Assignment3_Wiki

최범규(2022006726)

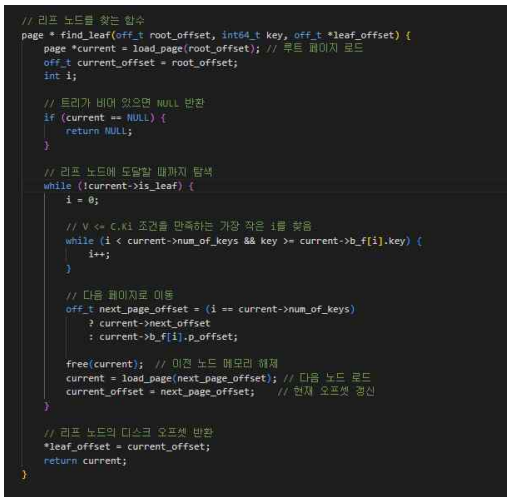
① Design

인자로 주어지는 key와 매칭되는 value를 찾는 find함수를 구현하기 위해 일단 그 key가 위치하는(위치할수 있는) leaf를 찾는 함수가 필요하다고 생각했다.(find_leaf함수) 이 함수를 통해 key가 주어지는 leaf 페이지의 오프셋을 찾게되면 그 오프셋의 페이지를 메모리에 올리고 실제 그 리프 페이지에 찾는 key가 있는지 검사한뒤, 실제로 존재하면 해당 key와 매칭되는 key를 리턴하도록 구현했다. (db_find함수)
인자로 주어지는 key와 value를 insert하는 함수를 구현하기 위해
일반적인 삽입에서는 find_leaf를 통해 key가 삽입될 leaf 노드를 탐색한 뒤, 해당 노드의 key 배열에서 새로운 key와 value를 삽입한다. key 배열이 정렬된 상태를 유지하기 위해 삽입 이후의 배열의 key들을 오른쪽으로 이동시키고, 삽입 위치에 새로운 key를 추가한다. leaf 페이지에 이미 LEAF_MAX이상의 key가 존재하는 경우 새로운 키를 삽입할수 없기 때문에 일반적인 삽입을 할수 없다. 이 경우 key-rotation이나 노드 분할과 같은 추가 처리가 필요하다. key-rotation은 leaf 노드의 가장 큰 key를 오른쪽 sibling 노드로 이동시켜 두 노드의 key 개수를 균형 있게 맞추는 방식으로, 옮겨준 키를 부모 노드의 key로 업데이트해준다.
오른쪽 sibling 노드가 없거나 있더라도 가득 차 있는 경우에는 leaf 노드를 분할하여 새 노드를 생성하고, 키를 절반씩 나누어 담는다. 새로운 노드의 첫 번째 key를 부모노드에 insert해줘야 하므로 부모노드에 key를 insert해주는 함수가 필요할 것이다.
이 함수 (insert_into_parent)는 leaf노드를 split해준 다음 그로 인해 만들어진 leaf노드의 첫 번째 key를 입자로 받게 될텐데, 이 key를 insert해주는 과정에서도 일반적인 상황에서는 그냥 단순히 insert만 해주면 되지만, 이미 부모노드가 가득 차있는 경우(INTERNAL_MAX이상의 키 존재)는 부모노드또한 split을 시켜줘야 한다.(split_internal함수호출) split_internal 함수에서도 새 페이지를 생성하고 가득찬 노드와 insert하는 key를 합친 다음 키를 절반씩 원래 노드와 새 노드에 나눠 담는 과정을 거쳐 split을 하면 되지만 이 경우는 앞선 leaf의 split과는 차이점을 가진다.
leaf노드에서 split이 일어나는 경우는 새롭게 만들어진 leaf노드의 첫 번째 key를 부모노드에 insert해줘야 했지만(수업때 언급한 copy), internal노드에서 split이 일어나느 경우는 새롭게 만들어진 노드의 첫 번째 key자체를 부모노드로 옮겨줘야 한다(copy가 아닌 move). insert가 아닌 move이므로 split_index의 그 다음 key부터 새롭게 만든 노드에 채워야 한다. 앞선 leaf노드에서의 split때나 internal의 split때나 부모노드에 key를 copy또는 move시켜주는 과정에서 부모노드가 만약 없는 경우는 새로운 루트를 생성하는 함수(create_new_root)를 호출

해서 새로운 페이지를 만들고 모든 값을 적절하게 초기화해준 다음 hp->rpo를 새롭게 만들어진 루트 노드의 오프셋으로 업데이트시켜주는 과정이 필요하다.
db_delete 함수는 B+ 트리에서 특정 key를 제거하고, 트리의 균형 조건을 유지하는 역할을 한다. key 삭제로 인해 노드의 key 개수가 최소값을 만족하지 못할 경우, 병합(coalesce) 또는 재분배(redistribute)를 통해 균형을 유지해야 한다.
먼저, 삭제할 key가 포함된 leaf 노드를 찾기 위해 find_leaf를 호출한다. leaf 노드에 도달한 후 key를 검색하여 제거하며, 이후의 key들을 앞으로 이동시켜 빈 공간을 채운다.(remove_entry_from_node함수) 삭제 후에도 leaf 노드가 최소 key 개수를 유지하면 추가 작업 없이 삭제를 완료한다.(일반적인 상황)
삭제로 인해 key개수가 최소 key개수보다 작아진 경우, 병합 또는 재분배를 해야 하는데,
형제노드(일반적인 경우 왼쪽노드를 선택.만약 leaf노드가 맨 왼쪽 노드면 오른쪽 노드 선택)와 leaf 노드 모든 key개수를 합쳐도 최대 key개수(LEAF_MAX)이하일때는 병합을 선택하고(coalesce_nodes함수 호출) 그 외의 경우는 재분배 (redistribute_nodes함수 호출)를 선택한다.
병합의 경우 삭제된 노드의 데이터를 이웃 노드에 추가하고, 이웃 노드의 next_offset을 삭제된 노드의 next_offset으로 업데이트해준다. 수업때 다룬 ppt를 참고하여 왼쪽 노드를 남기고 오른쪽 노드를 없애는 방향으로 구현했다. 만약 이 병합의 과정이 internal노드에서 진행되면, 부모 노드에서 연결 key를 가져와 병합된 노드에 추가하고, 삭제된 노드의 모든 key와 포인터를 이웃 노드로 옮겨준다. 이렇게 병합을 마치고 나면 부모 노드에서 연결 key를 제거해줘야 하고 이 때문에 다시 delete_entry함수를 호출한다. 이후 부모노드도 최소 키 개수 조건을 검사하고 재귀적으로 병합 또는 재분배 작업이 진행될수 있어야 한다.
만약 병합으로 인해 루트노드가 없어지는 경우 원래 루트노드의 왼쪽 child노드였던 노드가 root노드로 올라가며 트리의 높이가 1줄어든다. (수업 ppt의 35p상황)
재분배의 경우, 이웃노드에서 키를 하나 가져와서 부족한 노드에 추가해주는 건데, leaf노드에서와 internal노드에서의 재분배 상황을 구분해 줘야한다.
leaf노드에서의 재분배는 부모 노드의 연결 키를 이웃 노드의 첫 번째 키로 업데이트해줘야하며, internal노드에서의 재분배는 parent의 key로 부족해진 노드를 채우고 parent key에 neighbor의 마지막 키를 옮겨준다. (ppt 33p의 상황)
insert와 delete과정중에 변경된 페이지를 디스크 상에 기록해야될 필요가 있는 경우 pwrite 함수를 이용하여 데이터의 변경사항을 디스크에 반영할 필요가 있다.
새로운 노드를 만들 때 new_page를 통해 freelist에 사용가능한 페이지를 새로운 페이지로 추가해주고, 삭제된 노드의 경우 freelist에 추가해줘야 하므로 usetofree함수를 사용했다.
작업중 루트 노드가 비어 있을 경우, 루트를 조정하여 트리의 구조를 유지한다. 루트가 비어있는 leaf라면 헤더 페이지에서 루트 offset을 초기화하고, internal 노드라면 자식 노드를 승격하여 새로운 루트로 만든다. 이 경우 rpo를 수정해서 디스크에 기록했다.

