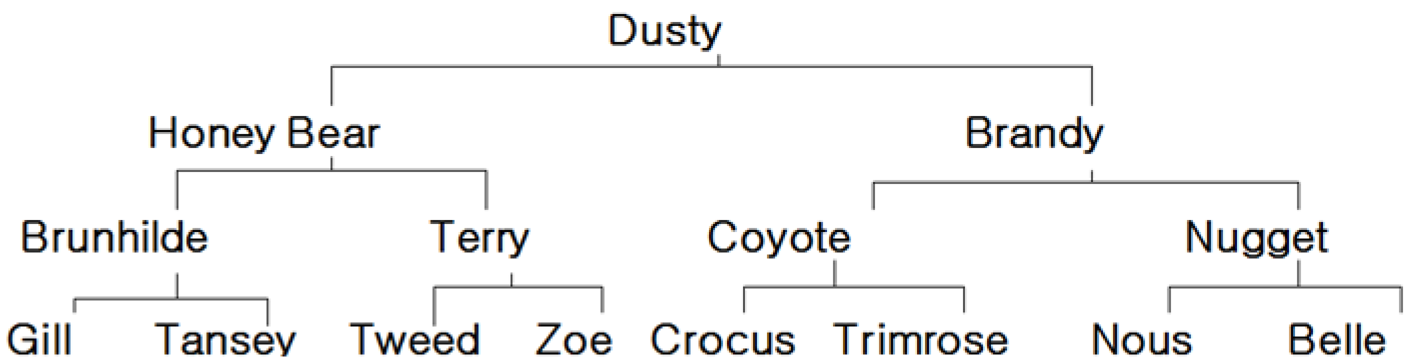
Chapter 5 : TREES

5.1 INTRODUCTION

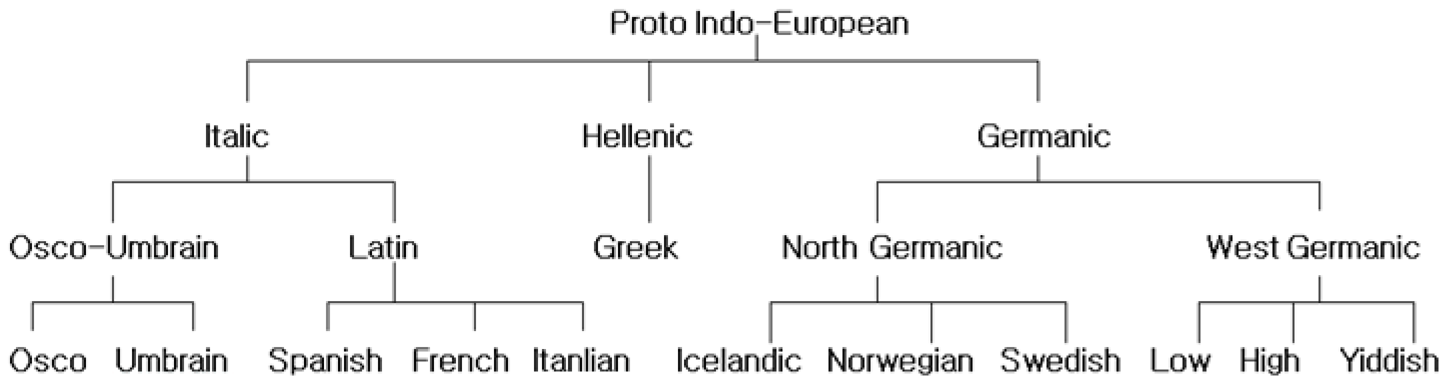
5.1.1 Terminology

The intuitive concept of a tree implies that we organize the data in a hierarchical manner.

[Figure 5.1] Two types of genealogical charts



(a) Pedigree



(b) Lineal

Definition : 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. □

Terms used when referring to trees:

- A ***node*** stands for the item of information and the branches to other nodes.
- The ***degree*** of a node is the number of subtrees of the node.
- The ***degree of a tree*** is the maximum degree of the nodes in the tree.
- A node with degree zero is a ***leaf*** or ***terminal*** node.
- A node that has subtrees is the ***parent*** of the roots of the subtrees, and the roots of the subtrees are the ***children*** of the node.
- Children of the same parent are ***siblings***.
- The ***ancestors*** of a node are all the nodes along the path from the root to the node. Conversely, the ***descendants*** of a node are all the nodes that are in its subtrees.

Terms used when referring to trees:

- The ***level*** of a node is defined by :

  Initially letting the root be at level one.

  For all subsequent nodes, the level of a node is the level of the node's parent plus one.

- The ***height*** or ***depth*** of a tree is the maximum level of any node in the tree.

5.1.2 Representation Of Trees

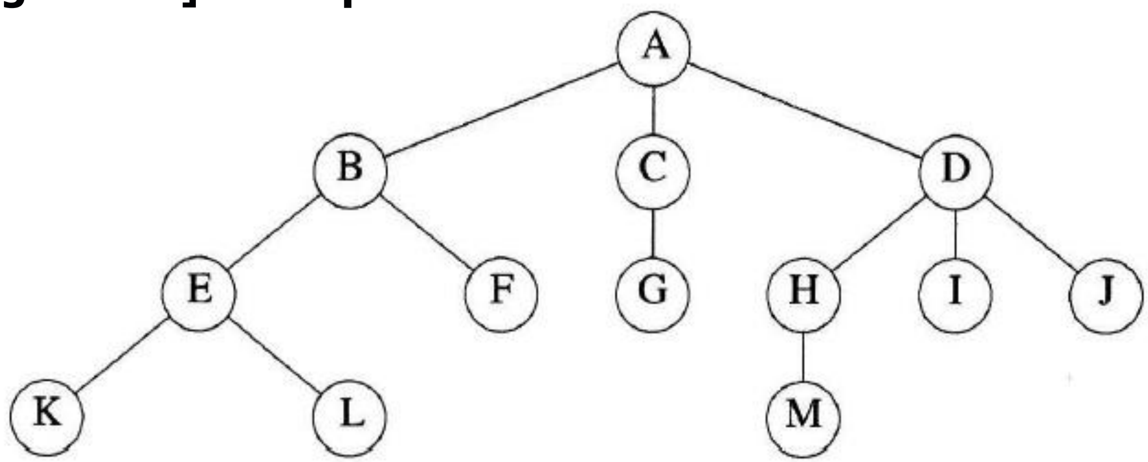
List Representation

Representing a tree as a list in which each of the subtrees is also a list.

For example, the tree of Figure 5.2 is written as :

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

[Figure 5.2] A sample tree



5.1.2 Representation Of Trees

If we wish to use linked lists, then a node must have a varying number of pointer fields depending on the number of children.

[Figure 5.4] Possible node structure for a tree of degree k

DATA	CHILD 1	CHILD 2	...	CHILD k

It is often easier to work with nodes of a fixed size.

5.1.2 Representation Of Trees

Left Child-Right Sibling Representation

The representations we consider require exactly two link or pointer fields per node.

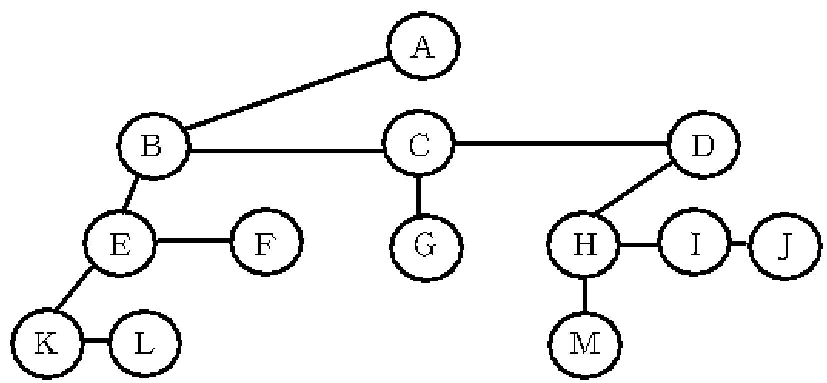
Note that every node has at most one leftmost child and at most one closest right sibling.

(* Strictly speaking, the order of children in a tree is not important. *)

[Figure 5.5]

data	
left child	right sibling

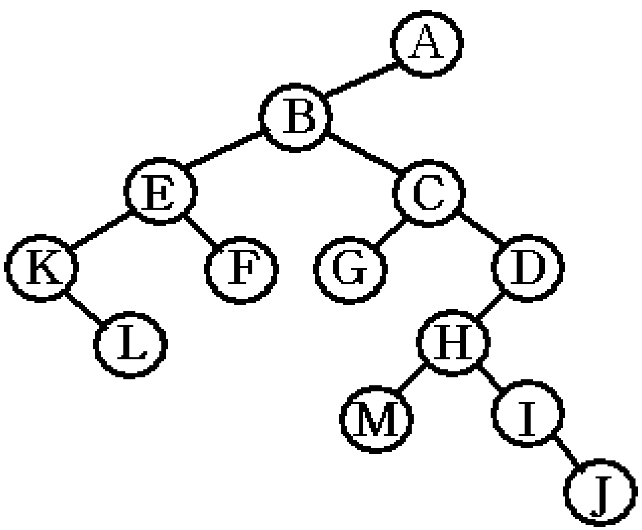
[Figure 5.6]



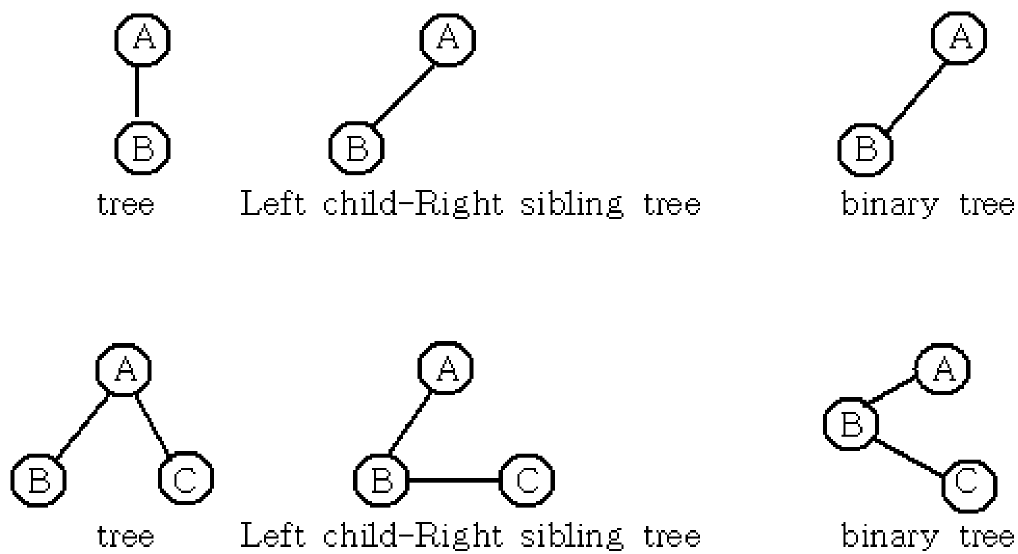
Representation As A Degree-Two Tree

- Obtained by rotating the right-sibling pointer in a left child-right sibling tree clockwise by 45 degrees.
- Two children of a node are called the left and right children.
- Notice that the right child of the root node of the tree is empty.
- Left child-right child trees are also known as *binary trees*.

[Figure 5.7]



[Figure 5.8]



5.2 BINARY TREES

5.2.1 The Abstract Data Type

- The chief characteristic of a binary tree is the stipulation that the degree of any given node must not exceed two.

- For binary trees, we distinguish between the left subtree and the right subtree, while for trees the order of the subtrees is irrelevant.

- <u>Definition</u> : A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

[Distinctions between a binary tree and a tree]

(1) There is no tree having zero nodes, but there is an empty binary tree.

(2) In a binary tree, we distinguish between the order of the children while in a tree we do not.

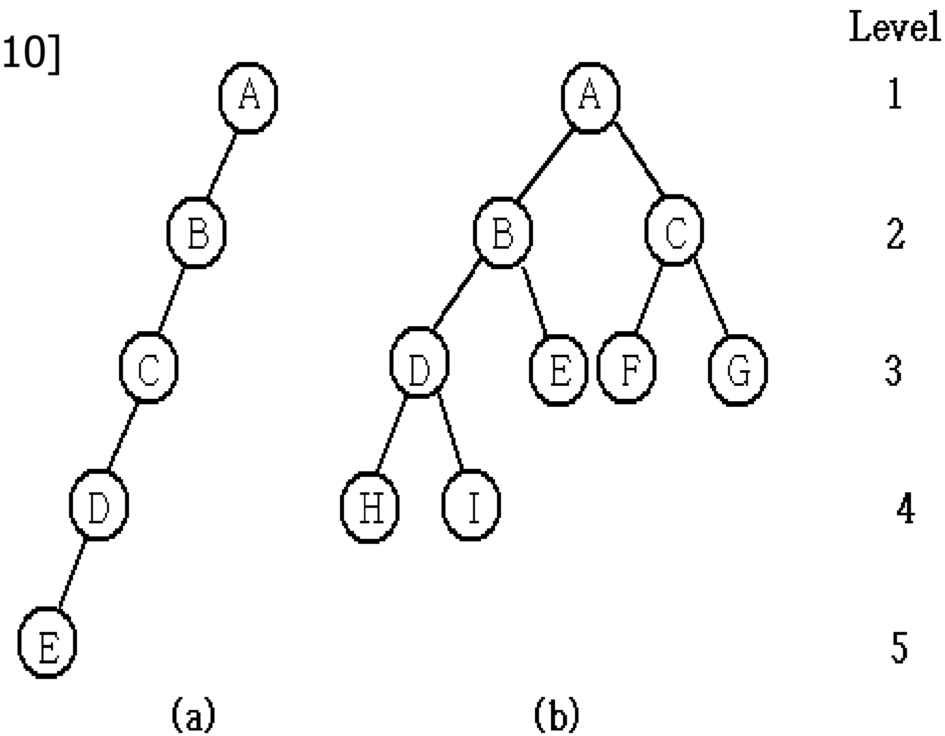
Example : [Figure 5.9] Two different binary trees

Special Types of binary trees

(a) Skewed tree

(b) Complete binary tree (will be defined formally later)

[Figure 5.10]



The same terminology we used to describe trees applies to binary trees.

- node
- degree of a node, degree of a tree
- leaf or terminal
- parent, children (left child, right child), sibling
- ancestor, descendant
- level of a node, height or depth

ADT 5.1: Abstract data type Binary_Tree

ADT Binary_Tree (abbreviated BinTree) is

objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree.

functions:

for all bt, bt1, bt2 $\in$ BinTree, item $\in$ element

BinTree Create() ::= creates an empty binary tree

Boolean IsEmpty(*bt*) ::= **if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*

BinTree MakeBT(*bt1*, *item*, *bt2*) ::= **return** a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*.

BinTree Lchild(*bt*) ::= **if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.

element Data(bt) ::= **if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.

BinTree Rchild(*bt*) ::= **if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

5.2.2 Properties Of Binary Trees

Lemma 5.2 [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

<proof> The proof is by induction on i.

Induction Base : The root is the only node on level $i = 1$. Hence, the maximum number of nodes on level $i = 1$ is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis : Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level $i-1$ is 2^{i-2}.

Induction step : The maximum number of nodes on level $i-1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level $i-1$, or 2^{i-1}.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

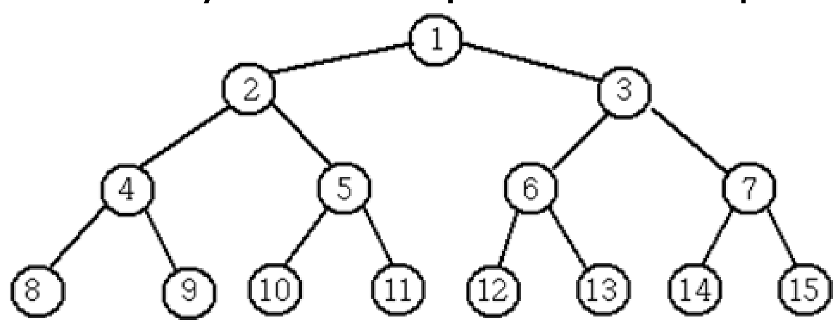
<proof>
The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k}(maximum\ number\ of\ nodes\ on\ level\ i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 \ \square$$

<u>Definition</u> : A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$. □

We can number the nodes in a full binary tree, starting with the root on level 1, continuing with the nodes on level 2, and so on. Nodes on any level are numbered from left to right.

[Figure 5.11] Full binary tree of depth 4 with sequential node numbers



<u>Definition</u> : A binary tree with n nodes and depth k is *complete* iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k. □

5.2.3 Binary Tree Representations

Array Representation

By using the numbering scheme shown in Figure 5.11, we can use a one-dimensional array to store the nodes in a binary tree. (We do not use the 0-th position of the array.)

Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
If $i = 1$, i is at the root and has no parent.

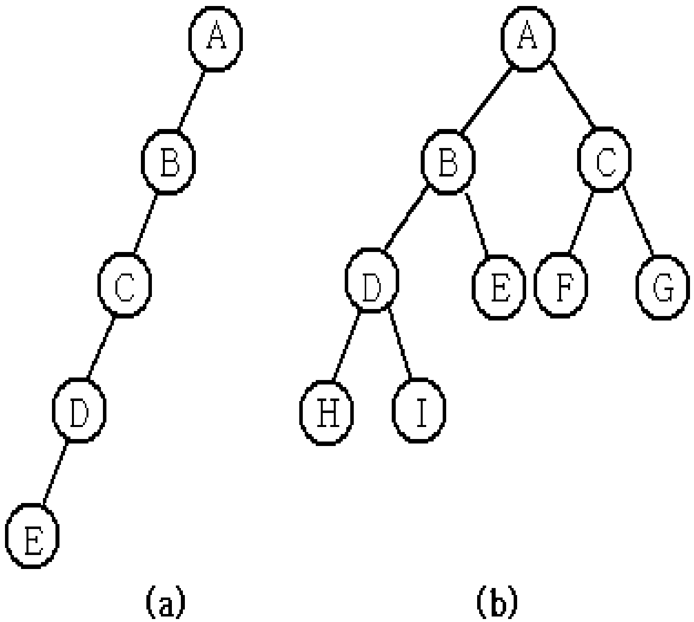
(2) left_child(i) is at $2i$ if $2i \le n$. If $2i > n$, then i has no left child.

(3) right_child(i) is at $2i + 1$ if $2i + 1 \le n$.
If $2i + 1 > n$, then i has no right child.

<proof> We proof (2). (3) is an immediate consequence of (2) and the numbering of nodes on the same level from left to right. (1) follows from (2) and (3).

We prove (2) by induction on i. For $i = 1$, clearly the left child is at 2 unless $2 > n$, in which case i has no left child. Now assume that for all j, $1 \leq j \leq i$, left_child(j) is at $2j$. Then the two nodes immediately preceding left_child($i + 1$) are the right and left children of i. The left child is at $2i$. Hence, the left child of $i + 1$ is at $2i + 2 = 2(i + 1)$ unless $2(i + 1) > n$, in which case $i + 1$ has no left child. $\square$

[Figure 5.10]



[Figure 5.12] Array representation of the binary trees of Figure 5.10

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

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

Linked Representation

Array representation suffers from the general inadequacies of sequential representations.

e.g., Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes.

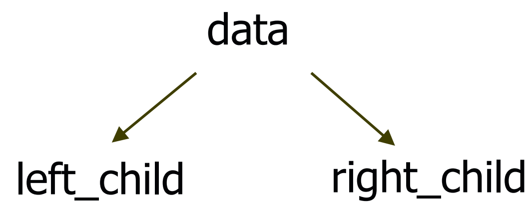
These problems can be overcome easily through the use of a linked representation.

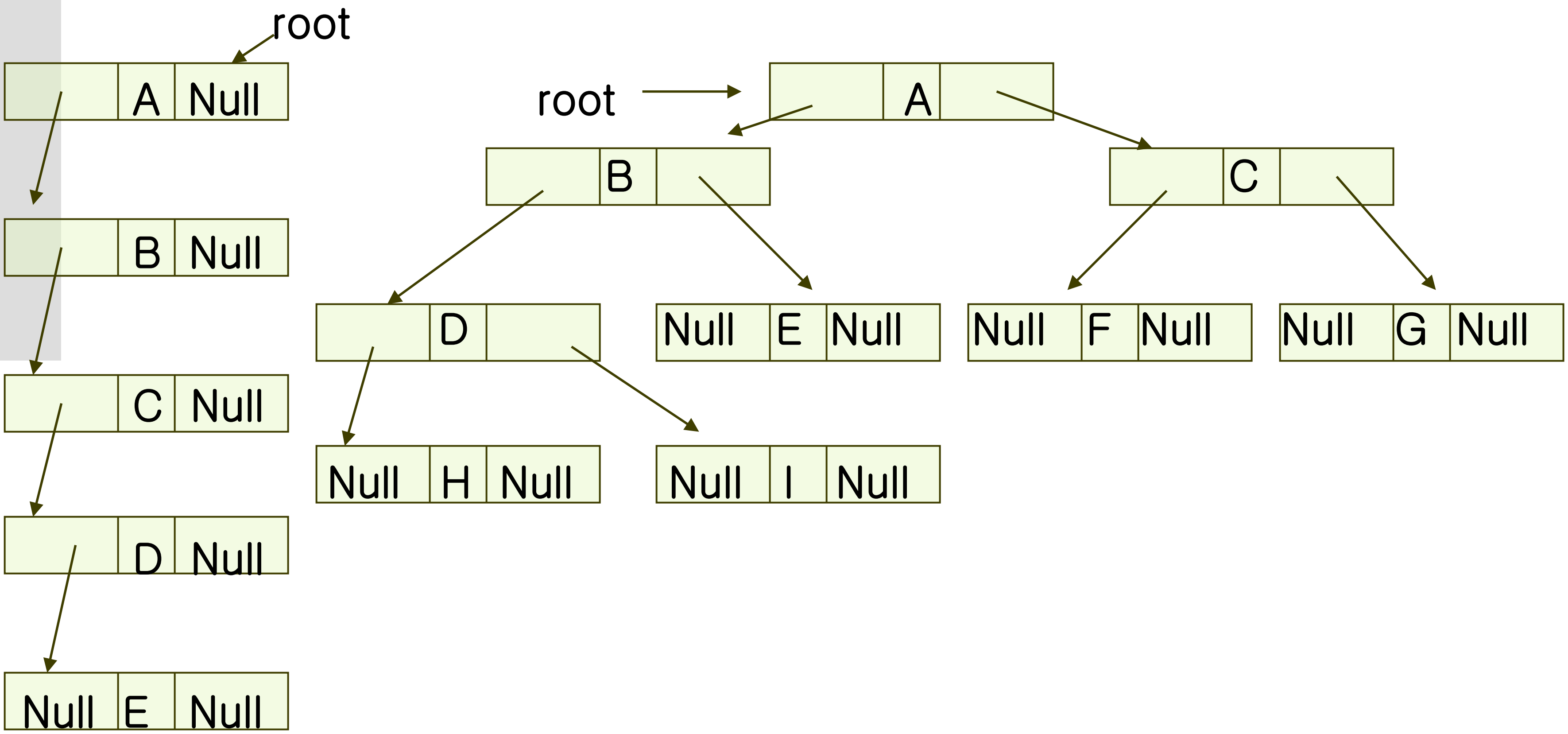
Node structure :

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```

[Figure 5.13] Node representations

left_child	data	right_child



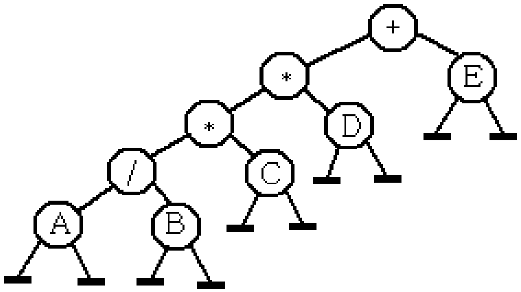


[Figure 5.14] Linked representation for the binary trees of Figure 5.10.

5.3 BINARY TREE TRAVERSALS

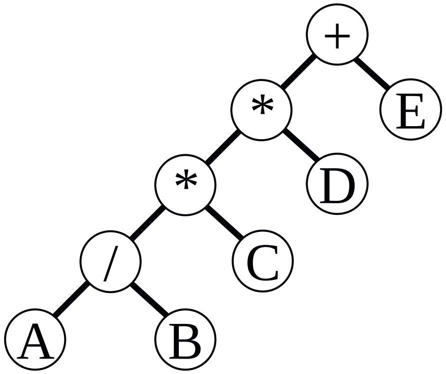
- One of the operations that arises frequently is traversing a tree, that is, visiting each node in the tree exactly once.
- A full traversal produces a linear order for the information in a tree.
- When traversing a tree we want to treat each node and its subtrees in the same way.
- Let, for each node in a tree,
  L stand for moving left,
  V stand for visiting the node (e.g., printing out the data field),
  R stand for moving right.

- Six possible combinations of traversal :
  LVR, LRV, VLR, VRL, RVL and RLV
- If we adopt the convention that we traverse left before right, then only three traversals remain:
  LVR : inorder traversal
  LRV : postorder traversal
  VLR : preorder traversal
- There is a natural correspondence between these traversals and producing the infix, postfix, and prefix forms of an expression.
- [Figure 5.16] Binary tree with arithmetic expression



