Database Systems
2022074057
홍바다

B+ Tree Implementation Assignment

Summary of Algorithm

I. Tree Structure

- class Pair는 key, value, 그리고 왼쪽 자식 노드 left_child를 포함한다.
- class BPlusTreeNode에는 각 노드에 대한 정보를 저장하기 위해 사용된다. 해당 노드가 가지고 있는 key의 개수인 m, Pair를 저장하는 list p, rightmost child node 또는 right sibling node를 가리키는 r, 해당 노드가 leaf인지 아닌지 알려주는 is_leaf, 그리고 부모 노드를 가리키는 parent가 포함된다.
- class BPlusTree는 B+ Tree에 대해 load, save, insertion, deletion, single key search, 그리고 ranged search를 실행한다. 해당 클래스에는 tree의 degree와 BPlusTreeNode 타입인 root가 있다.

II. Loading and Saving B+Tree

- load_tree(): dat 파일에 저장되어 있는 트리를 build_node와 build_tree_from_data를 이용해서 로드한다.
- save_tree(): tree의 degree를 먼저 저장한 후, 각 노드 정보를 저장한다.
    - _save_node를 이용해서 각 노드의 leaf node인지(1) 아닌지(0), key 개수, 그리고 가지고 있는 Pair 정보를 나열한다. 이때, Pair는 “key,value ” 형식으로 저장된다.

III. Data File Creation

- command line에서 받은 degree 정보를 BPlusTree의 degree에 저장한 후 첫 줄에 정보를 입력하여 dat 파일을 생성한다.

IV. Insertion

- load_tree를 통해 dat 파일의 BPlusTree 정보를 가져온다. input이 저장된 csv 파일에서 받은 “key,value “ 값을 각각 받아 insertion()을 실행한다.
- find_leaf_node() 함수를 이용하여 해당 key를 포함할 수 있는 범위를 가진 leaf node를 recursive하게 찾는다. 해당 leaf node에서 key가 들어갈 수 있는 인덱스, key보다 작은 값을 가지는 자리의 다음 인덱스를 찾는다. 만약 key가 이미 존재한다면 리턴한다.
- 인덱스를 찾았다면, add_at_index를 이용하여 leaf node에 저장한다.
- 삽입 후 만약 leaf의 key 개수가 degree - 1보다 많으면 split_leaf_node를 통해 split을 해준다.
- split_leaf_node에서는 leaf node의 중간값을 찾아 두개의 노드로 나눈 후 propagate_split을 통해 두개로 나뉘어진 노드를 중간값을 포함한 부모 노드와 연결시켜준다. 만약 leaf 노드가 root였다면 단순히 중간값을 새로운 노드에 저장하여 root로 만들어 준다.
- 중간값을 넣으면서 부모의 key 개수가 초과된다면 이제 split_nonleaf_node를 이용하여 다시 부모 노드에 대하여 분배해준다. 부모 노드의 중간값을 구해 두개의 노드로 나누어 주고, 이를 propagate_split에서 중간값을 포함한 부모의 부모 노드와 연결시켜준다. 만약 부모의 부모 노드의 key 개수가 degree 이상이면 다시 split_nonleaf_node와 propagate_split을 초과되지 않을 때까지 반복한다.
- 이렇게 overflow 문제를 해결한 완성된 tree의 정보를 save_tree를 통해 dat 파일에 저장한다.

