Assignment 3 Wiki

2020081958 송재휘

1. Design

우선, 명세에 정의된 Header page, Internal page, Free page, Leaf page 에 대한 기본적인 layout 이 제공된 bpt.h 파일에 정의가 되어 있음으로, 해야할 것은 크게 구조 이해하기와 db_find 구현, db_insert 구현, db_delete 구현이었다. In-memory 에서 동작하는 B+tree 에 대한 소스코드를 함께 첨부해주셨기 때문에, 구현하기 전 가장 먼저 한 것은, 제공된 bpt.h 와 bpt.c 에 정의된 함수 및 구조체 등을 이해하고, 해당 내용을 주석으로 남기는 것이었다. 이후, 크게 3 단계로 나눠 차례로 B+ tree 의 find, insert, delete 를 차례로 구현하는 계획을 세웠다. 이때 각 함수를 구현하는 과정에서 제공된 in-memory b+tree 소스코드의 흐름을 적극 활용했다. In-memory b+tree 와 제공된 명세에 따른 disk based b+tree 의 경우 page 구조체의 구조가 약간 달랐다. 특히 order 가 n 이라고 했을 때, n 개의 pointer 와 n-1 개의 key 값으로 구성된 page 와 달리 이번 과제에서는 internal page 의 경우 가장 왼쪽 포인터를 next_offset 필드를 사용하고, leaf_page 의 경우 right sibling 을 가리키기 위해 사용하는 만큼 이를 고려해 적절한 수정이 가장 중요했다. 또한 디스크 기반의 B+tree 인 만큼 디스크에 적절히 write 해주는 것도 중요했다.

우선 db_find 함수를 구현하기 위해서는 key 값이 주어지면 해당 key 값이 존재하는 leaf page 를 우선 찾을 수 있도록 별도의 함수를 만들어야겠다는 디자인 계획을 세웠다. Leaf 에 도달하기 위해선 당연히 루트부터 훑고 내려가야 할 것이고, 이렇게 leaf 에 도달했을 때 해당 leaf 의 disk 내 offset 값을 리턴하도록 계획했다. 이번 과제를 하는데 있어 또 하나 중요하게 생각해야 하는 것은 각 페이지의 fd 로부터의 offset 값이었다. 제공된 메모리 기반 B+tree 에서는 바로 Node 라는 구조체를 가리키도록 pointer 가 설정되어 있었지만, 이번엔 offset 값으로 가리켜야 하는 만큼, 페이지를 곧 바로 넘기기 보다는 되도록이면 offset 값을 사용하도록 함수를 구성했다. 이렇게 leaf 를 찾으면, 우선 leaf 가 있는지 확인하고, 없다면 NULL 을 반환하며 종료시키도록 할 것이다. 만약 있다면, 해당 leaf page 내의 key 값을 차례로 탐색해 인자로 넘어온 key 에 해당하는 records 의 index 를 찾아서 해당 값을 반환해줄 것이다.

두번째로, db_insert 함수의 경우, 일반적인 B+tree 에 key-rotation insert 라는 기능을 추가해야 하는데, 우선 일반적인 B+tree 를 먼저 구현을 완료한 뒤에 key-rotation insert 기능을 추가하겠다는 계획을 세웠다. 이렇게 한 이유는 직접 그 차이를 컴파일을 통해 경험해보고자 했기 때문이다. Db_insert 의 경우에는 B+tree 에 추가할 key 와 value 가 주어지는데, key 값의 중복을 방지하기 위해 앞서 만든 db_find 를 활용해 중복 여부를 확인할 것이다. 이후, key 와 value 를 담은 record 를 하나 생성하고, 만약 root 가 없는 새로운 파일일 경우 root 를 새로 생성해줄 것이다. 물론 이 경우, header page 에 대한 고려가 필요하다. Root 의 생성은 이미 구현된 함수인 start_new_file 을 사용할 것이다. 이후, 만약 tree 가 없는 빈 파일이 아니라면, 똑같이 leaf 의 offset 을 찾고 해당 leaf page 를 읽어서 만약 새로운 record 가 들어갈 자리가 있다면 바로 넣어주고 종료 시키도록 할 것이다. 하지만, 자리가 없는 경우 split 을 해줘야 한다. 이번 명세에서는 자리가 없는 경우 split 대신 key-rotation insert 를 적용해보고 안되면 split 을 적용하도록 한 만큼, 우선 처음에는 바로 split 을 하도록 구현해놓은 뒤 delete 까지 마치고 돌아와, key_rotation 을 구현할 것이다. Key_rotation 의 경우 로직은 우선 right sibling page 를 찾고

해당 페이지에 들어갈 자리가 있는지 확인한다. 만약 없다면 split 을 수행한다. 하지만, 자리가 있다면 기존 leaf 페이지에 새로운 record 를 추가한 뒤 가장 큰 record 를 오른쪽 sibling page 로 넘기게 된다. 이렇게 값을 넘기는 것은 간단한데 문제는 이렇게 넘긴 값을 부모 페이지에 쓰는 것이다. 여기에는 두가지 경우가 존재한다. 첫번째는, 현재 leaf page 와 right sibling page 의 부모 페이지가 같은 경우이다. 이 경우엔, 간단하게 right sibling page 에게 넘긴 첫번째 key 값을 부모 페이지에 기존 key 값에 넣어주면 된다. 하지만, 만약 부모가 다를 경우, tree 를 타고 root 까지 올라가면서 만약 부모 페이지에 그 서로를 가르는 offset 값을 결정하는 그 사이의 key 값이 새로 들어온 값보다 큰 경우, 해당 값을 새로 들어온 key 값으로 교체하도록 할 것이다. 이렇게 돼야, 새로 right sibling page 로 옮겨진 값에 접근이 가능해진다. 물론 이렇게 page 를 수정하게 되면 항상 disk 에 써주는 과정을 포함할 것이다. 이렇게 key rotation 을 구현할 것이고, 만약 이게 안된다면 split 을 구현해야 한다. Split 의 경우 수업에서 배운 대로, 임의의 LEAF_MAX +1 의 배열을 만들어, 새로 추가할 값을 key index 를 고려하여 올바른 순서로 sorting 해 저장한다. 이후, 절반의 사용량을 보장해야하기 때문에 절반은 기존 leaf 페이지에 넣고, 나머지는 새로 생성한 leaf page 에 넣어줄 것이다. 이후 새롭게 생성된 leaf page 를 parent page 에 반영해야 한다. 만약 기존 leaf 가 root 였다면 새로운 root 를 생성해 연결해주도록 할 것이고, 그게 아니라면 만약 부모 페이지에 b_f 에 들어갈 자리가 있다면 그대로 값을 순서에 맞게 넣고 종료 한다. 하지만 그게 아니라면 부모 page 를 split 한다. Split 의 과정은 동일하고, 이후 부모의 부모 페이지에 자리가 있거나 부모의 부모 페이지가 루트 페이지이기 전까지 위 과정을 반복하도록 코드를 짤 것이다.

마지막으로, db_delete 의 경우에는 in-memory B+tree 의 delete 와 동일하다. 다만 앞에서와 동일하게 disk 에 써주는 행위만 추가될 뿐이다. Delete 의 경우 똑같이 우선 해당 key 값이 tree 에 존재하는지 여부를 확인한다. 만약 없다면 delete 를 곧바로 종료한다. 반대로 있다면 정상적인 delete 를 진행하도록 할 것이다. Delete 의 경우 구현에 필요한 로직의 순서는 우선 주어진 key 값을 바탕으로 해당 key 가 저장된 leaf page 의 record 값을 제거한다. 만약 leaf page 가 leaf 이자 root 였다면, 해당 page 가 비었는지 확인한다. 만약 비어 있지 않다면 그대로 종료하면 되지만, 비었다면 해당 페이지를 free 해줘야 한다. 해당 페이지를 비우고, free page list 에 달아둘 것이다. 만약 루트가 아니라면, 우선 확인해야 할 것은 해당 페이지가 절반이상을 사용하고 있는지이다. 만약 절반이상을 사용 중이라면 그대로 delete 를 종료할 것이다. 그렇지 않다면, merge 혹은 redistribute 를 해줘야 하는데 둘의 차이는 merge 의 경우 지워진 leaf 페이지의 옆의 neighbor 페이지를 찾고, 해당 페이지와 합칠 수 있다면 merge 를 수행한다. Merge 의 경우, 말그대로 두 개의 페이지의 값을 합치게 되는데, 만약 target 페이지가 leaf 라면 neighbor page 로 값을 다 옮기고 right sibling page 를 가리키는 값을 기존 값으로 수정해줄 것이다. 이때 neighbor 페이지는 왼쪽에 있다고 일반화 할 것이다. 하지만 만약 leaf 가 아니라면, p_offset 을 고려해 차례로 옆으로 넘겨줄 것이다. 하지만 이때, 둘의 부모 페이지에 저장 돼 둘을 구분하던 key 값을 기존 neighbor 페이지의 마지막 key 값에 추가해준다. 이후 차례로 target_page 의 값을 넘겨줄 것이다. 이렇게 합칠 때 기존에 구분하던 키를 넣어주고 상위 부모의 b_f 에서 삭제해 줄 것이다. 이후 타켓 페이지는 값을 다 neighbor 페이지에 넘겼음으로 비우고, free page list 로 관리할 것이다. 이후 상위 부모에서 삭제해주는 과정이 재귀적으로 일어난다. 마지막으로 redistribute 의 경우는, target page 와 neighbor page 가 합치지 못하는 경우에 발생하는데, neighbor page 에서 하나의 record 를 가져오도록 구성할 것이다. 이때, neighbor 가 타겟 페이지의 왼쪽인지

오른쪽인지를 구분해 값을 가져올 것이다. 하지만 redistribute 의 경우 상위 부모가 서로를 가르는 key 값을 가지고 내려오고 새로운 값을 올려주는 과정을 동반하는 만큼 상위 부모의 서로를 가르는 key 값을 target page 에 추가해주고, 전달할 neighbor 의 키 값을 위로 올리도록 수업에서 배운 내용을 기반으로 구현하겠다는 디자인 계획을 세웠다.

