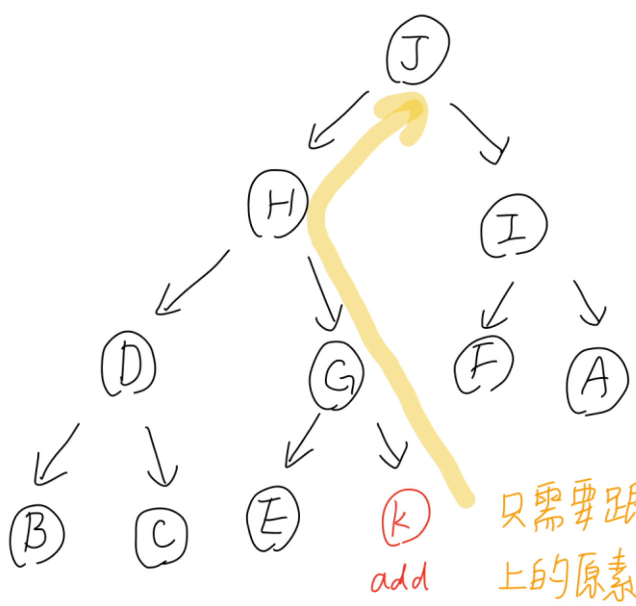
§ Priority Queues

*檢傷自動化

- build a heap

```
struct Heaptype {
    void ReheapDown (int , int);  // 刪除
    void ReheapUp (int, int);     // 新增
    ItemType *elements;
    int numElements;
};
```

1. Insert a new element (ReheapUp function)



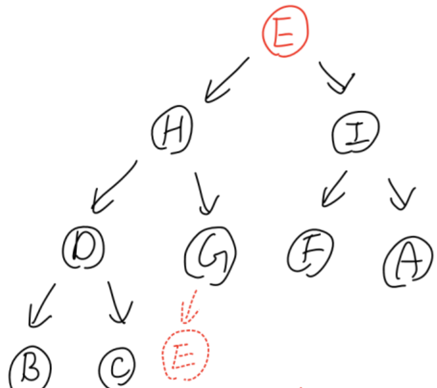
只需要跟這條上的原素比

(1) Insert the new element in the next bottom rightmost place

(2) fix the heap property by calling ReheapUp

O(log n)

2. Delete the largest element



原本E在這

(1) copy the bottom rightmost element to the root

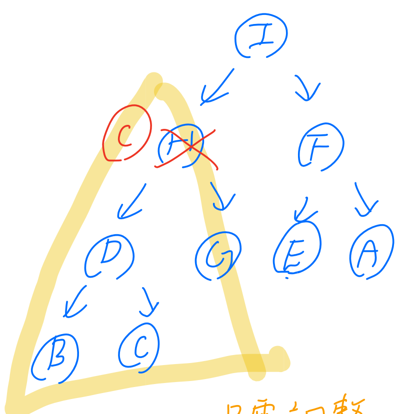
(2) Delete the bottom rightmost node

(3) Fix the heap property by calling Reheap Down

$O(\log n)$

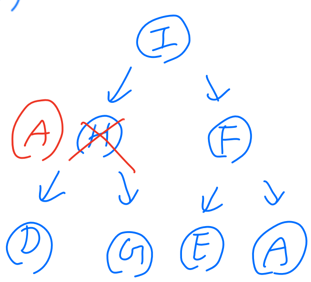
Think: 刪除中間節點要如何維護 heap ?

(1)



只需調整 此"Heap"

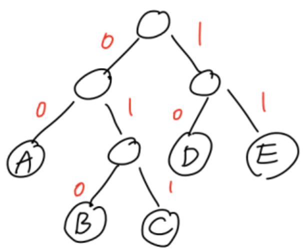
(2)