V. Deletion

- Insertion과 같이 load_tree를 통해 tree 정보를 가져온다. csv에 저장되어 있는 삭제될 key 값을 받아 deletion()을 실행한다.
- find_leaf_node() 함수를 이용하여 삭제하고자 하는 key가 포함 될 수 있는 leaf node를 찾는다. 해당 key가 존재하는 인덱스를 찾은 후, 없다면 에러 메시지와 함께 리턴한다.
- delete_from_leaf를 통해 노드로부터 해당 key를 삭제한다. 삭제 후 만약 key가 해당 노드에서 최소값이었다면 key를 갖고 있는 internal node들의 값을 update_parent를 통해 삭제 후 최소값으로 바꿔준다. 그리고 삭제 후 underflow가 일어나지 않는다면 (degree/2 - 1을 충족한다면), 바로 위 부모 노드의 값만 업데이트 해준 후 리턴한다.
- 만약 underflow가 일어났고 leaf가 root도 아니라면, borrow 또는 merge를 통해 조건을 충족해준다. 먼저, left sibling node가 최소 개수(degree/2 - 1) 초과면, 왼쪽에서 마지막 Pair 하나를 borrow한다. left sibling이 존재하지 않거나 최소 개수만을 가지고 있다면 right sibling에서 첫번째 Pair 하나를 borrow한다. 만약 right sibling도 조건을 충족하지 못하면 merge_leaf를 통해 두 노드를 합병한다. left sibling이 존재하면 왼쪽 노드와 합병을, 아니면 right sibling과 합병한다.
- merge_leaf에서는 왼쪽 노드에 오른쪽 노드를 합치고, 둘을 분리하고 있는 부모 노드의 Pair를 find_parent_index로 인덱스를 찾아 삭제한다. 부모 노드의 삭제된 다음 인덱스의 left_child를 합쳐진 왼쪽 노드로 설정해준다. 만약 마지막 Pair라면 parent.r에 저장해준다.
- 부모 노드에서 하나의 Pair를 삭제 후 부모 노드에서 underflow가 일어나면, root인 경우 tree의 height를 한단계 낮춰주고, 그것이 아니라면 handle_internal_underflow를 통해 nonleaf node의 underflow를 해결해준다. leaf 노드에 대해 한 것처럼, nonleaf도 underflow가 일어나면 left sibling의 마지막 key 또는 right sibling의 첫 key를 borrow한다. 이때에는 두 노드를 분리하는 부모노드의 key 값도 함께 빌려온다. 각각에 left_child와 parent 설정을 해준 후 left sibling의 마지막 key 또는 right sibling의 첫번째 key를 삭제한다. 만약 둘 다 조건을 충족하지 못하면 merge_nonleaf를 통해 두 노드를 합병한다. left sibling이 존재하면 왼쪽 노드와 합병을, 아니면 right sibling과 합병한다.
- merge_nonleaf에서는 먼저 left node에 두 노드를 분리하는 Pair를 삽입한 후 right node의 전체 p를 삽입한다. 각각에 부모와 자식 설정을 해준 후 둘을 나누는 부모 노드의 Pair를 삭제한다. 여기서 부모 노드에서 underflow가 일어난다면, root일 경우 tree의 height을 낮춰주고, 아니면 다시 handle_internal_underflow로 가 부모 노드가 조건을 충족할 때까지 과정을 반복한다.
- 모든 underflow 문제를 해결하면 완성된 tree의 정보를 save_tree를 통해 dat 파일에 저장한다.

VI. Single Key Search

- tree의 root부터 시작해서 leaf를 찾을 때까지 왼쪽 자식 노드를 통해 recursive한 과정을 거친다. leaf에 도달하기까지 거친 internal node들의 정보를 출력해준다. 찾고 싶은 key를 포함할 수 있는 범위를 가진 leaf를 찾으면 key가 있는지 확인 후 value를 출력해준다. 만약 key를 못 찾을 경우 “NOT FOUND”를 출력한다.

VII. Ranged Search

- 시작 key가 포함될 수 있는 범위를 가진 leaf 노드를 find_leaf_node를 통해 찾는다. 시작과 끝 key 사이의 범위에 해당하는 모든 Pair의 값을 출력해준다. B+ tree의 장점인 선형탐색을 위해 BPlusTreeNode의 변수 r(right sibling)을 이용하여 옆 leaf node로 이동하며 값을 확인한다.

Detailed Description of Codes

I. class Pair

- 각 노드에 저장되는 key와 value, 그리고 해당 key의 왼쪽 자식 노드를 가리키는 left_child를 저장한다. leaf 노드에 존재하는 Pair라면 left_child에는 None이 저장된다.

```
def __init__(self, key, value, left_child=None):
    self.key = key
    self.value = value
    self.left_child = left_child # left child pointer
```

II. class BPlusTreeNode

- B+ tree에 존재하는 각 노드의 key 개수 m, key를 저장하는 Pair 타입의 리스트 p, leaf node일 경우에는 right sibling, nonleaf일 경우에는 rightmost child 노드를 가리키는 r, leaf인지 아닌지 알려주는 is_leaf, 그리고 해당 노드의 부모노드를 가리키는 parent를 저장한다.

```
def __init__(self, is_leaf=False):
    self.m = 0 # number of keys
    self.p = [] # array of Pair
    self.r = None # pointer to rightmost child node or right sibling node
    self.is_leaf = is_leaf
    self.parent = None
```

- def add_pair(self, key, value, node=None):
  Pair 값을 p 리스트의 끝에 덧붙여준 후 m의 크기를 1 증가시킨다.
- def add_at_index(self, key, value, index):
  Pair 값을 p 리스트의 index 자리에 추가해준 후 m의 크기를 1 증가시킨다.
- def remove_pair(self, index):
  p 리스트의 index 자리에 존재하는 Pair 값을 삭제해준 후 m의 크기를 1 감소시킨다.
- def set_left_child(self, index, node):
  set 함수로, 해당 노드의 index에 존재하는 Pair에 left_child 값으로 node를 설정해준다 (self.p[index].left_child = node). 만약 index 값이 m과 같다면 rightmost child로 설정하기 위해 self.r = node를 해준다.
- def get_left_child(self, index):
  해당 index 자리의 left_child 값을 반환해준다. 만약 index 값이 m과 같다면 rightmost child이므로 r 값을 반환해준다.

