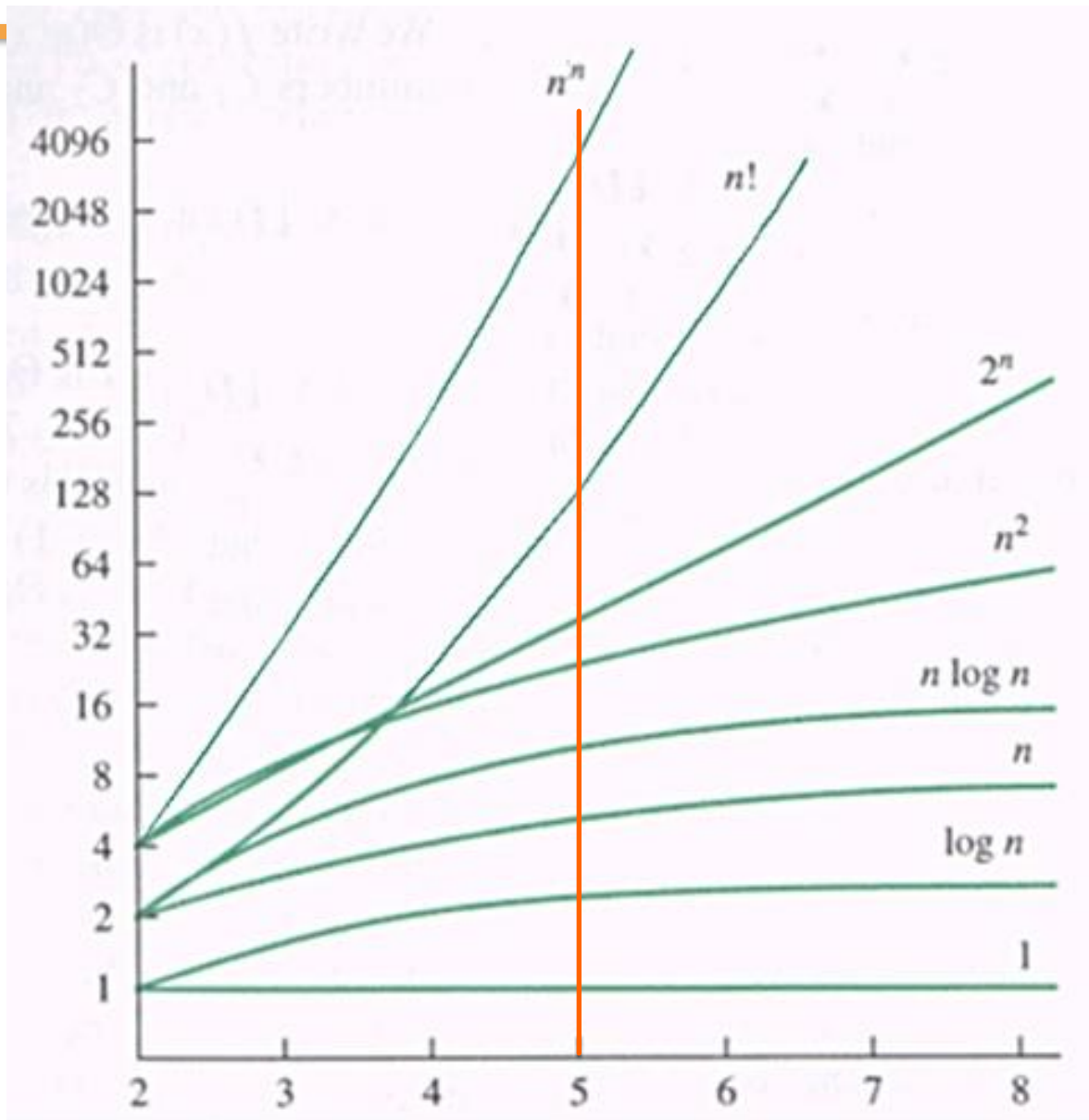




Module4—Tree

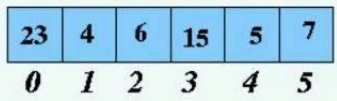
	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



