Project#2 Milestone1

On-disk B+ tree code 분석

1. INSERT

```
char buf[120];
case 'i':
                scanf("%ld %s", &input, buf);
                db_insert(input, buf);
                break;
```

‘i’를 입력할 경우, 입력한 인자와 함께 db_insert(input, buf) 함수가 실행 된다.

```
//record 구조체
typedef struct record{
    int64_t key;
    char value[120];
}record;

//Page 구조체
typedef struct Page{
    off_t parent_page_offset;
    int is_leaf;
    int num_of_keys;
    char reserved[104];
    off_t next_offset;
    union{
        I_R b_f[248];
        record records[31];
    };
}page;

//header page 구조체
typedef struct Header_Page{
    off_t fpo;              //
    off_t rpo;              //root
    int64_t num_of_pages;
    char reserved[4072];
}H_P;

page * rt = NULL; //root is declared as global
```

global로 rt 이름의 page 구조체가 선언되어 있다. rt는 root를 의미한다.

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

    record nr;
    nr.key = key;
    strcpy(nr.value, value);
    if (rt == NULL) {
        start_new_file(nr);
        return 0;
    }

    char * dupcheck;
    dupcheck = db_find(key);
    if (dupcheck != NULL) {
        free(dupcheck);
```

```
        return -1;
    }
    free(dupcheck);

    off_t leaf = find_leaf(key);

    page * leafp = load_page(leaf);

    if (leafp->num_of_keys < LEAF_MAX) {
        insert_into_leaf(leaf, nr);
        free(leafp);
        return 0;
    }

    insert_into_leaf_as(leaf, nr);
    free(leafp);
    //why double free?
    return 0;

}
```

nr 이라는 record type의 변수를 선언하고, 인자로 넣은 key와 value 값을 nr의 key와 value값에 저장한다.

만약 rt가 null 값이라면, 이 nr 정보를 rt 에 입력한다.

```
  nr.key = key;
  strcpy(nr.value, value);
  if (rt == NULL) {
      start_new_file(nr);
      return 0;
  }

void start_new_file(record rec) {

    page * root;
    off_t ro;
    ro = new_page();
    rt = load_page(ro);
    hp->rpo = ro;
    pwrite(fd, hp, sizeof(H_P), 0);
    free(hp);
    hp = load_header(0);
    rt->num_of_keys = 1;
    rt->is_leaf = 1;
    rt->records[0] = rec;
    pwrite(fd, rt, sizeof(page), hp->rpo);
    free(rt);
    rt = load_page(hp->rpo);
    //printf("new file is made\n");
}
```

rt(=root)가 이미 존재한다면 입력한 데이터가 존재하는지 중복을 확인한다.

```
char * dupcheck;
dupcheck = db_find(key);
if (dupcheck != NULL) {
    free(dupcheck);
    return -1;
}
free(dupcheck);
```

중복이 있다면, -1을 return 하여 inert를 종료하고, 존재하지 않다면 이어서 다음 명령을 수행한다.

```
off_t leaf = find_leaf(key);

page * leafp = load_page(leaf);

if (leafp->num_of_keys < LEAF_MAX) {
    insert_into_leaf(leaf, nr);
```

```
    free(leafp);
    return 0;
}

insert_into_leaf_as(leaf, nr);
free(leafp);
//why double free?
return 0;
```

key에 대한 leaf를 찾아 page leafp구조체에 leaf 정보를 담는다. leafp의 key의 갯수를 확인하여, 이 크기가 LEAD_MAX(=31) 보다 작다면 insert_into_leaf 로 insert 로 진행하고 그렇지 않다면 insert_into_leaf_as 로 insert를 진행한다. split 관련 내용은 아래의 3.SPLIT 에서 자세히 다룬다.

---

2. DELETE

main 함수에서 ‘d’를 입력하여 db_delete 를 진행합니다.

