5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

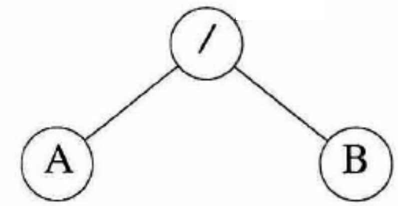
5.4 Additional Binary Tree Operations

5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees

5.3 Binary Tree Traversal



- Tree traversal
  - Visiting each node in the tree exactly once
- When traversing a binary tree
  - L, V, R : moving left, visiting the node, moving right (= 왼쪽 서브노드로 이동)
  - Six possible combinations of traversal
    - LVR, LRV, VLR, VRL, RVL, RLV
  - If we traverse left before right, only tree remains
    - LVR (inorder): … 중위 순회
    - LRV (postorder): … 후위 순회
    - VLR (preorder): … 전위 순회

```c
typedef struct node * treePointer;
typedef struct node{
        int data;
        treePointer leftChild, rightChild;
        };
```

Expression:

A / B * C * D + E

LVR

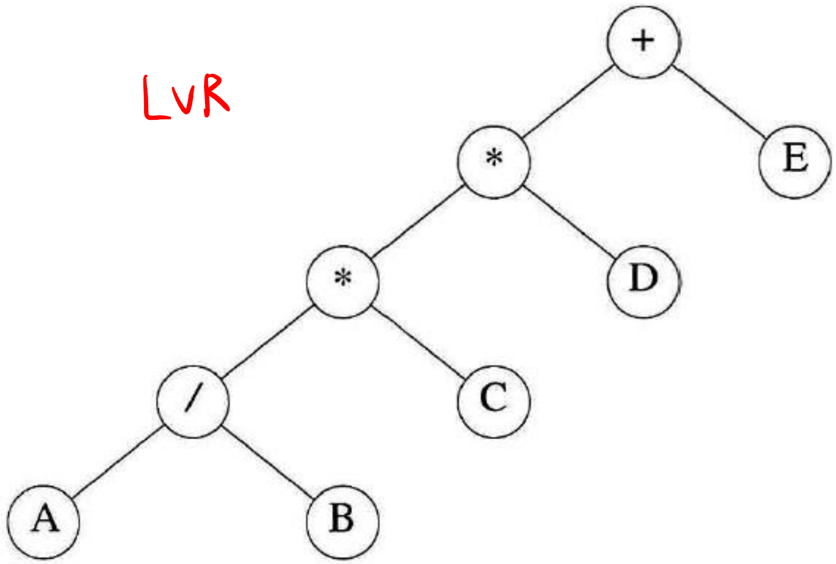
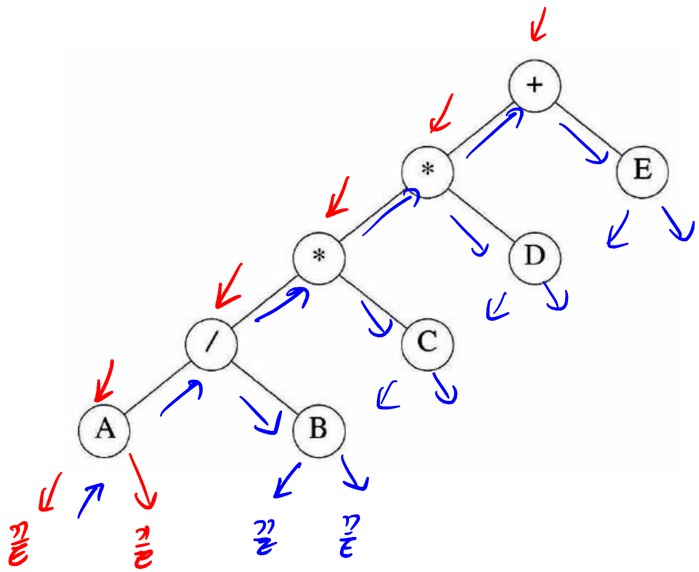


Figure 5.16: Binary tree with arithmetic expression

5.3.Inorder Traversal

- LVR :



