2020-2 데이터베이스시스템 B+tree 보고서

2019097347 이주은

#비플트리에서 노드의 키 개수는 (degree-1)/2이상, degree-1이하이다. 루트는 1개 이상의 키 값을 가진다.

#키 값이 부족할 때를 underflow(언더 플로우), 초과하는 경우를 overflow(오버 플로우)로 지칭한다.

#비플트리의 연산은 삽입, 삭제, 탐색(하나의 키, 범위)으로 이루어진다.

#비플트리의 리프 노드는 오른쪽 노드를 가리키며 이어져 있다.

- 보고서 목차
  i. 함수 설명 및 알고리즘
  ii. 실행하기(instruction)
  iii. 데이터 저장과 불러오기

1. 함수 설명 및 알고리즘 (코드 위에서부터 입니다)

- Node class

  Node class에는 과제 공지에 나와있는 an array of*<key, left_child_node>* pairs을 구현하기 위해서, 노드 클래스 내에 inner class key를 만들어서 그 안에 key,value,left child를 저장했다. 노드 클래스의 right는 리프 노드의 경우 오른쪽 형제를 가리키고, non-leaf node의 경우 제일 오른쪽 자식을 가리키도록 했다. 노드 안에는 List<key>가 있어 키와 value를 묶어서 배열에 저장해 두었다.

- Node class 내 is_leaf

  지금 현재 노드가 leaf node이면 true, 아니면 false를 리턴 해주는 클래스 내부 함수이다. 이 함수는 여러 연산과정에서 유용하게 쓰일 예정이다. 현 노드의 child가 null이 아니라면 이는 non-leaf 노드임을 알 수 있다.

- Node class 내 sorting

  노드 내의 키들은, 키 값을 기준으로 오름차순으로 정렬된 상태여야 한다. 그래서 key 값을 기준으로 정렬하는 클래스 내부 함수이다. 키 값을 기준으로 정렬하기 위하여 class key는 Comparable<key>를 implements하고, 내부에 compareTo 함수를 오버 라이딩 해준다.

- Node class 내 delete

  현재 노드에서 어떤 키 값을 지우고 싶은데, 인덱스를 이용하여 지우는 방법(리스트의 remove(idx)함수)도 있지만 인덱스 말고 특정 키 값만 알 때 지우기 위해 만든 함수이다. 다시

말해, 코드 내에서 노드 안의 키 값을 지우는 방식을 두 가지로 하였다. 하나는 List의 remove함수, 하나는 이 함수로. 이 함수는 인덱스 값이 아닌 지우는 키 값을 입력하면 그 키를 찾아서 지운다.

- Findroot

  루트 노드를 찾는 함수이다. 부모가 null이면 종료하도록 하였다. 삽입 과정이 끝나고 나면 항상 이 함수를 호출하여 루트 노드를 찾도록 하였다. 찾는 방법은 루트의 부모가 null일때까지 반복문을 도는 것이다. 루트에 도달하면 부모가 null이므로 함수는 종료된다.

- Where_go

  어떤 키 값이 주어졌을 때, 이 키 값이 들어갈 노드를 루트에서부터 내려가면서 찾는 함수이다. 예를 들어, 어떤 새로운 키를 삽입하려고 할 때 이 키 값이 어떤 노드로 들어가야 할지 찾아 그 노드를 리턴 해준다. 루트에서부터 내려가며 반복문을 돌면서 노드의 키 값들과 비교하여 들어갈 노드 위치를 찾는다.

- Leaf_split

- 변수

  (where : 지금 스플릿 하려는 리프 노드)

  (right : 스플릿을 위해 새로 만든 노드)

- 설명

  이 함수는 키를 삽입할 때, 리프 노드에서 오버 플로우가 났을 경우 그 리프 노드를 스플릿 하는 함수이다. right라는 새로운 노드를 만들어 where노드의 키를 나누어 삽입한다. 쪼갠 후 왼쪽 노드에는 (degree/2)개의 키가, 오른쪽에는 (총 키수-degree/2)개의 키가 들어가게 된다.

  노드를 쪼갠 다음, 오른쪽 노드의 가장 작은 키 값을 부모 노드로 올려준다. 여기서 부모 노드가 없는 경우에는 부모 노드를 생성하여 그 키 값을 넣고 where과 child와 부모를 이어준다. 부모가 존재하는 경우에는, 키 값을 부모로 올려주고, right와 부모를 이어준다.

