2020 B+ Tree Implementation Assignment

Database Systems (ITE2038)

컴퓨터소프트웨어학부

2019054957 이용우

1. Environment

OS: Windows

Language: Java (jdk 1.8.0_241)

2. Configuration Files

BpTree.java

– Root 노드에 대하여 Insert, Delete, Search를 수행한다.

– Disk에 노드 정보를 저장하고, 이를 다시 메모리로 불러오는 함수가 구현되어 있다.

Node.java

– Abstract Class로 만들어 LeafNode와 NonLeafNode의 공통된 structure와 함수가 구현되어 있다.

LeafNode.java

– LeafNode를 구성하는 멤버 변수와 함수들이 구현되어 있다.

NonLeafNode.java

– NonLeafNode를 구성하는 멤버 변수와 함수들이 구현되어 있다.

Main.java

– Command Line에서의 명령을 처리한다.

3. Summary Algorithm && Description of Codes

Insertion

- LeafNode.java

```
public Node insert(Node root, int key, int value)
```

Key 가 들어갈 위치를 Binary Search 로 찾아 삽입하여 정렬된 상태를 유지한다.

Key 를 삽입하여 List 의 크기가 Degree – 1 을 초과하면 Split 한다.

새로 NonLeafNode 를 생성하여 Right Sibling 의 Leftmost Key 값을 넣어준다.

Split 이 일어나는 노드가 Root 노드라면 부모 노드를 반환하여 Root 노드를 갱신한다.

이미 존재하는 Key 를 삽입할 때에는 예외처리를 한다.

- NonLeafNode.java

```
public Node insert(Node root, int key, int value)
```

Child 의 위치를 Binary Search 로 찾아 Insert 를 재귀적으로 수행한다.

해당 Child 에 Overflow 가 발생하면 Split 을 수행한다.

Child 가 LeafNode 이면 Right Sibling 의 Leftmost Key 값을, NonLeafNode 이면 중간 지점의 Key 값을 새로운 NonLeafNode 에 넣어준다.

Split 이 일어나는 노드가 Root 노드라면 부모 노드를 반환하여 Root 노드를 갱신한다.

```
public Node split()
```

SplitNode 를 생성하여 Key 와 values 를 절반으로 나눈다.

LeafNode 에서는 Right Node 를 가리키는 포인터를 연결해준다.

- BpTree.java

```
public void insert(int key, int value) {
    Node temp = root.insert(root, key, value);
    if(temp != null) {
        root = temp;
    }
}
```

Insert 수행 후 Split 에 의해 새로운 노드가 반환되면 Root 노드를 갱신한다.

Deletion

- LeafNode.java

```
public Node delete(Node root, int key)
```

해당 노드에 Key 와 일치하는 Key 값이 있는지 찾는다.

존재하면 Key 와 Value 를 지워주고 존재하지 않으면 예외처리를 한다.

- NonLeafNode.java

```
public Node delete(Node root, int key)
```

Child 의 위치를 Binary Search 로 찾아 Delete 를 재귀적으로 수행한다.

LeafNode 의 Leftmost Key 가 지워지면 부모 노드의 Key 를 갱신한다.

Child 에 Underflow 가 일어나면 Sibling 을 찾아 Borrow 나 Merge 를 수행한다.

Root 노드에 Merge 가 일어날 경우 Root 노드를 자식 노드로 갱신한다.

- Node.java

```
public boolean canBorrow() {
    return getSize() - 1 >= (degree - 1) / 2;
}
```

Borrow 가 가능한 지 확인한다.

```
void borrowFromLeft(Node parentNode, Node siblingNode, int loc)
```

Left Sibling 에 대해서 Borrow 를 수행한다.

LeafNode 에서는 Left Sibling 에게서 Key 를 Borrow 하고 부모 노드의 Key 를 갱신한다.

NonLeafNode 에서는 부모 노드의 Key 를 하나 가져오고, 그 위치에 Left Sibling 의 Key 값으로 채운다.

```
void borrowFromRight(Node parentNode, Node siblingNode, int loc)
```

Right Sibling 에 대해서 Borrow 를 수행한다.

```
public void merge(Node parentNode, Node siblingNode, int loc)
```

노드를 병합하는 과정이다.

LeafNode 에서는 병합 후에 부모 노드의 키를 지워준다.