```
case 'd':
                scanf("%ld", &input);
                db_delete(input);
                break;
```

```
int db_delete(int64_t key) {

    if (rt->num_of_keys == 0) {
        //printf("root is empty\n");
        return -1;
    }
    char * check = db_find(key);
    if (check== NULL) {
        free(check);
        //printf("There are no key to delete\n");
        return -1;
    }
    free(check);
    off_t deloff = find_leaf(key);
    delete_entry(key, deloff);
    return 0;

}//fin
```

db_delete 함수에서 key 값을 인자로 받아 db_find 함수를 통해 key 가 rt에 존재하는지 확인한다. 존재하지 않다면, -1를 반환하여 삭제할 key가 없음을 나타낸다. 만약 키가 존재한다면, find_leaf 를 통해 key에 대한 leaf 정보를 받아 delete_entry 함수로 전달한다.

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

    remove_entry_from_page(key, deloff);

    if (deloff == hp->rpo) {
        adjust_root(deloff);
        return;
    }
    page * not_enough = load_page(deloff);
    int check = not_enough->is_leaf ? cut(LEAF_MAX) : cut(INTERNAL_MAX);
    if (not_enough->num_of_keys >= check){
      free(not_enough);
      //printf("just delete\n");
      return;
    }

    int neighbor_index, k_prime_index;
```

```
    off_t neighbor_offset, parent_offset;
    int64_t k_prime;
    parent_offset = not_enough->parent_page_offset;
    page * parent = load_page(parent_offset);

    if (parent->next_offset == deloff) {
        neighbor_index = -2;
        neighbor_offset = parent->b_f[0].p_offset;
        k_prime = parent->b_f[0].key;
        k_prime_index = 0;
    }
    else if(parent->b_f[0].p_offset == deloff) {
        neighbor_index = -1;
        neighbor_offset = parent->next_offset;
        k_prime_index = 0;
        k_prime = parent->b_f[0].key;
    }
    else {
        int i;

        for (i = 0; i <= parent->num_of_keys; i++)
            if (parent->b_f[i].p_offset == deloff) break;
        neighbor_index = i - 1;
        neighbor_offset = parent->b_f[i - 1].p_offset;
        k_prime_index = i;
        k_prime = parent->b_f[i].key;
    }

    page * neighbor = load_page(neighbor_offset);
    int max = not_enough->is_leaf ? LEAF_MAX : INTERNAL_MAX - 1;
    int why = neighbor->num_of_keys + not_enough->num_of_keys;
    //printf("%d %d\n",why, max);
    if (why <= max) {
        free(not_enough);
        free(parent);
        free(neighbor);
        coalesce_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime);
    }
    else {
        free(not_enough);
        free(parent);
        free(neighbor);
        redistribute_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime, k_prime_index);

    }

    return;

}
```

remove_entry_from_page 를 통해 key 값을 삭제하고 INTERNAL_MAX 값을 하나 줄여준다.

int max = not_enough->is_leaf ? LEAF_MAX : INTERNAL_MAX - 1;
int why = neighbor->num_of_keys + not_enough->num_of_keys;

이 두 변수를 비교하여, why < max이면 page를 합치는 coalesce_pages를 실행하고, 그 반대는 redistribute_pages를 진행하여 page를 분리한다.

3. SPLIT

Insert 시, leaf가 31개가 넘으면 insert_into_leaf_as 함수를 실행하여 split을 진행한다.

```
if (leafp->num_of_keys < LEAF_MAX) {
        insert_into_leaf(leaf, nr);
        free(leafp);
        return 0;
    }

    insert_into_leaf_as(leaf, nr);
    free(leafp);
```

```
    //why double free?
    return 0;
```

LEAF_MAX의 값과 현재 leafp 에 있는 key의 갯수가 같다면 insert_into_lead_as 함수에서 split을 진행한다.

insert_into_leaf_as 에서 새로만들 leaf page인 new_leaf를 만들고 넣고자하는 key 값을 비교하여 추가할 key값의 위치를 찾아 insertion_index에 위치를 저정한다.

```
off_t insert_into_leaf_as(off_t leaf, record inst) {
    off_t new_leaf;
    record * temp;
    int insertion_index, split, i, j;
    int64_t new_key;
    new_leaf = new_page();

    page * nl = load_page(new_leaf);
    nl->is_leaf = 1;
    temp = (record *)calloc(LEAF_MAX + 1, sizeof(record));
    if (temp == NULL) {
        perror("Temporary records array");
        exit(EXIT_FAILURE);
    }
    insertion_index = 0;
    page * ol = load_page(leaf);
    while (insertion_index < LEAF_MAX && ol->records[insertion_index].key < inst.key) {
        insertion_index++;
    }
```

기존의 leaf 인 ol의 값을 record 구조체의 temp 값에 insert할 index에 넣고자하는 key 값을 그 외에는 차례대로 기존의 값과 동일하게 저장한다. 그 후, split 할 값을 cut 함수를 통해 진행한다.

