

Home work #2 B10815057 廖聖郝

1. Thanos Finger Snap

BST tree 的節點 class，包含父節點指標與左右小孩的指標，並重載了一系列的大於小於符號，方便之後程式撰寫，因為整棵樹的節點都是動態配置記憶體出來的，所以用一個 function 來遞迴釋放記憶體。

```
template<typename T>
class BST_node { //BST節點的class
public:
    BST_node() {}

    BST_node(T _data):data(_data) {}

    bool operator>(const BST_node& i) { return data > i.data; }
    bool operator>(const T& i) { return data > i; }
    bool operator>=(const BST_node& i) { return data >= i.data; }
    bool operator>=(const T& i) { return data >= i; }
    bool operator<(const BST_node& i) { return data < i.data; }
    bool operator<(const T& i) { return data < i; }
    bool operator<=(const BST_node& i) { return data <= i.data; }
    bool operator<=(const T& i) { return data <= i; }

    void del() { //遞迴刪除所有子孫節點，最後刪除自己
        if (left != nullptr) {
            left->del();
        }
        if (right != nullptr) {
            right->del();
        }
        delete this;
    }

private:
    T data;
    BST_node<T>* father = nullptr; //爸爸
    BST_node<T>* left = nullptr; //左邊的小孩
    BST_node<T>* right = nullptr; //右邊的小孩
    template<typename> friend class BST;
};
```

接下來是 BST tree 的 class，成員有根節點指標，與插入、刪除 function...，為了保持物件封裝性，所以在 class 的 public 處都只有簡單的介面與防呆，真正的實作會放在 private 裡(重載函式)。

insert 資料進去 BST 內有一個點要注意的就是 root 為 null 時必須配置記憶體，而不能直接呼叫遞迴 insert。

```
template<typename T>
class BST { //BST的class
private:
    BST_node<T>* root = nullptr; //根節點的指標
public:
    BST() {}
    ~BST() { //釋放記憶體
        root->del();
        root = nullptr;
    }
    void insert(T new_data) { //這個是從根結點出發的insert，另有重載一個可從任意指標insert的遞迴function
        if (root == nullptr) { //如果甚麼都沒有
            root = new BST_node<T>(new_data); //配置記憶體並賦值
        }
        else {
            if (*root > new_data) { //新資料較小，往左走
                insert(&root->left, new_data, root); //呼叫真正的遞迴insert
            }
            else { //新資料較大，往右走
                insert(&root->right, new_data, root); //呼叫真正的遞迴insert
            }
        }
    }
    void erase(T data) { //呼叫search，找到指標後，呼叫erase的重載function
        erase(search(data));
    }
    //以下函式是為了簡化介面，真正實作的function重載在private裡
    BST_node<T>* search(T data) { return search(root, data); } //搜尋節點，回傳目標節點指標
    BST_node<T>* largest() { return largest(root); } //回傳樹最大節點的指標
    BST_node<T>* smallest() { return smallest(root); } //回傳樹最小節點的指標
    void pre_order_output() { pre_order_output(root); } //輸出前序
    void in_order_output() { in_order_output(root); } //輸出中序
    void post_order_output() { post_order_output(root); } //輸出後序
    int height() { return height(root); } //回傳樹高
```

遞迴 insert，這個 function 不能只傳入指標，而是要傳入指標的指標，否則配置記憶體時給予到的指標會是函式呼叫產生的新指標，而不是我們想要的節點內的指標。

根據 BST 的規則，新資料較小，就往左走，較大反之，一直走到 leaf node 後 insert 在 leaf node 之下，所以當遞迴走到 nullptr 時，就配置記憶體，產生新節點。

```
//使用指標的指標才能在下次遞迴中配置記憶體並連接上一個node
void insert(BST_node<T>*& now, T new_data, BST_node<T>* father) {

    if (*now == nullptr) { //如果甚麼都沒有

        *now = new BST_node<T>(new_data); //配置記憶體並賦值

        (*now)->father = father;

    }

    else {

        if (**now > new_data) { //新資料較小，往左走

            insert(&(*now)->left, new_data, *now); //遞迴

        }

        else { //新資料較大，往右走

            insert(&(*now)->right, new_data, *now); //遞迴

        }

    }

}
```

以下是幾個之後會用到的函式，都是用遞迴實作，原理已寫成註解。

```
BST_node<T>* search(BST_node<T>* now, T data) { //搜尋節點

    if (now == nullptr) return nullptr;

    if (*now > data) return search(now->left, data); //要找的資料小於目前節點的資料，往左走

    else if (*now < data) return search(now->right, data); //要找的資料大於目前節點的資料，往右走

    else return now; //既非大於也非小於，那就是等於

}

BST_node<T>* largest(BST_node<T>* now) { //回傳now以下最大值的節點，一直往右走即可

    if (now == nullptr) return nullptr;

    else if (now->right == nullptr) return now;

    return largest(now->right);

}

BST_node<T>* smallest(BST_node<T>* now) { //回傳now以下最小值的節點，一直往左走即可

    if (now == nullptr) return nullptr;

    else if (now->left == nullptr) return now;

    return smallest(now->left);

}
```

前面的`void erase(T data) { erase(search(data)); }`，先使用`search`找到指標，然後交給下面的`erase function`處理。

刪除節點可分為 2 種情形處理：

- (1) 0 或 1 個小孩，把替代指標(replacement)設為存在的小孩或`null`(0 個小孩時)，判斷是否為`root`(沒爸爸)，是就特別處理後直接結束，一般狀況就將父節點指向自己的指標更新為替代指標，若小孩存在，就把小孩的父節點指標設為自己的爸爸。
- (2) 2 邊都有小孩，就取左子樹中最大值，替代自己原本的值，然後刪除那個節點(遞迴)。

```
void erase(BST_node<T>* target) { //傳入節點指標，刪除該節點

    if (target == nullptr) return; //防呆

    int child_num = (target->left != nullptr) + (target->right != nullptr);
    //計算小孩數量，nullptr代表沒小孩，回傳0，反之回傳1，因此這2個判斷式相加就是小孩數量

    switch (child_num)
    {
        case 0: case 1: { //0個或1個小孩

            BST_node<T>* replacement = target->left != nullptr ? target->left : target->right;
            //替代節點指標，若2邊為空，則為空，若左邊為空，代表右邊有小孩，若右邊為空，代表左邊有小孩

            if (target->father == nullptr) { //如果是根結點，刪除後設root為替代節點指標

                delete target;

                root = replacement;

                return;

            }

            if (target->father->left == target) { //如果自己是爸爸的左小孩

                target->father->left = replacement; //爸爸左指標設為替代節點指標

            }

            else { //如果自己是爸爸的右小孩

                target->father->right = replacement; //爸爸右指標設為替代節點指標

            }

            if (replacement != nullptr) replacement->father = target->father;
            //如果替代節點指標不為空，將小孩的爸爸設為自己的爸爸

            delete target; //刪除目標

        } break;

        case 2: { //左右都有小孩

            BST_node<T>* replacement = largest(target->left); //找左子樹中最大

            target->data = replacement->data; //用左子樹中最大值替代原本的值

            erase(replacement); //刪除左子樹中最大值的node

        } break;

        default: break;

    }

}
```

