Design and Analysis of Data Structures and Algorithms :: Tree part 1

อ.ดร.วรินทร์ วัฒนพรพรหม

โครงสร้างข้อมูลแบบต้นไม้ (Tree)

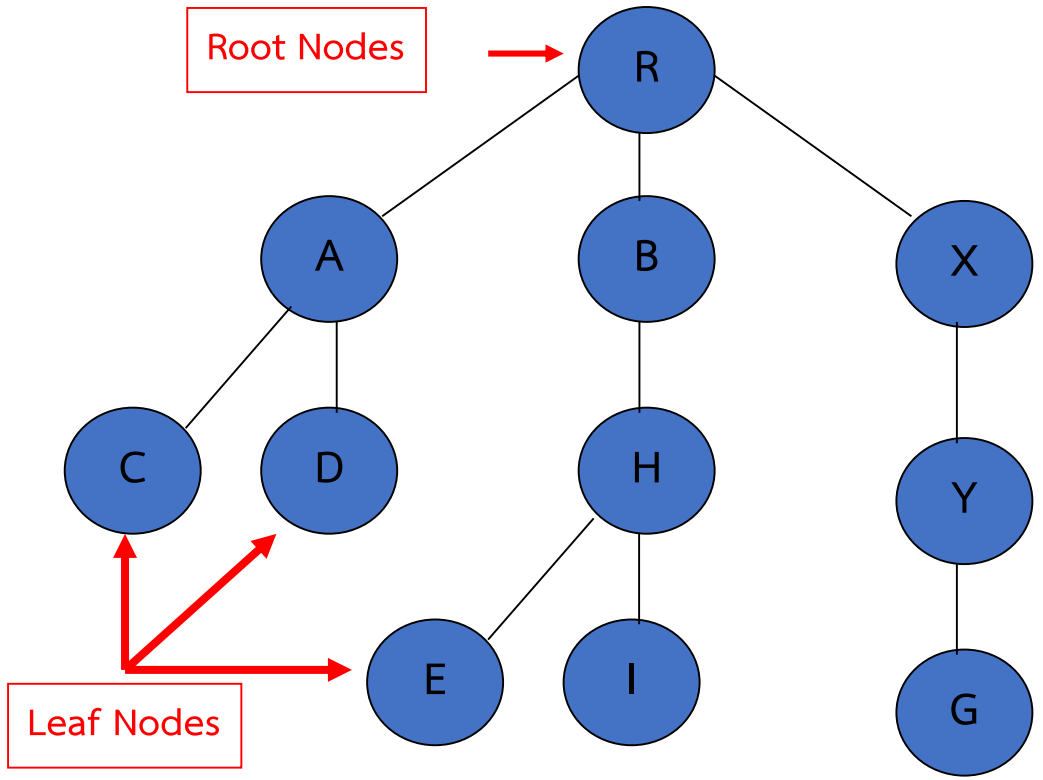
นิยามโครงสร้างต้นไม้ (Tree) เป็นโครงสร้างชนิดไม่เชิงเส้น (Non-linear) มีลักษณะเป็น recursive

- Tree เป็นกลุ่มของโหนดที่มีสมาชิกหนึ่งโหนดหรือมากกว่า โดยที่

    - มีโหนดพิเศษโหนดหนึ่งเรียกว่า Root Node

    - โหนดอื่นๆ จะถูกแบ่งออกเป็นเซตที่ไม่มี สมาชิกร่วมกันซึ่งแต่ละเซตจะมีโครงสร้างเป็น Tree ที่เรียกว่า Subtree ของ Root Node

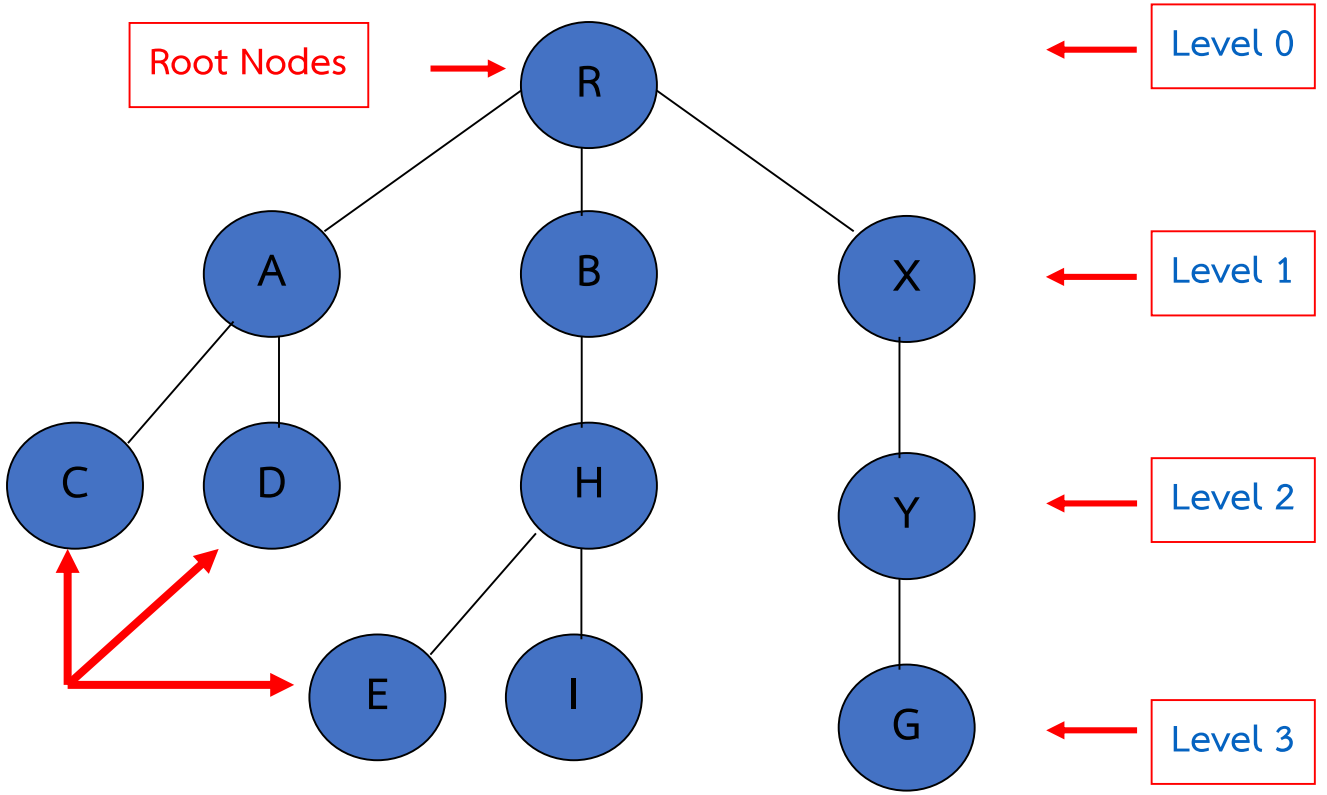
โครงสร้างต้นไม้ (Tree Structure)



Nodes X เป็น Immediate Predecessor ของ Y หรือ X เป็น Parent ของ Y

Nodes Y เป็น Immediate Successor ของ X หรือ Y เป็น Child ของ X

โครงสร้างต้นไม้ (Tree Structure)



Level แสดงถึงหน่วยระยะทางตามแนวดิ่งของโหนดว่าอยู่ห่างจาก Root Node เป็นระยะเท่าไรและทุกกิ่งมีความยาวเท่ากันคือ 1 หน่วย

โครงสร้างต้นไม้ (Tree Structure)

Root

- **root node โหนดแรกสุดไม่มีข้อมูลก่อนหน้า**

Leaf

- **โหนดที่อยู่ปลายสุด ไม่มีข้อมูลตามหลัง**

Level

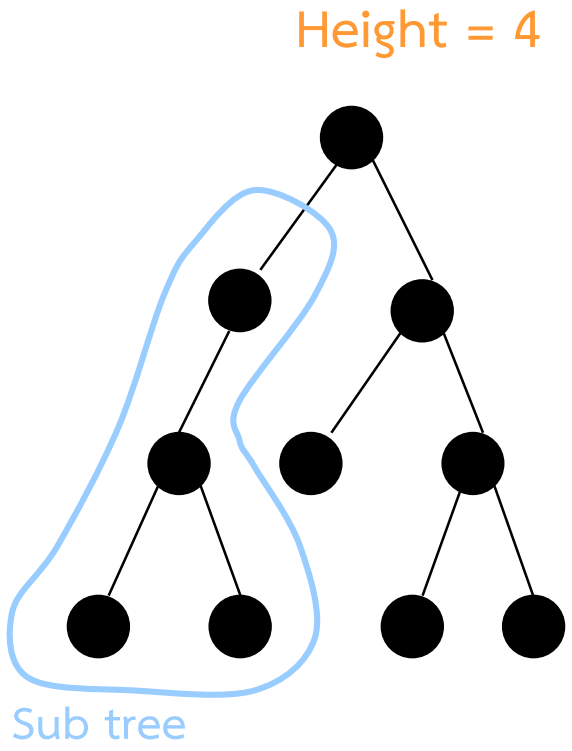
- **ระดับของโหนด ระยะห่างแนวดิ่งจาก root**

Tree Height

- **จำนวนระดับของต้นไม้**

Sub tree

- **ต้นไม่ย่อย ส่วนย่อยของต้นไม้ทั้งหมด**



โครงสร้างต้นไม้ (Tree Structure)

