Project #2: B+ Tree Wiki

데이터베이스시스템 | 여주안 | 2017027265

1. 프로그램 실행 🚪

터미널 On Disk B+ 트리 프로그램의 경로로 이동해서 make 명령어를 실행합니다.

```
(base) juan@June disk_bpt % ls
Makefile        include         lib             src
(base) juan@June disk_bpt % make
gcc -g -fPIC -I include/   -c -o src/main.o src/main.c
src/main.c:12:33: warning: format specifies type 'long *' but the argument has
      type 'int64_t *' (aka 'long long *') [-Wformat]
                scanf("%ld %s", &input, buf);
                       ~~~      ^~~~~~
                       %lld
```

gcc make를 통해 main 파일이 새롭게 생성됩니다. 생성된 파일을 통해 프로그램을 실행합니다.

(make clean 명령어를 입력해 생성된 main 파일을 지울 수 있습니다.)

```
(base) juan@June disk_bpt % ./main
i 3 three
i 5 five
i 4 four
i 7 seven
f 5
Key: 5, Value: five
d 5
^C
```

- 데이터 삽입: “i 키(정수) 값(문자열)” 형태로 입력
- 데이터 삭제: “d 키(정수)” 형태로 입력
- 데이터 검색: “f 키(정수)” 형태로 입력 → “Key: 키, Value: 값” 형태로 출력

추가 구현 기능

- **경로 출력**: “h 키(정수)” 형태로 입력

  Leaf node에 이르기까지 경로를 순차적으로 출력

- **트리 출력(Leaf)**: “p” 형태로 입력

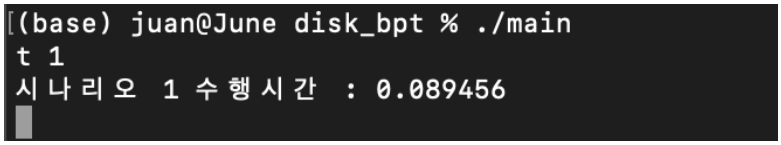
  생성된 트리를 확인할 수 있는 기능. 모든 리프 노드를 순차적으로 출력

```
p
Leaf: 1,2,3,| 4,5,6,|  Validation : OK
```

  추가 기능: 트리 구조가 올바른지 확인하는 Validation 기능

- **테스트 시나리오 자동 실행**: “t 시나리오번호(정수)” 형태로 입

  설정한 시나리오에 맞춰 여러번의 삽입, 삭제를 자동으로 실행. 프로그램 실행에 걸린 시간을 초 단위로 출력



  시나리오 목록

  - 시나리오 1 - 순차 삽입 시나리오: 1부터 1000까지 순차적으로 입력한다

```
void db_test_scenario_1() {
    int i;
    for (i = 0; i < 1000; i++) {
        db_insert(i, "value");
    }
}
```

  - 시나리오 2 - 패턴 삽입/삭제 시나리오: 정해진 패턴에 따라 삽입과 삭제를 980번 반복한다

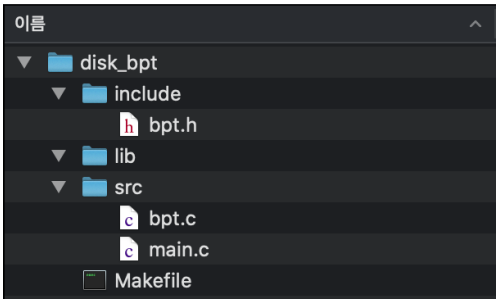
```
void db_test_scenario_2() {
    int a[14] = {1, 2, 3, 4, 5, -4, -5, 4, 5, -4, -3, 3, 4, 6};
    int i, j;

    for (j = 0; j < 70; j++) {
        for (i = 0; i < 14; i++) {
            if (a[i] > 0) {
                db_insert(a[i]+(6*j), "value");
            } else {
                db_delete(-a[i]+(6*j));
            }
        }
    }
}
```

