WIKI

학과: 컴퓨터소프트웨어학부

학번: 2022006135

이름: 문준영

Design

B+ tree의 설계에 앞서 먼저 구조를 살펴보았다.

```
//Page는 총 4096 bytes => 31개의 record가 들어가거나 inter_record 248개가 들어갈 수 있음
typedef struct Page{
    //8 bytes
    off_t parent_page_offset;
    //4 bytes
    int is_leaf;
    //4 bytes
    int num_of_keys;
    //104 bytes
    char reserved[104];
    //8 bytes
    off_t next_offset;
    //3968 bytes
    //leaf면 records를 저장하고 있으므로 records를 사용함
    //non-leaf (internal node) 에 대해서는 I_R을 사용
    union{
        I_R b_f[248];
        record records[31];
    };
}page;
```

현재 우리가 만들려고 하는 B+ tree는 DiskBase B+ tree이다. 때문에 모든 node가 4096 bytes의 page로 이루어져 있다. 먼저 off_t라는 (long int) 변수를 이용해 pointer를 표현한다. In-memory의 경우 pointer로 다른 노드를 접근할 것이지만 DiskBase이기 때문에 page 단위로 load를 해야하여 offset 변수가 pointer 대신 주어진다.

Page에서 제일 중요한 점은 두 가지 있다. 먼저 union으로 묶여있는 b_f와 records이다. 이는 internal node에 대해서는 b_f로 key에 접근해야 하는 것이고 (internal node에서는 key에 해당하는 value를 저장하지 않기 때문), leaf node에 대해서는 records로 접근을 해야하는 것이다. 더하여 records와 b_f의 개수가 서로 다른데 이는 internal과 leaf에서 서

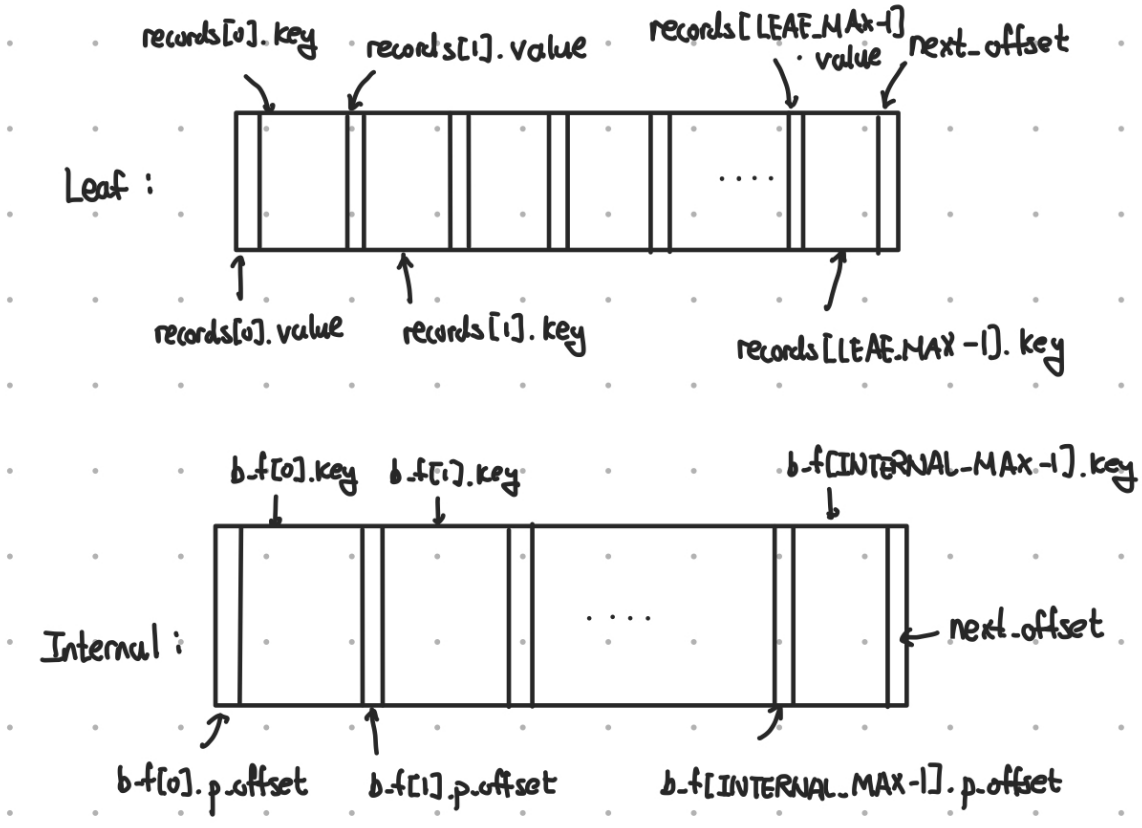
로 order가 다르다는 것을 의미한다. 실제로 LEAF_MAX와 INTERNAL_MAX 값이 서로 다른 것을 확인 할 수 있다. Bpt.h에서 말하는

```
*/
#define LEAF_MAX 31
#define INTERNAL_MAX 248
```

LEAF_MAX는 leaf node의 "key" 최대 개수이며 때문에 leaf node의 order는 32이다. (31 + 1 (pointer)) 또한, INTERNAL_MAX 역시 internal node의 "key" 최대 개수이기 때문에 internal node의 order는 최대 249이다. (248 + 1 (pointer))

마지막으로 bpt.h에서 알아둬야 할 중요한 것은 page struct의 next_offset이다. Next_offset 변수는 internal node와 leaf_node에서 사용 방법이 서로 다르다. Leaf_node에서는 next_offset은 다음(오른쪽) child를 가리키게 하는 pointer로써 활용된다. 반면, internal node에서 next_offset은 자식을 가리키는 하나의 추가적인 pointer로서 활용된다.

때문에 leaf node와 internal node의 구조는 다음 그림과 같다.



Find

Find 함수의 경우 구현에서 딱히 생각할 것이 많이 없다. 주어진 key에 대해서 해당 key를 포함하는 leaf node까지 찾아가서 해당 leaf node에서 key를 찾도록 할 것이다.

Insert

Insert의 경우 leaf와 internal의 order가 다른 것을 고려하여 구현 할 것이다. 먼저 주어진 key, value에 대해 삽입이 가능하다면 삽입을 할 것이다. 하지만 만약 해당 node가 꽉 차있을 경우에는 먼저 key-rotation을 수행할 것이다. Key-rotation은 오른쪽 형제에 여유가 있어야 가능한 것이다. 만약 오른쪽 형제가 여유롭지 않다면 split을 통해 새로운 node를 만들어 낼 것이다. Split을 하는 기준이 leaf와 internal이 다르기에 새로운 함수를 만들어 재귀적으로 호출하여 split을 수행할 것이다.

Delete

Delete의 경우 역시 leaf와 internal의 order가 다른 것을 고려하여 구현 할 것이다. 먼저, 주어진 key에 대한 것을 삭제한다. 그리고 만약 최소 개수가 충족되지 않으면 먼저 merge를 수행한다. Merge를 수행하기 위해서는 sibling을 찾아야 하는데 **우선 왼쪽 sibling을 우선적으로 보고 왼쪽 sibling이 merge 할 수 있는 condition을 만족하지 못하면 오른쪽 형제를 선택해 merge 하도록 구현할 것이다.**

만약 양쪽 형제 모두가 merge 가능한 condition이 아니라면 redistribution을 수행한다. 여기서 중요한 점은 merge condition을 양쪽 형제가 만족시키지 못하고 redistribution을 수행하는 것이기에 **양쪽 형제 모두가 redistribution condition은 만족할 것이라는 것이다.** 때문에, 삭제가 일어나는 node가 같은 부모에서 제일 왼쪽 자식이 아닌 경우 (왼쪽 sibling이 없는 경우가 아닌 경우)에는 왼쪽 sibling으로부터 redistribution이 일어나도록 설계하였다. 만약 삭제가 일어나는 node가 부모에서 제일 왼쪽 자식이라면 오른쪽 sibling으로부터 redistribution이 일어나도록 설계할 것이다.

Implementation

Insert

Insert의 경우 key-rotation과 split으로 이루어져 있다. 먼저 구체적인 단계로 살펴보기에 앞서 db_insert 함수에서는 leaf_node에 대한 처리만 하게 하였고 insert_in_parent함수와 update_parent함수에서 internal node들에 대한 처리를 하게 해주었다. Insert의 각 단계를 구체적으로 살펴보자.

```
int db_insert(int64_t key, char * value) {

    //현재 tree가 비어있을 경우
    if(hp->rpo == 0){
        //새로운 root생성
        page * new_root = (page*)malloc(sizeof(page));
        //새로운 페이지 할당받기
        off_t new_root_offset = new_page();
        //header page에서 루트를 가리키게 하기
        hp->rpo = new_root_offset;
        //root는 parent가 없음
        new_root->parent_page_offset = 0;
        //root의 sibling은 없으므로 next_offset을 0으로 처리
        new_root->next_offset = 0;
        //새로운 root는 root이자 leaf node
        new_root->is_leaf = 1;
        //하나의 새로운 키가 삽입되므로 num_of_keys를 1로 설정
        new_root->num_of_keys = 1;
        //next offset 값을 0으로 초기화 할까 했는데 0은 header page를 가리키므로 좀 그럼... 그래서 그냥 안 설정해둠
        //루트노드에 key값 value값 넣기 전달
        new_root->records[0].key = key;
        strcpy(new_root->records[0].value, value);
        //새로운 루트를 해당 페이지에 써주기
        pwrite(fd, new_root, sizeof(page), new_root_offset);
        //header page에도 업데이트가 일어났으니 disk write해주기
        pwrite(fd, hp, sizeof(H_P), 0);
        free(new_root);
        //이 과정에서 hp를 다시 load해줄 필요는 없는데
        //위의 함수에서 모두 이렇게 했길래 convention을 따름
        free(hp);
        hp = load_header(0);
        return 0;
    }
```

