2021 B+tree Instruction

과목: Database Systems ( )

교수: 교수님

조교: 조교님, 조교님

소속:

학번:

이름: 송재헌

목차

1. Environment

2. Summary of your algorithm

3. Detailed description of your codes (for each function)

4. Instructions for compiling your source codes at TA's computer (with screenshot)

5. any other specification of your implementation and testing

1. Environment

- OS: Windows 10
- Language: Java (openjdk, Amazon Corretto 17)
- IDE: IntelliJ IDEA 2020.3
- Encoding: UTF-8

2. Summary of your algorithm

모든 *class* 와 *node* 는 *Bptree.java* 내 *public class Bptree* 안에 모두 들어있습니다.

Bptree class 에서 제일 먼저 구현한 *main* 함수는 *cmd* 의 *arguments* 를 해석해 필요한 함수(*insert, delete, searchKey, rangedSearch*)를 실행합니다. 스스로 정의한 *index.dat* 구조를 사용하였고, 이는 *readIndex, writeIndex, BFS*(*writeIndex* 보조 함수) 함수를 통해 읽고 쓸 수 있습니다.

Bptree class 는 자신의 *degree, root* 를 갖습니다.

abstract Node class 는 기능 구현을 위한 *abstract* 함수들(*insertVal, deleteVal, searchKey, rangeSearchKey, isExists, isOverflow, isUnderflow, isLendable, isMergeable*)을 갖고 이 함수들은 *abstract Node class* 를 상속한 *concrete class*(*InternalNode, LeafNode*)에서 다르게 구현됩니다.

InternalNode class 는 내부 배열로 *arr*(*key* 저장), *nArr*(*child* 저장)을 갖고, 자기 자신의 *size* 를 갖습니다. *insertVal* 함수는 *insertion* 시 사용되며, *parent* 라는 변수를 이용해 *insertion* 시 발생하는 *child node* 에서 올라온 *node* 를 확인해, 이를 통합합니다. 이 과정은 재귀적으로 확인되어, 연속적인 *overflow* 를 해결할 수 있습니다. *deleteVal* 함수는 *deletion* 시 사용되며, *isUnderflow* 함수를 통해 *child node* 의 *underflow* 를 확인해 *child node* 의 *InternalNode* 여부, *borrow* 의 좌우 방향, *merge* 의 좌우 방향에 따라 8 가지로 경우를 나누어 해결합니다. *tree* 의 *root* 의 *underflow* 도 확인해 이를 해결합니다. *searchKey* 와 *rangeSearchKey* 는 2 가지 *search* 를 *LeafNode* 의 *override* 된 동일명 함수와 함께 조건에 맞게 실행합니다. *isExists* 함수는 *override* 된 *LeafNode* 의 *isExists* 함수와 함께 해당 *key* 존재 여부를 판별합니다. *isOverflow, isUnderflow, isLendable, isMergeable*

함수들은 *deletion* 에서 해당 조건 여부를 판별합니다. *findLoc* 함수는 *insertion, searchKey, rangedSearch* 에 사용됩니다. *findLocSearchKey* 는 *searchKey* 에 사용됩니다. *getRightChildLeftmost* 함수는 *deletion* 할 때 *InternalNode* 의 *key* 가 삭제된 경우, 그 자리를 메꾸는데 사용됩니다.

LeafNode class 는 내부 배열로 *arr*(*key* 저장), *arr2*(*value* 저장)을 갖고, 자기 자신의 *size* 를 갖습니다. 또한 *siblingNode* 를 오른쪽 *LeafNode* 로의 포인터로 가집니다.

insertVal 함수는 *insertion* 시 사용되며, *overflow* 가 발생하면 *parent* 노드를 생성해, *InternalNode* 에서 활용할 수 있도록 합니다. *deleteVal* 함수는 *deletion* 을 실행합니다. *searchKey* 함수는 *searchKey* 를 실행합니다. *rangeSearchKey* 함수는 *rangedSearch* 를 조건에 맞게 실행하는데, *siblingNode* 를 통해 옆 *node* 로 연속적으로 넘어가며 *search* 합니다. *isExists, isOverflow, isUnderflow, isLendable, isMergeable* 은 해당 조건 여부를 판별하는 *override* 된 함수들입니다.