② Implement

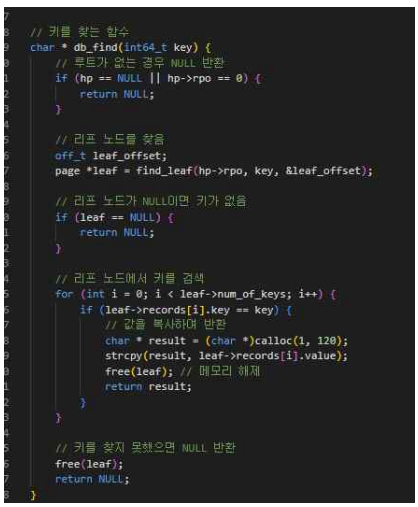
find_leaf함수



```c
// 리프 노드를 찾는 함수
page * find_leaf(off_t root_offset, int64_t key, off_t *leaf_offset) {
    page *current = load_page(root_offset); // 루트 페이지 로드
    off_t current_offset = root_offset;
    int i;

    // 트리가 비어 있으면 NULL 반환
    if (current == NULL) {
        return NULL;
    }

    // 리프 노드에 도달할 때까지 탐색
    while (!current->is_leaf) {
        i = 0;

        // V <= C.Ki 조건을 만족하는 가장 작은 i를 찾음
        while (i < current->num_of_keys && key >= current->b_f[i].key) {
            i++;
        }

        // 다음 페이지로 이동
        off_t next_page_offset = (i == current->num_of_keys)
            ? current->next_offset
            : current->b_f[i].p_offset;

        free(current);  // 이전 노드 메모리 해제
        current = load_page(next_page_offset); // 다음 노드 로드
        current_offset = next_page_offset;    // 현재 오프셋 갱신
    }

    // 리프 노드의 디스크 오프셋 반환
    *leaf_offset = current_offset;
    return current;
}
```

주어진 키를 포함할수 있는(실제 포함할수도 있고 아닐수도 있다) 리프 노드를 찾는 함수로 단순히 key와 매칭되는 value를 찾는 기능 뿐만이 아니라 insert와 delete기능에서도 필요하기에 구현했다. V <= C.Ki 조건을 만족하는 가장 작은 i를 찾는데 만약 못찾아서 num_of_keys까지 온 경우 부모노드의 키들보다 큰 값이기에 next_offset을 다음 offset으로 넣어 준다는 점을 새롭게 구현했다.

db_find함수



```c
// 키를 찾는 함수
char * db_find(int64_t key) {
    // 루트가 없는 경우 NULL 반환
    if (hp == NULL || hp->rpo == 0) {
        return NULL;
    }

    // 리프 노드를 찾음
    off_t leaf_offset;
    page *leaf = find_leaf(hp->rpo, key, &leaf_offset);

    // 리프 노드가 NULL이면 키가 없음
    if (leaf == NULL) {
        return NULL;
    }

    // 리프 노드에서 키를 검색
    for (int i = 0; i < leaf->num_of_keys; i++) {
        if (leaf->records[i].key == key) {
            // 값을 복사하여 반환
            char * result = (char *)calloc(1, 120);
            strcpy(result, leaf->records[i].value);
            free(leaf); // 메모리 해제
            return result;
        }
    }

    // 키를 찾지 못했으면 NULL 반환
    free(leaf);
    return NULL;
}
```

find_leaf를 사용해서 해당 키가 존재할 가능성이 잇는 리프노드를 찾은 후에 리프노드에서 실제로 키가 존재하는지를 검사한다.
키를 찾으면 해당 값을 반환, 찾지 못하면 NULL 반환한다.

insert_into_leaf함수

```c
int insert_into_leaf(page *leaf, int64_t key, const char *value) {
    // 삽입 위치를 찾기 위한 변수
    int i, insertion_point = 0;

    // 삽입 위치 탐색
    while (insertion_point < leaf->num_of_keys && leaf->records[insertion_point].key < key) {
        insertion_point++;
    }

    // 기존 데이터를 오른쪽으로 이동하여 삽입 공간 확보
    for (i = leaf->num_of_keys; i > insertion_point; i--) {
        leaf->records[i] = leaf->records[i - 1];
    }

    // 새로운 키와 값을 삽입
    leaf->records[insertion_point].key = key;
    strncpy(leaf->records[insertion_point].value, value, sizeof(leaf->records[insertion_point].value) - 1);
    leaf->records[insertion_point].value[sizeof(leaf->records[insertion_point].value) - 1] = '\0';

    // 키 개수 증가
    leaf->num_of_keys++;

    return 0; // 성공
}
```

insert를 하는 경우 리프노드가 가득 차지 않았을 때 새로운 키를 삽입하는 함수이다.
키를 삽입할 적절한 위치를 찾아 데이터를 오른쪽으로 밀어 공간 확보하는 과정을 for문으로 구현했다.

key_rotation_insert함수

```c
int key_rotation_insert(int64_t key, char* value, page* leaf, page* sibling, off_t leaf_offset) {
    // Step 1: 임시 배열 생성 및 데이터 복사
    record temp_records[LEAF_MAX + 1]; // 임시 배열 (최대 32개 키 저장 가능)
    int temp_keys = LEAF_MAX;

    // 기존 leaf 노드의 키와 값을 임시 배열에 복사시켜주자. 일단 insert하는 key값은 안들어간다.
    for (int i = 0; i < leaf->num_of_keys; i++) {
        temp_records[i] = leaf->records[i];
    }

    // insert해줘야 하는 키 값을 넣어줘야 하는 위치를 찾는다.
    int insertion_point = 0;
    while (insertion_point < temp_keys && temp_records[insertion_point].key < key) {
        insertion_point++;
    }

    // 찾은 위치 이후의 key들은 한칸씩 뒤로 옮기기
    for (int i = temp_keys; i > insertion_point; i--) {
        temp_records[i] = temp_records[i - 1];
    }

    // insert하는 key값 넣어주고, value도 같이 넣어준다. value는 문자열 배열이어서 안전하게 복사해주기 위해 아래와 같이 복사.
    temp_records[insertion_point].key = key;
    strncpy(temp_records[insertion_point].value, value, sizeof(temp_records[insertion_point].value) - 1);
    temp_records[insertion_point].value[sizeof(temp_records[insertion_point].value) - 1] = '\0';
    temp_keys++;

    // temp_records에서 가장 큰 값을 sibling의 왼쪽으로 옮긴다. sibling의 기존 키들은 모두 한칸씩 오른쪽으로.
    sibling->num_of_keys++;
    for (int i = sibling->num_of_keys - 1; i > 0; i--) {
        sibling->records[i] = sibling->records[i - 1 ];
    }
    sibling->records[0] = temp_records[temp_keys - 1]; // 가장 큰 키를 sibling으로 이동
    temp_keys--; // 임시 배열에서 가장 큰 키 제거

    // 다시 temp_records를 leaf로 옮겨준다.
    leaf->num_of_keys = temp_keys;
    for (int i = 0; i < temp_keys; i++) {
        leaf->records[i] = temp_records[i];
    }
    //printf("key rotation ing"); //debugging
```