  - 시나리오 3 - 랜덤 삽입/삭제 시나리오: 1부터 421까지의 수를 랜덤으로 삽입하고 삭제하는 것을 1000번 반복한다

```
void db_test_scenario_3() {
    int max_count = 425;
    int keys[max_count], flag[max_count], i;

    srand(1);
    for (i = 0; i < 1000; i++) {
        int r = rand() % 421 + 1;

        if (flag[r] == 1) {
            db_delete(r);
            flag[r] = 0;
            //printf("DELETE %d\n", r);
        } else {
            db_insert(r, "value");
            flag[r] = 1;
            //printf("INSERT %d\n", r);
        }
        //db_print();
    }
}
```

2. 프로그램 구조 분석 🔬

2-1. 파일 구조 Files



main.c: 사용자 입력을 받아 해당하는 동작의 함수를 bpt.c에서 호출하는 파일

bpt.c: 실질적인 검색, 삽입, 삭제 로직이 구현된 파일

On Disk B+ 트리의 기능이 구현 되었는지 분석하기 위해 **2-2**부터 **2-4**까지 **bpt.c**의 내용을 세부적으로 살핌

2-2. 구조체 Structs

- **record: 리프 노드에서 키와 값을 저장하는 구조**

  key (int64_t)

  value (char[120])
- **I_R: 중간 노드에서 키와 포인터를 저장하는 구조**

  key (int64_t)

  p_offset (off_t)
- **page: 노드에 해당하는 페이지 구조**

  parent_page_offset (off_t)

  is_leaf (int)

  num_of_keys (int)

  reserved (char[104])

  next_offset (off_t)

  b_f (I_R[248]) - 중간 노드의 경우

  records (record[31]) - 리프 노드의 경우
- **H_P: 전체 페이지를 관리하기 위한 헤더 페이지 구조**

  fpo (off_t)

  rpo (off_t)

  num_of_pages (int64_t)

  reserved (char[4072])

2-3. 변수 Variables

- **fd** (int): 파일 디스크립터
- **rt** (page *): 루트 페이지 포인터
- **hp** (H_P *): 헤더 페이지 포인터

2-4. 함수 Functions

int **open_table**(**char** * pathname);

> test.db 파일을 열고 초기 세팅을 할 때 사용되는 함수

pathname에 해당하는 파일을 열고 파일 디스크립터를 **fd**에 저장. 헤더 페이지의 초기값을 설정하고 로드하여 **hp**에 저장. 루트 페이지를 로드하여 **rt**에 저장.

H_P * **load_header**(off_t off);

> 메모리를 할당하여 헤더 페이지를 로드하는 함수

page * **load_page**(off_t off);

> 메모리를 할당하여 페이지를 로드하는 함수

off_t **find_leaf**(int64_t key);

> key 값을 (범위에) 포함하는 리프 노드를 찾아서 반환하는 함수

리프 노드에 도달할 때까지 반복 > key보다 큰 키 값이 발견될 때까지 이동. 첫번째에 걸리면 next_offset으로, 중간/끝에서 걸리면 저장된 p_offset으로 이동. 해당 offset으로 페이지를 로드한다.

char * **db_find**(int64_t key);

> B+ 트리에서 원하는 값을 검색하는 함수

main.c에서 f 입력으로 호출. 입력한 key를 포함하는 리프 노드를 찾고 해당 노드에서 value 검색하여 값이 존재할 경우 반환.

int **cut**(**int** length);

> length를 절반/절반보다 0.5 크게 으로 나누는 함수

void **start_new_file**(record rec);

> 루트 노드를 생성하는 함수 (record 값 입력)

int **db_insert**(int64_t key, **char** * value);

> B+ 트리에 새로운 key, value 쌍을 삽입하는 함수

record 객체를 만들고 루트 노드가 존재하지 않을 경우 start_new_file 실행. db_find로 키 값이 이미 존재하는지 검증. key 값에 해당하는 리프 노드를 찾고 리프 페이지를 로드.

