CHAPTER 5

Trees and Binary Trees

Trees

Definition:

不可為ϕ

A tree is a finite set of one or more nodes such that:

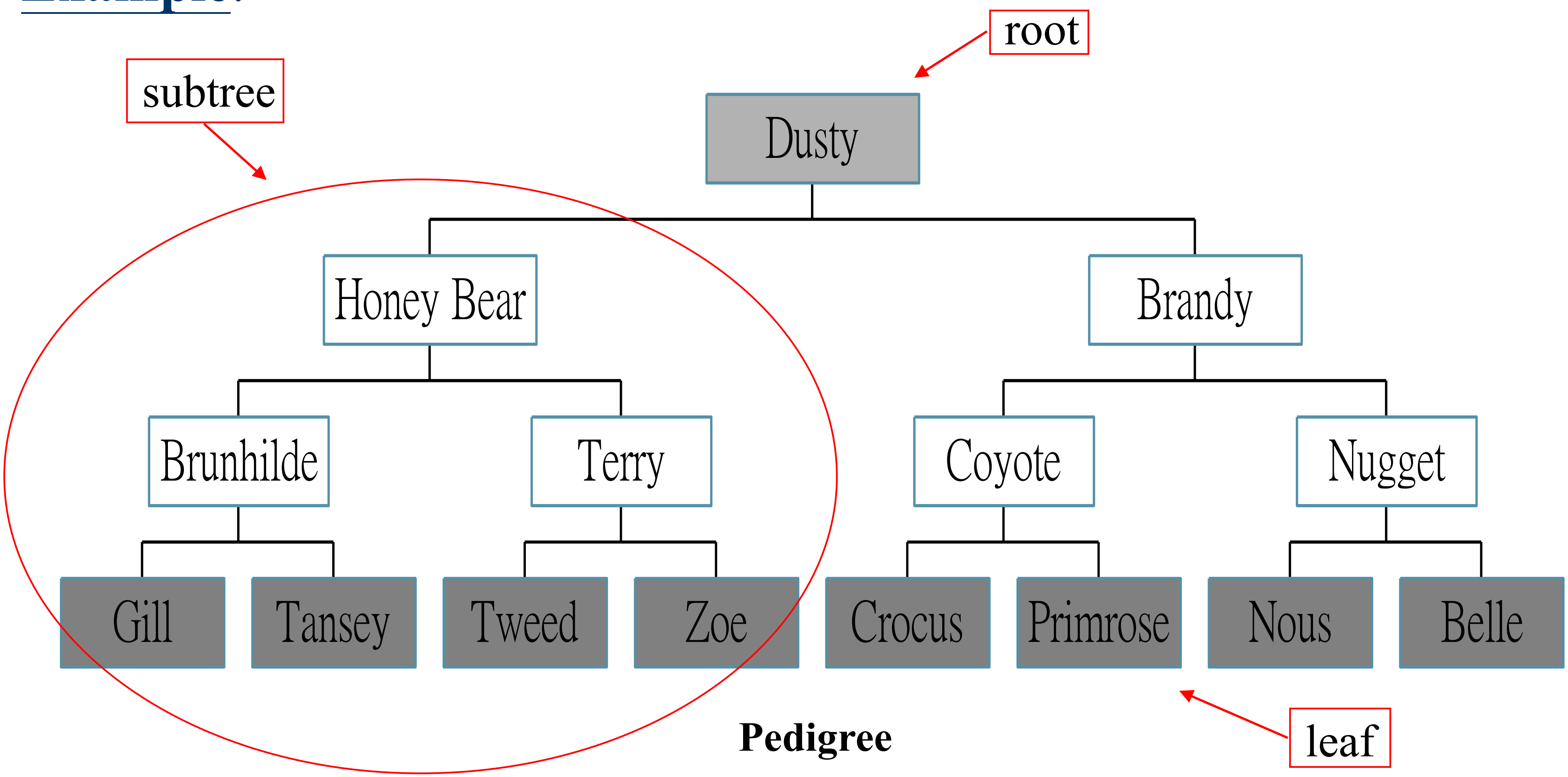
(1) There is a specially designated node called the root.

(2) The remaining nodes are partitioned into n (≥0) disjoint sets T_1, ..., T_n, where each of these sets is a tree.

(3) We call T_1, ..., T_n the subtrees of the root.

Trees

Example:



Pedigree

Trees

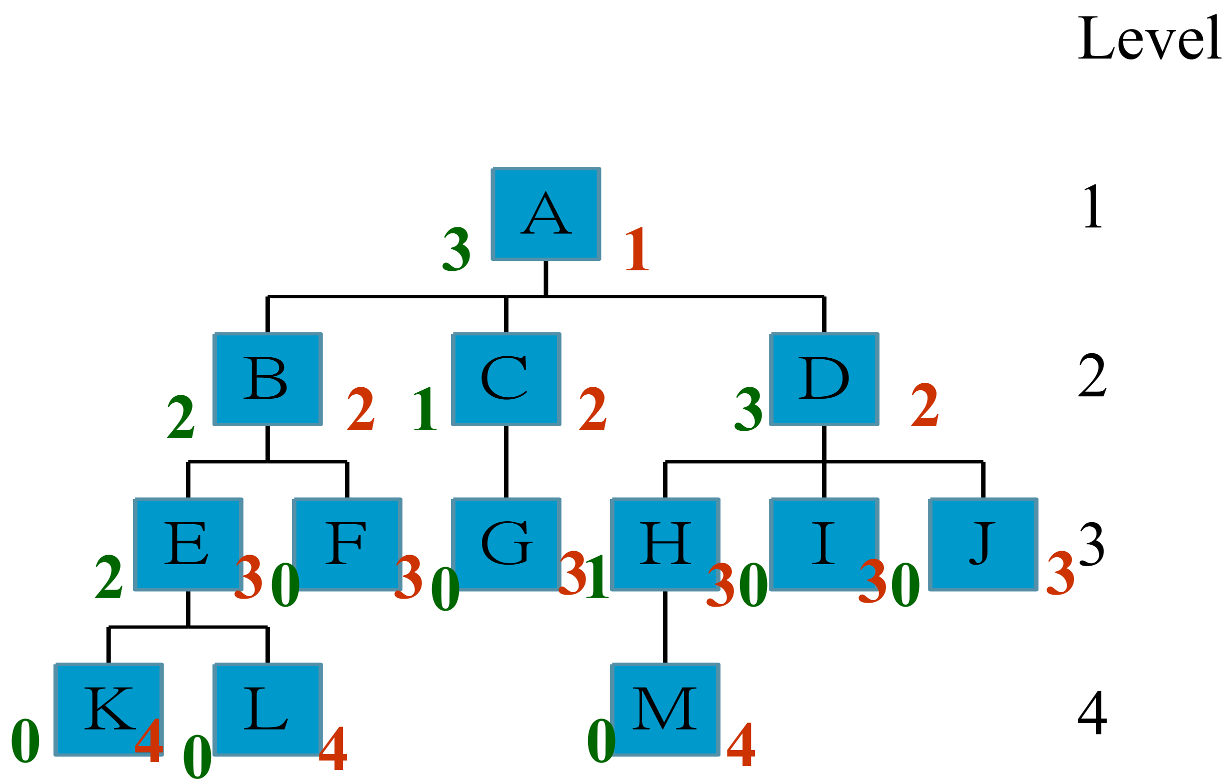
Terminology:

(1) The *degree* of a node is the number of subtrees of the node
(2) The *degree of a tree* is the maximum of the degree of the nodes in the tree.
(3) The node with degree 0 is a *leaf* or *terminal node*, and the other nodes are referred to as *nonterminals*.
(4) A node that has subtrees is the *parent* of the roots of the subtrees.
(5) The roots of these subtrees are the *children* of the node.
(6) Children of the same parent are *siblings*.
(7) The *ancestors* of a node are all the nodes along the path from the root to the node.
(8) The *level* of a node is defined by letting the root be at level. If a node is at level l, then its children are at level $l+1$.
(9) The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree.

Trees

Terminology:

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



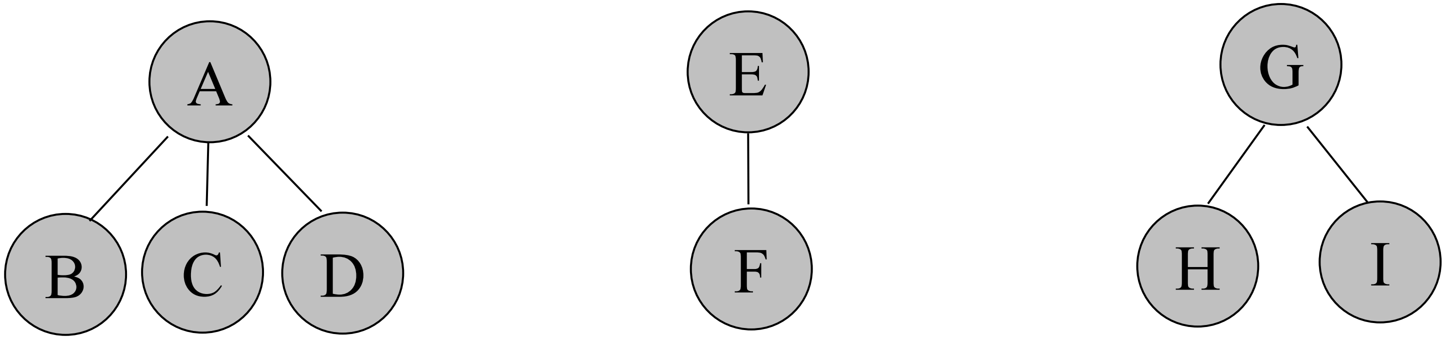
Forests

Definition:

