2018007856 강민석 BPlusTree Assignment

1. Instructions for Compiling

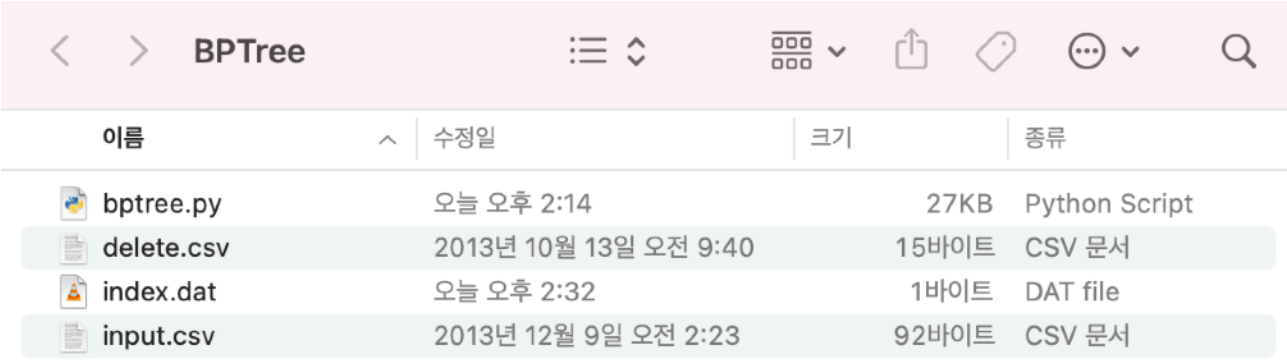
1. 소스코드(.py 파일), 삽입을 위한 input.csv, 삭제를 위한 delete.csv를 한 폴더 안에 넣어줍니다.

```
(base) Minui-MacBook-Pro-2:~ minsuk$ cd desktop/bptree
(base) Minui-MacBook-Pro-2:bptree minsuk$
```

2. 터미널에서 해당 디렉토리로 이동합니다.

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -c index.dat 3
```

3. index.dat 파일을 생성하기 위해 위와 같이 명령어를 칩니다. 과제 명세서와 같이 -c는 새로운 index 파일 생성을 의미합니다. 어떤 컴퓨터의 경우에는 python3 대신 python만 쳐도 컴파일이 될 수도 있으나 저의 경우에는 무조건 python3를 입력해야 컴파일이 됐습니다.



4. index.dat 파일이 이렇게 같은 폴더 안에 생성된 것을 볼 수 있습니다.

input.csv

```
26,1290832
10,84382
87,984796
86,67945
20,57455
9,87632
68,97321
84,431142
37,2132
```

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -i index.dat input.csv
```

5. -i는 삽입입니다. 위와 같이 입력하면 input.csv에 있는 값들이 index.dat에 삽입됩니다.

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -s index.dat 10
37
20
10
84382
```

7. -s는 탐색입니다. 위와 같이 입력하면 10을 탐색하고 value 값을 출력합니다.

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -r index.dat 10 30
10,84382
20,57455
26,1290832
```

8. -r는 범위 탐색입니다. 위와 같이 입력하면 10부터 30까지 탐색한 결과를 출력합니다.

delete.csv

```
26
10
20
9
```

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -d index.dat delete.csv
```

9. -d는 삭제입니다. 위와 같이 입력하면 delete.csv에 있는 키들을 찾아 index.dat에서 삭제합니다.

```
(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -s index.dat 10
84
68
NOT FOUND

(base) Minui-MacBook-Pro-2:bptree minsuk$ python3 bptree.py -r index.dat 10 30
NOT FOUND
```

10. 다시 탐색과 범위 탐색을 해주면 값들이 삭제되어 NOT FOUND를 출력하는 것을 볼 수 있습니다.

2. Summary of Algorithm

언어는 Python3으로 선택했고 Node Class와 BPTree Class를 만들어서 진행했습니다.

```
class Node:
    def __init__(self):
        self.num_of_keys = 0
        self.pairs = []
        self.next = None
        self.is_leaf = True
```

