B+Tree

2016059198 진승현

1. 프로그램 개요

실제 DB 인덱싱에 사용되는 검색 알고리즘인 B+Tree 이다.

<key, value> set을 입력 받아 key를 기준으로 B+Tree를 만들고 index file을 생성한다. (index file에는 fickle를 활용했다.)

실제 DB 인덱싱에서는 leaf-node에 value가 아닌 value의 주소가 들어가지만 본 프로그램에서는 value가 들어간 형태로 간략화되어 있다.

(※아직 Deletion이 구현되지 못했습니다. 추후 구현하여 재 업로드할 수 있다면 업로드 하도록 하겠습니다.)

2. B+Tree

B+Tree는 신규 노드 삽입 또는 삭제 시 모든 Leaf-node가 같은 level을 가지도록 자동으로 Balance를 맞춰주는 Tree구조이며 Database의 Index를 효율적으로 검색하는 데 사용되는 알고리즘이다. 기존 Index 방식의 경우 파일이 커질수록 성능이 하락해 주기적으로 재구성해주어야 하지만 B+Tree는 자동으로 Balance화 해주기 때문에 재구성의 필요성이 없다. 삽입/삭제는 비교적 비효율적인 구조로 되어있지만 Database의 경우 삽입/삭제 보다 검색의 비중이 월등히 크기 때문에 효율적이라 볼 수 있다.

최대 Node size가 N인 B+ Tree는 다음과 같은 특징을 만족한다.

- leaf까지 도달하는 모든 경로의 길이가 같다.
- root나 leaf가 아닌 각 노드들은 N/2 ~ N개의 자식 노드를 갖는다.
- 하나의 leaf node는 (N-1)/2 ~ n-1 사이의 value를 가진다.
- 만약 root가 leaf가 아니라면 최소 2개의 child를 가진다.
- 만약 root가 leaf라면 0 ~ N-1의 value를 가질 수 있다.

3. 코드 설명

1) Class Node

```
class Node():
    def __init__(self, keys=[], nodes=[], is_leaf = True, parent=True):
        self.is_leaf = is_leaf #initially leaf
        self.parent = parent  # 부모 노드 존재 유무 -> root 판별용
        self.keys = keys # keys for node(non-leaf and leaf)
        self.nodes = nodes # non-leaf node -> node pointers (항상 len(nodes) = len(leys)+1)
                           # #leaf-node -> value, (but nodes[-1]은 다음 leaf-node)
        # values for leaf-nodes are in BP_tree
    def insert_leaf(self, key, value):
        #print('cur_node', self.keys, self.nodes)

        if len(self.nodes) == 0:
            self.nodes.append(None)  # 맨 뒤에 임시 left-leaf, right-leaf node 추가
        # parent_key = cur_node[0] #부모의 키는 cur_node의 맨 앞키
        next_node = self.nodes.pop()  # 백업
        self.keys.append(key)
        self.nodes.append(value)
        self.keys, self.nodes = map(list, zip(*sorted(zip(self.keys, self.nodes))))  # 정렬
        self.nodes.append(next_node) # 복원
```

Parameter 설명

- is_leaf : Node가 leaf-node이면 True, 아니면 False
- parent : Node가 root-node이면 False, 아니면 True
- keys : Node가 보유한 key list
- nodes : non-leaf node일 경우 하위 노드를 가리키는 Node의 list로 구성, leaf-node일 경우 value의 리스트(단, leaf-node의 nodes의 마지막 element는 next leaf-node를 가리키는 Node)
- insert_leaf(self, key, value) : 호출한 Node에 key-value set을 추가하는 함수. 자동으로 정렬을 한다.

2) Class BP_tree

2-1) insert

```
def insert(self, cur_node, key, value):
    if not cur_node.is_leaf: #leaf노드가 아니라면
        for i in range(len(cur_node.keys)):
            if key < cur_node.keys[i]: #key가 작다면
                left, right = self.insert(cur_node.nodes[i], key, value) #재귀
                break
        if key >= cur_node.keys[-1]:#만약 모든 key와 비교해 크다면 우측 노드로
            left, right = self.insert(cur_node.nodes[len(cur_node.keys)], key, value)


        #한바퀴를 돌고서 나온 뒤
        if left != None: #만약 하위 노드에서 분할되어 올라온 노드가 있다면

            if right.is_leaf:  # 분할된 노드가 leaf라면
                insert_key = right.keys[0]  # 우측 노드 첫번째
            else:
                insert_key = right.keys.pop(0)  # 우측 노드 첫번째 삭제

            cur_node.keys.append(insert_key)  # 일단 집어넣기
            cur_node.keys.sort()  # 무조건 정렬하기
            changed_index = cur_node.keys.index(insert_key)  # 바뀐 키의 위치

            cur_node.nodes.pop(changed_index)
            cur_node.nodes.insert(changed_index, right)
            cur_node.nodes.insert(changed_index, left)


        #하위노드 처리까지 끝내고서
        if len(cur_node.keys) == self.degree:  # 만약 꽉 찼다면

            if cur_node.parent:  # 부모노드가 있다면 -> tree 높이는 변함 없음

                return self.spilt_node(cur_node)  # is_root = False
            else: # 없다면 = root -> root의 분할은 tree 높이가 한단계 올라감 -> non-leaf
                left, right = self.spilt_node(cur_node)  # is_root = True
                new_node = Node([], [], False, False)  # root이면서 leaf도 아님
                insert_key = right.keys.pop(0)  # 우측 노드 첫번째
                new_node.keys = [insert_key]  # key는 우측 첫번째
                new_node.nodes = [left, right]
                self.root = new_node  # root 갱신

    else: #leaf node라면
        cur_node.insert_leaf(key, value) # 키 입력

        if len(cur_node.keys) == self.degree: #꽉 찼다면

            if cur_node.parent: # 부모노드가 있다면 -> tree 높이는 변함 없음
                return self.spilt_node(cur_node) #is_root = False

            else: #없다면 = root -> root의 분할은 tree 높이가 한단계 올라감

                left, right= self.spilt_node(cur_node) #is_root = True
                new_node = Node([], [], False, False)  # root이면서 leaf도 아님
                insert_key = right.keys[0]  # 우측 노드 첫번째
                new_node.keys = [insert_key]  # key는 우측 첫번째
                new_node.nodes = [left, right]
                self.root = new_node #root 갱신

    return None, None #아무런 변동이 없을 때
```

