Project2 MileStone 1

2017030055 윤상현

B+ 트리는 B트리와 다르게 key 값에 value 값이 붙어 record형식을 취하고, leaf 노드에 실질적인 record들이 들어있고, 모든 record값들이 포함됩니다. 그리고 그 상위의 internal 노드들에는 record가 들어가있는 leaf로 가는 위치를 보여주는 key 값들이 들어있게 됩니다.

각 key들은 하나의 기준이 되어 해당 key의 왼쪽에는 해당 값보다 작은 값들을, 오른쪽에는 해당 값보다 큰 값들이 존재합니다.

이러한 형식으로 인해 leaf node 에선 leftmost record의 key 값이 부모 key값과 동일합니다.

아래는 선언된 구조체들입니다.

```
#define LEAF_MAX 31
#define INTERNAL_MAX 248
typedef struct record{ //record.  key, value 로 타입 정의
    int64_t key;
    char value[120];
}record;

typedef struct inter_record {
    int64_t key;
    off_t p_offset; //해당 키의 오른쪽 자식 offset
}I_R;

typedef struct Page{
    off_t parent_page_offset;
    int is_leaf;
    int num_of_keys;
    char reserved[104];
    off_t next_offset;   //leftmost offset. leftmost key의 left child이다.

    union{
        I_R b_f[248];  //key, 해당 key의 오른쪽 자식 offset 로 구성된 I_R을 모은 배열
        record records[31]; // record ( 실제 값 보유) key, value leaf에서 사용.
    };
}page;

typedef struct Header_Page{ //헤더 페이지. 전체적인 트리 구조를 보유.
    off_t fpo;
    off_t rpo;
    int64_t num_of_pages;
    char reserved[4072];
}H_P;

extern int fd;
extern page * rt; //루트
```

설명에 앞서 모든 디테일한 설명들은 각 코드 주석에 적어 놓았기에 참고해주시면 됩니다.

Record type은 bpt.h에 typedef로 정의되어 있습니다. Key값과 Value 값을 보유 중입니다.

각 페이지는 해당 노드가 leaf 인지 확인하는 boalean 타입과, 보유한 (key, 오른쪽 자식)을 각각 Key, offset으로 가지는 I_R이라는 구조체를 저장하는 배열을 보유하는데, 이때 leftmost key의 Left에 위치한 offset 정보가 존재하지 않기에 next_offset에 따로 저장해줍니다.

Find

1) Db_find

```
char * db_find(int64_t key) { //해당 key가 존재하는 leaf까지 내려가서 value 값을 리턴. 없으면 NULL
    char * value = (char*)malloc(sizeof(char) * 120); //값을 할당
    int i = 0;
    off_t fin = find_leaf(key); //key 보유한 leaf의 offset
    if (fin == 0) {//해당 key 보유 leaf x 시 find 실패
        return NULL;
    }
    page * p = load_page(fin); //찾은 leaf page를 offset으로 로드한다.

    for (; i < p->num_of_keys; i++) { // leaf에서 해당 키가 존재하는 곳에 i를 위치시킨다
        if (p->records[i].key == key) break;
    }
    if (i == p->num_of_keys) { //끝까지 찾지 못함 에러!
        free(p);
        return NULL;
    }
    else {
        strcpy(value, p->records[i].value); //"value"에 찾은 i 위치의 value 값을 복사
        free(p);
        return value;
    }
}

int cut(int length) {
    if (length % 2 == 0)
        return length / 2;
    else
        return length / 2 + 1;
}
```

해당 key가 존재하는지 찾고 value를 리턴 합니다. 올바르게 찾지못하면 NULL을 리턴합니다.

2) Find leaf

```
off_t find_leaf(int64_t key) { //해당 key가 위치할 leaf 노드를 찾는다.
    int i = 0;
    page * p;
    off_t loc = hp->rpo; //header page 의 rpo

    //printf("left = %ld, key = %ld, right = %ld, is_leaf = %d, now_root = %ld\n", rt->next_offset,
    //  rt->b_f[0].key, rt->b_f[0].p_offset, rt->is_leaf, hp->rpo);

    if (rt == NULL) { //root 존재 x 시 find 불가. ( 비어있는 tree)
        //printf("Empty tree.\n");
        return 0;
    }
    p = load_page(loc);  .. //페이지 할당받는다.

    while (!p->is_leaf) { //p가 leaf까지 내려갈때까지 반복
        i = 0;

        while (i < p->num_of_keys) { //0~ 해당 페이지의 key 끝까지 탐색
            if (key >= p->b_f[i].key) i++;  // 찾는 key보다 큰값 나올때까지 i 증가.
            else break;
        }
        //i: 찾는 key 값 이상의 값이 처음 나온 위치.

        if (i == 0) //찾는 key 값 이 b_f안에 없으면
            loc = p->next_offset; //leftmost로 이동
        else
            loc = p->b_f[i - 1].p_offset;  //그외의 값은 찾은 위치로 가져오기.


        //if (loc == 0)
        // return NULL;

        free(p);
        p = load_page(loc);  //p =key 값에 따라 타고 내려온 페이지

    }
    //while문을 벗어난 뒤 p 에는 key 값에 맞춰 leaf까지 타고 내려가 찾아진 leaf의 offset이 들어가있다.

    free(p);
    return loc;  //찾은 leaf의 offset

}
```

파라미터로 들어온 key가 위치할 적절한 leaf 노드(page)의 offset을 찾아서 리턴 합니다.