3. Detailed description of your codes (for each function)

1_Main 부

```
 // main 문은 switch 를 통해 arguments 를 구분해 필요한 함수를 실행합니다.
public static void main(String[] args) {

// 현 위치를 파악하여 파일 i/o 를 진행하기 위한 부분입니다.
    String str = "";
    String filePath = new File("").getAbsolutePath();


// input 파일을 형식에 맞춰 읽고 insert 합니다.
        // isExists 로 유무를 확인하고 insert 합니다.
// delete, search, rangeSearch 도 형식에 맞춰 읽고 실행합니다.
        case "-i": {
            String indexPath = filePath + "/" + args[1];
            String inputPath = filePath + "/" + args[2];
            BufferedReader buffer = null;

// index 파일은 첫 줄에 degree 를 기록합니다.
// 다음줄부터 Node 입니다. 모든 노드 사이엔 \t 으로 띄웁니다.
// InternalNode 의 각 노드는 size, key, key2,... 식입니다.
// 예를 들어 3,1,2,3 이면 size 3 이고 key 로 1,2,3 을 가진
// InternalNode 입니다.
// 마지막줄에 LeafNode 를 기록합니다.
// LeafNode 들은 \으로 시작하고, key 들 나열, value 나열입니다.
// 예를 들어 \2,1,2,3,4 이면 size 가 2 이고 <1,3>,<2,4>의
// <key,value> pair 를 가집니다.
public Bptree readIndex(String indexPath){
             ...
            // 형식에 맞춰 Leaf, Internal 을 구분합니다.

             char ch = strArr[0].charAt(0);
```

```
                if(ch == '\\') {          // leaf node
        ...
                // nodeList 에서 node 들을 꺼내 node 들을 연결합니다.

                // tmp 와 tmp2 변수를 이용해 LeafNode 들이 연속되어 있을 때
                // siblingNode 변수로 오른쪽 노드로 포인팅 해놓습니다.
                if(nodeList.size() > 0){
                    tree.root = nodeList.get(0);
                    Node tmpRoot = tree.root;
                    LeafNode tmp = new LeafNode();
                    LeafNode tmp2 = new LeafNode();
                    int i = 0, j = 1;
                    while(j < nodeList.size()){
```

```
    // index 파일을 구성하는 방식으로 tree 를 기록합니다.

    // queue2 는 각 height 의 가장 왼쪽 InternalNode 를 넣어둬
    // queue 에서 노드를 꺼낼 때 확인해 bplustree 의 height 가
    // 바뀜을 알아냅니다.
    public void writeIndex(Bptree tree, String indexPath){
        BufferedWriter buffer = null;
        try {
            buffer = Files.newBufferedWriter(Paths.get(indexPath));
            buffer.write(tree.degree + newLine);
            BFS(buffer, tree.root);
            buffer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    void BFS(BufferedWriter buffer, Node node){
        String str = "";
        Node tmp = null;
        Queue<Node> queue = new LinkedList<>();
        Queue<Node> queue2 = new LinkedList<>();
```

2_Bptree 부

```
// Bptree 의 variable 에 tmpIn, tmpLN 을 두어 .getClass()에 활용합니다.

    public static Node root;
    public static Node parent = null;
    public InternalNode tmpIN = new InternalNode();
    public LeafNode tmpLN = new LeafNode();
```

3_abstractNode 부

```
// abstract class 인 Node 입니다. 공통 variable 들과 override 할 함수들이 있습니다.
    abstract class Node{
        int degree = 0;
        int size = 0;

        int[] arr = null, arr2 = null;
        Node[] nArr = null;


        abstract Node insertVal(int key, int value);
        abstract Node deleteVal(int key);
        abstract void searchKey(int key);
        abstract void rangeSearchKey(int startKey, int endKey);
        abstract boolean isExists(int key);
        abstract boolean isOverflow();
        abstract boolean isUnderflow();
        abstract boolean isLendable();
        abstract boolean isMergeable();
    }
```