  여기서 중요한 것은 찢은 두 노드 중에서 왼쪽인 where 이 오른쪽 형제인 right를 가리키게 해야 하는 것이다. 비플 트리의 특징 중 하나는 리프 노드들이 연결되어 있다는 것이다. 이 함수는 리프 노드를 스플릿 하기 때문에 쪼개고 난 뒤 왼쪽 노드가 오른쪽 노드를 가리키게 하는 것이다.

- Non_leaf_split_delete

  이 함수는 delete과정에서 사용하는 스플릿 함수이다. 예를 들어 non-leaf 노드에서 언더 플로우가 발생하여 오른쪽 혹은 왼쪽 형제와 merge를 한 후에, 만약 키 개수가 degree이상이라면 split을 해야 하는 상황이 된다. 이 때 이 함수를 통해 오버 플로우가 발생한 non-leaf 노드를 스플릿 해준다.

  insert용과 delete용을 따로 만든 이유는, delete과정에서는 한 노드를 스플릿 한 이후에 그 위의 노드가 null은 아니지만 키 값이 하나도 있는 경우가 따로 있기 때문에 그 경우를 처리해주기 위해서이다.

- Non_leaf_split

- 변수

  (where : 이 함수의 매개변수, 스플릿 하려는 노드이다)

  (right : 스플릿을 하기 위해 새로 만든 노드)

- 설명

  이 함수는 위의 non_leaf_split_delete와 다르게 삽입 과정에서 사용되는 함수이다. 삽입 과정에서, non-leaf node에서 오버 플로우가 나서 이를 찢어주는 상황에 사용된다. 이 경우 중간 값을 올려주고 원래 노드를 양쪽으로 스플릿을 해준다.

  right라는 노드를 하나 만들어 where에서 중간값 이후 키 값들을 삽입한다. Right에 키를 새로 추가했다면 , where에서는 그 키 값들과 중간값을 remove해준다.

  만약 부모가 null이라면 새로운 노드를 하나 만들고 거기에 중간 값을 넣고 부모와 where, right를 이어준다. 만약 부모가 있다면 중간 값을 올려주고, where , right, 그리고 where.parent를 이어준다.

  키 값을 올린 다음, where.parent의 키 값이 오버 플로우가 난다면 where.parent를 한번 더 스플릿 해줘야 한다. 이 함수를 한 번 더 호출해주면 된다. 재귀 함수를 통해 오버 플로우가 생기지 않을 때까지 반복할 것이다. 이때 매개변수는 where.parent가 될 것이다. 만약 오버 플로우가 나지 않았다면 여기서 종료해준다.

- Insert

  root 가 null이라면, 지금 들어오는 키 값이 제일 처음의 키 값이므로, 먼저 루트를 생성하고 첫번째 키 값을 넣고 종료한다.

  Null이 아니라면 그 키 값이 들어갈 노드를 where_go를 통해 찾는다. 그 노드에 키 값이 들어가도 오버 플로우가 나지 않는다면 그냥 키를 넣고 정렬해주고 종료한다.

  만약 오버 플로우가 난다면 leaf_split함수를 호출한다. 리프 노드 스플릿 이후에 이 노드의 부모의 키가 오버 플로우라면, 부모 노드를 매개변수로 하는 non_leaf_split을 호출한다. insert함수가 한 번 끝날 때 마다 findroot를 호출하여 루트 위치를 찾도록 한다.

- Range_search

  어떤 두 수가 주어졌을 때 트리에서 이 사이 값들을 찾아 출력하는 함수이다. 루트 노드부터 타고 내려가면서 start key와 end key가 들어있는 (혹은 들어있을 수도 있는) 노드를 찾는다.

  두 노드를 찾은 후에, 반복문을 돌면서 시작 노드부터 end노드까지 범위 내에 포함되는 key와 value를 출력한다. End 노드까지 왔다면 반복문을 종료한다. 이때 범위의 경계값 (즉, start key, end key)가 트리 내에 존재한다면, 이들의 key와 value도 출력한다.

- Single_search

  루트에서부터 search하는 키 값이 들어있는(혹은 들어 있을 수도 있는 -> 찾으려는 키가 트리에 없을 수도 있다.) 노드를 찾으면서 내려간다.

  이때 리프까지 내려가는 경로를 저장하는 새로운 List<Node>를 하나 만들어 리프 노드까지 가는 길에 있는 노드를 add한다.