Insert

1) Db_insert

```
int db_insert(int64_t key, char * value) { //insert 명령어.
    /* scanf("%ld %s", &input, buf);
            db_insert(input, buf);  */

    record nr;  //record는 bpt.h에 정의되어있다.
    nr.key = key;  //nr에  파라미터로 들어온 key 값과
    strcpy(nr.value, value);  //value 값을 넣는다.
    if (rt == NULL) { //만약 루트 노드가 Null이면
        start_new_file(nr);  //새로운 파일을 생성한다.
        return 0;
    }

    char * dupcheck;
    dupcheck = db_find(key);  // 중복된 key 값을 찾아내고
    if (dupcheck != NULL) { //중복이 있다면
        free(dupcheck); //중복된 값을 free 시켜준다.
        return -1;
    }
    free(dupcheck); // duble check?

    off_t leaf = find_leaf(key);  //off_t > file offset value  해당 key를 가진 leaf의 file offset value 찾기

    page * leafp = load_page(leaf);  //해당 페이지를 로드한다.

    if (leafp->num_of_keys < LEAF_MAX) { //만약 리프가 속한 곳의 key의 수< Leaf_Max 일시  insert해도 split 발생 x
        insert_into_leaf(leaf, nr);
        free(leafp);
        return 0;
    }

    insert_into_leaf_as(leaf, nr);  //해당 리프가 속한 곳의 key 수 >= leaf_max 일시 insert하면 split 발생.
    free(leafp);
    //why double free? >> double check?
    return 0;
}
```

insert 명령의 기본입니다. Key를 넣을 위치를 찾은 뒤, leaf노드에 key를 추가해줍니다.

만약 아직 트리에 아무것도 들어가 있지 않다면, 새로운 root를 생성한 뒤 key, value를 넣고 리턴하고, 이미 tree에 존재하는 값을 insert하면 안되기에 미리 db_find로 해당 key가 존재하는지 체크해줍니다. 만약 없다면, find_leaf 함수를 통해 입력 값이 위치할 leaf 노드를 찾습니다.

이 외의 경우엔 2가지 케이스가 존재합니다.

key가 추가된 노드에서 최대 개수(M) 보다 작아 split이 불필요 하다면 insert_into_leaf를 통해 실제 insert를 진행합니다.

만약 최대개수 초과할 시 분할(split)이 발생합니다. 이때는 insert_into_leaf_as 함수로 타고 들어가게 됩니다.

2) Insert_into_leaf

```
off_t insert_into_leaf(off_t leaf, record inst) { //split 없이 insert

    page * p = load_page(leaf);
    //if (p->is_leaf == 0) printf("iii error : it is not leaf page₩n");
    int i, insertion_point;
    insertion_point = 0;

    //    insert할 idx를 record의 key 값에 맞게 설정한다. K(i-1) < K(i) < K(i+1)  //단, leaf_max 벗어나지 않게
    while (insertion_point < p->num_of_keys && p->records[insertion_point].key < inst.key) {
        insertion_point++;
    }
    for (i = p->num_of_keys; i > insertion_point; i--) {
        p->records[i] = p->records[i - 1]; //insert할 위치 전까지 record ( key, value 쌍) 을 복사 해준다
    }
    p->records[insertion_point] = inst; //값 insert
    p->num_of_keys++;
    pwrite(fd, p, sizeof(page), leaf); //insert된 페이지 write
    //printf("insertion %ld is complete %d, %ld₩n", inst.key, p->num_of_keys, leaf);
    free(p);
    return leaf;
}
```

Split이 필요 없이 단순 insert하는 메소드입니다.

값이 위치할 적절한 leaf node를 파라미터로 받았고, 이 leaf 안에서 해당 record가 위치할 위치를 찾은 뒤, insert를 진행합니다. 이후 write을 통해 변경사항을 write 해준 뒤 leaf를 리턴하며 종료됩니다.

3) Insert_into_leaf_as

Split 이 발생하는 insert에 대한 메소드 입니다.

```
off_t insert_into_leaf_as(off_t leaf, record inst) { //split 발생하는 insert
                    //  해당 leaf의 file offset value( find key 로 찾음) , record 값)

    //leaf 의 insert에서 split 발생시 중앙값 or inorder successor의 key를 오른쪽 자식에 붙여 줘야한다.

    off_t new_leaf;
    record * temp;
    int insertion_index, split, i, j;
    int64_t new_key;
    new_leaf = new_page();
    //printf("%n%ld is new_leaf offset%n%n", new_leaf);
    page * nl = load_page(new_leaf); //새로운 leaf 노드 생성
    nl->is_leaf = 1; //leaf 노드라는 표시 해준다.
    temp = (record *)calloc(LEAF_MAX + 1, sizeof(record)); // temp에  leaf_max+1  +  record 크기 만큼 공간 할당
    if (temp == NULL) {
        perror("Temporary records array");
        exit(EXIT_FAILURE);
    }
    insertion_index = 0;
    page * ol = load_page(leaf); //

    //     insert할 idx를 record의 key 값에 맞게 설정한다. K(i-1) < K(i) < K(i+1)   //단, leaf_max 벗어나지 않게
    while (insertion_index < LEAF_MAX && ol->records[insertion_index].key < inst.key) {
        insertion_index++;
    }
    for (i = 0, j = 0; i < ol->num_of_keys; i++, j++) {
        if (j == insertion_index) j++; // temp[insert할 위치] 에는 기존 record 값 넣지 않고 비워둔다.
        temp[j] = ol->records[i];  //insert할 위치 제외하고 record ( key, value 쌍) 를 복사 해준다.
    }
    temp[insertion_index] = inst; //위에서 비워둔 insert 위치에 insert 할 record 값인 inst를 넣는다.
    ol->num_of_keys = 0;
    split = cut(LEAF_MAX); //split 발생시의 분기점 짝수>> ex) 길이 4일시   2>>  0,1,//   2,3      오른쪽의 왼쪽
    // 홀수> 중간 값


    //왼쪽
    for (i = 0; i < split; i++) {   //분기점 전까진 유지
        ol->records[i] = temp[i];
        ol->num_of_keys++;
    }


    //오른쪽  >> 오른쪽 자식에 중앙값( or inorder_ successor 의 값) 포함.
    for (i = split, j = 0; i < LEAF_MAX + 1; i++, j++) { //분기점 ~ 끝까지 >> 새로운 leaf 노드에 넣는다.
        nl->records[j] = temp[i]; //nl: 처음에 생성한 비어있는 leaf 노드
        nl->num_of_keys++;
    }
```

왼쪽 노드, 중앙 키 값, 오른쪽 노드로 나눈 뒤 중앙 키 값을 부모로 올린 뒤 해당 키의 왼쪽, 오른쪽에 각각 연결해줍니다. 이때 부모key에 대응하는 값이 leaf에서 사라지게 되므로 모든 leaf를 뒤져보면 모든 record (key,value)를 얻을 수 있어야 한다는 조건이 깨지게 됩니다.