그 후, split 할 index 부터 leaf_max 전까지의 값을 nl page의 record 에 저장하여, ol page 와 nl page 를 구분한다. 그리고 ol page의 next_offset은 new_leaf의 값으로 저장한

```
    for (i = 0, j = 0; i < ol->num_of_keys; i++, j++) {
        if (j == insertion_index) j++;
        temp[j] = ol->records[i];
    }
    temp[insertion_index] = inst;
    ol->num_of_keys = 0;
    split = cut(LEAF_MAX);
    for (i = 0; i < split; i++) {
        ol->records[i] = temp[i];
        ol->num_of_keys++;
    }

    for (i = split, j = 0; i < LEAF_MAX + 1; i++, j++) {
        nl->records[j] = temp[i];
        nl->num_of_keys++;
    }

    free(temp);

    nl->next_offset = ol->next_offset;
    ol->next_offset = new_leaf;
```

ol 의 record 와 nl의 record 의 나머지 값을 0으로 초기화하고, nl의 record[0] 의 key 값을 parent 로 올려보내기 위해 new_key 값에 저장하고, insert_into_parent(leaf, new_key, new_leaf) 함수로, 새로 만든 leaf와 올려보낼 key 값, ol leaf와 동일한 leaf 를 인자로 넣는다.

```
    for (i = ol->num_of_keys; i < LEAF_MAX; i++) {
        ol->records[i].key = 0;
        //strcpy(ol->records[i].value, NULL);
    }
```

```
    for (i = nl->num_of_keys; i < LEAF_MAX; i++) {
        nl->records[i].key = 0;
        //strcpy(nl->records[i].value, NULL);
    }
    nl->parent_page_offset = ol->parent_page_offset;
    new_key = nl->records[0].key;

    pwrite(fd, nl, sizeof(page), new_leaf);
    pwrite(fd, ol, sizeof(page), leaf);
    free(ol);
    free(nl);
    //printf("split_leaf is complete\n");

    return insert_into_parent(leaf, new_key, new_leaf);

}
```

insert_into_parent 함수에서는 old page를 왼쪽으로 만들고 그 page의 parent_page_offset 이 0 이라면 새로운 key 값을 root 로 만든다.

```
off_t insert_into_parent(off_t old, int64_t key, off_t newp) {

    int left_index;
    off_t bumo;
    page * left;
    left = load_page(old);

    bumo = left->parent_page_offset;
    free(left);

    if (bumo == 0)
        return insert_into_new_root(old, key, newp);
```

root 가 아니라면, internal page 에 left index 값을 찾아 INTERNAL_MAX 보다 parent 의 key 수가 작아 질때 까지 반복하여 작아지면 insert_into_internal을 진행하여 넣는다. 작지 않다면 insert_into_internal_as 함수를 실행하며, 이는 위의 insert_into_leaf_as와 동일하게 동작한다.

```
    left_index = get_left_index(old);

    page * parent = load_page(bumo);
    //printf("\nbumo is %ld\n", bumo);
    if (parent->num_of_keys < INTERNAL_MAX) {
        free(parent);
        //printf("\nuntil here is ok\n");
        return insert_into_internal(bumo, left_index, key, newp);
    }
    free(parent);
    return insert_into_internal_as(bumo, left_index, key, newp);
}
```

4.MERGE

선택한 offset이 있는 key를 삭제하는 remove_entry_from_page 함수가 실행되고 삭제가 일어난 node가 root node가 아니고, 삭제가 일어난 page의 key 개수가 최소 key의 개수보다 작을 때, merge나 redistribution이 발생한다.

삭제할 offset이 있는 not_enough page 의 parent_page_offset 에 해당하는 parent page를 불러온다. 현재 page의 위치를 확인하여, 0일 경우 neighbor_index는 -1이 된다. neighbor_offset 값으로는 i-1인 왼쪽 형제의 page가 들어가는데 만약, 현재

