ITE 2038 – Database System:

Project: B+ Tree Implementation

2019014266 Lim Kyu Min

Project: B+ Tree Implementation – JAVA

2019014266 Lim Kyu Min

1. Instruction

Compile Location

🡪 BPlusTree\B\_Plus\_Tree\_Assignment

Constraints

1. 인덱스 파일, 입력 파일, 삭제 파일의 이름은 Predefined입니다. 파일명의 이름을 어길 시 실행이 되지 않도록 했습니다.

🡪 각각 index.dat, input.csv, delete.csv입니다.

2. Insertion을 실행할 시, input.csv에 입력 값이 존재해야 합니다.

🡪 예를 들어, Key가 3이고 Value가 10인 값을 넣고 싶을 때,

3,10

과 같이 input.csv에 값이 들어가 있어야 합니다.

3. Deletion 마찬가지로, delete할 key값이 delete.csv에 존재해야 합니다.

🡪 예를 들어, Key가 3인 값을 지우고 싶을 때,

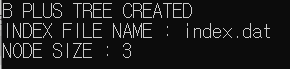
3

과 같이 delete.csv에 들어가 있어야 합니다.

1) Create B+ Tree (Initialize)

🡪 java -jar BPlusTree.jar "-c" "*INDEXFILE.NAME*" "*SIZE*"

Ex) java -jar BPlusTree.jar "-c" "index.dat" "3"

OutPut) 

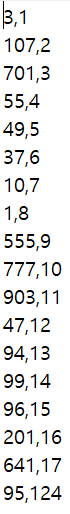
2) Insert B+ Tree (Insertion)

🡪 java -jar BPlusTree.jar “-I” “*INDEXFILE.NAME*” “*INPUTFILE.NAME*”

Ex) java -jar BPlusTree.jar "-i" "index.dat" "input.csv"

Output) 

* Input.csv



* Output.csv (Print all leaf nodes)

3) Single Search B+ Tree (Single Search)

🡪 java -jar BPlusTree.jar "-s" " *INDEXFILE.NAME* " "*TARGET\_KEY*"

Ex) java -jar BPlusTree.jar "-s" "index.dat" "107"

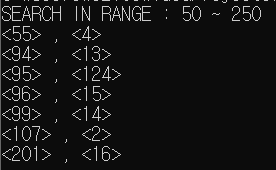
 OutPut)

* I : Index Node
* L : Leaf Node
* V : Value

4) Ranged Search B+ Tree (Range Search)

🡪 java -jar BPlusTree.jar "-r" " *INDEXFILE.NAME* " "*START\_KEY*" “END\_KEY”

Ex) java -jar BPlusTree.jar "-4" "index.dat" "50" "250"

 Output)

* Format : <key> , <Value>

5) Delete B+ Tree (Deletion)

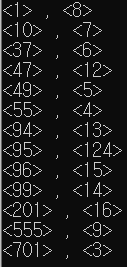
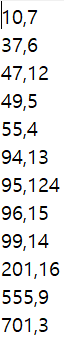
🡪 java -jar BPlusTree.jar "-s" " *INDEXFILE.NAME* " "*DELETEFILE.NAME* "

Ex) java -jar BPlusTree.jar "-d" "index.dat" "delete.csv"

 OutPut)

* Delete.csv



* Output of Range search - java -jar BPlusTree.jar "-4" "index.dat" "1" "1000"
* Output.csv (Deleted output)
* Code Analysis (각 기능을 수반하는 함수는 설명으로 대체했습니다.)

0. Structure

|  |
| --- |
| Class BPlusTree |
| * 전체 트리를 관리하는 클래스입니다. * 트리의 root를 설정해주는 root관련 함수와, 대부분의 B+ Tree관련 함수가 존재합니다. (Search, Insert…etc) |

|  |
| --- |
| Class BPTNode |
| * 노드를 표현하는 클래스이며, BPlusTree를 상속합니다. * 해당 노드가 리프인지 아닌지 표현하는 isLeaf와 최소 키의 개수를 나타내는 minKey등의 정보를 가지고 있으며, <Key,Value>를 TreeMap을 사용하여 저장합니다. * 각각의 정보를 계산하여 저장하는 함수들이 존재합니다. |

1. Create

|  |
| --- |
| bptCreate (Line 96 ~ 102) |
| BPlusTree bptCreate(BPlusTree bPlusTree, int size) {  BPTNode root = new BPTNode();// Initially the root node is a leaf.  root.determineLeaf(true);  root.setNodeInfo(size);  bPlusTree.setRoot(bPlusTree, root);  return bPlusTree; } |
| * B+ Tree를 생성하는 초기 설정하는 함수입니다. Root노드를 새로 형성하고, 그 노드를 해당 트리의 root로 설정합니다. * determineLeaf함수를 사용하여 해당 루트노드를 리프로 초기 설정 해줍니다. * setRootInfo메소드를 활용하여 초기 노드 정보를 설정해줍니다. (최소 키, 최대 키…등등) |