→ A/B*C*D+E

→ 2x9+1 ← Empty Node

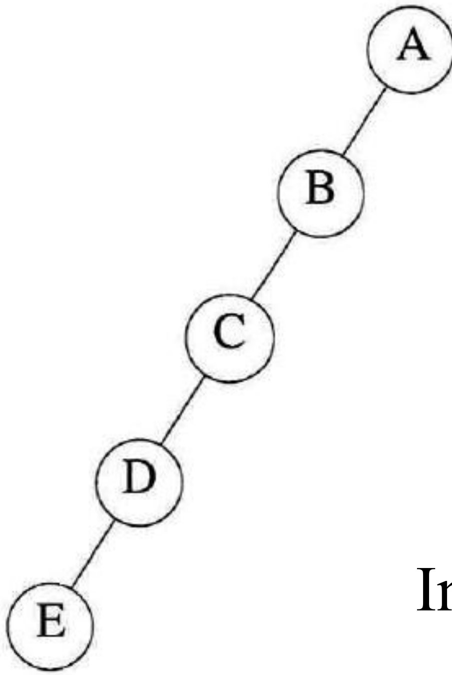
19번호출

→ A의 subtree(=NULL) 만날때 까지

```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

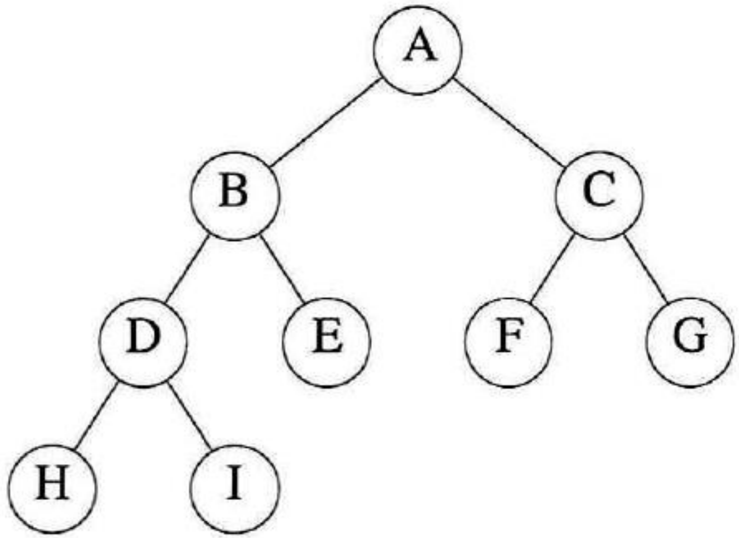
Program 5.1: Inorder traversal of a binary tree

→ # of inorder() : …

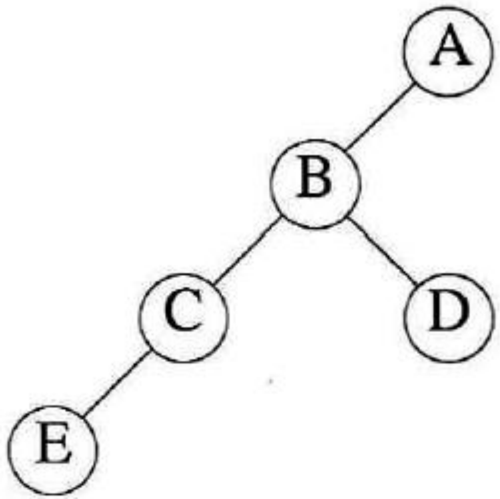


Inorder: …

EDCBA

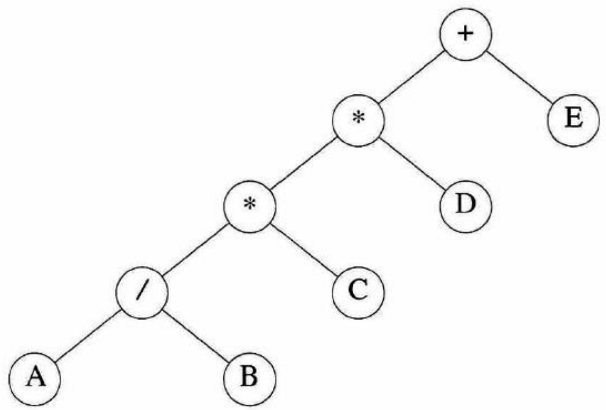


Inorder: ... HDIBEAFCG



Inorder: ... ECBDA

- LVR : → A/B*C*D+E