리프노드가 가득찼지만 오른쪽 형제노드가 여유가 있는 경우 키를 하나 넘겨줌으로써 최대키 개수 제한을 만족시키게끔 한다. 먼저 키를 임시 배열에 복사하고 새로 삽입할 키도 넣어준다음 가장 큰 키를 오른쪽 sibling노드에 넘겨주고 다시 임시배열을 leaf노드에 업데이트해준다. 오른쪽 sibling의 첫 번째 key가 바뀌었기 때문에 parent의 key를 바뀐 키로 업데이트 해주는 내용도 구현하였다.

db_insert함수

```c
int db_insert(int64_t key, char * value) {
    if (hp == NULL) {
        printf("Error: Header page not initialized.\n");
        return -1;
    }

    // 트리가 비어 있는 경우 새로운 루트를 생성
    if (hp->rpo == 0) {
        // 새로운 루트를 생성
        off_t root_offset = new_page();
        page *root = load_page(root_offset);
        if (root == NULL) {
            printf("Error: Failed to create new root.\n");
            return -1;
        }

        // 루트 노드 초기화
        root->is_leaf = 1; // 루트는 리프 노드로 초기화
        root->num_of_keys = 1;
        root->records[0].key = key; // 키 설정
        strncpy(root->records[0].value, value, sizeof(root->records[0].value) - 1);
        root->records[0].value[sizeof(root->records[0].value) - 1] = '\0'; // 문자열 종료
        root->next_offset = 0; // 리프 노드의 next_offset 초기화
        root->parent_page_offset = 0; // 루트의 부모 없음

        // 헤더 페이지 갱신
        hp->rpo = root_offset;
        if (pwrite(fd, hp, sizeof(H_P), 0) != sizeof(H_P)) {
            printf("Error: Failed to update header page.\n");
            free(root);
            return -1;
        }

        // 루트 노드 디스크에 저장
        if (pwrite(fd, root, sizeof(page), root_offset) != sizeof(page)) {
            printf("Error: Failed to write new root to disk.\n");
            free(root);
            return -1;
        }

        free(root);
        return 0; // 성공적으로 삽입 완료
    }
```

새로운 키를 삽입할 때 가장 먼저 호출되는 함수로써, 트리가 비어있는 경우 새로운 루트를 만들어주고 루트노드 초기화후 헤더페이지 저장한다. 그 외의 경우는 삽입할 leaf노드를 찾은 다음 그 leaf노드의 key개수가 최대개수면 key rotation-insert 또는 split을 해야 하는데 오른쪽 sibling이 key를 더 받을 여유가 있는지 검사후 가능하면 key-rotation-insert를, 그렇지 못한 경우 leaf_split을 진행한다. leaf노드의 key개수가 최대개수 leaf에 key가 삽입가능하므로 insert_into_leaf함수를 호출해 insert를 진행한다.

split_leaf함수

```c
}
int split_leaf(int64_t key, const char *value, page *leaf, off_t leaf_offset) {
    off_t new_leaf_offset = new_page();
    page *new_leaf = load_page(new_leaf_offset);
    if (new_leaf == NULL) {
        printf("Error: Failed to create new leaf node.\n");
        return -1;
    }

    // 초기화
    new_leaf->is_leaf = 1;
    new_leaf->parent_page_offset = leaf->parent_page_offset;
    // 임시 공간에 기존 key와 새로운 키를 모두 저장
    record temp_records[LEAF_MAX + 1];
    int temp_keys = 0;
    for (int i = 0; i < leaf->num_of_keys; i++) {//일단 기존의 키 모두 복붙이 저장
        temp_records[temp_keys++] = leaf->records[i];
    }

    // 새로운 키와 값을 삽입할 위치 탐색
    int insertion_point = 0;
    while (insertion_point < temp_keys && temp_records[insertion_point].key < key) {
        insertion_point++;
    }

    // 새로 넣을 key공간과 값을 위한 빈공간
    for (int i = temp_keys; i > insertion_point; i--) {
        temp_records[i] = temp_records[i - 1];
    }
    //새로운 키와 value를 넣어줌.
    temp_records[insertion_point].key = key;
    strncpy(temp_records[insertion_point].value, value, sizeof(temp_records[insertion_point].value) - 1);
    temp_records[insertion_point].value[sizeof(temp_records[insertion_point].value) - 1] = '\0';
    temp_keys++;

    int split_index = cut(LEAF_MAX); // debugging OK.
    //기존 리프 노드 절반 (왼쪽 절반 유지)
    leaf->num_of_keys = split_index;
    for (int i = 0; i < split_index; i++) {
        leaf->records[i] = temp_records[i];
    }

    // 새 리프 노드 채우기 (오른쪽 절반 이동)
    new_leaf->num_of_keys = temp_keys - split_index;
    for (int j = split_index; j < temp_keys; j++) {
        new_leaf->records[j-split_index] = temp_records[j];
    }

    // 리프 노드 간 연결
    new_leaf->next_offset = leaf->next_offset;
    leaf->next_offset = new_leaf_offset;
```

```c
//부모노드 업데이트 부분 해줘야함.
//부모 노드에 insert하는 함수 구현 필요.
//split해줘야 하는 부모 노드가 root면 새 root만들어줘야함.
int64_t new_key = new_leaf->records[0].key; // 새 노드의 첫 번째 키
if (insert_into_parent(leaf, new_key, new_leaf, leaf_offset, new_leaf_offset) != 0) {
    printf("Error: Failed to insert into parent.\n");
    free(new_leaf);
    return -1;
}

// Step 8: 디스크에 변경 사항 저장
if (pwrite(fd, leaf, sizeof(page), leaf_offset) != sizeof(page) ||
    pwrite(fd, new_leaf, sizeof(page), new_leaf_offset) != sizeof(page)) {
    printf("Error: Failed to write split leaf nodes to disk.\n");
    free(new_leaf);
    return -1;
}

free(new_leaf);
return 0; // 성공
}
```

리프노드가 가득 찼을 때 split을 하는 상황에 호출되는 함수로 새로운 리프노드를 생성하고 절반을 나누어 왼쪽 노드, 오른쪽 노드에 반반씩 나누어 가진다. 오른쪽 노드에 첫 번째 key를 parent노드에 insert를 해줘야 하므로 오른쪽과 같이 insert_into_parent함수를 호출한다.

