MiohitoKiri5474

Data Structure

Prefix
Segment Tree
Binary Indexed Tree
Segment Tree with Lazy Tag
Disjoint Set and Union
Segment Tree Extra
High Dimension Data Structure
Treap and Persistent Treap

先看個題目：題目 #1

- 給定一初始序列，並有 Q 筆查詢區間 $[l, r]$ 內的和
- $|$序列長度$| = N, Q \leq 10^5$
- $1 \leq l, r, \leq N$

然後手順寫了這東西

```cpp
#include<bits/stdc++.h>

using namespace std;

int lib[100005];

int main() {
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, ans, l, r;
    cin >> n >> m;
    for ( int i = 0 ; i < n ; i++ )
        cin >> lib[i];
    while ( m-- ) {
        cin >> l >> r;
        ans = 0;
        for ( int i = l ; i <= r ; i++ )
            ans += lib[i];
        cout << ans << '\n';
    }
}
```

Prefix

前綴和

- 每一格儲存的是 $S(i) = \sum_{i=1}^{n} a_i$
- 預處理 $S(i) = a_i + S(i-1)$
- 想要知道某一個區間 $S(r) - S(l-1)$
- 預處理 $O(N)$，查詢 $O(1)$

Code

```cpp
// by. MiohitoKiri5474
#include<bits/stdc++.h>

using namespace std;

#define max N 100005

int pre[maxN];

int main() {
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, l, r;
    cin >> n >> m;
    for ( int i = 1 ; i < n ; i++ ) {
        cin >> pre[i];
        pre[i] += pre[i - 1];
    }
    while ( m-- ) {
        cin >> l >> r;
        cout << pre[r] - pre[l - 1] << '\n';
    }
}
```

假設題目做了點變化：題目 #2

- 給定一初始序列，需要能夠執行以下兩種操作
  - 1. 修改成員的值
  - 2. 查詢區間 $[l, r]$ 內的最大值
- 序列長度 $\leq 10^5$
- 題目連結可以看這邊

於是手順刻了這東西

```cpp
#include<bits/stdc++.h>
using namespace std;
int lib[100005];
int main() {
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );
    int n, m, l, r, type, ma;
    cin >> n >> m;
    for ( int i = 0 ; i < n ; i++ )
        cin >> lib[i];
    while ( m-- ) {
        cin >> type >> l >> r;
        if ( type == 1 )
            lib[l] = r;
        else {
            ma = -1;
            for ( int i = l ; i <= r ; i++ )
                ma = max ( lib[i], ma );
            cout << ma << '\n';
        }
    }
}
```

然後就TLE了

oj.leo900807.tw/submissions/3032

NCKU Online Judge | Bulletin | Problems | Submissions | Contests | Discuss | About | Rank | lltzpp | Sign out

Results of submission #3032

Back to Submissions List | Rejudge | Edit | Destroy

Problem	Total Time (ms)	Max Memory (KiB)	Verdict	Score
9. Seeeeeeeeeeeeeg	50052	3600	Time Limit Exceeded	0

Subtask Results

Subtask no.	Testdata Range	Score
1	0~54	0 / 100

Testdata Results

Testdata no.	Subtasks	Time (ms)	Memory (KiB)	Verdict	Score
0	1	5	3300	Accepted	100
1	1	5	3460	Accepted	100
2	1	5	3520	Accepted	100
3	1	6	3468	Accepted	100
4	1	5	3488	Accepted	100
5	1	5	3444	Accepted	100
6	1	5	3468	Accepted	100
7	1	4	3348	Accepted	100
8	1	6	3296	Accepted	100

先看個複雜度

- emmm, 看起來是 $O(NM)$, 看起來要 $O(N\log N)$ 才會過

```
while ( m-- ) {
    cin >> type >> l >> r;
    if ( type == 1 )
        lib[l] = r;
    else {
        ma = -1;
        for ( int i = l ; i <= r ; i++ )
            ma = max ( lib[i], ma );
        cout << ma << '\n';
    }
}
```

Segment Tree

此線段樹非彼線段樹

- 如果在 google 上直接敲線段樹會出現這東西

zh.wikipedia.org/wiki/線段樹

維基百科
自由的百科全書

首页
分类索引
特色内容
新闻动态
最近更改
随机条目

帮助
帮助
维基社群
方针与指引
互助客栈
知识问答
字词转换
IRC即时聊天
联络我们
关于维基百科
资助维基百科

打印/导出
下载为PDF
打印页面

工具
链入页面
相关更改
上传文件
特殊页面
固定链接
页面信息
维基数据项
引用本页
左侧跳顶连接
短链接

没有登录 讨论 贡献 创建账户 登录

条目 讨论 不转换 汉 漢 阅读 编辑 查看历史 搜索维基百科

維基台北寫作聚於每月第二個禮拜六舉行，歡迎報名參與 [关闭]

線段樹 [编辑]

维基百科，自由的百科全书

本文介紹的是一種儲存區間的資料結構。關於一種提供區間查詢的資料結構，請見「线段树 (区间查询)」。

此條目没有列出任何参考或来源。(2016年9月19日)
維基百科所有的內容都應該可供查證。请协助添加来自可靠来源的引用以改善这篇条目。无法查证的内容可能被提出异议而移除。

線段樹（英語：Segment tree）是一種二元樹形資料結構，1977年由Jon Louis Bentley發明[1]，用以儲存區間或線段，並且允許快速查詢結構內包含某一點的所有區間。

一個包含n個區間的線段樹，空間複雜度為$O(n)$，查詢的時間複雜度則為$O(\log n + k)$，其中k是符合條件的區間數量。

此資料結構亦可推廣到高維度。

結構 [编辑]

本處以一維的線段樹為例。

令S是一維線段的集合。將這些線段的端點坐標由小到大排序，令其為$x_1, x_2, \cdots, x_m$。我們將被這些端點切分的每一個區間稱為「單位區間」（每個端點所在的位置會單獨成為一個單位區間），從左到右包含：

$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], \ldots, (x_{m-1}, x_m), [x_m, x_m], (x_m, +\infty)$$

線段樹的結構為一個二元樹，每個節點都代表一個坐標區間，節點N所代表的區間記為Int(N)，則其需符合以下條件：

- 其每一個葉節點，從左到右代表每個單位區間。
- 其內部節點代表的區間是其兩個兒子代表的區間之聯集。
- 每個節點（包含葉子）中有一個儲存線段的資料結構。若一個線段S的坐標區間包含Int(N)但不包含Int(parent(N))，則節點N中會儲存線段S。

線段樹結構示意，其儲存的線段顯示在圖片的下方

参考資料 [编辑]

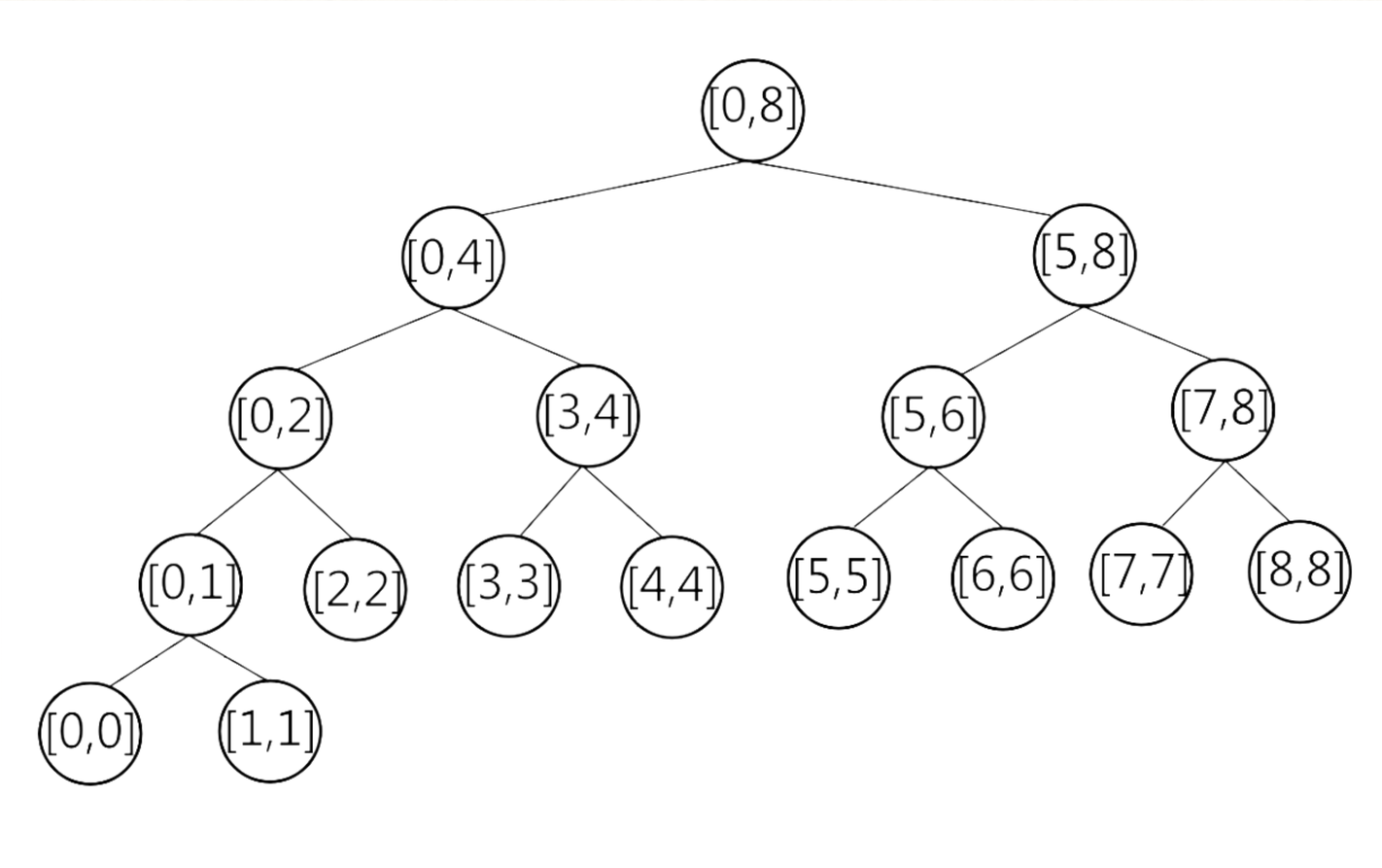
1. ^ (de Berg 等人 2000，p.229)
- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried. More Geometric Data

線段樹

- 由競程選手自行開發出的資料結構
- 為樹狀結構，每個節點都維護一個區間，將此區間對切往左右遞迴給左右子節點，並且合併結果
- 可以處理區間操作

好像有點難懂

直接看



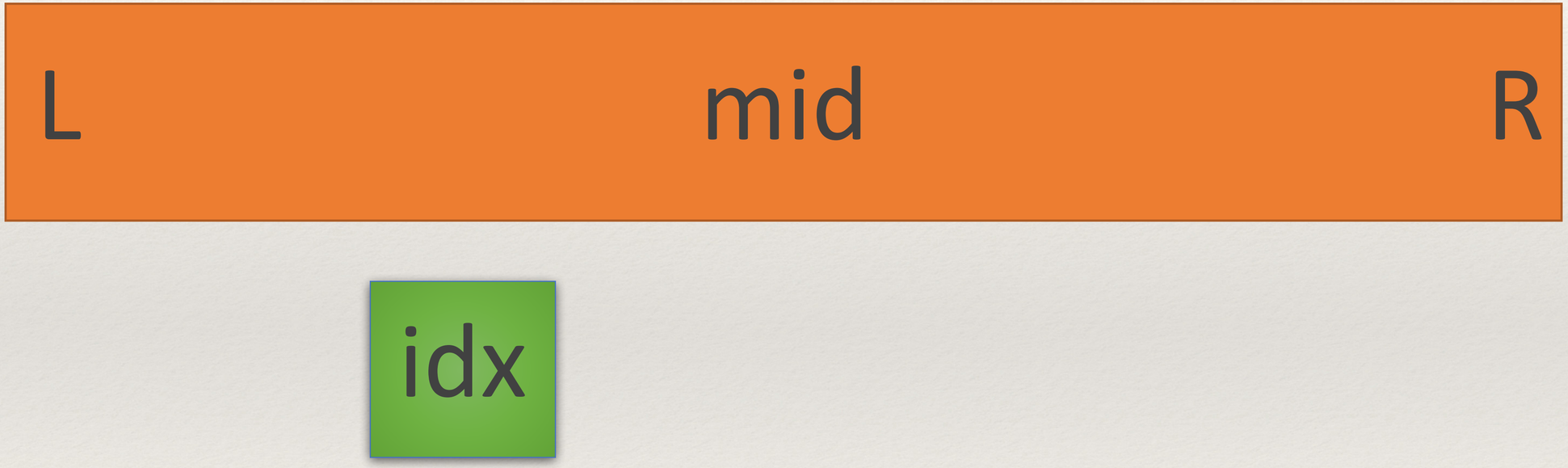
Functions of Segment Tree

- build
- update (modify)
- query

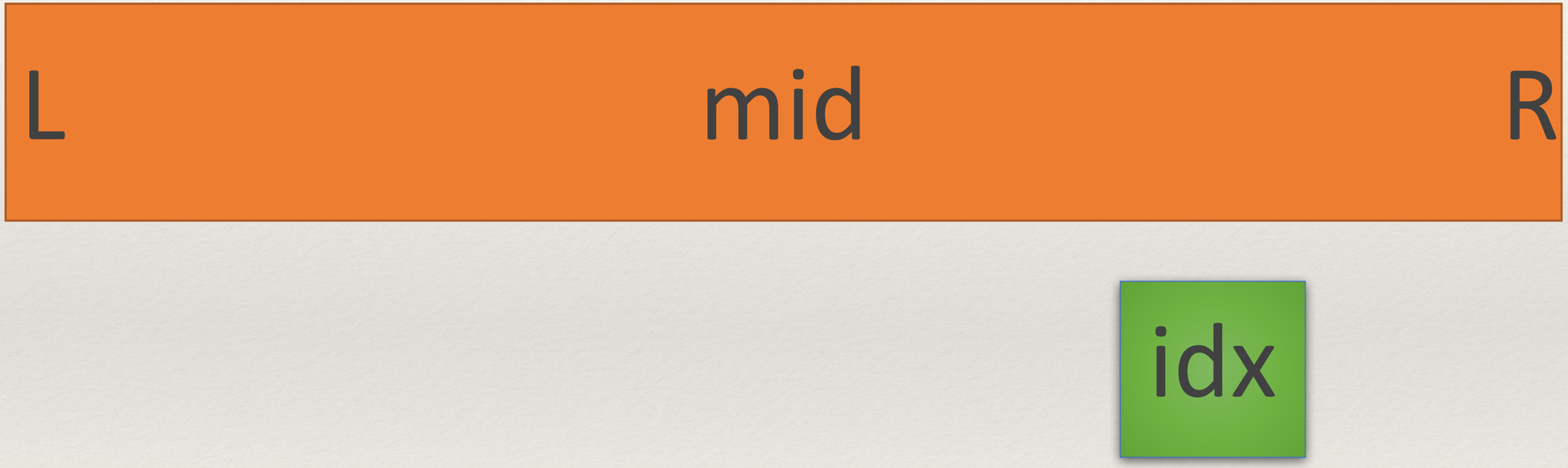
update (modify)