리프 노드까지 내려왔을 때 그 노드에서 search하는 키 값이 있다면 리스트에 저장해둔 지나온 노드들의 키 값과 value를 지나온 경로 순서대로 출력한다. 마지막 줄에는 찾은 키 값의 value를 출력한다. 만약 그 키 값이 리프 노드에 없다면 트리에 없는 것이므로 not found를 출력한다.

- Revise(int del, int rep)

  Delete 과정에서 어떤 키를 지우면, 그 키 값을 non-leaf node에서 찾아서 지우고 다른 값으로 바꾸는 과정이 필요하다. 이 함수는 non-leaf-node에서 입력 받은 매개변수 del이 있다면, 이를 찾아 rep으로 바꾸는 역할을 한다.

  루트에서부터 내려오면서 각 노드에 del값이 있는지 검사한다. 만약 키 값이 있다면 rep으로 바꾼 다음, 함수를 즉시 종료한다. 만약 없다면 계속 찾아 내려가서 리프 노드까지 가면 함수가 종료된다.

- Non_leaf_merge

- 변수

- (left : merge하려는 노드의 왼쪽 노드)

  (right : merge하려는 노드의 오른쪽 노드)

  (where : merge하려는 노드, 이 함수의 매개변수로 받음)

- 설명

  딜리트 과정에서 non_leaf_node에서 언더 플로우가 발생하면, 그 노드의 형제와 merge를 해야 한다. 이 과정을 하는 함수이다. 먼저 언더 플로우가 발생한 노드의 왼쪽 , 오른쪽 형제를 찾는다. 이 때, 1.null이 아니고, 2.부모가 같은 형제와 merge를 한다. 만약 둘 다 null이 아닐 때는 왼쪽 형제와 merge한다.

  먼저, 부모 노드에서 언더 플로우가 난 노드 where과 merge할 노드(left or right) 사이에 있는 키를 삭제하고, 밑의 노드에 추가한다. 그리고 merge할 노드의 한쪽 키 값과 child들을 다른 한 쪽에 모두 넣어준다. 왼쪽과 merge하는 경우, 왼쪽에 키 값을 넣고, 오른쪽과 merge하는 경우에도 왼쪽에 있는 노드에 키 값을 넣는다. 이 과정에서 합치려는 두 노드의 child-parent관계도 다시 설정해준다. 그리고 합친 노드의 부모와 이와의 관계도 다시 설정해준다.

  이렇게 merge가 끝나고 난 후에, 합친 노드의 사이즈가 degree이상이라면 오버 플로우이기 때문에 스플릿을 해야 한다. 이때 delete과정이므로 non_leaf_split_delete(딜리트 버전)을 호출하여 스플릿 한다.

  그리고 합친 노드의 부모가 루트인데, 키 개수가 0이라면 합친 노드가 곧 루트 노드가 된다. 만약 부모가 루트 노드가 아닌데 언더 플로우가 난 경우에는 부모를 non_leaf_merge하도록 한다.

- Delete

- 변수

  (ltmp : 지우려는 키 값이 있는 노드)

  (fkey : ltmp에서 키를 지우기 전, 가장 앞에 있는 키를 미리 저장해둔다.)

  (left : findleft함수를 통해 찾은 ltmp의 왼쪽 노드)

- 설명

  먼저 지울 키 값이 있는 노드(ltmp)를 찾는다. 만약 키 값을 포함하는 노드에서 키 값을 지워도 언더 플로우가 발생하지 않는다면, 그냥 그 키를 지우고, revise를 호출하여 non-leaf node에서 그 키 값이 있다면 키를 지운다. 그리고 리프 노드의 가장 앞의 값(키를 지우고 난 뒤)으로 바꾼다. 여기서 revise는 지우는 키 값이 원래 가장 앞에 있었던 키일 때 호출된다.

  만약, 언더 플로우가 발생한다면 형제 노드에게서 빌릴 수 있는지 확인을 한다. 여기서 빌릴 노드는 부모가 같은 노드에게서만 빌리도록 한다. 키를 빌려오고 빌려준 노드에서 빌려준 키를 삭제한다.

  왼쪽에서 빌리는 경우, 새로 빌려온 키는 원래 노드의 키 값들보다 작을 것이다. 이 경우, 미리 저장해둔 ltmp의 원래 가장 앞에 있던 키(fkey)를 '삭제 후 ltmp의 가장 앞에 있는 값'으로 바꿔준다.(revise함수를 이용한다.)

