演算法 搜尋

- 常用在即時資料或資料庫
  搜尋 單一資料 (key) 或 字串 (string)
- 循序搜尋、二元搜尋
- 二元搜尋樹、平衡樹

循序搜尋

- 循序搜尋(Sequential search) 暴力法(Brute force)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E

```
int SequentailSearch(int A[], int n, int key){
        int i;
        for (i=0; i<=n-1;i++)
                if (key==A[i]) return i;
        return -1;
}
```

循序搜尋效能分析

- 失敗：需要經過 n 次比對 (假設為機率 p)
- 成功：平均為 (n+1)/2 次比對 (機率 1-p)

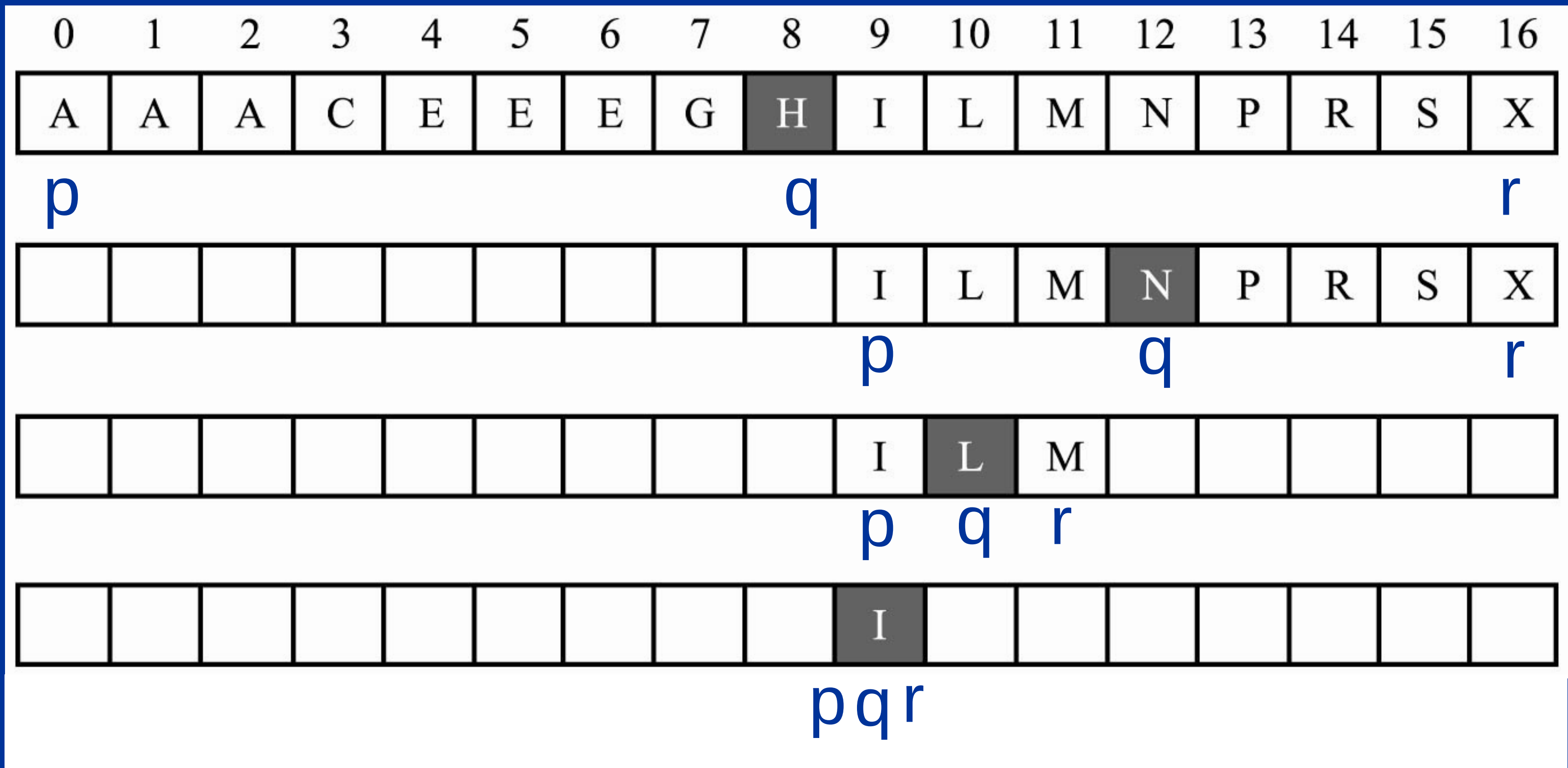
- 平均搜尋時間

  $T(n)=T_0(pn + (1-p)(n+1)/2)$

平均時間複雜度為 $\Theta(n)$

二元搜尋 (binary search)

- 資料先排序, 時間複雜度 Θ(n log n)
  之後使用二元搜尋，時間複雜度 Θ(log n)



範例 7-1

二元搜尋

```
int BinarySearch(int A[], int n, int key){
   int p=0, q, r=n-1;
   while( r>=p ){
        q=(p+r)/2;
        if ( key==A[q]) return q;
        if ( key<A[q]) r=q-1;  else p=q+1;
   }
   return -1;
}
```

測試

範例 7-1

```
main(){
    int n=10;    int A[n];    int k;   srand(time(0));
    for(k=0; k<n; k++) A[k]=rand()%n;
    QuickSort(A, 0, n-1);  //排序
        for(k=0; k<n; k++) printf("%d ", A[k]);
              printf("\n");
        k=rand()%n; printf("key=%d\n", A[k]);
    k=BinarySearch(A, n, A[k] );  //二元搜尋
        if(k!=-1) printf("found %d", A[k]);
}
```

測試

- n=10
  0 2 3 4 5 8 8 9 9 9
  key=2
  found 2

- n=15
  0 2 2 3 4 5 6 6 9 10 11 12 12 13 14
  key=14
  found 14

範例 7-2

二元搜尋(遞迴)

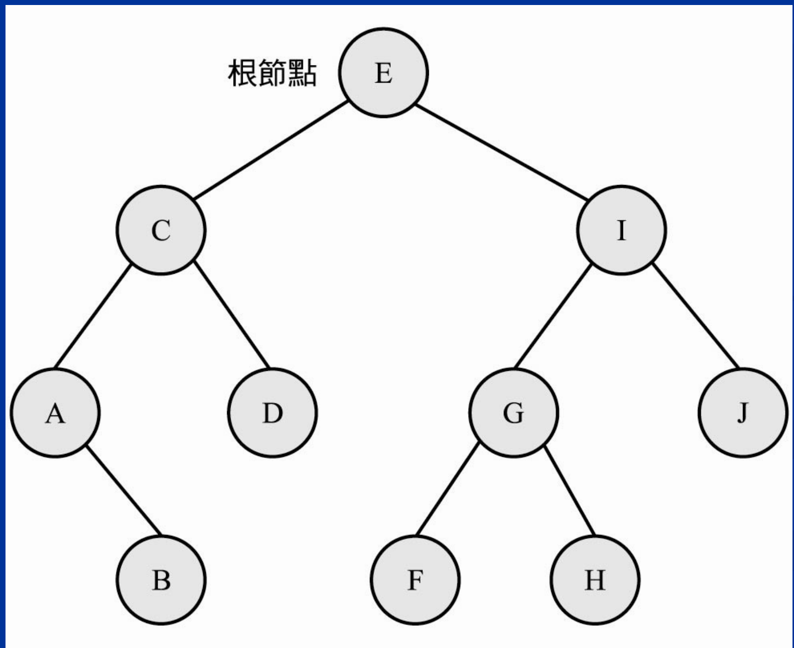
```
int BinarySearch(int A[], int p, int r, int key){
   int q;
   if( r>=p ){  q=(p+r)/2;
        if ( key==A[q]) return q;
        if ( key<A[q])
                return BinarySearch(A, p, q-1, key);
          else
                return BinarySearch(A, q+1, r, key);
   } else return -1;
}
```

