Review of Data Structures

Algorithms

Data Strcuctures +

Algorithms +

Coding Skills =

Programming

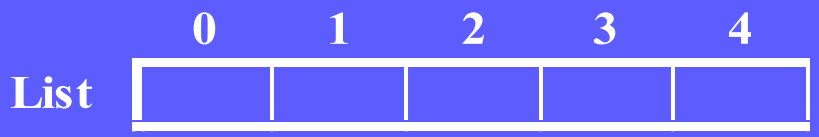
有哪些資料結構 ？

Data Structures

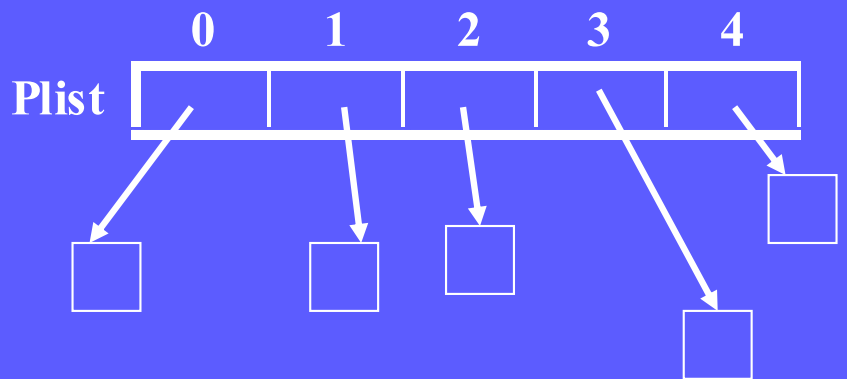
- Array
- Linked List
- Stack
- Queue
- Tree
- Binary Search Tree, AVL Tree, B-Tree, Red-Black Tree
- Heap
- Hashing (Dictionary)
- Graph

Arrays in C Language

- **declaration**
  - **int list[5]: list[0], …, list[4] each contains an integer**



  - **int *plist[5]: plist[0], …, plist[4] each contains a pointer to an integer**



Linked List

- **A linked list**

- **Linked list with actual pointer values**

Importance

Given all the issues working against the team, you might think it was hard to identify a single large source of bugs, but based on my experiences the biggest problems in StarCraft related to the use of doubly-linked linked lists. Linked lists were used extensively in the engine to track units with shared behavior. With twice the number of units of its predecessor - StarCraft had a maximum of 1600, up from 800 in Warcraft 2 - it became essential to optimize the search for units of specific types by keeping them linked together in lists. Recalling from distant memory, there were lists for each player's units and buildings, lists for each player's "power-generating" buildings, a list for each Carrier's fighter drones, and many many others. All of these lists were doubly-linked to make it possible to add and remove elements from the list in constant time - O(1) - without the necessity to traverse the list looking for the element to remove - O(N).

由於橫互在開發團隊面前的所有問題，你可能認為在龐大的程式碼中很難找出臭蟲，但是 根據我的經驗，星海爭霸最大的問題其實來自於使用了雙向鏈結串列。 鏈結串列在引擎中被廣泛地始用，用來記錄單位及其共享行為。魔獸爭霸2的單位數目有 800個，但星海爭霸最多有1600個，整整增加了一倍。這使得我們必需對尋找特定類型的單位這件事進行優化，方法是將他們鏈結在一起存放於串列中。 回顧遙遠的記憶，有很多用來儲存玩家的單位及建築的串列，很多用來儲存每個玩家“發 電”建築的串列，一個儲存每一架航空母艦無人戰鬥機的串列，以及其他很多很多其他的串 列。 所有的串列都是雙向鏈結的，使得我們可以在常數時間(O(1))裡新增及移除節點，而不需要搜尋整個串列找到欲移除的節點(O(N))。

不幸的是，每個串列都是"手工維護"的 - 沒有共用的函式去鏈結及取消鏈結這些串列，程 式設計師全憑手動在需要鏈結及取消鏈結的地方加入行內函式。手工程式碼遠比使用已經 過除錯的常式來的容易出錯。 有些鏈結欄位是由多個串列共享，所以必須要確切知道這個物件被哪些串列鏈結，好安全 地移除鏈結。一些鏈結欄位使用C語言的union型別儲存，可以最小化記憶體的使用

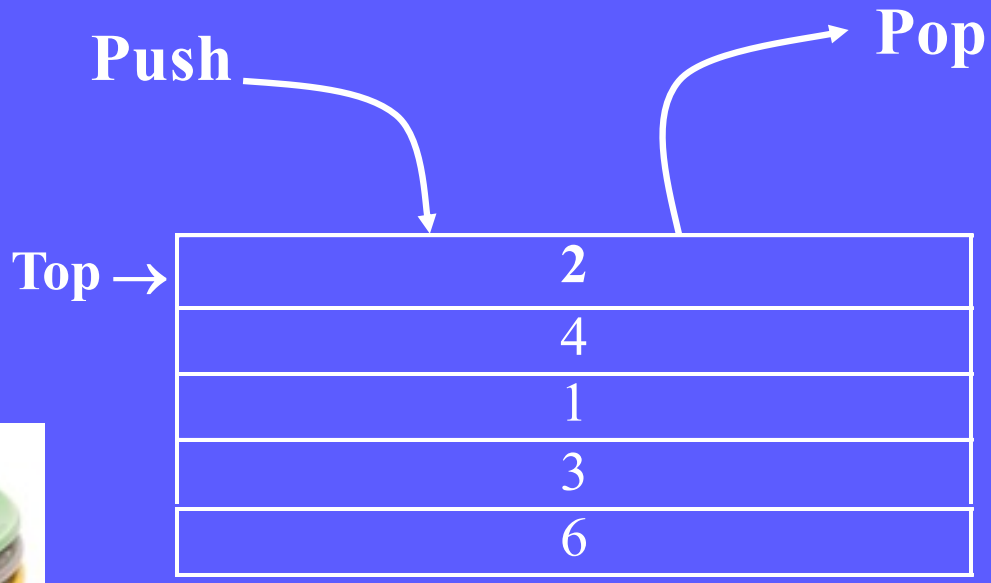
關於製作星海爭霸有多困難這件事，你已經聽我大吐苦水，很大一部分是因為公司的每一 個層級對於遊戲方向、科技以及設計，作了錯誤的選擇。

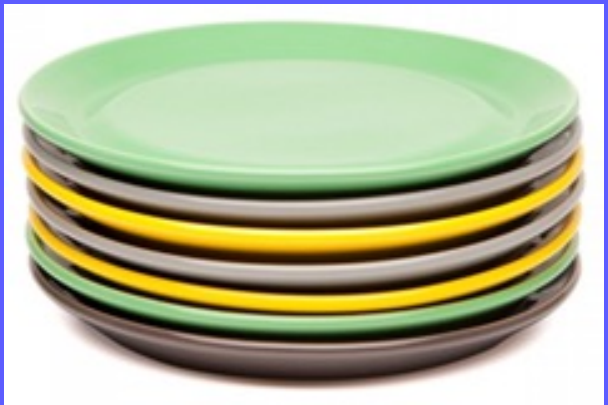
我們很幸運我們是一支鐵頭又強悍的團隊，且我們的洞察力讓我們贏得最後的勝利。在收 尾階段，我們繼續埋頭苦幹，並停止增加新功能，確定遊戲時間夠長，可以推出了。玩家 絕不會發現底下發生的恐怖事情，或許這是編譯式語言另一個勝過像是Javascript這種腳 本語言的地方 - 終端使用者絕不會看見火車失事！在這篇文章的第二部分，我將會談論更多技術性的話題，並講述為什麼大部份的程式設計 師總是把鏈結串列搞砸了，接著提出成功使用於暗黑破壞神、battle.net以及激戰的另一 種解決方案。

即使你不使用鏈結串列，相同的解決方案甚至會使用更複雜的資料結構，像是雜湊表、 B樹以及優先佇列。我相信基本思想足以概括所有的程式設計面向。但讓我們慢慢來，那屬 於另一篇文章的內容。

The Stack ADT

- **Stack: Last In First Out (LIFO)**
- **Operations**
  - **Push**
  - **Pop**





如何以Linked List 實作Stack ？

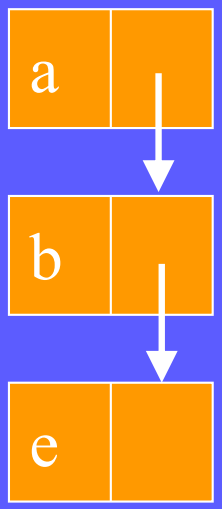
(請以先後push e, b, a

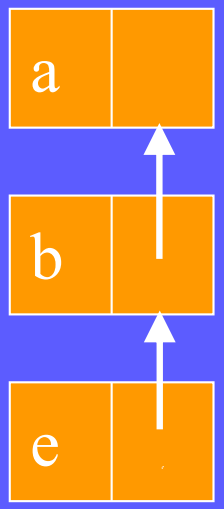
為例繪圖說明Stack的push)

如何以Linked List 實作Stack ？

(請以先後push e, b, a

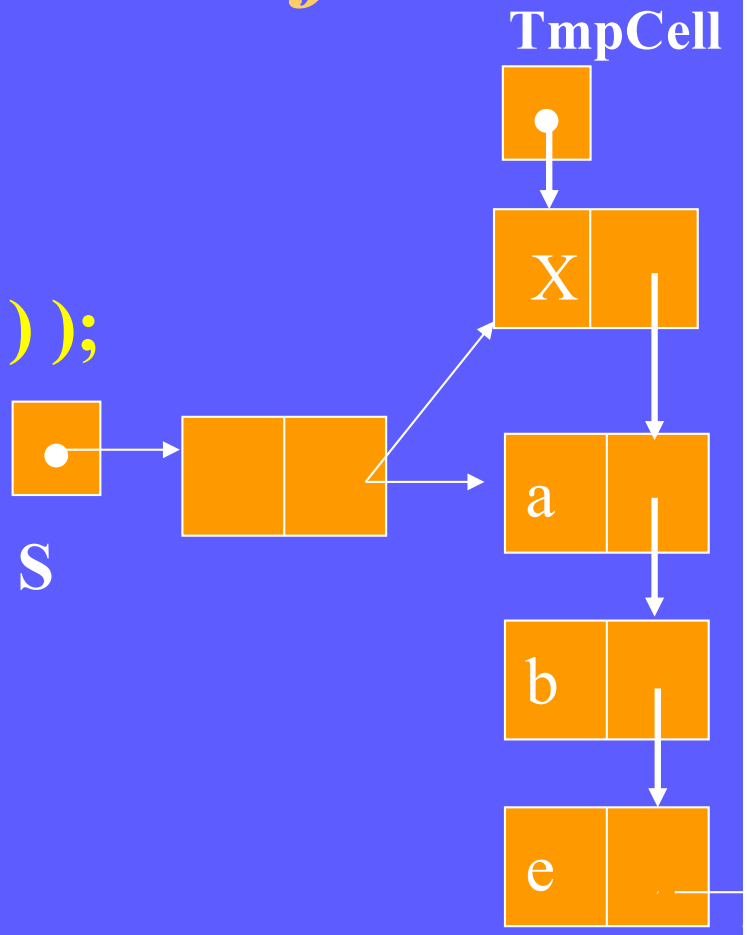
為例繪圖說明Stack的push)





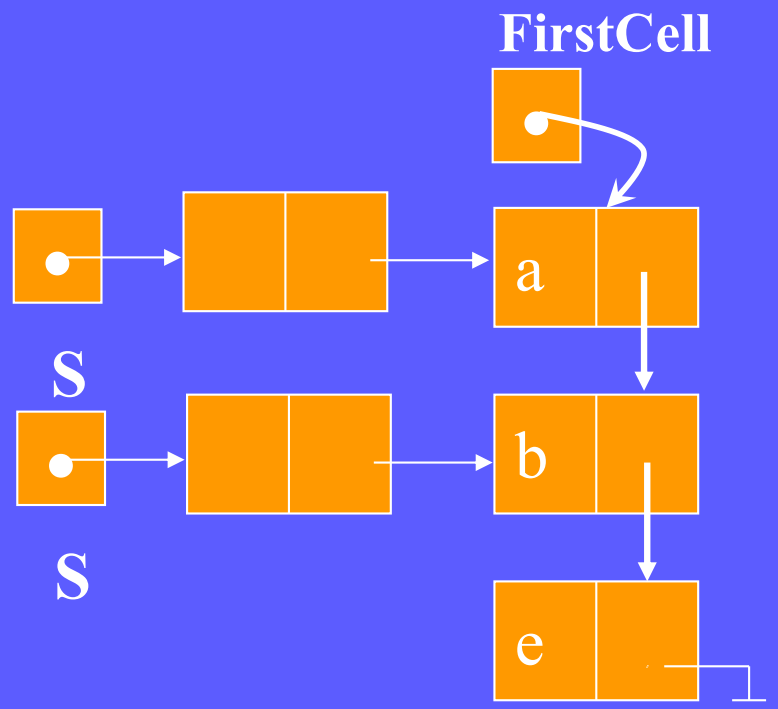
Linked List Implementation of Stack

```
void  Push( ElementType X, Stack S )
{   PtrToNode TmpCell;
    TmpCell = malloc( sizeof( struct Node ) );
    if ( TmpCell == NULL )
       FatalError( "Out of space!!!" );
   else
  {     TmpCell->Element = X;
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
  }
}
```



Linked List Implementation of Stack (Cont.)

```
void  Pop( Stack S )
{   PtrToNode FirstCell;
    if   ( IsEmpty( S ) )
         Error( "Empty stack" );
    else
    {    FirstCell = S->Next;
         S->Next = S->Next->Next;
         free( FirstCell );
    }
}
```



Christos Papadimitrou and Bill Gates Flip Pancakes

Sorting By Reversals: Example

- **Goal: Given a permutation, find a shortest series of reversals that transforms it into the identity permutation (1 2 … n)**

- **Example :**

input: π = 3 4 2 1 5 6 7 10 9 8

output: 4 3 2 1 5 6 7 10 9 8

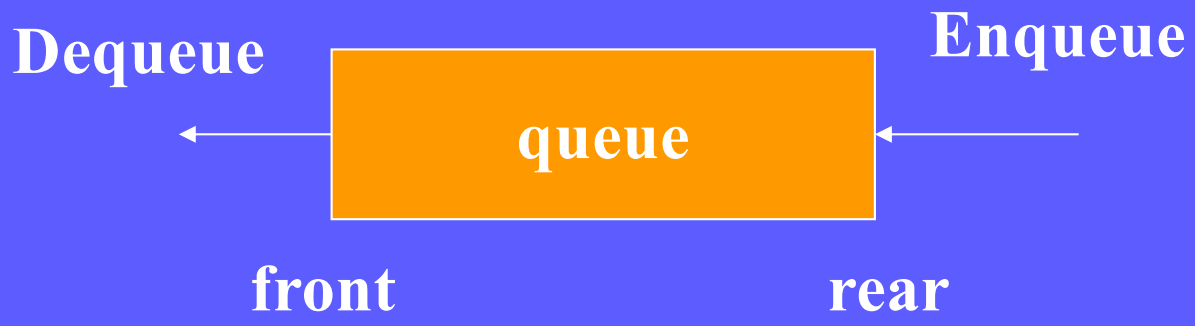
4 3 2 1 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

So $t = 3$

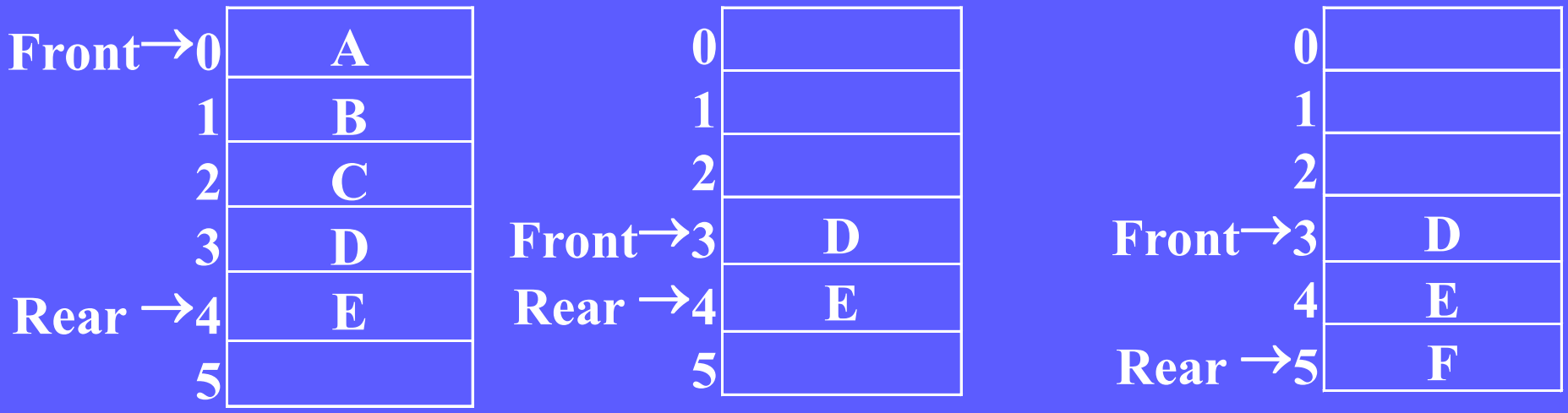
The Queue ADT

- **Queue: First In First Out (FIFO)**
- **Operation**
  - **Enqueue at rear (end)**
  - **Dequeue at front (start)**

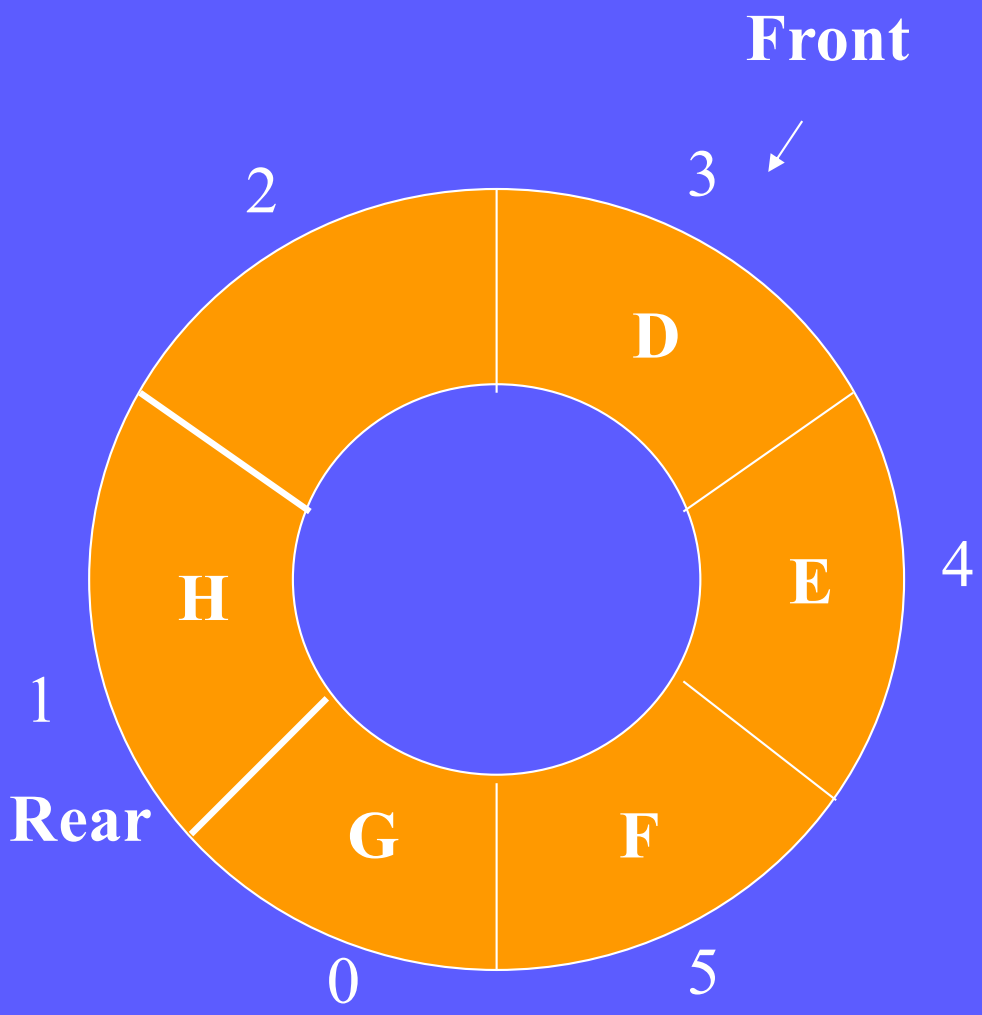


Array Implementation of Queue

- **Problem: queue appears to be full, but actually not**
- **Solution: circular queue**



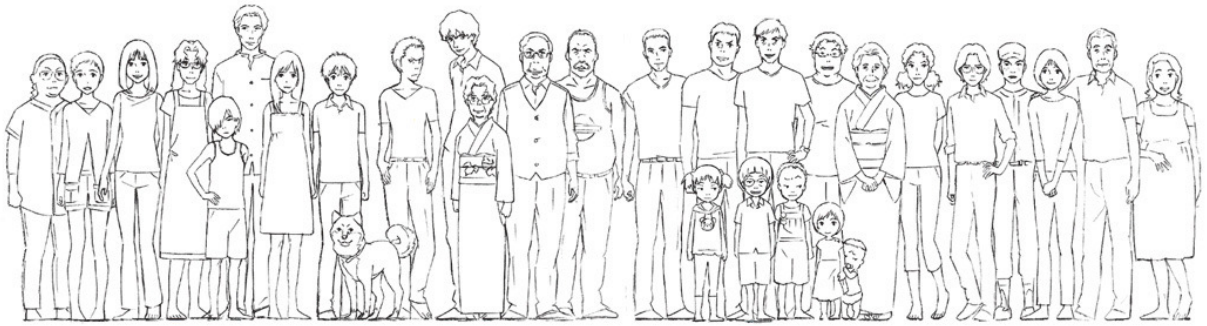
Circular Queue



0	G
Rear → 1	H
2	
Front → 3	D
4	E
5	F

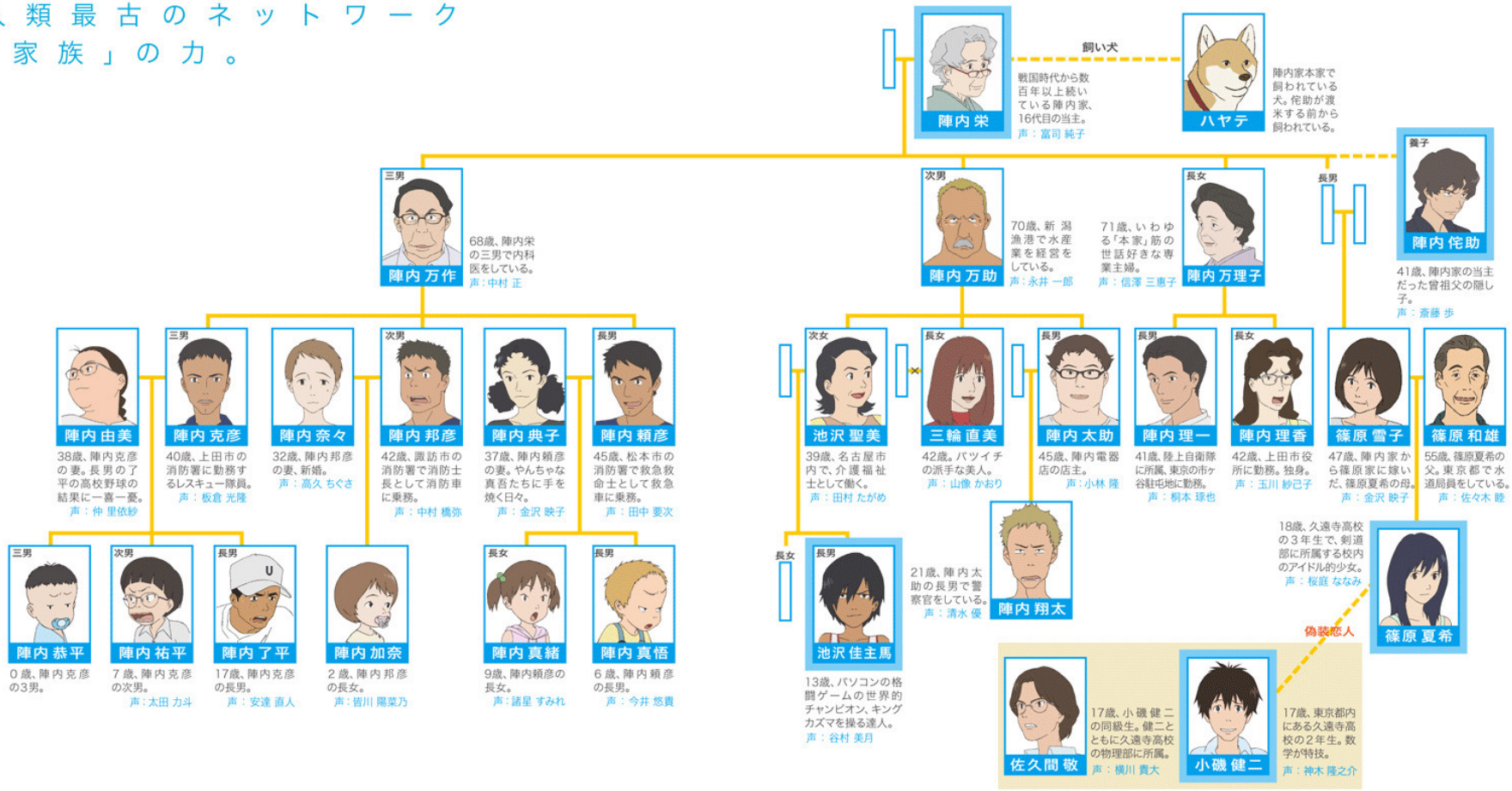
Tree

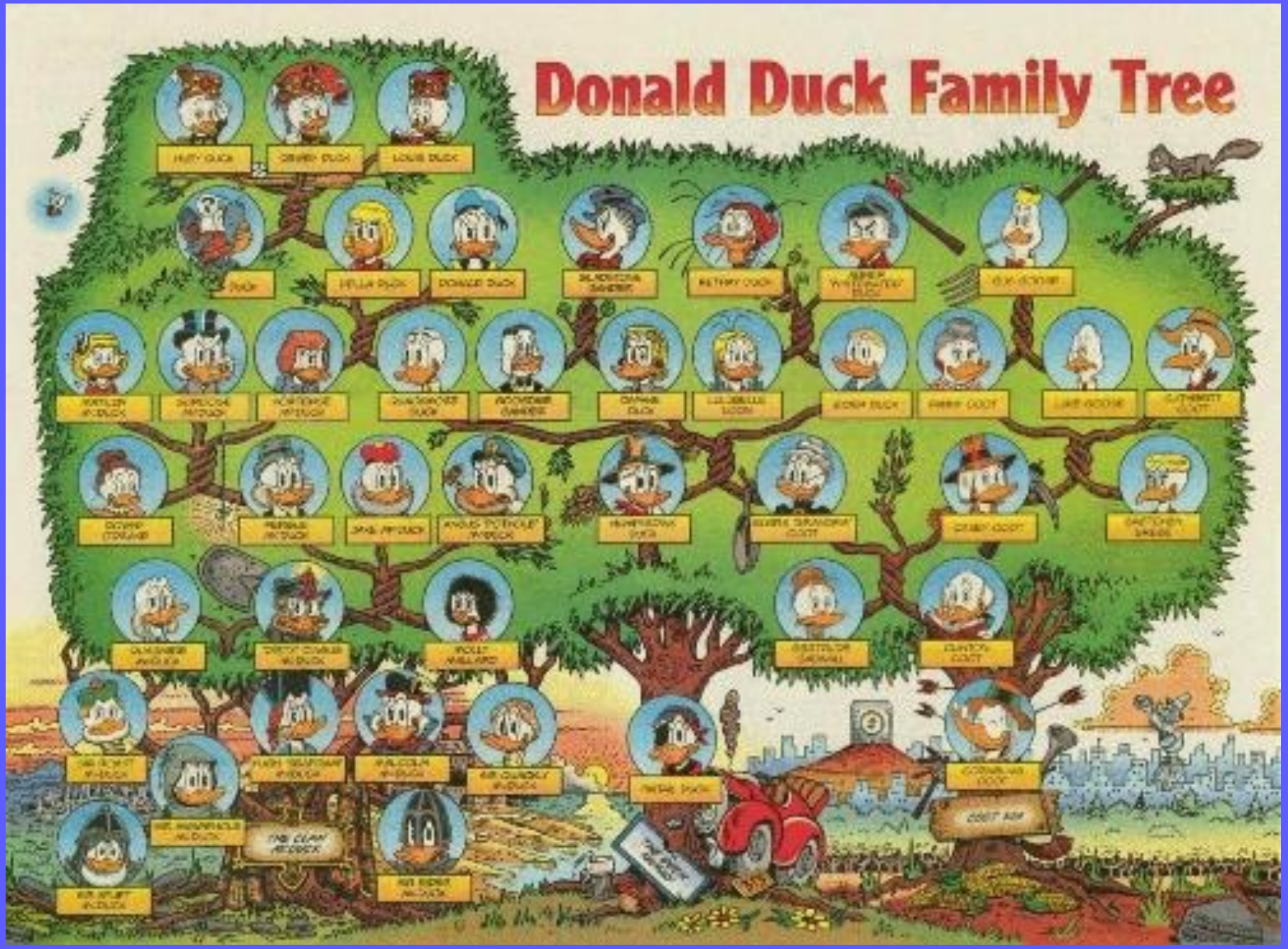

伊丽莎白二世
（英国女王）
生于1926年
1947年结婚
飞利浦亲王
（爱丁堡公爵）
生于1921年
查尔斯王储
（威尔士亲王）
生于1948年
安妮长公主
生于1950年
安德鲁
（约克公爵）
生于1960年
爱德华
（埃塞克斯伯爵）
生于1964年
1981年结婚
1996年离婚
戴安娜王妃
1961—1997
1973年结婚
1992年离婚
马克菲利普斯
生于1948年
1986年结婚
1996年离婚
莎拉·弗格森
生于1965年
1999年结婚
Sophoe
Rhys Jones
生于1965年
威廉王子
生于1982年
哈里王子
生于1984年
皮特菲利普斯
生于1977年
扎拉菲利普斯
生于1981年
比阿特丽斯公主
生于1988年
欧吉妮公主
生于1990年
路易斯温莎小姐
生于2003年
瑟玟子爵
生于2007年
2011年4月29日
结婚
凯特·米德尔顿
生于1982年
2008年结婚
奥特姆·凯莉
生于1978年
2005年结婚
卡米拉王妃殿下
生于1947年
1992年结婚
蒂姆·劳伦斯
生于1955年
凤凰網 資訊
凤凰网资讯中心专题组出品