NonLeafNode 에서는 부모 노드의 키를 가져온 후 Sibling 과 병합을 수행한다.

- BpTree.java

```
public void delete(int key) {
    Node temp = root.delete(root, key);
    if(temp != null) {
        root = temp;
    }
}
```

Delete 수행 후 Merge 에 의해 자식 노드가 반환되면 Root 노드를 자식 노드로 갱신한다.

Single Key Search

- LeafNode.java

```
public int getValue(int key)
```

Key 가 존재하면 Value 를 반환한다.

- NonLeafNode.java

```
public int getValue(int key)
```

Child 의 위치를 Binary Search 로 찾아 재귀적으로 Search 를 수행한다.

Search 경로 중 NonLeafNode 에 방문하면 그 노드의 모든 Key 값을 출력한다. (Stdout)

LeafNode 에 도착하면 Value 만 출력한다.

- BpTree.java

```
public int keySearch(int key) {
    return root.getValue(key);
}
```

Search 결과인 Value 를 그대로 반환한다.

- Main.java

```
BpTree bptree = BpTree.load(input, order);
int value = bptree.keySearch(key);
if(value != -1) {
    System.out.println(value);
}
else {
    System.out.println("NOT FOUND");
}
```

받아온 Value 를 다음과 같이 핸들링한다.

Ranged Search

- LeafNode.java

```
void rangeSearch(int startKey, int endKey)
```

Keys 를 루프를 돌면서 startKey 와 endKey 의 범위 안에 있으면 각각 출력한다. (Stdout)

Right Node 가 nulll 이 아니면 Right Node 에 대해 재귀적으로 Search 를 수행한다.

방문한 노드의 Keys 의 Leftmost Key 가 endKey 보다 크다면 Return 하여 재귀함수를 끝낸다.

- NonLeafNode.java

```
void rangeSearch(int startKey, int endKey)
```

Child 의 위치를 Binary Search 로 찾아 rangeSearch 를 재귀적으로 수행한다.

- BpTree.java

```
public void rangeSearch(int start, int end) {
    root.rangeSearch(start, end);
}
```

Root 노드에 대해서 rangeSearch 를 수행한다.

Save

index.dat 에 Tree 정보를 저장한다.

- BpTree.java

```
public void save(PrintWriter output, Node node) {
    if(node == null) {
        return;
    }
    if(node instanceof NonLeafNode) {
        NonLeafNode nonLeafNode = (NonLeafNode)node;
        for (int i = 0; i < nonLeafNode.getChildNode().size(); i++) {
            save(output, nonLeafNode.getChildNode().get(i));
        }
        output.print(nonLeafNode.toString() + "\n");
    }
    else if(node instanceof LeafNode) {
        output.print(node.toString() + "\n");
    }
}
```

Post-Order Traversal 을 이용하여 LeafNode 들의 정보부터 파일에 기록한다.

형식) LeafNode: key1,key2,key3... value1,value2,value3

NonLeafNode: key1,key2,key3...

Load

- BpTree.java

```
public static BpTree load(Scanner input, int order)
```

Index.dat 에 기록된 데이터를 메모리로 불러온다.

```
while(input.hasNextLine()) {
    String info = input.nextLine();
    String[] nodeInfo = info.split(" ");
```

데이터를 줄 단위로 읽어와서 파싱한다.

```
for(int i = 0; i < count.size(); i++) {
    if(keysInfo.length + 1 == count.get(i).size()) {
        NonLeafNode nonLeafNode = new NonLeafNode(order, keys, count.get(i));
        count.get(i).clear();
        if(count.size() == i + 1) {
            count.add(new ArrayList<>());
        }
        count.get(i + 1).add(nonLeafNode);
        root = nonLeafNode;
        break;
    }
}
keys.clear();
```

트리의 Level 별로 담아 놓는 count 라는 2D List 를 만든다.

Child 노드들의 수는 Keys + 1 이라는 것을 이용하여, NonLeafNode 의 Child 들을 구분한다.

Child 가 전부 count[i]에 들어가면 NonLeafNode 를 새로 만들어 count[i]를 자식 노드로 설정한다.

새로 만들어진 NonLeafNode 를 Root 로 설정한다.