2. Insert

|  |
| --- |
| bptInsert (Line 104 ~ 182) |
| BPTNode bptInsert(BPlusTree bPlusTree, BPTNode root, Integer key, BPTNode leftChild, BPTNode rightChild, int size) { //Insert Index  //System.out.println("Key " + key + "become index!");  boolean isBiggest = true;   if (!root.isLeaf) {//Current node is index node  Set<Integer> keySet = root.p.keySet();  if (!keySet.isEmpty()) {  for (Integer i : keySet) {  //System.out.println(i);  if (key.equals(i)) {  //System.out.println("WARNING: Duplicated key is not allowed. - Input Ignored");  break;  } else if (key < i) {  //System.out.println("CASE 2");  //Case 2: The target key will be inserted to the middle of the node  root.p.put(key, leftChild);  root.updateElementNum(root);  isBiggest = false;   //System.out.println("Change Right KEY : " + i);  root.p.put(i, rightChild);  //System.out.println("End of Operation");  break;  }  }  } else { //There is nothing; current node is newly created index node  root.p.put(key, leftChild);  root.updateElementNum(root);  }  if (isBiggest) { // Case 1 : The target key will be inserted to the rightmost location  //System.out.println("Rightmost Index");  root.p.put(key, leftChild);  root.updateElementNum(root);  //System.out.println("Current Index Element: " + root.checkElementNum());   root.setRightChild(rightChild);  }  }  //System.out.println("Index Node Set");  return root; }  BPTNode bptInsert(BPlusTree bPlusTree, BPTNode root, Integer key, Integer value, int size) { //Insert Leaf  if (root.isLeaf) {//Current node is leaf node  //System.out.println("SET: [ " + key +" , " + value + " ]");   Set<Integer> keySet = root.v.keySet();   if (keySet.contains(key)) {  System.*out*.println("WARNING: Duplicated key is not allowed. - Input Ignored");  } else {  root.v.put(key, value);  root.updateElementNum(root);  //System.out.println("Leaf Element : " + root.checkElementNum());  }   if (root.checkElementNum() > root.getMaxKeys()) {  //System.out.println("Leaf Overflow! " + root.checkElementNum());  root = bptLeafSplit(bPlusTree, root, size);   }  } else { //Current node isn't leaf node  boolean isRecursiveCall = false;  Set<Integer> keySet = root.p.keySet();  for (Integer i : keySet) {  //System.out.println("Traverse: " + i);  if (key < i) {  bptInsert(bPlusTree, root.p.get(i), key, value, size);  isRecursiveCall = true;  break;  }  }  if (root.hasRightChild() && !isRecursiveCall) {  bptInsert(bPlusTree, root.getRightChild(), key, value, size);  }  }   return root; } |
| * insert\_CSV()함수 (Line 17) input.csv의 한 줄 데이터를 읽을 때마다 호출되는 함수입니다. 위의 것은 Index 노드를 삽입할 때 호출되는 함수이고, 밑의 함수는 Leaf 노드를 삽입할 때 호출됩니다. * Leaf와 Index의 저장방식의 차이, 각각 Split 방법의 차이 때문에 함수를 다르게 했습니다. * Leaf에서의 split은 leafSplit()함수를 통하여 호출합니다. Split이 일어나는 mid (혹은 pivot)값은 최대 element개수/2를 올림한 후, 만일 element의 개수가 짝수라면 1을 추가했습니다. * indexSplit()함수의 split 매커니즘은 리프와 비슷하나, leaf와 달리 pivot으로 parent가 되는 노드는 split과정에서 완전히 나누었습니다   🡪 즉, leaf에서의 스플릿에서 mid값은 그대로 rightChild의 첫번째 element가 되는 반먄, index에서의 mid값은 parent가 될 뿐, child에서 찾아볼 수 없게 되었습니다. |