Program 5.1: Inorder Traversal of a binary tree

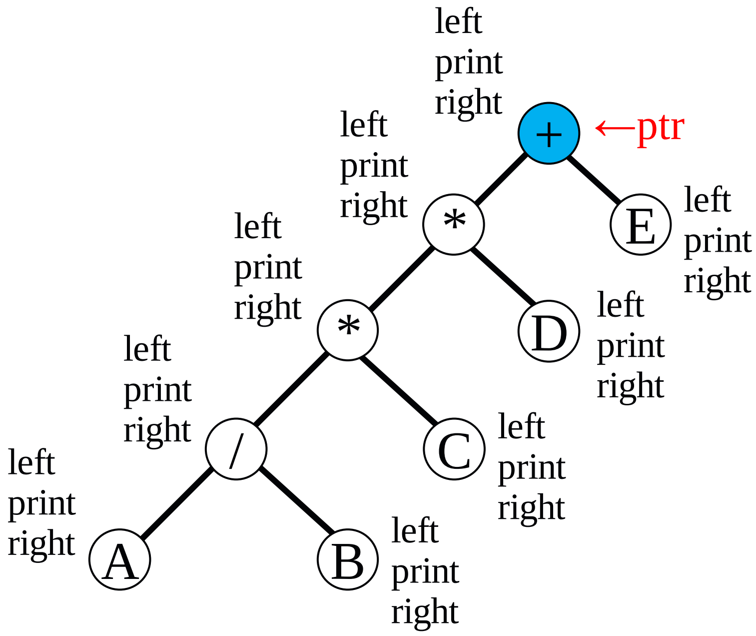
```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



- The data fields of Figure 5.16 are output in the order :

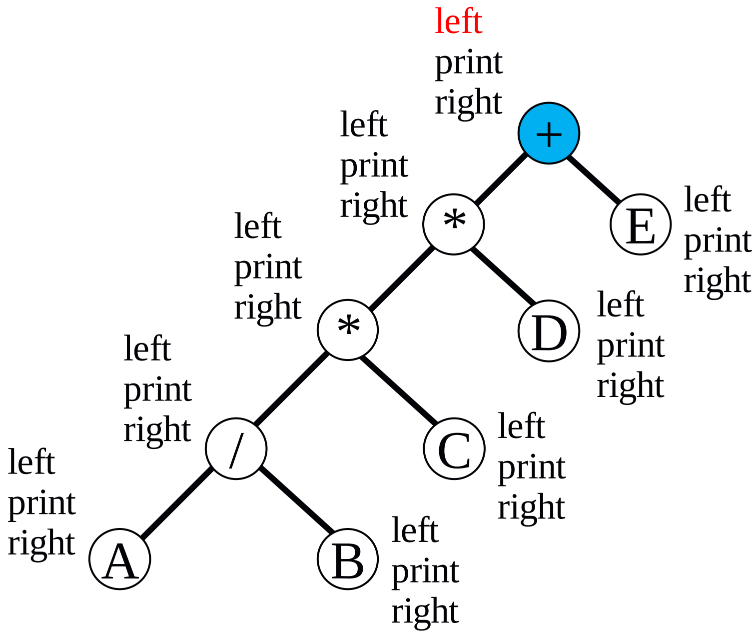
  A / B * C * D + E

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



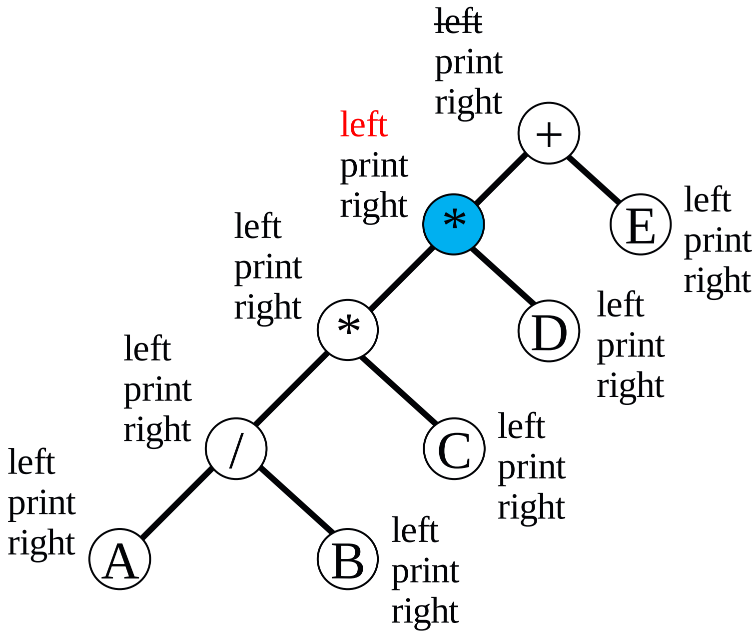
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



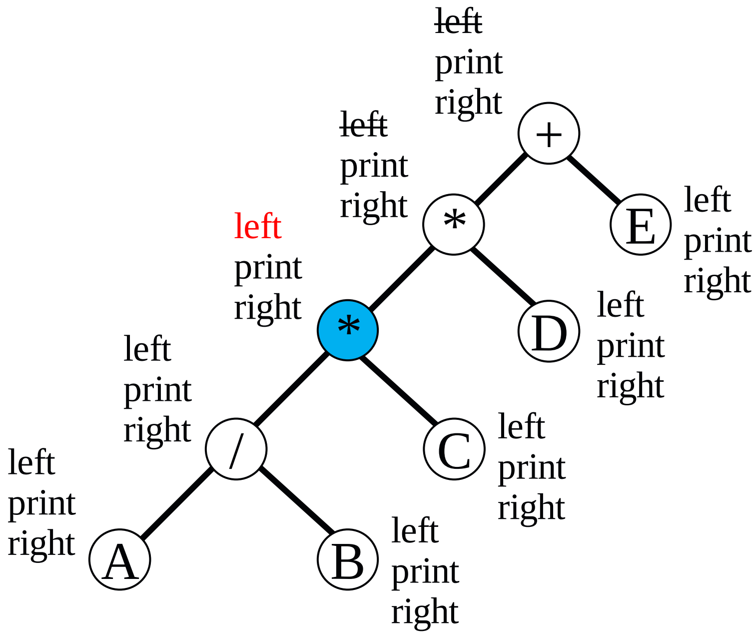
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



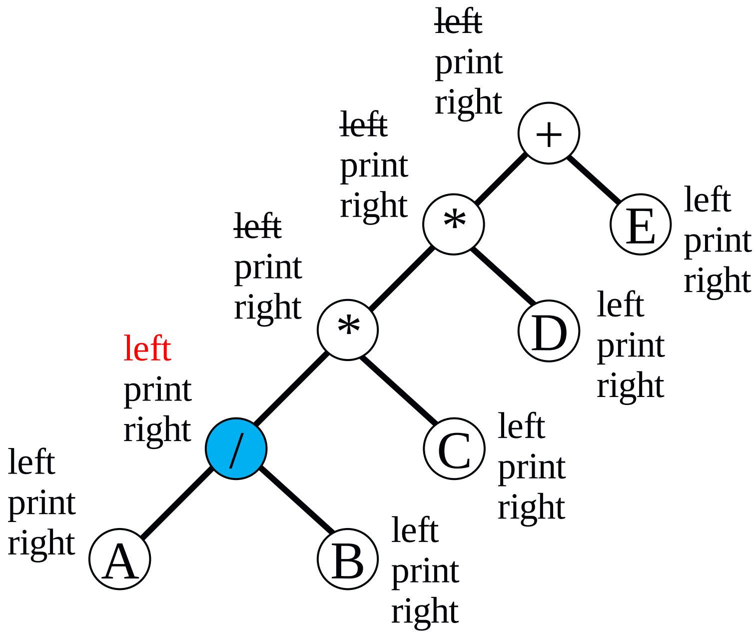
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



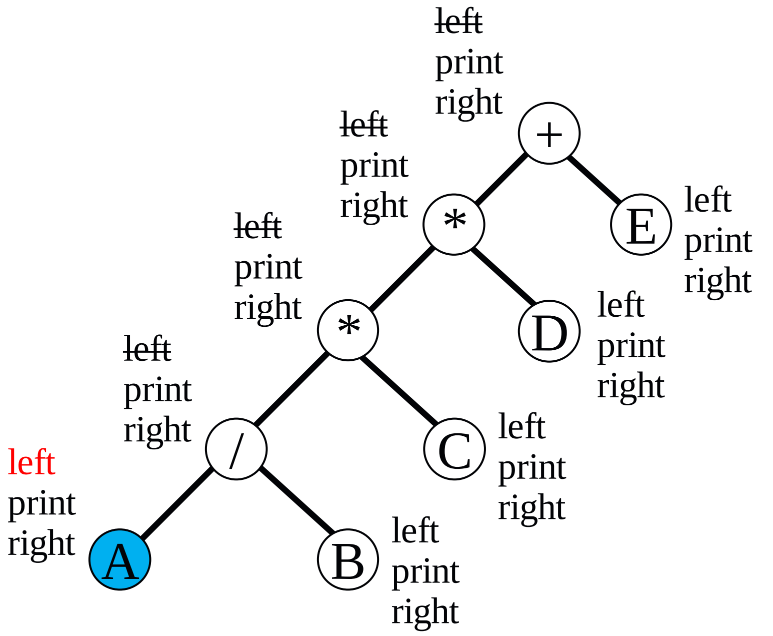
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



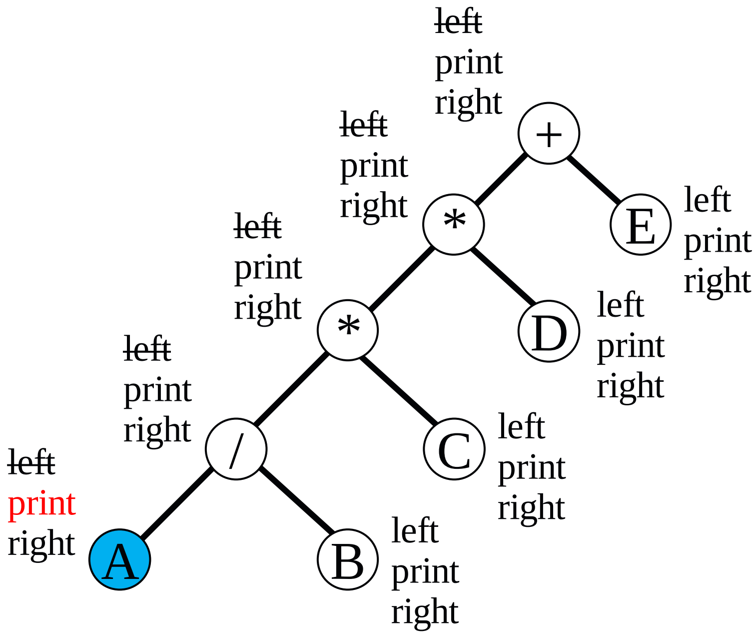
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



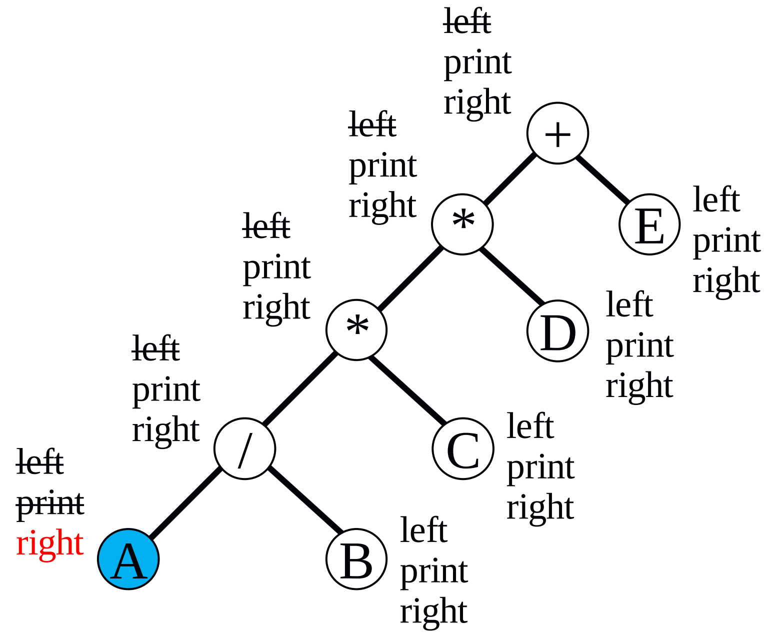
output:

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



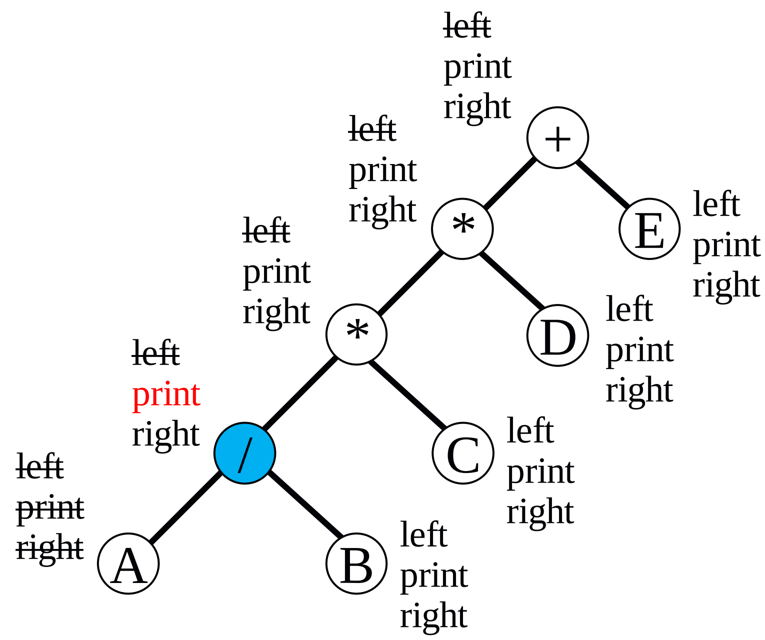
output: A

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



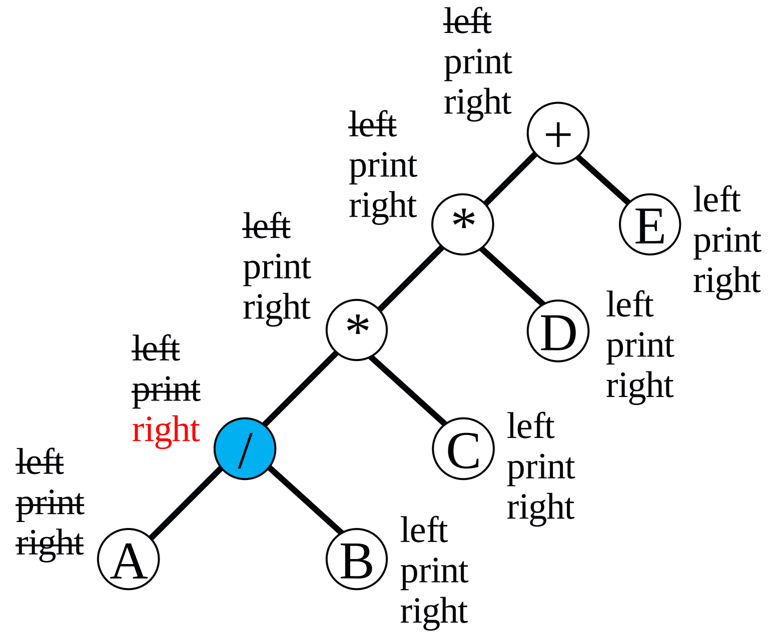
output: A

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



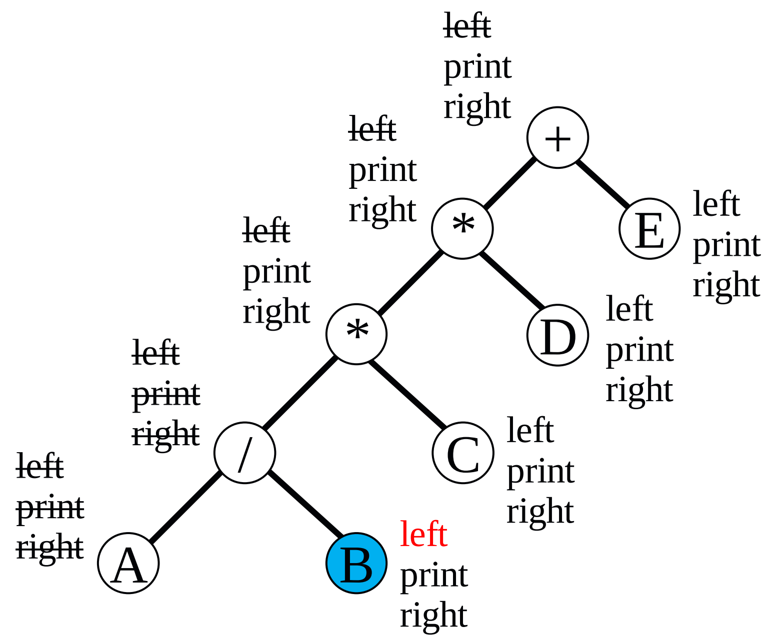
output: A/

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                    inorder (ptr -> left_child);
                    printf ("%c", ptr -> data);
                    inorder (ptr -> right_child);
        }
}
```



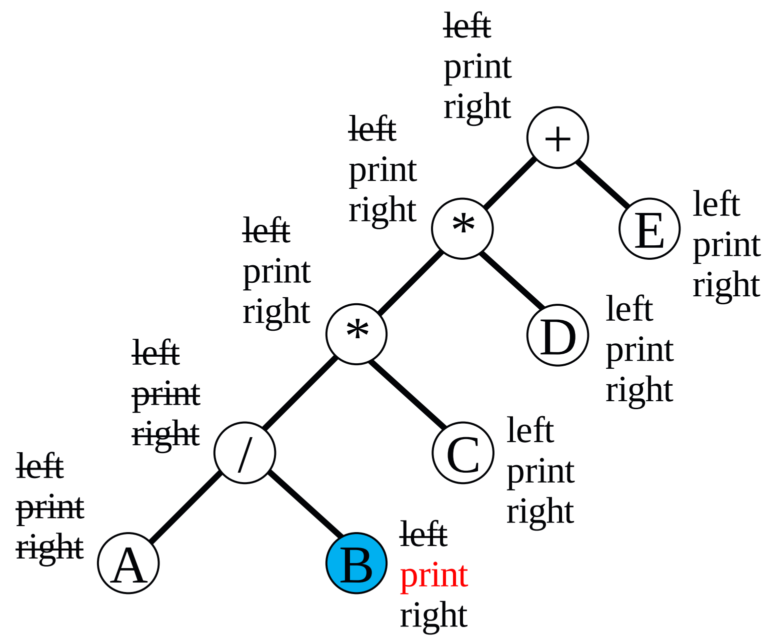
output: A/

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



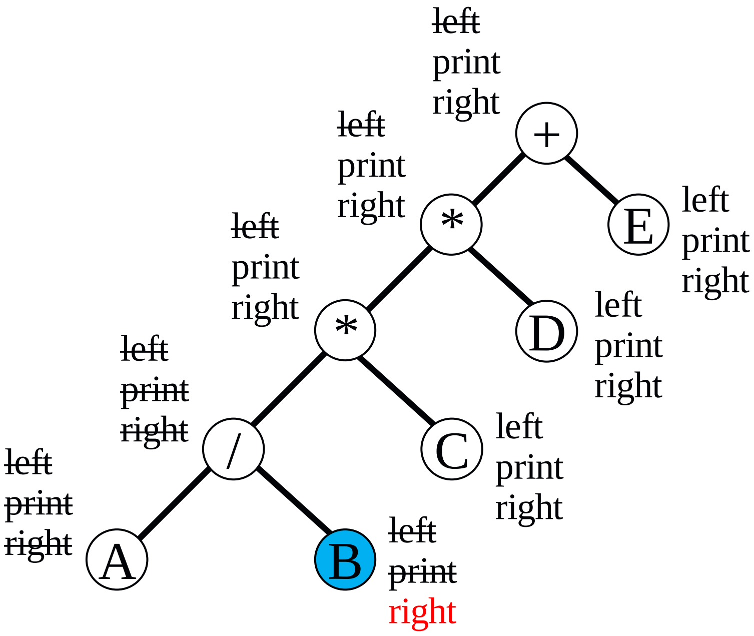
output: A/

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



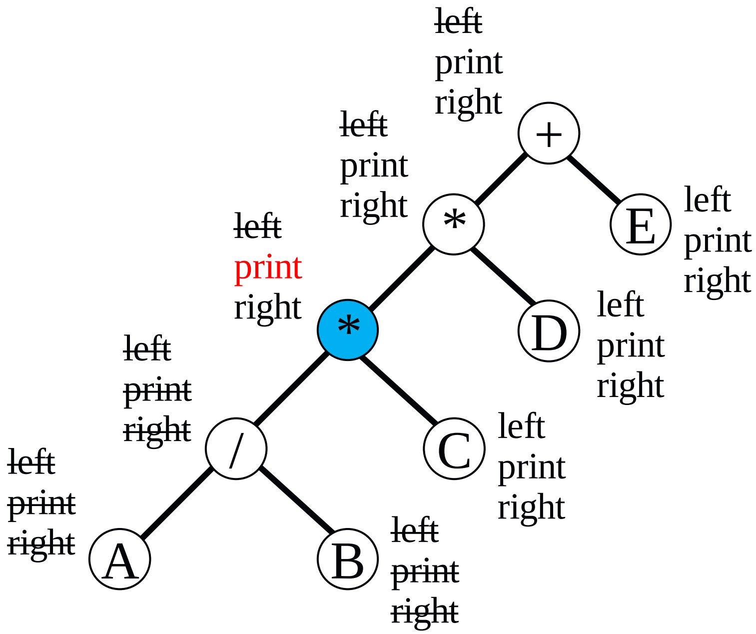
output: A/B

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



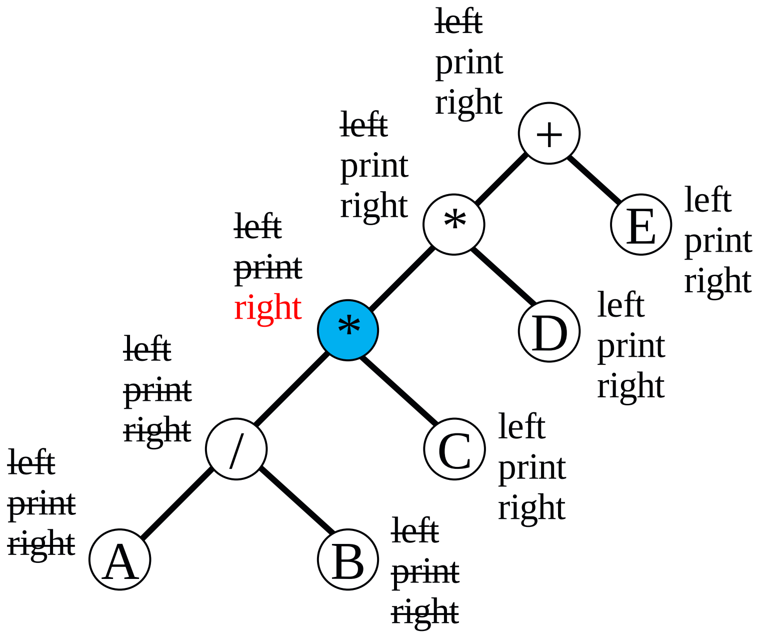
output: A/B

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



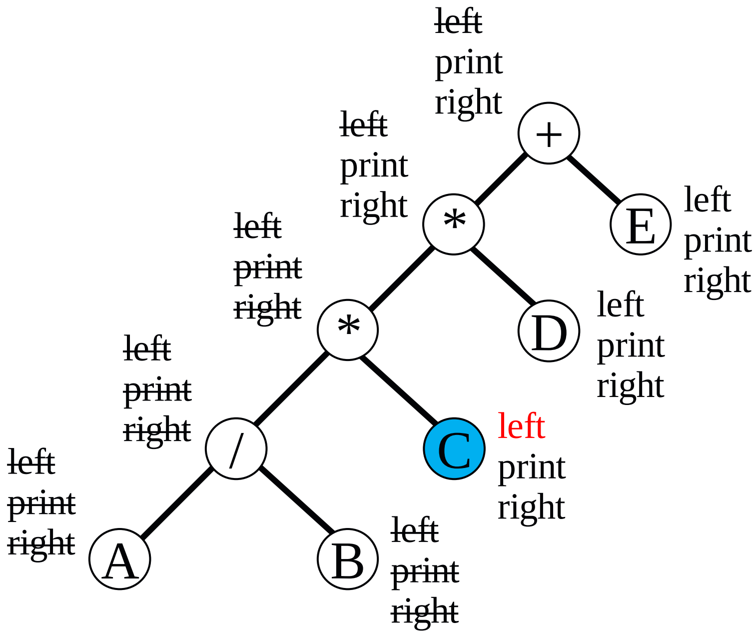
output: A/B*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



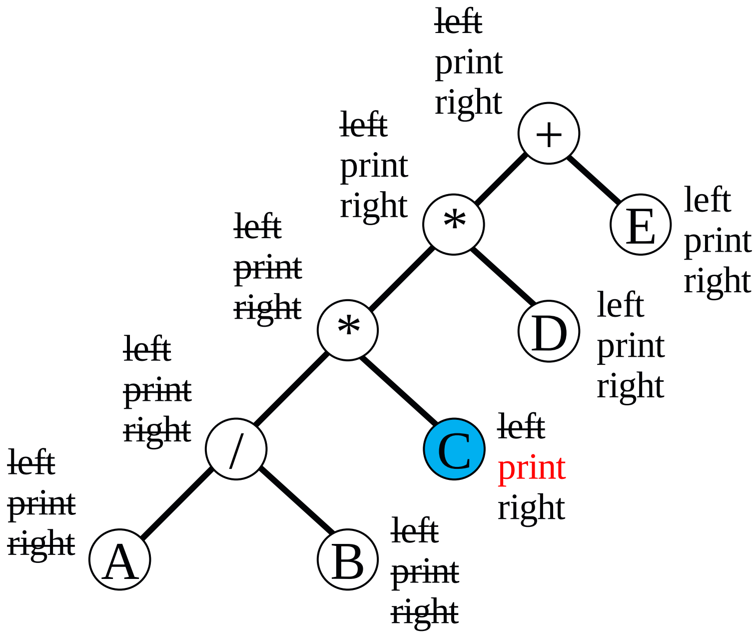
output: A/B*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



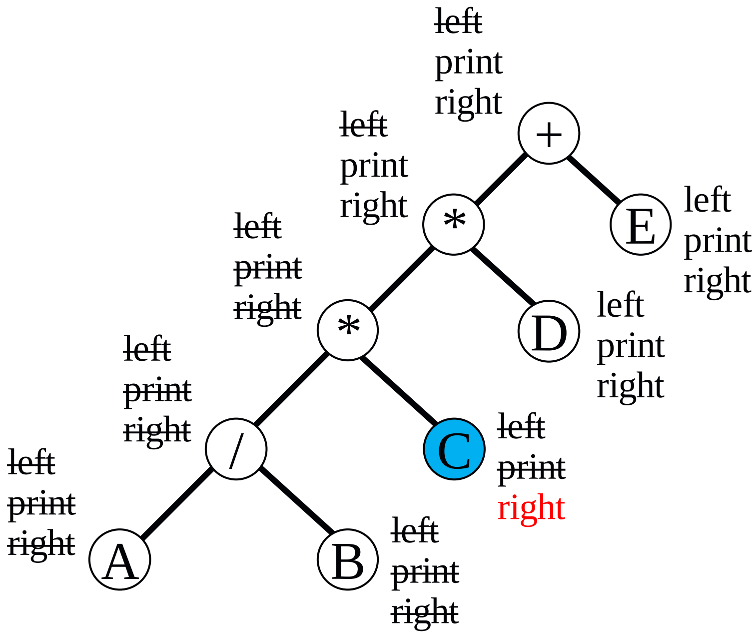
output: A/B*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



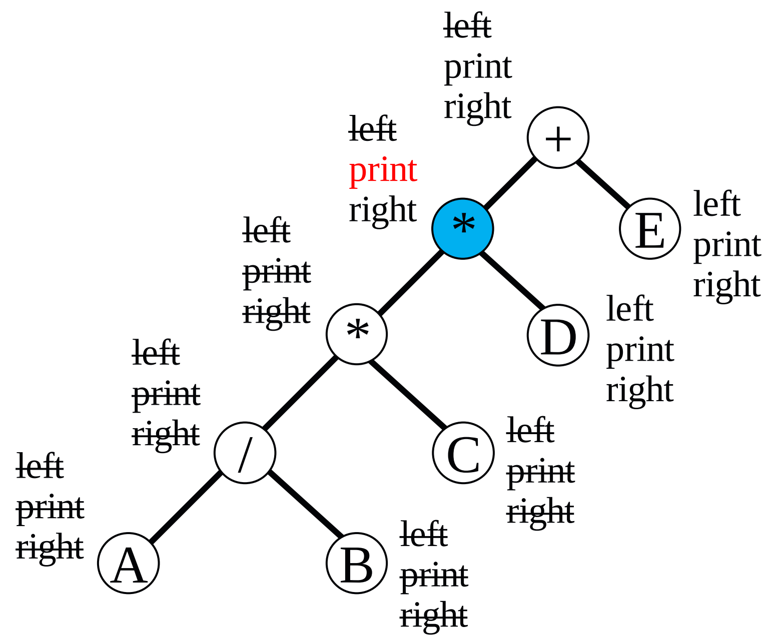
output: A/B*C

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



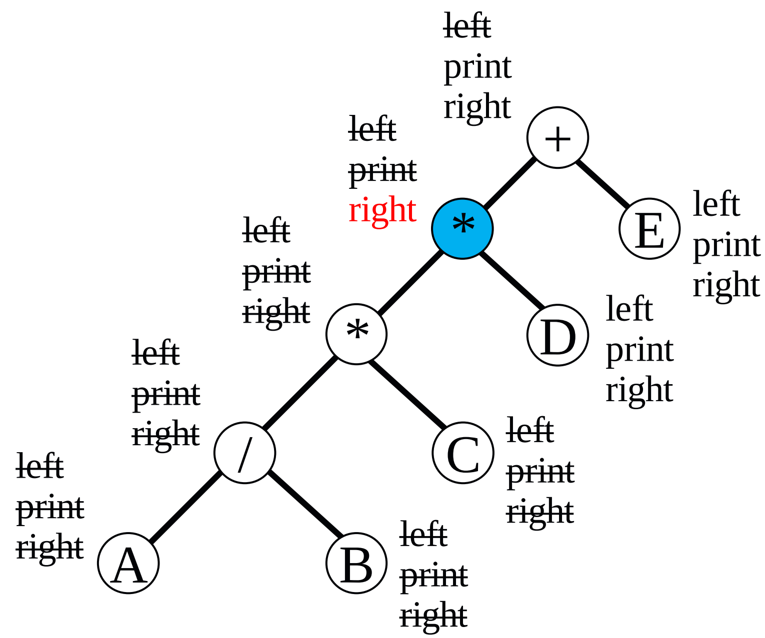
output: A/B*C

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



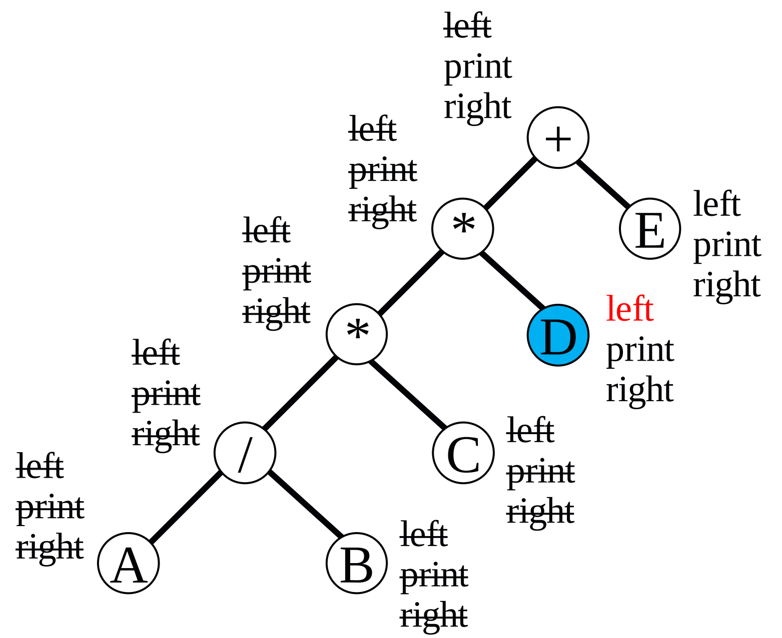
output: A/B*C*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



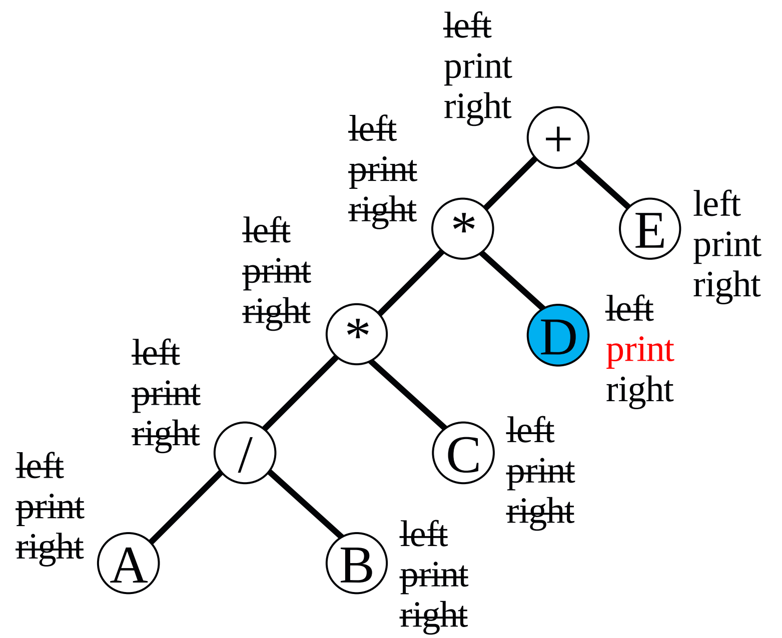
output: A/B*C*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



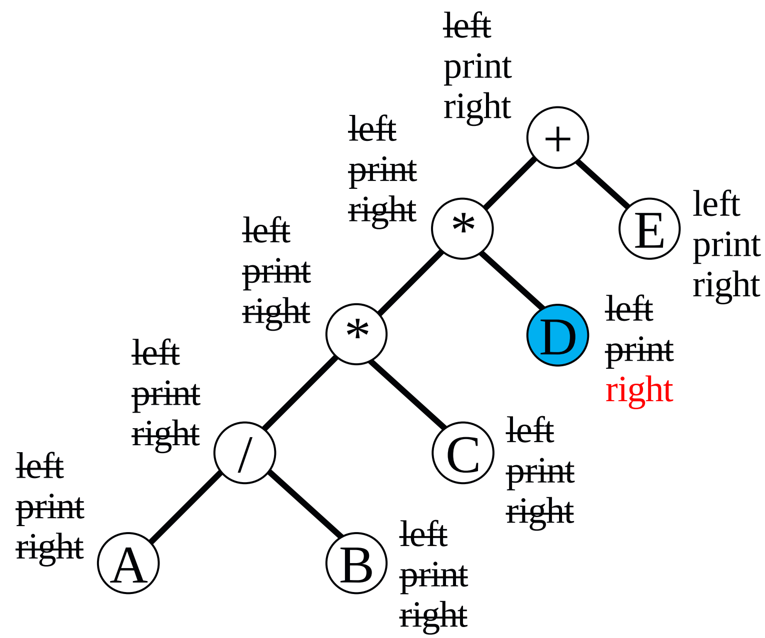
output: A/B*C*

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



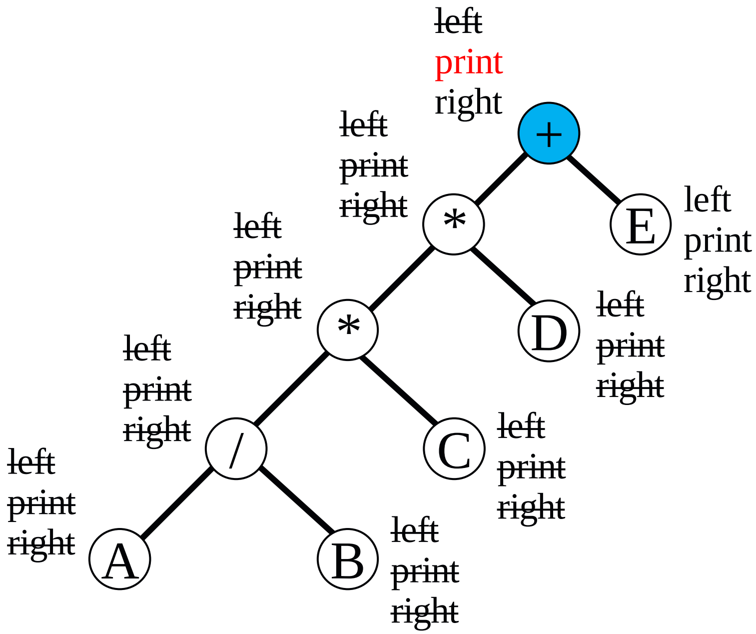
output: A/B*C*D

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



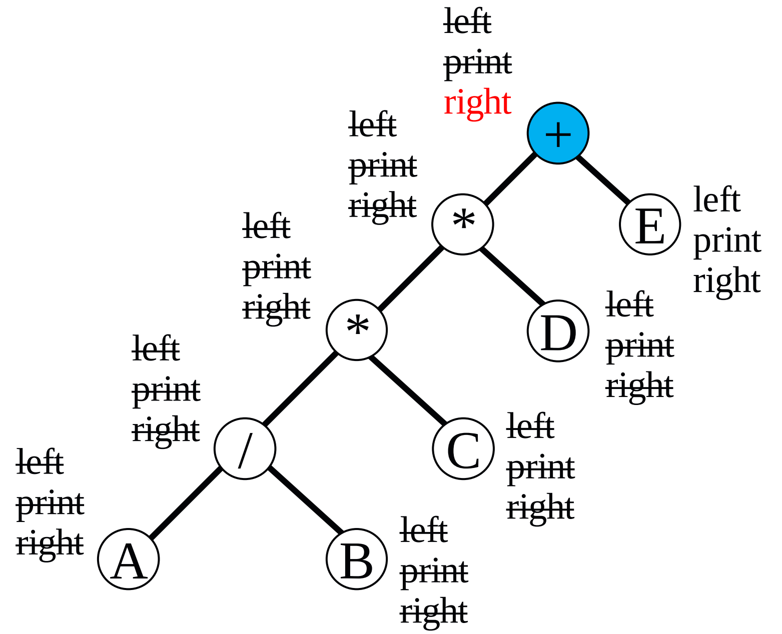
output: A/B*C*D

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



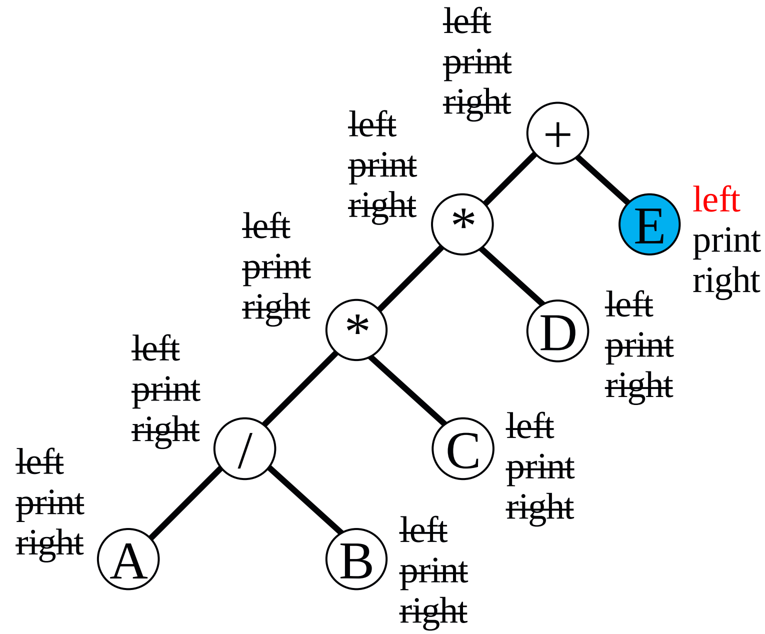
output: A/B*C*D+

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



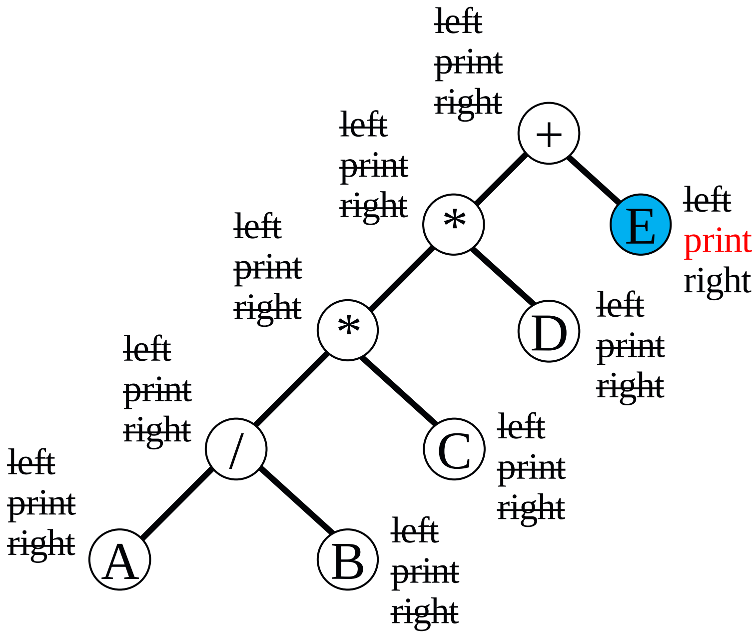
output: A/B*C*D+

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



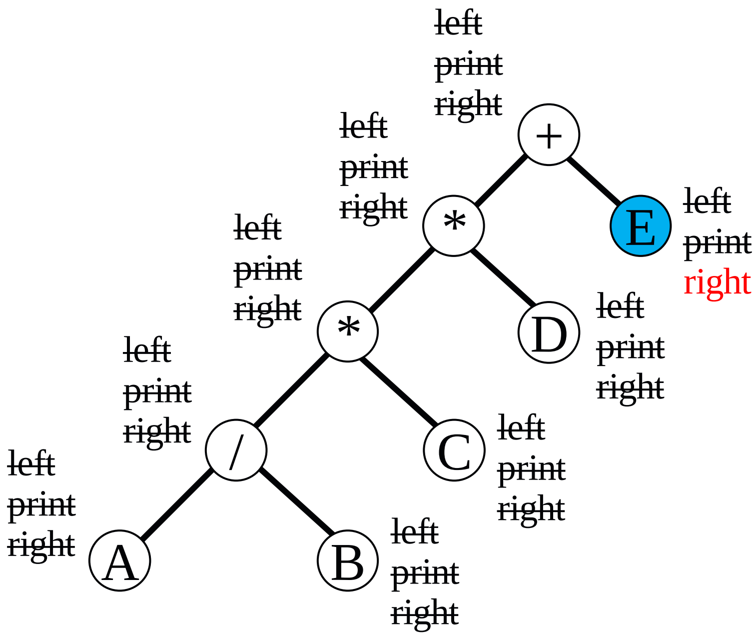
output: A/B*C*D+

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



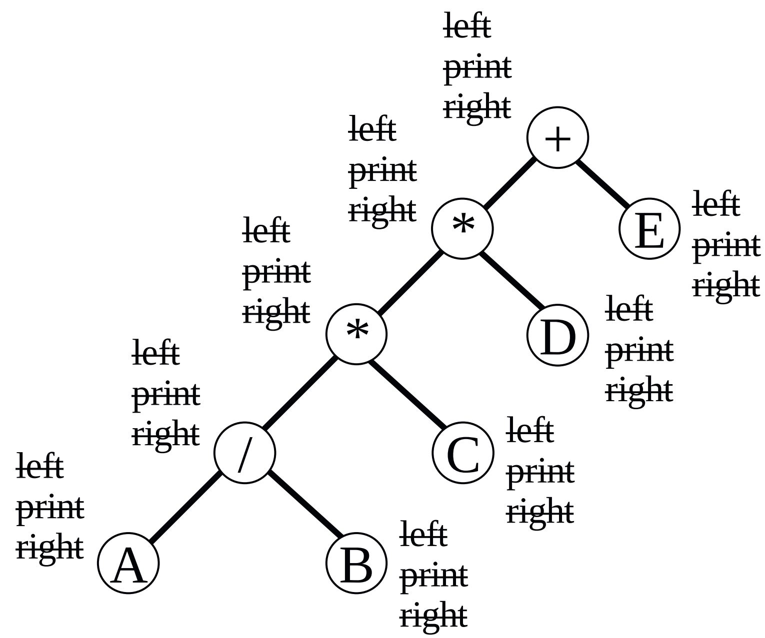
output: A/B*C*D+E

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



output: A/B*C*D+E

```
void  inorder (tree_pointer  ptr)
{ /* inorder tree traversal */
        if  (ptr)  {
                inorder (ptr -> left_child);
                printf ("%c", ptr -> data);
                inorder (ptr -> right_child);
        }
}
```