二元搜尋分析

- 屬於 divide and conquer, 資料須先排序
- 新增/刪除新的資料, 時間複雜度仍為 Θ (n)
  → 適用於新增資料頻率較少的情況
- 如何降低搜尋、新增、刪除時間為 Θ (log n)
  → 陣列轉換為二元搜尋樹

二元搜尋樹(binary search tree)

- 二元搜尋樹為二元樹資料結構的一種
- 通常使用鏈結串列 (list), 節點內含鍵值 (key)
  左邊子樹鍵值 ≤ 節點鍵值 ≤ 右邊子樹鍵值



二元搜尋樹的節點

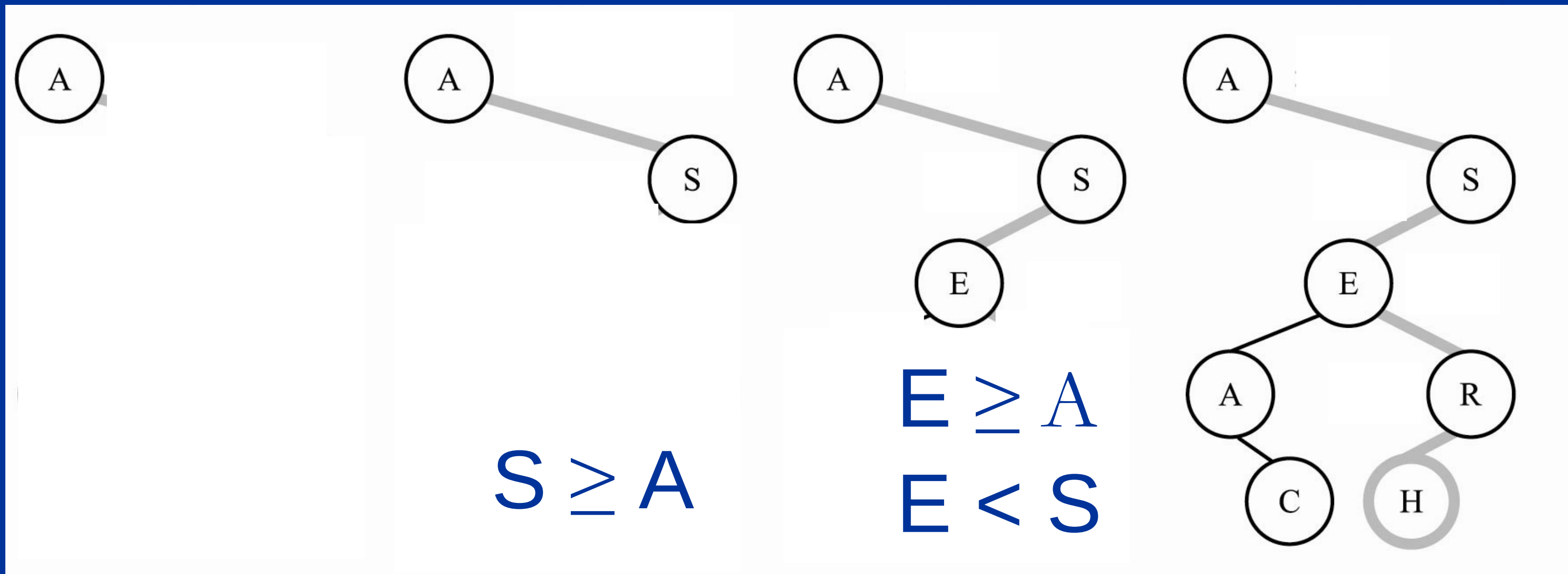
```
typedef struct node {
    int key;    struct node *left, *right;
} NODE;

NODE* new_node(int key){   //配置一個新節點
    NODE *ptr;
    ptr = (NODE*) malloc(sizeof(NODE));
    ptr->key=key;       ptr->left=ptr->right=NULL;
    return ptr;
}
```

指標 NULL 的值為 0

- 建構二元搜尋樹

{ A, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E }

小於放左邊　　大於等於放右邊



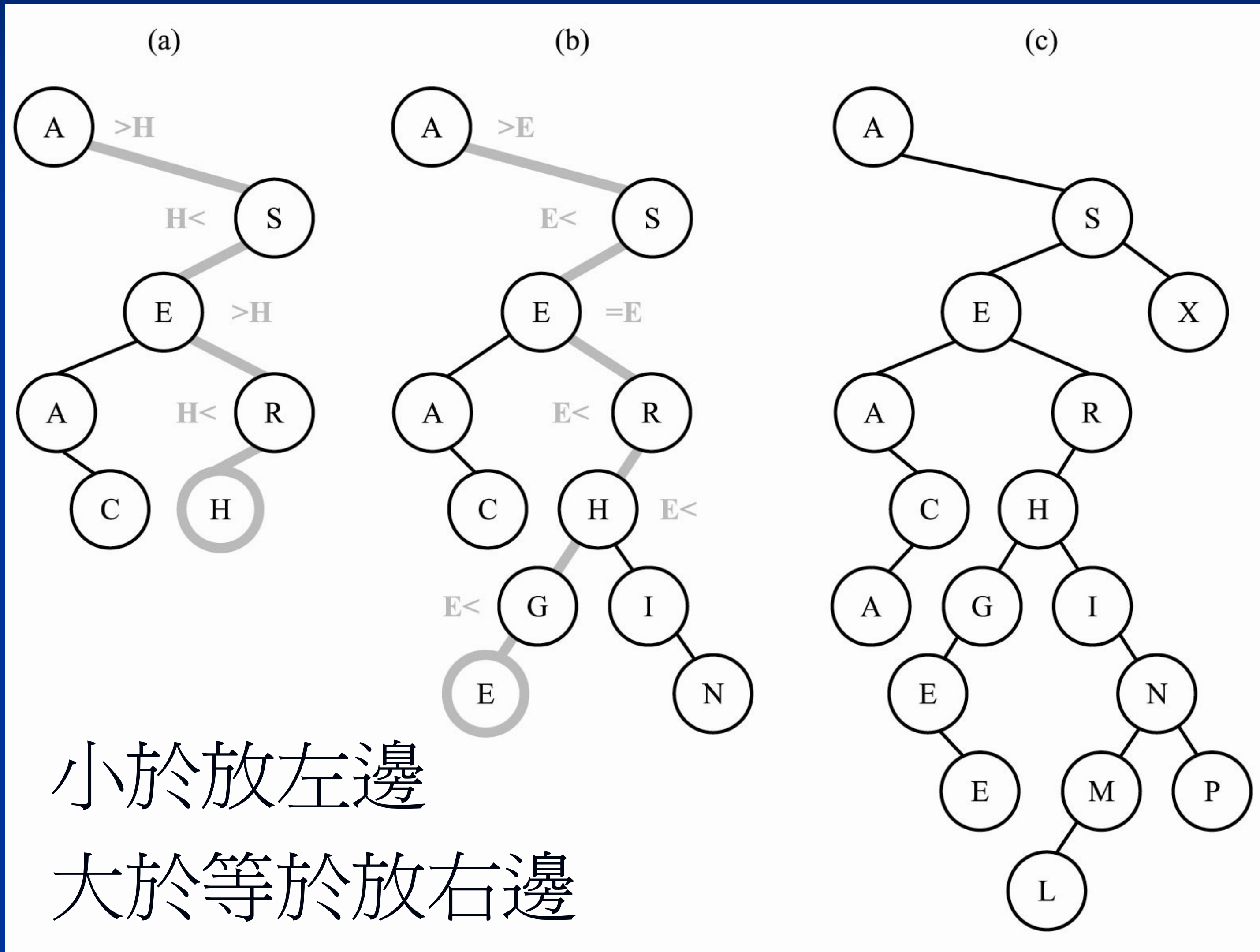
二元搜尋樹的新增

```
void insert_node(NODE *root, NODE *node){
    NODE *ptr, *parent;              //欲新增的節點

    ptr=root;
    while( ptr !=NULL) {
        parent = ptr;
        if (node->key < ptr->key) ptr=ptr->left;
        else ptr=ptr->right;
    }
    if (node->key < parent->key) parent->left=node;
    else parent->right=node;
}
```

