基礎線段樹

日月卦長

經典題

- 給你一個長度為 n 的陣列 a，再給你 q 個操作，操作有兩種：
- $query(ql, qr)$:
  查詢 $a_{ql} + a_{ql+1} + \cdots + a_{qr}$ 的值
- $update(p, val)$:
  將 $a_p = val$
- $1 \leq n, q \leq 10^6$

如果每個操作都暴力做
複雜度很容易會變成 $\boldsymbol{O(nq)}$

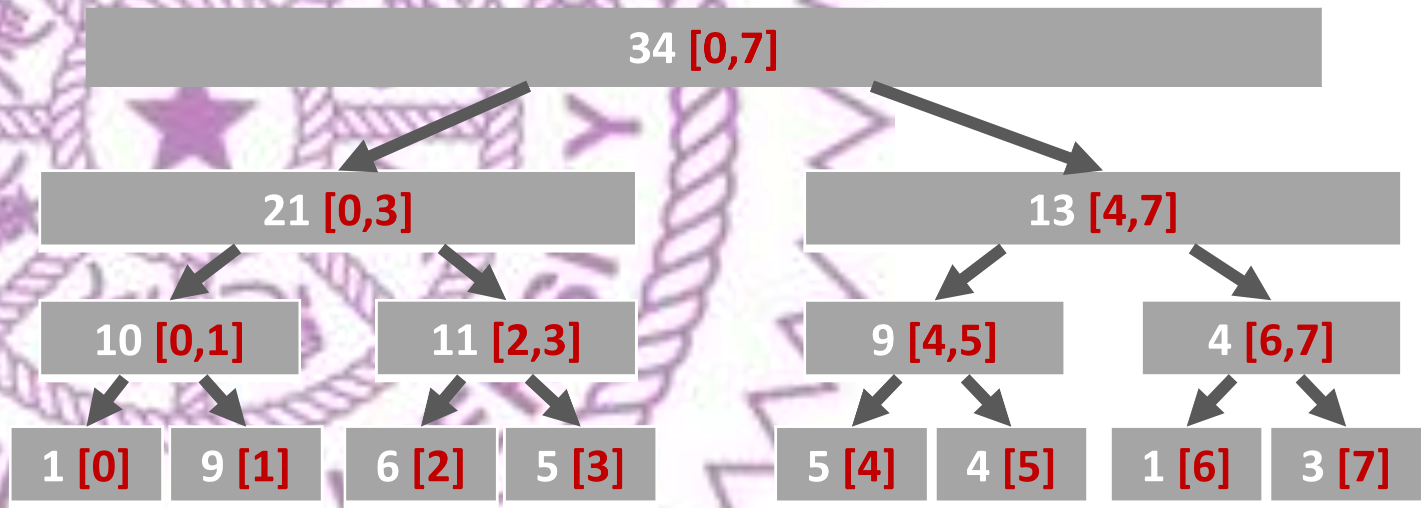
	0	1	2	3	4	5	6	7
a	1	9	6	5	5	4	1	3

線段樹的結構

以 $n = 8$ 為例

0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3

```
void build_segment_tree(L, R) {
  把[L, R] 分成一塊
  if (L == R) return;
  int m = (L + R) / 2;
  // 左右兩邊在各自遞迴分塊
  build_segment_tree(L, m);
  build_segment_tree(m + 1, R);
}
```

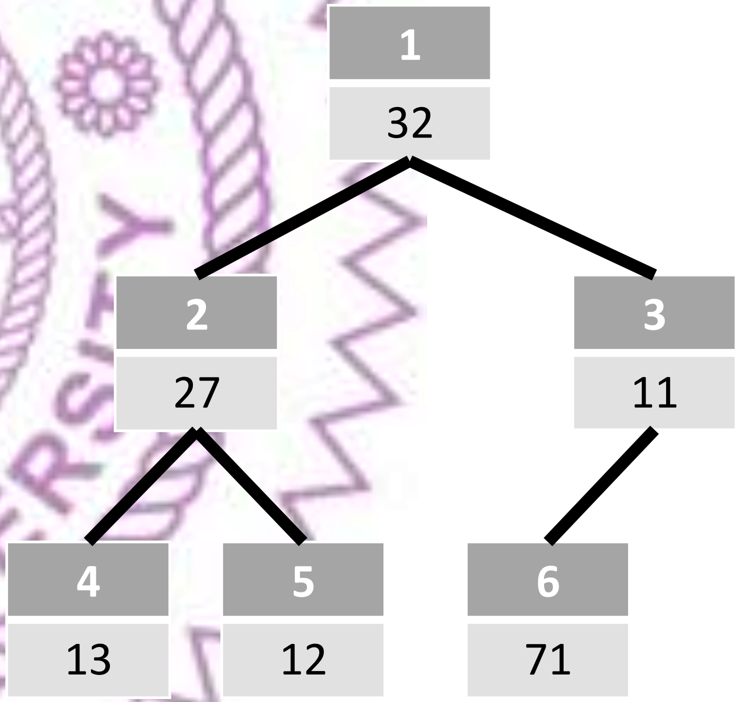


Complete Binary Tree

0	1	2	3	4	5	6
	32	27	11	13	12	71

- 一種可以用**陣列**存的二元樹
- 最後一層的點全部靠左
  其他層的點都是全滿的

- 如果某個點編號為 id：
  - Left child : $id \times 2$
  - Right child: $id \times 2 + 1$
  - Parent: $\lfloor id/2 \rfloor$

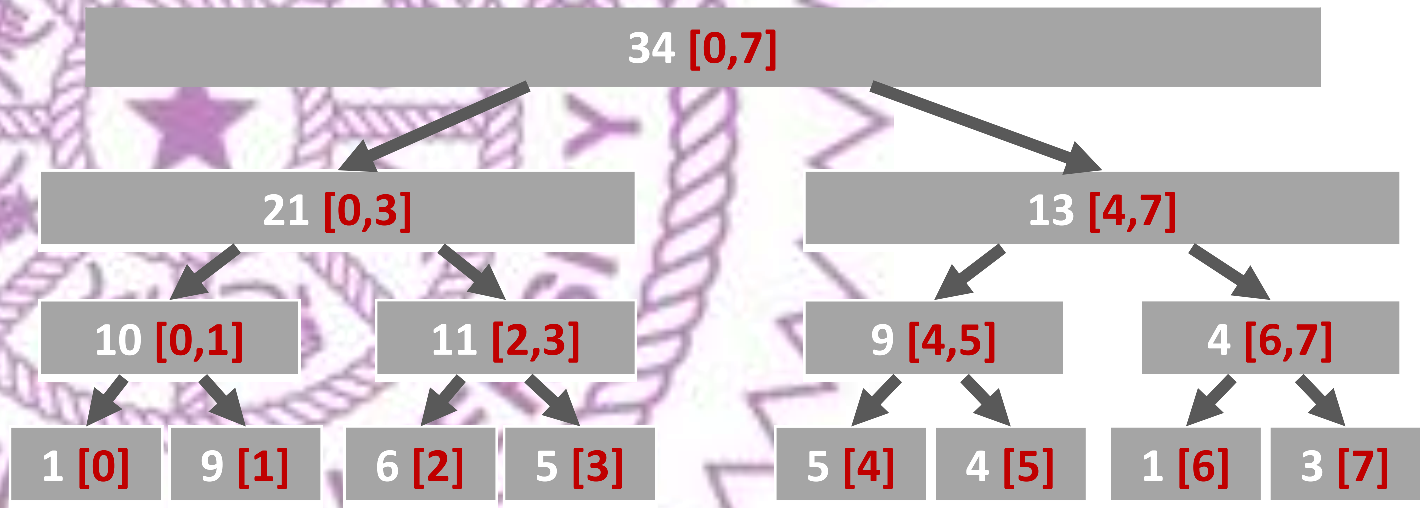


線段樹的性質

- n 個葉節點
- $2n-1$ 個節點
- 深度為 $\lceil \log_2 n \rceil + 1$
  - 取上高斯是因為 n 不一定是 2 的冪次

以 $n = 8$ 為例

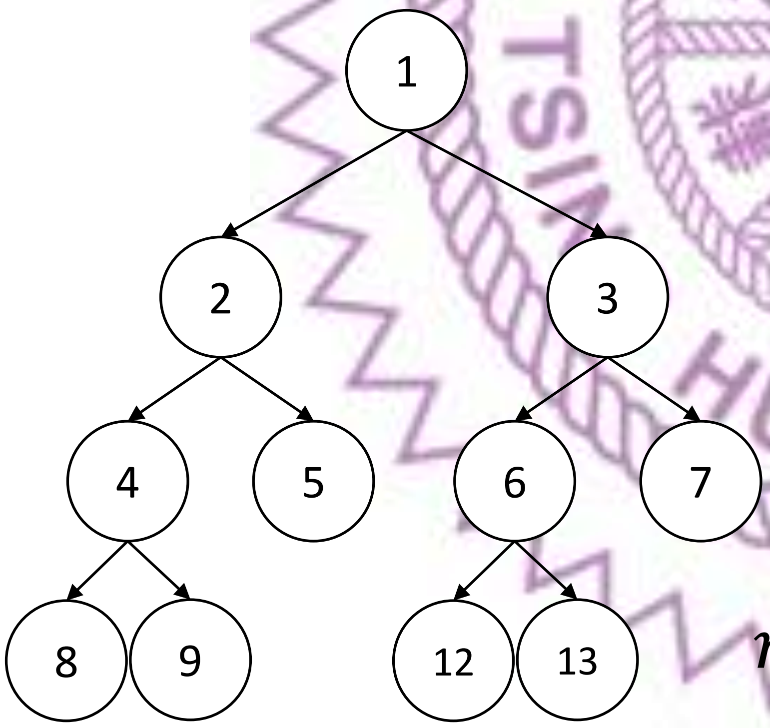
0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3



線段樹的空間

- 設 id 為用 Complete Binary Tree 陣列存線段樹時的最大編號

$$id \leq 2^{\lceil \log_2 n \rceil + 1} - 1 < 2^{\lceil \log_2 n \rceil + 2} \leq 4n$$



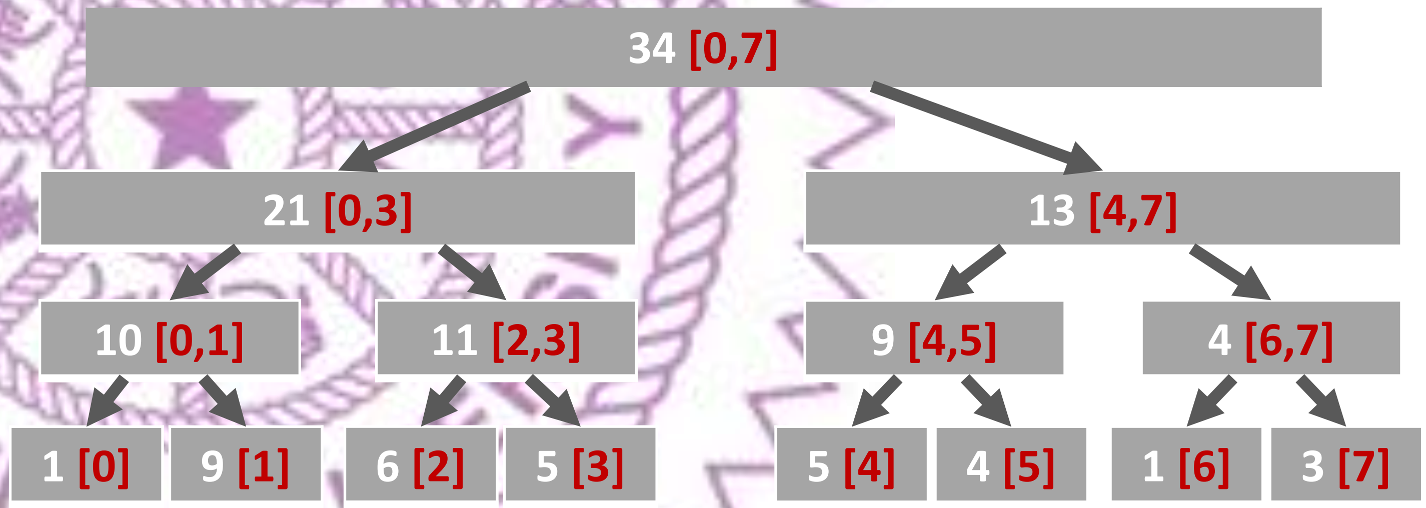
$n = 6$ 的線段樹形狀

線段樹的構造

以 $n = 8$ 為例

0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3

```
int seg[4 * MAXN];
void build(int l, int r, int id = 1)
{
  if (l == r) {
    seg[id] = a[l];
    return;
  }
  int m = (l + r) / 2;
  build(l, m, 2 * id);
  build(m + 1, r, 2 * id + 1);
  pull(id);
}
build(0, n - 1);
```



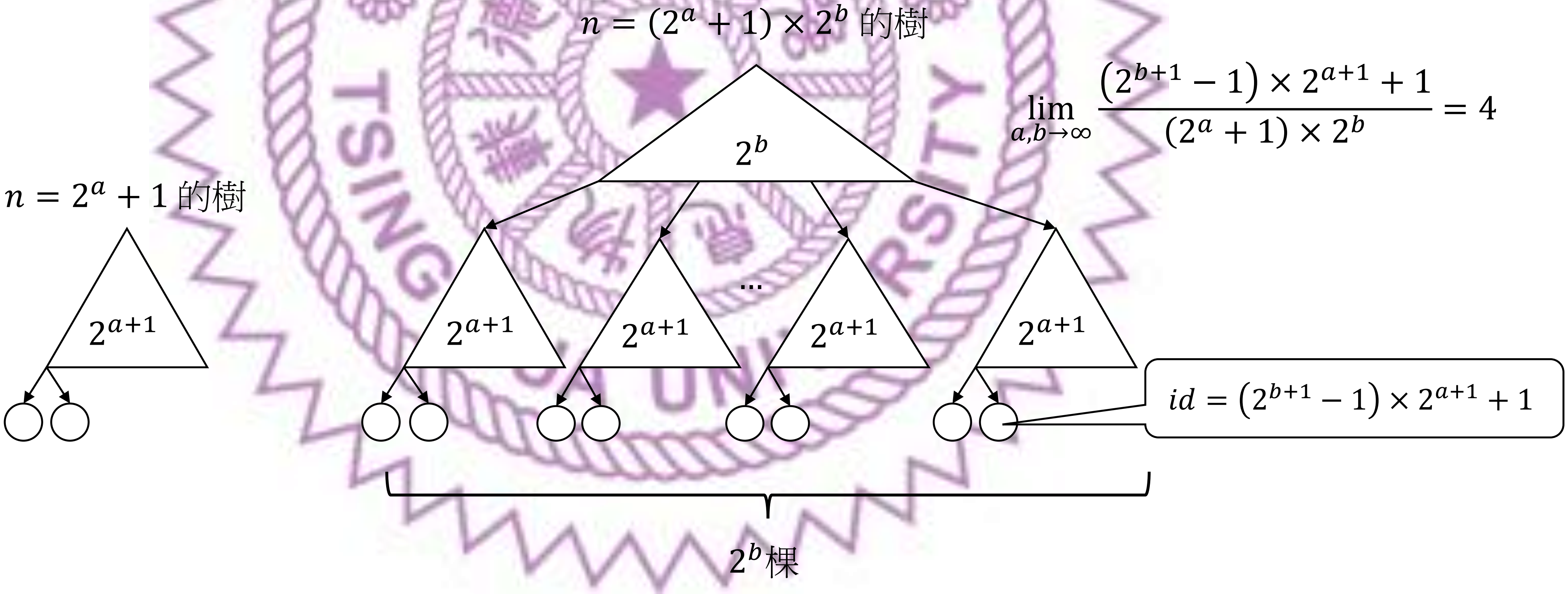

```
void pull(int id) {
  seg[id] = seg[id * 2] + seg[id * 2 + 1];
}
```

