B+tree_Assignment

1. 알고리즘 요약

:기본적으로 b+tree의 알고리즘을 따르며 삽입,삭제,탐색의 기능들이 구현이 되어 있습니다.

명세서 상에서 명시가 되어있었던 CLI를 이용하여서 입력과 실행이 진행 되어야 함으로

어떠한 명령을 실행할지를 받아주는 부분은 sys.argv[1]에서 저장이 되며

index.dat를 받는 부분은 sys.argv[2]부분에서 저장이 되고

input.csv, delete.csv를 받아 주기 위한 부분은 sys.argv[3]에서 저장이 되어 cli명령을 수행할수 있게 하였습니다.

index file 구조는

트리 구조를 저장하고 불러오는 방식을 생각해보았는데 좋은 방법을 떠올리지 못해서

4 #degree

i 45 451361 #insert할시 key, value

...

d 15 #delete할시 key

등의 방식으로 index.dat에 입력받은 모든 데이터를 입력 받도록 하였습니다.

(중복입력이 된 수들은 저장하지 않는 방식으로 하였습니다)

그래서 매번 실행 할때마다 모든 데이터를 다시 입력하고 트리를 구성하며

input.csv, delete.csv 혹은 single,range search 의 기능들을 실행할시 여러번

insert,delete를 거친 후에도 지금까지의 모든 명령이 담겨 있어서

실행하고자 하는 기능들이 실행이 되도록 하였습니다.

데이터 구조

:plustree, LEAFNODE,NODE의 세가지 형태로 구분하여서 코드를 작성하였습니다.

LEADNODE와 NODE는 보고서에서 명시되어 있는 것처럼

m,p,r의 개념들이 들어가 있습니다.

NODE,LEAFNODE에 공통적으로 들어가 있는 부분부터 설명 드리자면

m은 key들의 집합입니다. 이부분에서는 data의 삽입을 위해 넣어준 key들이 들어가는 부분입니다.

그리고 degree는 처음에 해당 노드가 가질수 있는 최대 포인터 개수를 나타내는데 둘다 동일하게 가질수 있도록 하였습니다.

parent는 각 노드의 부모노드를 나타내어 줍니다.

서로 다르게 들어가게 되는 부분은

value인데 NODE에서는 VALUE부분에 자식 노드들을 가리키는 주소값, 즉 포인터

가 들어가게 되고 LEAFNODE에서는 처음에 KEY, VALUE 로 입력 받았을때의

VALUE값이 value에 들어가게 됩니다.

그리 하여서 탐색을 통하여 찾게 되었을시 LEAFNODE에서는 주소값이 아닌

KEY값이 가리켜야할 VALUE값을 출력해줄수 있게 하였습니다.

그리고 r 또한 서로 다르게 들어가게 되는 부분인데

NODE에서는 가장 오른쪽 자식 노드를 가리키게 하여주며

LEAFNODE에서는 다음 LEAFNODE를 향하는 포인터 주소를 가지고 있습니다.

삽입:

1. 맨처음 삽입을 할 때(아직 SPLIT이 발생하기 전)

삽입이 발생할때는 우선 리프노드에 저장을 시켜 줍니다.

이때 리프노드에 저장시킬 때 저장시켜줄 위치를 find함수를 통해서

리프 노드내에서 key가 오름차순으로 저장이 되어 있는데, 적절한 위치를 찾게 하여 주어서, 들어가야할 위치에 저장을 시켜줍니다.

이때 key와 value는 인덱스 상으로 같은 곳에 저장이 됩니다.

2. 노드의 크기가 꽉차서 SPLIT이 발생!

이때는 SPLIT을 통하여 하나의 리프노드는 총 세 개로 쪼개어 지게 됩니다.

하나는 위로 올라갈 노드, 올라간 노드에서 왼쪽 자식에 해당될 노드,

마지막 하나는 올라간 노드에서 오른쪽 자식에 해당될 노드.

