Final Review

String Matching

```
Rabin-Karp-Matcher(T,P,d,q)
n=T.length
m=P.length
h=d^(m-1) mod q
p=0
t=0
for i=1 to m
        p=(dp+P[i]) mod q
        t=(dt+T[i]) mod q
for s=0 to n-m
        if p==t
                if P[1..m]==T[s+1..s+m]
                        print "Pattern occurs with shift" s
        if s<n-m
                t=(d(t-T[s+1]h)+T[s+m+1]) mod q
```

T: string to be searched
P: pattern to be matched
d: size of the character set
q: max number

Pre-processing: $O(m)$

迴圈跑n-m+1次

Hit的時候比對: $O(m)$

average running time : O(n)

KMP

$T[1:n]$: string to be search
$P[1:m]$: pattern to be matched

```
KMP_Matcher(T, P, n, m)
  pi[1:m] = Prefix_Function(P, m)
  q = 0

  for i = 1 to n
    while q > 0 and P[q+1] != T[i]
      q = pi[q]
    if P[q+1] == T[i]
      q = q + 1
    if q == m
      find valid shift at (i - m)
      q = pi[q]
```

```
Prefix_Function(P, m)
  allocate pi[1:m]
  pi[1] = 0
  k = 0
  for q = 2 to m
    while k > 0 and P[k+1] != P[q]
      k = pi[k]
    if P[k+1] == P[q]
      k = k+1
    pi[q] = k
  return pi[1:m]
```

preprocessing : O(m)

match : O(n)

Linear-Time Sorting

Counting Sort

A: input array
n: input總個數
B: output array
K: 可能出現的input element總數

```
void CountingSort (int A[], int n, int B[], int K) {
        int C[K], i, j,;
O(K)    for (i=0;i<=K;i++)
                C[i]=0;
O(n)    for (j=1;j<=n;j++)
                C[A[j]]=C[A[j]]+1;
O(K)    for (i=1;i<=K;i++)
                C[i]=C[i]+C[i-1];
O(n)    for (j=n;j>=1;j--) {
                B[C[A[j]]]=A[j];
                C[A[j]]--;
        }
}
```

Counting: 數每種element共幾個

"CMF": 數比每種element小或相等的共幾個

從input array最後一個開始依序放入output array

$\boxed{O(n+K)=O(n)}$

Radix Sort

假設:有"多個key"

	以個位數排序	以十位數排序	以百位數排序
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

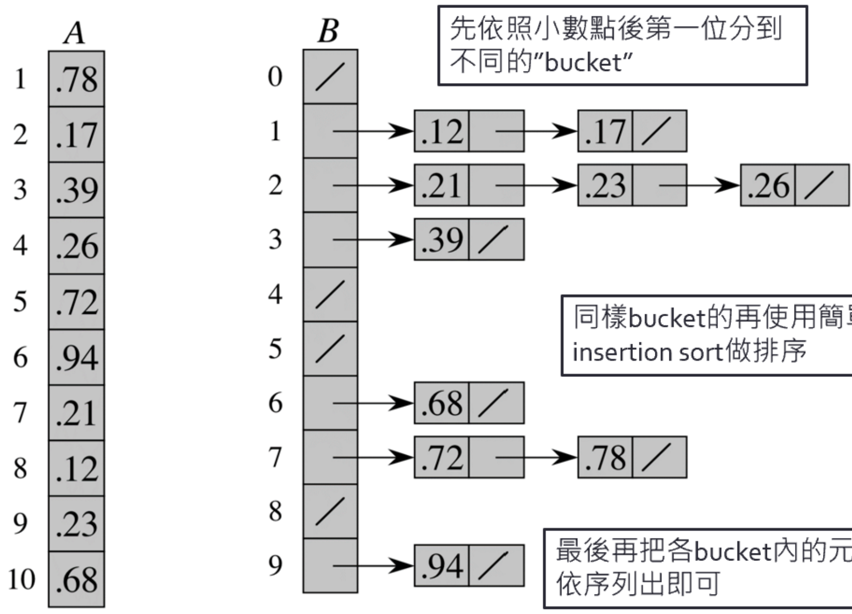
注意:
十進位一樣的部分
會照前一回合的順序
(也就是個位數的順序!)

