

2020 B+Tree Implementation Assignment

2017029916

양동해

1. 주요 함수 및 코드 알고리즘

- Node.java

Node 클래스는 추상 클래스로 만들어져 있고, NonLeafNode와 LeafNode가 이를 상속했다.

```
5 public abstract class Node {  
6     int degree;  
7     ArrayList<Integer> keys; // 노드마다 가지고 있을 키가 보관될 변수  
8     int nISplitKey = -1; // NonLeafNode에서 split 될 때 사용하는 값  
9  
10    abstract Node[] insert(int key, int value);  
11    abstract int delete(int key, Node parentNode, Node tmpRoot);  
12    abstract void singleKeySearch(int findKey);  
13    abstract void rangedSearch(int startKey, int endKey);  
14  
15    abstract Node split();  
16    abstract int merge(Node leftSibling, Node rightSibling, Node parentNode);  
17  
18    abstract void print();  
19    abstract ArrayList<Node> getNode(ArrayList<Node> nodes);  
20 }  
21
```

degree는 트리의 degree를 뜻하고 keys는 노드의 key들이 들어있는 arraylist이다. NonLeafNode와 LeafNode 모두 key들을 가지고 있기 때문에 Node 클래스 내에 선언했다. nISplitKey는 NonLeafNode에서 split되었을 때, 부모노드로 올려주어야 할 key 값을 임시로 갖고 있는 변수이다. 나머지 함수들은 LeafNode와 NonLeafNode에서 각각 자세히 설명할 예정이다.

- BplusTree.java

BplusTree 클래스에는 degree와 root를 저장하는 변수가 선언되어 있고, 함수로는 insert(), delete(), singleKeySearch(), rangedSearch()가 각각 구현되어 있다. 함수 명에서 보여지는 것처럼 각각의 함수들은 삽입, 삭제, 단일 키 탐색, 범위 탐색을 제공한다.

```
9 public class BplusTree {  
10    int degree;  
11    Node root;  
12  
13    BplusTree(int degree){  
14        this.degree = degree;  
15        this.root = new LeafNode(degree); // root는 처음에 리프노드  
16    }  
17  
18    void insert(int key, int value) {  
19        if(root.getClass() == LeafNode.class) { // root가 LeafNode일 때  
20            Node[] nodes = root.insert(key, value);  
21            if(nodes[1] != null) { // root 노드 -> NonLeafNode로 승격  
22                NonLeafNode newRoot = new NonLeafNode(degree);  
23  
24                int tmpKey = ((LeafNode)nodes[1]).keys.get(0); // 오른쪽 노드의 맨 왼쪽 값 가져옴  
25  
26                newRoot.keys.add(tmpKey);  
27                newRoot.childNodes.add(nodes[0]);  
28                newRoot.childNodes.add(nodes[1]);  
29  
30                root = newRoot;  
31            }  
32        } else { // root가 NonLeafNode일 때  
33            Node[] nodes = root.insert(key, value);  
34            if(nodes[1] != null) { // root 노드 -> 위로 한칸 올라감  
35                NonLeafNode newRoot = new NonLeafNode(degree);  
36  
37                newRoot.keys.add(nodes[1].nISplitKey);  
38                newRoot.childNodes.add(nodes[0]);  
39                newRoot.childNodes.add(nodes[1]);  
40                nodes[1].nISplitKey = -1;  
41  
42                root = newRoot;  
43            }  
44        }  
45    }
```

우선 insert를 보면 key와 value를 인자로 받아 트리에 삽입하는 과정을 진행한다. root가 LeafNode인 경우와 NonLeafNode인 경우를 구분하여 삽입을 진행하며, nodes에는 바로 아래 자식 노드가 반환되어 들어간다. 자식이 split으로 분화 되었을 경우 nodes[1]에 들어가며, 자식이

분화되지 않고 올라온 경우는 nodes[0]에만 자식 노드가 들어가 있다. nodes[1]에 자식 노드가 있을 경우 새로운 root를 만들어 적절히 key와 childNodes에 넣어주어 B+Tree 구조를 유지한다.

```

46④    void delete(int key) {
47        int deleteIdx = root.delete(key, null, root); // root의 부모는 null, tmpRoot는 root
48
49        if(deleteIdx != -1) { // 삭제할 인덱스가 남아있는 상태
50            // 여기 들어온다면 끝은 리프노드면 리프노드에서 이미 삭제 진행하고 인덱스 남아있지 않으므로 무조건 논리프노드
51            NonLeafNode newRoot = (NonLeafNode) root;
52            newRoot.keys.remove(deleteIdx);
53
54            // 루트에 키가 있는 경우
55            if(newRoot.keys.size() == 0) root = newRoot.childNodes.get(0); // 맨 왼쪽 child 노드를 가지고 온 다음 그 노드를 루트로 줌
56            else root = newRoot;
57        }
58
59        if(root.keys.size() == 0) root = new LeafNode(degree); // 루트가 비어있는 경우
60
61        // 논리프노드에서 삭제되지 않은 키를 위해 삭제를 한번 더 진행하여 키 업데이트 (리프노드의 키 바꿔주는 알고리즘 이용하기 위해)
62        root.delete(key, null, root);
63    }
64
65    void singleKeySearch(int findKey) { root.singleKeySearch(findKey); }
66    void rangedSearch(int startKey, int endKey) { root.rangedSearch(startKey, endKey); }