이렇게 세 개로 나누어 지게 되는데, 올라갈 노드는 degree를 2로 나눈 몫에 해당하는 인덱스에 위치한 노드를 올리게 됩니다.

이때 올라간 노드는 더 이상 리프 노드가 아닌 그냥 노드가 될것이 므로

새롭게 노드로써 생성을 하여 주며, value에 분할된 노드들을 왼쪽, 오른쪽 자식으로 가지게 하여 줍니다.

이때 올라간 노드 위에 부모 노드가 존재한다면 똑같이 들어갈 위치를 찾아 넣고

또 그 크기가 SPLIT이 발생 하여야 한다면 SPLIT을 하여 줍니다.

이렇게 해서 루트 노드까지 올라가면서 SPLIT의 발생여부를 확인 하여 줍니다.

삭제:

1.노드에서 삭제가 가능한 상태여서 삭제할수 있는 경우(underflow가 발생하지 않은 경우)

삭제는 삭제하려는 키 값이 리프노드에만 존재하는가? 아니면 리프노드와

노드에 둘다 존재하는가? 로 나뉘었습니다.

1-1.삭제하려는 키값이 리프노드에만 존재할시:

1)키값이 리프노드에만 존재하는데 노드의 맨앞에 위치하지 않는 경우

:이때는 그저 해당 노드에서 지워야 하는 key값과 value값을 삭제하여 줍니다.

2) 키값이 리프노드에서 삭제할시 삭제하게 되면 underflow가 발생할시

(이경우는 여기에 넣는게 전체적으로 더 나을거 같아서 넣었습니다)

: 이때 에는 형제노드(부모 노드에 자식 노드에 속해 있는 노드중 현재 삭제하고자 하는 노드의 왼쪽 오른쪽 노드를 나타냅니다)

에서 키를 빌려와서 해결하여 줍니다.

만약 빌려올수 없는 상황일때는 merge작업을 진행하여 줍니다.

2.삭제할 키 값이 리프노드와 내부 노드 모두에 존재하는 경우

2-1)리프노드에서 삭제가 가능할시

: 이경우에는 삭제를 한다해도 노드가 underflow가 발생하지 않는 경우로써 해당되는 리프 노드에서 삭제하고자하는 key값을 삭제 할수 있는 경우입니다. 물론 key값에 해당이 되었던 value값을 삭제 하여주고

해당 노드가 삭제 된뒤 내부 노드 상에서도 똑같이 존재하는 key값을 찾아 내어서 들어가야 하는 알맞은 값을 넣어줍니다.

2-2)삭제 하고자 하는 리프 노드 상에서 삭제할시 underflow가 되는 경우

:이때 에는 삭제한다면 리프노드가 가질수 있는 최소한의 값보다 적은

개수를 가지게 됩니다. 그렇다면 부모 노드에서 현재 삭제하고자 하는

key값을 가지고 있는 노드의 형제 노드 들을 찾습니다.

(현재 노드를 가리키는 value인덱스의 -1,+1이 각각 왼쪽 형제, 오른쪽

형제가 됩니다.)

형제 노드에서 빌리는 작업은 왼쪽 형제노드가 빌려줄수 있는 상태

(현재 형제 노드가 가지고 있는 키 개수에서 -1을 한다고 해도
underflow가 되지 않음)

라면 빌리고 그렇지 않다면 오른쪽 노드에서 빌립니다.

그러나 둘다 빌릴수 없는 상태라면 다시 왼쪽->오른쪽 형제 노드 순서로

merge작업을 진행 합니다.

2-3) 2-1) 의 경우처럼 부모 노드에 위치해 있는 삭제하려는 key값을

알맞은 키값으로 교체하여주는 작업이 진행 됩니다.

이 경우에는 borrow을 할수 없는 상황이라 ,merge작업이 진행이 되는데 merge를 한후에 삭제된 부모 노드에 위치한 키값을 알맞은 키값으로 채워주는 작업 입니다.