可以為ϕ

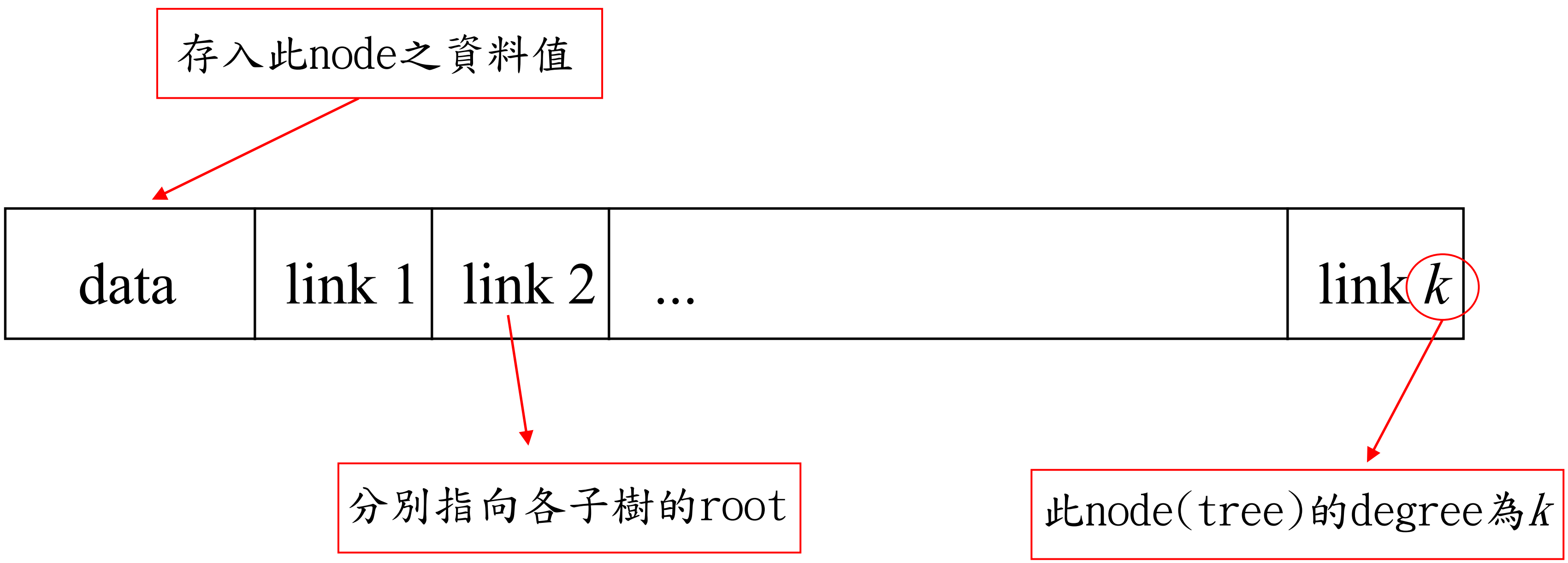
A forest is a set of $n \geq 0$ disjoint trees

Example:



Representation of Trees

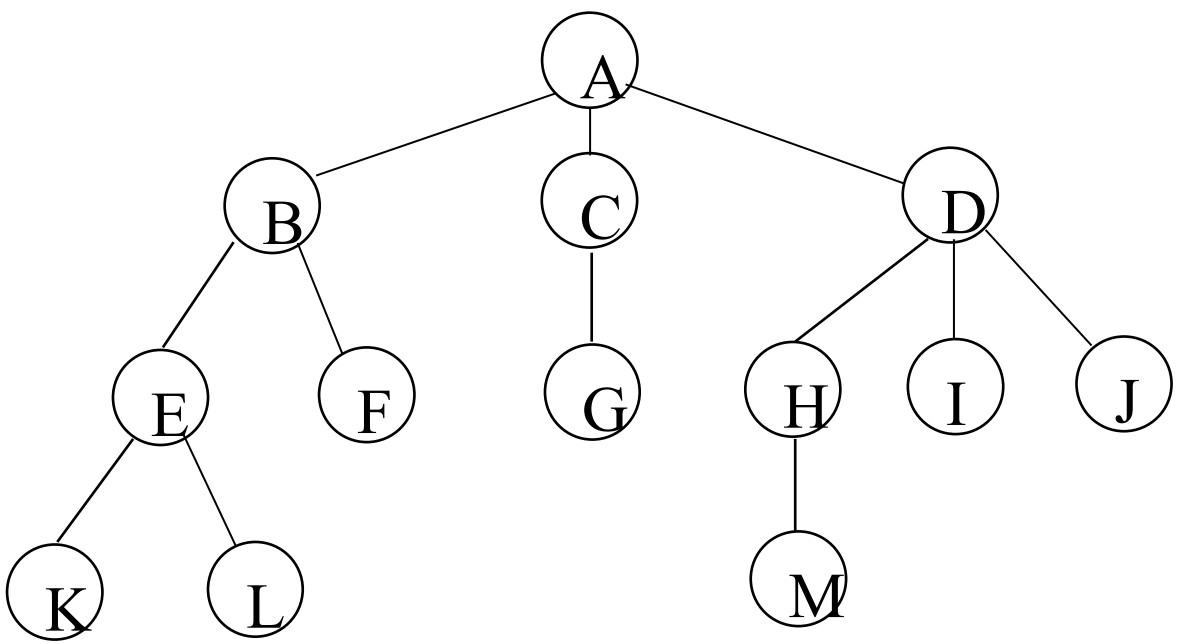
List Representation:



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



Question:

假設某一tree有n個nodes，tree的degree為k

(1) 則共有多少個link field？

(2) 真正用到的link field有幾個？

(3) 浪費的link field 有幾個？

(4) 浪費的比例為何？

Summary:

若要降低link field浪費的比例，則k=2，其浪費比例約為1/2

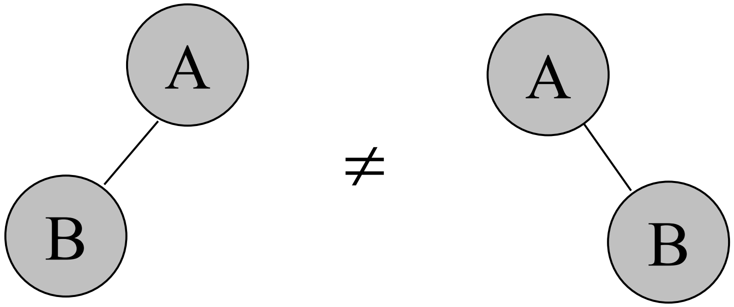
Binary Trees

Definition:

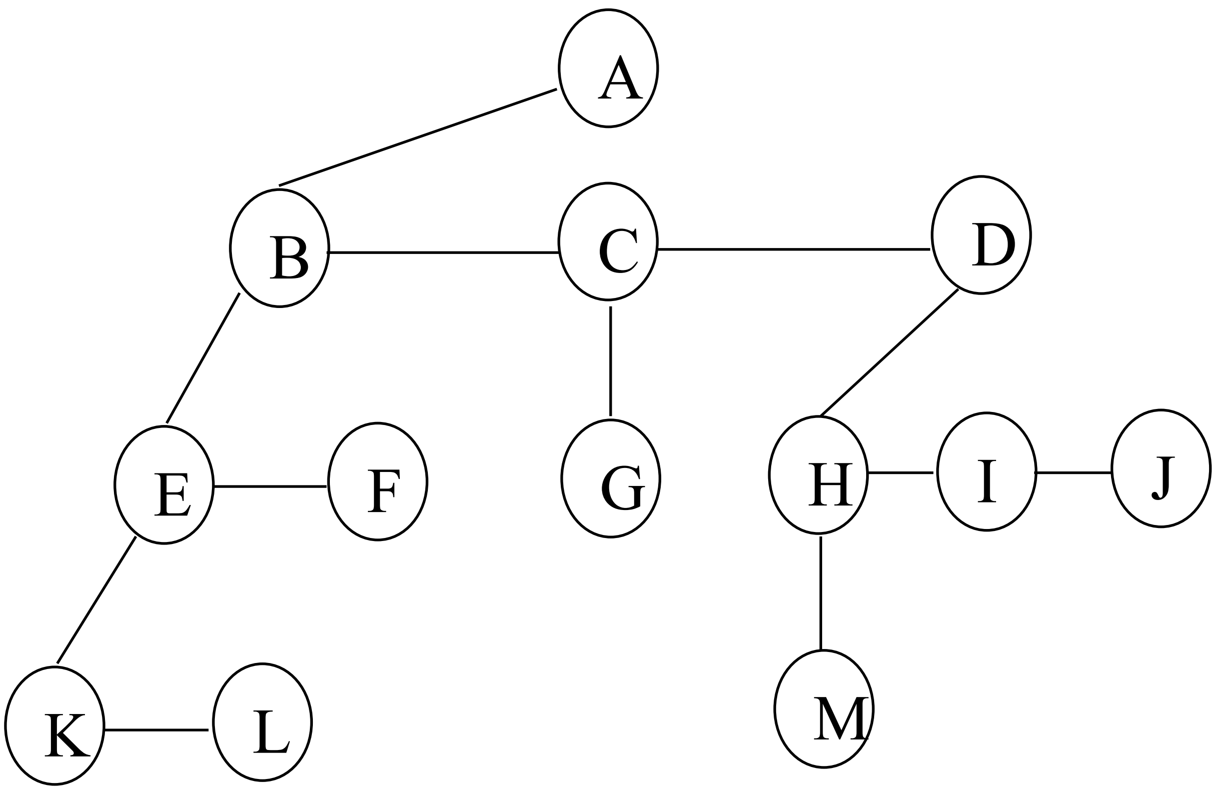
A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left subtree* and the *right subtree*.

(1) Any tree can be transformed into binary tree by *left child-right sibling* representation.

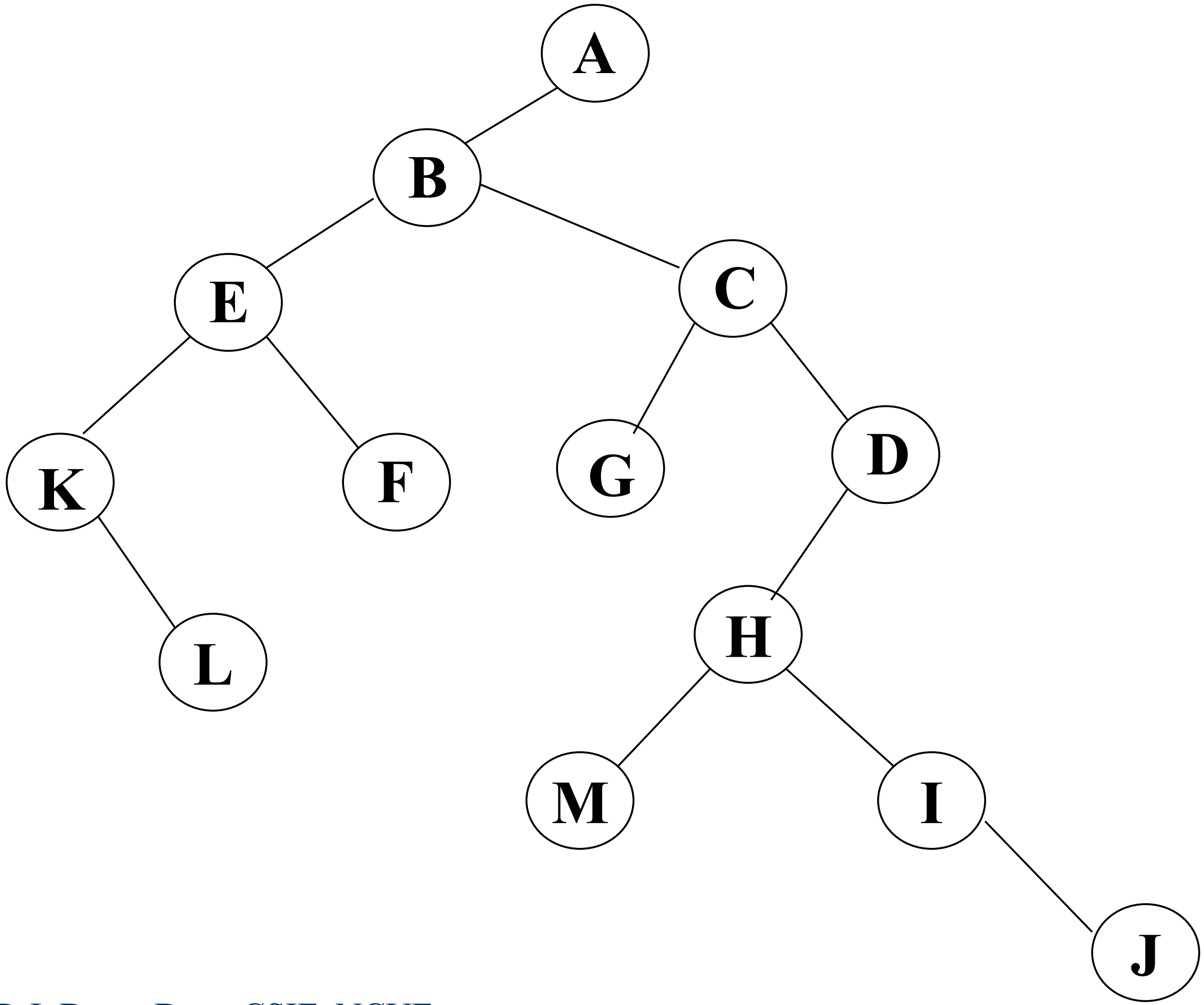
(2) The left subtree and the right subtree are distinguished.



