B+tree implementation assignment

컴퓨터소프트웨어학부 2019079907 이가은

1. Summary of algorithm

■ class

Node 와 Pair 을 클래스로 정의한다.

Node 는 Pair 의 개수인 m, Pair 의 목록인 p, 가장 오른쪽 child 또는 오른쪽 leaf 인 r 을 가진다.

Pair 은 key, value, child 를 가진다.

■ 데이터 저장

1. 차수를 가장 먼저 저장한다.
2. root 부터 시작해서 모든 노드를 저장한다.
3. node 를 발견할 때마다 count++를 통해 index 를 매긴다.
4. 부모노드의 index, node 의 index 를 저장한 후 node 의 모든 pair 의 key, value 를 저장한다.
5. child 가 없는 leaf node 일 경우 index leaf 에 따로 저장한다.

```
5
* 0 1
26 1290832 / 84 431142 /
* 1 2
9 87632 / 10 84382 / 20 57455 /
* 1 3
26 1290832 / 37 2132 / 68 97321 /
* 1 4
84 431142 / 86 67945 / 87 984796 /
# 2 3 4
null
```

■ 데이터 읽기

1. 첫줄을 M 으로 저장한다.

2. #이 나올 때까지 한줄씩 읽으며 부모 인덱스와 내 인덱스를 저장하고 node 를 저장하는 저장한다. Tree 의 내 인덱스에 pair 들을 차례대로 저장한다.
3. #로 시작하는 줄은 leaf 이므로 r 로 연결한다.

■ 삽입

1. key 를 삽입할 leaf node 를 찾는다.
2. leafNode 의 m 이 M 보다 작다면 종료한다.
3. 그렇지 않다면 splitLeaf 를 통해 쪼갠다. (이때 split 한 노드에도 중앙값이 있어야한다.)
4. 쪼개진 노드의 부모노드에 노드의 key 의 중앙값을 삽입하고 두개의 노드를 연결한다.
5. 쪼개진 노드가 root 일 경우 노드의 key 의 중앙값을 가진 새로운 노드를 만들어서 그 노드를 root node 로 변경한다.
6. 부모노드의 m 이 M 보다 작다면 종료한다.
7. 그렇지 않다면 splitParent 를 통해 쪼갠다. (이때 split 한 노드에 중앙값이 포함되지 않는다.)
8. 4~7 번을 6 번이 발생할 때까지 반복한다.

■ 삭제

1. findCorrectLeaf 를 통해 key 가 존재하는 leafNode 를 찾고 key 를 삭제한다.
2. 삭제된 key 가 leafnode 이외에도 있을 경우 leafnode 에서 가장 작은 key 로 그 값을 변경한다.
3. leafNode 의 m 과 (M-1)/2 를 비교해서 m 이 (M-1)/2 이상이라면 그대로 끝낸다.
4. 그렇지 않다면 sibling node 의 m 을 확인한다.
5. m 이 (M-1)/2 이상이면 key 를 가져온다.
6. 그렇지 않다면 key 를 삭제한 leafNode 의 나머지 pair 를 sibling node 중 하나에 넣는다.
7. 부모노드의 m 이 (M-1)/2 이상인경우 종료한다.

8. 그렇지 않다면 부모노드의 pair 들을 sibling node 중 하나에 전부 넣는다. (부모노드가 root 이면 m 이 1 이상이면 끝낸다.)
9. node 를 삭제하고 node 의 pair 들을 두 sibling 중 하나에 넣는다.
10. 7 이 발생할 때까지 7-8 과정을 반복한다.

■ 단일 키 검색

1. root node 에서 시작한다.
2. findKey 보다 큰 최소 key 를 가진 pair 의 child 로 이동한다.
3. 모든 키가 findKey 보다 크다면 node 의 r 로 이동한다.
4. leaf node 를 찾을 때 까지 2-3 과정을 반복한다.
5. leaf node 에 도달하면 findKey 를 찾는다.

■ 범위 검색

1. root node 에서 시작한다.
2. 단일키검색과 유사하게 leaf node 까지 찾아간다.
3. fromKey 가 존재하는 leaf node 를 찾으면 fromKey 이상의 pair 들을 모두 출력한다.
4. leaf 의 모든 pair 을 출력하면 r 로 넘어가고 toKey 보다 큰 pair 을 만날 때까지 이과정을 반복한다.

