B+ tree implementation assignment

컴퓨터소프트웨어학부 2021025205 강태욱

1. Summary of algorithm

A. loadTree, buildTree, saveTree

i. loadTree : DAT 파일에 저장된 데이터를 불러온다.

ii. buildTree : loadTree를 통해 불러온 데이터를 바탕으로 B+Tree를 구성한다.

iii. saveTree : B+Tree를 특정한 형식에 맞게 DAT 파일에 저장한다.

B. Data file creation

i. Command line을 통해 입력된 두번째 값을 정수형으로 바꾸어 M에 저장한다.

ii. saveTree method를 이용해 첫 줄에 M 값이 저장된 DAT 파일을 생성한다.

C. Insertion

i. loadTree method를 이용해 M 값과 DAT 파일에 저장된 B+Tree를 불러온다.

ii. CSV file에서 key와 value를 받아와 insertNode method에 넣는다.

iii. insertNode method

1. findOperateNode method를 이용해 key가 들어갈 leaf node를 찾는다.
2. B+Tree에 parameter로 받은 key 값이 존재하는지 확인하고, 만약 존재한다면 에러 메시지를 출력한 후 method를 종료한다.
3. findIndex method를 이용해 leafNode에 key가 들어갈 index를 찾는다.
4. Key와 value를 leafNode의 index 위치에 넣고, leafNode의 크기를 1 증가시킨다.
5. 만약 leafNode의 크기가 M보다 크다면 leafNode를 split한다.
   A. leafNode가 root라면 splitRoot method를 실행한다.
   B. leafNode가 root가 아니라면 splitLeaf method를 실행한다.

iv. splitRoot method

1. right node를 생성한 뒤 split할 값을 right에 넣는다.
2. leafNode에서 right에 들어간 값을 삭제한다.
3. parent node를 만들고 leafNode, right와 연결한다.

v. splitLeaf method

1. right node를 생성한 뒤 split할 값을 right에 넣는다.
2. leafNode에서 right에 들어간 값을 삭제한다.
3. 기존의 parent node에 split된 값을 삽입하고 node를 연결한다. 이때, parent의 크기가 M 이상이라면, splitNonLeaf method를 실행한다.

vi. splitNonLeaf method

1. split시 parent node에 들어가는 key와 value, split 전 그 key의 left_child_node를 따로 저장한다.
2. right node를 만들어 split 시 오른쪽으로 이동하는 값을 삽입힌다.

3. node에서 right node에 들어갈 값을 삭제한다.
4. node와 right의 r을 재설정하고, r의 left_child_node의 parent를 right로 설정한다.
5. right.r의 parent와 node.r의 parent를 재설정한다.
6. parent node를 새로 만드는 경우, 미리 저장한 key와 value를 parent node에 삽입한다. 그 후 B+Tree의 시작 부분을 parent로 설정한다.
7. parent node가 이미 존재하는 경우, parent node의 적절한 위치에 미리 저장한 key와 value를 삽입한다. 만약 값 삽입이 끝난 parent node의 크기가 M 이상이라면, 다시 parent node에 대해 splitNonLeaf method를 실행한다.

D. Deletion

i. loadTree를 이용해 B+Tree를 불러온다.

ii. CSV file에서 key를 불러와 deleteNode method에 넣는다.

iii. deleteNode method

1. key가 있는 leaf node와 그 node의 왼쪽 leaf node를 찾는다.
2. B+Tree에 key가 없으면 method를 종료한다.
3. leaf node에서 key를 제거한다.
4. 만약 제거된 key가 node의 가장 왼쪽 key라면, non-leaf node에서 key의 값을 key가 제거된 node의 가장 왼쪽 key의 값으로 대체한다.
5. node.m이 규칙에 맞는 크기라면 method를 종료한다.
6. 그렇지 않다면, node의 왼쪽 leaf node인 left와 오른쪽 leaf node인 right를 찾는다.
7. 만약 left의 key 하나를 빌려올 수 있다면, 빌려와 node에 넣는다.
8. 만약 right의 key 하나를 빌려올 수 있다면, 빌려와 node에 넣는다.
9. 그렇지 않다면, merge method를 실행해 node를 merge한다.