Parent

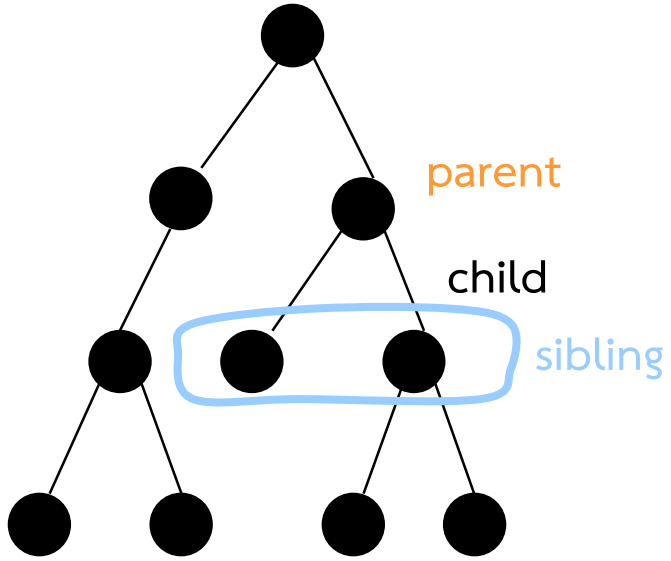
- **โหนดที่มีระดับอยู่ก่อนหน้าหนึ่งระดับ**

Child

- **โหนดที่ตามหลังในระดับถัดไป**

Sibling

- **โหนดที่มี parent เดียวกัน**



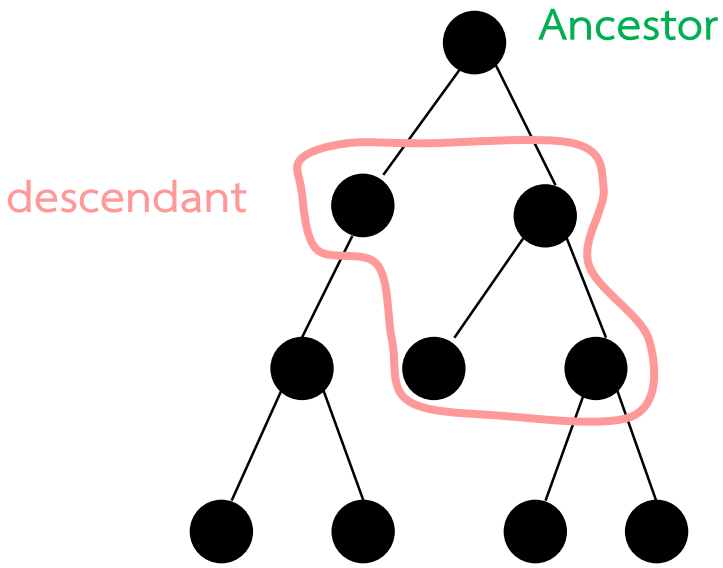
โครงสร้างต้นไม้ (Tree Structure)

Predecessor / Ancestor

- **บรรพบุรุษ โหนดที่อยู่ก่อนหน้า**

Successor / Descendant

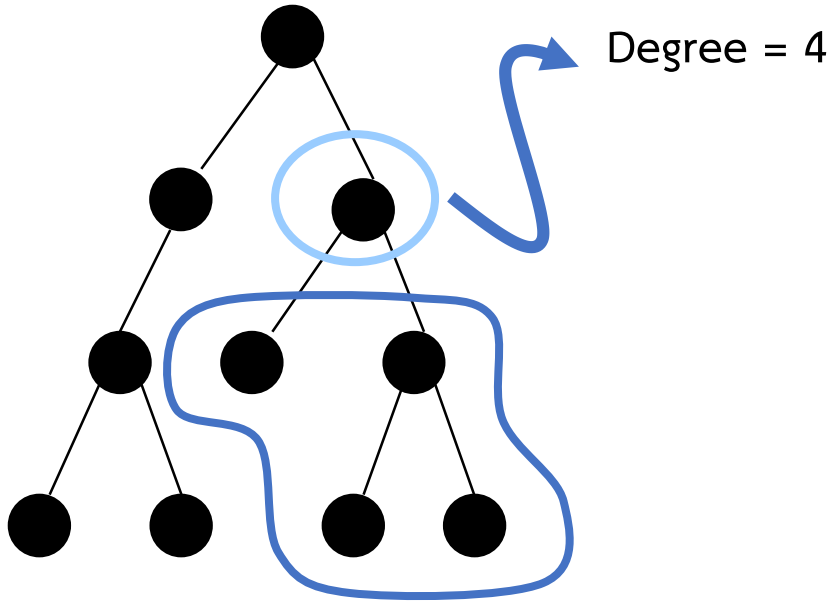
- **ลูกหลาน โหนดที่อยู่ตามหลัง**



โครงสร้างต้นไม้ (Tree Structure)

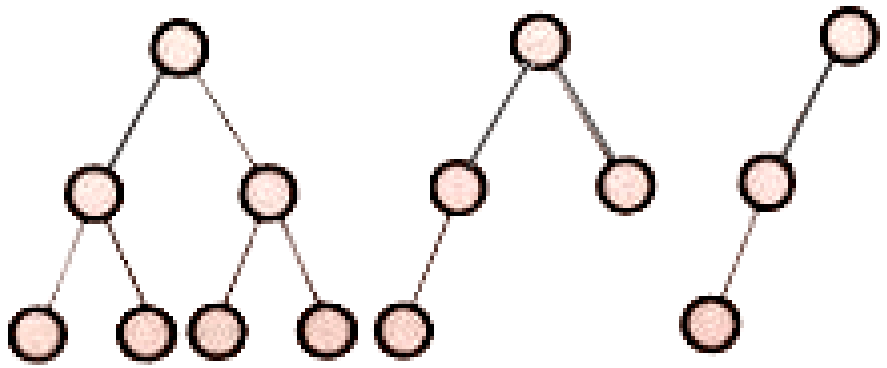
Degree

- **จำนวนลูกหลาน (descendant) ทั้งหมดของโหนดนั้น**



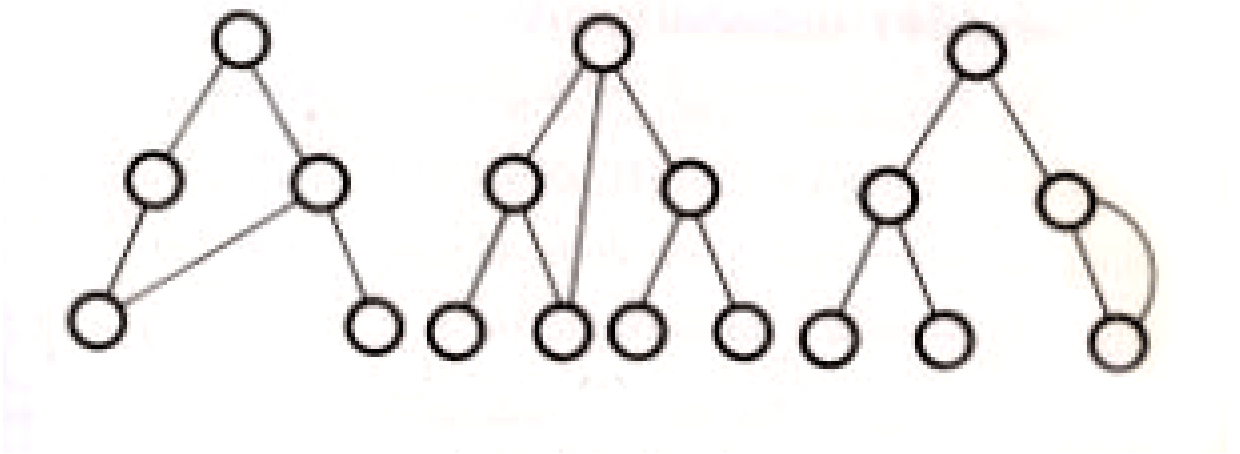
ต้นไม้แบบไบนารี (Binary Tree)

- ลักษณะพิเศษของ Binary Tree
  - ลักษณะทั่วไปเหมือนกันต้นไม้ทั่วไป
  - มีแค่ Root Node ที่ว่างเปล่า (ไม่มี Child)
  - มี Root Node และ Subtree 2 กลุ่ม ซึ่งเรียกว่า Left Subtree และ Right Subtree
  - มี Child ของ Parent ไม่เกินกว่า 2 โหนด



ต้นไม้แบบไบนารี (Binary Tree)

- ลักษณะที่ไม่ใช่ต้นไม้แบบไบนารี (Binary Tree)
  - ต้นไม้ที่มีวงจรปิด หรือ Loop



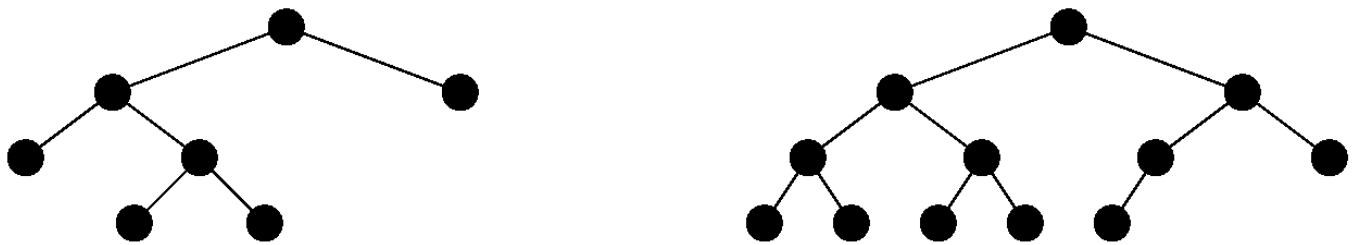
ต้นไม้ไบนารีแบบสมบูรณ์ (Complete Binary Tree)

