

<2021 B+Tree implementation assignment>

경제금융학부 2019039125 이선민

I . Summary of your Algorithm

-index file에 있는 데이터 (비벨트리의 degree, 비벨트리 리프 노드들의 {key:value})을 읽어 비벨트리를 만들고, 비벨트리에 키와 데이터 값을 삽입, 삭제하며, 찾고자 하는 단일 키 혹은 키의 범위에 맞는 value를 출력한다. 다시 비벨트리에서 index file로 저장할 때, (만든 비벨트리를 index file로 저장할 때) 비벨트리 리프 노드들의 {key:value}로 저장한다. index file에 비벨트리를 저장하지 않은 이유는 index file의 용량이 커져서 index file을 읽고 쓰는데 시간이 오래 걸리기 때문이다. 자세한 함수들의 설명은 II. Detailed description of your codes에 있다.

II . Detailed description of your codes

-각 Class의 변수 설명

1. saving

: index file에 저장하는 정보이다.

self.degree = 비벨트리가 가질 수 있는 최대 자식의 수

self.pair = {key : value} 형태의 딕셔너리, 여기서 key와 value는 비벨트리의 리프노드의 key와 value이다.

2. Non_Leaf

: Leaf 노드가 아닌 노드이다.

self.cnt_key = 노드가 가진 키의 개수

self.pair = {key : left child node}의 쌍, 딕셔너리로 표현

self.key_list = 노드의 키 집합, left child node를 불러올 때 주로 사용함

self.right_node = 가장 오른쪽에 있는 자식 노드를 가리킴

self.leaf = 리프 노드인지 아닌지 나타냄, Non Leaf node는 False이다.

self.parent = 부모 노드를 가리키는 변수

3. Leaf

: Leaf 노드이다.

self.cnt_key = 노드가 가진 키의 개수

self.pair = {key : value}의 쌍, 딕셔너리로 표현

self.key_list = 노드의 키 집합, value를 불러올 때 주로 사용함

self.right_node = 오른쪽에 있는 리프 노드를 가리킴, 맨 오른쪽 노드는 이 값이 None이다.

self.prev_node = 왼쪽에 있는 리프 노드를 가리킴, 맨 왼쪽 노드는 이 값이 None이다.

self.leaf = 리프 노드인지 아닌지 나타냄, Leaf node는 True이다.

self.parent = 부모 노드를 가리키는 변수

4. BPTree

: 비벨트리이다.

self.root = 비벨트리의 루트 노드

self.size = 비벨트리의 차수(degree)

self.max_child = 최대 가질 수 있는 자식의 수, 이는 비벨트리의 차수와 동일하다.

self.max_keys = 최대 가질 수 있는 키의 개수, 이는 비벨트리의 차수에서 1을 뺀 값이다.

self.min_keys = 최소 키의 개수, $\lceil \text{degree}/2 \rceil$ 이다. 이때 $\lceil \rceil$ 는 올림을 뜻한다.

-각 Class에서의 함수들

1. saving

-AddPair(self, pair): self.pair의 값을 설정하는 함수

이는 인수로 받은 pair를 self.pair로 설정한다.

-ClearPair(self): self.pair의 값을 초기화 하는 함수

이는 인덱스 파일을 불러와서 값을 조정하고 다시 인덱스 파일에 저장하는 과정에서 원래 있던 pair의 값을 초기화 시키고 새로운 pair를 저장할 때 사용한다.

-createTree(self): 주어진 degree와 pair의 정보를 토대로 트리를 구성하는 함수

인덱스 파일에는 트리 정보를 저장하지 않기 때문에, 인덱스 파일에 저장된 정보 (saving)을 통해 비벨트리를 만드는 함수가 필요하다.

2. Non_Leaf

-split(self): 해당 노드(self)를 쪼개는 함수

해당 노드를 쪼개는 기준(mid)은 key의 개수를 2로 나눈 값을 내림한 값이다.

이 노드를 Right와 Left로 나누고, 가운데에 해당하는 노드 (부모 노드로 올라갈 노드)는 top이라고 하였다.

Right 노드와 Left 노드 모두 Non_Leaf 클래스에 해당한다.

1. Right 노드의 데이터 정리

→ Right 노드는 self의 mid+1번째 인덱스부터 끝까지 해당하는 정보들을 가진다.

→ Right 노드의 right_node는 self의 right_node이다.

→ 혹시 몰라서 key를 기준으로 오름차순 정렬을 해주었다.

2. Left 노드의 데이터 정리

→ Right 노드와 동일한 방식으로 진행했다.

→ Left 노드는 self의 처음부터 mid-1번째 인덱스까지 해당하는 정보들을 가진다.

→ Left 노드의 right_node는 self의 mid에 해당하는 왼쪽 자식 노드이다.

3. 만일 Left와 Right가 Non Leaf이므로, 밑에 자식 노드의 parent 값을 재설정해야 한다.

4. 부모 노드로 올라간 mid의 키를 가진 top노드의 정보를 정리해준다.