```
class BPTree:
    def __init__(self, degree):
        self.root = Node()
        self.degree = degree
```

Node Class에는 num_of_keys(키의 개수), pairs(키와 포인터 pair들을 갖는 리스트), next(다음 노드를 가리키는 포인터), is_leaf(leaf node인지 확인하는 bool값)들이 포함되어 있습니다.
Node가 index node인 경우에 pairs에는 키와 left child를 가리키는 포인터가 하나의 pair가 되어 들어가고, next는 rightmost child를 가리키는 포인터가 됩니다.
Node가 child node인 경우에 pairs에는 키와 value를 가리키는 포인터(input.csv로 불러오는 value값이 value를 가리키는 포인터라고 가정하였습니다)가 하나의 pair가 되어 들어가고, next는 right sibling을 가리키는 포인터가 됩니다.
BPTree Class에서는 root 변수를 만들어 초기화된 Node를 생성해 root로 지정해주고, degree 값을 받아 degree 변수에 저장해줍니다.
Node Class에 leaf node에 데이터를 삽입하는 insert_leaf 함수를 만들고 나머지 함수들은 BPTree Class에 작성하였습니다.

1) Data File Creation
Command Line Interface를 지원하기 위해 argparse 모듈을 import하여 과제 명세서에 나오는 대로 명령을 입력하면 프로그램이 작동하도록 하였습니다.
Data file creation같은 경우에는 새로운 index.dat을 만들고 index.dat에 트리의 degree(max number of child nodes)만 입력되도록 하였습니다. 이후에 insertion이 진행될 때 가장 먼저 degree부터 읽어서 Empty 트리를 생성한 후 insertion이 진행됩니다.

2) Insertion
input.csv에 있는 key,value를 한 줄씩 읽어서 BPTree Class의 insert 함수에 [key, value]로 묶여서 하나씩 insert 됩니다. [key, value] pair는 모두 leaf node에 저장되기 때문에 key값을 토대로 데이터가 저장될 leaf node를 찾아갑니다. 이 때 stack을 이용해 root부터 leaf node에 도달하기 직전 index node까지 stack에 저장합니다. Leaf node에 도달하면 Node Class의 insert_leaf 함수를 호출해 만약 키가 중복되면 바로 함수를 종료시키고 그렇지 않은 경우 해당되는 위치에 데이터를 입력하고 종료합니다. 다시 insert 함수로 돌아왔을 때 leaf node의 키의 개수가 degree와 일치하는지 확인하고 일치한다면 split_leaf 함수를 호출합니다. 이 함수는 leaf node가 꽉 찼을 때 split하는 역할을 하고, split이 되면 바로 종료합니다. 이후에 stack을 통해 leaf node의 parent node를 pop해서 만약 parent node의 키의 개수도 degree와 일치한다면 split_index 함수를 호출해 노드를 split합니다. 이런 식으로 계속해서 키의 개수가 degree와 일치하면 계속 stack을 pop하면서 split을 반복합니다. 모든 작업이 끝나고 트리가 balanced되면 insert 함수가 종료됩니다.

3) Deletion

delete.csv에 있는 key를 한 줄씩 읽어서 트리의 root와 키값을 delete_leaf 함수에 넣어줍니다. delete할 키값이 들어있는 leaf node를 재귀적으로 찾아서 들어가는데 delete할 때 parent node가 필요하기 때문에 delete_leaf 함수에 child node와 해당 node를 인자로 같이 계속 넣어줍니다. Leaf node에 도달하면 키값을 하나씩 읽으면서 delete할 키값과 일치하는 키가 있는지 확인합니다. 없다면 함수를 종료하고, 있다면 노드 안에서 그 키의 position을 pos에 저장합니다. Leaf node에서 pos값을 이용해 키값을 삭제한 후 만약 이 노드가 root가 아닌 경우에 키의 개수가 ((degree/2)를 올림한 값 - 1) 보다 작다면 merge 혹은 redistribution을 진행합니다. Left 또는 right neighbor 노드가 존재하고 neightbor 노드의 키의 개수가 여유가 있다면(최소 키 개수보다 많다면) redistribution을 진행하고 neighbor 노드의 키 개수가 여유롭지 않다면 merge합니다. 이러면 함수가 한 번 종료되는데 이후에 바로 replace 함수가 호출됩니다. 이 함수는 parent node를 인자로 받아서 노드에 delete한 키 값이 존재하는지 확인하고 그렇다면 그 값을 successor로 대체합니다. 이후에 balance_tree 함수가 호출되는데 이 함수는 index node의 키의 개수가 부족해질 경우 이를 merge 또는 redistribution 해주는 함수이고 기본 원리는 delete_leaf 함수와 동일합니다. 재귀를 이용해 계속해서 delete_leaf와 balance_tree 함수를 호출하면서 tree를 balance시키고 전부 완료되면 함수를 종료합니다.

4) Search와 Range Search

Search는 간단하게 재귀로 leaf node까지 찾아들어간 후 해당 키가 노드에 존재하면 value를 print하고 그렇지 않으면 NOT FOUND를 print하도록 하였습니다. Leaf node에 도달하기 전에 노드를 하나 방문할 때마다 노드의 모든 키값을 print합니다.

Range Search는 만약 start 값이 end 값보다 크면 함수를 종료하고, 그렇지 않으면 search와 마찬가지로 start값을 토대로 leaf node까지 찾아들어간 후 start값보다 크거나 같은 키값이 있으면 거기서부터 시작해 key,value pair를 계속 print합니다. node.next를 통해 right sibling으로 넘어가서 end에 도달할 때까지 print를 반복하고 만약 end에 도달하면 함수를 종료합니다. 그 어떤 키값도 찾지 못했으면 NOT FOUND를 print하고 함수를 종료합니다.

5) Serialize와 Deserialize

비플트리를 저장하는 방법 중 Bulk Loading이란 방법을 보게 되었고 그것을 그대로 구현하는 것은 어렵다고 느껴서 거기서 아이디어를 얻어서 비플트리를 저장하게 되었습니다.

우선 index.dat에 파일을 저장할 때 serialize 함수는 degree를 가장 첫줄에 저장합니다. 이후 가장 왼쪽에 있는 leaf node까지 찾아 들어간 다음 그 노드부터 가장 오른쪽에 있는 leaf node까지의 key, value pair를 한 줄씩 전부 저장합니다. 마치 input.csv에 들어있는 형태로 저장을 하는데 이때 차이점은 key,value pair가 key에 대해서 오름차순으로 정렬되어 저장되는 것입니다. 정렬되어 있다는 사실이 매우 중요한데 이후에 deserialize 함수로 index.dat으로부터 트리를 생성할 때 이 사실이 중요하게 작용합니다. index.dat에 있는 key,value들은 전부 정렬되어있기 때문에 항상 새로 들어오는 key는 가장 큰 키일 수밖에 없습니다. 그러므로 새로운 pair는 항상 rightmost leaf node에 저장됩니다. deserialize 함수는 항상 rightmost leaf node에 새로운 pair를 저장하므로 임의의 값을 삽입하는 insert 함수보다 훨씬 더 효율적입니다. 이 때도 node가 꽉 차면 split을 진행해주어야 하므로 bulk loading에서 이름을 따와 bulk_split_leaf와 bulk_split_index 함수를 이용해 split을 해줍니다. 모든 과정이 끝나면 index.dat으로부터 트리가 생성됩니다.

3. Detailed Description of Code