Left Child - Right Sibling



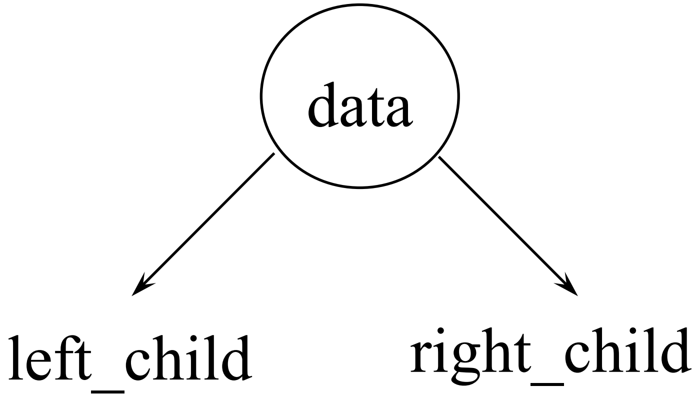
Left Child - Right Sibling



Linked Representation

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

left_child	data	right_child



Properties of Binary Trees

Lemma:

The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

Properties of Binary Trees

Lemma:

The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.

Question:

若某一二元樹具有*n*個nodes，則此二元樹之

(1) 最大高度為何？

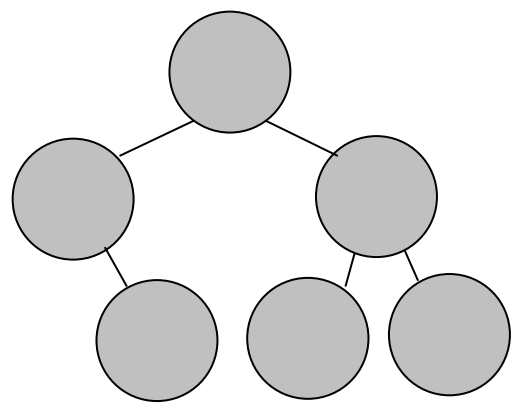
(2) 最小高度為何？

Properties of Binary Trees

Lemma [Relation between number of leaf nodes and degree-2 nodes]:

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 $n_0=n_2+1$

Example:

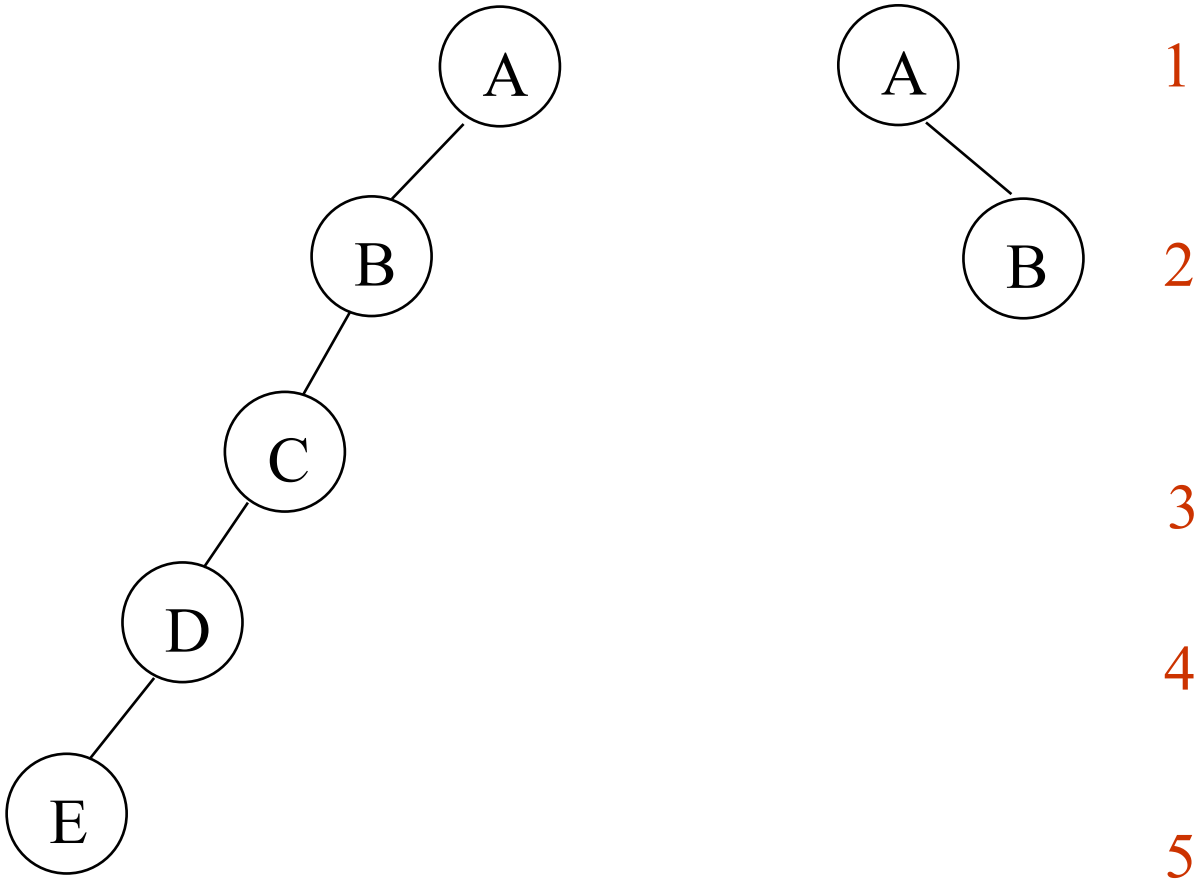


$n_0=3=2+1=n_2+1$

Question:

If a tree has a node of degree one, two nodes of degree two, three nodes of degree three....., *n* nodes of degree *n*, how many leaf nodes are there in this tree?

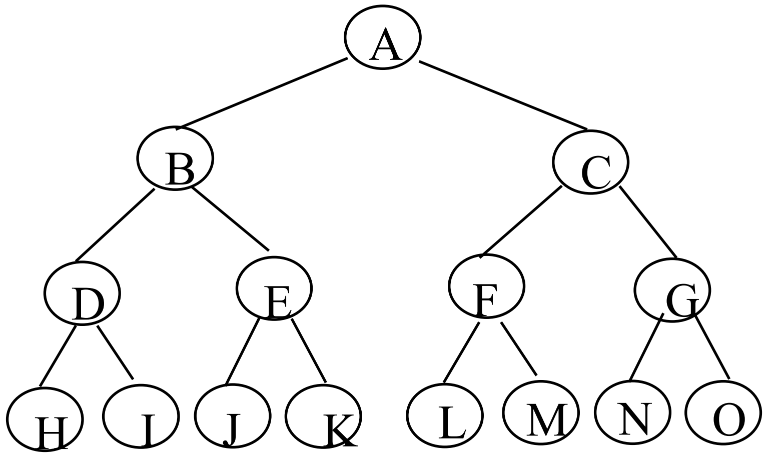
Skewed Binary Trees



Full Binary Trees

Definition:

A full binary tree of depth k is a binary tree of depth k having 2^k -1 nodes, where $k \geq 0$.

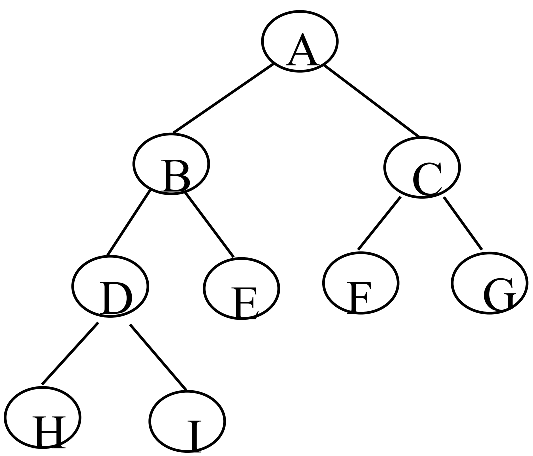


Full binary tree of depth 4

Complete Binary Trees

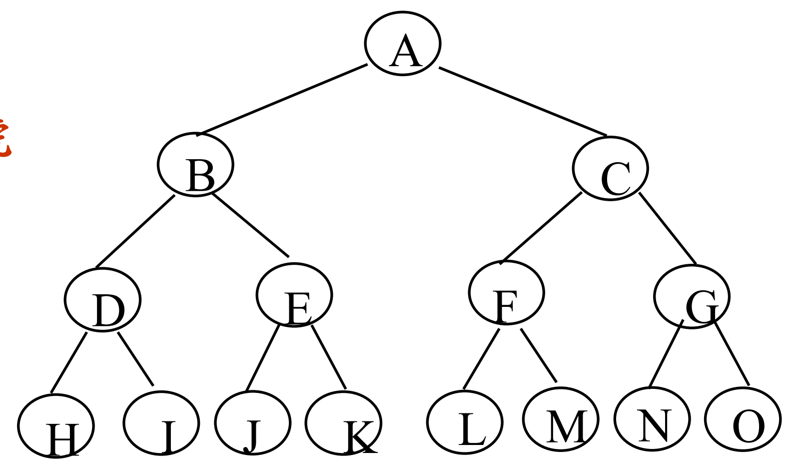
Definition:

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.