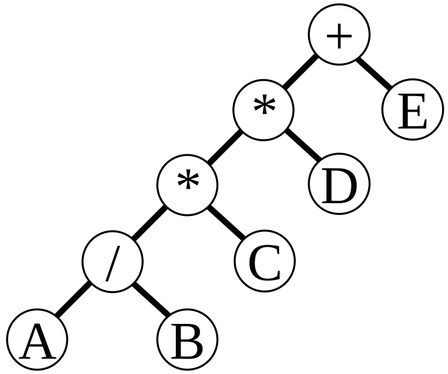
output: A/B*C*D+E

[Figure 5.17] Trace of Program 5.1

Call of inorder	Value in root	Action	inorder	in root	Value 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	

- **Program 5.2: Preorder traversal of a binary tree**

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
         }
 }
```

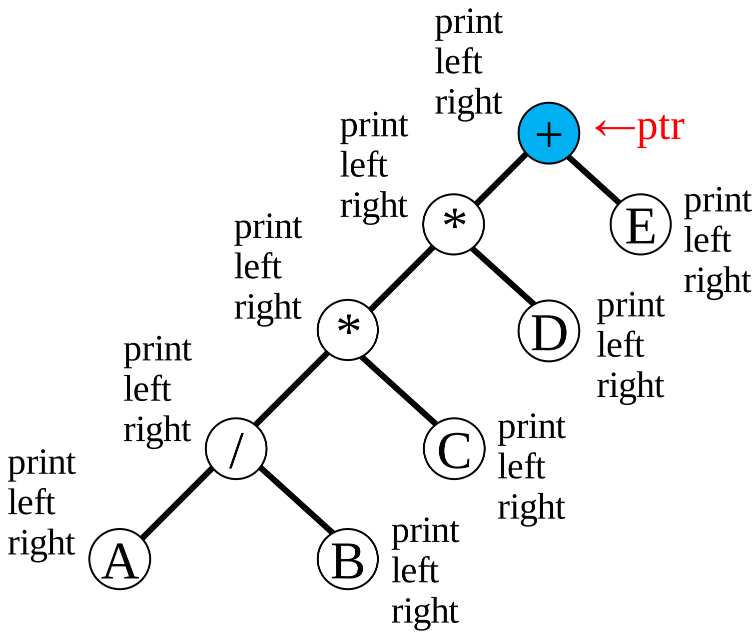


- The data fields of Figure 5.16 are output in the order :

  \+ * * / A B C D E

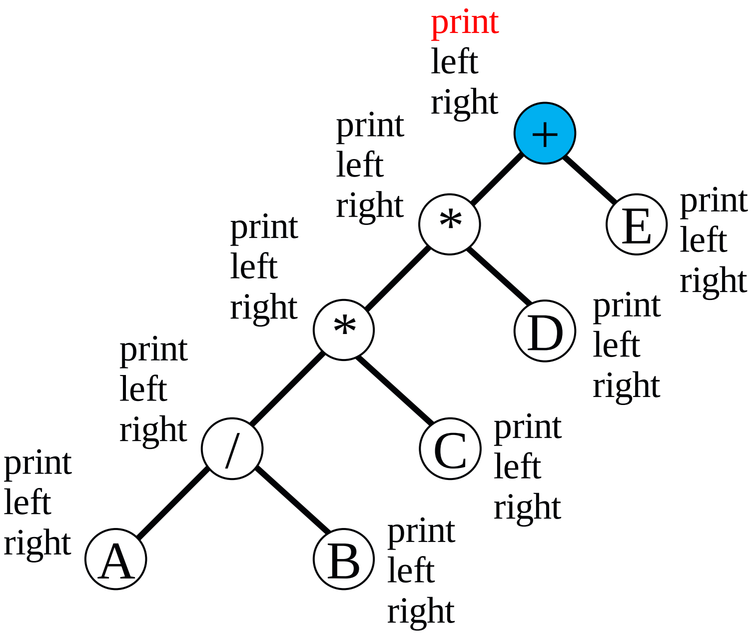
Sogang University

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



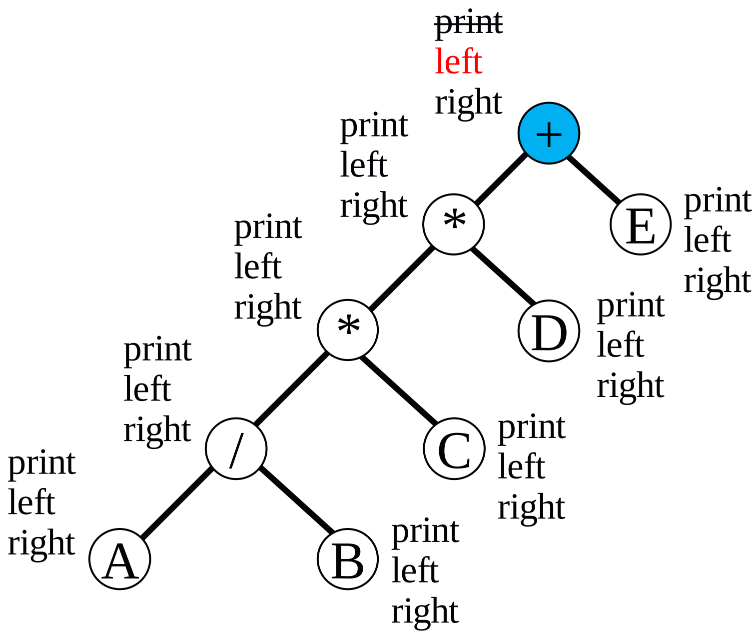
output:

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



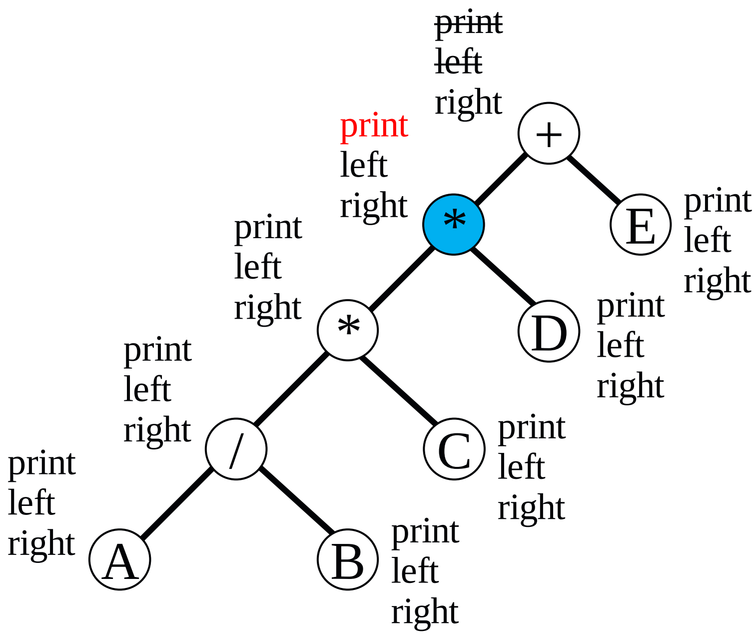
output: +

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



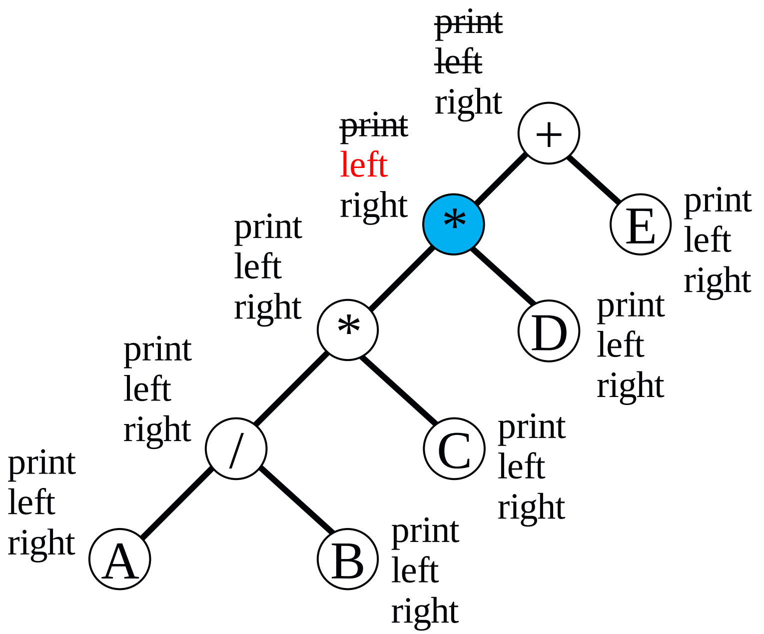
output: +

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



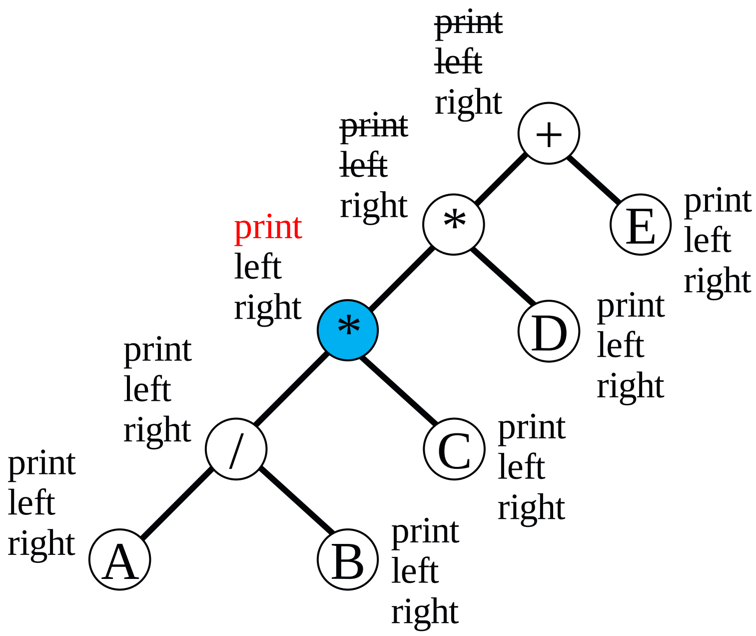
output: +*

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



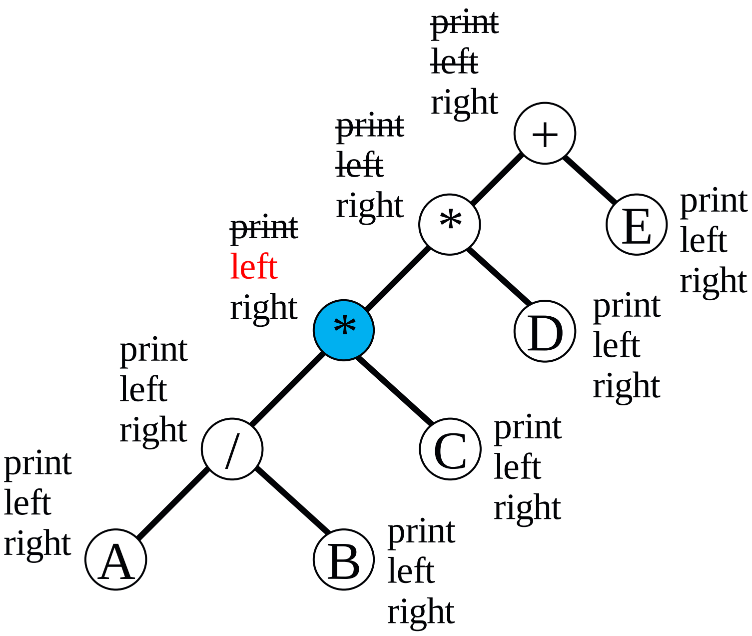
output: +*

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



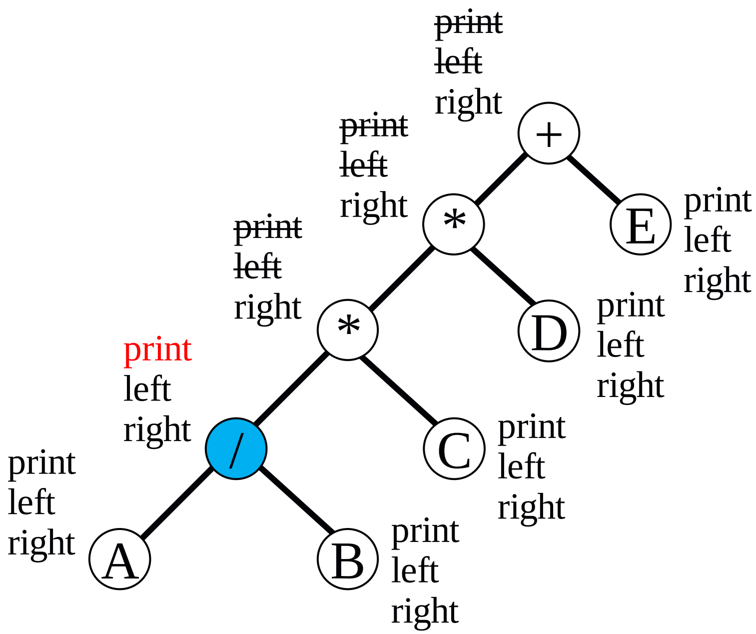
output: +**

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
         }
 }
```



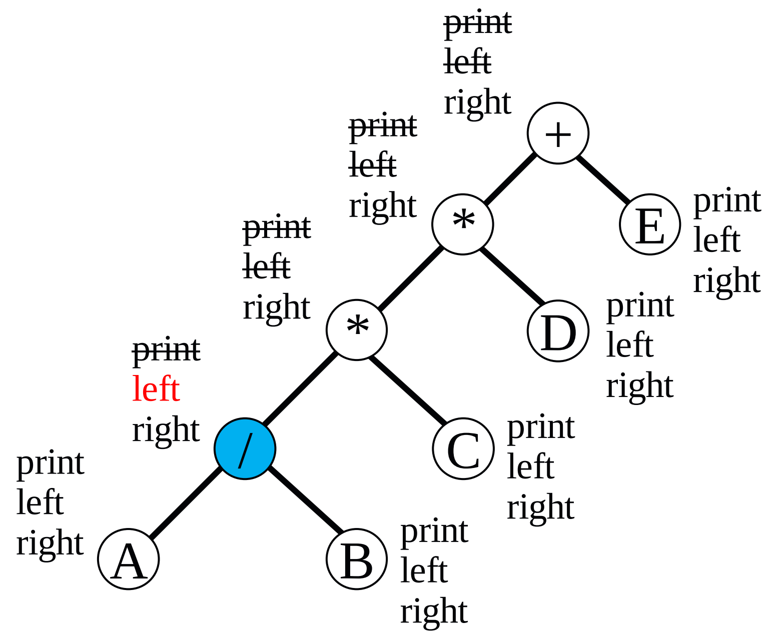
output: +**

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



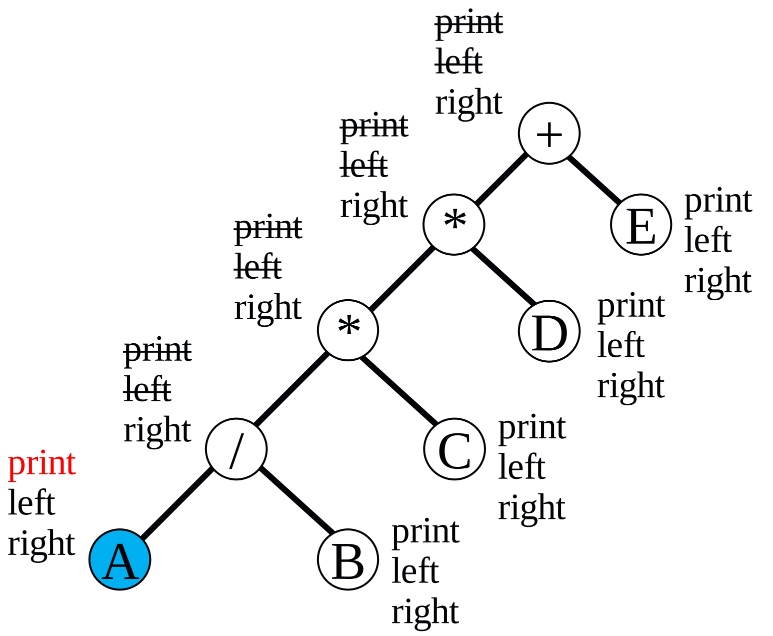
output: +**/

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
         }
 }
```



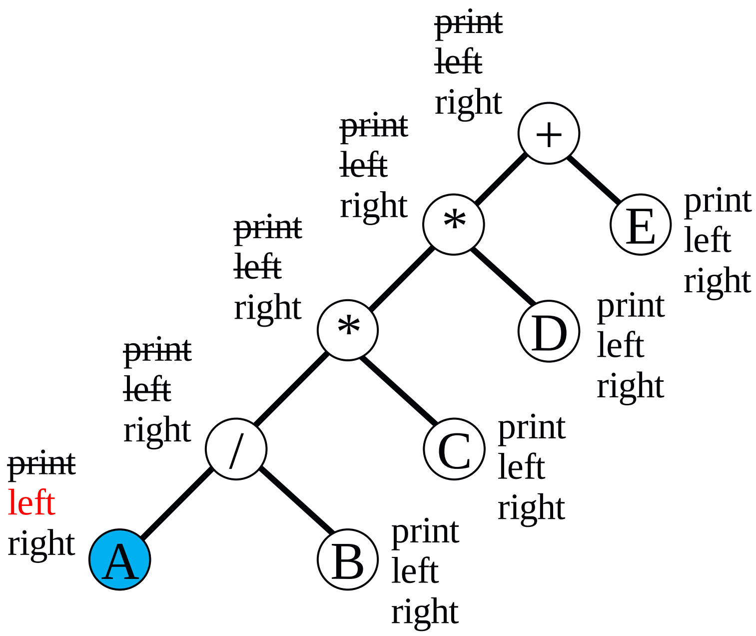
output: +**/

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



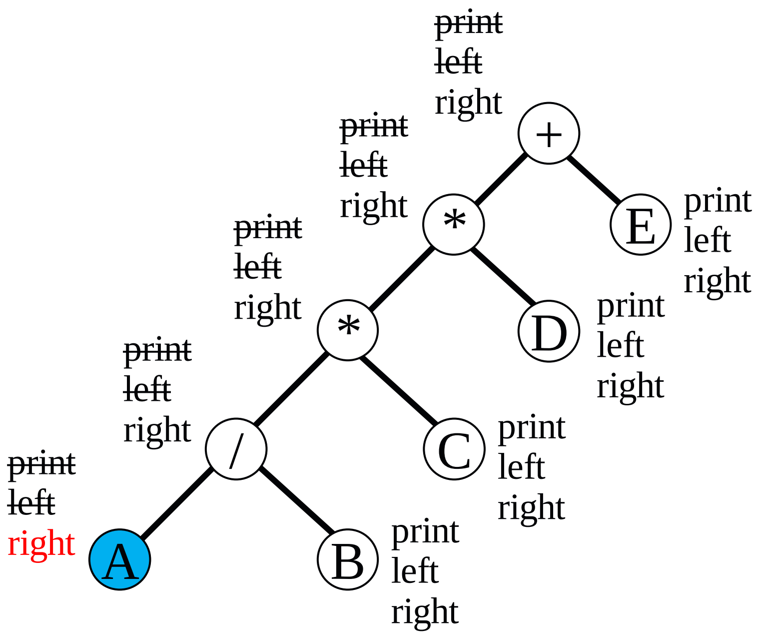
output: +**/A

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
         }
 }
```