注意節點所代表的區間範圍
以及編號是動態計算的

會讓空間接近 $4n$ 的 case

- 當 $n = (2^a + 1) \times 2^b,\ a, b \geq 3$ 時
  陣列實作的線段樹使用的空間會接近 $4n$

https://ideone.com/BeYSpZ

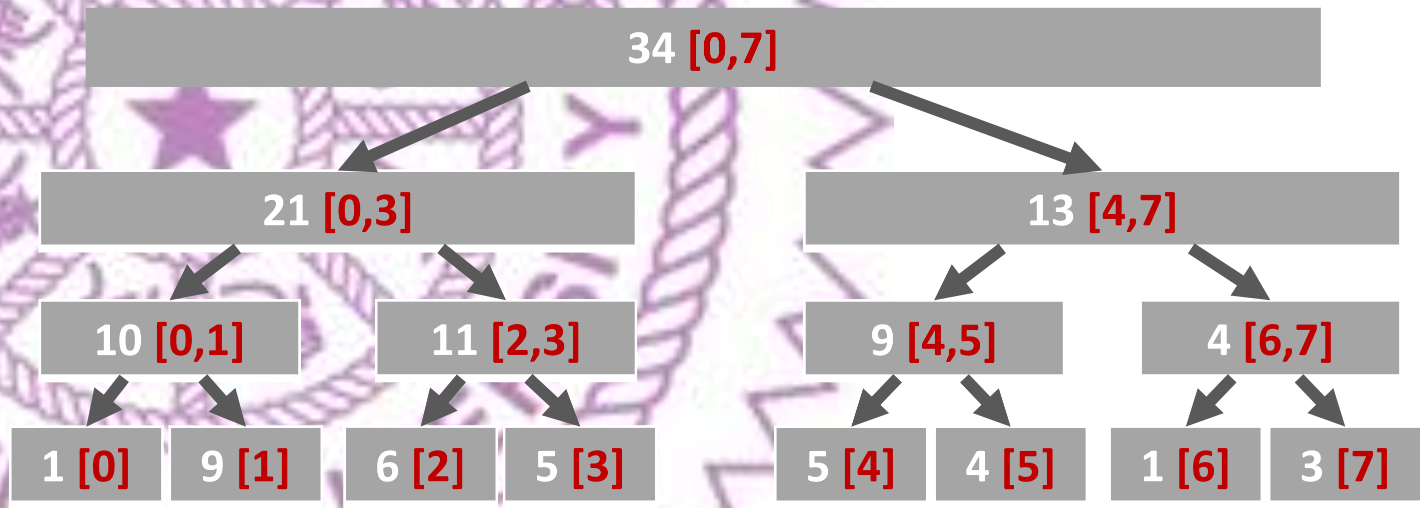


$$\lim_{a,b \to \infty} \frac{(2^{b+1} - 1) \times 2^{a+1} + 1}{(2^a + 1) \times 2^b} = 4$$

查詢區間總和

- 以區間 [2, 6] 為例
- 從 root 開始做 DFS
  可以分成 3 個 case

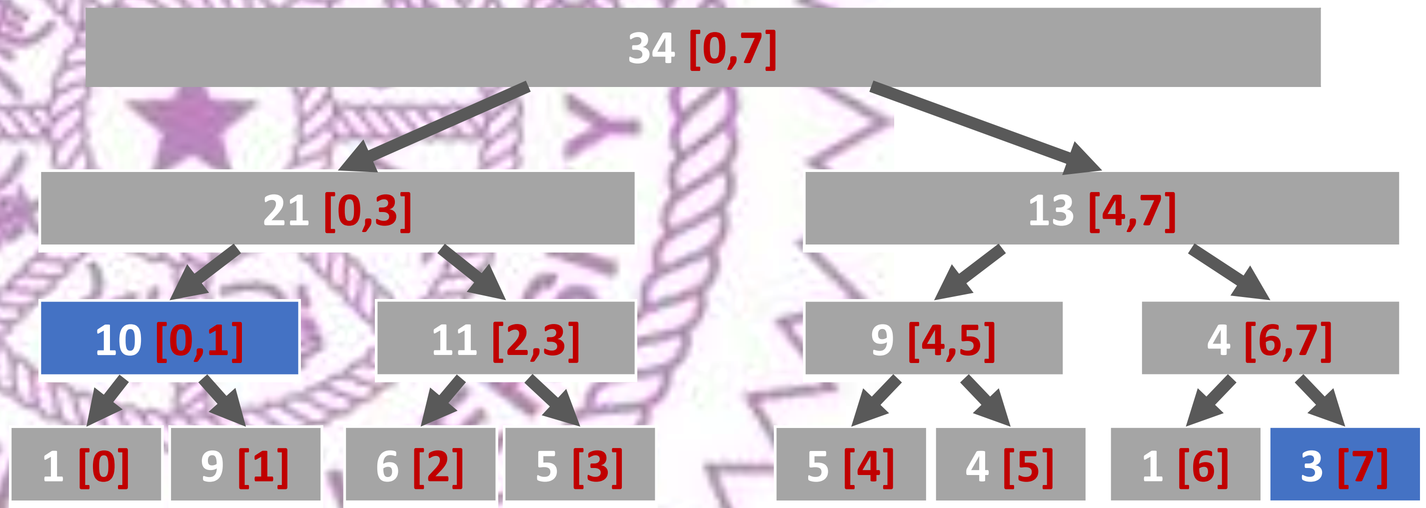
0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3



查詢區間總和

- 以區間 [2, 6] 為例
- 從 root 開始做 DFS
  可以分成 3 個 case
- Case 1: 不在範圍內
  - 直接 return 0 就行了

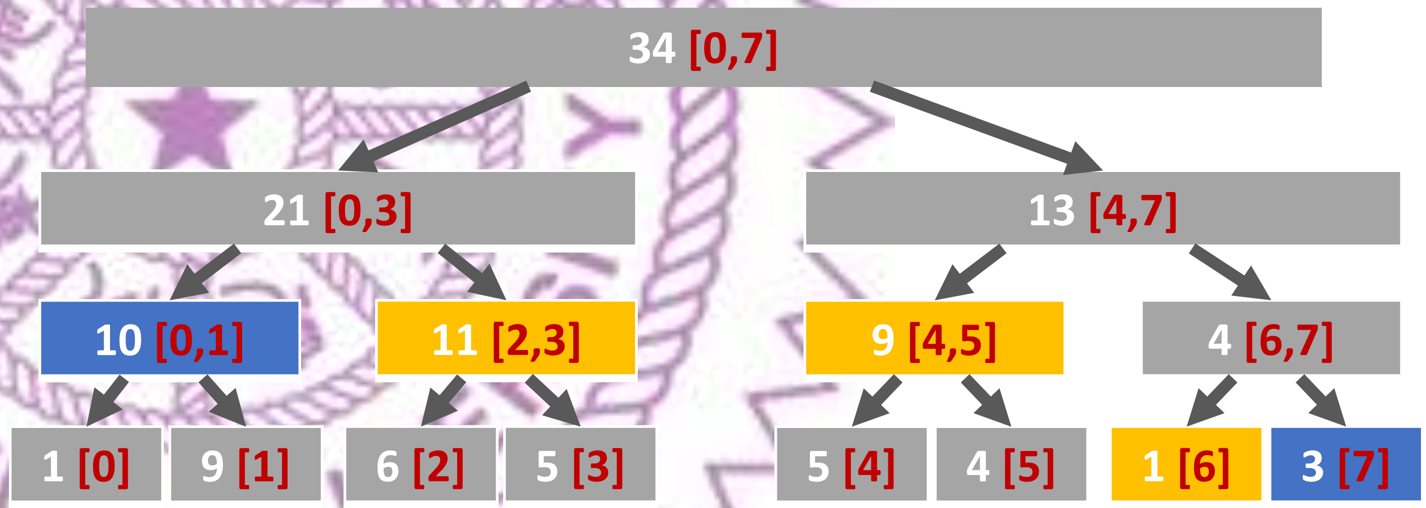
0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3



查詢區間總和

- 以區間 [2, 6] 為例
- 從 root 開始做 DFS
  可以分成 3 個 case
- Case 1: 不在範圍內
  - 直接 return 0 就行了
- Case 2: 完全位於範圍內
  - 直接 return 該點的值

0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3

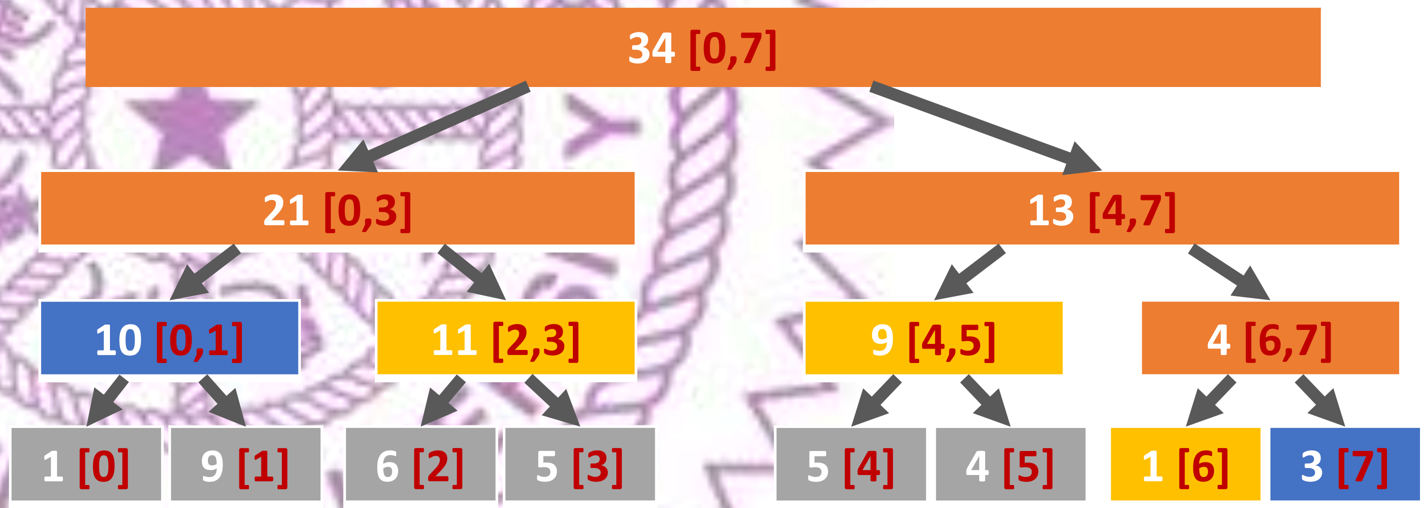


$0 + 11 + 9 + 1 + 0 = 21$

查詢區間總和

- 以區間 [2, 6] 為例
- 從 root 開始做 DFS
  可以分成 3 個 case
- Case 1: 不在範圍內
  - 直接 return 0 就行了
- Case 2: 完全位於範圍內
  - 直接 return 該點的值
- Case 3: 部分位於範圍內
  - 遞迴左右小孩

0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3



$0 + 11 + 9 + 1 + 0 = 21$

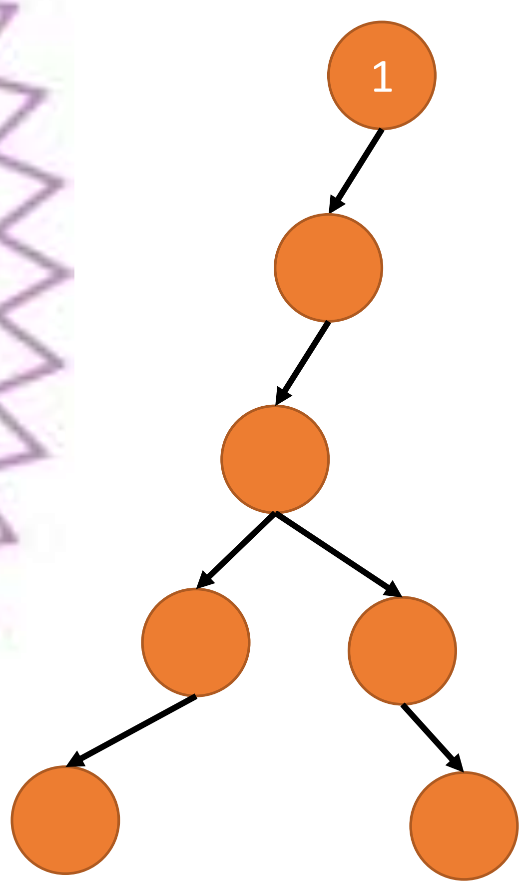
查詢區間總和

```
int query(int ql, int qr, int l, int r, int id = 1) {
  if (qr < l || r < ql) // [l,r] 不在 [ql,qr] 的範圍
    return 0;
  if (ql <= l && r <= qr) // [l,r] 被 [ql,qr] 完全包含
    return seg[id];
  int m = (l + r) / 2; // 剩下就遞迴處理
  return query(ql, qr, l, m, id * 2) + query(ql, qr, m + 1, r, id * 2 + 1);
}
query(ql, qr, 0, n - 1);
```

時間複雜度

- Case 3 節點最多有 $(\lceil\log_2 n\rceil + 1) \times 2 - 1$ 個
- Case 1, Case 2 的節點只會是 Case 3 的小孩
  所以他們最多共有 $(\lceil\log_2 n\rceil + 1) \times 2$ 個

$$\text{總時間} = \text{經過的點數}$$
$$\leq (\lceil\log_2 n\rceil + 1) \times 4 - 1 = O(\log n)$$

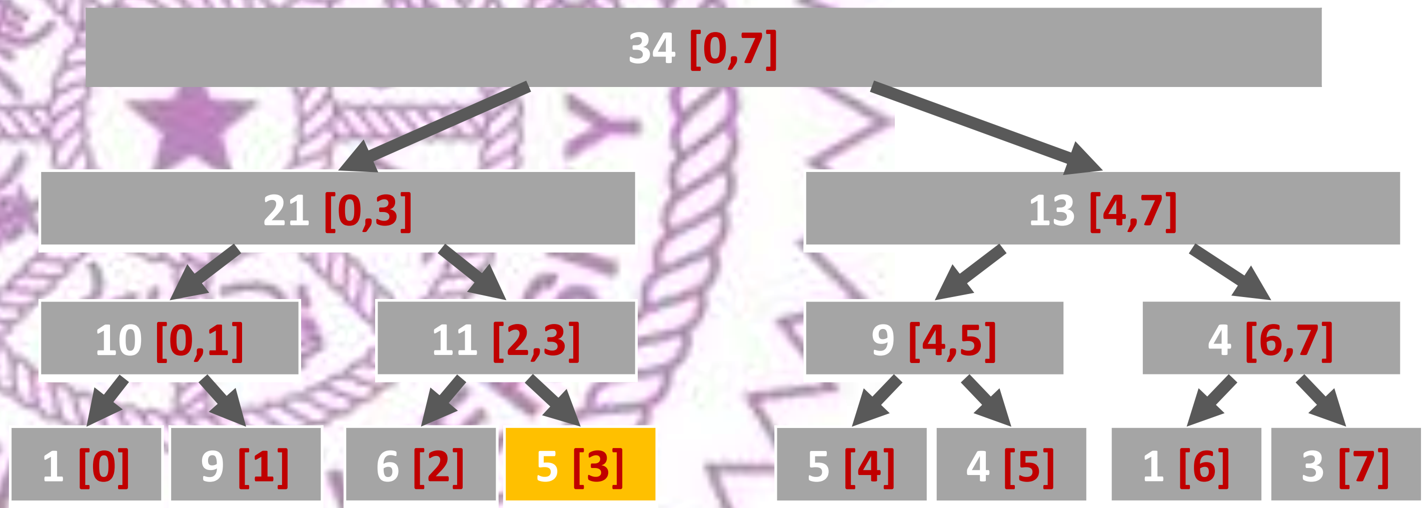


Case 3 節點的連接情況

單點修改

- 以將 a_3 改成 9 為例
- 先找到要改的點

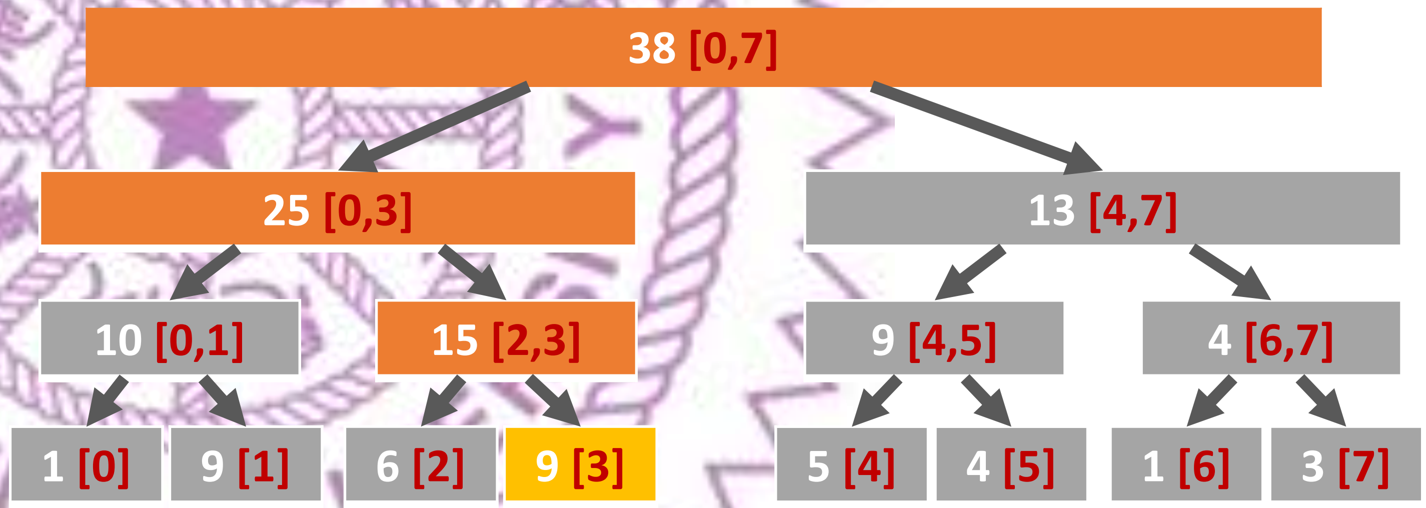
0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3



單點修改

- 以將 a_3 改成 9 為例
- 先找到要改的點
- 向上更新到 root

0	1	2	3	4	5	6	7
1	9	6	9	5	4	1	3



單點修改

```
void update(int p, int val, int l, int r, int id = 1) {
  if (p < l || r < p) // 範圍外
    return;
  if (l == r) { // 範圍內
    seg[id] = val;
    return;
  }
  // 分兩半
  int m = (l + r) / 2;
  update(p, val, l, m, id * 2);
  update(p, val, m + 1, r, id * 2 + 1);
  pull(id); // 不要忘了他，待會是重點之一
}
update(p, val, 0, n - 1);
```