iv. merge method

1. node의 왼쪽 leaf node인 left와 오른쪽 leaf node인 right를 찾는다.
2. 만약 left가 null이 아니라면, node의 모든 pair를 left로 옮긴다. 그 후 부모 node에서 node와 연결된 pair를 삭제한다.
   A. 삭제 후 node의 parent가 root이고, node.parent에 pointer가 없다면 left를 root로 지정한다.
   B. 삭제 후 node.parent의 pointer 개수가 규칙보다 적다면, mergeParent method를 실행한다.
3. 만약 right가 null이 아니라면, node의 모든 pair를 right로 옮긴다. 그 후 부모 node에서 node와 연결된 pair를 삭제한다.
   A. 삭제 후 node의 parent가 root이고, node.parent에 pointer가 없다면 right을 root로 지정한다.
   B. 삭제 후 node.parent의 pointer 개수가 규칙보다 적다면, mergeParent method를 실행한다.

v. mergeParent method
    1. node의 왼쪽 leaf node인 left와 오른쪽 leaf node인 right를 찾는다.
    2. 만약 left가 null이 아닐 때
        A. left에 node.parent에 left와 연결된 pair를 left로 옮긴다.
        B. node의 pair를 left로 옮긴다.
        C. 만약 left.m이 M보다 크다면, splitNonLeaf method를 실행한다.
        D. 만약 left.parent가 root이고 pointer가 없다면, left를 root로 지정한다.
        E. 만약 left.parent의 pointer 개수가 규칙보다 적다면, mergeParent method를 실행한다.
    3. 만약 right가 null이 아닐 때
        A. node에 node.parent에 node와 연결된 pair를 node로 옮긴다.
        B. right의 pair를 node로 옮긴다.
        C. 만약 node.m이 M보다 크다면, splitNonLeaf method를 실행한다.
        D. 만약 node.parent가 root이고 pointer가 없다면, node를 root로 지정한다.
        E. 만약 node.parent의 pointer 개수가 규칙보다 적다면, mergeParent method를 실행한다.

E. Single key search

i. loadTree method를 이용해 B+Tree를 불러온다.
ii. root node가 leaf node인 경우 key에 해당하는 value를 찾아 출력하거나, 찾지 못할 시 “Not found”를 출력한다.
iii. leafNode에 도달할때까지 path에 탐색 과정에서 사용된 key를 저장한다.
iv. leafNode에 도달하면 path에 저장된 key를 출력한다.
    1. 만약 leafNode에서 key를 찾지 못했다면 “Not found”를 출력한다.
    2. leafNode에서 key를 찾았다면 해당 key에 맞는 value를 출력한다.

F. Ranged search

i. loadTree method를 이용해 B+Tree를 불러온다.
ii. B+Tree가 빈 경우 “Not found”를 출력한다.
iii. findOperateNode method를 통해 start_key에 맞는 node를 찾는다.
iv. start_key보다 큰 key 중 가장 작은 key를 index로 설정한다.
v. 만약 start_key보다 큰 key가 node에 없다면, node.r로 이동하고 index를 0으로 설정한다. 이때, 이동 전 node가 Tree의 맨 오른쪽 node이면 start_key가 Tree의 최댓값보다 큰 것이므로 “Not found”를 출력하고 프로그램을 종료한다.
vi. node에서 선형적으로 탐색한 key와 value를 출력한다. 이때 node의 맨 오른쪽 key에 접근한 후에는 그 오른쪽 node로 넘어서 key가 end_key 이하의 값일때까지 key와 value를 출력한다.

vii. start_key와 end_key 사이에 key가 없는 경우에는 “Not found”를 출력한다.

2. Detailed description of codes

A. Class Pair : Node에 저장된 각 key, value와 자식 노드를 저장하는 단위

B. Class Node : B+Tree의 각 node

i. private int m : node에 들어있는 key의 수

ii. private ArrayList<Pair> p : key, value, 자식 노드를 저장한 Pair의 list

iii. private Node r : node의 오른쪽 자식 노드 혹은 node의 오른쪽 leaf node

iv. private Node parent : node의 부모 노드

v. public Node getChildNode(int index) : node의 index번째 pair의 자식 노드를 구하는 함수

1. 만약 index가 해당 node에 들어있는 key의 수와 같다면, 해당 node의 자식 노드는 node.r이다.
2. 만약 그렇지 않다면, 해당 node의 자식 노드는 node의 index번째 pair의 left_child_node이다.