- ต้นไม้ไบนารีสมบูรณ์ คือ ต้นไม้ไบนารีที่โหนดในระดับใดๆ จะมีโหนดเต็มจำนวนเท่ากับ 2^n ในทุกระดับ คือ 1,2,4,8,16,...
  - ความสัมพันธ์ระหว่าง Level กับจำนวนโหนดของไบนารีแบบสมบูรณ์ เขียนได้

$$n = 2^l - 1$$

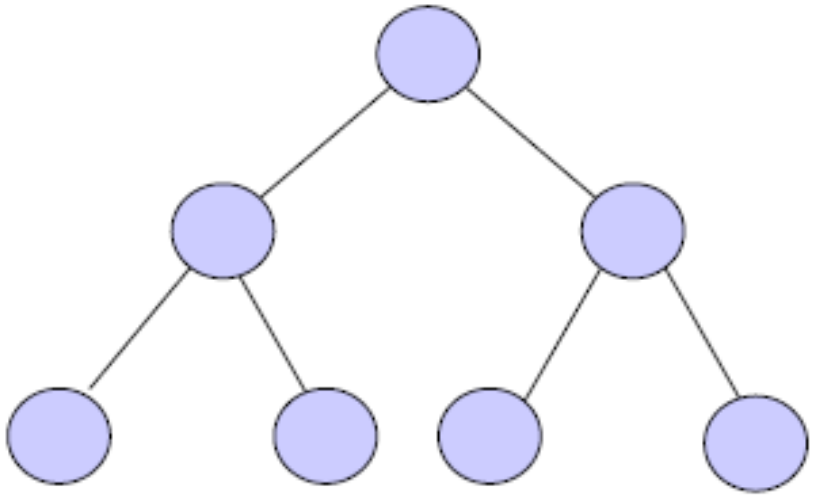
n = จำนวนโหนดทั้งหมดในต้นไม้ไบนารีแบบสมบูรณ์

l = Level ของต้นไม้ไบนารีแบบสมบูรณ์



(a) (b)

ต้นไม้ไบนารีแบบสมบูรณ์ (Complete Binary Tree)

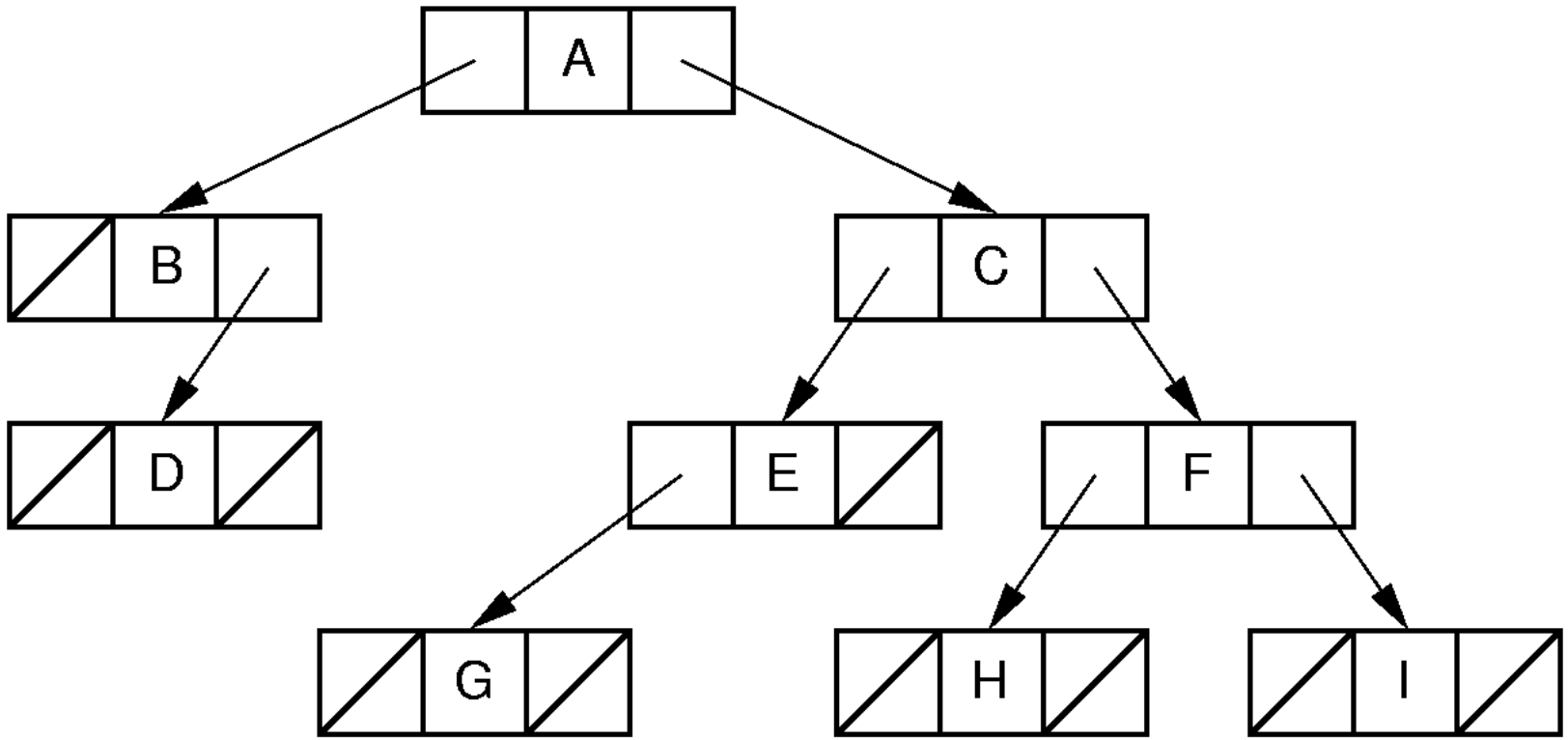


$$n = 2^3 - 1$$

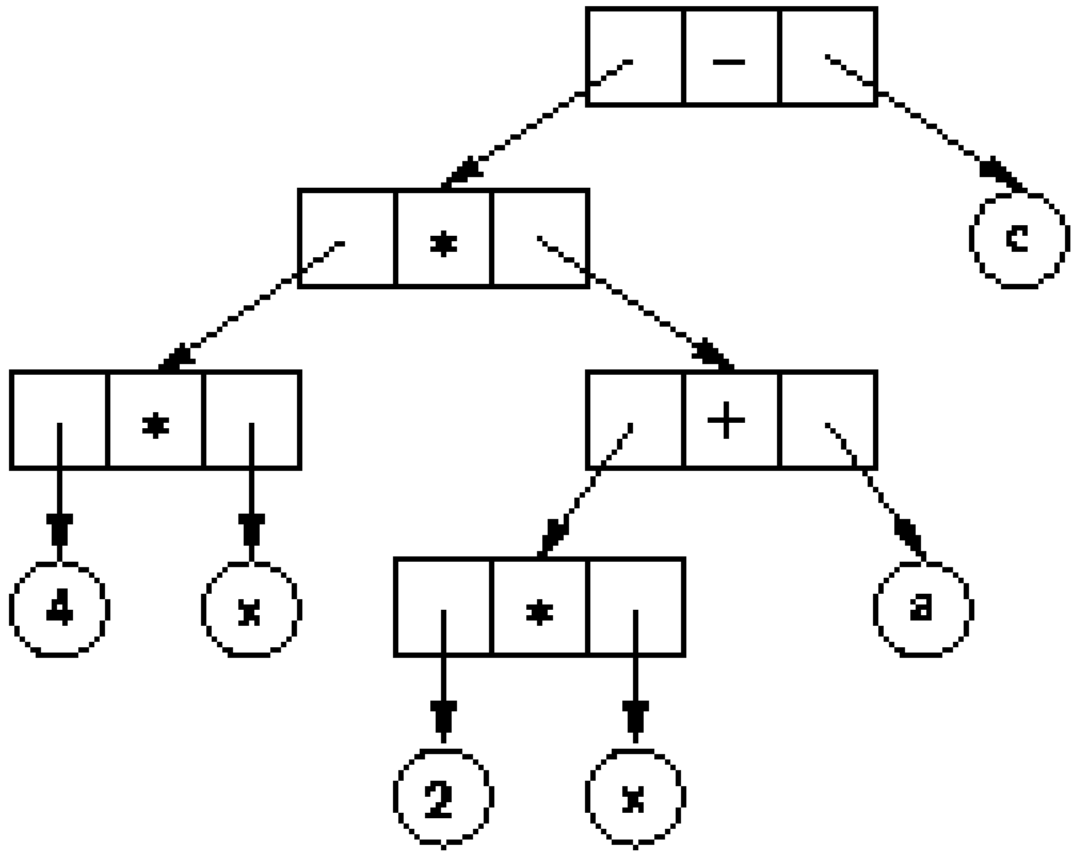
$$= 8 - 1$$

$$= 7$$

Binary Tree Implementation (1)



Binary Tree Implementation (2)



Binary Tree Node Class

```
// Binary tree node class
template <class Elem>
class BinNodePtr : public BinNode<Elem> {
private:
  Elem it;           // The node's value
  BinNodePtr* lc;    // Pointer to left child
  BinNodePtr* rc;    // Pointer to right child
public:
  BinNodePtr() { lc = rc = NULL; }
  BinNodePtr(Elem e, BinNodePtr* l =NULL,
                     BinNodePtr* r =NULL)
    { it = e; lc = l; rc = r; }
```

Binary Tree Node Class (2)

```
  Elem& val() { return it; }
  void setVal(const Elem& e) { it = e; }
  inline BinNode<Elem>* left() const
    { return lc; }
  void setLeft(BinNode<Elem>* b)
    { lc = (BinNodePtr*)b; }
  inline BinNode<Elem>* right() const
    { return rc; }
  void setRight(BinNode<Elem>* b)
    { rc = (BinNodePtr*)b; }
  bool isLeaf()
    { return (lc == NULL) && (rc == NULL); }
};
```