時間複雜度

總時間 ＝ 經過的點數

$$= (\lceil \log_2 n \rceil + 1) = O(\log n)$$

查詢、修改總結

- 不再範圍內，不處理
- 剛好位於範圍中，直接處理
- 剩下的情況，分兩半遞迴

經典題 2

- 給你一個長度為 n 的陣列 a，再給你 q 個操作，操作有兩種：
- $query(ql, qr)$:
  查詢 $a_{ql} + a_{ql+1} + \cdots + a_{qr}$ 的值
- $update(ql, qr, val)$:
  將 $a_{ql}, a_{ql+1}, \ldots, a_{qr}$ 都加上 val
- $1 \le n, q \le 10^6$

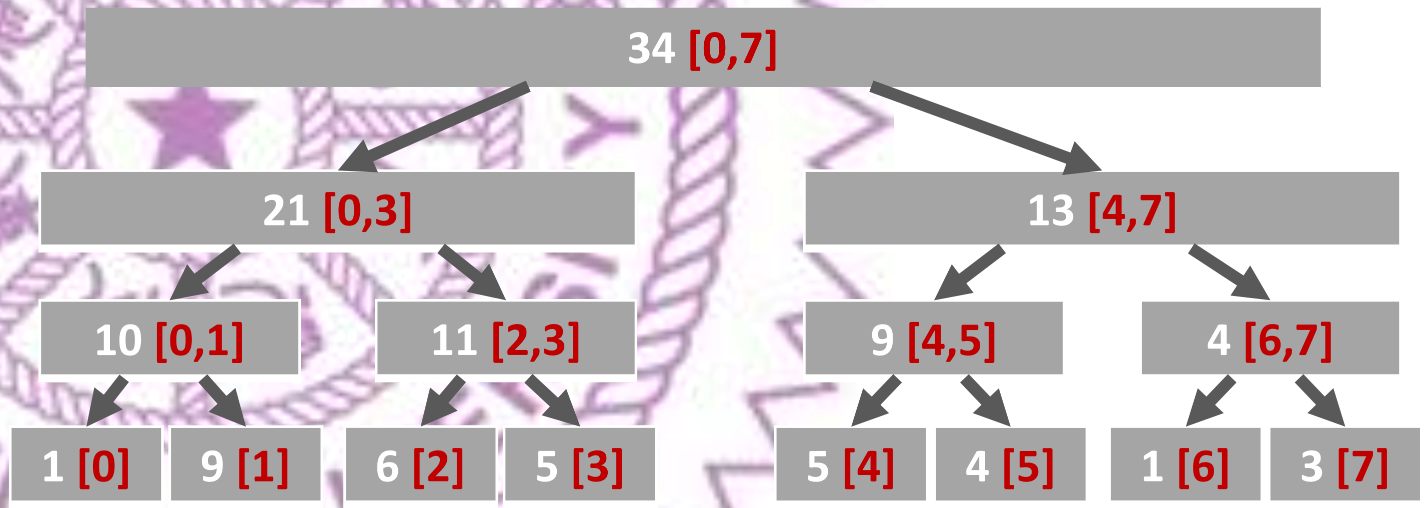
add 要怎麼做?

a	0	1	2	3	4	5	6	7
	1	9	6	5	5	4	1	3

懶惰標記

- 如果一個節點 $[l,r]$ 有設立懶惰標記的話，表示這一個區間的每一個元素都要做同樣的操作

0	1	2	3	4	5	6	7
1	9	6	5	5	4	1	3

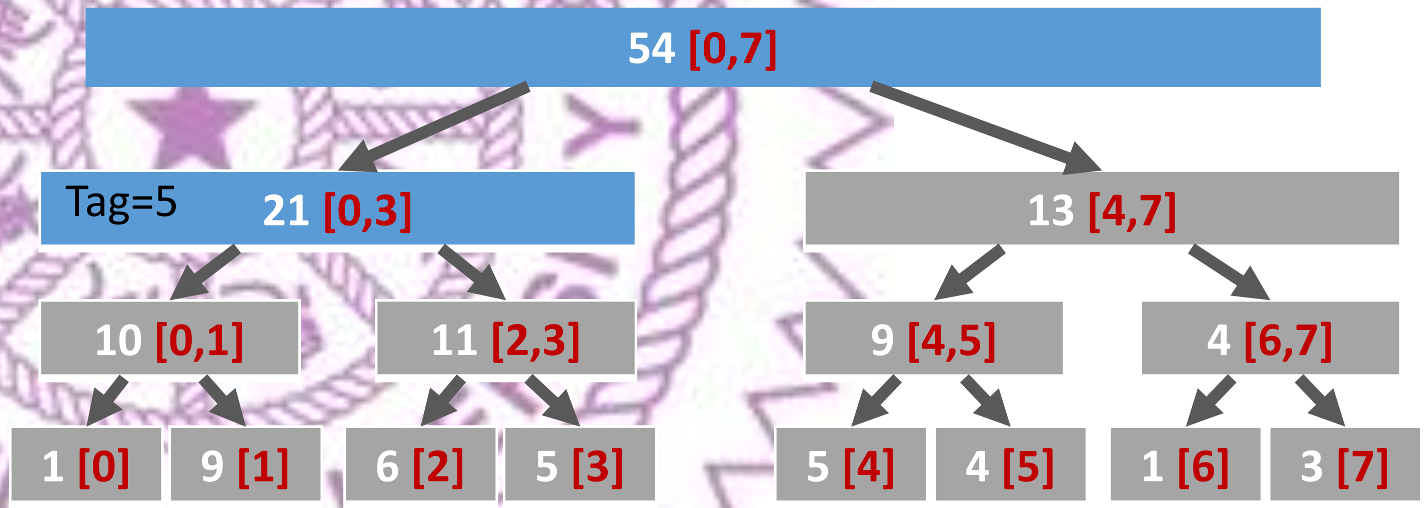


懶惰標記

- 如果一個節點 $[l,r]$ 有設立懶惰標記的話，表示這一個區間的每一個元素都要做同樣的操作

- 但實際上**還沒有做**

0	1	2	3	4	5	6	7
6	14	11	10	5	4	1	3



紀錄懶惰標記

- 為了簡化實作，我們用 get_val 函數來取得一個節點的值
- pull 也要增加一些參數

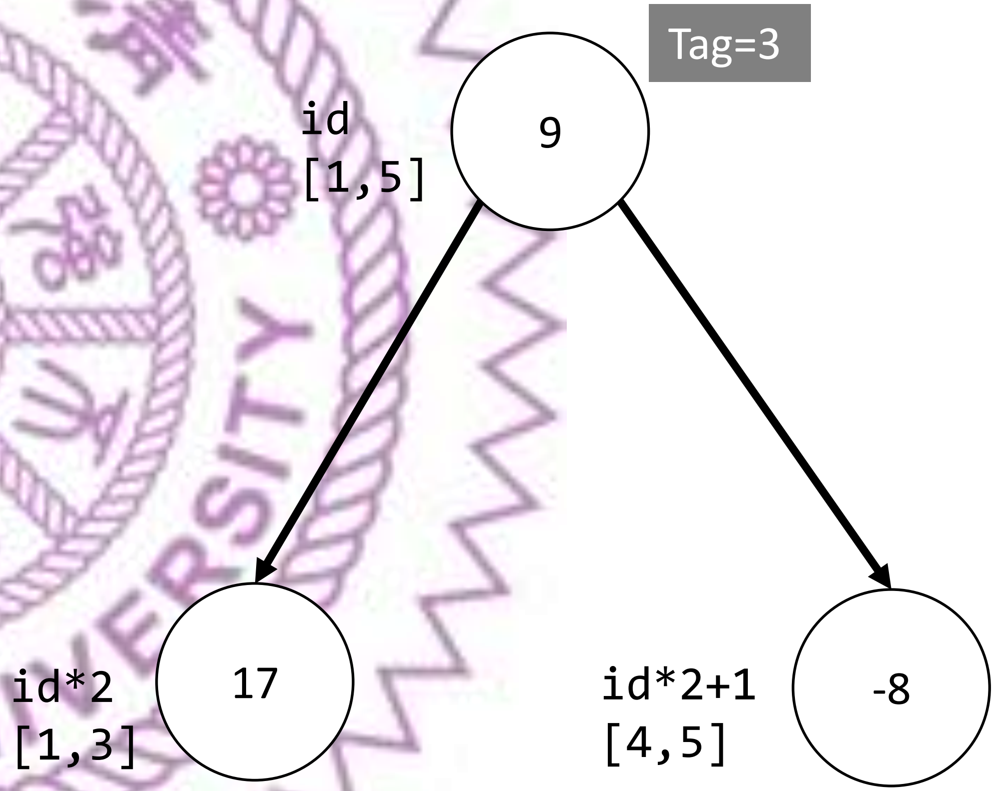
```
struct Node {
  int data, tag;
};
Node seg[4 * MAXN];
```

```
int get_val(int l, int r, int id) {
  return (r - l + 1) * seg[id].tag + seg[id].data;
}
```

```
void pull(int l, int r, int id) {
  int m = (l + r) / 2;
  seg[id].data = get_val(l, m, id * 2) + get_val(m + 1, r, id * 2 + 1);
}
```

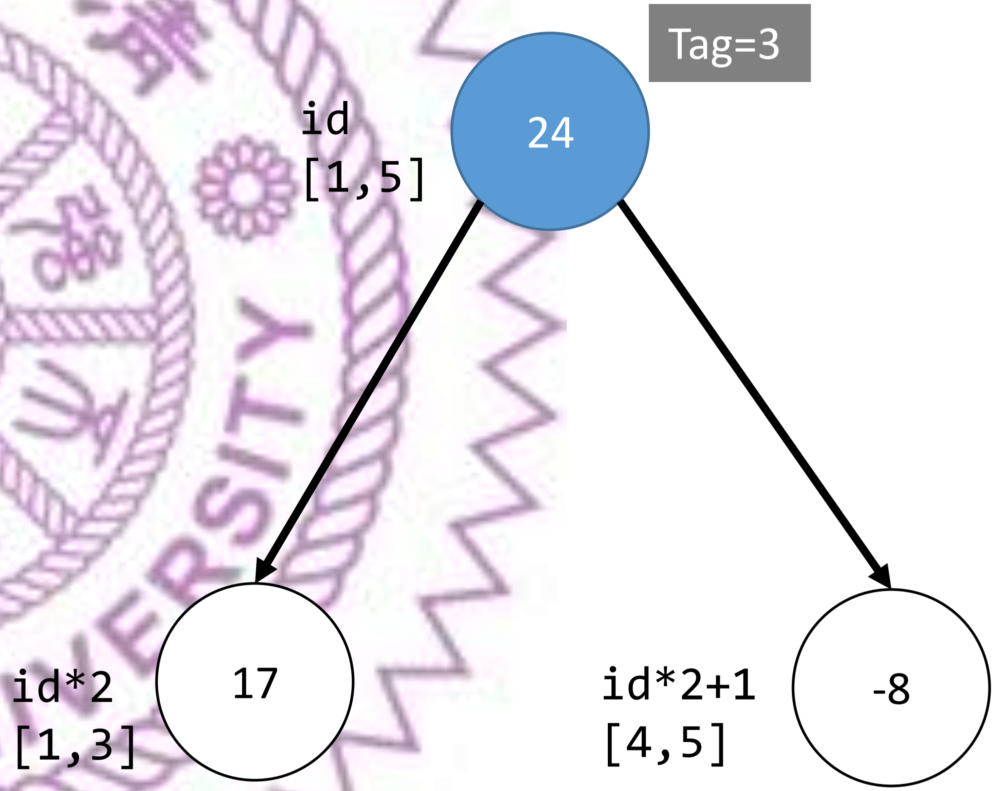
懶惰標記下推

```
void push(int l, int r, int id) {
  seg[id].data = get_val(l, r, id);
  seg[id * 2].tag += seg[id].tag;
  seg[id * 2 + 1].tag += seg[id].tag;
  seg[id].tag = 0;
}
```



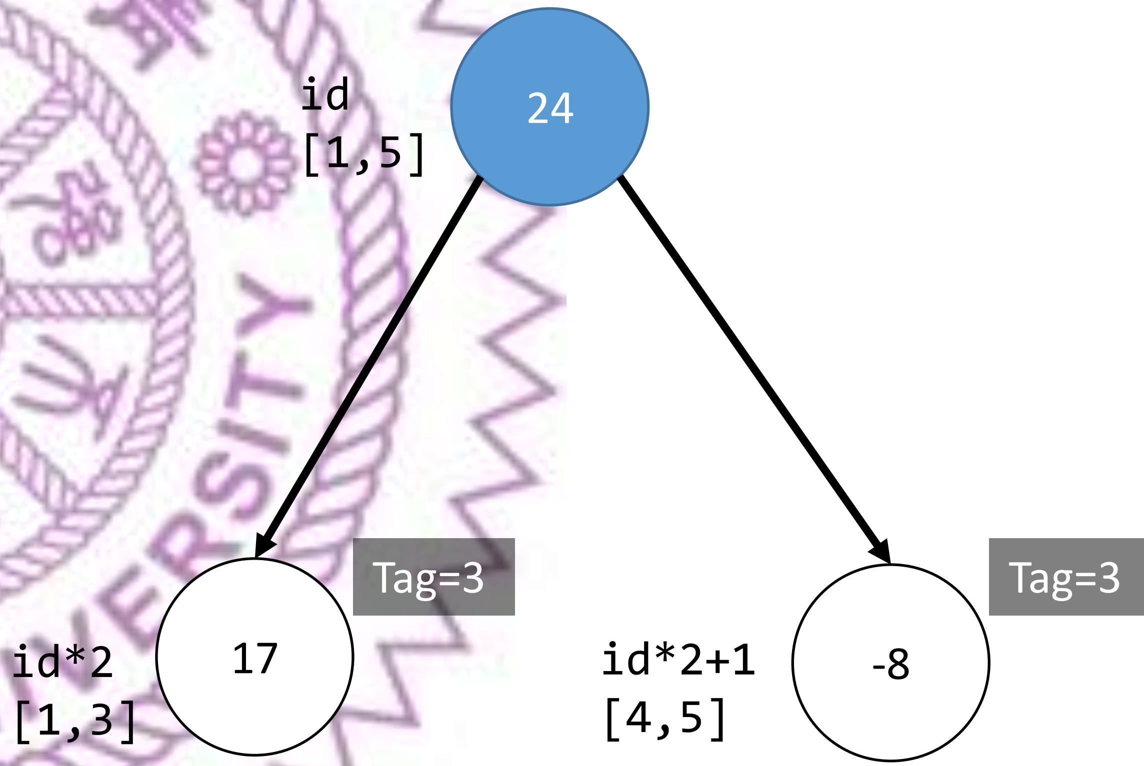
懶惰標記下推

```
void push(int l, int r, int id) {
  seg[id].data = get_val(l, r, id);
  seg[id * 2].tag += seg[id].tag;
  seg[id * 2 + 1].tag += seg[id].tag;
  seg[id].tag = 0;
}
```



懶惰標記下推

```
void push(int l, int r, int id) {
  seg[id].data = get_val(l, r, id);
  seg[id * 2].tag += seg[id].tag;
  seg[id * 2 + 1].tag += seg[id].tag;
  seg[id].tag = 0;
}
```



查詢區間總和

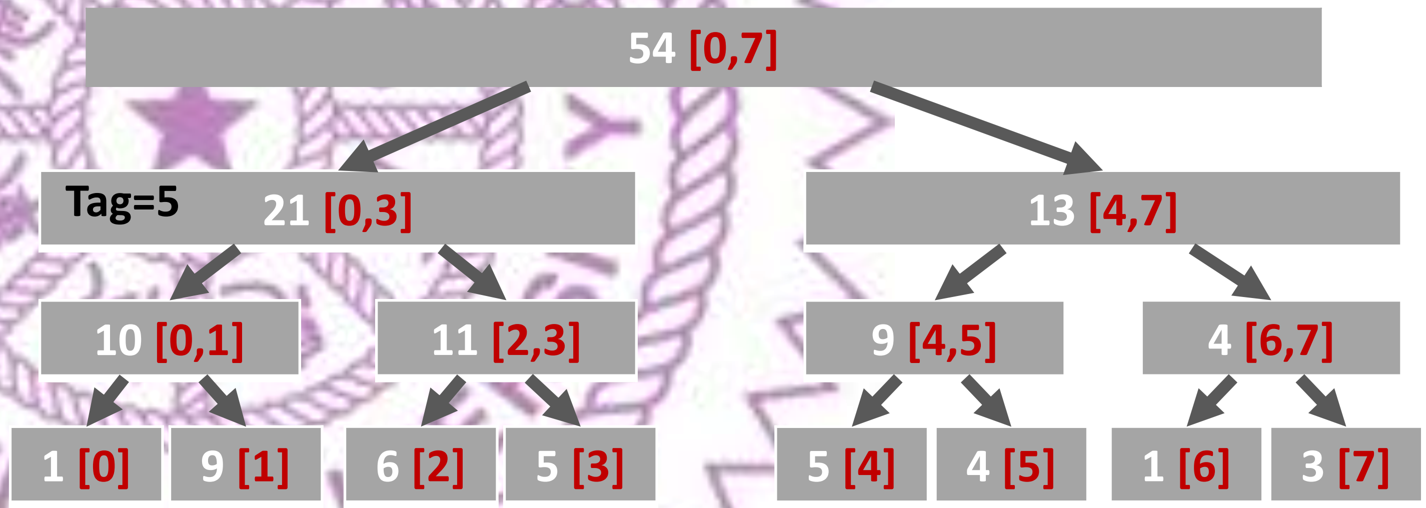
- 與沒有懶惰標記的情況差不多
- 最大的改動是分兩半遞迴前先 push

```
int query(int ql, int qr, int l, int r, int id = 1) {
  if (qr < l || r < ql)
    return 0;
  if (ql <= l && r <= qr)
    return get_val(l, r, id); // 注意計算方式
  push(l, r, id);             // 多了 push ，注意其被呼叫的位置
  int m = (l + r) / 2;
  return query(ql, qr, l, m, id * 2) + query(ql, qr, m + 1, r, id * 2 + 1);
}
query(ql, qr, 0, n - 1);
```