```
def insert_leaf(self, new_pair) -> bool:     # new_pair는 새로운 [key,value] pair
    if self.num_of_keys == 0:                # 비어있는 노드라면 pairs에 바로 new_pair를 append
        self.pairs.append(new_pair)
    else:                                    # 비어있지 않다면 new_pair가 들어갈 위치를 찾음
        for i, pair in enumerate(self.pairs):
            if new_pair[0] < pair[0]:        # 새로운 키가 기존 키보다 작으면 그 위치에 new_pair 삽입
                self.pairs = self.pairs[:i] + [new_pair] + self.pairs[i:]
                self.num_of_keys += 1
                return True
            elif new_pair[0] == pair[0]:     # 중복키가 존재하면 False를 return하고 함수 종료
                return False
        self.pairs.append(new_pair)          # 새로운 키가 가장 크면 마지막에 append
    self.num_of_keys += 1
    return True
```

insert_leaf: 유일하게 Node Class에 있는 함수입니다. insert_leaf 함수는 새로운 [key, value] pair(리스트 타입)를 인자로 받습니다. 노드가 비어 있다면 노드의 pairs 리스트에 바로 new_pair를 추가합니다. 노드가 비어있지 않다면 for문을 돌면서 새로운 키가 기존 키보다 작으면 해당 위치에 new_pair를 삽입합니다. 만약 새로운 키가 기존 키와 같다면 중복키이므로 False를 return하고 함수를 종료합니다. for문이 끝났는데도 함수가 종료되지 않았으면 새로운 키가 기존 키들보다 크다는 뜻이므로 pairs에 마지막에 오도록 new_pair를 append해줍니다. 이 함수는 삽입이 성공하면 True를, 중복키가 발견된다면 False를 return합니다.

```
def insert(self, new_pair):
    node = self.root
    prev = None          # node의 부모 노드를 가리키는 변수 (초기에는 None)
    stack = []           # root 노드부터 순차적으로 index 노드들을 저장할 비어있는 스택을 생성
    while not node.is_leaf:                  # 리프노드에 도달할 때까지 반복
        for i, pair in enumerate(node.pairs):
            if new_pair[0] == pair[0]:       # 중복된 키면 함수 종료
                return
            elif new_pair[0] < pair[0]:
                stack.append(node)
                node = pair[1]               # node에 pair가 가리키는 left child를 할당
                break
            elif i == len(node.pairs) - 1:   # 새로운 키가 가장 커서 끝에 다다른 경우
                stack.append(node)
                node = node.next             # node에 rightmost child를 할당
                break
    if not node.insert_leaf(new_pair):  # leaf 레벨에서 삽입이 실패하면 함수 종료
        return

    if stack:
        prev = stack.pop()                   # prev에 leaf 노드의 parent 노드 할당
    if node.num_of_keys == self.degree:
        self.split_leaf(prev, node)          # node의 키 개수가 초과되면 split
        if prev and prev.num_of_keys == self.degree:
            self.split_index(stack, prev)    # parent 노드도 키 개수 초과하면 split
```

insert: insert 함수는 새로운 key, value pair를 받아 키값을 토대로 new_pair가 삽입될 leaf node를 찾아가면서 index node들을 스택에 저장합니다. Leaf node에 도달하면 insert_leaf 함수를 호출합니다. insert_leaf가 false를 반환하면 삽입이 실패한 것이므로 함수를 종료합니다. 그렇지 않은 경우 스택에서 parent node를 뽑아내고 node의 키 개수가 초과되었는지 확인합니다. 만약 초과되었으면 split_leaf 함수를 호출합니다. 이때 parent node의 키 개수도 초과되면 split_index 함수를 호출합니다.

```
def split_leaf(self, parent: Node, node: Node):
    if parent is None:  # root가 리프노드인 경우에 부모가 존재하지 않으므로 새로 생성
        parent = Node()

    new_sibling = Node()            # 새로운 오른쪽 형제 노드를 생성
    new_sibling.next = node.next    # node가 가리키던 right sibling을 new_sibling이 가리키도록 함
    node.next = new_sibling         # node는 이제 새로 만들어진 new_sibling을 가리킴

    mid = self.degree // 2
    mid_key = node.pairs[mid][0]    # 부모 노드로 올라갈 키 지정 (degree/2을 내림한 값이 기준)

    if parent.num_of_keys == 0:     # 부모가 새로 생성된 경우
        parent.pairs.append([mid_key, node])    # mid_key를 올리고 left child로 node를 가리키게 함
        parent.next = new_sibling               # 부모의 rightmost child는 new_sibling이 됨
    else:                           # 부모가 원래 있던 경우
        for i, pair in enumerate(parent.pairs):
            if mid_key < pair[0]:
                parent.pairs[i][1] = new_sibling    # 기존에 node를 가리키던 포인터가 new_sibling을 가리키도록 바꿈
                parent.pairs = parent.pairs[:i] + [[mid_key, node]] + parent.pairs[i:]  # node를 가리키는 새로운 pair 삽입
                break
            elif i == len(parent.pairs) - 1:        # mid_key가 가장 큰 경우
                parent.pairs.append([mid_key, node])    # 가장 마지막에 새로운 pair 삽입
                parent.next = new_sibling               # rightmost child를 new_sibling으로 지정
                break
    parent.num_of_keys += 1

