: {node.keys}")

    if node.isLeaf:
        # Leaf node 출력 형태
        print(f"{indent}Pointers: {node.pointers}")
    else:
        # Internal node 출력 형태 (루트도 여기 포함..)
        for i, child in enumerate(node.pointers):
            print(f"{indent}Children {i}: {id(child)}")
            if child:
                self.print_tree(child, level+1)

```

- Tree를 출력하는 함수다.
- Leaf node는 key, value 쌍으로 출력하고, internal node는 key 집합만 출력한다.
- 편의상, root node도 internal node로 출력되게 했다.

```

def range_search(self, start, end):
    current_node, _ = self.search(start)

    while current_node is not None:
        for i in range(len(current_node.keys)):
            if start <= current_node.keys[i] <= end:
                # key, value 쌍으로 출력
                print(f"{current_node.keys[i]}, {current_node.pointers[i]}")
            elif current_node.keys[i] > end:
                return

        current_node = current_node.nextKey

```

- 범위 내에 있는 key, value 쌍들을 출력해주는 함수다.

```

def single_key_search(self, key):
    current_node = self.root

    while not current_node.isLeaf:
        print(", ".join(map(str, current_node.keys))) # 경로 출력
        found = False
        for i in range(len(current_node.keys)):
            if key < current_node.keys[i]:
                current_node = current_node.pointers[i] # to the left child
                found = True
                break
        if not found:
            current_node = current_node.pointers[-1]

        print(", ".join(map(str, current_node.keys)))
        for i in range(len(current_node.keys)):
            if key == current_node.keys[i]:
                print(current_node.pointers[i])

        return True

    print("NOT FOUND")
    return False

```

- 특정 key를 찾는 함수다.
- Key를 찾아가는 경로를 출력한다.

File I/O Functions

```
def create_index_file(index_file, node_size):
    try:
        with open(index_file, 'w') as f:
            f.write(f"B+ Tree Index File - Node Size: {node_size}\n")

            bpt = bptree(node_size) # 루트만 만들어서
            save_tree_node(f, bpt.root) # 파일에 저장
            print(f"Created index_file {index_file} with node size {node_size}.")
    except Exception as e:
        print(f"Failed to create the file: {e}")
```

- Index_file을 생성하는 함수다.
- 입력받은 index_file에 "B+ Tree Index File - Node Size: order"를 write하고 그 결과를 저장한다.

```
index.txt
1 B+ Tree Index File - Node Size: 3
2 Leaf Node Key-Value Pairs: []
3
```

- Create_index_file을 실행하고 나면 이렇게 저장된다.

```
def insert_from_csv(bpt, input_file, data_file):
    #csv파일로부터 읽어서 bptree 저장
    with open(data_file, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            if len(row) == 2:
                key, value = int(row[0]), int(row[1])
                bpt.insert(key,value)
    save_tree_to_index_file(bpt, input_file) #
```

- Data_file로부터 읽은 데이터를 input_file에 저장하는 함수다.
- Key, value값을 읽어서 tree에 insert를 하고, 그 결과를 저장한다.

```
index.txt
1 B+ Tree Index File - Node Size: 3
2 Internal Node Keys: [26]
3 Internal Node Keys: [10]
4 Leaf Node Key-Value Pairs: [(9, 87632)]
5 Leaf Node Key-Value Pairs: [(10, 84382), (20, 57455)]
6 Internal Node Keys: [68, 86]
7 Leaf Node Key-Value Pairs: [(26, 1290832), (37, 2132)]
8 Leaf Node Key-Value Pairs: [(68, 97321), (84, 431142)]
9 Leaf Node Key-Value Pairs: [(86, 67945), (87, 984796)]
10
```

- Insert한 결과이다.
- 이때, 편의상 루트 노드도 internal node로 출력되게 했다. (출력만 이렇다.)

```
def save_tree_to_index_file(bpt, index_file):
    with open(index_file, 'w') as file:
        file.write(f"B+ Tree Index File - Node Size: {bpt.order}\n")

        save_tree_node(file, bpt.root)
```

- Tree를 index file에 저장하는 함수다.

