資料結構與演算法

陳怡芬

什麼是Data structure ?

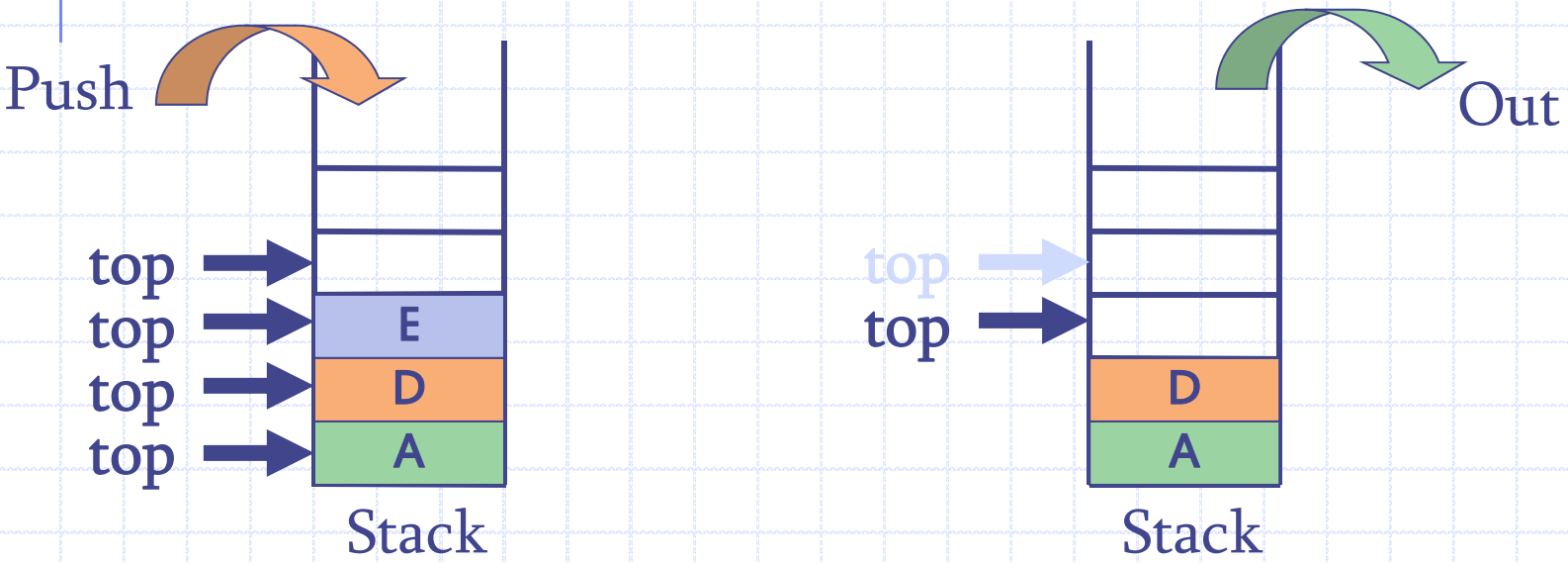
- 將資料群組織起來的抽象資料型態，稱為資料結構。

典型的資料結構

- 資料表格(Table)
- 堆疊(stack)
- 佇列(queue)
- 串列(list)
- 樹(tree)
- 圖形(graph)
  - table, stack, queue：可用陣列表現出來
    List , tree, graph：適合用指標表現出來。

堆疊(Stack)

- 將資料依序從堆疊下面儲存起來，並視需要從堆疊的上面將資料取出的方式之資料結構，稱為堆疊。



堆疊(Stack)

- 【特性】：後進先出(LIFO)
  LIFO：last in first out。
- 【動作】：**push** ：儲存資料進堆疊。
  pop ：將資料從堆疊中取出。

Push 的演算法：

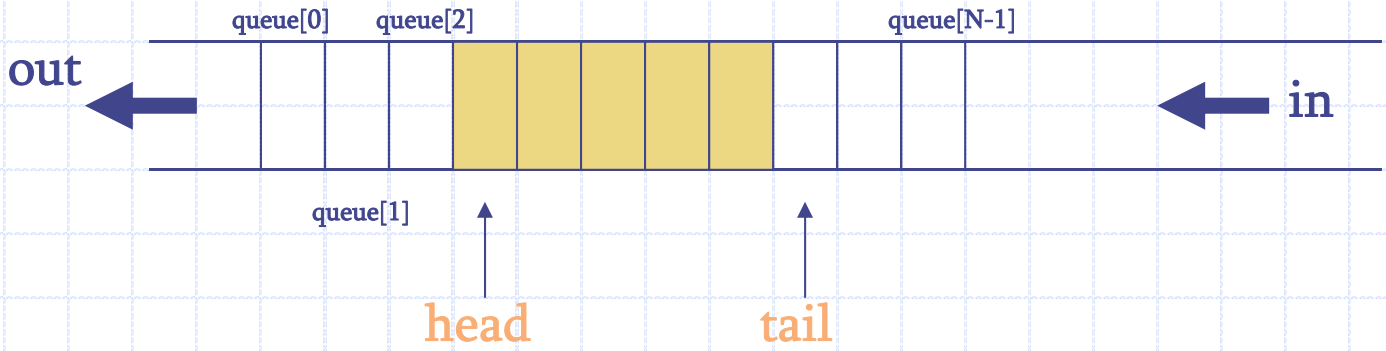
```
int top=0;    //top初值為0
push(n)
{
    if (top<MaxSize){
      stack[top]=n;
      top++;
      return 0;
    }
      else
        return -1;
}
```

Pop 的演算法：

```
 pop( )
{
      if (top>0){
        top--;
        k=stack[top];
        return k;
      }
      else return -1;
}
```

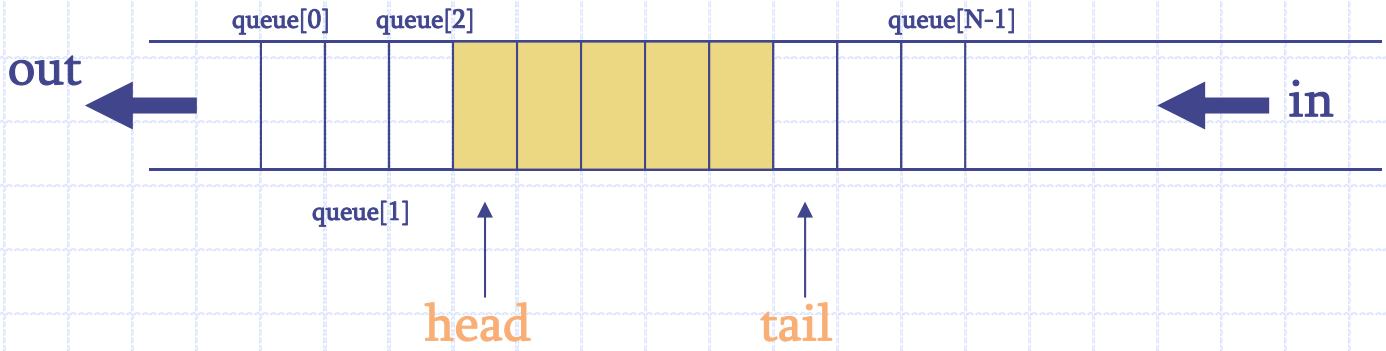
佇列(queue)

- 處理資料的方式先進先出。
- FIFO：First In First Out



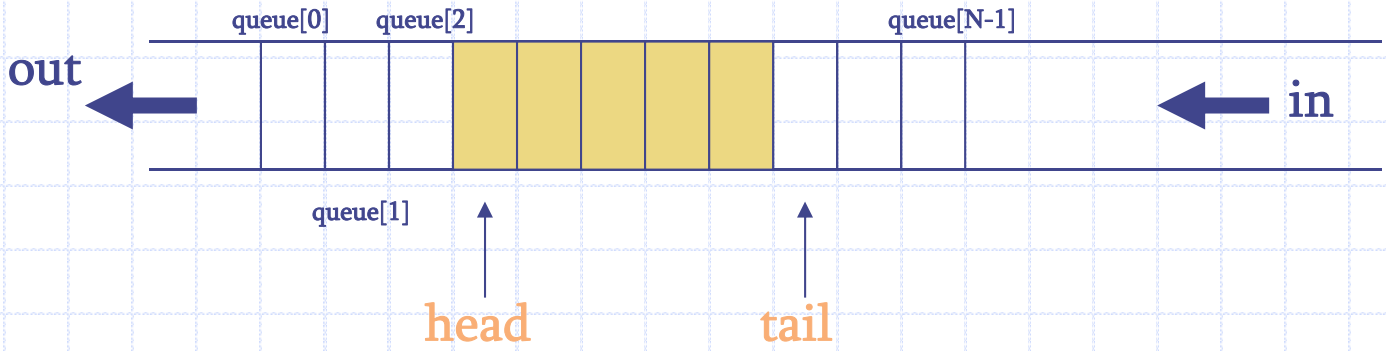
佇列(queue)

