B+ Tree Implementation

1. Summary of algoritm

1. Struct summary

1. Pair

   key, value를 갖는 class.

2. Node

   a. List<Pair> list : pair들을 담는 arraylist.

   b. isleaf, isroot : 현재 노드가 leaf인지, root인지를 나타내 주는 boolean.

   c. child : 현재 노드의 child를 저장하는 arraylist. 이 arraylist 안엔 Node타입의 node들이 들어가 있습니다.

   d. leftsibling, rightsibling : 현재 노드가 leaf일때, 각각 왼쪽과 오른쪽 sibling을 가리켜주는 Node type 변수. (Double Linked list 개념입니다.)

   e. nodesize : 현재 노드의 크기를 저장하는 변수.

   Node() 생성자에서는, 기본적으로 생성되는 node는 leaf라 가정하고 boolean들을 만들어주고, 특히 나중에 insert와 delete에서 index 접근을 위해서 list와 child 두개의 arraylist들을 for문을 이용해서 null들 넣어줌으로써, 크기를 maxSize + 2로 늘려놓았습니다.

3. static 변수들

   a. root : 현재 b+tree의 실제 root정보를 담아주는 Node type static 변수.

   b. maxSize : 한 node가 담을 수 있는 최대 pair의 갯수.

   c. minkeyNum : 한 node가 담아야만 하는 최소 pair의 갯수.

   d. Tree : read_file method에서 index.dat를 읽어들여서 tree구조를 재구축하기 위해 node정보들을 담는 arraylist.

2. B+Tree 구현 관련 Method summary

1. CreateTree()

   command line에서 입력받은 order로 static변수인 minkeynum을 초기화 해주고, static변수인 root를 만들어주는 method.

   여기서 minkeynum초기화 이후에 maxSize를 하나 줄여주는데, 이유는 과제 초반 구현중, command line에서 입력받는 order가 한 node가 담을 수 있는 최대 pair의 갯수로 생각하고 insert 구현중에 있었지만, 나중에 max_degree(b+tree가 가질 수 있는 최대 child 간선의 갯수)로 재 명시가 올라와서, split함수에서 쓰이는 maxSize를 제 의도에 맞게 실행시키려면 부득이하게 maxSize를 하나 줄여주어야 했습니다.

2. Insert(Node CurNode, int key, int value)

   항상 Insert는 root에서부터 시작해서, 삽입하려는 key와 현재 노드의 list에 들어있는 pair의 key값과 비교하면서 leaf까지 내려갑니다. root에서부터 내려가면서 278 line의 재귀함수를 실행시키면서 최종적으로 leaf에 key값을 삽입시켜주고, leaf에 삽입이 완료되면 재귀함수가 풀리면서 반대로 root로 올라가면서, curNode의 child의 overflow를 검사해 줍니다. 이때 case가 크게 2가지로 나뉩니다.

   case 1 : 처음부터 curNode가 root이면서, leaf인 경우.

      case 1 - 1 : root가 overflow가 나는 경우.

         해당 node를 split하여 생긴 newParent node를 새로운 root로 지정시켜주고, 함수를 종료시킵니다.

      case 1 - 2 : overflow가 나지 않는 경우.

         B+ Tree 성질에 위배되지 않으므로, curNode를 return하고 함수를 종료시킵니다.

   case 2 : curNode가 root이지만, leaf가 아닌 경우.

      curNode의 child로 이동하면서 key값이 들어가야 하는 최종 leaf node를 탐색하여 key값을 삽입하고, 재귀함수를 종료시키면 2 - 1, 2 - 2과정을 반복합니다. 한번 2 - 1 혹은 2 - 2과정이 실행되면 curNode는 curNode의 부모로 이동하게 됩니다.(재귀함수는 선언된 순서의 반대로 종료되므로.)

      case 2 - 1 : curNode의 child가 overflow가 나는 경우에는, curNode에, child를 split하여 나온 newNode의 자리를 찾아서 붙여주고, curNode를 return시킵니다.

case 2 - 2 : child - overflow가 나지 않는 경우에는, child split이 필요가 없어집니다. 그렇다면 curNode 또한 overflow가 나지 않으므로 curNode를 return해줍니다.