小於放左邊

大於等於放右邊

- 建構二元搜尋樹 { A, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E } 。



二元搜尋樹的搜尋

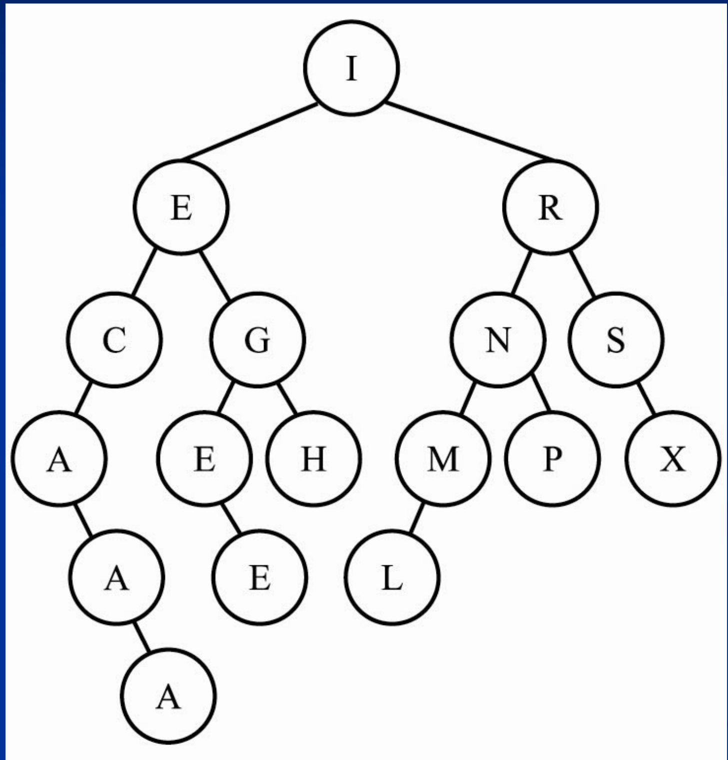
```
NODE*
binary_tree_search(NODE* root, int key){
   NODE *ptr;  ptr=root;
   while( ptr !=NULL ) {
        if ( key == ptr->key ) return ptr;      //搜尋成功
        if ( key < ptr->key ) ptr=ptr->left;
        else ptr=ptr->right;
   }
   return NULL;
}
```

小於放左邊

大於等於放右邊

- 建構二元搜尋樹

  { I, E, C, G, E, E, H, A, A, A, R, N, M, L, P, S, X }



- 輸入順序，樹形狀不同，二元搜尋樹的新增、搜尋時間複雜度將介於 $\Theta(\log n)$ 和 $\Theta(n)$ 之間

- 範例 7-3 將隨機陣列轉換爲二元搜尋樹後, 選擇隨機一個陣列值進行搜尋.

```
main(){ int n=10;    int A[n]; int k;   srand(time(0));
    NODE *ptr, *root;
    for(k=0; k<n; k++) {
        A[k]=rand()%n;    printf("%d ", A[k]);
    }       printf("\n");
    root=new_node(A[0]);
    for(k=1; k<n; k++)
        insert_node(root, new_node(A[k]) );
    k=rand()%n; printf("key=%d\n",A[k]);
    ptr=binary_tree_search(root, A[k] );
    if(ptr!=NULL)  printf("found %d", ptr->key);
}
```

測試

- n=10

  8 5 1 0 7 8 6 5 1 5

  key=1

  found 1

- n=15

  3 4 2 12 7 9 5 3 3 2 11 3 13 1 10

  key=4

  found 4

範例 7-4

二元搜尋樹的搜尋(遞迴)

```
NODE*
binary_tree_search(NODE* ptr, int key){
    if ( ptr == NULL ) return NULL;
    if ( key == ptr->key ) return ptr;
    if ( key < ptr->key ) ptr=ptr->left;
    else ptr=ptr->right;
    return binary_tree_search( ptr, key);
}
```

