제4장 트리

트리, 이진트리, 트리순회, 트리연산, 상호배타적 집합

트리

- **배열이나 연결리스트의 단점**
  - 데이터를 일렬로 저장하기 때문에 탐색 연산이 순차적으로 수행되는 단점
  - 배열은 미리 정렬해 놓으면 이진탐색을 통해 효율적인 탐색이 가능하지만,
    삽입이나 삭제 후에도 정렬 상태를 유지해야 하므로
    삽입이나 삭제하는데 O(N) 시간 소요
- **트리를 이용하면 O(logN)에 기반한 자료저장/삭제/탐색이 가능**

응용

- 조직이나 기관의 계층구조
- 컴퓨터 운영체제의 파일 시스템
- 자바 클래스 계층구조 등
- 트리는 일반적인 트리와 이진트리(Binary Tree)로 구분
- 다양한 탐색트리(Search Tree), 힙(Heap) 자료구조,
  컴파일러의 수식을 위한 구문트리(Syntax Tree) 등의
  기본이 되는 자료구조로서 광범위하게 응용

4.1 트리

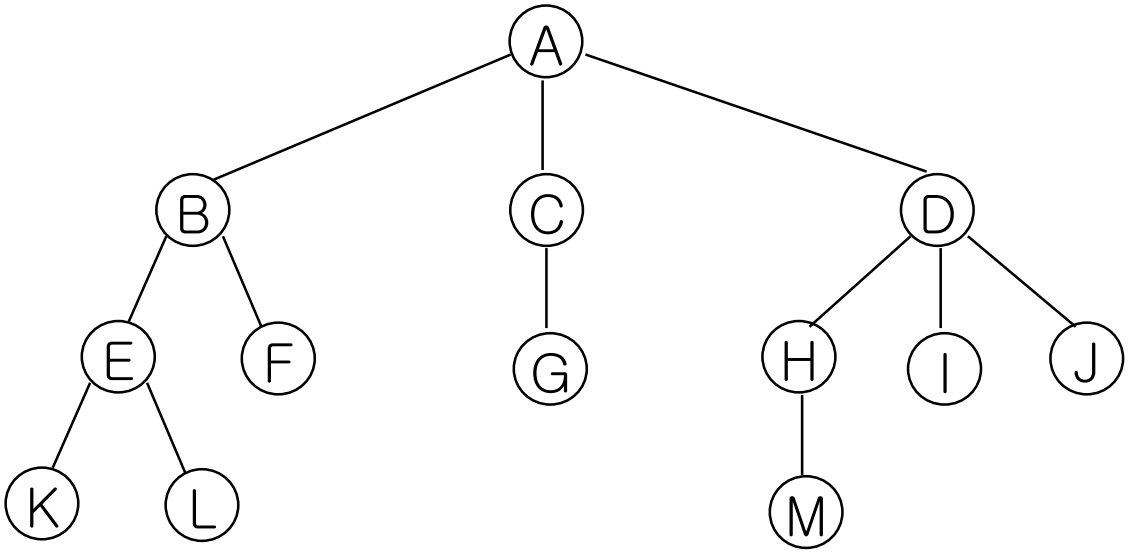
- **일반적인 트리(General Tree)는 실제 트리를 거꾸로 세워 놓은 형태의 자료구조**
  - 응용 분야 : HTML과 XML 의 문서 트리, 자바 클래스 계층구조, 운영체제의 파일시스템, 탐색트리, 이항(Binomial)힙, 피보나치(Fibonacci)힙과 같은 우선순위큐(7장)에서 사용
- **일반적인 트리의 정의**

트리는 empty이거나,
empty가 아니면 루트노드 R과 트리의 집합으로 구성되는데
각 트리의 루트노드는 R의 자식노드이다.
단, 트리의 집합은 공집합일 수도 있다

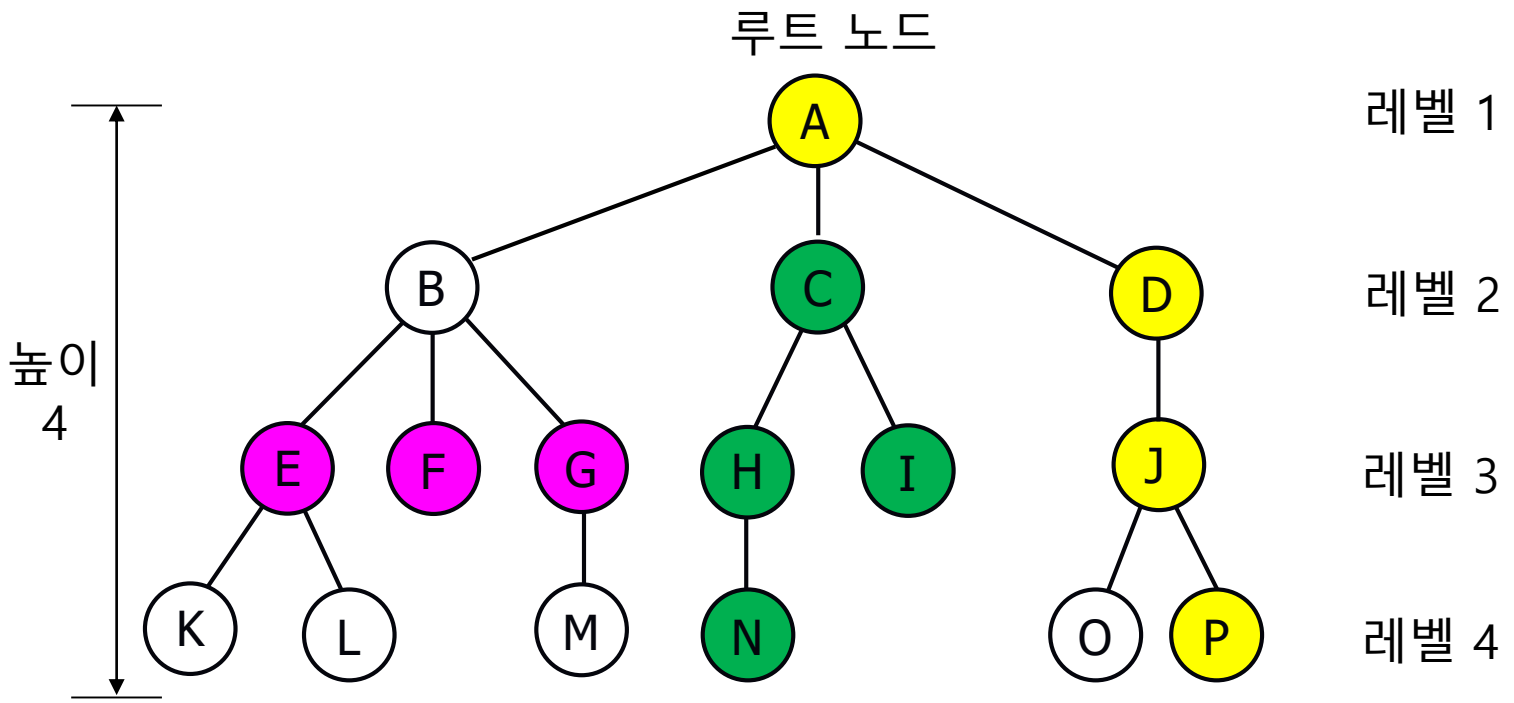
트리의 정의

- **트리 : 하나 이상의 노드(node)로 이루어진 유한집합**
  - ① 하나의 루트(root) 노드
  - ② 나머지 노드들은 n(≥0)개의 분리 집합T_1, T_2, … , T_n으로 분할
    (T_i : 루트의 subtree)

  - node : 한 정보 아이템 + 다른 노드로 뻗어진 가지
  - 노드의 차수(degree) : 노드의 서브트리 수
  - 트리의 차수 = max{노드의 차수}



용어	정의
루트(Root)노드	트리의 최상위에 있는 노드
자식(Child)노드	노드 하위에 연결된 노드
차수(Degree)	자식노드 수
부모(Parent)노드	노드의 상위에 연결된 노드
이파리(Leaf)노드	자식이 없는 노드
형제(Sibling)노드	동일한 부모를 가지는 노드
조상(Ancestor)노드	루트노드까지의 경로상에 있는 모든 노드들의 집합
후손(Descendant)노드	노드 아래로 매달린 모든 노드들의 집합
서브트리(Subtree)	노드 자신과 후손노드로 구성된 트리
레벨(Level)	루트노드는 레벨 1, 아래 층으로 내려가며 레벨이 1씩 증가. 깊이(Depth)와 동일한 용어
높이(Height)	트리의 최대 레벨
키(Key)	탐색에 사용되는 노드에 저장된 정보

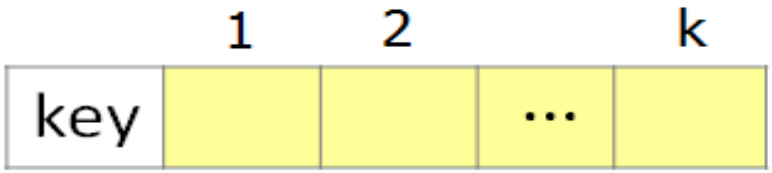


- 트리 height: 4
- A: 트리의 root node
- B, C, D: 각각 A의 child node
- A의 degree : 3
- B, C, D의 parent node: A
- C의 descendent : {H, I, N}

- K, L, F, M, N, I, O, P: leaf nodes
- E, F, G의 부모가 B로 모두 같으므로 이들은 서로 sibling node
- {B, C, D}, {H, I}, {K, L}, {O, P}도 각각 서로 sibling node
- C를 root node로 하는 subtree는 C와 C의 descendent node들로 구성된 트리
- P의 ancestor node: {J, D, A}

추가 용어

- 단말(Terminal)노드 또는 외부(External)노드 : leaf node 의 다른 말
- 내부(Internal)노드 또는 비 단말(Non-Terminal)노드
  : 이파리가 아닌 노드
- 일반적인 트리를 메모리에 저장하려면
  각 노드에 키와 자식 수만큼의 레퍼런스 저장 필요
  - 노드의 최대 차수가 k라면, k개의 레퍼런스 필드를 다음과 같이 선언해야



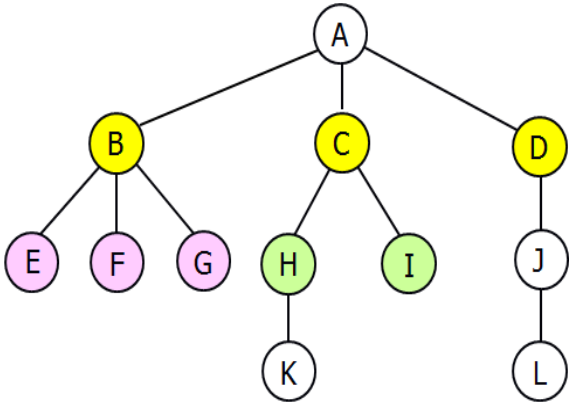
트리의 특성

- **N개의 노드가 있는 최대 차수가 k인 트리**
  - null 레퍼런스 수 = Nk - (N-1) = N(k-1) + 1
    - Nk = 총 레퍼런스의 수
    - (N-1) = 트리에서 부모-자식을 연결하는 레퍼런스 수
- **k가 클수록 메모리의 낭비가 심해지는 것은 물론 트리를 탐색하는 과정에서 null 레퍼런스를 확인해야 하므로 시간적으로도 매우 비효율적**

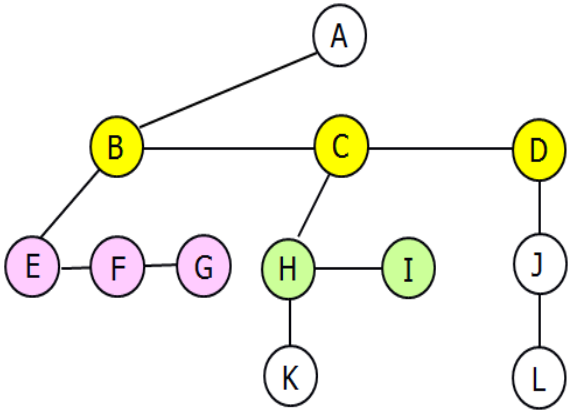
왼쪽자식-오른쪽형제(Left Child-Right Sibling) 표현

- 노드의 왼쪽 자식과 왼쪽 자식의 오른쪽 형제노드를 가리키는 2개의 레퍼런스만을 사용

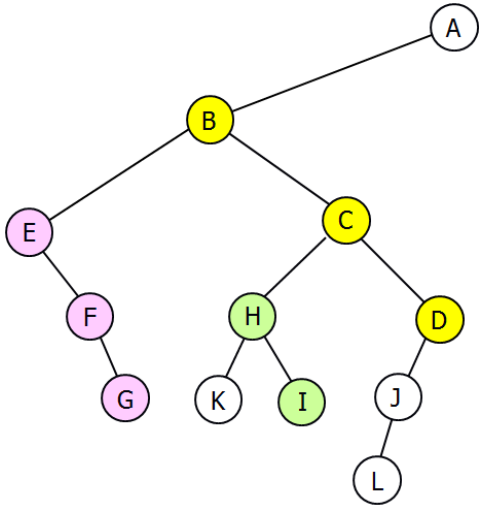
- [예제] (a)의 트리를 왼쪽자식-오른쪽형제 표현으로 변환하면, (b)의 트리를 얻으며, (c)는 (b)의 트리를 45° 시계 방향으로 회전한 것



(a)



(b)



(c)

