Summary of B+tree

수업시간 : 월 13:00 ~ 15:00 목 17:00 ~ 19:00

2018008559 컴퓨터소프트웨어학부 신상윤

0. 서론

저는 java를 써본 적이 없습니다. c++로 작성한 코드를 java로 옮겼고 그래서 모든 함수나 class, 변수 등을 모두 static, public으로 선언했습니다.
불편하시더라도 이쁘게 봐주시면 감사하겠습니다.

1. 전역변수와 class

가. 전역변수

```
static int max_degree; //b+tree가 가질 수 있는 최대 key 개수
static int min_key; //b+tree가 최소한 가져야 하는 key 개수
static node root; //b+tree의 루트 주소, split 함수에서 대입하기 위해 전역변수로 설정
```

밑에는 파일 입출력 부분에서 설명

```
static node[] adress;
static String[] treeinfo;
static String[] outputstring;
static int cnt;
```

나. Class

```
static class data{
        public int gap;
        public data(int v){
                gap = v;
        }
}
```

data class, 우리가 저장할 어떤 data의 정보를 저장하는 객체이다. 이번 과제에서는 value값만 저장한 data지만 실생활에서는 정보들이 추가로 있을 수 있다.

```
static class node{
        public int m; //key의 개수를 저장
        public boolean isleaf; //laef 여부를 저장
        public int[] key;  //key값들을 저장
        public data[] value; //leaf일 때 key들이 가리키는 data를 저장
        public node[] left_child_node; //nonleaf일 때 가리키는 자식들을 저장
```

```
        public node rightnode; //leaf면 다음 형제노드를 아니면 가장 오른쪽에
있는 자식을 저장
        public int num; //트리를 내보내기 위해 각 node에 임의의 값을 부여
        public node(int k,int v,boolean b){
        //기본생성자로 insert할 때 쓰인다.
        }
        public node(String s){
        //String 생성자로 index.dat에서 불러올 때 쓰인다.
        }
}
```

2. 각 명령에 따라 쓰이는 함수들

가. Insertion

1) in

key, value값을 입력받아 tree에 추가해주는 함수이다. 실제 입력을 받아주는 함수는 insertion인데 구현 중 root가 max_degree가 되었을 때를 insertion 내부에 구현해 주는 방법을 모르겠어서 따로 split을 한번더 해준 것이다.

2) insertion

key는 leaf node에서 추가된다. 그래서 leaf인 경우와 아닌 경우로 나누었다.
leaf가 아닐 때는 leaf node가 나올 때까지 key가 가야 할 next node를 찾아 주소를 update 시켜준다. 이때 max_degree가 되었다면 split을 진행한다. 이렇게 하면 leaf node부터 split을 버텀업으로 root를 제외하고 모두 해줄 수 있다.
leaf일때는 key 자리를 찾아가 일단 대입을 한다. 대입 이후에는 size를 증가시켜 준다. 이때 max_degree가 되어도 split을 하지 않아도 된다. 자식의 주소를 return 받아 parent가 확인하여 split을 해야 할지 결정한다.

3) split

split은 항상 자식의 크기가 max_degree일 때 실행된다.
일단 max_degree가 짝수일수도 홀수일수도 있는데 어떤 경우든 2로 나누고 내림하면 끊는 곳을 구할 수 있다 -> int pos = max_degree/2;
원래 자식은 child 그대로고 새로운 오른쪽에 생길 자식 new_node를 만들었다.
오른쪽 자식을 새로 만드는 이유는 child의 왼쪽 형제가 왼쪽 자식(child)을 가리키고 있을수 있기 때문이다(leaf 일때). new_node는 child와 level이 같아 leaf 여부가 같

다. pos를 기준으로 왼쪽 자식(child)과 오른쪽 자식(new_node)으로 나뉘었는데
child는 항상 크기가 pos로 일정하므로 미리 설정해주었다. (0 ~ pos-1 이므로)

이제 자식(child)이 leaf인지 아닌지에 따라 new_node가 달라지는데

1) 자식이 leaf인 경우
leaf인 경우 오른쪽 자식은 pos를 포함해야 한다. 따라서 new_node의 key들은
원래 자식의 (pos ~ max_degree-1)의 key값들과 같다. 오른쪽 자식의 size는
max_degree - 1 - pos + 1 = max_degree - pos 이다.
이제 연결만 해주면 되는데
부모와의 연결은 자식이 leaf든 아니든 상관없이 L,R 연결하면 되므로 밑에서 한번에 연결한다.
child -> sibling에서 L -> R -> sibling 이 되므로
new_node -> rightnode = child -> rightnode;
child -> rightnode = new_node; 이다.
원래 자식은 크기만 작아지고 왼쪽자식이 되었다. node는 따로 삭제를 하지 않았는데 항상 node의 크기만큼 무언가를 하기 때문에 상관없다.

