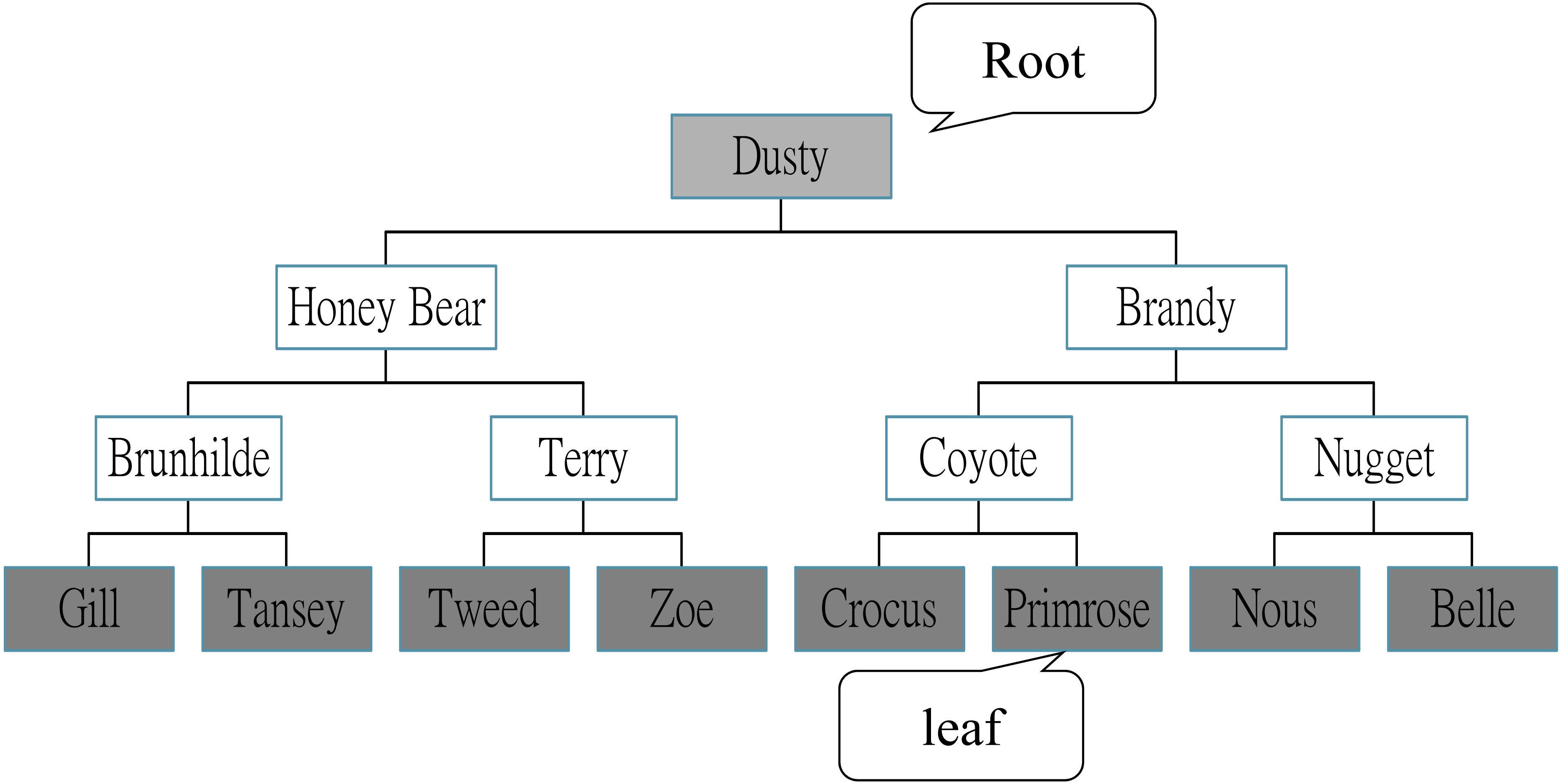
CHAPTER 5

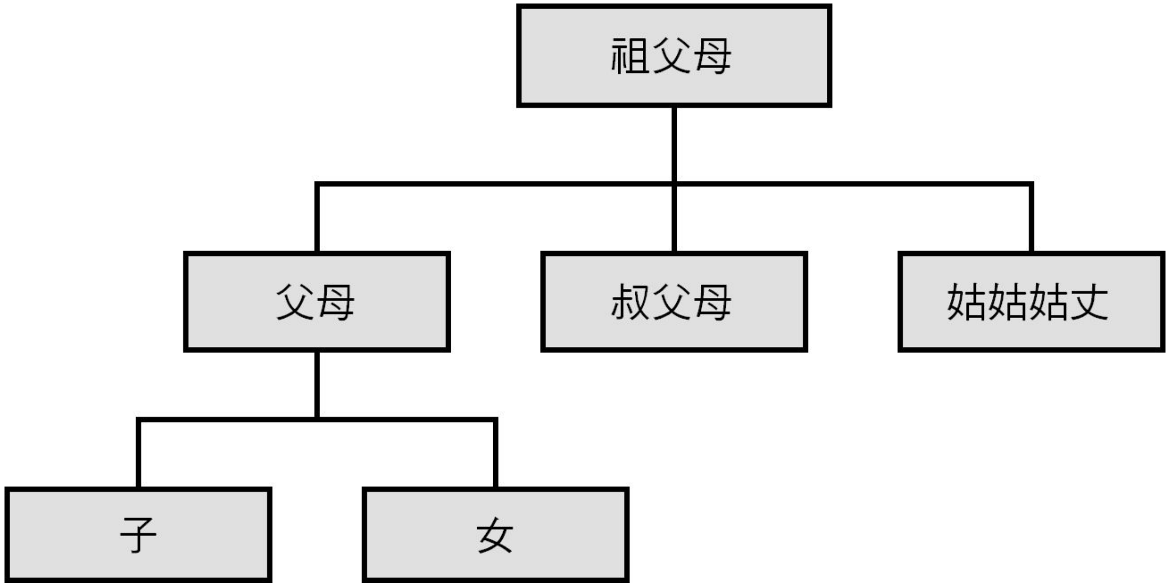
Trees

Trees


Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

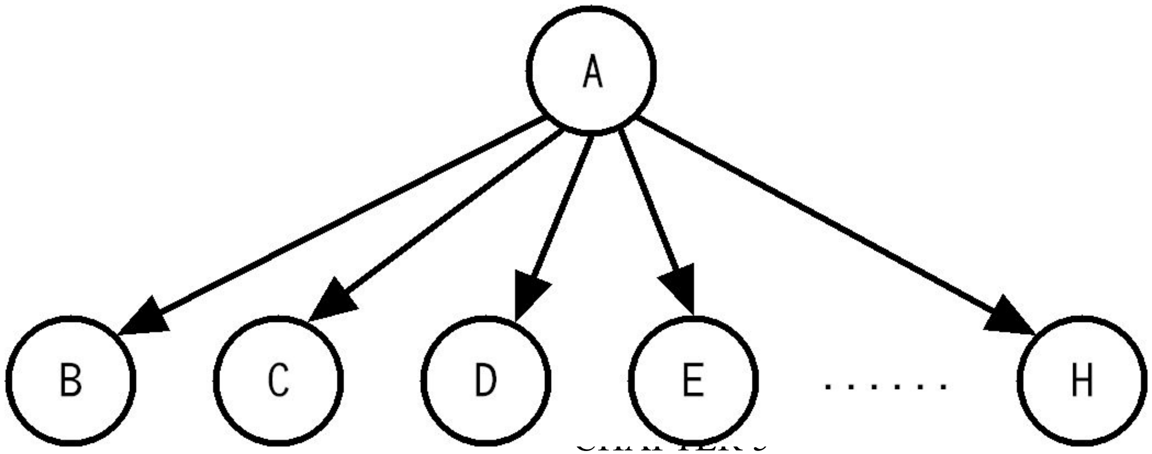
樹的基本觀念-說明

- 「樹」（Trees）是一種模擬現實生活中樹幹和樹枝的資料結構，屬於一種階層架構的非線性資料結構，例如：家族族譜，如下圖所示：



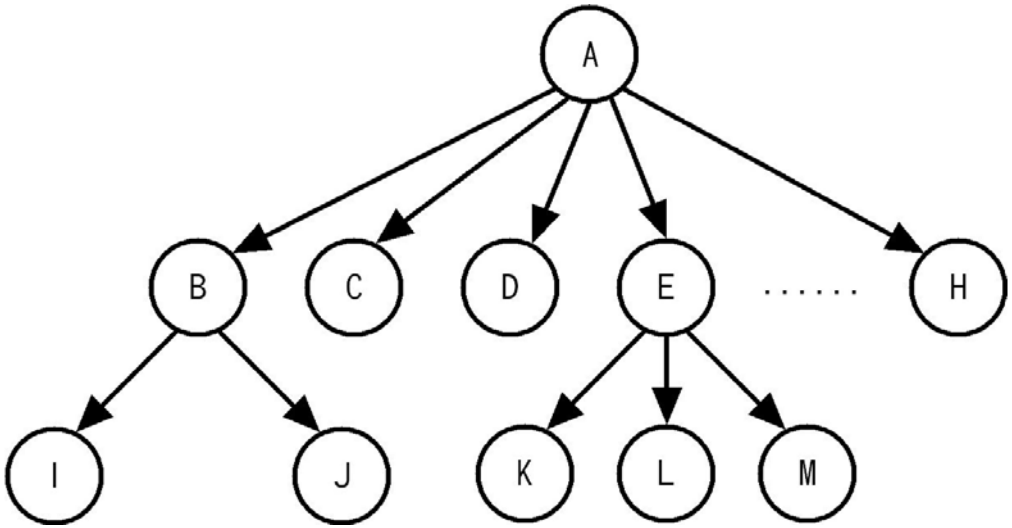
樹的基本觀念-架構1

- 樹的樹根稱為「根節點」（Root），在根節點之下是樹的樹枝，擁有0到n個「子節點」（Children），即樹的「分支」（Branch），節點A是樹的根節點，B、C、D….和H是節點A的子節點，即樹枝，如下圖所示：



樹的基本觀念-架構2

- 在樹枝下還可以擁有下一層樹枝，I和J是B的子節點，K、L和M是E的子節點，節點B是I和J的「父節點」（Parent），節點E是K、L和M的父節點，節點I和J擁有共同父節點，稱為「兄弟節點」（Siblings），K、L和M 是兄弟節點，B、C…和H節點也是兄弟節點，如下圖所示：



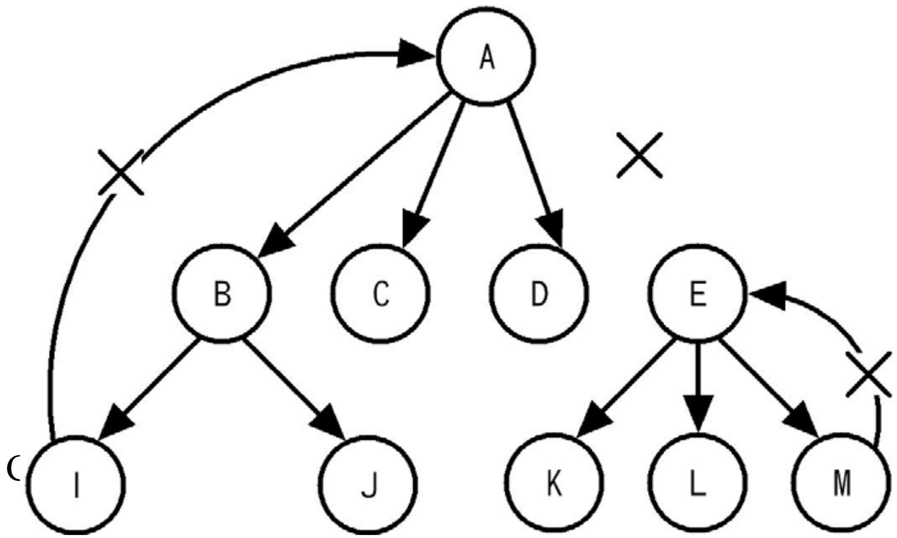
樹的基本觀念-定義

定義 7.1：樹的節點個數是一或多個有限集合，且：

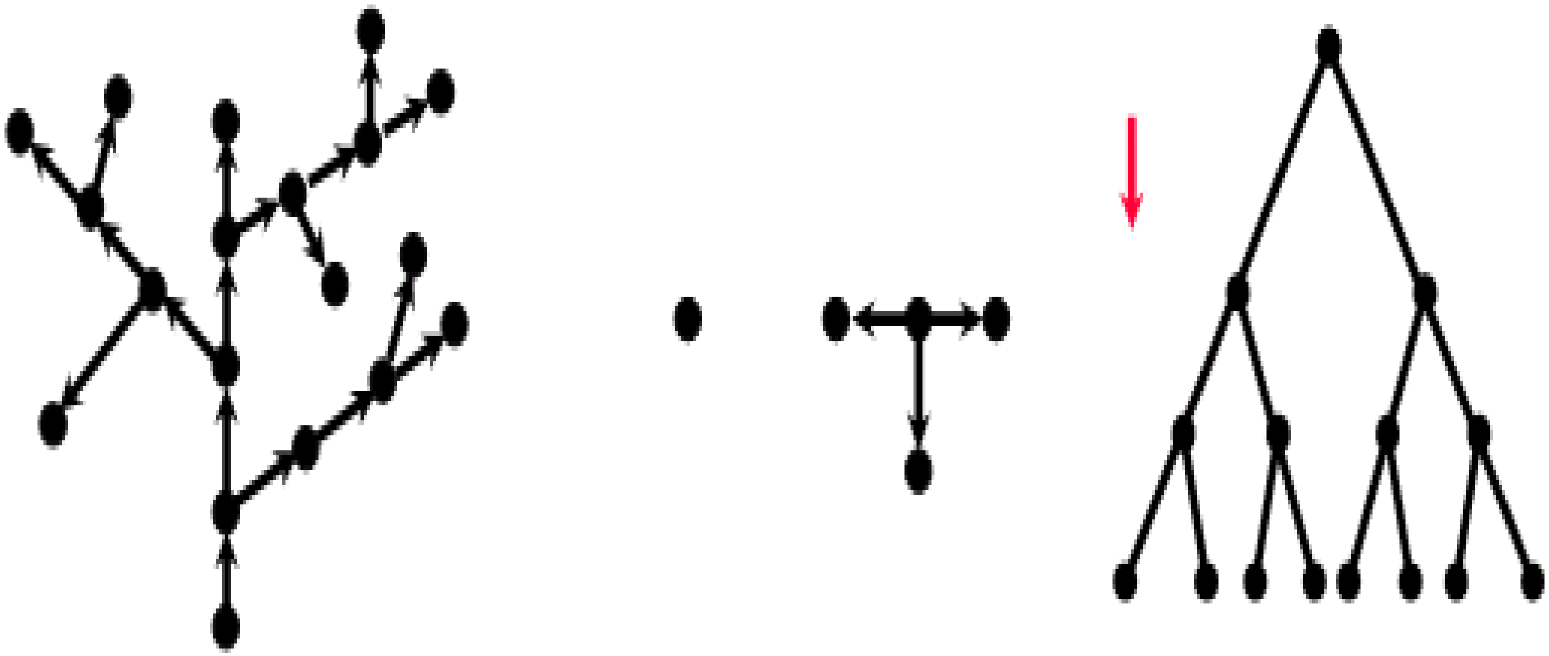
(1) 存在一個節點稱為根節點。

(2) 在根節點下的節點分成n >= 0 個沒有交集的多個子集合 t1、t2…, tn，每一個子集合也是一棵樹，而這些樹稱為根節點的「子樹」（Subtree）。

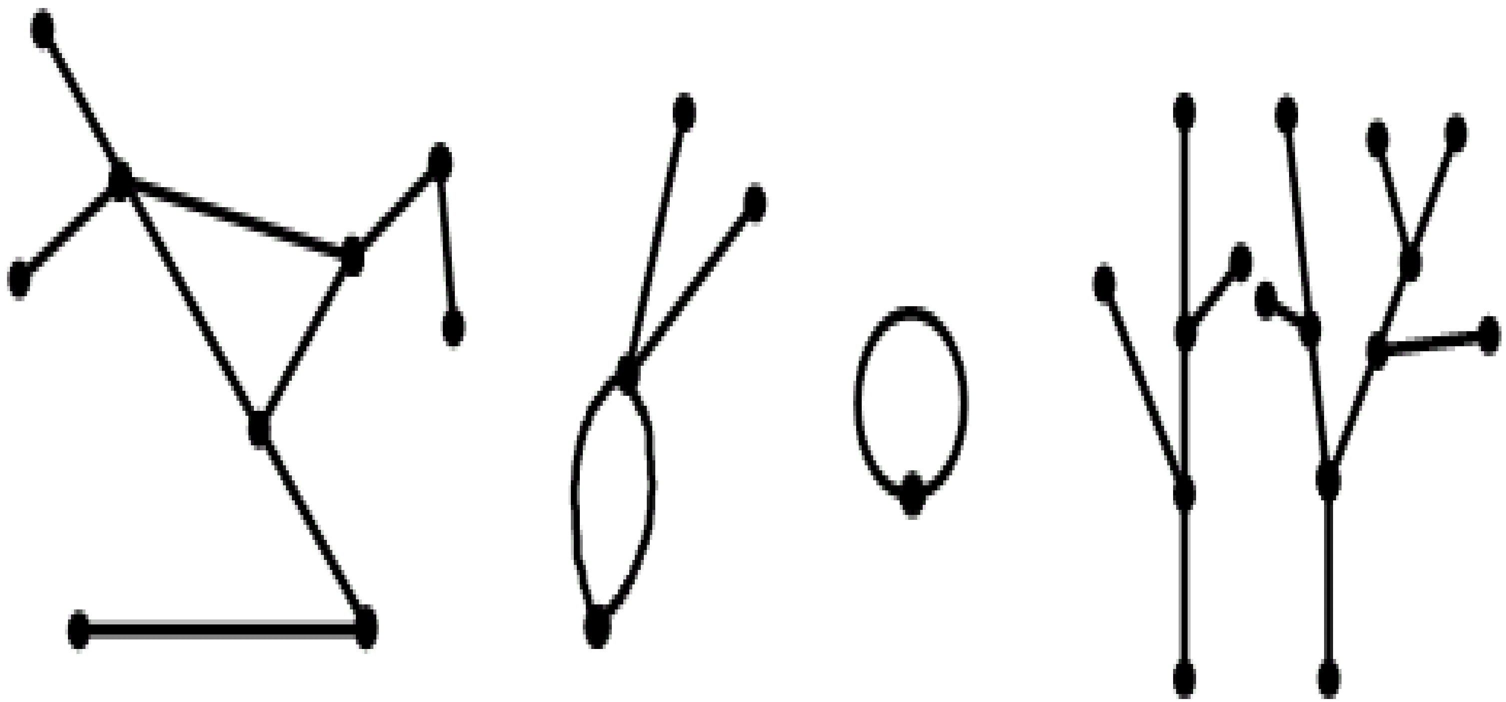
- 樹在各節點之間不可以有迴圈，或不連結的左、右子樹，如下圖所示：



樹的範例



非樹的範例



樹的基本觀念-相關術語1

- n元樹：樹的一個節點最多擁有n個子節點。
- 二元樹（Binary Trees）：樹的節點最多只有兩個子節點。
- 根節點（Root）：沒有父節點的節點是根節點。例如：節點A。
- 葉節點（Leaf）：節點沒有子節點的節點稱為葉節點。例如：節點I、J、C、D、K、L、M、F、G和H。
- 祖先節點（Ancenstors）：指某節點到根節點之間所經過的所有節點，都是此節點的祖先節點。

樹的基本觀念-相關術語2

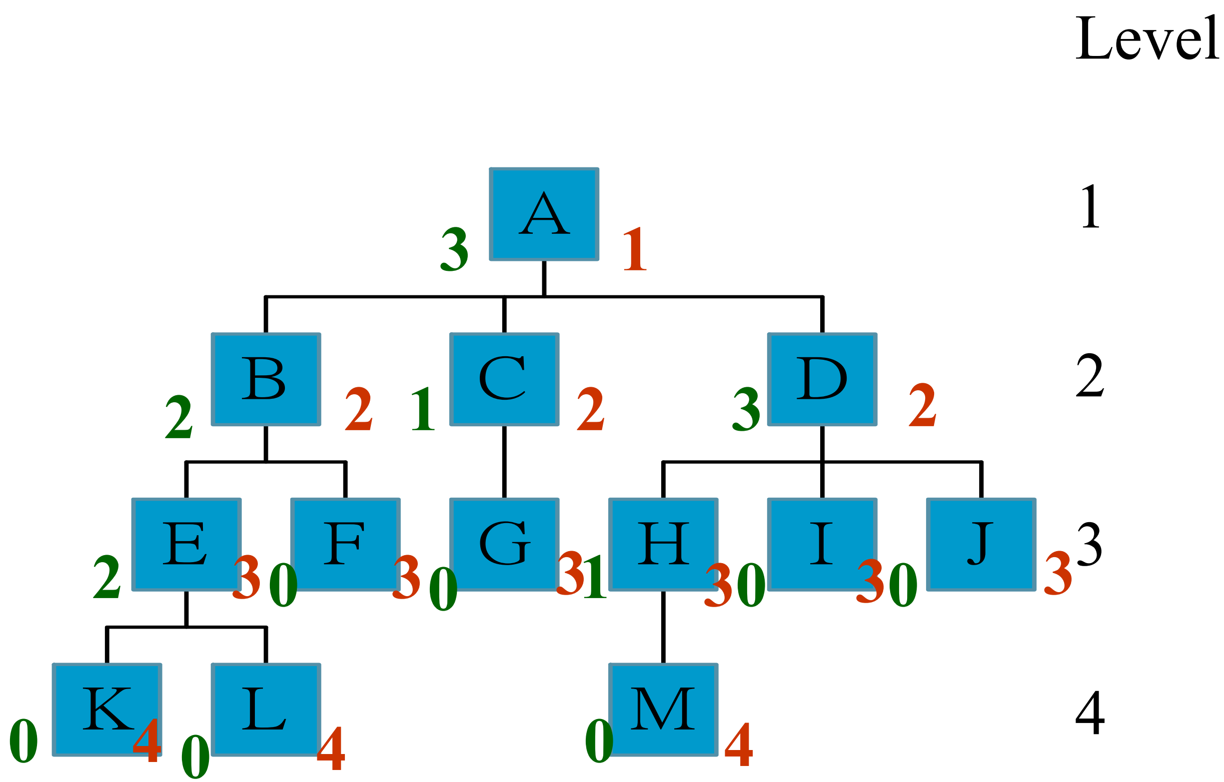
- 非終端節點（Non-terminal Nodes）：除了葉節點之外的其它節點稱為非終端節點。例如：節點A、B和E是非終端節點。
- 分支度（Dregree）：指每個節點擁有的子節點數。例如：節點B的分支度是2，節點E的分支度是3。
- 階層（Level）：如果樹根是1，其子節點是2，依序可以計算出樹的階層數。例如：上述圖例的節點A階層是1，B、C到H是階層2，I、J到M是階層3。
- 樹高（Height）：樹高又稱為樹深（Depth），指樹的最大階層數。例如：上述圖例的樹高是3。

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets T_1, ..., T_n, where each of these sets is a tree.
- We call T_1, ..., T_n the subtrees of the root.

Level and Depth

node (13)
degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
level of a node
height of a tree (4)

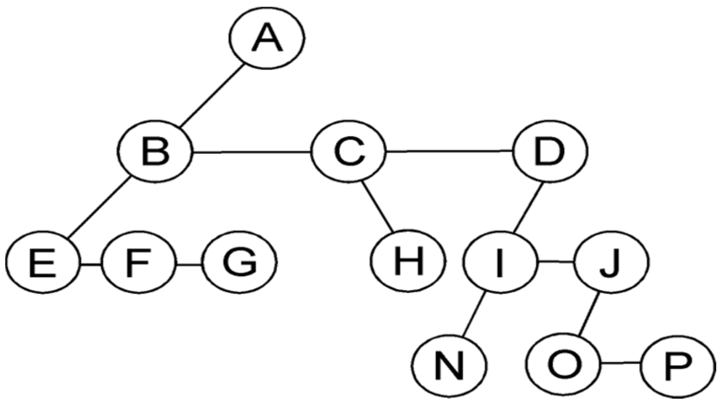


Terminology

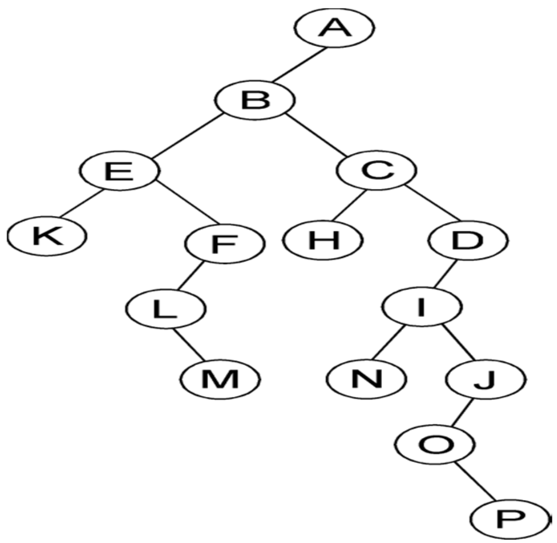
- The degree of a node is the number of subtrees of the node
  – The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

5.1.2 樹的表示法

- 一般化的串列表示
- 左子右兄弟表示法(圖一)
- 分支度為2的樹示法(圖二)

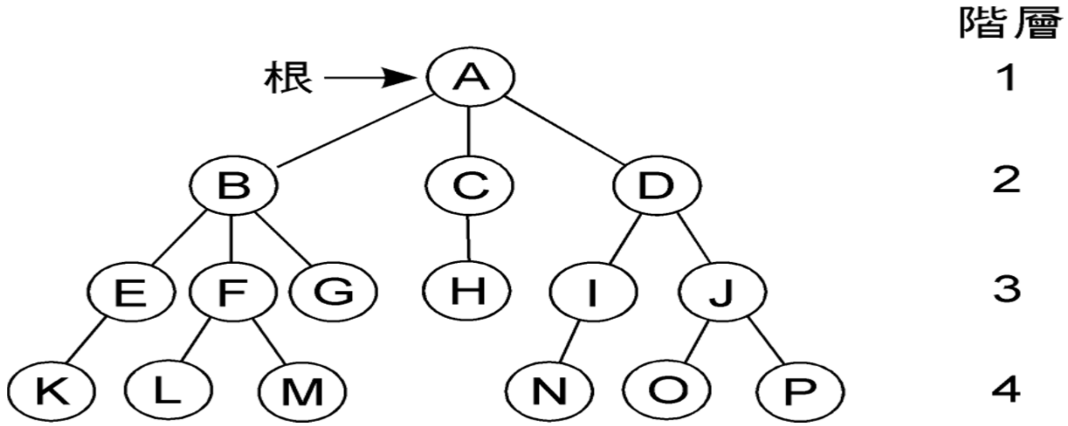


圖一



圖二

一般化的串列表示法



上圖的樹可表示成下面的一般化串列：

(A, (B, (E, K), (F, L, M), G), (C, H), (D, (I, N), (J, O, P)))

若將節點A的三兒子B、C、D所形成的3個子樹，分別取名為T1、T2、T3，則此樹可簡化成

(A, T1, T2, T3)

其中

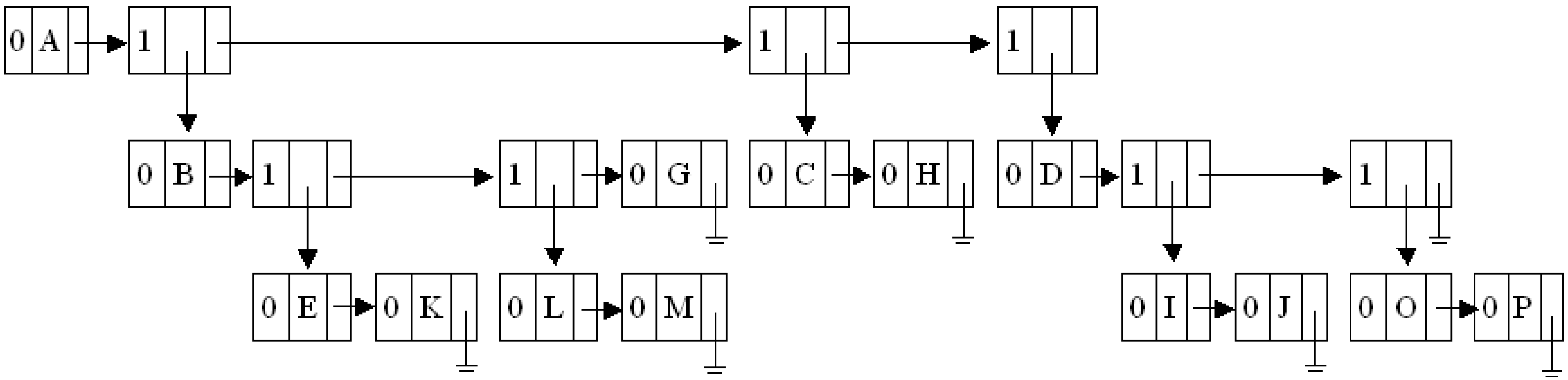
T1 = (B, (E, K), (F, L, M), G)

T2 = (C, H)

T3 = (D, (I, N), (J, O, P))

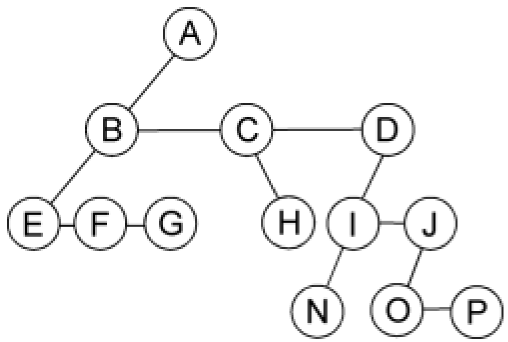
程式　一般化串列鏈結節點的宣告

```
struct TreeNode
{   int tag;
    union
    {    int data;
         struct TreeNode *Tlink;
    } node;
    struct TreeNode *link;
};
```

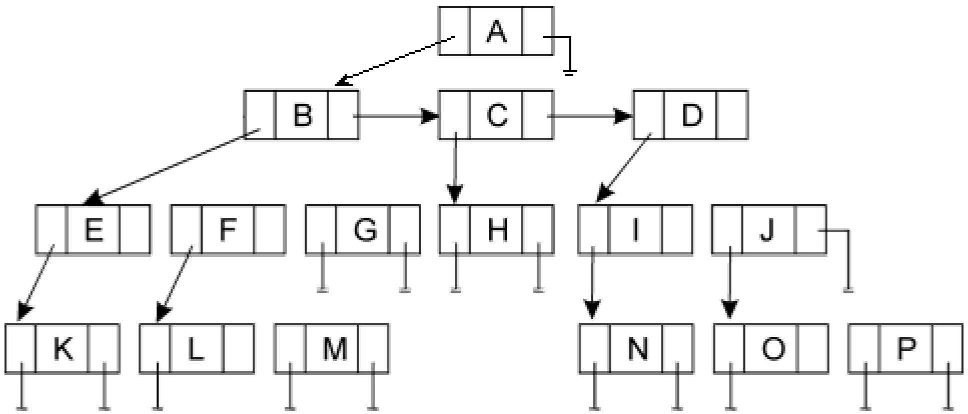


左子右兄弟表示法

- 每個節點都有唯一的最左兒子 **(leftmost child)** ；
- 每個節點都有最靠近它的右兄弟。



(a) 樹的左子右兄弟表示



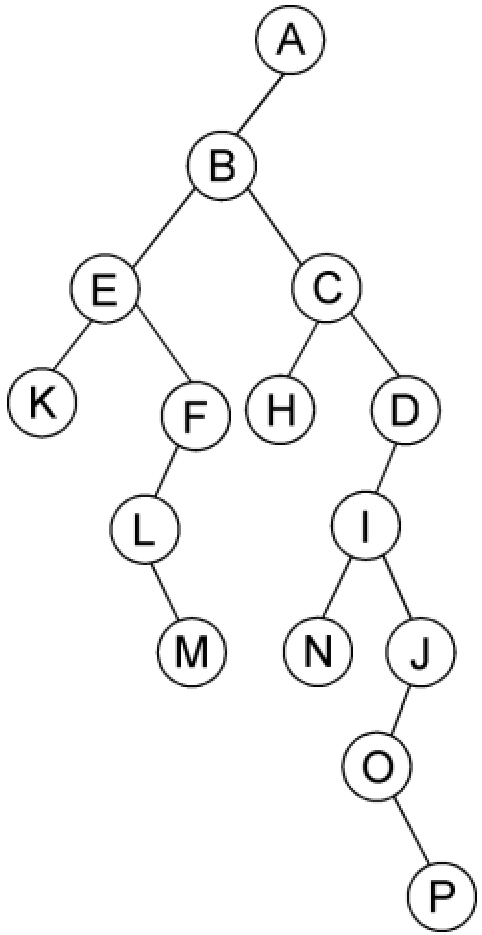
(b) 樹的左子右兄弟鏈結串列

分支度為2的樹表示法

- 分支度為2的樹又稱為二元樹 (**binary tree**)。
- 二元樹中任一節點皆有2個指標分別指向該節點的左子樹和右子樹。

Leftchild	data	rightchild

二元樹的節點構造



分支度為2的樹表示法

Representation of Trees

- List Representation
  – (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  – The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n

How many link fields are needed in such a representation?

Left Child - Right Sibling

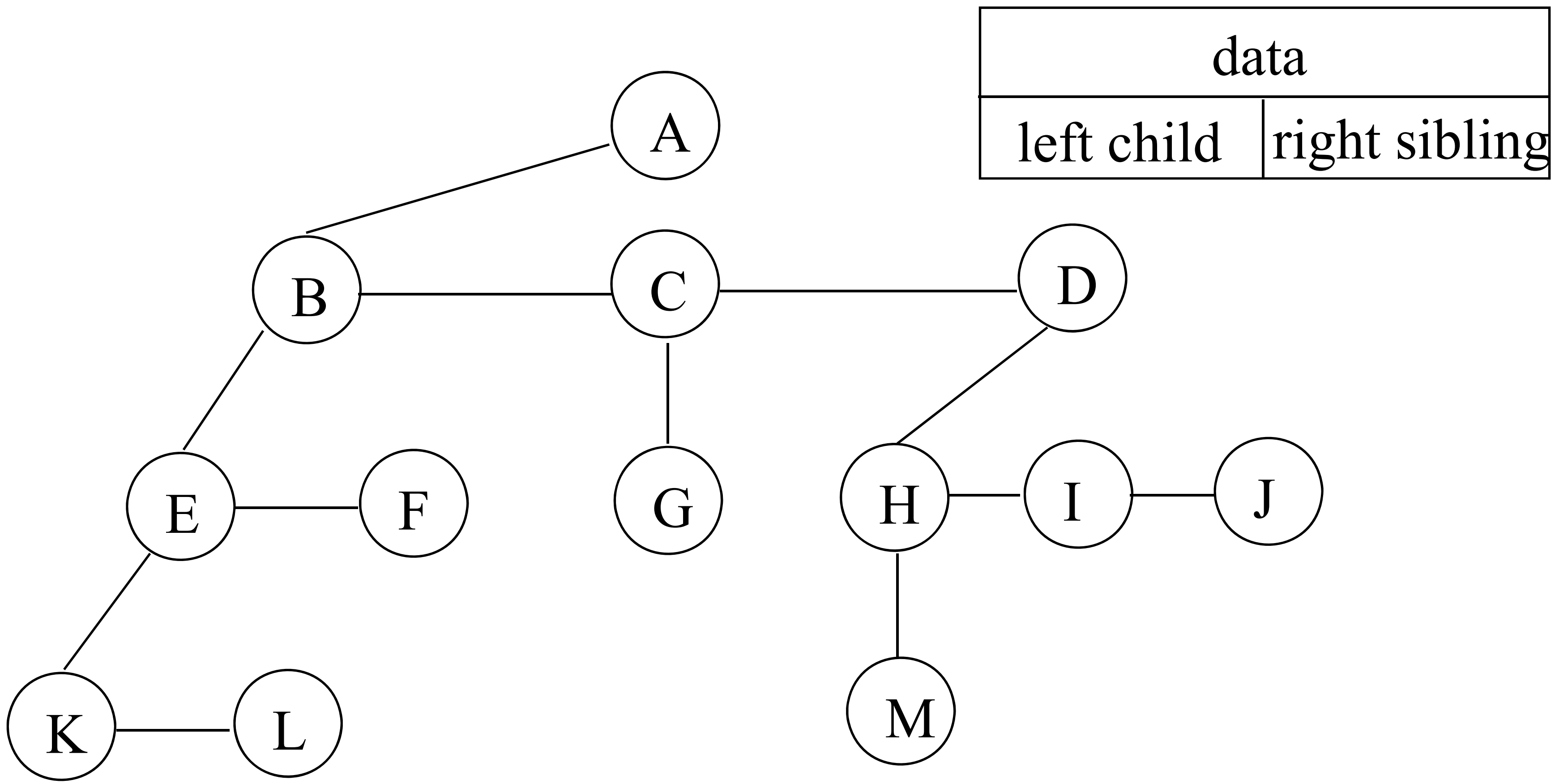
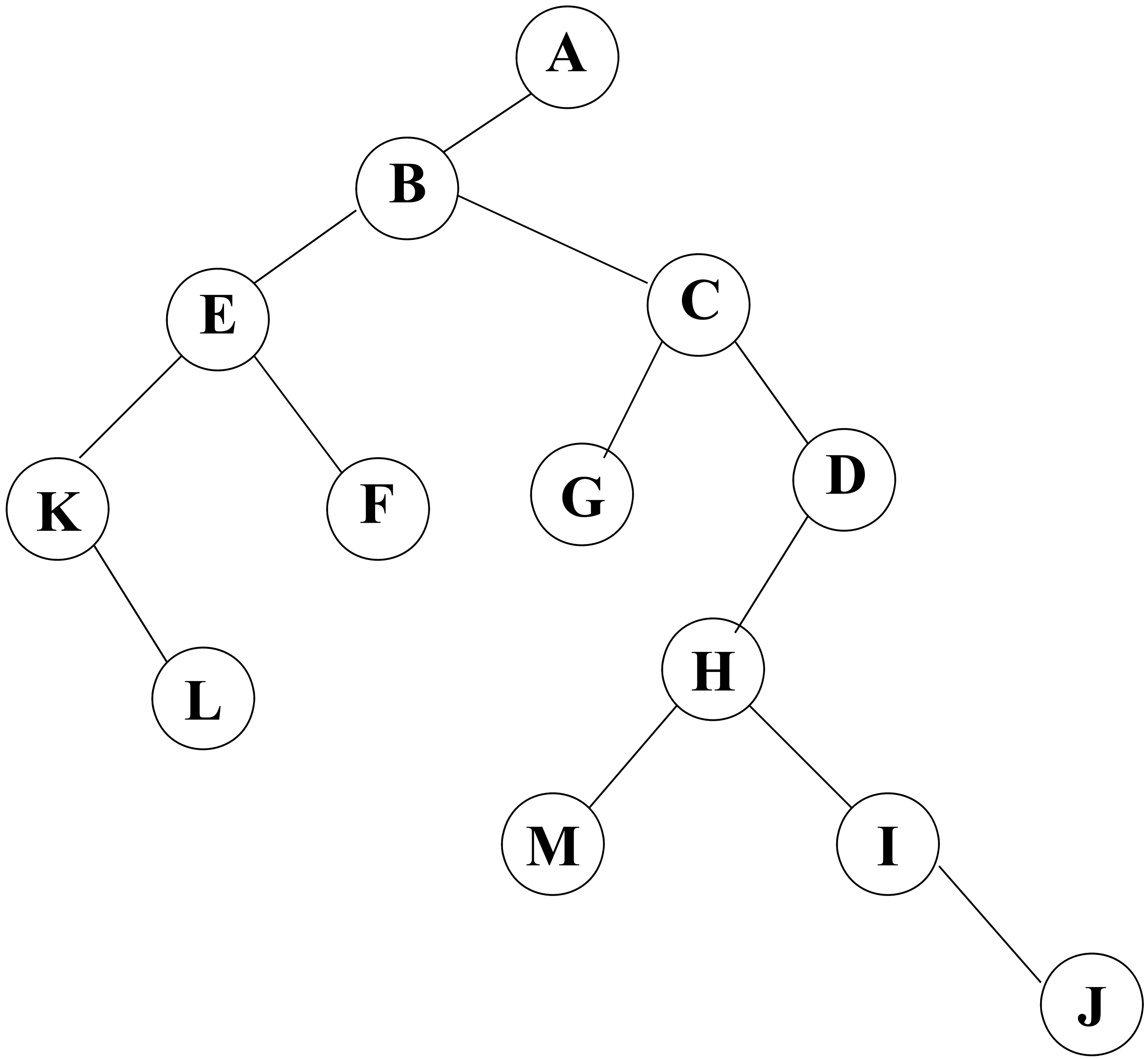


***Figure 5.7:** Left child-right child tree representation of a tree (p.197)

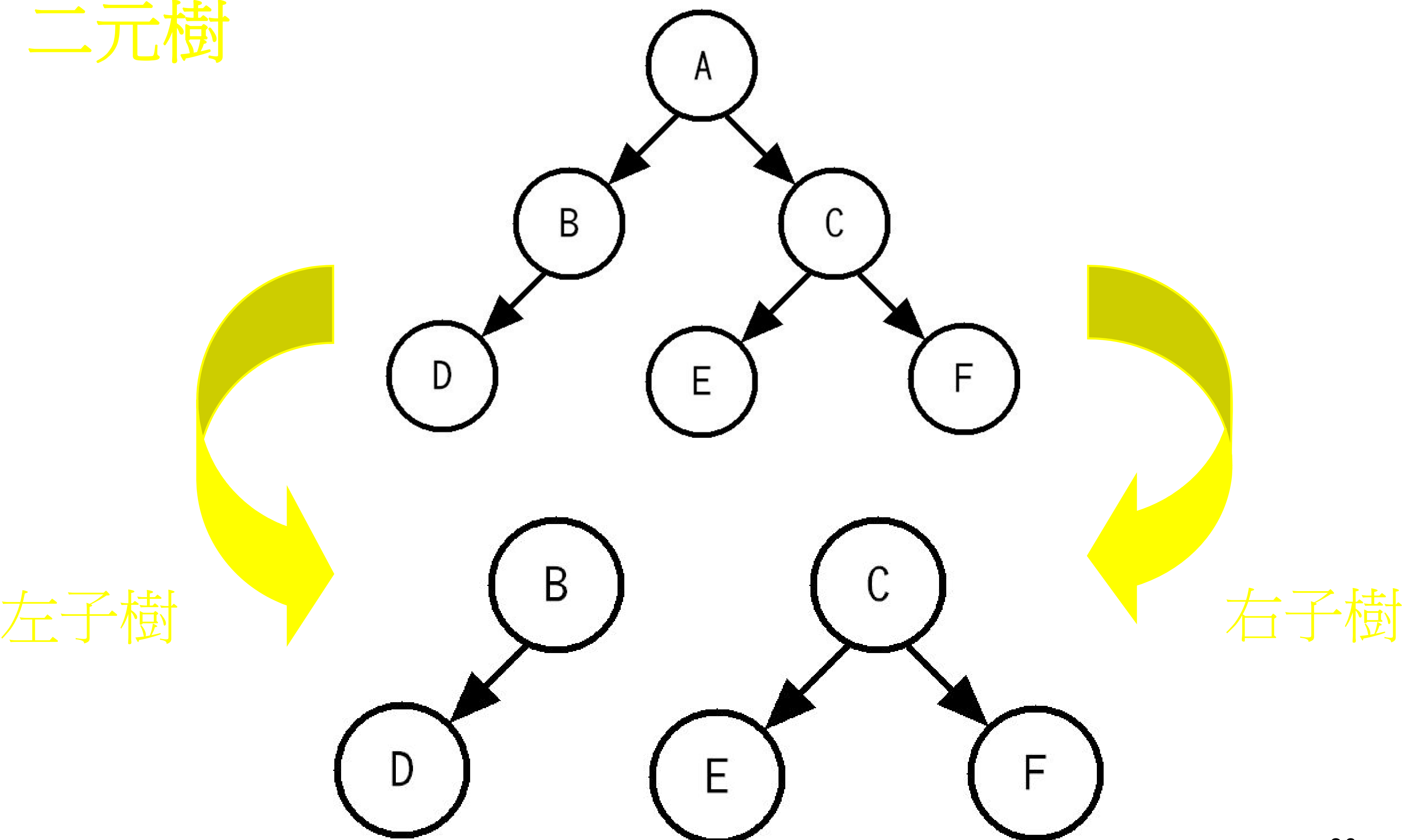


5.2 二元樹的基礎-定義

- 樹依不同分支度可以區分成很多種，在資料結構中最廣泛使用的樹狀結構是「二元樹」（Binary Trees），二元樹是指樹中的每一個「節點」（Nodes）最多只能擁有2個子節點，即分支度小於或等於2。
- 二元樹的定義如下所示：

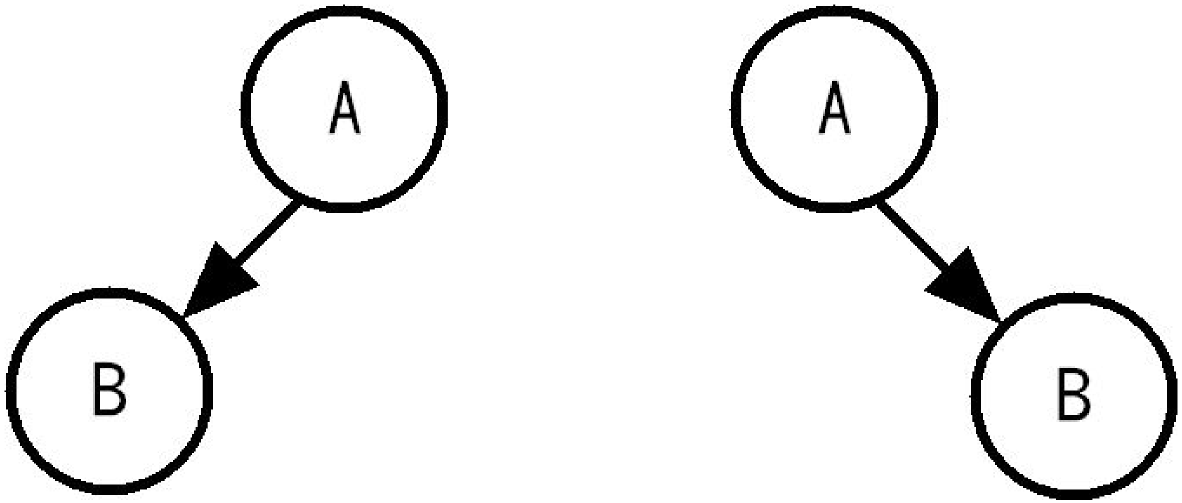
定義 7.2：二元樹的節點個數是一個有限集合，或是沒有節點的空集合。二元樹的節點可以分成兩個沒有交集的子樹，稱為「左子樹」（Left Subtree）和「右子樹」（Right Subtree）。

二元樹的基礎-圖例



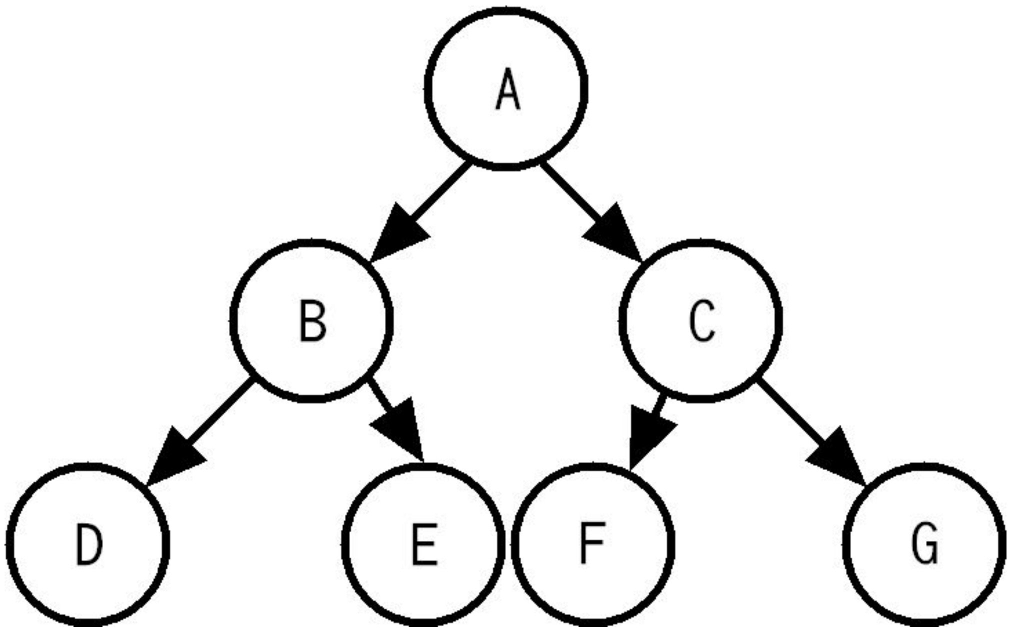
二元樹的基礎-歪斜樹

- 左邊這棵樹沒有右子樹，右邊這棵樹沒有左子樹，雖然擁有相同節點，但是這是兩棵不同的二元樹，因為所有節點都是向左子樹或右子樹歪斜，稱為「歪斜樹」（Skewed Tree），如下圖所示：



二元樹的基礎-
完滿二元樹(說明)

- 若二元樹的樹高是h且二元樹的節點數是 2^h-1，滿足此條件的樹稱為「完滿二元樹」（Full Binary Tree），如下圖所示：



二元樹的基礎-
完滿二元樹(節點數)

- 因為二元樹的每一個節點有2個子節點，二元樹樹高是3，也就是有3個階層（Level），各階層的節點數，如下所示：

  第1階：$1 = 2^{(1-1)} = 2^0 = 1$

  第2階：第1階節點數的2倍，$1*2 = 2^{(2-1)} = 2$

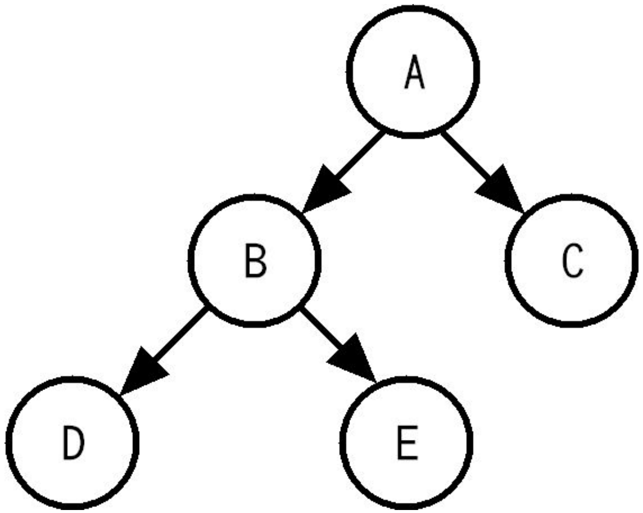
  第3階：第2階節點數的2倍，$2*2 = 2^{(3-1)} = 4$

- 以此類推，可以得到每一階層的最大節點數是：$2^{(l-1)}$，l是階層數，整棵二元樹的節點數一共是：$2^0+2^1+2^2 = 7$個，即2^{3-1}，可以得到：

  $2^0+2^1+2^2+....+2^{(h-1)} = 2^{h-1}$，h是樹高

二元樹的基礎-完整二元樹

- 若二元樹的節點不是葉節點，一定擁有2個子節點，不過節點總數不足2^h-1，其中h是樹高，而且其節點編號是對應相同高度完滿二元樹的１至2^h-1的節點編號，滿足此條件稱為完整二元樹（Complete Binary Tree），如下圖所示：



樹和二元樹的基本性質

樹或二元樹皆擁有下面的性質：

定理**5-1**：若一棵樹**T**的總節點數為**V**，總邊數為 **E**，則

$$\mathbf{V = E + 1}$$ 。

- 若二元樹終端節點總數為n_0，分求度等於２的節點總數為n_2，則$n_0=n_2+1$。

假設節點總數為n，分支度等於１的節點總數為n_1，

$=>n=n_0+n_1+n_2$

假設節點分支數為m，

$=>m=n-1$

$=>m=1*n_1+2*n_2$

$=>1*n_1+2*n_2=n_0+n_1+n_2-1$

Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all *bt*, *bt1*, *bt2* $\in$ *BinTree*, *item* $\in$ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

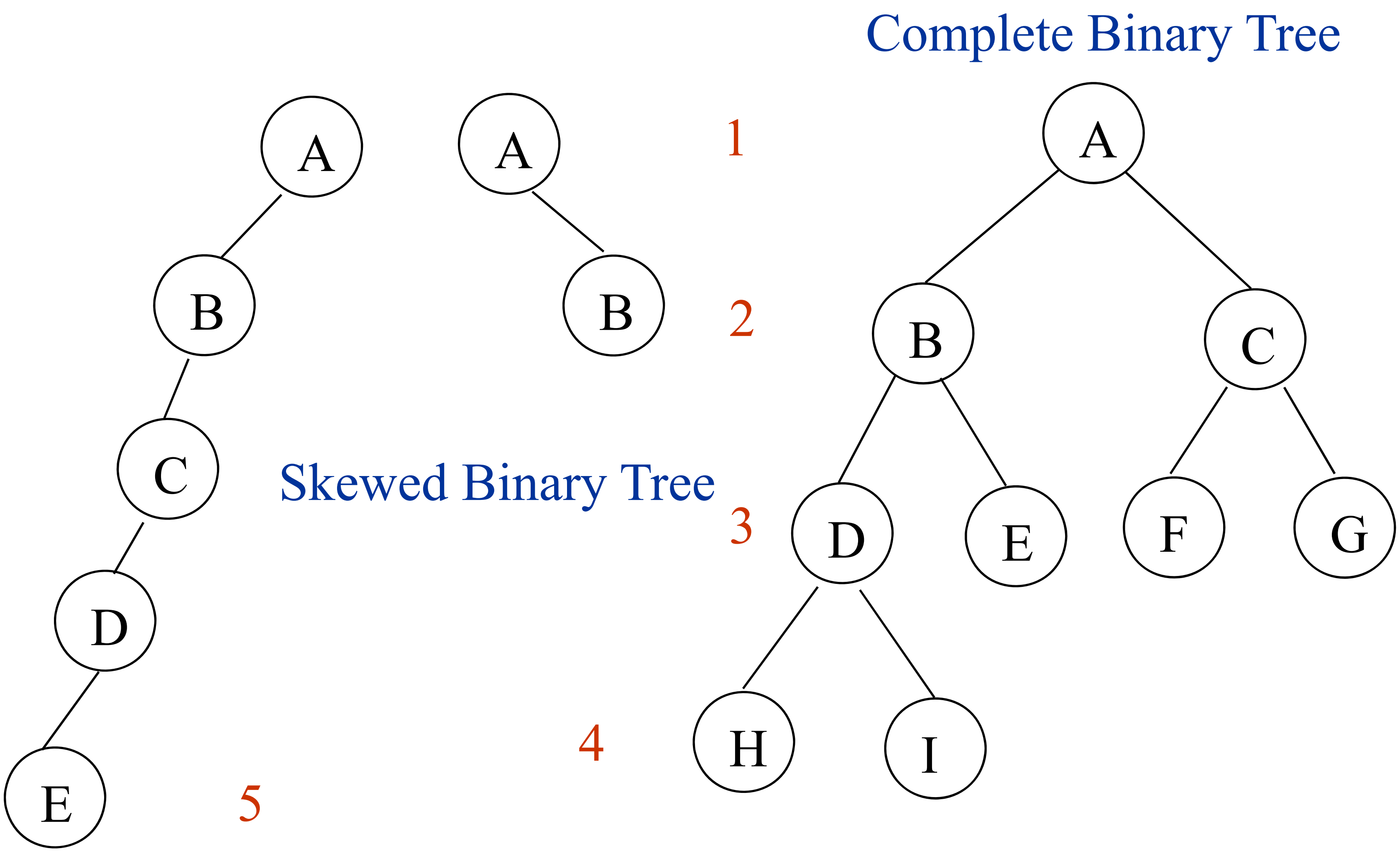
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Samples of Trees



Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a
i-1 binary tree is 2^{i-1}, i>=1.
- The maximum nubmer of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction.