따라서 split된 inorder successor(오른쪽 자식의 leftmost)에 중앙값을 넣습니다.

```
    free(temp); //임시공간을 free 시켜준다.

    //    기존 > 기존에 가르키던 곳        >>>>split 발생!
    //     split되어 나온 노드  기존 > split되어 나온 것 > 기존에 가르키던 곳
    nl->next_offset = ol->next_offset;
    ol->next_offset = new_leaf; //

    for (i = ol->num_of_keys; i < LEAF_MAX; i++) { //의도한 record가 들어있는 공간 외의 나머지 공간은 비워둔다.
        ol->records[i].key = 0;
        //strcpy(ol->records[i].value, NULL);
    }

    for (i = nl->num_of_keys; i < LEAF_MAX; i++) {//의도한 record가 들어있는 공간 외의 나머지 공간은 비워둔다.
        nl->records[i].key = 0;
        //strcpy(nl->records[i].value, NULL);
    }

    nl->parent_page_offset = ol->parent_page_offset; //새롭게 split되어 나온 오른쪽 page도 기존과 같은 부모 page를 받아온다.
    new_key = nl->records[0].key; //오른쪽 자식의 첫번째 >> 새로운 부모 key 가 된다! new_key에 해당 값 저장.

    pwrite(fd, nl, sizeof(page), new_leaf); //  split된 오른쪽 write
    pwrite(fd, ol, sizeof(page), leaf);  //split된 왼쪽 write ( 기존 공간 재사용)
    free(ol);
    free(nl);
    //printf("split_leaf is complete₩n");

    return insert_into_parent(leaf, new_key, new_leaf);
```

Leaf에 대한 작업을 수행하고, split되어 빠진 중앙값을 부모 노드에 넣는 작업을 수행합니다.

4) Insert into parent

```
off_t insert_into_parent(off_t old, int64_t key, off_t newp) {  // split 된 왼쪽 , 새로운 부모 key, split된 오른쪽.

    //              2
    //           1 2 3 4
    //
    //           1 2  //   3 4
    //
    //                3  (2>3으로 )
    //            1,2       3,4

    //부모key record = key 값이 가장작은(왼쪽의) 오른쪽 자식의 record

    //여기서 split이 일어나면 split 발생 후 다시 부모에게 key 전달   or    split x 시 key를 정상적으로 insert 해준다.
    int left_index;
    off_t bumo;
    page * left;
    left = load_page(old);

    bumo = left->parent_page_offset; //부모 page 의 offset 가져오기.
    free(left);

    if (bumo == 0) //부모의 offest 이 0 일시 (없을 시)
        return insert_into_new_root(old, key, newp);  //새로 생성한 부모의 key 설정 후 old, newp를 왼쪽, 오른쪽 자식으로 설정.

    left_index = get_left_index(old);  //old(왼쪽 자식) 의 index 가르키는 포인터 위치(key 위치) get

    page * parent = load_page(bumo); //부모 페이지 로드
    //printf("₩nbumo is %ld₩n", bumo);
    if (parent->num_of_keys < INTERNAL_MAX) {//해당 부모 page가 가득 차지 않았다면, 새로운 key를 부모 key로 insert
        free(parent);
        //printf("₩nuntil here is ok₩n");
        return insert_into_internal(bumo, left_index, key, newp); //internal = non-leaf
    }
    free(parent);
    return insert_into_internal_as(bumo, left_index, key, newp); //해당 부모 page가 가득 찼다면 split 해야한다!
}
```

여기서부터 재귀적인 특성이 보이기 시작합니다. 부모에게 키를 추가했는데, 그 부모가 또다시 사이즈 초과로 split되어 부모에 key를 추가하고, 또 추가를 반복하며 root를 타고 갈때까지 재귀적인 반복이 될 수 있는 코드입니다. 부모에게 중앙값 key를 insert 해주는데, leaf에 추가할 때와 마찬가지로 split 되거나, 되지 않거나 2가지 케이스로 나누어 insert합니다. 모든 부모 노드들은 leaf가아닌 internal 노드이기에 insert_into_internal로 split이 발생하지 않고 internal 노드에 넣는 메소드를 구현하고, insert_into_internal_as로 split 이 발생하며 internal 노드에 insert 하는 케이스를 구현합니다.