查詢區間總和

• 一樣以區間 [2, 6] 為例

0	1	2	3	4	5	6	7
6	14	11	10	5	4	1	3

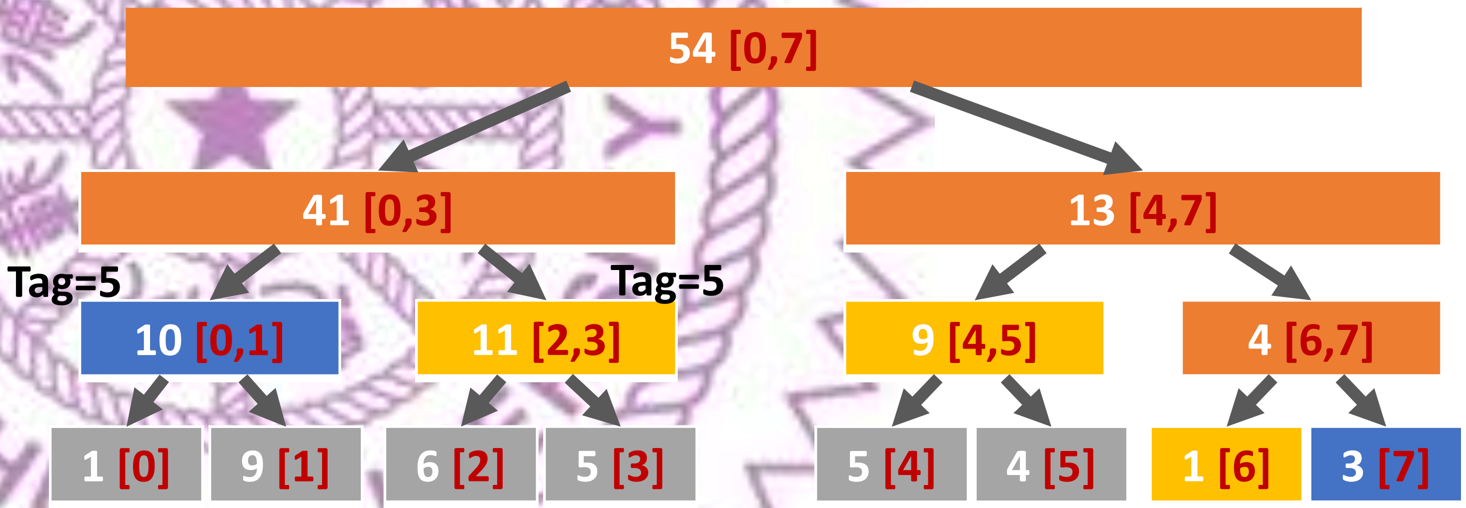


查詢區間總和

- 一樣以區間 [2, 6] 為例

0	1	2	3	4	5	6	7
6	14	11	10	5	4	1	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



- Case 3 的懶惰標記被推到 Case 1,2 的點上答案才是對的

$0 + 21 + 9 + 1 + 0 = 31$

區間加值

- 和查詢區間總和結構相同，只是改成位於範圍內修改標記
  由於值有變所以要記得 pull

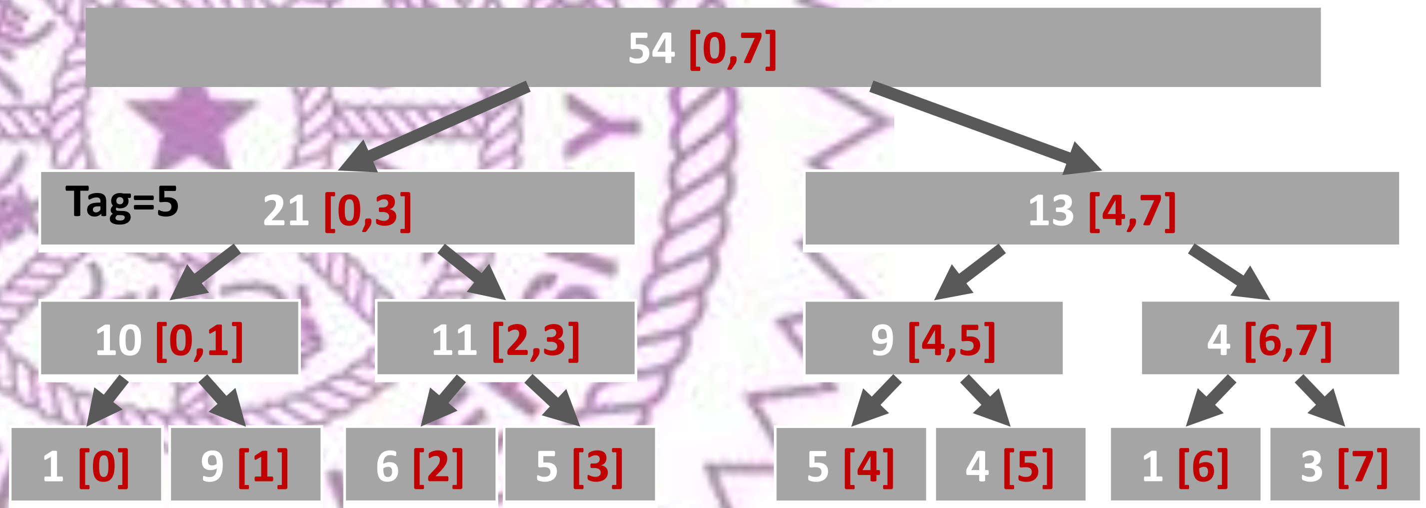
```
void update(int ql, int qr, int val, int l, int r, int id = 1) {
  if (qr < l || r < ql) // 完全不在範圍內，不做事
    return;
  if (ql <= l && r <= qr) { // 完全位於範圍內，直接改 tag
    seg[id].tag += val;
    return;
  }
  push(l, r, id); // 切兩半前 push
  int m = (l + r) / 2;
  update(ql, qr, val, l, m, id * 2);
  update(ql, qr, val, m + 1, r, id * 2 + 1);
  pull(l, r, id); // 結束後 pull
}
query(ql, qr, val, 0, n - 1);
```

區間加值

- 區間 [2, 6] 都加上 3 為例

0	1	2	3	4	5	6	7
6	14	11	10	5	4	1	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

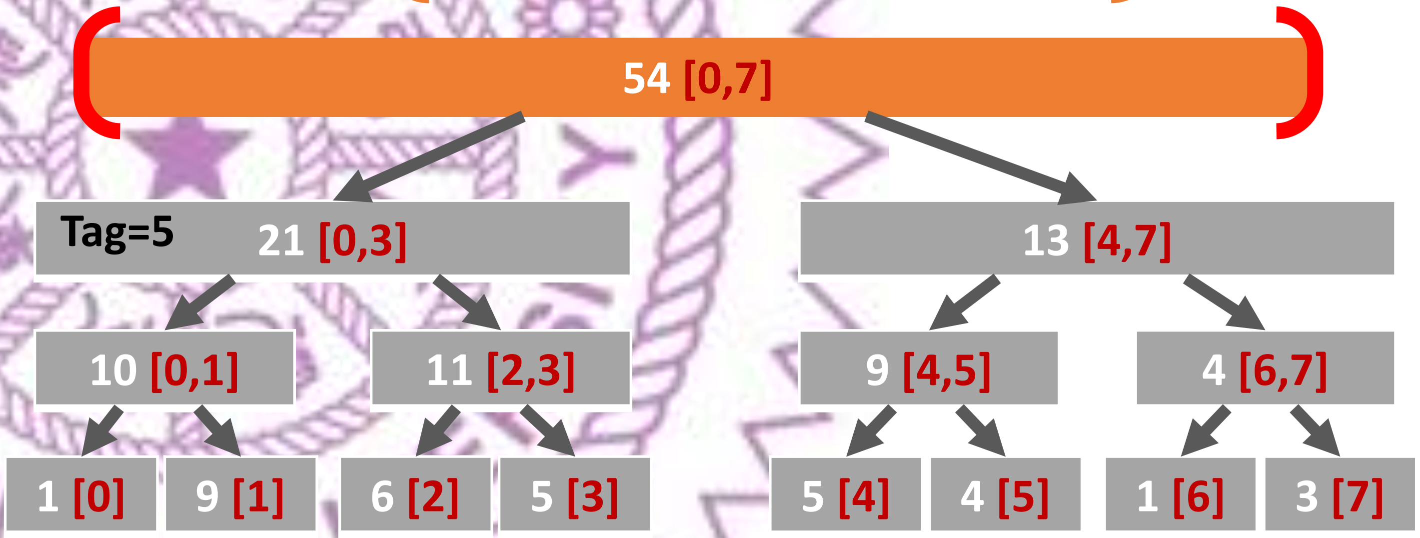


區間加值

- 區間 [2, 6] 都加上 3 為例

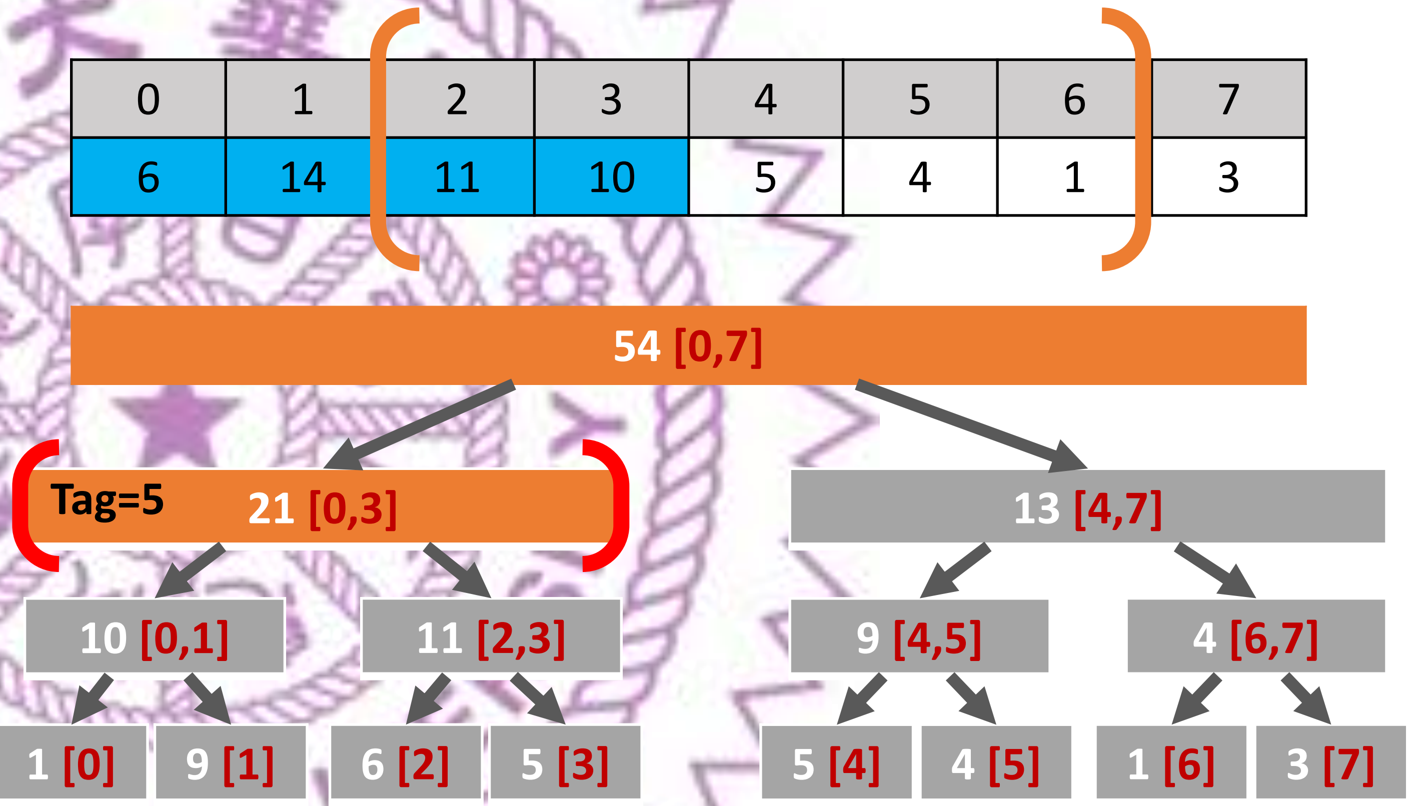
0	1	2	3	4	5	6	7
6	14	11	10	5	4	1	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

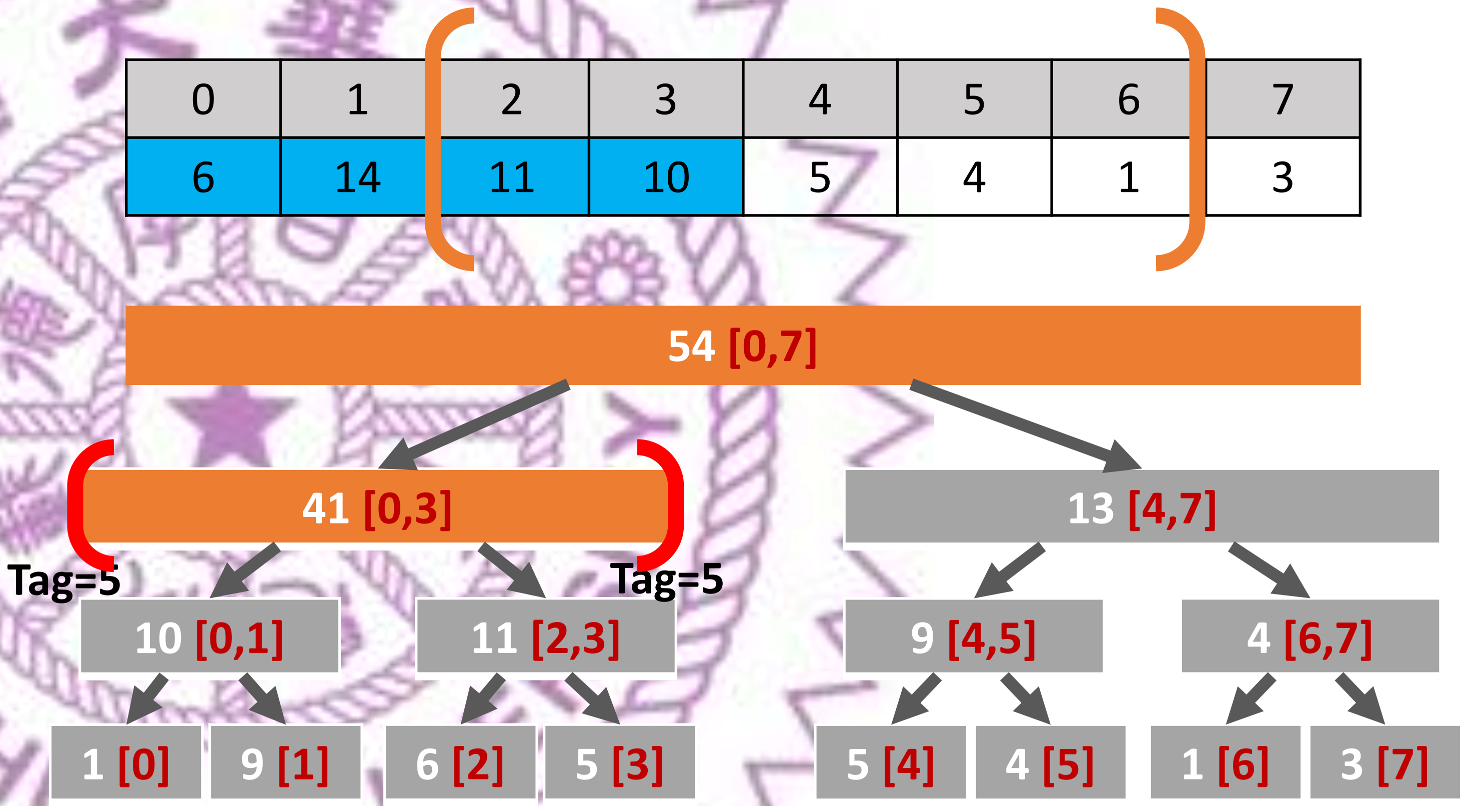
- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例

