

【第十一講】

樹

講師: 李根逸 (Ken-Yi Lee), E-mail: feis.tw@gmail.com

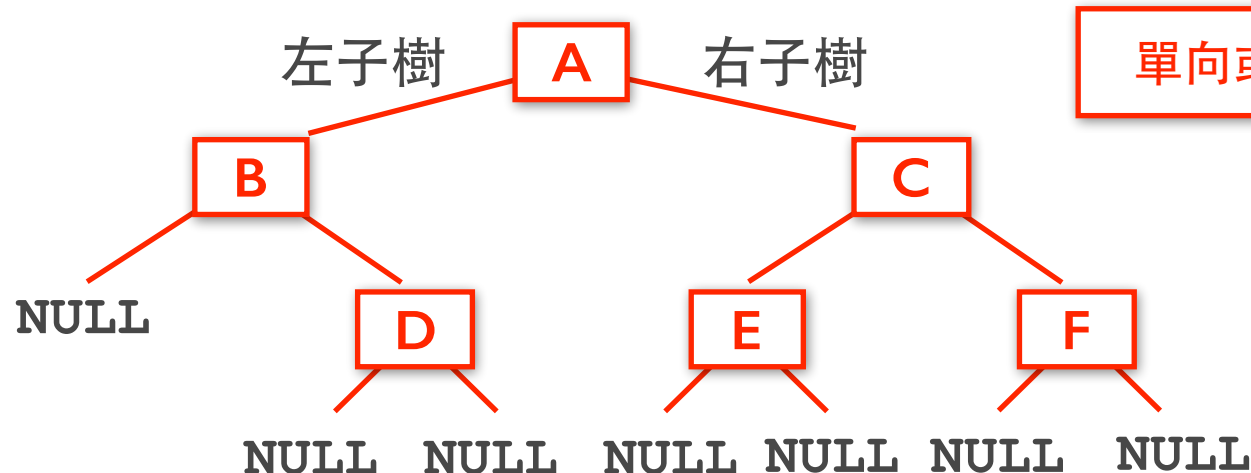


課程大綱

- 二元樹 (**binary tree**)
- 堆積 (**heap**)
- 使用 **STL** 的堆積
- 二元搜尋樹 (**binary search tree**)
 - ▶ 二元搜尋樹的刪除
- 使用 **STL** 的集合與映射 (**std::set, std::map**)

二元樹

- 樹 (**tree**) 由節點構成，在樹中任兩個節點之間不重複經過的路徑只有一條
 - ▶ 大部分節點都有一個親節點，唯一沒有親節點的我們稱為樹根 (**root**)
 - ▶ 沒有子節點的節點稱為樹葉 (**leaf**)
- 二元樹指的是每個節點最多可以有兩個子節點