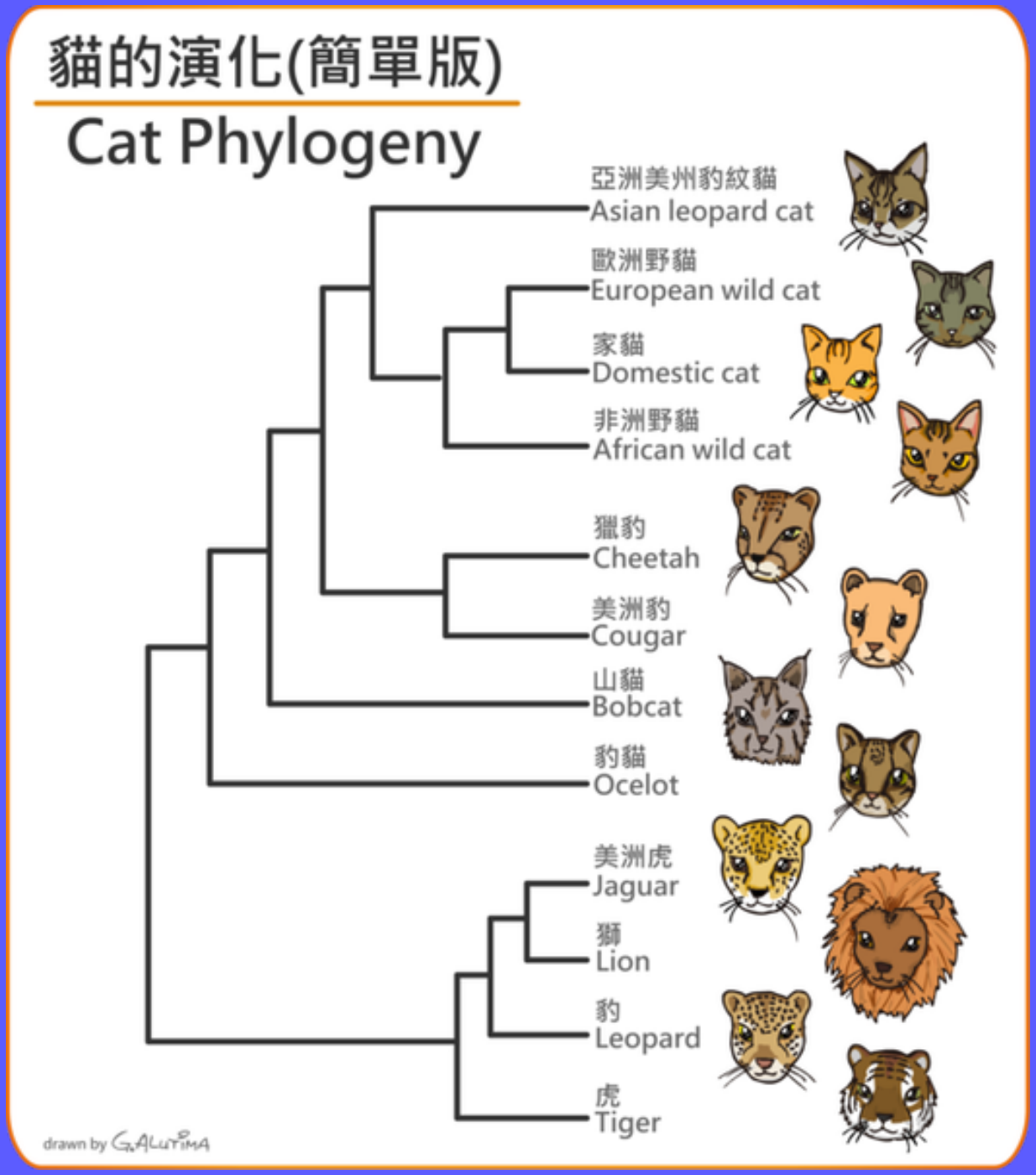
SUMMER WARS FAMILY TREE

人類最古のネットワーク
「家族」の力。





M. K. Shan, CS, NCCU


貓的演化(簡單版)
Cat Phylogeny
亞洲美州豹紋貓
Asian leopard cat
歐洲野貓
European wild cat
家貓
Domestic cat
非洲野貓
African wild cat
獵豹
Cheetah
美洲豹
Cougar
山貓
Bobcat
豹貓
Ocelot
美洲虎
Jaguar
獅
Lion
豹
Leopard
虎
Tiger
drawn by GALUPIMA

Binary Trees

- **Binary Trees:**

  a tree in which no node can have more than two children

- **Properties of binary tree**
  - **depth of an average binary tree is considerably smaller than N**
  - **binary search tree (special type): O(logN)**
  - **worst case: O(N) (degenerate to linear list)**

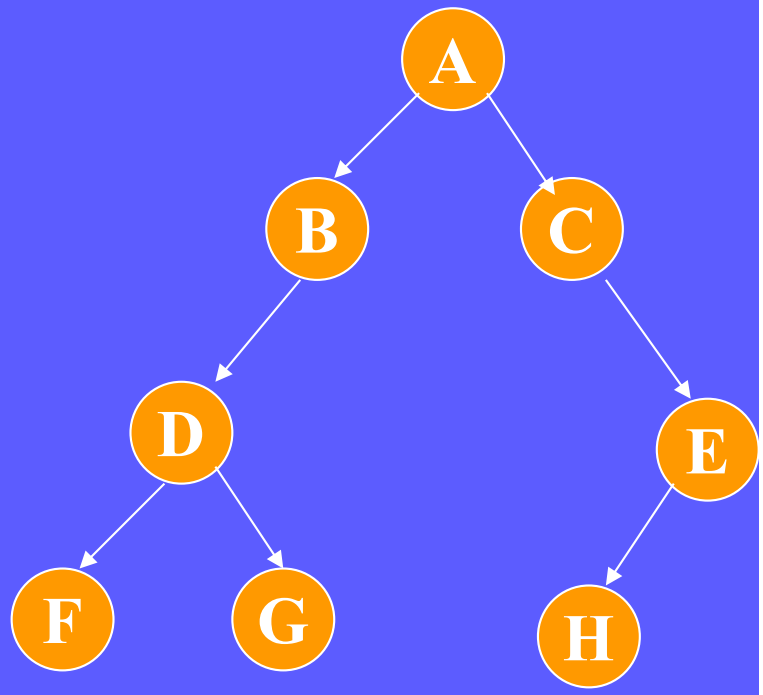
Implementation of Binary Trees

- **Pointer implementation**

```
typedef struct TreeNode *PtrToNode;
typedef struct PtrToNode Tree;
struct treeNode
{
  ElemenType      Element;
  Tree            Left;
  Tree            Right;
}
```

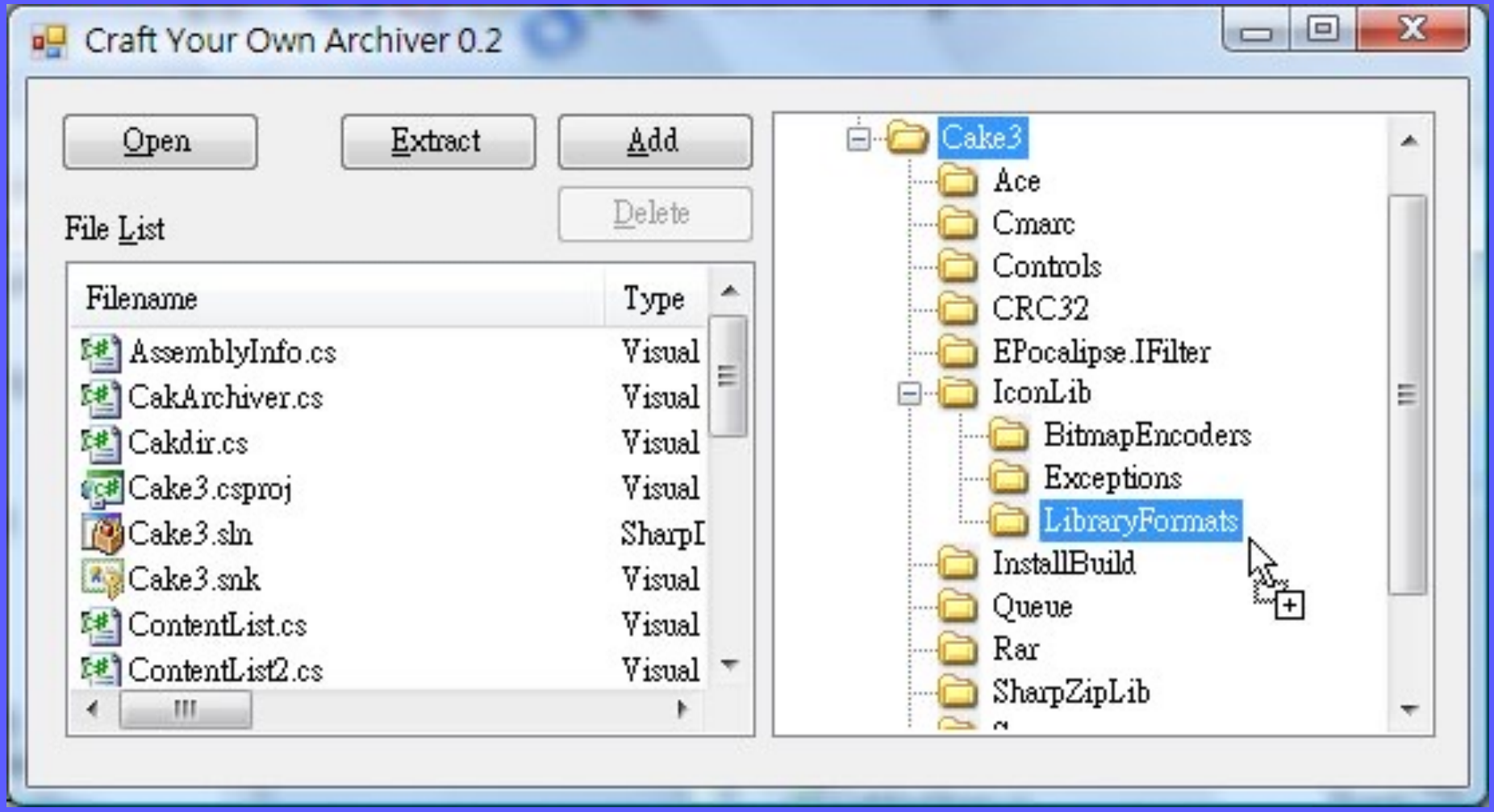
Implementation of Binary Trees

- **Array implementation**
  - **parent(i): ⌊i/2⌋**
  - **left(i): 2i**
  - **right(i): 2i+1**



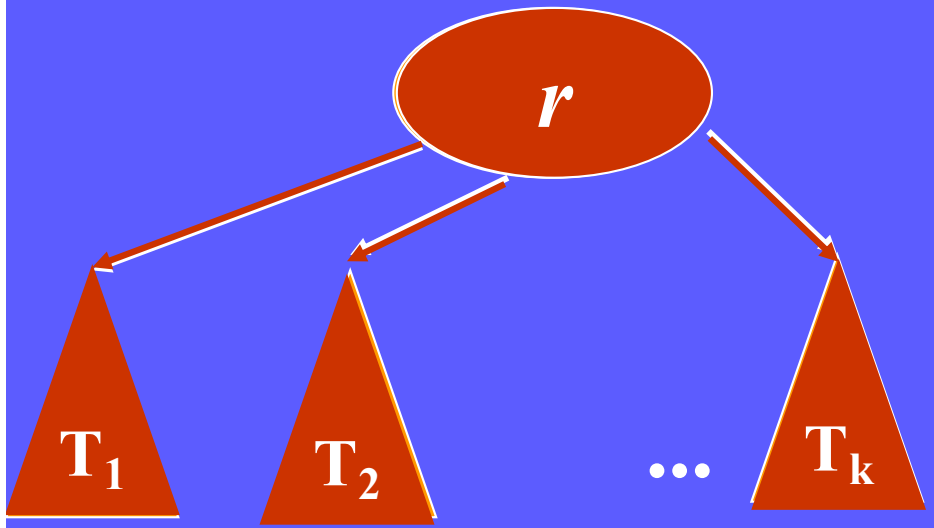
A	B	C	D			E	F	G		H
1	2	3	4	5	6	7	8	9	…	14

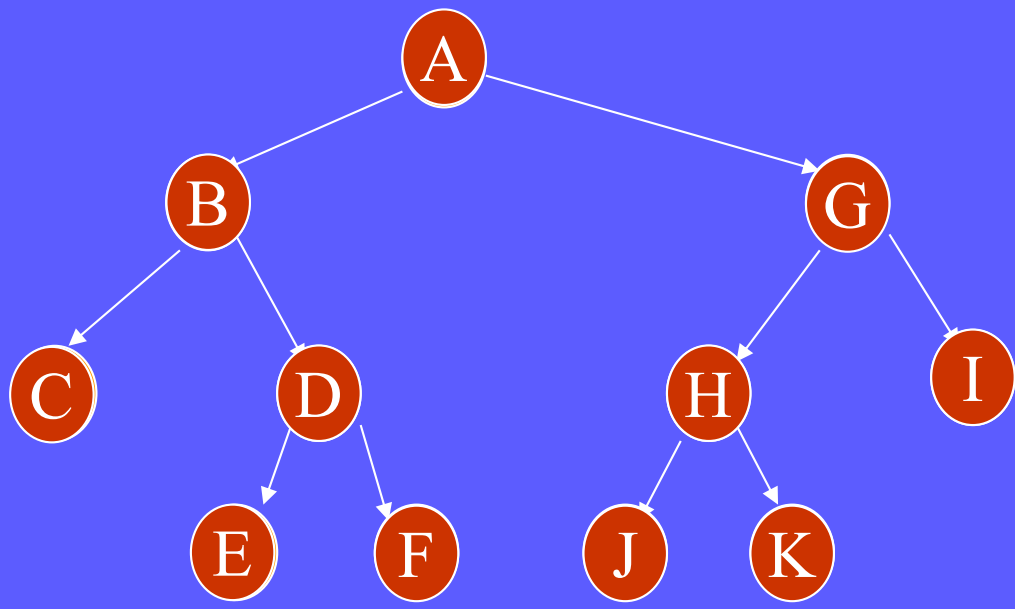
Tree Traversal


Craft Your Own Archiver 0.2
Open
Extract
Add
Delete
File List
Filename
Type
AssemblyInfo.cs
CakArchiver.cs
Cakdir.cs
Cake3.csproj
Cake3.sln
Cake3.snk
ContentList.cs
ContentList2.cs
Visual
SharpI
Cake3
Ace
Cmarc
Controls
CRC32
EPocalipse.IFilter
IconLib
BitmapEncoders
Exceptions
LibraryFormats
InstallBuild
Queue
Rar
SharpZipLib

Tree Traversals-preorder

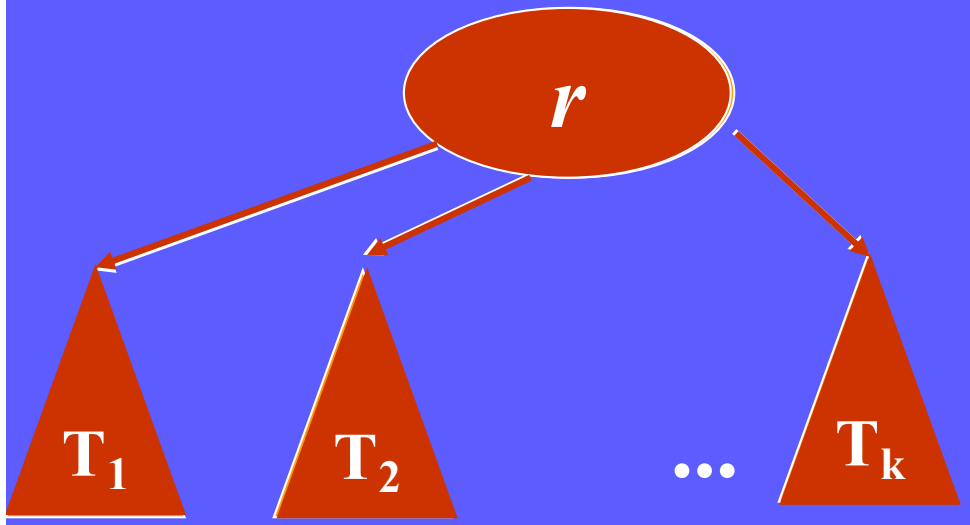
- **Pre-order (Root → Left Subtree → Right Subtree)**

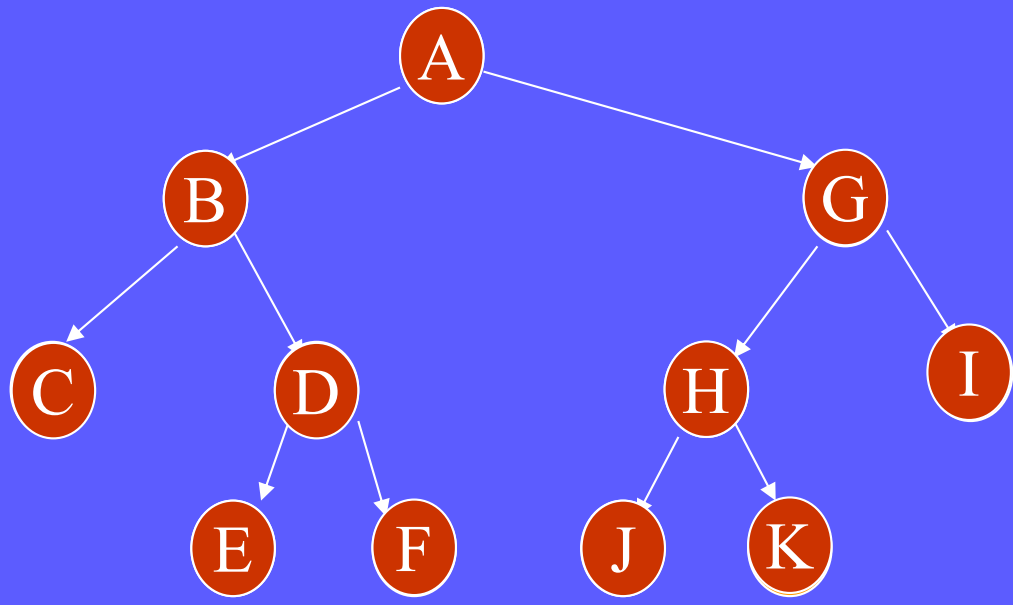




Tree Traversals-inorder

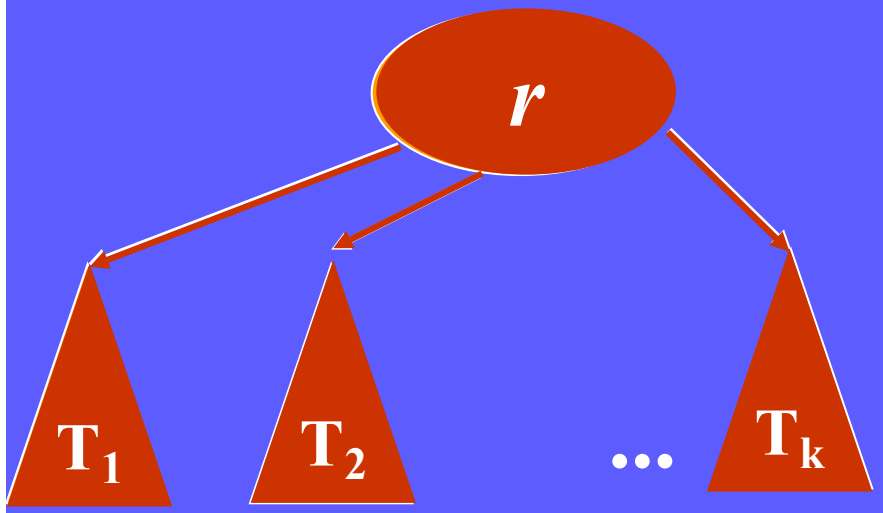
- **In-order (Left Subtree → Root → Right Subtree)**

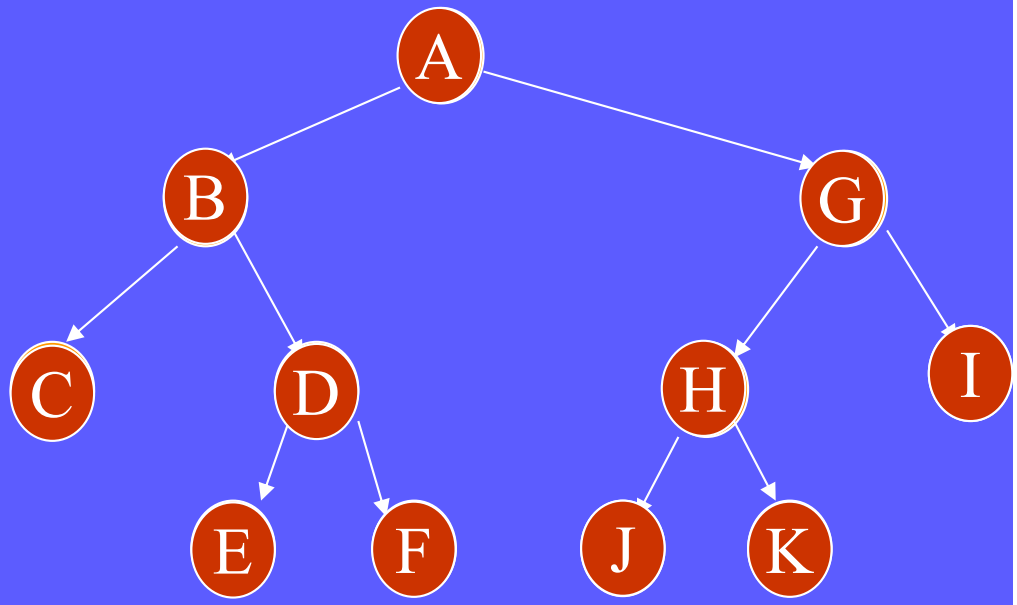




Tree Traversals-postorder

- **Post-order (Left Subtree → Right Subtree → Root)**





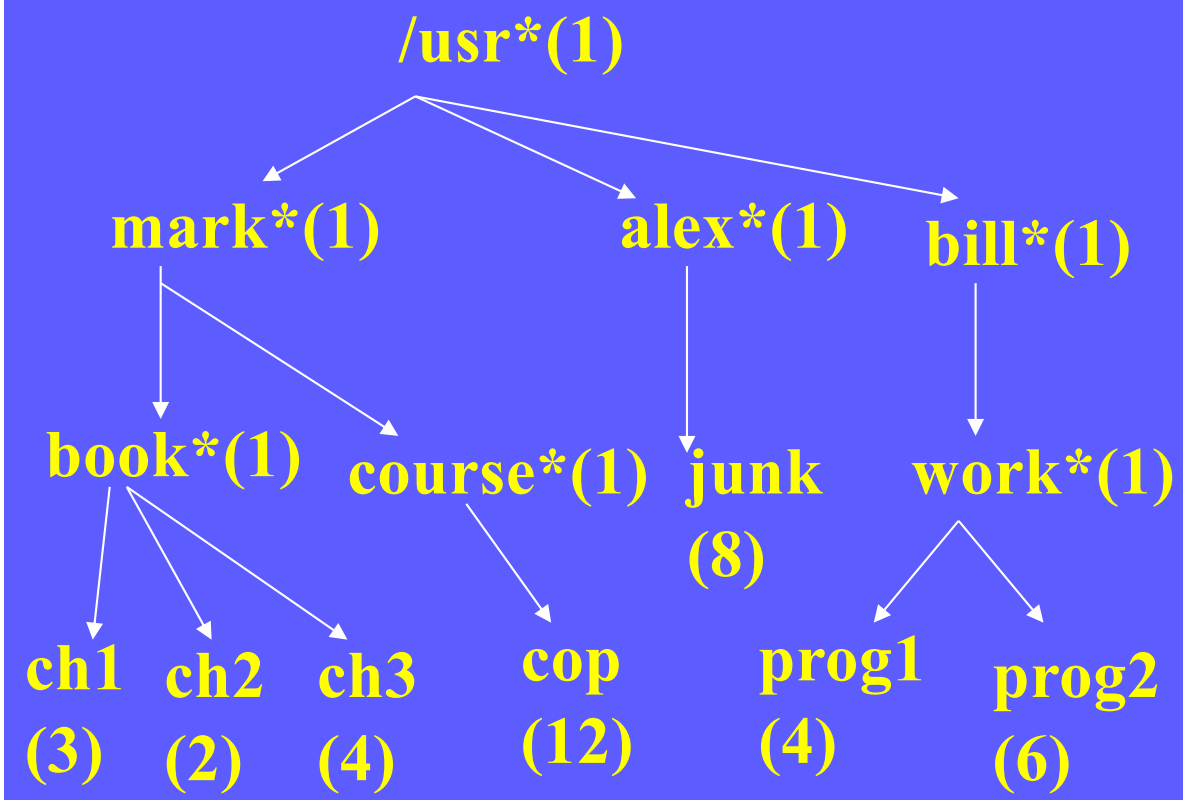
Pre-order Traversals with Application


```
/usr
mark
   book
      ch1
      ch2
      ch3
   course
      cop
alex
   junk
bill
   work
      prog1
      prog2
```