    new_sibling.pairs = node.pairs[mid:]                # node의 오른쪽 반을 new_sibling이 갖도록 함
    new_sibling.num_of_keys = len(new_sibling.pairs)
    node.pairs = node.pairs[:mid]                       # node를 반으로 줄임 (왼쪽 반만 갖도록)
    node.num_of_keys = len(node.pairs)

    if parent.is_leaf:              # 새로 부모를 생성하면 초기 상태이므로 is_leaf가 True
        parent.is_leaf = False      # 리프 노드가 아니므로 False로 바꿔줌
        self.root = parent          # 새 부모가 root가 되므로 root에 parent을 할당
```

split_leaf: 이 함수는 리프의 부모와 리프 노드를 인자로 받아 리프 노드를 split해주는 함수입니다. 만약 부모가 존재하지 않는다면 리프 노드가 root 노드라는 뜻이므로 부모를 새로 생성합니다. 그리고 split을 진행하기 위해 새로운 오른쪽 형제 노드(new_sibling)를 생성합니다. (degree / 2)를 내림한 값을 기준 위치로 잡고 기준 위치에 있는 key를 부모 노드로 올립니다. 이때 부모 노드에서 기존에 node를 가리키던 포인터를 갖고 있던 pair가 new_sibling을 가리키도록 바꿔주고 [mid_key, node]로 이루어진 새로운 pair를 부모 노드에 삽입합니다. mid_key가 기존에 있던 키들보다 크다면 맨 마지막에 새로운 pair를 append하고 부모의 rightmost child를 new_sibling으로 지정합니다. 만약 부모가 새로 생성돼서 비어있는 상태라면 새로운 pair를 바로 추가하고 부모의 rightmost child를 new_sibling으로 지정합니다. 그 다음 new_sibling이 현재 노드의 오른쪽 반을 갖게 하고 현재 노드 또한 왼쪽 반만 갖도록 자릅니다. 만약 부모가 새로 생성된 부모면 is_leaf의 초깃값이 True이므로 False로 바꿔준 다음 이 부모 노드를 root로 지정합니다.

```
def split_index(self, stack, node: Node):
    parent = None
    if stack:
        parent = stack.pop()    # node의 부모 노드를 스택을 통해 할당
    else:
        parent = Node()         # 부모가 없다면 새로 생성

    mid = self.degree // 2

    new_sibling = Node()                          # 오른쪽 형제 노드 생성
    new_sibling.is_leaf = False
    new_sibling.next = node.next                  # node가 가리키던 rightmost child를 new_sibling이 가리키도록 함
    new_sibling.pairs = node.pairs[mid + 1:]      # 기준 키를 제외한 오른쪽 반을 new_sibling에 넘김
    new_sibling.num_of_keys = len(new_sibling.pairs)
```

(중략 - 새로운 pair를 부모 노드에 추가. split_leaf와 동일)

```
node.next = node.pairs[mid][1]              # node의 rightmost child를 기준 키가 가리키던 left child로 변경
node.pairs = node.pairs[:mid]               # 기준 키를 제외한 나머지 왼쪽 반을 node가 갖도록 함
node.num_of_keys = len(node.pairs)

if parent.is_leaf:                          # 새로운 부모면 root로 지정
    parent.is_leaf = False
    self.root = parent

if parent.num_of_keys == self.degree:       # 부모 노드도 키 개수 초과하면 재귀로 함수 호출
    self.split_index(stack, parent)
```

split_index: 이 함수는 스택과 index 노드(node)를 인자로 받아 index 노드를 split하는 함수입니다. 스택을 통해 부모를 할당하고 스택이 비어있다면 부모를 새로 만듭니다. 새로운 오른쪽 형제 노드(new_sibling)를 만들어서 node가 가리키던 rightmost child를 new_sibling이 가리키게 합니다. split_leaf 함수와 다르게 기준 키를 제외한 오른쪽 반을 new_sibling에게 넘깁니다. 이후 split_leaf와 동일한 과정으로 새로운 pair를 부모 노드에 추가합니다. 이 작업이 끝나면 node의 rightmost child를 기준 키가 들어있는 pair [기준 키, left child]의 left child로 변경합니다. 그리고 node가 기준 키를 제외한 나머지 왼쪽 반만 갖도록 합니다. 부모가 새로운 생성된 부모면 root로 지정해주고, 만약 부모 노드의 키 개수도 초과되었다면 재귀로 함수를 다시 호출합니다.

---

delete_leaf: delete_leaf 함수는 leaf까지 찾아들어가는 부분, Redistribution, Merge 세 부분으로 나뉩니다.

```
def delete_leaf(self, parent: Node, node: Node, key: int, idx: int):
    min_keys = math.ceil(self.degree / 2) - 1   # root를 제외한 노드가 갖는 최소 키 개수