用一條水平線切開 BST 樹，切後的節點數要最接近原本的一半，一開始先建立儲存每層數量的陣列 `layer_count`，然後用遞迴 function：

`get_layer_number_of_node`，去找到每層的節點數，接下來，跑過每種水平線，算出哪條水平線最接近我們要的，找到後水平線後，建立一個森林，存放所有切開後下半部產生的樹，我把切開的實作與建立森林寫在同一個遞迴 function: `cut_layer_and_create_forest`，做完後，輸出樹切開後的中序，與森林中每棵樹的前序，最後再將森林內的每棵樹釋放記憶體。

```
void cut_half() { //Thanos Finger Snap 找到最接近切半的水平線後，將水平線以下node全部刪除

    int H = height(); //樹高(有幾層)

    int* layer_count = new int[H] {}; //動態配置陣列，儲存每層的node數

    get_layer_number_of_node(root, layer_count, 0); //遞迴計算每層的node數，結果存於layer_count

    int min_diff = 2147483647; //水平線上數量與線下數量差，一開始先設很大，避免被超過

    int cut_layer = 0; //儲存最佳的水平線(層數)

    for (int i = 0; i < H; i++) { //跑過每條水平線，找最好的

        int up = 0, down = 0; //分別存水平線上數量與線下數量

        for (int j = 0; j < H; j++) { //掃過每層，把數量加到up或down

            (j <= i ? up : down) += layer_count[j]; //數量加總 <= i 屬於上半層，反之下半層

        }

        if (abs(up - down) < min_diff) { //若數量差小於之前所偵測到的最小差

            min_diff = abs(up - down); //更新最小數量差

            cut_layer = i; //更新最佳水平線

        }

    }

    BST_node<T>** forest = new BST_node<T> * [layer_count[cut_layer + 1]]{}; //配置森林記憶體，
    共有(水平線+1)層之節點數

    int forest_index = 0; //用一個變數才能call by reference，存現在種了幾棵樹

    cut_layer_and_create_forest(root, forest, forest_index, cut_layer + 1, 0); //切除水平線下節
    點並存到森林中，遞迴函式

    in_order_output(); //切完後，輸出中序

    cout << endl;

    for (int i = 0; i < layer_count[cut_layer + 1]; i++) { //跑過森林裡的每棵樹

        pre_order_output(forest[i]); //輸出前序

        forest[i]->del(); //刪除此樹

    }

    delete[] layer_count; //釋放每層的node數陣列

    delete[] forest; //釋放森林

}
```

樹高函式:

```
int height(BST_node<T>* now) { //回傳now的高度，leaf node為高度為1

    if (now == nullptr) return 0;

    int LH = height(now->left);

    int RH = height(now->right);

    return (LH > RH ? LH : RH) + 1; //左右子樹高度選大的並加上自己(1)

}
```

遞迴計算 BST 樹中每層的節點數，結果存到 layer_count 裡

```
void get_layer_number_of_node(BST_node<T>* now, int* layer_count, int layer) { //計算每層node數

    if (now == nullptr) return; //防呆

    layer_count[layer]++; //該層node數加1

    get_layer_number_of_node(now->left, layer_count, layer + 1);

    //往左遞迴下去，下一層為layer+1

    get_layer_number_of_node(now->right, layer_count, layer + 1);

    //往右遞迴下去，下一層為layer+1

}
```

切割與建立森林函式，會先遞迴到要切割的地方，指標存入森林，然後更改父節點內指到自己的指標為 null(切除)

//now存現在遞迴到哪個節點，forest存切除後的樹，forest_index存現在是第幾棵樹，target_layer為(最佳水平線+1)層，now_layer為now所在層數(root為0層)

```
void cut_layer_and_create_forest(BST_node<T>* now, BST_node<T>** forest, int& forest_index, int target_layer, int now_layer) {

    if (now == nullptr) return; //防呆

    if (now_layer == target_layer) { //遞迴到正確層數

        forest[forest_index] = now; //切除後的樹存入森林

        forest_index++; //下一棵樹，因為是call by reference，所以同層級的遞迴也能被更改到

        if (now->father->left == now) { //如果自己是爸爸的左小孩

            now->father->left = nullptr; //爸爸的左指標設為空，切除

        }

        else { //如果自己是爸爸的右小孩

            now->father->right = nullptr; //爸爸的右指標設為空，切除

        }

    }

    else {

        cut_layer_and_create_forest(now->left, forest, forest_index, target_layer, now_layer + 1); //從左邊開始掃，所以先遞迴左邊

        cut_layer_and_create_forest(now->right, forest, forest_index, target_layer, now_layer + 1); //遞迴右邊

    }

}
```

前中後，三序的差異就只是遞迴與輸出的先後順序不同，因為最後一個輸出不能加上空白，但這件事有點難在遞迴裡做到，所以我使用讓每個遞迴可共用的 `static` 變數，來記錄現在是不是第一次輸出，如果是就不在輸出 `data` 前加上空白。

```
void pre_order_output(BST_node<T>* now) { //前序輸出，輸出該節點，然後繼續遞迴
    if (now == nullptr) return;
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
    pre_order_output(now->left); //遞迴左子樹
    pre_order_output(now->right); //遞迴右子樹
}

void in_order_output(BST_node<T>* now) { //中序輸出，先遞迴左子樹，然後輸出該節點，遞迴右子樹
    if (now == nullptr) return;
    in_order_output(now->left); //遞迴左子樹
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
    in_order_output(now->right); //遞迴右子樹
}

void post_order_output(BST_node<T>* now) { //後序輸出，遞迴左子樹與右子樹，然後輸出該節點
    if (now == nullptr) return;
    post_order_output(now->left); //遞迴左子樹
    post_order_output(now->right); //遞迴右子樹
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
}

}
```