- 更新元素用
- 查詢需修改位置位於何處，往左往右遞迴
- 假設當前區間大小為一，則更新值

Case 1



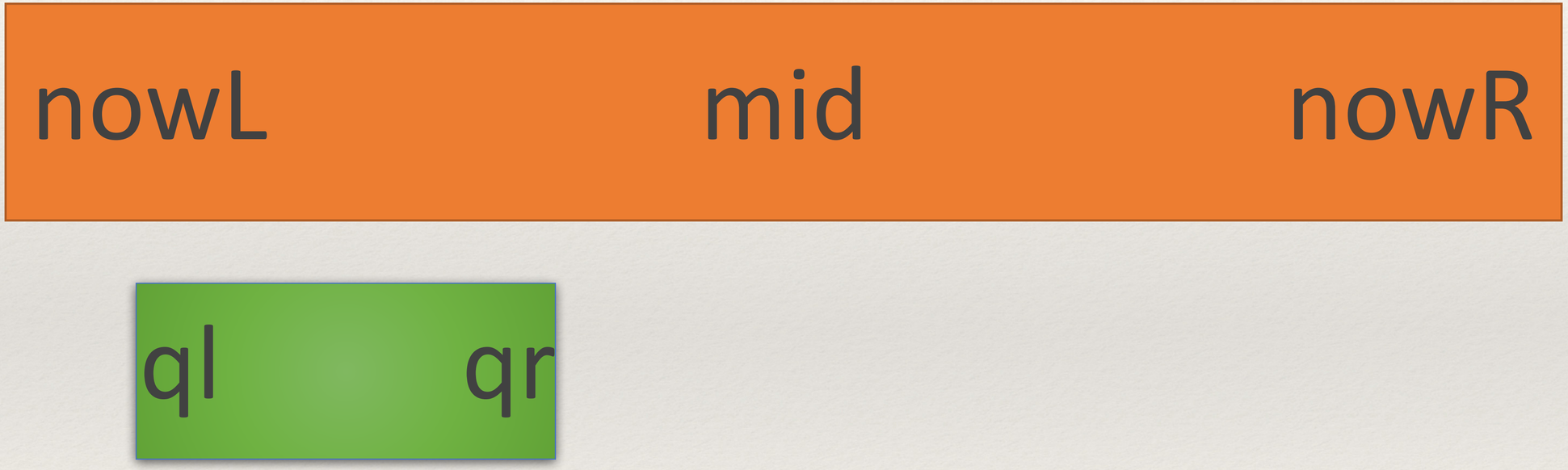
Case 2



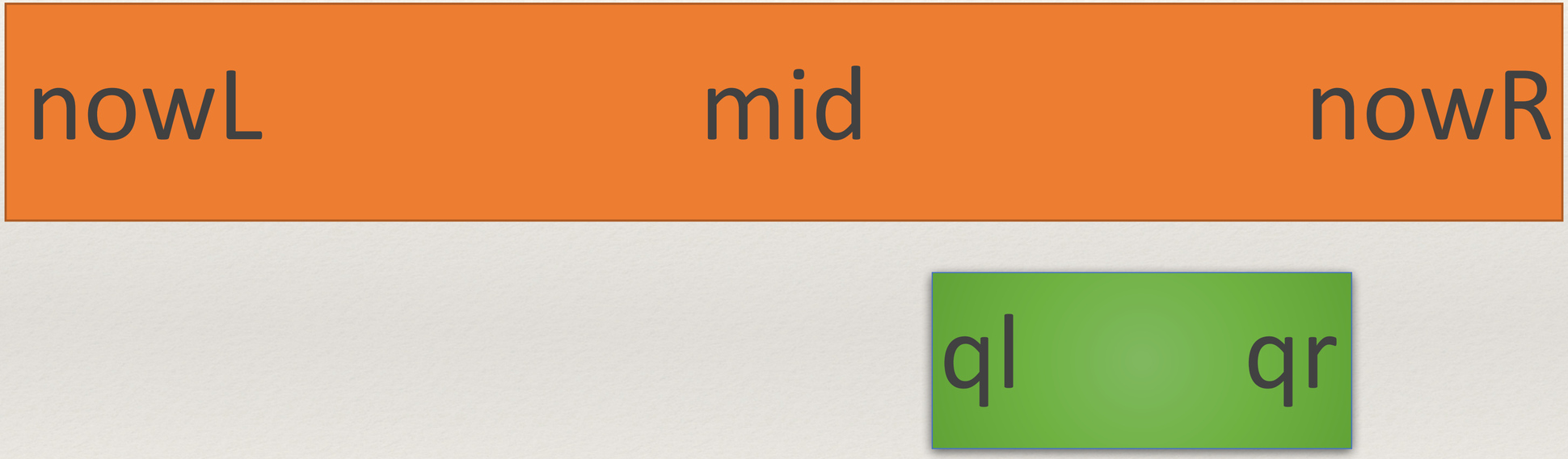
query

- 查詢區間的答案
- 檢查區間位置來決定操作，大致上來說是往左右遞迴

Case 1



Case 2



Case 3

build

- 建構線段樹
- 把左右子區間往下遞迴
- 把左右子區間的結果合併（有點 merge sort 的概念）
- 也可以把所有點都單點修改一次啦

線段樹實作

- 指標
- 陣列

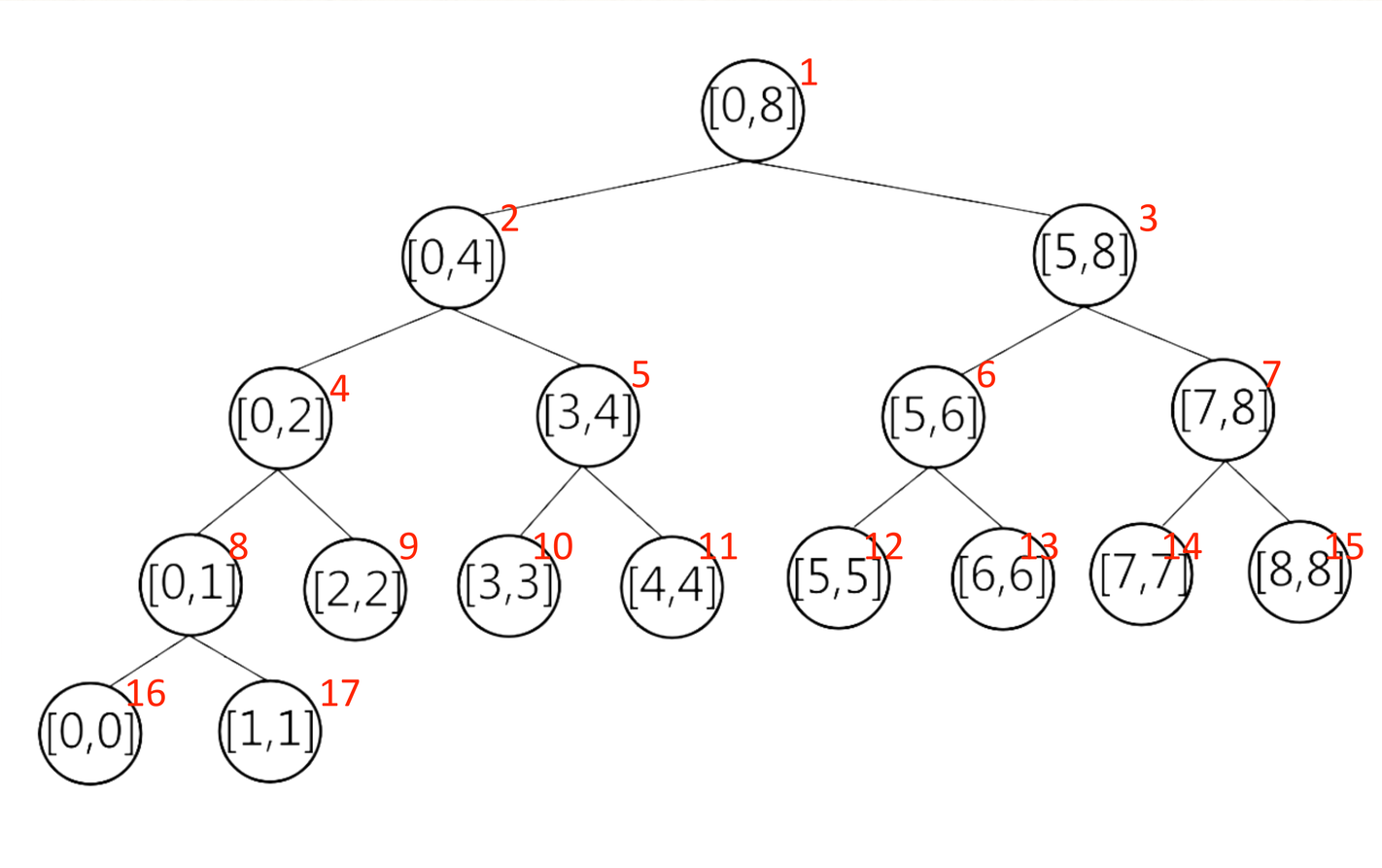
指標版本

- 每一個節點存著左子節點與右子節點的位置
- 葉節點的左子節點與右子節點皆為 NULL (nullptr in C++11 or later version)

陣列版本

- 相較於指標版本，陣列可以一次把記憶體開好開滿
- 如果現在的節點編號為 N
  - 左子節點編號為 $2N$，右邊為 $2N+1$
    （編號從 1 開始的情況）
  - 左子節點編號為 $2N+1$，右邊為 $2N+2$
    （編號從 0 開始的情況）

陣列版本：編號


[0,8]
1
[0,4]
2
[5,8]
3
[0,2]
4
[3,4]
5
[5,6]
6
[7,8]
7
[0,1]
8
[2,2]
9
[3,3]
10
[4,4]
11
[5,5]
12
[6,6]
13
[7,7]
14
[8,8]
15
[0,0]
16
[1,1]
17

Code

```
void update ( int l, int r, int index, int value, int n ) {
    // 當前區間左邊界、當前區間右邊界、欲更新位置、欲更新位置之值、目前節點編號
    if ( l == r )
        seg[n] = value;
    else {
        int mid = ( l + r ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        // int mid = ( l + r ) / 2, leftSon = n * 2, rightSon = leftSon + 1;
        if ( index <= mid ) // case 1
            update ( l, mid, index, value, leftSon );
        else // case 2
            update ( mid + 1, r, index, value, rightSon );

        seg[n] = max ( seg[leftSon], seg[rightSon] );
        // seg[n] = up ( seg[leftSon], seg[rightSon] );
    }
}
```

Code

```
int query ( int l, int r, int nowL, int nowR, int n ) {
    // 欲查詢範圍左邊界、欲查詢範圍右邊界、當前區間左邊界、當前區間右邊界、當前區間編號
    if ( l <= nowL && nowR <= r )
        return seg[n];
    int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
    if ( r <= mid ) // case 1
        return query ( l, r, nowL, mid, leftSon );
    if ( mid < l ) // case 2
        return query ( l, r, mid + 1, nowR, rightSon );

    // case 3
    return max ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
    // up ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
}
```

分析一下複雜度

- update (modify), query
  - 跟二分搜一樣的概念 $\rightarrow O(\log N)$
- 如果題目有多筆（M）查詢
  - 總複雜度 $\rightarrow O(M \log N)$

回頭看題目 #1的變化：題目 #3

- 給定一初始序列，並有 Q 筆操作

  1. 查詢區間 $[l, r]$ 內的和
  2. 修改其中一個元素

- $|\text{序列長度}| = N, Q \leq 10^5$
- $1 \leq l, r \leq N$

前綴和88

- 前綴和一旦修改其中一個值，就需要修改 $S(i) \sim S(n)$
  $\rightarrow$ 複雜度 $O(N)$，聽起來跟暴力差不多
- 也可以用線段樹寫，不過線段樹的 coding 量（ry

~~樹狀數組（Fenwick Tree）~~

二元索引樹（Binary Indexed Tree, BIT）

BIT

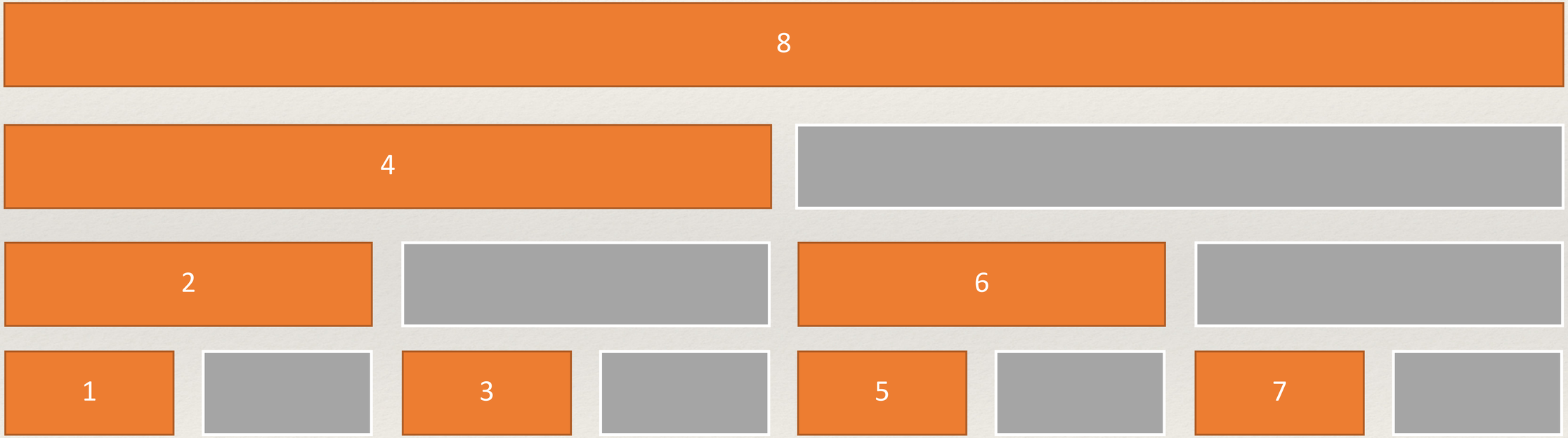
- 由 Peter M. Fenwick 於 1994 年提出
- 結構上就是把線段樹的右子樹拔掉