5. Left와 Right 노드의 parent 값을 top으로 재설정해준다.

-Add_NonLeaf(self, key, left_child): Non Leaf 노드(self)에 {key : left child} 정보를 추가하는 함수

매개변수로 주어진 key가 self.key_list의 원소보다 작으면 그 원소의 인덱스에 {key:left child} 정보가 추가된다.
만일 key가 self.key_list의 모든 원소들 보다 크면, self.right_node는 매개변수로 받은 left child가 되고, {key : self.right_node} 정보가 추가된다. (원래 있던 right_node가 key의 left child에 해당하기 때문)
혹시 몰라서 각 과정마다 self.pair와 self.key_list를 key를 기준으로 오름차순 정렬을 해주었다.

-change(self, ex_key, new_key): 해당 노드(self)의 ex_key를 new_key로 바꾸는 함수

이 함수는 ex_key로 받은 키 값만 new_key로 바꿔주는 함수이다.

ex_key 값이 존재하면, 함수 수행 후 1을 반환하며, 존재하지 않는 경우에는 -1을 반환한다.

이 함수를 쓰는 이유는 후에 나오는 Leaf 노드의 경우, 형제 노드에서 키 값을 하나 빌려오는 경우 이에 맞춰 부모 노드의 키 값도 변해야 하기 때문이다.

-Delete_NonLeaf(self, del_key): del_key를 부모 노드(self)에서 지우는 함수

이 함수는 del_key에 해당하는 key값과 value값을 self에서 지우는 함수이다.

1. self의 key가 하나인 경우,

→ self의 정보를 초기화 해준다.

→ 이 경우, 이 함수를 호출하기 전에 해당 노드의 key와 child값을 따로 저장하므로 걱정하지 않아도 된다.

2. self의 key가 하나보다 더 많은 경우

→ 가장 처음에 해당하는 {key : left child} 쌍을 삭제할 때

(만일 자식 노드가 리프 노드라면, prev_node와 right_node를 재설정해줘야 한다.)

→ 가장 처음에 해당하지 않을 경우

→ del_key값이 self.key_list에서 가장 큰 경우는 지우려고 하는 노드가 가장 오른쪽에 해당하는 노드인 경우이다. 이때는 self의 맨 마지막 pair와 키를 삭제하고 이를 right node로 옮겨주면 된다.

-find_NonLeaf(self, child_key): 찾고자 하는 child 노드가 몇 번째 인덱스에 있는지 return하는 함수

이 함수는 child 노드의 가장 작은 키 값을 매개변수 child_key로 받고 self.key_list의 원소들과 비교하여 찾고자 하는 child 노드의 인덱스 값을 반환하는 함수이다.

만일 child_key가 self.key_list[i]보다 작을 경우, child노드는 self의 i번째 pair에 존재하는 노드이므로 i를 반환한다.

하지만, child_key가 self.key_list의 모든 원소들보다 클 경우, child노드는 self.right_node에 위치한 것이며, -1을 반환한다.

-**Empty_node(self, min_keys, max_keys, key, child)**: 원래 키가 하나였던 노드에서 키가 하나 지워져 남은 자식 노드를 child로 받고 지워지기 전 key를 매개변수 key로 받는다.

이 노드는 키가 지워지므로, 형제 노드와 합쳐지는 과정을 진행한다.

1. 왼쪽 형제 노드와 합쳐지는 경우 (노드가 맨 왼쪽에 있지 않으면 이 방법을 사용)

→ 왼쪽 형제 노드에 해당하는 인덱스에 맞는 부모 노드의 키 값과 왼쪽 형제 노드의 right_node를 추가해주고, 왼쪽 형제 노드의 right_node는 child가 된다.

→ child의 부모 노드가 왼쪽 형제 노드로 변경된다.

→ 부모 노드가 완전히 지워지는 것을 방지하여 부모 노드의 키 값과 방금 정리한 왼쪽 자식 노드를 tmp_key와 child로 저장함

→ 부모 노드에서 왼쪽 자식 노드로 내려갔으므로 현재 노드를 지워준다

→ 만일 자식 노드가 리프 노드라면, parent, prev_node, right_node 연결을 다시 해줘야 한다.

2. 오른쪽 형제 노드와 합쳐지는 경우 (노드가 맨 왼쪽에 위치해 있을 때 이 방법을 사용)

→ 1번과 동일한 논리이다.

-**Balance(self, min_keys, max_keys, key=None, child=None)**: 현재 노드인 NonLeaf 노드가 키 범위안에 들지 않을때, 재조정하는 함수이다.

이 함수는 루트 노드에 도달할 때까지 반복한다.

이 때, 현재 노드의 키 개수가 하나일 때 Empty_node 함수로 이동한다.

그 외의 상황에서

1. 해당 노드가 최소 키 범위보다 작은 경우, 형제 노드와 합쳐야 한다.

①왼쪽 형제와 합치는 경우 (해당 노드가 부모 노드의 첫 번째 노드가 아닌 경우)

→ 앞서 Empty_node를 진행한 논리와 비슷하다

→ 왼쪽 노드에 부모 노드에 해당하는 인덱스 값 정보를 추가한다.