```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

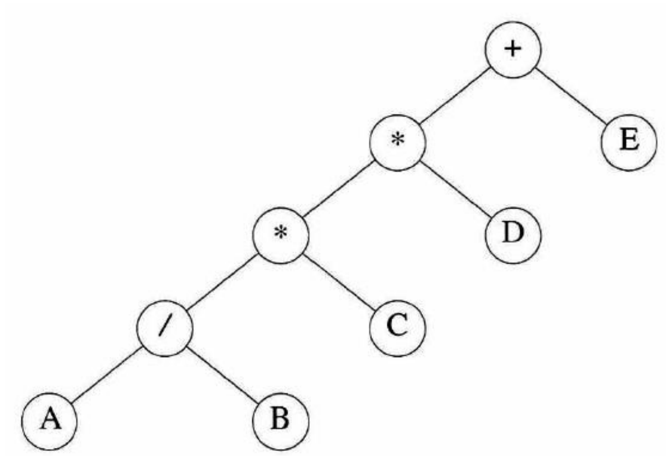
Program 5.1: Inorder traversal of a binary tree

Call of *inorder*	Value in *root*	Action	Call of *inorder*	Value in *root*	Action
1	+		11	*C*	
2	*		12	*NULL*	
3	*		11	*C*	**printf**
4	/		13	*NULL*	
5	*A*		2	*	**printf**
6	*NULL*		14	*D*	
5	*A*	**printf**	15	*NULL*	
7	*NULL*		14	*D*	**printf**
4	/	**printf**	16	*NULL*	
8	*B*		1	+	**printf**
9	*NULL*		17	*E*	
8	*B*	**printf**	18	*NULL*	
10	*NULL*		17	*E*	**printf**
3	*	**printf**	19	*NULL*	

5.3.2 Preorder Traversal

Root을 먼저 출력

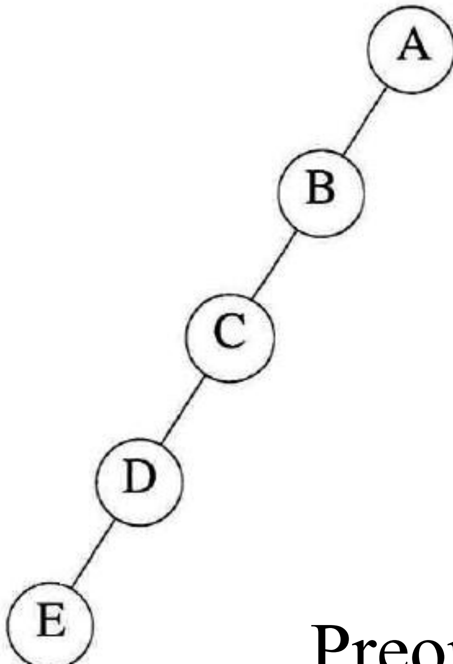
- **V**LR :



