

Project #2 Final

Final Wiki

I . Design

1. 문제 상황 설정

본 과제에서는 특정한 Workload를 설정하고 이에 대한 오버헤드를 줄이는 방안을 찾아내야 한다. 해결해야 하는 문제 상황은 다음과 같이 정의하였다. 쓰기/삽입 비율이 높은 문제 상황, 즉 쓰기 집약적인 상황을 본 과제에서 설정한 workload이다. B+tree 인덱스 구조의 단점 중 하나는 임의의 쓰기/삽입으로 인해 성능이 상당히 저하될 수 있다는 점이다. 만약 쓰기/삽입이 인덱스의 정렬 순서와 일치하지 않는 순서로 수행된다고 가정하자. 그러면 각각의 쓰기/삽입 작업은 다른 leaf node에서 발생할 확률이 크고 이는 많은 양의 디스크 탐색 시간을 유발한다. 그러므로 기본적인 B+tree 구조는 많은 수의 쓰기/삽입을 지원하는 응용 프로그램에는 적합하지 않다.

2. 필요한 자료구조와 알고리즘

위와 같은 상황을 해결하기 위해 등장한 자료구조는 LSM tree이다. LSM tree의 정의는 다음과 같다. LSM tree는 몇 개의 B+tree(memory에 구현되어 있는 인메모리 트리과 디스크 상에 존재하는 트리)로 구성된다. 이 때 각 계층은 k 의 값에 따라 달라질 수 있다. Patrick O'Neil에 따르면 LSM tree의 간단한 구조로 아래의 그림과 같이 2계층 구조가 있다.

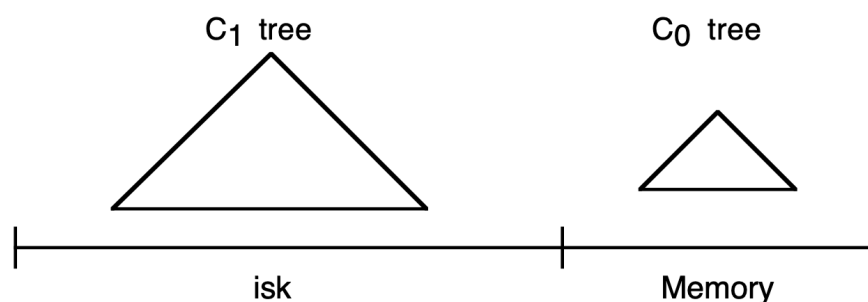


Figure 2.1. Schematic picture of an LSM-tree of two components

C_0 tree는 메모리에 구현된 tree-like 자료구조이고 C_1 tree가 실제로 disk에 저장되는 B+tree이다.

이 점에 착안하여 일종의 버퍼를 메모리에 구현하고 버퍼가 다 찰 때까지 메모리의 버퍼에서 쓰기/삽입 연산을 진행하는 자료구조를 고안하였다. 이때, 버퍼는 AVL tree로써 구현하기로 결정하였는데 그 이유는 다음과 같다.

- 쓰기 집약적인 상황에서 배열이나 리스트를 버퍼로 구현할 경우 삽입이나 삭제 연산에 많은 시간이 소요된다.
- 일반적인 이진 트리로 구현할 경우, 많은 쓰기 작업 중에 비균형적인 트리가 생길 가능성이 크다.
- 메모리에서 구동되기 때문에 I/O를 생각하지 않아도 되므로 구현이 비교적 간단한 트리를 사용한다.
- Red-Black 트리가 쓰기 속도는 AVL 트리보다 빠르지만 공간을 더 사용한다. 많은 양을 메모리에 담아두는 것이 목적이므로 AVL 트리를 선택했다.