→ 여기서는 해당 노드의 키 값이 1개 이상이므로, 이에 맞게 for문을 사용해 해당 노드의 {key : left_child}, key_list정보도 왼쪽 노드에 추가해준다.

→ 왼쪽 노드의 right_node는 원래 노드의 right_node가 된다.

→ 부모 노드에서 인덱스 값을 지우기 전에 부모 노드가 비워질 수도 있으므로 parent.key_list[0]과 방금 정리한 왼쪽 노드를 key와 child로 저장한다.

→ 자식 노드의 parent를 왼쪽 형제 노드로 다시 정리해준다.

→ 만일 자식 노드가 리프 노드라면, prev_node와 right_node연결을 다시 해준다.

→ 만일 이렇게 만든 자식 노드가 키 범위 안에 해당되며, 루크 노드였던 부모 노드가 지워지게 되면, 이 자식 노드가 루트 노드가 되며, 이 함수를 반환한다.

②오른쪽 형제와 합치는 경우 (해당 노드가 부모 노드의 첫 번째 노드인 경우)

→ 위 ①에서 진행한 방식과 똑같은 논리로 진행한다.

2. 해당 노드가 최대 키 범위보다 큰 경우, split을 진행해줘야 한다.

→ split후 부모 노드로 올라가는 노드를 top으로 받는다.

→ 부모 노드가 비워져 있는 경우, top이 저절로 parent가 된다.

→ 부모 노드가 비워져 있지 않으면, 부모 노드에 top을 합친다.

(이때, top과 부모 노드의 key_list 원소를 비교해 top이 들어갈 위치를 찾은 후 더해준다.)

→ top의 자식 노드를 parent로 재설정해준다.

3. 해당 노드가 키 범위 안에 잘 들어온 경우, 해당 부모 노드를 Balance함수를 진행해, parent노드가 키 범위 안에 드는 지 확인한다.

① 만일 부모 노드가 지워졌을 때, 비워지며, 부모 노드가 루트 노드인 경우

→ 해당 노드의 부모 노드가 None이 되며, 이 노드가 루트 노드가 된다.

→ 그냥 이 노드를 반환해도 되지만 Balance 함수를 한번 더 들어가서 반환했다.

② 만일 부모 노드가 지워지게 되면, Empty_node 함수로 이동한다.

③ 만일 부모 노드가 지워지지 않게 되면, 계속해서 부모 노드가 키 범위 조건을 충족하는 지 확인한다.

3. Leaf

-Add_Leaf(self, input_key, input_value):input_key와 value 값을 {key : value}쌍으로 리프 노드에 추가하는 함수

→ 이 노드가 비워져 있는 경우에 이 함수를 호출하는 경우, 재정렬할 필요 없이 정보를 추가해준다.

→ 만일 이미 존재하는 key의 값을 넣었을 때, "Duplicated keys are not allowed!"를 print하며 break한다.

→ 만일 input_key가 존재하지 않는다면, Leaf노드게 삽입하고, 이후에 키를 기준으로 오름차순 정렬을 한다.

-split(self): 리프 노드를 두개로 쪼개고 부모 노드에 split key를 넣어주는 함수

두 개로 쪼개지는 리프 노드를 Right, Left로 설정하며, 두 노드를 리프 노드로 초기화 시켜준다.

두 개로 쪼개지는 기준(mid)는 key개수를 2로 나누고 내림한다.

1. Right 리프 노드 데이터 정리

→ NonLeaf 노드와 달리, 리프 노드가 쪼개지면, 기준으로 올라가는 노드의 키 값과 리프 노드의 키 값이 중복 된다.

→ Right 리프 노드는 해당 노드의 mid부터 끝까지 해당하는 정보를 가지고 있다.

→ Right 노드의 right_node와 prev_node를 정리해준다. 여기서 Left노드를 self로 받을 것이므로, Right 노드의 prev_node는 self가 된다.

2. Left 리프 노드 데이터 정리

→ Left노드를 self로 받고, pair, cnt_key, key_list 데이터를 초기화 시킨 후 해당되는 정보를 추가시켜준다.

→ Left노드의 데이터는 해당 노드(self)의 처음부터 mid-1에 해당하는 정보를 가지고 있다.

3. top 노드의 데이터 정리

→ 중간 키 값이 부모 노드가 되므로, 리프 노드가 아닌, Non Leaf로 초기화 시켜준다.

→ top의 {key : left child}는 {Right.key_list[0] : Left}가 된다. 왜냐하면 위에 언급했듯이 리프 노드가 쪼개지면, 해당 키 값과 리프 노드의 키 값이 중복되기 때문이다.

이와 같은 과정이 끝난 후, Left와 Right를 부모 노드인 top과 연결시켜준다.

-Search_Leaf(self, key): 해당 키 값 (key)이 리프 노드에 존재하는지 확인하는 함수

→ 리프 노드의 키 개수 만큼 for문을 실행하면서 key와 리프 노드의 key_list의 원소와 비교하며 key_list에서의 인덱스 값을 반환한다.