```
def save_tree_node(file, node, level=0):
    indent = "  " * level
    if node.isLeaf:
        file.write(f"{indent}Leaf Node Key-Value Pairs: {[k,p] for k,p in zip(node.keys, node.pointers)}\n")

    else:
        file.write(f"{indent}Internal Node Keys: {node.keys}\n")
        for child in node.pointers:
            if child:
                save_tree_node(file, child, level+1)
```

- 각각의 노드를 저장하는 함수다.
- 형식에 맞춰서 트리 구조를 파일에 쓰는 역할을 한다.
- Internal node일 때는 자식을 재귀적으로 저장한다.

```
def load_tree_from_index_file(index_file):

    with open(index_file, 'r') as file:
        lines = file.readlines()

        if "Node Size" in lines[0]:
            order = int(lines[0].split(":")[1].strip())
        else:
            raise ValueError("Index file is missing node size information.")

        bpt = bptree(order)

        root = parse_tree_nodes(lines[1:], order)
        #print(root is None)
        #print(root.key)
        bpt.root =root

        return bpt
```

- 기존의 index_file을 읽어 들이는 함수다.
- 첫번째 줄에서 order를 읽어서 bptree를 만들고, bpt에 저장한다.
- bpt의 루트에 parse_tree_nodes() 한 결과를 저장하고, bpt를 return한다.

```
def parse_tree_nodes(lines, order):
    # 텍스트 형태의 트리를 실제 트리로 변환해주는 함수

    root = None
    parent_stack = []
    current_level = -1
    prev_leaf = None

    for line in lines:
        indent_level = line.count(" ") # 들여쓰기를 통해 레벨 파악
        line_content = line.strip()
        node = None

        if "Leaf Node Key-Value Pairs:" in line_content:
            pairs = eval(line_content.split("Leaf Node Key-Value Pairs:")[1].strip()) # key-value 쌍을 리스트로 변환
            node = Node(order)
            node.keys = [k for k, _ in pairs]
            node.pointers = [p for _, p in pairs]
            node.isLeaf = True

            if prev_leaf is not None:
                prev_leaf.nextKey = node
            prev_leaf = node

        elif "Internal Node Keys:" in line_content:
            keys = eval(line_content.split("Internal Node Keys:")[1].strip()) # 키 값을 리스트로 변환
            node = Node(order)
            node.keys = keys
            node.isLeaf = False

        else:
            print(f"Unknown line format, skipping: {line_content}")
            continue
```

- 이 함수도 길어서 나누어 설명하겠다.
- 이 함수는 텍스트 형태의 트리를 실제 트리로 변환해주는 함수다.
- Index_file로부터 순서대로 읽는다.
- 들여쓰기를 통해 레벨을 파악하고, 루트 노드인지, 중간 노드인지, 리프 노드인지 알 수 있다.
- 라인을 읽었을 때, leaf node이면, key, value pair를 저장한다.
- Leaf Node Key-Value Pairs: 다음에 적힌 부분을 리스트로 만들어서 pairs라는 변수에 저장한다.
- 각각 노드의 key, pointers 배열에 쪼개서 넣어주고, isLeaf는 True로 설정한다.
- nextKey도 업데이트 해준다.

```
elif "Internal Node Keys:" in line_content:
    keys = eval(line_content.split("Internal Node Keys:")[1].strip()) # 키 값을 리스트로 변환
    node = Node(order)
    node.keys = keys
    node.isLeaf = False

else:
    print(f"Unknown line format, skipping: {line_content}")
    continue
```

- internal node인 경우라면, key만 저장한다.
- 둘 다 아닌 경우에는 "Unknown line format , skipping: {}"문구를 출력한다.

```
if node is not None:
    # 부모 자식 관계 설정
    if indent_level == current_level + 1:
        # 새로운 자식 노드
        if parent_stack:
            node.parent = parent_stack[-1] # 부모 설정 후
            parent_stack[-1].pointers.append(node) # 부모에 node 추가
```

- Node의 indent_level에 따라 처리 방식이 달라진다.
- 이 경우에는 직전의 노드와 부모-자식 간의 관계다.
- 따라서 노드의 parent를 넣어주고, 부모 스택에 노드를 추가한다.

```
elif indent_level == current_level:
    # 형제 노드

    if parent_stack:
        parent_stack.pop()

        parent_stack[-1].pointers.append(node)
        node.parent = parent_stack[-1]
```

- 이 경우는 직전의 노드와 형제 관계다.
- 따라서 부모 스택에 있던 값을 하나 뺀 후, 부모 스택의 마지막 값을 부모로 설정한다.
- 그리고 부모의 포인터에 node를 append한다.