การท่องต้นไม้ (Tree Traversal)

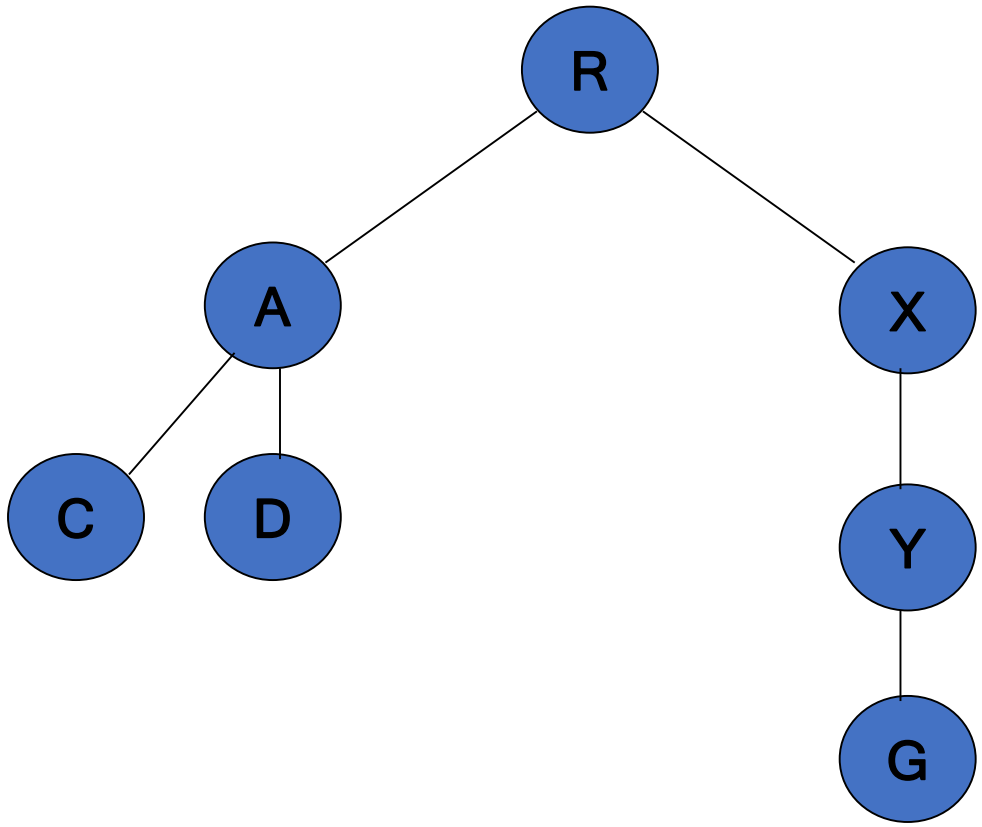
Tree Traversal หมายถึงการไปยังโหนดเพื่อประมวลผลบางอย่างที่ต้องการกระทำกับโหนดนั้น เช่น หาข่าวสาร

แบ่งออกเป็น 3 วิธี (ที่นิยมใช้)

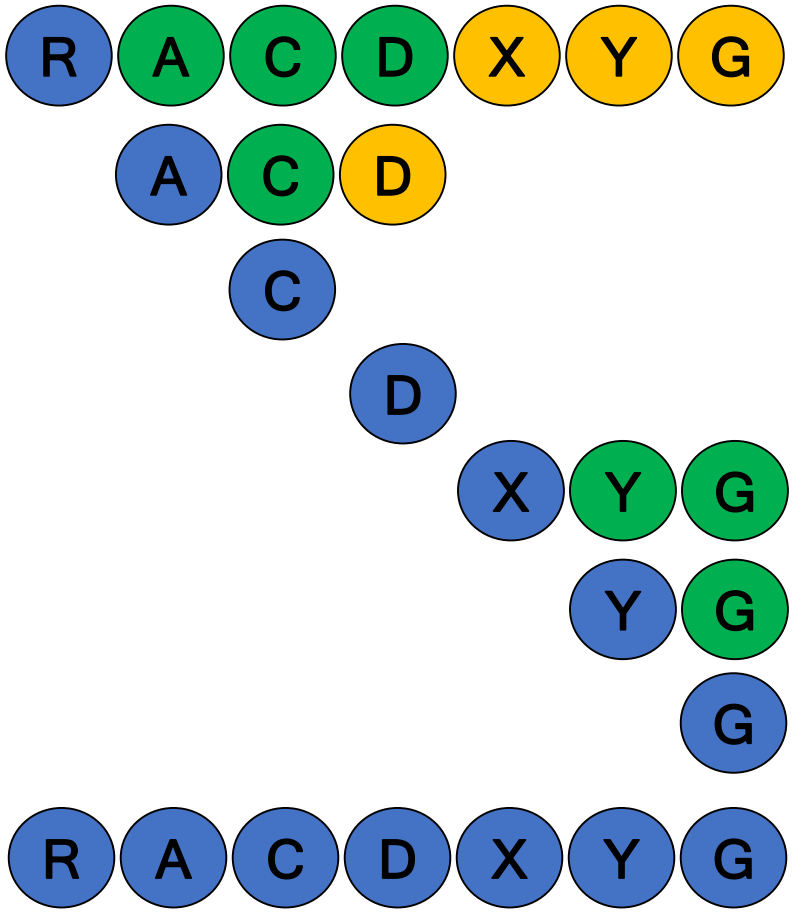
1. Pre-Order Traversal **(RT_LT_R)**
2. In-Order Traversal **(T_LRT_R)**
3. Post-Order Traversal **(T_LT_RR)**

- การข้ามผ่านที่แสดงรายการทุกโหนดในทรี**ครั้งเดียว**จะเรียกว่าการแจงนับโหนดของทรี

Pre-Order Traversal (RT_LT_R)



Result

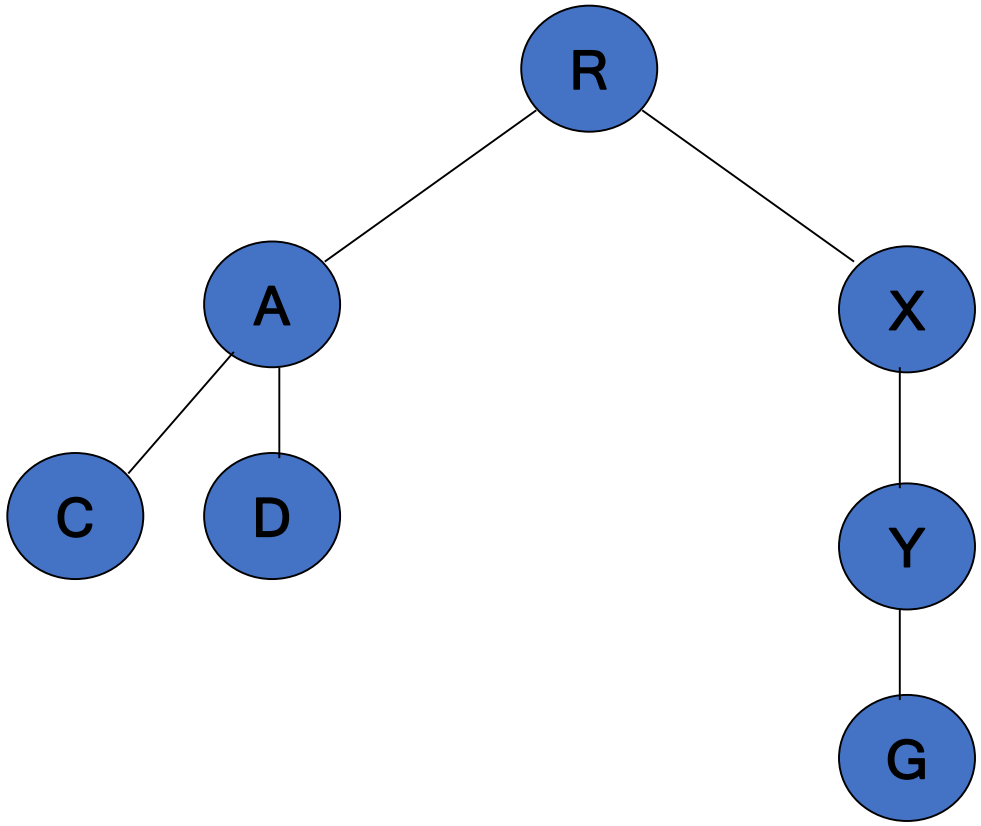


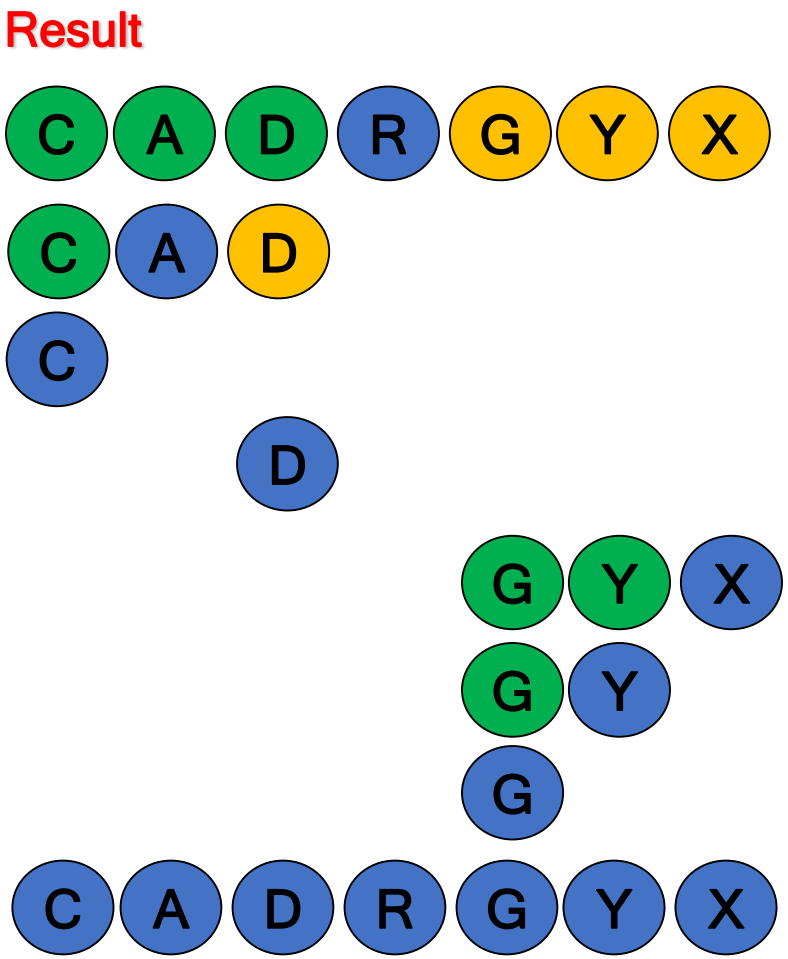
Preorder traversal: Visit each node before visiting its children.