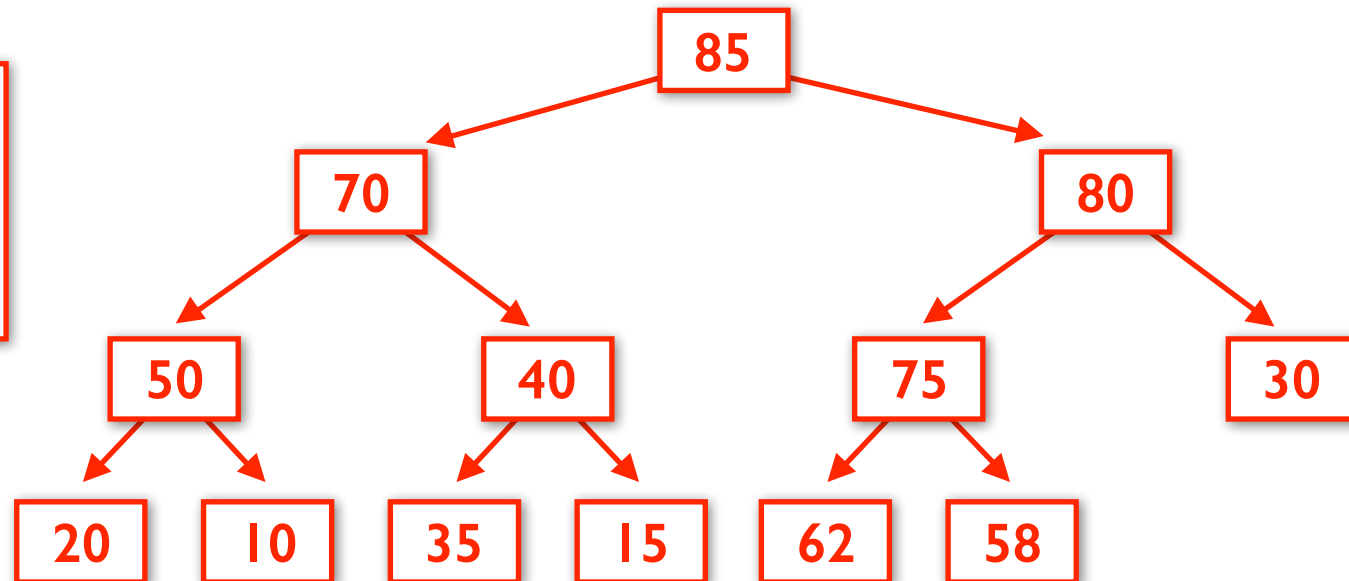
堆積

- 堆積 (**heap**) 裡，第 **k** 號元素要“大於等於”第 **2k+1** 與 **2k+2** 號元素

- ▶ 上面這稱為 max-heap；“小於等於”則稱為 min-heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

