B+tree implementation assignment

2019049716 안재국

1. 클래스 구성

1-1. Node

자신의 부모 노드를 가리키는 parent 변수를 갖고 있으며 base class입니다.

1-2. NonLeafNode

key(int)와 child를 배열로 갖고 있는 internal Node로 Node 클래스를 상속받아 사용합니다.

이론상으로 overflow가 난 경우에도 실제로는 overflow가 나지 않도록 배열을 하나 크게 만들어줍니다.

1-3. key

key와 value를 변수로 갖는 클래스로 Comparable 인터페이스를 상속해 키 값을 기준으로 정렬할 수 있도록 했습니다.

1-4. LeafNode

key(class)배열과 자신의 왼쪽, 오른쪽 sibling을 가리키는 left, right 변수를 갖고 있으며 Node 클래스를 상속받아 사용합니다.

명세상 left는 굳이 필요하지 않지만 편의상 만들어주었습니다.

LeafNode도 마찬가지로 overflow가 난 상황에 실제로는 overflow가 나지않도록 key배열을 이론상 사이즈보다 1 크게 만들어줍니다.

1-5. BPlusTree

M : 트리의 degree입니다.

NonLeafNode root : 트리의 최상위 노드인 root를 가리킵니다.

LeafNode Leaf : 트리의 리프노드 중 가장 왼쪽에 있는 리프노드를 가리킵니다.

static LeafNode lastLeaf : index.dat에 저장된 트리를 복구할 때 사용되는 변수로 가장 최근에 복구된 리프노드를 가리켜 리프노드끼리 연결될 수 있도록 해줍니다.

그리고 여러 method() 들을 가지고 있습니다.

2. 데이터 저장 방식

터미널에서 명령을 호출할 때마다 insert를 통해 트리를 재구성하는 것은 시간 낭비가 심할 것이라 예상되어 트리의 구조를 그대로 index.dat 파일에 옮겼습니다.

상위에 "N" 또는 "L"을 적어 새로 생성할 노드가 NonLeafNode인지, LeafNode인지 표기했습니다.

밑에는 이어서 해당 노드의 자식의 수, 자식 노드의 종류(N or L), 자식의 key 값을 차례로 기록했습니다.

재귀를 통해 루트부터 리프 노드까지 내려가며 노드들에 대한 정보가 기록됩니다.

3. 데이터 생성 및 저장, 읽기

A. saveTree()

Tree의 정보를 index.dat 파일에 저장하는 함수입니다.

현재 기록하려는 노드가 root 노드일 때만 본인의 key 값과, 자식의 key 값들을 모두 저장합니다. root 노드가 아니라면 자식들의 key값만 기록합니다. 만약 현재 노드가 LeafNode라면 자식 노드가 없기 때문에 아무것도 기록하지 않고 리턴합니다.

B. makeTree()

index.dat에 적혀있는 정보를 바탕으로 Tree를 복구하는 역할을 합니다.

복구 시 가장 최근에 복구된 LeafNode를 lastLeaf라는 스태틱 변수에 저장하여 리프 노드를 새로 만들 시 lastLeaf와 새로 만들어진 리프를 연결시켜 모든 리프가 Linked List 형태로 연결될 수 있도록 합니다. 그리고 재귀적으로 함수를 구성하여 부모 노드와 자식노드간의 연결이 잘 이루어질 수 있도록 하였습니다.

4. Insertion 함수

A. findLeafNode()

새로운 키가 들어갈 리프 노드를 찾아주는 함수입니다.

B. getMidIdx()

스플릿의 기준이 될 기준점(floor(M/2)을 리턴해주는 함수입니다.

C. Insert()

findLeafNode() 함수를 통해 키를 삽입할 리프 노드를 찾아 새로운 키 값을 넣어준 후 정렬합니다. 키 값을 넣은 후 isOverFlow()함수를 통해 오버플로우가 발생했는지 확인하고 만약 오버플로우가 발생했다면 SplitLeafNode() 함수를 호출합니다.