Complete binary tree

由上至下、
左至右編號



Full binary tree of depth 4

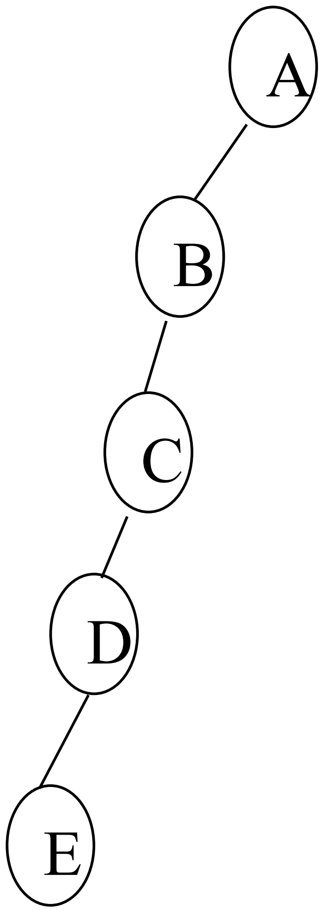
Properties of Complete Binary Trees

Lemma:

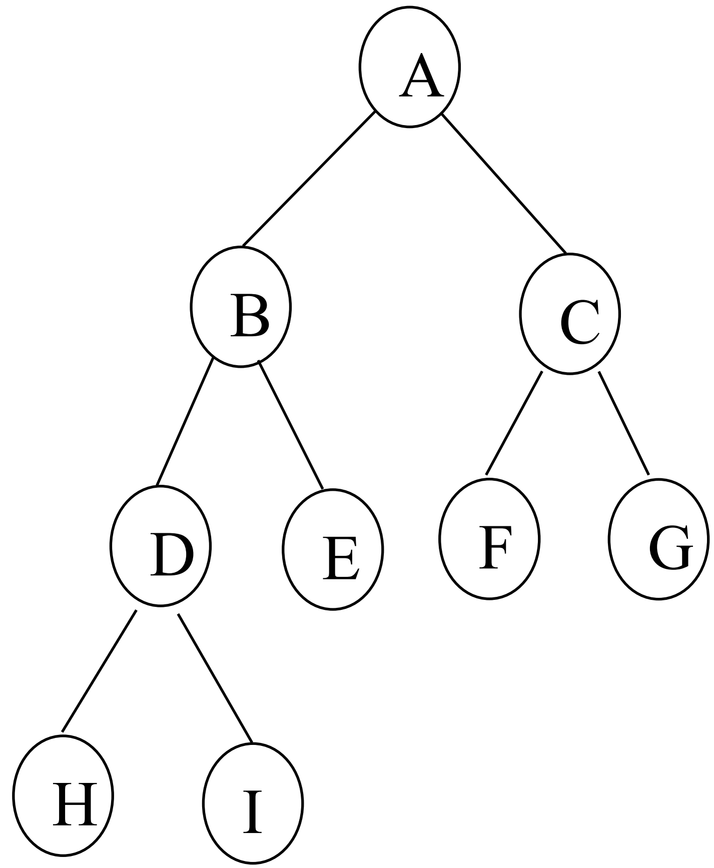
If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have:

(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i=1$, i is at the root and has no parent.
(2) left_child(i) is at $2i$ if $2i \le n$. If $2i > n$, then i has no left child.
(3) right_child(i) ia at $2i+1$ if $2i+1 \le n$. If $2i+1 > n$, then i has no right child.

Sequential Representation



[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

Binary Tree Traversals

<u>**Definition**</u>:

(1) Let L, V, and R stand for moving left, visiting the node, and moving right.

(2) There are six possible combinations of traversal

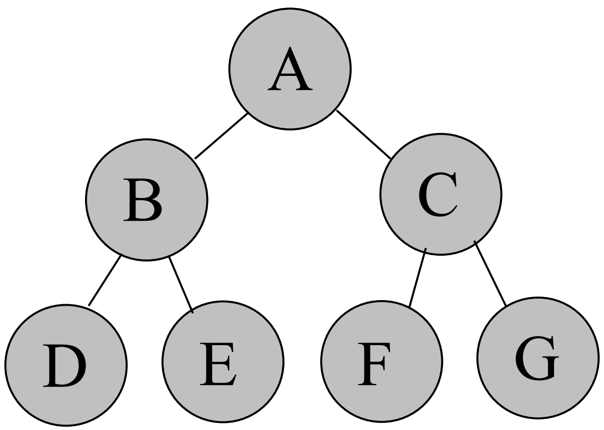
LVR, LRV, VLR, VRL, RVL, RLV

(3) Adopt convention that we traverse left before right, only 3 traversals remain:

LVR, LRV, VLR ↔ inorder, postorder, preorder

Question:

給定一Binary Tree，求其Preorder、Inorder、Postorder



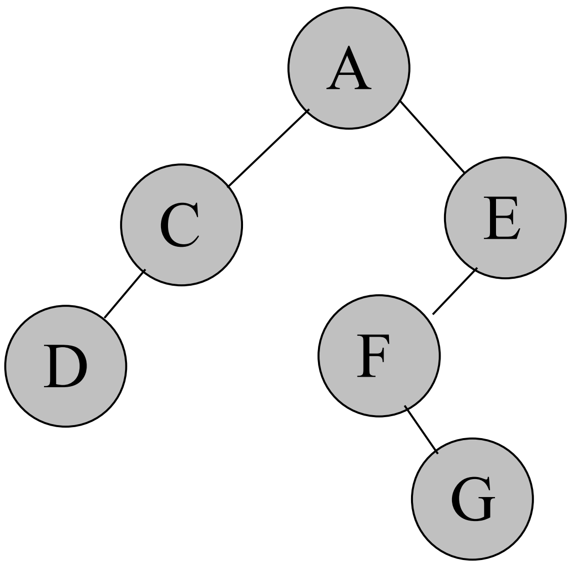
Preorder: ABDECFG

Inorder: DBEAFCG

Postorder: DEBFGCA

Question:

給定一Binary Tree，求其Preorder、Inorder、Postorder



Preorder: ACDEFG

Inorder: DCAFGE

Postorder: DCGFEA

Question:

Preorder: ABCDEFGHI，Inorder: BCAEDGHFI

(1) 求此Binary Tree

(2) 其Postorder為何？

Question:

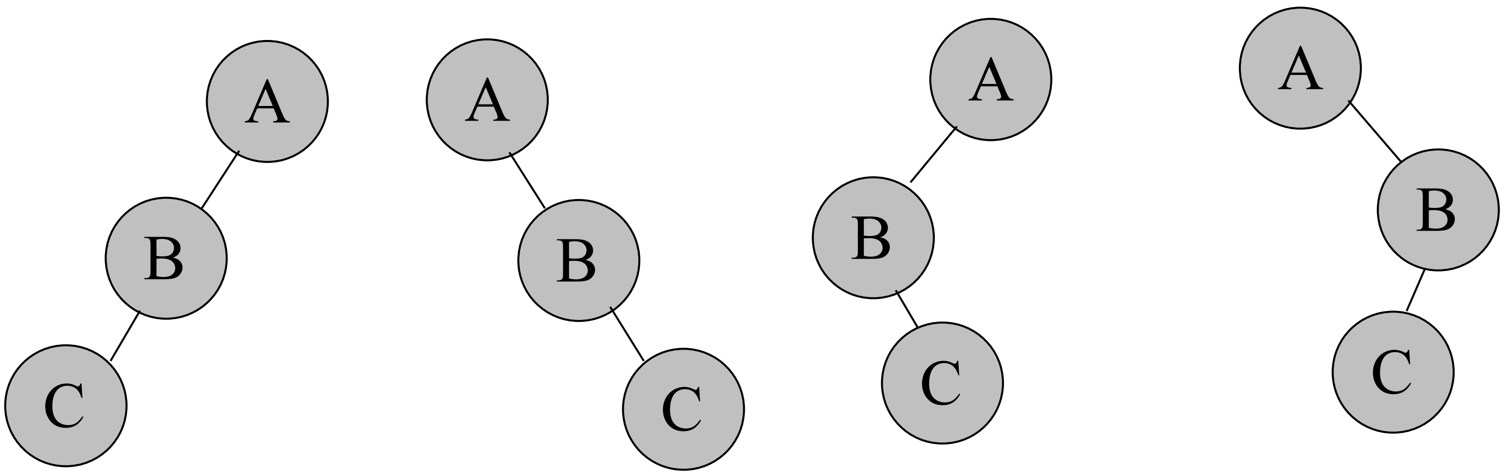
Postorder: DBEGFCA，Inorder: BDAECFG

(1) 求此Binary Tree

(2) 其Preorder為何？

Question:

試舉例說明為何給定前序與後序無法決定唯一的二元樹



Preorder: ABC　　　　Postorder: CBA

Theorem:

給定前序(後序)和中序可決定一唯一的二元樹

Inorder Traversal

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr→left_child);
        printf(“%d”, ptr → data);
        indorder(ptr → right_child);
    }
}
```

Preorder Traversal

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr → data);
        preorder(ptr → left_child);
        predorder(ptr → right_child);
    }
}
```

Postorder Traversal

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

Copying Binary Trees

```
procedure copy(original: tree_pointer): tree_pointer
{
  tree_pointer temp=nil;
  if (original≠nil)
  {
   temp → left_child=copy(original → left_child);
   temp → right_child=copy(original → right_child);
   temp → data=original → data;
  }
  return temp;
}
```

Equality of Binary Trees

```
procedure equal(first, second: tree_pointer): boolean
{
  equal=false
  if (first==nil) and (second==nil) then equal=true;
  else if (first≠nil) and (second≠nil) then
          {
          if (first →data == second →data) then
            if equal(first →left_child, second → left_child) then
            equal= equal(first → right_child, second → right_child);
          }
  return equal;
}
```

Count the no. of nodes

```
procedure count(T: tree_pointer):
{
 if (T≠nil) then
      {
       nL=count(T → left_child);
       nR=count(T → right_child);
       return (nL+nR+1);
      }
  else return 0
}
```