小於放左邊

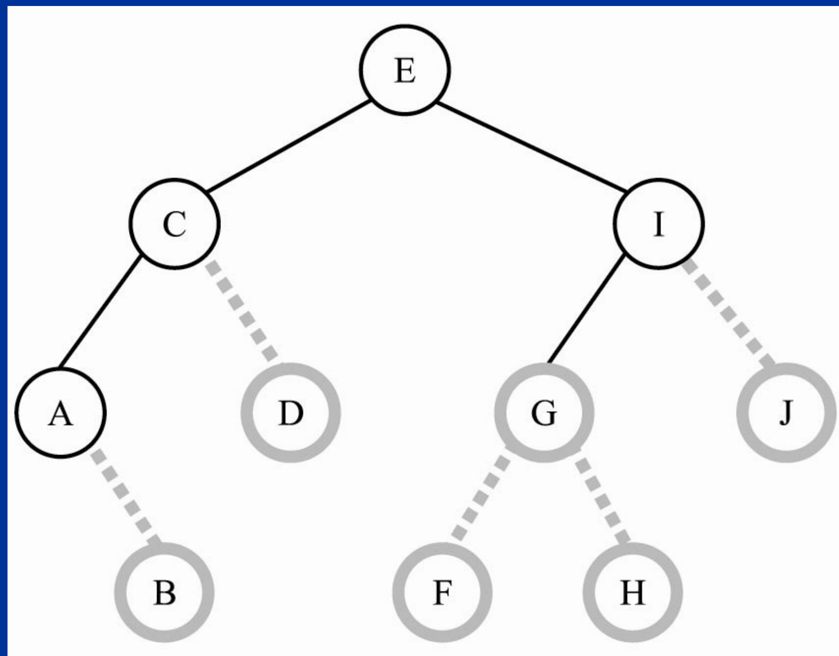
大於等於放右邊

刪除二元搜尋樹的節點

先搜尋該節點, 分成三種情況

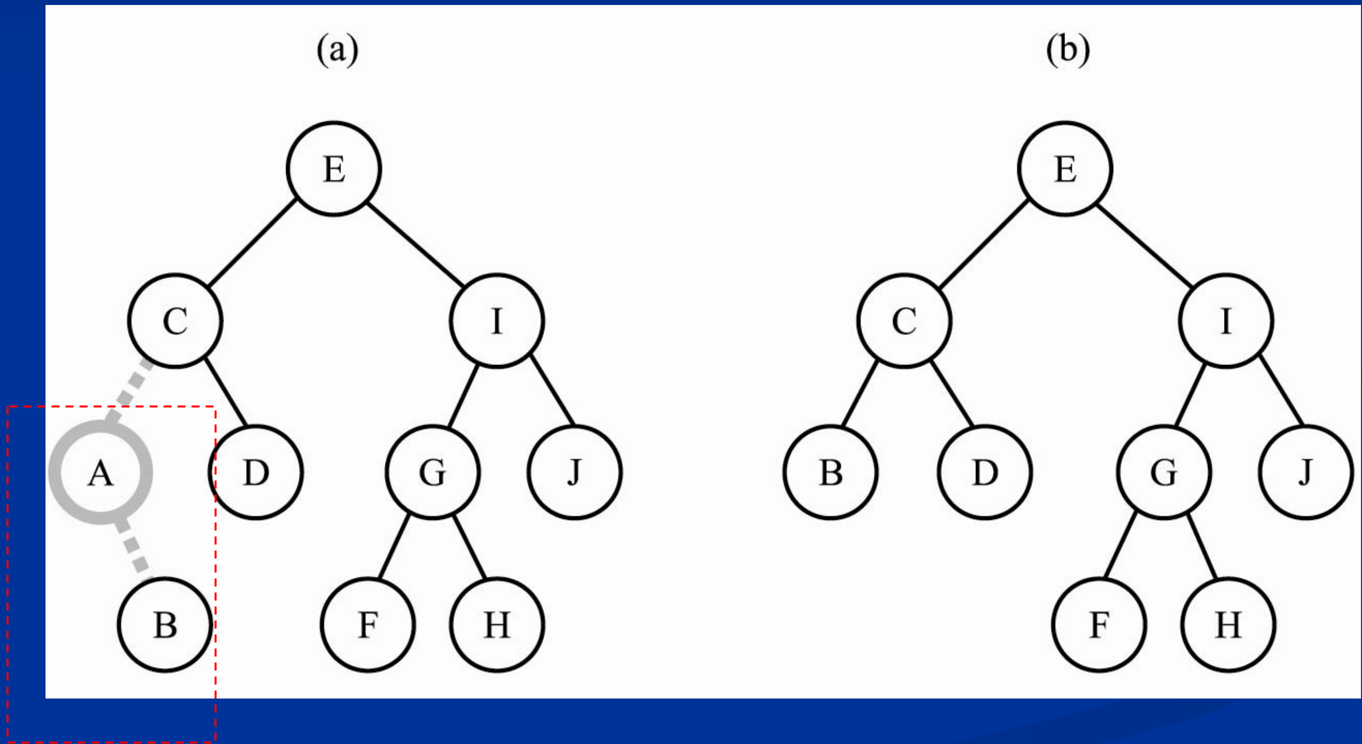
Case 1： 欲刪除的節點沒有子節點

設定**parent**指向此節點的指標為**NULL**



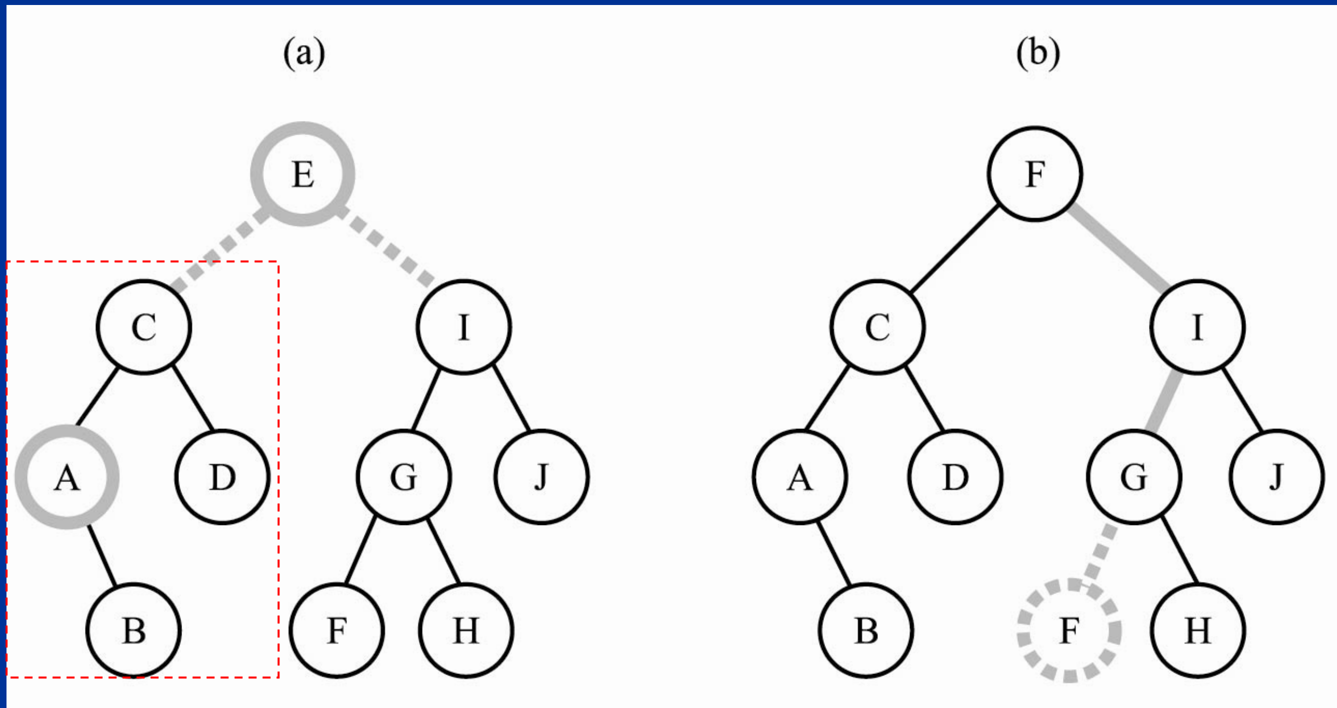
Case 2： 欲刪除的節點只有一個子節點

設定**parent**指向此節點的指標為此節點的子節點



Case 3： 欲刪除的節點具有兩個子節點

bad: 將左子樹所有節點重新加入右子樹
good: 將右子樹的最小節點取代欲刪除的節點



平衡樹 (balanced search tree)