3. Single Search

|  |
| --- |
| bptSingleSearch() (Line 324 ~358) |
| boolean bptSingleSearch(BPTNode root, int key) {   boolean recursiveCall = false;   if (root != null) {  if (root.isLeaf) { // Current node is leaf node  Set<Integer> keySet = root.v.keySet();  for (Integer i : keySet) {  System.*out*.print("\nL: <"+i+"> ");  if (i == key) {  System.*out*.println("V: <" + root.v.get(i)+">");  return true; // Matching key found  }  }  System.*out*.println("\nKey " + key + " not found");  return true;  } else { //Current node is index node;  Set<Integer> keySet = root.p.keySet();  for (Integer i : keySet) {  if (!recursiveCall) {  System.*out*.print("I: <"+i+"> ");  if (key < i) {  //System.out.println("Goes to Left Child!");  recursiveCall = bptSingleSearch(root.p.get(i), key);//Search its child if i < key  }  }  }  if (root.hasRightChild() && !recursiveCall) {  //System.out.println("Goes to Right Child!");  bptSingleSearch(root.getRightChild(), key);//If the node has right child  }  }  }  return true; } |
| * 다음과 같은 노드의 구성을 활용하여 찾습니다.   🡪 Index에서 현재 Key는 Key보다 작은 Children Node를 가리킵니다. 즉, 해당 Index에 적혀있는 key를 찾으려면 해당 Key **다음에** 존재하는 Key의 제일 왼쪽 리프에 접근해야합니다.   * 각 Element의 가장 큰 Key 값은 rightChild에 저장되어 있습니다. * Leaf의 첫번째 index는 연결되어있는 Parent key의 **앞에** 존재합니다. 즉, RightChild의 인덱스 키값은 Parent의 가장 마지막 키값이며, 가장 첫번째 리프의 키값은 Parent의 Parent, 즉 부모의 부모 노드의 첫번쨰 키값으로 저장되어 있습니다. * 해당 메커니즘을 활용하여 찾고자 하는 key값과 같거나 처음으로 더 큰 Key의 Child에 접근, 해당 과정을 재귀적으로 만들어 Leaf까지 접근했습니다. |

4. Ranged Search

|  |
| --- |
| bptRangeSearch() (Line 360 ~382) |
| void bptRangeSearch(BPTNode root, int start, int end) {   //System.out.println("\*\*\*Performing Range Search...\*\*\*");   BPTNode firstLeaf = root;  while (!firstLeaf.isLeaf) { //Search for leftmost leaf (smallest data)  Set<Integer> keySet = firstLeaf.p.keySet();  //System.out.println("Cur Leaf : " + keySet.iterator().next());  firstLeaf = firstLeaf.p.get(keySet.iterator().next()); //Reach to the leftmost data  }   //System.out.println("Starting Operation: Searching for values ranges in " + start + " and " + end);   while (firstLeaf != null) { //Search all leaf nodes  for (Integer i : firstLeaf.v.keySet()) {  //System.out.println("Traverse: " + i);  if (i >= start && i <= end)  System.*out*.println("<" + i + "> , <" + firstLeaf.v.get(i)+">");  }  firstLeaf = firstLeaf.r;  }  } |
| * 다음과 같은 노드의 구성을 활용하여 찾습니다.   🡪 각각의 Leaf는 연결되어 있으며, 이를 활용하여 손쉽게 전체 Leaf Value에 접근할 수 있습니다.   * 해당 메커니즘을 활용하여 모든 Index의 가장 첫번째 child에 접근하여 첫번째 Key값을 찾은 후, RightChild으로 접근하며 전체 Leaf에 접근했습니다. 해당 과정에서 범위 내에 있는 Key와 Value만 출력하였습니다. |

5. Delete

|  |
| --- |
| bptDelete() (Line 447~468) |
| void bptDelete(BPlusTree bPlusTree, BPTNode root, Integer key) {  BPTNode target = reachToTarget(root, key);  BPTNode duplicatedIndex = findDuplicates(target, key);  if (target.checkElementNum() > target.getMinKeys()) { //Case 1: Current Key has enough key to be deleted.  target.v.remove(key);  target.updateElementNum(target);  if (!Objects.*isNull*(duplicatedIndex))  renewKey(duplicatedIndex, key, target.v.keySet().iterator().next());  } else { //Deficiency  //System.out.println("DEFICIENCY");  //Case 2: Borrow & Rotation  //System.out.println("Case 2");  boolean isSuccessful = false;  isSuccessful = rotation(target, key, target.isLeaf);  if (!isSuccessful) { //Case 3: Merge  //System.out.println("Case 3");  target.v.remove(key);  target.updateElementNum(target);  mergeNode(bPlusTree, target, key);  }  } } |
| * 3가지 Case로 나뉩니다. * 1. 해당 Element가 Delete를 하기에 충분한 양이 있을 때:   🡪 별 다른 과정 없이 해당 노드를 지웁니다. 만약 중복된 노드가 존재하면 해당 노드를 올바른 값으로 정정합니다.   * 2. 해당 Element가 Underflow상태이지만, 키 값을 주변 Sibling에서 빌려올 수 있을 때.   🡪 RightSibling을 우선으로 하여 Rotation을 시도합니다. 만약 해당 sibling의 Element가 충분할 때, 값을 빌려오고 인덱스파일을 알맞게 수정합니다.   * 3. Element가 Underflow상태이며, 주변 Sibling에서 값을 빌려올 수 없을 때   🡪 Merge를 시도합니다. Parent와 Sibling끼리 Merge하고, 이로 인해서 변하는 인덱스 값들은 Recursive하게 delete처리를 합니다. |