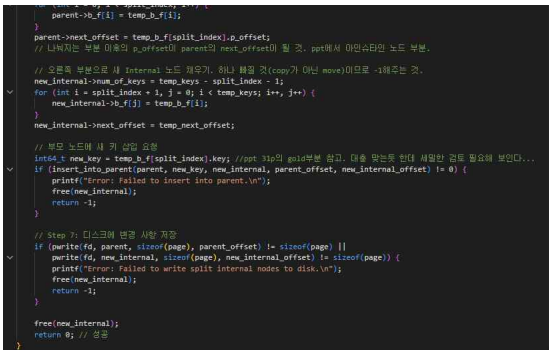
insert_into_parent 함수

```c
int insert_into_parent(page *left, int64_t key, page *right, off_t left_offset, off_t right_offset) {
    off_t parent_offset = left->parent_page_offset;

    // 부모가 없는 경우 새로운 루트 생성
    if (parent_offset == 0) {
        return create_new_root(left, key, right, left_offset, right_offset);
    }

    page *parent = load_page(parent_offset);
    if (parent == NULL) {
        printf("Error: Failed to load parent node.\n");
        return -1;
    }

    //  부모 노드에서 삽입 위치 탐색
    int insertion_point = 0;
    while (insertion_point < parent->num_of_keys &&
           parent->b_f[insertion_point].key < key) {
        insertion_point++;
    }

    //  부모가 가득 찬 경우 처리, 부모노드 split
    if (parent->num_of_keys >= INTERNAL_MAX) {
        int result= split_internal(parent, key, left_offset, right_offset, parent_offset);
        free(parent);
        return result;
    }

    // 부모 노드에 새로운 Key-Offset 삽입
    for (int i = parent->num_of_keys; i > insertion_point; i--) {
        parent->b_f[i] = parent->b_f[i - 1];
    }
    parent->b_f[insertion_point].key = key;
    parent->b_f[insertion_point].p_offset = left_offset;
    // parent노드에 올려준 key와 leaf노드 offset(left_offset)이 한쌍이다.
    // 그 다음 key와 new_leaf노드 offset(right_offset)이 한쌍이다.
    // 새로 올려준 key가 parent노드의 맨 오른쪽에 위치하는 경우 next_offset이 right_offset(new_leaf노드의 offset이 되어야 함)
    if (insertion_point == parent->num_of_keys) {
        parent->next_offset = right_offset;
    } else {
        parent->b_f[insertion_point + 1].p_offset = right_offset;
    }

    parent->num_of_keys++;

    // 부모 노드 디스크에 저장
    if (pwrite(fd, parent, sizeof(page), parent_offset) != sizeof(page)) {
        printf("Error: Failed to write parent node to disk.\n");
        free(parent);
        return -1;
    }
```

부모 노드에 새로운 키를 삽입하는 함수로 leaf노드에서 split이 일어났을 때 호출된다.
부모 노드가 꽉 차있는 경우는 삽입할수 없으므로 split_internal함수를 호출하고 그 외의 경우는 그냥 삽입만 하면 되는데 여기서 조심해야될 부분이 있다. 만약 새로 올려준 key가 parent노드에서 맨 오른쪽에 위치하는 경우 next_offset이 new_leaf노드의 offset이 되어야 한다. 그 외의 경우는 inserttion_point+1 offset이 new_leaf의 offset이 되면 된다.

split_internal함수

```c
//internal page가 split이 필요할때
int split_internal(page *parent, int64_t key, off_t left_offset, off_t right_offset, off_t parent_offset) {
    // Step 1: 새 Internal 페이지 생성
    off_t new_internal_offset = new_page();
    page *new_internal = load_page(new_internal_offset);
    if (new_internal == NULL) {
        printf("Error: Failed to create new internal page.\n");
        return -1;
    }
    new_internal->is_leaf = 0;
    new_internal->parent_page_offset = parent->parent_page_offset;

    // Step 2: 임시 공간에 기존 키, 포인터, 새로운 키 추가
    I_R temp_b_f[INTERNAL_MAX + 1];
    off_t temp_next_offset = parent->next_offset;

    int temp_keys = 0;
    for (int i = 0; i < parent->num_of_keys; i++) {
        temp_b_f[temp_keys++] = parent->b_f[i];
    }

    // 새로운 키와 포인터 삽입
    int insertion_point = 0;
    while (insertion_point < temp_keys && temp_b_f[insertion_point].key < key) {
        insertion_point++;
    }

    for (int i = temp_keys; i > insertion_point; i--) {
        temp_b_f[i] = temp_b_f[i - 1];
    }
    temp_b_f[insertion_point].key = key;
    temp_b_f[insertion_point].p_offset = left_offset;

    // Rightmost Case 처리
    if (insertion_point == temp_keys) {
        temp_next_offset = right_offset;
    } else {
        temp_b_f[insertion_point + 1].p_offset = right_offset;
    }
    temp_keys++;
    //여기까지는 일반적인 부모 노드에 새로운 Key-Offset 삽입하는 코드와 동일하다. 부모 노드가 temp_b_f일뿐.

    // Split 지점 계산
    int split_index = temp_keys / 2;

    // Step 4: 기존 부모 노드 갱신 (왼쪽 절반 유지)
    parent->num_of_keys = split_index;
    for (int i = 0; i < split_index; i++) {
        parent->b_f[i] = temp_b_f[i];
    }
    parent->next_offset = temp_b_f[split_index].p_offset;
```

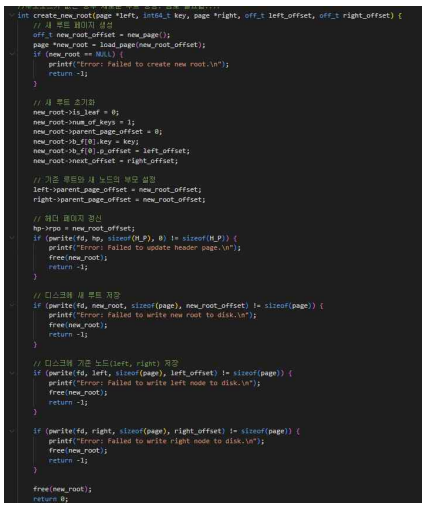


```c
        parent->b_f[i] = temp_b_f[i];
    }
    parent->next_offset = temp_b_f[split_index].p_offset;
    // 나눠지는 부분 이후의 p_offset이 parent의 next_offset이 될 것. ppt에서 아인슈타인 노드 부분.

    // 오른쪽 부분으로 새 Internal 노드 채우기. 하나 빼줄 것(copy가 아닌 move)이므로 -1해주는 것.
    new_internal->num_of_keys = temp_keys - split_index - 1;
    for (int i = split_index + 1, j = 0; i < temp_keys; i++, j++) {
        new_internal->b_f[j] = temp_b_f[i];
    }
    new_internal->next_offset = temp_next_offset;

    // 부모 노드에 새 키 삽입 요청
    int64_t new_key = temp_b_f[split_index].key; //ppt 31p의 gold부분 참고. 대충 맞는듯 한데 새멀한 검토 필요해 보인다...
    if (insert_into_parent(parent, new_key, new_internal, parent_offset, new_internal_offset) != 0) {
        printf("Error: Failed to insert into parent.\n");
        free(new_internal);
        return -1;
    }

    // Step 7: 디스크에 변경 사항 저장
    if (pwrite(fd, parent, sizeof(page), parent_offset) != sizeof(page) ||
        pwrite(fd, new_internal, sizeof(page), new_internal_offset) != sizeof(page)) {
        printf("Error: Failed to write split internal nodes to disk.\n");
        free(new_internal);
        return -1;
    }

    free(new_internal);
    return 0; // 성공
}
```

internal노드가 가득찼을 때 절반으로 나누어 새로운 노드를 생성해준다. leaf에서의 split과 차이점이 있는데 split_internal의 경우에는 부모노드에 새로운 키를 copy해주는 것이 아니라 move하는 것이기 때문에 이 부분을 조심해서 구현했다. 나눠지는 부분 이후의 p_offset이

parent의 next_offset이 되기 때문에
parent->next_offset = temp_b_f[split_index].p_offset;와 같이 처리해주었다.

