B+Tree Implement

컴퓨터소프트웨어학부 2016024957 이원석

- Module 목록

- Bisect : leaf-node에서 key의 position을 찾기 위함
- Math : minimum key 계산할때 올림 처리를 위함
- Sys : command line input
- Csv : csv파일 읽기위해서.
- Pickle : 트리정보를 파일에 저장하기 위함.=

- 함수 목록

Class node()

노드 단위 함수

```
def __init__(self,tree,order):
 self.tree=tree #노드의 트리를 참조하기 위해서 저장해두기 (상위 메모리 참조(?))
        self.keys=[]
        self.data=[] # store data if leaf// else store node
        self.next = None
        self.is_leaf=True
        self.order=order
```

초기화

```
def insert(self,index,key,data,parents):
```

삽입

```
def split(self): #self(left) / right
```

스플릿(오버플로우)

```
def overflow(self,parents): #grow up
```

오버플로우 났을 때 스플릿함수 호출 -> 값을 split 값을 위로 올려보냄

```
def underflow(self,parents): #if underflow after remove, borrow or merge
```

언더플로우 났을 때 borrow or merge 작업 수행

```
 def delete(self,index,parents):
```

키,데이터 값 제거

Class BplusTree()

트리 단위 함수

```
class BplusTree(node):
   def __init__(self,order):
        self.order=order
        self.root = node(self,order)
        self.root.tree=self
```

```
    def find_path(self,key): #경로찾기
```

특정 key 값에 대한 node 위치와 index 위치 반환

```
    def insert(self,key,data):
```

트리단위 값 삽입

```
    def delete(self,key):
```

트리단위 값 제거

```
    def single_search(self,key):
```

key 값에 대한 data 출력

```
    def range_search(self,key1,key2):
```

key1 과 key2 사이의 key,data 값 출력

대략적 알고리즘 설명은

트리 단위 삽입,제거함수와 노드 단위 삽입,제거 함수를 분할하여 -> 삽입과 제거가 일어나면 -> 트리단위에서 삽입과 제거함수 수행 -> 트리 단위 삽입,제거 함수에는 find_path 함수가 있어서 노드 단위 삽입과 제거함수를 호출하게 되어있음.
이때 overflow 와 underflow 는 노드 단위에서 일어나기 때문에 그에 따른 작업을 하는 함수는 모두 노드 단위 함수입니다.

구체적인 함수 설명은 기존 코드에 주석을 더 추가하여 코드내에서 작성했습니다.