```
elif indent_level < current_level:
    # 부모로 돌아가서 새로운 노드 추가
    while len(parent_stack) > indent_level:
        parent_stack.pop()
    if parent_stack:

        parent_stack[-1].pointers.append(node)
        node.parent = parent_stack[-1]
```

- 이 경우는 부모로 돌아가서 새로운 노드를 추가하는 상황이다.

```
2 Internal Node Keys: [26]
3 Internal Node Keys: [10]
4 Leaf Node Key-Value Pairs: [(9, 87632)]
5 Leaf Node Key-Value Pairs: [(10, 84382), (20, 57455)]
6 Internal Node Keys: [68, 86]
7 Leaf Node Key-Value Pairs: [(26, 1290832), (37, 2132)]
8 Leaf Node Key-Value Pairs: [(68, 97321), (84, 431142)]
9 Leaf Node Key-Value Pairs: [(86, 67945), (87, 984796)]
10
```

- 5 → 6번 라인으로 넘어갈 때와 같은 상황이다.
- 부모 스택에서 pop한 후, 값이 여전히 존재하면 마지막 값의 포인터에 노드를 추가하고 부모를 업데이트한다.

```
parent_stack.append(node)
current_level = indent_level

if root is None: # 제일 처음에만 이 조건문 실행.
    root = node

return root
```


- 부모 스택에 노드를 추가하고, current_level을 조정한다.
- 마지막 if문은 제일 처음에만 실행된다.
- Root를 return 한다.

```
def delete_from_index_file(bpt,input_file, data_file):
    with open(data_file, 'r') as file:
        for line in file:
            key = int(line.strip())
            bpt.delete(key)

    save_tree_to_index_file(bpt, input_file) # delete 한 후 파일에 저장
```

- 이 함수는 data_file에 저장된 key들을 index_file 트리에서 삭제하는 함수다.
- Key를 차례대로 삭제한 다음, 그 결과를 index_file에 다시 저장해준다.

Run&Result

Command Line 명령어

- 1) Data File Creation

```
% python bptree.py -c index.txt 3
```

- 2) Insertion

```
% python bptree.py -i index.txt input.csv
```

- 3) Single key search

```
% python bptree.py -s index.txt 26
```

- 4) Range search

```
% python bptree.py -r index.txt 60 85
```

- 5) Deletion

```
% python bptree.py -d index.txt delete.csv
```

Output

1) Data File Creation

```
Created index_file index.txt with node size 3.
```

[Terminal Output]

```
≡ index.txt
1  B+ Tree Index File - Node Size: 3
2  Leaf Node Key-Value Pairs: []
3  |
```

[Index File]

2) Insertion

```
≡ index.txt
1  B+ Tree Index File - Node Size: 3
2  Internal Node Keys: [26]
3  | Internal Node Keys: [10]
4  |   Leaf Node Key-Value Pairs: [(9, 87632)]
5  |   Leaf Node Key-Value Pairs: [(10, 84382), (20, 57455)]
6  | Internal Node Keys: [68, 86]
7  |   Leaf Node Key-Value Pairs: [(26, 1290832), (37, 2132)]
8  |   Leaf Node Key-Value Pairs: [(68, 97321), (84, 431142)]
9  |   Leaf Node Key-Value Pairs: [(86, 67945), (87, 984796)]
10 |
```

[Index File]

3) Single key search

```
Searching for 26 in index.txt
26
68,86
26,37
1290832
```

[Terminal output]

- 출력결과는 [26](root)→[68,86]→[26,37]→ value pointer [] 를 의미한다.

4) Range search

```
Performing range search from 60 to 85
68,97321
84,431142
```

[Terminal Output]

- 출력결과는 range 내에 있는 [key,value] pair를 의미한다.

5) Deletion

```
Deleting data from index.txt
```

[Terminal Output]

```
≡ index.txt
1  B+ Tree Index File - Node Size: 3
2  Internal Node Keys: [68, 86]
3  | Leaf Node Key-Value Pairs: [(37, 2132)]
4  | Leaf Node Key-Value Pairs: [(68, 97321), (84, 431142)]
5  | Leaf Node Key-Value Pairs: [(86, 67945), (87, 984796)]
6  |
```

[Index File]

1,000,000개 Test Result

1) Insert

