B05611047 化學三 鍾宜樺

Problem 1 k-th Largest Element

1-1 走一 postorder traversal(LRN)，記下對於每個 node 的左子樹加上右子樹的加上本身的 node 數有幾個。因為一開始先走 leaf，然而 leaf 沒有左子樹和右子樹，因此 leaf 的的 sz[leaf-node] = 1，然後一路走 postorder traversal 走上去。並且在這個計算 sz[x]的 function 裡面，要多令兩個變數 int l_num, r_num，分別左子樹和右子樹的 sz number，以便求得在這個子樹中的 root 的 sz[sun_tree_root] = l_num + r_num + 1(本身須包含)。一路走上去就會把全部的 sz[x]都求得了。

時間複雜度：因為是走 postorder traversal 將所有 n 個 node 走完，因此時間複雜度為 O(N)。

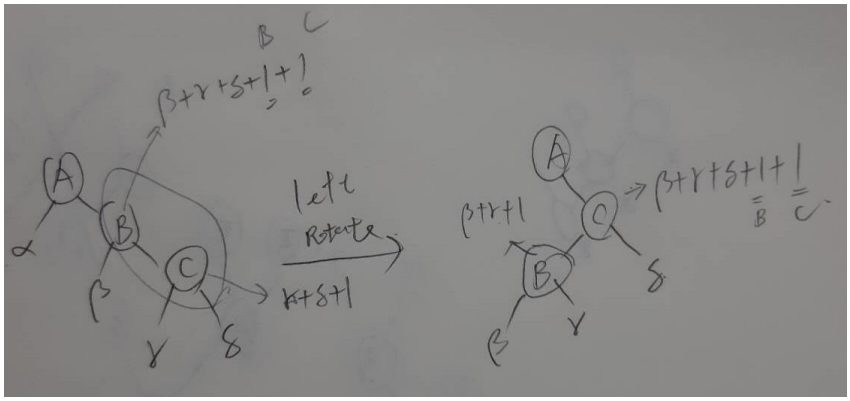
1-2 因為沒有 rotate，因此相對位置不動，故在 insert 之後，會被改變的 sz[x]值只有背 insert 的那個 node 的祖宗，因此需將所有祖先的 sz[x]都+1 即可，其他部分不用改。作法就是在一開始 insert 的時候，每比對一次，被 insert 往下移動經過一個 node 的時候，那個 node 的 sz 值就須+1;。好比假設被 insert 的 Node 的祖先有 5 個值，則從一開始 root 下來的時候這五個 node 的 sz[x]都會被+1，因為他們下面多被安插了一個 node。因此 descendants 會多 1，故加一。

時間複雜度：會改變的只有祖先，祖先數量和樹高成正比。因此時間複雜度 O(logN)。

1-3 同理 1-2，因為題目說沒有 rotation，因此即便刪掉一個 node，整棵樹的 node 與 node 的相對位置並沒有改變，因此會被改變的 sz 值只有祖先的。反之 1-2，即被刪一的 node 的祖先的 sz 值都需要減一，因為他們下面少了一個 node。

時間複雜度：會改變的只有祖先，祖先數量和樹高成正比。因此時間複雜度 O(logN)。

1-4 發生左旋轉的狀況：



從圖中可以看到，B、C 發生左旋轉的時候 C 的左子樹會被轉成 B 的右子樹，且 B 會變成 C 的左子樹。然而在 BC 下方的子樹內容和裡面的 node 的相對位置不改變，因此他

