樹堆 Treap

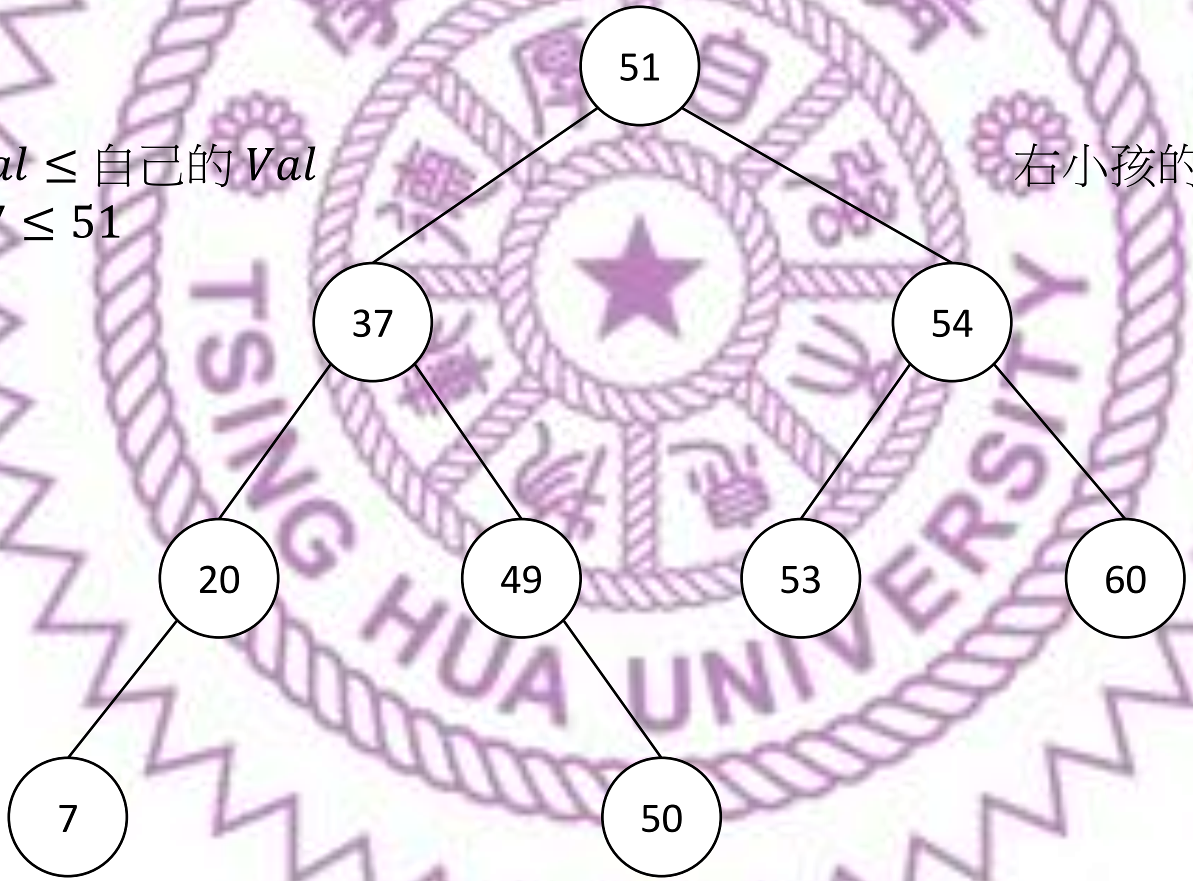
日月卦長

二元搜尋樹

每個點存一個值 Val

左小孩的 $Val \leq$ 自己的 Val
$37 \leq 51$

右小孩的 $Val \geq$ 自己的 Val
$51 \leq 54$



平衡二元搜尋樹

- 保證深度為 $O(\log n)$ 等級的二元搜尋樹
- C++ 中有內建：
  - std::set、std::map
- pb_ds 黑魔法：

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
```

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using ll = long long;
tree<ll, null_type, less<ll>, rb_tree_tag, tree_order_statistics_node_update> BST;
```

名次樹

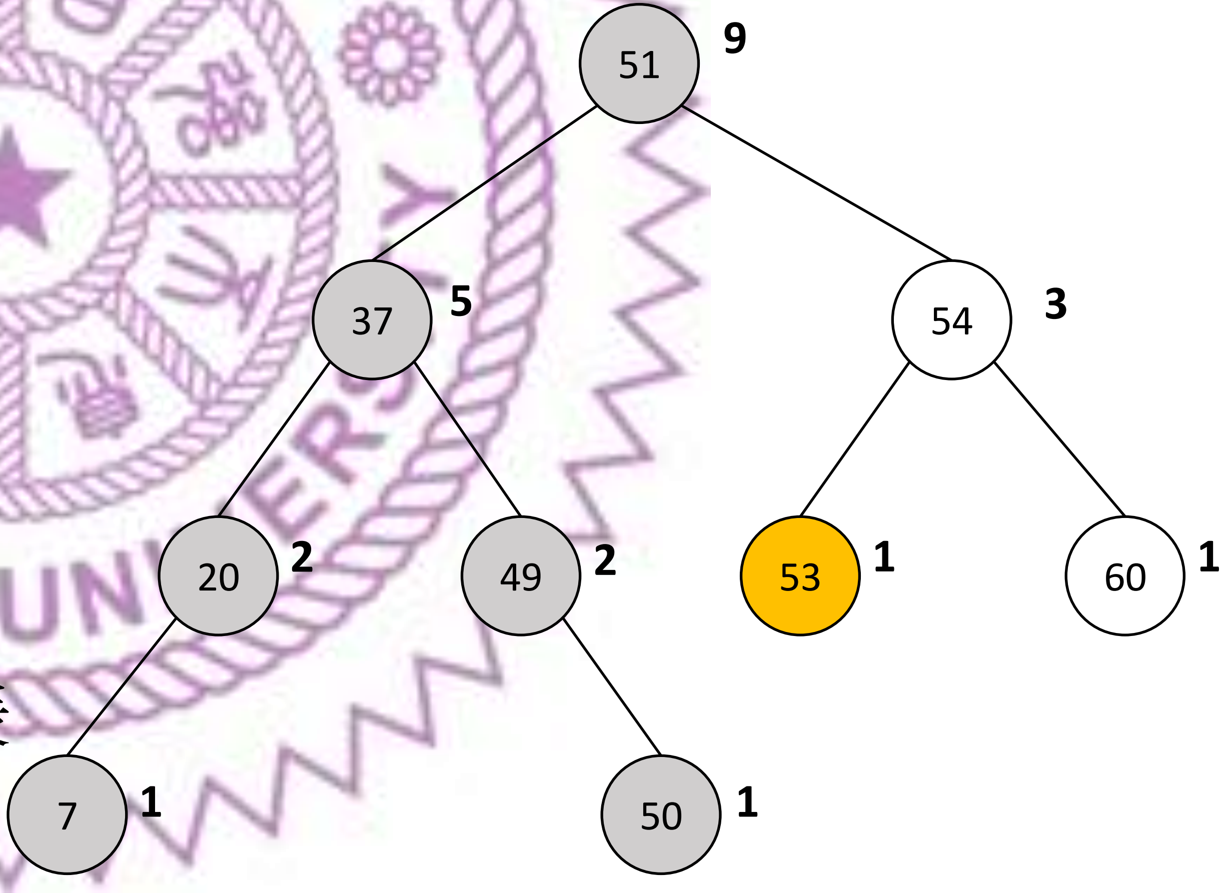
- 比普通二元搜尋樹多兩個操作

- $rank(x)$:
  - 查詢樹中 $< x$ 的元素有幾個
- $kth(K)$:
  - 查詢樹中第 K 小的元素

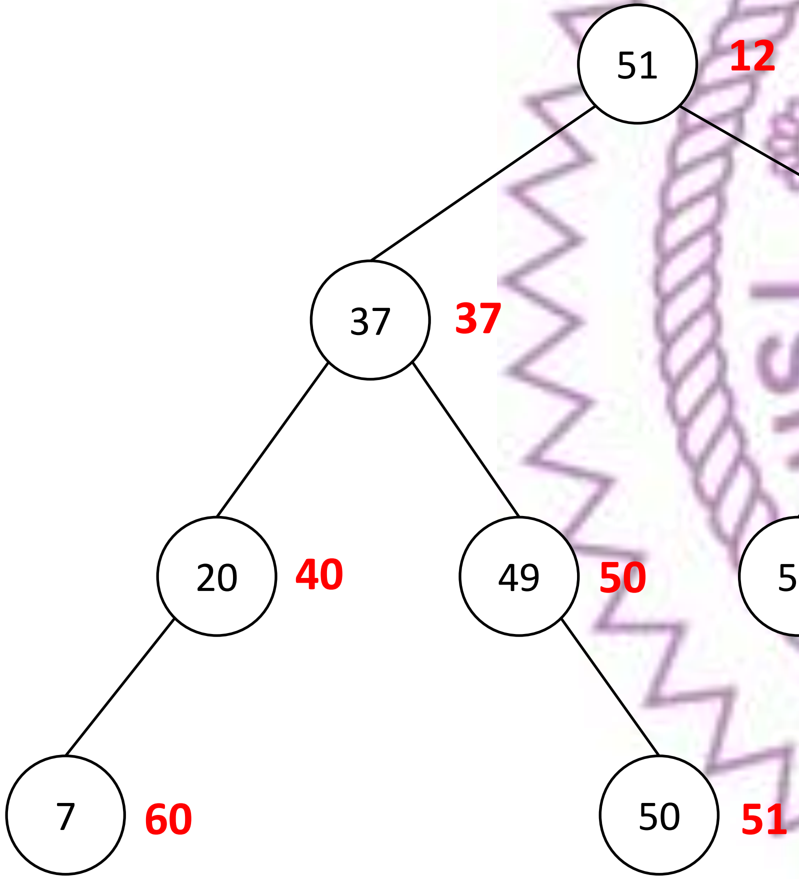
- 每個點要記錄 size
- 不想自己寫的話只有 pb_ds 支援

$rank(53) = 6$

$Kth(2) = 20$



Treap = (Binary Search) Tree + Heap



Key 滿足二元搜尋樹性質
pri 隨機生成，滿足 Heap 性質
size 用來計算名次

```
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  unsigned pri, size;
  int Key;
  Treap(int Key) :
    pri(rand()), size(1), Key(Key) {}
  void pull();
};
unsigned size(Treap *x) {
  return x ? x->size : 0;
}
void Treap::pull() {
  size = 1u + ::size(lc) + ::size(rc);
}
```

Treap 深度

- 如果 pri 值足夠隨機
  可以讓一顆 n 個點的 Treap ，每個點有 $\frac{1}{n}$ 的機率當根

- 因此如果你相信 quick sort 的平均遞迴深度是 $O(\log n)$
  那 n 個點 Treap 平均深度就會是 $O(\log n)$

基本操作

一般 Binary Search Tree

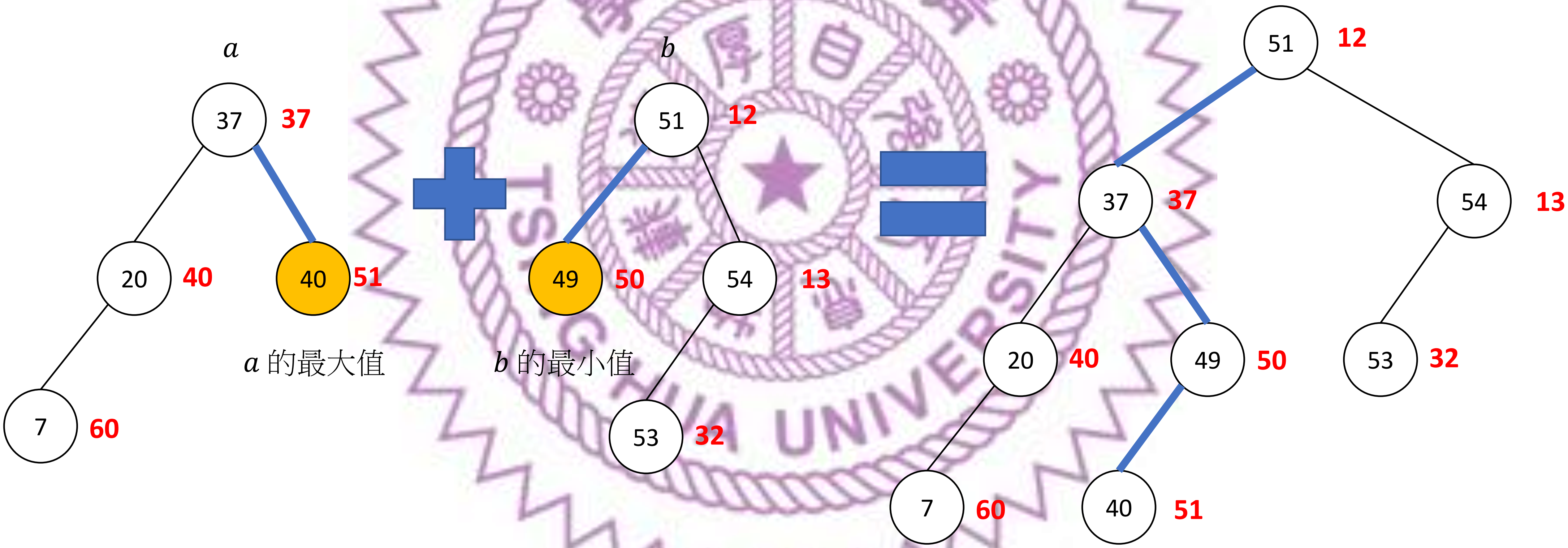
- $insert(root, Key)$
- $erase(root, Key)$
- $find(root, Key)$

Treap

- $merge(a, b)$
- $split(x, Key)$
- 用這兩個操作能拼出 insert、erase、find 三種操作

$merge(a,b)$

只有在a 的最大值小於等於 b 的最大值時才能呼叫，否則結果不會是二元搜尋樹

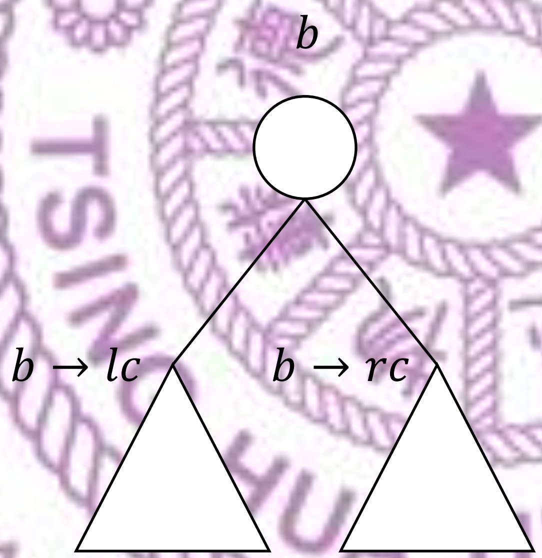


誰要當根?

