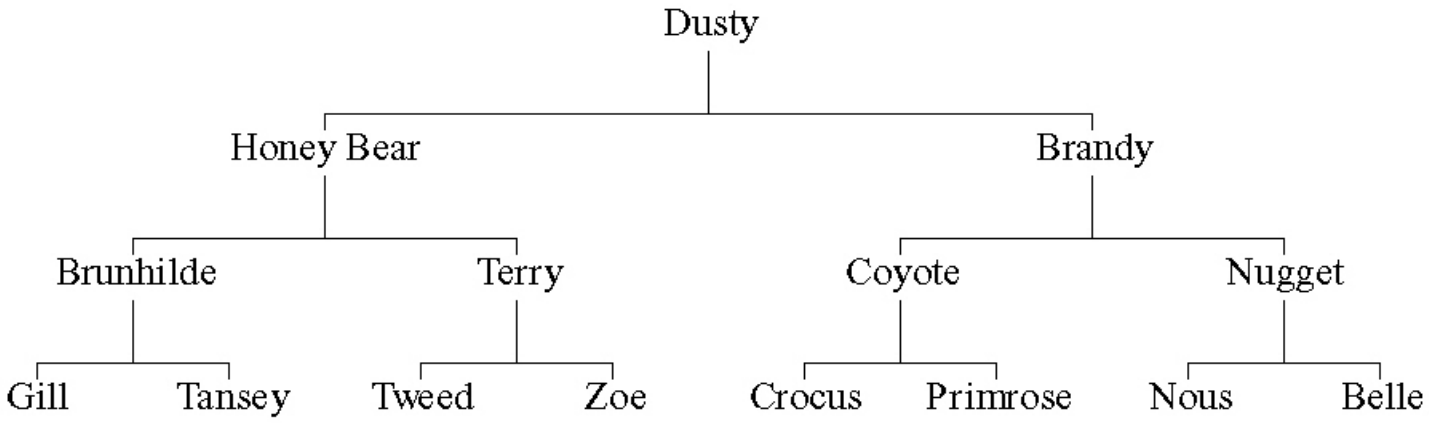
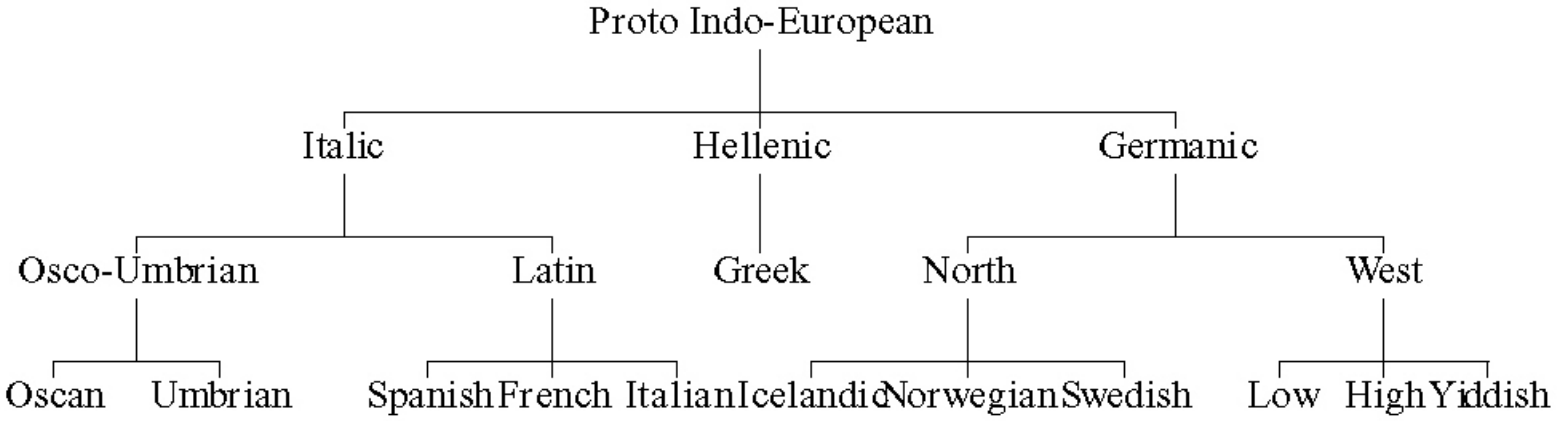
CHAPTER 5
Trees

트리 구조



(a) Pedigree



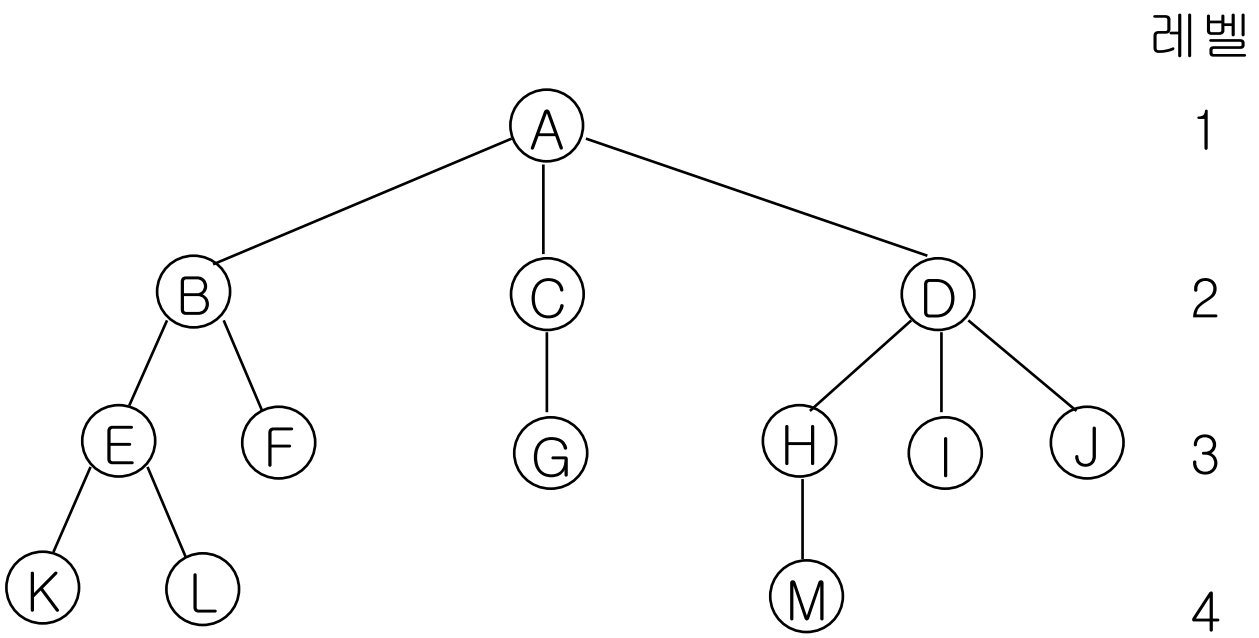
(b) Lineal

Trees

◆ (정의) 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 $n \geq 0$ disjoint sets $T_1, T_2, \ldots, T_n$,where each of these sets is a tree. $T_1, T_2, \ldots, T_n$ are called the subtrees of the root.



트리의 표현

–A node stands for the item of information plus the branches to other nodes.

–노드의 차수(degree) : 노드의 서브트리 수

(예) A의 차수=3, C의 차수=1, F의 차수=0

–Leaf node(=terminal node): 차수가 0인 노드

(예)K,L,F,G,M,I,J

–비단말 노드(nonterminal node) : 차수 ≠ 0인 노드

–노드 X의 children = 노드 X의 서브트리의 루트(roots)

(예) D의 children은 H,I,and J (반대 개념은 parent)

–Sibling(형제) : parent가 같은 children (예) H, I, J

–트리의 차수 = max{노드의 차수} (예) 그림은 차수 3의 트리

–조상(ancestors) : (예)M의 ancestors는 A, D, and H.

–노드 레벨 : 루트-레벨1, 자식 레벨=부모 레벨+1

–트리의 높이(깊이) = max{노드 레벨}

리스트 표현

– 트리의 리스트 표현

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

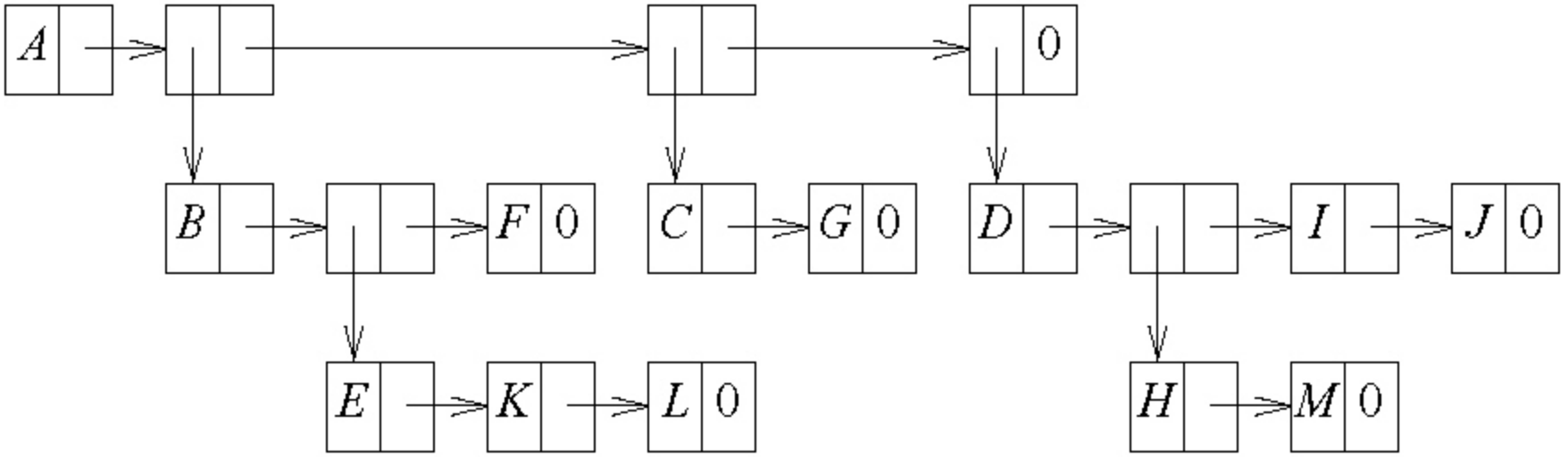


Fig.5.4 Possible node structure for a tree of degree *k*

DATA	CHILD 1	CHILD 2	···	CHILD *k*

– 공간 낭비 많음

◆ *k*-ary tree에서 노드 수가 *n*이면 *nk*의 자식 필드 중 *n*(*k*-1)+1개 필드가 0

왼쪽 자식-오른쪽 형제 표현

◆ 노드 구조

data	
left child	*right sibling*

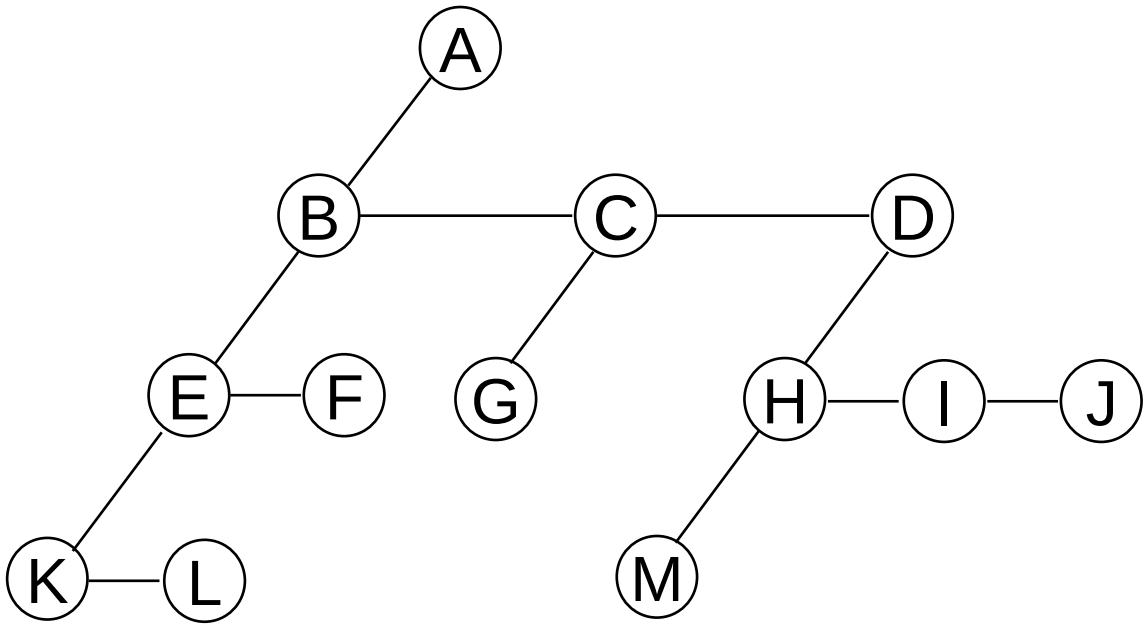
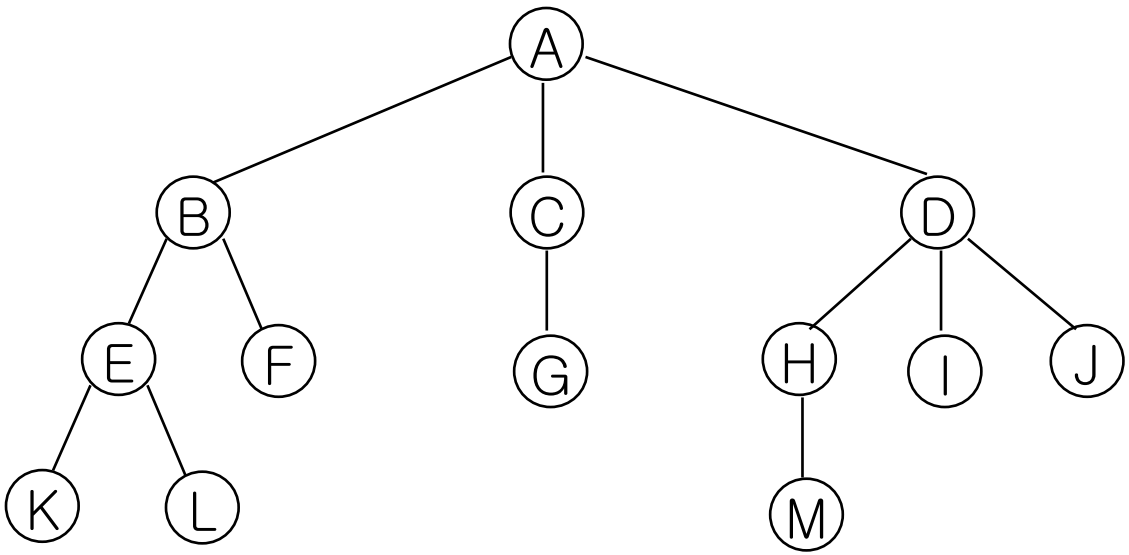


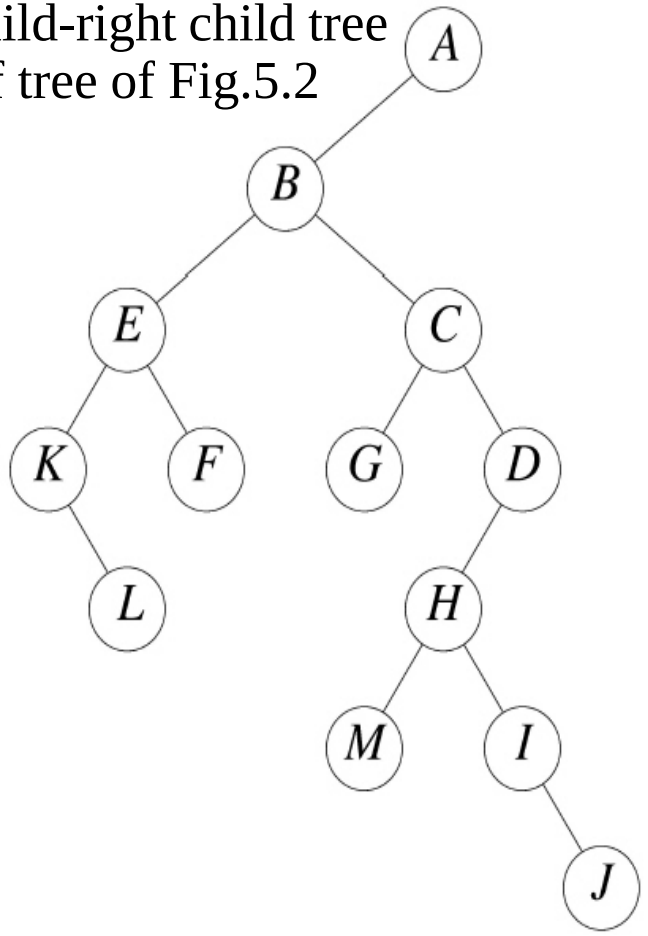
Fig.5.6 Left child-right sibling representation of tree of Fig.5.2

Representation as a Degree-Two Tree

- 차수가 2인 트리
  - 왼쪽 자식-오른쪽 형제 트리의 오른쪽 형제 포인터를 45°회전
  - 루트 노드의 오른쪽 자식은 공백

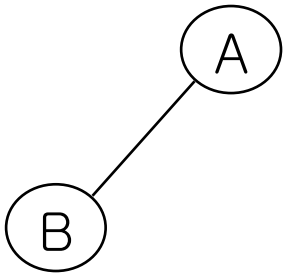
- 이진 트리(**binary tree**)
  - Fig.5.7: Left-child-right child tree representation of tree of Fig.5.2



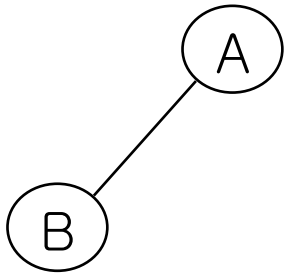


Tree representations

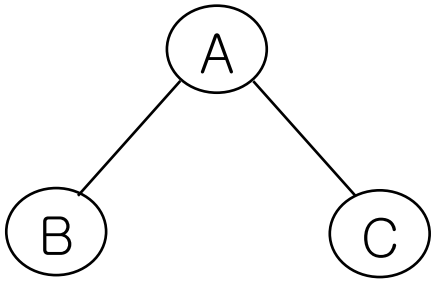
트리



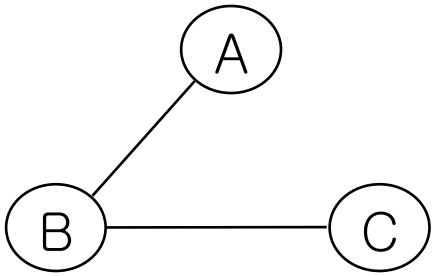
왼쪽 자식-오른쪽 형제 트리



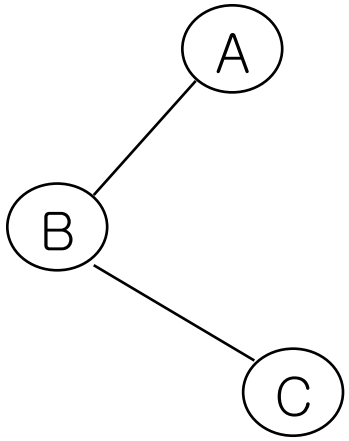
이진트리



트리



왼쪽 자식-오른쪽 형제 트리



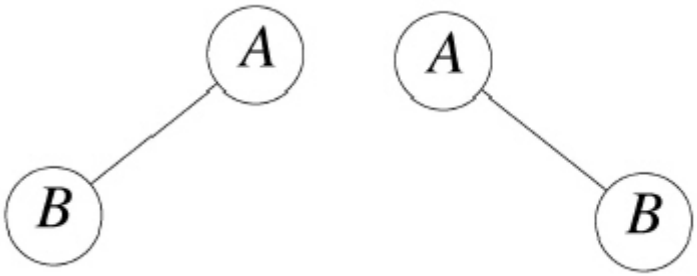
이진트리

Binary Trees (1)

- **Binary Tree(이진 트리)의 정의**
  - A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

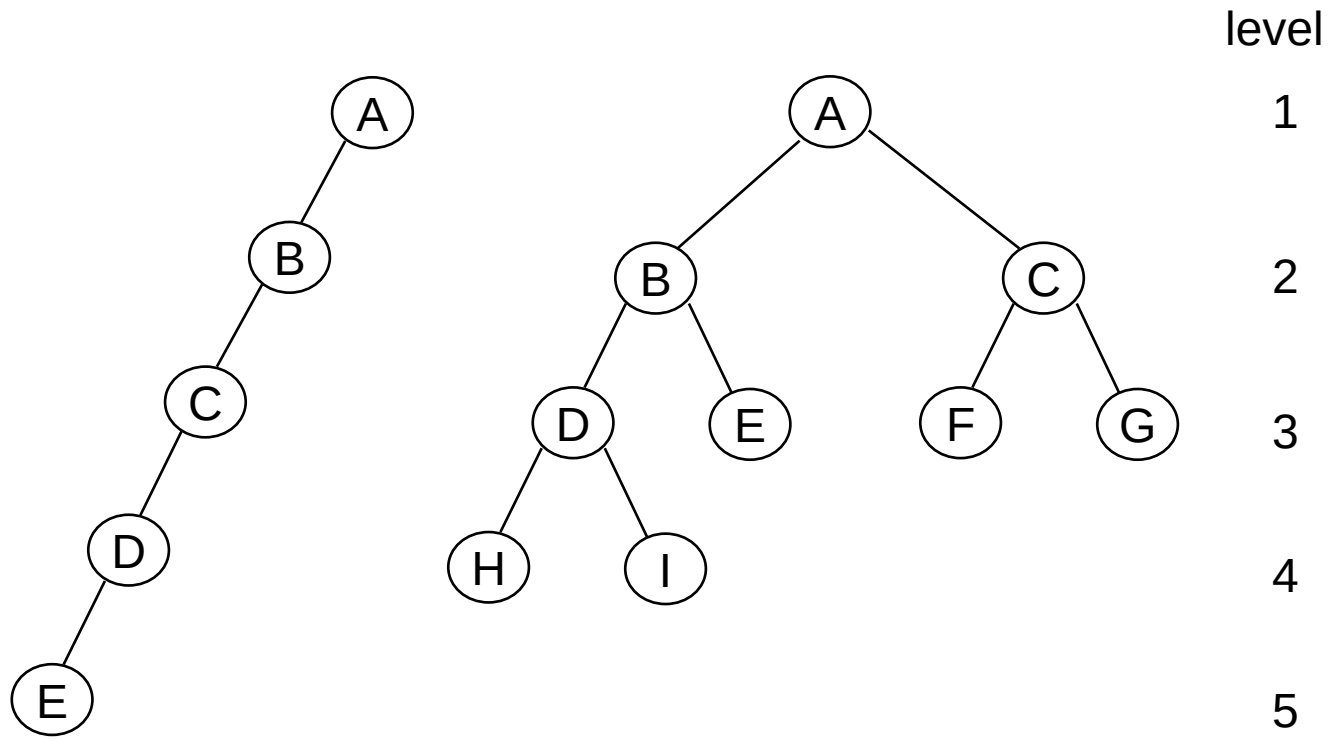
Binary Trees (2)

- 이진 트리와 일반 트리의 차이점
  - Empty tree도 binary tree(공백 이진 트리)이지만, tree는 최소 한 개의 root node가 있어야 함: Definition상의문제일 뿐
  - 트리에서는 어느 노드의 child를 순서로 구분. 이진트리에서는 child가 한 개인 경우도 left child인가 right child인가에 따라 다른 이진트리로 간주됨
    - 다음의 두 이진 트리는 다른 이진 트리다.