create_new_root함수



```
int create_new_root(page *left, int64_t key, page *right, off_t left_offset, off_t right_offset) {
    // 새 루트 페이지 생성
    off_t new_root_offset = new_page();
    page *new_root = load_page(new_root_offset);
    if (new_root == NULL) {
        printf("Error: Failed to create new root.\n");
        return -1;
    }

    // 새 루트 초기화
    new_root->is_leaf = 0;
    new_root->num_of_keys = 1;
    new_root->parent_page_offset = 0;
    new_root->b_f[0].key = key;
    new_root->b_f[0].p_offset = left_offset;
    new_root->next_offset = right_offset;

    // 기존 루트와 새 노드의 부모 설정
    left->parent_page_offset = new_root_offset;
    right->parent_page_offset = new_root_offset;

    // 헤더 페이지 갱신
    hp->rpo = new_root_offset;
    if (pwrite(fd, hp, sizeof(H_P), 0) != sizeof(H_P)) {
        printf("Error: Failed to update header page.\n");
        free(new_root);
        return -1;
    }

    // 디스크에 새 루트 저장
    if (pwrite(fd, new_root, sizeof(page), new_root_offset) != sizeof(page)) {
        printf("Error: Failed to write new root to disk.\n");
        free(new_root);
        return -1;
    }

    // 디스크에 기존 노드(left, right) 저장
    if (pwrite(fd, left, sizeof(page), left_offset) != sizeof(page)) {
        printf("Error: Failed to write left node to disk.\n");
        free(new_root);
        return -1;
    }

    if (pwrite(fd, right, sizeof(page), right_offset) != sizeof(page)) {
        printf("Error: Failed to write right node to disk.\n");
        free(new_root);
        return -1;
    }

    free(new_root);
    return 0;
```

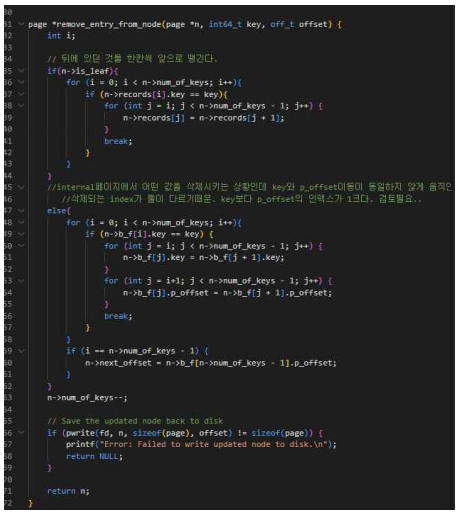
insert_into_parent함수에서 부모에 키를 삽입해야 되는데 부모가 없는 경우에 새로운 루트를 생성해야 하므로 이때 호출되는 함수이다. 새로운 페이지를 할당해서 초기화해주고 헤더페이지를 갱신하는 내용을 넣어 구현하였다. 이 함수를 통해 트리의 높이가 증가한다.

db_delete함수

```
int db_delete(int64_t key) {
    rt = load_page(hp->rpo);
    if (hp == NULL || rt==NULL || rt->num_of_keys == 0) {
        printf("Error: Tree is empty.\n");
        return -1;
    }

    // 삭제할 키가 있는 리프 노드 찾기
    off_t leaf_offset;
    page *leaf = find_leaf(hp->rpo, key, &leaf_offset);

    if (leaf == NULL) {
        printf("Error: Key not found.\n");
        return -1;
    }
    // 리프 노드에서 키를 검색하여 삭제. 찾은 leaf에 key가 없을수도 있기때문에 이렇게 구현.
    for (int i = 0; i < leaf->num_of_keys; i++) {
        if (leaf->records[i].key == key) {
            delete_entry(leaf_offset, key);
            free(leaf);
            return 0;
        }
    }

    free(leaf);
    printf("Error: Key not found in leaf node.\n");
    return -1;
}//fin
```

주어진 키를 트리에서 삭제하는 함수로 main함수에서 호출된다.
삭제할 키가 있는 리프노드를 찾아서 그 리프노드의 오프셋과 함께 키 delete_entry함수의 인자로 넣어준다.

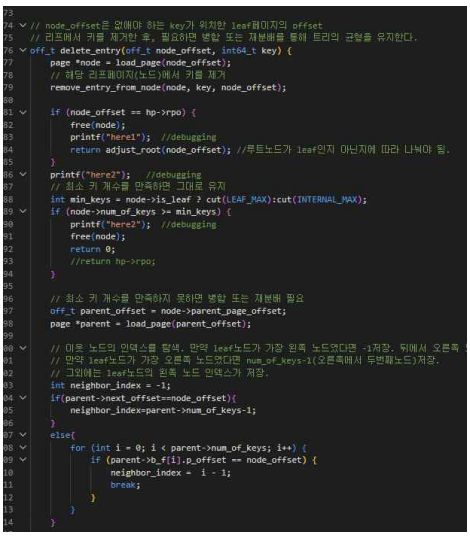
remove_entry_from_node함수



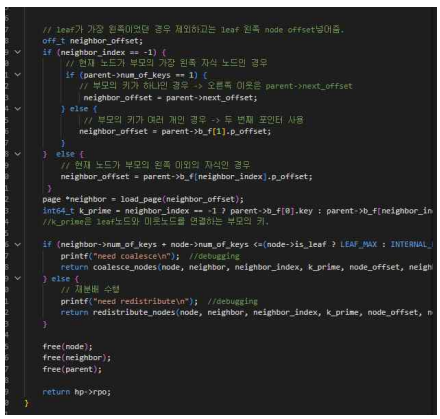
```
page *remove_entry_from_node(page *n, int64_t key, off_t offset) {
    int i;

    // 뒤에 있던 것들 한칸씩 앞으로 땡긴다.
    if(n->is_leaf){
        for (i = 0; i < n->num_of_keys; i++){
            if (n->records[i].key == key){
                for (int j = i; j < n->num_of_keys - 1; j++) {
                    n->records[j] = n->records[j + 1];
                }
                break;
            }
        }
    }
    //internal페이지에서 어떤 값을 삭제시키는 상황인데 key와 p_offset이동이 동일하지 않게 움직인
        //삭제되는 index가 둘이 다르기때문. key보다 p_offset의 인덱스가 1크다. 검토필요..
    else{
        for (i = 0; i < n->num_of_keys; i++){
            if (n->b_f[i].key == key) {
                for (int j = i; j < n->num_of_keys - 1; j++) {
                    n->b_f[j].key = n->b_f[j + 1].key;
                }
                for (int j = i+1; j < n->num_of_keys - 1; j++) {
                    n->b_f[j].p_offset = n->b_f[j + 1].p_offset;
                }
                break;
            }
        }
        if (i == n->num_of_keys - 1) {
            n->next_offset = n->b_f[n->num_of_keys - 1].p_offset;
        }
    }
    n->num_of_keys--;

    // Save the updated node back to disk
    if (pwrite(fd, n, sizeof(page), offset) != sizeof(page)) {
        printf("Error: Failed to write updated node to disk.\n");
        return NULL;
    }

    return n;
}
```

key를 없애고 뒤에 있는 key들을 한칸씩 앞으로 옮겨주는 역할을 한다. internal page에서의 상황(else)의 구현이 어려웠는데, 어떤 값을 삭제시키는 상황에서 key와 p_offset의 쌍이 인덱스가 다르므로 key의 인덱스보다 1큰 인덱스의 p_offset을 없애줘야 한다.