-B+Tree 의 Node 구현 알고리즘

```
class node():

    def __init__(self,tree,order):
        self.tree=tree #노드의 트리를 참조하기 위해서 저장해두기 (상위 메모리 참조(?))
        self.keys=[] #노드 키값 저장
        self.data=[] # store data if leaf// else store node
        self.next = None #next 노드 지정
        self.is_leaf=True # 처음 노드가 생길때는 모두 리프노드로 받아들임
        self.order=order # 자기 자신의 order 를 참조.


    def insert(self,index,key,data,parents): #node 단위 insert->simply insert
        self.keys.insert(index,key)
        self.data.insert(index,data)
        if len(self.keys)>=self.order: #키값이 order 를 넘어서면 overflow 호출
            self.overflow(parents)

    def split(self): #self(left) / right
        mid=self.order//2 # right biased tree 를 구현하려고 미드값을 이렇게 설정했습니다.
        right=node(self.tree,self.order)
        push_key=self.keys[mid]
        if self.is_leaf:
            right.keys=self.keys[mid:]
            right.data=self.data[mid:]
            self.data=self.data[:mid]   ##리프일때는 자신의 데이터값을 유지

        else:
            #리프가 아니면 미드값을 올려보내고 다시 내릴필요 없음
            right.keys=self.keys[mid+1:]
            right.data=self.data[mid+1:]
            self.data=self.data[:mid+1] # mid 값만 올라가기때문에 왼쪽에서 자식노드 하나를 더 가지고 가야됨

        right.is_leaf=self.is_leaf
       # self(left)와 right 는 서로 같은 레벨에 위치하므로 left 의 next 는 right 로 설정.
        right.next=self.next
        self.next=right
        self.keys=self.keys[:mid]

        self.next=right
        return right,push_key
        #left -> stay at same position right.keys[0] goes to parent key
        # right -> parent.data[index+1] (append)

    def overflow(self,parents): #grow up
```

```
        right_sibling,new_key=self.split()
        if parents: # if parents exist -> 값 올리기
            parent,parent_index=parents.pop()
            parent.keys.insert(parent_index,new_key)
            parent.data.insert(parent_index+1,right_sibling)#data는 한칸 뒤에 기록해줘야함.
            if len(parent.keys)>=self.order: #since parent size up, check overflow again
                parent.overflow(parents)
            #부모에서 overflow난거랑 리프에서 난거랑 구분해야됨

        else: # 부모가 없으면 새로운 부모를 만들어서 split값을 올려줘야 함.
            new_node=node(self.tree,self.order)
            new_node.keys.append(new_key)
            new_node.data.append(self)
            new_node.data.append(right_sibling)
            new_node.is_leaf=False # 부모노드니까 리프노드가 아님.
            new_node.tree.root=new_node # 새롭게 생긴 부모는 트리의 루트가 됨.

    #underflow -> check sibling exist-> check sibling size-> can lend?lend func:merge
    def lend(self,parent,parent_index,borrower,borrower_parent_index):
        # overflow가 났을때 leaf_node 단위에서 값을 빌려주는 함수

        #try begging left node first
        #size check은 호출 전에 underflow에서 하자.
        if parent_index<borrower_parent_index:
            # borrow from right / self=left sibling
            borrower.keys.insert(0,self.keys.pop())
            borrower.data.insert(0,self.data.pop())
            parent.keys[parent_index]=borrower.keys[0]

        else:
            #borrow from left
            borrower.keys.append(self.keys.pop(0))
            borrower.data.append(self.data.pop(0))

            parent.keys[borrower_parent_index]=self.keys[0]

    def underflow(self,parents): #if underflow after remove, borrow or merge
        minimum=math.ceil(self.order/2)-1
        parent,parent_index=parents.pop()
        #if right_sib>min borrow
        left_sibling=None
        right_sibling=None
        if parent_index+1<len(parent.data): # right_sibling이 존재할 조건
            right_sibling=parent.data[parent_index+1]
        if parent_index: #left_sibling이 존재할 조건
            left_sibling=parent.data[parent_index-1]


        if self.is_leaf: #leaf 단위 underflow와 internal node 단위 underflow는 서로 다름.

            #borrow <from> right sibling
            if right_sibling:
                if len(right_sibling.keys)>minimum:
                    right_sibling.lend(parent,parent_index+1,self,parent_index)
                    return
            #borrow <from> left
            if left_sibling:
                if len(left_sibling.keys)>minimum:
                    left_sibling.lend(parent,parent_index-1,self,parent_index)
                    return

            #if not borrow, merge
            #merge with left
            if left_sibling:
                left_sibling.keys.extend(self.keys)
                left_sibling.data.extend(self.data)
                left_sibling.next=self.next
                #delete split key
```

```
                parent.delete(parent_index-1,parents)
                return
            #merge with right
            if right_sibling:
                self.keys.extend(right_sibling.keys)
                self.data.extend(right_sibling.data)
                self.next=right_sibling.next
                #delete split key
                parent.delete(parent_index,parents)
                return

        elif not self.is_leaf:
            #borrow from parent or merge with sibling, parent(split value)
            #internal node 에서의 underflow는 parent에서 값을 borrow 하거나
            self + sibing + parent 세 개를 merge한다 이때 parent는 key값만 내려옴

            #borrow from parent
            if left_sibling:
                if len(left_sibling.keys)>minimum:
                    self.keys.insert(0,parent.keys.pop(parent_index-1))
                    self.data.insert(0,left_sibling.data.pop())
                    parent.keys.insert(parent_index-1,left_sibling.keys.pop())
                    return
            if right_sibling:
                if len(right_sibling.keys)>minimum:
                    self.keys.append(parent.keys.pop(parent_index))
                    self.data.append(right_sibling.data.pop(0))
                    parent.keys.insert(parent_index,right_sibling.keys.pop(0))
                    return

            #merge self + parent + sib
            #case1 : merge with left
            if left_sibling:
                left_sibling.keys.append(parent.keys.pop(parent_index-1)) #split 키를 가져옴
                parents.data.pop(parent_index-1)
                left_sibling.keys.extend(self.keys)
                left_sibling.data.extend(self.data)
                left_sibling.next=self.next
                if parent==self.tree.root: #parent와 merge했는데, parent가 root라면
                    if len(parent.keys) > 1:
                        parent.delete(parent_index-1,parents)
                    else: # parent에 key가 없다면 root가 바뀐거임.
                        self.tree.root=self
                return

            #case2 : merge with right
            if right_sibling:
                print(parent.keys[parent_index])
                self.keys.append(parent.keys.pop(parent_index))
               # print(parent.keys)
                print(len(parent.keys))
                parent.data.pop(parent_index)
                self.keys.extend(right_sibling.keys)
                self.data.extend(right_sibling.data)
                self.next=right_sibling.next
                if parent==self.tree.root:#parent와 merge했는데, parent가 root라면
                    if len(parent.keys) > 1:
                        parent.delete(parent_index,parents)
                    else:
                        self.tree.root=self # 트리 root 변경
                return




    def delete(self,index,parents):

        # tree단위 delete에서 find_path를 통해 노드와 인덱스 부모를 미리 find해놓고
```

```
        # leaf인지 아닌지 케이스 나누기
        minimum=math.ceil(self.order/2) - 1
        key=self.keys[index]
        if self.is_leaf:
            self.data.pop(index)
            self.keys.pop(index)

            if index==0: # delete하면 key값에 해당되는 index를 지워야함
                if len(self.keys)>0:
                   smallest=self.keys[0]
                else:
                    smallest=self.next.keys[0]
                for i,parent in enumerate(parents):  #여기서 i는 enumerate쓰는데 헷갈려서 인덱스 인자로 넣었습니다
                    _Node=parent[0]
                    Index=parent[1]
                    if _Node==parents[-1][0]:
                        if len(self.keys)<minimum:
                         # underflow나면 어차피 borrow나 lend를 통해서 index값이 변경되게 되어있으므로 pass
                            break

                    if key in _Node.keys:
                        _Node.keys[Index-1]=smallest # index값을 가장 작은 값으로 대체해줌
                        break


        else:
        #leaf 노드가 아닌 internal_node 일때
            self.keys.pop(index)
            self.data.pop(index+1) #internal node 일때는 data에서 index+1 자리가 pop됨

        if len(self.keys)<minimum:
            self.underflow(parents) #merge or borrow 재귀적으로 수행.
```