Binary Trees (3)

- 편향 **(skewed)** 이진 트리와 완전**(complete)** 이진 트리



(a) 편향 이진 트리 (b) 완전 이진 트리

Properties of Binary Trees (1)

- **Lemma 5.2 [Maximum number of nodes]**
  (1) 레벨 i에서의 최대 노드수 : $2^{i-1}(i \geq 1)$
  (2) 깊이가 k인 이진 트리가 가질 수 있는 최대 노드수
  : $2^k - 1(k \geq 1)$
  - 증명 (1)
    - 귀납 기초 : 레벨 i=1일 때 루트만이 유일한 노드이므로
      레벨 i=1에서의 최대 노드 수 : $2^{1-1} = 2^0 = 1$
    - 귀납 가설 : i를 1보다 큰 임의의 양수라고 가정.
      레벨 i-1에서의 최대 노드 수는 2^{i-2}라고 가정
    - 귀납 과정 : 가설에 의해 레벨 i-1의 최대 노드 수는 2^{i-2}
    -
      각 노드의 최대 차수는 2이므로
      레벨 i의 최대 노드 수는 레벨 i-1에서의 최대 노드 수의 2배
      $\therefore 2^{i-1}$
  - 증명 (2)

$$\sum_{i=1}^{k}(\text{레벨 } i\text{의 최대 노드 수}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Properties of Binary Trees (2)

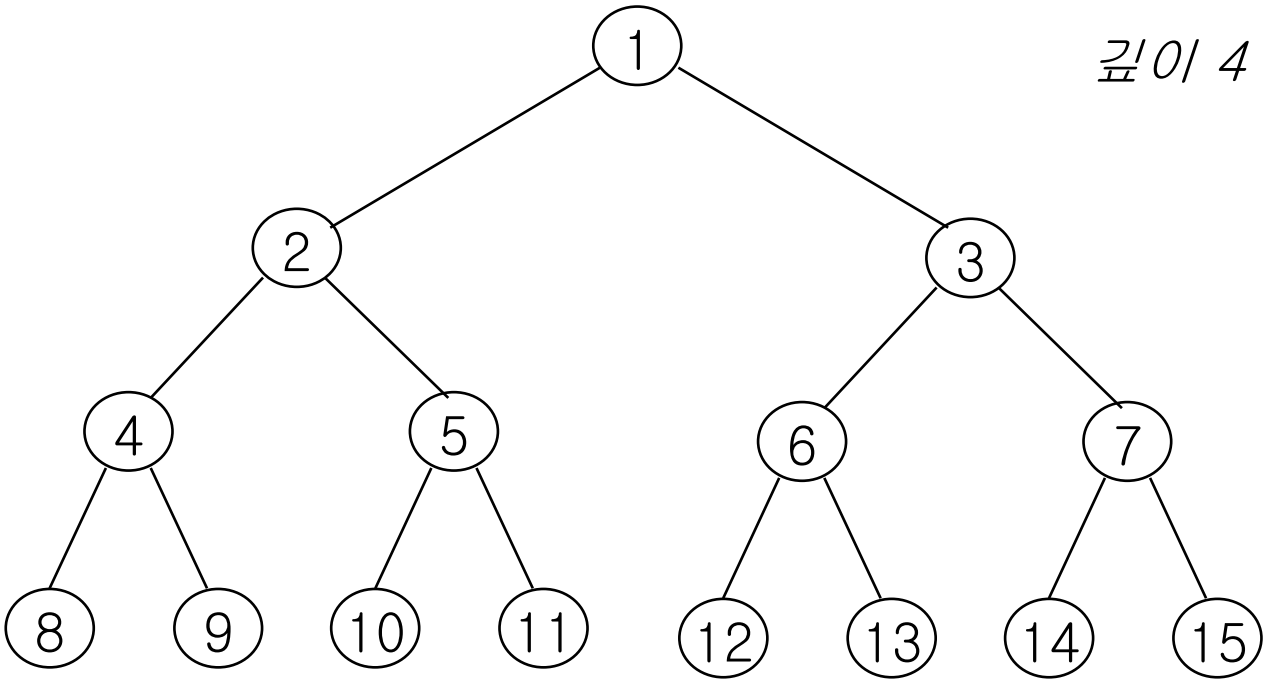
- 리프 노드 수와 차수가 2인 노드 수와의 관계
  - $n_0 = n_2 + 1$
    - n_0 : 리프 노드 수
    - n_2 : 차수가 2인 노드 수
  - 증명
    - n_1 : 차수 1인 노드 수, n : 총 노드 수, B : 총 가지 수
    - $n = n_0 + n_1 + n_2$ (1)
    - 루트를 제외한 모든 노드들은 들어오는 가지가 하나씩 있으므로 $n = B + 1$
    - 모든 가지들은 차수가 2 또는 1인 노드에서 뻗어 나오므로 $B = n_1 + 2n_2$

    $\therefore\ n = B + 1 = n_1 + 2n_2 + 1$ (2)

    (1)과 (2)에서 $n_0 = n_2 + 1$

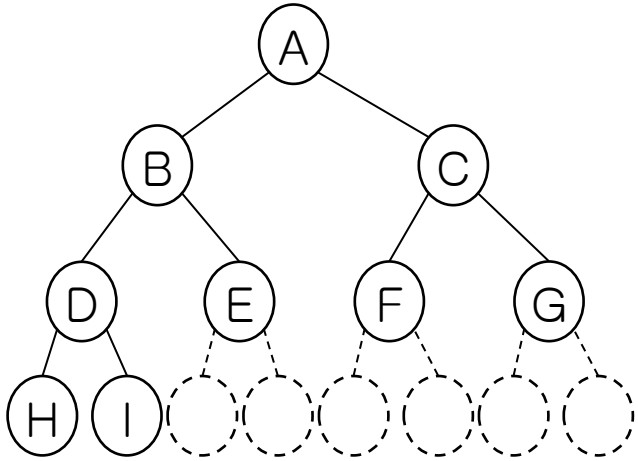
Properties of Binary Trees (3)

- **포화 이진 트리(full binary tree)**
  - 깊이가 k이고, 노드수가 2^k-1 ($k \geq 0$)인 이진 트리
  - 노드 번호 $1,\ldots,2^k-1$ 까지 순차적 부여 가능



Properties of Binary Trees (4)

- **완전 이진 트리(Complete binary tree)**
  - 깊이가 k이고 노드 수가 n인 이진 트리
  - 각 노드들이 깊이가 k인 포화 이진 트리에서 1부터 n까지 번호를 붙인 노드와 1대 1로 일치
  - n 노드 완전 이진 트리의 높이 : $\lceil \log_2(n+1) \rceil$



이진 트리의 표현 (1)

◆ 배열 표현

- 1차원 배열에 노드를 저장
- 보조 정리 5.4
  - n 개의 노드를 가진 완전이진트리

① $parent(i)$: $\lfloor i/2 \rfloor$ **if** $i \neq 1$

② $leftChild(i)$: $2i$ **if** $2i \leq n$

왼쪽 자식 없음 **if** $2i > n$

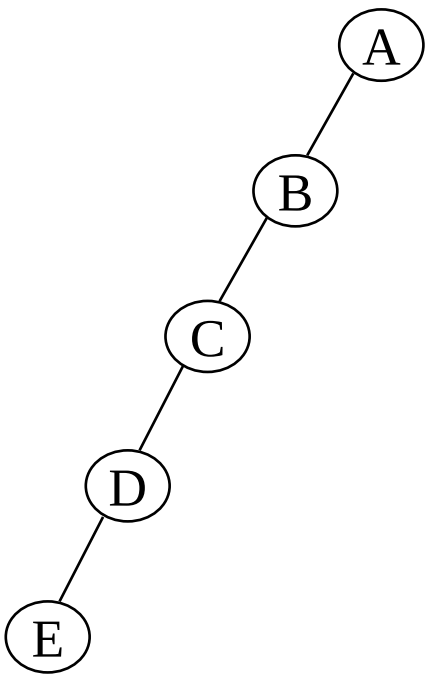
③ $rightChild(i)$: $2i+1$ **if** $2i+1 \leq n$

오른쪽 자식 없음 **if** $2i + 1 > n$

- 완전 이진 트리 : 낭비 되는 공간 없음
- 편향 트리 : 공간 낭비
  - 최악의 경우, 깊이 k 편향 트리는 2^k-1중 k개만 사용

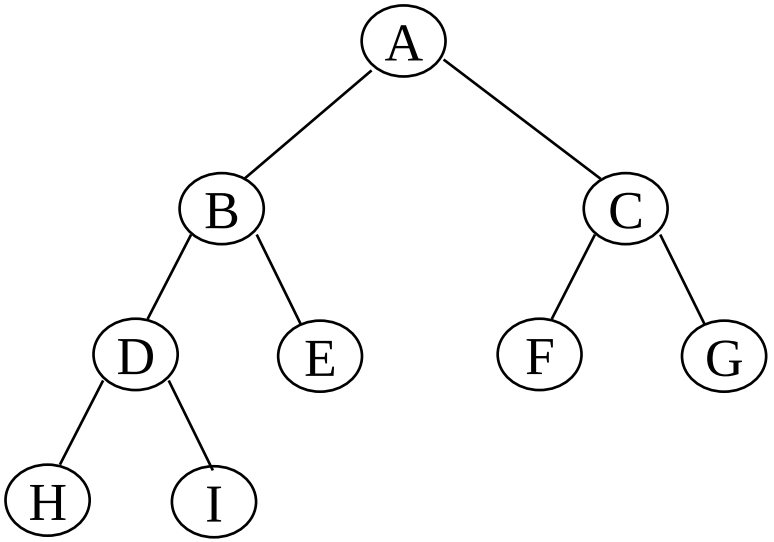
이진 트리의 표현 (2)

편향 트리



[0]	-
[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
.	.
.	.
.	.
[16]	E

완전 이진 트리

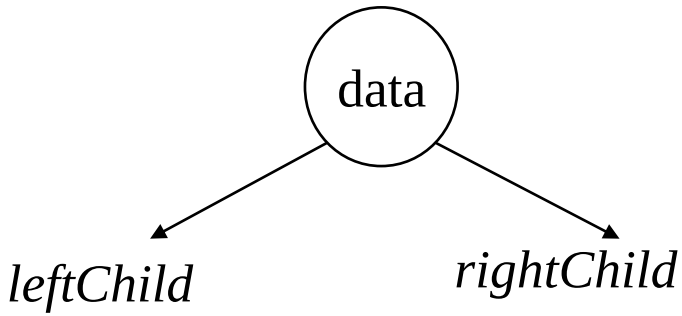


[0]	-
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

이진 트리의 표현 (3)

- 노드 표현

leftChild	*data*	*rightChild*



- 문제점: parent를 알기 어려움
  - *parent* 필드 추가

이진 트리의 표현 (3)

- 클래스 정의

```
template <class T> class Tree; //전방 선언
template <class T>
class TreeNode{
template <class T2>     //교재p.257 수정 필요
friend class Tree<T2>;
private:
   T data;
   TreeNode<T> *leftChild;
   TreeNode<T> *rightChild;
};
```

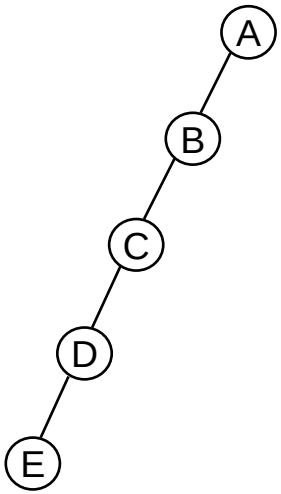
노드는 다음 같이 struct로 사용해도 무방함

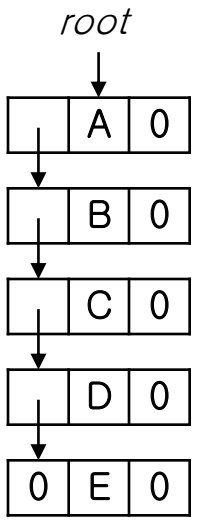
```
template <class T>
struct TreeNode {
    T data;
    TreeNode<T> *leftChild;
    TreeNode<T> *rightChild;
};
```

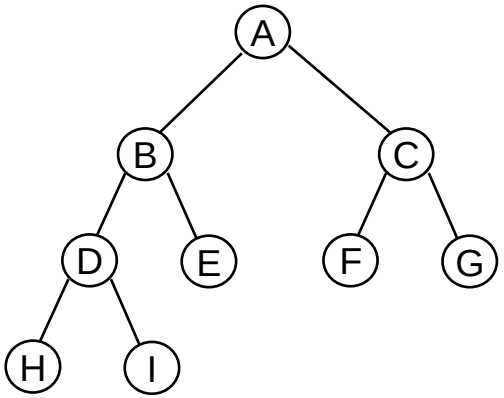
```
template <class T>
class Tree {
public:
      // 트리 연산들
      ...
private:
      TreeNode<T> *root;
};
```

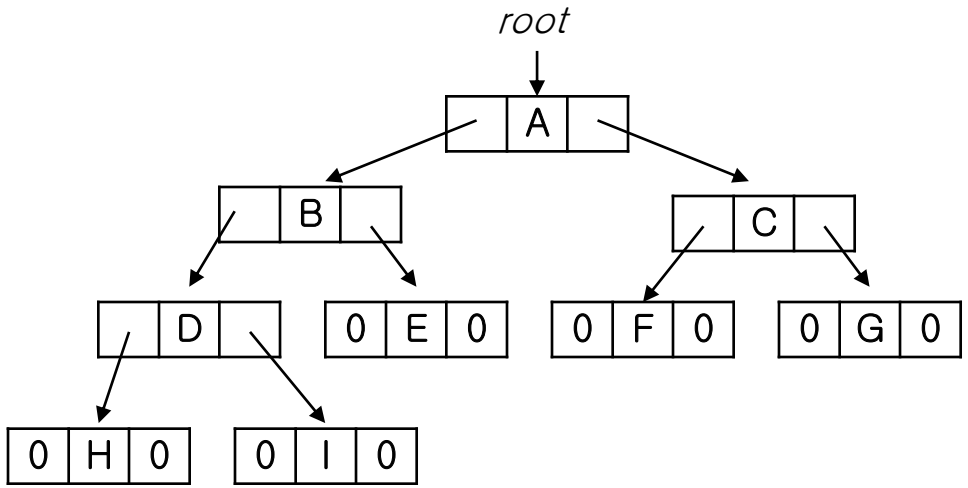
이진 트리의 표현 (4)

- 연결 표현의 예



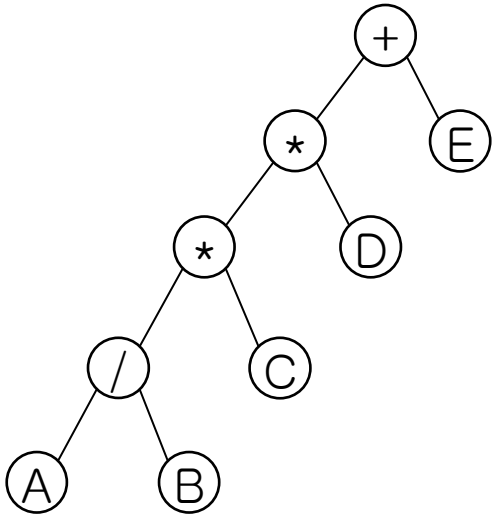






이진 트리 순회와 트리 반복자

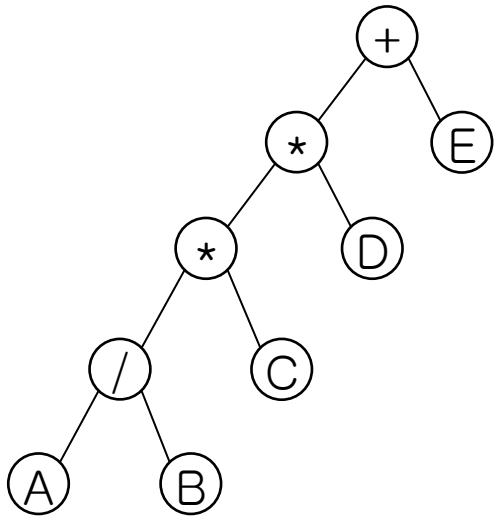
- **트리 순회(tree traversal)**
  - 트리에 있는 모든 노드를 한 번씩만 방문
  - 순회 방법 : LVR, LRV, VLR, VRL, RVL, RLV
    - L : 왼쪽 이동, V : 노드방문, R : 오른쪽 이동
  - 왼쪽을 오른쪽보다 먼저 방문(LR)
    - LVR : 중위(inorder) 순회
    - VLR : 전위(preorder) 순회
    - LRV : 후위(postorder) 순회
  - 산술식의 이진트리 표현



Inorder Traversal

- 왼쪽 자식 방문 -> 루트 방문 ->오른쪽 자식 방문
- 출력 결과: **A / B * C * D + E**

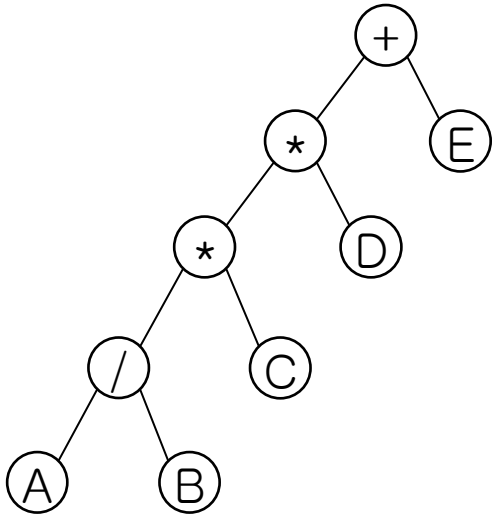
```
template <class T>
void Tree<T>::Inorder()
{//이진 트리의 Inorder Traversal
  Inorder(root);
}

template <class T>
void Tree<T>::Inorder(TreeNode<T> *currentNode)
{// Workhorse traverses the subtree rooted at currentNode.
 // The workhorse is declared
 // as a private member function of Tree.
  if (currentNode){
    Inorder(currentNode->leftChild);
    Visit(currentNode);
    Inorder(currentNode->rightChild);
  }
}
```



Inorder Traversal(non-recursive version)

- 왼쪽 자식 방문 -> 루트 방문 ->오른쪽 자식 방문
- 출력 결과: **A / B * C * D + E**

```
template <class T>
void Tree<T>::NonrecInorder()
{//이진 트리의 Nonrecursive Inorder Traversal
  Stack<TreeNode<T> *> s;
  TreeNode<T> *currentNode = root;
  while (1) {
    while (currentNode) { // Move down leftChild fields
      s.Push(currentNode); // add to stack
      currentNode = currentNode->leftChild
    }
    if (s.IsEmpty()) return;
    currentNode = s.Top(); s.Pop(); // stack 에서 꺼냄
    Visit(currentNode);
    currentNode = currentNode->rightChild;
  }
}
```



Inorder Traversal

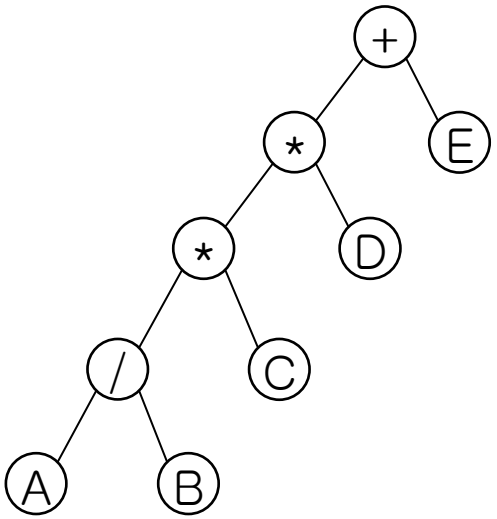
호출	*currentNode*의 값	조치	호출	*currentNode*의 값	조치
Driver	+		10	C	
1	*		11	0	
2	*		10	C	cout<<‘C’
3	/		12	0	
4	A		1	*	cout<<‘*’
5	0		13	D	
4	A	cout<<‘A’	14	0	
6	0		13	D	cout<<‘D’
3	/	cout<<‘/’	15	0	
7	B		Driver	+	cout<<‘+’
8	0		16	E	
7	B	cout<<‘B’	17	0	
9	0		16	E	cout<<‘E’
2	*	cout<<‘*’	18	0	