```
index.txt
1 B+ Tree Index File - Node Size: 25
2 Internal Node Keys: [77481, 156739, 225746, 287954, 359214, 424607, 478806, 536625, 666368, 777519, 837872, 894532]
3 Internal Node Keys: [5714, 12140, 18440, 24643, 28186, 31711, 35229, 38853, 42994, 46874, 53348, 56647, 60199, 66553, 71621]
4 Internal Node Keys: [221, 456, 687, 918, 1144, 1350, 1579, 1820, 2233, 2650, 3047, 3433, 3688, 3933, 4188, 4443, 4698, 4953, 5208, 5463, 5718, 5973, 6228, 6483, 6738, 6993, 7248, 7503, 7758, 8013, 8268, 8523, 8778, 9033, 9288, 9543, 9798, 10053, 10308, 10563, 10818, 11073, 11328, 11583, 11838, 12093, 12348, 12603, 12858, 13113, 13368, 13623, 13878, 14133, 14388, 14643, 14898, 15153, 15408, 15663, 15918, 16173, 16428, 16683, 16938, 17193, 17448, 17703, 17958, 18213, 18468, 18723, 18978, 19233, 19488, 19743, 20000]
5 Internal Node Keys: [19, 40, 61, 77, 92, 110, 124, 141, 155, 168, 187, 206]
6 Leaf Node Key-Value Pairs: [(1, 5), (2, 2), (3, 95), (4, 51), (5, 34), (6, 99), (7, 20), (8, 100), (9, 101), (10, 102), (11, 103), (12, 104), (13, 105), (14, 106), (15, 107), (16, 108), (17, 109), (18, 110), (19, 111), (20, 112), (21, 113), (22, 114), (23, 115), (24, 116), (25, 117)]
7 Leaf Node Key-Value Pairs: [(19, 92), (20, 71), (21, 21), (22, 23), (23, 99), (24, 64), (25, 100), (26, 101), (27, 102), (28, 103), (29, 104), (30, 105), (31, 106), (32, 107), (33, 108), (34, 109), (35, 110), (36, 111), (37, 112), (38, 113), (39, 114), (40, 115), (41, 116), (42, 117), (43, 118), (44, 119), (45, 120), (46, 121), (47, 122), (48, 123), (49, 124), (50, 125), (51, 126), (52, 127), (53, 128), (54, 129), (55, 130), (56, 131), (57, 132), (58, 133), (59, 134), (60, 135), (61, 136), (62, 137), (63, 138), (64, 139), (65, 140), (66, 141), (67, 142), (68, 143), (69, 144), (70, 145), (71, 146), (72, 147), (73, 148), (74, 149), (75, 150), (76, 151), (77, 152), (78, 153), (79, 154), (80, 155), (81, 156), (82, 157), (83, 158), (84, 159), (85, 160), (86, 161), (87, 162), (88, 163), (89, 164), (90, 165), (91, 166), (92, 167), (93, 168), (94, 169), (95, 170), (96, 171), (97, 172), (98, 173), (99, 174), (100, 175), (101, 176), (102, 177), (103, 178), (104, 179), (105, 180), (106, 181), (107, 182), (108, 183), (109, 184), (110, 185), (111, 186), (112, 187), (113, 188), (114, 189), (115, 190), (116, 191), (117, 192), (118, 193), (119, 194), (120, 195), (121, 196), (122, 197), (123, 198), (124, 199), (125, 200), (126, 201), (127, 202), (128, 203), (129, 204), (130, 205), (131, 206), (132, 207), (133, 208), (134, 209), (135, 210), (136, 211), (137, 212), (138, 213), (139, 214), (140, 215), (141, 216), (142, 217), (143, 218), (144, 219), (145, 220), (146, 221), (147, 222), (148, 223), (149, 224), (150, 225), (151, 226), (152, 227), (153, 228), (154, 229), (155, 230), (156, 231), (157, 232), (158, 233), (159, 234), (160, 235), (161, 236), (162, 237), (163, 238), (164, 239), (165, 240), (166, 241), (167, 242), (168, 243), (169, 244), (170, 245), (171, 246), (172, 247), (173, 248), (174, 249), (175, 250), (176, 251), (177, 252), (178, 253), (179, 254), (180, 255), (181, 256), (182, 257), (183, 258), (184, 259), (185, 260), (186, 261), (187, 