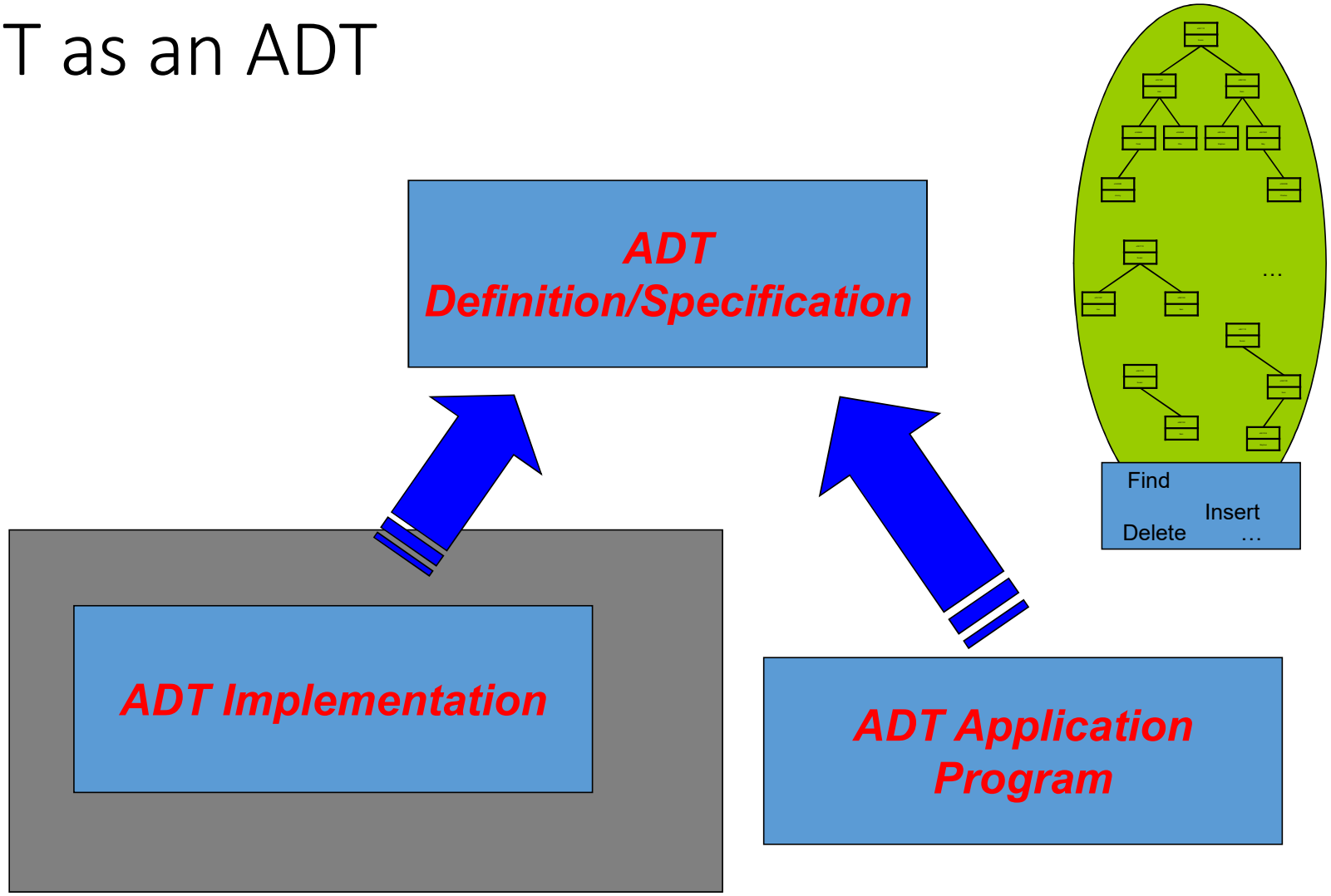
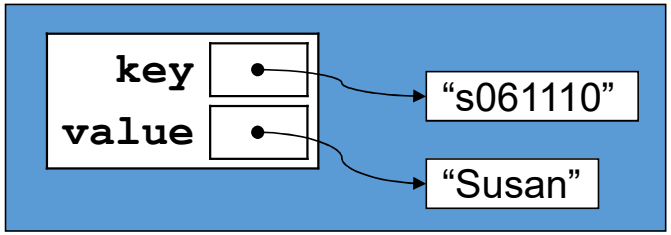
BST as an ADT


ADT
Definition/Specification
ADT Implementation
ADT Application
Program
Find
Insert
Delete
…

Implementing the Tree Node ADT

treenode.c

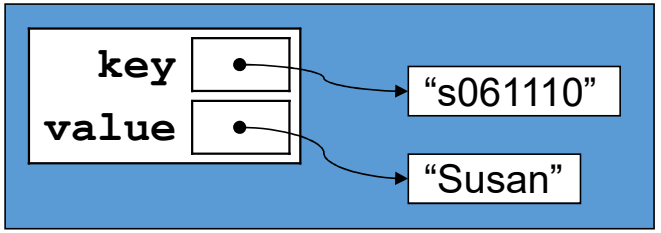
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "treenode.h"

struct treeNodeCDT {
   char *key;
   void *value;
};

treeNodeADT NewNode(char *key, void *value) {
   treeNodeADT node;
   node = (treeNodeADT)malloc(sizeof(struct treeNodeCDT));
   node->key = (char *)malloc((strlen(key) + 1) * sizeof(char));
   strcpy(node->key, key);
   node->value = value;
   return node;
}