vi. private int m : node에 들어있는 pointer의 수

vii. private ArrayList<Pair> p : key, value, 자식 노드를 저장한 Pair의 list

viii. private Node r : node의 오른쪽 자식 노드 혹은 node의 오른쪽 leaf node

ix. private Node parent : node의 부모 노드

x. public Node getChildNode(int index) : node의 index번째 pair의 자식 노드를 구하는 함수

1. 만약 index가 해당 node에 들어있는 pair의 수와 같다면, 해당 node의 자식 노드는 node.r이다.
2. 만약 그렇지 않다면, 해당 node의 자식 노드는 node의 index번째 pair의 left_child_node이다.

xi. public void setChildNode(int index, Node node) : node의 index번째 pair의 자식 node를 node로 설정하는 함수

1. 만약 index가 해당 node에 들어있는 pair의 수와 같다면, 해당 node의 자식 node를 node로 설정한다.
2. 만약 그렇지 않다면, node의 index번째 pair의 left_child_node를 node로 설정한다.

xii. public Node findLeft() : node의 parent node에서 해당 node의 왼쪽에 있는 node를 찾는다.

xiii. public Node findRight() : node의 parent node에서 해당 node의 오른쪽에 있는 node를 찾는다.

C. public static Node bPlusTree : bPlusTree를 저장하는 객체

D. public static int M : B+Tree의 차수

E. public static BufferedReader br, public static BufferedWriter bw : 파일 입출력에 사용

F. public static void main(String[] args)

i. data file creation
   1. args[2]를 정수로 바꾸어 M에 저장한다.
   2. saveTree method를 이용해 (args[1]).dat 파일을 생성한다.

ii. insertion
   1. insert method를 이용해 삽입 수행 후 이를 DAT 파일에 저장한다.

iii. deletion
   1. delete method를 이용해 삭제 수행 후 이를 DAT 파일에 저장한다.

iv. single key search
   1. args[2]를 정수형으로 바꾼 후 single search를 수행한다.

v. ranged search
   1. args[2]와 args[3]을 정수형으로 바꾼 후 각각을 start_key, end_key로 하는 ranged search를 수행한다.

G. public static void loadTree(String index_file)

i. index_file의 이름을 갖는 file인 indexFile을 불러온다.

ii. BufferedReader를 이용해 indexFile로부터 숫자를 읽는다.

iii. indexFile의 첫번째 입력값을 M으로 설정한다.

iv. buildTree method를 실행하고 이를 bPlusTree로 할당한다.

v. 만약 bPlusTree가 null이라면, 이는 DAT 파일이 처음 만들어지는 경우를 의미한다. 따라서, bPlusTree에 Node를 할당해준다.

H. public static Node buildTree() : DFS 방식으로 저장된 DAT 파일의 값을 읽어 B+Tree 구성

i. BufferedReader를 이용해 data를 한줄씩 입력받는다.

ii. split method를 이용해 각각의 데이터를 " " 단위로 분리해 nodeInfo에 저장한다.

iii. node를 생성하고, nodeInfo[0]을 정수형으로 변환한 값을 m으로 설정한 뒤 node의 m을 m으로 설정한다.

iv. nodeInfo[1]을 정수형으로 변환해 isLeaf에 저장한다. isLeaf가 1이면 해당 node는 leaf node이고, isLeaf가 0이면 해당 node는 non-leaf node이다.

v. nodeInfo[0]과 nodeInfo[1]을 제외한 nodeInfo의 값을 ","을 기준으로 분리해 dataArr에 넣은 뒤 node.p에 저장한다.

vi. node가 non-leaf node일 때
   1. i가 0부터 m이 될때까지 다음 for문을 반복한다.
   2. Node child를 만들고 이에 buildTree method의 실행값을 저장시킨다.
      A. saveTree method에 대한 설명을 보면 위의 행위가 DFS 방식으로 저장된 Tree를 불러오는 과정이라는 것을 알 수 있다.
   3. setChildNode method를 이용해 child를 node의 자식 노드로 설정한다.
   4. child의 부모 노드를 node로 설정한다.
   5. node의 왼쪽 child의 가장 오른쪽 leaf node의 r을 node의 오른쪽 child의 가장 왼

쪽 leaf node로 설정한다.

vii. node가 leaf node일때

1. node의 child를 모두 nuII로 설정한다.

viii. node를 return한다.

