**데이터베이스 시스템 및 응용**

**B+tree 구현 과제 코드 해설**

**2019055078 신채영**

IntelliJ Community 2020에서 코드를 작성했고 한글 인코딩이 안 되는 관계로 자세한 코드 한 줄 한 줄에 대한 설명은 주석 형식으로 달았습니다. 그 외의 추가 설명은 코드와 클래스, 함수의 용도와 실행 방법, 실행 예시로 대신하겠습니다. (보고서의 코드와 제출 코드의 순서가 조금 다를 수 있습니다. 또 코드 전체를 넣어 설명하기 때문에 해설문서가 조금 긴 것 양해부탁드립니다.)

**클래스:**

1. pair 클래스

public class pair {  
 int key;  
 int val;  
 Node child; //left child node  
 public pair(int key, int val, Node child) {  
 super();  
 this.key = key;  
 this.val = val;  
 this.child = child;  
 }  
}

node 안에 들어가는 <key, value>의 페어입니다. 추가로 자식 노드를 가지게 했습니다.

1. Node 클래스

public class Node {  
 int m; //#of keys  
 List<pair> p; //array of pairs  
 Node r;//right-sibling node or right-most child node  
 Node parent;  
 public Node(){  
 super();  
 this.m=0;  
 this.p=new ArrayList<>();  
 this.r=null;  
 this.parent=null;  
 }  
}

멤버로 m(가진 페어의 개수), p(가진 페어들로 이뤄진 리스트), r(오른쪽 형제노드/오른쪽 자식), parent(부모 노드)를 가집니다.

1. Command 클래스

public class commands {  
 int key;  
 int val;  
 char com; //'i' or 'd'  
 public commands(int key,int val,char com){  
 this.key=key;  
 this.val=val;  
 this.com=com;  
 }  
}

트리를 저장할 때, 그리고 불러올 때 필요한 클래스로 명령(I,d)와 key, val를 멤버로 가집니다. (delete의 경우 value를 0으로 가지게 했습니다)

1. bptree 클래스

public class bptree {  
 static Node *root* = new Node();  
 static int *nodeSize* = 0; //node size  
  
 static List<commands> *commands* = new ArrayList<>();

메인 클래스가 들어있는 클래스로 트리의 루트와 차수, 그리고 커맨드를 저장할 리스트를 가집니다. 모든 함수에서의 접근을 위해 모두 static으로 선언합니다.

**함수:**

* **메인함수**

//////main//////  
public static void main(String[] args) throws Exception {  
 String command = args[0]; //read command (c,i,d,s,r)  
 String savedFile = args[1];//read file to save the tree info  
 //create  
 if (command.equals("-c")) {  
 *nodeSize* = Integer.*parseInt*(args[2]);  
 *writeNodeSize*(savedFile);//write the nodesize of the tree. ex) nodeSize:3  
 } else {  
 *readSavedTree*(savedFile); //read the commands given previously  
 //insert  
 if (command.equals("-i")) {  
 *readInsertion*(args[2]);  
 *saveTree*(savedFile);  
 }  
 //single key search  
 else if (command.equals("-s")) {  
 int search = Integer.*parseInt*(args[2]);  
 *single\_search*(search);  
 }  
 //ranged search  
 else if (command.equals("-r")) {  
 int start = Integer.*parseInt*(args[2]);  
 int end = Integer.*parseInt*(args[3]);  
 *ranged\_search*(start, end);  
 }  
 //delete  
 else if (command.equals("-d")) {  
 *readDeletion*(args[2]);  
 *saveTree*(savedFile);  
 }  
 //saveTree when there is modification (insert & delete)  
 }  
}

커맨드 라인에서 실행하여 “java bptree+~”를 입력하여 뒤에 붙는 명령 (c,I,s,r,d)에 따라 다음 실행할 코드를 찾아가는 함수입니다.

처음 트리를 생성하는 -c를 제외한 다른 명령들은 모두 먼저 index.dat파일에서 저장된 트리의 정보를 읽어와야 합니다. (readSavedTree)

또 트리의 정보를 수정하는 i(insert)와 d(delete)는 삽입 또는 삭제를 한 후에 index.dat파일에 업데이트를 하기 위해 saveTree를 해줍니다.

* **readInsertion:**

////////FileRW////////  
//read key and value to insert and do the insertion  
public static void readInsertion(String file) throws Exception {  
 File insert = new File(file);  
 String line;  
 try {  
 BufferedReader b = new BufferedReader(new FileReader(insert));  
 while ((line = b.readLine()) != null) {//read till the end  
 String[] keyAndVal = line.split(","); //read csv(comma separated) file  
 int key = Integer.*parseInt*(keyAndVal[0]);  
 int val = Integer.*parseInt*(keyAndVal[1]);  
 *insert*(key, val);//insert  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}

삽입할 키를 읽어오는 함수입니다. 버퍼리더를 이용해 삽입할 페어의 key와 value를 읽어오고 insert함수를 실행합니다.

//read key to delete and do the deletion  
public static void readDeletion(String file) throws Exception {  
 File delete = new File(file);  
 String line;  
 try {  
 BufferedReader b = new BufferedReader(new FileReader(delete));  
 while ((line = b.readLine()) != null) { //read line till the end  
 String key = line;  
 *delete*(Integer.*parseInt*(key));//delete  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}

같은 방법으로 delete를 읽어오는 함수입니다.

* **writeNodeSize:**

//write the node size of the tree on the top of the index.dat file  
public static void writeNodeSize(String File) {  
 try {  
 FileWriter fw = new FileWriter(File, false);  
 fw.write("nodeSize: " + *nodeSize*);  
 fw.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

Index.dat파일에 트리를 저장할 때 맨 윗줄에 트리의 차수(nodesize)를 입력해주는 함수입니다.

* **writeNodes:**

//save commands info  
public static void WriteNodes(FileWriter fw) {  
 try {  
 for (commands command : *commands*) { //all the commands saved  
 if (command.com == 'i')  
 fw.write(command.com + ". key: " + command.key + ", value: " + command.val + "\n");  
 else  
 fw.write(command.com + ". key: " + command.key + "\n");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

Command 리스트에 저장된 명령들을 index.dat에 다시 write하는 함수입니다

Ex) <1,10>을 삽입했었다면 I. key:1, value:10

* **saveTree**

//save tree info in the file  
public static void saveTree(String file) {  
 try {  
 FileWriter fw = new FileWriter(file, false); //clean-and-write everytime  
 fw.write("nodeSize: " + *nodeSize* + "\n");//size of node  
 *WriteNodes*(fw);//write the commands made  
 //ex) insert <1,9> -> i. key: 1, value: 9  
 fw.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

삽입이나 삭제를 했을 경우 index.dat파일에 트리의 상태를 업데이트해주는 함수입니다. 매번 지우고 다시 입력하는 방식으로 filewriter(file,false)를 사용했습니다. 첫 줄에 차수를 입력하고 두번째 줄부터는 writeNode를 이용해 삽입/삭제 정보를 입력합니다.