佇列(queue)



- 現設有queue[0]至queue[n-1]共n個元素的陣列，表示佇列開頭的指標設為head，表示佇列尾端的指標設為tail。
- 取出資料：在head進行。head=head+1。
- 儲存資料：在tail位置進行。tail=tail+1;
- 佇列初始狀態：head=tail=0
- 佇列為空：當head=tail時。
- 佇列為滿：tail+1為 n 時。

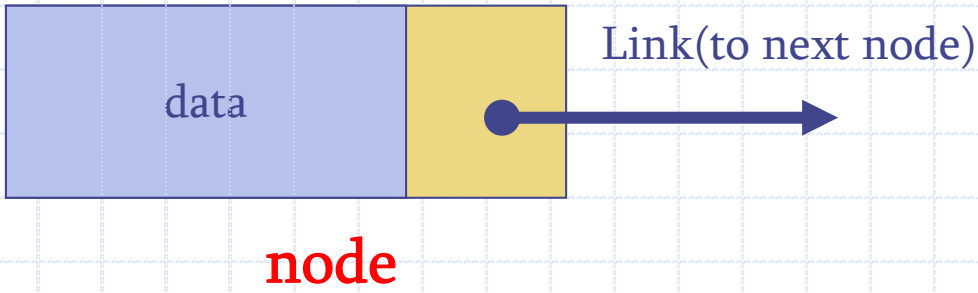
佇列(queue)

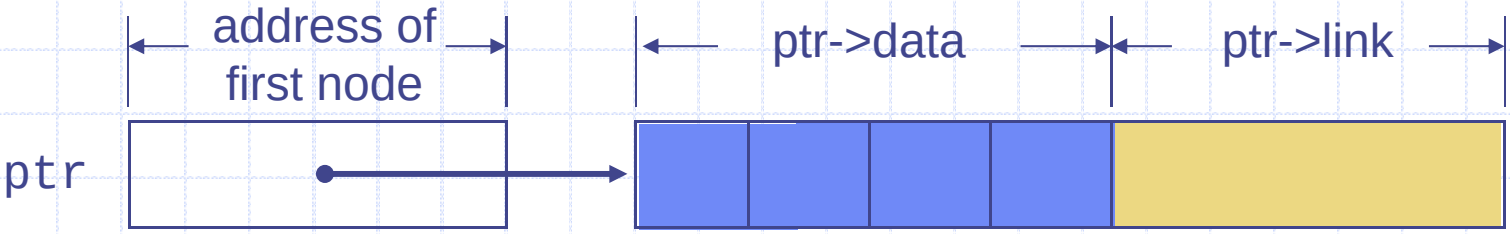


- 解決佇列使用一次就無法使用的問題：當head=tail=n時。
- 將陣列尾端queue[n-1]與開頭queue[0]連結起來成為一個環形佇列。

鏈結串列 (Linked List)

- 鏈結串列是一種有順序的串列





Linked List的優點

- 它比循序的串列(sequential list ，如陣列)更容易做任意資料的「插入」(insertions)與「刪除」(deletions)。
- 在「*mat*」和「*cat*」之間加入「 *sat* 」:
  1. Get a node that is currently unused; let its address be paddr.
  2. Set the data field of this node to *mat*.
  3. Set paddr’s link field to point to the address found in the link field of the node containing *cat*.
  4. Set the link field of the node containing *cat* to point to paddr.

以結構定義一個Link List

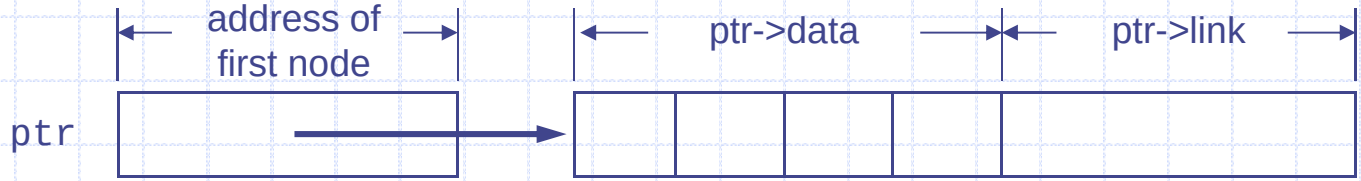
- 必要的宣告如下：

```
typedef struct list_node *list_pointer;
typedef struct list_node {
        char data[4];
        list_pointer link;
        };
list_pointer ptr = NULL;
```

建立一個新節點(node)

- 產生一個新的 node

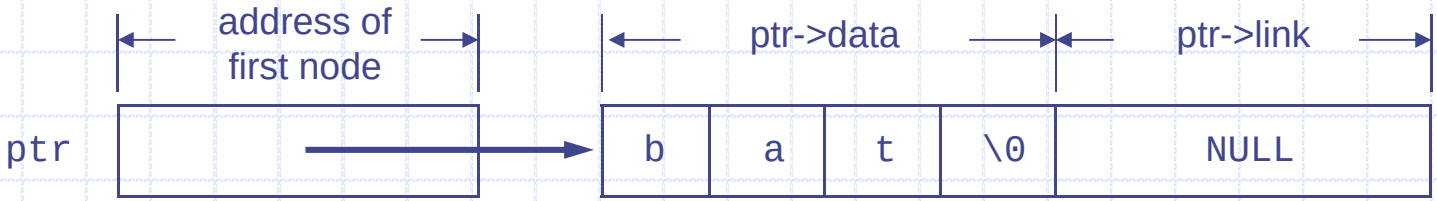
  ptr = (list_pointer) malloc(sizeof(list_node)); //配置一個指標空間



- 把字串資料 “ *bat* “放入 list 中

  strcpy(ptr->data, “bat”);

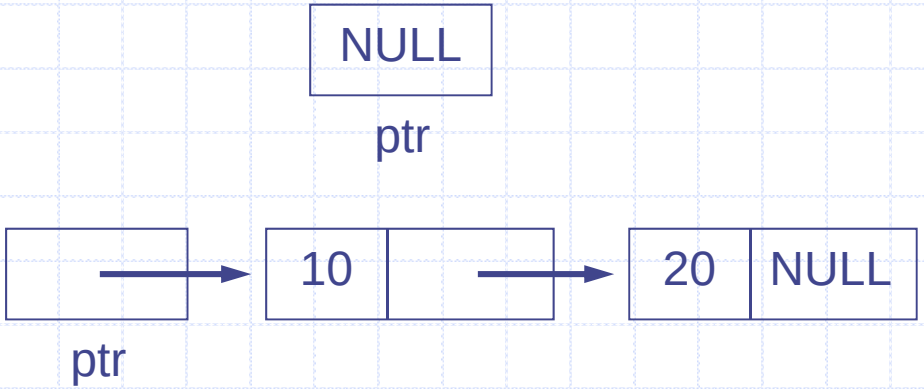
  ptr->link = NULL;



Create a two-node list

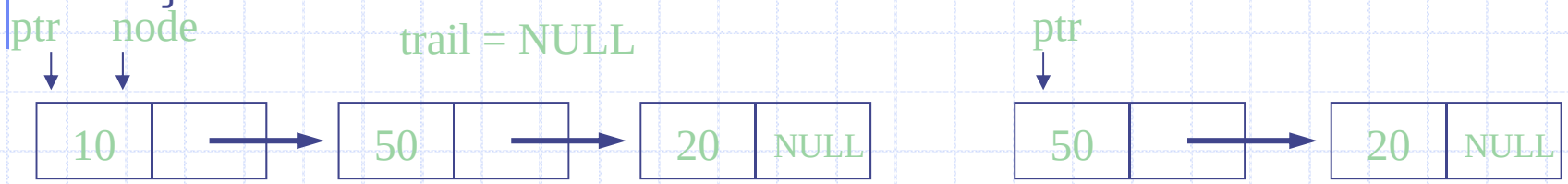
```
typedef struct list_node *list_pointer;
typedef struct list_node {
        int data;
        list_pointer link;
        };
list_pointer ptr = NULL;

list_pointer create2()
{
  list_pointer first, second;
  first = (list_pointer) malloc(sizeof(list_node));
  second = (list_pointer) malloc(sizeof(list_node));
  second->link = NULL;
  second->data = 20;
  first->data = 10;
  first->link = second;
  return first;
}
```