-B+Tree Tree 구현 알고리즘

```
class BplusTree(node):
    def __init__(self,order):
        self.order=order
        self.root = node(self,order) # order=m인 루트노드 생성
        self.root.tree=self # Tree는 root만 저장하면 나머지는 다 참조할 수 있음


    def find_path(self,key): #경로찾기
        cur_node=self.root
        parents=[]

        while not cur_node.is_leaf:

            for i,item in enumerate(cur_node.keys):
                if key<item:
                    index=i
                    break
                elif key==item:
                    index=i+1
                    break
                elif i == len(cur_node.keys)-1: #data가 key보다 1개 더 많으므로
                    index=i+1

            parents.append((cur_node,index))
            cur_node=cur_node.data[index]


        index=bisect.bisect_left(cur_node.keys,key) # find에서 정확한 position은 bisect로 찾음.
```

```
        parents.append((cur_node,index))
        return parents #앞에서부터 차례대로 부모와 인덱스정보가 들어있음 맨 앞에는 루트 맨뒤에는
                       #해당 key 값이 들어있는 leaf node


    def insert(self,key,data):
        parents=self.find_path(key)
        node,index = parents.pop()
        node.insert(index, key,data,parents)


    def delete(self,key):
        parents=self.find_path(key)
        node,index=parents.pop()
        if node.key[index] !=key: # 트리에 없는 값이면 delete 수행 하지 않음.
            return
        node.delete(index,parents)


    def single_search(self,key):
        parents = self.find_path(key)

        Size=len(parents)
        i=0
        check_node,check_index=parents.pop()
        if(check_node.keys==[]  or check_index>=len(check_node.keys) or check_node.keys[check_index]!=key ):
            print("NOT FOUND\n")
            #빈 트리일때, 해당 값이 없을 때는 찾지 못함.
            return

        while i<Size-1:
            cur=parents[i][0]
            print(cur.keys,"\n") # 리프노드까지의 경로에 있는 모든 키를 출력
            i+=1

        print(check_node.data[check_index],'\n') #leaf node 에서 키에 해당하는 데이터 출력

    def range_search(self,key1,key2):
        #key1 이상 key2 미만인 값들을 모두 출력.
        parents=self.find_path(key1)

        check_node,check_index=parents[-1]
        i=check_index
        cur=check_node
        while True:
            if cur.keys[i]>=key1:
                if cur.keys[i]> key2:
                    break
                elif cur.keys[i]<=key2:
                    print(cur.keys[i],cur.data[i])
                    i+=1
                    if i>=len(cur.keys):
                        if cur.next:
                            cur=cur.next
                            i=0
                        else:
                            break
```

