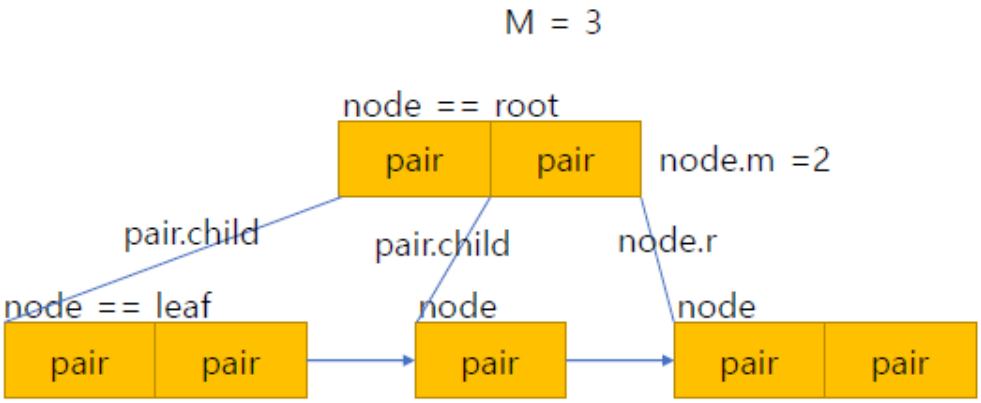
B+tree implementation assignment

컴퓨터소프트웨어학부 2018008177 김찬위

1. 알고리즘 요약

- **클래스**



- **데이터 저장**

1. 차수를 가장 먼저 저장한다.
2. DFS 을 이용하여 root 부터 시작해 모든 node 를 저장한다.
3. node 를 발견할 때 마다 count++로 index 를 부여한다.
4. 부모 node 의 index 와 현재 node 의 index 를 먼저 저장한 뒤, 현재 node 에 담긴 pair 의 key, value 를 모두 저장한다.
5. child 가 없는 leaf node 일 경우에는 index leaf 에 따로 저장한다.

Ex)

```
3
# 0 1 / 3 102 /
# 1 2 / 2 101 /
# 2 3 / 1 100 /
# 2 4 / 2 101 /
# 1 5 / 4 103 /
# 5 6 / 3 102 /
# 5 7 / 4 103 / 5 104 /
@ 3 4 6 7
```

- 데이터 읽어오기

1. 첫 줄은 M 으로 저장한다,
2. $로 시작하는 줄 첫번째 pair 는 각각 부모 인덱스와 내 인덱스로 저장한다.
3. tree 의 내 인덱스에 나머지 pair 를 차례대로 저장한다.
4. @로 시작하는 줄은 leaf 인 node 들의 인덱스기 때문에 오른쪽 leaf 로 연결한다.

- **삽입**

1. key 를 삽입할 leaf node 를 찾아가서 삽입한다.
2. leaf node 의 m 을 M 과 비교한다.
 2-1. m 이 M 보다 작다면 그대로 끝낸다.
 2-2. 그렇지 않다면 두 개의 node 로 split 하여 M/2 번째를 부모 node 에 올린다. (이때 split 한 노드에도 M/2 pair 가 남아 있어야한다.)
3. split key 를 올린 부모 node 의 m 을 M 과 비교한다.
 3-1. 부모 node 의 m 이 M 보다 작다면 그대로 끝낸다.
 3-2. 그렇지 않다면 두 개의 node 로 split 하여 M/2 번째를 부모 node 에 올린다. (이때는 split 한 노드에 M/2 pair 가 남아 있으면 안된다.)
4. 3-1 이 발생할 때까지 3 번 과정을 반복한다.

+ split 해야 하는 노드가 root 라면 새로 부모 node 를 만들고 M/2 번째 pair 를 올린 다음 부모 node 를 root 로 정한다.