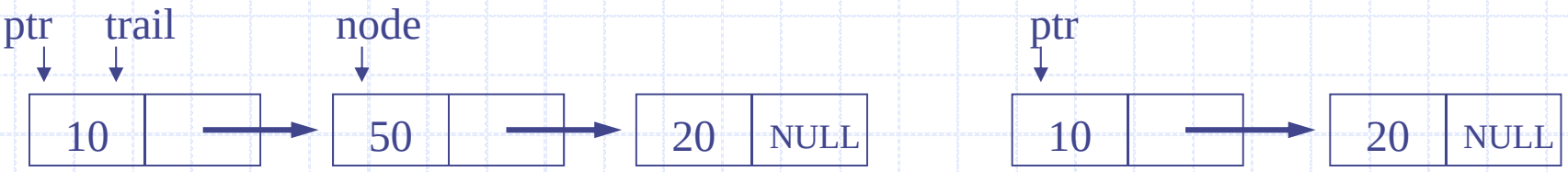


刪除節點(Deletion)from a list

```
void delete(list_pointer *ptr, list_pointer trail,
   list_pointer node)
{  /* ptr may change, pass in the address of ptr */
   /* trail is the preceding node, ptr is the head of the
   list */
   if (trail) trail->link = node->link;
   else *ptr = (*ptr)->link;
   free(node);
}
```



delete(&ptr, NULL, ptr)



delete(&ptr, ptr, ptr->link)

Linked List的應用

- 多項式(Polynomials)表示

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

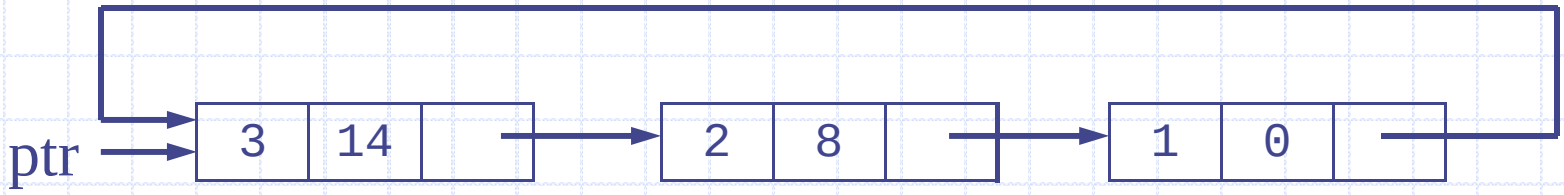
```
typedef struct poly_node
*poly_pointer;
typedef struct poly_node
{
        int coef;
        int expon;
        poly_pointer link;
        };
poly_pointer a, b, d;
```

coef	expon	link

環形linked lists

- If the link field of the last node points to the first node in the list, all the nodes of a polynomial can be freed **more efficiently**.
- Circular 表示方法：

$$ptr = 3x^{14} + 2x^{8} + 1$$



樹(Tree)

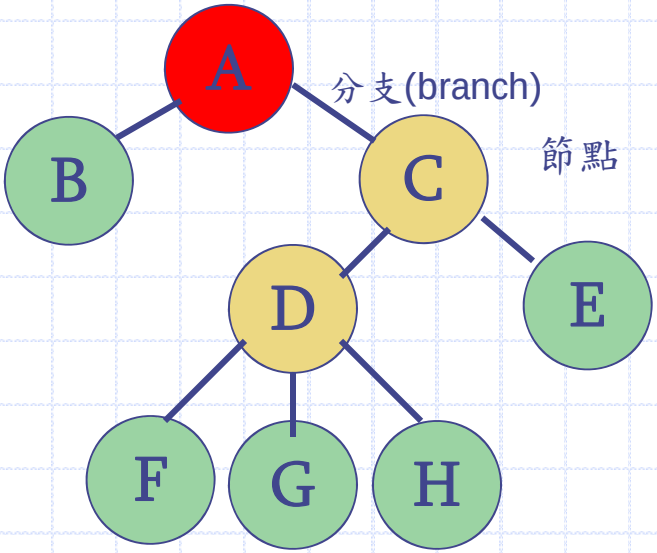
- Tree是資料結構中最重要且常用的單位。
- 將各種須分類的事物(如家譜或公司的組織圖)有階層的顯示出來的方式，正如一棵樹一樣，由根而枝而樹葉地展開。

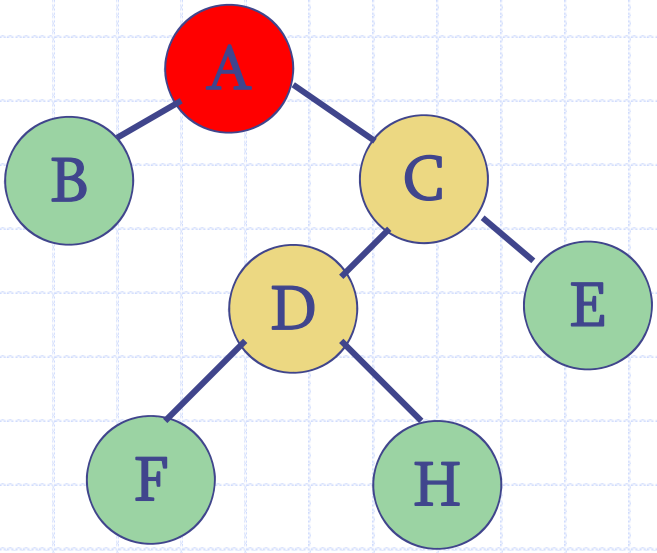
樹(Tree)

- 樹由數個節點（node)與將節點連接起來的分支（branch)所組成。
- 節點分支出來的節點稱為「子節點」，其上層的節點稱為「父節點」。
- 樹的最上面節點稱為「根」（root)。
- 底下沒有子節點的節點稱為樹葉（leaf)

樹(Tree)

- 樹的各節點分支如在2個以下（含2個），則稱為二元樹（binary tree)

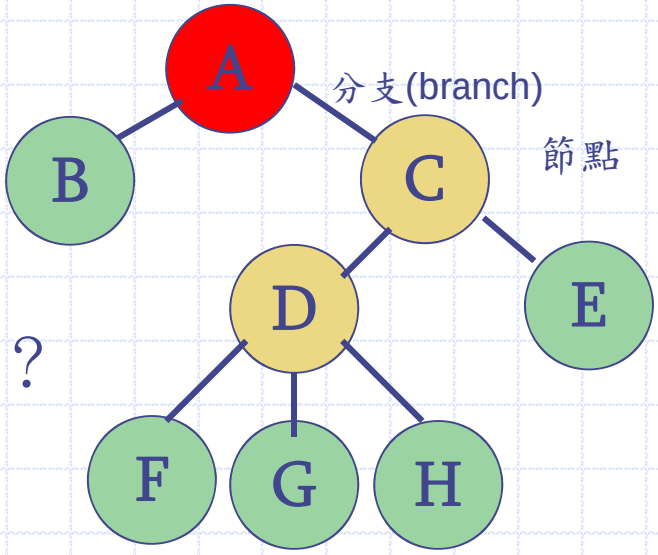




二元樹(binary tree)

樹(Tree)