2) 자식이 leaf가 아닌 경우
leaf가 아니면 오른쪽 자식은 pos를 포함하지 않아도 되므로
new_node의 key들은 원래 자식의 (pos+1 ~ max_degree-1) key값들이다.
new_node의 크기는 max_degree - pos - 1 이고
부모와의 연결은 다음과 같다.
new_node -> rightnode = child -> rightnode;
오른쪽 자식의 rightnode는 원래자식의 rightnode와 같고
child -> rightnode = child -> left_child_node[pos];
원래자식(이제 왼쪽 자식)의 rightnode는 올라갈 예정인 pos의 left_child_node와 같다.

이제 자식들 간의 연결은 끝났고 부모들의 형제와 비교하여 자리를 찾아가야한다.
먼저 root가 가득 찬 경우를 부모가 NULL인 경우로 표현했는데 이때는 아예 새로운 부모(결론적으로 root)를 만들고 연결만 하면 끝난다. 이때 새로 생기는 root는 무조건 non leaf이다.

마찬가지로 부모는 무조건 non leaf이다. 미리 잉여 key가 올라갈 자리를 미리 입력받았기 때문에 parent에서 공간 확보 작업해주고 key만 대입만 해주면 된다.
이때 잉여 key는 child의 pos 번째 key이다.

나. Deletion

1) del

삭제에서 가장 중요한 건 삭제 이후에도 그 자료구조가 잘 유지가 되어야 한다는 점이다. B+ tree가 유지해야 할 특성은 크게 살펴보면 2가지인데

1. nonleaf node에 있으면 leaf에도 있다. 즉, 어떤 key를 leaf node에서 삭제했을 때 그 key가 nonleaf node에도 있다면 그 node의 key도 삭제해 주어야 한다.
2. root node를 제외하고는 모든 node에서 key의 개수는 몇 개 이상이다. 이는 insertion에서 확인할 수 있는데 결론적으로 항상 모든 node는 (max_degree-1)/2 개 이상의 key를 가져야 한다.

2는 deletion 함수에서 만족하도록 했고, 1은 change 함수에서 만족하도록 구현했다. 구체적으로는 insertion과 비슷하게 일단 leafnode에서 key를 지우고 계속 트리를 유지할 수 있도록 아래에서 위로 balance 함수를 실행한다. 이제 트리의 조건을 모두 만족한다면 아직 nonleafnode에 지울 key가 남아있을 수 있으므로 그 key가 살아있다면 어느 leafnode에 있었을지 찾아가 그 leafnode의 첫 번째 key를 대신 대입해주면 된다. 이는 deletion 이후에도 key를 포함하는 위치는 무조건 1개이기 때문에 그 위치를 대표하는 leafnode key를 가져다 쓰면 문제가 없기 때문이다.

2) change

deletion 함수 실행 이후에 nonleaf node에서 삭제된 key가 남아있다면 그 key 대신 그 자리를 대표하는 leafnode key를 대입한다.

3) deletion

insertion 함수와 비슷하게 먼저 leafnode에서 key를 지우고 리턴해서, 부모가 자식을 봤을 때 자식이 모자르면 형제들로부터 빌려오거나, 합쳐서 트리의 특성을 유지하도록 한다. 이는 balance 함수가 진행한다.

4) balance

부모와 자식을 확인해서 자식의 key 개수가 적당하면 넘어가고, 부족하면 조정한다.
조정을 할때에는 형제 node를 확인하는데, 항상 왼쪽 형제부터 확인한다.
크게 2가지 경우가 있는데

1. 형제의 자식을 하나 가져올 수 있다.
그냥 가져오면 트리의 balancing이 끝난다.