- **삭제**

1. key 가 존재하는 leaf node 를 찾아가서 삭제한다.
2. index 에도 삭제된 key 가 존재할 경우 leaf node 에서 가장 작은 key 로 그 값을 변경한다.
3. leaf node 의 m 을 (M-1)/2 와 비교한다.
 3-1. m 이 (M-1)/2 이상이라면 그대로 끝낸다.
 3-2. 그렇지 않다면 형제 node 의 m 을 확인한다.
  3-2-1. m 이 (M-1)/2 이상이라면 pair 하나를 가져온다.
  3-2-2. 그렇지 않다면 key 를 삭제한 leaf node 의 나머지 pair 를 형제 node 중 하나에 넣는다.

4. 부모 node 의 m 을 (M-1)/2 와 비교한다.

4-1. m 이 (M-1)/2 이상이라면 그대로 끝낸다.

4-2. 그렇지 않다면 부모 node 의 pair 들을 형제 node 중 하나에 전부 넣는다.

5. 4-1 이 발생할 때까지 4 과정을 반복한다.

+ 부모 node 가 root 일 경우 m 이 1 이상이면 끝낸다.

- **단일 키 검색**

1. root node 에서부터 시작한다.
2. find_key 의 삽입할 자리를 찾는 방식으로 leaf node 까지 찾아간다.
3. 지나온 경로를 모두 기록한다.
4. 찾은 leaf node 의 pair 중에서 find_key 를 찾는다.

4-1. 찾지 못한 다면 이 tree 에 find_key 는 없다.

- **범위 검색**

1. root node 에서부터 시작한다.
2. start_key 의 삽입할 자리를 찾는 방식으로 leaf node 까지 찾아간다.
3. 도착한 leaf 의 key 값이 end_key 보다 작거나 같을 경우 저장한다.
4. leaf 의 모든 pair 를 확인했다면 오른쪽 leaf 로 이동한다.
5. 3~4 과정을 반복한다.
6. leaf 의 key 가 end_key 보다 크다면 반복을 멈춘다.
7. 저장한 key 와 value 를 모두 출력한다.

2. 코드 상세 설명

- **클래스**
  - class node
    - m: pair 의 개수
    - p: pair 클래스를 ArrayList 로 저장
    - r: 가장 오른쪽 자식 / 오른쪽 형제 노드
    - parent: 부모 노드

- class pair
  - key: key
  - val: value
  - child: 자식 노드

- **전역 변수**
  - M: 차수
  - count: 노드 번호 매기기, 트리 구조 저장할 때 사용한다.
  - root: root
  - tree: node 클래스를 ArrayList 로 저장, 파일 읽을 때 사용한다.
  - leaf: leaf node 의 번호만 따로 저장, 파일 읽을 때 사용한다.

- **데이터 저장**
  - save(FileWriter w, node now, int parent)

    1. 함수가 호출 될 때 마다 count++ 한다

       1-1. now 의 인덱스는 count 이다.

    2. 인자로 들어온 부모 인덱스와 now 의 인덱스를 저장한다.
    3. now 의 pair 들을 key value 형태로 모두 저장한다.
    4. now 의 pair 의 child 에 접근한다.

       4-1. pair 의 child 가 존재한다면 save 함수를 호출한다.
       - now = child
       - parent = now 의 인덱스 번호

    5. now 가 non-leaf node 라면 save 함수를 호출한다.
       - now = 가장 오른쪽 자식
       - parent = now 의 인덱스 번호

    6. now 가 leaf node 라면 전역변수 leaf 에 now 의 인덱스를 저장한다.

  - save_layout(String index_flie)

    차수를 저장하고 save 함수를 호출한 뒤 leaf node 를 저장한다.

- **데이터 읽어오기**
  - read_data(String filename, String act)

    1. 데이터 파일을 한 줄 씩 읽는다.
    2. ","를 기준으로 split 한다.
    3. act 에 따라 insert or delete 함수를 호출한다.
      3-1. "-i" 이면 insert 를 호출한다.
      3-2. 그 외의 경우 delete 를 호출한다.
    4. 파일을 다 읽을 때 까지 1~3 을 반복한다.