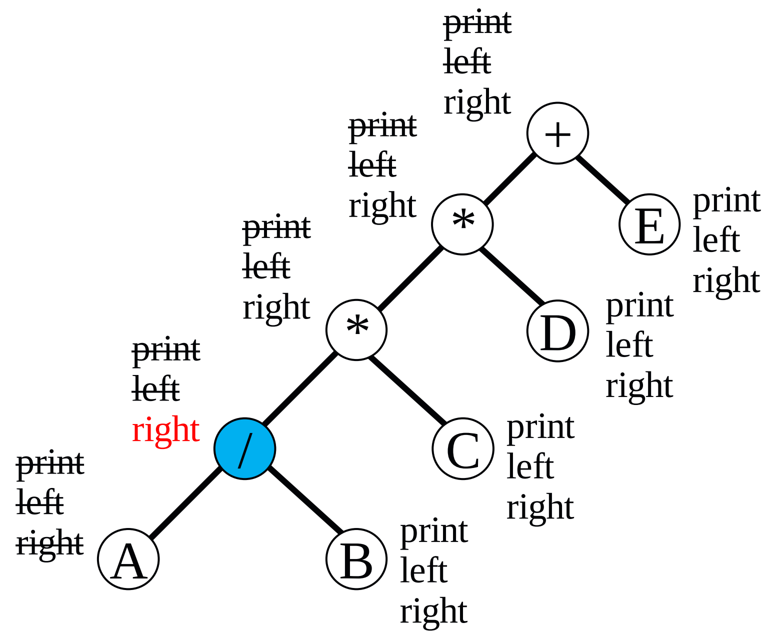
output: +**/A

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



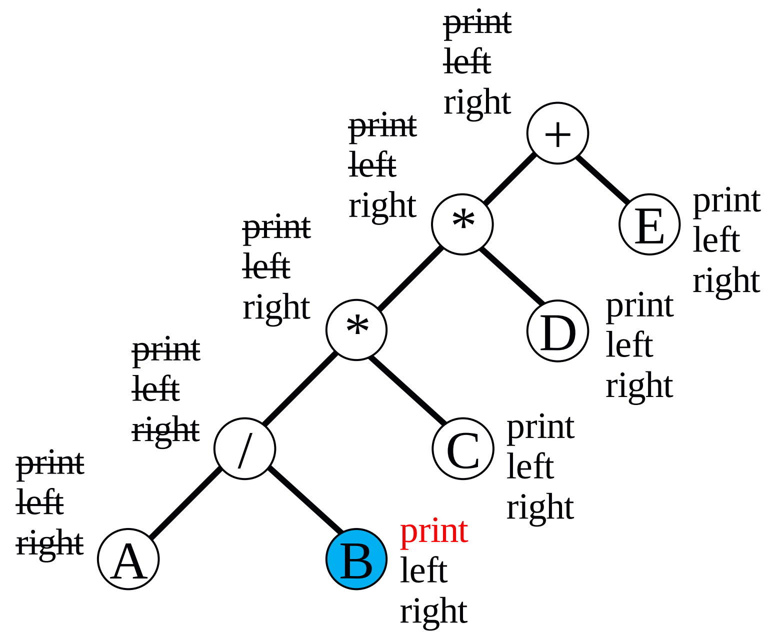
output: +**/A

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                    printf ("%c", ptr -> data);
                    preorder (ptr -> left_child);
                    preorder (ptr -> right_child);
         }
 }
```



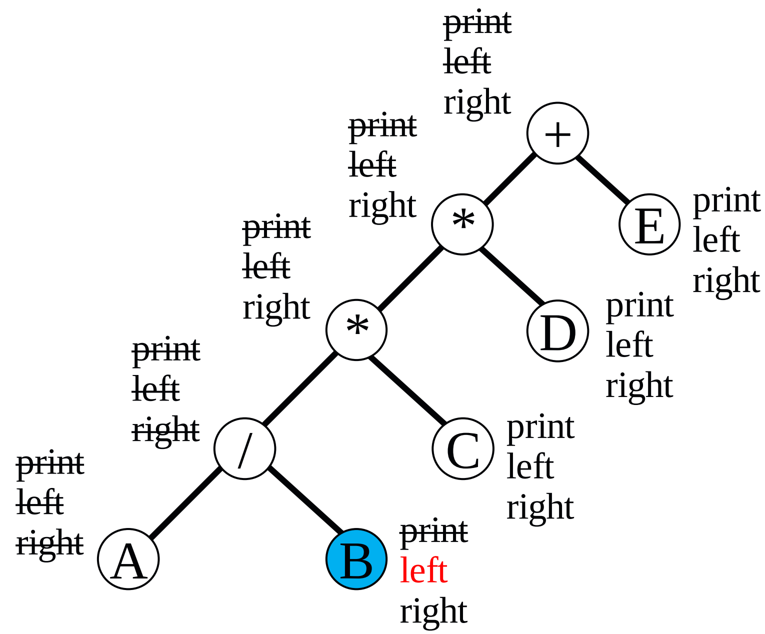
output: +**/A

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



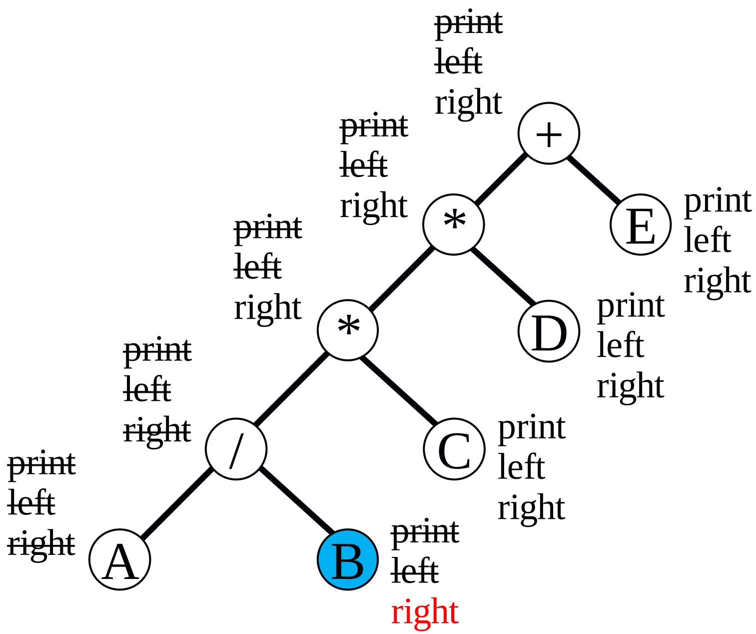
output: +**/AB

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



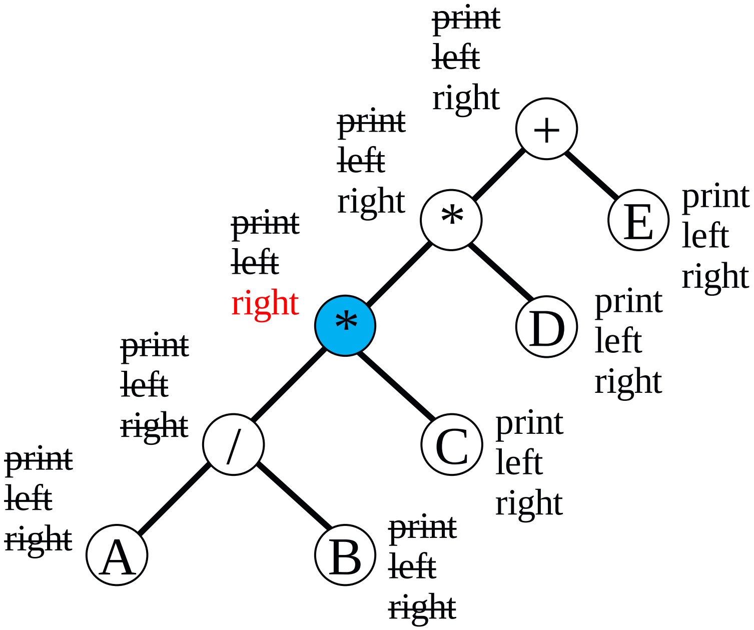
output: +**/AB

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



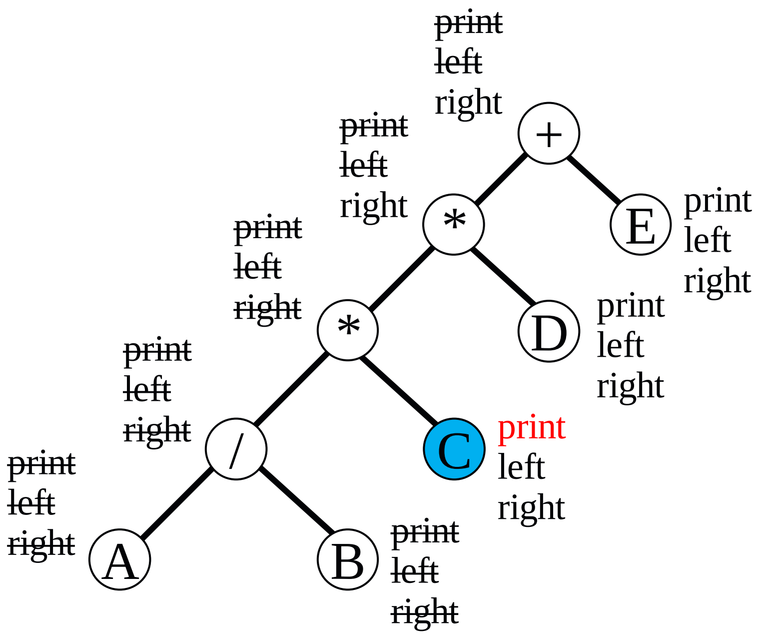
output: +**/AB

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



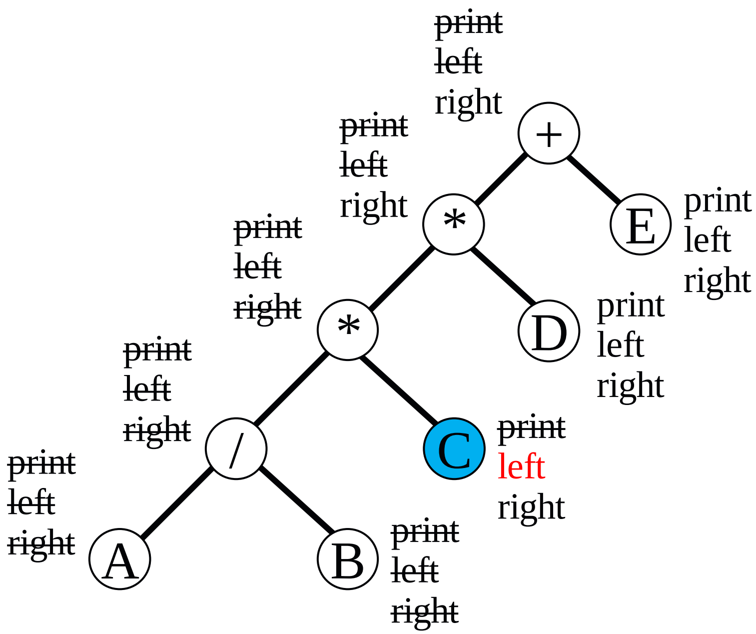
output: +**/AB

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



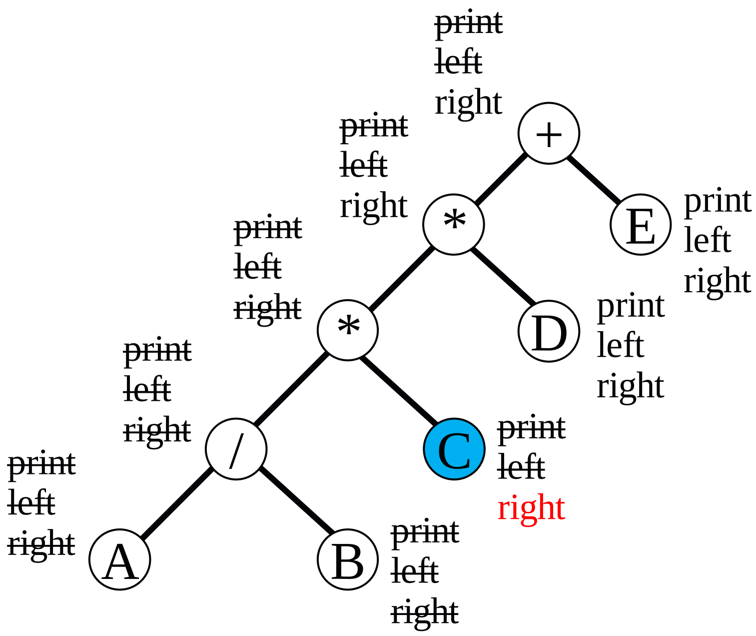
output: +**/ABC

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



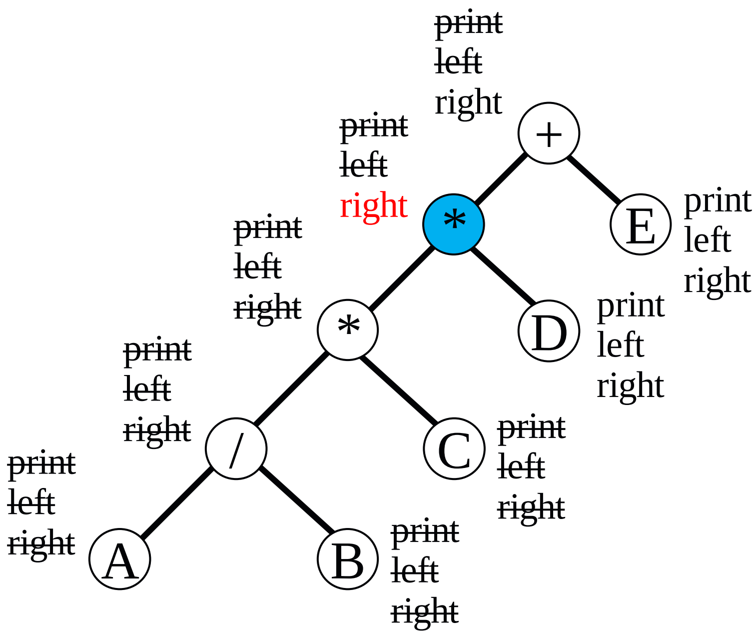
output: +**/ABC

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



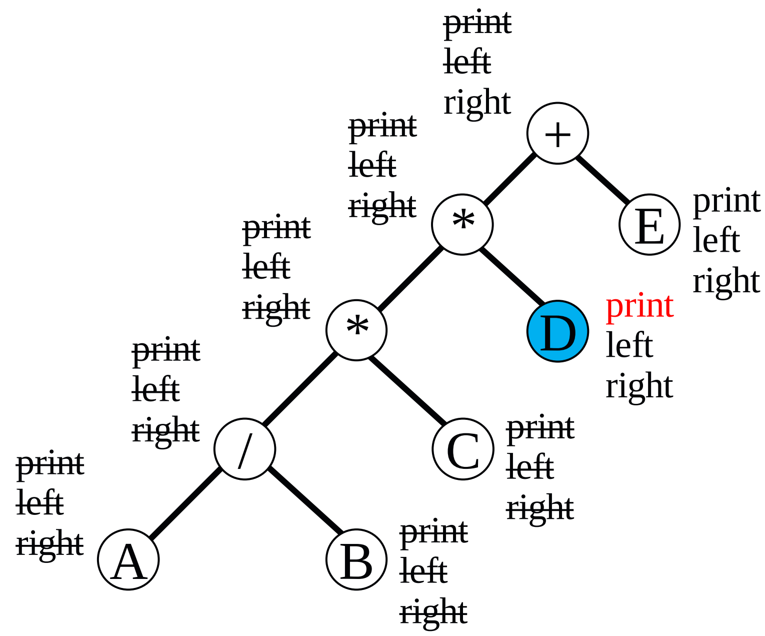
output: +**/ABC

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



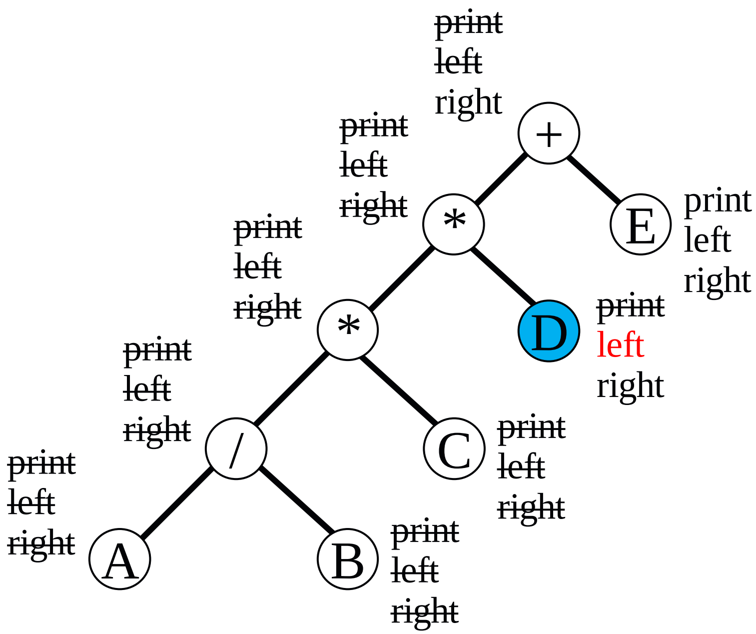
output: +**/ABC

```c
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
 }
```



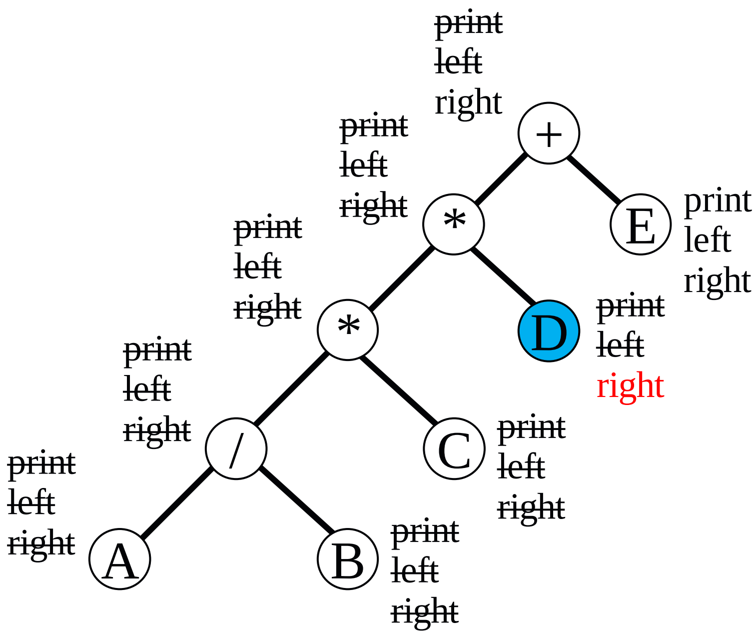
output: +**/ABCD

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



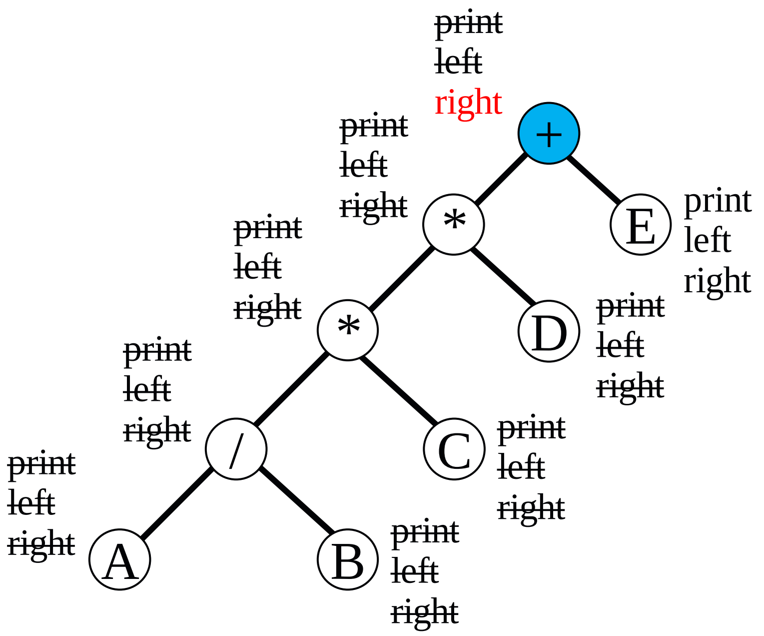
output: +**/ABCD

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



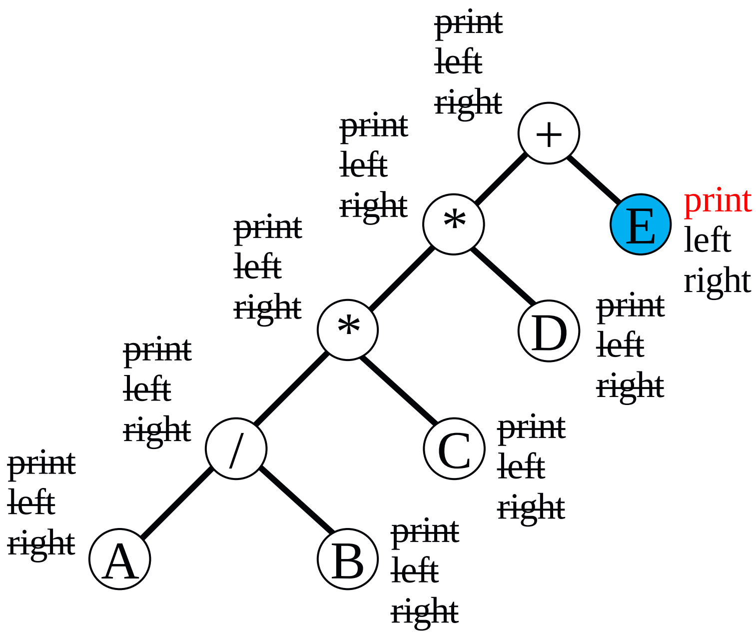
output: +**/ABCD

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



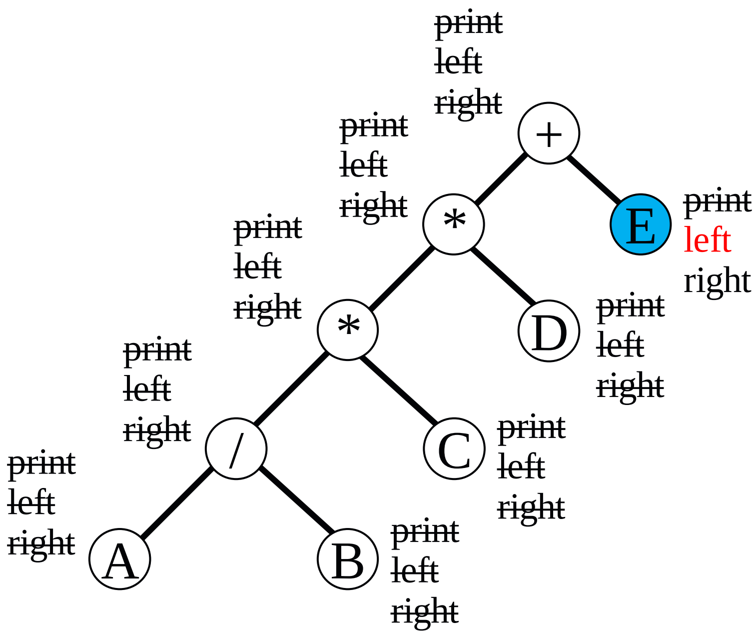
output: +**/ABCD

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



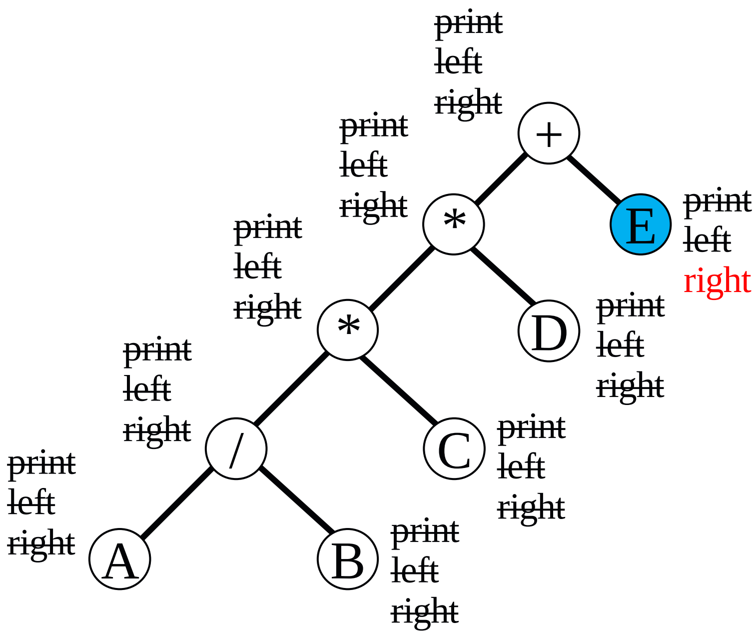
output: +**/ABCDE

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



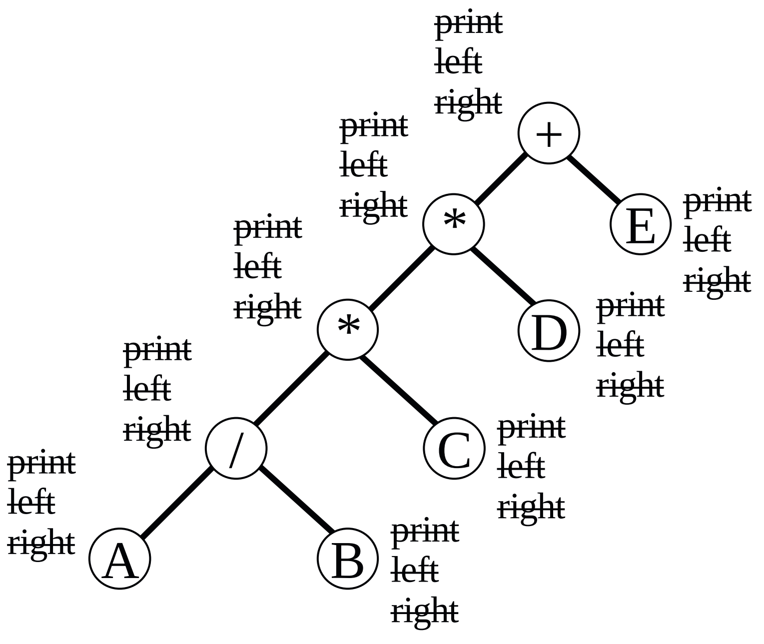
output: +**/ABCDE

```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



output: +**/ABCDE

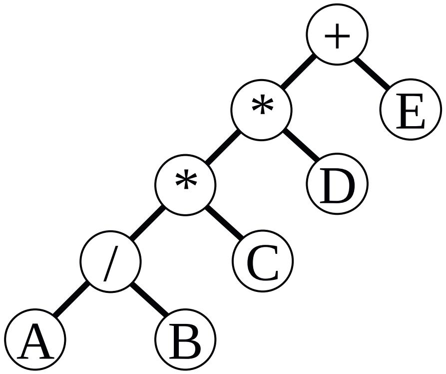
```
void  preorder (tree_pointer  ptr)
{ /* preorder tree traversal */
        if  (ptr)  {
                printf ("%c", ptr -> data);
                preorder (ptr -> left_child);
                preorder (ptr -> right_child);
        }
}
```



output: +**/ABCDE

Program 5.3: Postorder traversal of a binary tree

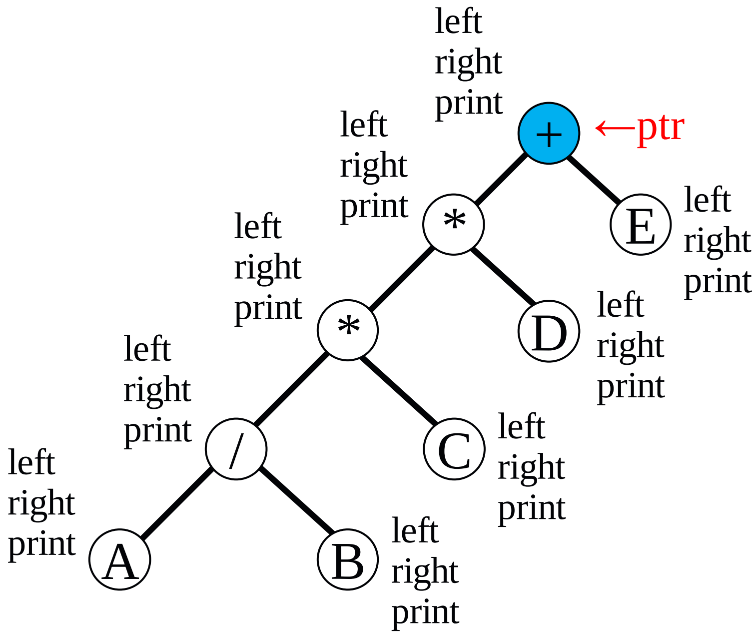
```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



- The data fields of Figure 5.16 are output in the order :

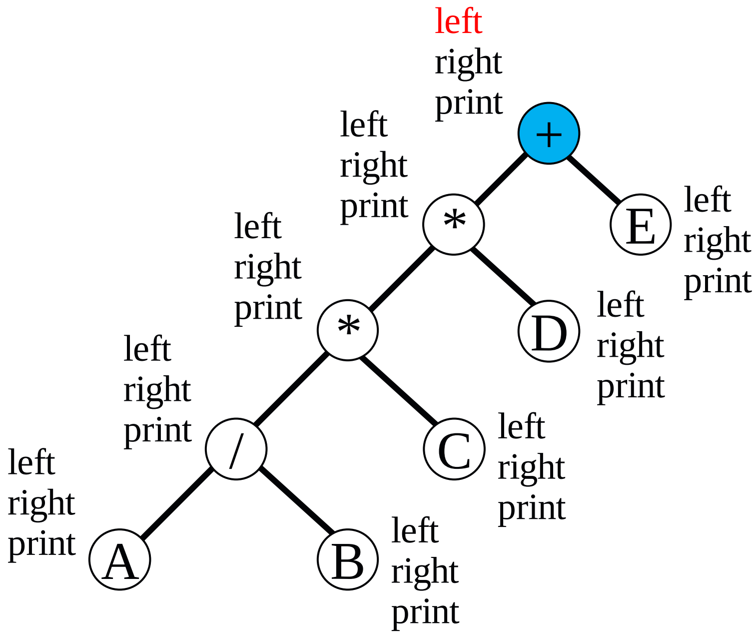
  A B / C * D * E +

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



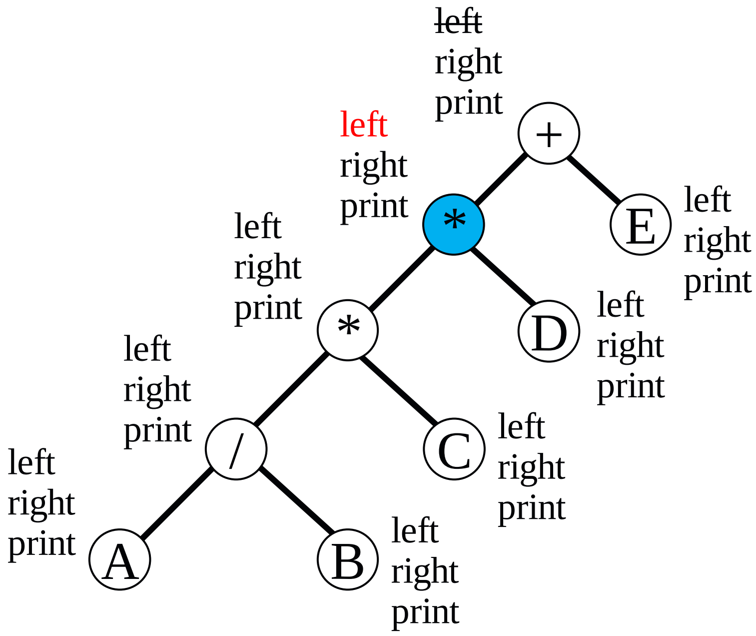
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



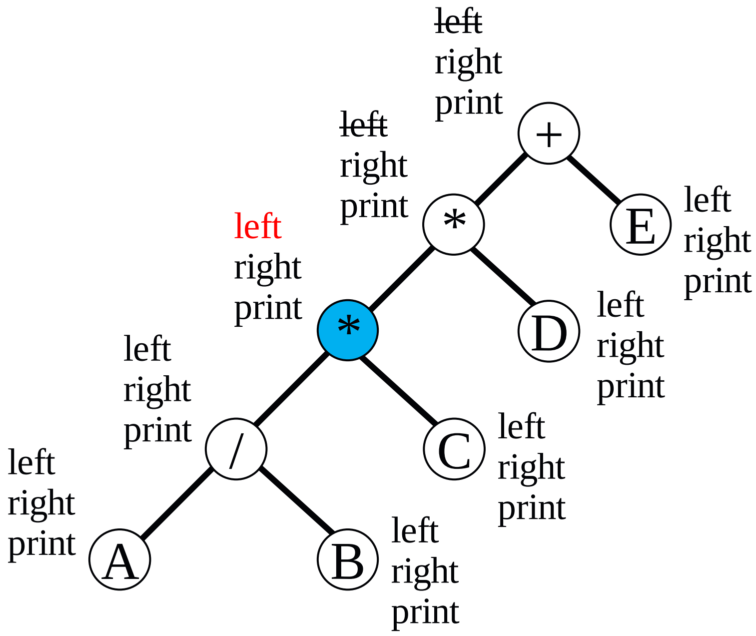
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



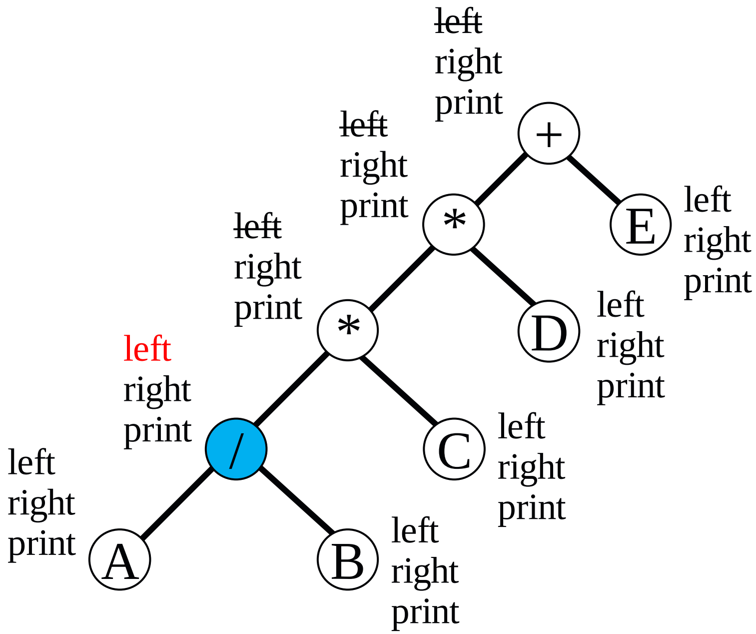
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



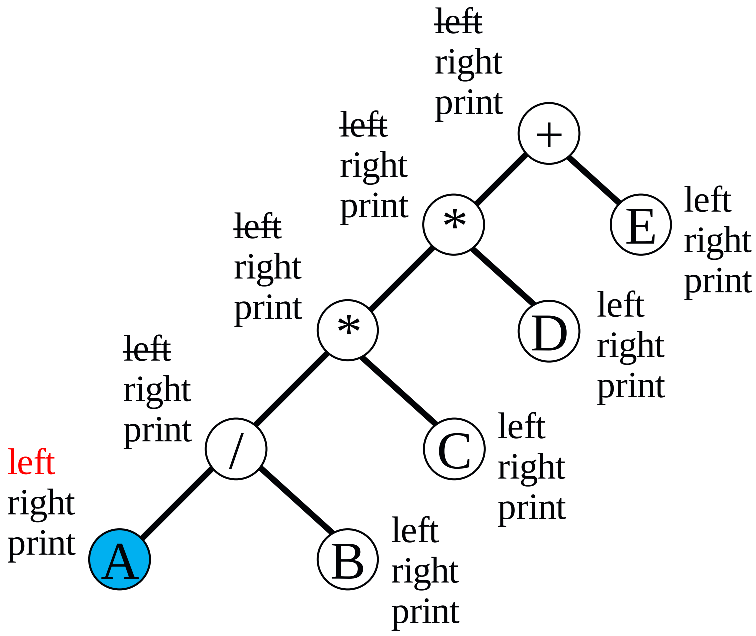
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



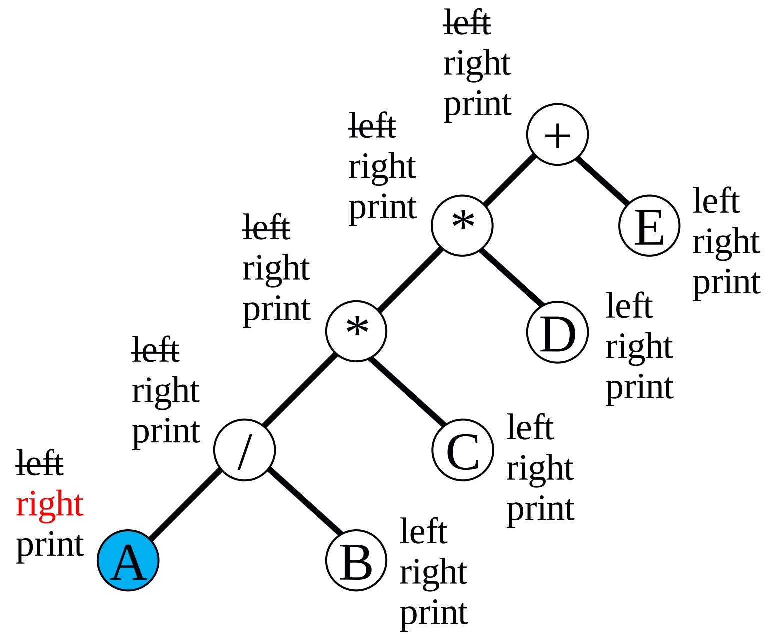
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
              postorder (ptr -> left_child);
              postorder (ptr -> right_child);
              printf ("%c", ptr -> data);
      }
}
```



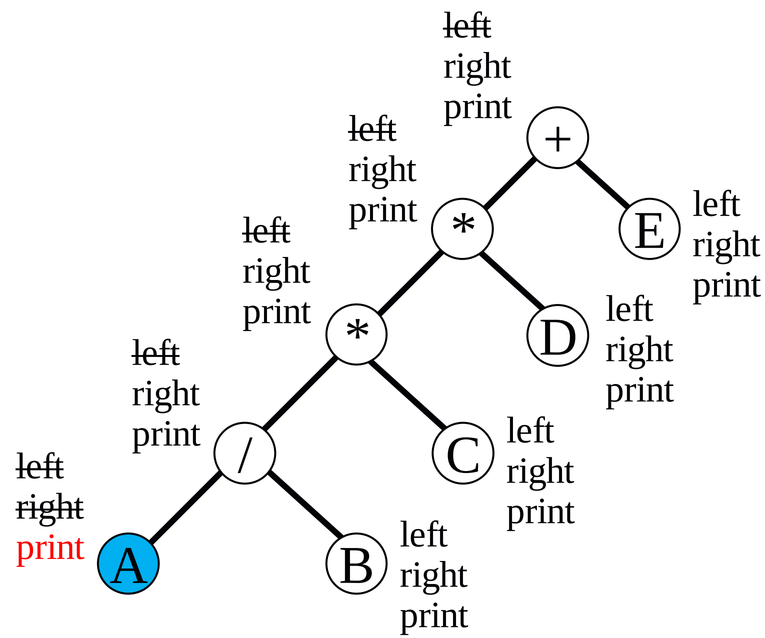
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



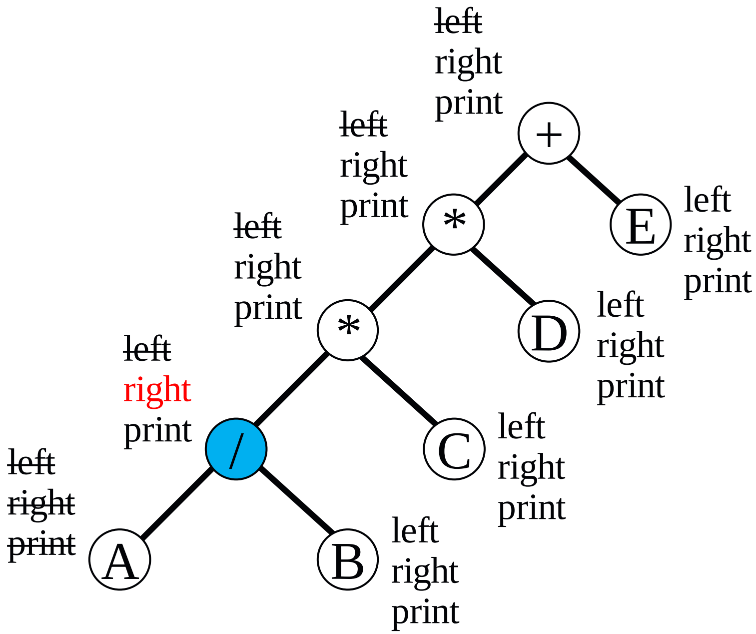
output:

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
               postorder (ptr -> left_child);
               postorder (ptr -> right_child);
               printf ("%c", ptr -> data);
      }
}
```



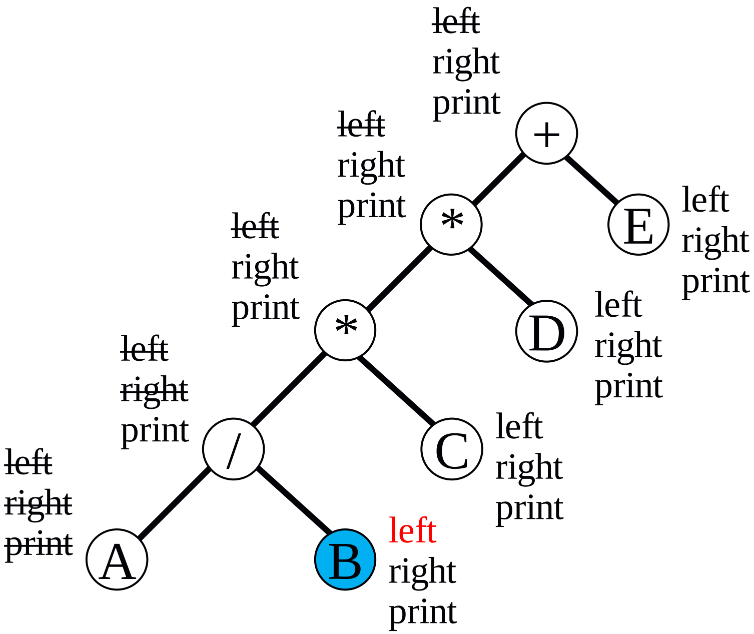
output: A

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                    postorder (ptr -> left_child);
                    postorder (ptr -> right_child);
                    printf ("%c", ptr -> data);
      }
}
```