출력 : A/B*C*D+E

Preorder Traversal

- 루트 방문 -> 왼쪽 자식 방문 -> 오른쪽 자식 방문
- 출력결과: **+ * * / A B C D E**

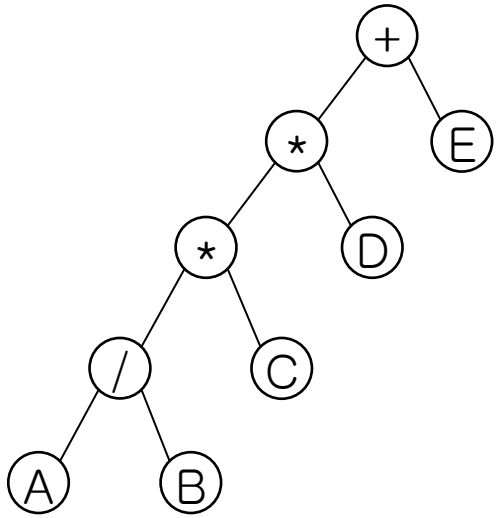
```
template <class T>
void Tree<T>::Preorder()
{//이진 트리의 Preorder Traversal
  Preorder(root);
}

template <class T>
void Tree<T>::Preorder(TreeNode<T> *currentNode)
{// Workhorse traverses the subtree rooted at currentNode.
 // The workhorse is declared
 // as a private member function of Tree.
  if (currentNode){
    Visit(currentNode);
    Preorder(currentNode->leftChild);
    Preorder(currentNode->rightChild);
  }
}
```



Postorder Traversal

- 왼쪽 자식 방문 ->오른쪽 자식 방문 -> 루트 방문
- 출력결과: **A B / C * D * E +**

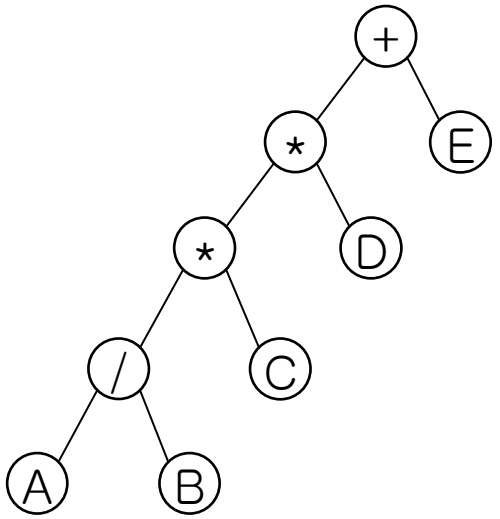
```
template <class T>
void Tree<T>::Postorder()
{//이진 트리의 Postorder Traversal
  Postorder(root);
}

template <class T>
void Tree<T>::Postorder(TreeNode<T> *currentNode)
{// Workhorse traverses the subtree rooted at currentNode.
 // The workhorse is declared
 // as a private member function of Tree.
  if (currentNode){
    Postorder(currentNode->leftChild);
    Postorder(currentNode->rightChild);
    Visit(currentNode);
  }
}
```



Level Order Traversal(레벨 순서 순회)

- 큐 사용하여 구현
- 루트 방문->왼쪽 자식 방문->오른쪽 자식 방문하되 각 레벨별로 최좌측 노드에서 최 우측 노드를 모두 방문함
- 출력결과: **+ * E * D / C A B**

```
template <class T> //이진 트리의 레벨 순서 순회
void Tree<T>::Levelorder() {
  Queue<TreeNode<T>*> q;  // 큐를 사용하자
  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();  // 큐에서 꺼내자.
    q.Pop();
  }
}
```



Traversal without a Stack

- **Recursive 호출 시 스택을 사용한다.**

- **각 노드에 *parent* (부모)필드 추가**
  - 스택을 사용하지 않아도 루트 노드로 올라갈 수 있음

- **Threaded binary tree로 표현**
  - 각 노드마다 두 비트 필요
  - leaf의 *leftChild* 필드와 *rightChild* 필드를 사용하여 루트로 돌아가는 경로를 유지함
  - The stack of addresses is stored in the leaf nodes.

The Satisfiability Problem (1)

- 변수 $x_1, x_2, \ldots, x_n$ 과 연산자 ∧, ∨, ¬으로 이루어진 식
  - 변수의 값은 true 나 false

- **Expression이란?**

  (1) A variable is an expression

  (2) If x and y are expressions then x∧y, x∨y, and ¬x are expressions

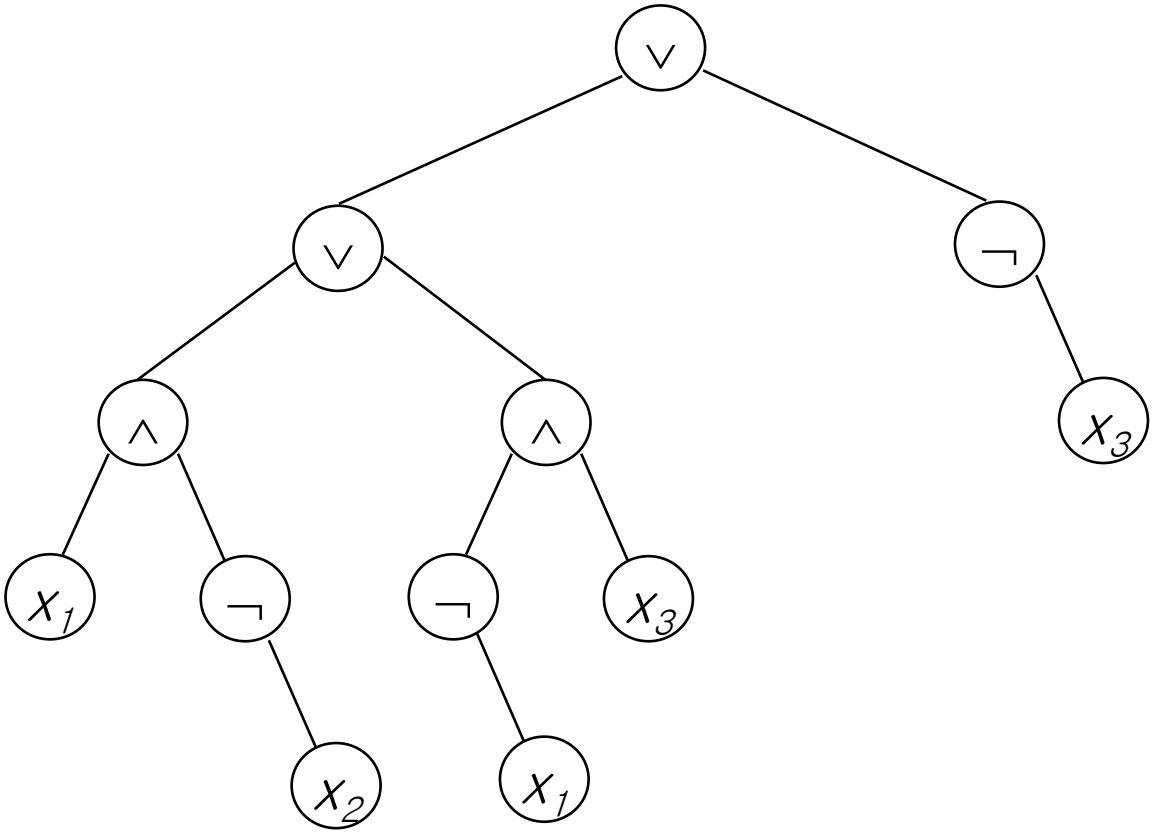
  - operator 계산 순서: 괄호, ¬, ∧, ∨

- **The formulas of the propositional calculus**
  - 위 규칙을 이용해서 구성한 식
  - $x_1 \vee (x_2 \wedge \neg x_3)$

The Satisfiability Problem (2)

- **명제식의 만족성(satisfiability) 문제**
  - 식의 값이 true가 되도록, 변수에 값을 지정할 수 있는 방법이 있는가?

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



The Satisfiability Problem (3)

- 가능한 모든 **true**와 **false**의 조합을 대입 : **O($g\ 2^n$)**
  - g : 식을 계산하는데 필요한 시간
  - 전체 식이 하나의 값이 될 때까지 서브 트리들을 계산하면서 트리를 후위 순회(postorder traversal)

x_1	x_2	x_3	formula
T	T	T	?
T	T	F	?
T	F	T	?
T	F	F	?
F	T	T	?
F	T	F	?
F	F	T	?
F	F	F	?

The Satisfiability Problem (4)

- 후위 순회 계산
  - Tree 템플릿 클래스를 *T=pair<Operator*, **bool**>로 인스턴스화
  - **enum** *Operator* { *Not*, *And*, *Or*, *True*, *False*};

```
for n개 변수에 대한 2^n개의 가능한 truth vale 조합
{
  다음 조합을 생성;
  그들의 값을 변수에 대입;
  명제식을 나타내는 트리를 후위 순회로 계산;
  if (formula.Data().second())
    {cout << current combination; return;}
}
cout  <<  "no satisfiable combination";
```

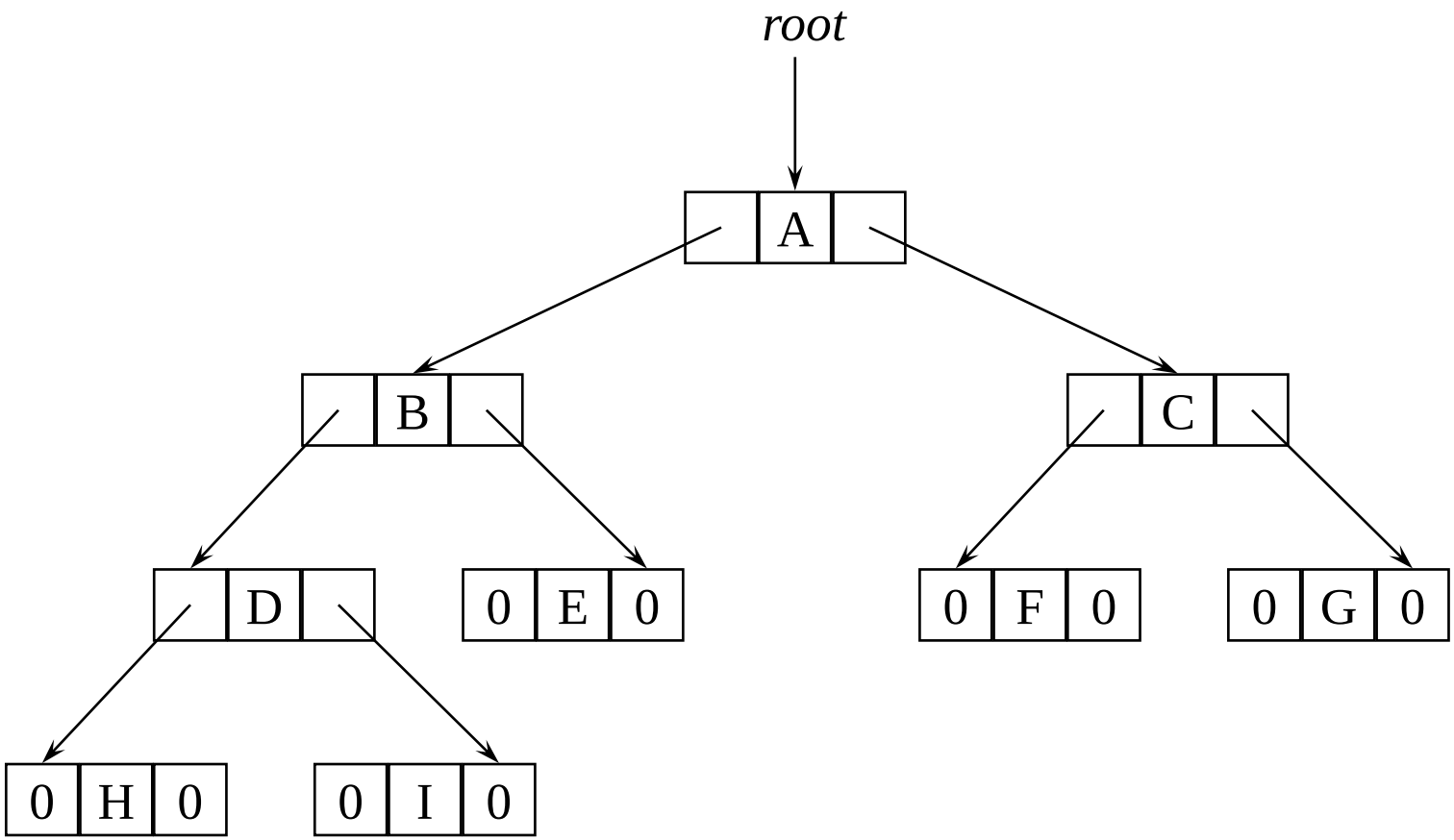
The Satisfiability Problem (5)

◆ 후위 순회 계산시 **Visit**에서 할 일

```
// Visit the node pointed at by p
switch (p->data.first) {
   case Not:  p->data.second = !p->rightChild->data.second; break;
   case And:  p->data.second =
                  p->leftChild->data.second && p->rightChild->data.second;
              break;
   case Or: p->data.second =
                  p->leftChild->data.second || p->rightChild->data.second;
              break;
   case True:  p->data.second = true; break;
   case False:  p->data.second = false; break;
}
```

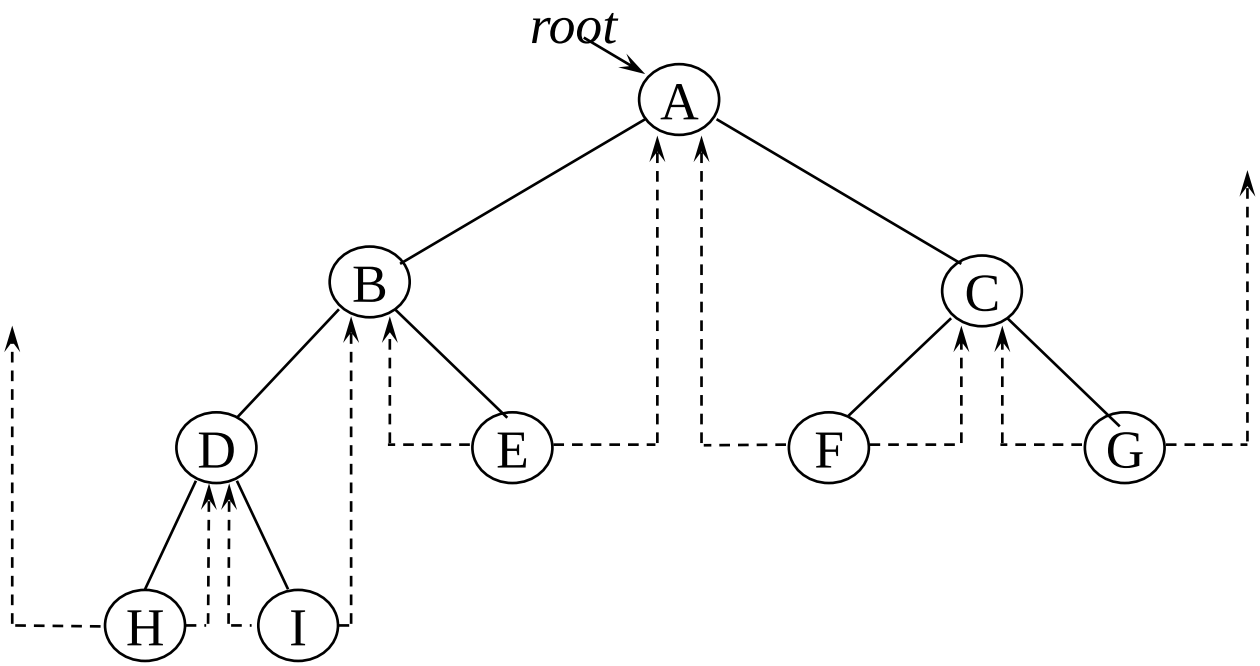
Threaded Binary Trees (1)

- ***n*** 노드 이진 트리의 연결 표현
  - 총 링크의 수 : $2n$
  - 0 링크의 수 : $n+1$



Threaded Binary Trees (2)

- **스레드(Thread)**
  - 0 링크 필드를 다른 노드를 가리키는 포인터로 대치
  - if *p*->*rightChild* == 0,
    p->*rightChild* = *p*의 중위 후속자successor에 대한 포인터
  - if *p*->*leftChild* = 0,
    p->*leftChild* = *p*의 중위 선행자predecessor에 대한 포인터



Threaded Binary Trees (3)

- 노드 구조
  - *lefttThread* == false : *leftChild* ← 포인터
    == true : *leftChild* ← 스레드
  - *rightThread* == false : *rightChild* ← 포인터
    == true : *rightChild* ← 스레드
- 헤드 노드
  - Inorder상 가장 왼쪽노드의 *leftChild* 및 가장 오른쪽 노드의 *rightChild*가 가리키게 함

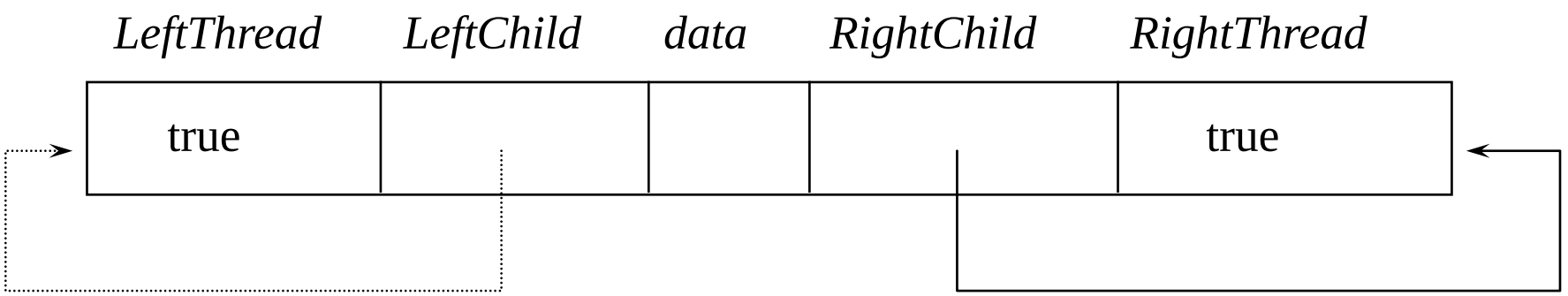
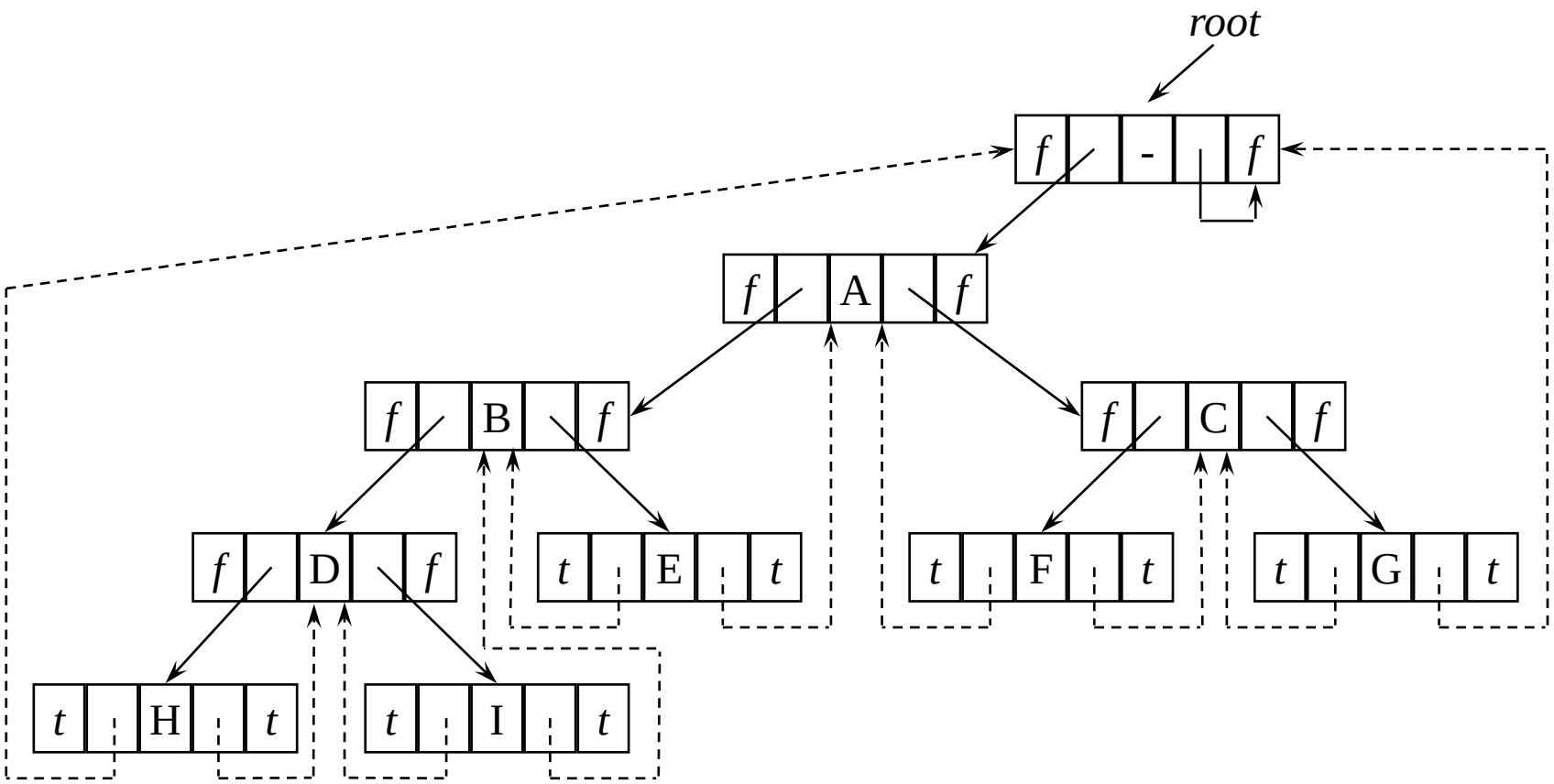