  - read_tree(String fileName)

    1. 첫번째 줄은 split 하지 않고 바로 M 에 저장한다.
    2. 첫번째 문자를 확인한다.
      2-1. "$" 이면 tree 에 저장한다.
        2-1-1. $ 바로 뒤의 두 숫자는 각각 부모의 index 와 내 index 이다.
        2-2-2. / key value / 형식으로 저장된 데이터를 pair 에 저장한다.
        2-2-3. 부모 index 와 내 index 를 통해 child 와 parent 를 연결한다.

    (DFS 방식으로 저장하면서 index 를 정했기 때문에 순차적으로 부모 node pair 의 child 가 된다.)

        2-2-4. child 가 꽉 찼다면 맨 오른쪽 자식으로 연결한다.
      2-2. 그렇지 않으면 leaf 에 저장한다.
        2-2-1. leaf node 를 tree 에서 접근해 차례대로 오른쪽 node 로 연결한다.

- **데이터 파일 생성**
  - create(String index_file)

    : FileWriter 를 이용하여 파일을 생성하고 M(차수)을 저장한다.

- **삽입**
  - leaf_find(int find_key)

    : find_key 가 어느 node 삽입되어야 하는지 알기 위함이다.

    - node now = root;

1. now 의 자식이 없다면 now 를 return 한다.

2. now 의 pair 들의 key 를 find_key 와 비교한다.

2-1. find_key 가 pair 의 키보다 작을 경우 now 는 그 pair 자식이 된다.

3. now 의 모든 pair 의 key 보다 find_key 가 크다면 now 는 now 의 가장 오른쪽 자식이 된다.

4. leaf node 에 도달할 때까지 1~3 과정을 반복한다.

- index_find(node now, int find_key)

: now 에서 찾고자 하는 키가 몇 번째 pair 에 존재하는지 알기 위함이다.

➢ find_key 가 있는 node 를 인자로 받아야한다.

1. find_key 가 pair 의 key 보다 작다면 그때의 인덱스를 return 한다.

2. 그렇지 않다면 now.m 을 return 한다.

- leaf_split(node now, node right)

: leaf node 에서 split 하는 함수이다.

1. split 해야하는 now 의 M/2 번째 pair 부터 마지막 pair 까지 right 에 새로 넣는다.

2. right 에게 넘긴 pair 들을 now 는 지운다.

- parent_split(node now)

: non-leaf node 에서 split 하는 함수이다.

➢ node right = new node( );

➢ size == now.m

1. now 의 size/2 번째 pair 는 따로 남겨둔다.

2. split 해야 하는 now 의 size/2+1 번째 pair 부터 마지막 pair 까지 right 에 새로 넣는다.

3. now 는 size/2 번째 pair 부터 마지막 pair 까지 지운다.

4. now 의 가장 오른쪽 자식을 right 의 오른쪽 자식으로 넘긴다.

5. now 의 가장 오른쪽 자식은 따로 남겨둔 size/2 번째 pair 의 자식이다.

6. now 와 right 모두 가장 오른쪽 자식의 부모는 자신이다.

7. now 가 root node 인지 확인한다.

7-1. now == root node

7-1-1. parent node 를 새로 만들고 따로 남겨둔 pair 를 넣으며 자식 node 는 now 로 설정한다.

7-1-2. parent 의 오른쪽 자식은 right 이다.

7-1-3. root 를 새로 만든 parent 로 설정한다.

7-1-4. now 와 right 모두 부모는 parent 이다.

7-2. now != root node

7-2-1. now 가 맨 오른쪽 node 일 경우

7-2-1-1. now 의 부모에게 남겨둔 pair 를 추가한다.