1) 리프 페이지에 공간이 남아있는 경우 (< LEAF_MAX=31) → **insert_into_leaf** 실행

2) 리프 페이지가 꽉 찬 경우 (== LEAF_MAX=31) → **insert_into_leaf_as** 실행

off_t **insert_into_leaf**(off_t leaf, record inst);

> 리프 노드에 값을 삽입하는 함수

주어진 record의 key 보다 큰 key가 나올때 까지 반복하여 insertion_point를 찾음. insertion_point 뒤에 있는 record들을 한 칸씩 뒤로 이동. insertion_point에 주어진 record 삽입하고 num_of_keys++. 리프 노드를 파일에 업데이트.

off_t **insert_into_leaf_as**(off_t leaf, record inst);

> 리프 노드를 분할하여 값을 삽입하는 함수

new_page로 새로운 리프 노드를 생성. 주어진 record의 key 보다 큰 key가 나올때 까지 반복하여 insertion_point를 찾음. insertion_point를 제외한 모든 record를 임시 배열에 저장. 임시 배열의 insertion_point에 주어진 record 삽입. 임시 배열에 저장된 record를 원래 리프 노드와 새로운 리프 노드에 반반으로 나누어 저장. 새로운 리프 노드는 원래 next_offset을 물려받고 원래 리프 노드의 next_offset을 새로운 리프 노드로 지정. 두 리프 노드를 파일에 업데이트. 새로운 리프의 첫번째 record의 키로 **insert_into_parent** 실행

off_t **insert_into_parent**(off_t old, int64_t key, off_t newp);

> 상위 노드에 새로운 리프 노드를 삽입하는 함수

parent_page_offset이 0인 경우 새로운 루트 노드에 삽입하기 위해 **insert_into_new_root** 실행. 상위 노드에서 원래 리프 노드에 해당하는 인덱스를 찾음.

1) 상위 노드에 공간이 남아있는 경우 (< INTERNAL_MAX=248) → **insert_into_internal** 실행

2) 상위 노드가 꽉 찬 경우 (== INTERNAL_MAX=248) → **insert_into_internal_as** 실행

int **get_left_index**(off_t left);

> 리프 노드의 상위 노드에서 해당 리프 노드의 인덱스를 찾는 함수

off_t **insert_into_new_root**(off_t old, int64_t key, off_t newp);

> 두 개의 리프 노드로 새로운 루트 노드를 생성하는 함수

새로운 루트 노드를 생성하고 첫번째 record에 새로운 리프 노드의 key와 포인터를 저장. 두 리프 노드의 parent_page_offset을 루트 노드로 설정.

off_t **insert_into_internal**(off_t bumo, **int** left_index, int64_t key, off_t newp);

> 중간 노드에 새로운 리프 노드를 삽입하는 함수

left_index+1(원래 리프 노드 다음 인덱스) 뒤에 있는 record들을 한칸씩 뒤로 이동. left_index+1에 새로운 리프 노드의 key와 포인터를 삽입하고 num_of_keys++. 중간 노드를 파일에 업데이트.

off_t **insert_into_internal_as**(off_t bumo, **int** left_index, int64_t key, off_t newp);

> 중간 노드를 분할하여 새로운 리프 노드를 삽입하는 함수

new_page로 새로운 중간 노드를 생성. left_index+1를 제외한 모든 record를 임시 배열에 저장. 임시 배열의 left_index+1에 새로운 리프 노드의 key와 포인터를 삽입. 임시 배열에 저장된 record를 원래 중간 노드와 새로운 중간 노드에 반반으로 나누어 저장. 중간 key 값은 k_prime으로 지정하고 둘 중 어디에도 넣지 않고 따로 저장. k_prime에 해당하는 노드(next_offset)와 모든 자식 노드를 찾고 parent_page_offset을 새로운 중간 노드로 설정. 모든 중간 노드를 파일에 업데이트. k_prime을 키로 **insert_into_parent** 실행

int **db_delete**(int64_t key);

> B+ 트리에 주어진 key에 해당하는 데이터를 삭제하는 함수

db_find로 삭제할 키 값이 존재하는지 검증. key 값에 해당하는 리프 노드를 찾고 **delete_entry** 실행

void **delete_entry**(int64_t key, off_t deloff);

> 데이터를 삭제하고 필요할 경우 상위 노드를 조정하는 함수

deloff에서 해당 데이터 삭제하기 위해 **remove_entry_from_page** 실행. 높이 1의 구조에 루트 노드에서 삭제하는 경우 **adjust_root** 실행 후 종료.