因此用來排序的演算法
必須是stable!

先依照least significant digit sort, 然後依序往most significant key sort
過去: Least Significant Digit first (LSD) sorting

Bucket Sort

假設: input是從uniform distribution取出來的

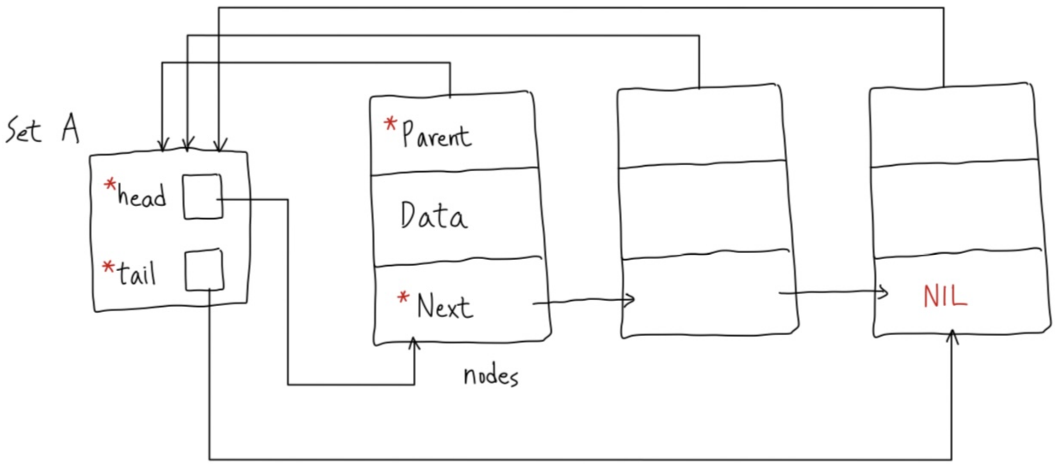


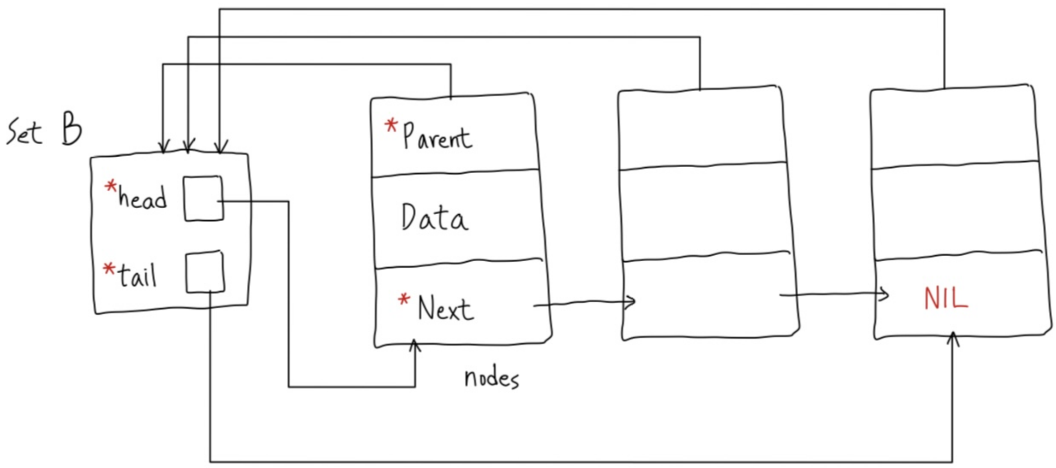
先依照小數點後第一位分到
不同的"bucket"