4_InternalNode 부

```
// InternalNode 의 생성자입니다. degree+1 로 배열을 생성해, overflow 에 대비합니다.
        InternalNode(int degree){
            arr = new int[degree+1];
            nArr = new Node[degree+1];
            this.degree = degree;
            this.size = 0;
        }


// InternalNode 의 insert 입니다.
// parent 변수는 child 에서 overflow 가 발생 시 부모 node 로 올라오는 key 입니다.
// 재귀적으로 호출해, 현재에서 overflow 가 연쇄 발생하면 다시 부모 node 에서 해결합니다.
        @Override
        Node insertVal(int key, int value){
            parent = null;
            int loc = findLoc(key);
            nArr[loc].insertVal(key, value);
            if(parent != null){
                this.nArr[this.size+1] = this.nArr[this.size];
                for(int i = this.size; i > loc; i--){
                    ...
                parent = null;
                if(this.isOverflow()) {
```

```
            ...
            return parent;
        }
    }
    return this;
}


// InternalNode 의 delete 입니다.
// flag 변수는 삭제하는 key 값이 internalNode 에 있을 경우 이를 기록합니다.
// child 로 삭제를 재귀적으로 호출하고, child 에서 underflow 발생 시 이를 8 경우로 해결합니다.
// child 의 LeafNode 여부, borrow 의 좌/우 경우, merge 의 좌/우 경우로 8 가지 경우 입니다.
// flag 는 마지막에 getRightChildLeftmost 함수를 호출해 올바른 key 값으로 수정합니다.
@Override
Node deleteVal(int key){
    boolean flag = false;
    int i;
    for(i = 0; i < this.size; i++){
        if(key == this.arr[i]){
            flag = true;
        ...
    }
    Node tmpChild = nArr[i];
    tmpChild.deleteVal(key);
    if(tmpChild.isUnderflow()){
        ...
         //아래부터 8 가지 경우로 나눕니다. 다소 생략하겠습니다.
           if((tmpChildLeftSibling != null) && (tmpChildLeftSibling.isLendable())){
               for (int j = tmpChild.size; j > 0; j--){
                   tmpChild.arr[j] = tmpChild.arr[j-1];
                   tmpChild.arr2[j] = tmpChild.arr2[j-1];
               }
               tmpChild.size++;
               tmpChild.arr[0] = tmpChildLeftSibling.arr[tmpChildLeftSibling.size-1];
               tmpChild.arr2[0] = tmpChildLeftSibling.arr2[tmpChildLeftSibling.size-1];
               tmpChildLeftSibling.size--;
               this.arr[i-1] = tmpChildLeftSibling.arr[tmpChildLeftSibling.size];
           }else if((tmpChildRightSibling != null) && (tmpChildRightSibling.isLendable())){
               tmpChild.arr[tmpChild.size] = tmpChildRightSibling.arr[0];
               tmpChild.arr2[tmpChild.size] = tmpChildRightSibling.arr2[0];
               tmpChild.size++;
               for (int j = 0; j < tmpChildRightSibling.size-1; j++){
                   tmpChildRightSibling.arr[j] = tmpChildRightSibling.arr[j+1];
                   tmpChildRightSibling.arr2[j] = tmpChildRightSibling.arr2[j+1];
               ...
           }
        }else{      // Root 가 아닌 InternalNode
           if((tmpChildLeftSibling != null) && (tmpChildLeftSibling.isLendable())){
```

```
                ...
    }
    if((this.size == 0) && (this.getClass() == tmpIN.getClass())){// collapse
        root = root.nArr[0];
    }else if(flag){
        if(this.getClass()==tmpIN.getClass()){
            this.arr[i-1] = this.getRightChildLeftmost(i-1);
        }
    }
    return root;
}


// 재귀적으로 search 를 호출합니다.
@Override
void searchKey(int key){
    int loc = findLocSearchKey(key);
    this.nArr[loc].searchKey(key);
}
@Override
void rangeSearchKey(int startKey, int endKey){
    int loc = findLoc(startKey);
    this.nArr[loc].rangeSearchKey(startKey, endKey);
}


 // key 의 존재여부를 확인하는 isExists 함수입니다.
// 재귀적으로 호출해 존재 여부를 확인합니다.
// while 문으로 overhead 를 줄이려고 했습니다.
@Override
boolean isExists(int key){
    int i = 0;
    Node tmp = this;
    while(tmp.getClass() == tmpIN.getClass()) {
        for (i = 0; i < tmp.size; i++) {
            if(key == tmp.arr[i]){
                return true;
            }else if (key > tmp.arr[i]) {
                continue;
            }else{
                break;
            }
        }
        tmp = tmp.nArr[i];
    }
    return tmp.isExists(key);
}
```

```
// InternalNode 에서 overflow, underflow,
// lendable(borrow 해줄 수 있는지), mergeable 등을 판단합니다.
@Override
boolean isOverflow() {
    if(this.size == this.degree){
        return true;
    }else{
        return false;
    }
}
@Override
boolean isUnderflow(){
    if(this.size < (int)(this.degree-1)/2){
        return true;
    }else{
        return false;
    }
}
@Override
boolean isLendable(){
    if(this.size > (int)(this.degree-1)/2){
        return true;
    }else{
        return false;
    }
}
@Override
boolean isMergeable(){
    if(this.size <= (int)(this.degree-1)/2){
        return true;
    }else{
        return false;
    }
}


 // tree 안에서 올바른 key 위치를 찾는데 활용됩니다.
// insert, search, rangedSearch 등에 활용.
int findLoc(int key){
    for(int i = 0; i < size; i++){
        if(key >= this.arr[i]){
            continue;
        }else{
            return i;
        }
    }
    return size;
}
```

```
int findLocSearchKey(int key){
    System.out.print(this.arr[0]);
    if(key < this.arr[0]){
        System.out.println();
        return 0;
    }
    int i;
    for(i = 1; i < size; i++){
        System.out.print("," + this.arr[i]);
        if(key < this.arr[i]){
            break;
        }
    }
    System.out.println();
    return i;
}


 // InternalNode 의 key 에 삭제한 key 가 있을 경우 올바른 key 를 찾습니다.
int getRightChildLeftmost(int idx){
    Node tmp = null;
    tmp = this.nArr[idx+1];
    while(tmp.getClass() != tmpLN.getClass()){
        tmp = tmp.nArr[0];
    }
    return tmp.arr[0];
}
```

5_LeafNode 부

```
// LeafNode 는 variable siblingNode 를 가져 오른쪽 LeafNode 로의 포인터를 가집니다.
class LeafNode extends Node{
    LeafNode siblingNode;      // a pointer to the right sibling node.

    LeafNode(){ };

    LeafNode(int degree){
        arr = new int[degree+1];  // for keys
        arr2 = new int[degree+1];  // for values
        this.degree = degree;
        size = 0;
        siblingNode = null;
    }
```

```
// LeafNode 의 insert 입니다.
// overflow 발생 시 부모 node 로 parent node 를 올려 부모 node 에서 처리합니다.
@Override
Node insertVal(int key, int value){
    size++;
    int i;
    for(i = 0; i < size-1; i++){
        if(key > arr[i]) continue;
        else break;
    }
    if(i == size-1){
        arr[i] = key;
        arr2[i] = value;
    }else{
        for(int j = size; j > i; j--){
            arr[j] = arr[j-1];
            arr2[j] = arr2[j-1];
        }
        arr[i] = key;
        arr2[i] = value;
    }
    if(this.isOverflow()){   // 오버플로우 발생하는 경우
        int newSize = this.degree/2;
        LeafNode newSibling = new LeafNode(getDegree());
        for(i = 0; i < (degree-newSize); i++){
            newSibling.arr[i] = this.arr[newSize+i];
            newSibling.arr2[i] = this.arr2[newSize+i];
        }
        newSibling.size = this.size - newSize;
        this.size = newSize;
        newSibling.siblingNode = this.siblingNode;
        this.siblingNode = newSibling;
        // 가운데 노드를 부모 노드로 올려줘야함
        parent = new InternalNode(getDegree());
        parent.size = 1;
        parent.arr[0] = newSibling.arr[0];
        parent.nArr[0] = this;
        parent.nArr[1] = newSibling;
        return parent;
    }
    return this;
}
```