1) 해당 노드의 키 개수가 정상적이면: 종료

2) 지정된 MAX 값의 절반보다 작을 경우: 재분배 필요 → 부모 노드에서 deloff 이전 노드(neighbor_index, neighbor_offset)을 찾음.

   1) deloff와 이웃 노드의 키 개수의 합이 MAX보다 작은 경우 → 두 노드 합치기 위해 **coalesce_pages** 실행

   2) deloff와 이웃 노드의 키 개수의 합이 MAX보다 큰 경우 → 재분배 위해 **redistribute_pages** 실행

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);

> 삭제한 데이터가 포함된 노드와 이웃 노드의 record를 재분배하는 함수

void **coalesce_pages**(off_t will_be_coal, **int** nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime);

> 삭제한 데이터가 포함된 노드와 이웃 노드의 record를 합치는 함수

void **adjust_root**(off_t deloff);

> 삭제한 데이터가 루트 노드에 포함된 경우 루트 노드를 조정하는 함수

void **remove_entry_from_page**(int64_t key, off_t deloff);

> 특정 노드에서 데이터를 삭제하는 함수

---

3. B+ 트리 Overhead 성능 개선 🚀

On Disk B+ 트리 성능 개선을 위해 다음과 같은 전략을 고안하였습니다.

1. 재분배 삽입 (Insert Redistribution)

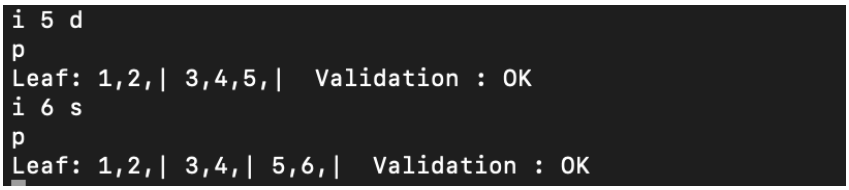
2. 인덱스 기반 Merge/재분배 (Index-Based Merge Pass)

모든 전략은 B+ 트리 프로그램을 분석한 후에 스스로 생각하여 만들었습니다.

3장에서 각각의 전략이 어떤 의미인지, 왜 타당한지 논하고 4장에서 해당 전략을 적용하였을 때 성능이 어떻게 발전했는지 다뤄보겠습니다.

전략 1: 재분배 삽입 (Insert Redistribution)

B+ 트리 프로그램을 이용하여 1부터 6까지의 숫자를 순차적으로 삽입했을 때(LEAF_MAX = 3) 다음과 같이 3개의 리프 노드가 만들어지는 것을 볼 수 있습니다. 이는 기존의 3, 4, 5가 들어있던 노드가 가득 차면서 2개로 분할된 결과입니다.



위와 같이 기존 B+ 트리 프로그램에서 숫자를 순차적으로 삽입하는 경우 각각의 노드가 가진 공간(LEAF_MAX = 3)을 모두 사용하지 못하는 경우가 빈번하게 발생하였습니다.

실제로는 1부터 6까지의 숫자를 3개씩 나눠 아래와 같이 두 개의 리프 노드에 저장할 수 있습니다. 이렇게 **리프가 가진 공간을 최대한 활용**하여 프로그램의 Overhead를 개선하는 전략으로 재분배를 통한 삽입을 구상하였습니다.

재분배 삽입 전략은 다음과 같습니다.

1. 1, 2 | 3, 4, 5 상태에서 6을 삽입하는 경우에 앞의 노드(1, 2|)에 공간이 있는지 확인합니다
2. 공간이 있으면 새로운 노드를 생성하지 않고 3을 앞의 노드로 옮기고 6을 삽입합니다 (재분배)
3. 결과적으로 1, 2, 3 | 4, 5, 6의 형태가 됩니다.