I. public static void saveTree(String index_file)

i. index_file을 이름으로 갖는 dat파일인 indexFile 불러온다.

ii. indexFile의 첫 줄에 M을 입력한다.

iii. saveNode method를 실행해 B+Tree를 특정한 형식에 맞춰 DAT 파일에 저장한다.

J. public static void saveNode(Node node)

i. 만약 node가 nuII이거나 node에 아무런 pair가 들어가있지 않으면 method를 종료한다.

ii. indexFile에 node.m을 입력한다.

iii. node.getChildNode(0)을 통해 해당 node가 leaf node인지 확인한다. 만약 node.getChildNode(0)가 nuII이라면, 해당 node는 leaf node이다.

iv. node가 leaf node라면, indexFile에 1을 입력한다. 그렇지 않다면 0을 입력한다.

v. 해당 node의 pair의 값을 key,value 와 같은 형식으로 indexFile에 입력한다.

vi. 입력 과정에서 데이터는 " "을 기준으로 구분하도록 데이터 사이에 " "을 입력해준다.

vii. 만약 node가 non-leaf node라면, node의 child node를 왼쪽부터 오른쪽 순서대로 saveNode method의 parameter로 넘겨 저장한다. 즉, node 저장 후 child node를 recursive하게 저장하므로 DFS와 같은 스타일로 B+Tree가 저장된다.

viii. 원래 B+Tree의 index에선 non-leaf node에 value가 저장되지 않지만, 본 과제에서는 코드를 보다 간결하게 구현하기 위해 편의상 non-leaf node에도 value를 같이 저장했다. 다만, 아래에 나와있는 substituteKey method에서 볼 수 있듯 non-leaf node에서 value는 중요한 데이터가 아니므로 Tree의 구조에 변화가 생기더라도 non-leaf node에서는 key에 따라 value가 변화되지 않도록 코드를 작성하였다.

ix. 다음은 데이터 저장 후 DAT 파일의 모습의 예시이다.

```
3
2 0 2,2 3,3
1 1 1,1
2 1 2,2
3 1 3,3 4,4
```

K. public void static insert(String index_file, String data_file)

i. loadTree method를 이용해 B+Tree를 불러온다.

ii. BufferedReader를 이용해 csv 파일의 key와 value pair를 한 줄씩 읽는다.

iii. 읽은 key와 value pair를 insertNode method에 넣는다.

iv. CSV file을 모두 읽었다면 saveTree method를 이용해 B+Tree를 저장한다.

L. public static void insertNode(int key, int value)

i. findOperateNode를 이용해 key가 들어갈 leaf node를 찾아 leafNode에 저장한다.
ii. leafNode 안에 이미 key가 존재할 경우 에러 메세지를 출력하고 method를 종료한다.
iii. leafNode에 key가 들어갈 index를 찾는다.
iv. leafNode의 index 위치에 key와 value pair를 삽입한 후, leafNode.m을 1만큼 늘린다
v. M이 leafNode.m보다 큰 경우, 즉 split이 일어날 경우
    1. 만약 leafNode가 root라면, splitRoot method를 실행한다.
    2. 만약 그렇지 않다면, splitLeaf method를 실행한다.

M. public static Node findOperateNode(int key, String commad)

i. node에 root를 저장하고, 다음의 반복문을 수행한다.
ii. key가 node의 제일 오른쪽에 있어야 함을 나타내는 biggest 변수를 설정한다.
iii. 만약 node에 pointer가 없거나, node의 자식이 없다면 node를 return한다.
iv. 만약 command가 "leaf"이고 key가 node의 i번째 key보다 작은 상황을 찾으면 node를 node의 i번째 왼쪽 자식으로 설정한다. 그 후 biggest를 false로 설정하고 for문을 탈출한다.
v. 만약 command가 "leftLeaf"이고 key가 node의 i번째 key보다 작거나 같은 상황을 찾으면 node를 node의 i번째 왼쪽 자식으로 설정한다. 그 후 biggest를 false로 설정하고 for문을 탈출한다.
vi. 만약 biggest가 true라면 key가 node의 가장 오른쪽 값보다 크거나, 크거나 같은 상황이므로 node를 node.r로 설정한다.

