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 [m/2]-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 [m/2] 이상인 경우이다.
- 왼쪽 형제부터 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 마찬가지다. 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 마찬가지다.
- 노드를 쪼갠 후에 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 succesor
    # 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, 오른족 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])

            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') # 맡애서 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
```

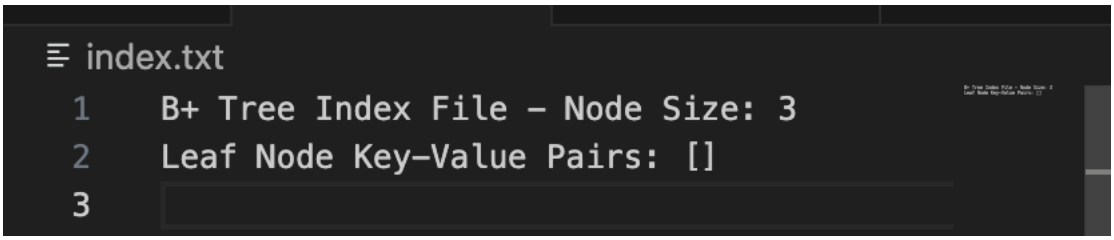
- 특정 key를 찾는 함수다.
- Key를 찾아가는 경로를 출력한다.

File I/O Functions

```python
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}")
```

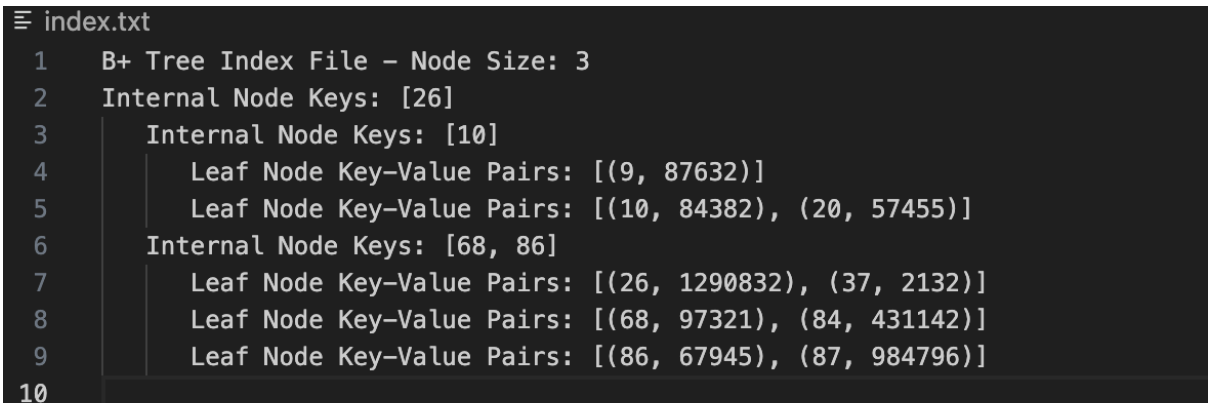
- Index_file을 생성하는 함수다.
- 입력받은 index_file에 "B+ Tree Index File – Node Size: order"를 write하고 그 결과를 저장한다.



- Create_index_file을 실행하고 나면 이렇게 저장된다.

```python
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.



- 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.

```python
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
```

- 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.

```python
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.

```python
  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
```

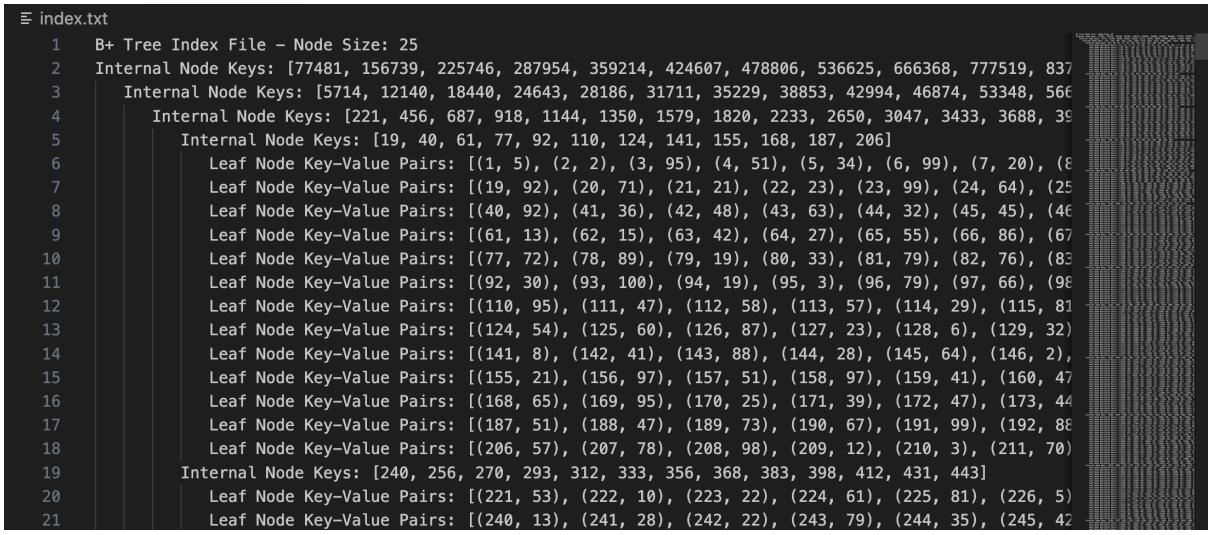
[Terminal Output]

```
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
```

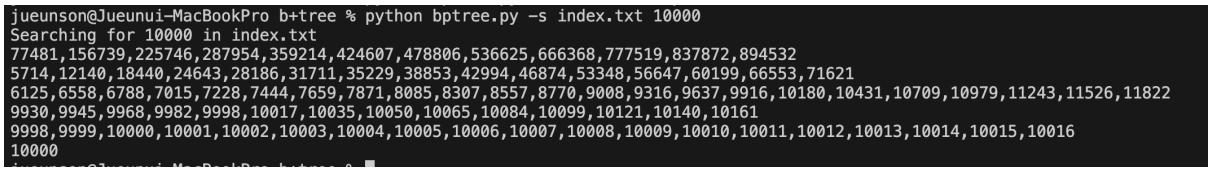
[Index File]

1,000,000개 Test Result

1) Insert

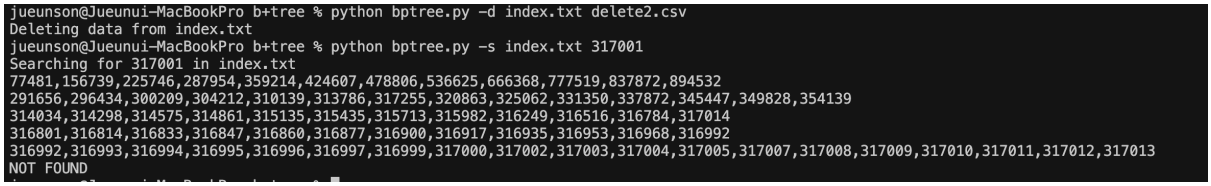


2) Single key search



3) Range Search

4) Delete



- Delete.csv에 317001 존재.

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 수정하니까 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.