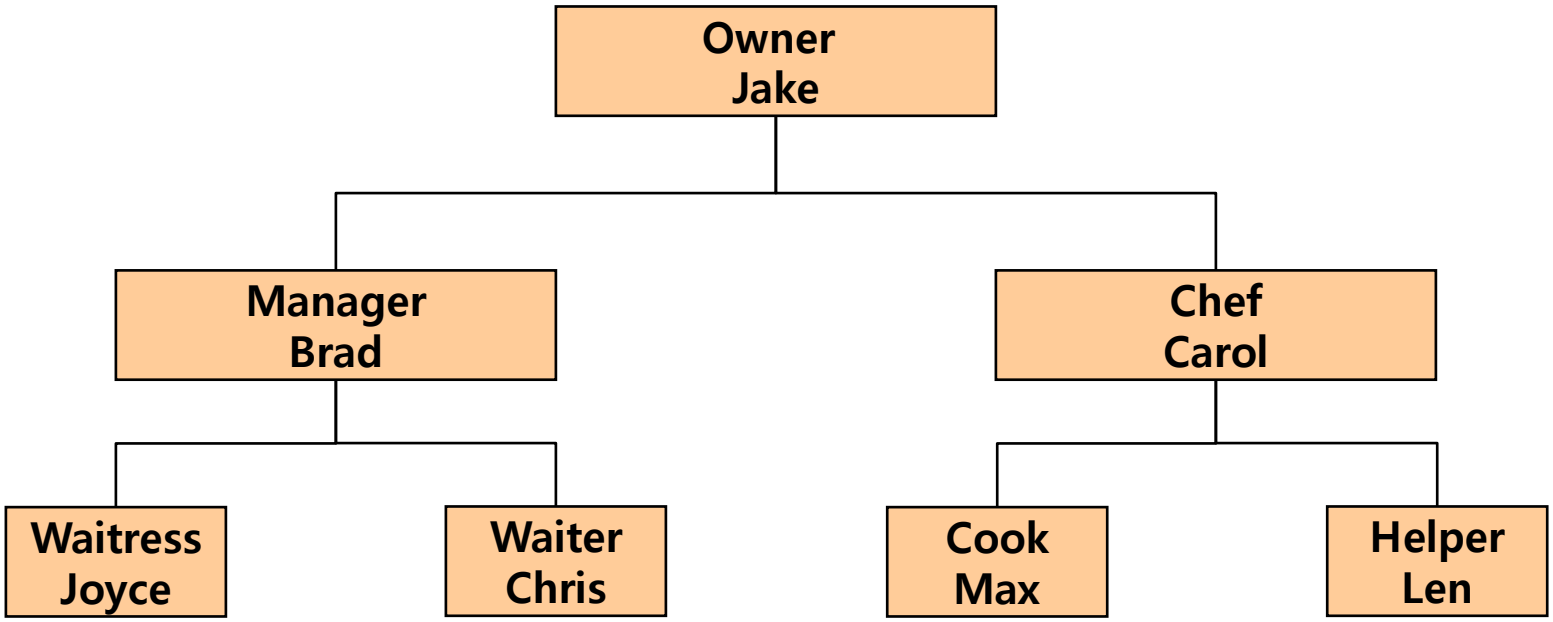
Data Structures

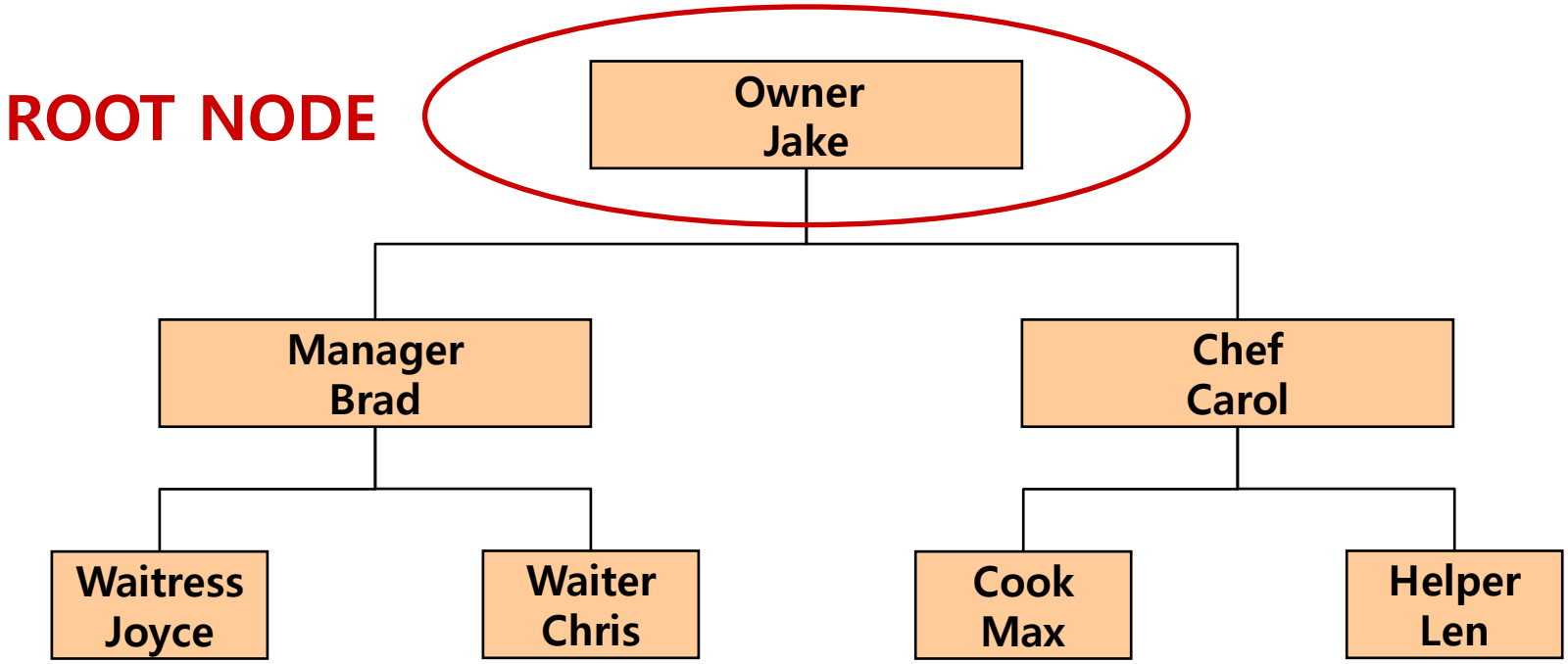
Part VIII: Binary Search Trees

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

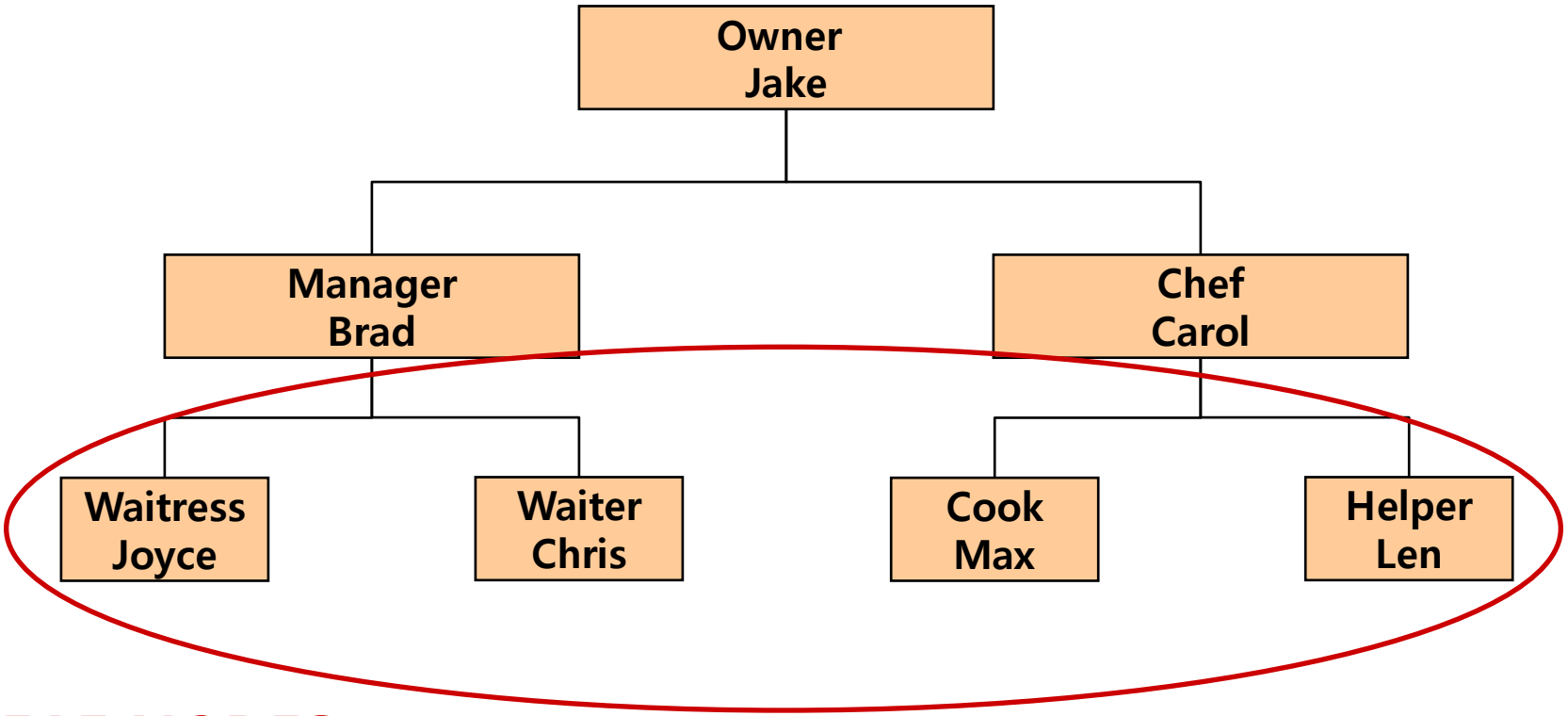
Example: Jake's Pizza Shop



A Tree Has a Root Node

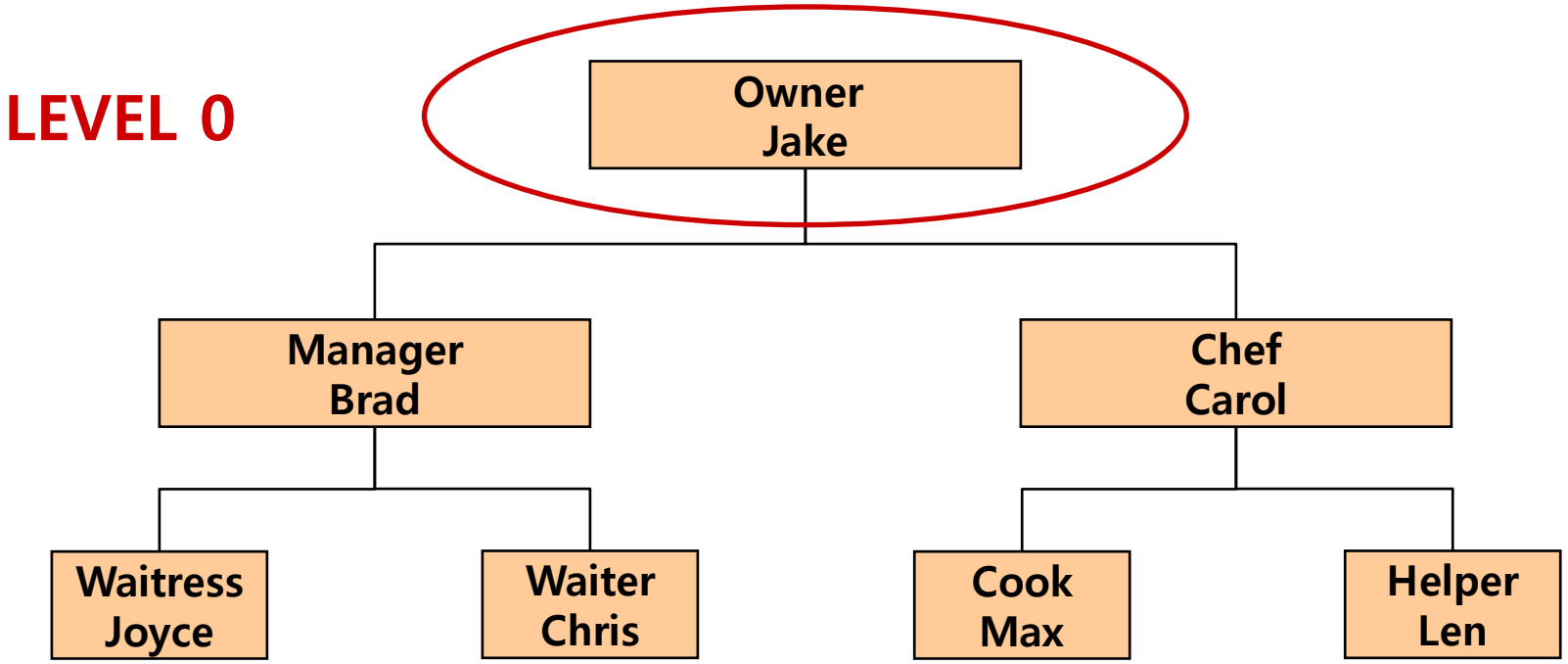


Leaf nodes have no children

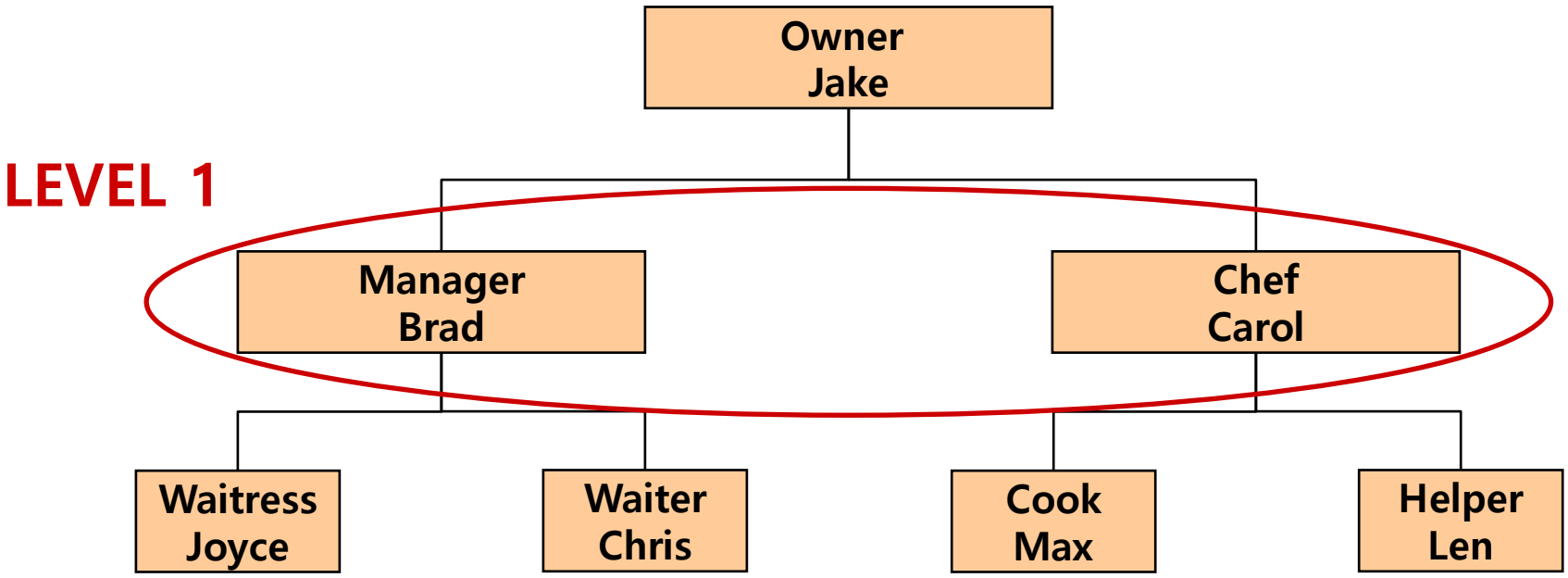


LEAF NODES

A Tree Has Levels

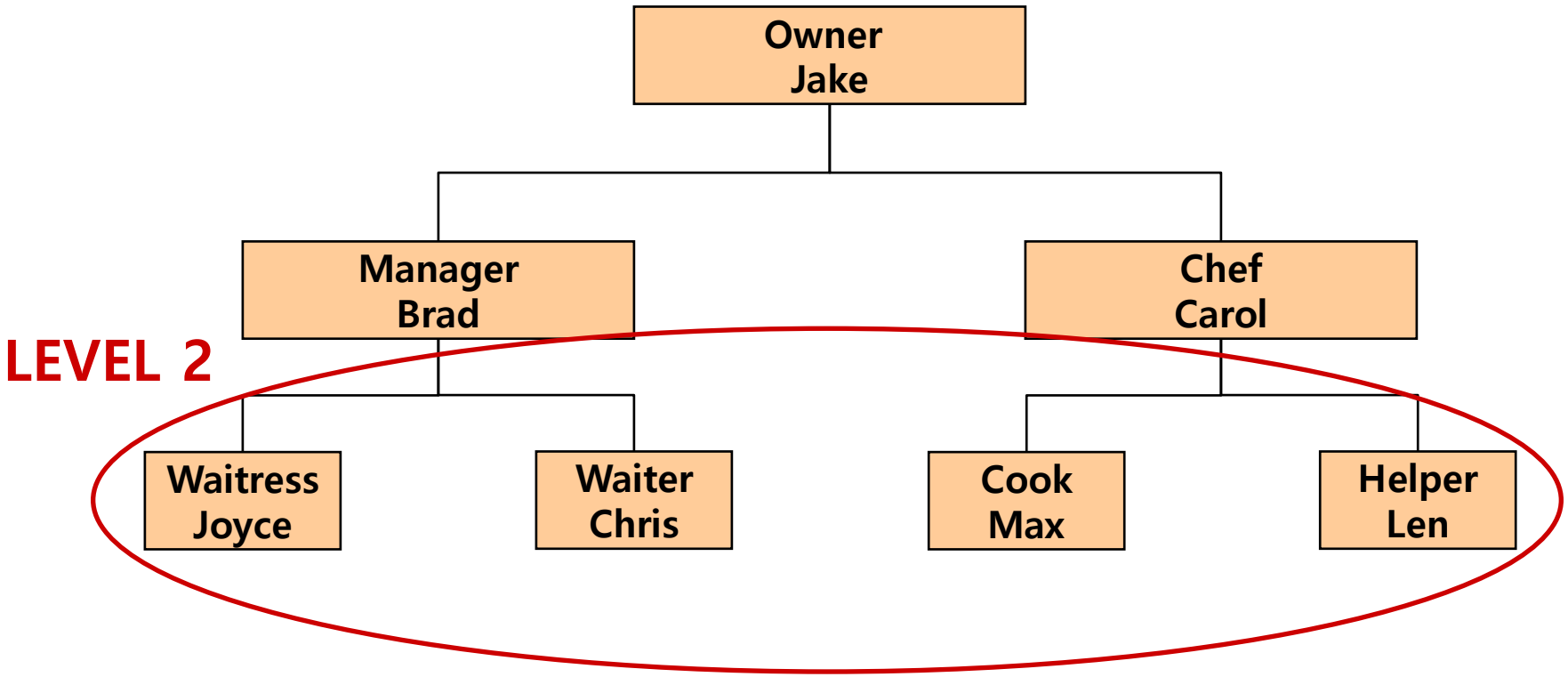


Level One

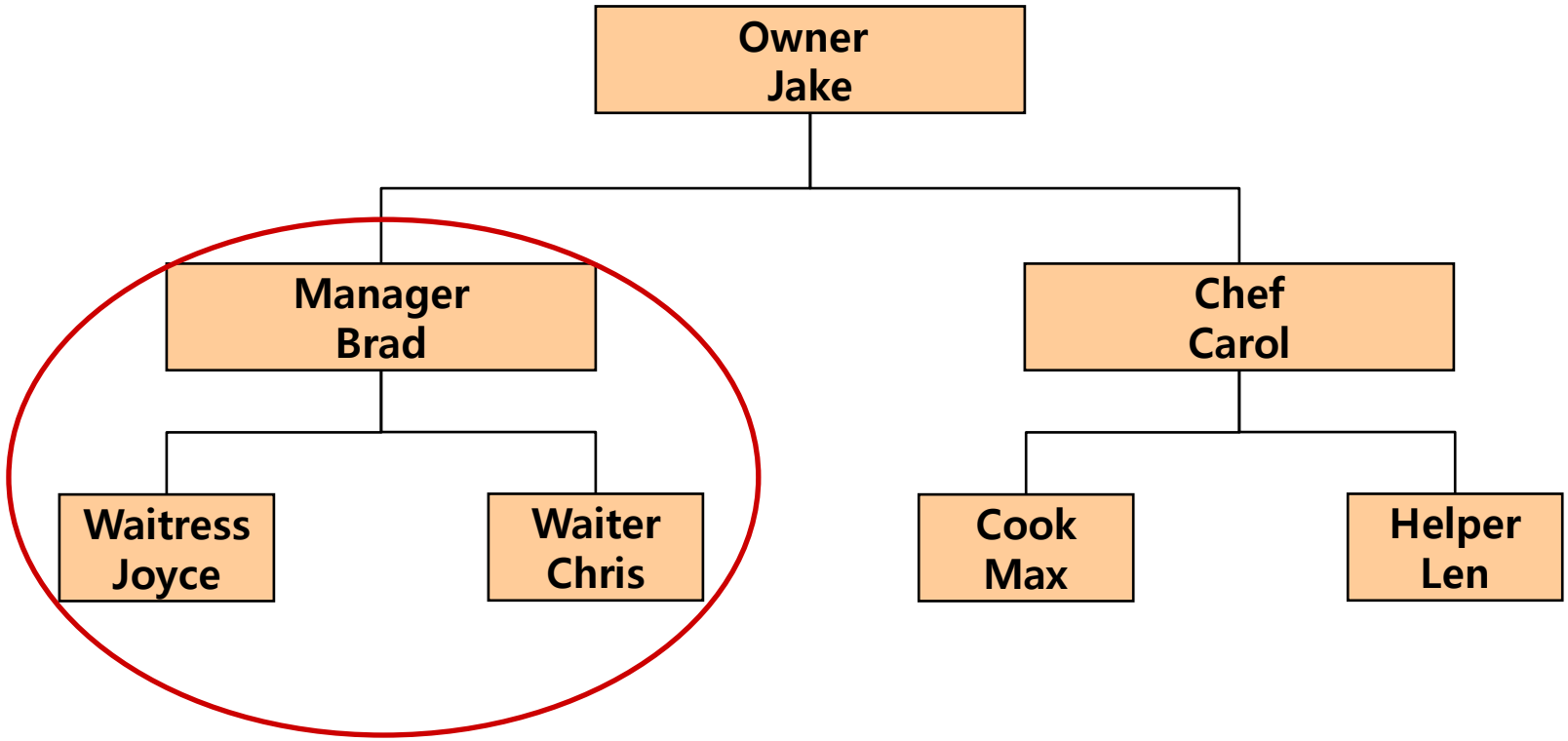

LEVEL 1
Owner