Pre-order Traversals (RT_LT_R)

```
template <class Elem> // Bad implementation
void preorder2(BinNode<Elem>* subroot) {
  visit(subroot);  // Perform some action
  if (subroot->left() != NULL)
    preorder2(subroot->left());
  if (subroot->right() != NULL)
    preorder2(subroot->right());
}

template <class Elem> // Good implementation
void preorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty
  visit(subroot);  // Perform some action
  preorder(subroot->left());
  preorder(subroot->right());
}
```

In-Order Traversal ($T_L R T_R$)



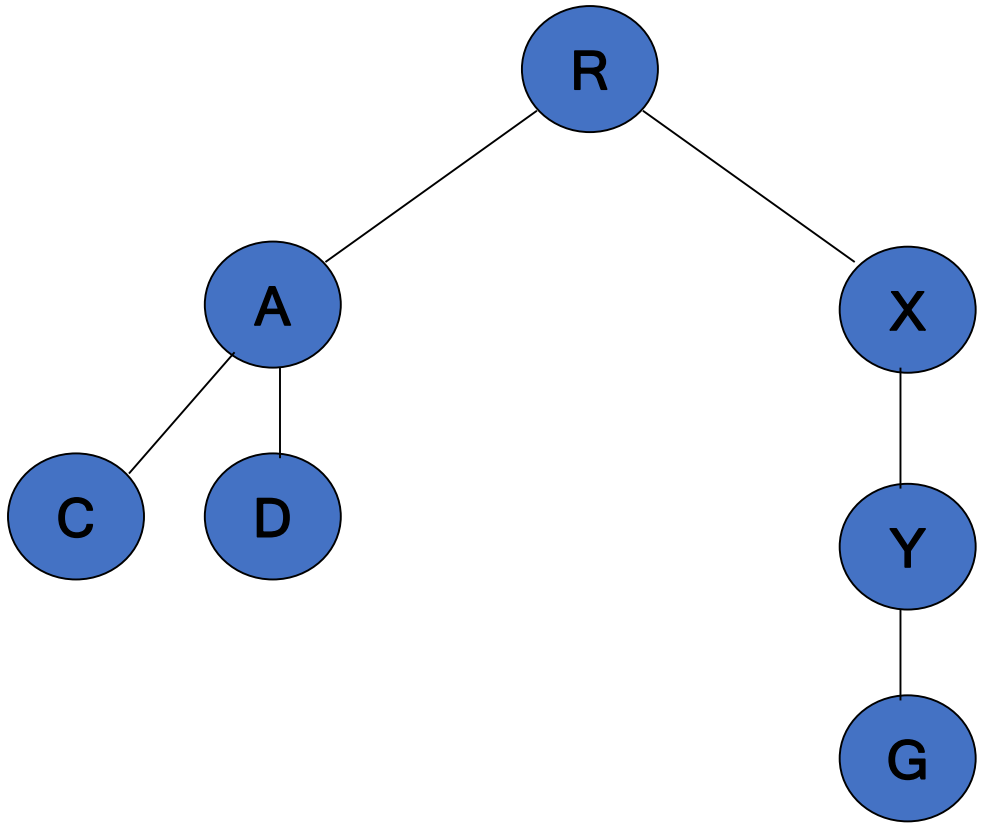


Inorder traversal: Visit the left subtree, then the node, then the right subtree.

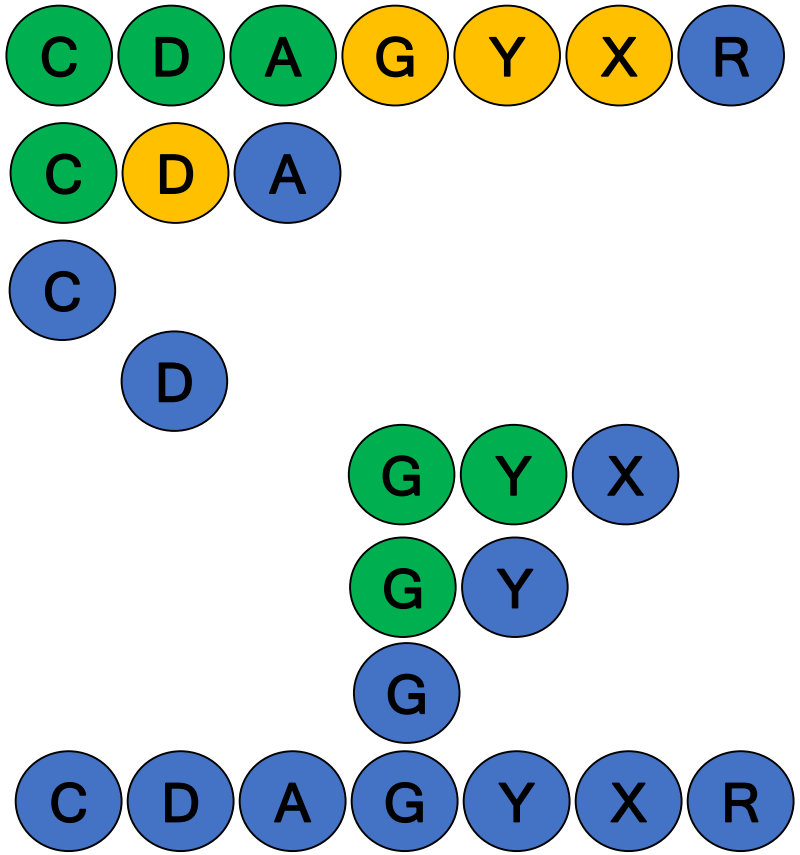
In-Order Traversals ($T_L R T_R$)

```
template <class Elem> // Good implementation
void inorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty
  inorder(subroot->left());
  visit(subroot);  // Perform some action
  inorder(subroot->right());
}
```

Post-Order Traversal ($T_L T_R R$)



Result



Postorder traversal: Visit each node after visiting its children.

Post-Order Traversals ($T_L T_R R$)

```
template <class Elem> // Good implementation
void postorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty

Try to write your own code!!!




}
```

Traversal Example

```
// Return the number of nodes in the tree
template <class Elem>
int count(BinNode<Elem>* subroot) {
  if (subroot == NULL)
    return 0;  // Nothing to count
  return 1 + count(subroot->left())
           + count(subroot->right());
}
```

Space Overhead (1)

From the Full Binary Tree Theorem:

- Half of the pointers are null.

If leaves store only data, then overhead depends on whether the tree is full.

Ex: All nodes the same, with two pointers to children:

- Total space required is $(2p + d)n$
- Overhead: $2pn$
- If $p = d$, this means $2p/(2p + d) = 2/3$ overhead.

Space Overhead (2)

Eliminate pointers from the leaf nodes:

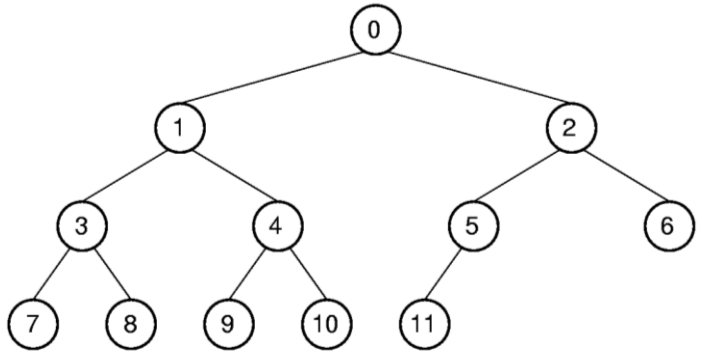
$$\frac{n/2(2p)}{n/2(2p) + dn} = \frac{p}{p + d}$$

This is $1/2$ if $p = d$.

$2p/(2p + d)$ if data only at leaves $\Rightarrow 2/3$ overhead.

Note that some method is needed to distinguish leaves from internal nodes.

