REPORT

응용미술교육과
2018030328 신영은

Instructions :

Create: python3 main.py c [index.dat]

```
sin-yeong-eun-ui-MacBookPro:Source youngeun$ python3 main.py c index.dat 3
```

Insert: python3 main.py i [input.csv] [index.dat]

```
sin-yeong-eun-ui-MacBookPro:Source youngeun$ python3 main.py i input.csv index.dat
```

Delete: python3 main.py d [delete.csv] [index.dat]

```
sin-yeong-eun-ui-MacBookPro:Source youngeun$ python3 main.py d delete.csv index.dat
```

SingleSearch: python3 main.py s [key] [index.dat]

```
sin-yeong-eun-ui-MacBookPro:Source youngeun$ python3 main.py s 86 index.dat
26
68,86
86,87
<67945>
```

RangeSearch: python3 main.py r [start_key] [end_key] [index.dat]

```
sin-yeong-eun-ui-MacBookPro:Source youngeun$ python3 main.py r 21 100 index.dat
<26><1290832>
<37><2132>
<68><97321>
<84><431142>
<86><67945>
<87><984796>
```

Class 설명 :

`class BpNode(object):` 일반 index 노드 용 class

`def __init__(self, max):`	
`self.max = max`	m(최대 child 수) 저장
`self.key = []`	key 저장
`self.parent: BpNode = None`	parent node 저장
`self.children: BpNode[max] = None`	child node 저장 (리스트 형태)
`def isLeaf(self):`	Leaf node 가 아니므로 false 반환
`def isFull(self):`	최대 key 값(max) 만큼 차 있는지 확인
`def isUnderflow(self):`	최소 |m/2|를 충족하지 않는지 확인
`def cannotBorrow(self):`	형제 node 에게 빌려줄 수 없는지 확인
`def split(self):`	일반 index node 용 split
`def borrowLeft(self, sibling, index):`	일반 index 용 좌측 형제에게 빌려오기
`def borrowRight(self, sibling, index):`	일반 index 용 우측 형제에게 빌려오기

`class BpLeaf(BpNode):` leaf 노드 용 class

`def __init__(self, max):`	
`super().__init__(max)`	
`self.right: BpLeaf = None`	우측 leaf 노드 저장
`self.value = []`	value 저장
`def isLeaf(self):`	Leaf node 이므로 True 반환
`def split(self):`	Leaf node 용 split
`def newItem(self,input_key,input_value):`	새로운 key, value 넣기
`def borrowLeft(self, sibling, index):`	leaf 용 좌측 형제에게 빌려오기
`def borrowRight(self, sibling, index):`	leaf 용 우측 형제에게 빌려오기

`class BpTree(object):` tree class

`def __init__(self, max):`	
`self.root: BpNode = BpLeaf(max)`	root node 저장
`self.max: int = max`	m 저장
`self.leaves: BpLeaf[max] = []`	build 시 leaf node 연결 용 리스트
`def createTree(self, parent, lines):`	index.dat 불러와 트리 빌드
`def connectLeaf(self):`	leaf node 우측으로 연결
`def saveTree(self, node, f):`	트리를 index.dat 로 저장
`def indexSearch(self, key):`	리프노드까지 인덱스 따라 내려가기
`def singleSearch(self, key):`	leaf node 에서 key 에 해당하는 value 찾기 (없으면 false 반환)
`def rangeSearch(self,start_key,end_key):`	시작 리프와 index 를 찾은 후 end_key 이하까지 우측 노드를 따라 가며 출력
`def searchPlace(self, key):`	insert 와 delete 할 위치 찾기
`def insertNode(self, input):`	넣을 node 를 찾은 후, newItem 을 호출해 새로운 data 넣기
`def deleteNode(self, input):`	삭제할 data 를 찾은 후 양 옆 형제노드의 상태를 확인해 적절한 삭제 알고리즘 호출
`def leftSibling(self, node, index):`	좌측 형제노드 불러오기
`def rightSibling(self, node, index):`	우측 형제노드 불러오기
`def merge(self,leftNode,rightNode,key):`	두 형제 노드를 합친 후, 부모 노드에서 우측 노드의 인덱스에 해당하는 key 제거

`def main():`	cli 에서 입력한 명령에 맞는 함수 실행

알고리즘 설명 :

1. insertNode :
   먼저 insert 할 위치를 searchPlace 함수를 호출하여 찾는다. 만약 찾은 leaf 에 빈 공간이 있으면 newItem 을 호출하여 리프 내 적절한 위치에 삽입한다.
   만약 leaf 에 공간이 없으면 먼저 newItem 함수로 적절한 위치에 삽입한 후 오버플로우 된 상태의 노드를 split 해준다.