탐색:

single key search:

b+ 트리의 루트 노드에서 부터 시작하여 찾고자 하는 키 값을 기준으로

자식 노드들을 찾아가면서 자식노드들의 모든 키 값을 출력하여 주며

리프노드까지 내려갔는데 value값이 있을시 출력을 하여줍니다.

없으면 NOT FOUND를 출력시켜주었습니다.

ranged search:

찾고자 하는 키값을 따라 쭉 리프노드까지 내려간 후에

리프노드 는 형제노드를 가리키는 포인터가 있으니 찾고자 하는 키값보다 같거나 큰 key, value값을 출력을 하며 end_key가 가리키는 값까지 따라가서 출력을 하여주었습니다.

2. 각 코드에 대한 함수설명

2-1) 클래스 설명:

class plustree

: 이 클래스는 우선 초기에 LEAFNODE로써 초기화 되어 생성이 되며,

초기 생성시 루트는 자기 자신으로 지정이 되며 degree또한 입력을 받아 생성이 됩니다.

class LEAFNODE

:리프노드는 초기화시에 m,value,degree,parent,right등의 변수들을 초기화 시켜주며 insert시에 필요한 split함수를 작성하여 놓았습니다,

split이 발생할때 NODE에서의 split과 다르게 value 부분에 포인터(주소값)이 아닌 처음에 입력 받았던 value값들을 넣어 줍니다.

class NODE

:노드는 LEAFNODE와 비슷하게 진행이 되며

split이 발생할때 LEAFNODE에서의 split과 다르게 value 부분에 포인터(주소값)가 들어가게 됩니다.

class plustree

-def insert(self,key,value)

:node=self.root을 통하여 가장 최상단의 루트 노드임을 나타내며

type(node)로 리프 노드인지 그냥 노드인지 판별을 하며 리프 노드까지 내려가 줍니다.

이때에는 self.find(node,key)로써 내려가는 포인터 위치를 판단하게 하여 줍니다.

그렇게 하고 나서 가장 처음에 삽입을 하여야 하는지 그게 아니라면 들어가야 하는 위치에 넣어야 할 key, value값을 삽입을 하여준후

if len(node.m)>=node.degree로 써 split이 발생하여야 할 조건에 해당이 된다면

해당 조건을 실행 시키게 하여 줍니다.

split-1.리프노드에서 split 2.노드에서 split으로 나뉘는데

1.리프노드에서 split인 경우라면

: 나눠야할 세개의 노드로 쪼개어서 올릴 노드의 위치가 위에 있는 노드의 위치에

가장 앞에 들어갈지, 가장 마지막에 들어갈지, 앞의 두 경우와 다르게 중간에 들어갈지로 분류해서 진행을 시켜 줍니다.

이때에 올라가게 된 노드의 value(자식)들의 parent는 올리게 된 한 노드가 아니라 올라가서 속하게 된 노드가 parent가 될테니 그점을 처리하는 코드또한 넣어 주었습니다.

그후 if node.parent == None일시 루트 노드를 node로 처리를 하여 주었습니다.

그 이유는 node의 부모가 더이상 없다는 것은 루트 노드를 의미하여서 이렇게 하였습니다.

이렇게 한후 node=node.parent(코드상에서는 pi=node.parent인데 좀더 잘 설명을 하기 위해 이렇게 적었습니다)로써 부모 노드로 올라 가면서

len(node.m)>=node.degree로써 split을 발생 시켜야 하는지 체크 하였습니다.

2. 노드에서 split
이것 또한 위에서 설명한 것과 같이 같은 과정으로써 진행이 되는데 올려주려는 노드가 중간 부분에 들어갈시 들어가고자 하는 인덱스에 존재하던 포인터를 제거한후 그위치에 들어가야할 포인터를 넣어주는 작업을 하였습니다.

def delete(self,key)
:삭제 기능을 맡고 있는 기능을 하여 주는 함수인데 리프 노드까지 find함수를 통하여 내려 가게 되며 삭제하려는값이 리프 노드 뿐만이 아닌 내부 노드에도 존재를 하는지 탐색하게 하여 주었습니다.