Array implementation



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

Array implementation - Root

```
public void Root(String key)
    {
        str[0] = key;
    }
```

Array implementation - set_Left

```
/*create left child of root*/
public void set_Left(String key, int root)
{
    int t = (root * 2) + 1;

    if(str[root] == null){
        System.out.printf("Can't set child at %d, no parent found\n",t);
    }else{
        str[t] = key;
    }
}
```

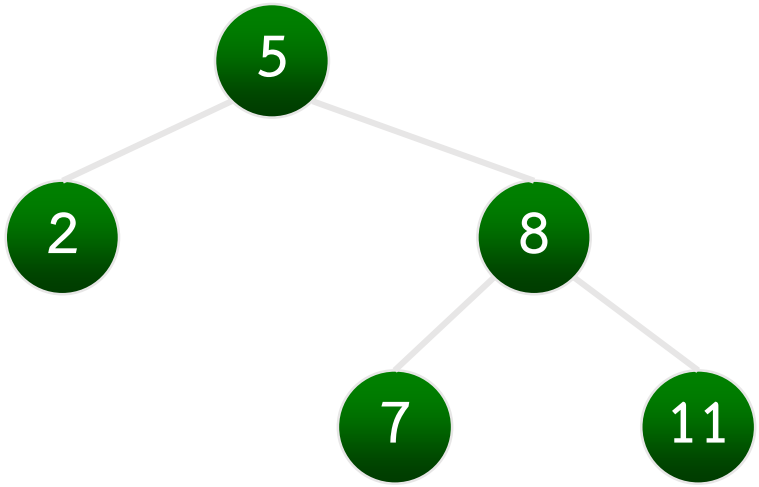
Array implementation - set_Right

```
/*create Right child of root*/
public void set_Left(String key, int root)
{
    int t = (root * 2) + 2;

    if(str[root] == null){
        System.out.printf("Can't set child at %d, no parent found\n",t);
    }else{
        str[t] = key;
    }
}
```

Binary Search Tree

มีคุณสมบัติเป็น Binary Tree โดยที่

- ลูกทางซ้ายต้องน้อยกว่า Parent
- ลูกทางขวาต้องมากกว่า Parent
- ถ้าข้อมูลซ้ำ จะไม่เก็บ

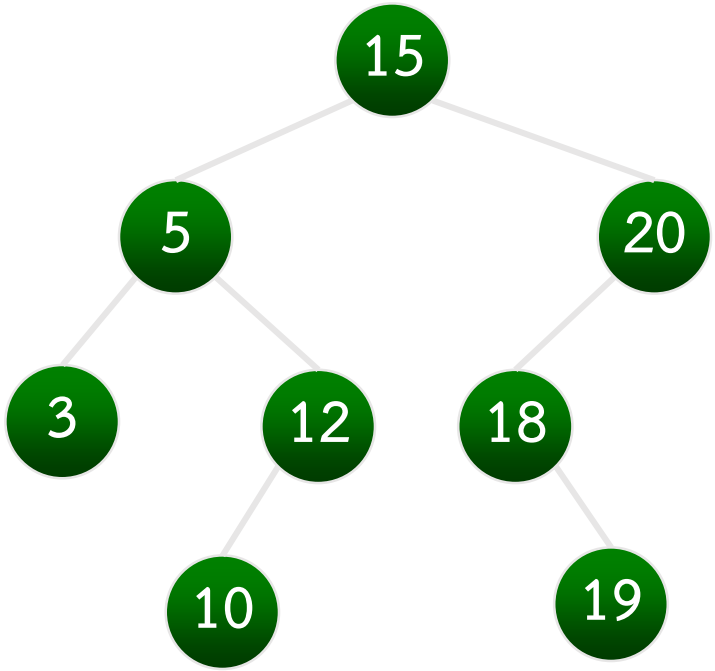


Binary Search Tree

การใส่ข้อมูลใน Binary Search Tree

15 5 12 20 3 18 10 19

- ทุก Node ทาง Left Sub Tree จะน้อยกว่า Root
- ทุก Node ทาง Right Sub Tree จะมากกว่า Root

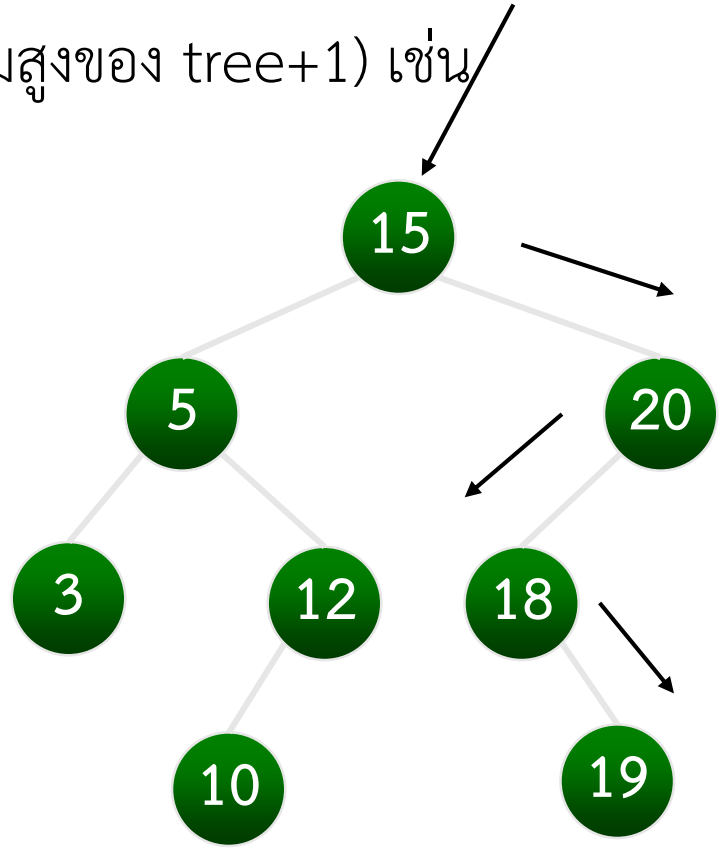


Binary Search Tree

ในการค้นหาข้อมูล ทำได้เร็ว คือไม่เกิน Height+1 (ความสูงของ tree+1) เช่น

- Tree นี้ Height = 3 สมมุติว่าต้องการหาว่า 19 มีหรือไม่
- จะใช้จำนวนครั้งในการหา = 4 ครั้ง

  คือที่ Node 15, 20, 18, 19



BST ADT(1)

```
// BST implementation for the Dictionary ADT
template <class Key, class Elem,
          class KEComp, class EEComp>
class BST : public Dictionary<Key, Elem,
                              KEComp, EEComp> {
private:
  BinNode<Elem>* root;    // Root of the BST
  int nodecount;          // Number of nodes
  void clearhelp(BinNode<Elem>*);
  BinNode<Elem>*
    inserthelp(BinNode<Elem>*, const Elem&);
  BinNode<Elem>*
    deletemin(BinNode<Elem>*,BinNode<Elem>*&);
  BinNode<Elem>* removehelp(BinNode<Elem>*,
                  const Key&, BinNode<Elem>*&);
  bool findhelp(BinNode<Elem>*, const Key&,
                Elem&) const;
  void printhelp(BinNode<Elem>*, int) const;
```

BST ADT(2)

```
public:
  BST() { root = NULL; nodecount = 0; }
  ~BST() { clearhelp(root); }
  void clear() { clearhelp(root); root = NULL;
                 nodecount = 0; }
  bool insert(const Elem& e) {
    root = inserthelp(root, e);
    nodecount++;
    return true; }
  bool remove(const Key& K, Elem& e) {
    BinNode<Elem>* t = NULL;
    root = removehelp(root, K, t);
    if (t == NULL) return false;
    e = t->val();
    nodecount--;
    delete t;
    return true; }
```

BST ADT(3)

```
bool removeAny(Elem& e) { // Delete min value
  if (root == NULL) return false; // Empty
  BinNode<Elem>* t;
  root = deletemin(root, t);
  e = t->val();
  delete t;
  nodecount--;
  return true;
}
bool find(const Key& K, Elem& e) const
  { return findhelp(root, K, e); }
int size() { return nodecount; }
void print() const {
  if (root == NULL)
    cout << "The BST is empty.\n";
  else printhelp(root, 0);
}
```