2. Detailed description of algorithm

- save data

  -c [indexfile] [m]

  1. M 에 args[2]를 넣는다.
  2. indexFile 을 생성한 후 M 을 쓴다.

- save data

Indexfile 의 형식은 맨윗줄에 M, * 뒤의 숫자가 노드의 부모노드의 index, 노드의 index 이다. 그 다음줄은 노드의 pair 들이다. #뒤에는 leaf 의 index 목록이다. (root 의 parent 는 0 이다.)

1. 차수를 저장한 후 save 함수를 호출하고 leaf 를 저장한다.

2. saveTree 에서는 함수가 호출 될 때마다 count++를 해준다.

3. 노드의 인덱스인 thisN 은 count 이다.

4. 인자로 들어온 부모 인덱스와 thisN 을 저장한다.

5. 노드의 pair 들을 key value 형태로 모두 저장한다.

6. pair 의 child 에 접근한다.

7. 이때 child 가 존재하면 saveTree 를 호출한다.

8. 노드가 non-leaf node 라면 saveTree 를 호출한다.

9. 노드가 leaf 라면 leaf 에 node 의 인덱스를 저장한다.

- read tree

  Index file 에 저장된 트리를 메모리로 가져온다.

  1. save 에서 저장한 양식에 따라 한줄씩 읽는다.

  2. 첫줄은 M 에 저장한다.

  3. #이 나올때까지 한줄씩 읽으며 index 와 pair 을 저장한다.

  4. 인덱스를 통해 child 와 parent 를 연결한다.

  5. #뒤에 있는 것들은 leaf 에 저장한다. leaf node 를 tree 에 접근해 차례대로 오른쪽 node 로 연결한다.

- Insertion

  1. insertion 에서 index file 과 data file 을 입력받는다.

  2. readTree()를 통해 index file 에서 tree 를 불러온다.

  3. data file 을 한줄씩 읽으며 ,를 통해 나누어서 key, value 를 읽고 inser()를 실행한다.

4. insert 에서 findCorrectLeaf 를 통해 key 가 어떤 노드에 삽입되어야 하는지 찾는다.

5. findIndex 를 통해 키가 몇번째 pair 에 존재하는지 찾는다.

6. key 와 val 을 알맞은 자리에 삽입한다.

7. 만약 nowN.m 이 M 보다 작다면 종료하고 크다면 splitLeaf()를 호출한다.

8. splitLeaf()에서 right child 에 해당하는 right 을 만들어준다. right.r 은 node.r 이다.

9. m/2 이상의 노드들을 기존 node 에서 제거하고 right child 에 추가한다.

10. 기존 노드가 root 일 경우 left node 를 만들어주고 m/2 미만의 노드들을 추가해준다. Root 는 중앙값 이외의 값들을 모두 삭제하고 left child 와 right child 를 연결한다.

11. 기존 노드가 root 가 아닐 경우 m/2 이상의 노드들을 다 삭제한다.

12. findParentNode 를 통해 기존 노드의 부모노드를 찾는다.

13. 부모노드의 p 에 중앙값을 삽입하고 left 와 right 를 연결한다.

14. 부모노드의 m 이 M 보다 클 때 splitParent()를 실행한다.

15. splitParent 는 splitLeaf 와 유사하며 중앙값을 포함하지 않는다.

16. 10-15 번을 반복한다.

- Deletion

1. insertion 에서 index file 과 data file 을 입력받는다.

2. readTree()를 통해 index file 에서 tree 를 불러온다.

3. data file 을 한줄씩 읽으며 지울 key 읽고 delete() 를 실행한다.

4. findCorrectLeaf()를 통해 지울 key 를 leaf 에서 찾고 삭제한다. 없을 경우 존재하지 않음을 알려주고 deletion 을 종료한다.

5. nowN 의 m 이 (M-1)/2 보다 크거나 같을 때 changeKey()를 호출하여 key 를 바뀐 node 인 nowN 의 첫번째 key 로 바꾼다.

6. changeKey 에서는 leaf 가 아닌 노드에 존재하던 키가 leaf 에서 사라지거나 위치가 달라졌을 때 해당 키를 다른 키로 바꾼다.

7. sibling 을 찾기위해 findSibling()를 호출하는데 findSibling 에서는 보낸 노드와 같은 부모노드를 가진 left 또는 right node 를 찾는다.

8. 키가 삭제된 후 노드의 m 이 (M-1)/2 보다 작을 경우 left sibling 이 존재하며 pair 가 하나 없어도 m 이 (M-1)/2 이상일 때, right sibling 이 존재하며 pair 가 하나 없어도 m 이 (M-1)/2 이상일 때, 두가지 경우를 만족하지 못하여 leaf 하나가 삭제되고 병합되어야 할 때 3 가지로 나뉜다.