이러한 전략을 사용하게 되면 새로운 노드를 만들어 분할하는 동작을 적게 할 수 있고, 결과적으로 **노드의 수가 줄어드는 효과**가 있습니다. 노드의 수가 줄어들게 되면 I/O 횟수를 줄여 프로그램을 더욱 효율적으로 만들 수 있을거라고 생각하였습니다.

재분배 삽입 구현 1 (db_insert 함수)

db_insert 함수에서 이전 노드의 키의 개수를 구하여 재분배 삽입을 실행할지 판단하고, 이전 노드에 공간이 있는 경우 부모 노드의 k_prime_index 정보를 구해 재분배 함수를 호출하도록 구현하였습니다.

이전 노드의 정보를 알기 위해 page 구조에 이전 노드의 주소값을 저장하는 prev_offset 변수를 추가하였습니다.

```
/*
START: Insert Redistribution Code
*/
off_t prev_off = leafp->prev_offset;
page * leaf_prev = load_page(prev_off);

int prev_keys = LEAF_MAX;
if (leaf_prev != NULL && leaf_prev->num_of_keys != 0) {
    prev_keys = leaf_prev->num_of_keys;
}

page * parent = load_page(leafp->parent_page_offset);
int k_prime_index = -1;
int prev_in_range = 0;
int i;

if (parent != NULL) {
    k_prime_index = -2;
    for (i = 0; i <= parent->num_of_keys; i++) {
        if (parent->b_f[i].p_offset == leaf) {
            k_prime_index = i;
        }
        if (parent->b_f[i].p_offset == prev_off) {
            prev_in_range = 1;
        }
    }
    if (parent->next_offset == prev_off) {
        prev_in_range = 1;
    }
}

//printf("K Prime: %d, In Range: %d \n, Prev Keys: %d", k_prime_index, prev_in_range, prev_keys);
//find_leaf_path(leaf_prev->records[0].key);
free(leaf_prev);
free(parent);
if (prev_keys < LEAF_MAX && prev_in_range == 1 && k_prime_index != -2) {
    //printf("REDISTRIBUTION - record: %d \n", leafp->records[0].key);
    insert_into_leaf_redistribute(leaf, leafp->prev_offset, k_prime_index, nr);
    free(leafp);
    return 0;
}
/*
END: Insert Redistribution Code
*/
```

재분배 삽입 실행 여부를 판단하고 함수를 호출하는 부분

재분배 삽입 구현 2 (insert_into_leaf_redistribute 함수)

insert_into_leaf_redistribute 함수에서 일부 key를 앞의 노드로 옮기고 새로운 key를 삽입한 뒤, 부모 노드에 새로운 key prime 값을 업데이트 합니다. 재분배 삽입 함수는 아래와 같이 구현하였습니다. (중간 생략)

```
off_t insert_into_leaf_redistribute(off_t leaf, off_t prev_offset, int k_prime_index, record inst) {
    page * leafp = load_page(leaf);
    page * prev = load_page(prev_offset);

    record * temp;
    int insertion_index, split, k, i, j;

    temp = (record *)calloc(2 * LEAF_MAX, sizeof(record));
    ...
```

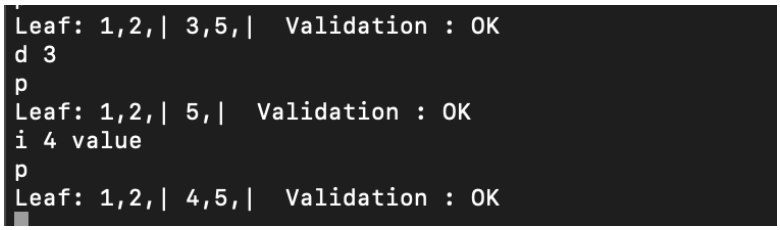
```
    insertion_index += prev->num_of_keys;
    for (i = 0, j = k; i < leafp->num_of_keys; i++, j++) {
        if (j == insertion_index) j++;
        temp[j] = leafp->records[i];
    }
    temp[insertion_index] = inst;
    ...
    split = cut(k);
    ...

    for (i = 0; i < split; i++) {
        prev->records[i] = temp[i];
        prev->num_of_keys++;
    }
    ...

    if (k_prime_index != -1) {
        page * parent = load_page(leafp->parent_page_offset);

        //printf("PARENT - record: %d", parent->b_f[k_prime_index].key);
        parent->b_f[k_prime_index].key = leafp->records[0].key;
        pwrite(fd, parent, sizeof(page), leafp->parent_page_offset);
        free(parent);
    }
    ...
    return leaf;
}
```