5) insert_into_internal

```
off_t insert_into_internal(off_t bumo, int left_index, int64_t key, off_t newp) { //non-leaf node에 key 값 삽입, split x 경우
    //                  부모 offset,  왼쪽 자식 index ,  삽입할 key , 오른쪽 자식 offset

    //(하위 leaf에서 split 해서 key 가 bumo로 왔기에 internal에서 insert 발생한거
    // 즉 여기서 bumo 에게 key를 insert한 것은 bumo 의 직계 자식이 split하여 부모 key를 위로 올렸기 때문.
    // left 자식: 기존 노드 값 가진 노드 ,  right 자식: 새로 생성하여 split 값 분배한 노드.
    //leaf의 경우 오른쪽 자식의 가장 왼쪽에 부모 key 값 보유


    page * parent = load_page(bumo); //부모 page 로드
    int i;

    for (i = parent->num_of_keys; i > left_index + 1; i--) {// 끝에서 처음+2 (left_index+2) 까지
        parent->b_f[i] = parent->b_f[i - 1]; //한칸씩 앞으로 땡긴다.  left_index+2 값이 left_index+1에 들어간다.
    }
    //left_index +1 >> left child의 바로 오른쪽 index
    parent->b_f[left_index + 1].key = key; // split 일어나기 전의 자식을 가르켰던 index 오른쪽에 key 삽입

    //                  아래 트리에 17 insert 시?
    //          12            16
    // 10           12 13         16 18
    //
    //                   (bumo)                                  (bumo)
    //          12       16                        12           16        (17)
    // 10           12 13        16 17 18          >>10          12 13         16       18
    //
    //                   (split 전 노드)           (16: split전 노드정보 보유한 left)    (18: newp, 새로만든 오른쪽 노드)
            //   하위에서 split 발생해 오버된 상태
            //
    // 왼쪽 자식 가르키는 포인터(16 가르키는 key) 의 오른쪽 포인터 (17,18 가르키는 포인터) 에 삽입할 key 값인 17 insert 한것.


    parent->b_f[left_index + 1].p_offset = newp;  // 새로 삽입한 부모 키의 오른쪽 = newp(오른쪽 자식) 가르키게
    parent->num_of_keys++;
    pwrite(fd, parent, sizeof(page), bumo); //변경된 부모 write
    free(parent); //temporary하게 값 저장했던 parent free 시켜줌
    if (bumo == hp->rpo) {
        free(rt);
        rt = load_page(bumo); //bumo를 새로운 루트로 설정.
        //printf("₩nrt->numofkeys%lld₩n", rt->num_of_keys);

    }
    return hp->rpo;
}
```

부모 노드에 단순 insert를 수행해주면 됩니다.

6) insert_into_internal_as

```
off_t insert_into_internal_as(off_t bumo, int left_index, int64_t key, off_t newp) {
    //부모에 key 값 넣었을 때 size 초과로 split 발생
    //non-leaf 의 insert에서 split 발생시 중앙값 or inorder successor의 key를 오른쪽 자식에 붙여 놓을 필요 없다.

     //                  아래 트리에 17 insert 시?
     //        12          (13 16)
     // 10         ( 12 13)          (16 18)               xx:기존 13,16의 부모페이지    [16: new_parent의 parent]에 넘겨진다
     //
     //                    [bumo]                                    xx                xx16
     //        12          (13 16)                    12          (13  16  17 )    >>    13    17
     // 10          12 13           16 17 18          >>10         12 13           16  (17 18)       16   (17 18 )
     //                                                                                          [16: nn]
     //                    [split 전 노드]                                                    13:old_parent(bumo),  17: new_parent (
     //          //   하위에서 split 발생해 오버된 상태

    int i, j, split;
    int64_t k_prime;
    off_t new_p, child;
    I_R * temp;

    temp = (I_R *)calloc(INTERNAL_MAX + 1, sizeof(I_R));  //internal_record(자식 가르키는 포인터) 공간 새로 할당

    page * old_parent = load_page(bumo); //기존 bumo 페이지 로드

    for (i = 0, j = 0; i < old_parent->num_of_keys; i++, j++) {
        if (j == left_index + 1) j++;   //key를 삽입할 left_index+1 공간은 남겨두고 기존 포인터들 복사
        temp[j] = old_parent->b_f[i];
    }

    temp[left_index + 1].key = key; //key 삽입
    temp[left_index + 1].p_offset = newp; //오른쪽 자식 붙여넣기

    split = cut(INTERNAL_MAX); // 중앙값(or inorder successor)= 분기점 저장.
    new_p = new_page(); //새로운 페이지 공간.  ( 새롭게 만들어진 오른쪽 공간 용)
    page * new_parent = load_page(new_p); // 새로운 페이지 공간 로드
    //old_parent: split 발생했을 때 왼쪽 자식, 기존 노드 정보 보유
    //new_parent: split 발생 시 오른쪽 자식. 새롭게 생성됨.
    old_parent->num_of_keys = 0; //기존 parent 공간 비웠기에 key의 개수 =0 으로
    for (i = 0; i < split; i++) {  //가장 왼쪽 ~ 분기점 이전까지 old_parent에 저장
        old_parent->b_f[i] = temp[i];
        old_parent->num_of_keys++;
    }
    k_prime = temp[i].key;  //split 지점 key를 저장
    new_parent->next_offset = temp[i].p_offset; //새롭게 만든 오른쪽 자식의 왼쪽 노드 = split지점에서 가르키는 오른쪽 자식 노드.
```