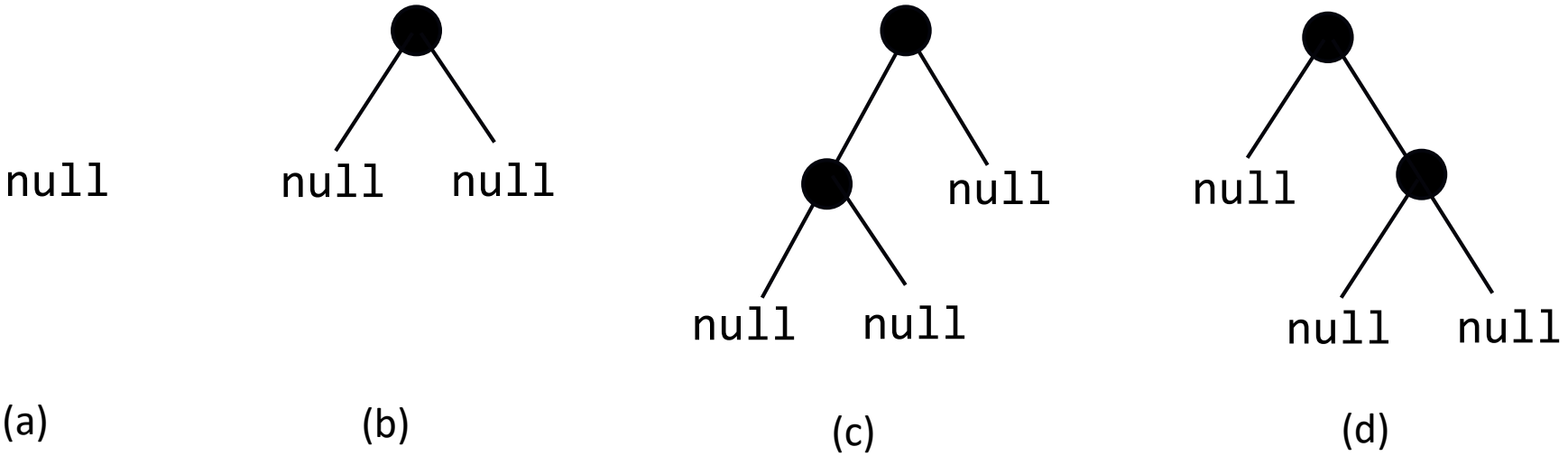
4.2 이진트리

- **이진트리(Binary Tree): 각 노드의 자식 수가 2 이하인 트리**
- **컴퓨터 분야에서 널리 활용되는 기본적인 자료구조**
  - 이진트리가 데이터의 구조적인 관계를 잘 반영하고,
    효율적인 삽입과 탐색을 가능하게 하며,
    이진트리의 서브트리를 다른 이진트리의 서브트리와 교환하는 것이 용이
- **이진트리에 대한 용어는 일반적인 트리에 대한 용어와 동일**

이진 트리의 정의

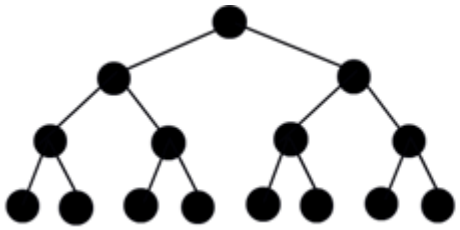
[정의] 이진트리는empty이거나, empty가 아니면,
루트노드와 2개의 이진트리인 왼쪽 서브트리와 오른쪽 서브트리로 구성된다.

- (a) empty 트리
- (b) 루트노드만 있는 이진트리
- (c) 루트노드의 오른쪽 서브트리가 없는(empty) 이진트리
- (d) 루트노드의 왼쪽 서브트리가 없는 이진트리

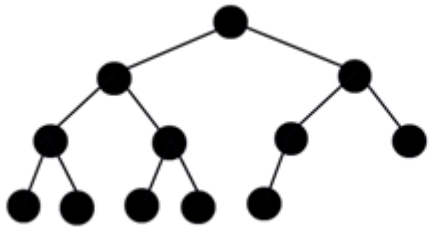


특별한 형태의 이진트리

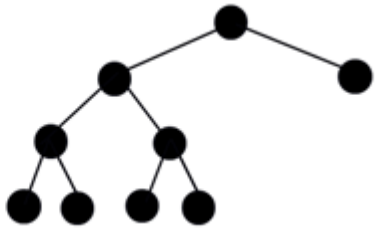
- **포화이진트리(Full Binary Tree)**
  - 각 내부노드가 2개의 자식노드를 가지는 트리
- **완전이진트리(Complete Binary Tree)**
  - 마지막 레벨을 제외한 각 레벨이 노드들로 꽉 차있고,
    마지막 레벨에는 노드들이 왼쪽부터 빠짐없이 채워진 트리
  - 포화이진트리는 완전이진트리이다.



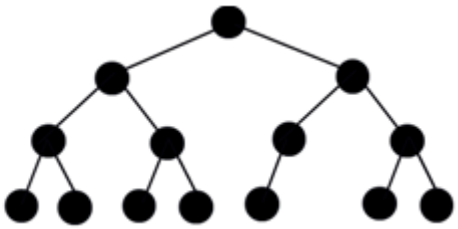
(a) 포화이진트리



(b) 완전이진트리



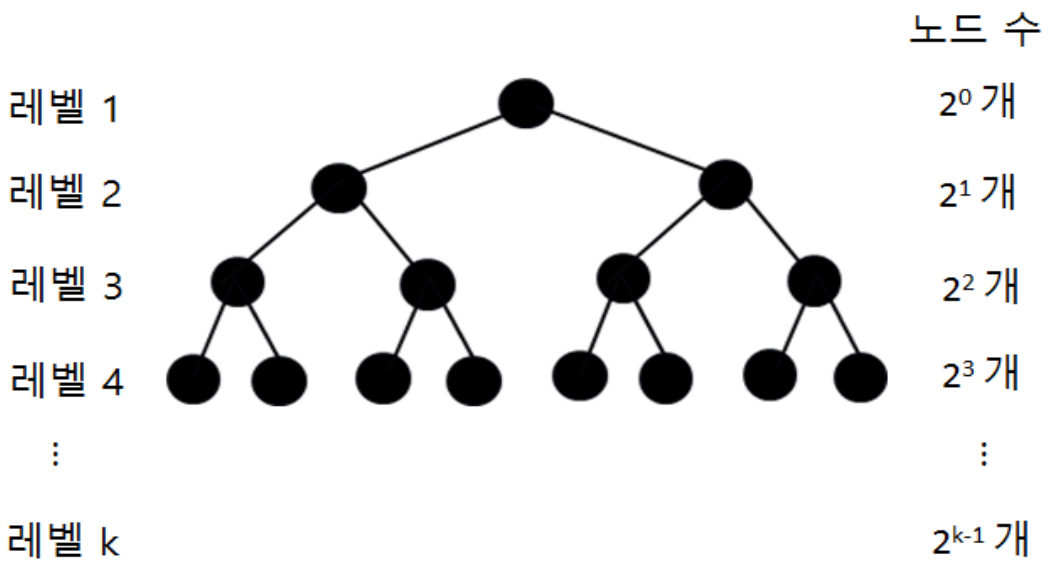
(c) 불완전한 이진트리



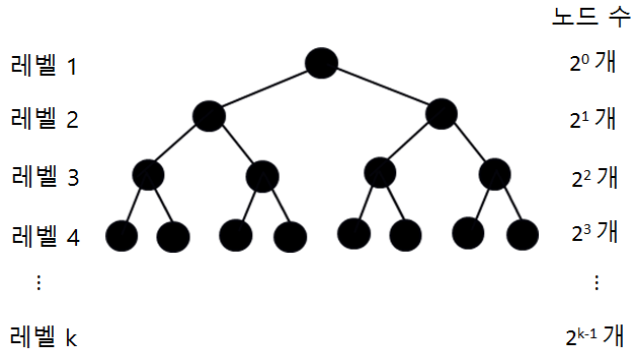
(d) 불완전한 이진트리

이진트리의 속성

- 레벨 k에 있는 최대 노드 수 = 2^{k-1} , k = 1, 2, 3, …
- 높이가 h인 포화이진트리에 있는 노드 수 = $2^h - 1$
- N개의 노드를 가진 완전이진트리의 높이 = $\lceil \log_2(N+1) \rceil$



이진 트리의 속성



- **레벨 1에 2^0 = 1개, 레벨 2에 2^1 = 2개,**
- **…, 레벨 k에 최대 2^{k-1}개의 노드**
- **즉, 한 층에 존재할 수 있는 최대 노드 수는 이전 층에 있는 최대 노드 수의 2배**
  - 이전 층에 있는 각 노드가 최대 2개의 자식노드를 가지므로

- **높이가 h인 포화이진트리에 있는 노드 수**
  - $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$
  - 노드 수가 N이면, $h = \log_2(N+1)$

이진 트리의 속성

- **N개의 노드를 가진 완전이진트리에서 N이 $2^h - 1$보다 클 수 없으므로,**
- **높이 $h = \lceil \log_2(N+1) \rceil$**
- **높이가 h인 완전이진트리에 존재 할 수 있는 최대 노드 수 N은**

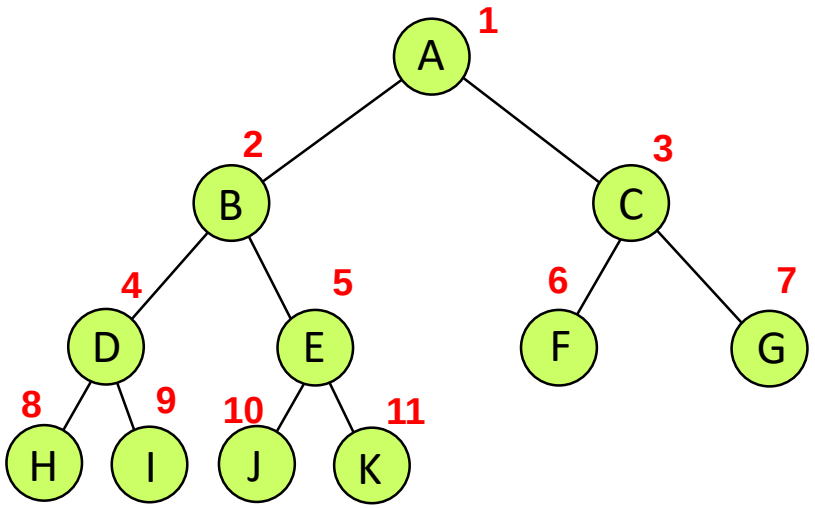
$$2^{h-1} <= N <= 2^h - 1$$

  - 노드 수가 2^{h-1}보다 작으면 높이 = (h - 1)
  - $2^h - 1$보다 크면 높이 = (h + 1)

이진 트리의 속성

- **리프 노드 수와 차수가 2인 노드 수와의 관계**
  - $n_0 = n_2 + 1$
    - n_0 : 리프 노드 수
    - n_2 : 차수가 2인 노드 수
  - 증명
    - n_1 : 차수 $_1$인 노드 수, n : 총 노드 수, B : 총 가지 수
    - $n = n_0 + n_1 + n_2$
    - 루트를 제외한 모든 노드들은 들어오는 가지가 하나씩 있으므로
      모든 가지의 수 B에 대해서 $n = B + 1$
    - 모든 가지들은 차수가 2 또는 1인 노드에서 뻗어 나오므로 $B = n_1 + 2n_2$
    - $\therefore n = B + 1 = n_1 + 2n_2 + 1$
    - $n = n_1 + 2n_2 + 1$ 와 $n = n_0 + n_1 + n_2$ 에서 $n_0 = n_2 + 1$

배열을 이용한 이진 트리의 구현



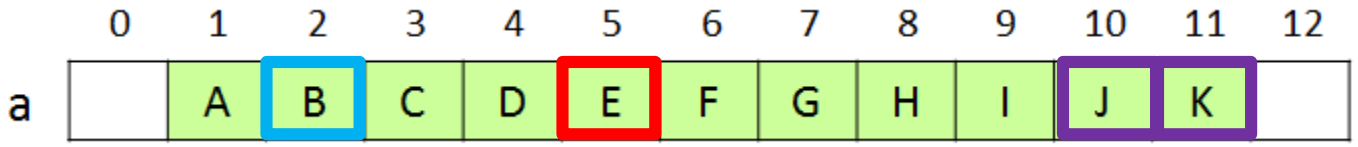
배열에 저장된 이진트리

	0	1	2	3	4	5	6	7	8	9	10	11	12
a		A	B	C	D	E	F	G	H	I	J	K	

- **배열에 저장하면 노드의 부모노드와 자식노드가 배열의 어디에 저장되어 있는지를 다음과 같은 규칙을 통해 쉽게 알 수 있다. 단, 트리에 N개의 노드가 있다고 가정**

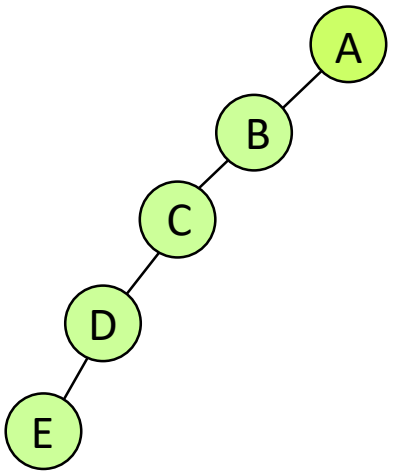
배열을 이용한 이진 트리의 구현

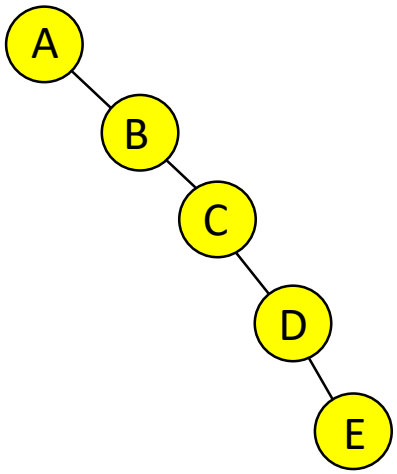
- a[i]의 부모노드는 a[i/2]에 있다. 단, i > 1이다.
- a[i]의 왼쪽 자식노드는 a[2i]에 있다. 단, 2i ≤ N이다.
- a[i]의 오른쪽 자식노드는 a[2i+1]에 있다. 단, 2i + 1 ≤ N이다.