재귀적으로 leaf-node를 찾아 들어가 값을 insert하고 return으로 나올 때 일괄적으로 split 처리를 해주었다. split함수로 반환된 2개의 sub node를 root이면 새로운 node를 만들어 처리했고 non-root이면 기존 node의 key와 nodes의 넣어줄 index를 찾아 각각 넣어줬다.

2-2) split

```
def spilt_node(self, node): #spilt root node
    m = len(node.keys)
    mid = m // 2 - 1 if m % 2 == 0 else m // 2
    if node.is_leaf:  # leaf node라면
        right = Node(node.keys[mid:], node.nodes[mid:])  # 우측 분할
        node.parent = True  # node는 left
        node.keys = node.keys[:mid]
        node.nodes = node.nodes[:mid] + [right]
        node.is_leaf = True
        return node, right #leaf-node root

    else:  # non-leaf라면
        right = Node(node.keys[mid:], node.nodes[mid + 1:], False)  # 우측 분할
        left = Node(node.keys[:mid], node.nodes[:mid + 1], False)  # 좌측 분할
        return left, right #non-leaf node root
```

leaf-node라면 연결리스트의 순서를 유지하기 위해 새로운 right node와 함께 기존 입력 받은 node를 변환시켜 left node로 변환시켜 반환하였고, non-leaf node라면 각각 새로운 left node와 right node를 만들어 반환했다.

여기서 non-leaf의 경우 원래는 key를 [mid:] 와 [:mid+1]으로 분할해서 [mid]를 상위 노드로 전달해야 하나 편의상 [mid:]와 [:mid]로 분할해 넘겨 insert에서 [mid]를 분할하는 것으로 처리했다.

2-3) deletion (미구현)

2-4) find_leaf

```
#single_search는 get_path = True
def find_leaf(self, key, print_path = False):
    cur_node = self.root
    while not cur_node.is_leaf:
        if print_path: #print_path == True라면 경로 출력
            print(",".join([str(key) for key in cur_node.keys]))
        for i in range(len(cur_node.keys)):
            if cur_node.keys[i] > key: #cur_node가 key보다 크다면 (같은 키는 없으므로)
                cur_node = cur_node.nodes[i] #해당 좌측 하위 노드로
                break
            elif i == len(cur_node.keys) - 1: #cur_node보다 key가 크거나 같은데 마지막 노드이면
                cur_node = cur_node.nodes[i+1]#우측 하위 노드로
                break

    return cur_node
```

single_key search와 ranged_search에 모두 쓰일 수 있도록 print_path함수를 두어 True일 때만 경로를 표시할 수 있도록 했다.

2-5) single_key_search

```
#key까지 가는 path 출력
def single_key_search(self, key):

    leaf = self.find_leaf(key, print_path=True)

    if key not in leaf.keys:
        print("NOT FOUND!")
        return
    else:
        value = leaf.nodes[leaf.keys.index(key)] #list에서 가져옴
        print(value) #value 출력
```

single key 출력

2-6) ranged_search

```
#start부터 end까지 출력
def ranged_search(self, start, end):
    result = self.find_leaf(start)

    while 1:

        for i in range(len(result.keys)):
            #print(result.keys[i])
            if start <= result.keys[i] <= end: #key가 start와 end 사이에 있다면
                print(result.keys[i], result.nodes[i], sep=',') #출력
            elif result.keys[i] > end: # 더 커지면
                break

        if result.nodes[-1] == None: #다음 리프노드가 없으면
            break

        elif result.keys[-1] <= end:  #마지막 키가 end보다 작거나 같다면
            result = result.nodes[-1] #next leaf node로

        else: #마지막 키가 end보다 크다면
            break
```

ranged search 출력. leaf-node의 연결성을 이용해 선형적으로 검색하여 빠르게 검색할 수 있다.