    if not node.is_leaf:                        # 리프 노드에 도달할 때까지 재귀로 함수 호출
        for i, pair in enumerate(node.pairs):
            if key < pair[0]:
                successor = self.delete_leaf(node, pair[1], key, i)     # 현재 노드, 자식 노드, 키, 자식 노드의 index를 넘겨줌
                self.replace(parent, key, successor)                    # 부모 노드에서 삭제된 키가 존재하면 successor로 대체
                self.balance_tree(parent, node, key)                    # 부모와 현재 노드를 넘겨서 tree를 balance시킴
                return successor
            elif i == len(node.pairs) - 1:
                successor = self.delete_leaf(node, node.next, key, i + 1)
                self.replace(parent, key, successor)
                self.balance_tree(parent, node, key)
                return successor
```

(중략 - 삭제할 키가 있는 위치 찾고 pos에 저장, 키가 존재하지 않으면 종료. replace_key(index 노드에서 삭제한 키가 발견됐을 때를 대비한 대체키) 선언. 삭제할 키의 다음 키가 노드에 존재하면 replace_key에 할당)

첫번째 부분: delete_leaf 함수는 부모, 현재 노드, 삭제할 키, 자식 노드의 index를 인자로 받는 함수입니다. 삭제할 때 부모 노드가 필요하기 때문에 재귀로 leaf 노드까지 찾아가는 과정에서 항상 현재 노드를 가리키는 포인터를 넘깁니다. leaf 레벨에서 삭제를 하면 replace 함수와 balance_tree 함수를 호출합니다. 이때 successor는 index 노드에서 삭제된 키가 발견됐을 때 이를 대체하기 위한 키를 넘겨받는 변수입니다. 이를 replace 함수에 넣어서 키를 대체합니다. balance_tree는 leaf 레벨에서 삭제를 진행한 후 혹시 index 노드에서 키 부족 현상이 발생할 경우 이를 해결하기 위한 함수입니다. 따라서 부모 노드와 현재 노드를 인자로 넘깁니다.

```
node.pairs = node.pairs[:pos] + node.pairs[pos + 1:]    # 노드에서 키 삭제
node.num_of_keys -= 1

if node is self.root:
    return

if node.num_of_keys >= min_keys:    # 삭제 이후에도 키 개수가 충분한 경우
    if pos == 0 and idx > 0:                            # 가장 첫번째 키가 삭제됐으면 부모의 키도 삭제되어야함
        parent.pairs[idx - 1][0] = node.pairs[0][0]     # 삭제된 키의 다음 키로 부모 키를 대체함
else:                                # 키 개수가 부족한 경우
    left_neighbor, right_neighbor = None, None
    if idx > 0:                          # delete_leaf 함수의 마지막 인자 idx(부모 노드에서 현재 노드의 위치) 활용
        left_neighbor = parent.pairs[idx - 1][1]    # 왼쪽 키의 left child가 left_neighbor
    if idx < len(parent.pairs) - 1:
        right_neighbor = parent.pairs[idx + 1][1]   # 오른쪽 키의 left child가 right_neighbor
    elif idx == len(parent.pairs) - 1:
        right_neighbor = parent.next                # 더이상 오른쪽 키가 없다면 부모의 rightmost child가 right_neighbor

    # Redistribution
    if left_neighbor and left_neighbor.num_of_keys > min_keys:  # 왼쪽 형제가 존재하고 키의 개수가 많은 경우
        node.pairs = [left_neighbor.pairs[-1]] + node.pairs     # 왼쪽 형제의 마지막 키를 가져옴
        node.num_of_keys += 1
        left_neighbor.pairs = left_neighbor.pairs[:-1]          # 왼쪽 형제의 마지막 키 삭제
        left_neighbor.num_of_keys -= 1
        parent.pairs[idx - 1][0] = node.pairs[0][0]             # 부모 노드의 키 업데이트
    elif right_neighbor and right_neighbor.num_of_keys > min_keys:  # 오른쪽 형제가 존재하고 키의 개수가 많은 경우
        if node.num_of_keys == 0:                               # 노드가 비어있으면 대체키는 오른쪽 형제의 첫번째 키
            replace_key = right_neighbor.pairs[0][0]
        node.pairs.append(right_neighbor.pairs[0])          # 오른쪽 형제의 첫번째 키를 추가
        node.num_of_keys += 1
        right_neighbor.pairs = right_neighbor.pairs[1:]     # 오른쪽 형제의 첫번째 키 삭제
        right_neighbor.num_of_keys -= 1
        parent.pairs[idx][0] = right_neighbor.pairs[0][0]   # 부모 노드의 키 업데이트
```

두번째 부분: pos를 이용해 노드에서 키를 삭제하고 만약 노드가 root였으면 그대로 함수를 종료합니다. 삭제 이후에도 노드의 키 개수가 여유있는 경우에는 부모의 키만 바꿉니다. 왼쪽 형제 또는 오른쪽 형제가 존재하는 경우 지정해주고 아니면 None을 할당합니다. 삭제 이후 키 개수가 부족해졌을 때 왼쪽 또는 오른쪽 형제의 키 개수가 여유가 있다면(최소 키 개수보다 많다면) Redistribution을 진행합니다. 왼쪽 형제에서 키를 가져오는 경우에는 왼쪽 형제의 마지막 키를 가져오고 오른쪽 형제에서 키를 가져오는 경우에는 오른쪽 형제의 첫번째 키를 가져옵니다. 이후 부모 노드의 키를 업데이트합니다.

```
    # Merge
    elif left_neighbor:                 # 왼쪽 형제의 키 개수가 충분하지 않은 경우 (node를 왼쪽 형제에 흡수시킴)
        left_neighbor.next = node.next      # node의 rightmost child는 이제 왼쪽 형제의 rightmost child
        left_neighbor.pairs = left_neighbor.pairs + node.pairs
        left_neighbor.num_of_keys += node.num_of_keys
        if idx < len(parent.pairs):         # node가 부모의 rightmost child가 아닌 경우
            parent.pairs[idx][1] = left_neighbor                    # node를 가리키던 포인터가 왼쪽 형제를 가리키도록 변경
        else:                               # node가 부모의 rightmost child인 경우
            parent.next = left_neighbor                             # 부모의 rightmost child를 왼쪽 형제로 변경
        parent.pairs = parent.pairs[:idx - 1] + parent.pairs[idx:]  # 부모 노드에서 기존에 왼쪽 형제 가리키던 pair 삭제
        parent.num_of_keys -= 1