處理輸入與主流程的 `main` 函式:

```
int main() {
    BST<int> tree;
    string line; //存一行
    int data; //暫存數字
    getline(cin, line);
    stringstream s(line);
    while (s >> data) { //用stringstream讀數字
        tree.insert(data); //插入所有數字
    }
    tree.post_order_output(); //輸出後序
    cout << endl;
    getline(cin, line);
    s.clear();
    s.str(line);
    while (s >> data) { //用stringstream讀數字
        tree.erase(data); //刪除Avengers
    }
    tree.cut_half(); //刪除一半node後輸出上半部(中序)、下半部(前序)
    return 0;
}
```

2. 2-3 Tree

因為 2-3 樹就只是 order 為 3 的 B 樹，所以我直接實作出 B 樹，以增加未來的擴充性，以下是 B 樹節點的 class，有儲存父節點指標的 father、child 小孩指標陣列、data 資料陣列、data_num 目前節點內資料數量，2 個主要的陣列都多了一個位置，這是因為爆掉時的大小會比最大空間還多 1 格，有了這個空間，爆掉時才可以更好的整頓節點，不然會很難實作。

```
template<typename T,unsigned int order>
class B_tree_node { //B樹節點
public:
    //constructor
    B_tree_node() {set_null();} //啥都沒
    B_tree_node(B_tree_node<T, order>* f) : father(f) {set_null();} //有給爸爸
    B_tree_node(T new_data) { //有給資料
        set_null();
        data[0] = new_data;
        data_num = 1;
    }
    B_tree_node(T new_data, B_tree_node<T, order>* f): father(f){ //有給爸爸跟資料
        set_null();
        data[0] = new_data;
        data_num = 1;
    }

    B_tree_node<T, order>* father = nullptr; //爸爸指標，爆掉時找爸爸求救用
    B_tree_node<T, order>* child[order + 1]; //放小孩指標，多一個位置用於放置爆掉時多出來的東西
    T data[order]; //放資料，多一個位置用於放置爆掉時多出來的東西
    int data_num = 0; //資料數量 指標數 == 資料數量 + 1
```

```
    void set_null() { for (int i = 0; i <= order; i++) child[i] = nullptr; } //所有小孩設成空
```

main 函式，template 的第二個參數填 order，因為是 2-3 樹，所以填 3

```
int main() {
    B_tree<int,3> tree; //2 3樹就是order為3的B樹
    int new_data; //暫存每個數字的變數
    while (cin >> new_data) {
        tree.insert(new_data); //插入
    }
    tree.output(); //輸出整棵樹
    return 0;
}
```


B 樹的 insert 都是先插到 leaf node 裡，如果超過最大數量就會切割 node，並將中間值往上丟，所以我實作了 2 種 insert function，一種是 leaf_insert，專門處理新資料的插入，這個函式會遞迴的往正確的 leaf node 走，最後呼叫另一個 insert function 插入到 leaf node 裡，這個 insert 是對某個 node 去做插入資料用的，node 爆掉時也會使用到。

```
void leaf_insert(T new_data) {  
    //B樹的insert都是從leaf node開始，爆掉了再往上長，所以這個function用於新資料的insert，是專  
    屬於整棵樹的insert function  
    //B樹leaf node以外的node都至少有一個小孩，所以拿0號小孩指標確認有無小孩  
    if (child[0] != nullptr) { //不是leaf node，遞迴找到leaf node再用普通的insert  
        for (int i = 0; i < data_num; i++) {  
            if (new_data < data[i]) { //找到第一個大於的資料，就是插入後要放的地方  
                child[i]->leaf_insert(new_data); //遞迴下一個節點  
                return; //結束掉，不然會跑到下面那行  
            }  
        }  
        child[data_num]->leaf_insert(new_data);  
        //上面都沒進入遞迴就會跑到這行，代表新資料大於node裡的所有資料，找最右邊的指標遞迴  
    }  
    else { //是leaf node  
        insert(new_data, nullptr);  
        //經過許多遞迴後成功找到leaf node，這是專屬於node的insert function，由於是新資料所以  
        沒有分割後的指標，填null就好  
    }  
}
```

接下來這個遞迴函式就是對 node 插入資料，第二個參數是在節點爆掉時產生的新節點指標，會這樣做是因為中間值與新節點指標是相鄰的，一起插入比較方便，這個函式中，會需要處理一個特例，也就是 root node 爆掉時產生的新 root node，因為這時候裡面甚麼資料都沒有，所以要特別去處理，還有一個狀況就是新資料大於現有的所有資料，此時就要插到最後面，插入完資料後就檢查是否爆掉，如果爆掉的話就找爸爸來分割，沒爸爸(root node)，就產生一個新 root node，這種狀況就是上面提到的特例，下面就是這個 function 的實作程式碼。

```

void insert(T new_data, B_tree_node<T, order>* split_node) {

    //第一個參數是新資料，第二個參數是當爆掉時node分割後的指標(右邊那側的分割)

    //特例: 新root node

    if (data_num == 0) { //如果資料數為0，代表這是原root node爆掉後產生的新root node

        data[0] = new_data;

        child[1] = split_node; //child[0]是分割後左側(在下面的程式碼賦值)，child[1]是分割後右側

        data_num = 1;

        return;

    }

    int i;

    for (i = 0; i < data_num; i++) {

        if (new_data < data[i]) { //找到自己的位置

            child[order] = child[order - 1];

            //因為要插入新資料，把舊資料往後移，但指標多一個，所以額外處理最後一個指標

            for (int j = order - 1; j > i; j--) {

                data[j] = data[j - 1]; //把舊資料往後移

                child[j] = child[j - 1]; //把舊指標往後移

            }

            data[i] = new_data; //插入

            child[i] = child[i + 1]; //分割後的左側指標不變，但被上面的迴圈搬動了，所以搬回來

            child[i + 1] = split_node; //插入分割後的右側指標

            break; //找到位置後插入就不用再繼續找了

        }

    }

    if (i == data_num) { //如果新資料大於所有舊資料，不用做任何搬動，放到最後就好

        data[i] = new_data;

        child[i + 1] = split_node;

    }

    data_num++; //資料數+1

    if (data_num == order) { //資料數 == order 代表node滿了，node爆掉

        if (father == nullptr) { //沒有爸爸(root node)，記憶體配置出一個爸爸

            father = new B_tree_node<T, order>;

            father->child[0] = this; //爆掉後自己就變成分割後的左側，右側由爸爸產生並搬家

        }

        father->split_child(this); //爆掉後呼叫的function，產生分割後的右側並搬家

        data_num /= 2; //分割後的資料數是原資料數除以2

    }

}
}