```

delete에서는 key만 인자로 받아, 그 key를 트리에서 찾아서 삭제를 진행한다. root에서부터 delete를 진행하며 deletelidx는 NonLeafNode 중 삭제해야 할 키가 남아있으면 그 key의 인덱스를 주고, 나머지 코드를 통해 삭제를 진행한다. 삭제를 할 때 key.size()가 0인 경우는 아래 자식 노드에서 이미 merge된 경우이므로 그 자식을 root로 할당해 준다. 그런데 그 자식 마저 key가 없다는 뜻은 트리가 비었다는 뜻이므로 새로운 LeafNode를 만들어 root에 할당해 준다. 맨 마지막 줄에서 삭제를 한 번 더 진행하는 이유는 NonLeafNode에서 key 삭제가 제대로 진행되지 않았을 경우를 대비하여 한번 더 진행하는 것이다.(사실 이 과정은 굳이 하지 않아도 B+Tree 구조를 해치지 않는다)

singleKeySearch와 rangedSearch는 각각 root에서, Node 클래스에 정의되어 있던 함수들이 실행된다.

```

77④    void writeFile(String indexPath) {
78        // 저장방식
79        // 1. degree 저장
80        // 2. Node 갯수 저장
81        // 3. Leaf, NonLeaf 구분하여 각각 저장
82        // Leaf 저장해야할 정보: 0(리프노드), keys.size(), (키와, 벌류 각각) * 리스트 크기만큼, rSiblingNode가 가리키고 있는 노드의 index
83        // NonLeaf 저장해야할 정보: 1(논리프노드), keys.size(), (키와, childNodes에 있는 노드의 index 각각) * 리스트 크기만큼, childNodes의 맨 마지막 child의 index
84        DataOutputStream writer = null;
85
86        try {
87            writer = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(indexPath)));
88            writer.writeInt(degree); // 1. degree 정보 저장
89
90            ArrayList<Node> nodes = new ArrayList<Node>();
91            nodes = getNode(nodes); // 노드 이 ArrayList에 모두 삽입
92            writer.writeInt(nodes.size()); // 2. Node 갯수 저장
93
94            for(Node node: nodes) {
95                if(node.getClass() == LeafNode.class) { // 3-1. 리프노드
96                    writer.writeInt(0); // 리프노드
97                    LeafNode tmpNode = (LeafNode) node;
98                    writer.writeInt(tmpNode.keys.size()); // keys.size()
99                    for(int i=0; i<tmpNode.keys.size(); i++) {
100                        writer.writeInt(tmpNode.keys.get(i)); // 키
101                        writer.writeInt(tmpNode.values.get(i)); // 벌류
102                    }
103                    writer.writeInt(nodes.indexOf(tmpNode.rSiblingNode)); // rSiblingNode가 가리키고 있는 노드의 index
104                } else if(node.getClass() == NonLeafNode.class) { // 3-2. 논리프노드
105                    writer.writeInt(1); // 논리프노드
106                    NonLeafNode tmpNode = (NonLeafNode) node;
107                    writer.writeInt(tmpNode.keys.size()); // keys.size()
108                    for(int i=0; i<tmpNode.keys.size(); i++) {
109                        writer.writeInt(tmpNode.keys.get(i)); // 키
110                        writer.writeInt(nodes.indexOf(tmpNode.childNodes.get(i))); // childNodes에 있는 노드의 index
111                    }
112                    writer.writeInt(nodes.indexOf(tmpNode.childNodes.get(tmpNode.childNodes.size()-1))); // childNodes의 맨 마지막 child의 index
113                }
114            }
115        } catch (IOException e) {
116            System.out.println("BplusTree.writeFile ERROR");
117        } finally {
118            try {
119                writer.close();
120            } catch (IOException e) {
121                System.out.println("BplusTree.writeFile.writer.close() ERROR");
122            }
123        }
124    }
125}

```

writeFile은 파일을 만들어주고, 그 파일에 트리의 정보를 저장하는 함수이다. 인자로 파일 이름을 받고 그 파일 명에 해당하는 파일을 만든다. 파일에는 degree와 전체 node 개수, 각 LeafNode와 NonLeafNode에 대한 정보가 저장된다. LeafNode의 rSiblingNode와 NonLeafNode의 childNodes는 그 노드가 가리키고 있는 노드의 인덱스를 저장한다. 이렇게 한 이유는, 파일을 불

러울 때 차례대로 노드를 arrayList에 넣어주고 다 넣어준 뒤, 각 위치에 있는 노드의 위치를 통해 인덱싱 작업을 하여 할당 해주기 위함이다. 파일에 저장하는 자세한 방식은 첫 줄부터 여섯 번째 줄까지 주석에 적혀 있다.

- NonLeafNode.java

NonLeafNode에서는 childNodes라는 arrayList 변수를 새로 정의하였다. 여기에는 자식 노드가 들어가기 때문에, key와 childNode가 따로 분리되어 저장된다. Right most child는 childNodes의 맨 마지막 값을 의미하므로 따로 변수로 설정하지 않았다.

```

5  public class NonLeafNode extends Node {
6      ArrayList<Node> childNodes; // childNode를, right most child는 이 리스트의 맨 끝 값
7
8=    NonLeafNode(int degree){
9        this.degree = degree;
10       this.keys = new ArrayList<Integer>();
11       this.childNodes = new ArrayList<Nodes>();
12   }
13
14=  Node[] insert(int key, int value) {
15      Node[] nodes = new Node[2];
16      nodes[0] = this;
17      nodes[1] = null; // spilt이 되면 여기에 객체 넣고 리턴
18
19      int index = 0;
20      for(; index < keys.size(); index++) {
21          if(key < keys.get(index)) break; // Node 내에서 값은 linear search
22          else if(key == keys.get(index)) return nodes; // 중복된 키
23      }
24
25      Node[] tmpNodes;
26      tmpNodes = childNodes.get(index).insert(key, value);
27
28      if(tmpNodes[1] != null && tmpNodes[1].getClass() == LeafNode.class) { // 리프에서 split 되었을 때
29          int tmpKey = ((LeafNode)tmpNodes[1]).keys.get(0);
30
31          keys.add(index, tmpKey);
32          childNodes.add(index+1, tmpNodes[1]);
33      } else if(tmpNodes[1] != null && tmpNodes[1].getClass() == NonLeafNode.class) { // non리프에서 split 되었을 때
34          int tmpKey = tmpNodes[1].nlSplitKey; // 위로 올라간 키 받아준 것 => 이걸로 NonLeafNode 키 만들어주면 됨
35          tmpNodes[1].nlSplitKey = -1;
36
37          keys.add(index, tmpKey);
38          childNodes.set(index, tmpNodes[0]);
39          childNodes.add(index+1, tmpNodes[1]);
40      }
41
42      if(keys.size() >= degree) nodes[1] = split(); // 노드가 꽉 차면 split!!
43
44      return nodes;
45  }