부모에 중앙값을 insert함으로 또다시 초과가 발생하여 split하는 case를 다룹니다.

```
    for (++i, j = 0; i < INTERNAL_MAX + 1; i++, j++) { //++i 부터! split 지점 key right child에서 가지지 않게 설정.
        //분기점+1 ~ 끝 까지의 값을 new_parent 에 복사
        new_parent->b_f[j] = temp[i];
        new_parent->num_of_keys++;
    }

    //new_parent : temporary 공간 for 오른쪽    ,  new_p : write 하기 위한 용도 for 오른쪽
    new_parent->parent_page_offset = old_parent->parent_page_offset; //기존 정보가진 왼쪽 자식 노드로 부모 정보 로드
    page * nn;
    nn = load_page(new_parent->next_offset);  //inorder successor 새롭게 생성된 오른쪽 자식의 왼쪽 노드 페이지 로드
    nn->parent_page_offset = new_p; // 새로운 오른쪽이 가리켜야하는 자식들이 새로운 오른쪽(new_p) 가르키게
    pwrite(fd, nn, sizeof(page), new_parent->next_offset);// inorder succesor 정보 write
    free(nn);
    for (i = 0; i < new_parent->num_of_keys; i++) {   //오른쪽 자식의 처음~ 끝
        child = new_parent->b_f[i].p_offset;
        page * ch = load_page(child);
        ch->parent_page_offset = new_p;// 오른쪽 new_parent가 가르키는 자식들에게 새로운 부모 정보 기입
        pwrite(fd, ch, sizeof(page), child);  //오른쪽 자식의 모든 child page 정보 새롭게 write 해서 갱신
        free(ch);
    }

    pwrite(fd, old_parent, sizeof(page), bumo); //기존의 노드 값 가진 왼쪽 자식 갱신
    pwrite(fd, new_parent, sizeof(page), new_p);//새롭게 생성된 오른쪽 자식 갱신
    free(old_parent);
    free(new_parent);
    free(temp);
    //printf("split internal is complete₩n");
    return insert_into_parent(bumo, k_prime, new_p); //split 된 두 자식과 둘의 부모에 넣어야할 key 값 전달
}
```

재귀적으로 다시 parent에 key를 넣는 과정을 요청합니다. 이는 부모에게 중앙값을 insert해도 split이 일어나지 않을 때까지 반복되며 상위로 이동할 것입니다.

Delete

우선 삭제하기 전, 삭제할 키가 없거나, 해당 트리 루트가 비어 있는 경우를 예외처리해 준 뒤,

Delete_entry에서 본격적인 삭제를 처리합니다.

```
int db_delete(int64_t key) { //해당 key를 가진 leaf 내려가서 delete

    if (rt->num_of_keys == 0) { //루트가 비어있으면 삭제할게 없다.
        //printf("root is empty₩n");
        return -1;
    }
    char * check = db_find(key); //해당 key가 가진 value 를 저장
    if (check== NULL) { //해당 key가 root 내에 없으므로 삭제할게 없다.
        free(check);
        //printf("There are no key to delete₩n");
        return -1;
    }
    free(check);
    //key 가 root 내에 존재 할 경우
    off_t deloff = find_leaf(key);   //해당 key가 위치한 leaf 의 offset
    delete_entry(key, deloff); //
    return 0;

}//fin
```

1) Remove_entry_from_page

```
void remove_entry_from_page(int64_t key, off_t deloff) {

    int i = 0;
    page * lp = load_page(deloff); //삭제할 페이지를 offset으로 로드.
    if (lp->is_leaf) { // leaf가 아닌 노드를 지우는 경우.
        //printf("remove leaf key %ld₩n", key);
        while (lp->records[i].key != key)
            i++;

        for (++i; i < lp->num_of_keys; i++) //해당 키 뒤의 key들을 한칸씩 앞으로 당겨서 제거
            lp->records[i - 1] = lp->records[i];
        lp->num_of_keys--;
        pwrite(fd, lp, sizeof(page), deloff);
        if (deloff == hp->rpo) {
            free(lp);
            free(rt);
            rt = load_page(deloff);
            return;
        }

        free(lp);
        return;
    }
    else { //leaf 노드를 지우는 경우
        //printf("remove interanl key %ld₩n", key);
        while (lp->b_f[i].key != key) //i를 해당 key가 위치한 index까지 이동

        for (++i; i < lp->num_of_keys; i++) //(해당 key위치한 index +1) ~ 끝까지 왼쪽으로 한칸씩 당겨 해당 key 제거
            lp->b_f[i - 1] = lp->b_f[i];
        lp->num_of_keys--;//개수 줄이기
        pwrite(fd, lp, sizeof(page), deloff); //
        if (deloff == hp->rpo) {
            free(lp);
            free(rt);
            rt = load_page(deloff);
            return;
        }

        free(lp);
        return;
    }
```

해당하는 key를 단순히 삭제합니다. 모든 delete는 단순히 key를 지운 뒤, b+ 트리의 조건이 위배되었다면 tree modification을 수행하는 것으로 진행됩니다.

2) Delete_entry

```
void delete_entry( int64_t key, off_t deloff) {
    //해당 노드에서 해당 key 제거 후 문제가 생기면 적절한 tree modificatoin 전략 수행. 재귀적으로 사용한다!


    remove_entry_from_page(key, deloff); //key 제거. >> leaf / nonleaf 구분해서 제거한다.

    if (deloff == hp->rpo) { //트리 조정 (root에 key가 없다면 root를 조정한다)
        //: 최소개수 충족, 삭제할 key가 non-leaf(internal) 에 없는 경우.
        adjust_root(deloff);
        return;
    }
    page * not_enough = load_page(deloff); //지운 key가 위치한 노드를 로딩.
    int check = not_enough->is_leaf ? cut(LEAF_MAX) : cut(INTERNAL_MAX);
    //지운 노드가 leaf or non leaf 따라, 짝,홀 따라 적절한 중앙값 => 노드에서 key 개수의 최소조건 m/2 를 get.

    if (not_enough->num_of_keys >= check){  // key를 지웠는데 여전히 최소 조건 만족.  그냥 key 만 지우면 된다.
      free(not_enough);
      //printf("just deleteWn");
      return;  //단순히 key 지운 형태로 리턴.
    }

    int neighbor_index, k_prime_index;
    off_t neighbor_offset, parent_offset;
    int64_t k_prime; //logical하게 지워진 key의 오른쪽.
    parent_offset = not_enough->parent_page_offset;
    page * parent = load_page(parent_offset); //key를 지운 노드의 부모 노드 로딩.

    //   P.K
    // N1    N2    이런식으로
    //맨 왼쪽 인접노드들은 같은 부모키 가진다 *****
    if (parent->next_offset == deloff){ //next_offset key를 지운 경우, neighber_index 예외처리 ( 기존엔 left가 neighber node )
        neighbor_index = -2;//logical 하게 해당 위치를 -2로 지정
        neighbor_offset = parent->b_f[0].p_offset; //next_offset,b_f[0],b_f[1],....
        //즉 next_offset의 오른쪽 노드인 b_f[0]를 neighber 노드로 예외처리해준다.
        k_prime = parent->b_f[0].key;//부모키 설정
        k_prime_index = 0;
    }
    else if(parent->b_f[0].p_offset == deloff) {// b_f[0] 의 neighber은 next_offset 노드이다.
        neighbor_index = -1; //logical 하게 next_offset의 위치를 -1로 지정.
        neighbor_offset = parent->next_offset; //neighbor 노드 예외처리.
        k_prime_index = 0;
        k_prime = parent->b_f[0].key;//부모키 설정.
        // 지워진 key의 index에는 지운 key의 바로 오른쪽에 있던 노드정보가 들어감. (한칸씩 앞으로 당겨짐)
    }
```