BST Search

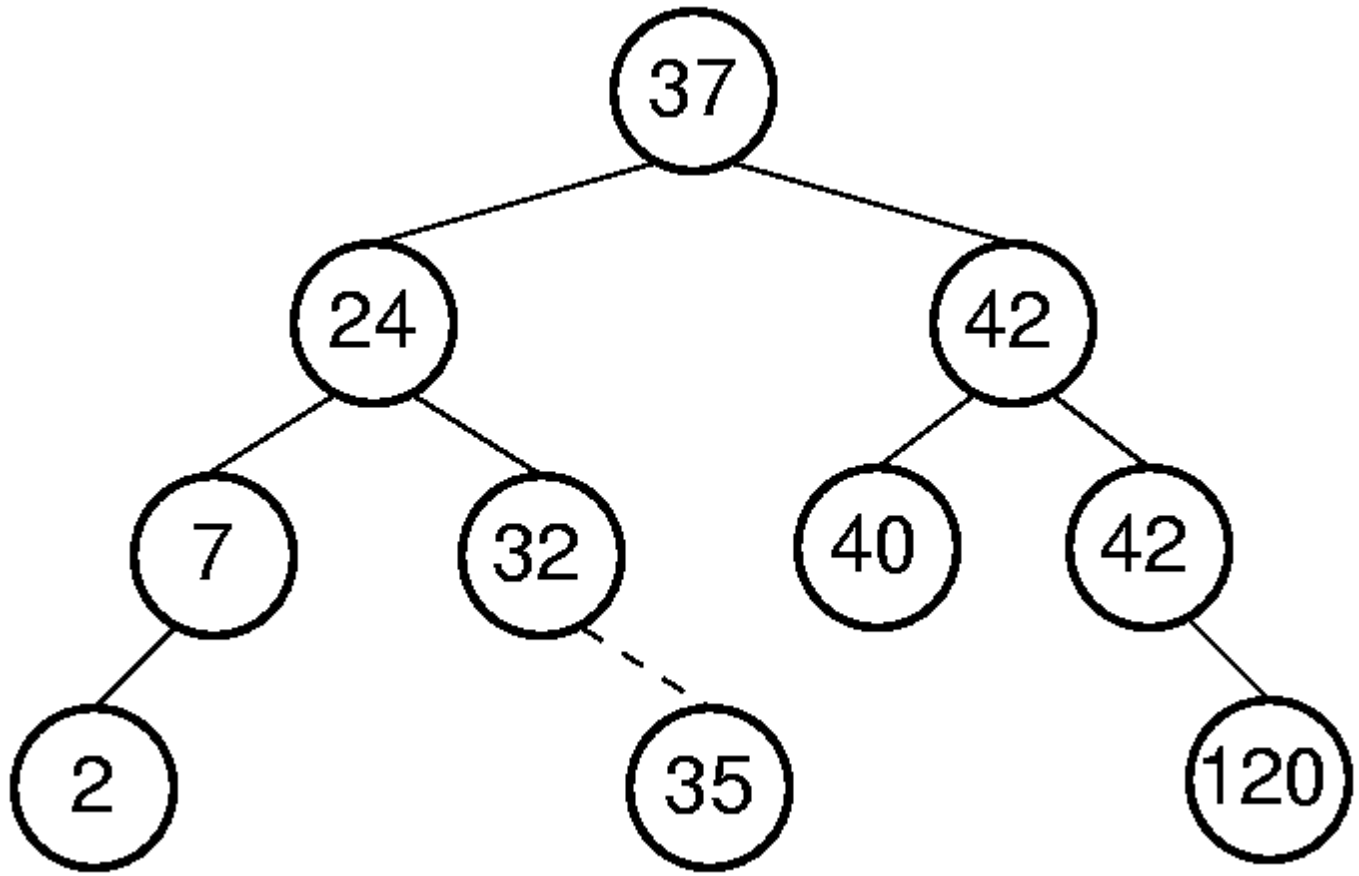
```
template <class Key, class Elem,
          class KEComp, class EEComp>
bool BST<Key, Elem, KEComp, EEComp>::
  findhelp(BinNode<Elem>* subroot,
           const Key& K, Elem& e) const {
  if (subroot == NULL)
      return false;
  else if (KEComp::lt(K, subroot->val()))
      return findhelp(subroot->left(), K, e);
  else if (KEComp::gt(K, subroot->val()))
      return findhelp(subroot->right(), K, e);
  else { e = subroot->val();  return true; }
}
```

BST Search – simpler implementation

```
struct node* search(struct node* root, int key)     // n
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);           // n/2

    // Key is smaller than root's key
    return search(root->left, key);               // n/2
}
```

O(height) = $O(\log_2 n)$

BST Insert



BST Insert

```
template <class Key, class Elem,
          class KEComp, class EEComp>
BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
 inserthelp(BinNode<Elem>* subroot,
             const Elem& val) {
 if (subroot == NULL) // Empty: create node
   return new BinNodePtr<Elem>(val,NULL,NULL);
 if (EEComp::lt(val, subroot->val()))
   subroot->setLeft(inserthelp(subroot->left(),val));
 else
   subroot->setRight(inserthelp(subroot->right(), val));
 // Return subtree with node inserted
 return subroot;
}
```

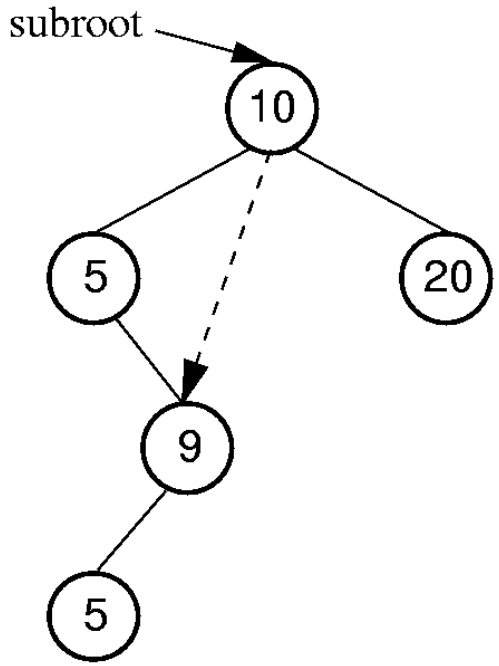
BST Insert – Simpler implementation

```
struct node* insert(struct node* node, int key)    // n
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);     // n/2
    else if (key > node->key)
        node->right = insert(node->right, key);    // n/2

    /* return the (unchanged) node pointer */
    return node;
}
```

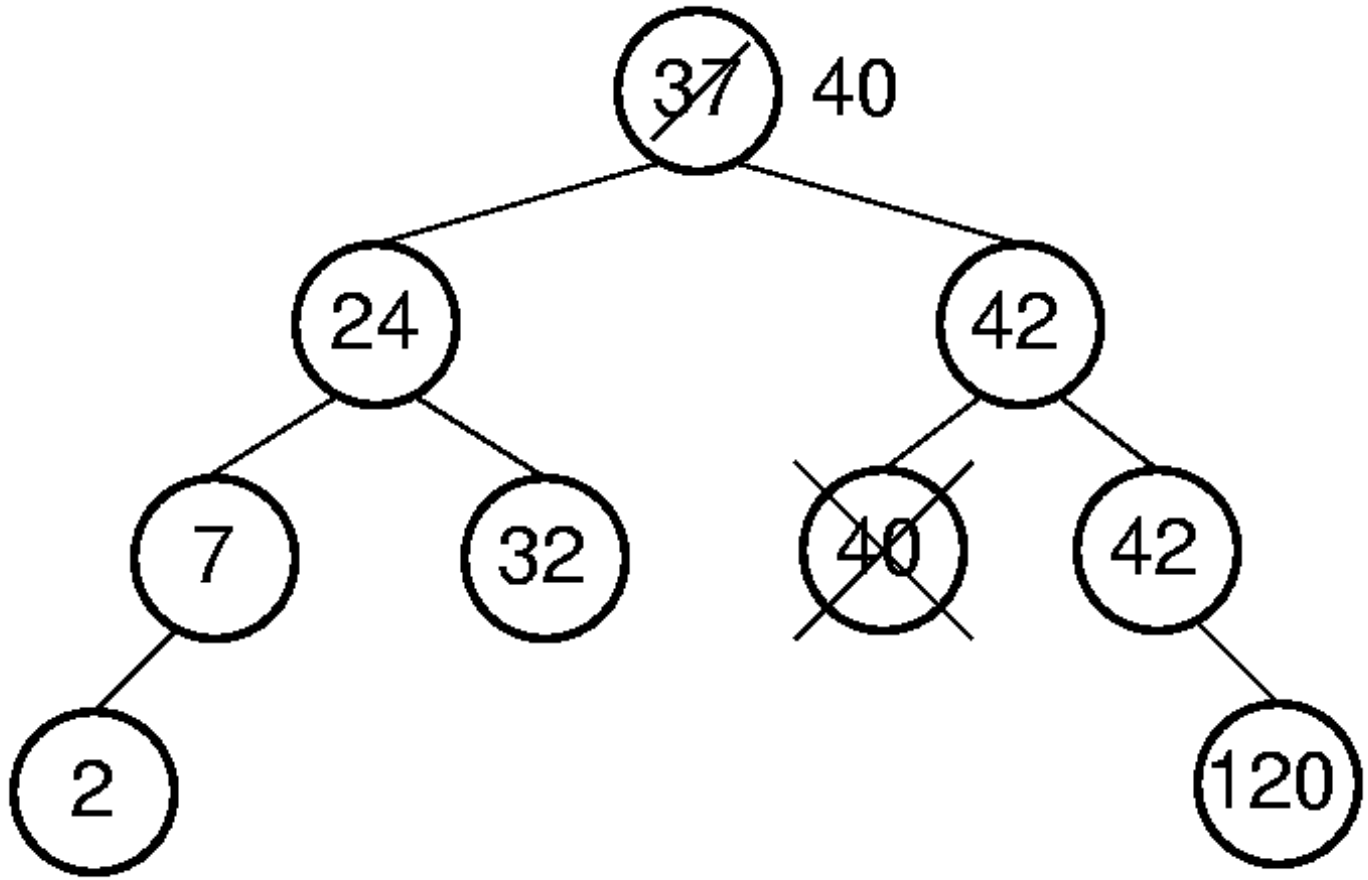
O(height) = $O(\log_2 n)$

Remove Minimum Value

```
template <class Key, class Elem,
     class KEComp, class EEComp>
BinNode<Elem>* BST<Key, Elem,
                  KEComp, EEComp>::
deletemin(BinNode<Elem>* subroot,
          BinNode<Elem>*& min) {
  if (subroot->left() == NULL) {
    min = subroot;
    return subroot->right();
  }
  else { // Continue left
    subroot->setLeft(
      deletemin(subroot->left(), min));
    return subroot;
  }
}
```