2. Implement

앞에 design 과정에서 계획한대로 db_find, db_insert, db_delete 순서대로 구현을 완료했다. Key-rotation insert 같은 경우에는 db_delete 까지 구현을 완료하고, 구현을 했지만 이번 Wiki 에서는 카테고리 별로 순서대로 설명하고자 한다.

(1) db_find

Db_find 를 구현하는 과정은 insert 와 delete 에 비해 훨씬 간단했다. 우선 특정 key 값이 인자로 주어지면, 해당 key 값에 해당하는 value 를 찾아야 하는데, 이를 위해서는 해당 key 값이 존재하는 leaf page 의 offset 값을 찾아내야 했다. 이로 인해, find_leaf 라는 함수를 새로 구현했다. 해당 함수는 key 값을 인자로 받아. Root page 부터 차례로 훑으며 key 가 존재하는 leaf 를 찾아낸다. 만약 해당 key 가 존재하는 leaf 페이지가 없거나 아예 root 가 없는 상황이라면 0 을 return 한다. 0 을 리턴할 수 있는 이유는 offset 0 에 header page 가 존재하기 때문에, 만약 다른 페이지가 0 이라는 오프셋 값을 갖게 되면 이는 존재하지 않는다고 생각할 수 있다. 만약 있는 경우, leaf 에 다다를 때까지 tree 를 타고 내려간다. Leaf 에 도착한 경우, load_page 에서 디스크에서 읽어와 동적할당 한 페이지를 free 해주고 해당 offset 값을 반환해준다.

```
while (!c->is_leaf) { // search B+tree internal pages
       i = 0;
       while (i < c->num_of_keys) {
          if (key >= c->b_f[i].key) i++;
          else break;
       }
       if (i == 0) { // If finding key is smaller than the left most key value.
          leaf_offset = c->next_offset;
       }
       else {
          leaf_offset = c->b_f[i - 1].p_offset;
       }
       if (c != rt) free(c);
       c = load_page(leaf_offset); // set c to lower page containing key value.
    }
    if (c != rt) free(c);
    return leaf_offset; /
```

이렇게, find_leaf(key)를 통해 leaf page 의 offset 값을 받게 되면, db_find 함수에서는 해당 값이 0 인지 확인한다. 0 이면 존재하지 않는 key 라는 의미이기 때문에, NULL 을 반환한다. 만약 존재한다면 해당 offset 에 해당하는 leaf 값을 load_page 를 통해 읽고, leaf_page->num_of_keys 만큼 반복하며 해당 key 가 존재하는 leaf page 의 index 를 찾는다. 만약 못 찾았다면, NULL 을 반환하고, 찾은 경우 동적할 당을 통해 value 값을 저장 후, 해당 값을 return 해준다. 해당값은 main.c 에서 free 하게 된다.

```
for (i = 0; i < leaf_page->num_of_keys; i++)
       if (leaf_page->records[i].key == key)
```

```
            break; // if key matches

    if (i == leaf_page->num_of_keys) { // key value does not exist in data file
        // printf("Key not in leaf!\n");
        free(leaf_page);
        return NULL;
    }
    else {
        value = (char *)malloc(strlen(leaf_page->records[i].value) + 1);
        if (value != NULL) {
            strcpy(value, leaf_page->records[i].value); // return matching value
        }
        free(leaf_page);
        return value;
    }
```

(2) db_insert

db_insert 의 경우 key 와 value 가 인자로 정해지게 되는데, 우선 key 값이 중복 되면 안되기 때문에, 방금 구현한 db_find 를 사용해 key 의 중복 여부를 확인한다.

```
// key duplication error.
    if (db_find(key) != NULL) {
        printf("Insertion fail! Key duplication error!\n");
        return -1;
    }
```

만약 이 조건을 무사히 통과했다면, key 와 value 를 가지고 leaf 에 넣기 위한 새로운 record 를 생성한다. Record 생성을 위해서는 make_record 라는 함수를 만들었는데, 해당 함수는 단순히 새로운 record 타입의 변수를 생성해 key 와 value 값을 저장해 반환해준다.

```
// Make new record struct and return
record make_record(int64_t key, char * value) {
    record new_rec;
    // copy key
    new_rec.key = key;
    // copy value
    strncpy(new_rec.value, value, sizeof(new_rec.value) - 1);

    new_rec.value[sizeof(new_rec.value) - 1] = '\0'; // set last character as null

    return new_rec;
}
```

이 때, value 에 마지막 character 는 null 로 끝날 수 있도록 처리를 해주었다.

이렇게 record 가 생성되고 나면, 이제 record 를 최하단 페이지의 올바른 위치에 넣어야 하는데, 총 4 가지 경우가 존재한다. 첫번째는 만약, 해당 페이지에 tree 구조가 존재하지 않는 경우이다. DB 파일을 처음 생성한 경우나 생성은 했지만 아무것도 없는 경우에 해당한다. 이 경우 root 자체가 rt 에 정의되어 있지 않기 때문에, 이미 구현되어 있던 start_new_file 에 생성한 record 를 인자로 넘겨 호출한다. 해당 파일에서는 new_page 함수를 통해 새로운 페이지를 생성하고, 해당 페이지를 루트로 만든다. 이를 위해 hp->rpo 를 수정하고, 루트이자 leaf 로서의 사전 준비를

과정을 처리하게 된다. 이미 제공된 함수임으로 생략하겠다. 이렇게 tree 구조가 없는 경우 새로 루트 생성 후 db_insert 를 종료한다. 이때 성공한 경우 0 을 반환 실패한 경우 -1 을 반환한다. 만약 이미 tree 구조가 존재한다면, find_key 함수를 사용해 key 가 존재하는 leaf 의 offset 을 찾는다. 이후 똑같이 해당 offset 이 0 이라는 값을 갖는 경우, -1 을 반환하며 프로그램을 종료한다. 그게 아닌 경우, 총 3 가지 경우가 존재하는데, 가장 간단한 경우는 leaf 에 존재하는 num_of_keys 의 값이 LEAF_MAX 를 넘지 않는 경우이다. 이 경우에는 그냥 leaf 에 새로운 record 를 추가해주면 끝이 난다. 이 과정을 처리하기 위해 다음과 같은 코드를 db_insert 에서 작성했다.

```
// If leaf has room for record, just insert it.
    if (leaf->num_of_keys < LEAF_MAX) {
        insert_into_leaf(leaf_offset, leaf, rec);
        // printf("Insertion successful to leaf / Key: %ld, Value: %s\n", key,
value);
        return 0;
    }
```

이번 과제를 위해 함수를 구현할 때 기존에 주어진 in memory B+tree 에서는 page 를 곧바로 인자로 넘긴 것과 다르게 대부분 offset 값을 넘기도록 구현했는데, 이는 p_offset 에 offset 값이 저장되기 때문이다. 따라서, insert_into_leaf 함수의 동작을 보면, leaf page 의 key 의 개수만큼 순회하며 새로운 record 의 key 값이 들어가 insertion_index 를 찾는다. 찾고 난 뒤에는 해당 index 기준으로 이후의 것들은 오른쪽으로 한칸씩 밀게 된다. 이후 새로 생긴 insertion_index 위치에 record 를 넣어주고 num_of_keys++을 해준다. 이렇게 leaf 를 수정하고 나면, 디스크에 써줘야 한다.

```
// write to disk and reload
    pwrite(fd, leaf, sizeof(page), leaf_offset);
```

위와 같은 방식으로 disk write 을 진행한다. 이렇게 이 함수는 종료되고, 다시 db_insert 로 컨트롤이 넘어오는데, leaf 에 자리가 있는 경우엔 leaf 에 새로운 record 를 넣기만 하고 0 을 반환하며 종료한다. 하지만 leaf 에 충분한 공간이 없어서 넣지 못하는 경우, 원래라면 split 을 진행해야 하지만, 이번 과제의 경우 key_rotation insert 가 가능한지 여부를 확인한다. 이를 확인하기 위해 try_key_rotation_insert 라는 함수를 생성했다.

```
// If key-rotation is possible, do key-rotation.
    if (try_key_rotation_insert(leaf_offset, leaf, rec)) {
        free(leaf);
        return 0;
    }
```

해당 함수에서는 인자로 받은 leaf의 next_offset 값이 0인지 확인한다. 만약 이 값이 0이라면, right sibling page가 없다는 의미이기 때문에 0을 반환하며 함수를 종료한다. 이 함수에서 반환값 0은 실패를, 1은 성공을 의미한다. 그게 아니라면, leaf->next_offset에 해당하는 right sibling page 를 로드한다. Key-rotation insert가 가능하려면 위의 조건 말고도 right sibling page에 새로운 record를 넣을 공간이 있어야하기 때문에 if (right_sibling->num_of_keys < LEAF_MAX)의 조건일 때만 실행한다. 공간이 있다면 right sibling page의 leftmost record에 새로운 record를 넣을 것이기 때문에 기존의 record를 모두 오른쪽으로 한칸 움직인다.

```
// Move records in the right sibling to right by 1 to make space for the new key
```

```
        for (i = right_sibling->num_of_keys; i > 0; i--) {
            right_sibling->records[i] = right_sibling->records[i - 1];
        }
```

이후, leaf 에 새로운 record 가 들어갈 위치를 찾기 위해 index 를 찾아내야 하는데, 다음과 같은 방법으로 insertion_index 를 찾게 된다.

```
// Calculate insertion index of new key value
        insertion_index = 0;
        while (insertion_index < leaf->num_of_keys && leaf-
>records[insertion_index].key < rec.key)
            insertion_index++;
```

이때 두가지 경우가 존재하는데, 만약 insertion_index 가 leaf->num_of_keys 와 같은 경우, 즉 새로운 record 의 key 값이 가장 큰 경우, 해당 record 를 곧바로 right sibling 의 첫번째 record 에 넣어준다.

```
if (insertion_index == leaf->num_of_keys) { // New key value is larger than other
key values in leaf page.
            // Directly move new record to the right sibling page
            right_sibling->records[0] = rec;
            right_sibling->num_of_keys++;
        }
```