Fig.5.21: An empty threaded binary tree

Memory representation of threaded tree



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

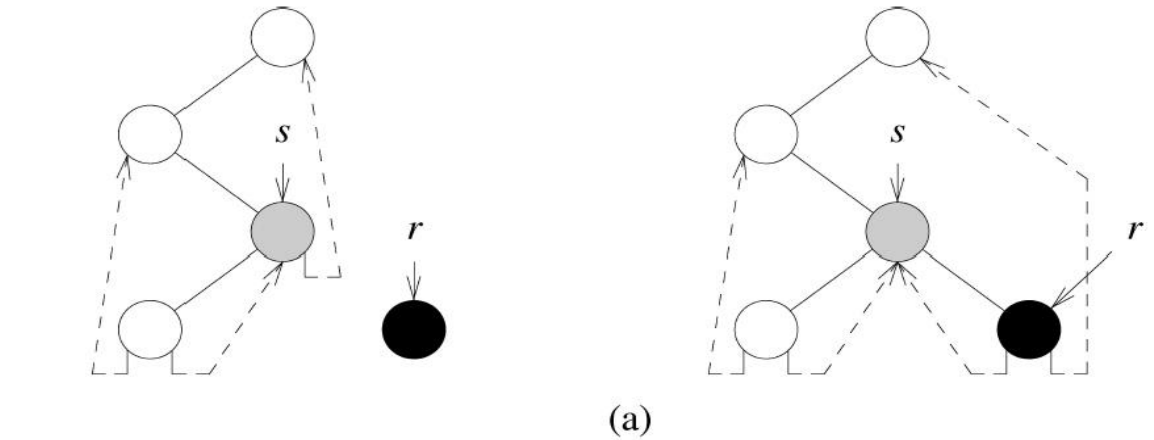
Inorder Traversal of a Threaded Binary Tree

- 스택을 이용하지 않고 중위 순회**(Inorder Traversal)** 가능
- 중위 순회의 후속자**(Inorder successor)**
  - *x->rightThread* == true : *x->rightChild*
    == false : 오른쪽 자식의 왼쪽 자식 링크를 따라 가서 *leftThread*==true인 노드
- 스레드 이진 트리에서 중위 후속자**(Inorder successor)**를 찾는 함수

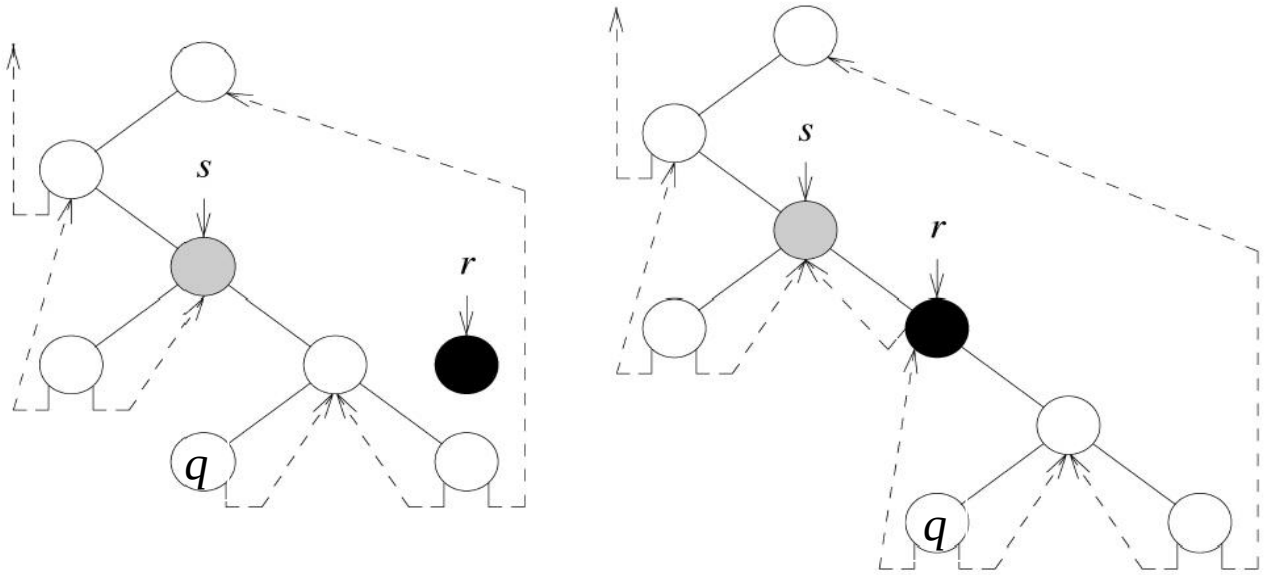
```
T* ThreadedInorderIterator::Next()
{//스레드 이진 트리에서 currentNode의 중위 후속자를 반환
  ThreadedNode<T> *temp = currentNode->rightChild;
  if (!currentNode->rightThread)
    while (!temp->leftThread) temp = temp->leftChild;
  currentNode = temp;
  if (currentNode == root) return 0;
  else return &currentNode->data;
}
```

Inserting a Node into a Threaded Binary Tree (1)

- **Inserting *r* as the right child of a node *s***



(a)



before (b) after

Inserting a Node into a Threaded Binary Tree (2)

◆ **Inserting *r* as the right child of a node *s***

```
template <class T>
void ThreadedTree<T>::InsertRight(ThreadedNode<T> *s,
                                  ThreadedNode<T> *r)
{// Insert r as the right child of s.
    r ->rightChild = s->rightChild;
    r->rightThread = s->rightThread;
    r->leftChild = s;
    r->leftThread = true; // leftChild is a thread
    s->rightChild = r;
    s->rightThread = false;
    if (! r->rightThread) {
        ThreadedNode<T> *temp = InorderSucc( r );
                                  // returns the inorder successor of  r
        temp->leftChild = r;
    }
}
```

◆ **Inserting *r* as the left child of a node *s* : 연습문제**

Priority Queues(우선순위 큐)

- 우선순위가 가장 높은(낮은) 원소를 먼저 삭제
- 임의의 우선순위를 가진 원소 삽입 가능
- 최대 우선순위 큐(**A max priority queue**)

```
template <class T>
class MaxPQ{
public:
  virtual ~MaxPQ(){}
    //가상 파괴자
  virtual bool IsEmpty() const = 0;
    //우선순위 큐가 공백이면 true를 반환
  virtual const T& Top() const =0;
    //최대 원소에 대한 참조를 반환
  virtual void Push(const T&) = 0;
    //우선순위 큐에 원소를 삽입
  virtual void Pop() = 0;
    //최대 우선순위를 가진 원소를 삭제
};
```

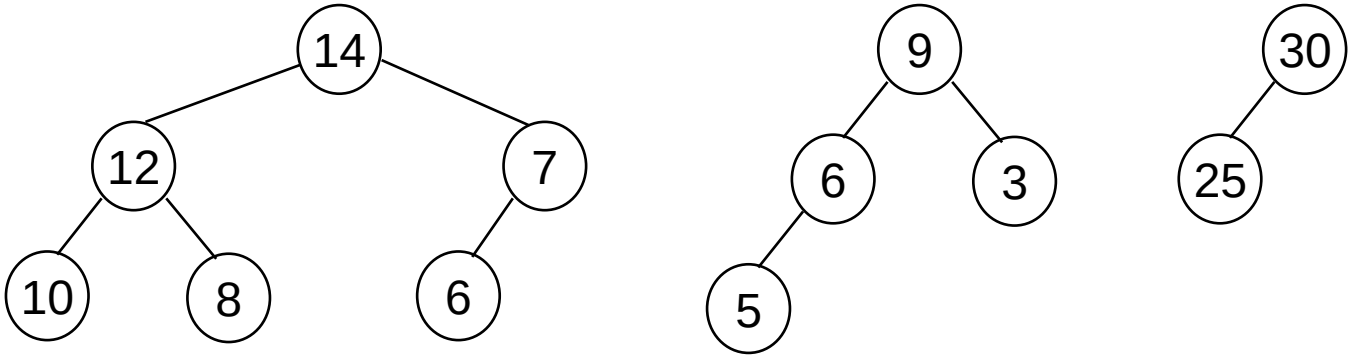
우선순위 큐의 설계

- **표현 방법**
  - 무순서 선형 리스트
    - *IsEmpty*() : O(1)
    - *Push*() : O(1)
    - *Top*() : Θ(*n*)
    - *Pop*() : Θ(*n*)
  - 최대 히프(Max heaps)
    - *IsEmpty*() : O(1)
    - *Top*() : O(1)
    - *Push*() : O(log *n*)
    - *Pop*() : O(log *n*)

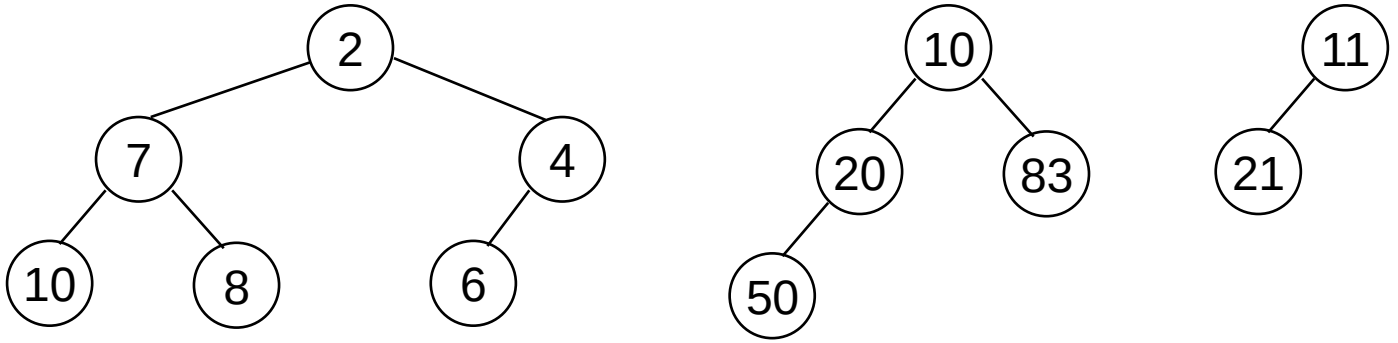
Max Heap(최대 히프)의 정의

- 최대(최소)트리: 각 노드의 키 값이 그 **children**의 키 값보다 작지(크지) 않은 트리.
  (Def.) A *max (min) tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
- 최대히프: 최대 트리이면서 완전 이진 트리.
  (Def.) A *max heap* is a complete binary tree that is also a max tree.
- 최소히프: 최소 트리이면서 완전 이진 트리.
  (Def.) A *min heap* is a complete binary tree that is also a min tree.

최대 히프

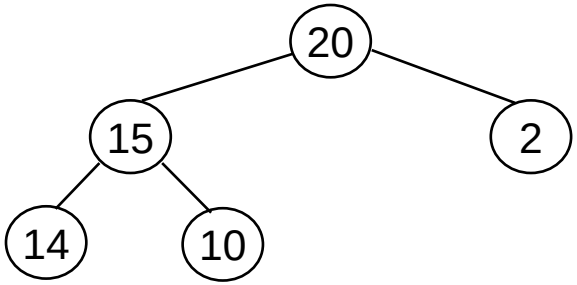


최소 히프

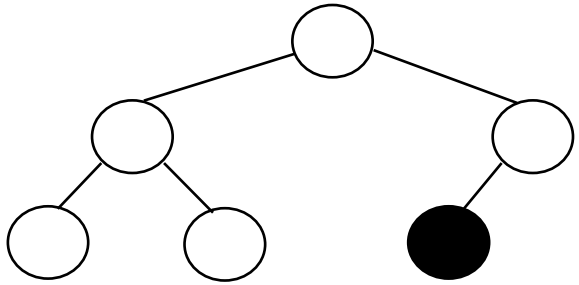


최대 히프에서의 삽입

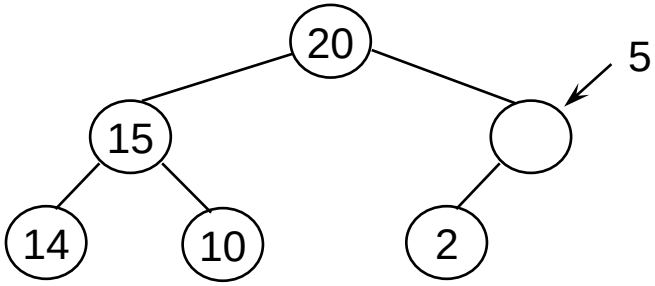
- 삽입 후에도 최대 히프 유지
  - 새로 삽입된 원소는 parent 원소와 비교하면서 최대 히프가 되는 것이 확인될 때까지 위로 올라감



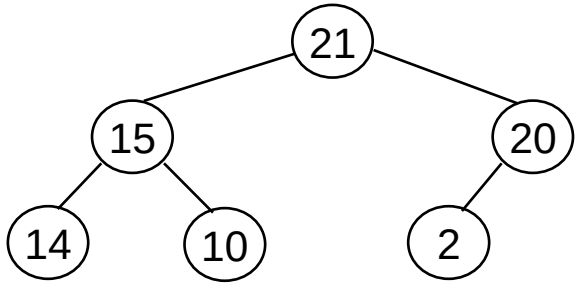
(a)



(b)



(c) <5 삽입 경우>



(d) <21 삽입 경우>

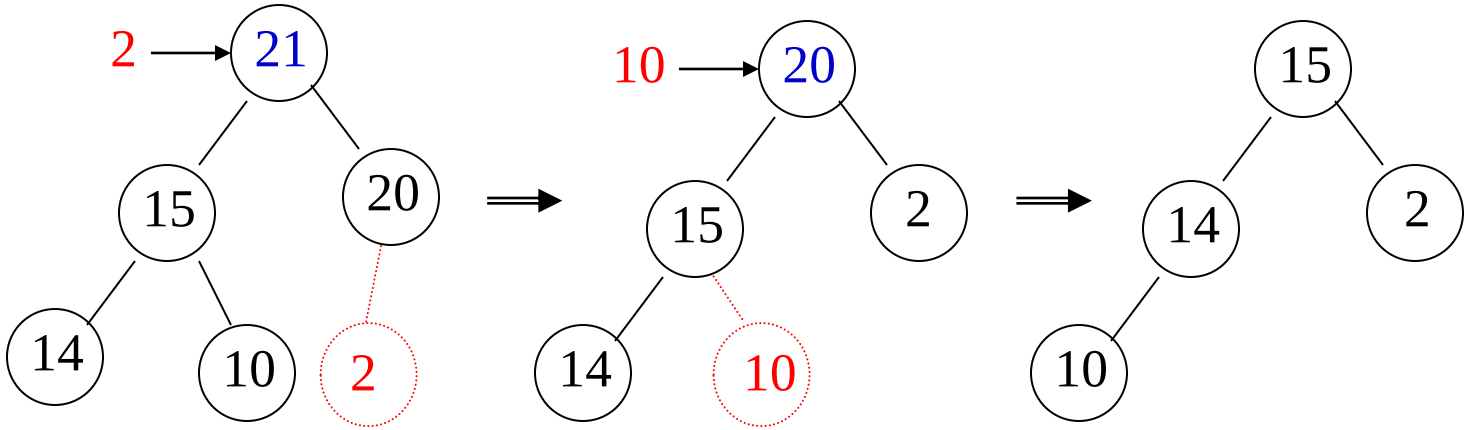
최대 히프에서의 삽입

- 삽입 후에도 최대 히프 유지
  - 새로 삽입된 원소는 parent 원소와 비교하면서 최대 히프가 되는 것이 확인될 때까지 위로 올라감

```
template <class T>
void MaxHeap<T>::Push(const T& e)
{// Insert e into the max heap.
    if (heapSize == capacity) { // 교재에서 capacity는 저장가능한 노드수로 정의되었음
        ChangeSize1D(heap, capacity+1, 2*capacity+1); // 교재 수정요망(1개 더 필요)
        capacity *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < e)
    {// bubble up
        heap[currentNode] = heap[currentNode/2]; // move parent down
        currentNode /= 2;
    }
    heap[currentNode] = e;
}
```

최대 히프에서의 삭제

- **루트 삭제**
  (1)루트삭제 후, 마지막 위치의 노드를 루트 위치로 가져오고 노드수 1 감소
  (2)현재 노드 = 루트노드;
  (3)현재 노드가 현재 노드의 두 child중 큰 것보다도 클 경우 return
  (4) 두 child중 큰 child의 data와 현재 노드를 교환하고, 그 아들 노드를 현재 노드로 간주하여 스텝 (3)으로 가서 반복



<루트값 21을 삭제>　　<루트값 20을 삭제>

최대 히프에서의 삭제

```
template <class T>
void MaxHeap<T>::Pop()
{// Delete max element.
    if (IsEmpty()) throw "heap is empty. Cannot delete.";
    heap[1].~T(); // delete max element
    T lastE = heap[heapSize--]; // remove last element from heap

    // trickle down
    int currentNode = 1; // root
    int child = 2;              // a child of currentNode
    while (child <= heapSize)    // 아들이 있는 경우
    {   // set child to larger child of currentNode
        if (child < heapSize && heap[child] < heap[child+1]) child++;
        if (lastE >= heap[child]) break;  // can we put lastE in currentNode?

         heap[currentNode] = heap[child];  // move child up
         currentNode = child; child *= 2;    // move down a level
    }
    heap[currentNode] = lastE;
}
```

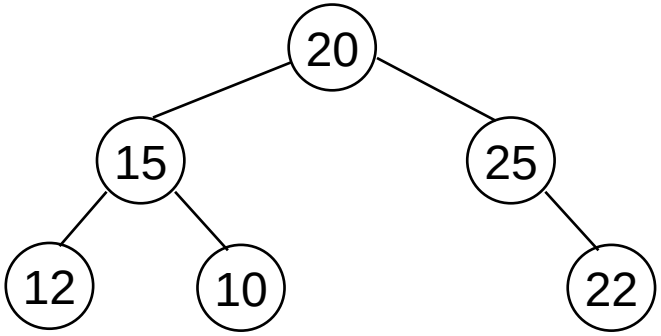
Binary Search Trees(이진 탐색 트리, 이원 탐색트리)

- **사전(dictionary)**
  - pair<키, 원소>의 집합

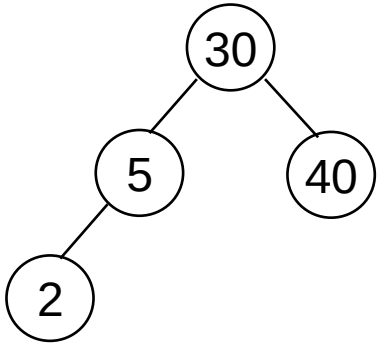
```
template <class K, class E>
class Dictionary{
public:
        virtual bool IsEmpty() const=0;
          //공백이면 true 반환
        virtual pair<K,E>*Get(const K&) const = 0;
          //지시한 키 값을 가진 쌍에 대한 포인터 반환, 쌍이 없으면 0 반환
        virtual void Insert(const pair<K,E>&)=0;
          //쌍을 삽입, 키가 중복되면 관련 원소 갱신
        virtual void Delete(const K&)=0;
          //지시된 키를 가진 쌍 삭제
};
```

Binary Search Trees

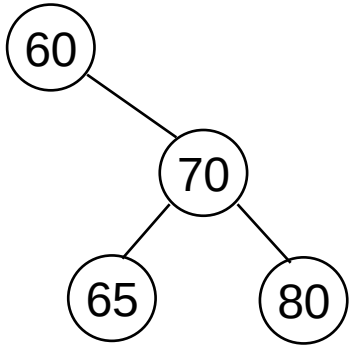
- **Definition:** A *binary search tree* is a binary tree. It may be empty. If it is not empty then it satisfies the following properties.
  (1) 모든 원소는 서로 상이한 키를 갖는다.
  (2) 왼쪽 서브트리의 키들은 그 루트의 키보다 작다.
  (3) 오른쪽 서브트리의 키들은 그 루트의 키보다 크다
  (4) 왼쪽과 오른쪽 서브트리도 이원 탐색 트리이다.
- 이원 탐색 트리



(a) X



(b) O



(c) O

Searching a Binary Search Tree

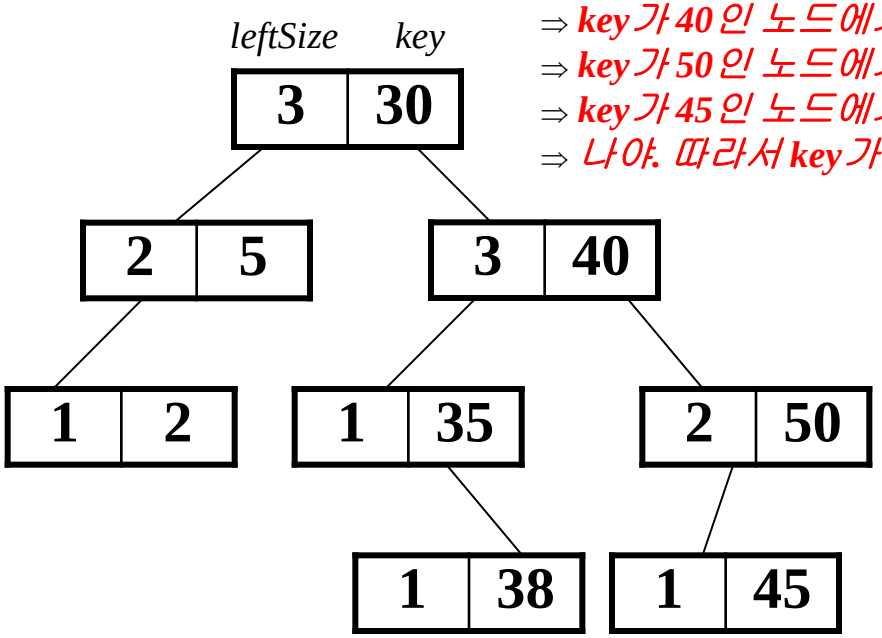
- ***k*** = 루트의 키 : 성공적 종료
- ***k*** < 루트의 키 : 왼쪽 서브트리 탐색
- ***k*** > 루트의 키 : 오른쪽 서브트리 탐색

```
template <class K, class E> //Driver
pair<K,E>* BST<K,E>::Get(const K& k)
{//키 k를 가진 쌍을 이원 탐색 트리(*this)에서 탐색
// 쌍을 발견하면 포인터 반환, 아니면 0 반환
   return Get(root, k);
}

template <class K, class E> //Workhorse
pair<K,E>* BST<K,E>::Get(TreeNode <pair <K,E> >*p, const K& k)
{
   if (!p) return 0;
   if (k < p->data.first) return Get(p->leftChild,k);
   if (k > p->data.first) return Get(p->rightChild,k);
   return &p->data;
}
```

>와 > 사이에 빈칸 삽입 필요!!