262), (188, 263), (189, 264), (190, 265), (191, 266), (192, 267), (193, 268), (194, 269), (195, 270), (196, 271), (197, 272), (198, 273), (199, 274), (200, 275), (201, 276), (202, 277), (203, 278), (204, 279), (205, 280), (206, 281), (207, 282), (208, 283), (209, 284), (210, 285), (211, 286), (212, 287), (213, 288), (214, 289), (215, 290), (216, 291), (217, 292), (218, 293), (219, 294), (220, 295), (221, 296), (222, 297), (223, 298), (224, 299), (225, 300), (226, 301), (227, 302), (228, 303), (229, 304), (230, 305), (231, 306), (232, 307), (233, 308), (234, 309), (235, 310), (236, 311), (237, 312), (238, 313), (239, 314), (240, 315), (241, 316), (242, 317), (243, 318), (244, 319), (245, 320), (246, 321), (247, 322), (248, 323), (249, 324), (250, 325), (251, 326), (252, 327), (253, 328), (254, 329), (255, 330), (256, 331), (257, 332), (258, 333), (259, 334), (260, 335), (261, 336), (262, 337), (263, 338), (264, 339), (265, 340), (266, 341), (267, 342), (268, 343), (269, 344), (270, 345), (271, 346), (272, 347), (273, 348), (274, 349), (275, 350), (276, 351), (277, 352), (278, 353), (279, 354), (280, 355), (281, 356), (282, 357), (283, 358), (284, 359), (285, 360), (286, 361), (287, 362), (288, 363), (289, 364), (290, 365), (291, 366), (292, 367), (293, 368), (294, 369), (295, 370), (296, 371), (297, 372), (298, 373), (299, 374), (300, 375), (301, 376), (302, 377), (303, 378), (304, 379), (305, 380), (306, 381), (307, 382), (308, 383), (309, 384), (310, 385), (311, 386), (312, 387), (313, 388), (314, 389), (315, 390), (316, 391), (317, 392), (318, 393), (319, 394), (320, 395), (321, 396), (322, 397), (323, 398), (324, 399), (325, 400), (326, 401), (327, 402), (328, 403), (329, 404), (330, 405), (331, 406), (332, 407), (333, 408), (334, 409), (335, 410), (336, 411), (337, 412), (338, 413), (339, 414), (340, 415), (341, 416), (342, 417), (343, 418), (344, 419), (345, 420), (346, 421), (347, 422), (348, 423), (349, 424), (350, 425), (351, 426), (352, 427), (353, 428), (354, 429), (355, 430), (356, 431), (357, 432), (358, 433), (359, 434), (360, 435), (361, 436), (362, 437), (363, 438), (364, 439), (365, 440), (366, 441), (367, 442), (368, 443), (369, 444), (370, 445), (371, 446), (372, 447), (373, 448), (374, 449), (375, 450), (376, 451), (377, 452), (378, 453), (379, 454), (380, 455), (381, 456), (382, 457), (383, 458), (384, 459), (385, 460), (386, 461), (387, 462), (388, 463), (389, 464), (390, 465), (391, 466), (392, 467), (393, 468), (394, 469), (395, 470), (396, 471), (397, 472), (398, 473), (399, 474), (400, 475), (401, 476), (402, 477), (403, 478), (404, 479), (405, 480), (406, 481), (407, 482), (408, 483), (409, 484), (410, 485), (411, 486), (412, 487), (413, 488), (414, 489), (415, 490), (416, 491), (417, 492), (418, 493), (419, 494), (420, 495), (421, 496), (422, 497), (423, 498), (424, 499), (425, 500), (426, 501), (427, 502), (428, 503), (429, 504), (430, 505), (431, 506), (432, 507), (433, 508), (434, 509), (435, 510), (436, 511), (437, 512), (438, 513), (439, 514), (440, 515), (441, 516), (442, 517), (443, 518), (444, 519), (445, 520), (446, 521), (447, 522), (448, 