III. class BPlusTree

- B+ tree 정보를 저장하는 클래스로 root를 가르키는 BPlusTreeNode 타입의 root와 해당 트리의 degree 정보를 저장하는 degree 값을 저장한다.

```
def __init__(self, degree=0):
    self.root = BPlusTreeNode()
    self.degree = degree # degree from input
```

- **def load_tree(self, index_file):**
  index_file에 저장되어 있는 정보를 read mode로 파일을 연다. 해당 함수는 data file creation에 대한 command가 미리 실행 되었다는 가정하에 실행된다. 파일의 첫번째 줄의 내용은 트리의 degree 정보이므로 self.degree에 해당 정보를 저장한다. 그 뒤에 모든 내용은 nodes에 대한 데이터로, nodes_data에 저장한다. 만약 nodes_data에 아무 정보가 없다면, 즉 트리가 비어있다면 self.root를 is_leaf=True를 통해 리프

노드로 만들어준 후 아무 내용도 저장하지 않고 리턴한다. 노드에 대한 내용이 있다면 build_tree_from_data(0)을 실행한다.
해당 함수에서만 쓰이는 두개의 내장함수가 있다:

a. **def build_node(data_line):**

- 데이터를 이용해 노드를 생성하는 함수이다.
- 먼저, line에 저장되어 있는 데이터를 space를 기준으로 split해준다. 각 line의 첫번째 정보는 리프노드인지 아닌지 1과 0으로 저장되므로 is_leaf에 0 또는 1 값을 bool 타입으로 저장해준다. 두번째 정보는 key 개수이므로 m에 정수 타입으로 저장해준다.
- BPlusTreeNode(is_leaf=is_leaf)를 이용해서 leaf 또는 internal node를 생성해준다. 해당 노드의 node.m값은 데이터에서 받아온 m 값이 된다.
- 두번째 정보 후의 모두 데이터는 key-value pair의 정보다. 이들은 모두 “key,value”로 이루어져있으므로 “,”를 기준으로 key와 value 값을 저장해주고, node.p에 Pair(key, value)를 저장해준다. 그리고 생성된 노드를 리턴해준다.

b. **def build_tree_from_data(index, last_leaf=None):**

- 생성된 노드를 이용해서 tree를 만드는 함수이다.
- 먼저, index는 가져온 데이터에서의 위치이다. 만약 노드정보가 nodes_data에 더이상 남아있지 않다면 None 값의 노드와 함께 바로 리턴한다.
- build_node(nodes_data[index])를 이용해서 현재 위치하는 index의 데이터를 이용해 노드를 생성한다. 그리고 index를 1 증가시켜 다음 위치를 나타낸다.
- 만약 생성된 노드가 leaf가 아니라면, 다음 index의 정보는 자식노드에 대한 정보이다. internal node는 node.m+1만큼 자식을 가질 수 있으니 그만큼 for문을 반복한다. build_tree_from_data(index, last_leaf)를 recursive하게 불러 자식 노드들을 생성한다. 만약 자식 노드가 존재한다면 child.parent는 해당 노드가 되고, 해당 노드의 왼쪽 자식 노드는 node.set_left_child(i, child)를 이용해서 설정해준다. 모든 자식에 대해 설정이 끝났으면 rightmost child node는 node.get_left_child(node.m)으로 설정한다.
- 만약 생성된 노드가 leaf라면, last_leaf의 right sibling을 node로 설정해준다. 그리고 다시, last_leaf = node로 설정해준다. 마지막으로 해당 노드, index 값, last_leaf를 반환한다.
- 이를 모든 데이터에 대해 재귀적으로 반복하면 끝이난다.

모든 과정이 끝나면, build_tree_from_data(0)에서의 반환값 중 노드 값은 self.root가 된다.

- **def save_tree(self, index_file):**

먼저, index_file을 write mode로 연다. 파일의 첫번째 줄에는 tree의 degree 정보를 저장하기 위해 self.degree를 쓴다. 그리고 재귀 함수인 save_node 함수를 불러 전체 트리 정보를 저장한다.

a. **def save_node(self, node, index_file):**

- 먼저, 만약 노드가 비어있다면 리턴한다.
- 존재한다면 index_file에 첫 column에는 node.is_leaf 정보 (0 또는 1)을 저장하고 두번째에는 node.m을 저장한다. 그 뒤의 같은 줄에는 node.p에 대한 정보를 저장한다. “key,value ”형식으로 모든 Pair 정보를 저장하고 줄바꿈을 해준다.
- 만약 해당 노드가 leaf가 아니라면 자식 노드가 존재하니 node.m+1만큼 for문을 돌려주고, node.get_left_child(i), 즉 노드의 i번째 인덱스의 왼쪽 자식 노드에 대해 save_node(child, index_file)를 recursive하게 저장한다.
- 이를 root부터 leaf에 도달할때까지 반복하면 끝이난다.