D. SplitLeafNode()

오버플로우가 난 리프노드를 스플릿을 통해 균형을 맞춰줍니다. getMidIdx()를 호출하여 기준이 될 인덱스를 정합니다. 인덱스는 노드를 반으로 나누기 위한 기준이 됩니다. 이 인덱스에 해당하는 키는 부모 노드의 새로운 키가 됩니다. 이때 이 인덱스에 해당하는 키는 노드를 반으로 나눴을 때 오른쪽 노드에 포함되도록 했습니다.

스필릿할 리프 노드의 parent 노드가 null이라면 아직 루트노드가 존재하지 않는다는 뜻이므로, 새로운 부모 노드를 만들고 그 노드를 root 노드로 지정합니다.

만약 부모 노드가 존재한다면 rightSibling이라는 LeafNode를 새로 만들어 현재 스플릿 되어야 할 노드의 키 절반을 rightSibling으로 옮겨줍니다. 그리고 부모 노드에서 새로운 키 값(기준이 되었던 인덱스)이 들어갈 자리를 찾아 넣어주고 rightSibling을 새로운 자식으로 삽입합니다. rightSibling의 parent는 스플릿된 노드의 parent와 일치시켜 줍니다. 또한 rightSibling을 현재 스플릿된 노드의 right로 지정하여 리프 노드 간의 연결이 끊기지 않도록 합니다.

이때 부모 노드에 새로운 키 값이 들어가게 되면서 오버플로우가 났다면 SplitNonLeafNode() 함수를 호출하여 균형을 맞춰줍니다.

D. SplitNonLeafNode()

오버플로우가 난 NonLeafNode를 스플릿을 통해 균형을 맞춰줍니다.

이 함수는 SplitLeafNode()가 발생한 후 연쇄적으로 호출되며 단독적으로는 호출되지 않습니다.

LeafNode를 스플릿할 때와 마찬가지로 getMidIdx()함수를 통해 인덱스를 얻으며, 인덱스에 해당하는 키값은 위로 올라가야 하기 때문에 오른쪽 노드에 포함되지 않도록 해야 합니다.

newNode라는 새로운 인터널 노드를 만들어 준 후, 절반을 기준으로 오른쪽에 있는 키 값들을 newNode의 키로 넣어줍니다.

리프노드를 스플릿할 때와는 달리 인터널노드를 스플릿하게 되면 자식 노드들도 newNode의 자식으로 옮겨주어야 합니다. 이때 옮겨지는 자식 노드들의 부모를 newNode로 만들어주는 것이 중요합니다.

스플릿할 노드의 parent가 null값이라면 현재 루트를 스플릿하고 있다는 의미이므로 새로운 루트를 만들어주어야 합니다. 만약 parent가 null이 아니라면 parent에 스플릿의 기준이 된 키를 넣어주고, newNode를 자식으로 삽입합니다.

여기서 parent에서 또다시 오버플로우가 나면 계속해서 SplitNonLeafNode()를 오버플로우가 나지 않을 때까지 반복하여 호출합니다.

5. Deletion 함수

A. Delete()

insert 때와 마찬가지로 findLeafNode()함수를 통해 삭제할 키가 들어있는 리프노드를 찾아 삭제할 key를 찾은 후 삭제합니다. 삭제한 키가 들어있는 리프 노드가 모든 리프 노드 중 가장 왼쪽에 있는 리프 노드 즉 BPlusTree.Leaf가 아니면서 삭제한 키가 리프 노드의 키들 중 가장 왼쪽 즉, 가장 작은 값이라면 해당 키 값이 인터널 노드에 존재한다는 뜻이므로 updateKey()함수를 이용해 인터널 노드의 키 값을 현재 상황에 맞게 갱신합니다.