그후 부모 노드에서 형제 노드의 인덱스를 알아내기 위한 부분을 진행한후 찾아낸 인덱스들이 부모 노드에 진짜 속해있는지를 탐색하기 위한 부분을 넣어 주었습니다. 그렇게 해서 진짜 존재한다면, left_t,right_t로써 각 각 존재 하는 형제 노드들을 넣어 주었습니다.
그후 1. 키가 리프 노드에만 존재, 2. 키가 내부 노드 에도 또한 존재
의 경우로 나누어서 진행 하였습니다.

1. 키가 리프 노드에만 존재에서는 ok_left,ok_right로써 양쪽 노드의 존재여부를 파악후 ok_left_borrow,ok_right_borrow을 진행하여 각 형제노드가 borrow을 할수 있는지 체크를 하여줍니다.
1-1)삭제시 노드안에서 그저 삭제할수 있다면(underflow가 발생하지 않는 상황) 삭제를 진행하여 주고, 그렇지 않다면 borrow작업을 통하여 삭제 과정을 진행 하여 주었습니다 또 borrow작업 또한 진행을 할수 없다면 merge작업을 하여 주었습니다.
(borrow는 left_left,right_leaf,left_node,right_node의 경우로 나누어서 작성을 하였는데 밑에서 따로 함수를 설명할때 기능 구현 내용에 대해서 적겠습니다!)

2. 키가 내부노드에도 존재하는 경우
2-1) 그냥 리프노드에서 삭제가 가능할시

:위에서 진행했던 경우와 같이 그저 삭제를 하여주고
node2(:키값이 존재하는 부모노드) 에 존재하는 키값을 삭제가 되었던
리프 노드의 가장 첫번째 위치하는 키 값으로 변경시켜 줍니다.

2-2) 이 상태 에서는 삭제를 진행하려고 하는 리프노드에서 삭제시
underflow가 발생을 하여 borrorw 진행을 하며 그것도 안될시 merge작업을 진행
하게 하였습니다.

그후에 그렇게 삭제를 진행한후 삭제가 된후 node=node.parent를 하여서
부모 노드로 올라가면서 올라가게 된 노드들이 최소 노드 보다 적은 갯수를 가지고 있
다거나 혹은 가지고 있는 키값들의 수에 비해 적은 value(자식노드 가리키는 포인터)의
 개수를 가지고 있다면 이때는 노드에서 borrow,merge작업을 진행하게 하여
balance를 지키기 위한 작업을 하였습니다.
이때에 root노드는 노드가 1개여도 상관이 없어서 root노드는 계속 올라가면서 탐색 하
였던 조건과 상관이 없으니 root 노드가 나오게 된다면 탐색을 멈추게 하였습니다.

def borrow_left_leaf(self,left_t,node,parent,key)
:이 함수는 왼쪽 형제 노드의 자식을 빌려올수 있는 상황에 실행이 되는 함수입니다.
왼쪽 형제 에서 빌려올것이니 key,value는 가장 마지막에 위치한 것들을 빌려올
작업을 진행하고자 합니다.
빌려 올 키 값은 부모 노드의 키값으로 올라가 주어야 합니다 ,
그래서 그 작업을 진행후 빌려올 key,value들을 왼쪽 형제 노드에서 제거후
빌려온 값들을 delete작업을 진행해 주어야 할 값인 key 가 위차하는 곳에
빌려온 key,value값들을 넣어 주었습니다.
그후 부모의 부모가 없는 상황 즉, 부모가 root가 되야 한다면 부모가 root가 되도록
하여 주었습니다.