- **def insertion(self, key, value):**
    a. 값을 삽입하기 위해 해당 key가 들어갈 수 있는 적절한 leaf node를 찾기위해 find_leaf_node(key)를 이용한다.
    b. leaf 노드에서 key가 들어갈 수 있는 인덱스 i를 찾는다. 만약 해당 자리에 key와 같은 값이 이미 저장되어 있다면 이미 존재한다는 메세지와 함께 리턴한다.
    c. 해당 leaf 노드에 i번째에 leaf.add_at_index(key, value, i)를 이용해서 삽입해준다. 만약 overflow, 즉 leaf.m이 self.degree - 1을 초과하지 않는다면 삽입이 끝난다.
    d. 그렇지 않고 overflow가 일어났다면, self.split_leaf_node(leaf)를 통해 leaf 노드를 분리해서 조건을 충족시켜준다.
- **def find_leaf_node(self, key):**
    a. self.root부터 시작한다. 만약 해당 노드의 i번째에 해당하는 key가 삽입하려는 key보다 크면 해당 i 위치의 왼쪽 자식 노드로 이동한다. 해당 과정을 리프노드에 도달할때까지 while문으로 반복한다. 리프노드에 도달했다면 해당 노드를 반환한다.
- **def split_leaf_node(self, leaf):**
    a. self.degree - 1를 초과한 leaf 노드를 split할때 사용한다. 중앙값인 self.degree // 2의 위치를 기준으로 leaf에 p의 first half의 정보를 저장하고, 나머지는 새로운 리프노드에 저장한다. 이때, new_leaf.r = leaf.r로 right sibling을 leaf의 right_sibling으로 설정한다. leaf.r은 이제 new_leaf가 된다. 그리고, self.propagate_split(leaf, new_leaf, new_leaf.p[0].key, new_leaf.p[0].value)을 불러 부모 노드에 대한 설정을 해준다.
- **def propagate_split(self, old_node, new_node, new_key, new_value):**
    a. 만약 old_node가 root였다면, 새로운 root를 만들어 old_node를 왼쪽 자식 노드로 설정한다. rightmost child로는 new_node를 저장한다. 새로운 root의 key와 value 값은 new_node의 key, value 값이 된다. 이제 root는 new_root가 되고, old_node와 new_node의 부모는 새로운 root가 된다.
    b. root가 아니라면, 부모 노드에서 old_node가 자식 노드로 존재할 수 있는 index i를 찾고, 거기에 new_node의 key, value 값을 삽입한다. 그리고, i의 왼쪽 자식 노드는 old_node, i의 오른쪽 자식 노드, 즉 다음 인덱스의 왼쪽 자식 노드는 new_node로 설정한다. new_node.parent를 old_node의 부모로 설정한다.
    c. 만약 부모 노드에서도 overflow가 일어나면, middle_key, middle_value, new_parent = self.split_nonleaf_node(parent)와 self.propagate_split(parent, new_parent, middle_key, middle_value)을 실행시킨다.
    d. 해당 과정은 root에 도달하거나 부모 노드가 overflow가 되지 않을 때까지 재귀적으로 반복한다.
- **def split_nonleaf_node(self, node):**
    a. overflow가 일어난 nonleaf 노드에 대해 두개의 노드로 분리시키는 함수이다.
    b. split_leaf_node처럼 새로운 노드와 기존의 노드에 대해 값을 분배한다.
    c. 자식 노드에 대한 설정을 위해 노드 분리 후 node.r에 node.get_left_child(middle)을 설정한다.

- **def deletion(self, key):**
    a. key가 존재할 수 있는 leaf 노드를 찾기 위해 find_leaf_node(key)를 사용한다.

b. leaf 노드에서 key의 값이 존재하는 인덱스 i를 찾는다. 만약 i가 leaf.m 값이라면 key가 존재하지 않으므로 반환한다.
c. self.delete_from_leaf(key, leaf, i)를 통해 key를 트리에서 지운다.

- **def delete_from_leaf(self, key, leaf, index):**
  a. 최소 key 개수를 min = (self.degree - 1) // 2로 설정한다. leaf에서 index에 해당하는 key를 leaf.remove_pair(index)로 삭제한다.
  b. 만약 해당 key가 노드의 첫번째 key였다면 부모 노드의 값을 leaf.p[0].key로 self.update_parent(leaf, leaf.p[0].key, leaf.p[0].value, key)를 통해 업데이트 해준다.
  c. 만약 leaf.m이 underflow가 아니라면, 즉 min값 이상이면 deletion을 끝낸다. 그전에 key가 첫번째 key였다면 바로 위 부모의 값을 self.update_parent_key(leaf)로 업데이트해준다.
  d. 만약 삭제 후 underflow가 일어났다면, 먼저 root인지 확인한다. root는 1 이상만 가지고 있으면 되므로 underflow가 일어나도 상관없다. 그러니 deletion 과정을 끝낸다.
  e. root도 아니고 underflow도 일어났다면, borrow 또는 merge를 통해 해결한다. 먼저 leaf가 왼쪽 자식으로 있는 부모 노드의 인덱스를 구한다. left_sibling과 right_sibling값을 부모 노드를 통해 아래와 같이 설정하여 구한다.

```
left_sibling = parent.get_left_child(parent_index - 1) if parent_index > 0 else None
right_sibling = parent.get_left_child(parent_index + 1) if parent_index < parent.m else parent.r
```