Data Structure

By: @pythoncodess

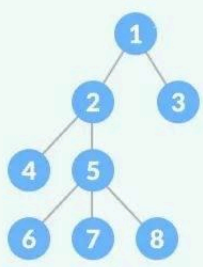
Array



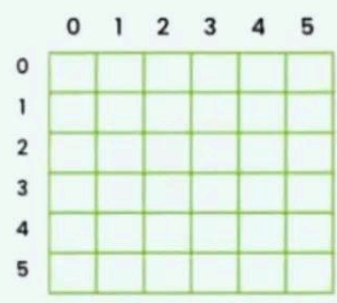
Queue



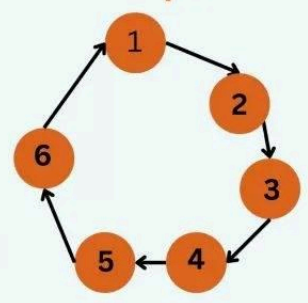
Tree



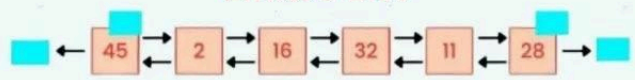
Matrix



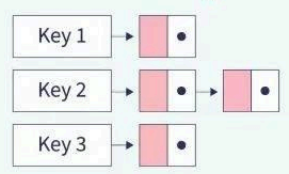
Graph



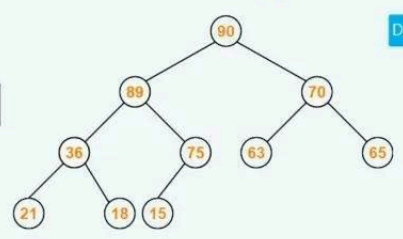
Linked List



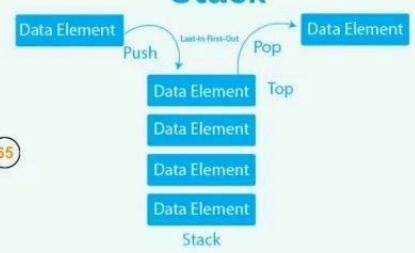
Hashmap



Max Heap



Stack





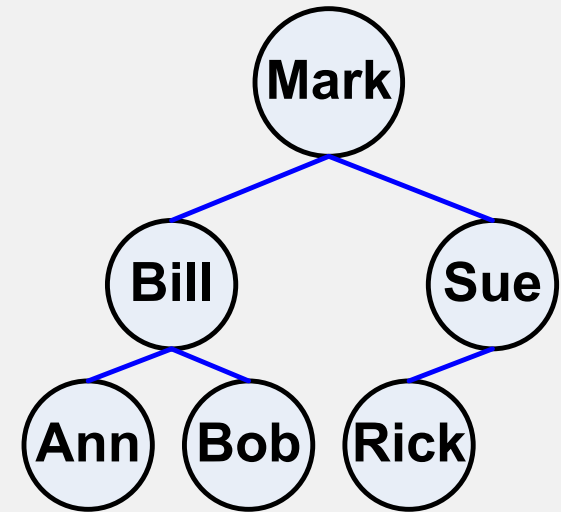
4 Trees

Problem

- Linear time access of linked list.
- Running time of operation $O(n)$.

Correct : Trees

- Average time $O(\log_n)$.
- Worst case $O(n)$.

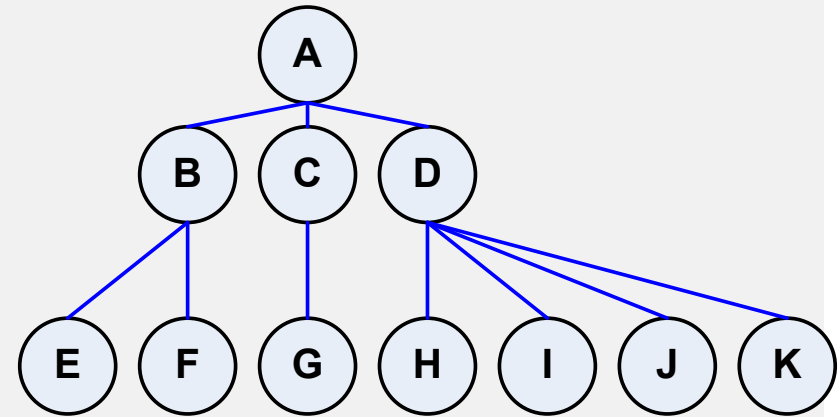




4.1 Tree Definition

โครงสร้างข้อมูลต้นไม้ (Tree Data Structure) หรือเรียกสั้นๆว่าทรี (Tree) เป็นโครงสร้างข้อมูลรูปแบบหนึ่งในลักษณะ

- โครงสร้างข้อมูลชนิดไม่เชิงเส้น (Non-Linear)
- สมาชิกแต่ละตัวในทรีสามารถเชื่อมโยงไปยังสมาชิกตัวถัดไป (Successor) ได้มากกว่าหนึ่งตัว
- และเชื่อมโยงถึงกันในลักษณะเป็นระดับคล้ายกับการแตก กิ่งก้านสาขาออกไปของต้นไม้
- ความสัมพันธ์ของสมาชิกข้อมูลในทรี จึงมีลักษณะลำดับชั้น (Hierarchical Relationship) คือ มีการเชื่อมโยงของแต่ละโหนดเป็นแบบทางเดียวจากบนลงล่าง
- โครงสร้างข้อมูลทรีประกอบด้วย **โหนด** (Node) สำหรับจัดเก็บข้อมูล และ **กิ่งหรือเส้น** ที่เชื่อมโยง

