```
/*We have discussed BST search and insert operations.
In this post, the delete operation is discussed.
 When we delete a node, three possibilities arise.

1) Node to be deleted is the leaf: Simply remove from the t

              50                            50
           /     \         delete(20)      /   \
          30      70       --------->    30     70
         /  \    /  \                     \    /  \
       20   40  60   80                    40  60   80
2) Node to be deleted has only one child:
 Copy the child to the node and delete the child

              50                            50
           /     \         delete(30)      /   \
          30      70       --------->    40     70
            \    /  \                          /  \
            40  60   80                       60   80
3) Node to be deleted has two children: Find inorder succes
 Copy contents of the inorder successor to the node and del
 Note that inorder predecessor can also be used.


              50                            60
           /     \         delete(50)      /   \
          40      70       --------->    40    70
                 /  \                           \
                60   80                          80
The important thing to note is,
inorder successor is needed only when the right child is no
In this particular case, inorder successor can be obtained
the minimum value in the right child of the node.*/

// C program to demonstrate
// delete operation in binary
// search tree
#include <stdio.h>
#include <stdlib.h>
```

```
40:
41: struct node {
42:     int key;
43:     struct node *left, *right;
44: };
45:
46: // A utility function to create a new BST node
47: struct node* newNode(int item)
48: {
49:     struct node* temp
50:         = (struct node*)malloc(sizeof(struct node));
51:     temp->key = item;
52:     temp->left = temp->right = NULL;
53:     return temp;
54: }
55:
56: // A utility function to do inorder traversal of BST
57: void inorder(struct node* root)
58: {
59:     if (root != NULL) {
60:         inorder(root->left);
61:         printf("%d ", root->key);
62:         inorder(root->right);
63:     }
64: }
65:
66: /* A utility function to
67: insert a new node with given key in
68: * BST */
69: struct node* insert(struct node* node, int key)
70: {
71:     /* If the tree is empty, return a new node */
72:     if (node == NULL)
73:         return newNode(key);
74:
75:     /* Otherwise, recur down the tree */
76:     if (key < node->key)
77:         node->left = insert(node->left, key);
78:     else
```

```
 79:         node->right = insert(node->right, key);
 80:
 81:     /* return the (unchanged) node pointer */
 82:     return node;
 83: }
 84:
 85: /* Given a non-empty binary search
 86: tree, return the node
 87: with minimum key value found in
 88: that tree. Note that the
 89: entire tree does not need to be searched. */
 90: struct node* minValueNode(struct node* node)
 91: {
 92:     struct node* current = node;
 93:
 94:     /* loop down to find the leftmost leaf */
 95:     while (current && current->left != NULL)
 96:         current = current->left;
 97:
 98:     return current;
 99: }
100:
101: /* Given a binary search tree
102: and a key, this function
103: deletes the key and
104: returns the new root */
105: struct node* deleteNode(struct node* root, int key)
106: {
107:     // base case
108:     if (root == NULL)
109:         return root;
110:
111:     // If the key to be deleted
112:     // is smaller than the root's
113:     // key, then it lies in left subtree
114:     if (key < root->key)
115:         root->left = deleteNode(root->left, key);
116:
117:     // If the key to be deleted
```

```
118:     // is greater than the root's
119:     // key, then it lies in right subtree
120:     else if (key > root->key)
121:         root->right = deleteNode(root->right, key);
122: 
123:     // if key is same as root's key,
124:     // then This is the node
125:     // to be deleted
126:     else {
127:         // node with only one child or no child
128:         if (root->left == NULL) {
129:             struct node* temp = root->right;
130:             free(root);
131:             return temp;
132:         }
133:         else if (root->right == NULL) {
134:             struct node* temp = root->left;
135:             free(root);
136:             return temp;
137:         }
138: 
139:         // node with two children:
140:         // Get the inorder successor
141:         // (smallest in the right subtree)
142:         struct node* temp = minValueNode(root->right);
143: 
144:         // Copy the inorder
145:         // successor's content to this node
146:         root->key = temp->key;
147: 
148:         // Delete the inorder successor
149:         root->right = deleteNode(root->right, temp->key);
150:     }
151:     return root;
152: }
153: 
154: // Driver Code
155: int main()
156: {
```

```
157:     /* Let us create following BST
158:               50
159:            /     \
160:            30    70
161:            / \   / \
162:     20 40 60 80 */
163:     struct node* root = NULL;
164:     root = insert(root, 50);
165:     root = insert(root, 30);
166:     root = insert(root, 20);
167:     root = insert(root, 40);
168:     root = insert(root, 70);
169:     root = insert(root, 60);
170:     root = insert(root, 80);
171:
172:     printf("Inorder traversal of the given tree \n");
173:     inorder(root);
174:
175:     printf("\nDelete 20\n");
176:     root = deleteNode(root, 20);
177:     printf("Inorder traversal of the modified tree \n");
178:     inorder(root);
179:
180:     printf("\nDelete 30\n");
181:     root = deleteNode(root, 30);
182:     printf("Inorder traversal of the modified tree \n");
183:     inorder(root);
184:
185:     printf("\nDelete 50\n");
186:     root = deleteNode(root, 50);
187:     printf("Inorder traversal of the modified tree \n");
188:     inorder(root);
189:
190:     return 0;
191: }
192:
```