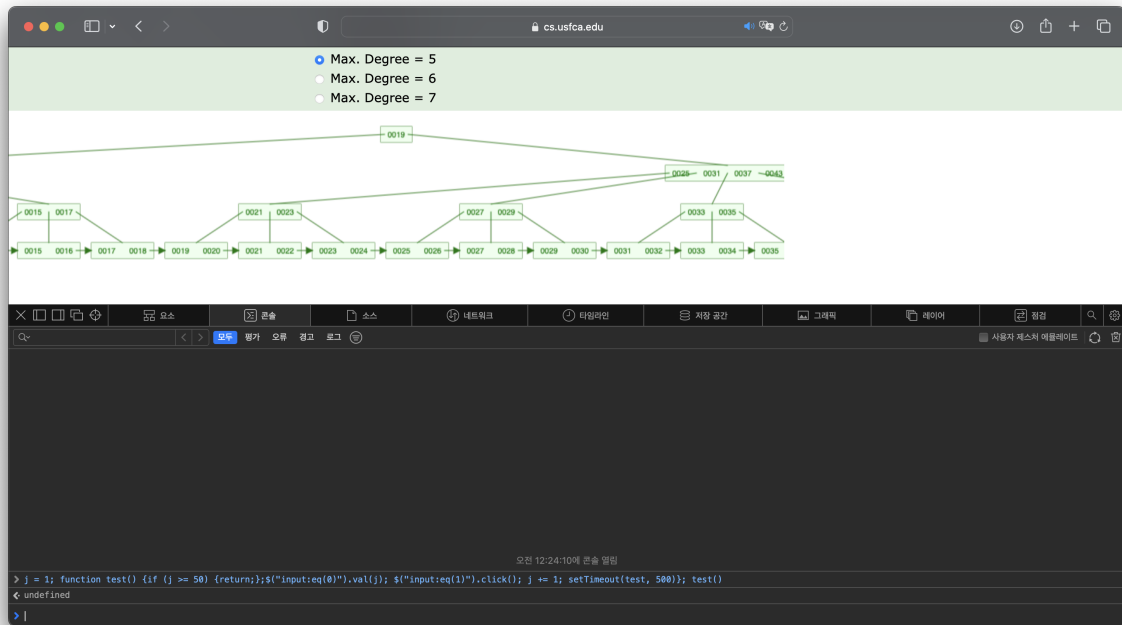
AVL tree를 버퍼로 이용한 LSM-like tree(이하 G-tree)를 고안하고 관련된 특성에 대해 고찰한 결과 다음과 같은 특성을 발견하였다. G-tree의 버퍼가 가득 차면 그 때 disk에 존재하는 C_1 tree에 삽입을 진행한다. 그렇다면 이미 쓰기 작업을 충분히 진행한 상태라고 간주할 수 있고 이 상태가 쉽게 변하지 않을 것임을 알 수 있다. 따라서 C_1 tree의 단말 노드를 제약 조건에 맞는 최대의 key를 가진 상태(order가 n 일 경우 $n - 1$ 개의 key를 가진 상태)로 만든다면 더욱 효율적인 구조가 될 것이라고 생각했다.

key 값이 정렬된 상태라면 이를 그대로 삽입할 경우 삽입은 오른쪽 단말 노드에서만 지속적으로 발생하고 한번 분할이 일어난 노드는 상태가 변하지 않는다. 따라서 상당히 비효율적이다. 이를 해결하기 위한 방안을 생각했는데 다음과 같다.

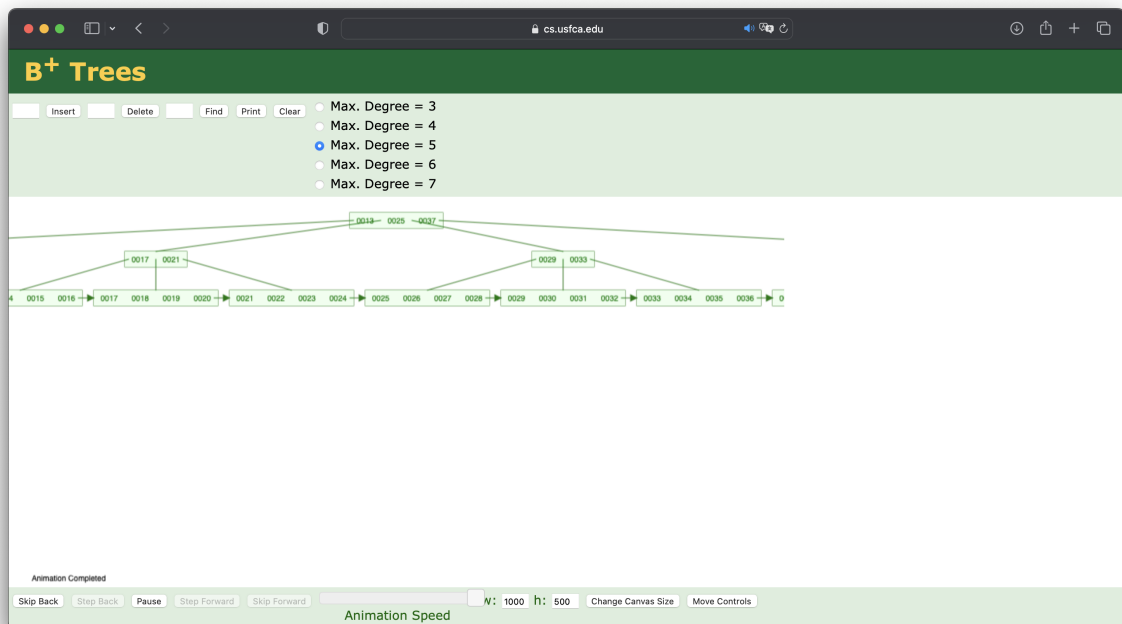
- 단조 증가하거나 단조 감소하는 정렬 상태로 삽입하는 것이 문제 상황이다.
- 따라서 홀수번째 인덱스를 순정렬한 상태로 삽입한 후, 짝수번째 인덱스를 역순정렬하여 삽입하면 각 단말 노드에서 가질 수 있는 최대의 key값을 가질 수 있다.