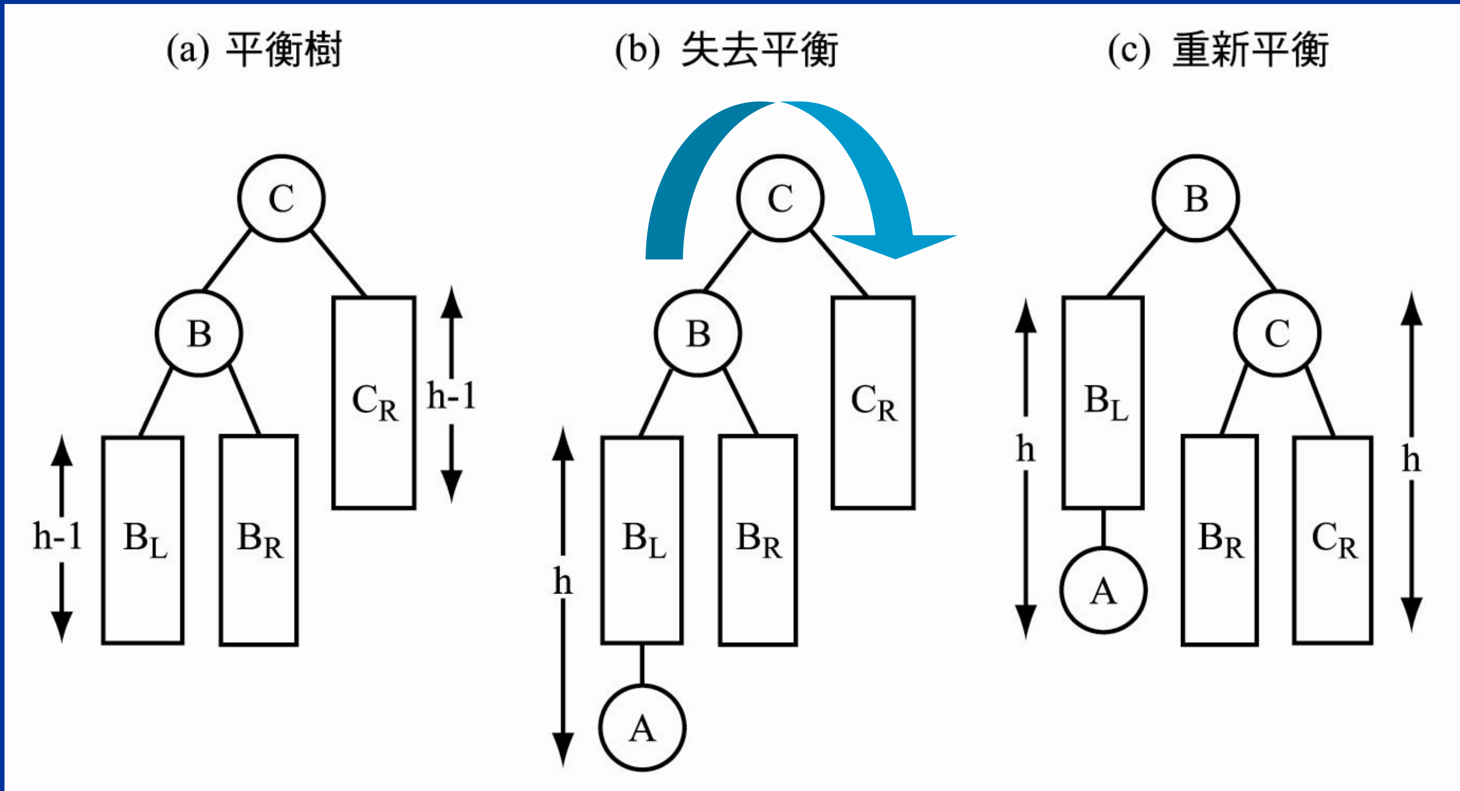
- 動態地調整樹狀結構以維持最低的高度
  $h \rightarrow [\log_2 n]$
  以保證其搜尋的效率爲 $O(\log n)$
- AVL 樹、紅黑樹、B 樹

AVL 樹 (Adelson-Velsky, Landis)

- 如果 T 是一顆非空的二元樹，那麼當T滿足以下條件時，T 則爲 AVL 樹：
  - 其左右子樹皆爲 AVL 樹
  - 左右子樹的高度相差小或等於 1
- 滿足 AVL 樹定義的二元搜尋樹稱爲 AVL 搜尋樹