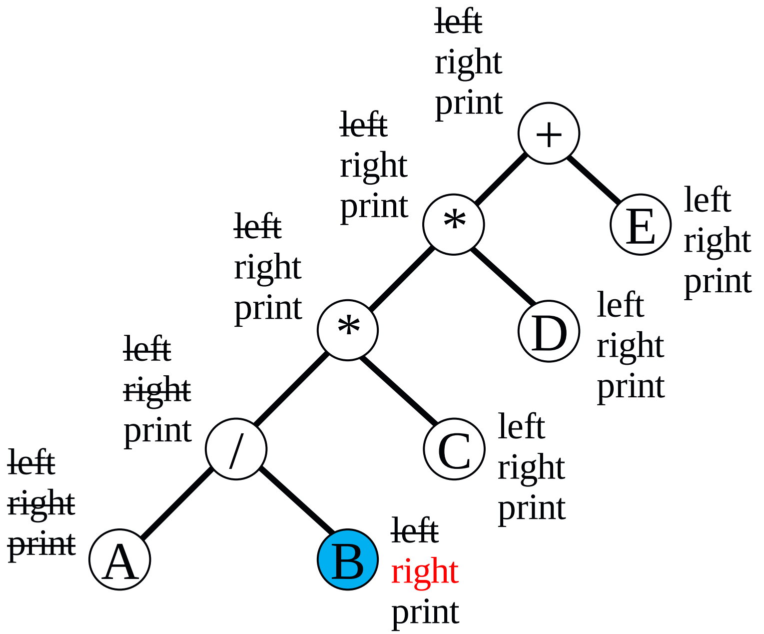
output: A

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



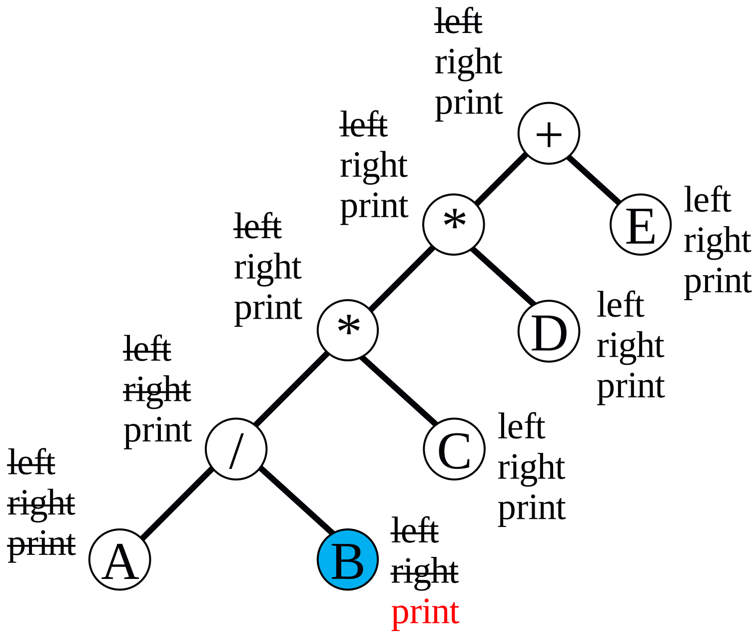
output: A

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



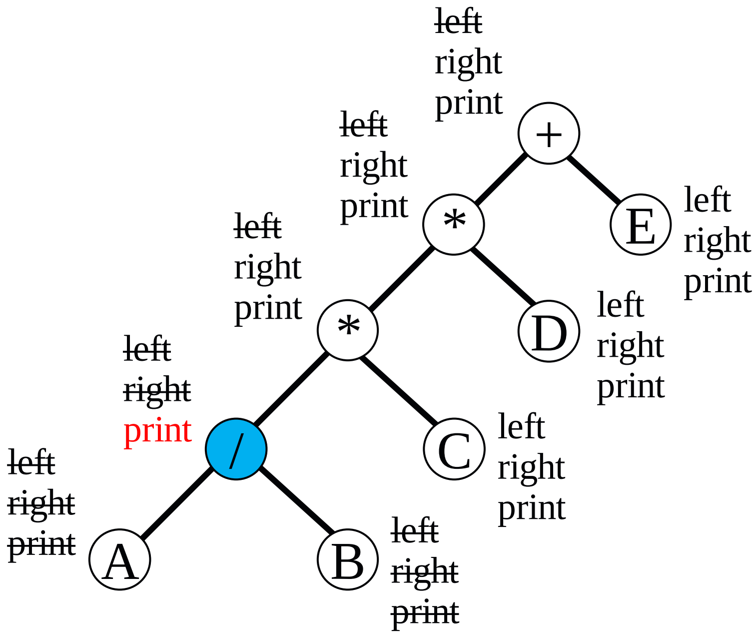
output: A

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
              postorder (ptr -> left_child);
              postorder (ptr -> right_child);
              printf ("%c", ptr -> data);
      }
}
```



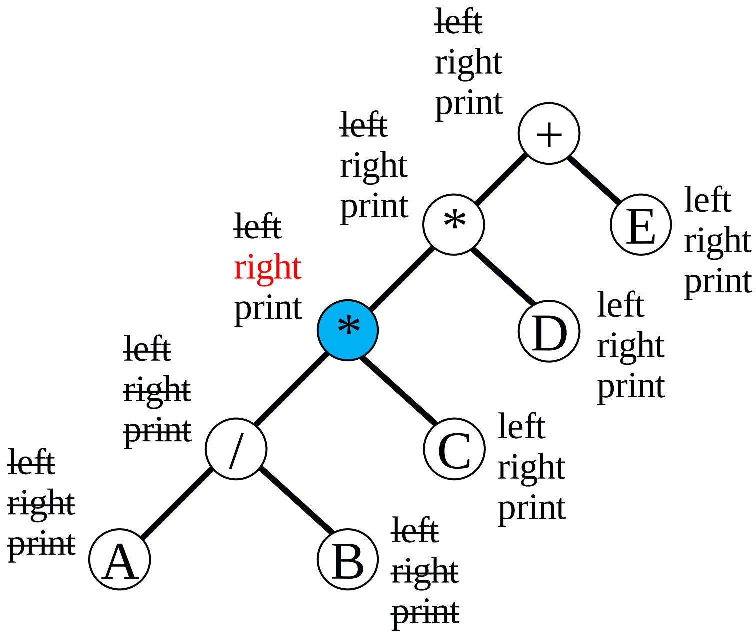
output: AB

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



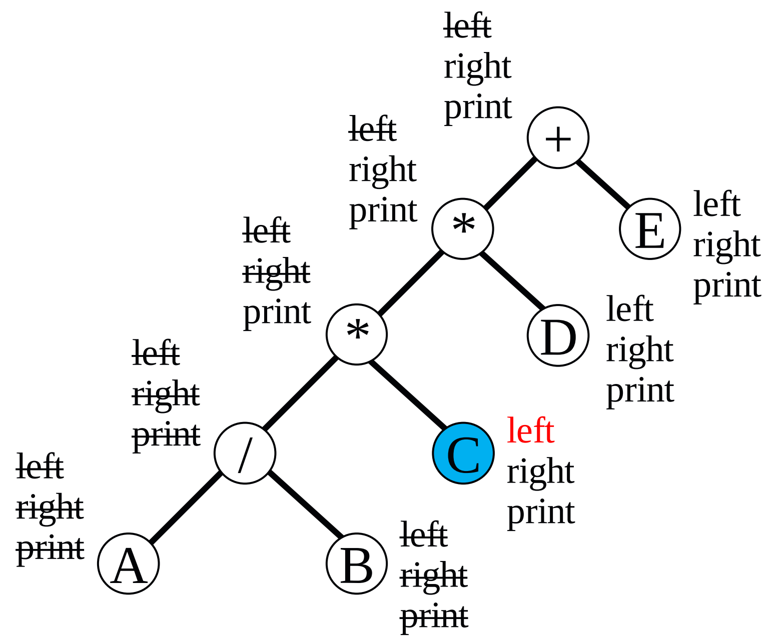
output: AB/

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



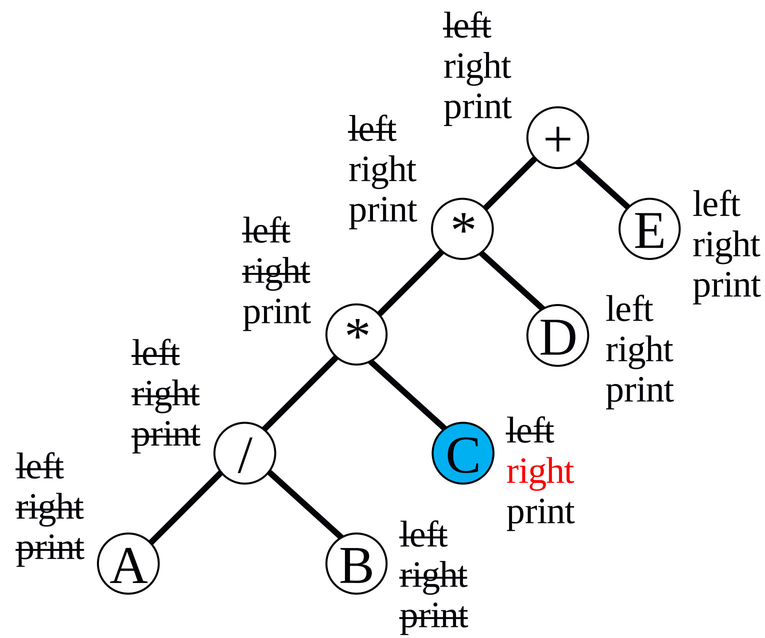
output: AB/

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
        if  (ptr)  {
                        postorder (ptr -> left_child);
                        postorder (ptr -> right_child);
                        printf ("%c", ptr -> data);
        }
}
```



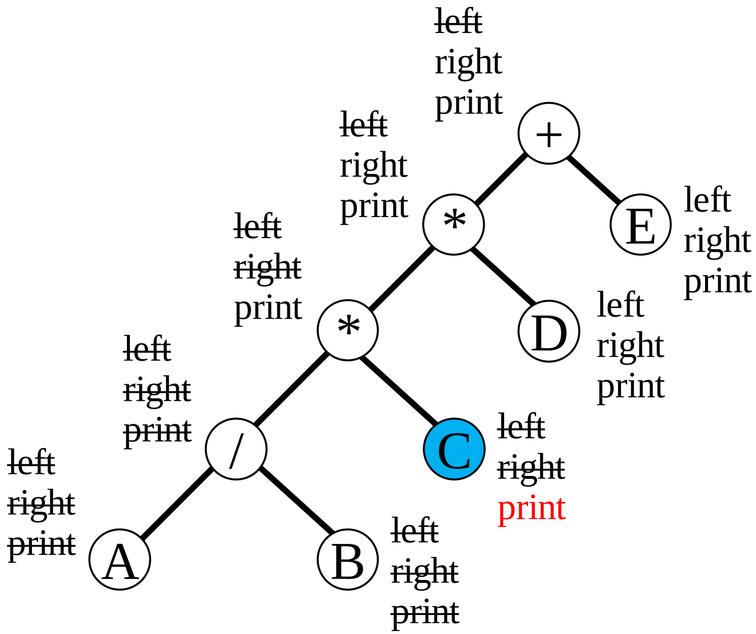
output: AB/

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                postorder (ptr -> left_child);
                postorder (ptr -> right_child);
                printf ("%c", ptr -> data);
      }
}
```



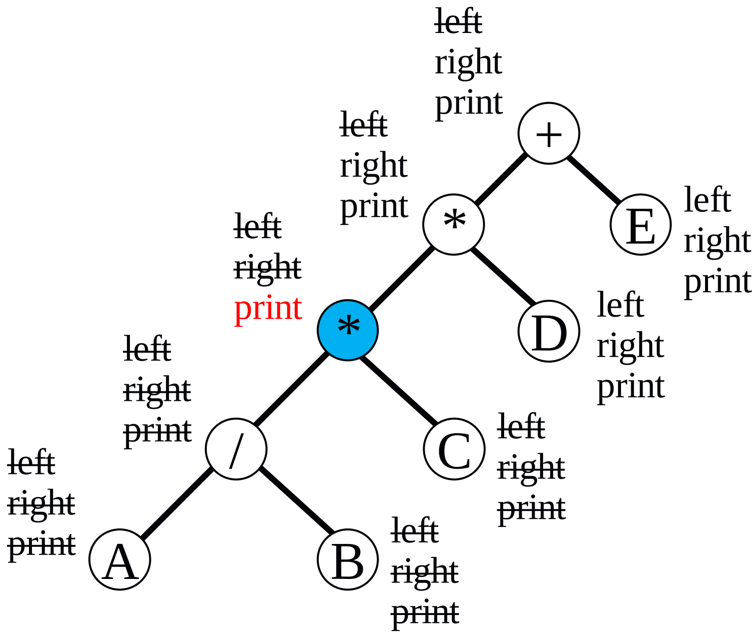
output: AB/

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



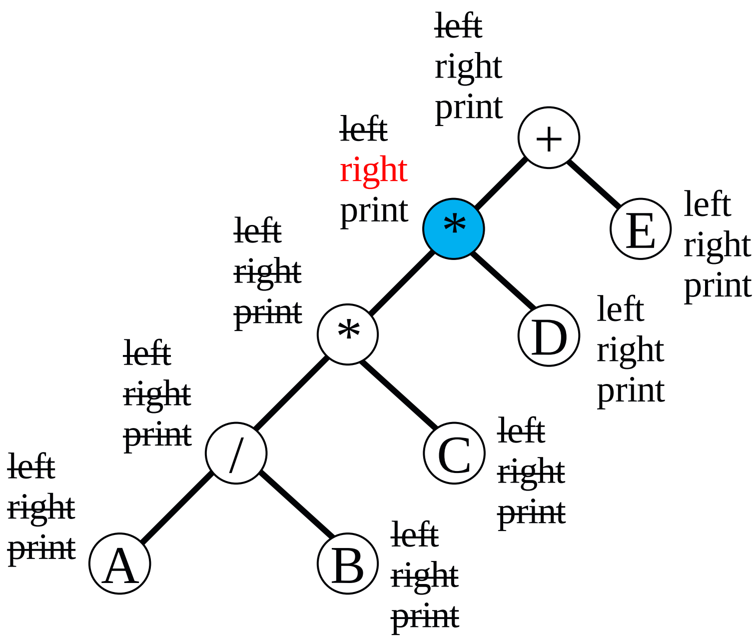
output: AB/C

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                postorder (ptr -> left_child);
                postorder (ptr -> right_child);
                printf ("%c", ptr -> data);
      }
}
```



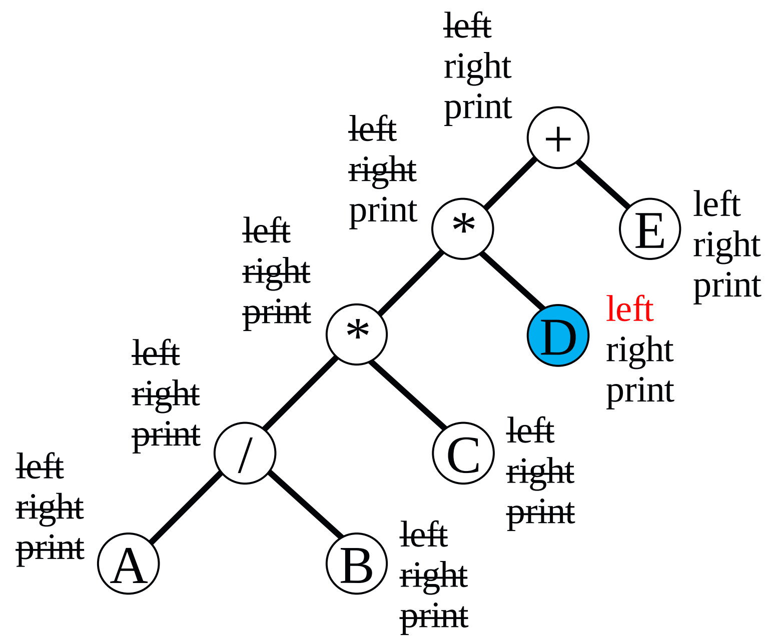
output: AB/C*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



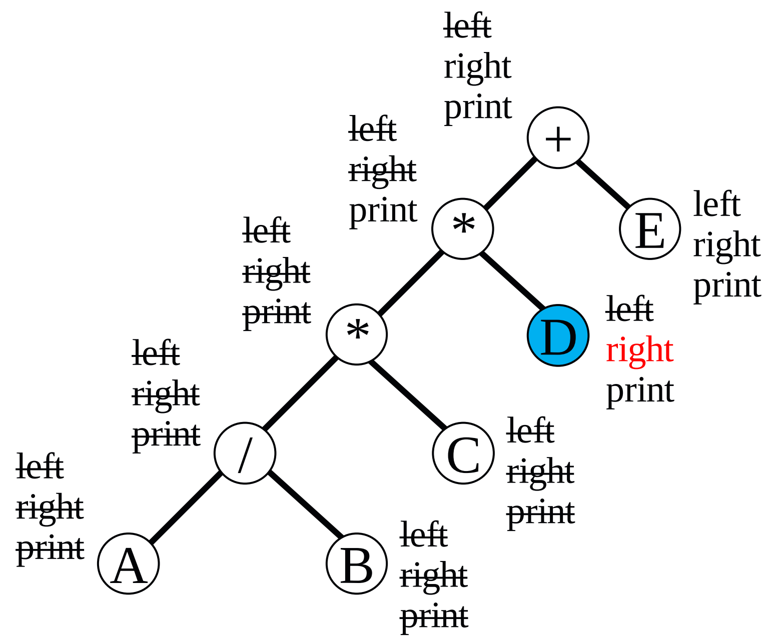
output: AB/C*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



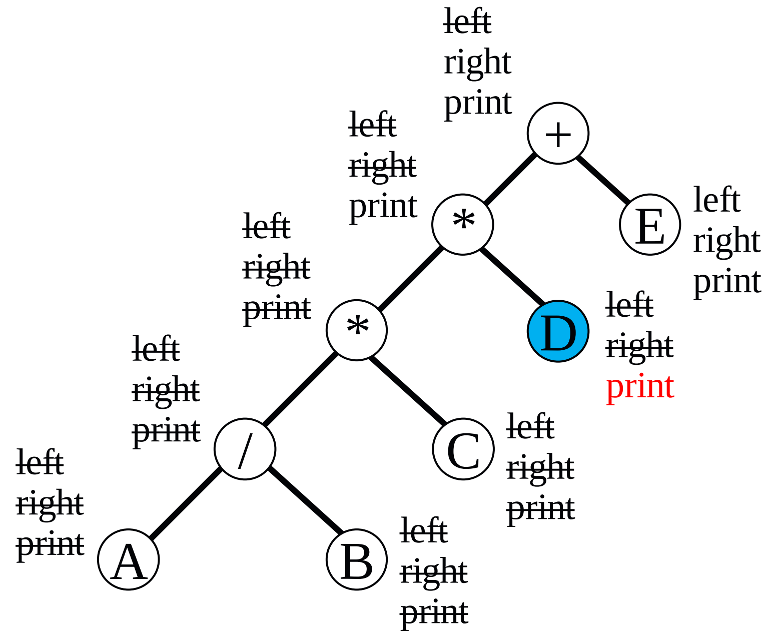
output: AB/C*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
        if  (ptr)  {
                        postorder (ptr -> left_child);
                        postorder (ptr -> right_child);
                        printf ("%c", ptr -> data);
        }
}
```



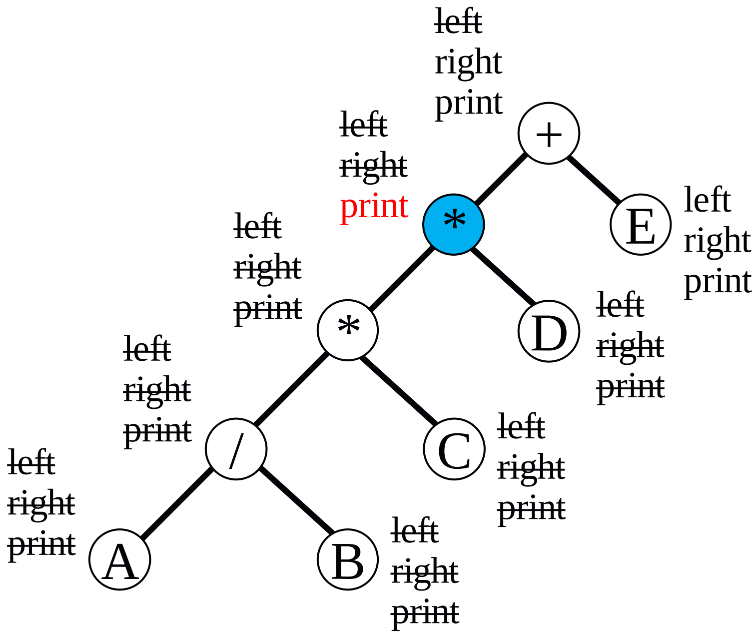
output: AB/C*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



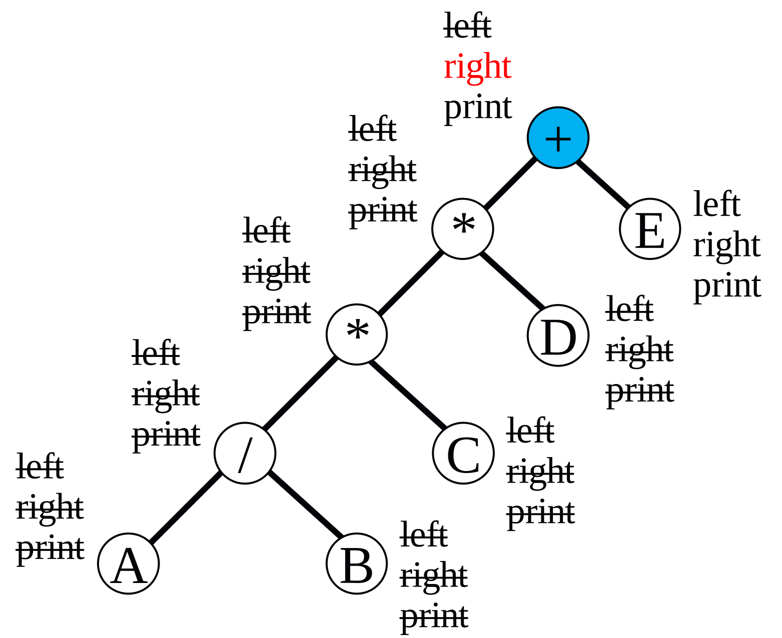
output: AB/C*D

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



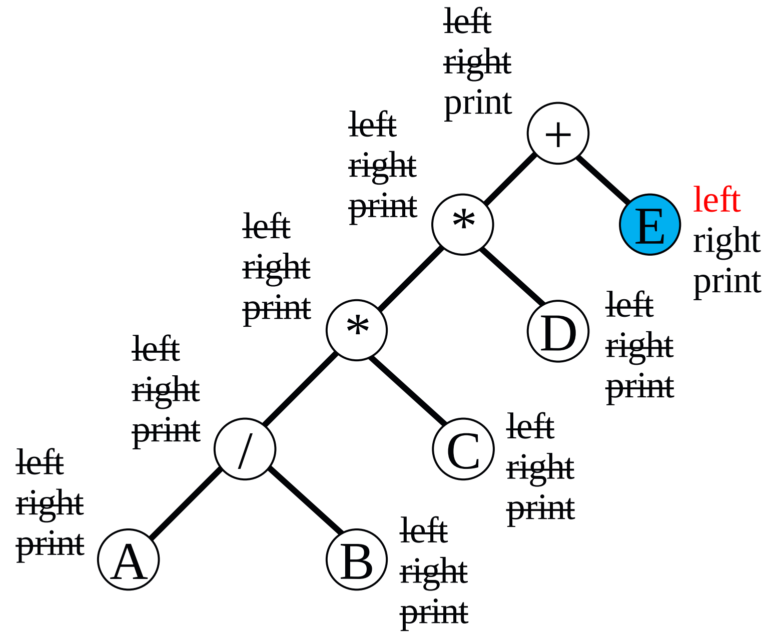
output: AB/C*D*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



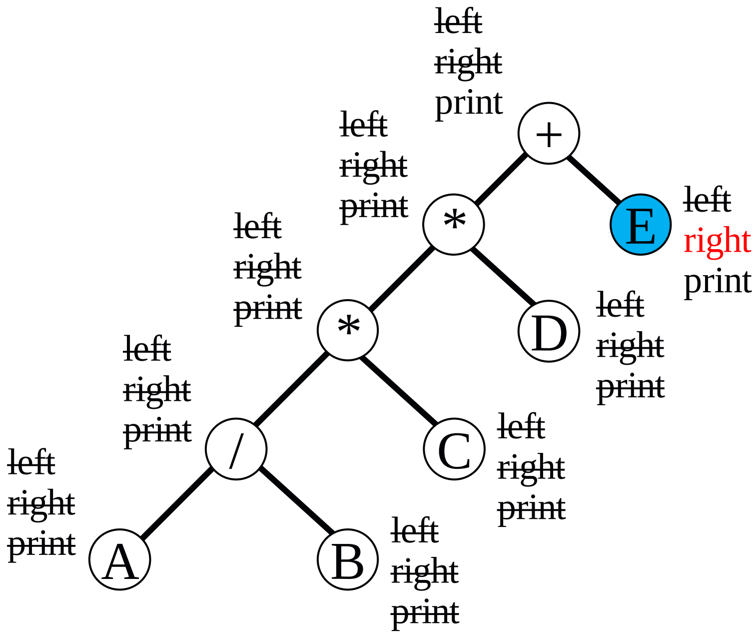
output: AB/C*D*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



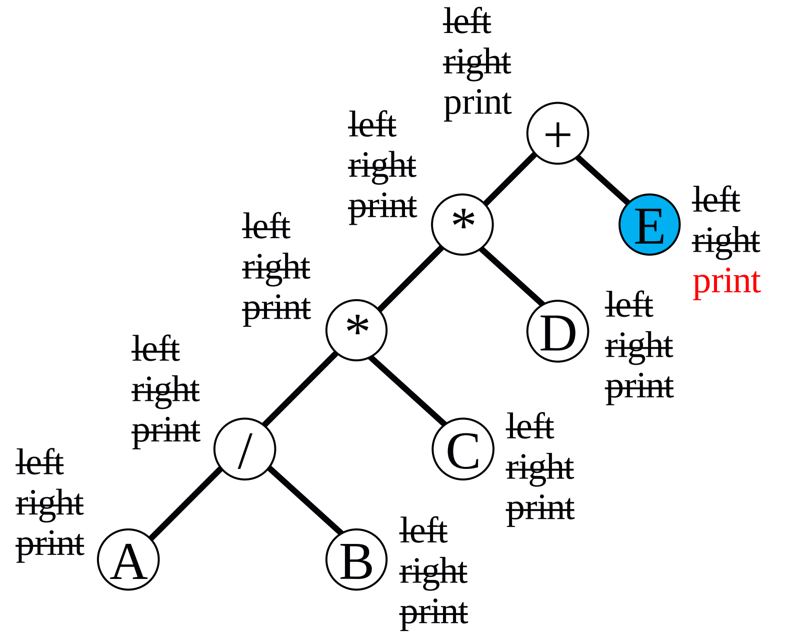
output: AB/C*D*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
            postorder (ptr -> left_child);
            postorder (ptr -> right_child);
            printf ("%c", ptr -> data);
      }
}
```



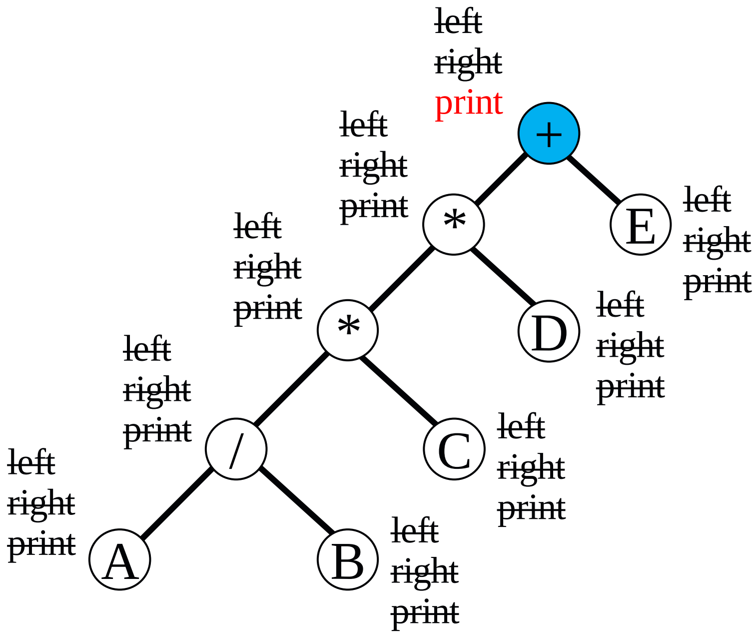
output: AB/C*D*

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



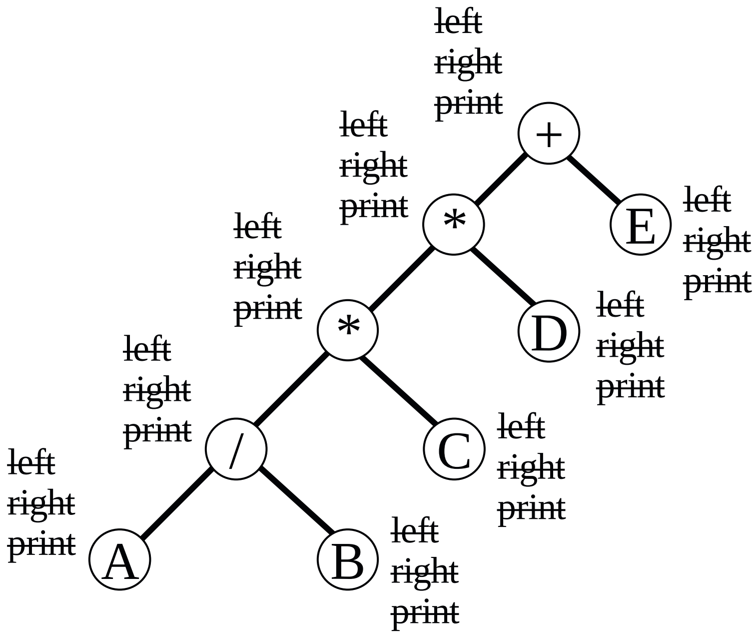
output: AB/C*D*E

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



output: AB/C*D*E+

```
void  postorder (tree_pointer  ptr)
{ /* postorder tree traversal */
      if  (ptr)  {
                  postorder (ptr -> left_child);
                  postorder (ptr -> right_child);
                  printf ("%c", ptr -> data);
      }
}
```



output: AB/C*D*E+

Call of inorder	Value in root	Action	inorder	in root	Value 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	

■ Iterative Inorder Traversal

Figure 5.17 implicitly shows the stacking and unstacking of Program 5.1.

- a node that has no action indicates that the node is added to the stack,
- while a node that has a printf action indicates that the node is removed from the stack.

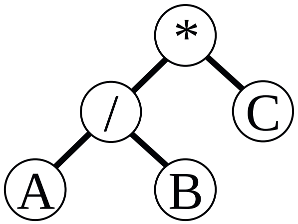
Notice that :

- the left nodes are stacked until a null node is reached,
- the node is then removed from the stack, and
- the node's right child is stacked.