Case: a 或 b 其中一個是 $nullptr$

a

$nullptr$

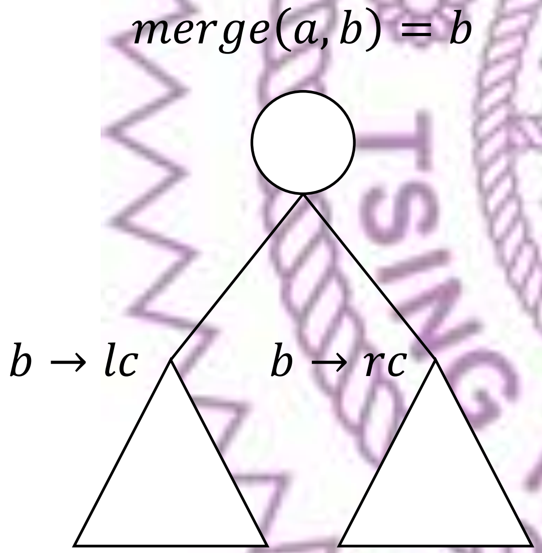



```
Treap *merge(Treap *a, Treap *b) {

}
```

誰要當根?

Case: a 或 b 其中一個是 $nullptr$

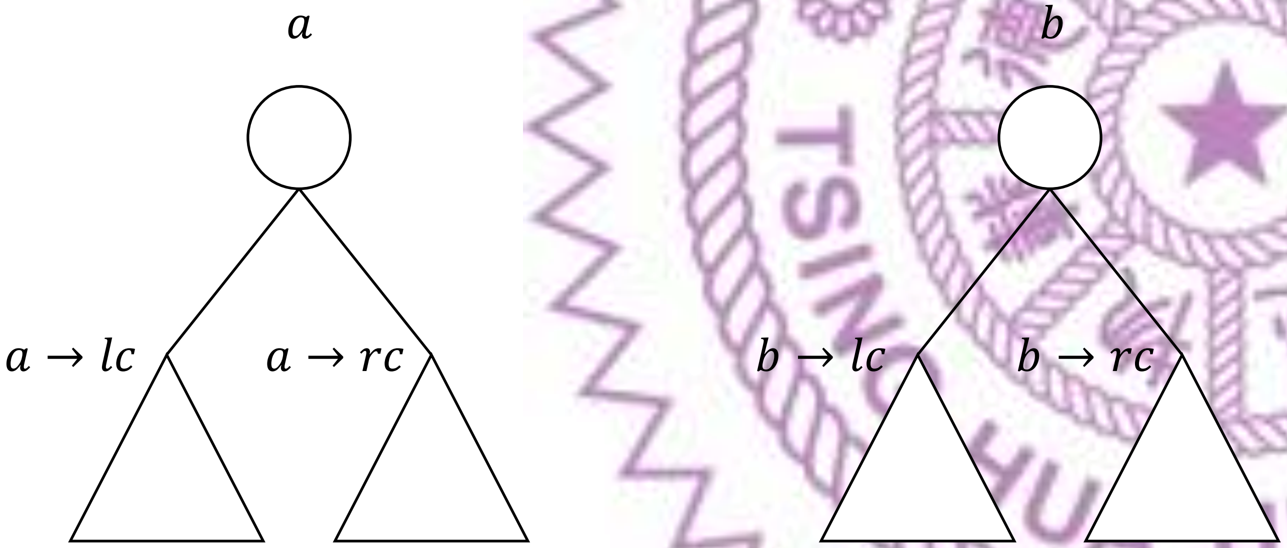
回傳不是 $nullptr$ 的那個




```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;







}
```

誰要當根?

Case: $a \to pri < b \to pri$

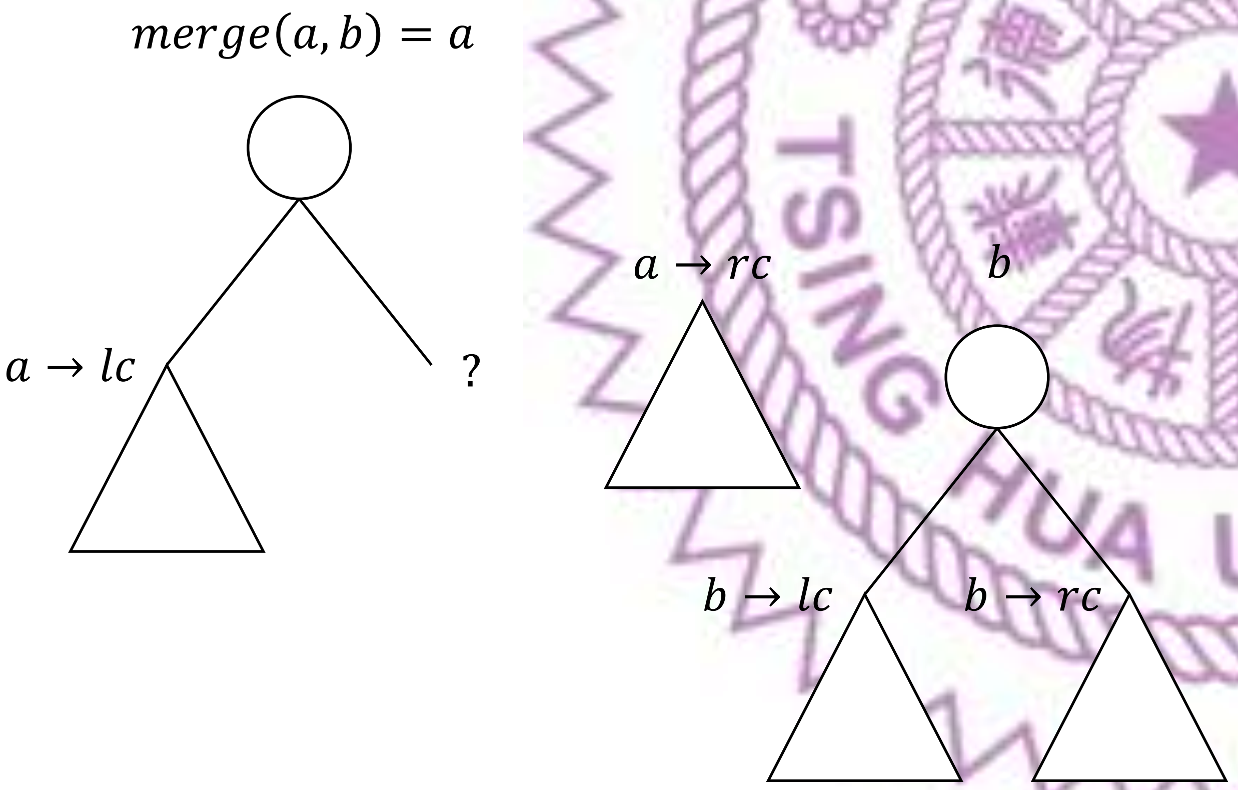



```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {









}
```

誰要當根?

Case: $a \rightarrow pri \ < \ b \rightarrow pri$

a 要當根，根的左子樹依然是原本 $a \rightarrow lc$



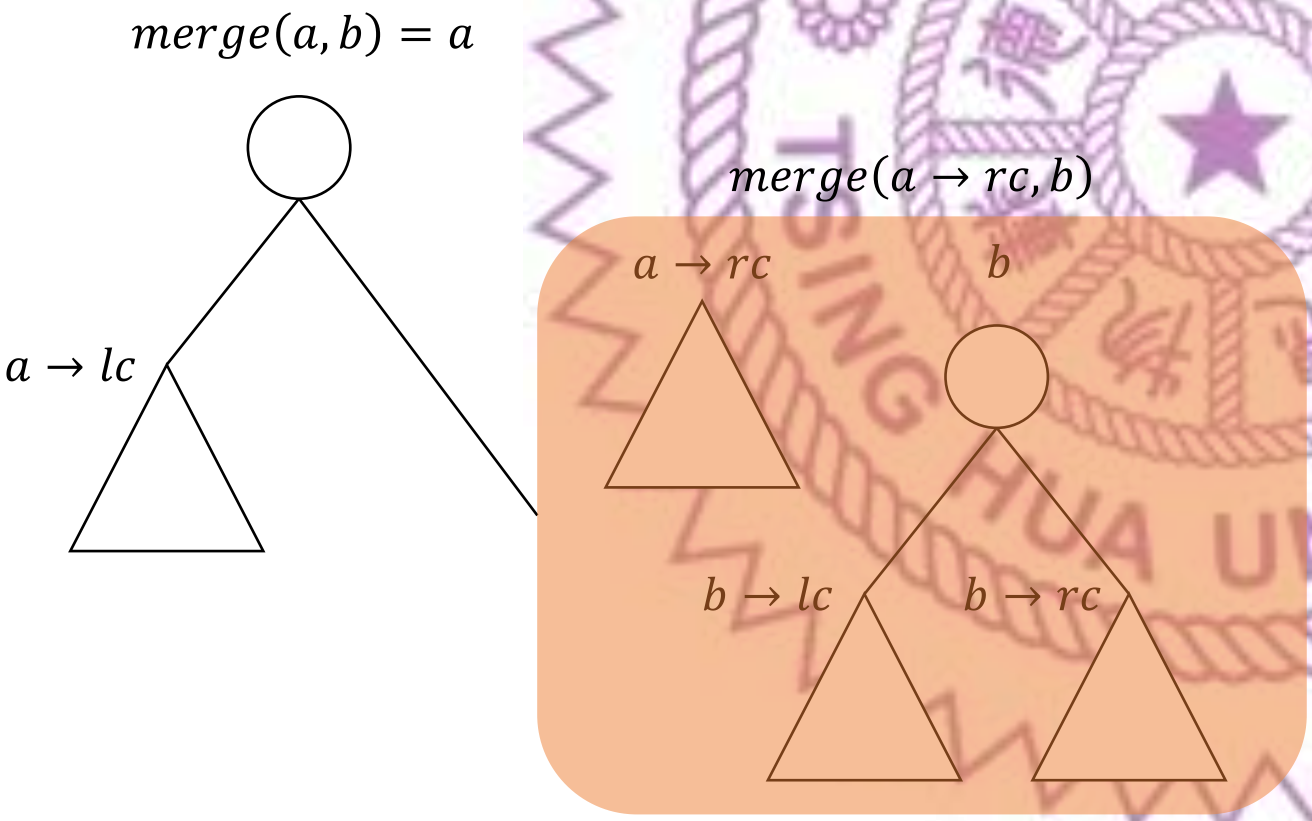

```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {


}
```

誰要當根?

Case: $a \to pri \ < \ b \to pri$

a 要當根，根的左子樹依然是原本 $a \to lc$

根的右子樹就遞迴計算！

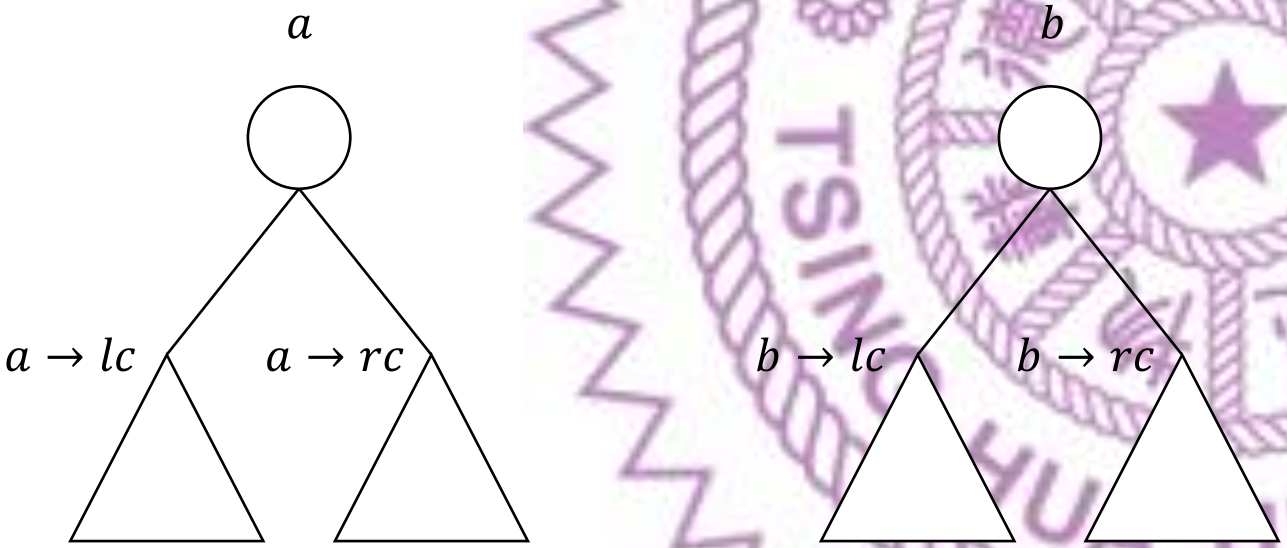


樹的結構確定後要記得呼叫 pull 算出正確的 size

```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {
    a->rc = merge(a->rc, b);
    a->pull();
    return a;
  } else {
  
  
  
  
}
```

誰要當根?