2. 형제도 줄 수 없는 상태이다.
이때에는 형제와 merge를 진행한다. 항상 왼쪽 node를 기준으로 합친다. 자식이 leafnode 일때는 합친 후 nonleaf node의 key값을 하나 수정해주고, nonleafnode 이면 합칠 때 부모의 key값을 가져온 후 합쳐야 한다.
두 경우 모두 merge를 진행해도 key는 최대 max_degree - 1개 이다.
이 함수는 자식의 key 개수가 0개 일 때에도 실행이 되므로 deletion을 실행 후 root의 key 개수가 0개 일수도 있다. 이때에는 root의 left_child_node[0]가 root가 된다.

다. Single key search

1) serch (c++에서 search가 이미 있는 함수여서 a를 제거했다)

serch는 항상 leaf node에서 이루어진다. 따라서 leaf가 아니면 node안에 있는 모든 key값을 출력한 후 key값에 해당하는 다음 node로 다시 serch를 진행하고, key값에 해당하는 leaf node라면 leaf node안에서 key값을 찾으면 된다. 여기 없으면 없는 것이다.

라. Ranged search

1) findpointer

range_serch에 사용하기 위해 key값을 포함하는 leafnode의 주소를 리턴한다.

2) range_serch

각 leaf node들은 모두 연결되어 있으므로 leafnode들만 순회해도 key,value를 비교할 수 있다. 따라서 범위가 [x,y]이라면 x를 포함하는 leafnode 주소 L, y를 포함하는 leafnode 주소 R을 먼저 구해준다.
이때 L = R이면 하나의 leafnode만 검사하면 되므로 검사해주고
아니라면 L -> rightnode부터 순회를 시작해서 L == R일때만 범위 검사를 해주고
나머지 경우에는 무조건 L ~ R 범위 안이므로 검사를 안하고 바로 출력하면 된다.
순회는 L -> rightnode로 해준다.

마. index.dat 불러오기

1) 전역변수

```
static String[] treeinfo; //index.dat의 각 줄을 저장한다.
static node[] adress; //treeinfo로 node를 만들고 주소를 저장한다.
```

2) connect

각 node가 모두 할당된 후 저장된 연결상태를 가져와서 각 node를 연결시킨다.
원래는 할당하면서 동시에 연결을 진행하는 방법으로 구현했는데 data 100만 개에서 stack이 터져서 각각 구현했다.

3) Main 안에서

```
max_degree = Integer.parseInt(treeinfo[0]); //첫 줄은 트리 크기에 대한 정보이다.
int nodenum = treeinfo.length - 1; //index.dat의 줄 개수
if(nodenum > 0) { //줄이 2줄이상이어야 트리의 node에 대한 정보가 있는 것이다.
        for(int i=1; i<=nodenum; i++) {
                adress[i] = new node(treeinfo[i]); //생성자로 각 node주소 할당
        }
        for(int i=1; i<=nodenum; i++) {
                connect(treeinfo[i],i); //주소가 모두 할당된 후 연결
        }
        root = adress[1]; //항상 루트가 1번이다.
}
```

바. index.dat 내보내기

1) 전역변수

```
static String[] outputstring; //각 node의 정보를 한 줄에 저장한다.
static int cnt; //각 node에 번호를 임의로 부여하기 위해 전역변수를 설정했다.
```

2) dfsgetnum

b+tree를 root부터 dfs로 돌면서 각 node에 임의로 num을 부여해준다. node의 정보는 index.dat의 num+1번째 줄에 기록된다.

3) dfsgetstr

마찬가지로 root부터 dfs로 돌면서 할당된 num을 가지고 각 node의 정보를 String으로 만든후 outputstring에 넣어준다.

3. index.dat 파일의 구성

가. 첫째 줄

첫째 줄은 b+tree의 max degree를 나타낸다.

나. 나머지 줄

각줄 마다 node의 정보를 나타낸다. 순서대로 key개수, leaf여부, key들, value or left_child의 num들, rightnode의 num이 사이에 공백을 가지고 주어진다.