2. split:
   노드의 중간 값을 기준으로 2 개의 노드로 분리해준다. rightNode 를 새로 생성한 후 기존 노드의 중간 값 부터 마지막 값 까지 key 와 value (index 노드는 children)을 rightNode 로 이동한다.
   split 하는 노드가 부모가 없으면, 새 index 노드 parent 를 생성한 후 중간 key 를 parent 로 복사한다. Parent node 의 children 에 자신과 새로 만든 rightNode 를 넣고, 자신과 rightNode 의 부모 노드에 parent 를 저장한다.
   이미 부모가 있으면 rightNode 의 부모 노드를 자신의 부모 노드로 한다. 그리고 부모 노드에 새로운 key 를 삽입할 index 를 찾고, 해당 위치에 중간 key 를 삽입한다. 마찬가지로 부모 노드와 두 자식 노드를 연결한다. 이 때 부모 노드가 오버플로우 된 경우, 부모 노드에 대해서 split 을 호출한다.
   split 한 node 가 leaf 인 경우 - 중간 값을 leaf 에 그대로 보존하면서 key 를 복사해서 부모로 올리기
   split 한 node 가 index 인 경우 - 중간 값을 보존하지 않고 부모에 삽입

3. deleteNode :
   먼저 delete 할 node 를 searchPlace 함수를 호출하여 찾은 후 제거한다. 만약 key 를 제거한 노드가 underflow 되면 좌우의 sibling 노드를 불러온다.
   리프 노드의 경우 만약 오른쪽 sibling 이 존재하고 오른쪽 sibling 에서 하나를 빌려와도 underflow 가 일어나지 않으면 오른쪽 sibling 의 최소 key 값을 자신의 key 마지막에 붙이고, 오른쪽(빌려 준) 노드의 부모 인덱스 key 를 노드 key 의 첫번째 값으로 바꾼다. 해당 조건이 불가능한 경우 왼쪽 sibling 이 존재하고 빌려올 수 있을 때, 왼쪽 sibling 의 마지막 값을 자신의 key 첫번째에 붙이고, 자신의 부모 key 를 새로 붙인 값으로 바꾼다. 양 측에서 모두 빌려올 수 없으면 오른쪽 sibling 이 존재하면 오른쪽과 자신을 merge 한다. 오른쪽이 없으면 왼쪽 sibling 과 자신을 merge 한다.
   인덱스 노드의 경우 만약 오른쪽 sibling 이 존재하고 빌려올 수 있으면 오른쪽 sibling 의 첫번째 key 를 부모 index 위치로 올리고, 부모의 key 는 rotate 되어 자신의 마지막 key 로 내려온다. 불가능한 경우 왼쪽 sibling 이 존재하고 빌려올 수 있으면 왼쪽 형제의 마지막 key 를 부모 key 로 올리고, 부모 key 는 rotate 되어 자신의 첫번째 key 로 내려온다. 양 측 모두 빌려올 수 없으면 오른쪽 형제가 있는 경우 오른쪽과 merge, 왼쪽 형제가 있으면 왼쪽과 merge 한다.
   merge 가 수행되면 부모 노드의 key 개수가 하나 줄어들기 때문에 부모 노드에서 underflow 가 일어날 수 있으므로 root 까지 올라가면서 underflow 발생 여부를 확인하며 위 과정을 반복한다.

4. merge :
   리프노드끼리 merge 할 경우 오른쪽 노드의 부모의 인덱스를 지워버리고 왼쪽 노드에 오른쪽 노드를 합친다. 이 때 리프노드가 아닌 경우 부모 노드 key 를 두 노드 사이로 데려온다. 리프노드는 children 을 고려할 필요가 없기 때문에 부모 key 를 데리고 내려올 필요 없이 두 노드를 합치기만 하면 된다.

5. singleSearch :
   key 를 보고 index 노드를 따라 내려간다. Index 노드의 key 값을 따라 가면서 만약 찾는 key 보다 노드의 n 번째 key 값이 더 크면 n 번째 자식으로 내려간다. Leaf 노드에 도달하면 찾는 key 와 같은 key 가 있으면 value 를 리턴하고, 없으면 None 을 리턴한다.

6. rangeSearch :
   singleSearch 와 같은 방법으로 start_key 를 찾아 내려간다. 리프노드에 도달하여 시작점을 찾으면 node 의 key 와 value 를 순서대로 출력하고, end_key 를 만나기 전에 리프 노드의 끝에 도달하면 해당 노드의 right node 로 넘어가 같은 과정을 반복한다.