* **readSavedTree**

//read the commands given previously and make a new tree according to the history  
public static void readSavedTree(String file) {  
 File savedFile = new File(file);  
 Node n = new Node();  
 String read;  
 try {  
 BufferedReader br = new BufferedReader(new FileReader(savedFile));  
 read = br.readLine();  
 read = read.substring(10);  
 *nodeSize* = Integer.*parseInt*(read); //read the size of node  
 Node now = null;  
 while ((read = br.readLine()) != null) {  
 if (read.substring(0, 1).equals("i")) { //insert  
 String[] keyAndVal = read.substring(8).split(", value: "); //the index of int value==8  
 int key = Integer.*parseInt*(keyAndVal[0]); //ex) i. key: 1, value: 9  
 int val = Integer.*parseInt*(keyAndVal[1]);  
 *insert*(key, val);  
  
 } else {//delete  
 String keyString = read.substring(8);  
 int key = Integer.*parseInt*(keyString);//ex) d. key: 7  
 *delete*(key);  
 }  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
}

트리 생성을 제외한 다른 명령을 수행하기 전에 index.dat에 저장된 트리를 가져오는 함수입니다. substring과 split함수를 이용해 명령(i/d), key, value 세가지를 읽어와 트리를 다시 구성할 수 있습니다. 파일의 끝까지 읽었다면 종료합니다.

* **WhichNodeToInsert:**

///////insert/////////  
//look for the node which the pair should be inserted in  
public static Node WhichNodeToInsert(int key) {  
 Node now = *root*; //search from root  
 while (true) {  
 //if now has no child-nodes return now(leaf)  
 if (now.m == 0)  
 return now;  
 if (now.p.get(0).child == null)  
 return now;  
  
 //check if now has to go down (to the child-node of it)  
 boolean biggest = true;  
 for (int i = 0; i < now.m; i++) {  
 if (key < now.p.get(i).key) { //once the key is smaller than the pair in node,  
 now = now.p.get(i).child;//the index to inserted is found  
 biggest = false;  
 break;  
 }  
 }  
 //if all the pairs in the node are bigger than key -> check the rightmost child (continue)  
 if (biggest==true){  
 now = now.r;  
 }  
 }  
}

삽입을 할 때 삽입할 키가 트리의 어느 노드에 들어가야 하는지 찾는 함수입니다. leaf노드에 도달할 때까지 (자식이 없을 때까지) key가 들어갈 노드를 찾아 내려갑니다. (key보다 큰 노드가 나타나면 그 노드의 자식 노드로 가서 계속 탐색하는 방식)

* **findInsertIndex:**

//returns the index where the key should be inserted  
public static int findInsertIndex(Node node, int key) {  
 for (int i = 0; i < node.m; i++) { //from the start of node pairs to the end  
 if (key < node.p.get(i).key) //once the key is smaller than the key in node  
 return i; //the index is found  
 }  
 return node.m;  
}

위에서 삽입할 노드를 찾았다면 그 노드에서 어느 인덱스에 삽입해야 할지 찾는 함수입니다. 노드의 m만큼 for 문을 돌면서 key보다 큰 첫번째 pair의 인덱스를 출력합니다. 모든 페어가 key보다 작다면 now.m을 출력합니다.

* **leafSplit:**

//solve leaf overflow problems by splitting  
public static void leafSplit(Node now, Node r) {  
 //move the right half of 'now' to its right sibling  
 for (int i = *nodeSize* / 2; i < *nodeSize*; i++) {  
 r.p.add(new pair(now.p.get(i).key, now.p.get(i).val, null));  
 r.m++;  
 }  
 //then remove the right half in 'now'  
 for (int i = *nodeSize* / 2; i < *nodeSize*; i++) {  
 now.p.remove(*nodeSize* / 2);  
 now.m--;  
 }  
}

리프노드에서 오버플로우가 일어날 경우 스플릿해주는 함수입니다. 노드의 오른쪽 절반을 새로운

노드에 저장해줍니다. (매개변수로 받은 두번째 노드)

그리고 now노드의 오른쪽 절반을 삭제합니다.

* **NonLeafSplit:**

//solve non-leaf(parent) overflow problems by splitting  
public static void nonLeafSplit(Node now) {  
 Node split = new Node();//node to place the split part  
 //save the middle pair for later use  
 pair newRootPair = new pair(now.p.get(now.m / 2).key, now.p.get(now.m / 2).val, now.p.get(now.m / 2).child);  
  
 int M = now.m;  
 //place pairs in the right half of the node in 'split'  
 for (int i = M / 2 + 1; i < M; i++) {  
 split.p.add(new pair(now.p.get(i).key, now.p.get(i).val, now.p.get(i).child));  
 now.p.get(i).child.parent = split;  
 split.m++;  
 }  
 //remove the pairs in 'now'  
 for (int i = M / 2; i < M; i++) {  
 now.m--;  
 now.p.remove(M / 2);  
 }  
 split.r = now.r;//move the rightmost child too  
 now.r = newRootPair.child;//the middle pair is now the child on the split bound  
 split.r.parent = split;//link parent nodes and child nodes  
 now.r.parent = now;  
 //the split node is not the root of the tree  
 if (now != *root*) {  
 int nodeIndex = *IndexInParent*(now);  
 now.parent.p.add(nodeIndex, new pair(newRootPair.key, newRootPair.val, now));  
 now.parent.m++;  
 //if the split node is not the leftmost child of it parent node  
 if(nodeIndex<now.parent.m-1)  
 now.parent.p.get(nodeIndex + 1).child = split;//the child node next to it is split  
 //if it is the leftmost child  
 else  
 now.parent.r = split;//split goes down to the right child of split node  
 split.parent = now.parent;//they share the same parent  
 //if parent overflow occurs because of splitting  
 if (now.parent.m >= *nodeSize*) {  
 *nonLeafSplit*(now.parent);//call split function again  
 }  
 }  
 //the split node is the root of the tree  
 else {  
 //make a new root for the tree  
 Node newRoot = new Node();  
 //the saved pair (the middle pair) is the new root  
 pair rootPair=new pair(newRootPair.key, newRootPair.val, now);  
  
 newRoot.p.add(rootPair);  
 newRoot.m=1;  
 newRoot.r = split; //the children of new root : now(left) and split(right)  
 *root* = newRoot;//new root  
 now.parent = *root*;//link root and its child nodes  
 split.parent = *root*;  
  
 }  
}

논리프,부모 노드에서 오버플로우가 일어났을 경우 스플릿해주는 함수입니다.

먼저 중간 페어를 저장해둡니다

Now의 오른쪽 절반을 새로운 노드split로 옮겨줍니다. Now의 오른쪽 자식을 split로 넘겨주고 now의 새로운 오른쪽 자식으로는 저장해둔 중간 페어의 자식을 줍니다 (중간 페어가 노드를 반으로 자르는 듯이)

1. Now가 루트가 아닌 경우

Now가 맨 오른쪽 노드면 저장해둔 페어를 now부모 노드에 추가하고 그 페어의 오른쪽 자식으로 split으로 줍니다.

맨 오른쪽 노드가 아니라면 now의 부모 페어 자리에 중간 페어를 넣습니다. 그리고 원래 있던 now의 부모 페어는 split을 자식으로 갖습니다.

1. Now가 루트인 경우

왼쪽 자식이 now, 오른쪽 자식이 split인 새로운 부모 노드를 만듭니다. 이 부모 노드가 이제 이 트리의 루트입니다.