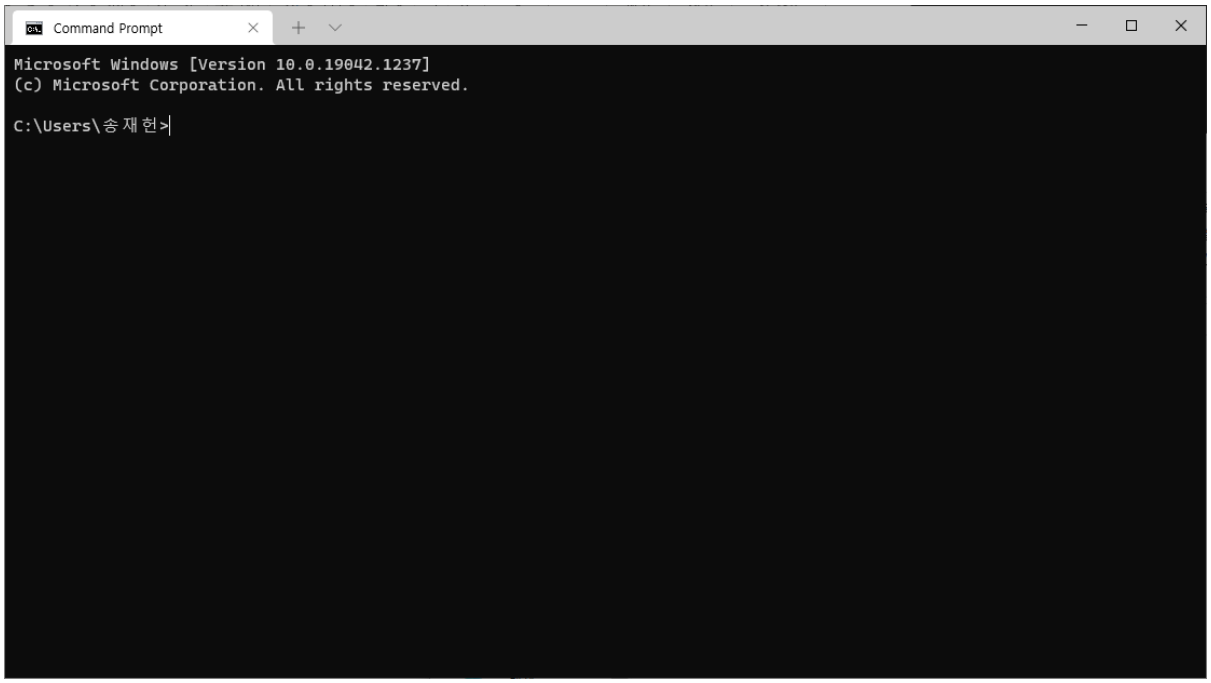
```
// LeafNode 의 delete 입니다.
@Override
Node deleteVal(int key){
    int i;
    for(i = 0; i < this.size; i++){
        if(key == this.arr[i])
            break;
    }
    for(int j = i; j < this.size-1; j++){
        this.arr[j] = this.arr[j+1];
        this.arr2[j] = this.arr2[j+1];
    }
    this.size--;
    return this;
}


// LeafNode 의 search 와 rangedSearch 입니다.
// rangedSearch 에서 LeafNode 의 siblingNode 가 활용됩니다.
@Override
void searchKey(int key){
    for(int i = 0; i < size; i++){
        if(key == this.arr[i]){
            System.out.println(arr2[i]);
            return;
        }
    }
    System.out.println("NOT FOUND");
}
@Override
void rangeSearchKey(int startKey, int endKey){
    boolean flag = true;
    int i;
    for(i = 0; i < size; i++){
        if(startKey > this.arr[i]){
            continue;
        }else{
            flag = false;
            LeafNode tmp = this;
            while(tmp.arr[i] <= endKey){
                System.out.println(tmp.arr[i] + "," + tmp.arr2[i]);
                i++;
                if(i == tmp.size){
                    if(tmp.siblingNode != null){
                        tmp = tmp.siblingNode;
                        i = 0;
```

```
                    }
                }
            }
            return;
        }
    }
    if(flag){
        if(this.siblingNode != null){
            LeafNode tmp = this.siblingNode;
            i = 0;
            while(tmp.arr[i] <= endKey){
                System.out.println(tmp.arr[i] + "," + tmp.arr2[i]);
                i++;
                if(i == tmp.size){
                    if(tmp.siblingNode != null){
                        tmp = tmp.siblingNode;
                        i = 0;
                    }
                }
            }
        }
    }
}


 // key 의 존재여부를 확인하는 isExists 함수와
// LeafNode 에서 overflow, underflow,
// lendable(borrow 해줄 수 있는지), mergeable 등을 판단합니다.
@Override
boolean isExists(int key) {
    for(int i = 0; i < this.size; i++){
        if(key == this.arr[i]){
            return true;
        }else{
            continue;
        }
    }
    return false;
}

@Override
boolean isOverflow() {
    if(this.size == this.degree){
        return true;
    }else{
        return false;
    }
}
```