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 *is the number of leaf nodes and* n_2 *the number of nodes of degree 2, then* $n_0=n_2+1$

proof:

Let n and B denote the total number of nodes & branches in T.

Let n_0, n_1, n_2 represent the nodes with no children single child, and two children respectively.

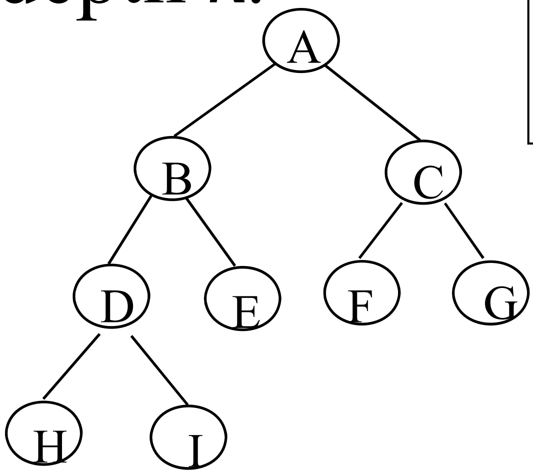
$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$

$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

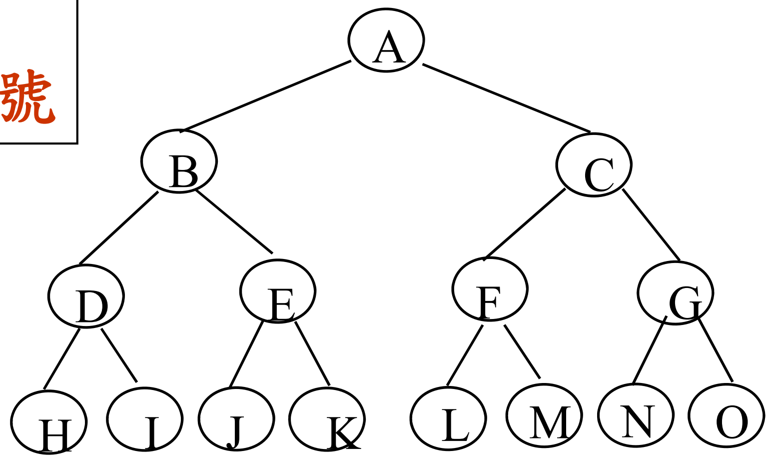
Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree of depth *k* having 2^k-1 nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

由上至下，
由左至右編號



Complete binary tree



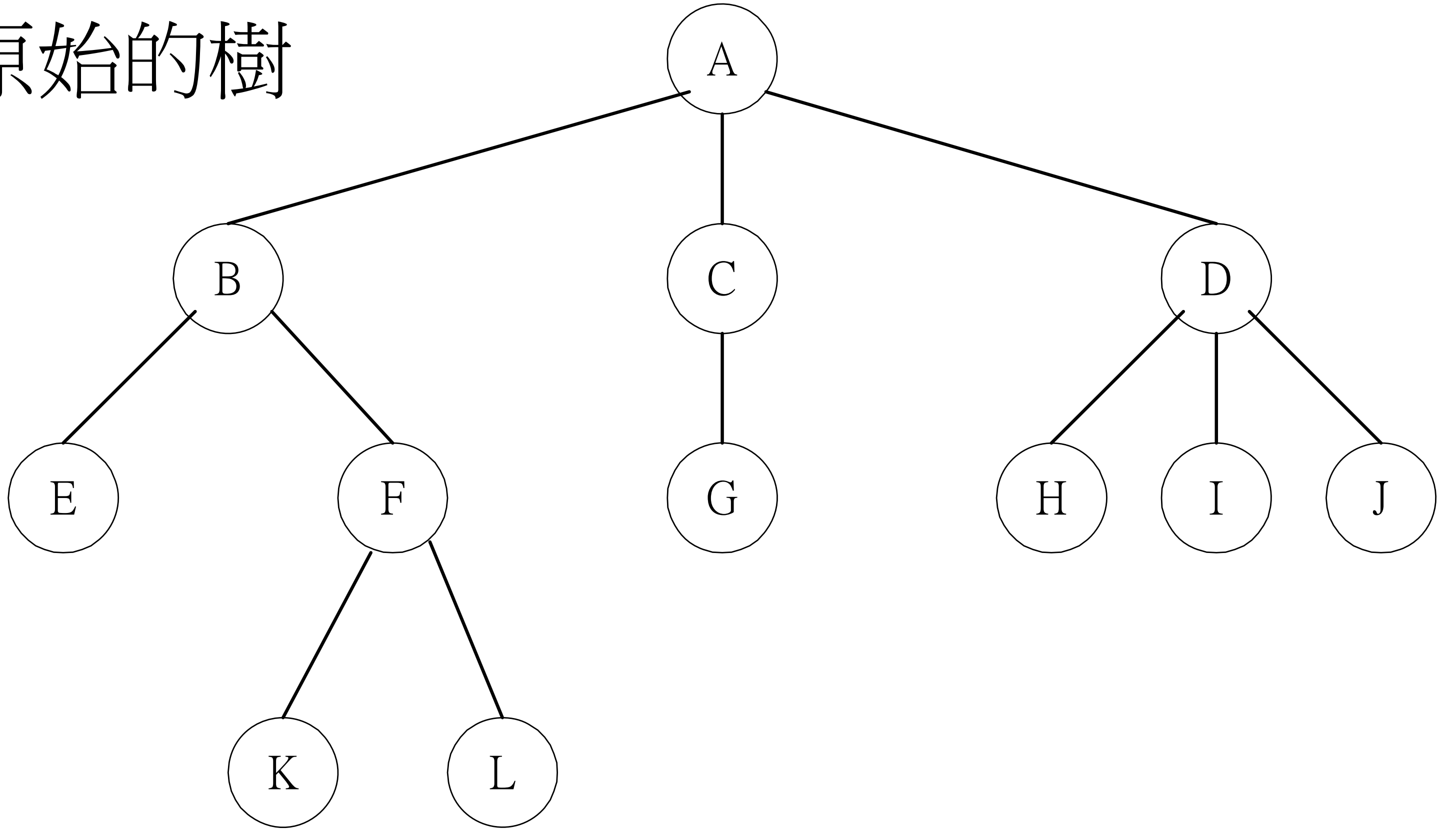
Full binary tree of depth 4

樹轉換成二元樹

- 刪除所有節點的右鏈結，只保留下左鏈結；
- 將原來樹中同屬於一個父節點的兄弟用鏈結連接起來；
- 將整個圖形以順時針方向旋轉45°。

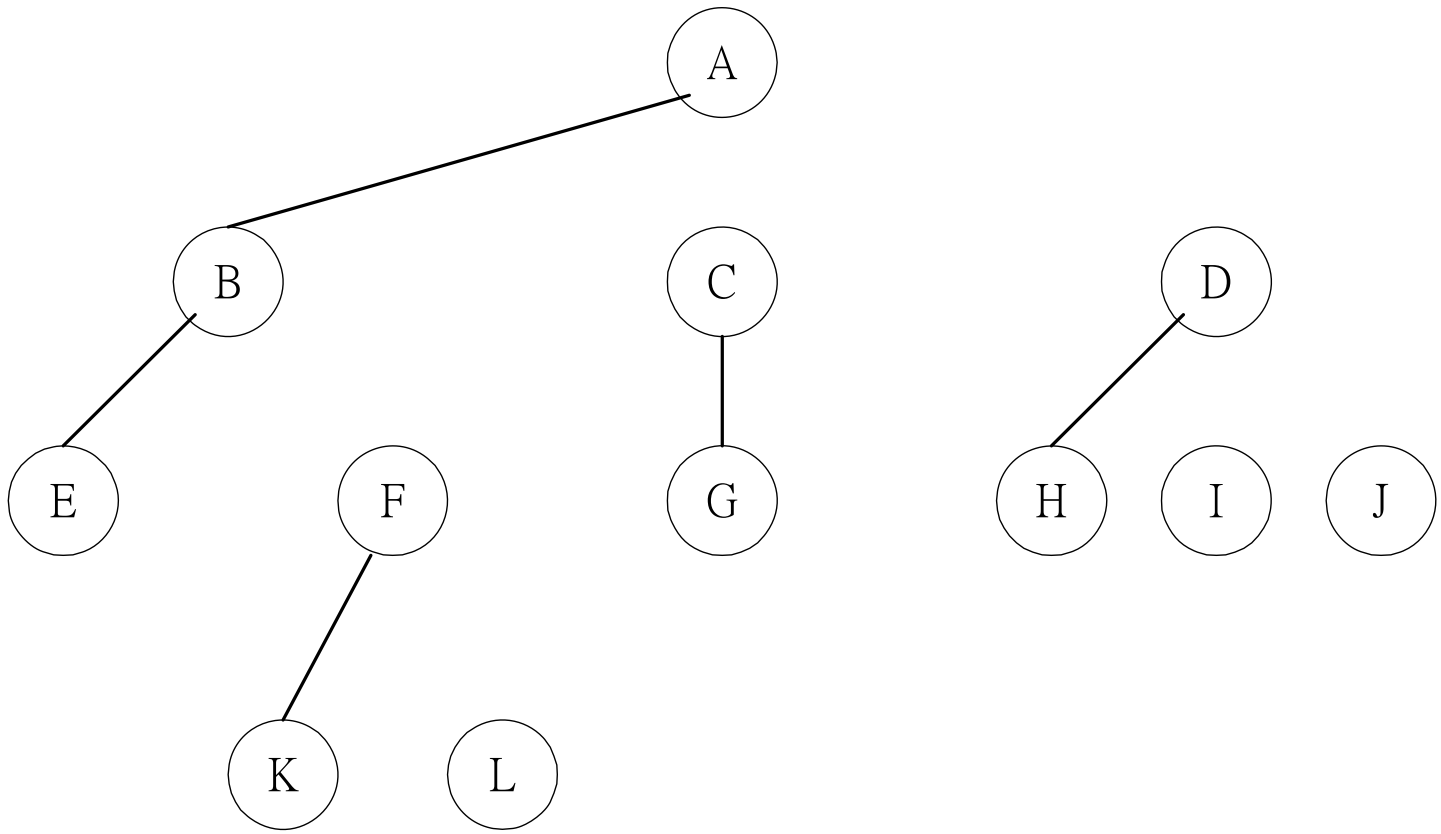
樹轉換成二元樹

- 原始的樹



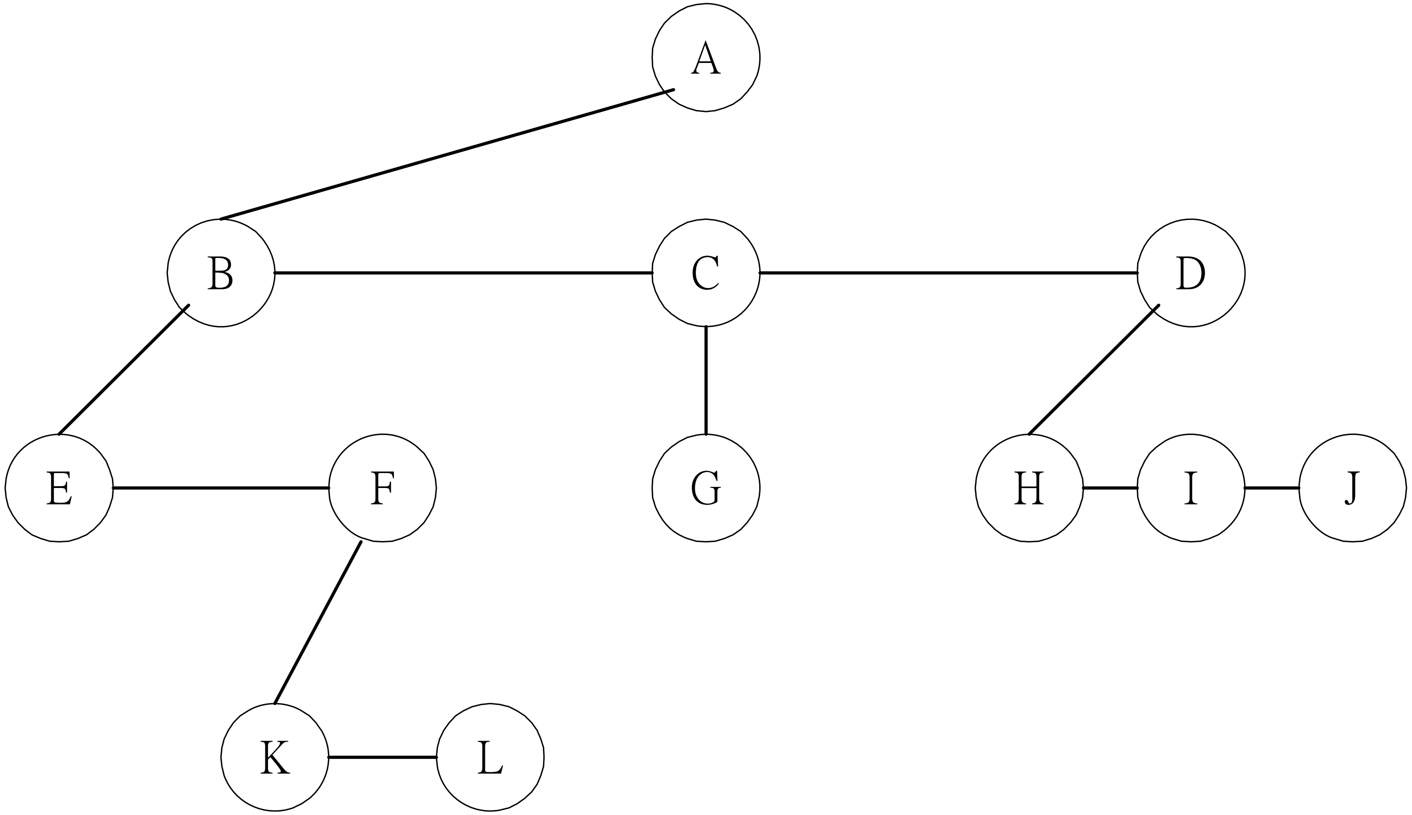
樹轉換成二元樹

- 保留所有節點的左鏈結，其餘都刪除，



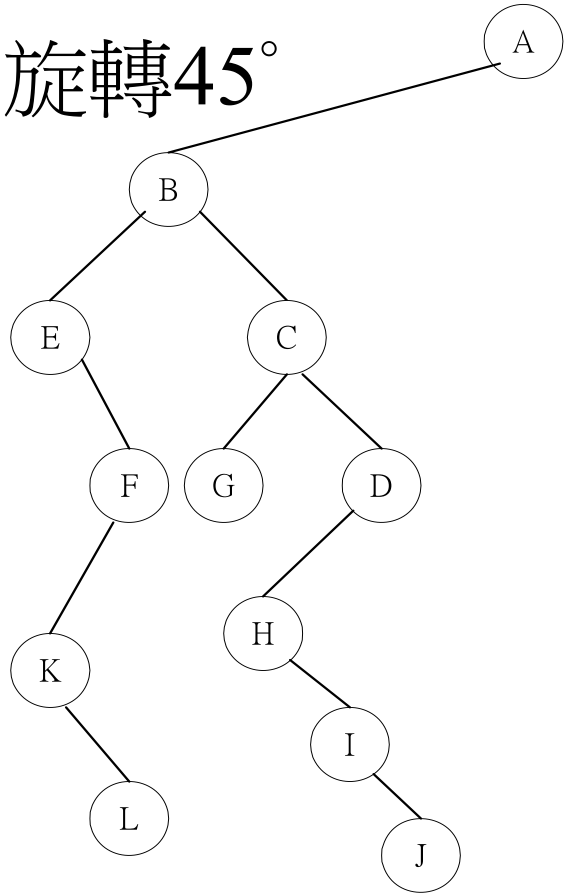
樹轉換成二元樹

- 將原來的兄弟節點用鏈結連接起來，



樹轉換成二元樹

- 順時針方向旋轉45°

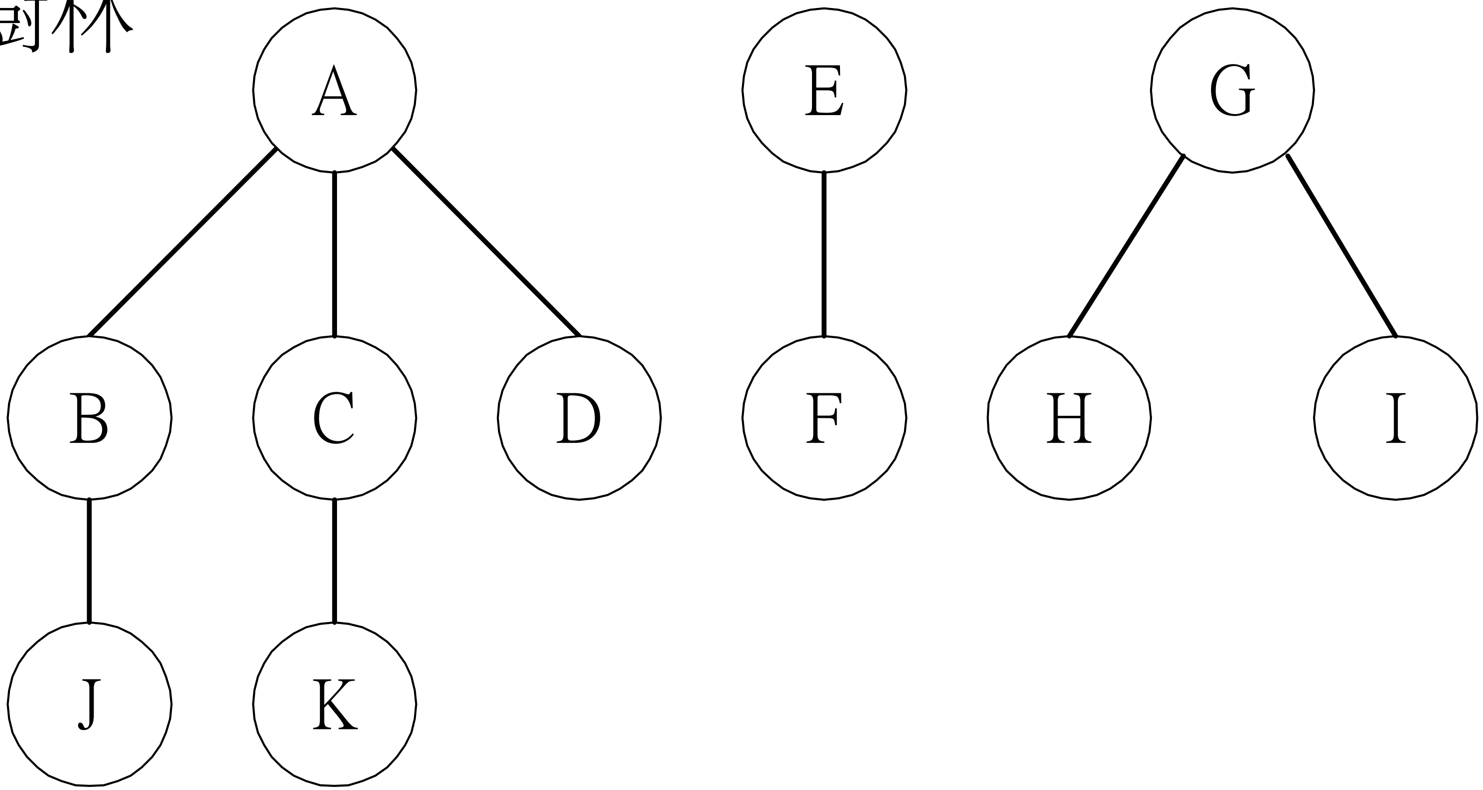


將樹林轉成二元樹的方法

- 首先將各樹樹根連結起來；
- 刪除所有節點之右鏈結，只留下左鏈結；
- 將原來樹中同屬於一個父節點的兄弟用鏈結連接起來；
- 將整個圖形以順時針方向旋轉45°

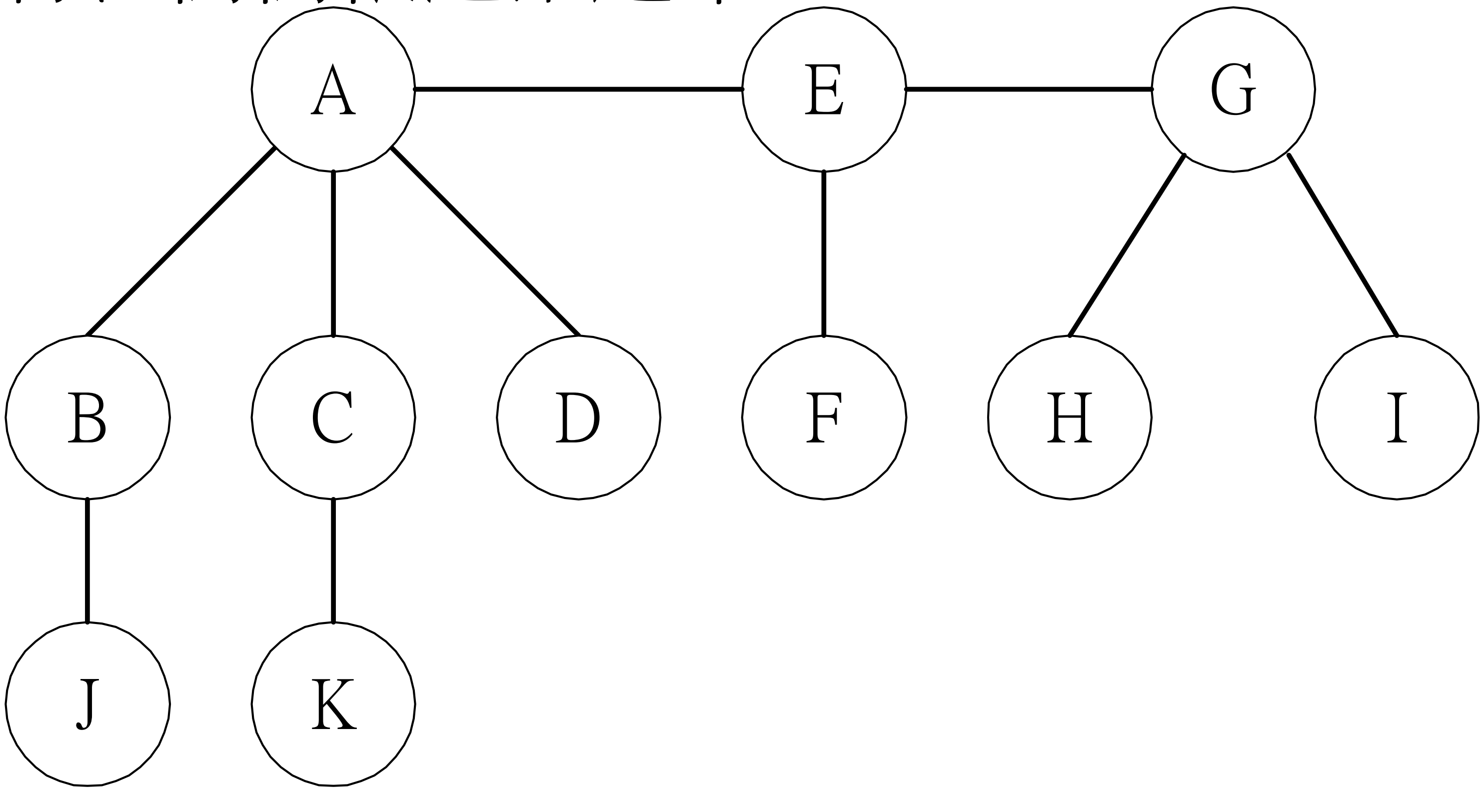
將樹林轉成二元樹的方法

- 樹林



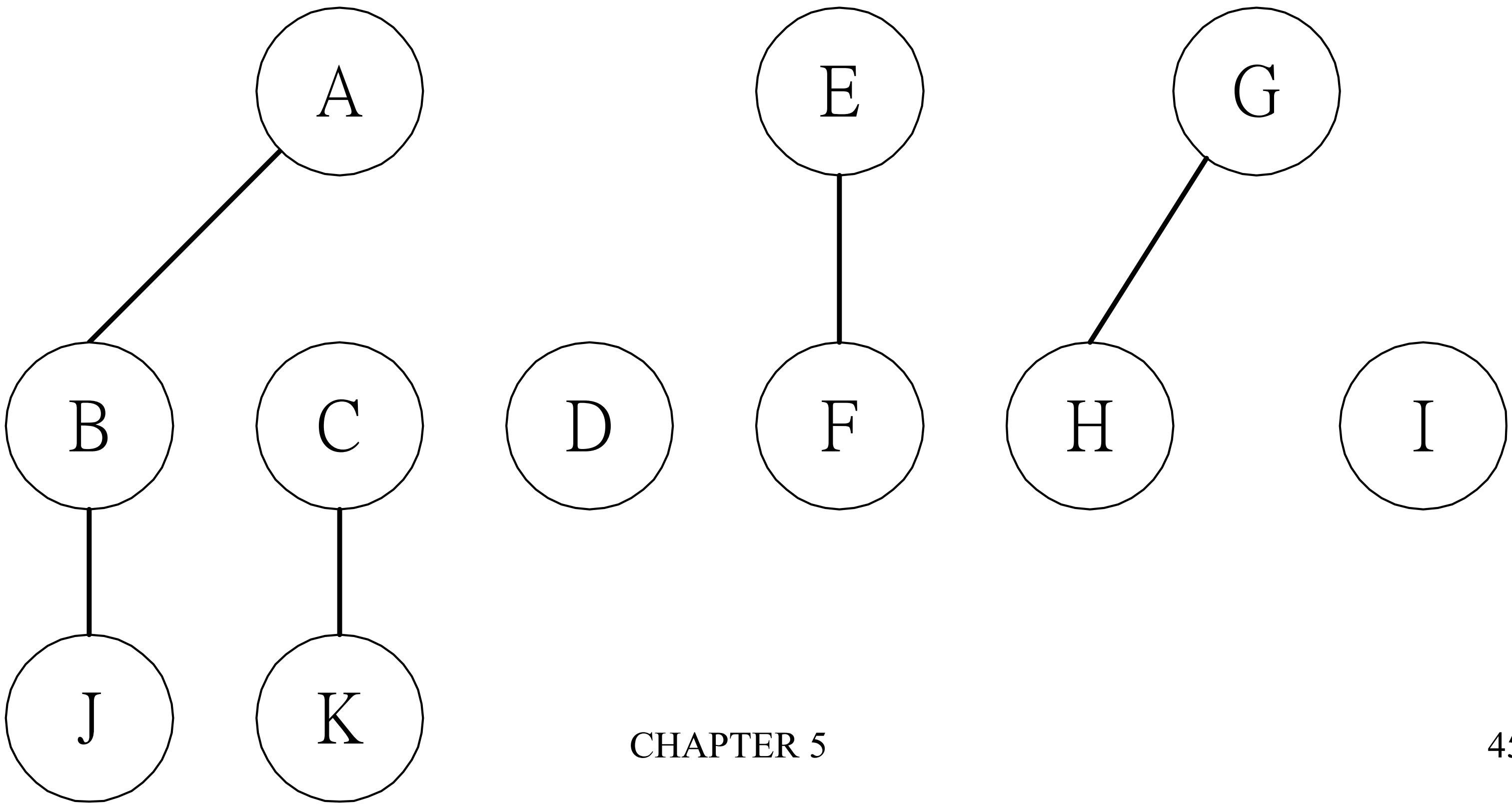
將樹林轉成二元樹的方法

- 將各樹樹根連結起來



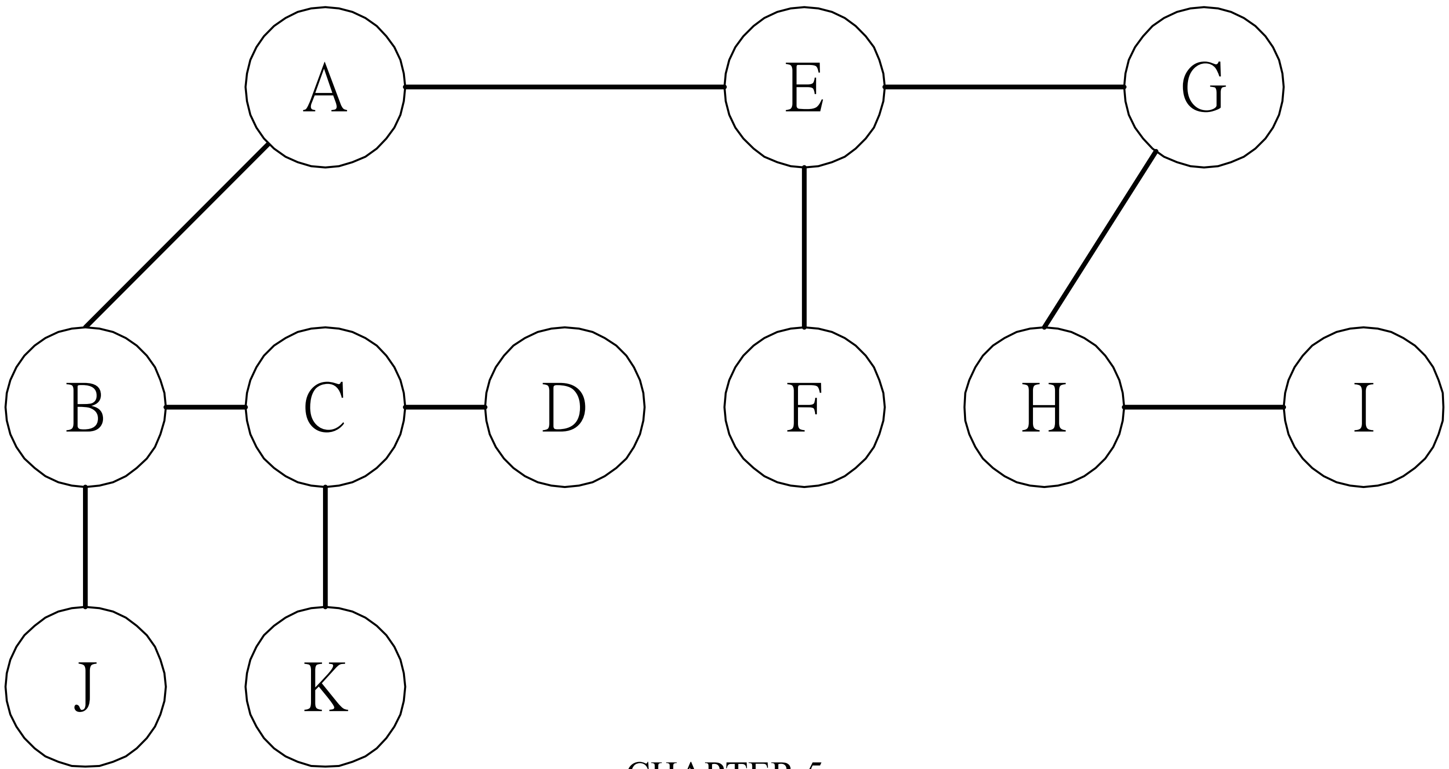
將樹林轉成二元樹的方法

- 刪除所有的右鏈結，只留下左鏈結



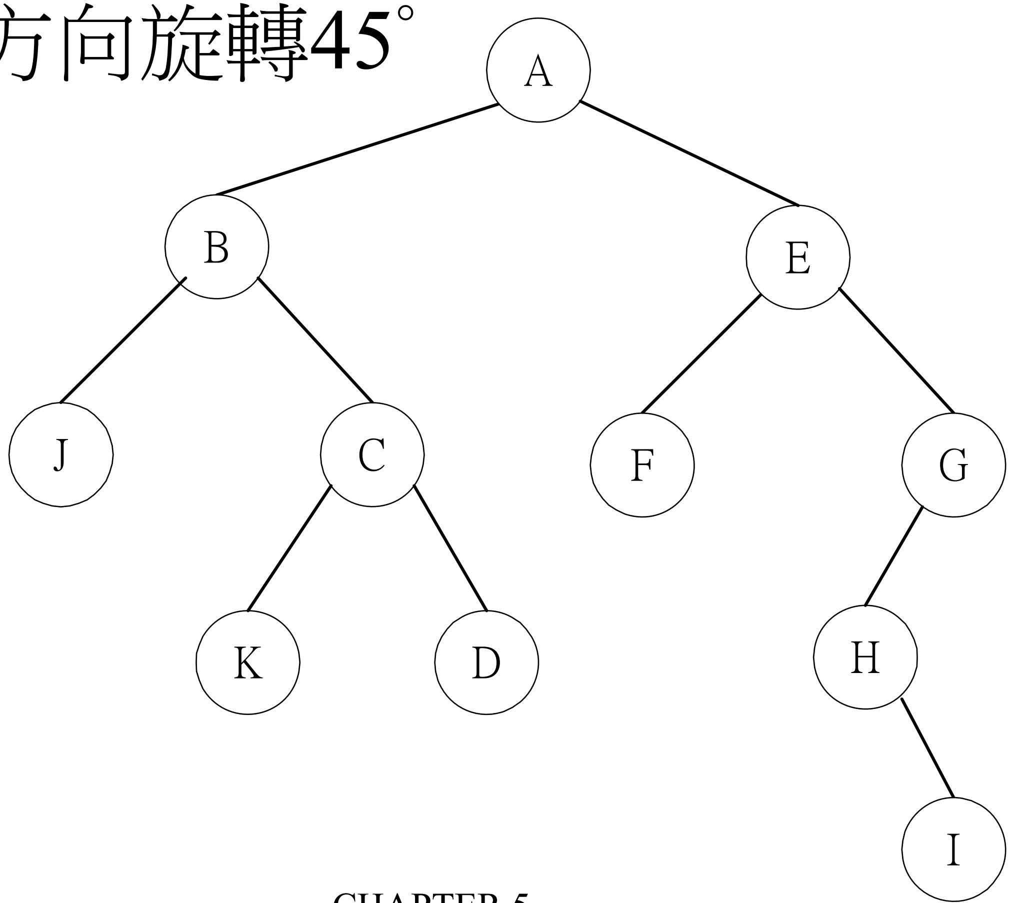
將樹林轉成二元樹的方法

- 將原來樹中的兄弟用鏈結連接起來



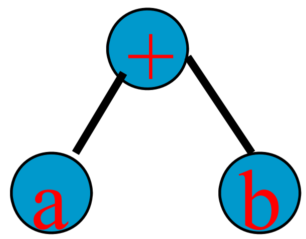
將樹林轉成二元樹的方法

- 順時針方向旋轉45°

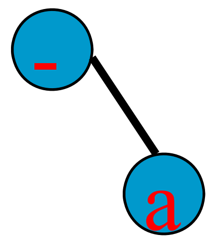


建立算術運算之二元樹

- a + b

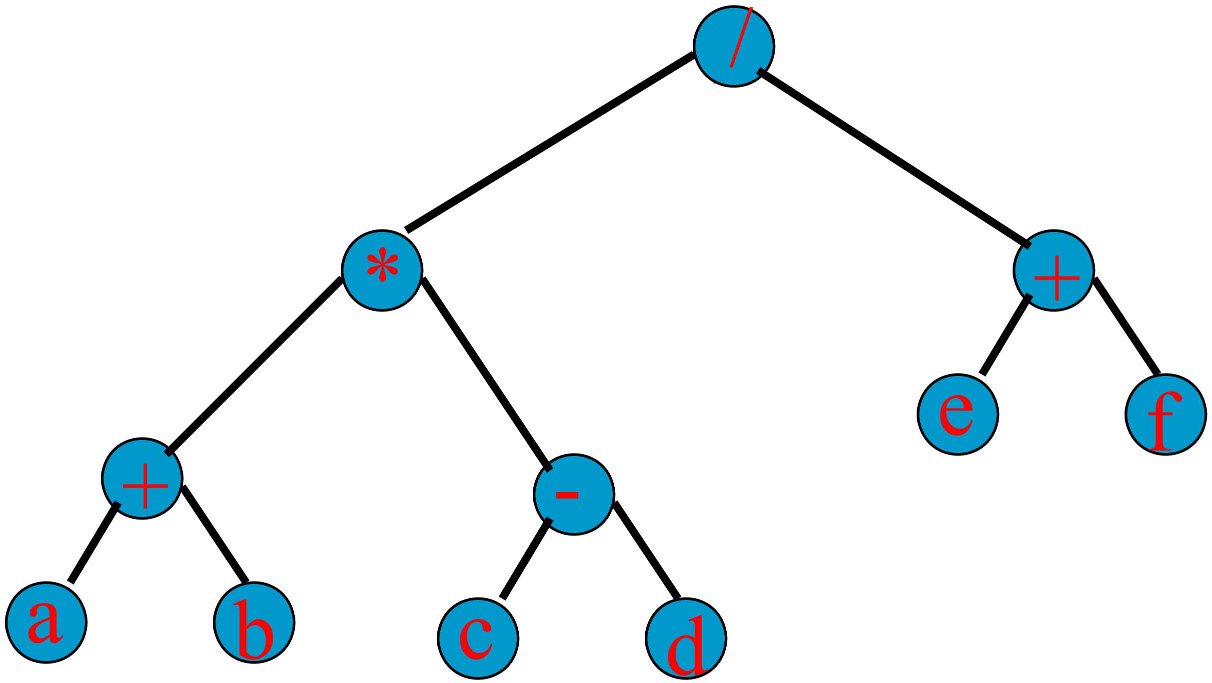


- \- a



建立算術運算之二元樹

- (a + b) * (c – d) / (e + f)

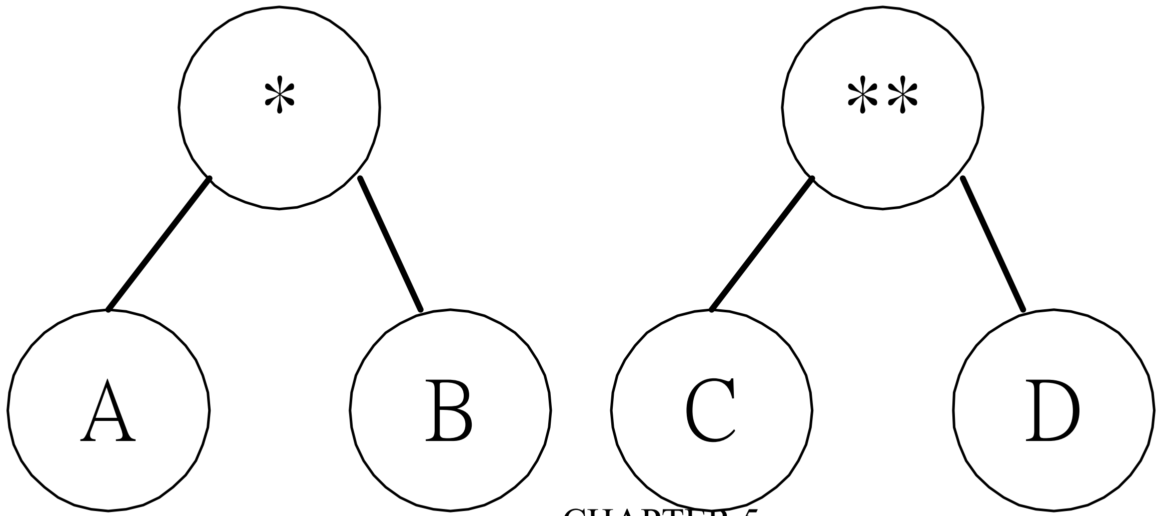


建立算術運算之二元樹

- 考慮運算子的優先次序與結合性，適當地加以括號；
- 由內層的括號逐次向外，並且以運算子當樹根，左邊運算元當左子樹，右邊運算元當右子樹。

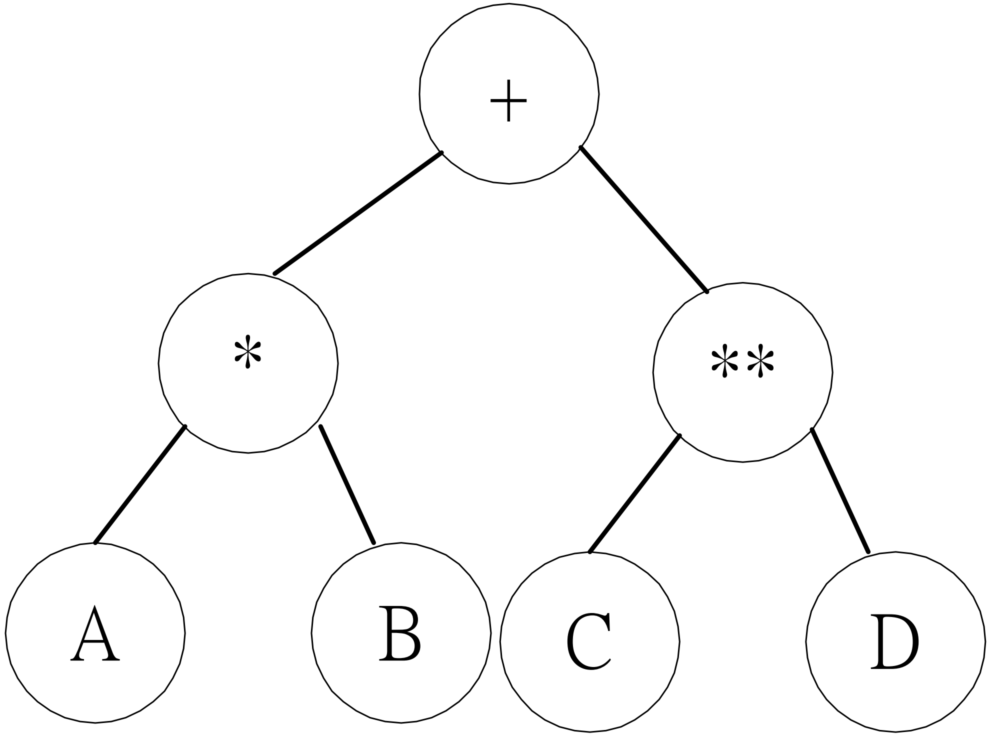
建立算術運算之二元樹

- 將A*B+C**D-E建成二元樹
- 按照運算子的優先權和結合性加以適當括號，得到(((A*B)+(C**D))-E)



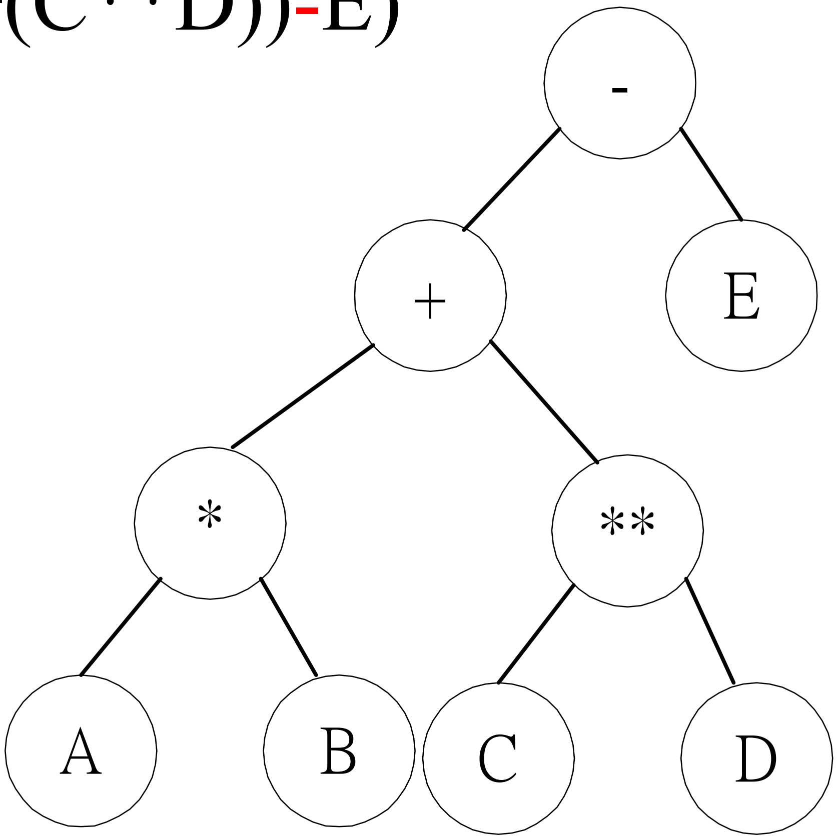
建立算術運算之二元樹

- ((A*B)+(C**D))



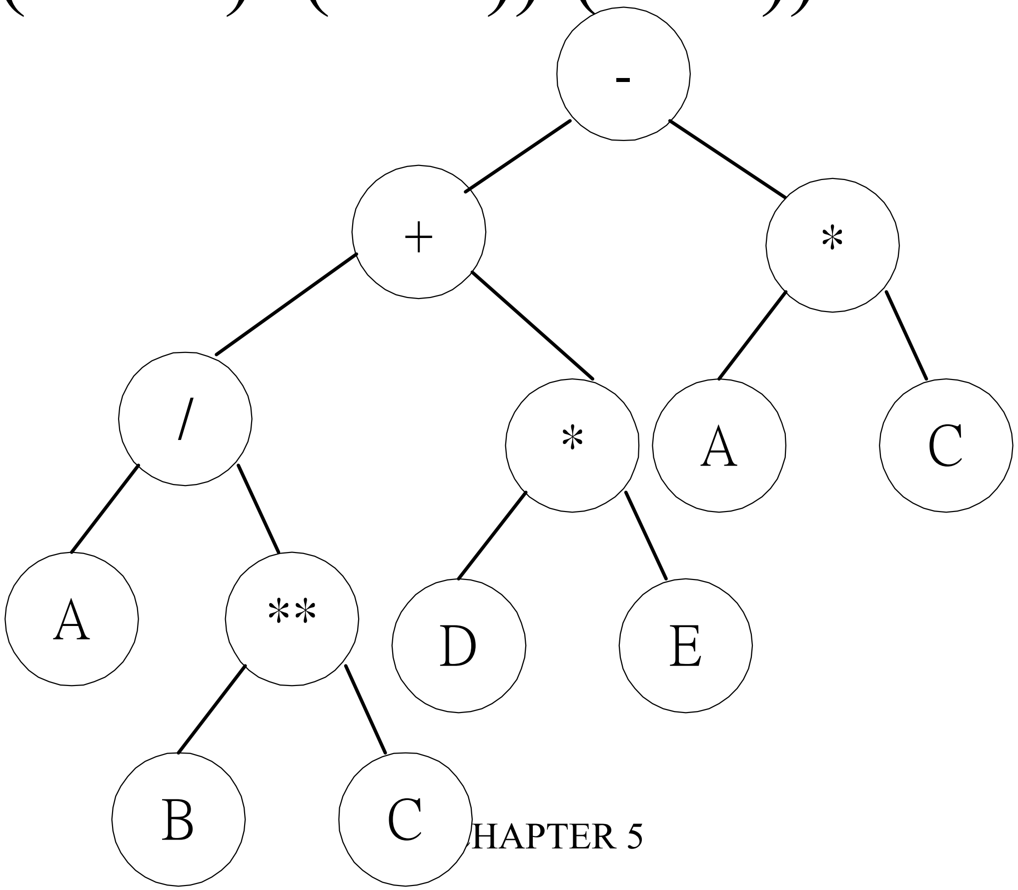
建立算術運算之二元樹

- (((A*B)+(C**D))-E)



建立算術運算之二元樹

- (((A/(B**C)+(D*E))-(A*C))

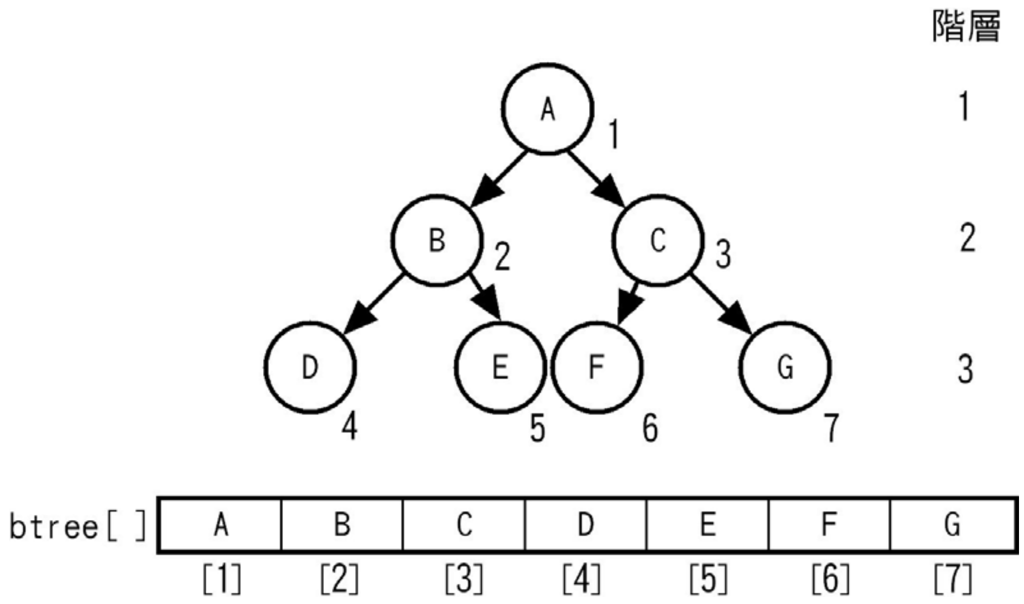


5.2.3 二元樹的表示法

- 二元樹在實作上有多種方法可以建立二元樹，常用的方法有二種，如下所示：
  - 二元樹陣列表示法。
  - 二元樹鏈結表示法。

二元樹陣列表示法-說明1

- 完滿二元樹是一棵樹高h擁有2^h-1個節點的二元樹，這是二元樹在樹高h所能擁有的最大節點數，換句話說，只需配置2^h-1個元素，我們就可以儲存樹高h的二元樹，如下圖所示：



二元樹陣列表示法-說明2

- 二元樹的節點編號擁有循序性，根節點1的子節點是節點2和節點3，節點2是4和5，依此類推可以得到節點編號的規則，如下所示：
  - 左子樹是父節點編號乘以2。
  - 右子樹是父節點編號乘以2加1。

二元樹陣列表示法-標頭檔

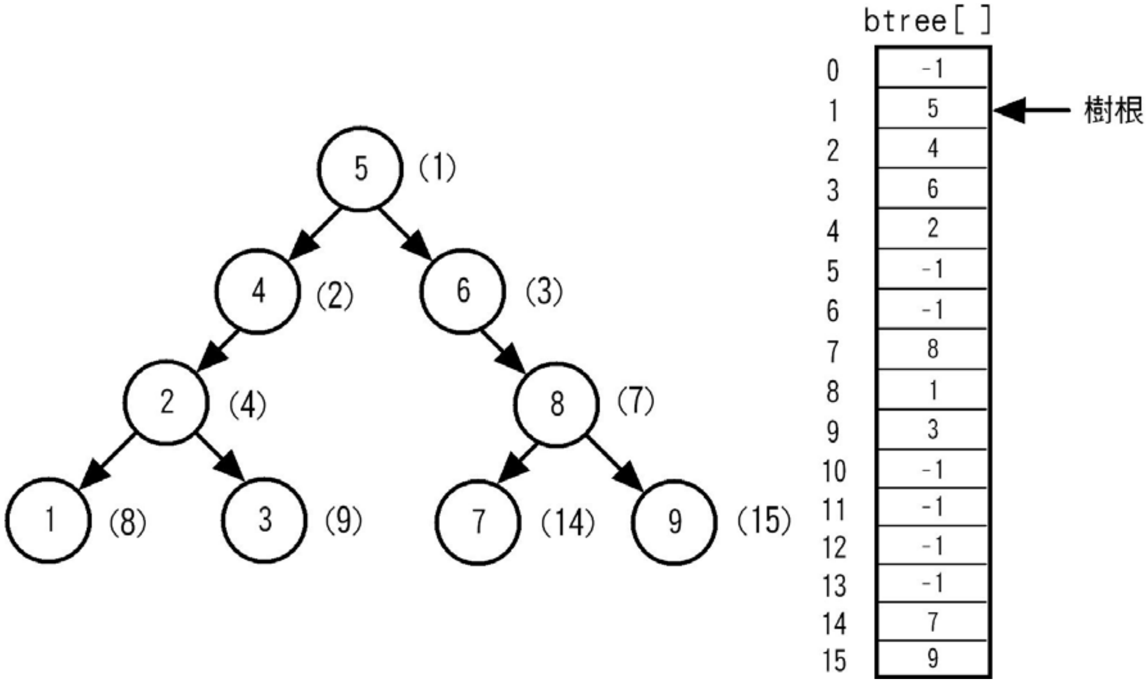
```
#define MAX_LENGTH   16 /* 最大陣列尺寸 */
int btree[MAX_LENGTH];  /* 二元樹陣列宣告 */
/* 抽象資料型態的操作函數宣告 */
extern void createBTree(int len, int *array);
extern void printBTree();
```