아래는 시뮬레이션한 결과이다.

- 일반적인 정렬된 순서로 삽입했을 경우



- 번갈아가며 삽입했을 경우



개선방안을 정리하자면 다음과 같다.

1. 쓰기 집약적인 상황에서, LSM tree 구조에서 영감을 받아 AVL tree를 버퍼로 사용하는 2계층 구조의 트리를 구현하여 많은 수의 쓰기/삽입 연산의 효율성을 높인다.

2. 버퍼에 충분한 양의 데이터가 쌓였을 때, 단순히 단조 증가하도록 정렬하여 삽입하지 않고 왕복 순환식으로 삽입하여 단말 노드의 효율성을 극대화한다. 이 상태에서 삽입이 들어가면 노드가 깨지면서 분할이 일어날 수 있지만 이미 버퍼에서 쓰기 작업을 다 완료했다고 유추할 수 있으므로 이러한 일은 잘 발생하지 않는다.

II. Implement

1. 버퍼 트리 구현 - AVL tree를 중심으로

위에서 설명한 바와 같이 버퍼로 AVL tree를 사용하였다. AVL tree에서 제일 중요한 점은 바로 rotation 작업이다. rotation은 크게 두 종류로 나뉜다. rotation을 한번 하는지, 두번 하는지로 나눌 수 있고 또 각각 왼쪽 혹은 오른쪽에서 시작하느냐로 총 4개의 rotation 수행이 가능하다. 아래는 구현한 4종류의 rotation이다.

```
Position SingleRotateWithLeft(Position node) { // Start rotation from Left
    Position k2 = node;
    Position k1 = node->left;
    k2->left = k1->right;
    k1->right = k2;

    k2->height = max(Height(k2->left), Height(k2->right)) + 1;
    k1->height = max(Height(k1->left), Height(k1->right)) + 1;

    return k1;
}

Position SingleRotateWithRight(Position node) {
    Position k2 = node;
    Position k1 = node->right;
    k2->right = k1->left;
    k1->left = k2;

    k2->height = max(Height(k2->left), Height(k2->right)) + 1;
    k1->height = max(Height(k1->left), Height(k1->right)) + 1;

    return k1;
}

Position DoubleRotateWithLeft(Position node) { // Rotate right subtree of left child
    node->left = SingleRotateWithRight(node->left);
    return SingleRotateWithLeft(node);
}

Position DoubleRotateWithRight(Position node) {
    node->right = SingleRotateWithLeft(node->right);
    return SingleRotateWithRight(node);
}
```

위의 rotation 작업이 일어날 수 있는 경우는 삽입 혹은 삭제 연산을 할 때이다. sub-tree의 height 차이가 2 이상 발생할 수 있으므로 이 두 작업을 실시한 후에는 항상 height를 검증한 후 어긋나는 경우가 있을 경우 위의 rotation 함수를 호출하는 식으로 로직을 작성하였다. 아래는 그 중 Delete 의 경우이다.

```
AVLTree Delete(int64_t key, AVLTree T) {
    if (T == NULL) {
        return T;
    } else if (key == T->element.key) { // 지워야 하는 대상을 찾은 상태
        if (T->left != NULL && T->right != NULL) { // 대상 노드가 왼쪽 오른쪽 둘 다 child가
            있을 때,
                Position tmp = FindMin(T->right);
                T->element = tmp->element;
                T->right = Delete(tmp->element.key, T->right);
        } else { // 아닐 때, : 한 쪽에만 있을 때
            Position tmp = T;
            if (T->left == NULL) { // left child가 없을 때 : right child만 있을 때
                T = T->right;
            } else {
                T = T->left;
            }
            bufferSize--;
            free(tmp);
        }
    } else if (key < T->element.key) {
        T->left = Delete(key, T->left);
        if (Height(T->right) - Height(T->left) >= 2) { // 왼쪽을 없앴으므로 오른쪽이 더 무거워
            진다.
                if (Height(T->right->right) >= Height(T->right->left)) {
                    T = SingleRotateWithRight(T);
                } else {
                    T = DoubleRotateWithRight(T);
                }
            }
        } else if (key > T->element.key) {
            T->right = Delete(key, T->right);
            if (Height(T->left) - Height(T->right) >= 2) { // 오른쪽을 없앴으므로 왼쪽이 더 무거워
            진다.
                if (Height(T->left->left) >= Height(T->left->right)) { // 왼쪽의 왼쪽이 더 무
                거움 : LL
                    T = SingleRotateWithLeft(T);
                } else { // RL
                    T = DoubleRotateWithLeft(T);
                }
            }
        }

        if (T != NULL) // 삭제한 노드가 leaf node면 height를 대입할 수 없다.
            T->height = max(Height(T->left), Height(T->right)) + 1;
        return T;
    }
}
```