7-2-1-2. now 의 부모의 가장 오른쪽 자식은 right 이다.

7-2-2. 그렇지 않을 경우

7-2-2-1. now 를 자식으로 갖던 pair 자리에 남겨둔 pair 를 삽입한다.

7-2-2-2. 밀려난 pair 의 자식은 right 이다.

7-2-3. right 의 부모는 now 의 부모이다.

7-2-4. now 의 부모가 split 이 필요하다면 parent_split 을 호출한다.

■ insert(int key, int val)

: leaf node 에서 insert 하는 함수이다.

➢ node now = leaf_find(key);

1. 인자로 받은 key 와 val 을 now 의 알맞은 자리에 삽입한다.

2. now.m 이 차수보다 작다면 그대로 끝낸다.

3. leaf_split 함수를 호출한다.

4. now 가 root node 인지 확인한다.

4-1. now == root node

4-1-1. parent node 를 새로 만들고 right 의 첫번째 pair 를 넣으며 자식 node 는 now 로 설정한다.

4-1-2. parent 의 오른쪽 자식은 right 이다.

4-1-3. root 를 새로 만든 parent 로 설정한다.

4-1-4. now 와 right 모두 부모는 parent 이다.

4-1-5. right 의 오른쪽 node 는 now 의 오른쪽 node 이다.

4-1-6. now 의 오른쪽 node 는 right 이다.

4-2. now != root node

4-2-1. now 가 맨 오른쪽 node 일 경우

4-2-1-1. now 의 부모에게 right 의 첫번째 pair 를 추가한다.

4-2-1-2. now 의 부모의 가장 오른쪽 자식은 right 이다.

4-2-2. 그렇지 않을 경우

4-2-2-1. now 를 자식으로 갖던 pair 자리에 right 의 첫번째 pair 를 삽입한다.

4-2-2-2. 밀려난 pair 의 자식은 right 이다.

4-2-3. right 의 오른쪽 node 는 now 의 오른쪽 node 이다.

4-2-4. now 의 오른쪽 node 는 right 이다.

4-2-5. right 의 부모는 now 의 부모이다.

4-2-6. now 의 부모가 split 이 필요하다면 parent_split 을 호출한다.

- **삭제**
  - leaf_find_left(int find_key)

    : 나와 같은 level 에서 바로 왼쪽 노드를 찾는 함수이다.

    ➢ node now = root;

    1. now 가 leaf node 라면 now 를 return 한다.
    2. now 의 pair 들의 key 를 find_key 와 비교한다.

       2-1. find_key 가 pair 의 키보다 작거나 같을 경우 now 는 그 pair 자식이 된다.

    3. now 의 모든 pair 의 key 보다 find_key 가 크다면 now 는 now 의 가장 오른쪽 자식이 된다.
    4. leaf node 에 도달할 때까지 1~3 과정을 반복한다.

  - swap_key(int delete_key, int swap_key)

    : delete 할 key 가 index 에도 있을 경우 key 를 지우기(바꿔주기) 위한 함수이다.

    1. 찾는 방식은 leaf_find 와 동일하다.
    2. 단, delete_key 를 찾았다면 곧바로 swap_key 로 덮어쓴 뒤 끝낸다.

■ index_find_parent(node now)

: now 가 부모의 몇 번째 pair 자식인지 알려주는 함수이다.

1. now 가 root 가 아닐 경우

  1-1. now 부모의 pair 들의 자식이 now 와 같을 때의 index 를 return 한다.

  1-2. 맨 오른쪽 자식이라면 부모의 m 을 return 한다.

2. return 되지 않았다면 -1 을 return 한다.

■ find_sibling(node now, String act)

: now 의 왼쪽 또는 오른쪽 형제 노드를 알려주는 함수이다.

1. now 가 root 면 null 을 return 한다.

2. now 가 부모 node 에서 몇 번째 pair 의 자식인지 알기 위해 index_find_parent 함수를 호출한다.