def borrow_right_leaf(self,right_t,node,parent,key)
:이때 borrow작업을 진행하고자 할시에 오른쪽 형제 노드의 가장 앞에 있는 값들을
빌려 오게 될것입니다.
이때에는 빌려오게 된다면 부모노드에서 오른쪽 형제 노드를 가리키는 키 값또한

오른쪽 형제 노드의 가장 앞에 위치한 키값을 빌려주게 되니 그 키값을 제거하게 되면 가장 앞에 오게될 인덱스 1 에 위치한 키값을 부모 노드가 오른쪽 형제 노드를 가리키는 키값으로 교체 하여줍니다.
그후 오른쪽 형제 노드에서 빌려올 값들을 제거 해 주고
빌려오고 난후 빌린 노드에 넣어주는 작업을 진행하였습니다.

def merge_left_leaf(self,left_t,node,parent,key)
:이때 에는 왼쪽 형제 노드에 위치하는 값과 합쳐야 하는 상황일때 진행이 되는데
오른쪽에 있는node를 left_t로 옮기는 작업을 진행하고자 합니다.
이때에 m,value를 삭제를 한후에 부모 노드 에 올려야 하는 값을 node의 가장 첫번째 m 의 인덱스에 위치한 값을 옮겨주고 현재 node를 가리키는 포인터 또한 제거 하여 줍니다.
그후 왼쪽 노드에 node가 가지고 있던 m,value를 붙여주고 left_t의 right 리프노드를 가리키는 포인터를 node가 가지고 있던 right로 넣어줍니다.

def merge_right_leaf(self,right_t,node,parent,key)
:이때 에는 오른쪽 형제 리프 노드와 병합을 진행하여야 하는데
부모 노드에서 오른쪽 형제 노드를 가리키는 인덱스를 찾아내고
왼쪽 노드에 오른쪽 형제 노드가 가지고 있던 m,value들을 붙여 주었습니다
그후 삭제하고자 했던 key를 m에서 지우고 key가 가리키던 value또한 지워 주었습니다.
그렇게 한후 부모 노드에서도 오른쪽 노드를 가리키던 m,value를 지워주어서
리프 노드들이 오른쪽 형제 리프 노드들을 가리키는 순서를 잘 나타 내어 줄수 있도록 하였습니다.

def borrow_left_node(self,left_t,node,parent)
:이 함수는 이제 리프노드가 아닌 내부 노드에서 node를 빌려오는 작업입니다.

왼쪽에 위치하고 있는 형제노드 left_t로부터 value를 빌려오는 작업인데
left_t,node의 부모인 parent에 left_t의 가장 오른쪽 에 있는 키값을 parent에서 left_t를 가리키는 인덱스에 해당되는 m값을 변경 시켜주고
left_t에서는 가장 오른쪽 에 있는 m과 value를 빼주었습니다.
그후 오른쪽 노드인 node에는 가장 왼쪽에 빌려온 m,value 를 추가 시켜 주었습니다.
그후 빌려온 value의 부모 또한 node로 되게 설정해 주었습니다.
그후 키값의 순서를 맞추기 위해 첫번째 인덱스에 있는값을 node의 m값으로 입력시켜 주어서 과정을 마무리 지어 주었습니다.

def borrow_right_node(self,right_t,node,parent)
:방금 위에서의 borrow_left_node와 같이 과정 자체는 비슷하나 이번에는
왼쪽 노드와는 달리 가장 right_t의 가장 앞에 위치한 m,value값들을 빌려와서 처리를 하여 주었습니다.