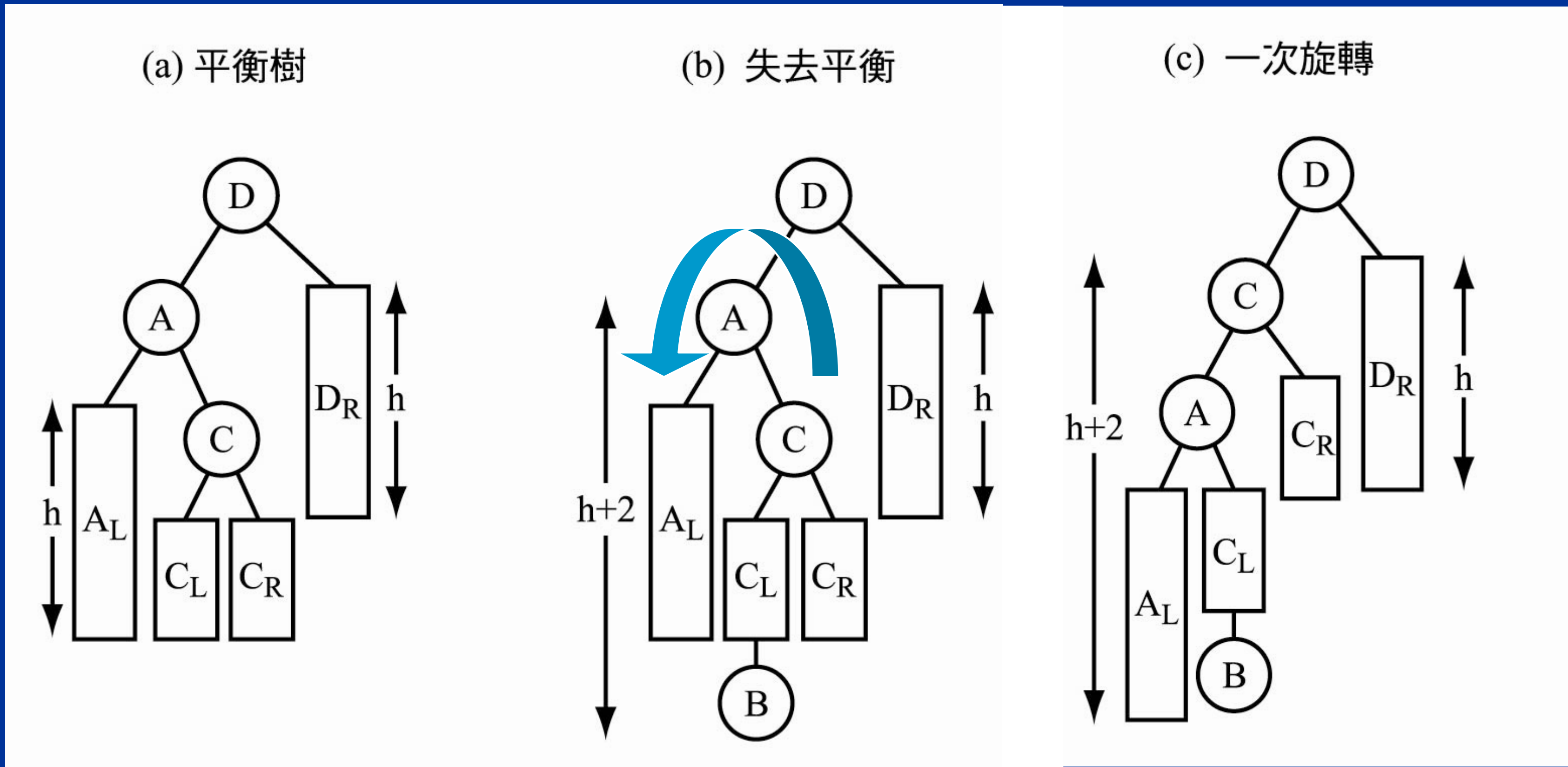
LL/RR 型不平衡

- 新節點插入到父節點其左子節點的左子樹
- 新節點插入到父節點其右子節點的右子樹

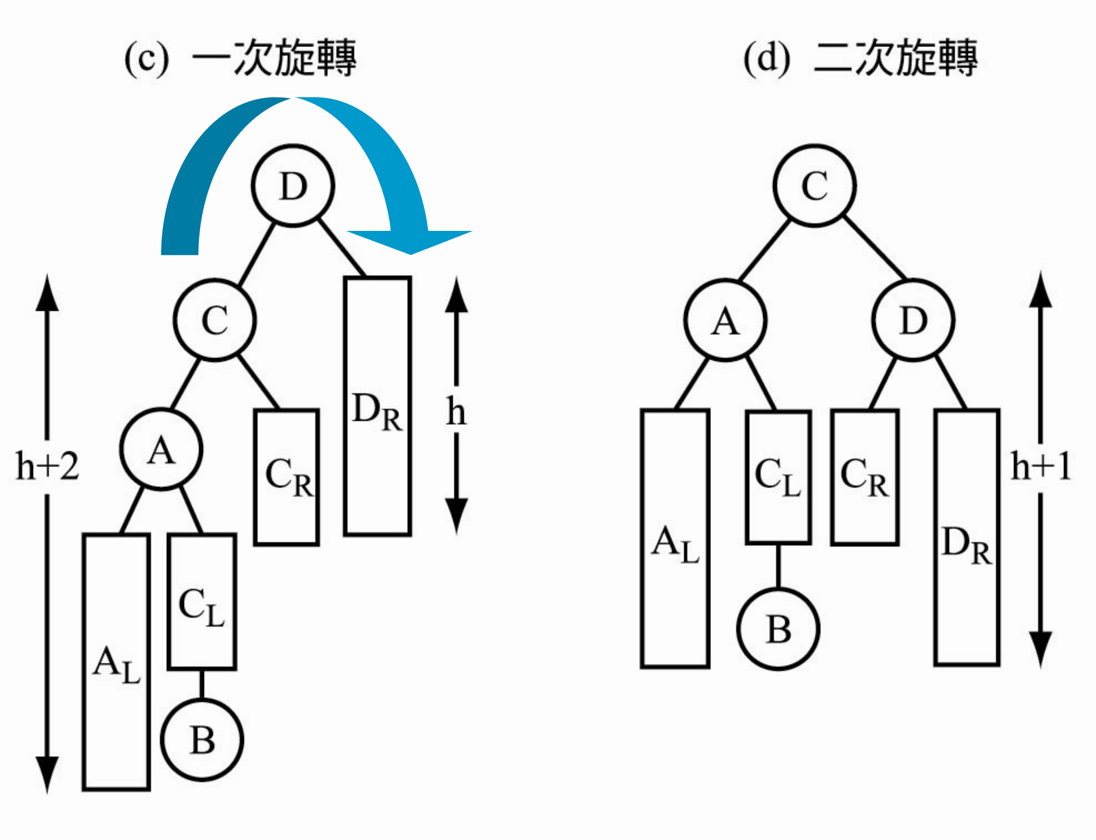


LR/RL型不平衡

- 新節點插入到父節點其左子節點的右子樹
- 新節點插入到父節點其右子節點的左子樹



LR/RL型不平衡



AVL 樹分析

- 樹高

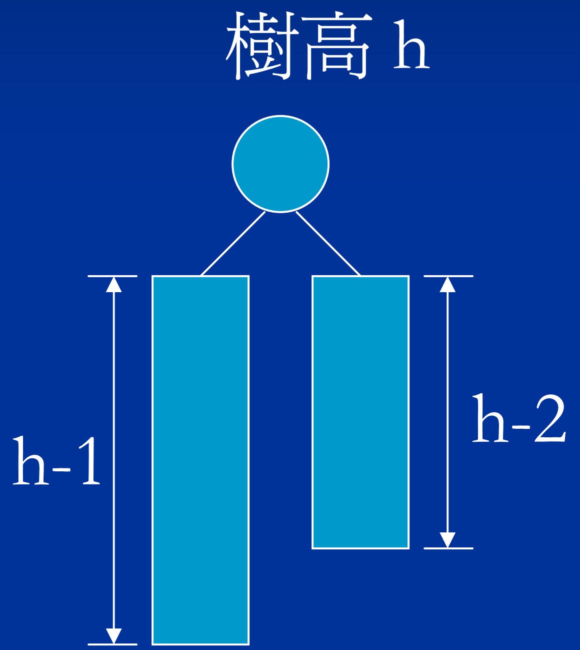
  $$\lceil \log_2 n \rceil \le h \le h_{max}$$