等等，拔掉右子樹？！

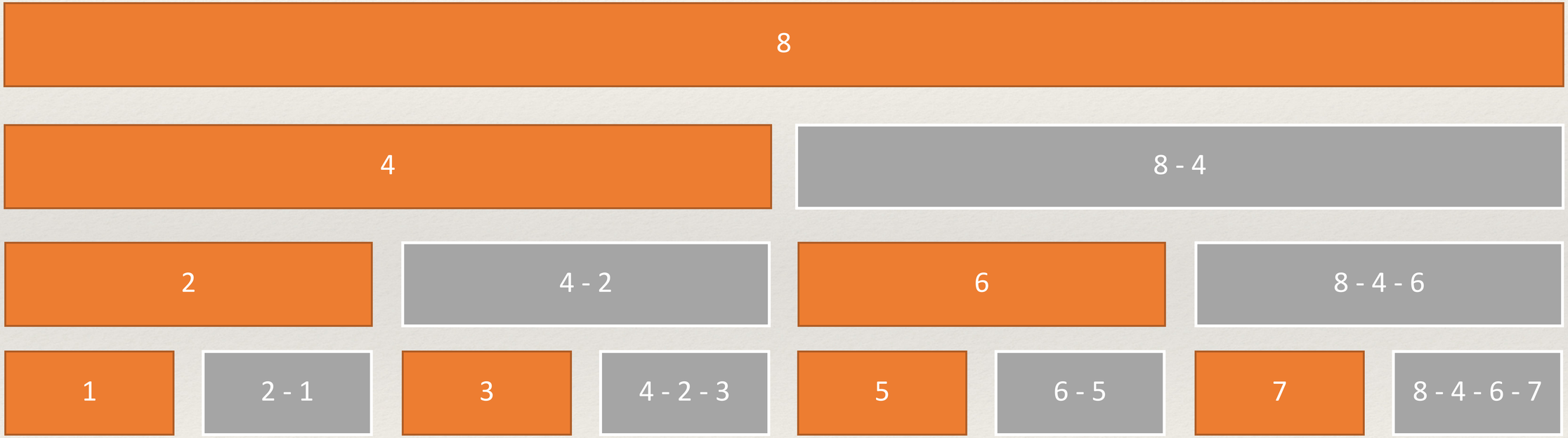
回想一下

- 由前面前綴和的觀念，我們知道 $a_i = S(i) - S(i-1)$
- 所以如果把右子樹都拔掉，也不影響

不對不對，感覺還是怪怪的



那麼這樣呢



用數學表達的話

- $$BIT_i = \sum_{j=i-lowbit(i)+1}^{n} a_j$$
- lowbit:
  - 現在這個數字二進位中，最低位的 1 所代表的值

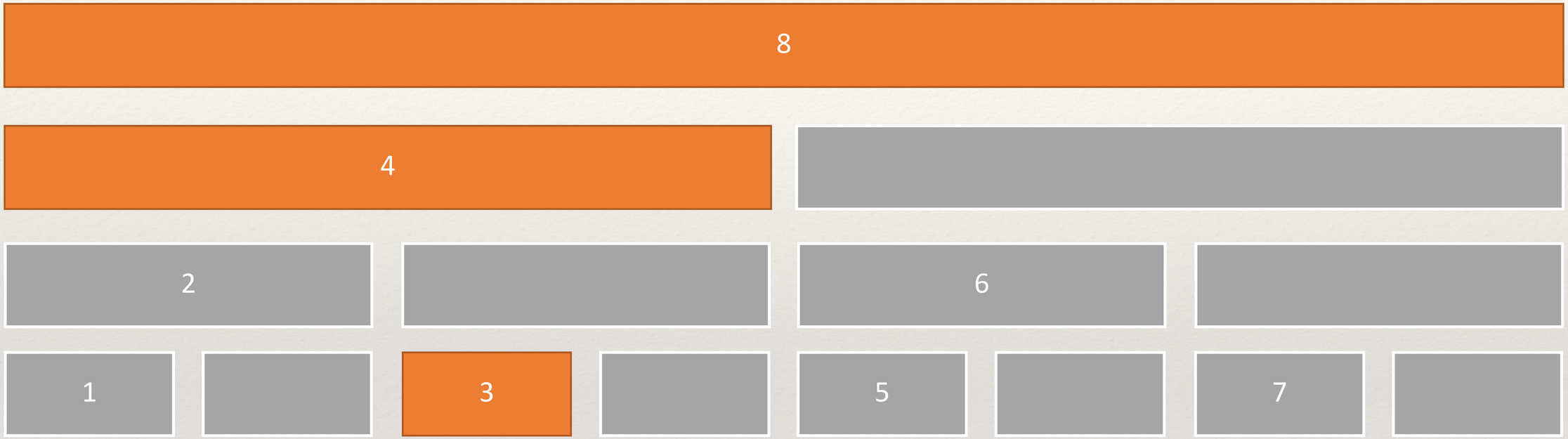
    for example: $lowbit(1) = 1, lowbit(2) = 2, lowbit(3) = 1$
  - 在 C++ 可以用位元運算達到這件事（i & -i）

修改 (add)

- 將所有包含 idx 的區間加上變異量
-
```
while ( idx <= n ){
    bit[idx] += delta;
    idx += idx & -idx;
}
```
- 修改一個點有 $\log N$ 個區間

  $\rightarrow$ 複雜度 $O(\log N)$

修改 a_3



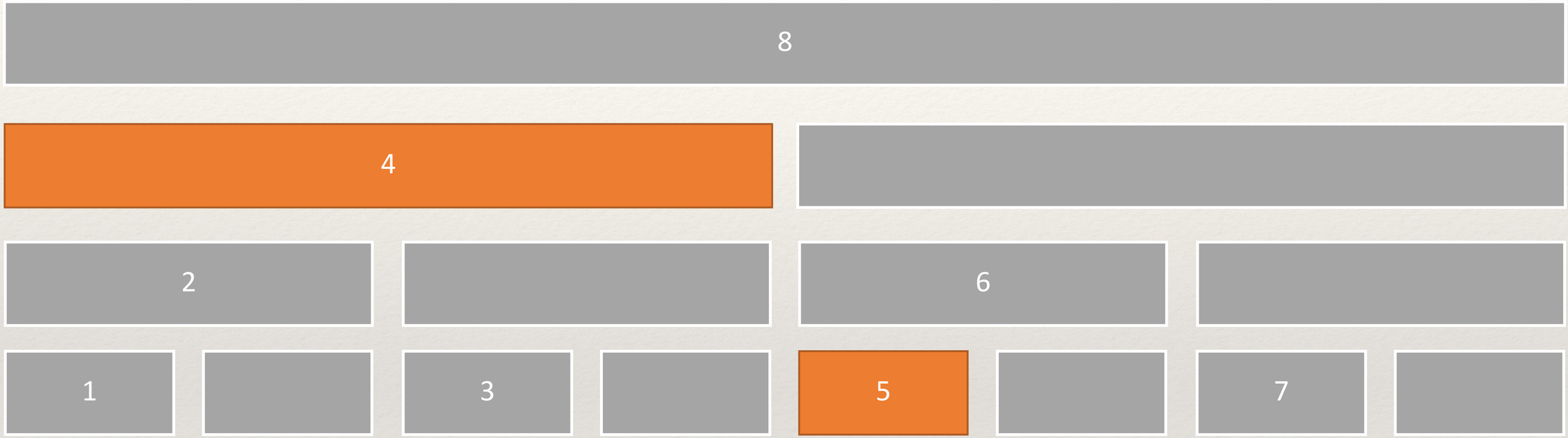
- $3 = (11)_2 \rightarrow 3 + lowbit(3) = (11)_2 + (1)_2 = 4(100)_2$
  $4 = (100)_2 \rightarrow 4 + lowbit(4) = (100)_2 + (100)_2 = 8(1000)_2$

查詢 (sum)

- 將有覆蓋到的範圍都加起來
- while (idx){
  sum += bit[idx];
  idx -= idx & -idx;
  }
- 最多會覆蓋到 $\log N$ 個區間

  $\rightarrow$ 複雜度 $O(\log N)$

查詢 $S(5)$



- $5 = (101)_2 \rightarrow 5 - lowbit(5) = (101)_2 - (1)_2 = 4(100)_2$
  $4 = (100)_2 \rightarrow 4 - lowbit(4) = (100)_2 - (100)_2 = 0(0)_2$

完整的 code

```cpp
// by. Miohitokiri5474
#include<bits/stdc++.h>

using namespace std;

#define maxN 100005

int bit[maxN], basic[maxN];
int n;

void add ( int idx, int delta ) {
    while ( idx <= n ) {
        bit[idx] += delta;
        idx += idx & -idx;
    }
}

int sum ( int idx ) {
    int res = 0;
    while ( idx ){
        res += bit[idx];
        idx -= idx & -idx;
    }

    return res;
}
```

```cpp
int main() {
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int m, l, r, type;
    cin >> n >> m;
    for ( int i = 1 ; i <= n ; i++ ) {
        cin >> basic[i];
        add ( i, basic[i] );
    }

    while ( m-- ) {
        cin >> type >> l >> r;
        if ( type == 1 ) {
            add ( l, r - basic[l] );
            basic[l] = r;
        }
        else
            cout << sum ( r ) - sum ( l ) << '\n';
    }
}
```

還有一個題目 #4

- 給定一初始序列，並有 Q 筆操作

  查詢區間 $[l, r]$ 內的最大值

  區間 $[l, r]$ 的元素加上 k

- $|\text{序列長度}| = N, Q \leq 10^5$
- $1 \leq l, r \leq N$

呃...對 $[l, r]$ 做單點修改？

- Emmm好像不是不可行
- 那先看一下複雜度， $O(N\log N)$， 可能要做 Q 次

  $\rightarrow$ 最差 $O(NQ\log N)$ 聽起來有點糟

Segment Tree with Lazy Tag

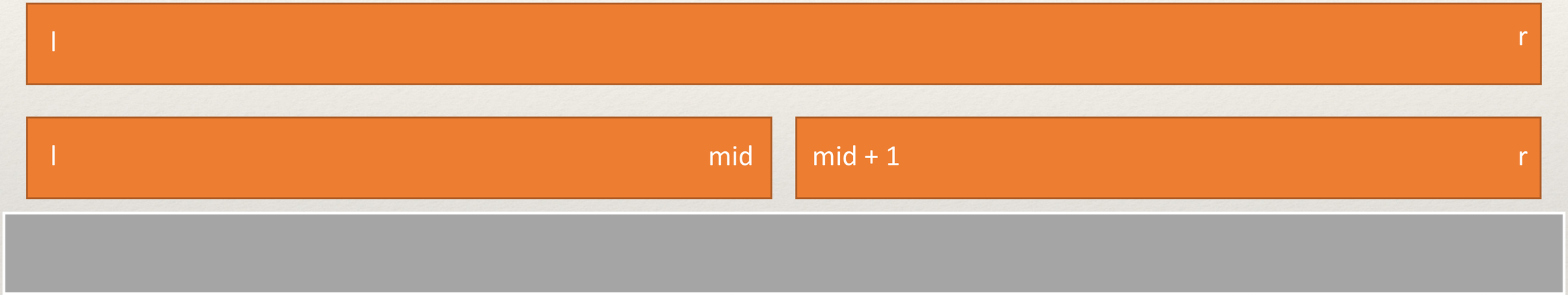
Lazy Tag

- 每次操作不是真的操作
- 只有當前節點範圍 $[nowL, nowR]$ 包含於區間修改區間 $[l, r]$ 內，才會對這個節點打上標記，否則往左與右節點遞迴
- 減少修改次數，假設現在一個區間要修改 k 次，如果狀態可以疊加那麼一次推下去的效率顯然比更新 k 次好得多

Lazy Tag 實作

- 一個節點會有兩個變數，一個是紀錄 Lazy Tag，另外一個是紀錄原本的值
- 通常個人會用 pair，first 是紀錄最大值，second 是 tag

Case 1



Case 2

Case 3

push

- 將懶惰標記往下推
- 在原本的函數中，要往下一層遞迴時先把標記 push 下去

```
void push ( int n ) {
    if ( seg[n].S ) {
        int leftSon = n << 1, rightSon = n << 1 | 1, value = seg[n].S;
        seg[n].S = 0;
        seg[leftSon].F += value, seg[rightSon].F += value;
        seg[leftSon].S += value, seg[rightSon].S += value;
    }
}
```

區間修改 (modify)

```
void modify ( int l, int r, int nowL, int nowR, int value, int n ){
    if ( l <= nowL && nowR <= r )
        seg[n].F += value, seg[n].S += value;
    else {
        int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n ); // lazy tag
        if ( r <= mid )
            modify ( l, r, nowL, mid, value, leftSon );
        else if ( mid < l )
            modify ( l, r, mid + 1, nowR, value, rightSon );
        else {
            modify ( l, r, nowL, mid, value, leftSon );
            modify ( l, r, mid + 1, nowR, value, rightSon );
        }
        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}
```

code

```cpp
// by. Miohitokiri5474
#include<bits/stdc++.h>

using namespace std;

#define maxN 100005
#define F first
#define S second
typedef pair < int, int > pii;

pii seg[maxN << 2];

void push ( int n ) {
    if ( seg[n].S ) {
        int leftSon = n << 1, rightSon = n << 1 | 1, value = seg[n].S;
        seg[n].S = 0;
        seg[leftSon].F += value, seg[rightSon].F += value;
        seg[leftSon].S += value, seg[rightSon].S += value;
    }
}

void update ( int l, int r, int index, int value, int n ) {
    if ( l == r )
        seg[n].F = value;
    else {
        int mid = ( l + r ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n ); // lazy tag
        if ( index <= mid )
            update ( l, mid, index, value, leftSon );
        else
            update ( mid + 1, r, index, value, rightSon );

        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}
```

```cpp
int query ( int l, int r, int nowL, int nowR, int n ){
    if ( l <= nowL && nowR <= r )
        return seg[n].F;
    int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
    push ( n ); // lazy tag
    if ( r <= mid )
        return query ( l, r, nowL, mid, leftSon );
    if ( mid < l )
        return query ( l, r, mid + 1, nowR, rightSon );
    return max ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
}

void modify ( int l, int r, int nowL, int nowR, int value, int n ){
    if ( l <= nowL && nowR <= r )
        seg[n].F += value, seg[n].S += value;
    else {
        int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n ); // lazy tag
        if ( r <= mid )
            modify ( l, r, nowL, mid, value, leftSon );
        else if ( mid < l )
            modify ( l, r, mid + 1, nowR, value, rightSon );
        else {
            modify ( l, r, nowL, mid, value, leftSon );
            modify ( l, r, mid + 1, nowR, value, rightSon );
        }
        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}
```

code (con’t)

```
int main(){
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, l, r, in, type;
    cin >> n >> m;

    // build
    for ( int i = 1 ; i <= n ; i++ ){
        cin >> in;
        update ( 1, n, i, in, 1 );
    }

    while ( m-- ) {
        // type 1: 單點修改
        // type 2: 區間查詢
        // type 3: 區間修改
        cin >> type >> l >> r;
        if ( type == 1 ) {
            update ( 1, n, l, r, 1 );
        }
        else if ( type == 2 )
            cout << query ( l, r, 1, n, 1 ) << '\n';
        else {
            cin >> in;
            modify ( l, r, 1, n, in, 1 );
        }
    }
}
```