二元樹陣列表示法-建立二元樹(規則)

- 函數createBTree()讀取一維陣列的元素建立二元樹，其建立的規則，如下所示：
  - 將第1個陣列元素插入成為二元樹的根節點。
  - 將陣列元素值與二元樹的節點值比較，如果元素值大於節點值，將元素值插入成為節點的右子節點，如果右子節點不是空的，重覆比較節點值，直到找到插入位置後，將元素值插入二元樹。
  - 如果元素值小於節點值，將元素值插入成為節點的左子節點，如果左子節點不是空的，繼續重覆比較，以便將元素值插入二元樹。

二元樹陣列表示法-建立二元樹(圖例)

- 二元樹陣列表示法圖例的索引值0並沒有使用，整個二元樹在16個陣列元素中使用的元素一共有9個，括號內是陣列的索引值，如下圖所示：



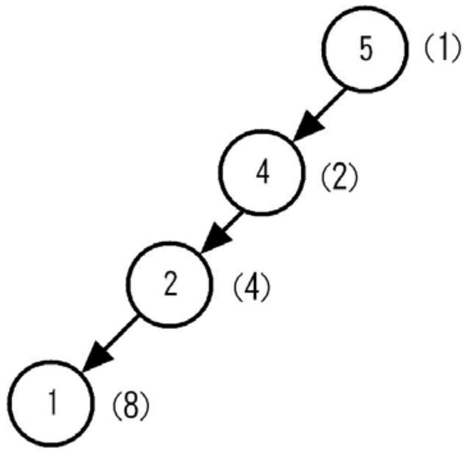
二元樹陣列表示法-顯示二元樹

<u>函數printBTree()：顯示二元樹</u>

- 函數printBTree()走訪btree[]陣列，將元素值不是-1的元素都顯示出來。

二元樹陣列表示法-問題

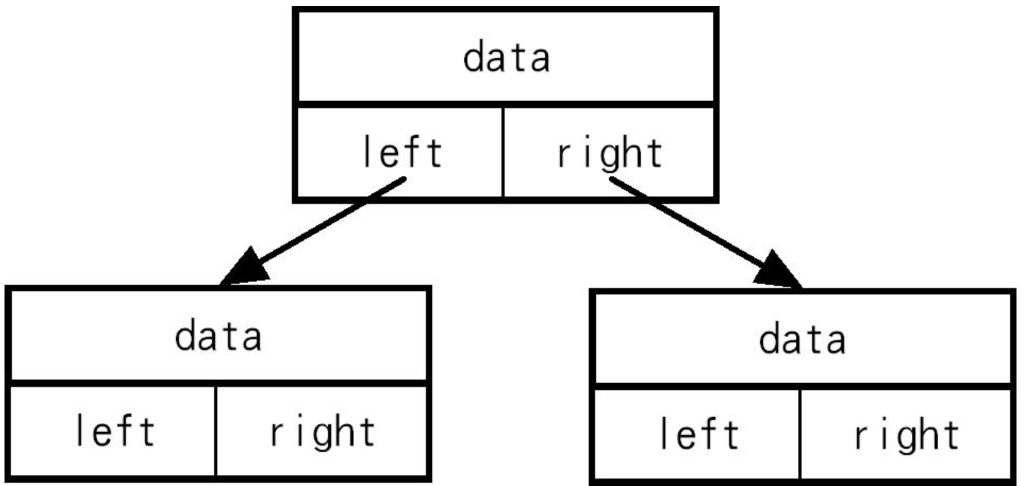
- 一棵歪斜樹的二元樹陣列表示法使用不到三分之一的陣列元素4/16，因為二元樹的節點是以循序方式儲存在陣列中，如果需要插入或刪除節點，都需要在陣列中搬移大量元素，如下圖所示：



btree[ ]	-1	5	4	-1	2	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

二元樹鏈結表示法-說明

- 二元樹鏈結表示法是使用動態記憶體配置來建立二元樹，類似結構陣列表示法的節點結構，只是成員變數改成兩個指向左和右子樹的指標，如下圖所示：

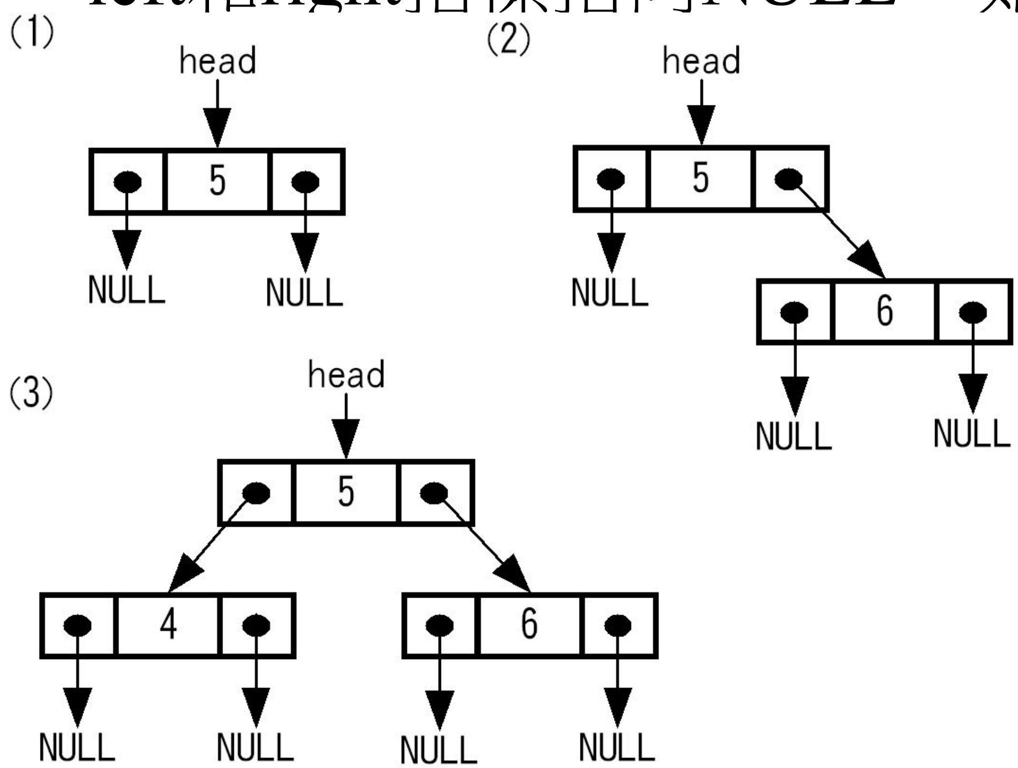


二元樹鏈結表示法-標頭檔

```
struct Node {           /* 二元樹的節點宣告 */
    int data;           /* 儲存節點資料 */
    struct Node *left;  /* 指向左子樹的指標 */
    struct Node *right; /* 指向右子樹的指標 */
};
```

二元樹鏈結表示法-建立二元樹1

- 函數createBTree()使用for迴圈走訪參數的陣列元素，依序呼叫insertBTreeNode()函數將一個一個陣列元素的節點插入二元樹。首先是二元樹的根節點5，left和right指標指向NULL，如下圖所示：

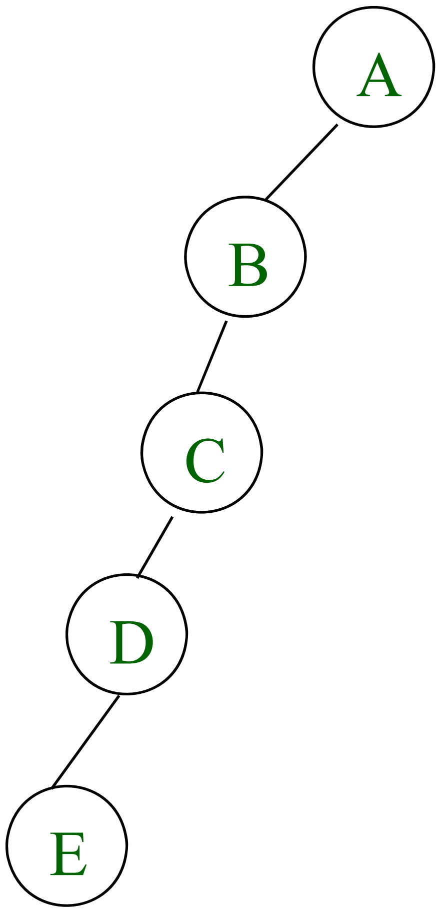


Binary Tree Representations

- If a complete binary tree with n nodes (depth = log n + 1) is represented sequentially, then for any node with index i, $1<=i<=n$, we have:
  - *parent*(i) is at $i/2$ if $i!=1$. If $i=1$, i is at the root and has no parent.
  - *left_child*(i) ia at $2i$ if $2i<=n$. If $2i>n$, then i has no left child.
  - *right_child*(i) ia at $2i+1$ if $2i+1<=n$. If $2i+1>n$, then i has no right child.

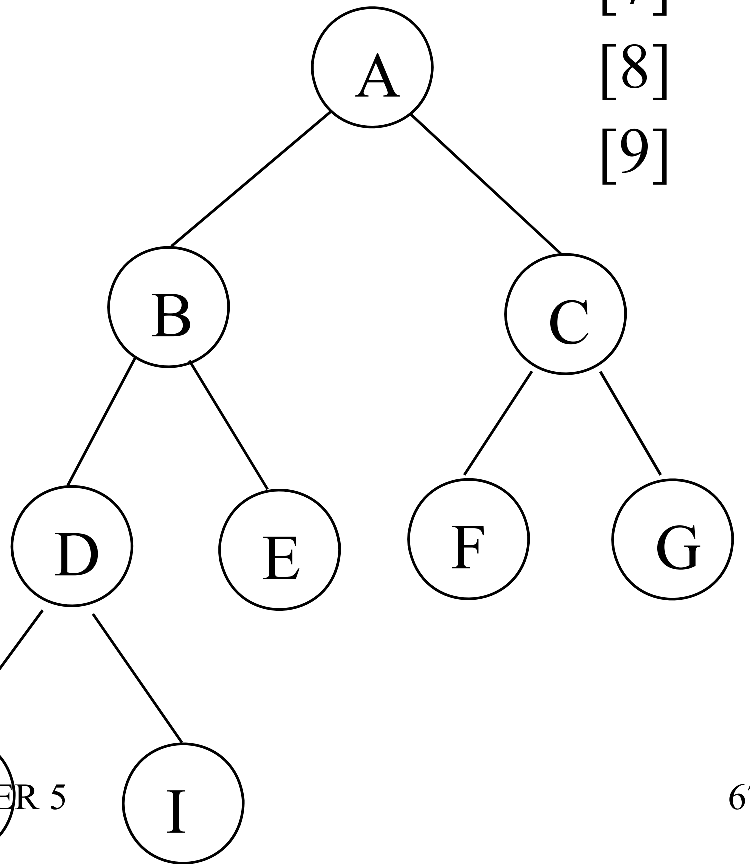
Sequential Representation

(1) waste space
(2) insertion/deletion problem



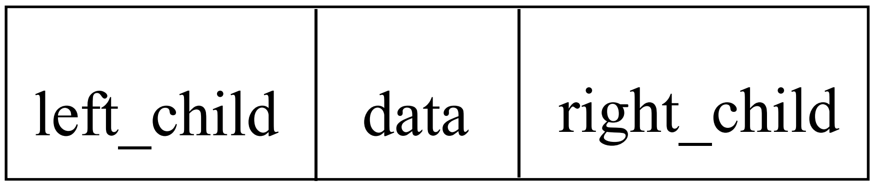
[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

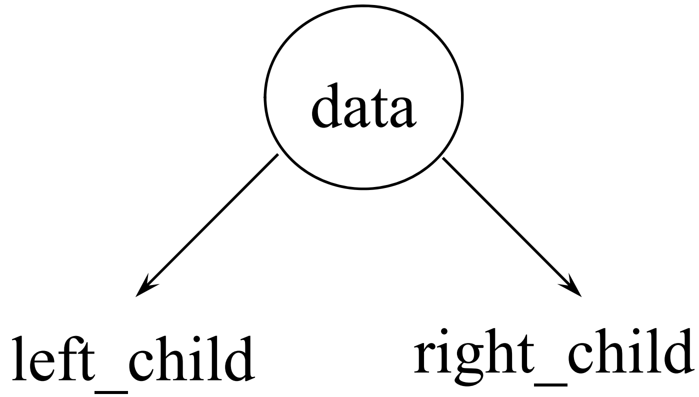
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I



Linked Representation

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```





作業

pp. 204: ex3

5.3 二元樹的走訪

- 假如一組資料已用二元樹的組織在一起，總有需求對其全數資料做動作，例如計算所有數目、印出所有資料、在所有資料中搜尋某項資料、...等。此時即須對此二元樹做走訪 **(traversal)** 的運算，利用走訪二元樹的同時，將計算、列印或搜尋的動作完成。
- 事實上走訪即在決定二元樹上資料被處理（計算、列印或搜尋）的順序。我們也希望走訪的演算法對任何節點皆一致，是容易撰寫程式實作的。
- 若對一個二元樹上的節點而言，**V**表示處理節點上的資料，**L**表示走訪其左子樹，**R**表示走訪其右子樹。

二元樹的走訪(續)

- 因為對稱的緣故，我們可以只考慮先走訪左邊再走訪右邊的情形，那麼圖**5-20 (b)** 則只剩下三種走訪方式，我們以**V**所在的相對位置，分別對此三種走訪方式取名如下：
  - **(1) LVR**中序走訪 **(inorder traversal)** 或中序表示法 **(infix notation)**；
  - **(2) LRV**後序走訪 **(postorder traversal)** 或後序表示法 **(postfix notation)**；
  - **(3) VLR**前序走訪 **(preorder traversal)** 或前序表示法 **(prefix notation)**。

中序走訪方式-說明

- 中序走訪是沿著二元樹的左方往下走，直到無法繼續前進後，顯示節點，退回到父節點顯示父節點，然後繼續往右走，如果右方都無法前進，顯示節點，再退回到上一層。

中序走訪方式-演算法

- 中序走訪的遞迴函數inOrder()使用二元樹指標ptr進行走訪，中序走訪的步驟，如下所示：
  - Step 1：檢查是否可以繼續前進，即指標ptr不等於NULL。
  - Step 2：如果可以前進，其處理方式如下所示：
    - (1) 遞迴呼叫inOrder(ptr->left)向左走。
    - (2) 處理目前的節點，顯示節點資料。
    - (3) 遞迴呼叫inOrder(ptr->right)向右走。

中序走訪

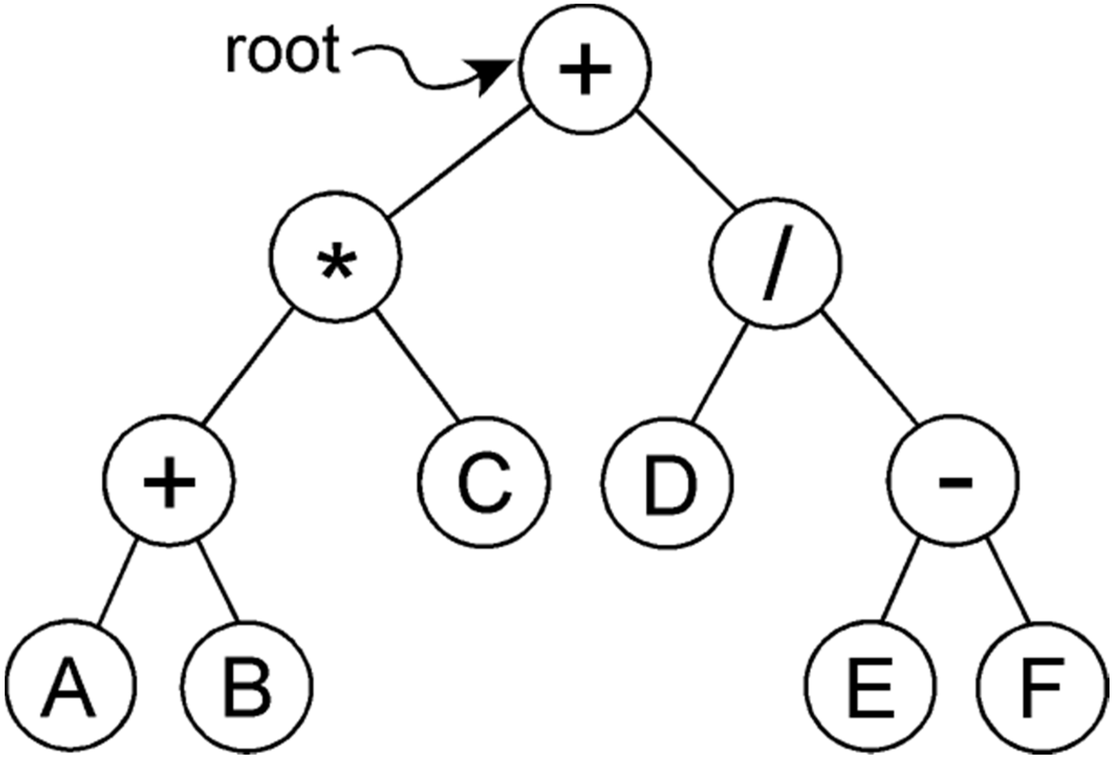
- 利用遞迴的方式撰寫中序走訪的程序；希望把二元樹中序走訪的順序印出來。

程式　二元樹的中序走訪

```
struct BTreeNode
{   struct BTreeNode *leftchild;
    char data;
    struct BTreeNode *rightchild;
};
struct BTreeNode *root;
void inorder(struct BTreeNode *node)
{   if (node != NULL)
    {    inorder(node->leftchild);
              cout << node->data;
              inorder(node->rightchild);
         }
    }
```

範例

下圖為一棵運算式二元樹。



其對應中序表示運算式為：

(A+B)*C+D/(E-F)。

後序走訪方式-說明

- 後序走訪方式剛好和前序走訪相反，它是等到節點的2個子節點都走訪過後才執行處理，顯示節點資料。

後序走訪方式-演算法

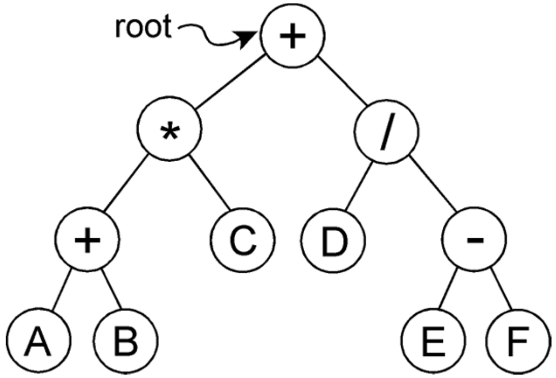
- 後序走訪的遞迴函數postOrder()使用二元樹指標ptr進行走訪，後序走訪的步驟，如下所示：
  - Step 1：先檢查是否已經到達葉節點，就是指標ptr等於NULL。
  - Step 2：如果不是葉節點表示可以繼續走，其處理方式如下所示：
    - (1) 遞迴呼叫postOrder(ptr->left)向左走。
    - (2) 遞迴呼叫postOrder(ptr->right)向右走。
    - (3) 處理目前的節點，顯示節點資料。

後序走訪

茲將後序走訪的程序撰寫如下：

程式　二元樹的後序走訪

```
void postorder(struct BTreeNode *node)
{     if (node !=NULL)
      {     postorder(node->leftchild);
            postorder(node->rightchild);
            cout << node->data;
      }
}
```



以後序走訪，列出右圖之運算式二元樹中的所有資料，可得到：

AB+C*D-/+

其正為對應的後序運算式。

前序走訪方式-說明

- 前序走訪方式是走訪到的二元樹節點，就立刻顯示節點資料，走訪的順序是先向樹的左方走直到無法前進後，才轉往右方走。

前序走訪方式-演算法

- 前序走訪的遞迴函數preOrder()使用二元樹指標ptr進行走訪，前序走訪的步驟，如下所示：
  - Step 1：先檢查是否已經到達葉節點，也就是指標ptr等於NULL。
  - Step 2：如果不是葉節點表示可以繼續走，其處理方式如下所示：
    - (1) 處理目前的節點，顯示節點資料。
    - (2) 遞迴呼叫preOrder(ptr->left)向左走。
    - (3) 遞迴呼叫preOrder(ptr->right)向右走。

前序走訪

前序走訪的程序可撰寫如下：

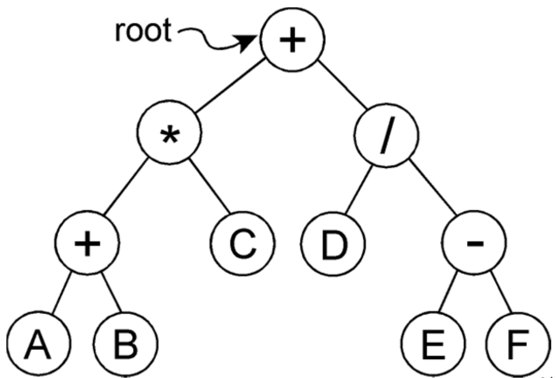
程式**5-5**二元樹的前序走訪

```
void preorder(struct BTreeNode *node)
{    if (node != NULL)
     {    cout << node->data;
          preorder(node->leftchild);
          preorder(node->rightchild);
     }
}
```

以前序走訪，列出右圖之運算式二元樹中的所有資料，可得到：

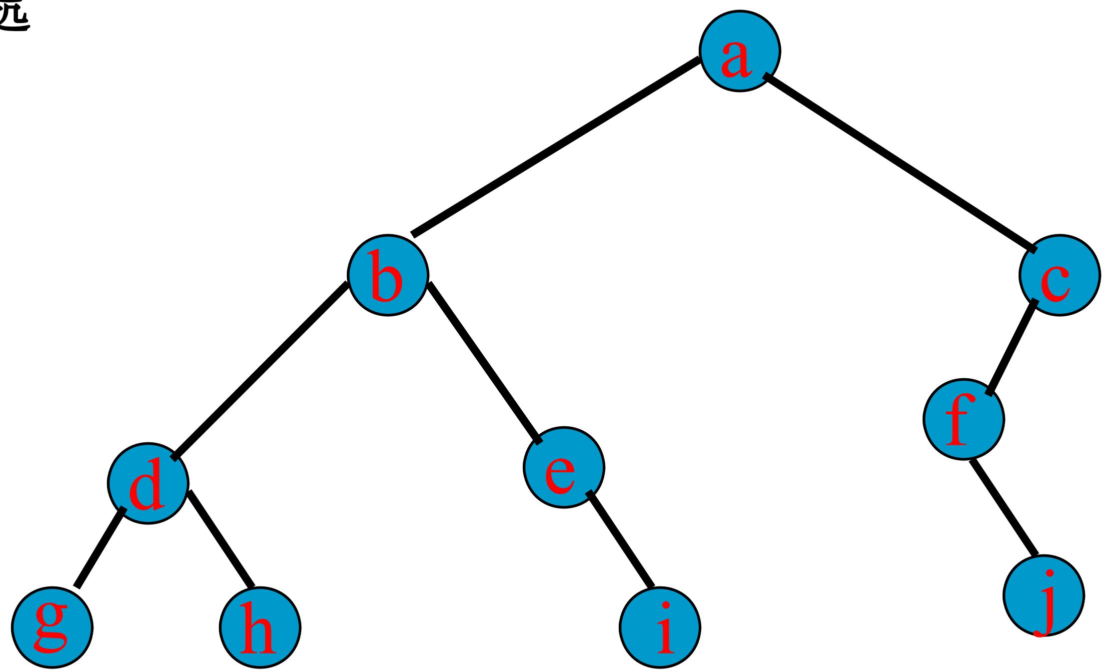
+*+ABC/D-EF

其正為對應的前序運算式。



階層走訪

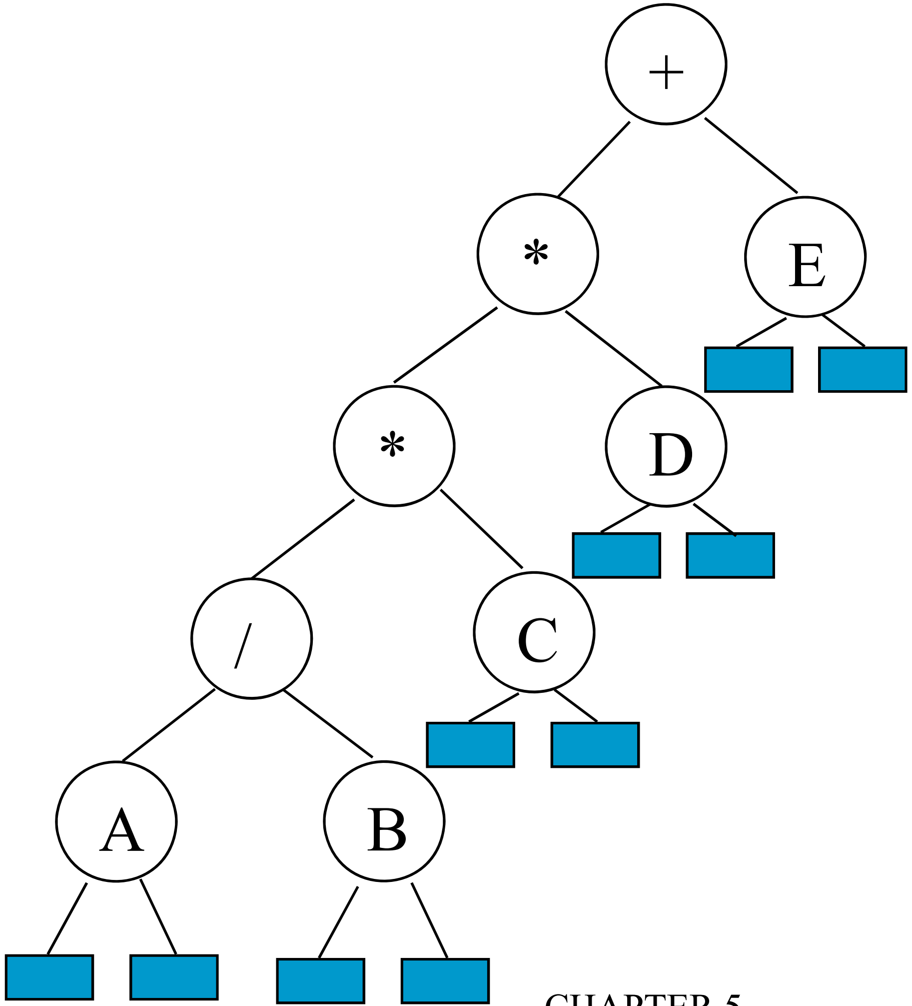
- 「階層走訪」 **(level-order traversal)** 是依階層的順序，進行二元樹的走訪，先走訪階層小的節點，後走訪階層大的節點，同一階層者則依自左向右的順序走訪。對下圖的二元樹，進行階層走訪的結果為：**ABCDEFGHI**。
- 對同一階層而言，先走訪的節點，其子節點亦在下一階層中先被走訪。這種「先進先出」的特性，恰可用佇列予以儲存與處



Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression

preorder traversal
+ * * / A B C D E
prefix expression

postorder traversal
A B / C * D * E +
postfix expression

level order traversal
+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        indorder(ptr->right_child);
    }
}
```

$\boxed{A / B * C * D + E}$

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf(“%d”, ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        printf(“%d”, ptr->data);
    }
}
```

A B / C * D * E +

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(tree_pointer node)
{
  int top= -1; /* initialize stack */
  tree_pointer stack[MAX_STACK_SIZE];
  for (;;) {
   for (; node; node=node->left_child)
     add(&top, node);/* add to stack */
   node= delete(&top);
                    /* delete from stack */
   if (!node) break; /* empty stack */
   printf("%D", node->data);
   node = node->right_child;
 }
}
```

O(n)

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

作業

pp. 210: ex1, ex3

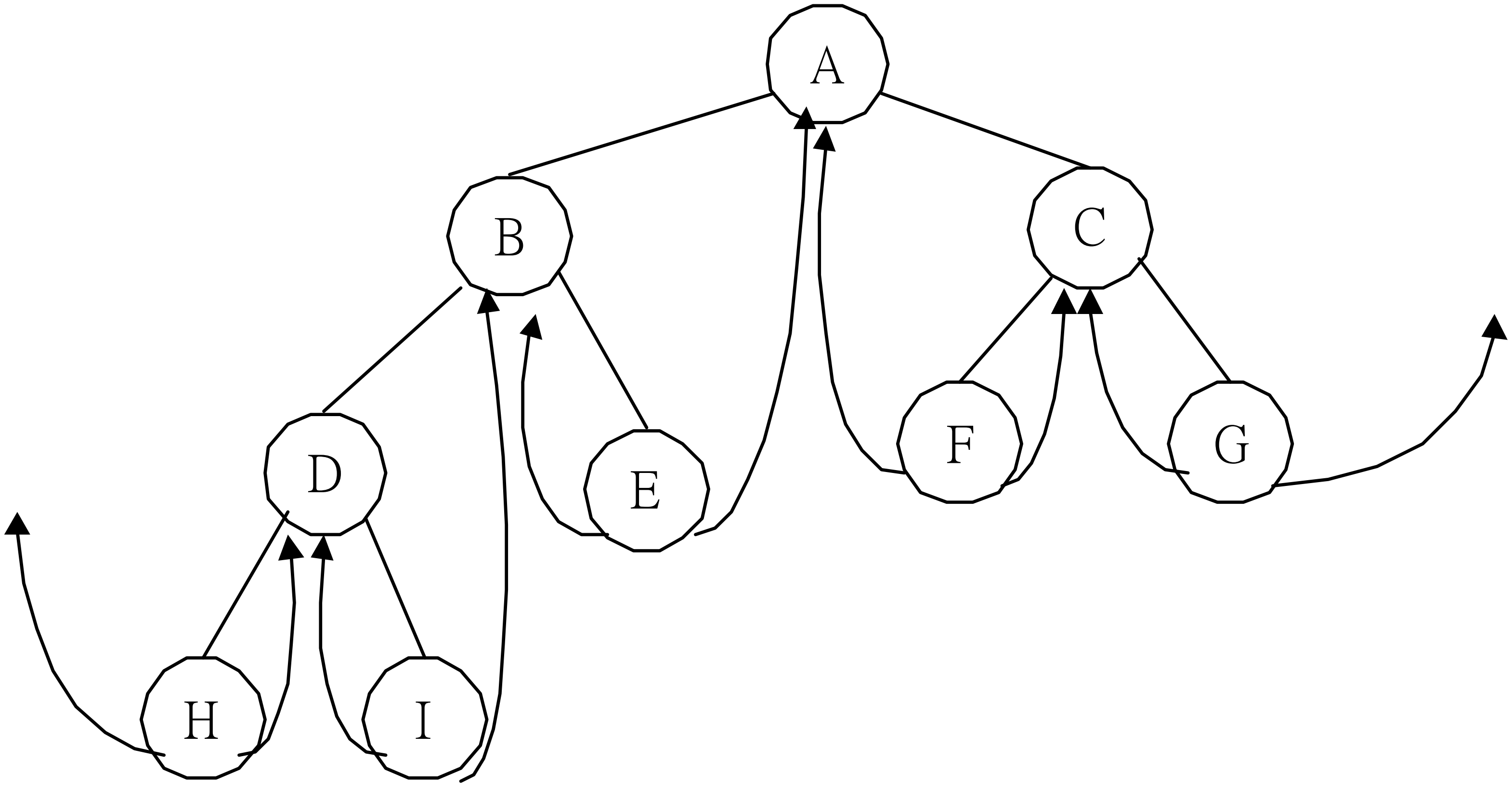
5.5 引線二元樹 (Threaded BT

- 在二元樹的鏈結表示中，樹葉節點的兩個子樹指標皆指向**NULL**；由定理**5-1**得知**n = E+1**，其中**n**為節點個數，**E**為分支數。
- 而**n**個鏈結節點，有**2n**個指標空間，每個分支恰佔用一個指標空間，共有**E (=n-1)** 個指標是用到的（不是空的），而共有**n+1**個指標空間是空指標**(NULL)** ，比非空的節點還多。
- 有學者提出對這些「放空指標的空間」加以利用的概念—與其放空指標，不如放指向其它節點的指標，稱之為引線 **(thread)**，使得某些運算（如：走訪、**…**等）可以加快。
- 這個概念發展出了引線二元樹 **(threaded binary tree)** 這種資料結構。

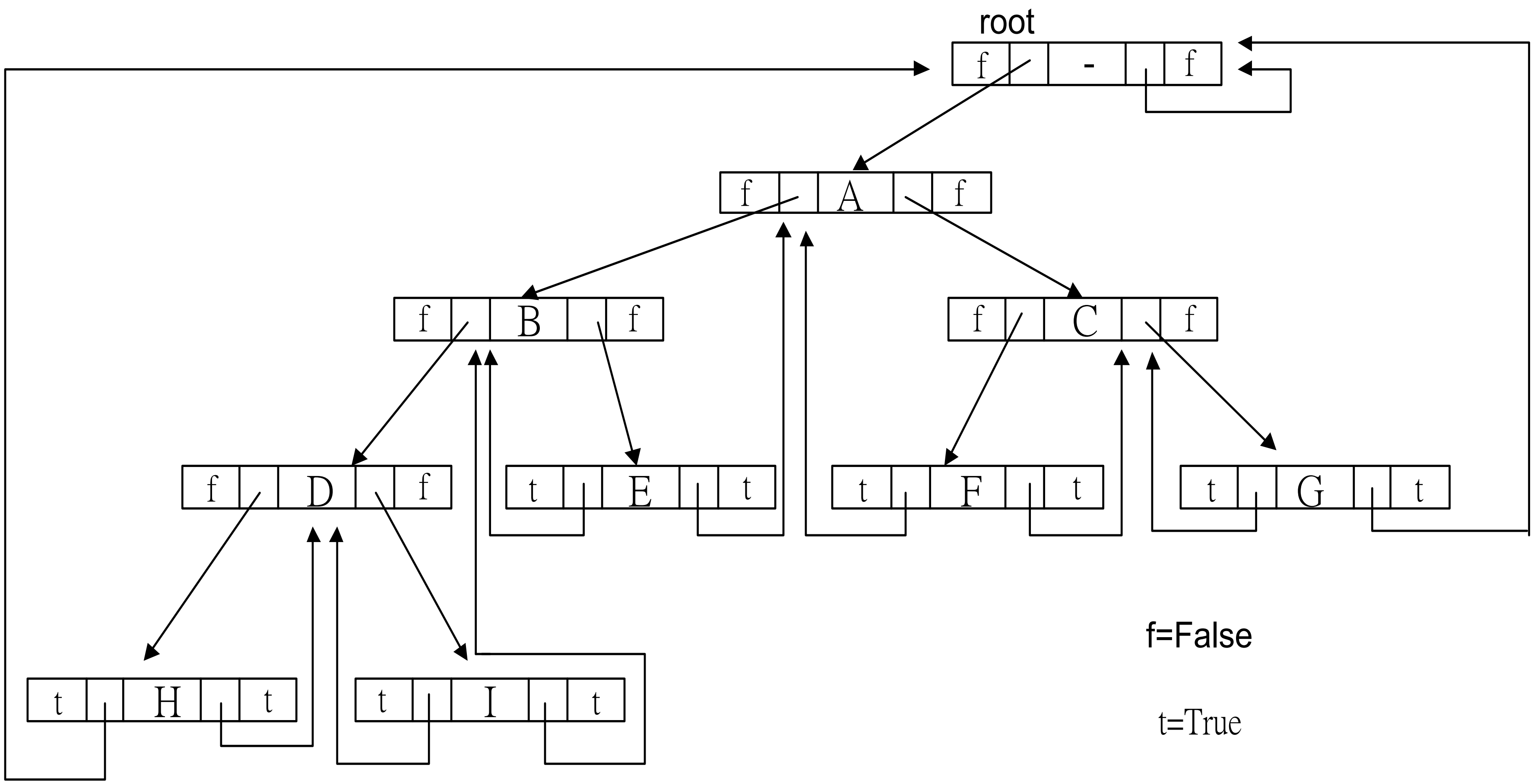
引線二元樹*

- 如果節點V的左鏈結指向NULL時，將這個鏈結改指向一個節點，被指到的節點為節點V的中序前行者
- 如果節點V的右鏈結指向NULL時，將這個鏈結取代成指向一個節點，被指到的節點為節點V的中序後繼者

引線二元樹*



引線二元樹*



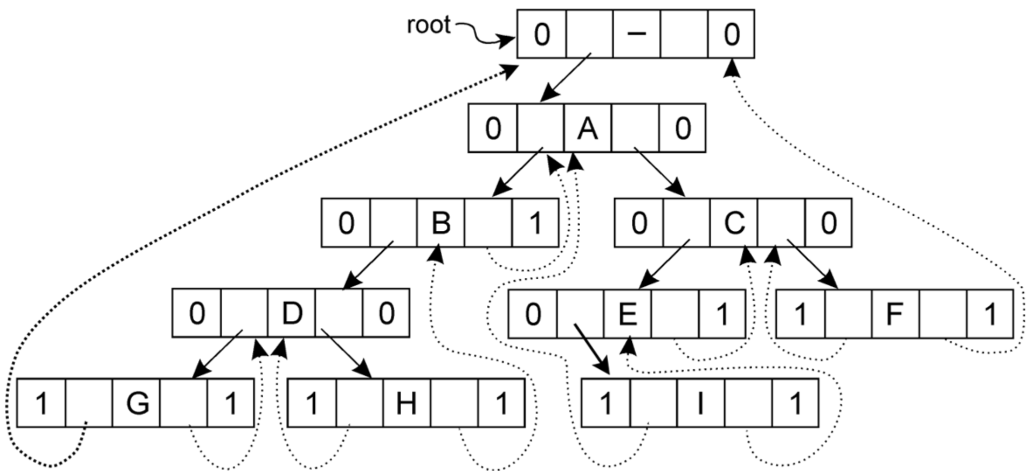
範例

為了區別節點指標放的是一般指標（內部節點）、抑或是引線指標（樹葉節點），我們另以**2**個欄位分別區別左和右子樹指標空間存放的對象。其節點的記憶體配置有如下圖所示。

lefthread	leftchild	data	rightchild	righthread

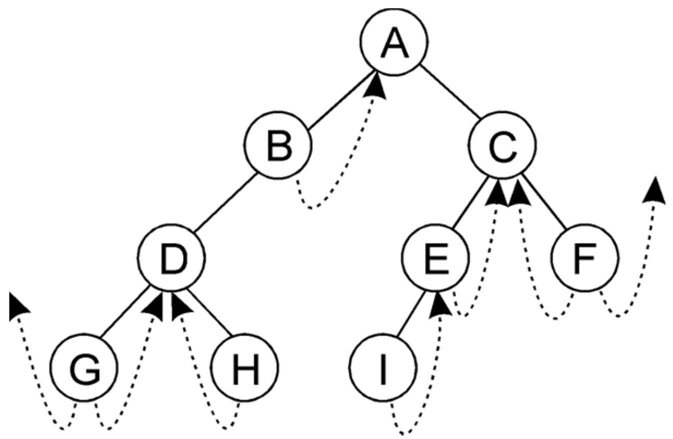
其中**leftthread (rightthread)** 為**0**時，表示**leftchild (rightchild)** 存放的是一般節點指標；**leftthread (rightthread)** 為**1**時，表示**leftchild (rightchild)** 存放的是引線指標。

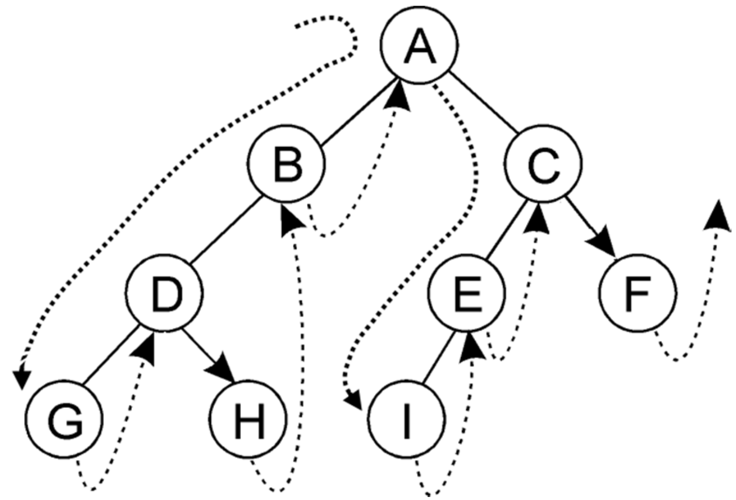
由上頁圖的引線二元樹可知，有最左和最右兩條引線尚無妥善的安排，我們另設計一空的引線節點做為樹根，那麼範例**5-17**的引線二元樹，將如下圖所示。



範例

下圖為加了引線的二元樹。





- 這些加入的引線將使二元樹的中序走訪更加便利。
- 沿著上右圖箭頭所指示的順序走訪，即可完成此二元樹的中序走訪；
- **GDHBAIECF**為此二元樹的中序表示。

引線二元樹的中序走訪

茲將引線二元樹所需節點的宣告詳列於程式**5-10**中，並定義決定節點**p**在中序表示中的後繼節點**q**（簡稱：中序後繼點）的程序：

程式　決定節點**node**的中序後繼點

```
struct TBTreeNode
{   int   leftthread;
    struct TBTreeNode *leftchild;
    char data;
    struct TBTreeNode *rightchild;
    int rightthread;
};
struct TBTreeNode *root;
struct TBTreeNode *Next(struct TBTreeNode *node)
{   struct TBTreeNode *temp;
    temp = node->rightchild;
    if (node->rightthead) return temp;
    while (!temp->leftthead) temp = temp->leftchild;
          return temp;
};
```

引線二元樹的中序走訪(續)

於是要得到引線二元樹的中序表示，只須從中序表示中的第一節點起，讓每一節點**node**，都執行**Next(node)**程序，即可得：

程式 引線二元樹的中序表示（上接程式**5-10**）

```
void InorderTBTree()
{  struct TBTreeNode *node;
   for(node = Next(root);
   node != root ; node = Next(node))
       cout << node->data;
}
```

在引線二元樹中加入節點

在引線二元樹中，樹葉節點節點的右子樹指標所指向、或內部節點右子樹的最左樹葉，為其中序後繼節點；而且任一節點若無左子樹，則其左子樹指標乃指向其中序前接節點...；這些性質必須在節點新增進入引線二元樹時，也要保持。

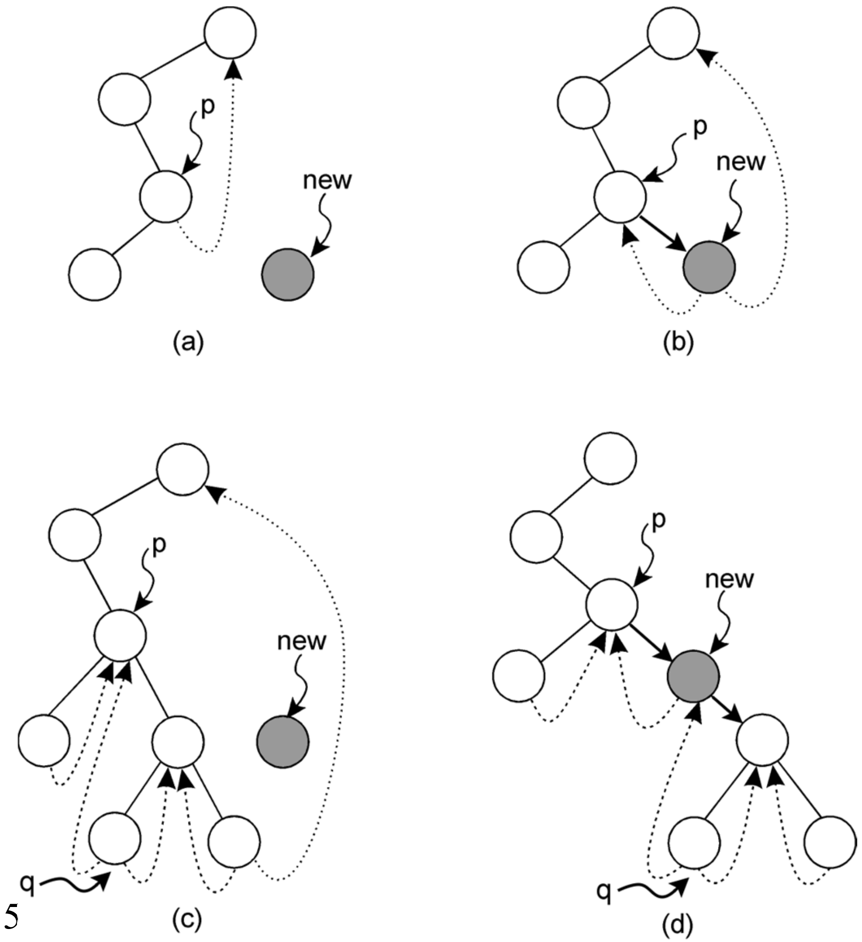
在本節中我們討論節點***new***插入成為引線二元樹的節點**p**的右子節點的情形；至於插入成為左子節點的情形，則讓各位模擬推敲。

在引線二元樹中加入節點(續)

new會成為**p**的右子節點的情況，共有以下兩種：

(1) 若p沒有右子節點 (`p->rightthread`為1) ，如圖(a) ，則***new***插入後應形成如圖(b) 所示。

(2) 若p有右子節點 (`p->rightthread`為0) ，如圖(c) ，則***new***插入後應形成如圖(d) 所示。



插入節點new成為引線二元樹中節點p的右子節點

```
void InsertRight(struct TBTreeNode *p,
                 struct TBTreeNode *new)
{   struct TBTreeNode *q;
    new->rightchild = p->rightchild;
    new->rightthead = p->rightthread;
    new->leftchild = p;
    new->leftthread = 1;
    p->rightchild = new;
    p->rightthead = 0;
         if (!new->rightthread)
         {    q = Next(new);
              q->leftchild = new;
         }
    }
```

Threaded Binary Trees

- Two many null pointers in current representation of binary trees

  n: number of nodes

  number of non-null links: n-1

  total links: 2n

  null links: 2n-(n-1)=n+1

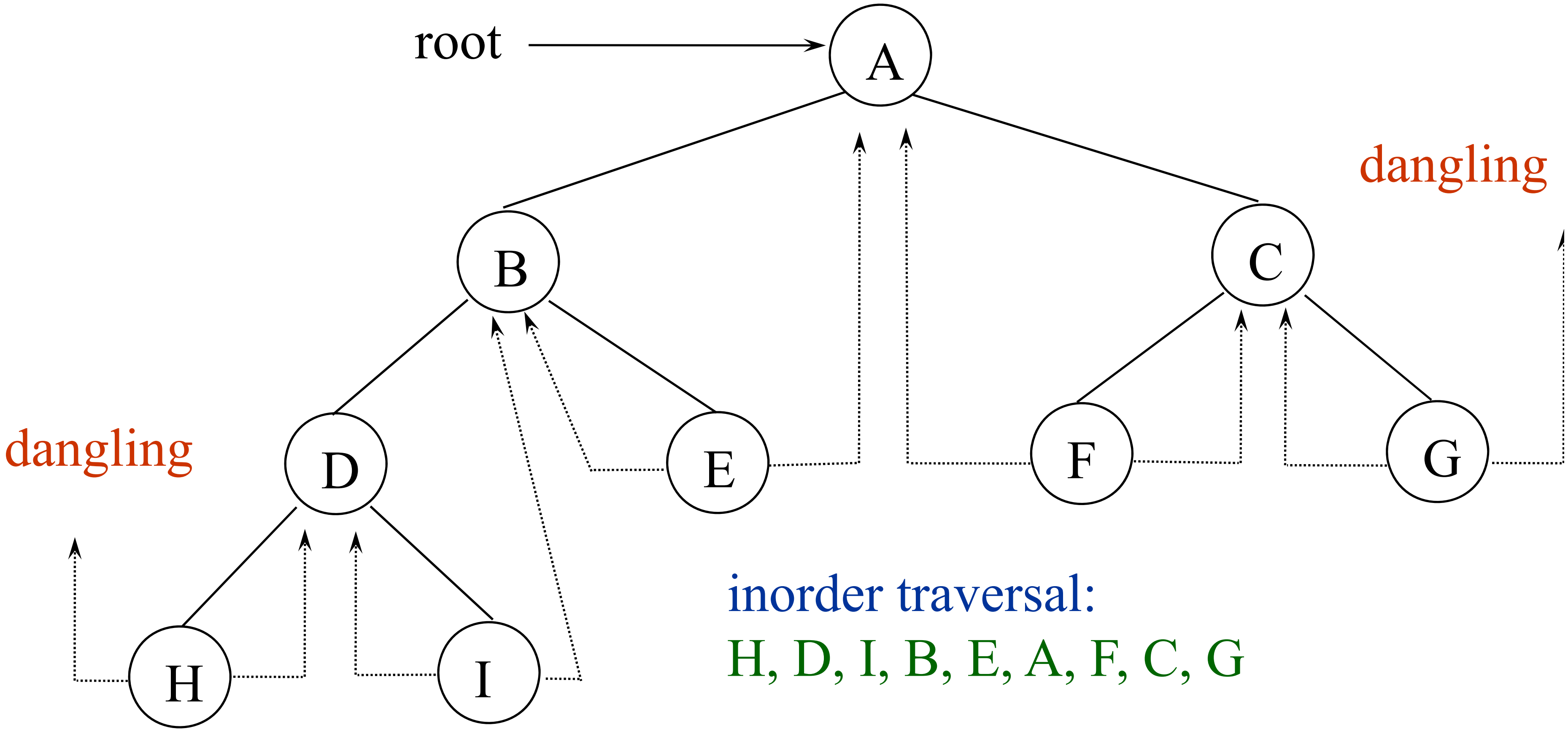
- Replace these null pointers with some useful "threads".

Threaded Binary Trees (*Continued*)

If `ptr->left_child` is null,
replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*

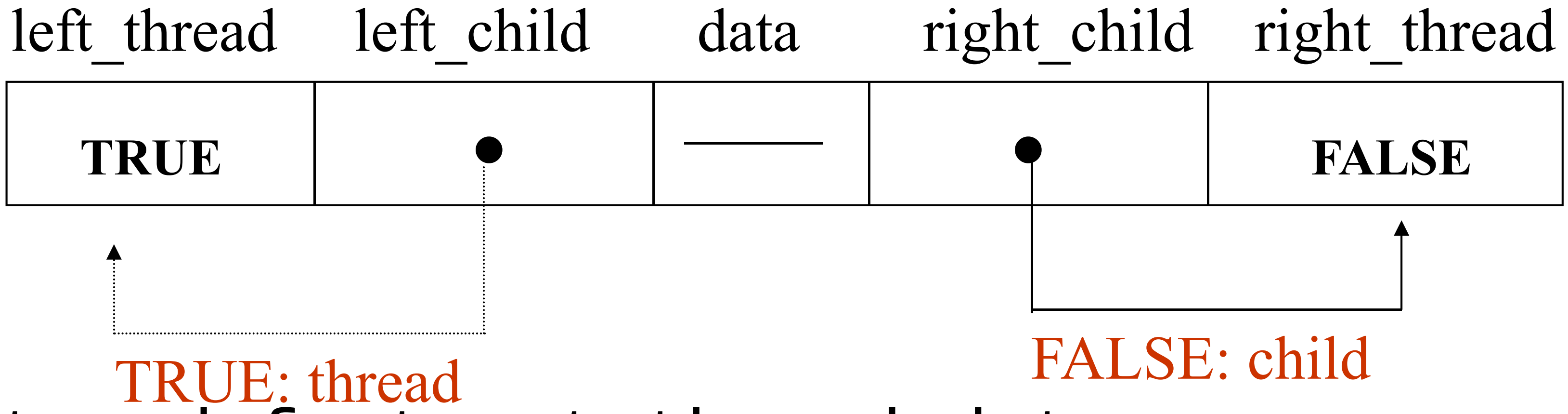
If `ptr->right_child` is null,
replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

A Threaded Binary Tree



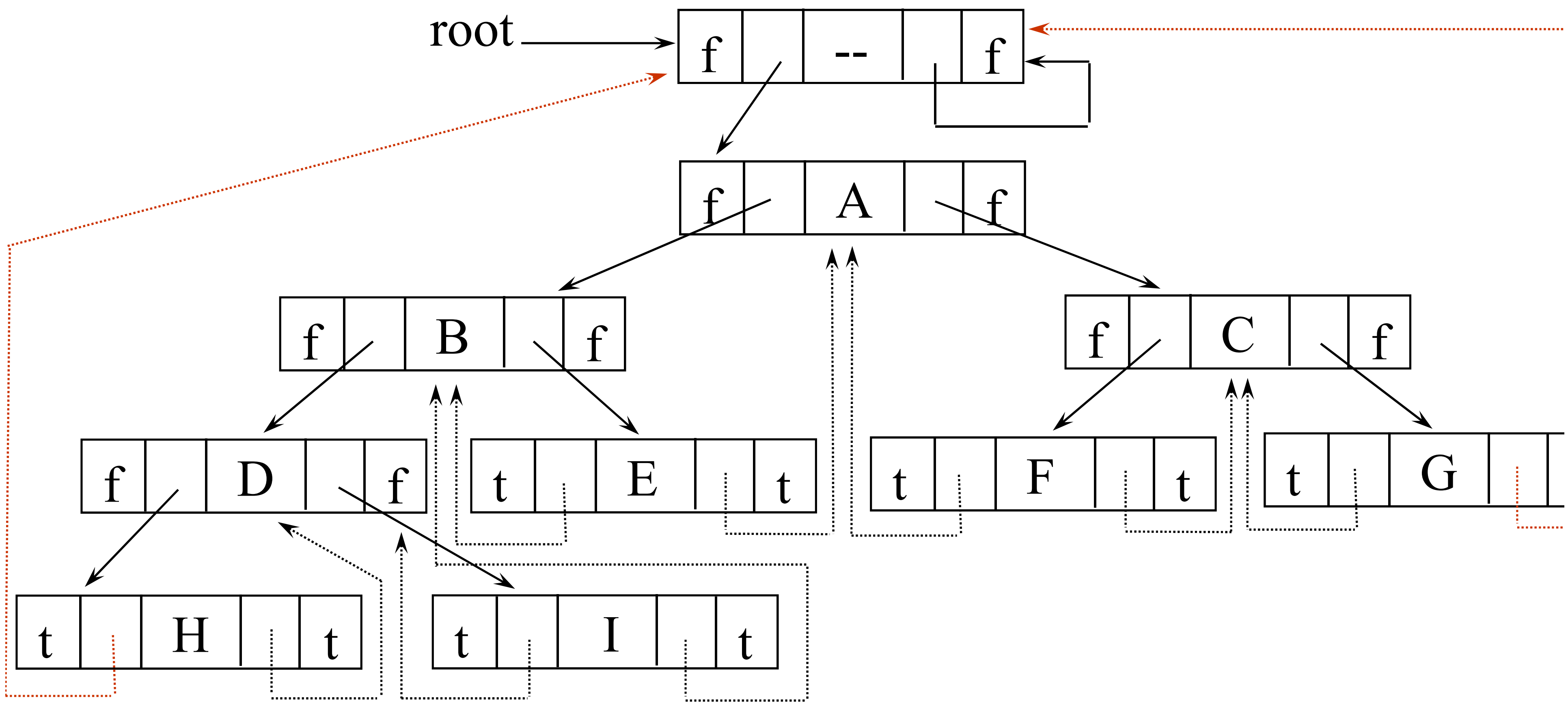
inorder traversal:
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



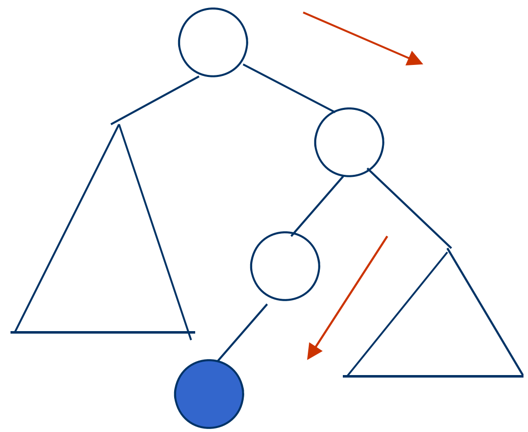

```
typedef struct threaded_tree
  *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer
  tree)
{
  threaded_pointer temp;
  temp = tree->right_child;
  if (!tree->right_thread)
    while (!temp->left_thread)
      temp = temp->left_child;
  return temp;
}
```



Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
/* traverse the threaded binary tree
  inorder */
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp==tree) break;
        printf(“%3c”, temp->data);
    }
}
```

O(n)

Inserting Nodes into Threaded BTs

- Insert `child` as the right child of node `parent`
  - change `parent->right_thread` to FALSE
  - set `child->left_thread` and `child->right_thread` to TRUE
  - set `child->left_child` to point to `parent`
  - set `child->right_child` to `parent->right_child`
  - change `parent->right_child` to point to `child`

Examples

Insert a node D as a right child of B.