* **Insert:**

//insert pair at the proper place (key and val)  
public static void insert(int key, int val) {  
 //save the command in the array ex) i. key: 1, value: 9  
 *commands*.add(new commands(key, val, 'i'));  
 pair insertP = new pair(key, val, null);//pair to insert  
  
 Node now = *WhichNodeToInsert*(key);//find the node which the key should be inserted in  
 int insertIndex = *findInsertIndex*(now, key);//and the index  
 now.p.add(insertIndex, insertP);//add it at the proper place  
  
 if (++now.m >= *nodeSize*) { //needs splitting  
 Node split = new Node(); //make a node to save the split part  
 *leafSplit*(now, split); //split(leaf node)  
  
 if (now.parent != null) {//not root  
 int nodeIndex = *findInsertIndex*(now.parent, key);  
 if (nodeIndex == now.parent.m) {//rightmost child  
 now.parent.p.add(new pair(split.p.get(0).key, split.p.get(0).val, now)); //add the first pair of split to now.parent  
 now.parent.r = split; //split is the rightmost of now.parent  
 now.parent.m++;  
 split.r = now.r;//the right sibling of now moves to the right of split (split goes between now and now.r)  
 now.r = split; //split is in the right to now  
 split.parent = now.parent;//share the same parent  
 }  
 else {  
 pair newPair = new pair(split.p.get(0).key, split.p.get(0).val, now);  
 now.parent.p.add(nodeIndex, newPair); //add the first pair of split to where the parent of now was  
 now.parent.p.get(nodeIndex + 1).child = split; // child node of origin pair now.parent.get(nodeIndex) == split  
 now.parent.m++;  
 split.r = now.r; //same as the process done when key was in the rightmost child  
 now.r = split;  
 split.parent = now.parent;  
 }  
 if (now.parent.m >= *nodeSize*) //parent overflow  
 *nonLeafSplit*(now.parent);  
 }  
 else { //now ==root  
 //new parent for now and split  
 Node newParent = new Node();  
 pair p = new pair(split.p.get(0).key, split.p.get(0).val,null);  
 p.child=now;//left child is now. right child is split  
 newParent.p.add(p);  
 newParent.r = split;  
 newParent.m = 1;  
  
 *root* = newParent;//new root  
 now.parent = newParent;//share the same parent  
 split.parent = newParent;  
 split.r = now.r; //split goes between now and now.r  
 now.r = split;  
 }  
 }  
}

Key 와 value를 트리에 삽입하는 함수입니다. 먼저 명령 리스트에 명령을 기록합니다. 그리고 만들어 둔 삽입할 자리를 찾는 함수들을 이용해 leaf노드에 페어를 삽입해줍니다.

leaf노드에서 오버플로우가 일어난 경우 먼저 leaf 스플릿 작업을 해줍니다. 그 후에

1. Now가 루트가 아닌 경우

맨 오른쪽 노드라면 split을 부모의 마지막 자식으로 주고

아니라면 now의 부모 페어의 자리에 split의 첫번째 페어를 넣고 원래 부모 페어의 자식으로 split을 줍니다. Now의 오른쪽이던 노드를 split의 오른쪽 노드로 줍니다. 그리고 now의 오른쪽은 split입니다. 마지막으로 now와 split이 같은 부모를 갖게 합니다.

이 과정에서 부모 노드에 오버플로우가 일어난다면 nonLeafSplit을 call합니다.

1. Now가 루트라면

Now가 왼쪽 자식, split가 오른쪽 자식인 부모 노드를 새로 만듭니다. 그리고 이 노드가 이 트리의 새로운 루트입니다.

* **IndexInParent**

//return the index of the parent pair of 'now'  
public static int IndexInParent(Node now) {  
 if (now.parent != null) { //not root  
 for (int i = 0; i < now.parent.m; i++) {//return index if now == child of i th pair of the parent node  
 if (now == now.parent.p.get(i).child) {  
 return i;  
 }  
 }  
 if (now.parent.r == now) { //found the parent pair in the rightmost node  
 return now.parent.m;  
 }  
 }  
 return -1;//roots don't have parent nodes.  
}

부모 노드의 몇 번째 페어의 자식인지 찾는 함수입니다. 루트라면 -1을 출력하고 now를 자식으로 갖는 pair가 나왔을 때 그 인덱스를 출력합니다.

* **Single\_search:**

////////single key search///////  
public static void single\_search(int key) {  
 Node now;  
 List<Node> path = new ArrayList<>();//save the inverse path of single key search  
 now = *root*;//search from the root  
  
 //if there is no nodes in the tree, you don't need to look for it (return)  
 if(now.m==0){  
 System.*out*.println("Nothing in the tree!");  
 return;  
 }  
  
 int i;  
 path.add(*root*); //the root is always on the path  
 boolean pairIsFound = false;  
 //search till the node is leaf node  
 while (now.p.get(0).child != null) {  
 pairIsFound = false;//find the pair bigger than the key you are searching  
 for (i = 0; i < now.m; i++) {  
 if (key < now.p.get(i).key) { //once the pair bigger than the key is found  
 now = now.p.get(i).child; //continue searching from the child of the node  
 pairIsFound = true;  
 break;  
 }  
 }  
  
 if (pairIsFound == false) { //came to the end but did not find the pair bigger than the key  
 now = now.r;//move on to the right child of the key  
 }  
 path.add(now);//add path  
 }  
 int value = 0;  
 boolean keyIsFound = false;//look for the key  
 int j;  
 //search the pair that has the key as its key (in the node)  
 for (j = 0; j < now.m; j++) {  
 if (key == now.p.get(j).key) { //if the key is found  
 value = now.p.get(j).val; //save its value to display on console  
 for (Node node : path) {  
 *printSingleSearchPath*(node);  
 } //and print the path  
 keyIsFound = true;  
 }  
 }  
 if (!keyIsFound) { //if the key is not found  
 System.*out*.println("Not found!"); //print "not found"  
 } else {  
 System.*out*.println("\nPair Found:\nkey: " + key + " value: " + value); // print value of the key  
 }  
}

Key하나를 탐색하고 탐색 경로를 출력하는 함수입니다. 루트부터 시작해 만들어 둔 노드와 인덱스를 찾는 함수들을 이용해 key를 찾습니다. 그 과정에서 지나가는 모든 노드들의 모든 페어의 키를 리스트에 저장합니다. 키를 찾았다면 그 리스트를 거꾸로 출력하면 탐색경로가 나옵니다. 키를 못 찾았다면 not found를 출력합니다.

//print the path (single key search)  
public static void printSingleSearchPath(Node n) {  
 //print all the keys in the list in inverse order  
 for (int i = 0; i < n.m; i++) {  
 System.*out*.print(n.p.get(i).key);  
 if (i < n.m - 1)  
 System.*out*.print(", "); // add comma if it is not the last one  
 }  
 System.*out*.println(); //enter  
}

탐색 경로에 있는 노드를 변수로 받고 그 노드에 있는 모든 페어의 키를 출력하는 함수입니다.

* **rangedSearch**

////////single key search///////  
public static void ranged\_search(int start, int end) {  
 /\*first, find the starting node  
 same as the way we search for a single key.  
 -> single\_search(start) but don't have to save the path  
 \*/  
 Node now;  
 now = *root*;  
 boolean isFound = false;  
  
 // if there is nothing in the tree, don't have to search for keys  
 if(now.m==0){  
 System.*out*.println("Nothing in the tree!");  
 return;  
 }  
  
 //same as single key search  
 while (now.p.get(0).child != null) {  
 isFound = false;  
 for (int i = 0; i < now.m; i++) {  
 if (start < now.p.get(i).key) {  
 now = now.p.get(i).child;  
 isFound = true;  
 break;  
 }  
 }  
 if (isFound == false) { //came to the end but not found  
 now = now.r;  
 }  
 }  
 //print the leaf keys on the right side of the found key  
 //until the right sibling is bigger than end  
 while (true) {  
 if(now.m==0) {  
 System.*out*.println(now.r.p.get(0).key); continue;}  
 if(now.p.get(0).key > end)  
 break;  
 *printRangedSearch*(now, start, end);//print leaf keys  
 if (now.r == null) //is the end of the tree (the biggest)  
 break;  
 now = now.r;//move on to its right sibling  
 }  
}

singleSearch와 같은 방법으로 start를 찾고 start보다 크거나 같은 노드부터 시작해 end보다 큰 키가 나올 때까지 모든 leaf 노드의 키들을 출력하면 범위 탐색이 됩니다.

