B+Tree Project

2020059152 오주영

1. Summary of your algorithm

Python(3.10.0)으로 구현했다.

Insertion 알고리즘

Insertion은 자료구조 수업 때 배운 B-Tree의 Insertion과 유사하게 구현하였다. ROOT에서부터 시작하여, records 배열에 담긴 key값들에 대한 선형 탐색으로 하위 child Node에 재귀적으로 insert을 한다. Leaf Node에 도달하면, Record를 추가한다. 이후 재귀가 풀리며(unwinding recursion) 삽입 당한 child Node가 overflow이라면 적절히 split을 해주고, 부모 Node와 병합해준다. 이 병합 과정으로 인해 부모 Node가 overflow된다면 또 부모 Node의 부모 Node에서 처리를 해주는 방식이다. 따라서 부모 Node가 없는 ROOT Node에 overflow가 발생하는 경우엔 ROOT 자체를 split해준다.

여기까진 B-Tree의 Insertion과 동일하지만, split과 그 이후의 처리과정이 다르다. split을 수행한 Node가 Leaf Node인지 아닌지에 따라 알고리즘에 차이가 있는데, 이는 아래 splitNode(node) 함수 부분에 기술하겠다.

Deletion 알고리즘

Deletion은 B-Tree에서 구현해본적이 없었다. 그래도 Insertion과 비슷한 원리, 즉 재귀적으로 구현을 시작했으나 underflow 처리 방식이 난관이었다. 인터넷에서 참고한 자료들 사이에서도 underflow가 발생할 시 수행할 과정들에 대한 규칙(Ruleset)이 많이 차이가 났다. 결국 B+Tree를 시각화해주는 사이트 https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html 에 여러 값을 넣었다 빼며 그 패턴을 분석해 underflow를 처리할 과정에 대한 나의 Ruleset을 정했다. 다음과 같다.

삭제가 일어난 Node를 childNode, childNode의 부모를 parentNode, childNode의 왼쪽, 오른쪽 Sibling Node들을 각각 leftSibling, rightSibling이라고 했을때

1. leftSibling이 존재하고, key의 개수가 여유 있다면
    a. childNode가 Leaf Node이라면 leftSibling으로부터 record 하나를 빼온다(Steal).
    b. childNode가 Non-leaf Node이라면 parentNode의 record를 childNode으로 전달하고, leftSibling의 record를 parentNode으로 전달한다(Redistribute).
2. rightSibling이 존재하고, key의 개수가 여유 있다면

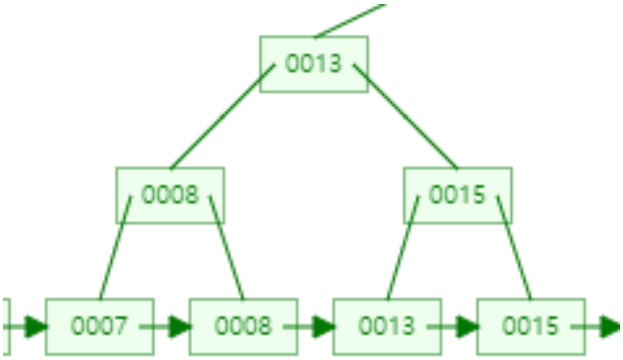
a. childNode가 Leaf Node이라면 rightSibling으로부터 record 하나를 빼온다(Steal).
b. childNode가 Non-leaf Node이라면 parentNode의 record를 childNode으로 전달하고, rightSibling의 record를 parentNode으로 전달한다(Redistribute).

3. leftSibling이 존재한다면 leftSibling과 합친다(Merge).
4. rightSibling이 존재한다면 rightSibling과 합친다(Merge).

0. 재귀가 풀리며 Non-leaf Node에도 지울 key값이 존재한다면, childNode의 후계자의 key로 바꾼다.

underflow의 기준은 key 개수가 {(DEGREE/2)의 올림 -1}보다 작아지는 것이다. 예를 들어 DEGREE가 5일 때 key 개수는 최소 2개는 있어야 한다. 위 Ruleset에서 “key의 개수가 여유 있다”는 말은, key를 하나 줄여도 underflow가 일어나지 않음을 의미한다. 즉 DEGREE가 5일 때 key 개수가 3개 이상인 Node들을 여유가 있다고 하겠다.

또 Sibling은 immediate sibling이어야 한다. 즉 childNode의 부모와 Sibling의 부모가 같아야 한다.



위 사진을 예로 들자면, Leaf 단계에 있는 13의 leftSibling은 8이 아닌 None이고, 8의 rightSibling은 13이 아닌 None이다.

사실 Steal과 Redistribute 과정은 childNode의 Leaf 여부에 따로 구분하지 않고 한 로직만으로 구현할 수 있지 않을까 싶다. 하지만 나의 코드 디자인 상으론 위 Ruleset의 0번인 Non-leaf Node에서도 값을 바꿔주는 과정이 수행되어야 하는 시기가 뒤엉키게 되었다. 예를 들어 Steal을 하는 상황에서는 Steal이 끝난 다음에야 0번 과정을 수행해야 했지만, 나머지 Redistribute과 Merge는 수행하기 전에 0번 과정을 미리 해주어야 했다. 좀 더 간결한 코드를 작성할 수 있을텐데, 실력이 부족해 그러지 못한 것이 아쉽다.