→ 만일 존재하지 않는다면, -1을 반환한다.

-Delete_Leaf(self, index): 노드에서 해당 인덱스에 해당하는 쌍을 삭제하는 함수

→ 해당 노드의 prev_node와 right_node를 미리 저장해둔다.

→ pair에서 해당 index쌍을 삭제한다. key_list에서도 삭제를 하며, cnt_key를 하나 줄인다.

→ 만일 index가 0이면 (노드의 첫 번째에 위치한 경우), self.prev_node를 미리 저장해둔 prev_node로 지정한다.

→ 만일 index가 cnt_key-1이거나 -1이면 (노드의 맨 마지막에 위치한 경우), self.right_node를 미리 저장해둔 right_node로 지정한다.

-isRight(self): 오른쪽 형제 노드가 있는지 확인하는 함수

만일 right_node가 존재하며, right_node의 parent와 해당 노드(self)의 parent가 동일하면, 오른쪽 형제 노드가 있으므로 True를 반환하고, 그렇지 않으면 False를 반환한다.

-isPrev(self): 왼쪽 형제 노드가 있는지 확인하는 함수

만일 prev_node가 존재하며, left_node의 parent와 해당 노드(self)의 parent가 동일하면, 왼쪽 형제 노드가 있으므로 True를 반환하고, 그렇지 않으면 False를 반환한다.

-Borrow_From_Left(self, index): 해당 노드의 키를 삭제할 때, 왼쪽 형제 노드에서 키 값을 가져오는 함수

왼쪽 형제 노드(Left)는 해당 노드의 prev_node이다. (self.prev_node)

Left에서 가져오는 키를 L_key라고 하며, 이는 Left의 가장 마지막에 위치한 키(Left.key_list[-1])이다.

왼쪽 형제 노드에서 키 값을 하나 가지고 오게 되면, 해당 노드(self)의 부모 노드의 key_list도 변한다.

이 때문에 앞서 NonLeaf 클래스에 있었던 change함수를 쓸 예정이므로, 부모 노드에서 바뀌기 전 키 값 (ex_key)를 미리 구해두는데 이는 원래 노드의 첫 번째 키 값 (self.key_list[0])이다.

→ 원래 노드에 있던 key와 value를 삭제한다.

→ 노드에 형제 노드의 키와 값을 추가한다.

→ 앞서 설명해듯이 부모 노드의 키 값이 ex_key에서 왼쪽 형제에서 가져온 키(L_key)로 바뀌어야 하므로, change(ex_key, L_key)를 해준다.

-Borrow_From_Right(self, index): 해당 노드의 키를 삭제할 때, 오른쪽 형제 노드에서 키 값을 가져오는 함수
오른쪽 형제 노드(Right)는 해당 노드의 right_node이다.
위에 설명한 Borrow_From_Left 함수와 동일한 논리이다.
하지만, 여기서는 오른쪽 형제 노드에서 키를 가져오기 때문에, 가져오는 키 값은 Right노드의 가장 첫 번째 key(R_key, Right.key_list[0])이다.
또한, 부모 노드의 키가 R_key에서 삭제를 진행한 Right노드의 첫 번째 key값 (Right.key_list[0])이다.

-merge(self): 형제 리프 노드와 병합하는 함수, 병합한 형제 리프 노드를 반환한다.

1. 왼쪽 형제 노드가 존재할 때, 왼쪽 형제 노드와 병합을 진행한다.
왼쪽 형제 노드가 존재하면, self.isPrev() == True이다.
왼쪽 형제 노드(Left)는 해당 노드의 prev_node (self.prev_node)이다.
Left 노드에 해당 노드의 key와 value정보들을 삽입해준다. 이때 NonLeaf 노드와 달리 부모 노드의 키 값을 넣지 않아도 된다 (이미 리프 노드에 존재해있음)
Left 노드의 right_node는 해당 노드의 right_node이다.
해당 노드가 맨 오른쪽 노드가 아닌 경우 해당 노드의 prev_node를 Left로 연결시켜준다.
2. 왼쪽 형제 노드가 존재하지 않지만, 오른쪽 형제 노드가 존재할 때, 오른쪽 형제 노드와 병합을 진행한다.
앞서 설명한 1번과 동일한 과정을 거친다.
오른쪽 형제 노드(Right)는 해당 노드의 right_node이다.
만일 해당 노드가 맨 처음 노드가 아닌 경우, 해당 노드의 right_node를 Right로 연결시켜준다.

4. BPTree

-find(self, node, key): node(NonLeaf)에서 key값을 비교해 해당되는 자식 노드로 이동하는 함수

1. 찾으려는 key 값이 node의 key_list에 있는 key값보다 작으면, 왼쪽으로 이동함
→ node.pair[node.key_list[i]]로 이동, 여기서 node는 NonLeaf이므로 pair의 value에 해당하는 값은 왼쪽 자식 노드를 가리킨다.
2. 찾으려는 key 값이 node의 key_list에 있는 가장 큰 key값과 같거나 크면, 오른쪽으로 이동함
→ node.right_node로 이동, 여기서 node는 NonLeaf이므로 right_node는 가장 오른쪽에 위치한 자식 노드이다.