  오른쪽에서 빌린 경우에는 , 오른쪽 노드의 가장 앞의 키를 ltmp에게 빌려주게 된다. 그러므로 키 값을 빌려주고 난 뒤, revise함수를 호출하여 non-leaf노드에서 빌려준 키를 찾아 그 값을 빌려주고 난 뒤 가장 앞의 값으로 바꿔주는 과정이 필요하다. 여기서 만약, ltmp에서 지웠던 키 값이 원래 노드의 제일 앞쪽 키였다면 non-leaf에서 그 값을 찾아 지우고 난 뒤 현재 ltmp에서 제일 앞의 키 값으로 바꿔주어야 한다.

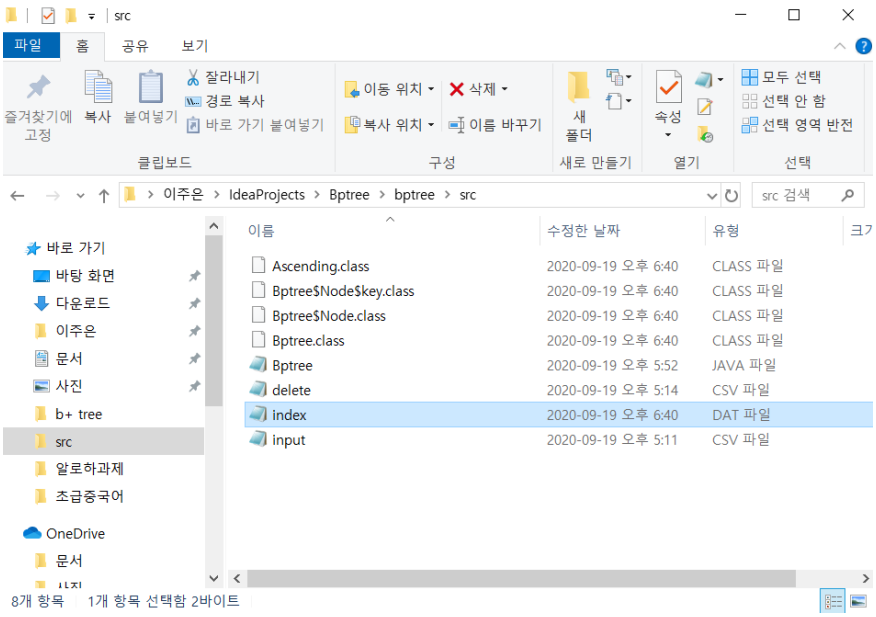
  만약 오른쪽, 왼쪽 둘 다로부터 키를 빌릴 수 없다면 merge를 해야 한다. 여기서도 마찬가지로 merge하는 두 노드는 부모가 같아야 한다. 왼쪽 노드와 merge하는 경우, 왼쪽 노드에 ltmp의 키 값들을 넣어주고, 오른쪽과 merge하는 경우는 오른쪽에 키 값들을 넣어준다. merge후 leaf-node 형제 관계와 부모 관계를 수정한다. merge하는 두 노드 사이에 있는 부모의 키 값은 부모 노드에서 없애 준다. 이 때, 오른쪽 노드와 merge하는 경우에서, 만약 현재 지우는 키 값이 ltmp의 가장 앞에 있는 값이었다면 revise를 호출하여 non-leaf에서 지우는 키 값을 찾아 merge후 합친 노드의 가장 앞의 값으로 바꿔준다.

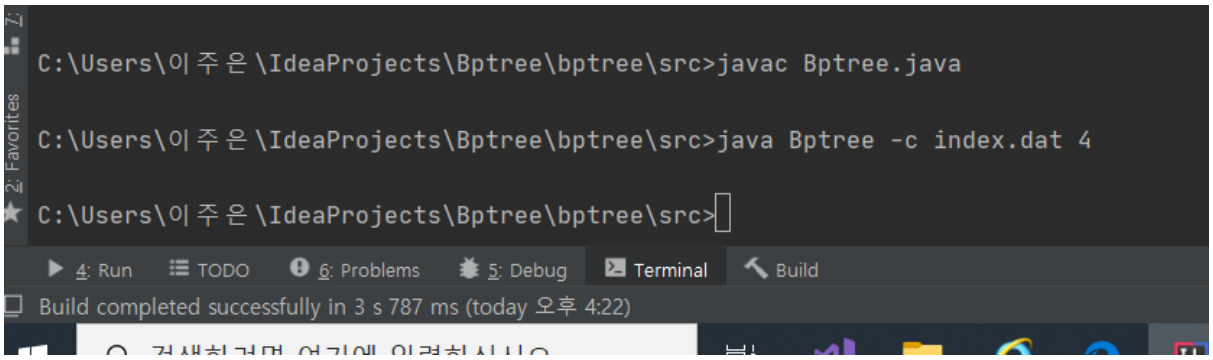
  Merge가 끝난 후, 만약 부모 노드가 루트 노드인데 키가 하나도 없다면 합친 노드가 곧 루트 노드가 될 것이다. 루트 노드가 아닌데 만약 언더 플로우가 발생한다면 , 부모 노드에서 non_leaf_merge를 해준다.