這個情況較複雜

Think Think ~

- 應用: Huffman coding

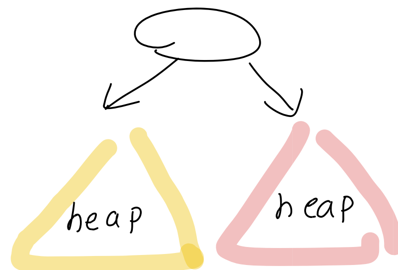


A = 00　D = 10

B = 010　E = 11

C = 011

* semi-heap

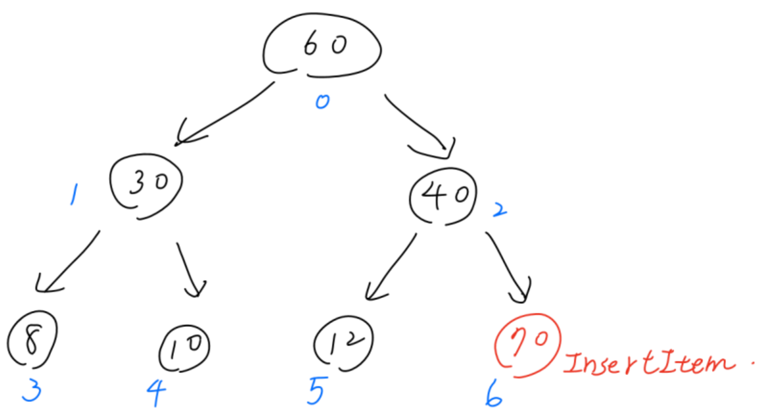


e.g. EAEBAECDEA

⇒ 1100110100011011101100

- Heap Operations

0	60
1	30
2	40
3	8
4	10
5	12
6	70



找父節點 = $\frac{n-1}{2}$

找子節點 = $(n\times 2)+1$ or $(n\times 2)+2$

1. Insert

```
void heapInsert ( & Item) {
    if (size >= Maxheap) return ;
    items[size] = newItem ;
    int place = size, parent = (place-1) / 2 ;
    while( parent >= 0) & (items[place] > items[parent]) {
        temp = items[parent];
        items[parent] = items [place];
        items [place] = temp ;
        place = parent;
        parent = (place-1) /2 ;
    }// while
    size ++;
}// void
```

2. delete

```
void heapDelete ( &Item) {
    if (heapisEmpty) {
        rootItem = items[0];
        size--;
    }
    if ( isnotEmpty) {
        items [0] = items[size];
        heapRebuild (0);
    }
}//void
```

3. heapRebuild

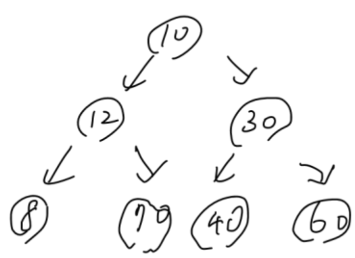
```
void heapRebuild(int root){
    int child = 2*root+1;
    if(child<size){
        int right = child+1;
        if(right<size)&&(items[right]>items[child]))
            child = right;

        if(items[root]<items[child]){
            temp = items[root];
            items[root] = items[child];
            items[child] = temp;
            heapRebuild(child);
        }
}// void
```

max-heap.

- Rebuild 的時候要從底部開始看，如果從root開始不能保證root是最大值.

e.g



- Heap Sort

  time: $O(\boxed{n} * \log n)$

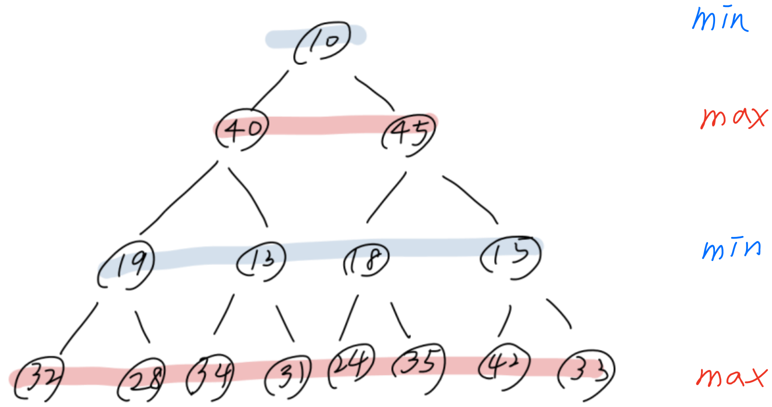
  有幾層

  要刪除幾次東西

§ 堆積變形

(一) Double-ended Priority Queue (DEPQ)

(Min-max Heap)



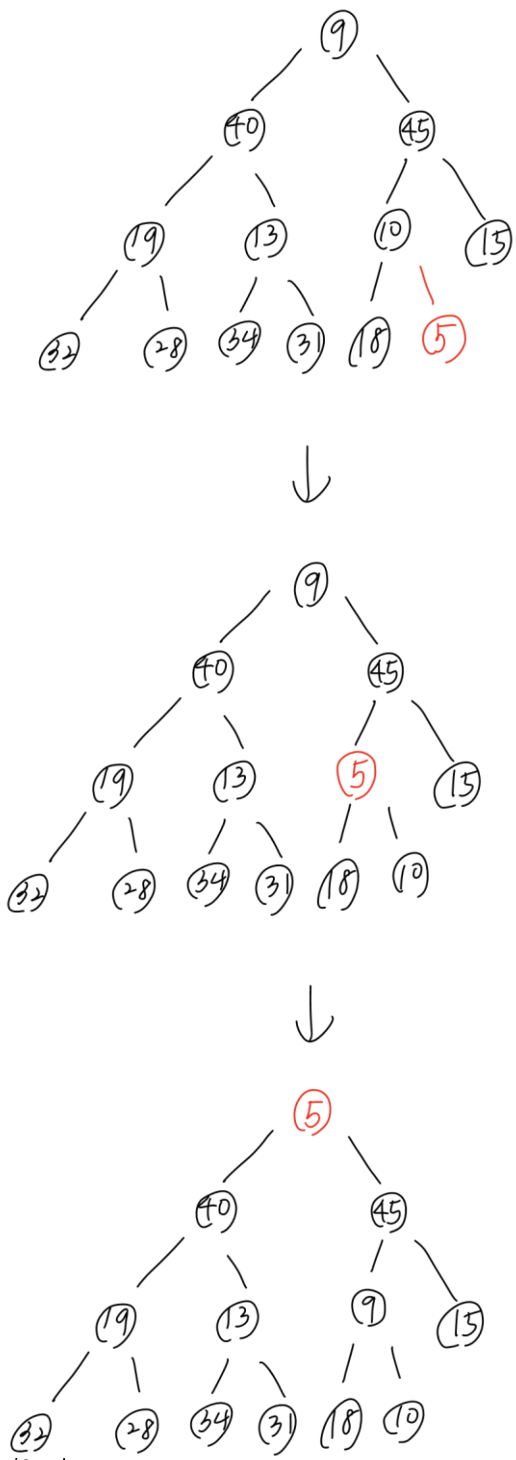
好處：容易找出最大值、最小值

- Insert

  1. decide which level → min or max

  2. check whether to swap with parent.