page가 맨왼쪽에 있는 경우라면, neighbor_offset 값에는 parent ->next_offset 값이 들어가며, 부모 page에 연결된 현재 page와 다른 형제 page에 존재하는 key의 위치 값을 k_prime_index에 넣고 그 key 값은 k_prime에 저장한다. 그리고 삭제가 일어난 page와 그 형제 page의 key의 갯수를 더해 LEAF_MAX 혹은 INTERNAL_MAX-1 값과 비교한다.

```
    int neighbor_index, k_prime_index;
    off_t neighbor_offset, parent_offset;
    int64_t k_prime;
    parent_offset = not_enough->parent_page_offset;
    page * parent = load_page(parent_offset);

    if (parent->next_offset == deloff) {
        neighbor_index = -2;
        neighbor_offset = parent->b_f[0].p_offset;
        k_prime = parent->b_f[0].key;
        k_prime_index = 0;
    }
    else if(parent->b_f[0].p_offset == deloff) {
        neighbor_index = -1;
        neighbor_offset = parent->next_offset;
        k_prime_index = 0;
        k_prime = parent->b_f[0].key;
    }
    else {
        int i;

        for (i = 0; i <= parent->num_of_keys; i++)
            if (parent->b_f[i].p_offset == deloff) break;
        neighbor_index = i - 1;
        neighbor_offset = parent->b_f[i - 1].p_offset;
        k_prime_index = i;
        k_prime = parent->b_f[i].key;
    }

    page * neighbor = load_page(neighbor_offset);
    int max = not_enough->is_leaf ? LEAF_MAX : INTERNAL_MAX - 1;
    int why = neighbor->num_of_keys + not_enough->num_of_keys;
```

- 삭제가 일어난 page와 그 형제 page의 key의 갯수를 더해 LEAF_MAX 혹은 INTERNAL_MAX-1 값보다 작을 경우, coalesce_pages 함수를 실행하여 두개의 page를 merge 한다.

```
  if (why <= max) {
        free(not_enough);
        free(parent);
        free(neighbor);
        coalesce_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime);
    }
```

인자로 들어간 neighbor_index인 nbor_index 값을 확인하여 -2인 경우 가장 왼쪽에 있으므로, 삭제할 page와 neighbor을 교환한다.

```
//merge하는 함수
void coalesce_pages(off_t will_be_coal, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime) {

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

    if (nbor_index == -2) {
        //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;
    }
    else {
        wbc = load_page(will_be_coal); nbor = load_page(nbor_off); parent = load_page(par_off);
        newp = nbor_off; wbf = will_be_coal;
    }
```

merge를 진행할 때, page가 leaf인지 아닌지에 따라 진행된다. leaf가 아니라면, k_prime을 neighbor key의 맨마지막에 추가하고, 그 뒤에 병합할 page의 key를 추가한다. for 문을 진행하면서, 같은 page내에서 동일한 부모 page 를 가리키게 설정한다.

```
    int point = nbor->num_of_keys;
    int le = wbc->num_of_keys;
    int i, j;
    if (!wbc->is_leaf) {
        //printf("coal internal\n");
        nbor->b_f[point].key = k_prime;
        nbor->b_f[point].p_offset = wbc->next_offset;
        nbor->num_of_keys++;

        for (i = point + 1, j = 0; j < le; i++, j++) {
            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++) {
            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);
        }
    }
```

leaf인 경우, neighbor key 뒤에 wbc의 key를 추가하고, 한 neighbotr의 offset에는 wbc의 next_offset 값을 넣어준다. merge 과 완료된 후, 부모 page에서도 delete가 발생할 수도 있으므로, delete_entry 함수에 k_prime과 parent_offset을 인자로 넣어 실행한다.

```
    else {
        //printf("coal leaf\n");
        int range = wbc->num_of_keys;
        for (i = point, j = 0; j < range; i++, j++) {

            nbor->records[i] = wbc->records[j];
            nbor->num_of_keys++;
            wbc->num_of_keys--;
        }
        nbor->next_offset = wbc->next_offset;
    }
    pwrite(fd, nbor, sizeof(page), newp);

    delete_entry(k_prime, par_off);
    free(wbc);
    usetofree(wbf);
    free(nbor);
    free(parent);
    return;

}//fin
```

- 삭제가 일어난 page와 그 형제 page의 key의 갯수를 더해 LEAF_MAX 혹은 INTERNAL_MAX-1 값보다 클 경우, redistribute_pages 함수를 실행하여 page를 이동시킨다.

```
    else {
        free(not_enough);
        free(parent);
        free(neighbor);
        redistribute_pages(deloff, neighbor_index, neighbor_offset, parent_offset, k_prime, k_prime_index);

    }
```