  $$h_{max} \sim 1.44 \log_2 n$$

- 搜尋、新增或刪除的效率為 O(log n)
- 缺點：旋轉過多 →降低效能

  需要個別紀錄子樹高

AVL 樹分析



節點數目 $F(h)=F(h-1)+F(h-2)+1$

$\{F(h)+1\}=\{F(h-1)+1\} + \{F(h-2)+1\}$

$F(0)=1 \quad F(1)=2 \quad F(h)=n$

F(h)+1 爲 Fibonacci number

$F(h)+1=n+1=Ka^h$

$a \sim 1.618$

$h = \log_a(n+1)-\log_a K$

$h \sim 1.44 \log_2 n \qquad 1/\log_2 a \sim 1.44$

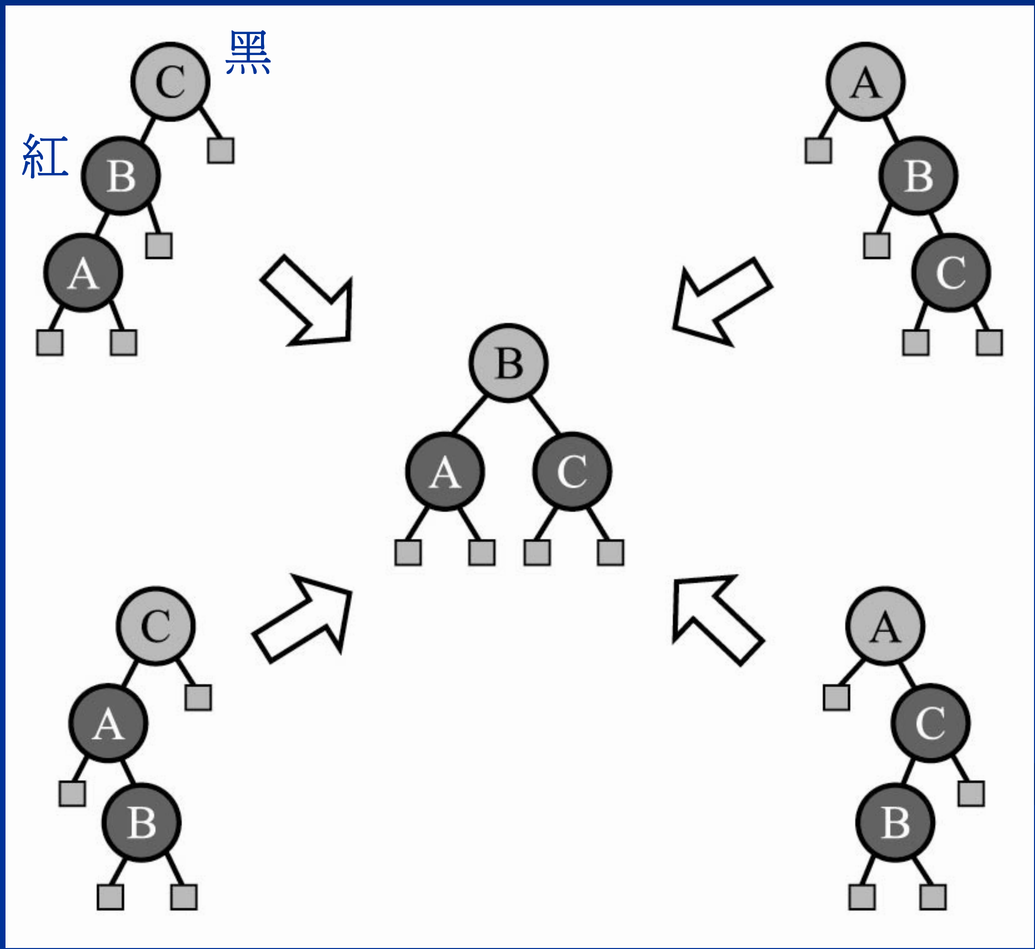
紅黑樹 (red-black tree)

- 紅黑樹的特性：
  - 節點分成紅色、黑色
  - 黑色：根節點, 葉節點(此處指空節點)
  - 從根節點到樹葉的路徑都有相同數目的黑節點
  - 不能有連續兩個以上的紅節點

- 樹高可為最短子樹高的兩倍
  →降低新增或刪除的旋轉次數, 以提升效能

紅黑樹的建構與旋轉

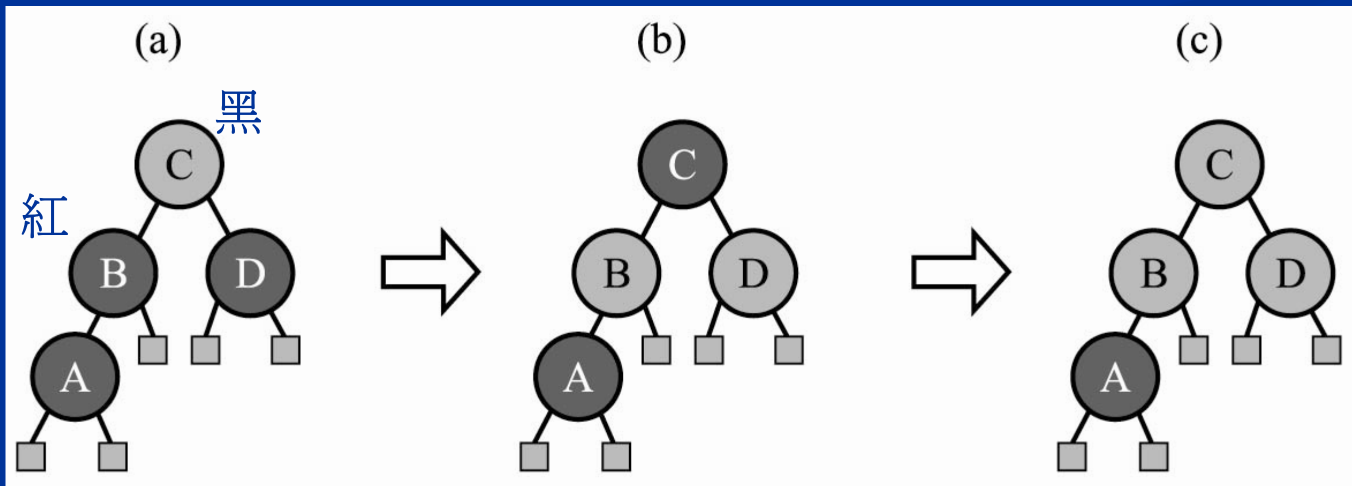
- 紅黑樹的根節點和所有的葉節點(此處指空節點)都是黑色
- 每個新加入紅黑樹的節點都是紅色的
- 父子兩個節點都是紅色的情形，利用旋轉來避免



紅黑樹的建構與旋轉

- 當使用旋轉也不能避免連續兩個紅色節點
  →互換顏色

- 如果根節點被換成紅色時，就要重新塗成黑色



C和{B, D}互換顏色　根節點C必須為黑色

紅黑樹分析

- 樹高 h

  $n \leq 2^{h/2} - 1$ (樹高可為最短子樹高的兩倍)

  $h \leq 2\log_2(n+1)$

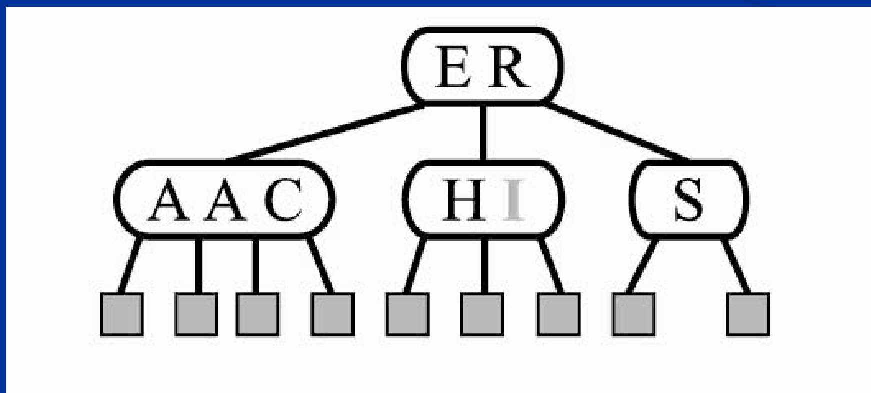
- 容許樹高差距大，旋轉次數較少
  →新增、刪除效能較 AVL 樹高

- 新增、刪除、搜尋時間複雜度 O(log n)