- ▶ 概念上可對應到一個完整二元樹：

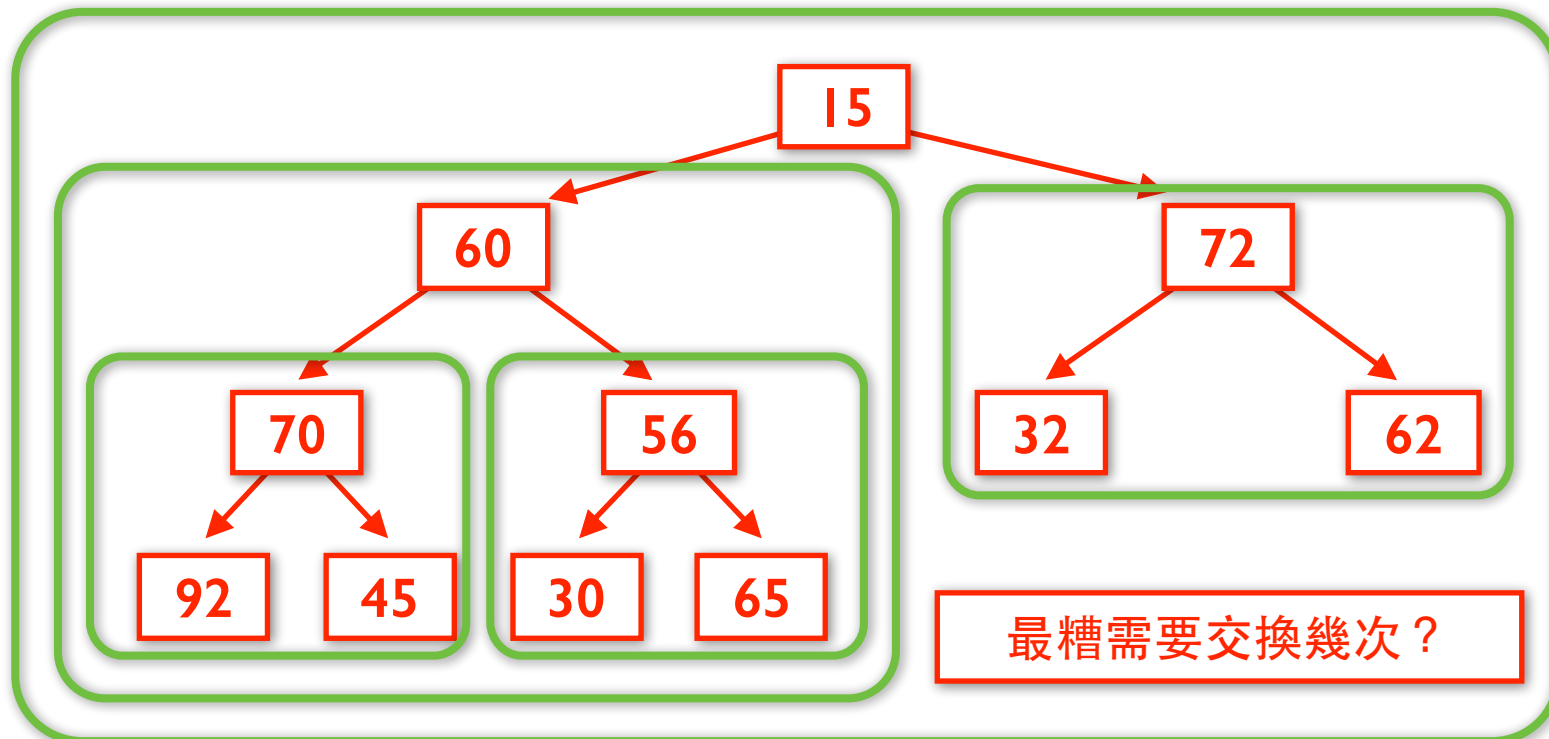


如果有 **N** 個元素的話，這個樹的高度 (從 root 走到 leaf) 會是多少？

【範例】 建立堆積

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
15	60	72	70	56	32	62	92	45	30	65

參考動畫：<http://www.cs.usfca.edu/~galles/visualization/Heap.html>



由下往上執行

上面的要比下面的大

[範例] `heapify.cpp`

【範例】使用堆積排序

- 建立堆積
- 放置在堆積的樹根（第一個）的值，為堆積中值最大的。我們可以利用這點來做排序：
 - ▶ 重複執行以上步驟直到堆積大小為零：
 - 將堆積中第一個與堆積中最後一個交換
 - 將堆積的大小少一
 - 此時第一個不一定是剩下的堆積中最大的，所以由第一個開始重新整理堆積（由上往下）

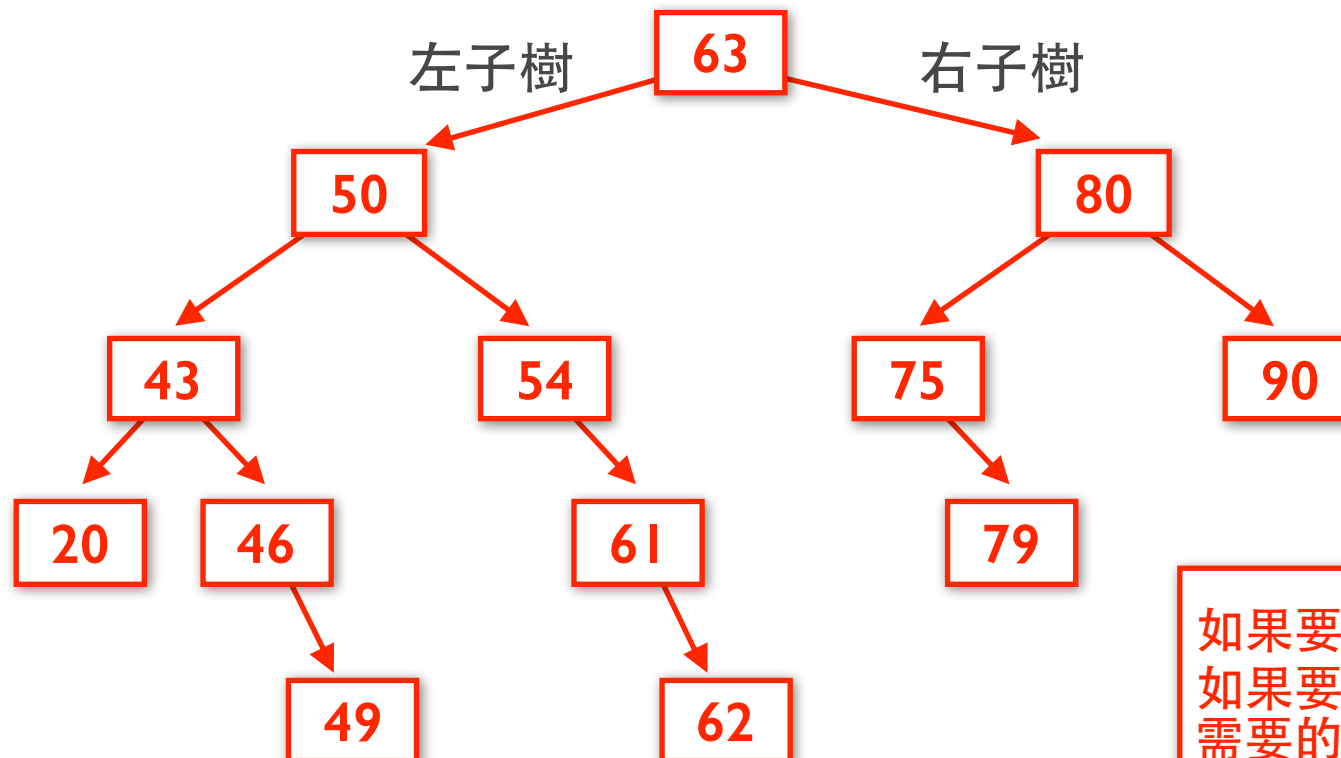
參考動畫：<http://www.cs.usfca.edu/~galles/visualization/HeapSort.html>