  f. 만약 왼쪽 노드가 존재하고 min을 초과하는 길이를 갖고 있으면 왼쪽에서 빌린다. leaf 노드에 left_sibling의 가장 마지막 값을 맨 앞에 삽입하고 left_sibling에서는 삭제한다. 각 노드의 바로 위 부모 노드의 값을 update_parent_key를 통해 업데이트 해준다. leaf의 첫번째 key 값을 갖는 상위 노드들의 값도 update_parent를 통해 업데이트 해준다.
  g. 만약 왼쪽에서 빌릴 수 없고, 오른쪽 노드가 존재하고 min 보다 많은 개수를 가지고 있으면 오른쪽에서 빌린다. leaf 노드의 마지막에 right_sibling의 첫번째 key를 삽입하고 right_sibling에서는 삭제한다. 여기에서도 바로 위 부모와 상위 노드들에 대해 값을 업데이트 해준다.
  h. 양쪽 모두에서 빌릴 수 없다면 merge를 한다. 왼쪽이 존재하면 왼쪽에서, 아니면 오른쪽과 합병한다. 왼쪽일 때는 self.merge_leaf(left_sibling, leaf, parent, parent_index - 1)로, 오른쪽일 때는 self.merge_leaf(leaf, right_sibling, parent, parent_index)로 실행한다.
  i. 모든 과정이 끝나면 leaf 노드의 바로 위 부모의 값을 다시 한번 확인차 업데이트 시켜준다.

- **def merge_leaf(self, left, right, parent, separator):**
  a. 모든 상황에서 left 노드에 right 노드를 합병한 후 부모 인덱스의 오른쪽에 설정한다.
  b. 먼저, left에 right를 삽입하고 m과 r을 설정한다.
  c. 부모 노드의 왼쪽 자식 노드를 설정하기 위해 인덱스가 m보다 작을 때, 인덱스와 같을 때를 고려하여 left를 자식 노드로 설정한다. m과 같은 길이일때는 parent.r = left로 설정한다. 그리고 부모 노드에서 left와 right를 분리시키던 key를 삭제하고, 부모 노드 key도 left의 첫번째 key로 바꿔준다.
  d. 만약 삭제 후 부모 노드가 underflow된다면, 즉 (self.degree - 1) // 2 보다 적고 root도 아니라면 self.handle_internal_underflow(parent)를 실행한다. 만약 root이고 길이가 0이면 left를 새로운 root로 설정한다.
  e. 모든 과정 이후 left 노드의 바로 위 부모의 값을 다시 한번 확인차 업데이트 시켜준다.

- **def handle_internal_underflow(self, node):**
    a. nonleaf 노드도 leaf 노드처럼 borrow와 merge를 통해 underflow를 해결한다.
    b. min = (self.degree - 1) // 2으로 설정하고, 부모 노드에서 find_parent_index를 통해 node가 자식으로 존재하는 인덱스를 찾는다.
    c. left_sibling과 right_sibling을 부모 인덱스를 사용하여 아래와 같이 설정한다.

```
left_sibling = parent.get_left_child(parent_index - 1) if parent_index > 0 else None
right_sibling = parent.get_left_child(parent_index + 1) if parent_index < parent.m else parent.r
```