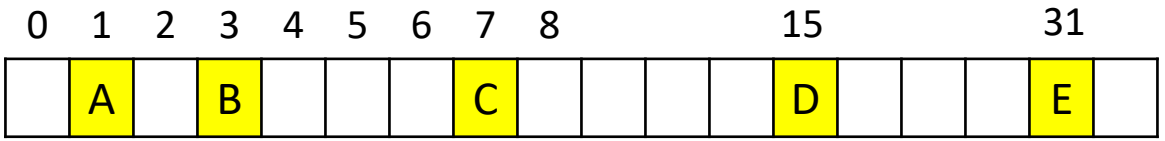


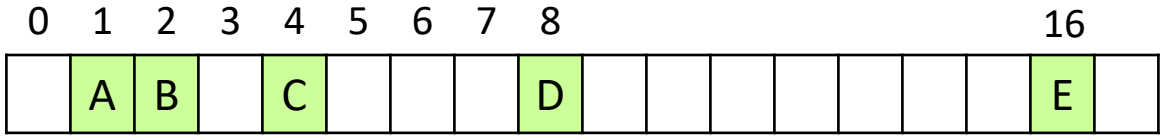
- E의 부모노드는 a[5/2] = a[2]에 있는 B
- E의 왼쪽과 오른쪽 자식은 각각 a[2x5] = a[10]과 a[2x5+1] = a[11]에 저장된 J와 K

편향 이진 트리





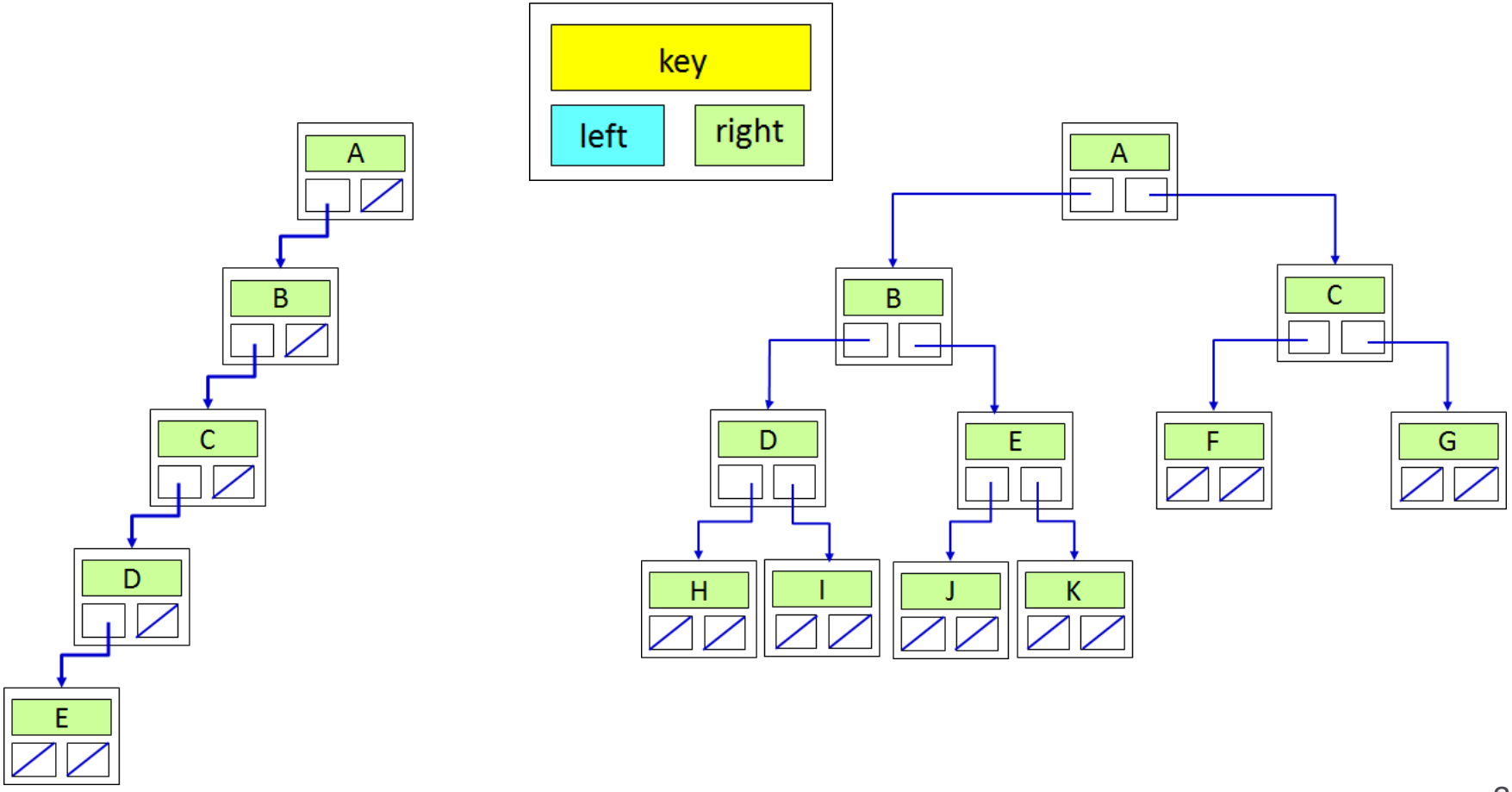




- 편향(Skewed)이진트리를 배열에 저장하는 경우, 트리의 높이가 커질 수록 메모리 낭비가 심화됨

재귀 구조를 이용한 트리의 구현

- **이진트리의 노드는 키와 2개의 레퍼런스 필드, 즉, left와 right를 가지는 이진트리의 노드로 구현**
  - 키와 관련된 정보도 노드에 저장되나 생략한다



이진 트리를 위한 Node 클래스

```
01 public class Node<Key extends Comparable<Key>> {
02     private  Key        item;
03     private  Node<Key> left;
04     private  Node<Key> right;
05     public Node( Key newItem, Node lt, Node rt ) { // 노드 생성자
06         item = newItem; left = lt; right = rt; }
07     public Key        getKey( )   {return item; }
08     public Node<Key> getLeft( )  {return left; }
09     public Node<Key> getRight( ) {return right;}
10     public void setKey(Key  newItem)   {item  = newItem;}
11     public void setLeft(Node<Key> lt)  {left  = lt;}
12     public void setRight(Node<Key> rt) {right = rt;}
13 }
```

- Line 01: Key를 generic 타입으로 사용하여 데이터를 노드에 저장하고, Comparable 인터페이스는 compareTo() 메소드를 통해 2개의 키를 비교하기 위해
- Line 05~06: Node 객체를 위한 생성자
- Line 07~12: Node 객체에 대한 get, set 메소드들

이진 트리 BinaryTree 클래스

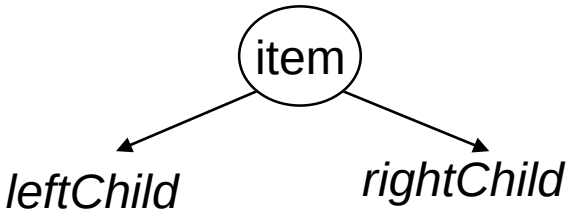
```
01 import java.util.*;
02 public class BinaryTree<Key extends Comparable<Key>>  {
03     private Node root;
04     public BinaryTree( ) { root = null; } // 트리 생성자
05     public Node getRoot( )              { return root; }
06     public void setRoot(Node newRoot) { root = newRoot; }
07     public boolean isEmpty( ) { return root == null; }
       // preorder(), inorder(), postorder(), levelorder(),
       // size(), height(), isEqual() 메소드 선언
   }
```

- BinaryTree 클래스의 생성자: Node 객체인 root만을 가지는 BinaryTree 객체를 line 04에서 생성
- Line 05: root를 리턴하는 메소드
- Line 06: 트리의 루트노드인 newRoot를 root가 가리키게 함
- Line 07: 트리가 empty인지를 체크하는 메소드
- 이 후는 이진트리를 4종류의 방식으로 순회하는 메소드들과 기타 기본 연산들을 위한 메소드들을 선언
- 각 메소드를 완성시킨 프로그램은 부록 V 참조

C++에서의 Binary Tree

- **연결 표현**
  - 노드 표현

left child	item	right child



  - 클래스 정의

```
template <class T>
class TreeNode{
private:
   T item ;
   TreeNode<T> *leftChild;
   TreeNode<T> *rightChild;
public:
    // 노드 연산들
    T getItem() ;
    TreeNode getLeftChild() ;
    ..
    void setItem(T newItem) ;
    …
};
```

```
template <class T>
class Tree {
private:
      TreeNode<T> *root;
public:
      // 트리 연산들
      Tree() { root = null ; }
      TreeNode<T> *getRoot { return root ; }
      void setRoot(TreeNode<T> *r) { root = r ; }
      bool isEmpty() { return root == null ; }
      ...
};
// T 형이 비교연산자에 대한
// operator overloading이 되어 있어야
```

4.3 이진트리의 연산

- **이진트리에서 수행되는 기본 연산들은 트리를 순회(Traversal)하며 진행**
- **이진트리의 4가지 순회하는 방식**
  - 방식은 각각 다르지만 순회는 항상 트리의 루트노드부터 시작
  - 전위순회(Preorder Traversal)
  - 중위순회(Inorder Traversal)
  - 후위순회(Postorder Traversal)
  - 레벨순회(Levelorder Traversal)

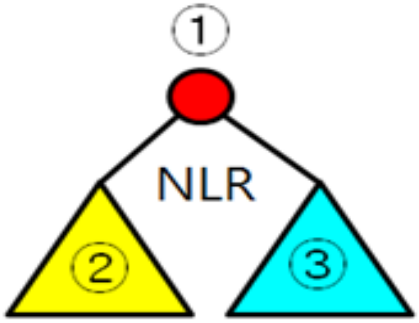
이진트리의 순회

- **트리를 순회하는 중에 노드를 방문하는 시점에 따라 구분**
  - 전위(VLR), 중위(LVR), 후위(LRV)
    - 모두 루트노드로부터 동일한 순서로 이진트리의 노드를 순회
    - 특정 노드에 도착하자마자 그 노드를 방문하는지, 일단 지나치고 나중에 방문하는 지에 따라 구분

    - 집을 노드로 보면, 노드를 방문하는 것은 문을 열고 집안에 들어가는 것
    - 사람이 노드(집)에는 도착했으나
      집을 방문하는 것을 나중으로 미루고
      왼쪽이나 오른쪽 길로 다른 집을 찾아 나설 수 있음
    - 모든 순회 방식은 루트노드로부터 순회를 시작하여
      트리의 각 노드를 반드시 1 번씩 방문해야 순회 종료

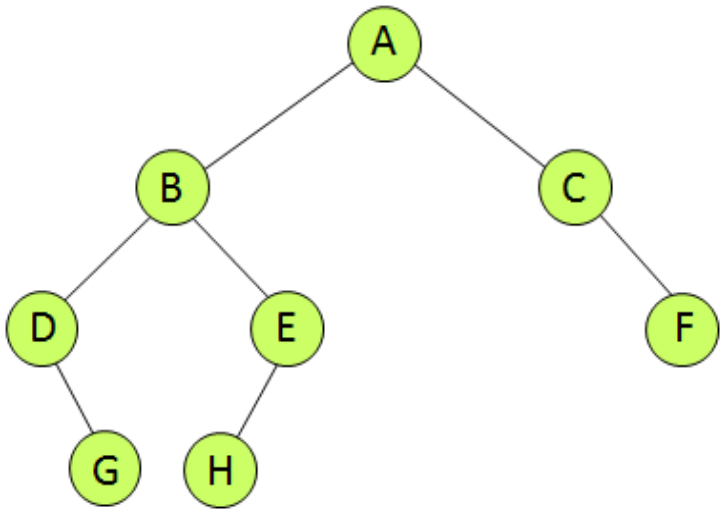
Preorder Traversal

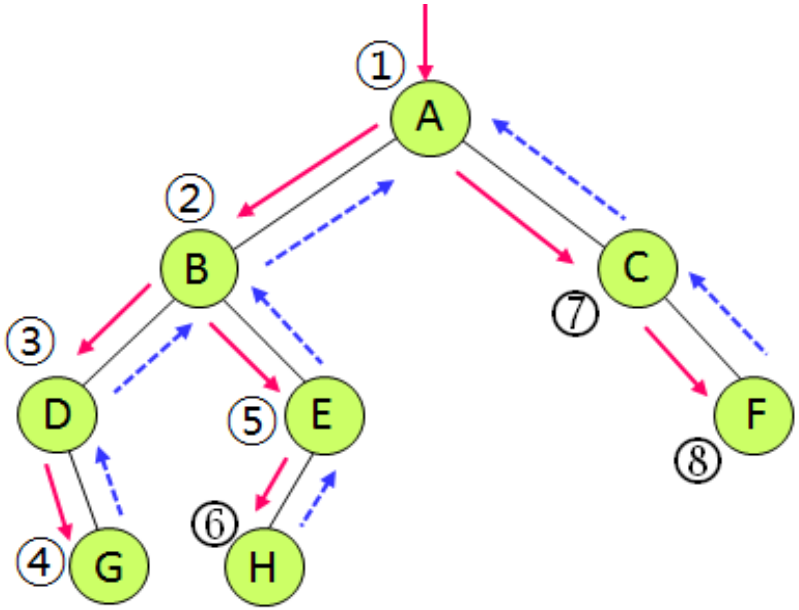
- **순회 방식**
  - 전위순회는 노드 x에 도착했을 때 x를 먼저 방문
  - 그 다음에 x의 왼쪽 자식노드로 순회를 계속
  - 왼쪽 서브트리의 모든 노드들을 방문한 후에 오른쪽 서브트리의 모든 후손 노드들을 방문
- **전위순회의 방문 규칙**



- **각 서브트리의 방문은 동일한 방식으로**
- **전위순회 순서를 NLR 또는 VLR로 표현**
  - 여기서 N은 노드(Node)를 방문한다는 뜻이고, V는 Visit(방문)을 의미
  - L은 왼쪽, R은 오른쪽 서브트리로 순회를 진행한다는 뜻

Preorder Traversal





- **실선 화살표를 따라서A, B, D, G, E, H, C, F 순으로 방문**
  - 점선 화살표는 노드의 서브트리에 있는 모든 노드들을 방문한 후에 부모노드로 복귀하는 것을 표시
  - 복귀하는 것은 프로그램에서 메소드 호출이 완료된 후에 리턴하는 것과 동일
  - 단 노드를 방문 하는 것은 노드의 key를 출력

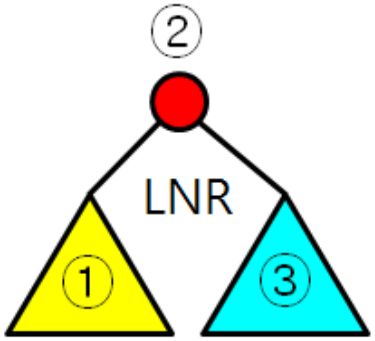
Preorder Traversal

```
01 public void preorder(Node n) { // 전위순회
02     if (n != null) {
03         System.out.print(n.getKey()+" ");  // 노드 n 방문
04         preorder(n.getLeft());  // n의 왼쪽  서브트리를 순회하기 위해
05         preorder(n.getRight()); // n의 오른쪽 서브트리를 순회하기 위해
06     }
07 }
```