【範例】使用 STL 的堆積

- C++ STL 的 `<queue>` 內有 `std::priority_queue` 可以用來快速找出目前所有元素的極值
- ▶ `std::priority_queue` 也是個自適應容器，可以使用不同的底層容器去實作：
 - 底層容器需支援隨機存取
 - 底層容器需支援 `front()`, `push_back()`, `pop_back()`
- ▶ 可選擇 `std::vector` (預設) 或 `std::deque`
- C++ STL 的 `<algorithm>` 內有 `make_heap()`, `push_heap()`, `pop_heap()` 和 `sort_heap()` 等函式可以用來以堆積方式處理容器

二元搜尋樹

- 任何一個節點的值會比該節點的左子樹的值都大，會比右子樹的值都小
 - ▶ 堆積是任何一個節點的值都會大於兩邊的子樹值



Search (75) ;
Search (62) ;
Search (44) ;

Insert (76) ;
Insert (60) ;
Insert (44) ;

如果要找值的話怎麼找？
如果要插入新的值怎麼做？
需要的時間跟個數有關？

【範例】二元搜尋樹

```
template<class ElemType>
class BinarySearchTree {
public:
    BinarySearchTree();
    BinarySearchTree(const BinarySearchTree<ElemType> &rhs);
    ~BinarySearchTree();

    void Insert(const ElemType &elem);
    bool Search(const ElemType &elem);
    void Erase(const ElemType &elem);

    const BinarySearchTree<ElemType> &operator=(
        const BinarySearchTree<ElemType> &rhs);
};
```

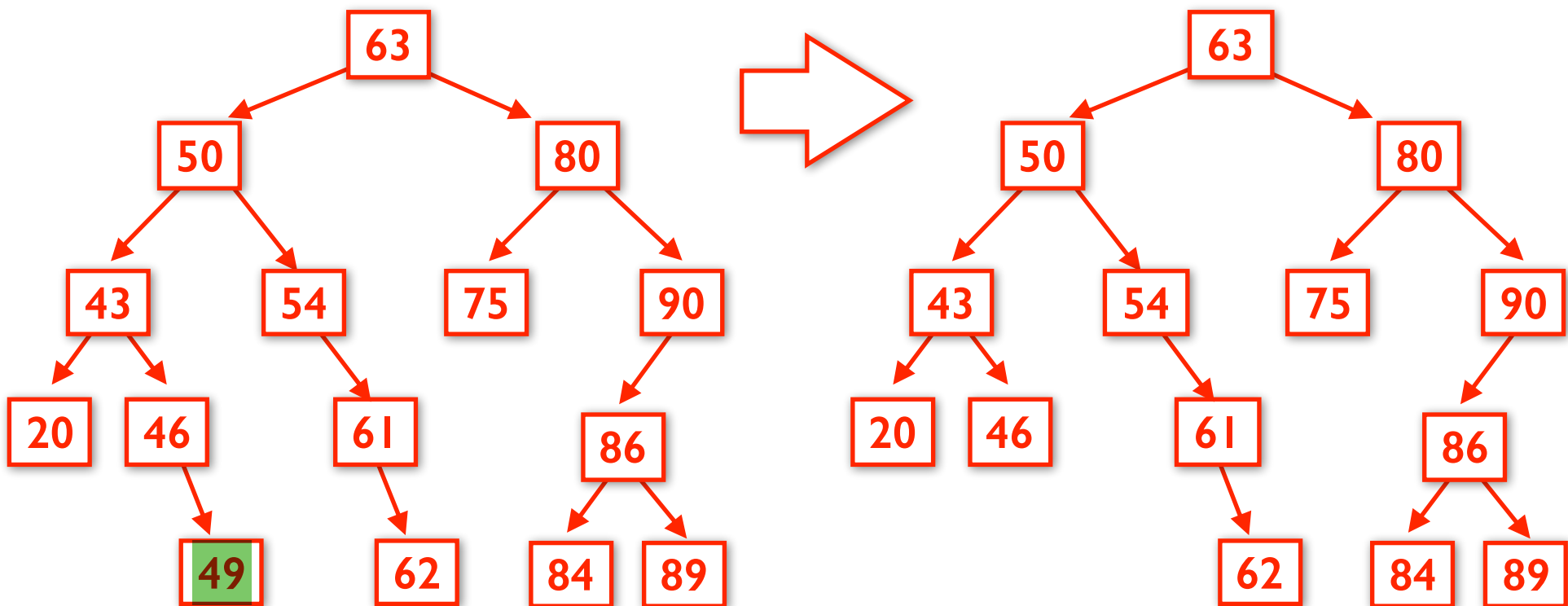
參考動畫：<http://www.cs.usfca.edu/~galles/visualization/BST.html>

