Binary Search Trees
（二叉查找树、二叉排序树）

- DEFINITION:(a recursive definition)
  - A *binary search tree* is a binary tree that is either empty or in which every node has a key（关键字）and satisfies the following conditions:
    - The key of the root is greater than the key in any node in the left subtree of the root.
    - The key of the root is less than the key in any node in the right subtree of the root..
    - The left and right subtrees of the root are again binary search trees.

二叉查找树（Binary Search Tree）

- <u>定义</u>

　　二叉查找树或者是一棵空树，或者是具有下列性质的二叉树：

✓ 每个结点都有一个作为查找依据的关键码(key)，所有结点的关键码互不相同。

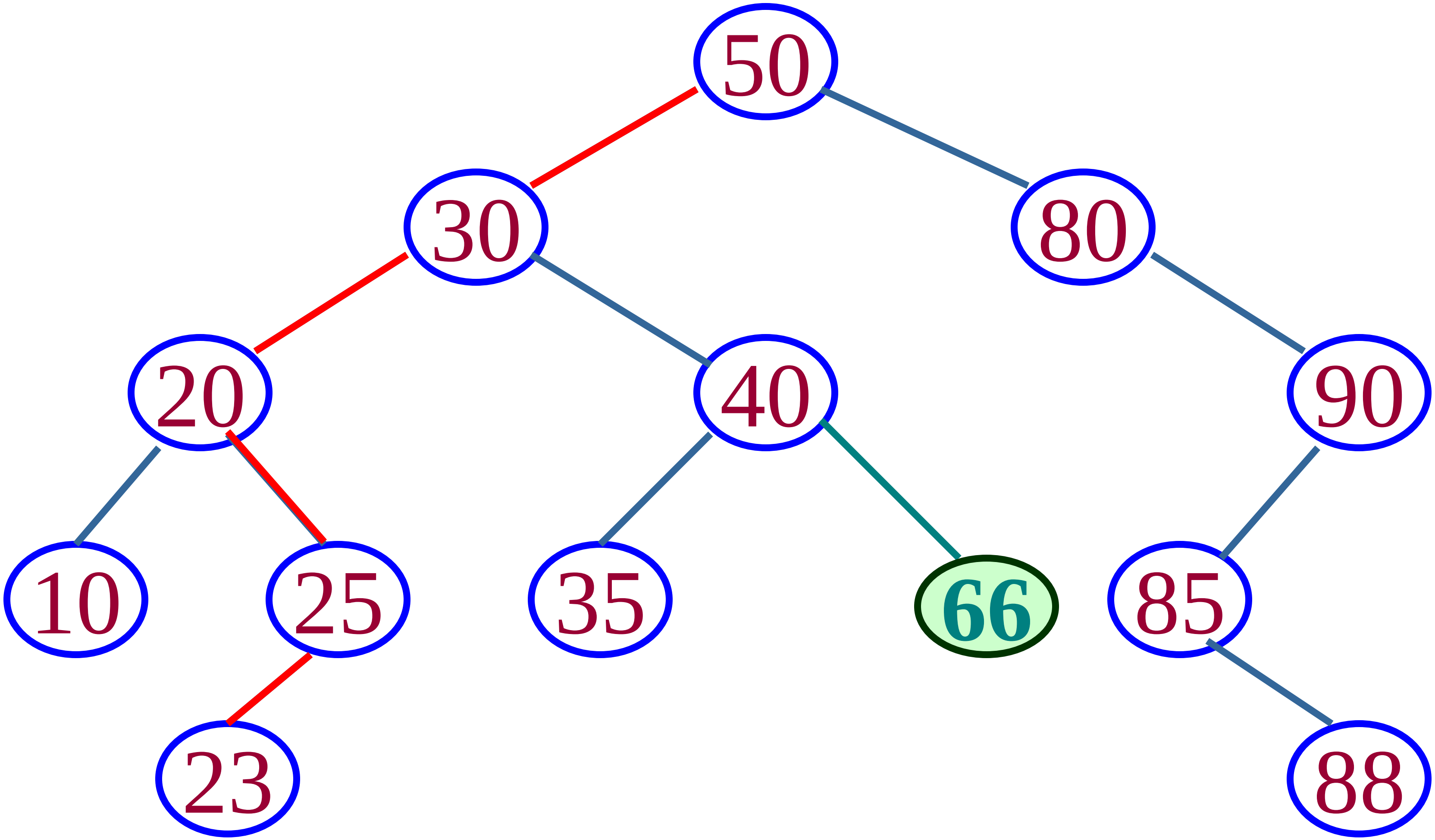
✓ 左子树（如果非空）上所有结点的关键码都小于根结点的关键码。

✓ 右子树（如果非空）上所有结点的关键码都大于根结点的关键码。

✓ 左子树和右子树也是二叉查找树。

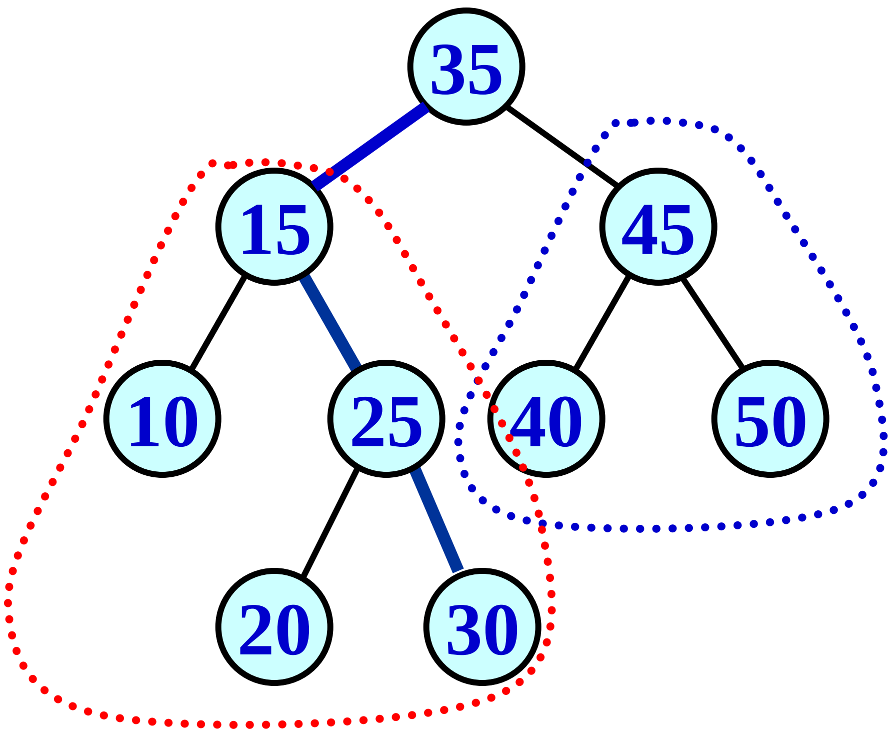
Binary Search Trees

- DEFINITION:
  - No two entries in a binary search tree may have equal keys.(没有两个记录的关键字是一样的）
  - We can regard binary search trees as a new ADT.
  - We may regard binary search trees as a specialization of binary trees.（二叉查找树是二叉树的一个特例）
  - We may study binary search trees as a new implementation of the ADT *ordered list*.（可以把二叉查找树作为有序表的一种实现方法。）