Program 5.4: Iterative inorder traversal

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

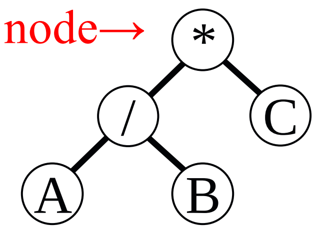
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output:

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

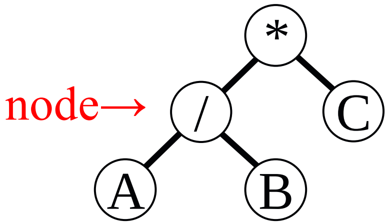


stack

	*	

output:

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

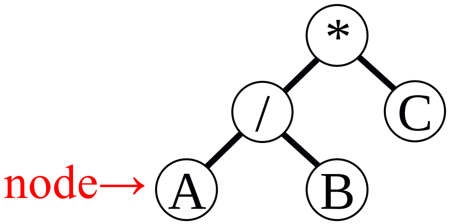


stack

	/	
	*	

output:

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

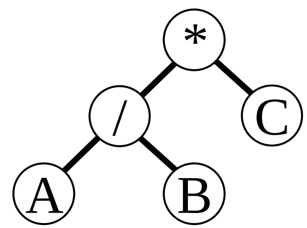


stack

	A	
	/	
	*	

output:

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



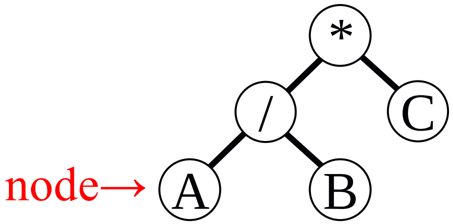
node=NULL

stack

	A	
	/	
	*	

output:

```c
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

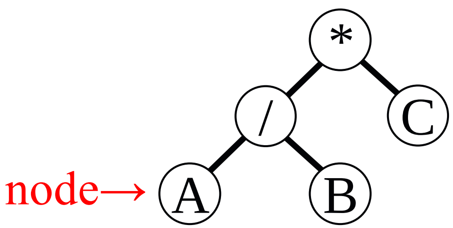


stack

	/	
	*	

output:

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

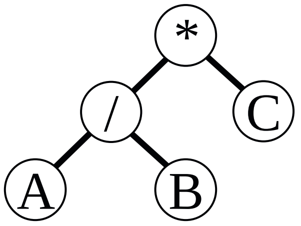


stack

	/	
	*	

output: A

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



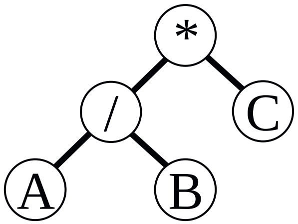
node=NULL

stack

	/	
	*	

output: A

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



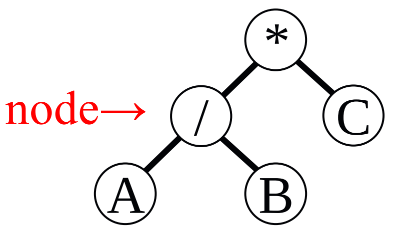
node=NULL

stack

	/	
	*	

output: A

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

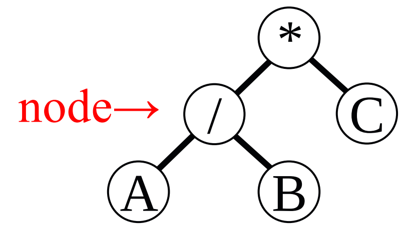


stack

	*	

output: A

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

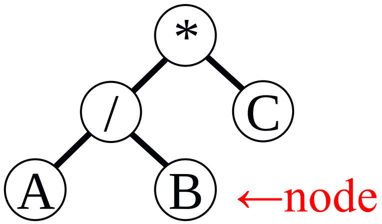


stack

	*	

output: A/

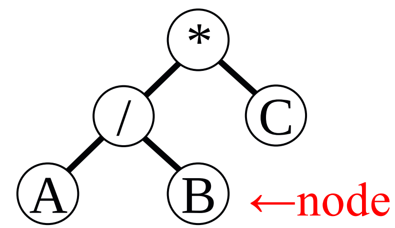
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack | | * |

output: A/

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

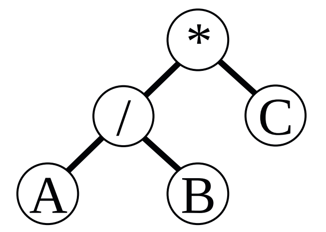


stack

	B	
	*	

output: A/

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



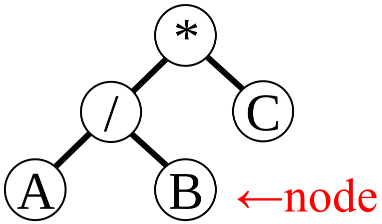
node=NULL

stack

	B	
	*	

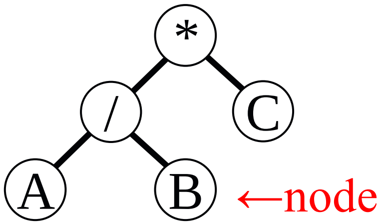
output: A/

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



output: A/

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

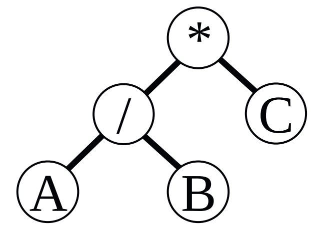


stack

	*	

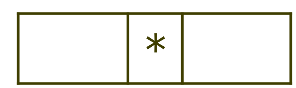
output: A/B

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                 push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



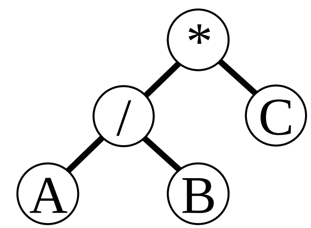
node=NULL

stack



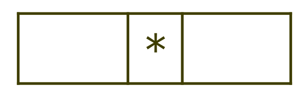
output: A/B

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



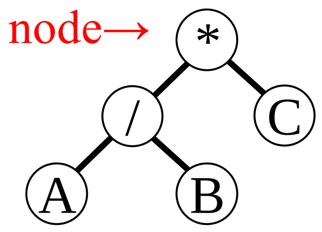
node=NULL

stack



output: A/B

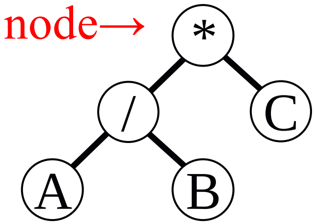
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output: A/B

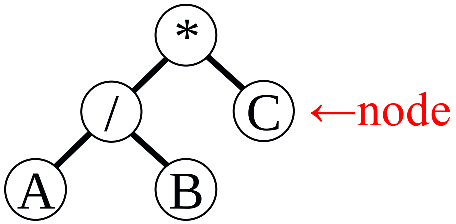
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                 push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output: A/B*

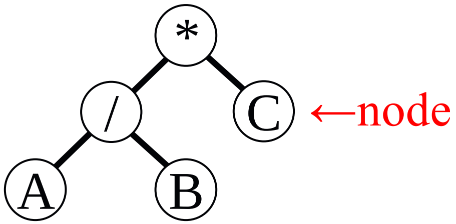
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output: A/B*

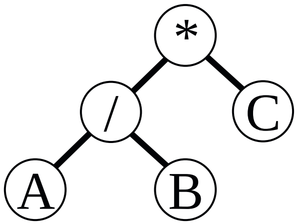
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack | | C | |

output: A/B*

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



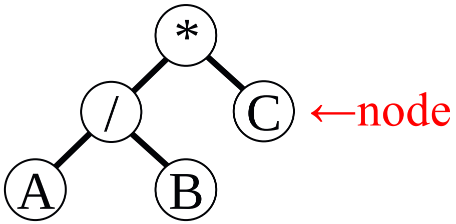
node=NULL

stack

	C	

output: A/B*

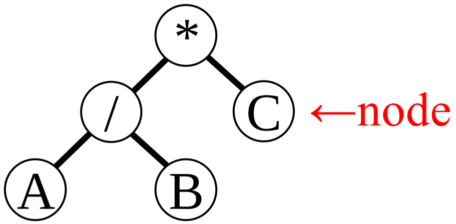
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output: A/B*

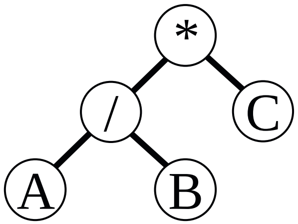
```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



stack

output: A/B*C

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

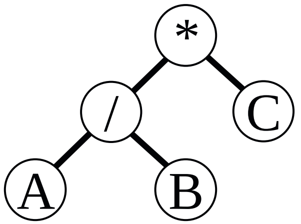


node=NULL

stack

output: A/B*C

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

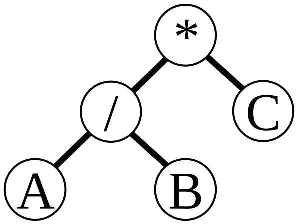


node=NULL

stack

output: A/B*C

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```

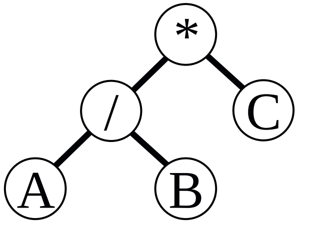


node=NULL

stack

output: A/B*C

```
void iter_inorder(tree_pointer node)
{
        int  top = -1;  /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for ( ; ; )  {
            for ( ; node; node = node -> left_child)
                push(node);   /* add to stack */
            node = pop();    /* delete from stack */
            if (!node)  break;   /* empty stack  */
            printf ("%c", node -> data);
            node = node -> right_child;
        }
}
```



node=NULL

stack

output: A/B*C

Analysis of iter_inorder

Let n be the number of nodes in the tree.

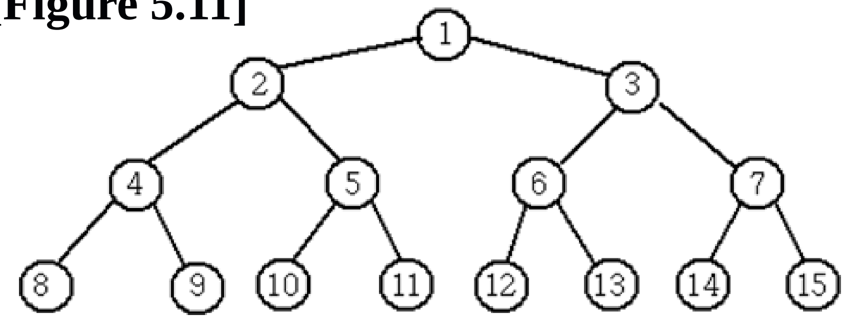
Note that every node of the tree is placed on and removed from the stack exactly once.

→ The time complexity is $O(n)$.

→ The space complexity is equal to the depth of the tree which is $O(n)$.

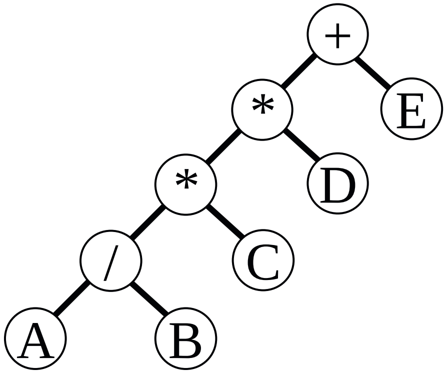
Level Order Traversal

[Figure 5.11]



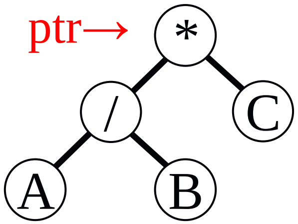
A traversal that requires a queue.

Level order traversal visits the nodes using the ordering scheme suggested in Figure 5.11.



$\longrightarrow$ $+ * E * D / C\ A\ B$

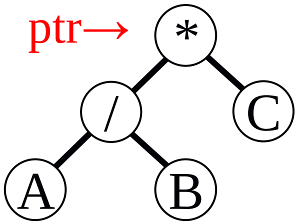
```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
      int  front = rear = 0;
      tree_pointer  queue[MAX_QUEUE_SIZE];
      if (!ptr)  return;   /* empty tree */
      addq(ptr);
      for ( ; ; )  {
            ptr = deleteq(); /*empty queue returns NULL*/
            if (ptr)  {
                  printf("%c", ptr->data);
                  if (ptr->left_child)
                        addq(ptr->left_child);
                  if (ptr->right_child)
                        addq(ptr->right_child);
            }
            else  break;
      }
}
```



queue

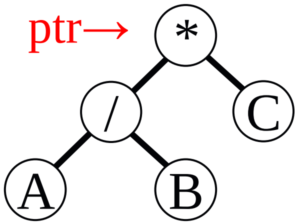
output:

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



output:

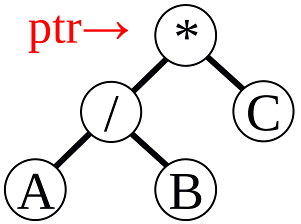
```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return;  /* empty tree */
    addq(ptr);
    for ( ; ; ) {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr) {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else break;
    }
}
```



queue

output:

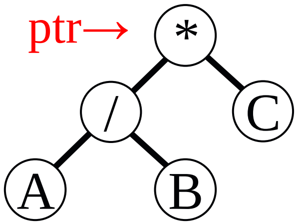
```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



queue

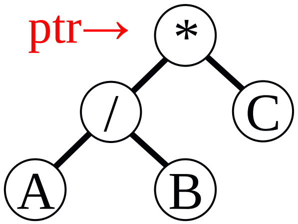
output: *

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



output: *

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

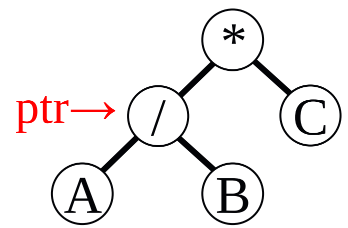


queue

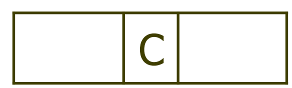
	/	
	C	

output: *

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

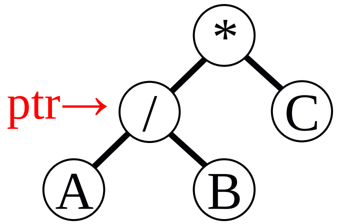


queue



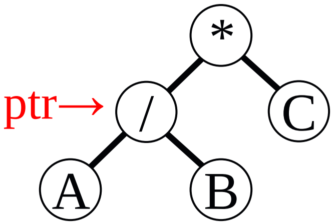
output: *

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



output: */

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

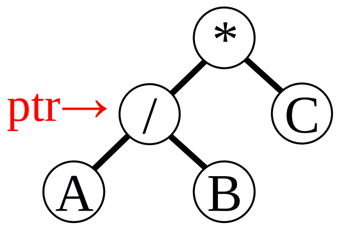


queue

	C	
	A	

output: */

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

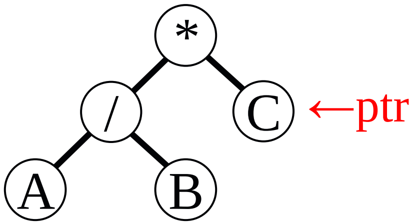


queue

	C	
	A	
	B	

output: */

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
      int  front = rear = 0;
      tree_pointer  queue[MAX_QUEUE_SIZE];
      if (!ptr)  return;   /* empty tree */
      addq(ptr);
      for ( ; ; )  {
            ptr = deleteq(); /*empty queue returns NULL*/
            if (ptr)  {
                  printf("%c", ptr->data);
                  if (ptr->left_child)
                        addq(ptr->left_child);
                  if (ptr->right_child)
                        addq(ptr->right_child);
            }
            else  break;
      }
}
```

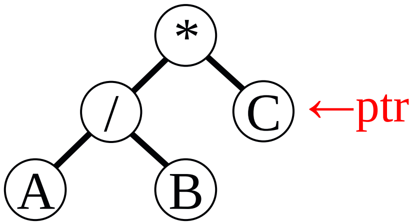


queue

	A	
	B	

output: */

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

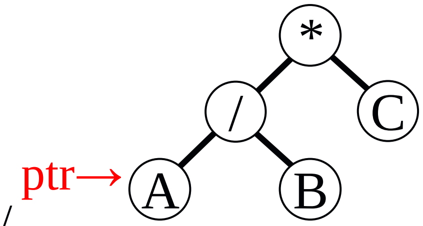


queue

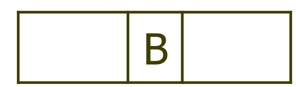
	A	
	B	

output: */C

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
      int  front = rear = 0;
      tree_pointer  queue[MAX_QUEUE_SIZE];
      if (!ptr)  return;   /* empty tree */
      addq(ptr);
      for ( ; ; )  {
            ptr = deleteq(); /*empty queue returns NULL*/
            if (ptr)  {
                  printf("%c", ptr->data);
                  if (ptr->left_child)
                        addq(ptr->left_child);
                  if (ptr->right_child)
                        addq(ptr->right_child);
            }
            else  break;
      }
}
```

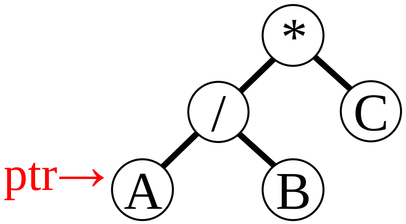


queue

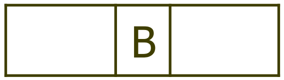


output: */C

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

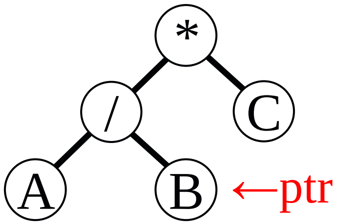


queue



output: */CA

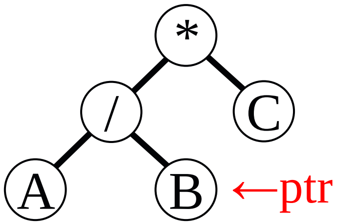
```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



queue

output: */CA

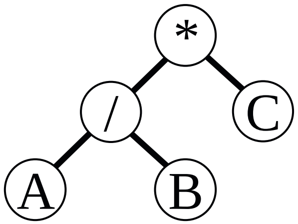
```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



queue

output: */CAB

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```

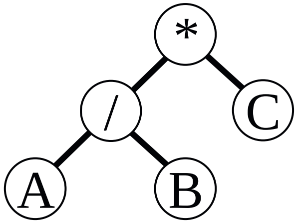


ptr = NULL

queue

output: */CAB

```
void level_order(tree_pointer ptr)
{ /* level order tree traversal */
    int  front = rear = 0;
    tree_pointer  queue[MAX_QUEUE_SIZE];
    if (!ptr)  return;   /* empty tree */
    addq(ptr);
    for ( ; ; )  {
        ptr = deleteq(); /*empty queue returns NULL*/
        if (ptr)  {
            printf("%c", ptr->data);
            if (ptr->left_child)
                addq(ptr->left_child);
            if (ptr->right_child)
                addq(ptr->right_child);
        }
        else  break;
    }
}
```



ptr = NULL

queue

output: */CAB

5.4 ADDITIONAL BINARY TREE OPERATIONS

- By using the definition of a binary tree and

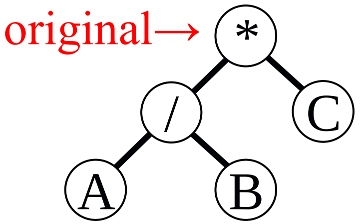
the recursive versions of inorder, preorder, and postorder traversals, we can easily create C functions for other binary tree operations.

- Copying Binary Trees

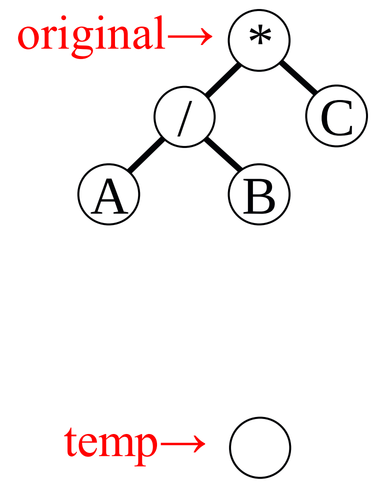
One practical operation is copying a binary tree. (Program 5.6)
Note that this function is only a slightly modified version of postorder (Program 5.3)

[Program 5.6] Copying a bainary tree

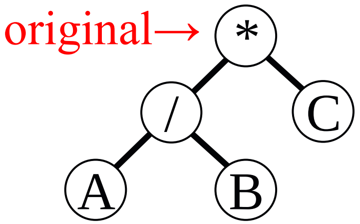
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```

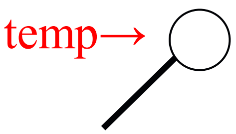


```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```

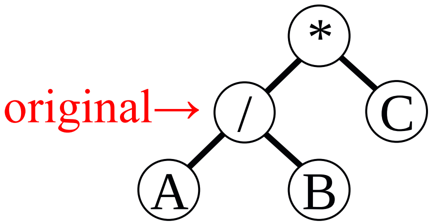


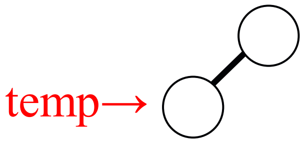
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
      tree_pointer  temp;
      if (original)  {
            temp = (tree_pointer) malloc(sizeof(node));
            if (IS_FULL(temp)) {
                  fprintf(stderr, "The memory is full\n");
                  exit(1);
            }
            temp->left_child  = copy(original->left_child);
            temp->right_child = copy(original->right_child);
            temp->data = original->data;
            return  temp;
      }
      return  NULL;
}
```



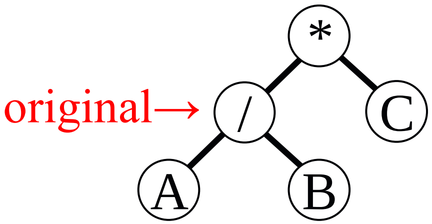


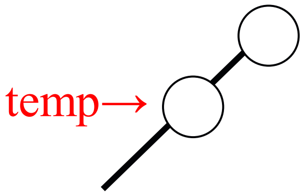
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



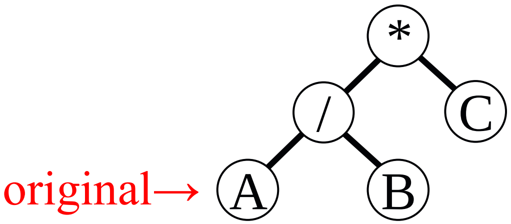


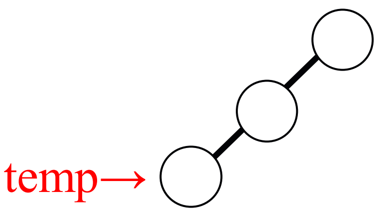
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
       of the original tree */
      tree_pointer  temp;
      if (original)  {
            temp = (tree_pointer) malloc(sizeof(node));
            if (IS_FULL(temp)) {
                  fprintf(stderr, "The memory is full\n");
                  exit(1);
            }
            temp->left_child  = copy(original->left_child);
            temp->right_child = copy(original->right_child);
            temp->data = original->data;
            return  temp;
      }
      return  NULL;
}
```



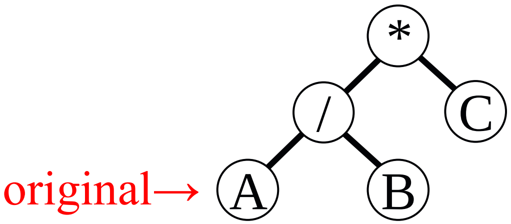


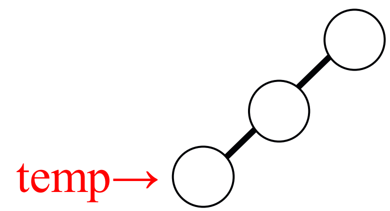
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```



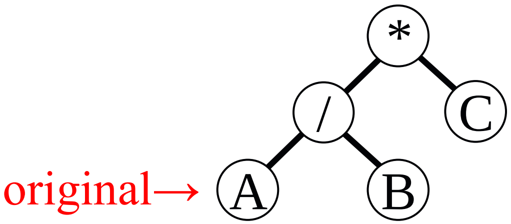


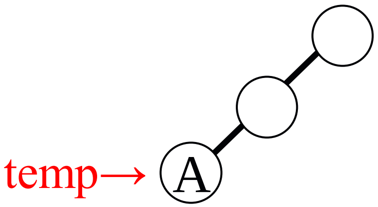
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child  = copy(original->left_child);
        temp->right_child  = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```



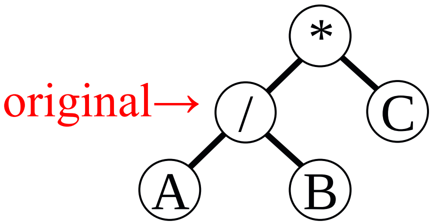


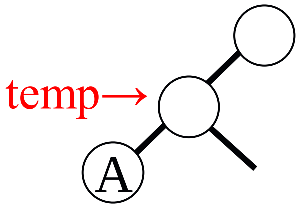
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```



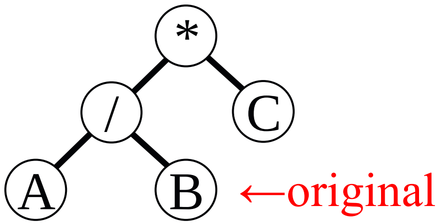


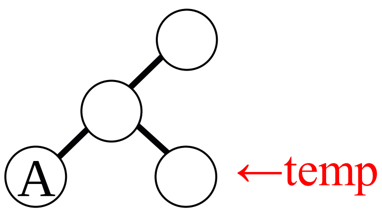
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



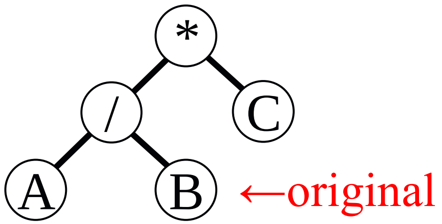


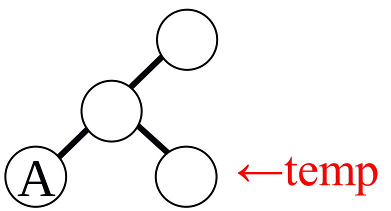
```c
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



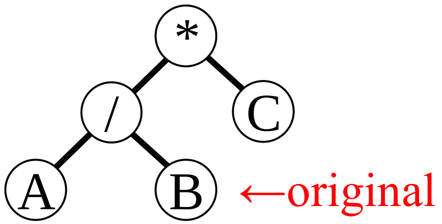


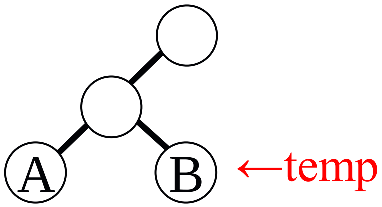
```c
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
       of the original tree */
      tree_pointer  temp;
      if (original)  {
            temp = (tree_pointer) malloc(sizeof(node));
            if (IS_FULL(temp)) {
                  fprintf(stderr, "The memory is full\n");
                  exit(1);
            }
            temp->left_child = copy(original->left_child);
            temp->right_child = copy(original->right_child);
            temp->data = original->data;
            return  temp;
      }
      return  NULL;
}
```



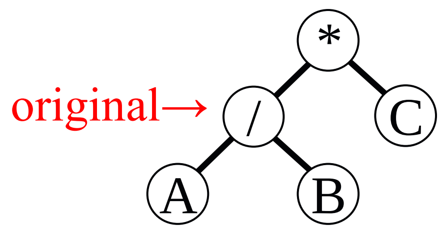


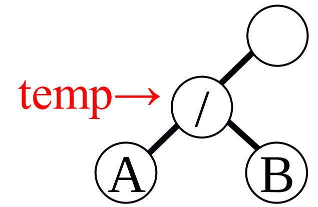
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



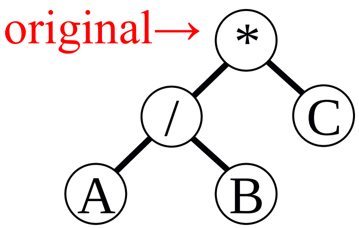


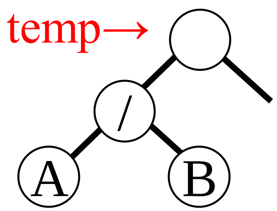
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```



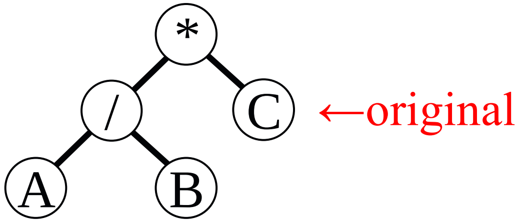


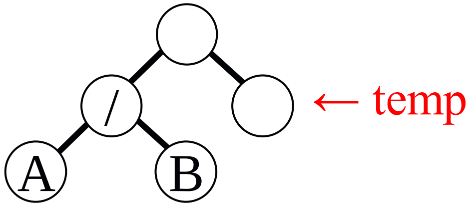
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



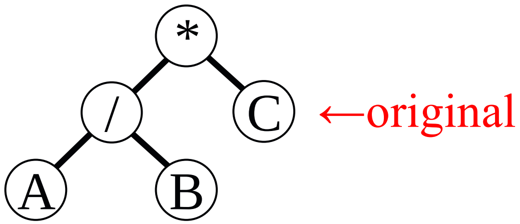


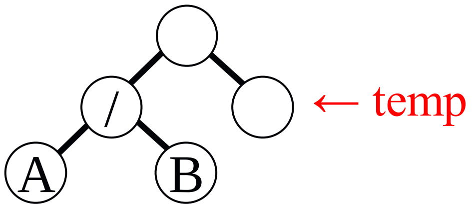
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child = copy(original->left_child);
          temp->right_child = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



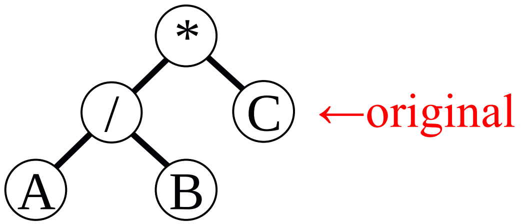


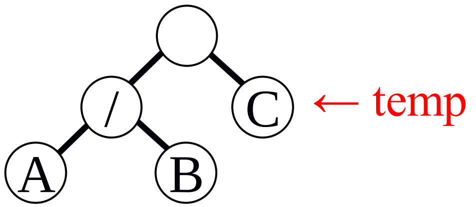
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
     tree_pointer  temp;
     if (original)  {
          temp = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(temp)) {
               fprintf(stderr, "The memory is full\n");
               exit(1);
          }
          temp->left_child  = copy(original->left_child);
          temp->right_child  = copy(original->right_child);
          temp->data = original->data;
          return  temp;
     }
     return  NULL;
}
```



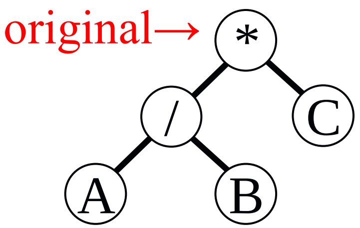


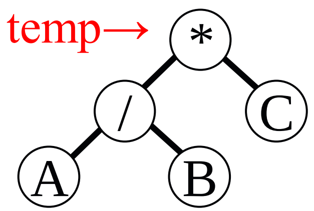
```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
      of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```





```
tree_pointer  copy(tree_pointer original)
{ /* this function returns a tree_pointer to an exact copy
     of the original tree */
    tree_pointer  temp;
    if (original)  {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return  temp;
    }
    return  NULL;
}
```





Testing For Equality Of Binary Trees

Equivalent binary trees have the same structure and the same information in the corresponding nodes.