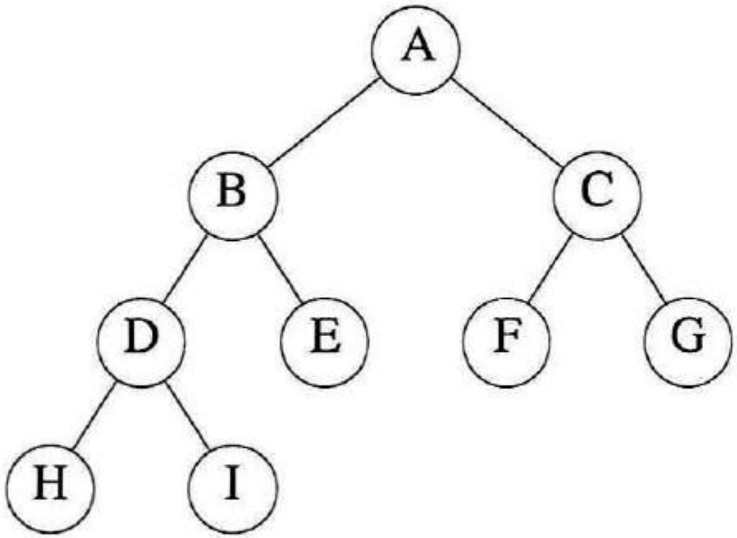

```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree

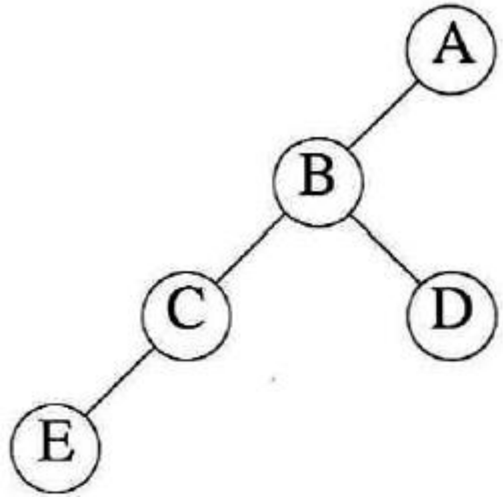
→ + * * / A B C D E



Preorder (VLR): … ABCDE



Preorder (VLR): ... ABDHIECFG

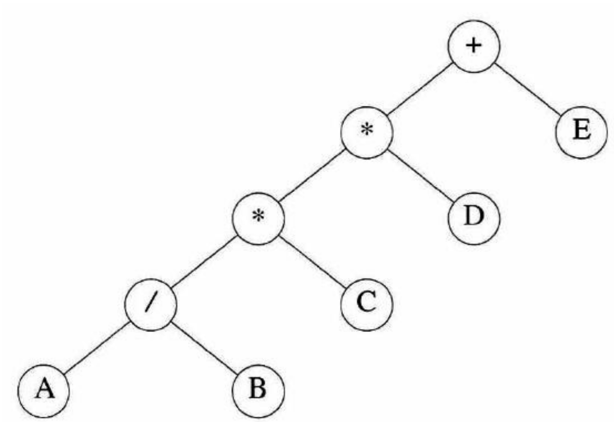


Preorder (VLR): ... ABCED

5.3.3 Postorder Traversal

계산값 출력

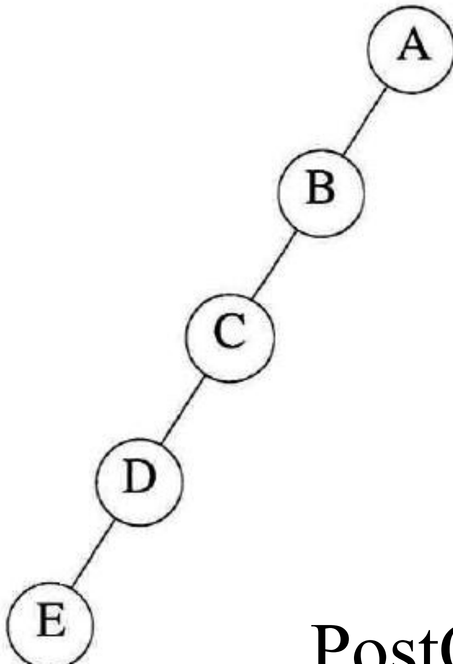
- LRV :



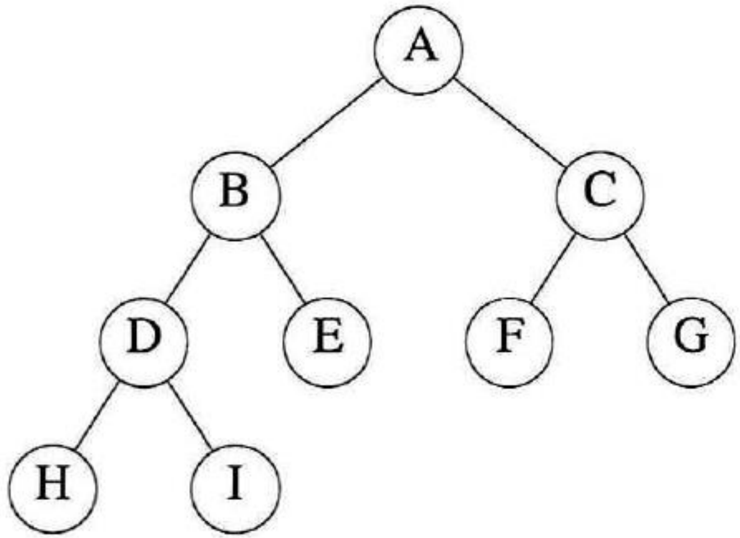