```

以下是呼叫父節點分割爆掉子節點的 **function**，記憶體配置出一個新節點，然後把爆掉節點右半邊的資料與指標搬到新節點裡，被搬動的小孩指標也需要將此小孩的父節點指標設為新節點，因為指標比資料多一個，所以會額外處理最後一個指標的小孩，新節點的資料數就是剛剛搬的數量，最後將爆掉節點的中間值插入到自己裡面，遞迴下去。

```
void split_child(B_tree_node<T, order>* full) {  
    //把小孩分割，並搬家，然後把小孩中間值insert到自己內部  
  
    B_tree_node<T, order>* split = new B_tree_node<T, order>(this);  
    //配置新的node，放置分割後的右側  
  
    int index = 0; //用於搬家的變數，順便可得知新node資料數  
  
    for (int i = full->data_num / 2 + 1; i < order; i++, index++) { //把小孩右側搬到新node  
        split->data[index] = full->data[i]; //搬資料  
        split->child[index] = full->child[i]; //搬指標  
  
        if (split->child[index] != nullptr) { //如果小孩的小孩不為空，代表有孫子  
            split->child[index]->father = split;  
            //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸  
        }  
    }  
  
    split->child[index] = full->child[order]; //因為指標多一個，額外搬最後一個指標  
  
    if (split->child[index] != nullptr) {  
        split->child[index]->father = split;  
        //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸  
    }  
  
    split->data_num = index; //新node的資料數就是剛剛搬的數量  
  
    insert(full->middle(), split);  
  
    //把小孩的中間值塞到自己內部，如果自己也爆了，就再找爸爸求救，遞迴下去  
}
```

middle 回傳中間值，**output** 輸出這個節點的資料(最後不加空白)

```
T& middle() { //回傳中間值  
    return data[data_num / 2];  
}  
  
void output() { //輸出node資料，最後一筆不加空格  
    for (int i = 0; i < data_num; i++) {  
        cout << data[i] << ((i != data_num - 1) ? " " : "");  
    }  
}
```

以下是幾個內建在 node class 裡的 function，find_root 回傳 root node 指標，只要一直往上走就能找到，height 回傳節點的高度，要判斷是不是 leaf node 只需判斷第一個小孩存不存在就好，del 是釋放記憶體用的 function，會遞迴刪除所有小孩，最後再刪除自己。

```
B_tree_node<T, order>* find_root() {
    //因為root node爆掉後會換新的root node，所以用這個找到新的root node
    if (father == nullptr) return this;
    return father->find_root();
}

int height() {
    //該node的高度，leaf node高度為1，往上加1，因為B樹除了leaf node外都至少有一個小孩，且樹是平均高度的，所以直接用0號小孩遞迴即可
    if (child[0] == nullptr) return 1;
    return 1 + child[0]->height();
}

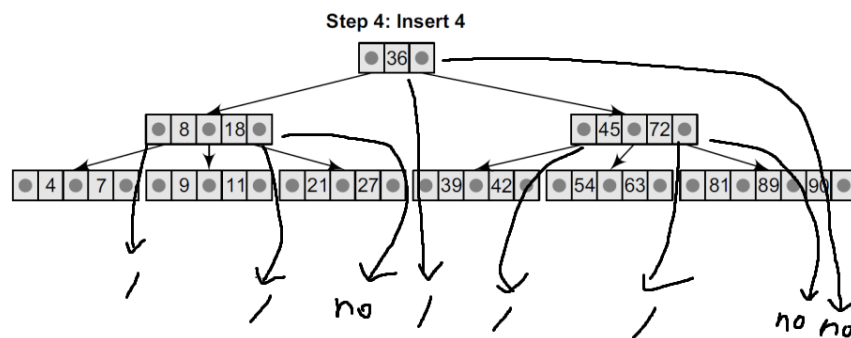
void del() { //遞迴刪除
    if (child[0] != nullptr) {
        for (int i = 0; i <= data_num; i++) {
            child[i]->del();
        }
    }
    delete this;
}
```

以下是 B 樹的 class，有 root node 指標，insert function 加入新資料會從這裡開始呼叫 node 內的 leaf_insert，加入新資料完後也要更新 root node。

```
template<typename T, unsigned int order>
class B_tree { //B樹
private:
    B_tree_node<T, order>* root = nullptr;
public:
    ~B_tree() { //釋放記憶體，遞迴刪除
        root->del();
        root = nullptr;
    }
    void insert(T new_data) {
        if (root == nullptr) root = new B_tree_node<T, order>(new_data);
        //如果甚麼都還沒有，就配置記憶體給root
        else {
            root->leaf_insert(new_data); //B樹是從leaf node插入的
            root = root->find_root(); //root node可能會變，所以要更新
        }
    }
}
```

處理輸出的函式，先算出樹高，然後用 for 迴圈跑過每層，每層用 output_layer 遞迴函式輸出該層所有 node，每個遞迴結束後都要輸出/區隔，但最後一個遞迴則不用。

每個箭頭都是遞迴結束後的輸出，例如 8、18 節點輸出 4、7 後結束遞迴，然後輸出斜線，但輸出完 21、27 後則不輸出，因為是最後一個遞迴，但這節點並不是最該層最後一節點(81、89、90 才是)，所以 21、27 節點與 39、42 節點之間必須有斜線輸出，這時候，上一層的遞迴就派上用場了，36 節點遞迴到 8、18 節點完後就會輸出一個斜線，這個斜線恰好解決了剛剛的疑問，所以不管要輸出哪層，只要最後一個遞迴出去後不輸出斜線就可以成功。



```
void output() { //一層一層的輸出整個樹
    int H = root->height(); //樹高
    for (int i = 0; i < H; i++) {
        output_layer(root, i, 0); //每一層的輸出，遞迴function
        cout << endl;
    }
}

private:
void output_layer(B_tree_node<T, order>* now, int target_layer, int now_layer) {
    //輸出target_layer層的所有資料，now_layer是now指標指向node的層數
    if (now == nullptr) return; //leaf node的小孩為nullptr，防呆用
    if (target_layer != now_layer) { //不是想要的layer，就遞迴往下層走
        for (int i = 0; i <= now->data_num; i++) { //每一個小孩都遞迴出去
            output_layer(now->child[i], target_layer, now_layer + 1);
            //下一層的now_layer就是這層+1
            cout << (i != now->data_num ? " / " : "");
            //node輸出後要用/區分，最後一個node輸出則不用
        }
    }
    else { //是想要輸出的層，輸出該node
        now->output(); //呼叫node內部的輸出function
    }
}
```