3. 맨 왼쪽 자식일 경우

  3-1. act 가 left 라면 null 을 return 한다.

  3-2. act 가 right 일 경우

    3-2-1. pair 의 자식이 아니라면 부모 node 의 맨 오른쪽 자식을 return 한다.

    3-2-2. pair 의 자식이라면 바로 오른쪽 pair 의 child 를 return 한다.

4. 맨 왼쪽 자식이 아닌 경우

  4-1. act 가 left 라면 pair 의 바로 왼쪽 pair 의 child 를 return 한다.

  4-2. act 가 right 일 경우

    4-2-1. 내가 맨 오른쪽 자식일 경우 null 을 return 한다.

    4-2-2. pair 의 자식이 아니라면 부모 node 의 맨 오른쪽 자식을 return 한다.

    4-2-3. 모두 그렇지 않다면 바로 오른쪽 pair 의 child 를 return 한다.

■ merge_parent(node now)

: 부모 node 의 pair 가 충분하지 않을 때 형제 node 와 합치기 위한 함수이다.

1. find_sibling 함수를 호출하여 left, right 를 찾는다.

2. left 가 존재할 경우

  2-1. left 를 자식으로 갖는 key 를 left 에 추가한다.

  2-2. 그 key 를 갖는 부모의 pair 를 지운다.

2-3. left 에 now 의 pair 를 모두 추가한다.

2-4. left 의 맨 오른쪽 자식은 now 의 맨 오른쪽 자식이다.

2-5. left 의 맨 오른쪽 자식의 부모는 left 이다.

2-6. now 의 부모와 자식을 알맞게 연결한다.

2-7. left 가 M 보다 크거나 같다면 parent_split 을 호출한다.

2-8. now 의 부모가 root 이고 비어있다면 root 는 left 이다.

2-9. now 의 부모가 root 가 아닌데 pair 가 충분하지 않다면 merge_parent 를 호출한다.

3. right 가 존재할 경우

3-1. now 를 자식으로 갖는 key 를 now 에 추가한다.

3-2. 그 key 를 갖는 부모의 pair 를 지운다.

3-3. now 에 right 의 pair 를 모두 추가한다.

3-4. now 의 맨 오른쪽 자식은 right 의 맨 오른쪽 자식이다.

3-5. now 의 맨 오른쪽 자식의 부모는 right 이다.

3-6. right 의 부모와 자식을 알맞게 연결한다.

3-7. now 가 M 보다 크거나 같다면 parent_split 을 호출한다.

2-8. right 의 부모가 root 이고 비어있다면 root 는 now 이다.

2-9. right 의 부모가 root 가 아닌데 pair 가 충분하지 않다면 merge_parent 를 호출한다.

- delete(int key)
  - node now = leaf_find(key);
  - node left_leaf = leaf_find_left(now.p.get(0).key);
  - int toChange = now.p.get(0).key;

1. leaf_find 함수를 호출해 지워야 하는 key 가 있는 leaf node 에 접근한다

2. 지울 키가 없다면 끝낸다.

3. now 의 왼쪽 형제 node 와 부모에 올라가 있는 key 값을 따로 저장한다.

4. delete 를 한 후 now.m 이 M-1/2 보다 크거나 같다면 toChange 와 부모 node 로 올려야 하는 key 를 인자로 넣어 swap_key 함수를 호출한 후 끝낸다.

5. now 의 왼쪽 형제와 오른쪽 형제를 find_sibling 함수를 호출해 저장한다.

6. left 가 존재하고 빌려올 수 있다면

6-1. left 의 가장 오른쪽 pair 를 now 의 첫번째 pair 로 추가한다.

6-2. now 에게 준 pair 를 left 에서 삭제한다.

6-3. toChange 와 부모 node 로 올려야 하는 now 의 key 를 인자로 넣어 swap_key 함수를 호출한다.