        if parent == self.root and parent.num_of_keys == 0:
            self.root = left_neighbor                               # 부모가 비었는데 root였다면 root를 새로 지정
    elif right_neighbor:                # 오른쪽 형제의 키 개수가 충분하지 않은 경우 (오른쪽 형제를 node에 흡수시킴)
        if node.num_of_keys == 0:
            replace_key = right_neighbor.pairs[0][0]
        node.next = right_neighbor.next                 # 오른쪽 형제의 rightmost child는 이제 node의 rightmost child
        node.pairs = node.pairs + right_neighbor.pairs
        node.num_of_keys += right_neighbor.num_of_keys
        if idx == len(parent.pairs) - 1:        # node가 부모의 오른쪽에서 두번째 child인 경우
            parent.next = node                          # 부모의 rightmost child를 node로 변경
        else:                                   # node가 부모의 오른쪽에서 세번째~마지막 child인 경우
            parent.pairs[idx + 1][1] = node             # 오른쪽 형제를 가리키던 포인터를 node로 바꿈
        parent.pairs = parent.pairs[:idx] + parent.pairs[idx + 1:]  # 기존에 오른쪽 형제 가리키던 pair 삭제
        parent.num_of_keys -= 1

        if parent == self.root and parent.num_of_keys == 0:
            self.root = node                                        # 부모가 비었는데 root였다면 root를 새로 지정

return replace_key
```

세번째 부분: 왼쪽 또는 오른쪽 형제가 존재하지만 최소 키 개수만 가지고 있어 키를 가져올 수 없는 경우에는 Merge를 진행합니다. 왼쪽 형제와 merge하는 경우 왼쪽 형제가 node를 흡수합니다. 기존에 node를 가리키던 포인터가 왼쪽 형제를 가리키도록 바꾸고 부모 노드에서 기존에 왼쪽 형제를 가리키던 pair를 삭제합니다. 부모가 비었는데 root였다면 root를 왼쪽 형제로 지정합니다. 오른쪽 형제와 merge하는 경우 node가 오른쪽 형제를 흡수합니다. 기존에 오른쪽 형제를 가리키던 포인터가 node를 가리키도록 바꾸고 부모 노드에서 기존에 오른쪽 형제를 가리키던 pair를 삭제합니다. 부모가 비었는데 root였다면 root를 node로 지정합니다. 이 과정에서 replace_key에 적절한 값이 있으면 할당하고 없으면 None이 되도록 하고 마지막에 replace_key를 return합니다.

```
def replace(self, parent: Node, key: int, successor):
    if parent and successor is not None:
        for i, pair in enumerate(parent.pairs):
            if pair[0] == key:
                parent.pairs[i][0] = successor      # 삭제된 키를 찾으면 successor로 대체하고 종료
                return
```

replace: delete_leaf 함수에서 replace_key로 return한 값을 successor 변수가 받습니다. 노드의 각 키와 비교하면서 키가 삭제된 키와 같다면 successor로 대체해주고 함수를 종료합니다.

---

balance_tree: 이 함수는 delete_leaf 함수와 거의 유사하지만 leaf 레벨이 아닌 index 노드에서 키 부족 현상이 발생했을 경우 merge 또는 redistribution을 진행하는 함수입니다. delete_leaf와 똑같이 세 부분으로 나뉩니다.

```
def balance_tree(self, parent: Node, node: Node, key: int):
    min_keys = math.ceil(self.degree / 2) - 1
    if not parent or node.num_of_keys >= min_keys:  # 부모가 없거나 노드의 키 개수가 충분하면 함수 종료
        return

    idx = -1
    for i, pair in enumerate(parent.pairs):     # 부모 노드에서 현재 노드의 index를 idx에 저장
        if pair[1] == node:
            idx = i
            break
        if parent.next == node:                 # 부모의 rightmost child면 부모의 길이가 index가 됨
            idx = len(parent.pairs)
            break
```

(중략 - 왼쪽 또는 오른쪽 형제가 존재하면 left_neighbor, right_neighbor에 저장. 없다면 None 할당)

첫번째 부분: 부모가 없거나 노드의 키 개수가 충분하면 함수를 종료합니다. 부모 노드에서 현재 노드의 index를 idx 변수에 저장합니다.

```
# Redistribution - Rotate하듯이 키를 옮김
if left_neighbor and left_neighbor.num_of_keys > min_keys:
    # idx를 이용해서 왼쪽 형제의 키를 부모로 옮기고 부모의 키를 node로 옮김
    # 왼쪽 형제의 마지막 키를 가져오므로 왼쪽 형제의 rightmost child가 node의 leftmost child가 됨
    node.pairs = [[parent.pairs[idx - 1][0], left_neighbor.next]] + node.pairs  # 부모 노드의 키를 node로 옮김
    left_neighbor.next = left_neighbor.pairs[-1][1]         # 왼쪽 형제의 rightmost child를 오른쪽에서 두번째 child로 변경
    parent.pairs[idx - 1][0] = left_neighbor.pairs[-1][0]   # 왼쪽 형제의 마지막 키를 부모 노드로 옮김
    left_neighbor.pairs = left_neighbor.pairs[:-1]
    node.num_of_keys += 1
    left_neighbor.num_of_keys -= 1
```

(중략 - 오른쪽 형제에서 키를 빌려오는 경우도 구조적으로 동일)

두번째 부분: 왼쪽 또는 오른쪽 형제의 키 개수가 최소 키 개수보다 많다면 Redistribution을 진행합니다. idx로 부모 노드에서 현재 노드의 위치를 저장했으므로 이를 이용합니다. 왼쪽 형제에서 키를 빌려오는 경우, 왼쪽 형제의 마지막 키를 부모로 옮기고 부모의 키를 현재 노드의 첫번째 키가 되도록 옮깁니다. 이때 왼쪽 형제의 rightmost child가 현재 노드의 leftmost child가 됩니다. 마지막으로 왼쪽 형제의 rightmost child를 오른쪽에서 두번째 child로 변경합니다. 오른쪽 형제에서 빌려오는 경우도 구조적으로 동일합니다.