Case: $a \to pri \geq b \to pri$

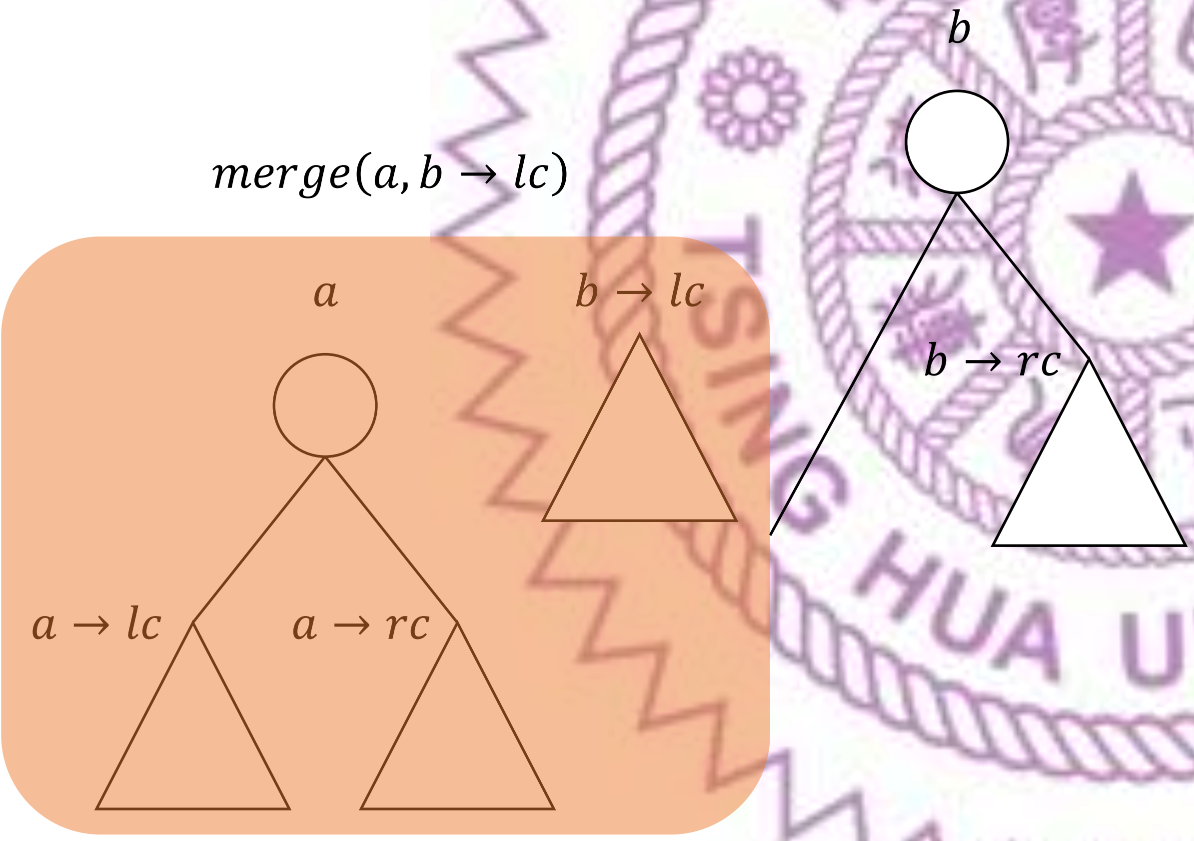



```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {
    a->rc = merge(a->rc, b);
    a->pull();
    return a;
  } else {




  }
}
```

誰要當根?

Case: $a \to pri \geq b \to pri$

反之亦然




```
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {
    a->rc = merge(a->rc, b);
    a->pull();
    return a;
  } else {
    b->lc = merge(a, b->lc);
    b->pull();
    return b;
  }
}
```

簡化遞迴想法

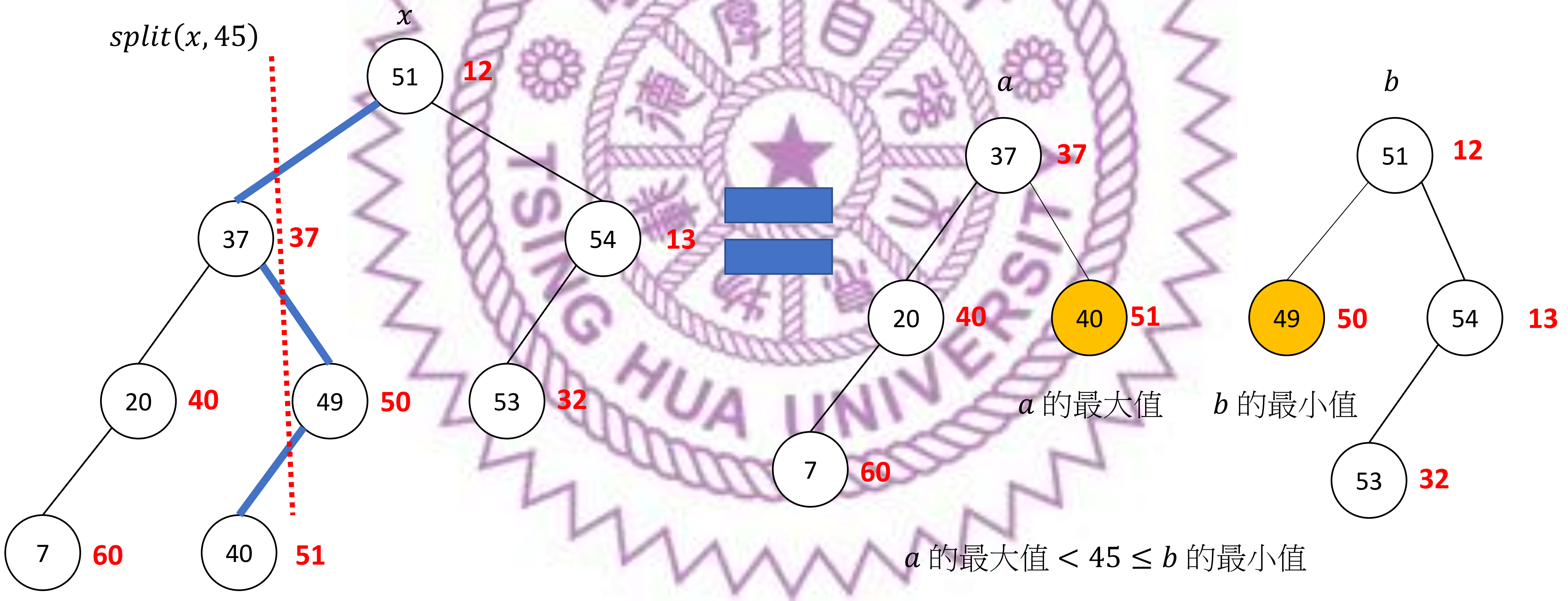
做完了嗎?

誰要當根?

哪邊還沒做完?

$split(x, Key)$

把 Treap x 切成 a, b 兩棵 Treap。其中 a 的最大值 $< Key$ ；b 的最小值 $\geq Key$



a 的最大值 $< 45 \leq b$ 的最小值

根要送給誰?

Case: x 是 $nullptr$

x

$nullptr$

a

b

```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;





  return {a, b};
}
```

根要送給誰?

Case: x 是 $nullptr$

顯然 a, b 都是 $nullptr$

x

$nullptr$

a

$nullptr$

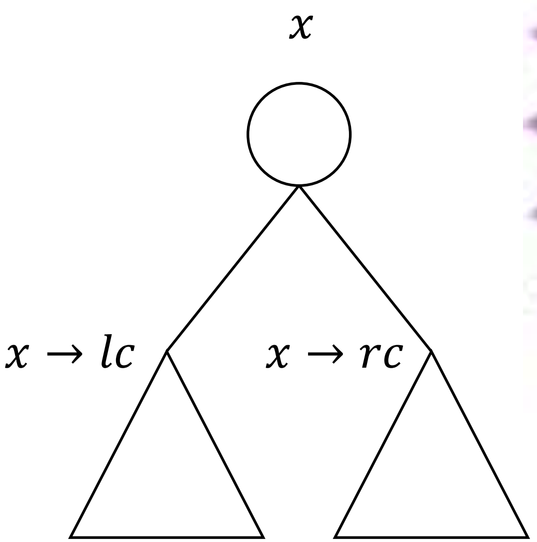
b

$nullptr$

```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};



  return {a, b};
}
```

根要送給誰?

Case: $x \to Key < Key$

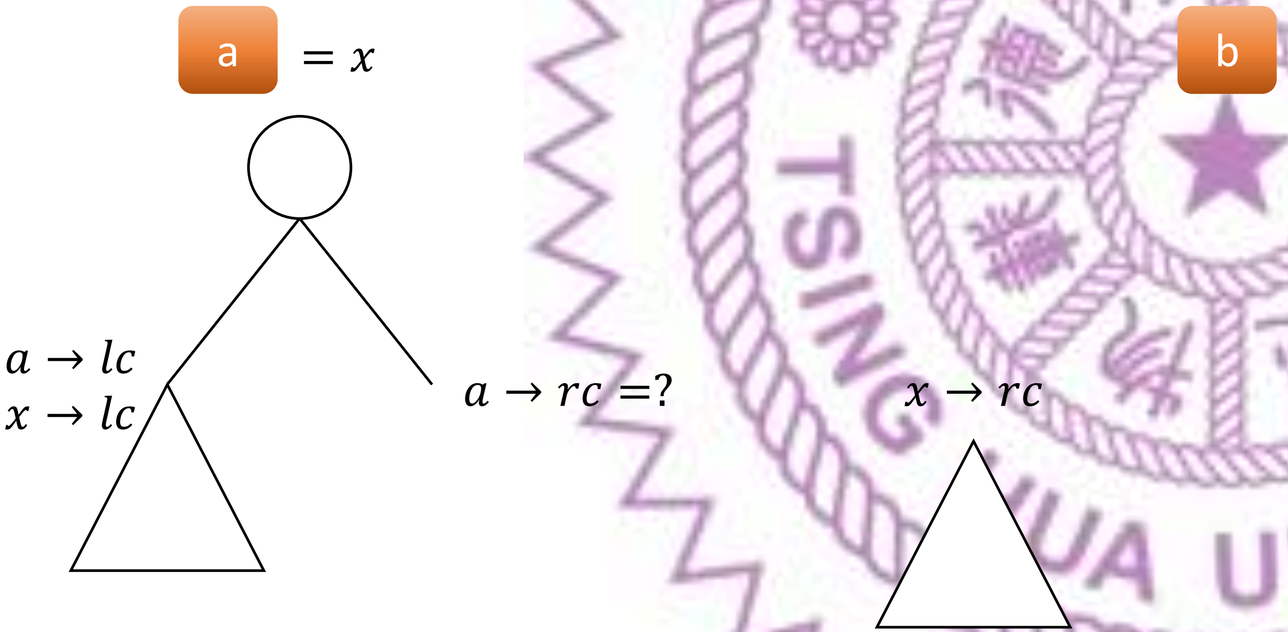



```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {









  return {a, b};
}
```

根要送給誰?

Case: $x \rightarrow Key < Key$

因為 a 的最大值 $< Key$，要把根送給 a



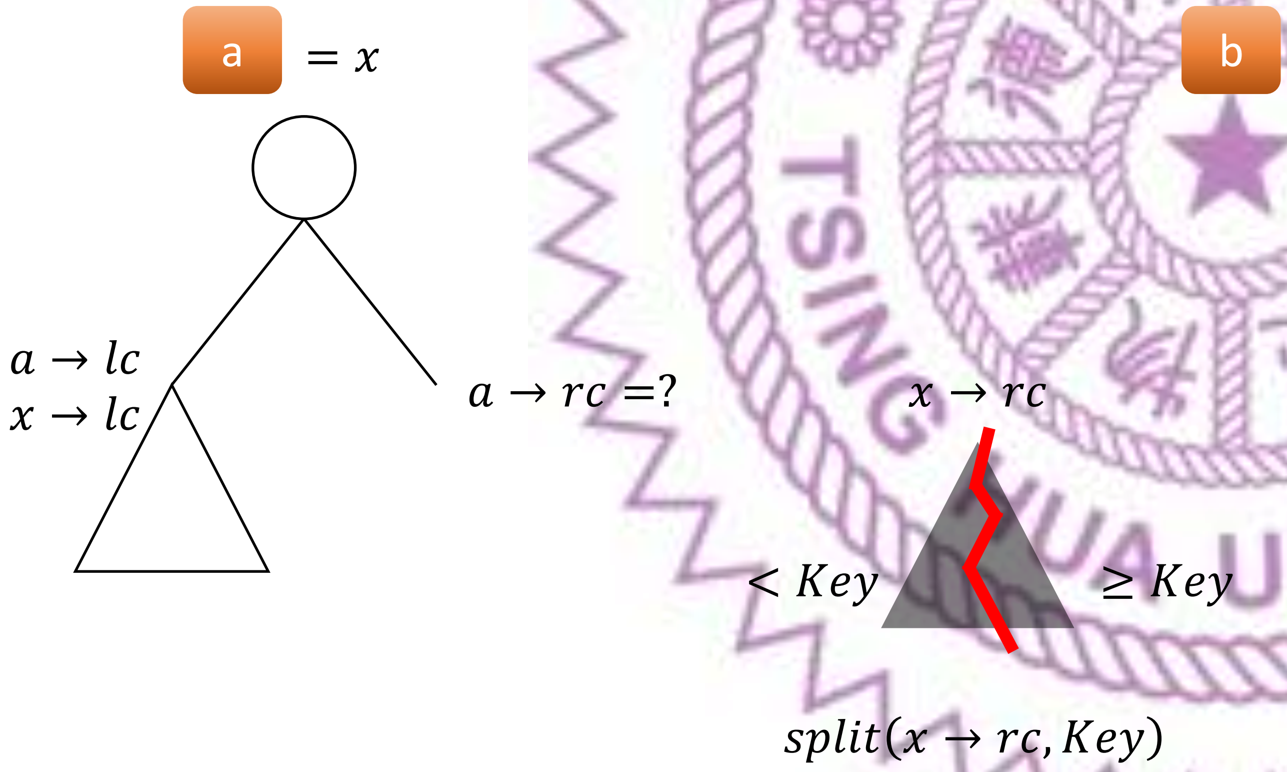

```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;









  return {a, b};
}
```

根要送給誰?