---

2. Detailed description of your codes

라이브러리

```
import csv;
import sys;
```

csv는 csv(Comma Separated Values) 파일을 읽어오는데 필요한 라이브러리, sys는 sys.argv 와 같은 명령 인자를 받아오는데 필요한 라이브러리이다.

클래스

```
class Record:
    def __init__(self, key, value = None, child = None):
        self.key = key;
        self.value = value;
        self.child = child;
    def __str__(self):
        return str(self.key)+","+str(self.value);
```

Pair 역할을 갖는 클래스이다. child는 왼쪽 child node를 가리키는 포인터의 역할을 한다. 원칙상 Non-leaf Node의 Record에는 key와 child만 존재하고, Leaf Node의 Record에는 key와 value만이 있어야 한다. 하지만 B+Tree의 insert이나 delete 과정에서 Leaf Node가 Non-leaf Node가 되기도 하고, Non-leaf Node가 Leaf Node로 내려오는 일이 빈번해 우선 value와 child를 둘다 포함시켰다.

```
class Node:
    def __init__(self):
        self.isLeaf = True;
        self.keyCount = 0;
        self.records = [];
        self.next = None;

    def addRecord(self, pos, record):
        self.keyCount+=1;
        self.records.insert(pos, record);

    def removeRecord(self, pos=0):
        self.keyCount-=1;
        return self.records.pop(pos);

    def getChild(self, pos=0):
        return self.records[pos].child;

    def __str__(self):
        res = "";
        if self.keyCount==0:
            return res;
        for i in range(len(self.records)):
            res+=str(self.records[i].key);
            if self.isLeaf:
                res+=":"+str(self.records[i].value);
            if i != len(self.records)-1: res+=",";
        return res;
```

isLeaf는 Node가 Leaf Node인지 아닌지에 대한 boolean, **keyCount**는 Node가 갖고 있는 key의 개수, **records**는 Record들을 담을 List이다. **next**는 Leaf Node의 경우 Right Sibling을, Non-leaf Node의 경우 가장

오른쪽 Child Node를 가리키는 포인터의 역할을 하는 변수이다.

addRecord(self, pos, record)는 keyCount를 하나 증가시키면서 records 배열의 pos번째 인덱스에 record를 추가하는 함수이다. Python의 List Class에서 제공하는 insert 함수를 활용했기에, records의 pos번째 인덱스에 이미 값이 있다면 pos 부터 마지막 인덱스에 있는 값들을 전부 오른쪽으로 옮기고 pos 위치에 record를 삽입할 수 있다.

removeRecord(self, pos)도 비슷하게 keyCount를 하나 줄이면서 records의 pos번째 인덱스에 존재하는 record를 지우고 반환하는 함수이다.

getChild(self, pos)는 records의 pos번째 인덱스에 있는 record의 child를 반환하는 함수이다.

__**str**__(self)는 Java의 toString() 함수 개념이다. Node가 갖고 있는 모든 record들을 문자열로 반환하도록 재정의하였다. Leaf Node의 경우 record들을 “key:value” 형식으로 출력하고, Non-leaf Node는 “key” 형식으로 출력된다. 디버깅, 그리고 파일 입출력에서 요긴하게 사용하였다.

전역변수

```
DEGREE = 0;
ROOT = None;
```

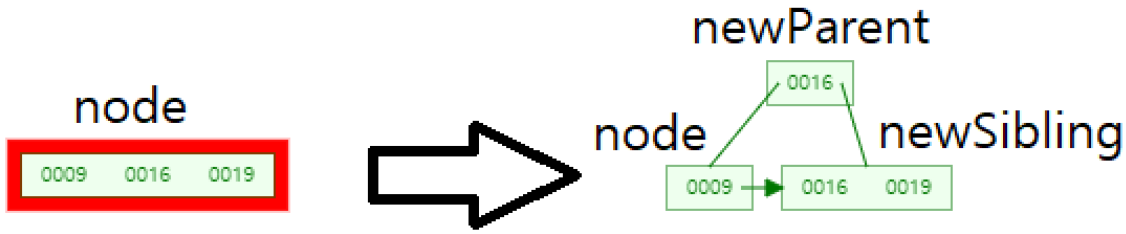
DEGREE는 Node의 최대 child 개수이다. 즉 Node는 최대 DEGREE-1개의 key를 가질 수 있다. Node의 keyCount가 DEGREE에 도달하면, 적절히 split을 해야한다.

ROOT는 B+Tree의 루트 Node가 된다. insert나 delete 과정에서 어떤 Node가 루트 Node인지 아닌지에 따라 실행과정이 다를 수 있기에 전역변수로 선언해두었다.