가장 기본적인 경우는 tree가 비어있는 경우이다. Tree가 비어있는 경우 root를 새롭게 생성하도록 하였다. 이 때 root는 root이자 leaf가 된다.

```
page * curr_page = load_page(hp->rpo);
int i;
off_t p_offset = hp->rpo;

while (!curr_page->is_leaf) {
    //internal node를 돌면서 key값 비교
    for (i = 0; i < curr_page->num_of_keys; i++) {
        if (key <= curr_page->b_f[i].key) {
            break;
        }
    }
    //끝에 노드까지 도달한 경우 loop를 빠져나오면서 i가 num_of_keys가 됨
    //key는 0~num_of_keys-1까지 있겠지만 p_offset의 개수는 0~num_of_keys만큼 있을 것임
    if(i == curr_page->num_of_keys){
        //key가 꽉 차있을 경우 결국 맨 오른쪽 pointer 이용해야함
        if(curr_page->num_of_keys == INTERNAL_MAX){
            //맨 오른쪽 pointer는 next_offset에 저장되기에 p_offset을 맨 오른쪽 pointer로 설
            p_offset = curr_page->next_offset;
        }
        //만약 key가 꽉 안 차있을 경우 num_of_keys의 pointer를 이용하면 됨
        else{
            p_offset = curr_page->b_f[i].p_offset;
        }
    }
    //이 때는 key가 딱 curr_page의 key와 동일해서 break 문을 밟고 온 것
    //이 때는 Loop에서 1이 더 증가하지 않은 상태로 나옴
    //때문에 p_offset을 그냥 i+1로 해당 key의 오른쪽 offset으로 접근하게 함
    else if(key == curr_page->b_f[i].key){
        //여기서도 i+1이 num_of_keys와 같으면 맨 오른쪽 pointer로 이동시켜줘야 함
        if(i+1 == INTERNAL_MAX){
            p_offset = curr_page->next_offset;
        }
        //꽉 차있지 않은 경우에는 그냥 현재 key의 오른쪽 pointer 이용하면 됨
        else{
            p_offset = curr_page->b_f[i+1].p_offset;
        }
    }
    //key가 search key보다 작을 때에는 그냥 왼쪽 pointer 따라가게 함
    else{
        p_offset = curr_page->b_f[i].p_offset;
    }
    // 현재 페이지를 해제하고 다음 페이지를 로드
    free(curr_page);
    curr_page = load_page(p_offset);
}
```

만약 tree가 비어있지 않은 경우에는 주어진 key가 들어갈 leaf node를 찾는다.

Leaf node를 찾은 후에는 중복된 key값이 해당 node에 있는 지 검사한다.

```
for(int i = 0; i < curr_page->num_of_keys; i++){
    if(curr_page->records[i].key == key){
        printf("Key : %ld already exists\n", key);
        return -1;
    }
}
```

중복된 key가 있으면 insert를 실패하도록 한다.

```
//이제 curr_page는 leaf node에 도달함
//LEAF_MAX보다 작아서 아직 leaf node에 여유공간이 남아 있다면
if(curr_page->num_of_keys < LEAF_MAX){
    free(curr_page);
    insert_in_leaf(p_offset, key, value);
    return 0;
}
```

Leaf에 도달하고 만약 해당 leaf node에 여유공간이 있다면 Leaf node에 주어진 key와 value를 삽입한다. (insert in leaf는 주어진 key가 들어갈 위치를 찾아 삽입하는 함수이다.)

Key_Rotation

만약 Leaf node에 여유공간이 없다면 key-rotation이 가능한지 먼저 확인한다.

```
//만약 leaf node가 꽉 차 있다면
//Key-Rotation
//만약 오른쪽 sibiling node가 있다면
if(curr_page->next_offset != 0){
    //sibling leaf node를 불러오기
    page * next_page = load_page(curr_page->next_offset);
    off_t next_page_offset = curr_page->next_offset;
    //sibling에 여유공간이 있다면
```

먼저 현재 page (leaf node)의 next_offset이 0이 아니라면 (next_offset의 default value를

0으로 설정하도록 하였다) 오른쪽 sibling이 존재한다는 것이다,

```
        //sibling에 여유공간이 있다면
        if(next_page->num_of_keys < LEAF_MAX){
            //memory에 임시 배열을 할당
            record * temp = (record*)malloc((LEAF_MAX + 1) * sizeof(record));
```

오른쪽 sibling에 여유공간이 있으면 key-rotation을 수행한다.

```
    record * temp = (record*)malloc((LEAF_MAX + 1) * sizeof(record));
    int p = 0;
    //key가 curr_page record의 key보다 클때는 curr_page의 record 정보 temp에 복사
    while(p < curr_page->num_of_keys && key > curr_page->records[p].key){
        temp[p].key = curr_page->records[p].key;
        strcpy(temp[p].value, curr_page->records[p].value);
        p++;
    }
    //key가 들어갈 위치를 찾았으니 key, value값 temp에 넣어주기
    temp[p].key = key;
    strcpy(temp[p].value, value);
    //curr_page의 나머지 애들을 temp에 넣어줌
    //temp에는 key값이 들어갔으므로 p+1부터 넣어줌
    for(; p < curr_page->num_of_keys; p++){
        temp[p+1].key = curr_page->records[p].key;
        strcpy(temp[p+1].value, curr_page->records[p].value);
    }
```

Temp라는 임의의 공간을 memory에서 할당받아 현재 page (insert가 일어나고 있는 page)의 모든 값과 새롭게 insert할 key, value값을 모두 temp에 넣어준다. (오름차순으로)

```
//이제 temp에 모든 값이 들어감
//sibiling에 여유공간이 있으므로 temp에서 제일 큰 값을 sibling에 넣어줌
//당연히 제일 처음에 들어갈 것임
//먼저 한 칸씩 밀어주기
for(int j = next_page->num_of_keys; j > 0; j--){
    next_page->records[j].key = next_page->records[j-1].key;
    strcpy(next_page->records[j].value, next_page->records[j-1].value);
}
//sibling 제일 처음에 insert 시 가장 큰 값 곱하기
next_page->records[0].key = temp[LEAF_MAX].key;
strcpy(next_page->records[0].value, temp[LEAF_MAX].value);
next_page->num_of_keys++;

//temp의 마지막 값을 제외한 나머지를 다시 curr_page에 돌려놓기
for(int i = 0; i < LEAF_MAX; i++){
    curr_page->records[i].key = temp[i].key;
    strcpy(curr_page->records[i].value, temp[i].value);
}
//next_page에 대한 작성이 끝났으므로 더이상 이를 추가적으로 활용할 일 없으니 output함
pwrite(fd, next_page, sizeof(page), next_page_offset);
//curr_page에도 작성이 끝났으므로 더이상 이를 차가적으로 활용할 일 없으니 output 시킴
pwrite(fd, curr_page, sizeof(page), p_offset);
```

Temp에 모든 값이 들어가면 오른쪽 sibling에 temp의 제일 마지막 key, value 값을 넣어준다. (가장 큰 값)

```
int64_t new_key = next_page->records[0].key;
off_t left_child_offset = p_offset;
//부모 노드 update
update_parent(curr_page->parent_page_offset, left_child_offset, new_key);
//이제 부모노드의 수정도 끝났으므로 다 free해줌 (이 함수에서는 next_page를 제외하고는 변경사항이 없으므로 free만 해주기)
free(curr_page);
free(next_page);
free(temp);
return 0;
```

Parent에서 현재 page와 오른쪽 sibling page를 나누어주던 방향 지시등 값이 변해야 하므로 new_key라는 변수를 만들어 update_parent 함수를 호출해 parent에서 방향지시등 값을 바꾸어주도록 한다.

Update_parent함수를 살펴보자.

```
//key_rotation에서 internal node의 key를 업데이트 하는 과정
void update_parent(off_t parent_offset, off_t left_child_offset, int64_t new_key){
    //parent page를 load
    page * parent_page = load_page(parent_offset);
    for(int i = 0; i <= parent_page->num_of_keys; i++){
        //parent page가 꽉 차 있는 경우
        if(parent_page->num_of_keys == INTERNAL_MAX){
            if(i == parent_page->num_of_keys){
                if(parent_page->next_offset == left_child_offset){
                    break;
                }
            }
            //꽉 찼지만 같은 부모인 경우
            else{
                if(parent_page->b_f[i].p_offset == left_child_offset){
                    parent_page->b_f[i].key = new_key;
                    pwrite(fd, parent_page, sizeof(page), parent_offset);
                    free(parent_page);
                    return;
                }
            }
        }
        //parent가 꽉 안 차있는 경우
        else{
            if(i == parent_page->num_of_keys){
                if(parent_page->b_f[i].p_offset == left_child_offset){
                    break;
                }
            }
            else{
                if(parent_page->b_f[i].p_offset == left_child_offset){
                    parent_page->b_f[i].key = new_key;
                    pwrite(fd, parent_page, sizeof(page), parent_offset);
                    free(parent_page);
                    return;
                }
            }
        }
    }
```

Update_parent 함수에서는 먼저 현재 page와 sibling node를 가르던 방향지시등 key를 찾는다.

만약 방향지시등 key를 찾았다면 이를 새로운 key로 교체해주고 끝이 난다. 하지만 여기서 고려해야 할 점은 방향지시등 key를 못 찾았을 경우이다. 못 찾았을 경우는 현재

page와 sibling node가 서로 다른 부모를 가지고 있음을 의미한다. 이 경우 parent의 parent를 계속 재귀적으로 (root까지) 타고 올라가서 두 node를 가르는 방향지시등을 찾아 new_key로 바꾸어 주어야 한다.

```
    }
    //old key를 못찾았을 경우에는 parent의 parent에 old key 값이 있다는 것
    //이를 재귀적으로 타고 올라가서 고쳐줘야 함
    off_t g_parent_offset = parent_page->parent_page_offset;
    //header page가 아닐 때 까지 올라가서 계속 old key를 찾음
    if(g_parent_offset != 0){
        //한 번 왼쪽에 있던것은 계속 왼쪽 자손
        update_parent(g_parent_offset, parent_offset, new_key);
    }
    //parent_page를 free 시켜줌
    free(parent_page);
}
```

다음과 같이 재귀적으로 root까지 타고올라가 방향지시등 값을 new_key로 바꾸어 준다. 이와 같이 key-rotation이 끝이난다.