Jake
Manager
Brad
Chef
Carol
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

Level Two

- Tree의 높이는?

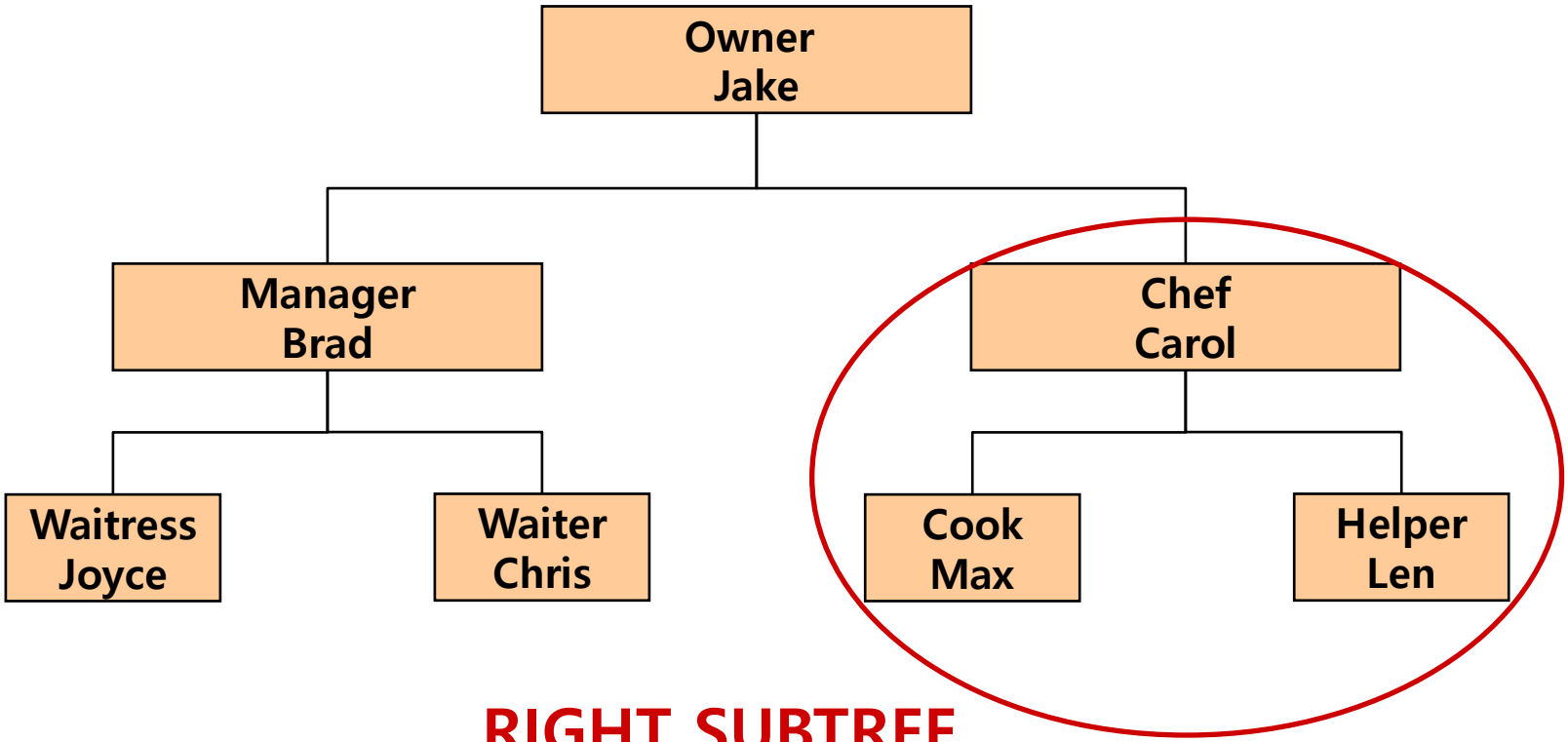


A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree



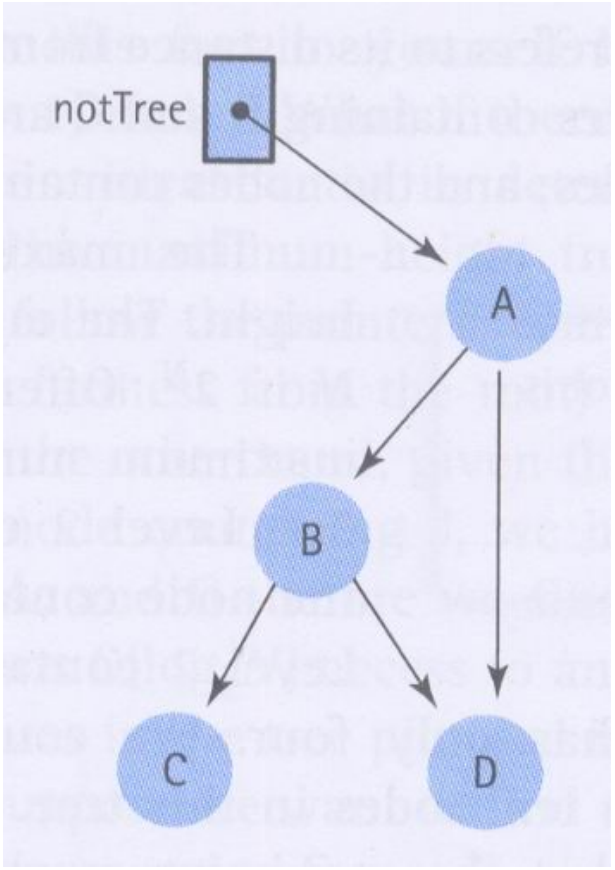
RIGHT SUBTREE OF ROOT NODE

Binary Tree

- 이진트리의 구조
  A binary tree is a structure in which:
  - ✓ 각 노드는 최대 2개의 자식을 가질 수 있으며,
    [1] Each node can have at most two children, and
  - ✓ 루트에서 다른 노트까지 고유 경로가 존재
    [2] in which a unique path exists from the root to every other node.