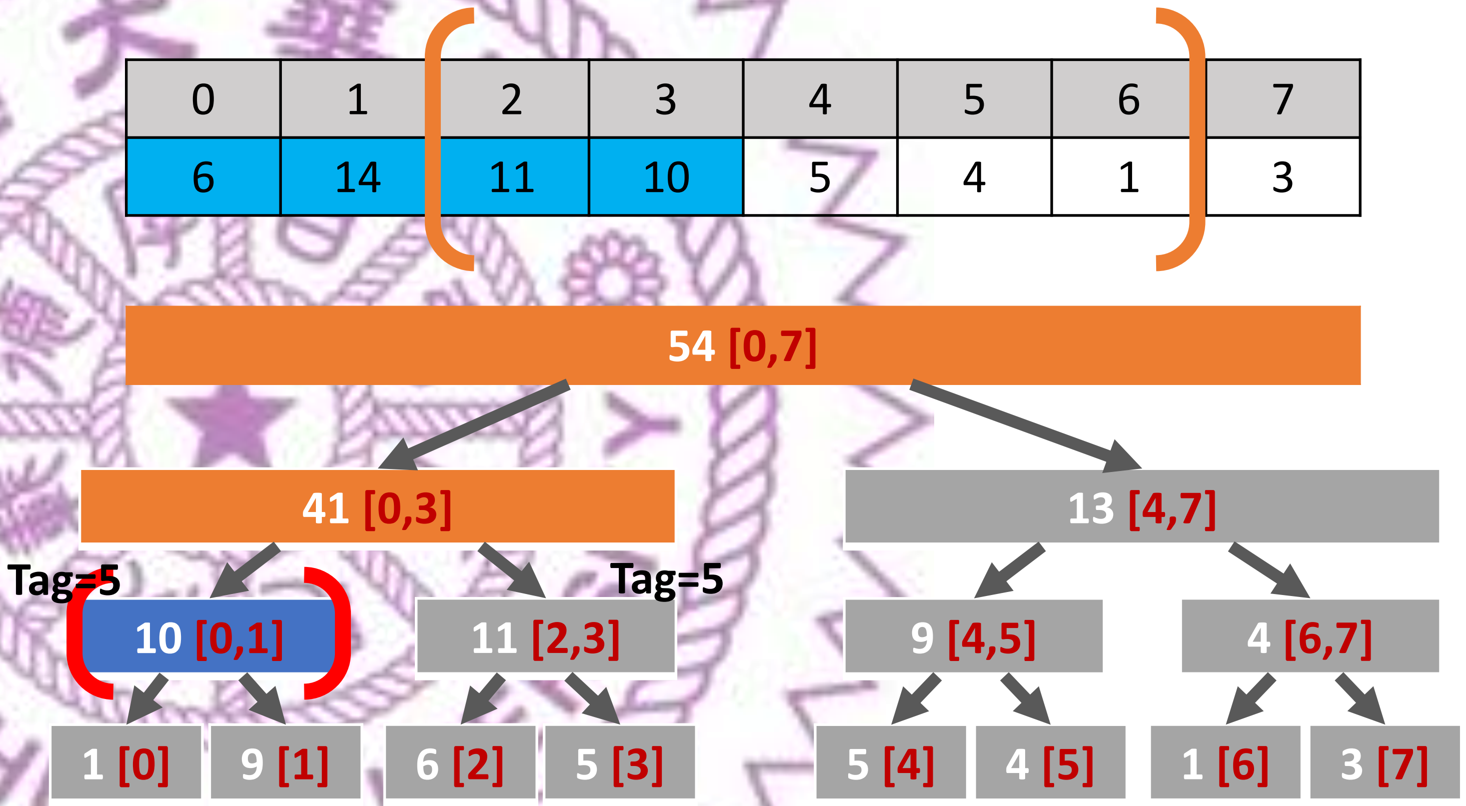
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

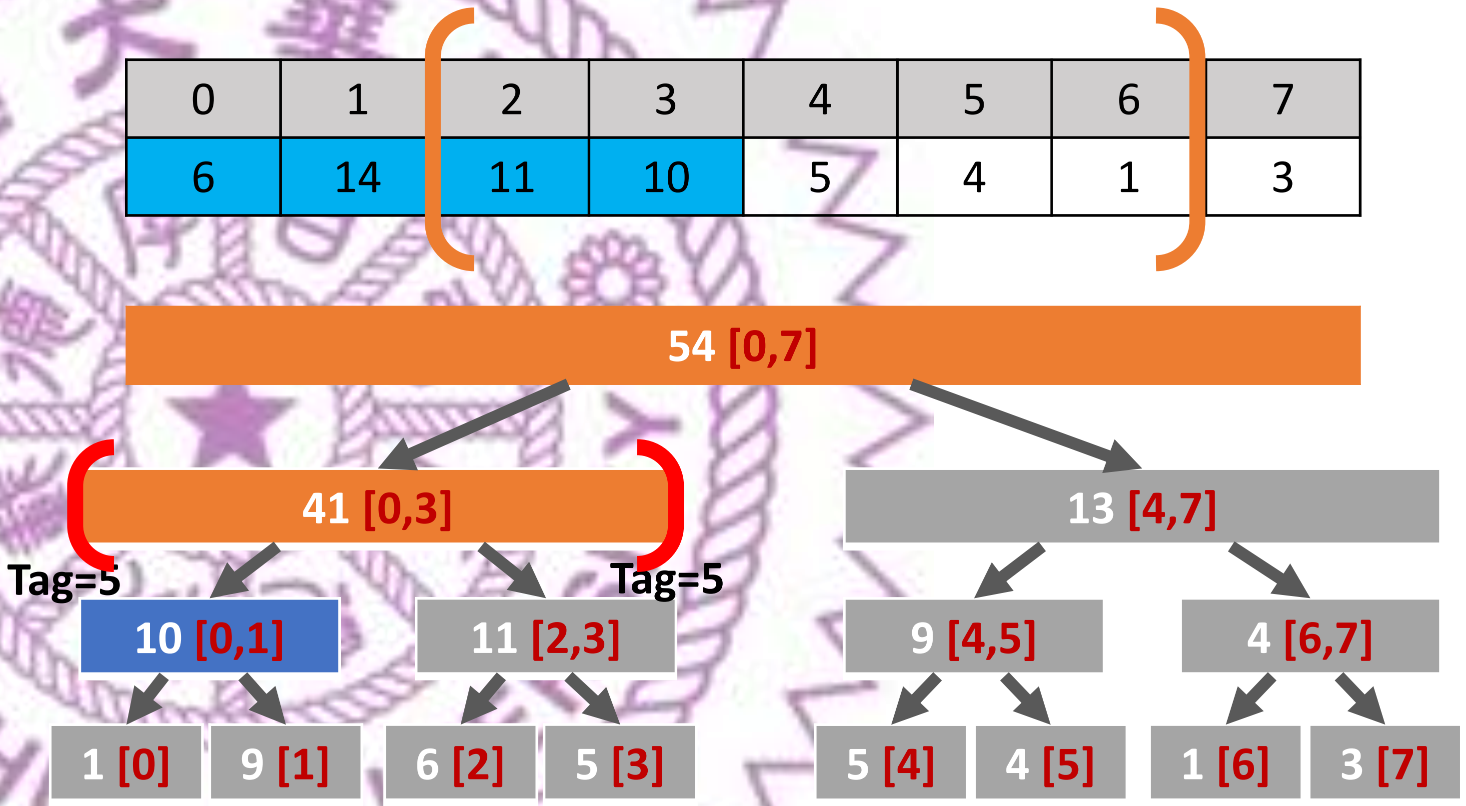
- 區間 [2, 6] 都加上 3 為例

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

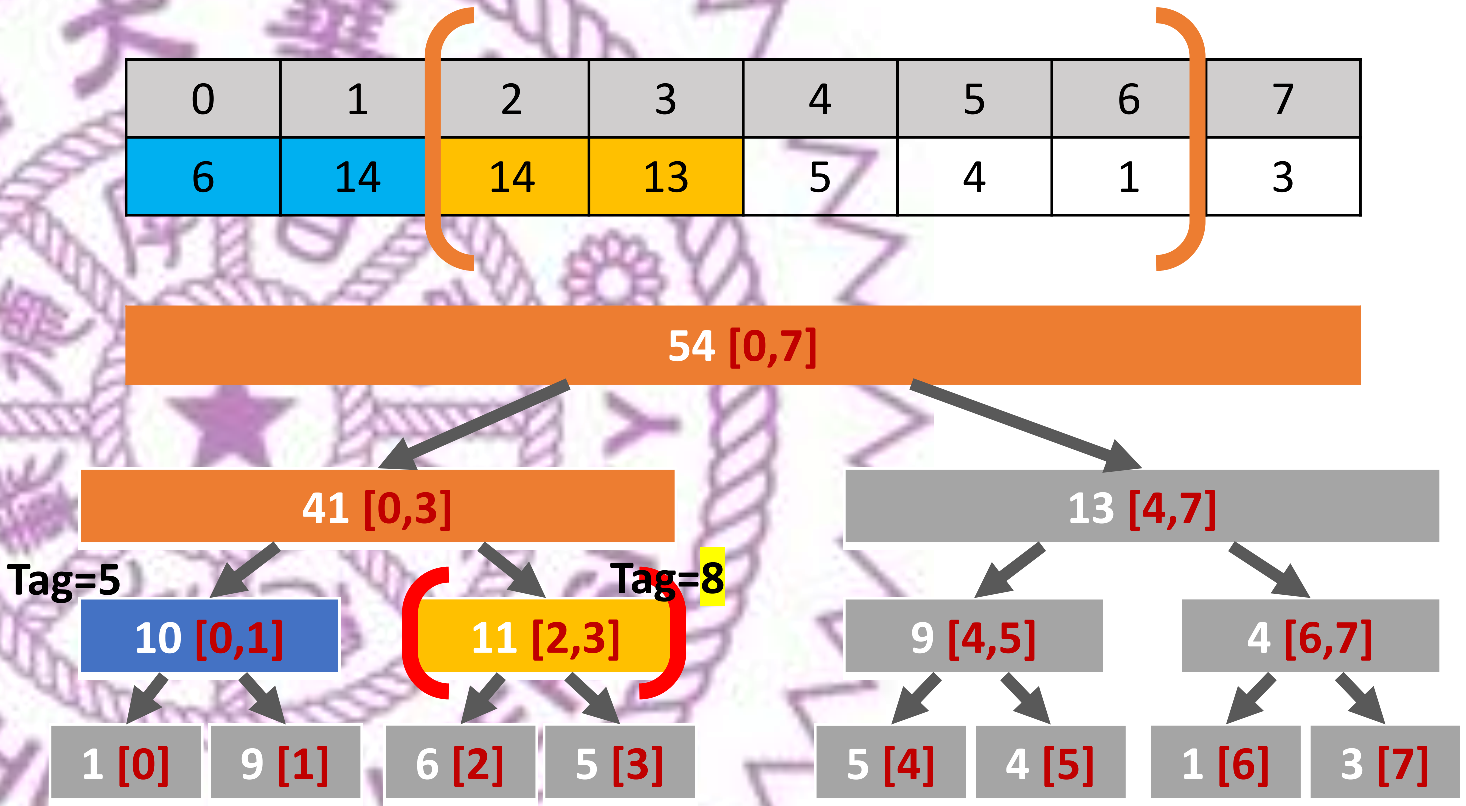
- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

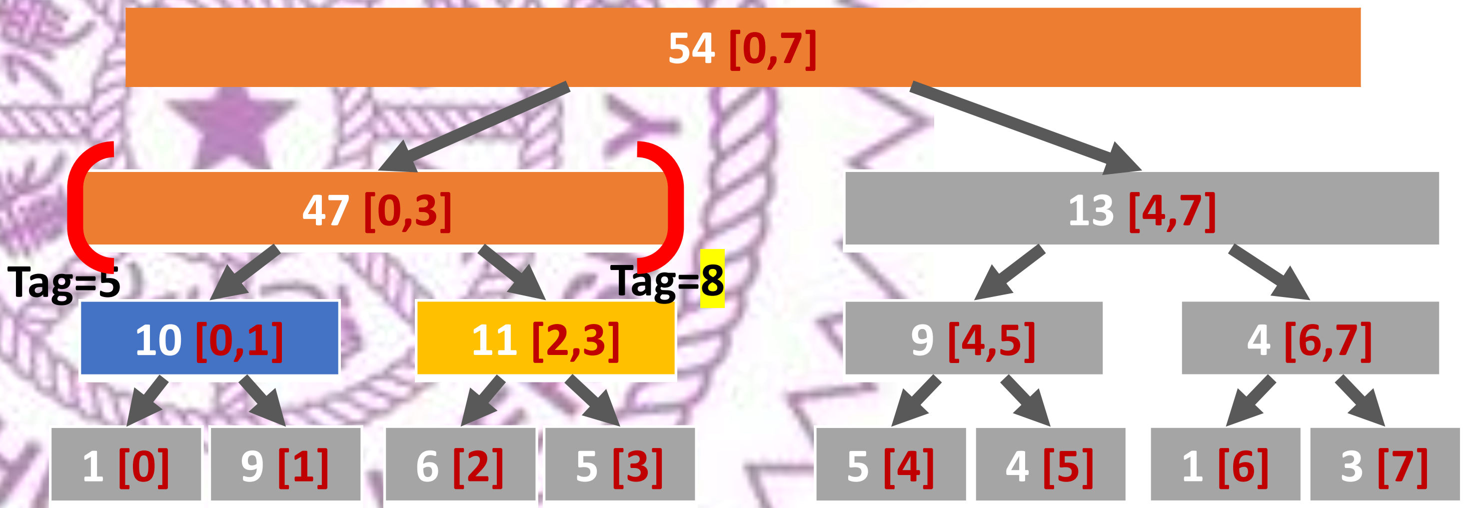


區間加值

- 區間 [2, 6] 都加上 3 為例

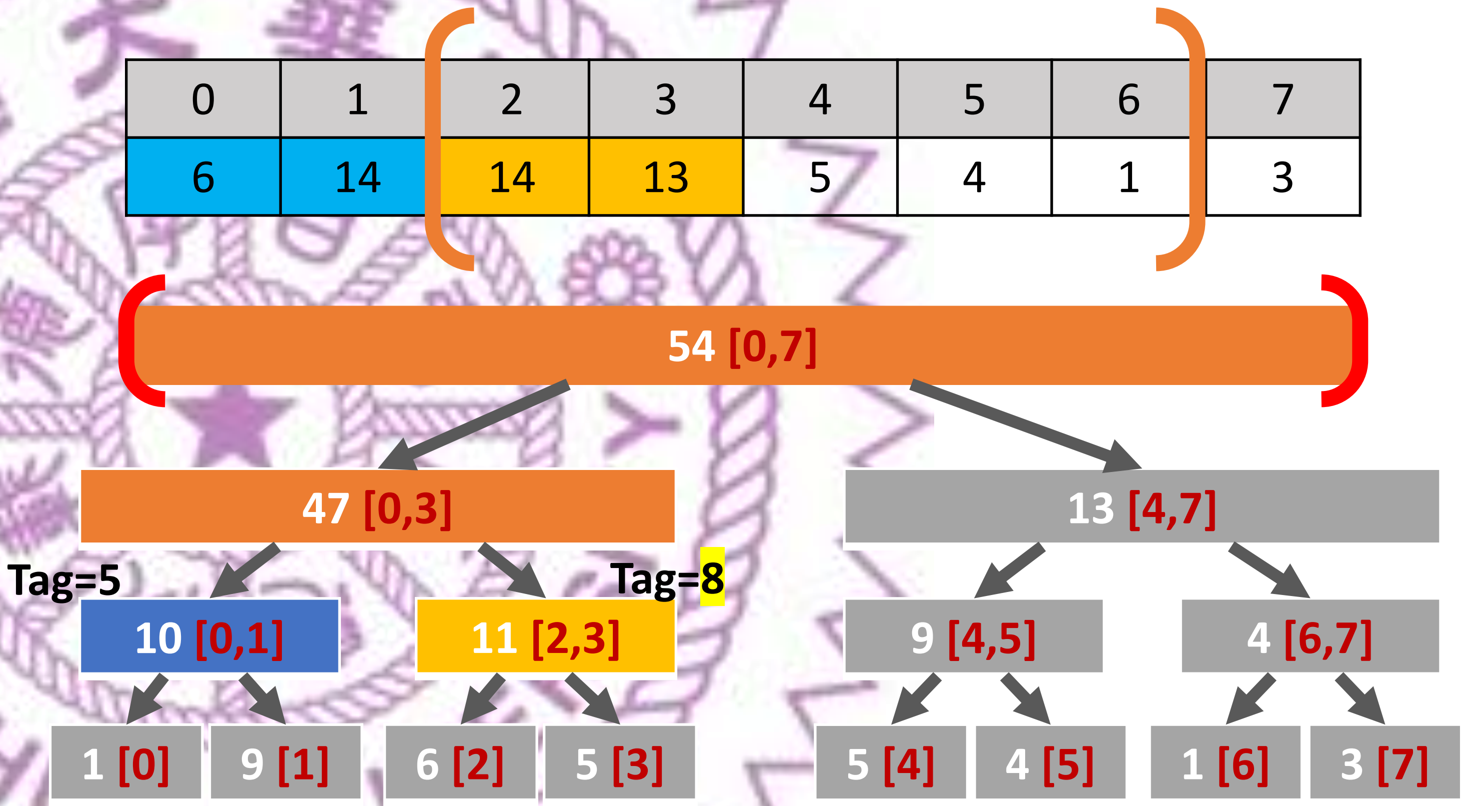
0	1	2	3	4	5	6	7
6	14	14	13	5	4	1	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

