E5. *Write a method and the corresponding recursive function to count the leaves (i.e., the nodes with both subtrees empty) of a linked binary tree.*

Answer

```
template <class Entry>
int Binary_tree<Entry> :: recursive_leaf_count
                (Binary_node<Entry> *sub_root) const
/* Post: The number of leaves in the subtree rooted at sub_root is returned. */
{
  if (sub_root == NULL) return 0;
  if (sub_root->left == NULL && sub_root->right == NULL) return 1;
  return recursive_leaf_count(sub_root->left)
          + recursive_leaf_count(sub_root->right);
}
template <class Entry>
int Binary_tree<Entry> :: leaf_count() const
/* Post: The number of leaves in the tree is returned. */
{
  return recursive_leaf_count(root);
}
```

E6. *Write a method and the corresponding recursive function to find the height of a linked binary tree, where an empty tree is considered to have height 0 and a tree with only one node has height 1.*

Answer

```
template <class Entry>
int Binary_tree<Entry> :: height() const
/* Post: The height of the binary tree is returned. */
{
  return recursive_height(root);
}

template <class Entry>
int Binary_tree<Entry> :: recursive_height(Binary_node<Entry> *sub_root) const
/* Post: The height of the subtree rooted at sub_root is returned. */
{
  if (sub_root == NULL) return 0;
  int l = recursive_height(sub_root->left);
  int r = recursive_height(sub_root->right);
  if (l > r) return 1 + l;
  else return 1 + r;
}
```

E7. *Write a method and the corresponding recursive function to insert an* Entry*, passed as a parameter, into a linked binary tree. If the* root *is empty, the new entry should be inserted into the root, otherwise it should be inserted into the shorter of the two subtrees of the root (or into the left subtree if both subtrees have the same height).*

Binary_tree insert

Answer

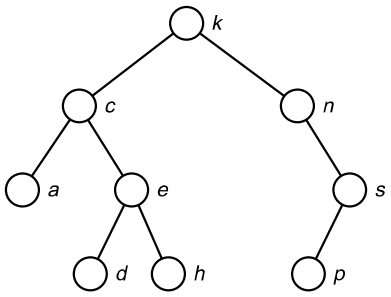
```
template <class Entry>
void Binary_tree<Entry> :: insert(const Entry &x)
/* Post: The Entry x is added to the binary tree. */
{
  recursive_insert(root, x);
}

template <class Entry>
void Binary_tree<Entry> :: recursive_insert(Binary_node<Entry> * &sub_root,
                                            const Entry &x)
/* Pre:  sub_root is either NULL or points to a subtree of the Binary_tree.
   Post: The Entry x has been inserted into the subtree in such a way that the properties of a
         binary search tree have been preserved.
   Uses: The functions recursive_insert, recursive_height */
```

10.2 BINARY SEARCH TREES

Exercises 10.2

The first several exercises are based on the following binary search tree. Answer each part of each exercise independently, using the original tree as the basis for each part.



E1. *Show the keys with which each of the following targets will be compared in a search of the preceding binary search tree.*

(a) *c*

Answer *k, c*

(b) *s*

Answer *k, n, s*

(c) *k*

Answer *k*

(d) *a*

k, c, a

(e) *d*

k, c, e, d

(f) *m*

k, n (fails)

(g) *f*

k, c, e, h (fails)

(h) *b*

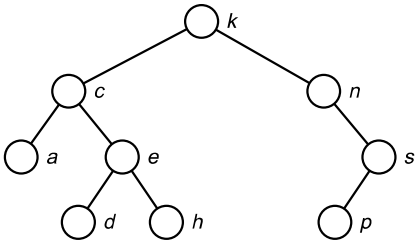
k, c, a (fails)

(i) *t*

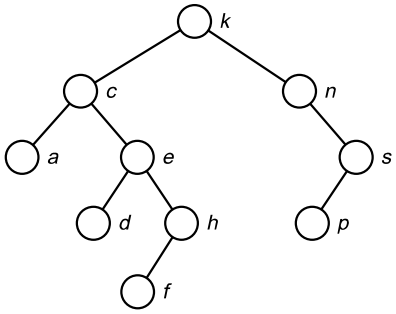
k, n, s (fails)

E2. *Insert each of the following keys into the preceding binary search tree. Show the comparisons of keys that will be made in each case. Do each part independently, inserting the key into the original tree.*

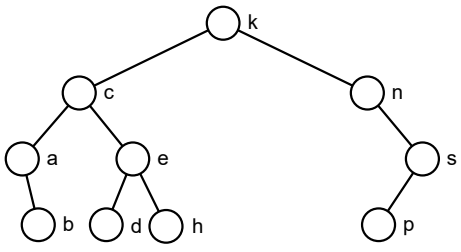
(a) *m*



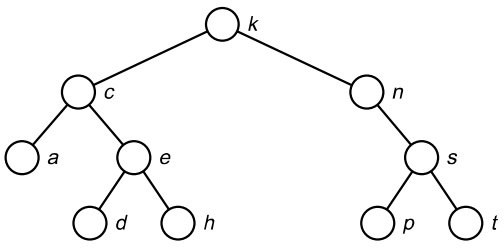
(b) *f*



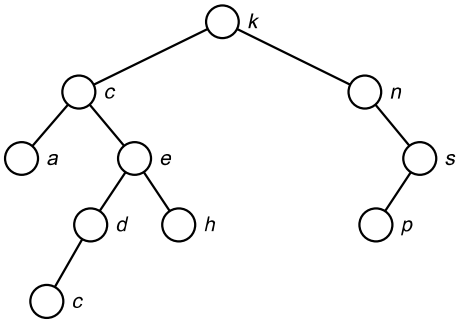
(c) *b*



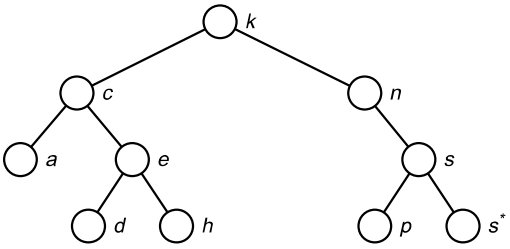
(d) *t*



(e) *c*

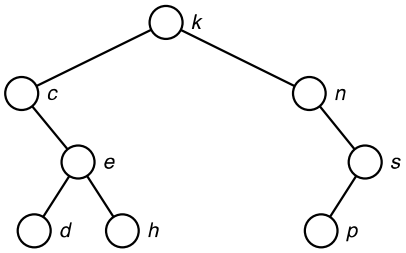


(f) *s*

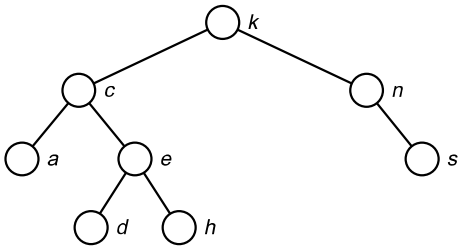


E3. *Delete each of the following keys from the preceding binary search tree, using the algorithm developed in this section. Do each part independently, deleting the key from the original tree.*

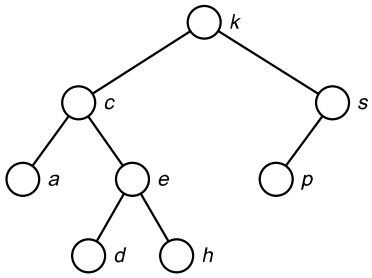
(a) *a*



(b) *p*



(c) *n*



(d) *s*