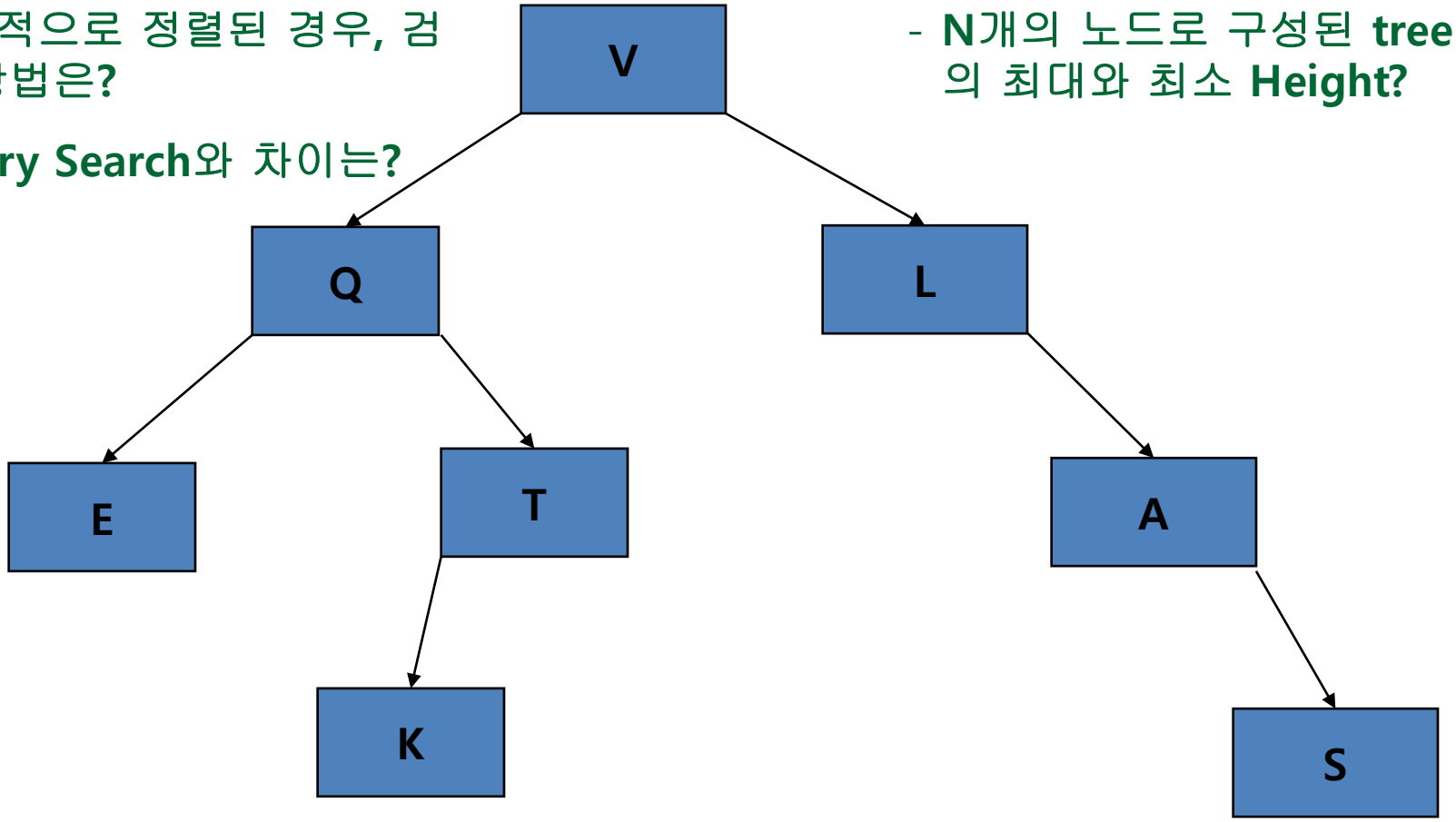
- 노드의 두 자식을 왼쪽자식, 오른쪽자식이라 함
  The two children of a node are called the left child and the right child, if they exist.

Binary Tree? If not, why?



A Binary Tree

- 순서적으로 정렬된 경우, 검색 방법은?
- **Binary Search**와 차이는?



- **N**개의 노드로 구성된 **tree**의 최대와 최소 **Height?**

Level & Balance

- N level tree가 가질 수 있는 최대 노드 수?
- 탐색 시간과 level과의 관계는?
- 각 level이 가질 수 있는 최대 노드의 수는?

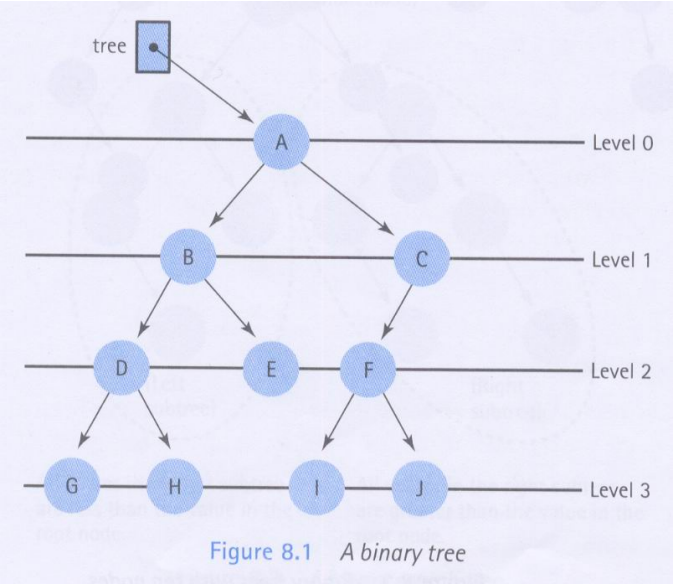


Figure 8.1 *A binary tree*

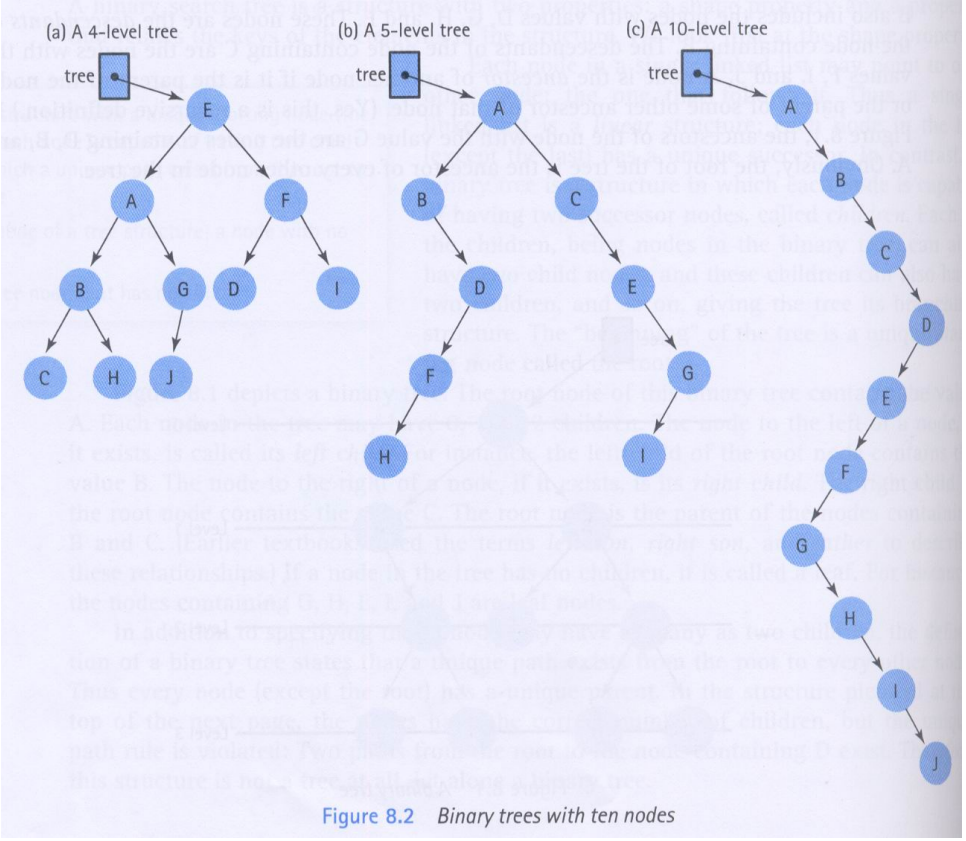
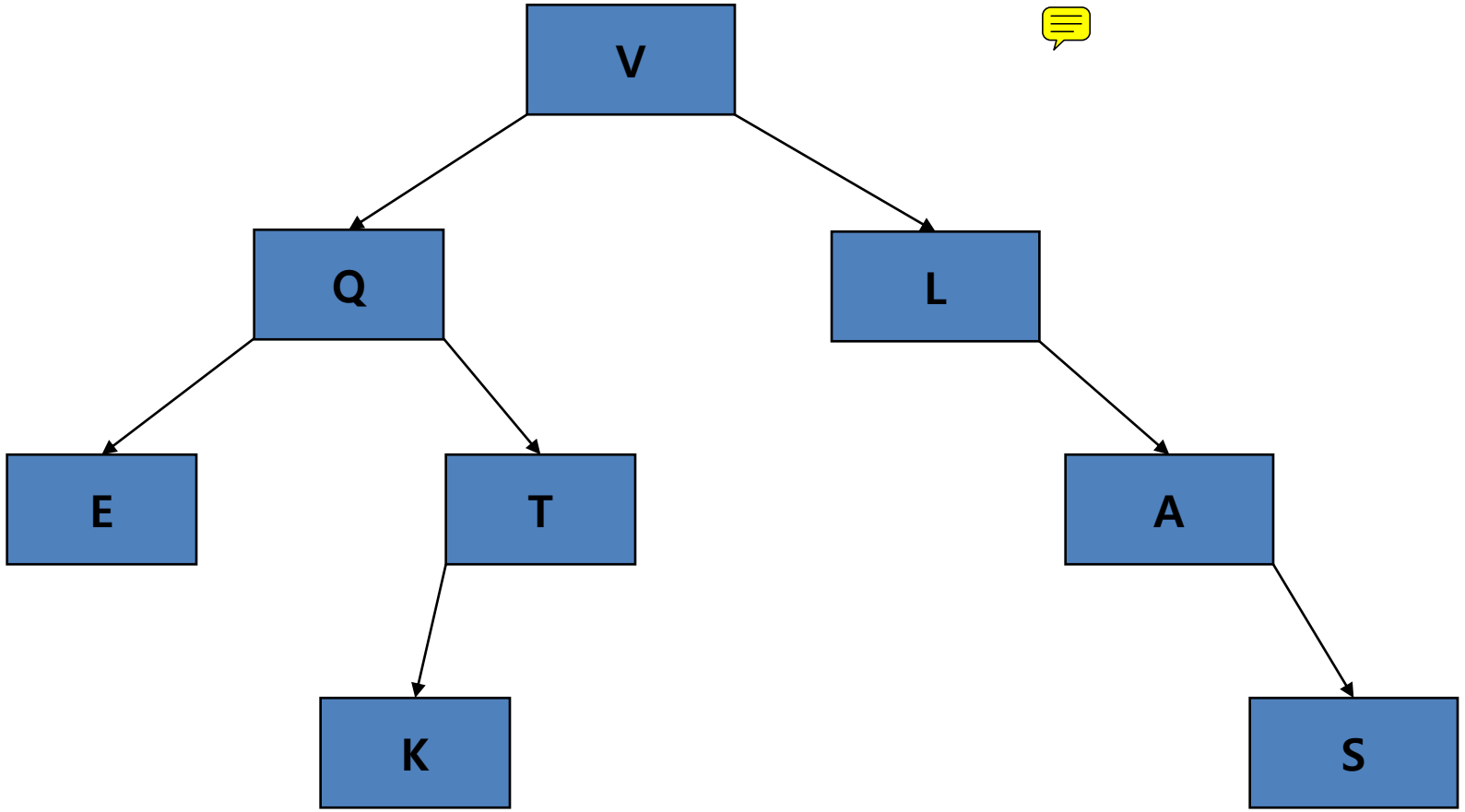
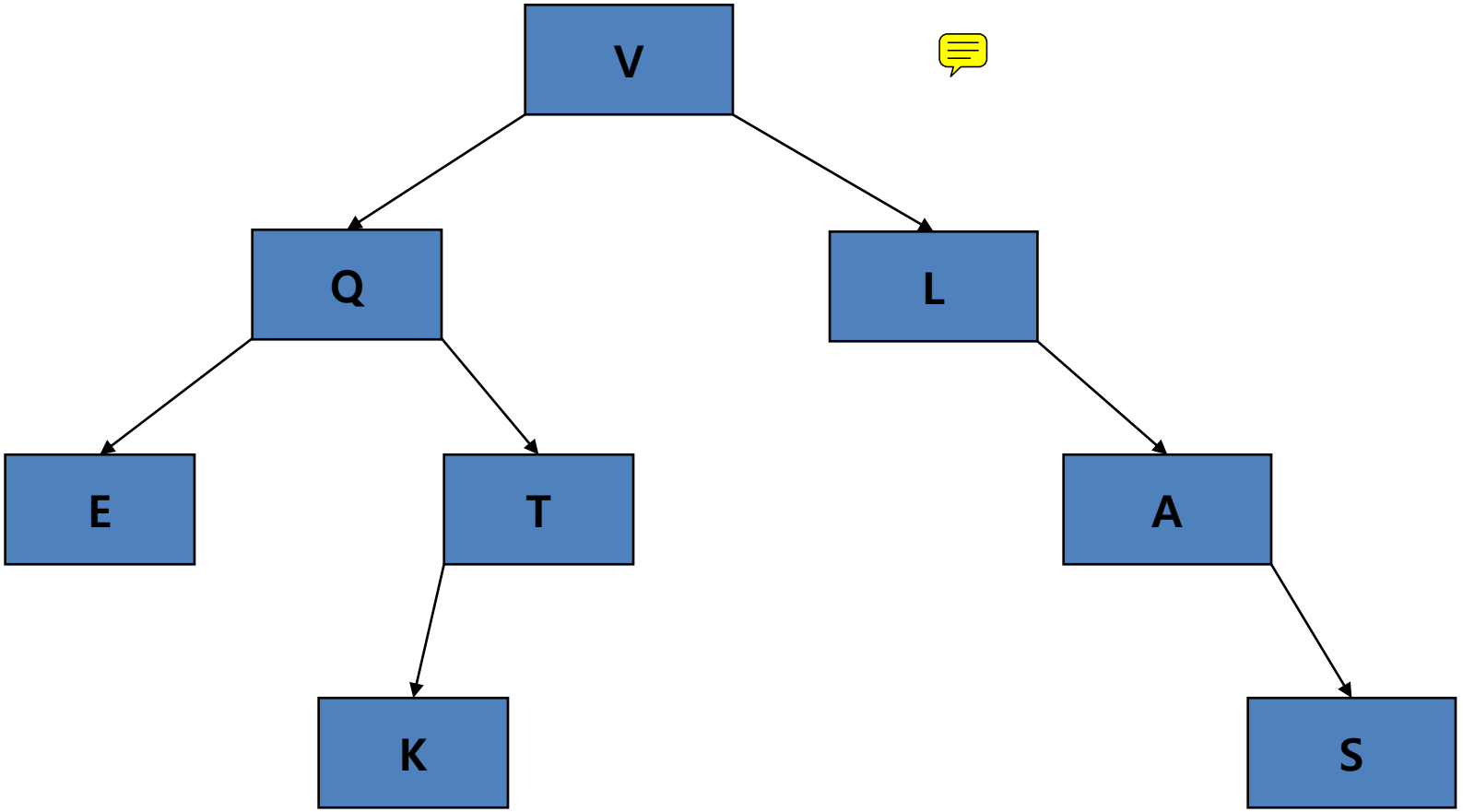