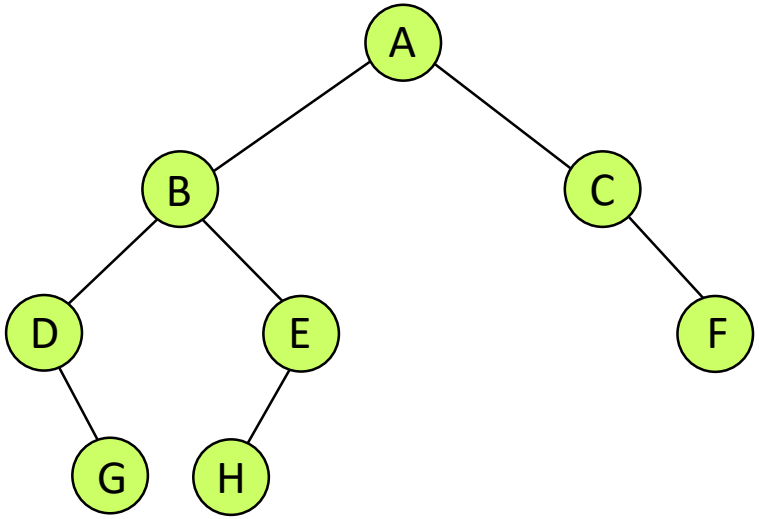
- preorder(): 트리의 루트노드를 인자로 전달하여 호출
- Line 02: 노드 n이 null 인지를 검사하고 null이면 이전 호출된 곳으로 돌아가고, null이 아니면 line 03 에서 노드 n을 방문
- Line 04: 노드 n의 왼쪽 자식노드로 재귀호출하여 왼쪽 서브트리의 모든 노드들을 방문한 후
- Line 05: 노드 n의 오른쪽 자식노드로 재귀호출하고 오른쪽 서브트리의 모든 노드들을 방문

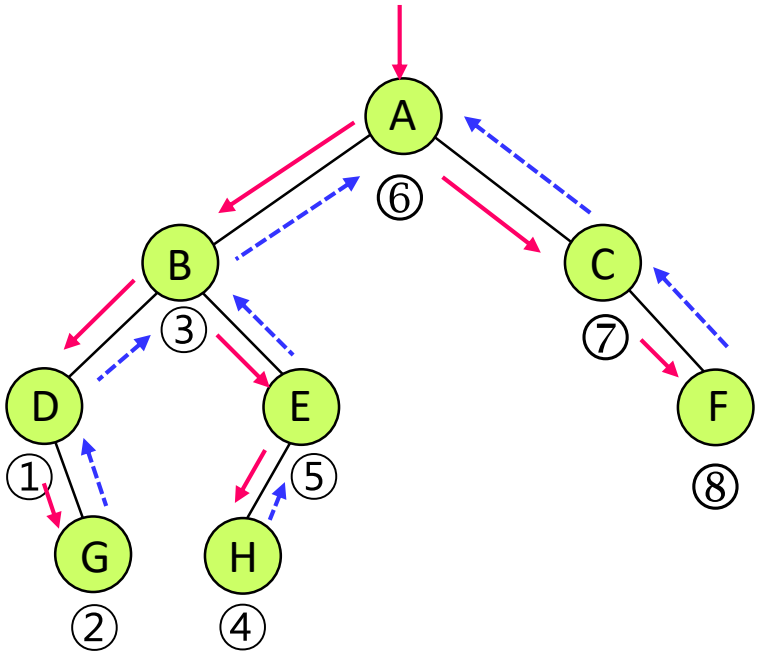
Inorder Traversal

- **중위순회는 노드 x에 도착하면 x의 방문을 보류하고 x의 왼쪽 서브트리로 순회를 먼저 진행**
  - 왼쪽 서브트리의 모든 노드들을 방문한 후에 x를 방문
- **x를 방문한 후에는x의 오른쪽 서브트리를 같은 방식으로 방문**
- **중위순회 순서를 LNR 또는 LVR로 표현**



Inorder Traversal





- 중위순회: D, G, B, H, E, A, C, F 순으로 방문

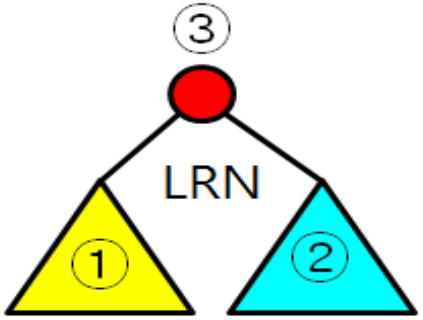
Inorder Traversal

```
01 public void inorder(Node n){  // 중위순회
02     if (n != null) {
03         inorder(n.getLeft());    // n의 왼쪽 서브트리를 순회하기 위해
04         System.out.print(n.getKey()+" ");  // 노드 n 방문
05         inorder(n.getRight());  // n의 오른쪽 서브트리를 순회하기 위해
06     }
07 }
```

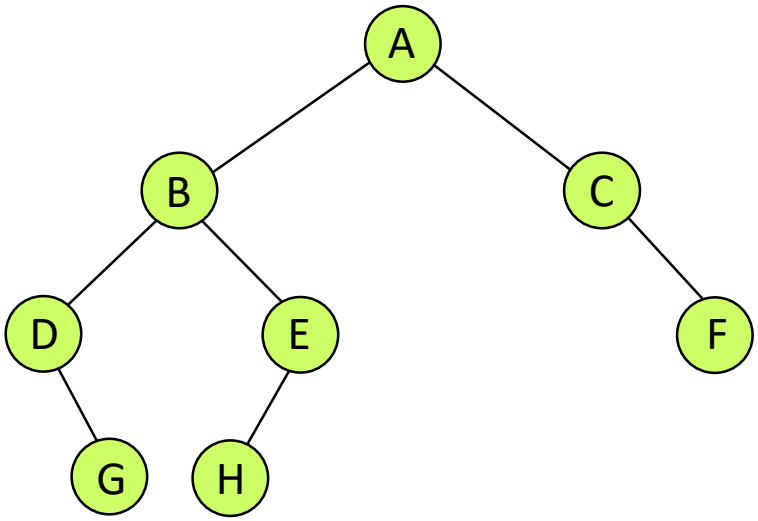
- inorder() 메소드: 트리의 루트노드를 인자로 전달하여 호출
- Line 02: 노드 n이 null 인지를 검사하고, null이면 이전 호출된 곳으로 돌아가고, null이 아니면
- Line 03: 노드 n의 왼쪽 자식노드로 재귀호출하여 왼쪽 서브트리의 모든 노드들을 방문한 후에
- Line 04: 노드 n을 방문
- Line 05: 노드 n의 오른쪽 자식노드로 재귀호출하고 오른쪽 서브트리의 모든 노드들을 방문

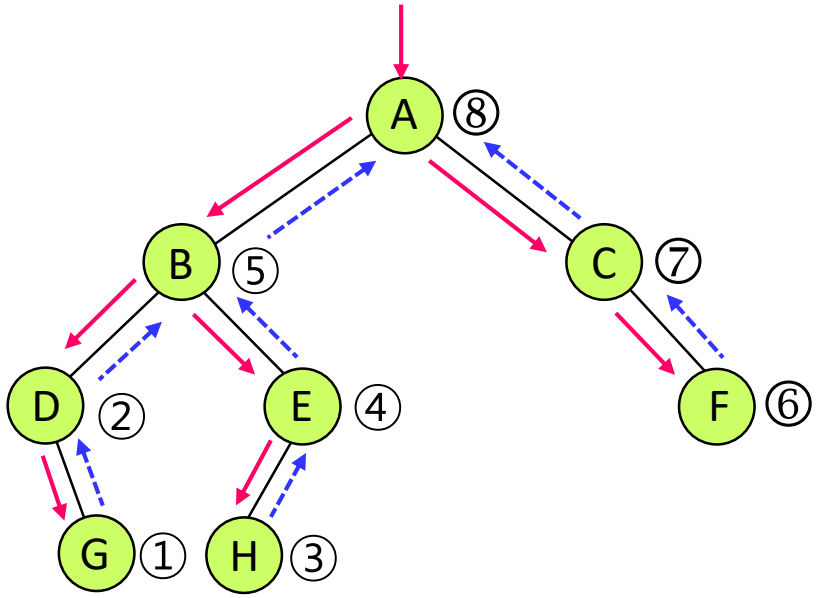
Postorder Traversal

- 후위순회는 노드 x에 도착하면 x의 방문을 보류하고
  x의 왼쪽 서브트리로 순회를 먼저 진행
- x의 왼쪽 서브트리를 방문한 후에는 x의 오른쪽 서브트리를 같은 방식으로 방문
- 마지막에 x를 방문한다.
- 후위순회 순서를 LRN 또는 LRV로 표현



Postorder Traversal





후위순회: G, D, H, E, B, F, C, A 순으로 방문

Postorder Traversal

```
01  public void postorder(Node n) { // 후위순회
02      if (n != null) {
03          postorder(n.getLeft());  // n의 왼쪽  서브트리를 순회하기 위해
04          postorder(n.getRight()); // n의 오른쪽 서브트리를 순회하기 위해
05          System.out.print(n.getKey()+" "); // 노드 n 방문
06      }
07  }
```

- **postorder() 메소드: 트리의 루트노드를 인자로 전달하여 호출**
  - Line 02: 노드 n이 null 인지를 검사하고, null이면 이전 호출된 곳으로 돌아가고, null이 아니면
  - Line 03: 노드 n의 왼쪽 자식노드로 재귀호출하여 왼쪽 서브트리의 모든 노드들을 방문한 후에
  - Line 04: 노드 n의 오른쪽 자식노드로 재귀호출하고 오른쪽 서브트리의 모든 노드들을 방문
  - 끝으로line 05에서 노드 n을 방문

C++에서의 tree post order travesal

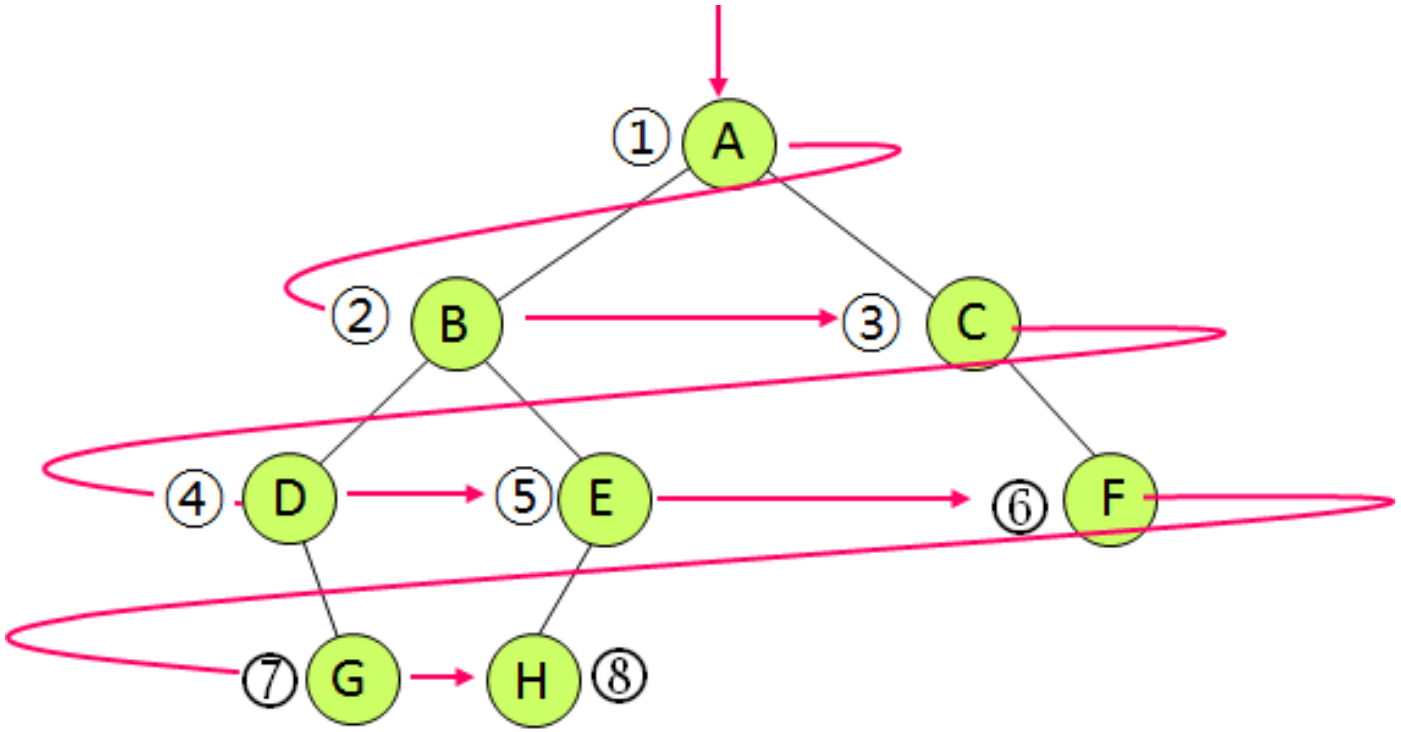
```
 1 template <class T>
 2 void Tree<T>::postorder()
 3 {
 4    postorder(root);
 5 }

 6 template <class T>
 7 void Tree<T>::postorder(TreeNode<T> *currentNode)
 8 {
 9     if (currentNode) {
10         postorder(currentNode->leftChild);
11         postorder(currentNode->rightChild);
12         visit(currentNode); // 현재 값을 출력하는 함수
13     }
14 }
```

C++에서의 non-recursive inorder traversal

```
==============================================
 1 template <class T>
 2 void Tree<T>::nonRrecInorder()
 3 {//  stack을 직접 이용하는 중위 순회
 4      Stack<TreeNode<T>*>  s;  // 스택 선언
 5      TreeNode<T> *currentNode = root;
 6      while(1) {
 7           while (currentNode) { // 제일 왼쪽 left로 이동
 8                s.push(currentNode);  // add to stack
 9                currentNode = currentNode->leftChild;
10           }
11           if (s.isEmpty()) return;
12           currentNode = s.top();   // 스택에 제일 마지막에 들어간 노드
13           s.pop();                          // 꺼내서 처리
14           visit(currentNode);
15           currentNode = currentNode->rightChild;
16      } // 스택이 empty가 될 때까지 반복
17 }
==============================================
```

Level order Traversal

- 레벨순회는 루트노드가 있는 최상위 레벨부터 시작하여 각 레벨마다 좌에서 우로 노드들을 방문



Level order Traversal

```
01   public void levelorder(Node root) { // 레벨순회
02       Queue<Node> q = new LinkedList<Node>(); // 큐 자료구조 이용
03       Node t;
04       q.add(root);  // 루트 노드 큐에 삽입
05       while (!q.isEmpty()) {
06           t = q.remove();    //큐에서 가장 앞에 있는 노드 제거
07           System.out.print(t.getKey()+" "); // 제거된 노드 출력(방문)
08           if (t.getLeft() != null)       // 제거된 왼쪽 자식이 null이 아니면
09               q.add(t.getLeft());        // 큐에 왼쪽 자식 삽입
10           if (t.getRight() != null)      // 제거된 오른쪽 자식이 null이 아니면
11               q.add(t.getRight());       // 큐에 오른쪽 자식 삽입
12       }
13   }
```

- **levelorder() 메소드: 큐 자료구조를 활용**
  - Line 02: 자바 라이브러리의 LinkedList를 사용해 구현한 Queue 사용
    - Line 03: q에서 삭제된 노드를 참조하기 위해 Node 타입의 지역변수를 선언
  - Line 04: 트리의 루트노드인 root를 q에 추가한 후
  - Line 05의 while-루프를 수행
    - 루프 내에서는 line 06에서 q의 가장 앞에 있는 노드를 삭제하고 t가 이 노드를 참조
    - Line 07: q에서 삭제된 노드를 방문하고,
    - Line 08~11: t의 왼쪽 자식과 오른쪽 자식을 q에 차례로 삽입
    - 자식이 null인 경우, 삽입 과정을 건너뛴다.