순위에 의한 이진 탐색 트리의 탐색

- **순위(rank)**
  - 중위 순서(inorder)에서의 위치
  - *leftSize* = 왼쪽 서브 트리의 원소 수 + 1
    - 나를 루트로 하는 트리에서는 내가 *leftSize* 등이야!
- **순위에 의한 이진 탐색 트리의 탐색의 동작 예(7등이 누구니?)**



key가 30인 노드에게 너를 루트로하는 트리에서 7등이 누구니?
⇒ *key가 40인 노드에게 너를 루트로하는 트리에서 4(7-3)등이 누구니?*
⇒ *key가 50인 노드에게 너를 루트로 하는 트리에서 1등이 누구니?*
⇒ *key가 45인 노드에게 너를 루트로 하는 트리에서 1등이 누구니?*
⇒ *나야. 따라서 key가 45인 노드의 data를 반환한다.*

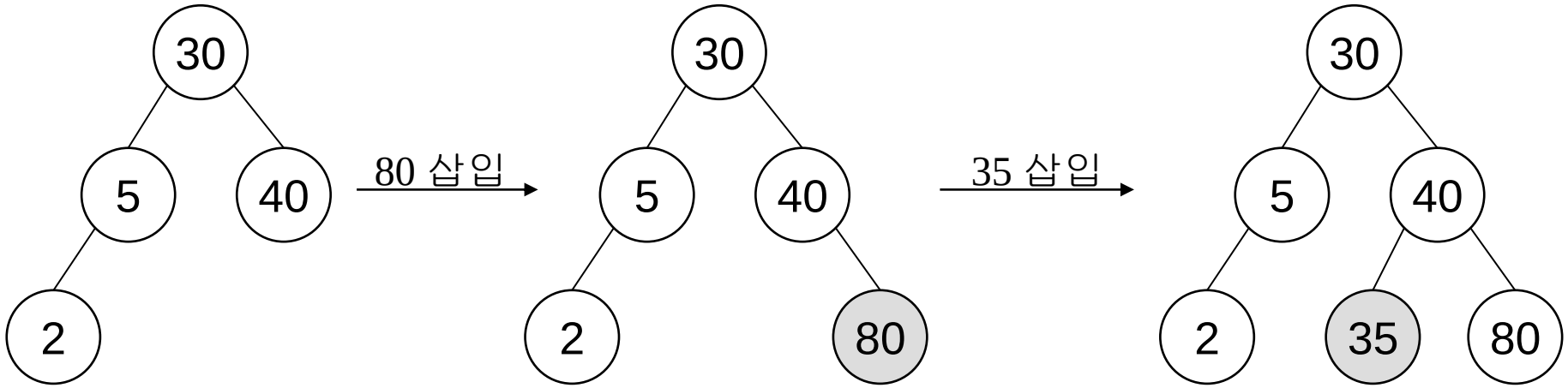
순위에 의한 이진 탐색 트리의 탐색

◆ 순위에 의한 이진 탐색 트리의 탐색 알고리즘

```
template <class K, class E> //순위에 의한 탐색
pair<K,E>* BST<K,E>::RankGet(int r) // r등을 찾아 return
{//r번째 작은 쌍을 탐색한다.
  TreeNode<pair<K,E> > *currentNode = root; // > > 사이 빈칸 !!
  while (currentNode)
    if ( r < currentNode->leftSize )
       currentNode = currentNode->leftChild;
    else if  (r > currentNode->leftSize ) {
       r  -= currentNode->leftSize;
       currentNode = currentNode->rightChild;
    }
    else return &currentNode->data;
  return 0;
}
```

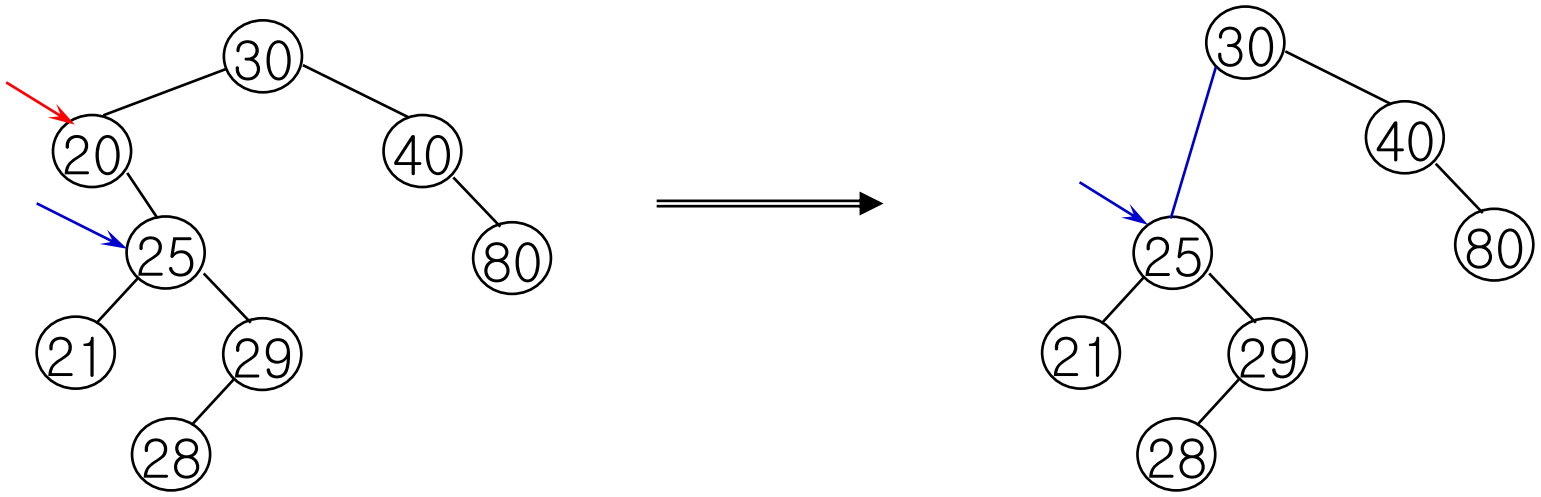
이진 탐색 트리에서의 삽입

- ***x*의 key값을 가진 노드를 탐색**
- 탐색이 성공하면 이 키에 연관된 원소를 변경
- 탐색이 실패하면 탐색이 끝난 지점에 **pair**를 삽입



이진 탐색 트리에서의 삭제

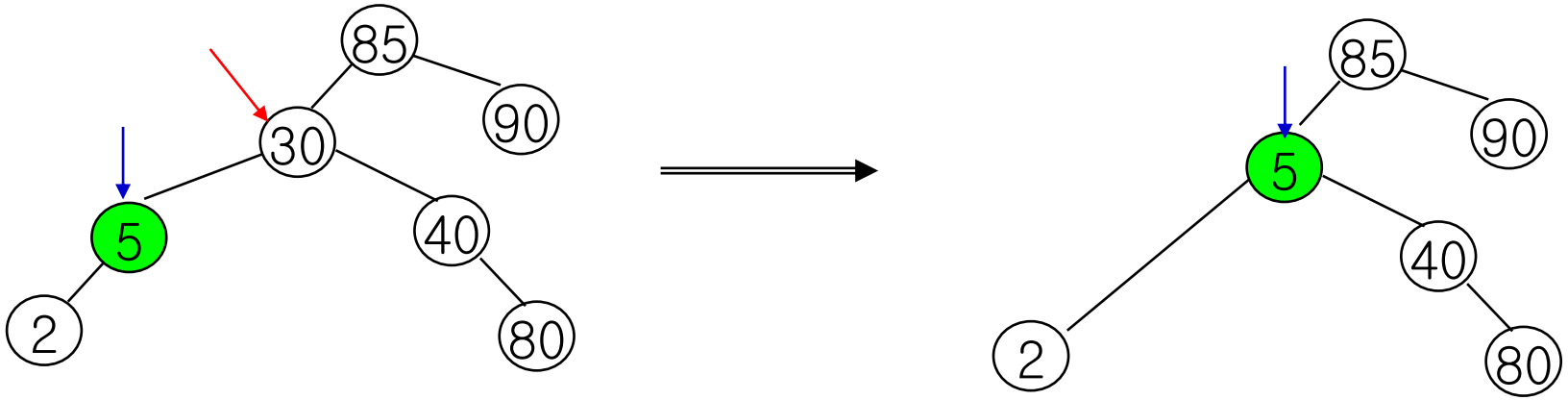
- **리프 원소(leaf node)의 삭제**
  - 삭제할 노드의 포인터(아빠 노드의 해당 자식 필드) 에 0을 삽입
  - 삭제할 노드를 heap 영역에 반납
- **하나의 아들(child)을 가진 비리프 노드(non-leaf node)의 삭제**
  - 삭제할 노드의 그 유일한 아들을 삭제할 노드의 아빠노드의 아들로 삼음(삭제할 노드가 왼쪽아들이었으면 왼쪽 아들로, 오른쪽 아들이었으면 오른쪽 아들로)
  - 삭제할 노드를 heap 영역에 반납



[problem] 아들이 하나인 노드 20을 삭제하기
(a) 20이 그 아빠 30의 **왼쪽** 아들임에 유의함
(b) 20의 그 유일한 아들(25)을 30의 **왼쪽** 아들로 삼음
(c) 20을 가진 노드를 heap영역으로 반납

이진 탐색 트리에서의 삭제

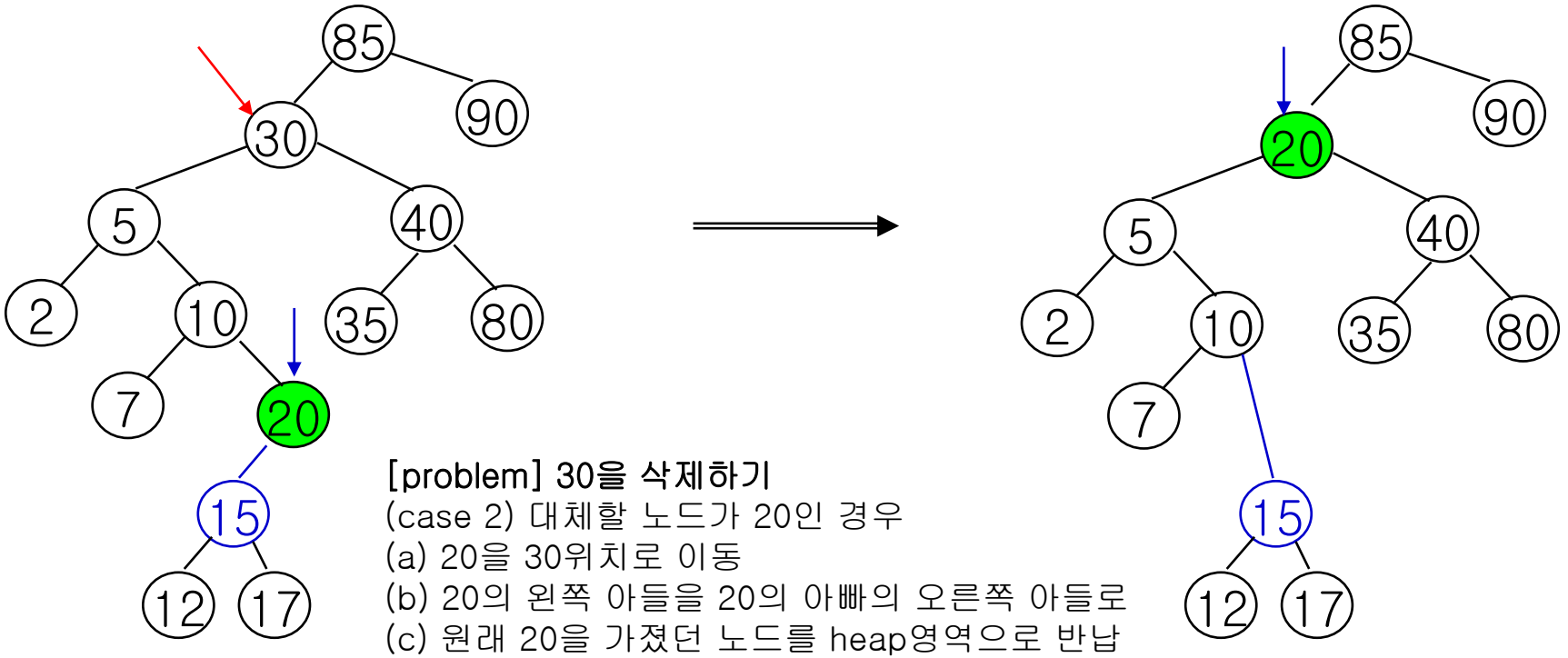
- 아들이 둘인 **Non-leaf node(x라 하자)**의 삭제
  - 왼쪽 서브트리에서 가장 큰 원소나 오른쪽 서브트리에서 가장 작은 원소로 대체가능
  - 여기서는 왼쪽 서브트리에서 가장 큰 원소로 설명하자.
  - (case 1) x의 왼쪽 아들(y)이 바로 대체할 아들인 경우
    - y의 키와 자료를 x로 복사한다.
    - y의 왼쪽 아들을 x의 왼쪽 아들로 삼는다.
    - 원래 y노드는 heap영역으로 반납한다.



[problem] 30을 삭제하기
(case 1) 대체할 노드가 바로 왼쪽 아들 5인 경우
(a) 5를 30위치로 이동
(b) 5의 왼쪽 아들을 지울려는 노드의 왼쪽 아들로
(c) 원래 5를 가졌던 노드를 heap영역으로 반납

이진 탐색 트리에서의 삭제

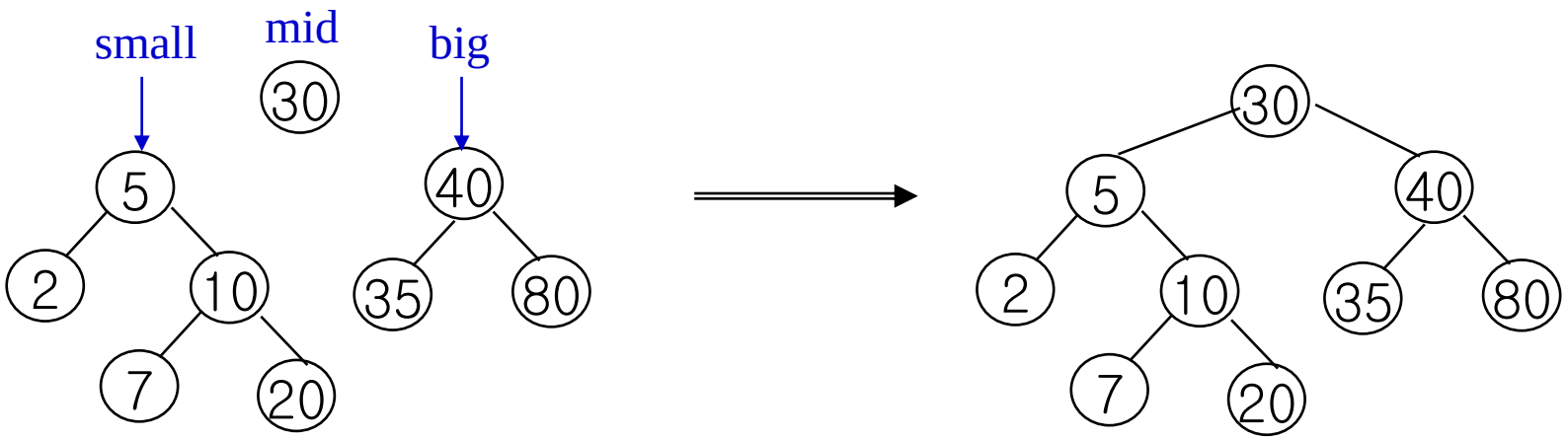
- **2 아들 가진 Non-leaf node(x라 하자)의 삭제:시간 복잡도 = $O(h)$**
  - (case 2) x의 왼쪽 아들의 오른쪽을 끝까지 찾아가 대체할 노드(y) 찾은 경우
    - y의 키와 자료를 x로 복사한다.
    - y의 왼쪽 아들을 y의 아빠의 오른쪽 아들로 삼는다.
    - 원래 y노드는 heap영역으로 반납한다.
    - [주의] y의 아빠를 알 수 있도록 코딩하는 게 중요



[problem] 30을 삭제하기
(case 2) 대체할 노드가 20인 경우
(a) 20을 30위치로 이동
(b) 20의 왼쪽 아들을 20의 아빠의 오른쪽 아들로
(c) 원래 20을 가졌던 노드를 heap영역으로 반납

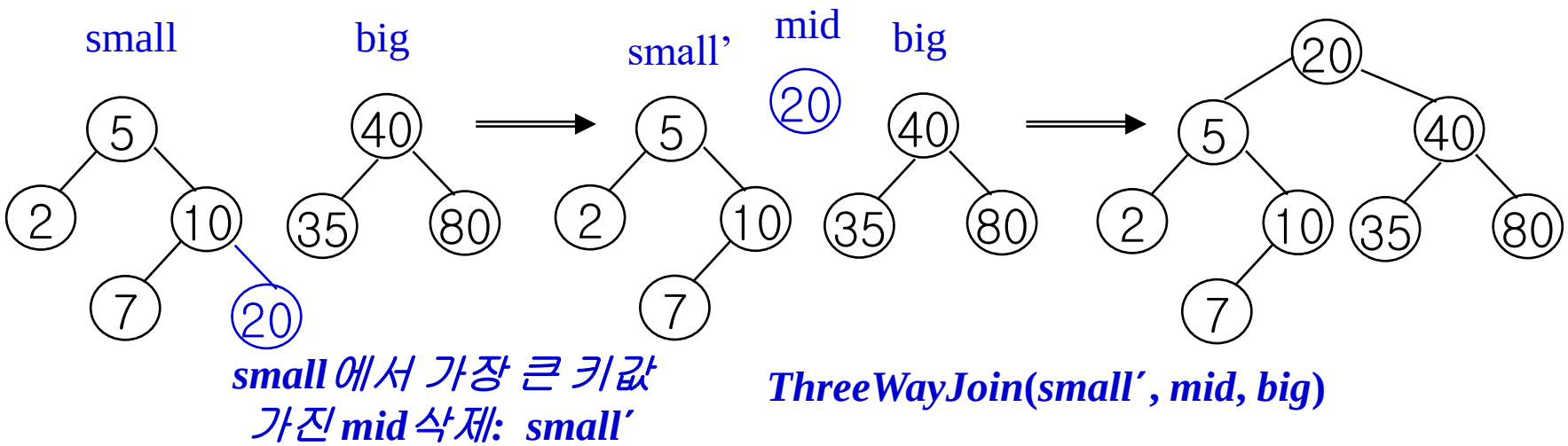
이진 탐색 트리의 조인과 분할 (1)

- ***ThreeWayJoin*(*small*, *mid*, *big*): 동작시간 O(1)**
  - 두개의 이진 탐색 트리 *small*과 *big*에 있는 pair들과 pair *mid*로 구성되는 이진 탐색 트리를 생성한다.
  - [가정]
    - *small*에 있는 각 key는 *big*에 있는 각 *key*보다 작다.
    - *mid*의 *key*(즉 *mid.first*)는 *small*의 어떤 *key*보다도 크고, *big*의 어떤 key보다도 작다.
    - *Join*이 끝나면 *small*과 *big*은 empty 트리가 된다.
  - [구현]
    - mid를 data로, small을 leftChild로, big을 rightChild로 하는 새 노드를 만든 다음, 그 노드를 *this의 root로 만든다.