def merge_left_node(self,left_t,node,parent)
:왼쪽 노드와의 병합 을 진행 하여야 하는 수행할 작성한 함수입니다.
우선 node를 가리키는 인덱스를 parent의 value에서 찾은후(그 값을 hk라고 하겠습니다) 찾았던 node를 가리키는 부분은 이제 merge를 하여서 사라져야 하니 parent에서 제거를 하여주고 parent의 m에서 node를 가리키던 인덱스 hk-1의 key값을
left_t에 붙여줍니다, 또한 parent에서 제거작업도 하여 줍니다.
그후 node의 m를 left_t에 붙여주고 value들 또한 붙여줍니다.
이렇게 하여서 merge작업이 마무리가 되는데,
부모 노드의 키 값이 없어서 그 길이가 0 이 된다면
plustree의 루트는 left_t로 하여 줍니다.

def merge_right_node(self,right_t,node,parent)
:바로 위에서 나왔던 merge_left_node에서 기술된 것과 같이 비슷하게 진행이 됩니다.
부모 노드에서 node를 가리키는 인덱스를 찾으며 부모 노드에서 right_t에 붙여 주어야 할 m값을 넣어주고 오른쪽 노드에 왼쪽 노드가 가지고 있던 m,value들을 덧붙여 줍니다.

def find(self, ro,fin)

ro:노드, fin:찾고자 하는 키값

ro라는 노드안에 있는 키값들의 인덱스가 존재할텐데 이 인덱스 들을 차례대로 탐색을 하면서 찾고자 하는 key가 존재하는 인덱스의 키 값보다 작으면 그 위치가

이 함수가 찾고자 하는 위치를 나타 냅니다.

만약 끝 까지 탐색 했는 데도 그 위치를 찾아 내지 못했다면

찾고자 하는 인덱스는 ro의 길이의 정수형 수(인덱스)에 해당하게 됩니다.

리턴값은 찾고자하는 위치의 인덱스에 해당하는 value값을 리턴 시켜 줍니다.

def find_internal(self,ro,fin)

:이 함수는 찾고자 하는 fin값이 인터널 노드에 존재 하는지를 탐색 하고자 하는 함수입니다. 이함수가 탐색하고자 하는 ro노드가 리프노드라면 내부 노드에 그 값이 존재 하는 것이 아니니 None을 리턴 시켜주고, 그렇지 않다면 탐색을 통해 값이 존재 하는 지를 찾아 냅니다.

def singlesearch_key(self,key)

:이함수는 탐색 기능중에 하나인 단일 키 값을 찾아 내는 함수입니다.

insert,delete시에 중복된 값이 존재하는지 찾아내기 위해서 만든 함수입니다.

리프 노드까지 내려가서 key가 존재한다면 1을 아니라면 0을 리턴시켜 주었습니다.

def singlesearch(self,key)

:이 함수는 명세서 상에서의 single key search 기능을 수행하기 위하여 만든

함수 입니다.

트리를 탐색하면서 탐색하게된 노드 안의 모든 키값을 차례대로 넣어준후

리프노드까지 탐색을 하여 탐색 하는 동안 만난 노드 들의 모든 키값들을 출력

하여 주고 찾았으면 찾은 키값의 value값을, 못 찾았으면 NOT FOUND를 실행시키게 하였습니다.

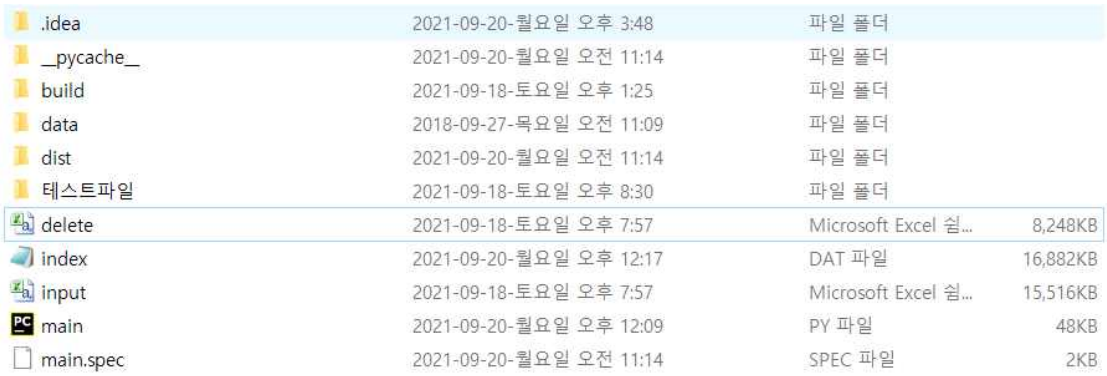
def rangesearch(self, start,end)

:이함수는 명세서 상의 Ranged Search 함수 기능을 동작하게 하기 위하여 만든

함수입니다.