```
# Merge
elif left_neighbor:     # node가 왼쪽 형제 흡수
    # 부모의 키와 왼쪽 형제의 rightmost child를 새로운 pair로 묶어서 왼쪽 형제 마지막에 추가
    left_neighbor.pairs.append([parent.pairs[idx - 1][0], left_neighbor.next])
    node.pairs = left_neighbor.pairs + node.pairs               # node 앞에 왼쪽 형제 추가
    node.num_of_keys = len(node.pairs)
    parent.pairs = parent.pairs[:idx - 1] + parent.pairs[idx:]  # 부모에서 index 삭제
    parent.num_of_keys -= 1
    if parent.num_of_keys == 0 and parent == self.root:     # 부모가 비었는데 root였으면 root를 node로 지정
        self.root = node
```

(중략 - 오른쪽 형제와 merge하는 과정도 구조적으로 동일)

세번째 부분: 왼쪽 또는 오른쪽 형제의 키 개수가 최소 키 개수라면 Merge를 진행합니다. 왼쪽 형제가 존재한다면 현재 노드가 왼쪽 형제를 흡수합니다. 부모의 키와 왼쪽 형제의 rightmost child를 새로운 pair로 묶어서 왼쪽 형제 마지막에 추가하고 왼쪽 형제를 현재 노드 앞에 들어오도록 추가합니다. 이후 부모에서 해당 index를 삭제합니다. 만약 부모가 collapse했는데 부모가 root였으면 현재 노드가 root가 되도록 지정합니다. 오른쪽 형제와 merge하는 과정도 구조적으로 동일합니다.

```python
def search(self, node: Node, key: int) -> bool:
    if not node.is_leaf:                        # 리프 노드에 도달할 때까지 재귀로 탐색
        for i, pair in enumerate(node.pairs):   # 노드의 모든 키 출력
            if i < len(node.pairs) - 1:
                print(pair[0], end=',')
            else:
                print(pair[0])

        for i, pair in enumerate(node.pairs):
            if key < pair[0]:                   # 탐색할 키가 노드에 있는 키보다 작으면 left child로 이동
                return self.search(pair[1], key)
            elif i == len(node.pairs) - 1:      # 탐색할 키가 가장 크면 rightmost child로 이동
                return self.search(node.next, key)

    for i, pair in enumerate(node.pairs):       # 리프 노드에 키가 있는지 확인하고, 없으면 True 있으면 False return
        if key == pair[0]:
            print(pair[1])
            return True
    return False
```

search: 노드와 키를 받아 재귀로 탐색한 후 찾으면 True 못 찾으면 False를 반환하는 함수입니다. 리프 노드에 도달하기 전에 index 노드에서 모든 키를 출력합니다. 탐색할 키를 토대로 어떤 child로 들어갈지 결정하고 리프 노드에 도달하면 모든 키와 대조해서 같은 키가 있으면 True, 없으면 False를 return합니다.

```python
def range_search(self, node: Node, start: int, end: int):
    if start > end:                     # start가 end보다 크면 탐색 조건이 성립하지 않음 (예외 처리)
        print("Lower bound bigger than Upper bound")
        return

    if not node.is_leaf:                # search 함수와 마찬가지
        for i, pair in enumerate(node.pairs):
            if start < pair[0]:
                self.range_search(pair[1], start, end)
                return
            elif i == len(node.pairs) - 1:
                self.range_search(node.next, start, end)
                return

    found = False                       # range에 맞는 값을 발견하면 True를 할당하기 위한 초기 설정
    for i, pair in enumerate(node.pairs):
        if start <= pair[0] <= end:     # 범위에 맞는 값 발견하면 거기서부터 순차적으로 출력
            print(f"{pair[0]},{pair[1]}")
            found = True
```

```
    node = node.next                    # right sibling으로 계속 넘어감
    while node:
        for i, pair in enumerate(node.pairs):
            if start <= pair[0] <= end:
                print(f"{pair[0]},{pair[1]}")
                found = True
            elif pair[0] > end:         # 만약 키가 end보다 크면 더 이상 탐색할 필요 없으므로 함수 종료
                if not found:
                    print("NOT FOUND")
                return
        node = node.next

    if not found:                       # found가 False면 못 찾은 것이므로 NOT FOUND 출력
        print("NOT FOUND")
```

range_search: start와 end값을 입력하면 start와 end를 포함한 키들을 전부 출력합니다. 예외 처리를 위해 start가 end보다 크면 함수를 종료합니다. search 함수와 마찬가지로 리프노드까지 찾아들어간 다음 found 변수 (bool 타입)을 선언하고 False를 할당합니다. 이는 나중에 아무 키도 찾지 못했을 때 Not Found를 출력해주기 위한 변수입니다. 찾은 리프 노드에서 순차적으로 탐색을 해서 범위에 맞는 키를 찾으면 거기서부터 계속 출력을 하고 노드의 끝에 도달하면 right sibling으로 넘어갑니다. 키가 end보다 크면 함수를 종료합니다. 키를 하나라도 찾는 순간 found는 True가 되므로 마지막에 found의 truth value를 확인하고 못 찾았으면 Not Found를 출력합니다.