---

splitNode(node)

인자로 받은 node를 아래의 그림처럼 3개의 node로 쪼개주는 함수이다.




```
def splitNode(node):

    mid = DEGREE // 2;

    newSibling = Node();
    newParent = Node();

    newParent.isLeaf = False;
    newSibling.isLeaf = node.isLeaf;

    newParent.addRecord(0, node.records[mid]);
    newParent.records[0].child = node;
    newParent.next = newSibling;

    # LeafNode들은 LinkedList처럼
    if node.isLeaf: node.next = newSibling;
    else: node.next = node.getChild(mid);

    t = mid if DEGREE%2!=0 else mid-1;

    for i in range(t):
        newSibling.addRecord(i, node.removeRecord(mid+1));
    if node.isLeaf:
        newSibling.addRecord(0, node.records[mid]);
    node.removeRecord(mid);

    return newParent;
```

1. mid는 DEGREE를 2로 나눈 몫, 즉 이 node의 중간 인덱스가 된다. newSibling과 newParent를 생성하고, 각각의 isLeaf를 조정해준다. newParent는 당연히 child를 가지므로 isLeaf일 수 없고, newSibling은 node와 같은 높이에 생기게 되므로 node의 isLeaf와 같다.
2. newParent에 node의 가운데 있던 record를 올려주고, newParent의 왼쪽 child를 node로 설정해준다. newParent의 next를 newSibling으로 정해준다.
3. node의 next를 정해준다.
4. newSibling에게 node의 record를 t개 넘긴다. t는 DEGREE가 홀수인지 짝수인지에 따라 알고리즘이 다르다.
5. node가 Leaf 단계인 경우에만 newSibling에게 가운데 값을 넘긴다. 위 사진에서 newSibling에 16을 주는 과정이다.
6. newParent에 이미 node의 mid에 있는 key를 올렸으니, newParent의 왼쪽 child인 node는 더 이상 이 key를 가질 수 없으므로 node에서는 지워준다.
7. newParent를 return한다.

insertKey(node, record)

```
def insertKey(node, record):

    pos = 0; # 들어갈 위치
    while pos < node.keyCount:
        if node.isLeaf and record.key == node.records[pos].key:
            # Duplicate keys not allowed
            return node;
        if record.key < node.records[pos].key: break;
        pos+=1;

    # Leaf이면 그냥 삽입
    if node.isLeaf:
        node.addRecord(pos, record);
        # leaf이면서 ROOT인 경우에 split.
        if node.keyCount == DEGREE and node == ROOT:
            node = splitNode(node);
        # 그냥 leaf면 부모에서 처리

    # non-leaf
    else:
        childNode = None; #삽입 당할 아이
        #맨 오른쪽
        if pos == node.keyCount: childNode = node.next;
        else: childNode = node.getChild(pos);

        childNode = insertKey(childNode, record);

        #자식 터짐.
        if childNode.keyCount == DEGREE:

            tempL = childNode.getChild(DEGREE//2);
            tempR = childNode.next;

            childNode = splitNode(childNode);

            childNode.records[0].child.next = tempL;
            childNode.next.next = tempR;

            #맨 오른쪽에 들어갔으면 전체(node)의 next 갱신
            if pos == node.keyCount: node.next = childNode.next;
            #왼쪽 child 갱신
            else: node.records[pos].child = childNode.next;

            # 병합
            node.addRecord(pos, childNode.records[0]);

            #split에서 해줬는데 왜 안되는지 모르겠음
            #leaf node 연결
            if childNode.getChild().isLeaf:
                # print("child is leaf; update linkedlist")
                childNode.records[0].child.next = childNode.next;

        if node.keyCount == DEGREE and node == ROOT:
            # printTree(node, Color.RED);
            # print("ROOT overflow. Splitting:",node);

            tempL = node.getChild(DEGREE//2);
            tempR = node.next;

            node = splitNode(node);

            node.records[0].child.next = tempL;
            node.next.next = tempR;

    return node;
```

1. 선형 탐색을 통해 삽입할 위치를 정한다.
    a. Leaf Node인데 이미 같은 key값이 있으면 삽입 없이 종료한다.
2. 해당 node가 Leaf Node이라면 addRecord를 통해 record를 삽입해준다.
    a. 삽입 이후 해당 node의 overflow 여부를 확인하는데, node가 ROOT인 경우에만 split을 수행한다.
3. node가 Non-leaf Node라면 하위 child Node에 재귀적으로 삽입을 해주어야 한다. 위 선형 탐색에서 정한 pos가 node.keyCount과 같다는 것은, 삽입할 key값이 node의 records의 모든 key값보다 크거나, 맨 마지막 record의 key값과 동일할 경우이다. 이 경우 node의 가장 오른쪽 Node, 즉 node.next이 childNode가 된다. 그 외엔 node.getChild(pos)이 childNode가 된다. childNode에 insertKey를 해준다.
4. 만약 삽입 후 childNode에 overflow가 발생한다면 childNode를 splitNode해주고, node와 병합해준다. splitNode에서 이미 수행한 과정들인데, 반환이 되면서 일부 정보가 제대로 반영이 안된 것 같다. 이 부분은 끝내 해결을 못했으며 때문에 코드가 좀 지저분해지게 되었다.
5. 병합 이후 node의 overflow 여부를 확인하는데, node가 ROOT인 경우에만 split을 수행한다. 이 부분도 상당히 지저분하다.
6. node를 return한다.