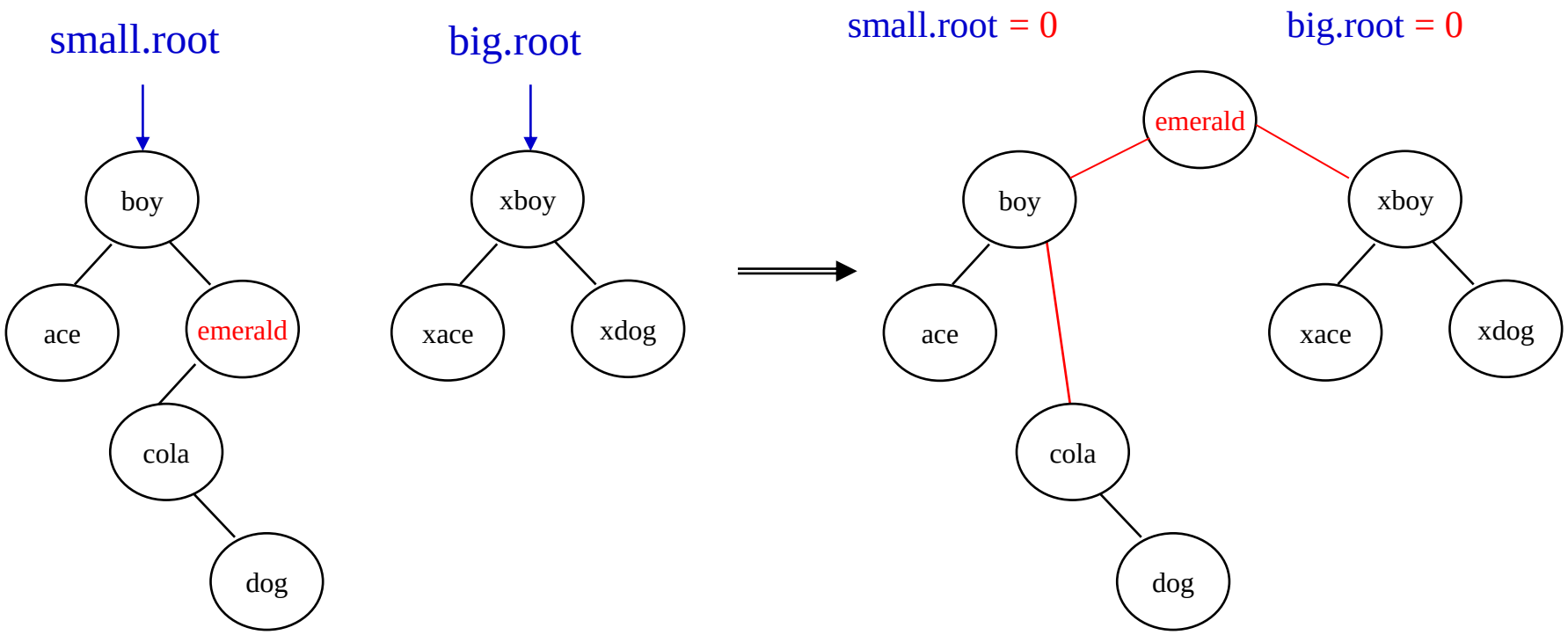


이진 탐색 트리의 조인과 분할 (2)

- ***TwoWayJoin*(*small*, *big*): 동작시간 O(*height*(*small*))**
  - 두개의 이진 탐색 트리 *small*과 *big*에 있는 pair들로 구성되는 이진 탐색 트리를 생성한다.
  - [가정]
    - *small*에 있는 각 key는 *big*에 있는 각 *key*보다 작다.
    - *Join*이 끝나면 *small*과 *big*은 empty 트리가 된다.
  - [구현]
    - *small* 또는 *big*중 하나가 empty일 경우: non-empty 트리를 반환
    - 둘 다 non-empty 트리일 경우:
      - *small*에서 가장 큰 키 값을 가진 *mid* 쌍 삭제 : *small´*
      - *ThreeWayJoin*(*small´*, *mid*, *big*) 수행



이진 탐색 트리의 조인과 분할 (3)



TwoWayJoin(small, big)

이진 탐색 트리의 조인과 분할 (4)

- ***Split*(*k*, *small*, *mid*, *big*)**
  - 이진 탐색 트리 *this를 세 부분으로 분할
  - *small*은 *this에서 키 값이 *k*보다 작은 모든 pair로 구성된 트리
  - *mid*는 *this의 원소 중 키 값이 *k*인 pair
  - *big*은 *this에서 키 값이 *k*보다 큰 모든 pair로 구성된 트리
  - [구현]: 자세한 내용은 See program 5.22
  - *k* = *root->data.first*인 경우
    - *small* = *this의 왼쪽 서브트리
    - *mid* = 루트에 있는 쌍
    - *big* = *this의 오른쪽 서브 트리
  - *k* < *root->data.first*인 경우
    - *big* ⊇ 루트, 오른쪽 서브트리
  - *k* > *root->data.first*
    - *small* ⊇ 루트, 왼쪽 서브트리

이진 탐색 트리의 높이

- 이진 탐색 트리의 원소 수 : ***n***

- **이진 탐색 트리에서 탐색 속도 = 트리의 높이(height)**
- **최악의 트리**
  - 키 [1,2,3,...,*n*]을 순서대로 삽입한 경우에 트리의 높이 = *n*
- **최상의 트리: 균형 탐색 트리(balanced search tree)**
  - 최악의 경우에도 height = O(log *n*)이 되는 트리
  - 탐색, 삽입, 삭제의 시간 복잡도 : O(*h*)
  - AVL, 2-3, 2-3-4, 레드-블랙(red-black), B 트리, B+ 트리 등
- **random 삽입, 삭제 시**
  - 평균이진 탐색 트리의 높이 = O(log *n*)
  - 최악의 경우 = O(*n*) 일 가능성도 있음

승자 트리(*WinnerTree*) (1)

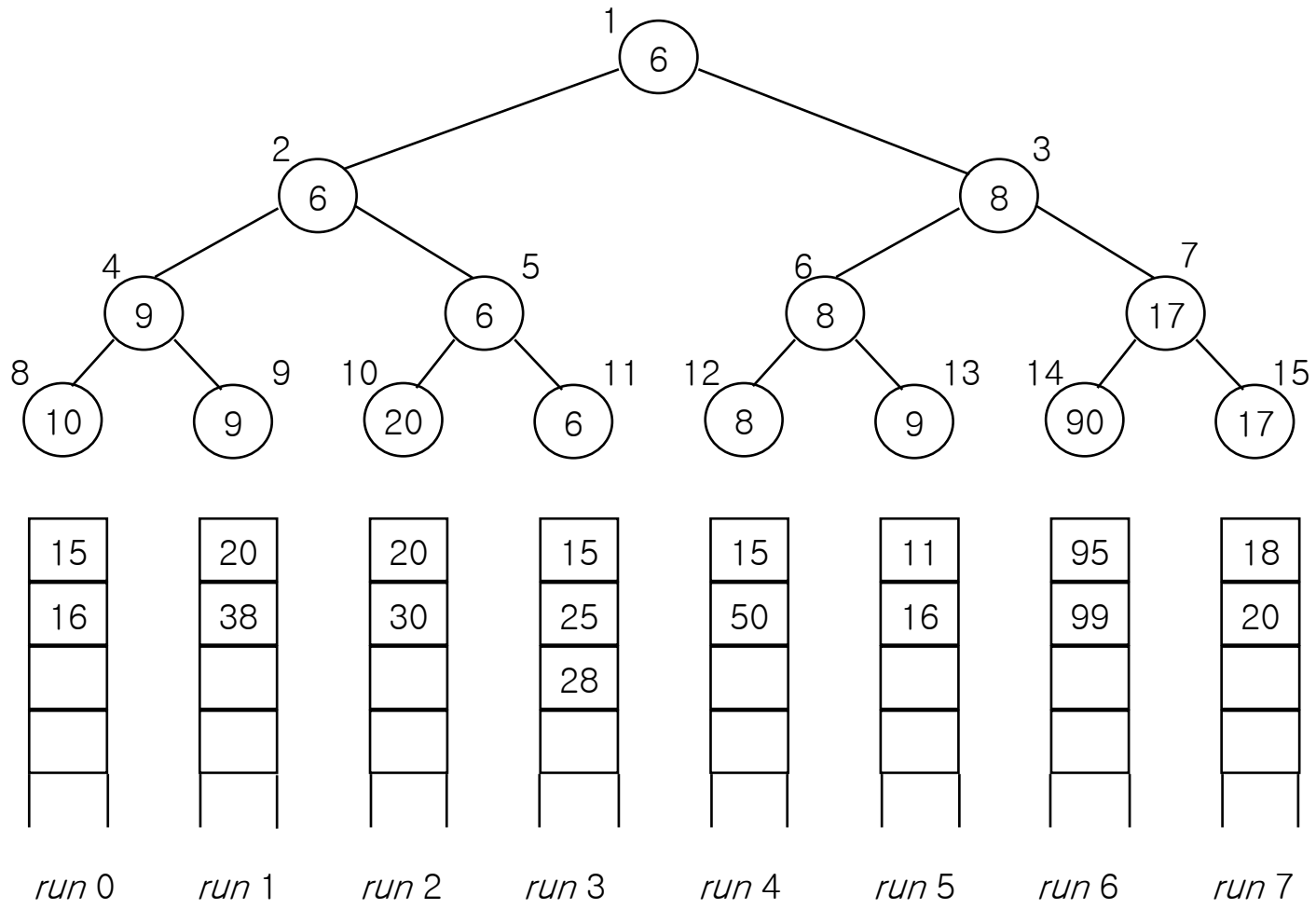
- ***run* = ordered sequence**
  - 각 run은 record들로 구성되며, key의 nondecreasing order임.
- **다수의 *run*들의 합병 (*run*의 수=*k*, 총 record수=*n*)**
  - How? By repeatedly outputting the record with the smallest key.
  - The most direct way : *k*-1번 비교로 다음 출력할 record선택.
  - For $k > 2$, 비교 수를 줄이기 위해 winner trees and loser trees

- **승자 트리(*winner tree*)**
  - 각 노드가 두 개의 자식 노드 중 더 작은 노드를 나타내는 완전 이진 트리
  - 루트 노드 : 트리에서 가장 작은 노드
  - 리프 노드 : 각 런(run)의 첫 번째 레코드
  - 이진 트리에 대한 순차 할당 기법으로 표현

승자 트리 (2)

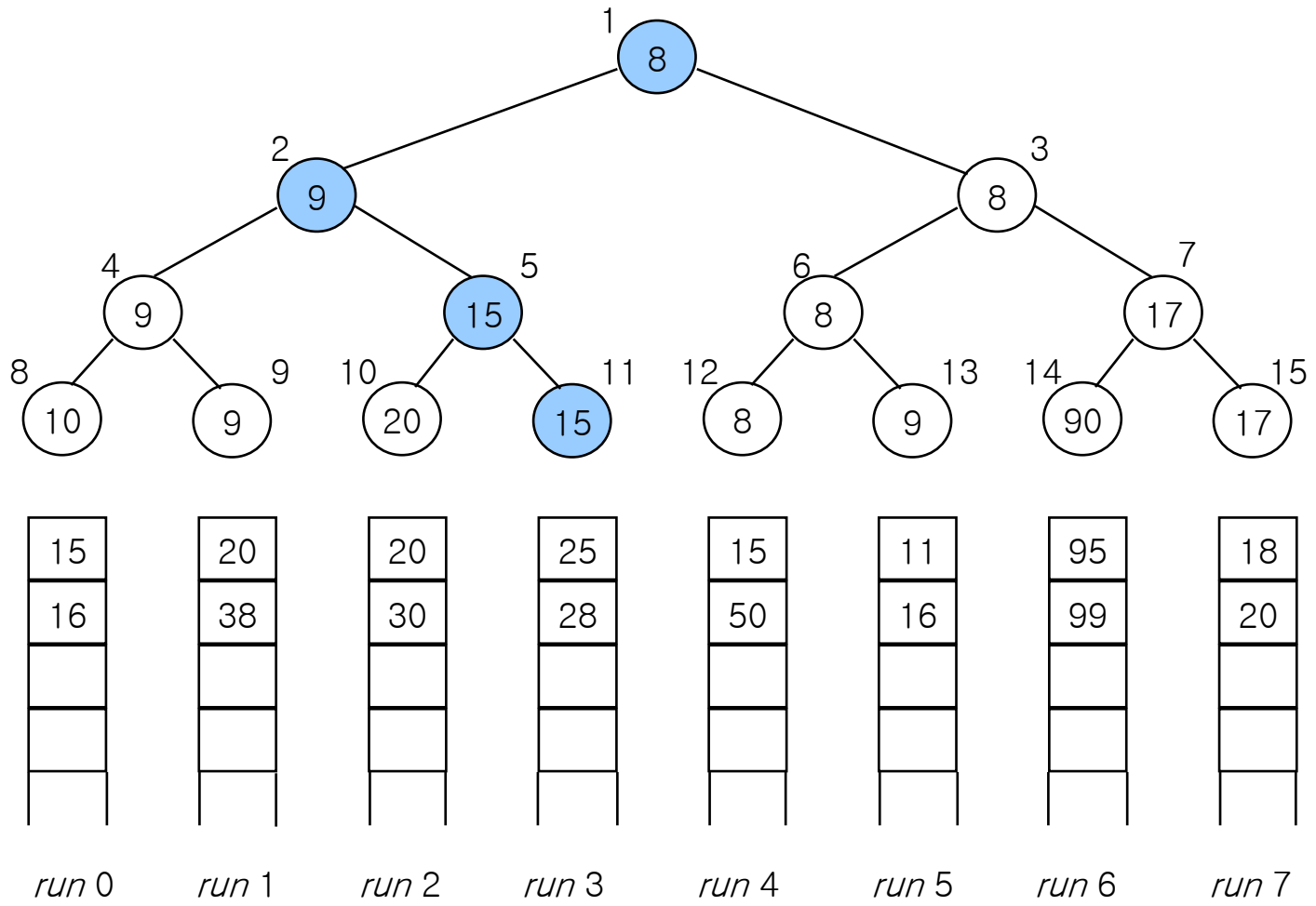
- *k*=8인 경우의 승자 트리

  [구현 시 주의] Winner tree의 각 노드에는 실제로는 run번호를 저장함



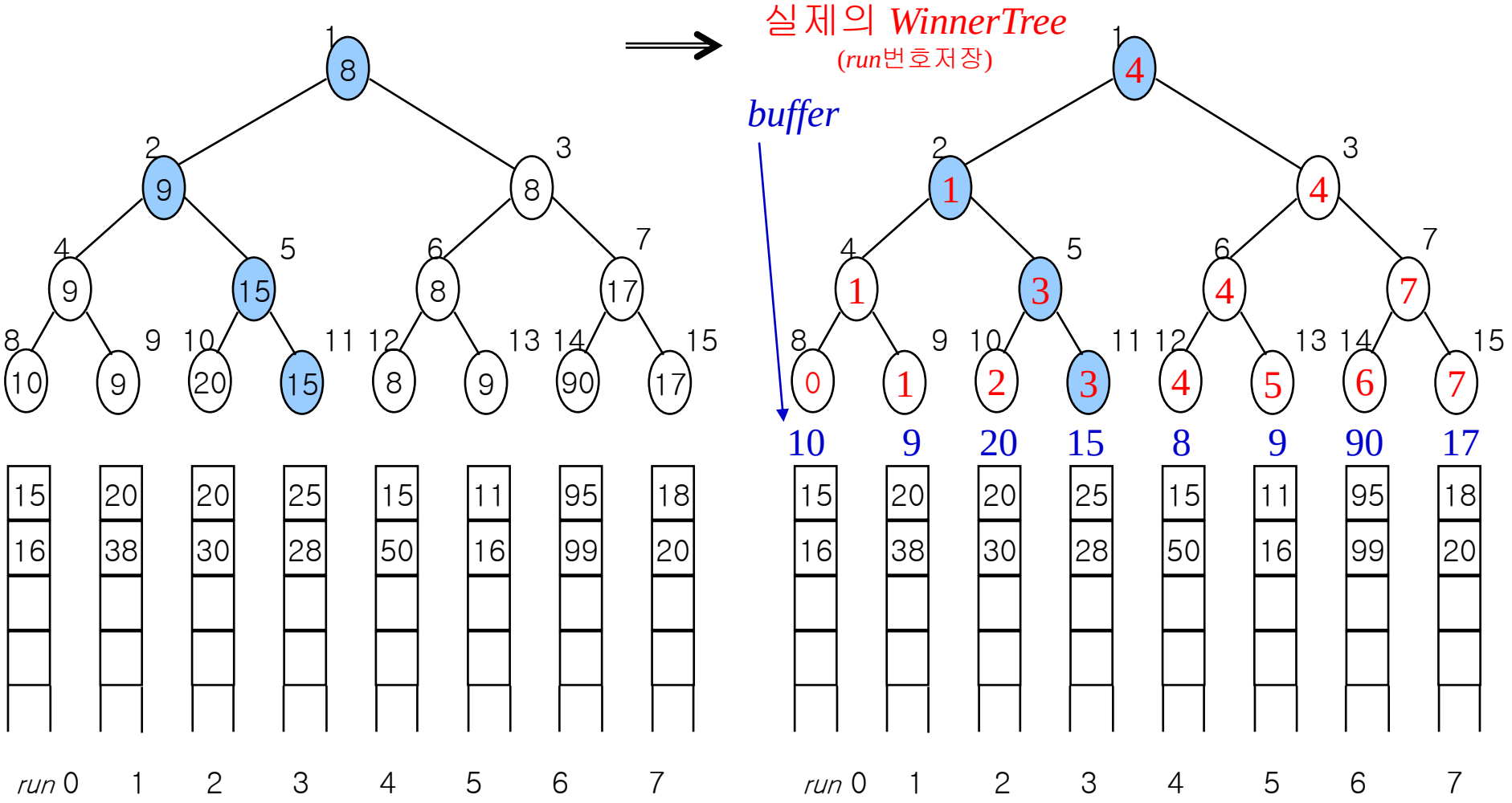
승자 트리 (3)

- 승자 트리에서 한 레코드가 출력되고 나서 재구성
  - 새로 삽입된 노드에서 루트까지의 경로를 따라 토너먼트 재수행



승자 트리 (4)

- 실제 구현시 *Winnertree*는 *run*번호 저장하는 배열
- 각 입력 *run* 의 자료는 *buffer*[*run*]에

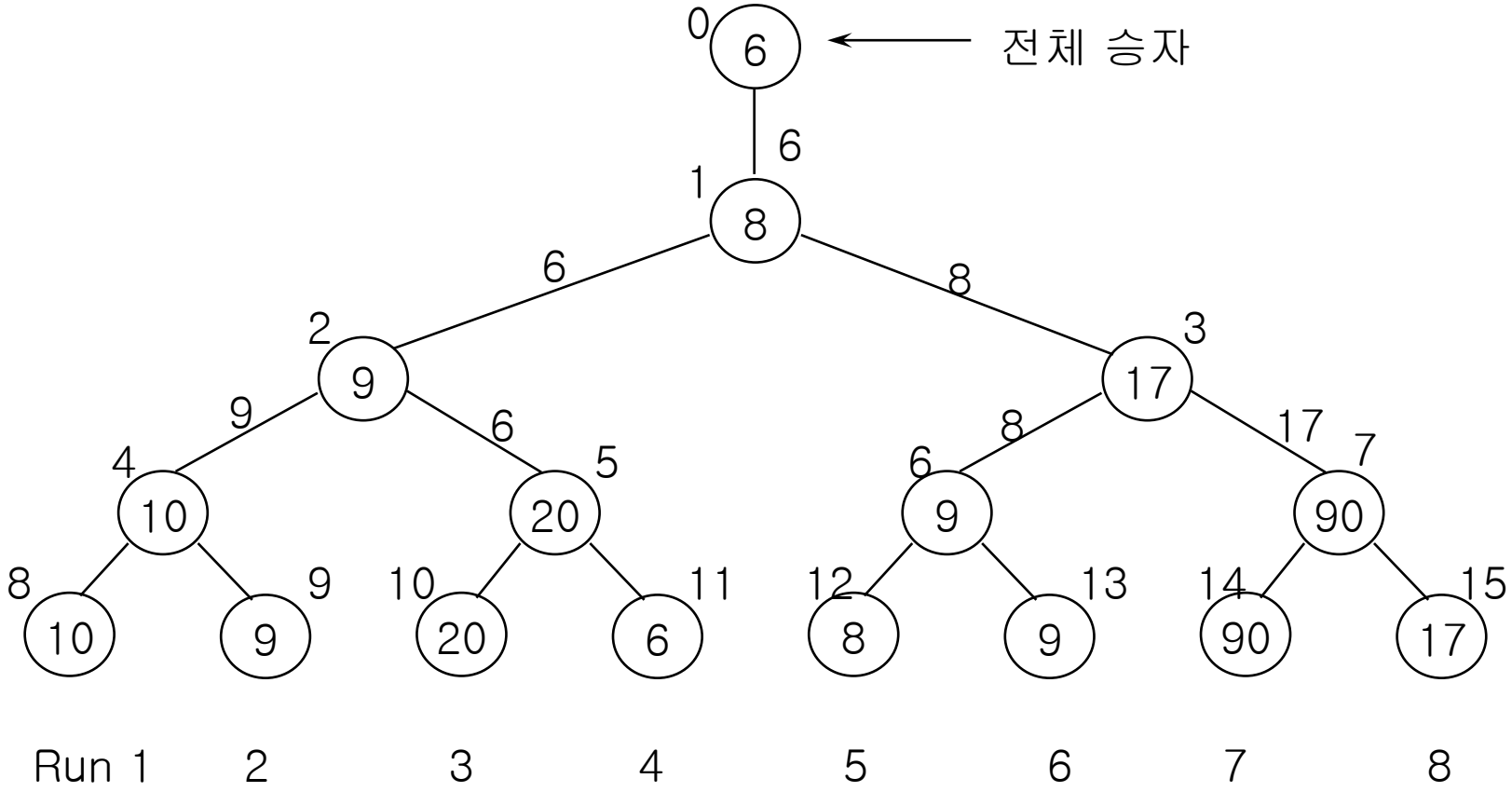


패자 트리(LoserTree) (1)

- **최소키 레코드 출력 후 winner tree의 재구성시**
  - 출력된 노드가 속한 *run*에서 *root*까지의 경로 상에서 sibling node들간의 tournament가 발생한다.
  - 이 sibling node들은 종전 tournament에서의 loser들이다.