Split

다시 db_insert로 돌아오자. Key-rotation이 불가능한 경우, 즉, **오른쪽 sibling이 존재하지 않거나 오른쪽 sibling이 존재해도 그 page에 여유공간이 없는 경우 split을 수행한다.**

```
//이제 split!
//새로운 노드 만들기
off_t new_leaf_offset = new_page();
//new_page()에서 파일 시스템의 최대 크기 도달하여 더이상 페이지를 할당할 수 없는 경우
//-1을 리턴하기 때문에 이와 같이 수정해줌
if(new_leaf_offset == -1){
    printf("No more Page\n");
    return -1;
}
page * new_leaf = load_page(new_leaf_offset);
//새로운 노드는 leaf node이므로 is_leaf를 1로 설정
new_leaf->is_leaf = 1;
new_leaf->parent_page_offset = curr_page->parent_page_offset;
new_leaf->num_of_keys = 0;
```

먼저 새로운 page를 할당받는다. 만약 더이상 파일에 여유공간이 없으면 insert를 할

수 없으니 insert를 실패하도록 하였다.

앞서 언급한 바와 같이, db_insert함수에서는 leaf node에 대한 처리만을 해준다. 후에 insert_in_parent함수가 internal node에 대해 처리해줄 것이다. **때문에 새롭게 생성된 page는 leaf node이다.**

```
record * temp = (record*)malloc((LEAF_MAX + 1) * sizeof(record));
int k = 0;
//key보다 작은 것들 먼저 temp에 넣기
while(k < curr_page->num_of_keys && key > curr_page->records[k].key){
    temp[k].key = curr_page->records[k].key;
    strcpy(temp[k].value, curr_page->records[k].value);
    k++;
}
//insert할 새로운 key, value 값 넣기
temp[k].key = key;
strcpy(temp[k].value, value);

//나머지 record를 temp에 넣기
for(; k < curr_page->num_of_keys; k++){
    temp[k+1].key = curr_page->records[k].key;
    strcpy(temp[k+1].value, curr_page->records[k].value);
}
```

먼저 temp라는 memory 공간을 할당받아 현재 Page의 모든 element와 새롭게 추가할 key, value값을 넣어준다. (오름차순으로)

```
//Memory 임시 공간에 이제 새로운 key, value를 포함해서 모두 올라감
//sibling pointer 설정 -> 새로운 노드가 오른쪽에 형성된다고 하자
new_leaf->next_offset = curr_page->next_offset;
curr_page->next_offset = new_leaf_offset;
```

Split시 새로운 node는 오른쪽에 생긴다고 가정하자. 이 때 sibling을 연결해주는 작업을 수행한다.

```
//비교 로직중에 key와 비교하는 것이 많으므로 0으로 초기화하면 문제가 발생할 수 있기에 모든 값을 int64_t의 최대로 초기화
for(int i = 0; i < curr_page->num_of_keys; i++){
    curr_page->records[i].key = __INT64_MAX__;
    memset(curr_page->records[i].value, '\0', sizeof(curr_page->records[i].value));
}
//이제 초기화했으니 들어있는 key의 개수는 없음
curr_page->num_of_keys = 0;
```

현재 page를 초기화해준다.

```
//Leaf node의 order는 31(record pointer) + 1(next_offset)이므로 32
int middle = cut(LEAF_MAX + 1);
//먼저 왼쪽 node의 반 넣기
for(int i = 0; i < middle; i++){
    curr_page->records[i].key = temp[i].key;
    strcpy(curr_page->records[i].value, temp[i].value);
    curr_page->num_of_keys++;
}

//새로운 노드에 나머지 넣기
for(int i = middle; i < LEAF_MAX + 1; i++){
    new_leaf->records[i-middle].key = temp[i].key;
    strcpy(new_leaf->records[i-middle].value, temp[i].value);
    new_leaf->num_of_keys++;
}
```

LEAF_MAX는 key의 개수이기에 order는 LEAF_MAX + 1이다. Middle (중앙값)을 잡고 temp의 값을 현재 page와 새로운 page에 나누어 넣어준다.

```
//부모에 대해 재귀적으로 돌기 위해 new_leaf에 최소 key값과 부모 node의 offset 가져오기
int64_t V_key = new_leaf->records[0].key;
int parent_offset = curr_page->parent_page_offset;

//이제 leaf에서 작업은 끝남 output해주고 free해줌
pwrite(fd, curr_page, sizeof(page), p_offset);
pwrite(fd, new_leaf, sizeof(page), new_leaf_offset);
free(curr_page);
free(new_leaf);
free(temp);
//부모에 대해서 재귀적으로 돌게함
return insert_in_parent(parent_offset, p_offset, V_key, new_leaf_offset);
```

부모 page에 새롭게 들어가게 될 (부모 기준에서 새롭게 insert가 될) key를 잡고 disk에 업데이트 내용을 모두 작성 후 insert_in_parent를 호출하여 internal node에서 split 처리를 하도록 한다.

Insert_in_parent를 살펴보도록 하자.

```
int insert_in_parent(off_t parent_offset, off_t left_child, int64_t key, off_t right_child){
    //당연히 left child가 원래 존재했던 node고 right child가 새로 생긴 node이기에 당연히 left child가 root인지 검사해야함
    //이때 새로운 root는 분명 leaf가 아닐것임 때문에 b_f에 값을 추가해줘야함
    if(hp->rpo == left_child){
        off_t new_node_offset = new_page();
        if(new_node_offset == -1){
            printf("No more page\n");
            return -1;
        }
        page * new_node = load_page(new_node_offset);
        page * left_child_node = load_page(left_child);
        page * right_child_node = load_page(right_child);
        //새로운 root가 internal node임을 명시
        new_node->is_leaf = 0;
        //새로운 root에 key와 pointer 값 넣어주기
        new_node->b_f[0].p_offset = left_child;
        new_node->b_f[0].key = key;
        new_node->b_f[1].p_offset = right_child;
        //header page의 root pointer 값 바꿔주기
        hp->rpo = new_node_offset;
        new_node->num_of_keys = 1;
        pwrite(fd, new_node, sizeof(page), new_node_offset);
        pwrite(fd, hp, sizeof(H_P), 0);
        free(hp);
        hp = load_header(0);

        //부모 노드 바꿔주기
        left_child_node->parent_page_offset = new_node_offset;
        right_child_node->parent_page_offset = new_node_offset;
        pwrite(fd, left_child_node, sizeof(page), left_child);
        pwrite(fd, right_child_node, sizeof(page), right_child);
        free(left_child_node);
        free(right_child_node);
        free(new_node);
        return 0;
    }
```

먼저 left_child (leaf기준에서는 현재 page)가 root라면 root에 split이 일어난 경우이므로 새로운 root가 필요하다. 때문에 새로운 root를 형성하여 child로 split된 node들을 가지도록 한다.

```
//Internal node의 order는 leaf와 다르기 때문에 249임
//부모에게 여유 공간이 있을 때
//먼저 parent page를 load
page * parent_page = load_page(parent_offset);
if(parent_page->num_of_keys < INTERNAL_MAX){
    int i;
    //left child pointer를 찾기
    for(i = 0; i <= parent_page->num_of_keys; i++){
        if(parent_page->b_f[i].p_offset == left_child){
            break;
        }
    }
```

Child가 root가 아닌 경우 그리고 parent page에 여유공간이 남아 있어 삽입이 바로 되는 경우, 먼저 parent_page를 load하고 parent_page에서 child의 위치를 찾는다. (새로운 key가 insert 될 위치)

```
//i번째에 지금 left child pointer가 속해있으므로 i번째 key와 i+1번째 p_offset부터 옮겨줌
//여기서부터 next_offset 변수 때문에 두 가지 케이스로 나뉨
//Case 1) 새로운 key와 pointer가 추가되었을 때 꽉 차는 경우
if(parent_page->num_of_keys == INTERNAL_MAX - 1){
    //나머지 것들 이동시켜줌
    for(int j = parent_page->num_of_keys; j > i; j--){
        //next_offset으로 pointer를 이동해줘야 하는 경우
        if(j == parent_page->num_of_keys){
            parent_page->b_f[parent_page->num_of_keys].key = parent_page->b_f[parent_page->num_of_keys - 1].key;
            parent_page->next_offset = parent_page->b_f[parent_page->num_of_keys].p_offset;
        }
        //pointer의 index값이 이 경우 key의 index 값보다 1이 크므로
        else{
            parent_page->b_f[j+1].p_offset = parent_page->b_f[j].p_offset;
            parent_page->b_f[j].key = parent_page->b_f[j-1].key;
        }
    }
}
//Case 2) 좀 공간이 널널하게 남아 있을 때
else{
    for(int j = parent_page->num_of_keys; j > i; j--){
        parent_page->b_f[j+1].p_offset = parent_page->b_f[j].p_offset;
        parent_page->b_f[j].key = parent_page->b_f[j-1].key;
    }
}

//밀어준 후에 key, pointer 삽입
if(parent_page->num_of_keys == INTERNAL_MAX - 1 && i == parent_page->num_of_keys){
    parent_page->b_f[i].key = key;
    parent_page->next_offset = right_child;
}
else{
    parent_page->b_f[i].key = key;
    parent_page->b_f[i+1].p_offset = right_child;
}
//parent_page key 개수 증가시키기
parent_page->num_of_keys++;
//parent page에 삽입이 끝났으니 output해주기

pwrite(fd, parent_page, sizeof(page), parent_offset);
free(parent_page);
return 0;
```

주어진 key를 parent_page에 insert한다.