```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

Program 5.3: Postorder traversal of a binary tree

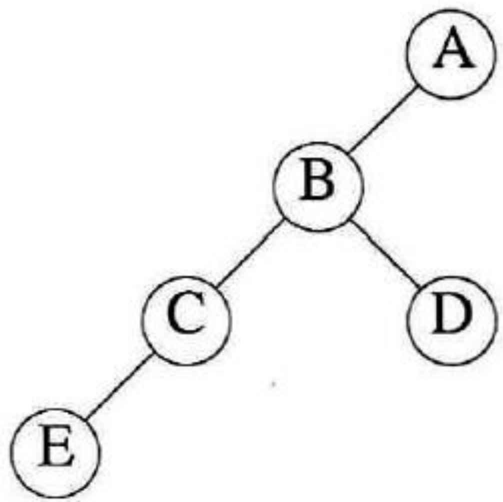
→ A B / C * D * E +



PostOrder (LRV): … EDCBA



PostOrder (LRV): … HIDEBAFGC



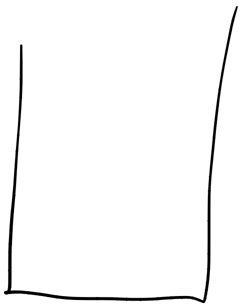
PostOrder (LRV): … ECDBA

반복

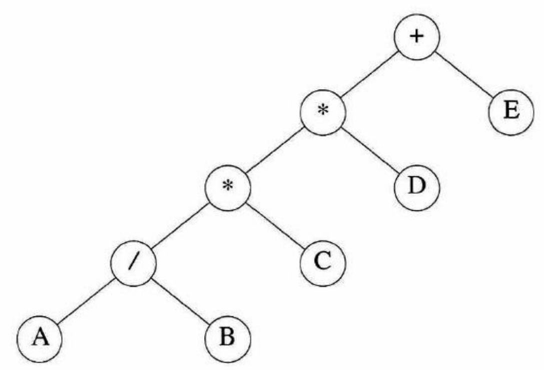
5.3.4 Iterative Inorder Traversal

```
void iterInorder(treePointer node)
{
  int top = -1; /* initialize stack */
  treePointer stack[MAX_STACK_SIZE];
  for (;;) {
    for(; node; node = node→leftChild)
      push(node); /* add to stack */
    node = pop(); /* delete from stack */
    if (!node) break; /* empty stack */
    printf("%d", node→data);
    node = node→rightChild;
  }
}
```