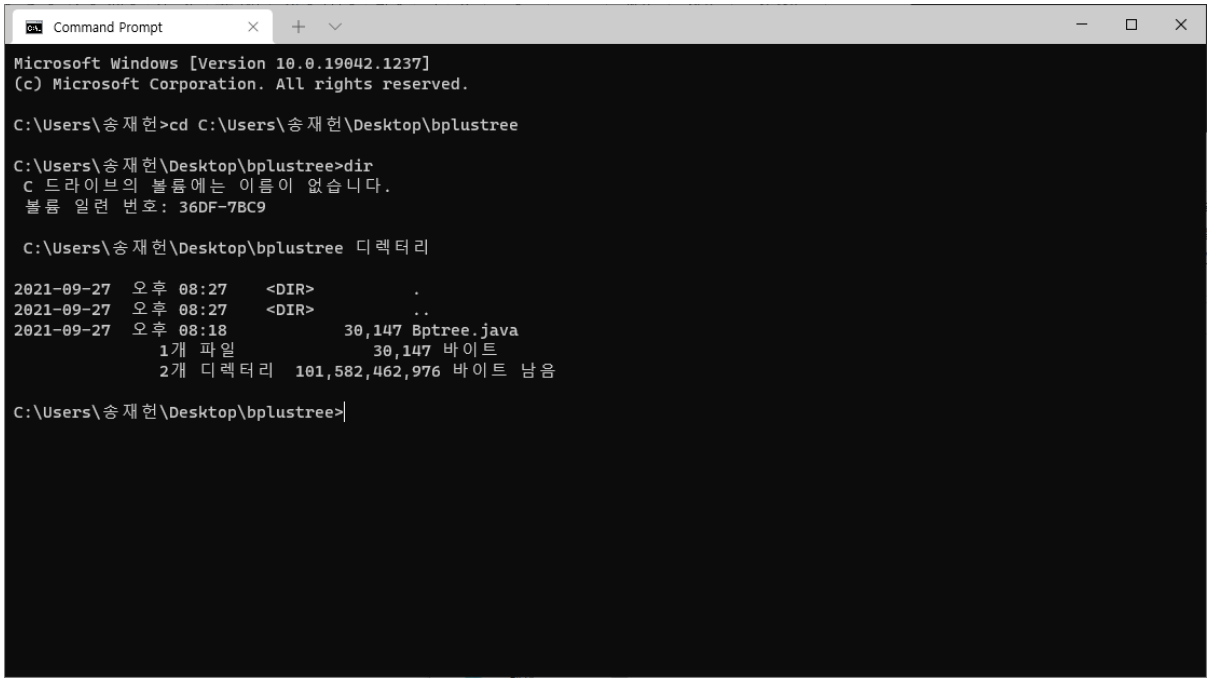
```
@Override
boolean isUnderflow(){
    if(this.size < (int)(this.degree/2)){
        return true;
    }else{
        return false;
    }
}

@Override
boolean isLendable(){
    if(this.size > (int)(this.degree/2)){
        return true;
    }else{
        return false;
    }
}

@Override
boolean isMergeable(){
    if(this.size <= (int)(this.degree/2)){
        return true;
    }else{
        return false;
    }
}
```

4. Instructions for compiling your source codes at TA's computer (with screenshot)

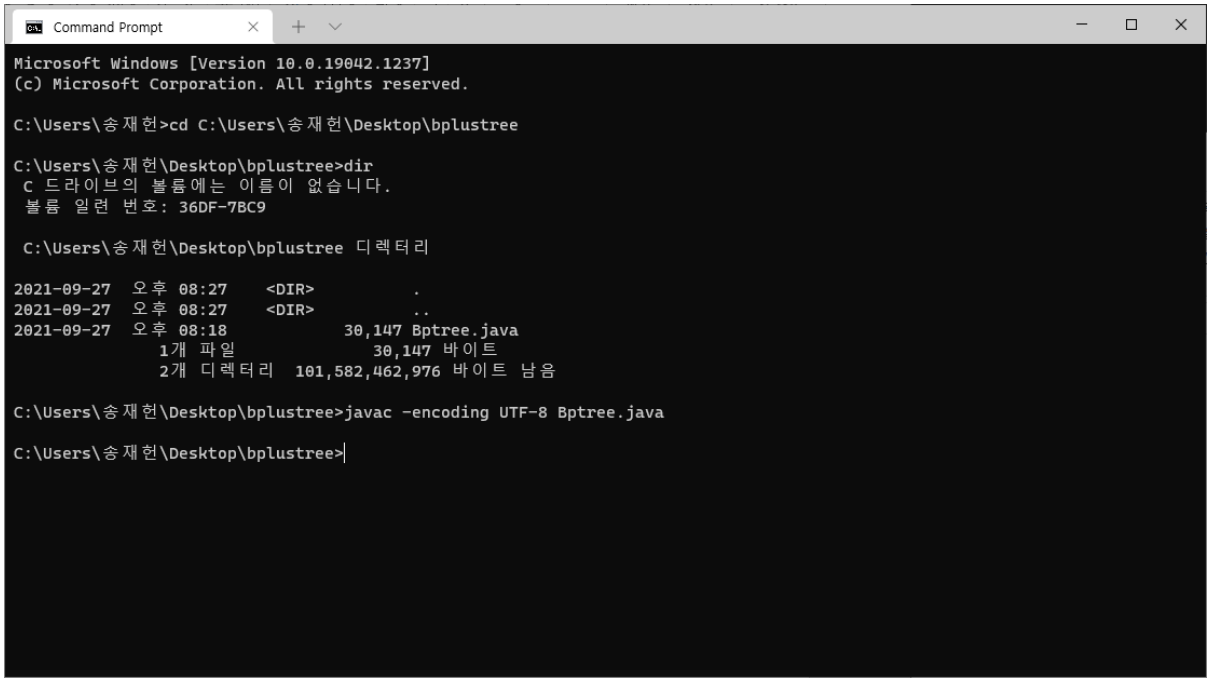
설명은 사진과 그에 대한 설명 순서입니다.