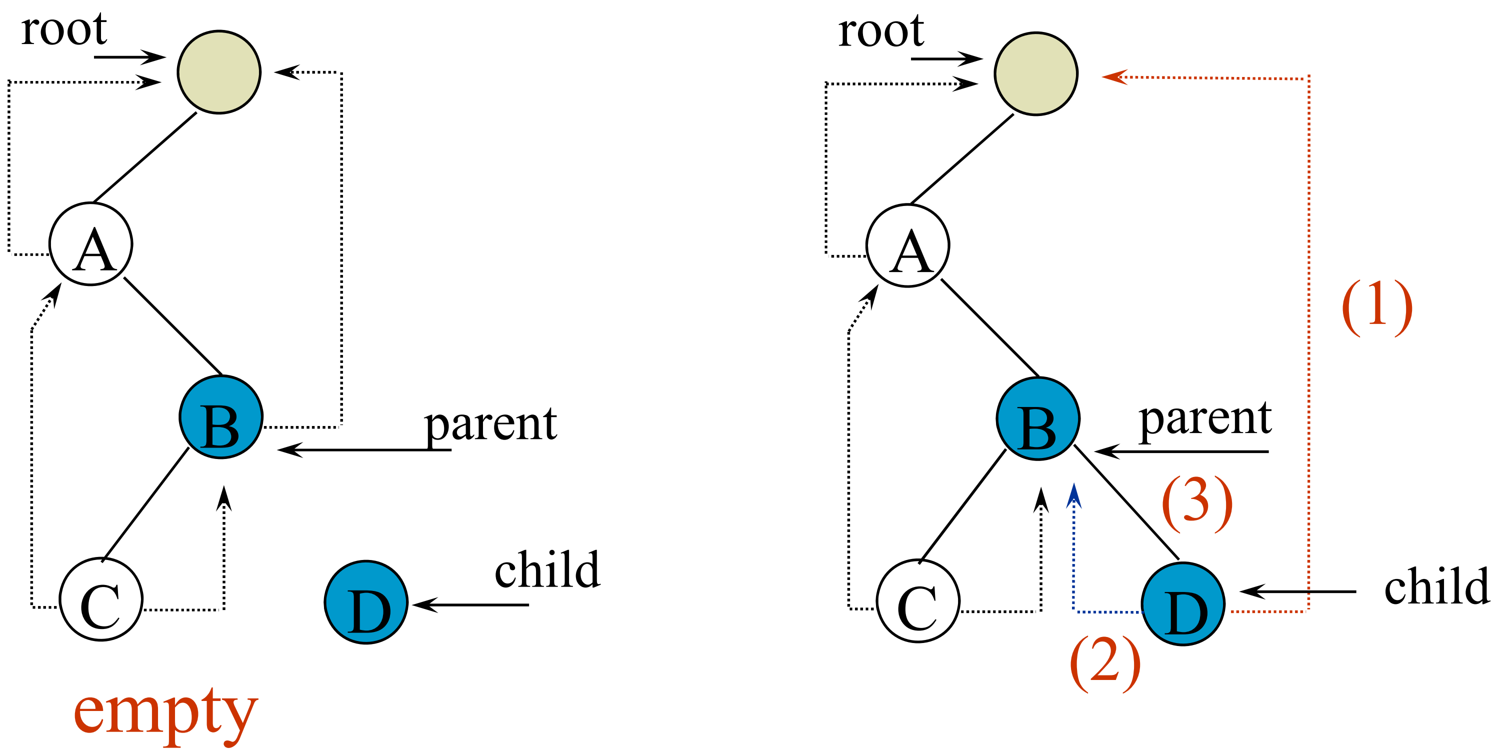
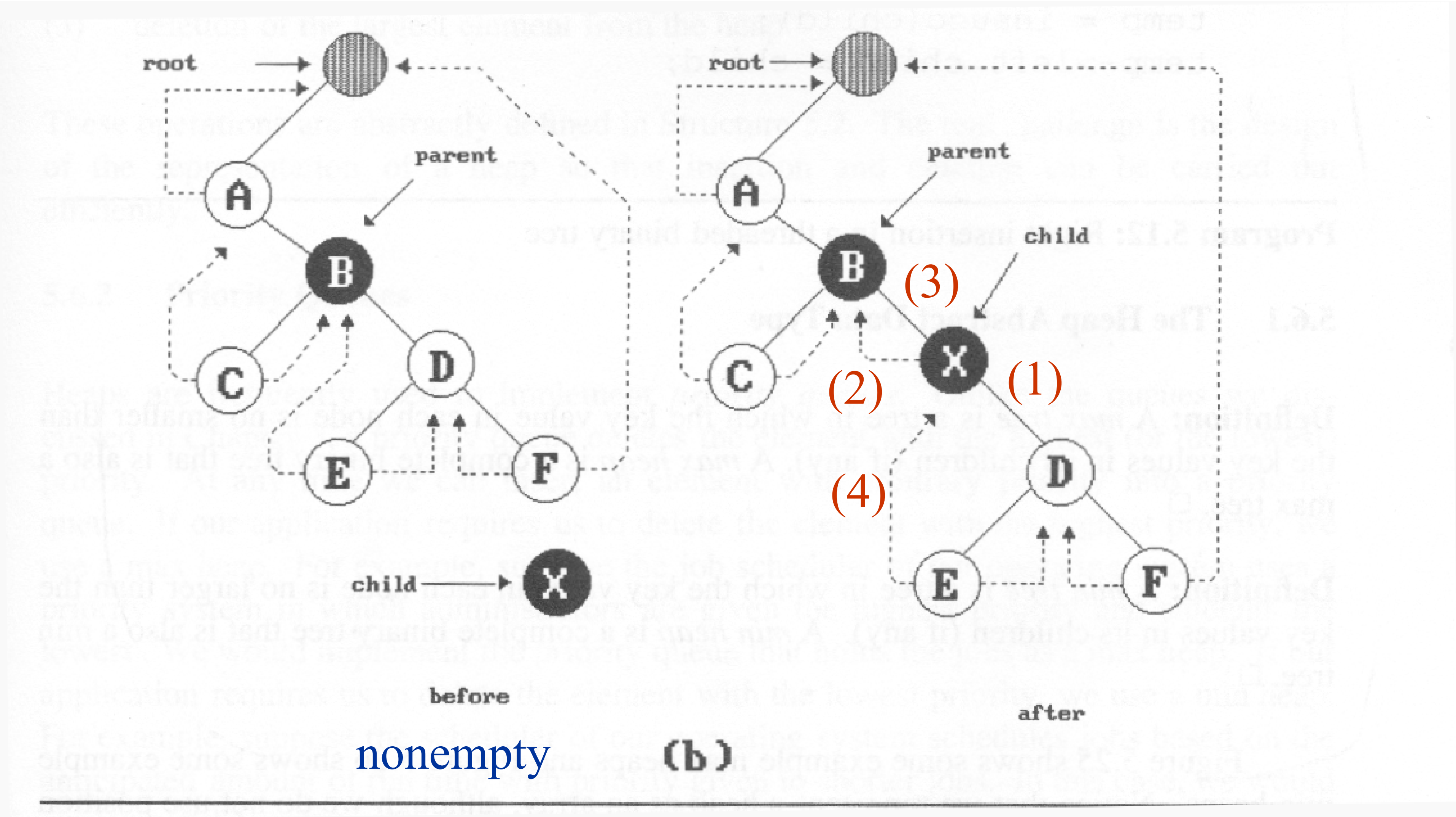


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.222)



Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
  threaded_pointer temp;
(1)child->right_child = parent->right_child;
  child->right_thread = parent->right_thread;
(2)child->left_child = parent;  case (a)
  child->left_thread = TRUE;
(3)parent->right_child = child;
  parent->right_thread = FALSE;
  if (!child->right_thread) { case (b)
 (4)temp = insucc(child);
    temp->left_child = child;
  }
}
```

作業

pp. 221: ex1

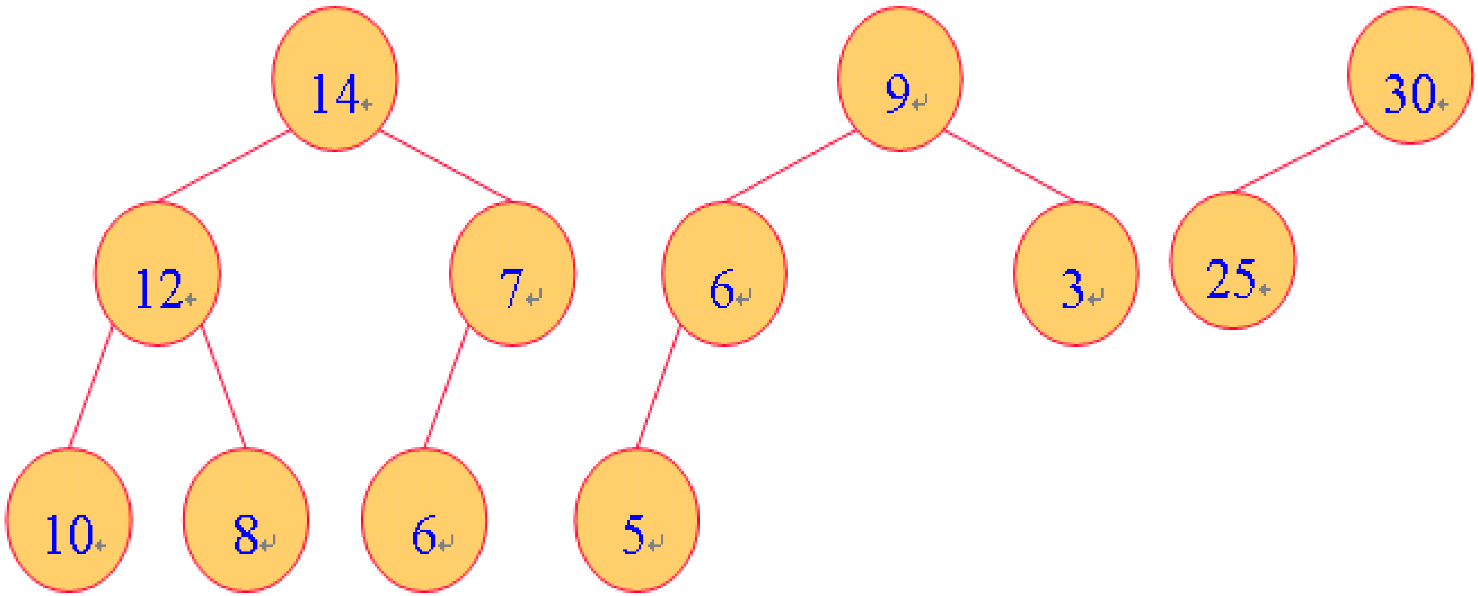
5.6 堆積

（Heaps）

定義

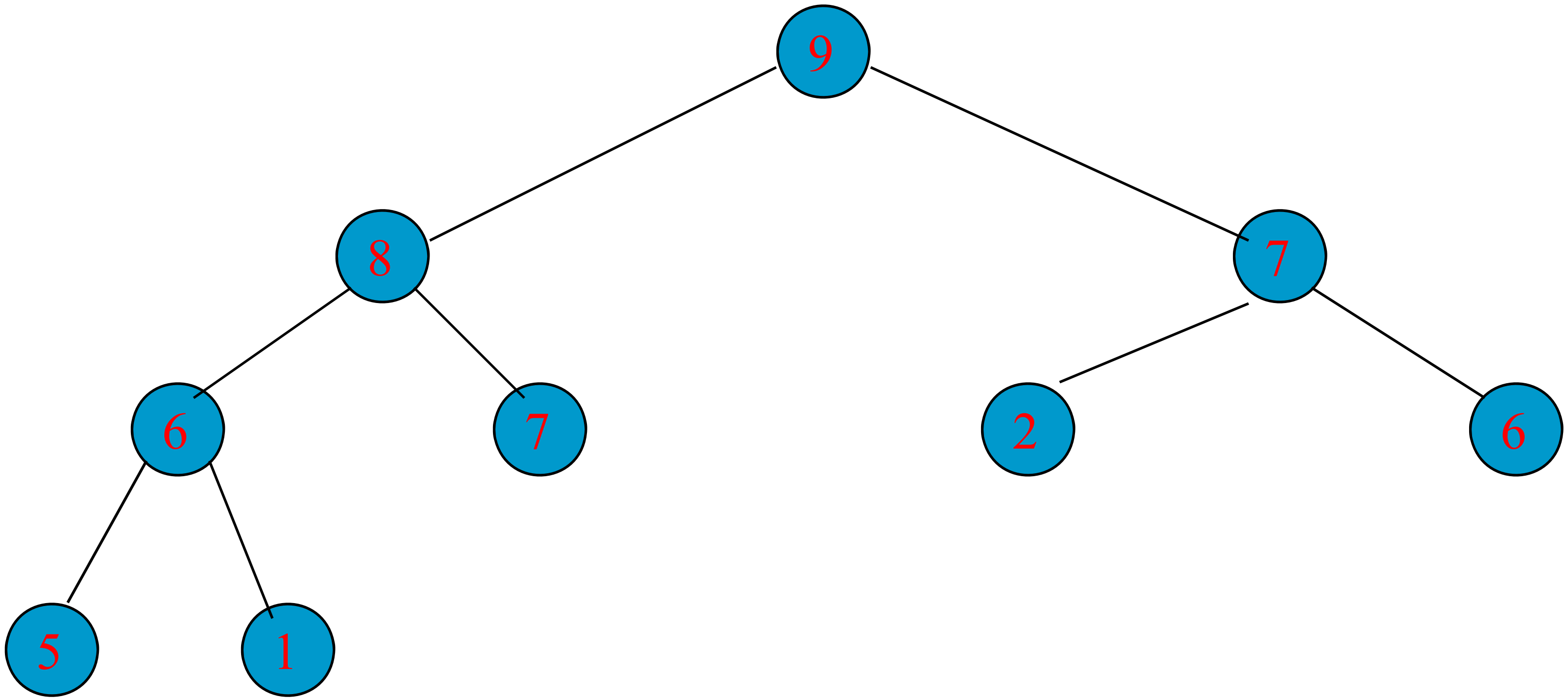
- 堆積是一棵二元樹，而且必須符合完整二元樹。
- 堆積可細分為max-heap、min-heap等

- 最大樹(A *max tree*)
是樹的一種，每個節點的鍵值都不小於其子節點的的鍵值。
- 最大堆積(A *max heap*)
是最大樹的完整二元樹(a complete binary tree)。



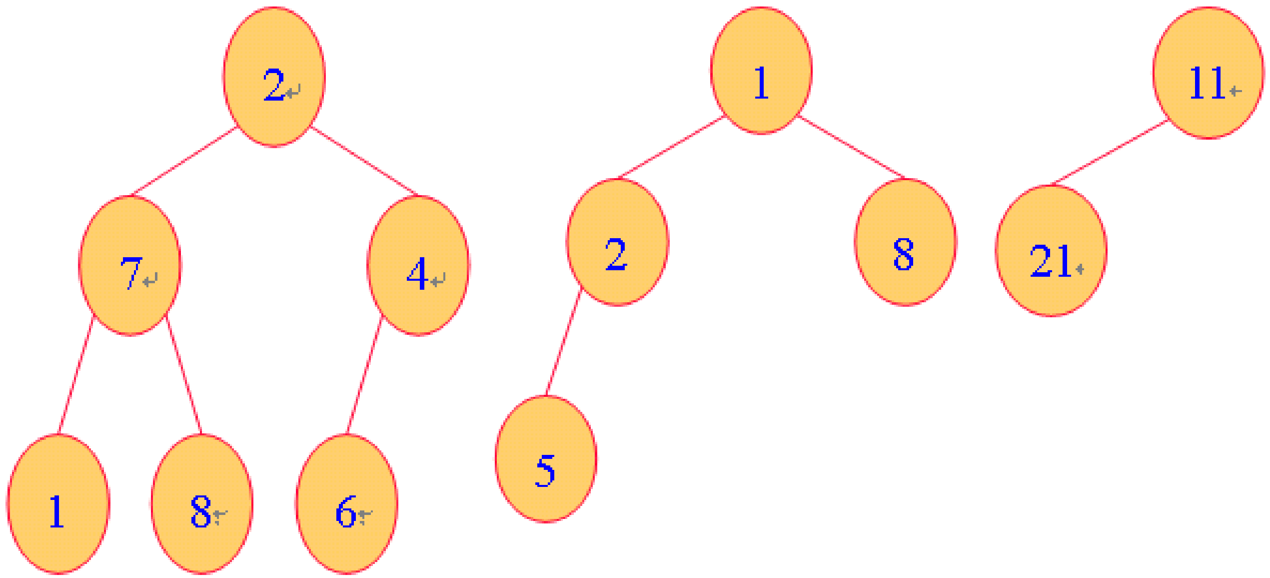
max-heap tree

最大堆積



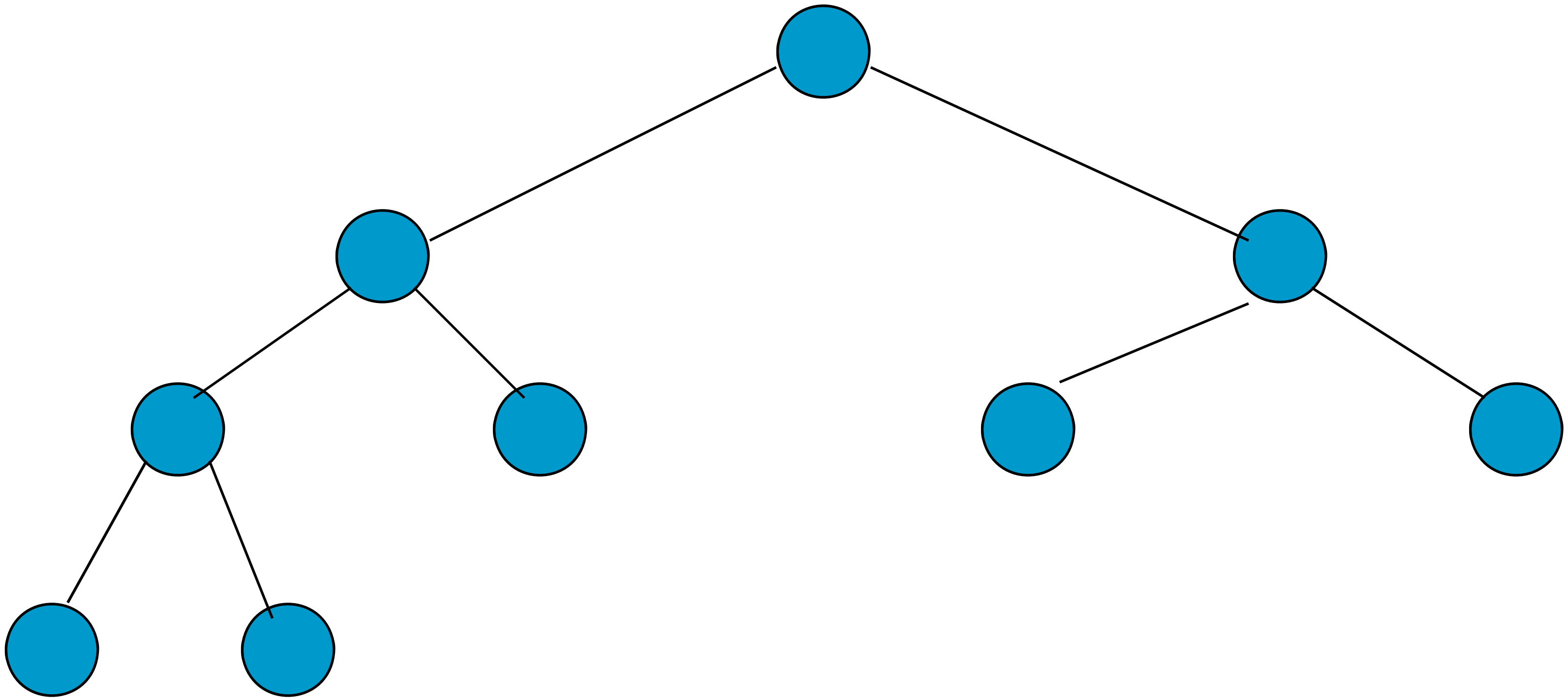
也是最大堆積的9 個節點的完整二元樹

- 最小樹(A *min tree*)
  是樹的一種，每個節點的鍵值都不大於其子節點的的鍵值。
- 最小堆積(A *min heap*)
  是最小樹的完整二元樹(a complete binary tree)。



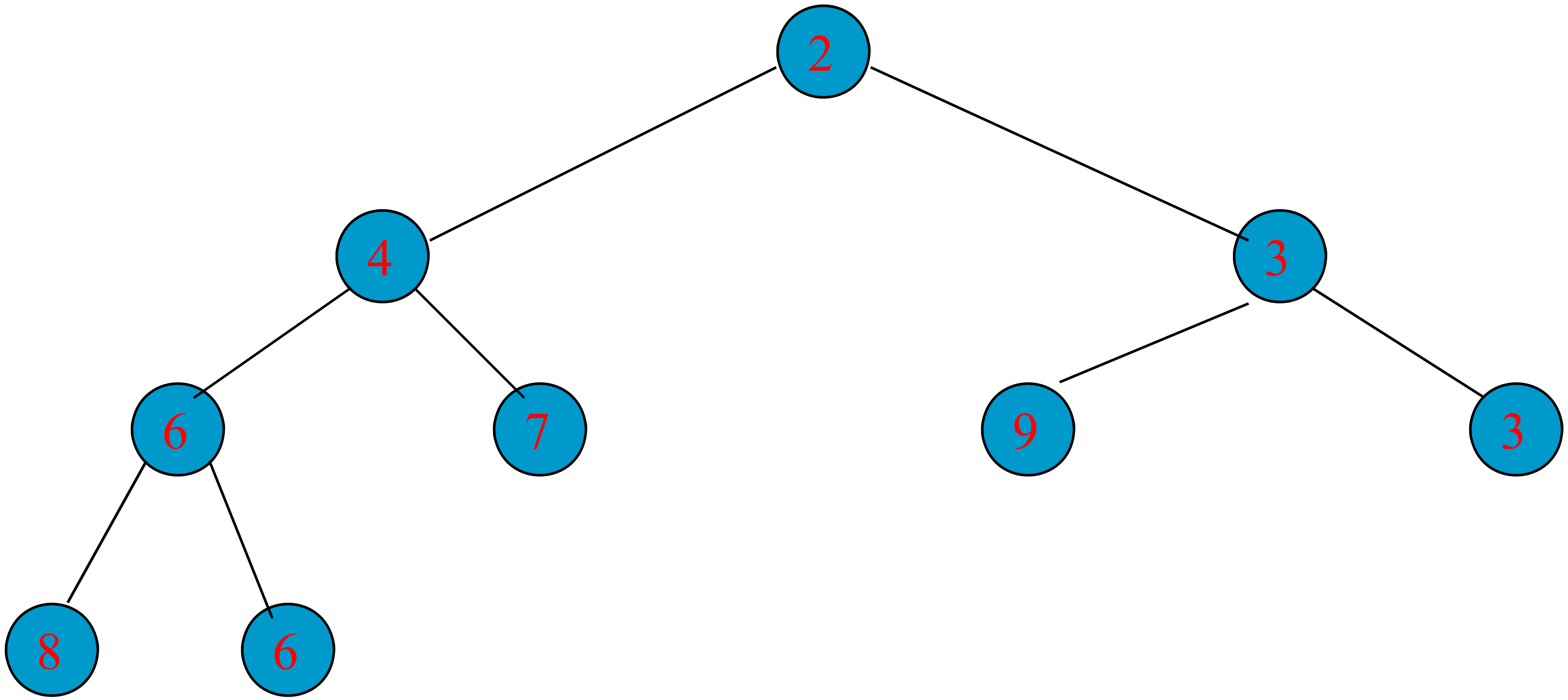
min-heap tree

最小堆積



9 個節點的完整二元樹

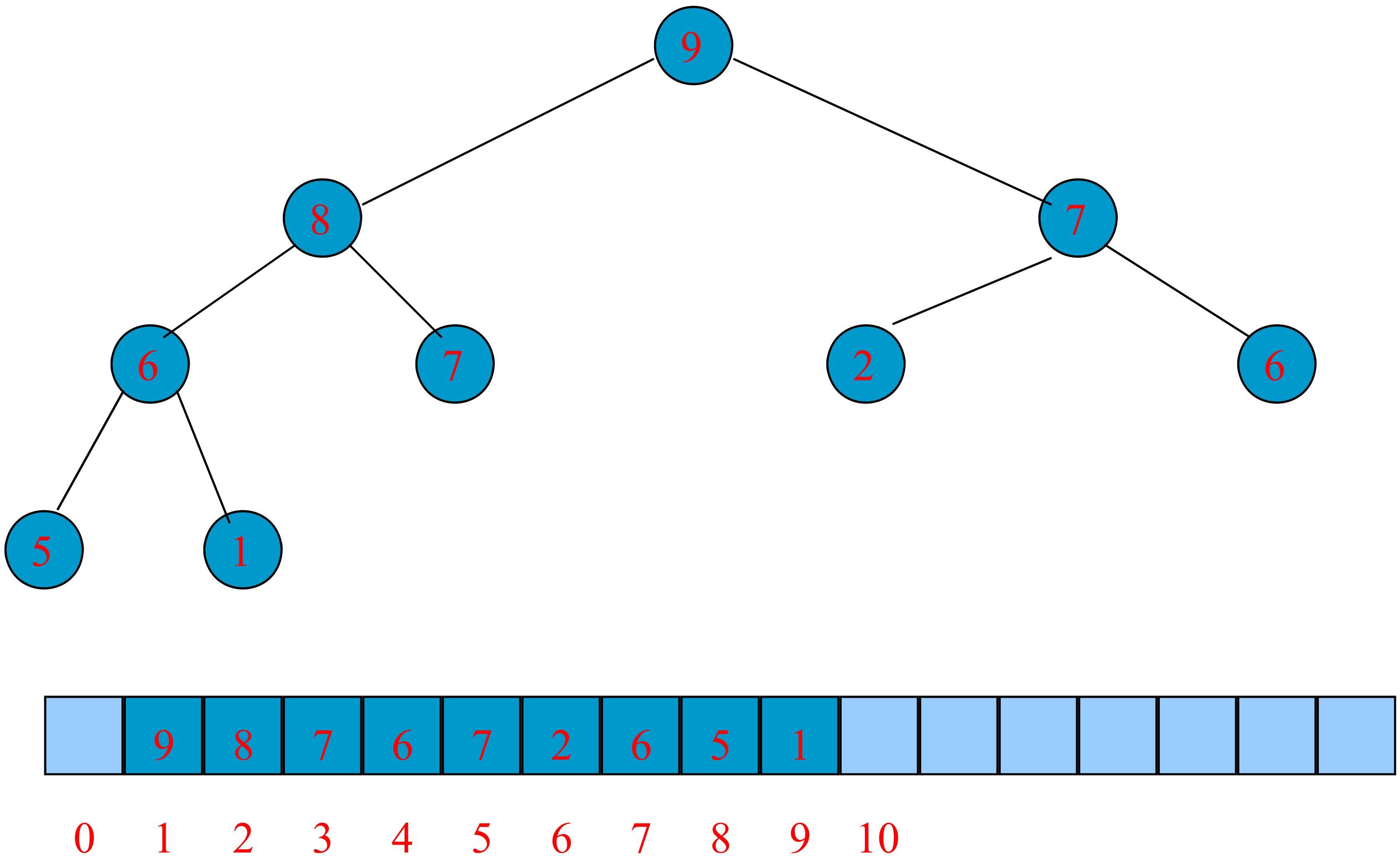
最小堆積



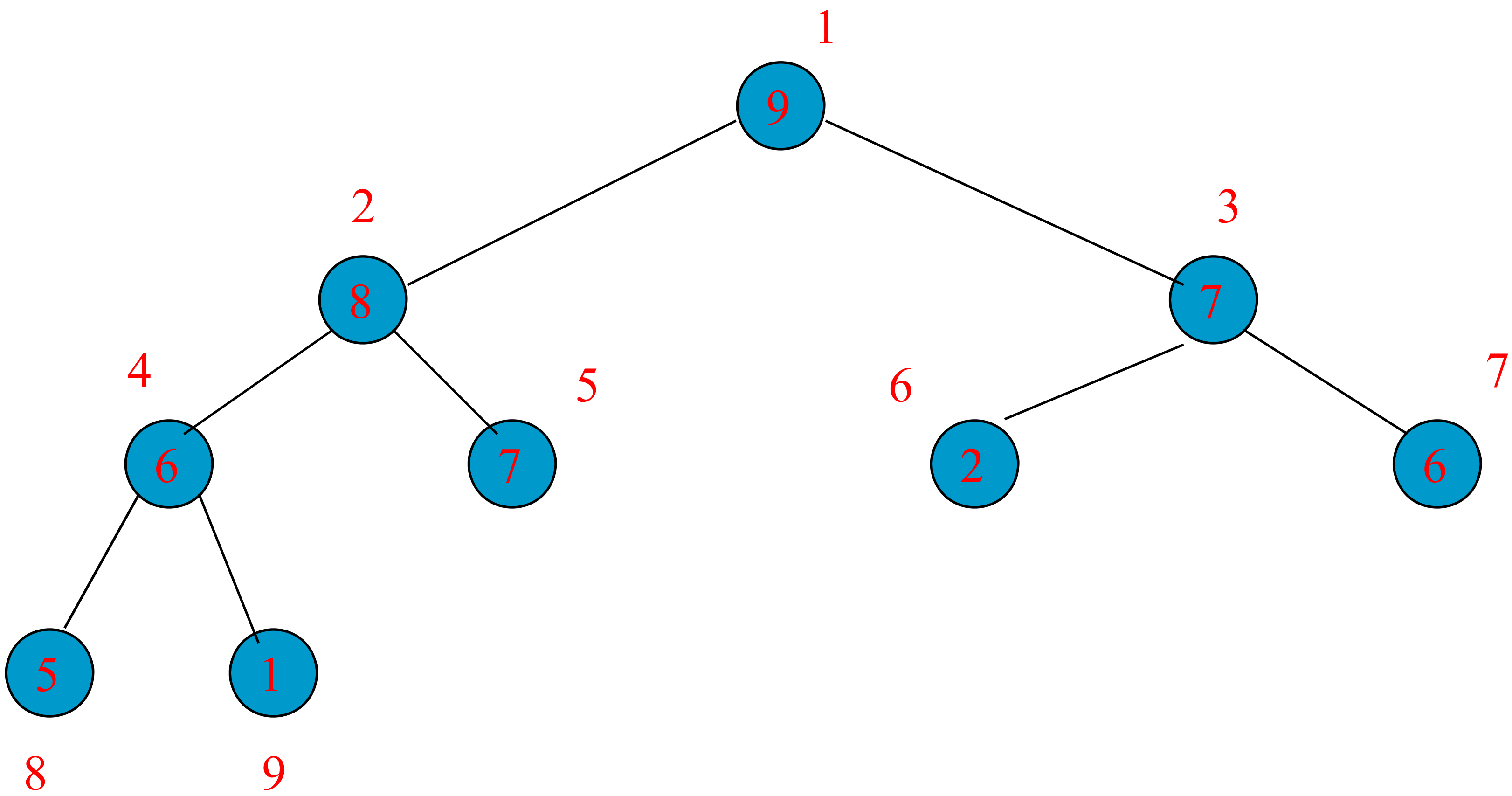
也是最小堆積的9 個節點的完整二元樹

基本運算及表示法

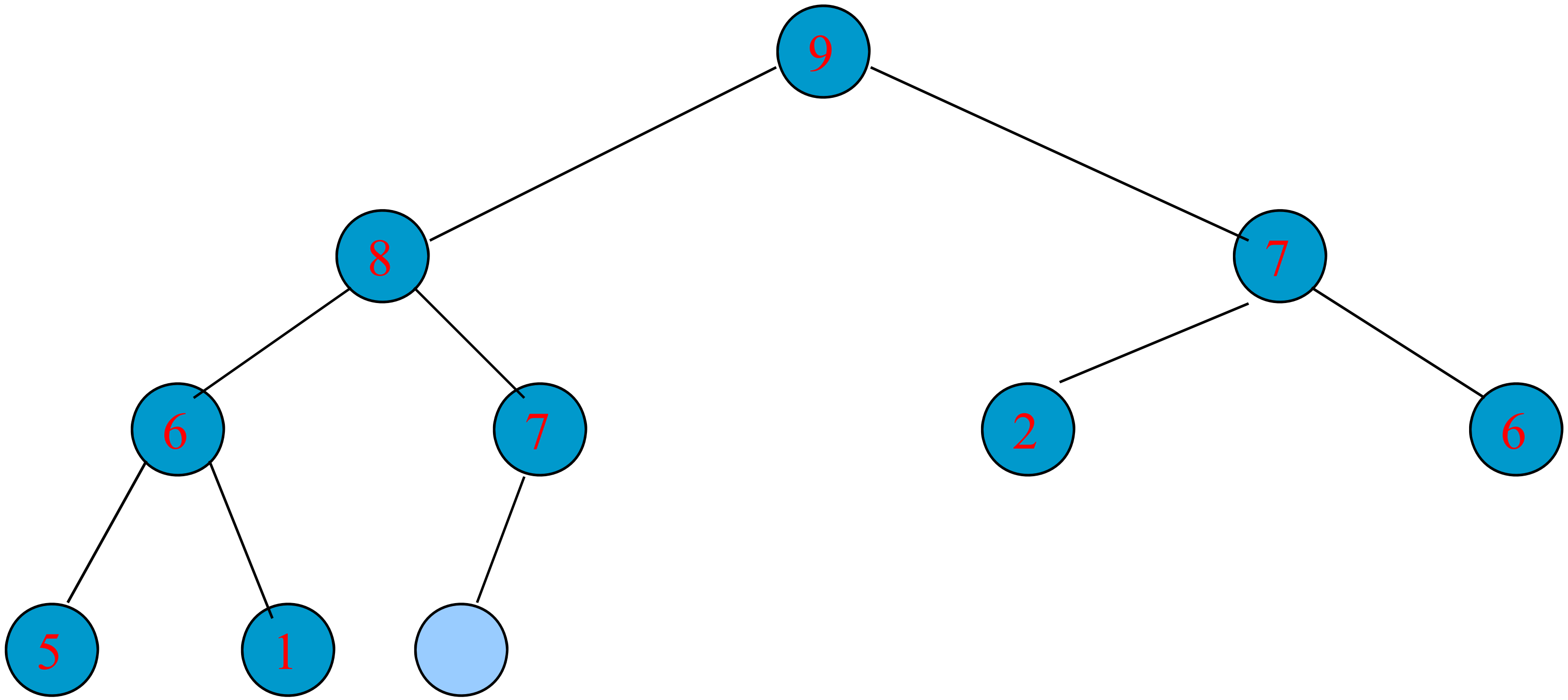
陣列可以有效地表示堆積



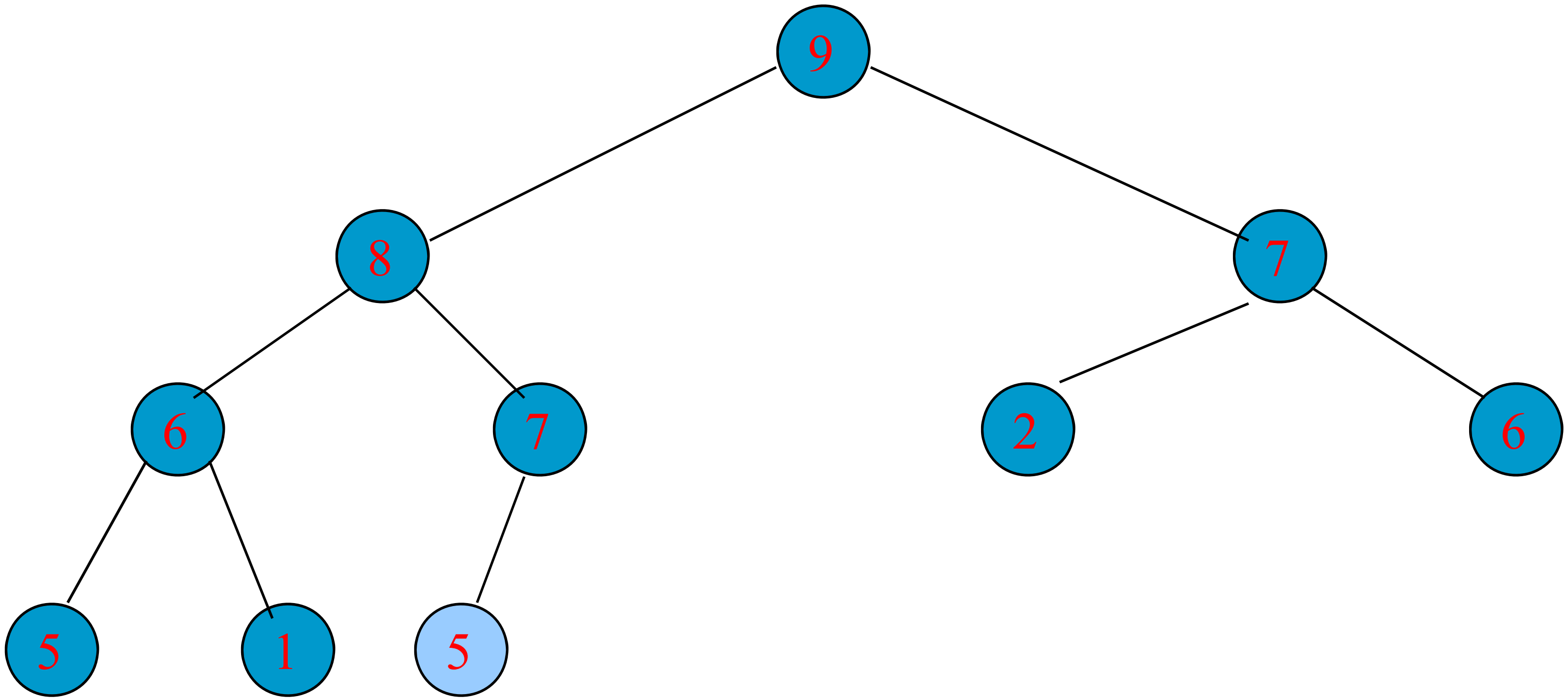
堆積之向上下移動



插入新的元素到堆積

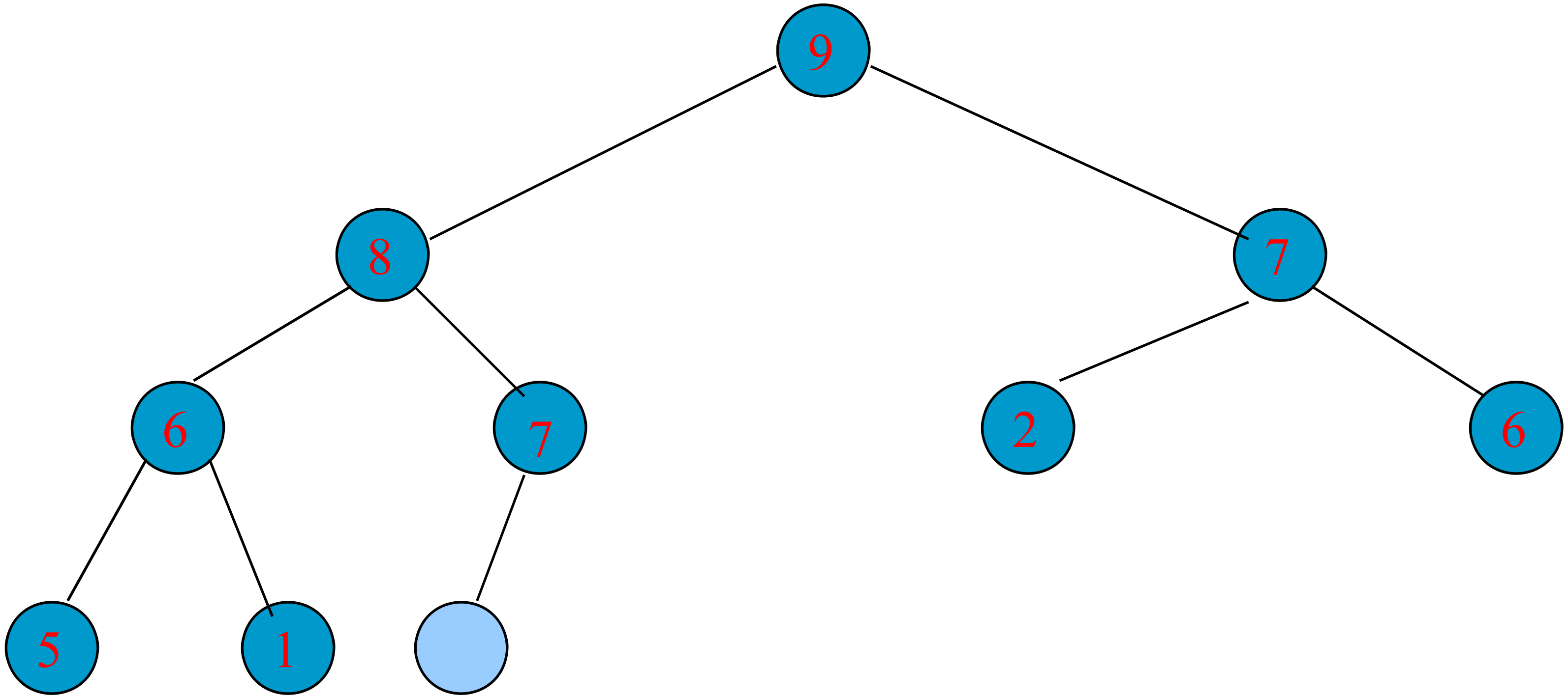


插入新的元素到堆積



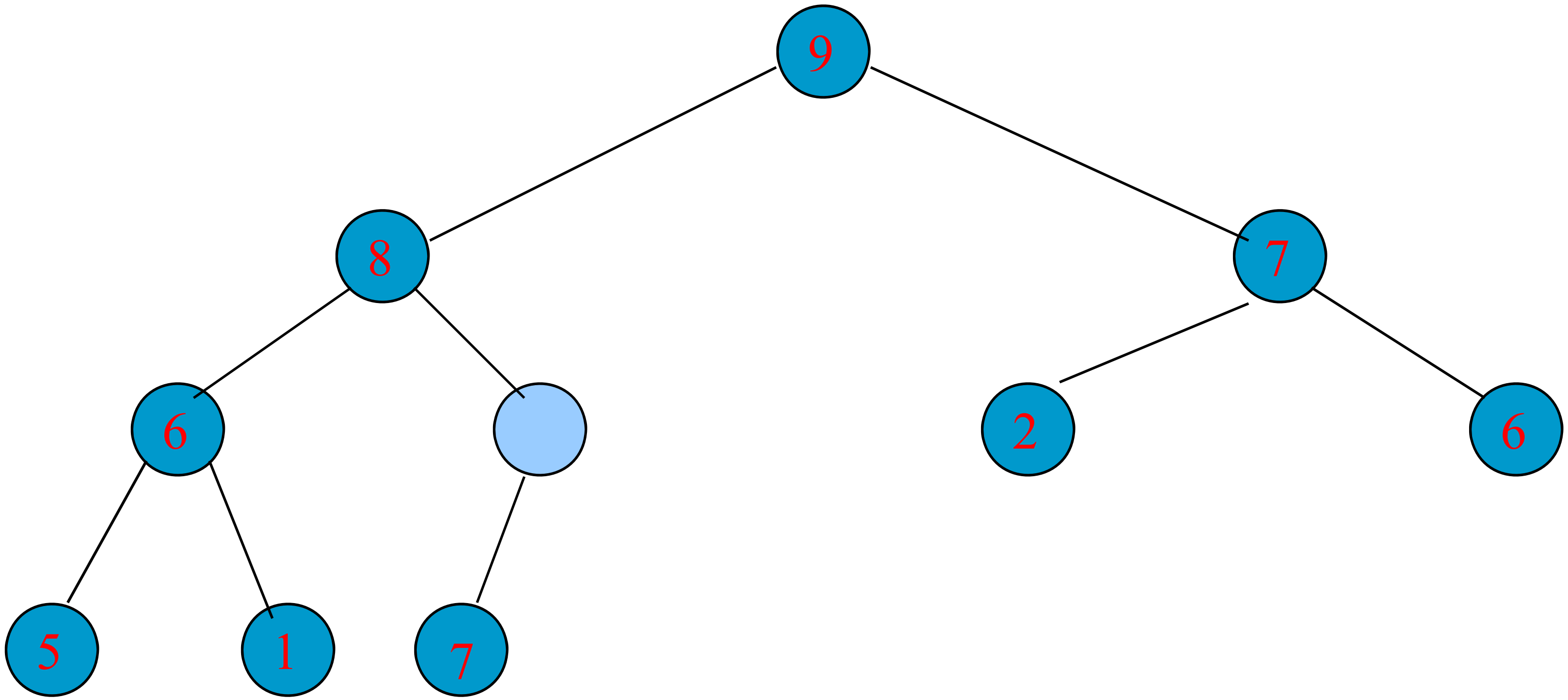
加入 **5**

插入新的元素到堆積



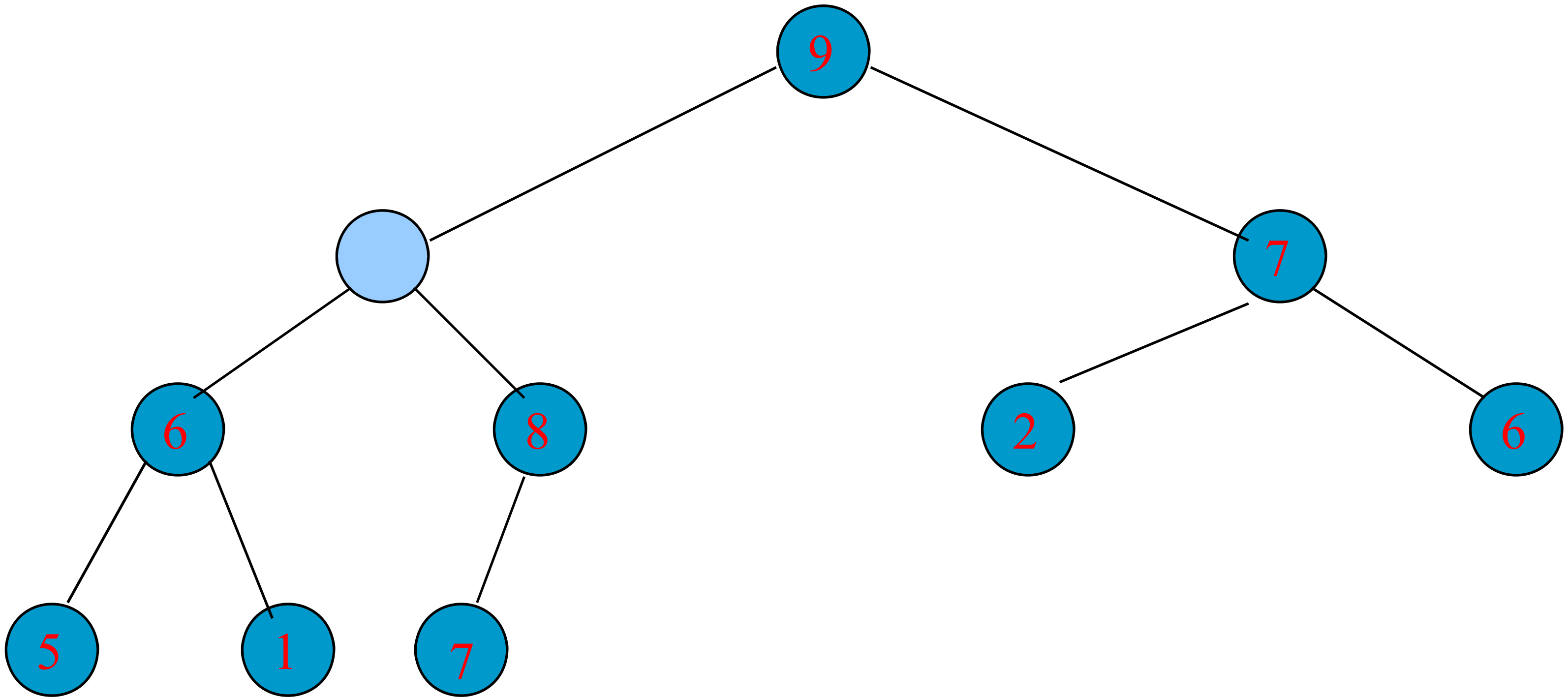
加入 20

插入新的元素到堆積



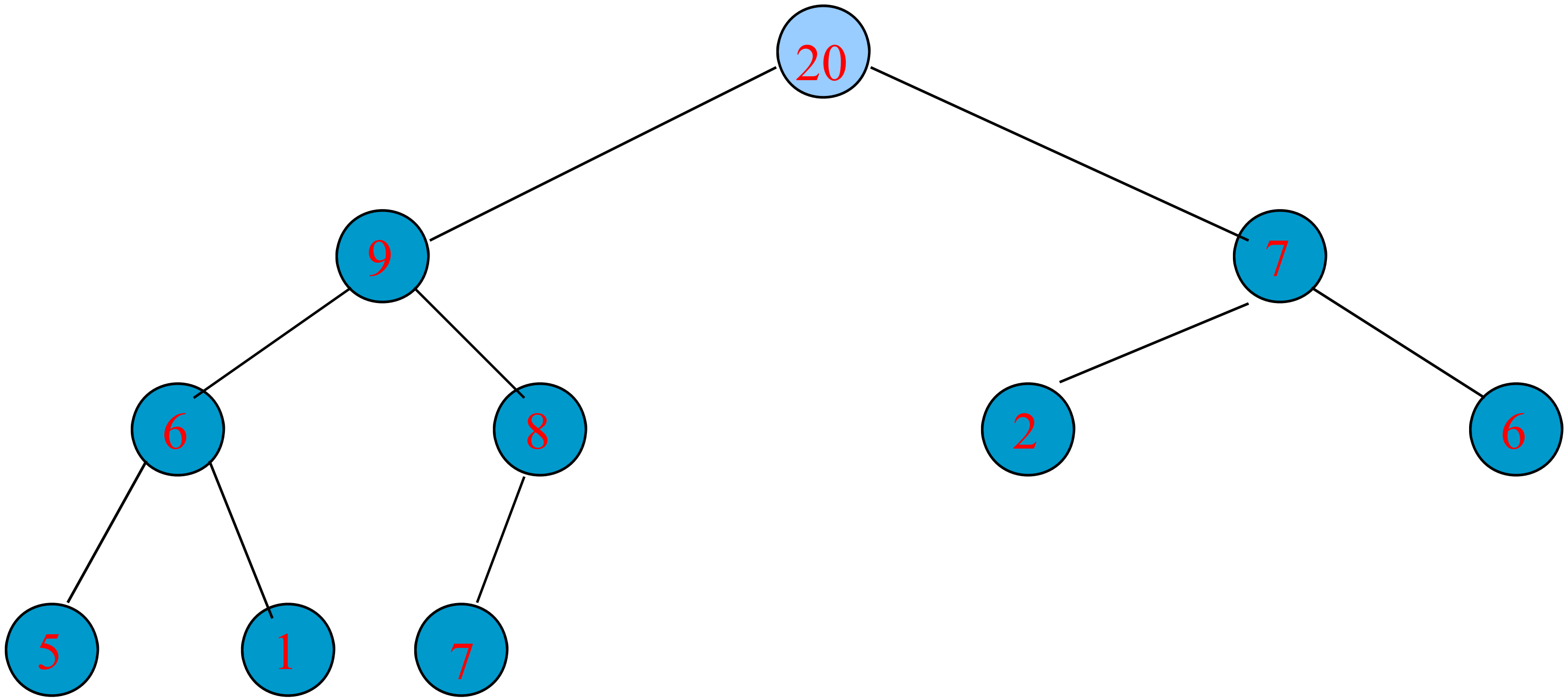
加入 20

插入新的元素到堆積



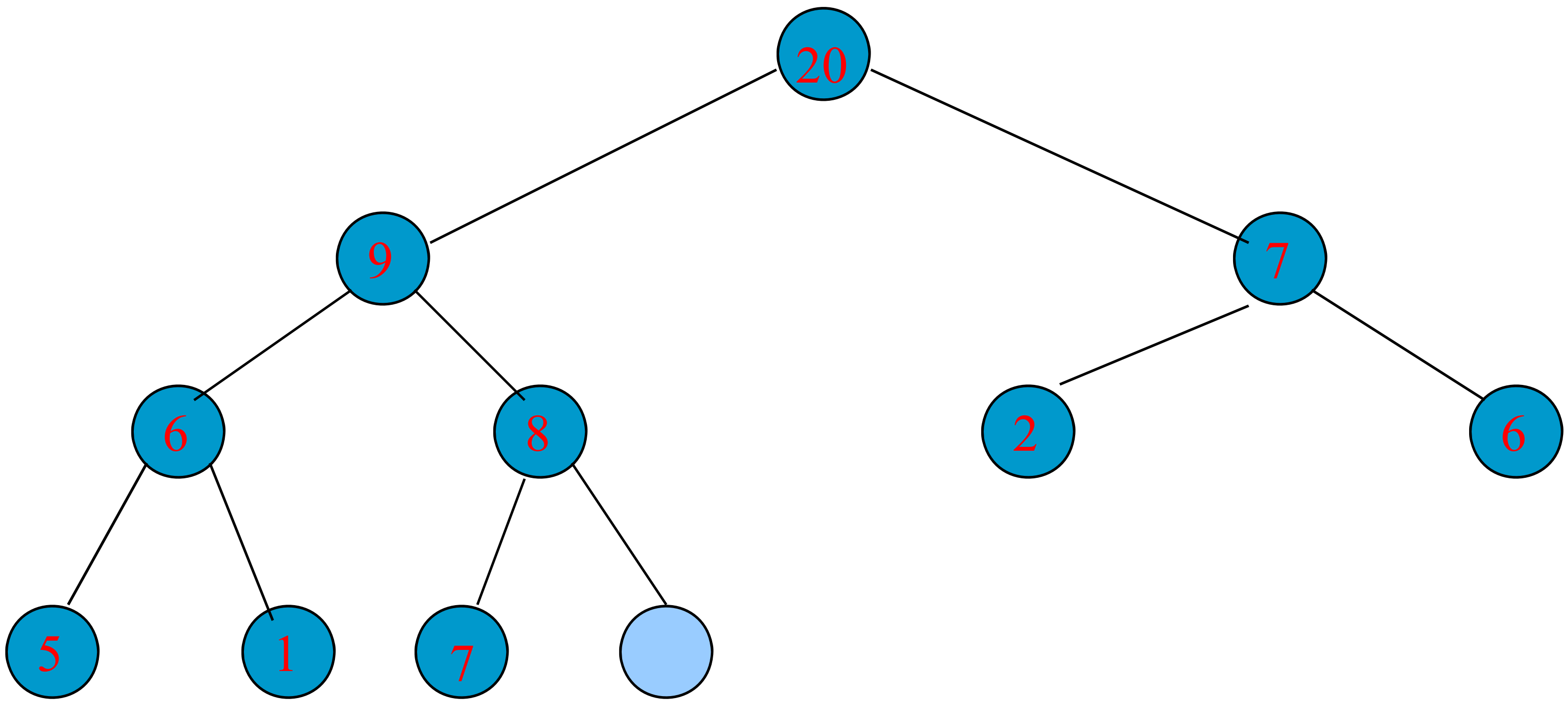
加入 20

插入新的元素到堆積



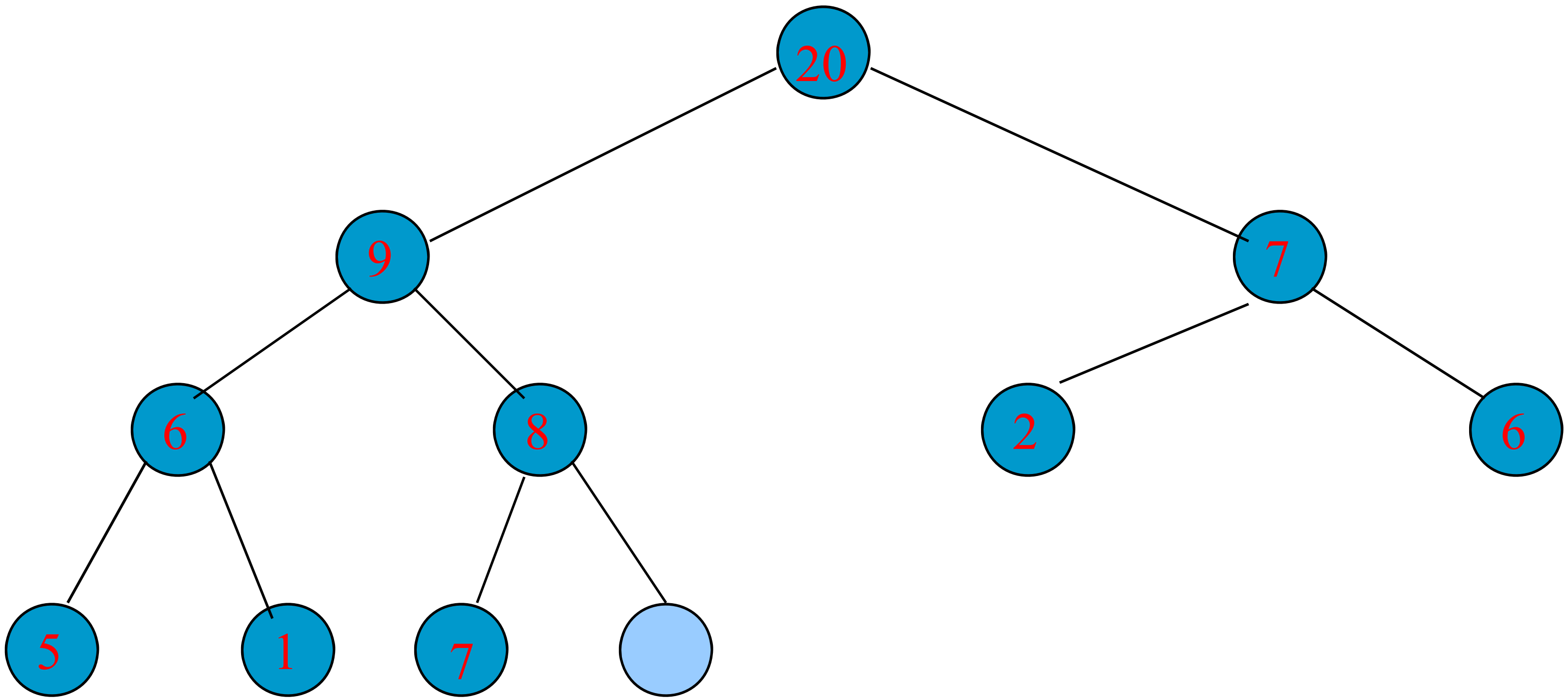
加入 20

插入新的元素到堆積



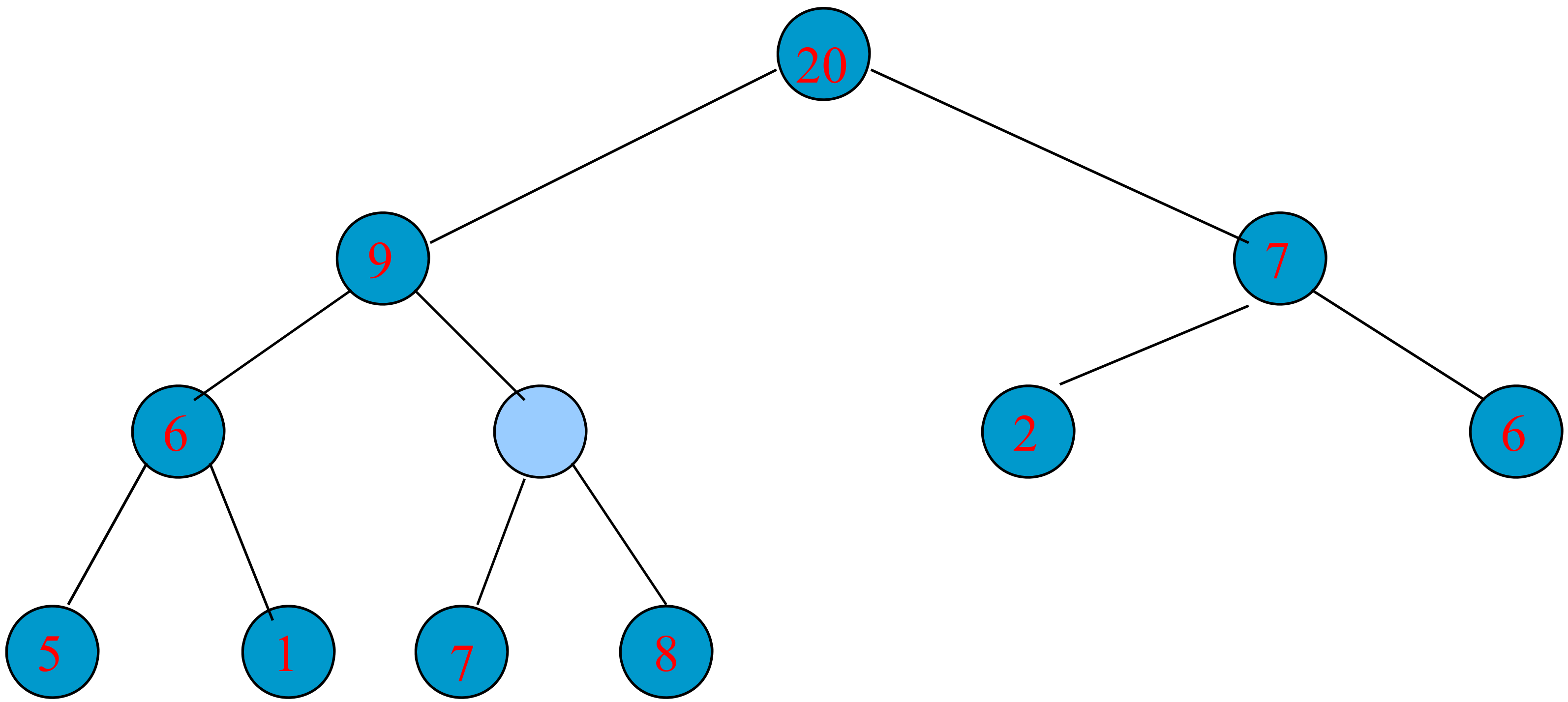
完成

插入新的元素到堆積



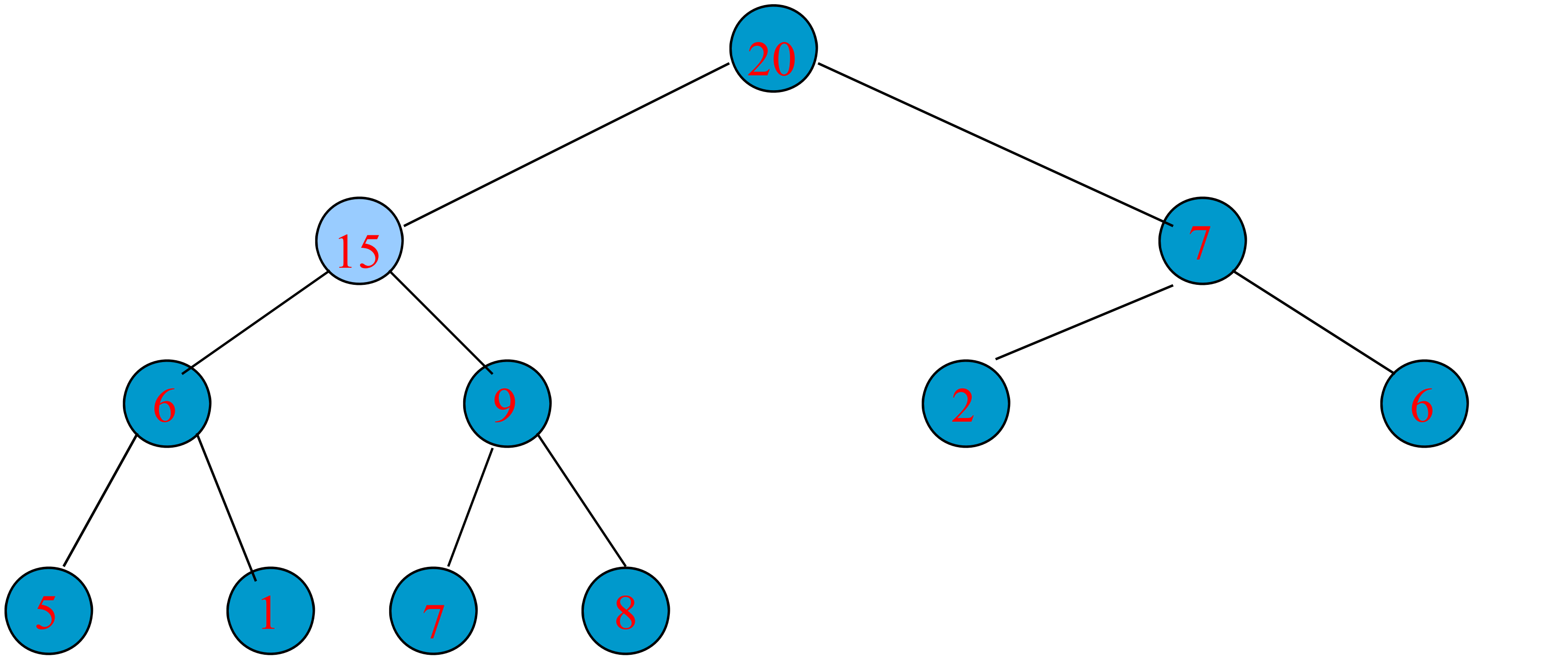
加入 15

插入新的元素到堆積



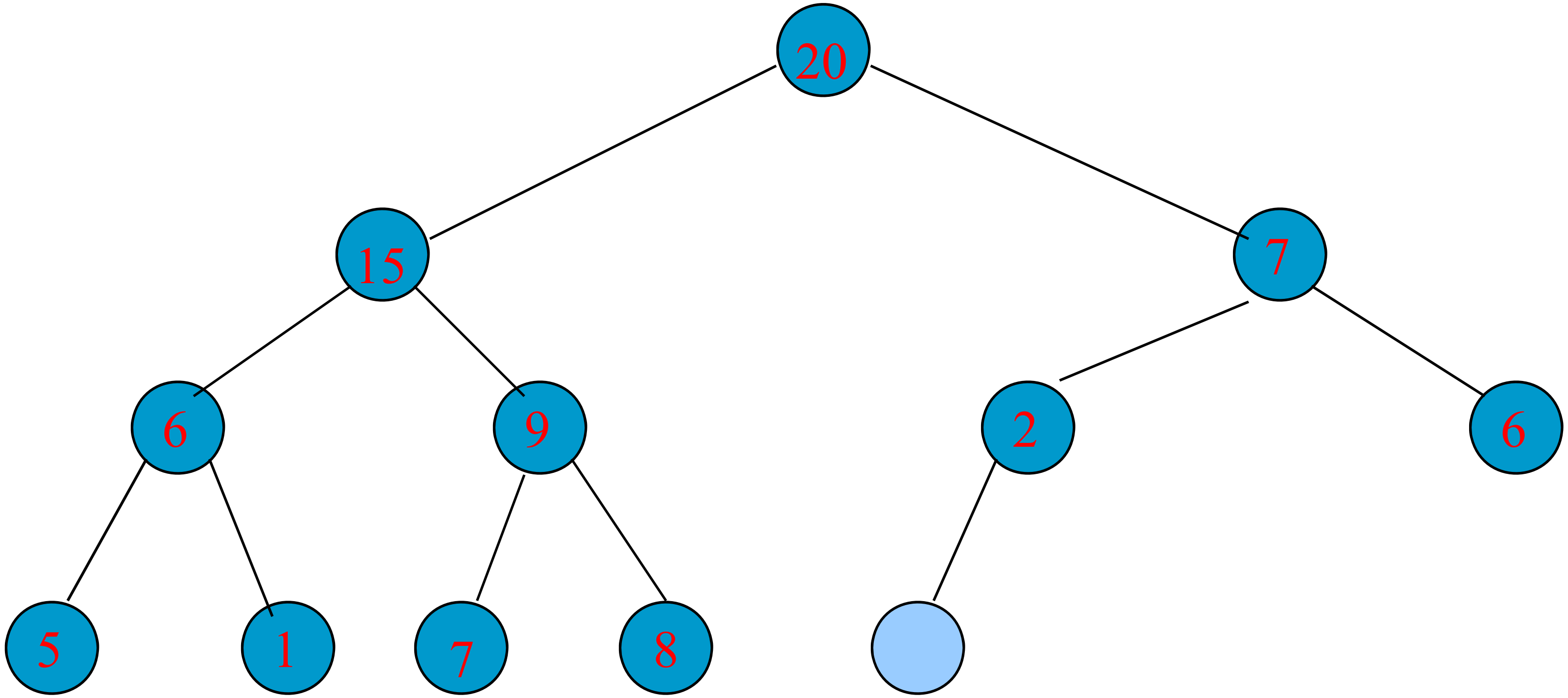
加入 15

插入新的元素到堆積



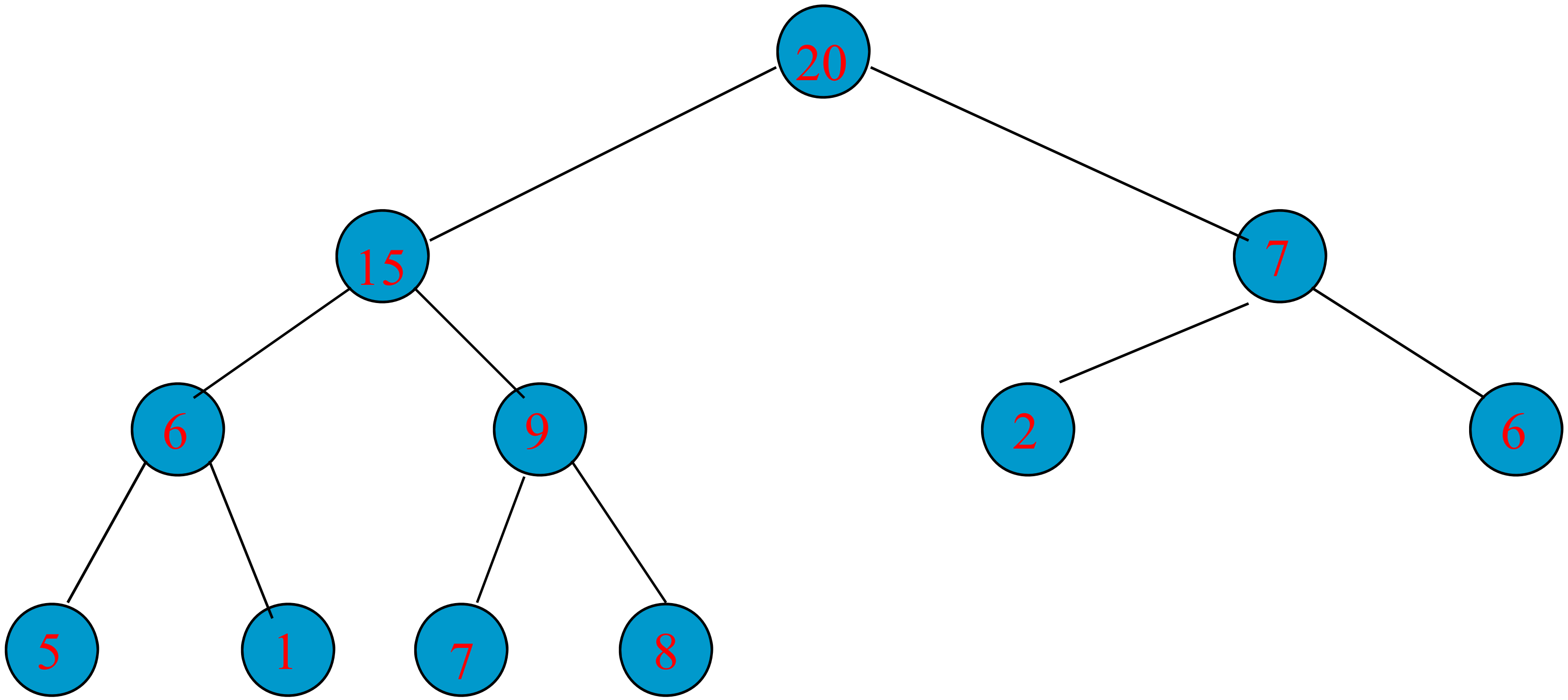
加入 **15**

加入新元素之複雜度



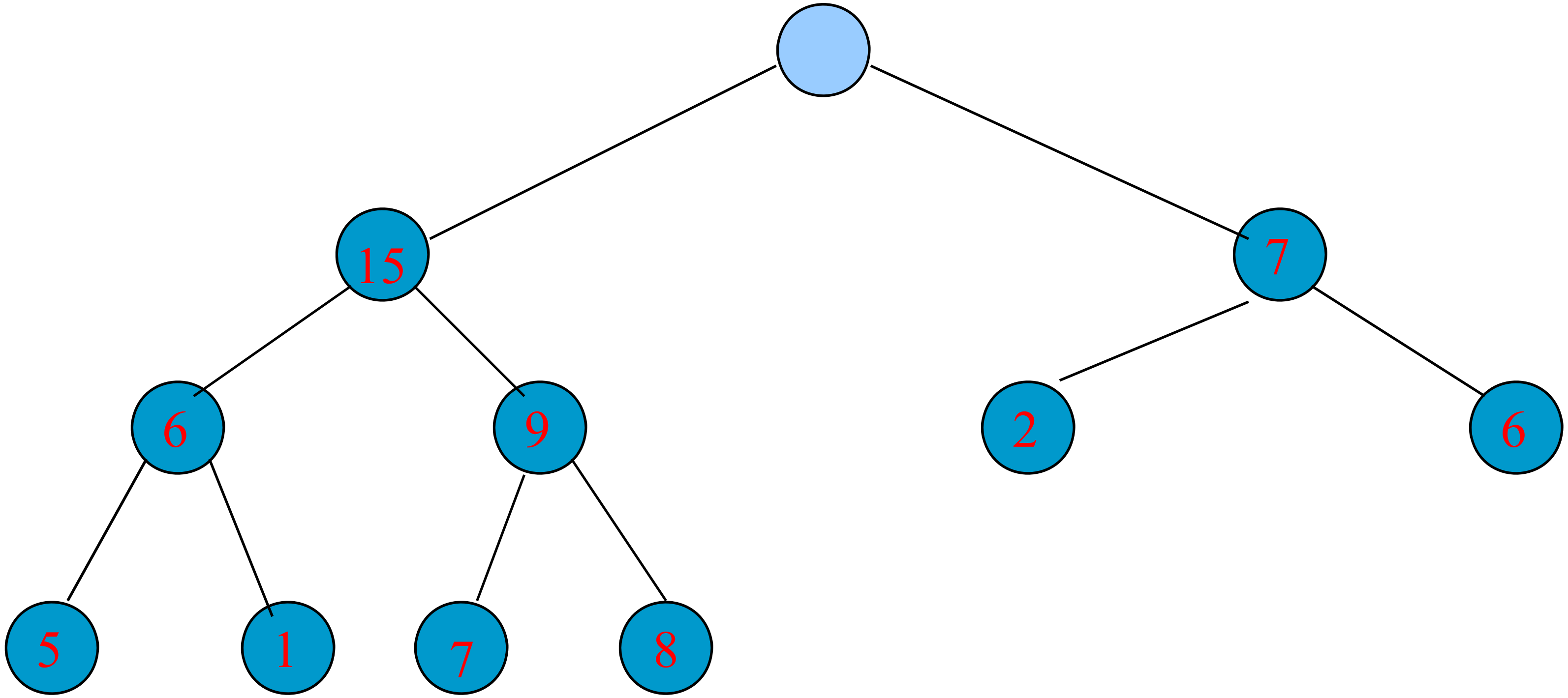
複雜度為O(log n), 其中 n 為堆積之大小

從堆積中刪除最大的元素



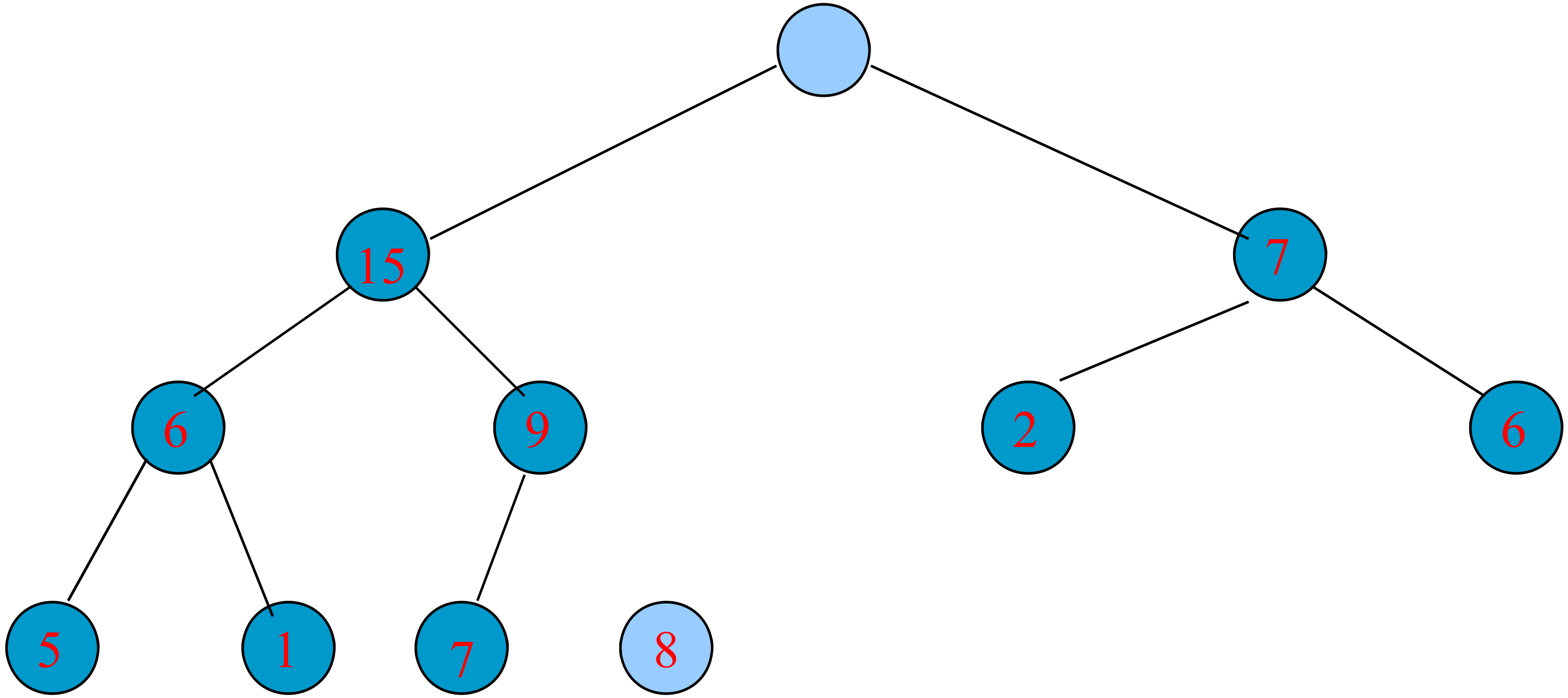
最大元素在樹根

從堆積中刪除最大的元素



刪除最大元素後

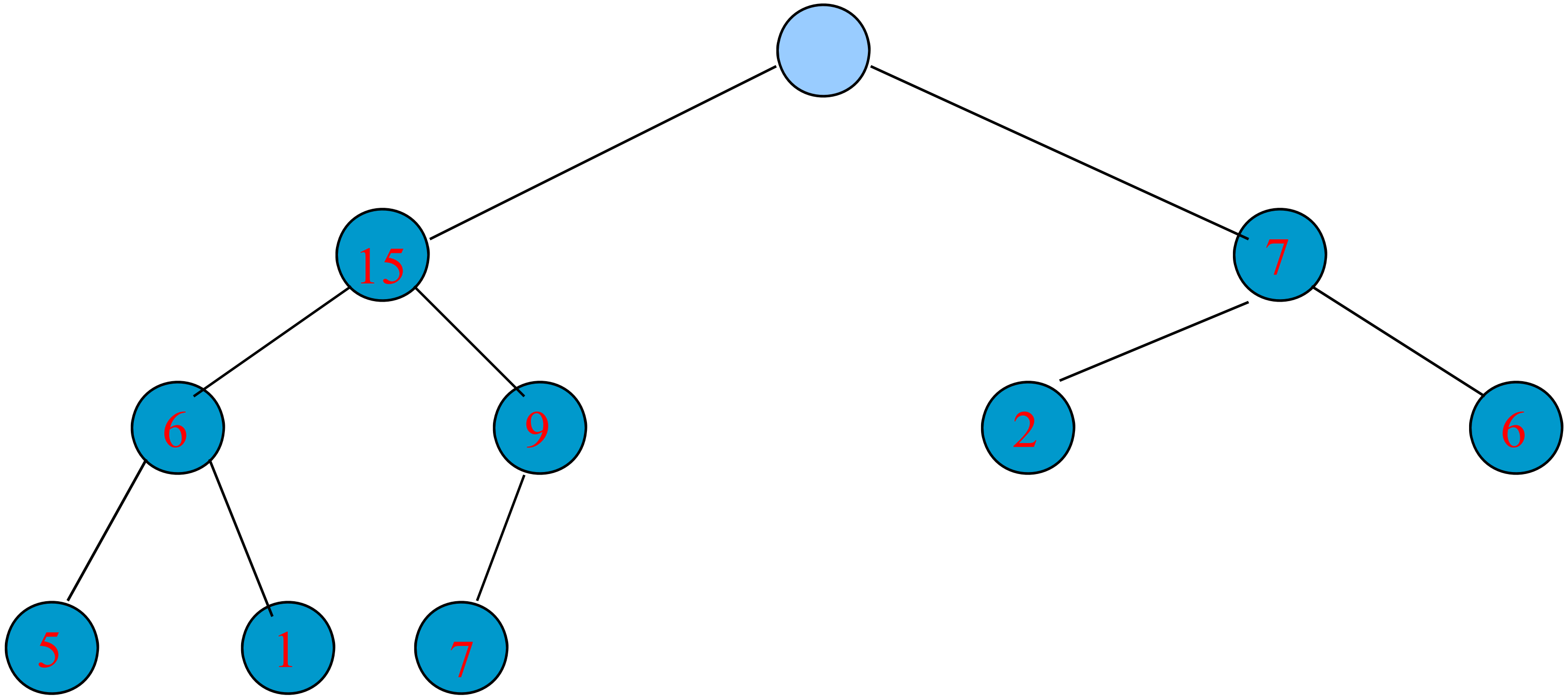
從堆積中刪除最大的元素



10 個節點的堆積

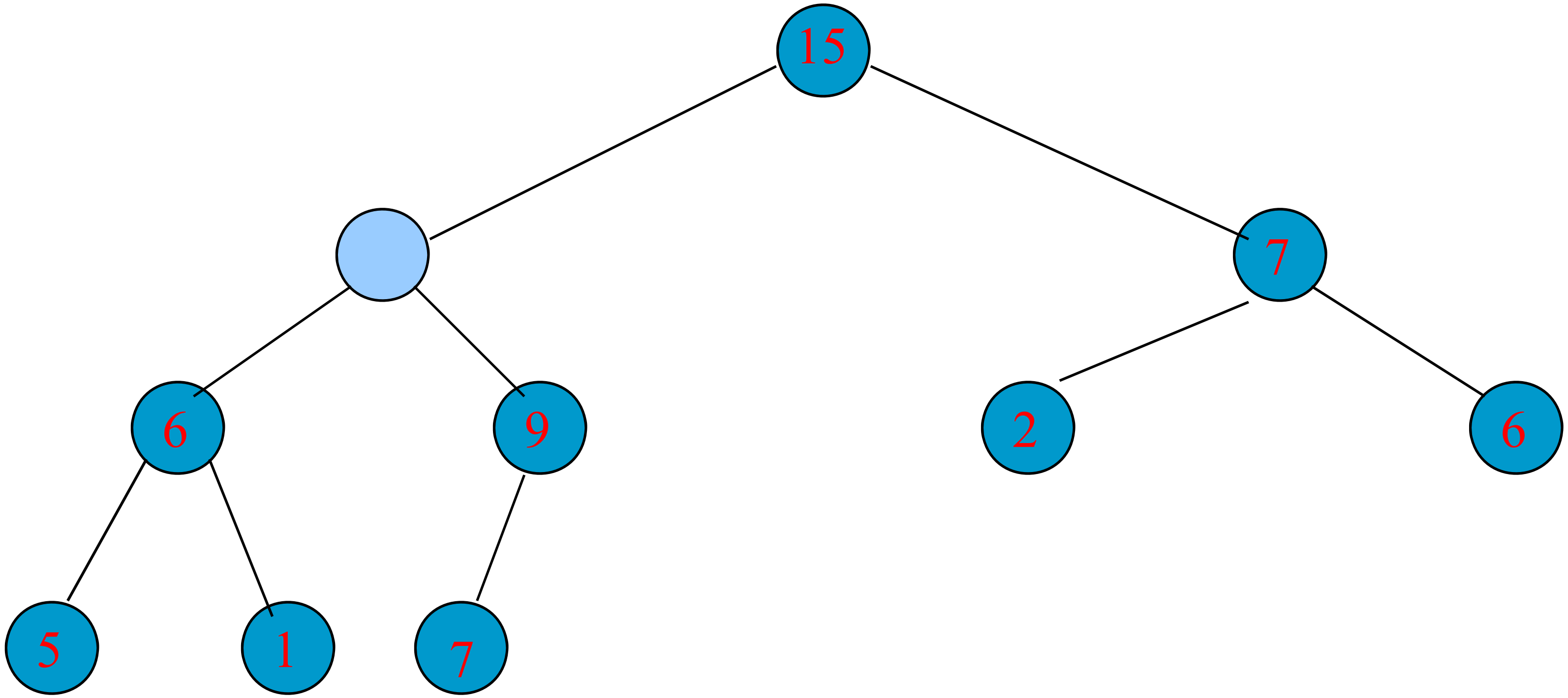
重新將8插入堆積

從堆積中刪除最大的元素



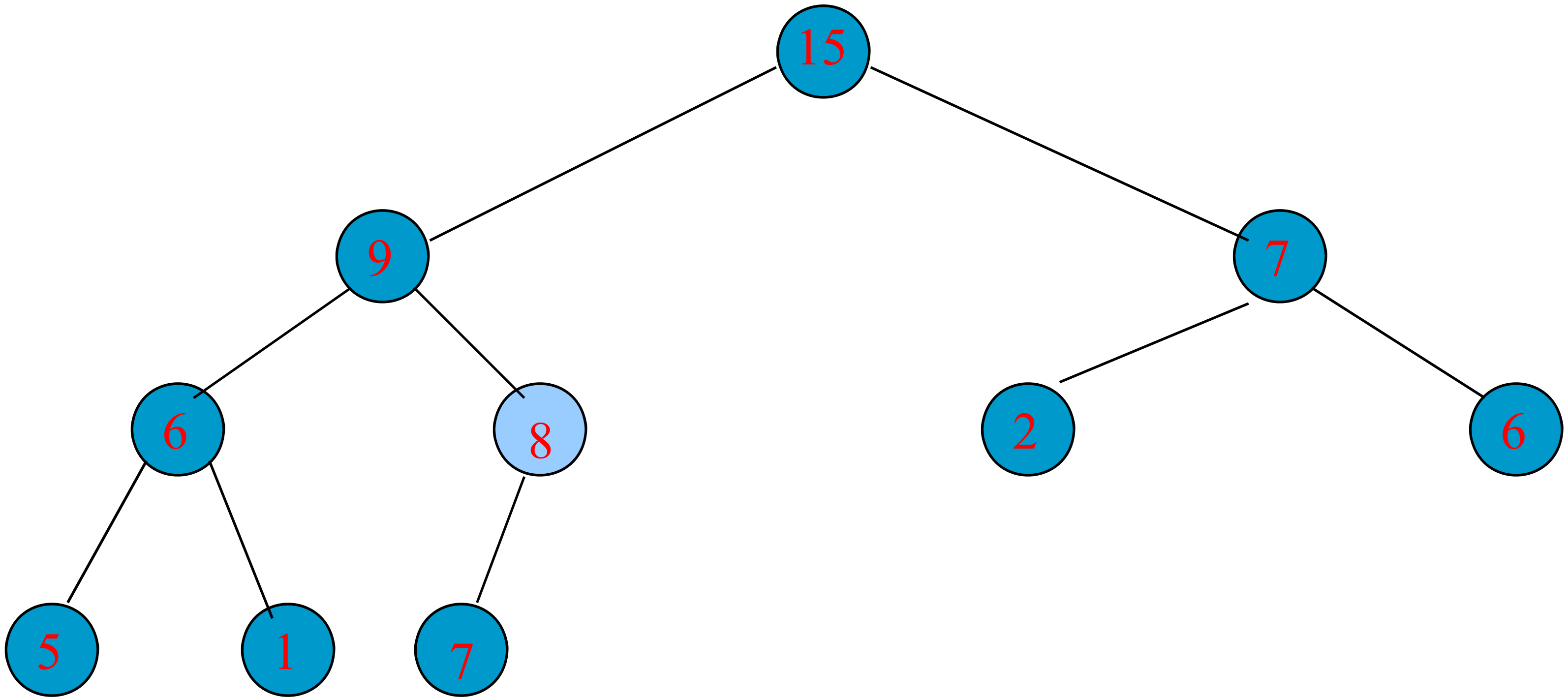
重新將8插入堆積

從堆積中刪除最大的元素



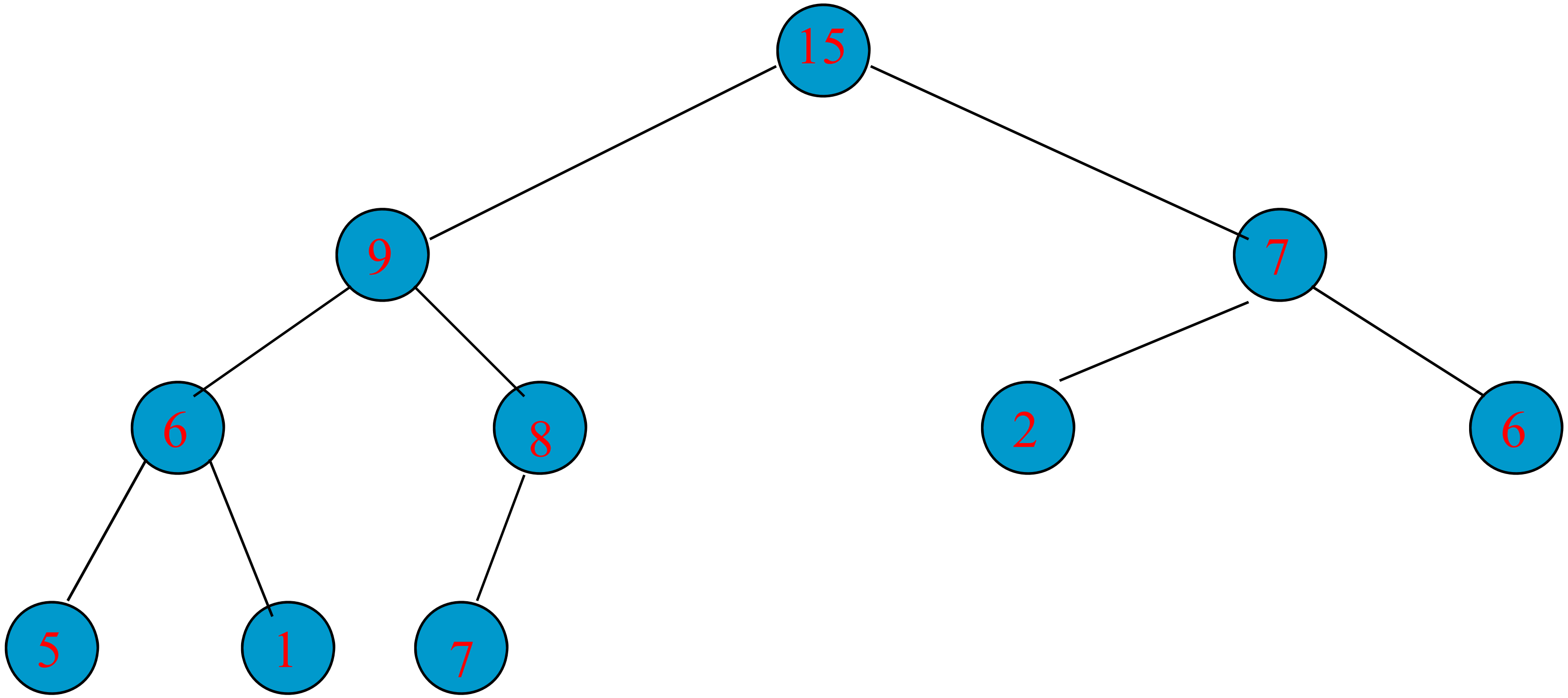
重新將8插入堆積

從堆積中刪除最大的元素



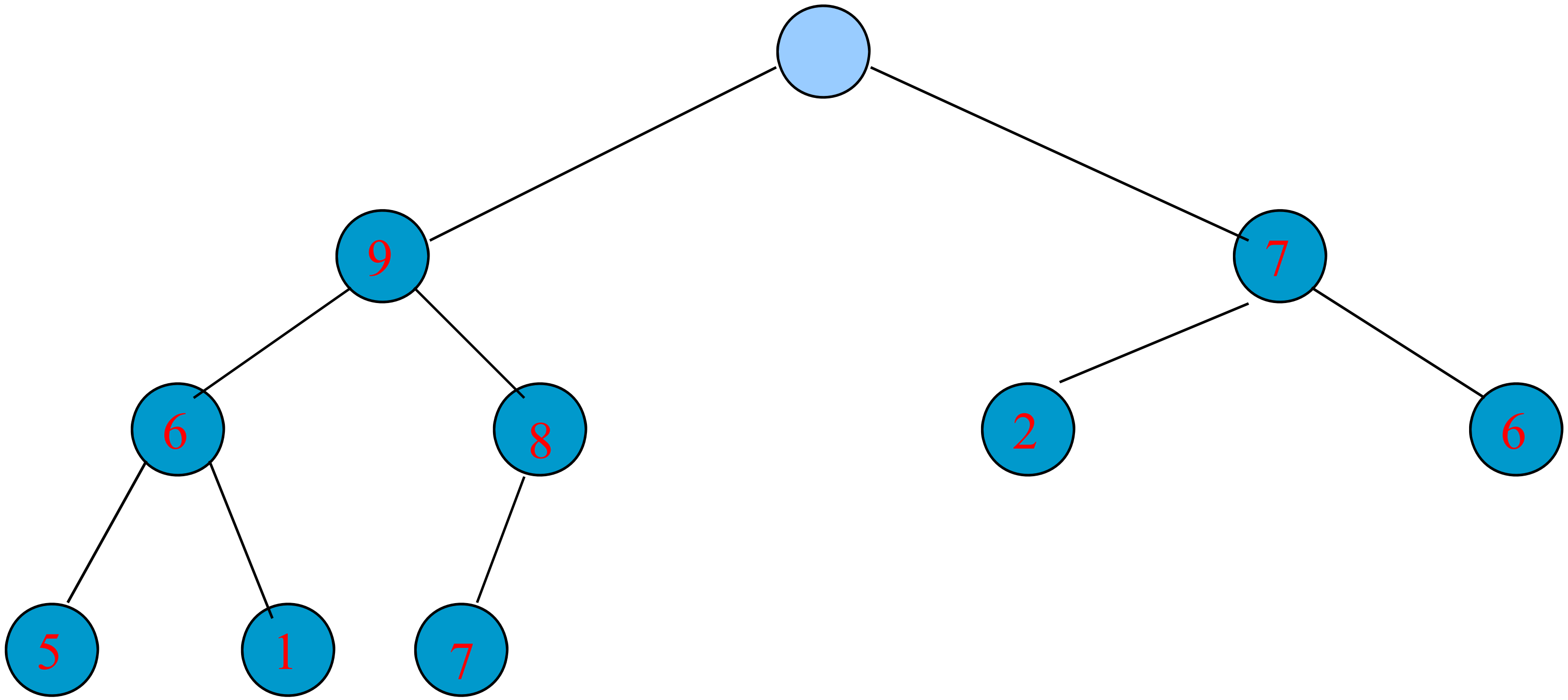
重新將8插入堆積

從堆積中刪除最大的元素



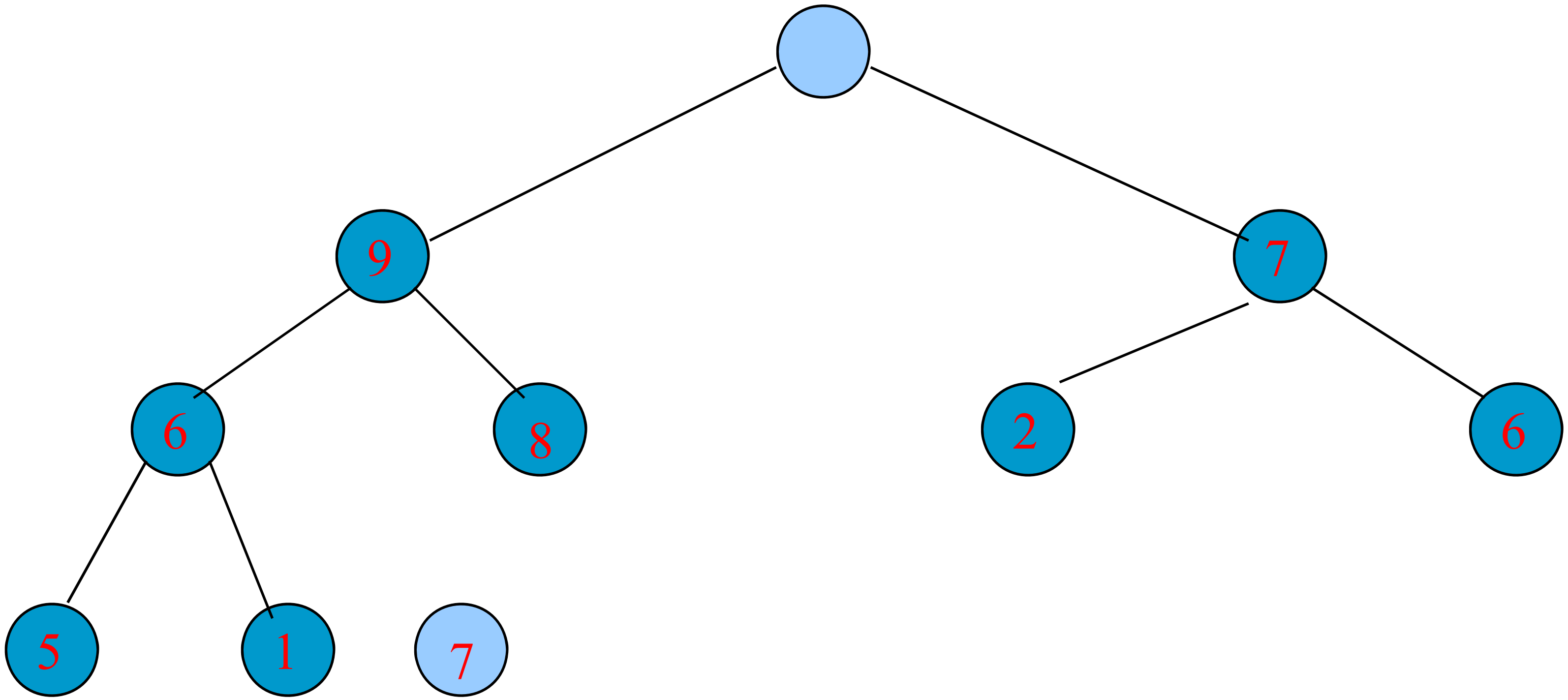
最大元素為 15

從堆積中刪除最大的元素



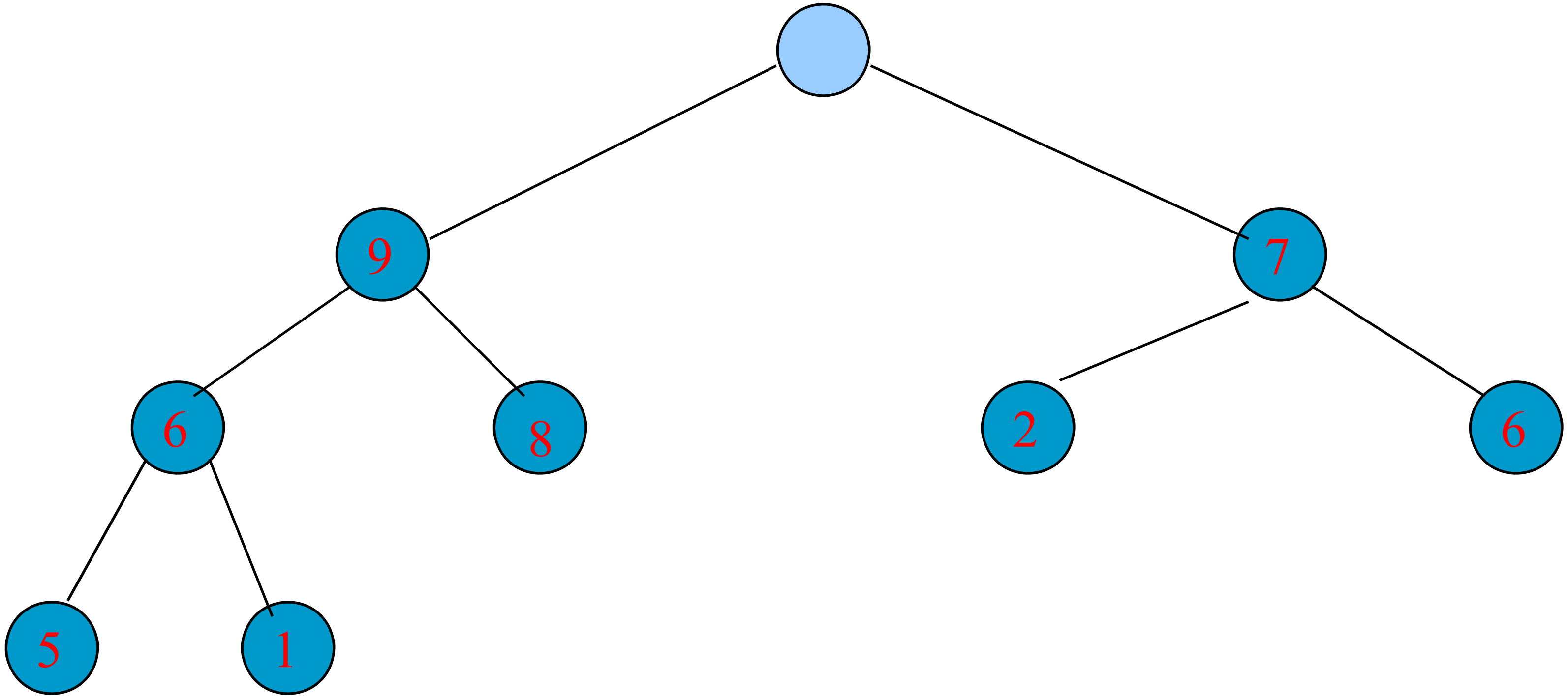
刪除最大元素後

從堆積中刪除最大的元素



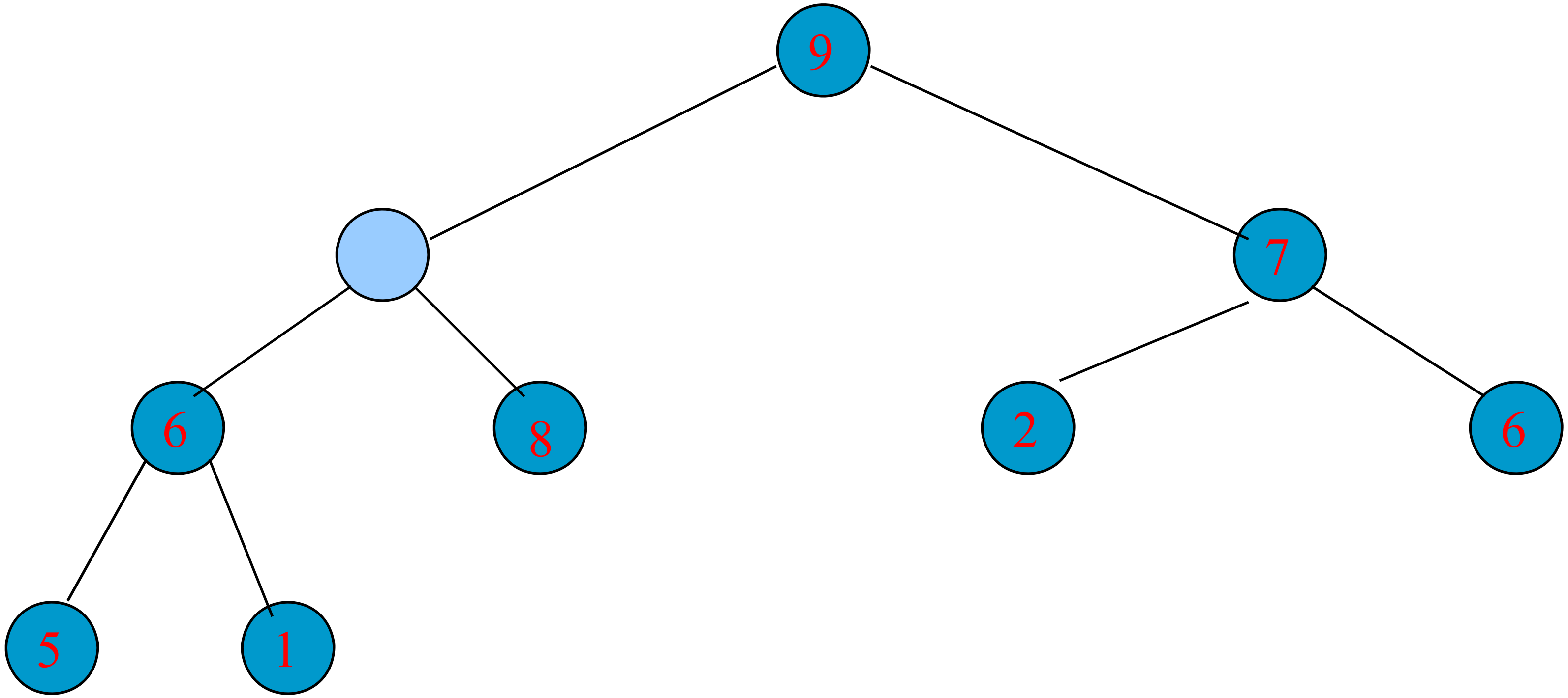
9個節點的堆積

從堆積中刪除最大的元素



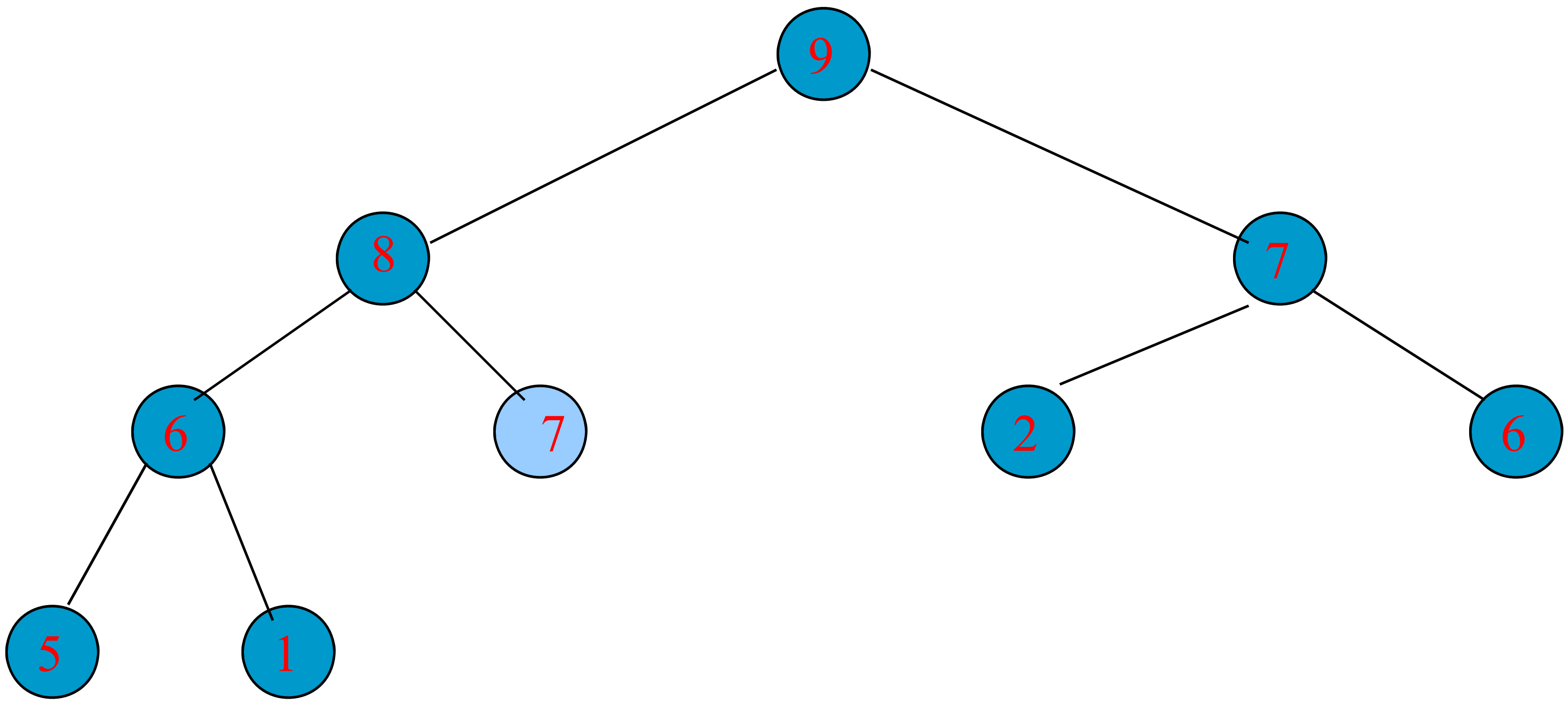
重新將7插入堆積

從堆積中刪除最大的元素



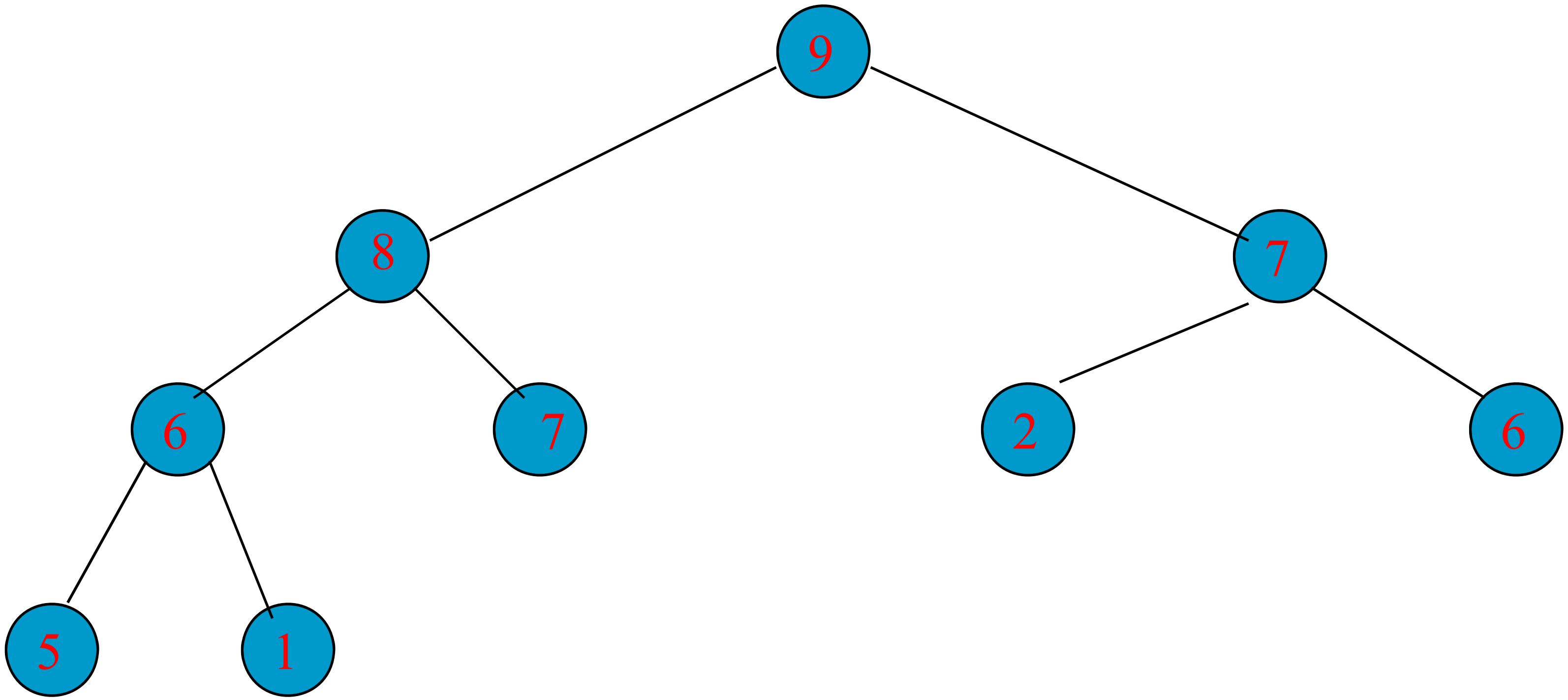
重新將7插入堆積

從堆積中刪除最大的元素



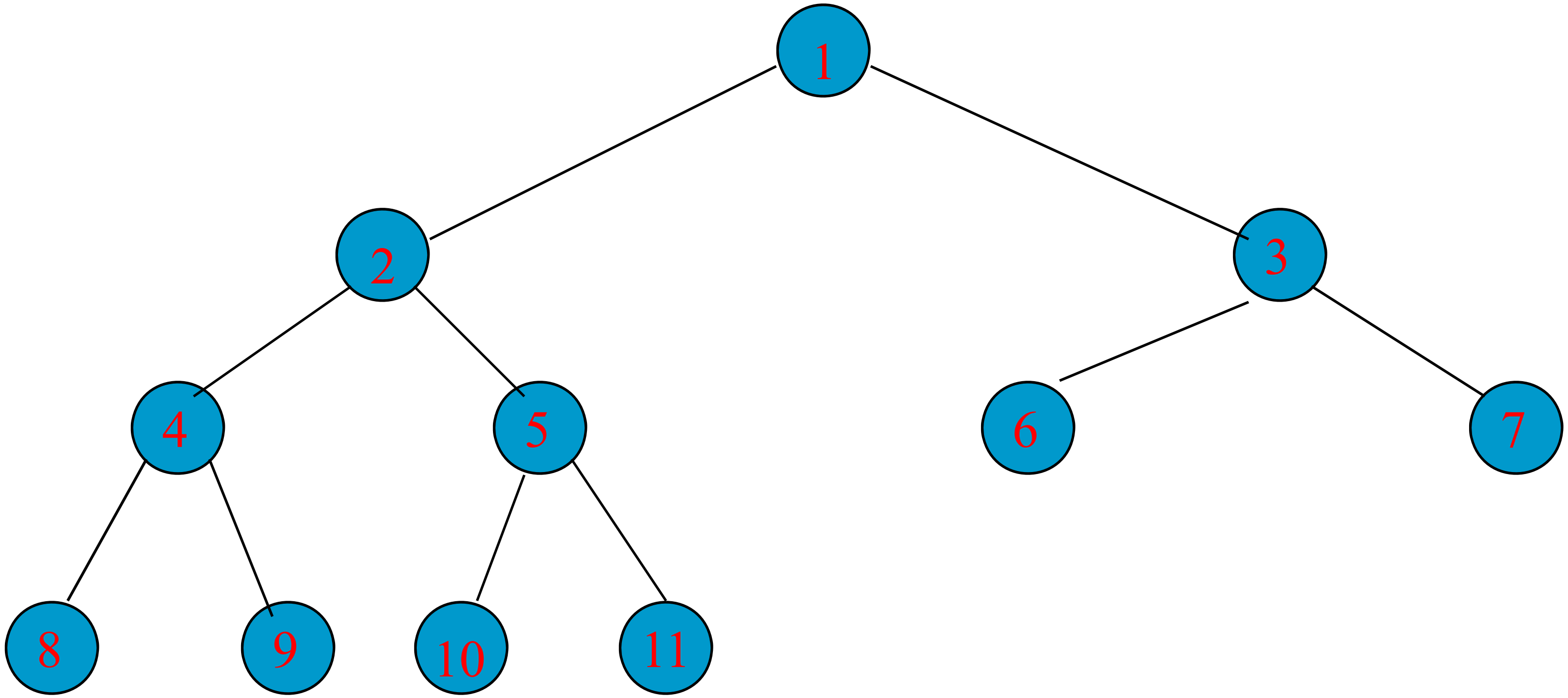
重新將7插入堆積

刪除最大的元素之複雜度



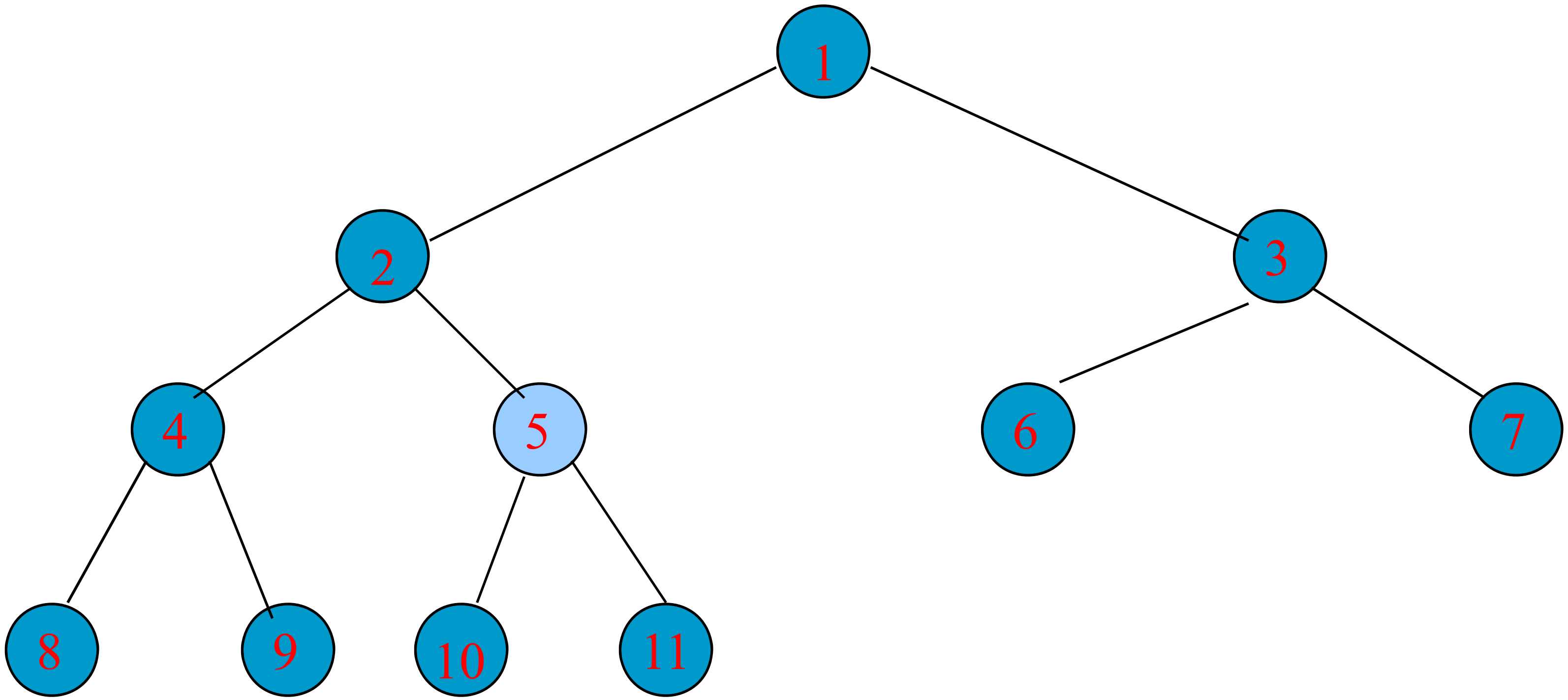
複雜度為O(log n).

最大堆積之起始



輸入 array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

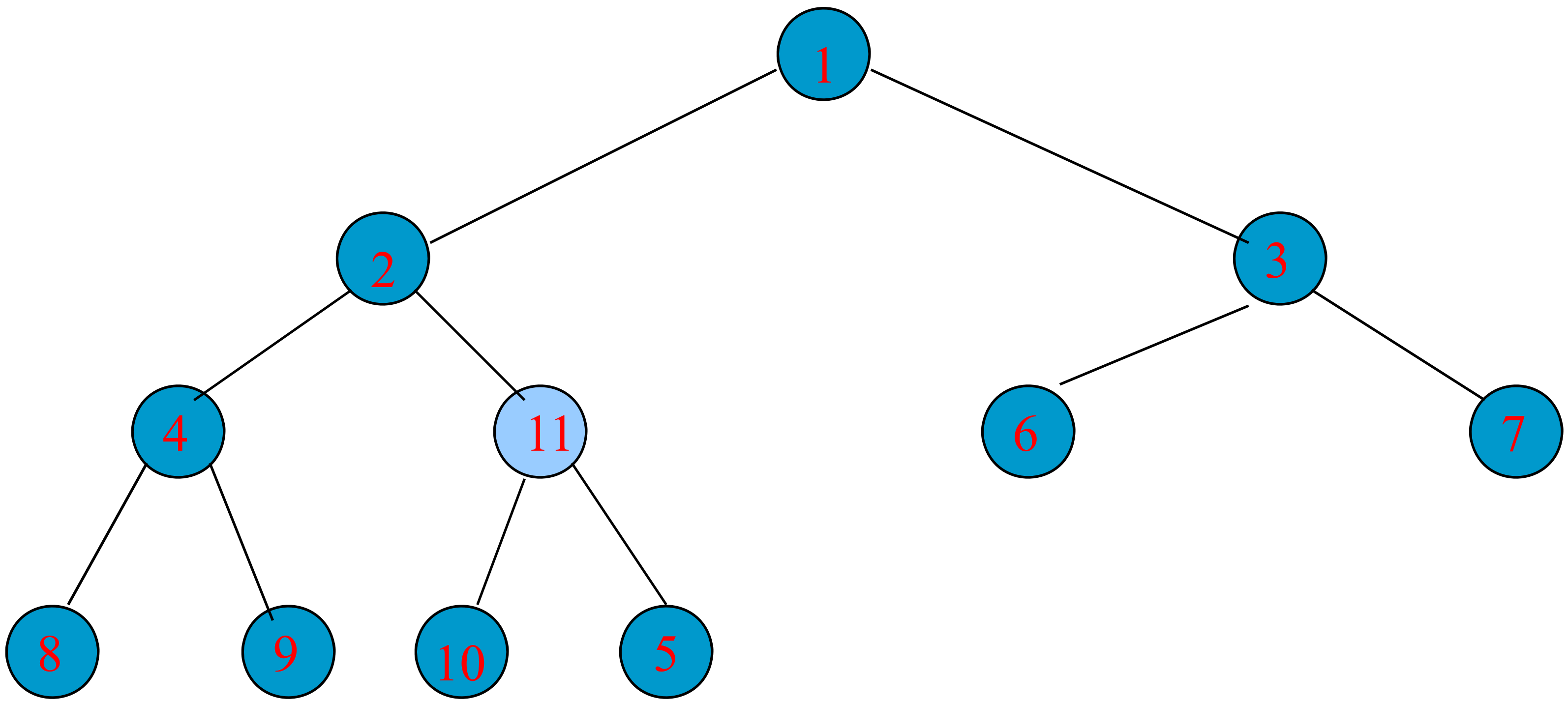
最大堆積之起始



從有兒子節點的最後一個節點開始

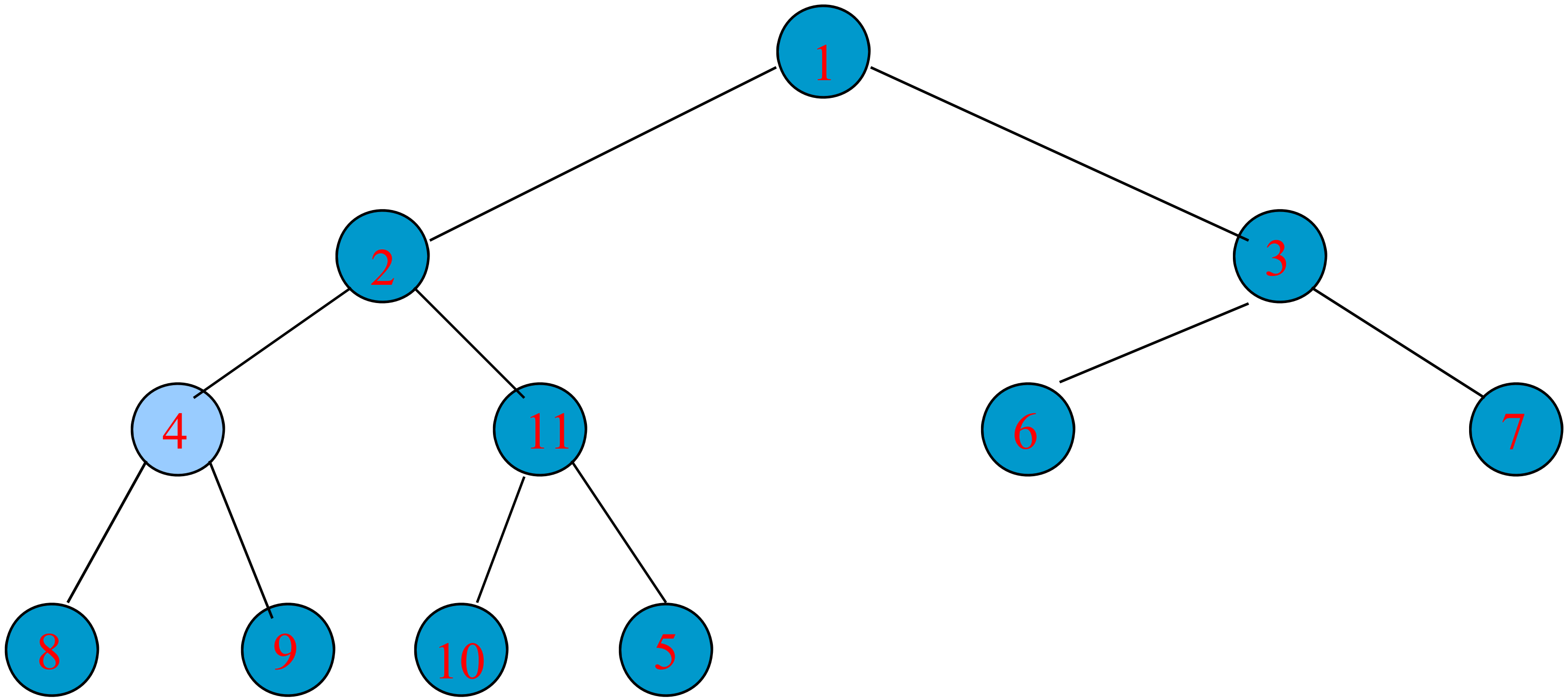
索引值為 **n/2**

最大堆積之起始

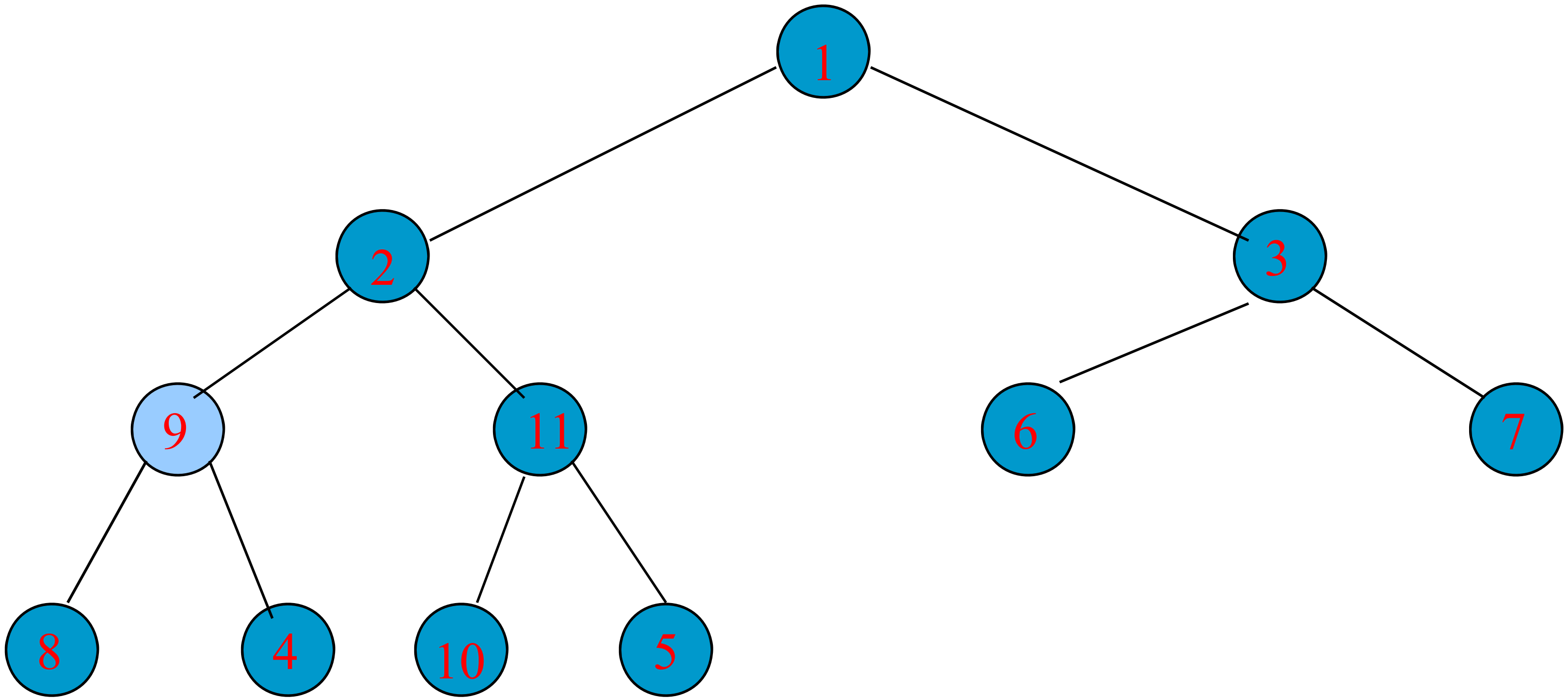


換下一個位置

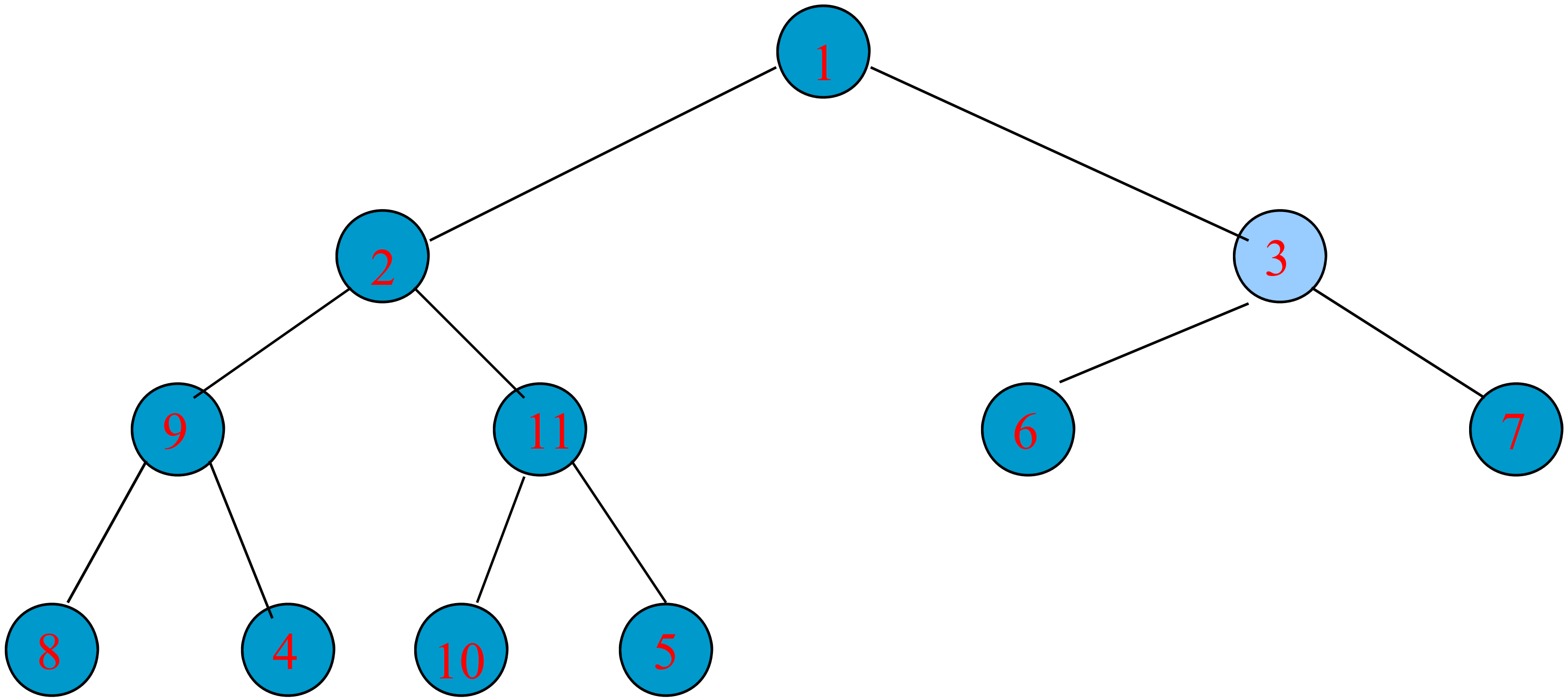
最大堆積之起始



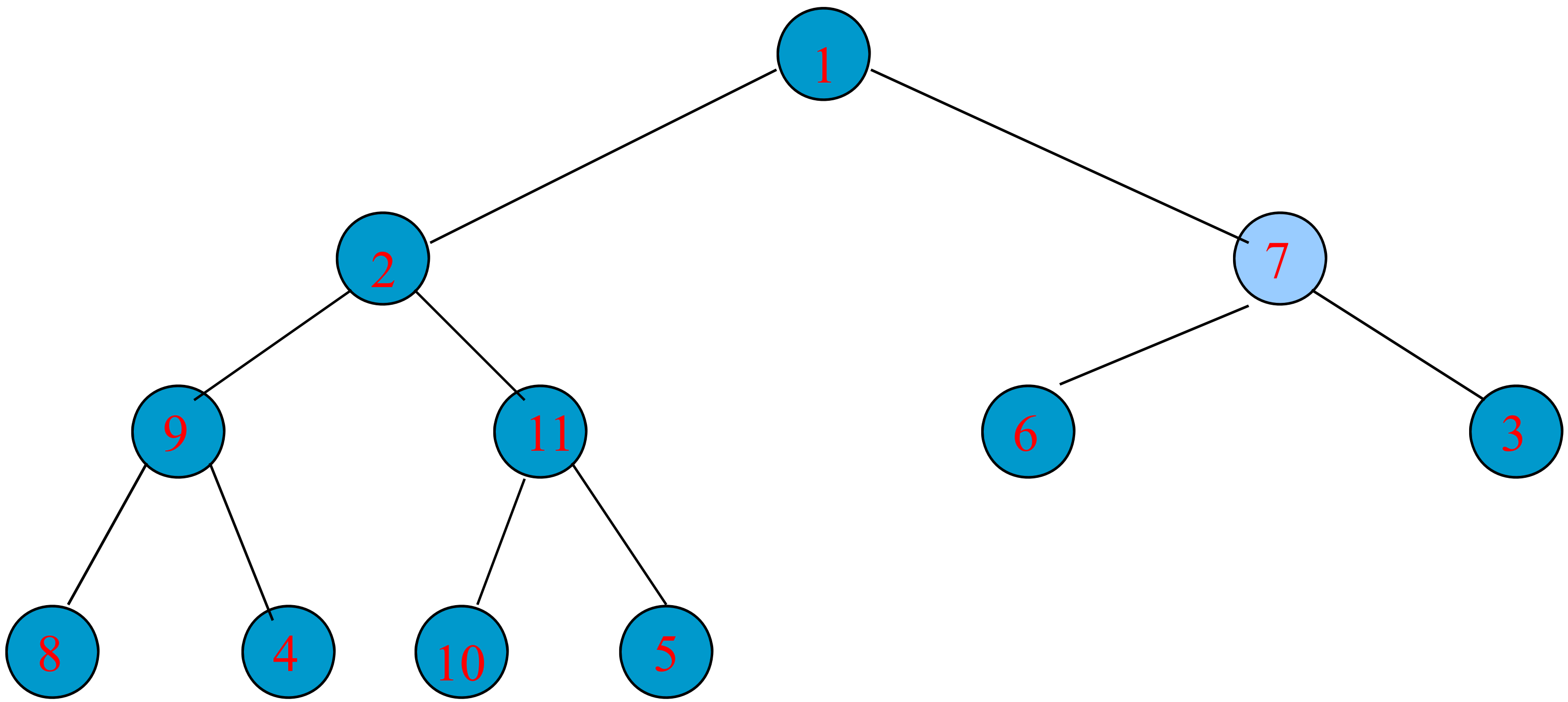
最大堆積之起始



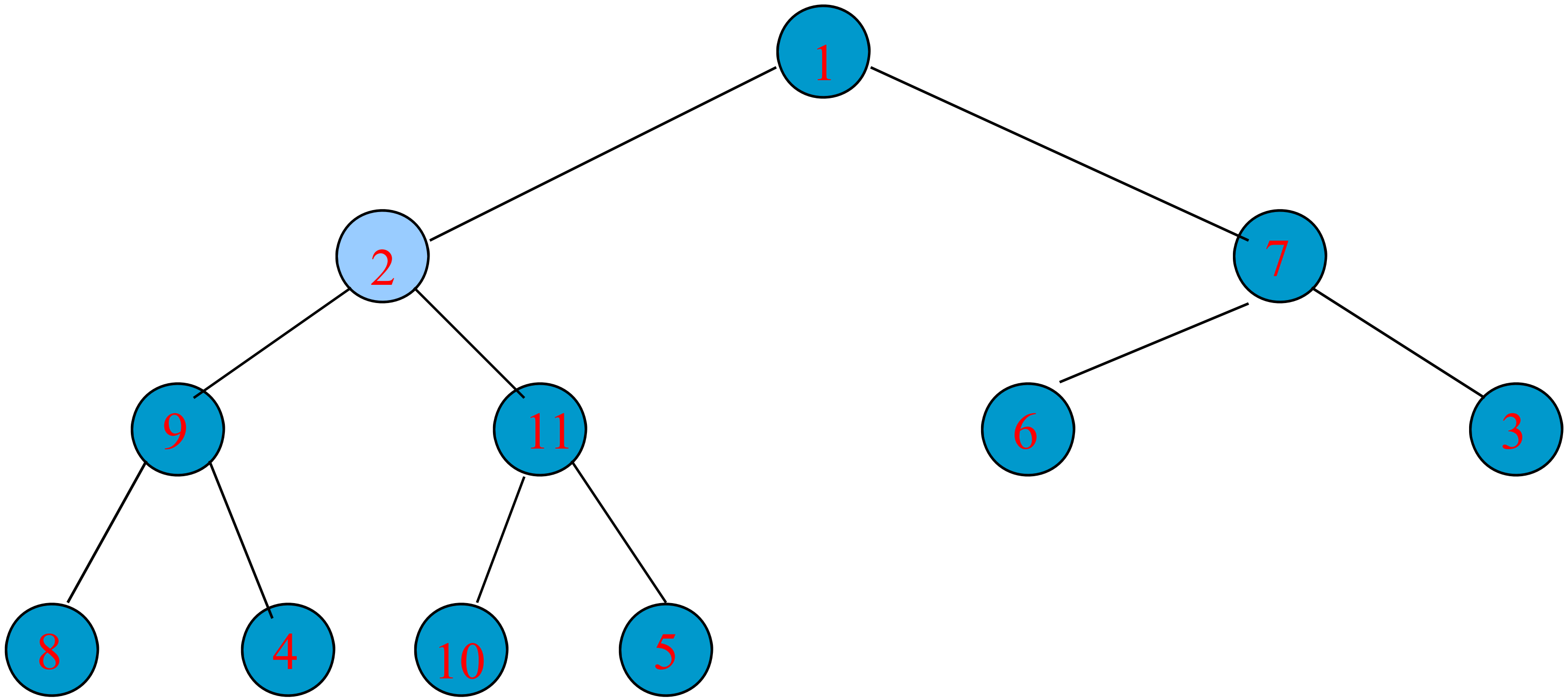
最大堆積之起始



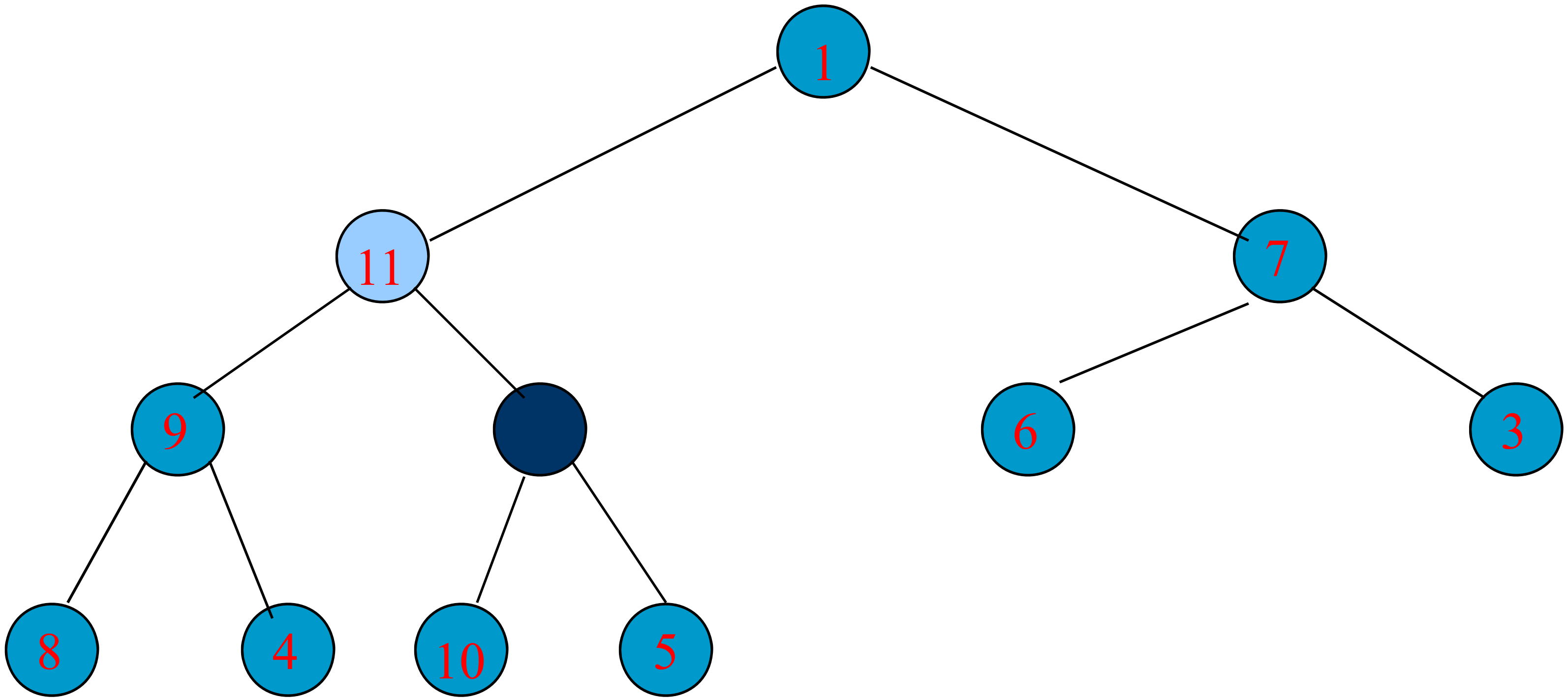
最大堆積之起始



最大堆積之起始

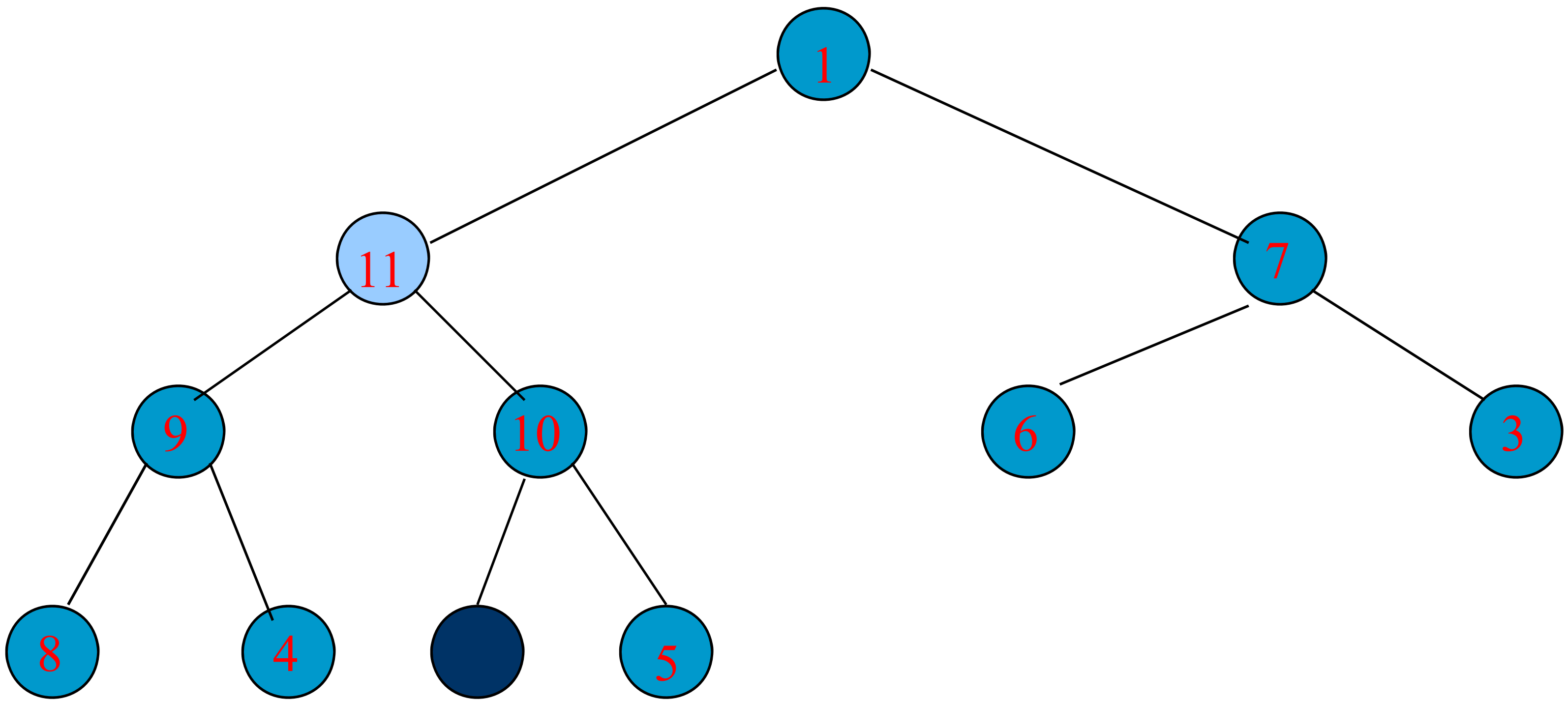


最大堆積之起始



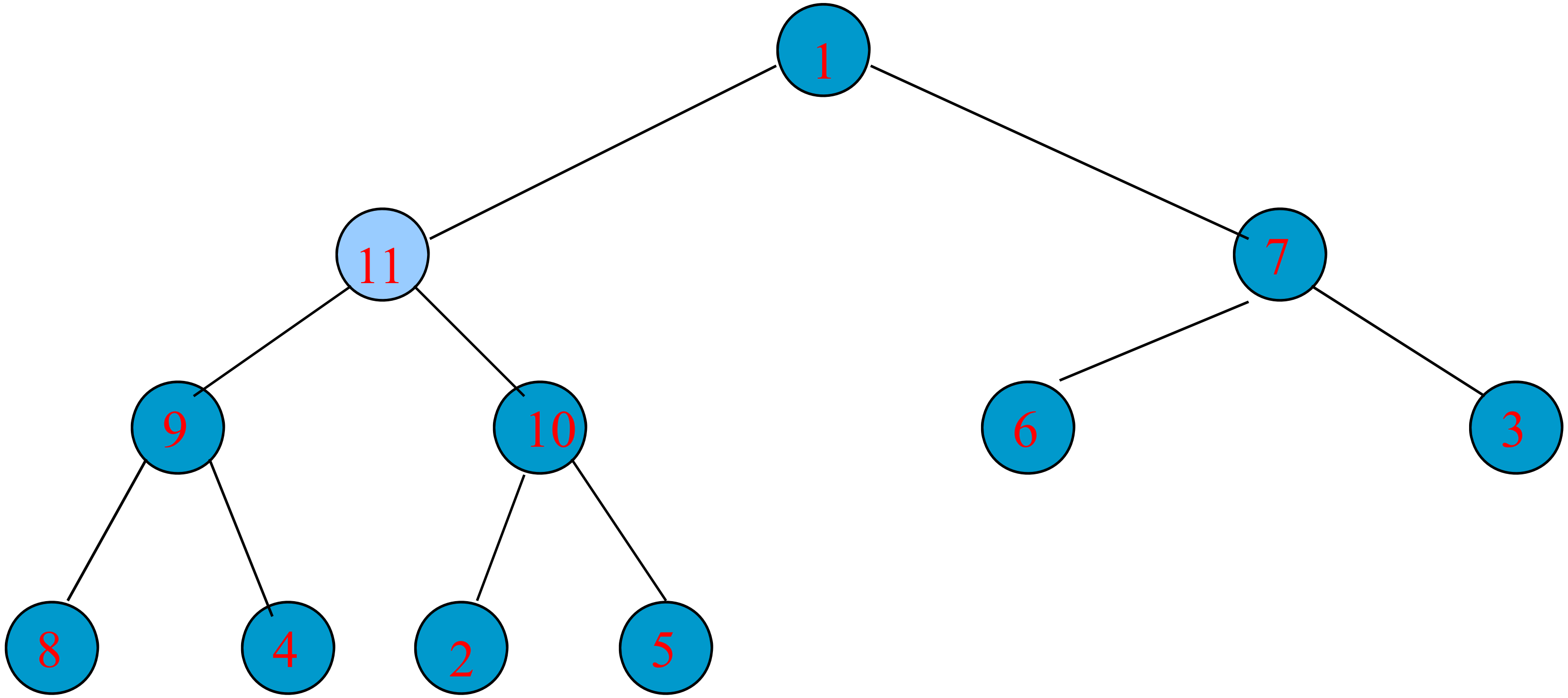
給 2找家

最大堆積之起始



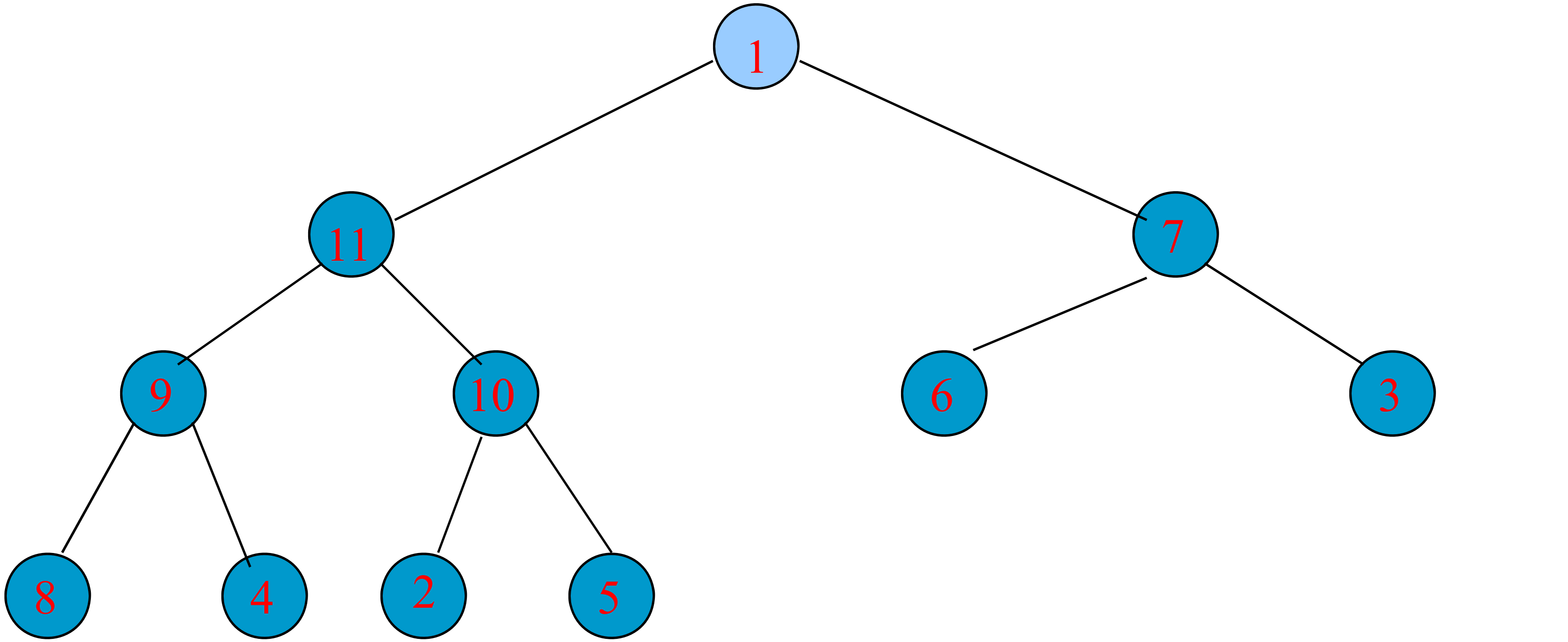
給 2找家

最大堆積之起始



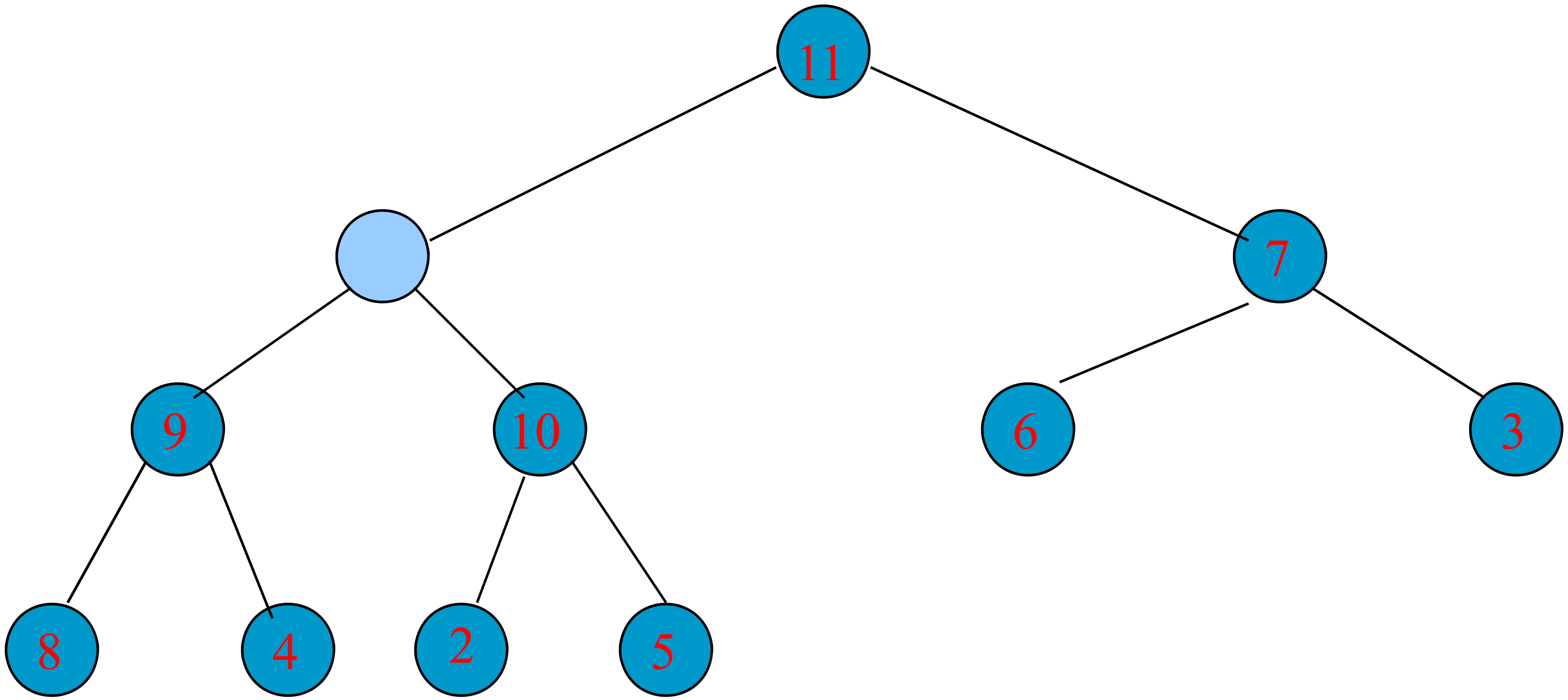
完成！換下一個位置

最大堆積之起始



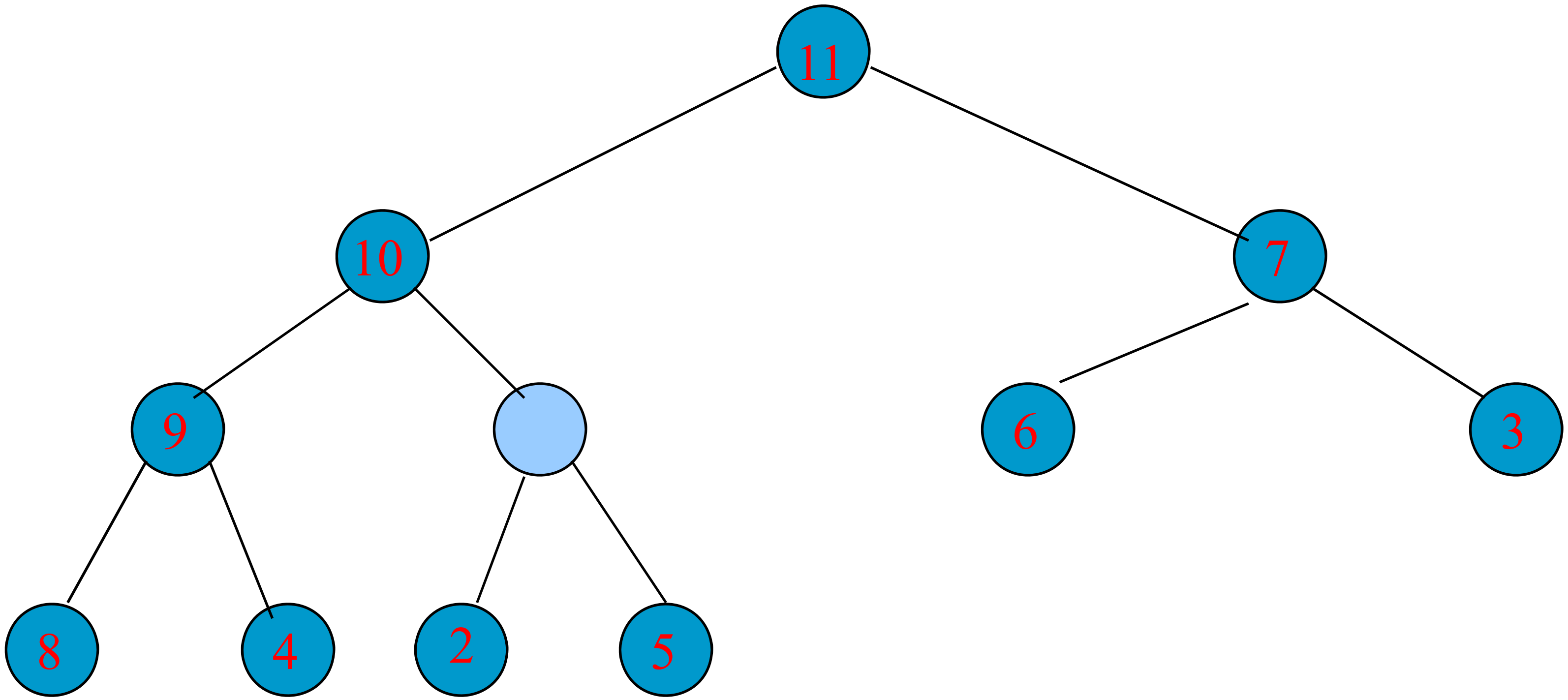
給 1找家

最大堆積之起始



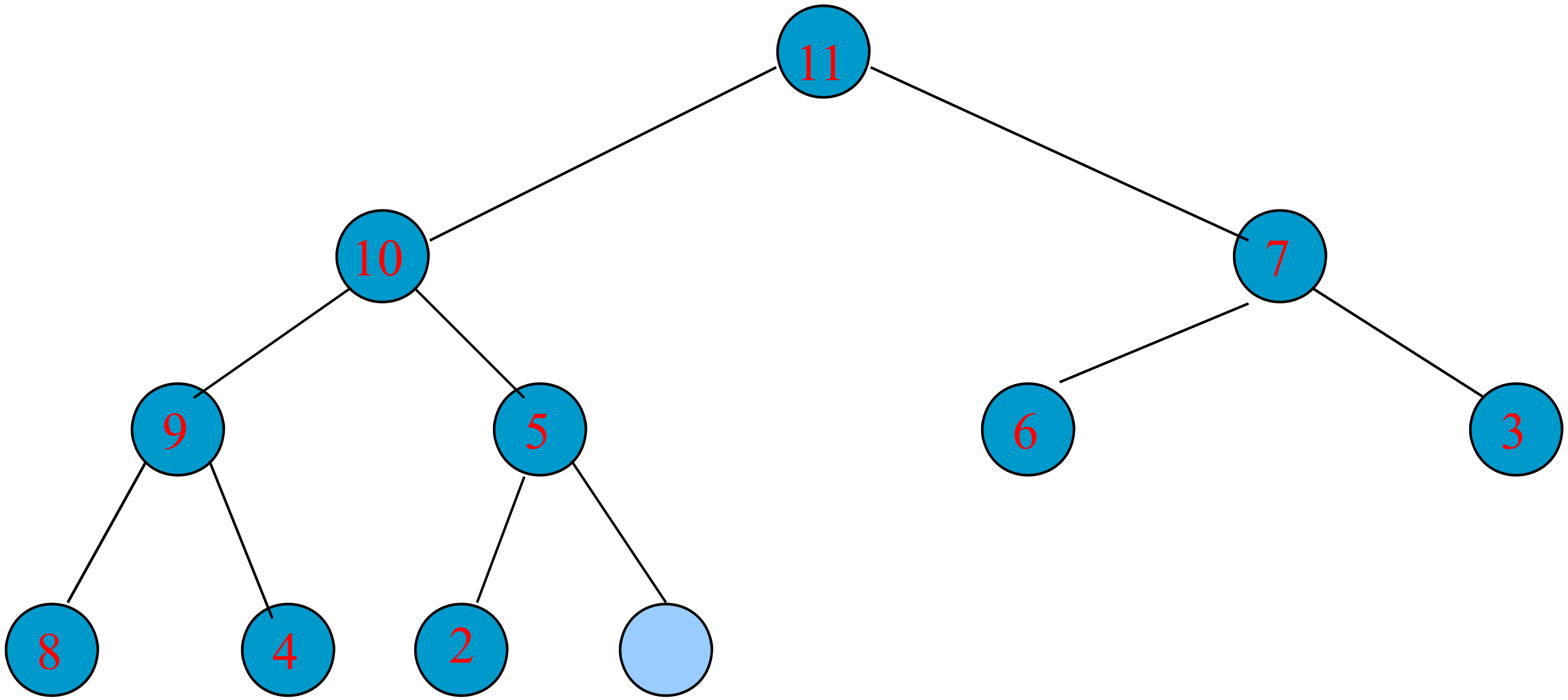
給 1找家

最大堆積之起始



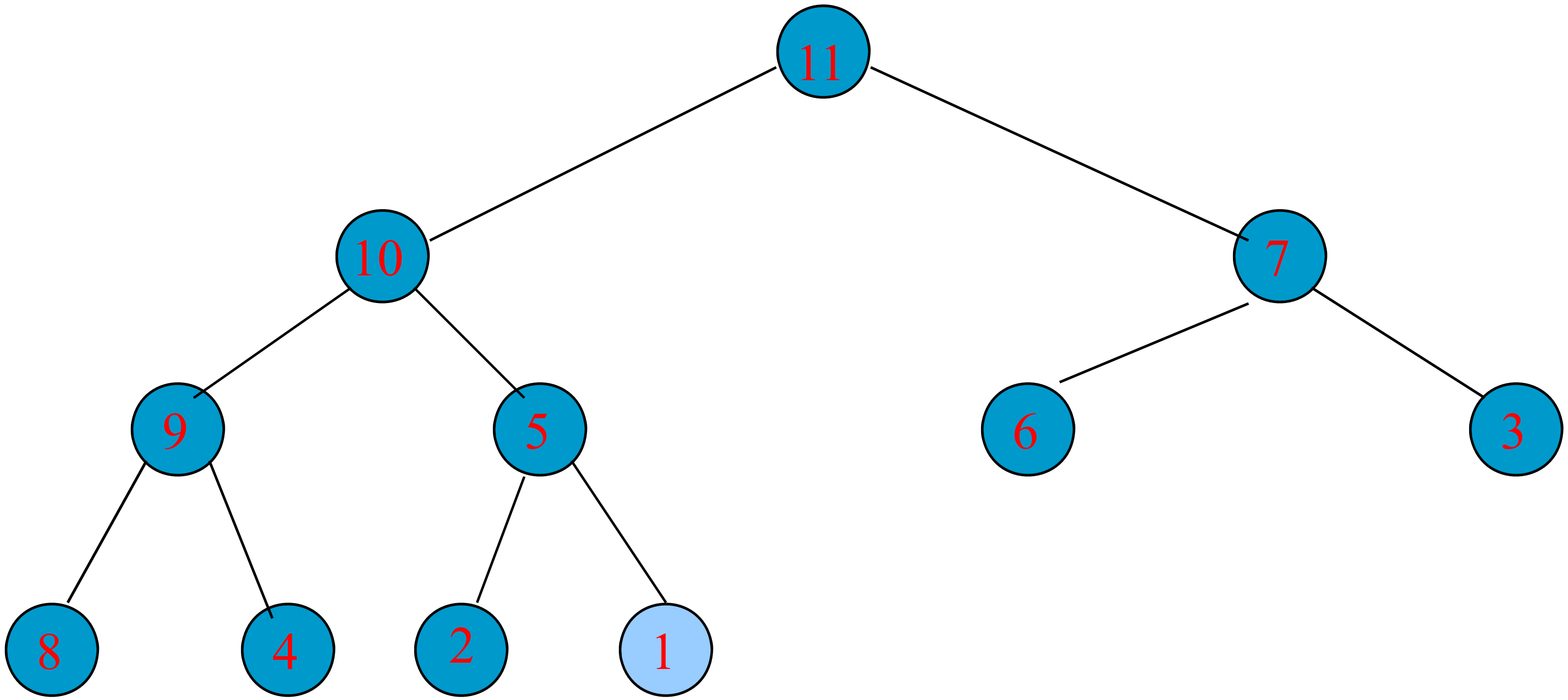
給 1找家

最大堆積之起始



給 1找家

最大堆積之起始

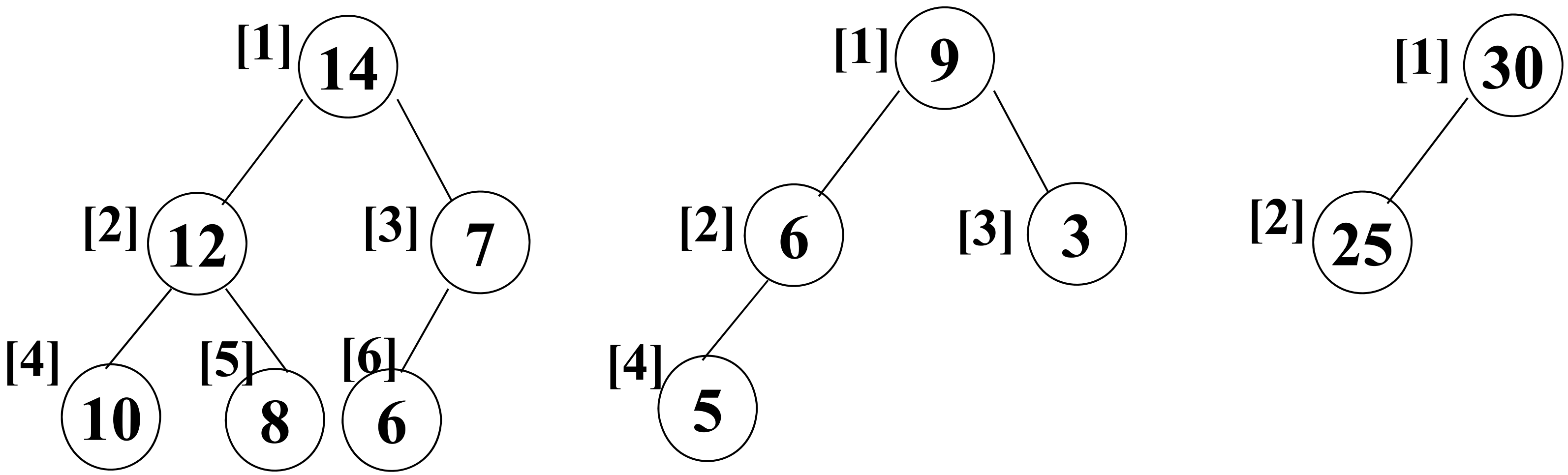


完成！

Heap

- A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.
