2020 B+ tree implementation report

2016024893 오성준

1. Summary of algorithm

제가 짠 알고리즘은 기본적으로 b+ 트리의 기본적인 알고리즘을 따랐습니다. 순서대로 요약해서 설명해보겠습니다.

1)insert

- 우선 해당 키가 들어갈 노드를 확인하고 해당 키가 있으면 insert함수를 종료합니다.

- 해당 키가 없다면, 해당하는 키와 데이터를 저장합니다.

- overflow가 일어났는지 확인하고 일어났다면 split 함수를 이용해서 현재 코드를 분할하고 mid라는 지점의 값을 부모 노드의 키에 삽입합니다. 그리고 좌우에 split으로 나눈 child들을 삽입합니다.

- 키가 삽입된 노드가 overflow가 일어나지 않을 때까지 반복합니다.

2)search

1] single search

- 해당하는 key가 있어야하는 노드를 확인하고 해당 키가 있으면 key를 표시합니다.

2] range search

- begin값에 해당하는 노드로 가서, 그 노드에 있는 키들부터 순서대로 key와 value를 print합니다. right_Child로 오른쪽 노드가 뭔지 알 수 있기 때문에 간편하게 print 할 수 있습니다.

3)delete

- 우선 해당 키가 존재할 노드를 확인하고 해당 키가 없다면 delete함수를 종료합니다.

- 해당 키가 있다면, 해당 키를 지웁니다. 키가 지워졌을 때는 다음과 같은 case로 나뉩니다. (leaf node)

1] 해당 노드에서 underflow가 나지 않을 때 => 그대로 놔둡니다.

2] 해당 노드에서 underflow가 날 때

* 만약 sibling들 중에 underflow_predict를 통해서 한 개를 주고도 underflow가 나지 않

는 sibling이 있다면, 그 노드에서 key값을 가져옵니다.

* sibling들 중에서 키를 줄 수 있는 게 없다면, 왼쪽이나 오른쪽 중에서 하나를 골라서 merge를 하고 split key를 삭제합니다.

- 만약에 merge를 통해서 index node의 key가 삭제된다면, 이는 leaf node에서와 비슷하지만 다른 과정을 겪습니다. (index node)

1] 해당 노드에서 underflow가 나지 않을 때 => 그대로 놔둡니다.

2] 해당 노드에서 underflow가 날 때

* 만약 sibling들 중에 underflow_predict를 통해서 한 개를 주고도 underflow가 나지 않는 sibling이 있다면, 그 노드에서 key값을 split key로 올리고, 원래 있던 split key를 하나가 삭제된 index node에 집어넣습니다.

* sibling들 중에서 키를 줄 수 있는 게 없다면, 왼쪽이나 오른쪽 중에서 하나를 골라서 merge를 하고 split key를 삭제합니다. 이 때, split key가 삭제되기 때문에, parent node 또한 위와 같은 과정을 재귀적으로 밟습니다. 계속해서 반복합니다.

2. Detailed description of code(for each function)

* 코드가 767줄인 관계로 코드를 한줄 한줄 설명할 수는 없는 적 양해 부탁바랍니다. 대신에, 각각의 함수들의 역할이 무엇이고, 어느 상황에 쓰는지, 알고리즘은 무엇인지 설명하겠습니다.

1) max_degree=0

- 전역 변수로 설정하여서, 자유롭게 사용할 수 있게 했습니다. 제 생각에는 bptree에 집어넣는 것보다 밖에 놔두는 것이 한줄이라도 짧게 쓰는 길이라고 생각했습니다.

2) node의 성분

```
def __init__(self):
    self.is_leaf=False
    self.key=[]
    self.value=[] #leaf node일때 값들을 가진다
    self.child=[] #index node일때 data를 가진다
    self.parent=None
    self.right_child=None
    self.rightmost_child=None
    self.node_num=-1
```

Is_leaf: True면 leaf node입니다

Key, value, child, parent, rightmost_child: 각각 이름 그대로를 의미합니다. Is_leaf에 따라서 값이 존재하기도 안하기도 합니다.