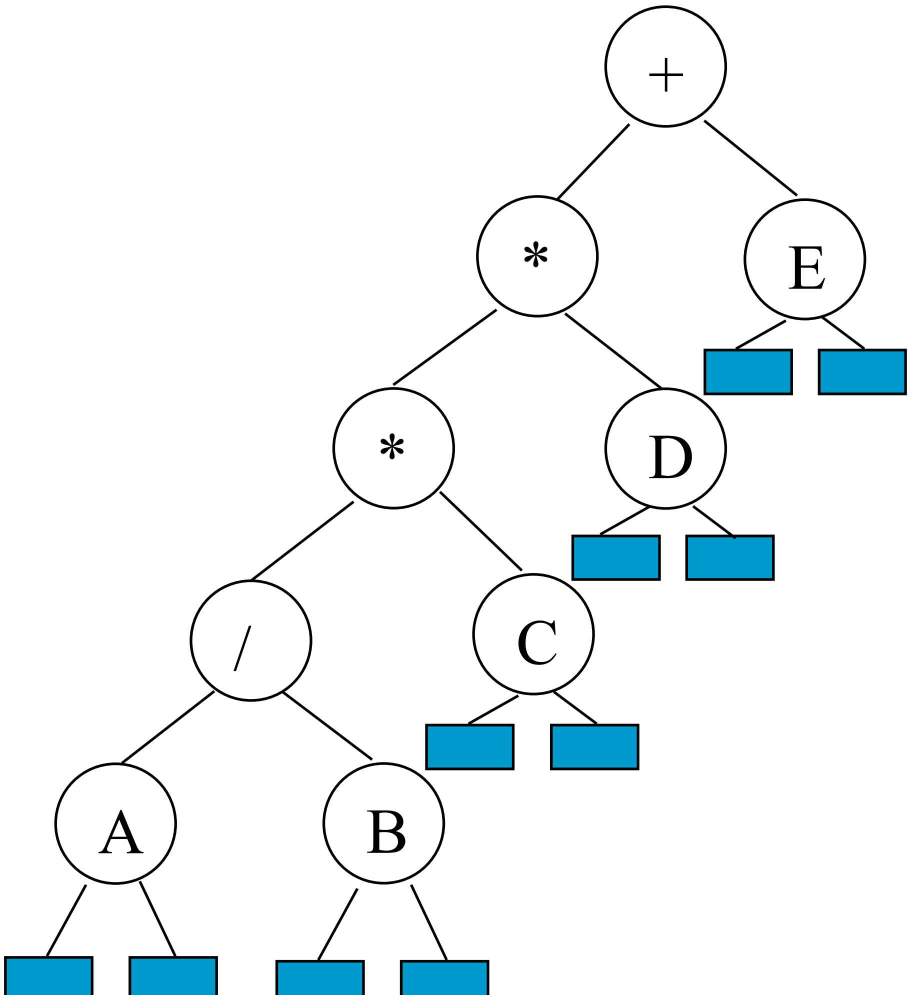
Height of Binary Trees

```
procedure height(T: tree_pointer):
{
 if (T≠nil) then
      {
       HL=height(T → left_child);
       HR=height(T → right_child);
       return  max(HL, HR)+1;
      }
  else return 0
}
```

Exercise:

Write an algorithm, SwapTree(), that takes a binary tree and swaps the left and right children of every node.

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

Propositional Calculus Expression

A variable is an expression.

If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.

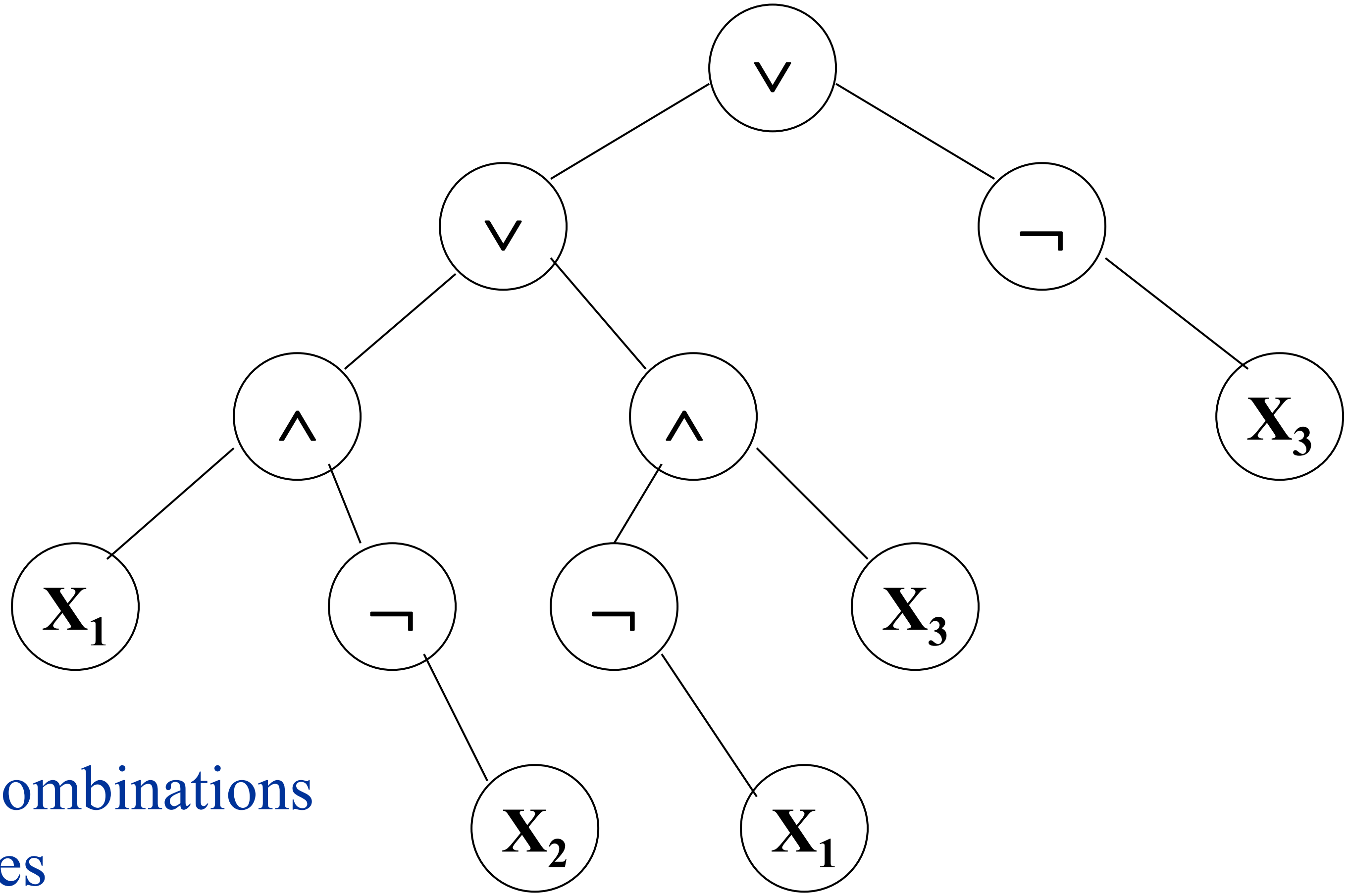
Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$).

Example: $x_1 \vee (x_2 \wedge \neg x_3)$

satisfiability problem: Is there an assignment to make an expression true?

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)



2^n possible combinations for n variables

postorder traversal (postfix evaluation)

node structure

left_child	***data***	***value***	***right_child***

```
typedef emun {not, and, or, true, false } logical;
typedef struct node *tree_pointer;
typedef struct node {
          tree_pointer  list_child;
          logical       data;
          short int     value;
          tree_pointer  right_child;
          } ;
```

Post-order-eval function

```
void post_order_eval(tree_pointer node)
{
  /* modified post order traversal to evaluate a propositional calculus
     tree */
   if (node) {
      post_order_eval(node->left_child);
      post_order_eval(node->right_child);
      switch(node->data) {
        case not:  node->value =
                 !node->right_child->value;
                 break;
```

```
  case and:     node->value =
            node->right_child->value &&
            node->left_child->value;
            break;
     case or:        node->value =
            node->right_child->value | |
            node->left_child->value;
            break;
      case true:    node->value = TRUE;
            break;
      case false:  node->value = FALSE;
     }
   }
 }
```

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

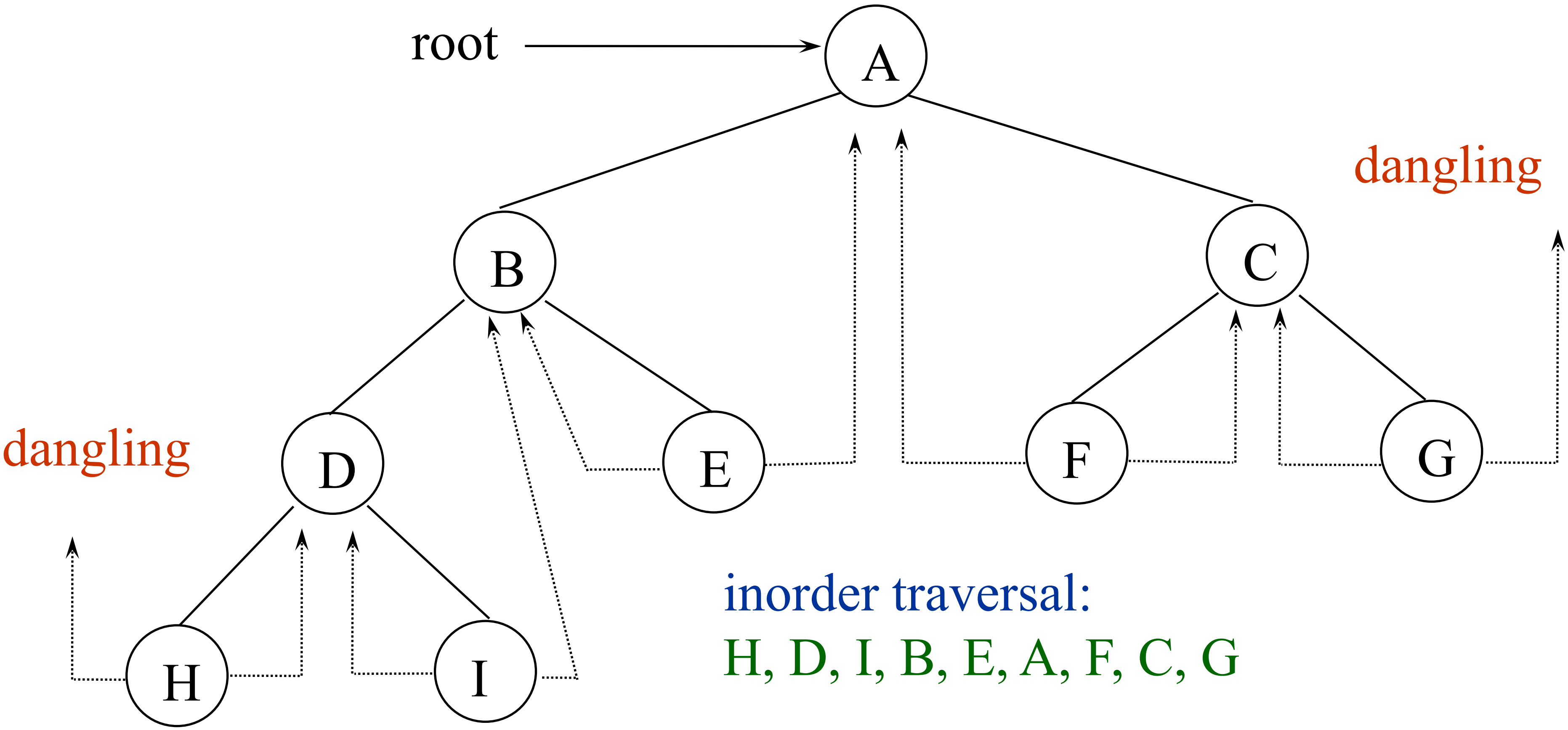
Replace these null pointers with some useful “threads”.