우선 앞서 설명한 remove_ entry_from_page 함수를 통해 key 가 위치한 leaf 가서 단순히 해당 key를 제거합니다.

Key를 제거했는데 key최소 개수(m/2 이상) 만족하고, non-leaf(internal) 에 해당 key가 없다면 (leaf의 leafmost에 위치한 key 가 아니다) 단순 삭제가 가능합니다.

Adjust root는 root에서 구조가 변경할 시 root 구조를 조정하는 메소드입니다.

```
    else {//그외. next_offset, b_f[0] 제외
        int i;

        for (i = 0; i <= parent->num_of_keys; i++)
            if (parent->b_f[i].p_offset == deloff) break;//key를 지운 노드의 부모 key 찾기.
        //(부모 노드에서 key를 지운 노드를 가르키는 key의 index 찾기)
        neighbor_index = i - 1;//자신의 왼쪽 노드가 neighbor 노드
        neighbor_offset = parent->b_f[i - 1].p_offset;//자신의 왼쪽 노드가 neighbor 노드
        k_prime_index = i; // 부모키 설정 ( 부모 노드에서 지워진 자식 노드를 가르키던)
        k_prime = parent->b_f[i].key;
    }

    page * neighbor = load_page(neighbor_offset); //neighbor 노드(페이지) 로딩.
    int max = not_enough->is_leaf ? LEAF_MAX : INTERNAL_MAX - 1; // leaf or non_leaf 따라 max 값 지정.
    int why = neighbor->num_of_keys + not_enough->num_of_keys; //이웃 노드 key 개수+key가 지워진 노드에 남은 key 개수
                                                               //해당 합이 최대 key 수 보다 작으면 merge로 해결 ㅇㅋ
                                                               //  아니면 redistribution 필요.
    //printf("%d %dWn",why, max);

    //deloff>> 기존 key가 삭제된 트리. neighbor node>> 왼쪽
    //
    //따라서 결합할때 deloff로 가져온 페이지(노드)는 왼쪽자식,  neigbhor_off 로 가져온 건 오른쪽 자식이다.
    if (why <= max) { // 트리 merge 해야한다.
        free(not_enough);
        free(parent);
        free(neighbor);

        coalesce_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime); //트리 merge
    }
    else {    //트리 redistribute 필요
        free(not_enough);
        free(parent);
        free(neighbor);
        redistribute_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime, k_prime_index); //트리 재분배

    }

    return; //그냥 리턴하면 된다.
}
```

Key 제거했는데 최소개수(m/2 이상) 만족하지 않을 시에는 두가지 케이스로 나뉩니다.

2-1) 이웃 node(왼쪽 노드)의 key 개수 +삭제된 노드 key 개수가

최대조건(m 이하) 만족시 이웃 트리와 key가 삭제된 트리 merge(coalse) 수행

2-2) 위조건을 만족하지 않는다면 tree redistribution을 수행합니다. 이는 neighbor 에게서

가장 인접한 key를 빌려오는 과정입니다.

3) Merge

```
void coalesce_pages(off_t will_be_coal, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime) {
    //prime_index는 부모키(key가 삭제된 노드 가르키는 중).

    page *wbc, *nbor, *parent;
    off_t newp, wbf;

    if (nbor_index == -2) { //left most 와 leftmost의 오른쪽 노드 결합 예외처리
        //printf("leftmost₩n");
        wbc = load_page(nbor_off); nbor = load_page(will_be_coal); parent = load_page(par_off);
        newp = will_be_coal; wbf = nbor_off;
        //결합되어 사라질 노드( 기존 왼쪽) 이웃(오른쪽) ,부모 노드 로딩.
    }
    else {
        wbc = load_page(will_be_coal); nbor = load_page(nbor_off); parent = load_page(par_off);
        newp = nbor_off; wbf = will_be_coal;
    }
    //결합되어 사라질 노드(기존 오른쪽), 이웃(왼쪽) , 부모 노드( 페이지) 로딩.

    int point = nbor->num_of_keys; //이웃의 key 수   nbor->b_f[i].key >> i: 0~ point-1 까지 값이 들어가있다.
    int le = wbc->num_of_keys; //결합할 곳(key가 삭제된 곳) 의 key 수
    int i, j;
    if (!wbc->is_leaf) { //leaf 노드 아닐경우 부모 key를 가져와서 왼쪽자식 + 가져온 부모 key + 오른쪽 자식  구조로 결합.
        //printf("coal internal₩n");
        nbor->b_f[point].key = k_prime;//이웃노드의 끝에 k_prime 삽입.( 중간에 가져온 부모키 넣기)
        nbor->b_f[point].p_offset = wbc->next_offset;
        nbor->num_of_keys++;//key의 개수 늘린다.

        for (i = point + 1, j = 0; j < le; i++, j++) { //neighbor 노드의 끝에 wbc노드의 값을 1개씩 삽입하여 병합한다.
            nbor->b_f[i] = wbc->b_f[j];
            nbor->num_of_keys++;
            wbc->num_of_keys--;
        }

        for (i = point; i < nbor->num_of_keys; i++) {//neighbor 노드에 새로 추가된 key가 가르키는 child 들에 부모정보 업데이트
            //기존에 병합되어 사라진 노드를 부모노드로 가진 child 들이 병합되어진 neighbor 노드를 부모로 가지게 설정.
            page * child = load_page(nbor->b_f[i].p_offset);
            child->parent_page_offset = newp;
            pwrite(fd, child, sizeof(page), nbor->b_f[i].p_offset);
            free(child);
        }

    }
```