延伸題目：大龍貓

- 先上連結：https://toj.tfcis.org/oj/pro/365/
- 給定一個序列長度為 N 的序列，有 M 筆操作
  - 1. 修改其中一個元素的值
  - 2. 查詢區間中最長的「好序列」長度
- 來定義一下好區間：該區間中的每個元素，剛好都是前一個元素 +1

RMQ...嗎？

- 稍微想一下就會發現可以用線段樹做
- 每個 node 都紀錄其維護區間中最長的好區間長度，並且在將區間結果合併時，同時確認在兩個區間的交界處是否會行程一個更長的好區間

好多 struct

- 為了能夠在合併時快速的找出交界處是否有好區間
  因此線段樹上的每個 node 會分別紀錄
  - 該區間從頭開始的好區間
  - 該區間從尾開始的好區間
  - 該區間中最長的好區間

合併區間

- 合併區間時，該區間的最大好區間會是以下三個區間中的其中一個
  - 左子節點的最長好區間
  - 右子節點的最長好區間
  - 左右子節點合併處形成的好區間

a PIECE of cake

- 為了方便紀錄，會開一個名稱為 piece 的 struct，紀錄該好區間的資訊
- 還有為了方便確認兩個是否為相同的區間，寫了個 operator（或 function 也行）來確認

```
 8 struct piece {
 9     int l, r, sz;
10
11     // 檢查兩個 piece 是否相同
12     bool operator == ( const piece b ){
13         return l == b.l && r == b.r;
14     }
15 };
16
17 // 不想寫 operator（或是覺得太麻煩）可以這樣寫
18 inline bool same ( piece a, piece b ) {
19     return a.l == b.l && a.r == b.r;
20 }
```

node

- 然後是 node，裡面包三個 piece 紀錄剛剛提到的那些區間

```
struct node {
    piece front, back, ma;
} seg[maxN << 2];
```

merge

```
inline node merge ( node L, node R ) {
    node res;
    res.front = L.front, res.back = R.back, res.ma = ( L.ma.sz > R.ma.sz ? L.ma : R.ma );

    if ( basic[L.back.r] + 1 == basic[R.front.l] ) {
        piece swp = piece { L.back.l, R.front.r, R.front.r - L.back.l + 1 };

        if ( L.front == L.back )
            res.front = swp;
        if ( R.front == R.back )
            res.back = swp;

        res.ma = ( swp.sz > res.ma.sz ? swp : res.ma );
    }

    return res;
}
```

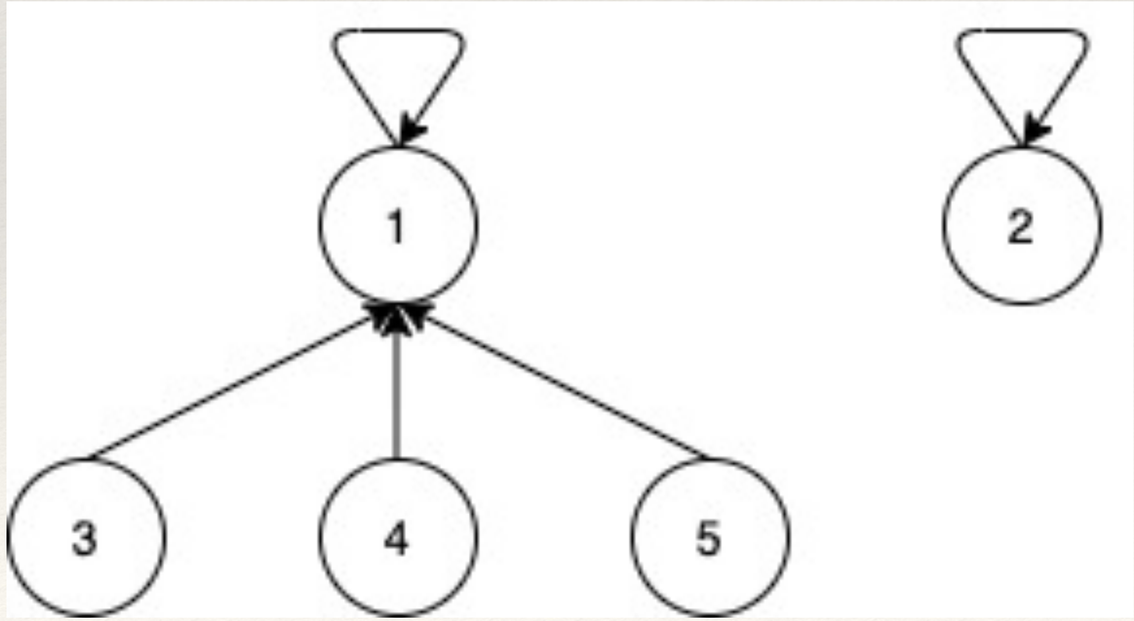
Disjoint Set and Union

Disjoint Set and Union

- 中文名稱為互斥集，其中包含以下功能
  - 對於集合 P 中的任意元素 p 都恰有一個集合包含該元素
  - 可以用 same (a, b) 確認兩個元素是否於同一個集合中
  - 可以用 Union (a, b) 合併兩個元素所屬的兩個集合

Disjoint Set with Union

- 用領袖選舉的方式來處理 dsu，開一個陣列紀錄該節點的上層領袖是誰
- 如果某一節點為該集合的最高領導者，則該節點的領導者為自己



function of DSU

- 在寫並查集的時候，通常會有以下四個 function
  - init
  - find
  - Union
  - same

init

- 初始化，先將所有節點的領導者設定為自己

```
inline void dsu::init ( void ) {
    for ( int i = 0 ; i < maxN ; i++ )
        dis[i] = i;
}
```

find

- 尋找節點的最高領導者，藉由遞迴的方式直到 dis[n] == n

```
int dsu::find ( int n ) {
    if ( dis[n] == n )
        return n;
    return find ( dis[n] );
}
```

Union

- 將兩個集合合併，只要將其中一個集合的最高領導者改為另一個集合的最高領導者即可

```
inline void dsu::Union ( int a, int b ) {
    dis[find ( a )] = find ( b );
}
```

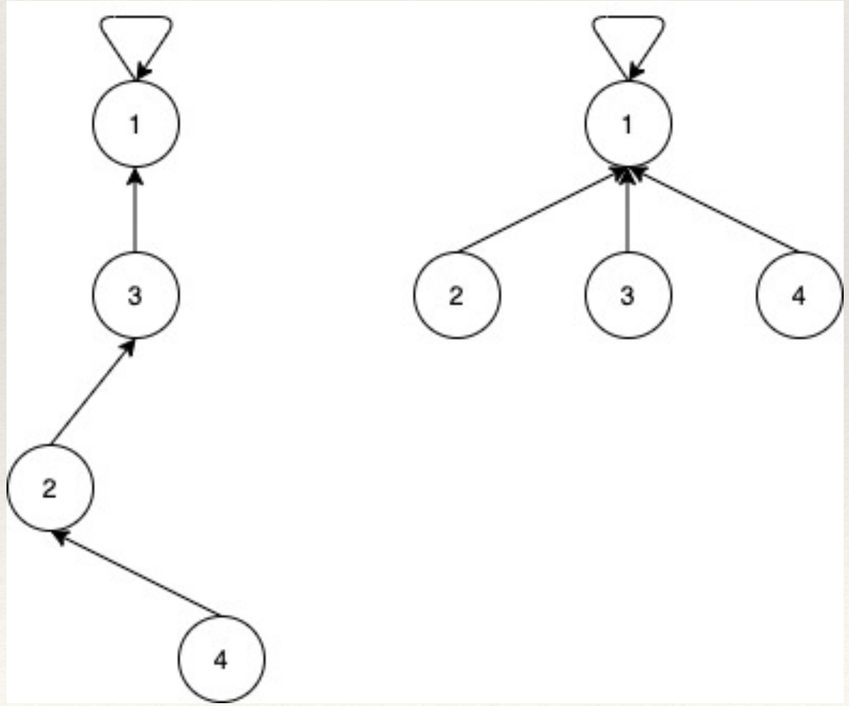
same

- 確認兩個元素是否在同一個集團中，只要檢查兩個的 find 是否一樣即可

```
inline bool dsu::same ( int a, int b ) {
    return find ( a ) == find ( b );
}
```

路徑壓縮

- 想一下會發現 find 的複雜度極差（$O(N)$）
- 因此通常使用時會加上一些優化方法，最簡單的便是路徑壓縮
- 路徑壓縮為在往回 find 的同時紀錄結果



路徑壓縮

- 改用路徑壓縮後的複雜度會降成 $O(logN)$

```
// path compression version
int dsu::find ( int n ) {
    if ( dis[n] == n )
        return n;
    return dis[n] = find ( dis[n] );
}
```

Union by Rank

- 雖然一般來說使用路徑壓縮的 dsu 就足夠了，不過還是無法避免 find 時遞迴過深的問題
- 這時候可以順便紀錄每個節點的 rank，在 Union 時將 rank 小集合接在大的集合上，使得深度平均
- 使用路徑壓縮以及 union by rank 的 dsu，複雜度下降到驚人的 $O(\alpha(N))$

Union by Rank

```
// Union by Rank version
inline void dsu::Union2 ( int a, int b ) {
    if ( same ( a, b ) )
        return;
    a = find ( a ), b = find ( b );
    if ( rank[a] < rank[b] )
        swap ( a, b );
    dis[[dis[b]] = dis[a];
    rank[a]++;
}
```

Take a Break

總而言之還是看個題目 #5

- 給定一初始序列，支援單點修改、區間查詢最大值以及復原操作。

每次修改完複製一棵樹？

- 聽起來有搞頭，但是想一下要燒多少記憶體（頭痛
- 觀念接近了，不過還差一點

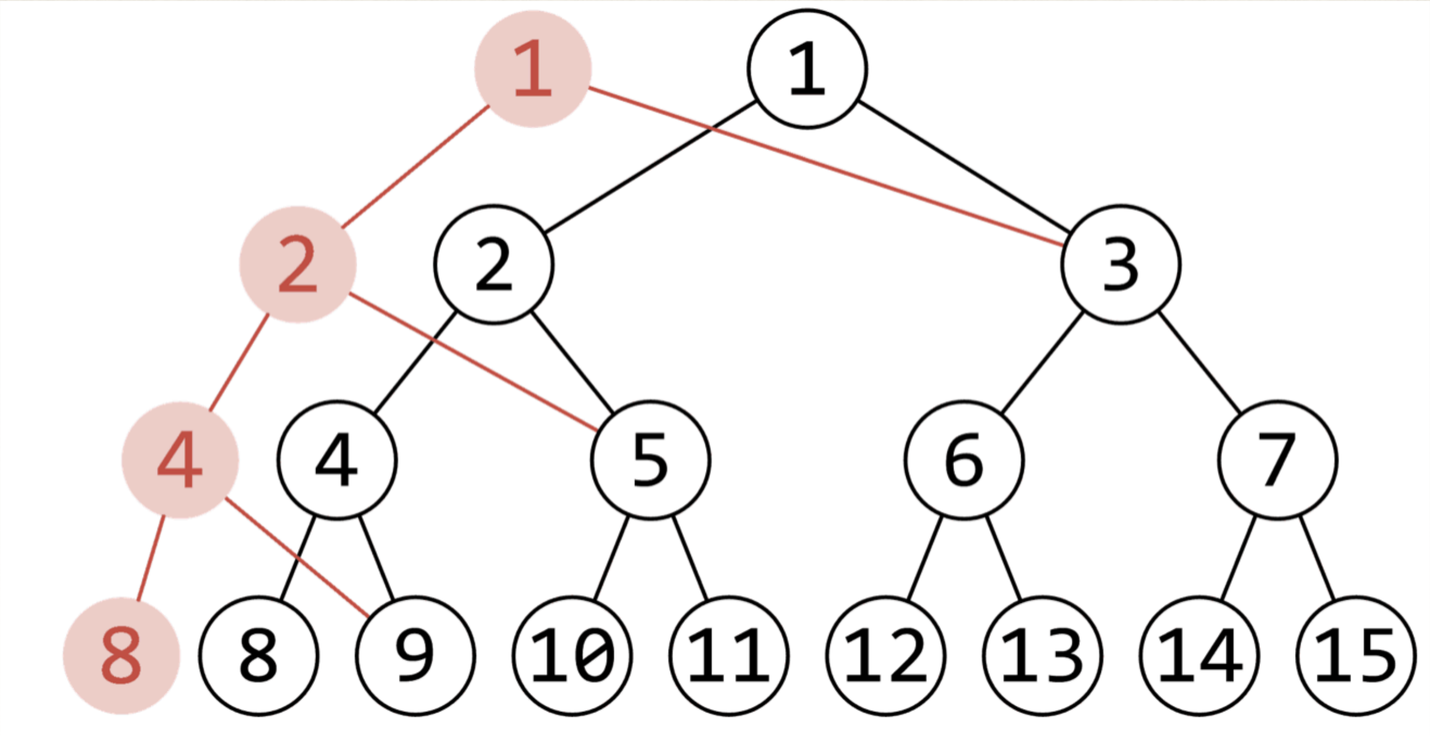
Persistent Segment Tree

持久化線段樹

- 還是線段樹，但與平常的線段樹不太相同，是指標版本
- 藉由指標的性質使線段樹持久化變為可能

持久化線段樹

- 持久化線段樹的觀念很簡單，我們只複製出會更改到的節點，剩下的沿用舊的
- 也就是說沿著遞迴路徑一路開新的節點，並且複製舊的其餘部分



持久化線段樹

- 然後因為 root 一定會是新的指標，因此我們只需要維護 root 的版本即可維護所有版本

線段樹指標化

- 首先我們先將線段樹轉換成指標版本
- 基本上跟原本的線段樹差不多一樣有三個函數要寫
  - build
  - upadte (modify)
  - query

