**코드 명세서**

**B-TREE**

컴퓨터소프트웨어학부

2019040591

박진수

**개발 및 실행 환경**

OS: Windows\_NT x64 10.0.19045 언어 버전: Python 3.10

IDE 종류: Pycharm 2024.1 (Professional)

**코드 설명**

class BTreeNode:  
 def *\_\_init\_\_*(*self*, *key\_value\_list*, *children*, *is\_leaf*):  
 *"""  
 Initialize a BTreeNode.* ***:param*** *key\_value\_list: A list of tuples(key, value).* ***:param*** *children: A list of children(BTreeNode).* ***:param*** *is\_leaf: A boolean value representing whether it is a leaf node.  
 """  
 self*.key\_value\_list = *key\_value\_list  
 self*.children = *children  
 self*.is\_leaf = *is\_leaf*

B-tree의 각 노드를 구성하는 BTreeNode 클래스입니다. 해당 클래스의 인스턴스 변수로는 (key, value) pair를 담는 리스트인 key\_value\_list, 자식 노드 객체들을 담는 children, 그리고 해당 노드가 리프노드인지 여부(Boolean)을 담는 is\_leaf 가 있습니다.

class BTree:  
 def *\_\_init\_\_*(*self*, *t*):  
 *"""  
 create an empty root node.* ***:param*** *t: The minimum degree of the B-tree. Every node other than the root must have at least t-1 keys.  
 Every internal node other than the root thus has at least t children.  
 Every node may contain at most 2t-1 keys. Therefore, internal node may have at most 2t children.  
  
 """  
 self*.root = BTreeNode([], [], True)  
 *self*.t = *t*

B-Tree를 나타내는 클래스로 BTree를 만들었습니다. 해당 클래스는 루트노드를 가리키는 root 인스턴스 변수와, 노드가 가질 수 있는 최소 child의 개수를 표현하는 t (minimum degree)라는 인스턴스 변수를 가지고 있습니다. B-tree의 차수(Order)는 2\*t로 표현할 수 있습니다. 생성자에서는 빈 노드를 루트 노드로 설정하고, is\_leaf를 True로 설정합니다. 또한, 해당 트리의 속성인 t값을 설정해줍니다.

def b\_tree\_search(*self*, *x*, *k*):  
 *"""  
 b\_tree\_search takes as input an object of root node x of a subtree  
 and a key k to be searched for in that subtree.* ***:param*** *x: A root node of a subtree* ***:param*** *k: A key to be searched for* ***:return****: A tuple (node, index) where 'node' is the node containing the key 'k', and its index is 'i'. Returns None if 'k' is not found.  
 """* i = 0  
 while i < *len*(*x*.key\_value\_list) and *k* > *x*.key\_value\_list[i][0]:  
 i = i + 1  
 if i < *len*(*x*.key\_value\_list) and *k* == *x*.key\_value\_list[i][0]:  
 return (*x*, i)  
 elif *x*.is\_leaf:  
 return None  
 else:  
 return *self*.b\_tree\_search(*x*.children[i], *k*)

트리에 k라는 key가 있는지 탐색하여 있는 경우, 찾은 경우 해당 노드 객체와 해당 키의 인덱스 값을 튜플 형식으로 반환하고, 존재하지 않으면 None을 반환합니다. 해당 함수는 노드 객체와 찾으려는 key 값을 인자로 받고, 유저 프로그램에서 호출될 때는 x에 트리의 루트 노드가 인자로 들어가 호출됩니다.

def b\_tree\_split\_child(*self*, *x*, *i*):  
 *"""  
 The procedure splits the child(x.c[i]) in two and adjusts x so that it has an additional child.  
 parent: x  
 left-child: y  
 right\_child(new): z  
  
 Node y originally has 2t children(2t - 1 keys) but is reduced to t children (t - 1 keys) by this operation.* ***:param*** *x: A non-full internal node x* ***:param*** *i: An index i such that x.c[i] is a full child of x.* ***:return****: None. This function performs its operation without returning a value.  
 """* y = *x*.children[*i*]  
 z = BTreeNode([], [], y.is\_leaf)  
 z.key\_value\_list.extend(y.key\_value\_list[*self*.t: 2 \* *self*.t - 1])  
 for j in *range*(*self*.t, 2 \* *self*.t - 1):  
 y.key\_value\_list.pop()  
 if not y.is\_leaf:  
 z.children.extend(y.children[*self*.t: 2 \* *self*.t])  
 for j in *range*(*self*.t, 2 \* *self*.t):  
 y.children.pop()  
 *x*.children.insert(*i*+1, z)  
 *x*.key\_value\_list.insert(*i*, y.key\_value\_list[*self*.t - 1])  
 y.key\_value\_list.pop()

x라는 노드(부모)의 i번째 자식에 해당하는 노드를 split하고 해당 자식 노드의 가운데(median) 값에 해당하는 데이터를 x로 올려 삽입하는 함수입니다. x가 부모, y가 왼쪽(기존에 꽉 찼던 자식 노드), z가 새로 만들어져서 오른쪽 자식 노드가 됩니다. x의 i번째 자식객체를 y가 가리키게 한 뒤, y의 중간 오른쪽 데이터들을 z로 전부 이동시키고, 가운데 노드는 x의 i번째로 끼워넣습니다. 자식의 경우 i+1(오른쪽 자식이 되기 때문)인덱스에 끼워넣습니다.