Right_child: leaf node에서 오른쪽 노드를 저장한 것입니다. 다만 index file을 만드는 편의상 index node에도 오른쪽 노드가 존재한다면 저장되어 있습니다.

Node_num: 이 것 또한, index file을 만들 때 편의를 위해서 만든 변수입니다.

3) node class method

```
def split(self): ...
def set_key_node(self, key, left, right): ...
def set_key_value(self, key, val): ...
def underflow_predict(self): ...
def underflow(self): ...
def delete_key_value(self, place): ...
def delete_key(self, place): ...
def delete_child(self, place): ...
def set_rightmost(self): ...
```

- split함수는 노드를 반으로 나누는 함수입니다. Overflow인 노드를 self와 right로 나누어서 return 합니다. 이 때, left라는 새로운 노드를 만들지 않는 이유는 self를 right_child로 가지는 노드가 있기 때문에, self를 직접 바꿔서 그 상태를 유지하려 했습니다.

-set_key_node함수는 split을 하고나서 left와 right로 나눠진 2개의 노드를 child로 가지고 그 split key 또한 제대로 된 위치에 집어넣는 함수입니다. insert함수에서 index node에 주로 사용합니다.

-set_key_value함수는 set_key_node함수와 비슷하지만, 노드 대신에 value값을 받아서 제대로 된 자리에 집어넣습니다. insert함수에서 leaf node에 주로 사용합니다.

-underflow predict 함수는 키 한 개를 다른 노드에 주면 underflow가 나는지 예측하는 함수이고 underflow함수는 underflow가 지금 상태에서 났는지 확인하는 함수입니다. 중복해서 여러 번 쓰기 싫어서 함수로 만들었습니다. 또한 이렇게해서 코드를 읽을 때, 의미가 명확히 보이도록 했습니다. delete에서 사용됩니다.

-delete_key_value는 leaf 노드에서 place의 위치에 있는 key와 value를 동시에 지워버리는 함수입니다. 위치를 사용해서 함수에서 찾지 않고 또한, key[place:]와 같은 간편한 tool들을 사용해서 함수를 짤 수 있도록 했습니다. Delete_key와 delete_child도 각각 key와 child를 지우는 함수입니다. 이 때 두개의 함수를 나눈 것은 index node에서 child는 key보다 항상 한 개가 더 많기 때문에 다를 것이라 판단해서 함수를 나누었습니다. Delete_key_value는 leaf node에서 delete_key와

delete_child는 index node에서 사용합니다.

- set_rightmost는 index node에서 가장 오른쪽에 있는 child를 node에 있는 rightmost_child에 집어넣어 줍니다.

4) bptree의 성분

```
def __init__(self):
    self.root=node()
    self.root.is_leaf=True
```

- 성분은 root가 존재하고 root는 최초에는 leaf node이기 때문에 is_leaf를 true로 설정해둡니다.

5) bptree class method

```
def rightmost(self, node): ...
def traversal(self, node): ...
def node_num_ch(self, node): ...
def parent_ch(self,node): ...
def search_node(self, key): ...
def insert(self, key, value): ...
def read_index_file(self,file_name): ...
def write_index_file(self,file_name): ...
def make_node_list(self): ...
def find_key(self, key): ...
def find_range(self, left, right): ...
def find_and_change_key(self, start_node, find_key, change_key): ...
def delete(self,key): ...
def change_index(self, current_node, key): ...
```

- rightmost함수는 dfs를 돌면서 각 node의 rightmost_child값을 설정해줍니다. 저는 그냥 할 때마다, len함수를 이용해서 사용했기 때문에 큰 도움이 되지는 않았습니다.

- traversal 함수는 디버깅을 위해서 만든 함수로, dfs로 돌면서 부모와 child들을 print합니다.

- node_num_ch 함수는 디버깅을 위해서 만든 함수로, 각 노드의 node_num을 print해줍니다.

- parent_ch 함수는 디버깅을 위해서 만든 함수로, dfs로 돌면서 해당 child 노드의 parent에 뭐가 저장되어 있는지 알 수 있습니다.