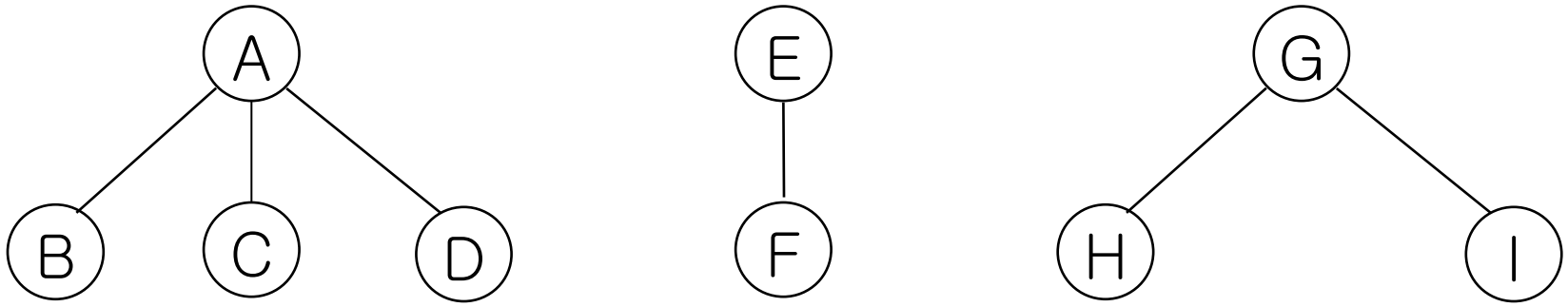
- **패자 트리(loser tree)**
  - 비리프 노드에 그 children들간의 싸움에서의 loser를 남기고 winner는 계속 올라가면서 비교한다.
  - 전체 토너먼트의 승자를 표현하기 위한 노드 0 첨가
  - sibling node(형제 노드)에는 접근 하지 않음

패자 트리 (2)


0
6
전체 승자
6
1
8
6
8
2
9
3
17
9
6
8
17
7
4
10
5
20
6
9
90
8
10
9
10
20
11
6
12
8
13
9
14
90
15
17
Run 1
2
3
4
5
6
7
8

포리스트

- **포리스트(forest)**
  - *n*(≥0)개의 분리(disjoint) 트리들의 집합



포리스트를 이진 트리로 변환 (1)

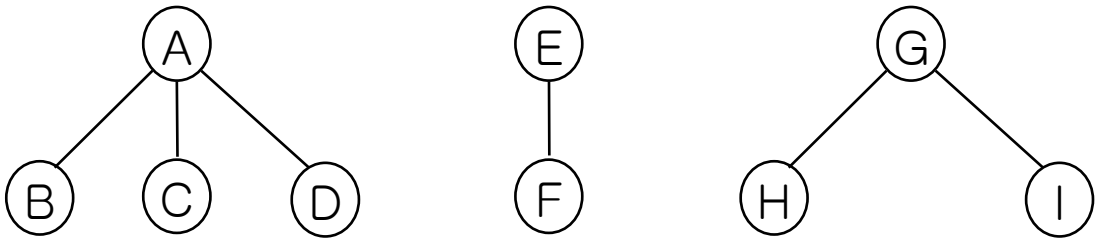
- 각 트리를 이진 트리로 변환
- 변환된 모든 이진 트리들을 루트의 ***rightChild***로 연결

- 정의
  - 만일 $T_1, T_2, \ldots, T_n$이 트리로 된 포리스트라 하면 이 포리스트에 대응하는 이진 트리, $B(T_1, T_2, \ldots, T_n)$은
  - n=0이면 공백
  - B 의 루트 = T_1의 루트
    왼쪽 서브 트리 = $B(T_{11}, T_{12}, \ldots, T_{1m})$
    - $T_{11}, T_{12}, \ldots, T_{1m}$는 T_1의 루트의 서브트리

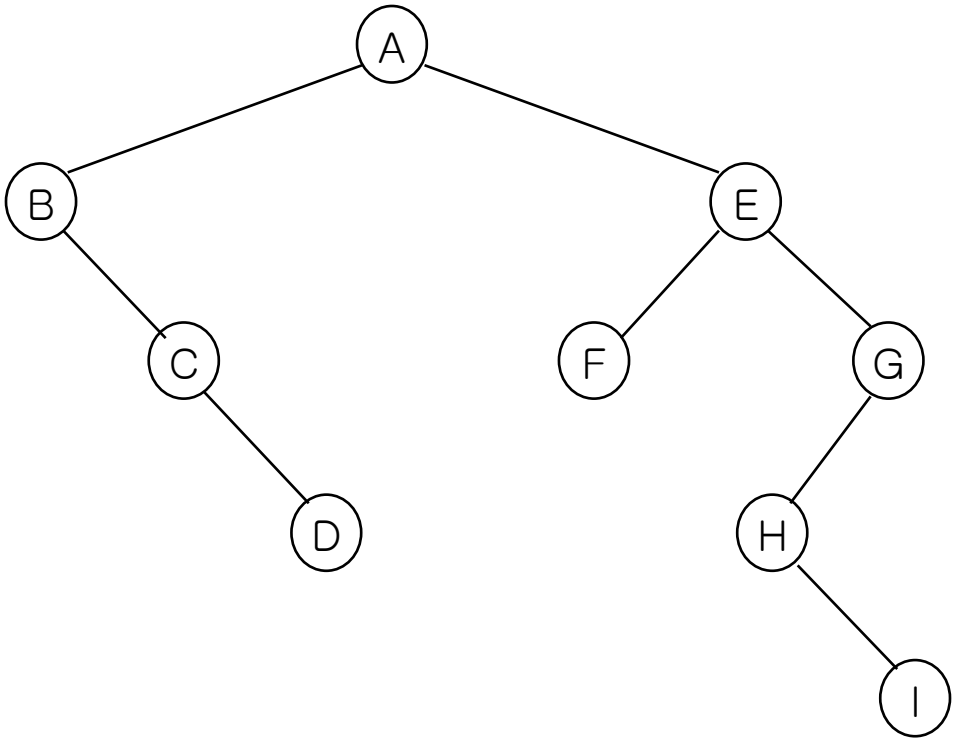
    오른쪽 서브트리 = $B(T_2, \ldots, T_n)$

포리스트를 이진 트리로 변환 (2)

- 세 개의 트리로 구성된 포리스트



- 위 그림의 이진 트리 표현



포리스트 순회 (1)

- **포리스트 전위(forest preorder)**
  - F가 공백이면 return
  - F의 첫 번째 트리의 루트 방문
  - 첫 번째 트리의 서브트리들을 포리스트 전위 순회
  - F의 나머지 트리들을 포리스트 전위로 순회
- **포리스트 중위(forest inorder)**
  - F가 공백이면 return
  - F의 첫 번째 트리의 서브트리를 포리스트 중위로 순회
  - 첫 번째 트리의 루트 방문
  - 나머지 트리들을 포리스트 중위로 순회

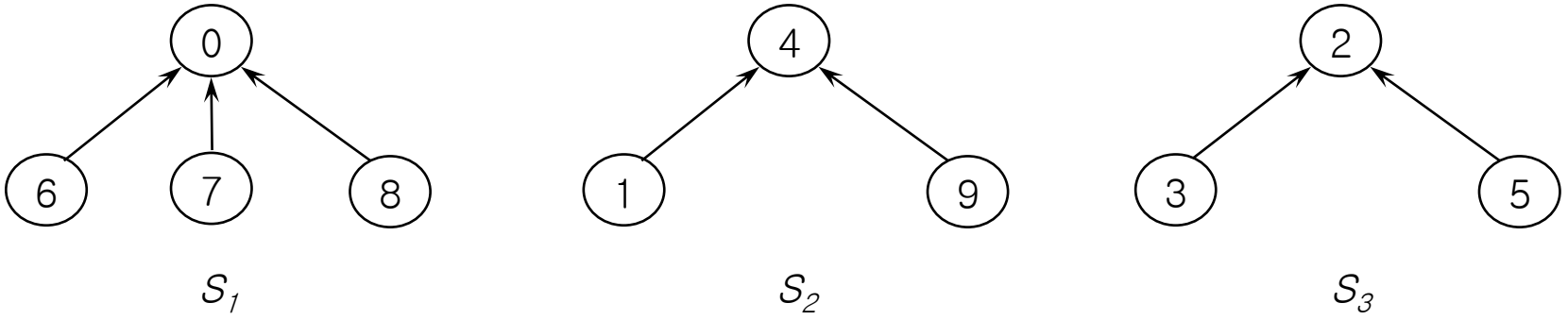
포리스트 순회(2)

- **포리스트 후위(forest postorder)**
  - F가 공백이면 return
  - F의 첫 트리의 서브트리를 포리스트 후위로 순회
  - 나머지 트리들을 포리스트 후위로 순회
  - F의 첫 트리의 루트를 방문

- **포리스트 레벨 순서 순회**
  - 포리스트의 각 루트부터 시작하여 노드를 레벨 순으로 방문
  - 레벨 내에서는 왼쪽에서부터 오른쪽으로 차례로 방문

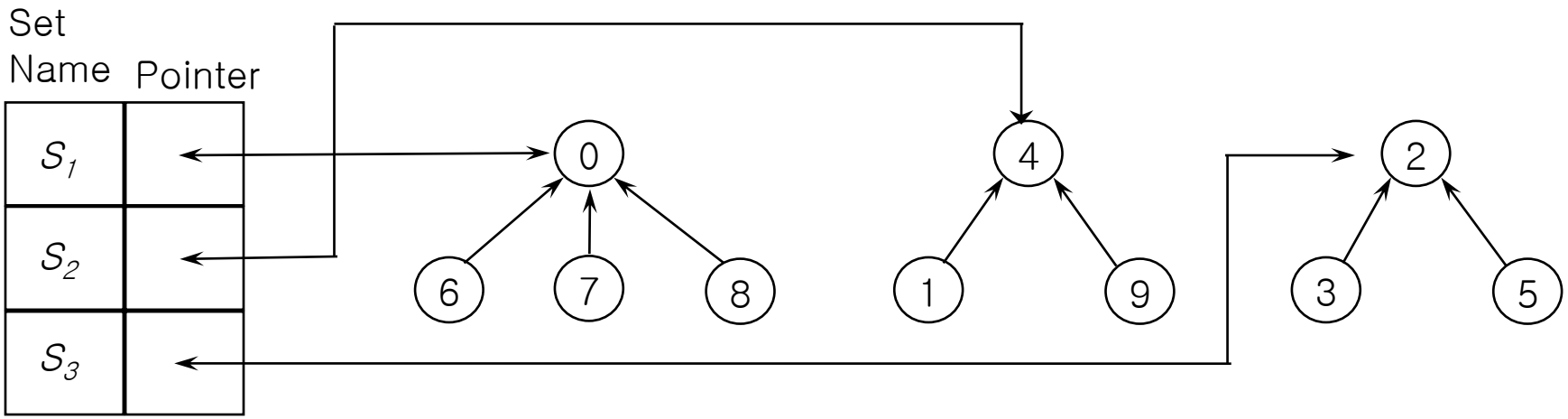
Disjoint Set의 표현 (1)

- **disjoint set의 트리 표현**
  - 집합의 모든 원소는 수 0,1,2,...,n-1이라고 가정
  - 모든 집합들은 pairwise disjoint라고 가정
    - 즉 i != j일 때 두 집합 S_i와 S_j가 공유한 원소가 없다고 가정
  - 자식(child)에서부터 부모(parent)로 가는 링크로 연결



Disjoint Set의 표현 (2)

◆ S_1, S_2, S_3의 데이타 표현



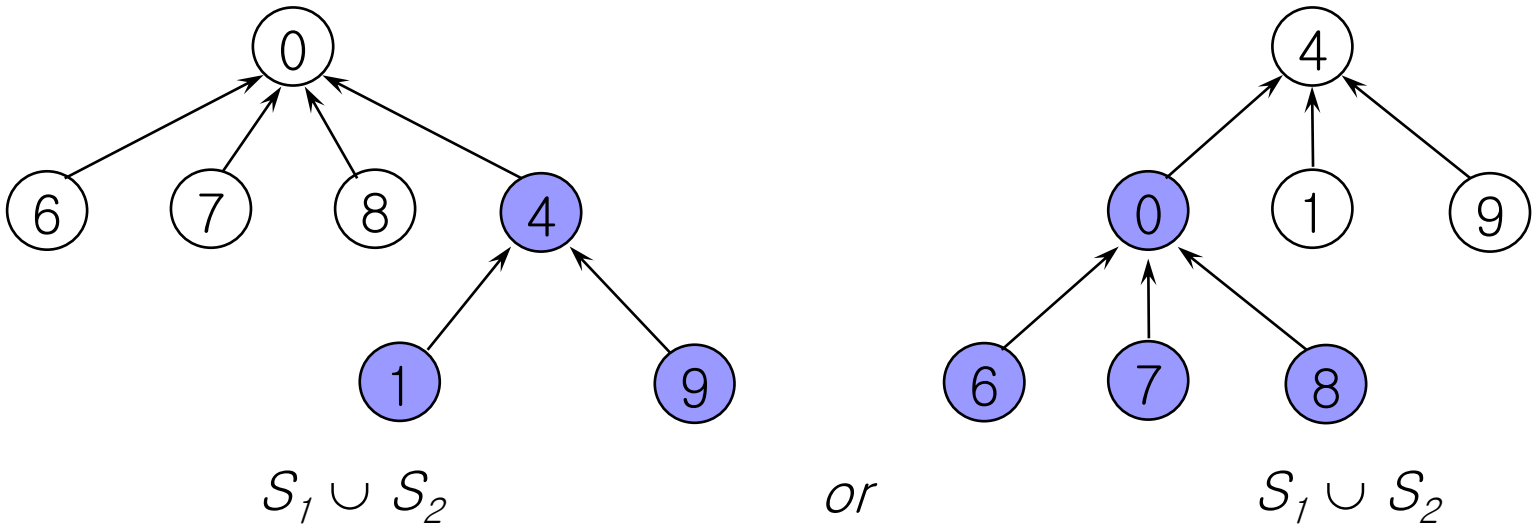
◆ S_1, S_2, S_3의 배열 표현

- i번째 원소 : 원소 i를 포함하는 트리 노드
- 원소 : 대응되는 트리 노드의 부모 포인터
- 루트의 parent는 -1

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	−1	4	−1	2	−1	2	0	0	0	4

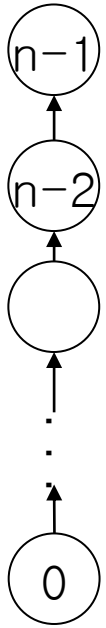
Disjoint Set Union

- **disjoint set 들의 union**
  - $S_i \cup S_j = \{x | x \text{ is in } S_i \text{ or } S_j\}$
  - 두 트리 중의 하나를 다른 트리의 서브트리로 넣음



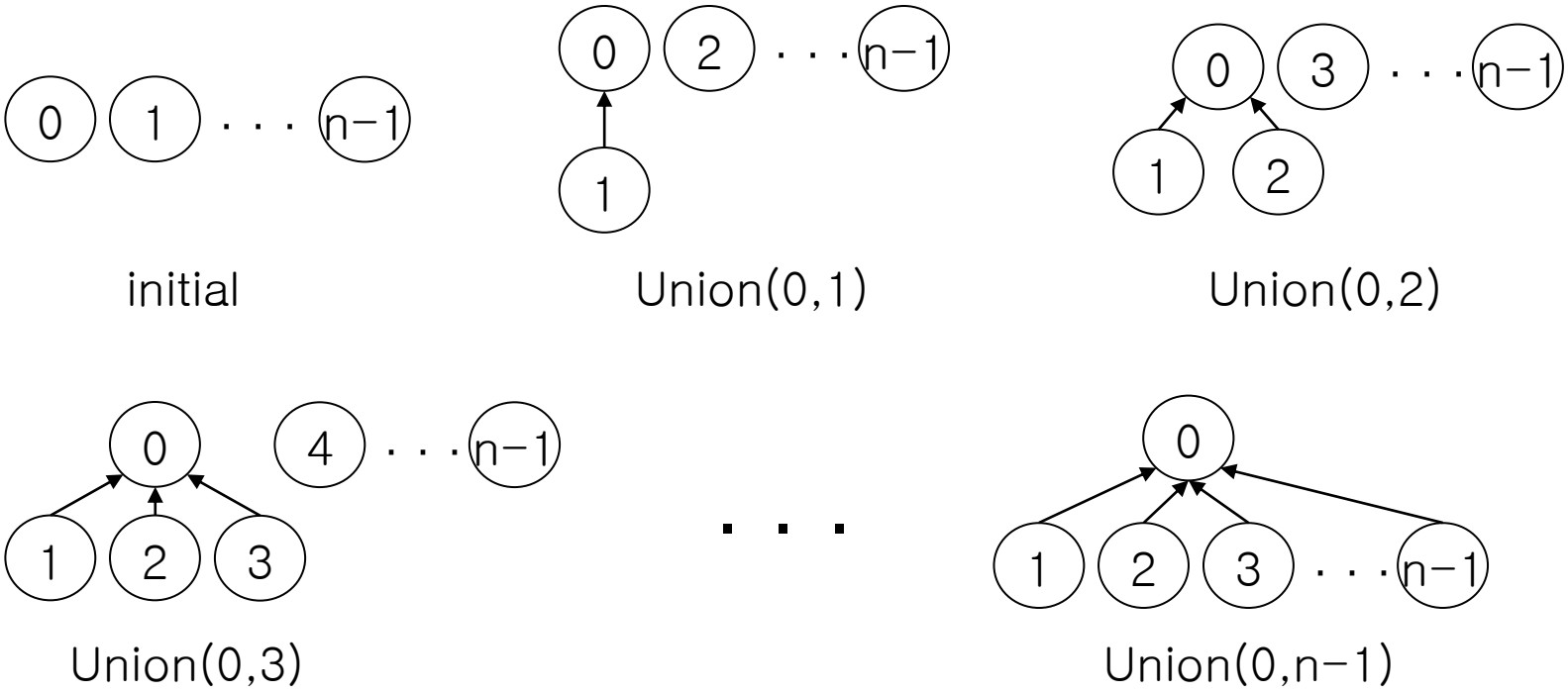
SimpleUnion과 SimpleFind 분석

- **변질(degenerate) 트리**
  - n개의 원소가 각각 n개의 집합에 하나씩 포함된 경우
    - $S_i=\{i\}$ ($0 \le i < n$)
  - 초기상태 : n개의 노드들로 이루어진 포리스트
  - parent[i]= -1 ($0 \le i \le n$)
  - Union(0,1), Union(1,2),Union(2,3), ..., Union(n-2,n-1)
    - n-1번의 합집합이 O(n) 시간에 수행
  - Find(0), Find(1), ..., Find(n-1) 수행
    - 레벨 i에 있는 원소에 대한 수행 시간 : O(i)
    - n번의 Find 수행 : $O(\sum_{i=1}^{n} i) = O(n^2)$



Weighting rule을 적용한 합집합 (1)

- **Union(i,j)를 위한 가중 규칙**
  - 루트 i를 가진 트리의 노드 수 < 루트 j를 가진 트리의 노드 수
    : j를 i의 부모로
  - otherwise
    : i를 j의 부모로



Weighting rule을 이용한 합집합 (2)

- 모든 트리의 루트에 **count(계수) 필드 유지**
  - 루트의 parent는 -1이므로, 루트의 parent에 –count 유지

```
void sets::WeightedUnion(int i, int j)
// 가중규칙을 이용하여 루트가 i와 j(i≠j)인 집합의 합을 구함
// parent[i] = -count[i] 이며 parent[j] = -count[j] 임.
{
   int temp = parent[i]  + parent [j];
   if (parent[i] > parent[j]) {  // i의 노드 수가 적으면
      parent[i] = j; // j를 새 루트로 만든다
      parent[j] = temp;
   }
   else {  // j의 노드 수가 적거나 같으면
      parent[j] = i; // i를 새 루트로 만든다
      parent[i] = temp;
   }
}
```

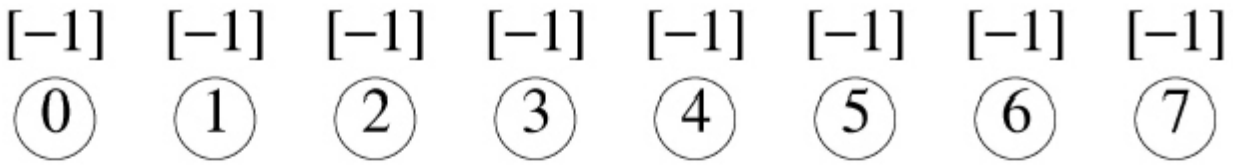
WeightedUnion과 WeightedFind의 분석

- **WeightedUnion : O(1)**

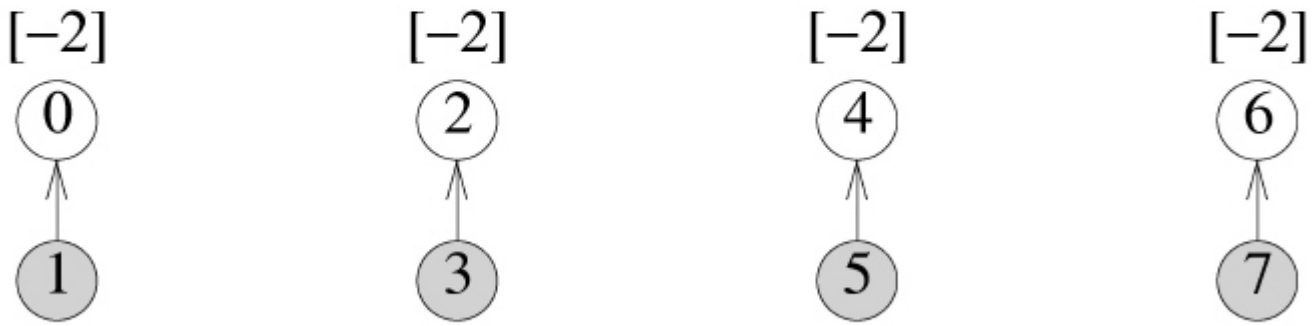
- **WeightedFind(=SimpleFind) : O(log *m*)**
  - 합집합의 결과가 *m*개의 노드를 가진 트리의 높이≤ $\lfloor \log_2 m \rfloor + 1$

- ***u*-1개의 합집합 + *f*개의 탐색 : O(*u*+*f* log *u*)**
  - 트리의 노드 수 ≤ *u*

최악의 경우의 트리 (1)