최종적으로 leaf 삽입이 종료되고, 모든 재귀가 종료되고 curNode가 다시 root로 도달한다면, 마지막으로 root의 overflow를 검사하고 overflow가 난다면 해당 root를 split해서 나온 newParent를 새로운 root로 지정하고 최종적으로 한 번의 insertion이 끝나게 됩니다.

3. split(Node curNode) :

   순수하게 curNode를 split만 하고 curNode의 부모인 newParent를 return해주는 함수입니다.

   newSibling은 curNode와 같은 level에 위치하기에 curNode의 isleaf와 같고, curNode가 root였다면 newParent는 새로운 root가 되게 됩니다.

   또한, leftsibling, rightsibling의 갱신은 항상 leaf node가 split될 때만 이루어지기 때문에, split함수 안에서 sibling들을 조정했습니다.

   sibling 조정에서 가장 중요한 포인트는, 쪼개지기 전의 curNode의 rightsibling은 쪼개진 후의 newSibling의 rightsibling이 되어야 하기 때문에, Double Linked List 자료구조에서의 insert할 때의 idea를 도입하여 newSibling의 rightsibling은 curNode의 rightsibling으로 선언해주고, 그 후에 newSibling의 leftsibling을 curNode로 지정해주는 코드를 작성했습니다.(111 ~ 116 line)

   또한 leaf에서의 split, non-leaf에서의 split은 newParent의 key값이 복사 유무에 따라 달라지므로, curNode가 leaf인지, non-leaf인지에 따라 case를 2가지로 나누어 split함수를 구현하였습니다.

4. singleKeysearch(int key) :

   항상 root에서부터 key값을 탐색합니다. node안에 들어있는 list가 갖고 있는 pair의 key값과 찾으려는 key값을 비교하면서 child로 이동해 주고, 방문하는 pair의 key값을 출력하도록 구현하였고, 최종 찾으려는 key값이 leaf node에 있다면, 해당 key값의 value를 출력하였고, leaf node에 도달하고도 해당 key값을 찾지 못한다면 tree 안에 존재하지 않으므로 NOT FOUND를 출력하였습니다. (방문하는 노드의 모든 key값을 출력하는게 아닌, 반복문으로 찾으려는 key와 비교되는 pair의 key들만 출력하였습니다.)

5. rangedSearch(int start, int end) :

출력의 시작은 항상 start값 이상이어야 하므로, singleKeysearch에서와 같이 root에서부터 탐색을 시작하여 start값을 포함하는 leaf node에 도달합니다. 해당 node에서 start값을 찾고, 그 이후로 rightsibling 변수를 이용하여 node를 이동하면서, 각 node에 포함되는 key , value값을 출력하도록 하였습니다.

예외 상황으로는 leaf에 남아있는 최소 key값이 애초에 end값보다 큰 경우가 있는데, 이 경우에는 아무 값도 출력하지 않고 함수를 종료시켰습니다.(216 line 조건문.)

6. delete(Node curNode, int key) :

   concept 자체는 insert와 비슷하게, 재귀함수로 구현하였습니다. 재귀함수로 leaf까지 들어가는 도중에 internal-node에서 해당 key값을 발견하면, 일단 해당 list의 pair값을 null값으로 바꿔주고, leaf까지 들어가서 해당 key값을 삭제해주고, 종료시켜서 curNode의 child node가 underflow가 나는지 검사하는 구조로 구현하였습니다. 이때, curNode의 child node가 underflow가 나지 않는다면, null값으로 바뀐 internal-node의 pair값을, 해당 node의 inorder successor node에 해당하는 node의 첫 번째 pair를 복사해서 가져온 후에 curNode를 return하지만, curNode의 child가 underflow가 난다면, tree 구조를 재구축해야 하는데, 재구축 방법에는 큰 case로는 2가지, 세부족인 case를 총 8가지로 잡았습니다.