- search_node함수는 입력받은 key값에 해당하는 leaf node를 찾아주는 함수입니다. B+ tree의 삭제와 삽입은 항상 leaf node에서 일어나기 때문에 굉장히 중요한 함수입니다. 저는 self.root에서부터 값을 비교해가면서 집어넣었습니다.

- read_index_file과 write_index_file, make_node_list는 모두 index file을 만드는데 사용되는 함수들입니다. 우선 index file의 format같은 경우는 4번에서 언급할 예정이고, 그와 상관없이 제가 index file에서 어떻게 값을 읽고 쓰는지 설명하겠습니다.

1] 파일에 트리를 쓸 때

=> 저는 root부터 순서대로 차곡차곡 node_list_w라는 배열에다가 node들을 append했습니다. 다만 그 순서는 bfs의 순서를 가지도록 했습니다. bfs의 순서를 가지도록 하는 방법은 제일 왼쪽 노노드들부터 시작해서 그 노드의 right_child로 계속 오른쪽으로 가면서 저장하는 방법을 썼습니다. 그 방법이 있는 함수가 make_node_list입니다. 그리고 for문을 돌면서 node_list_w에 있는 노드들의 node_num을 저장했습니다.

```
for i in range(len(node_list_w)):
    node_list_w[i].node_num=i
```

i번째 노드면 node_num은 i가 됩니다.

그러고 난 후에, write_index_file 함수를 사용해서 4번에 적어놓은 형태처럼 저장합니다. 이 때, child나 parent처럼 node들을 쓰는 대신에 해당하는 노드의 node_num을 쓰게 했습니다.

2] 파일에서 트리를 불러올 때

=> 첫번째 줄은 max_degree를 가르치고 그 이후로 7줄씩 하나의 노드에 관한 정보를 담고 있습니다. 이 때, 우선 노드들을 전부 받아놓고, child나 parent, right_chlid처럼 노드와 관련된 정보들은 따로 배열을 만들어서 해당하는 저장해두고, node들을 node_list_r이라는 배열에 담아둡니다. 그런 이후에 미리 저장해둔 노드들의 정보대로, 해당 노드에 필요한 정보를 저장하면 됩니다.

```
for i in range(len(node_list_r)):
    if node_child_num[i][0]!=-1:
        for j in range(len(node_child_num[i])):
            turn=node_child_num[i][j]
            node_list_r[i].child.append(node_list_r[turn])

    if node_right_num[i]!=-1:
        turn=node_right_num[i]
        node_list_r[i].right_child=node_list_r[turn]

if len(node_list_r)>0: self.root=node_list_r[0]
```

모든 정보가 입력된 후, node_list_r을 돌면서 해당하는 node_num을 append형식을 통해서 받습니다. Node_num이 배열에서의 위치와 같을 것이기 때문에 쉽고 간편하게 data를 받을 수 있습니다.

- 그 이후에 insert함수와 delete함수, find_key함수, find_range함수는 각각 summary에서 말한

insert, delete, single search, range search의 알고리즘을 따르고 있습니다. 다만 중요한 내용은 insert와 delete함수는 leaf node와 index node에서의 코드를 다르게 구현했습니다. 왜냐하면 leaf node는 value가 index node에서는 child를 사용하기 때문에 차이가 있기 때문입니다. 이 차이는 제가 처음부터 전체를 재귀로 구현하지 못하게 했습니다. 향후에 전체를 재귀로 구현하는 걸 생각해보고 싶기도 합니다.

- find_and_change_key 함수는 leaf node에서 지워진 key가 index node에 있을 수 있기 때문에 이 값을 대체된 값으로 바꿔주는 역할을 합니다. node부터 parent를 통해서 타고 올라가면서 찾습니다. Key가 중복되지 않기 때문에 하나를 찾으면 함수가 끝나도록 만들었습니다. insert보다는 delete에서 많이 사용됐습니다. 특히나, 제일 왼쪽에 있는 value가 지워지면 한단계든 두단계든 위에서 값이 변경되야 하는 상황이 나오기 때문에 적당한 값을 넣으려 노력했습니다.