```

insert의 경우 nodes라는 배열을 하나 선언한 뒤 nodes[0]에 자기 자신을 할당 한다. 그리고 key를 찾아 insert를 진행한 뒤, 거기서 반환된 배열을 tmpNodes에 저장한다. tmpNodes[1]에 노드가 있다는 뜻은 자식 노드가 split되어 올라왔다는 뜻이다. tmpNodes[1]가 LeafNode인지 NonLeafNode인지를 각각 나누어 분석하고, 각각에 맞게 keys와 childNodes에 할당해준다.(이는 BplusTree의 insert와 과정이 비슷하다) 만약 자식노드에서 split(분화)되지 않았다면 자신의 상태를 분석해 spilt을 해야하면 진행하고 그렇지 않으면 nodes를 반환한다.

```

47=  int delete(int key, Node parentNode, Node tmpRoot) {
48      int parentIdx = 0; // 부모노드에서 child로 들어온 위치를 갖는 인덱스
49      if(parentNode != null) {
50          NonLeafNode tmpParent = (NonLeafNode) parentNode;
51          for(; parentIdx < tmpParent.childNodes.size(); parentIdx++) {
52              if(tmpParent.childNodes.get(parentIdx) == this) break;
53          }
54      }
55
56      int index = 0;
57      for(; index < keys.size(); index++) {
58          if(key < keys.get(index)) break; // Node 내에서 값은 linear search
59      }
60
61      int deleteIdx = -1; // 삭제해야할 인덱스가 남아있으면 여타 넣어줌
62      deleteIdx = childNodes.get(index).delete(key, this, tmpRoot);
63
64      if(deleteIdx != -1) { // 삭제할 인덱스가 남아있다면
65          if(this == tmpRoot) return deleteIdx; // 현재 노드가 루트면 루트에서 삭제
66
67          keys.remove(deleteIdx); // 키 삭제
68
69          if(keys.size() < degree/2) { // 키가 부족하면 merge!!
70              NonLeafNode siblingNode;
71
72              if(parentIdx > 0) { // 원쪽 sibling이 있는 경우
73                  siblingNode = (NonLeafNode) ((NonLeafNode)parentNode).childNodes.get(parentIdx-1);
74                  return merge(siblingNode, this, parentNode);
75              } else { // 원쪽 sibling이 없는 경우 => 오른쪽 sibling
76                  siblingNode = (NonLeafNode) ((NonLeafNode)parentNode).childNodes.get(parentIdx+1);
77                  return merge(this, siblingNode, parentNode);
78              }
79          }
80      }
81
82      return -1;
83  }

```

delete에서는 key뿐만 아니라 parentNode와 tmpRoot라는 변수들을 인자로 받는데, parentNode는 부모노드이고 tmpRoot는 root노드이다. parentIdx는 부모노드에서 자신이 위치해있는 인덱스를 갖고 있는 변수이다. 삭제하고자 하는 key의 위치를 따라 삭제를 진행하고 deleteIdx에 반환 값을 넣어둔다. deleteIdx는 BplusTree에서 본 것처럼 삭제해야 할 key가 남아있을 경우 그 인덱스를 넣어두는 변수이다. 삭제해야 할 인덱스가 남아있다면 삭제를 진행하고 언더플로우 여부를 확인한다. 언더플로우가 나면 각 위치에 맞게 왼쪽 형제노드 혹은 오른쪽 형제노드랑 merge를 진행한다.

```

90    void singleKeySearch(int findKey) {
91        int index = 0;
92        for(int i=0; i<keys.size(); i++) {
93            System.out.print(keys.get(i)); // 키 값들 출력
94            if(i+1 != keys.size()) System.out.print(", ");
95            else System.out.println();
96        }
97        if(findKey >= keys.get(i)) index++; // 찾고자 하는 위치 index 설정
98    }
99
100    childNodes.get(index).singleKeySearch(findKey);
101}
102
103    void rangedSearch(int startKey, int endKey) {
104        int index = 0;
105        for(; index<keys.size(); index++) {
106            if(startKey < keys.get(index)) break; // Node 내에서 같은 linear search
107        }
108        childNodes.get(index).rangedSearch(startKey, endKey);
109    }
110}

```

singleKeySearch에서는 현재 노드의 키들을 출력해주며, 출력해주는 동시에 들어가야 할 자식 노드의 index를 찾고 그 자식 노드에서 다시 한번 singleKeySearch를 진행한다.

rangedSearch에서는 들어가야 할 자식 노드의 index를 찾고 그 자식 노드에서 다시 한번 rangedSearch를 진행한다.

```

112    Node split() {
113        NonLeafNode newNode = new NonLeafNode(degree);
114
115        int index = degree/2 + 1;
116        int count = degree - index; // 반복 횟수
117
118        for(int i=0; i<count; i++) {
119            int tmpKey = this.keys.get(index);
120            Node tmpChild = this.childNodes.get(index);
121
122            newNode.keys.add(tmpKey);
123            newNode.childNodes.add(tmpChild);
124
125            this.keys.remove(index);
126            this.childNodes.remove(index);
127        }
128
129        newNode.childNodes.add(this.childNodes.remove(index)); // 새로만든 오른쪽 노드에 맨 오른쪽 끝 child 추가
130
131        newNode.nlSplitKey = this.keys.get(index-1); // 위로 올려줄 key값
132        this.keys.remove(index-1);
133
134        return newNode;
135    }