treeNodeADT CopyNode(treeNodeADT n) {
   return NewNode(n->key, n->value);
}
```



Implementing the Tree Node ADT

treenode.c (continue)

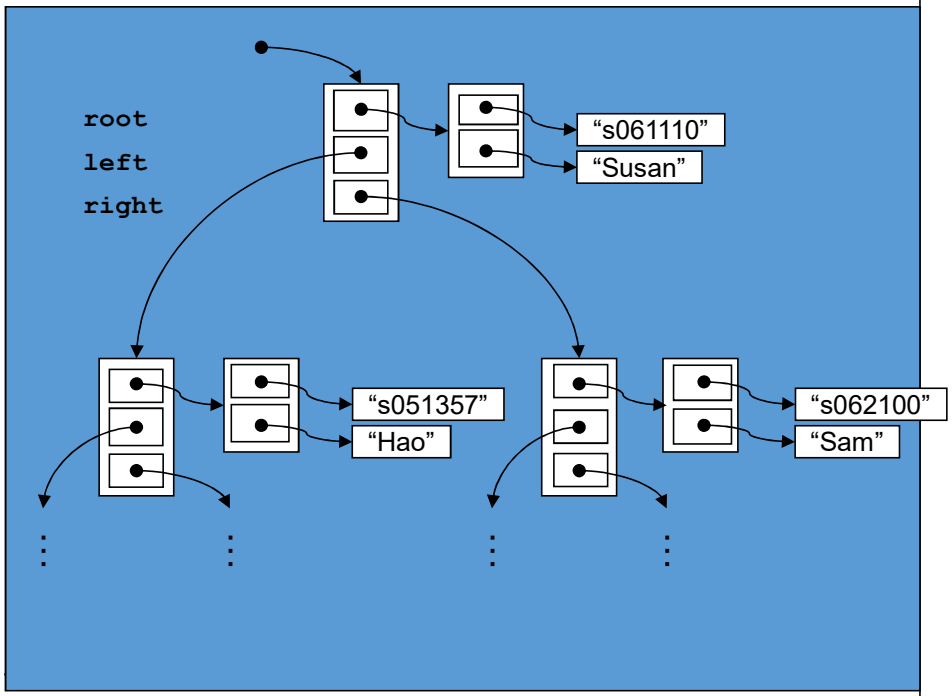
```
char *GetNodeKey(treeNodeADT n) {
   if (n == NULL)
      exit(0);
   return n->key;
}

void *GetNodeValue(treeNodeADT n) {
   if (n == NULL)
      exit(0);
   return n->value;
}

void DelNode(treeNodeADT n) {
   free(n->key);
   free(n);
}
```



Implementing the BST ADT

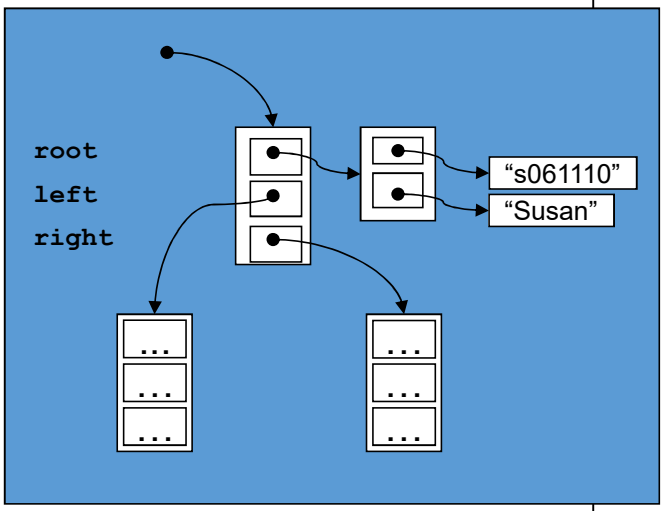
bst.c

```
#include <stdio.h>
#include <stdlib.h>
#indlude <string.h>
#include "bst.h"

struct bstCDT {
    treeNodeADT root;
    bstADT left;
    bstADT right;
};

bstADT EmptyBST() {
    return NULL;
}

int BSTIsEmpty(bstADT t)
    return (t == NULL);
}
```



bst.c (continue)

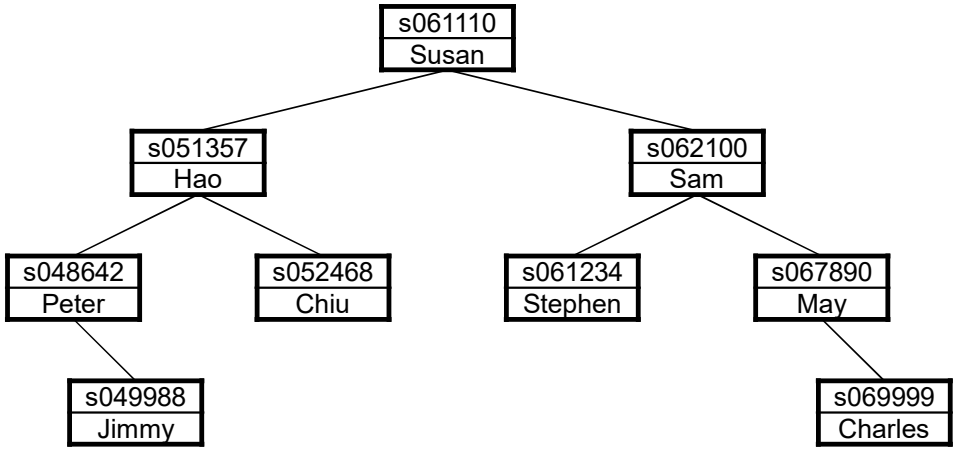
```
bstADT MakeBST(treeNodeADT root, bstADT left, bstADT right) {
   bstADT t;
   t = (bstADT)malloc(sizeof(struct bstCDT));
   t->root = root;
   t->left = left;
   t->right = right;
   return t;
}

treeNodeADT Root(bstADT t) {
   if ((BSTIsEmpty(t)))
      exit(0);
   return t->root;
}

bstADT LeftSubtree(bstADT t) {
   if ((BSTIsEmpty(t)))
      exit(0);
   return t->left;
}

bstADT RightSubtree(bstADT t) {
   if ((BSTIsEmpty(t)))
      exit(0);
   return t->right;
}
```