하지만, 반대로 새로운 record 가 leaf 의 중간에 들어가는 경우, leaf 의 기존의 가장 마지막 record 를 right sibling page 의 첫번째 record 에 저장하고, leaf 에 기존 record 를 움직여 새로운 공간을 확보한 뒤 새로운 record 를 추가한다.

```
else {
            // Move rightmost record in leaf page to right sibling page
            right_sibling->records[0] = leaf->records[leaf->num_of_keys - 1];
            right_sibling->num_of_keys++;

            // Make a room for new record.
            for(i = insertion_index; i < leaf->num_of_keys - 1; i++) {
                leaf->records[i + 1] = leaf->records[i];
            }

            // Add new record in right place.
            leaf->records[insertion_index] = rec;
        }
```

이렇게 되면 leaf 로의 insert 는 끝이 나고, parent 만 고려해주면 되는데, 이 경우도 두 가지 경우로 나눌 수 있다. 만약 leaf 와 right sibling 의 부모 페이지가 같은 경우, 단순히 right sibling 의 첫번째 record 의 key 값으로 기존 둘을 가르던 key 값을 update 해주면 된다. 물론 페이지 값이 변하면 항상 disk write 을 수행해준다.

```
// Update the parent key if necessary
        if (leaf->parent_page_offset == right_sibling->parent_page_offset) { // If
they have same parent page. Just consider direct parent page.
            parent = load_page(leaf->parent_page_offset);
            // Update key value between target leaf and right sibling
            for (i = 0; i < parent->num_of_keys; i++) {
                if (parent->b_f[i].p_offset == leaf->next_offset) {
                    parent->b_f[i].key = right_sibling->records[0].key;
```

```
                break;
            }
        }
        // Write changes to disk
        pwrite(fd, parent, sizeof(page), leaf->parent_page_offset);
        free(parent);
    }
```

반대로, 부모가 다른 경우, root 까지 tree 를 타고 올라가 새로 들어온 record 의 key 에 대해 조건을 위반하는 key 값이 상위 parent page 에 있는 경우 해당 key 값을 새로 들어온 값으로 교체해주도록 했다. 이때, 만약 하위 페이지로의 offset 값이 부모 페이지의 next_offset 필드에 있다면 수정해야 할 key 값이 없기 때문에 곧바로 상위 부모로 다시 이동하도록 수정했다. Tree 를 타고 상위로 올라가는 과정은 lower_page_offset 과 parent_offset 을 주기적으로 update 를 해주는 것을 통해 구현했다.

```
else { // If they have different parent page, then need to search every parent
pages to root.
        lower_page_offset = leaf->next_offset;
        parent_offset = right_sibling->parent_page_offset;
        while (parent_offset != hp->rpo) { // until it reaches to root
            parent = load_page(parent_offset);

            // If the pointer to right sibling page is leftmost pointer of parent
page. (Do not have to consider.)
            if (lower_page_offset == parent->next_offset) {
                lower_page_offset = parent_offset;
                parent_offset = parent->parent_page_offset; // search parent's
parent
                free(parent);
                continue;
            }

            for (i = 0; i < parent->num_of_keys; i++) {
                if (parent->b_f[i].p_offset == lower_page_offset) {
                    if (parent->b_f[i].key > right_sibling->records[0].key) {
                        parent->b_f[i].key = right_sibling->records[0].key; //
Update upper parent's key value with right sibling's first key value
                    }
                    break;
                }
            }

            // Write changes to disk
            pwrite(fd, parent, sizeof(page), parent_offset);

            // Update parent and lower page offset. (Climb the tree)
            lower_page_offset = parent_offset;
            parent_offset = parent->parent_page_offset;
            free(parent);
        }
    }
```

이렇게 되면, 동적할당한 변수를 free 해주고, 디스크에 변경 내역을 쓴 뒤 1 을 반환하며 함수를 종료한다. 이후엔 db_insert 로 컨트롤이 넘어오게 되고, 반환 값이 1 인 경우 0 을 리턴하며 함수를 종료한다. 하지만 key_rotation insert 적용이 불가능하다면, 원래대로 split 을 수행한다. Split 은 insert_into_leaf_after_splitting 함수를 사용해 수행된다.

```
// Else, split and insert.
    insert_into_leaf_after_splitting(leaf_offset, leaf, rec);
    free(leaf);
    // printf("Insertion successful with splitting / Key: %ld, Value: %s\n", key,
value);
    return 0;
```

이 함수에선 우선 leaf 의 split 하게 되는데, split 하기 위해선 LEAF_MAX +1 크기의 record 배열을 생성하고, 순서 관계를 고려해 새로운 record 가 들어갈 insertion_index 를 찾는다. 이후 해당 index 를 비우기 위해서 오른쪽 레코드를 한칸씩 오른쪽으로 미뤄서 새로운 배열에 복사하고, 새로운 레코드를 insertion_index 에 추가한다.

```
// Allocate temporary memory for split.
    record temp_records[LEAF_MAX + 1];

    insertion_index = 0;
    // Find the matching insertion point.
    while (insertion_index < LEAF_MAX && leaf->records[insertion_index].key <
rec.key)
        insertion_index++;

    for (i = 0, j = 0; i < leaf->num_of_keys; i++, j++) {
        if (j == insertion_index) j++;
        temp_records[j] = leaf->records[i]; // save original leaf items and make
insertion space empty.
    }

    temp_records[insertion_index] = rec; // allocate new record to insertion index
```

이렇게 하고 나면 이제 기본적인 할당량을 충족하기 위해 cut 함수를 사용해 반에 해당하는 수량을 계산하고, 반만큼 기존 leaf page 에 나머지는 새로운 leaf page 에 넣는다. 물론 이때, num_of_key 값을 점진적으로 하나씩 올려준다. 이후, 새로운 leaf page 가 생겼기 때문에 새로운 l eaf page 의 right sibling page 를 가리키는 next_offest 필드의 값을 기존 leaf->next_offset 으로 업데이트 하고, leaf->next_offset 을 새로 생성한 leaf_page 의 offeset 으로 업데이트한다. 또한 새로 생성한 leaf page 의 부모 page offset 을 기존 leaf 와 동일하게 설정한다.

```
// Make new leaf page
    new_offset = new_page();
    new_leaf = load_page(new_offset);
    new_leaf->is_leaf = 1;

    leaf->num_of_keys = 0;
```

```
    // Calculate ceil of n/2
    split = cut(LEAF_MAX);

    // Save half (= ceil of n/2) records into original leaf node
    for (i = 0; i < split; i++) {
        leaf->records[i] = temp_records[i];
        leaf->num_of_keys++;
    }

    // Save remaining records into new leaf node
    for (i = split, j = 0; i < LEAF_MAX + 1; i++, j++) {
        new_leaf->records[j] = temp_records[i];
        new_leaf->num_of_keys++;
    }

    // Link using right sibling page offset [120-127] in page header.
    new_leaf->next_offset = leaf->next_offset;
    leaf->next_offset = new_offset;

    // Update parent page offset
    new_leaf->parent_page_offset = leaf->parent_page_offset;
```

이후 변경 사항을 pwrite 를 통해 disk 에 업데이트 해주게 된다. 이렇게 되면 leaf page 의 split 과 insert 는 완료가 된다. 하지만, 변경 내역을 parent page 에 반영해줘야 하기 때문에 새롭게 생성한 leaf 의 첫번째 레코드의 key 값을 new_key 로 하여 insert_into_parent 라는 함수를 호출한다.

```
// Write to disk
    pwrite(fd, leaf, sizeof(page), leaf_offset);
    pwrite(fd, new_leaf, sizeof(page), new_offset);

    // key for updating parent page
    new_key = new_leaf->records[0].key;

    insert_into_parent(leaf_offset, leaf, new_key, new_offset, new_leaf);

    free(new_leaf);
```

해당 함수에서는 기존 leaf 와 새로운 leaf 를 구분하기 위해 new_key 를 사용해 해당 key 를 기준으로 둘을 구분하는 b_f 를 추가한다. Insert_into_parent 함수에서는 우선 해당 leaf 의 parent page 를 load 하고, 만약 parent_offset 이 없을 경우, 즉, record 를 넣은 leaf 가 leaf 이자 root 였다면 새로운 root page 를 생성하기 위해 insert_into_new_root 라는 함수를 실행한다. 해당 함수에서는 start_new_file 함수와 비슷하게 새로운 루트 페이지를 rt 에 로드하고, hp->rpo 를 해당 root 의 offset 으로 업데이트 한다. 이후, hp 를 로드한다. 또한 새로운 rt 에 새로운 key 값을 추가해줘야 함으로 num_of_keys++을 해주고, next_offset 에 기존 leaf 의 offset 을, b_f[0].key 에 위에서 넘긴 새로운 key 값을 b_f[0].p_offset 에 새로 생성한 leaf 의 offset 을 저장해주고, 각 leaf page 의 부모를 새로운 루트 페이지로 설정 후 disk 에 업데이트 해주며 종료한다.

```
// Make new root page and update
```

```
void insert_into_new_root(off_t left_offset, page * left, int64_t key, off_t
right_offset, page * right) {
    page * root;
    off_t ro;

    // Make new root page and write in header page root page offset field.
    ro = new_page();
    rt = load_page(ro);
    hp->rpo = ro;
    pwrite(fd, hp, sizeof(H_P), 0);
    free(hp);
    hp = load_header(0);

    // Set initial root page attributes.
    rt->num_of_keys++;
    rt->next_offset = left_offset;
    rt->b_f[0].key = key;
    rt->b_f[0].p_offset = right_offset;
    // rt->parent_page_offset = 0;

    // write the new root page into disk
    pwrite(fd, rt, sizeof(page), hp->rpo);
    // free(rt);
    // rt = load_page(hp->rpo);

    // Change leaf parent page offset with new root page.
    left->parent_page_offset = ro;
    right->parent_page_offset = ro;

    // write updated child pages into disk
    pwrite(fd, left, sizeof(page), left_offset);
    pwrite(fd, right, sizeof(page), right_offset);
}
```