Case: $x \to Key < Key$

因為 a 的最大值 $< Key$，要把根送給 a

$a \to rc$ 和 b 就從 $x \to rc$ 繼續遞迴分割！



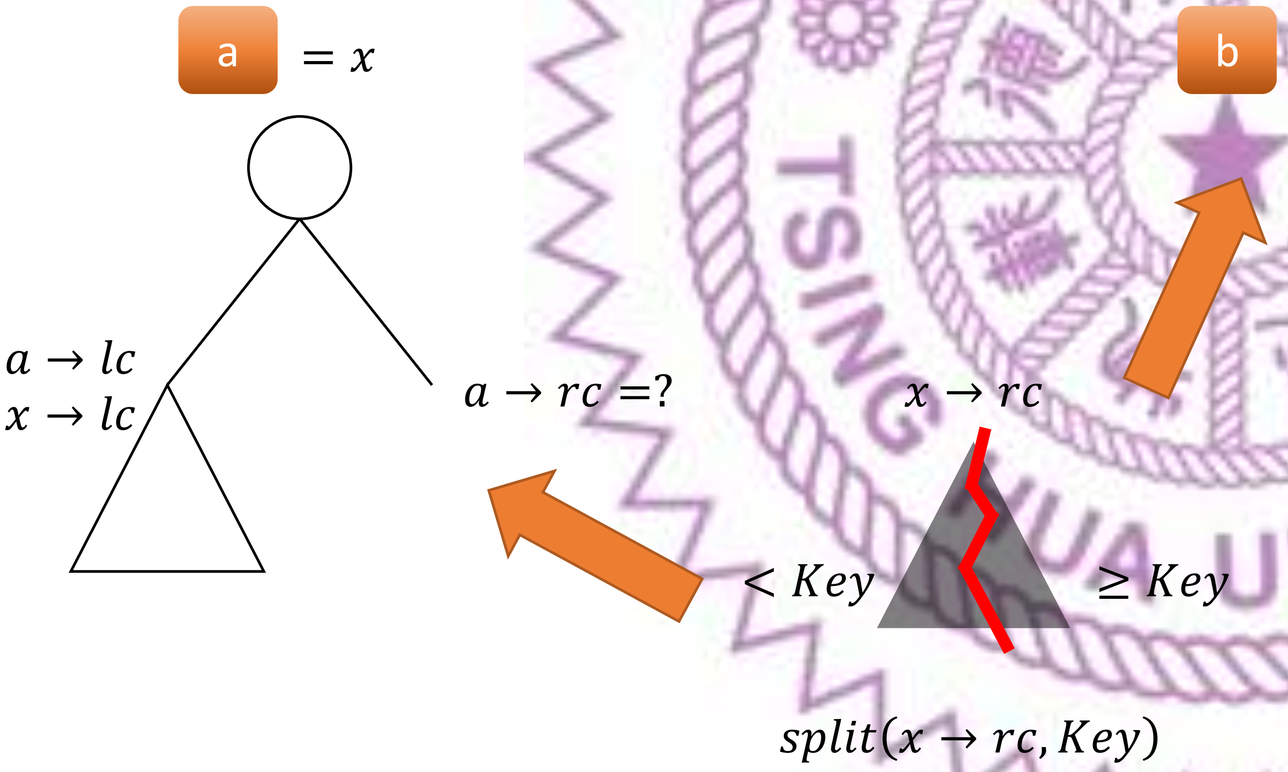

```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {



  return {a, b};
}
```

根要送給誰?

Case: $x \rightarrow Key < Key$

因為 a 的最大值 $< Key$，要把根送給 a

$a \rightarrow rc$ 和 b 就從 $x \rightarrow rc$ 繼續遞迴分割！



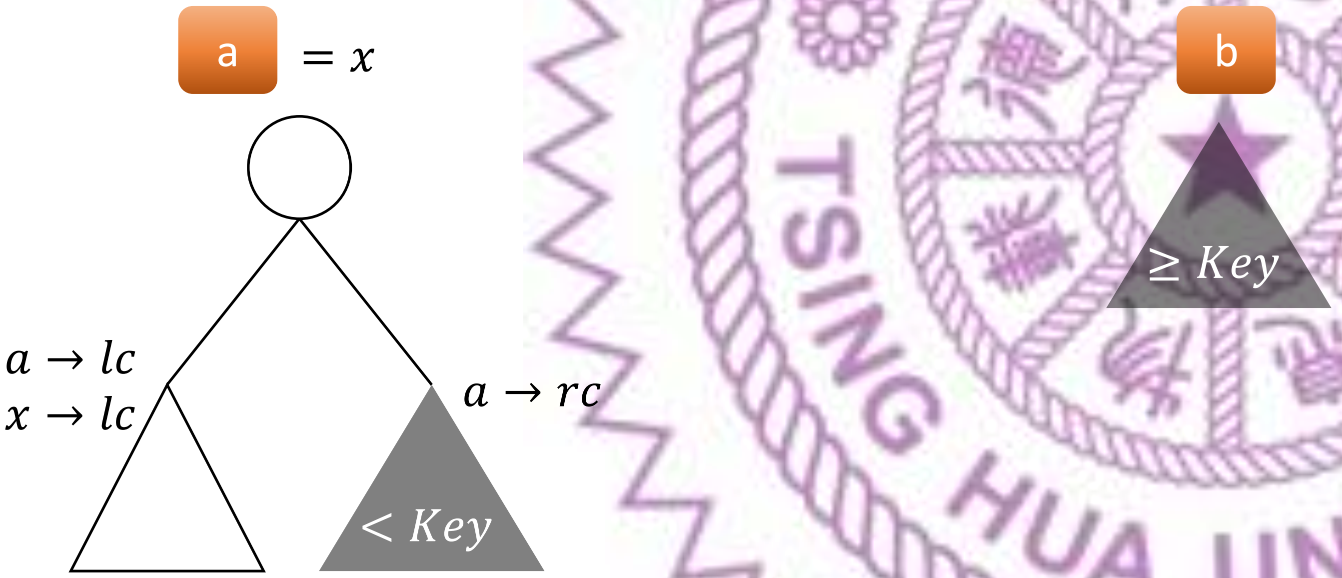

```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {



  return {a, b};
}
```

根要送給誰?

Case: $x \rightarrow Key < Key$

因為 a 的最大值 $< Key$，要把根送給 a

$a \rightarrow rc$ 和 b 就從 $x \rightarrow rc$ 繼續遞迴分割！

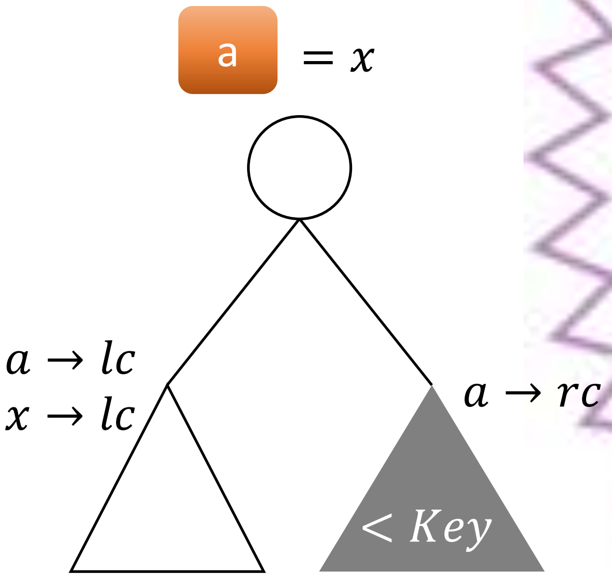



```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {



  return {a, b};
}
```

根要送給誰?

Case: $x \to Key < Key$

最後記得 pull

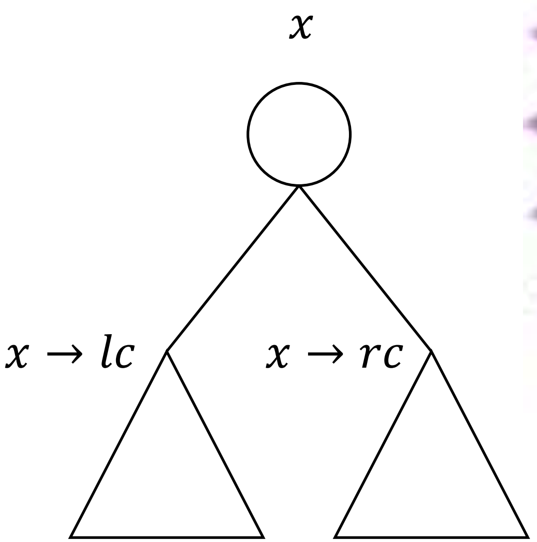



```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {


  }
  x->pull();
  return {a, b};
}
```

遞迴結束後所有點的資訊都是正確的
灰色部分的 size 早就算好了
因此只需要對 a 做 pull

根要送給誰?

Case: $x \rightarrow Key \geq Key$



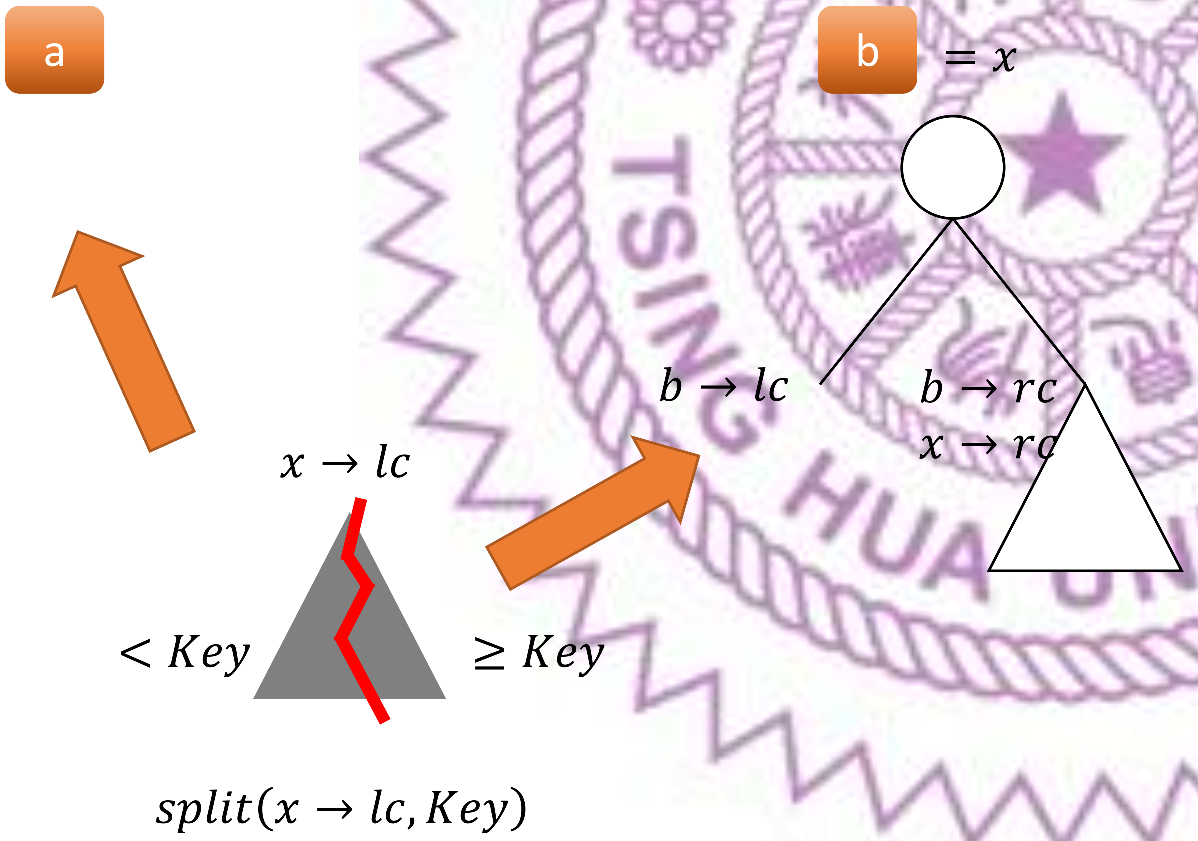
a b

```cpp
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {


  }
  x->pull();
  return {a, b};
}
```

根要送給誰?

Case: $x \to Key \geq Key$

反之亦然

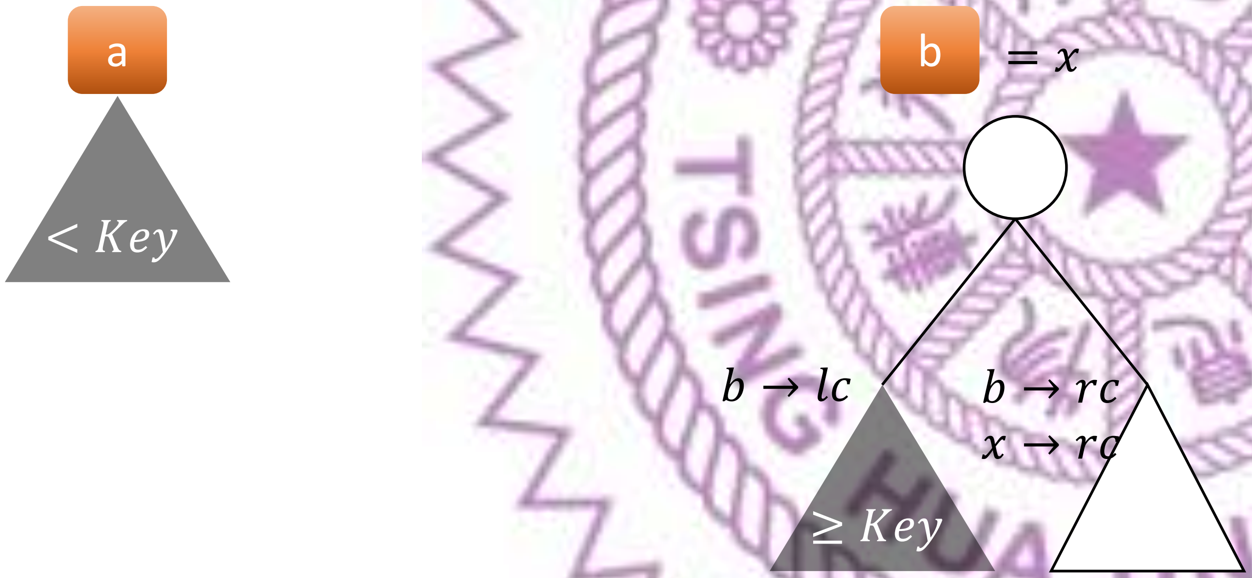



```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {
    b = x;
    tie(a, b->lc) = split(x->lc, Key);
  }
  x->pull();
  return {a, b};
}
```

根要送給誰?

Case: $x \rightarrow Key \geq Key$

反之亦然




```
pair<Treap *, Treap *>
split(Treap *x, int Key) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  if (x->Key < Key) {
    a = x;
    tie(a->rc, b) = split(x->rc, Key);
  } else {
    b = x;
    tie(a, b->lc) = split(x->lc, Key);
  }
  x->pull();
  return {a, b};
}
```

簡化遞迴想法

做完了嗎?

根要送給誰?

哪邊還沒做完?