버퍼가 가득 찼을 경우에 key 값이 정렬된 상태로 빠져나온 후에 그 값들이 왕복 순환을 통해 disk에 삽입된다. 이를 구현하기 위해서 AVL tree에 존재하는 값을 순회하면서 buffer 배열에 담는 방식을 택했다. 트리를 순회할 때 in-order 형식으로 순회하면 AVL tree의 특성 상 정렬된 값으로 순회할 수 있다. 구현한 코드는 다음과 같다.

```
void PopAllNode(AVLTree T) {
    if (T == NULL) return;
    PopAllNode(T->left);
    buffer[bufferSize].key = T->element.key;
    strcpy(buffer[bufferSize].value, T->element.value);
    bufferSize++;
    PopAllNode(T->right);
}
```

2. AVL tree 와 on-disk B+tree의 결합

본 단락에서는 AVL tree에서의 find, delete, insert 함수와 on-disk B+tree의 find, delete, insert 함수를 종속적으로 결합시키는 방법에 대해서 서술되어 있다. 기본적인 알고리즘은 버퍼에 우선 함수를 적용하는 것이다. 버퍼에 작용하는 함수 bf_find, bf_delete, bf_insert 함수를 만들고 AVL tree에 적용되는 함수를 먼저 실행한다. 만약 버퍼 내에서 처리가 가능한 명령이라면 버퍼 내에서만 처리를 하고 종료한다. 만약 버퍼로만 해결할 수 없는 명령이라면 on-disk B+tree의 find, delete, insert 함수를 적용하여 처리한다. 아래는 차례대로 bf_find, bf_delete, bf_insert 함수이다.

- bf_find

```
char* bf_find(int64_t key) {
    // 우선 buffer 내에 있는지 탐색
    // buffer에 존재하지 않으면 동일 key에 대해 db_find 결과를 return
    char *result = Find(key, bufferTree);
    return result ? result : db_find(key);
}
```

- bf_delete

```
int bf_delete(int64_t key) {
    if (Find(key, bufferTree)) { // 만약 버퍼 내에 있는 key 라면,
        bufferTree = Delete(key, bufferTree); // 버퍼 내에서만 지우고 종료한다.
        printf("delete operation in memory\n");
    }
```

```

        return 1;
    } else {
        printf("delete operation in disk\n"); // 버퍼 내에 없는 원소를 지우려 한다면 실제로 b+t
ree에 있을 수 있으므로
        // db_delete(key)를 실행한다.
        return db_delete(key);
    }
}

```

- bf_insert

```

int bf_insert(int64_t key, char *value) {
    int ret = -1;
    if (bufferSize < BUFFER_MAX) {
        // just put in buffer
        // 중복은 넣지 않아야 한다.
        bufferTree = Insert(key, value, bufferTree);
        bufferSize++;
        ret = 0;
    }

    if (bufferSize == BUFFER_MAX) {
        // flush to b+tree
        bf_flush();
    }
    return ret;
}

```

bf_insert 함수에서는 만약 버퍼의 크기에 여유가 있다면 AVL tree에 먼저 삽입한다. 만약 크기다 다 찼다면 버퍼를 flush해주는 bf_flush함수를 호출한 후 종료한다.

- bf_flush

```

void bf_flush(void) {
    bufferSize = 0;
    PopAllNode(bufferTree);
    DeleteTree(bufferTree);

    bufferSize--;
    int i = 0;
    while (i <= bufferSize) {
        printf("i : %d, key : %lld, value : %s\n", i, buffer[i].key, buffer[i].value);
        fflush(stdout);
        db_insert(buffer[i].key, buffer[i].value);
        i += 2;
    }
    int j = bufferSize % 2 ? bufferSize : bufferSize - 1; // 개수가 짝홀에 따라 마지막 inde

```

x가 달라짐.

```
while (j > 0) {
    printf("j : %d, key : %lld, value : %s\n", j, buffer[j].key, buffer[j].value);
    fflush(stdout);
    db_insert(buffer[j].key, buffer[j].value);
    j -= 2;
}
bufferSize = 0;
bufferTree = NULL;
}
```

bf_flush 함수는 다른 bf_ 함수와는 다르게 복잡하다. 이 함수에서 버퍼의 크기가 다 찼을 때 실제로 flush하는 역할을 담당하기 때문이다. 버퍼 역할을 하는 AVL tree 뿐만 아니라 전역변수로 buffer 배열도 존재한다. 위에서 살펴본 전체 순회 함수 (PopAllNode())를 통해 정렬된 상태로 buffer 배열에 저장된다. 이후 bufferSize에 맞추어 왕복 순회를 통해 단말 노드의 공간 효율을 최대로 할 수 있다.