이렇게 새로운 루트를 생성한 경우, 위의 처리를 해주고 db_insert 를 종료한다. 하지만 만약 이미 parent page 가 있는 경우 두가지로 나뉘는데, parent page 에 새로운 internal record 를 추가할 공간이 있는 경우와 없는 경우이다. 우선, 이 두 경우모두 넣을 index 를 찾아야하기 때문에, get_left_index 라는 함수를 통해 부모 페이지에서 left leaf 를 가리키는 포인터의 index 를 계산한다. 만약 맨쪽의 next_offest 에 있는 경우 -1 을 이외의 경우는 b_f 의 index 를 리턴한다.

```
int get_left_index(page * parent, off_t left_offset) {
    int left_idx = 0;

    // If left offset is in next_offset field
    if (parent->next_offset == left_offset)
        return -1;

    // Search while matching offset is found.
    while (left_idx < parent->num_of_keys && parent->b_f[left_idx].p_offset !=
left_offset)
```

```
        left_idx++;

    return left_idx;
}
```

이후 추가할 공간이 있는 경우, insert_into_node 함수를 실행한다.

```
// Simple case: the new key fits into parent page.
    if (parent->num_of_keys < INTERNAL_MAX) {
        // printf("Insert into node\n");
        insert_into_node(parent_offset, left_index, key, right_offset);
        return;
    }
```

해당 함수에서는 leaf 에 공간이 있어서 넣는 것과 비슷한 구조를 띈다. 하지만 이젠 부모 페이지에 넣어야 하기 때문에, 우선 부모 페이지를 로드한다. 그리고 위에서 계산한 left leaf 를 가리키는 parent page 의 offset index 를 사용해 해당 index 이후의 internal record 를 한칸씩 오른쪽으로 움직인다.

```
if (left_index == -1) { // left page is in next_offset field
        for (i = parent->num_of_keys; i > 0; i--) {
            parent->b_f[i] = parent->b_f[i - 1];
        }
    }
    else {
        // move parent page's internal record to right by 1
        for (i = parent->num_of_keys - 1; i > left_index; i--)
            parent->b_f[i + 1] = parent->b_f[i];
    }
```

이후, 만들어진 left_index + 1 의 위치에 새로운 key 와 오른쪽 새로 생성한 leaf 의 offset 을 넣어준다. 이후 동일하게 디스크에 부모 페이지를 write 하고, rt 를 다시 업데이트 해준 뒤 함수를 종료한다.

```
// add new internal records to target position
    parent->b_f[left_index + 1].key = key;
    parent->b_f[left_index + 1].p_offset = right_offset;
    parent->num_of_keys++;

    // write parent page into disk
    pwrite(fd, parent, sizeof(page), parent_offset);
    free(parent);

    // reload rt
    free(rt);
    rt = load_page(hp->rpo);
```

이렇게 되면 이제 남은 케이스는 부모 페이지에 새로운 internal record 를 넣지 못하는 상황인데 이경우, 부모 페이지의 split 을 추가로 진행해야 한다. 이 경우를 처리하기 위해 insert_into_node_after_splitting 이라는 함수를 구현했고, 이를 호출하게 된다.

```
// Hard case: split parent page again and insert new internal record.
    // printf("insert into node after splitting!\n");
    insert_into_node_after_splitting(parent_offset, left_index, key, right_offset);
```

이 경우도, leaf page splitting 과 비슷한데, INTERNAL_MAX + 1 만큼 I_R 타입의 배열을 생성한다. 이후, 부모 페이지의 internal 레코드와 새로 추가할 internal record 를 순서에 맞게 sorting 해 추가해준다. 이 과정도 마찬가지로, 이미 계산한 left_index 를 사용해 left_index + 1 의 위치는 비워놓도록 코드를 구성했다. 이렇게 INTERNAL_MAX + 1 개의 internal record 를 복사해서 temp_b_f 배열에 복사해 붙여 놓았고, 이를 사용해 cut 을 통해 반은 기존 parent page 에 넣고, 나머지는 새로운 parent page 를 생성해 복사해둔다. 이 경우, p_offset 이 하나씩 더 존재하기 때문에 중간의 split 에 해당하는 key 값은 k_prime 에 따로 빼둔다. 추가로, parent_page_offset 도 맞춰주는 과정이 필요하다.

```
// Make a room for new internal record and order records in temporary b_f.
    for (i = 0, j = 0; i < old_page->num_of_keys; i++, j++) {
        if (j == left_index + 1) j++;
        temp_b_f[j] = old_page->b_f[i];
    }

    // Insert new internal record
    temp_b_f[left_index + 1].key = key;
    temp_b_f[left_index + 1].p_offset = right_offset;

    split = cut(INTERNAL_MAX + 1); // minimum pointers
    // Create new internal page
    n_page_offset = new_page();
    n_page = load_page(n_page_offset);

    old_page->num_of_keys = 0;
    // save first half of b_f in old page
    for (i = 0; i < split; i++) {
        old_page->b_f[i] = temp_b_f[i];
        old_page->num_of_keys++;
    }

    // Update first offset of new page and save the value of key in the middle to
pass to upper page.
    n_page->next_offset = temp_b_f[i].p_offset;
    k_prime = temp_b_f[split].key;

    // save remaining half into new page
    for (++i, j = 0; i < INTERNAL_MAX + 1; i++, j++) {
        n_page->b_f[j] = temp_b_f[i];
        n_page->num_of_keys++;
    }

    // Update parent page offset value of new internal page and its descendants.
    n_page->parent_page_offset = old_page->parent_page_offset;
```

이후 새로운 parent 로 옮긴 p_offset 값에 해당하는 leaf page 들의 parent_page_offset 값을 새로운 parent page 의 offset 으로 업데이트 해주고, 하나하나 디스크에 적어주도록 했다.

```
// Update parent page offset value of new internal page and its descendants.
```

```
    n_page->parent_page_offset = old_page->parent_page_offset;

    child = load_page(n_page->next_offset);
    child->parent_page_offset = n_page_offset;
    pwrite(fd, child, sizeof(page), n_page->next_offset);
    free(child);

    for (i = 0; i < n_page->num_of_keys; i++) {
        child = load_page(n_page->b_f[i].p_offset);
        child->parent_page_offset = n_page_offset;
        pwrite(fd, child, sizeof(page), n_page->b_f[i].p_offset);
        free(child);
    }

    // write old and new page to disk
    pwrite(fd, old_page, sizeof(page), parent_offset);
    pwrite(fd, n_page, sizeof(page), n_page_offset);
```

이렇게 되면 부모 페이지 split 은 완료했는데, 이 경우에도 부모 페이지의 부모에 아까 빼둔 k_prime 값으로의 업데이트가 필요하기 때문에 다시 insert_into_parent 를 새로운 left, right page 와 새로운 key 값을 사용해 호출하게 된다. 이렇게 재귀적으로 처리가 진행되고, insert_into_parent 의 종료 조건이었던 root 가 새로 생성되거나, 부모에 새로운 internal recor 를 넣을 공간이 있는 경우 db_insert 자체가 종료되도록 구현했다.

(3) Db_delete

Delete 를 하기 위해선 우선 인자로 넘어온 Key 에 해당하는 leaf page 를 찾아야 하기 때문에 앞에서 구현한 find leaf 함수를 사용해 leaf_offset 을 찾도록 했다. 이후 만약 해당 페이지의 offest 값이 0 인 경우, 해당 페이지는 존재하지 않음을 의미하기 때문에 -1 을 반환하며 함수를 종료하도록 구현했다. 반대로 있는 경우, 해당 페이지를 로드하고 해당 leaf 페이지의 records 를 돌며 해당 key 값이 존재하는 record 의 인덱스를 찾아낸다. 그렇게 찾아낸 index i 가 leaf_page->num_of_keys 와 같은 경우, 해당 key 가 존재하지 않음을 의미하기에 -1 을 반환하며 함수를 종료한다. 이 경우도 아니라면, 해당 key 값은 leaf 페이지 내부의 특정 인덱스의 레코드에 존재한다는 의미이기 때문에 delete_entry 라는 함수에 leaf_offset 과, key 값, 그리고 초기 pointer 0 값을 넘겨주며 실행시키도록 했다.

```
// find leaf page containing key value.
    leaf_offset = find_leaf(key);
    if (leaf_offset == 0) { // Key not found
        return -1;
    }
    leaf_page = load_page(leaf_offset);

    for (i = 0; i < leaf_page->num_of_keys; i++) {
        if (leaf_page->records[i].key == key)
            break;
    }

    // Key not found
```

```
    if (i == leaf_page->num_of_keys) {
        free(leaf_page);
        return -1;
    }

    delete_entry(leaf_offset, key, 0);
```

void delete_entry(off_t page_offset, int64_t key, off_t pointer_offset) 함수에서는 전반적인 B+tree 의 특정 page 에서 특정 key 값에 해당하는 레코드를 상황에 맞게 지우도록 하는 역할을 수행하게 된다. 해당 함수는 다음과 같다. 우선 remove_entry_from_node 함수를 실행해 인자로 받은 것들을 그대로 인자로 넘겨주는데, 해당 함수는 직접적으로 해당 page 에서 key 를 지우는 역할을 한다. Internal page 와 leaf page 의 구조가 다르기 때문에 직접 지워주기 위해선 두 케이스를 나누어 실행해야 했다. 우선 만약 인자로 받은 page_offset 을 사용해 load 한 타겟 페이지가 leaf 페이지인 경우 단순히 해당 key 값에 해당하는 record 의 index 를 찾은 후 해당 값을 없애기 위해 해당 인덱스 오른쪽의 레코드들을 왼쪽으로 한칸씩 당겨오도록 했다.

```
// Remove the record containing the key, and move the remaining records to left
    if (target_page->is_leaf) {  // Case : Leaf page
        while (target_page->records[i].key != key)
            i++;
        for (++i; i < target_page->num_of_keys; i++)
            target_page->records[i - 1] = target_page->records[i];
    }
```