delete_entry함수



```
// node_offset은 없애야 하는 key가 위치한 leaf페이지의 offset
// 리프에서 키를 제거한 후, 필요하면 병합 또는 재분배를 통해 트리의 균형을 유지한다.
off_t delete_entry(off_t node_offset, int64_t key) {
    page *node = load_page(node_offset);
    // 해당 리프페이지(노드)에서 키를 제거
    remove_entry_from_node(node, key, node_offset);

    if (node_offset == hp->rpo) {
        free(node);
        printf("here1");  //debugging
        return adjust_root(node_offset); //루트노드가 leaf인지 아닌지에 따라 나눠야 됨.
    }
    printf("here2");   //debugging
    // 최소 키 개수를 만족하면 그대로 유지
    int min_keys = node->is_leaf ? cut(LEAF_MAX):cut(INTERNAL_MAX);
    if (node->num_of_keys >= min_keys) {
        printf("here2");  //debugging
        free(node);
        return 0;
        //return hp->rpo;
    }

    // 최소 키 개수를 만족하지 못하면 병합 또는 재분배 필요
    off_t parent_offset = node->parent_page_offset;
    page *parent = load_page(parent_offset);

    // 이웃 노드의 인덱스를 탐색. 만약 leaf노드가 가장 왼쪽 노드였다면 -1저장. 뒤에서 오른쪽
    // 만약 leaf노드가 가장 오른쪽 노드였다면 num_of_keys-1(오른쪽에서 두번째노드)저장.
    // 그외에는 leaf노드의 왼쪽 노드 인덱스가 저장.
    int neighbor_index = -1;
    if(parent->next_offset==node_offset){
        neighbor_index=parent->num_of_keys-1;
    }
    else{
        for (int i = 0; i < parent->num_of_keys; i++) {
            if (parent->b_f[i].p_offset == node_offset) {
                neighbor_index =  i - 1;
                break;
            }
        }
    }
```



```
    // leaf가 가장 왼쪽이었던 경우 제외하고는 leaf 왼쪽 node offset방아옴.
    off_t neighbor_offset;
    if (neighbor_index == -1) {
        // 현재 노드가 부모의 가장 왼쪽 자식 노드인 경우
        if (parent->num_of_keys == 1) {
            // 부모의 키가 하나인 경우 -> 오른쪽 이웃은 parent->next_offset
            neighbor_offset = parent->next_offset;
        } else {
            // 부모의 키가 여러 개인 경우 -> 두 번째 포인터 사용
            neighbor_offset = parent->b_f[1].p_offset;
        }
    } else {
        // 현재 노드가 부모의 왼쪽 이외의 자식인 경우
        neighbor_offset = parent->b_f[neighbor_index].p_offset;
    }
    page *neighbor = load_page(neighbor_offset);
    int64_t k_prime = neighbor_index == -1 ? parent->b_f[0].key : parent->b_f[neighbor_in
    //k_prime은 leaf노드와 이웃노드를 연결하는 부모의 키.

    if (neighbor->num_of_keys + node->num_of_keys <=(node->is_leaf ? LEAF_MAX : INTERNAL_
        printf("need coalesce\n");  //debugging
        return coalesce_nodes(node, neighbor, neighbor_index, k_prime, node_offset, neigh
    } else {
        // 재분배 수행
        printf("need redistribute\n");  //debugging
        return redistribute_nodes(node, neighbor, neighbor_index, k_prime, node_offset, n
    }

    free(node);
    free(neighbor);
    free(parent);

    return hp->rpo;
}
```

remove_entry_from_node함수를 통해 key를 노드에서 지우고 그 이후에 해야될 행동을 한다. 그 노드가 root노드이면 따로 해야될 행동이 많으니 adjust_root함수를 호출하여 해결하

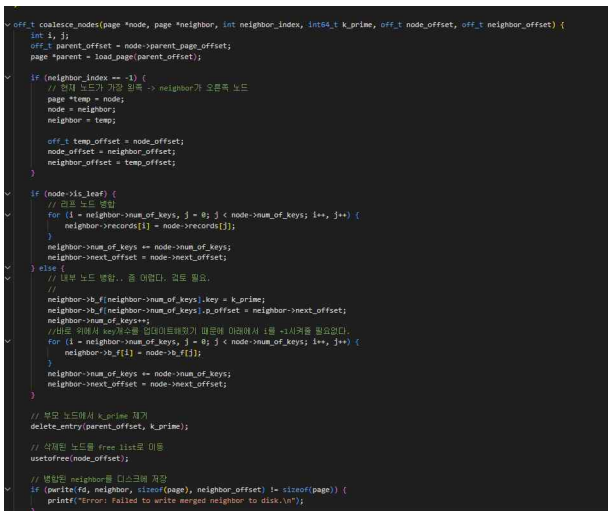
고, 그 외의 경우는 node가 최소키개수보다 적게 있지는 않은지 검사를 한다. 삭제 후에도 최소키개수이상의 키가 존재하면 아무 이상없으니 그대로 종료해주고 그 외의 경우는 합병 또는 재분배를 통해 트리의 균형을 맞춰줘야 한다. 일반적으로는 이웃 노드을 왼쪽 노드로 설정하지만, node_offset의 노드가 맨 왼쪽 노드라면 이웃노드는 오른쪽 노드로 설정한다. 노드와 이웃노드의 키 개수합이 최대키개수이하일 때, 합병이 가능한 상태이므로 coalesce_node함수를 호출해주고, 그 외의 경우는 redistribute_node함수를 호출해준다.
앞선 구현에서 조심해야 될 점은 leaf노드와 이웃노드를 연결하는 부모의 키 k_prime을 지정하는 과정에서 만약 neighbor_index가 -1이었다면 이웃노드가 두 번째노드(리프노드가 맨왼쪽 노드이므로)라는 의미이므로 k_prime의 값을 parent->b_f[0].key의 값으로 지정을 해줘야 한다.

adjust_root함수

```c
//없애준 값이 root페이지에 있었을때 호출.
off_t adjust_root(off_t root_offset) {
    page *root = load_page(root_offset);

    // 루트노드가 텅빈게 아니라면 그냥 냅둔다.
    if (root->num_of_keys > 0) {
        free(root);
        return root_offset;
    }

    off_t new_root_offset = 0;

    // 루트노드가 텅비었는데 그게 leaf노드라면 싹다 초기화.
    if (root->is_leaf) {
        free(rt);
        rt = NULL;
        usetofree(hp->rpo);
        hp->rpo = 0;
        pwrite(fd, hp, sizeof(hp), 0);
        free(hp);
        hp = load_header(0);
        return 0;
    } else {
        // 루트노드는 비었지만 루트노드가 internal노드면 child(왼쪽 child)를 새 root로 올리기
        new_root_offset = root->b_f[0].p_offset;
        page *new_root = load_page(new_root_offset);
        new_root->parent_page_offset = 0;

        if (pwrite(fd, new_root, sizeof(page), new_root_offset) != sizeof(page)) {
            printf("Error: Failed to write new root to disk.\n");
            free(new_root);
            free(root);
            return -1;
        }

        free(new_root);
    }

    hp->rpo = new_root_offset;
    if (pwrite(fd, hp, sizeof(H_P), 0) != sizeof(H_P)) {
        printf("Error: Failed to update header page.\n");
        free(root);
        return -1;
```

delete해줘야 되는 값이 root페이지에 있을 때 호출되는 함수로써 delete_entry함수에서 호출된다.
루트노드가 empty인 것이 아니라면 그냥 냅두면 되지만, 루트 노드가 empty이면서 그 노드가 leaf라면 tree자체가 텅 빈것이므로 이 노드의 오프셋을 프리페이지로 넘겨주고, 모든 값을 초기화시켜준다.
만약 루트 노드는 empty지만 internal노드라면 왼쪽 child노드를 새 root페이지로 올린다.