전략 2: 인덱스 기반 Merge/재분배 (Index-Based Merge Pass)

B+ 트리 프로그램에서 특정 데이터를 삭제하여 key의 개수가 최댓값(LEAF_MAX)의 1/2보다 작아질 경우 다음 두 동작 중 하나를 하게 됩니다. 1. 이웃한 노드와 합병한다 2. 이웃한 노드의 key를 하나 옮겨 재분배한다. 결과적으로 모든 노드 안의 데이터 개수를 절반 이상으로 유지하게 됩니다.

저는 key값, 즉 **데이터의 인덱스 값이 discrete하다는 특성을 사용하여 이 과정을 효율적으로 개선**하고자 하였습니다. 예를 들어 이전 노드의 마지막 key값이 4이고 다음 노드의 첫번째 key값이 7이라면 이전 노드에 더 들어갈 수 있는 key값은 오직 5, 6 입니다. 정수는 이산적이기 때문에 5.5, 6.7과 같은 소수값들이 중간에 들어갈 수 없습니다. 이러한 특성을 이용해 **미래에 해당 노드에 몇 개의 key값이 들어갈지 추정**할 수 있습니다.

따라서 제가 선택한 전략은 노드의 남은 공간에 남은 key값(5, 6)이 들어갈 수 있는 공간이 있다면 데이터 개수가 절반 이하로 떨어지더라도 노드를 합병하거나 재분배하지 않는 것입니다. 왜냐하면 절반 이하로 떨어졌다는 이유로 합병/재분배를 하면 미래에 5나 6과 같은 key가 추가될 경우 노드를 다시 분할하는 경우가 발생하기 때문입니다. 저는 **앞뒤 노드의 마지막과 첫번째 key값을 비교해 미래에 해당 노드에 어떤 데이터가 더 들어갈지 추정**하고 이에 맞춰 합병과 재분배를 유보하는 전략을 사용했습니다.



위와 같이 1, 2 | 3, 5 |로 나눠져있을 때 3을 삭제할 경우 1, 2, 5|로 합병하는 것을 유보하고 1, 2 | 5 |로 놓습니다. 이후에 3이나 4를 추가했을 때 별도의 분할 없이 바로 추가할 수 있습니다 (세번째 과정). 결과적으로 **합병과 분할에 소요되는 I/O를 절약**할 수 있어 프로그램 성능을 개선할 수 있습니다. 다만 이 때 최종적으로 데이터베이스에 저장되는 **인덱스 key값이 촘촘해야 추정이 정확해지고 전략이 효과적**으로 사용될 수 있습니다.

인덱스 기반 합병/재분배 구현 (delete_entry 함수)

delete_entry 함수에서 인덱스 key값의 차를 구하고 이를 check(MAX의 절반)와 비교하여 합병/재분배를 유보할지, 진행할지 판단하는 부분을 아래와 같이 구현하였습니다.

```
/*
START: Index-Based Merge Pass Code
*/
if (not_enough->is_leaf && not_enough->num_of_keys != 0) {
    int next_index = not_enough->records[0].key;
    int prev_index = neighbor->records[neighbor->num_of_keys-1].key;
    if (neighbor_index == -2) {
        prev_index = not_enough->records[not_enough->num_of_keys-1].key;
        next_index = neighbor->records[0].key;
    }
    //printf("prev index: %d, next index: %d, diff: %d, check: %d", prev_index, next_index, next_index-prev_index, check);

    if (next_index - prev_index > check) {
        free(not_enough);
        free(parent);
        free(neighbor);
        return;
    }
}
/*
END: Index-Based Merge Pass Code
*/
```

인덱스의 차로 미래에 노드에 추가로 들어갈 값을 추정하고 합병/재분배를 유보하는 부분

---

4. 성능 실험결과 ⚖️

테스트 함수를 만들어 여러번의 삽입/삭제를 시나리오에 맞춰 실시하고 소요되는 시간을 측정하였습니다.