523), (449, 524), (450, 525), (451, 526), (452, 527), (453, 528), (454, 529), (455, 530), (456, 531), (457, 532), (458, 533), (459, 534), (460, 535), (461, 536), (462, 537), (463, 538), (464, 539), (465, 540), (466, 541), (467, 542), (468, 543), (469, 544), (470, 545), (471, 546), (472, 547), (473, 548), (474, 549), (475, 550), (476, 551), (477, 552), (478, 553), (479, 554), (480, 555), (481, 556), (482, 557), (483, 558), (484, 559), (485, 560), (486, 561), (487, 562), (488, 563), (489, 564), (490, 565), (491, 566), (492, 567), (493, 568), (494, 569), (495, 570), (496, 571), (497, 572), (498, 573), (499, 574), (500, 575), (501, 576), (502, 577), (503, 578), (504, 579), (505, 580), (506, 581), (507, 582), (508, 583), (509, 584), (510, 585), (511, 586), (512, 587), (513, 588), (514, 589), (515, 590), (516, 591), (517, 592), (518, 593), (519, 594), (520, 595), (521, 596), (522, 597), (523, 598), (524, 599), (525, 600), (526, 601), (527, 602), (528, 603), (529, 604), (530, 605), (531, 606), (532, 607), (533, 608), (534, 609), (535, 610), (536, 611), (537, 612), (538, 613), (539, 614), (540, 615), (541, 616), (542, 617), (543, 618), (544, 619), (545, 620), (546, 621), (547, 622), (548, 623), (549, 624), (550, 625), (551, 626), (552, 627), (553, 628), (554, 629), (555, 630), (556, 631), (557, 632), (558, 633), (559, 634), (560, 635), (561, 636), (562, 637), (563, 638), (564, 639), (565, 640), (566, 641), (567, 642), (568, 643), (569, 644), (570, 645), (571, 646), (572, 647), (573, 648), (574, 649), (575, 650), (576, 651), (577, 652), (578, 653), (579, 654), (580, 655), (581, 656), (582, 657), (583, 658), (584, 659), (585, 660), (586, 661), (587, 662), (588, 663), (589, 664), (590, 665), (591, 666), (592, 667), (593, 668), (594, 669), (595, 670), (596, 671), (597, 672), (598, 673), (599, 674), (600, 675), (601, 676), (602, 677), (603, 678), (604, 679), (605, 680), (606, 681), (607, 682), (608, 683), (609, 684), (610, 685), (611, 686), (612, 687), (613, 688), (614, 689), (615, 690), (616, 691), (617, 692), (618, 693), (619, 694), (620, 695), (621, 696), (622, 697), (623, 698), (624, 699), (625, 700), (626, 701), (627, 702), (628, 703), (629, 704), (630, 705), (631, 706), (632, 707), (633, 708), (634, 709), (635, 710), (636, 711), (637, 712), (638, 713), (639, 714), (640, 715), (641, 716), (642, 717), (643, 718), (644, 719), (645, 720), (646, 721), (647, 722), (648, 723), (649, 724), (650, 725), (651, 726), (652, 727), (653, 728), (654, 729), (655, 730), (656, 731), (657, 732), (658, 733), (659, 734), (660, 735), (661, 736), (662, 737), (663, 738), (664, 739), (665, 740), (666, 741), (667, 742), (668, 743), (669, 744), (670, 745), (671, 746), (672, 747), (673, 748), (674, 749), (675, 750), (676, 751), (677, 752), (678, 753), (679, 754), (680, 755), (681, 756), (682, 757), (683, 758), (684, 759), (685, 760), (686, 761), (687, 762), (688, 763), (689, 764), (690, 765), (691, 766), (692, 767), (693, 768), (694, 769), (695, 770), (696, 771), (697, 772), (698, 773), (699, 774), (700, 775), (701, 776), (702, 777), (703, 778), (704, 779), (705, 780), (706, 781), (707, 782), (708, 783), (709, 784), (710, 