```
if Tree is empty
     return NULL;
if search key = key of root
     return value root;
   else if (search key < key of root)
     return search from LeftSubtree;
   else
     return search from RightSubtree;
```

FindNode

Find the node in the BST that matches the key.



FindNode

bst.c (continue)

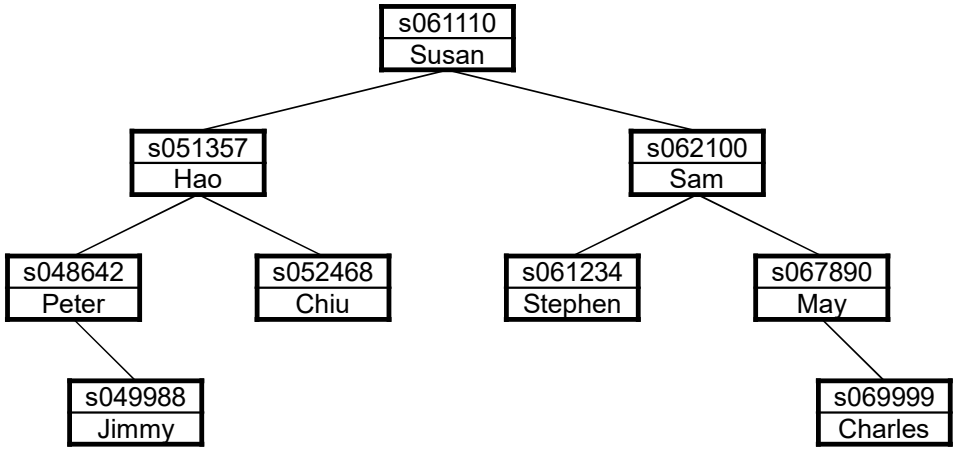
```
treeNodeADT FindNode(bstADT t, char *key) {
   int sign;
   if (BSTIsEmpty(t))
      return NULL;
   sign = strcmp(key, GetNodeKey(Root(t)));
   if (sign == 0)
      return Root(t);
   else if (sign < 0)
      return FindNode(LeftSubtree(t), key);
   else
      return FindNode(RightSubtree(t), key);
}
```

Compare two strings s1 and s2

- returns 0 if they are identical
- -ve if the ASCII value of s1 < s2
- +ve is the ASCII value of s1 > s2

Example 1:

"s061110"?

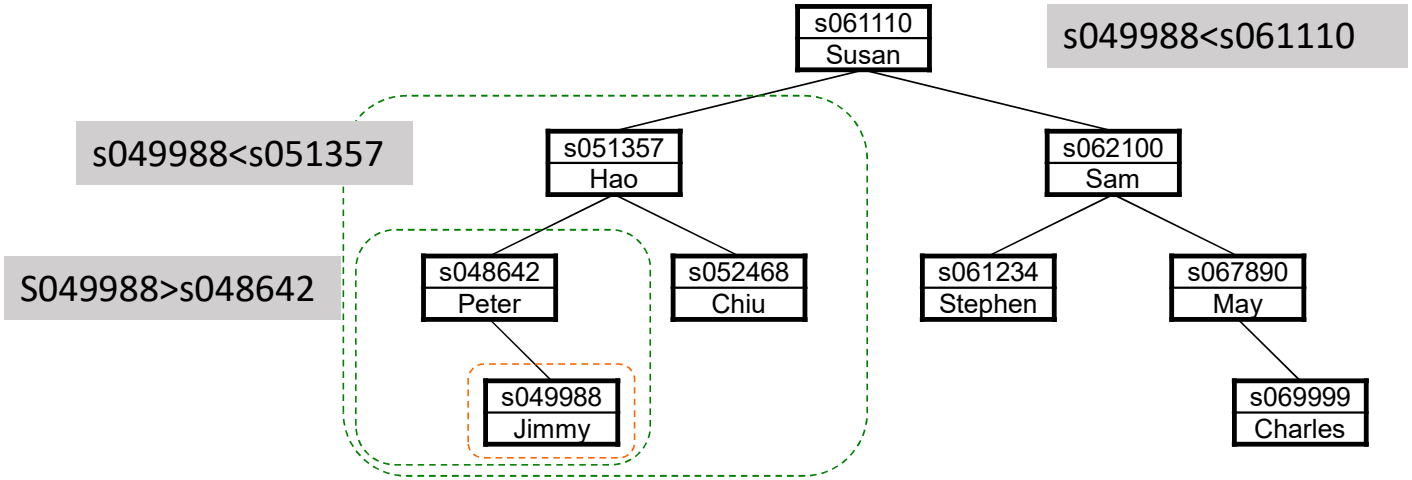


FindNode

bst.c (continue)

```
treeNodeADT FindNode(bstADT t, char *key) {
   int sign;
   if (BSTIsEmpty(t))
      return NULL;
   sign = strcmp(key, GetNodeKey(Root(t)));
   if (sign == 0)
      return Root(t);
   else if (sign < 0)
      return FindNode(LeftSubtree(t), key);
   else
      return FindNode(RightSubtree(t), key);
}
```

Example 2:

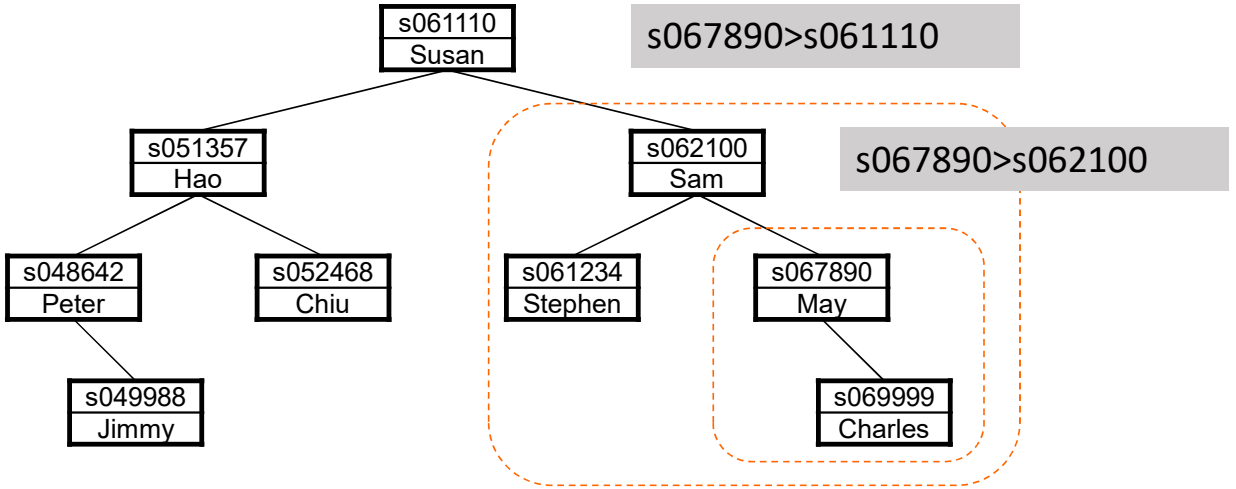


FindNode

bst.c (continue)