파일 끝까지 반복하면 트리를 메모리에 모두 불러올 수 있다.

Save 된 Index.dat 파일의 구성은 아래 테스트 부분에서 자세히 확인할 수 있다.

4. How to Compile Source Codes

```
C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>java -version
java version "1.8.0_241"
Java(TM) SE Runtime Environment (build 1.8.0_241-b07)
Java HotSpot(TM) 64-Bit Server VM (build 25.241-b07, mixed mode)

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>cd Source

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment₩Source>ls
Main.java  bpTree

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment₩Source>javac Main.java bpTree/*.java

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment₩Source>ls
Main.class  Main.java  bpTree

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment₩Source>java Main -c index.dat 20

C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment₩Source>ls
Main.class  Main.java  bpTree  index.dat
```

1) Source 폴더로 이동한다. (Main.java 와 bpTree 패키지가 존재한다.)
2) javac Main.java bpTree/*.java 명령어를 실행시킨다.

   (Main.java 와 bpTree 폴더 내의 java 확장자의 모든 파일을 컴파일한다.)
3) class 파일이 생성된 것을 확인한다.

컴파일에 문제가 생긴다면 jdk 1.8.0_241 버전을 다운받아 환경변수로 설정한다.

실행 방법

1) Main.class 를 이용할 경우 다음과 같은 형식의 명령어를 사용한다.

   Ex) java Main -c index.dat 20
2) 제공된 jar 파일을 이용할 경우 B-tree_Assignment 폴더에 있는

   B-tree_Assignment.jar 로 다음과 같은 형식의 명령어를 사용한다.

   Ex) java -jar B-tree_Assignment.jar -c index.dat 20

5. Testing

과제 채점과 동일하게 백만개 정도의 데이터를 가지고 테스트를 진행했다.

dataCreater.py > ...

```python
import random

select = input("Insert(i) or Delete(d) : ")

keys = [i for i in range(1, 1000001)]

random.shuffle(keys)

if select == "i":
    with open("insertTest.csv", "w") as f:
        for key in keys:
            f.write(str(key) + "," + str(key) + "\n")

elif select == "d":
    with open("deleteTest.csv", "w") as f:
        for i in range(1, 300001):
            f.write(str(keys[i]) + "\n")

else:
    print("Input case is only (i) or (d).")
```

다음과 같은 코드를 작성하여 테스트할 csv 파일을 만들었다.

Insert 파일은 1 에서 1000000 까지의 숫자를 무작위로 넣어주었다.

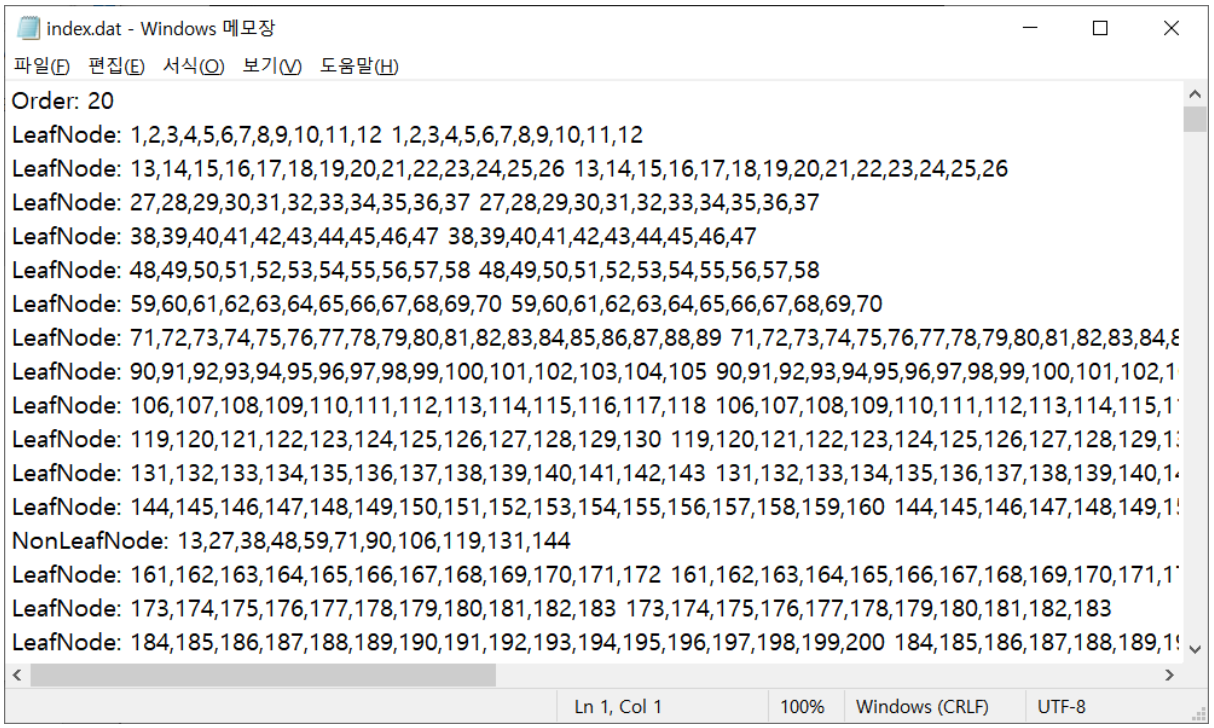
Delete 파일은 1 에서 1000000 까지의 숫자 중 300000 개를 무작위로 넣어주었다.