```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
            second are not equal, otherwise it returns TRUE */
    return ((!first && !second) || (first && second &&
              (first->data == second->data) &&
              equal(first->left_child, second->left_child) &&
              equal(first->right_child, second->right_child))
}
```

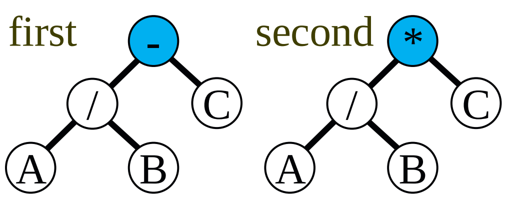
```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
      second are not equal, otherwise it returns TRUE */
     return ((!first && !second) || (first && second &&
                 (first->data == second->data) &&
                 equal(first->left_child, second->left_child) &&
                 equal(first->right_child, second->right_child))
}
```

TRUE	FALSE
first == NULL and second == NULL	first != NULL or second != NULL

```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
       second are not equal, otherwise it returns TRUE */
      return ((!first && !second) || (first && second &&
                  (first->data == second->data) &&
                  equal(first->left_child, second->left_child) &&
                  equal(first->right_child, second->right_child))
}
```

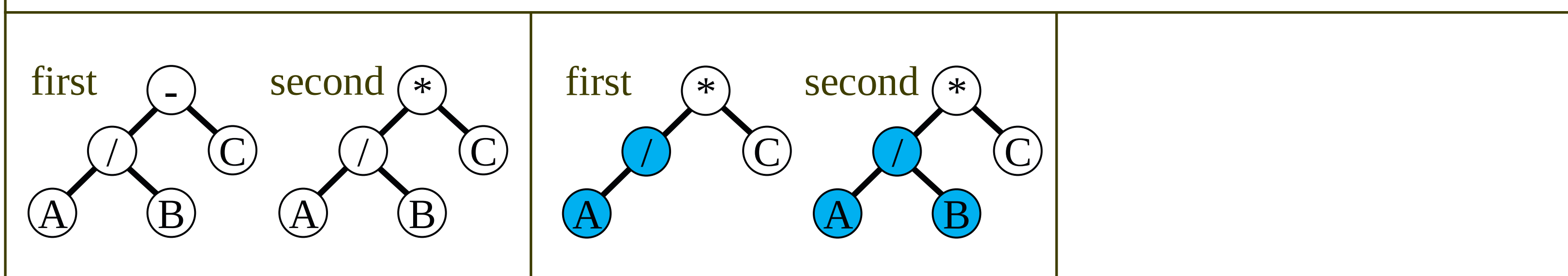
FALSE		

```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
      second are not equal, otherwise it returns TRUE */
     return ((!first && !second) || (first && second &&
              (first->data == second->data) &&
              equal(first->left_child, second->left_child) &&
              equal(first->right_child, second->right_child))
}
```

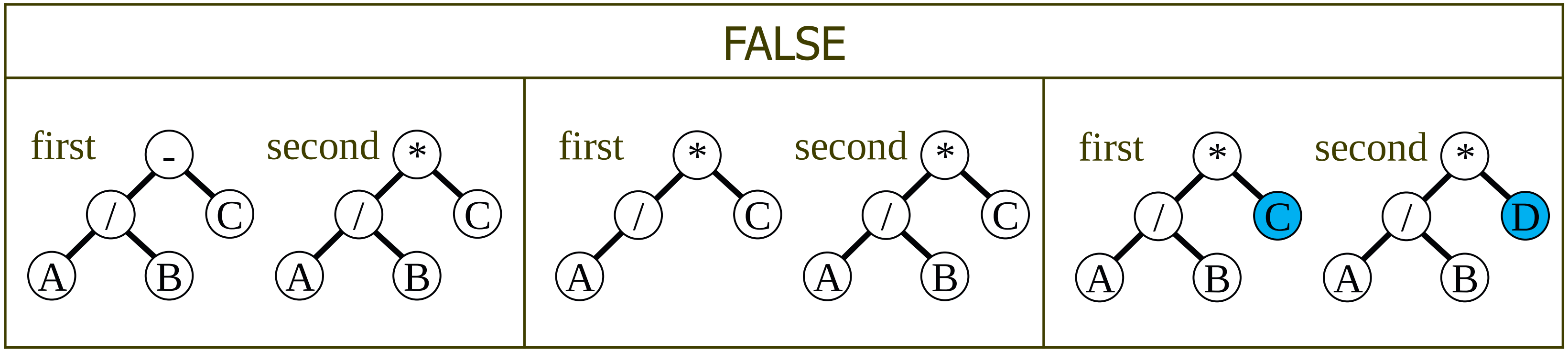
FALSE		
		

```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
       second are not equal, otherwise it returns TRUE */
      return ((!first && !second) || (first && second &&
                  (first->data == second->data) &&
                  equal(first->left_child, second->left_child) &&
                  equal(first->right_child, second->right_child))
}
```

FALSE



```
int equal(tree_pointer first,  tree_pointer second)
{ /* function returns FALSE if the binary trees first and
      second are not equal, otherwise it returns TRUE */
     return ((!first && !second) || (first && second &&
             (first->data == second->data) &&
             equal(first->left_child, second->left_child) &&
             equal(first->right_child, second->right_child))
}
```

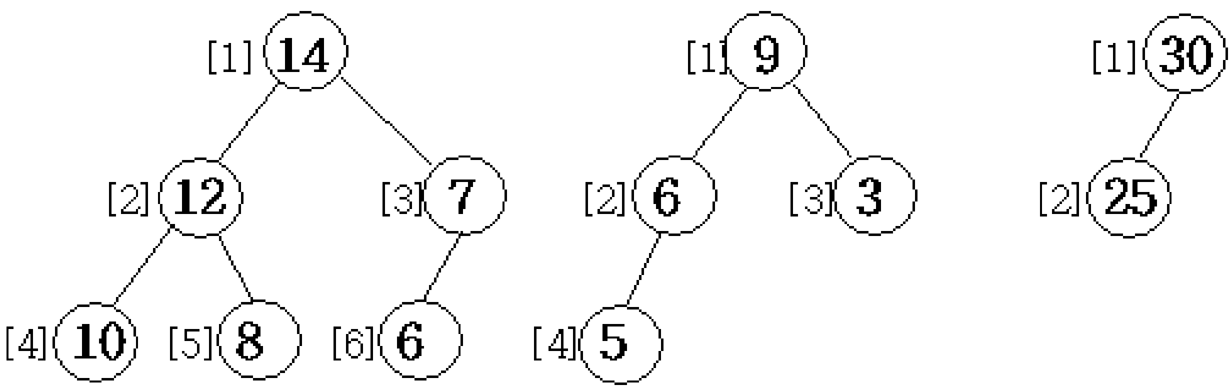


5.6 HEAPS

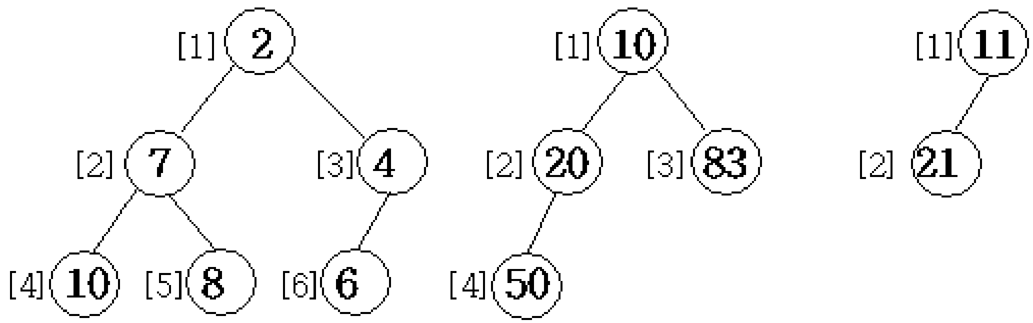
Definition : A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children (if any).
A *max heap* is a complete binary tree that is also a max tree.

Definition : A *min tree* is a tree in which the key value in each node is no larger than the key values in its children (if any).
A *min heap* is a complete binary tree that is also a min tree.

[Figure 5.25] Max heaps



[Figure 5.26] Min heaps



Notice that we represent a heap as an array, although we do not use position 0.

From the heap definitions it follows that

- the root of a min tree contains the smallest key in the tree.
- the root of a max tree contains the largest key in the tree.

Basic operations on a max heap :

(1) Creation of an empty heap

(2) Insertion of a new element into the heap

(3) Deletion of the largest element from the heap

Abstract data type MaxHeap.

ADT MaxHeap is

objects: a complete binary tree of n>=0 elements organized so that
the value in each node is at least as large as those in its children

functions:

for all heap ∈ MaxHeap, item ∈ Element, n, max_size ∈ integer

MaxHeap Create(max_size) ::= create an empty heap that can hold a maximum of
max_size elements.

Boolean HeapFull(heap, n) ::= if (n == max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n) ::= if (!HeapFull(heap, n)) insert an item into heap and
return the resulting heap
else return error.

Boolean HeapEmpty(heap, n) ::= if (n<=0) return TRUE
else return FALSE

MaxHeap Delete(heap, n) ::= if (!HeapEmpty(heap, n)) return one of the largest element
in the heap and remove it from the heap
else return error.

Priority Queues

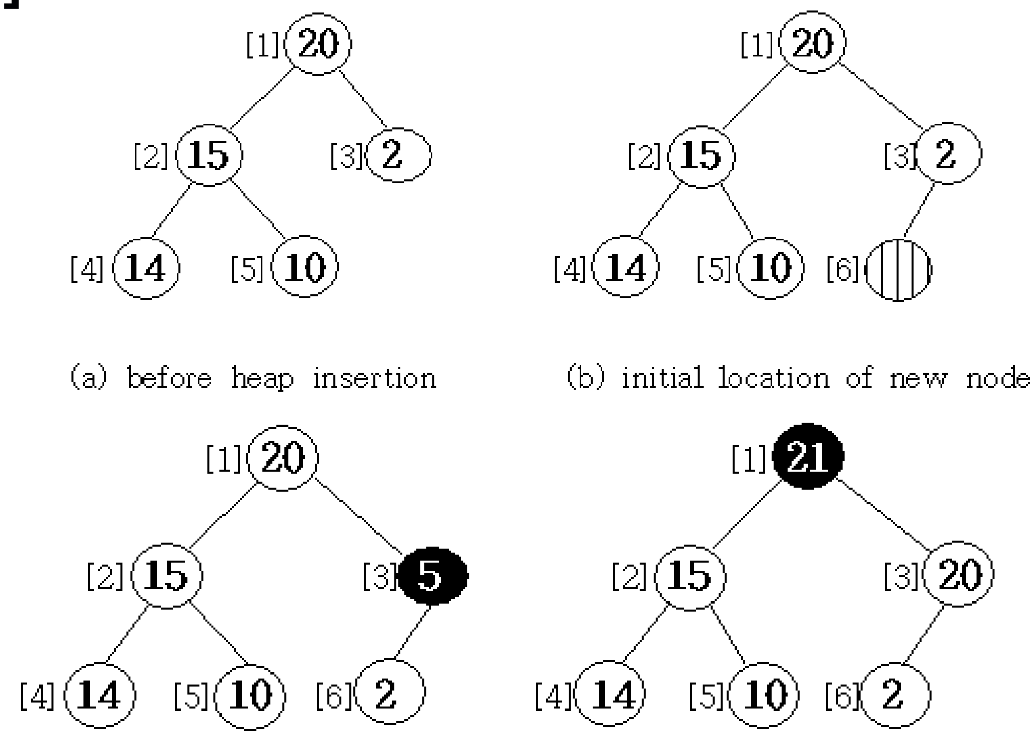
Heaps are frequently used to implement *priority queues*.

Unlike the queues, FIFO lists, a priority queue deletes the element with the highest (or the lowest) priority.

At any time, an element with arbitrary priority can be inserted into a priority queue.

5.6.3 Insertion Into A Max Heap

[Figure 5.27]



(a) before heap insertion

(b) initial location of new node

(c) insertion 5 into heap(a)

(d) insertion 21 into heap(a)

We assume that the heap is created using the following C declaration:

```
#define MAX_ELEMENTS 200    /*maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

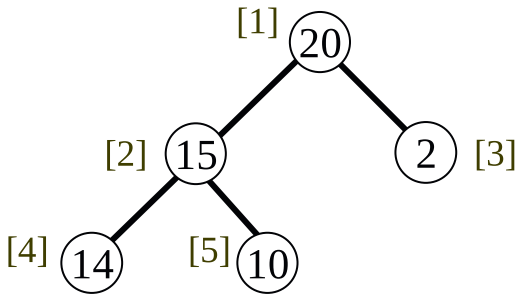
We can insert a new element in a heap with n elements by following the steps below :

(1) place the element in the new node (i.e., n+1 th position)
(2) move along the path from the new node to the root,
    if the element of the current node is larger than its parent
    then interchange them and repeat.

[Program 5.13] Insertion into a max heap

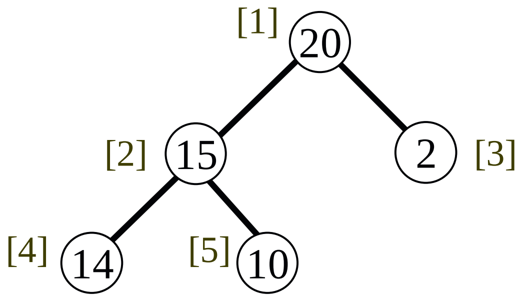
```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



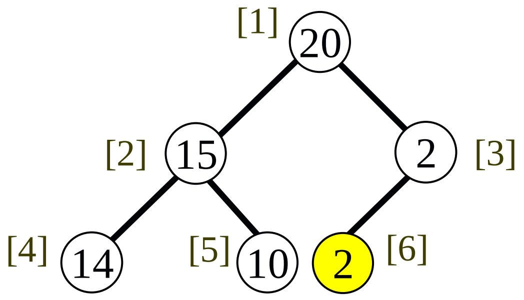
item.key	21
*n	5
i	
i/2	

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



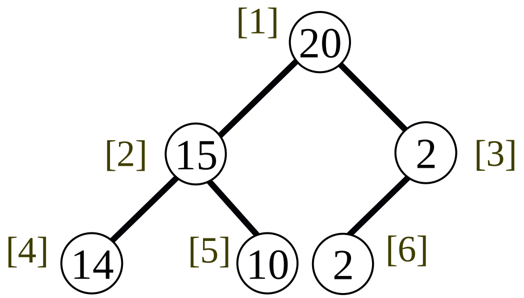
item.key	21
*n	5
i	6
i/2	3

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



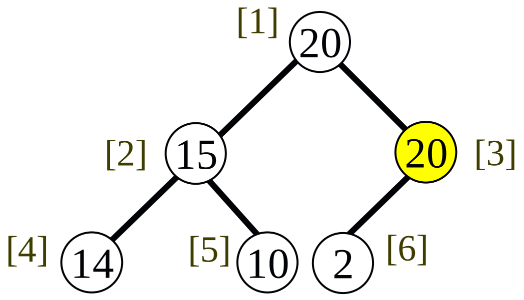
item.key	21
*n	5
i	6
i/2	3

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



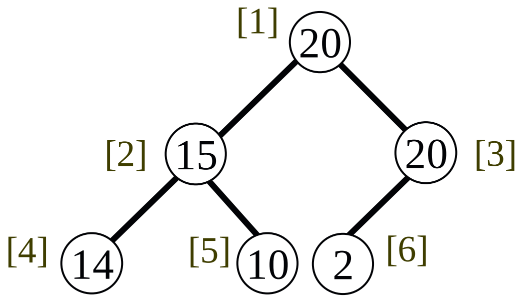
item.key	21
*n	5
i	3
i/2	1

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



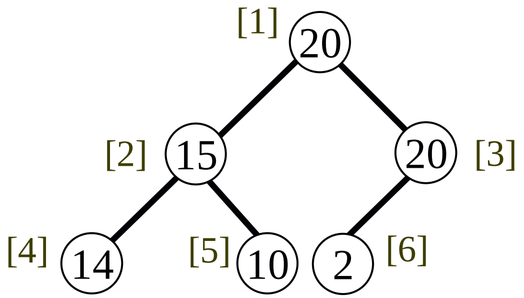
item.key	21
*n	5
i	3
i/2	1

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
      int i;
      if (HEAP_FULL(*n)) {
            fprintf(stderr, "The heap is full. \n");
            exit(EXIT_FAILURE);
      }
      i = ++(*n);
      while ((i != 1) && (item.key > heap[i/2].key)) {
            heap[i] = heap[i/2];
            i /= 2;
      }
      heap[i] = item;
}
```



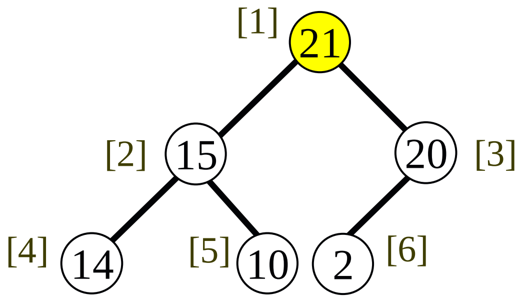
item.key	21
*n	5
i	1
i/2	0

```c
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



item.key	21
*n	5
i	1
i/2	0

```
void push(element item, int *n)
{ /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```



item.key	21
*n	5
i	1
i/2	0

Analysis of *push* :

The function first checks for a full heap.

If not, sets *i* to the size of the new heap (n+1).

Then determines the correct position of item in the heap by using the while loop.

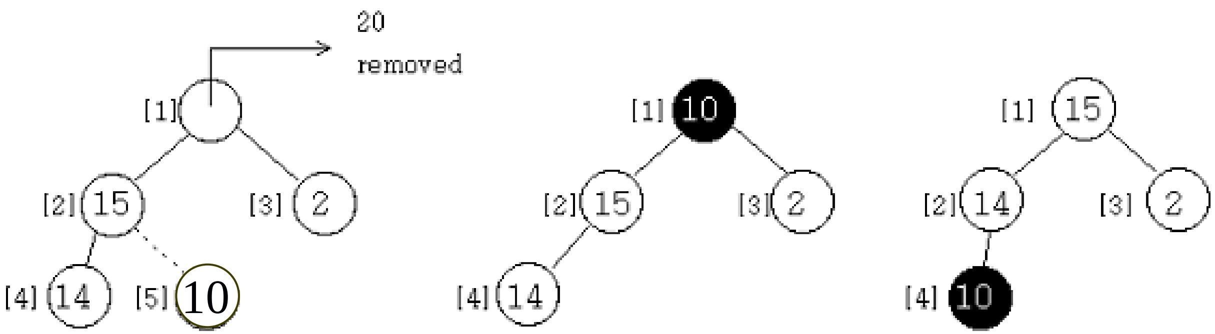
This while loop is iterated O($\log_2 n$) times.

Hence the time complexity is O($\log_2 n$).

5.6.4 Deletion From A Max Heap

When we delete an element from a max heap, we always take it from the root of the heap.
If the heap had n elements, after deleting the element in the root,
the heap must become a complete binary tree with one less nodes, i.e., (n-1) elements.
We place the element in the node at position n in the root node and
move down the heap, comparing the parent node with its children and exchanging
out-of-order elements until the heap is reestablished.

[Figure 5.28] Deletion from a max heap



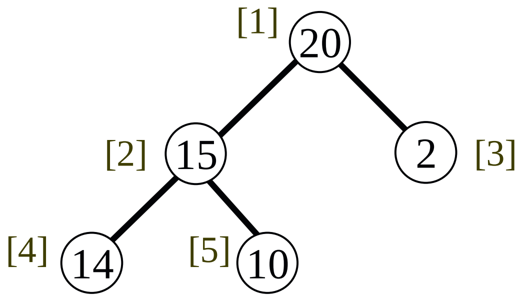
(c) heap structure (b) 10 inserted at the root (c) final heap

[Program 5.14] Deletion from a max heap

```
element pop(int *n)
{ /* delete element with the highest key from the heap */
    int  parent, child;
    element  item, temp;
    if (HEAP_EMPTY(*n))  {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    /* save value of the element with the highest key */
    item = heap[1];
    /* use last element in heap to adjust heap  */
    temp = heap[(*n)--];
    parent = 1;   child = 2;
```

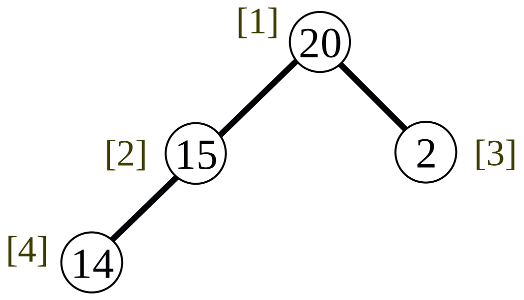
```
    while (child <= *n)  {
        /* find the larger child of the current parent  */
        if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key)  break;
        /* move to the next lower level  */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return  item;
}
```

```
element pop(int *n)
{ /* delete element with the highest key from the heap */
    int  parent, child;
    element  item, temp;
    if (HEAP_EMPTY(*n))  {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    /* save value of the element with the highest key */
    item = heap[1];
    /* use last element in heap to adjust heap  */
    temp = heap[(*n)--];
    parent = 1;   child = 2;
```



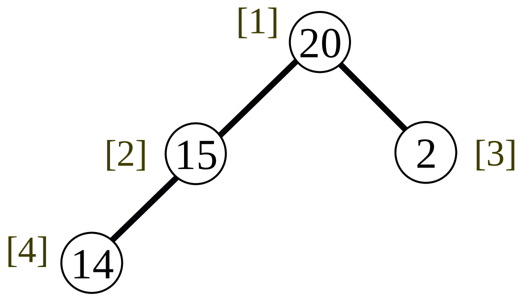
item.key	20
*n	5
temp.key	
parent	
child	

```
element pop(int *n)
{ /* delete element with the highest key from the heap */
    int  parent, child;
    element  item, temp;
    if (HEAP_EMPTY(*n))  {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    /* save value of the element with the highest key */
    item = heap[1];
    /* use last element in heap to adjust heap  */
    temp = heap[(*n)--];
    parent = 1;   child = 2;
```



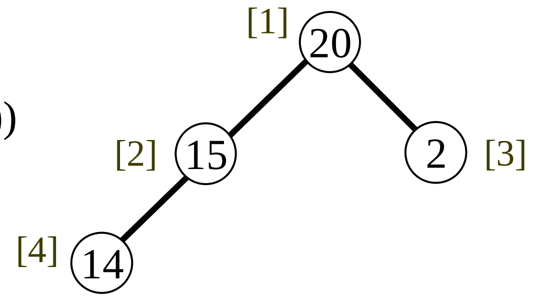
item.key	20
*n	4
temp.key	10
parent	
child	

```
element pop(int *n)
{ /* delete element with the highest key from the heap */
    int  parent, child;
    element  item, temp;
    if (HEAP_EMPTY(*n))  {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    /* save value of the element with the highest key */
    item = heap[1];
    /* use last element in heap to adjust heap  */
    temp = heap[(*n)--];
    parent = 1;   child = 2;
```



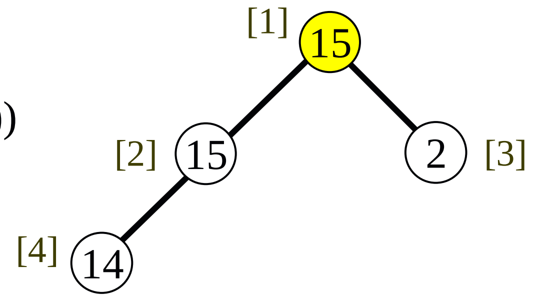
item.key	20
*n	4
temp.key	10
parent	1
child	2

```
    while (child <= *n)  {
        /* find the larger child of the current parent  */
        if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key)  break;
        /* move to the next lower level  */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return  item;
}
```



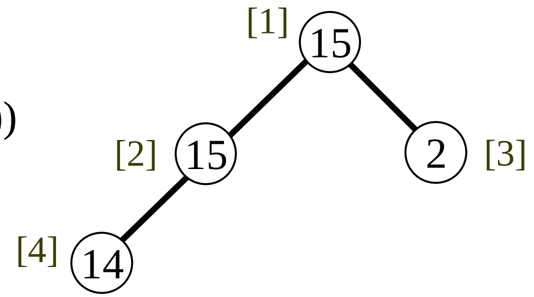
item.key	20
*n	4
temp.key	10
parent	1
child	2

```
    while (child <= *n)  {
        /* find the larger child of the current parent  */
        if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key)  break;
        /* move to the next lower level  */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return  item;
}
```



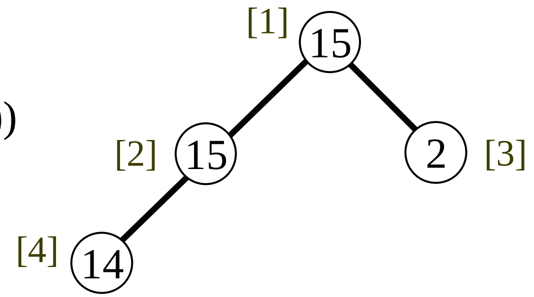
item.key	20
*n	4
temp.key	10
parent	1
child	2

```
    while (child <= *n)  {
        /* find the larger child of the current parent  */
        if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key)  break;
        /* move to the next lower level  */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return  item;
}
```



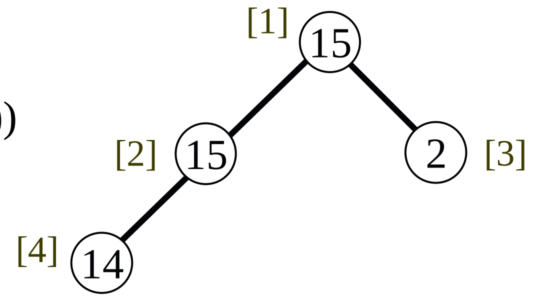
item.key	20
*n	4
temp.key	10
parent	2
child	2

```
while (child <= *n)  {
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



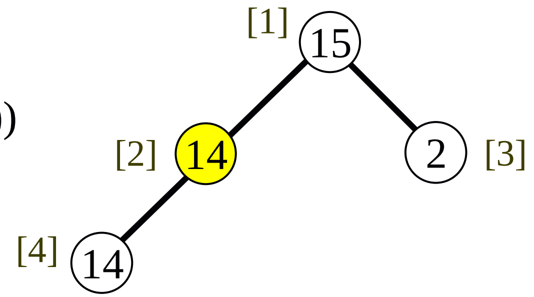
item.key	20
*n	4
temp.key	10
parent	2
child	4

```
while (child <= *n)  {
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



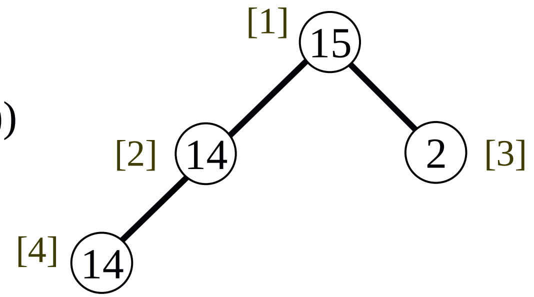
item.key	20
*n	4
temp.key	10
parent	2
child	4

```
    while (child <= *n)  {
        /* find the larger child of the current parent  */
        if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key)  break;
        /* move to the next lower level  */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return  item;
}
```



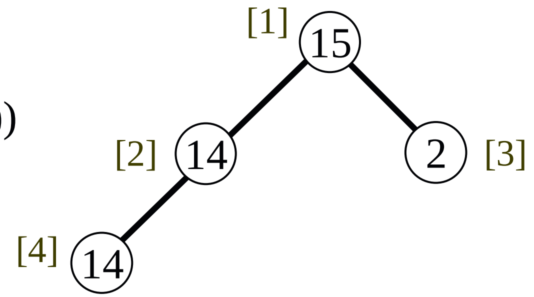
item.key	20
*n	4
temp.key	10
parent	2
child	4

```
while (child <= *n)  {
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



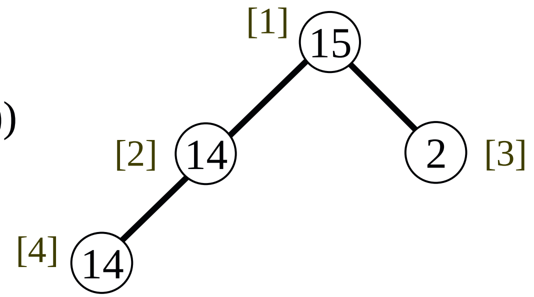
item.key	20
*n	4
temp.key	10
parent	4
child	4

```
while (child <= *n)  {
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



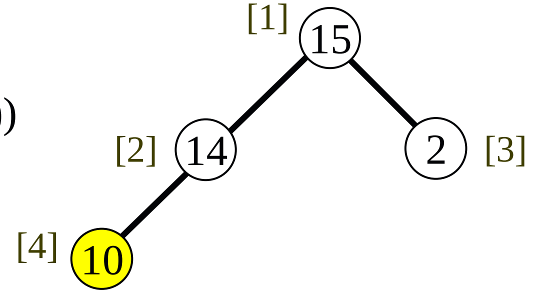
item.key	20
*n	4
temp.key	10
parent	4
child	8

```
while (child <= *n)  {   →break
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



item.key	20
*n	4
temp.key	10
parent	4
child	8

```
while (child <= *n)  {
      /* find the larger child of the current parent  */
      if ((child < *n) &&(heap[child].key< heap[child+1].key))
            child++;
      if (temp.key >= heap[child].key)  break;
      /* move to the next lower level  */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
}
heap[parent] = temp;
return  item;
}
```



item.key	20
*n	4
temp.key	10
parent	4
child	8

Analysis of *pop* :

The function *pop* operates by moving down the heap, comparing and exchanging parent and child nodes until the heap definition is re-established.

Since the height of a heap with n elements is $\lceil \log_2(n+1) \rceil$, the while loop is iterated O($\log_2 n$) times.

Hence the time complexity is O($\log_2 n$).

5.7 BINARY SEARCH TREES

5.7.1 Introduction

While a heap is well suited for applications that require priority queues, it is not well suited for applications in which we delete and search arbitrary elements.

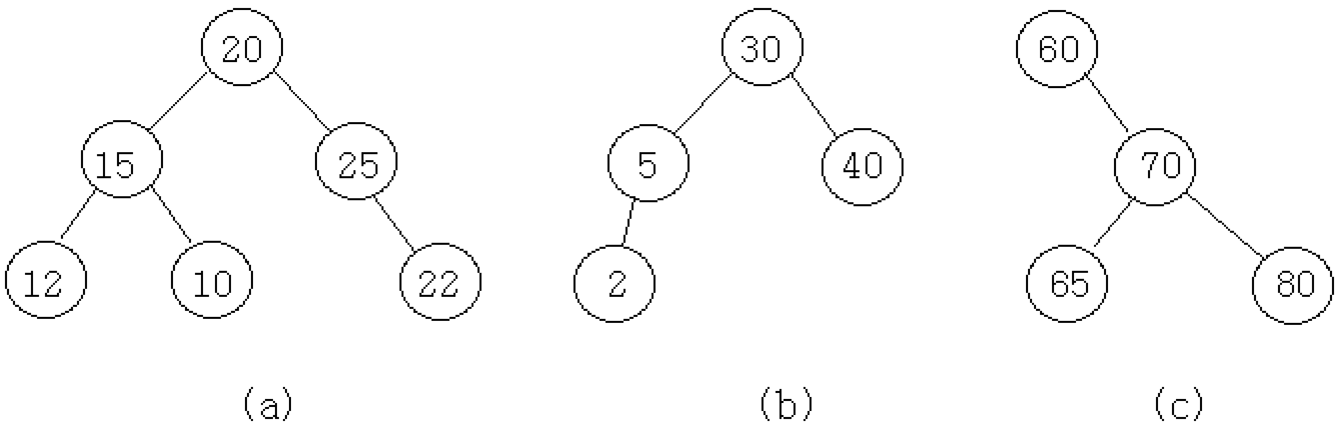
A *binary search tree* has a better performance than any of the data structures studied so far when the functions to be performed are search, insert, and delete.

In fact, with a binary search tree, these functions can be performed both by key value (e.g., delete the element with key k)
and by rank (e.g., delete the fifth smallest element).

Definition: A *binary search tree* is a binary tree. It may be empty.
If it is not empty, it satisfies the following properties :

(1) Each node has exactly one key and the keys in the tree are distinct.
(2) The keys (if any) in the left subtree are smaller than
the key in the root.
(3) The keys (if any) in the right subtree are larger than
the key in the root.
(4) The left and right subtrees are also binary search trees. □

[Figure 5.29] Binary trees



not binary search tree   binary search tree   binary search tree

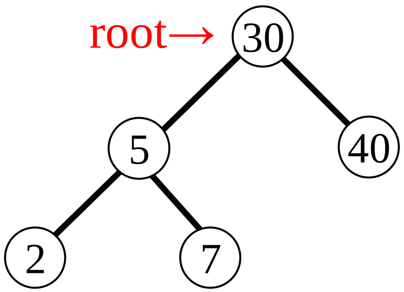
If we traverse a binary search tree in inorder and print the data of the nodes in the order visited, what would be the order of data printed?

5.7.2 Searching A Binary Search Tree

[Program 5.15] Recursive search of a binary search tree

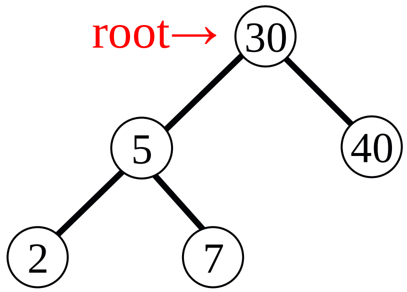
```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    if (!root)  return NULL;
    if (key == root->data)  return root;
    if (key < root->data)
        return search(root->left_child, key);
    return search(root->right_child, key);
}
```

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
     If there is no such node, return NULL. */
     if (!root)  return NULL;
     if (key == root->data)  return root;
     if (key < root->data)
          return search(root->left_child, key);
     return search(root->right_child, key);
}
```



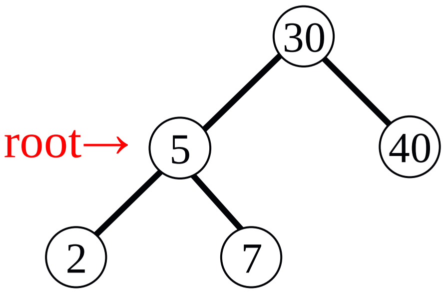
key = 7

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
     If there is no such node, return NULL. */
     if (!root)  return NULL;
     if (key == root->data)  return root;
     if (key < root->data)
          return search(root->left_child, key);
     return search(root->right_child, key);
}
```



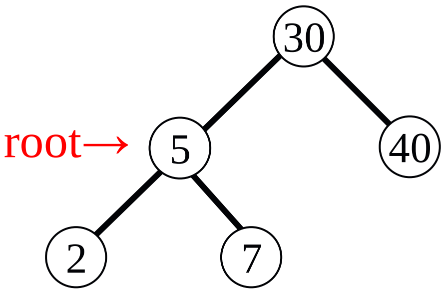
key = 7

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    if (!root)  return NULL;
    if (key == root->data)  return root;
    if (key < root->data)
        return search(root->left_child, key);
    return search(root->right_child, key);
}
```



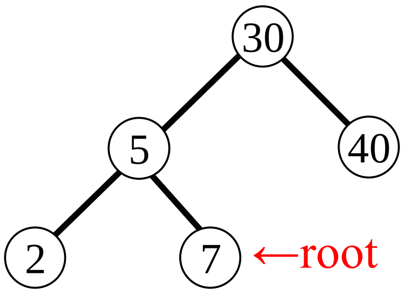
key = 7

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    if (!root)  return NULL;
    if (key == root->data)  return root;
    if (key < root->data)
        return search(root->left_child, key);
    return search(root->right_child, key);
}
```



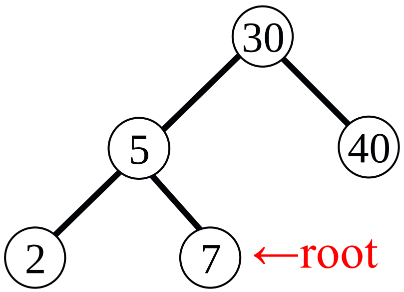
key = 7

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
     If there is no such node, return NULL. */
     if (!root)  return NULL;
     if (key == root->data)  return root;
     if (key < root->data)
          return search(root->left_child, key);
     return search(root->right_child, key);
}
```



key = 7

```
tree_pointer search(tree_pointer root, int key)
{ /* return a pointer to the node that contains key.
     If there is no such node, return NULL. */
     if (!root)  return NULL;
     if (key == root->data)  return root;
     if (key < root->data)
          return search(root->left_child, key);
     return search(root->right_child, key);
}
```

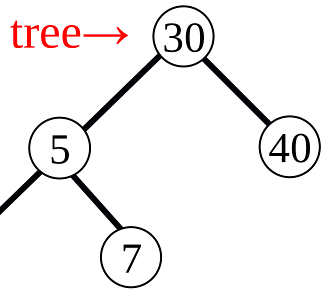


key = 7

[Program 5.16] Iterative search of a binary search tree

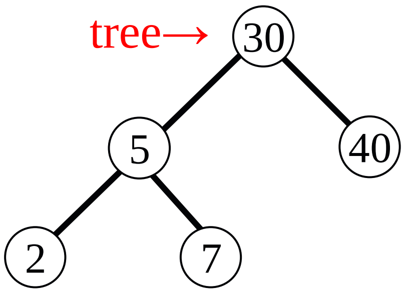
```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```

```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```



key = 7

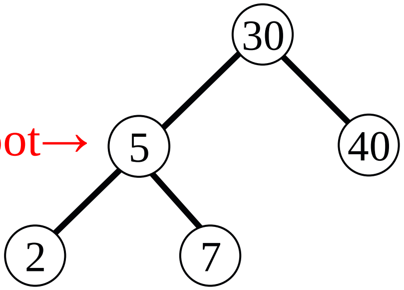
```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```



key = 7

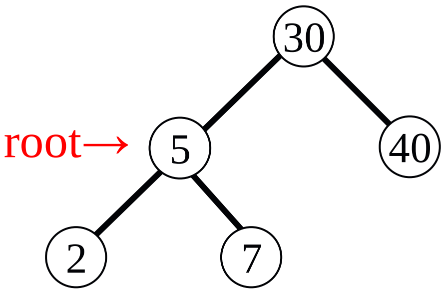
```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```

root→



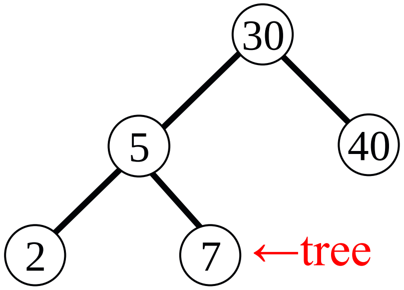
key = 7

```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```



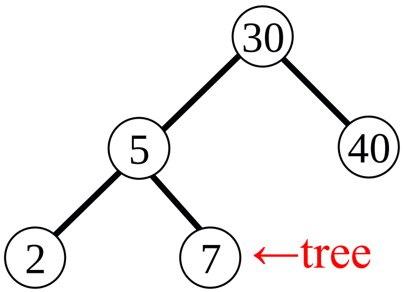
key = 7

```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```



key = 7

```
tree_pointer iter_search(tree_pointer tree, int key)
{ /* return a pointer to the node that contains key.
    If there is no such node, return NULL. */
    while (tree)  {
        if (key == tree->data)  return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
        }
    return NULL;
}
```



key = 7

Analysis of *search* and *iter_search* :

If h is the height of the binary search tree,
then the time complexity of both *search* and *iter_search* is O(h).
However, *search* has an additional stack space requirement which is O(h).