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*

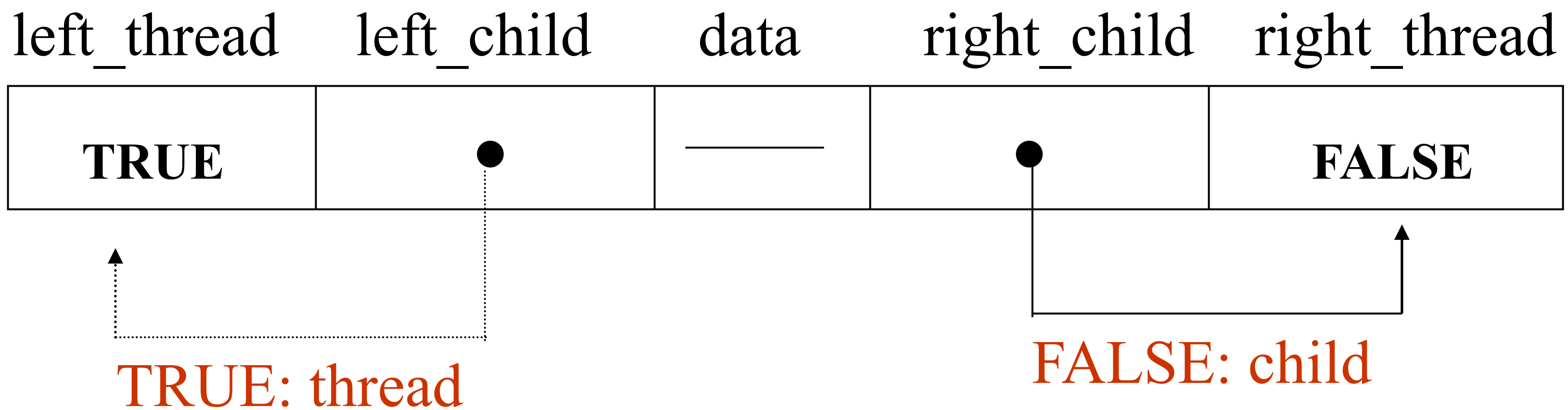
If `ptr->right_child` is null,
replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

A Threaded Binary Tree



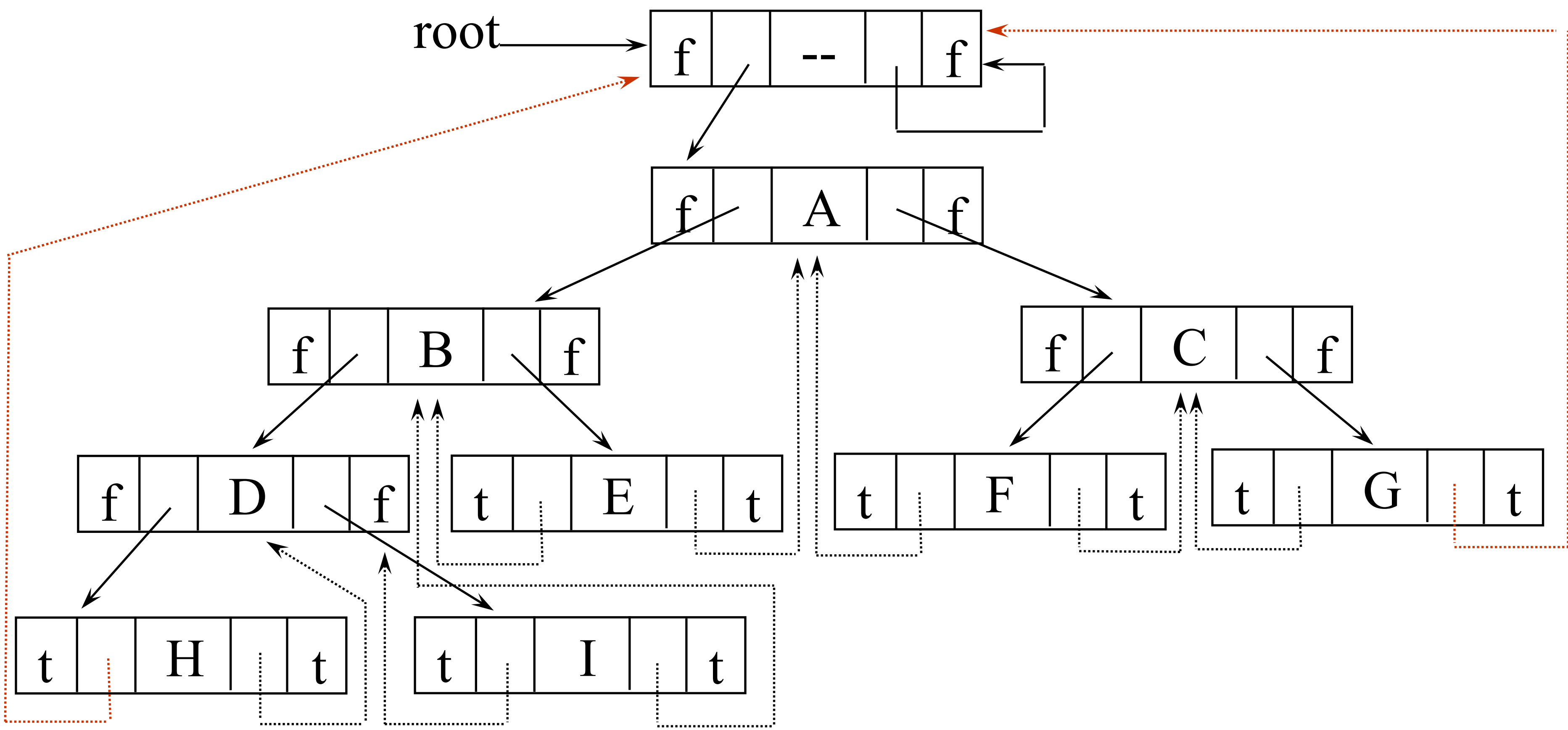
inorder traversal:
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