785), (711, 786), (712, 787), (713, 788), (714, 789), (715, 790), (716, 791), (717, 792), (718, 793), (719, 794), (720, 795), (721, 796), (722, 797), (723, 798), (724, 799), (725, 800), (726, 801), (727, 802), (728, 803), (729, 804), (730, 805), (731, 806), (732, 807), (733, 808), (734, 809), (735, 810), (736, 811), (737, 812), (738, 813), (739, 814), (740, 815), (741, 816), (742, 817), (743, 818), (744, 819), (745, 820), (746, 821), (747, 822), (748, 823), (749, 824), (750, 825), (751, 826), (752, 827), (753, 828), (754, 829), (755, 830), (756, 831), (757, 832), (758, 833), (759, 834), (760, 835), (761, 836), (762, 837), (763, 838), (764, 839), (765, 840), (766, 841), (767, 842), (768, 843), (769, 844), (770, 845), (771, 846), (772, 847), (773, 848), (774, 849), (775, 850), (776, 851), (777, 852), (778, 853), (779, 854), (780, 855), (781, 856), (782, 857), (783, 858), (784, 859), (785, 860), (786, 861), (787, 862), (788, 863), (789, 864), (790, 865), (791, 866), (792, 867), (793, 868), (794, 869), (795, 870), (796, 871), (797, 872), (798, 873), (799, 874), (800, 875), (801, 876), (802, 877), (803, 878), (804, 879), (805, 880), (806, 881), (807, 882), (808, 883), (809, 884), (810, 885), (811, 886), (812, 887), (813, 888), (814, 889), (815, 890), (816, 891), (817, 892), (818, 893), (819, 894), (820, 895), (821, 896), (822, 897), (823, 898), (824, 899), (825, 900), (826, 901), (827, 902), (828, 903), (829, 904), (830, 905), (831, 906), (832, 907), (833, 908), (834, 909), (835, 910), (836, 911), (837, 912), (838, 913), (839, 914), (840, 915), (841, 916), (842, 917), (843, 918), (844, 919), (845, 920), (846, 921), (847, 922), (848, 923), (849, 924), (850, 925), (851, 926), (852, 927), (853, 928), (854, 929), (855, 930), (856, 931), (857, 932), (858, 933), (859, 934), (860, 935), (861, 936), (862, 937), (863, 938), (864, 939), (865, 940), (866, 941), (867, 942), (868, 943), (869, 944), (870, 945), (871, 946), (872, 947), (873, 948), (874, 949), (875, 950), (876, 951), (877, 952), (878, 953), (879, 954), (880, 955), (881, 956), (882, 957), (883, 958), (884, 959), (885, 960), (886, 961), (887, 962), (888, 963), (889, 964), (890, 965), (891, 966), (892, 967), (893, 968), (894, 969), (895, 970), (896, 971), (897, 972), (898, 973), (899, 974), (900, 975), (901, 976), (902, 977), (903, 978), (904, 979), (905, 980), (906, 981), (907, 982), (908, 983), (909, 984), (910, 985), (911, 986), (912, 987), (913, 988), (914, 989), (915, 990), (916, 991), (917, 992), (918, 993), (919, 994), (920, 995), (921, 996), (922, 997), (923, 998), (924, 999), (925, 1000), (926, 1001), (927, 1002), (928, 1003), (929, 1004), (930, 1005), (931, 1006), (932, 1007), (933, 1008), (934, 1009), (935, 1010), (936, 1011), (937, 1012), (938, 1013), (939, 1014), (940, 1015), (941, 1016), (942, 1017), (943, 1018), (944, 1019), (945, 1020), (946, 1021), (947, 1022), (948, 1023), (949, 1024), (950, 1025), (951, 1026), (952, 1027), (953, 1028), (954, 1029), (955, 1030), (956, 1031), (957, 1032), (958, 1033), (959, 1034), (960, 1035), (961, 1036), (962, 1037), (963, 1038), (964, 1039), (965, 1040), (966, 1041), (967, 1042), (968, 