  e.g



※5要跟祖父節點比大小

父節點：$\frac{n-1}{2}$

祖父節點：$\frac{\left(\frac{n-1}{2}\right)-1}{2}$.

- Delete

1. replace the root with the last element

2. check whether to swap with its smaller child

- 判断 level

  level $= [\log_2(n+1)]+1$

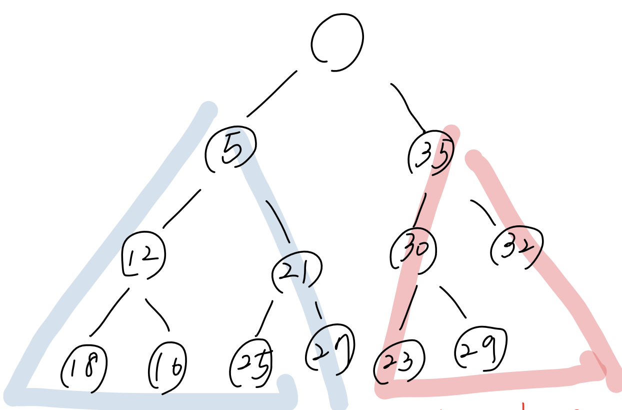
  奇层 min，偶层 max

- 孙子：$i*4+j$

  i为本人，j为 3,4,5,6

  祖父：$\frac{[\frac{n-1}{2}]-1}{2}$

(二) Doubled-ended Heap (DEAP)