다. example

```
1 10
2 7 false 13822126 27837210 39065150 49252828 62577648 79775972 9096763
  2 26186 52646 73830 93012 118168 150421 171534
3 8 false 1931021 3270693 4519810 5772078 7204139 8751778 10988684
  12598530 3 3621 6158 8508 10867 13603 16492 20805 23840
4 8 false 169815 307039 576804 750480 902467 1143103 1386000 1624455 4
  329 579 1094 1420 1705 2143 2611 3059
5 5 false 27680 56654 90419 114429 135627 5 62 113 178 227 264
6 5 false 4634 8651 12527 16786 22070 6 15 24 33 40 51
7 7 false 580 1194 1704 2384 2915 3428 3938 7 8 9 10 11 12 13 14
8 6 true 14 43 71 87 488 517 77815000 32929472 7646975 90594381 9658125
  74074571 8
9 6 true 580 715 799 918 1019 1106 48748566 31352135 19496113 20711100
  10799014 94819059 9
10 5 true 1194 1308 1326 1415 1689 18185462 40847917 67269056 6265097
  49039968 10
11 7 true 1704 1754 1869 1923 1958 1967 2019 30451209 96482152 19074275
  75476516 81179615 28157411 76582888 11
12 6 true 2384 2454 2462 2483 2660 2743 41357073 95033533 70516557 29446
  66320432 42805558 12
13 8 true 2915 2927 2935 2943 3039 3060 3168 3208 49079325 35852175
  97750386 21773175 68269793 28403988 97578427 20297318 13
14 7 true 3428 3439 3505 3552 3750 3820 3861 8848493 24219464 83419674
  47130155 45447730 29263816 18334999 14
15 8 true 3938 3977 3990 4017 4106 4206 4299 4541 56851384 94242573
  53333508 25777185 46914930 99492540 66114356 34790061 16
16 7 false 5087 5686 5948 6217 7054 7665 8195 16 17 18 19 20 21 22 23
```

4. Instructions

https://youtu.be/tYmLZp6-ucQ 실행 과정을 직접 동영상으로 찍었습니다.

5. Discussion

가. Time Complexity & Space Complexity

일반적으로 알려진 B+tree의 탐색에서의 Time Complexity는 트리의 높이인 O(log n)이다. 직접 구현한 B+tree에서 데이터가 1048596개 일 때 max_degree(이하 max)가 4에서 트리의 높이는 13이 나왔고 max가 10일 때에는 높이가 7이 나왔다. 각 경우에서 트리 최악의 높이와 최선의 높이를 구해보자. (계산의 편의를 위해서 높이는 root부터 leaf까지 node 개수로 했다.)
1. max = 4일 때 root를 제외한 나머지 node는 key를 최소 1개 최대 3개를 가질 수 있다.

최악의 경우 : 각 key의 개수는 1 , 1 , 1 , 1 , ...
1048596 = 2^x - 1, x = 20

최선의 경우 : 각 key의 개수는 3 , 3 , 3 , 3 , 3, ...
1048596 = 4^x - 1, x = 10

2. max = 10일 때 root를 제외한 나머지 node는 key를 최소 4개 최대 9개를 가질 수 있다.

최악의 경우 : 각 key의 개수는 1 , 4,4 , 4,4,4,4,4,4,4,4,4, ,4,4,4...
1048596 = 2 * 5^(x-1) - 1, x = 9

최선의 경우 : 9 , 9 , 9 , 9 , ...
1048596 = 10^x - 1, x = 6

두 경우 모두 트리의 높이가 사이에 있어 잘 구성되었다. 즉, 검색 시간이 O(log n)이다.

공간복잡도는 2 * max만큼의 자료가 저장되어있어야 하고, index.dat를 가져올 때 최대 3 * node < 3 * max로 O(cn) = O(n)이다. (c는 상수이다.)

두 경우 모두 n은 key의 총개수를 의미한다.

나. 실생활에서 쓰이는 data

이번 과제에서는 data의 종류가 value 1가지였다. 하지만 실생활에서는 data의 종류가 더 많다. 당장 이번에 같이 데이터베이스시스템 수업을 듣는 분들의 이름, 학과, 학번, 역할(학생, 조교, 교수) 등등의 data를 잘 저장해야 할 것이다. 이번 과제를 넘어서 위의 data를 추가로 저장하려면 어떻게 수정해야 할까?

1. data class에서 추가로 String 변수들을 여러 개 설정하고 각 생성자도 수정해주어야 한다.
2. index.dat로 내보낼 때 변수의 개수를 max_degree 옆에 저장해주고 각 줄을 불러올 때마다 leaf일 때에는 변수 개수만큼 끊어서 읽으면 된다.

세부적으로 몇 가지 추가해주어야 할 부분이 있겠지만 크게 위의 2가지를 해야 한다.

6. Reference

[1] https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html