1043), (969, 1044), (970, 1045), (971, 1046), (972, 1047), (973, 1048), (974, 1049), (975, 1050), (976, 1051), (977, 1052), (978, 1053), (979, 1054), (980, 1055), (981, 1056), (982, 1057), (983, 1058), (984, 1059), (985, 1060), (986, 1061), (987, 1062), (988, 1063), (989, 1064), (990, 1065), (991, 1066), (992, 1067), (993, 1068), (994, 1069), (995, 1070), (996, 1071), (997, 1072), (998, 1073), (999, 1074), (1000, 1075), (1001, 1076), (1002, 1077), (1003, 1078), (1004, 1079), (1005, 1080), (1006, 1081), (1007, 1082), (1008, 1083), (1009, 1084), (1010, 1085), (1011, 1086), (1012, 1087), (1013, 1088), (1014, 1089), (1015, 1090), (1016, 1091), (1017, 1092), (1018, 1093), (1019, 1094), (1020, 1095), (1021, 1096), (1022, 1097), (1023, 1098), (1024, 1099), (1025, 1100), (1026, 1101), (1027, 1102), (1028, 1103), (1029, 1104), (1030, 1105), (1031, 1106), (1032, 1107), (1033, 1108), (1034, 1109), (1035, 1110), (1036, 1111), (1037, 1112), (1038, 1113), (1039, 1114), (1040, 1115), (1041, 1116), (1042, 1117), (1043, 1118), (1044, 1119), (1045, 1120), (1046, 1121), (1047, 1122), (1048, 1123), (1049, 1124), (1050, 1125), (1051, 1126), (1052, 1127), (1053, 1128), (1054, 1129), (1055, 1130), (1056, 1131), (1057, 1132), (1058, 1133), (1059, 1134), (1060, 1135), (1061, 1136), (1062, 1137), (1063, 1138), (1064, 1139), (1065, 1140), (1066, 1141), (1067, 1142), (1068, 1143), (1069, 1144), (1070, 1145), (1071, 1146), (1072, 1147), (1073, 1148), (1074, 1149), (1075, 1150), (1076, 1151), (1077, 1152), (1078, 1153), (1079, 1154), (1080, 1155), (1081, 1156), (1082, 1157), (1083, 1158), (1084, 1159), (1085, 1160), (1086, 1161), (1087, 1162), (1088, 1163), (1089, 1164), (1090, 1165), (1091, 1166), (1092, 1167), (1093, 1168), (1094, 1169), (1095, 1170), (1096, 1171), (1097, 1172), (1098, 1173), (1099, 1174), (1100, 1175), (1101, 1176), (1102, 1177), (1103, 1178), (1104, 1179), (1105, 1180), (1106, 1181), (1107, 1182), (1108, 1183), (1109, 1184), (1110, 1185), (1111, 1186), (1112, 1187), (1113, 1188), (1114, 1189), (1115, 1190), (1116, 1191), (1117, 1192), (1118, 1193), (1119, 1194), (1120, 1195
```

Trouble shooting

① Delete – merge

- merge 부분을 처음에 잘못 이해했다. merge를 하려면 우선 부모에게서 키를 빌려오고, 형제와 merge해야 하는데, 부모에게서 키를 빌려오는 부분을 빠뜨렸다.
- 부모에게서 빌려온 후, 형제와 merge하게 수정하니깐 일부 문제가 해결되었다.
- 하지만, 삭제할 키가 부모 노드의 키에 있는 경우에는 빌려오면 안된다. 그대로 빌려오면 결국 리프 노드에 그 키가 또 남게 되기 때문이다.
- 이런 경우에, flagged_node를 기록하는 방법과 후임자를 찾아서 위치를 바꾸는 방법을 사용했더니 문제가 해결되었다.
- 왼쪽 형제와 합칠 때, 오른쪽 형제와 합칠 때가 미세하게 다른데 오른쪽 편향 구조상 달라지는 부분이다.

② Delete – 형제에게서 빌려 오기

- 조건을 잘못 써서 삭제 돼야할 키가 부모 노드에서 자식 노드로 내려오면서 삭제가 안되는 문제가 있었다.
- 순서를 살짝 조정하여 해결할 수 있었다.

③ split_node

- Leaf node와 internal node를 구분하지 않고 코드를 짜서 문제가 있었다.
- 'mid' 값 포함 여부에 따라 경우를 나누어서 생각했다.

④ handle_underflow

- 경우의 수가 많아서 해결하는데 시간이 오래 걸린 부분이다.
- 루트가 비었는데 왼쪽, 오른쪽 자식 포인터가 살아있는 경우에, 후임자를 루트로 올렸다.
- 루트가 비었는데 왼쪽, 오른쪽 자식 포인터 중 하나만 살아있는 경우에, 자식 포인터를 루트로 설정해주었다.

⑤ file IO

- 10000개, 1000000개 테스트를 할 때 디버거를 써서 에러를 발견했다.
- 이때 찾아낸 error 중 하나가 위에서 언급한 ②번 에러다.
- 또 다른 하나는, parse_tree_nodes함수에서 internal node일 때, 포인터 배열을 개수만큼 초기화하는 코드를 짰었는데 이게 문제가 되었다.
- 포인터 배열에 append하는 코드가 있었는데, 이렇게 해버리면 배열의 처음부터 삽입되지 않고 초기화 된 부분 뒤부터 삽입이 돼서 구조가 이상해진다.