neighbor 노드(왼쪽 노드)에 key가 삭제된 노드를 merge합니다.

단 여기서 주의할 사항이 leftmost의 경우 오른쪽이 neighbor이 되게 예외처리가 되어있기에 이를 고려하여 구현되었습니다. 이후 부모에서 병합되어 사라진 노드(오른쪽 노드) 를 가르쳤던 key

제거합니다.

케이스는 두가지입니다.

3-1) internal 노드에서 merge 발생. 이때는 neighbor 노드에 부모키를 삽입 하고, 그 뒤에 쭉 key 가 삭제되어 병합될 노드들의 값을 넣어 병합합니다.

3-2) leaf 노드에서 merge 발생. 이때는 병합될 노드(오른쪽 child) 의 leftmost에 부모키에 해당하는 값이 존재하기에 부모키를 사이에 끼워줄 필요없이 neighbor노드에 key가 지워진 노드를 결합해줍니다.

```
    else { //leaf 노드일 경우  부모key와 오른쪽 자식의 key가  같기 때문에  부모 key 가져오는 과정 생략하고 원쪽 오른쪽 자식 병합
        //printf("coal leaf\n");
        int range = wbc->num_of_keys;
        for (i = point, j = 0; j < range; i++, j++) { //neighbor 노드 와 병합되어 사라질 노드를 병합한다( 1개씩 insert해서)

            nbor->records[i] = wbc->records[j];
            nbor->num_of_keys++;
            wbc->num_of_keys--;
        }
        nbor->next_offset = wbc->next_offset;//  N < wbc < wbc_next    =>     N < wbc_next
    }
    pwrite(fd, nbor, sizeof(page), newp);//수정된 정보 write

    delete_entry(k_prime, par_off);  //부모노드에서 prime_key(병합되어 사라진 노드 가르쳤던 키) 제거.
    //두개의 노드를 병합> 두 노드를 가진 부모노드에서 1개의 key를 지워야한다.  > 만약 또 문제 발생시 재귀적으로 실행된다.
    free(wbc);
    usetofree(wbf);
    free(nbor);
    free(parent);
    return; //변경된 트리 완성.
}//fin
```

두개의 노드가 merge되면, 두개 노드의 부모는 1개의 자식을 잃었기에, 부모에서 사라진 노드의 key를 제거하는 과정을 delete_entry를 재귀적으로 호출하여 수행합니다.

4) tree redistribution

```
void redistribute_pages(off_t need_sore, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime, int k_prime_index) {
    // 이웃의 rightmost key를 부모로 올리고, 부모 key를 leftmost로 가져온다.
    page *need, *nbor, *parent;
    int i;
    need = load_page(need_sore);
    nbor = load_page(nbor_off);
    parent = load_page(par_off);
    if (nbor_index != -2) {// next_offset 의 특수 케이스 제외 정상적인 경우 이웃 - 원쪽(

        if (!need->is_leaf) {// key를 오정하는 노드가 non-leaf
            //부모key 값을 넘겨주고, 이웃 노드의 인접키를 부모키로 올린다.
            //=> 부모key 값을 오른쪽 자식의 leftmost에 넘겨주고, 이웃 노드의 rightmost를 부모키로 올린다.

            for (i = need->num_of_keys; i > 0; i--) //이웃 , need    => need의 모든 key를 오른쪽으로 당기고
                need->b_f[i] = need->b_f[i - 1];

            //  next_offset, b_f[0].key, b_f[1].offset..... 구조.
            need->b_f[0].key = k_prime;// 부모키 값 맨 앞에 가져오기
            need->b_f[0].p_offset = need->next_offset; //기존 leftmost는 오른쪽으로 한칸 이동
            need->next_offset = nbor->b_f[nbor->num_of_keys - 1].p_offset; // 새로운 leftmost는 이웃의 rightmost 가르키게.

            page * child = load_page(need->next_offset);
            child->parent_page_offset = need_sore; //이웃이 건내준 child가 need 가르키게 설정.
            pwrite(fd, child, sizeof(page), need->next_offset); //write
            free(child);
            parent->b_f[k_prime_index].key = nbor->b_f[nbor->num_of_keys - 1].key; //기존 부모키의 위치에 형제키 올린다!

        }
        else {  // leaf 어차피 부모키의 값이 오른쪽 자식의 leftmost에 들어있다.
            //이웃에서 가장 인접한 키를 가져와서 넣고, 해당 key를 부모 key에도 넣는다.
            // 이웃은 원쪽 => 이웃의 rightmost  record를 leftmost에 넣고 key를 부모key에 넣는다.
            //printf("redis average leaf\n");
            for (i = need->num_of_keys; i > 0; i--){ //  이웃 , need    => need의 모든 key를 오른쪽으로 당기고
                need->records[i] = need->records[i - 1];
            }
            need->records[0] = nbor->records[nbor->num_of_keys - 1];
            //need의 leftmost(inorder successor)에 이웃의 rightmost값(inorder precedecessor) 을 넣는다.

            nbor->records[nbor->num_of_keys - 1].key = 0; // neighber노드가 준 값의 key 값 제거
            parent->b_f[k_prime_index].key = need->records[0].key; //부모키를 inorder sucessor로 변경
        }

    }
```

merge를 수행하면 최대 key 개수를 넘어버릴 경우, 형제키를 빌려옵니다.