不是二叉查找树。

例如：


35
15
45
10
25
40
50
20
30

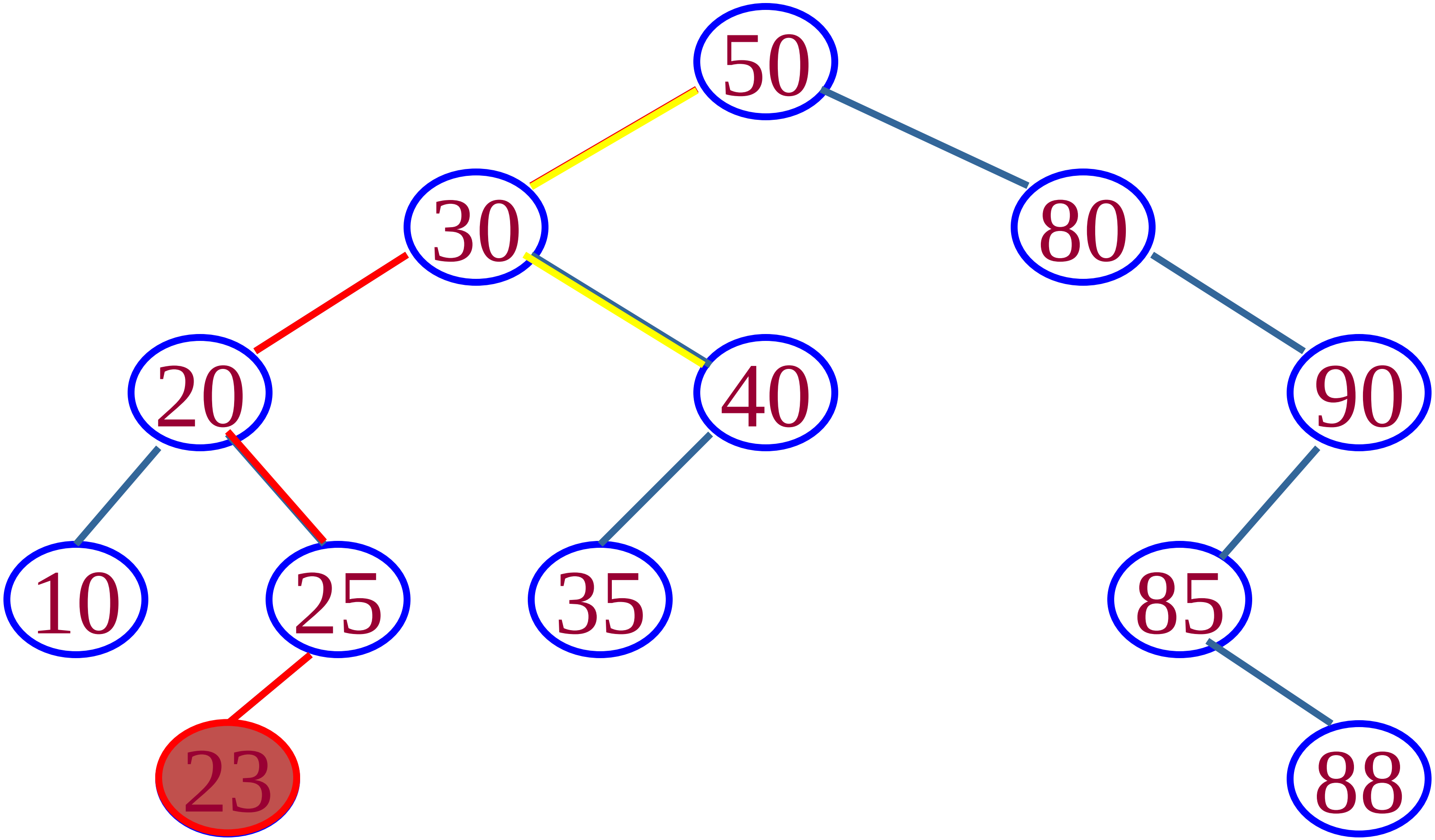
Binary Search Trees

- implementations:
  - **The Binary Search Tree Class（二叉查找树类）**
    - The binary search tree class will be *derived* from the binary tree class; hence all binary tree methods are inherited（继承）.