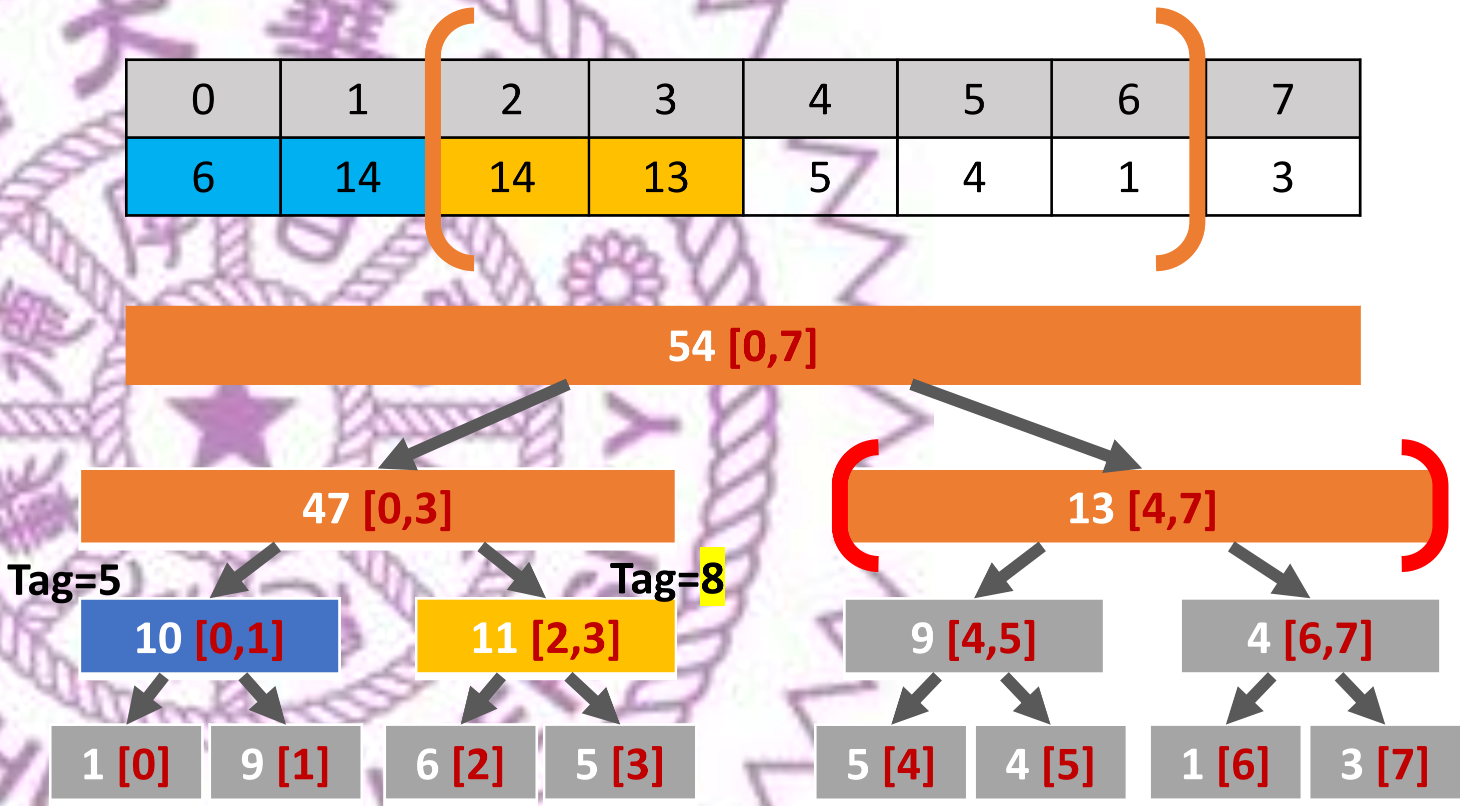
- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

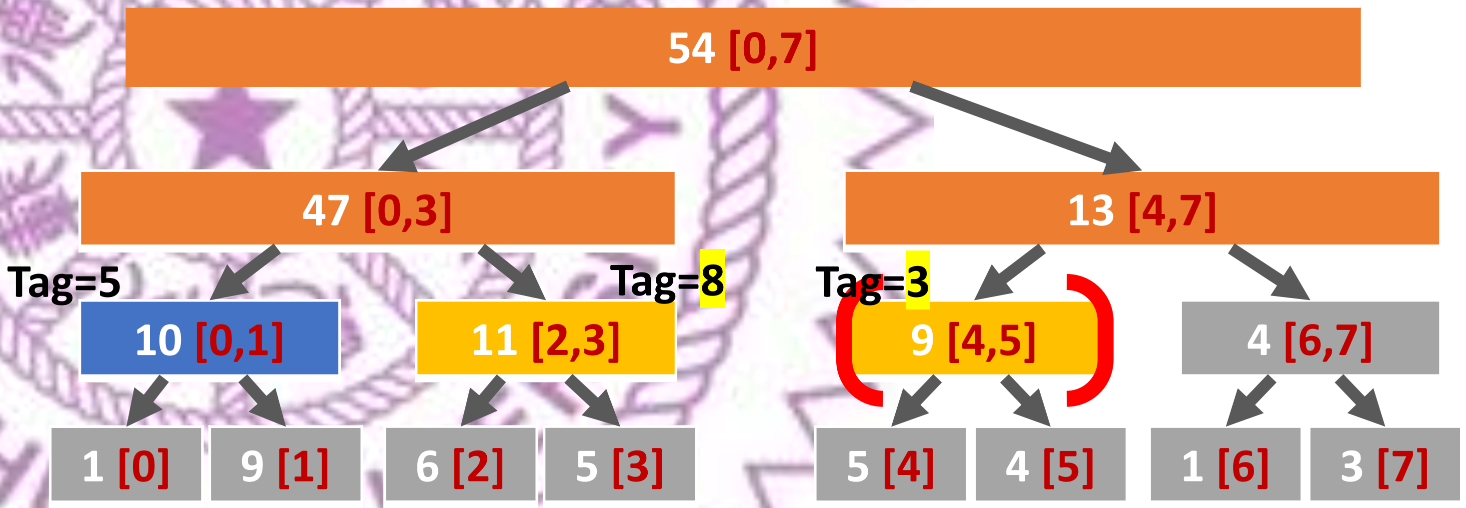


區間加值

- 區間 [2, 6] 都加上 3 為例

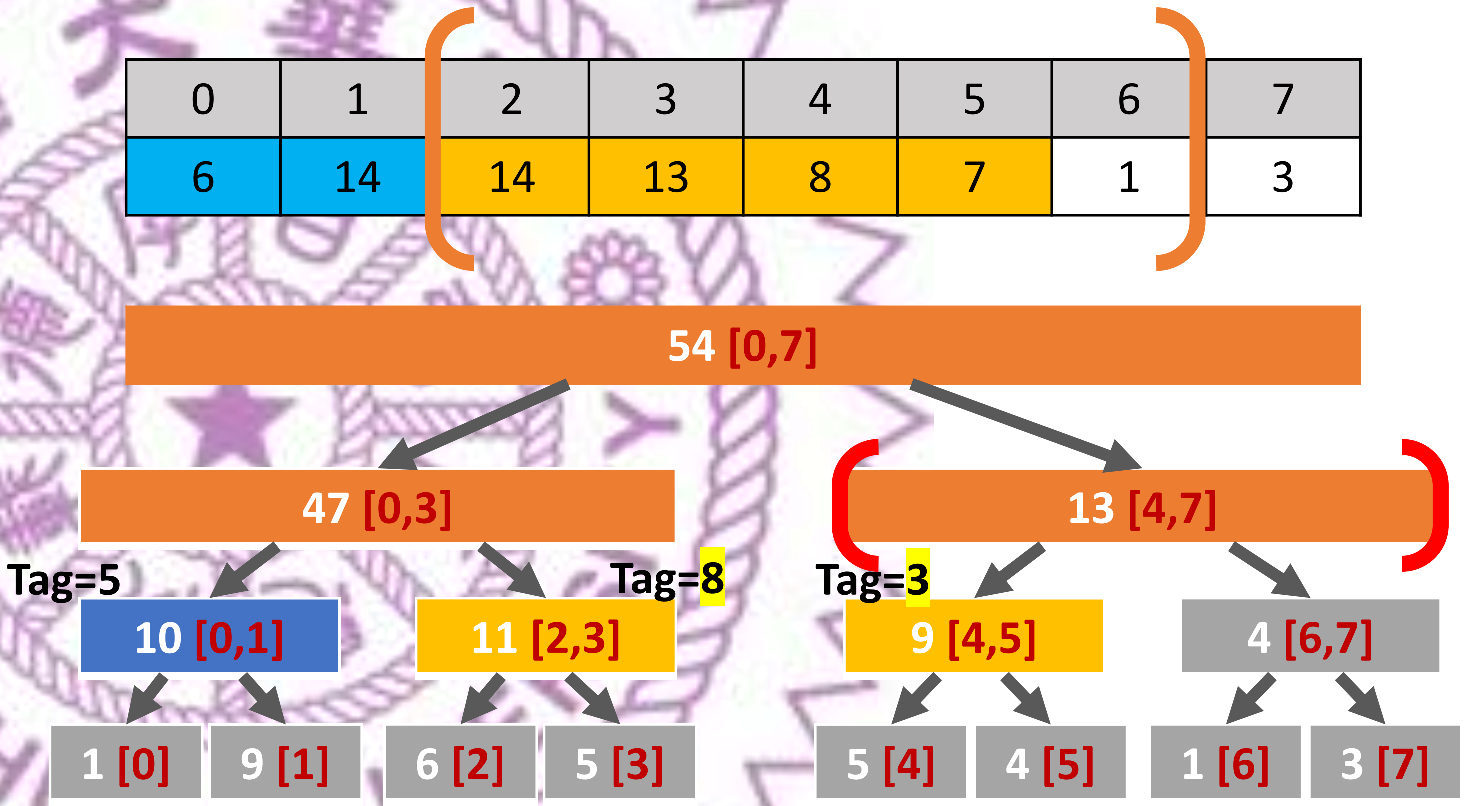
0	1	2	3	4	5	6	7
6	14	14	13	8	7	1	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



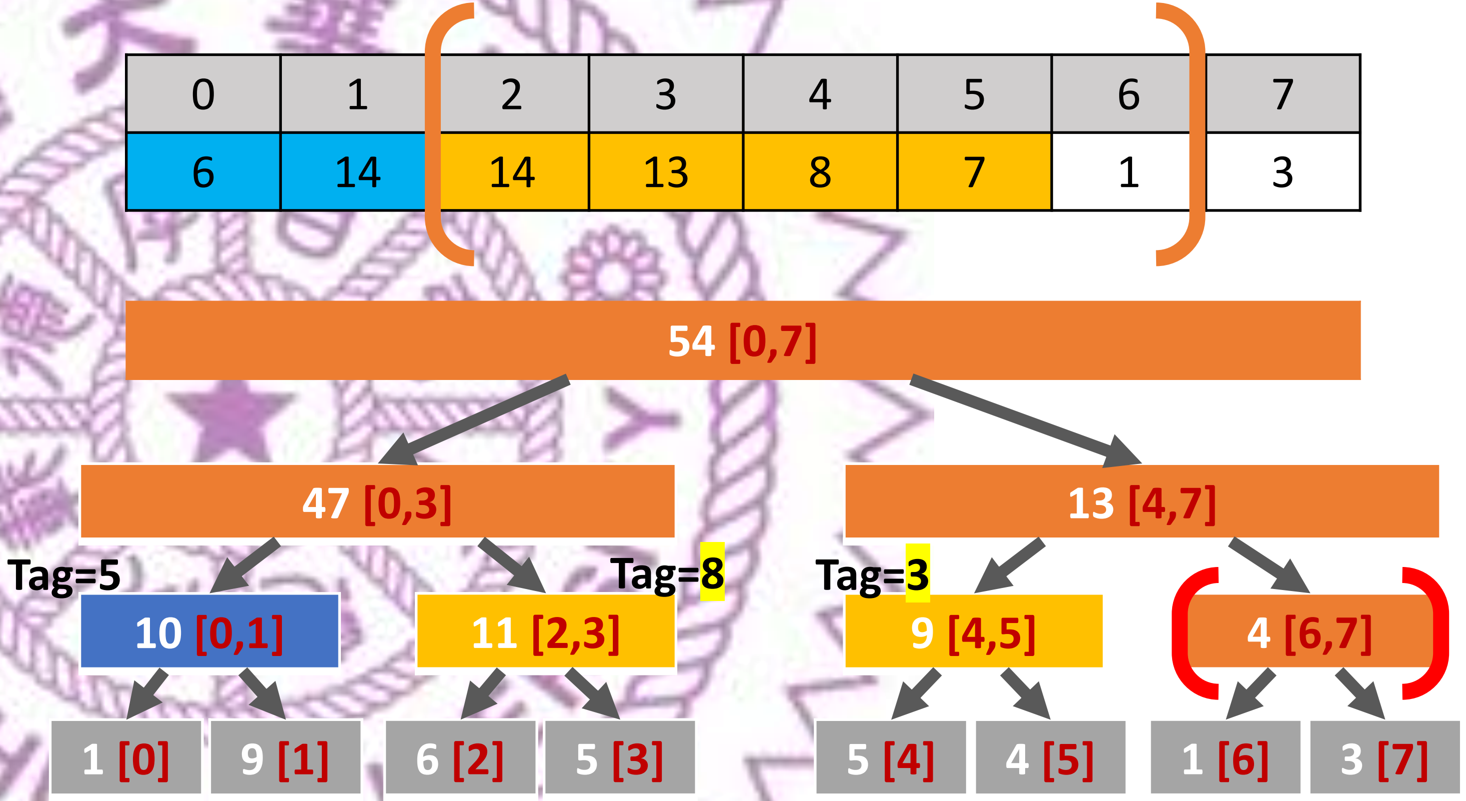
區間加值

- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

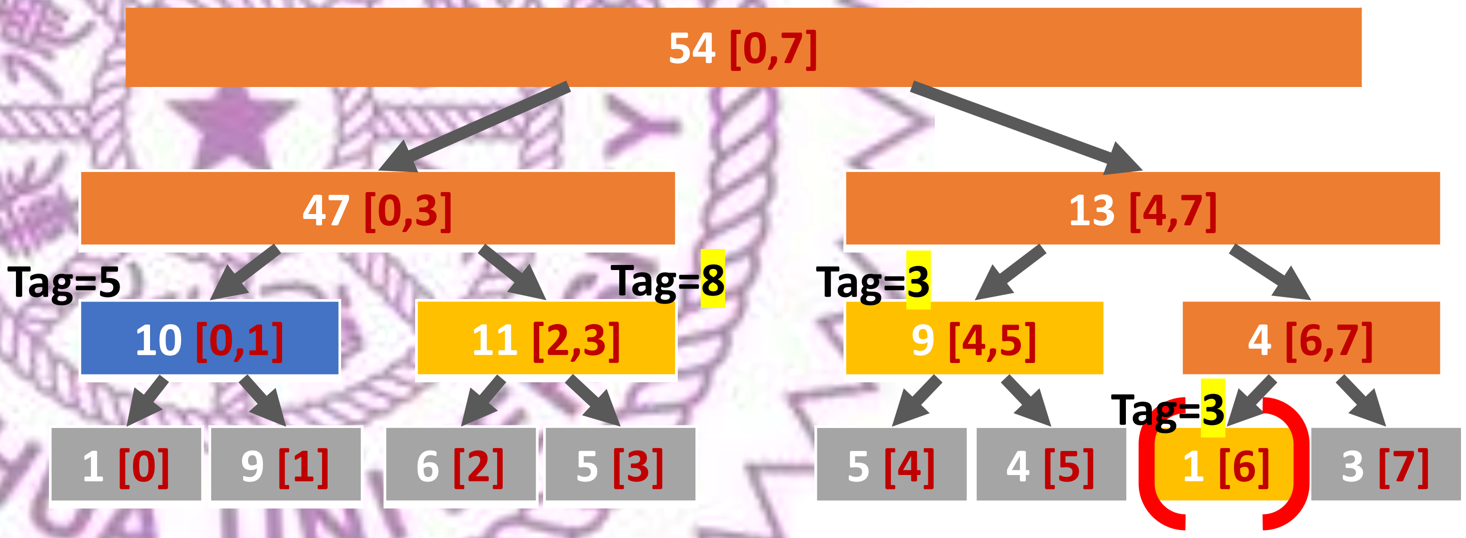
- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