Pre-order Traversals with Application

```
static void ListDir(DirectoryOrFile D, int Depth)
{
        if (D is a legitimate entry)
        {
                PrintName(D, Depth);
                if (D is a directory)
                        for each child, C of D
                                ListDir(C, Depth+1);
        }
}
void ListDirectory(DirectoryOrFile D)
{
  ListDir(D,0);
}
```

Post-order Traversals with Application



ch1	3
ch2	2
ch3	4
book	10
cop	12
course	13
mark	24
junk	8
alex	9
prog1	4
prog2	6
work	11
bill	12
/usr	46

Post-order Traversals with Application

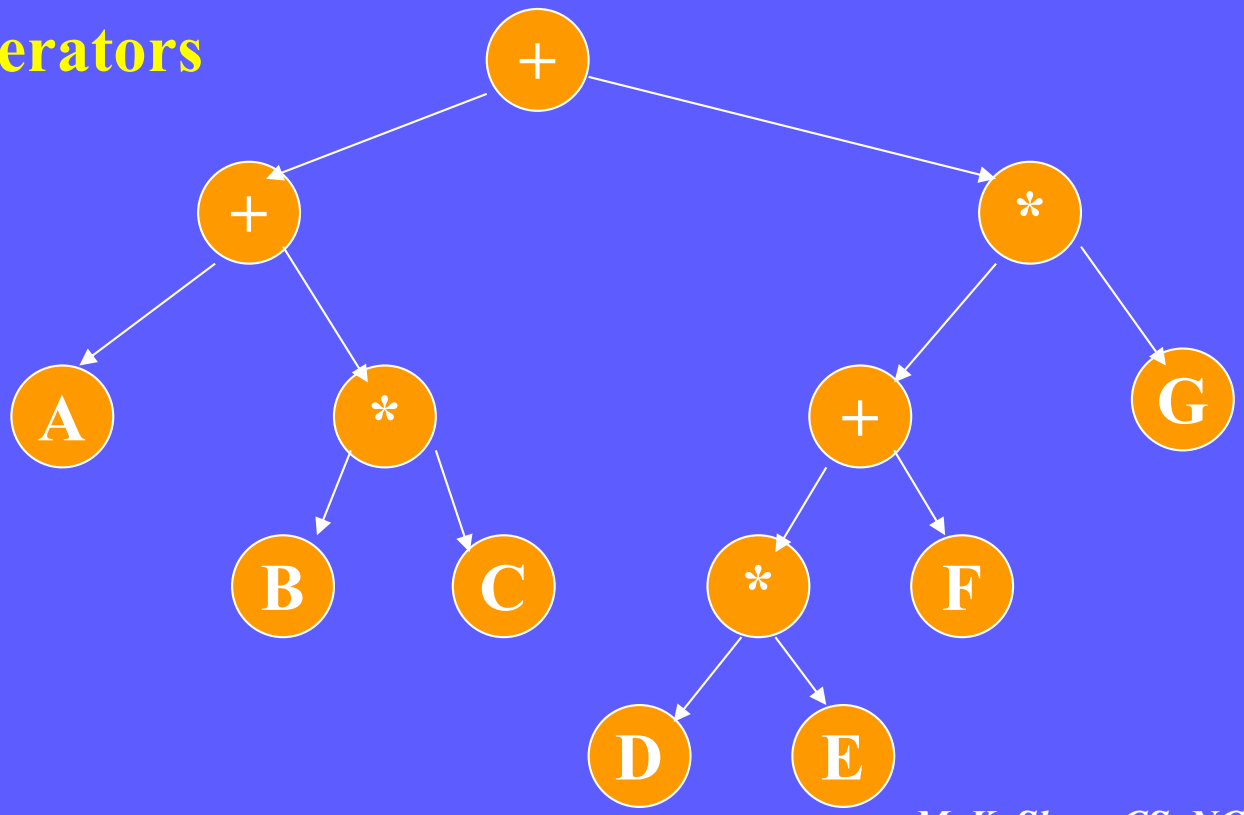
```
static void SizeDirectory(DirectoryOrFile D)
{
        int TotalSize;

        TotalSize = 0;
        if (D is a legitimate entry)
        {
                // TotalSize = FileSize(D);
                if (D is a directory)
                        for each child, C of D
                                TotalSize = TotalSize + SizeDirectory (C);
                        TotalSize=Totalsize+FileSize(D);
        }
        return TotalSize;
}
```

Expression Tree

- **Expression trees**
  - **leaves: operands**
  - **non-leaf nodes: operators**

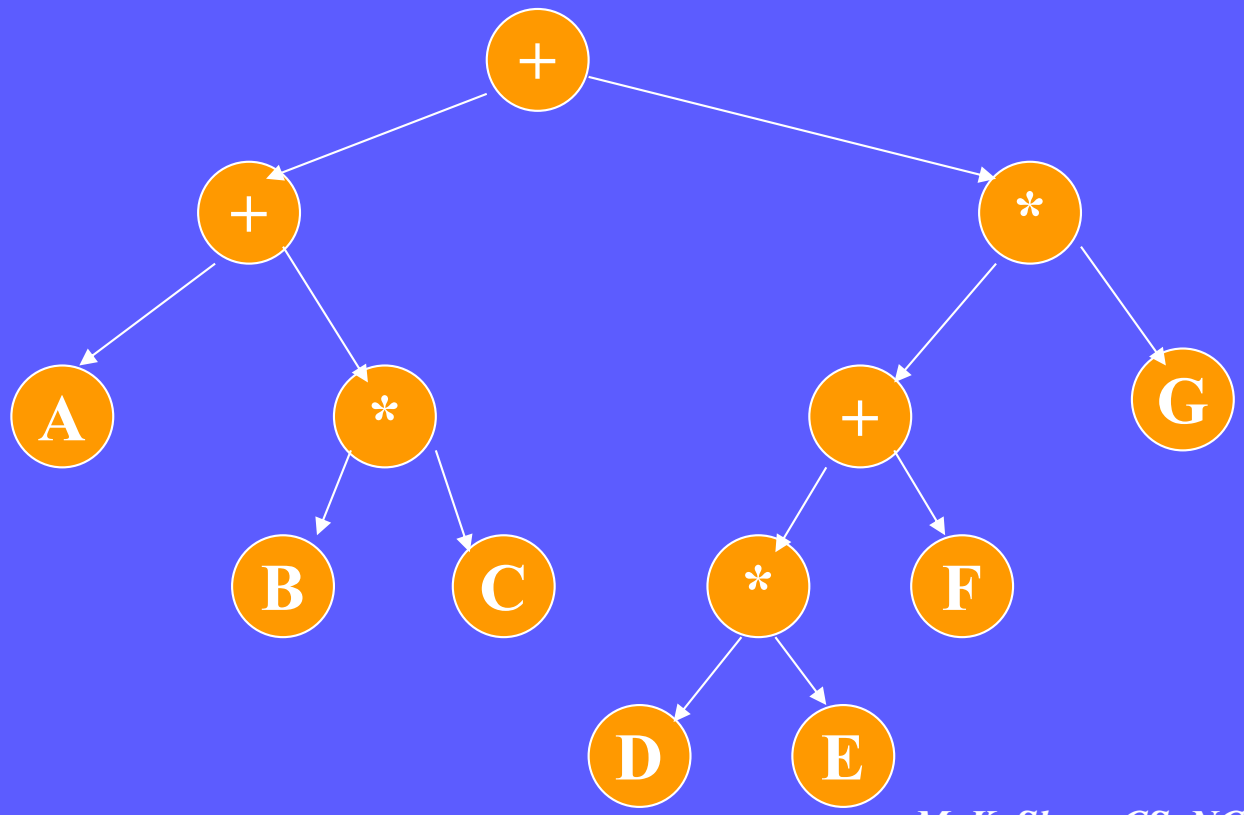
A+B*C+(D*E+F)*G

(A+(B*C))+(((D*E)+F)*G)

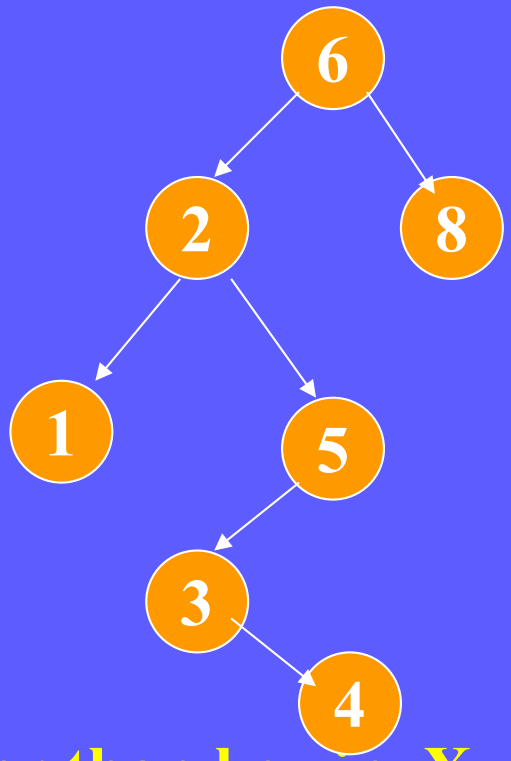


Expression Tree (cont.)

給定 A, B, C, D, E, F, G 的數值，
如何由 Expression Tree 算出此運算式的結果?



The Search Tree ADT: Binary Search Trees

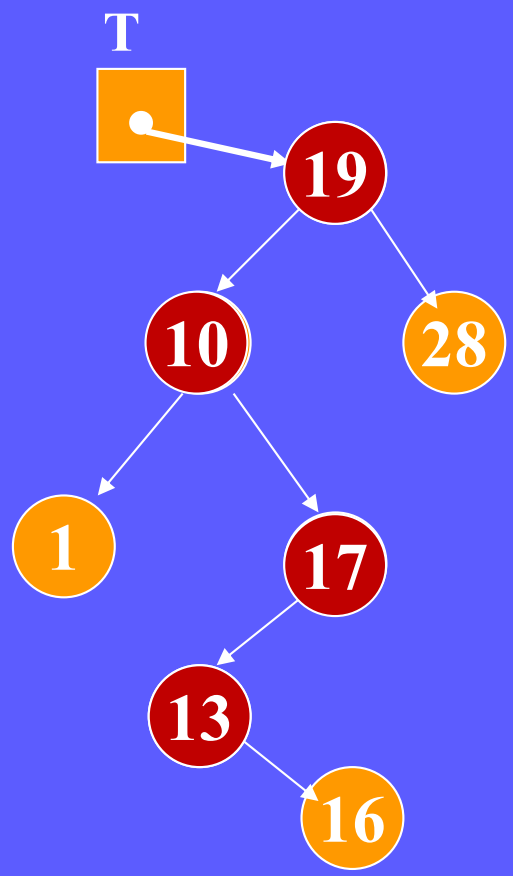


- **Binary Trees**
- **Binary Search Trees**
  - **binary tree**
  - **for every node X**
    - **values of all keys in left subtree are smaller than key in X**
    - **values of all keys in right subtree are larger than key in X**

```
#ifndef _Tree_H
  struct TreeNode;
  typedef struct TreeNode *Position;
  typedef struct TreeNode *SearchTree;
  SearchTree MakeEmpty( SearchTree T );
  Position Find( ElementType X, SearchTree T );
  Position FindMin( SearchTree T );
  Position FindMax( SearchTree T );
  SearchTree Insert( ElementType X, SearchTree T );
  SearchTree Delete( ElementType X, SearchTree T );

#endif

struct TreeNode
{
        ElementType Element;
        SearchTree  Left;
        SearchTree  Right;
};
```

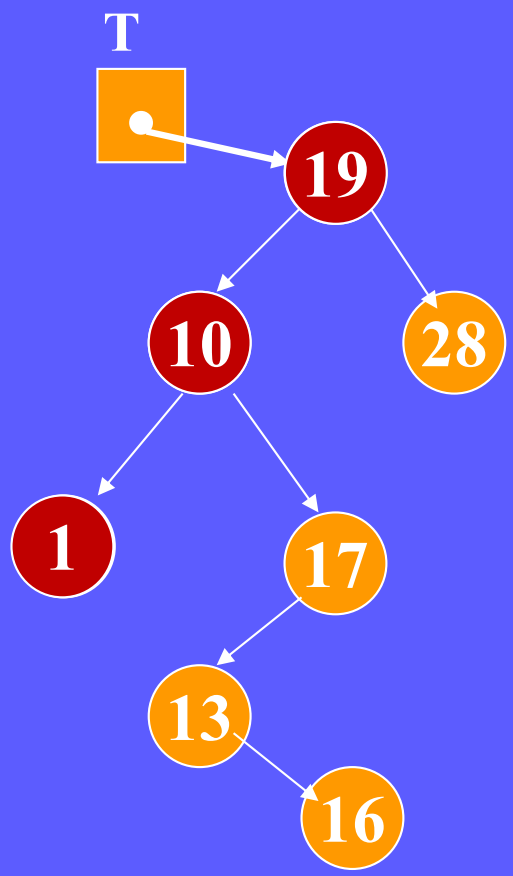
Binary Search Tree-Find

```
Position  Find( ElementType X, SearchTree T )
{
        if( T == NULL )
            return NULL;
        if( X < T->Element )
            return Find( X, T->Left );
        else
        if( X > T->Element )
            return Find( X, T->Right );
        else
            return T;
}
```



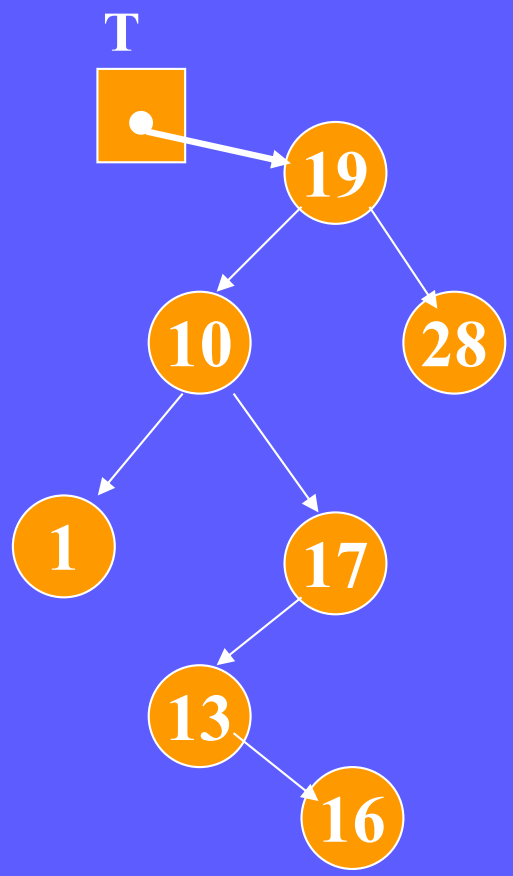
Binary Search Tree-Find Minimum

```
Position  FindMin( SearchTree T )
{
        if( T == NULL )
            return NULL;
        else
        if( T->Left == NULL )
            return T;
        else
            return FindMin( T->Left );
}
```



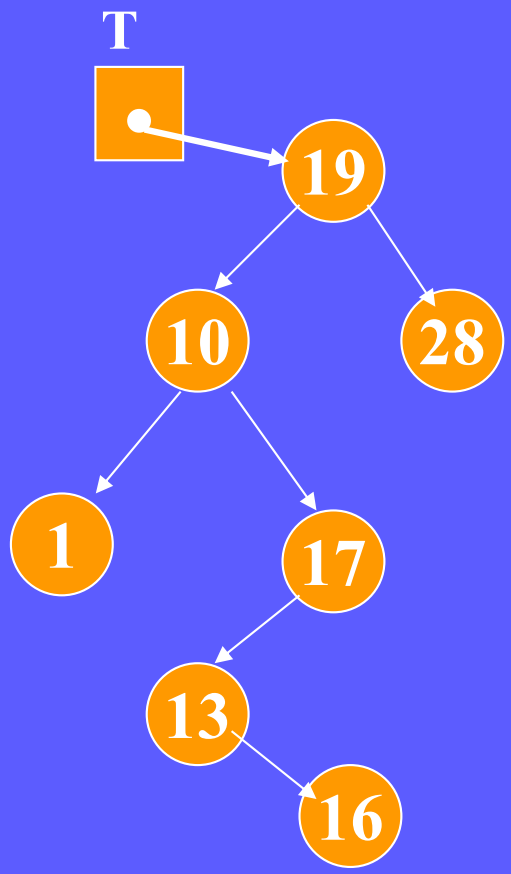
Binary Search Tree-Find Maximum

```
Position  FindMax( SearchTree T )
{
        if( T != NULL )
            while( T->Right != NULL )
                T = T->Right;

        return T;
}
```



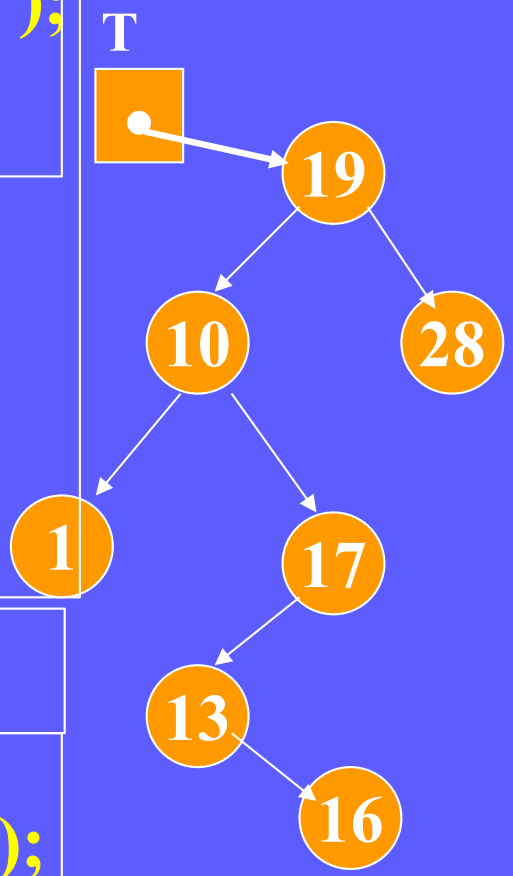
Algorithm of Binary Search Tree-Insert

```
Tree   Insert( Element X, Tree T )
{      If ( X < T.Key )
                Insert (X, T's left child);
       else if ( X > T.Key )
                Insert( X, T's right child);
       else if T is Null
                 Create Node N & fill in X
       return T;
   }
```



Binary Search Tree-Insert

```
SearchTree   Insert( ElementType X, SearchTree T )
{
    if  ( T == NULL )
    {
      T = malloc( sizeof( struct TreeNode ) );
      if   ( T == NULL )
          FatalError( "Out of space!!!" );
      else
      {
          T->Element = X;
          T->Left = T->Right = NULL;
      }
    }
    else if  ( X < T->Element )
            T->Left = Insert( X, T->Left );
    else if  ( X > T->Element )
            T->Right = Insert( X, T->Right );
    return T;
}
```

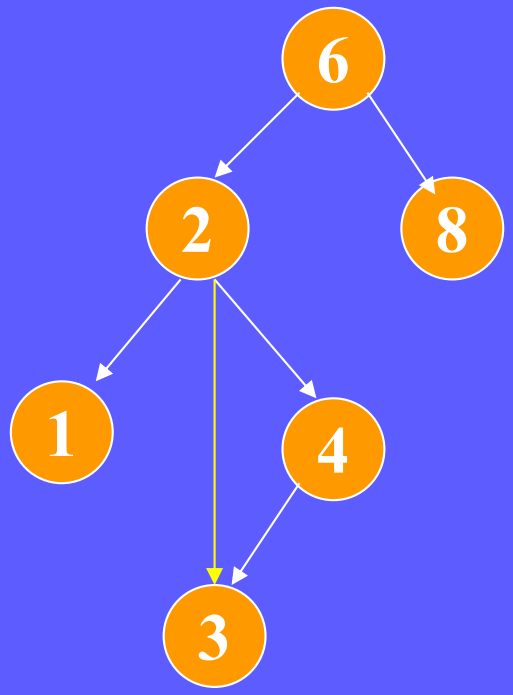


Binary Search Tree-Delete

- **Three possibilities of the deleted node**
  - **leaf: delete directly**
  - **one child:**
    - **adjusts parent a pointer to bypass the node**
    - **delete the node**
  - **two children**
    - **replace with the smallest data of right subtree**
    - **delete this smallest node**

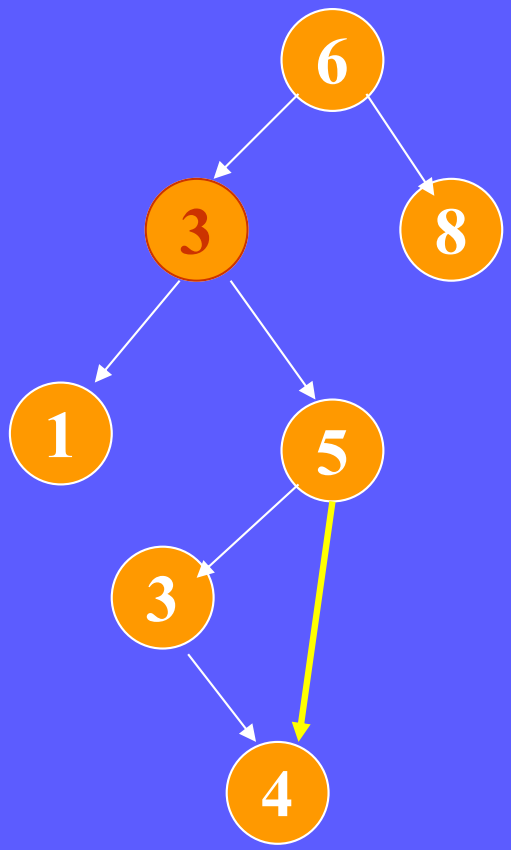
Binary Search Tree-Delete (cont.)

- **one child:**
  - **adjusts parent a pointer to bypass the node**
  - **delete the node**



Binary Search Tree-Delete (cont.)

- **two children:**
  - **replace with the smallest data of right subtree**
  - **delete this smallest node**



Binary Search Tree-Delete (cont.)

```
Tree  Delete( Element X, Tree T )
{
    if    T is Null
          output ( “not found" );
    else   if    ( X < T.Key )
                 Delete( X, T->Left );
    else   if  ( X > T->Element )
                 Delete( X, T->Right );
    else  if   T has 2 children
          {     Find the minimun m of T’s right subtree ;
                Replace T.key with m;
                Delete m;   }
    else  {     Adjusts T’s parent a pointer to bypass the node
                Delete T;  }
    return T;
}
```

Binary Search Tree-Delete (cont.)

```
SearchTree  Delete( ElementType X, SearchTree T )
{   Position TmpCell;
    if   ( T == NULL )
         Error( "Element not found" );
    else   if    ( X < T->Element )
                 T->Left = Delete( X, T->Left );
    else   if  ( X > T->Element )
                 T->Right = Delete( X, T->Right );
    else  if   ( T->Left && T->Right )
           {     TmpCell = FindMin( T->Right );
                 T->Element = TmpCell->Element;
                 T->Right = Delete( T->Element, T->Right );  }
    else {     TmpCell = T;
                 if    ( T->Left == NULL )
                       T = T->Right;
                 else if ( T->Right == NULL )
                       T = T->Left;
                free( TmpCell );                }
    return T;
}
```

Discussions on Binary Search Trees

- **Different input orders produce different binary search tree**

  <e.g.> comes in presorted ⇒ degenerates to linked list

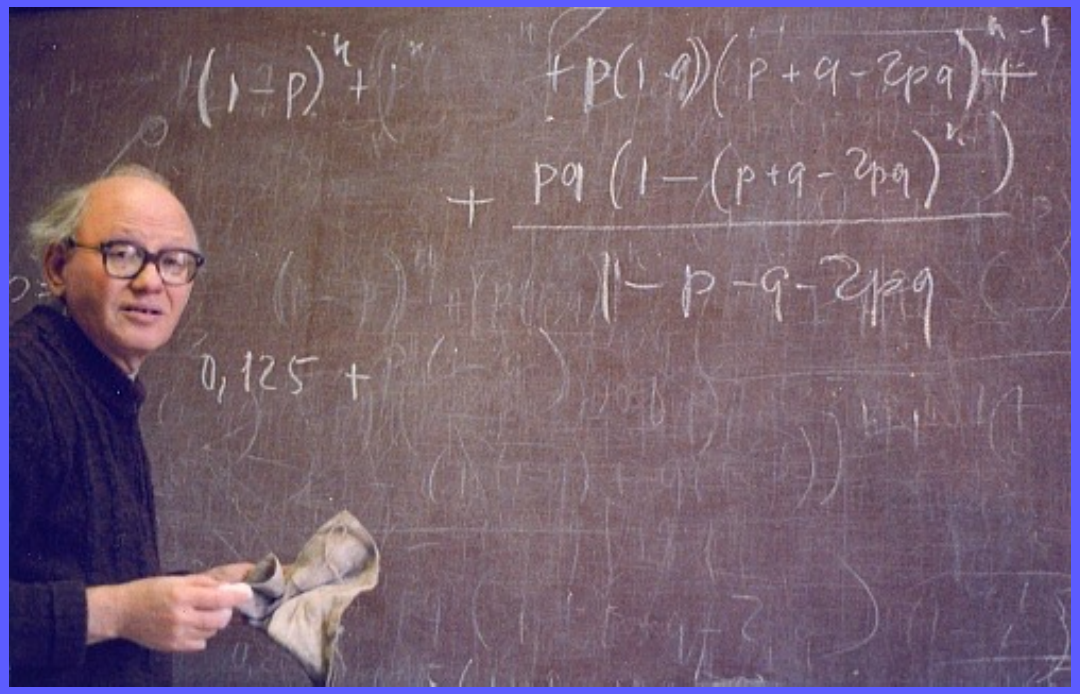
- **Given a binary search tree with depth *D***
  - **Insert: best O(1), worst O(*D*)**
  - **Retrieve: best O(1), worst O(*D*)**

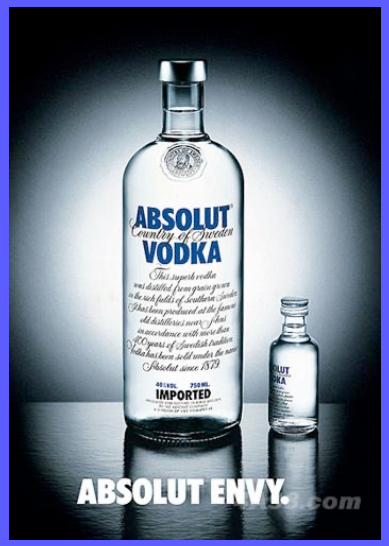
  *** Given N nodes, *D:* best O(logN), worst O(N)**