* **printRangedSearch**

//print all the keys between start and end  
public static void printRangedSearch(Node n, int start, int end) {  
 //in the node, both keys smaller than end and keys bigger than end can be there.  
 for (int i = 0; i < n.m; i++) {  
 if (n.p.get(i).key > end) { //if the key is bigger than end, stop  
 return;  
 }  
 // if it is between start and end, print its key and value  
 if (n.p.get(i).key <= end && n.p.get(i).key >= start) {  
 System.*out*.println(n.p.get(i).key + ", " + n.p.get(i).val);  
 }  
 }  
}

시작하는 노드를 첫번째 변수로 받고 그 노드 안에서 start보다 크고 end보다 작은 key를 출력하는 함수입니다. Key가 end보다 커지면 종료합니다.

* **Left&right siblings**

//find its left sibling  
public static Node leftSibling(Node now) {  
 int NodeIndex = *IndexInParent*(now);  
 if (now.parent==null||NodeIndex == 0)//now==leftmost node  
 return null;  
 else  
 return now.parent.p.get(NodeIndex - 1).child;  
}  
  
//find its right sibling  
public static Node rightSibling(Node now) {  
 int NodeIndex = *IndexInParent*(now);  
 if (now.parent==null||NodeIndex == now.parent.m) //now==rightmost node  
 return null;  
  
 if (NodeIndex == 0) {//now==leftmost child  
 if (now.parent.m == 1) // sib is a rightmost child  
 return now.parent.r;  
 return now.parent.p.get(NodeIndex + 1).child;  
 }  
 if (now.parent.m == NodeIndex + 1) // sib is a rightmost child  
 return now.parent.r;  
 return now.parent.p.get(NodeIndex + 1).child;  
  
}

왼쪽 형제 노드, 오른쪽 형제 노드를 출력하는 두개의 함수입니다. (parent를 통해 접근합니다)\

* **Delete:**

public static void delete(int key){  
 Node now;  
 //now=WhichNodeToInsert(key);  
 now=*root*;  
 int index;  
  
 //find node where the key is & its index  
 while (true) {  
 index = now.m;  
 for (int i = 0; i < now.m; i++) {  
 if (key < now.p.get(i).key) {  
 index = i;  
 break;  
 }  
 }  
  
 if (now.m == 0 || now.p.get(0).child == null)//has nothing or is leaf  
 break;  
 if (index == now.m)//is the rightmost node -> move on to its right child  
 now = now.r;  
 else //else move on to its left child  
 now = now.p.get(index).child;  
  
 }  
 int indexInNode= *findIndex*(now,key);  
 Node lefts=*leftSibling*(now);  
 int parentKey = now.p.get(0).key; //node that can be swapped to be the parent if needed later  
  
 if(indexInNode!=-1) { //if key is found in the tree  
 now.p.remove(indexInNode); //delete  
 now.m--;  
 *commands*.add(new commands(key,0,'d'));  
 }  
 else{  
 System.*out*.println(key+" is not in the tree.");  
 }  
  
 if(now.m<(*nodeSize*-1)/2) { //underflow  
 Node right = *rightSibling*(now);  
 Node left = *leftSibling*(now); //save left and right nodes  
  
 //borrow from left  
 if (left != null && left.m > (*nodeSize* - 1) / 2) {  
 int lastIndex=left.m-1;  
 pair p= new pair(left.p.get(lastIndex).key,left.p.get(lastIndex).val,null);  
 now.p.add(0,p);  
 now.m++; //add the last pair of 'left' to now (in the front)  
 left.p.remove(left.m - 1);  
 left.m--; //remove the pair from 'left'  
 *deleteIfNonLeaf*(parentKey, now.p.get(0).key); //check non-leaf node  
 }  
 //borrow from right  
 else if (right != null && right.m > (*nodeSize* - 1) / 2) {  
 pair p=new pair(right.p.get(0).key,right.p.get(0).val,null);  
 now.p.add(p);  
 now.m++;//add the first pair of 'right' to now (at the end)  
 right.p.remove(0);  
 right.m--;  
 *deleteIfNonLeaf*(now.p.get(now.m - 1).key, right.p.get(0).key);//check non-leaf node  
  
 if (indexInNode == 0)//if now == the first child of its parent  
 *deleteIfNonLeaf*(parentKey, now.p.get(0).key);  
  
 } else {//cannot borrow pairs from siblings  
  
 if (now.parent != null) {//has parent Node  
  
 int indexinP = *IndexInParent*(now);  
  
 if (left != null) { //has left-sibling  
  
 for (int i = 0; i < now.m; i++) { //move all pairs and child of 'now' to 'left'  
 pair move = new pair(now.p.get(i).key, now.p.get(i).val, null);  
 left.m++;  
 }  
 left.r = now.r; //move the last child(r) to left  
  
 //put left where now were before  
 if (indexinP == now.parent.m) //if now=rightmost child of its parent node  
 now.parent.r = left;  
 else  
 now.parent.p.get(indexinP).child = left;  
  
 //remove pair on the left side of the parent of 'now'  
 now.parent.p.remove(indexinP - 1);  
 now.parent.m--;  
  
 if(now.parent!=null){  
 if (now.parent.m >= (*nodeSize* - 1) / 2)//has a parent node && no underflow  
 return;  
 else  
 *parentUnderflow\_Merge*(now.parent); //has a parent node && underflow -> merge  
 }  
 else if (now.parent==null&&now.parent.m==0){ //now.parent==root&&root is empty -> 'left' is the new root  
 *root* = left;  
 *root*.parent = null;  
 }  
 else{  
 return; //good  
 }  
 }  
 else { //no left siblings  
 //move pairs in 'now' to 'right'(in the front)  
 for(int i=0;i<now.m;i++){  
  
 right.p.add(0,new pair(now.p.get(now.m-i-1).key,now.p.get(now.m-i-1).val,null));  
 right.m++;  
 }  
  
 if (indexInNode == 0) //leftmost node  
 *deleteIfNonLeaf*(key, right.p.get(0).key);  
  
 now.parent.p.remove(indexinP); //remove child node with the deleted key  
 now.parent.m--;  
  
 if (now.parent == *root*) {  
 if(*root*.m==0){ //root is empty  
 *root* = right; //right is the new root  
 *root*.parent = null; //roots don't have parents  
 }  
 else  
 return; //good  
  
 } else {  
 if (now.parent.m < (*nodeSize* - 1) / 2)//parent underflow  
 *parentUnderflow\_Merge*(now.parent);  
  
 }  
  
 }  
  
 }  
 }  
 }  
 else { //no underflows -> now.m>=(nodeSize-1)/2  
 *deleteIfNonLeaf*(parentKey, now.p.get(0).key); //check if key is in a non\_leaf node and end deletion  
 return;  
 }  
  
}

키를 받아 트리에서 삭제하는 함수입니다. 코드가 복잡해 특히나 주석에 많은 설명을 달았습니다. 여기서는 간단히 삭제 방법 설명을 하겠습니다.