0	1	2	3	4	5	6	7
6	14	14	13	8	7	4	3

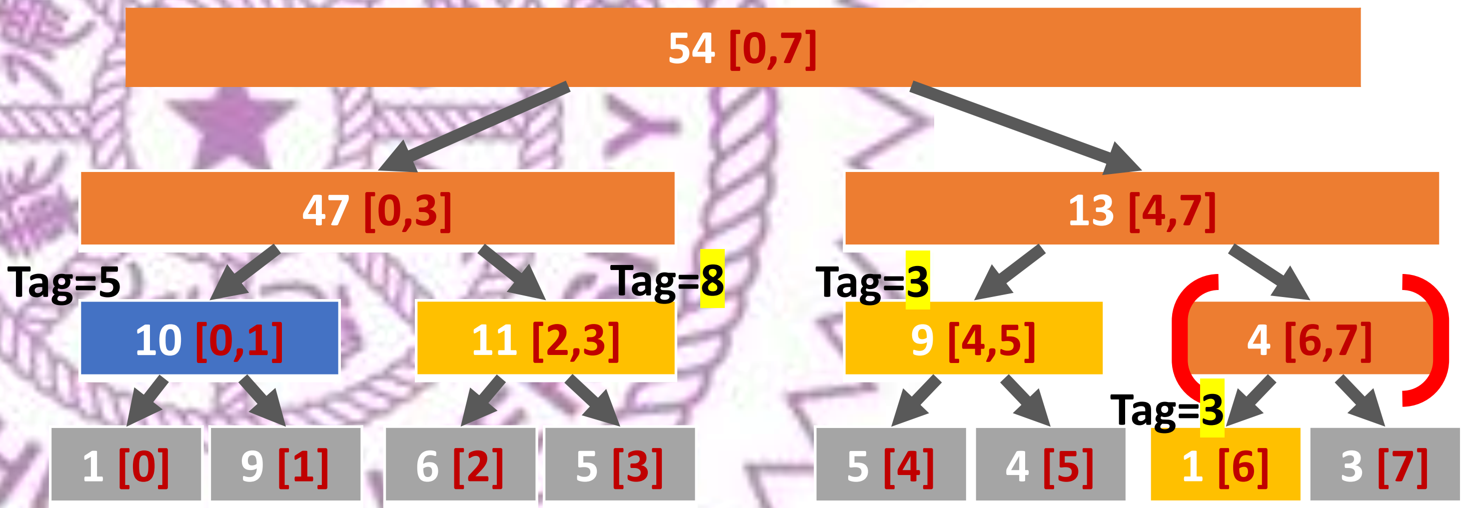


區間加值

- 區間 [2, 6] 都加上 3 為例

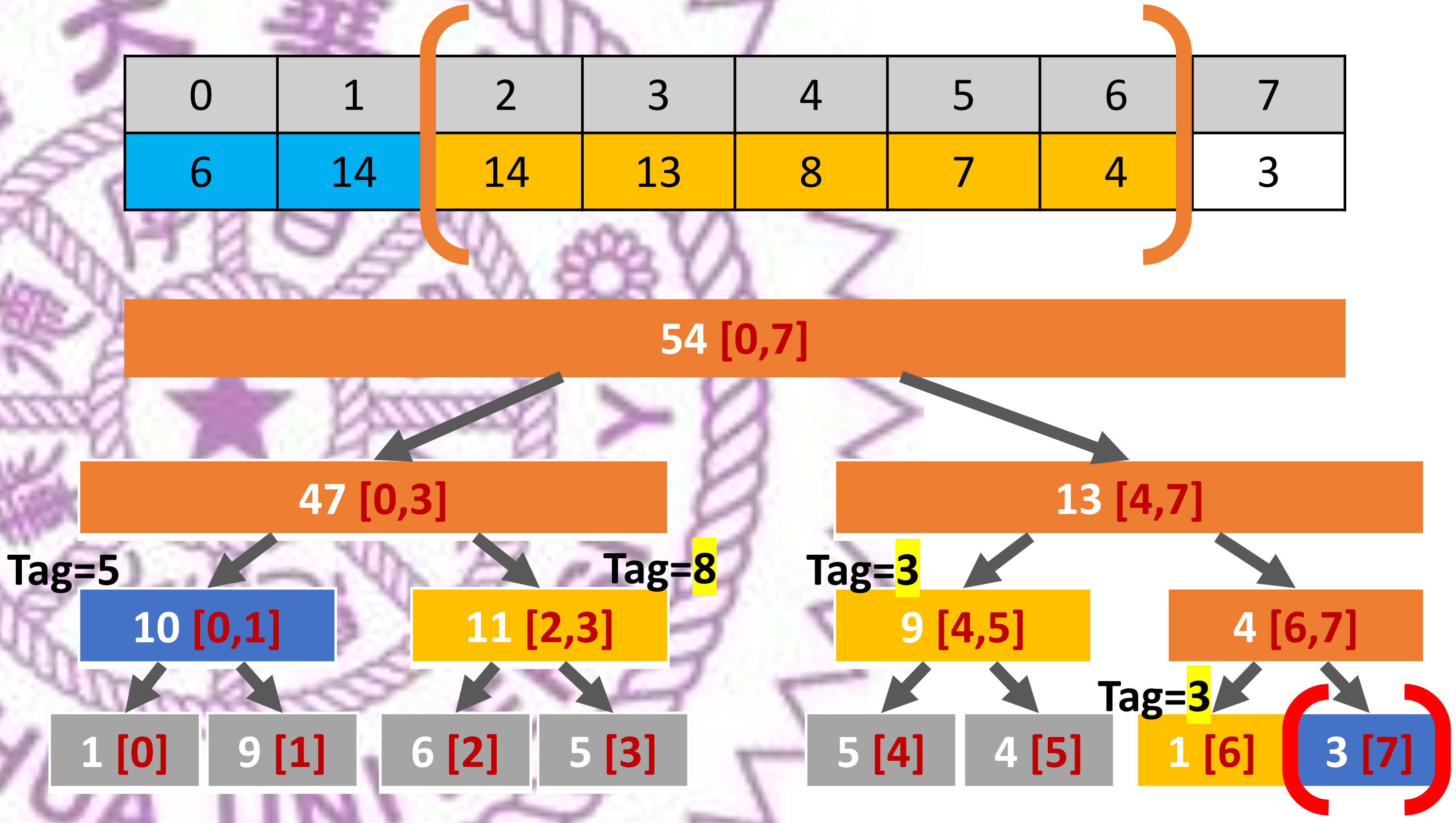
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

0	1	2	3	4	5	6	7
6	14	14	13	8	7	4	3



區間加值

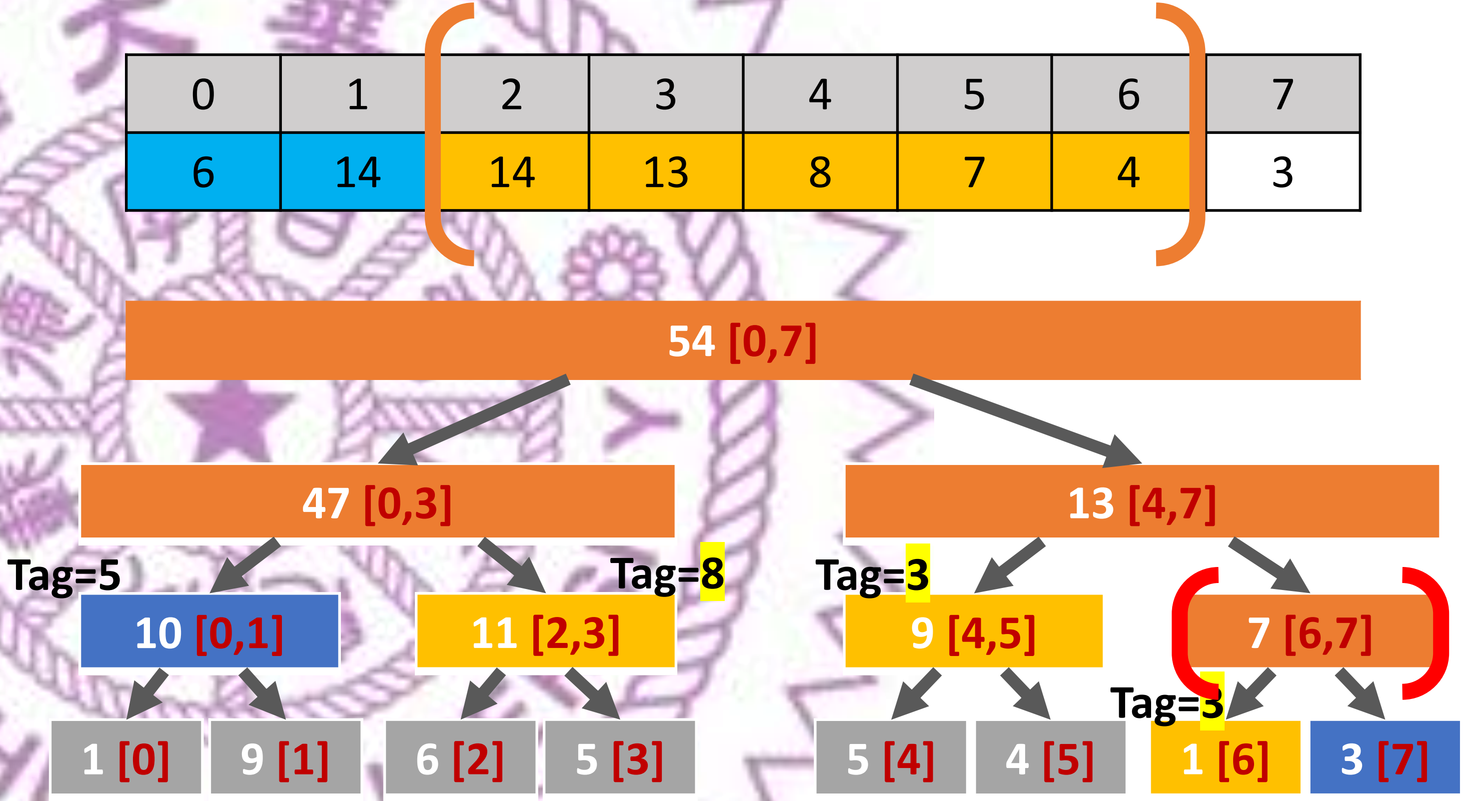
- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內

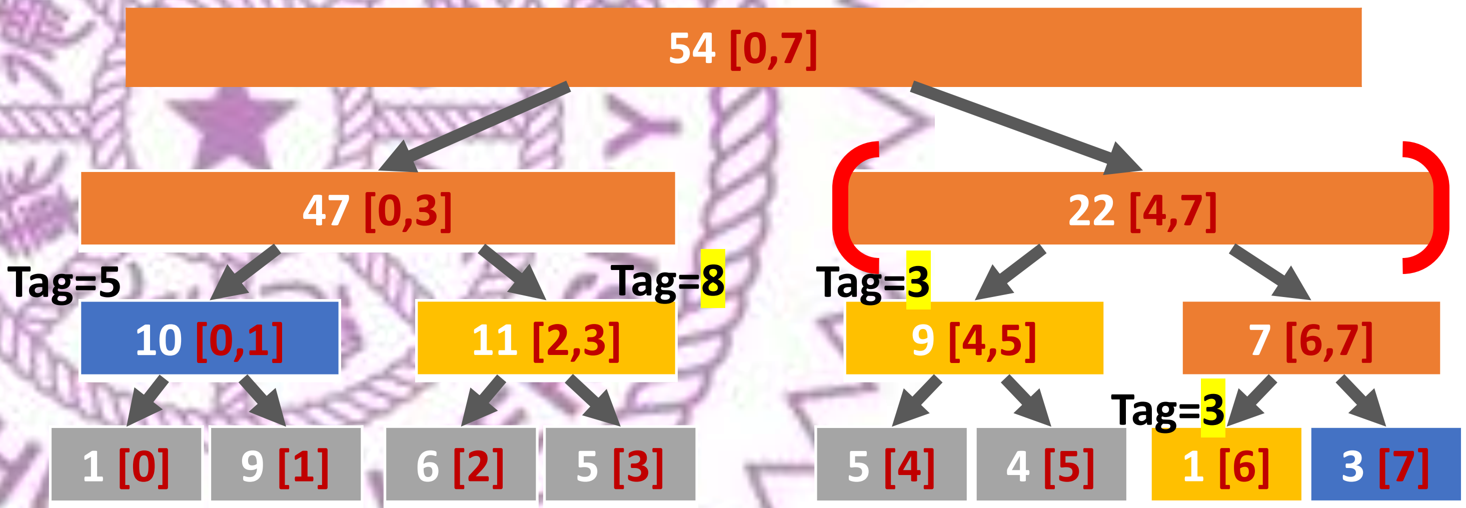


區間加值

- 區間 [2, 6] 都加上 3 為例

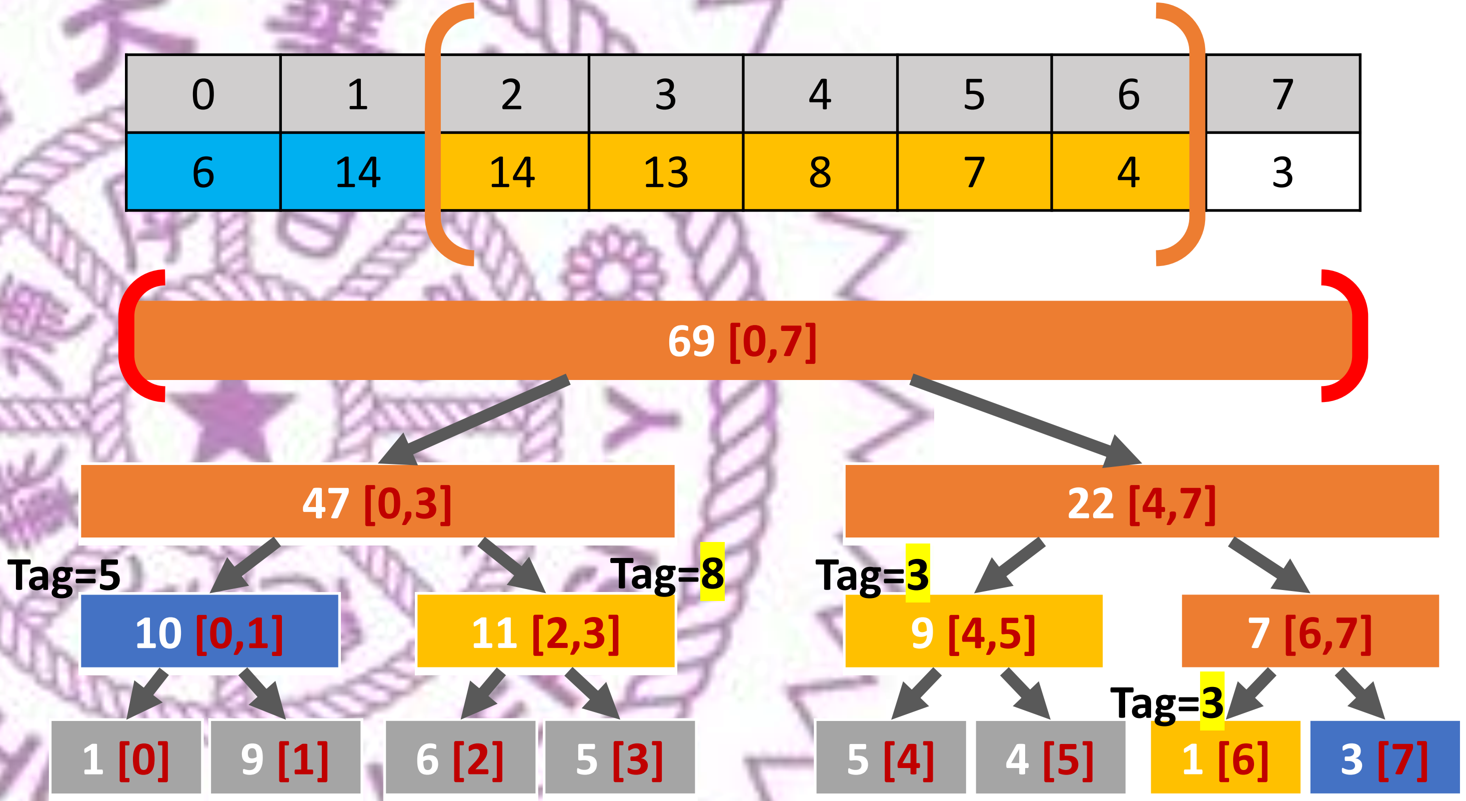
0	1	2	3	4	5	6	7
6	14	14	13	8	7	4	3

- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



區間加值

- 區間 [2, 6] 都加上 3 為例
- Case 1: 不在範圍內
- Case 2: 完全位於範圍內
- Case 3: 部分位於範圍內



時間複雜度

- 與區間查詢經過的點一模一樣
- $O(\log n)$

總結

查詢、修改

- Case 1: 不再範圍內，不處理
- Case 2: 剛好位於範圍中，直接處理
- Case 3: 剩下的情況，分兩半遞迴

懶惰標記

- 節點取值時要記得考慮
- Case 3 分兩半前 push
- 如果是修改操作遞迴結束後 pull
- 懶惰標記不只一個時要考慮先後順序

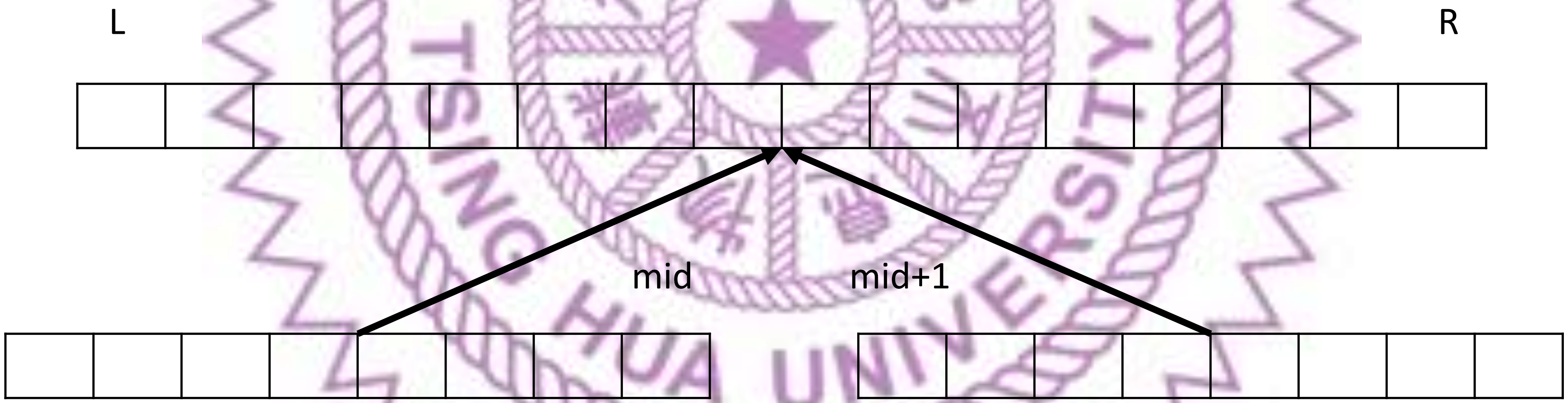
區間最大連續和 Maximum Consecutive Sum

- 給你一個長度為 n 的陣列 a，再給你 q 個操作，操作有兩種：
- $query(ql, qr)$:
  查詢閉區間 $[a_{ql}, a_{qr}]$ 中的最大連續和
- $update(p, val)$:
  將 $a_p = val$
- $1 \leq n, q \leq 10^6$