-merge(self, parent, child): child 노드가 parent node에 합쳐지는 경우

이때 child노드는 리프 노드를 split한 후, 부모 노드로 올라가는 노드이다. (split()한 후 반환되는 노드이다.)
child 노드의 가장 첫 번째에 해당하는 key(부모 노드로 삽입할 key)를 index로 받는다.
index와 부모 노드의 키 값들을 비교해 child 노드의 키가 들어갈 자리를 찾아 merge한다.
→ 만일 삽입하려는 키가 parent노드의 마지막이 아닐 때
{index : child.pair[index]}를 부모 노드에 추가하고 키를 기준으로 오름차순 정렬을 해준다.
parent.pair[parent.key_list[i+1]]의 값이 child노드의 right_node가 된다.
child의 자식 노드의 부모 노드가 parent로 변경이 된다.

→ 만일 삽입하려는 키가 parent노드의 마지막일 때

{index : child.pair[index]}를 부모 노드에 추가하고 키를 기준으로 오름차순 정렬을 해준다.

parent.right_node 의 값이 child노드의 right_node가 된다.

child의 자식 노드의 부모 노드가 parent로 변경이 된다

-insert(self, key, value): 데이터(key와 value)를 삽입하는 함수

비벨트리의 루트 노드부터 시작해서 key가 들어가야할 리프 노드를 찾아 key와 value데이터를 추가해준다.

(비벨트리의 삽입과 삭제는 리프 노드에서 이뤄진다.)

먼저 child를 루트 노드로 받는다. 이때 parent = None이다.

child가 리프 노드가 될 때까지 find함수를 실행시킨다.

key가 추가되어야 할 리프 노드(while문이 끝난 child)에 데이터를 추가시킨다. (child.Add_Leaf(key, value))

만일 child에 키를 추가했을 때, 최대 키 범위를 넘게 되면, split을 실행한다.

(이때 만일 child가 루트 노드이면, split을 실행시켰을 때 부모 노드가 되는 노드가 루트 노드가 된다. 하지만, child가 루트 노드가 아니면, 이미 존재하는 parent와 merge를 진행해야 한다.)

-search(self, key): 찾으려는 key가 있는 리프 노드와 인덱스를 반환한다.

위에 insert와 비슷한 논리로, 루트 노드부터 시작해서 찾으려는 key가 있는 리프 노드에 도착할 때까지 find함수를 실행시킨다.

while문 실행이 끝난 child는 찾으려는 key가 있는 리프 노드이다.

리프 노드(child)에서 key의 위치(index)를 반환하는 함수인 Search_Leaf 함수를 실행한다.

만일 Search_Leaf 함수에서 -1을 반환하면, key가 child노드에 존재하지 않는 경우이다.

search 함수는 찾으려는 key가 있는 리프 노드(child)와 인덱스(index)를 반환한다.

-restruct(self, node): node의 키 값을 재구성하는 함수, 맨 처음에 트리의 루트 노드를 node값으로 받는다.

삽입, 삭제 등의 과정을 진행하다보면, 리프 노드의 키 값이 바뀌면서 부모 노드의 키 값도 바뀌는 경우가 있다.

그래서 모든 트리 조정 과정을 마친 이후 리프 노드의 키 값을 참고하여 부모 노드들의 키 값을 조정한다.

부모 노드(node)의 키 값은 오른쪽에 위치한 자식의 리프 노드에서 가장 작은 값이다.

1. 키가 하나만 있을 경우

→ child = node의 오른쪽 자식 노드, 이 경우에는 node.right_node

→ child노드에서 리프 노드로 갈 때까지 계속해서 왼쪽(child.pair[child.key_list[0]])으로 이동

→ node의 새로운 키(new_key)는 리프 노드에서 가장 작은 키 값이다.

→ node의 node.cnt_key-1번째 pair와 key_list를 모두 new_key로 변경한다.

2. 키가 여러 개 있을 경우

① node.key_list[node.cnt_key-2]까지의 key의 경우

→ child = node의 오른쪽 자식 노드, 이 경우에는 node.pair[node.key_list[i+1]]

→ 위 1번과 동일한 방식으로 node의 key_list의 원소들의 값을 변경한다.

② node.key_list[node.cnt_key-1]의 경우

→ child = node의 오른쪽 자식 노드, 이 경우에는 node.right_node

→ 위 1번과 동일한 방식으로 node의 key_list[node.cnt_key-1]의 값을 변경한다.

node의 모든 자식 노드들에 대해서도 재구성의 과정을 거친다. 이를 위해 재귀함수를 사용한다.

-delete(self, key): key에 해당하는 데이터를 지우는 함수

search함수를 통해 key가 존재하는 리프 노드와 리프 노드에서의 인덱스 값을 받는다.

만일, search함수에서 None을 반환하면, 해당하는 키가 트리에 존재하지 않는 경우이므로 delete 함수를 진행하지 않는다.