전략 1: 재분배 삽입 (Insert Redistribution)

실험 내용: [시나리오 1] 1부터 1000까지의 숫자를 순차적으로 삽입한다 (**MAX = 3**)

- **기존 B+ 트리 프로그램**

  소요시간: **0.098468초**
- **전략 1을 적용한 B+ 트리 프로그램**

  소요시간: **0.086352초** (12.3% 단축)

실험 내용: [시나리오 1] 1부터 1000까지의 숫자를 순차적으로 삽입한다 (**MAX = 12**)

- **기존 B+ 트리 프로그램**

  소요시간: **0.048900초**
- **전략 1을 적용한 B+ 트리 프로그램**

  소요시간: **0.046755초** (4.4% 단축)

전략 1을 적용한 경우 프로그램이 더 효율적으로 동작하는 것을 확인하였습니다. 이러한 개선 방법은 노드에 저장될 수 있는 데이터의 최대수가 적을 수록 효과적입니다. MAX가 3일 때는 전략 1을 사용해서 소요 시간을 12.3% 단축했지만 MAX가 12로 커지자 4.4%만을 단축했습니다.

전략 2: 인덱스 기반 Merge/재분배 (Index-Based Merge Pass)

실험 내용: [시나리오 2] 14단계로 구성된 삽입/삭제 패턴을 값을 증가하며 70번 반복한다.

패턴: **int** pattern[14] = {1, 2, 3, 4, 5, -4, -5, 4, 5, -4, -3, 3, 4, 6} (양수는 삽입, 음수는 삭제)

- **기존 B+ 트리 프로그램**

  소요시간: **0.138633초**

- **전략 2을 적용한 B+ 트리 프로그램**

  소요시간: **0.103931초** (25.0% 단축)

- **전략 1, 2을 적용한 B+ 트리 프로그램**

  소요시간: **0.075137초** (45.8% 단축)

전략 2을 적용한 경우 소요시간이 25.0% 단축되었고 전략 1, 2를 모두 적용한 경우 45.8%가 단축되었습니다. 시간단축 폭이 큰 이유는 설정한 패턴이 전략 1, 2에 효율적으로 설계되었기 때문입니다. 결과적으로 전략 2를 사용할 경우 인덱스 값으로 미래에 값이 추가될 것을 추정하여 합병/분할에 사용되는 I/O 동작을 절약할 수 있었습니다.

⭐ 개선된 B+ 트리 프로그램 성능 분석

최종적으로 전략 1, 2를 모두 사용하여 개선한 프로그램이 실제 상황에서 어떤 성능을 내는지 확인하기 위해 랜덤으로 삽입/삭제를 반복하는 시나리오를 만들었습니다.

실험 내용: [시나리오 3] 1부터 421까지의 숫자를 랜덤으로 삽입/삭제하는 동작을 1000번 반복한다. (단, 실제 삽입/삭제 횟수는 1000보다 작다)