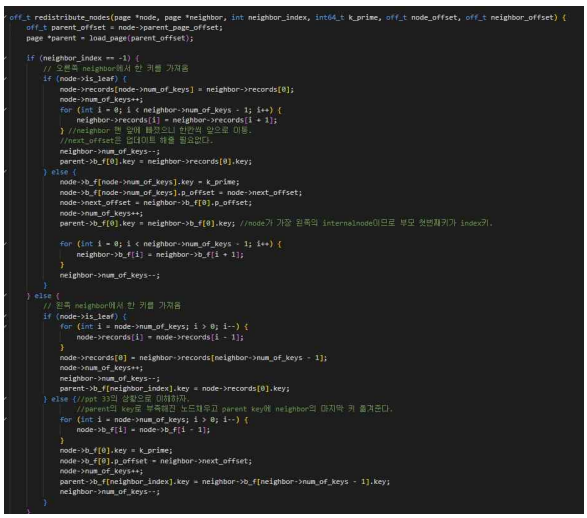
coalesce_nodes함수



```
off_t coalesce_nodes(page *node, page *neighbor, int neighbor_index, int64_t k_prime, off_t node_offset, off_t neighbor_offset) {
    int i, j;
    off_t parent_offset = node->parent_page_offset;
    page *parent = load_page(parent_offset);

    if (neighbor_index == -1) {
        // 현재 노드가 가장 왼쪽 -> neighbor가 오른쪽 노드
        page *temp = node;
        node = neighbor;
        neighbor = temp;

        off_t temp_offset = node_offset;
        node_offset = neighbor_offset;
        neighbor_offset = temp_offset;
    }

    if (node->is_leaf) {
        // 리프 노드 병합
        for (i = neighbor->num_of_keys, j = 0; j < node->num_of_keys; i++, j++) {
            neighbor->records[i] = node->records[j];
        }
        neighbor->num_of_keys += node->num_of_keys;
        neighbor->next_offset = node->next_offset;
    } else {
        // 내부 노드 병합.. 좀 어렵다. 검토 필요.
        //
        neighbor->b_f[neighbor->num_of_keys].key = k_prime;
        neighbor->b_f[neighbor->num_of_keys].p_offset = neighbor->next_offset;
        neighbor->num_of_keys++;
        //바로 위에서 key개수를 업데이트해뒀기 때문에 아래에서 i를 +1시켜줄 필요없다.
        for (i = neighbor->num_of_keys, j = 0; j < node->num_of_keys; i++, j++) {
            neighbor->b_f[i] = node->b_f[j];
        }
        neighbor->num_of_keys += node->num_of_keys;
        neighbor->next_offset = node->next_offset;
    }

    // 부모 노드에서 k_prime 제거
    delete_entry(parent_offset, k_prime);

    // 삭제된 노드를 free list로 이동
    usetofree(node_offset);

    // 병합된 neighbor를 디스크에 저장
    if (pwrite(fd, neighbor, sizeof(page), neighbor_offset) != sizeof(page)) {
        printf("Error: Failed to write merged neighbor to disk.\n");
    }
```

두 노드를 병합시켜주기 때문에 하나의 노드로 모든 키와 오프셋을 옮겨주고 다른 노드는 프리페이지로 넘겨준다. 부모노드에서 두 노드를 연결짓는 key값 k_prime값을 없애야 되므로 delete_entry함수를 호출해서 이 값을 없애준다.

redistribute_nodes함수



```
off_t redistribute_nodes(page *node, page *neighbor, int neighbor_index, int64_t k_prime, off_t node_offset, off_t neighbor_offset) {
    off_t parent_offset = node->parent_page_offset;
    page *parent = load_page(parent_offset);

    if (neighbor_index == -1) {
        // 오른쪽 neighbor에서 한 키를 가져옴
        if (node->is_leaf) {
            node->records[node->num_of_keys] = neighbor->records[0];
            node->num_of_keys++;
            for (int i = 0; i < neighbor->num_of_keys - 1; i++) {
                neighbor->records[i] = neighbor->records[i + 1];
            } //neighbor 첫 인덱 배열이니 한칸씩 앞으로 이동.
            //next_offset은 업데이트 해줄 필요없다.
            neighbor->num_of_keys--;
            parent->b_f[0].key = neighbor->records[0].key;
        } else {
            node->b_f[node->num_of_keys].key = k_prime;
            node->b_f[node->num_of_keys].p_offset = node->next_offset;
            node->next_offset = neighbor->b_f[0].p_offset;
            node->num_of_keys++;
            parent->b_f[0].key = neighbor->b_f[0].key; //node가 가장 왼쪽의 internalnode이므로 부모 첫번째키가 index키.

            for (int i = 0; i < neighbor->num_of_keys - 1; i++) {
                neighbor->b_f[i] = neighbor->b_f[i + 1];
            }
            neighbor->num_of_keys--;
        }
    } else {
        // 왼쪽 neighbor에서 한 키를 가져옴
        if (node->is_leaf) {
            for (int i = node->num_of_keys; i > 0; i--) {
                node->records[i] = node->records[i - 1];
            }
            node->records[0] = neighbor->records[neighbor->num_of_keys - 1];
            node->num_of_keys++;
            neighbor->num_of_keys--;
            parent->b_f[neighbor_index].key = node->records[0].key;
        } else {//ppt 33의 상황으로 이해하자.
            //parent의 key도 부족해진 노드채우고 parent key에 neighbor의 마지막 키 올려준다.
            for (int i = node->num_of_keys; i > 0; i--) {
                node->b_f[i] = node->b_f[i - 1];
            }
            node->b_f[0].key = k_prime;
            node->b_f[0].p_offset = neighbor->next_offset;
            node->num_of_keys++;
            parent->b_f[neighbor_index].key = neighbor->b_f[neighbor->num_of_keys - 1].key;
            neighbor->num_of_keys--;
        }
```

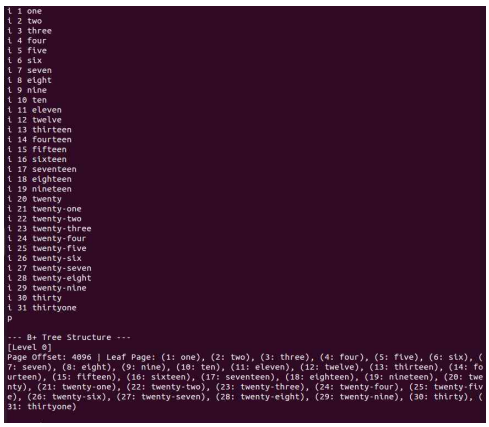
neighbor_index가 -1인 경우 오른쪽 노드가 neighbor노드이므로 neighbor노드의 첫 번째 키를 원래 노드로 옮겨준다. 이에 따른 parent노드의 key도 업데이트가 필요하다. 그 외의 경우는 왼쪽 neighbor에서 한 키를 가져와야 되는데 이 경우 조심해야 되는 부분이 만약 internal노드의 상황이라면 parent의 key로 부족해진 노드를 채우고 parent key에 neighbor의 맨 오른쪽 키를 올려주는 코드를 다음과 같이 구현해줬다.

node->b_f[0].p_offset = neighbor->next_offset;

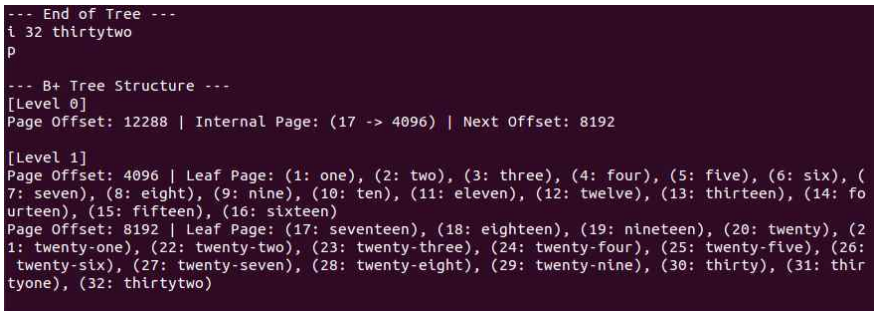
parent->b_f[neighbor_index].key = neighbor->b_f[neighbor->num_of_keys - 1].key;

③ Result

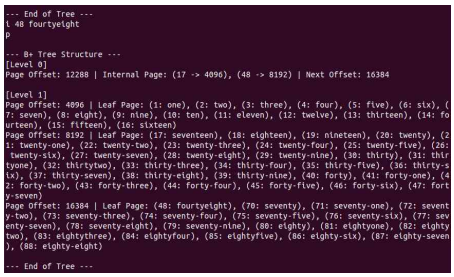
실행결과를 잘 나오는지 보기 위해 INERNAL_MAX를 4로 설정하고 print_tree함수를 만들어서 실행해보았습니다.