C++에서의 level order traversal

```
template <class T>
void Tree<T>::LevelOrder()
{//이진 트리의 레벨 순서 순회
    Queue<TreeNode<T>*> q;
    TreeNode<T> * currentNode = root;

    while(currentNode){
        visit(currentNode);
        if(currentNode->leftChild)
              q.push(currentNode->leftChild);
        if(currentNode->rightChild)
             q.push(currentNode->rightChild);
        if(q.IsEmpty())
             return;
        currentNode = q.front();
        q.pop();
    }
}
```

기타 이진트리 연산

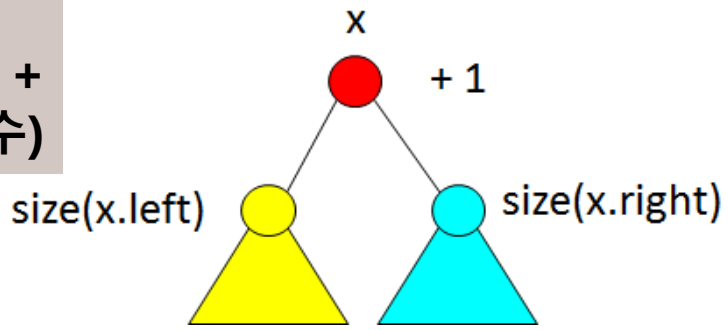
- size(): 트리의 노드 수 계산
- height(): 트리의 높이 계산
- isEqual(): 2개의 이진트리에 대한 동일성 검사
- size()와 height()는 후위순회에 기반하고,
  isEqual()은 전위순회에 기반

트리의 노드 수

- 트리의 노드 수를 계산하는 것은 트리의 아래에서 위로
  각 자식의 후손노드 수를 합하며 올라가는 과정을 통해 수행되며,
  최종적으로 루트노드에서 총 합을 구함
- 트리의 높이도 아래에서 위로 두 자식을 각각 루트노드로 하는
  서브트리의 높이를 비교하여 보다 큰 높이에 1을 더하는 것으로
  자신의 높이를 계산하며,
  최종적으로 루트노드의 높이가 트리의 높이가 됨
- 2개의 이진트리를 비교하는 것은 다른 부분을 발견하는
  즉시 비교 연산을 멈추기 위해 전위순회 방법을 사용

트리의 노드 수 계산

[핵심 아이디어]
트리의 노드 수 = 1 +
(루트노드의 왼쪽 서브트리에 있는 노드 수) +
(루트노드의 오른쪽 서브트리에 있는 노드 수)

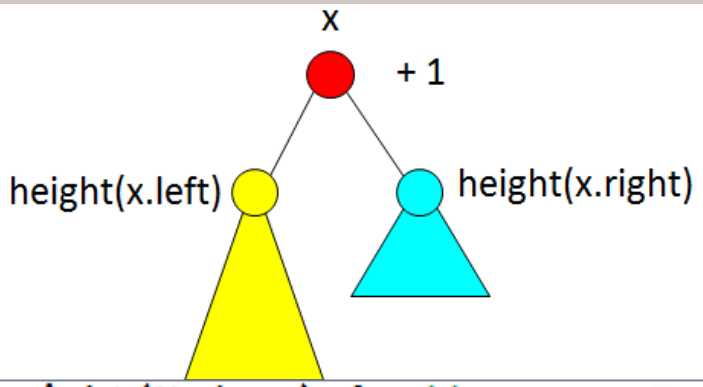


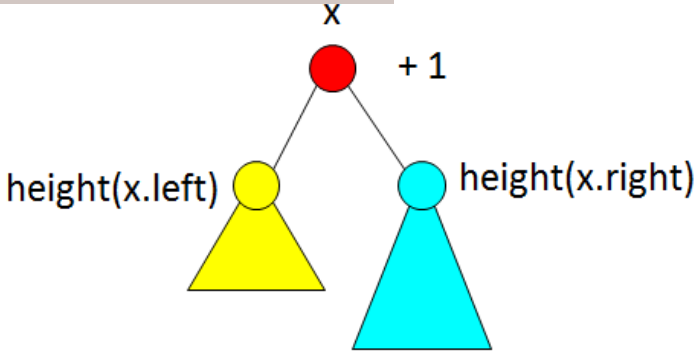

```
01  public int size(Node n) {  // n를 루트로하는 (서브)트리에 있는 노드 수
02      if (n == null)
03          return 0;    // null이면 0 리턴
04      else
05          return (1 + size( n.getLeft() ) + size( n.getRight() ));
06  }
```

- **size() 메소드: 루트노드를 인자로 전달하여 호출**
  - Line 02: 노드가 null이면 line 03에서 0을 리턴하고,
  - null이 아니면 line 05에서 왼쪽 자식노드를 루트노드로 하는 서브트리의 노드 수와 오른쪽 자식노드를 루트노드로 하는 서브트리의 노드 수를 더한 결과에 1을 더한 값을 리턴

트리의 높이

[핵심 아이디어]
트리의 높이 = 1 +
max (루트의 왼쪽 서브트리의 높이, 루트의 오른쪽 서브트리의 높이)






```
01  public int height(Node n) {  // n를 루트로하는 (서브)트리의 높이
02      if (n == null)
03          return 0;     // null이면 0 리턴
04      else
05          return (1 + Math.max(height(n.getLeft()), height(n.getRight())));
06  }
```

- **height() 메소드: 루트노드를 인자로 전달하여 호출**
  - Line 02: 노드가 null이면, line 03에서 0을 리턴하고,
  - null이 아니면 line 05에서 왼쪽 자식노드를 루트노드로 하는 서브트리 높이와 오른쪽 자식노드를 루트노드로 하는 서브트리의 높이 중에서 보다 큰 높이에 1을 더한 값을 리턴

이진트리 비교

[핵심 아이디어]

전위순회 과정에서 다른 점이 발견되는 순간 false를 리턴

```
01   public static boolean isEqual(Node n, Node m){  // 두 트리의 동일성 검사
02       if(n==null || m==null)  // 둘중에 하나라도 null이면
03           return n == m;      // 둘다 null이면 true, 아니면 false
04
05       if (n.getKey().compareTo(m.getKey()) != 0) // 둘다 null이 아니면 item 비교
06           return false;
07
08       return( isEqual(n.getLeft(), m.getLeft()) &&  // item이 같으면 왼쪽 자식 재귀호출
09               isEqual(n.getRight(), m.getRight())); // 오른쪽 자식 재귀호출
10   }
```

- **isEqual() 메소드: 비교하려는 두 트리의 루트노드들을 인자로 전달**
  - Line 02: 노드 n과 m 둘 중에 하나가 null인 경우
    - 만일 둘 다 null이면 true를 리턴하고
    - 한 쪽만 null이면 트리가 다른 것이므로 false를 리턴
  - Line 05 (만일 둘 다 null이 아니면): 두 노드의 키를 비교하여 다르면 (1 또는 -1인 경우) false를 리턴
    - 0이면 같은 key값을 갖는 경우이므로 line 08~09에서 각 트리의 왼쪽 자식노드와 오른쪽 자식노드를 인자로 하여 isEqual() 메소드를 재귀호출

수행시간

- 앞서 설명된 각 연산은 트리의 각 노드를 한 번씩만 방문하므로 O(N) 시간이 소요

스레드 이진트리

- **이진트리 기본 연산들은 레벨순회를 제외하고 모두 스택 자료구조 사용**
  - 메소드의 재귀호출은 시스템 스택을 사용하므로 스택 자료구조를 사용한 것으로 간주
- **스택에 사용되는 메모리 공간의 크기는 트리의 높이에 비례**
- **스택 없이 이진트리의 연산을 구현하는 2 가지 방법**
  - [1] Node 객체에 부모노드를 가리키는 레퍼런스 필드를 추가로 선언하여 순회에 사용하는 방법
  - [2] 노드의 null 레퍼런스들을 활용하는 것 (스레드 이진트리 (Threaded Binary Tree)
    - null 레퍼런스 공간에 다음에 방문할 노드의 레퍼런스를 저장
    - 스레드는 운영체제에서 스케줄러가 운영하는 독립적인 수행 단위인 스레드와는 전혀 관계 없는 단어

스레드 이진트리

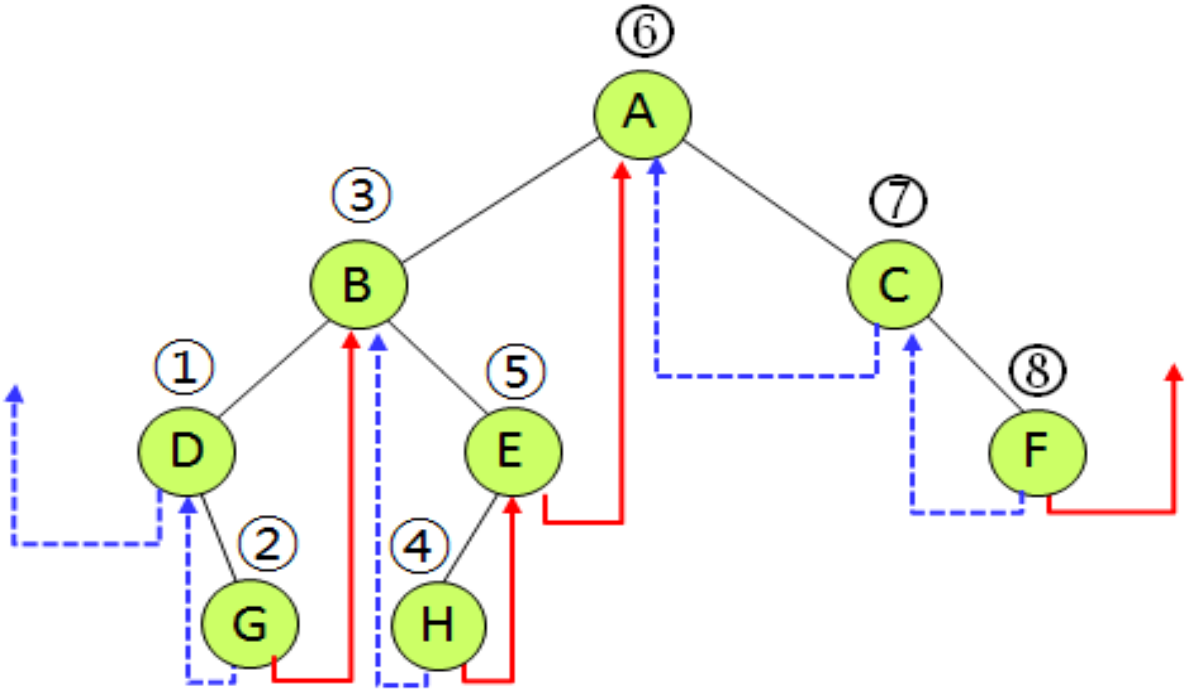
- **N개의 노드가 있는 이진트리의 null 레퍼런스 필드 수 = (N+1)**
  - 왜냐하면 각 노드마다 2개의 레퍼런스 필드(left와 right)가 있으므로 총 2N개의 레퍼런스 필드가 존재하고,
    이 중에서 부모 자식을 연결하는 레퍼런스는 N-1개이기 때문
  - 부모 자식을 연결하는 레퍼런스가 N-1개인 이유는
    루트노드를 제외한 각 노드가 1개의 부모노드를 갖기 때문
- **스레드 이진트리: N+1개의 null 레퍼런스를 활용하여,
  이전에 방문한 노드와 다음에 방문할 노드를 가리키도록 만들어
  순회 연산이 스택 없이도 수행될 수 있도록 만든 트리**