Part B: 插入 20, 45, 30, 50, 100, 70, 40, 10, 87 到 2-3 樹裡

(1) 插入 20

由於此時 root node 是 null，因此會執行這行:

```
if (root == nullptr) root = new B_tree_node<T, order>(new_data);
```

樹內的資料:  data_num == 1

(2) 插入 45

進入 leaf_insert root->leaf_insert(45);

節點 insert: insert(45, nullptr);

找到正確位置插入，45 大於節點內所有資料，所以放到最後，資料數+1

```
if (i == data_num) { //如果新資料大於所有舊資料，不用做任何搬動，放到最後就好
```


```
    data[i] = new_data;
```

```
    child[i + 1] = split_node;
```

```
}
```

```
data_num++; //資料數+1
```

檢查有無爆掉->沒有

樹內資料: 

(3) 插入 30

進入 leaf_insert root->leaf_insert(30);

節點 insert: insert(30, nullptr);

找到正確位置插入，用 for 迴圈跑，找到第一個大於 30 的資料=>data[1]

45，先把 45 以後(含 45)的資料與指標往後搬，然後把 30 放到 45 原本的位置

```
for (i = 0; i < data_num; i++) {
```

```
    if (new_data < data[i]) { //找到自己的位置
```

```
        child[order] = child[order - 1];
```

```
        //因為要插入新資料，把舊資料往後移，但指標多一個，所以額外處理最後一個指標
```

```
        for (int j = order - 1; j > i; j--) {
```

```
            data[j] = data[j - 1]; //把舊資料往後移
```

```
            child[j] = child[j - 1]; //把舊指標往後移
```

```
        }
```

```
        data[i] = new_data; //插入
```

```
        child[i] = child[i + 1];
```

```
        //分割後的左側指標不變，但被上面的迴圈搬動了，所以搬回來
```

```
        child[i + 1] = split_node; //插入分割後的右側指標
```

```
        break; //找到位置後插入就不用再繼續找了
```

```
    }
```

```
}
```

此時節點內的資料:20,30,45

判斷是否爆掉->是 `if (data_num == order)` `//資料數 == order` 代表 node 滿了，node 爆掉

父節點指標是否為空(是否為 root node)->是，new 一個新爸爸出來，並指派自己成為第一個小孩

```
if (father == nullptr) //沒有爸爸(root node)，記憶體配置出一個爸爸
    father = new B_tree_node<T, order>;
    father->child[0] = this;//爆掉後自己就變成分割後的左側，右側由爸爸產生並搬家
}
```

然後呼叫父節點分割爆掉的節點，分割後的資料數除以 2

`father->split_child(this);``//爆掉後呼叫的function`，產生分割後的右側並搬家

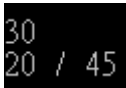
`data_num /= 2;``//分割後的資料數是原資料數除以 2`

父節點分割子節點函式，new 一個新節點，並將 45 與指標(現在是 null)搬到新節點，最後將中間值 30 insert 到自己內部

```
void split_child(B_tree_node<T, order>* full) {
    //把小孩分割，並搬家，然後把小孩中間值insert到自己內部
    B_tree_node<T, order>* split = new B_tree_node<T, order>(this);
    //配置新的node，放置分割後的右側
    int index = 0;//用於搬家的變數，順便可得知新node資料數
    for (int i = full->data_num / 2 + 1; i < order; i++, index++) //把小孩右側搬到新node
        split->data[index] = full->data[i];//搬資料
        split->child[index] = full->child[i];//搬指標
        if (split->child[index] != nullptr) //如果小孩的小孩不為空，代表有孫子
            split->child[index]->father = split;
            //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸
        }
    }
    split->child[index] = full->child[order];//因為指標多一個，額外搬最後一個指標
    if (split->child[index] != nullptr) {
        split->child[index]->father = split;
        //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸
    }
    split->data_num = index;//新node的資料數就是剛剛搬的數量
    insert(full->middle(), split);
    //把小孩的中間值塞到自己內部，如果自己也爆了，就再找爸爸求救，遞迴下去
}
```


在 insert function 內，新的 root node 屬於特例，特別處理它

```
if (data_num == 0) {  
    //如果資料數為0，代表這是原root node爆掉後產生的新root node  
  
    data[0] = new_data;  
    child[1] = split_node;  
  
    //child[0]是分割後左側(在下面的程式碼賦值)，child[1]是分割後右側  
  
    data_num = 1;  
    return;  
}
```

樹內資料: 

(4) 插入 50

進入 leaf_insert root->leaf_insert(50);


遞迴一次後到達 leaf ndoe: 

leaf ndoe 節點 insert: insert(50,nullptr);


因為 50 大於 45，所以放到最後面就好，該節點資料數+1

```
if (i == data_num) { //如果新資料大於所有舊資料，不用做任何搬動，放到最後就好  
    data[i] = new_data;  
    child[i + 1] = split_node;  
}  
  
data_num++; //資料數+1
```

判斷是否爆掉->否

樹內資料: 

(5) 插入 100

跟前面的步驟一樣，遞迴到  節點後，呼叫插入

插入完後節點內資料:45,50,100

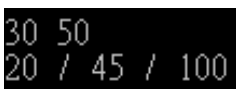
判斷是否爆掉->是

是否為 root node->否

呼叫父節點  分割爆掉的節點，分割後的資料數除以 2

父節點: new 出節點，把 100 搬過去，50 插入到  節點。

判斷是否爆掉->否


樹內資料: 

(6) 插入 70

跟前面的步驟一樣，遞迴到 100 節點後，呼叫插入

插入完後節點內資料:70,100

判斷是否爆掉->否


樹內資料: 

(7) 插入 40

跟前面的步驟一樣，遞迴到 45 節點後，呼叫插入

插入完後節點內資料:40,45

判斷是否爆掉->否


樹內資料: 

(8) 插入 10

跟前面的步驟一樣，遞迴到 20 節點後，呼叫插入

插入完後節點內資料:10,20

判斷是否爆掉->否

樹內資料: 

(9) 插入 87

跟前面的步驟一樣，遞迴到 70 100 節點後，呼叫插入

插入完後節點內資料:70,87,100

判斷是否爆掉->是

是否為 root node->否

呼叫父節點 30 50 分割爆掉的節點，分割後的資料數除以 2

父節點: new 出節點，把 100 搬過去，87 插入到 30 50 節點。

插入完後節點內資料:30,50,87

判斷是否爆掉->是

是否為 root node->是

new 出新 root node，並指派自己成為第一個小孩，呼叫新爸爸分割節點

新 root node: new 出新 node，把 87 後(含)的指標與資料，搬到新 node，

50 插到新 root node，插入完後的新 root node: 50

判斷是否爆掉->否

樹內資料: 