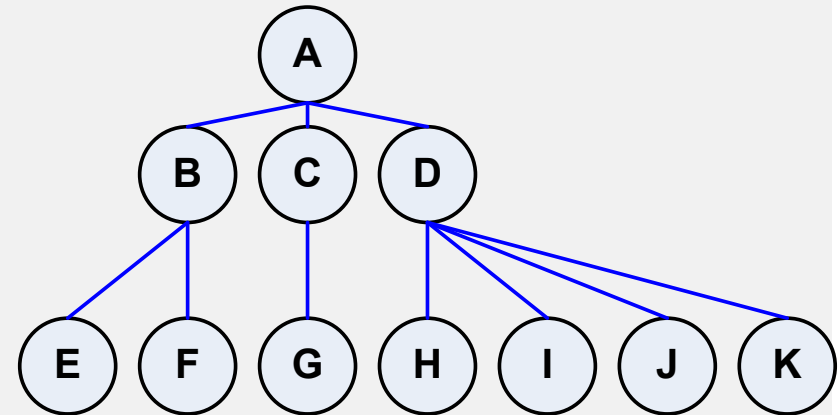


<https://www2.cs.science.cmu.ac.th/courses/204251/lib/exe/fetch.php?media=tree.pdf>



4.1 Tree Definition

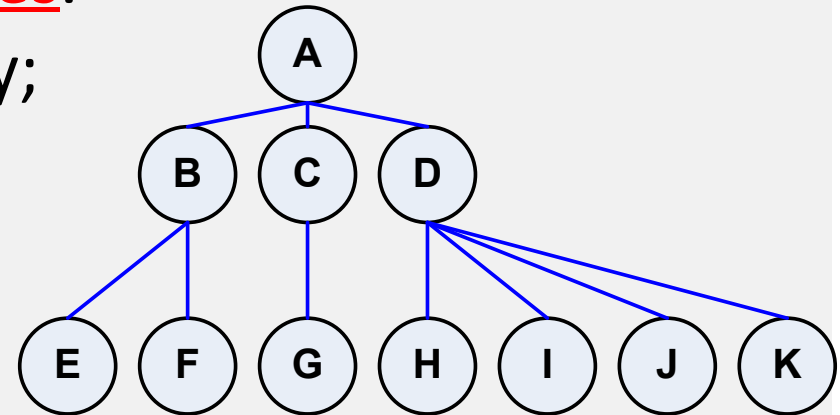
A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each **node** is a data structure consisting of a value,



together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

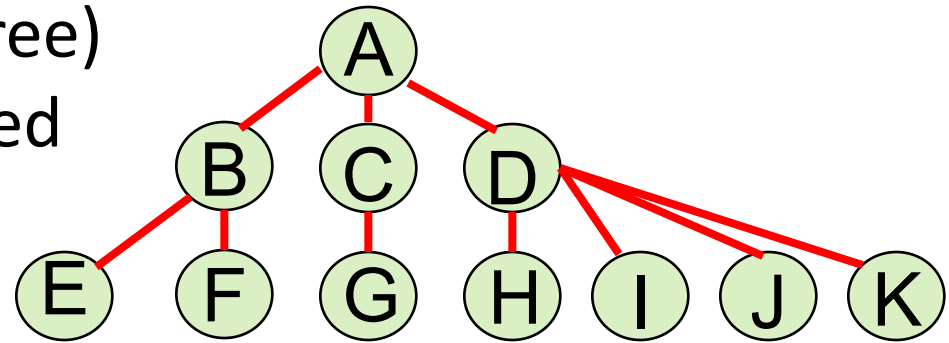


- ❑ A tree is a collection of nodes.
- ❑ The collection can be empty;
- ❑ Otherwise,
 - ❑ a tree consists of a distinguished node r , called the root,
 - ❑ and zero or more nonempty (subtrees), T_1, T_2, \dots, T_k





- ❑ each of whose root(Sub tree) are connected by a directed edge from r.



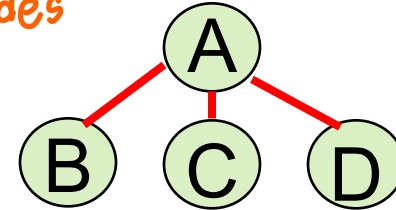
- ❑ A root of each subtree is said to be a child of r,
- ❑ And r is the parent of each subtree root.



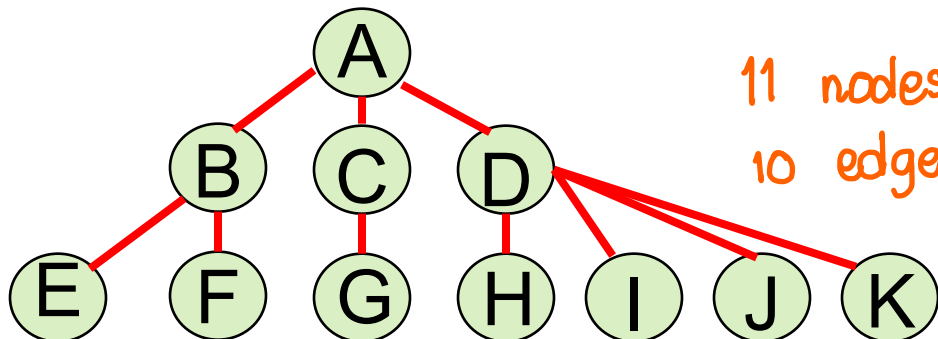
Recursive definition

- ❑ A tree is a collection of N nodes,
- ❑ one of which is the root, and $N-1$ edges.
- ❑ That there are $N-1$ edges follows from the fact that each edge connects some node to its parents.
- ❑ And every node except the root has one parent.