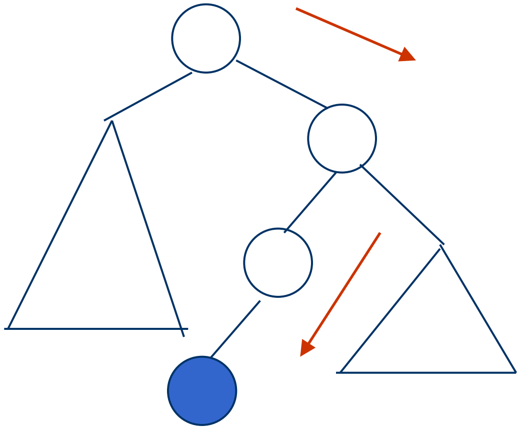

```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

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)

Binary Search Tree

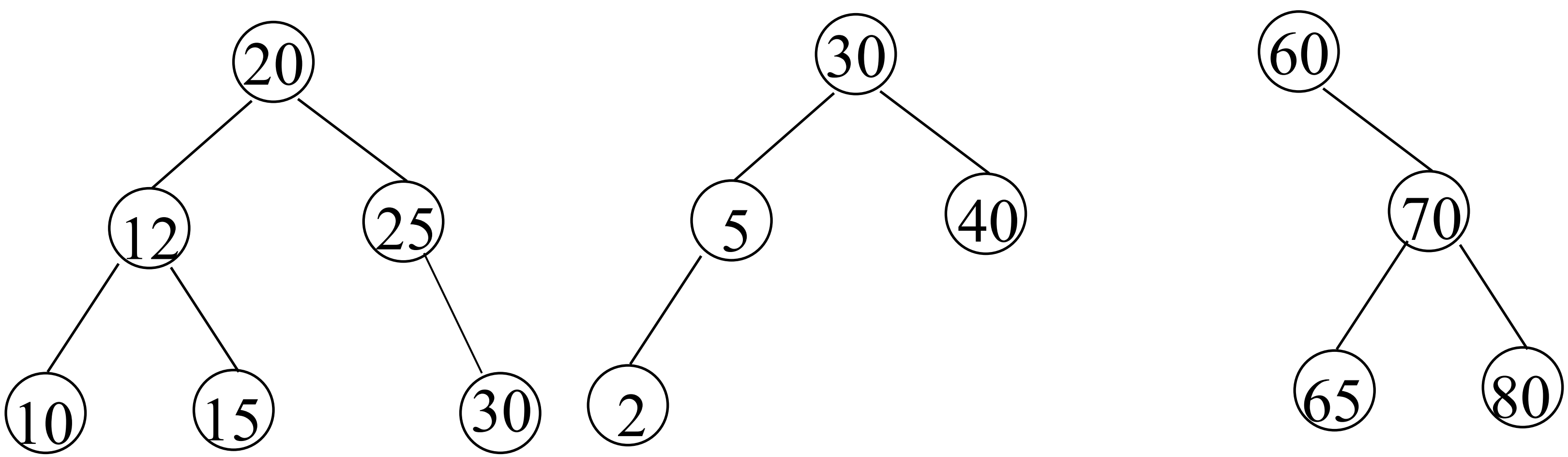
Definition:

(1) Every element has a unique key
(2) The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
(3) The left and right subtrees are also binary search trees.

Purpose:

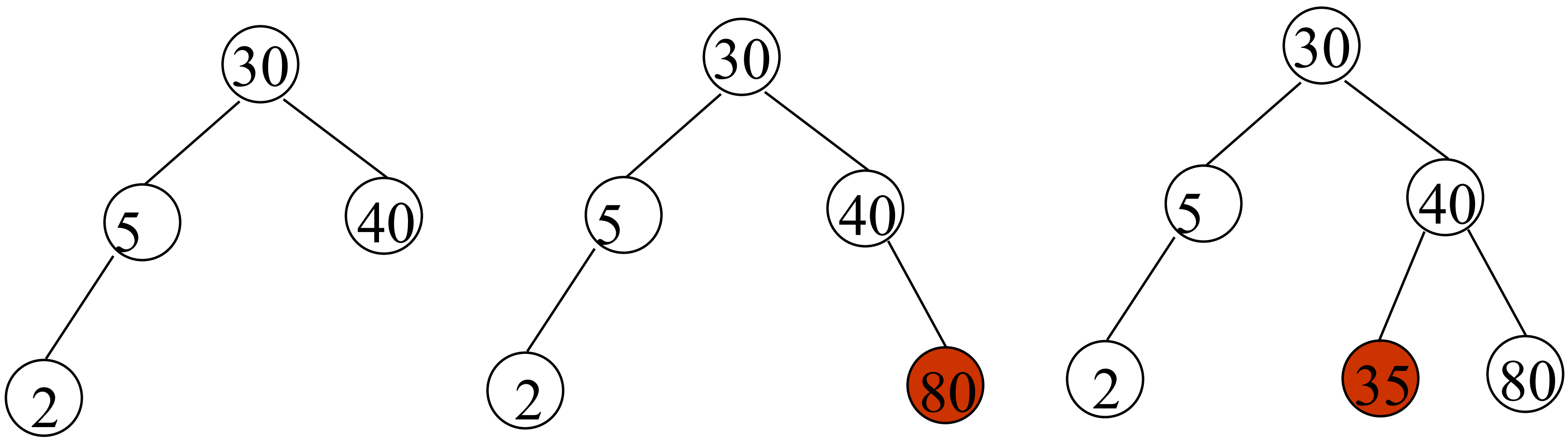
(1) Search
(2) Sorting

Examples of Binary Search Trees



Binary Search Tree

Build or Insert:



Insert 80

Insert 35

Question:

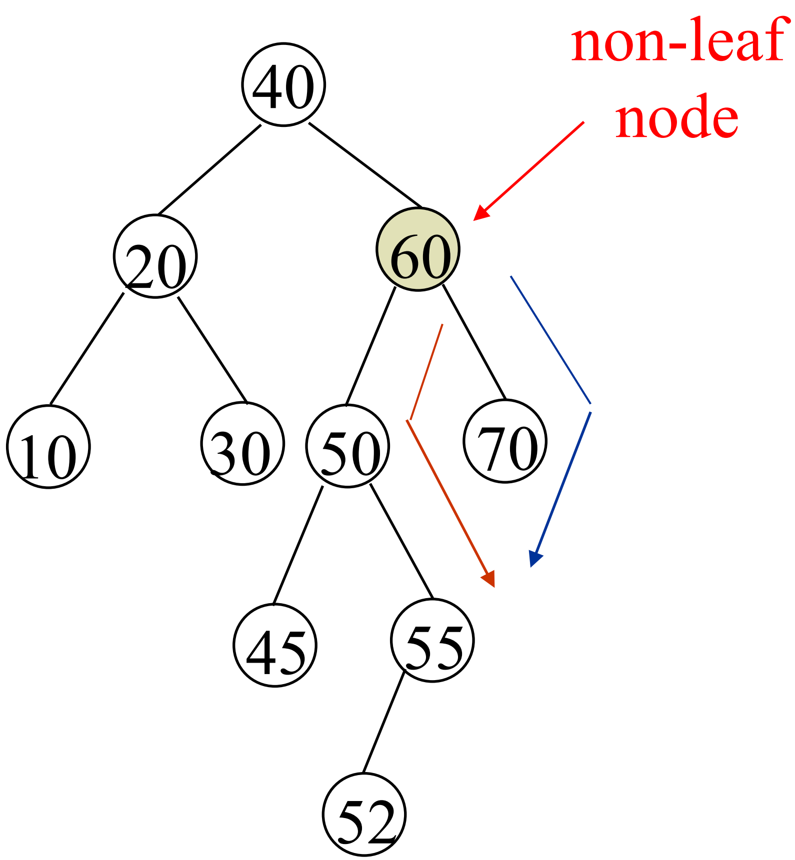
請依據下列資料輸入順序建立 Binary Search Tree
26, 5, 33, 77, 19, 2, 13, 18

Binary Search Tree

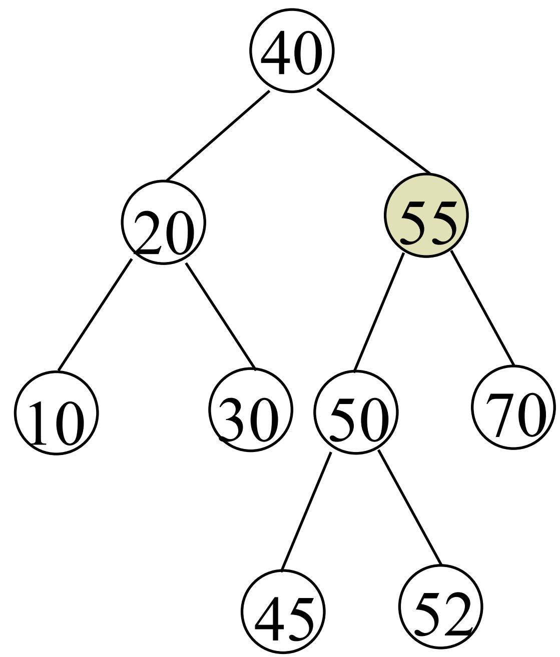
Delete:

(1) 先找到 X 之所在位置
(2) 若 X 為 leaf ，則直接刪除
(3) 若 X 有一個 child ，則向上取代 X
(4) 若 X 有 subtree，則取左子樹中最大或右子樹中最小者取代X ，goto (2)

Example:

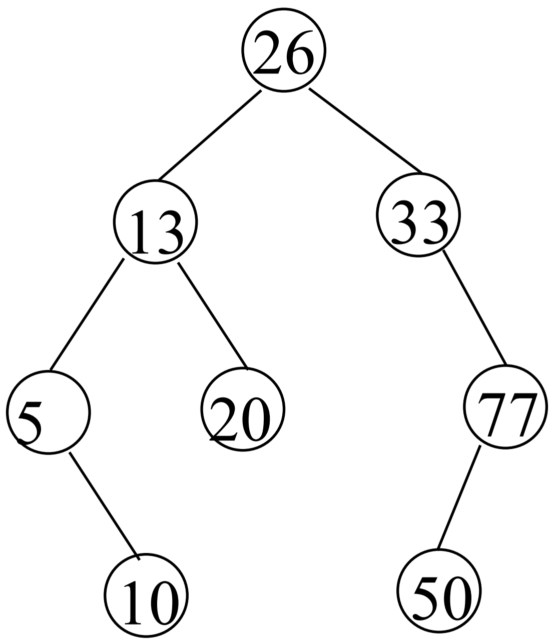


Before deleting 60



After deleting 60

Question:



(1) Delete 50
(2) Delete 5
(3) Delete 26

Searching a Binary Search Tree