- A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap;
  - deletion of the largest element from the heap

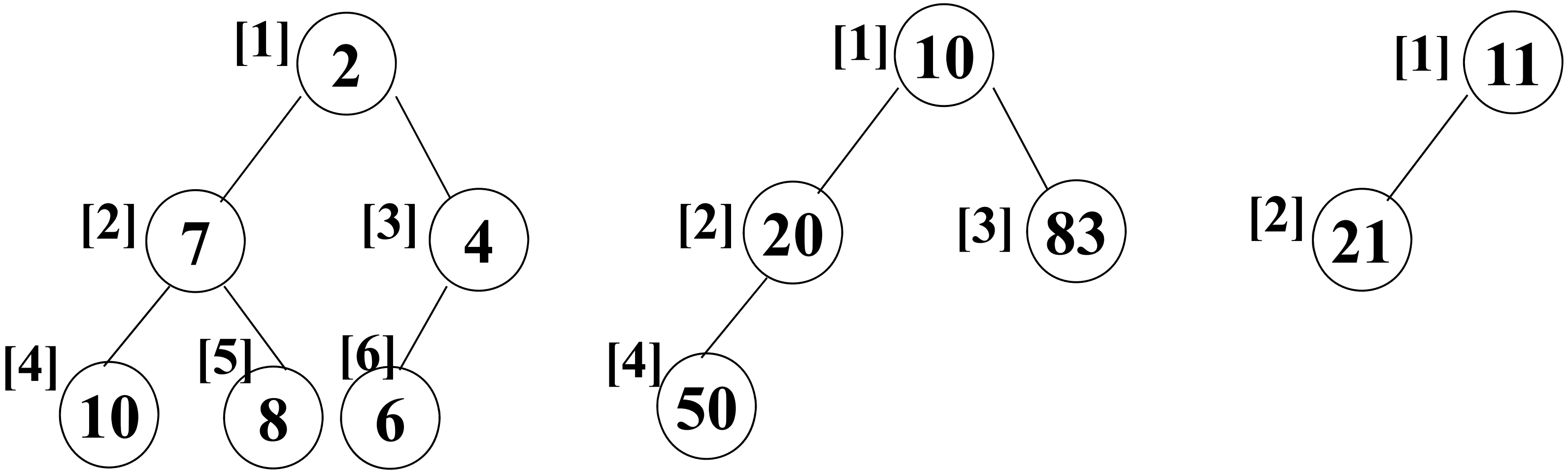
***Figure 5.25:** Sample max heaps (p.219)



Property:

The root of max heap (min heap) contains the largest (smallest).

***Figure 5.26:**Sample min heaps (p.220)



ADT for Max Heap

structure MaxHeap

objects: a complete binary tree of n > 0 elements organized so that the value in each node is at least as large as those in its children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*, *max_size* belong to integer

MaxHeap Create(max_size)::= create an empty heap that can hold a maximum of max_size elements

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap
else return error

Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the largest element in the heap and remove it from the heap
else return error

Application: priority queue

- machine service
  - amount of time (min heap)
  - amount of payment (max heap)
- factory
  - time tag

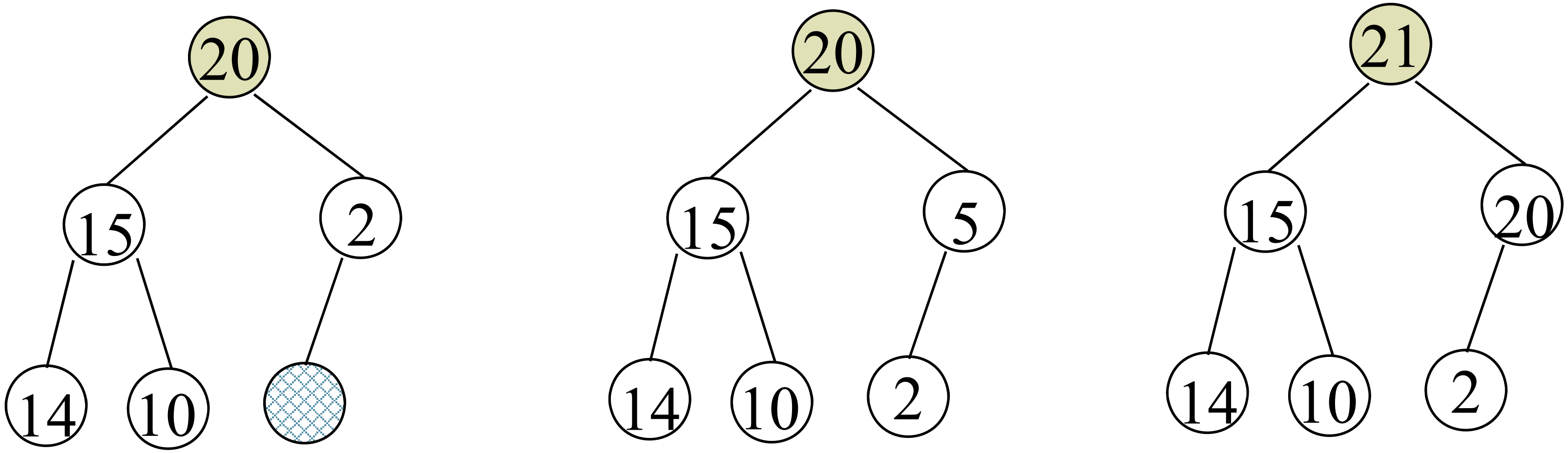
Data Structures

- unordered linked list
- unordered array
- sorted linked list
- sorted array
- heap

*<u>**Figure 5.27:** Priority queue representations (p.221)</u>

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	O(n)	$\Theta(1)$
Sorted linked list	O(n)	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

Example of Insertion to Max Heap



initial location of new node   insert 5 into heap   insert 21 into heap

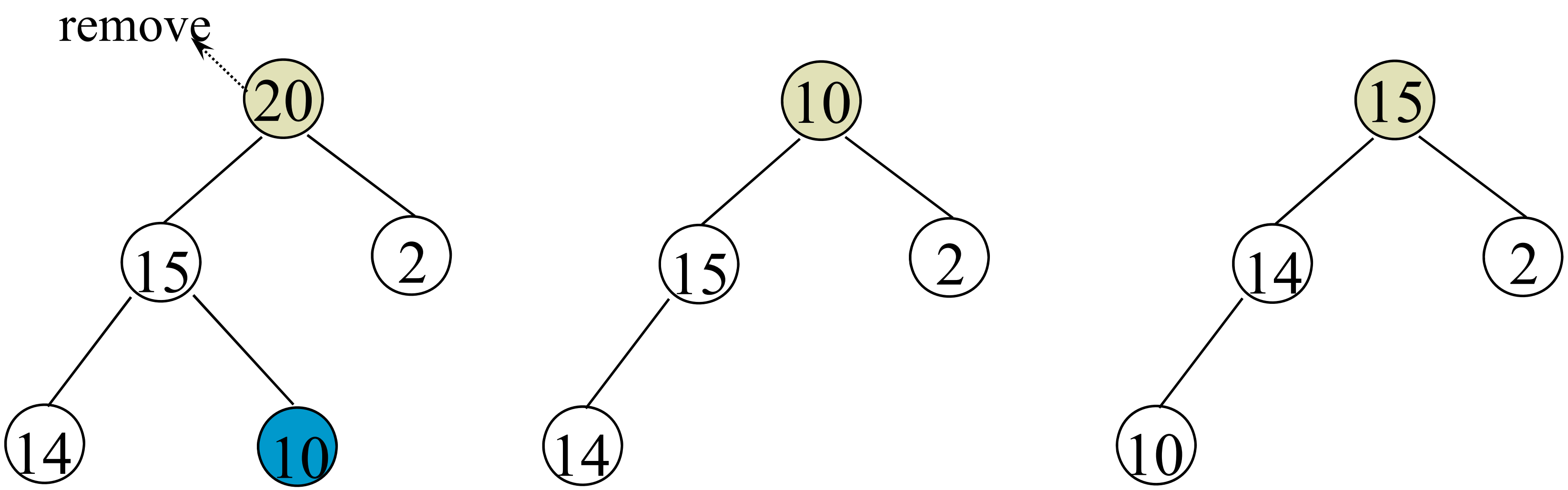
Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, “the heap is full.\n”);
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

$2^k-1=n \implies k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Example of Deletion from Max Heap



Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
```

```
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
        (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

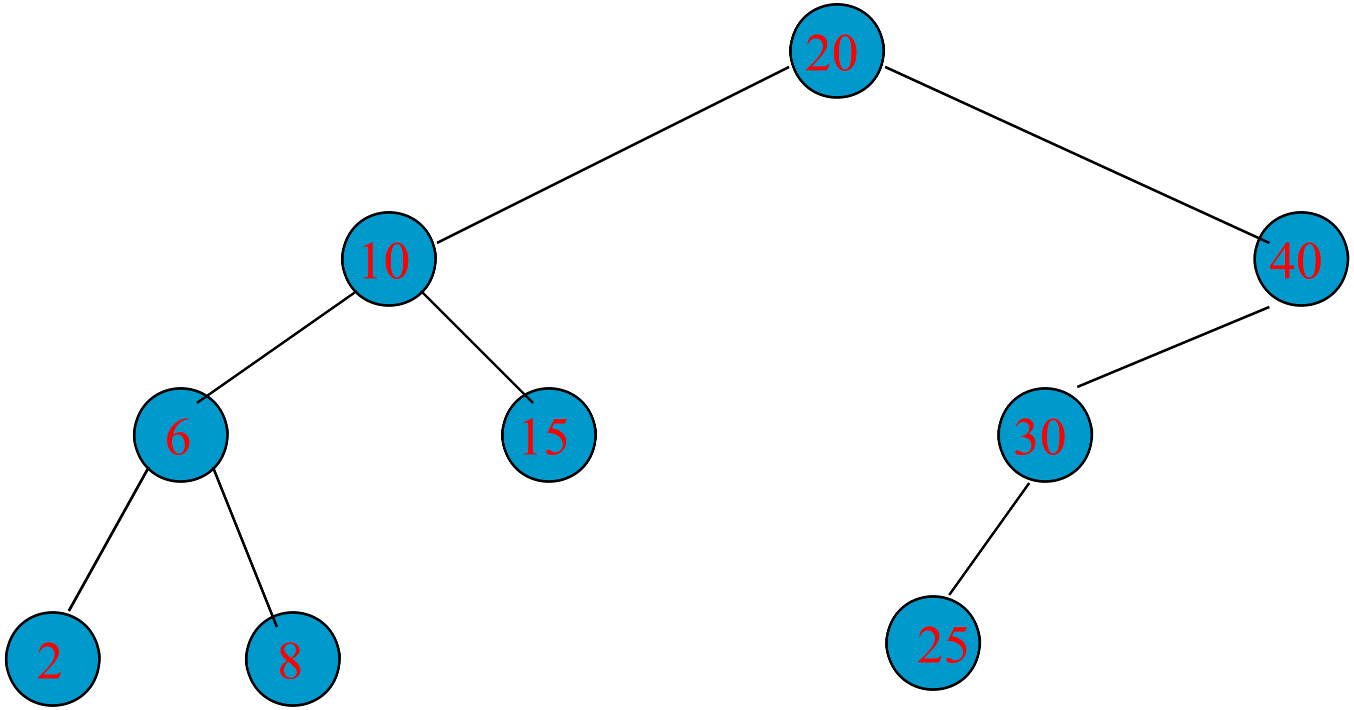
作業

pp. 229: ex1

5.7 二元搜尋樹(Binary Search Tree)

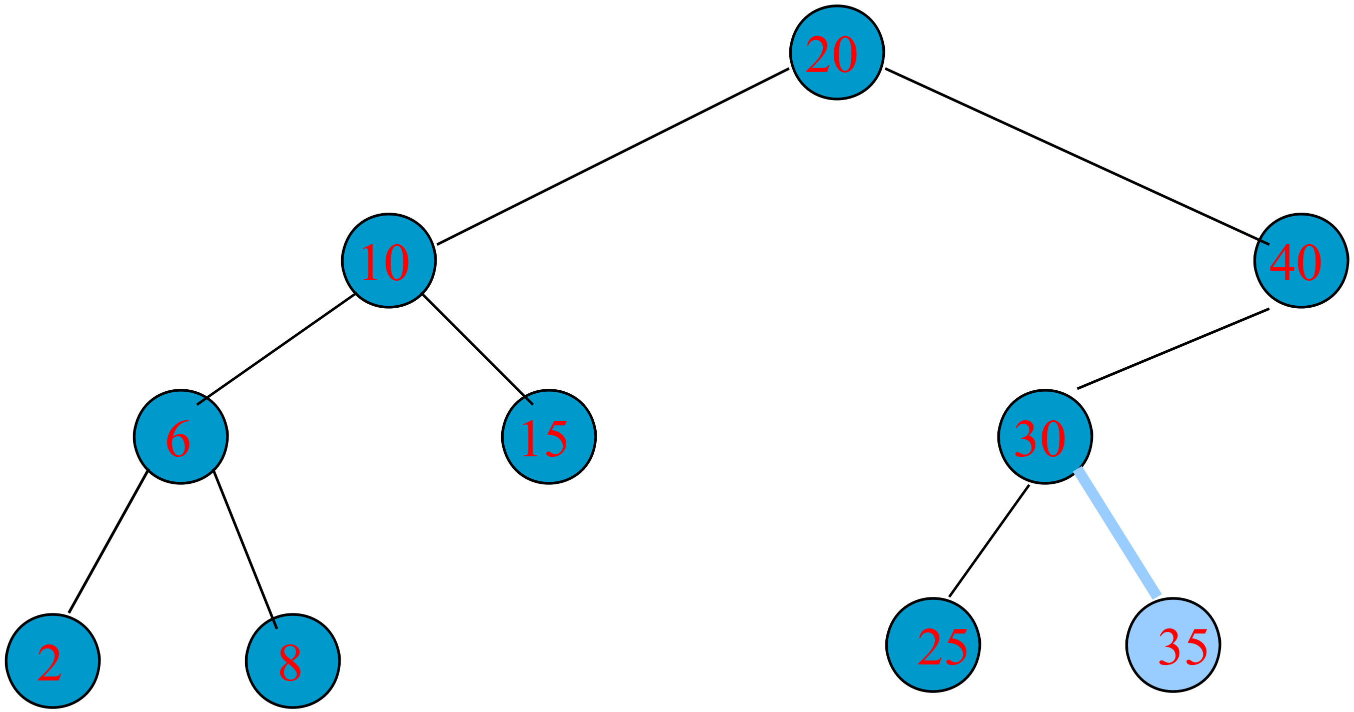
- 二元樹
- 每一個元素有一鍵值，而且每一個元素的鍵值都不相同，即鍵值是唯一的。
- 非空的左子樹上的鍵值必須小於該子樹的根節點之鍵值。
- 在非空的右子樹上的鍵值必須大於在該子樹的根節點之鍵值。
- 左子樹和右子樹也都是二元搜尋樹。

二元搜尋樹



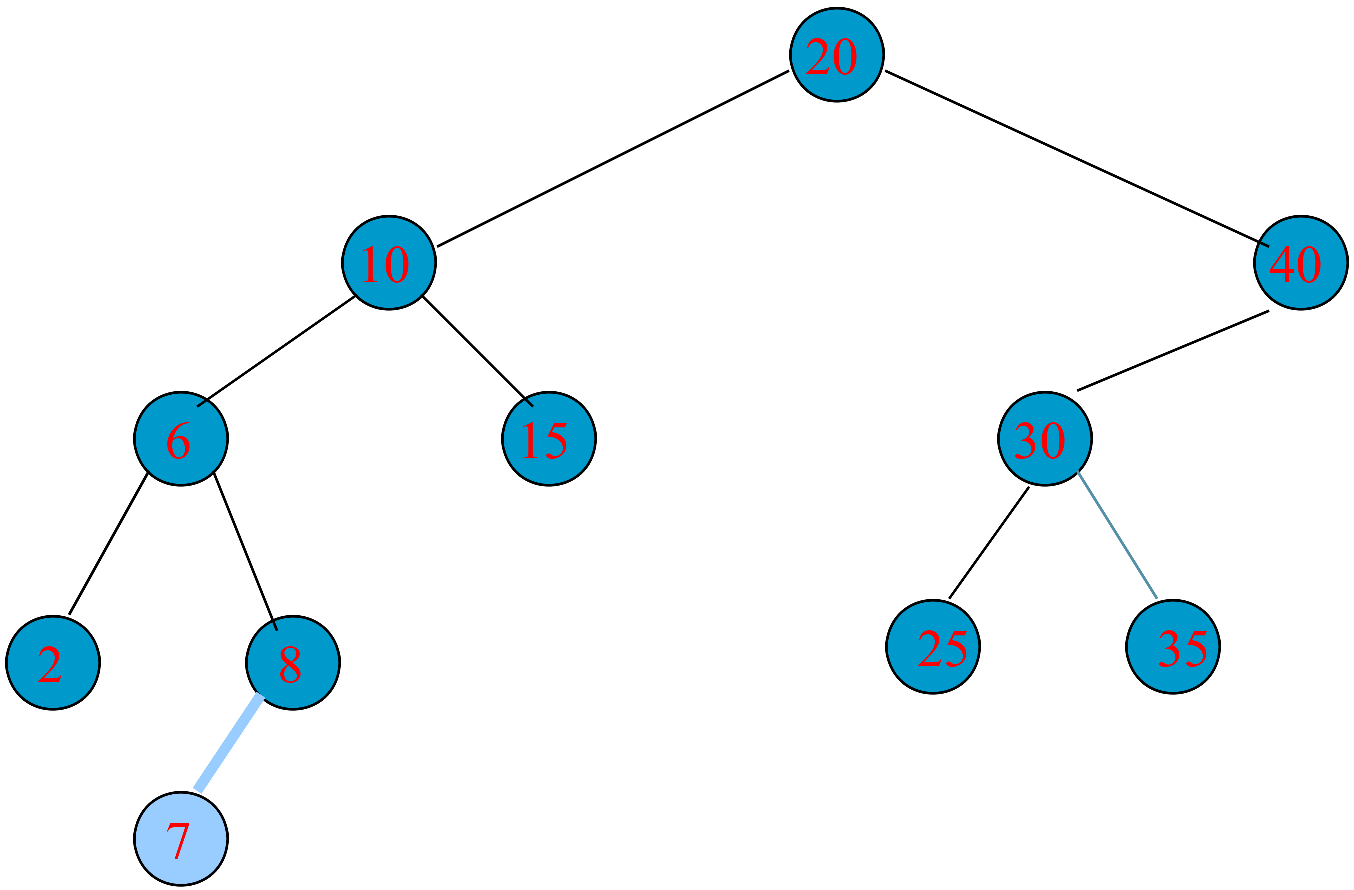
只顯示鍵值

加入一個元素



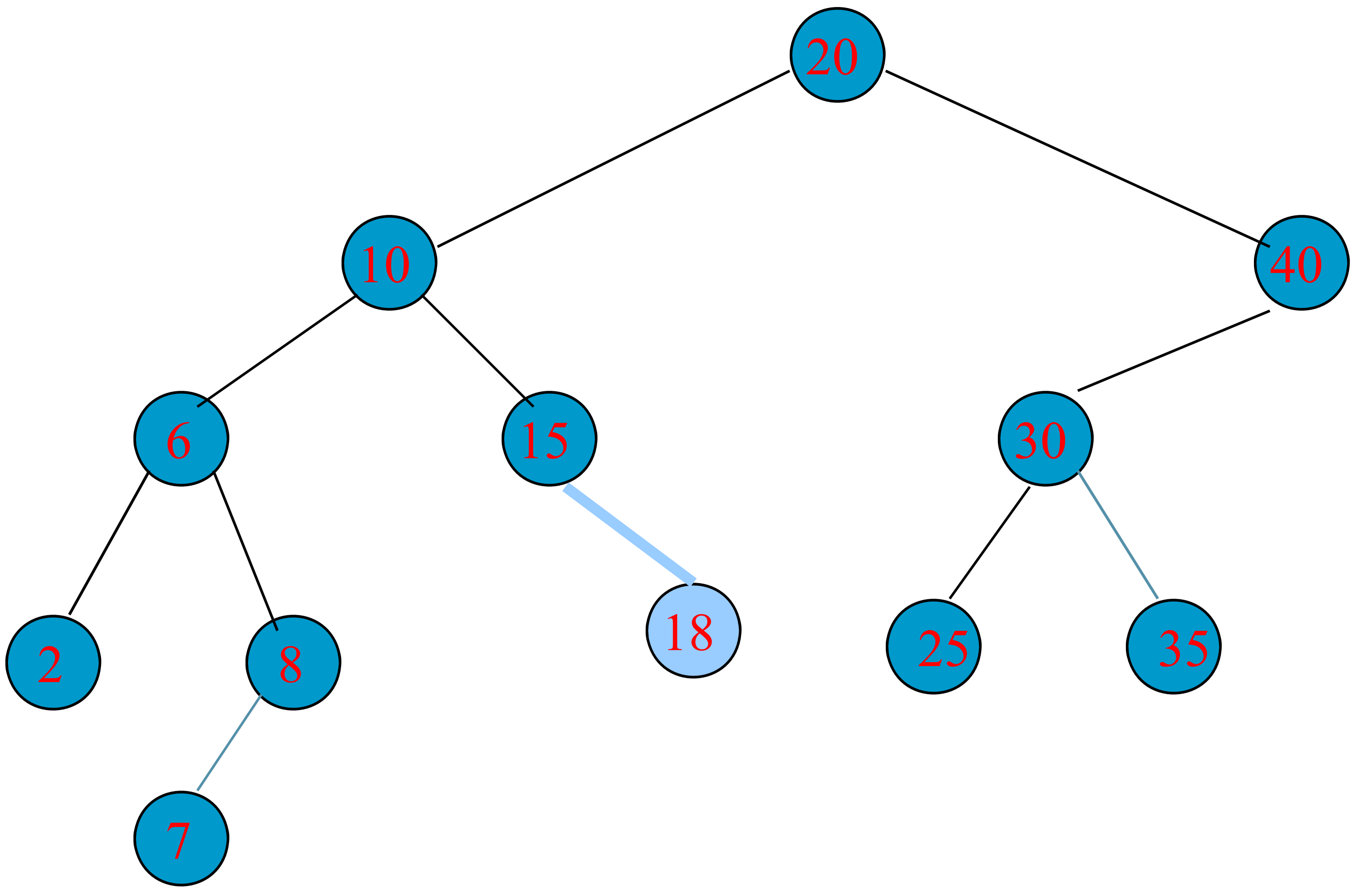
加入一個鍵值為 35 的元素

加入一個元素



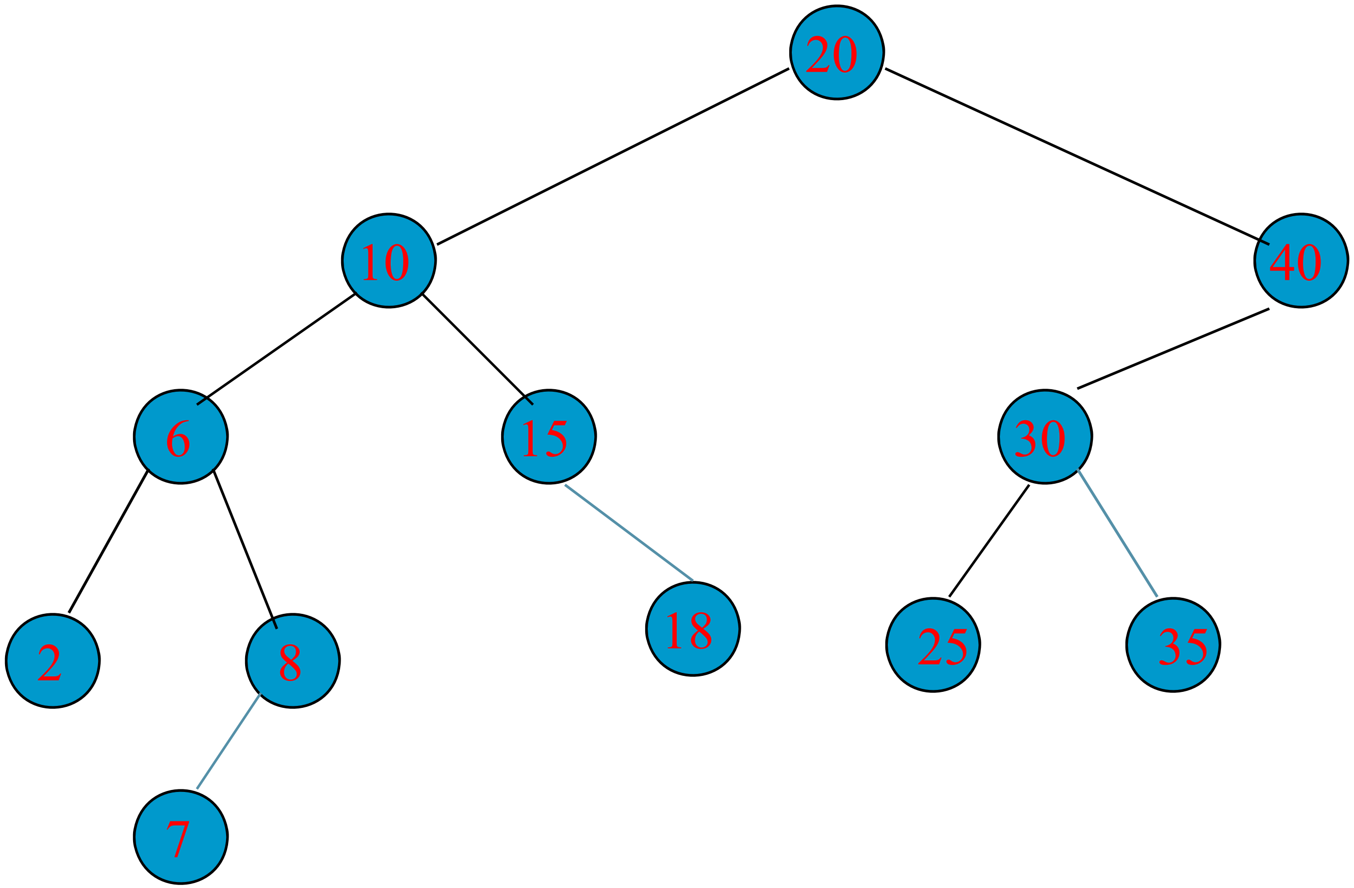
加入一個鍵值為 7 的元素

加入一個元素



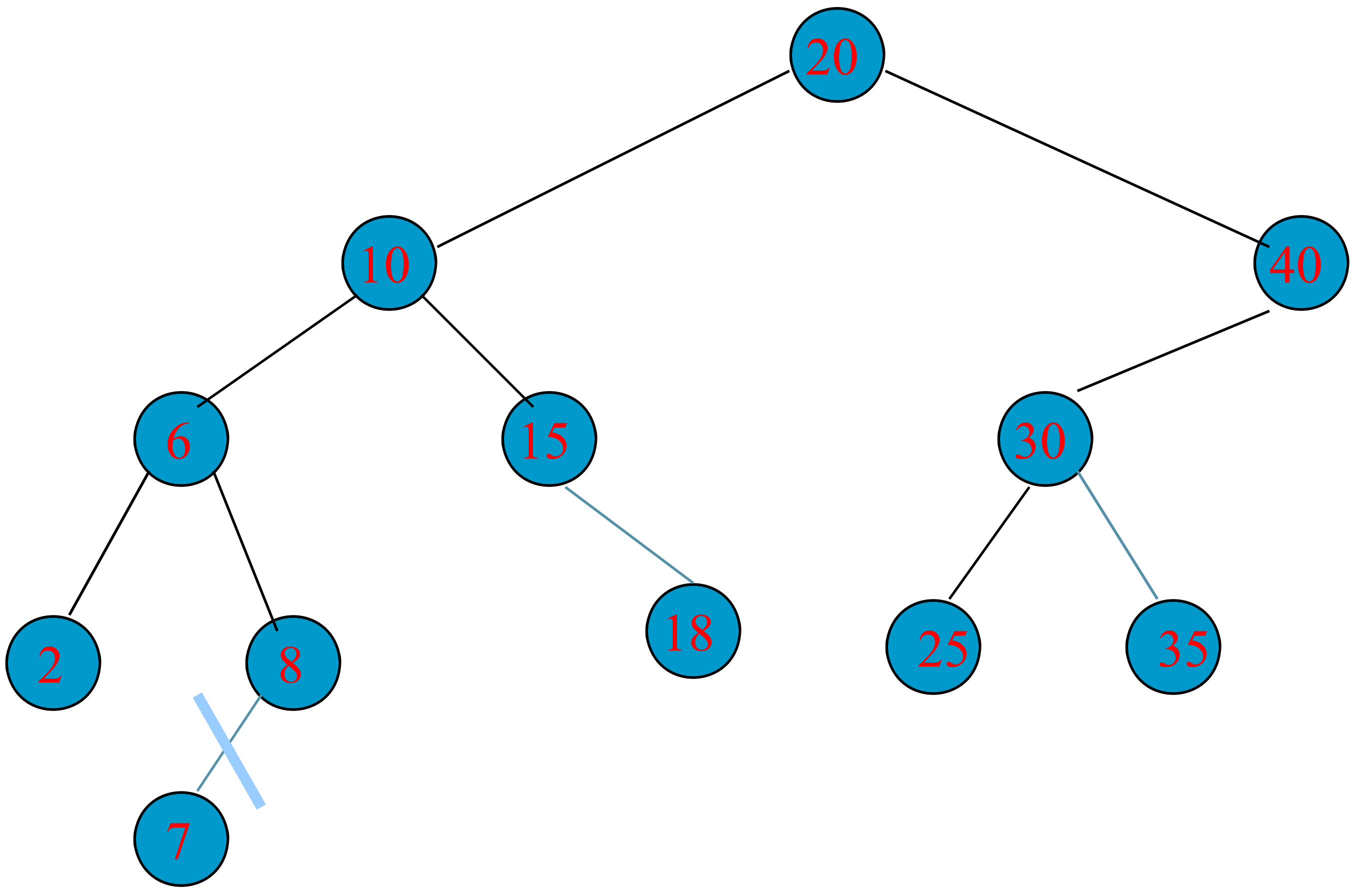
加入一個鍵值為 18 的元素

加入一個元素



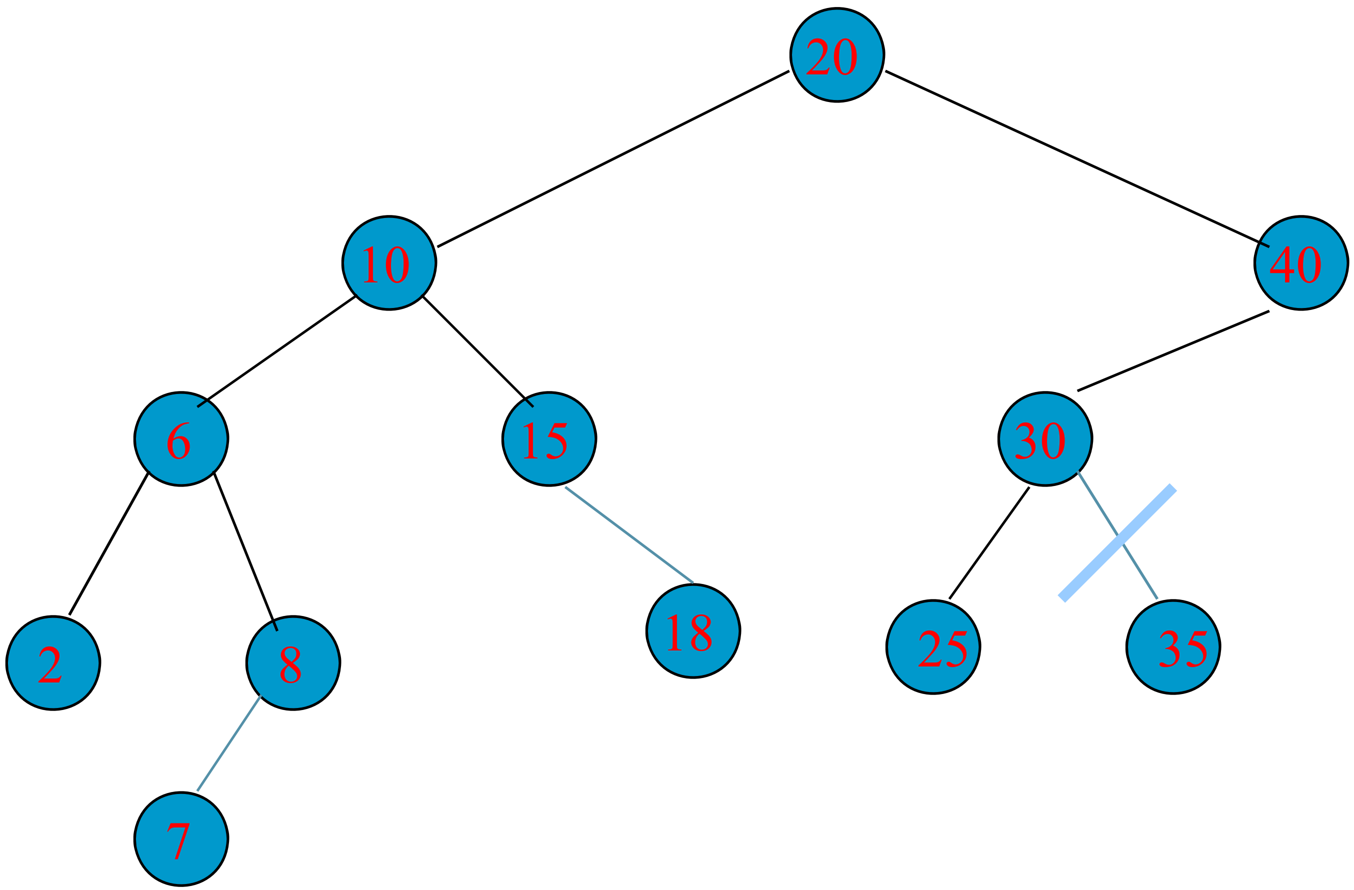
複雜度為 O(height)

刪除一個元素



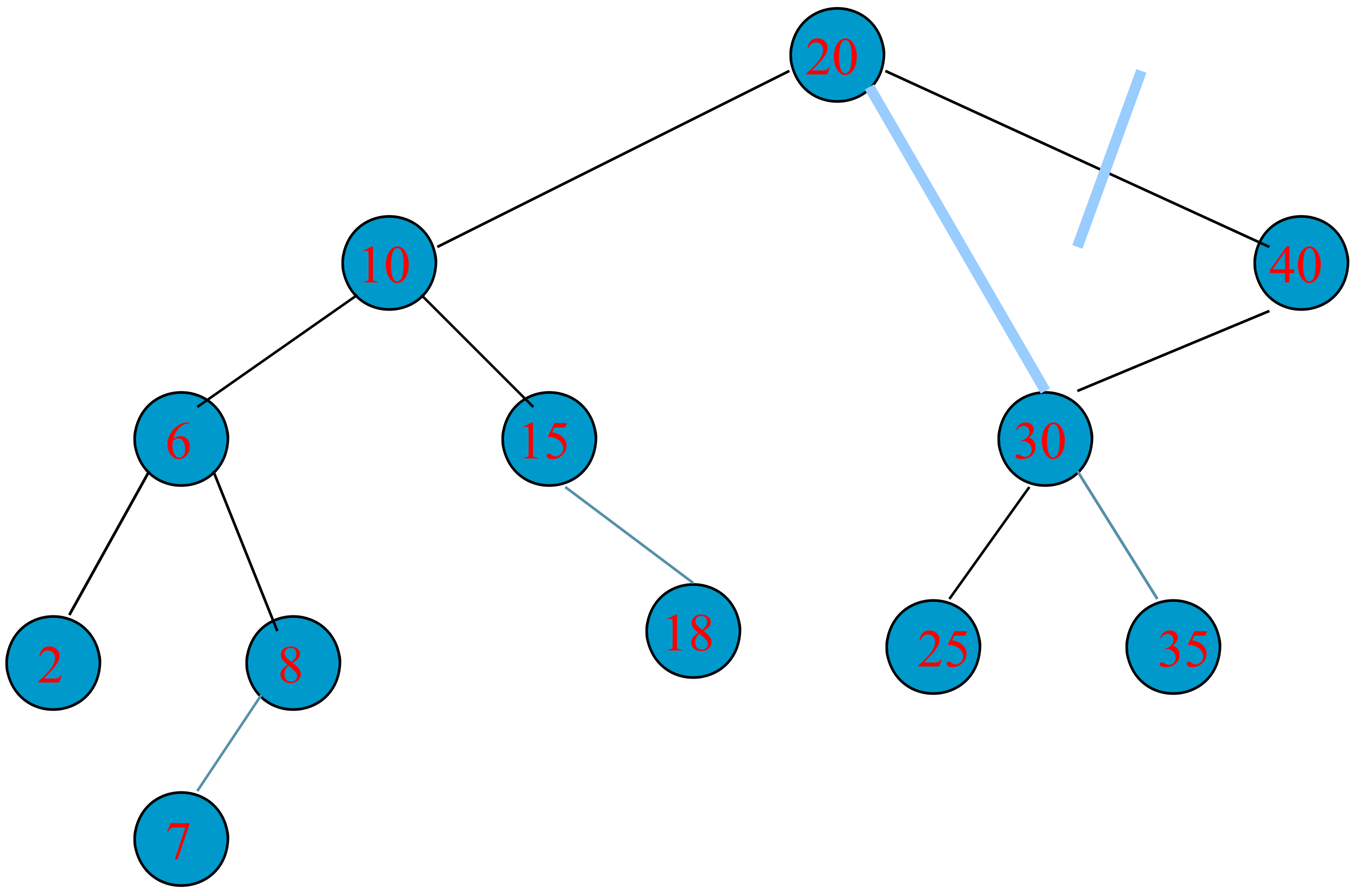
刪除一個鍵值為 7 的樹葉

刪除一個元素



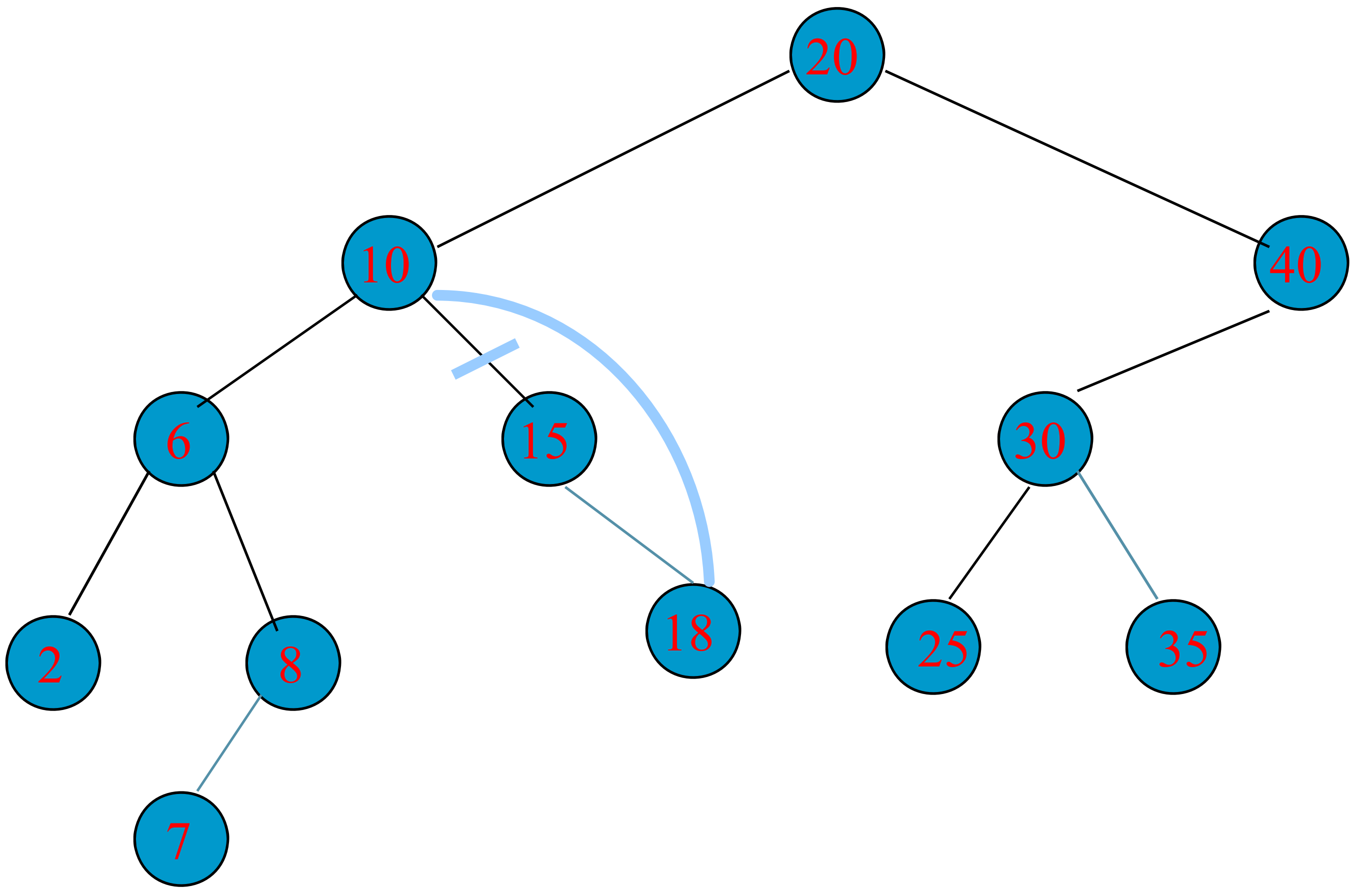
刪除一個鍵值為 35 的樹葉

刪除一個元素



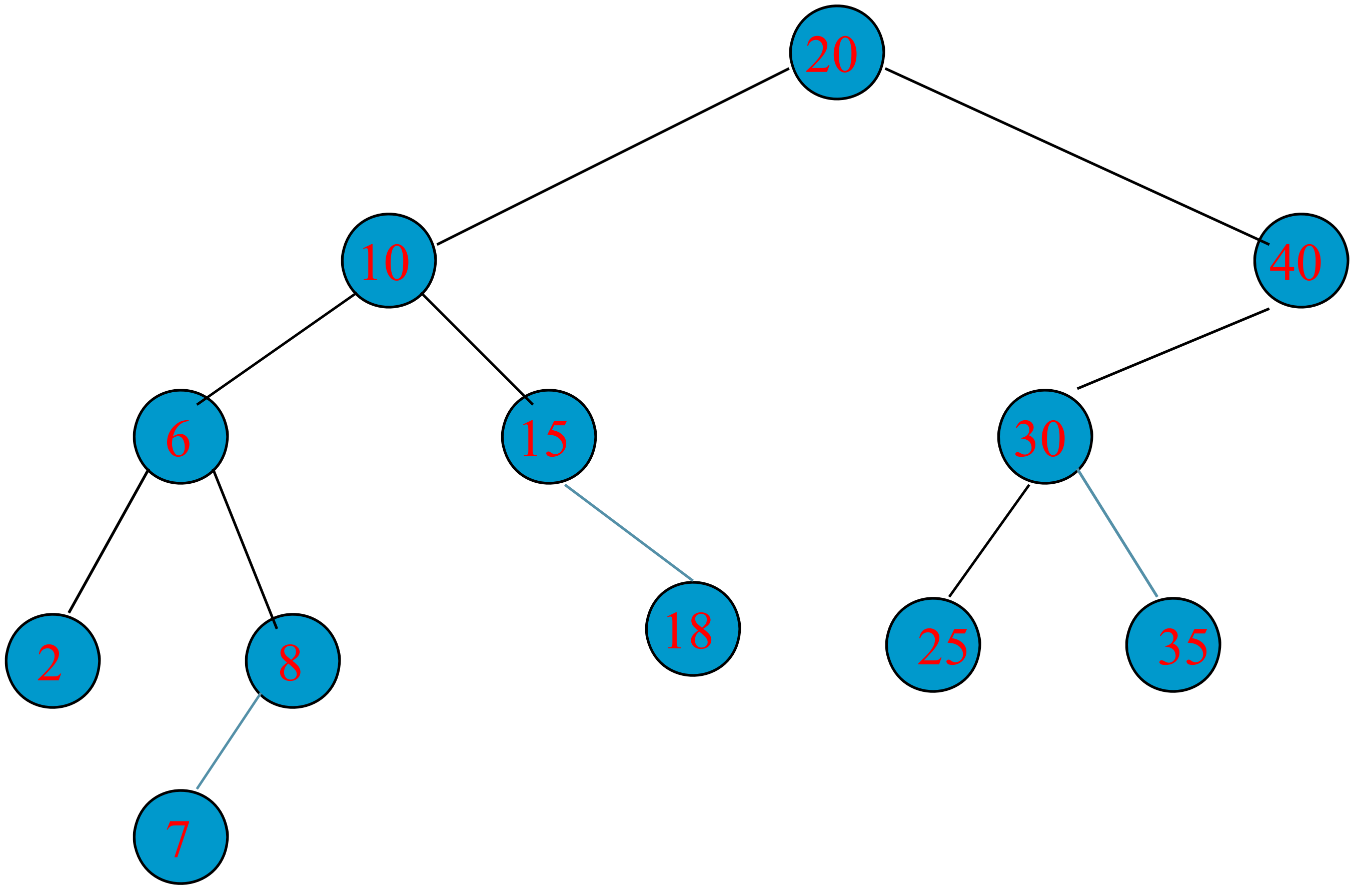
刪除一個鍵值為 40 而且分支度為 1 的節點

刪除一個元素



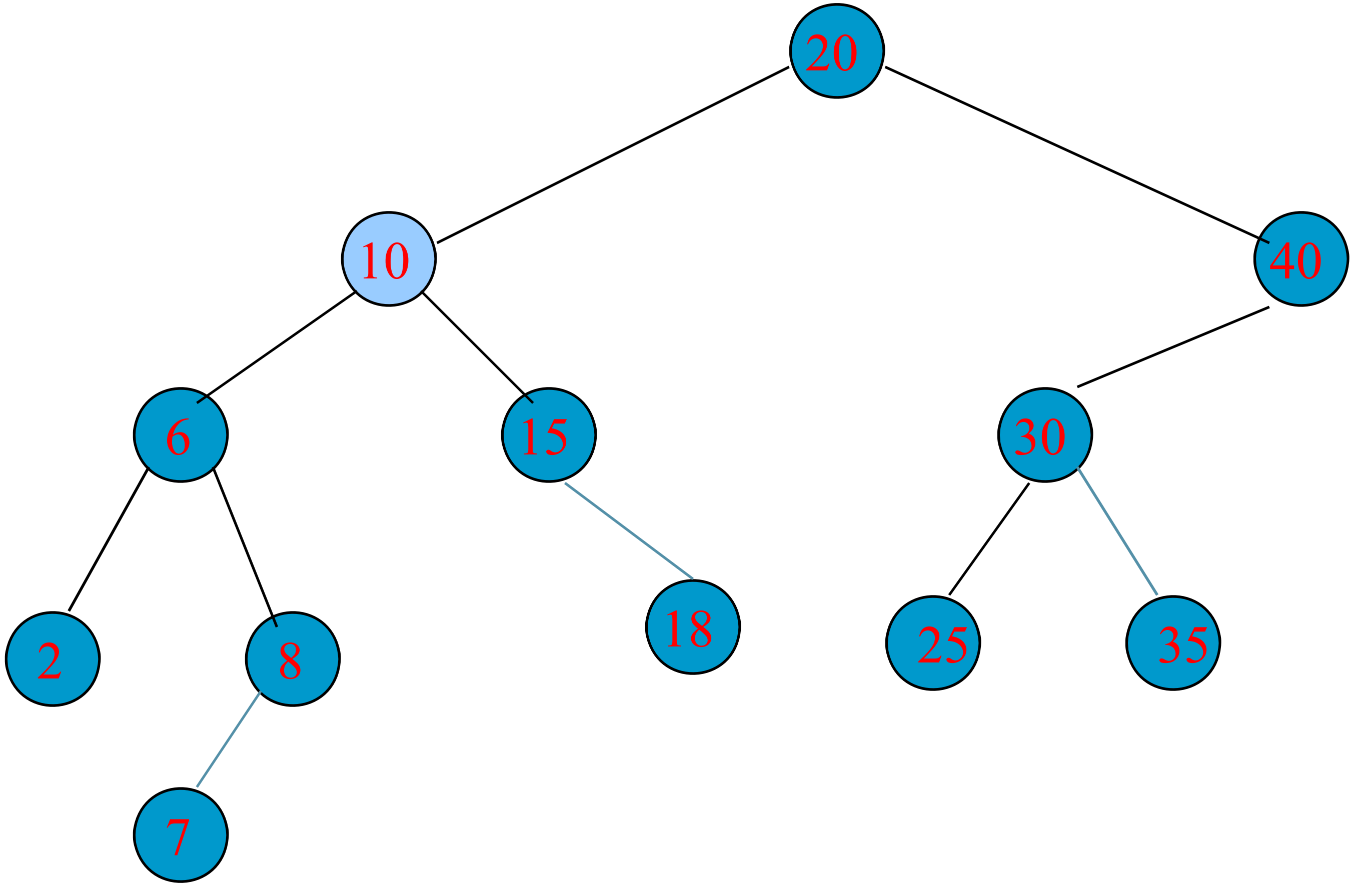
刪除一個鍵值為 15 而且分支度為 1 的節點

刪除一個元素



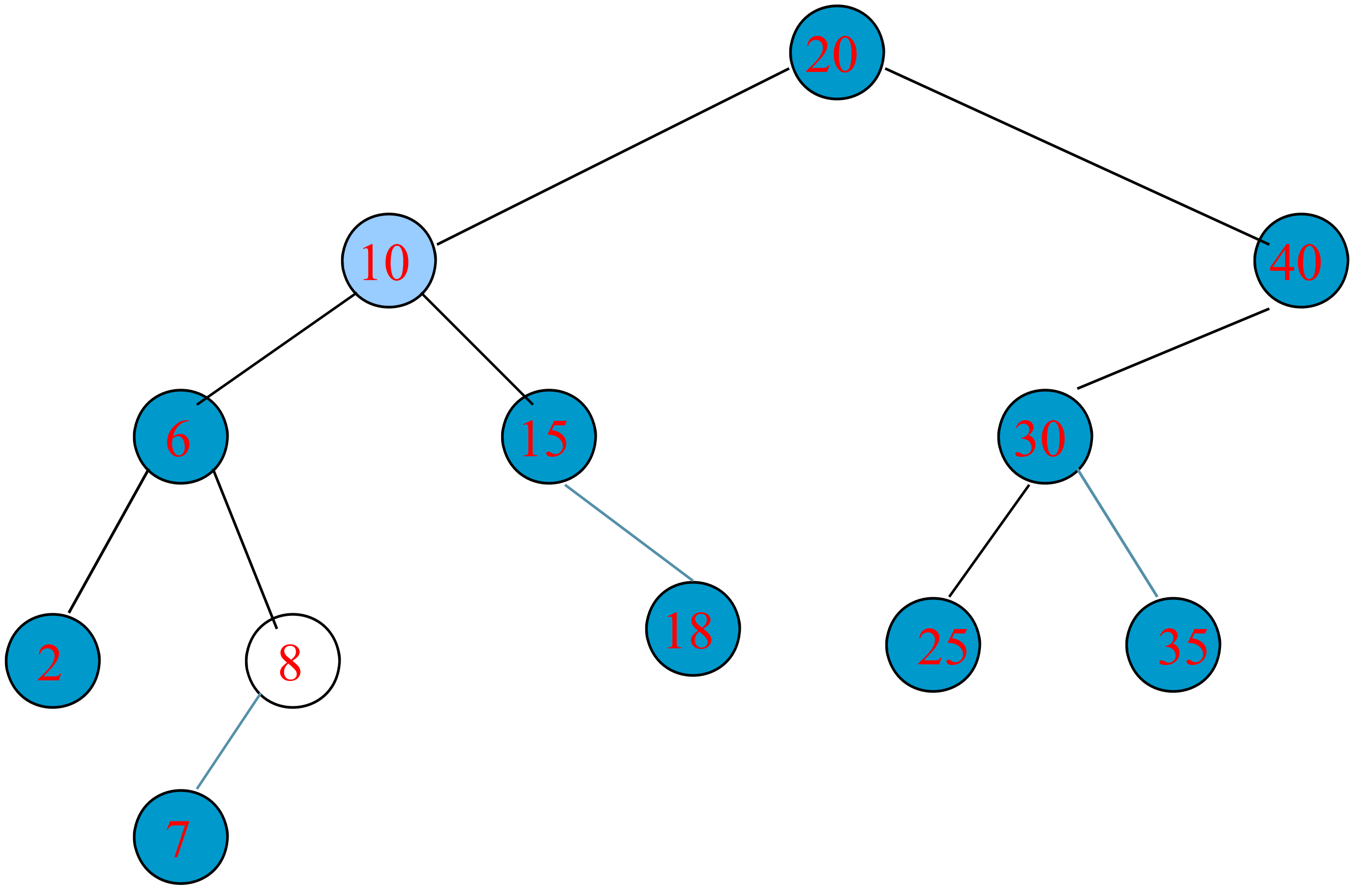
刪除一個鍵值為 10 而且分支度為 2 的節點

刪除一個元素



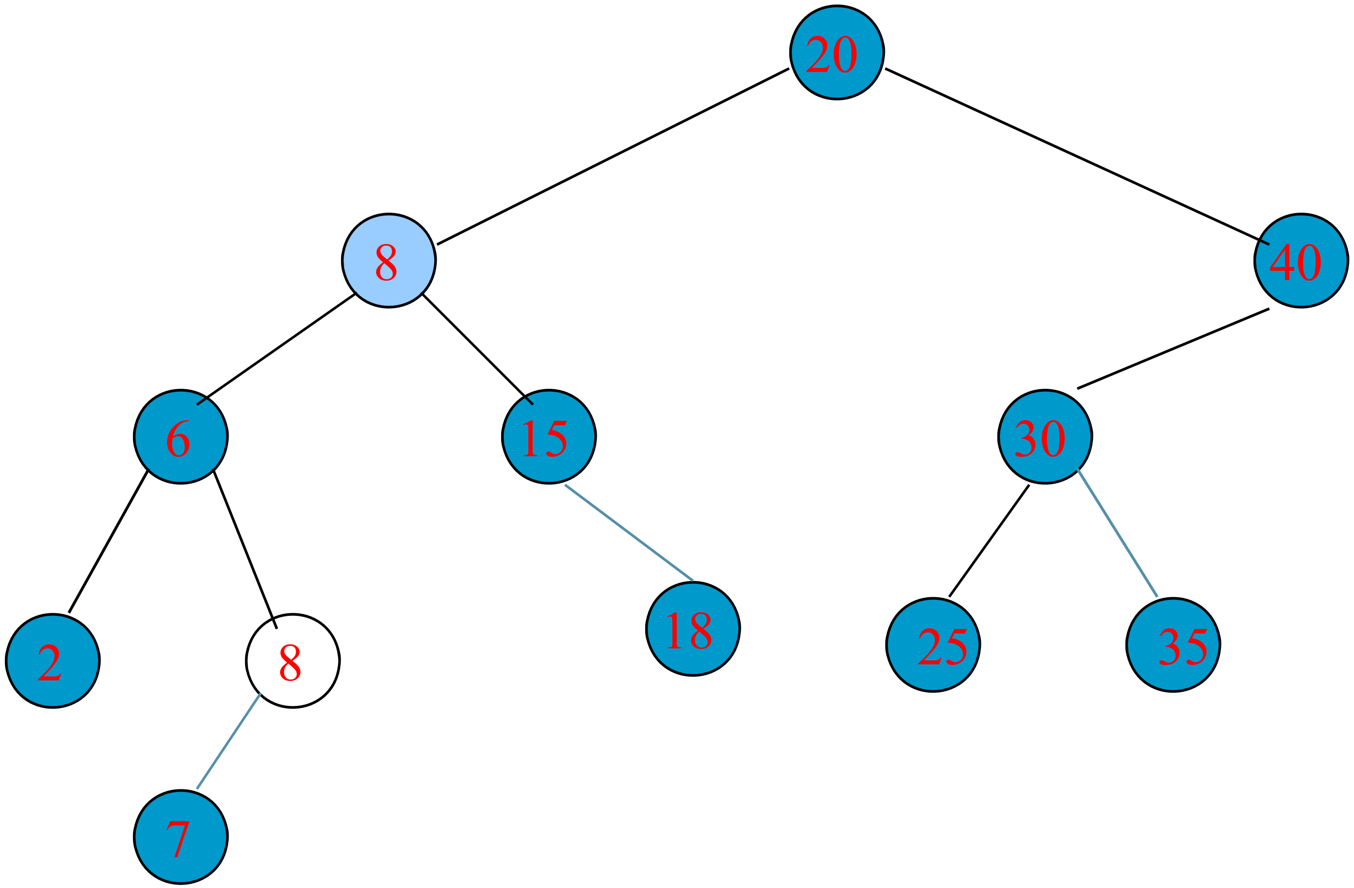
跟左子樹中具有最大值的節點或者右子樹中含最小值的節點互換

刪除一個元素



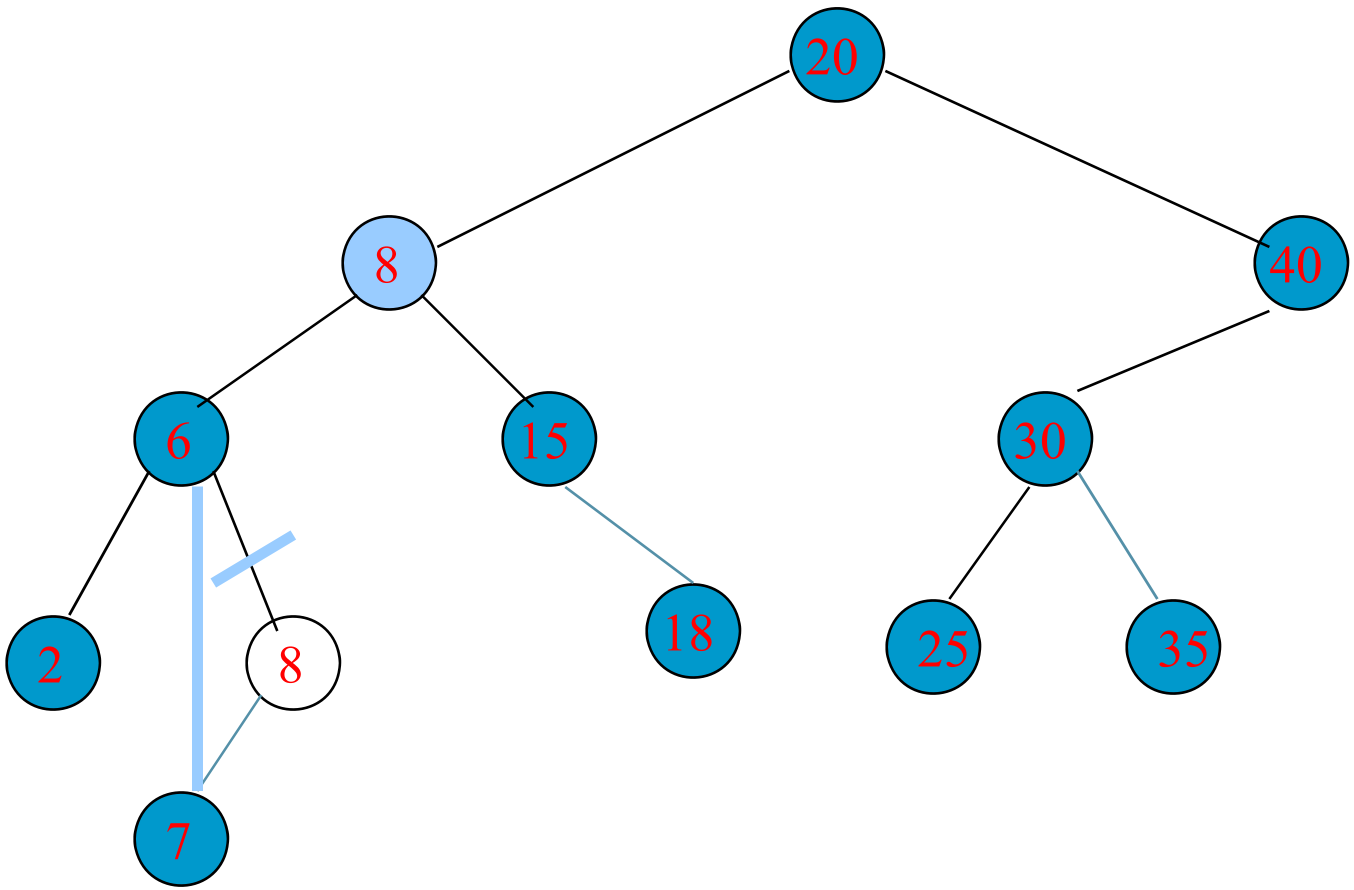
跟左子樹中具有最大值的節點或者右子樹中含最小值的節點互換

刪除一個元素



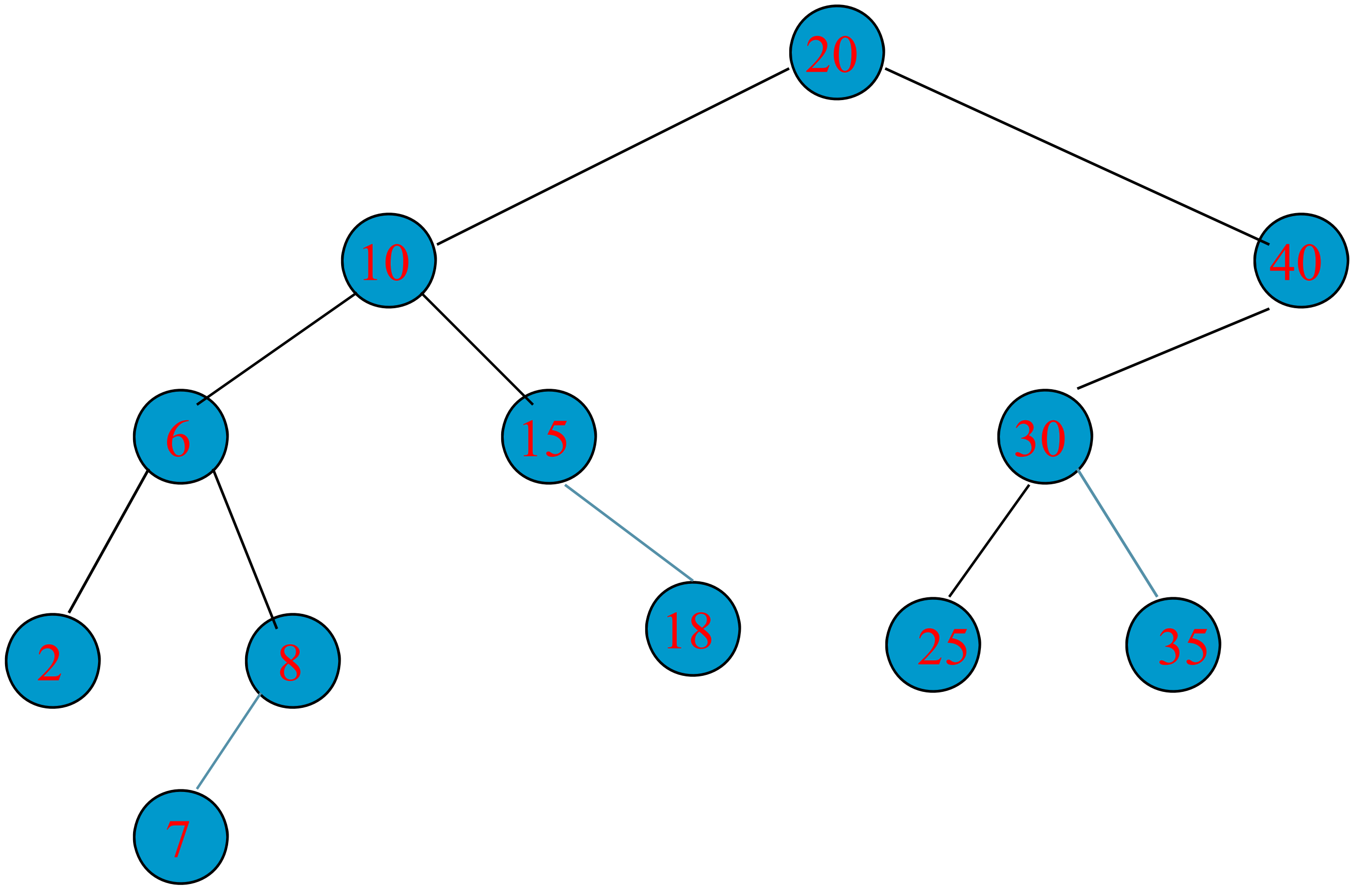
跟左子樹中具有最大值的節點或者右子樹中含最小值的節點互換

刪除一個元素



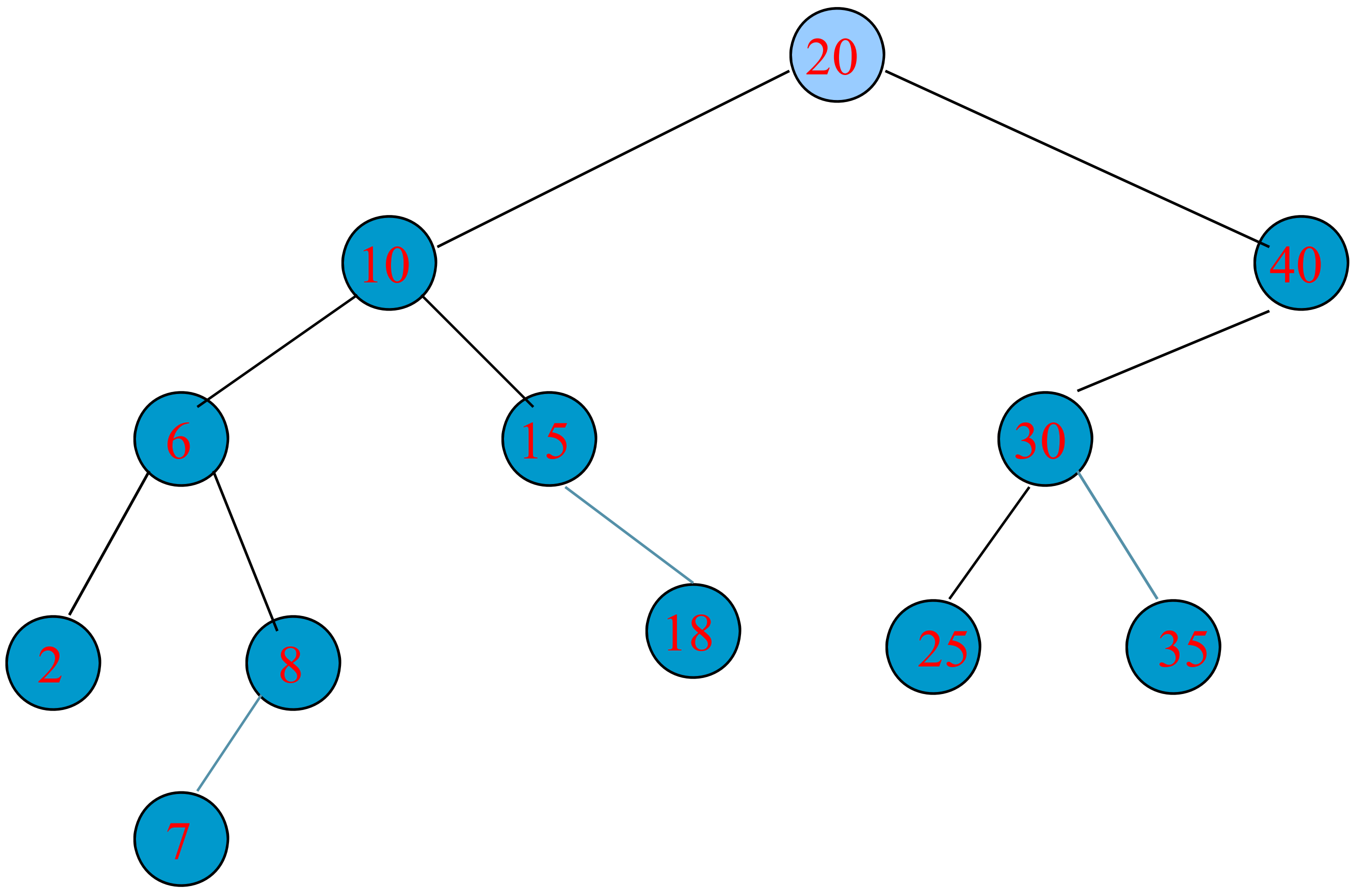
最大鍵值必定在樹葉或者分支度為1的節點

刪除一個元素



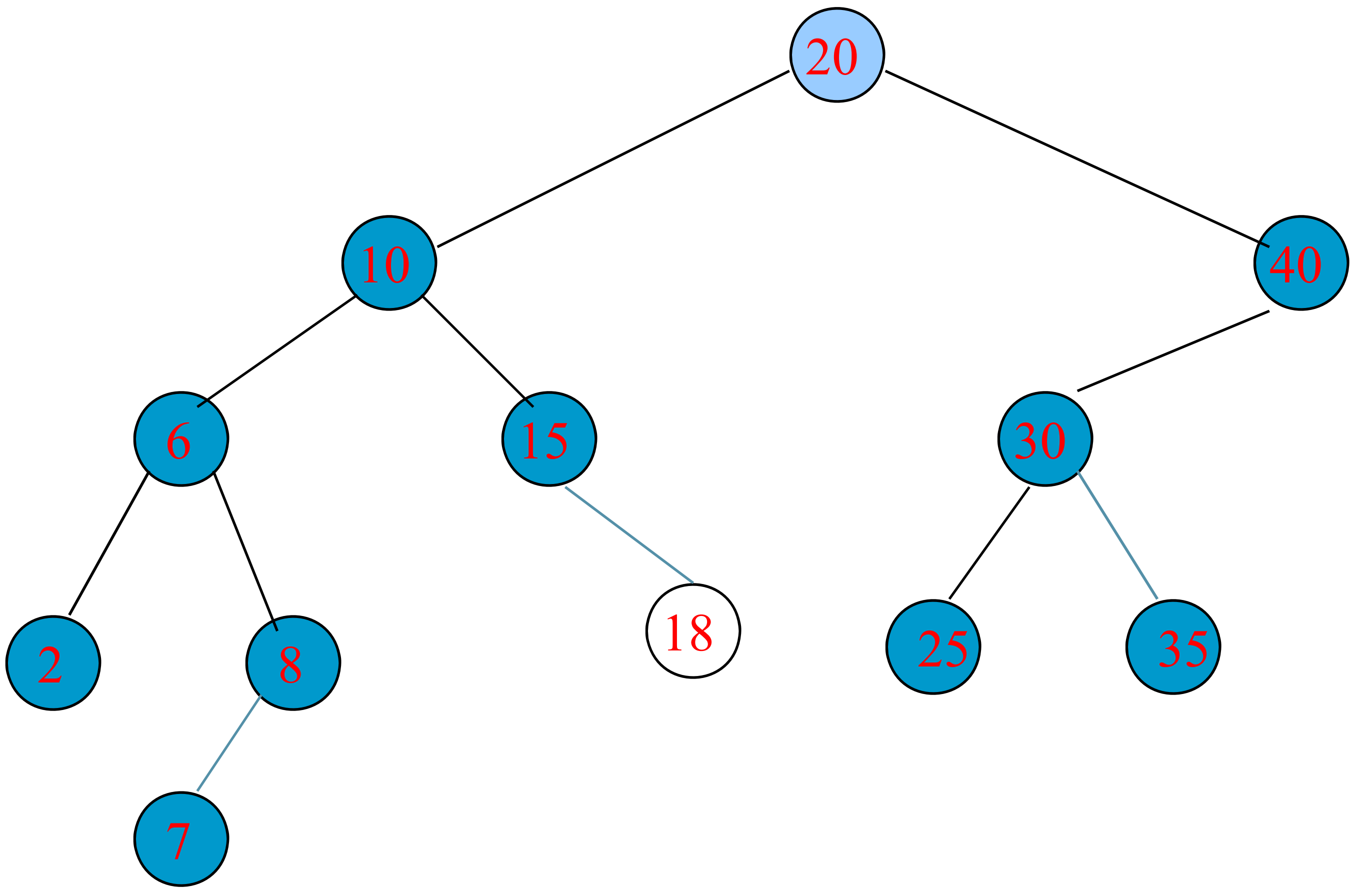
刪除一個鍵值為 20 而且分支度為 2 的節點

刪除一個元素



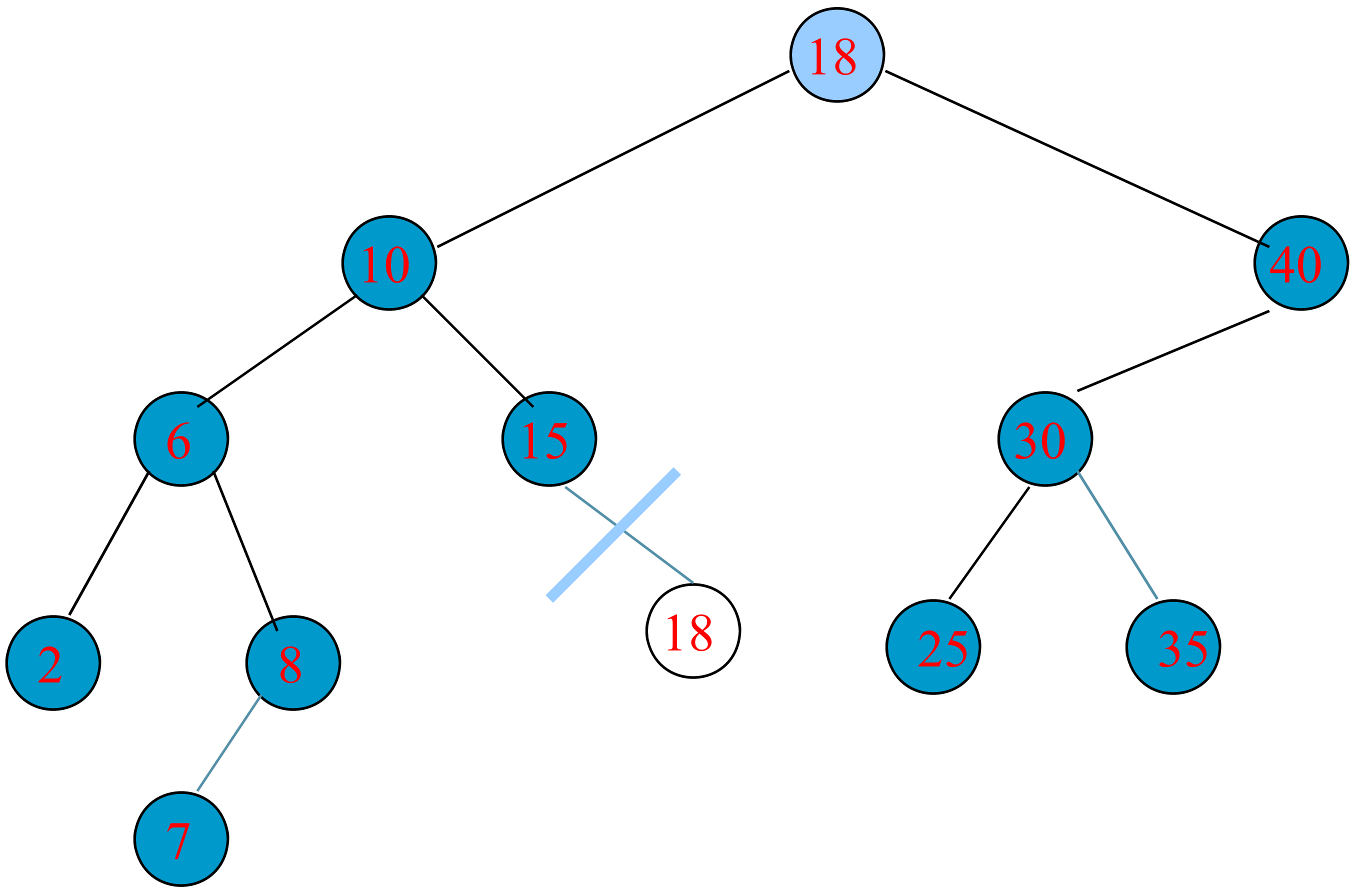
跟左子樹中具有最大值的節點互換

刪除一個元素



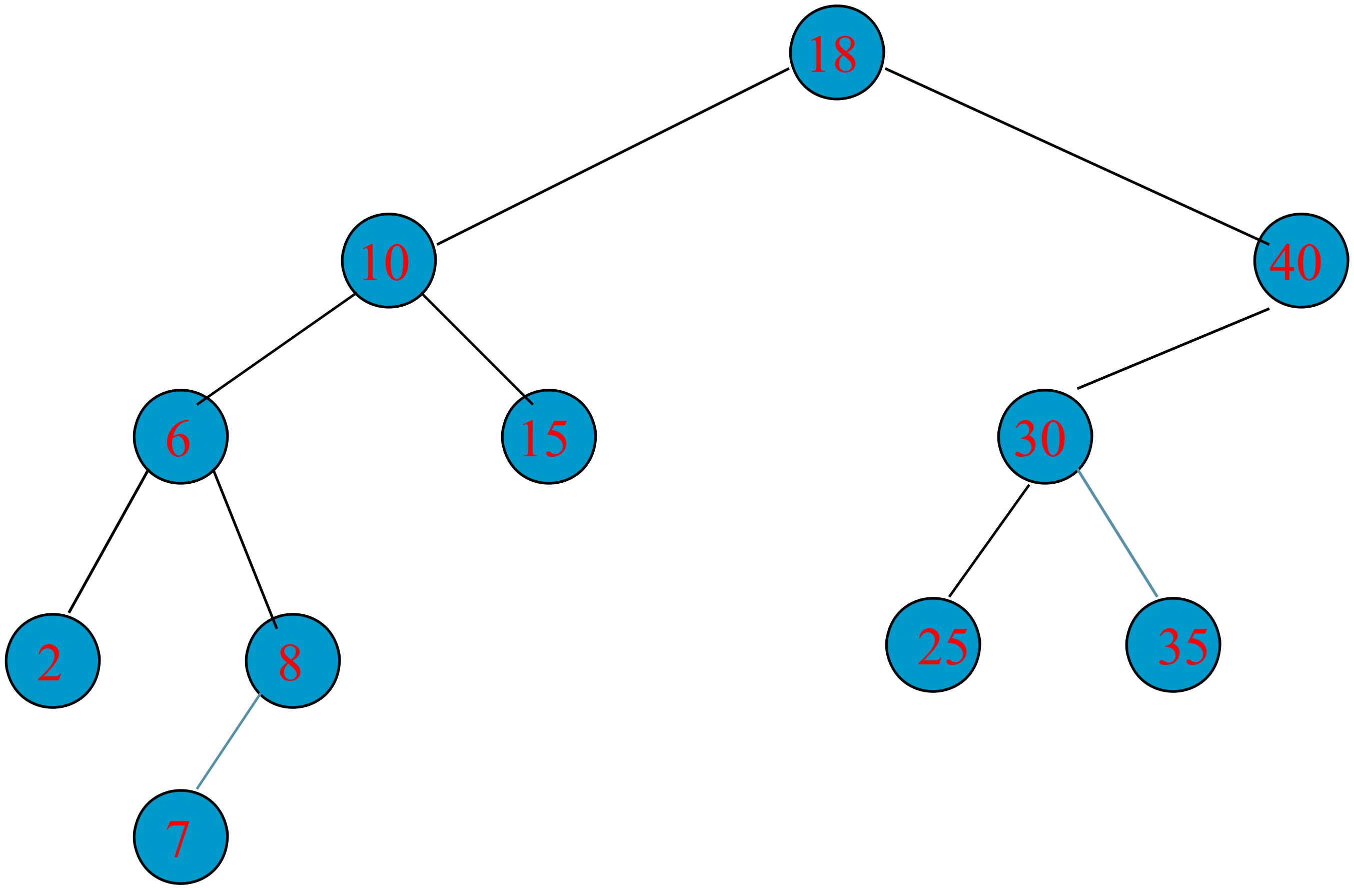
跟左子樹中具有最大值的節點互換

刪除一個元素



跟左子樹中具有最大值的節點互換

刪除一個元素

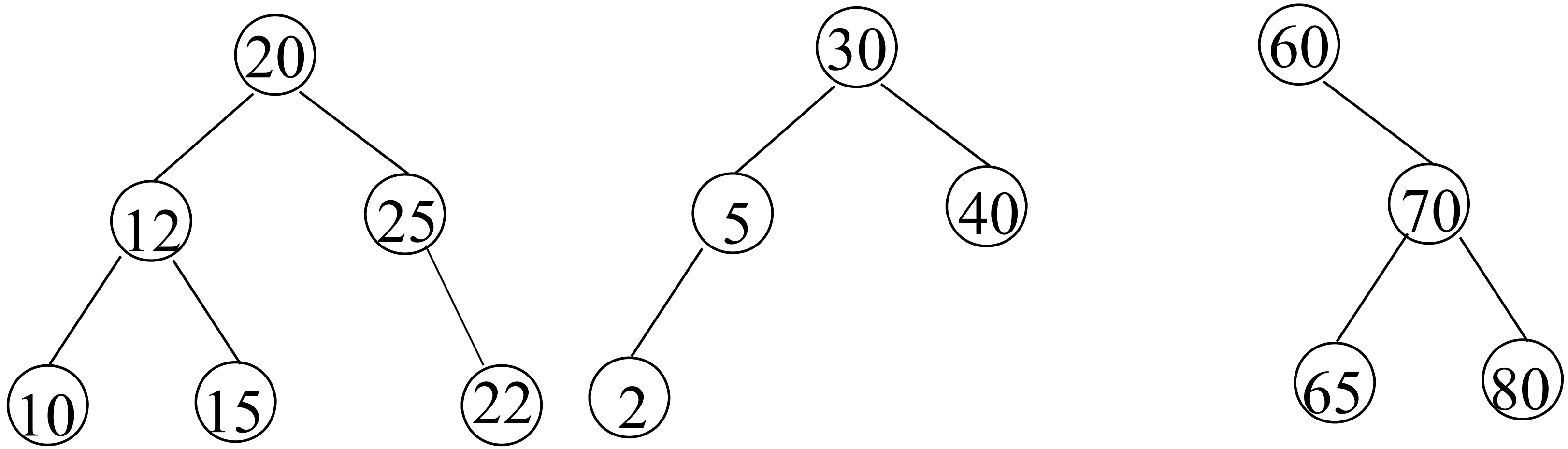


複雜度為 O(height)

Binary Search Tree

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

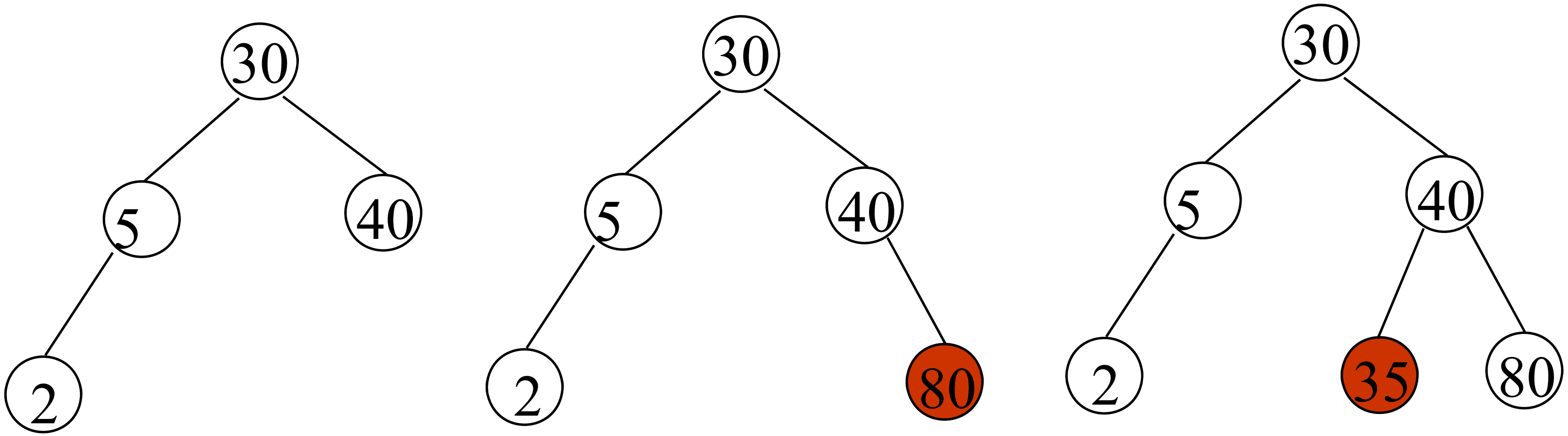
```
tree_pointer search(tree_pointer root,
                    int key)
{
/* return a pointer to the node that
  contains key. If there is no such
  node, return NULL */