- main 함수 수정

```
while(scanf("%c", &instruction) != EOF){
    switch(instruction){
        case 'i':
            scanf("%ld %s", &input, buf);
            bf_insert(input, buf);
            break;
        case 'f':
            scanf("%ld", &input);
            result = bf_find(input);
            if (result) {
                printf("Key: %ld, Value: %s\n", input, result);
            }
            else
                printf("Not Exists\n");

            fflush(stdout);
            break;
        case 'd':
            scanf("%ld", &input);
            bf_delete(input);
            break;
        case 'p':
            bf_flush();
            break;
        case 'q':
            while (getchar() != (int)'\n');
            bf_flush();
            return EXIT_SUCCESS;
            break;
    }
    while (getchar() != (int)'\n');
}
```


버퍼를 이용하기 때문에 사용자의 입력 중 'p'(push)를 추가로 받도록 구현했다. 사용자가 판단했을 때 버퍼가 다 차지 않더라도 임의로 flush 가능하도록 하였다. 또한 사용자의 실수로 버퍼에 남겨놓고 종료했을 경우에도 우선 버퍼를 다 flush하도록 구현했다.

3. 알고리즘의 정당성 증명

본 과제의 해결방안은 크게 두 가지로 나뉜다. 첫째, AVL tree를 버퍼 트리로 이용하여 쓰기 작업의 효율을 높인다. 둘째, 버퍼가 가득 찼을 때 왕복 순회를 통해 단말 노드의 크기를 최대한으로 하여 공간을 효율성을 높인다. 이 중 첫번째가 일반적인 B+tree보다 쓰기 작업에서 효율성이 높다는 것은 자명하다. 이 단락에서는 두번째 방안이 왜 단말 노드를 가득 차게 할 수 있는지 수학적으로 증명하는 과정을 담고 있다. 전제조건은 분할이 일어날 때, 비단말 노드로 올라가는 key는 분할 후 왼쪽 단말 노드에 붙지 않고 오른쪽 단말 노드에 붙는다고 가정한다.

본 증명은 두 개의 Lemma를 이용한다. 각각의 증명과정은 다음과 같다

- *Lemma 1.* 만약 degree가 $2k + 1$ ($k = 0, 1, 2, \dots$) 이라면 정렬 후 한 칸씩 띄워 삽입했을 경우, leaf node 간에 첫번째 key 값의 index 차이는 $2k$ 이다.

pf) degree가 $2k + 1$ 이라면 단말 노드 상에서 존재할 수 있는 key의 개수 l 은 $k \leq l \leq 2k$ 이다. 현재 삽입할 단말 노드의 첫번째 key를 E_i , 분할이 일어나지 않는 최대 개수인 $2k$ 개를 삽입했을 때 마지막 key를 E_j 라고 하자. $l = 2k$ 일 경우 $E_j = E_{4k+i-2}$ 이고 이때가 최대의 key를 가지는 상태이다. 이 상태에서 하나가 더 들어오면 분할이 시작된다. 즉, $l = 2k + 1$ 일때, $E_j = E_{4k+i}$ 이고 분할이 되는 대상 key는 $E_{(4k+i+i)/2} = E_{2k+i}$ 이다. 이때 E_i, E_{2k+i} 의 차이는 $2k$ 이다. ■

- *Lemma 2.* 이미 존재하는 노드들 사이에서 정렬 후 삽입이 일어나면, key의 수가 k 가 되지 않는 단말 노드의 수는 최대 양 끝 단말 노드 2개이다.

pf) 정렬한 후 삽입하므로 양 끝을 제외하고는 분할된 후에 추가적인 삽입이 일어나지 않는다. 즉, 양 끝 단말 노드에서만 key의 수가 k 가 아닐 수 있다. 이제 key의 수가 k 가 아닌 단말 노드가 2개보다 많이 존재한다고 가정하자. 그렇다면 비둘기집 원리에 의해 양끝 단말 노드를 제외하고 중간에 존재하는 단말 노드에 key의 수가 k 개가 아닌 노드가 적어도 하나 존재한다. 즉, 순서대로 삽입이 일어나지 않고 중간에 정렬되지 않은 값이 삽입되었다는 의미이다. 이는 정렬 후 삽입 된다는 가정에 모순이다. 즉, *Lemma 2.*는 참이다. ■

Lemma 1. 과 *Lemma 2.* 를 이용하면 다음과 같이 본래 증명을 완료할 수 있다.