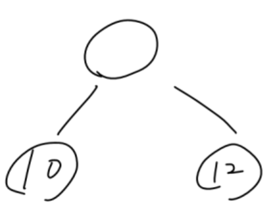


- Insert

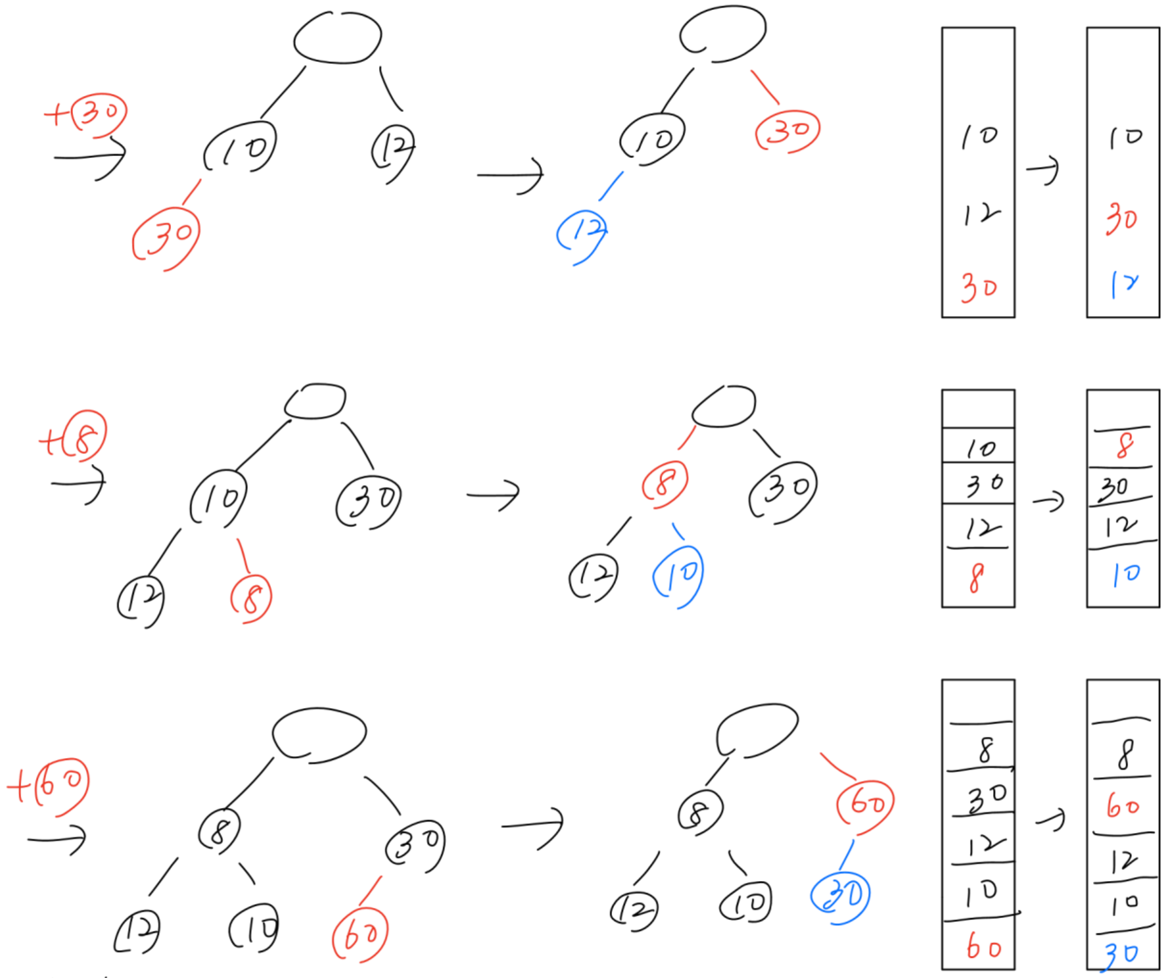
  对应的節点

  1. examine the corresponding nodes : left < right
  2. ReheapUp necessary

  e.g. 10, 12, 30, 8, 60



0	
1	10
2	12



- Delete

1. Replace the root of min-heap with the last element
2. ReheapDown if necessary.
3. Examine the corresponding nodes : left < right

how?

※ how?

$2^{i-1} < n < 2^{i}$ $\left(i = [\log_2(n+1)]+1 \right)$

right $= n + [(2^{i} - 2^{i-1})/2]$

(三) 堆積變型應用

1. Double-ended Priority Queues

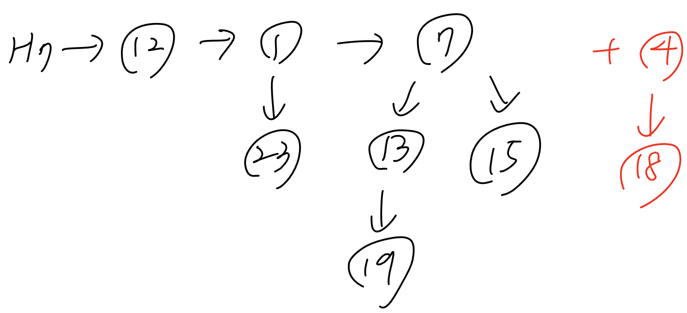
數據非常大量時可使用 (quick sorted + heap sorted)

2. Mergeable Priority Queues

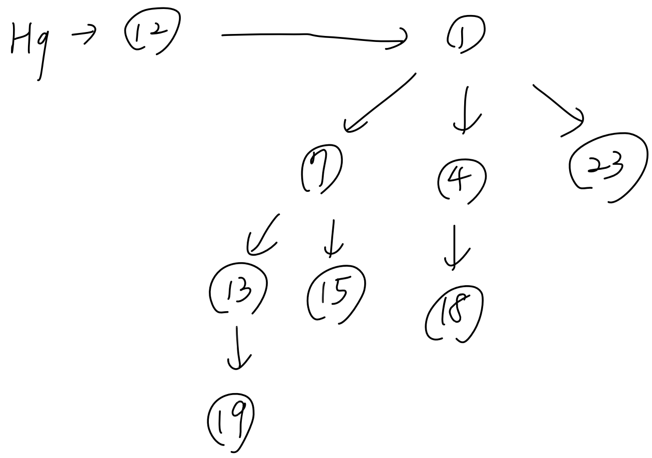
合併 2個 Queues.

3. Binomial Heap

① $7 = 2^0 + 2^1 + 2^2$



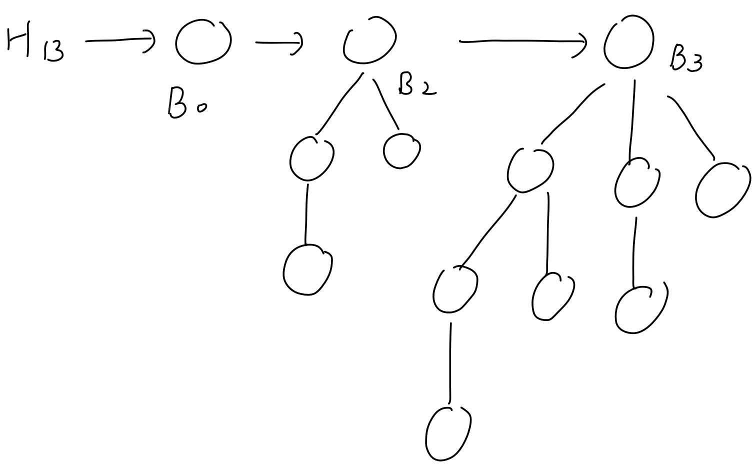
② $9 = 2^0 + 2^1 + 2^1 + 2^2 = 2^0 + 2^3$