삭제한 후 isUnderFlow() 함수를 통해 해당 리프 노드에 언더플로우가 발생했는 지 확인합니다. 만약 언더플로우가 발생했다면 balanceLeafNode()함수를 호출하여 균형을 맞춰줍니다.

B. updateKey()

인터널 노드에 존재하는 키 값이 삭제되었을 때 해당 함수를 호출하여 인터널 노드의 키 값을 갱신합니다. 이때 키 값은 삭제된 키가 들어있는 인터널 노드의 오른쪽 subtree의 key들 중 가장 작은 값으로 갱신되어야 합니다. 가장 작은 키 값은 subtree의 리프노드들 중 가장 왼쪽에 있는 리프노드의 0번째 인덱스에 들어있으므로 findLeftMostKey()함수를 호출하여 해당 값을 찾아냅니다.

이때 제 코드의 경우 키 삭제 이후에 updateKey를 진행하기 때문에 findLeftMostKey()를 호출했을 때 오른쪽 subtree의 맨 왼쪽 리프노드가 비어있을 수도 있습니다(M이 4 이하인 경우). 그 경우에는 맨 왼쪽 리프노드의 right sibling의 0번째 키를 리턴합니다.

C. balanceLeafNode()

리프 노드에 있는 키 값이 삭제되었을 때 언더플로우가 발생한다면 이 함수를 호출하여 트리의 균형을 맞추어 줍니다.

균형을 맞출 때, merge를 하는 것 보다는 키를 빌려오는 게 조금 더 간단하기

에 키를 빌려올 수 있다면 키를 빌리는 것을 우선적으로 시행했습니다. 만약 왼쪽 오른쪽 sibling 중 키를 빌려줄 sibling이 없다면 merge를 하는 것으로 했습니다. 또한 균형을 맞춰줄 리프노드의 상황을 3가지로 나누어 생각했습니다.

1. 리프노드가 부모 노드의 가장 왼쪽 자식인 경우

: 이 경우에는 오직 오른쪽 sibling만 존재하기 때문에 왼쪽 sibling에서 키를 빌려오거나 merge하는 경우는 생각하지 않았습니다.

1-1. right sibling에게서 키를 빌릴 수 있다면 right sibling의 키 값중 가장 작은 값을 현재 노드의 가장 큰 키로 빌려옵니다.

1-2. right sibling에게서 키를 빌릴 수 없다면, right sibling과 merge를 해야합니다. 현재 노드의 키들이 right sibling의 키 값보다 모두 작은 값이므로 right sibling의 앞쪽부터 현재 노드의 키를 넣어줍니다. 그리고 현재 노드의 left의 right를 right sibling으로 해서 리프 노드간의 연결이 끊기지 않도록 합니다. 현재 노드는 부모의 0번째 자식이므로 부모 노드의 0번째 키와 0번째 자식을 삭제합니다.

2. 리프노드가 부모 노드의 가장 오른쪽 자식인 경우

: 이 경우에는 오직 왼쪽 sibling만 존재하기 때문에 오른쪽 sibling에서 키를 빌려오거나 merge하는 경우는 생각하지 않았습니다.

2-1. left sibling에게서 키를 빌려올 수 있다면 left sibling의 키 값들 중 가장 큰 값을 현재 노드의 가장 작은 키로 빌려옵니다.

2-2. left sibling에게서 키를 빌려올 수 없다면 left sibling과 merge 해야합니다. 현재 노드의 키들은 left sibling의 키값들보다 모두 큰 값이므로 left sibling의 기존의 키 값들 뒤에 넣어줍니다. 이때 역시 left sibling의 right를 현재 노드의 right로 설정하여 리프노드들 간의 연결이 끊기지 않도록 합니다.

3. 두 가지 모두 아닌 경우

: 이 경우에는 왼쪽, 오른쪽 sibling이 모두 존재하기에 양쪽에서 키를 빌리거나 merge하는 경우를 모두 생각했습니다.