먼저 Key가 있는 노드를 찾아 삭제를 해줍니다. 만약 이 key가 leaf node가 아니라면 가장 작은 key를 올립니다.

Underflow가 생겼다면 형제 노드들을 확인합니다.

(차수-1)/2보다 형제 노드의 m이 더 크다면 형제 페어에서 하나를 가져옵니다.

빌려올 수 없다면 삭제된 페어가 있던 노드의 남은 페어들을 전부 다 형제 노드에 넣습니다.

이 과정을 모든 노드에 underflow가 일어나지 않을 때까지 반복합니다

* **deleteIfNonLeaf**

//delete/change key if the key is in a non-leaf Node  
public static void deleteIfNonLeaf(int delete, int newKey) {  
 Node now = *root*;//look for 'delete' in a non-leaf node  
 while (true) {  
 boolean foundParent = false;//found parent node  
 if (now.m == 0 || now.p.get(0).child == null)  
 break;  
  
 for (int i = 0; i < now.m; i++) {  
  
 if (delete < now.p.get(i).key) {  
 now = now.p.get(i).child; //delete is child of new now  
 foundParent = true; //found parent  
 break;  
 }  
 if (delete== now.p.get(i).key) {  
 now.p.get(i).key = newKey; //delete is a nonleaf node  
 return;  
  
 }  
 }  
 if (foundParent==false)  
 now = now.r;//check rightmost child of 'now'(while loop)  
 }  
}

지울 key가 부모 노드에도 있을 경우 키를 바꿔주는 함수입니다. (가장 작은 키)

//solve parent underflow problems (by combining with sibling nodes)  
public static void parentUnderflow\_Merge(Node now) {  
 int indexinP = *IndexInParent*(now);// now is idx.th child of now.parent  
  
 if (now.parent != null) {  
 Node leftSib = *leftSibling*(now);//left sib  
 Node rightSib = *rightSibling*(now);//right sib  
  
 if (rightSib != null) {  
 //add the pair with 'now' as child -> to now  
 pair p = new pair(now.parent.p.get(indexinP).key, now.parent.p.get(indexinP).val, now.r); // and its new child is now.r  
 now.p.add(p);  
 now.m++;  
 //remove the pair with the key  
 now.parent.p.remove(indexinP);  
 now.parent.m--;  
  
 //add all pairs in right to 'now'  
 for (int i = 0; i < rightSib.m; i++) {  
 now.p.add(rightSib.p.get(i));  
 now.m++;  
 rightSib.p.get(i).child.parent = now;  
 }  
 now.r = rightSib.r; // move rightmost child to rightmost child of 'now'  
 now.r.parent = now;  
  
 //link the children and parents  
 if (rightSib.parent.m > 0) {  
 if (rightSib.parent.m != indexinP) {  
 rightSib.parent.p.get(indexinP).child = now;  
 }  
 }  
 else  
 rightSib.parent.r = now;  
 if (now.m >= *nodeSize*) //parent-overflow occurred after merging  
 *nonLeafSplit*(now); //then split parents  
  
 //check parent node  
 if (rightSib.parent != *root*) {  
 if (rightSib.parent.m < (*nodeSize* - 1) / 2) { //underflow occurred in rightSib.parent  
 *parentUnderflow\_Merge*(rightSib.parent); //call merge again  
 }  
 } else {  
 if (rightSib.parent.m == 0) { //rightSib.parent is root and is empty.  
 *root* = now;  
 *root*.parent = null; //now is the new root  
 }  
 }  
  
 } else if (leftSib != null) {  
 int leftIdx = indexinP - 1;//index of leftSib in the parent's node  
  
 //add pair with left as child  
 pair p = new pair(now.parent.p.get(leftIdx).key, now.parent.p.get(leftIdx).val, leftSib.r);//and its new child is leftSib.r  
 leftSib.p.add(p);  
 leftSib.m++;  
 //remove the pair  
 now.parent.p.remove(leftIdx);  
 now.parent.m--;  
  
 //add all pairs in now to leftSib  
 for (int i = 0; i < now.m; i++) {  
 leftSib.p.add(now.p.get(i));  
 leftSib.m++;  
 now.p.get(i).child.parent = leftSib;  
 }  
  
 leftSib.r = now.r;//move the rightmost child to the rightmost child of now  
 leftSib.parent = leftSib;  
  
  
 //link the children and parents  
 if (now.parent.m > 0) {  
 if (leftIdx != now.parent.m) {  
 now.parent.p.get(leftIdx).child = leftSib;  
 }  
 } else  
 now.parent.r = leftSib;  
  
 if (leftSib.m >= *nodeSize*) //parent overflow occurred after merging  
 *nonLeafSplit*(leftSib);//split  
  
  
 if (now.parent!=null) {  
 if (now.parent.m < (*nodeSize* - 1) / 2) { //underflow occurred in rightSib.parent(except when parent==root)  
 *parentUnderflow\_Merge*(now.parent);  
 }  
 }  
 else { //parent==root  
 if (now.parent.m == 0) { //needs new root  
 *root* = leftSib; //leftSib is the new root  
 *root*.parent = null;  
 }  
 }  
 }  
 }  
}

delete과정에서 부모 노드의 underflow가 일어난 경우 형제 노드와 merge하는 함수입니다.

마찬가지로 주석에 모든 설명이 있습니다.

먼저 양옆 형제 노드를 만들어 둔 함수를 이용해 찾습니다.

오른쪽 형제를 우선으로 보는 것으로 코딩했습니다.

1. 오른쪽 형제가 있다면

Now가 자식인 페어를 now에 추가하고 그 키의 부모 페어를 삭제합니다.

Now에 rightSib를 전부 옮겨주고 now에서 overflow가 일어난다면 nonLeapSplit를 call합니다

Right 부모가 루트가 아닌데 (차수-1)/2보다 m이 작다면 underflow가 일어난 것이기 때문에 parentUnderflow\_Merge를 다시 호출합니다.

만약 right 부모가 루트인데 아무 것도 없다면 now를 새로운 루트로 지정합니다.

1. 왼쪽 형제가 있다면

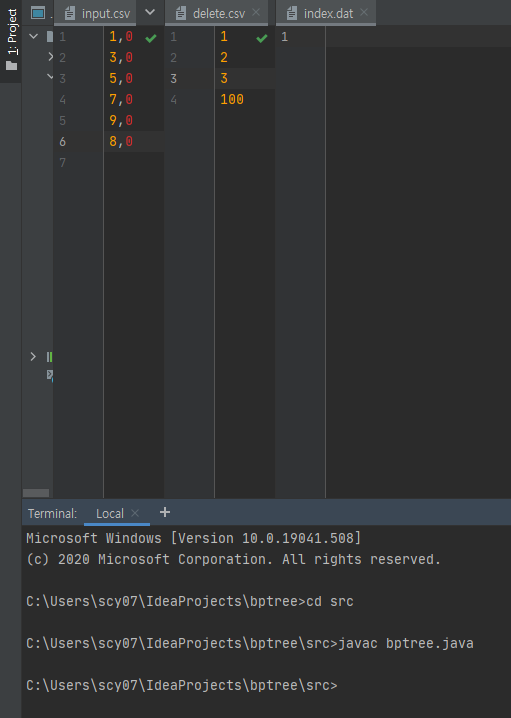
Leftsib가 자식인 페어를 left에 추가하고 그 키의 부모 페어를 삭제합니다.