N. public static int findIndex(Node node, int key)

i. 만약 node에 pointer가 없으면 index를 0으로 설정한다.
ii. 만약 node의 node.m-1번째 key가 key보다 크다면, 해당 node에 node보다 큰 key가 없으므로 index를 node.m으로 설정한다.
iii. 만약 위 상황이 모두 아니라면, node의 i번째 key가 key보다 작으면 index를 i로 정한다.
iv. index를 return한다.

O. public static void splitRoot(Node root) : root를 split

i. right라는 node를 만들고, divideNum을 M / 2로 설정한다.
ii. root의 divideNum부터 M-1번째 pair를 right에 넣는다.
iii. right에 넣은 pair를 root에서 삭제한다.
iv. parent node를 생성한다.
v. right의 첫번째 pair를 parent node에 넣는다.
vi. parent.r를 right로 설정하고, parent의 첫번째 left_child_node를 root로 설정한다.
vii. root와 right의 parent node를 parent로 설정한다.
viii. right.r을 root.r로 설정하고, root.r을 right로 설정한다.

ix. parent를 root(bPlusTree)로 지정한다.

P. public static void splitLeaf(Node leafNode) : leafNode를 split

i. right라는 node를 만들고, divideNum을 M / 2로 설정한다.

ii. leafNode의 divideNum부터 M-1번째 pair를 right에 넣는다.

iii. right에 넣은 pair를 leafNode에서 삭제한다.

iv. right의 첫번째 key가 leafNode.parent의 맨 오른쪽에 들어가야 한다면

1. leafNode의 맨 오른쪽에 right의 첫번째 pair를 삽입한다.
2. leafNode.parent의 가장 오른쪽 pair의 left_child_node를 leafNode로 설정한다/
3. leafNode.parent의 r을 right로 설정한다.

v. 그렇지 않다면

1. leafNode.parent에 right의 첫번째 key가 들어갈 위치를 찾아 index에 저장한다.
2. leafNode.parent의 index 위치에 right의 첫번째 pair를 삽입하고 그 node의 left_child_node를 leafNode로 지정한다.
3. leafNode.parent의 index+1번째 left_child_node를 right으로 지정한다.

vi. leafNode.paremt.m을 1만큼 늘리고, right.r을 leafNode.r로 지정한 다음 leafNode.r을 right로 지정한다.

vii. right의 부모 node를 지정한다.

viii. 만약 leafNode.parent.m이 M 이상이라면 parent node에서 split이 일어나는 경우이므로 splitNonLeaf method를 실행한다.

Q. public static void splitNonLeaf(Node node)

i. right node를 생성하고 len을 node.m, divideNum을 len / 2로 설정한다.

ii. node의 divideNum번째 pair의 값을 각각 insertKey, insertValue, insertNode에 저장한다.

iii. node의 divideNum+1번째부터 M-1번째 pair를 right에 넣는다.

iv. right에 넣은 node의 pair를 삭제한다.

v. right.r을 node.r로 설정하고, node.r을 insertNode로 지정한다.

vi. right에 저장된 pair들의 left_child_node의 parent를 right로 지정한다.

vii. right.r과 node.r의 parent node를 각각 right와 node로 지정한다.

viii. 만약 node.parent가 null인 경우 (즉, parent node를 새로 만드는 경우)

1. parent node를 새로 생성한다.
2. parent에 (insertKey, insertValue, node) pair를 저장하고 parent.r을 right으로 설정한다.
3. node와 right의 parent node를 각각 parent로 설정하고 bPlusTree를 parent로 지정한다.

ix. 그렇지 않다면 (즉, parent node가 이미 존재하는 경우)

1. 만약 node.parent에서 insertKey의 index가 node.parent.m과 같다면
   A. node.parent의 맨 오른쪽에 (insertKey, insertValue, node) pair를 삽입한다.

B. node.parent.r을 right으로 설정한다.

2. 그렇지 않다면
   A. node의 index를 찾는다.
   B. node.parent의 index번째에 (insertKey, insertValue, node) pair를 삽입한다.
   C. node.parent의 index+1번째 pair의 left_child_node를 right으로 지정한다.
3. node.parent.m을 1만큼 늘리고 right.parent를 node.parent로 지정한다.
4. 만약 node.parent.m이 M 이상이라면, split이 일어나는 경우이므로 splitNonLeaf method를 실행한다.