```
Procedure Search (BST tree_pointer T, x)
{ if (T≠nil) then
      {
        switch (compare x, T →data)
        { case '=' : return "found"
          case '<' : return Search (T →left_child, x)
          case '>' : return Search (T →right_child, x) }
        return "not found"
      }
  return "not found"
}
```

Time Complexity:

(1) Worst case: $O(n)$

(2) Best case: $O(\log n)$

(3) Average case ?

Question:

高度為 h 的 Full Binary Search Tree 其平均比較次數為何？

$$S = 2^0 \cdot 1 + 2^1 \cdot 2 + 2^2 \cdot 3 + ... + 2^{h-1} \cdot h$$

$$= -2^0 - 2^1 - 2^2 ..... - 2^{h-1} + 2^h \cdot h$$

$$= 2^h \cdot h - 2^h + 1$$

$$\therefore \quad Ave. = \frac{h \cdot 2^h - 2^h + 1}{2^h - 1}$$

Heap

Definition:

(1) 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.

(2) 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.

Heap

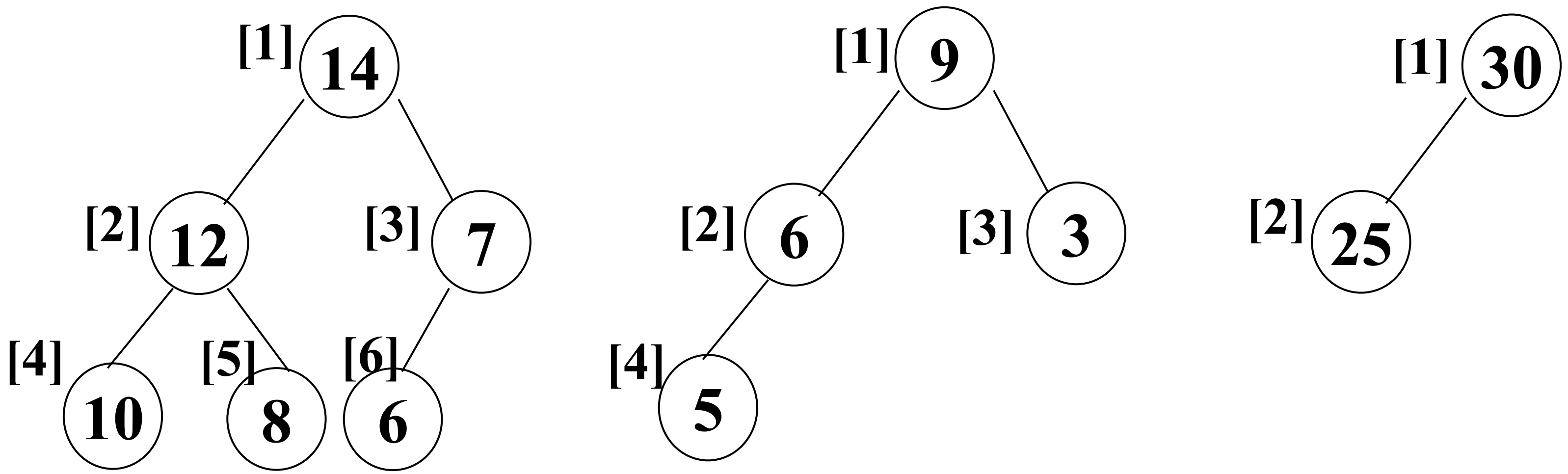
Operation:

(1) Insert
(2) Delete Max/Min
(3) Create

Application:

Priority Queue

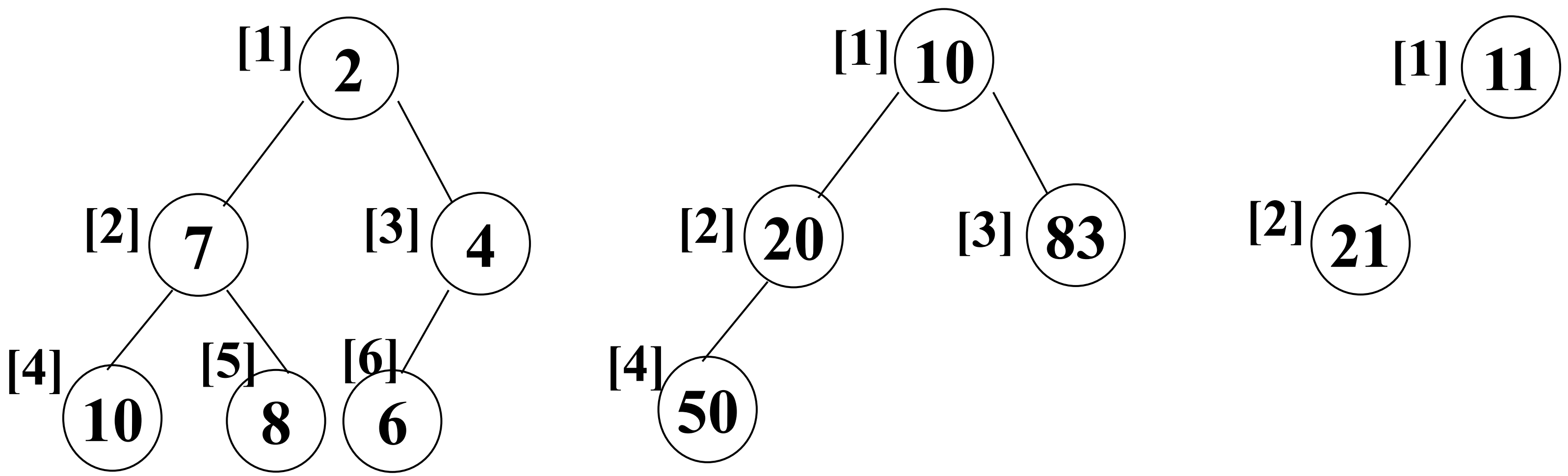
Example (Max Heap):



Property:

The root of max heap contains the largest value.

Example (Min Heap):



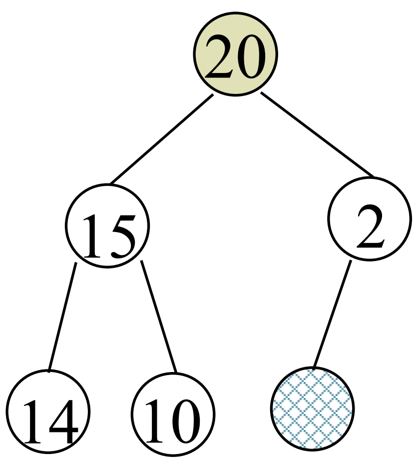
Property:

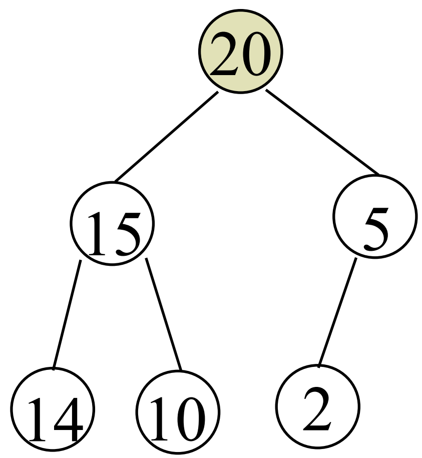
The root of min heap contains the smallest value.

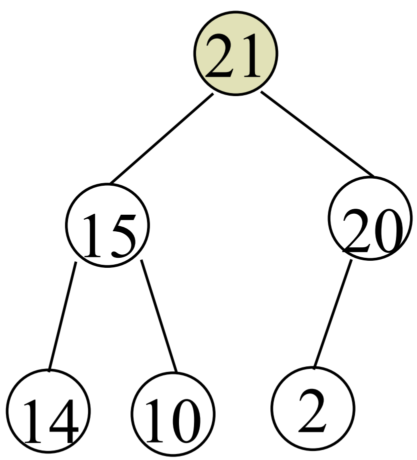
Heap

Insert (Max Heap):

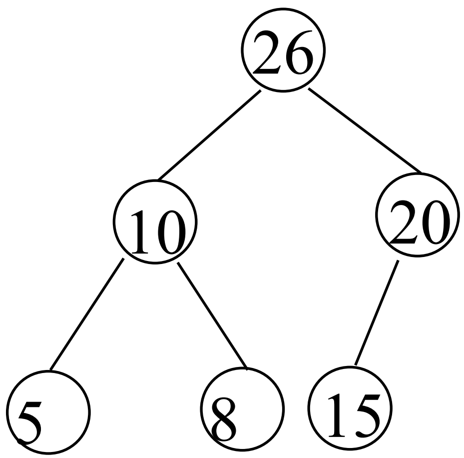
(1) 將 X 置於 last node 之後 (why?)
(2) 向上挑戰 parent node ，直到挑戰失敗或 parent node 不存在為止







Question:



連續的執行下列動作:

- Insert 80
- Insert 40
- Insert 100

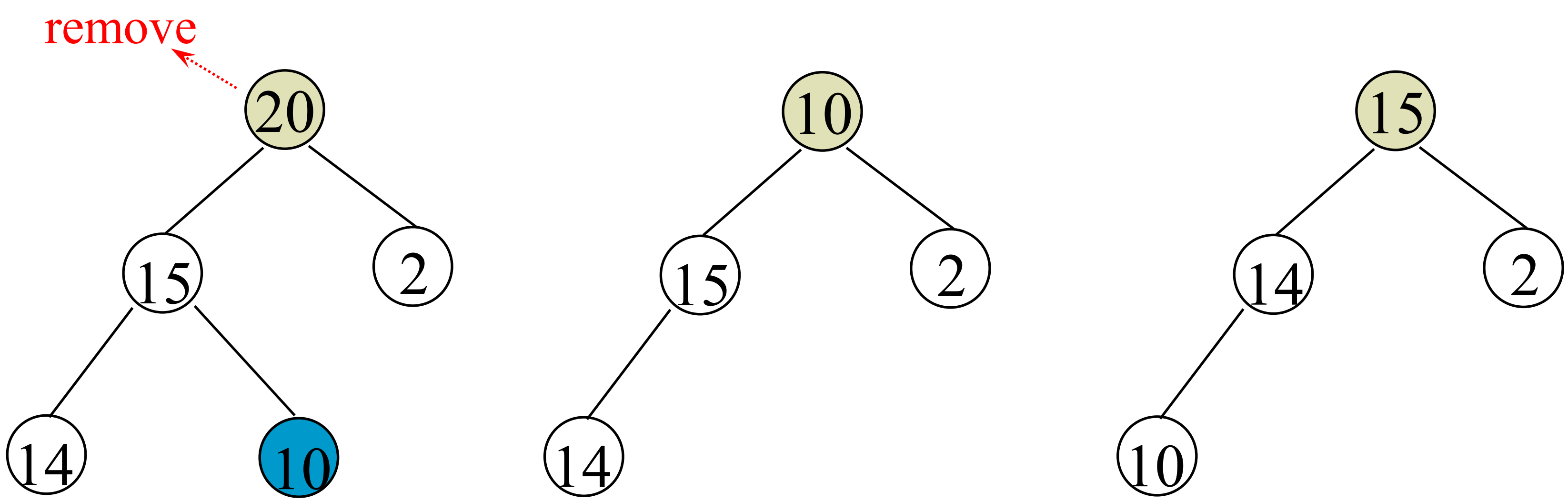
Time Complexity:

O(logn)

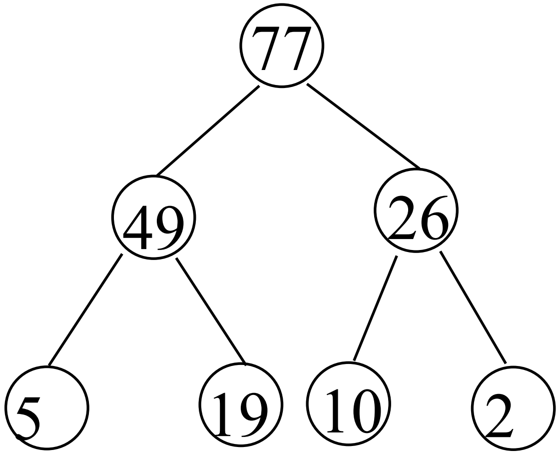
Heap

Delete (Max Heap):

(1) Delete root
(2) 將 last node 置於 root
(3) 從 root 開始往下調整



Question:



連續執行兩次 delete

Time Complexity:

O (logn)

Heap

Create (Max Heap):

(1) Top-Down: 連續執行 insert
(2) Bottom-Up: Heapify

Question:

請以 Top-Down 的方式依下列資料建立 Heap

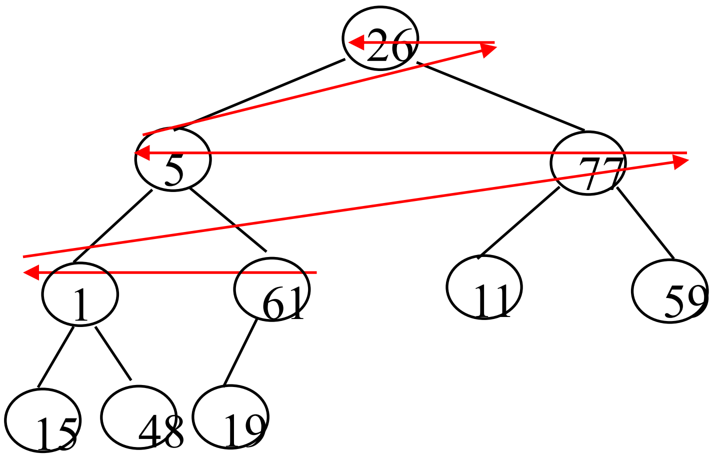
26, 5, 77, 1, 61, 11, 59, 15, 48, 19,

Heapify:

(1) 先將資料建成 Complete Binary Tree
(2) 從 last parent 開始往 root 方向調整，直到每棵子樹均為 Max-Heap

Example:

請以 Bottom-Up 的方式將下列資料建立 Heap
26, 5, 77, 1, 61, 11, 59, 15, 48, 19,

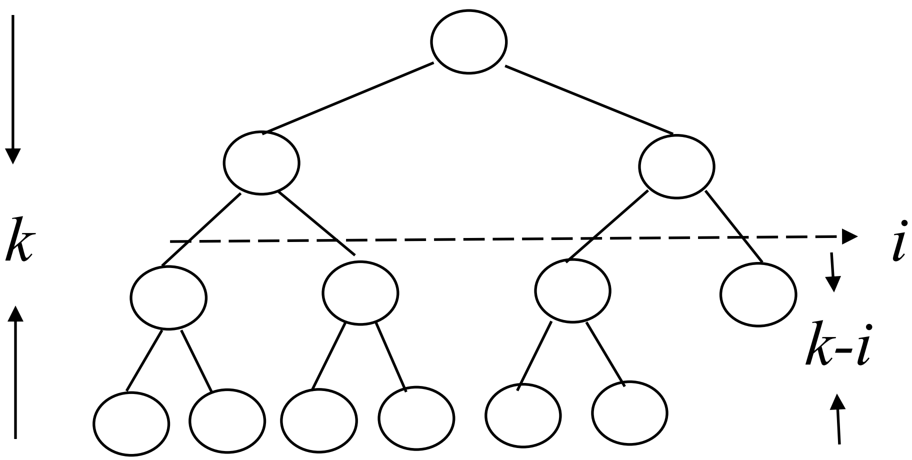


Summary:

Operation	Time Complexity
Insert	O (log n)
Delete	O (log n)
Search Max/Min	O (1)
Build	O (n)

Heapify

Time Complexity:



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

$i=1$: $2^0 \times (k-1)$

$i=k-1$: $2^{k-1-1} \times 1$

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

$i=1$

$$\leq n \cdot \sum_{i=1}^{k-1} \frac{i}{2^i} < 2 \cdot n = O(n)$$

$\boxed{2^k - 1 = n}$