- Depth（深度）：由根到某個節點間所通過的分支數，稱為該節點的深度。
- Height(高度）：由根到最深節點的深度，稱為樹的高度。



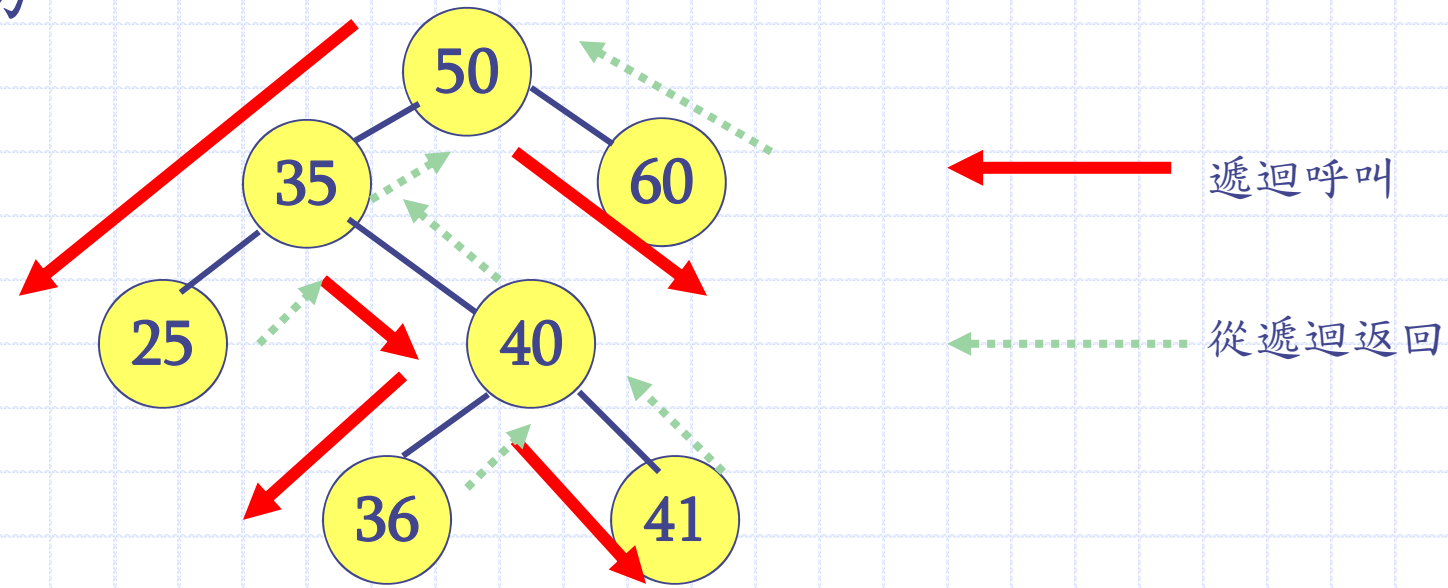
- 問：
  1. 節點B、G、E的深度分別為？
  2. 樹的高度為？
  3. 節點A(根）的深度為？

二元搜尋樹（binary search tree）

- 二元樹之中，各個節點的資料可以相互比較，父節點與子節點的關係按某種規則（大、小）排列的樹稱為**二元搜尋樹（binary search tree）**。

二元搜尋樹的追蹤（traversal)

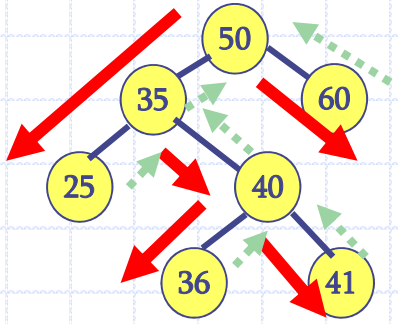
- 以一定的步驟巡視樹的所有節點，稱為樹的追蹤（traversal)。也有人稱為尋訪。



樹的追蹤

二元搜尋樹的追蹤（traversal）

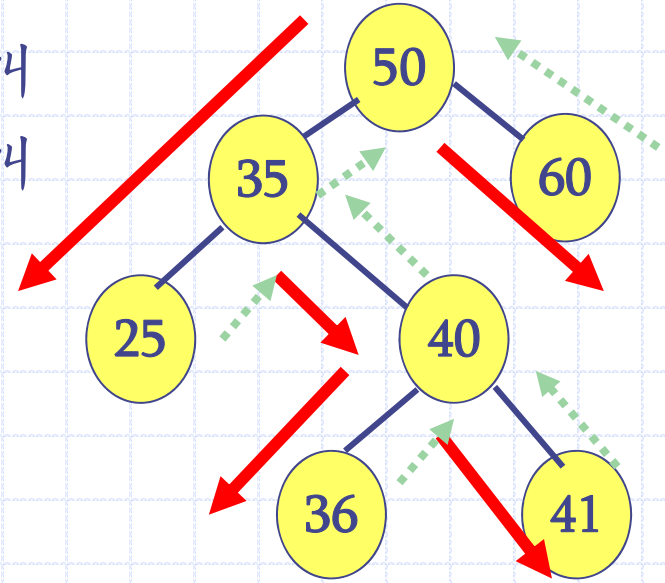
- 樹的追蹤過程中，「巡視過的節點顯示處理」方式分為：
  - 前序追蹤（preorder traversal）
  - 中序追蹤（inorder traversal）
  - 後序追蹤（postorder traversal）



樹的追蹤

前序追蹤（preorder traversal）

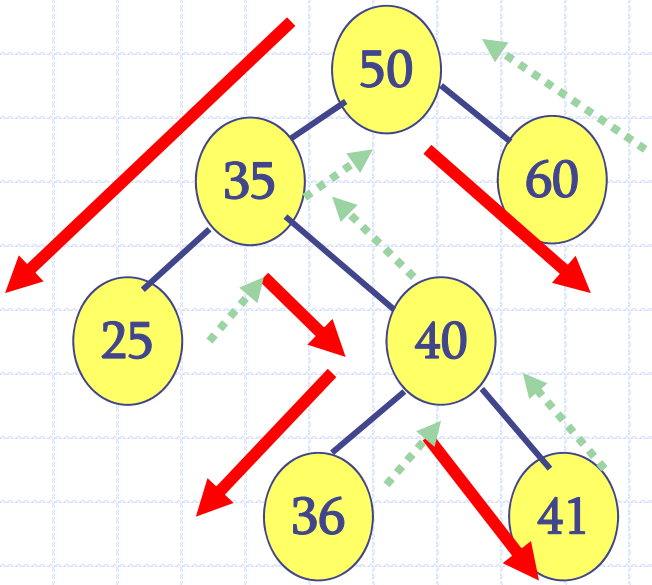
- 中左右
  1. 顯示節點
  2. 巡視左側樹的遞迴呼叫
  3. 巡視右側樹的遞迴呼叫
- 資料顯示順序：
  - 50 35 25 40 36 41 60



樹的追蹤

中序追蹤（inorder traversal）

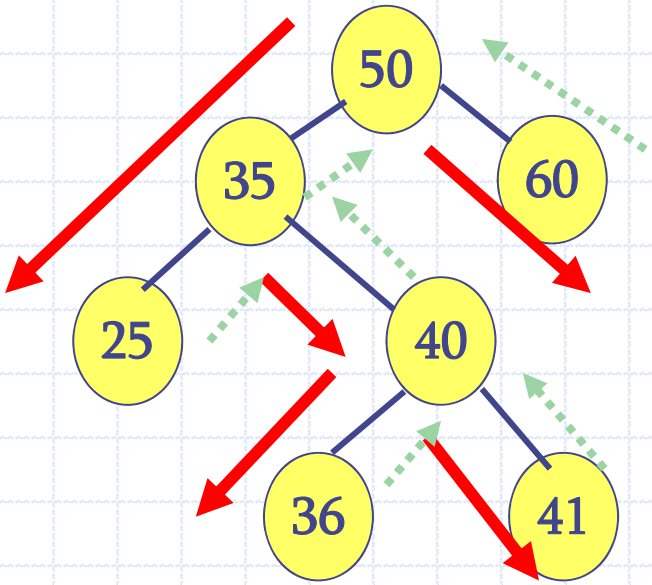
- 左中右
  1. 巡視左側樹的遞迴呼叫
  2. 顯示節點
  3. 巡視右側樹的遞迴呼叫
- 資料顯示順序：
  - 25 35 36 40 41 50 60



樹的追蹤

後序追蹤（postorder traversal）

- 左右中
  1. 巡視左側樹的遞迴呼叫
  2. 巡視右側樹的遞迴呼叫
  3. 顯示節點
- 資料顯示順序：
  - 25 36 41 40 35 60 50