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
    Error_code insert(const Record &new_data);
    Error_code remove(const Record &old_data);
    Error_code tree_search(Record &target) const;
private: // Add auxiliary function prototypes here.
……
};
```

Binary Search Trees

- implementations:
  - The inherited methods include the constructors, the destructor, clear, empty, size, height, and the traversals preorder, inorder, and postorder.
  - A binary search tree also admits specialized methods called insert, remove(删除）, and tree search（查找）.
  - The class Record has the behavior outlined in Chapter 7: Each Record is associated with a Key. The keys can be compared with the usual comparison operators(比较运算符）. By casting records to their corresponding keys, the comparison operators apply to records as well as to keys.



查找23：45

Binary Search Trees

- **Tree Search**
- **Public method for tree search:**

Error_code Search_tree<Record> **::**Tree_search(Record &target) **const;**

/**Post*: If there is an entry in the tree whose key matches(符合） that in target, the parameter（参数） target is replaced by the corresponding record from the tree and a code of success is returned. Otherwise a code of not_present is returned.*/

- This method will often be called with a parameter target that contains only a key value. The method will fill target with the complete data belonging to any corresponding Record in the tree.

Binary Search Trees

- implementations:
  - **Public method for tree search:**

```
template <class Record>
Error_code Search_tree<Record> ::tree_search(Record &target) const
{
      Error_code result = success;
      Binary_node<Record> *found = search_for_node(root,
         target);
      if (found == NULL)
               result = not_present;
      else
               target = found->data;
      return result;
}
```

Binary Search Trees

- The auxiliary search function
  - We first compare it with the entry at the root of the tree. If their keys match, then we are finished.
  - Otherwise, we go to the left subtree or right subtree as appropriate and repeat the search in that subtree.
  - The process terminates(终止) when it either finds the target or hits an empty subtree.
  - returns a pointer to the node that contains the target back to the calling program.

Binary Search Trees

- implementations

```
Binary_node<Record> *Search_tree<Record> :: search_for_node(
Binary_node<Record>* sub_root, const Record &target) const;
/*Pre: sub_root is NULL or points to a subtree of a Search_tree
Post: If the key of target is not in the subtree, a result of NULL is
   returned. Otherwise, a pointer to the subtree node containing the
   target is returned.*/
```

Recursive function