- *Lemma 1.*을 통해 삽입 후 분할된 단말 노드에는 최대 $2k$ 개를 담을 수 있을 때, k 개를 유지하고 있음을 알 수 있고, 왕복 순회를 할 때 다시 정확히 k 개의 원소만 각각 분할된 노드들에 들어감을 증명할 수 있다. 즉, 왕복 순회를 통해 단말 노드의 key 값을 $2k$ 개로 가득 차게 할 수 있음을 증명했다.
- *Lemma 2.*을 통해 이미 노드가 존재하는 상황에서 사이에 삽입하는 경우가 발생하더라도 최대 개수를 지닌 단말 노드는 양끝 단말 노드를 제외하고는 정확히 구성될 수 있음을 증명하였다. 즉, 트리를 처음에 삽입하는 경우가 아니더라도 위 알고리즘은 정확히 동작함을 증명했다.

III. Result

G-tree의 정상 동작을 시험하기 위해서 다음과 같은 제약 조건을 설정하였다.

1. degree의 값을 홀수로 맞추기 위해서 31로 변경
2. 코드 상의 버퍼의 크기는 degree의 값의 2배로 설정하였으나 그렇다면 작동 예시가 까다로워지므로 본 단락에서만 크기를 8로 설정

- bf_insert 동작 예시

The screenshot shows a code editor with a C file named `bpt.c` containing an AVL tree implementation. The `bf_insert` function is highlighted. Below the code, a terminal window shows the compilation and execution of the program. The output displays the insertion of nodes 'a' through 'h' into the AVL tree, showing the resulting tree structure with keys and values.

```

12 // AVL tree implementation
13
14 int max(int a, int b) {
15     return a >= b ? a : b;
16 }
17
18 void bf_insert(AVL *root, int key, int value) {
19     // Insertion logic
20 }

```

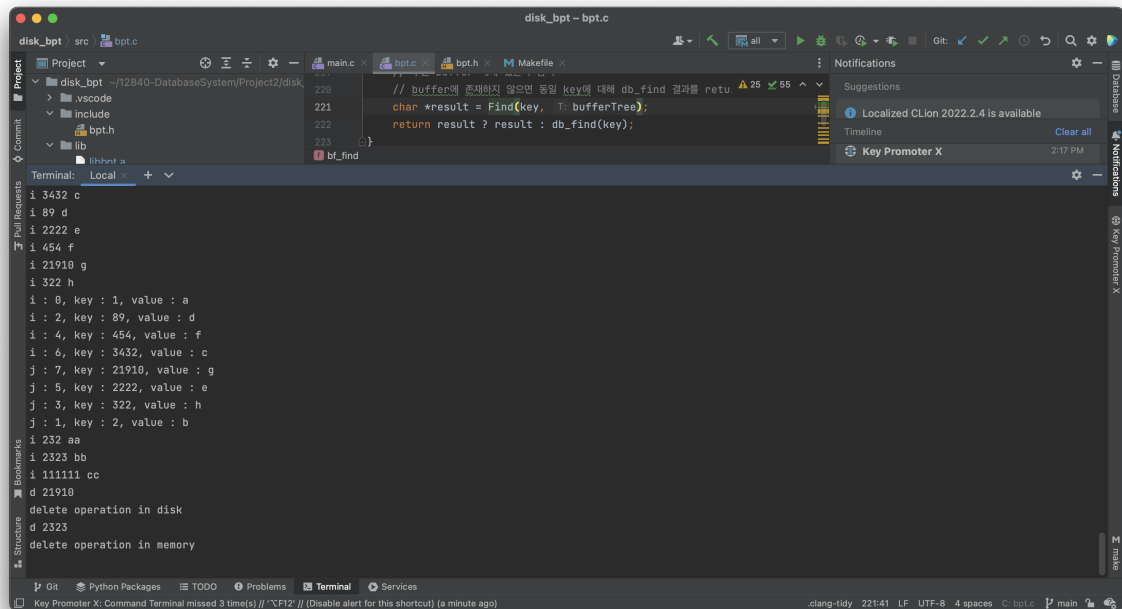
```

gcc -g -fPIC -I include/ -o src/bpt.o -c src/bpt.c
make static_library
ar cr lib/libbpt.a src/bpt.o
gcc -g -fPIC -I include/ -o main src/main.o -L lib/ -lbpt
(base) disk_bpt$ ./main
New File created
i 1 a
i 2 b
i 45 c
i 32 d
i 88 e
i 72 f
i 7 g
i 5 h
i : 0, key : 1, value : a
i : 2, key : 5, value : h
i : 4, key : 32, value : d
i : 6, key : 72, value : f
j : 7, key : 88, value : e
j : 5, key : 45, value : c
j : 3, key : 7, value : g
j : 1, key : 2, value : b

```

버퍼 크기까지 원소가 찾을 때, AVL tree 내에 존재하는 모든 원소를 on-disk b+tree에 삽입한다. in-order 탐색을 진행하므로 정렬된 값을 얻어낼 수 있고, 왕복 순회를 통해 노드를 최대 크기로 유지할 수 있다.