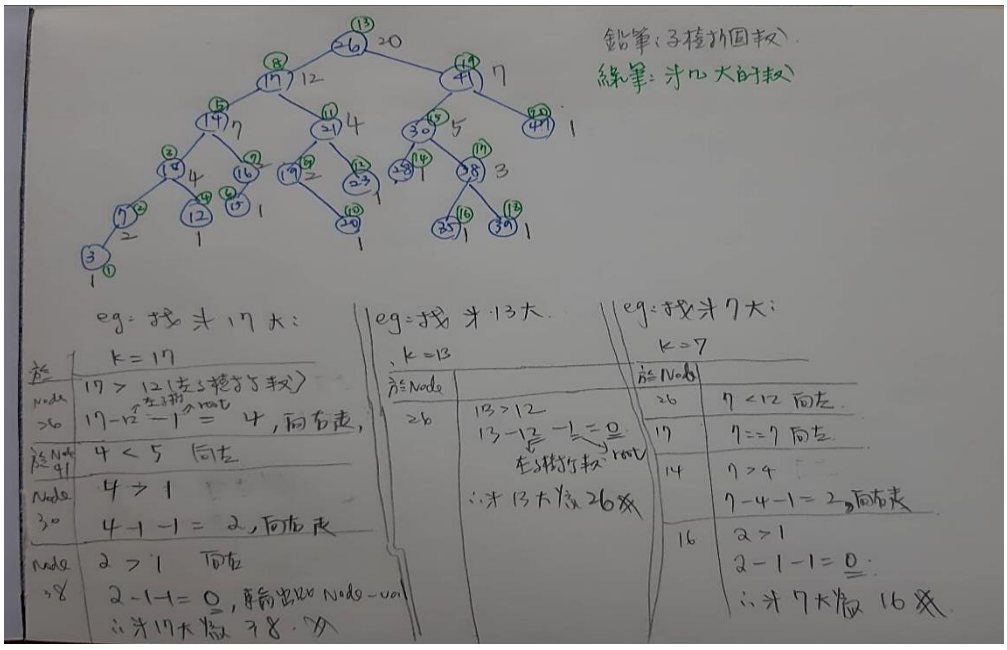
們的 sz 值不需要被更動，又對於 BC 上面的 node 而言，他們下面子樹的總 node 數也沒有改變，因此他們的 sz 值亦不需更動。因此可知，sz 值會改變的只有被旋轉的兩個點，又因為被往上轉的點對於他下面新的子代們的相對位置，就同於在還沒轉動時的 B(如圖所示)，即 C 取代 B 變成原本 B 下面所有點的子樹的 root，因此 sz[C] = sz[B]，新的 C 的 sz 值會是 B 原本的 sz 值。而對於 B 而言，他被往下轉之後，所失去的子代便是未被旋轉的 C 的右子樹以及 C 本身這個 node。

因此新的 sz[B] = [(原本的 sz[B] - 1(Node C) – 原本 C 的右子樹的個數值(圖中之δ))]。

時間複雜度：因為只需改變兩個 sz 值，且都是做一條加減法的算式，因此時間複雜度為 O(1)。

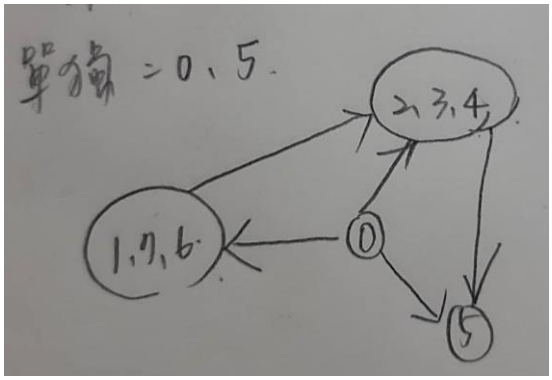
1-5 方法：因為已經儲存每個子樹中的結點個數。就可以知道當想要找地 k 大的值時，若左子樹的節點個數小於等於 k 時，即可知道，所要找的 k 在左子樹中，因此向左走。反之，若左子樹節點數 < k 時，則代表要向右走。並且，在向右走去右子樹的時候便將 k 更新，k’ = k- (左子樹個數 + 1)，其中 1 代表 root。要扣掉的原因是因為，在目前的子樹中，要向右走，走到一個新的子樹前，已找到(左子樹個數 + 1)個比所求還要小的數值，因此我們需要在新的子樹中，找 k’th 大的值。一路這樣找下去，最後當 k 變成 0 的時候，此點的 value 即為所求。怕前面講的不清楚@@因此下面附上例子，此紅黑數取自 PPT(沒有標上顏色之類的細節。詳情請洽 PPTXD)。