두가지 케이스로 나뉩니다.

4-1) Internal node에서 발생: 부모key 값을 넘겨주고, 이웃 노드의 인접키를 부모키로 올린다. Leftmost의 경우를 제외하면 부모key 값을 오른쪽 자식의 leftmost에 넘겨주고, 이웃 노드의 rightmost를 부모키로 올린다.

4-2) Leaf node에서 발생: 이웃에서 가장 인접한 키를 가져와서 넣고, 해당 key를 부모 key에도 넣는다. Leftmost의 경우를 제외하고 말하면 부모키의 key값이 오른쪽 자식의 leftmost에 존재. (leaf의 leftmost는 부모key값) 이웃의 rightmost를 부모키, 맨 앞에 넣어주면 끝. 부모키에 넣는 건 key만, 맨 앞에 넣는 건 record 값.

단 leftmost의 경우는 neighbor이 오른쪽으로 예외처리 되있습니다. 따라서 좌우를 뒤집어서 그대로 적용하면 됩니다.

```
    else {//next_offset와 b_f[0] >이라는 예외케이스 이웃 = 오른쪽!

        if (need->is_leaf) {//leaf
            //printf("redis leftmost leaf₩n");
            //이웃에서 가장 인접한 키를 가져와서 넣고, 해당 key를 부모 key에도 넣는다.
        // 이웃은 오른쪽 => 이웃의 leftmost  record를 rightmost에 넣고 key를 부모key에 넣는다.
            need->records[need->num_of_keys] = nbor->records[0];  //오른쪽이 neighbor이므로 이웃의 leftmost를 맨 끝에 추가.
            for (i = 0; i < nbor->num_of_keys - 1; i++)
                nbor->records[i] = nbor->records[i + 1];// 한칸씩 당겨옴
            parent->b_f[k_prime_index].key = nbor->records[0].key;//부모에 이웃의 leftmost key값 넣기.


        }
        else { //non-leaf
            //부모key 값을 넘겨주고, 이웃 노드의 인접키를 부모키로 올린다.
            //>> 부모key 값을 왼쪽 자식의 rightmost에 넘겨주고, 이웃 노드의 leftmost를 부모키로 올린다.
            //printf("redis leftmost internal₩n");
            need->b_f[need->num_of_keys].key = k_prime;
            need->b_f[need->num_of_keys].p_offset = nbor->next_offset;
            page = child = load_page(need->b_f[need->num_of_keys].p_offset);
            child->parent_page_offset = need_more;
            pwrite(fd, child, sizeof(page), need->b_f[need->num_of_keys].p_offset);
            free(child);
            parent->b_f[k_prime_index].key = nbor->b_f[0].key;
            nbor->next_offset = nbor->b_f[0].p_offset;
            for (i = 0; i < nbor->num_of_keys - 1 ; i++)
                nbor->b_f[i] = nbor->b_f[i + 1];

        }
    }
    nbor->num_of_keys--;
    need->num_of_keys++;
    pwrite(fd, parent, sizeof(page), par_off);
    pwrite(fd, nbor, sizeof(page), nbor_off);
    pwrite(fd, need, sizeof(page), need_more);
    free(parent); free(nbor); free(need);
    return;
}
```

모든 작업이 완료되면, write하여 변경사항을 저장 후, 리턴 합니다.

Development

1) Delay merge: 키를 삭제한 뒤에 merge를 최대한 지연시켜 실질적인 merge 작업 수행횟수를 줄이는 방식입니다. key 가 delete 가 일어났을 때, merge가 필요하더라도 logical하게 했다고 가정한 뒤, 실질적인 merge는 가능한 미룹니다. search key 등 tree의 값을 가져와야 하는 경우가 발생할 때까지 실제 delete에 의한 merge를 수행하지 않고, 그 사이에 key가 지워진 노드가 merge 하지 않아도 될 정도로 insert 되면 해당 트리는 실질적인 merge를 수행하지 않고 모자란 키를 새롭게 insert받은 키로 채우면 됩니다.하지만 만약 그전에 search 등 실질적인 트리 접근 요청되어지면, 그 때 미뤄둔 실제 merge를 수행하여 올바른 값을 받아올 수 있게 합니다. 이를 통하여 실질적인 tree modification 수행횟수가 줄어들고, 이는 disk I/O 횟수 감소와 직결되어 b+ tree의 overhead를 상당 부분 감소시킬 수 있습니다.

2) cache 메모리: 어떠한 key의 위치에 대한 cache 메모리를 생성하는 것입니다. disk I/O가 발생하면 cache 메모리에 disk에서 가져온 key와 그 위치를 기록하고, 만약 특정 key를

search 하거나 제거할 때, 해당 key의 위치를 cache메모리에서 가져옵니다. 만약 key가 삭제, 변경될 경우, cache 메모리에 변경사항을 반영하고, 만약 cache메모리가 가득 차 공간을 마련해야하는 경우, LRU를 통하여 가장 재사용 가능성이 낮을 가능성이 높은 정보를 cache 메모리에서 내보냅니다. 이때, 쫓겨난 데이터는 disk에 I/O 되어야 하며 이때 모든 변경사항들을 반영해야 합니다. 이를 통해 실제 disk I/O 횟수를 감소시킬 수 있고, 짧은 시간 내에 한번 find를 수행하여 cache에 위치 정보가 들어있는 key의 경우, find함수를 통해 찾는 과정 없이 바로 위치를 받아올 수 있어 overhead가 감소합니다.