   6 - 1 : curNode의 child가 leaf인 경우.

   (모든 redistribute 과정에는 sibling의 size는 하나 줄어들고, child의 size는 하나 늘어납니다.)

   6 - 1 - 1 : redistribute from left.

   curNode 기준으로 underflow가 난 child의 left sibling의 nodeSize를 검사합니다. 만약 nodeSize가 minkeyNum보다 큰 경우, underflow가 난 child(이하 child)의 list에 들어있는 pair들을 오른쪽으로 한 칸씩 밀어주고(왼쪽에서 빌려올 것이므로.), 먼저 left sibling의 가장 오른쪽 끝 pair을 curNode로 가져오고, 그 값을 다시 child에 ‘복사’하고 curNode를 return해줍니다. (B+ Tree의 정의상 leaf node의 key값과 parent의 key값은 같으므로.)

   6 - 1 - 2 : redistribute from right.

   underflow가 난 child(이하 child)의 right sibling의 nodeSize를 검사하고, right sibling의 nodeSize가 minkeyNum보다 큰 경우, right sibling에서 값을 빌려올 것

이므로, child의 list는 따로 조정할 필요가 없습니다.(list 가장 끝에 빌려온 pair를 붙여줄 것이므로.)

그저 curNode의 pair를 먼저 child에 붙여주고, 내려준 pair를 right sibling에서 하나 가져와주고, right sibling의 list를 왼쪽으로 한 칸씩 땡겨주고 curNode를 return 해줍니다.

underflow가 난 child(이하 child)의 left, rightsibling 둘다 nodeSize가 minkeyNum일 경우에는, merge연산을 합니다. merge란, sibling을 child에 붙여주는 작업이므로, sibling.nodeSize + child.nodeSize가 maxSize 이하여야 합니다. 또한 merge작업은 curNode가 leaf이던 아니던, 결과적으로 curNode의 nodeSize가 하나씩 줄어듭니다.

6 - 1 - 3 : merge to right.

left sibling을 child로 옮겨주는 작업입니다. 먼저 left에서 pair들을 받을 것이므로, leftsibling의 nodeSize만큼, child의 pair들을 오른쪽으로 밀어줍니다. 또한, leftsibling의 부모의 pair를 삭제해주고, list까지 조정해줍니다.

merge시킬 조건이 갖춰졌으므로, left sibling의 모든 key값을 child로 옮겨주고, 두 node의 nodeSize 조정 이후에, curNode의 child가 하나 없어진 꼴이므로, curNode의 child를 왼쪽으로 한 칸씩 밀어줍니다.

만약 curNode가 root node였고, merge 이후 curNode.nodeSize가 0이 된다면, merge가 된 child node를 새로운 root로 지정해줍니다. (shrink)

6 - 1 - 4 : merge to left.

right sibling을 child로 옮겨주는 작업입니다. right에서 left로 merge하므로, left list 조정은 필요없습니다. merge to right과 같이 부모 pair를 삭제하고, 부모 list를 조정합니다. child로 merge 이후에도 똑같이 curNode의 child조정을 하고, shrink 작업 역시 동일합니다.

6 - 2: curNode의 child가 non-leaf인 경우.

6 - 2 - 1 : redistribute from left.

6 - 2 - 2 : redistribute from right. 두 redistribution 다 leaf node에서의 작업과 동일합니다. 다만, sibling의 child를 underflow가 난 child의 child로 붙여주는 작업이 추가됩니다.

6 - 2 - 3 : merge to left

6 - 2 - 4 : merge to right

leaf node에서의 merge와 다른 점은, leaf node에서는 parent의 pair를 단순히 '삭제'하였지만, non-leaf node에서의 merge는 parent의 pair를 삭제하지 않고, child로 pair값을 내려주는 작업을 합니다. shrink작업 역시 같습니다.

