F74109016_hw25

程式碼

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define DEGREE 3
#define ORDER (DEGREE * 2)
typedef struct node* node_t;
struct node{
    int keys[ORDER - 1];
    node_t childs[ORDER];
    int n;
    bool leaf;
};

node_t node_new(bool);
void tree_delete(node_t);
node_t search(node_t, int);
int search_index(node_t, int, int*);

void node_insert(int);
void insert_nonfull(node_t, int);
void split_child(node_t, int, node_t);

bool node_remove(node_t, int);
void remove_nonleaf(node_t, int);
void merge(node_t, int);
void borrow_prev(node_t, int);
void borrow_next(node_t, int);


node_t root = NULL;
```

```c
node_t node_new(bool leaf){
    node_t node = malloc(sizeof(struct node));
    int i;
    for(i = 0; i < ORDER; i++)
        node->childs[i] = NULL;
    node->n = 0;
    node->leaf = leaf;

    return node;
}
void tree_delete(node_t node){
    int i;
    if (node == NULL)
        return;

    if(!node->leaf)
        for (i = 0; i <= node->n; i++)
            tree_delete(node->childs[i]);

    free(node);
}
```

```c
node_t search(node_t node, int key){
    int i;
    for (i = 0; i < node->n && node->keys[i] < key; i++);

    if (node->keys[i] == key)
        return node;

    if (node->leaf)
        return NULL;

    return search(node->childs[i], key);
}
```

```c
int search_index(node_t node, int index, int* count){
    if (!node->leaf){
        int temp = search_index(node->childs[0], index, count);
        if (*count == -1)
            return temp;
        int i;
        for (i = 0; i < node->n; i++){
            if (index == *count){
                *count = -1;
                return node->keys[i];
            }
            (*count)++;

            int temp = search_index(node->childs[i + 1], index, count);
            if (*count == -1)
                return temp;
        }
    }
    else{
        if (node->n > index - *count){
            index -= *count;
            *count = -1;
            return node->keys[index];
        }
        else
            *count += node->n;
    }
    return 0;
}
```

```c
void node_insert(int key){
    if (root == NULL){
        root = node_new(true);
        root->keys[0] = key;
        root->n = 1;
        return;
    }
    if (root->n == ORDER - 1){
        node_t node = node_new(false);
        node->childs[0] = root;

        split_child(node, 0, root);

        int i = 0;
        if (node->keys[0] < key)
            i++;
        insert_nonfull(node->childs[i], key);

        root = node;
    }
    else
        insert_nonfull(root, key);
}
```

```c
void insert_nonfull(node_t node, int key){
    int i;
    if (node->leaf) {
        for (i = node->n - 1; i >= 0 && node->keys[i] > key; i--)
            node->keys[i + 1] = node->keys[i];

        node->keys[i + 1] = key;
        node->n++;
    }
    else{
        for (i = node->n - 1; i >= 0 && node->keys[i] > key; i--);

        if (node->childs[i + 1]->n == ORDER - 1){
            split_child(node, i + 1, node->childs[i + 1]);
            if (node->keys[i + 1] < key)
                i++;
        }
        insert_nonfull(node->childs[i + 1], key);
    }
}
```

```
void split_child(node_t parent, int i, node_t child){
    node_t node = node_new(child->leaf);
    node->n = DEGREE - 1;
    int j;
    for (j = 0; j < DEGREE - 1; j++)
        node->keys[j] = child->keys[j + DEGREE];

    if(!child->leaf)
        for (j = 0; j < DEGREE; j++)
            node->childs[j] = child->childs[j + DEGREE];

    child->n = DEGREE - 1;

    for (j = parent->n; j >= i + 1; j--)
        parent->childs[j + 1] = parent->childs[j];

    parent->childs[i + 1] = node;

    for (j = parent->n - 1; j >= i; j--)
        parent->keys[j + 1] = parent->keys[j];

    parent->keys[i] = child->keys[DEGREE - 1];
    parent->n++;
}
bool node_remove(node_t node, int key){
    int i;
    if (node == NULL)
        return false;

    int index;
    for (index = 0; index < node->n && node->keys[index] < key; index++);

    if (index < node->n && node->keys[index] == key){
        if (node->leaf) {
            for (i = index + 1; i < node->n; i++)
                node->keys[i - 1] = node->keys[i];
            node->n--;
        }
        else
            remove_nonleaf(node, index);
    }
    else if (!node->leaf){
        bool flag = (index == node->n) ? true : false;

        if (node->childs[index]->n < DEGREE){
            if (index != 0 && node->childs[index - 1]->n >= DEGREE)
                borrow_prev(node, index);
            else if (index != node->n && node->childs[index + 1]->n >= DEGREE)
                borrow_next(node, index);
            else{
                if (index != node->n)
                    merge(node, index);
                else
                    merge(node, index - 1);
            }
        }

        if (flag && index > node->n)
            return node_remove(node->childs[index - 1], key);
        else
            return node_remove(node->childs[index], key);
    }
    else
        return false;

    return true;
}
```

```c
void remove_nonleaf(node_t node, int index){
    int k = node->keys[index];

    if (node->childs[index]->n >= DEGREE){
        node_t temp = node->childs[index];

        while (!temp->leaf)
            temp = temp->childs[temp->n];

        node->keys[index] = temp->keys[temp->n - 1];
        node_remove(node->childs[index], temp->keys[temp->n - 1]);
    }

    else if (node->childs[index + 1]->n >= DEGREE){
        node_t temp = node->childs[index + 1];

        while (!temp->leaf)
            temp = temp->childs[0];

        node->keys[index] = temp->keys[0];
        node_remove(node->childs[index + 1], temp->keys[0]);
    }

    else{
        merge(node, index);
        node_remove(node->childs[index], k);
    }
}
```