반대로, leaf page 가 아닌 경우에는, pointer 를 하나 더 사용하기 때문에, key 값은 위와 동일한 로직으로 처리되는 반면, p_offset 의 경우에는, next_offset 에 대한 고려가 필요하다. Internal page 의 경우 인자로 지울 record 가 포함한 p_offset 을 함께 제공하는데, 해당 값과 일치하는 internal record 의 인덱스를 찾는다. 만약 next_offset 에 존재할 경우 -1 을 반환하고, 그게 아니라면 자신의 index 를 반환한다. 이후 해당 index 에 1 을 더해주고, 해당 값이 0 이라면 가장 왼쪽을 의미하기 때문에 b_f[0].p_offset 값을 next_offset 으로 옮겨준다. 그게 아니라면, 일관되게 j 번째 b_f 를 j-1 번째로 옮기도록 했다. 이렇게 처리하면, record 삭제가 완성되고, 이제 해당 page 의 num_of_keys—를 해준 뒤 해당 값을 disk 에 write 한다.

```
else { // Case : Internal page (Need to consider next_offset field)
        // Delete key value
        while (target_page->b_f[i].key != key)
            i++;
        for (++i; i < target_page->num_of_keys; i++)
            target_page->b_f[i - 1].key = target_page->b_f[i].key;

        // Delete p_offset value (if target pointer offset is in next_offset field,
delete and move all remainings left)
        while (target_page->b_f[j].p_offset != pointer_offset) {
            if (target_page->next_offset == pointer_offset) {
                j = -1;
                break;
            }
            j++;
        }
        for (++j; j < target_page->num_of_keys; j++) {
```

```
            if (j == 0)  {
                target_page->next_offset = target_page->b_f[j].p_offset;
                continue;
            }
            target_page->b_f[j - 1].p_offset = target_page->b_f[j].p_offset;
        }
    }

    target_page->num_of_keys--;

    pwrite(fd, target_page, sizeof(page), page_offset);

    free(target_page);
```

이후, 다시 delete_entry 로 돌아와 해당 page_offset 에 해당하는 페이지를 로드한다. 만약 root 로부터의 deletion 이었다면, adjust_root()함수를 실행시키도록할 건데, 해당 함수에는 만약 rt 가 leaf 인 경우와 leaf 가 아닌 경우로 나뉘어서 처리된다. 만약 rt->num_of_keys > 0 이라면 지워도 문제가 되지 않기 때문에 바로 종료하지만, 0 이 되는 경우 해당 root 페이지를 없애줘야 한다. 만약 root 가 leaf 가 아니라면 rt 가 가리키는 next_offset 의 페이지 (첫번째 child)를 루트로 만든다. 이후 hp->rpo 를 수정해주고, 기존 root 페이지를 usetofree 함수를 통해 비운 뒤 free page list 로 매달아 둔다. 이후 루트와 hp 를 reload 하게 된다.

```
// Case: empty root
    // If it has a child, promote the first child as the new root.
    if (!rt->is_leaf) {
        new_root = load_page(rt->next_offset);
        new_root->parent_page_offset = 0;
        pwrite(fd, new_root, sizeof(page), rt->next_offset);
        usetofree(hp->rpo); // free original root page and dangle to free page list
        hp->rpo = rt->next_offset;  // Change header page's root page offset with
first child offset.
        pwrite(fd, hp, sizeof(H_P), 0);
        free(hp);
        hp = load_header(0);
        free(rt);
        rt = new_root; // Update new root.
        free(new_root);
    }
```

반대로 rt 가 leaf 인 경우, 곧 바로 해당 페이지를 usetofree 해주고, hp 와 rt 를 초기화 시킨다. 이렇게 되면 전체 tree 가 삭제되게 된다.

```
else { // If root is leaf (has no children), then whole tree is empty (free root
page and dangle to free page list)
        usetofree(hp->rpo);
        hp->rpo = 0;
        pwrite(fd, hp, sizeof(H_P), 0);
        free(hp);
        hp = load_header(0);
        free(rt);
        rt = NULL;
    }
```

다시 delete_entry 로 넘어와서 root 가 아닌 경우에는, delete 를 하고 나서 유지해야 하는 최소 size 를 계산하도록 했다. 만약 현재 target page 가 leaf 라면 LEAF_MAX 의 반의 ceil 을 leaf 가 아니라면 INTERNAL_MAX + 1 의 반의 ceil -1 을 최소 key 의 개수로 설정한다. 이후 만약 특정 record 를 지운 target_page 의 num_of_keys 가 해당 min_keys 보다 크거나 같다면, 문제가 없기 때문에 그대로 함수를 종료한다. 하지만, 최소 key 의 개수보다 작은 경우, merge 혹은 redistribute 를 진행해야 하는데, 이를 판단하기 위해 현재 페이지의 parent_page 를 로드했다. 이후, merge 와 redistribute 에서 sibling page 의 위치를 알아야 하기 때문에 get_neighbor_index 라는 함수를 통해 neighbor_index 를 불러오도록 했다. Get_neighbor_index 는 현재 페이지의 가장 가까운 neighbor 의 위치를 찾도록 했다. 위치란 것은 타겟 페이지의 부모 페이지의 p_offset index 를 의미한다. 하지만 만약 해당 page 가 parent_page->next_offset 에 있다면 가장 왼쪽 페이지를 의미하기 때문에 곧바로 -2 를 반환하도록 했고, 그게 아니라면 b_f[i].p_offset 에서 현재 page 의 offset 값에 해당하는 index i 를 찾고, 그 왼쪽은 i – 1 을 반환하도록 했다.

```
// Find the index of a page's nearest neighbor (sibling) from the left if one
exits. (If not return -2)
int get_neighbor_index(page * target_page, off_t page_offset) {
    int i;
    page * parent_page = load_page(target_page->parent_page_offset);

    // find the index of left sibling page
    if (parent_page->next_offset == page_offset) { // if target page is leftmost
page, return -2
        free(parent_page);
        return -2;
    }
    for (i = 0; i < parent_page->num_of_keys; i++) { // else return index of left
page (index -1 means next_offset field)
        if (parent_page->b_f[i].p_offset == page_offset) {
            free(parent_page);
            return i - 1;
        }
    }
    // Error state
    printf("Search for nonexistent pointer to page in parent.\n");
    exit(-1);
}
```

이후, delete_entry 로 다시 돌아 와서 neighbor_index 값을 사용해 k_prime_index 를 계산하게 되는데 이 경우, 다 자신의 neighbor index 를 그대로 사용하돼, -2 인 경우에만 -1 로 바꿔주었다. 이렇게 한 이유는 -2 인 경우는 왼쪽에 neighbor page 가 없기 때문에 그냥 오른쪽 page 을 neighbor 로 사용하기 위함이다. 이렇게 k_prime_index 를 계산한 뒤에는, 현재 page 와 neighbor page 를 구분하는 parent 의 key 값을 찾아야 하는데 parent_page->b_f[k_prime_index + 1].key 를 통해 해당 값을 찾고 k_prime 에 저장했다. 이후 neighbor_offset 의 경우 앞서 말한대로 neighbor_index 가 -2 인 경우, 그 오른쪽인 b_f[0].p_offset 을 사용했고, -1 인 경우 가장 왼쪽인 next_offset 값을 그게 아닌 경우 parent_page->b_f[neighbor_index].p_offset 값을 사용했다.

```
// Case: page falls below minimum. (Merge of Redistribute)
    parent_page = load_page(target_page->parent_page_offset);
    neighbor_index = get_neighbor_index(target_page, page_offset); // get nearest
sibling page index
    k_prime_index = neighbor_index == -2 ? -1 : neighbor_index; // if target page
is leftmost page
    k_prime = parent_page->b_f[k_prime_index + 1].key; // key of left page
    if (neighbor_index == -2) { // target_page is in parent page's next_offset
field (leftmost page)
        neighbor_offset = parent_page->b_f[0].p_offset;
    }
    else if (neighbor_index == -1) { // target_page is first index of b_f array.
(left sibling is next_offset field)
        neighbor_offset = parent_page->next_offset;
    }
    else { // other cases
        neighbor_offset = parent_page->b_f[neighbor_index].p_offset;
    }
```

이제 merge 와 redistribute 를 위해 필요한 정보는 다 찾았고, 둘 중에 어떤 걸 실행할지 판단하기 위해 capacity 를 확인해야 한다. 이미 들어갈 수 있는 max 값은 정의가 되어 있기 때문에 만약 타겟 페이지가 leaf 라면 LEAF_MAX 를 아니라면 INTERNAL_MAX 를 capacity 로 설정했다. 이후 neighbor_offset 에 해당하는 페이지를 로드 하고, neighbor_page 의 키의 개수와 target_page 의 키의 개수를 더해서 해당 값이 capacity 보다 작은 경우, merge 를 실행하도록 하고, 더 했을 때 capacity 를 초과하는 경우 redistribute 를 실행한다.

```
capacity = target_page->is_leaf ? LEAF_MAX : INTERNAL_MAX;

    neighbor_page = load_page(neighbor_offset);

    // If there is room for merge
    if (neighbor_page->num_of_keys + target_page->num_of_keys < capacity) {
        // printf("merge start!\n");
        merge_pages(page_offset, target_page, neighbor_offset, neighbor_page,
neighbor_index, k_prime);
    }
    // Else, redistribute
    else {
        // printf("redistribute start!\n");
        redistribute_pages(page_offset, target_page, neighbor_offset, neighbor_page,
neighbor_index, k_prime_index, k_prime);
    }
```