Program 5.4: Iterative inorder traversal



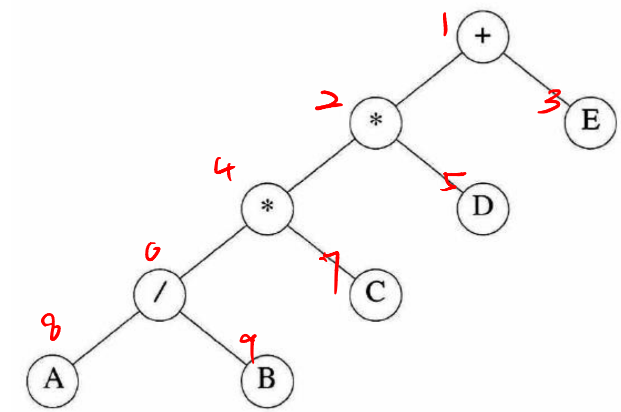
A/B*C*D+E



→ LVR: A/B*C*D+E

```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

Program 5.4: Iterative inorder traversal



→ LVR: A/B*C*D+E

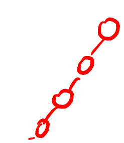
Time complexity:

… O(n)

Space complexity:

… O(n)

↳ tree의 depth



이런 트리면
노드의 개수 = depth

Queue로 구현

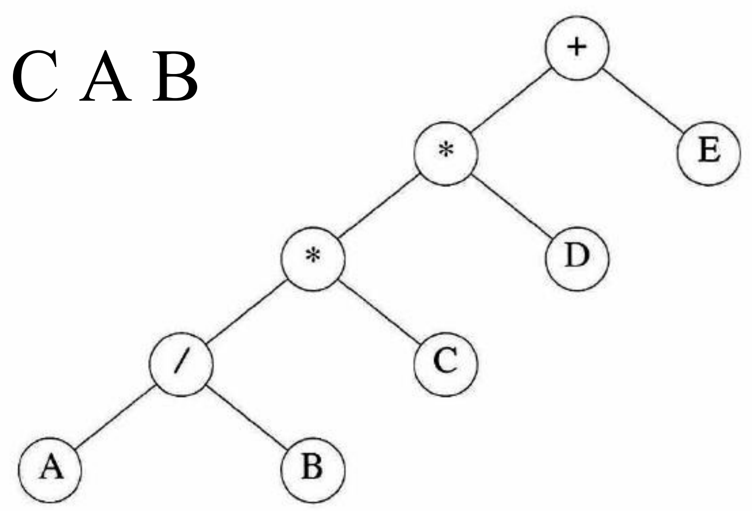
Pre, in, Post order → Stack, Recursion
Level-Order → Queue

5.3.5 Level-Order Traversal

Level 별로 순차적으로

- Requires a queue
  - Visit the **root** first, the root's **left child**, followed by the root's **right child**
  - Continue, visiting the node at each new level from the leftmost node to the rightmost node

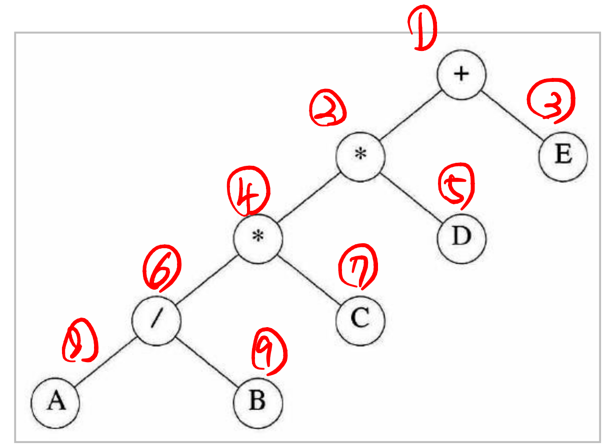
- LO Traversal: + * E * D / C A B



Queue = FIFO

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   int front = rear = 0;
   treePointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

Program 5.5: Level-order traversal of a binary tree



→ + * E * D / C A B

+*E*D

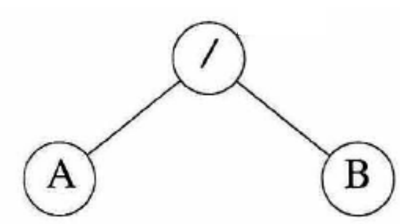
/C

5.4.1 Copying Binary Trees

- This function is a modified version of postorder (Program 5.3)

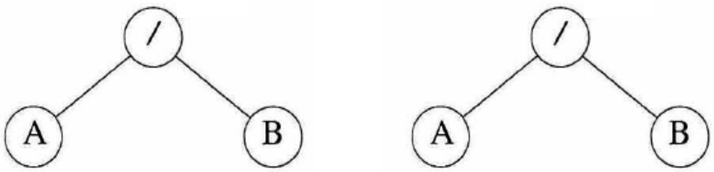
```
treePointer copy(treePointer original) {
/* this function returns a treePointer to an exact copy of the original tree */
        treePointer temp;
        if (orginal) {
                MALLOC(temp, sizeof(*temp));
                temp->leftChild = copy(original->leftChild);
                temp->rightChild = copy(original->rightChild);
                temp->data = original->data;
                return temp;
        }
        return NULL;
}
```

LRV 방식으로 순회



Program 5.6: Copying a binary tree

5.4.2 Testing Equality



- Equivalent BTs
  - have the same structure and the same information in the corresponding nodes
  - Modification of preorder traversal