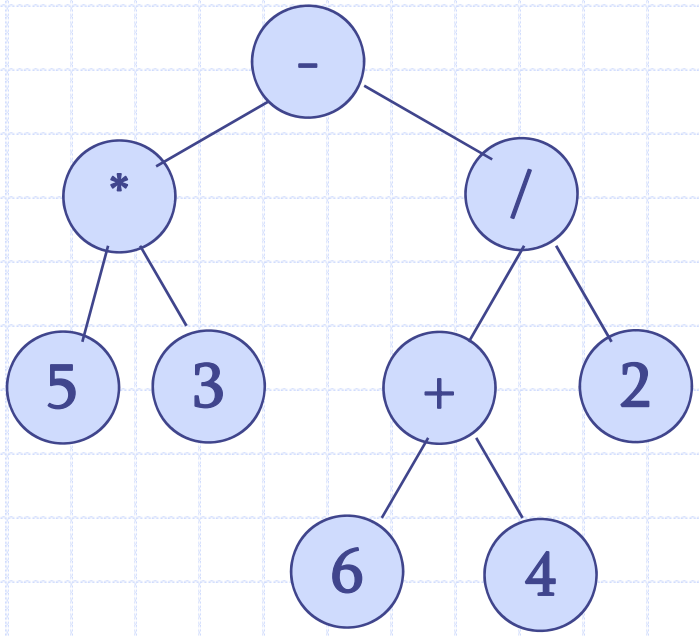
樹的追蹤

計算式樹

- 有一計算式樹，三種traversal方式分別如下：
  - 前序：-*ab+/cde
  - 中序：a*b-c+d/e
  - 後序：ab*cd+e/-
- 請畫出此樹。

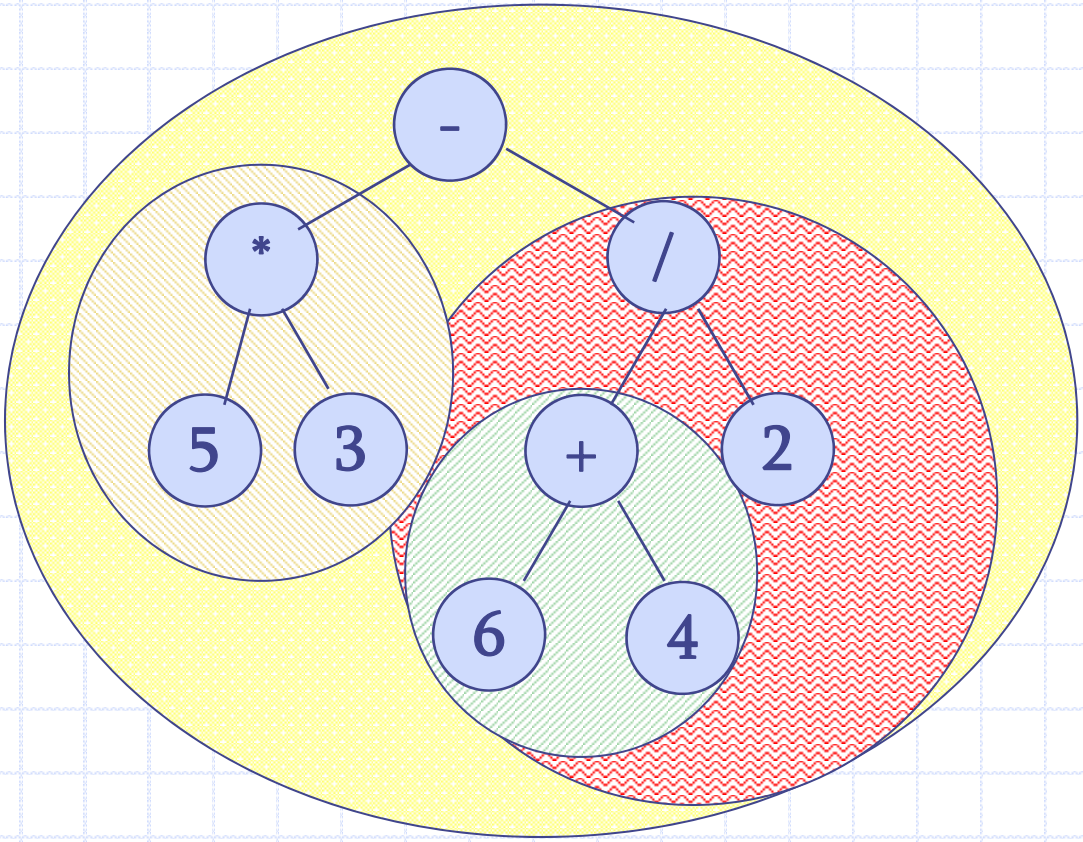
計算式樹的計算

- 以後序追蹤計算式樹，請寫下結果。



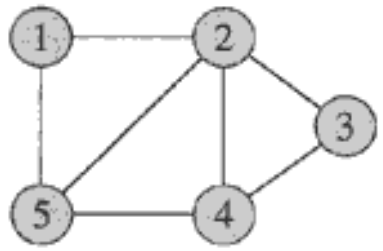
計算式樹的計算

- 以後序追蹤計算式樹，請寫下結果。

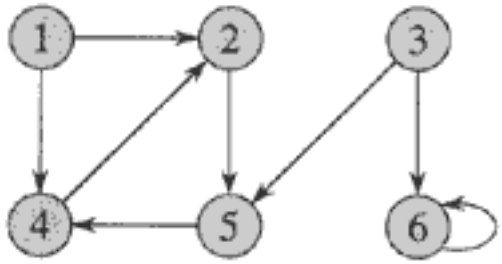


圖形（Graph)

- 圖形(Graph)是指以邊（Edge)將節點（node, Vertex)連接起來的物件。
- G=(V, E)
  - V = vertex set
  - E = edge set
- 圖形表示法：
  - Adjacency list
  - Adjacency matrix



Undirected Graph

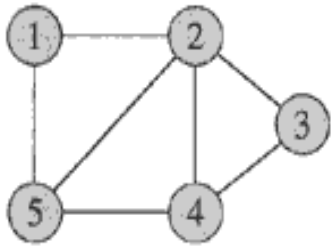


Directed Graph

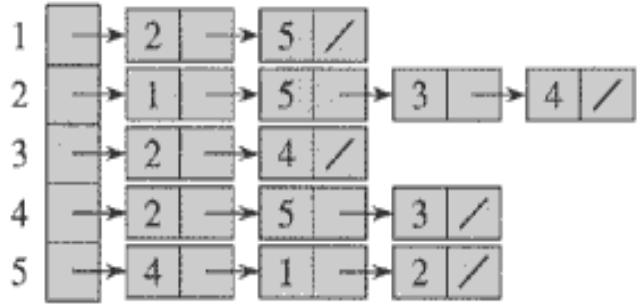
圖形的搜尋

- Breadth-first search (BFS)
- Depth-first search (DFS)
  - Topological sort
  - Strongly connected components

無向圖（undirected Graph）表示法



(a)



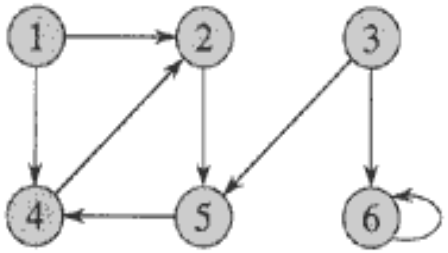
(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

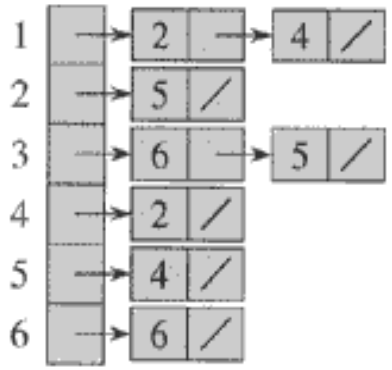
(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

有向圖（directed Graph）表示法



(a)