- bf_delete 동작 예시

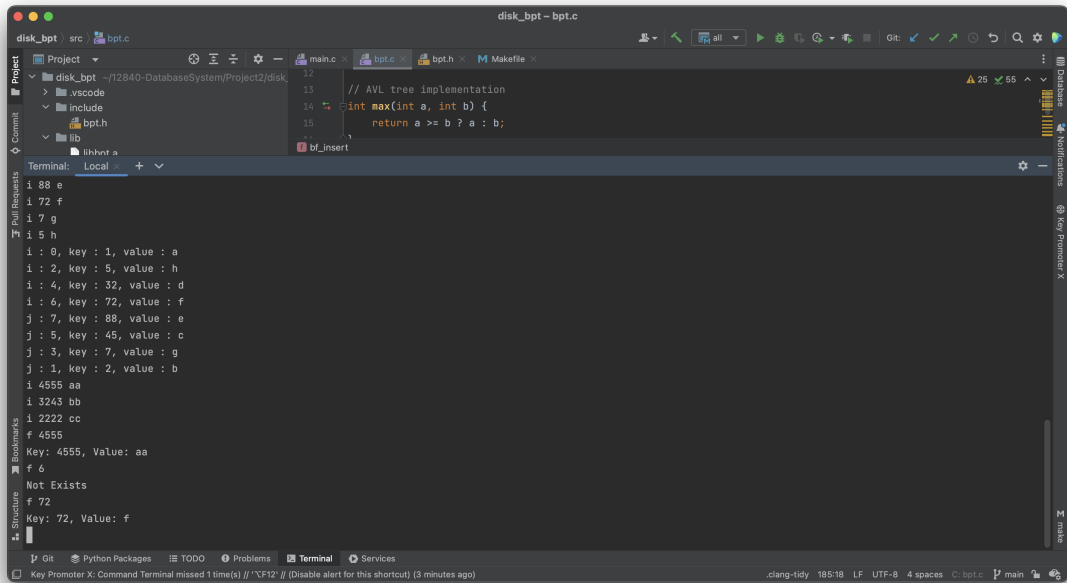


```
disk_bpt - bpt.c
Project: disk_bpt
src: bpt.c
main.c x bpt.c x Makefile x
220 // buffer에 존재하지 않으면 key에 대해 db_find 결과를 return
221 char *result = Find(key, T, bufferTree);
222 return result ? result : db_find(key);
223 }
bf_find

Terminal: Local x
i 3432 c
i 89 d
i 2222 e
i 454 f
i 21910 g
i 322 h
i : 0, key : 1, value : a
i : 2, key : 89, value : d
i : 4, key : 454, value : f
i : 6, key : 3432, value : c
j : 7, key : 21910, value : g
j : 5, key : 2222, value : e
j : 3, key : 322, value : h
j : 1, key : 2, value : b
i 232 aa
i 2323 bb
i 111111 cc
d 21910
delete operation in disk
d 2323
delete operation in memory
```

우선 버퍼에 해당하는 원소가 존재하는지 탐색하고 있다면 memory에서만 삭제 연산을 진행한다. 이 과정을 “delete operation in memory” 를 출력함으로 검증할 수 있다. 반대로 이미 disk에 삽입됐을 경우도 “delete operation in disk”로써 검증할 수 있다.