7. rigth 가 존재하고 빌려올 수 있다면

7-1. right 의 첫번째 pair 를 now 의 pair 로 추가한다.

7-2. now 에게 준 pair 를 right 에서 삭제한다.

7-3. now 가 첫번째 자식이었다면 toChange 와 부모 node 로 올려야 하는 key 를 인자로 넣어 swap_key 함수를 호출한다.

7-4. 부모에게 올려진 right 의 키와 바뀐 right 의 키를 인자로 넣어 swap_key 함수를 호출한다.

8. 형제 node 에게서 빌려오는 것이 불가능 하다면

8-1. now 의 부모가 존재할 경우

8-1-1. left 가 존재할 경우

8-1-1-1. left 에 now 의 모든 pair 를 추가한다.

8-1-1-2. left 의 맨 오른쪽 자식은 now 의 맨 오른쪽 자식이다.

8-1-1-3. 부모 node 에서 now 의 자리를 찾아 left 를 위치시킨다.

8-1-1-4. now 를 child 로 갖는 pair 의 바로 왼쪽 pair 를 지운다.

8-1-1-5. now 의 부모가 root 이고 root 가 비었다면 root 는 left 이다.

8-1-2. 그렇지 않을 경우

8-1-2-1. right 의 0 번째에 now 의 pair 를 뒤에서부터 차례대로 넣는다.

8-1-2-2. now 가 맨 왼쪽 자식이라면 right 에서 부모에게 올려진 key 자리에 바뀐 right 의 키를 인자로 넣어 swap_key 함수를 호출한다.

8-1-2-3. 왼쪽이 존재한다면 왼쪽 leaf 의 오른쪽 leaf 는 right 이다.

8-1-2-4. now 를 자식으로 갖는 child 를 지운다.

8-1-2-5. now 의 부모가 root 이고 root 가 비었다면 root 는 right 이다.

9. now 의 부모의 pair 가 충분하다면 그대로 끝낸다.

10. 그렇지 않다면 merge_parent 를 호출한다.

- **단일 키 검색**
  - Single_key_find(int find_key)
    - List<node> path = new ArrayList<>();

    1. now 가 leaf node 라면 path 에 now 를 저장하고 반복을 끝낸다.
    2. now 의 pair 들의 key 를 find_key 와 비교한다.
      2-1. find_key 가 pair 의 key 보다 작거나 같을 경우
        2-1-1. now 를 path 에 저장한다.
        2-1-2. now 는 그 pair 의 자식이다.
        2-1-3. 2 번 과정을 끝낸다.
    3. now 의 모든 pair 의 key 보다 find_key 가 클 경우
      3-1. now 를 path 에 저장한다.
      3-2. now 는 now 의 가장 오른쪽 자식이 된다.
    4. 1~3 과정을 반복한다.
    5. 반복이 끝나면 now 에서 find_key 를 찾는다.
    6. find_key 가 없는 경우 "NOT FOUND", 있는 경우 과제 설명서에 맞춰 출력한다.

- **범위 검색**
  - ranged_search(int start_key, int end_key)
    - node now = leaf_find(start_key);
    - List<node> path = new ArrayList<>();

    1. now == null 이면 반복을 끝낸다.
    2. now 의 pair 들의 key 와 start_key 를 비교한다.
      2-1. pair 의 key 가 end_key 보다 크다면 2 과정을 끝낸다.
      2-2. pair 의 key 가 start_key 보다 크거나 같으면 path 에 저장한다.
    3. 2-1 로 2 번을 끝냈다면 반복을 끝낸다.
    4. now 는 now 의 오른쪽 node 이다.
    5. 1~4 과정을 반복한다.
    6. 과제 설명서에 맞춰 출력한다.

3. 컴파일 방법

terminal(cmd, bash 등)에서 파일이 있는 경로까지 이동 후 아래 명령어 실행한다.

```
javac bptree.java
```