만약 parent_page에도 여유공간이 없는 경우에는 parent_page에서도 split이 일어나야 된다. 이 경우 parent_page의 parent에도 insert가 일어나는 상황이므로 재귀적으로 처리해줘야 한다.

```
//이제 부모 노드에 여유공간이 없을 때
//부모노드에서도 split이 일어나야 함
//split을 위한 노드를 하나 만듦
off_t new_parent_offset = new_page();
if(new_parent_offset == -1){
    printf("No more page\n");
    return -1;
}
```

먼저 새로운 page를 할당하여 split을 준비한다. 더이상 file에 남은 공간이 없으면 insert를 실패하게 한다.

```
}
page * new_parent = load_page(new_parent_offset);
new_parent->parent_page_offset = parent_page->parent_page_offset;
//새로운 노드는 internal node일 것이므로 is_leaf를 0으로 설정
new_parent->is_leaf = 0;
//Memory에 임시 저장할 공간을 둠 => pointer의 개수가 Internal_Max + 1개이고 추가할 pointer가 있으므로 +1 해서 INTERNAL_MAX + 2 만큼 공간 할당
//temp의 마지막 key는 활용하지 않는 것임
I_R * temp = (I_R*)malloc((INTERNAL_MAX + 2) * sizeof(I_R));
int k = 0;
//Memory 공간에 key보다 작은 것까지 복사
while(k < parent_page->num_of_keys && key > parent_page->b_f[k].key){
    temp[k].key = parent_page->b_f[k].key;
    temp[k].p_offset = parent_page->b_f[k].p_offset;
    k++;
}
//여기서 궁금할 수 있는 점은 next_offset까지 도달했을 때 이를 복사 안하냐는 것인데
//어차피 key때문에 위에서 모든 값이 복사되고 next_offset만 복사가 안됐다면
//그 next_offset에 들어있는 pointer는 left_child 일 것임
//아무튼 일반적으로는 중간에 삽입하는 가정이고 이는 끝단에서도 잘 먹힘
temp[k].key = key;
temp[k].p_offset = left_child;
temp[k+1].p_offset = right_child;

//나머지를 memory 공간에 삽입
for(int j = k; j < parent_page->num_of_keys; j++){
    temp[j+1].key = parent_page->b_f[j].key;
    //pointer를 next_offset을 따로 추가해줘야 하므로 분기
    if(j == INTERNAL_MAX - 1){
        temp[j+2].p_offset = parent_page->next_offset;
    }
    else{
        temp[j+2].p_offset = parent_page->b_f[j+1].p_offset;
    }
}
```

Memory 공간을 할당받고 새롭게 추가된 key와 모든 pointer를 옮겨준다.

```
}
//temp에 새로운 key, pointer 값 모두가 복사됨
//햔제 parent_page의 모든 내용 삭제
for(int i = 0; i < parent_page->num_of_keys; i++){
    //key들을 int64max로 초기화
    parent_page->b_f[i].key = __INT64_MAX__;
    //p_offset을 0으로 초기화
    parent_page->b_f[i].p_offset = 0;
}

//마지막 offset을 0으로 초기화
parent_page->next_offset = 0;
//num_of_keys를 0으로 초기화
parent_page->num_of_keys = 0;
```

모든 key, pointer를 temp에 옮긴 후 parent_page를 초기화해준다.

```
    //Internal node의 ordersms 248(b_f) + 1(next_offset
    int middle = cut(INTERNAL_MAX + 1);
    //반을 왼쪽에 추가
    for(int i = 0; i < middle; i++){
        if(i != middle - 1){
            parent_page->b_f[i].key = temp[i].key;
            parent_page->num_of_keys++;
        }
        parent_page->b_f[i].p_offset = temp[i].p_offset;
    }
    //부모로 전달해줄 key를 냅두고
    int64_t new_key = temp[middle-1].key;

    //부모가 split이 일어날 때 자식 노드의 parent_page_offset을 바꿔주기 위함
    page* child_temp;
    //새로운 노드에 추가해주기
    for(int i = middle; i < INTERNAL_MAX + 2; i++){
        new_parent->b_f[i-middle].p_offset = temp[i].p_offset;
        if(i != INTERNAL_MAX + 1){
            new_parent->b_f[i-middle].key = temp[i].key;
            new_parent->num_of_keys++;
        }
        //여기서 문제가 발생 => 부모가 split이 일어나 자식 노드들의 parent_page_offset 정보의 수정이 필요해짐
        child_temp = load_page(temp[i].p_offset);
        child_temp->parent_page_offset = new_parent_offset;
        pwrite(fd, child_temp, sizeof(page), temp[i].p_offset);
        free(child_temp);
    }
    //여기까지 parent_page와 new_parent의 역할을 서로 다했기 때문에 이들을 다시 disk에 output해줌
    //먼저 gparent offset을 받아놓기
    off_t gparent_offset = parent_page->parent_page_offset;

    pwrite(fd, parent_page, sizeof(page), parent_offset);
    pwrite(fd, new_parent, sizeof(page), new_parent_offset);
    free(parent_page);
    free(new_parent);
    free(temp);
    insert_in_parent(gparent_offset, parent_offset, new_key, new_parent_offset);
}
```

Split을 수행한다. Middle 변수를 두어 temp의 key, pointer 값들을 parent_page와 새로운 new_parent가 나누어 가지게 한다. 모두 옮기고 나면 disk에 output해주고 다음 부모에게도 Insert가 일어나는 것이니 insert_in_parent 함수를 재귀적으로 호출해준다.

이와 같이 key_rotation과 split을 사용하여 insert를 구현하였다,

Delete

Delete의 경우 merge와 redistribution으로 이루어져있다. 단계별로 살펴보도록 하자. Db_delete함수에서는 leaf_node를 찾는 것을 주되게 하고 delete_entry함수에서 본격적인 delete를 한다.

```
int db_delete(int64_t key) {
    page * curr_page = load_page(hp->rpo);
    int i;
    off_t p_offset = hp->rpo;

    while (!curr_page->is_leaf) {
        //internal node를 돌면서 key값 비교
        for (i = 0; i < curr_page->num_of_keys; i++) {
            if (key <= curr_page->b_f[i].key) {
                break;
            }
        }
        //끝에 노드까지 도달한 경우 loop를 빠져나오면서 i가 num_of_keys가 됨
        //key는 0~num_of_keys-1까지 있겠지만 p_offset의 개수는 0~num_of_keys만큼 있을 것임
        if(i == curr_page->num_of_keys){
            //key가 꽉 차있을 경우 결국 맨 오른쪽 pointer 이용해야함
            if(curr_page->num_of_keys == INTERNAL_MAX){
                //맨 오른쪽 pointer는 next_offset에 저장되기에 p_offset을 맨 오른쪽 pointer로 설
                p_offset = curr_page->next_offset;
            }
            //만약 key가 꽉 안 차있을 경우 num_of_keys의 pointer를 이용하면 됨
            else{
                p_offset = curr_page->b_f[i].p_offset;
            }
        }
        //이 때는 key가 딱 curr_page의 key와 동일해서 break 문을 밟고 온 것
        //이 때는 Loop에서 1이 더 증가하지 않은 상태로 나옴
        //때문에 p_offset을 그냥 i+1로 해당 key의 오른쪽 offset으로 접근하게 함
        else if(key == curr_page->b_f[i].key){
            //여기서도 i+1이 num_of_keys와 같으면 맨 오른쪽 pointer로 이동시켜줘야 함
            if(i+1 == INTERNAL_MAX){
                p_offset = curr_page->next_offset;
            }
            //꽉 차있지 않은 경우에는 그냥 현재 key의 오른쪽 pointer 이용하면 됨
            else{
                p_offset = curr_page->b_f[i+1].p_offset;
            }
        }
        //key가 search key보다 작을 때에는 그냥 왼쪽 pointer 따라가게 함
        else{
            p_offset = curr_page->b_f[i].p_offset;
        }
        // 현재 페이지를 해제하고 다음 페이지를 로드
        free(curr_page);
        curr_page = load_page(p_offset);
    }
```

먼저 delete할 key가 포함된 leaf node를 찾는다.

```
//이제 curr_page가 leaf_node에 도달
//p_offset이 이를 가리킴
free(curr_page);
return delete_entry(p_offset, key);
```

delete할 key를 찾았으니 delete_entry 함수를 호출하여 본격적으로 delete를 수행한다.

```
int delete_entry(off_t node_offset, int64_t key){
    page * node = load_page(node_offset);
    //먼저 entry를 삭제할 건데 leaf랑 internal이랑 구분해야됨
    //Leaf인 경우
    if(node->is_leaf){
        int i = 0;
        int flag = 0;
        //leaf node에서 삭제할 key 찾기
        for(; i < node->num_of_keys; i++){
            if(node->records[i].key == key){
                flag = 1;
                break;
            }
        }
        if(flag == 0){
            printf("Entry Not found\n");
            return -1;
        }

        //한 칸씩 당겨주기 (삭제시키기)
        for(; i < node->num_of_keys - 1; i++){
            node->records[i].key = node->records[i+1].key;
            strcpy(node->records[i].value, node->records[i+1].value);
        }

        //마지막 노드는 초기화
        node->records[node->num_of_keys - 1].key = __INT64_MAX__;
        memset(node->records[node->num_of_keys - 1].value, '\0', sizeof(node->records[node->num_of_keys - 1].value));

        //삭제가 완료되었으므로 key 개수 한개 줄이기
        node->num_of_keys--;
        //일단 삭제가 완료되었으므로 output
        pwrite(fd, node, sizeof(page), node_offset);
    }
```

먼저 leaf node에서 해당 entry를 찾아 delete를 한다. Delete 하는 매커니즘이 leaf와 internal node가 서로 다르다. 왜냐하면 leaf에서는 records에 모든 데이터를 담고 있지만 internal에서는 b_f에 모든 데이터를 담고 있기 때문이다. 때문에 이 둘을 분리하여 구현하였다. 또한, leaf에서 삭제하려는 node가 해당 leaf node에 존재하지 않을 때에는 delete를 실패하도록 하였다. (internal에서의 delete는 merge시 parent에도 delete가 발생하는 것이므로 그 때 재귀적인 호출을 위해 구현해두었다)

```
}
//internal node인 경우
//주어지는 pointer는 무조건 key의 오른쪽 pointer일 것임
//왜냐하면 delete를 하고 merge를 한 경우
else{
    int i = 0;
    //internal node에서 key 찾기
    for(; i < node->num_of_keys; i++){
        if(node->b_f[i].key == key){
            //만약 node가 꽉 차있고 삭제할 node와 pointer가 맨 마지막 것 일때
            if(node->num_of_keys == INTERNAL_MAX && i == node->num_of_keys - 1){
                //마지막 key와 pointer 삭제
                node->b_f[i].key = __INT64_MAX__;
                node->next_offset = 0;
            }
            //일반적으로 삭제할 key, pointer가 중간에 있을 때
            else{
                for(int j = i; j < node->num_of_keys - 1; j++){
                    if(node->num_of_keys == INTERNAL_MAX && j == node->num_of_keys - 2){
                        node->b_f[j].key = node->b_f[j+1].key;
                        node->b_f[j+1].p_offset = node->next_offset;
                    }
                    else{
                        node->b_f[j].key = node->b_f[j+1].key;
                        node->b_f[j+1].p_offset = node->b_f[j+2].p_offset;
                    }
                }
            }
            //삭제가 완료되었으므로 num_of_keys - 1
            node->num_of_keys--;
            break;
        }
    }
    //삭제가 완료되었으므로 output
    pwrite(fd, node, sizeof(page), node_offset);
}
```