스레드 이진트리

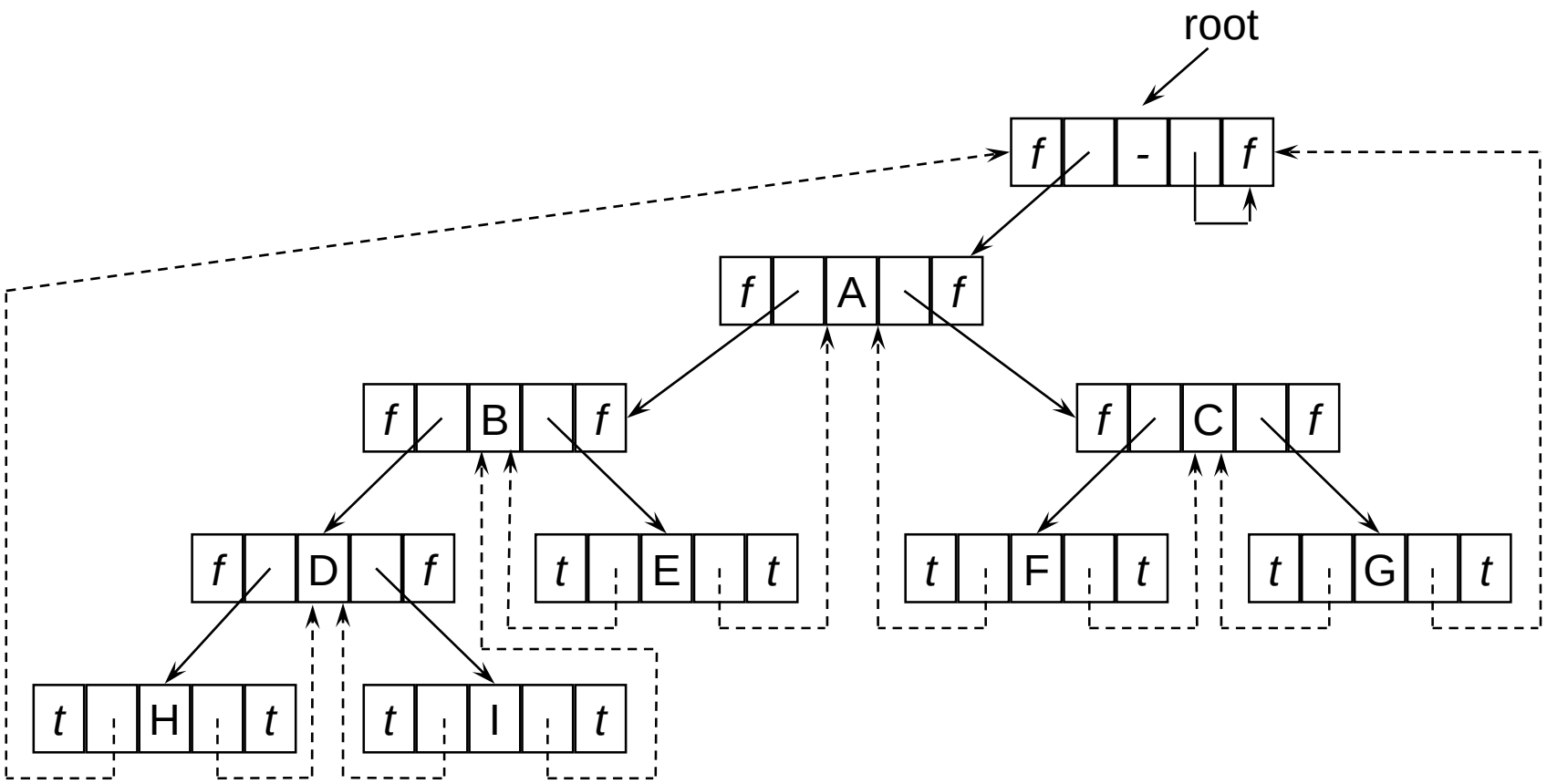
- 스레드 이진트리는 대부분의 경우 중위순회에 기반하여 구현
  - 전위순회이나 후위순회에 기반하여 스레드 트리를 구현할 수도 있음
- 장점 : 스택을 사용하는 순회보다 빠르고 메모리 공간도 적게 차지
- 단점: 데이터의 삽입과 삭제가 잦은 경우 그 구현이 비교적 복잡한 편이므로 좋은 성능을 보여주지 못함
- Node 객체에 2개의 boolean 필드를 사용하여
  레퍼런스가 스레드(다음 방문할 노드를 가리키는)로 사용되는 것인지
  아니면 left나 right가 트리의 부모 자식 사이의 레퍼런스인지를
  각각 true 와 false로 표시해주어야 함

중위순회 스레드 이진트리

- 점선 화살표는 직전 방문 노드를 가리키는 스레드이고, 실선 화살표는 다음에 방문 노드를 가리키는 스레드



스레드 이진 트리의 메모리 표현



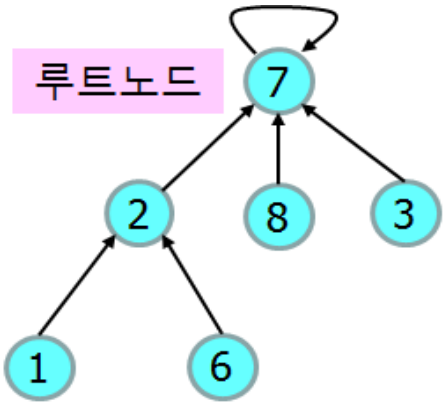
f = **false**; *t* = **true**

4.4 상호배타적 집합을 위한 트리 연산

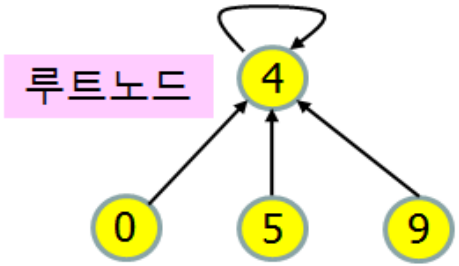
- **집합에 관련된 연산:**
  - 합집합(union) 연산
  - 주어진 원소에 대해 어느 집합에 속해 있는지를 계산하는 find 연산
- **상호배타적 집합(Disjoint Set): 어느 두 집합도 중복된 원소를 갖지 않는 집합들**
- **상호배타적 집합의 union과 find연산은 9.4절의 Kruskal의 최소신장트리 알고리즘을 구현하는데 활용**
- **상호배타적 집합들을 메모리에 저장하기 위해 1차원 배열 사용**
- **원소를 0, 1, 2, …, N-1로 놓으면 이를 배열의 인덱스로 활용**
- **집합에 속한 원소들 사이에 특정한 순서가 없고, 또 중복된 원소도 없음**

상호배타적 집합을 위한 트리 표현

- 2개의 상호배타적 집합을 일반적인 트리로 표현하여 1차원 배열에 저장
- 각 집합은 루트가 대표하고, 루트의 배열 원소에는 루트 자신이 저장되며, 루트가 아닌 노드의 원소에는 부모노드를 저장



집합 {7, 2, 8, 3, 1, 6}



집합 {4, 0, 5, 8}

	0	1	2	3	4	5	6	7	8	9
a	4	2	7	7	4	4	2	7	7	4

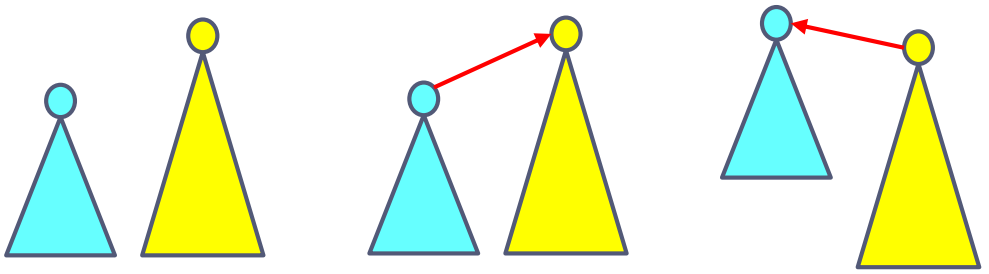
상호배타적 집합을 위한 트리 연산

- **상호배타적 집합에 대한 연산**
- **union 연산: 2개의 집합을 하나의 집합으로 만드는 연산**
- **find 연산: 인자로 주어지는 x가 속한 집합의 대표 노드,**
  즉, 루트를 찾는 연산
  - find(6)은 a[6] = 2를 통해 6의 부모노드인 2를 찾고, a[2] = 7으로 2의 부모 노드를 찾으며, 마지막으로 a[7] = 7이기 때문에 7를 리턴
  - 즉, “6은 7이 대표 노드인 집합에 속해 있다”는 것을 리턴
  - find(3) = 7이므로, 6과 3은 동일한 집합에 속함
  - find(9) = 4이므로, 6과 9는 서로 다른 집합에 속함

Union 연산

[핵심 아이디어]
먼저 union 연산은 rank에 기반하여 (union-by-rank) rank가 높은 루트가 union 후에도 승자(합쳐진 트리의 루트)가 되도록 한다.

- 트리의 노드 수에 기반(즉, 노드 수가 많은 트리의 루트가 승자가 되도록)하여 union 을 수행해도 rank 기반 union과 동등한 성능을 보임
- 루트의 rank는 트리의 높이와 일단은 같다고 생각해도 됨
- rank가 높은 루트를 승자로 만드는 이유는 합쳐진 트리가 더 커지지 않게
- 만일 두 트리의 높이가 같은 경우에는 둘 중 하나의 루트가 승자가 되고 합쳐진 트리의 높이는 1 증가
  - (c) 비효율적인 union 연산: 합쳐진 트리의 높이가 1 증가하므로 비효율적



(a) union 수행 전 (b) union 수행 후 (c) 비효율적인 union

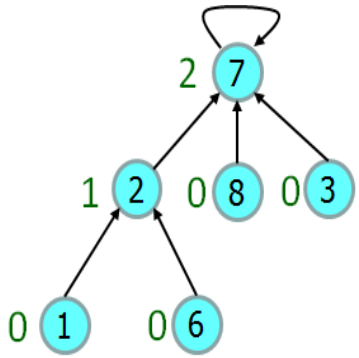
rank를 기반한 union 연산

Union 연산

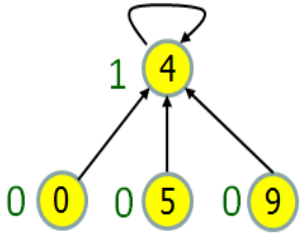
- **rank에 기반한 union연산의 목적은 두 트리가 하나로 합쳐진 후에 트리의 높이가 커지는 것을 방지하기 위함**
  - find 연산을 수행할 때 루트노드까지 올라가야 하므로 트리의 높이가 낮을 수록 find의 수행시간을 줄일 수 있기 때문
  - 단, 두 트리의 루트노드들의 rank가 같으면 어쩔 수 없이 하나의 루트노드가 승자가 되고 승자의 rank도 1 증가시켜야 함

Union 연산

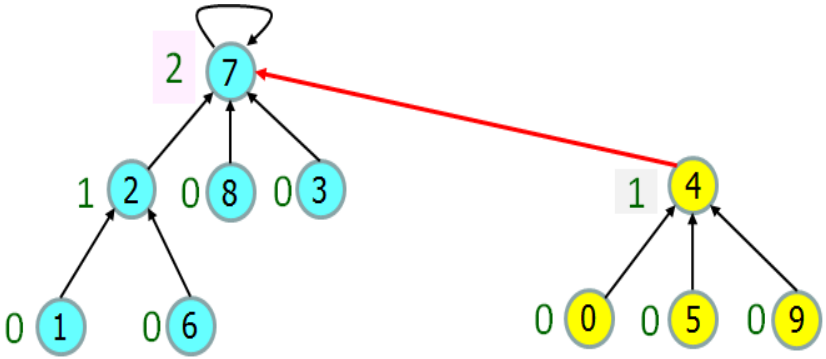
- **[예제] (a) union(7,4) 수행 (b) 수행 결과**
  - a[4] =7로 갱신되었고 트리도 하나로 합쳐짐
  - 각 노드 옆의 숫자는 노드의 rank로서 두 루트의 rank가 다르므로 union 수행 후에도 승자인 7의 rank 값도 변하지 않음



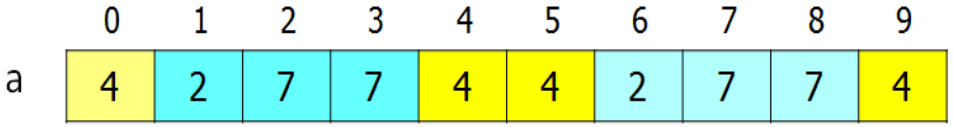
집합 {7, 2, 8, 3, 1, 6}



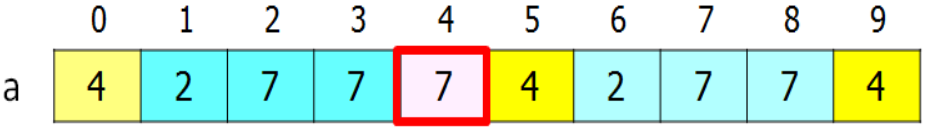
집합 {4, 0, 5, 8}



집합 {7, 2, 8, 3, 1, 6, 4, 0, 5, 8}



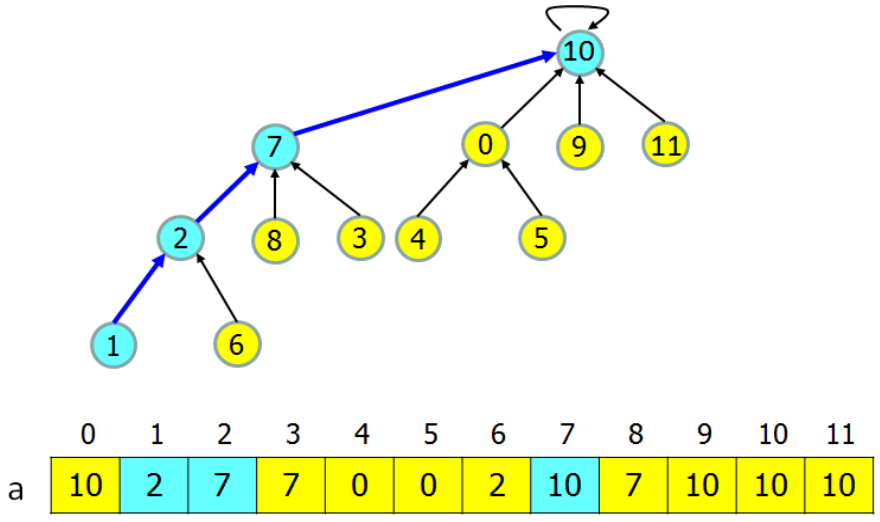
(a) union 수행 전



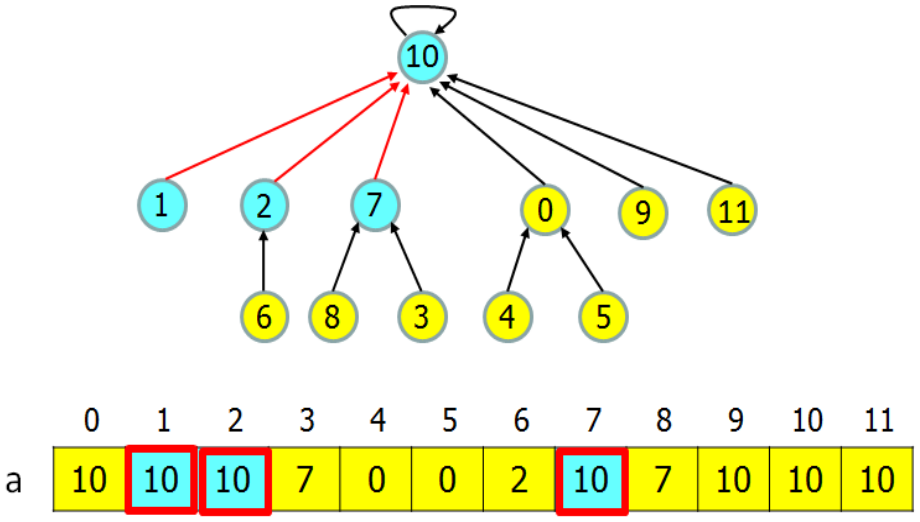
(b) union 수행 후

Find 연산

[핵심 아이디어]
find 연산을 수행하면서 루트노드까지 올라가는 경로 상의 각 노드의 부모 노드를 루트로 갱신한다. 이를 경로압축(Path Compression)이라고 한다.