---

getLeftMostKeyAndValue(node)

인자로 받아온 node 기준으로 가장 왼쪽 Leaf Node의 key와 value를 return하는 함수이다. delete 과정에서 Non-leaf Node에 있는 키 값을 대체하는 상황에서 후계자를 찾는 용도로 썼다.

```
def getLeftMostKeyAndValue(node):

    successor = node;

    while not successor.isLeaf:
        if successor.keyCount != 0: successor = successor.getChild();
        # merge과정 후 node의 child가 1개가 되는 불가피한 상황을 위해서
        else: successor = successor.next;

    if successor.keyCount == 0:
        return None, None;
    else:
        # print("successor key =",successor.records[0].key)
        return successor.records[0].key, successor.records[0].value;
```

1. node에서 시작해 successor가 Leaf Node로 내려갈때까지
    a. successor.keyCount가 0이면 successor의 next로 내려간다. delete 과정에서, 특히 DEGREE가 3, 4일때 merge를 하면 종종 Node가 비어버리는 상황이 생겨 이렇게 처리했다.
    b. successor.keyCount가 0이 아니면 successor의 가장 왼쪽 child인 successor.getChild()로 내려간다. (getChild(pos=0)으로 optional parameter를 사용했다)
2. Leaf Node에도 keyCount가 0이면 None, None을 return 한다.
3. Leaf Node의 가장 왼쪽, 즉 0번째 인덱스의 Record의 key와 value를 return한다.

deleteKey(node, key)