```
template <class Record>
Binary_node<Record> *Search_ tree<Record> ::
search_for_node(
Binary_node<Record>* sub_root, const Record &target) const
{
if (sub_root == NULL || sub_root->data == target)
     return sub_root;
else if (sub_root->data < target)
     return search_for_ node(sub_ root->right, target);
      else return search_ for_ node(sub_ root->left, target);
}
```

Nonrecursive version:

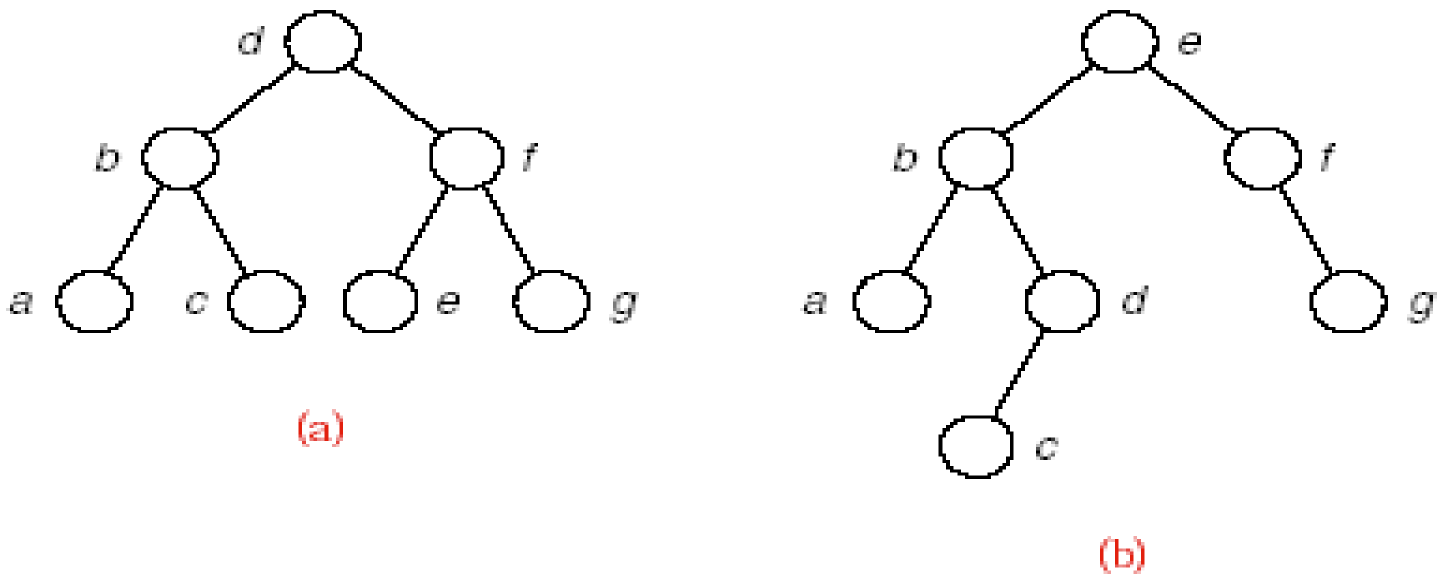
```
template <class Record>
Binary_ node<Record> *Search_ tree<Record> :: search_for_ node(
Binary_ node<Record> *sub_ root, const Record &target) const
{
while (sub_ root != NULL && sub_root->data != target)
        if (sub_ root->data < target)
                sub_ root = sub_ root->right;
        else
                sub_ root = sub_ root->left;
return sub_ root;
}
```

Binary Search Trees

- implementations:
  - **Analysis of Tree Search**
    - Draw the comparison tree(比较树） for a binary search (on an ordered list). Binary search on the list does exactly the same comparisons as tree search will do if it is applied to the comparison tree.
    - By Section 7.4, binary search performs O（log n） comparisons for a list of length n. This performance（性能） is excellent in comparison to other methods, since log$n$ grows very slowly as n increases.

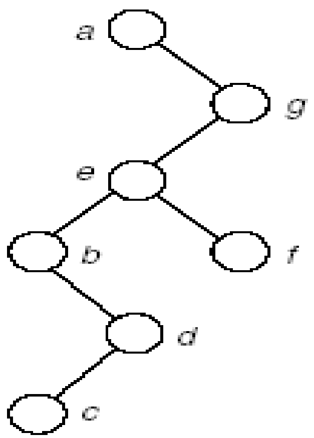
Binary Search Trees

- **Binary Search Trees with the Same Keys(具有同一组关键字的二叉查找树）**

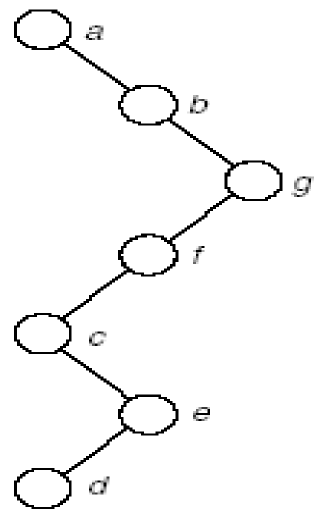


Binary Search Trees

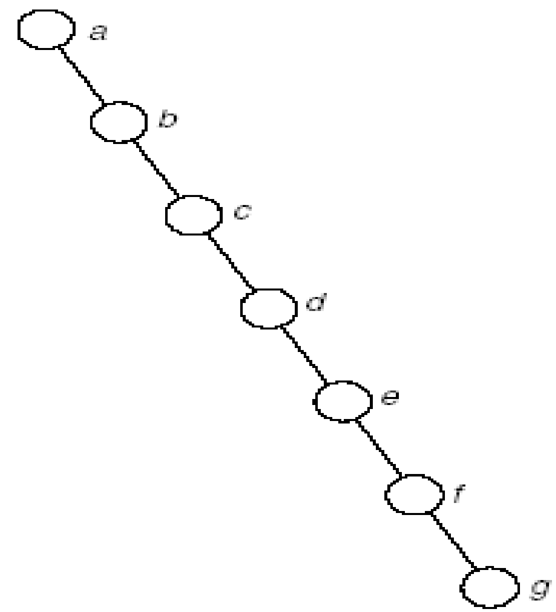
- **Binary Search Trees with the Same Keys(具有同一组关键字的二叉查找树）**



(c)



(d)



(e)

- **Analysis of Tree Search**
  - The same keys may be built into binary search trees of many different shapes.
  - If a binary search tree is nearly completely balanced(平衡的）(“bushy”), then tree search on a tree with *n* vertices（顶点）will also do *O*（log *n*） comparisons of keys.