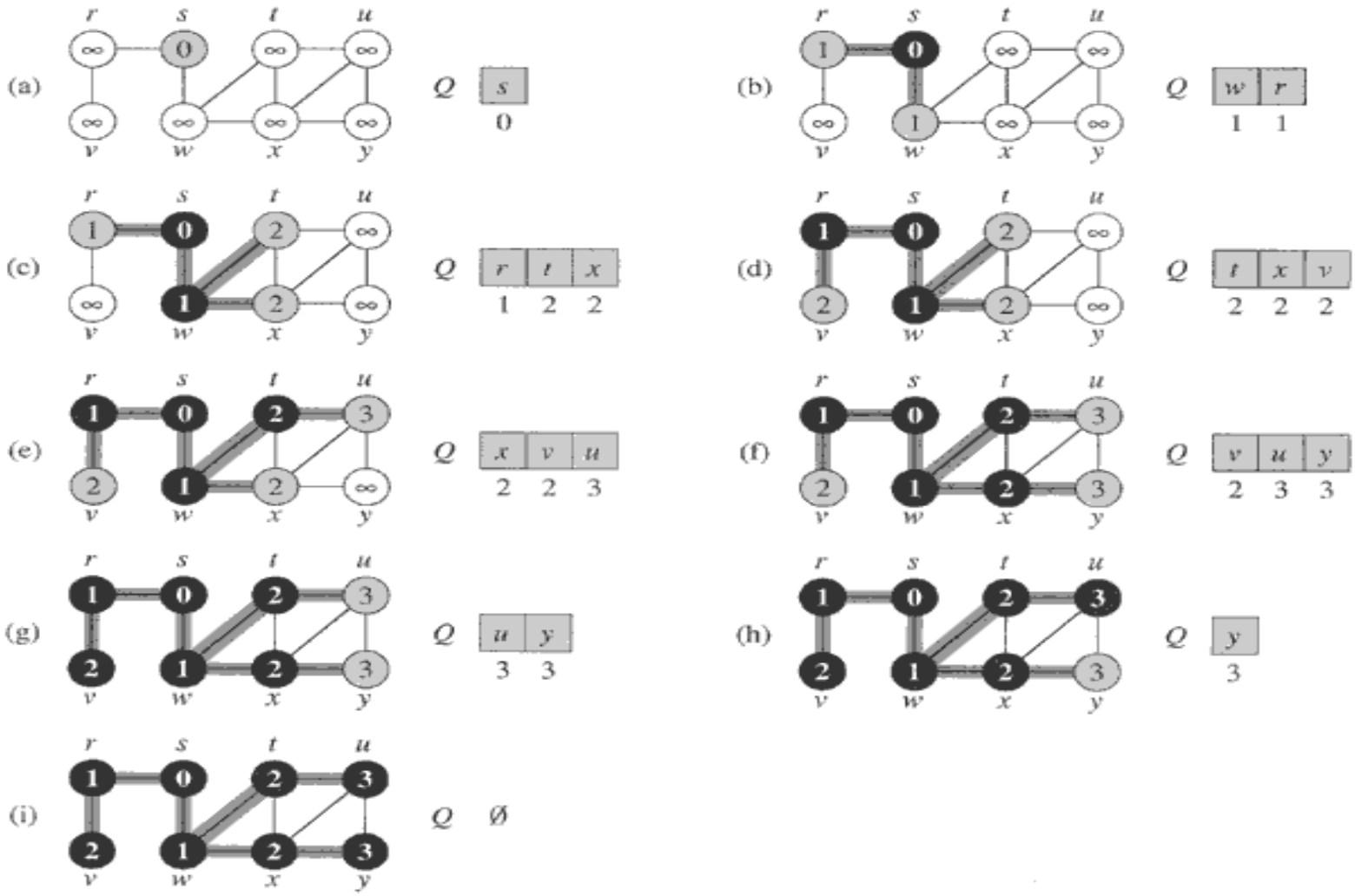
(b)

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

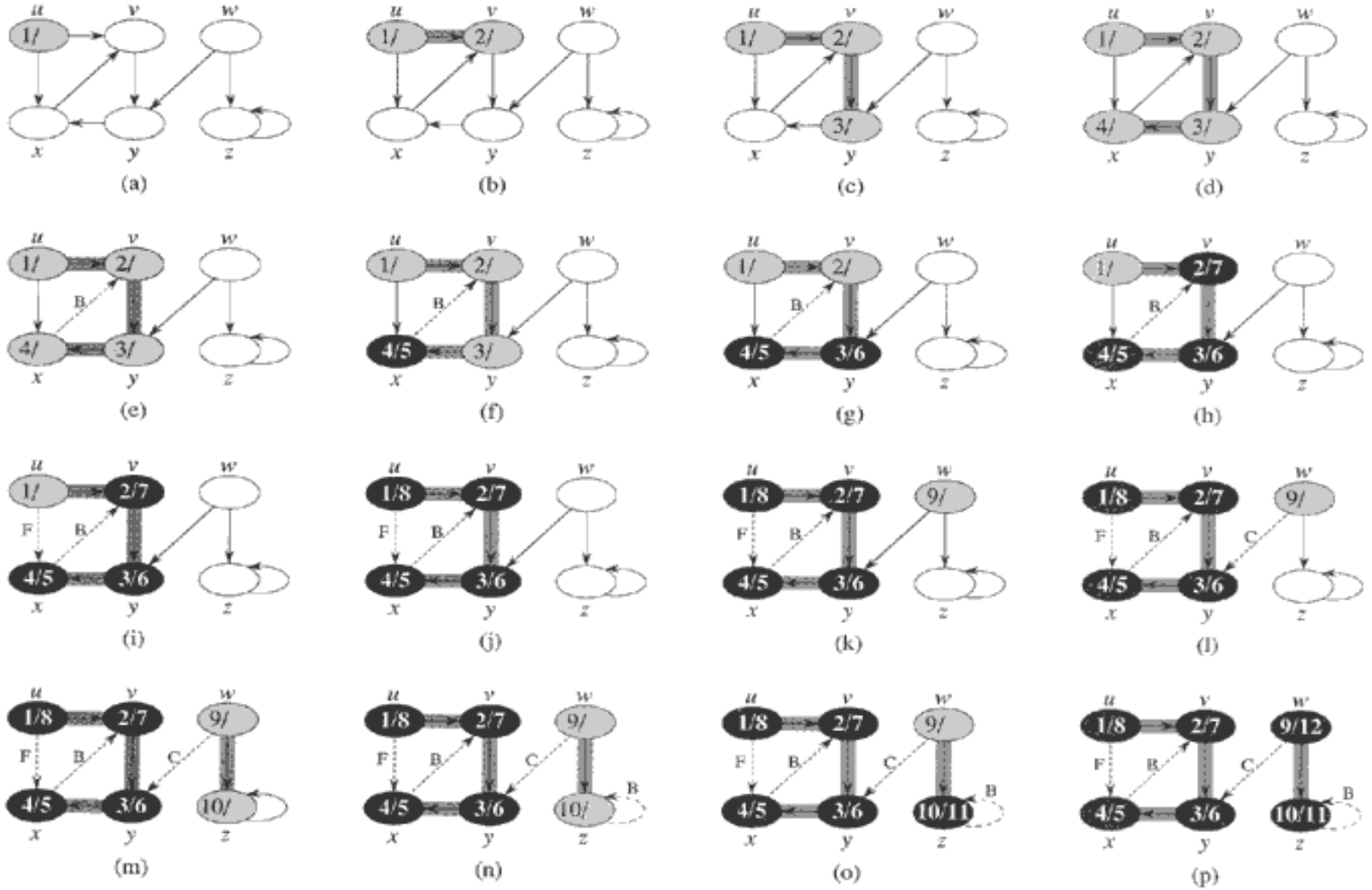
(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Breadth-First Search (BFS)