先定義一下 struct

```
struct node {
    node *l, *r;
    int value;

    node ( int _val ): l ( nullptr ), r ( nullptr ), value ( _val ) {}

    inline void up ( void ) {
        value = -1;
        if ( l )
            value = max ( value, l -> value );
        if ( r )
            value = max ( value, r -> value );
    }
} *seg = nullptr;
```

build

```
void build ( int l, int r, node *&o ) {
    o = new node ( 0 );
    if ( l == r )
        return;
    int mid = ( l + r ) >> 1;
    build ( l, mid, o -> l );
    build ( mid + 1, r, o -> r );

    o -> up();
}
```

update (modify)

```
void update ( int l, int r, int index, int value, node *&o ) {
    if ( l == r )
        o -> value = value;
    else{
        int mid = ( l + r ) >> 1;
        if ( index <= mid )
            update ( l, mid, index, value, o -> l );
        else
            update ( mid + 1, r, index, value, o -> r );

        o -> up();
    }
}
```

query

```
int query ( int l, int r, int nowL, int nowR, node *o ) {
    if ( l <= nowR && nowR <= r )
        return o -> value;
    int mid = ( nowL + nowR ) >> 1;
    if ( r <= mid )
        return query ( l, r, nowL, mid, o -> l );
    if ( mid < l )
        return query ( l, r, mid + 1, nowR, o -> r );

    return max ( query ( l, r, nowL, mid, o -> l ), query ( l, r, mid + 1, nowR, o -> r ) );
}
```

對於持久化的改動

```
struct node {
    node *l, *r;
    int value;

    node ( int _val ): l ( nullptr ), r ( nullptr ), value ( _val ) {}
    // 新增一個建構子，可以直接複製原本的 l, r
    node ( node *o ): l ( o -> l ), r ( o -> r ), value ( o -> value ) {}

    inline void up ( void ){
        value = -1;
        if ( l )
            value = max ( value, l -> value );
        if ( r )
            value = max ( value, r -> value );
    }
} *seg = nullptr;
```

對於持久化的改動 (con't)

```
void update ( int l, int r, int index, int value, node *&o ) {
    // 把需要修改的節點在修改前先開一個新位置出來
    o = new node ( o );
    if ( l == r )
        o -> value = value;
    else {
        int mid = ( l + r ) >> 1;
        if ( index <= mid )
            update ( l, mid, index, value, o -> l );
        else
            update ( mid + 1, r, index, value, o -> r );

        o -> up();
    }
}
```

還是一個題目 #6

- 給定一個只有 0、長度為 N 的序列，並且有以下操作
  - 將區間 $[l, r]$ 內的結點都改成 0 或 1
  - 查詢區間內，只由 1 構成且長度最長的區間
- $N \leq 10^9, Q \leq 10^5$
- 詳細內容可以參考 TOJ 的色彩繽紛

memory 君的靈壓快消失了

- 怎麼看都會 MLE，而且 compile 應該就不會過了

離散化？

- 也不是不行
- 但是要先將所有輸入都讀進來，sort 一次
- 每次操作前還要做 lower_bound
- 複雜度算一下大概是 $O(Q(logQ + logN))$，有爆掉的風險

Dynamic Open Point Segment Tree

動態開點...？

- 有使用到的區間才會開位置給他，沒開便假設他是 0
- 然後在 query 時則會檢查這個區間是否存在，不存在則回傳 0

update

```
void update ( int l, int r, int nowL, int nowR, int value, node *&o ) {
    if ( !o )
        o = new node ( 0 );
    if ( o -> val.value == value )
        return;
    if ( o -> val.value != value && o -> val.value == 1 ) {
        if ( o -> l )
            delete o -> l;
        o -> l = new node ( o -> val.value );
        if ( o -> r )
            delete o -> r;
        o -> r = new node ( o -> val.value );
    }
    if ( l <= nowL && nowR <= r ) {
        delete o;
        o = new node ( value );
    }
    else {
        int mid = ( nowL + nowR ) >> 1;
        if ( r <= mid )
            update ( l, r, nowL, mid, value, o -> l );
        else if ( mid < l )
            update ( l, r, mid + 1, nowR, value, o -> r );
        else {
            update ( l, mid, nowL, mid, value, o -> l );
            update ( mid + 1, r, mid + 1, nowR, value, o -> r );
        }

        o -> val = up ( o -> l, o -> r );
    }
}
```

query

```
piece query ( int l, int r, int nowL, int nowR, int value, node *&o ) {
    if ( !o ) {
        if ( value == 1 )
            return piece { 1, 1, 1, 1 };
        else
            return piece { 0, 0, 0, 0 };
    }
    value = o -> val.value;
    if ( l <= nowL && nowR <= r )
        return o -> val;
    int mid = ( nowL + nowR ) >> 1;
    if ( r <= mid )
        return query ( l, r, nowL, mid, value, o -> l );
    if ( mid < l )
        return query ( l, r, mid + 1, nowR, value, o -> r );

    return up ( query ( l, mid, nowL, mid, value, o -> l ), query ( mid + 1, r, mid + 1, nowR, value, o -> r ) );
}
```

又有一個題目 #6

- 對於一個大小為 $N \times M$ 的平面，求其中一區塊的元素中的最大值
- $N, M \leq 10^4$

開 M 棵線段樹？

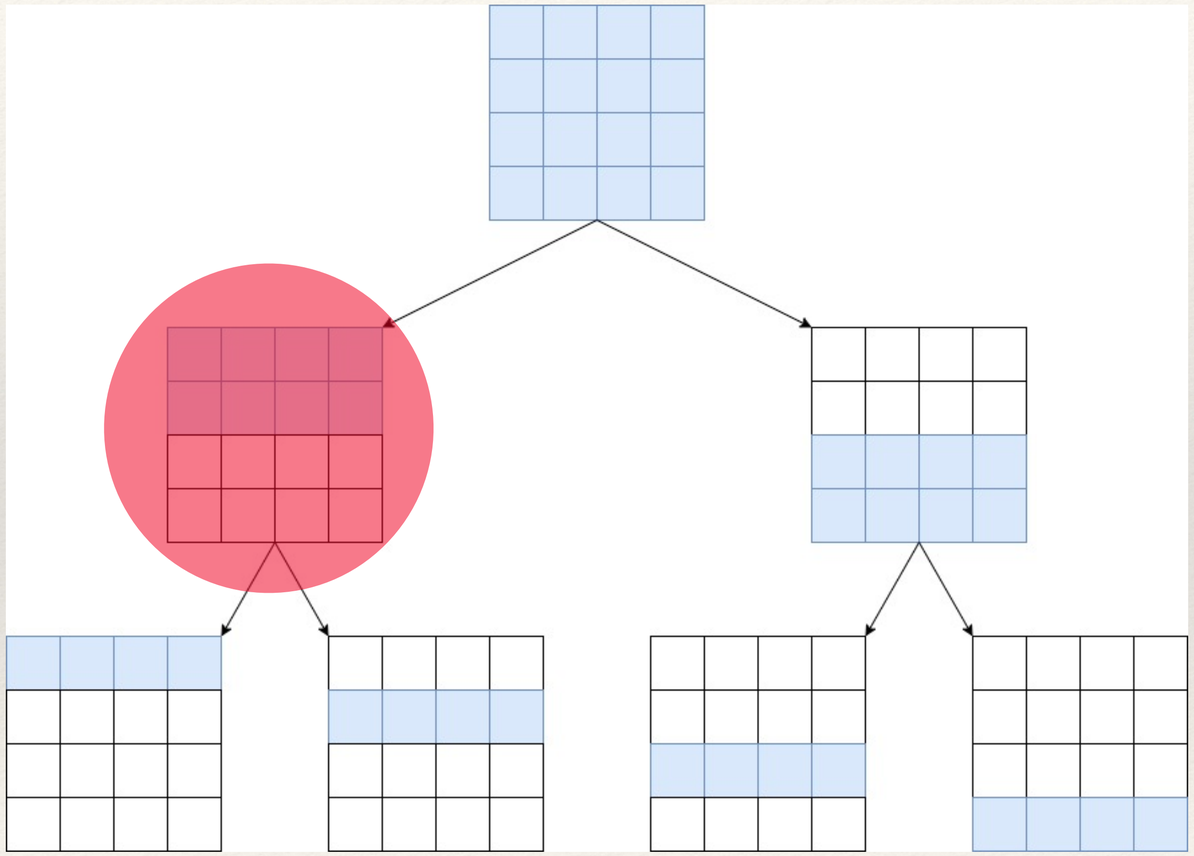
- 聽起來好像不是不行，但是算一下複雜度可能會有點緊（$O(QMlogN)$）

High Dimension Data Structure

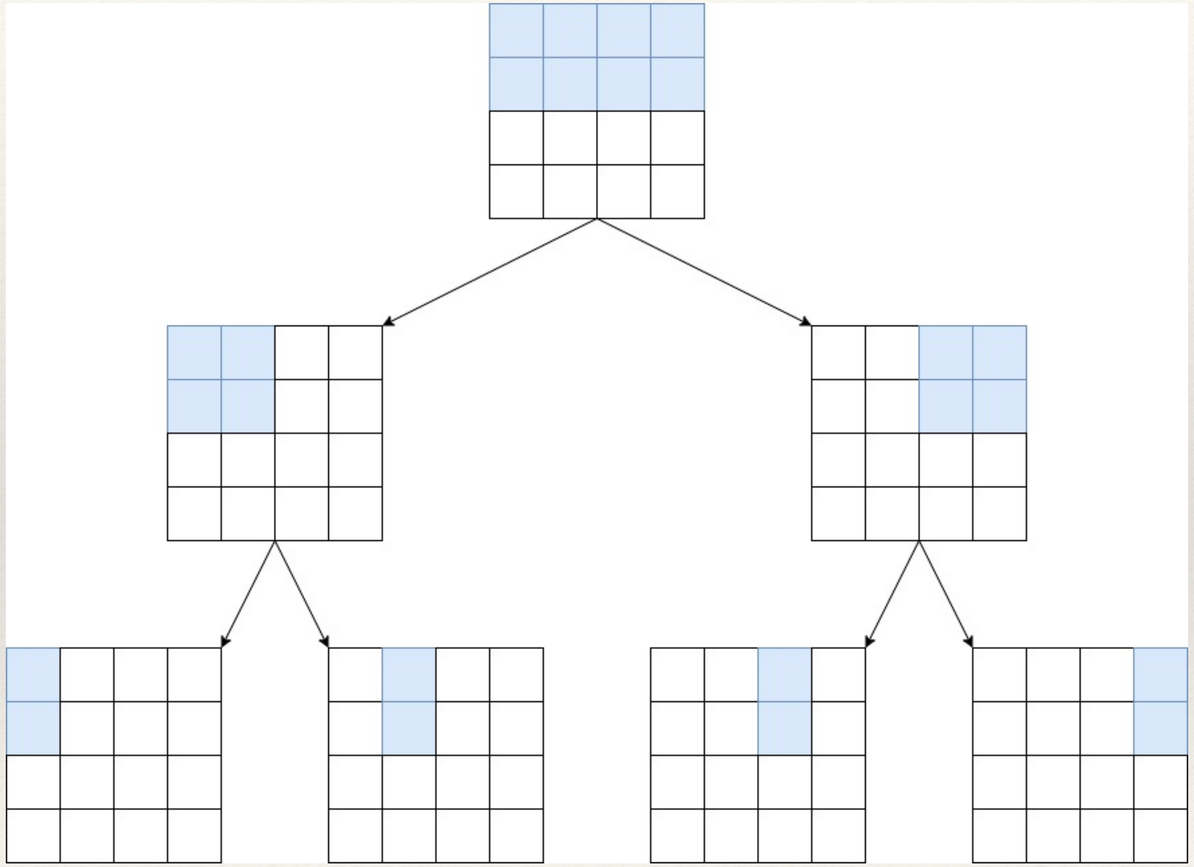
高維度資料結構

- 基本上都算是基礎資料結構的推廣
- 例如二維線段樹的每個節點都是一顆一維線段樹（因此有些人會叫做樹套樹）

用圖來表示的話會像是這樣



其中一個節點的圖



封裝一維線段樹

- 為了能夠好好維護每棵一維線段樹，個人習慣會把一維線段樹包成一個 struct

```
struct node1 {
    int seg[maxN << 2];

    inline void init ( void );
    void update ( int, int, int, int, int );
    int query ( int, int, int, int, int );
    inline void merge ( const node1, const node1, int l, int r, int n );
};
```

node1::init

- 保險起見還是在使用前初始化一下

```
inline void node1::init ( void ) {
    memset ( seg, 0, sizeof seg );
}
```

node1::update & node1::query

- update & query 的部分跟普通的一維線段樹差不多

```
void node1::update ( int l, int r, int index, int value, int n ) {
    if ( l == r )
        seg[n] = value;
    else {
        int mid = ( l + r ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;

        if ( index <= mid )
            update ( l, mid, index, value, leftSon );
        else
            update ( mid + 1, r, index, value, rightSon );

        seg[n] = max ( seg[leftSon], seg[rightSon] );
    }
}

int node1::query ( int l, int r, int nowL, int nowR, int n ) {
    if ( l <= nowL && nowR <= r )
        return seg[n];

    int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;

    if ( r <= mid )
        return query ( l, r, nowL, mid, leftSon );
    if ( mid < l )
        return query ( l, r, mid + 1, nowR, rightSon );
    return max ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
}
```

node1::merge

- 用來合併兩棵一維線段樹，複雜度為 $O(N)$

```
inline void node1::merge ( const node1 &a, const node1 &b, int l, int r, int n ) {
    seg[n] = max ( a.seg[n], b.seg[n] );
    if ( l == r )
        return;
    int mid = ( l + r ) >> 1;
    merge ( a, b, l, mid, n << 1 );
    merge ( a, b, mid + 1, r, ( n << 1 ) | 1 );
}
```

結構定義

- 為了能夠讓二維線段樹被更高維度的地方取用，所以這邊也封裝起來

```
struct node2 {
    node1 seg[maxN << 2];
    int len;

    inline void init ( int, int, int );
    void build ( void );
    void update ( int, int, int, int, int, int );
    int query ( int, int, int, int, int, int, int );
};
```

node2::init & node2::build

- 一樣有初始化跟建構，build 會跑過所有節點並且呼叫該節點的一維線段樹初始化

```
inline void node2::init ( int u, int d, int _len ) {
    len = _len;
    build ( u, d, 1 );
}

void node2::build ( int u, int d, int n ) {
    int mid = ( u + d ) >> 1;
    seg[n].init();
    build ( u, mid );
    build ( mid + 1, d );
}
```

node2::update

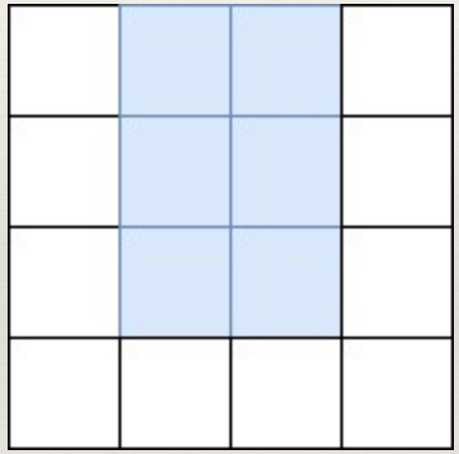
- update 的部分跟一維大同小異

```
void node2::update ( int u, int d, int x, int y, int value, int n ) {
    if ( u == d )
        seg[n].update ( 1, len, y, value, 1 );
    else{
        int mid = ( u + d ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        if ( x <= mid )
            update ( u, mid, x, y, value, leftSon );
        else
            update ( mid + 1, d, x, y, value, rightSon );

        merge ( seg[n], seg[leftSon], seg[rightSon], 1, len, 1 );
    }
}
```