Internal node에서 해당 key를 포함한 entry를 삭제하는 부분이다.

```
//현재 페이지가 루트 페이지이면서 키가 하나도 없을 경우
if(hp->rpo == node_offset){
    if(node->num_of_keys == 0){
        //현재 노드가 leaf가 아니면서 key가 더이상 존재하지 않고 오직 하나의 child만 가지고 있을 때
        if(!node->is_leaf && node->b_f[0].p_offset != 0){
            //root page를 child로 바꿈
            hp->rpo = node->b_f[0].p_offset;
            //새로운 root page 로드
            page * child = load_page(hp->rpo);
            //새로운 root의 parent page를 header page로 설정
            child->parent_page_offset = 0;
            pwrite(fd, child, sizeof(page), hp->rpo);
            //header page의 root가 바뀌었으므로 write
            free(child);
            //usetofree 함수에 header page를 load하는 부분이 존재함 때문에 여기서 미리 output을 해줘야함
            pwrite(fd, hp, sizeof(H_P), 0);
            free(hp);
            hp = load_header(0);
            //node, 즉, 원래의 root page를 삭제해주고 Linked list에 추가해줌
            usetofree(node_offset);
        }
        //현재 node가 leaf node이고 더이상 key가 남아 있지 않을 때
        //사실상 tree가 삭제된 케이스
        else if(node->is_leaf){
            //root page를 삭제해주고 header page의 정보를 update해줌
            hp->rpo = 0;
            //header page에 변화가 생겼으므로 output
            //usetofree함수때문에 미리 다시 load
            pwrite(fd, hp, sizeof(H_P), 0);
            free(hp);
            hp = load_header(0);
            usetofree(node_offset);
        }
        else{
            //에러가 발생함
            return -1;
        }
        free(node);
        //delete 성공했으므로 0 리턴
        return 0;
    }
    //key가 1개 이상이라면 root는 그냥 놔두어도 됨
    free(node);
    return 0;
}
```

Merge를 통한 재귀적인 호출 과정에서 root의 key가 삭제될 수도 있다. root의 key가 삭제되어 **root에 key가 하나도 남지 않고 pointer 하나만 (자식을 가리키는 pointer) 남았을 경우 root를 삭제하고 해당 자식을 새로운 root로 만들도록 하였다.** 또한, **root의 경우 삭제를 하더라도 key가 한 개만 있어도 충분하기 때문에 삭제 후 internal node 개수 검증을 거치지 않고 리턴하게 하였다.**

```
if(node->is_leaf){
    //아직 공간이 충분하므로
    if(node->num_of_keys >= cut(LEAF_MAX)){
        free(node);
        return 0;
    }
}
else{
    if(node->num_of_keys >= cut(INTERNAL_MAX + 1) - 1){
        free(node);
        return 0;
    }
}
```

삭제 후 leaf 인 경우와 internal인 경우 개수 검증을 하게 하였다. 만약 개수가 충분하다면 delete를 성공하게 하고 리턴하게 하였다.

Merge

```
//이제는 진짜 여유 공간이 없음
//Merge & Redistribution
//한 Node에 몰아줄 수 있는 경우와 그렇지 않은 경우로 나뉨
//여기서도 leaf인지 internal인지 경우가 나뉨
//먼저 leaf인 경우 => parent는 무조건 internal node일 것

//sibling node가 왼쪽일지 오른쪽일지 알려주는 variable => 왼쪽이면 1, 오른쪽이면 0
int is_left_sibling = 0;
off_t parent_offset = node->parent_page_offset;
int k_prime;
int sibling_offset = get_neighbor_offset(node_offset, parent_offset, node->num_of_keys, &is_left_sibling, &k_prime);
```

이제는 해당 page에서 element 삭제 이후 element의 개수가 충분하지 않다. 때문에 merge 또는 redistribution이 일어날 것이다. 먼저 merge부터 수행한다. Get_neighbor_offset함수를 통해 여유로운 sibling을 찾는다. Get_neighbor_offset 함수가 몹시 길어 WIKI에는 첨부하지 않겠다. 이 함수의 대략적인 구조는 우선적으로 왼쪽 sibling을 살펴보아 해당 sibling에 merge할 수 있는 condition이 만족되면 왼쪽 sibling을 선택하고 왼쪽 sibling이 이를 만족시키지 못한다면 그제서야 오른쪽 sibling을 보아 merge condition을 만족하는지 확인한다. 여기서 말하는 merge condition은 대략적으로 두 node의 element 개수를 합했을 때 (K_prime 값도 포함) LEAF_MAX 또는 INTERNAL_MAX를 초과하지 않는지 이다. (이 구현과정이 매우 긴 이유는 leaf와 internal의 케이스를 구분해야 하고 맨 왼쪽 자식이라 왼쪽 sibling이 없는 경우 등등 고려해야 할 케이스가 많기 때문이다.)

Merge를 수행하는 과정도 4가지 케이스로 나뉜다.

- Leaf node에서 left sibling에 대해 merge
- Leaf node에서 right sibling에 대해 merge
- Internal node에서 left sibling에 대해 merge
- Internal node에서 right sibling에 대해 merge

여기서는 left sibling에 대한 merge만 보도록 하겠다.

```
//만약 여유공간이 있는 sibling을 발견했으면
if(sibling_offset != -1){
    page * node_sibling = load_page(sibling_offset);
    //sibling node가 왼쪽 sibling이면
    if(is_left_sibling){
        //만약 node가 leaf node이면 records로 옮겨줘야 함
        if(node->is_leaf){
            //node에 있는 모든것을 왼쪽 sibling에게 넘겨주기
            for(int i = node_sibling->num_of_keys ;i < node_sibling->num_of_keys + node->num_of_keys; i++){
                node_sibling->records[i].key = node->records[i-node_sibling->num_of_keys].key;
                strcpy(node_sibling->records[i].value, node->records[i].value);
            }
            node_sibling->num_of_keys += node->num_of_keys;
            node->num_of_keys = 0;
            //sibling 연결해주기
            node_sibling->next_offset = node->next_offset;

            //sibling에 일단 write가 되었으므로 output 시키기
            pwrite(fd, node_sibling, sizeof(page), sibling_offset);
            usetofree(node_offset);
            free(node);
            free(node_sibling);
            return delete_entry(parent_offset, k_prime);
        }
```

먼저 leaf node에 대해 left sibling에 대한 merge 과정이다. 간단하게 설명하면 left sibling에 현재 node의 모든것을 넘겨주고 node를 삭제하는 것이다. K_prime은 부모 노드에서 원래 left sibling과 node를 갈라주던 방향지시등인데 delete_entry 함수를 재귀적으로 호출하여 부모에서 k_prime을 삭제토록 한다.

```
//sibling node가 왼쪽이고 현재 node가 internal node인 경우
else{
    //k_prime을 sibling에 넣어주기
    node_sibling->b_f[node_sibling->num_of_keys].key = k_prime;
    //node의 첫 pointer 넘겨주기
    node_sibling->b_f[node_sibling->num_of_keys + 1].p_offset = node->b_f[0].p_offset;
    //sibling node값 증가시켜주기
    node_sibling->num_of_keys++;

    for(int i = 0; i < node->num_of_keys; i++){
        //마지막 node와 pointer를 넘겨주는데 sibling node가 거의 꽉 차 있음
        if(i == node->num_of_keys - 1 && node_sibling->num_of_keys == INTERNAL_MAX - 1){
            node_sibling->b_f[node_sibling->num_of_keys].key = node->b_f[i].key;
            node_sibling->next_offset = node->b_f[i+1].p_offset;
        }
        //일반적인 경우 node의 모든 pointer와 값들을 넣어줌
        else{
            node_sibling->b_f[node_sibling->num_of_keys].key = node->b_f[i].key;
            node_sibling->b_f[node_sibling->num_of_keys + 1].p_offset = node->b_f[i+1].p_offset;
        }
        node_sibling->num_of_keys++;
    }

    //모든 node의 자식들의 parent_page_offset을 node_sibling으로 옮즘
    //어차피 node는 좀 비어있을테니 next_offset을 접근 안해도 됨
    for(int j = 0; j <= node->num_of_keys; j++){
        page * temp = load_page(node->b_f[j].p_offset);
        temp->parent_page_offset = sibling_offset;
        pwrite(fd, temp, sizeof(page), node->b_f[j].p_offset);
        free(temp);
    }

    //node_sibling으로 합치기가 끝났으므로 output 해줌
    pwrite(fd, node_sibling, sizeof(page), sibling_offset);
    //node는 삭제시켜줌
    usetofree(node_offset);
    free(node_sibling);
    free(node);
    return delete_entry(parent_offset, k_prime);
}
```

Internal node의 경우 leaf node와 거의 동일하다. 주의할 점은 삭제될 internal node의 자식들의 parent_page_offset을 모두 옮겨주는 것이다.

Redistribution

Merge가 실패했을 경우 (sibling에서 merge를 할 sibling을 찾지 못했을 경우) redistribution을 수행한다.

```
//이제 redistribution
off_t sibling_node_offset = get_neighbor_for_redistribution(node_offset, parent_offset, &is_left_sibling, &k_prime);
```

이와 같이 get_neighbor_for_redistribution 함수를 활용해 redistribute할 sibling을 선택한다. Redistribute를 수행할 sibling을 고르는 과정은 merge 할 sibling을 고르는 과정과 다르다. **왜냐하면 필자의 구현에서는 merge를 하지 못했을 때 redistribution을 수행하**

는 것인데 merge를 하지 못했다는 것은, 즉, merge를 할 sibling을 찾지 못했다는 것, 즉, 양쪽 모든 sibling이 merge condition을 만족하지 못했다는 것이다. 이말인 즉슨, 양쪽 모든 sibling이 element개수가 redistribute 하기 충분하다는 소리다. 때문에 redistribute를 수행할 때 sibling 선택과정은 다소 간단하다. 먼저, 왼쪽 sibling을 우선적으로 redistribution 할 sibling으로 선택한다. 하지만 만약 맨 왼쪽 node의 경우에만 오른쪽 sibling을 통해 redistribution을 수행하도록 구현하였다.