平均時間複雜度

- $merge(a, b)$: $O\left(\log\left(size(a) + size(b)\right)\right)$
- $split(x, Key)$: $O\left(\log\left(size(x)\right)\right)$

名次樹五大操作

$insert(root, Key)$

- 在 $root$ 中加入值為 Key 的點

$find(root, Key)$

- 查找 $root$ 中是否有值為 Key 的點

$erase(root, Key)$

- 刪除 $root$ 中的一個值為 Key 的點

$Rank(root, Key)$

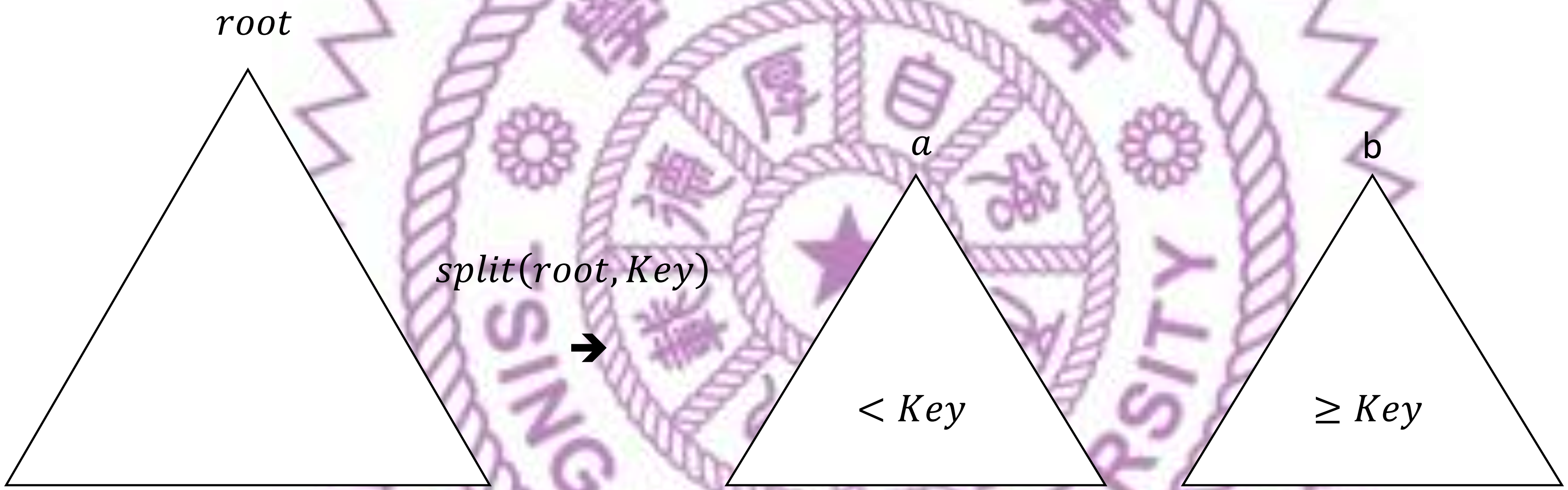
- 查找 $root$ 中值小於 Key 的點有幾個

$Kth(root, K)$

- 查找 $root$ 中值由小到大排名第 K 的點

只有它我們還不會做

$insert(root, Key)$



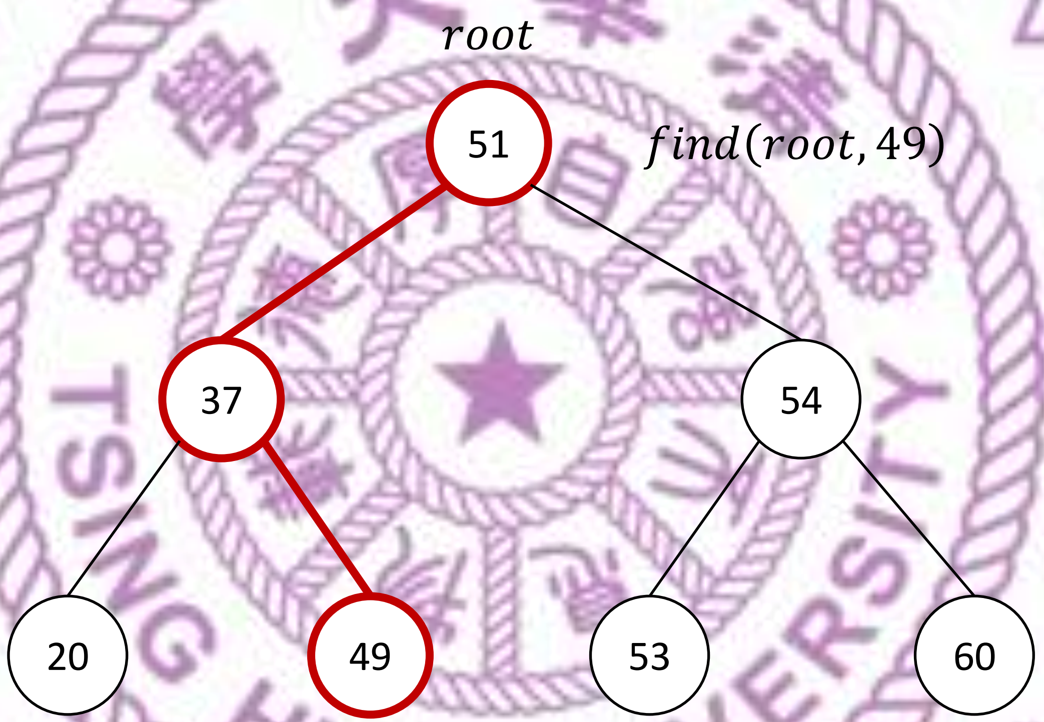

```cpp
void insert(Treap *&root, int Key) {
  auto [a, b] = split(root, Key);
  root = merge(a, merge(new Treap(Key), b));
}
```

$insert(root, Key)$

`root = merge(a,merge(new Treap(Key),b))`

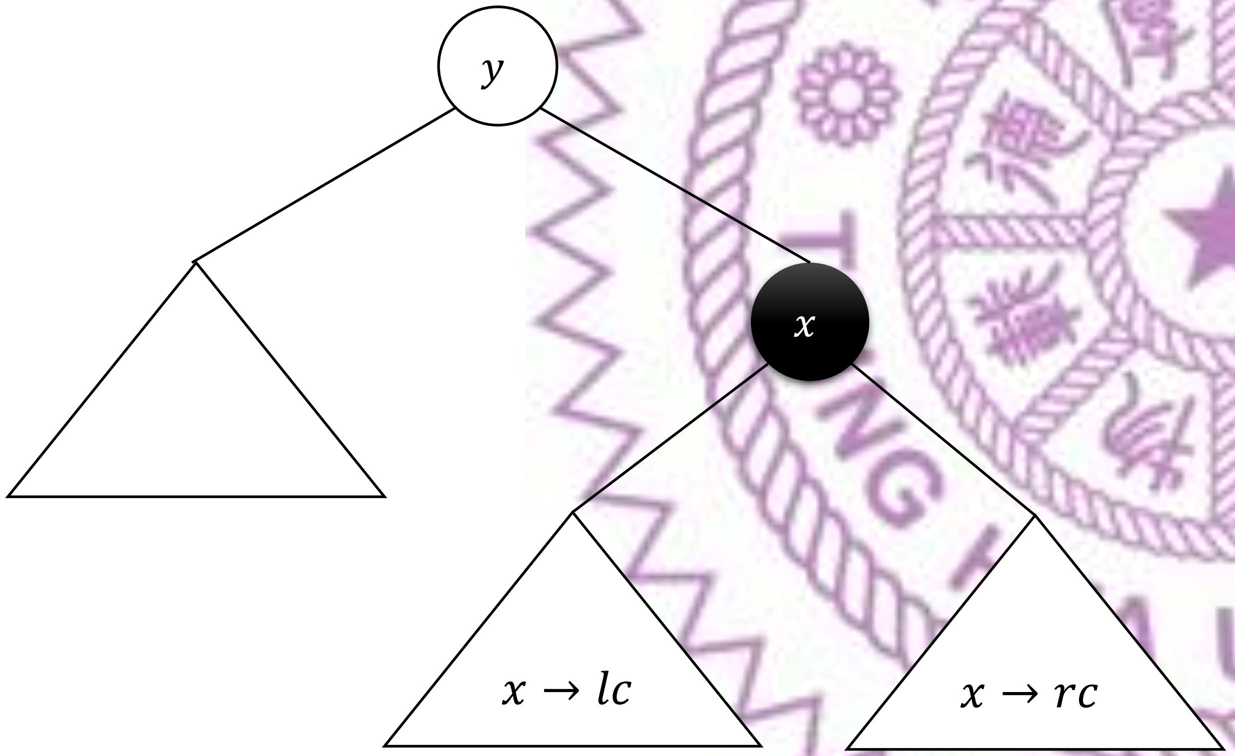

```cpp
void insert(Treap *&root, int Key) {
  auto [a, b] = split(root, Key);
  root = merge(a, merge(new Treap(Key), b));
}
```

$find(root, Key)$



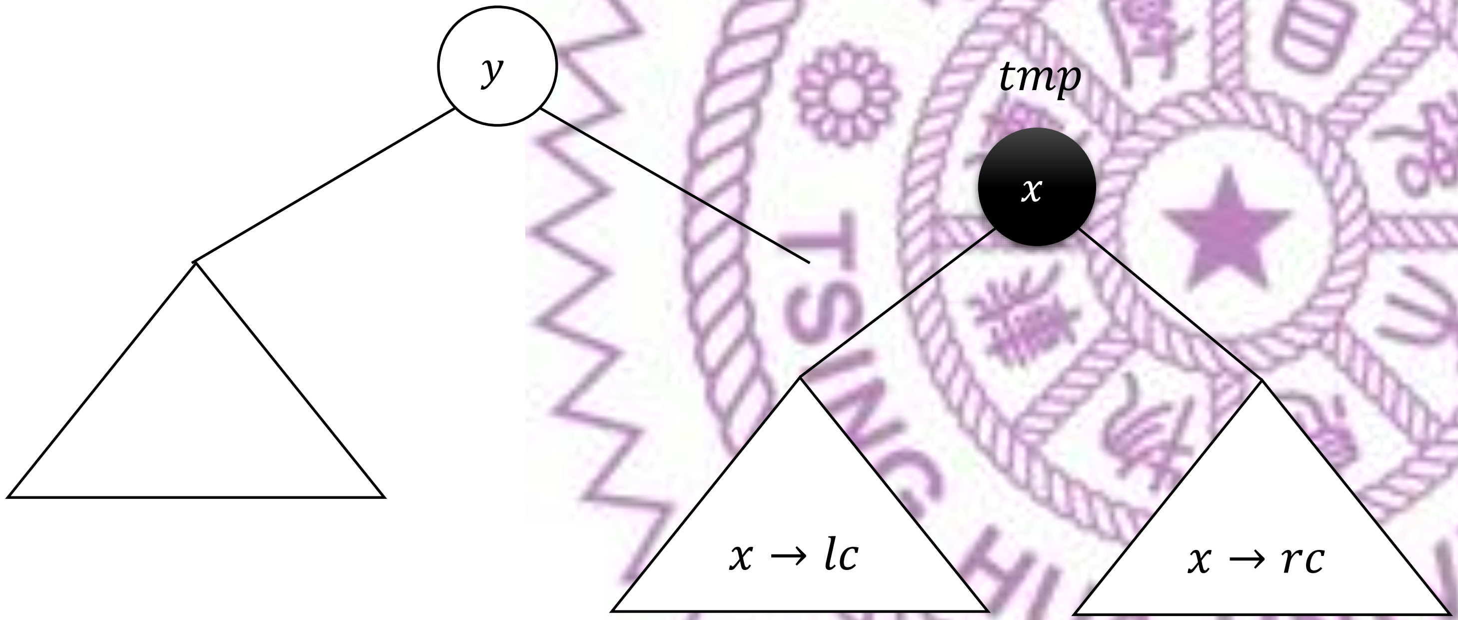

```
Treap *&find(Treap *&root, int Key) {
  if (!root || root->Key == Key) return root;
  return find(Key < root->Key ? root->lc : root->rc, Key);
}
```

$erase(root, Key)$



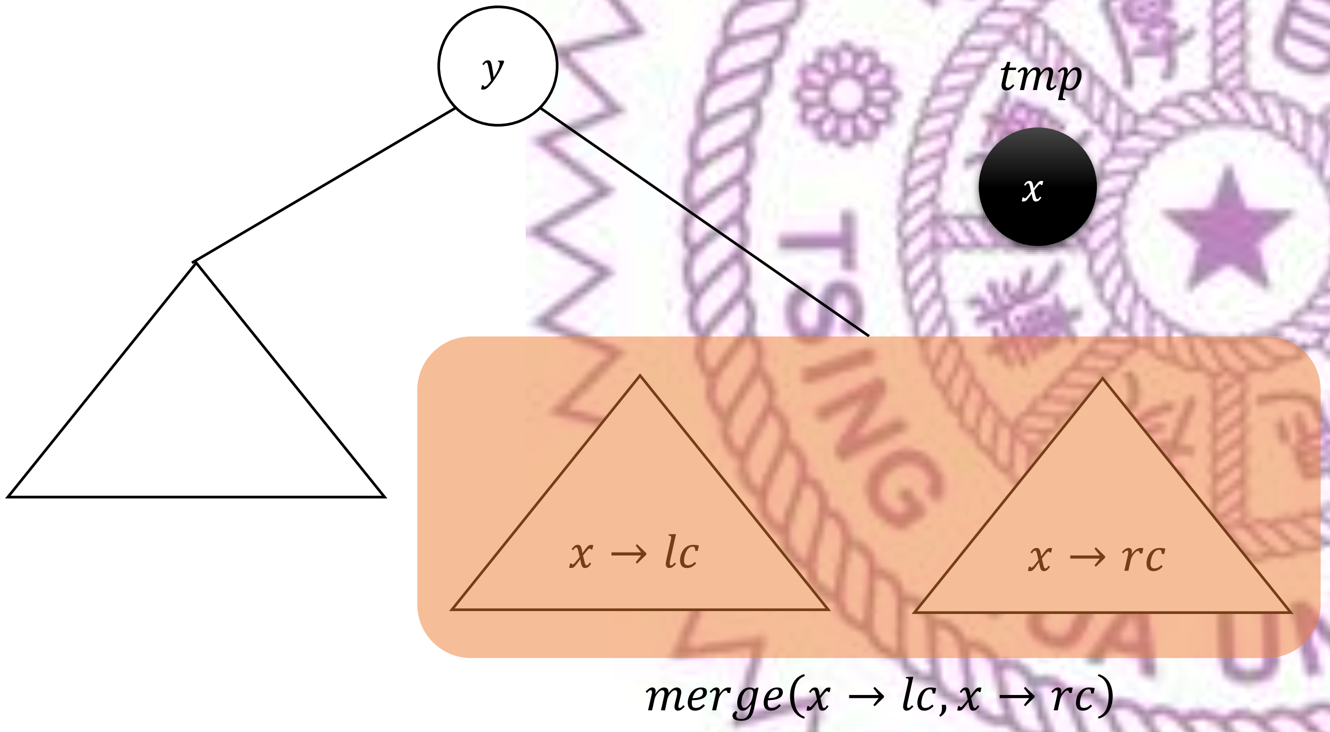

```
bool erase(Treap *&root, int Key) {
  Treap *&x = find(root, Key);
  if (!x) return false;
  Treap *tmp = x;
  x = merge(x->lc, x->rc);
  delete tmp;
  return true;
}
```