코드를 짜기 시작할 때 큰 틀을 제대로 판단하지 못했던 것 같다. 상당히 지저분하고, 시간 그리고 메모리 관점에서도 정말 아쉬운 코드가 되어버렸다. Ruleset은 위 Summary에서 기술한대로이다.

```
def deleteKey(node, key):
    # 삭제/탐색 위치
    pos = 0;

    if node.isLeaf:

        while pos < node.keyCount:
            if key == node.records[pos].key: break;
            pos+=1;

        if pos == node.keyCount:
            return node;
        # leaf에서 그냥 삭제
        node.removeRecord(pos);


    # child로 내려가서 삭제
    else:
        # non-leaf node에도 존재한다면 이후 지워줘야함
        inIndexNode = False;

        while pos < node.keyCount:
            if key < node.records[pos].key: break;
            if key == node.records[pos].key:
                # print("Key to delete found in indexNode",node);
                inIndexNode = True;
            pos+=1;

        # key 지울 childNode
        childNode = None;
        if pos == node.keyCount: childNode = node.next;
        else: childNode = node.getChild(pos);

        childNode = deleteKey(childNode, key);

        minKeys = DEGREE//2-1 if DEGREE%2==0 else DEGREE//2;

        # 자식 underflow
        if childNode.keyCount < minKeys:
            # print("childNode underflow",childNode);

            leftChild = None;
            rightChild = None;

            if pos > 0: leftChild = node.getChild(pos-1);

            if pos+1 == node.keyCount: rightChild = node.next;
            elif pos+1 < node.keyCount: rightChild = node.getChild(pos+1);

            # 왼쪽에서 Steal
            if leftChild is not None and leftChild.keyCount > minKeys:

                if childNode.isLeaf:
                    # print("Stealing",leftChild.records[leftChild.keyCount-1].key,"from leftChild",leftChild);
                    stealKey = leftChild.records[leftChild.keyCount-1].key
                    stealValue = leftChild.removeRecord(leftChild.keyCount-1).value;
                    # 이렇게 새로 Record 안 만들어주면 이상해짐
                    childNode.addRecord(0, Record(stealKey, stealValue));
                    node.records[pos-1] = Record(stealKey, stealValue, leftChild);
```

```
            # 부모 통해서 뺏기
            else:
                # redistribute 전에도 미리 해줘야 한다.
                if inIndexNode:
                    for i in range(node.keyCount):
                        if key == node.records[i].key:
                            # print("updating indexnode",node);
                            nKey, nVal = getLeftMostKeyAndValue(childNode);
                            node.records[i].key = nKey;
                            node.records[i].value = nVal;
                            # print("updated indexnode",node);
                            break;
                    inIndexNode = False;

                # print("redistributing with leftChild");

                # 부모
                stealKey = node.records[pos-1].key;
                stealValue = node.removeRecord(pos-1).value;
                # print(stealKey,"from parent, for",childNode);

                # 부모에서 childNode로
                childNode.addRecord(0, Record(stealKey, stealValue, leftChild.next));

                # leftChild의 new next
                tempR = leftChild.getChild(leftChild.keyCount-1);

                # leftChild
                stealKey = leftChild.records[leftChild.keyCount-1].key;
                stealValue = leftChild.removeRecord(leftChild.keyCount-1).value;
                # print(stealKey,"from leftChild, for",node);

                # leftChild에서 부모로
                node.addRecord(pos-1, Record(stealKey, stealValue, leftChild));

                leftChild.next = tempR;

        # 오른쪽에서 Steal
        elif rightChild is not None and rightChild.keyCount > minKeys:

            if childNode.isLeaf:
                # print("Stealing",rightChild.records[0].key,"from rightChild",rightChild);

                stealKey= rightChild.records[0].key
                stealValue = rightChild.removeRecord(0).value;

                childNode.addRecord(childNode.keyCount, Record(stealKey, stealValue));
                # 여기선 rightChild에서 successor 찾는게 맞다
                # node.swapKey(pos, getLeftMostKey(rightChild));
                # node.records[pos].key = getLeftMostKey(rightChild);
                stealKey= rightChild.records[0].key
                stealValue = rightChild.records[0].value;

                node.records[pos] = Record(stealKey, stealValue, childNode);

            else:
                # redistribute 전에도 미리 해줘야 한다.
                if inIndexNode:
                    for i in range(node.keyCount):
                        if key == node.records[i].key:
                            # print("updating indexnode",node);
                            nKey, nVal = getLeftMostKeyAndValue(childNode);
                            node.records[i].key = nKey;
                            node.records[i].value = nVal;
                            # print("updated indexnode",node);
                            break;
                    inIndexNode = False;

                # print("redistributing with rightChild");

                stealKey = node.records[pos].key;
```

```
                    stealValue = node.removeRecord(pos).value;

                    # 부모에서 childNode로
                    childNode.addRecord(childNode.keyCount, Record(stealKey, stealValue, childNode.next));

                    # childNode의 new next
                    tempR = rightChild.getChild();

                    stealKey = rightChild.records[0].key;
                    stealValue = rightChild.removeRecord(0).value;

                    # rightChild에서 부모로
                    node.addRecord(pos, Record(stealKey, stealValue, childNode));

                    # childNode의 next 갱신
                    childNode.next = tempR;

            # Merge
            else:
                # Merge할땐 이걸 미리 해줘야 할 듯?
                if inIndexNode:
                    for i in range(node.keyCount):
                        if key == node.records[i].key:
                            # print("updating indexnode",node);
                            nKey, nVal = getLeftMostKeyAndValue(childNode);
                            node.records[i].key = nKey;
                            node.records[i].value = nVal;
                            # print("updated indexnode",node);
                            break;
                    inIndexNode = False;

                if leftChild is not None:

                    # print("merging with leftChild", leftChild);

                    temp = leftChild.keyCount;

                    # childNode를 leftChild로 이동
                    for i in range(childNode.keyCount):
                        leftChild.addRecord(leftChild.keyCount+1, childNode.removeRecord(0));

                    # childNode를 가리키던 child포인터 leftChild를 가리키도록 갱신
                    if pos == node.keyCount: node.next = leftChild;
                    else: node.records[pos].child = leftChild;

                    # parentNode에서 빼주기
                    if childNode.isLeaf:
                        # leftChild가 존재하므로 pos-1은 무조건 있
                        node.removeRecord(pos-1);
                    else:
                        # parentNode에서 하나 주기
                        # print("Getting",node.records[pos-1].key,"from parent")
                        # print("temp =",temp)
                        leftChild.addRecord(temp, node.removeRecord(pos-1));
                        leftChild.records[temp].child = leftChild.next;

                    # leftChild.next 갱신
                    leftChild.next = childNode.next;

                    if node == ROOT and node.keyCount == 0:
                        node = leftChild;

                # 사실상 mergeRight는 childNode가 가장 왼쪽에 있을때만 실행
                else:
                    # print("merging with rightChild", rightChild);

                    # ROOT의 record가 들어갈 위치
                    temp = childNode.keyCount;

                    # rightChild에서 childNode으로 이동
                    for i in range(rightChild.keyCount):
                        childNode.addRecord(temp+i, rightChild.removeRecord(0));
```

```
                    # rightChild를 가리키던 포인터 childNode로 갱신
                    if pos+1 == node.keyCount: node.next = childNode;
                    else: node.records[pos+1].child = childNode;

                    if childNode.isLeaf:
                        node.removeRecord(pos);
                    else:
                        # print("Getting",node.records[pos].key,"from parent")
                        childNode.addRecord(temp, node.removeRecord(pos))
                        childNode.records[temp].child = childNode.next;

                    childNode.next = rightChild.next;

                    if node == ROOT and node.keyCount == 0:
                        # print("fixing empty ROOT.")
                        node = node.next;

                    # printTree(ROOT, Color.BLUE)

        if inIndexNode:
            for i in range(node.keyCount):
                if key == node.records[i].key:
                    # print("updating indexnode",node);
                    nKey, nVal = getLeftMostKeyAndValue(childNode);
                    node.records[i].key = nKey;
                    node.records[i].value = nVal;
                    # print("updated indexnode",node);
                    break;

    return node;
```