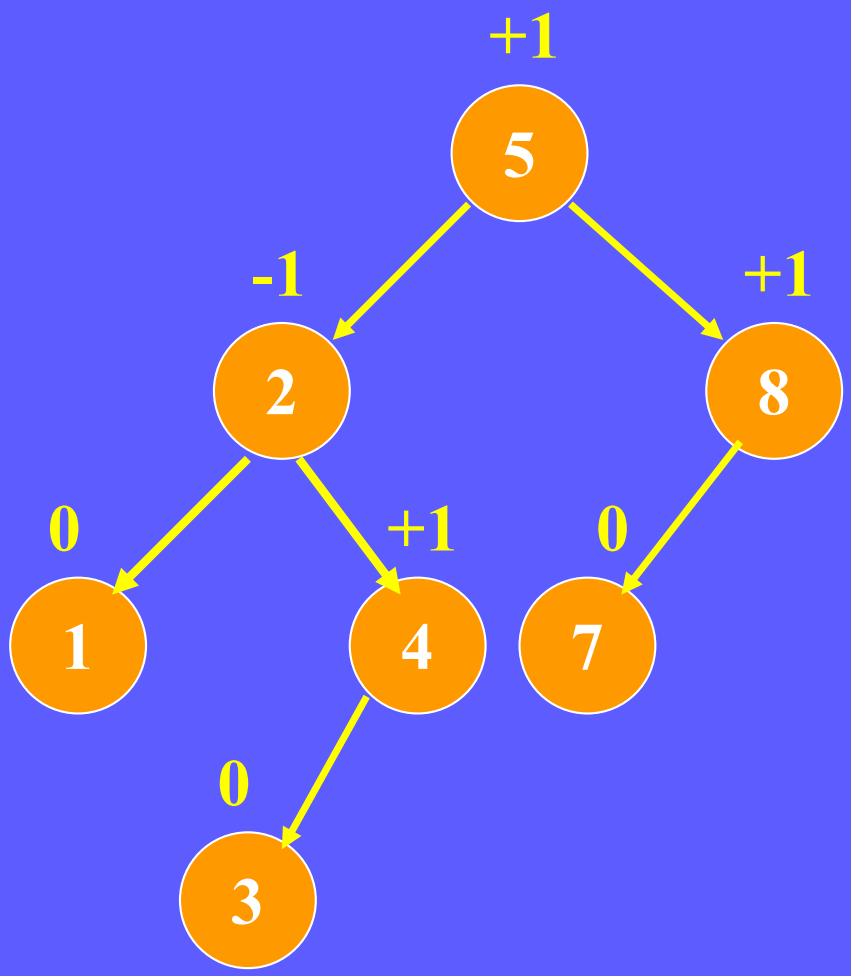
  ⇒ balanced binary search tree with *D* = O(logN)

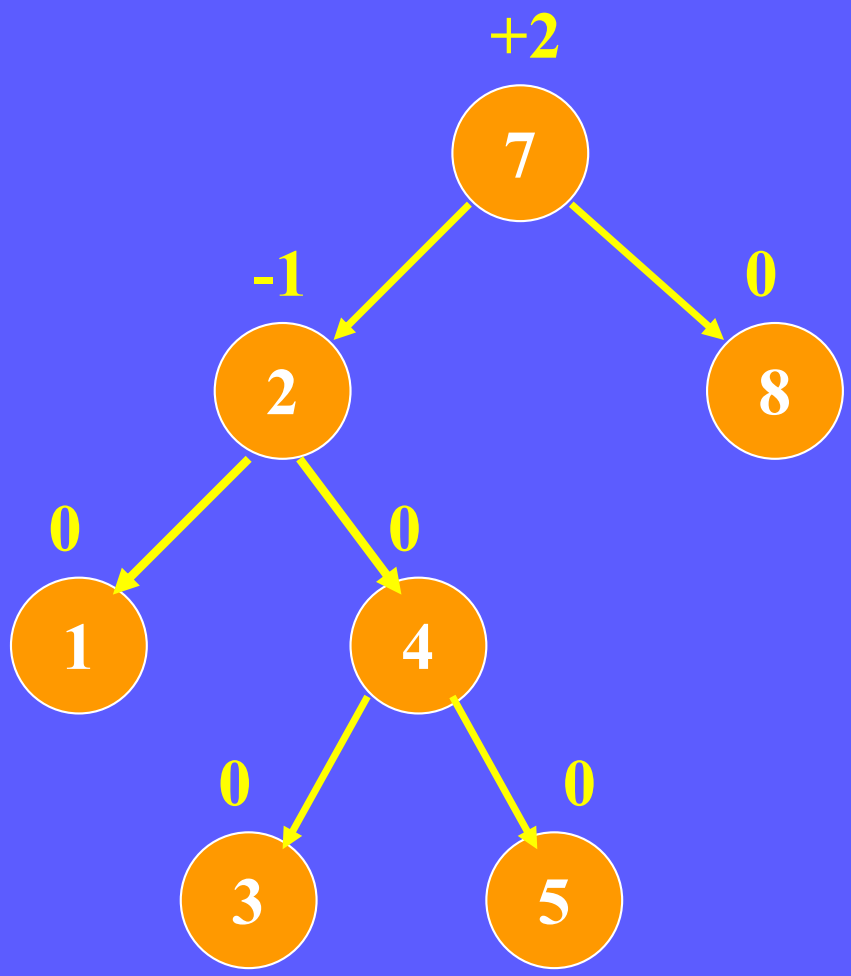
AVL Tree

- **AVL Tree**
  - **developed by Adelson-Velskii & Landis**
  - **binary search tree with a balance condition**
    : the height of the left & right subtrees can differ by at most 1
  - **easy to maintain balance condition**
  - **depth of AVL Tree is O(logN)**

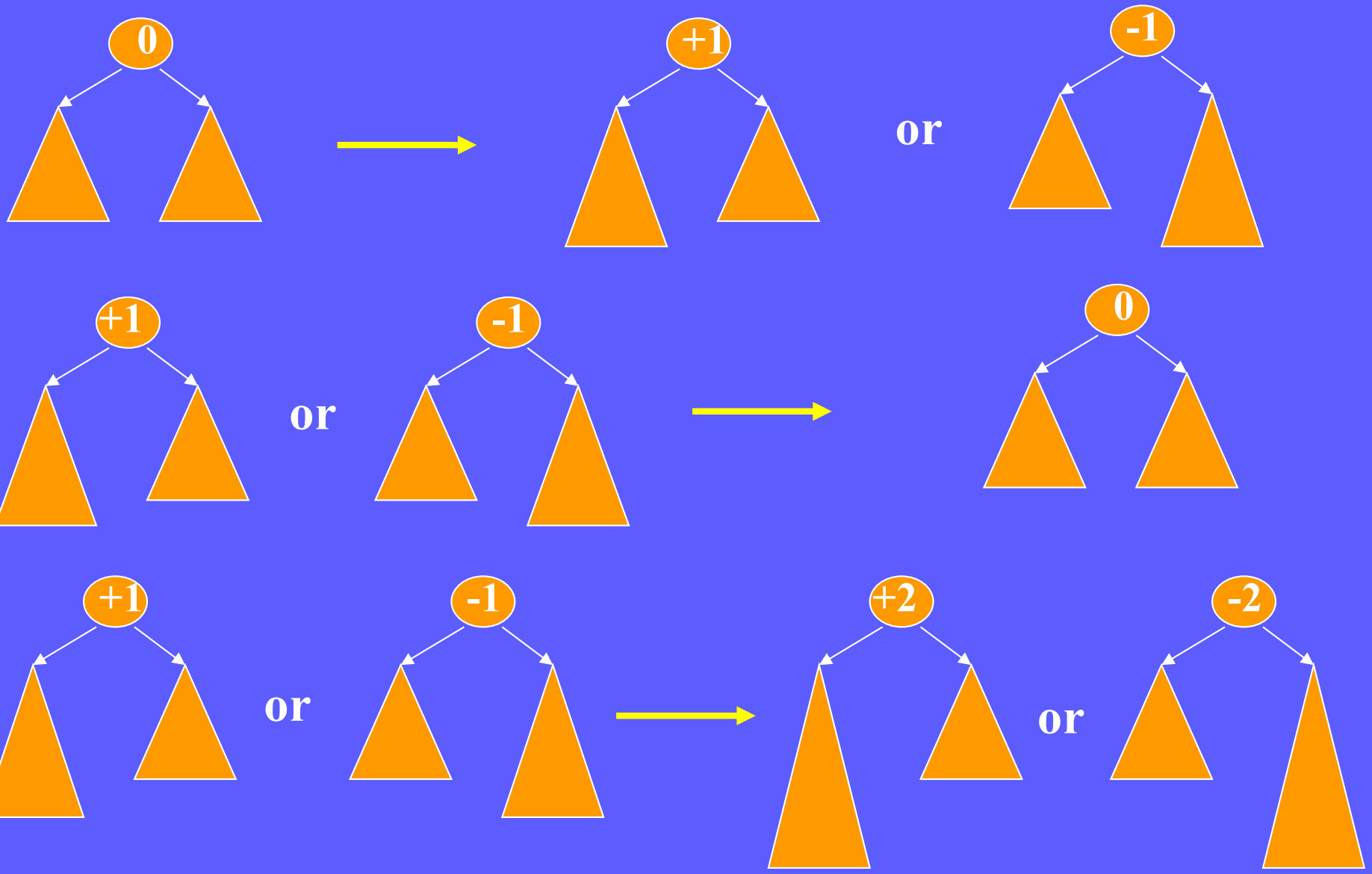

0,125 +
1-p-q-2pq


ABSOLUT
Country of Sweden
VODKA
IMPORTED
ABSOLUT ENVY.


+1
5
-1
+1
2
8
0
+1
0
1
4
7
0
3


+2
7
-1
0
2
8
0
0
1
4
0
0
3
5

3 Cases After Insertion of AVL Tree

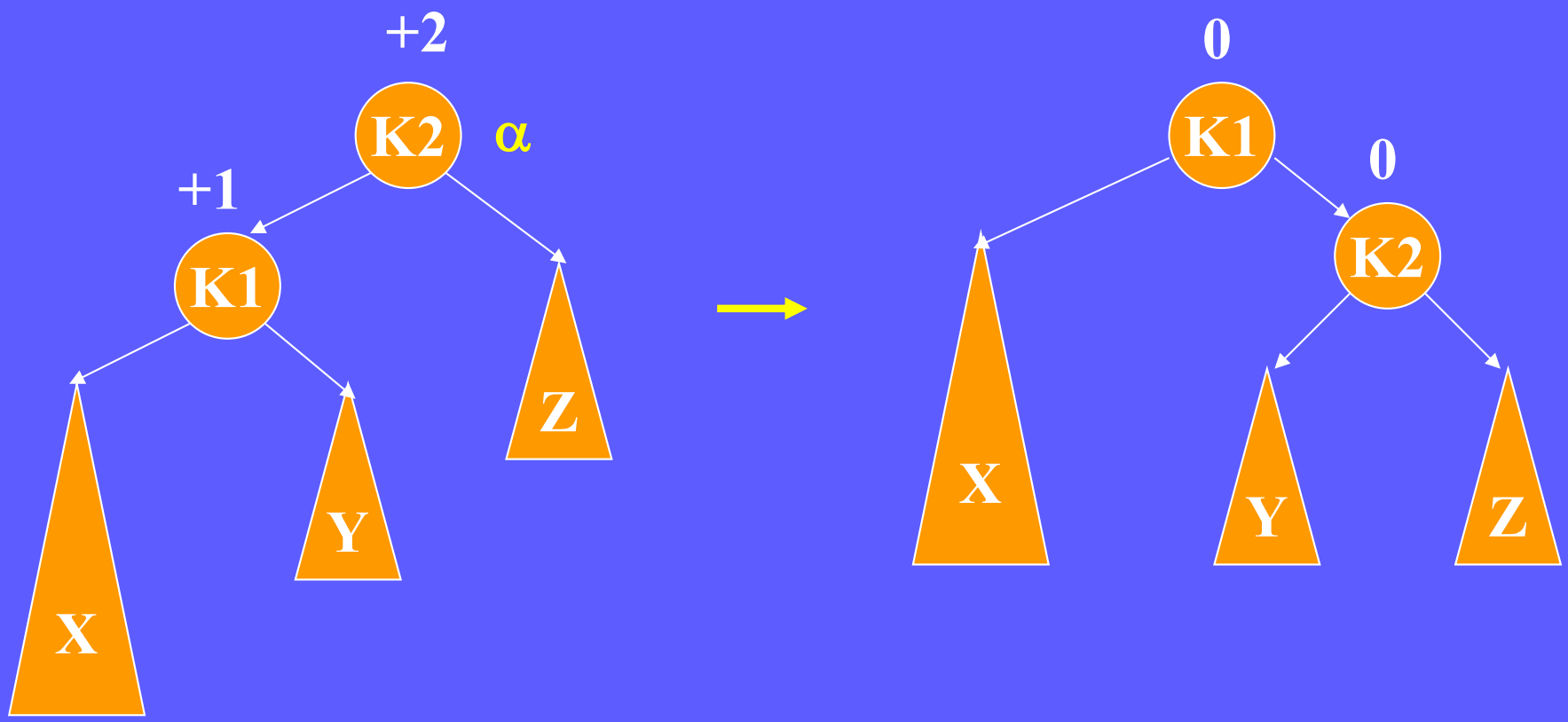


Rebalance (Rotation)

- **Four cases that need to rebalance**
  - **LL: insertion into the left subtree of the left child of α**
  - **RR: insertion into the right subtree of the right child of α**
  - **LR: insertion into the right subtree of the left child of α**
  - **RL: insertion into the left subtree of the right child of α**

*** α: the nearest imbalance ancestor of the insertion node**

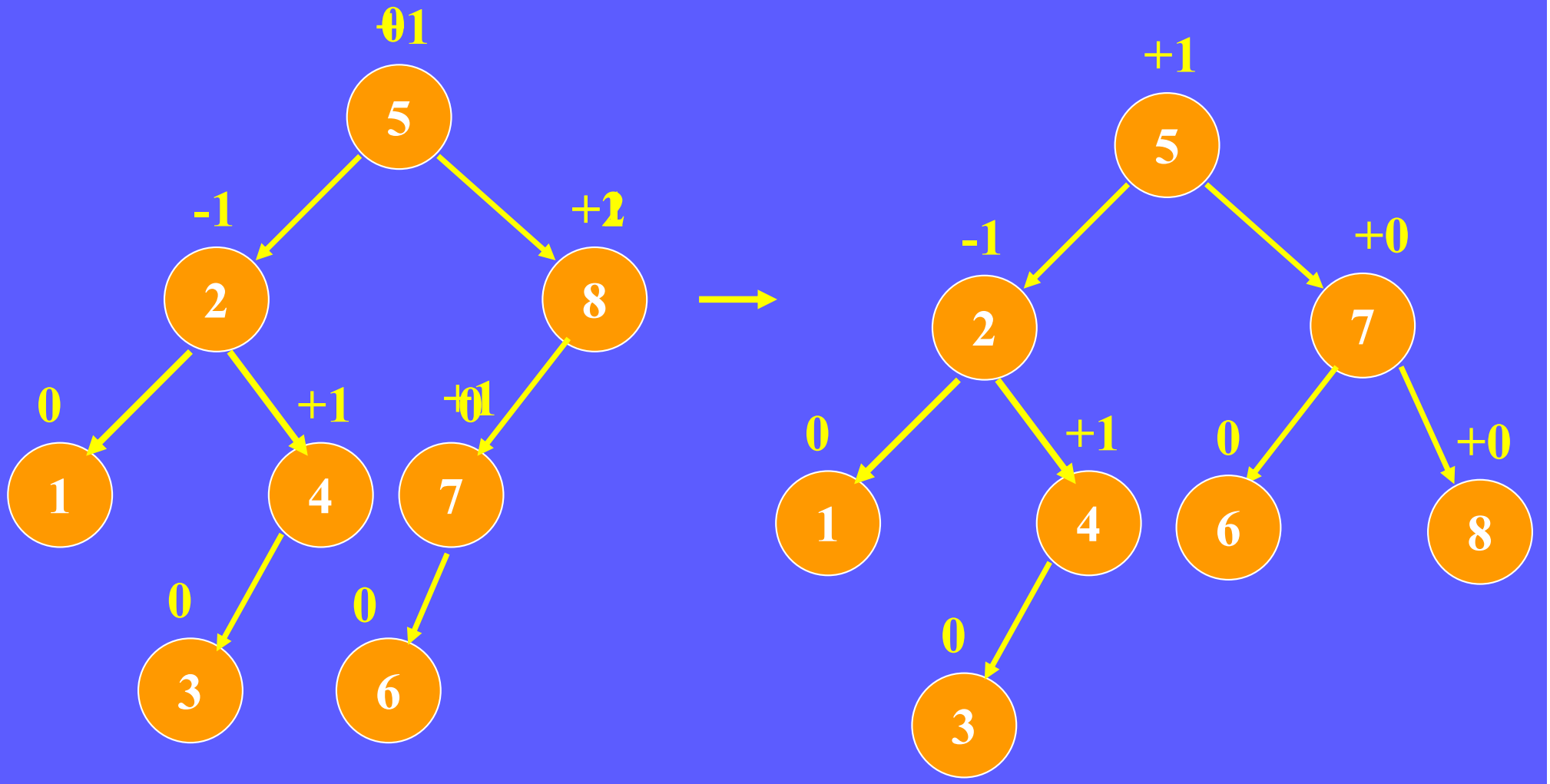
LL Rotation



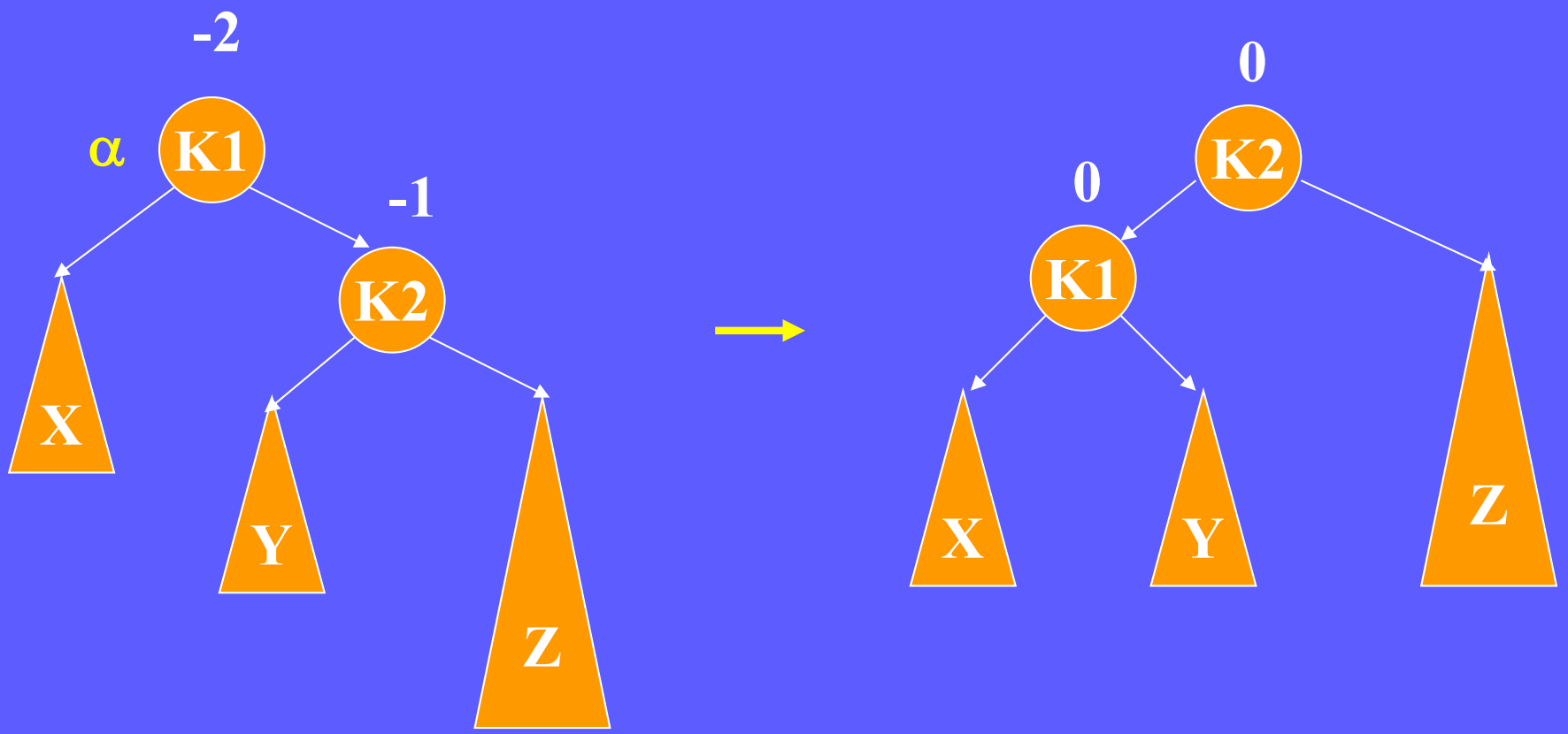
X<K1<Y<K2<Z

X<K1<Y<K2<Z

LL Rotation (cont.)



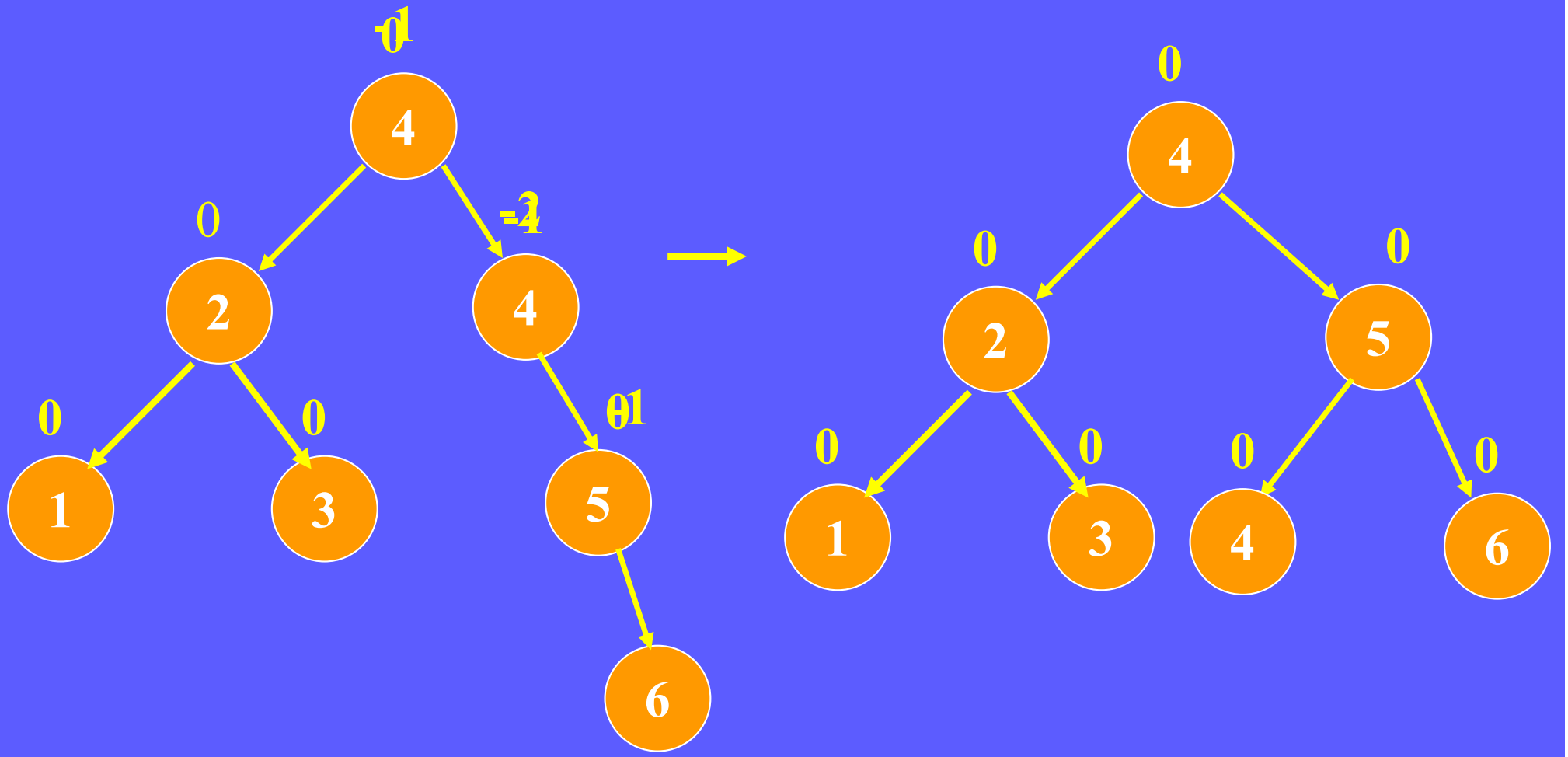
RR Rotation



X<K1<Y<K2<Z

X<K1<Y<K2<Z

RR Rotation (cont.)