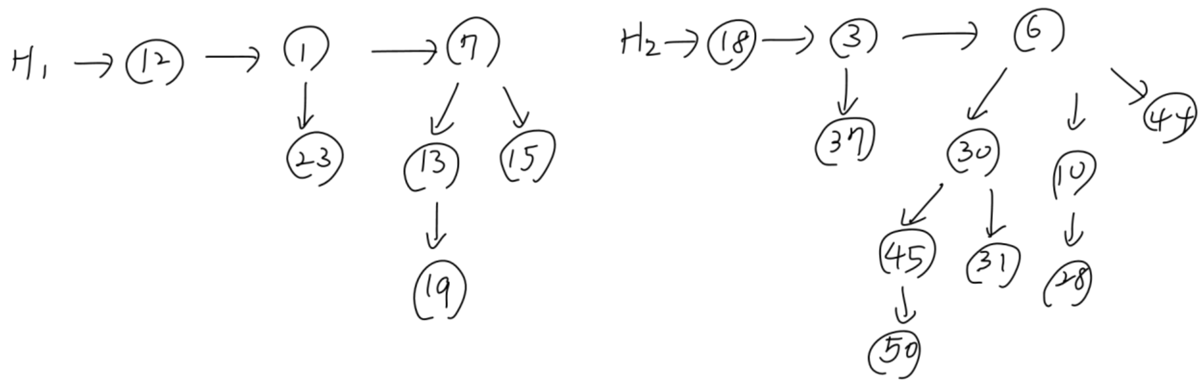
* $4 < 7$
所以新增一個
~~子~~孫在1下面

③ $13 = 2^0 + 0 + 2^2 + 2^3$

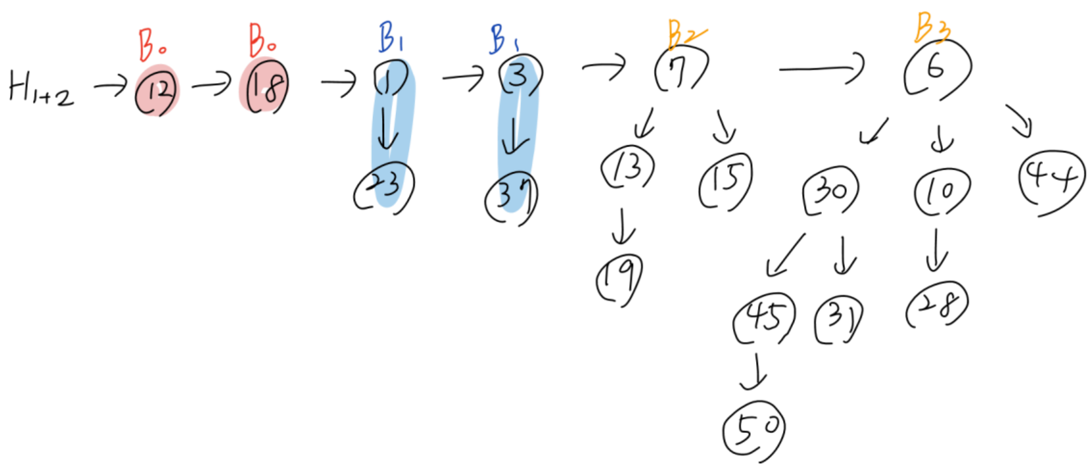
2 | 13
2 | 6 …… 1
2 | 3 …… 0
1 …… 1



• 合併 merge

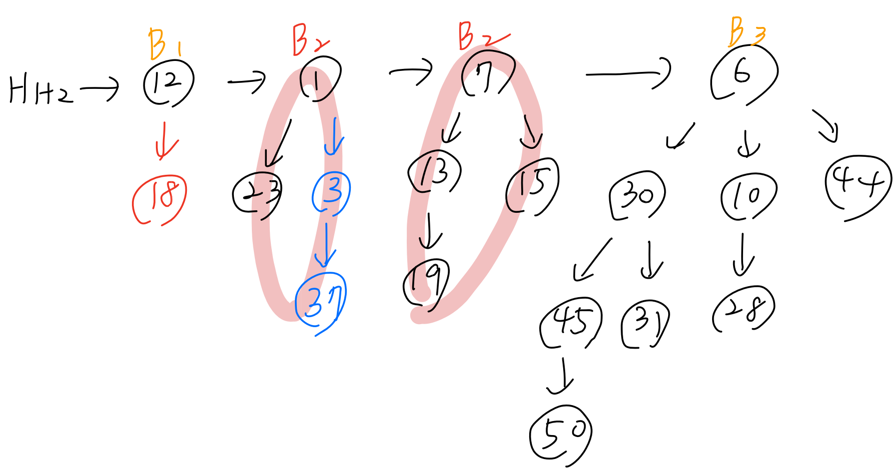


H1
12
1
7
23
13
15
19
H2
18
3
6
37
30
10
44
45
31
28
50

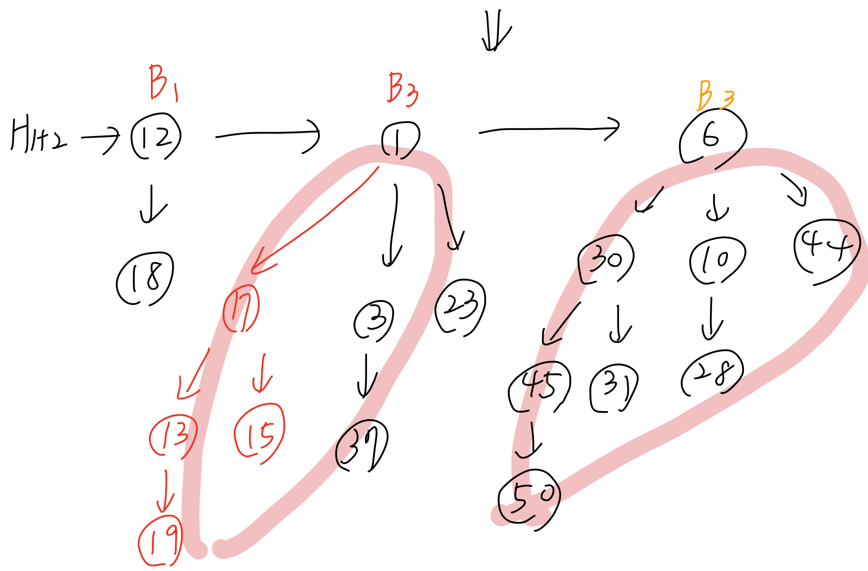


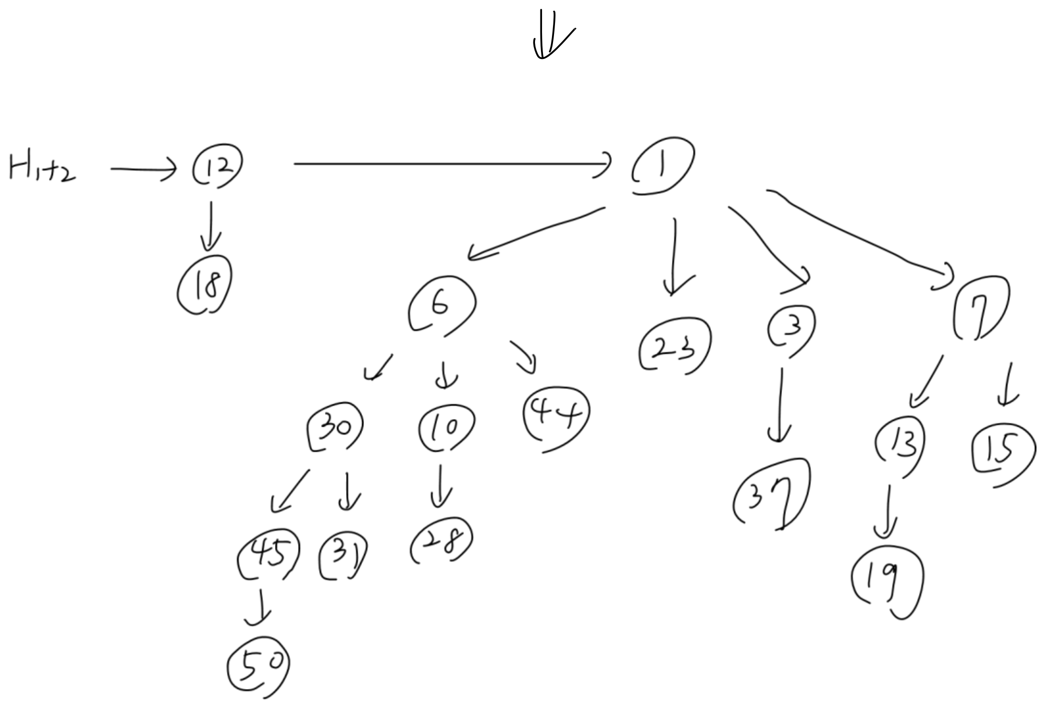
H1+2
B0
B0
B1
B1
B2
B3
12
18
1
3
7
6
23
37
13
15
30
10
44
19
45
31
28
50

⇓



HH2
B1
B2
B2
B3
12
1
7
6
18
23
3
37
13
15
19
30
10
44
45
31
28
50





* 新增、刪除都是合併的概念

* 每個 k 值只會出現一次　　* 效率：$O(\log n)$

2^k

Ch3