9. 첫번째 경우는 left sibling 의 마지막 pair 를 node 로 옮긴다. 그 후, changeKey()를 호출하여 옮기기 전 node 의 첫번째 키를 leftSibling 에서 옮긴 키로 바꾼다.

10. 두번째 경우는 right sibling 의 첫번째 pair 를 노드로 옮긴다. 그 후, changeKey()를 호출하여 rightSibling 에서 옮긴 첫번째 키를 옮긴 후의 rightSibling 의 첫번째 키로 바꾼다. 바꾼 후 지워진 키가 node 의 첫번째 키였다면 changeKey()를 실행하여 key 를 바뀐 노드의 첫번째 key 로 바꾼다.

11. 세번째 경우는 left sibling 이 존재할 때, right siblig 이 존재할 때 두가지로 나뉜다.

12. left sibling 이 존재할 때 노드의 모든 pair 을 left sibling 의 뒤로 옮기고 parentNode pair 을 하나 줄인다. 그리고 child 를 parent 에 연결 시켜 준다. 만약 parentNode 가 root 이고 parentNode.m 이 0 이라면 root 는 left sibling 이 된다.

13. right sibling 이 존재할 때 노드의 모든 pair 을 right sibling 의 뒤로 옮기고 parentNode 의 pair 을 하나 줄인다. 만약 지워진 key 가 node 의 첫번 째 key 라면 changeKey()를 실행해 key 를 바뀐 노드의 첫번째 키로 바꿔준다.

14. left leaf 가 존재할 때 findLeftLeaf()로 left leaf 를 찾아 right sibling 과 연결한다.

15. 노드를 병합한 후 parentNode 가 root 가 아니고 m 이 (M-1)/2 보다 작은 경우 mergeParent()를 호출한다.

16. mergeParent()는 재귀적으로 호출되며 findParentNode 를 통해 parentNode 를 찾으면 leftSibling 과 right Sibling 을 찾는다.

17. left sibling 이 존재하면 parent 에서 left 와 node 사이에 존재하는 key 하나를 삭제하고 left sibling 의 뒤에 추가한 후 node 의 모든 pair 를 left sibling 뒤로

옮긴다. node 를 삭제한후 left sibling 을 parent 의 child 에 연결하고 node 의 r 을 left 의 r 에 연결한다.

18. 만약 left 의 m 이 M 이상이면 splitParent()를 실행한다.

19. parentNode 가 root 고 pair 가 0 개면 root 를 right sibling 으로 바꾼다.

20. 15- 19 과정을 반복한다.

- single key search

  1. singleSearch 에서 indexFile 과 Findkey 를 입력받는다.

  2. readTree()를 통해 index file 에서 tree 를 불러온다.

  3. singleKeyFind 를 통해 key 를 찾는다. Key 를 찾기 위해 root 로 가는 반복문은 findCorrectLeaf 와 유사하다. root 부터 반복문을 통해 아래로 내려가고 node 에서 key 보다 값이 큰 pair 의 child 로 가는 과정을 반복한다.

  4. key 가 leaf 에 존재하면 값을 출력하고 존재하지 않으면 메시지를 출력한다.

- range key search

  1. rangedSearch 를 통해 indexFile, fromKey, toKey 를 입력받는다.

  2. readTree()를 통해 index file 에서 tree 를 불러온다.

  3. rangedKeyFind 에서 fromKey 를 찾는다. 이과정은 singleKeyFind 와 유사하다.

  4. from key 가 존재하는 리프토드를 찾으면 from key 이상의 pair 을 모두 출력한다.

  5. leaf 의 모든 pair 을 출력하면 r 로 넘어가고 toKey 보다 큰 pair 을 만날 때까지 이과정을 반복한다.

- find parent

  1. 부모노드를 찾기 위해 사용된다.

  2. findParentNode()는 재귀적으로 호출되고 nowN 과 target 을 입력받는다.

  3. DFS 방식으로 타겟노드가 자녀인 부모노드를 찾는다.

4. 전역변수인 parentNode 에 부모노드를 넣고 마찬가지로 전역변수인 parentIndex 에 부모노드에서 타겟노드의 index 를 넣는다.

3. Instructions for compiling source code

(1) 컴파일 방법

javac bptree.java

(2) creation

java bptree -c [index file] [m]

(3) insertion

java bptree -i [index file] [data file]

(4) deletion

java bptree -d [index file] [data file]

(5) single key search

java bptree -s [index file] [key]

(6) ranged key search

java bptree -r [index file] [from key] [to key]

(7) print file

java bptree -p [index file] [m]