$erase(root, Key)$




```cpp
bool erase(Treap *&root, int Key) {
  Treap *&x = find(root, Key);
  if (!x) return false;
  Treap *tmp = x;
  x = merge(x->lc, x->rc);
  delete tmp;
  return true;
}
```

$erase(root, Key)$

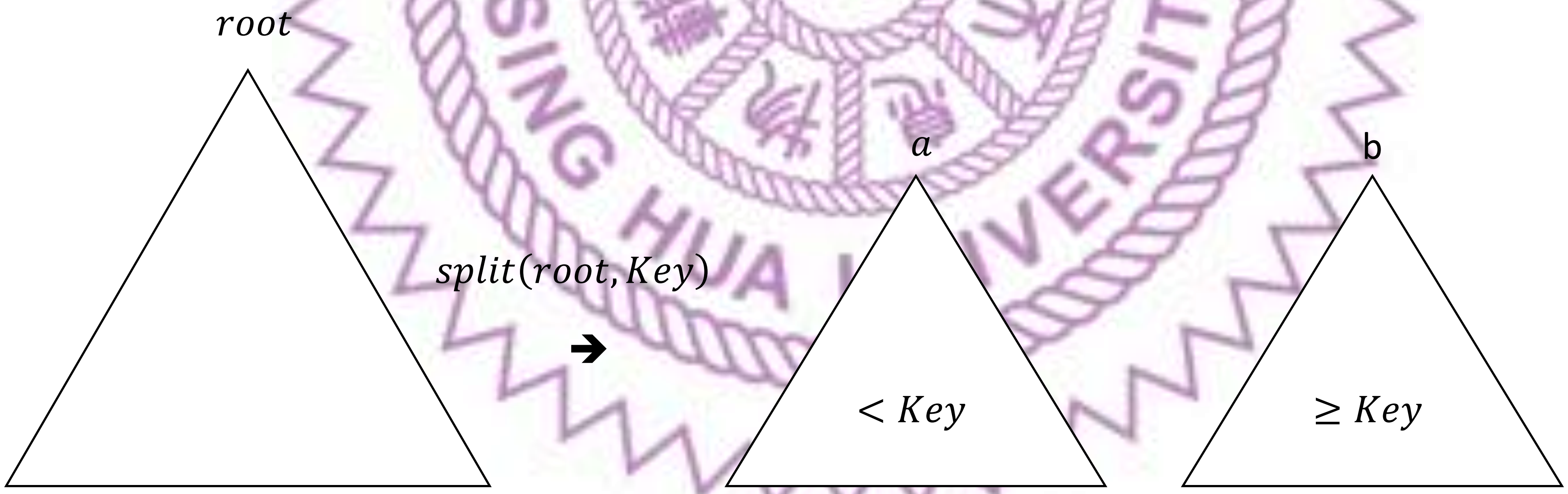



```
bool erase(Treap *&root, int Key) {
  Treap *&x = find(root, Key);
  if (!x) return false;
  Treap *tmp = x;
  x = merge(x->lc, x->rc);
  delete tmp;
  return true;
}
```

$Rank(root, Key)$

Treap a 包含了所有 $< Key$ 的點，因此 $size(a)$ 就是答案

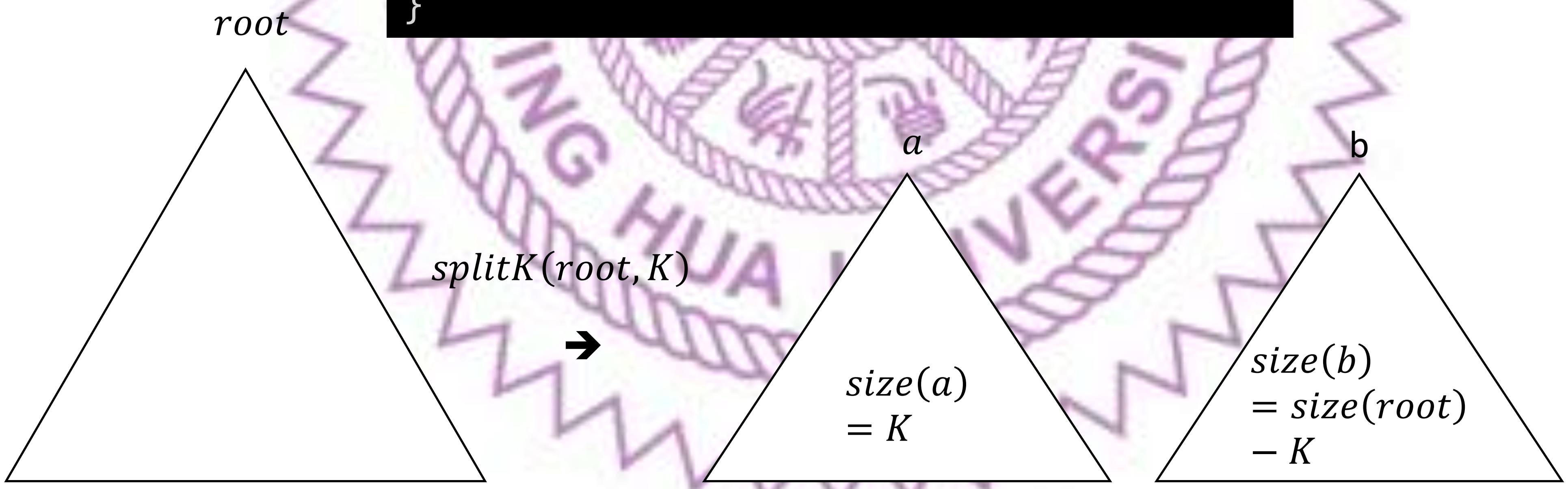
```cpp
unsigned Rank(Treap *&root, int Key) {
  auto [a, b] = split(root, Key);
  unsigned ans = size(a);
  root = merge(a, b);
  return ans;
}
```



$Kth(root, Key)$

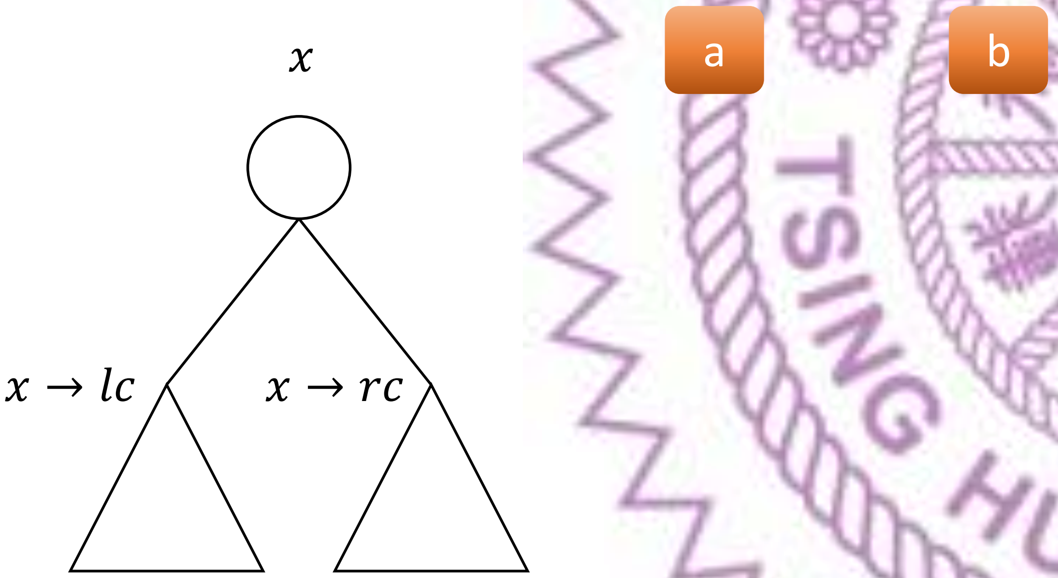
假設有一個 $[a, b] = splitK(x, K)$ 可以剛好使得 $\text{size}(a) = K$

```cpp
int Kth(Treap *&root, unsigned K) {
  auto [a, b] = splitK(root, K);
  auto [c, d] = splitK(a, K - 1);
  int ans = d->Key;
  root = merge(merge(c, d), b);
  return ans;
}
```



$splitK(x, K)$

Case: $K \geq size(x \rightarrow lc) + 1$

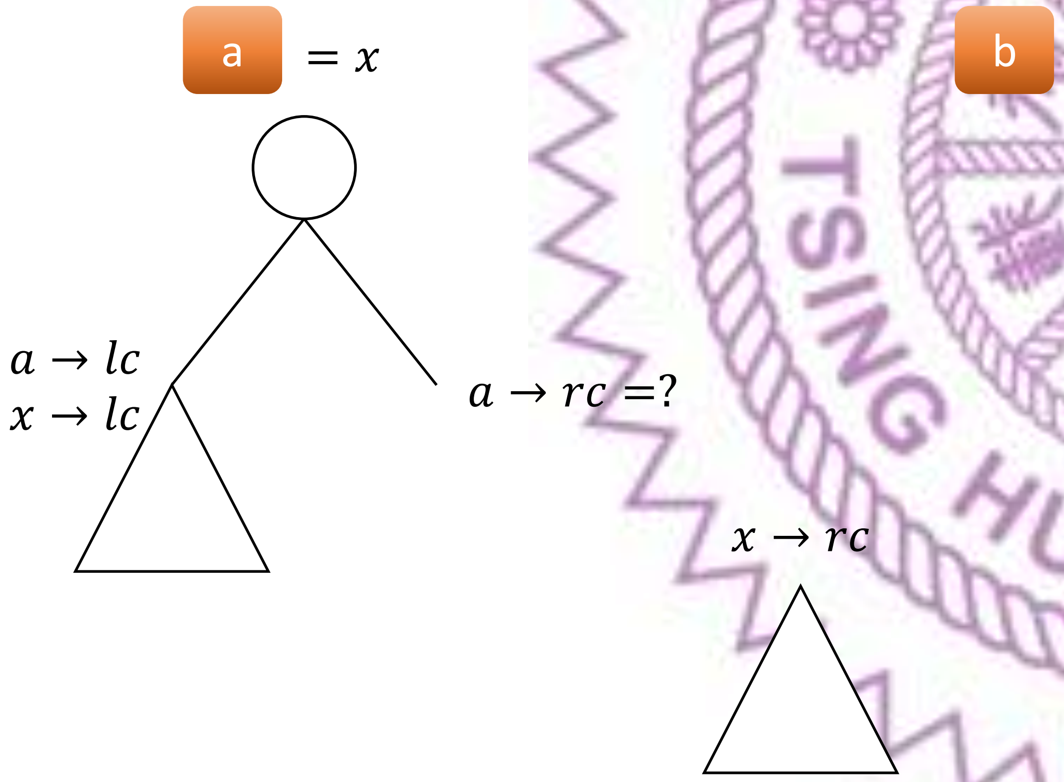



```
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {






  return {a, b};
}
```

$splitK(x, K)$

Case: $K \geq size(x \rightarrow lc) + 1$

x 和 $x \rightarrow lc$ 都要在 a 那邊




```
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {
    a = x;





  return {a, b};
}
```

$splitK(x,K)$

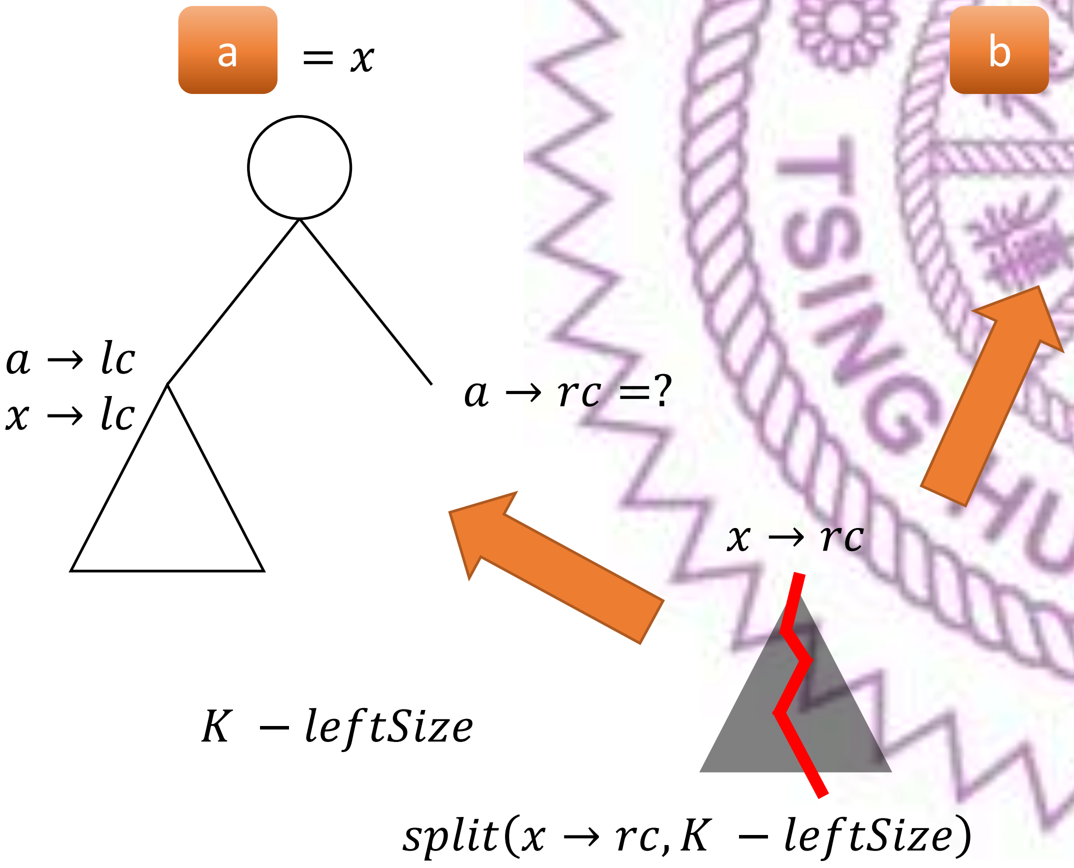
Case: $K \geq size(x \to lc) + 1$

x 和 $x \to lc$ 都要在 a 那邊

從 $x \to rc$ 切出 $K - leftSize$ 個點給 a

設 $leftSize = size(x \to lc) + 1$

```
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {
    a = x;
    tie(a->rc, b) = splitK(x->rc, K - leftSize);



  return {a, b};
}
```