또, 해당 리프 노드가 루트 노드라면 키 개수와 상관없이 해당 리프 노드의 키와 value 값만 고치면 된다.

1. node에 key를 지웠을 때 리프 노드에서 최소 키 개수를 넘을 때

→ 단순히 해당 리프 노드의 키와 value값만 고치면 된다. (Delete_Leaf)

2. node에 key를 지웠을 때 리프 노드에서 최소 키 개수를 넘지 못할 때

2-1) 형제 노드에서 원소를 가져와서 재분배

→ 형제 노드가 존재하며, 형제 노드가 최소 key개수를 초과한다.

2-1-1) 왼쪽 노드에서 가져오는 경우: node.Borrow_From_Left(index)

2-1-2) 오른쪽 노드에서 가져오는 경우: node.Borrow_From_Right (index)

2-2) 재분배 실패 -> 형제 노드와 병합

이때 부모 노드 입장에서 자식 노드가 하나 줄어들기 때문에 부모 노드의 키 개수에 따라 경우를 나눠야한다.

2-2-1) 부모 노드가 최소 키 개수를 넘을 경우

→ 해당 리프 노드에서 해당 key와 value값을 삭제한다.

→ 형제 노드와 병합한다. (merge)

→ 부모 노드에서 비워지는 자식 노드(node)를 지운다.

2-2-2) 부모 노드가 최소 키 개수를 넘지 못하는 경우, 재구조화(Balance)가 이뤄져야한다.

① 부모 노드가 루트 노드일 경우

→ 만일 부모 노드의 키 개수가 1개라서 사라지게 되면, 병합한 형제 노드가 루트 노드가 된다.

→ 만일 부모 노드가 사라지지 않게 되면, 루트 노드 유지한다.

② 부모 노드가 루트 노드가 아닌 경우

→ 부모 노드의 키가 1개만 존재할 수도 있으므로, 부모 노드의 키를 temp_key로 임시 저장한다.

→ 조정된 부모 노드의 키가 0개인지 아닌지에 따라 Balance함수로 넘어가는 매개변수 값이 다르다.

삭제를 모두 진행한 다음, Non Leaf의 인덱스 값이 삭제되어야 하는데 삭제되지 않은 경우가 있어 트리의 키 값들을 재구성하는 과정을 거쳐야한다. (self.restruct(self.root))

-Single_Search(self, key): 단일 키 (key) 탐색하는 함수

위에 진행한 search함수와 동일한 논리로 진행한다.

루트 노드에서부터 시작하여 해당 키가 있는 리프 노드를 찾는다. 거쳐간 노드의 모든 key값을 출력해야 하므로, print(", ".join(str_key))를 진행한다.

리프 노드로 이동한 후, key가 리프 노드의 어느 위치에 존재하는지 찾고, 이를 기반으로 value를 print한다.

만일 key가 리프 노드에 존재하지 않으면, index는 -1이 되므로 이때는 "NOT FOUND"를 출력한다.

-Ranged_Search(self, start_key, end_key): 범위 키(start_key ~ end_key) 탐색 함수

위에 진행한 Single_Search와 비슷한 논리로 진행한다.

마찬가지로 리프 노드에 도달할 때까지 while문을 실행한다.

키 범위에 해당하면 key와 value를 출력한다. 리프 노드의 첫 번째 키가 end_key보다 커질 때까지 right_node로 이동한다. (비벨 트리는 리프 노드가 모두 연결되어 있어서 가능하다.)

-list_Leaf(self): 모든 리프 노드의 {key:value}를 출력하는 함수 (인덱스 파일에 저장할 때 필요함)

비벨트리의 가장 첫번째 리프 노드로 이동할 때까지 while문을 실행한다.

마지막 리프 노드에 도달할 때까지 (child.right_node == None이 될때까지) {key:value} pair를 추가해준다.

-printTree(self, node): 트리를 출력하는 함수

만일 루트 노드가 없거나, 루트 노드의 키 개수가 0개이면, 비어 있는 트리이다. 이 경우에는 "This Tree Is Empty"를 출력한다.

리프 노드가 아닌 경우, 노드의 key_list를 출력한다. 그리고 자식노드들을 다시 printTree를 실행해 출력한다.

리프 노드일 경우, 노드의 key와 value를 출력한다.

