B+Tree Description & Instructions for Compiling

경제금융학부 2019076880 안수현

1. Summary of my algorithm

0) constructors

-class Node

```
class Node:
    def __init__(self, order):
        self.order = order # 자식 노드 최대 개수
        self.m = 0 # 현재 키 개수
        self.entries = [] # nonleaf entry 또는 leaf entry가 담기는 리스트
        self.right_node: Node = None # nonleaf 노드의 경우 가장 오른쪽 자식 노드, leaf노드의 경우 오른쪽 형제 노드
        self.prev_sibling: Node = None # leaf 노드의 왼쪽 형제 노드
        self.parent: Node = None # 부모 노드
        self.is_leaf: bool = True # 현재 노드의 leaf 여부
        self.p_idx = None # 현재 노드가 부모 노드의 몇 번째 인덱스에 위치하는지 저장
```

▶Node 클래스의 생성자, entries 리스트에 leaf_Entry 또는 nonleaf_Entry 클래스의 객체를 담음

-class Bptree

```
class Bptree:
    def __init__(self, order):
        self.root = Node(order)
```

▶Bptree 클래스의 생성자, 주어진 order에 맞게 루트 노드를 초기화함

1) index file

'tree_info' 라는 클래스로 트리의 정보를 저장하도록 한 다음, pickle 모듈을 활용하여 트리 정보를 담은 tree_info 클래스 객체를 index file에 바이너리 형태로 저장하는 방법으로 진행하였음.

```
## save information of the tree
class tree_info:
    def __init__(self, order):
        self.order = order
        self.l = []

    def make_list(self, first: 'Node'):
        self.l = [] # reset the list
        n = first
        while n != None: # save all entries in ascending order
            self.l += n.entries
            n = n.right_node
```

▶리프 노드의 모든 entries 리스트를 이어 붙여 하나의 리스트로 저장

```
## index file creation
if op == '-c':
    order = sys.argv[3] # save order
    Tree = tree_info(int(order))
    with open(filename, "w+b") as index_file: # 인덱스 파일 쓰기 모드로 생성
        pickle.dump(Tree, index_file) # 직렬화해서 index file에 저장
```

▶index file create 명령이 들어오면 새로운 tree_info 클래스 객체를 생성한 후 이 객체를 index file에 저장

insert, delete, search, ranged search 명령이 들어오면 index file에서 tree_info객체를 불러온 후 객체 안에 저장된 리스트로 트리 생성

```
# 저장된 정보로 현재까지의 비쁠트리 만들기
# 절반 나눠서 insert -> 트리 레벨 줄이기
bptree = Bptree(Tree.order)
for i in range(int(len(Tree.l) / 2)):
    bptree.insert(Tree.l[i].key, Tree.l[i].pointer)
for i in range(len(Tree.l) - 1, int(len(Tree.l) / 2) - 1, -1):
    bptree.insert(Tree.l[i].key, Tree.l[i].pointer)
```

▶이때 오름차순으로 정렬된 entry를 절반으로 나누어 뒤의 절반은 역순으로 insert하여서 트리의 level이 최대한 낮게 생성되도록 최적화함.

insert, delete 이후에는 갱신된 트리 정보를 다시 저장해서 index file을 업데이트 해 주었음.

2) insertion

insert(self, key, value)

키 값의 위치로 적절한 leaf node를 찾은 후 add 하고, 노드가 overflow 된 경우 split하는 방식으로 진행하였음. 중복된 키 값의 경우 에러 메시지를 띄운 후 insert 함수를 종료함.