BST Remove



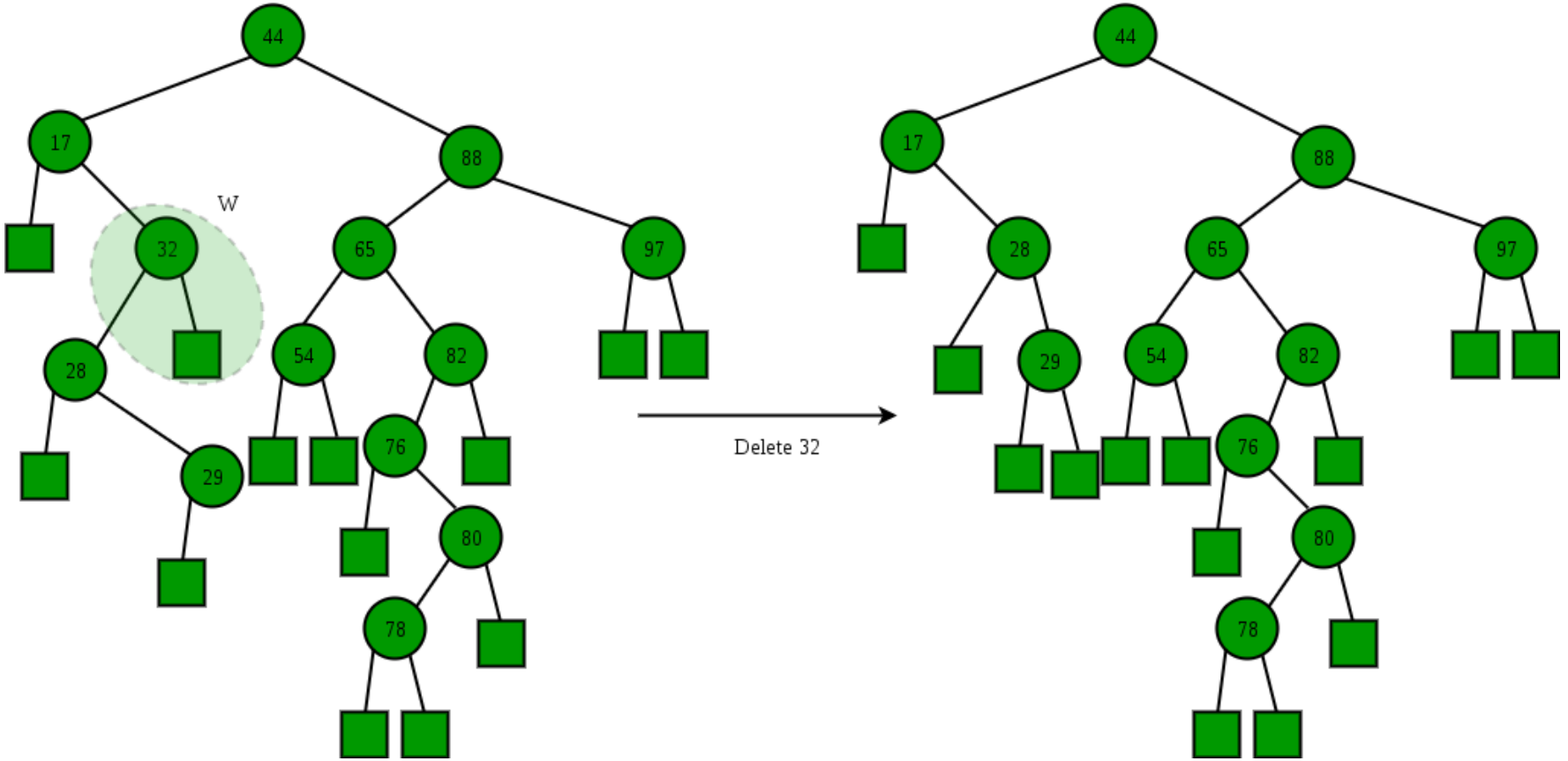
BST Remove

```
template <class Key, class Elem,
          class KEComp, class EEComp>
BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
removehelp(BinNode<Elem>* subroot,
           const Key& K, BinNode<Elem>*& t) {
  if (subroot == NULL) return NULL;
  else if (KEComp::lt(K, subroot->val()))
    subroot->setLeft(
        removehelp(subroot->left(), K, t));
  else if (KEComp::gt(K, subroot->val()))
    subroot->setRight(
        removehelp(subroot->right(), K, t));
```

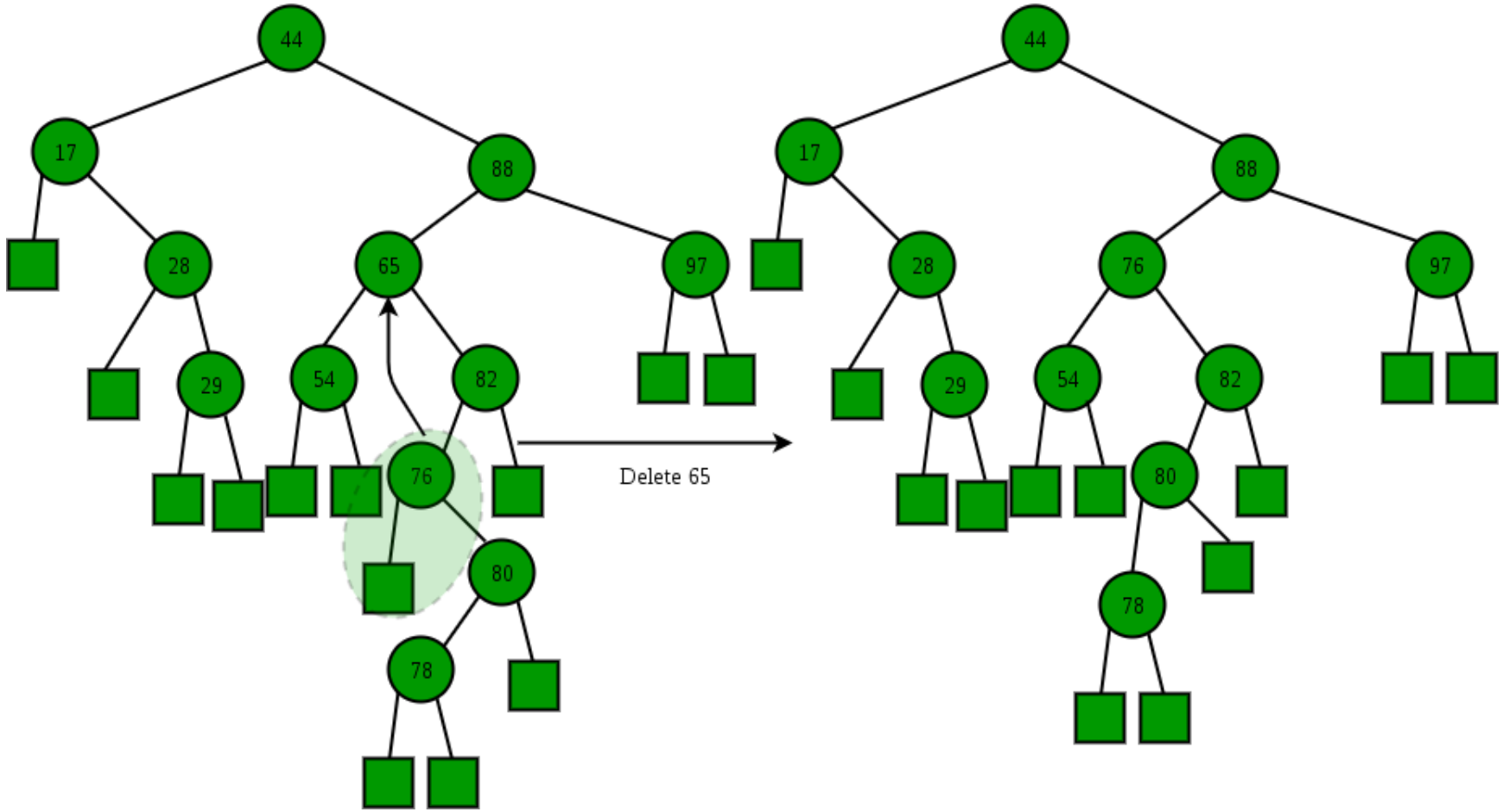
BST Remove

```
 else {               // Found it: remove it
    BinNode<Elem>* temp;
    t = subroot;
    if (subroot->left() == NULL)
      subroot = subroot->right();
    else if (subroot->right() == NULL)
      subroot = subroot->left();
    else {  // Both children are non-empty
      subroot->setRight(
          deletemin(subroot->right(), temp));
      Elem te = subroot->val();
      subroot->setVal(temp->val());
      temp->setVal(te);
      t = temp;
    } }
  return subroot;
}
```

BST Remove


44
17
88
W
32
65
97
28
54
82
29
76
80
78
Delete 32
44
17
88
28
65
97
29
54
82
76
80
78

BST Remove


44
17
88
28
65
97
29
54
82
76
80
78
Delete 65
44
17
88
28
76
97
29
54
82
80
78

```
struct node* deleteNode(struct node* root, int key) // n
{
    // base case
    if (root == NULL) return root;
    // If the key to be deleted is smaller than the root's key,  then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key); // n/2
    // If the key to be deleted is greater than the root's key,  then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key); // n/2
    // if key is same as root's key, then This is the node  to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {   struct node *temp = root->right;
            free(root);
            return temp; }
        else if (root->right == NULL)
        {   struct node *temp = root->left;
            free(root);
            return temp; }
        // node with two children: Get the inorder successor (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key); // n/2
    }
    return root;
}
```

O(height) = $O(\log_2 n)$

แบบฝึกหัด

1. จงเปลี่ยนนิพจน์ Infix ต่อไปนี้เป็น Postfix

- A + B * C – D / E
- (A - 2 * (B + C) – D * E) * F

2. จงเปลี่ยนนิพจน์ Infix ต่อไปนี้เป็น Prefix

(50 - 2 * (2 + 3) - 5 * 4) * 6

3. จงคำนวณหาค่าของนิพจน์ Prefix ที่ได้ในข้อ 2

แบบฝึกหัด

- ให้สร้าง Binary Search Tree จากข้อมูลดังนี้

  10, 8, 2, 4, 3, 15, 26, 30 ,17, 6

- จากรูป Tree ที่กำหนดให้ จงเขียนผลลัพธ์จากการท่องไปใน Tree ทั้ง 3 แบบได้แก่ Pre-order, In-order และ Post-order