```

split은 단순히 노드를 두개로 분리해주는 역할을 한다. 분리해준 노드는 newNode이며 새로 분리된 노드를 반환해준다. NonLeafNode에서는 분리될 때 부모 노드로 key를 옮겨주어야 하는데 그 key를 nlSplitKey에 임시로 넣어준다.

```

137    int merge(Node leftSibling, Node rightSibling, Node parentNode) {
138        NonLeafNode leftNode = (NonLeafNode) leftSibling;
139        NonLeafNode rightNode = (NonLeafNode) rightSibling;
140
141        int parentIdx = 0;
142        for(; parentIdx < parentNode.keys.size(); parentIdx++) { // 부모노드의 인덱스 찾기
143            NonLeafNode tmpParent = (NonLeafNode) parentNode;
144            if(tmpParent.childNodes.get(parentIdx) == leftNode && tmpParent.childNodes.get(parentIdx+1) == rightNode) break;
145        }
146
147        int nlMergeKey = parentNode.keys.get(parentIdx); // merge 한 뒤에 부모노드에서 자식으로 내려줄 키
148
149        if(leftNode.keys.size() + rightNode.keys.size() < degree) { // merge 가능
150            leftNode.keys.addAll(rightNode.keys);
151            leftNode.childNodes.addAll(rightNode.childNodes);
152
153            NonLeafNode tmpParent = (NonLeafNode) parentNode;
154            tmpParent.childNodes.remove(tmpParent.childNodes.indexOf(rightNode)); // 부모노드 자식 중 rightNode 삭제
155
156            if(leftNode.keys.size() >= degree) { // 여기서 오버플로우나면 split 해주어야 함
157                Node newNode = leftNode.split();
158
159                int tmpKey = newNode.nlSplitKey;
160                newNode.nlSplitKey = -1;
161
162                int tmpIdx = ((NonLeafNode)parentNode).childNodes.indexOf(leftNode);
163                ((NonLeafNode)parentNode).keys.add(tmpIdx, tmpKey);
164                ((NonLeafNode)parentNode).childNodes.add(tmpIdx+1, newNode);
165
166                return parentIdx+1;
167            }
168
169            return parentIdx;
170        } else { // merge 불가능
171            // merge 불가능하면 왼쪽이나 오른쪽에서 키, 차일드 하나 주는 것으로 해결!!
172        }
173    }

```

merge는 크게 두 가지를 수행한다. 첫째로 merge가 가능한 경우 merge를 수행하고(사진의 149 번째 줄), 다른 한 가지는 merge가 불가능 할 경우 왼쪽이나 오른쪽 형제노드에서 key와 child를 하나씩 가져오는 기능(사진의 171번째 줄)이다. 우선 parentIdx는 delete에서 말한 것처럼 부모노드 기준으로 자신의 위치를 의미하지만, 여기서는 부모의 keys 중에 삭제 되어야 할 key의 위치를 나타내기도 한다. nlMergeKey는 merge할 때 부모에서 자식으로 내려줄 키를 의미한다.(정확히 nlSplitKey와 반대되는 역할이다) merge가 가능한 경우 merge를 진행해주며, merge를 했는데 오버플로우가 발생하면 다시 split을 진행해준다.

```

171     } else { // merge 불가능
172         // merge가 불가능하면 왼쪽이나 오른쪽에서 키, 차일드 하나 주는 것으로 해결!
173         if(leftNode.keys.size() < degree/2) { // leftNode가 언더플로우
174             // 오른쪽에서 키, 차일드 하나빼고 왼쪽에다 증
175             int tmpKey = rightNode.keys.get(0);
176             Node tmpChild = rightNode.childNodes.get(0);
177
178             leftNode.keys.add(nlMergeKey);
179             leftNode.childNodes.add(tmpChild);
180
181             rightNode.keys.remove(0);
182             rightNode.childNodes.remove(0);
183
184             parentNode.keys.set(parentIdx, tmpKey); // 부모노드 키 업데이트
185         } else if(rightNode.keys.size() < degree/2) { // rightNode가 언더플로우
186             // 왼쪽에서 키, 차일드 하나빼고 오른쪽에다 증
187             int tmpKey = leftNode.keys.get(leftNode.keys.size()-1);
188             Node tmpChild = leftNode.childNodes.get(leftNode.childNodes.size()-1);
189
190             rightNode.keys.add(0, nlMergeKey);
191             rightNode.childNodes.add(0, tmpChild);
192
193             leftNode.keys.remove(leftNode.keys.size()-1);
194             leftNode.childNodes.remove(leftNode.childNodes.size()-1);
195
196             parentNode.keys.set(parentIdx, tmpKey); // 부모노드 키 업데이트
197         }
198     }
199
200     return -1;

```

else 이하 부분은 merge를 하지 못할 경우이다. 앞에서 언급했듯 왼쪽 형제 노드에서 key와 child를 가져올 수 있으면 거기서 가져오고 아닌 경우 오른쪽 형제 노드에서 가져온다. 왼쪽 노드에선 맨 끝 값을 가져오고, 오른쪽 노드에선 맨 첫 값을 가져오는 차이 외엔 구현 방식은 완벽히 동일하다.

- LeafNode.java

LeafNode에서는 values라는 arraylist 변수를 새로 정의하였다. 여기에는 value가 들어가기 때문에, key와 value가 따로 분리되어 저장된다. Right sibling node는 rSiblingNode에 넣는다.

```

5  public class LeafNode extends Node {
6      ArrayList<Integer> values; // value
7      LeafNode rSiblingNode; // right sibling node
8
9      LeafNode(int degree){
10         this.degree = degree;
11         this.keys = new ArrayList<Integer>(); // 애내 둘(아래 values)은 갯수가 같아야함!
12         this.values = new ArrayList<Integer>();
13         this.rSiblingNode = null;
14     }
15
16     Node[] insert(int key, int value) {
17         Node[] nodes = new Node[2];
18         nodes[0] = this;
19         nodes[1] = null; // split이 되면 여기에 객체 넣고 리턴
20
21         int index = 0;
22         for(; index < keys.size(); index++) {
23             if(key < keys.get(index)) break; // Node 내에서 값은 linear search
24             else if(key == keys.get(index)) return nodes; // 중복된 키
25         }
26
27         keys.add(index, key); // 키와 뱍류 추가
28         values.add(index, value);
29
30         if(keys.size() >= degree) nodes[1] = split(); // 노드가 꽉 차면 split!!
31
32     }
33 }