Leftsib에 now를 모두 옮기고 Leftsib에서 overflow가 일어난다면 nonLeapSplit를 call합니다.

Now의 부모가 루트가 아닌데 (차수-1)/2보다 m이 작다면 underflow가 일어난 것이기 때문에 parentUnderflow\_Merge를 다시 호출합니다.

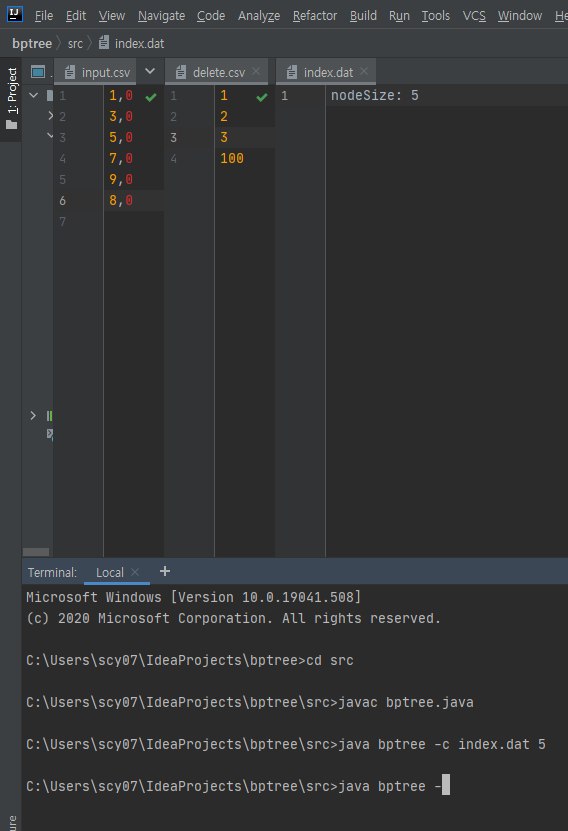
만약 now의 부모가 루트인데 아무 것도 없다면 leftSIb를 새로운 루트로 지정합니다.