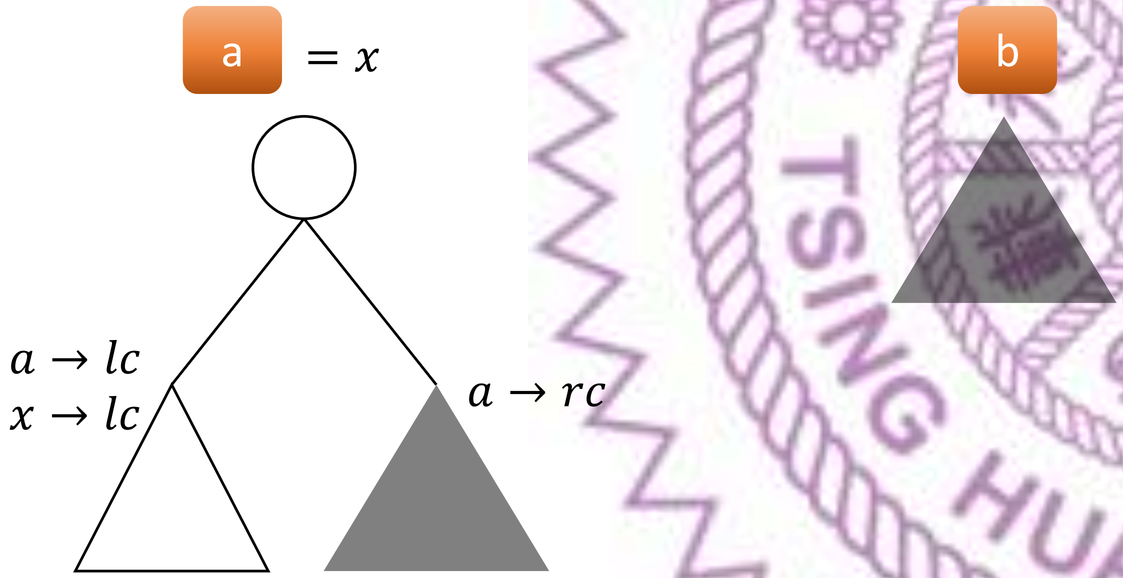


$splitK(x, K)$

Case: $K \geq size(x \to lc) + 1$

x 和 $x \to lc$ 都要在 a 那邊

遞迴結束後得到 $size(a) = K$



$size(a \to rc) = K - leftSize$

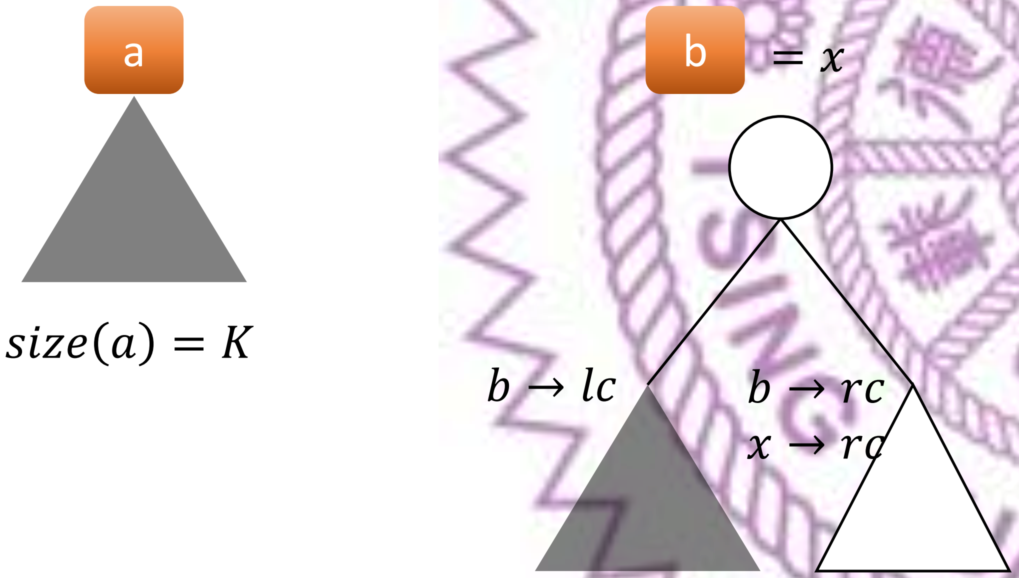
設 $leftSize = size(x \to lc) + 1$

```
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {
    a = x;
    tie(a->rc, b) = splitK(x->rc, K - leftSize);
  } else {


  }
  x->pull();
  return {a, b};
}
```

$splitK(x, K)$

Case:$K \leq size(x \to lc)$

反之亦然




```
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {
    a = x;
    tie(a->rc, b) = splitK(x->rc, K - leftSize);
  } else {
    b = x;
    tie(a, b->lc) = splitK(x->lc, K);
  }
  x->pull();
  return {a, b};
}
```

Treap 與區間操作

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
- $update(ql, qr)$:
  反轉區間 $[ql, qr]$
- $1 \leq n, q \leq 10^6$

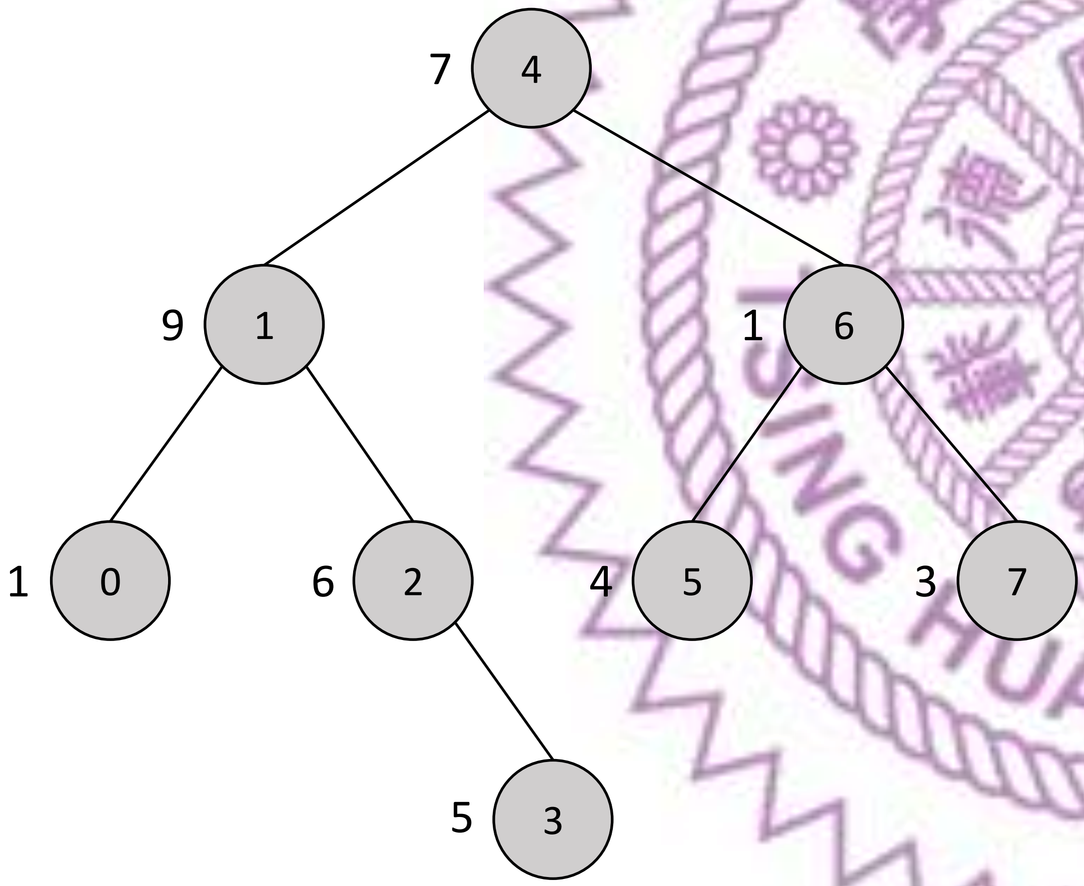
線段樹能做嗎?

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

$revert(1, 4)$

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

節點 Key 值存陣列 index

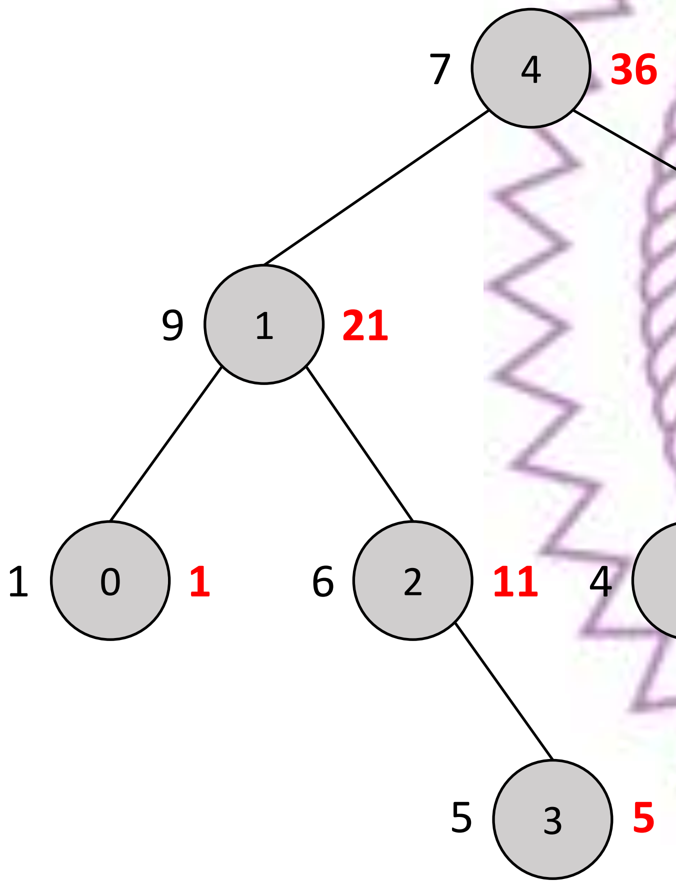


a

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

所有的點的 Key 依然滿足二元搜尋樹性質

每個點紀錄自己底下的總和

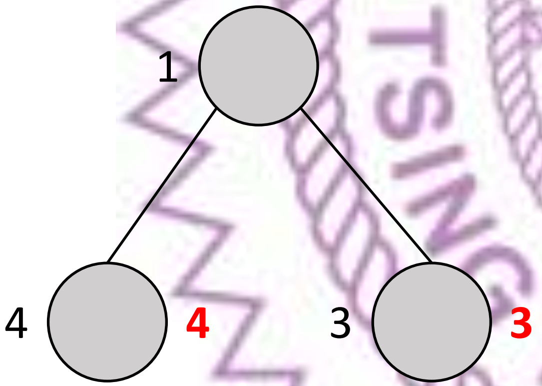


	0	1	2	3	4	5	6	7
a	1	9	6	5	7	4	1	3

所有的點的 Key 依然滿足二元搜尋樹性質
每個點多記錄一個 Val 值存 $a[Key]$ 的值

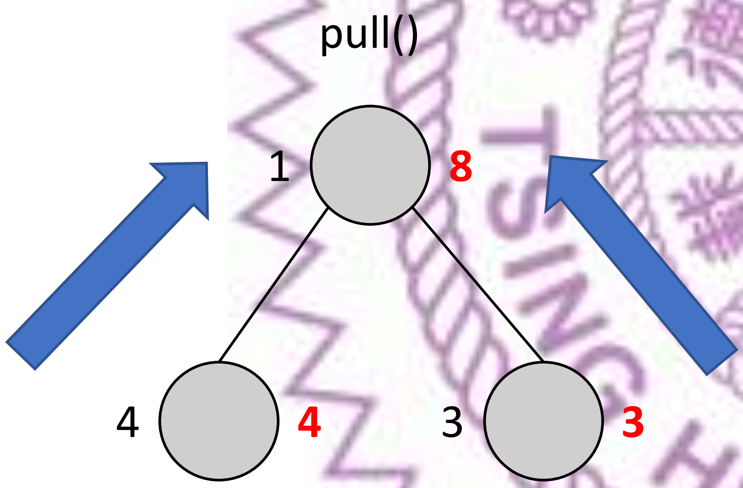
```
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  unsigned pri, size;
  int Key;
  long long Val, Sum;
  Treap(int Key, int Val):
    pri(rand()), size(1),
    Key(Key), Val(Val), Sum(Val) {}
  void pull();
};
```

pull(): 計算 size 和 Sum




```
void Treap::pull() {
  size = 1;
  Sum = Val;
  if (lc) {
    size += lc->size;
    Sum += lc->Sum;
  }
  if (rc) {
    size += rc->size;
    Sum += rc->Sum;
  }
}
```

pull(): 計算 size 和 Sum




```cpp
void Treap::pull() {
  size = 1;
  Sum = Val;
  if (lc) {
    size += lc->size;
    Sum += lc->Sum;
  }
  if (rc) {
    size += rc->size;
    Sum += rc->Sum;
  }
}
```