- change_index 함수는 delete과정에서 merge가 일어날 때, index node들에서 재귀적으로 상황을 판단해야 하기 때문에, 그 부분만 따로 떼서 함수로 만들었습니다. 이렇게 한 이유는 delete함수에서 이미 leaf node의 삭제와 borrow그리고 merge에서 너무나도 코드가 복잡해졌기 때문에, index 부분을 따로 떼서 적었습니다. 그리고 함수로 표현해서 재귀함수의 기능을 가지게 했습니다. delete같은 경우는 index node에서 merge를 할 때 빼고는 재귀적으로 사용할 필요가 없다고 생각해서 그 부분에만 집어넣었습니다. 사실 delete함수와 굉장히 유사합니다.

6) 외부 함수

```
def merge_leaf(left, right):...

def merge_index(left, right, split_key):...
```

- 각각의 함수들은 leaf node와 index node의 merge를 실행하는 함수들입니다. 이 것들을 밖에다 적은 이유는 이 함수들은 self가 필요 없다고 판단했기 때문입니다. 이 함수들도 split함수처럼 left 함수로 직접 값을 바꿔서 left를 right_child로 가지는 노드가 제대로 유지되도록 코드를 짰습니다.

7) main

```
def main(argv):
    global max_degree
    #argv[1]-> 명령의 종류가 저장됨
    if argv[1]=="-c": ...

    if argv[1]=="-s":#그냥 search ...

    if argv[1]=="-r":#범위 search ...

    if argv[1]=="-i": ...

    if argv[1]=="-d": ...
```

- main 함수로 각각의 상황에 맞게 작동하도록 했습니다. 명령이 실행되면, Tree를 선언하고 Tree의 method인 read_index_file 함수를 통해서 index file에 저장된 b+ tree의 정보를 저장하도록 했습니다. 그리고 -I 와 -d에서는 위에서 언급한 make_node_list함수와 write_index_file함수로 index file에 변경된 tree의 meta data를 저장했습니다. Rightmost도 또한 마지막에 배치했습니다. 제 생각에는 딱히 쓸모가 없어 보여서 마지막에 값만 가지도록 설정했습니다.

```
Tree=bptree()                          Tree.rightmost(Tree.root)
Tree.read_index_file(argv[2])          Tree.make_node_list()
                                       Tree.write_index_file(argv[2])
```

3. Instructions for compiling source codes at TA's computer

- sublime text로 작성했습니다.

-window cmd창에서 다음과 같이 실행할 수 있습니다.

```
C:₩Users₩joon2₩Desktop>py a.py -c data.dat 3

C:₩Users₩joon2₩Desktop>py a.py -i data.dat input.csv

C:₩Users₩joon2₩Desktop>py a.py -s data.dat 26
26
68,86
1290832

C:₩Users₩joon2₩Desktop>py a.py -r data.dat 10 70
10,84382
20,57455
26,1290832
37,2132
68,97321

C:₩Users₩joon2₩Desktop>py a.py -d data.dat delete.csv

C:₩Users₩joon2₩Desktop>py a.py -r data.dat 10 70
37,2132
68,97321
```

4. Any other specification of implementation and testing

- index file은 다음과 같이 구성됩니다.

1번째 줄: max_degree 값을 나타냅니다.

그 이후 7줄씩 다음과 같은 정보를 나타냅니다.

1번째줄->node_num

2번째줄->is_leaf

3번째줄->key

4번째줄->value

5번째줄->child들의 node_num

6번째줄->parent의 node_num

7번째줄->right_child의 node_num

이 때, 만약에 해당하는 data가 없다면 -1을 쓰게 만들었습니다. 제가 -1을 쓰는 이유는 아직 key나 value값이 음수인 예시를 본적이 없기 때문입니다. 네 제가 가지고 있는 교재들이나, 인터넷상에서 찾지 못했습니다. 다만 제가 index file을 구현한 방식은 확실히 key나 value가 -1값이 들어온다면 문제점이 발생할 수 있는 코드이긴 합니다. 하지만, 그런 예시를 본적이 없기 때문에 괜찮을 거라고 판단했습니다.