   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
       return search(root->left_child,
                     key);
   return search(root->right_child,key);
}
```

Another Searching Algorithm

```
tree_pointer search2(tree_pointer tree,
  int key)
{
  while (tree) {
    if (key == tree->data) return tree;
    if (key < tree->data)
        tree = tree->left_child;
    else tree = tree->right_child;
  }
  return NULL;
}
```

O(h)

Insert Node in Binary Search Tree

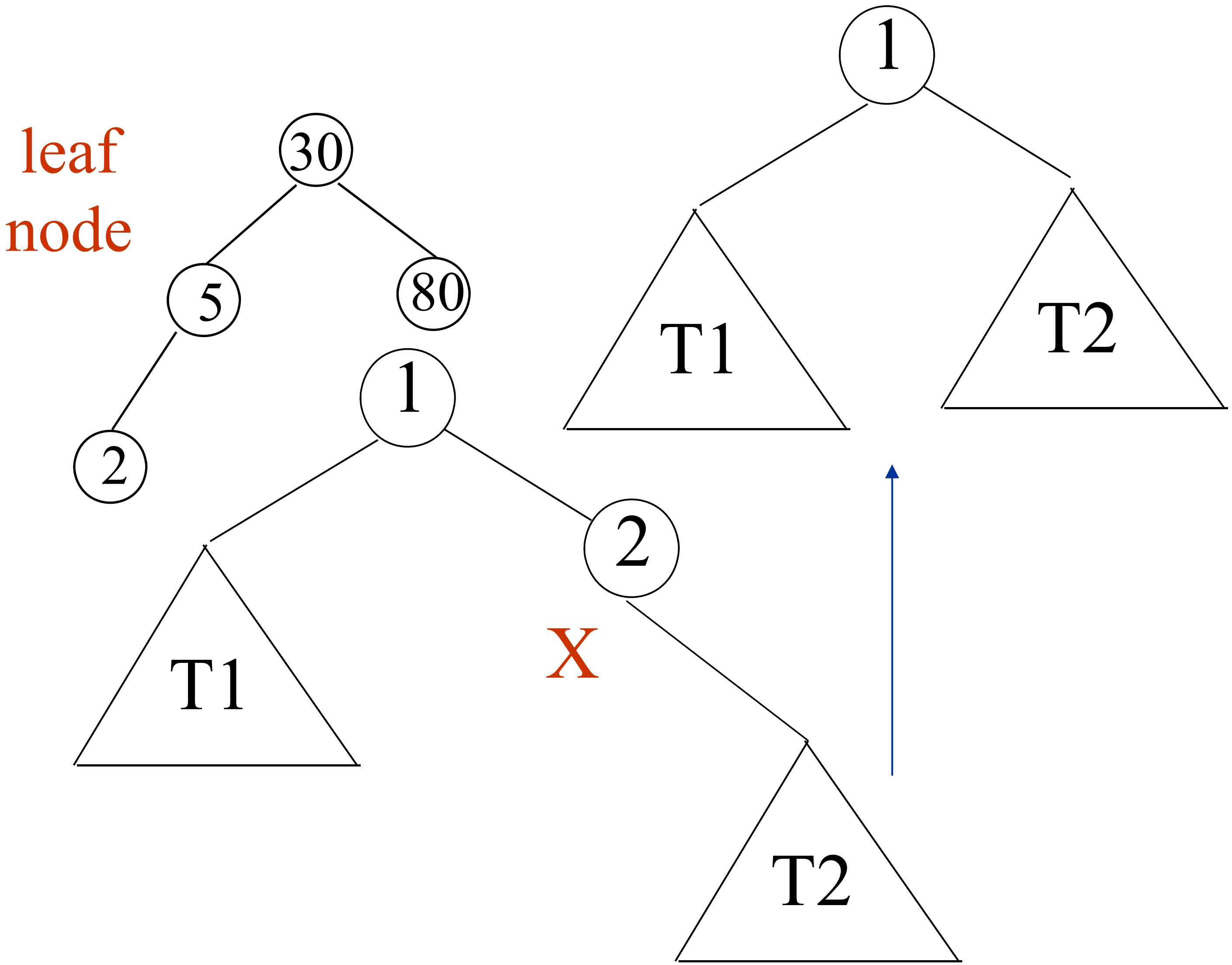


Insert 80 Insert 35

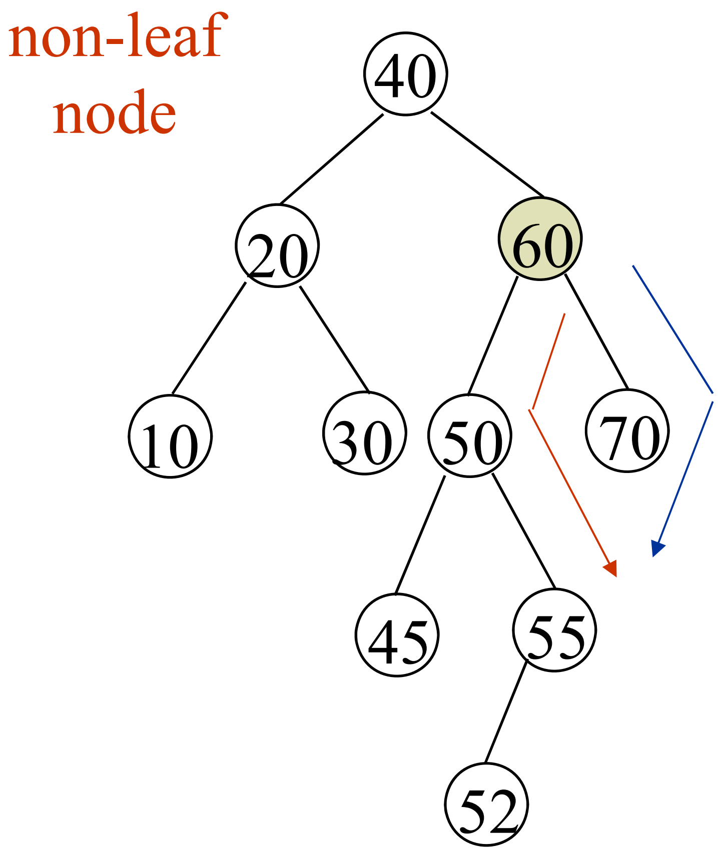
Insertion into A Binary Search Tree