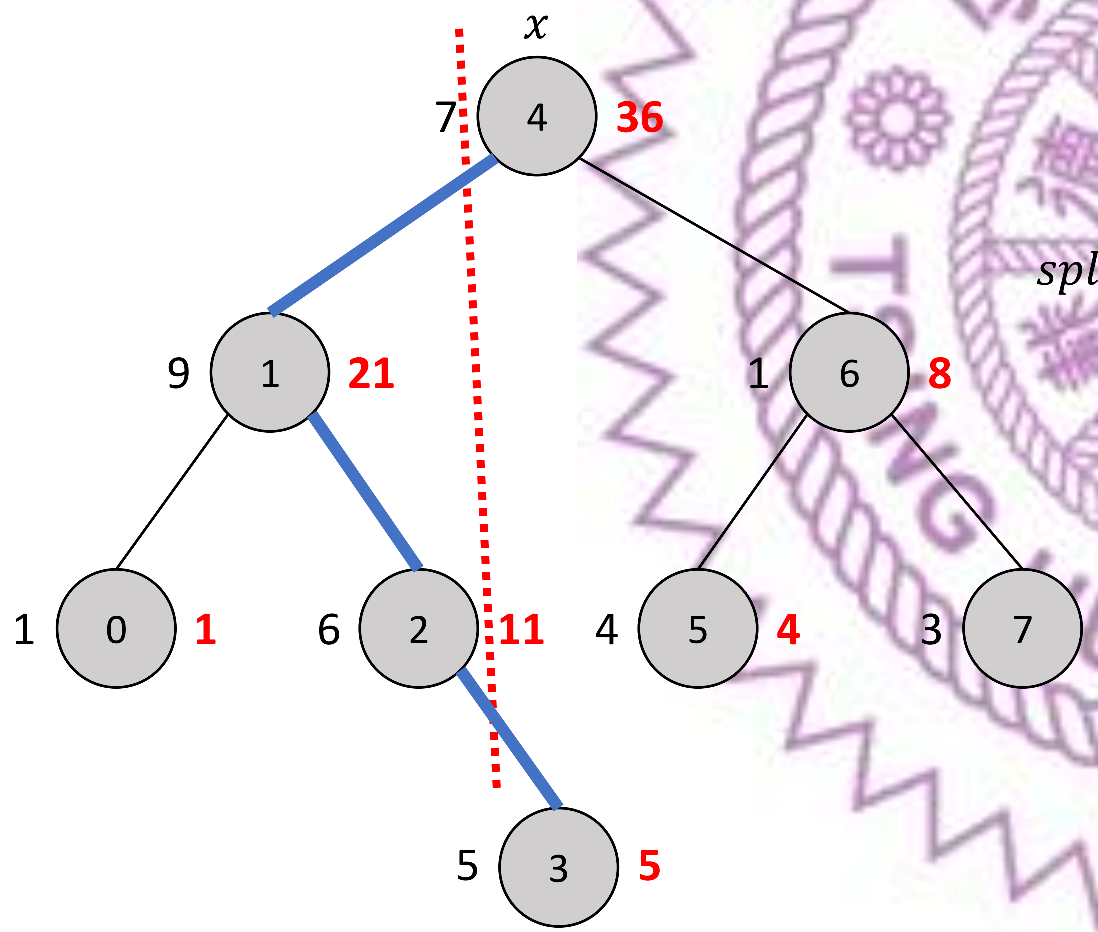
將陣列構造成 Treap

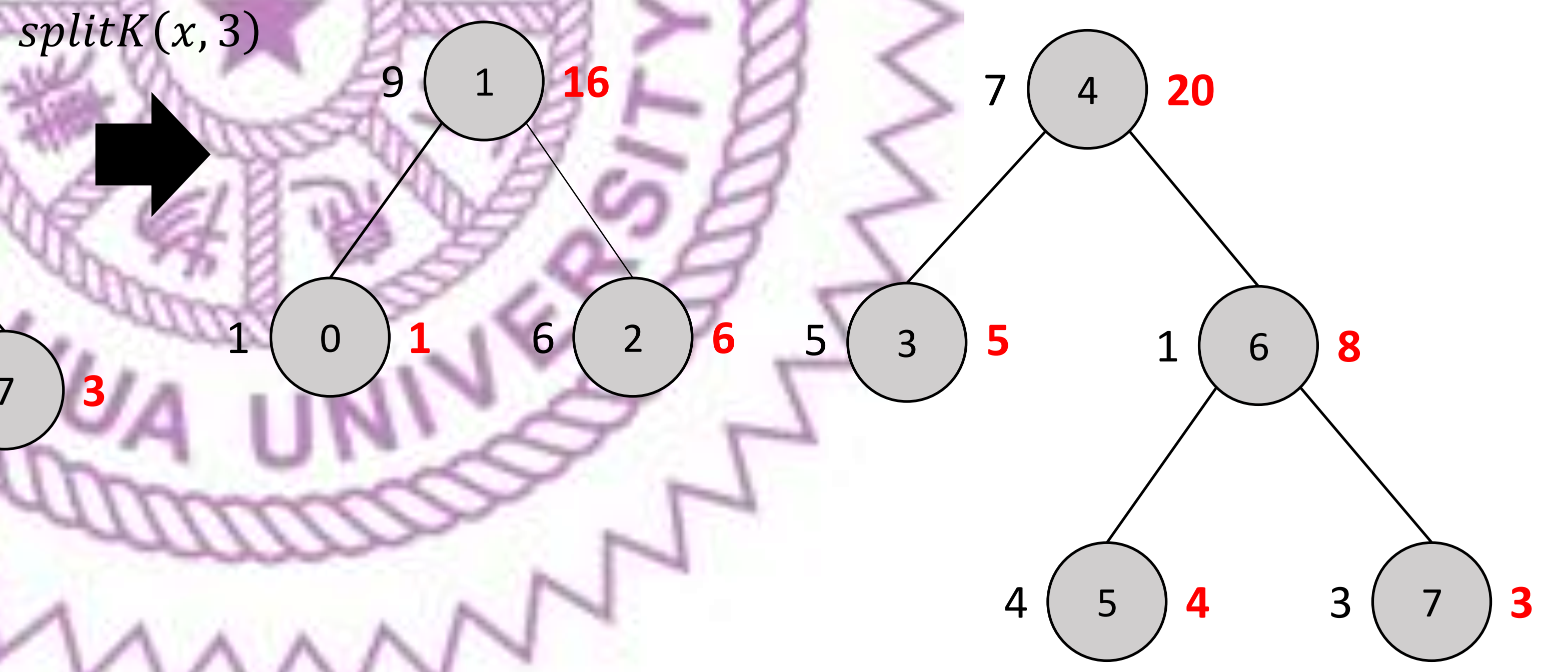
```
Treap *init(const vector<int> &a) {
  Treap *root = nullptr;
  for (size_t i = 0; i < a.size(); ++i) {
    root = merge(root, new Treap(i, a[i]));
  }
  return root;
}
```

觀察 $splitK(x, K)$

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

$a[0] + a[1] + a[2]$ $a[3] + a[4] + a[5] + a[6] + a[7]$





查詢只需要把想要的區域切出來問

```cpp
long long query(Treap *&root, unsigned ql, unsigned qr) {
  auto [a, b] = splitK(root, ql);
  auto [c, d] = splitK(b, qr - ql + 1);
  long long Sum = c->Sum;
  root = merge(a, merge(c, d));
  return Sum;
}
```

Key 值沒有任何函數用到?

- 沒錯！splitK、merge 本身就維護了節點順序，根本不用紀錄

```cpp
Treap *init(const vector<int> &a) {
  Treap *root = nullptr;
  for (size_t i = 0; i < a.size(); ++i) {
    root = merge(root, new Treap(i, a[i]));
  }
  return root;
}
```

```cpp
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  unsigned pri, size;
  int Key;
  long long Val, Sum;
  Treap(int Key, int Val):
    pri(rand()), size(1),
    Key(Key), Val(Val), Sum(Val) {}
  void pull();
};
```

Key 值沒有任何函數用到?

- 沒錯！splitK、merge 本身就維護了節點順序，根本不用紀錄

```
Treap *init(const vector<int> &a) {
  Treap *root = nullptr;
  for (size_t i = 0; i < a.size(); ++i) {
    root = merge(root, new Treap(a[i]));
  }
  return root;
}
```

```
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  unsigned pri, size;
  long long Val, Sum;
  Treap(int Val):
    pri(rand()), size(1),
    Val(Val), Sum(Val) {}
  void pull();
};
```

少了 *Key* 值就可以直接反轉區間，但很慢

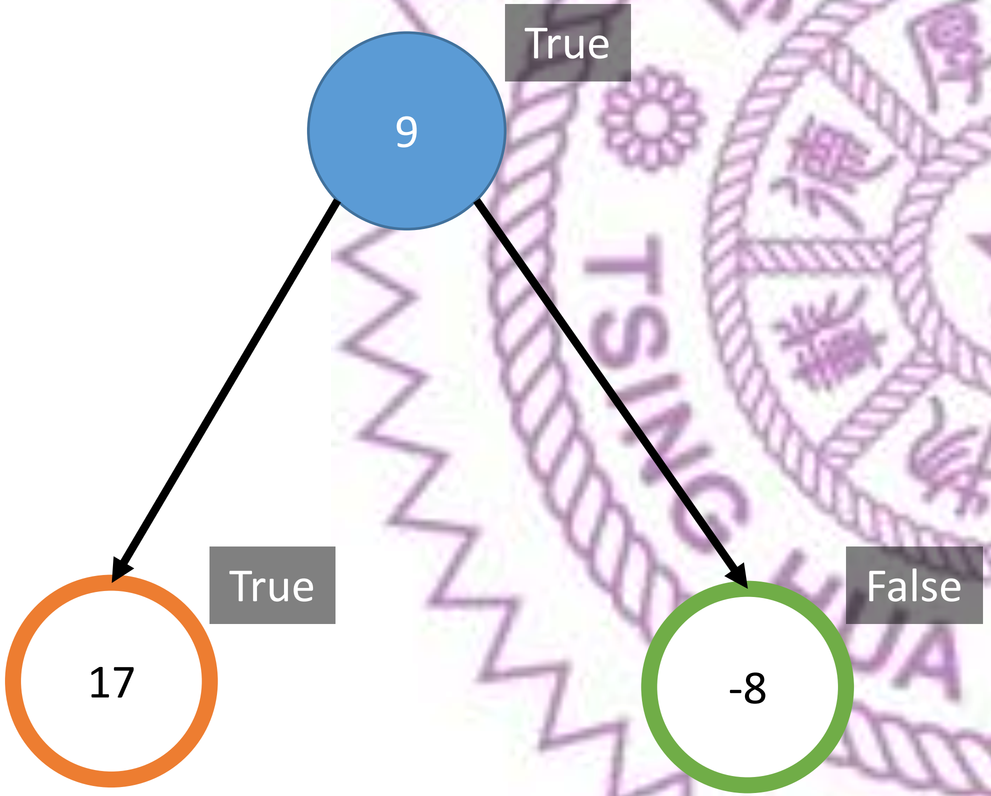
```
void reverse(Treap *root) {
  if (!root) return;
  swap(root->lc, root->rc);
  reverse(root->lc);
  reverse(root->rc);
}

void update(Treap *&root, unsigned ql, unsigned qr) {
  auto [a, b] = splitK(root, ql);
  auto [c, d] = splitK(b, qr - ql + 1);
  reverse(c);
  root = merge(a, merge(c, d));
}
```

懶惰標記

- 同線段樹，我們也可以在 Treap 上設懶惰標記
- Tag = false:
  正常節點
- Tag = true:
  該 Treap 要被反轉
  但**還沒做**

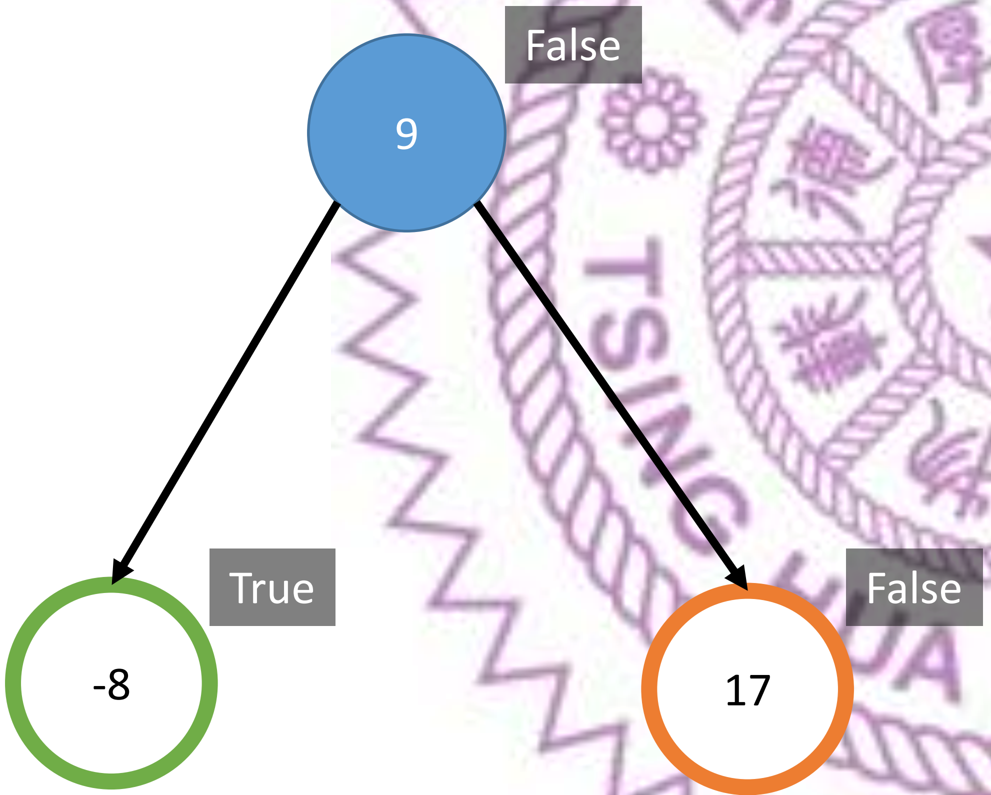
```
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  unsigned pri, size;
  long long Val, Sum;
  bool Tag;
  Treap(int Val):
    pri(rand()), size(1),
    Val(Val), Sum(Val), Tag(false) {}
  void pull();
  void push();
};
```

懶惰標記下推




```
void Treap::push() {
  if (!Tag) return;
  swap(lc, rc);
  if (lc) lc->Tag ^= Tag;
  if (rc) rc->Tag ^= Tag;
  Tag = false;
}
```

懶惰標記下推




```
void Treap::push() {
  if (!Tag) return;
  swap(lc, rc);
  if (lc) lc->Tag ^= Tag;
  if (rc) rc->Tag ^= Tag;
  Tag = false;
}
```

把 push 加在正確的位置

樹的結構改變前(遞迴前)呼叫

```cpp
Treap *merge(Treap *a, Treap *b) {
  if (!a || !b) return a ? a : b;
  if (a->pri < b->pri) {
    a->push();
    a->rc = merge(a->rc, b);
    a->pull();
    return a;
  } else {
    b->push();
    b->lc = merge(a, b->lc);
    b->pull();
    return b;
  }
}
```

```cpp
pair<Treap *, Treap *>
splitK(Treap *x, unsigned K) {
  Treap *a = nullptr, *b = nullptr;
  if (!x) return {a, b};
  x->push();
  unsigned leftSize = size(x->lc) + 1;
  if (K >= leftSize) {
    a = x;
    tie(a->rc, b) = splitK(x->rc, K - leftSize);
  } else {
    b = x;
    tie(a, b->lc) = splitK(x->lc, K);
  }
  x->pull();
  return {a, b};
}
```

修改與查詢

```cpp
void update(Treap *&root, unsigned ql, unsigned qr) {
  auto [a, b] = splitK(root, ql);
  auto [c, d] = splitK(b, qr - ql + 1);
  c->Tag ^= true;
  root = merge(a, merge(c, d));
}
```

雖然這題不加這行沒差
但有些懶惰標記的存在會影響答案
建議取得任何資訊前先做 push

```cpp
long long query(Treap *&root, unsigned ql, unsigned qr) {
  auto [a, b] = splitK(root, ql);
  auto [c, d] = splitK(b, qr - ql + 1);
  c->push();
  long long Sum = c->Sum;
  root = merge(a, merge(c, d));
  return Sum;
}
```