同樣bucket的再使用簡單的
insertion sort做排序

最後再把各bucket內的元素
依序列出即可

Disjoint Set





方法	FIND(x)	UNION(x,y)	m個MAKET-SET+ UNION+FIND-SET
方法一: array法	O(1)	O(n)	O(m+n^2)
方法三: linked list	O(1)	O(n)	O(m+n^2)
方法三: linked list+ Weighted Union	O(1)	O(log n)	O(m+n log n)
方法二: array (tree)法	O(n)	O(1)	O(m+n^2)
方法二: tree法 +Weighted Union	O(log n)	O(1)	O(m log n)
方法二: tree法 +Weighted Union+Path Compression	O(log n)	O(1)	O($m\ \alpha(n)$)≈ $O(m)$ $\alpha(n)$是一個長得很慢的function Ackermann's function 的反函式，大部分情形$\alpha(n) \leq 4$ (Cormen 21.4)

Hashing

Open addressing

m : hash table size

原本的 hashing function 是 $h(k)$

- **Linear probing : 一直往下一格戳，直到有空位**

  $h'(k,i) = h(k) + i \mod m$

  容易造成 clustering : 一連串的格子裡面都有東西

- **Quadratic probing : 加上一元二次**

  $h'(k,i) = h(k) + c_1 i + c_2 i^2 \mod m$

- **Double hashing : 用兩個 hash function**

  $h'(k, i) = h_1(k) + h_2(k) \times i \mod m$

  最接近 uniform hashing : 任何 probing sequence 出現機率一樣

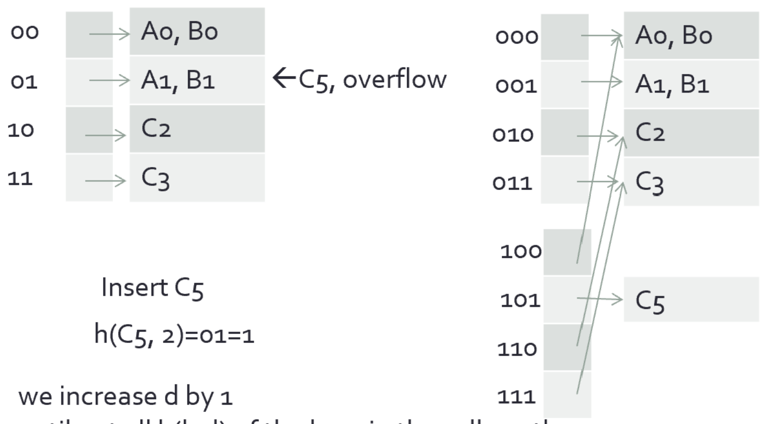
Chaining

多的就掛在後面(用 linked list , BST)

Dynamic hashing

Dynamic hashing using directories

directory depth=
number of bits of the index of the hash table



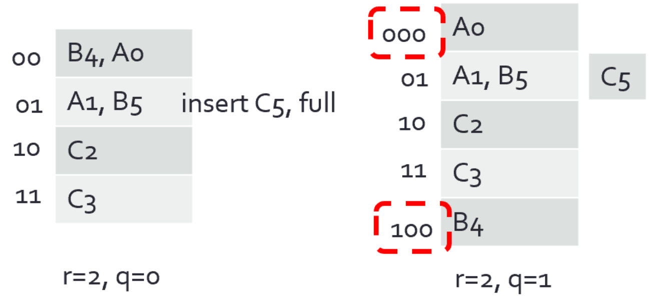
we increase d by 1
until not all h(k,d) of the keys in the cell are the same

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Directoryless Dynamic hashing

- 每次輸入的時候, 如果現在這個櫃子滿了
- 則開一個新的櫃子: $2^r + q$
- 原本q櫃子裡面的東西用
- h(k,r+1)分到q和$2^r + q$兩櫃子裡
- 注意有可能還是沒有解決問題
- 多出來的暫時用chain掛在櫃子下面

問:再加入C1呢? (Horowitz p.415)



k	h(k)
A0	100 000
A1	100 001
B4	101 100
B5	101 101
C1	110 001
C2	110 010
C3	110 011
C5	110 101