```
int equal(treePointer first, treePointer second)
{
        return ( (!first && !second) ||
                (first && second &&
                (first->data == second->data) &&
                equal(first->leftChild, second->leftChild) &&
                equal(first->rightChild, second->rightChild)
            );
}
```

Program 5.7: Testing for equality of binary trees

5.4.3 The Satisfiability Problem

- Expression
  e.g) $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$
  - variables $x_1, x_2, \ldots, x_n$: false/true
  - operators: $\wedge$(and), $\vee$(or), $\neg$(not)
- Satisfiability problem:
  - Asks if there is an assignment of values to the variables that causes the value of the expression to be true x_1, x_2, x_3를 어떤 값을주면 true가 될까?
- Solution?
  - Let (x_1, .., x_n) take on all possible combinations 2^n of true and false values and check the formula for each combination;
  - It will take exponential time

LVR
Inorder

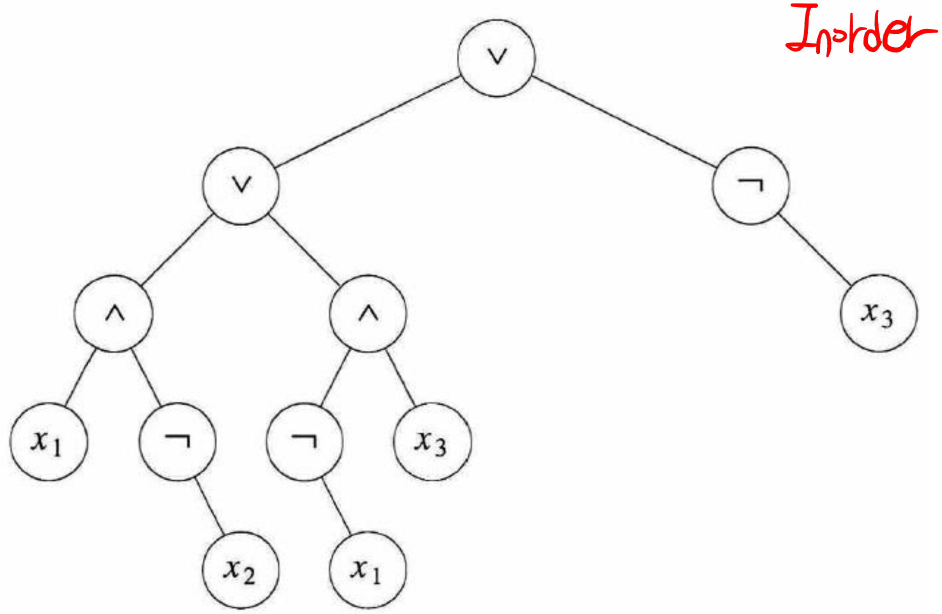


Figure 5.18: Propositional formula in a binary tree

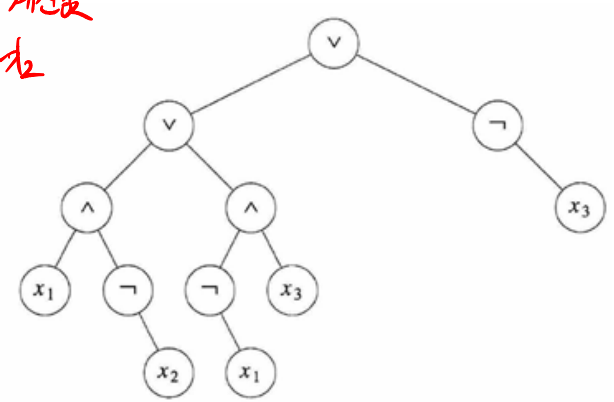
$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

leftChild	*data*	*value*	*rightChild*

Figure 5.19: Node structure for the satisfiability problem

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *treePointer;
typedef struct node {
        treePointer leftChild;
        logical      data;
        short int    value;
        treePointer rightChild;
        } ;
```