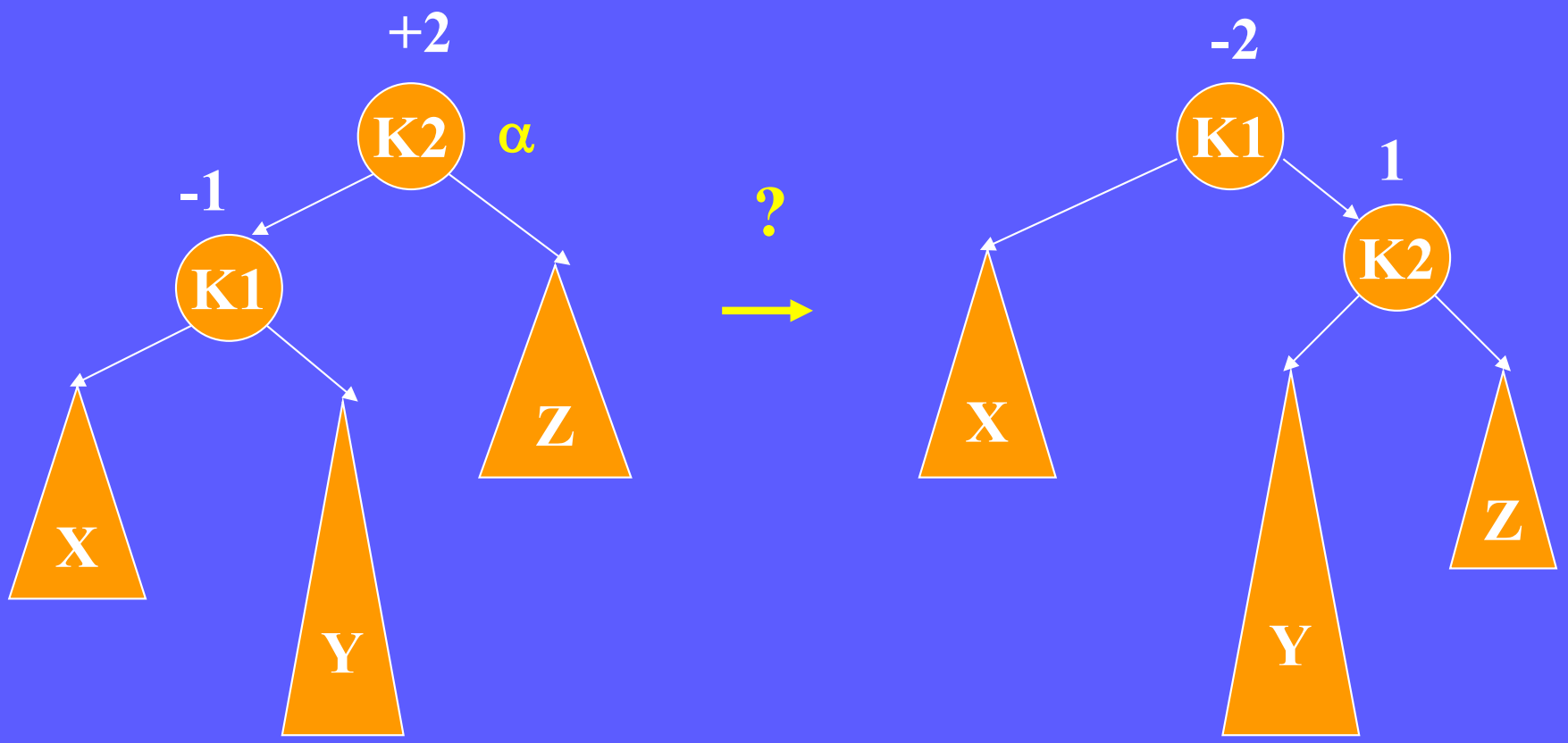


X<K1<Y<K2<Z

X<K1<Y<K2<Z

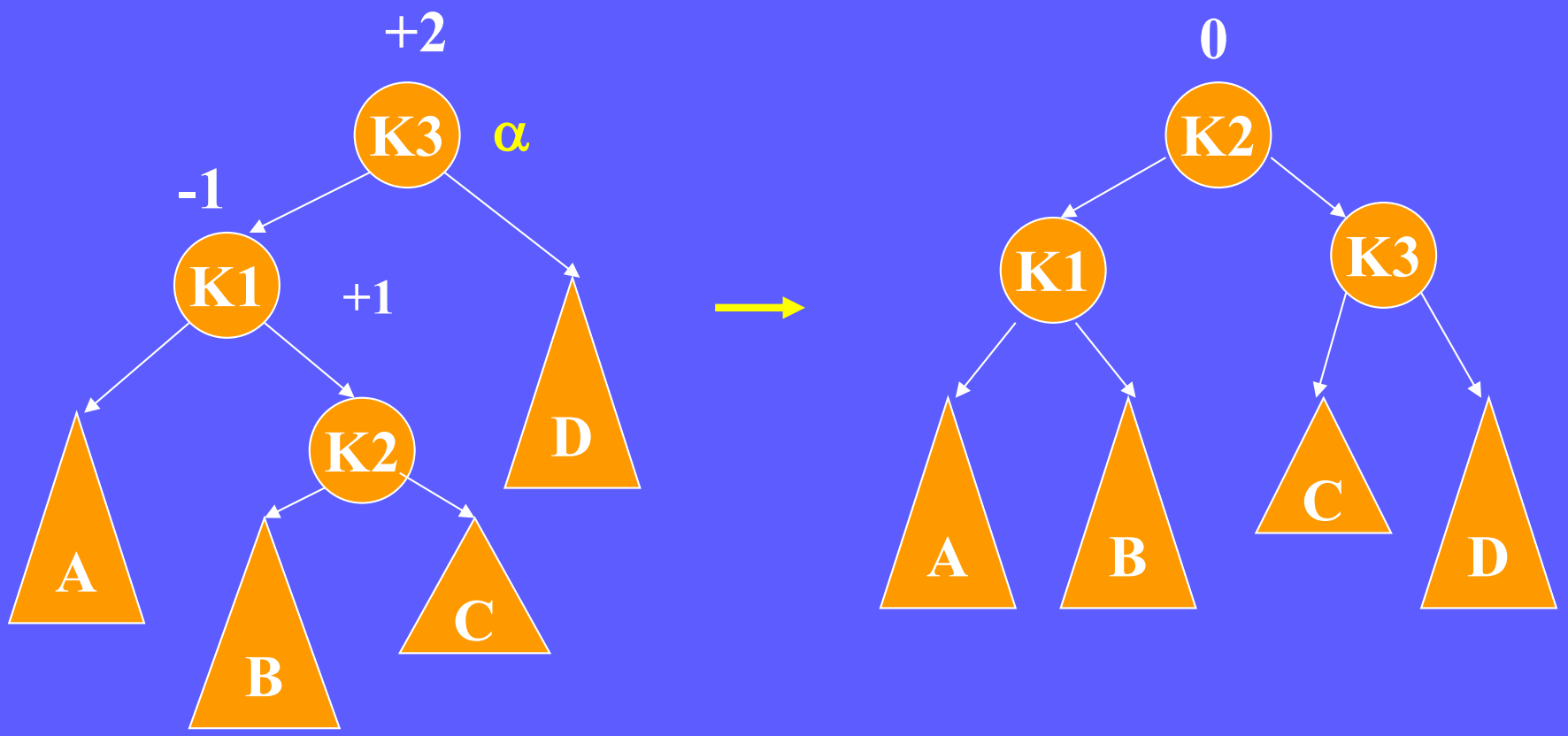
LR, RL可以如法泡製嗎？

LR Rotation



? Single rotation doesn't work.

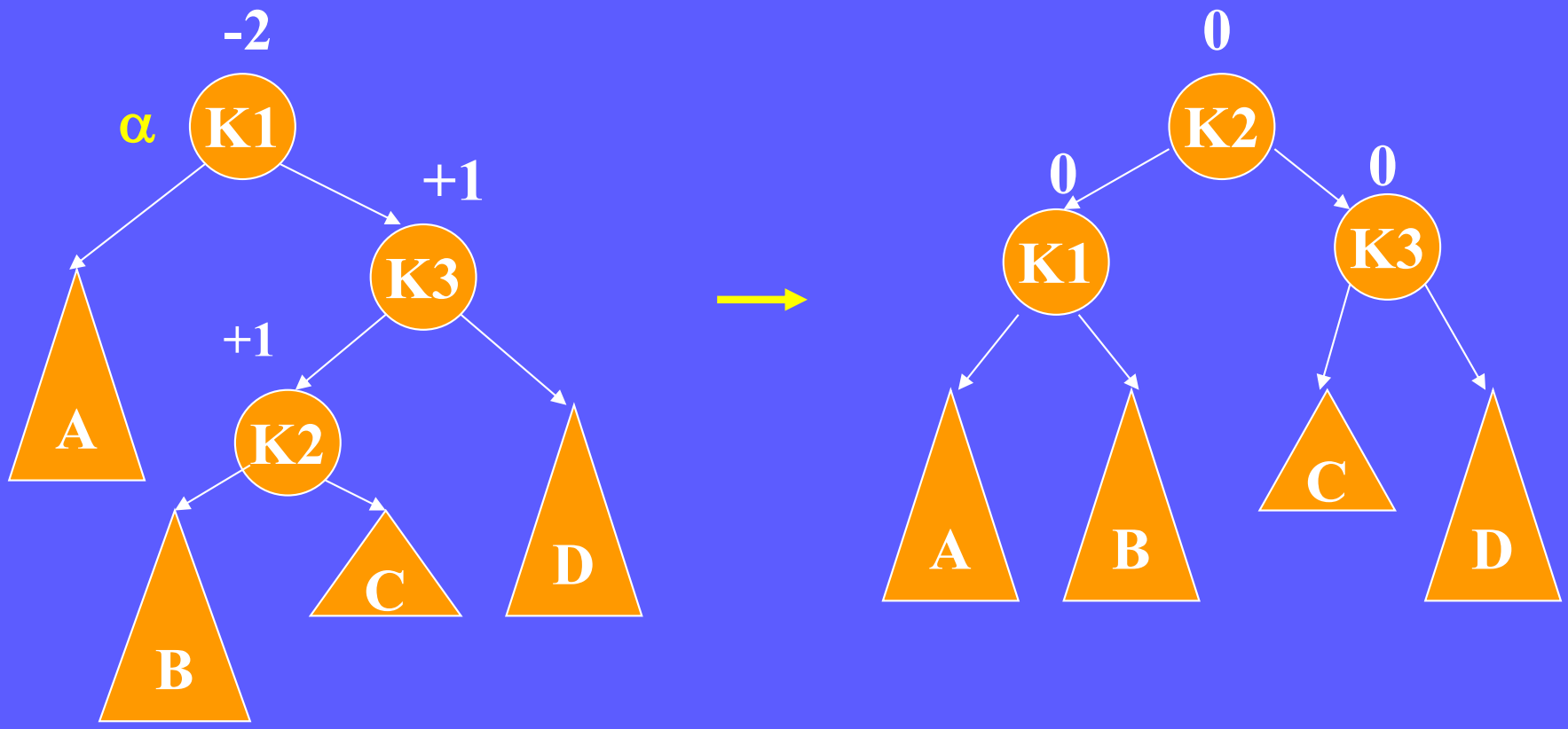
LR Rotation (Cont.)



A<K1<B<K2<C<K3<D

A<K1<B<K2<C<K3<D

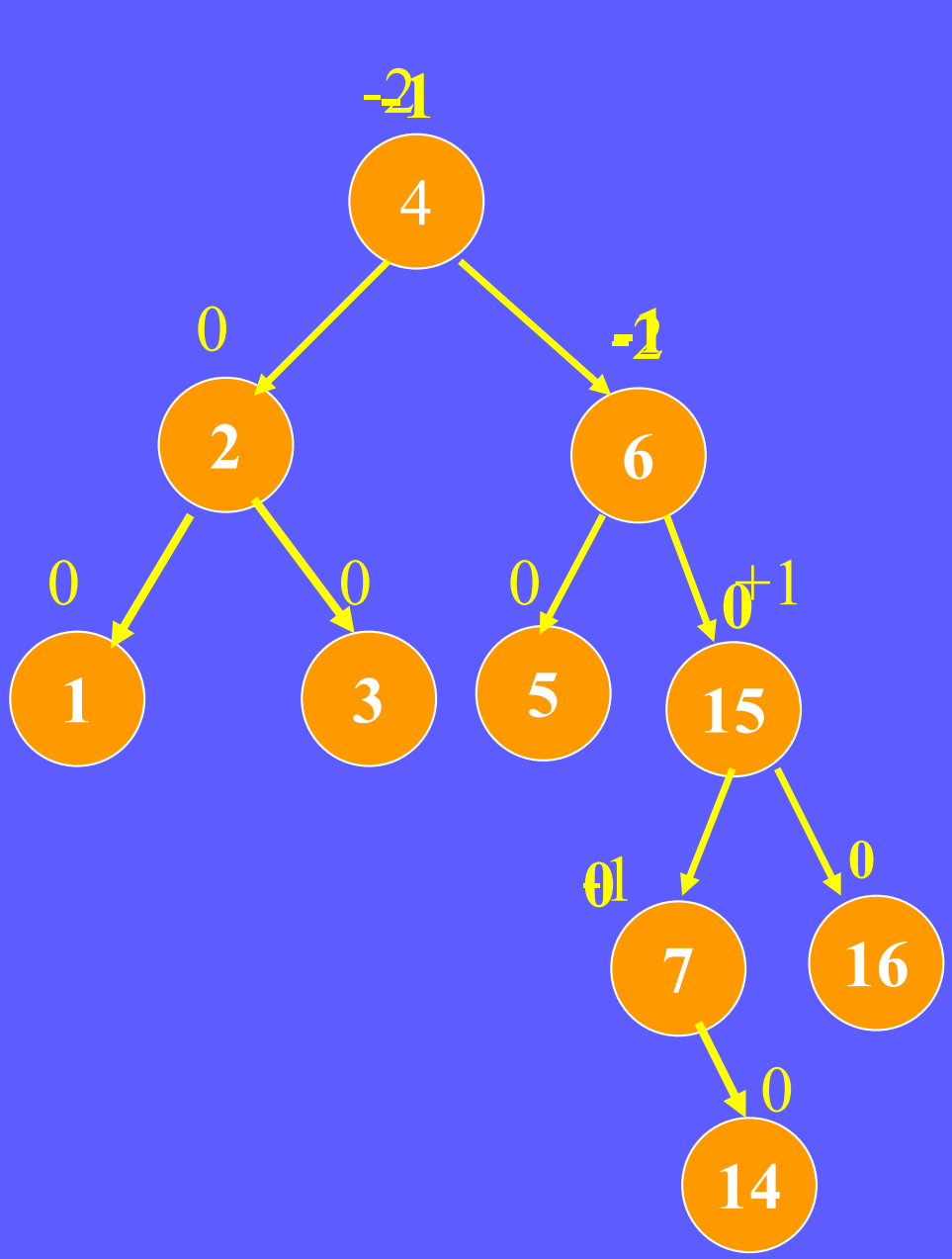
RL Rotation

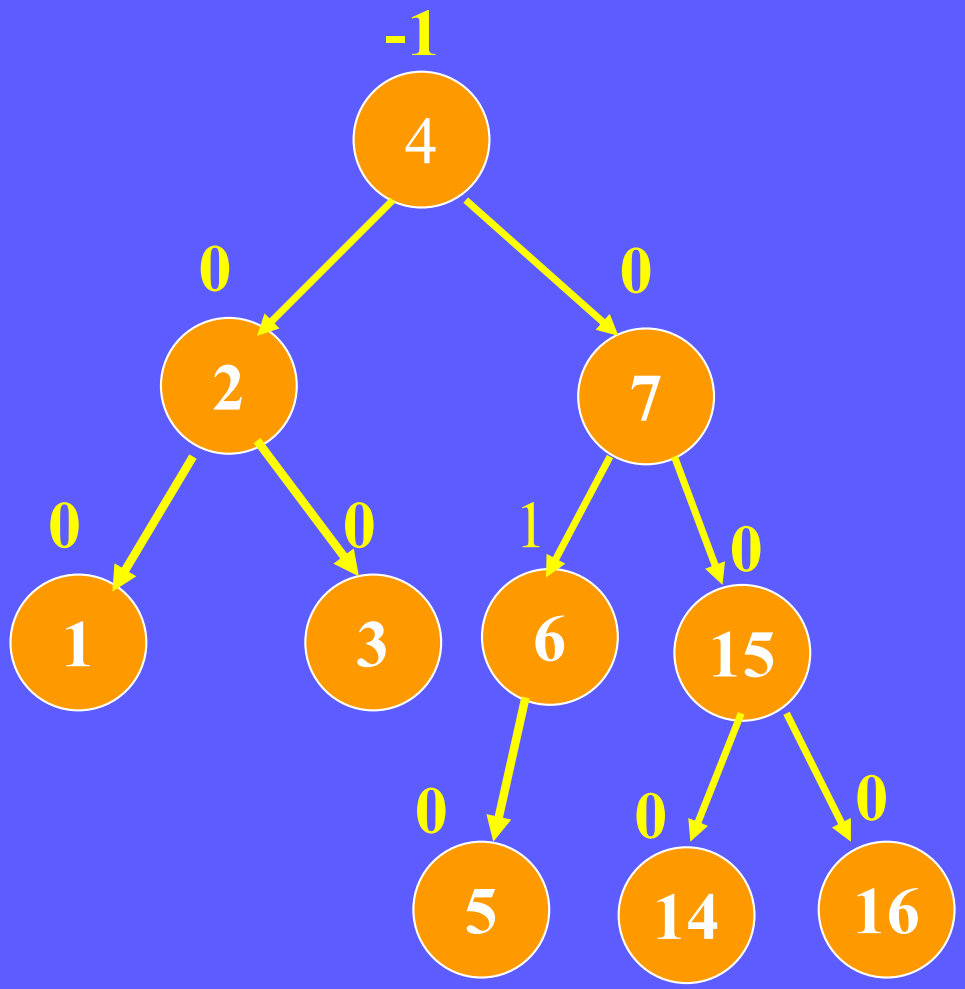


A<K1<B<K2<C<K3<D

A<K1<B<K2<C<K3<D

RL Rotation (cont.)



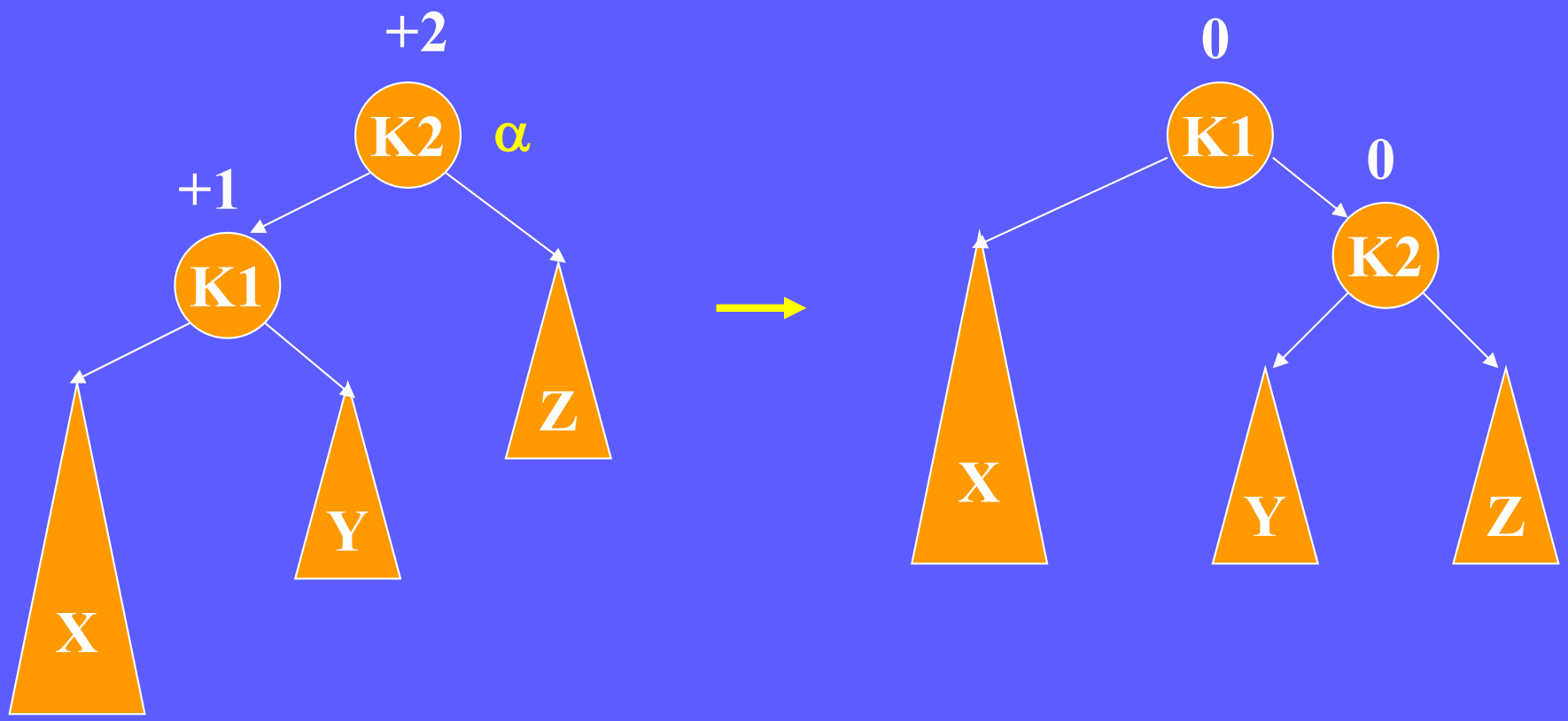


Rotation作法看起來很複雜，

程式是不是也很複雜嗎？

時間複雜度呢？

```
static Position    SingleRotateWithLeft( Position K2 )
{  Position K1;
   K1 = K2->Left;
   K2->Left = K1->Right;
   K1->Right = K2;
   K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
   K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;
   return K1;
}
```

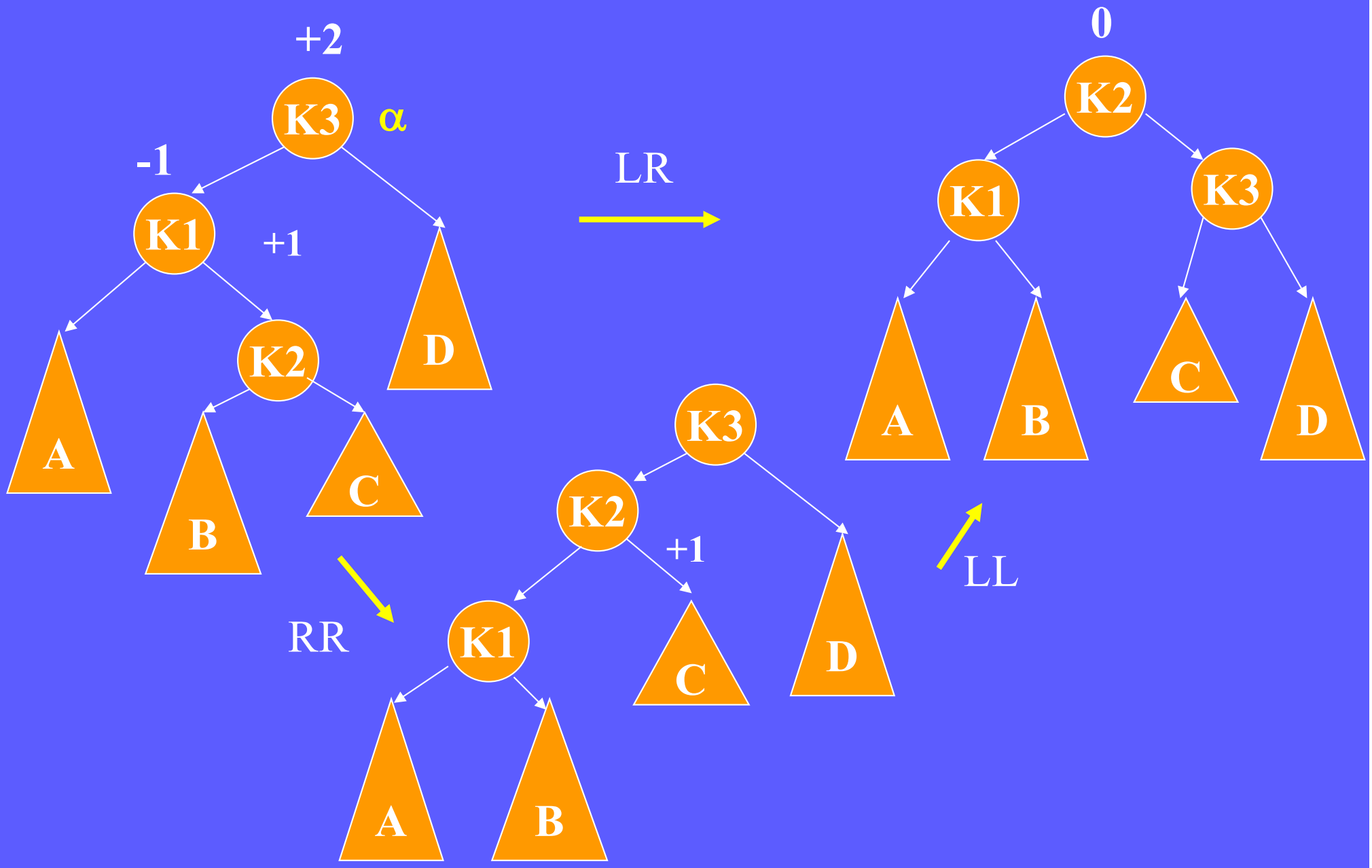


```
static Position    SingleRotateWithLeft( Position K2 )
{  Position K1;
   K1 = K2->Left;
   K2->Left = K1->Right;
   K1->Right = K2;
   K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
   K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;
   return K1;
}

 static Position   SingleRotateWithRight( Position K1 )
{   Position K2;
    K2 = K1->Right;
    K1->Right = K2->Left;
    K2->Left = K1;
    K1->Height = Max( Height( K1->Left ), Height( K1->Right ) ) + 1;
    K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;
    return K2;
}
```

LR, RL的程式是不是更複雜？

為何稱之為Double Rotation？

LR Rotation ：Double Rotation(RR+LL)



```
static Position   DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K3 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );
    /* Rotate between K3 and K2 */
   return  SingleRotateWithLeft( K3 );
}

static Position    DoubleRotateWithRight( Position K1 )
{
     /* Rotate between K3 and K2 */
    K1->Right = SingleRotateWithLeft( K1->Right );
    /* Rotate between K1 and K2 */
    return SingleRotateWithRight( K1 );
}
```

Algorithm of AVL Tree-Insert

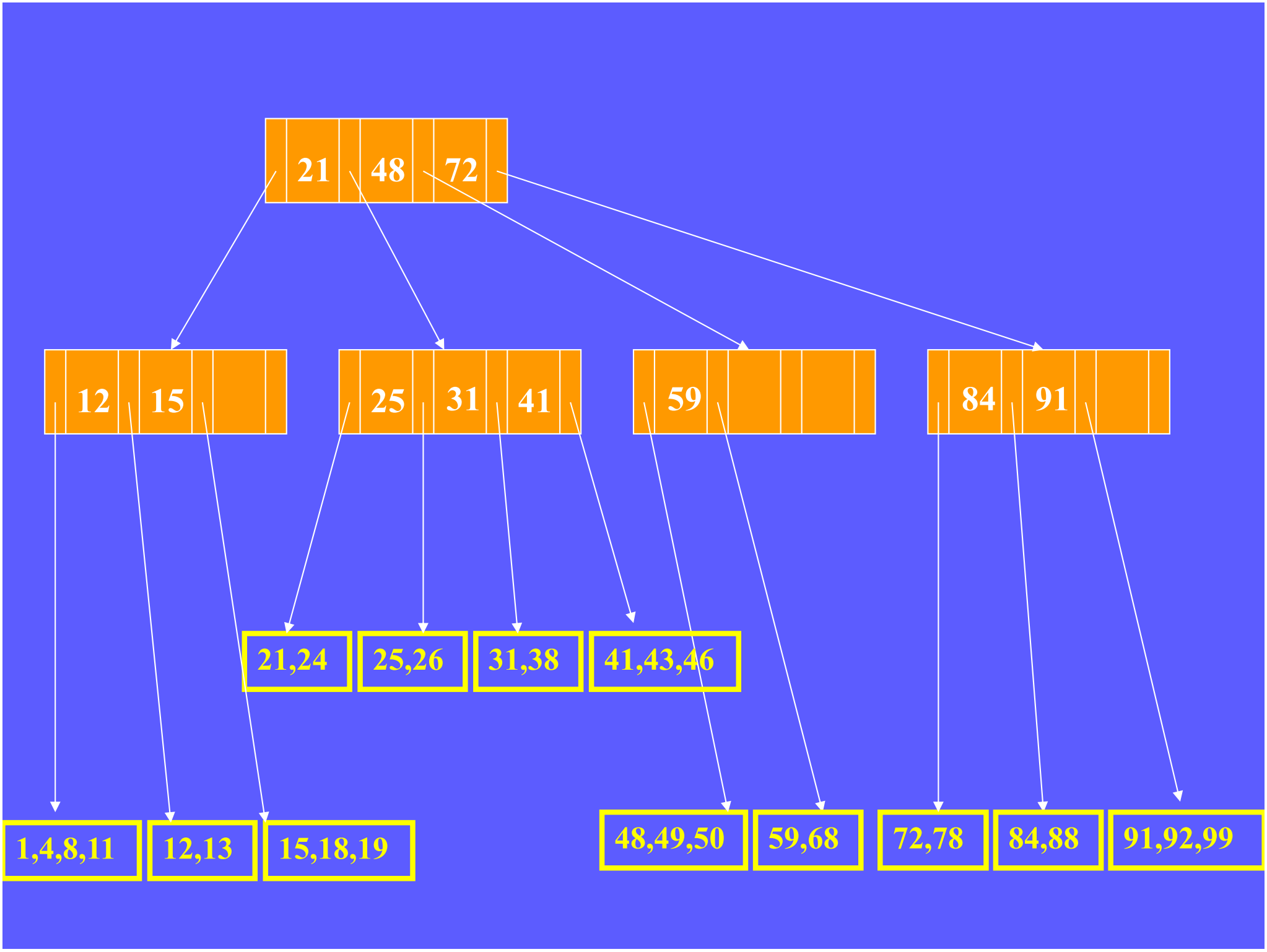
```
Tree   Insert( Element X, Tree T )
{      if T is Null
       {
          Create Node N & fill in X
          If there is imbalance along path from N to root
          {   α = the nearest imbalance ancestor of N
              case  N  of
              {  left subtree of the left child of α: LL(α);
                 right subtree of the right child of α: RR(α);
                 right subtree of the left child of α: LR(α);
                 left subtree of the right child of α: RL(α);
              }
          }
       }
       else if ( X < T.Key )
                 Insert (X, T's left child);
       else if( X > T.Key )
                 Insert( X, T's right child);
       return T;
}
```