1. Windows cmd 열어주세요.

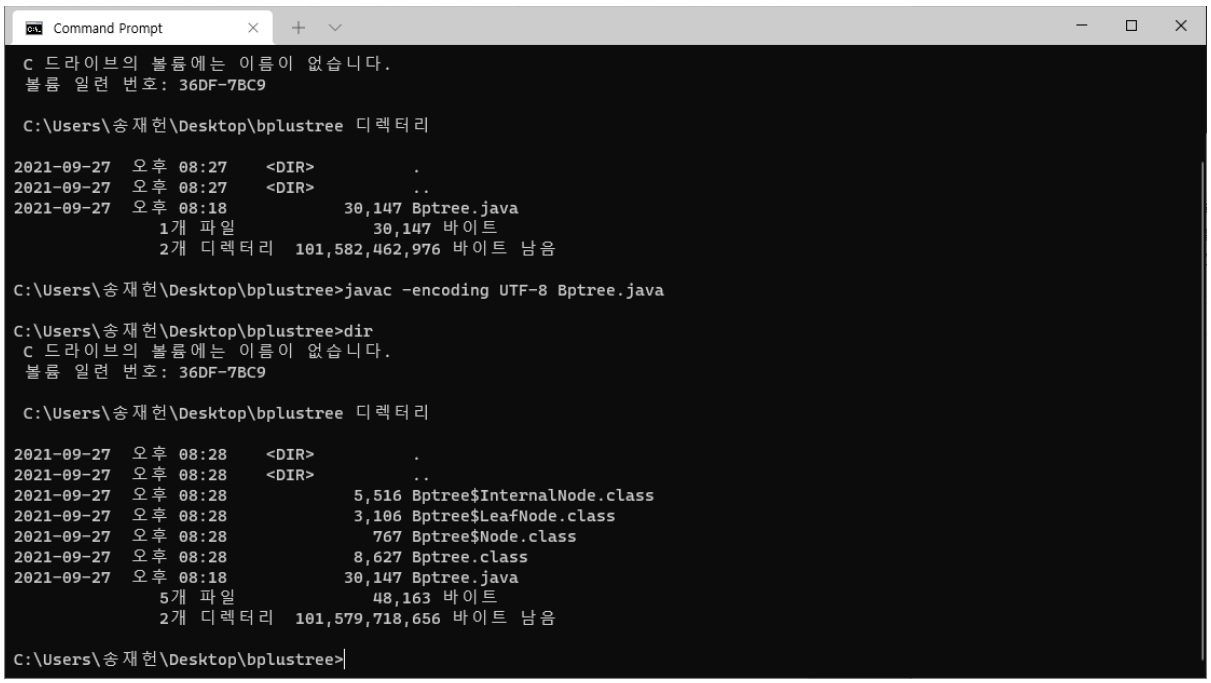


2. Bptree.java 가 있는 디렉토리로 이동해주세요.

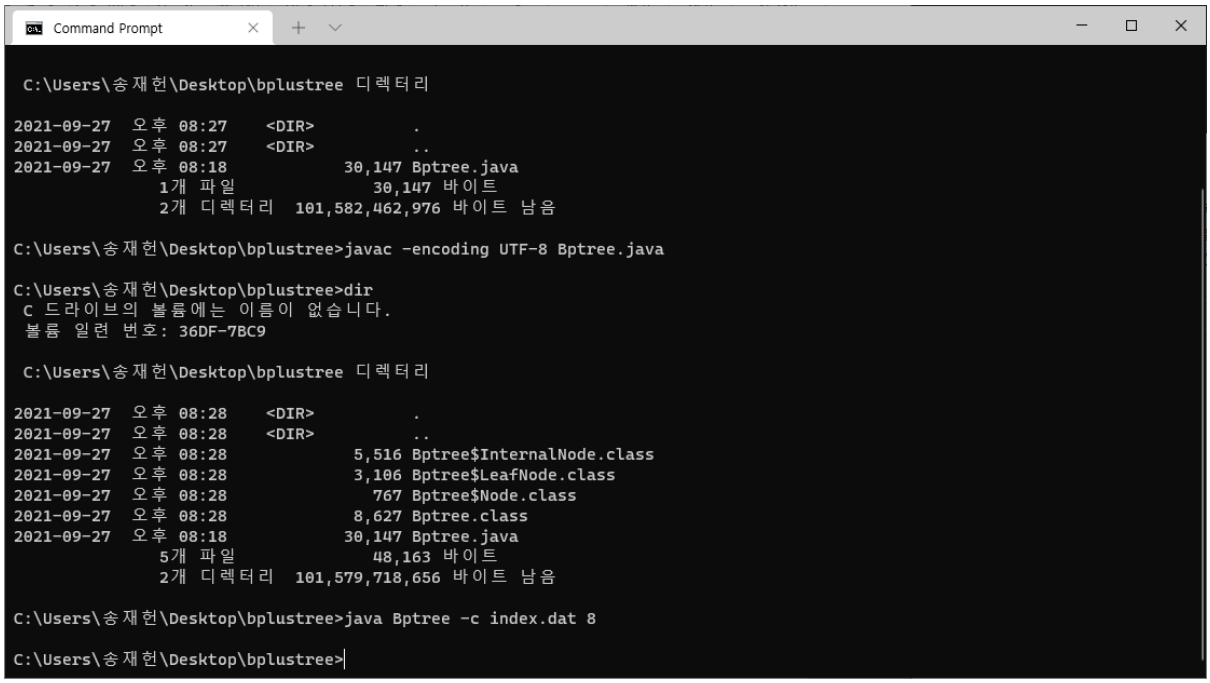


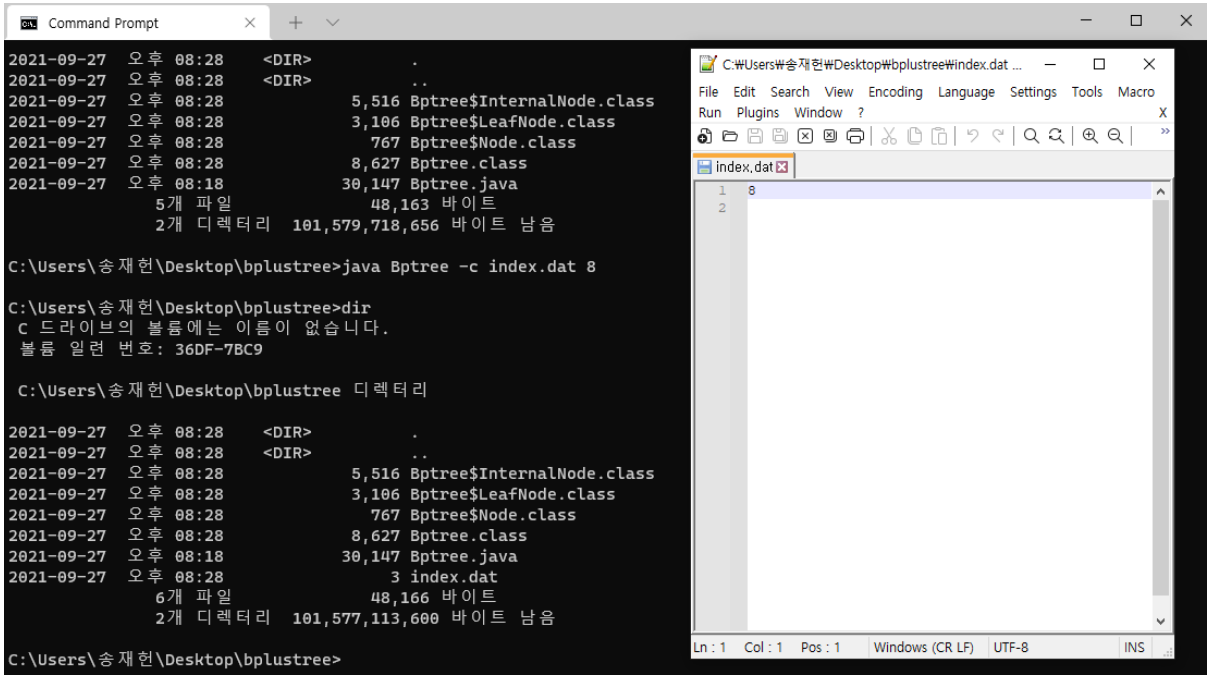