1. node가 Leaf Node인 경우
    a. pos로 선형 탐색을 한다.
    b. key값이 node에 존재하지 않는다면 종료한다.
    c. removeRecord로 해당 key를 제거한다.
2. node가 Non-leaf Node인 경우
    a. pos로 선형 탐색을 한다.
    b. node에도 key값이 존재한다면 inIndexNode를 True로 해준다.
    c. 위 insertKey와 마찬가지 방식으로 childNode를 정한다.
    d. 최소 키 개수, 즉 $ceil(Degree/2) - 1$를 minKeys에 저장한다.
    e. 만약 childNode의 keyCount가 minKeys보다 적으면
        0. leftChild와 rightChild을 정한다. 위 Summary에서 적은 leftSibling, rightSibling 역할이다.
        1. leftChild가 None이 아니고 keyCount에 여유가 있으면
            a. childNode가 Leaf Node이라면 # Steal Left
                i. leftChild의 가장 오른쪽 Record 하나를 Steal해 온다.
                ii. node의 pos번째 Record의 key도 이 Steal해온 key로 갱신해준다.
                iii. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.
            b. childNode가 Non-leaf Node이라면 # Redistribute with Left

i. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.

ii. node의 Record 하나를 childNode에게 준다.

iii. leftChild의 Record 하나를 node에게 준다.

2. rightChild가 None이 아니고 keyCount에 여유가 있으면

a. childNode가 Leaf Node이라면 # Steal Right

i. rightChild의 가장 왼쪽 Record 하나를 Steal해 온다.

ii. node의 pos번째 Record의 key를 rightChild의 가장 왼쪽 Record의 key로 갱신해준다.

iii. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.

b. childNode가 Non-leaf Node이라면 # Redistribute with Right

i. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.

ii. node의 Record 하나를 childNode에게 준다.

iii. rightChild의 Record 하나를 node에게 준다.

3. leftChild가 None이 아니라면 # Merge Left

a. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.

b. childNode의 모든 Record들을 leftChild에게로 옮긴다.

c. childNode가 Non-leaf Node라면 node에서 pos-1번째 Record를 leftChild에게 준다.

d. childNode가 Leaf Node라면 node의 pos-1번째 Record를 그냥 지운다.

e. 만약 node가 ROOT이면서 keyCount가 0으로 비어버리게 되었다면 node = leftChild로 ROOT를 갱신해준다.

4. rightChild가 None이 아니라면 # Merge Right

a. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다. inIndexNode를 False로 바꾼다.

b. rightChild의 모든 Record들을 childNode으로 옮긴다.

c. childNode가 Non-leaf Node라면 node에서 pos번째 Record를 childNode에게 준다.

d. childNode가 Leaf Node라면 node의 pos번째 Record를 그냥 지운다.

e. 만약 node가 ROOT이면서 keyCount가 0으로 비어버리게 되었다면 node = node.next로 ROOT를 갱신해준다.

f. inIndexNode이라면 다시 선형탐색을 통해 node에 있는 key와 value값을 getLeftMostKeyAndValue(childNode)로 갱신해준다.

3. node를 return한다.

inIndexNode일때 굳이 다시 선형탐색을 한 이유는, 나의 코드 디자인상 steal, redistribute, merge 과정 이후엔 이미 그 key값이 node에서 빠졌을 수 있기 때문이다.

singleKeySearch(key)

```
def singleKeySearch(key):

    search = ROOT;
    path = "";

    while not search.isLeaf:

        pos = 0;
        path+=str(search)+"\n";

        while pos < search.keyCount:
            if key <= search.records[pos].key:
                break;
            pos+=1;
        # 일치하면 그 오른쪽으로 들어가야하니까
        if pos < search.keyCount:
            if key == search.records[pos].key:
                pos+=1;

        if pos == search.keyCount:
            search = search.next;
        else:
            search = search.getChild(pos);

    pos = 0;
    while pos < search.keyCount:
        if key <= search.records[pos].key:
            break;
        pos+=1;

    if pos < search.keyCount and key == search.records[pos].key:
        path+=str(search.records[pos].value);
        print(path);

    else: print("NOT FOUND");
```

1. ROOT 부터 시작해서 search가 Leaf 단계로 내려갈 때까지
    a. path 문자열에 search을 추가한다.
    b. 선형 탐색으로 search할 다음 Node를 정한다.
2. Leaf Node으로 내려왔으니
    a. 선형 탐색으로 일치하는 key값을 가진 Record을 찾는다.
    b. 존재한다면 path에 해당 Record를 추가하고 출력한다.
    c. 존재하지 않으면 NOT FOUND를 출력한다.