```c
void merge(node_t node, int index){
     node_t child = node->childs[index];
     node_t sibling = node->childs[index + 1];
     int i;
     child->keys[DEGREE - 1] = node->keys[index];

     for (i = 0; i < sibling->n; i++)
         child->keys[i + DEGREE] = sibling->keys[i];

     if (!child->leaf)
         for (i = 0; i <= sibling->n; i++)
             child->childs[i + DEGREE] = sibling->childs[i];

     for (i = index + 1; i < node->n; i++)
         node->keys[i - 1] = node->keys[i];

     for (i = index + 2; i <= node->n; i++)
         node->childs[i - 1] = node->childs[i];

     child->n += sibling->n + 1;
     node->n--;

     free(sibling);
}
```

```c
void borrow_prev(node_t node, int index){
    node_t child = node->childs[index];
    node_t sibling = node->childs[index - 1];
    int i;
    for (i = child->n - 1; i >= 0; i--)
        child->keys[i + 1] = child->keys[i];

    if (!child->leaf)
        for (i = child->n; i >= 0; i--)
            child->childs[i + 1] = child->childs[i];

    child->keys[0] = node->keys[index - 1];

    if (!child->leaf)
        child->childs[0] = sibling->childs[sibling->n];

    node->keys[index - 1] = sibling->keys[sibling->n - 1];

    child->n++;
    sibling->n--;
}
```

```c
void borrow_next(node_t node, int index){
    node_t child = node->childs[index];
    node_t sibling = node->childs[index + 1];
    int i;
    child->keys[child->n] = node->keys[index];

    if (!child->leaf)
        child->childs[child->n + 1] = sibling->childs[0];

    node->keys[index] = sibling->keys[0];

    for (i = 1; i < sibling->n; i++)
        sibling->keys[i - 1] = sibling->keys[i];

    if (!sibling->leaf)
        for (i = 1; i <= sibling->n; i++)
            sibling->childs[i - 1] = sibling->childs[i];

    child->n++;
    sibling->n--;
}
```

```c
int main(){
    char s[8];
    int n, x,i;
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        scanf("%s", s);
        scanf("%d", &x);
        if (strcmp(s, "add") == 0){
            node_insert(x);
            printf("add(%d) = ok\n", x);
        }
        else if (strcmp(s, "get") == 0){
            node_t temp = search(root, x);
            if (temp == NULL)
                printf("get(%d) = not found\n", x);
            else
                printf("get(%d) = %d\n", x, x);
        }
        else if (strcmp(s, "getk") == 0){
            if (x <= 0)
                printf("getk(%d) = not found\n", x);
            else{
                int count = 1;
                int* count_ptr = &count;
                int key = search_index(root, x, count_ptr);
                if(*count_ptr == -1)
                    printf("getk(%d) = %d\n", x, key);
                else
                    printf("getk(%d) = not found\n", x);
            }

        }
        else if (strcmp(s, "remove") == 0){
            if(node_remove(root, x))
                printf("remove(%d) = %d\n", x, x);
            else
                printf("remove(%d) = not found\n", x);
        }
        else if (strcmp(s, "removek") == 0){
            if (x <= 0)
                printf("removek(%d) = not found\n", x);
            else{
                int count = 1;
                int* count_ptr = &count;
                int key = search_index(root, x, count_ptr);
                if (*count_ptr == -1 && node_remove(root, key))
                    printf("removek(%d) = %d\n", x, key);
                else
                    printf("removek(%d) = not found\n", x);
            }
        }
        else{
            printf("Wrong Instruction\n");
        }
    }
    tree_delete(root);
    return 0;
}
```

搜尋 getk

找到某個 index 的值為多少。Worst case 發生在那個 index 的是整個數最後的 index，需要走訪每個節點，計算所有節點的數量後才能找到。**時間複雜度是 O(n)。**

搜尋函數