```

insert의 전반적인 구조는 NonLeafNode와 동일하며, 현재 LeafNode이기 때문에 key와 value를 직접 추가한다. 그리고 오버플로우가 나면 split을 진행해준다.

```

  35◎ int delete(int key, Node parentNode, Node tmpRoot) {
  36     int parentIdx = 0; // 부모노드에서 child로 들어온 위치를 갖는 인덱스
  37     if(parentNode != null) {
  38         NonLeafNode tmpParent = (NonLeafNode) parentNode;
  39         for(; parentIdx < tmpParent.childNodes.size(); parentIdx++) {
  40             if(tmpParent.childNodes.get(parentIdx) == this) break;
  41         }
  42     }
  43
  44     int index = 0;
  45     for(; index < keys.size(); index++) {
  46         if(key == keys.get(index)) {
  47             keys.remove(index); // 키와 벨류 삭제
  48             values.remove(index);
  49             break;
  50         }
  51     }
  52
  53     if(this != tmpRoot && keys.size() < degree/2) { // 루트노드가 아닐때 언더플로우
  54         LeafNode siblingNode;
  55
  56         if(parentIdx > 0) { // 원쪽 sibling이 있는 경우
  57             siblingNode = (LeafNode) ((NonLeafNode)parentNode).childNodes.get(parentIdx-1);
  58             return merge(siblingNode, this, parentNode);
  59         } else { // 원쪽 sibling이 없는 경우 -> 오른쪽 sibling
  60             siblingNode = (LeafNode) ((NonLeafNode)parentNode).childNodes.get(parentIdx+1);
  61             return merge(this, siblingNode, parentNode);
  62         }
  63     } else if(keys.size() > 0) { // 사실상 나머지 경우
  64         // 노리프노드에서 키가 바뀌지 않은 경우 고려해 키 업데이트
  65         Node tmpNode = tmpRoot;
  66         int afterKey = keys.get(0);
  67
  68         while(tmpNode != null) {
  69             if(tmpNode.getClass() == LeafNode.class) return -1;
  70             NonLeafNode thisNode = (NonLeafNode) tmpNode;
  71
  72             int tmpIndex = 0;
  73             for(; tmpIndex < thisNode.keys.size(); tmpIndex++) {
  74                 if(key < thisNode.keys.get(tmpIndex)) break;
  75                 else if (key == thisNode.keys.get(tmpIndex)) {
  76                     thisNode.keys.set(tmpIndex, afterKey);
  77                     return -1;
  78                 }
  79             }
  80             tmpNode = thisNode.childNodes.get(tmpIndex);
  81         }
  82     }
  83
  84     return -1;
  85 }
  86 }
```

delete의 경우도 마찬가지로 NonLeafNode와 구조적으로 비슷하다. 현재 LeafNode이기 때문에 삭제가 직접적으로 이루어지며, 언더플로우 발생시 형제노드와 merge를 진행한다. else if 부분은 NonLeafNode에서 key가 제대로 바뀌지 않을 경우를 대비하여 넣어둔 코드이다. 전반적인 흐름은 인자로 주었던 tmpRoot, 즉 root에서부터 삭제된 key를 찾고 그 key가 NonLeafNode의 key로 남아있다면 현재 리프 노드의 첫번째 key로 바꾸어주는 역할을 한다.

```

  88◎ void singleKeySearch(int findKey) {
  89     for(int i=0; i<keys.size(); i++) {
  90         if(keys.get(i) == findKey) {
  91             System.out.println(values.get(i));
  92             return;
  93         }
  94     }
  95
  96     System.out.println("NOT FOUND"); // 키가 없으면 이 문장 출력
  97 }
  98
  99◎ void rangedSearch(int startKey, int endKey) {
 100     LeafNode tmpNode = this; // 현재 노드
 101
 102     while(tmpNode != null) {
 103         for(int i=0; i<tmpNode.keys.size(); i++) {
 104             if(tmpNode.keys.get(i) >= startKey && tmpNode.keys.get(i) <= endKey) System.out.println(tmpNode.keys.get(i) + ", " + tmpNode.values.get(i))
 105             else if(tmpNode.keys.get(i) > endKey) return;
 106         }
 107         tmpNode = tmpNode.rSiblingNode; // 현재 노드 다 보고나서 다음 노드로 이동
 108     }
 109 }
```

singleKeySearch에서는 key를 찾고 그 key에 해당하는 value를 출력해주며, 찾고자하는 key가 없다면 NOT FOUND를 출력해준다.

rangedSearch에서는 startKey와 endKey 사이에 해당하는 key와 value를 모두 출력해준다. 다음 노드로 이동할 때는 rSiblingNode를 이용한다.

```

 111◎ Node split() { // 현재 노드를 두 개로 나눈 후, 새로 만든 오른쪽 노드를 반환
 112     LeafNode newNode = new LeafNode(degree);
 113
 114     int index = degree/2;
 115     int count = degree - index; // 반복 횟수
 116     for(int i=0; i<count; i++) {
 117         newNode.keys.add(this.keys.get(index));
 118         newNode.values.add(this.values.get(index));
 119
 120         this.keys.remove(index);
 121         this.values.remove(index);
 122     }
 123
 124     newNode.rSiblingNode = this.rSiblingNode; // 현재 노드의 오른쪽 형제노드를 새로 만든 노드의 오른쪽 형제 노드로 줌
 125     this.rSiblingNode = newNode; // 새로 만든 노드를 현재 노드의 오른쪽 형제노드로 함
 126
 127     return newNode;
 128 }
```