```
treeNodeADT FindNode(bstADT t, char *key) {
   int sign;
   if (BSTIsEmpty(t))
      return NULL;
   sign = strcmp(key, GetNodeKey(Root(t)));
   if (sign == 0)
      return Root(t);
   else if (sign < 0)
      return FindNode(LeftSubtree(t), key);
   else
      return FindNode(RightSubtree(t), key);
}
```

Example 3:

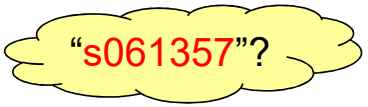


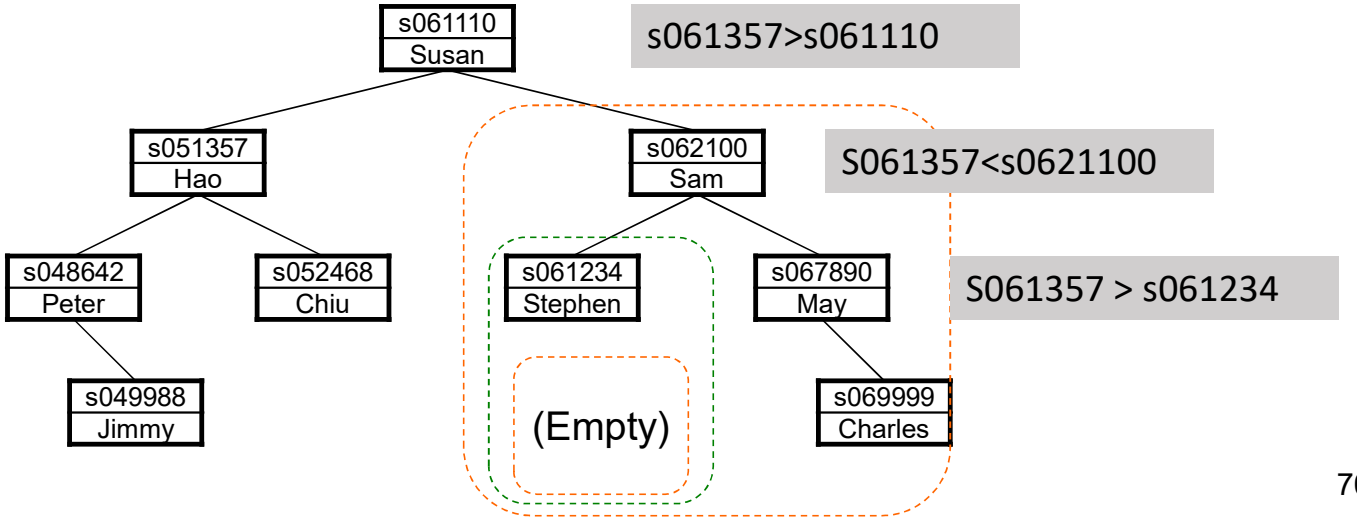
FindNode

bst.c (continue)

```
treeNodeADT FindNode(bstADT t, char *key) {
   int sign;
   if (BSTIsEmpty(t))
      return NULL;
   sign = strcmp(key, GetNodeKey(Root(t)));
   if (sign == 0)
      return Root(t);
   else if (sign < 0)
      return FindNode(LeftSubtree(t), key);
   else
      return FindNode(RightSubtree(t), key);
}
```

Example 4:

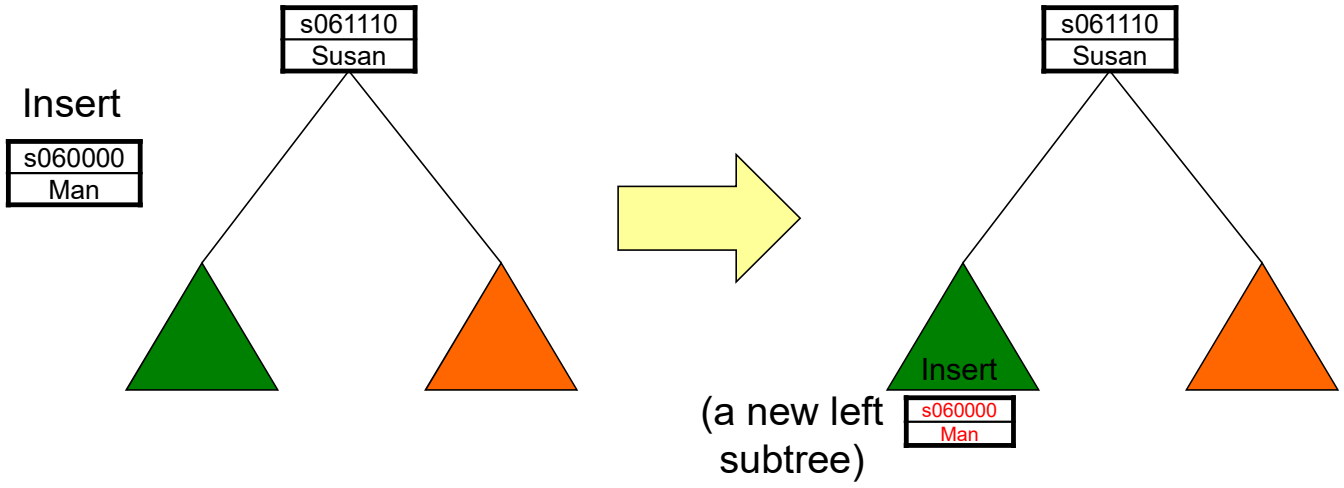