    d. 만약 왼쪽 노드가 존재하고 min을 초과하는 길이를 갖고 있으면 왼쪽에서 빌린다. node의 맨 첫번째 index에 두 노드를 분리시키는 부모 노드의 key 값을 저장한다. 그리고 해당 키의 왼쪽 자식 노드로 left_sibling.r, 즉 rightmost child node를 저장한다. 그리고 해당 자식 노드의 부모로 node를 설정한다. 그리고 left_sibling.r은 가장 마지막 left_child로 설정한다. 부모 노드의 키 값도 left_sibling의 마지막 key를 이용해서 업데이트 해주고, left_sibling.remove_pair(-1)로 가장 마지막 값을 삭제한다.
    e. 만약 왼쪽에서 빌릴 수 없고, 오른쪽 노드가 존재하고 min 보다 많은 개수를 가지고 있으면 오른쪽에서 빌린다. node 노드의 마지막에 부모 노드의 key 값을 저장하고 left_child로는 node.r을 설정한다. node.r은 right_sibling의 첫번째 key의 left_child로 바꾼다. 부모 노드의 키 값을 right_sibling의 첫번째 key로 업데이트 해주고, right_sibling에서 삭제한다.
    f. 양쪽 모두에서 빌릴 수 없다면 merge를 한다. 왼쪽이 존재하면 왼쪽에서, 아니면 오른쪽과 합병한다. 왼쪽일 때는 self.merge_nonleaf(left_sibling, node, parent, parent_index - 1)로, 오른쪽일 때는 self.merge_nonleaf(node, right_sibling, parent, parent_index)로 실행한다.
- **def merge_nonleaf(self, left, right, parent, separator):**
    a. leaf 노드를 합병할 때와는 다르게, 부모 노드의 separator key를 포함하여 합병한다. 모든 상황에서 left 노드에 separator key와 right 노드를 합병한 후 부모 인덱스의 오른쪽에 설정한다.
    b. 먼저, left에 부모 노드의 separator를 삽입하고 right를 합병한다. m과 r도 설정한다. 합병한 key들의 left_child의 부모에 대한 설정을 left로 해준다.
    c. 부모 노드의 왼쪽 자식 노드를 설정하기 위해 인덱스가 m보다 작을 때, 인덱스와 같을 때를 고려하여 left를 자식 노드로 설정한다. m과 같은 길이일때는 parent.r = left로 설정한다. 그 후 부모 노드에서 separator를 삭제한다.
    d. 만약 삭제 후 부모 노드가 underflow라면, self.handle_internal_underflow(parent)를 실행시킨다. 다만, root일 때, 길이가 0일 경우 self.root를 left로 설정 후 삭제 과정을 끝낸다.
    e. 해당 과정은 handle_internal_underflow와 함께 부모 노드가 root이거나 부모 노드가 최소 key 개수 조건을 충족할 때까지 반복된다.
- **def find_parent_index(self, node):**
    a. 부모 노드에서 node를 자식으로 갖는 인덱스를 찾는 함수이다. 왼쪽 자식으로 node를 갖는 인덱스를 반환하며, 만약 rightmost child일 경우에는 부모 노드의 m 길이를 반환한다. index를 찾지 못하면 -1을 반환한다.
- **def update_parent(self, node, new_key, new_value, old_key):**
    a. 모든 상위 부모 노드에 대해 old_key를 갖던 key들을 new_key로 바꾸는 함수이다. 해당 함수는 부모 key를 가장 작은 key 값으로 설정하기 위함이다.
    b. 만약 root까지 설정하였다면 리턴한다.
- **def update_parent_key(self, node):**

a. node의 바로 위 부모 노드의 key 값을 최소 값으로 업데이트 해주는 함수이다.
b. find_parent_index(node)를 사용하여 부모 노드의 인덱스를 찾아 바꾼다.

- **def single_key_search(self, key):**
  a. self.root부터 시작한다. 만약 root가 None이라면 empty tree이므로 리턴한다.
  b. 루트부터 leaf node에 도달할 때까지 while문을 돈다. leaf에 도달할 때까지 거치는 모든 상위 부모들의 p에 존재하는 key 값들을 “key1,key2,...,keyn\n”으로 출력한다. 그리고 다시 node는 key가 속하는 범위에 해당하는 인덱스의 left child로 업데이트 해준다.
  c. 위 과정을 반복 후 leaf 노드를 찾으면 node.p에서 key를 찾는다. 찾으면 “value”값을 출력해주고, 아니라면 “NOT FOUND”값을 출력한다.

- **def ranged_search(self, start_key, end_key):**
  a. start_key가 속할 수 있는 범위에 해당하는 leaf node를 찾기 위해 self.find_leaf_node(start_key)를 사용한다.
  b. 만약 해당 노드가 None이 아니라면 node.p에서 start_key <= pair.key <= end_key 범위에 해당하는 모든 key 값들을 “key,value\n” 형태로 출력한다. 만약 한 key가 end_key보다 크다면 리턴한다.
  c. B+ tree는 선형탐색이 가능하기 때문에 node = node.r을 통해 다음 right sibling으로 옮겨가 다시 위 과정을 반복한다.