■ 주요 함수

- split_leaf_node(self, root) -> 'Node'

- 오버플로우된 리프 노드를 둘로 쪼개는 함수
- 루트 노드 값을 반환함
- 왼쪽 노드는 새로운 노드를 생성하고 오른쪽 노드는 기존의 노드를 활용함. order가 홀수인 경우 오른쪽 노드에 더 많은 키가 할당됨
- 루트 노드이자 리프 노드인 노드를 split하는 경우와 일반적인 리프 노드를 split하는 두 가지 경우가 있음
- 루트 노드 split의 경우 새로운 루트 노드를 생성

- 일반적인 리프 노드 split의 경우 부모 노드의 오버플로우 여부를 검사하고, 만일 부모 노드가 오버플로우 되었다면 split_nonleaf_node로 부모 노드 split 진행함

-split_nonleaf_node

- nonleaf 노드를 split하는 함수
- leaf 노드와 달리 중앙의 키 값을 오른쪽 자식 노드에 남겨두지 않음
- 현재 루트 노드인 경우 새로운 루트 노드를 생성해서 중앙 키 값을 올림
- 현재 루트 노드가 아닌 경우 중앙 키 값을 받은 부모 노드의 오버플로우 여부를 확인해서 오버플로우 되었다면 다시 split_nonleaf_node함수 호출

-add(self, entry, idx) –> 'bool'

- key 기준으로 오름차순 정렬해서 entry를 노드의 entries 리스트에 넣어주는 함수
- 중복 키가 있는 경우 False값 반환
- 이 함수는 insert 뿐 아니라 delete과정에서도 사용됨
- idx 매개 변수로 entry가 들어갈 인덱스를 미리 아는 경우 탐색 과정 생략하고 바로 삽입하도록 해서 최적화함. (delete 함수에서 호출될 때 인덱스를 미리 아는 경우가 많음)

3) deletion

delete(self, key)

매개변수로 받은 키 값을 삭제하는 함수. 찾는 키 값이 트리에 없는 경우 에러 메시지를 띄우고 함수 종료함

삭제 이후 해당되는 케이스에 따라 트리 규칙을 맞춤

case 1: 루트 하나만 있는 트리이거나 삭제 이후에도 최소 키 개수 조건을 만족하는 경우

키 단순 삭제 -> 인덱스 재조정

>삭제 이후 최소 키 개수 조건을 만족하지 않는 경우

case 2: 형제 노드에서 키를 빌려올 수 있는 경우

키 삭제 -> 오른쪽 형제 노드와 왼쪽 형제 노드 중 키를 빌려 와도 최소 키 개수 조건을 만족하는 노드에서 키 빌려옴 -> 인덱스 재조정

case 3: 두개의 형제 노드 모두 키 개수가 최소인 경우

키 삭제 -> 형제 노드와 내려온 부모 키를 모아서 병합 -> 인덱스 재조정 -> 키를 빌려 온 부모

노드의 키 개수가 최소 키 개수보다 작아진 경우 nonleaf_merge 함수 호출

■ 주요 함수

-merge(self, p_idx, is_leftmost) -> 'Node'

- 리프 노드를 병합하는 함수
- p_idx는 병합하는 노드로 가져올 부모 인덱스, 부모 노드의 맨 왼쪽 자식 노드인 경우 is_leftmost 값이 True
- 부모 노드의 왼쪽 끝 리프 노드가 아닌 경우 왼쪽 형제 노드에 병합
- 부모 노드의 왼쪽 끝 리프 노드일 경우 오른쪽 형제 노드에 병합

-nonleaf_merge(self, root: 'Node')

- nonleaf 노드를 병합하는 함수
- 부모 노드의 맨 왼쪽 자식 노드가 아닌 경우 왼쪽으로 병합
- 부모 노드의 맨 왼쪽 자식 노드인 경우 오른쪽으로 병합
- 부모 노드에서 키 값을 가져오고 부모 노드가 비게 되는 경우 이후 이 노드를 병합할 때 필요한 부모 노드 인덱스를 미리 계산해 두어야 함
- 병합 이후 부모 노드의 키 개수가 최소 키 개수보다 작은 경우 다시 nonleaf_merge 함수 호출해서 병합
- 병합한 노드가 오버플로우 된 경우 split

-_restruct_index(self, now: 'Node')