(10) 插入 1

跟前面的步驟一樣，遞迴到 10 20 節點後，呼叫插入

插入完後節點內資料:1,10,20

判斷是否爆掉->是

是否為 root node->否

呼叫父節點 30 分割爆掉的節點，分割後的資料數除以 2

父節點: new 出節點，把 20 搬過去，10 插入到 30 節點。

插入完後節點內資料:10,30

判斷是否爆掉->否

樹內資料:

```
50
10 30 / 87
1 / 20 / 40 45 / 70 / 100
```

3. 程式碼

[2-1 Thanos Finger Snap gist](#) [imgur 圖片](#)

[2-2 2-3 Tree gist](#) [imgur 圖片](#)

2-1:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

template<typename T>
class BST_node { //BST節點的class
public:
    BST_node() {}
    BST_node(T _data):data(_data) {}
    bool operator>(const BST_node& i) { return data > i.data; }
    bool operator>(const T& i) { return data > i; }
    bool operator>=(const BST_node& i) { return data >= i.data; }
    bool operator>=(const T& i) { return data >= i; }
    bool operator<(const BST_node& i) { return data < i.data; }
    bool operator<(const T& i) { return data < i; }
    bool operator<=(const BST_node& i) { return data <= i.data; }
    bool operator<=(const T& i) { return data <= i; }
    void del() { //遞迴刪除所有子孫節點，最後刪除自己
        if (left != nullptr) {
            left->del();
        }
        if (right != nullptr) {
            right->del();
        }
        delete this;
    }
private:
    T data;
    BST_node<T>* father = nullptr; //爸爸
    BST_node<T>* left = nullptr; //左邊的小孩
    BST_node<T>* right = nullptr; //右邊的小孩
    template<typename> friend class BST;
};
```

```
template<typename T> <T> 提供適用於 IntelliSense 的樣本範本引數
class BST { //BST的class
private:
    BST_node<T>* root = nullptr; //根節點的指標
public:
    BST() {}
    ~BST() { //釋放記憶體
        root->del();
        root = nullptr;
    }
};
```

```
void insert(T new_data) { //這個是從根結點出發的insert，另有重載一個可從任意指標insert的遞迴function
    if (root == nullptr) { //如果甚麼都沒有
        root = new BST_node<T>(new_data); //配置記憶體並賦值
    }
    else {
        if (*root > new_data) { //新資料較小，往左走
            insert(&root->left, new_data, root); //呼叫真正的遞迴insert
        }
        else { //新資料較大，往右走
            insert(&root->right, new_data, root); //呼叫真正的遞迴insert
        }
    }
}

void erase(T data) { //呼叫search，找到指標後，呼叫erase的重載function
    erase(search(data));
}

//以下函式是為了簡化介面，真正實作的function重載在private裡
BST_node<T>* search(T data) { return search(root, data); } //搜尋節點，回傳目標節點指標
BST_node<T>* largest() { return largest(root); } //回傳樹最大節點的指標
BST_node<T>* smallest() { return smallest(root); } //回傳樹最小節點的指標
void pre_order_output() { pre_order_output(root); } //輸出前序
void in_order_output() { in_order_output(root); } //輸出中序
void post_order_output() { post_order_output(root); } //輸出後序
int height() { return height(root); } //回傳樹高
```

```

void cut_half() { //Thanos Finger Snap 找到最近切半的水平線後，將水平線以下node全部刪除
    int H = height(); //樹高(有幾層)
    int* layer_count = new int[H] {}; //動態配置陣列，儲存每層的node數
    get_layer_number_of_node(root, layer_count, 0); //遞迴計算每層的node數，結果存於layer_count
    int min_diff = 2147483647; //水平線上數量與線下數量差，一開始先設很大，避免被超過
    int cut_layer = 0; //儲存最佳的水平線(層數)
    for (int i = 0; i < H; i++) { //跑過每條水平線，找最好的
        int up = 0, down = 0; //分別存水平線上數量與線下數量
        for (int j = 0; j < H; j++) { //掃過每層，把數量加到up或down
            (j <= i ? up : down) += layer_count[j]; //數量加總 <= i 屬於上半層，反之下半層
        }
        if (abs(up - down) < min_diff) { //若數量差小於之前所偵測到的最小差
            min_diff = abs(up - down); //更新最小數量差
            cut_layer = i; //更新最佳水平線
        }
    }

    BST_node<T>* forest = new BST_node<T>* [layer_count[cut_layer + 1]] {}; //配置森林記憶體，共有(水平線+1)層之節點數
    int forest_index = 0; //用一個變數才能call by reference，存現在種了幾棵樹
    cut_layer_and_create_forest(root, forest, forest_index, cut_layer + 1, 0); //切除水平線下節點並存到森林中，遞迴函式
    in_order_output(); //切完後，輸出中序
    cout << endl;
    for (int i = 0; i < layer_count[cut_layer + 1]; i++) { //跑過森林裡的每棵樹
        pre_order_output(forest[i]); //輸出前序
        forest[i] -> del(); //刪除此樹
    }
    delete[] layer_count; //釋放每層的node數陣列
    delete[] forest; //釋放森林
}

//now存現在遞迴到哪個節點，forest存切除後的樹，forest_index存現在是第幾棵樹，target_layer為(最佳水平線+1)層，now_layer為now所在層數(root為0層)
void cut_layer_and_create_forest(BST_node<T>* now, BST_node<T>* forest, int& forest_index, int target_layer, int now_layer) {
    if (now == nullptr) return; //防呆
    if (now_layer == target_layer) { //遞迴到正確層數
        forest[forest_index] = now; //切除後的樹存入森林
        forest_index++; //下一棵樹，因為是call by reference，所以同層級的遞迴也能被更改到
        if (now -> father -> left == now) { //如果自己是爸爸的左小孩
            now -> father -> left = nullptr; //爸爸的左指標設為空，切除
        }
        else { //如果自己是爸爸的右小孩
            now -> father -> right = nullptr; //爸爸的右指標設為空，切除
        }
    }
    else {
        cut_layer_and_create_forest(now -> left, forest, forest_index, target_layer, now_layer + 1); //從左邊開始掃，所以先遞迴左邊
        cut_layer_and_create_forest(now -> right, forest, forest_index, target_layer, now_layer + 1); //遞迴右邊
    }
}

```