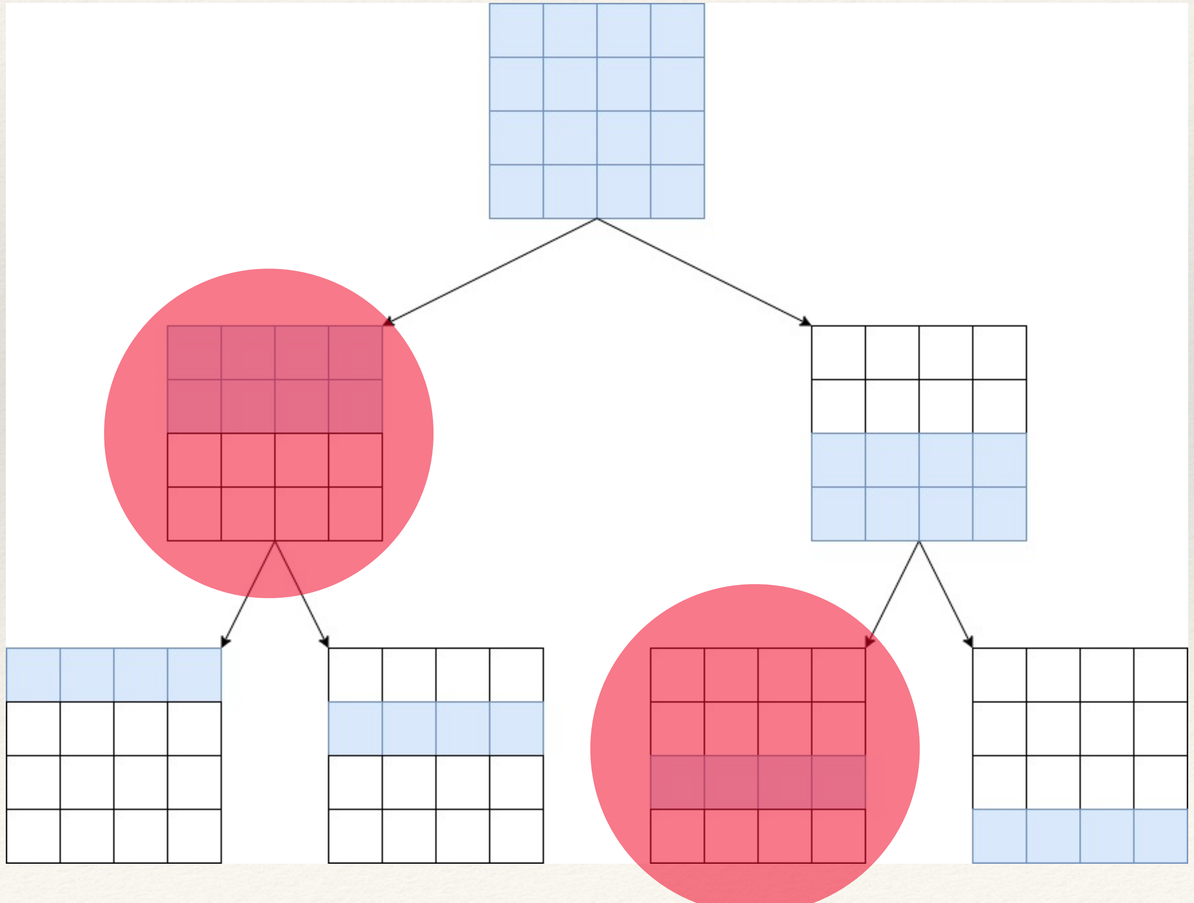
node2::query

- 假設這是我們想要查詢的範圍



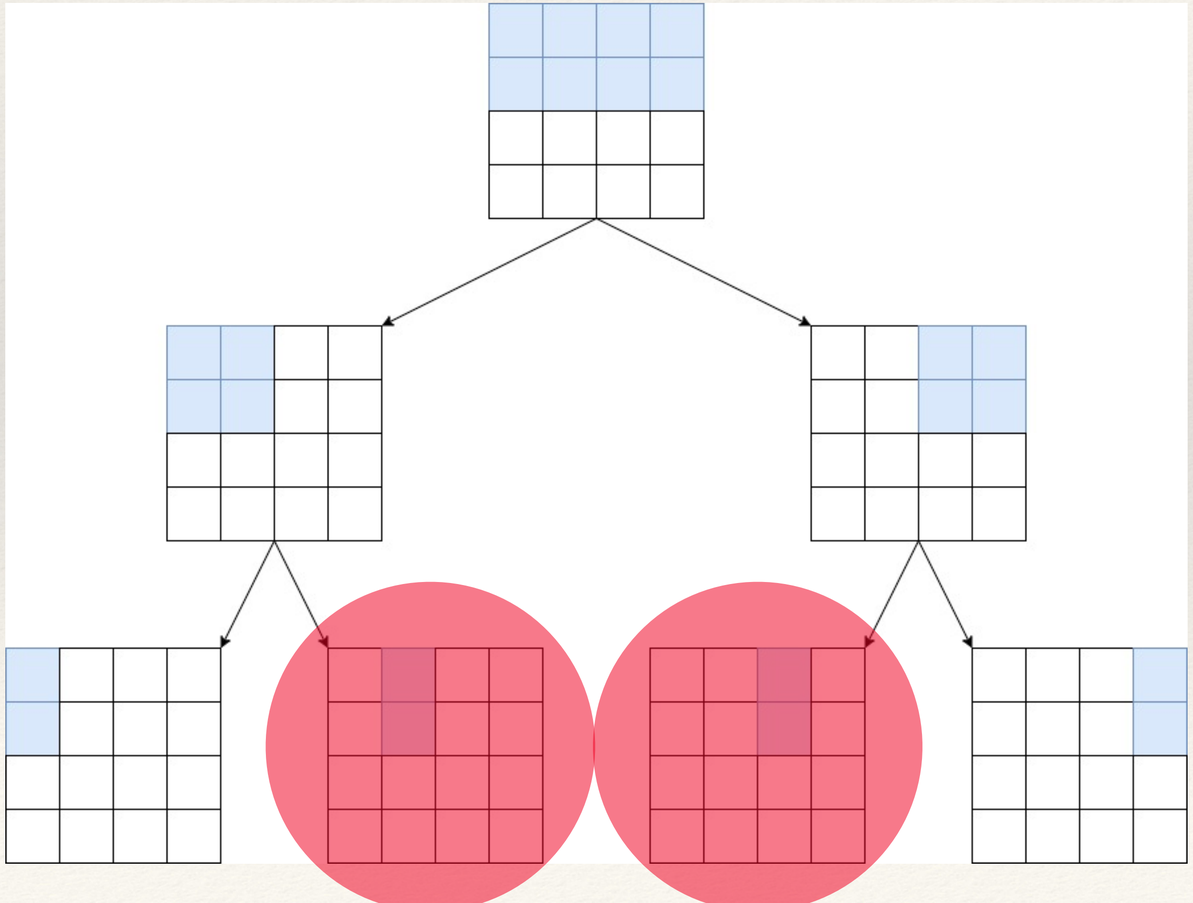
node2::query

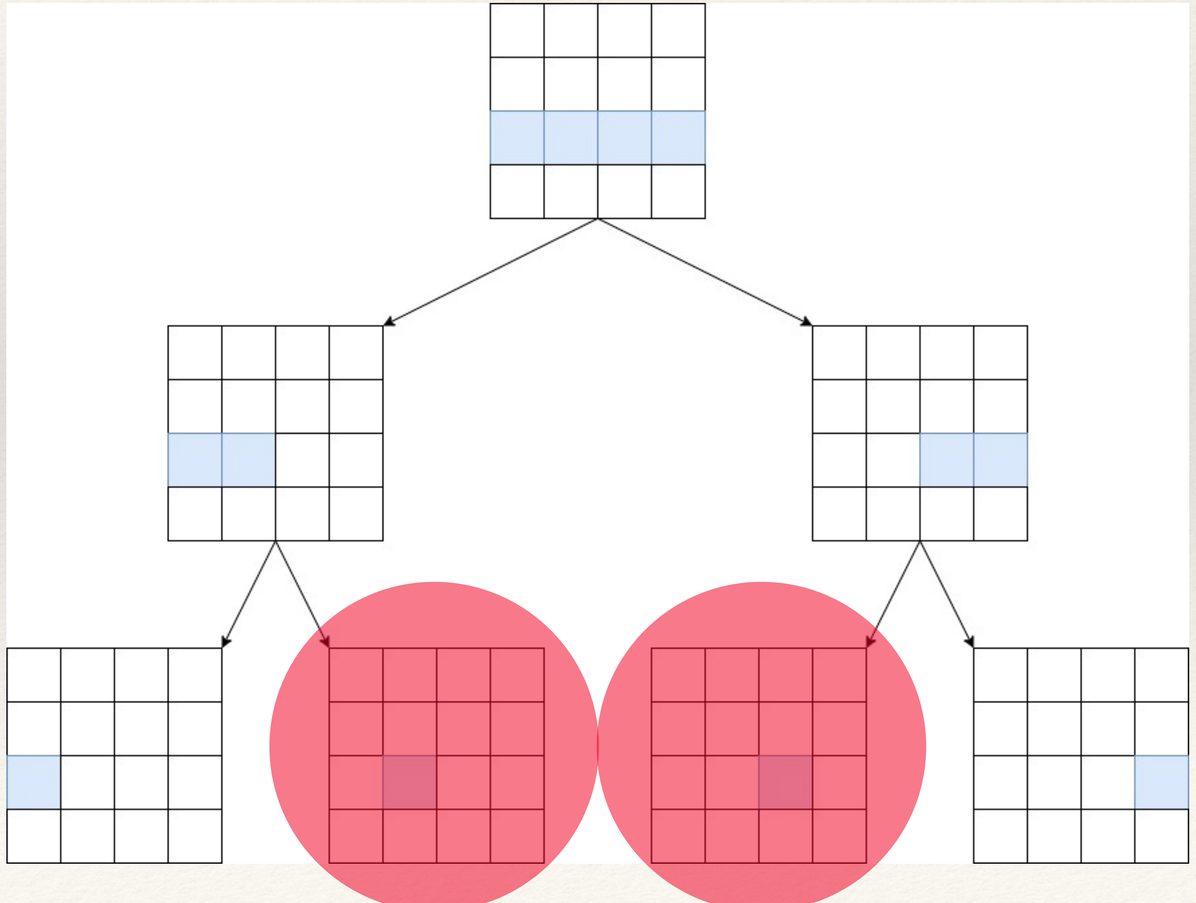
- 那麼我們會取用樹上的這些節點



node2::query

- 然後把那些節點展開後會取用的點





node2::query

- 所以 query 的部分長這樣

```
int node2::query ( int u, int d, int nowU, int nowD, int l, int r, int n ) {
    if ( u <= nowU && nowD <= d )
        return seg[n].query ( l, r, 1, len, 1 );
    int mid = ( nowU + nowD ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
    if ( d <= mid )
        return query ( u, d, nowU, mid, l, r, leftSon );
    if ( mid < u )
        return query ( u, d, mid + 1, nowD, l, r, rightSon );

    return max ( query ( u, d, nowU, mid, l, r, leftSon ), query ( u, d, mid + 1, nowD, l, r, rightSon ) );
}
```

二維線段樹的複雜度分析

- 每一個節點都是一棵線段樹，所以
  - 單次 update 的複雜度為 $\rightarrow O(logMlogN + M)$
  - 單次 query 的複雜度為 $\rightarrow O(logMlogN)$

更高維度的線段樹

- 在實際演示過我們發現，二維線段樹跟一維線段樹幾乎一樣
  - 二維的單點修改是改成呼叫一維的單點修改
  - 二維的區間查詢是呼叫一維的區間查詢
- 實際上的難點是 merge 兩顆同維度的樹時需要想清楚

二維線段樹上的懶惰標記

- 在二維線段樹上的懶惰標記只能選擇一個維度來紀錄
- 解決方案以下分兩種來介紹
  - 永久化標記
  - 四分樹

永久化標記

- 永久化標記的意思就是不把懶惰標記往下推
- 查詢時經過該節點才將該點紀錄的資訊加在答案中
- 並不是所有的懶惰標記都可以永久化

四分樹

- 四分樹算是另一種處理二維 RMQ 的作法
- 將平面四等分，並且像是線段樹遞迴劃分下去，三維的話就是八分樹、四維十六分...依此類推
- 不過缺點是四分樹的時間複雜度挺糟的，修改以及查詢皆為 $O(N+M)$，其中 N, M 分別是原始陣列的長寬，在動態開點的時候容易 TLE 或 MLE

\Treap/

Treap

- Treap，翻譯做樹堆，是以下兩種資料結構的複合體
  - 二元搜尋樹（Tree）
  - 二叉堆（Heap）
- 連名字都是這兩個字組成的（Tree + Heap = Treap）
- 通常只會用來處理複雜的區間操作（刪掉某區間、複製某區間...之類的）