時間複雜度：因為一開始是從整棵樹的 root 一路向下走，然後隨著每次判別選擇往左或者往右走，因此花費的總時間和樹高成正比，固時間複雜度為 O(logN)。



Problem 2 Strongly Connected Component

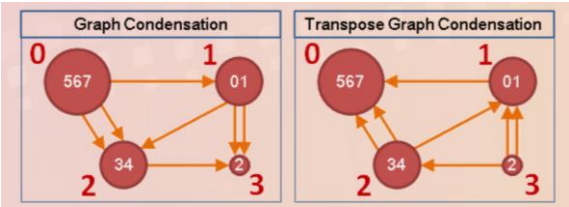
2-1



2-2

by the definition of the SCC, every vertex is reachable from every other vertex. 可知，如果在一開始的 G 上 v 點可以抵達 u 點，即 v, u 中間有條 path1，因此在 reversing the directions of all edges in graph 後，也可以知道 u 一定也可以抵達到 v (為原本的 path1 的完全反向)。又因此可知，若在 G 上為 SCC (即所有點可以相互抵達彼此) ，則在 Gr 之後同理也可相互抵達 (且路途為完全相反)，再次形成 SCC。然而若在一開始的 G 時 x 就已經不能抵達 y，則即使將所有的 edges 反向後，因為兩個點中間也不會生出一條路，所以反向後 y 仍不能抵達 x。也就是不能相互抵達的點與點之間，即使在反向後仍然不會相連。因此可知兩張 graph 不同的地方在於除了 SCC 的連線均為反向。且 G 和 Gr 所擁有的 SCC 都相同。因此得 $\overline{G^R} = \bar{G}^R$。

Eg: (http://www.csie.ntnu.edu.tw/~u91029/Component.html)