3. 파일입출력 관련 summary

cmd창에서 입력받는 -c, -i, -d, -s, -r를 switch, case문으로 나누어주었습니다.

methods

1. save(Node root, String data_file) :

   insert, delete가 완료되면 tree 구조가 변할 것이므로, 바뀐 tree 구조를 data_file에 입력해주는 method입니다.

   file에 입력하는 방식은 parameter로 들어온 root(실제로 root를 넣어줍니다.)의 번호를 매기고, Queue 자료구조에 넣어서, 해당 node의 child를 그 다음 번호를 매겨주는 bfs 방식을 채택했습니다.(처음엔 재귀함수(dfs)방식으로 node번호를 매기려 했으나 tree height에 따라 번호매겨주는 방식이 달라져서 bfs로 변경했습니다.) 또한, root node의 node 번호를 ` 항상 1번으로 넣어주었습니다.

   파일 입력 방식은 이렇습니다. 뒤에 소개드릴 read_File에서 parsing을 용이하게 하기 위해 node번호 / key, value / key, value …. / 이런 식으로 node에 들어있는 pair들을 '/' 단위로 끊어주었습니다. 이때, 해당 node가 leaf node라면 node번호 앞에 '#' 기호를 붙여주었습니다.

2. read_File(String data_file) :

   index.dat 파일을 읽어와서 string으로 된 tree구조를 실제 b+tree 구조로 구현해주는 method입니다. index.dat 파일 첫줄에는 항상 maxSize가 저장되어 있고, 그 다음 줄 부터 node번호 / (key, value) / (key, value) …. / … ../ 형태로 저장되어 있습니다. 또한, leaf node인 경우에는 node번호 앞에 #기호를 붙여주었습니다. 또한, node번호가 1이라면,

parsing 방법은, 첫번째 '/' 이전의 node번호(leaf node인 경우엔 '#'기호까지)를 읽고, 나머지 (key, value)를 '/' 기준으로 parsing하여, 해당 노드의 pair값들을 list로 만들어서, Tree arraylist에 node번호로 index를 접근하여 해당 node를 저장하였습니다.

index.dat에 저장된 노드들은 bfs 방식으로 node번호를 매겼으므로, node를 Tree arraylist에 저장할 때 역시 bfs방식으로 저장했습니다.

index.dat를 다 읽고 Tree arraylist에 저장 완료한 이후에는, 1번 노드인 root 노드를 꺼내서 Queue에 넣습니다.

root node부터 Queue에서 꺼내 읽을 때, 그 안에 들어있는 pair갯수는 곧 그 curNode의 nodeSize가 됩니다. 따라서 , 그 node 다음의 갯수만큼 Tree에 있는 Node들을 Queue에 넣고, 그 node들을 curNode의 child로 연결해주는 작업을 반복해줍니다.

command line commands

-c : args[2]를 maxSize로 저장하고, create_tree()method를 호출해줍니다.

-i : file을 읽기 전에, index.dat 파일에 저장되어 있는 tree가 있다면, read_file method로 tree 구조를 만들어 주고, while문으로 input.csv file을 한줄씩 parsing해서 읽어와서 insert를 진행합니다. 모든 연산이 끝나면 save method를 호출합니다.

-d : read_file method로 index.dat에 저장되어 있는 tree구조를 만들어 주고, while문으로 delete.csv file을 한줄씩 parsing해서 읽어와서 delete를 진행합니다. 모든 연산이 끝나면 save method를 호출합니다.

-s : read_file method로 index.dat에 저장되어 있는 tree구조를 만들어 주고, singlekeySearch method를 호출합니다.

-r : read_file method로 index.dat에 저장되어 있는 tree구조를 만들어 주고, rangedSearch method를 호출합니다.

4. 컴파일 방법

```
javac BplusTree/BplusTree.java
```

패키지 밖에서 BplusTree/BplusTree.java로 컴파일 해주시면 됩니다.

(첫번째는 패키지 이름, 두번째는 java파일명입니다.)