키를 빌려오는 것을 우선적으로 생각하고 불가능하다면 왼쪽 노드와 merge하는 방식으로 진행했습니다.

위의 규칙은 인터널 노드의 균형을 맞추어 줄 때도 동일하게 적용했습니다.

해당 리프노드가 가장 왼쪽 또는 가장 오른쪽 자식인지는 findIndexOfChild()함수를 통해 해당 노드가 부모 노드의 몇 번째 자식인지 찾아서 확인했습니다.

이때 merge를 하는 경우에는 부모 노드의 키 값을 하나 지우게 되는데, 이로 인해 부모 노드에 언더플로우가 발생한다면 balanceNonLeafNode()를 호출하여 균형을 맞추어 줍니다.

D. balanceNonLeafNode()

3가지 상황으로 나누어 생각하는 매커니즘은 balanceNonLeafNode()에서와 동일합니다.

1. 키를 빌리는 경우

키를 빌려올 때의 방식은 리프 노드의 균형을 맞추는 것과 동일합니다. 다만 키를 빌리거나 merge할 때 자식 노드들도 함께 옮겨야 한다는 것에 차이가 있습니다. left sibling에서 키를 빌리는 경우에 가장 오른쪽에 있는 키를 빌려오기 때문에 가장 오른쪽에 있는 자식을 현재 노드로 데려옵니다. 이때 데려오는 자식의 parent가 언더플로우가 발생한 노드가 되도록 합니다. 반대로 right sibling에서 키를 빌려올 때는 가장 값이 작은 키를 빌려오고, 가장 왼쪽에 있는 자식을 현재 노드의 가장 오른쪽 자식으로 데려옵니다. 이때도 역시 데려오

는 자식의 parent가 언더플로우가 발생한 노드가 될 수 있도록 합니다.

2. merge하는 경우

merge를 하는 경우에는 리프 노드의 균형을 맞출 때와 조금 차이가 있습니다. 리프 노드와는 달리 인터널 노드는 자식을 갖고 있는데, 자식의 수는 항상 키의 수보다 1 많아야 합니다. 그런데 merge를 하게 되면 한 쪽 노드의 키가 K1개 라면 자식은 (K1 + 1)개, 다른 한 쪽 노드의 키가 K2개라고 하면 자식은 (K2 + 1)개 두 노드를 합치면 키는 (K1 + K2)개가 되지만, 자식은 (K1 + K2 + 2)개가 되므로 맞지 않게 됩니다. 그래서 부모 노드로부터 적절한 키 값을 하나 가져와야 합니다. 이렇게 되면 부모 노드 입장에서는 키와 자식 각각 하나씩 삭제하게 되므로 지장이 없습니다. 단, 키가 하나 없어지면서 언더플로우가 발생하면 이때는 다시 balanceNonLeafNode()를 호출하여 균형을 맞추어줍니다. 루트 노드의 경우 키가 0개가 되는 경우가 아니라면 언더플로우가 발생해도 상관이 없지만, 키가 0개가 되어 버린다면 트리의 height를 1 낮추어야 합니다.

6. single_Search_Key 함수

루트 노드부터 출발하여 찾고자 하는 key값을 기준으로 리프 노드까지 타고 내려갑니다. 그리고 지나온 인터널 노드들의 키를 출력하기 위해 노드들을 Path라는 리스트에 넣어두었습니다. 리프 노드에서 원하는 값을 찾으면 Path에 있는 노드들의 키를 출력하고 마지막으로 찾은 키의 value값을 출력합니다

7. range_Search_Key 함수

findLeafNode() 함수를 이용해 원하는 key값이 들어있을 수 있는 리프노드를 찾아갑니다. 해당 리프 노드부터 right를 통해 오른쪽 sibling으로 계속 이동하면서 시작값 < key < 끝 값이라면 해당 key를 출력합니다. 만약 key >= 끝 값이 되는 순간에는 함수를 종료합니다.