III. Instructions for compiling your source codes at TA's computer

1. Data File Creation (index file : index.dat, size of each node = 8)
2. Insertion (index file : index.dat, data_file : input.csv)
3. Deletion (index file : index.dat, data_file : delete.csv)
4. Single Key Search (index file : index.dat, key : 87)
5. Ranged Search (index file : index, start_key : 1, end_key : 40)

순서대로 실행시킨 모습

```
C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -c index.dat 8
Data File Creation

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -i index.dat input.csv
Insertion

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -d index.dat delete.csv
Deletion

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -s index.dat 87
Sing key Search
984796

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -r index.dat 1 40
Ranged Search
37, 2132
```

→ 만일 insertion과 deletion 실행 시 트리 구조를 보고 싶으시다면, bptree.py의 990, 1022줄에 주석처리한 부분을 해제해주세요! (주석을 해제하면 다음과 같이 트리가 출력됩니다.

```
C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -i index.dat input.csv
Insertion
[68]

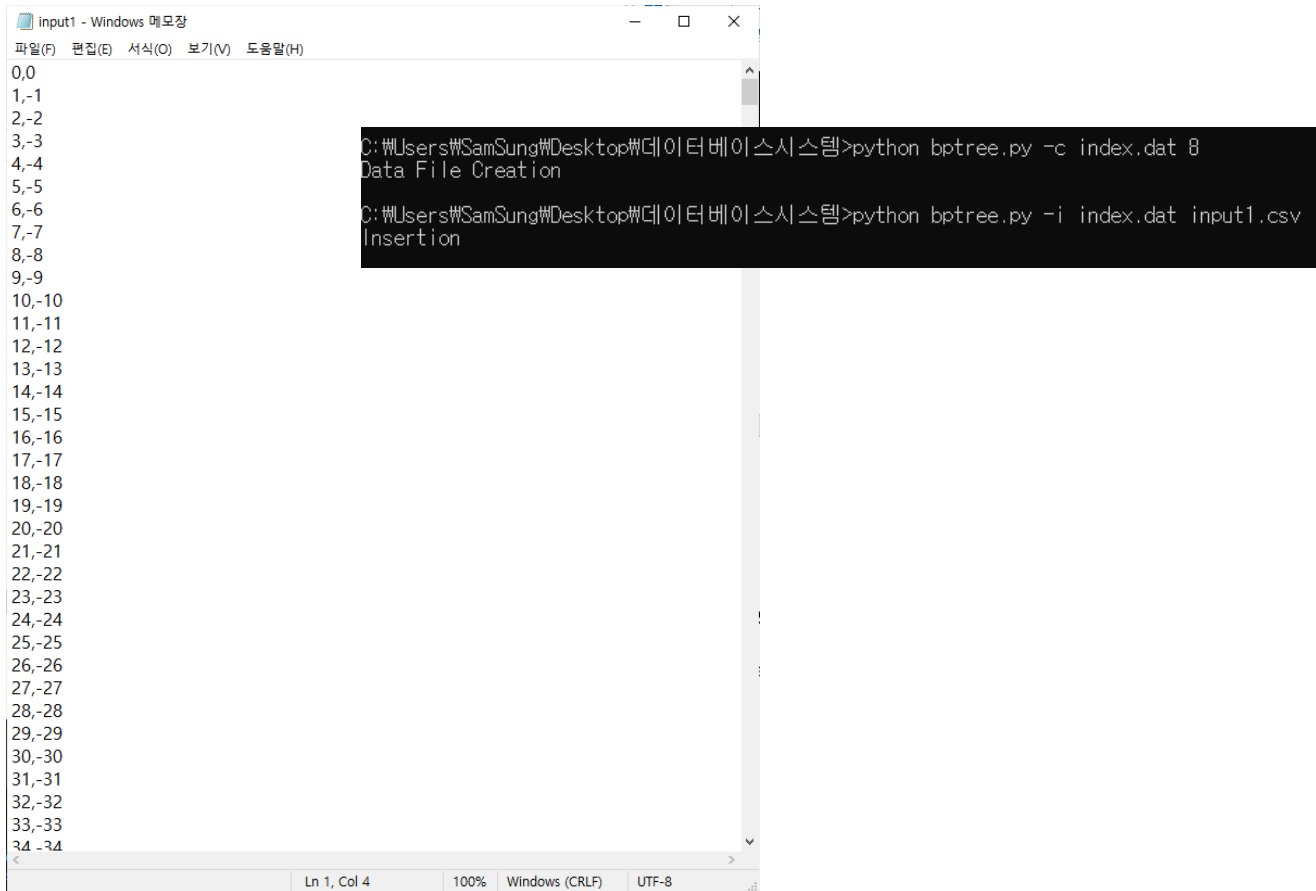
key 9 : value 87632
key 10 : value 84382
key 20 : value 57455
key 26 : value 1290832
key 37 : value 2132

key 68 : value 97321
key 84 : value 431142
key 86 : value 67945
key 87 : value 984796

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -d index.dat delete.csv
Deletion
key 37 : value 2132
key 68 : value 97321
key 84 : value 431142
key 86 : value 67945
key 87 : value 984796
```

IV. Other testing example

degree가 8인 비블트리에 0~999까지 다음처럼 생긴 같은 input data file (input1.csv)을 넣는다.



```
input1 - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
0,0
1,-1
2,-2
3,-3
4,-4
5,-5
6,-6
7,-7
8,-8
9,-9
10,-10
11,-11
12,-12
13,-13
14,-14
15,-15
16,-16
17,-17
18,-18
19,-19
20,-20
21,-21
22,-22
23,-23
24,-24
25,-25
26,-26
27,-27
28,-28
29,-29
30,-30
31,-31
32,-32
33,-33
34,-34
Ln 1, Col 4 100% Windows (CRLF) UTF-8
```

```
C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -c index.dat 8
Data File Creation

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -i index.dat input1.csv
Insertion
```