```
• if tree is empty
      return MakeBST(n, NULL, NULL)
• if root has the same key
      replace the root by the node
• if key < key of root
      insert node to the left subtree
• if key > key of root
      insert node to the right subtree
```

Insert Node

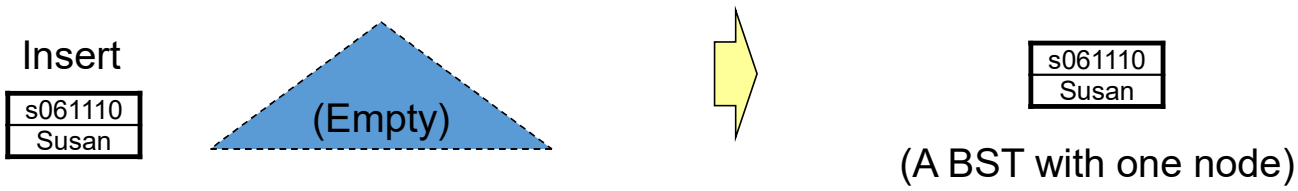
Insert a node (key + value) into the BST.



Insert Node

bst.c (continue)

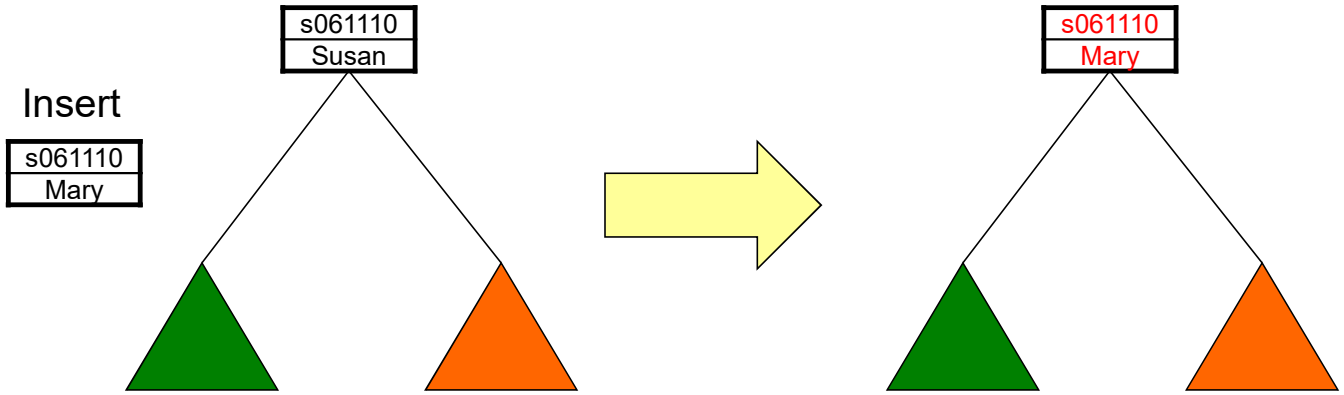
```
bstADT InsertNode(bstADT t, treeNodeADT n) {
    int sign;
    if (BSTIsEmpty(t))
        return MakeBST(n, NULL, NULL);
    sign = strcmp(n->key, GetNodeKey(Root(t)));
    if (sign == 0){
        DelNode( t->root );  t->root = n;
    }
    else if (sign < 0){
        t->left = InsertNode(LeftSubtree(t), n);
    else
        t->right = InsertNode(RightSubtree(t), n);
    return t;
}
```



Insert Node

bst.c (continue)

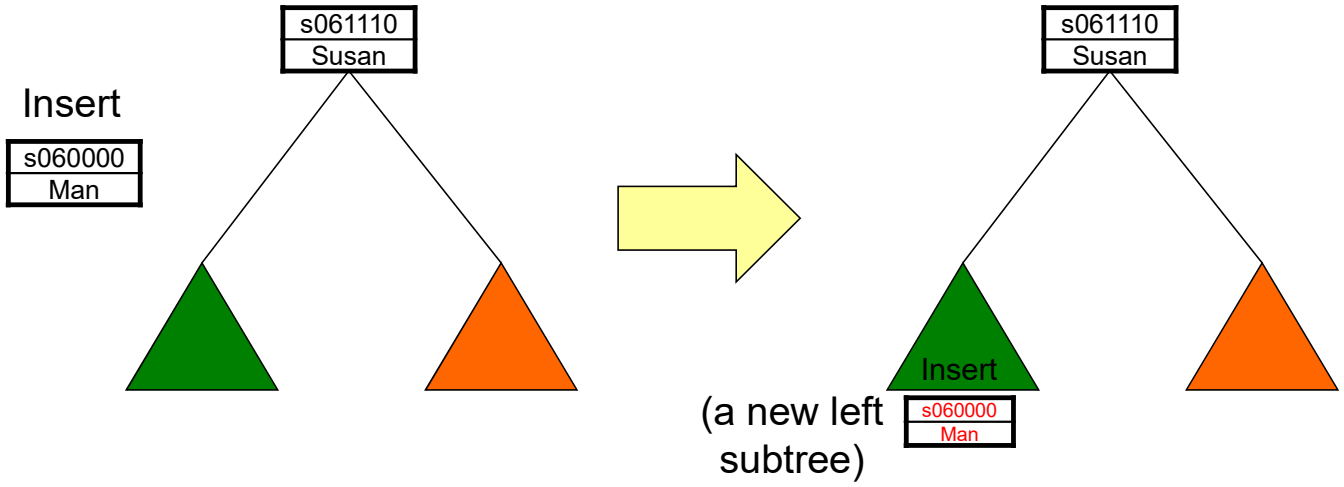
```