split은 단순히 노드를 두개로 분리해주는 역할을 한다. 분리해준 노드는 newNode이며 새로 분리된 노드를 반환해준다.

```

130     int merge(Node leftSibling, Node rightSibling, Node parentNode) {
131         LeafNode leftNode = (LeafNode) leftSibling;
132         LeafNode rightNode = (LeafNode) rightSibling;
133
134         int parentIdx = 0;
135
136         if(leftNode.keys.size() + rightNode.keys.size() < degree) { // merge 가능 -> 왼쪽 노드에 다 떠려넣음
137             leftNode.keys.addAll(rightNode.keys);
138             leftNode.values.addAll(rightNode.values);
139             leftNode.rSiblingNode = rightNode.rSiblingNode;
140
141             parentIdx = ((NonLeafNode)parentNode).childNodes.indexOf(rightNode);
142             ((NonLeafNode)parentNode).childNodes.remove(rightNode); // 부모에서 right child 삭제
143
144             return parentIdx - 1;
145         } else { // merge 불가능
146             // merge가 불가능하면 왼쪽이나 오른쪽에서 키, 블루 하나 주는 것으로 해결!!
147             if(leftNode.keys.size() < degree/2) { // 왼쪽노드가 인더플로우
148                 // 오른쪽에서 키, 블루 하나빼고 왼쪽에다 줌
149                 int tmpChildIdx = ((NonLeafNode)parentNode).childNodes.indexOf(rightNode);
150
151                 int tmpKey = rightNode.keys.get(0);
152                 int tmpValue = rightNode.values.get(0);
153
154                 leftNode.keys.add(tmpKey);
155                 leftNode.values.add(tmpValue);
156
157                 rightNode.keys.remove(0);
158                 rightNode.values.remove(0);
159
160                 parentNode.keys.set(tmpChildIdx-1, ((NonLeafNode)parentNode).childNodes.get(tmpChildIdx).keys.get(0)); // 부모노드 키 업데이트
161             } else { // 오른쪽 노드가 인더플로우
162                 // 왼쪽에서 키, 블루 하나빼고 오른쪽에다 줌
163                 int tmpChildIdx = ((NonLeafNode)parentNode).childNodes.indexOf(rightNode);
164
165                 int tmpKey = leftNode.keys.get(leftNode.keys.size()-1);
166                 int tmpValue = leftNode.values.get(leftNode.values.size()-1);
167
168                 rightNode.keys.add(0, tmpKey);
169                 rightNode.values.add(0, tmpValue);
170
171                 leftNode.keys.remove(leftNode.keys.size()-1);
172                 leftNode.values.remove(leftNode.values.size()-1);
173
174                 parentNode.keys.set(tmpChildIdx-1, ((NonLeafNode)parentNode).childNodes.get(tmpChildIdx).keys.get(0)); // 부모노드 키 업데이트
175             }
176         }
177
178         return -1;
179     }

```

merge는 NonLeafNode와 마찬가지로 두 가지 기능을 수행하며, NonLeafNode에서 설명한 기능 까지 거의 똑같다. 몇 가지 변수가 조금 다를 수 있으나 특별히 추가된 기능도 없고 split되는 기능도 없기 때문에 NonLeafNode의 설명을 바탕으로 나머지 내용은 생략한다.

- Main.java

main함수와 readInsertFile(), readDeleteFile()의 경우 매우 직관적인 코드 구조이기 때문에, 코드 일부 사진로 자세한 설명은 생략한다.

```

12 public class Main {
13
14     public static void main(String[] args) {
15         if(args.length < 3) return;
16
17         String cmd = args[0];
18         String indexFile = args[1];
19
20         if(cmd.equals("-c")) { // Creation
21             BplusTree bpt = new BplusTree(Integer.parseInt(args[2]));
22             bpt.writeFile(indexFile);
23         } else if(cmd.equals("-i")) { // Insertion / args[2] = data_file
24             BplusTree bpt = readFile("index.dat"); // 파일 읽어오기
25             ArrayList<Pair> insertArr = readInsertFile(args[2]);
26             for(Pair pair : insertArr) {
27                 bpt.insert(pair.key, pair.value);
28             }
29             bpt.writeFile(indexFile);
30         } else if(cmd.equals("-d")) { // deletion / args[2] = data_file
31             BplusTree bpt = readFile("index.dat"); // 파일 읽어오기
32             ArrayList<Integer> deleteArr = readDeleteFile(args[2]);
33             for(Integer key : deleteArr) {
34                 bpt.delete(key);
35             }
36             bpt.writeFile(indexFile);
37         } else if(cmd.equals("-s")) { // single key search / args[2] = key
38             BplusTree bpt = readFile("index.dat"); // 파일 읽어오기
39             bpt.singleKeySearch(Integer.parseInt(args[2]));
40         } else if(cmd.equals("-r")) { // ranged search / args[2] = start_key, args[3] = end_key
41             BplusTree bpt = readFile("index.dat"); // 파일 읽어오기
42             bpt.rangedSearch(Integer.parseInt(args[2]), Integer.parseInt(args[3]));
43         }
44     }

```

```

134@ public static ArrayList<Pair> readInsertFile(String dataFile) {
135    ArrayList<Pair> insertArr = new ArrayList<Pair>();
136    BufferedReader reader = null;
137
138    try {
139        reader = new BufferedReader(new FileReader(new File(dataFile)));
140        String line = "";
141
142        while ((line = reader.readLine()) != null) { // key, value 저장
143            String[] token = line.split(",");
144            insertArr.add(new Pair(Integer.parseInt(token[0]), Integer.parseInt(token[1])));
145        }
146
147        reader.close();
148    } catch (IOException e) {
149        System.out.println("Main.readFileInsert ERROR");
150    }
151
152    return insertArr;
153}
154
155@ public static ArrayList<Integer> readDeleteFile(String dataFile) {
156    ArrayList<Integer> deleteArr = new ArrayList<Integer>();
157    BufferedReader reader = null;
158
159    try {
160        reader = new BufferedReader(new FileReader(new File(dataFile)));
161        String line = "";
162
163        while ((line = reader.readLine()) != null) { // key 저장
164            deleteArr.add(Integer.parseInt(line));
165        }
166
167        reader.close();
168    } catch (IOException e) {
169        System.out.println("Main.readFileInsert ERROR");
170    }
171
172    return deleteArr;
173}
174
175}