**코드실행예시:**



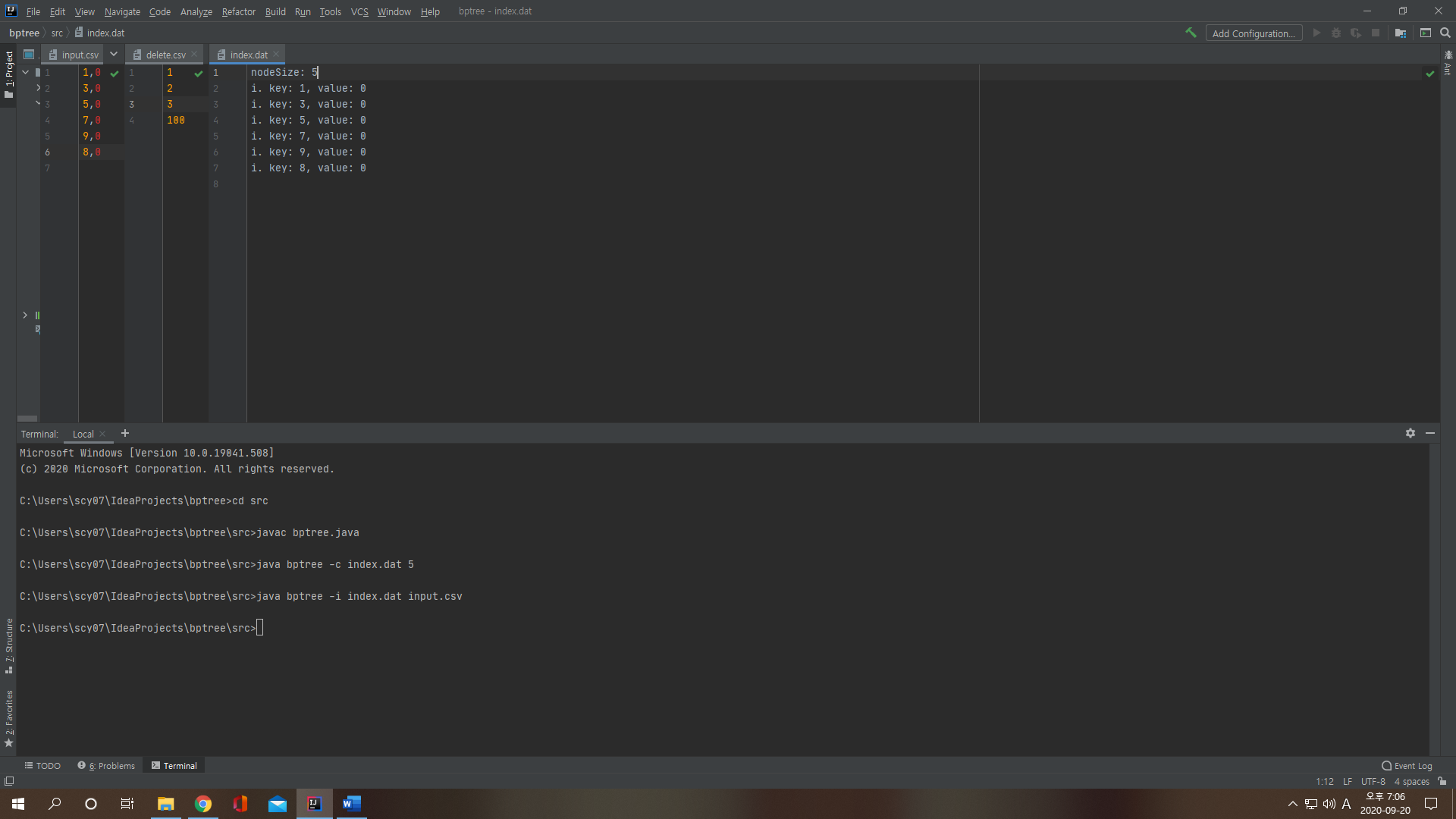
src파일까지 경로를 입력하고

javac bptree.java를 입력해 클래스 생성



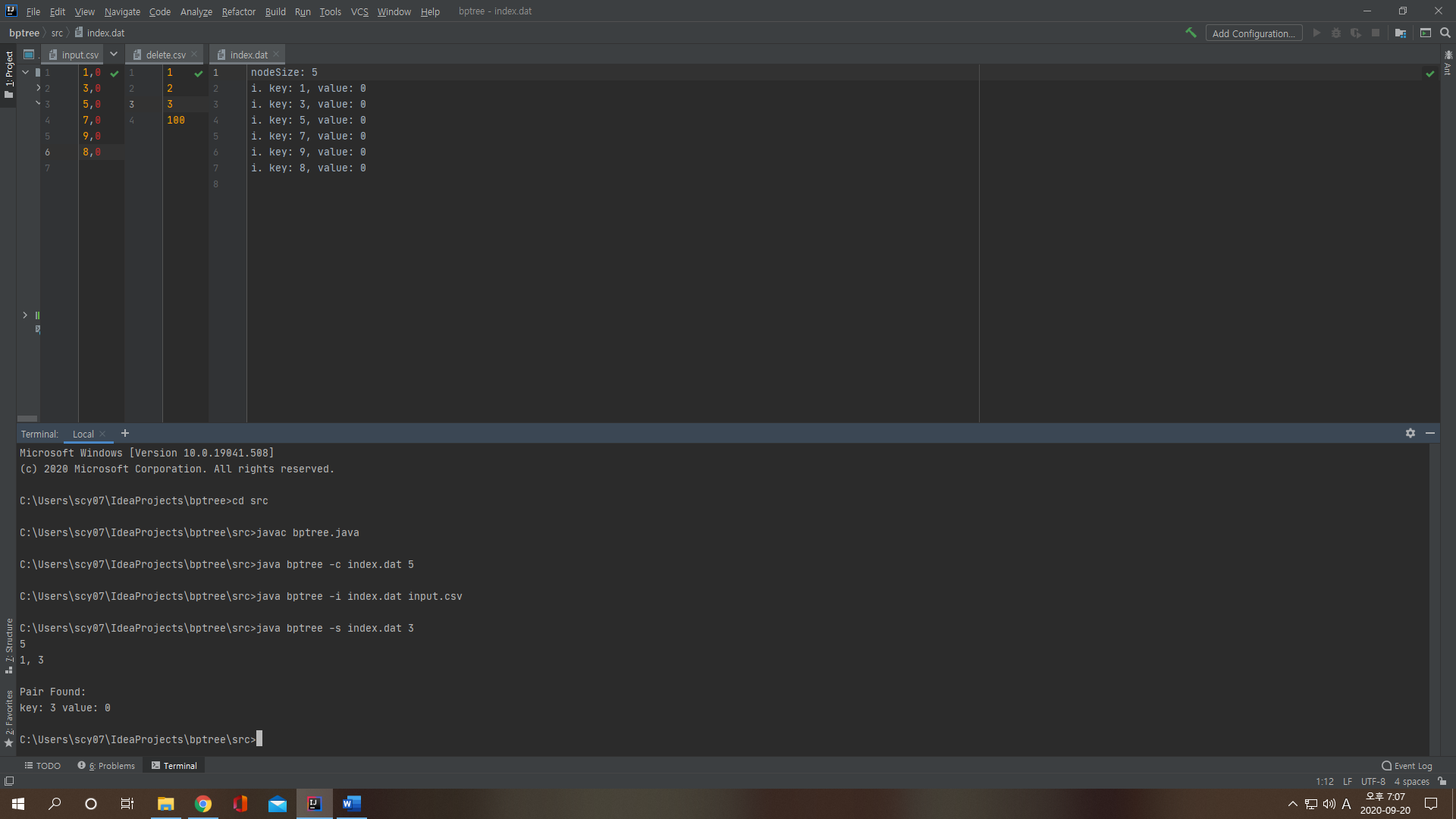
java bptree -c index.dat 5

* 차수가 5인 비플트리 생성
* Index.dat에 nodeside 적힘



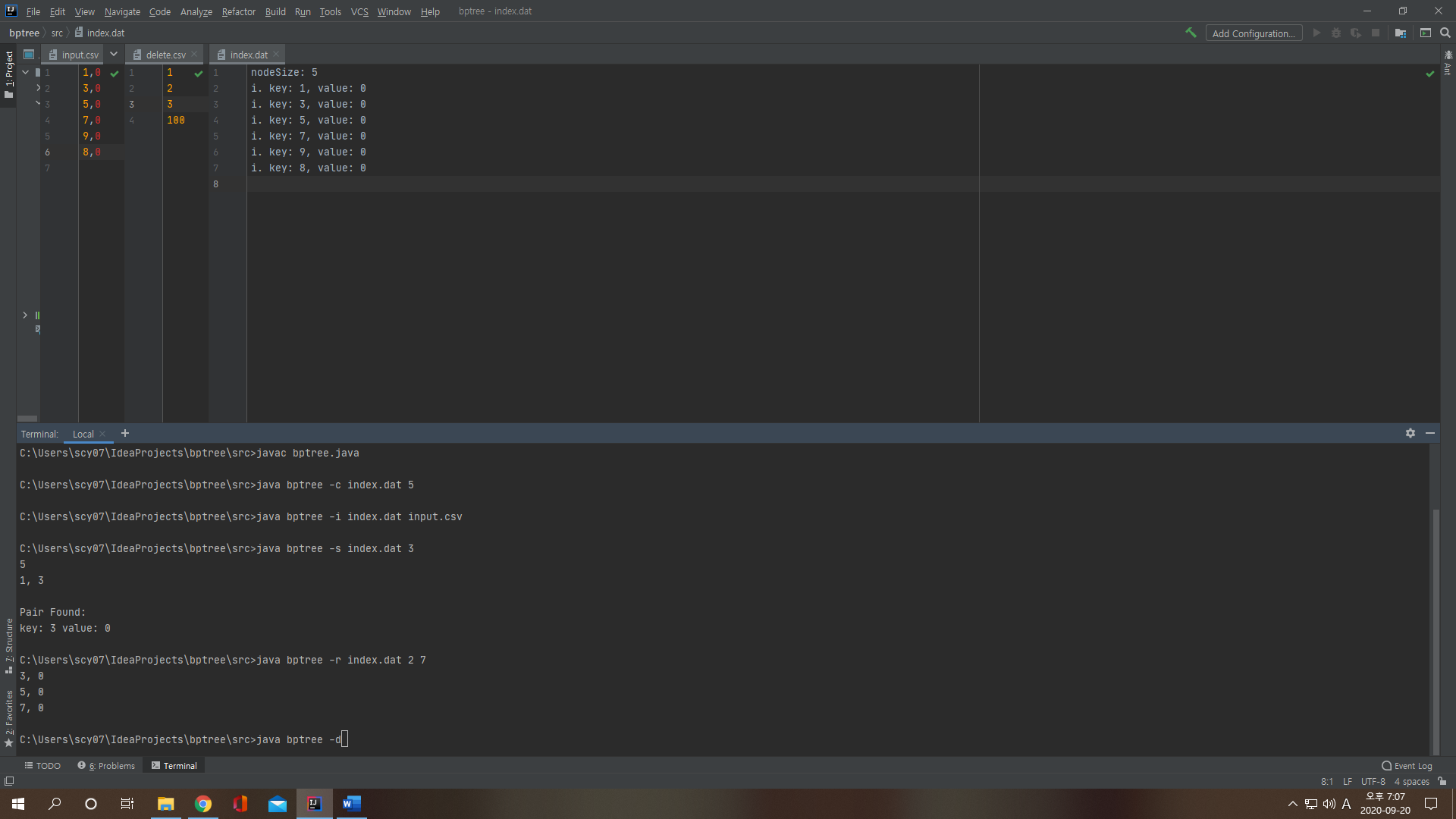
java bptree -I index.dat input.csv입력

* Index.dat에 I, key,val 순서대로 입렫됨

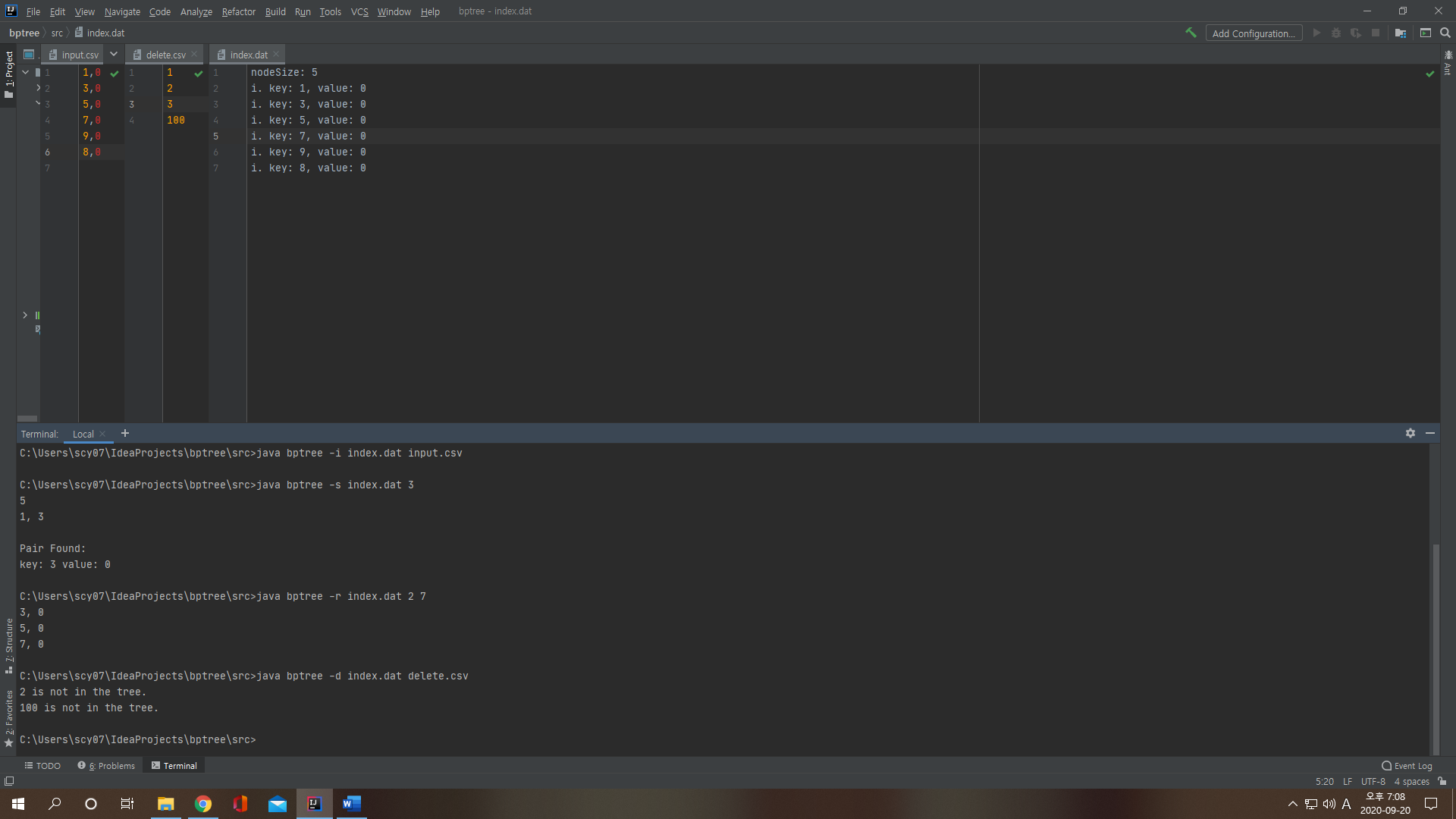