Redistirbution에서 또 **주요한 점은 재귀가 안 일어난다는 것이다**. 문자 그대로 재분배이기 때문에 parent에서 key의 교체가 일어나지 parent에서 다른 삭제 등이 일어나지 않기 때문에 재귀적으로 처리할 필요가 없다.

Redistribution이 일어나는 상황 역시 4가지 이다.

- Leaf node에서 left sibling에 대해 redistribute
- Internal node에서 left sibling에 대해 redistribute
- Leaf node에서 right sibling에 대해 redistribute
- Internal node에서 right sibling에 대해 redistribute

이번에도 역시 left sibling에 대한 것만 살펴보도록 하겠다.

```
//왼쪽 sibling에서 가져오기
if(is_left_sibling){
    //현재 node가 leaf node인 경우
    if(node->is_leaf){
        //일단 node를 밀어주기
        for(int i = node->num_of_keys; i > 0; i--){
            node->records[i].key = node->records[i-1].key;
            strcpy(node->records[i].value, node->records[i-1].value);
        }
        //왼쪽 sibling node의 마지막 key와 pointer를 node에 삽입
        node->records[0].key = sibling_node->records[sibling_node->num_of_keys - 1].key;
        strcpy(node->records[0].value, sibling_node->records[sibling_node->num_of_keys - 1].value);
        page * parent = load_page(parent_offset);
        //부모의 k_prime을 왼쪽 sibling의 마지막 key값으로 교체
        for(int i = 0; i < parent->num_of_keys; i++){
            if(parent->b_f[i].key == k_prime){
                parent->b_f[i].key = sibling_node->records[sibling_node->num_of_keys - 1].key;
            }
        }

        //각가의 node의 num_of_keys 값 조정해주기
        node->num_of_keys++;
        sibling_node->num_of_keys--;
        pwrite(fd, node, sizeof(page), node_offset);
        pwrite(fd, sibling_node, sizeof(page), sibling_node_offset);
        pwrite(fd, parent, sizeof(page), parent_offset);
        free(node);
        free(sibling_node);
        free(parent);
        return 0;
    }
```

먼저 leaf node에서의 redistribution을 살펴보자. 일단 현재 node의 맨 앞에 left sibling의 제일 마지막 key와 value를 가져와 삽입한다. 그리고 parent에서 원래 left sibling과 현재 node를 갈라주던 방향지시등 key (k_prime)을 새롭게 node에 삽입된 key로 바꾸어 준다.

Internal node의 경우 node의 제일 앞에 left sibling의 마지막 pointer와 k_prime을 가져온 후 parent에서 방향지시등 key를 왼쪽 sibling node의 제일 마지막 key로 바꾸어줌.

```
//Left sibling에서 가져오고 Internal node인 경우
else{
    //일단 node 밀어주기
    for(int i = node->num_of_keys; i > 0; i--){
        //key는 일반적으로 그냥 밀어주면 됨
        node->b_f[i].key = node->b_f[i-1].key;
        //pointer의 경우 next_offset을 고려해서 밀어줌
        //(key, pointer) 이렇게 한 쌍을 오른쪽으로 옮겨준다고 생각하면됨
        if(node->num_of_keys == INTERNAL_MAX - 1 && i == node->num_of_keys){
            node->next_offset = node->b_f[i+1].p_offset;
        }
        else{
            node->b_f[i+2].p_offset = node->b_f[i+1].p_offset;
        }
    }
    //아직 맨 처음 포인터가 안옮겨졌을 것임
    //맨 처음 포인터도 밀어주기
    node->b_f[1].p_offset = node->b_f[0].p_offset;
    //밀어주기 끝

    //node에 값을 가져옴 K_PRIME이랑 sibling_node의 마지막 pointer
    //이 값을 node의 첫 pointer와 key에 삽입
    //sibling node가 꽉 차 있는 경우에는 마지막 pointer가 next_offset이므로
    if(sibling_node->num_of_keys == INTERNAL_MAX){
        node->b_f[0].p_offset = sibling_node->next_offset;
    }
    else{
        node->b_f[0].p_offset = sibling_node->b_f[sibling_node->num_of_keys].p_offset;
    }
    node->b_f[0].key = k_prime;

    page * parent = load_page(parent_offset);

    //parent node의 key값 수정
    for(int i = 0; i < parent->num_of_keys; i++){
        if(parent->b_f[i].key == k_prime){
            parent->b_f[i].key = sibling_node->b_f[sibling_node->num_of_keys - 1].key;
        }
    }

    //key 개수 update
    node->num_of_keys++;
    sibling_node->num_of_keys--;

    pwrite(fd, node, sizeof(page), node_offset);
    pwrite(fd, sibling_node, sizeof(page), sibling_node_offset);
    pwrite(fd, parent, sizeof(page), parent_offset);
    free(node);
    free(parent);
    free(sibling_node);
    return 0;
}
```

이와 같이 redistribution을 수행하여 deletion을 완료한다.

Find

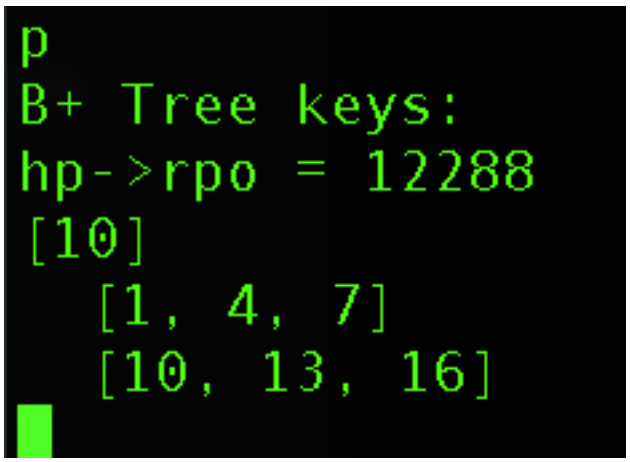
Find 함수는 간단하게 구현하였다. 해당 key가 없으면 Not Exist가 출력되게 main함수에서 수정해놨고 해당 key가 포함된 leaf를 먼저 찾고 그 leaf에서 해당 key의 value를 읽도록 하였다. 이 과정에서 함수 밖으로 key에 대응하는 value를 전달해야 하므로 malloc이 일어난다. 이를 main에서 free하여 memory leak을 방지하였다. (코드 첨부는 생략하도록 하겠다.)

Result

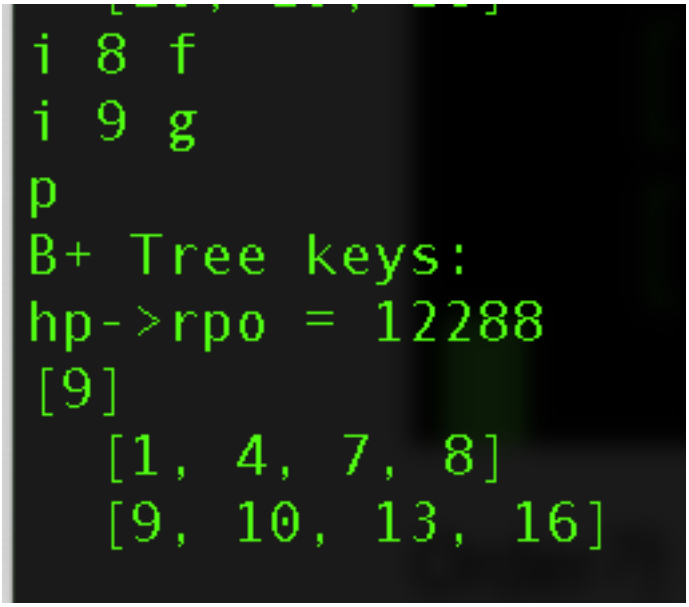
어떠한 결과를 보여줘야 될 지를 몰라서 수업시간에 다루었던 5가지 case에 대해 결과와 key-rotation 결과 두 가지를 제시하도록 하겠다.

B+ tree 출력함수를 만들어서 수행하므로 육안으로 결과를 확인할 수 있도록 LEAF_MAX와 INTERNAL_MAX를 4로 통일하도록 하겠다.

Case 1) Leaf key-rotation (Same parent)

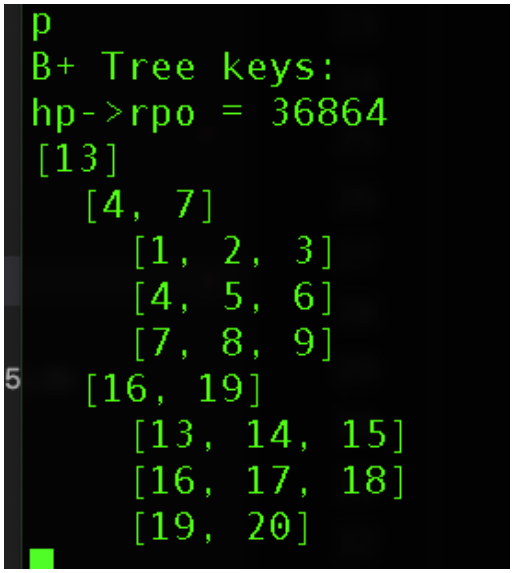


Order가 5인 (LEAF_MAX = INTERNAL_MAX = 4) 인 상황에서 key만 출력한 것이다. 여기서 8과 9를 연달아 insert하면 오른쪽 sibling에 여유공간이 있으므로 key-rotation이 일어난다.