[範例] `binary_search_tree.cpp`

二元搜尋樹的刪除 [1]

■ Case 1. 刪除葉子

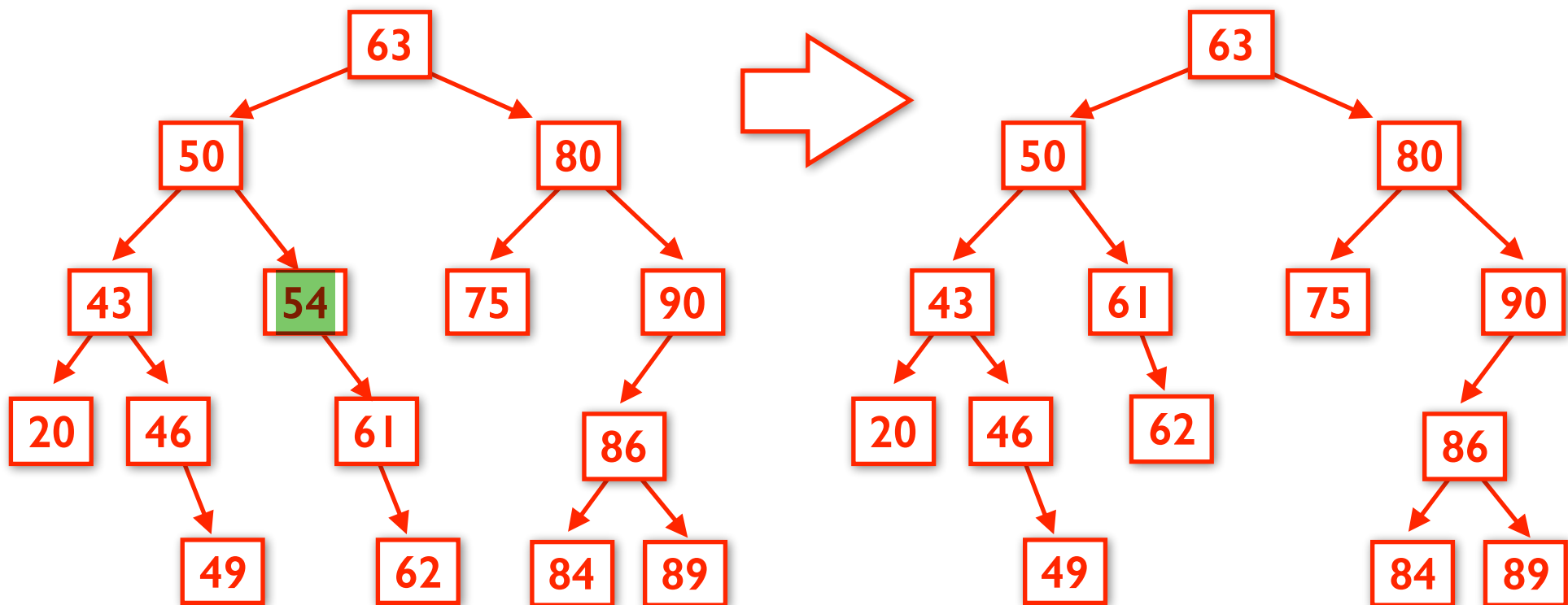
► `Erase(49);`



二元搜尋樹的刪除 [2]