Figure 8.2 *Binary trees with ten nodes*

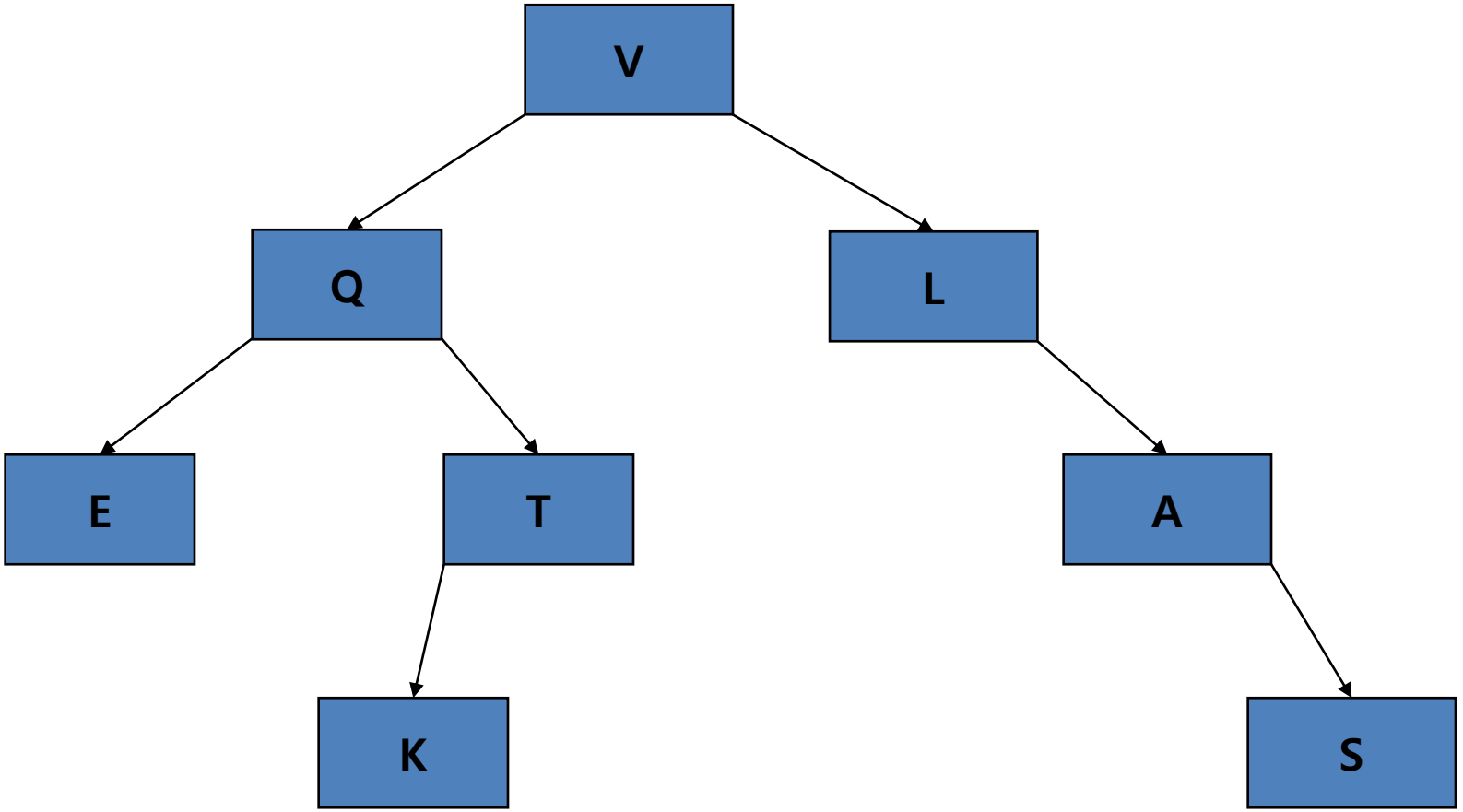
How many leaf nodes?



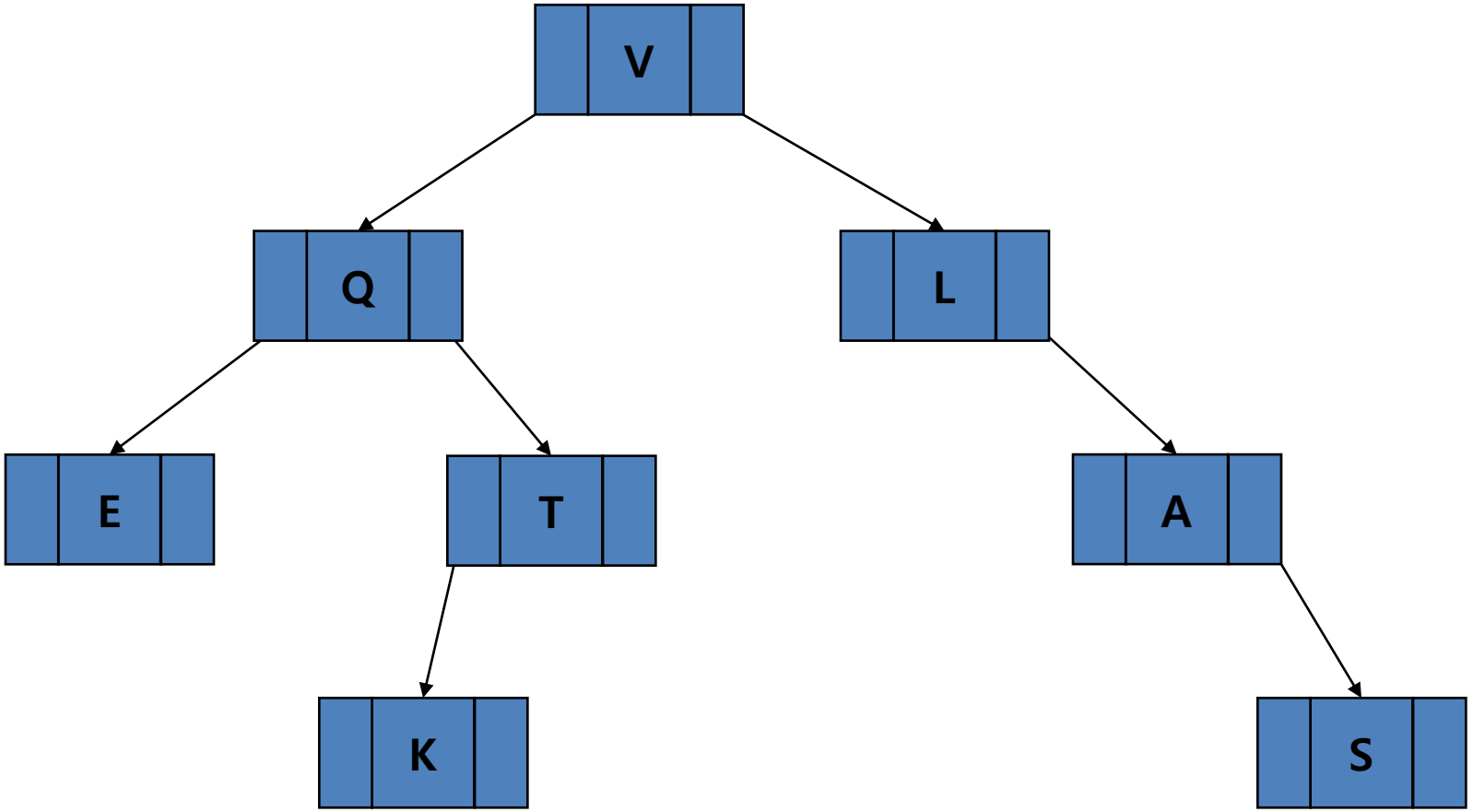
How many descendants of Q?



How many ancestors of K?

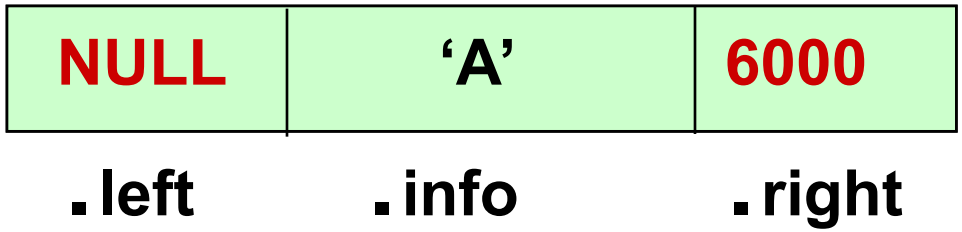


Implementing a Binary Tree with Pointers and Dynamic Data



Each node contains two pointers

```
template< class  ItemType >
struct   TreeNode
 {
    ItemType     info;                    // Data member
    TreeNode<ItemType>*  left;     // Pointer to left child
    TreeNode<ItemType>*  right;   // Pointer to right child
};
```



Class TreeType ADT

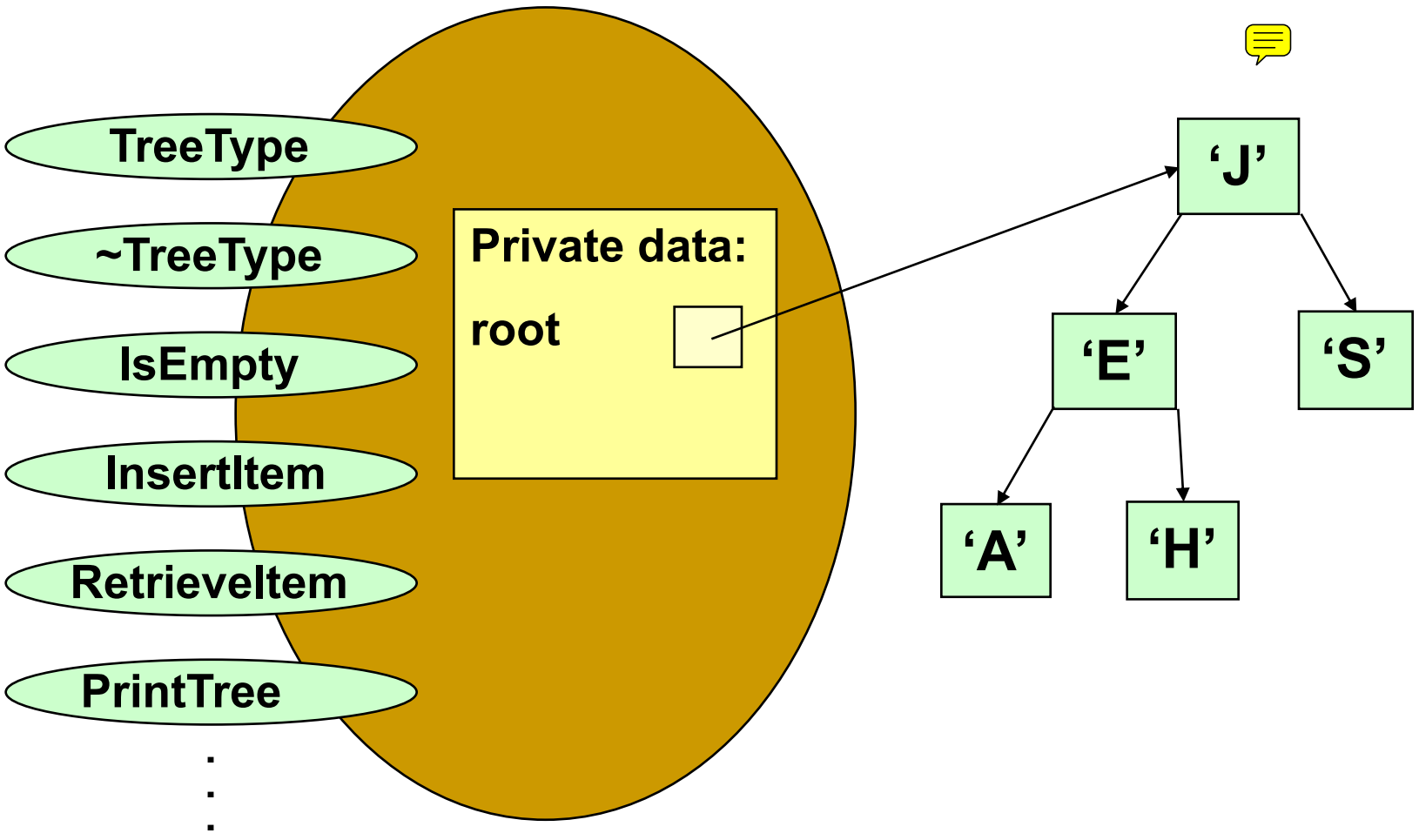
```
class TreeType
{
public:
    TreeType();                          // constructor
    ~TreeType();                         // destructor
    TreeType(const TreeType& originalTree); // copy constructor
    void operator=(const TreeType& originalTree);
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void ResetTree(OrderType order);
    int LengthIs() const;
    void RetrieveItem(ItemType& item, bool& found) const;
```

Class TreeType ADT (cont'd)

```
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void GetNextItem (ItemType& item, OrderType order, bool& finished);
    void Print(std::ofstream& outFile) const;
private:
    TreeNode* root;
};

struct TreeNode
{
    ItemType info;
    TreeNode* left;
    TreeNode* right;
};
```

TreeType<char> CharBST;



A Binary Search Tree (BST) is . . .