$N=4$ nodes



edge
 $= N-1$

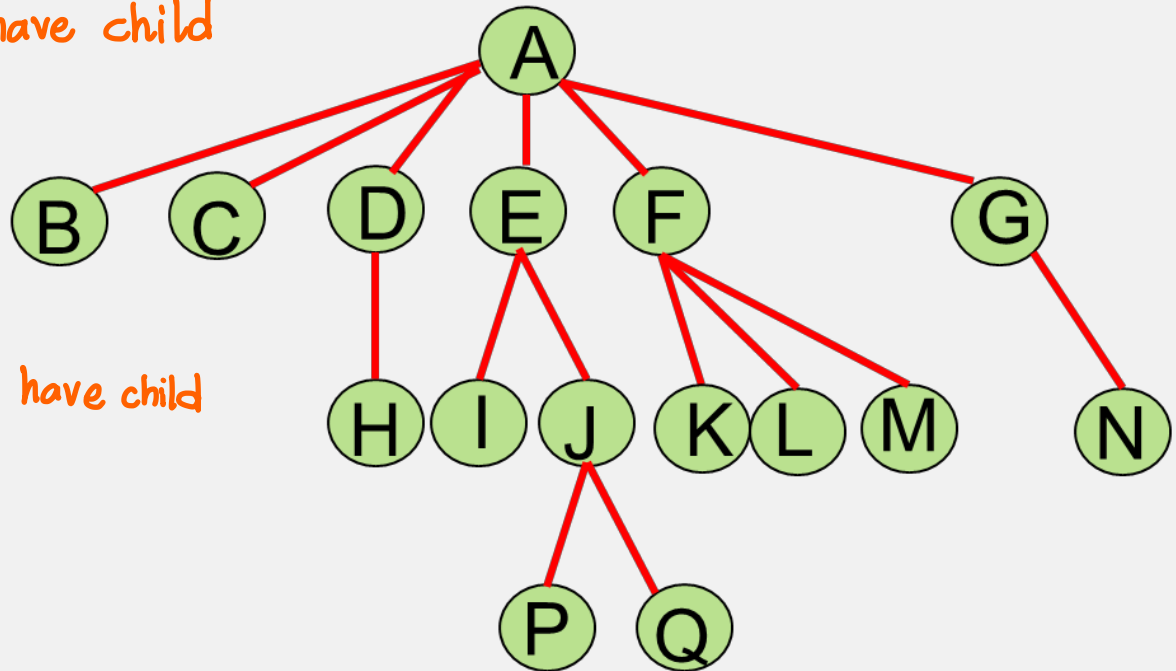


11 nodes
10 edges



นิยามที่ใช้กับ Tree

- ❑ Leaves *nodes not have child*
(Terminal)
- ❑ Parents
- ❑ Siblings *พี่น้อง*
- ❑ Non Leaves *nodes have child*
(Non terminal)





นิยามที่ใช้กับ Tree

- ❑ **Degree** : The number of children of a node x in a rooted tree T .

จำนวนลูก
ๆ

A = 6 E = 2
B = 0

- ❑ **Path** from node n_1 to n_k :

Sequence of nodes

n_1, n_2, \dots, n_k such that n_i is

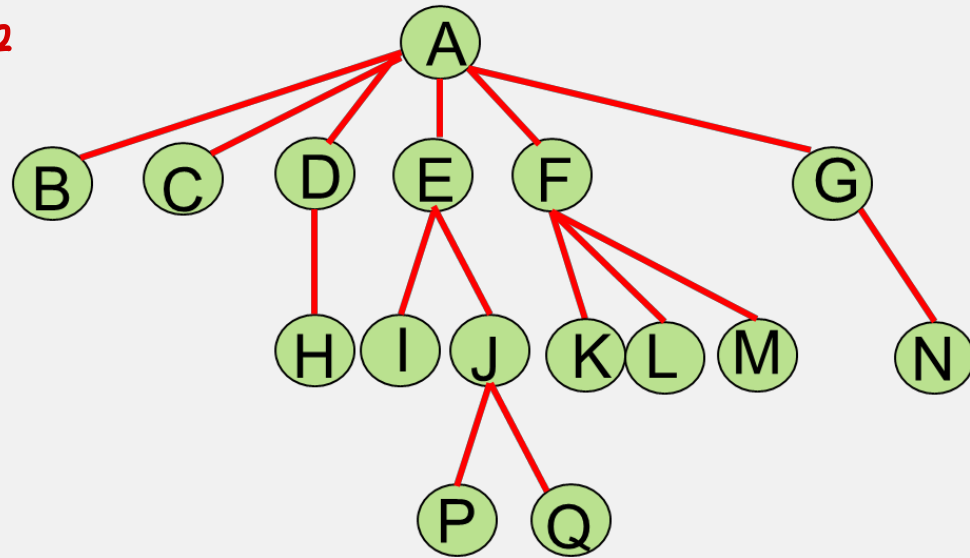
the parent of n_{i+1} for

$1 \leq i \leq k$.