Insert 시 Key 와 Value 는 동일하게 넣어주었다.

```
C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>java -jar B-tree_Assignment.jar -c index.dat 20
C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>java -jar B-tree_Assignment.jar -i index.dat insertTest.csv
C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>java -jar B-tree_Assignment.jar -d index.dat deleteTest.csv
C:₩Users₩dyddn₩2020_ite2038_2019054957₩B-tree_Assignment>
```

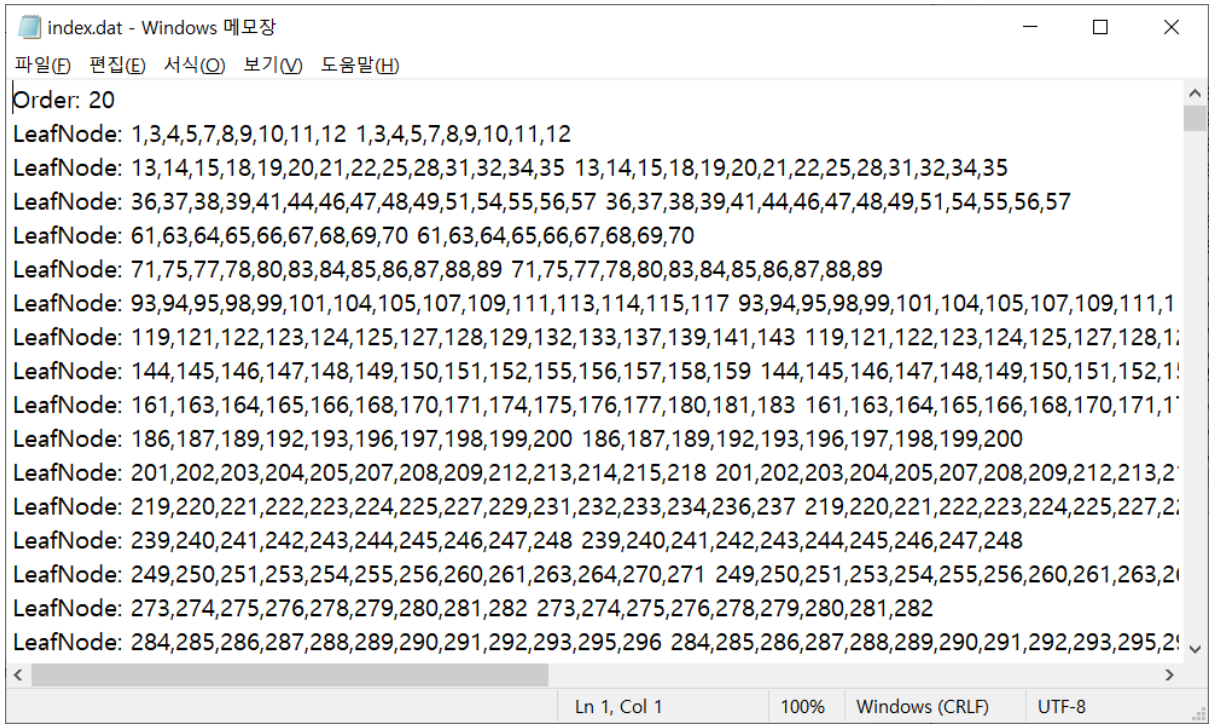
다음과 같이 명령어들을 입력하였다. (Create -> Insert -> Delete)

Degree 는 20 으로 설정했다.