```

private:
    //使用指標的指標才能在下次遞迴中配置記憶體並連接上一個node
    void insert(BST_node<T>* now, T new_data, BST_node<T>* father) {
        if (*now == nullptr) { //如果甚麼都沒有
            *now = new BST_node<T>(new_data); //配置記憶體並賦值
            (*now) -> father = father;
        }
        else {
            if (*now > new_data) { //新資料較小，往左走
                insert(&(*now) -> left, new_data, *now); //遞迴
            }
            else { //新資料較大，往右走
                insert(&(*now) -> right, new_data, *now); //遞迴
            }
        }
    }

    void erase(BST_node<T>* target) { //傳入節點指標，刪除該節點
        if (target == nullptr) return; //防呆
        int child_num = (target -> left != nullptr) + (target -> right != nullptr); //計算小孩數量，nullptr代表沒小孩，回傳0，反之回傳1
        switch (child_num)
        {
            case 0: case 1: { //0個或1個小孩
                BST_node<T>* replacement = target -> left != nullptr ? target -> left : target -> right; //替代節點指標，若2邊為空，則為空，若
                if (target -> father == nullptr) { //如果是根結點，刪除後設root為替代節點指標
                    delete target;
                    root = replacement;
                    return;
                }
                if (target -> father -> left == target) { //如果自己是爸爸的左小孩
                    target -> father -> left = replacement; //爸爸左指標設為替代節點指標
                }
                else { //如果自己是爸爸的右小孩
                    target -> father -> right = replacement; //爸爸右指標設為替代節點指標
                }
                if (replacement != nullptr) replacement -> father = target -> father; //如果替代節點指標不為空，將小孩的爸爸設為自己的爸爸
                delete target; //刪除目標
            } break;
            case 2: { //左右都有小孩
                BST_node<T>* replacement = largest(target -> left); //找左子樹中最大
                target -> data = replacement -> data; //用左子樹中最大值替代原本的值
                erase(replacement); //刪除左子樹中最大值的node
            } break;
            default: break;
        }
    }
}

```

```

BST_node<T>* search(BST_node<T>* now, T data) { //搜尋節點
    if (now == nullptr) return nullptr;
    if (*now > data) return search(now->left, data); //要找的資料小於目前節點的資料，往左走
    else if (*now < data) return search(now->right, data); //要找的資料大於目前節點的資料，往右走
    else return now; //既非大於也非小於，那就是等於
}

BST_node<T>* largest(BST_node<T>* now) { //回傳now以下最大值的節點，一直往右走即可
    if (now == nullptr) return nullptr;
    else if (now->right == nullptr) return now;
    return largest(now->right);
}

BST_node<T>* smallest(BST_node<T>* now) { //回傳now以下最小值的節點，一直往左走即可
    if (now == nullptr) return nullptr;
    else if (now->left == nullptr) return now;
    return smallest(now->left);
}

int height(BST_node<T>* now) { //回傳now的高度，leaf node為高度為1
    if (now == nullptr) return 0;
    int LH = height(now->left);
    int RH = height(now->right);
    return (LH > RH ? LH : RH) + 1; //左右子樹高度選大的並加上自己(1)
}

void get_layer_number_of_node(BST_node<T>* now, int* layer_count, int layer) { //計算每層node數
    if (now == nullptr) return; //防呆
    layer_count[layer]++; //該層node數加1
    get_layer_number_of_node(now->left, layer_count, layer + 1); //往左遞迴下去，下一層為layer+1
    get_layer_number_of_node(now->right, layer_count, layer + 1); //往右遞迴下去，下一層為layer+1
}

void pre_order_output(BST_node<T>* now) { //前序輸出，輸出該節點，然後繼續遞迴
    if (now == nullptr) return;
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
    pre_order_output(now->left); //遞迴左子樹
    pre_order_output(now->right); //遞迴右子樹
}

void in_order_output(BST_node<T>* now) { //中序輸出，先遞迴左子樹，然後輸出該節點，遞迴右子樹
    if (now == nullptr) return;
    in_order_output(now->left); //遞迴左子樹
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
    in_order_output(now->right); //遞迴右子樹
}

```

```

void post_order_output(BST_node<T>* now) { //後序輸出，遞迴左子樹與右子樹，然後輸出該節點
    if (now == nullptr) return;
    post_order_output(now->left); //遞迴左子樹
    post_order_output(now->right); //遞迴右子樹
    static bool first_out = true; //static讓每次遞迴都可以用相同的變數
    cout << (first_out ? "" : " ") << now->data; //第一次輸出不用加空白，後面都要
    first_out = false; //輸出完就不是第一次了
}

};

int main() {
    BST<int> tree;
    string line; //存一行
    int data; //暫存數字
    getline(cin, line);
    stringstream s(line);
    while (s >> data) { //用stringstream讀數字
        tree.insert(data); //插入所有數字
    }
    tree.post_order_output(); //輸出後序
    cout << endl;
    getline(cin, line);
    s.clear();
    s.str(line);
    while (s >> data) { //用stringstream讀數字
        tree.erase(data); //刪除Avengers
    }
    tree.cut_half(); //刪除一半node後輸出上半部(中序)、下半部(前序)
    return 0;
}

```

2-2:

```
#include <iostream>
using namespace std;
template<typename T, unsigned int order> <T> 提供適用於 IntelliSense 的樣本範本引數 -
class B_tree_node { //B樹節點
public:
    //constructor
    B_tree_node() {set_null();} //啥都沒
    B_tree_node(B_tree_node<T, order>* f) : father(f) {set_null();} //有給爸爸
    B_tree_node(T new_data) { //有給資料
        set_null();
        data[0] = new_data;
        data_rnum = 1;
    }
    B_tree_node(T new_data, B_tree_node<T, order>* f) : father(f) { //有給爸爸跟資料
        set_null();
        data[0] = new_data;
        data_rnum = 1;
    }