AVL Tree Time Complexity

- **Search: O(logN)**
- **Insertion: O(logN)**
  - **Rotation: O(1)**
- **Deletion: O(logN)**

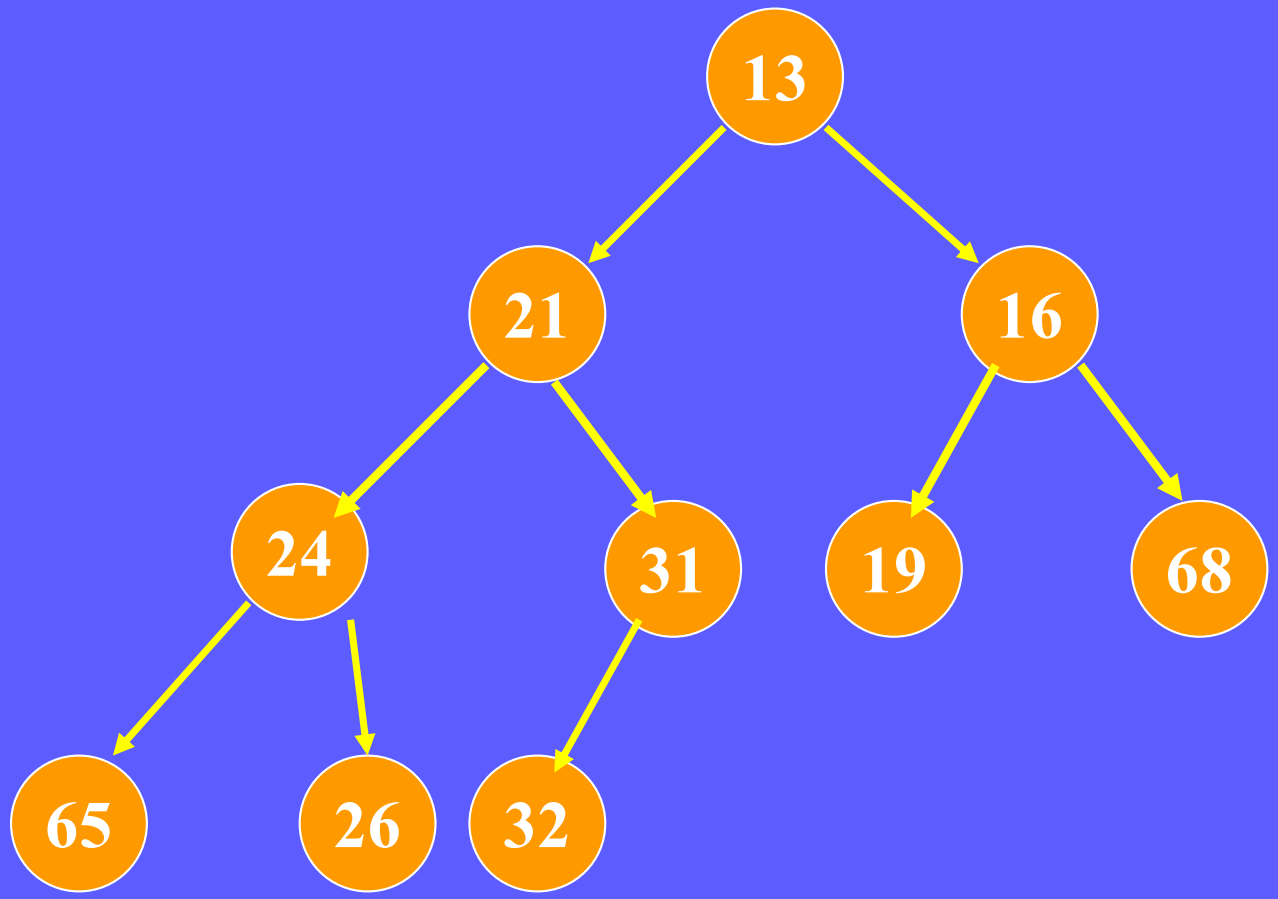
B-Trees

- **B-Trees of order M**
  - **M-way search tree**
  - **Root is either a leaf or has between 2 & M children**
  - **All nonleaf nodes have between $\lceil M/2 \rceil$ & M children**
  - **All leaves are at the same depth**
  - **In each interior node, there are pointers $P_0, P_1, \ldots, P_{M-1}$ to children and values $k_1, k_2, \ldots, k_{M-1}$, representing the smallest key found in the subtrees $P_2, P_3, \ldots, P_M$ respectively.**

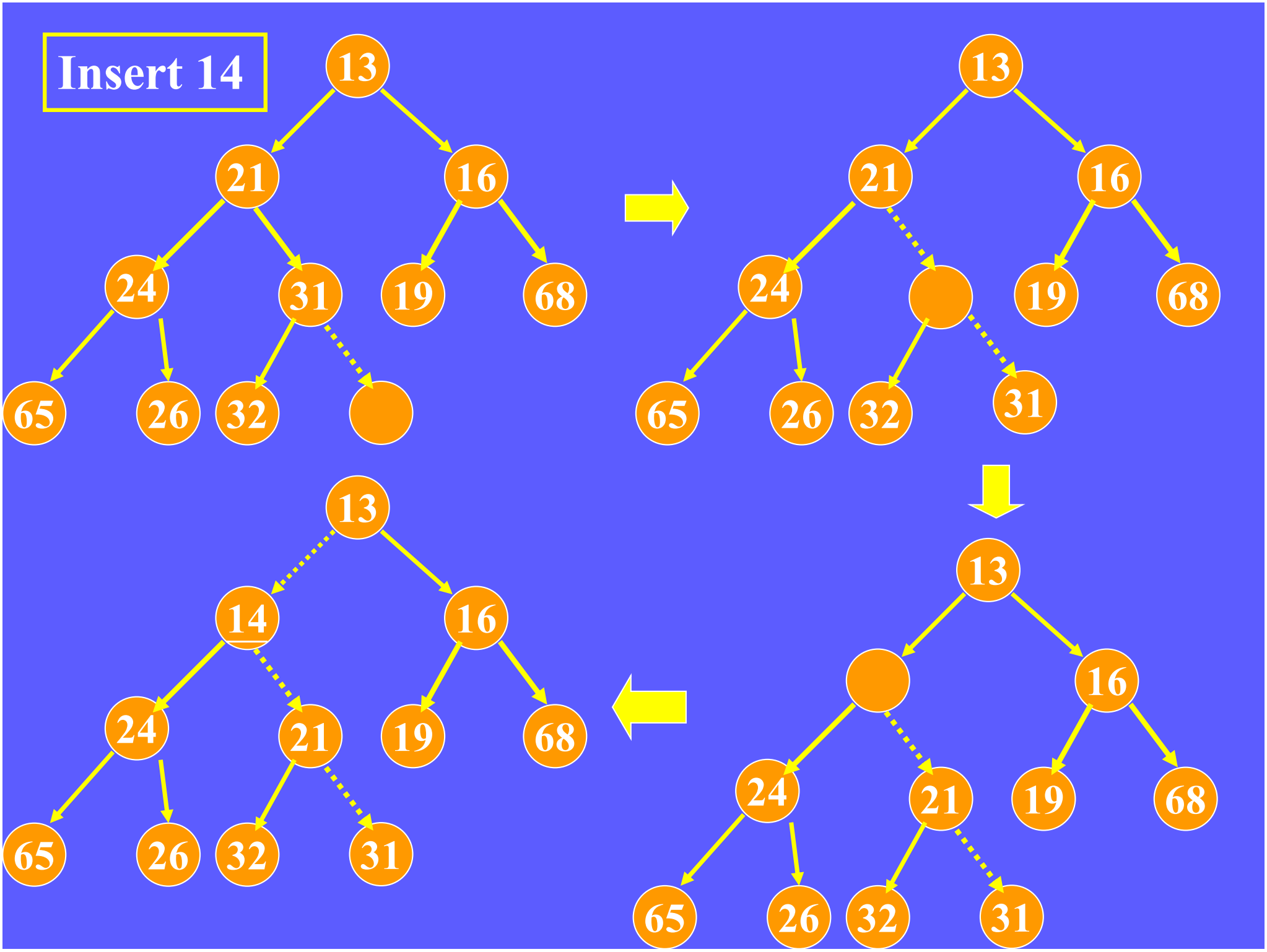

21 48 72
12 15
25 31 41
59
84 91
21,24
25,26
31,38
41,43,46
1,4,8,11
12,13
15,18,19
48,49,50
59,68
72,78
84,88
91,92,99

Heaps

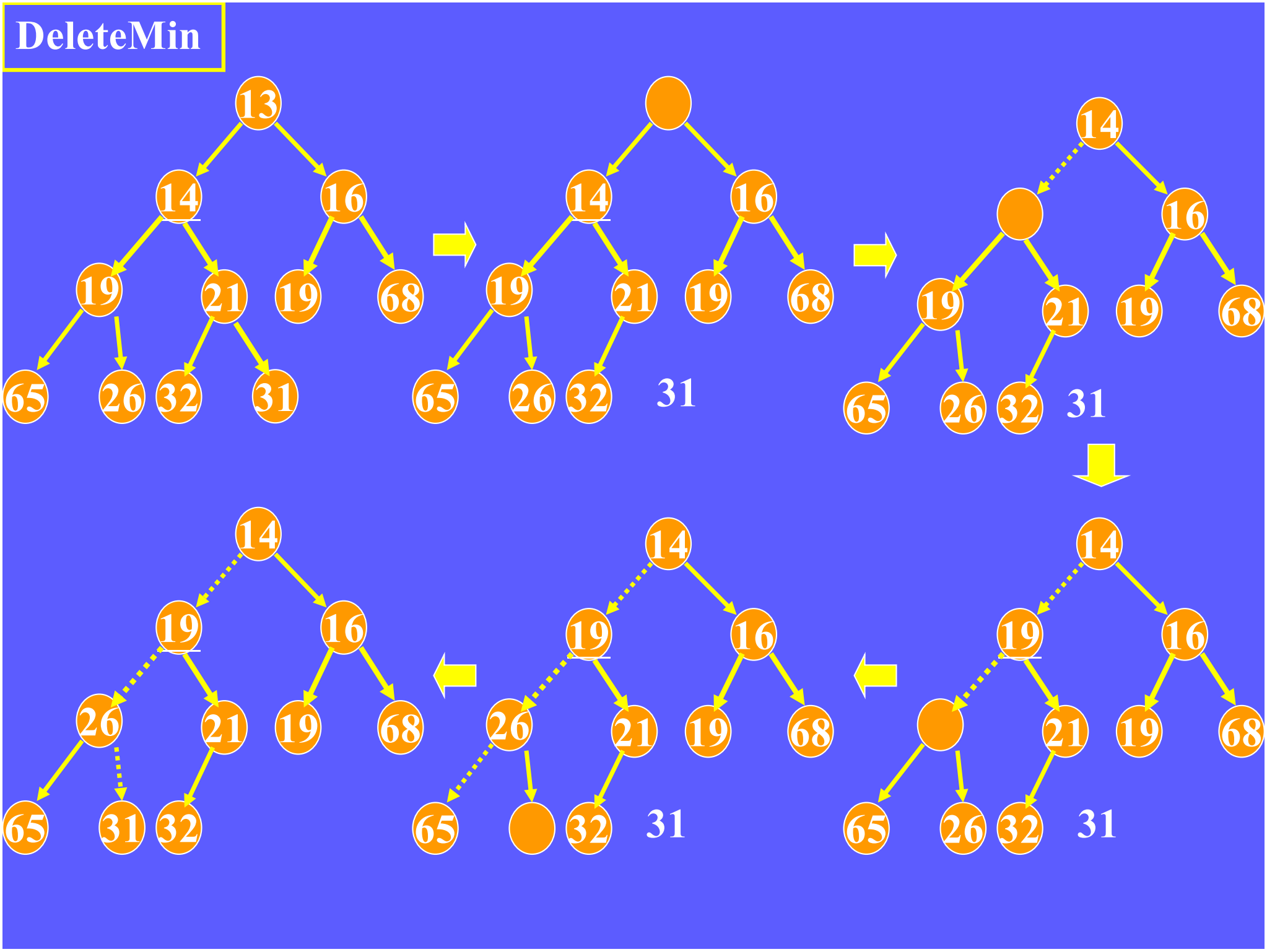
- **Heaps (Priority Queue)**
  - **Structure property: complete binary tree (Fig. 6.2)**
    - *** complete binary tree: height $\lfloor \log N \rfloor$**
    - *** array implementation of heap (Fig. 6.3)**
  - **Order property: for each node X, the key in the parent of X is smaller than or equal to the key in X. (Fig. 6.5)**



	13	21	16	24	31	19	68	65	26	32	
0	1	2	3	4	5	6	7	8	9	10	11


Insert 14
13
21
16
24
31
19
68
65
26
32
13
21
16
24
19
68
65
26
32
31
13
16
24
21
19
68
65
26
32
31
13
14
16
24
21
19
68
65
26
32
31

```
void Insert( ElementType X, PriorityQueue H )
{
    int i;
    if( IsFull( H ) )
    {
        Error( "Priority queue is full" );
        return;
    }
    for  ( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
            H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
```


DeleteMin
13
14
16
19
21
19
68
65
26
32
31
14
16
19
21
19
68
65
26
32
31
14
16
19
21
19
68
65
26
32
31
14
19
16
26
21
19
68
65
31
32
14
19
16
26
21
19
68
65
32
31
14
19
16
21
19
68
65
26
32
31

```
ElementType  DeleteMin( PriorityQueue H )
{     int i, Child;
      ElementType MinElement, LastElement;
      if   ( IsEmpty( H ) )  {
           Error( "Priority queue is empty" );
           return H->Elements[ 0 ];
      };
     MinElement = H->Elements[ 1 ];
     LastElement = H->Elements[ H->Size-- ];
     for( i = 1; i * 2 <= H->Size; i = Child )  {
           Child = i * 2;
           if  ( Child != H->Size && H->Elements[ Child + 1]
                                        < H->Elements[ Child ] )
                Child++;
           if  ( LastElement > H->Elements[ Child ] )
                H->Elements[ i ] = H->Elements[ Child ];
           else
               break;
     };
     H->Elements[ i ] = LastElement;
     return MinElement;
}
```

Searching Problem

學號	姓名	性別	生日	評語
1101	徐懷鈺	女	1978/03/03	yuki
2301	孫燕姿	女	1978/07/23	美少男殺手的對手
1102	卜學亮	男	1969/09/11	凡走過必留下痕跡
1201	蔡依林	女	1980/09/15	美少男殺手
1103	劉若英	女	1973/06/01	奶茶
1301	金城武	男	1973/10/11	美少女殺手
2302	周杰倫	男	1979/01/18	新美少女殺手

- Given 學號, find 姓名、性別、生日、評語

Searching Problem有哪些方法?

請各組至少列出3種作法及

其時間複雜度。

Searching Problem

學號	姓名	性別	生日	評語
1101	徐懷鈺	女	1978/03/03	yuki
2301	孫燕姿	女	1978/07/23	美少男殺手的對手
1102	卜學亮	男	1969/09/11	凡走過必留下痕跡
1201	蔡依林	女	1980/09/15	美少男殺手
1103	劉若英	女	1973/06/01	奶茶
1301	金城武	男	1973/10/11	美少女殺手
2302	周杰倫	男	1979/01/18	新美少女殺手

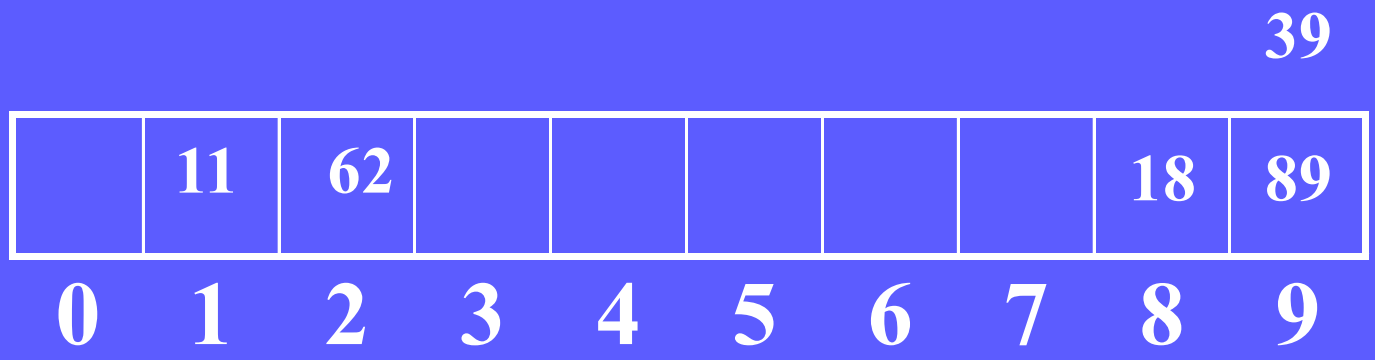
- Given 學號, find 姓名、性別、生日、評語
  - Linear search on a unsorted list
  - Binary search on a sorted list ordered by 學號
  - Search a binary search tree indexed by 學號
  - Search an AVL tree indexed by 學號
  - Search a B-tree indexed by 學號

Hash Table ADT

- **Implementation of hash tables: hashing**
- **Hashing: insertions, deletions & finds in constant time.**
  *** ordering is not supported**
- **General data structure of hashing**
  - **array of fixed size(TableSize) containing the keys**
  - **hash function**

Hashing

- $H(K) = K \% TS$,
- Example: $H(K) = K \% 10$



* insert 89, 18, 11, 62

為什麼稱之為Hash？

Hashing 搜尋的時間複雜度？

Hashing 搜尋的時間複雜度只要 O(1), 那有什麼缺點？

Collision & Overflow

- **Collision: two keys hash to the same value.**
- **Overflow: collision and the cell is full.**

Hashing

- **Static hashing**
  - **perfect hashing without collision**
  - **hashing with collision**
    - separate chaining
    - open addressing
      - linear probing
      - quadratic probing
      - double hashing
    - rehashing
- **Dynamic hashing**
  - **extensible hashing**
  - **linear hashing**

Hash Function

- **simple to compute**
- **minimize the number of collisions**
- **dependent upon all the characters in the identifiers**

  $\Rightarrow$ uniform hash function

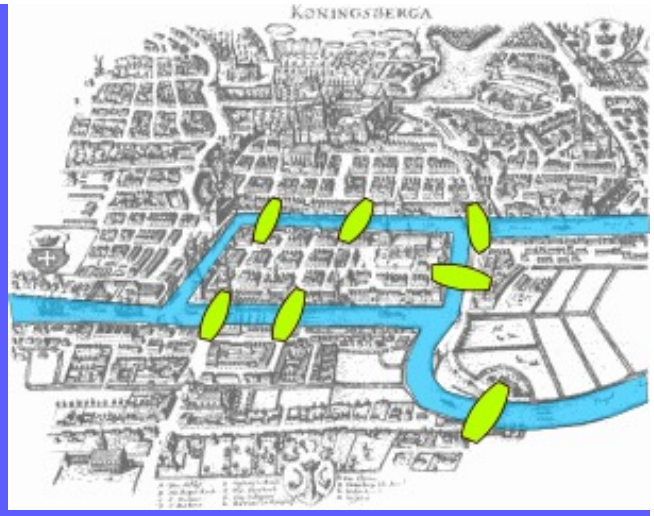
Examples of Hash Function

- **Keys are integer: h(K) = K % TS**
  - **TS(TableSize)**
  - **Bad choice of TS: a power of two, dividable by two**
  - **Good choice of TS: prime**
- **Keys are string**
  - **h(K) =($\sum$ asc(k_i)) % TS,**
  - **h(K) =(asc(k_0)+asc(k_1)*27+asc(k_2)*729) % TS,**
  - **h(K) =($\sum$ asc(k_i)*27^i) % TS, (Fig. 5.5)**

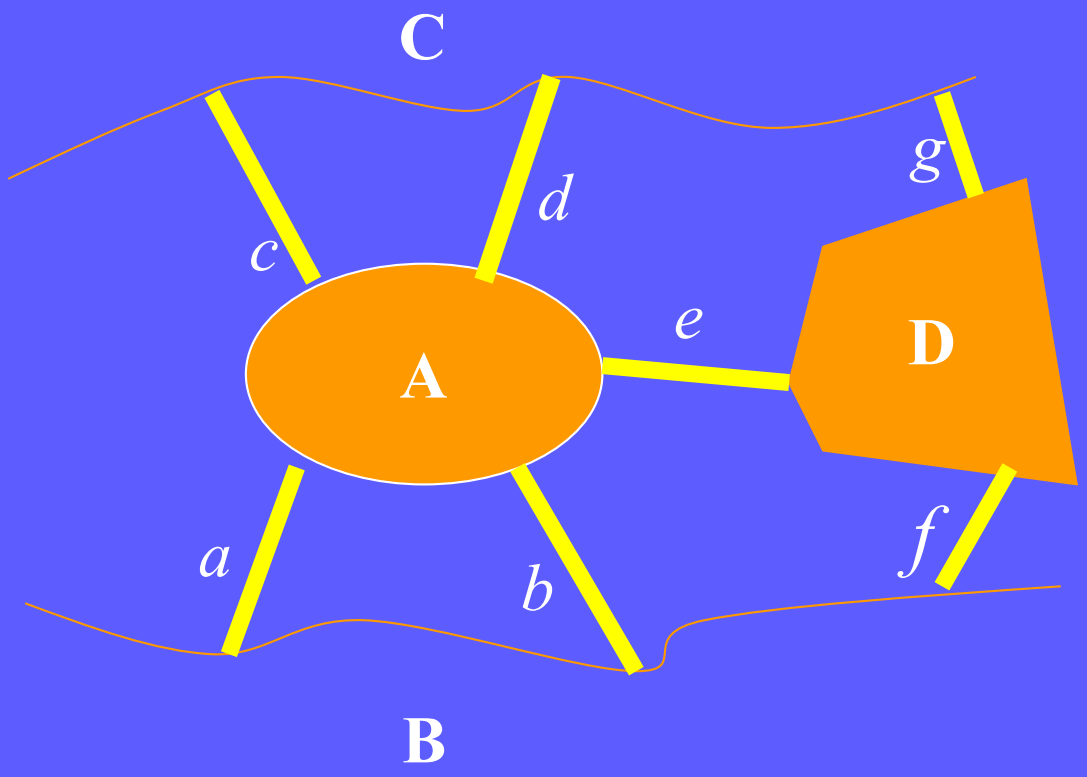
Strategies for Collisions

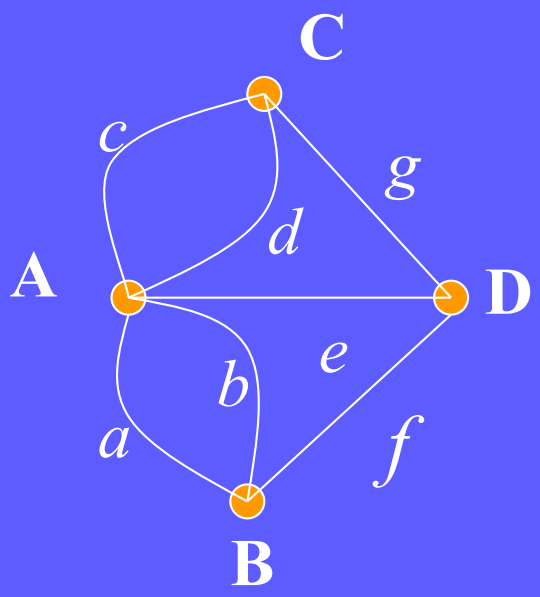
- **separate chaining**
- **open addressing**
  - **linear probing**
  - **quadratic probing**
  - **double hashing**
- **rehashing**

Graph Theory



- **1793, Euler's Koenigsberg bridge problem**



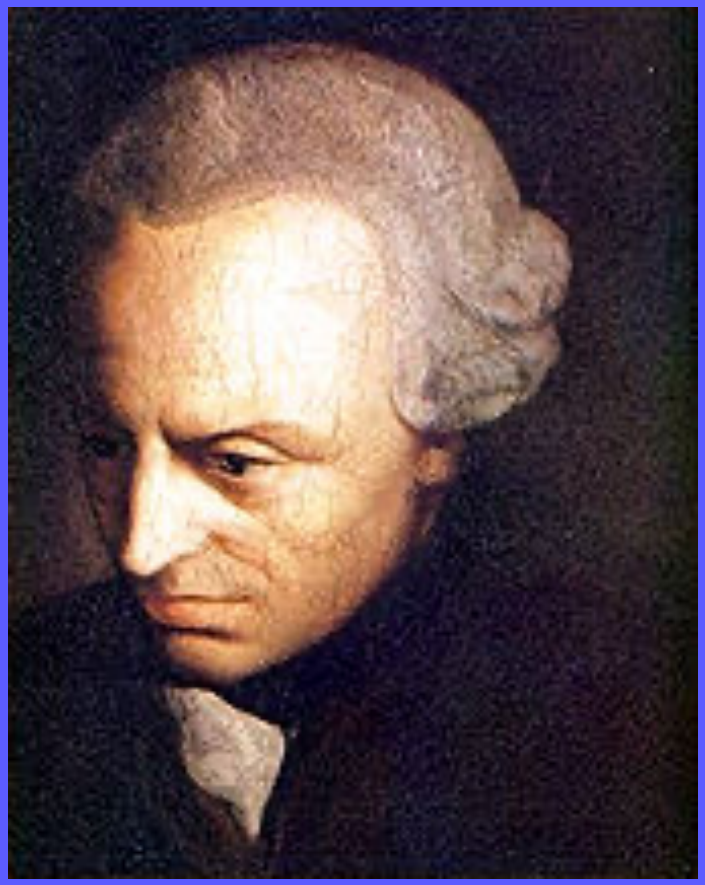


Leonhard Euler

Read Euler, read Euler, he is the master of us all

He ceased to calculate and to live

Immanuel Kant



**Two things fill the mind with ever new and increasing admiration and awe,
the more often and steadily we reflect upon them:
The starry heavens above me and the moral law within me.**

Terminology of Graph

- **Graph: G=(V, E), V: a set of vertices, E: a set of edges**
- **Edge (arc): a pair (v,w), where v, w $\in$ V**
- **Directed graph (Digraph):**
  graph if pairs are ordered (directed edge)
- **Adjacent: w is adjacent to v if (v,w) $\in$ E**
- **Undirected graph: if (v,w) $\in$ E, (v,w)=(w,v)**

人立方 Beta
Renlifang关系搜索

首页 关系百科 Q20读心机器人 微博关系图 六度搜索

人物关系图 六度搜索

林志玲

林志玲
同名人俱乐部

林志玲
标签:演员,主...