1부터 31까지의 값을 insert해줄 때 위와 같이 insert가 정상적으로 이뤄지는걸 볼수 있습니다.

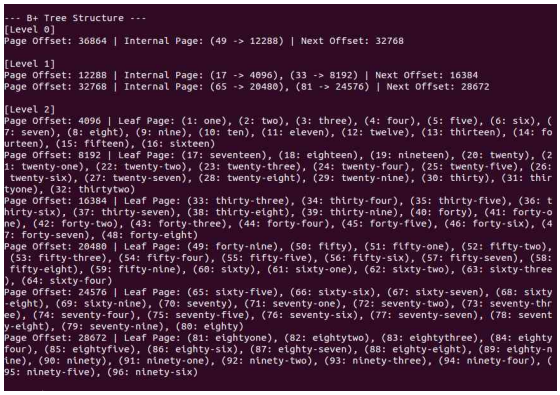


여기서 32를 insert해줬을 때 LEAF_MAX를 초과하므로 split이 일어나야 하고 정상적으로 작동하는 것을 확인 할수 있습니다.

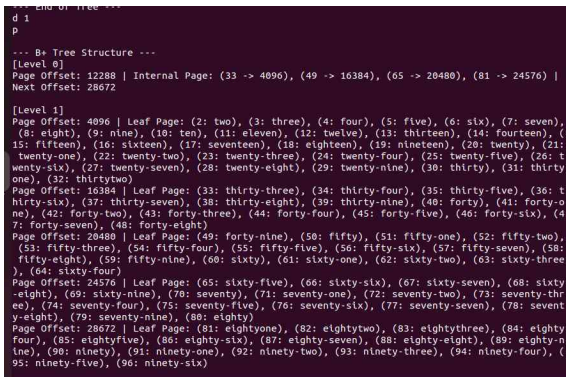


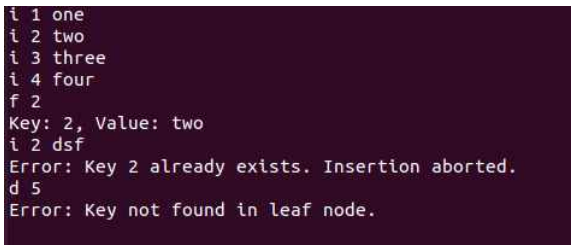
왼쪽과 같이 leaf노드가 꽉찼는데 여기서 그 노드에 48을 삽입할 때 오른쪽 sibling이 여유가 있는 상태에서 key rotation insert를 정상적으로 진행하는 것을 확인할수 있습니다.

delete가 정상적으로 작동하는지 확인하기 위해
1부터 96까지의 key를 insert시켜준 상황에서 1을 delete시켜보았습니다.



위와 같은 상태에서 d 1을 해주니까 leaf에서도 merge가 일어나고 그에 따라서 internal에서도 merge가 정상적으로 일어나는 것을 확인할수 있다.(아래 사진 참고)
루트노드가 사라져서 높이가 1줄어든 것도 정상적으로 작동하는 것을 확인할수 있다.





find 명령어도 정상적으로 작동하는 것을 확인할수 있고, 이미 존재하는 값을 insert시켜줄 때 다음과 같이 insert가 정상적으로 되지 않는 출력문을 띄웠습니다.
존재하지 않는 값을 delete할때는 다음과 같이 key not found 출력문을 보여주었습니다.

④ Trouble shooting

B+ 트리 구현 중, internal 노드에서 키 삭제를 처리하는 과정에서 어려움을 겪었습니다. 특히, remove_entry_from_node 함수에서 키(key)와 포인터(p_offset)의 이동 규칙이 다르게 작동하는 점이 많은 디버깅을 통해 구현한 코드인데 삭제되는 index가 둘이 다른 것을 알아내는게 포인트였습니다. 그 아래 노드에서의 merge로 parent노드에서 delete가 일어나는 경우 삭제시켜야 하는 key값과 같은 인덱스의 p_offset은 합병된 노드를 가리키므로 그 오른쪽 오프셋을 삭제시켜줘야 한다는 점을 찾기가 어려웠습니다.
또한, db_delete 함수에서 특정 키를 삭제한 후 병합(coalesce)이 필요한 상황에서 여러 문제점을 겪었는데 루트 노드가 병합 후 빈 상태가 되었을 때 새로운 루트를 설정하지 않거나 잘못된 루트를 설정해 트리 탐색이 정상적으로 되지않아 결과가 이상하게 뜨는 것을 수없이 확인했습니다. 수 많은 디버깅 과정을 통해 잘못 구현한 부분의 코드를 찾았고 보통 그럴때마다 문제점이 next_offset를 I_R의 p_offset과 구별하지 않고 구현하다 생긴 문제가 대다수 였습니다. next_offset이 맨 오른쪽의 offset이지만 이 값을 특별하게 취급해서 p_offset과 구별해서 코드를 구현해주는 방향으로 수정해주니 이와 관련된 모든 문제를 해결할수 있었습니다.

⑤ Key-rotation에 대한 고찰

Key Rotation Insert는 B+트리의 리프 노드가 최대 키 개수를 초과했을 때, Split 대신 초과된 가장 큰 키를 오른쪽 sibling 노드로 이동시켜 문제를 해결하는 방법이다. Split은 새로운 페이지를 할당하고 부모 노드의 키를 업데이트하는 등의 디스크 I/O가 발생하는데 key rotation insert의 경우 이러한 작업을 하지 않고 단순히 sibling 노드에 데이터 재배치만 해주면 되므로 디스크 I/O 비용을 감소시키는데 큰 역할을 할것으로 생각된다. 또한 이 방법은 split의 횟수를 줄이기 때문에 split으로 인한 부모 노드에 새로운 키가 삽입되는 경우의 수도 줄여 나감으로써 트리 높이가 증가할 가능성을 줄일수 있다고 생각한다. 또한, split이 일어나게 되면 split된 노드의 key개수가 최소키에 가까워짐으로써 한 노드에서 key를 저장하는 개수가 확연히 떨어지게 되므로 디스크를 비효율적으로 쓸 가능성이 높아지는데 key rotation은 split의 횟수을 줄임으로써 디스크를 조금 더 효율적으로 쓸수 있을 것이다.
다만 key rotation insert의 경우 특정노드에만 데이터가 몰리는 가능성이 높아지는 경향이 있다. 꽉차있는 노드 오른쪽의 노드는 key rotation insert로 인해 과하게 데이터가 많이 insert될수 있으며 이는 꽉차있는 노드 근처로 데이터가 몰릴수 있는 가능성이 있다. 이는 데이터 분포에 불균형을 이끌수 있다고 생각되며 결과적으로 find등의 함수에서 성능 악화의 결과를 초래할수 있다고 생각한다.