```
int search_index(node_t node, int index, int* count){
    if (!node->leaf){
        int temp = search_index(node->childs[0], index, count);
        if (*count == -1)
            return temp;
        int i;
        for (i = 0; i < node->n; i++){
            if (index == *count){
                *count = -1;
                return node->keys[i];
            }
            (*count)++;
            int temp = search_index(node->childs[i + 1], index, count);
            if (*count == -1)
                return temp;
        }
    }
    else{
        if (node->n > index - *count){
            index -= *count;
            *count = -1;
            return node->keys[index];
        }
        else
            *count += node->n;
    }
    return 0;
}
```

如果有小孩的話，就找小孩，從第一個小孩開始找

找到了，就回傳

從找下個小孩

從如果在下個小孩有找到，就回傳

如果是葉節點，就直接算該節點的數量加上目前的數量是否大於 index，如果大於，就找到了回傳那個 index 的值

如果該節點的數量加上目前的數量是否大於 index，如果小於，將目前的數量加上該節點的是量

刪除 removek

要先找到某個 index 的值為多少，其實時間複雜度為上面函數的時間複雜度，最壞的情況為要刪除的 index 為整棵樹的最後一個 index。**時間複雜度為 O(n)。**

刪除函數:

```
168 bool node_remove(node_t node, int key){
169     int i;
170     if (node == NULL)
171         return false;
172 
173     int index;
174     for (index = 0; index < node->n && node->keys[index] < key; index++);
175 
176     if (index < node->n && node->keys[index] == key){
177         if (node->leaf) {
178             for (i = index + 1; i < node->n; i++)
179                 node->keys[i - 1] = node->keys[i];
180             node->n--;
181         }
182         else
183             remove_nonleaf(node, index);
184     }
185     else if (!node->leaf){
186         bool flag = (index == node->n) ? true : false;
187 
188         if (node->childs[index]->n < DEGREE){
189             if (index != 0 && node->childs[index - 1]->n >= DEGREE)
190                 borrow_prev(node, index);
191             else if (index != node->n && node->childs[index + 1]->n
192                 borrow_next(node, index);
193             else{
194                 if (index != node->n)
195                     merge(node, index);
196                 else
197                     merge(node, index - 1);
198             }
199         }
```

如果找到節點，node 是葉節點，就直接將 key 的值往前移

Index 等於 node 值的數量或是在這個 node 沒有找到值

小孩不滿足 degree 的規則

如果前面可以合併就跟前面的節點合併

前面不行後面可以合併就跟後面的節點合併

Sibling 都不行合併就跟 parent 合併。

```
200 
201         if (flag && index > node->n)
202             return node_remove(node->childs[index - 1], key);
203         else
204             return node_remove(node->childs[index], key);
205     }
206     else
207         return false;
208 
209     return true;
210 }
```

找要 remove 的節點是在哪裡。

```
212 void remove_nonleaf(node_t node, int index){
213     int k = node->keys[index];
214
215     if (node->childs[index]->n >= DEGREE){
216         node_t temp = node->childs[index];
217
218         while (!temp->leaf)
219             temp = temp->childs[temp->n];
220
221         node->keys[index] = temp->keys[temp->n - 1];
222         node_remove(node->childs[index], temp->keys[temp->n - 1]);
223     }
224
225     else if (node->childs[index + 1]->n >= DEGREE){
226         node_t temp = node->childs[index + 1];
227
228         while (!temp->leaf)
229             temp = temp->childs[0];
230
231         node->keys[index] = temp->keys[0];
232         node_remove(node->childs[index + 1], temp->keys[0]);
233     }
234
235     else{
236         merge(node, index);
237         node_remove(node->childs[index], k);
238     }
239 }
240
```

非葉節點刪除某個 key 值，就要移動 child

將 node 最後一個值(最大值)刪掉，因為他已經到 parent 節點去了

將 node 最後第一個值(最小值)刪掉，因為他已經到 parent 節點去了

如果都不大於 degree 就直接合併了。

刪除動作的 Algorithm:

1.首先查詢 Btree 中需要刪除的元素。上面的搜尋函數。找到就到第二步驟。

2.刪除後，判斷該元素是否有左右小孩節點，如果有就將左小孩最大的值或右小孩小的值拉到現在被刪除的元素中。

如果某節點中的元素小於 degree/2-1，則看看 sibling 是否滿。

如果不滿就跟 sibling 合併。

如果滿就跟父節點借一個元素來放入被刪除的元素李。

Remove 的最壞情況就是一直不符合要求，需要跟上面的不斷合併。如例子中刪 5 的情況。其時間複雜度為樹的高度。

Btree 樹的高度:

根結點至少兩個 child，第二層有一兩個節點。

第三層至少有 2*((節點最大數量/2)取天花板)

第四層至少有 2*((節點最大數量/2)取天花板)^2

第 k 層至少有 2*((節點最大數量/2)取天花板)$^{k-2}$

如果 Btree 有 N 個 key 值，則數的 leaf 節點可以為 N+1 個，leaf 都在 k 層

N+1>=2*((節點最大數量/2)取天花板)$^{k-2}$

➔k<=log(((節點最大數量/2)取天花板)*(N+1)/2)+1

此為 remove 動作 worst case 的時間複雜度。

log(((節點最大數量/2)取天花板)*(N+1)/2)+1 < O(N) (找 index 的值)

所以 remove 的 worst case 時間複雜度也為 O(N)

刪除某個 key 值要一直合併的例子(最後刪 5)