```

아래에 나올 readFile()은 writeFile()을 통해 작성된 파일을 읽어오는 코드이다.

```

46@ public static BplusTree readFile(String indexPath) {
47    BplusTree bpt = null;
48    DataInputStream reader = null;
49
50    try {
51        reader = new DataInputStream(new BufferedInputStream(new FileInputStream(indexPath)));
52
53        ArrayList<Node> nodes = new ArrayList<Node>();
54        int degree = reader.readInt(); // 1. degree 정보 읽기
55        bpt = new BplusTree(degree);
56        int size = reader.readInt(); // 2. 노드 갯수 읽기
57
58        // 각 노드별 인덱스 값 저장
59        // degree + 2은 (NonLeaf(-1)와 Leaf(rSiblingNode) 구분, [node index ...](NonLeaf만))
60        int[][] tmpArr = new int[size][degree+1];
61
62        for(int i=0; i<size; i++) {
63            int check = reader.readInt();
64            if(check == 0) { // LeafNode
65                LeafNode tmpNode = new LeafNode(degree);
66                int count = reader.readInt(); // 반복 횟수 (keys.size())
67
68                for(int j=0; j<count; j++) {
69                    int tmpKey = reader.readInt();
70                    int tmpValue = reader.readInt();
71
72                    tmpNode.keys.add(tmpKey);
73                    tmpNode.values.add(tmpValue);
74                }
75
76                tmpArr[i][0] = reader.readInt(); // rSiblingNode이 가리키는 Node의 index
77
78                nodes.add(tmpNode);
79            } else if(check == 1) { // NonLeafNode
80                NonLeafNode tmpNode = new NonLeafNode(degree);
81                int count = reader.readInt(); // 반복 횟수 (keys.size())
82                tmpArr[i][0] = count; // NonLeaf와 Leaf 구분
83
84                for(int j=1; j<=count; j++) {
85                    int tmpKey = reader.readInt();
86                    int tmpNodeIndex = reader.readInt();
87
88                    tmpNode.keys.add(tmpKey);
89                    tmpArr[i][j] = tmpNodeIndex; // childNodes가 가리키는 Node의 index
90                }
91
92                tmpArr[i][count+1] = reader.readInt(); // childNodes의 맨 마지막 child가 가리키는 Node의 index
93
94                nodes.add(tmpNode);
95            }
96        }
97    }

```

```

98      // 인덱싱 작업
99      int index = 0;
100     for(Node node: nodes) {
101         if(node.getClass() == LeafNode.class) { // LeafNode
102             LeafNode tmpNode = (LeafNode) node;
103
104             if(index+1 == nodes.size()) continue; // 맨 마지막 노드 -> 예는 무조건 리프노드에 rSiblingNode가 없음
105             tmpNode.rSiblingNode = (LeafNode) nodes.get(tmpArr[index][0]); // rSiblingNode 인덱싱
106         } else if(node.getClass() == NonLeafNode.class) { // NonLeafNode
107             NonLeafNode tmpNode = (NonLeafNode) node;
108
109             int count = tmpArr[index][0];
110             for(int i=1; i<=count; i++) {
111                 Node tmpChildNode = nodes.get(tmpArr[index][i]);
112                 tmpNode.childNodes.add(i-1, tmpChildNode); // childNodes가 가리키는 Node 인덱싱
113             }
114             tmpNode.childNodes.add(nodes.get(tmpArr[index][count+1])); // rChildNode 인덱싱
115         }
116     }
117     index++;
118 }
119
120     bpt.root = nodes.get(0); // root 인덱싱
121 } catch (IOException e) {
122     System.out.println("Main.readFile ERROR");
123 } finally {
124     try {
125         reader.close();
126     } catch (IOException e) {
127         System.out.println("Main.readFile.reader.close() ERROR");
128     }
129 }
130
131     return bpt;
132 }

```

우선 nodes라는 arraylist와, tmpArr라는 2차원 배열을 선언했다. 파일을 읽을 때마다 노드는 arraylist에 하나씩 추가해주고, tmpArr에는 나중에 인덱스를 맞춰주기 위해 할당 받아야 할 노드들의 인덱스를 저장해준다. 사진의 99번째 줄부터 각 노드의 인덱스 정보를 보고 nodes에서 찾아서, childNode와 rSiblingNode에 알맞는 노드들을 할당해준다.

- 부가적인 클래스 및 함수

위에 설명한 클래스 외에도 Pair라는 클래스도 있고 NonLeafNode와 LeafNode 등에 있는 getNode(), print() 등 여러 함수가 있지만 필수적인 핵심 코드가 아니므로 생략했다. 자세한 내용은 전체 코드에서 확인할 수 있다.

2. 실행방법

- 컴파일을 통해 실행하기

자바 버전은 14.0.2를 사용했다. 프로그램을 실행하기 위해선 모든 파일들이 다 같은 디렉토리 내에 존재해야 한다. java파일을 컴파일하여 만든 클래스들과 input.csv, delete.csv 역시 같은 디렉토리 내에 존재해야 한다.

```

eastsea@Eastui-MacBookPro Source % ls
BplusTree.java      Main.java          NonLeafNode.java
LeafNode.java       Node.java          Pair.java
eastsea@Eastui-MacBookPro Source % javac BplusTree.java Main.java NonLeafNode.java LeafNode.java Node.java Pair.java
eastsea@Eastui-MacBookPro Source % java Main -c index.dat 8
eastsea@Eastui-MacBookPro Source % ls
BplusTree.class    LeafNode.java      Node.class        NonLeafNode.java
BplusTree.java      Main.class       Node.java        NonLeafNode.class
LeafNode.class      Main.java       NonLeafNode.class  Pair.java

```

먼저 Source 디렉토리안에 존재하는 BplusTree.java, Main.java, NonLeafNode.java, LeafNode.java, Node.java, Pair.java를 컴파일 하여 .class 파일을 생성한다. 컴파일 하는 방법은 “javac [java file] [...]” 식으로 진행하며, 자세한 명령어 사용법은 사진으로 첨부되어 있다.

그 뒤에는 Main 클래스를 실행하여 B+Tree를 만들 수 있다. 실행 방법은 “java Main [command line]”을 입력하여 실행하고 자세한 방법은 사진으로 첨부되어 있다.

```

eastsea@Eastui-MacBookPro Source % java Main -i index.dat input.csv

```

insert 및 delete, search 하는 방법 역시 모두 동일하며 command line 만 바꿔서 실행하면 된다. input.csv와 delete.csv를 사용하기 위해선 그 파일이 같은 디렉토리 내에 저장되어 있어야 한다. 자세한 구동 예시는 사진으로 첨부되어 있다.

```
eastsea@Eastui-MacBookPro Source % java Main -r index.dat 0 100
5, 1143
7, 16082
10, 4732
12, 26211
14, 57656
16, 78799
20, 1949
23, 72531
26, 56972
31, 4596
56, 70039
59, 20700
73, 47978
76, 68225
82, 85853
83, 50225
85, 23185
eastsea@Eastui-MacBookPro Source % java Main -s index.dat 1
191, 329, 540, 618, 857
16, 31, 76, 186, 144, 172
NOT FOUND
eastsea@Eastui-MacBookPro Source % java Main -s index.dat 5
191, 329, 540, 618, 857
16, 31, 76, 186, 144, 172
1143
```

위 사진은 input을 진행한 뒤 ranged search 와 single key search를 진행한 모습이다.

```
eastsea@Eastui-MacBookPro Source % java Main -d index.dat delete.csv
eastsea@Eastui-MacBookPro Source % java Main -r index.dat 0 100
5, 1143
7, 16082
26, 56972
82, 85853
83, 50225
85, 23185
```

다음으로는 delete를 진행하여 다시 ranged search를 진행한 모습이다.