```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
      temp = modified_search(*node, num);
  if (temp || !(*node)) {
   ptr = (tree_pointer) malloc(sizeof(node));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   if (*node)
     if (num<temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
   else *node = ptr;
  }
}
```

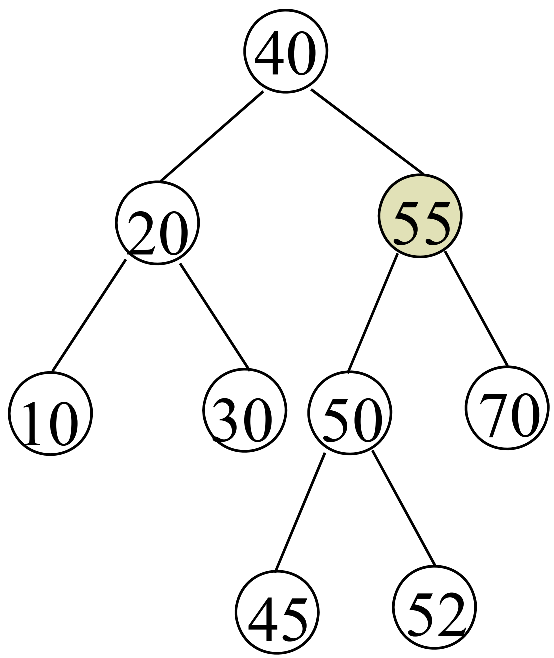
Deletion for A Binary Search Tree



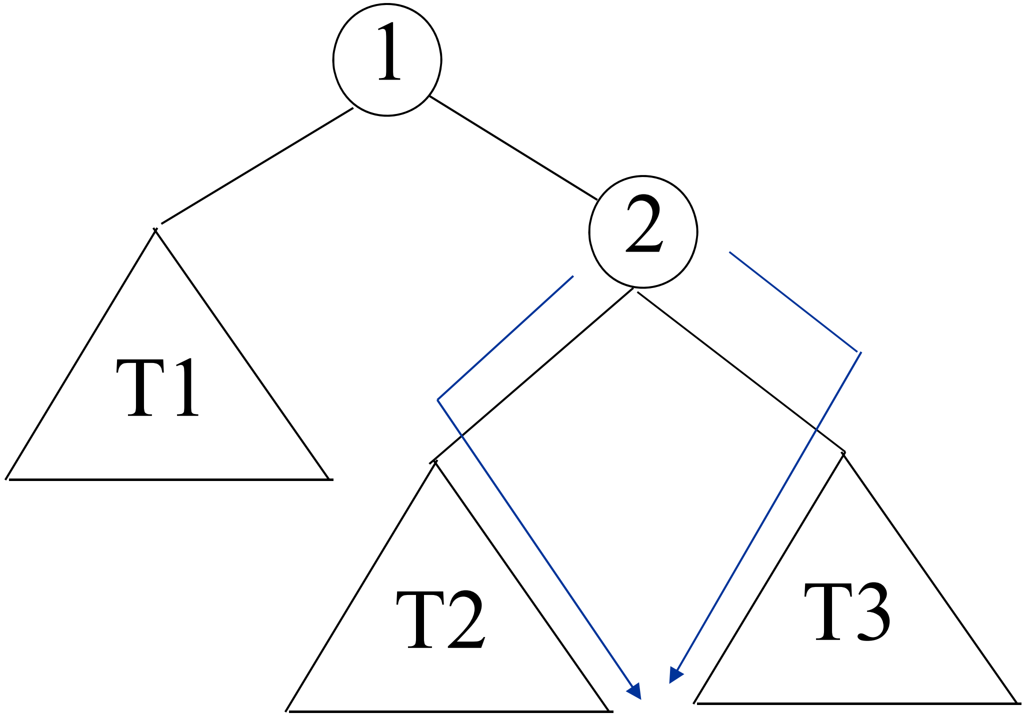
Deletion for A Binary Search Tree

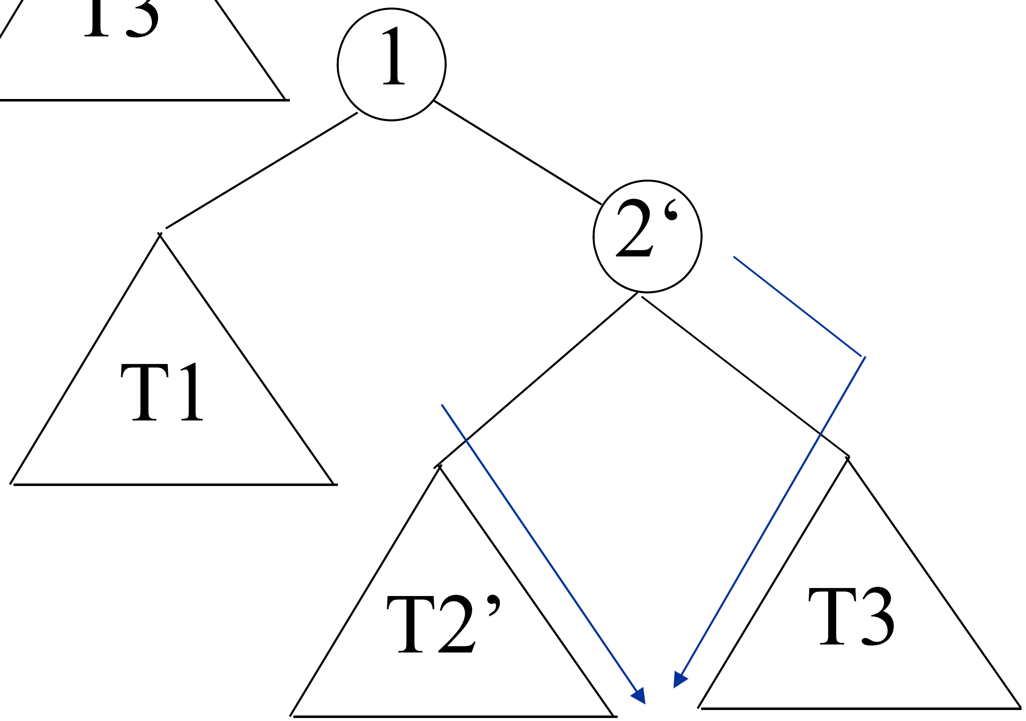


Before deleting 60



After deleting 60


1
2
T1
T2
T3


1
2‘
T1
T2’
T3