R. public static void delete(String index_file, String data_file)

i. loadTree method를 이용해 B+Tree를 불러온다.
ii. BufferedReader를 이용해 CSV file을 한 줄씩 읽는다.
iii. 읽은 data를 deleteNode method에 넣는다.
iv. CSV file을 다 읽고 모든 연산을 수행했으면 saveTree method를 이용해 B+Tree를 저장한다.

S. public static void deleteNode(int key)

i. findOperateNode method를 이용해 key가 삭제될 node와 그 node의 왼쪽 node를 찾아 각각 node와 leftLeaf에 저장한다.
ii. traverseNode에 node를 저장한다.
iii. 반복문을 통해 node에 key가 있는지 확인한다. 만약 없다면, method를 종료한다.
iv. 삭제 연산 수행 전 node의 첫번째 key를 firstKey에 저장한다.
v. divideNum을 (M – 1) / 2로 저장한다.
vi. node에서 key를 삭제한다.
vii. 만약 key가 삭제 후 node의 첫번째 key보다 작다면, substituteKey method를 실행한다.
viii. 만약 divdeNum이 node.m 이하라면, node에 들어가야 하는 최소 key 개수 조건을 충족하였기에 method를 종료한다.
ix. 그렇지 않다면, findLeft method와 findRight method를 이용해 node의 왼쪽, 오른쪽 node를 찾은 뒤 이를 left와 right에 저장한다.
x. 만약 left가 비어있지 않고 divideNum이 left.m보다 작다면 (left에서 borrow 가능)
   1. node의 맨 앞의 위치에 left의 마지막 pair를 삽입하고, left에선 마지막 pair를 삭제한다.
   2. node.parent부터 firstKey를 node의 첫번째 key로 교체한다.
   3. method를 종료한다.
xi. 만약 right가 비어있지 않고 divideNum이 right.m보다 작다면 (right에서 borrow 가능)
   1. right의 첫번째 pair를 node의 마지막에 넣는다.

2. right의 첫번째 pair를 삭제한다.
3. node.parent부터 node의 마지막 pair의 key를 right의 첫번째 key로 교체한다.
4. method를 종료한다.

xii. 만약 그렇지 않다면, merge method를 실행한다.

T. public static int findIndexParent(Node node) : node의 parent node에서 index 탐색

i. node.parent의 i번째 pair의 left_child_node가 node라면, i를 return한다.
ii. 만약 node.parent.r이 node라면, node.parent.m을 return한다.
iii. 그렇지 않다면 index가 존재하지 않는 경우이므로 -1을 return한다.

U. public static void substituteKey(int change, int delete, Node node)

i. node에서 root까지 올라가며 delete를 change로 바꾼다.
ii. 만약 delete가 node의 i번째 key와 같다면, node의 i번째 key를 change로 바꾼다.
iii. 이때, non-leaf node에선 pair의 value가 중요하지 않으므로 value는 변경하지 않는다.
iv. 만약 node가 root(bPlusTree)라면, method를 종료한다.
v. 그렇지 않다면, node를 node.parent로 설정한다.

V. public static void merge(Node node, Node leftLeaf)

i. node의 왼쪽, 오른쪽 node를 찾아 left와 right에 저장한다.
ii. left가 null이 아니라면
    1. left로 node pair를 옮긴다.
    2. left.r을 node.r로 설정한다.
    3. findIndexParent method를 이용해 node의 index를 찾은 뒤 index에 이를 저장한다.
    4. 부모 node에서 index-1번째 값을 삭제한다.
    5. Tree height가 2이고 node.parent.m이 0이라면 left를 root로 설정한다.
    6. 만약 node.parent.m이 (M-1)/2 미만이라면, mergeParent method를 실행한다.
iii. right가 null이 아니라면
    1. right로 node의 pair를 옮긴다.
    2. 만약 node.parent의 맨 왼쪽 자식이 node라면 Tree의 tmpMin을 right의 첫번째 key로 바꾼다.
    3. leftLeaf.r을 right으로 설정한다.
    4. 부모 node에서 index번째 값을 삭제한다.
    5. Tree height가 2이고 node.parent.m이 0이라면 right을 root로 설정한다.
    6. 만약 node.parent.m이 (M-1)/2 미만이라면, mergeParent method를 실행한다.