创建新的林志玲词条

分享到微博

林志玲 彭靖惠 周杰伦 陈绮贞

点击上方人物，自定义关系链

张悬 袁咏琳 钟成虎 彭靖惠 吴立琪 汤唯 陈绮贞 周杰伦 梁静茹 何欣穗 王怡仁 蔡健雅 林志玲 温岚 隋棠 言承旭

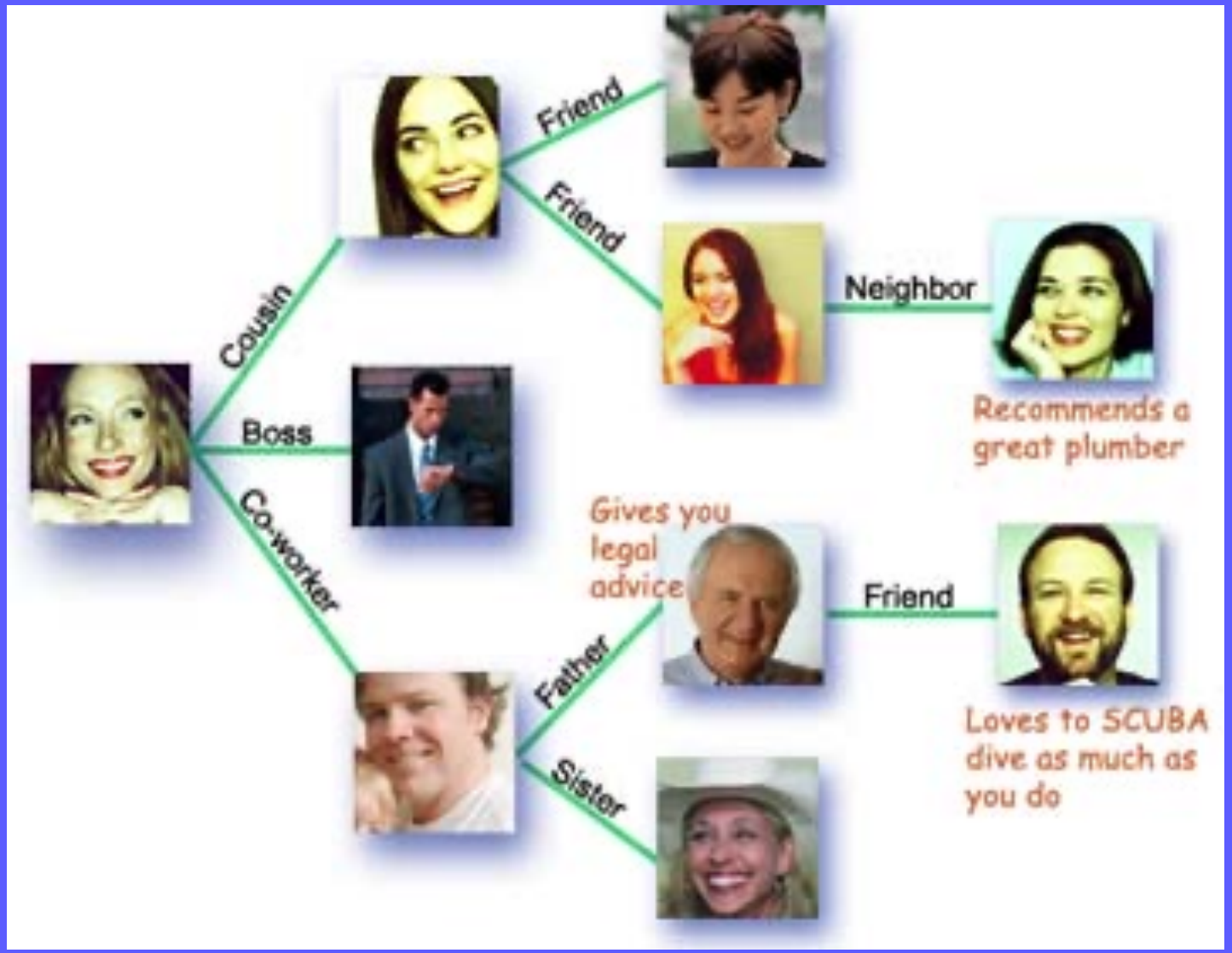
陈绮贞

陈绮贞
同名人俱乐部

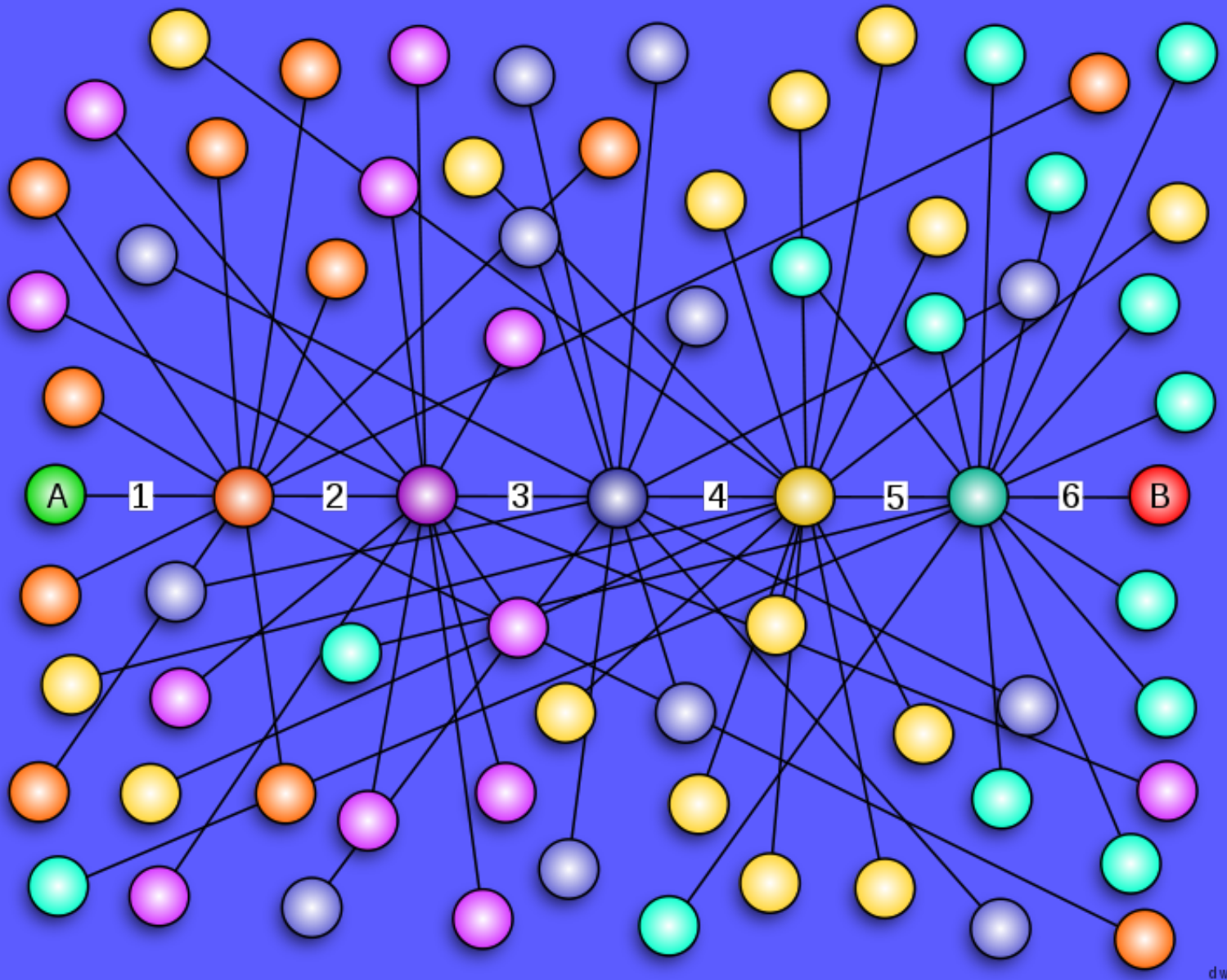
陈绮贞
标签:独立音乐...

创建新的陈绮贞词条

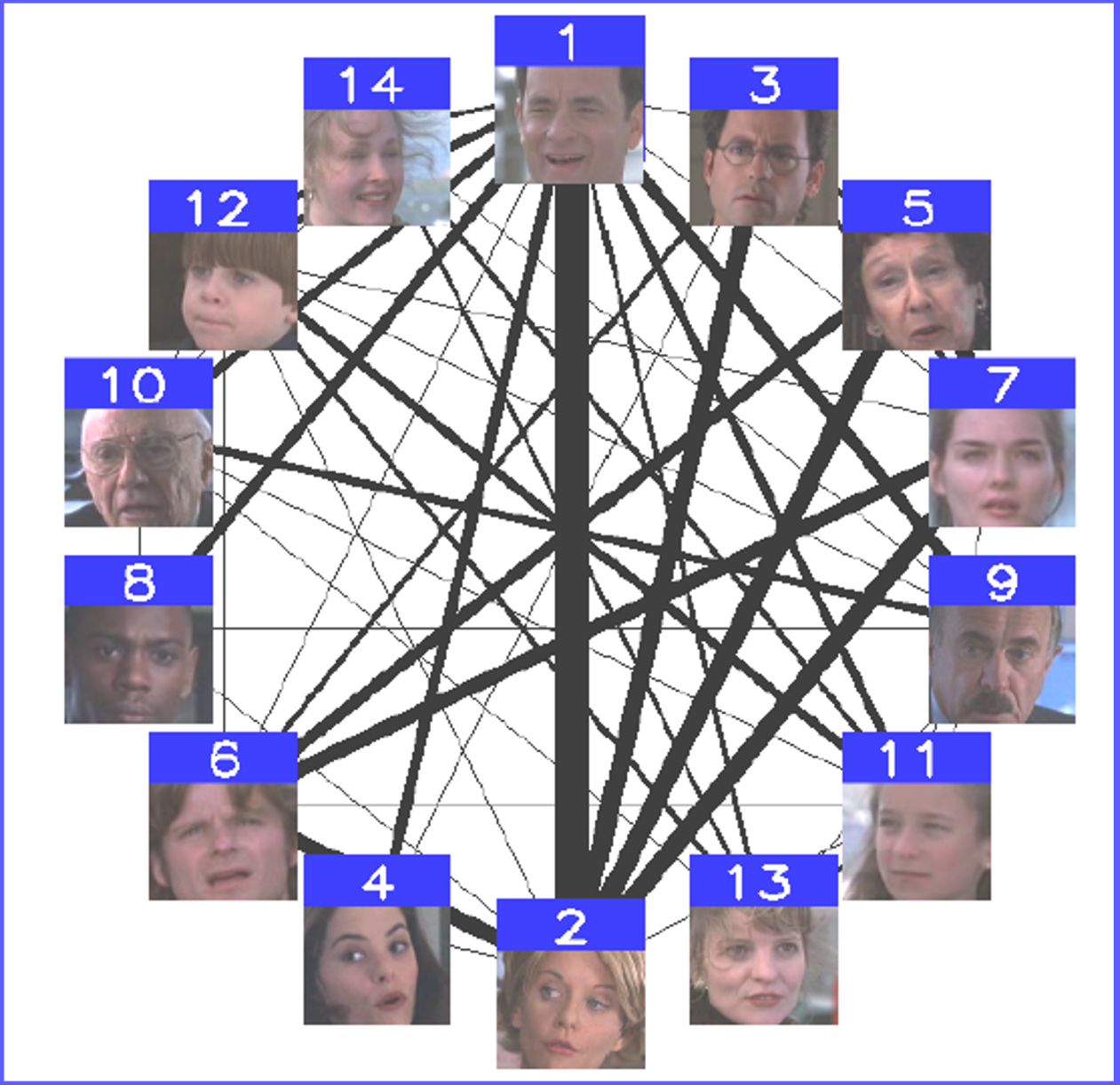
Social Networks



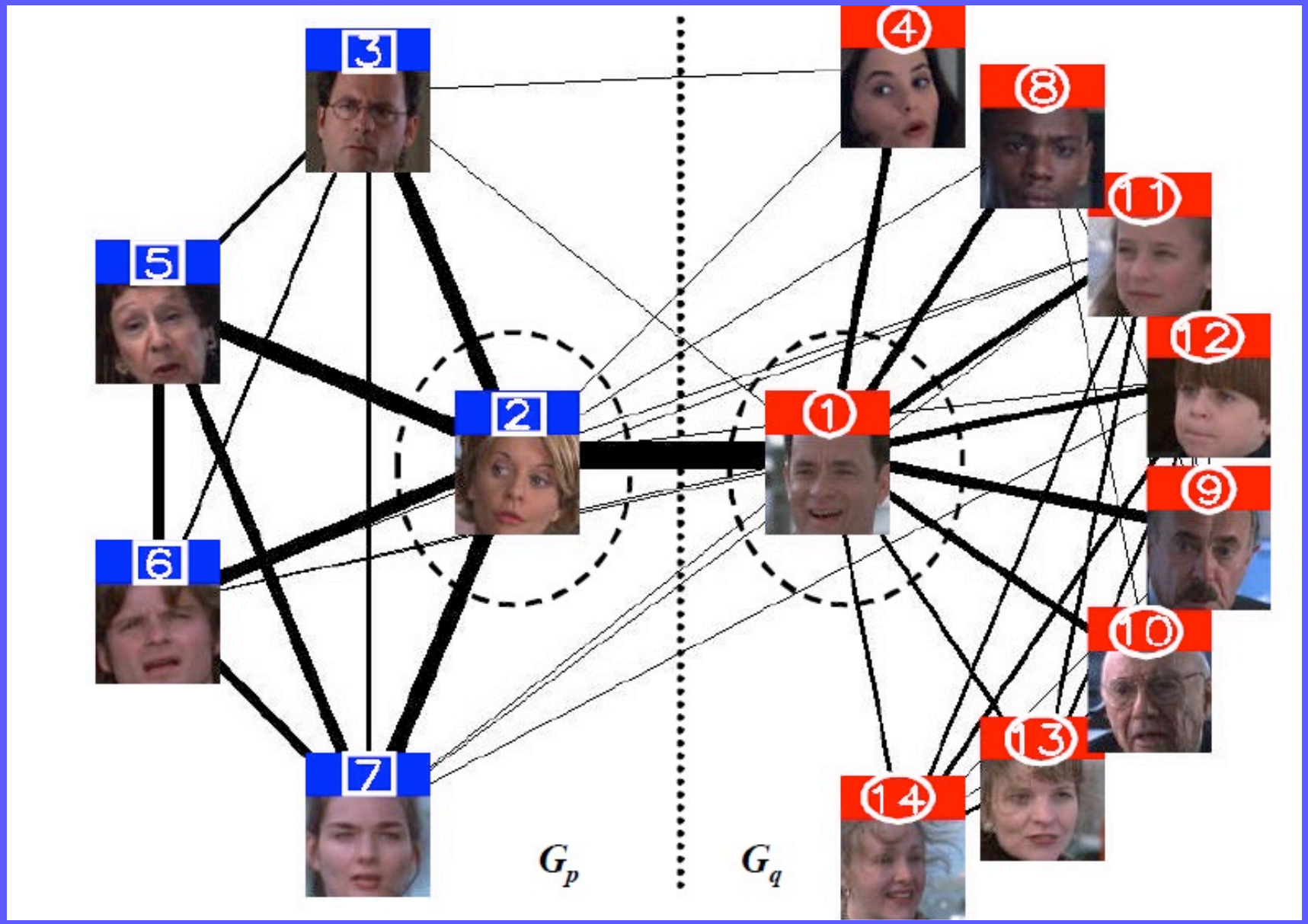
Six Degree of Separation



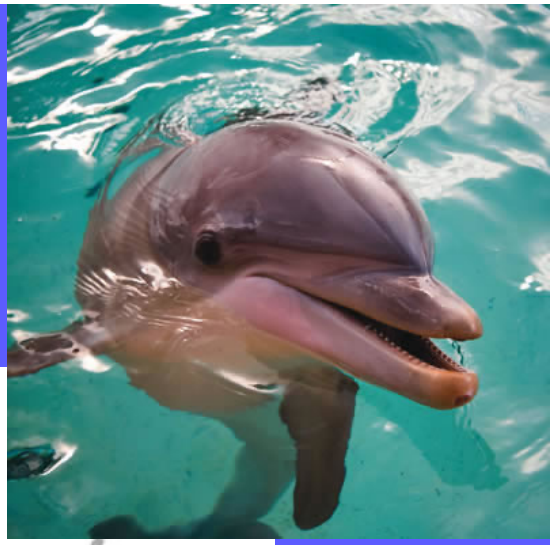
Community Detection

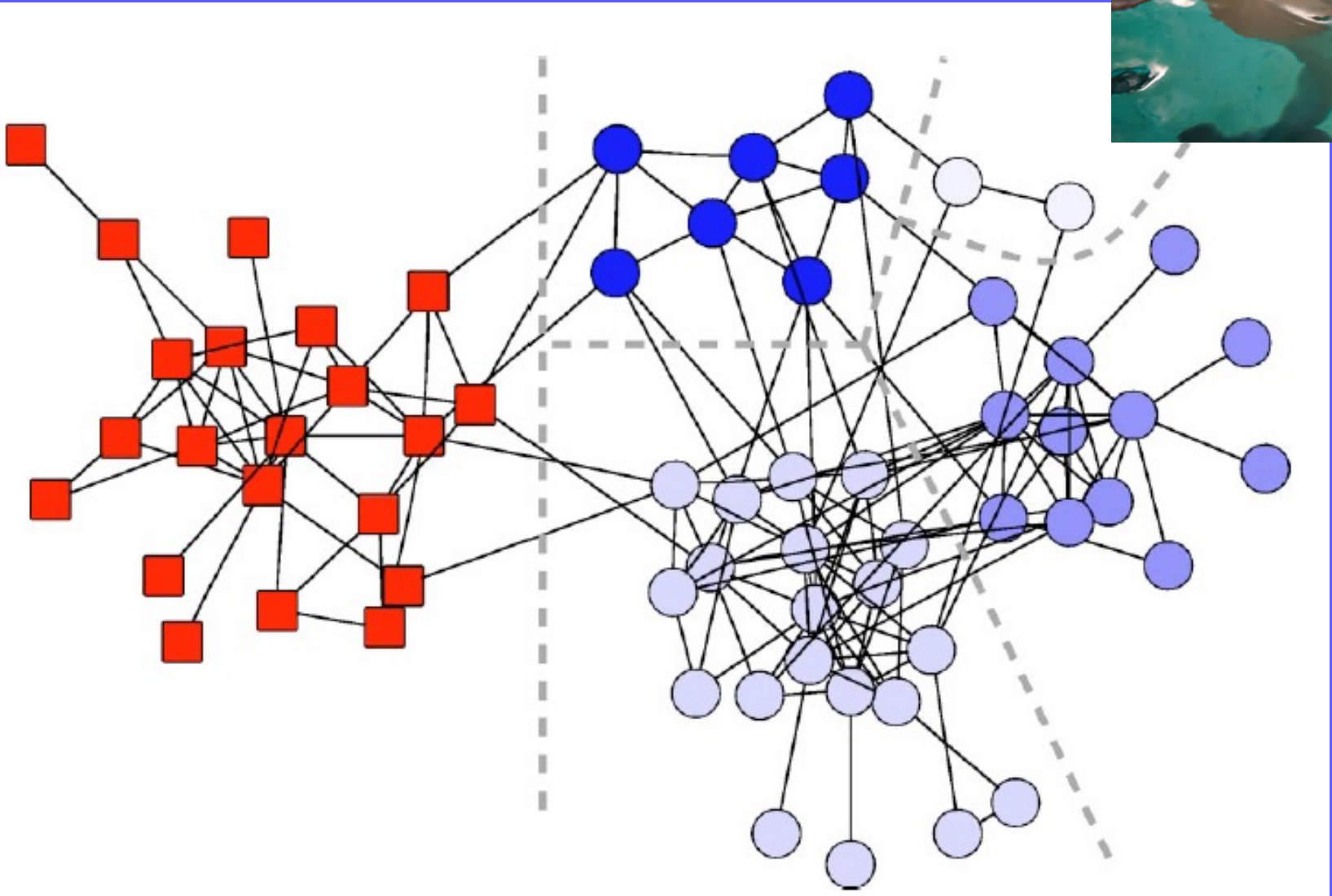


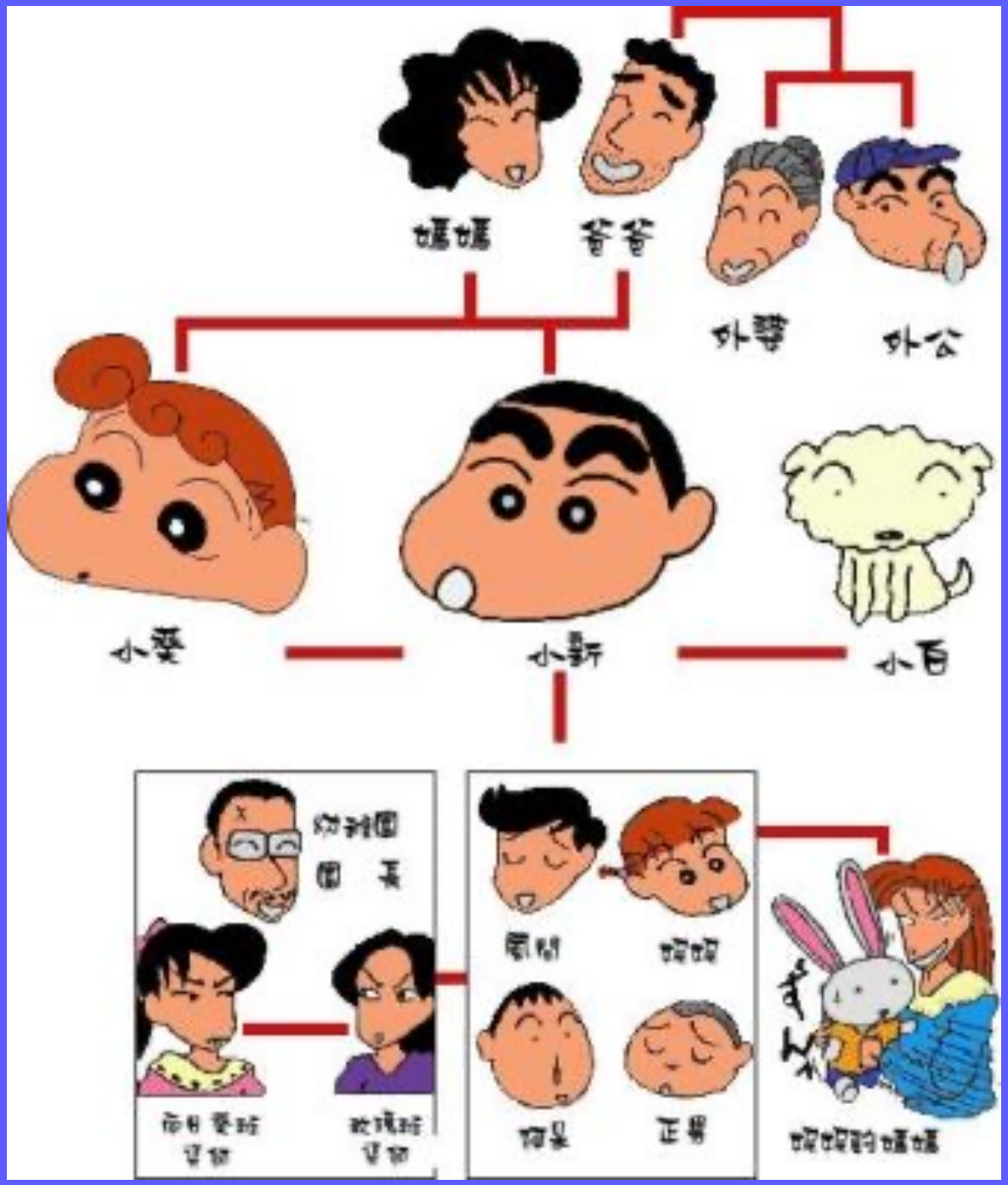
Community Detection (cont.)



Community Detection

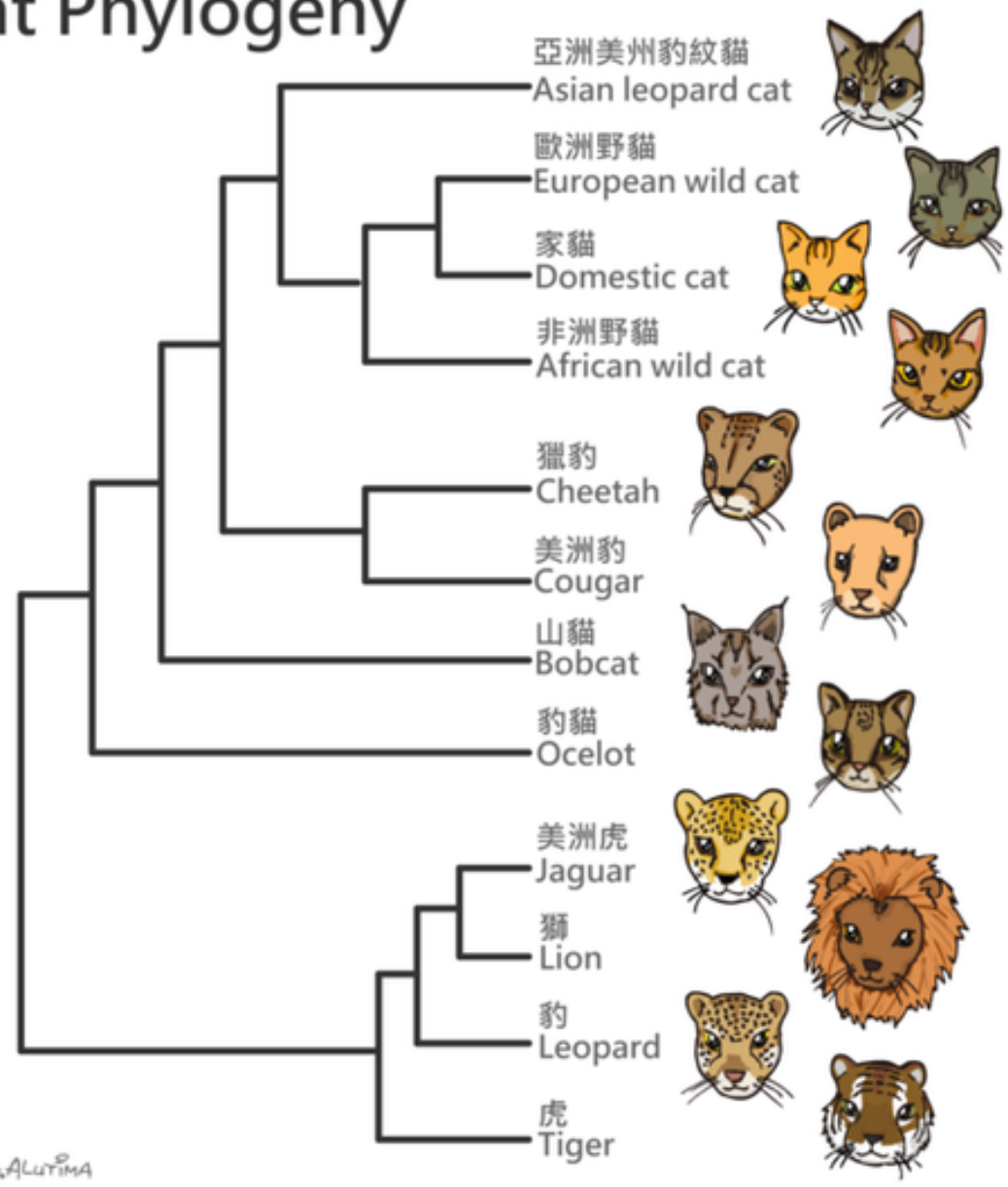





媽媽
爸爸
外婆
外公
小愛
小新
小白

貓的演化(簡單版)

Cat Phylogeny



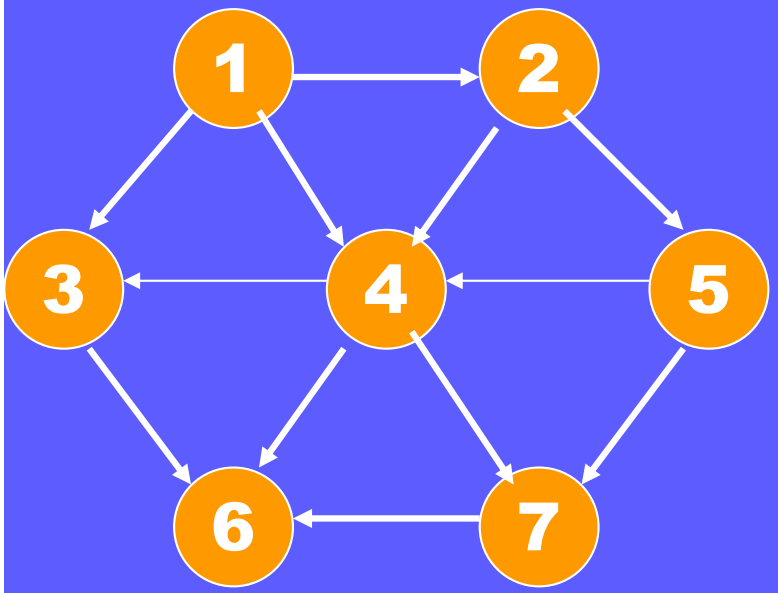
寫Graph的程式時，

如何表示Graph？

Representation of Graphs

- **Data structures for representation of graphs**
  - **adjacency matrix representation**
  - **adjacency list representation**