- 아래와 같은 특별한 성격을 가지는 이진트리
  A special kind of binary tree in which:

  ✓ 각 노드에는 고유한 데이터 값이 존재
  1. Each node contains a distinct data value,

  ✓ 트리의 키 값은 “보다 큼”과 “보다 작음”을 사용하여 비교 가능
  2. The key values in the tree can be compared using “greater than” and “less than”, and

  ✓ 트리에서 각 노드의 키 값은 오른쪽 하위 트리의 모든 키 값보다 작고 왼쪽 하위 트리의 모든 키 값보다 큼
  3. The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

Subtree of a binary tree

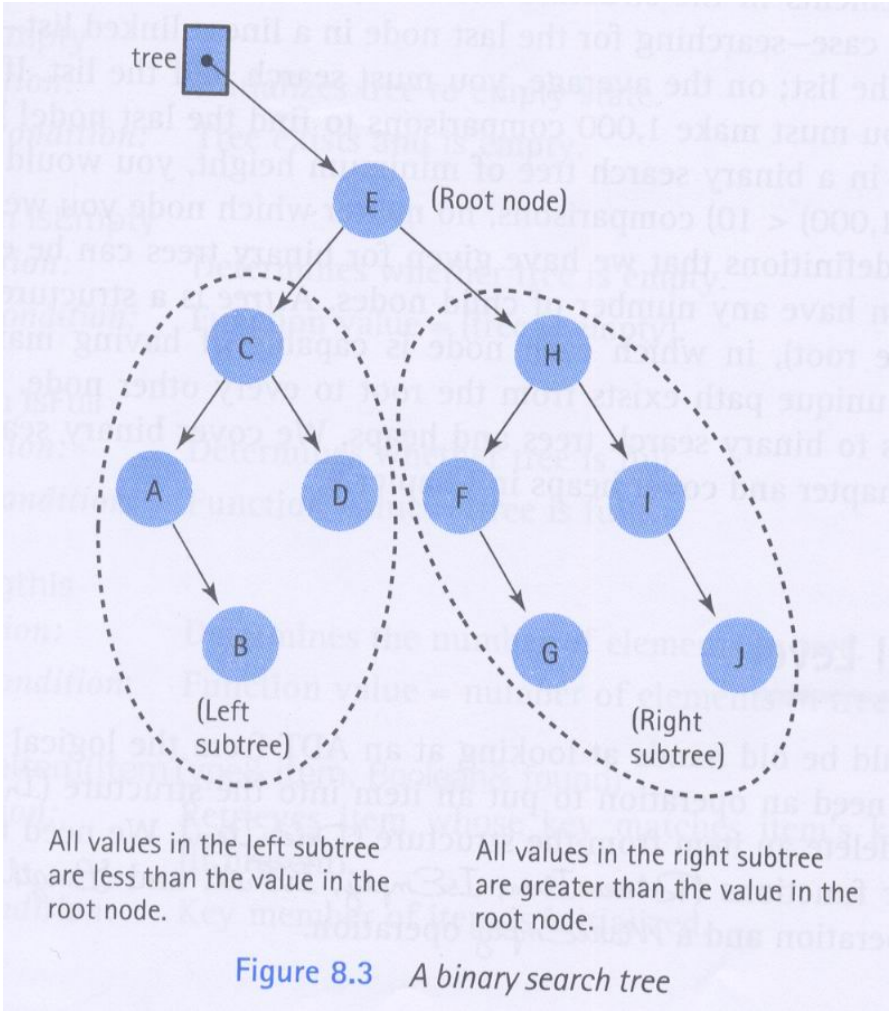


Figure 8.3 *A binary search tree*

Left Tree < Right Tree

Shape of a binary search tree . . .

- 키 값과 삽입 순서에 따라 다름
  Depends on its key values and their order of insertion.
- ‘J’, ‘E’, ‘F’, ‘T’, ‘A’ 순서대로 삽입
  Insert the elements ‘J’ ‘E’ ‘F’ ‘T’ ‘A’ in that order.
- 삽입 할 첫 번째 값이 루트 노드에 저장됨
  The first value to be inserted is put into the root node.

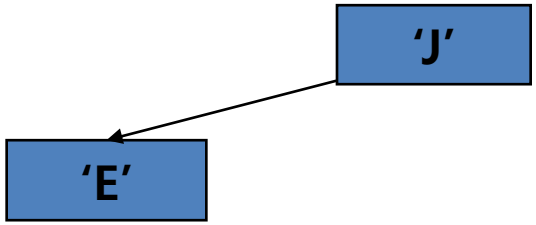
Inserting ‘E’ into the BST

- 다음, 삽입 할 각 값은 루트 노드의 값과 비교하여 더 작으면 왼쪽으로 이동하거나 더 크면 오른쪽으로 이동하여 시작됨

  Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left if it is less, or moving right if it is greater.

  - ✓ 새 리프에 삽입될 때까지 각 수준에서 계속됨

    This continues at each level until it can be inserted as a new leaf.



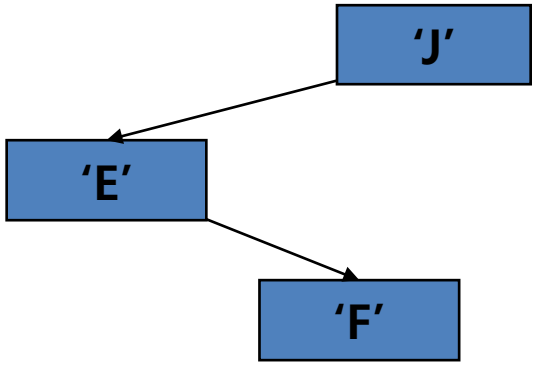
Inserting ‘F’ into the BST

- ‘F’를 루트 노드의 값과 비교하여 더 작으면 왼쪽으로 이동, 더 크면 오른쪽으로 이동하여 시작

  Begin by comparing ‘F’ to the value in the root node, moving left if it is less, or moving right if it is greater.

  - ✓ 리프로 삽입될 때까지 계속 반복

    This continues until it can be inserted as a leaf



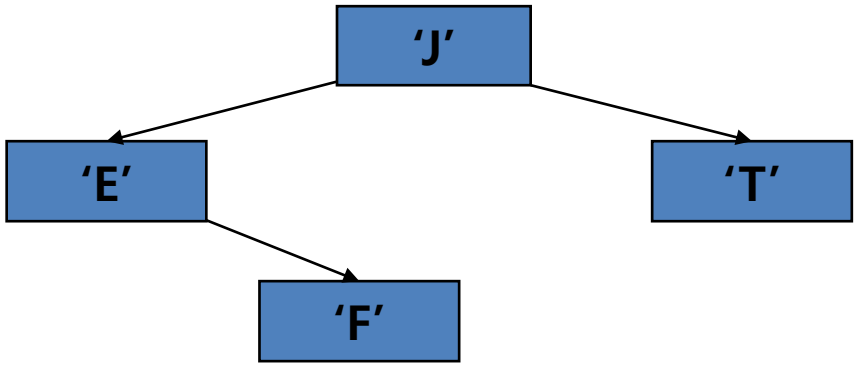
Inserting ‘T’ into the BST

- ‘T’를 루트 노드의 값과 비교하여 작으면 왼쪽, 크면 오른쪽으로 이동하여 시작

  Begin by comparing ‘T’ to the value in the root node, moving left if it is less, or moving right if it is greater.

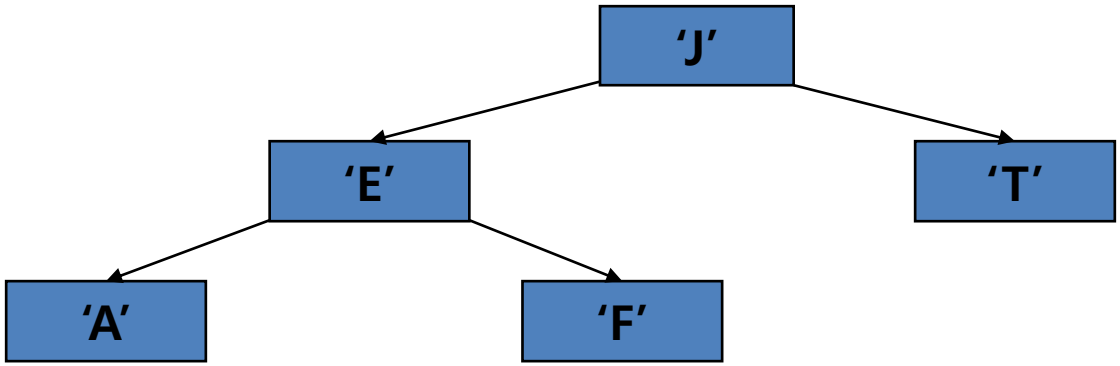
  - ✓ 역시나 리프로 삽입될 때까지 반복

    This continues until it can be inserted as a leaf.



Inserting ‘A’ into the BST

- ‘A’를 루트노드와 비교하여 작으면 왼쪽, 크면 오른쪽으로 이동하여 시작
  Begin by comparing ‘A’ to the value in the root node, moving left if it is less, or moving right if it is greater.
  - ✓ 또 역시나 리프로 삽입될 때까지 반복
    This continues until it can be inserted as a leaf.



Summarization of Insertion

- 재귀 연산
  Recursive operation

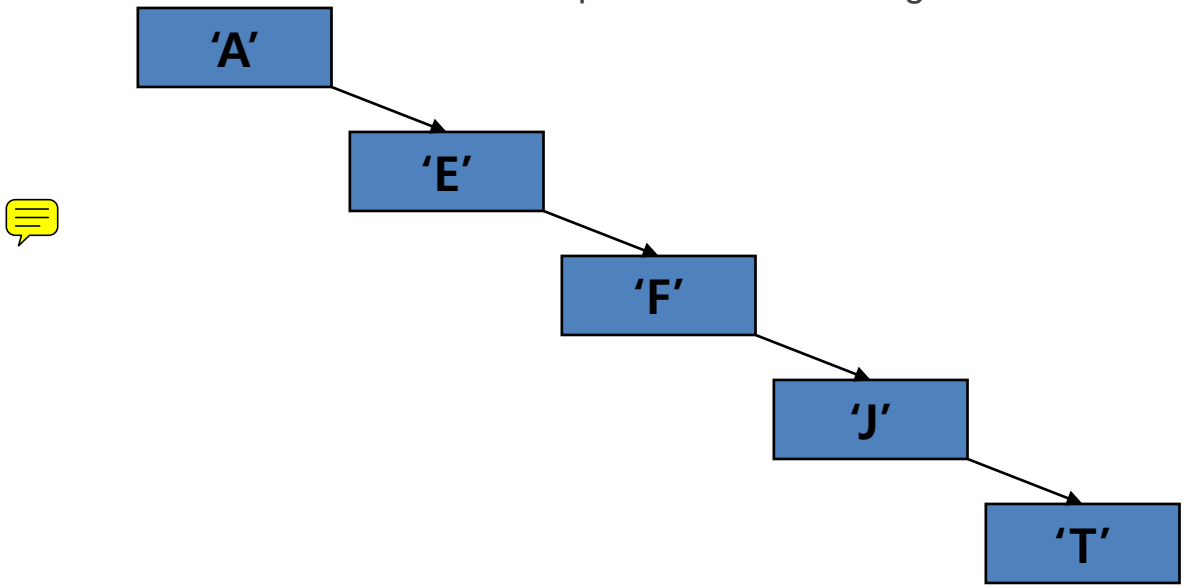
  - ✓ 현재 포인터가 NULL이면 삽입
    Insert item when current pointer is NULL
  - ✓ 키 값이 작으면 right subtree로 이동
    If key value is less, move to right subtree
  - ✓ 키 값이 크면 left subtree로 이동
    If key value is greater, move to left subtree
  - ✓ 키 값이 같으면 메시지 출력 후 종료
    If key value is same, print a message and exit

What binary search tree . . .

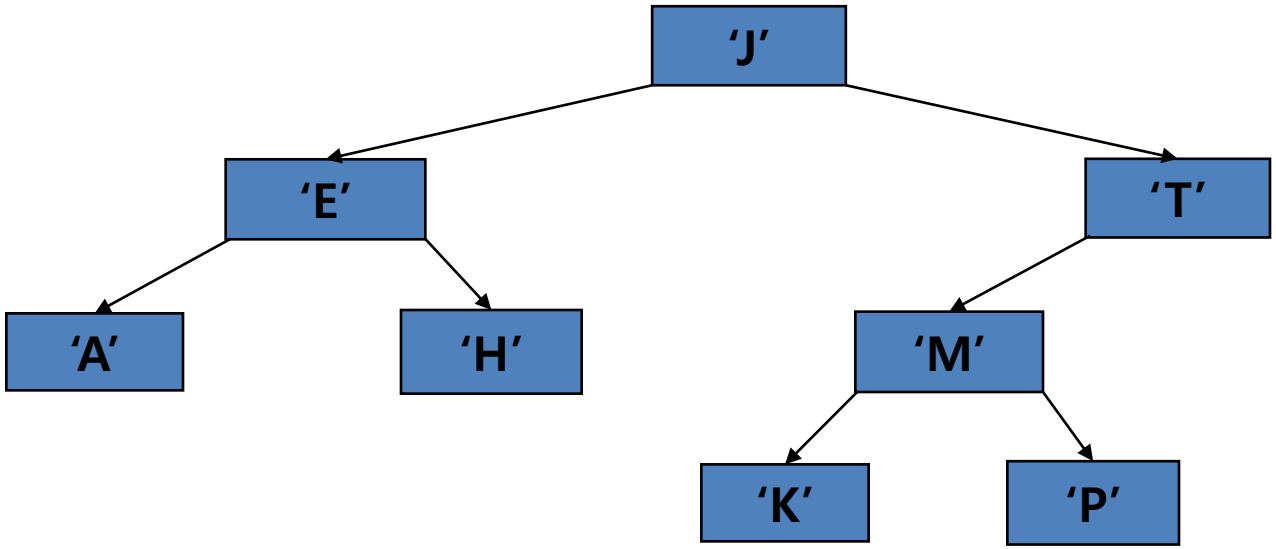
- ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ 요소를 순서대로 삽입하여 얻을 수 있나?
  is obtained by inserting the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?