RB Tree

Properties

每個 node 都分配一個顏色 紅或黑

沒有 children 的地方都補上 external node (= leaf = NIL)

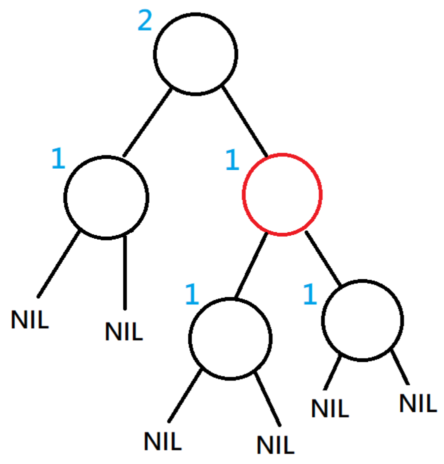
1. 每個 node 不是黑就是**紅**
2. root 是**黑**的
3. 所有 leaf / NIL 都是**黑**的
4. 如果一個 node 是**紅**的，那它的兩個小孩都是**黑**的 (但是黑 node 的小孩可以是黑的)
5. 從每個 node 到他的所有子孫 leaf 的 path 含有一樣多的**黑** node (不包含自己)

💡 一個有 n 個 nodes 的紅黑樹，$h \leq 2\log(n+1)$

Black Height

bh(x) = 從 x 到任何一個他的子孫 leaf 遇到的**黑** node 數量 (要把 leaf / NIL 也算進去喔)

- leaf / NIL 的 bh 為 0
- 對於一棵高度 h 的樹來說，$\text{bh} \geq \frac{h}{2}$

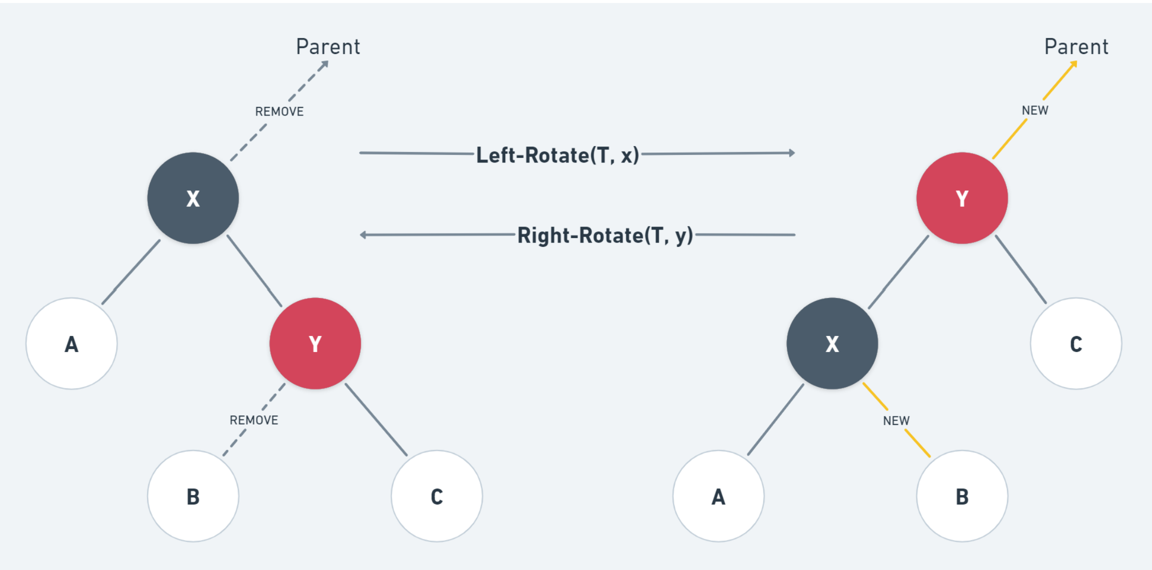


藍色數字是 black height

Rotate

在介紹 Insert 和 Remove 之前，我們必須要新增一些特殊動作以維持紅黑樹的特性

Rotate (樹, 比較高的節點)