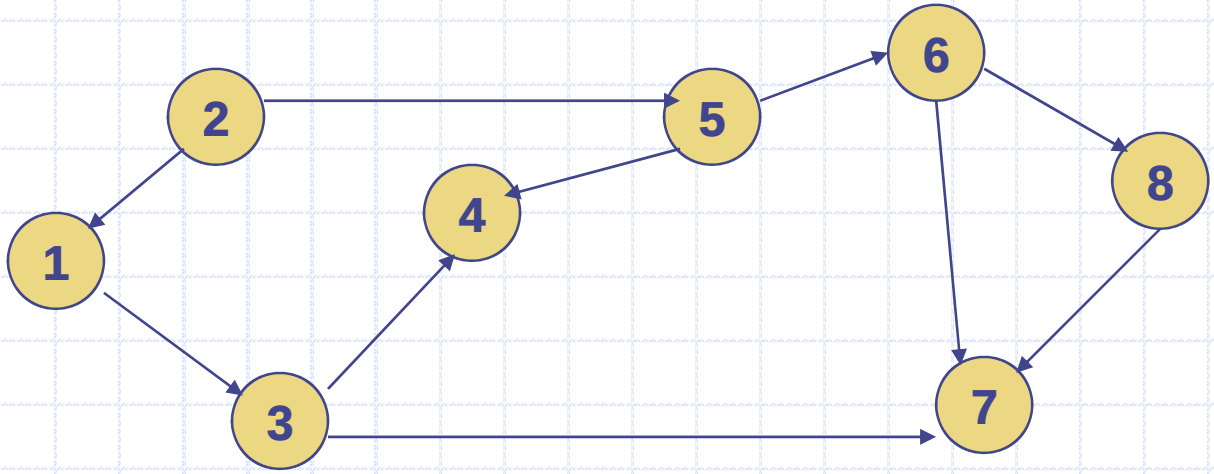


Depth-first search (DFS)



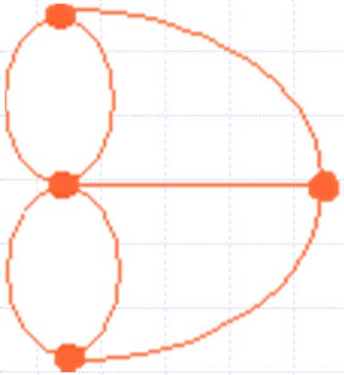
拓樸排序（Topological sort）

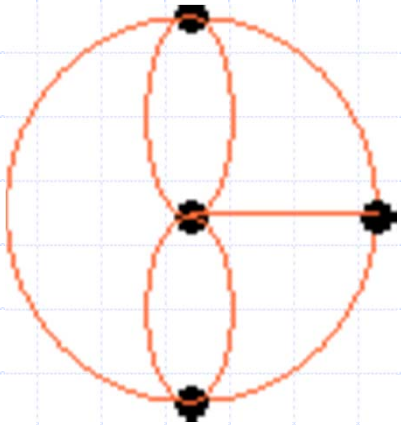
- 拓樸排序是指以某種規則將一有向圖形連接的節點排列成一列的情形。
- 方法不是唯一。

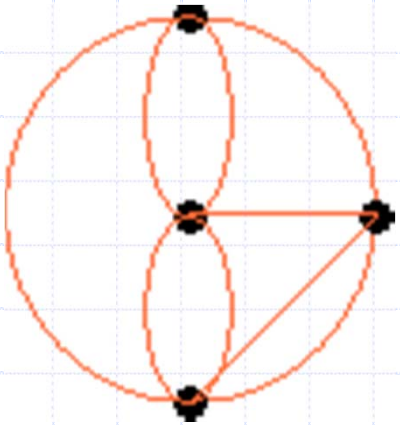


Euler的一筆畫

- 1736年，尤拉發表了他的「一筆劃定理」，大致如下：
  一個圖形要能一筆劃完成必須符合兩個狀況：
  1. 圖形是封閉連通的；
  2. 圖形中的奇點個數為0或2。







Spanning Tree(展開樹)

- A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree.
- A graph may have many spanning trees;
- for instance the complete graph on four vertices

Minimum spanning tree

- The weight of a tree is just the sum of weights of its edges.
- Lemma: Let X be any subset of the vertices of G, and let edge e be the smallest edge connecting X to G-X. Then e is part of the minimum spanning tree.

Kruskal's algorithm

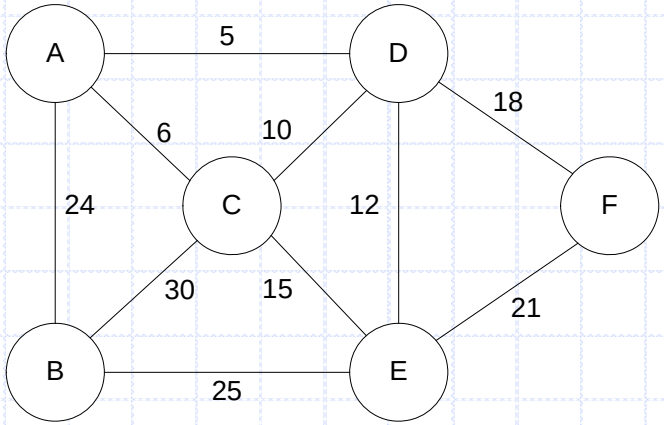
- 最易理解，也最易以手算的方法。

Kruskal's algorithm:

```
sort the edges of G in increasing order by length
keep a subgraph S of G, initially empty
for each edge e in sorted order
        if the endpoints of e are disconnected in S
        add e to S
return S
```

- 這是一個**Greedy method**（貪心演算法）

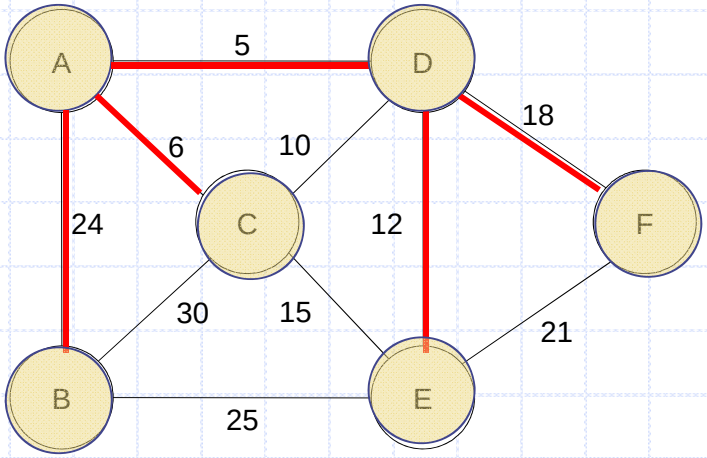
那個邊（edge）存在於下圖的最小成本生成樹（minimum-cost spanning trees）中？



（a）AB
（b）CD
（c）CE
（d）EF

Sort:

5,6,10,12,15,18,21, 24,25,30

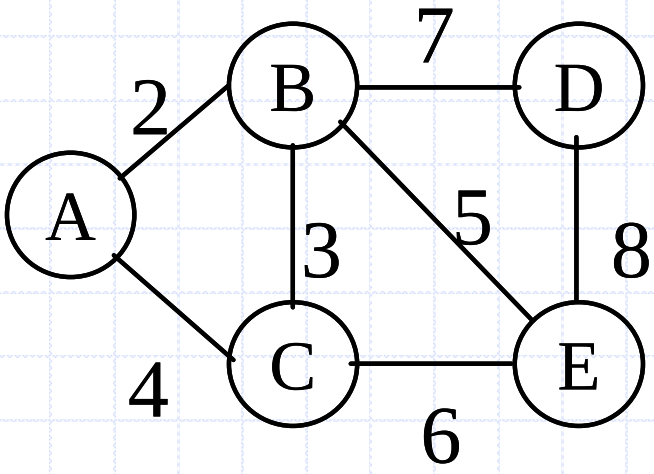


Minimum cost = 5+6+12+18+24 = 65

Minimum cost spanning tree

下圖中的最小成本擴張樹(Minimum cost spanning tree)的成本為

（a）17
（b）20
（c）22
（d）14

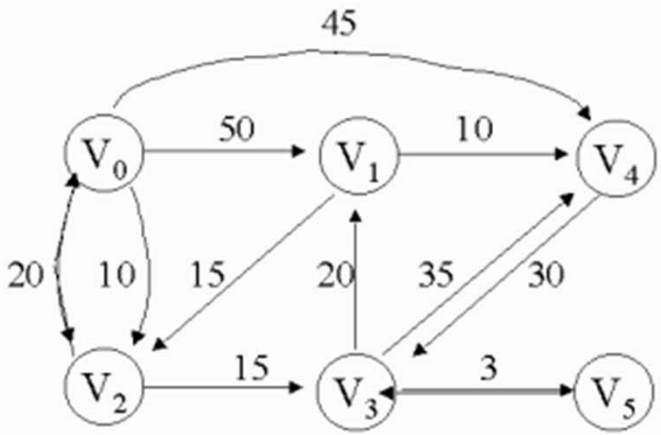


最短路徑(Shortest Path)

Single Source All Destinations

Determine the shortest paths from v0 to all the remaining vertices.