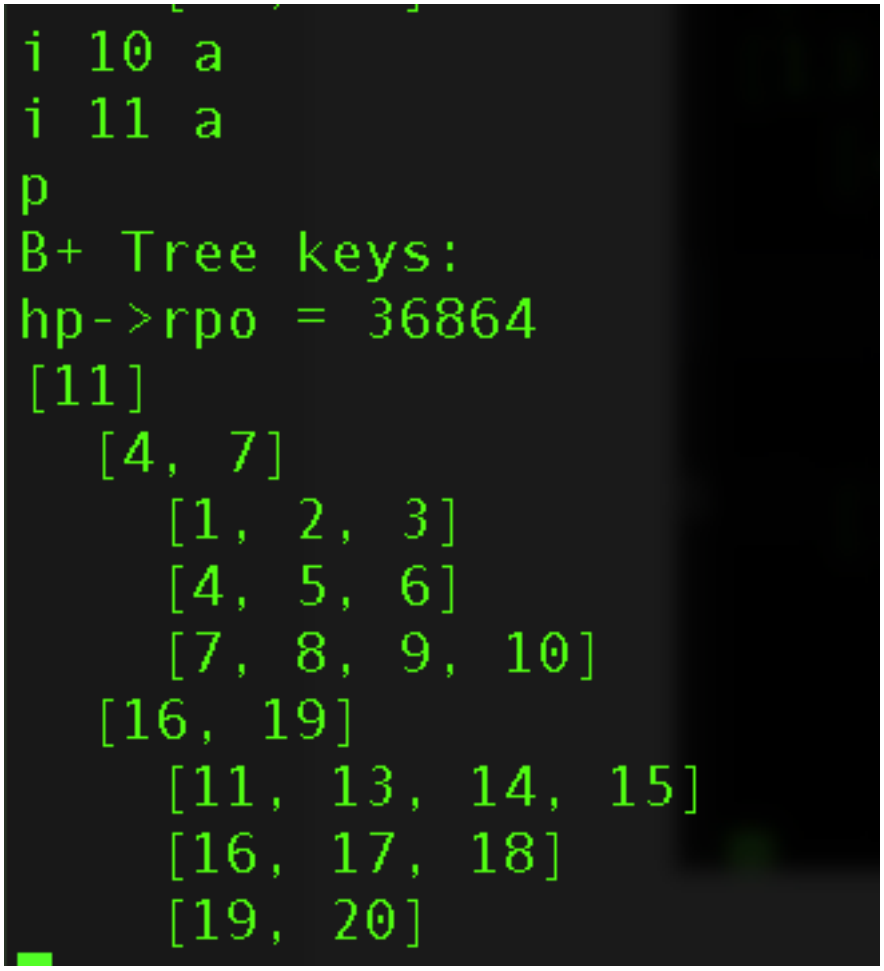
Parent의 key가 10에서 9로 바뀌고 9가 오른쪽 sibling으로 넘어갔음을 확인할 수 있다.

Case 2) key-rotation (Not same parent)



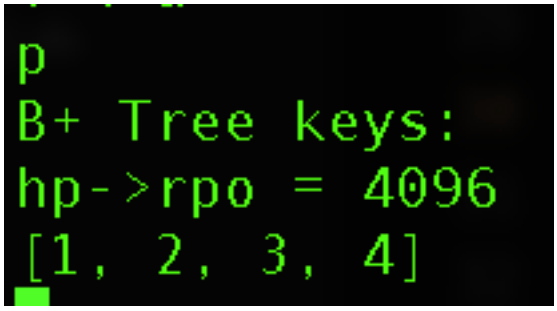
이러한 상황에서 10과 11을 insert하면 오른쪽 sibling으로 key-rotation이 일어나야 된다. 하지만 [7,8,9]와 [13,14,15]는 서로 다른 parent를 가지고 있어 parent의 parent에서 방향지시등을 찾아야 한다. 즉, 13을 교체해줘야 된다.

10과 11을 insert한 결과이다. Parent의 parent에서의 13이 11로 바뀐 것을 볼 수 있고 11이 오른쪽 sibling으로 넘어가 key-rotation이 수행된 것을 볼 수 있다.

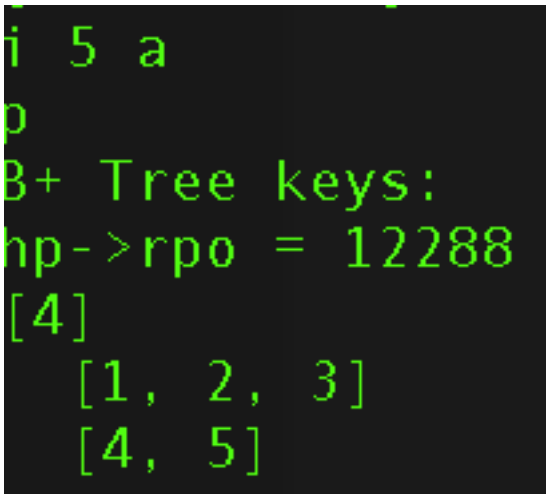


Case 3) Leaf split

Order가 5인 B+ tree에서 1, 2, 3, 4, 5를 연달아 insert하면 split이 일어나야 한다.

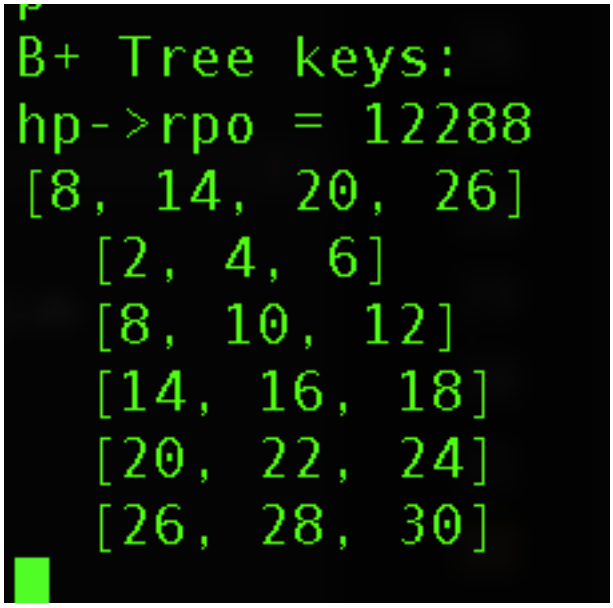


여기서 5를 insert하면 split이 일어난다.

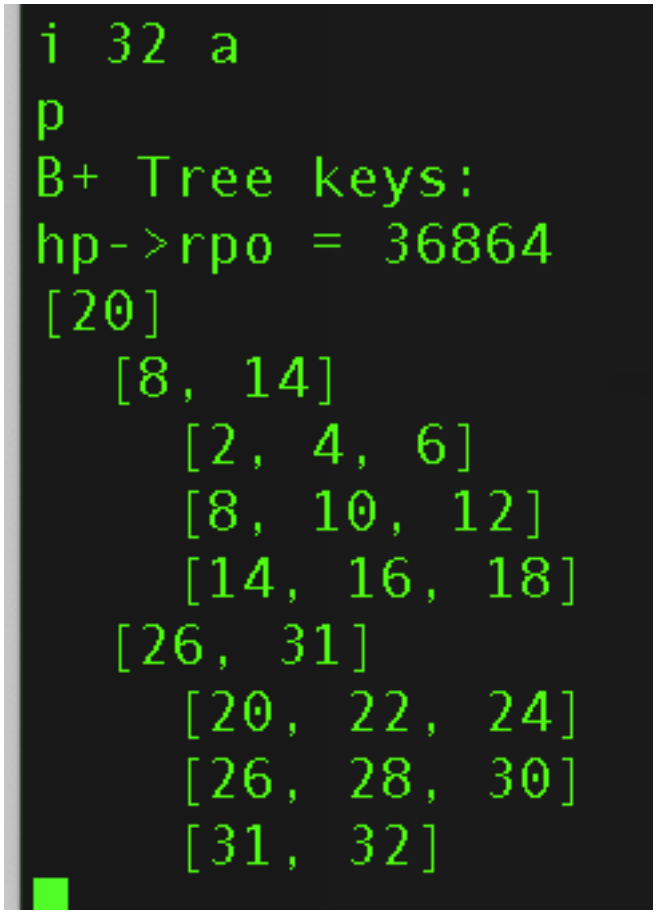


5를 집어넣으니 split이 잘 된 것을 확인할 수 있다.

Case 4) Leaf split, Internal split

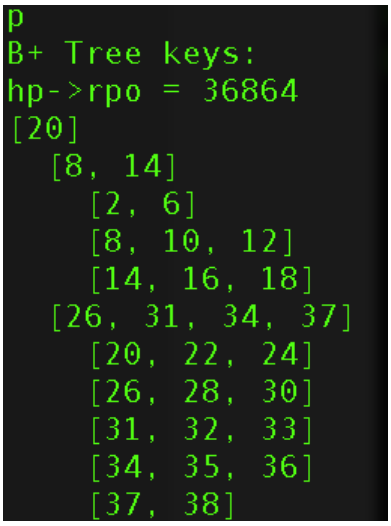


이러한 상황에서 31과 32를 insert하면 leaf에서 split이 일어나고 이에 따라 internal에서도 split이 일어난다.

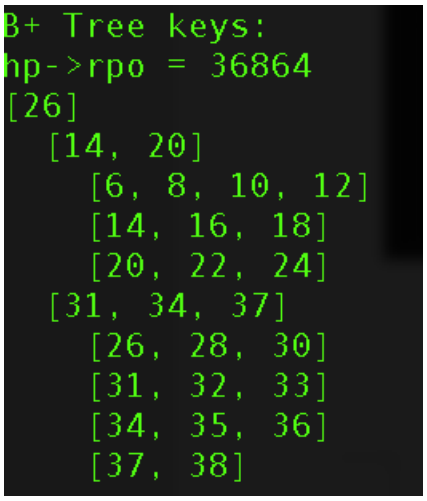


이와 같이 leaf와 internal 모두에서 split이 일어남을 볼 수 있다.

Case 5) Leaf merge, internal redistribute

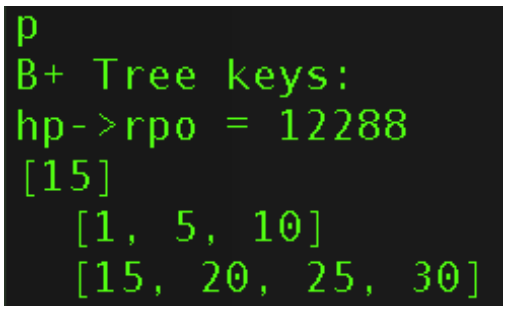


왼쪽과 같은 상황에서 2를 delete하면 6, 8, 10, 12가 merge가 일어날 것이며 이에 따라 internal node에서 8이 삭제가 될 것이고 internal node의 오른쪽 sibling이 꽉 차있기 때문에 merge가 불가능 할 것이기에 redistribute가 일어날 것이다.