rangedSearch(start, end)

```
def rangedSearch(start, end):

    search = ROOT;
```

```
    res = "";

    while not search.isLeaf:
        pos = 0;
        while pos < search.keyCount:
            if start <= search.records[pos].key:
                break;
            pos+=1;
        if pos < search.keyCount:
            if start == search.records[pos].key:
                pos+=1;

        if pos == search.keyCount:
            search = search.next;
        else:
            search = search.getChild(pos);

    while search is not None:
        for record in search.records:
            if record.key > end:
                if len(res) == 0:
                    res = "NOT FOUND";
                print(res);
                return;
            if start <= record.key:
                res+=str(record)+"\n";
        search = search.next;

    if len(res) == 0:
        res = "NOT FOUND";

    print(res);
```

1. ROOT 부터 시작해서 search가 Leaf 단계로 내려갈 때까지
    a. 선형 탐색으로 search할 다음 Node를 정한다.
2. Leaf 단계로 내려왔으니 search가 None일 때까지 linked list를 탐색하면서
    a. end 보다 큰 key를 가진 Record를 발견하면
        i. res 문자열이 비어 있다면 NOT FOUND를 출력하고 종료한다.
        ii. res 을 출력하고 종료한다.
    b. start 보다 key값이 큰 Record이면 res 문자열에 추가한다.
3. search가 None일때까지 탐색을 끝냈으면, 즉 linked list의 끝까지 탐색했으면
    a. res 문자열이 비어 있다면 NOT FOUND를 출력하고 종료한다.
    b. res 을 출력하고 종료한다.

---

saveTree(fileName)

```
def saveTree(fileName):
    global ROOT;

    with open(fileName, "w") as file:
        file.write("#"+str(DEGREE)+"\n");
        q = [ROOT];
        while len(q) != 0:
```

```
            t = len(q);
            for i in range(t):
                cur = q.pop(0);
                file.write(str(cur));
                if i != t-1:
                    file.write("/");
                if not cur.isLeaf:
                    for record in cur.records:
                        q.append(record.child);
                    q.append(cur.next);
            file.write("\n");
```

Save 과정에서는 너비우선탐색(BFS)을 활용하였다. 먼저 DEGREE를 첫줄에 입력한다. Queue에 ROOT를 push해둔다. 이제 pop을 하면서 파일에 Node정보를 저장하고, 또 Node의 각 Record의 child들, 마지막으로 Node의 next까지 Queue에 push해준다. 이 과정은 leafNode 단계까지 내려간다.

---

loadTree(fileName)

```
def loadTree(fileName):
    global DEGREE, ROOT;

    with open(fileName, "r") as file:

        lines = file.readlines();

        # 첫번째 줄
        DEGREE = int(lines.pop(0)[1:]);

        # with open 에서 return해도 close는 자동으로 실행
        if len(lines) == 0:
            ROOT = Node();
            return;

        # leafNode 정보, 마지막 줄
        lastLine = lines.pop();
        leafNodes = lastLine.split("/");

        # reverse bfs
        children = [];

        for leafNode in leafNodes:
            new = Node();

            # linkedList 연결
            if len(children) != 0:
                children[len(children)-1].next = new;

            records = leafNode.split(",");

            for record in records:
                key, value = map(int, record.split(":"));
                new.addRecord(new.keyCount, Record(key, value));

            children.append(new);

        # 역순으로 올라가면서 child연결
        for line in reversed(lines):
            nodes = line.split("/");

            for node in nodes:
                new = Node();
                new.isLeaf = False;
```

```
                records = node.split(",");

                for record in records:
                    key = int(record);
                    child = children.pop(0);
                    new.addRecord(new.keyCount, Record(key, None, child));

                new.next = children.pop(0);
                children.append(new);

        ROOT = children[0];
```

Load 과정도 Queue를 활용한다는 점에서 Save와 비슷하지만, 순서가 정반대이다. 일단 첫줄에서 DEGREE를 읽어 온다. ROOT에서 시작하지 않고, 가장 아랫줄 즉 Leaf Node 단계에서부터 읽어오며 Queue에 push한다. 다음 한 줄씩 올라가며 Node를 Queue에 push하고, 해당 Node의 각 record마다 Queue를 pop하며 child를 연결해주고 추가적으로 pop을 한번 더 해 Node의 next까지 연결해주는 방식이다. 반복문이 끝나면 Queue엔 가장 위의 Node, 즉 ROOT가 되어야할 Node만 남게 되는데, 전역변수인 ROOT를 이 Node로 갱신해준다.

---

main()

```
def main():
    global ROOT;
    # create
    if sys.argv[1] == "-c":
        with open(sys.argv[2], "w") as file:
            file.write("#"+sys.argv[3]);
    # insert
    elif sys.argv[1] == "-i":
        loadTree(sys.argv[2]);
        with open(sys.argv[3],"r") as file:
            insertList = csv.reader(file, delimiter=",");
            for insert in insertList:
                key = int(insert[0]);
                value = int(insert[1]);
                ROOT = insertKey(ROOT, Record(key, value));
        saveTree(sys.argv[2]);
    # delete
    elif sys.argv[1] == "-d":
        loadTree(sys.argv[2]);
        with open(sys.argv[3], "r") as file:
            deleteList = csv.reader(file);
            for delete in deleteList:
                ROOT = deleteKey(ROOT, int(delete[0]));
        saveTree(sys.argv[2]);
    # singleKeySearch
    elif sys.argv[1] == "-s":
        loadTree(sys.argv[2]);
        singleKeySearch(int(sys.argv[3]));
    # rangedSearch
    elif sys.argv[1] == "-r":
        loadTree(sys.argv[2]);
        start = int(sys.argv[3]);
        end = int(sys.argv[4]);
        rangedSearch(start, end);
```