    B_tree_node<T, order>* father = nullptr; //爸爸指標，爆掉時找爸爸求救用
    B_tree_node<T, order>* child[order + 1]; //放小孩指標，多一個位置用於放置爆掉時多出來的東西
    T data[order]; //放資料，多一個位置用於放置爆掉時多出來的東西
    int data_rnum = 0; //資料數量 指標數 = 資料數量 + 1

    void set_null() { for (int i = 0; i <= order; i++) child[i] = nullptr; } //所有小孩設成空
    void leaf_insert(T new_data) { //B樹的insert都是從leaf node開始，爆掉了再往上長，所以這個function用於新資料的insert，是專屬於整棵樹的insert function
        //B樹leaf node以外的node都至少有一個小孩，所以拿0號小孩指標確認有無小孩
        if (child[0] != nullptr) { //不是leaf node，遞迴找到leaf node再用普通的insert
            for (int i = 0; i < data_rnum; i++) {
                if (new_data < data[i]) { //找到第一個大於的資料，就是插入後要放的地方
                    child[i] -> leaf_insert(new_data); //遞迴下一個節點
                    return; //結束掉，不然會跑到下面那行
                }
            }
            child[data_rnum] -> leaf_insert(new_data); //上面都沒進入遞迴就會跑到這行，代表新資料大於node裡的所有資料，找最右邊的指標遞迴
        }
        else { //是leaf node
            insert(new_data, nullptr); //經過許多遞迴後成功找到leaf node，這是專屬於node的insert function，由於是新資料所以沒有分割後的指標，填null就好
        }
    }

    void insert(T new_data, B_tree_node<T, order>* split_node) { //第一個參數是新資料，第二個參數是當爆掉時node分割後的指標(右邊那側的分割)
        //特例：新root node
        if (data_rnum == 0) { //如果資料數為0，代表這是原root node爆掉後產生的新root node
            data[0] = new_data;
            child[1] = split_node; //child[0]是分割後左側(在下面的程式調取值)，child[1]是分割後右側
            data_rnum = 1;
            return;
        }
        int i;
        for (i = 0; i < data_rnum; i++) {
            if (new_data < data[i]) { //找到自己的位置

```

```
                child[order] = child[order - 1]; //因為要插入新資料，把舊資料往後移，但指標多一個，所以額外處理最後一個指標
                for (int j = order - 1; j > i; j--) {
                    data[j] = data[j - 1]; //把舊資料往後移
                    child[j] = child[j - 1]; //把舊指標往後移
                }
                data[i] = new_data; //插入
                child[i] = child[i + 1]; //分割後的左側指標不變，但被上面的迴圈搬動了，所以搬回來
                child[i + 1] = split_node; //插入分割後的右側指標
                break; //找到位置後插入就不用再繼續找了
            }
        }
        if (i == data_rnum) { //如果新資料大於所有舊資料，不用做任何搬動，放到最後就好
            data[i] = new_data;
            child[i + 1] = split_node;
        }
        data_rnum++; //資料數+1
        if (data_rnum == order) { //資料數 = order 代表node滿了，node爆掉
            if (father == nullptr) { //沒有爸爸(root node)，記憶體配置出一個爸爸
                father = new B_tree_node<T, order>;
                father -> child[0] = this; //爆掉後自己就變成分割後的左側，右側由爸爸產生並搬家
            }
            father -> split_child(this); //爆掉後呼叫的function，產生分割後的右側並搬家
            data_rnum /= 2; //分割後的資料數是原資料數除以2
        }
    }
}
```

```

void split_child(B_tree_node<T, order>* full) { //把小孩分割，並搬家，然後把小孩中間值insert到自己內部
    B_tree_node<T, order>* split = new B_tree_node<T, order>(this); //配置新的node，放置分割後的右側
    int index = 0; //用於搬家的變數，順便可得知新node資料數
    for (int i = full->data_num / 2 + 1; i < order; i++, index++) { //把小孩右側搬到新node
        split->data[index] = full->data[i]; //搬資料
        split->child[index] = full->child[i]; //搬指標
        if (split->child[index] != nullptr) { //如果小孩的小孩不為空，代表有孫子
            split->child[index]->father = split; //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸
        }
    }

    split->child[index] = full->child[order]; //因為指標多一個，額外搬最後一個指標
    if (split->child[index] != nullptr) {
        split->child[index]->father = split; //小孩的小孩(孫子)因為原爸爸被分割，所以要換新爸爸
    }

    split->data_num = index; //新node的資料數就是剛剛搬的數量
    insert(full->middle(), split); //把小孩的中間值塞到自己內部，如果自己也爆了，就再找爸爸求救，遞迴下去
}

T& middle() { //回傳中間值
    return data[data_num / 2];
}

void output() { //輸出node資料，最後一筆不加空格
    for (int i = 0; i < data_num; i++) {
        cout << data[i] << ((i != data_num - 1) ? " " : "");
    }
}

B_tree_node<T, order>* find_root() { //因為root node爆掉後會換新的root node，所以用這個找到新的root node
    if (father == nullptr) return this;
    return father->find_root();
}

int height() { //該node的高度，leaf node高度為1，往上加1，因為B樹除了leaf node外都至少有一個小孩，且樹是平均高度的，所以直接用0號小孩遞迴即可
    if (child[0] == nullptr) return 1;
    return 1 + child[0]->height();
}

void del() { //遞迴刪除
    if (child[0] != nullptr) {
        for (int i = 0; i <= data_num; i++) {
            child[i]->del();
        }
    }

    delete this;
}

};

```

```

template<typename T, unsigned int order>
class B_tree { //B樹
private:
    B_tree_node<T, order>* root = nullptr;
public:
    ~B_tree() { //釋放記憶體，遞迴刪除
        root->del();
        root = nullptr;
    }

    void insert(T new_data) {
        if (root == nullptr) root = new B_tree_node<T, order>(new_data); //如果甚麼都沒有，就配置記憶體給root
        else {
            root->leaf_insert(new_data); //B樹是從leaf node插入的
            root = root->find_root(); //root node可能會變，所以要更新
        }
    }

    void output() { //一層一層的輸出整個樹
        int H = root->height(); //樹高
        for (int i = 0; i < H; i++) {
            output_layer(root, i, 0); //每一層的輸出，遞迴function
            cout << endl;
        }
    }

private:
    void output_layer(B_tree_node<T, order>* now, int target_layer, int now_layer) { //輸出target_layer層的所有資料，now_layer是now指標指向node的層數
        if (now == nullptr) return; //leaf node的小孩為nullptr，防呆用
        if (target_layer != now_layer) { //不是想要的layer，就遞迴往下層走
            for (int i = 0; i <= now->data_num; i++) { //每一個小孩都遞迴出去
                output_layer(now->child[i], target_layer, now_layer + 1); //下一層的now_layer就是這層+1
                cout << ((i != now->data_num) ? " / " : ""); //node輸出後要用/區分，最後一個node輸出則不用
            }
        }
        else { //是想要輸出的層，輸出該node
            now->output(); //呼叫node內部的輸出function
        }
    }
};

//20 45 30 50 100 70 40 10 87 1
int main() {
    B_tree<int, 3> tree; //2 3樹就是order為3的B樹
    int new_data; //暫存每個數字變數
    while (cin >> new_data) {
        tree.insert(new_data); //插入
    }

    tree.output(); //輸出整棵樹
    return 0;
}

```