W. public static void mergeParent(Node node)

i. node의 왼쪽, 오른쪽 node를 찾아 left와 right에 저장한다.
ii. left가 null이 아니라면
    1. findIndexParent method를 이용해 left의 index를 찾은 뒤 index에 이를 저장한다.

2. 부모 node의 key를 left로 옮긴다.
3. 부모 node에서 index번째 값을 삭제한다.
4. node의 pair을 left로 옮긴다.
5. left.r를 node.r로 설정하고, left.r의 parent를 left로 설정한다.
6. left.parent의 자식 node들을 left로 지정한다.
7. left의 child node의 parent를 left로 지정한다.
8. 만약 left.m이 M보다 크다면, splitNonLeaf method를 실행한다.
9. Tree height가 2이고 node.parent.m이 0이라면 left를 root로 설정한다.
10. 만약 left.parent.m이 (M-1)/2보다 작다면, mergeParent method를 실행한다.

iii. right가 null이 아니라면

1. findIndexParent method를 이용해 node의 index를 찾은 뒤 index에 이를 저장한다.
2. 부모 node의 key를 node로 옮긴다.
3. 부모 node에서 index번째 값을 삭제한다.
4. right pair를 node로 옮긴다.
5. node.r를 right.r로 설정하고, node.r의 parent를 node로 설정한다.
6. node.parent의 자식 node를 node로 지정한다.
7. node의 child node의 parent를 node로 지정한다.
8. 만약 node.m이 M보다 크다면, splitNonLeaf method를 실행한다.
9. Tree height가 2이고 node.parent.m이 0이라면 node를 root로 설정한다.
10. 만약 node.parent.m이 (M-1)/2보다 작다면, mergeParent method를 실행한다.

X. public static void singleSearch(String index_file, int key)

i. loadTree method를 이용해 B+Tree를 불러온다.

ii. bPlusTree로 지정된 node, 탐색 경로를 저장할 path 변수를 만든다.

iii. root node가 leaf node인 경우, node에서 key를 찾은 뒤 value를 출력한다.

1. 만약 key를 찾지 못했다면, “Not found”를 출력한다.

iv. node가 leaf node인 경우 다음 반복문을 실행한다.

1. 만약 node가 null이라면, path에 저장된 값을 출력하고 “Not found”를 출력한다.
2. 만약 그렇지 않다면
   A. path를 출력한다.
   B. key가 leaf node의 맨 오른쪽에 있는 경우, 맨 오른쪽의 value를 출력한다.
   C. key가 0 ~ node.m – 2 사이에 있는 경우, 해당 index의 값을 찾아 출력한다.
   D. key를 찾지 못했다면 “Not found”를 출력한다.
   E. method를 종료한다.

v. node의 pointer가 1개라면, path에 key를 넣은 다음 그 다음 node로 이동한다.

vi. 만약 그렇지 않다면

A. key가 node의 맨 처음 혹은 맨 오른쪽에 있다면 path에 그 key를 삽입한다.
B. 그렇지 않다면 search 중 지나는 앞의 key와 그 다음 key를 path에 넣는다.
C. 다음 node로 이동한다.

vii. node search 중에서 해당 node의 key과 다음과 같을 경우 그 key의 값만 출력한다.
1. 해당 node에 key가 하나밖에 없는 경우
2. search 연산의 key가 해당 node의 맨 첫번째 key보다 작은 경우
3. search 연산의 key가 해당 node의 맨 마지막 key보다 큰 경우

viii. 위의 경우가 아니라면, 해당 node의 child node로 이동할 때 최종적으로 비교되는 두 key를 출력한다. 즉, search하고자 하는 key보다 작은 key 중에서 가장 큰 key와 search 하고자 하는 key보다 큰 key중에서 가장 작은 key가 child node로 이동할 때 최종적으로 비교되므로 이 두 key를 순서대로 출력한다.