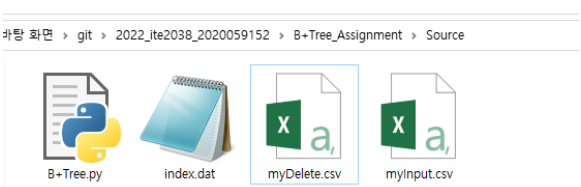
- create: 저장할 파일 첫줄에 DEGREE를 입력한다.
- insert: loadTree로 Tree 정보를 가져오고, csv 파일에 있는 값들을 순서대로 insertKey 해준다. 마치고 saveTree로 Tree를 파일에 저장한다.

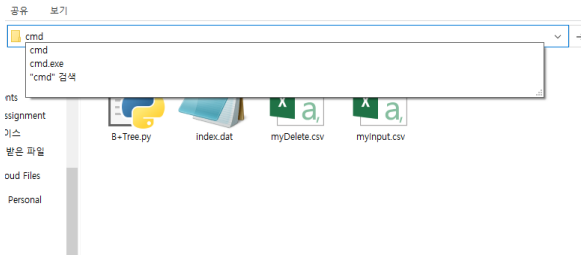
- delete: loadTree로 Tree 정보를 가져오고, csv 파일에 있는 값들을 순서대로 deleteKey 해준다. 마치고 saveTree로 Tree를 파일에 저장한다.
- singleKeySearch: loadTree로 Tree 정보를 가져오고, singleKeySearch를 호출한다.
- rangedSearch: loadTree로 Tree 정보를 가져오고, rangedSearch를 호출한다.

---

3. Instructions for compiling your source code

B+Tree_Assignment/Source에 소스코드 “B+Tree.py”가 있습니다. 이 경로에서 터미널(저는 cmd를 썼습니다)을 실행합니다.

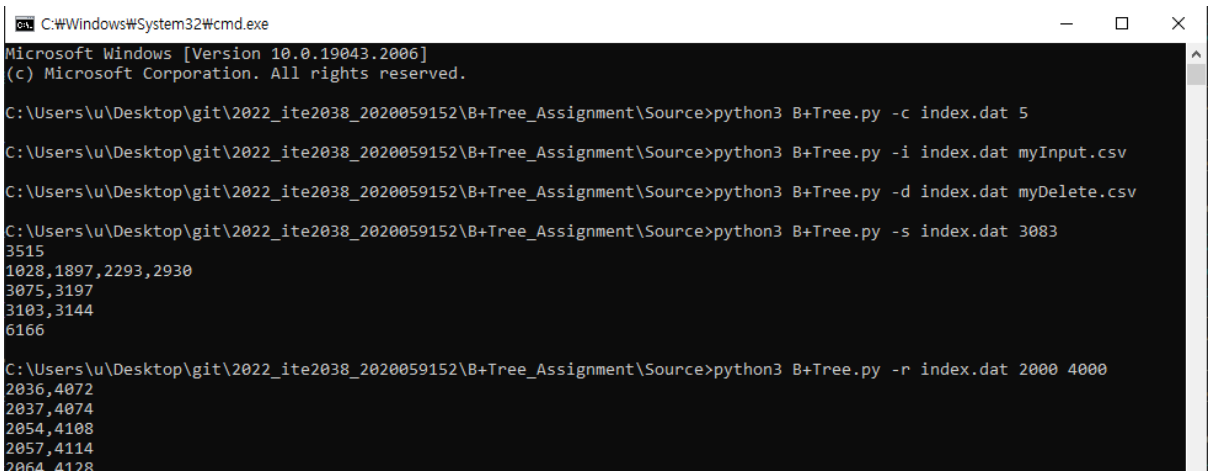




insert과 delete에서 쓰일 csv 파일들도 이 Source 폴더 아래 위치해주셔야 하며, index 파일도 이 경로에 생성됩니다.

아래와 같은 형식으로 명령어를 실행하면 됩니다.

python3 B+Tree.py -(명령어)



4. Any other specification of your implementation and testing

제 PC 기준으로 m = 5의 경우 약 1,000,000개의 insert, delete 작업이 각각 71초, 67초 정도로 측정되었습니다. 비록 많이 느리더리도, 수천 가지의 테스트 케이스를 입력해 본 결과 무한 반복문이나 비정상적인 프로그램 종료 등의 현상은 없었습니다. 이렇게 입력값이 많으면 insert나 delete의 과정이 오래 걸리므로 프로그램이 멈춘 것 처럼 보일 수 있으니 양해 부탁드립니다. 감사합니다.