§ 2-3 tree

(一) Insert.

1. Locate the leaf at which the search for I would terminate.

2. Insert the new item I into the leaf

3. If the leaf now contains two items, you are done.

4. If the leaf now contains three items, split the leaf into two nodes, n_1 and n_2.

code:

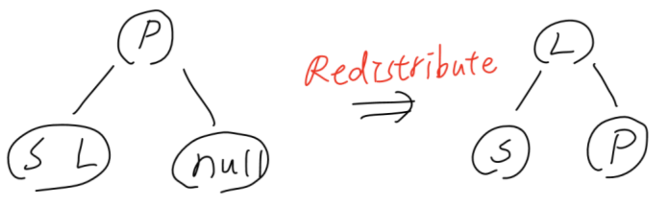
```
InsertItem() {
    if (size == 3) {
        split(leafnode);
    }
    else add to node.
}

split ( ) {
    if (treenode == root) create new root P
    else 取中間值变成 parent
} // 遞迴
```

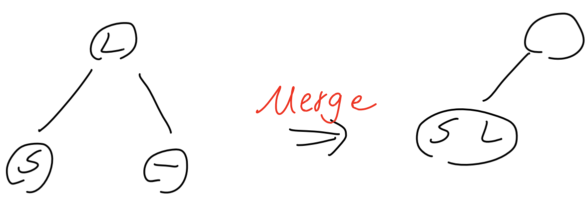
(二) Delete

Q: What if the node is empty?