Binary search tree . . .

- ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ 요소를 순서대로 삽입한 결과
  obtained by inserting the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order.
  - ✓ 논의사항
    Discussion
    - Worst case 시 비교 횟수는?
      What is the number of comparison in the worst case?
    - 탐색을 위해서 평균 비교 수가 최소가 되기 위해서는 어떻게 해야하나?
      How to make the minimum number of comparison for searching?



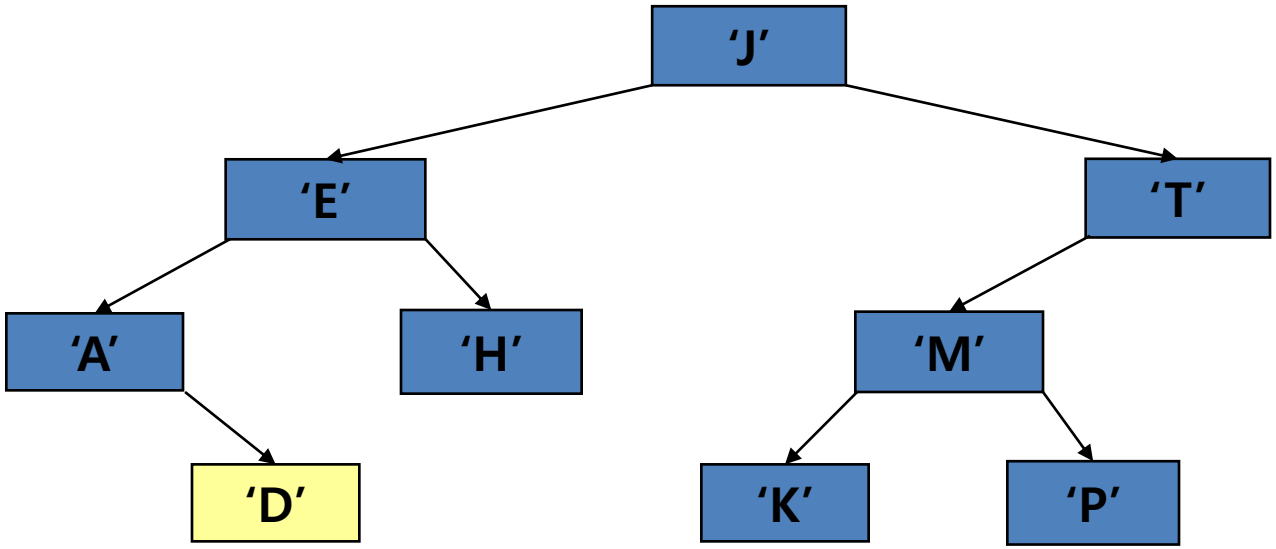
Another binary search tree



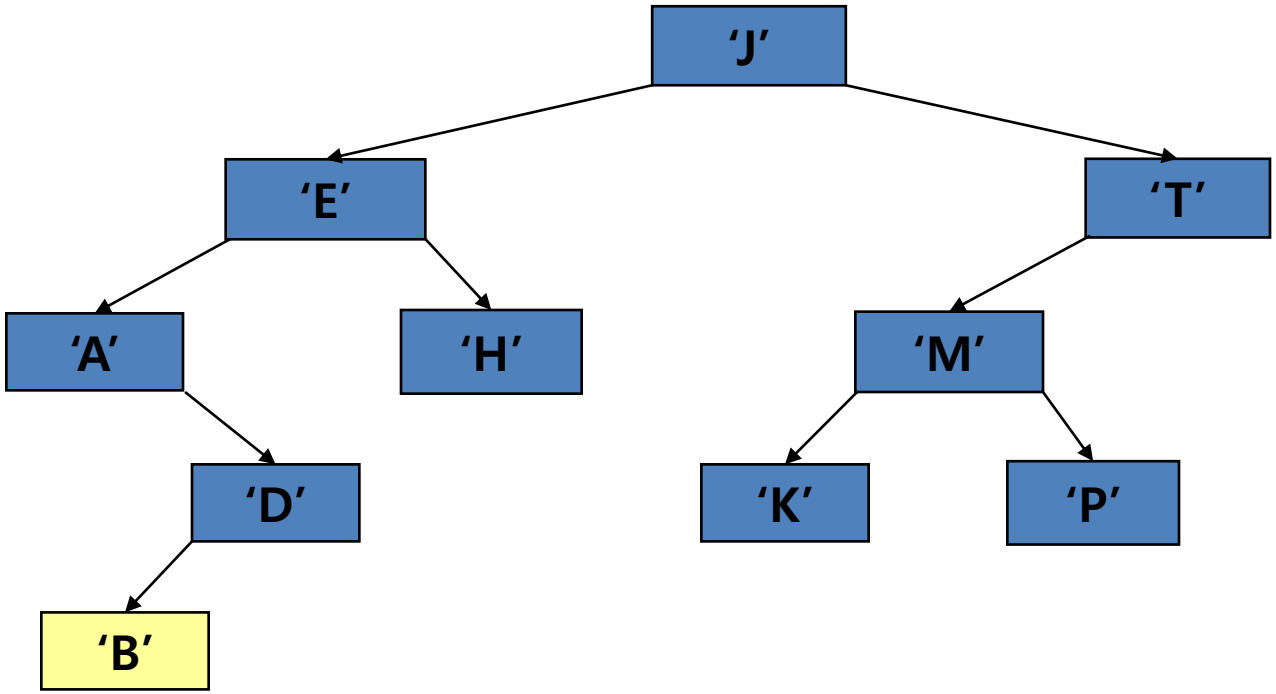
Add nodes containing these values in this order:

'D' 'B' 'L' 'Q' 'S' 'V' 'Z'

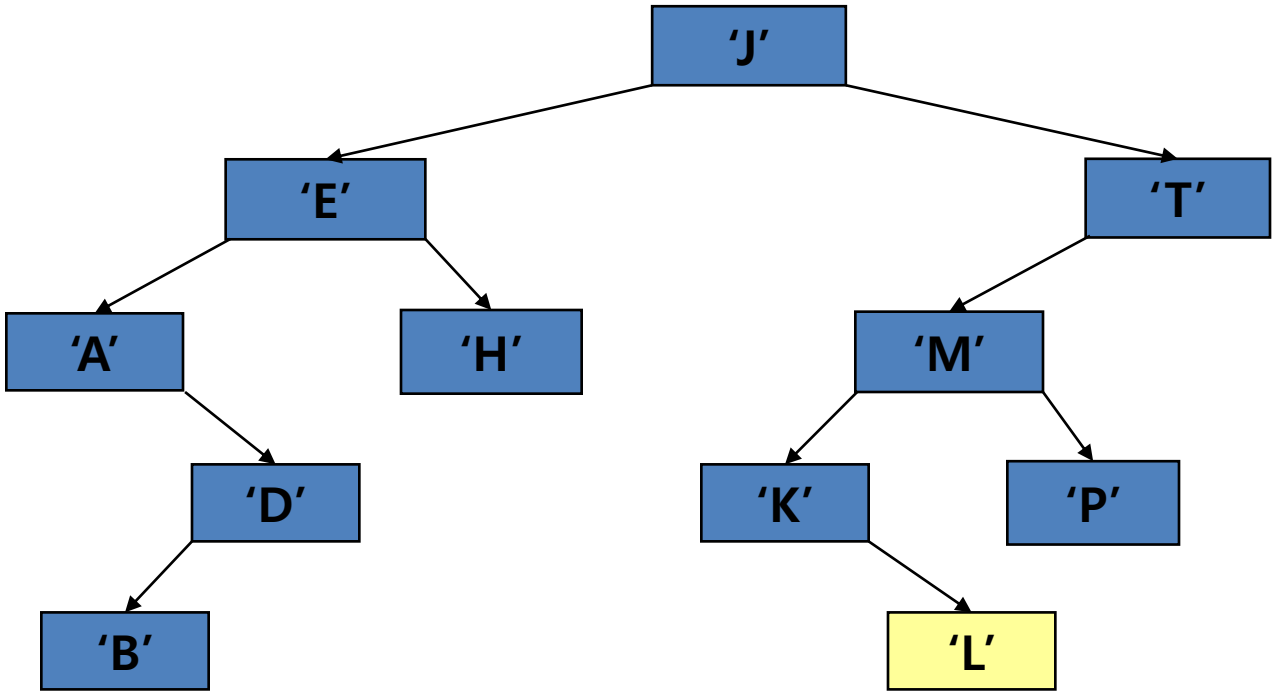
Insert ‘D’



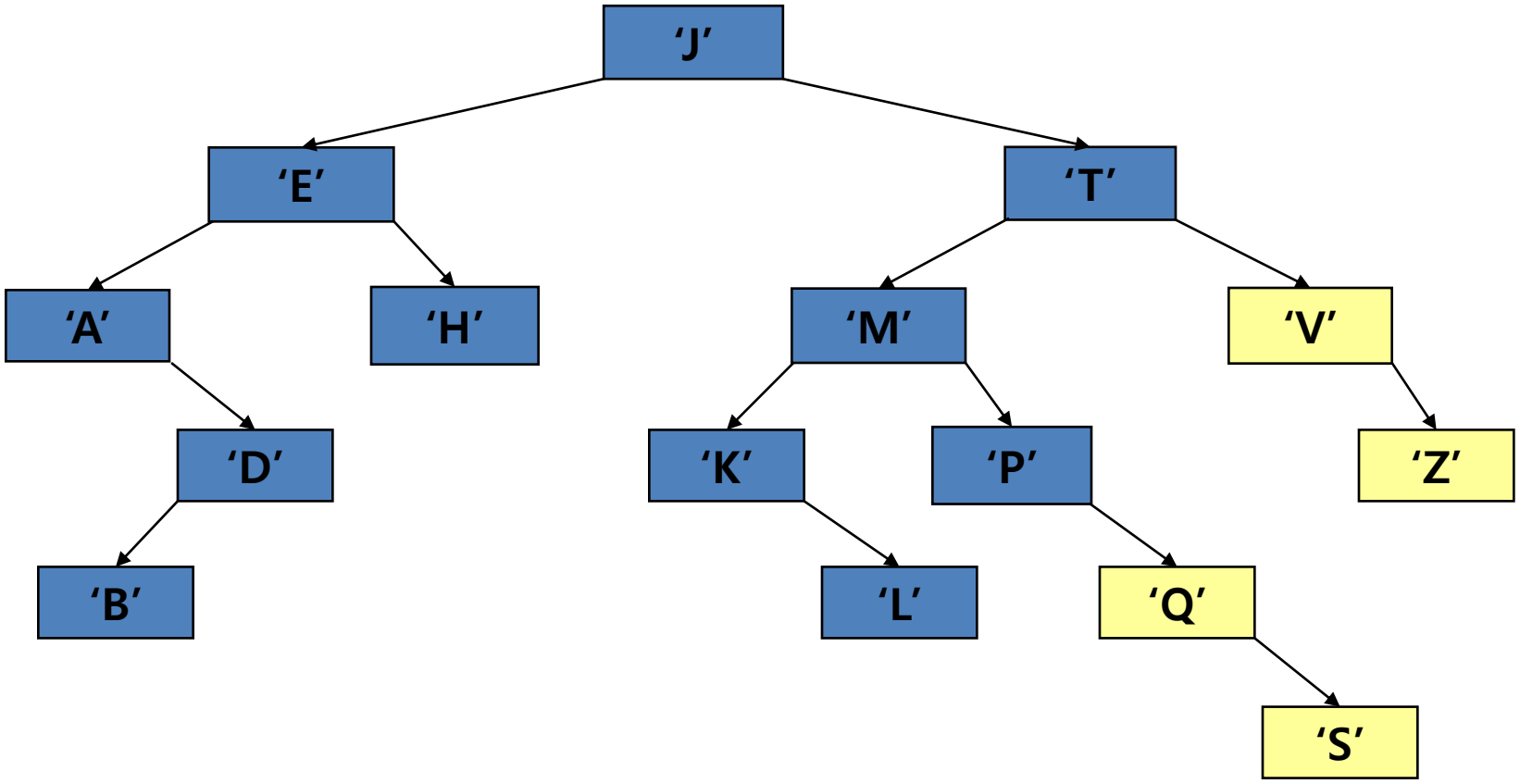
Insert ‘B’



Insert ‘L’



Insert ‘Q’, ‘S’, ‘V’, ‘Z’



Implementation of TreeType

```
template< class  ItemType >
class TreeType
{
public:
      TreeType();                              // constructor
      ~TreeType();                             // destructor
      TreeType(const TreeType& originalTree);        // copy constructor
      void operator =(const TreeType& originalTree);
      void MakeEmpty();
      bool IsEmpty() const;
      bool IsFull() const;
      void ResetTree(OrderType order);
      int LengthIs() const;
      void RetrieveItem(ItemType& item, bool& found) const;
```

Implementation of TreeType

```
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void GetNextItem (ItemType& item, OrderType order, bool& finished);
    void Print(std::ofstream& outFile) const;
private:
    TreeNode<ItemType>* root;
};

template< class  ItemType >
struct TreeNode
{
    ItemType info;
    TreeNode* left;
    TreeNode* right;
};
```

Implementation of TreeType

```
template< class  ItemType >
bool TreeType<ItemType> :: IsFull() const
// Returns true if there is no room for another item
//  on the free store; false otherwise.
{
        TreeNode* location;
        try
        {
                location = new TreeNode;
                delete location;
                return false;
        }
        catch(std::bad_alloc exception)
        {
                return true;
        }
}

template< class  ItemType >
bool TreeType<ItemType> :: IsEmpty() const
// Returns true if the tree is empty; false otherwise.
{
        return root == NULL;
}
```

The Function LengthIs

```
version1
if( Left(tree) == NULL ) and ( Right(tree) == NULL)
        return 1
else
        return CountNodes ( Left(tree) ) +
CountNodes(Right(tree)) + 1
```