Y. public static void rangedSearch(String index_file, int start_key, int end_key)

i. loadTree method를 이용해 B+Tree를 불러온다.
ii. B+Tree가 비어있을 경우 "Not found"를 출력하고 method를 종료한다.
iii. start_key가 들어갈 node를 찾아 node에 저장한다.
iv. node에서 start_key보다 큰 key중 가장 작은 값이 있는 i를 찾아 index에 저장한다.
v. 만약 node에 index가 없어 오른쪽 node로 넘어가는 경우 node를 node.r로 설정하고 index를 0으로 설정한다.
1. 만약 node가 null이면 start_key가 index에 저장된 key 값을 넘어선 경우이므로 "Not found"를 출력한다.

vi. node 안에서 다음 반복문을 수행한다.
1. range 안에 key가 없는 경우 "Not found"를 출력하고 method를 종료한다.
2. start_key가 node의 i번째 key보다 작거나 같은 경우 key와 path를 출력하고 flag를 true로 설정한다.
3. node의 맨 오른쪽 key가 end_key 이하인 경우
A. 만약 node.r이 null이면 맨 오른쪽 leaf node에 도달한 경우이므로 method를 종료한다.
B. 그렇지 않으면 node를 node.r로 설정해 다음 node로 넘어간다.

3. Instructions for compiling

A. 구현 및 실행 환경 : MacBook Air M1 8GB RAM
B. 실행 방법
i. jar 파일이 있는 경로에 삽입과 삭제를 수행할 데이터를 가진 csv file을 넣는다.
ii. terminal에서 jar 파일이 있는 경로까지 이동 후 아래 명령어를 실행한다.
1. java -jar BPlusTree.jar "명령, 파일 이름 등"

```
~ $ java -jar BPlusTree.jar -c index.dat 3
~ $ java -jar BPlusTree.jar -i index.dat input.csv
~ $ java -jar BPlusTree.jar -s index.dat 68
26, 68, 86
97321
~ $ java -jar BPlusTree.jar -r index.dat 10 90
10, 84382
20, 57455
26, 1290832
37, 2132
68, 97321
84, 431142
86, 67945
87, 984796
~ $ java -jar BPlusTree.jar -d index.dat delete.csv
~ $ java -jar BPlusTree.jar -r index.dat 1 90
37, 2132
68, 97321
84, 431142
86, 67945
87, 984796
```

4. Any other specification of implementation and testing

A. 다음과 같은 코드를 작성해 1부터 10,000,000까지의 수를 csv파일에 순차적으로 저장한 다음 역순으로 삭제하는 test case를 생성하였다.

```java
import java.io.*;

public class Generator {

    public static void main(String[] args) {
        String[] Path = {"/Users/kangtaeuk/eclipse-workspace/BPlusTree/src/data.csv", "/Users/kangtaeuk/eclipse-workspace/BPlusTree/src/delete.csv"};
        File dataFile = new File(Path[0]);
        File deleteFile = new File(Path[1]);
        BufferedWriter bw1 = null, bw2 = null;

        try {
            bw1 = new BufferedWriter(new FileWriter(dataFile));
            bw2 = new BufferedWriter(new FileWriter(deleteFile));

            for (int i = 1; i <= 10000000; i++) {
                String str = Integer.toString(i) + "," + Integer.toString(i);

                bw1.write(str);
                bw1.write("\n");
            }

            for (int i = 10000000; i >= 1; i--) {
                bw2.write(Integer.toString(i));
                bw2.write("\n");
            }

            bw1.flush();
            bw1.close();

            bw2.flush();
            bw2.close();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

B. 실행 결과

```
~ $ java -jar BPlusTree.jar -c index.dat 5
~ $ java -jar BPlusTree.jar -i index.dat data.csv
~ $ java -jar BPlusTree.jar -s index.dat 4987300
3188647, 6377293, 4251529, 5314411, 5078215, 4999483, 4990735, 4987819, 4986847,
 4987333, 4987279, 4987297, 4987315, 4987303, 4987299, 4987301
4987300
~ $ java -jar BPlusTree.jar -r index.dat 10000 10005
10000, 10000
10001, 10001
10002, 10002
10003, 10003
10004, 10004
10005, 10005
~ $ java -jar BPlusTree.jar -d index.dat delete.csv
~ $ java -jar BPlusTree.jar -r index.dat 1 10000000
Not found
~ $
```