Left-Rotate(T, x) 的示意圖，反過來就是 Right-Rotate(T, y)

Rotate 後的大小關係不會變喔！

A > X > B > Y > C

Insertion

用 BST insertion 找到要插入的位置 (插入的節點都預設為紅色)

如果 `parent` 是紅色，則需要修正 (假設 `parent` 是 `parent->parent` 的 `leftchild`)

case 1 : `uncle` 是紅色，不論新增的 node 是 `leftchild` 或是 `rightchild`

- 將 `parent` 變成黑色
- 將 `uncle` 變成黑色
- 將 `parent->parent` 變成紅色
- `current` 指到 `parent->parent`

case 2 : `uncle` 是黑色，新增的 node 是 `leftchild`

- 將 `parent` 變成黑色
- 將 `parent->parent` 變成紅色
- 對 `parent->parent` 進行 right rotation

case 3 : `uncle` 是黑色，新增的 node 是 `rightchild`

調整成 case 2

- `current` 指到 `parent`
- 對新的 `current` 進行 left rotation

做完之後對 current 繼續做判斷直到 `parent` 為黑色

(如果 `parent` 是 `parent->parent` 的 `rightchild` 則與上面對稱)

Deletion

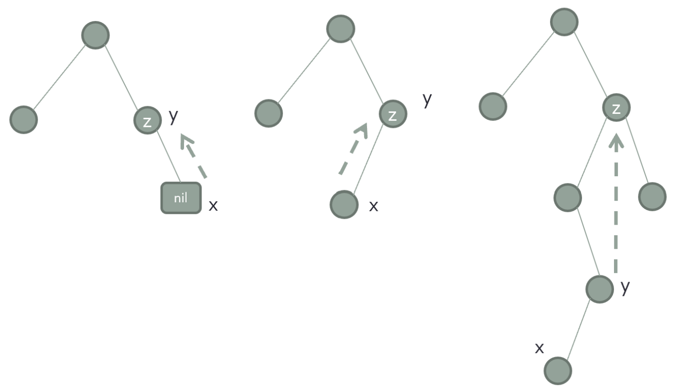
BST deletion :

z : 欲刪除之 node

y : 實際被刪除的 node

x : y 的 `child`

y 的資料要記起來 (之後要恢復)



case 1 : z 沒有 `child`

z = y，直接刪掉，y 原本的位置變成 `NULL`

case 2 : z 只有一個 `child`

z = y，直接刪掉，y 原本的位置變成 x

case 3 : z 有兩個 `child`

找替身，將 y 變成 z 的 successor 或是 predecessor (y 一定最多只有一個 `child`)

用 case 1 or 2 的方法刪除 y，然後將原本 y 的資料放到 z 裡面

Deletion fix :

如果實際被刪除的節點 y 是黑色且不是 root 時，則需要修正 (假設 `current` 是 `parent` 的 `leftchild`)

case 1 : `sibling` 是紅色

- 將 `sibling` 變成黑色
- 將 `current` 的 `parent` 變成紅色
- 對 `current` 的 `parent` 做 left rotation
- 將 `sibling` 指到 `current->parent` 的 `rightchild`

進入 case 2 or 3 or 4

case 2 : `sibling` 是黑色，而且 `sibling` 的兩個 `child` 都是黑色

- 將 `sibling` 變成紅色
- 將 `current` 指到 `current->parent`

若新的 `current` 是紅色，把 `current` 變成黑色即可結束修正

若新的 `current` 是黑色，而且不是 root，則繼續下一輪迴圈

case 3 : `sibling` 是黑色，而且 `sibling` 的 `rightchild` 是黑色，`leftchild` 是紅色

- 將 `sibling` 的 `leftchild` 變成黑色
- 將 `sibling` 變成紅色
- 對 `sibling` 進行 right rotation
- 將 `sibling` 指到 `current->parent` 的 `rightchild`