2-3

```
/*
Ref :
1. http://www.csie.ntnu.edu.tw/~u91029/Component.html
2. https://www.geeksforgeeks.org/graph-and-its-representations/
3. stack 的 structure 用這邊的 : https://www.geeksforgeeks.org/stack-data-structure/ (怕 code 太冗長，因此直接叫)

時間複雜度：時間複雜度為兩次 DFS 的時間，以及顛倒所有邊的時間。
```

```
資料結構是 adjacency lists ，需要顛倒所有邊，總時間複雜度 O(V+E) 。
下面利用 array 代表 stack
*/

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
//for DFS


struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};

struct AdjList
{
    struct AdjListNode *head;
};

struct Graph
{
    int V;
    struct AdjList* array;
};

struct Graph* graph ;
struct Graph* rev_gph;

struct AdjListNode* newAdjListNode(int dest)
{
    struct AdjListNode* newNode = (struct AdjListNode*)
malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
```

```
}

struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*) malloc( V * sizeof(struct
AdjList));
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

//src : source, dest :destination
void addEdge(struct Graph* graph, int src, int dest)
{
    //無向圖，加上兩邊
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;}

void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
```

```
        }
        printf("\n");
    }
}

//////////////////

struct Stack{
    int top;
    int Max_sz;
    int* data;
};
struct Stack *create_stack( int size ){
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->Max_sz = size -1;
    stack->top = -1;
    stack->data = (int*) malloc(stack->Max_sz * sizeof(int));
    return stack;
};
bool isFull( struct Stack *stack ){
    return stack->top == stack->Max_sz-1;
}


// Stack is empty when top is equal to -1
bool isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int num){
    if( isFull(stack) == true){
        printf("%s is bomb\n", stack);
        return;
    }
    else
        stack->data[++stack->top] = num;
}
```

```
// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack){
    if ( isEmpty(stack) == true){
        printf("QQ");
        return -1;
        //since there can't be no  return
    }
    else
        return (stack->data[stack->top--]);
}

int peek(struct Stack *stack) {
    return stack->data[stack->top];
}

struct stack* stk;
void push_in_stk ( int i) {
    push(stk, i);
}

void DFS (int i, int V, bool visited[]){
    printf("inside, visiting : %d\n", i);
    struct AdjListNode* tmp = graph->array[i].head; //point at the first
number
    while (tmp != NULL && visited[i] == false){
        visited[i] = true;
        for (int j = 0; j < V; j++){
            printf("visited[%d] = %d\n", j , visited[j]);
        }
        printf("\n");
        DFS( tmp->dest, V, visited);
        tmp = tmp->next;
        //加入 stack 中
        ///
        if (isFull(stk) == false)
            push_in_stk(i);
        printf("stack now is : %d\n", peek(stk));
        ///
```

```
    }
    if (tmp != NULL && visited[i] == true){
        printf("already visited %d .\n", i);
    }
    if (tmp == NULL && visited[i] == false){
        printf("case3\n");
        visited[i] = true;
        ///
        //加入 stack 中
        if (isFull(stk) == false)
            push_in_stk(i);
        printf("stack now is : %d\n", peek(stk));
        ///
        for (int j = 0; j < V; j++){
            printf("visited[%d] = %d\n", j , visited[j]);
        }
        printf("\n");
    }

//    for (int k = 0 ; k< V;k++){
//        printf("stk[%d] = %d", k, stk[k]);
//    }
}

void traversal (int V, bool visited[]){
    int c = V;
    while (c--){
        visited[c] = false;
    }
//    printf("initial\n");
//    for (int j = 0; j < V; j++){
//        printf("visited[%d] = %d\n", j , visited[j]);
//    }
    for (int i = 0; i < V; i++){
        if ( visited[i] == false){
            visited[i] == true;
            printf("\ninto DFS\n");
```

```
            DFS(i,V, visited);
        }
    }
}
////////////////////第一次DFS/////////////////

///////////第二次DFS/////////////////


void DFS_2 (int i, int V, int SCC_group, bool visited_2[] , int SCC[]){
    struct AdjListNode* tmp = rev_gph->array[i].head; //開始走反過來的
graph
    while (tmp != NULL && visited_2[i] == false){
        visited_2[i] = true;
        DFS_2( tmp->dest, V, SCC_group, visited_2, SCC);
        SCC[i] = SCC_group;
        tmp = tmp->next;
    }
    if (tmp != NULL && visited_2[i] == true){
        printf("already visited_2 %d .\n", i);
    }
    if (tmp == NULL && visited_2[i] == false){
        visited_2[i] = true;
        SCC[i] = SCC_group;
    }
}

void traversal_2 (int V, bool visited_2[], int SCC[] ){
    int c = V;
    while (c--){
        visited_2[c] = false;
    }
    while(isEmpty(stk) == false){
        int pp = pop(stk);
        if (visited_2[pp] == false){
            visited_2[pp] = true;
            SCC[pp] = pp;
```

```
            DFS_2( pp, V, pp, visited_2 , SCC);// 利用第一個跑進去作 DFS 的數字做為 SCC group 的代稱，
            //例如 node 1 2 3 是相同的 SCC，且第一個進去 DFS 的是 node 2 則 SCC[1] == SCC[2] == SCC[3] == 2
        }
    }
}

///////////第二次 DFS//////////////////


int main()
{
//
//    int V, E;
//    scanf("%d %d", &V, &E);
//    //for DFS
//    bool visited[V];
//    bool visited_2[V];
//    stk = create_stack(V);
//    graph = createGraph(V);
//    rev_gph = createGraph(V);
//
//    int arc[E][2];
//
//    int SCC[V];

//    int src, dest;
//    int tmp_E = E;
//    int i = 0;
//    while (E--){
//        scanf("%d %d", &src, &dest);
//        addEdge(graph, src, dest);
//        arc[i][1] = src;
//        arc[i][0] = dest;
//        i++;
//    }
//    //printGraph(graph);
```

```
//    ///
//    //將邊全部反過來
//    i = 0;
//    while (tmp_E --){
//        addEdge(rev_gph,arc[i][0], arc[i][1]);
//        i++;
//    }
//    //printGraph(rev_gph);
//    ///
//
//
//
//    traversal(V, visited);
//
////    printf("print out the stack\n");
////    for (int i = 0; i < V; i++){
////        printf("%d " ,pop(stk));
////    }
////    printf("\n");

    return 0;
}
```