```
def serialize(self, idx_file: str):
    f = open(idx_file, 'w')
    f.write(str(self.degree))
    f.write('\n')

    node = self.root
    # 트리의 가장 왼쪽 리프 노드로 이동
    while node.pairs and not node.is_leaf:
        node = node.pairs[0][1]

    # 가장 왼쪽 리프 노드부터 가장 오른쪽 리프 노드까지 모든 key,value를 한 줄씩 저장
    while node:
        for key, value in node.pairs:
            f.write(str(key))
            f.write(',')
            f.write(str(value))
            f.write('\n')
        node = node.next
    f.close()
```

serialize: 파일 이름을 str 타입으로 받습니다. 파일 이름을 이용해 파일을 생성하고 첫 줄에 degree를 입력합니다. 트리의 가장 왼쪽 리프 노드까지 이동한 후 가장 오른쪽 리프 노드에 도달할 때까지 모든 key,value를 한줄씩 저장합니다. 이렇게 되면 모든 key,value가 key에 대해서 오름차순으로 파일에 저장됩니다.

```
def deserialize(self, idx_file: str):
    f = open(idx_file, 'r')
    degree = int(f.readline())          # degree부터 읽은 다음 degree를 설정함
    self.degree = degree

    rightmost = self.root               # 새로운 키는 항상 가장 큰 키이므로 무조건 rightmost leaf node에 삽입됨
    parent = None

    while True:
        line = f.readline()
        if not line:
            break
        rightmost.pairs.append(list(map(int, line.split(','))))     # 읽은 key,value를 리스트 형태로 저장
        rightmost.num_of_keys += 1
```

```
    stack = []

    if rightmost.num_of_keys == self.degree:    # rightmost leaf가 키 최대 개수 초과하면 split 진행
        node = self.root
        while not node.is_leaf:
            stack.append(node)
            node = node.next            # rightmost leaf의 부모로 이동하려면 계속 rightmost child로 이동하면 됨
        if stack:
            parent = stack.pop()
        rightmost = self.bulk_split_leaf(parent, rightmost) # leaf 레벨에서 split 진행하고 rightmost leaf return함
        if parent and parent.num_of_keys == self.degree:    # 부모도 키 개수 초과하면 index 레벨에서 split 진행
            self.bulk_split_index(stack, parent)

f.close()
```

deserialize: 이 함수는 index file을 읽어서 비플트리를 생성하는 함수입니다. 첫줄에 있는 degree를 읽고 degree를 먼저 설정한 다음 순차적으로 key,value를 읽어 트리에 삽입합니다. 이때 index file의 키들은 오름차순으로 정렬되어있기 때문에 삽입은 무조건 가장 오른쪽 리프 노드에서 이루어집니다. root에서 출발해 항상 rightmost child를 방문하면서 스택에 저장합니다. 스택에 마지막으로 저장되는 노드가 리프 노드의 부모입니다. 리프 레벨에서 삽입을 진행했는데 초과가 발생하면 리프에서 split을 진행하는데 rightmost 변수가 항상 rightmost leaf node를 가리키도록 포인터를 리턴합니다. 부모 또한 초과가 발생하면 index 레벨에서 split을 진행합니다.

```
def bulk_split_leaf(self, parent: Node, node: Node) -> Node:
    if parent is None:
        parent = Node()

    new_sibling = Node()        # 오른쪽 형제 노드 생성 <- 이 노드가 rightmost leaf node가 됨
    node.next = new_sibling     # node의 오른쪽 형제를 new_sibling으로 변경

    mid = self.degree // 2
    mid_key = node.pairs[mid][0]

    parent.pairs.append([mid_key, node])    # for문 돌 필요 없이 항상 부모의 마지막에 pair가 추가됨
    parent.next = new_sibling               # 부모의 rightmost child는 new_sibling으로 바뀜
    parent.num_of_keys += 1
```

(중략 - split_leaf와 동일)

```
    return new_sibling
```

bulk_split_leaf: bulk loading에서 아이디어를 얻었기 때문에 bulk를 붙였습니다. deserialize 함수에서 leaf 노드를 split해야할 때 호출하는 함수입니다. rightmost leaf node가 될 오른쪽 형제 노드(new_sibling)를 새로 생성하고 현재 노드의 오른쪽 형제가 되도록 지정합니다. 앞에 split_leaf 함수와 다른 점은 for문으로 삽입할 위치를 탐색할 필요 없이 언제나 부모의 마지막에 새로운 pair가 추가된다는 것입니다. 부모의 rightmost child는 new_sibling이 됩니다. 부모에 pair를 추가한 이후 과정은 split_leaf와 동일하고 마지막에 new_sibling, 즉 rightmost leaf node를 반환하고 종료합니다.

```
def bulk_split_index(self, stack, node: Node):
```

(중략 - split_index함수와 앞부분 동일)

```
parent.pairs.append([mid_key, node])
parent.next = new_sibling
parent.num_of_keys += 1
```

(중략 - split_index함수와 뒷부분 동일)

bulk_split_index: deserialize 함수에서 index 노드를 split해야할 때 호출하는 함수입니다. split_index 함수와 매우 유사하게 새로운 오른쪽 형제 노드(new_sibling)을 생성하고, 현재 노드의 오른쪽 반을 new_sibling에게 넘겨준 다음 new_sibling이 현재 노드의 rightmost child를 가리키도록 합니다. 이후 bulk_split_leaf와 마찬가지로 for문을 돌지 않고 곧바로 부모 노드에 새로운 pair를 마지막에 추가합니다. 부모의 rightmost child는 new_sibling이 됩니다. 이후에 부모가 새로 생성된 부모면 root로 지정해주고 부모 노드의 키 개수도 초과되면 함수를 다시 호출하는 부분은 split_index 함수와 동일합니다.