3. 컴파일 코드: UTF-8 encoding 포맷 해석을 위해 -encoding UTF-8 을 함께 입력해주세요.

```
javac -encoding UTF-8 Bptree.java
```



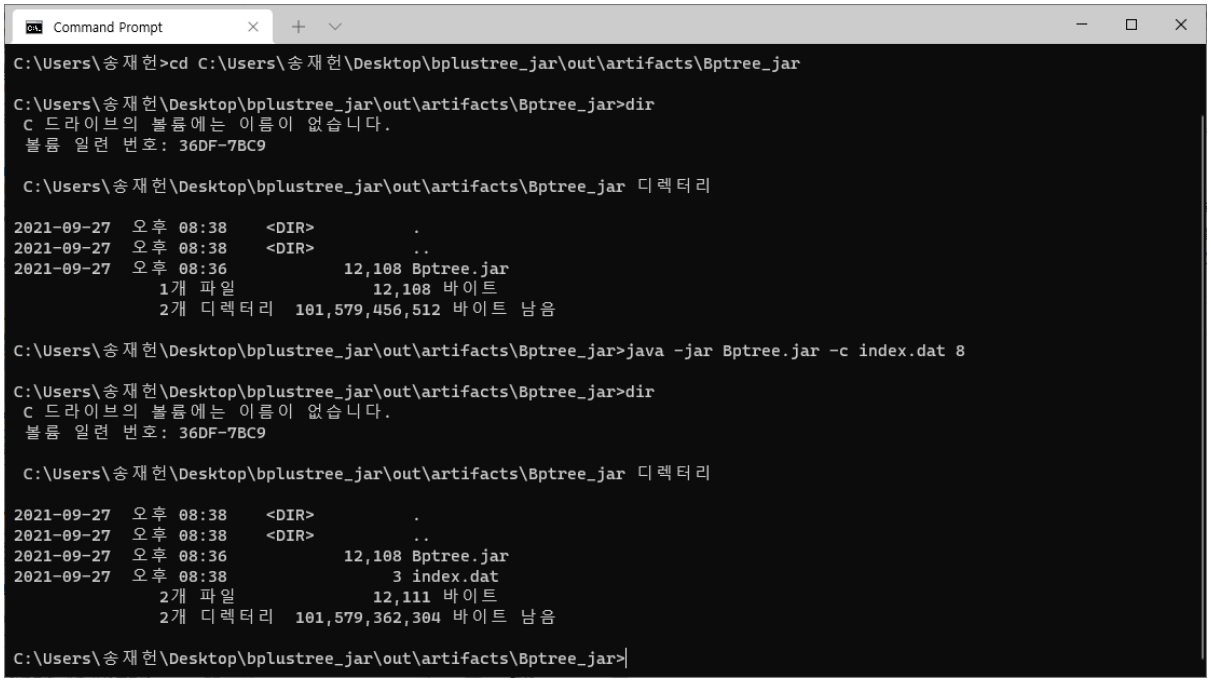
4. .class 파일들이 생성되었습니다.