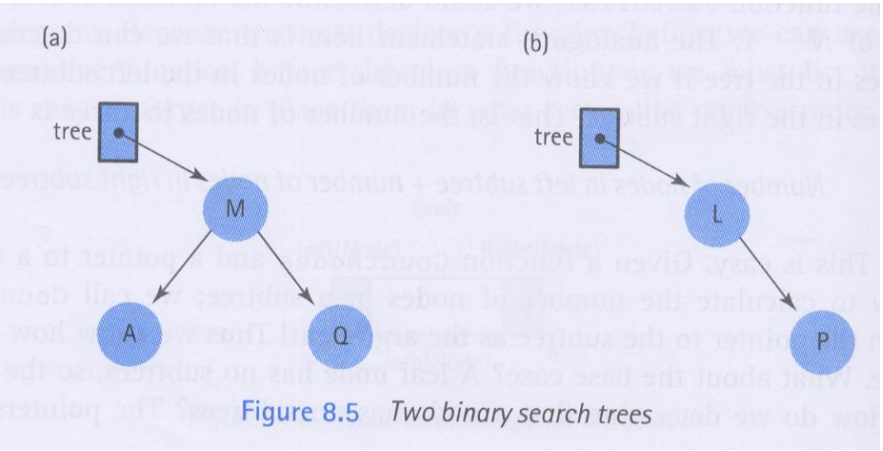


Figure 8.5 *Two binary search trees*

Definition: Count the Number of nodes in tree

Size: Number of nodes in tree

Base Case: If tree is NULL, return 0

General Case: Return
CountNodes(Left(tree))+
CountNodes(Right(tree))+1

CountNodes Version 2

```
if(Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1
```

CountNodes Version 3

```
if tree is NULL
    return 0
if(Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1
```

CountNodes Version 4

```
if tree is NULL
    return 0
else
    return CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1
```

Function LengthIs()

```
template< class  ItemType > int CountNodes(TreeNode<ItemType>* tree);

template< class  ItemType >
int TreeType<ItemType> :: LengthIs() const
// Calls recursive function CountNodes to count the
// nodes in the tree.
{
    return CountNodes(root);
}

template< class  ItemType >
int CountNodes(TreeNode<ItemType>* tree)
// Post: returns the number of nodes in the tree.
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) + CountNodes(tree->right) + 1;
}
```

Retrieve Operation

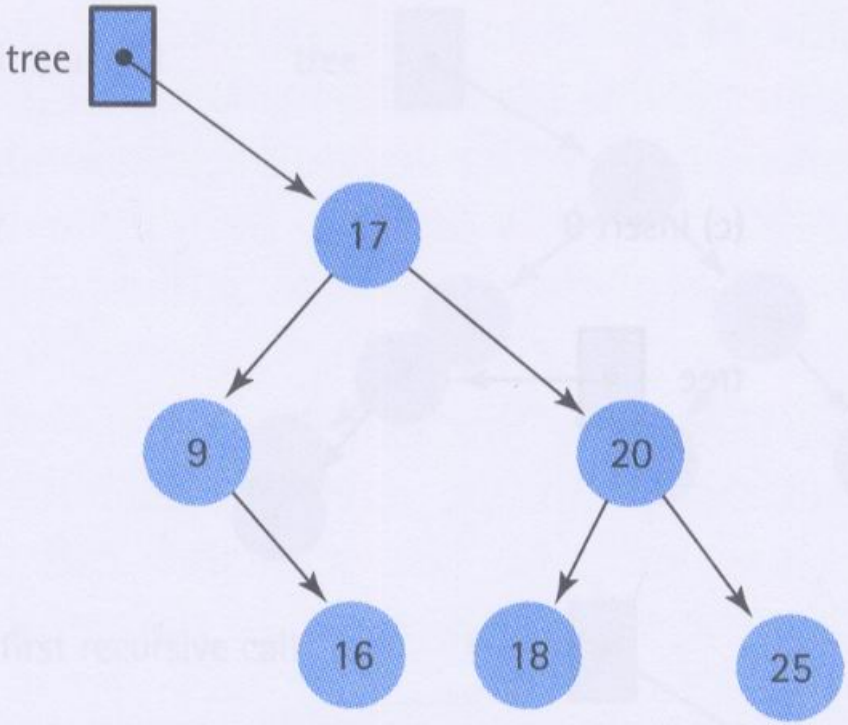


Figure 8.6 *Tracing the Retrieve operation*

-Retrieve 18

-Retrieve 21

Function Retrieve(tree, item, found)

Base Case:

-If item's key matches key in Info(tree), item is set to Info(tree) and found is true

-If tree==NULL, found is false

General Case:

If item's key is less than key in Info(tree), Retrieve(Left(tree), item, found);

Else Retrieve(Right(tree),item, found)

Function Retrieve()

```
template< class  ItemType >
void TreeType<ItemType> :: RetrieveItem ( ItemType&  item, bool&   found )
{
        Retrieve ( root, item,  found ) ;
}

template< class  ItemType >
void Retrieve ( TreeNode<ItemType>* ptr, ItemType&  item, bool&   found)
{
        if  ( ptr == NULL )
                found = false ;
        else if  ( item  <  ptr->info )                          // GO LEFT
                Retrieve( ptr->left , item,  found ) ;
        else if  ( item  >  ptr->info )                          // GO RIGHT
                Retrieve( ptr->right , item,  found ) ;
        else
        {
                item = ptr->info ;
                found = true ;
      }
}
```

The Function InsertItem

Insert 13

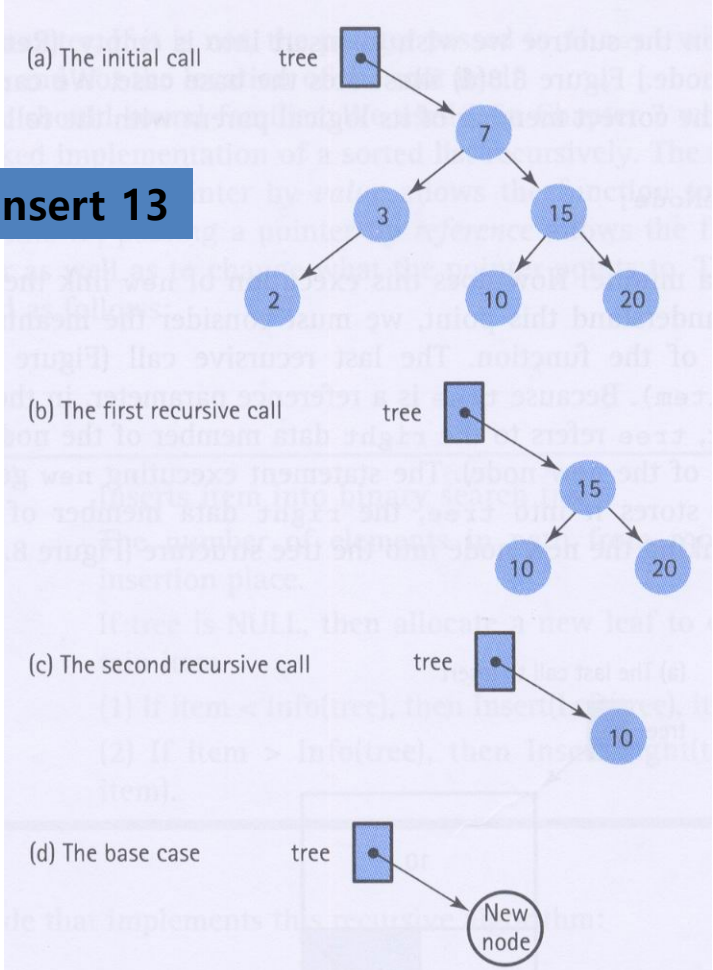


Figure 8.8 *The recursive Insert operation*

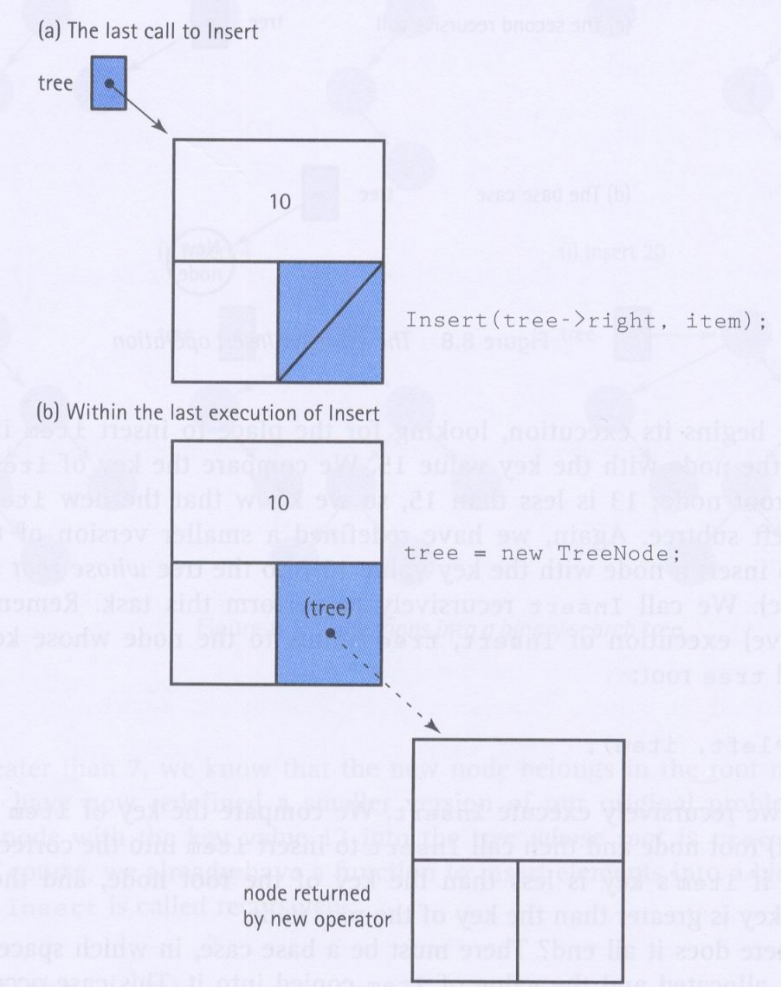


Figure 8.9 *The tree parameter is a pointer within the tree*

Function Insert()

```
template< class  ItemType >
void  TreeType<ItemType> :: InsertItem ( ItemType  item )
{
      Insert ( root, item ) ;
}

template< class  ItemType >
void   Insert ( TreeNode<ItemType>*&  ptr,   ItemType  item )
{
      if  ( ptr == NULL )
      {                                                  // INSERT item HERE AS LEAF
            ptr = new TreeNode<ItemType> ;
            ptr->right = NULL ;
            ptr->left = NULL ;
            ptr->info = item ;
      }
      else if  ( item  <  ptr->info )                    // GO LEFT
            Insert( ptr->left , item ) ;
      else if  ( item  >  ptr->info )                    // GO RIGHT
            Insert( ptr->right , item ) ;
}
```

The Function DeleteItem

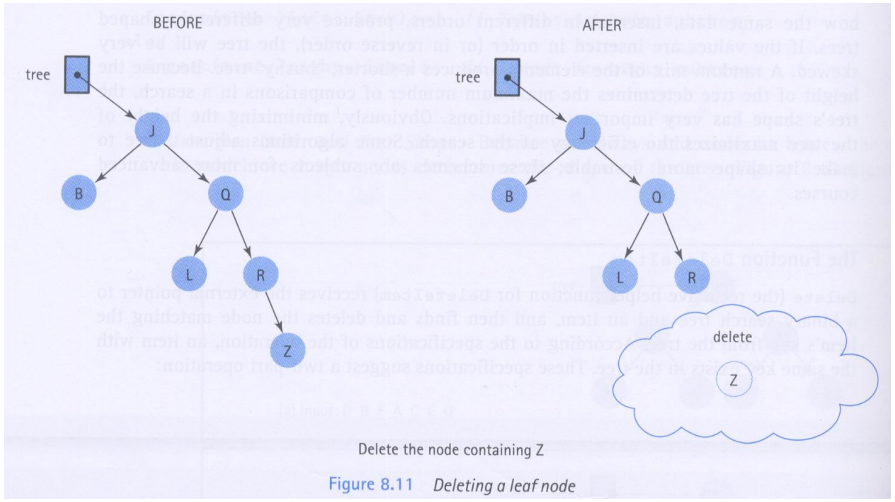


Figure 8.11 *Deleting a leaf node*

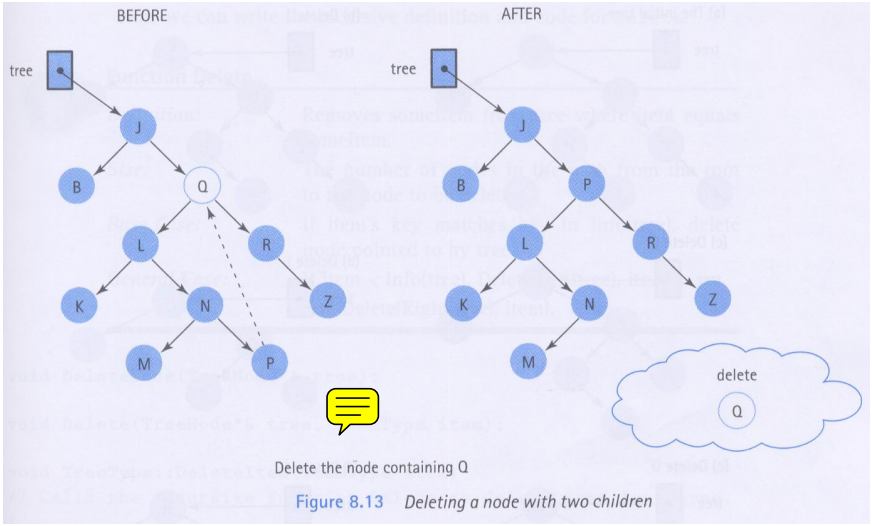


Figure 8.13 *Deleting a node with two children*

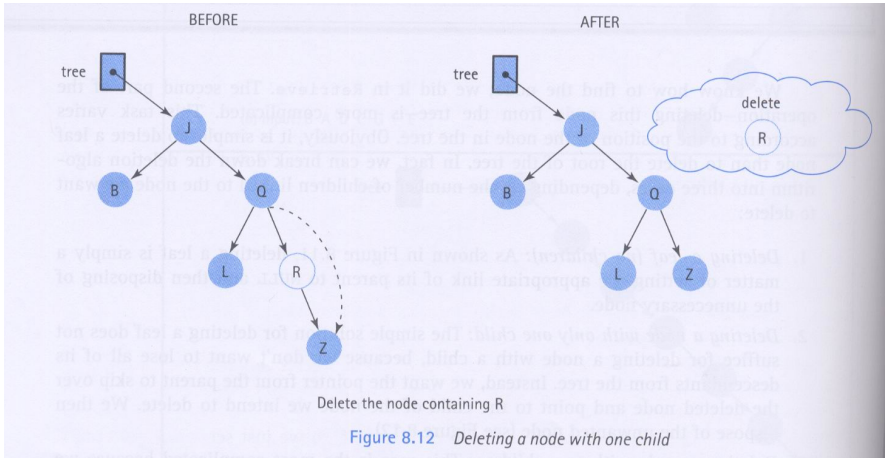


Figure 8.12 *Deleting a node with one child*

Base Case:

If item's key matches key in Info(tree), delete node

General Case:

If item<Info(tree), Delete(Left(tree),item);

Else Delete(Right(tree),item);

Function Delete()

```
template< class  ItemType > void Delete(TreeNode*& tree, ItemType item);

template< class  ItemType >
void TreeType<ItemType>::DeleteItem(ItemType item)
// Calls the recursive function Delete to delete item from tree.
{
        Delete(root, item);
}

template< class  ItemType >
void Delete(TreeNode*& tree, ItemType item)
// Deletes item from tree
// Post : item is not in tree
{
        if (item < tree->info)
                Delete(tree->left, item);               // Look in left subtree.
        else if (item > tree ->info)
                Delete(tree->right, item);              // Look in right subtree.
        else
                DeleteNode(tree);                       // Node found; call DeleteNode.
}
```

Function Delete()

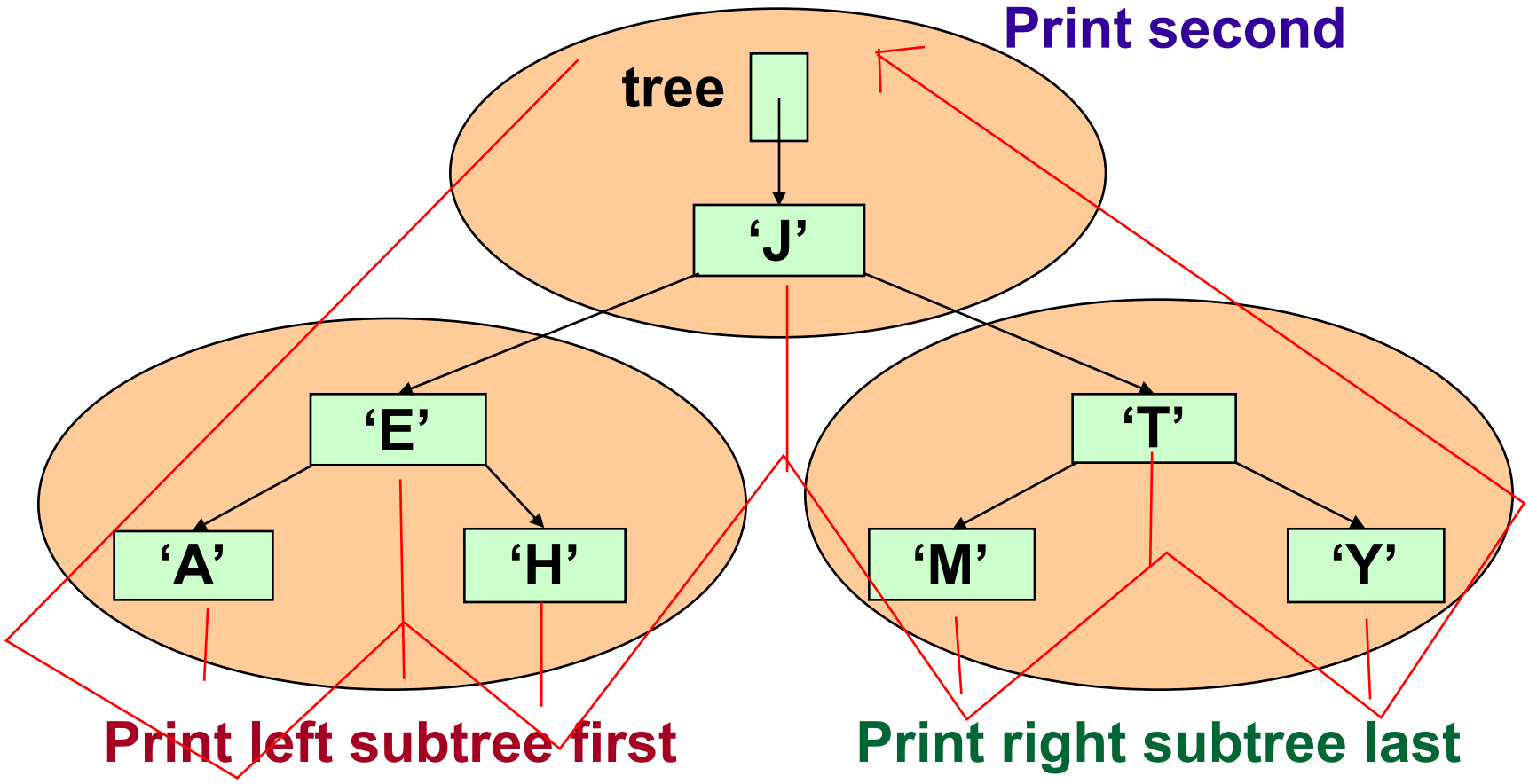
```
template< class  ItemType > void GetPredecessor(TreeNode* tree, ItemType& data);

template< class  ItemType >
void DeleteNode(TreeNode*& tree)
{
        ItemType data;
        TreeNode* tempPtr;
        tempPtr = tree;
        if(tree->left == NULL)
        {
                tree = tree->right;
                delete tempPtr;
        }
        else if (tree->right == NULL)
        {
                tree = tree->left;
                delete tempPtr;
        }
        else
        {       GetPredecessor(tree->left, data);
                tree->info = data;
                Delete(tree->left, data);                    //Delete predecessor node.
        }
}
```

Function Delete()

```
template< class  ItemType >
void GetPredecessor (TreeNode* tree, ItemType& data)
// Sets data to the info member of the rightmost
//  node in tree.
{
      while(tree->right != NULL)
            tree = tree->right;
      data = tree->info;
}
```

Inorder Traversal: A E H J M T Y

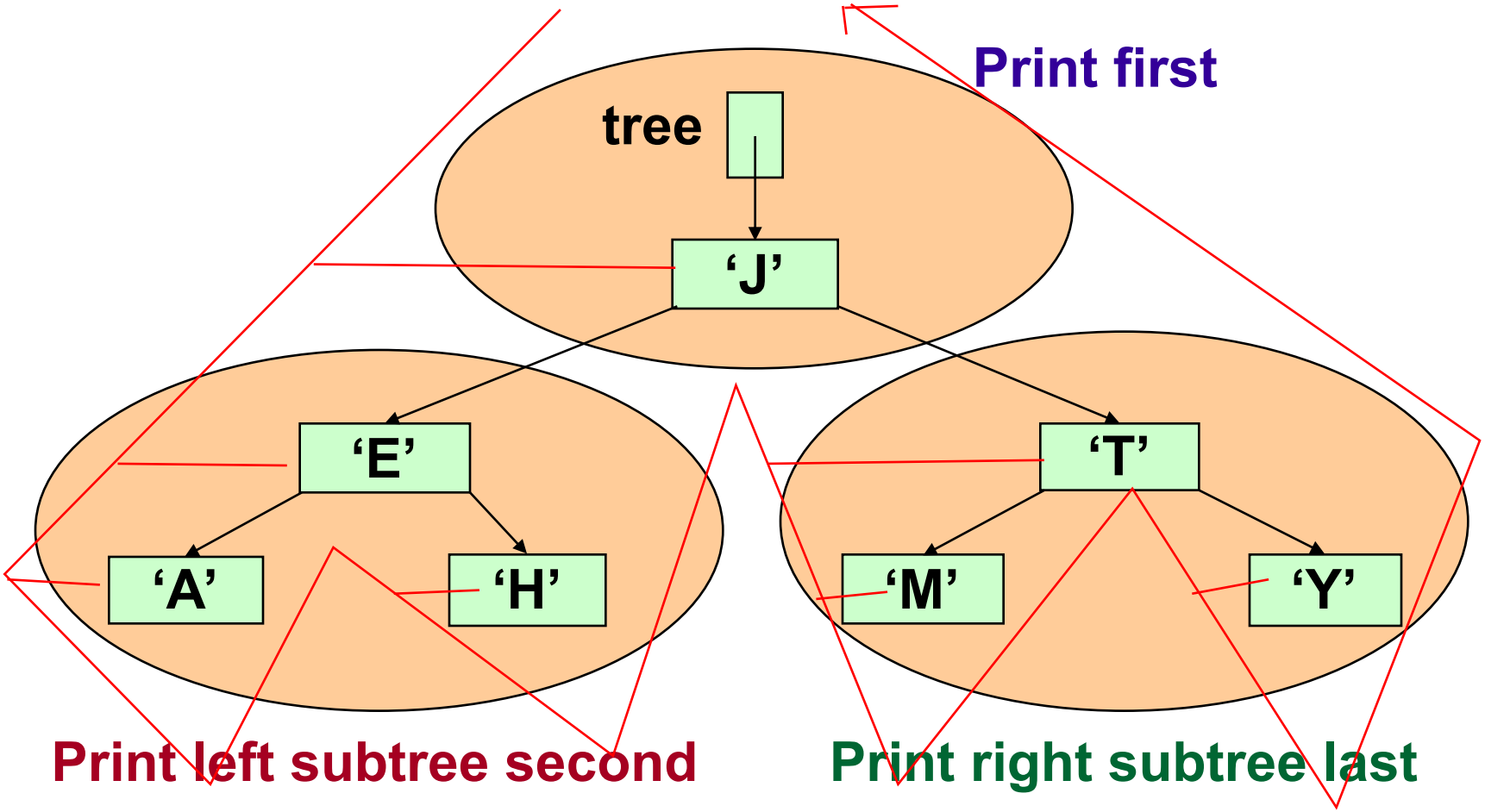


Inorder Traversal

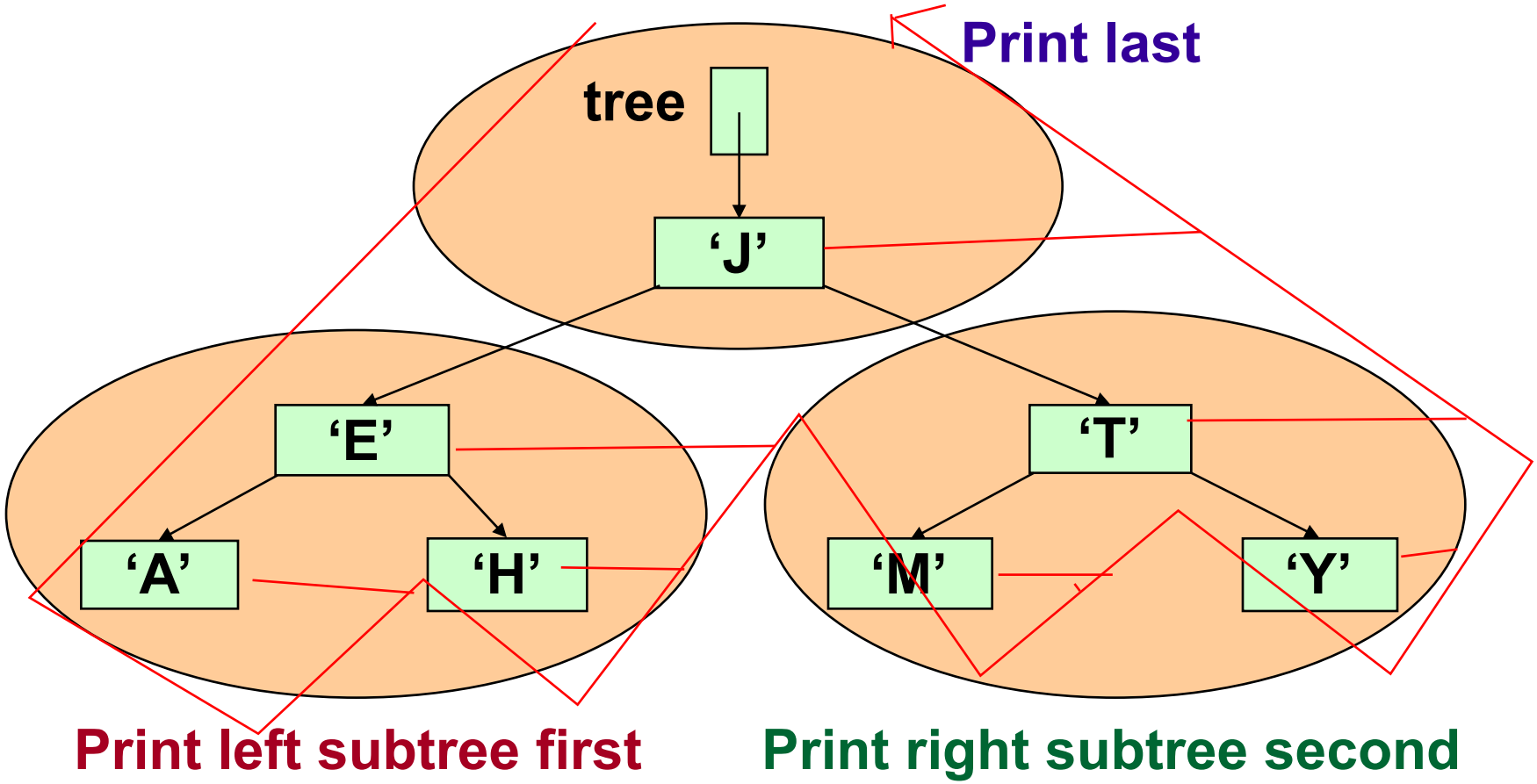
```
                                        // INORDER TRAVERSAL
template< class  ItemType >
void   TreeType<ItemType> :: PrintTree ( ofstream&  outFile ) const
{
      Print ( root,  outFile ) ;
}

template< class  ItemType >
void  Print( TreeNode<ItemType>* ptr,  std::ofstream&  outFile )
{
      if  ( ptr != NULL )
      {
            Print( ptr->left , outFile ) ;              // Print left subtree
            outFile  <<  ptr->info ;
            Print( ptr->right, outFile ) ;    // Print right subtree
      }
}
```

Preorder Traversal: J E A H T M Y



Postorder Traversal: A H E M Y T J



Other TreeType codes

```
template< class  ItemType >
TreeType<ItemType> :: ~TreeType (  )              // DESTRUCTOR
{
      Destroy ( root ) ;
}

template< class  ItemType >
void   Destroy ( TreeNode<ItemType>* ptr )
// Post:  All nodes of the tree pointed to by ptr are deallocated.
{
      if  ( ptr != NULL )
      {
            Destroy ( ptr->left ) ;
            Destroy ( ptr->right ) ;
            delete  ptr ;
      }
}
```

Thank You!
Q&A