def b\_tree\_insert(*self*, *k*, *v*):  
 *"""  
 Insert a key k and v pair(tuple) into the B-tree in a single pass down the tree.  
 The b\_tree\_insert procedure uses \_b\_tree\_split\_child to guarantee that the recursion never descends to a full node.* ***:param*** *k: A key, value pair to insert.* ***:return****: None. This function performs its operation without returning a value.  
 """* r = *self*.root  
 if *len*(*self*.root.key\_value\_list) == 2 \* *self*.t - 1:  
 s = BTreeNode([], [], False)  
 *self*.root = s  
 s.children.insert(0, r)  
 *self*.\_b\_tree\_split\_child(s, 0)  
 *self*.\_b\_tree\_insert\_nonfull(s, *k*, *v*)  
 else:  
 *self*.\_b\_tree\_insert\_nonfull(r, *k*, *v*)

유저 프로그램에서 Insert를 위해 호출하는 함수로, 루트노드가 꽉 차있는 경우 해당 루트 노드를 split해서 위로 올려야 하는데, 루트노드보다 부모 노드는 없기에, 새로운 노드를 하나 만들고, 기존처럼 split을 진행한 뒤, 바로 뒤에 나올 \_b\_tree\_insert\_nonfull함수를 호출합니다. 루트노드가 꽉 차지 않았을 경우, 바로 \_b\_tree\_insert\_nonfull 함수를 호출합니다. 루트 노드 처리를 위해 만들어 놓은 wrapper 함수입니다. 주요 insert 과정은 \_b\_tree\_insert\_nonull 함수에서 진행됩니다.

def \_b\_tree\_insert\_nonfull(*self*, *x*, *k*, *v*):  
 *"""  
 Insert key k and value v into the tree rooted at the nonfull root node.  
 \_b\_tree\_insert\_nonfull recurses as necessary down the tree, at all times guaranteeing  
 that the node to which it recurses is not full by calling \_b\_tree\_split\_child as necessary.* ***:param*** *x: A node to insert.* ***:param*** *k: A key to insert into node x.* ***:param*** *v: A value to insert into node x.* ***:return****: None. This function performs its operation without returning a value.  
 """* i = *len*(*x*.key\_value\_list) - 1  
 if *x*.is\_leaf:  
 *x*.key\_value\_list.append((None, None))  
 while i >= 0 and *k* < *x*.key\_value\_list[i][0]:  
 *x*.key\_value\_list[i + 1] = *x*.key\_value\_list[i]  
 i = i - 1  
 *x*.key\_value\_list[i + 1] = (*k*, *v*)  
 else:  
 while i >= 0 and *k* < *x*.key\_value\_list[i][0]:  
 i = i - 1  
 i = i + 1  
 if *len*(*x*.children[i].key\_value\_list) == 2 \* *self*.t - 1:  
 *self*.\_b\_tree\_split\_child(*x*, i)  
 if *k* > *x*.key\_value\_list[i][0]:  
 i = i + 1  
  
 *self*.\_b\_tree\_insert\_nonfull(*x*.children[i], *k*, *v*)

루트노드에서부터 호출되게 되며, 해당 키 k가 들어갈 자리를 찾아(오름차순을 만족하도록) 해당 자식이 꽉 찼으면 split을 진행하여 자리를 만들고, 꽉 차지 않았으면 해당 노드를 타고 들어가 재귀적으로 leaf까지 도달합니다. 그리고 마지막으로 leaf에 도달한 상황에서는 리프노드는 절대 꽉 차지 않게 됩니다.(바로 이전 재귀함수에서 split을 통해 자리를 만들어주게됨). 이후 오름차순에 맞는 위치에 해당 키 k와, 같이 쌍을 이루는 value값 v를 insert 합니다.

def print\_tree(*self*, *node*, *l*=0):  
 *print*("Level ", *l*, " ", *end*=":")  
 for i in *node*.key\_value\_list:  
 *print*(i, *end*=" ")  
 *print*()  
 *l* += 1  
 if *len*(*node*.children) > 0:  
 for i in *node*.children:  
 *self*.print\_tree(i, *l*)

테스트를 위해 트리 전체를 출력하는 함수입니다. 본 수행 과정에서는 실행되지 않습니다.

class UserInterface:

B-tree의 구현과는 별개로, 유저 프로그램에서 입력을 받고, 파일을 읽고 쓰고, 새로 생성할 파일 명을 만들고, 두 파일이 동일한지 비교하는 등 유저 인터페이스를 담당하는 함수들을 따로 UserInterface 클래스 안에 포함시켰습니다.

@classmethod  
def main(*cls*):

해당 클래스의 전체 기능을 한데 모아놓은 main 함수입니다. 이 외의 함수들은 다 이 main 함수 안에서만 쓰이고, 독립적으로 호출되지 않습니다. 유저 입력을 받아 해당하는 기능의 클래스함수를 호출합니다.

@classmethod  
def \_get\_new\_file\_name(*cls*, *file\_name*):

파일명 중간에 \_created를 추가한 새로운 파일명을 리턴해주는 함수입니다. 예를 들어 input.csv라는 파일명을 input\_created.csv로 만들어줍니다.

@classmethod  
def \_compare\_two\_files(*cls*, *file1\_path*, *file2\_path*):

두 파일 경로에 해당하는 파일들을 비교하여 동일한지 아닌지 여부를 리턴합니다.

@classmethod  
def \_insertion(*cls*):

파일을 읽고 해당 값들에 대해 insertion을 수행하고, 값들을 search하여 파일에 쓰고, 두 파일을 비교하는 전체적인 insertion 과정들을 구현한 함수입니다.

def main():  
 global b\_tree  
 b\_tree = BTree(3) *# t = 3, order = 6* UserInterface.main()  
 sys.exit(0)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

이 파일의 main함수로, 위 경우 b\_tree를 전역변수로 설정하고, b-tree의 차수는 6으로 설정한 이후, UserInterface.main()함수를 실행합니다.

감사합니다.