4. 실행 예제 (※자체적으로 random한 key-value set csv를 출력하는 python 프로그램을 만들어 테스트)

1)Argument Format

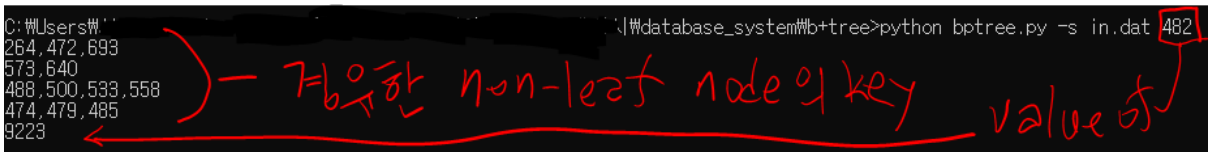
1-1) create index

```
python bptree.py -c in.dat 5
```

1-2) insertion

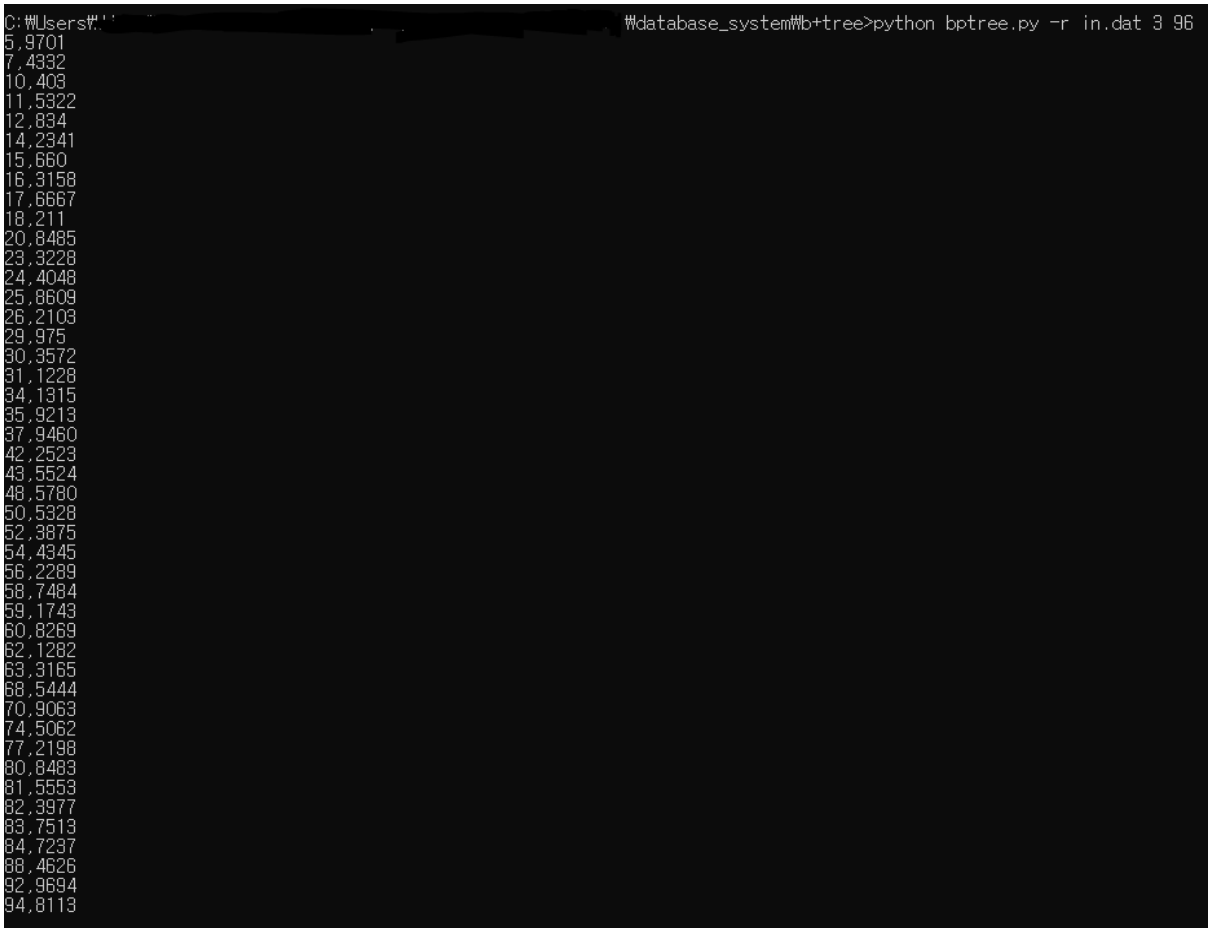
```
python bptree.py -i in.dat input_data.csv
```

1-3) single key search



경유한 non-leaf node의 key를 한 줄 씩 출력하고 마지막 줄에는 명령줄인수로 입력한 key에 해당하는 value 값을 출력

1-4) ranged search



지정된 범위 사이에 있는 key와 value set을 출력

5. Specification of Testing

OS : Windows 10 Home (x64)

Language version : Python 3.7.0