■ Case 2. 刪除無左樹的節點

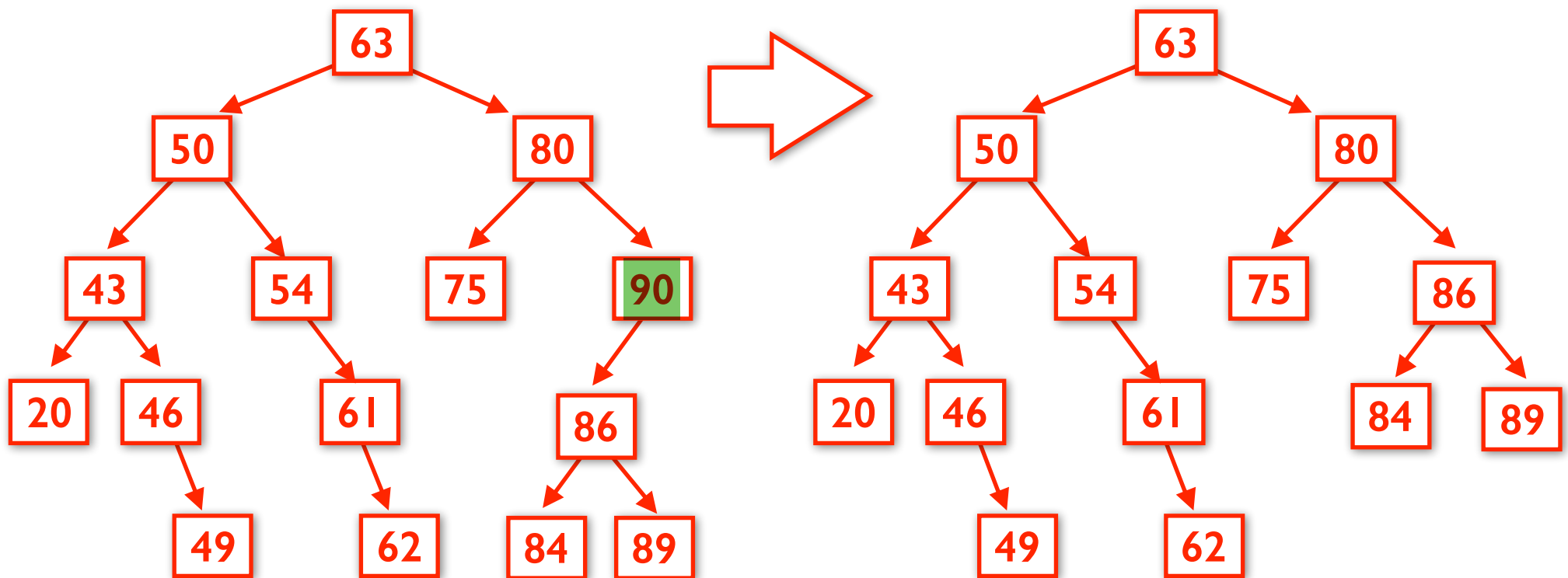
► `Erase(54);`



二元搜尋樹的刪除 [3]