우선 merge 의 경우, merge_pages 라는 함수로 실행된다. 이후, 이 merge 의 처리를 항상 target_page 를 오른쪽에 위치하고, neighbor_page 를 왼쪽에 위치하도록 고정하고 하기 위해서, 만약 인자로 전달한 neighbor_index 값이 -2 인 경우, target_page 가 왼쪽에 있음을 의미하기 때문에 neighbor_offset 과 target_offset 값을 서로 바꿔준다. 이후 neighbor_page 로 들어가야 하기 때문에, neighbor_insertion_index 를 가장 뒤의 index 인 neighbor_page->num_of_keys 로 설정한다. 이후 target_page 가 leaf 가 아닌 경우, k_prime 과 target_page->next_offset 값을 우선 neighbor_page 로 넘겨준다. 이후, 해당 이렇게 옮긴 p_offset 의 페이지가 neighbor_offset 을

parent_page 로 가리키도록 한다. 이 과정에서도 disk write 은 반드시 필요하다. 이후, 남은 target_page 의 internal records 를 neighbor_page 로 가져온다. 이후 해당 p_offset 페이지의 parent 또한 neighbor_offset 으로 바꿔준다.

```
// Case: non-leaf page
    // Append k_prime and the following pointer.
    // Append all pointers and keys from the neighbor.
    if (!target_page->is_leaf) {
        // Append k_prime & leftmost pointer of target_page
        neighbor_page->b_f[neighbor_insertion_index].key = k_prime;
        neighbor_page->b_f[neighbor_insertion_index].p_offset = target_page-
>next_offset;
        neighbor_page->num_of_keys++;

        // Update parent_page_offset field of child pages and write to disk.
        tmp_offset = neighbor_page->b_f[neighbor_insertion_index].p_offset;
        tmp_page = load_page(tmp_offset);
        tmp_page->parent_page_offset = neighbor_offset;
        pwrite(fd, tmp_page, sizeof(page), tmp_offset);
        free(tmp_page);

        target_page_end = target_page->num_of_keys;

        // Copy all remaining keys and pointers of target page to neighbor page
        for (i = neighbor_insertion_index + 1, j = 0; j < target_page_end; i++, j++)
{
            neighbor_page->b_f[i] = target_page->b_f[j];
            neighbor_page->num_of_keys++;
            target_page->num_of_keys--;

            // Update parent_page_offset field of child pages and write to disk.
            tmp_offset = neighbor_page->b_f[i].p_offset;
            tmp_page = load_page(tmp_offset);
            tmp_page->parent_page_offset = neighbor_offset;
            pwrite(fd, tmp_page, sizeof(page), tmp_offset);
            free(tmp_page);
        }
    }
```

반대로 leaf 인 경우, target_page 의 레코드들을 neighbor_page 로 모두 복사해 오고, neighbor_page->next_offset 을 기존 target_page 의 next_offset 으로 바꿔주고, disk update 를 진행한다.

```
// If target page is a leaf. (Append keys and pointer of target_page to neighbor
page and update next_offset field)
    else {
        // copy records to neighbor page
        for (i = neighbor_insertion_index, j = 0; j < target_page->num_of_keys; i++,
j++) {
            neighbor_page->records[i] = target_page->records[j];
            neighbor_page->num_of_keys++;
        }
```

```
        // Update neighbor page's pointer to next page with right page pointer
        neighbor_page->next_offset = target_page->next_offset;
    }
```

이후, 이렇게 두개가 합쳐지고 나면 target_page 는 사라지기 때문에 다시 delete_entry 를 호출해 parent 에서 해당 k_prime 값을 지우도록 한다. 이때, page_offset 을 함께 pointer 로 전달한다.

```
// Write updated neighbor page to disk
    pwrite(fd, neighbor_page, sizeof(page), neighbor_offset);

    // Delete pointer to target_page at parent page.
    delete_entry(target_page->parent_page_offset, k_prime, page_offset);

    // Free target page and dangle to free page list
    usetofree(page_offset);

    // Reload rt
    free(rt);
    rt = load_page(hp->rpo);
```

마지막으로, redistribute 의 경우, 두 케이스로 크게 나누어 만약 neighbor_index 가 -2 가 아니라면 현재 neighbor page 가 target_page 의 왼쪽에 존재함을 의미하기 때문에, neighbor page 의 가장 오른쪽 값의 pair 를 target page 의 가장 왼쪽으로 가져온다. 이 경우에도, leaf 는 records 이고, internal page 의 경우에는 b_f 이기 때문에 두 case 를 나눠서 코드를 구성했다. 이때 internal page 에서는 next_offset 에 대한 고려도 필요하다. 이때 중요한 점은 redistribute 의 과정에서 target page 가 leaf 가 아닌 경우에는 target_page 의 첫번째 key 값에 k_prime 값을 넣고, 기존 k_prime 이 존재하던 부모 페이지의 위치에 neighbor_page 의 마지막 key 값을 넣는다는 점이다. 반대로 leaf 인 경우, neighbor_page 의 가장 오른쪽 record 를 target page 의 가장 왼쪽에 가져온 뒤 가져온 첫번째 key 값을 부모 페이지의 k_prime 위치에 넣도록 코드를 작성했다.

```
// Case: target page has a neighbor page to the left
    // Pull the neighbor page's rightmost pair to target page's left end.
    if (neighbor_index != -2) {
        // Move records of target page to right by 1
        if (!target_page->is_leaf) { // If target page is internal page, need to
consider next_offset field
            for (i = target_page->num_of_keys; i > 0; i--) {
                target_page->b_f[i] = target_page->b_f[i - 1];
            }
            target_page->b_f[0].p_offset = target_page->next_offset;
        }
        else { // If target page is leaf page
            for (i = target_page->num_of_keys; i > 0; i--) {
                target_page->records[i] = target_page->records[i - 1];
            }
        }
        // Pull the neighbor page's rightmost pair to target page's left end.
        if (!target_page->is_leaf) { // If target page is internal page, need to
consider child page's parent_page_offset field
```

```
            target_page->next_offset = neighbor_page->b_f[neighbor_page->num_of_keys
- 1].p_offset;

            // Update child page's parent_page_offset field to target_page and write
to disk.
            tmp_offset = target_page->next_offset;
            tmp_page = load_page(tmp_offset);
            tmp_page->parent_page_offset = page_offset;
            pwrite(fd, tmp_page, sizeof(page), tmp_offset);
            free(tmp_page);

            // Update target_page's key value and parent page's key value and write
to disk.
            target_page->b_f[0].key = k_prime;
            tmp_page = load_page(target_page->parent_page_offset);
            tmp_page->b_f[k_prime_index + 1].key = neighbor_page->b_f[neighbor_page-
>num_of_keys - 1].key;
            pwrite(fd, tmp_page, sizeof(page), target_page->parent_page_offset);
            free(tmp_page);
        }
        else { // If target page is leaf page
            target_page->records[0] = neighbor_page->records[neighbor_page-
>num_of_keys - 1];

            // Update parent page's key with added leftmost key value of target page
and write to disk.
            tmp_page = load_page(target_page->parent_page_offset);
            tmp_page->b_f[k_prime_index + 1].key = target_page->records[0].key;
            pwrite(fd, tmp_page, sizeof(page), target_page->parent_page_offset);
            free(tmp_page);
        }
    }
```

반대로, target_page 가 가장 왼쪽 child 일 경우, target_page 가 leaf 라면 neighbor_page 의 가장 왼쪽 레코드를 타겟 페이지의 가장 오른쪽 레코드에 추가하고, neighbor_page 의 나머지 레코드를 옆을 한칸씩 밀어준다. 이때 부모 페이지의 k_prime_index + 1 의 key 값을 교체된 neighbor page 의 첫번째 레코드 key 값으로 교체한다.

```
if (target_page->is_leaf) { // If target page is leaf page
            // get the leftmost key & value of neighbor page to target page's
rightmost position
            target_page->records[target_page->num_of_keys] = neighbor_page-
>records[0];

            // Update parent page's key value with new leftmost key of neighbor page
and write to disk
            tmp_page = load_page(target_page->parent_page_offset);
            tmp_page->b_f[k_prime_index + 1].key = neighbor_page->records[1].key;
            pwrite(fd, tmp_page, sizeof(page), target_page->parent_page_offset);
            free(tmp_page);
```

```
            // Move remaining records in neighbor page to left by 1
            for (i = 0; i < neighbor_page->num_of_keys - 1; i++) {
                neighbor_page->records[i] = neighbor_page->records[i + 1];
            }
        }
```

반대로 target page 가 internal page 라면, target_page 의 가장 끝 record 의 key 값으로 k_prime 값을 p_offset 으로는 neighbor_page 의 첫번째 p_offset 값을 가져온다. 이후, 가져온 p_offset 페이지들의 parent_page_offset 을 해당 target page 의 offset 으로 변경한다. 이후 기존 부모 페이지의 key 값을 neighbor page 의 첫번째 key 값으로 교체한다. 이후의 과정은 남은 neighbor page 의 레코드들을 왼쪽으로 한칸씩 미는 과정이다. 이렇게 되면 redistribute 가 완료되고, target_page 의 key 개수를 1 증가, neighbor page 의 경우 1 감소 시킨 후 disk 에 적고, rt 를 다시 로드해준다.

```
else { // If target page is internal page
            target_page->b_f[target_page->num_of_keys].key = k_prime;
            target_page->b_f[target_page->num_of_keys].p_offset = neighbor_page-
>next_offset;

            // Update child page's parent_page_offset field to target_page and write
to disk.
            tmp_offset = target_page->b_f[target_page->num_of_keys].p_offset;
            tmp_page = load_page(tmp_offset);
            tmp_page->parent_page_offset = page_offset;
            pwrite(fd, tmp_page, sizeof(page), tmp_offset);
            free(tmp_page);

            // Update parent page's key value with neighbor page's leftmost key value
and write to disk.
            tmp_page = load_page(target_page->parent_page_offset);
            tmp_page->b_f[k_prime_index + 1].key = neighbor_page->b_f[0].key;
            pwrite(fd, tmp_page, sizeof(page), target_page->parent_page_offset);
            free(tmp_page);

            // Move remaining records in neighbor page to left by 1
            for (i = 0; i < neighbor_page->num_of_keys - 1; i++) {
                if (i == 0) {
                    neighbor_page->next_offset = neighbor_page->b_f[0].p_offset;
                }
                neighbor_page->b_f[i] = neighbor_page->b_f[i + 1];
            }
        }
// Update number of keys from both target page & neighbor page and write them to
disk.
    target_page->num_of_keys++;
    neighbor_page->num_of_keys--;
    pwrite(fd, target_page, sizeof(page), page_offset);
    pwrite(fd, neighbor_page, sizeof(page), neighbor_offset);

    // Reload rt
```