- jar파일을 이용하여 실행하기

BplusTree.jar파일은 위 클래스 파일들을 하나로 합쳐 만든 파일이다. 이 역시 input.csv, delete.csv는 같은 디렉토리 내에 존재해야 한다.

```
eastsea@Eastui-MacBookPro 2020_ite2038_2017029916 % java -jar BplusTree.jar -c index.dat 3
eastsea@Eastui-MacBookPro 2020_ite2038_2017029916 % ls
B-tree_Assignment      BplusTree.jar      delete.csv      index.dat      input.csv
eastsea@Eastui-MacBookPro 2020_ite2038_2017029916 % java -jar BplusTree.jar -i index.dat input.csv
eastsea@Eastui-MacBookPro 2020_ite2038_2017029916 % java -jar BplusTree.jar -s index.dat 26
493, 709
167, 355
31, 106
10, 16
20, 26
56972
eastsea@Eastui-MacBookPro 2020_ite2038_2017029916 %
```

커맨드 라인으로 “java -jar BplusTree.jar [command line]” 을 입력하면 실행되고, 자세한 구동 예시는 사진으로 첨부되어 있다.

3. 기타 구현

- Print preorder

B+Tree 내에서 특별히 추가적인 기능을 하는 것은 아니며, 말 그대로 pre order 형식으로 노드 내의 key와 value를 출력해준다. 이는 B+Tree 구현 과정 중 진행 상태를 확인하기 위해 만든 test 함수 중 하나이다.

각 line은 노드 한 개를 뜻하고, NonLeafNode의 경우 [](대괄호)를 이용하여 키들을 각각 감싸서 출력해준다. LeafNode의 경우엔 노드에 들어 있는 key와 value를 (key, value) 형식으로 출력해주며

key와 value가 여러 개일 경우 옆에다 출력을 이어서 한다.

이와 관련한 코드는 위 Main.java 코드 사진엔 나타나 있지 않는데, 실제 코드 내에선 확인할 수 있다.

실행 방법은 command line에 -p [index file]을 입력하면 확인할 수 있다. 자세한 구동 예시는 아래 사진으로 첨부했다.

```
eastsea@eastui-MacBookPro Source % java Main -p index.dat
[191][329][549][618][857]
[16][31][76][106][144][172]
(5, 1143)(7, 16082)(10, 4732)(12, 26211)(14, 57656)
(16, 78799)(20, 1949)(23, 72531)(26, 56972)
(31, 4596)(56, 70039)(59, 20700)(73, 47978)
(76, 68225)(82, 85853)(83, 56225)(85, 23185)
(106, 83075)(112, 57784)(114, 86511)(122, 86707)(126, 35827)(133, 20881)(135, 46116)
(144, 12188)(152, 68277)(159, 62409)(163, 29466)(167, 90933)(170, 25770)
(172, 69025)(181, 76119)(182, 53371)(190, 82219)
[210][242][276]
(191, 77005)(195, 79225)(207, 28277)(208, 29409)
(210, 28708)(213, 8600)(218, 99522)(221, 88658)(226, 19353)(230, 75460)(241, 49715)
(242, 34244)(246, 25757)(254, 11551)(259, 37229)(268, 23345)
(276, 15748)(284, 40628)(300, 88082)(307, 62507)(320, 34706)(325, 12532)
[355][375][419][472][569]
(329, 98325)(336, 55226)(345, 32866)(351, 51539)
(355, 74570)(362, 44218)(365, 54017)(373, 93935)
(375, 26782)(377, 82878)(406, 84454)(408, 81885)(409, 19651)
(410, 66944)(415, 58822)(427, 63753)(453, 93532)(457, 66293)(462, 18770)(471, 19003)
(472, 18076)(473, 56041)(479, 9738)(493, 25239)(496, 39751)(505, 58)
(509, 22458)(514, 48381)(517, 47794)(519, 12214)(521, 56309)(522, 18888)
[567][584][601]
(540, 74757)(555, 74303)(558, 71299)(560, 45661)(563, 47556)
(567, 35781)(570, 46073)(573, 27137)(580, 47071)
(584, 88832)(597, 77934)(599, 86793)(600, 27033)
(601, 16381)(609, 84276)(612, 79144)(614, 73421)
[634][656][691][709][758][793][826]
(618, 15638)(619, 58758)(627, 54442)(628, 58167)(630, 79545)
(634, 7840)(639, 82223)(642, 29170)(648, 67162)(649, 97184)
(656, 48704)(668, 90191)(669, 29423)(676, 9950)(689, 52521)
(691, 6829)(693, 4954)(698, 22245)(696, 8518)(697, 56034)(703, 43739)
(709, 92118)(721, 47261)(727, 19678)(729, 38835)(746, 27957)(753, 31638)
(758, 81225)(768, 51568)(780, 3438)(782, 31318)(785, 41453)
(793, 40387)(794, 73468)(799, 79803)(800, 897)(809, 32492)(819, 16145)
(826, 99273)(828, 71447)(829, 50493)(838, 2470)(849, 5086)(851, 47739)
[881][903][921][935][947][958]
(857, 79821)(863, 82479)(869, 95407)(870, 32274)(872, 67719)(879, 57089)(880, 85519)
(881, 26747)(883, 30427)(884, 69060)(886, 902)(898, 59554)(901, 52999)
(903, 39873)(907, 69199)(910, 51514)(914, 46648)
```