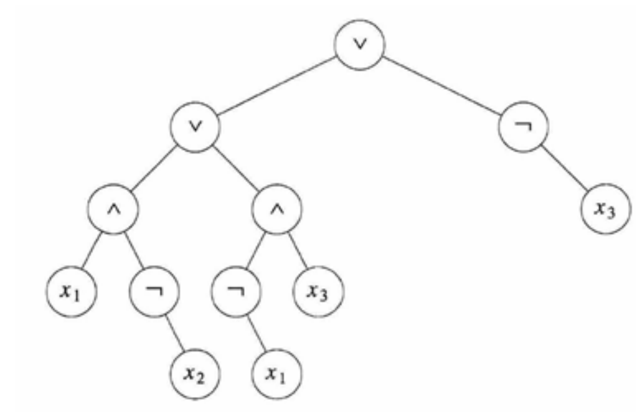
← 중간단계의 계산값
ex) $\sim x_2$

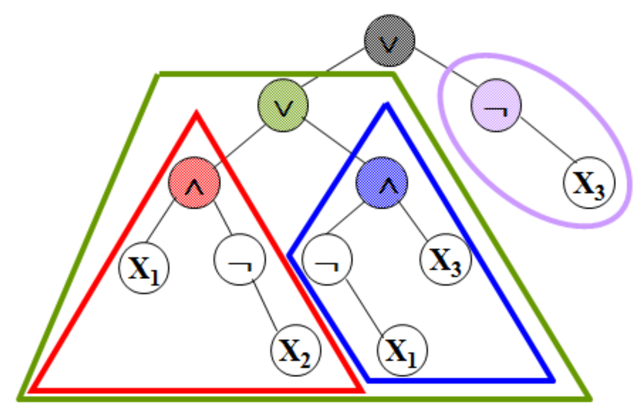


$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

```
for (all 2^n possible combinations)
{
        generate the next combination;
        replace the variables by their values;
        evaluate root by traversing it in postorder;
        if (root->value) {
                printf(<combination>);
                return;
        }
}
printf("No satisfiable combination\n");
```

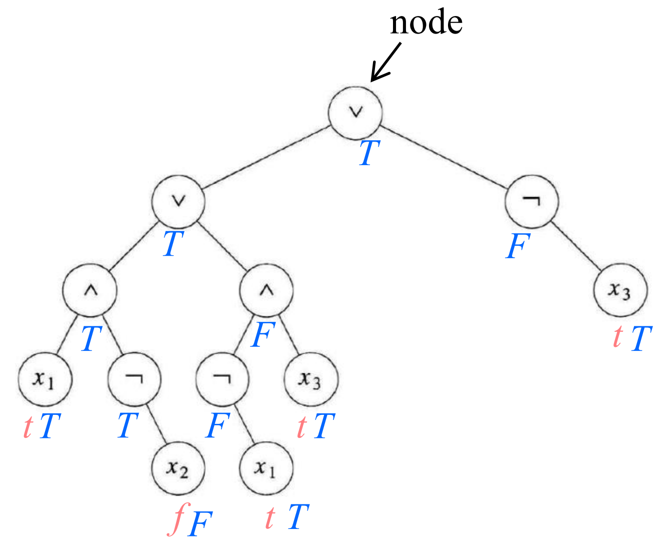
Program 5.8: First version of satisfiability algorithm





```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);
      postOrderEval(node→rightChild);
      switch(node→data) {          연산자 or 변수
 연산자  case not:    node→value =
                    !node→rightChild→value;
                    break;
         case and:    node→value =
                    node→rightChild→value &&
                    node→leftChild→value;
                    break;
         case or:     node→value =
                    node→rightChild→value ||
                    node→leftChild→value;
                    break;
 변수    case true:   node→value = TRUE;
                    break;
         case false:  node→value = FALSE;
      }
   }
}
```

Program 5.9: Postorder evaluation function



ex) $(x_1, x_2, x_3) = (t, f, t)$