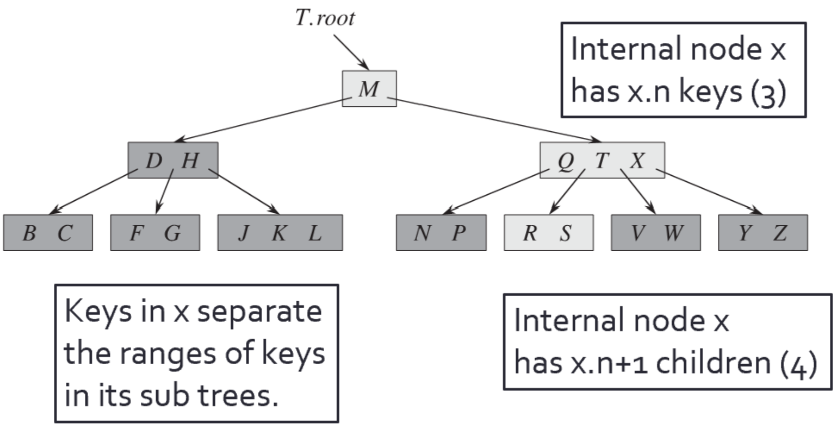
進入 case 4

case 4 : `sibling` 是黑色，而且 `sibling` 的 `rightchild` 是紅色

- 將 `sibling` 變成 `current` 的 `parent`的顏色
- 將 `parent` 變成黑色
- 將 `sibling` 的 `rightchild` 變成黑色
- 對 `parent` 進行 left rotation
- 將 `current` 指到 `root`，把 `root` 塗黑

結束修正

B-Tree



blanced search tree

all leaves have the same depth : h (tree's height)

除了 root 之外的每個 node :

$2 \leq t \leq$ `child` $\leq 2t$

$t-1 \leq$ `keys` $\leq 2t-1$

總共有 n 個 keys、高度 h、minimum degree t 的 B-tree : $h \leq \log_t \frac{n+1}{2}$

Search

Search in B-Tree

```
B-TREE-SEARCH(x, k)
i = 1
while i    x.n and k > x.key_i
    i = i + 1
if i    x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Input:
x: search from this node
k: key to be searched

Return value:
(x,i): key k is found at node ***x***'s ***i***-th key

CPU time: $O(t \log_t n)$
Disk I/O: $O(\log_t n)$

Insertion

node is full → split 然後往上補

Split the root is the only way to increase the height of a B-tree

```
B-TREE-SPLIT-CHILD(x, i)
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t   1
for j = 1 to t   1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t   1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

Split node **x**'s **i**-th child, which is full

CPU time: O(t)
Disk I/O: O(1)

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

$$\boxed{\begin{array}{ll}\text{CPU time:} & O(t\log_t n)\\ \text{Disk I/O:} & O(\log_t n)\end{array}}$$

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf
    while i    1 and k < x.key_i
        x.key_{i+1} = x.key_i
        i = i    1
    x.key_{i+1} = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i    1 and k < x.key_i
        i = i    1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t    1
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

Deletion

為了防止刪除後 keys 不夠 (< t-1)

我們在呼叫任何 node 之前都要確保至少有 t 個 keys

從 root 開始向下找欲刪除的 key k

假設我們現在一路找到了 node x，那麼會有三種情況

case 1 : x 是 leaf node

如果 k 在裡面，直接刪掉 <end>

如果 k 不在裡面，那麼整棵樹都不會有 k <end>

case 2 : x 是 internal node，而且裡面有 $k = x.\text{key}_i$

$x.c_i$: k 的前一個 `child`

$x.c_{i+1}$: k 的後一個 `child`

- **case 2a** : $x.c_i$ 有至少 t 個 keys

在 $x.c_i$ 裡面找 k 的 predecessor k'

在 $x.c_i$ 把 k' 刪除，並把 k 替換成 k'

<遞迴到 $x.c_i$ 判斷下一個 case>

- **case 2b** : $x.c_i$ 只有 t-1 個 keys，$x.c_{i+1}$ 有至少 t 個 keys

  在 $x.c_{i+1}$ 裡面找 k 的 successor k'