- bf_find 동작 예시

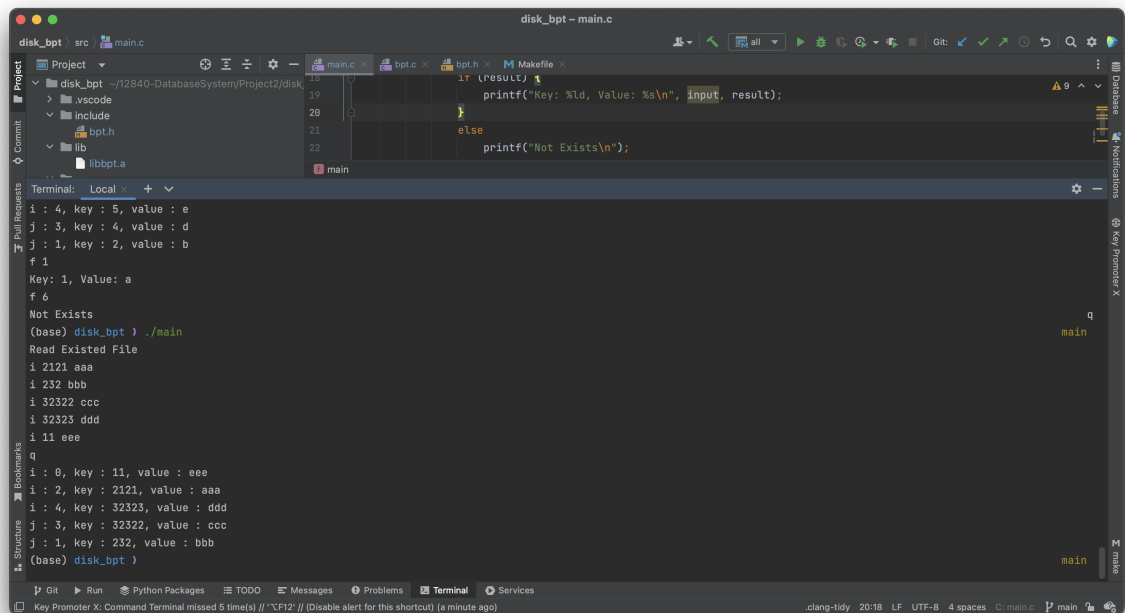


```
disk_bpt - bpt.c
12 // AVL tree implementation
13
14 int max(int a, int b) {
15     return a >= b ? a : b;
16 }
17
18 int main() {
19     // AVL tree implementation
20     // ... (code continues) ...
21 }
22
```

Terminal: Local

```
i 88 e
i 72 f
i 7 g
i 5 h
i : 0, key : 1, value : a
i : 2, key : 5, value : h
i : 4, key : 32, value : d
i : 6, key : 72, value : f
j : 7, key : 88, value : e
j : 5, key : 45, value : c
j : 3, key : 7, value : g
j : 1, key : 2, value : b
i 4555 aa
i 3243 bb
i 2222 cc
f 4555
Key: 4555, Value: aa
f 6
Not Exists
f 72
Key: 72, Value: f
```

- quit & flush 동작 예시



```
disk_bpt - main.c
28 if (result) {
29     printf("Key: %ld, Value: %s\n", input, result);
30 }
31 else {
32     printf("Not Exists\n");
33 }
34 }
35
```

Terminal: Local

```
i : 4, key : 5, value : e
j : 3, key : 4, value : d
j : 1, key : 2, value : b
f 1
Key: 1, Value: a
f 6
Not Exists
(base) disk_bpt $ ./main
Read Existed File
i 2121 aaa
i 232 bbb
i 32322 ccc
i 32323 ddd
i 11 eee
q
i : 0, key : 11, value : eee
i : 2, key : 2121, value : aaa
i : 4, key : 32323, value : ddd
j : 3, key : 32322, value : ccc
j : 1, key : 232, value : bbb
(base) disk_bpt $
```

IV. Trouble Shooting

1. 문제 해결 과정

1. AVL tree의 DeleteTree() 함수의 지속성 문제

버퍼의 크기가 다 차게 된다면 AVL tree에서 in-order 순으로 트리를 순회하며 원소를 뽑아 내야 한다. 단순히 원소에 접근하여 key, value를 버퍼에 저장하는 것뿐만 아니라 트리 자체를 destroy해야 한다. 이 destroy하는 과정을 처음에 고안했던 로직은 단순히 post-order traversal 을 사용하여 해당 원소를 free 해주는 것이었다. 하지만 처음 버퍼가 가득 찬 후에 다시 할당을 할 때 문제가 생겨 지속적인 트랜잭션을 구현할 수 없었다. free를 해도 정상작동 하지 않는 점에 착안하여 관련 문서를 찾아보니 원인은 다음과 같았다. free를 하면 단순히 포인터가 가리키던 heap의 메모리를 반납하는 것일뿐 아예 NULL을 가리키도록 하지는 않는다. 이 점에서 문제가 생겼던 것이었다. 따라서 free를 한 후에 해당 포인터를 NULL을 가리키도록 ptr = NULL; 을 추가하였더니 지속적으로 버퍼를 destroy하고 다시 construct 하는 과정이 가능해졌다.

2. 참고 문헌

Silberschatz, A., Korth, H. F., & Sudershan, S. (2019) *Database System Concepts*. MacGrawHill.

O'Neil, P., Cheng, E., Gawlick, D. *et al.* The log-structured merge-tree (LSM-tree). *Acta Informatica* **33**, 351–385 (1996). <https://doi.org/10.1007/s002360050048>