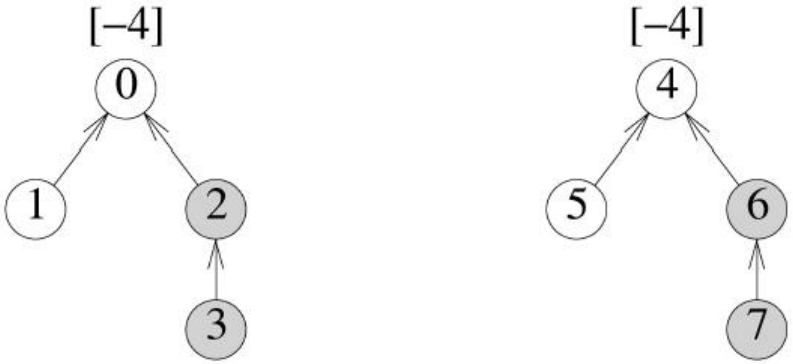


(a) 초기 높이-1 트리

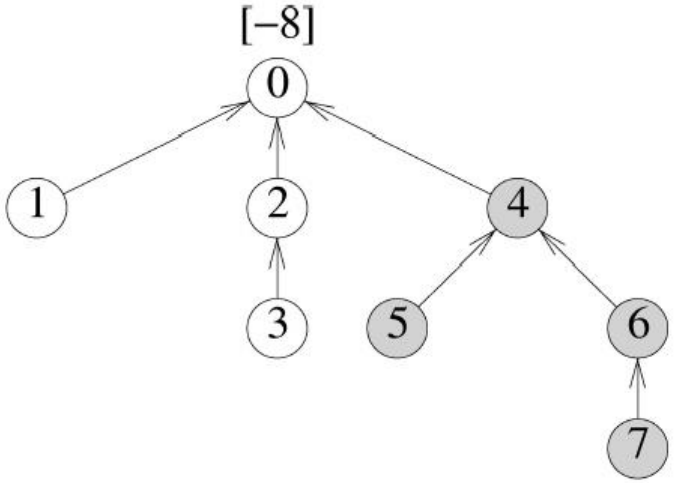


(b) Union(0, 1), (2, 3), (4, 5), (6, 7) 다음의 높이-2 트리

최악의 경우의 트리 (2)



(c) Union(0, 2), (4, 6) 다음의 높이-3 트리



(d) Union(0, 4) 다음의 높이-4 트리

붕괴규칙을 이용한 탐색 알고리즘

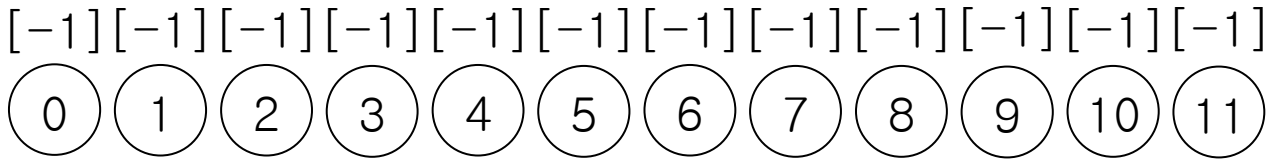
- **붕괴 규칙(collapsing rule)**
  - 만일 j가 i에서 루트로 가는 경로 상에 있고 parent[i]≠root(i)면 parent[j]를 root(i)로 지정

```
int Sets::CollapsingFind(int i)
{// 원소 i를 포함하는 루트를 찾음
// 붕괴 규칙을 이용하여 i로부터 루트로 가는 모든 노드를 붕괴시킴
   for (int r = i; parent[r] >= 0; r = parent[r]); // 루트 r을 찾음
   while( i != r) { //붕괴
     int s = parent[i];
     parent[i] = r;
     i = s;
   }
   return r;
}
```

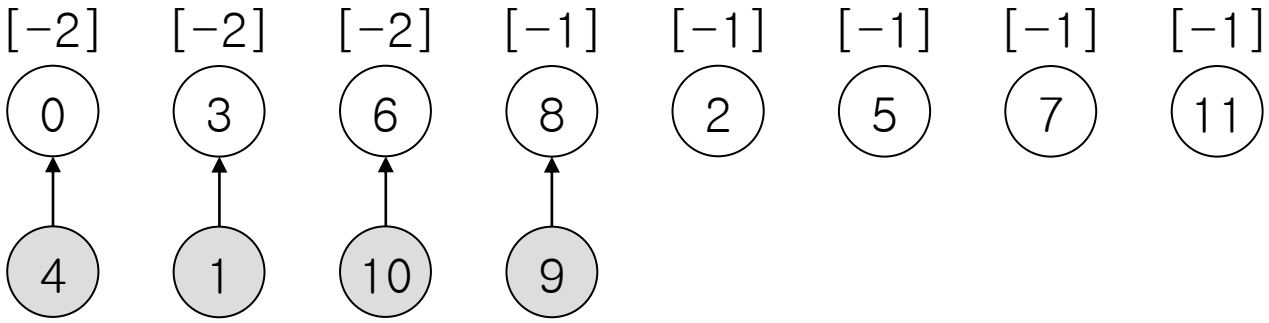
동치 부류의 응용 (1)

- **동치 쌍(equivalence pair)의 처리**
  - 동치 부류(equivalence class)를 집합으로 간주
  - $i \equiv j$
    - 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 : $O(n)$
    - $2m$개의 탐색, 최대 $\min\{n-1,m\}$개의 합집합
      : $O(n+max(2m, \min\{n-1,m\}))$

동치 부류의 응용 예제 (1)

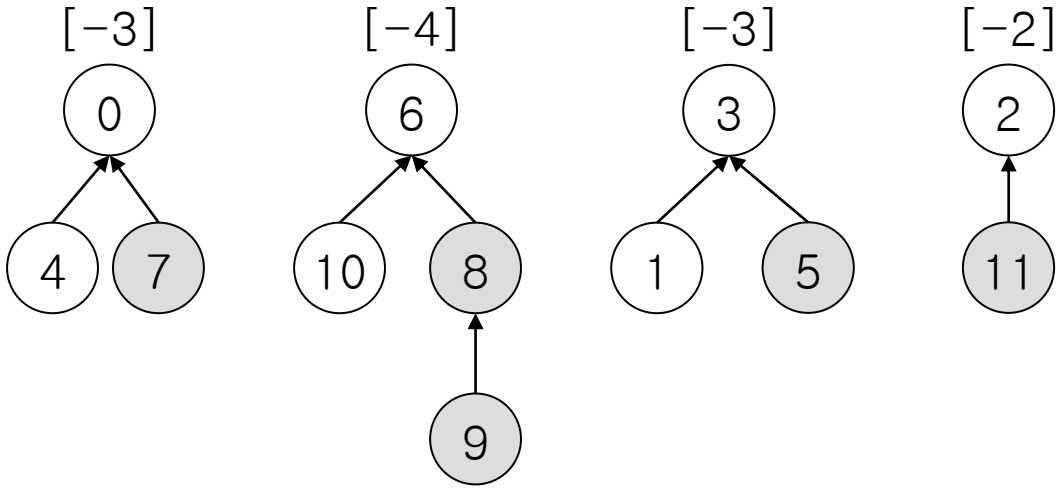


(a) Initial trees

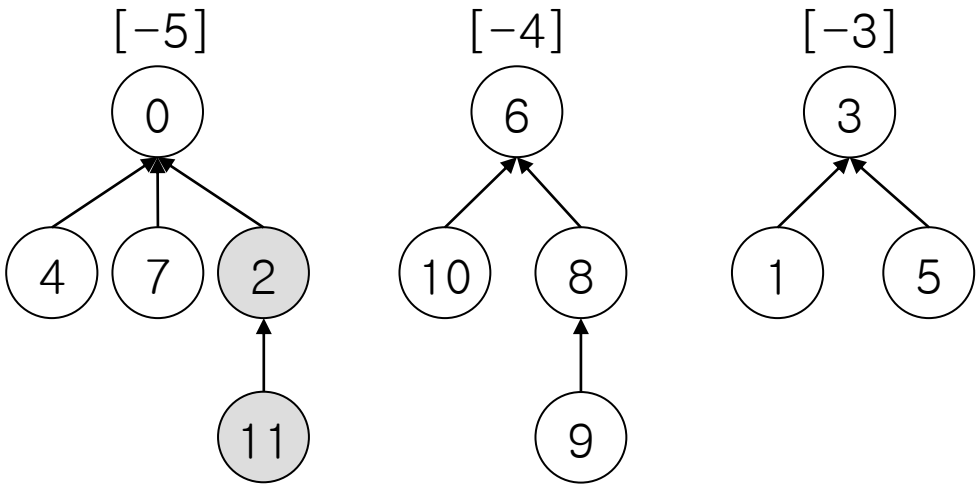


0≡4, 3≡1, 6≡10, and 8≡9 다음의 높이-2 트리

동치 부류의 응용 예제 (2)



(c) 7≡4, 6≡8, 3≡5,and 2≡11 다음의 트리

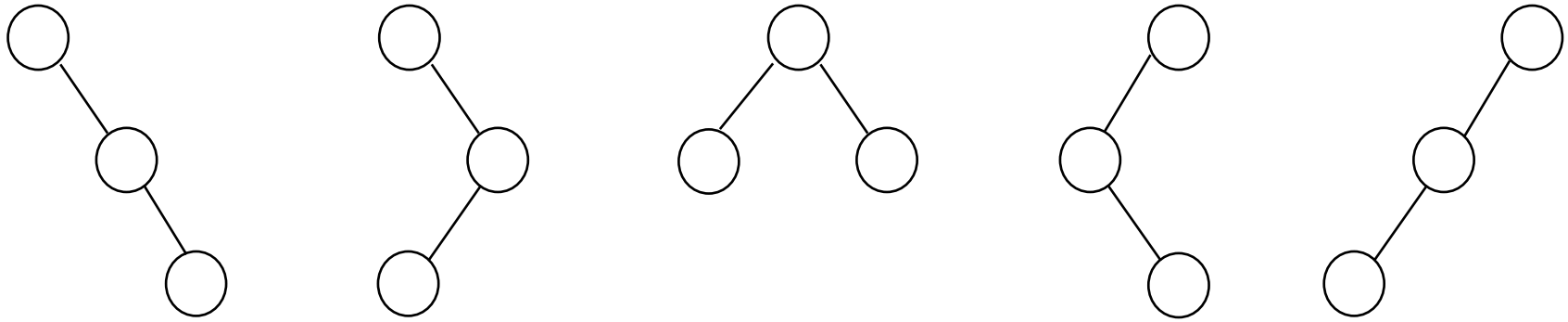


(d) 11≡0 다음의 트리

Counting Binary Trees

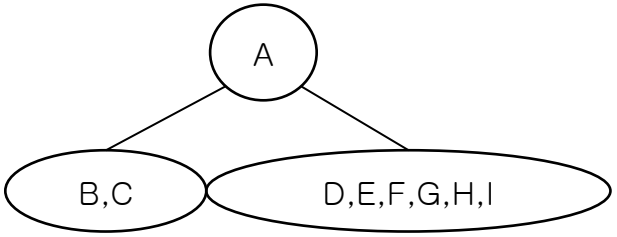
- 상이한 이진 트리
  - n=2 일 때 서로 다른 이진 트리 : 2개

  - n=3 일 때 서로 다른 이진 트리 : 5개

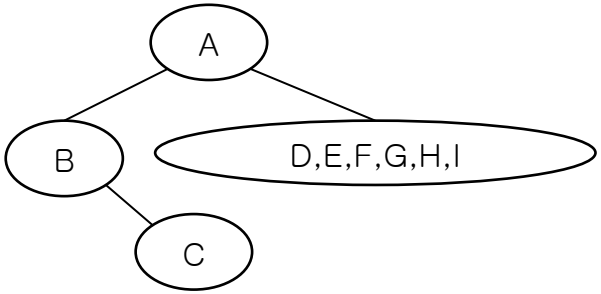


Permutations (1)

- 전위 순서 **ABCDEFGHI**가 중위 순서 **BC A EDGHFI**인 트리는?
  - 전위 순서: A는 트리의 루트
  - 중위 순서: BC는 A의 왼쪽 서브트리, EDGHFI는 오른쪽 서브트리

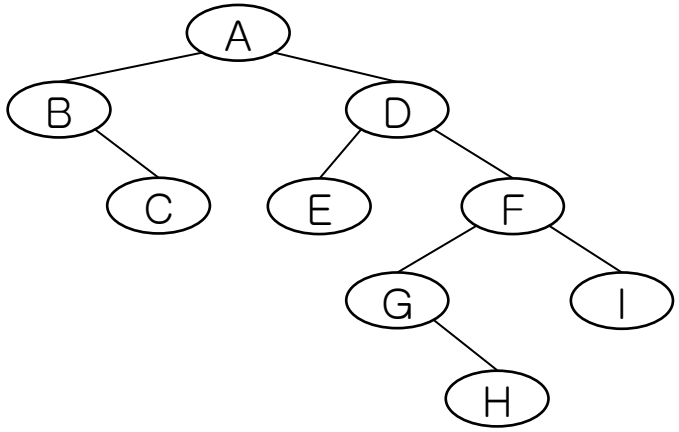


  - 전위 순서: B가 다음번 루트
  - 중위 순서: B의 왼쪽 서브트리는 empty, 오른쪽은 C



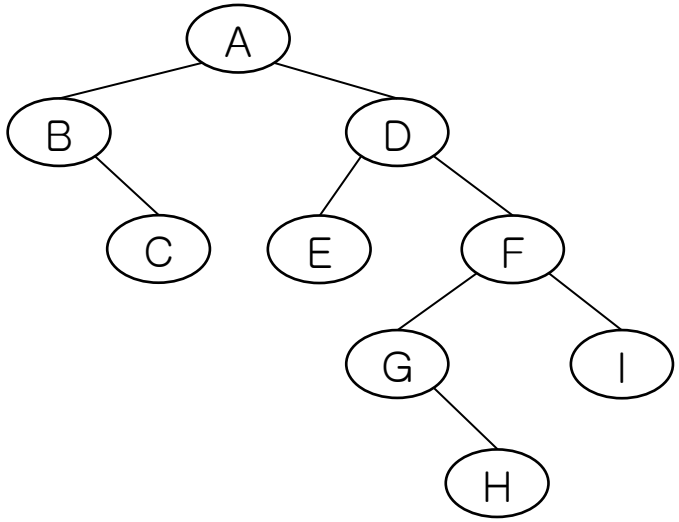
Permutations (2)

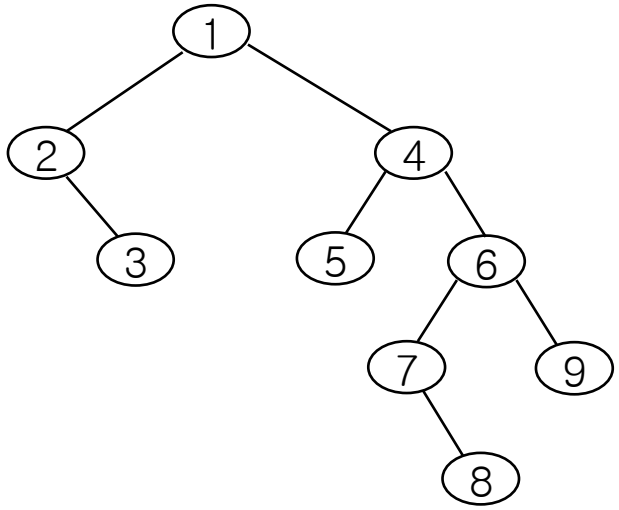
– 이를 반복하면 다음의 트리를 얻는다.



Permutations (3)

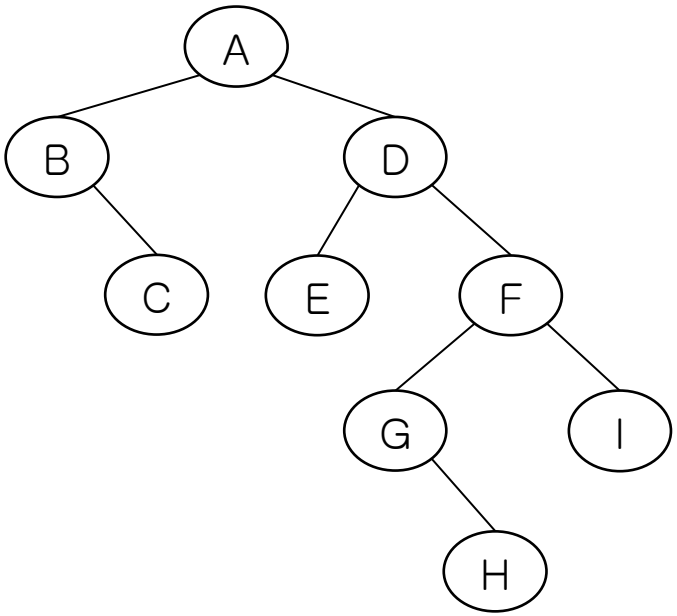
- **전위 순열(preorder permutation)**
  - (예) 좌측의 이진 트리를 전위 순회에 따라 visit(방문)한 노드들의 순서

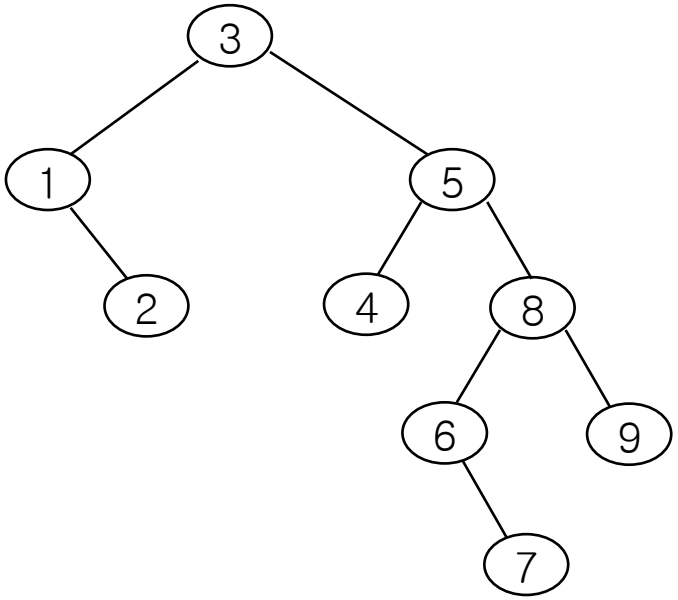




Permutations (4)

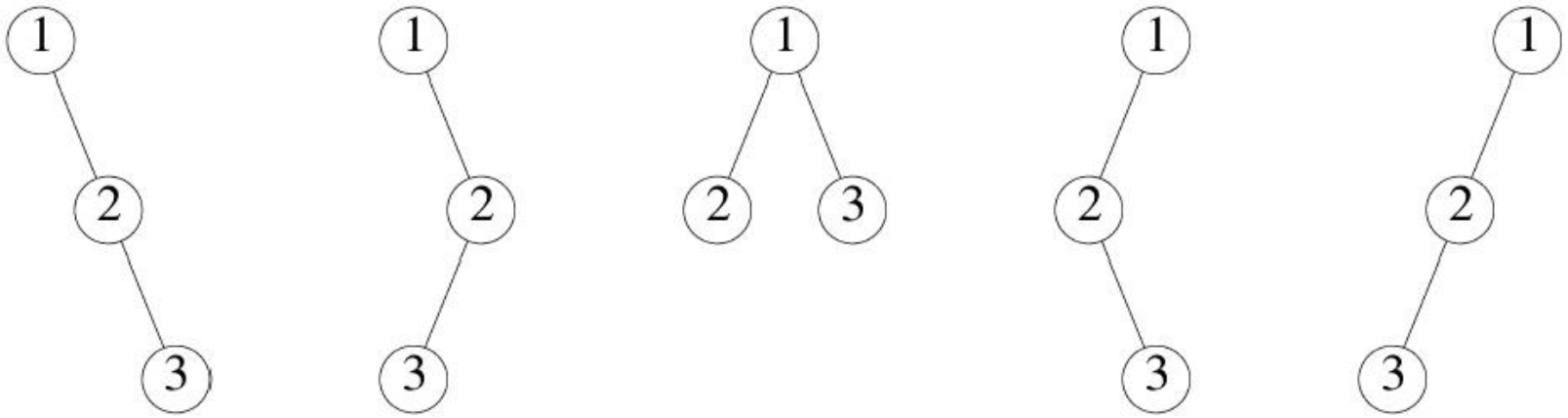
- **중위 순열(inorder permutation)**
  - (예)좌측의 이진 트리를 중위 순회에 따라 visit(방문)한 노드들의 순서





Permutations (5)

- 모든 이진 트리는 유일한 전위-중위 순서 쌍을 가짐
- n개의 노드를 가진 서로 다른 이진트리의 수
  = 1,2,...,n의 전위 순열을 가지는 이진 트리로부터 얻을 수 있는 중위 순열의 수
  =1부터 n까지의 수를 스택에 넣었다가 가능한 모든 방법으로 삭제하여 만들 수 있는 상이한 순열의 수



행렬 곱셈(Matrix Multiplication)

- **n**개 행렬의 곱셈 : $\mathbf{M_1*M_2*M_3*\ ...\ *M_n}$

- **n**개의 행렬을 곱하는 방법의 수를 $\mathbf{b_n}$ 이라 하자.
  - Then (b_1=1), b_2=1, b_3=2, b_4=5

- $\mathbf{M_{ij} = M_i*M_{i+1}*\ ...\ *M_j}$
- $\mathbf{M_{1n} = M_{1i}*M_{i+1,n}}$
  - M_{1i}_계산 방법 수 : b_i
  - $M_{i+1,n}$ 계산 방법 수 : b_{n-i}

$$\left.\begin{array}{l}\\\\\end{array}\right\} \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i},\ n > 1$$

- $\mathbf{b_n}$ = (루트와 노드 수가 $\mathbf{b_i}$, $\mathbf{b_{n-i-1}}$인 두 서브트리)로 된
- 이진트리의 갯수

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1},\ n \geq 1,\ \text{and}\ b_0 = 1$$