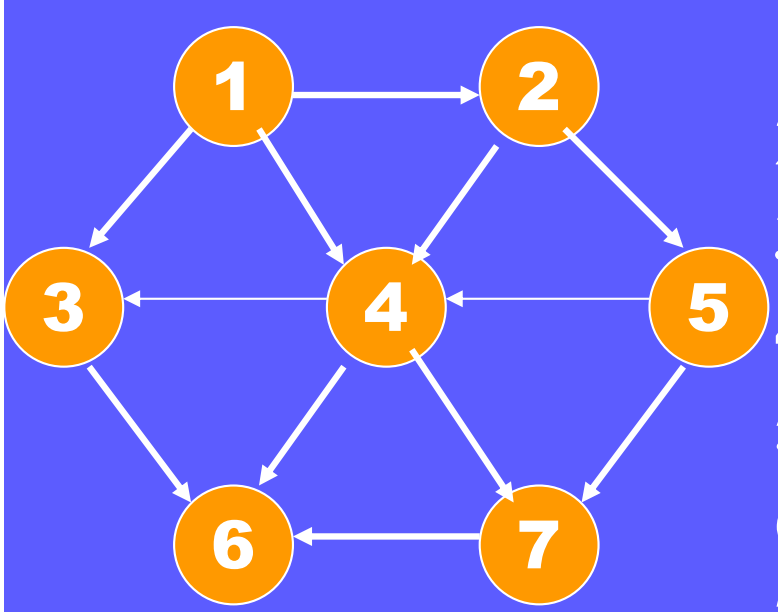
Adjacency Matrix Representation

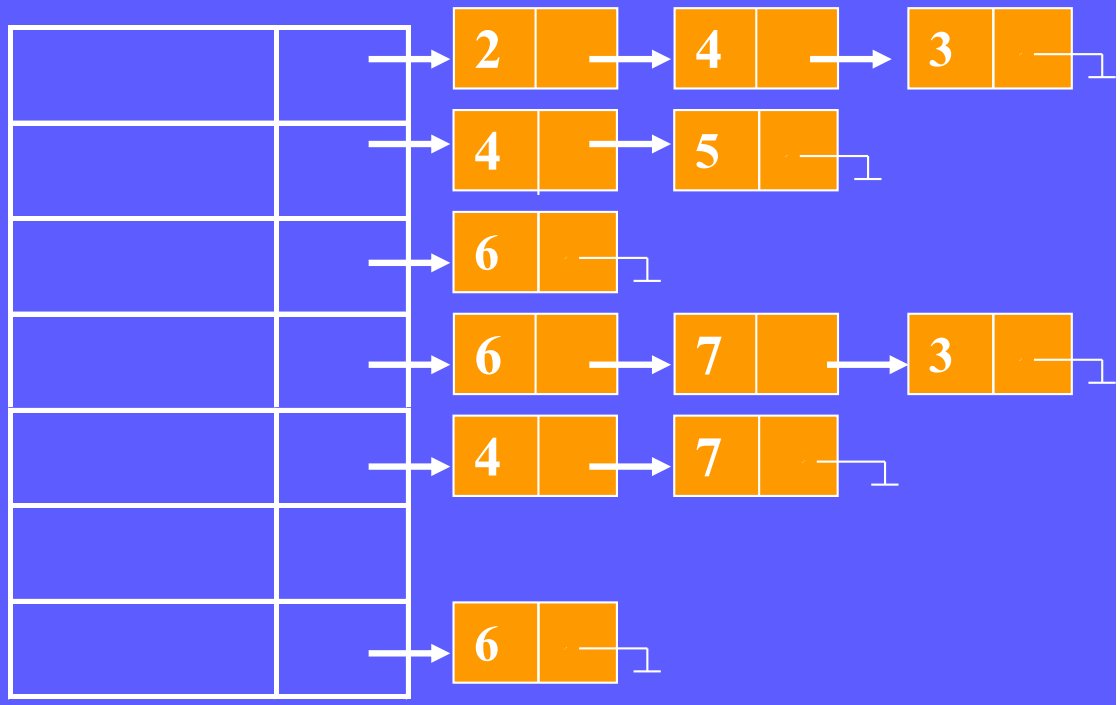


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

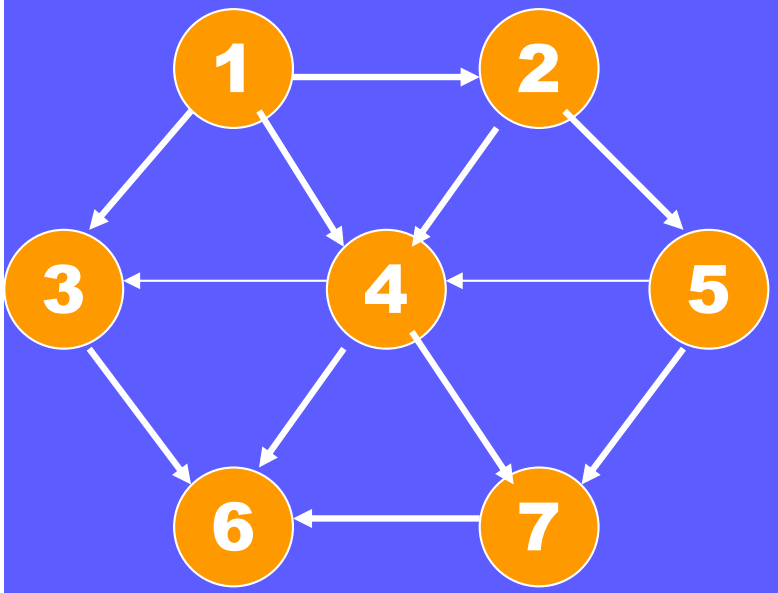
*** Undirected graph: symmetric matrix**

Adjacency List Representation





Adjacency Matrix Representation

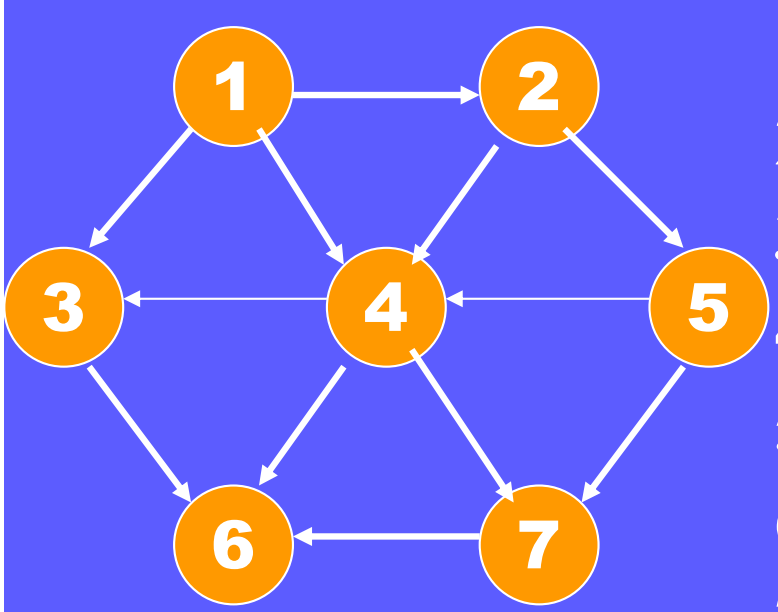


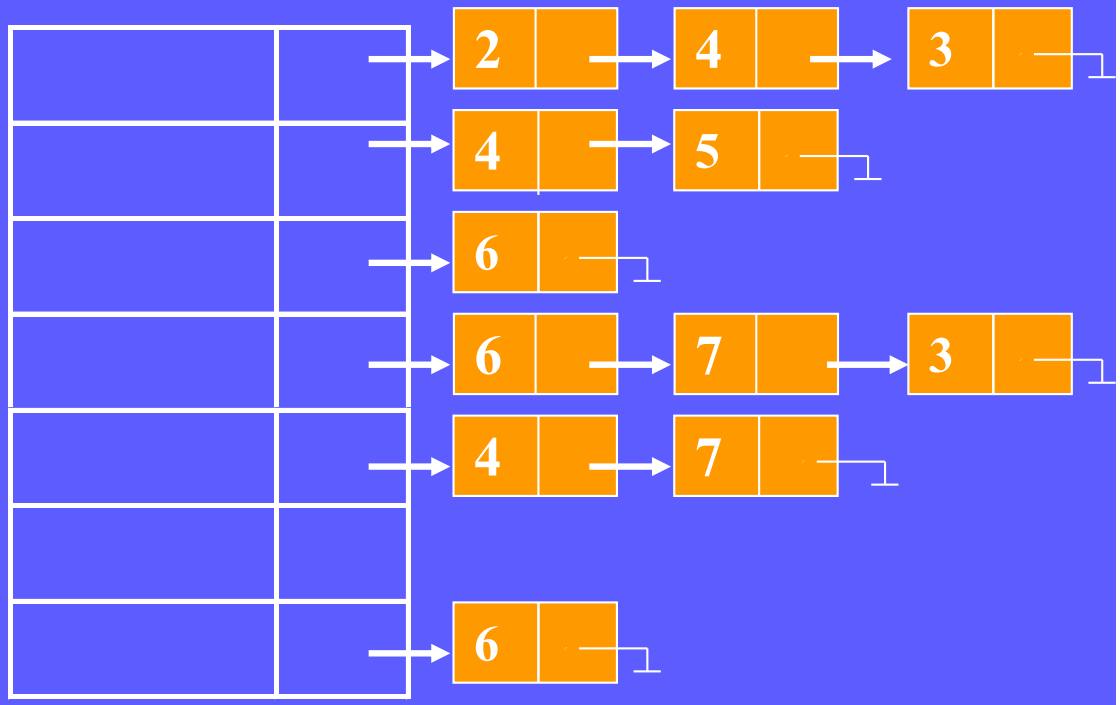
	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

- Space: $\Theta(|V|^2)$, good for dense, not for sparse

  * Undirected graph: symmetric matrix

Adjacency List Representation





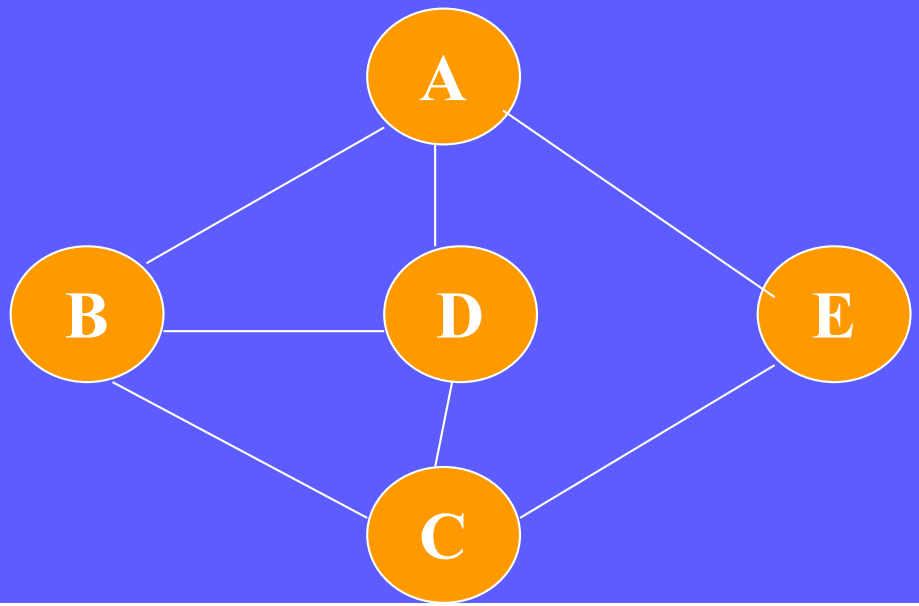
- Space: O(|V|+|E|) good for sparse

Facebook上的好友關係，
適合用哪種表示法？

Graph Traversal

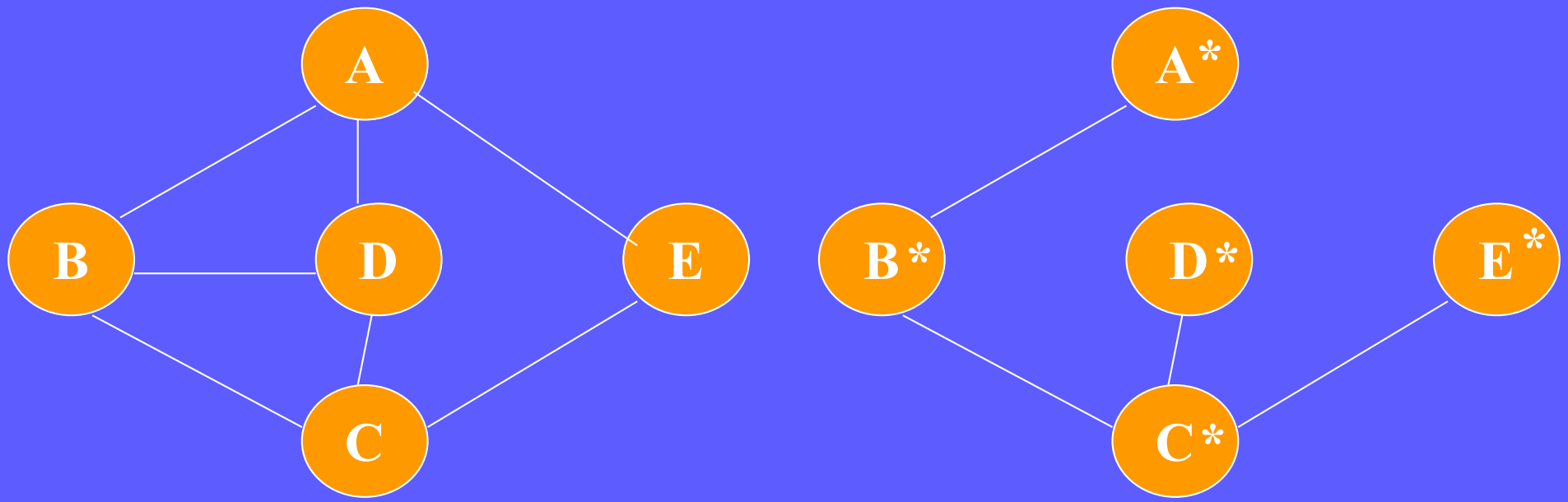
Traversal of Graph

- **Given a undirected graph, visit all the vertices that are reachable(connected) from vertex *v***
  - **Depth First Search (DFS)**
  - **Breadth First Search(BFS)**



Depth First Search

- Depth First Search: generalization of preorder traversal
- Starting from vertex *v*, process *v* & then recursively traverse all vertices adjacent to *v*.
- To avoid cycles, mark visited vertex

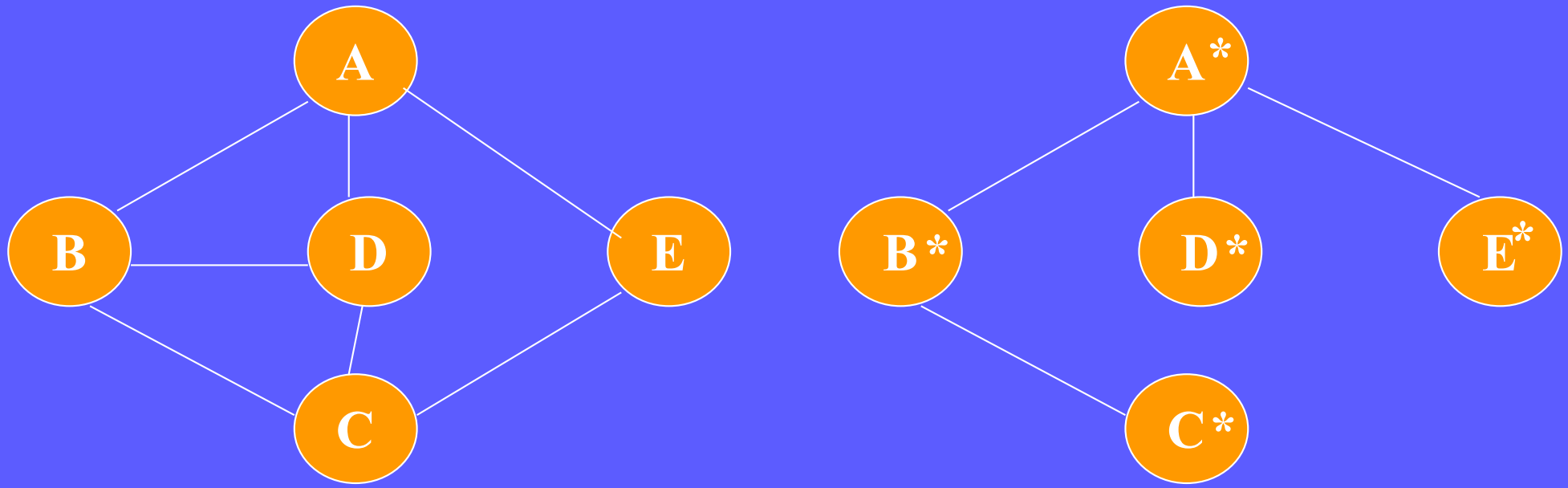


Algorithm of DFS

```
void DFS(Vertex V)
{
    Visited[V]=True;
    for each W adjacent to V
        if (!Visited[W])
            DFS(W);
}
```

Breadth First Search

- **Breadth First search (BFS): level order tree traversal**
- **BFS algorithm: using queue**



Breadth First Search Algorithm

```
void BFS(Graph G);    /*   O(|E|+|V|)   */
{    queue Q;
     vertex V,W;
     Q=CreateQueue(NumVertex);
     MakeEmpty(Q);
     Visted[V]=True;
     Enqueue(V,Q);
     While ( !IsEmpty(Q) )
     {      V=Dequeue(Q);
            for each W adjacent to V
                 if ( !Visited[W] )
                 {   Visited[W]=True;
                     Enqueue(W,Q);
                  }
     }
}
```

Topological Sorting

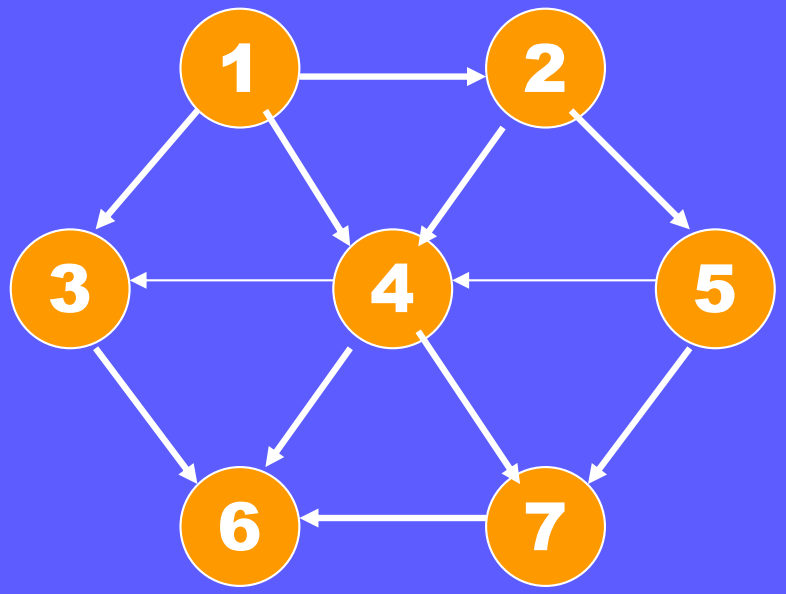
Topological Sorting

- **Topological sorting:**

  ordering of vertices in a DAG such that

  if there is a path from v_i to v_j,

  then v_j appears after v_i in the ordering.

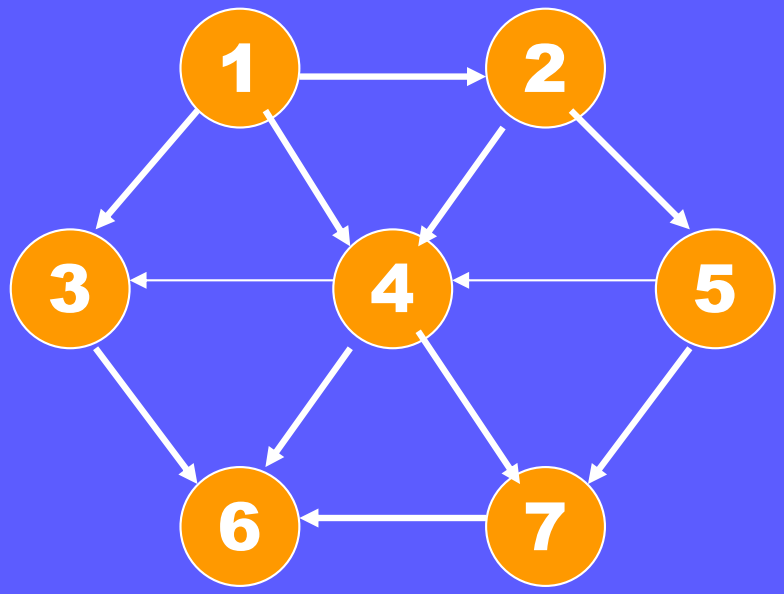


1, 2, 5, 4, 3, 7, 6

Topological Sorting有什麼應用 ？

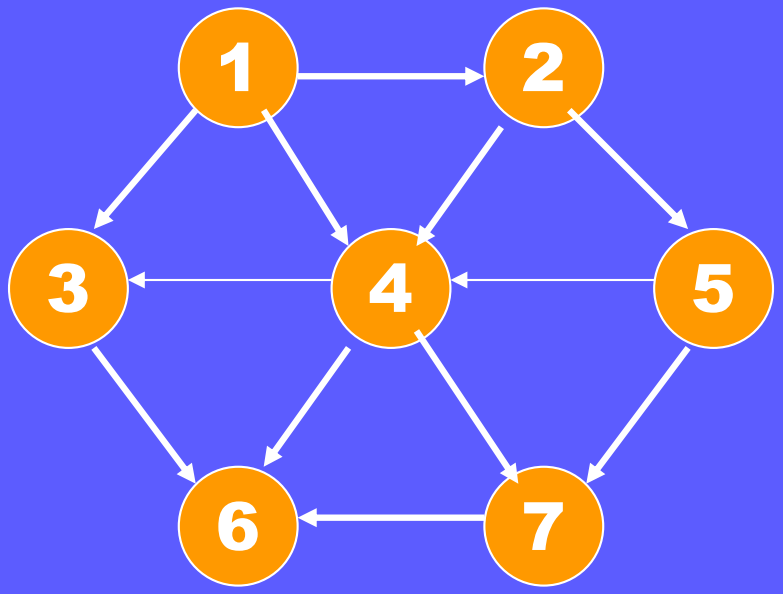
Topological Sorting

- if there is a path from v_i to v_j,
  then v_j appears after v_i in the ordering.
- Prerequisite of Courses
  - 2: {1}, 3: {1, 4}, 4:{1, 2, 5}, 5:{2}, 6:{3, 4, 7}, 7:{4, 5}
  - Ordering of course taking



1, 2, 5, 4, 3, 7, 6

如何做Topological Sorting ？



1, 2, 5, 4, 3, 7, 6

	Deg.	
1	0	1
2	1--	
3	2--	
4	3--	
5	1	
6	3	
7	2	

	Deg.	
1	0	1
2	0	2
3	1	
4	2--	
5	1--	
6	3	
7	2	

	Deg.	
1	0	1
2	0	2
3	1	
4	1--	
5	0	3
6	3	
7	2--	

	Deg.	
1	0	1
2	0	2
3	1--	
4	0	4
5	0	3
6	3--	
7	1--	

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	2--	
7	0	

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	1--	
7	0	6

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	0	7
7	0	6

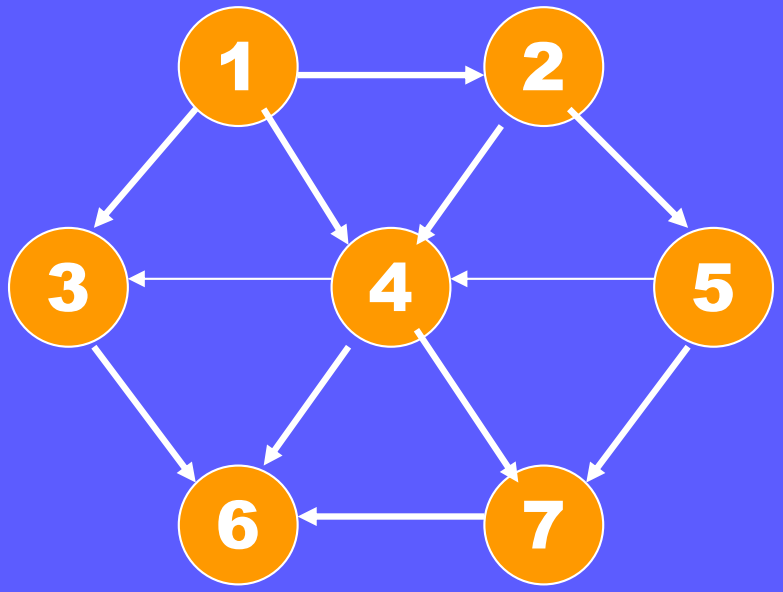
Algorithm for Topological Sorting

```
Void Topsort(Graph G)      /* O(|V|^2)  */
{
  int Counter;
  vertex V,W;
  for (Counter=0; Counter < NumVertex; Counter++)
  {
      V=FindNewVertexOfDegreeZero();
      TopNum[V]=Counter;
      For each W adjacent to V
         Indegree[W]--;
  };
};
```

這個演算法時間複雜度是 ？

時間複雜度可以降低嗎 ？

哪個步驟可以改進時間複雜度?



1, 2, 5, 4, 3, 7, 6

	Deg.	
1	0	1
2	1--	
3	2--	
4	3--	
5	1	
6	3	
7	2	

2

	Deg.	
1	0	1
2	0	2
3	1	
4	2--	
5	1--	
6	3	
7	2	

5

	Deg.	
1	0	1
2	0	2
3	1	
4	1--	
5	0	3
6	3	
7	2--	

4

	Deg.	
1	0	1
2	0	2
3	1--	
4	0	4
5	0	3
6	3--	
7	1--	

3
7

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	2--	
7	0	

3
7

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	1--	
7	0	6

6

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	0	7
7	0	6

Improved Algorithm for Topological Sorting

```
void Topsort(Graph G);    /*   O(|E|+|V|)   */
{    queue Q;
     int Counter=0;
     vertex V,W;
     Q=CreateQueue(NumVertex);     MakeEmpty(Q);
     for each vertex V
          if  (Indegree[V] == 0)
               Enqueue(V,Q);
     While (!IsEmpty(Q)) {
          V=Dequeue(Q);
          TopNum[V] = ++Counter;
          for each W adjacent to V
               if (--Indegree[W] == 0)
                    Enqueue(W,Q);        }
     if (Counter != NumVertex)
          Error("Cycle!");
     DisposeQueue(Q);
}
```

利用Queue之後，
時間複雜度是 ？

Summary: Data Structures

- Array
- Linked List
- Stack
- Queue
- Tree
- Binary Search Tree, AVL Tree, B-Tree, Red-Black Tree
- Heap
- Hashing (Dictionary)
- Graph