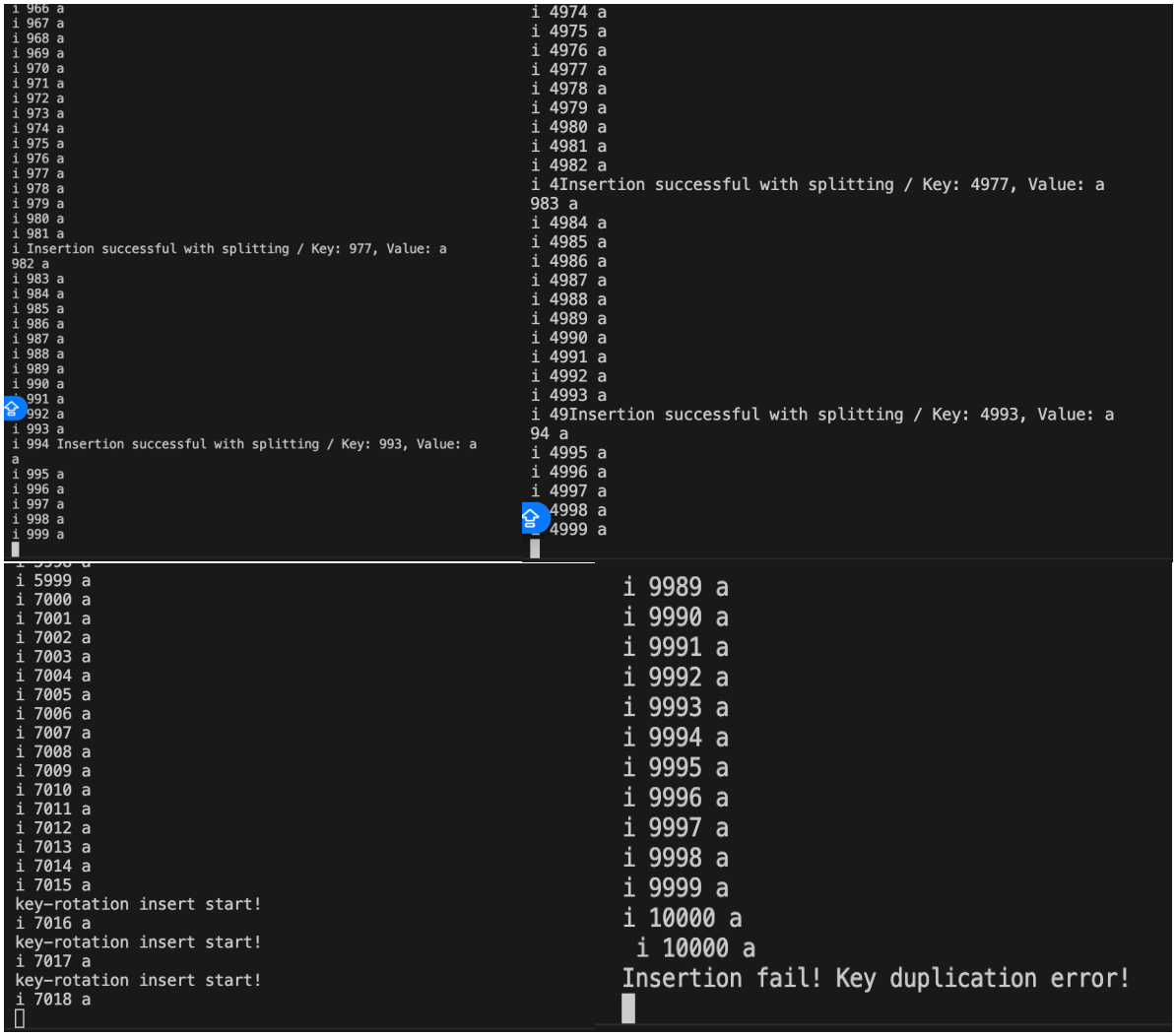
```
    free(rt);
    rt = load_page(hp->rpo);
```

이렇게 되면, redistribute 가 끝나고 db_delete 가 종료되게 된다.

3. Result

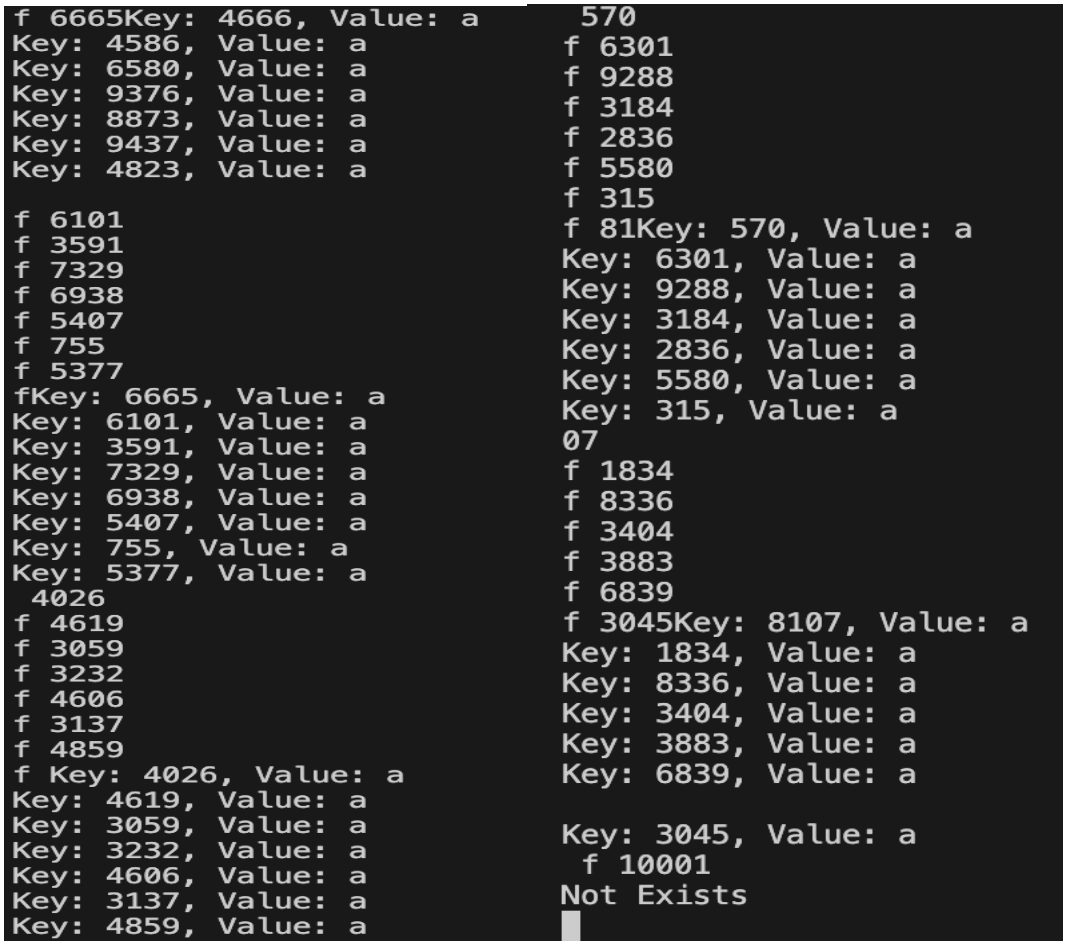
테스트를 위해 python script 를 짜, 1 부터 10000 까지의 수를 key 값으로 하고 value 는 a 로 통일한 째로 명령문을 만들어 파일에 저장습니다. 1000 개씩 홀수 순서로 먼저 넣고, 짝수 순서로 그 이후를 넣어보면서 insert 여부를 테스트했습니다. 이후 500 번의 1~10000 사이의 랜덤한 수에 대해 find 를 테스트해보았고, 마지막으로 2000 개의 랜덤 값을 delete 한 후 여부를 확인하고, f 를 통해 남은 값들에 대해 500 번 테스트를 진행했습니다.

(1) Insertion (1000 개씩 교차로 진행 0, 2, 4, 6, 8 이후 1, 3, 5, 7, 9)



위에서 언급한대로 1000 개씩 구간을 나누어 0~999, 2000~2999, 4000~4999, 6000~6999, 8000~8999, 1000~1999, 3000~3999, 5000~5999, 7000~7999, 9000~10000 까지 차례로 넣으면서 테스트해본 결과 key-rotation 과 split 이 필요한 경우 적절히 일어나는 것을 확인할 수 있었다. 곧바로 leaf 에 들어갈 수 있는 경우는 별도의 print 처리를 하지 않았다. 이후, 중복 값을 삽입할 때 올바른 문구를 띄워주는 것도 확인할 수 있었다.

(2) Db_find (랜덤하게 500 번)



500 번의 (1~10000 사이의 key 값에 대한) 랜덤한 find 에 대해서도 잘 동작하는 걸 확인해볼 수 있었다. 일부로, 10001 이라는 key 를 찾아보니 Not Exists 로 정상적으로 출력되었다.