- 키 값이 삭제된 이후 nonleaf 노드의 인덱스 키 값을 적절한 값으로 재조정해 주어야 함.
- 해당 키 값 오른쪽 자식 노드에서 왼쪽 자식 노드로 쭉 내려간 다음 만나는 리프 노드 첫번째 키 값을 가져와서 nonleaf 노드의 키 값으로 바꿈

4) search

주어진 키 값을 찾는 함수, bplustree의 규칙에 따라 재귀적으로 탐색함

5) ranged search

주어진 범위에 해당하는 키 값을 찾는 함수, search와 동일하게 bplustree의 규칙에 따라 start key를 찾고 end key까지 출력함

찾는 범위가 잘못되거나 범위 안에 해당하는 키 값이 없는 경우 아무것도 출력하지 않고 종료됨

2. Instruction for Compiling

과제는 윈도우 환경에서 구현하였고, powershell을 이용하여 테스트했습니다.

같은 디렉토리에 data file이 있는 상태에서, 과제 명세에서 요구한대로 command line으로 실행시키면 프로그램이 정상 작동합니다.

<실행 예시>

python bptree.py -c index.dat 8

python bptree.py -i index.dat data_file

python bptree.py -d index.dat data_file

python bptree.py -s index.dat 1

python bptree.py -r index.dat 1 100

```
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -c index.dat 16
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -i index.dat thousand.csv
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -s index.dat 26
65,129,193,257,321,385,449,569,641,713,785,857,929
9,17,25,33,41,49,57
260
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -d index.dat originaldelete.csv
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -s index.dat 26
69,133,197,261,325,389,453,569,641,713,785,857,929
11,19,29,37,45,53,61
NOT FOUND
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -d index.dat thousand_del.csv
Deletion Error (key : 26) : no such key in the tree!
Deletion Error (key : 20) : no such key in the tree!
Deletion Error (key : 10) : no such key in the tree!
Deletion Error (key : 9) : no such key in the tree!
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -s index.dat 9
NOT FOUND
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -r index.dat 1 1000
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -i index.dat thousand.csv
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree> python bptree.py -r index.dat 20 30
20,200
21,210
22,220
23,230
24,240
25,250
26,260
27,270
28,280
29,290
30,300
PS C:\Users\jtu11\Desktop\3-2\데베시\lab1_B+tree>
```

▶1~1000의 key, value 데이터를 insert, search, delete하는 예시 (originaldelete.csv에는 26, 10, 20, 9가 들어있고, thousand_del.csv에는 1~1000의 내림차순으로 정렬된 키 값이 들어 있습니다.)

3. Other specification

100만개 데이터를 insert하는 경우, order 값 설정에 따라 수행 시간에 차이가 있습니다. 8~16으로

order를 설정했을 때 가장 성능이 좋았는데, 평균 33초 정도의 시간에 insert를 수행합니다. order가 더 커질수록 수행 시간이 커지는데, 이는 노드 내에서 키 값을 정렬하는 시간이 오래 걸리기 때문입니다. 다만 order가 3~4인 경우 split 과정이 자주 일어나기 때문에 order가 작음에도 1분 정도의 시간이 필요합니다.

delete는 삭제 이후 노드를 병합하고 트리 전체를 순회하며 인덱스를 재조정하는 과정 등 insert보다 복잡한 과정을 거쳐 트리를 재구성하기 때문에 전체적으로 insert보다 많은 시간이 필요했습니다. 100만개의 데이터가 들어있는 트리에서 7개의 정렬되지 않은 데이터를 지우는 데에 19초 정도 소요됩니다.

과제에서 명시된 에러의 경우 해당 메시지를 출력하도록 처리했고, command line 실행에서 argument 개수에 오류가 있거나 옵션에 오류가 있는 경우 usage를 통해 사용 방법을 안내하도록 구현했습니다. 이 외에 다른 오류 (ex, input파일이 존재하지 않음, open 또는 read 오류)의 경우 파이썬의 오류 메시지가 그대로 출력되도록 두었습니다.