- **main:**
  a. 아래 코드와 같이 if-elif-else문을 사용하여 command line의 두번째 column의 정보에 따라 data file creation, insertion, deletion, single key search, ranged search를 진행하게 했다.
  b. data file creation: (sys.argv[1] == ‘c’) write mode로 파일을 열어 command line에서 받은 degree 값을 index file에 쓴다. 이때, 파일이 이미 존재한다면 덮어씌우게 되고, 존재하지 않다면 새로운 index file을 생성하여 degree를 저장한다.
  c. insertion: (sys.argv[1] == ‘i’) bptree를 load_tree를 통해 index file에서 데이터를 가져오고 저장한다. input csv file을 read mode로 열어 하나의 key와 value마다 insertion 함수를 통해 삽입을 한다. 모든 input에 대해 삽입을 완료하면 save_tree를 통해 bptree를 index file에 저장한다.
  d. deletion: (sys.argv[1] == ‘d’) bptree를 load_tree를 통해 index file에서 데이터를 가져오고 저장한다. delete csv file을 read mode로 열어 하나의 key를 deletion 함수를 통해 삭제한다. 모든 delete key 정보에 대해 삭제를 완료하면 save_tree를 통해 bptree를 index file에 저장한다.
  e. single key search: (sys.argv[1] == ‘s’) bptree를 load_tree를 통해 index file에서 데이터를 가져오고 저장한다. command line에서 받은 찾을 key 값을 single_key_search를 통해 찾는다.
  f. ranged_search: (sys.argv[1] == ‘r’) bptree를 load_tree를 통해 index file에서 데이터를 가져오고 저장한다. command line에서 받은 시작과 끝 범위에 해당하는 key들을 ranged_search를 통해 찾는다.
  g. 만약 잘못된 command line을 받았다면 에러 메시지를 띄운다.

```python
if __name__ == '__main__':
    if sys.argv[1] == '-c':
        with open(sys.argv[2], 'w') as file:
            file.write(f"{sys.argv[3]}\n")
    elif sys.argv[1] == '-i':
        bptree = BPlusTree()
        bptree.load_tree(sys.argv[2])
        with open(sys.argv[3], 'r') as csv_file:
            reader = csv.reader(csv_file)
            for row in reader:
                key, value = int(row[0]), int(row[1])
                bptree.insertion(key, value)
        bptree.save_tree(sys.argv[2])
    elif sys.argv[1] == '-s':
        bptree = BPlusTree()
        bptree.load_tree(sys.argv[2])
        bptree.single_key_search(sys.argv[3])
```

```python
    elif sys.argv[1] == '-r':
        bptree = BPlusTree()
        bptree.load_tree(sys.argv[2])
        bptree.ranged_search(sys.argv[3], sys.argv[4])
    elif sys.argv[1] == '-d':
        bptree = BPlusTree()
        bptree.load_tree(sys.argv[2])
        with open(sys.argv[3], 'r') as csv_file:
            reader=csv.reader(csv_file)
            for key in reader:
                bptree.deletion(int(key[0]))
        bptree.save_tree(sys.argv[2])
    else:
        print(f"error in command line")
```

Instructions for Compiling

Environment:

- macOS Monterey

I. 실행방법

1. 소스파일(bptree.py)가 존재하는 Source 폴더로 이동한다.

```
badahong@Badaui-MacBookAir B-tree_Assignment % cd Source
badahong@Badaui-MacBookAir Source % ls
bptree.py
badahong@Badaui-MacBookAir Source %
```

2. 필요 시 해당 폴더에 input.csv, delete.csv, 등 파일을 생성하고 내용을 넣어 저장한다.
3. 아래 command line를 이용해 실행한다.

Data File Creation:

insertion, deletion, single key search, range search를 하기 전 data file creation을 가장 먼저 실행해야만 한다.

- 실행 명령어: “python3 bptree.py -c [index_file] [b]”
- ex) python3 bptree.py -c index.dat 5

명령어:

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -c index.dat 5
```

결과:

Source > index.dat

```
1  5
2
```

Insertion:

- 실행 명령어: “python3 bptree.py -i [index_file] [data_file]”
- ex) python3 bptree.py -i index.dat input.csv

data file:

Source > input.csv

```
26,1290832
10,84382
87,984796
86,67945
20,57455
9,87632
68,97321
84,431142
37,2132
11,2345423
12,5436324
40,564353
41,63485
43,5435645
100,2345412
```

명령어:

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -i index.dat input.csv
```

결과:

Source > index.dat

```
5
0 4 11,2345423 26,1290832 40,564353 84,431142
1 2 9,87632 10,84382
1 3 11,2345423 12,5436324 20,57455
1 2 26,1290832 37,2132
1 4 40,564353 41,63485 43,5435645 68,97321
1 4 84,431142 86,67945 87,984796 100,2345412

```

Deletion:

- 실행 명령어: “python3 bptree.py -d [index_file] [data_file]”
- ex) python3 bptree.py -d index.dat delete.csv

data file:

Source > delete.csv

```
26
10
20
9
41
43
87
37
```

명령어:

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -d index.dat delete.csv
```

결과:

Source > index.dat

```
5
0 2 40,564353 84,431142
1 2 11,2345423 12,5436324
1 2 40,564353 68,97321
1 3 84,431142 86,67945 100,2345412

```

Single Key Search:

- 실행 명령어: “python3 bptree.py -s [index_file] [key]”
- ex) python3 bptree.py -s index.dat 100

명령어 & 결과:

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -s index.dat 43
40,84
NOT FOUND
```

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -s index.dat 100
40,84
2345412
```

Ranged Search:

- 실행 명령어: “python3 bptree.py -r [index_file] [start_key] [end_key]”
- ex) python3 bptree.py -r index.dat 5 100

명령어 & 결과:

```
badahong@Badaui-MacBookAir Source % python3 bptree.py -r index.dat 5 100
11,2345423
12,5436324
40,564353
68,97321
84,431142
86,67945
100,2345412
badahong@Badaui-MacBookAir Source %
```

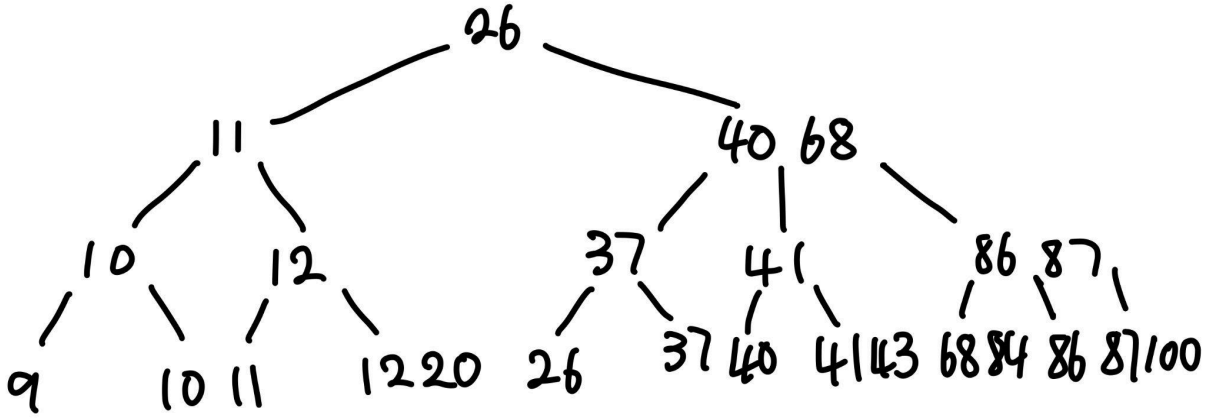
Other Specification of Implementation and Testing

I. **index file** 형식 예시

Source > index.dat

```
3
0 1 26,1290832
0 1 11,2345423
0 1 10,84382
1 1 9,87632
1 1 10,84382
0 1 12,5436324
1 1 11,2345423
1 2 12,5436324 20,57455
0 2 40,564353 68,97321
0 1 37,2132
1 1 26,1290832
1 1 37,2132
0 1 41,63485
1 1 40,564353
1 2 41,63485 43,5435645
0 2 86,67945 87,984796
1 2 68,97321 84,431142
1 1 86,67945
1 2 87,984796 100,2345412
```

→ 위 데이터는 아래 b+ tree를 나타낸다.



II. Testcase

아래와 같은 코드를 작성해 백만개의 데이터를 랜덤으로 생성 후 csv 파일에 저장하고, 해당 데이터들을 index file에 insertion으로 통해 삽입해보았다. 그 후, csv 파일에 있는 데이터 중 10000개를 deletion을 통해 삭제했다. 삭제된 값을 제외한 나머지 데이터들을 index file에서 single_key_search를 통해 찾았을 때, “NOT FOUND” 결과가 하나도 없었고, 모든 데이터들을 insertion과 deletion 후에도 찾을 수 있었다. 그리고, 1000~100000 사이의 key 값들을 ranged_search를 통해 찾았을 때, 모든 값들을 찾을 수 있음을 확인하였다.

Source copy > testcase2.py > ...

```python
import random
import csv

from bptree_d import BPlusTree

num_data = 1000000  # Number of data
key_min = 1          # Minimum key value
key_max = 100000000  # Maximum key value
output_file = 'random_data1.csv'  # Output file
delete_file = 'delete1.csv'
in_file = 'in_file.csv'

found_keys = []
no_searched_keys = []
searched_keys = []

keys = random.sample(range(key_min, key_max), num_data)

values = [random.randint(1, 100) for _ in range(num_data)]

random_keys = random.sample(keys, 10000)
random_keys_set = set(random_keys)

with open(output_file, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    for key, value in zip(keys, values):
        writer.writerow([key, value])

with open(delete_file, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    for key in random_keys:
        writer.writerow([key])
```

```python
bptree = BPlusTree()
bptree.load_tree('index.dat')

with open('random_data1.csv', 'r') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        key, value = int(row[0]), int(row[1])
        bptree.insertion(key, value)
bptree.save_tree('index.dat')


bptree = BPlusTree()
bptree.load_tree('index.dat')
with open('delete1.csv', 'r') as csv_file:
    reader=csv.reader(csv_file)
    for key in reader:
        result = bptree.deletion(int(key[0]))
        if result is not None:
            found_keys.append(key)
bptree.save_tree('index.dat')

temp = []
for key in keys:
    if key not in random_keys_set:
        temp.append(key)

bptree = BPlusTree()
bptree.load_tree('index.dat')
for key in temp:
    result = bptree.single_key_search(key)
    if result is not None:
        searched_keys.append(key)
    else:
        no_searched_keys.append(key)
```

```python
real = []

for key in searched_keys:
    if 1000 <= key <= 100000:
        real.append(key)

with open('not_found.dat', 'w') as no_searched_keys_file:
    for key in no_searched_keys:
        no_searched_keys_file.write(f"{key}\n")

with open('found.dat', 'w') as searched_keys_file:
    for key in real:
        searched_keys_file.write(f"{key}\n")

bptree = BPlusTree()
bptree.load_tree('index.dat')
result = bptree.ranged_search(1000, 100000)

```