Searching a binary tree is similar to the binary search of a sorted list.

5.7.3 Inserting Into A Binary Search Tree

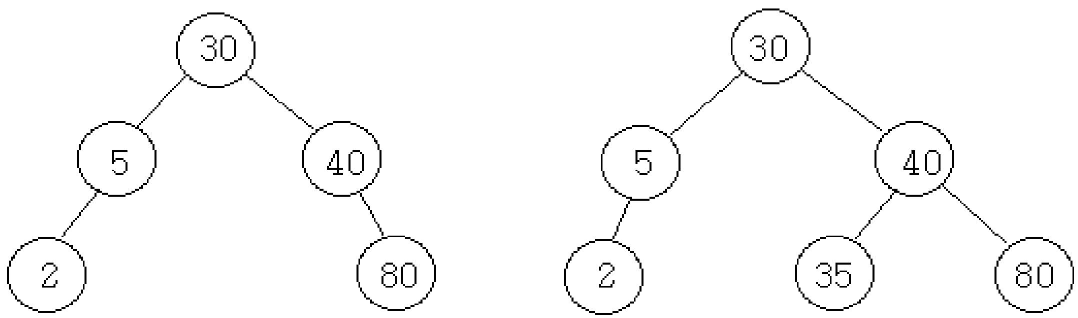
To insert a new element, key :

First, we verify that the key is different from those of existing elements by searching the tree.

If the search is unsuccessful,

then we insert the element at the point the search terminated.

[Figure 5.30]



(a) Insert 80 (b) Insert 35

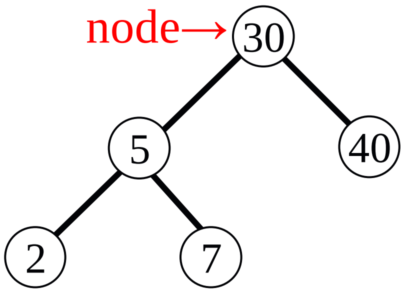
[Program 5.17] Inserting an element into a binary search tree

```
void insert(tree_pointer *node, int num)
{ /* If num is in the tree pointed at by node do nothing;
    otherwise add a new node with data = num  */
    tree_pointer  ptr, temp = modified_search(*node, num);
    if (temp || !(*node))  {
        /* num is not in the tree  */
        ptr = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(ptr)) {
            fprintf(stderr, "The memory is full");
            exit(1);
        }
```

```
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if (*node) /* insert as child of temp  */
            if (num < temp->data)
                temp->left_child = ptr;
            else  temp->right_child = ptr;
        else  *node = ptr;
    }
}
```

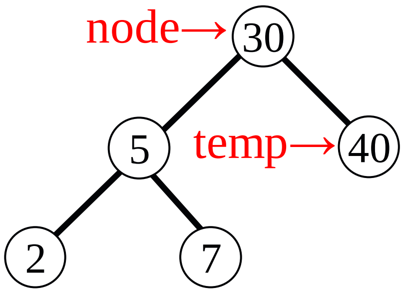
function *modified_search* searches the binary search tree *node for the key *num*. If the tree is empty or if *num* is presented, it returns NULL.
Otherwise, it returns a pointer to the last node of the tree that was encountered during the search.

```
void insert(tree_pointer *node, int num)
{ /* If num is in the tree pointed at by node do nothing;
    otherwise add a new node with data = num  */
    tree_pointer  ptr, temp = modified_search(*node, num);
    if (temp || !(*node))  {
        /* num is not in the tree  */
        ptr = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(ptr)) {
            fprintf(stderr, "The memory is full");
            exit(1);
        }
```



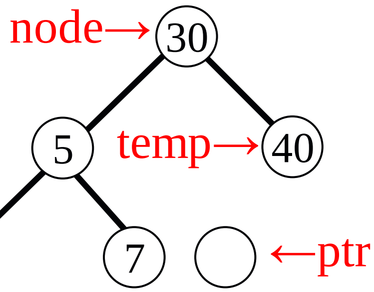
num = 35

```
void insert(tree_pointer *node, int num)
{ /* If num is in the tree pointed at by node do nothing;
     otherwise add a new node with data = num  */
     tree_pointer  ptr, temp = modified_search(*node, num);
     if (temp || !(*node))  {
          /* num is not in the tree  */
          ptr = (tree_pointer) malloc(sizeof(node));
          if (IS_FULL(ptr)) {
               fprintf(stderr, "The memory is full");
               exit(1);
          }
```



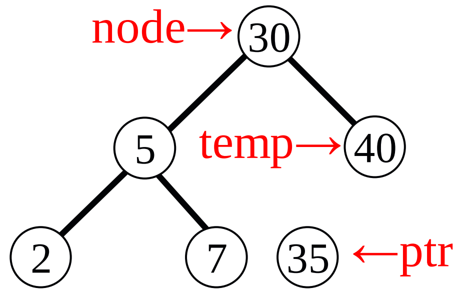
num = 35

```
void insert(tree_pointer *node, int num)
{ /* If num is in the tree pointed at by node do nothing;
    otherwise add a new node with data = num  */
    tree_pointer  ptr, temp = modified_search(*node, num);
    if (temp || !(*node))  {
        /* num is not in the tree  */
        ptr = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(ptr)) {
            fprintf(stderr, "The memory is full");
            exit(1);
        }
```



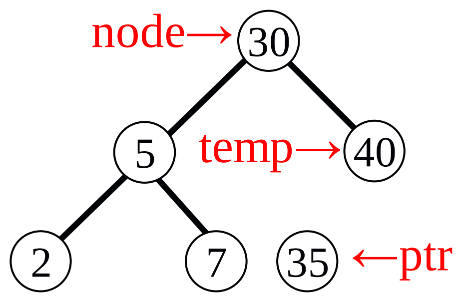
num = 35

```
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if (*node) /* insert as child of temp  */
            if (num < temp->data)
                temp->left_child = ptr;
            else  temp->right_child = ptr;
        else  *node = ptr;
    }
}
```



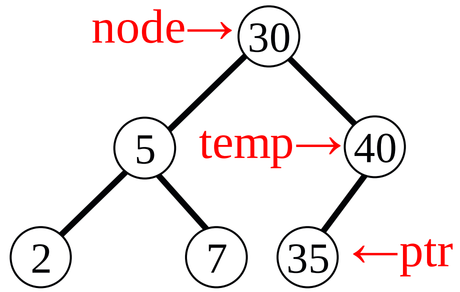
num = 35

```
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if (*node) /* insert as child of temp  */
            if (num < temp->data)
                temp->left_child = ptr;
            else  temp->right_child = ptr;
        else  *node = ptr;
    }
}
```



num = 35

```
ptr->data = num;
ptr->left_child = ptr->right_child = NULL;
if (*node) /* insert as child of temp  */
    if (num < temp->data)
        temp->left_child = ptr;
    else  temp->right_child = ptr;
else  *node = ptr;
}
}
```



num = 35

Analysis of *insert* :

The time required to search the tree for *num* is O(h) where h is its height. The remainder of the algorithm takes Θ(1) time.

So, the overall time needed by *insert* is O(h).

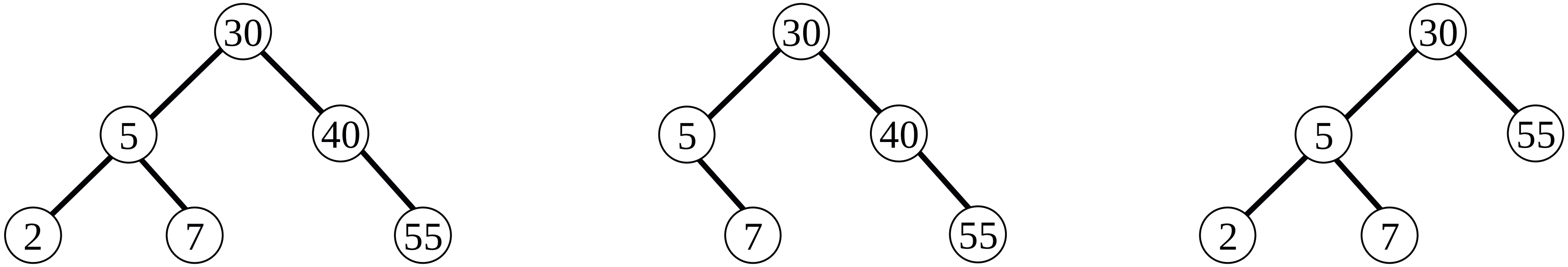
5.7.4 Deletion From A Binary Search Tree

· Deletion of a leaf node :

Set the corresponding child field of its parent to NULL and free the node.

· Deletion of a nonleaf node with single child :

Erase the node and then place the single child in the place of the erased node.



delete 2

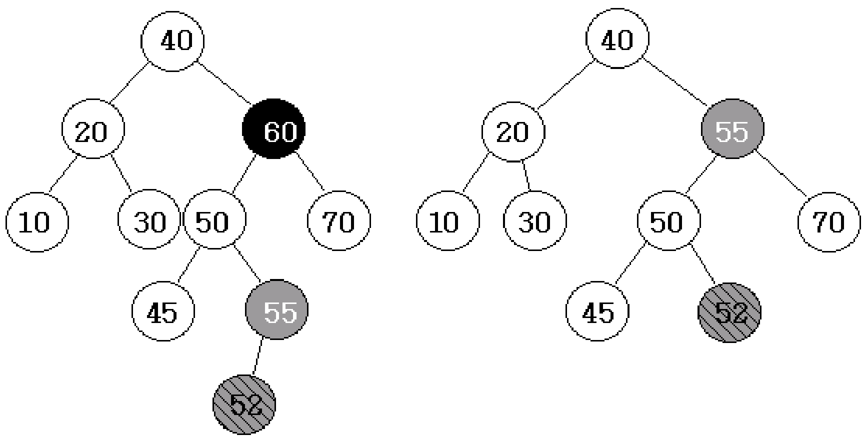
delete 40

· **Deletion of a nonleaf node with two children** :

Replace the node with either the largest element in its left subtree or the smallest element in its right subtree.

Then delete this replacing element from the subtree from which it was taken.

Note that the largest and smallest elements in a subtree are always in a node of degree zero or one.



(a) tree before deletion of 60 (b) tree after deletion of 60

It is easy to see that a deletion can be performed in O(h) time, where h is the height of the binary search tree.

5.7.6 Height Of A Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n.

However, when insertion and deletions are made at random, the height of the binary search tree is O($\log_2 n$), on the average.

Search trees with a worst case height of O($\log_2 n$) are called *balanced search trees*.

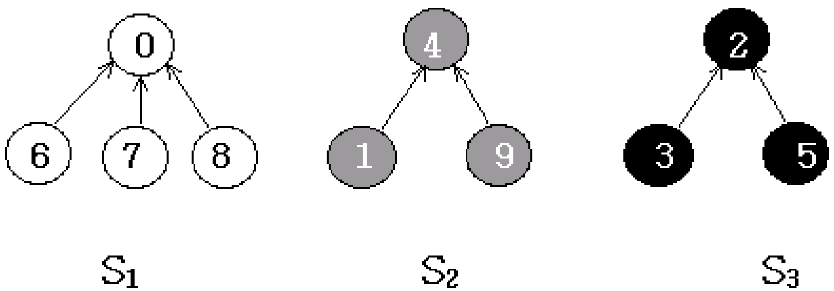
5.10 REPRESENTATION OF DISJOINT SETS

In this section, We study the use of trees in the representation of sets.

For simplicity, we assume that the elements of the sets are the numbers 0, 1, 2, . . ., n-1.

We also assume that the sets being represented are pairwise disjoint, that is, if S_i and S_j are two sets and $i \neq j$, then there is no element that is in both S_i and S_j.

[Figure 5.37] Possible tree representation of sets



Notice that for each set the nodes are linked from the children to the parent.

The operations to perform on these sets are:

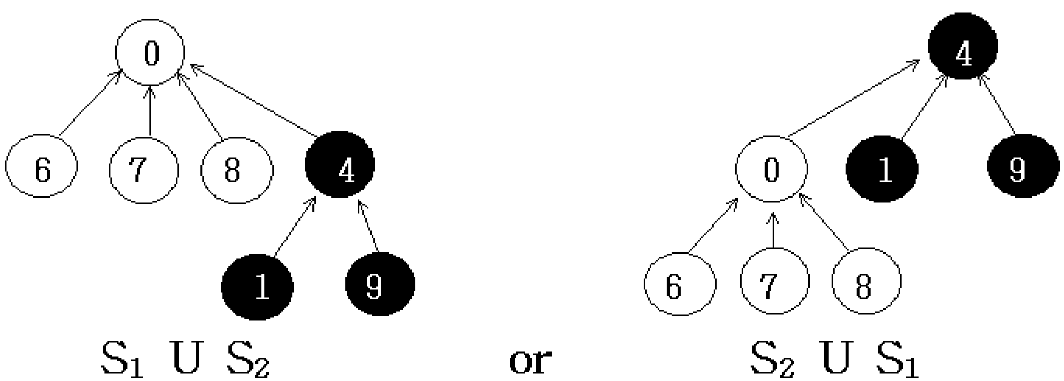
(1) *Disjoint set union*. If we wish to get the union of two disjoint sets Si and Sj, replace Si and Sj by Si∪Sj.

(2) *Find(i)*. Find the set containing the element, *i*.

5.10.1 Union and Find Operations

Suppose that we wish to obtain the union of S1 and S2.
We simply make one of the trees a subtree of the other.
S1∪S2 could have either of the representations of Figure 5.38

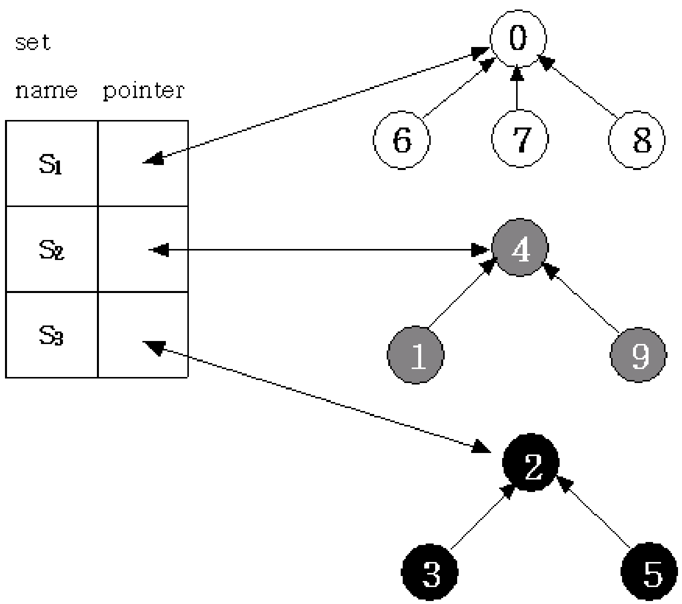
[Figure 5.38] Possible representation of S1∪S2



To implement the set union operation, we simply set the parent field of one of the roots to the other root.

Figure 5.39 shows how to name the sets.

[Figure 5.39] Data representation of S_1, S_2 and S_3



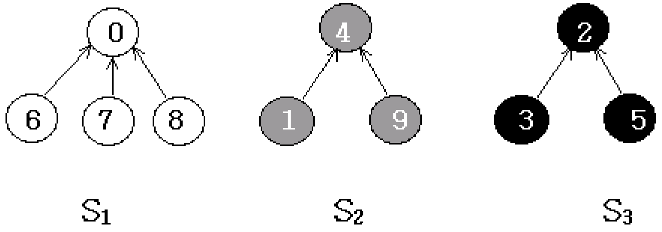
To simplify the discussion of the union and find algorithms, we will ignore the set names and identify the sets by the roots of the trees representing them.

Since the nodes in the trees are numbered 0 through n-1,
we can use the node's number as an index.

[Figure 5.40] Array representation of the trees in Figure 5.39.

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

Notice that root nodes have a parent of −1.



We can implement find(i) by simply following the indices starting at i and continuing until we reach a negative parent index.

[Program 5.19] Initial attempt at union-find functions.

```
int simple_find(int i)
{
    for ( ; parent[i] >= 0 ; i = parent[i])
        ;
    return  i;
}
void simple_union(int i, int j)
{
    parent[i] = j;
}
```

Analysis of *simple_union* and *simple_find* :

Let us process the following sequence of union-find operations:

union(0, 1),   find(0)

union(1, 2),   find(0)

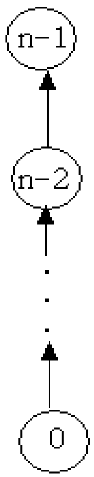
.

.

.

union(n-2, n-1),   find(0)

This sequence produces the degenerate tree of Figure 5.41.



[Figure 5.41] Degenerate tree

Since the time taken for a union is constant,
all the n-1 unions can be processed in time O(n).
For each *find*, if the element is at level *i*,
then the time required to find its root is O(*i*).
Hence the total time needed to process the n-1 finds is :

$$\sum_{i=2}^{n} i = O(n^2)$$

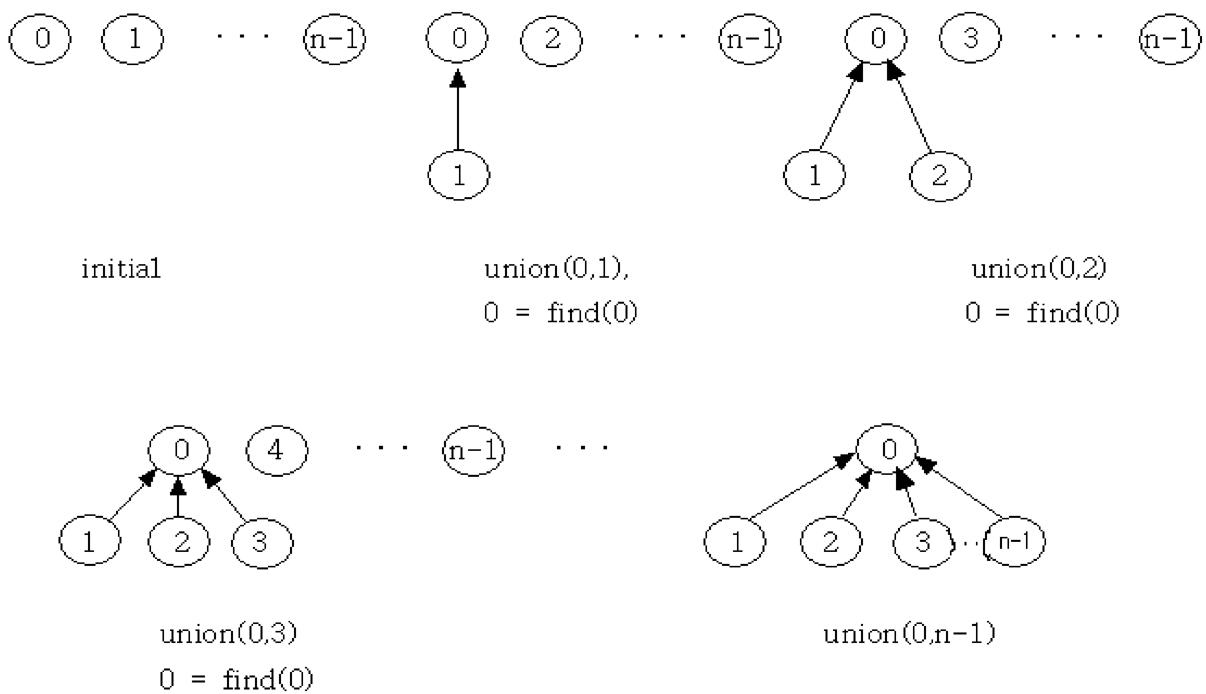
By avoiding the creation of degenerate trees,
we can attain far more efficient implementations of the union and find operations.

Definition : Weighting rule for *union*(i, j). If the number of nodes in tree i is less than the number in tree j then make j the parent of i; otherwise make i the parent of j. □

When we use this rule on the sequence of set unions described above,

we obtain the trees of Figure 5.42.

[Figure 5.42] Trees obtained using the weighting rule



To implement the weighting rule, we need to know how many nodes there are in every tree.

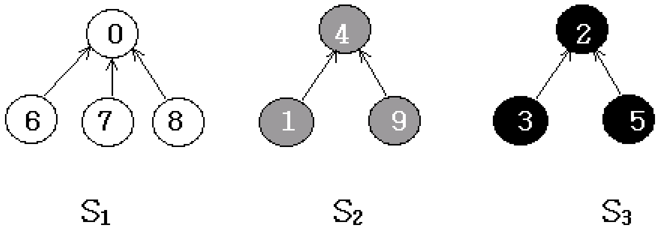
That is, we need to maintain a count field in the root of every tree.

We can maintain the count in the parent field of the roots as a negative number.

[Program 5.20] Union function using weighting rule

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j])   {
        parent[i] = j;   /* make j the new root  */
        parent[j] = temp;
    }
    else  {
        parent[j] = i;   /* make i the new root  */
        parent[i] = temp;
    }
}
```

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
      int temp = parent[i] + parent[j];
      if (parent[i] > parent[j])   {
            parent[i] = j;   /* make j the new root  */
            parent[j] = temp;
      }
      else  {
            parent[j] = i;   /* make i the new root  */
            parent[i] = temp;
      }
}
```

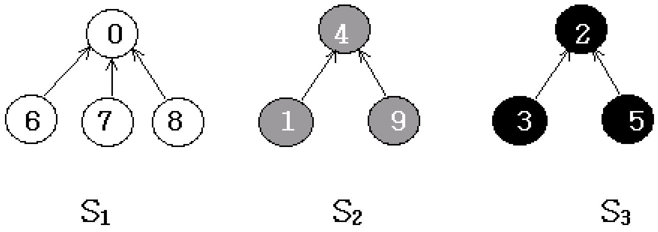


i	0
j	4
temp	

parent

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-4	4	-3	2	-3	2	0	0	0	4

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j])  {
        parent[i] = j;   /* make j the new root  */
        parent[j] = temp;
    }
    else  {
        parent[j] = i;   /* make i the new root  */
        parent[i] = temp;
    }
}
```

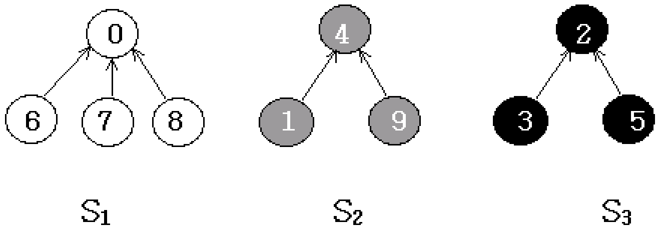


i	0
j	4
temp	-7

parent

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-4	4	-3	2	-3	2	0	0	0	4

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
      int temp = parent[i] + parent[j];
      if (parent[i] > parent[j])   {
            parent[i] = j;   /* make j the new root  */
            parent[j] = temp;
      }
      else  {
            parent[j] = i;   /* make i the new root  */
            parent[i] = temp;
      }
}
```

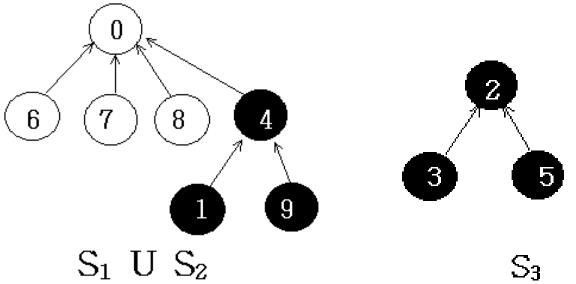


i	0
j	4
temp	-7

parent

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-4	4	-3	2	-3	2	0	0	0	4

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
     int temp = parent[i] + parent[j];
     if (parent[i] > parent[j])   {
          parent[i] = j;   /* make j the new root  */
          parent[j] = temp;
     }
     else  {
          parent[j] = i;   /* make i the new root  */
          parent[i] = temp;
     }
}
```

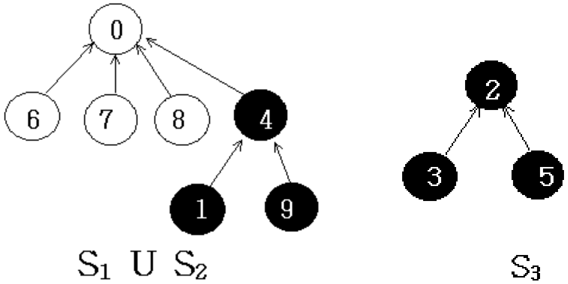


i	0
j	4
temp	-7

parent

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-4	4	-3	2	0	2	0	0	0	4

```
void weighted_union(int i, int j)
{ /*  parent[i] = -count[i] and parent[j] = -count[j]  */
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j])   {
        parent[i] = j;   /* make j the new root   */
        parent[j] = temp;
    }
    else  {
        parent[j] = i;   /* make i the new root   */
        parent[i] = temp;
    }
}
```



i	0
j	4
temp	-7

parent

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-7	4	-3	2	0	2	0	0	0	4

Lemma 5.5: Let T be a tree with n nodes created as a result of *weighted_union*. No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$.

<Proof>

The lemma is clearly true for $n = 1$.

Assume that it is true for all trees with i nodes, $i \leq n - 1$.

We show that it is also true for $i = n$.

Let T be a tree with n nodes created by *weighted_union*.

Consider the last union operation performed, *union(k, j)*.

Let m be the number of nodes in tree j and $n - m$, the number of nodes in k.

Without loss of generality, we may assume that $1 \leq m \leq n/2$.

Then the maximum level of any node in T is either the same as k or is one more than in j.

If the former is the case, then the maximum level in T is $\leq \lfloor \log_2 (n - m) \rfloor + 1 \leq \lfloor \log_2 n \rfloor + 1$.

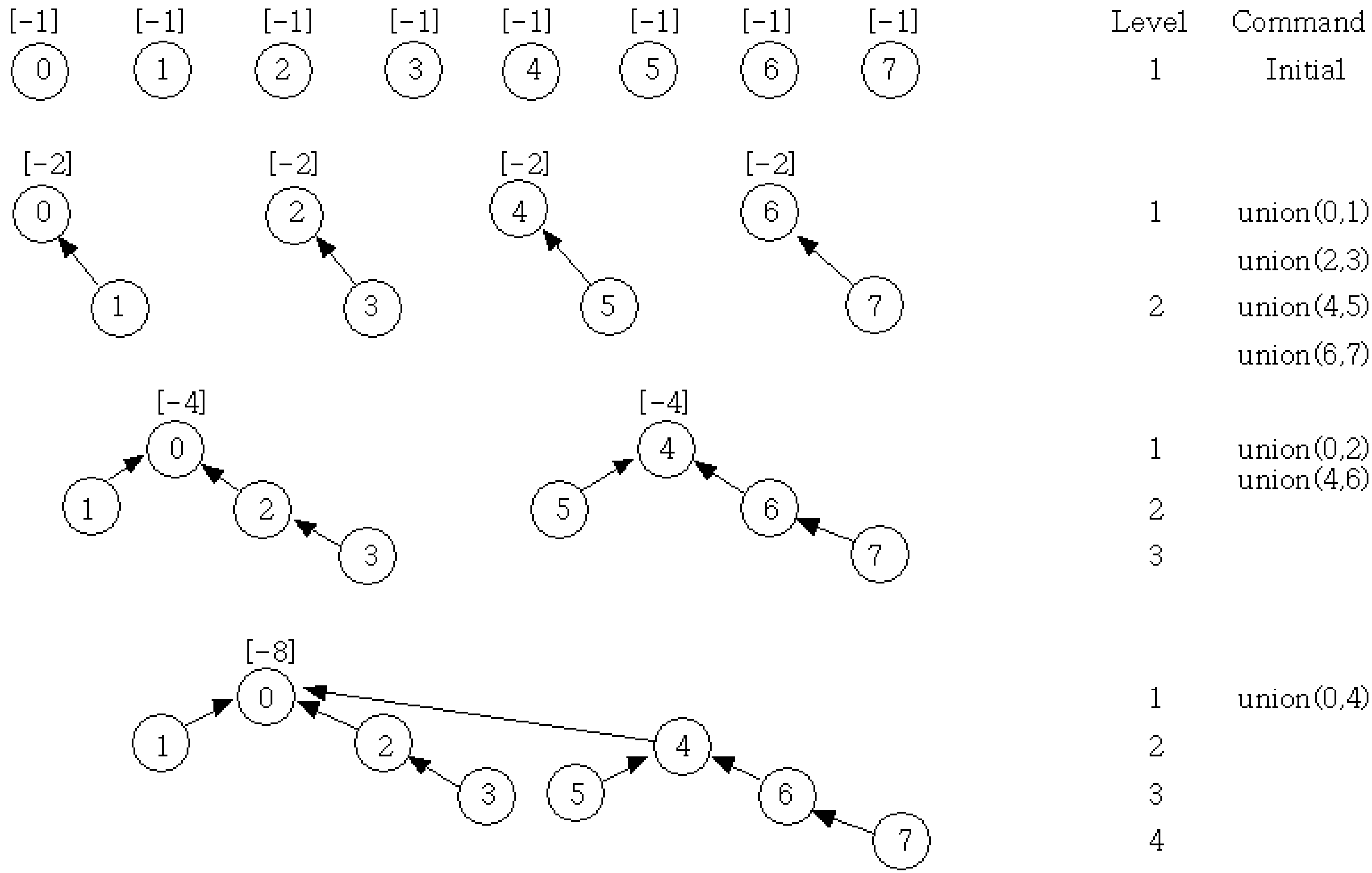
If the latter is the case, then the maximum level is $\leq \lfloor \log_2 m \rfloor + 2 \leq \left\lfloor \log_2 \frac{n}{2} \right\rfloor + 2 \leq \lfloor \log_2 n \rfloor + 1$. □

Example 5.3: Consider the behavior of weighted_union on the following sequence of unions starting from the initial configuration of *parent* [i] = -*count* [i] = -1, $0 \leq i < n = 2^3$:

union(*0*, *1*)   *union*(*2*, *3*)   *union*(*4*, *5*)   *union*(*6*, *7*)
union(*0*, *2*)   *union*(*4*, *6*)   *union*(*0*, *4*)

Figure 5.43 shows the result.

Figure 5.43: Trees achieving worst case bound



As is evident from this example, in the general case, the maximum level can be $\lfloor \log_2 m \rfloor + 1$ if the tree has m nodes.

As a result of Lemma 5.5,

the time to process a *find* in an m element tree is O($\log m$).

If we process an intermixed sequence of $u - 1$ *union* and

f *find* operations, then the time becomes O($u + f \log u$).