위의 merge와 같은 방식으로 neighbor index가 -2, 즉 가장 왼쪽일 경우와 아닐때로 나누어진다. 가장 왼쪽이 아닐경우, page를 모두 오른쪽으로 한칸씩 이동시킨다. 그리고 need page에 leaf가 없을 경우, neighbor 의 맨 뒤에 있는 offset을 맨 앞으로 넣고 이동한 page는 부모가 원래 neighbor 이므로, 이동 후의 page를 부모 page로 변경한다. leaf가 있을 경우, 위와 마찬가지로 neighbor의 마지막 offset을 맨 앞에 넣어주고, neighbot 맨 뒤의 key도 같은 방식으로 진행한다. 그리고 부모 page의 k_prime_index 번째 key는 need의 맨 첫번째 key로 변경한다.

```
void redistribute_pages(off_t need_more, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime, int k_prime_index) {

    page *need, *nbor, *parent;
    int i;
    need = load_page(need_more);
    nbor = load_page(nbor_off);
    parent = load_page(par_off);
    if (nbor_index != -2) {

       if (!need->is_leaf) {
            //printf("redis average interal\n");
            for (i = need->num_of_keys; i > 0; i--)
                need->b_f[i] = need->b_f[i - 1];

            need->b_f[0].key = k_prime;
            need->b_f[0].p_offset = need->next_offset;
            need->next_offset = nbor->b_f[nbor->num_of_keys - 1].p_offset;
            page * child = load_page(need->next_offset);
            child->parent_page_offset = need_more;
            pwrite(fd, child, sizeof(page), need->next_offset);
            free(child);
            parent->b_f[k_prime_index].key = nbor->b_f[nbor->num_of_keys - 1].key;

        }
        else {
            //printf("redis average leaf\n");
            for (i = need->num_of_keys; i > 0; i--){
                need->records[i] = need->records[i - 1];
            }
            need->records[0] = nbor->records[nbor->num_of_keys - 1];
            nbor->records[nbor->num_of_keys - 1].key = 0;
            parent->b_f[k_prime_index].key = need->records[0].key;
        }

    }
```

neighbor_index가 -2 일 경우, page가 가장 왼쪽에 있으므로, need가 leaf가 존재할때, need의 맨 마지막에 nbor의 첫번째 key를 넣어준다. 그리고 부모 page의 k_prime_index번째 key를 이동한 후, neighbor의 첫번째 key로 설정한다. leaf가 존재하지 않을 경우, need의 key와 offset 맨마지막에는 k_prime과 nbor의 첫번째 offset을 넣어준다. 그리고 부모 page의 k_prime_index 번째 key를 neighbor의 첫번째 key로 설정하고, neighbor의 모든 key값을 for문을 통해 왼쪽으로 한칸씩 옮긴다.

모든 redistribution이 완료되면, nbor의 key수는 하나 감소, need의 key 수는 한개 증가한다.

```
    else {
        //
        if (need->is_leaf) {
            //printf("redis leftmost leaf\n");
            need->records[need->num_of_keys] = nbor->records[0];
            for (i = 0; i < nbor->num_of_keys - 1; i++)
                nbor->records[i] = nbor->records[i + 1];
            parent->b_f[k_prime_index].key = nbor->records[0].key;


        }
        else {
            //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;
}
```

Reducing Overhead in B+ Tree : Delay-Merge

delete를 진행할 때, page안의 key 개수가 MAX_LEAF의 반 이하로 존재할때만 merge가 진행된다. 하지만 merge는 전체적으로 B+ Tree의 성능을 낮출수 있다. 그러므로, merge를 최대한 실행하지 않는 방향으로 개선해야한다. 이를 위해서 Delay-Merge 방식이 있다.

이 delay-merge 방식은 delete 시, merge를 진행하지 않고, 해당 page의 key를 delete 할 수 있을만큼 다 delete한 다음에 모든 key가 삭제되어 비어있을때, 해당 page를 삭제하는 방식이다. 이를 구현하기 위해서는 delete_entry 함수에 있는 coalesce_pages와 redistribute_pages 함수를 수정하여 key의 이동이 일어나지 않도록한다.

Delete를 진행할 때, merge와 distribute를 진행하지 않고, key의 개수를 확인하여 1개라면 삭제하고 그 page를 아에 삭제한 후, internal page를 조정하는 방향으로 구현한다.