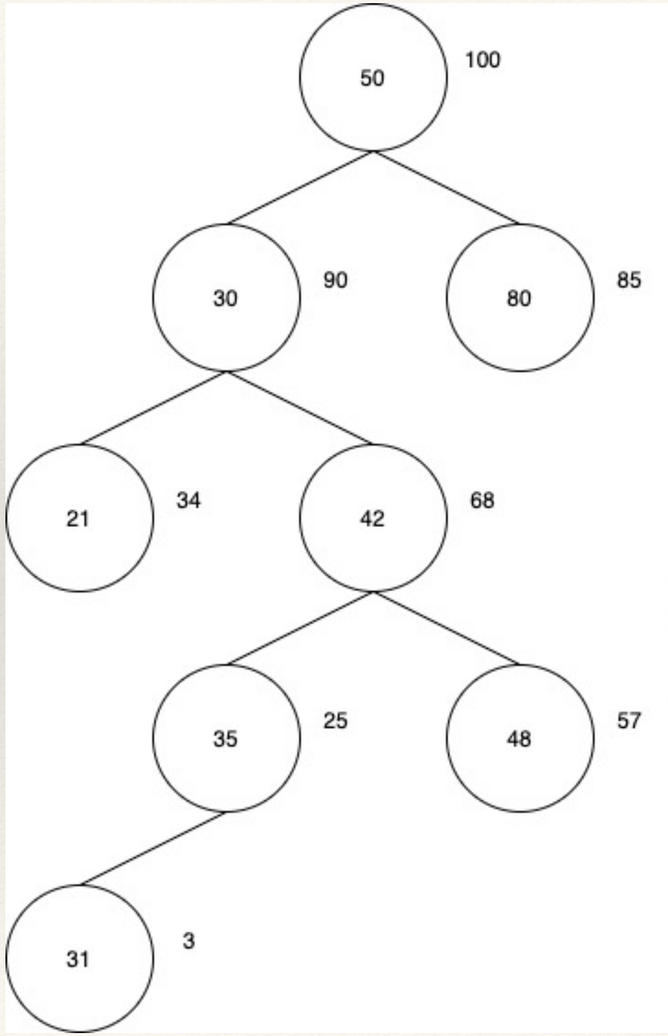
開始之前先複習一下：Tree

- 二元搜尋樹的性質（Tree）
  - 假設任意節點的左子樹不為空，則左子樹上所有節點的值都小於等於該節點的值
  - 假設任意節點的右子樹不為空，則左子樹上所有節點的值都大於等於該節點的值
  - 任意節點的子樹皆為二元搜尋樹

開始之前先複習一下：Heap

- 二叉堆的性質（Heap），這裡以最大堆作為例子
  - 假設任意節點的左子樹不為空，則左子樹上所有節點的值都大於等於該節點的值
  - 假設任意節點的右子樹不為空，則左子樹上所有節點的值都大於等於該節點的值
  - 任意節點的子樹也皆為最大堆

有點不懂？



Tree 的缺點

- 一般的二元搜尋樹不平衡的原因是容易因為針對測資變成一條長鍊，使得複雜度容易變成 $O(N)$
- 通常會為了平衡而使用旋轉以達到平衡，但旋轉不太好寫，不適合賽中使用

Treap 的改善方式

- Treap 因為使用一個隨機的值，使得 treap 的期望深度保持在 $O(logN)$等級
- 有人證明過 treap 深度不是 $O(logN)$ 的機率比 $\frac{1}{N^2}$ 還要小，換言之比 Quick Sort 還要穩定

  詳細的過程可以在 《Introduction to Algorithm》中找到

先定義一下結構

- 每個節點都需要紀錄以下資訊
  - 左子樹的指標
  - 右子樹的指標
  - key
  - priority
- 其中 key 會符合 tree 的性質，priority 會符合 heap 的性質

```
struct node {
    node *l, *r;
    int key, pri;

    node ( int _key ): l ( nullptr ), r ( nullptr ), key ( _key ), pri ( rand() ) {}
};
```

function of Treap

- Treap 也可以通過旋轉來達到平衡，不過這邊使用另一種方法來操作
- 只需要寫以下兩個 function 便能維護好 treap 的性質：
  - merge
  - split

merge

- 將兩棵 treap a, b 合併的遞迴函數，可以分成以下三個 case 來討論
  - case 1: a == NULL || b == NULL
  - case 2: a -> pri < b -> pri
  - case 3: a -> pri > b -> pri

merge: case 1

- 因為至少有一邊是 NULL（空的）， 所以只要直接回傳不是空的那邊就好了
- 假設兩邊都是空的則回傳 null

merge: case 2

- 假設 a -> pri < b -> pri，那麼會將 a 設為根節點
- 又因為按照 treap 的性質，a -> l 的 key 都小於等於 a，因此 a -> l 並不會變動
- 那 a -> r 呢？

merge: case 2

- a -> r 我們還不確定長怎樣，不過我們知道
  - a -> r 中所有 key 都大於等於 a，所以會在 merge 後的右子樹中
  - 按照 merge 的要求，b 中最小的 key 會大於等於 a 中最大的 key，所以也會在右子樹中
- 因此，a -> r = merge (a -> r, b)

merge: case 3

- case 3 則是 a -> pri > b -> pri 的狀況
- 由於推導過程類似這邊就先略過

merge: code

- case 1: 有一邊是 null 的話就直接回傳另一邊
- case 2: a 為根節點， a -> l 保留，a -> r 則為 merge (a -> r, b);
- case 3: b 為根節點， b -> r 保留，b -> l 則為 merge (a, b -> l);

```
node *merge ( node *a, node *b ) {
    if ( !a || !b )
        return a ? a : b;
    if ( a -> pri < b -> pri ) {
        a -> r = merge ( a -> r, b );
        return a;
    }
    b -> l = merge ( a, b -> l );
    return b;
}
```

merge: 分析一下複雜度吧

- 因為每跑一層 merge，a, b 的高度就會減一，直到 a 或 b 為 null 為止，所以最多跑 a + b 層，複雜度為 $O(Dep(a) + Dep(b))$
- 又一棵 treap 深度為 $O(logN)$ 的機率又很大（應該說不是 $O(logN)$ 的機率很小），所以 merge 的複雜度為 $O(logN)$

split

- 將一棵 treap 按照某個 key 值 K 切成兩棵 treap a 以及 b
- 保證 a 的中的所有 key 值都小於等於 K, 且 b 中的所有 key 值都大於 K
- 同樣分成三個 case
  - case 1: o == NULL
  - case 2: o -> key <= k
  - case 3: o -> key > k

split: case 1

- 由於 o == NULL，所以理所當然的 a 跟 b 也都是 NULL

split: case 2

- Case 2 的狀況是 o -> key < K, 可以確定 o -> l 以及 o 本身都會在 a 中, 所以把 o -> l 以及 o 都劃分在 a 中
  且 o 會是 a 的根節點
- 那剩餘的 o -> r 呢？

split: case 2

- 因為 o -> r 中的 key 值都保證大於等於 o，但是不一定都大於等於 K
- 因此我們會將他們在丟入一次 split 裡面遞迴繼續劃分，小於等於 K 的部分便成為 a -> r，其他則是 b

split: case 3

- 而 case 3 則是 o -> key > K 的狀況
- 因為也跟 split: case 2 的推導過程類似，這邊也不重複提了

split: code

- 因為 a, b 都是宣告成 reference of pointer，所以回傳值設為 void
- case 1: o == nullptr
  直接將 a, b 都設成 nullptr
- case 2: o -> key < k
  將 a 設成 o，右子樹遞迴處理
- case 3: o -> key > k
  將 b 設成 o，左子樹遞迴處理

```
void split ( node *o, node *&a, node *&b, int k ) {
    if ( !o ) {
        a = b = nullptr;
        return;
    }
    if ( o -> key < k ) {
        a = o;
        split ( o -> r, a -> r, b, k );
    }
    else {
        b = o;
        split ( o -> l, a, b -> l, k );
    }
}
```

split: 分析一下複雜度吧

- 每一次的遞迴都會把 o 的高度減少（至少減少一）
  因此最多需要跑 $Dep(o)$ 次 split
- 會一直持續到 o == nullptr 為止
- 其中我們知道一棵 treap 的高度約為 $O(logN)$
- 因此 split 的複雜度為 $O(logN)$

Treap Operator

- 既然我們已經有 merge 以及 split 了
- 那我們可以用這兩個 function 來寫一般的二元搜尋樹操作

insert

- 首先是 insert， 先按照要插入的值來分割 treap， 最後將他們 merge 起來

```
inline void insert ( node *&o, int k ) {
    node *a, *b;
    split ( o, a, b, k );
    o = merge ( merge ( a, new node ( k ) ), b );
}
```

erase

- 再來是 erase，把其中一個值刪除
- 有兩種做法
  - 先將 key = k 的位置切出來，剩下的地方 merge 起來
  - 對整棵 treap 二分搜，找到位置後將該位置用它的左右子樹 merge 起來後取代

erase: version 1

- 開三個 treap，將傳入的 treap 劃分成三塊
  - a 的所有 key 值小於 K
  - b 的所有 key 值等於 K
  - c 的所有 key 值大於 K
- 因為 b 可能不只一個元素
  為了避免一次把全部 key = K 的數字都拔掉
  所以 merge 時會包含 b -> l 以及 b -> r
  （只拔掉 b 的 root）

```
inline void erase1 ( node *&o, int k ) {
    if ( !o )
        return;
    node *a, *b, *c;
    split ( o, a, b, k );
    split ( b, b, c, k + 1 );
    o = merge ( a, c );
}
```

erase: version 2

- 對整棵 treap 跑 dfs，找到後直接把該位置刪除
- 不用擔心一次會拔超過一個點
- 速度較快（merge 以及 split 的範圍較小）

```
53 void erase2 ( node *&o, int k ) {
54     if ( !o )
55         return;
56     if ( o -> key == k ) {
57         node *t = o;
58         o = merge ( o -> l, o -> r );
59         delete t;
60         return;
61     }
62     node *&t = ( k < o -> key ? o -> l : o -> r );
63     erase2 ( t, k );
64 }
```

等等，我要 treap 幹嘛

- 如果只是要 insert 跟 erase，那用 set 或 map 就好了，沒事寫 treap 幹嘛
- 講完基本的搜尋樹操作，接著繼續看 treap 的應用

Treap 應用

- 這邊舉幾個 treap 會用到的應用
  - 動態查詢元素名次
  - 動態查詢整體第 K 大
  - 對於序列切開再剪接（ex. 把 abcde 切成 abc 以及 de， 然後接成 deabc）
  - 區間反轉、區間修改... etc.

動態查詢整元素名次

- 沒錯，這東西可以用 treap 實作
- 只要能查詢名次的資料結構都可以被稱為名次樹
- 查詢方法很簡單，在切出符合的 treap 後數元素個數就好了
- ...聽起來會 TLE

size

- 我們可以先維護 treap 下面的子樹大小
  這樣便可以在操作的同時維護 treap 大小了
- 先加一個用於向上更新的 function
- 個人習慣寫在 struct 中
  當然也可以寫在外面

```
struct node {
    node *l, *r;
    int key, pri, sz;

    node ( int _key ): l ( nullptr ), r ( nullptr ), key ( _key ), pri ( rand() ), sz ( 1 ) {}

    inline void up ( void ) {
        sz = 1;
        if ( l )
            sz += l -> sz;
        if ( r )
            sz += r -> sz;
    }
};
```

merge

- 因為多了東西要維護
  所以全部的 function 都需要做一點修改
- 在 return 前 up() 一下

```
node *merge ( node *a, node *b ) {
    if ( !a || !b )
        return a ? a : b;
    if ( a -> pri < b -> pri ) {
        a -> r = merge ( a -> r, b );
        a -> up();
        return a;
    }
    b -> l = merge ( a, b -> l );
    b -> up();
    return b;
}
```

split

- split 也是，做完 split 後將 o up() 一下以保證正確性

```
void split ( node *o, node *&a, node *&b, int k ) {
    if ( !o ) {
        a = b = nullptr;
        return;
    }
    if ( o -> key < k ) {
        a = o;
        split ( o -> r, a -> r, b, k );
    }
    else {
        b = o;
        split ( o -> l, a, b -> l, k );
    }
    o -> up();
}
```

erase: version 1

- 第一種版本的 erase 不用做修改，因為只有跑 split 以及 merge 因此在執行完 erase1 的時候就已經都是更新完的狀態了

erase: version 2

- 因為第二種版本的 erase 在對整棵 treap 做二分搜的時候並不會所有節點都跑 merge 以及 split，所以可能會有錯因此在經過的節點都跑一次 o -> up()

```
void erase2 ( node *&o, int k ) {
    if ( !o )
        return;
    if ( o -> key == k ) {
        node *t = o;
        o = merge ( o -> l, o -> r );
        delete t;
        return;
    }
    node *&t = ( k < o -> key ? o -> l : o -> r );
    erase2 ( t, k );
    if ( t )
        o -> up();
}
```

Rank

- 所以現在我們可以用 split 切出小於 K 的部分計算完成之後再把 treap 黏回去

```
inline int Rand ( node *o, int k ) {
    node *a, *b;
    split ( o, a, b, k );
    int res = ( a ? a -> sz : 0 );
    o = merge ( a, b );
    return res;
}
```

整體動態第 K 小

- 因為剛才的 split 都是按照 key 去切，所以好像沒辦法切出剛好 K 個元素出來
- 這時候我們可以改用 sz 去做 split，而不是 key

Size

- 為了方面快速查詢某 treap 的 size
  且同時避免該 treap 是 null 的可能性
  這邊寫一個 Size 以方面查詢

```
inline int Size ( node *o ) {
    return o ? o -> sz : 0;
}
```

split2

- 為了查詢第 K 大，我們寫另一個 split2
- 分割出來的結果，我們希望 a -> sz == k，也就是說 a 剛好是由前 k 個元素組成 b 則是剩下來的那些元素

Case of split2

- 跟原版的 split 一樣，也可以分成三個 case
  - case 1: o = NULL
  - case 2: o -> l -> sz < k
  - case 3: o -> l -> sz >= k

split2: case 1

- 一樣將 a, b 接設為 nullptr，然後 return

split2: case 2

- 假設現在 o -> l -> sz < k，代表我們可以把 o 以及 o -> l 全部劃分到 a 底下
- 因此 a = o 本身，a -> l = o -> l
- 那剩下的右子樹呢？

split2: case 2

- 想一下，既然前面已經切出 o -> l -> sz 個元素了
  代表我們只需要在最後的 o -> r 中，把剩下的部分切出來，劃分給 a -> r
  剩下的通通丟給 b 就完成了
- 所以往右遞迴一跑一次 split2 便可把剩下的部分做完

split2: case 3

- case 3 跟 case 2 的想法很類似，我們這邊就跳——

split2: case 3

- 現在假設 o -> l -> sz 大於 k，代表不能全部都劃分在 a 底下
- 所以 o -> r 中的元素一定不會被劃到 a 底下，可以直接分配給 b
- 因此 b = o 本身，b -> r = o -> r
- 而 b -> l 則吃 o -> l 切完的渣渣

split2: code

- case 1: o == NULL
  直接將 a, b 皆設為 nullptr，然後 return
- case 2: o -> l -> sz < k
  將左子樹分配給 a 之後，往右遞迴
- case 3: o -> l -> sz >= k
  將右子樹分配給 b 之後，往左遞迴

```
void split2 ( node *o, node *&a, node *&b, int k ) {
    if ( !o ) {
        a = b = nullptr;
        return;
    }

    if ( Size ( o -> l ) < k ) {
        a = o;
        split2 ( o -> r, a -> r, b, k - Size ( o -> l ) - 1 );
    }
    else {
        b = o;
        split2 ( o -> l, a, b -> l, k );
    }

    o -> up();
}
```

kth

- 先將前 K 個元素丟給 a，剩下丟給 c
- 將 a 的前 K - 1 個元素丟給 a，最後一個給 b
- 得到第 K 小的元素
- 最後記得 merge 把它們黏回去

```
inline int kth ( node *o, int k ) {
    node *a, *b, *c;
    split2 ( o, a, c, k );
    split2 ( a, a, b, k - 1 );
    int res = b -> key;
    o = merge ( a, merge ( b, c ) );
    return res;
}
```