(a) find (1) 연산 수행 전



(b) find 연산 수행 후

Find 연산

- **(a) find(1)을 수행하면 루트인 10까지 올라가는 경로 상의 모든 노드 1, 2, 7의 부모노드를 10으로 갱신하여 (b)와 같은 트리를 만든다.**
- **경로압축은 당장 find(1) 연산의 수행시간을 줄이지 않으나, 추후의 find(1)과 find(2)연산의 수행시간을 단축함**
- **경로압축으로 인해 루트의 rank는 트리의 높이와 달라질 수 있음**
  - 그림(a)의 트리 높이는 4인데, 경로압축 후에 트리의 높이는 3이다. 하지만 경로압축으로 인해 루트나 그 밖의 노드들의 rank 값은 변하지 않음
  - rank 를 사용하여 union 연산을 수행하는 이유는 집합들 전체에 대한 총괄적인 상각분석을 위함

상호배타적 집합을 위한 Node 클래스

```
01 public class Node {
02   int parent;
03   int rank;
04   public Node(int newParent, int newRank){
05       parent = newParent;
06       rank   = newRank;
07   }
08   public int getParent() {return parent;}
09   public int getRank()   {return rank;}
10   public void setParent(int newParent) {parent = newParent;}
11   public void setRank(int newRank)     {rank   = newRank;}
12 }
```

- **Node 객체는 int 타입의 parent와 rank 를 가짐**
  - parent는 노드의 부모노드의 레퍼런스를 저장
  - 초기에는 자기 자신을 부모노드로 초기화
  - rank는 0으로 초기화
- **Line 04~07: Node 객체의 생성자**
- **Line 08~11: Node 객체에 대한 get, set 메소드들**

상호배타적 집합을 위한 UnionFind 클래스

```
01 public class UnionFind {
02     protected  Node[] a;
03     public UnionFind(Node[] iarray) {  // 생성자
04         a = iarray;
05     }
06     //i가 속한 집합의 루트 노드를 재귀적으로 찾고 최종적으로 경로상의 각 원소의 부모를 루트 노드로 만든다.
07     protected int find(int i) { // 경로 압축
08         if ( i != a[i].getParent())
09             a[i].setParent(find(a[i].getParent()));  //리턴하며 경로상의 각 노드의 부모가 루트가되도록 만든다.
10         return a[i].getParent();
11     }
12     public void union(int i, int j) {  // Union 연산
13         int iroot = find(i);
14         int jroot = find(j);
15         if (iroot == jroot) return;  // 루트 노드가 동일하면 더이상의 수행없이 그대로 리턴
16         // rank가 높은 루트 노드가 승자가 된다.
17         if (a[iroot].getRank() > a[jroot].getRank())
18             a[jroot].setParent(iroot);                // iroot가 승자
19         else if (a[iroot].getRank() < a[jroot].getRank())
20             a[iroot].setParent(jroot);                // jroot가 승자
21         else {
22             a[jroot].setParent(iroot);  // 둘중에 하나 임의로 승자
23             int t = a[iroot].getRank() + 1;
24             a[iroot].setRank(t);         // iroot의 rank 1 증가
25         }
26     }
27 }
```

상호배타적 집합을 위한 UnionFind 클래스

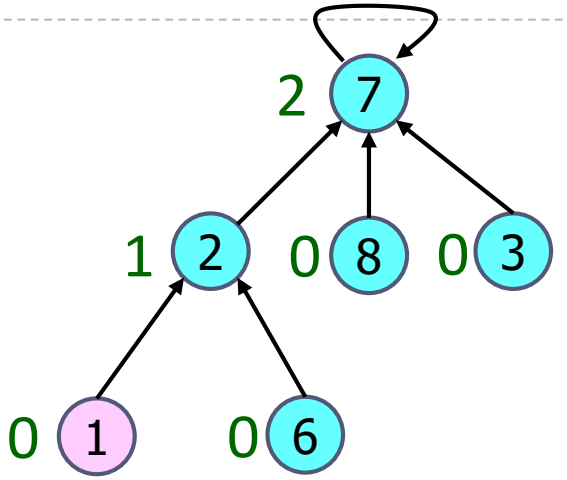
- Line 03 ~ 05: UnionFind 객체의 생성자
- 객체는 Node 객체를 원소로 하는 1차원 배열을 가짐
- Line 07 ~ 11: find() 메소드로 i가 속한 트리의 루트를 리턴하는 동시에 노드 i에서 루트까지의 경로 상의 모든 노드들에 대한 부모노드를 루트로 갱신하는 경로압축 수행
- 경로압축은 line 09에서 수행: find(a[i].getParent())의 값이 계속 루트노드로 리턴되면서 경로 상의 각 노드 i의 parent가 동일한 루트로 갱신
- Line 12 ~ 26: union() 메소드로 i가 속한 트리와 j가 속한 트리의 루트를 각각 line 13과 14에서 찾고, 만일 두 루트가 같으면 line 15에서 union을 수행하지 않고 리턴

상호배타적 집합을 위한 UnionFind 클래스

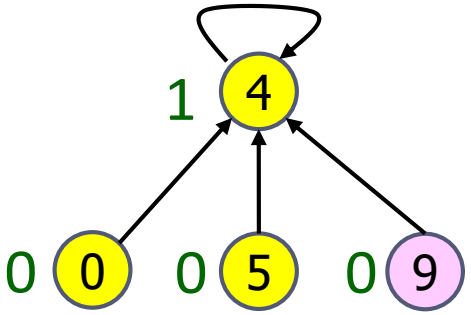
- 두 루트가 다르면, line 17~20에서 두 루트의 rank를 비교하여 큰 rank를 가진 루트(승자)가 작은 rank를 가진 루트의 부모노드로 대체
  - 승자가 합쳐진 트리의 루트로 남게 됨
  - 일련의 과정에서 승자의 rank는 변하지 않음에 주목
- Line 21~25: 두 루트의 rank가 같은 경우로, 둘 중에 하나가 승자가 됨
- 프로그램에서는 임의로 i가 속한 트리의 루트가 승자가 되도록 함.
- 마지막으로 승자의 rank를 1 증가시킴

```java
public class main {
    public static void main(String[ ] args) {
        int N = 10;
        Node[] a = new Node[N];

        for (int i = 0; i < N; i++)  // 10개 Node 객체 생성
            a[i] = new Node(i, 0);

        UnionFind uf = new UnionFind(a); // UnionFind 객체 생성

        uf.union(2, 1); uf.union(2, 6);
        uf.union(7, 3); uf.union(4, 5);
        uf.union(9, 5); uf.union(7, 2);
        uf.union(7, 8); uf.union(0, 4);

        System.out.print("8회의 union 연산 수행 후\n(i:parent,rank):");
        for(int i = 0; i < N; i++)
            System.out.print("("+i+":"+uf.a[i].getParent()+","+uf.a[i].getRank()+") ");

        uf.union(9, 1);
        System.out.print("\n\nunion(9,1) 수행 후\n(i:parent,rank):");
        for(int i = 0; i < N; i++)
            System.out.print("("+i+":"+uf.a[i].getParent()+","+uf.a[i].getRank()+") ");
        System.out.println();
    }
}
```

```
8회의 union 연산 수행 후
(i:parent,rank):(0:4,0) (1:2,0) (2:7,1) (3:7,0) (4:4,1) (5:4,0) (6:2,0) (7:7,2) (8:7,0) (9:4,0)

union(9,1) 수행 후
(i:parent,rank):(0:4,0) (1:7,0) (2:7,1) (3:7,0) (4:7,1) (5:4,0) (6:2,0) (7:7,2) (8:7,0) (9:4,0)
```



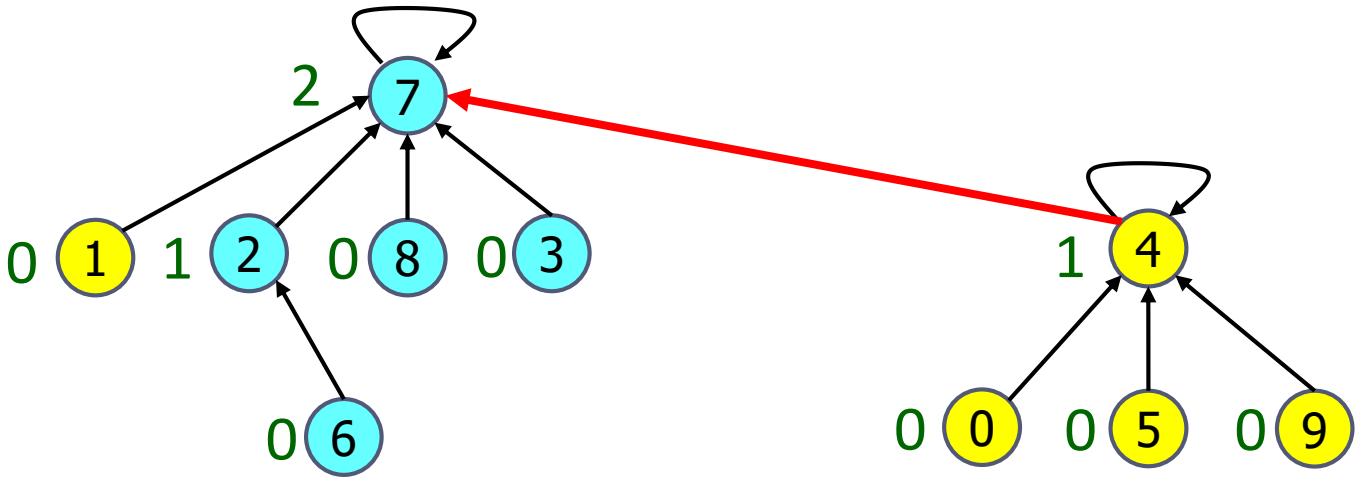
집합 {7, 2, 8, 3, 1, 6}



집합 {4, 0, 5, 8}

	0	1	2	3	4	5	6	7	8	9
a	4	2	7	7	4	4	2	7	7	4

union(9, 1) 연산 수행 전



집합 {7, 2, 8, 3, 1, 6, 4, 0, 5, 8}

	0	1	2	3	4	5	6	7	8	9
a	4	7	7	7	7	4	2	7	7	4

union(9, 1) 연산 수행 후

수행시간

- union 연산: 두 루트노드들을 각각 찾는 find 연산을 수행한 후에, rank를 비교하여 승자가 합쳐진 트리의 루트노드로 남음
- rank가 같은 경우엔 둘 중에 하나가 승자가 되고 승자의 rank를 1 증가시킴. 그러므로 find 연산을 제외한 순수 union 연산의 수행시간은 O(1) 시간
- find 연산의 수행시간: 최대 트리의 높이만큼 올라가야 하므로 트리의 높이에 비례
- find 연산을 수행하며 경로압축을 하므로, 경로상의 노드에 대해 추후에 수행되는 find 연산의 수행시간은 트리의 높이보다는 적게 소요

- **상각분석: O(N)번의 find와 union 연산들을 수행하여 걸린 총 시간을 연산 횟수로 나누어 1회의 연산 수행시간을 계산**
- **상각분석 결과: 1회의 find 연산의 수행시간이 O(log*N)**
  - log*N = 1이하의 값을 얻기 위해 N에다가 log연산을 연속적으로 수행해야 하는 횟수
  - N = 2, 4, 16, 65536, 265536일 때 log*N의 값

$$\log^*2 = 1,\ \log^*2^2 = 2,\ \log^*2^{2^2} = 3,\ \log^*2^{2^{2^2}} = 4,\ \log^*2^{2^{2^{2^2}}} = 5$$

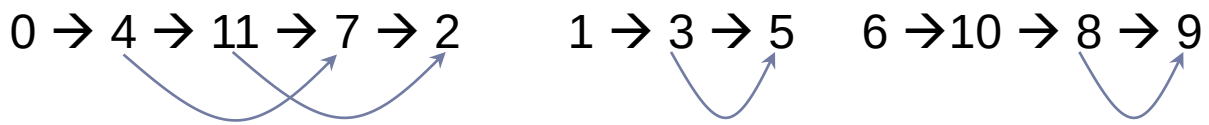
응용예 : 동치 관계 (1)

- **관계 ≡ 의 특성**
  - (1) 반사적(reflexive) : 다각형 x에 대해서, x≡x가 성립
  - (2) 대칭적(symmetric) : 다각형 x,y에 대해서, x≡y이면 y≡x
  - (3)이행적(transitive): 다각형 x, y, z에 대해서, x≡y이고 y≡z이면 x≡z
- **동치 관계(equivalence relation)**
  - 집합 S에 대하여 관계 ≡ 가 대칭적, 반사적, 이행적
- **동치부류(equivalence class)**
  - x≡y이면 x, y는 같은 동치부류
  - ex) 0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0
    →{ 0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

응용예 : 동치 관계 (2)

- **동치 결정 알고리즘**
  - 동치 쌍(equivalence pair) (i,j)를 읽어 기억
  - 0에서부터 시작해서 (0,j) 형태의 모든 쌍을 찾는다.
    (j,k)가 있으면 k도 0과 같은 동치부류에 포함시킨다.
  - 0이 포함된 동치 부류의 모든 객체를 찾아 표시하고 출력
  - 아직 출력되지 않은 객체가 있으면,
    새로운 동치 부류이므로 전과 같은 방법으로 수행하여 출력