เส้นทาง

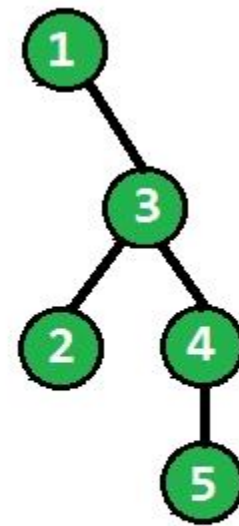
AQ

AE, EJ, JQ





- ❑ **Depth** : For Any node n_i , the **depth** of n_i is the length of the unique path from the root to n_i .
- ❑ **Height** : Is the longest path from n_i to a leaf. All leaves are at height 0. The height of a tree is equal to the Height of the root.



Depth 0

Depth 1

Depth 2

Depth 3

Height 3

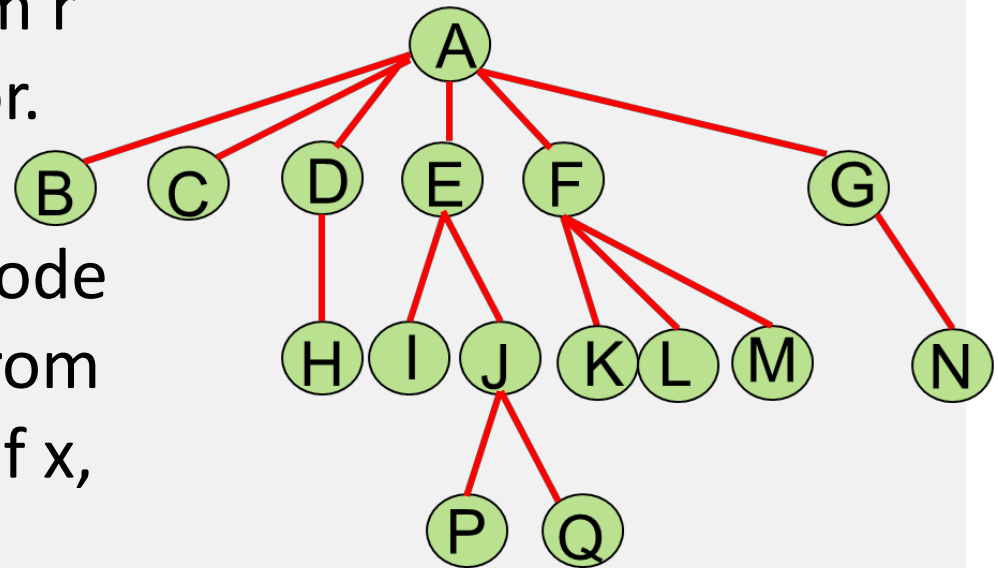
Notice that in a tree there is exactly one path from the root to each node.



- ❑ **Ancestor** of x : Any node y on the unique path from r to x is called an ancestor.