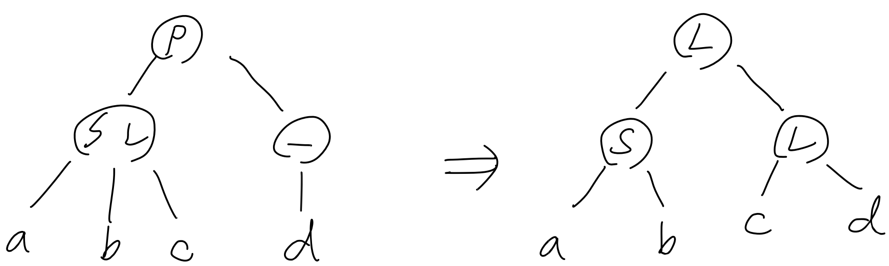
a. Redistribute (重新分配) values



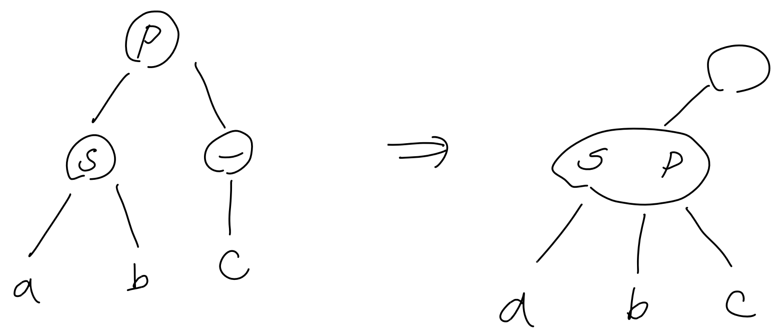
b. Merge into a leaf



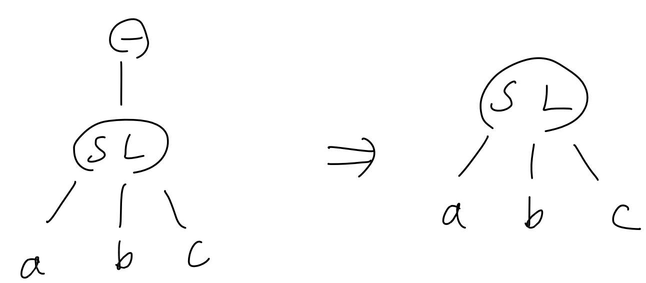
c. Redistribute values and children



d. Merge into an internal node



e. delete the root



- Steps
  1. Locate the leaf at which the search for I would terminate
  2. Delete I from the leaf
  3. If the leaf now contains one item, you are done.
  4. If the leaf now contains no item, choose one of the following operations to fix.

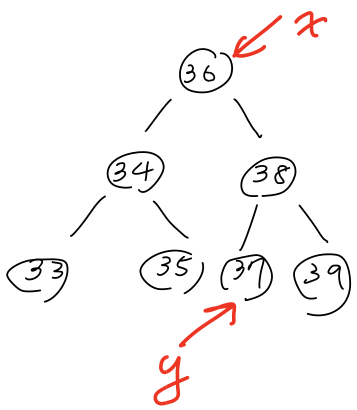
(a) Redistribute the values

(b) Merge into a leaf

(c) Redistribute values and children

(d) Merge into a internal node

```
void deleteItem {
    if (x is not a leaf)
        y = Successor(x);
        Swapkey(x, y);
        x = y;
    Delete key from x;
    if (x now has no item)
        fix(x);
}
```




```
void fix(x) {
    if (x == root) remove the root;
    else {
        p = parent of x;
        if (the nearest sibling of x has two items)
            Redistribute items among the sibling, p & x;
            if (x is not a leaf)
                Move appropriate child from sibling to x;
```

else // merge

S = the nearest sibling of x;

Move appropriate item down from p to S;

if (x is not a leaf)

    Move x's child to S;

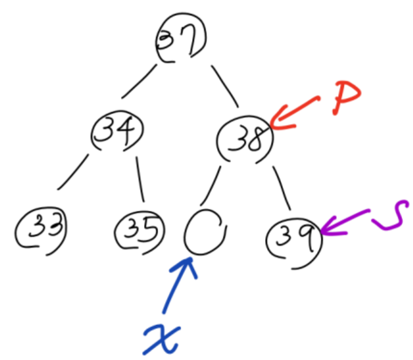
remove x;

if (p now has no item)

    fix(p);

}

}



§ AVL tree

(一) an AVL tree.

1. a balanced binary tree
2. 高度最低的搜索樹.