- **기존 B+ 트리 프로그램**

  소요시간: **0.097473초**

```
(base) juan@June disk_bpt % ./main
t 3
시나리오 3 수행시간 : 0.097473
p
Leaf: 1,5,6,| 7,9,| 11,14,| 16,17,| 18,20,| 21,23,| 25,26,27,| 28,29,30,| 31,33,
34,| 36,37,| 44,45,46,| 50,56,57,| 59,61,| 68,71,74,| 75,79,81,| 83,84,87,| 88,8
9,| 91,92,| 94,98,| 101,102,| 104,105,| 109,110,111,| 114,115,116,| 117,119,| 12
3,124,125,| 126,127,129,| 131,132,133,| 134,135,| 141,146,| 148,151,| 153,155,|
157,159,| 165,170,173,| 175,176,| 179,180,181,| 182,183,184,| 187,188,| 190,191,
192,| 193,194,197,| 200,206,207,| 208,209,210,| 213,216,| 217,219,222,| 223,225,
| 226,228,| 230,231,233,| 236,237,238,| 241,247,248,| 255,256,| 257,261,| 262,26
4,265,| 266,268,270,| 274,277,278,| 281,282,| 288,289,| 291,296,| 297,298,| 300,
301,302,| 303,305,306,| 307,308,309,| 312,313,| 317,319,| 320,321,324,| 329,330,
331,| 332,334,335,| 339,340,| 343,346,348,| 352,353,| 354,357,| 358,359,| 360,36
3,| 364,366,| 370,373,380,| 383,385,| 386,387,388,| 389,394,396,| 399,400,| 401,
402,| 403,404,| 405,406,| 408,409,| 410,411,| 416,418,| 419,420,421,|  Validatio
n : OK
```

- **개선된 B+ 트리 프로그램**

  소요시간: **0.073251초** (24.8% 단축)

```
(base) juan@June disk_bpt % ./main
t 3
시나리오 3 수행시간 : 0.073251
p
Leaf: 1,5,| 6,7,| 9,11,| 14,16,17,| 18,20,21,| 23,25,| 26,27,28,| 29,30,31,| 33,
34,| 36,37,| 44,45,46,| 50,| 56,57,59,| 61,68,71,| 74,75,79,| 81,83,84,| 87,88,|
 89,91,| 92,94,| 98,| 101,102,| 104,105,| 109,110,111,| 114,115,116,| 117,119,|
123,124,125,| 126,127,129,| 131,132,| 133,134,| 135,| 141,146,| 148,151,| 153,15
5,| 157,158,159,| 165,170,| 173,175,176,| 179,180,181,| 182,183,184,| 187,| 188,
190,191,| 192,193,194,| 197,| 200,| 206,207,208,| 209,210,| 213,216,| 217,219,22
2,| 223,225,226,| 228,230,| 231,233,| 236,237,238,| 241,| 247,248,| 255,256,| 25
7,261,262,| 264,265,266,| 268,270,| 274,277,278,| 281,282,| 288,289,| 291,| 296,
297,298,| 300,301,| 302,303,| 305,306,| 307,308,309,| 312,313,| 317,319,| 320,32
1,| 324,| 329,330,331,| 332,334,335,| 339,340,| 343,| 346,348,| 352,353,354,| 35
7,358,| 359,360,| 363,364,366,| 370,373,| 380,| 383,385,| 386,387,| 388,389,| 39
4,396,| 399,400,| 401,402,| 403,404,405,| 406,408,| 409,410,| 411,| 416,418,419,
| 420,421,|  Validation : OK
```

랜덤으로 삽입/삭제가 반복되는 환경에서 개선된 B+ 트리 프로그램은 전략 1, 2를 통해 소요시간을 24.8% 단축하였습니다.

⛈️ Project 2 요약

> 과제를 진행하면서 주어진 B+ 트리 프로그램을 이해하는데 많은 시간을 쏟았습니다. 이를 바탕으로 재분배 삽입 전략과 인덱스 기반 합병/재분배 전략을 고안하였습니다.
>
> 생각한 전략을 구현하는 과정에서 parent의 key값이 제대로 업데이트 되지 않는 등의 문제로 1, 2 | 8, 67 | 14, 15, 16 |과 같이 B+ 트리가 올바르지 않게 되는 오류를 고치는데 많은 노력이 필요했습니다.
>
> Validation과 같은 추가 기능을 함께 구현하며 프로그램을 완성했고 테스트까지 완료할 수 있었습니다. 개선된 B+ 트리 프로그램이 모든 경우에 항상 최적으로 동작할 수는 없겠지만 랜덤 테스트를 통해 최대한 실제와 가깝게 테스트하고자 했습니다.
>
> On Disk B+ 트리 프로그램을 개선하는 과정을 통해 스스로 생각하고 이해하는 경험을 많이 한 것 같습니다. 과제를 준비해주신 조교님께 감사드립니다 :)