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



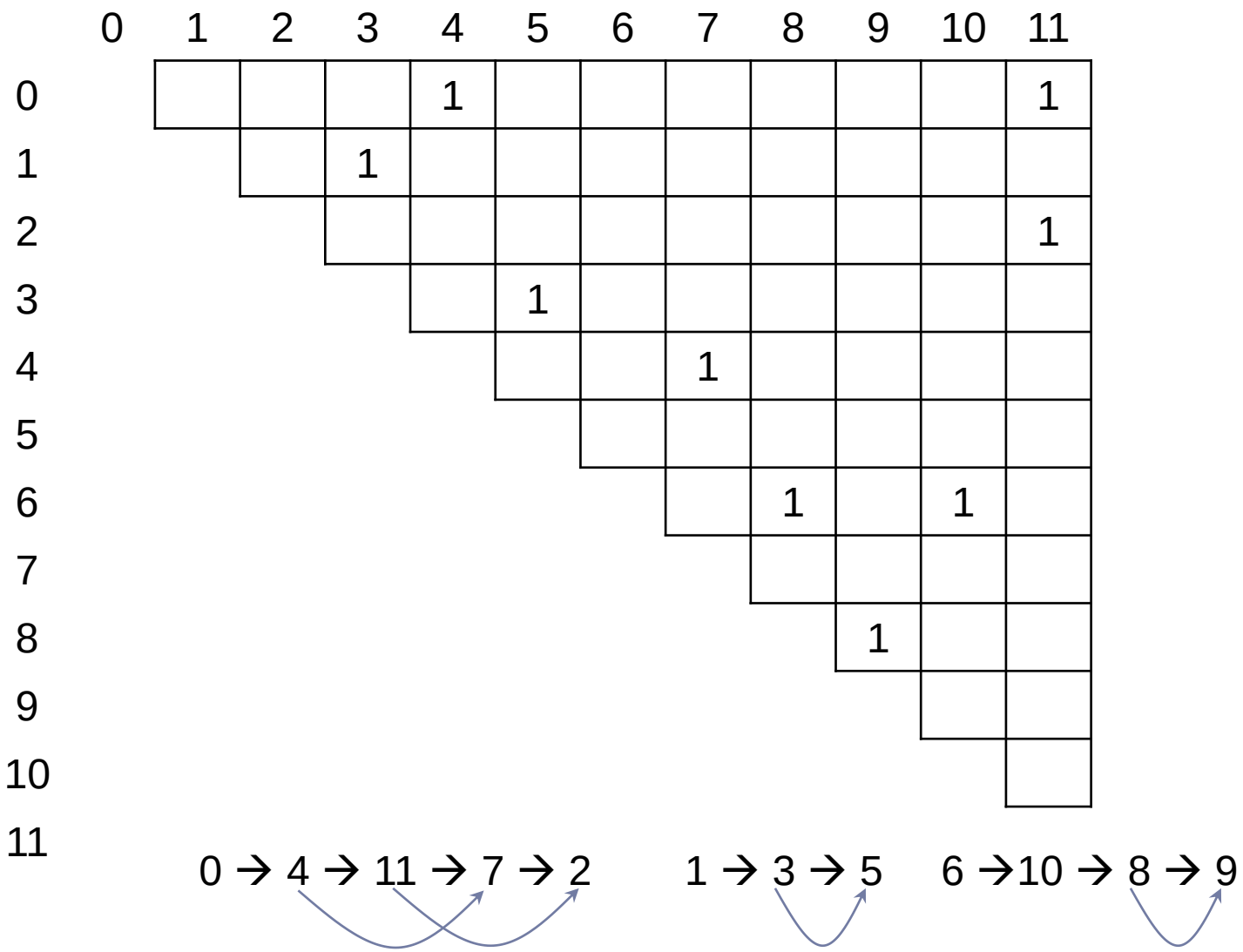
응용예 : 동치 관계 (3)

- **Boolean 배열을 사용한 동치 알고리즘**
  - m : 동치 쌍의 수 / n : 객체 수
  - pairs[i][j]= true : i와 j가 입력쌍

```
void equivalence()
{
        초기화;
        while 나머지 쌍
        {
              다음 쌍 (i,j)를 읽음;
              이 쌍을 처리;
        }
        출력을 위해 초기화;
        for (출력이 안된 객체에 대해)
              이 객체를 포함하고 있는 동치 부류를 출력;
 }
```

  - 배열의 극히 일부 원소만 사용 : 기억 장소 낭비
  - 배열의 초기화를 위해 $\Theta(n^2)$시간 필요

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0

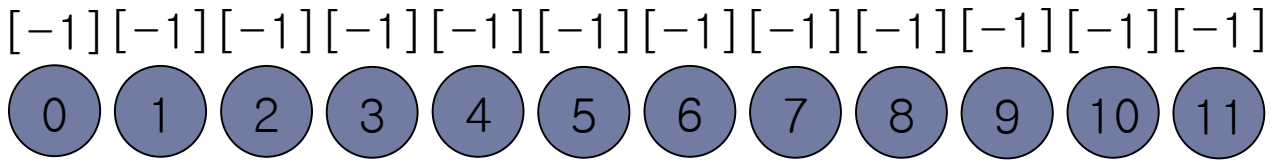


응용예 : 동치 관계 (4)

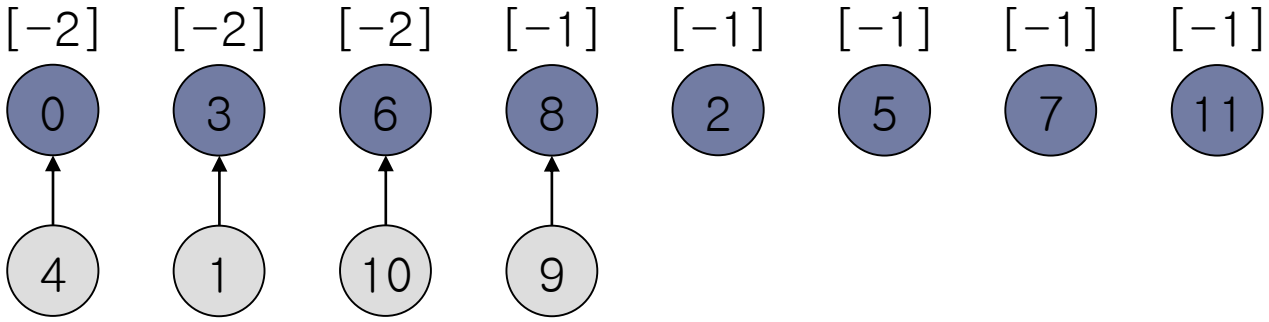
- **동치 쌍(equivalence pair)의 처리**
  - 동치 부류(equivalence class)를 집합으로 간주
  - i≡j
    - i와 j를 포함하고 있는 집합 찾기
    - 다른 집합에 포함된 경우 합집합으로 대체
    - 같을 때는 아무 작업도 수행할 필요 없음
  - n개의 변수, m개의 동치 쌍
    - 초기 포리스트 형성 : O(n)
    - 2m개의 탐색, 최대 min{n-1,m}개의 합집합
      : O(n+mα(2m, min{n-1,m}))

응용예 : 동치 관계 (5)

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



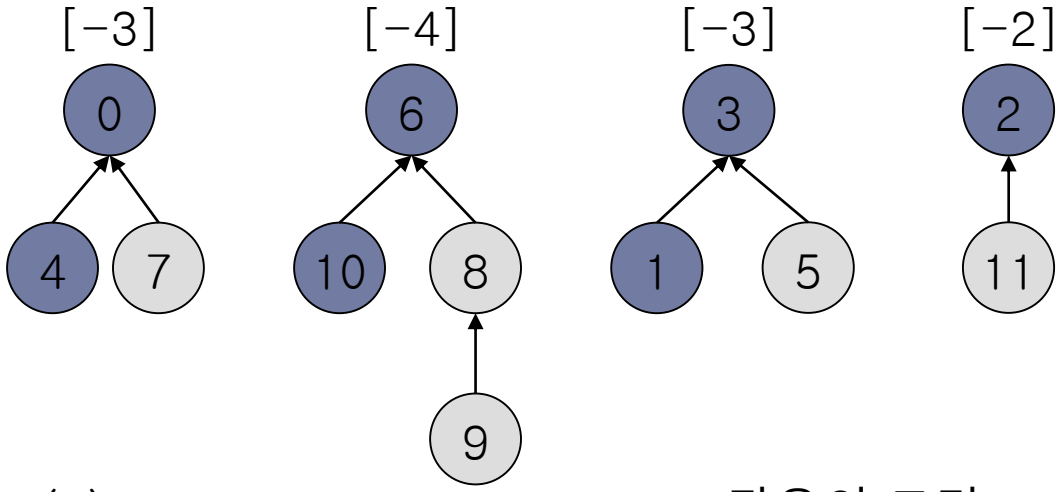
(a) Initial trees



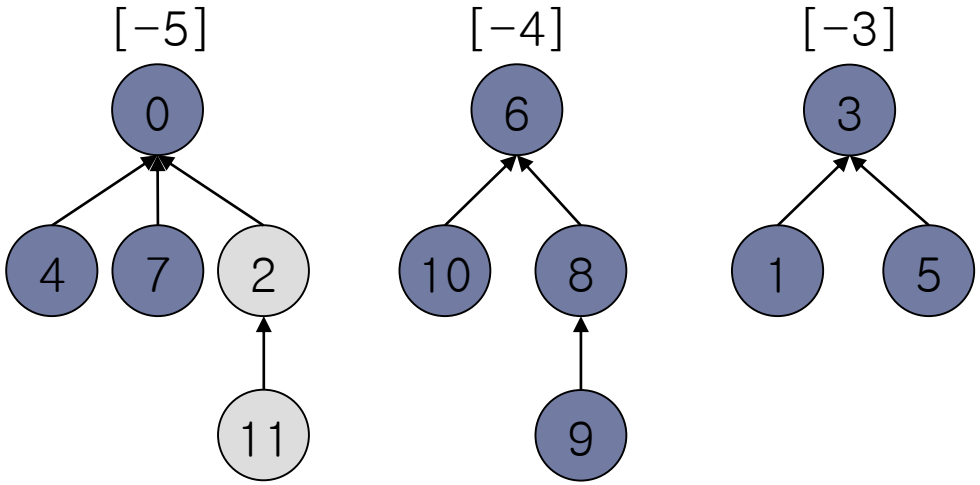
0≡4, 3≡1, 6≡10, and 8≡9 다음의 높이-2 트리

응용예 : 동치 관계 (6)

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



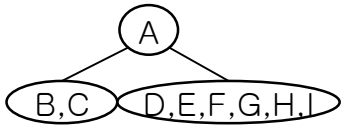
(c) 7≡4, 6≡8, 3≡5,and 2≡11 다음의 트리



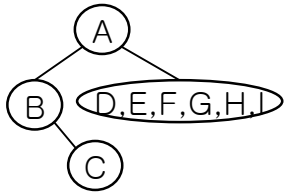
(d) 11≡0 다음의 트리

스택 순열 (1)

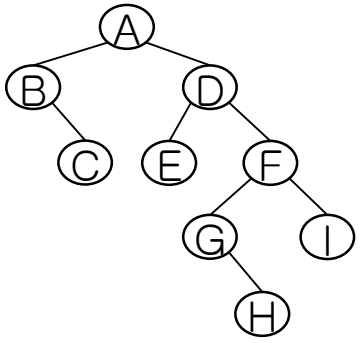
- **전위 순서 ABCDEFGHI, 중위 순서 BCAEDGHFI**
  - 전위 순서: A는 트리의 루트
  - 중위 순서: BC는 A의 왼쪽 서브트리, EDGHFI는 오른쪽 서브트리



  - 전위 순서: B가 다음번 루트
  - 중위 순서: B의 왼쪽 서브트리는 공백, 오른쪽은 C

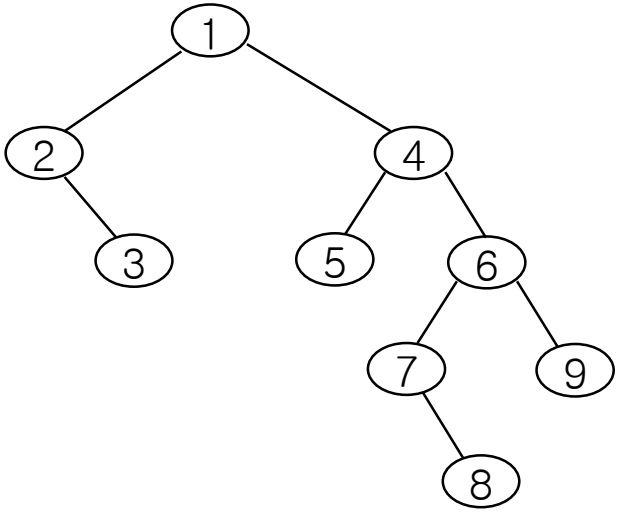


  - 반복



스택 순열 (2)

- **전위 순열(preorder permutation)**
  - 이진 트리를 전위 순회에 따라 방문한 노드들의 순서



- **중위 순열(inorder permutation)**
  - 이진 트리를 중위 순회에 따라 방문한 노드들의 순서

- **모든 이진 트리는 유일한 전위-중위 순서 쌍을 가짐**

응용

- 9.4절의 Kruskal의 최소신장트리 알고리즘
- 트리에서 가장 가까운 공통 조상노드(Least Common Ancestor) 찾기
- 네트워크의 연결 검사
- 퍼콜레이션(Percolation)
- 이미지 처리(Image Processing)
- 조각그림 맞추기(Jigsaw Puzzle)
- 바둑 같은 게임 등에 활용

요약

- 트리는 계층적 자료구조로서 배열이나 연결리스트의 단점을 보완하는 자료구조
- 왼쪽자식-오른쪽형제 표현은 노드의 차수가 일정하지 않은 일반적인 트리를 구현하는 매우 효율적인 자료구조
- 포화이진트리는 각 내부노드가 2개의 자식 노드를 가지는 트리
- 완전이진트리는 마지막 레벨을 제외한 각 레벨이 노드들로 꽉 차있고, 마지막 레벨에는 노드들이 왼쪽부터 빠짐없이 채워진 트리이다.
- 포화이진트리는 완전이진트리이다.

요약

- **이진트리의 순회 방법**
  - 전위순회(NLR)
  - 중위순회(LNR)
  - 후위순회(LRN)
  - 레벨순회-레벨순회는 큐 자료구조를 사용해서 구현
- **이진트리 높이 계산과 노드 수의 계산에는 후위순회가 적합, 이진트리의 비교에는 전위순회가 적합**
- **스택 없이 이진트리를 순회하기 위해 노드의 null 레퍼런스 대신 다음에 방문할 노드의 레퍼런스를 저장한 이진트리를 스레드 이진트리라고 함**
- **이진트리의 높이 및 노드 수의 계산, 각 트리 순회, 동일성 검사는 트리의 모든 노드들을 방문해야 하므로 각각 O(N) 시간이 소요**

요약

- 상호배타적 집합의 union과 find연산을 효율적으로 수행하기 위해, union 은 rank기반 연산을 수행하고, find 연산은 경로압축을 수행
- union연산의 수행시간: O(1) 시간
- find 연산의 수행시간: O(log*N)