  - If the tree degenerates into a long chain(链）, then tree search becomes the same as sequential search（顺序查找）, doing O(*n*) comparisons on *n* vertices. This is the worst case for tree search.
  - The number of vertices between the root and the target, inclusive（包括它们）,is the number of comparisons that must be done to find the target. The bushier the tree, the smaller the number of comparisons that will usually need to be done.

Binary Search Trees

- **Analysis of Tree Search**
  - It is often not possible to predict (预测） (in advance of building it) what shape of binary search tree will occur.
  - In practice, if the keys are built into a binary search tree in random order（以随机次序）, then it is extremely unlikely that a binary search tree degenerates badly; tree_search usually performs almost as well as binary search.

Binary Search Trees

- **Insertion into a Binary Search Tree**

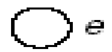
Error_code Search_tree<Record> ::insert(**const** Record &new_data)**;**

/**Post*: If a Record with a key matching that of new data already belongs to the Search_tree ,a code of duplicate(重复的） error is returned.

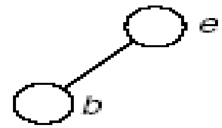
Otherwise, the Record new_data is inserted into the tree in such a way that the properties(性质） of a binary search tree are preserved, and a code of success is returned.*/

Binary Search Trees

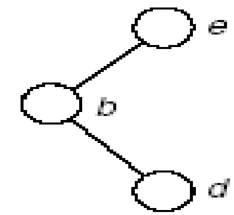
- implementations:
  - **Insertion into a Binary Search Tree**



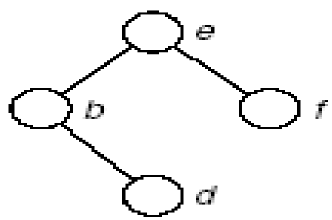
(a) Insert *e*



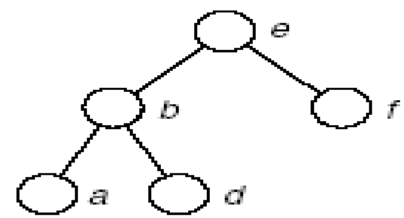
(b) Insert *b*



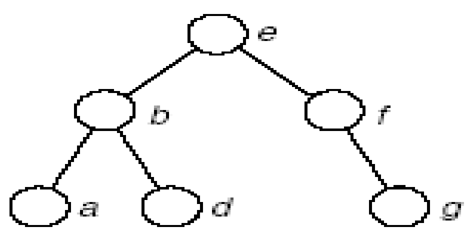
(c) Insert *d*



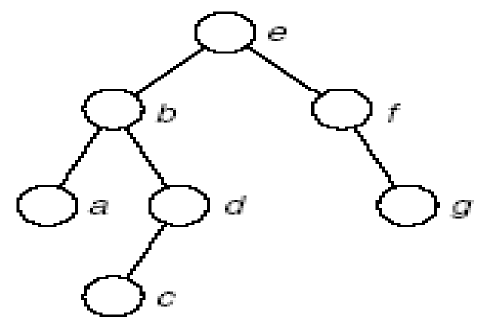
(d) Insert *f*



(e) Insert *a*



(f) Insert *g*



(g) Insert *c*