8. print_tree() 함수

중간 중간 트리의 상황을 출력하면서 확인하기 위해 만든 함수입니다.

9. 실행 커맨드

javac BPlusTree.java : BPlusTree.java 컴파일

java BPlusTree.java –c index.dat # : # 의 degree를 가진 index.dat 생성.

java BPlusTree.java –i index.dat input.csv : input.csv에 있는 데이터를 트리에 삽입.

java BPlusTree.java –d index.dat delete.csv : 인덱스 파일의 트리에서 delete.csv에 있는 키를 삭제.

java BPlusTree.java –s index.dat # : 인덱스 파일의 트리에서 #을 찾음.

java BPlusTree.java –r index.dat # # : 인덱스 파일의 트리에서 #과 # 사이에 있는 모든 키를 찾음.

10. 터미널에서 실행하는 모습

```
PS C:\Users\jk672\IdeaProjects\Assignment1> cd .\src\BPlusTree
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> javac BPlusTree.java
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree -c index.dat 4
Error: Could not find or load main class BPlusTree
Caused by: java.lang.NoClassDefFoundError: BPlusTree/BPlusTree (wrong name: BPlusTree)
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree.java -c index.dat 4
```

명령어 실행 시 .java를 붙이지 않으면 실행이 되지 않는데 이유를 찾지 못했습니다.

-c 명령 실행 시, index.dat 파일을 생성하여 최상단에 degree값을 기록합니다.

```
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree.java -i index.dat input.csv
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree>
```

-i 명령 실행 시, index.dat에서 기존에 저장되어있던 트리를 복구한 후 input.csv에서 값을 읽어 새로운 key를 트리에 삽입합니다.

index.dat

```
4
Root
N
3
20,68,86
4
L
2
9,87632
10,84382
3
20,57455
26,1290832
37,2132
2
68,97321
84,431142
2
86,67945
87,984796
```

Insertion 과정 이후, 트리의 상태를 index.dat에 저장한 모습입니다.
2번째 줄에 "Root" 또는 "Leaf"를 적어서 루트가 존재하는 지 또는 리프노드만 존재하는 지를 구별할 수 있게 하여 트리를 복구할 때 원활하게 할 수 있도록 했습니다. 이후 아래에는 루트 노드라면 본인을 포함하여 자식 노드들에 대한 정보고 같이 기록하게 했습니다. 루트 노드가 아닌 NonLeaf 노드라면 자식들에 대한 정보만

저장하도록 했습니다. 트리를 복구할 때 부모와 자식 간의 연결이 잘 될 수 있도록 하기 위해서입니다.

```
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree.java -d index.dat delete.csv
68 86
Leafs 37   Leafs 68 84   Leafs 86 87
```

-d 명령 실행 시, index.dat 파일로 트리를 복구한 후 delete.csv 파일에서 값을 읽어 해당하는 키를 삭제합니다. 아래에는 삭제 이후 트리의 모습을 출력해보았습니다.

index.dat

```
4
Root
N
2
68,86
3
L
1
37,2132
2
68,97321
84,431142
2
86,67945
87,984796

```

삭제 이후 변화된 트리의 모습을 저장한 index.dat 파일의 모습입니다.

```
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree.java -s index.dat 68
68,86
97321
```

-s 명령 실행 시, index.dat 파일에서 트리를 복구한 후 해당하는 키 값 트리에 있는지 탐색하고 경로와 value를 출력합니다. 없다면 "NOT FOUND" 메시지를 출력합니다.

```
PS C:\Users\jk672\IdeaProjects\Assignment1\src\BPlusTree> java BPlusTree.java -r index.dat 38 85
68,97321
84,431142
```

-r 명령 실행 시, index.dat 파일에서 트리를 복구한 후 범위 내에 있는 모든 키들의

key와 value 값을 콤마(,)로 구분하여 출력합니다.