B 樹 (B-tree)

- B樹允許多個鍵值存放在一個節點
  每個節點內的鍵值都經過排序

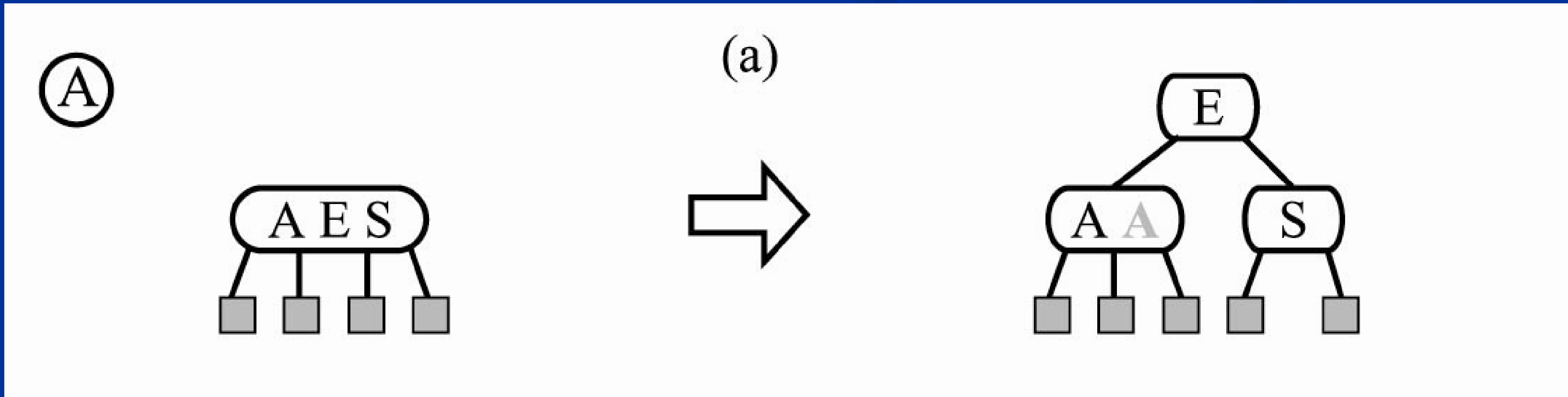
- 如果鍵值數目達到上限，就分裂成兩個
  →保證所有葉節點都處於同一個高度



B樹的建構

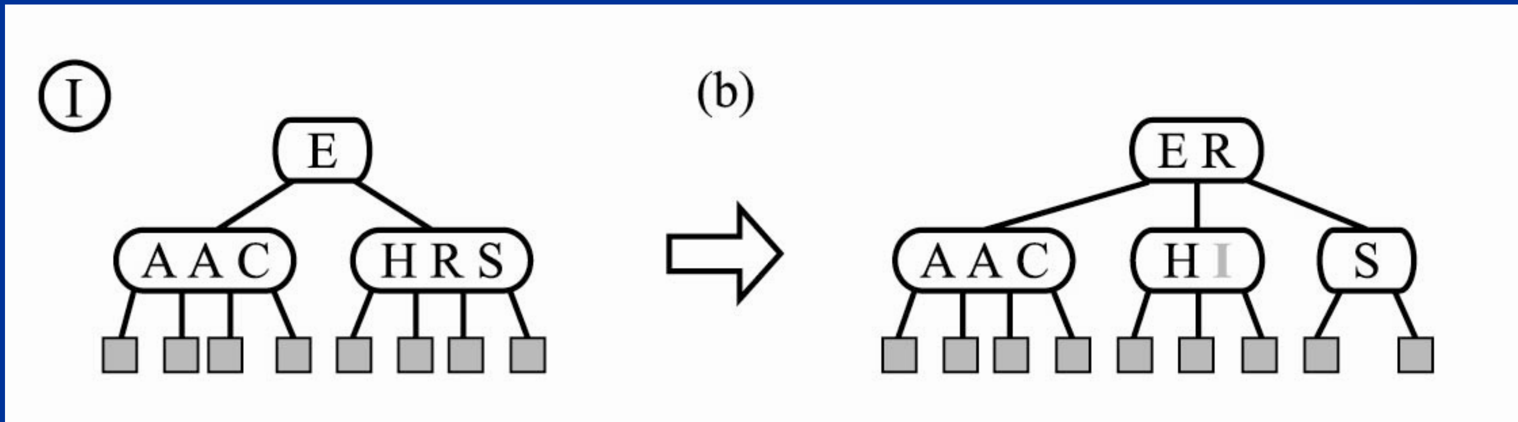
- 階(order) : B 樹節點擁有的最多子節點數目 m
- 4 階 B 樹: 每個節點最多有 3 個鍵值、4 個子節點 又稱爲 2-3-4 樹 (2-3-4 tree)

輸入{ A, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E }



B樹的建構

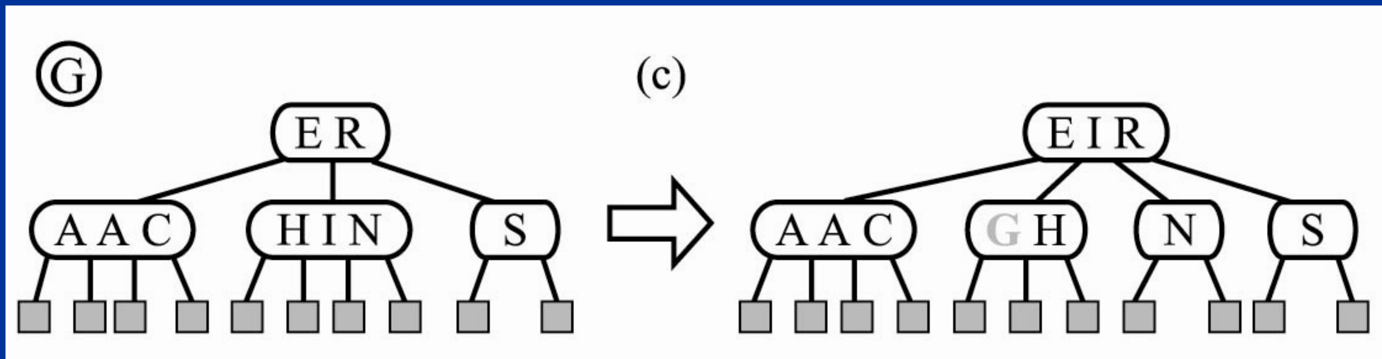
- {R, C, H} 先插入至子節點
- 插入 I 後，{H, I, R, S} 分裂成{H, I}和{S}
  中間值 R 插入父節點



輸入{ A, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E }

B樹的建構...

- 插入 G 後，{G, H, I, N} 需要分裂成 {G, H} 和 {N}
  中間值 I 則向上插入父節點



輸入{ A, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E }

B 樹分析

- 樹高 h

  $h \leq [\log_2 n]$

- 新增、刪除、搜尋時間複雜度 $O(\log n)$

- 適用於有大量資料、須使用硬碟的資料庫