Main 코드

```
if __name__ == '__main__':
    if sys.argv[1]=='-c':
        with open(sys.argv[2],'wb') as File:
            My_Tree=BplusTree(int(sys.argv[3])) # create tree with order argv[3]
            pickle.dump(My_Tree,File)

    elif sys.argv[1]=='-i':
```

```
        with open(sys.argv[2],'rb') as File:
            My_Tree=pickle.load(File)
            #삽입 수행 전 파일을 먼저 읽어오기
        with open(sys.argv[2],'wb') as File:
            with open(sys.argv[3],'r') as Insert_File:
                lines=csv.reader(Insert_File,delimiter=',')
                for line in lines:
                    My_Tree.insert(int(line[0]),int(line[1]))
                pickle.dump(My_Tree,File)
                #해당 파일에 트리정보 갱신

    elif sys.argv[1]=='-d':
        with open(sys.argv[2],'rb') as File:
            My_Tree=pickle.load(File)
            #삭제 수행 전 파일을 미리 읽어오기
        with open(sys.argv[2],'wb') as File:
            with open(sys.argv[3],'r') as Delete_File:
                lines=csv.reader(Delete_File,delimiter=',')
                for line in lines:
                    My_Tree.delete(int(line[0]))
                pickle.dump(My_Tree,File)
                #해당 파일에 트리 정보 갱신


    elif sys.argv[1] =='-s':
        with open(sys.argv[2],'rb') as File:
            My_Tree=pickle.load(File)
            My_Tree.single_search(int(sys.argv[3]))

    elif sys.argv[1] =='-r':
        with open(sys.argv[2],'rb') as File:
            My_Tree=pickle.load(File)
            My_Tree.range_search(int(sys.argv[3]),int(sys.argv[4]))

    elif sys.argv[1] =='-check':
        #트리의 전체적인 구조를 확인하기 위해서 임의로 추가한 명령어입니다.
        with open(sys.argv[2],'rb') as File:
            My_Tree=pickle.load(File)
        cur=My_Tree.root
        while True:
            leftmost=cur.data[0]
            while cur.next:
                print(cur.keys,end='     ')
                cur=cur.next
            print(cur.keys)
            if cur.is_leaf==True:
                break
            cur=leftmost
```

위 코드는 visual studio code 의 최신버전에서 작성한 코드이며
실행은 윈도우의 cmd 창을 통해서 실행했습니다.
명령어는 과제 예시를 준수하여 작성했습니다.

입력 예)
B+Tree.py –c index.dat ‘order’
B+Tree.py –I index.dat input.csv
B+Tree.py –d index.dat delete.csv
B+Tree.py –s index.dat ‘key’
B+Tree.py –r index.dat ‘key1’ ‘key2’

제 코드는 실행했을 때 데이터파일을 로드합니다. 하지만 로드한 이후 과정에서 에러가 생기면 데이터파일 저장코드가 실행이 되지 않기 때문에 데이터 파일의 정보가 모두 제거됩니다.