  在 $x.c_{i+1}$ 把 k' 刪除，並把 k 替換成 k'

  <遞迴到 $x.c_{i+1}$ 判斷下一個 case>

- **case 2c** : $x.c_i$ 和 $x.c_{i+1}$ 都只有 t-1 個 keys

  把 $x.c_i$, k , $x.c_{i+1}$ 所有的 keys 都合併到到 $x.c_i$

  這樣一來新的 $x.c_i$ 就會有 2t-1 個 keys

  在 $x.c_i$ 把 k' 刪除

  <遞迴到 $x.c_i$ 判斷下一個 case>

case 3 : x 是 internal node，但裡面沒有 k

我們要繼續往下找，但同時要確保經過的每個 node 都有至少 t 個 keys

假設 k 會在 $x.c_i$ 這個 `child` 裡面 (有可能是在 `child` 的某一個 `child` 裡面，或是 …)

如果 $x.c_i$ 的 keys 夠多，我們可以直接 <遞迴到 $x.c_i$ 判斷下一個 case>

如果不夠多，則會有下面兩種情況

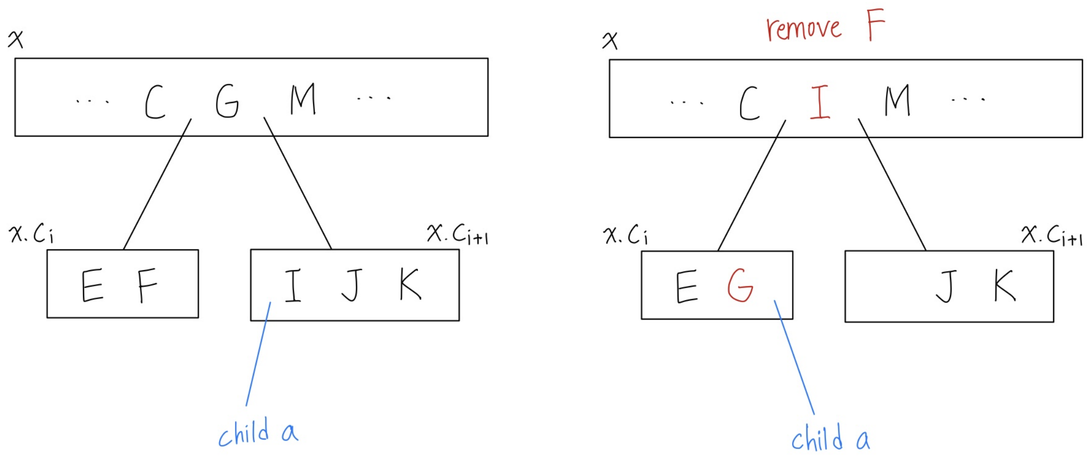
- **case 3a** : $x.c_i$ 只有 t-1 個 keys，但他的 immediate sibling ($x.c_{i-1}$ or $x.c_{i+1}$) 有至少 t 個 keys

  從 x 裡面拿「 $x.c_i$ 和 sibling 之間的 key 」放到 $x.c_i$ 裡面

  從 sibling 裡面拿「靠 $x.c_i$ 最近的那個 key」放到 x 裡面

  記得把 sibling 裡面對應的 `child` 改到 $x.c_i$ 裡面

  <遞迴到 $x.c_i$ 判斷下一個 case>



case 3a 的例子

- **case 3b** : $x.c_i$ 和它的 immediate sibling 都只有 t-1 個 keys

  把 $x.c_i$ 和其中一個 sibling 合併

  然後從 x 裡面拿「$x.c_i$ 和 sibling 之間的 key」放到新的 $x.c_i$ 當作中間的 key

  <遞迴到 $x.c_i$ 判斷下一個 case>

注意 : 對於 case 2c 和 3b，如果 x 是 root 的話，可能會有 key 不夠的問題 (< 2)，遇到這種情況，我們就把 x 刪除，並把 root 往上一格