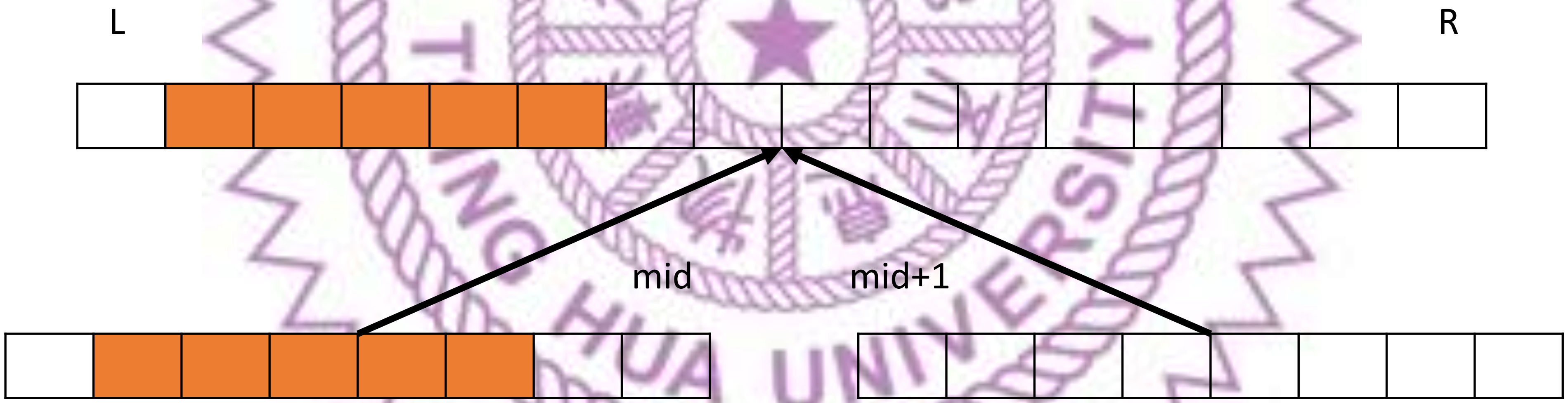
$query(3, 7) = 11$

a	0	1	2	3	4	5	6	7
	1	9	6	-5	5	4	-1	3

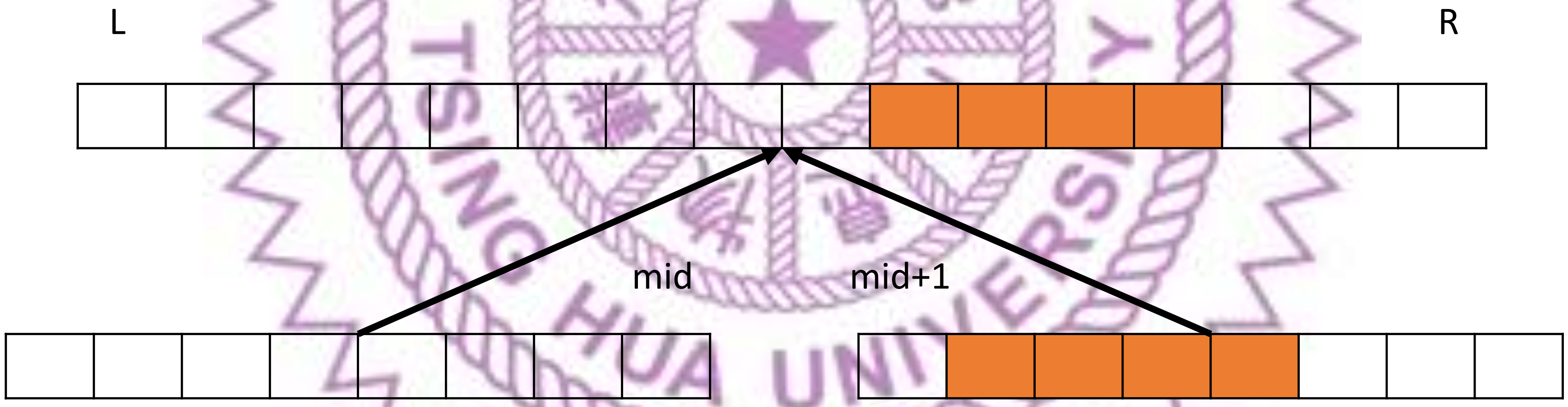
分治法 – 只有三種可能



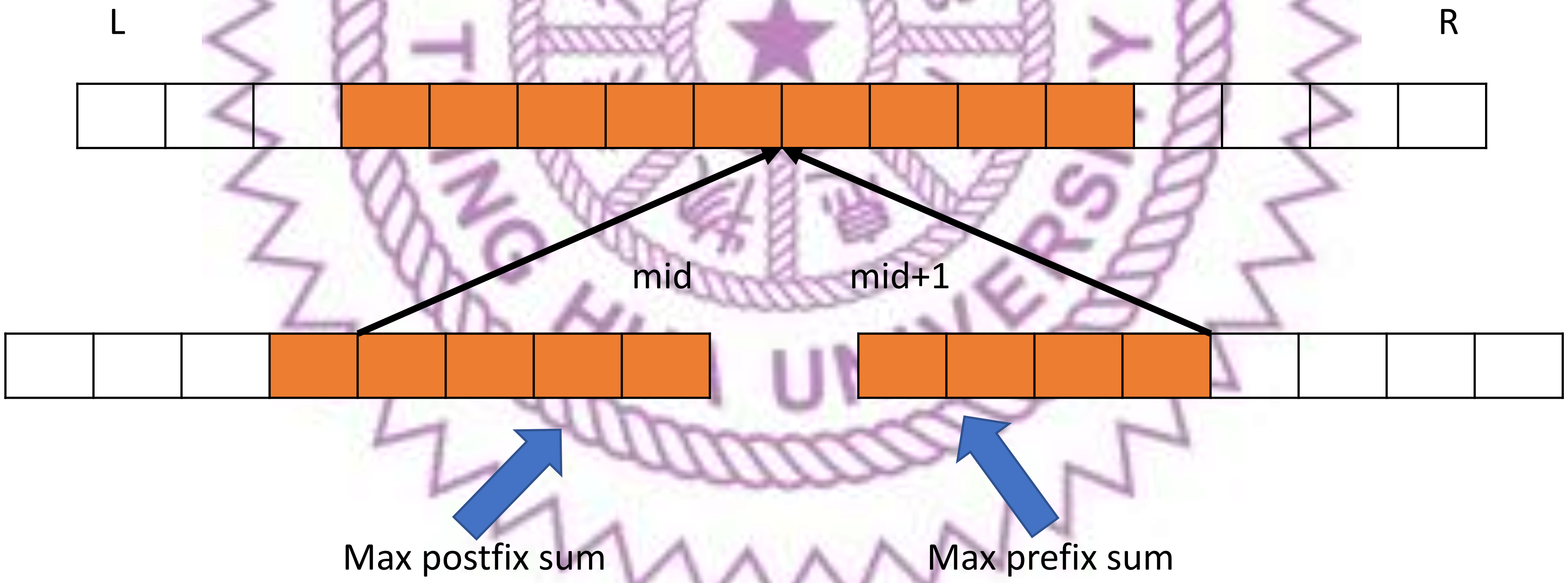
Case 1: 最大連續和在左邊



Case 2: 最大連續和在右邊



Case 3: 最大連續和橫跨左右兩邊

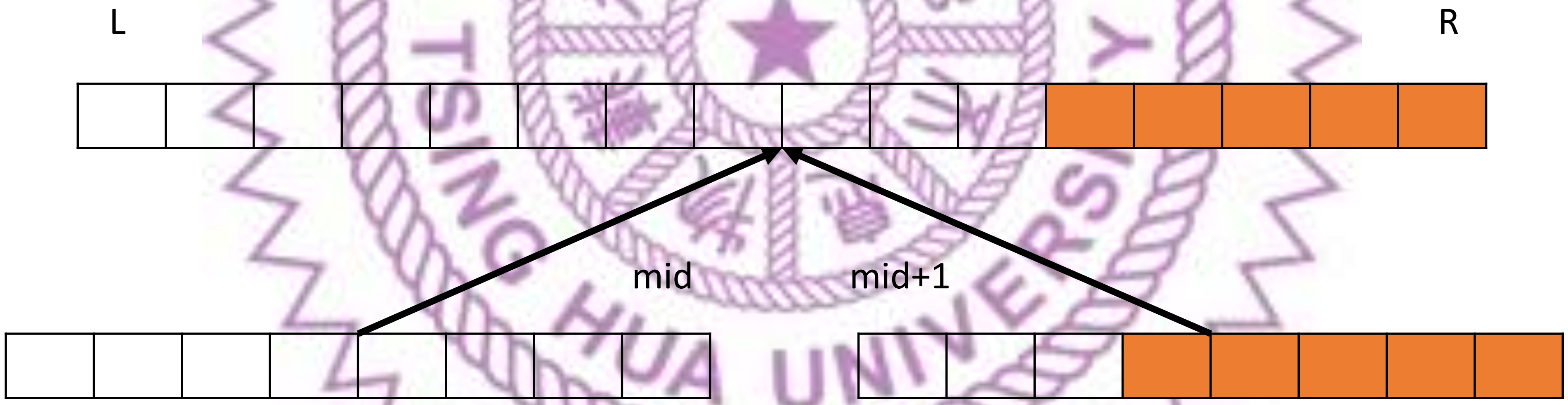


簡單的公式

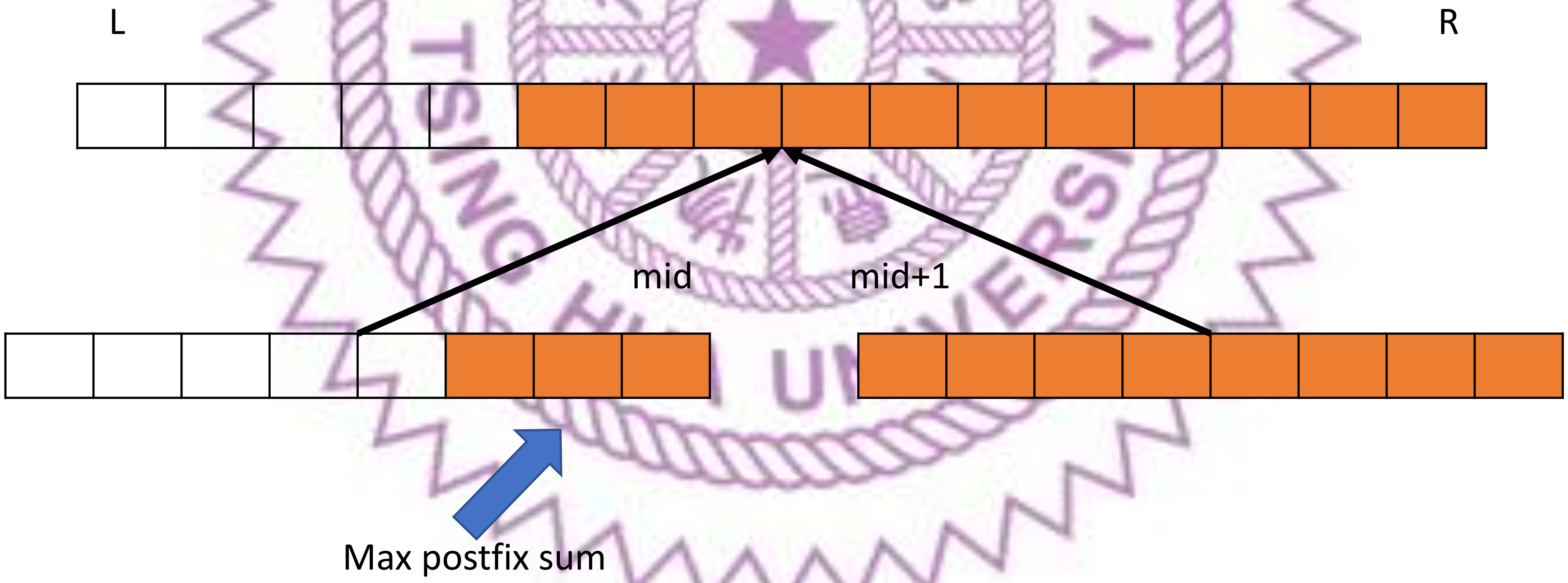
- 設 $MCS(l,r)$ 表示區間 $[l,r]$ 的最大連續和， $mid = \lfloor l + r \rfloor / 2$
  $prefix_max(mid+1,r)$表示區間 $[mid+1,r]$ 的 max prefix sum
  $postfix_max(l,mid)$表示區間 $[l,mid]$ 的 max postfix sum

$$MCS(l,r) = \max\begin{cases} MCS(l,mid) \\ MCS(mid+1,r) \\ postfix\_max(l,mid) + prefix\_max(mid+1,r) \end{cases}$$

Max postfix sum – 全部在右邊



Max postfix sum – 左右兩邊都有



簡單的公式

- 設 $sum(l,r)$ 表示區間 $[l,r]$ 的總和， $mid = \lfloor l + r \rfloor / 2$

$$postfix\_max(l,r) = \max \begin{cases} postfix\_max(l,mid) + sum(mid+1,r) \\ postfix\_max(mid+1,r) \end{cases}$$

Max prefix sum 同理

- 設 $sum(l,r)$ 表示區間 $[l,r]$ 的總和， $mid = \lfloor l + r \rfloor / 2$

$$prefix\_max(l,r) = \max\begin{cases} sum(l,mid) + prefix\_max(mid+1,r) \\ prefix\_max(l,mid) \end{cases}$$

線段樹的節點需要 4 個資訊

```
struct Item {
  int sum, MCS;
  int prefix_max, postfix_max;

  Item(int sum = 0)
      : sum(sum), MCS(max(sum, 0)), prefix_max(MCS), postfix_max(MCS) {}
  friend Item operator+(const Item &L, const Item &R) {
    Item res(L.sum + R.sum);
    res.MCS = max({L.MCS, R.MCS, L.postfix_max + R.prefix_max});
    res.prefix_max = max(L.prefix_max, L.sum + R.prefix_max);
    res.postfix_max = max(R.postfix_max, R.sum + L.postfix_max);
    return res;
  }
};
```

線段樹的構造

```
Item seg[4 * MAXN];
void build(int l, int r, int id = 1) {
  if (l == r) {
    seg[id] = Item(a[l]);
    return;
  }
  int m = (l + r) / 2;
  build(l, m, 2 * id);
  build(m + 1, r, 2 * id + 1);
  pull(id);
}
build(0, n - 1);
```

```
void pull(int id) {
  seg[id] = seg[id * 2] + seg[id * 2 + 1];
}
```

查詢區間最大連續和

```
Item query(int ql, int qr, int l, int r, int id = 1) {
  if (qr < l || r < ql) // [l,r] 不在 [ql,qr] 的範圍
    return Item(0);
  if (ql <= l && r <= qr) // [l,r] 被 [ql,qr] 完全包含
    return seg[id];
  int m = (l + r) / 2; // 剩下就遞迴處理
  return query(ql, qr, l, m, id * 2) + query(ql, qr, m + 1, r, id * 2 + 1);
}
query(ql, qr, 0, n - 1);
```