bstADT InsertNode(bstADT t, treeNodeADT n) {
   int sign;
   if (BSTIsEmpty(t))
      return MakeBST(n, NULL, NULL);
   sign = strcmp(n->key, GetNodeKey(Root(t)));
   if (sign == 0){
      DelNode( t->root );      t->root = n;
   }
   else if (sign < 0){
      t->left = InsertNode(LeftSubtree(t), n);
   else
      t->right = InsertNode(RightSubtree(t), n);
   return t;
}
```



Insert Node

bst.c (continue)

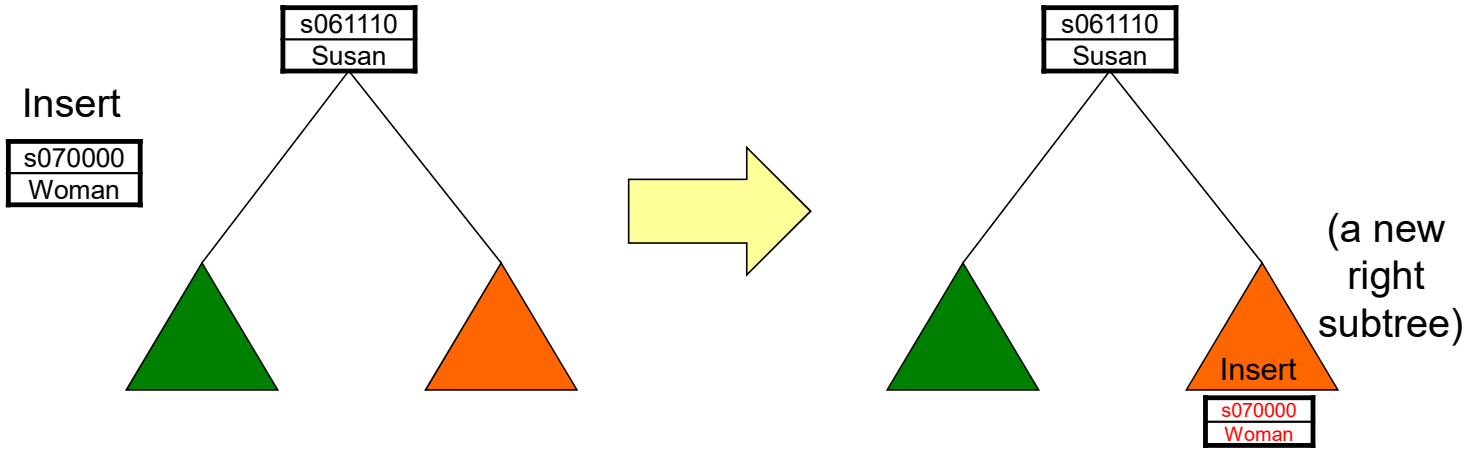
```
bstADT InsertNode(bstADT t, treeNodeADT n) {
    int sign;
    if (BSTIsEmpty(t))
        return MakeBST(n, NULL, NULL);
    sign = strcmp(n->key, GetNodeKey(Root(t)));
    if (sign == 0){
        DelNode( t->root );        t->root = n;
    }
    else if (sign < 0){
        t->left = InsertNode(LeftSubtree(t), n);
    else
        t->right = InsertNode(RightSubtree(t), n);
    return t;
}
```



Insert Node

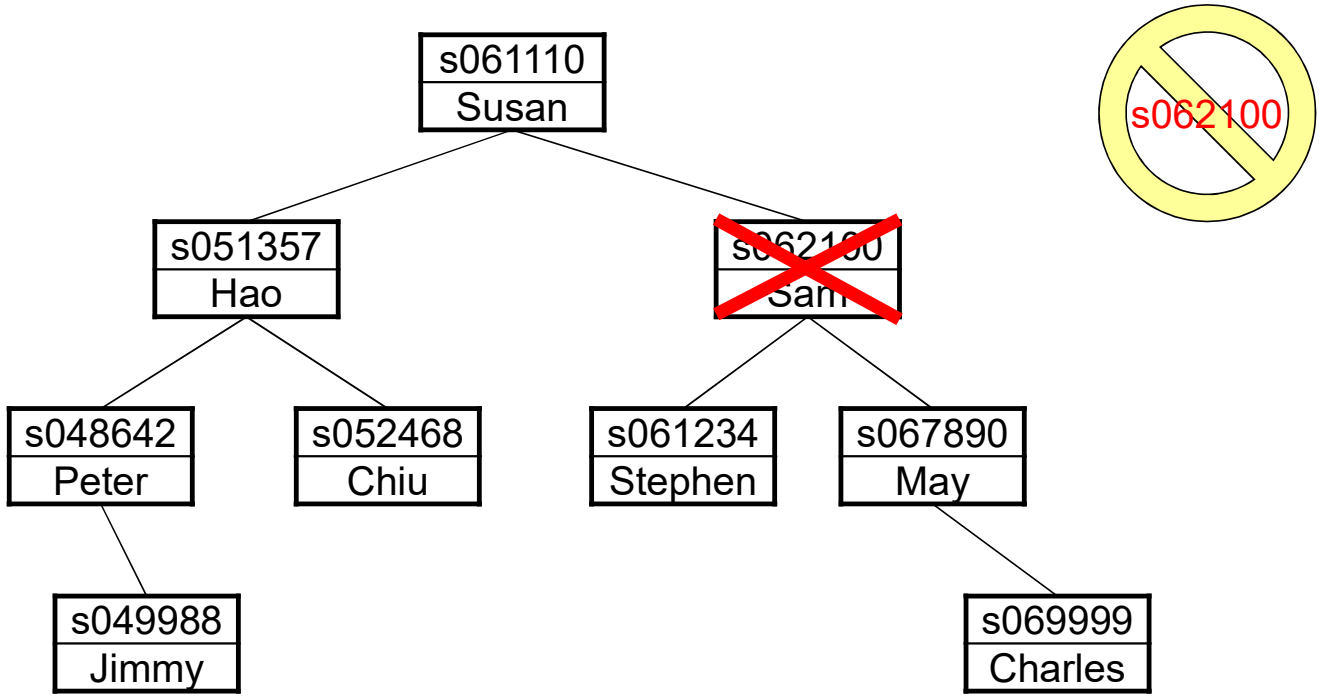
bst.c (continue)