- Findleft

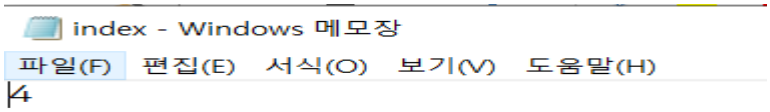
  특정 리프 노드의 왼쪽 노드를 리턴 해주는 함수이다. 삭제 과정에서, 만약 왼쪽 노드에서 빌리거나, 왼쪽 노드와 merge하는 경우에 이 함수를 이용하여 왼쪽 노드를 찾게 된다.

2. 실행하기 (instruction)

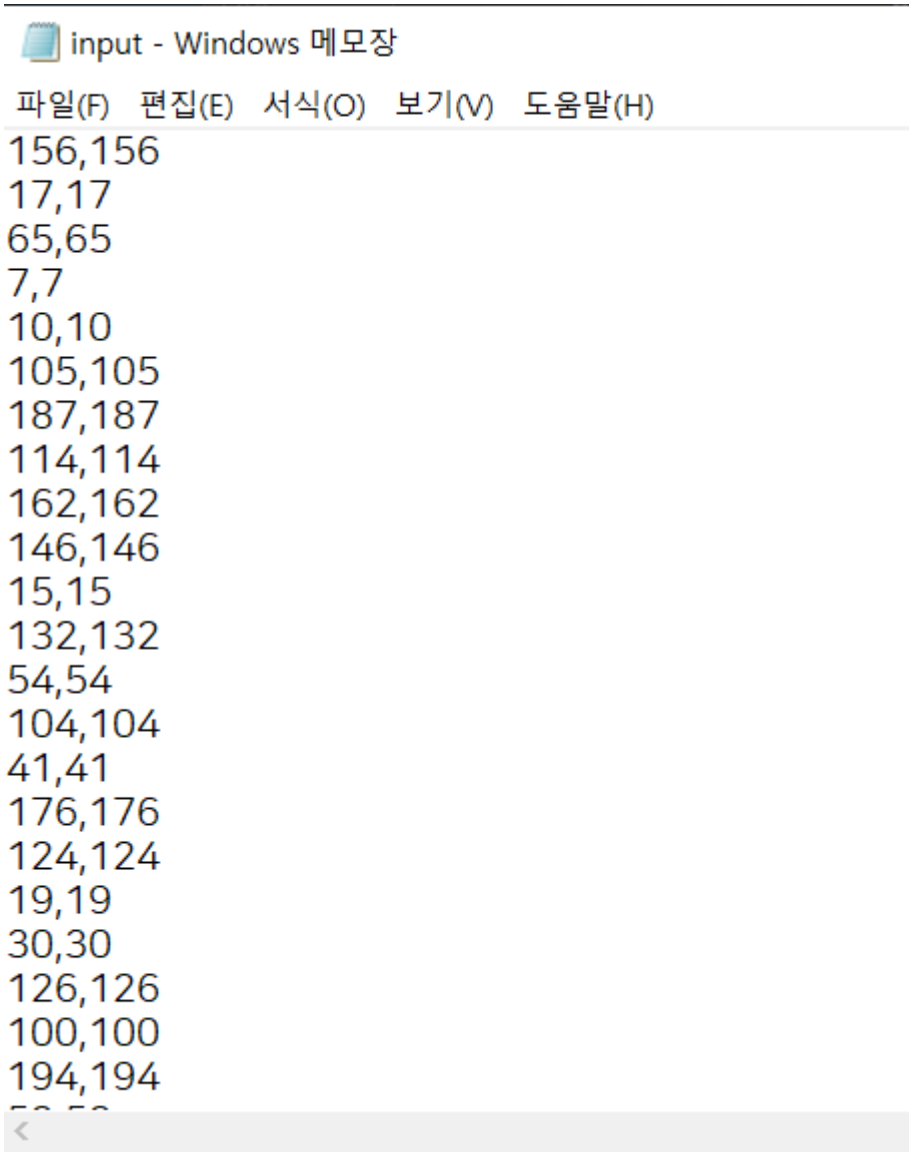




이렇게 먼저 -c를 해주고 , degree 4를 설정하면



Index.dat파일에 degree 4 가 쓰여집니다. (index.dat 연결프로그램은 메모장으로 했습니다.)