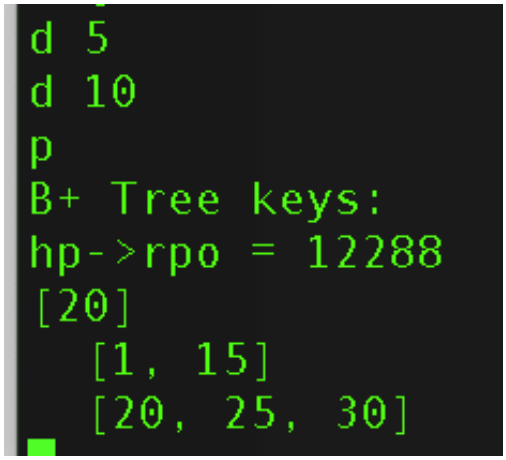


이와 같이 leaf merge와 internal redistribute가 잘 일어났음을 알 수 있다.

Case 6) Leaf redistribute

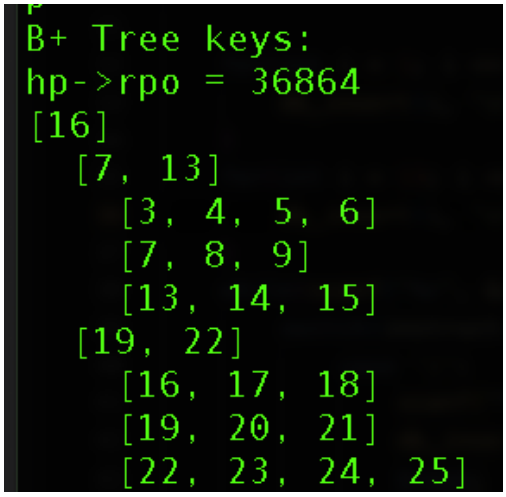


이러한 상황에서 간단하게 5와 10을 delete하면 leaf에서 redistribute가 일어날 것이다. (오른쪽 leaf node가 꽉 차 있어서 merge condition을 만족하지 못하기 때문)

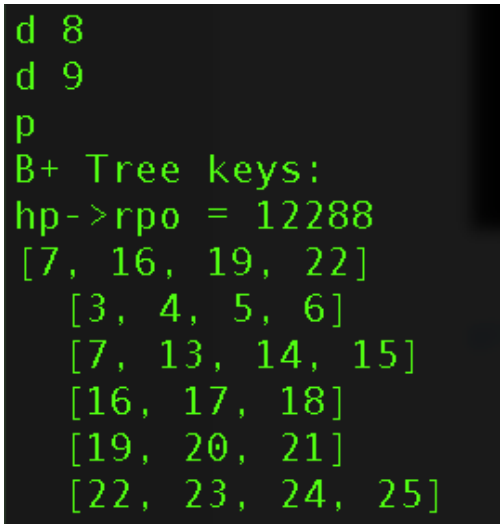


leaf node에서 redistribute가 잘 일어났음을 알 수 있다.

Case 7) Leaf merge, Internal merge



왼쪽과 같은 상황에서 8과 9를 연달아 지우게 되면 leaf node에서 7, 13, 14, 15이 merge를 할 것이고 (왼쪽 sibling node에서 merge가 안되기 때문) 이에 따라 internal node에서 4가 delete되어 internal merge가 일어날 것이다. 결과적으로 root가 제거될 것이다.



왼쪽 그림에서와 같이 8과 9를 연달아 삭제하니 leaf merge와 internal merge가 일어나 root가 삭제된 것을 볼 수 있다.

Case 8) Random insert & delete

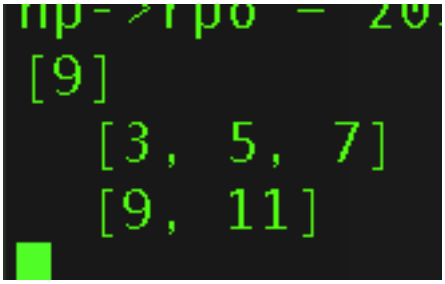
마지막 테스트 케이스는 INTERNAL_MAX = 248, LEAF_MAX = 31로 맞추고 (원래 세팅) 10000개의 element를 랜덤으로 넣었다가 뺄 것이다. 이 과정에서 에러가 나지 않고 결과적으로 tree가 비어있다면 order가 큰 것에대해서도 잘 수행한 것이라 볼 수 있다.

이러한 테스트 케이스를 활용할 것이다.

```
printf("start\n");
for(int i = 1; i <= 3000; i++){
    db_insert(i, "a");
}
for(int i = 3000; i > 2000; i--){
    db_delete(i);
}
for(int i = 5000; i <= 7000; i++){
    db_insert(i, "a");
}
for(int i = 1; i <= 2000; i++){
    db_delete(i);
}
for(int i = 9000; i <= 10000; i++){
    db_insert(i, "a");
}
for(int i = 7001; i < 9000; i++){
    db_insert(i, "a");
}
for(int i = 5000; i <= 10000; i++){
    db_delete(i);
}
printf("Random insertion and deletion completed\n");
```

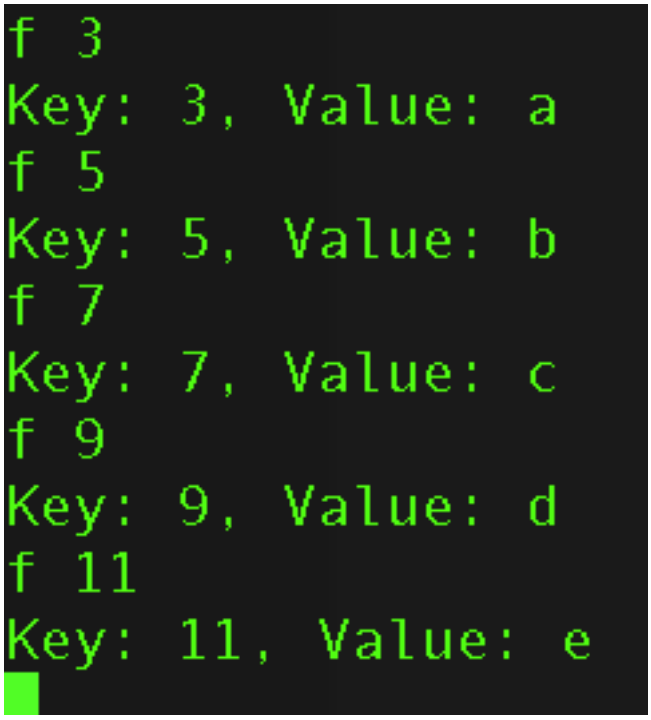
```
start
Random insertion and deletion completed
p
Tree is empty.
```

결과를 보면 모든 insertion & deletion 후 tree가 비었음을 볼 수 있다.



Find함수에 대한 결과는 간단하게 보도록 하겠다.

왼쪽가 같은 상황에서 3, 5, 7, 9, 11에 각각 value a, b, c, d, e가 들어가 있다. 확인해보자



결과는 왼쪽과 같다. Find가 잘 수행됨을 알 수 있다.

Trouble Shooting

본 과제를 수행하면서 다양한 문제를 겪었다. 본 과제에서 매우 까다로웠던 점은 배열의 index이다. Internal node에서 next_offset이 맨 오른쪽 pointer를 뜻하기에 이를 handling하기가 매우 까다로웠다. 또한, 케이스가 너무나도 많이 나눠져 이를 모두 고려하는 것이 너무 어려웠다.

다양한 문제를 겪었지만 대부분의 문제는 무언가를 코드 작성을 빼먹은 경우였다. 때문에 겪었던 문제중 가장 황당했던 문제에 대해 다루겠다. 새로운 root가 생성됐을 때 먼저 원래의 root를 usetofree함수로 초기화하고 header를 output하고 다시 load 해주었다. 하지만 이 과정에서 header가 output되기 전의 값이 계속 담겨있어 문제가 발생하였다. 이를 디버깅하던 중 usetofree 함수 내에서 header를 재로드 하는 과정이 포함되어 있다는 것을 알게 되었다. Header의 값을 업데이트 하기 전 값에 덮어씌워진 것이었다. 때문에 header의 값을 먼저 로드해 주고 원래의 root를 usetofree 함수로 삭제하면서 문제가

해결되었다.

Key-rotation Insert에 대한 고찰

필자는 본 과제를 수행하면서 key-rotation이 insert시 "전체적으로 좋아진다"라는 결론을 내렸다.

일반적으로 이를 말하기 위해서 I/O 횟수를 도입하기로 하겠다.

예를 들어, order = 5인 (앞에 Result에서의 order) 경우를 생각해보자. Depth가 4인 경우, split이 일어날 때 최소 I/O 횟수는 2번이다. (새로운 페이지 접근 (I/O), parent page에 새로운 값 삽입 (I/O)) 또한, key-rotation을 생각했을 때 일어나는 최소 I/O 횟수 역시 2번이다. (sibling node 접근 1번, parent 접근 1번) 때문에 최선의 경우에서는 key-rotation이 별 의미가 사라진다.

반면, 최악의 경우 split은 (2 * depth) + 1번의 I/O가 발생할 수 있다. (각 depth에서 split이 일어나는 경우, 각 depth에서 2번의 I/O가 발생하고 새로운 root가 생성되어 I/O가 한 번 더 발생) 하지만 key-rotation의 경우, 최악의 경우 depth + 1번의 I/O가 발생한다. (leaf node에서 현재 노드 + sibling 접근, 각 층에서 parent만 한 번씩 접근) 때문에 최악의 경우 key-rotation이 더 좋은 성능을 보임을 알 수 있다.

결론적으로, 특정 상황, split이 연쇄적으로 일어나는 상황에서는 key-rotaiton이 split보다 I/O를 덜 발생시켜 더 빠른 insert를 가능하게 한다. 뿐만 아니라, split을 하면 node의 사용률을 많이 확보할 수가 없다. Split을 한 후 split한 node를 추가적으로 채워주지 않으면 그 node는 계속해서 반만 채워진 채로 있을 것이다. 반면, key-rotation의 경우 node의 사용률이 split보다 높다. 일단 split을 안하고 실제 insert를 다른 노드에라도 수행하기 때문에 node의 사용률이 높다. (B tree를 사용하지 않는 이유 역시 depth가 B+ tree보다 월등히 높아져서이다.) **이는 결과적으로 B+ tree의 depth를 줄이는데 파격적인 역할을 한다.** 때문에, key-rotation을 사용하면 split만을 사용할 때보다 전체적으로 성능이 좋아진다.