```
bstADT InsertNode(bstADT t, treeNodeADT n) {
 int sign;
   if (BSTIsEmpty(t))
      return MakeBST(n, NULL, NULL);
   sign = strcmp(n->key, GetNodeKey(Root(t)));
   if (sign == 0){
      DelNode( t->root );       t->root = n;
   }
   else if (sign < 0){
      t->left = InsertNode(LeftSubtree(t), n);
   else
      t->right = InsertNode(RightSubtree(t), n);
   return t;
}
```



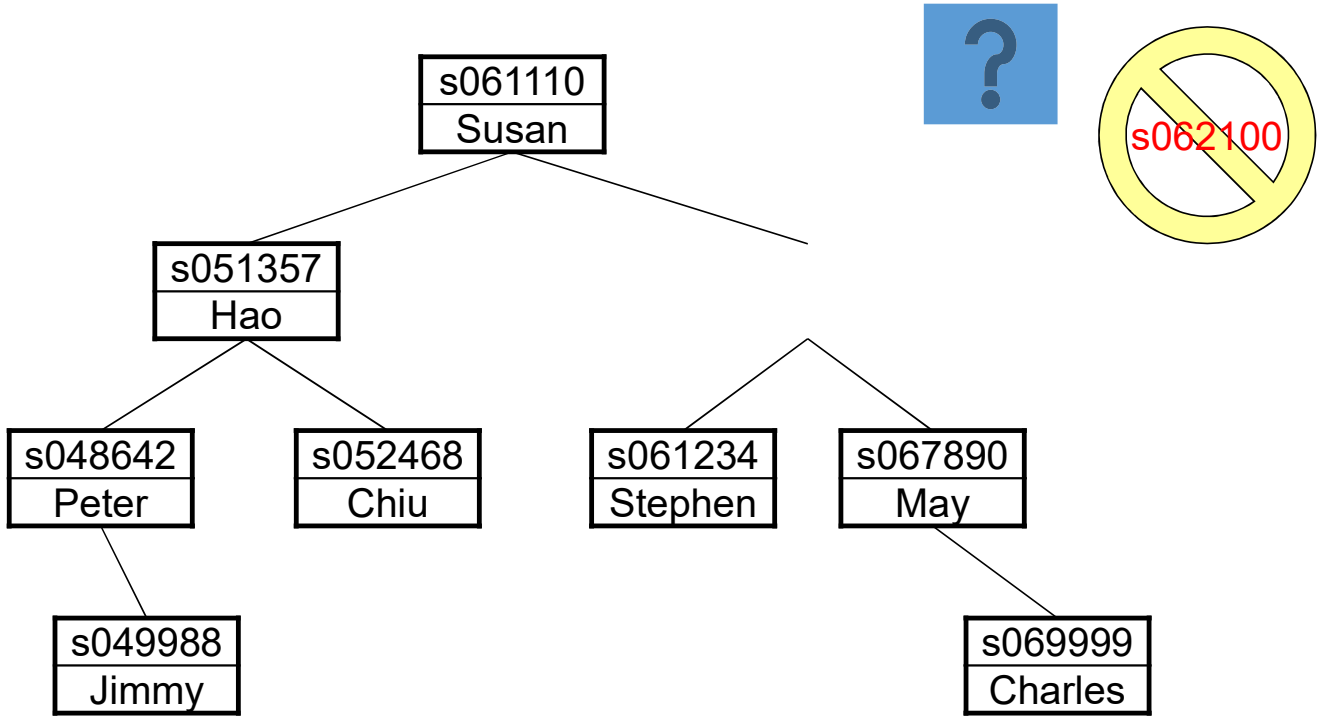
Deleting a Node

- Deleting a node from a BST is a little bit more complicated.



Deleting a Node

- Deleting a node from a BST is a little bit more complicated.

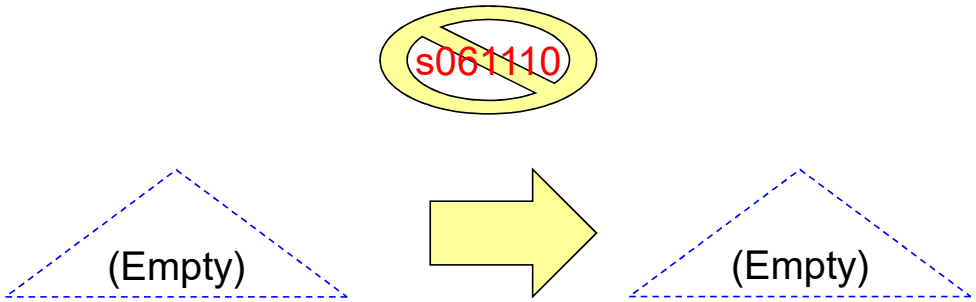


Four situations

- Case 1 : tree is empty
  Return the empty tree
- Case 2 : node to be deleted is in either the left or right sub-tree
  Delete the node from the left or right sub-tree
- Case 3 : node to be deleted is the root node, and one or both subtrees are empty
  Return the “other” sub-tree
- Case 4 : node to be deleted is the root node, and both the left and right subtrees are not empty
  Find a substitution from the sub-tree to replace the root node

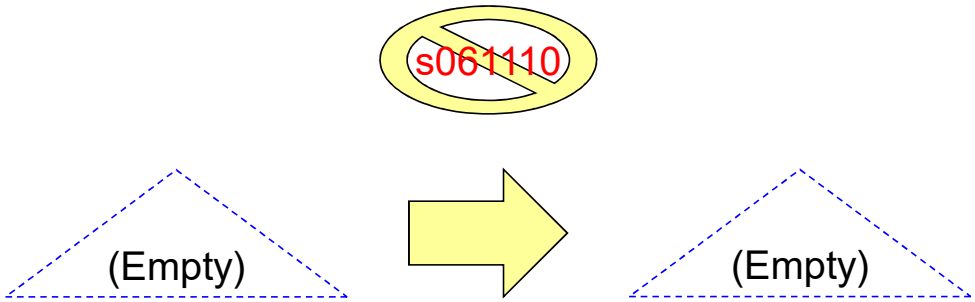
Deleting a Node

- Let's consider the easier cases first.
- Case 1: deleting anything from an *empty BST* still gives an *empty BST*.



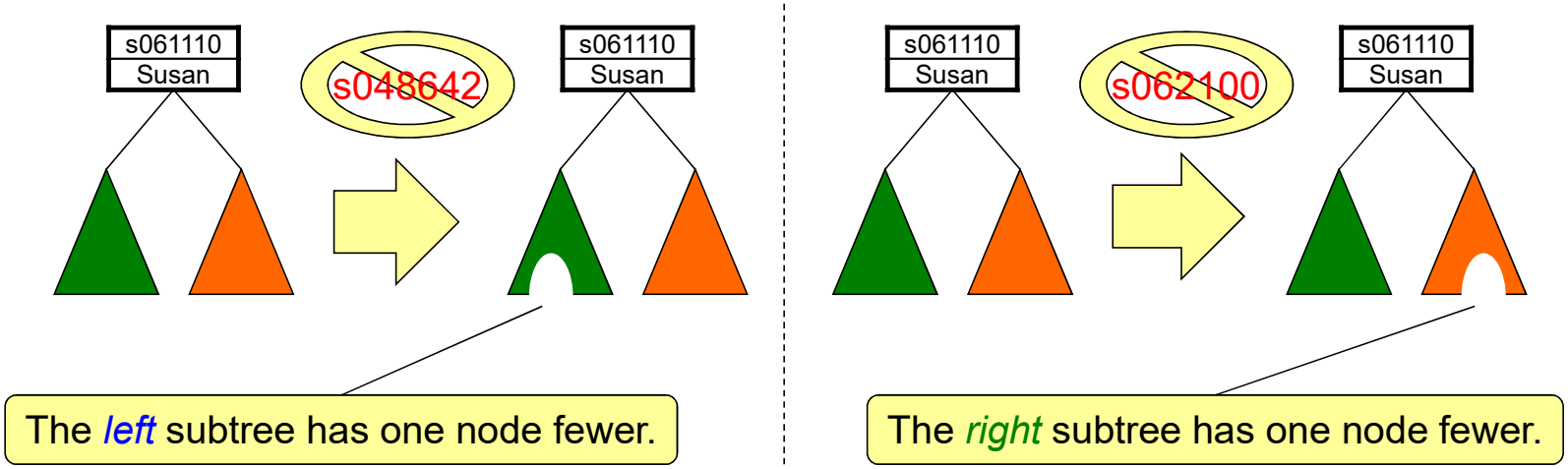
Node Deletion: Case 1