x to y

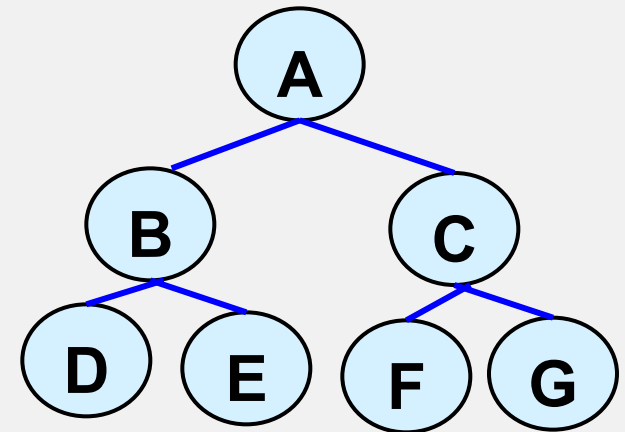
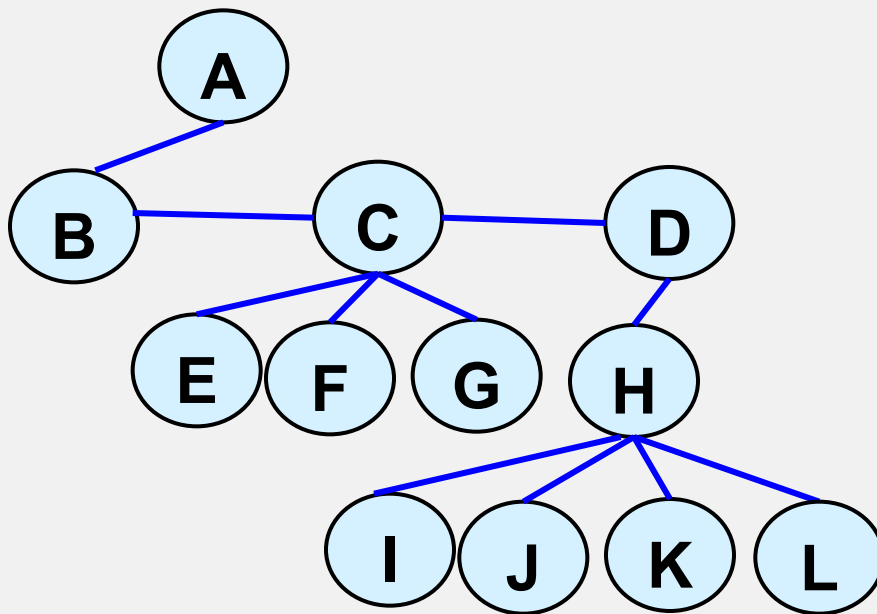
- ❑ **Descendant** of y : Any node x on the unique path from y to x , x is descendant of y . Every node is both an ancestor of and a descendant of itself.



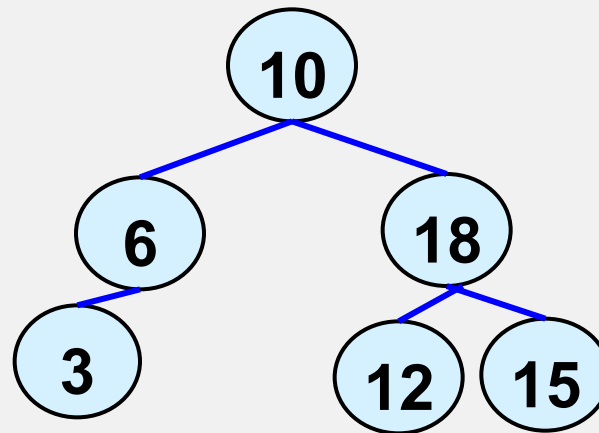
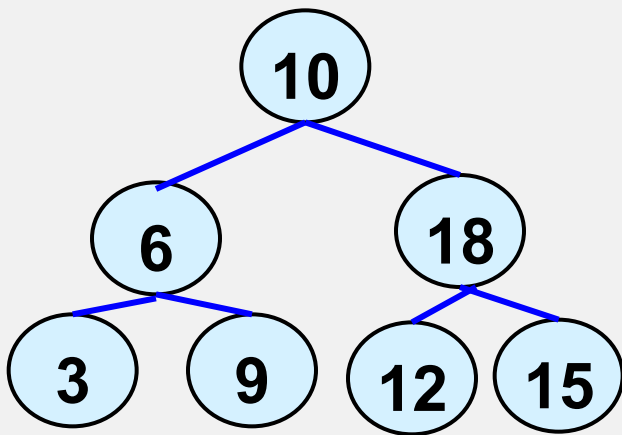
4.2 Binary tree

have child 0 or Not more than 2

1) A Binary tree is a tree in which no node can have more than two children.



2) Full Binary tree (Complete Binary tree) : Binary tree which each node is either a leaf or has a degree exactly 2

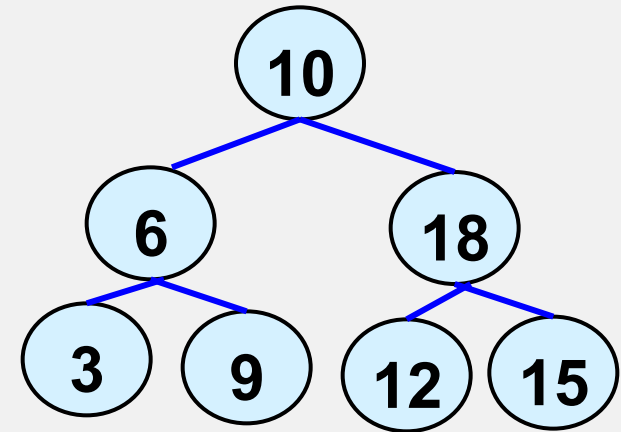




4.3 Binary search tree

- Special type of binary tree,
- The keys in a binary search tree are always stored in such a way as to satisfy the binary search tree property:

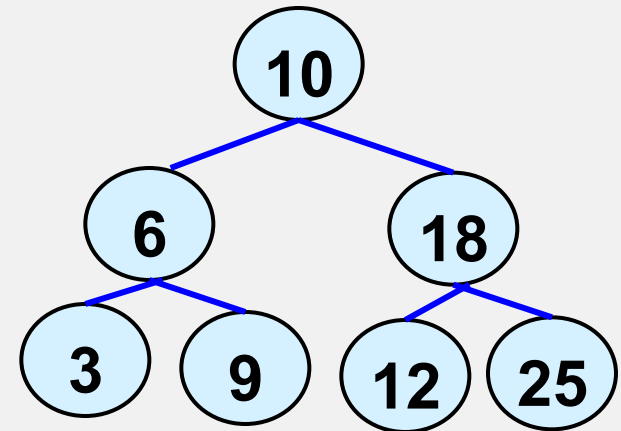
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x , then $\text{key } y \leq \text{key } x$. If y is a node in the right subtree of x , then $\text{key } x \leq \text{key } y$.





4.3.1 Tree Traversal

Binary search tree property allow us to print out all the keys in a tree in sorted order by a simple recursive algorithm called inorder **tree walk**.



1) Preorder

Root Left Right

2) Inorder

Left Root Right

จากน้อยไปมาก

3) Postorder

Left Right Root

Right Root Left จากมากไปน้อย

PreOrder : A B C D E F G H I J

InOrder : A E D F C B H G J I

PostOrder : E F D C H J I G B A



4.3.2 Operation

1. Insert
2. Delete
3. Print :
 - Preorder,
 - Inorder,
 - Postorder
4. Find

Example 1

```
#include <iostream>
#include <stdio.h>
using namespace std;
struct node
{ int value;
  struct node *left;
  struct node *right;
};
```



```
struct node *insert(struct node *tree, int x)
{
    .....
}

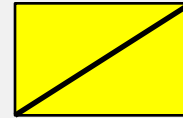
int main()
{
    struct record *tree=NULL;
    tree=insert(tree,5);
}
```



NULL 5

```
struct node *insert(struct node *tree, int x)
1 { if(tree==NULL)
2   { tree = new struct node;
3     tree->value = x;
4     tree->left = tree->right = NULL;
5   }
6   else
7   { if( x < tree->value )
8     tree->left = insert(tree->left, x);
9     else if(x > tree->value)
10    tree->right = insert(tree->right, x);
11  }
12  return tree; }
```

tree



tree



2000

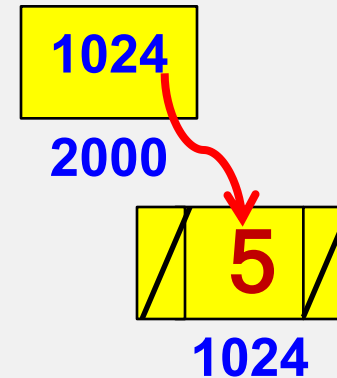


1024



```
struct node *insert(struct node *tree, int x)
1 { if(tree==NULL)
2   { tree = new struct node;
3     tree->value = x;
4     tree->left = tree->right = NULL;
5   }
6   else
7   { if( x < tree->value )
8     tree->left = insert(tree->left, x);
9     else if(x > tree->value)
10      tree->right = insert(tree->right, x);
11  }
  return tree; }
```

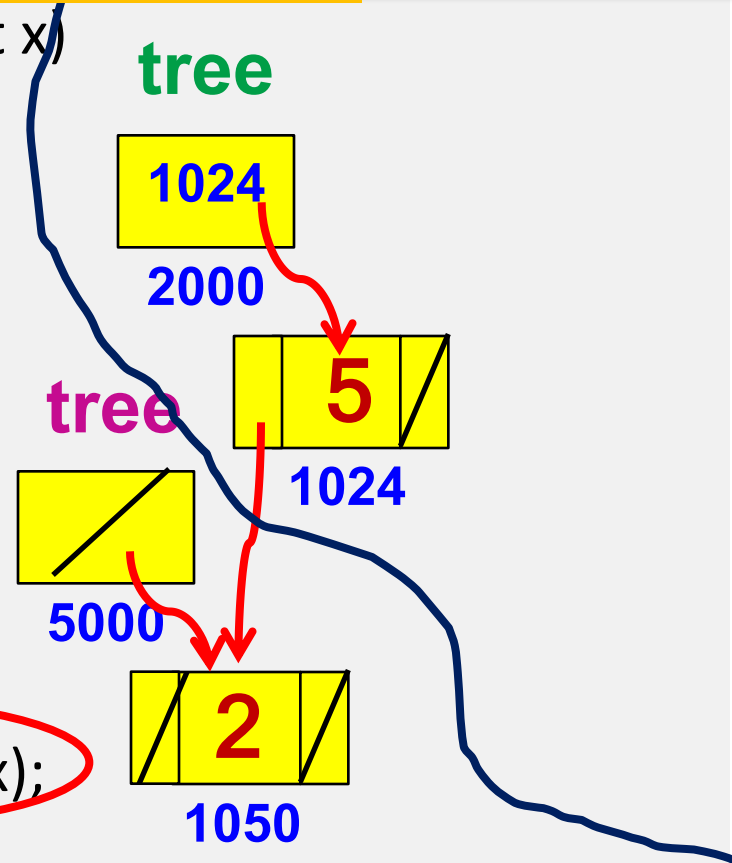
tree



```

struct node *insert(struct node *tree, int x)
1 { if(tree==NULL)
2   { tree = new struct node;
3     tree->value = x;
4     tree->left = tree->right = NULL;
5   }
6   else
7   { if( x < tree->value )
8     tree->left = insert(tree->left, x);
9     else if(x > tree->value)
10      tree->right = insert(tree->right, x);
11  }
    return tree; }