Treap 以及華麗的區間操作

- 區間修改，區間反轉，區間加值

區間反轉

- 區間修改以及區間查詢線段樹已經能處理了，不用大非周章寫 treap
- 所以這邊只提線段樹無法做到的區間反轉

區間反轉

- 把 treap 當作一個序列
- 想像一下，對 treap 做樹壓平（按照 dfs 順序走訪），便會是一條正常的序列

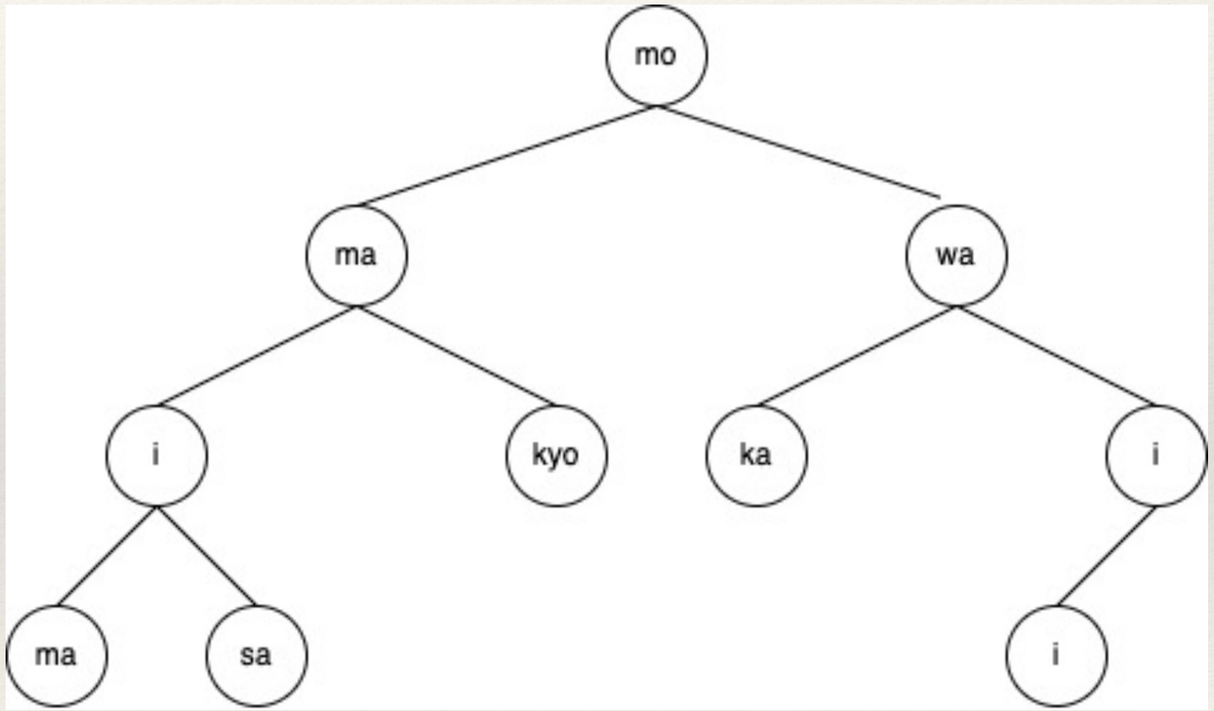
結構修改

- 在每個節點中多存一個 value，紀錄這個節點紀錄的值
- 按照中序把 value 抄下來就會是原本的序列
- kth (i) 可以想像成查詢 S[i] 的值
- 只要敲 code 的時候有注意好，key 基本上沒什麼作用了
  所以除了當作二元搜尋樹的時候，通常不會再特別紀錄 key

樹壓平/中序走訪

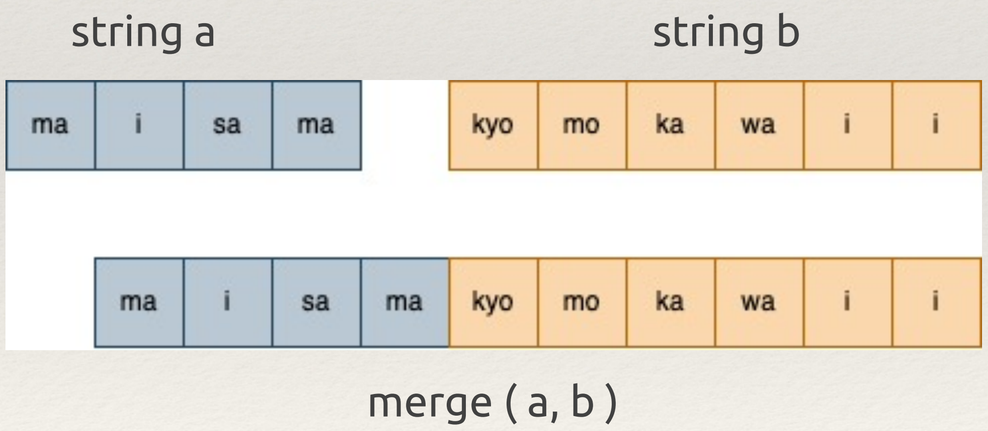
- 裡面存的值是一個平假名的羅馬拼音
- 按照中序走訪

  maisamakyomokawaii
  麻衣様今日も可愛い



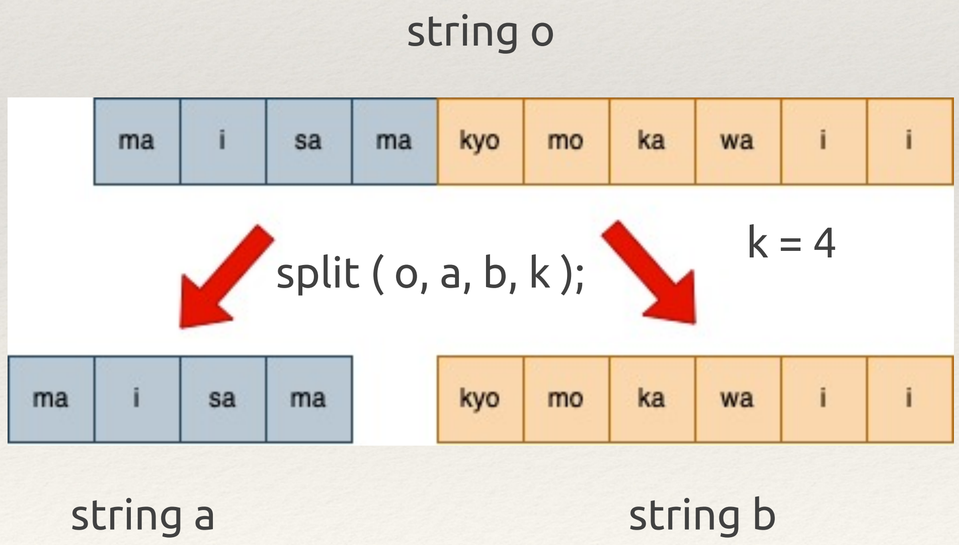
merge

- 如果把 treap 當作序列看， merge 則可以看成把兩個序列接起來



split

- 因為已經沒有 key 了，所以接下來說的 split 都是前面的 split2
- split 則可以看成是把一個序列前 K 個元素切出來



區間操作

- treap 上的區間操作相當直觀
- 基本上就是把想要操作的區塊切出來，處理完之後再貼回去
- 要打 lazy tag 也是沒問題的

帶區間翻轉的區間最大值

- 一樣也是區間最大值題目，但是多了區間翻轉
- Lazy tag + treap

Lazy Tag

- 這邊的 lazy tag 是紀錄是否翻轉過
  （對同一個區間翻轉再翻轉挺沒意義的）

```
struct node {
    node *l, *r;
    int value, pri, sz, rev, ma;

    node ( int _value );

    inline void up ( void );
    inline void down ( void );
};

node::node ( int _value ) {
    l = r = nullptr;
    value = ma = _value;
    sz = 1;
    pri = rand();
    rev = 0;
}
```

node::down

- 在需要下推 lazy tag 的時候向下推
- 以這邊為例，下推時 rev 為 1 代表需要翻轉
  會在交換完 l, r 後，確認左右子樹是否存在
  若存在則會將標記下推

```
inline void node::down ( void ) {
    if ( rev ) {
        swap ( l, r );
        if ( l )
            l -> rev ^= 1;
        if ( r )
            r -> rev ^= 1;
        rev = 0;
    }
}
```

node::up

- 基本上跟原本的相同，只是多了一個更新最大值
- 如果 lazy tag 會影響到值則需要先下推（這邊不影響，故沒下推）

```
inline void node::up ( void ) {
    sz = 1, ma = value;
    if ( l ) {
        // l -> down();
        sz += l -> sz;
        ma = max ( ma, l -> value );
    }
    if ( r ) {
        // r -> down();
        sz += r -> sz;
        ma = max ( ma, r -> value );
    }
}
```

merge/split

- merge 以及 split 也需要在對的地方加上 up 以及 down
- up 在前面已經提到該放哪了，重點是 down
- down 的重點在於必須要在 treap 結構被改變前執行

merge

```
52 node *merge ( node *a, node *b ) {
53     if ( !a || !b )
54         return a ? a : b;
55     if ( a -> pri < b -> pri ) {
56         a -> down();
57         a -> r = merge ( a -> r, b );
58         a -> up();
59         return a;
60     }
61     b -> down();
62     b -> l = merge ( a, b -> l );
63     b -> up();
64     return b;
65 }
```

split

```
void split ( node *o, node *&a, node *&b, int k ) {
    if ( !o ) {
        a = b = nullptr;
        return;
    }

    o -> down();

    if ( Size ( o -> l ) < k ) {
        a = o;
        split ( o -> r, a -> r, b, k - Size ( o -> l ) - 1 );
        a -> up();
    }
    else {
        b = o;
        split ( o -> l, a, b -> l, k );
        b -> up();
    }

    o -> up();
}
```

reverse

- Emmmm...就真的很簡單
- 把要操作的位置切出來
  打個 tag
  merge 黏回去

```
inline void reverse ( node *&o, int L, int R ) {
    if ( L > R )
        swap ( L, R );
    node *l, *r, *mid, *swp;
    split ( o, swp, r, R );
    split ( swp, l, mid, L - 1 );
    if ( mid )
        mid -> rev ^= 1;
    o = merge ( merge ( l, mid ), r );
}
```

query

- query 也是很簡單暴力的把範圍減下來
  抄完數字之後貼回去
- 而 107 行的標記下推很重要，因為有時
  候標記還沒下推會影響到答案
  雖然這題不影響啦

```
100 inline int query ( node *&o, int L, int R ) {
101     if ( L > R )
102         swap ( L, R );
103     node *l, *r, *mid, *swp;
104     split ( o, swp, r, R );
105     split ( swp, l, mid, L - 1 );
106
107     mid -> down();
108     int res = ( mid ? mid -> ma : 0 );
109     o = merge ( merge ( l, mid ), r );
110     return res;
111 }
```

Treap 複雜度分析

- 因為將深度維護在 $O(logN)$ 左右

  因此無論是 merge 或是 split 的複雜度皆為 $O(logN)$

- 而構造一棵 treap 可以想成是做 N 次 merge，所以複雜度為 $O(NlogN)$

Treap 的優缺

- 在上面的展示中，可以發現 treap 可以做到所有線段樹的工作
  甚至有些線段樹無法處理的操作（例如區間翻轉），treap 都可以輕鬆達成
- 不過由於 treap 的常數很大，所以能用線段樹寫的東西都盡量用線段樹

\Persistent Treap/

Treap 的持久化

- 很顯然易見的，treap 是支援持久化的資料結構
- 不過有小地方需要調整

複習一下持久化

- 持久化是將有修改過的節點多複製一組出來，舊的則會沿用
- 節點數量變多，造成有大量相同的 pri 值
  使 treap 的複雜度不再優秀
- 另外持久化 treap 可以做到如 a = merge (a, a) 的神奇操作

想一下

- 兩棵 treap 在 merge 時比較的是 root 的 pri，而 pri 又是隨機產生的

  可以算一下，a -> pri < b -> pri 的機率是 $\dfrac{Size(a)}{Size(a) + Size(b)}$

- 所以只要 random 出來的數字除以 $Size(a) + Size(b)$ 的餘數小於 $Size(a)$，a 就會當成根了

  $rand() \% (Size(a) + Size(b)) < Size(a)$

注意！

- 注意這邊的機率要維護好，如果只是單純的讓比較大的 treap 當根節點或是只是把機率設定成 $\frac{1}{2}$ 都是有辦法構造測資卡掉的
- 另外由於每次 merge 都要呼叫一次 rand，算是持久化 treap 的缺點

merge

- 除了 63 行以外的地方修改成計算機率以外
  還有 65 以及 71 行，在往下遞迴前先複製一個新的節點出來

```
60 node *merge ( node *a, node *b ) {
61     if ( !a || !b )
62         return a ? a : b;
63     if ( rand() % ( Size ( a ) + Size ( b ) ) < Size ( a ) ) {
64         a -> down();
65         a = new node ( *a );
66         a -> r = merge ( a -> r, b );
67         a -> up();
68         return a;
69     }
70     b -> down();
71     b = new node ( *b );
72     b -> l = merge ( a, b -> l );
73     b -> up();
74     return b;
75 }
```

split

- 同理， split 在往下切割前都要先 new 一個新的 node 出來

```
77 void split ( node *o, node *&a, node *&b, int k ) {
78     if ( !o ) {
79         a = b = nullptr;
80         return;
81     }
82
83     o -> down();
84     o = new node ( *o );
85
86     if ( Size ( o -> l ) < k ) {
87         a = o;
88         split ( o -> r, a -> r, b, k - Size ( o -> l ) - 1 );
89         a -> up();
90     }
91     else {
92         b = o;
93         split ( o -> l, a, b -> l, k );
94         b -> up();
95     }
96
97     o -> up();
98 }
```

codes on github

- https://brian.su/r/2022-winter-ds-codes

參考資料

- 2020 Spring NCKU ACM Training Courses - Week 9
  Author: MiohitoKiri5474
- ION Camp 2017：基礎資料結構、進階資料結構

Thanks for listening