사진은 input.csv 파일입니다. (1부터 200까지의 수가 랜덤하게 배열되어 있는 input 예시입니다.)

```
C:\Users\이주은\IdeaProjects\Bptree\bptree\src>javac Bptree.java

C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -c index.dat 4

C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -i index.dat input.csv
```

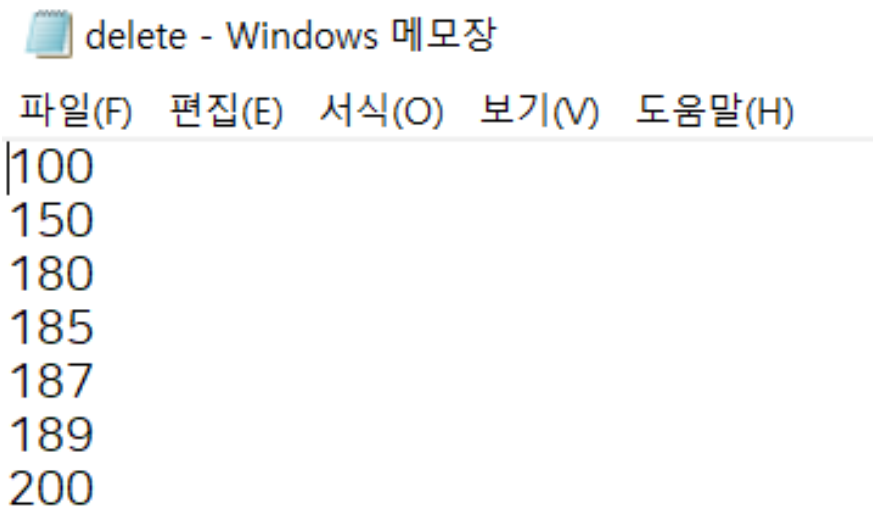
인풋 파일에 있는 값들을 트리에 insert합니다.

Index.dat파일에 이렇게 key , value를 저장합니다.

```
C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -s index.dat 300
NOT FOUND

C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -s index.dat 150
89 124
151 162 185
130 134 143
146 148
150
```

1-200의 숫자가 들어가 있으므로 없는 숫자인 300은 search 후 NOT FOUND를 출력합니다. 150의 경우 트리에 존재하므로 위에서부터 내려가고 경로에 있는 노드의 키를 출력합니다. 마지막 줄에는 150의 value인 150을 출력합니다.



Delete.csv 파일입니다. 지울 키 값들이 들어있습니다.

```
C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -d index.dat delete.csv

C:\Users\이주은\IdeaProjects\Bptree\bptree\src>java Bptree -r index.dat 180 200
181 , 181
182 , 182
183 , 183
184 , 184
186 , 186
188 , 188
190 , 190
191 , 191
192 , 192
193 , 193
194 , 194
195 , 195
196 , 196
197 , 197
198 , 198
199 , 199
```

Delete를 실행한 후 range_search를 했을 때 지운 키 값들을 제외한 나머지 범위 내 key와 value를 한 줄씩 출력하는 것을 볼 수 있습니다.

3. 데이터 저장 및 불러오는 방법

일단 java Bptree -c index.dat (number) 이 입력되면, index.dat파일을 만들어 제일 위에 degree를 쓰고 파일을 닫는다. -i,-d를 할 때는 먼저 FileReader를 통해 input.csv ,delete.csv를 읽어 내려가면서 삽입 혹은 삭제하려는 키를 index.dat파일에 써 내려간다. 이 과정이 index.dat에 트리의 정보를 저장하는 과정이다. 삽입의 경우 (key ,value) 로, 삭제의 경우 (key) 이렇게 기록한다. 그 다음, index.dat파일을 위에서부터 읽어 내려오면서 키 값을 삽입하거나 삭제하여 트리를 만든다. (key,value 이렇게 저장 되어 있으면 삽입, key값만 있으면 delete) Single/range search는 이렇게 index.dat를 이용해 트리 정보를 가지고 온 다음, 각각 single_search, range_search함수를 호출하여 그 결과값을 출력하도록 한다.