■ Case 3. 刪除無右樹的節點

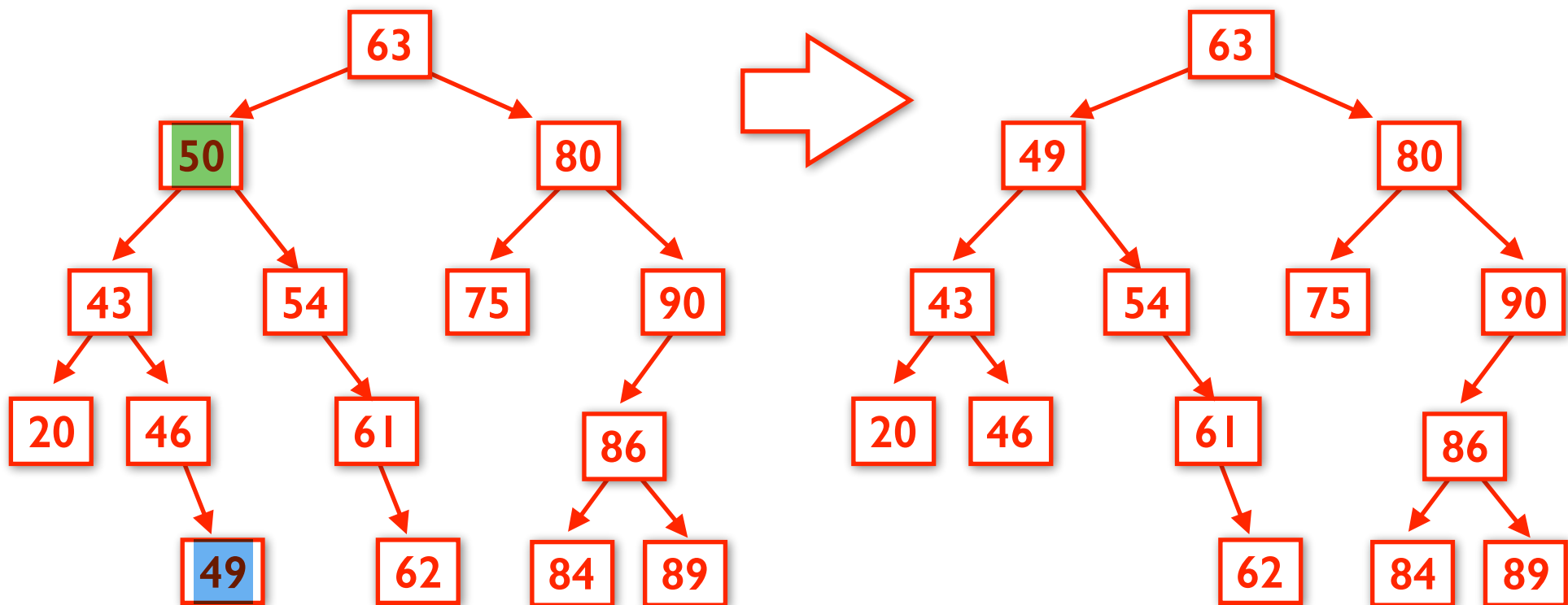
► `Erase(90);`



二元搜尋樹的刪除 [4]

■ Case 4. 兩邊都有子樹

► **Erase (50) ;** (找左邊子樹中最大的來交換)



二元搜尋樹分析

- 各種操作需要的時間比較：**1** 表示與原有個數無關，**N** 表示與原有個數成正比， **$\log_2(N)$** 表示與原有個數取對數成正比

操作	有序型陣列	有序型串列	二元搜尋樹
Search (搜尋值)	$\log_2(N)$	N	約為 $\log_2(N)$
Insert (插入值)	N	移動為 N 插入為 1	約為 $\log_2(N)$
Erase (刪除值)	N	移動為 N 插入為 1	約為 $\log_2(N)$

二元搜尋樹的效率與該樹長得是否『平衡、均勻』有關

使用 STL 的集合與映射

- C++ STL 的 `<set>` 內有 `std::set` 這個集合的類別模版可以使用
- C++ STL 的 `<map>` 內有 `std::map` 這個映射的類別模版可以使用
 - ▶ `map` 支援 `operator[]`
- 通常 C++ STL 會使用更複雜的二元搜尋樹結構（例如紅黑樹）實作來盡量保持『平衡、均勻』以提升效率

[範例] `set.cpp`

[範例] `map.cpp`

	線性搜尋 (linear search)	二元搜尋 (binary search)	雜湊 (hash)
相關函式	<code>std::find</code> , <code>std::find_if</code>	<code>std::lower_bound</code> , <code>std::upper_bound</code> , <code>std::binary_search</code>	<code>std::hash</code> (函式類別模版)
適用容器	一般容器： <code>std::array</code> <code>std::vector</code> <code>std::deque</code> <code>std::forward_list</code> <code>std::list</code>	排序後的一般容器 (如左格) 以及 <code>std::set</code> <code>std::map</code>	特殊容器： <code>std::unordered_set</code> , <code>std::unordered_map</code>
搜尋時間	與元素個數成正比	排序的成本 + 與元素個數的對數成正比	可能與元素個數無關，但 與碰撞程度相關
插入新增或修改元素時間	與使用容器相關 (無額外負擔)	與使用容器相關 + 維持排序的成本	可能與元素個數無關，但 可能需要重新配置雜湊 (與個數成正比)