5. 프로그램 실행: 예시로 index.dat 파일을 만들어보았습니다.

java Bptree -c index.dat 8

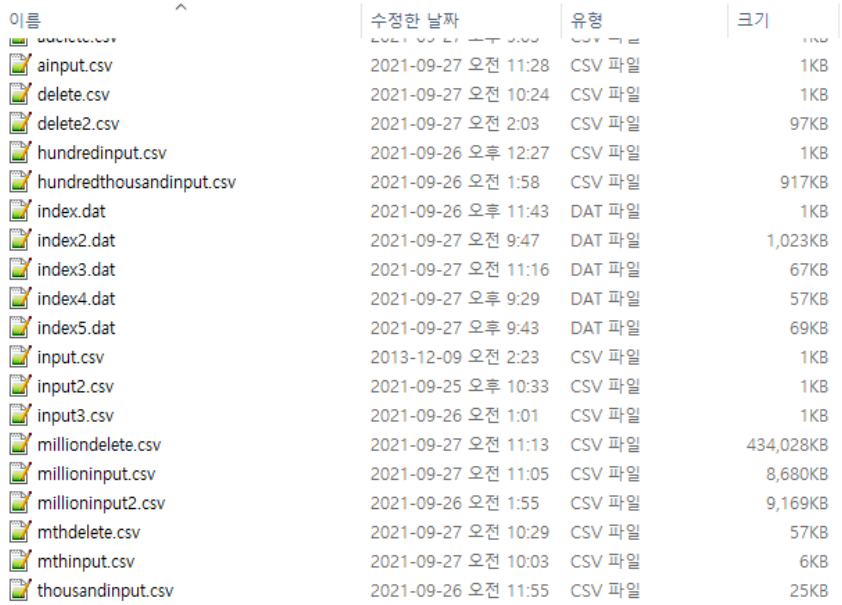


번외) .jar 실행: 해당 디렉토리로 이동 후 “java -jar Bptree.jar + <arguments>”로 실행합니다. 예시로 index.dat 파일을 만들어보았습니다.

5. any other specification of your implementation and testing

- OS: Windows 10
- Language: Java (openjdk, Amazon Corretto 17)
- IDE: IntelliJ IDEA 2020.3
- Encoding: UTF-8

```
// 아래와 같은 형식의 insert/delete maker 를 만들어 테스팅에 활용했습니다.
int main(int argc, char** args){
    srand((unsigned int)time(NULL));
    fout = fopen(args[1], "w");
    for(int i = 0; i < 1000000; i++){
        if(i%2 == 0){
            fprintf(fout, "%d,", Randomize()%10000000);
        }else{
            fprintf(fout, "%d\n", Randomize()%100000000);
        }
    }
    fclose(fout);
    return 0;
}
```



이름	수정한 날짜	유형	크기
ainput.csv	2021-09-27 오전 11:28	CSV 파일	1KB
delete.csv	2021-09-27 오전 10:24	CSV 파일	1KB
delete2.csv	2021-09-27 오전 2:03	CSV 파일	97KB
hundredinput.csv	2021-09-26 오후 12:27	CSV 파일	1KB
hundredthousandinput.csv	2021-09-26 오전 1:58	CSV 파일	917KB
index.dat	2021-09-26 오후 11:43	DAT 파일	1KB
index2.dat	2021-09-27 오전 9:47	DAT 파일	1,023KB
index3.dat	2021-09-27 오전 11:16	DAT 파일	67KB
index4.dat	2021-09-27 오후 9:29	DAT 파일	57KB
index5.dat	2021-09-27 오후 9:43	DAT 파일	69KB
input.csv	2013-12-09 오전 2:23	CSV 파일	1KB
input2.csv	2021-09-25 오후 10:33	CSV 파일	1KB
input3.csv	2021-09-26 오전 1:01	CSV 파일	1KB
milliondelete.csv	2021-09-27 오전 11:13	CSV 파일	434,028KB
millioninput.csv	2021-09-27 오전 11:05	CSV 파일	8,680KB
millioninput2.csv	2021-09-26 오전 1:55	CSV 파일	9,169KB
mthdelete.csv	2021-09-27 오전 10:29	CSV 파일	57KB
mthinput.csv	2021-09-27 오전 10:03	CSV 파일	6KB
thousandinput.csv	2021-09-26 오전 11:55	CSV 파일	25KB