우선 start값이 들어가야 하는 위치라고 할수 있는 리프노드까지 내려간후
end값과 같거나 작은 key, value를 출력을 하도록 하였습니다.

컴파일 방법 설명:
우선 저는 파이썬으로 프로그램을 만들었습니다.
그래서 파이썬이면은 실행 파일을 만들지 않아도 된다고 하였습니다.
그래서 우선적으로 작성하였던 소스코드 파일인 main.py파일을 실행 시키고자 할건데



.idea	2021-09-20-월요일 오후 3:48	파일 폴더	
__pycache__	2021-09-20-월요일 오전 11:14	파일 폴더	
build	2021-09-18-토요일 오후 1:25	파일 폴더	
data	2018-09-27-목요일 오전 11:09	파일 폴더	
dist	2021-09-20-월요일 오전 11:14	파일 폴더	
테스트파일	2021-09-18-토요일 오후 8:30	파일 폴더	
delete	2021-09-18-토요일 오후 7:57	Microsoft Excel 쉼...	8,248KB
index	2021-09-20-월요일 오후 12:17	DAT 파일	16,882KB
input	2021-09-18-토요일 오후 7:57	Microsoft Excel 쉼...	15,516KB
main	2021-09-20-월요일 오후 12:09	PY 파일	48KB
main.spec	2021-09-20-월요일 오전 11:14	SPEC 파일	2KB

이렇게 input과 delete파일과 main파일 이 같이 있는 상황에서
파이썬으로 열린 terminal 상에서

```
(database) C:\Users\Hello\Desktop\database>py main.py -c index.dat 4
```

이렇게 하여
Data File Creation명령을 실행이 가능하고

```
(database) C:\Users\Hello\Desktop\database>py main.py -i index.dat input.csv
```

이렇게 입력을 하면 Insertion이 가능하고

```
(database) C:\Users\Hello\Desktop\database>py main.py -d index.dat delete.csv
```

이렇게 입력을 하면 Deletion이 가능하며

```
(database) C:\Users\Hello\Desktop\database>py main.py -s index.dat 15
1273600,358299,743636,1056542,133047,224139,46555,78133,109229,10399,17939,28240,3497,7053,931,1998,446,623
,144,261,50,98,17,29,6,10
NOT FOUND
```

이렇게 하면 Single Key Search가 가능하고

```
(database) C:\Users\Hello\Desktop\database>py main.py -r index.dat 100 200
```

이렇게 입력시 Ranged Search가 가능하게 됩니다.

혹시나 executable file로 채점이 되어야 하는 상황이라면

dist폴더안의 main폴더에서

cmd 창으로 들어가신후에

Data File Creation

```
C:₩Users₩Hello₩Desktop₩database₩dist₩main>main.exe -c index.dat 5
```

Insertion

```
C:₩Users₩Hello₩Desktop₩database₩dist₩main>main.exe -i index.dat input.csv
```

Deletion

```
C:₩Users₩Hello₩Desktop₩database₩dist₩main>main.exe -d index.dat delete.csv
```

SingleKey search

```
C:₩Users₩Hello₩Desktop₩database₩dist₩main>main.exe -s index.dat 4
```

Ranged Search

```
C:₩Users₩Hello₩Desktop₩database₩dist₩main>main.exe -r index.dat 4 50
```

이렇게 컴파일 이 가능합니다.