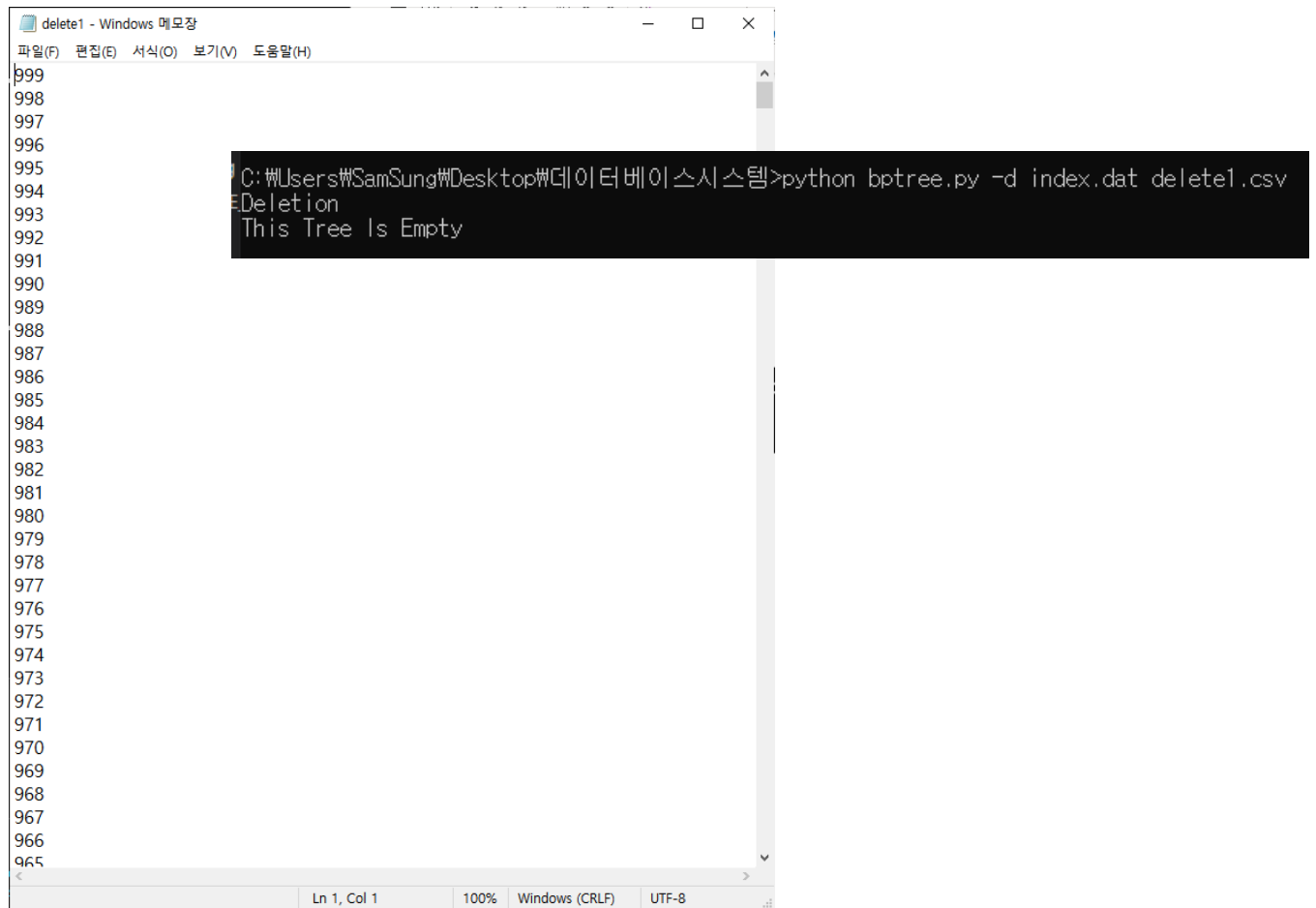
i) key = 50인 value를 찾는다.

```
C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -s index.dat 50
Sing key Search
500
100, 200, 300, 400
20, 40, 60, 80
44, 48, 52, 56
-50
```

ii) 10~20까지인 key의 value를 찾는다.

```
C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -r index.dat 10 20
Ranged Search
10, -10
11, -11
12, -12
13, -13
14, -14
15, -15
16, -16
17, -17
18, -18
19, -19
20, -20
```

iii) 다음과 같이 999~0으로 내림차순으로 정리된 key 값이 있는 delete1.csv을 삭제한다



```
delete1 - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
999
998
997
996
995
994
993
992
991
990
989
988
987
986
985
984
983
982
981
980
979
978
977
976
975
974
973
972
971
970
969
968
967
966
965

C:\Users\SamSung\Desktop\데이터베이스시스템>python bptree.py -d index.dat delete1.csv
Deletion
This Tree Is Empty
```