(二) After insertion or deletion

1. Insertion.

a. insert the new key as a new leaf

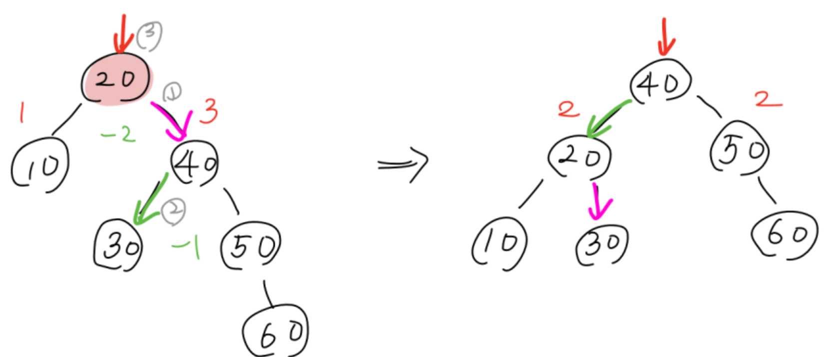
b. trace the path from the new leaf towards the root.

BF (平衡係數)

任選 node : |左子樹 height − 右子樹 height| <= 1

* 發現平衡係數不对時, 檢查重的那方是同號 (− or +) 使用 single rotation, 不同號則使用 Double rotation.

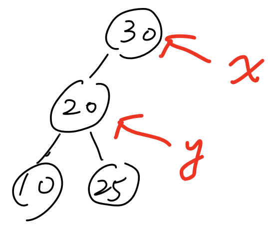
① single rotation.



$1-3=-2$ $\quad$ $2-2=0$

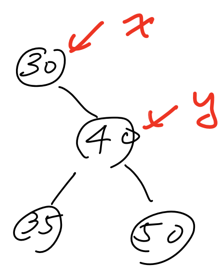
- LL (++ / +0)

```
node rotateLL(node x) {
    node y = x→left;
    x→left = y→right;
    y→right = x
    return y;
}
```

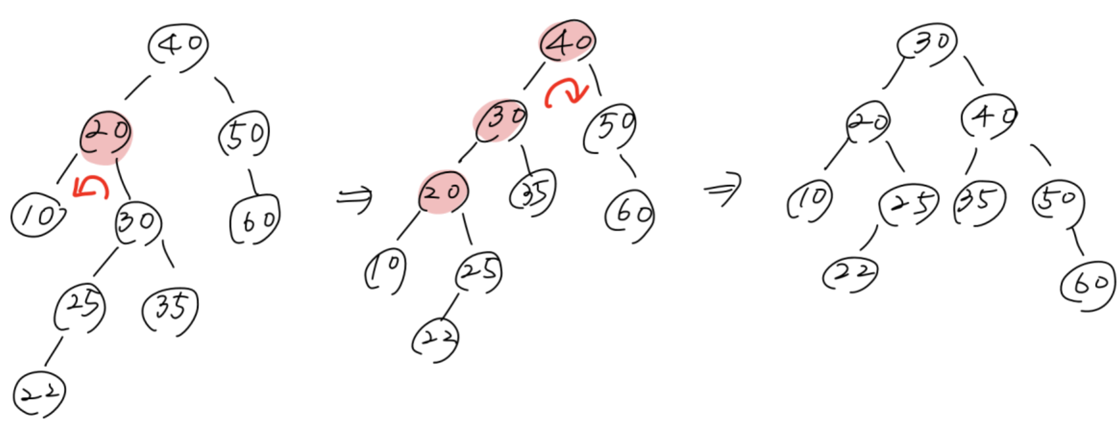


- RR (−− / −0)

```
node rotateRR(node x) {
    node y = x→right;
    x→right = y→left;
    y→left = x ;
    return y;
}
```

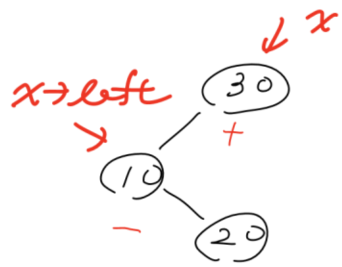


② Double rotation

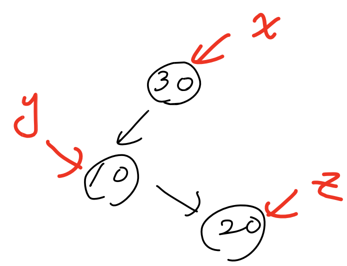


• LR (+−)

```
node rotateLR1(node x) {
  // 先RR後LL
    x -> left = rotateRR(x -> left);
    return rotateLL(x);
}
```

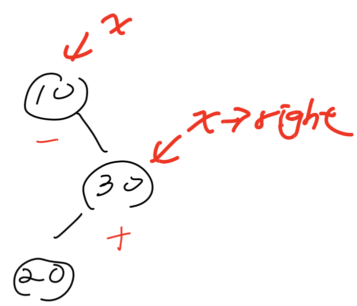



```
node rotateLR2(node x) {
    node y = x -> left;
    node z = y -> right;
    y -> right = z -> left;
    x -> left = z -> right;
    z -> right = x;
    z -> left = y;
    return z;
}
```



• RL (−+)

```
node rotateRL1(node x) {
// 先LL後RR
    x -> right = rotateLL(x -> right);
    return rotateRR(x);
```



```
}

node rotateRL2(node x {
    node y = x→left;
    node z = y→right;
    y→right = z→left;
    x→left = z→left;
    z→right = x;
    z→left = y;
    return z;
}
```