Java bptree -s index.dat 3 (3탐색)

* 3탐색 경로 출력
* 3 val 출력

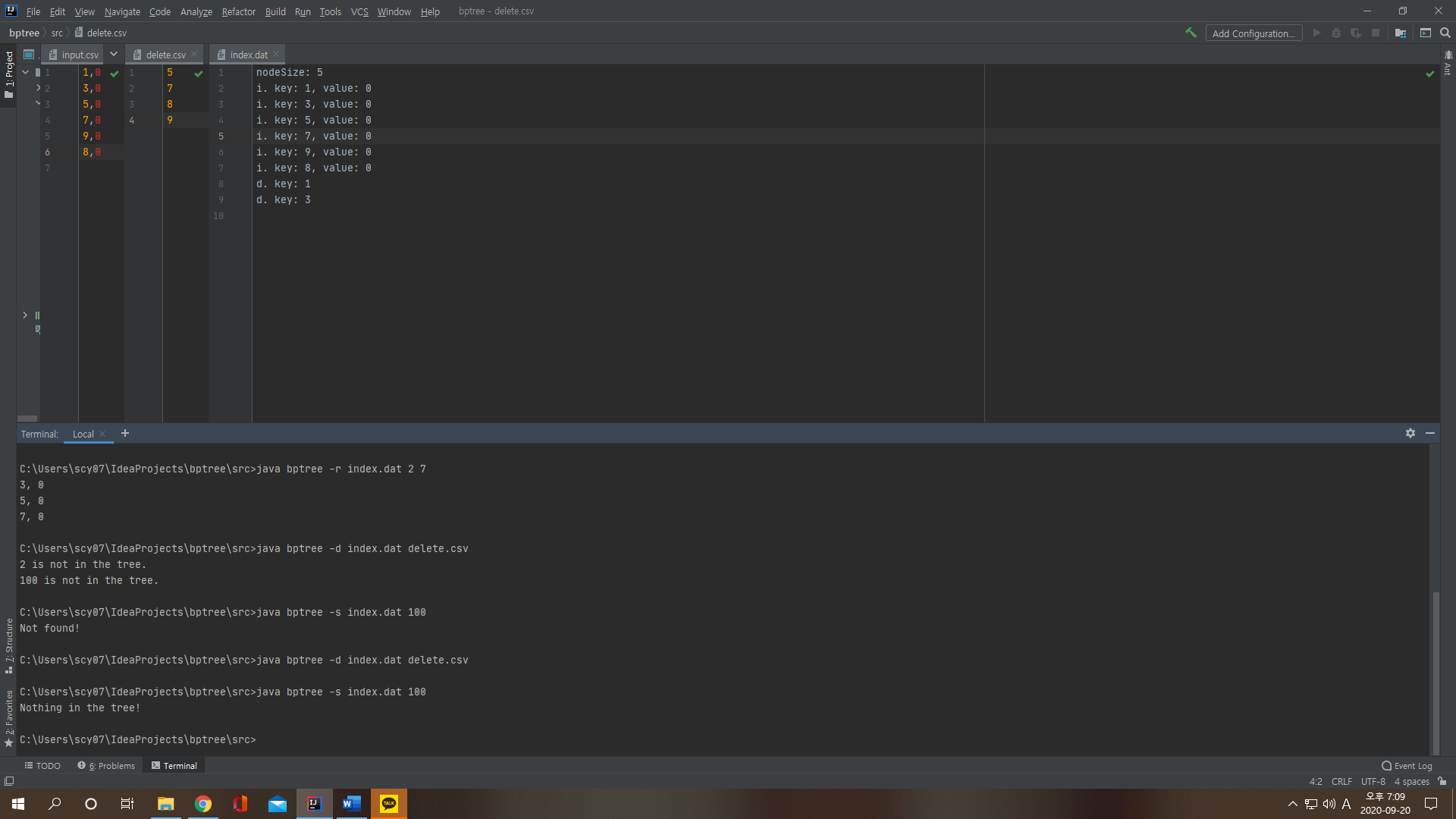
java bptree -r index.dat 2 7입력

* 2-7 범위탐색
* 3,5,7의 key, value출력



java bptree -d index.dat delete.csv 입력

* 있는 키는 지우고 없는 키는 is not int the tree출력
* Index.dat에도 적힘(다음 사진)



추가 실행)

* 없는 100을 단일 탐색하면 not found
* 모든 키를 지운 뒤 탐색하면 nothing in the tree

감사합니다.