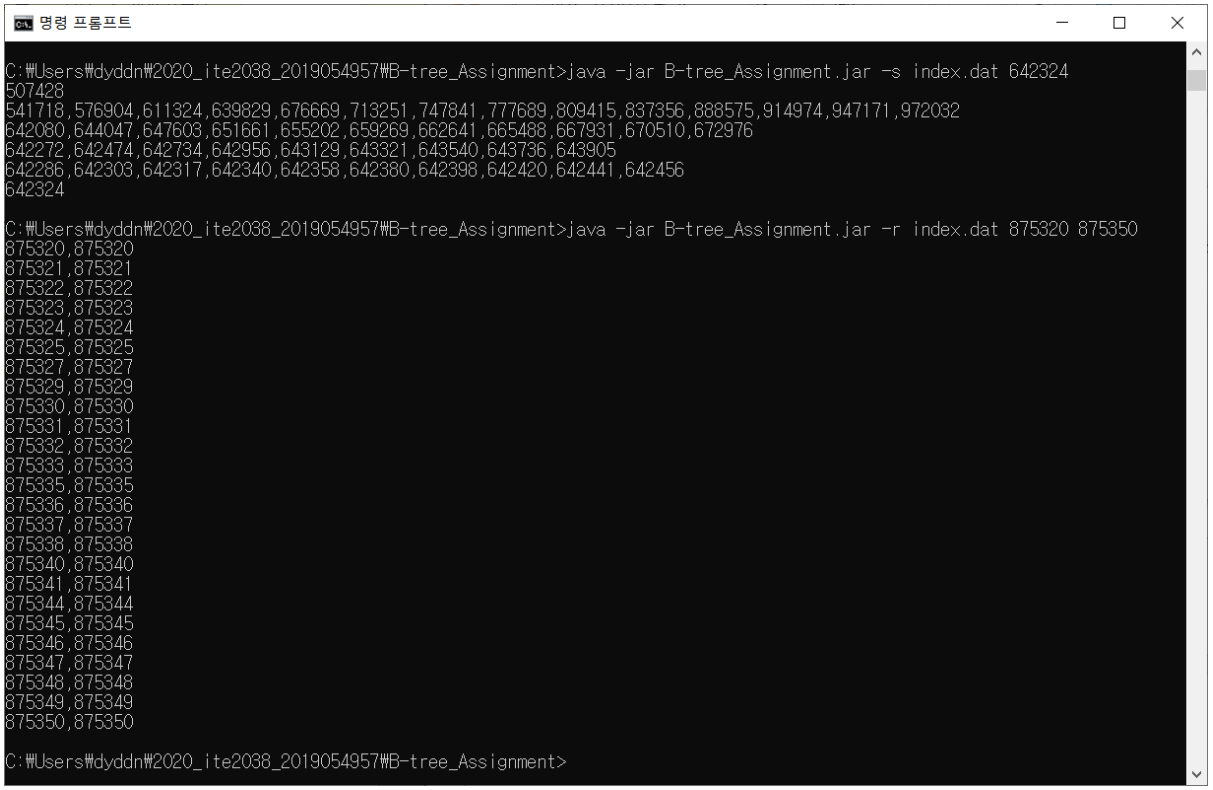
Insert 수행 시 index.dat

백만개의 데이터에 대하여 삽입을 수행했을 때 저사양 노트북 기준으로 5 초 정도가 소요되었다.



Delete 수행 시 index.dat

30 만개의 데이터를 삭제하여 몇몇 Key 들이 사라진 것을 확인할 수 있다.



임의의 숫자를 입력하여 Single Key Search 와 Ranged Search 를 수행하였다.

백만개라는 많은 데이터를 다룬 것이 처음인데, 빠른 시간내에 Value 들을 찾을 수 있었다.

Search 가 굉장히 빠르다는 점이 B+ Tree 의 강점임을 다시 한번 깨닫게 되었다.