***Figure 6.29:** Graph and shortest paths from v_0 (p.293)



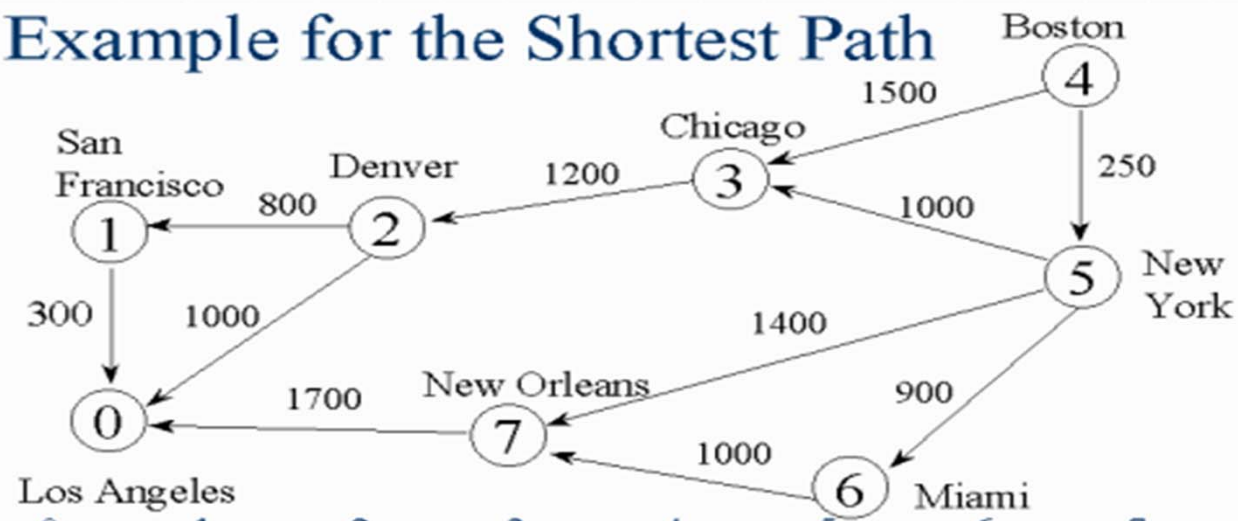
(a)

path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

(b)

最短路徑(Shortest Path)

Example for the Shortest Path



	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

Cost adjacency matrix

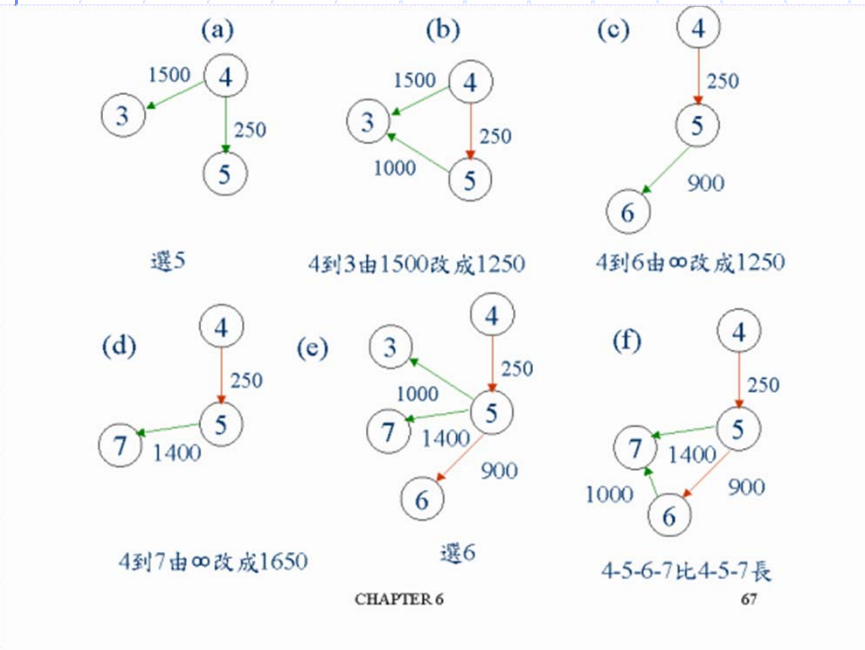
最短路徑(Shortest Path)

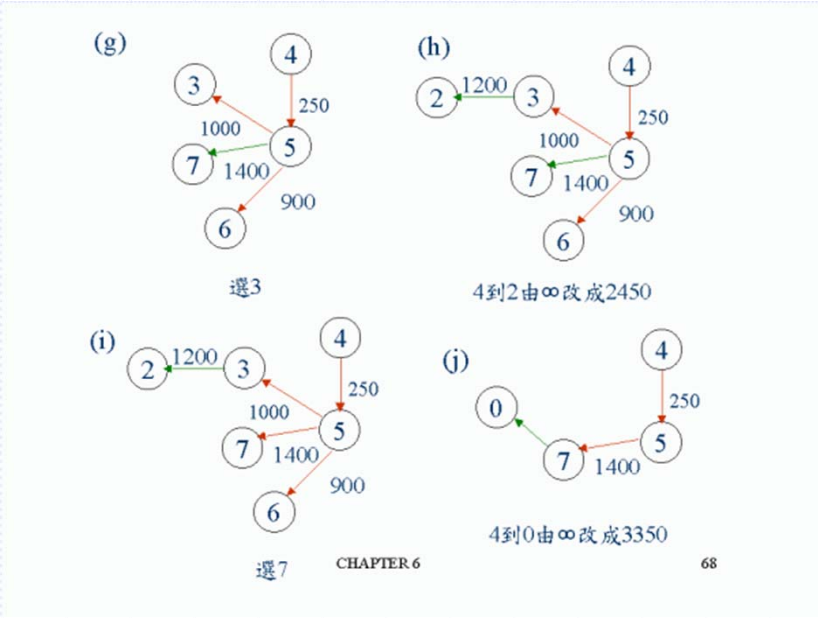
Single Source All Destinations

```
void shortestpath(int v, int
  cost[][MAX_ERXTICES], int distance[], int n,
  short int found[])
{
  int i, u, w;
  for (i=0; i<n; i++) {
    found[i] = FALSE;                O(n)
    distance[i] = cost[v][i];
  }
  found[v] = TRUE;
  distance[v] = 0;

 for (i=0; i<n-2; i++) {determine n-1 paths from v
   u = choose(distance, n, found);
   found[u] = TRUE;
   for (w=0; w<n; w++)
     if (!found[w])  與u相連的端點w
       if (distance[u]+cost[u][w]<distance[w])
         distance[w] = distance[u]+cost[u][w];
 }
}
```

最短路徑(Shortest Path)





排序(sort)

- 直接選擇法（基本選擇法）$O(n^2)$
- 泡泡排序法（基本交換法）$O(n^2)$
- 快速排序法（改良交換法）$O(n\log_2 n)$
- Heap 排序（改良選擇法）$O(n\log_2 n)$
- 合併排序(Merge sort) $O(n\log_2 n)$
- Shell 排序（改良插入法）$O(n^{1.2})$

Merge Sort

- **Definition:** A *sort* algorithm that splits the items to be sorted into two groups, *recursively* sorts each group, and *merges* them into a final, sorted sequence.
- Run time is Θ**(n log n).**

搜尋(Search)

- 循序搜尋法　$O(n)$
- 二元搜尋法　$O(\log_2 n)$

***Program 7.1:**Sequential search (p.321)

```
int seqsearch( int list[ ], int searchnum, int n )
{
/*search an array, list, that has n numbers. Return i, if
list[i]=searchnum. Return -1, if searchnum is not in the list */
    int i;
    list[n]=searchnum;    sentinel
    for (i=0; list[i] != searchnum; i++)
      ;
    return (( i<n) ? i : -1);
}
```

*Program 7.2: Binary search (p.322)

```
int binsearch(element list[ ], int searchnum, int n)
{
/* search list [0], ..., list[n-1]*/
   int left = 0, right = n-1, middle;
   while (left <= right) {
       middle = (left+ right)/2;
   switch (COMPARE(list[middle].key, searchnum)) {
       case -1: left = middle +1;
                break;
       case  0: return middle;
       case  1:right = middle - 1;
       }
     }
     return -1;
}
```

$O(\log_2 n)$