```





```
void print(struct node *tree)
```

```
1{  if ( tree == NULL )
```

```
2    return;
```

```
3  else
```

```
4  {  cout << tree->value << endl;
```

```
5    print(tree->left);
```

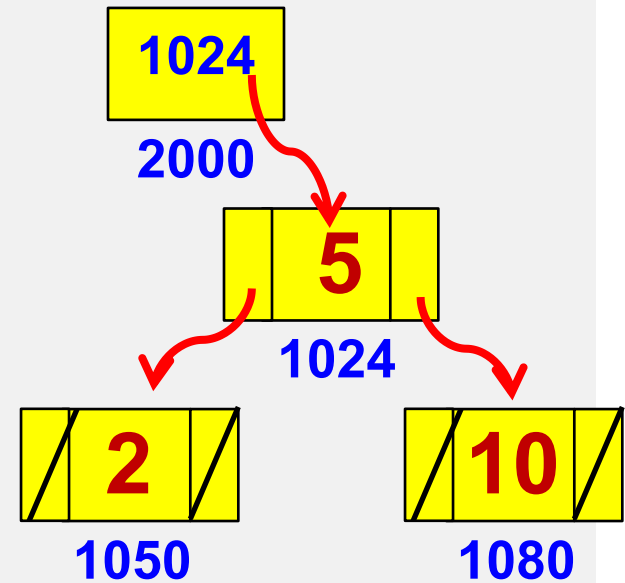
```
6    print(tree->right);
```

```
    }
```

```
7  return;
```

```
}
```

tree





Time complexity:

Best case: $O(1)$

Average case: When there is a balanced binary search tree (a binary search tree is called balanced if height difference of nodes on left and right subtree is not more than one), so height becomes $\log N$ where N is number of nodes in a tree.

searching is $O(\log N)$

Worst case: $O(N)$

Trees

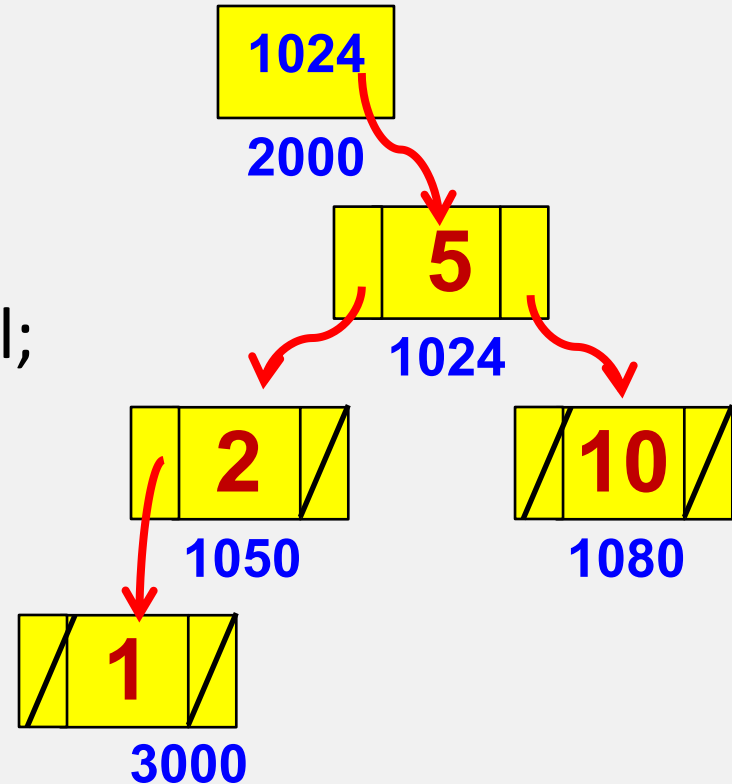
- Insert
- Print
- Search
- Find Min
- Delete



```
void print(struct node *tree)
{
    if ( tree == NULL )
        return;
    else
    {
        cout << tree->value << endl;
        print(tree->left);
        print(tree->right);
    }
    return;
}
```

Preorder

tree





```
void print(struct node *tree)
```

```
1{  if ( tree == NULL )
```

```
2    return;
```

```
3  else
```

```
4  {  print(tree->left);
```

```
5      cout << tree->value << endl;
```

```
6      print(tree->right);
```

```
    }
```

```
7  return;
```

```
}
```

tree