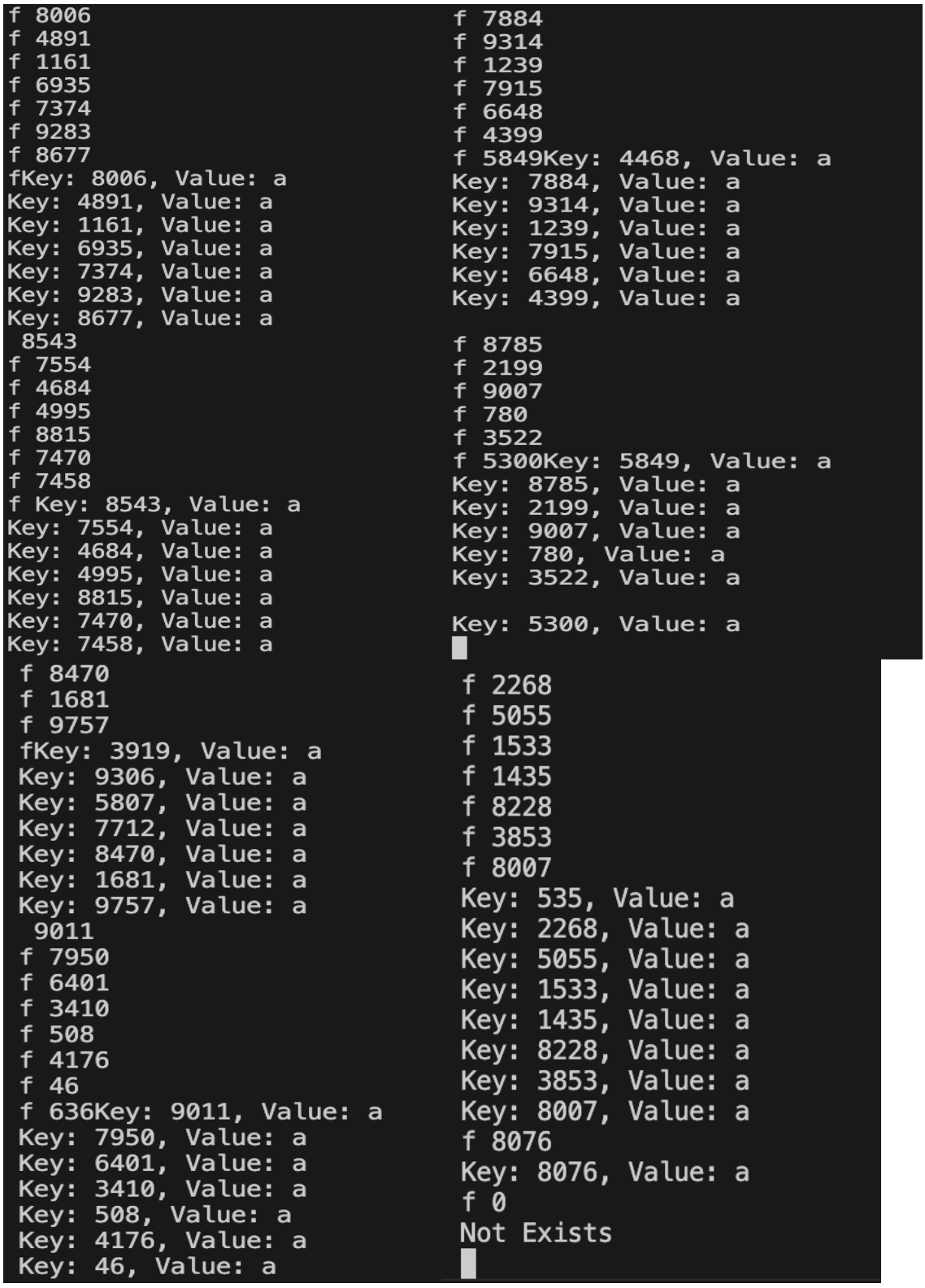
(3) Db_delete (1~10000 사이의 key 값에 대해 랜덤하게 2000 개 delete)

```
redistribute start!
Deletion successful Key: 1117
 880
d 8991
d 5788
d 519
d 6141
d 2809
d 3123
d 64redistribute start!
Deletion successful Key: 880
Deletion successful Key: 8991
Deletion successful Key: 5788
redistribute start!
Deletion successful Key: 6141
56
d 5335
d 226
d 2793
d 5444
d 7165
d 364
d 4782
Deletion successful Key: 6456
Deletion successful Key: 5335
Deletion successful Key: 5444
d 374
d 8060
d 4208
d 3487
d 8174
d 735redistribute start!
Deletion successful Key: 5039
redistribute start!
Deletion successful Key: 6180
redistribute start!
Deletion successful Key: 374
merge start!
Deletion successful Key: 8060
redistribute start!
Deletion successful Key: 4208
merge start!
Deletion successful Key: 3487
redistribute start!
Deletion successful Key: 8174
9
d 3517
d 4033
d 3115
d 8907
d 5079
d 7012
```

```
d 226
d 2793
d 5444
d 7165
d 364
d 4782
d 3677
d Deletion successful Key: 5335
Deletion successful Key: 226
Deletion successful Key: 2793
Deletion successful Key: 5444
Deletion successful Key: 7165
Deletion successful Key: 364
redistribute start!
Deletion successful Key: 4782
redistribute start!
Deletion successful Key: 3677
8413
d 7721
d 25
d 5889
d 9079
d 7020
d 5299
d 477Deletion successful Key: 8413
Deletion successful Key: 7721
Deletion successful Key: 25
Deletion successful Key: 5889
Deletion successful Key: 9079
redistribute start!
Deletion successful Key: 7020
Deletion successful Key: 5299
0
d 216Deletion successful Key: 4770
```

Db_delete 의 경우에도 상황에 적절하게 문제가 없는 경우 곧바로 삭제하거나 필요한 경우, merge 혹은 redistribute 를 적절하게 수행하는 것을 확인해볼 수 있었다,

(4) Db_find (지우고 난 값을 가지고 랜덤하게 500 번 find)



1~10000 사이의 key 에 대해 랜덤하게 2000 개 삭제 후 find 를 한 결과 모든 값을 적절하게 찾는 것을 확인할 수 있었다. 마지막에 테스트로 해본 f 0 에 대해서는 Not Exists 라는 값을 보여줬다. 이로써 db_find, db_insert, db_delete 모두에 문제 없이 잘 동작함을 확인했다.

4. Trouble Shooting

이번 과제는 그동안 있었던 2 번의 과제에 비해 단연 그 난이도가 높았던 과제였다. 이번 과제를 하면서 제공해주신 in-memory B+tree 코드를 분석하고, 이를 명세에 맞게 변형하는 과정에서 이해하는데 엄청 많은 시간을 소비한 것 같다. 제공해주신 소스코드를 분석하며 하나하나 주석을 달며 그 용도를 파악했다. 이후 구현 과정에서 정말 많은 에러를 만났지만, 그중 가장 기억 나는 두가지는 insert 와 delete 를 한 후에 특정 page 가 통째로 사라지는 문제였다. Find 를 통해 찾으려고 했지만, 해당 page 로의 참조하는 p_offset 이 중간에 소실되었다고 생각해. 해당 로직을 처리하는 함수를 10 번 정도 다시 읽어본 것 같다. 하지만, 코드 상의 문제를 발견하지 못했고 의아해 하던 와중에, 해당 문제가 발생한 이유가 insert 후 다른 여러 값들을 변동 시킨 이후 루트 페이지인 rt 를 reload 해주지 않아 발생한 문제였다. 이를 깨닫고, 수정한 결과 이 문제는 해결되었다. 또 하나는 merge 를 진행 했을 때 또 특정 key 값에 접근하지 못하게 되는 경우가 발생했다. 해당 문제점을 찾으려고 했으니 찾지 못했고, 결국 포기하려던 와중에 해당 명세를 작성하는 과정에서 문제점을 발견했다. Neighbor_offeset 이 -2 인 경우 neighbor_offset 과 target_offset 값을 서로 교환하는 과정이 있는데, offset 만 교환하고, 실제 page 를 로드하는 과정을 포함하지 않았다. 이로 인해, offset 값은 바꼈음에도 page 는 그대로 유지돼 이와 같은 문제가 발생했던 것이었다. 따라서 해당 내용을 수정하고, 다시 테스트 해본 결과, 명세의 모든 기능이 올바르게 실행되는 것을 확인할 수 있었다.

5. Key-rotation insert 에 대한 고찰

처음 key-rotation insert 라는 개념을 명세를 읽어보며 처음 접하게 되었을 때, 당연히 기존의 B+tree 보다 훨씬 좋겠다는 생각이 들었다. 하지만, 그렇다면 왜 사람들이 쓰지 않을까라는 의문을 갖게 되었다. 이러한 의문을 가진 채로 key-rotation insert 를 구현했는데, 구현 과정에서 그 원인을 찾을 수 있었다. Right sibling page 의 빈자리에 넣을 수 있다면 무조건 좋아보였지만, 실제로는 항상 그런 것만은 아니었다. Disk based B+tree 에서 페이지 한번을 읽는 것은 한번의 I/O 이기 때문에 이는 성능에 큰 영향을 준다. 따라서 자 I/O 횟수의 관점에서 평가해보고자 한다. Key-rotation insert 를 했을 때 성능이 좋아지려면 우선 right sibling page 와 현재 page 의 parent page 가 동일해야 한다. 만약, 서로의 부모가 다르다면 root 에 도착할 때까지 key 값을 확인하며 새로 들어온 값이 기존의 key 조건을 violate 하지 않는지 확인해야 한다. 이때 tree 의 높이 (log n/2 K) 만큼의 I/O 가 발생한다. 따라서, 이는 key-rotation insert 를 사용하지 않고, 그냥 split 했을 때, 부모 페이지에서도 계속 split 이 발생하는 최악의 상황을 제외하면 I/O overhead 가 훨씬 크다. 반대로, 우선 right sibling page 와 현재 page 의 parent page 가 동일하다면, 만약 key-rotation insert 를 사용하지 않았을 때 split 이후 parent page 의 공간이 부족해 부모 페이지에서도 split 이 발생하는 경우에 한해서 key-rotation insert 가 유리하게 된다. 그 이유는 key-rotation insert 의 경우, 부모 page 에 이미 존재했던 entry 의 key 값만 수정하면 되기 때문에 추가로 한번의 I/O 가 발생하게 된다. 하지만, split 의 경우 새로운 record 를 추가해줘야 하기 때문에 만약 부모 페이지에 자리가 있다면 똑같은 횟수의 I/O 가 발생할테지만, 부모 페이지에 공간이 없는 경우 추가적인 split 을 위한 overhead 가 발생하게 된다. 따라서, 이 경우를 제외하고는 key-rotation insert 를 사용해서 얻을 수 있는 성능상의 이점이 없다고 분석했다.