```
if (BSTIsEmpty(t))
   return t;
```

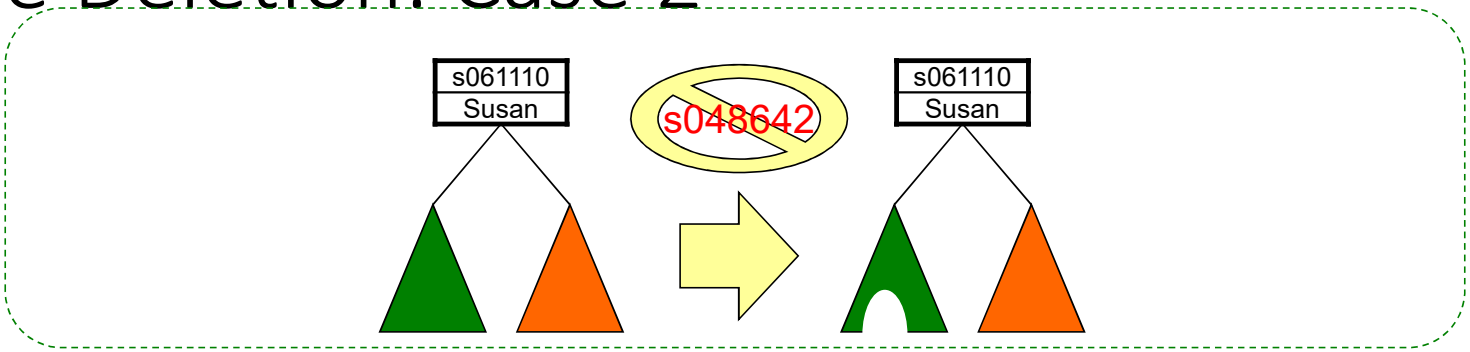


Deleting a Node

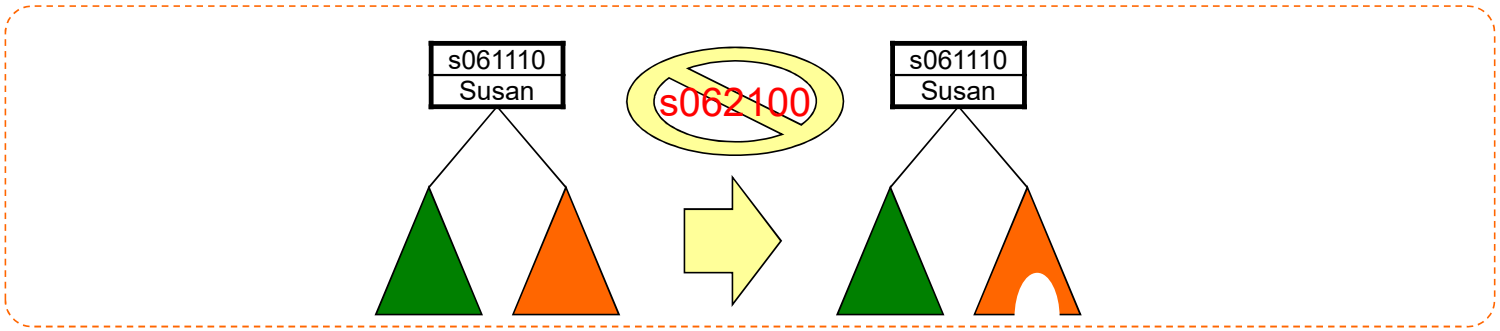
- Case 2: deleting a *non-root* node is easy. We simply rely on recursive calls.
  - *Recursively* delete the node in the left/right subtree.



Node Deletion: Case 2

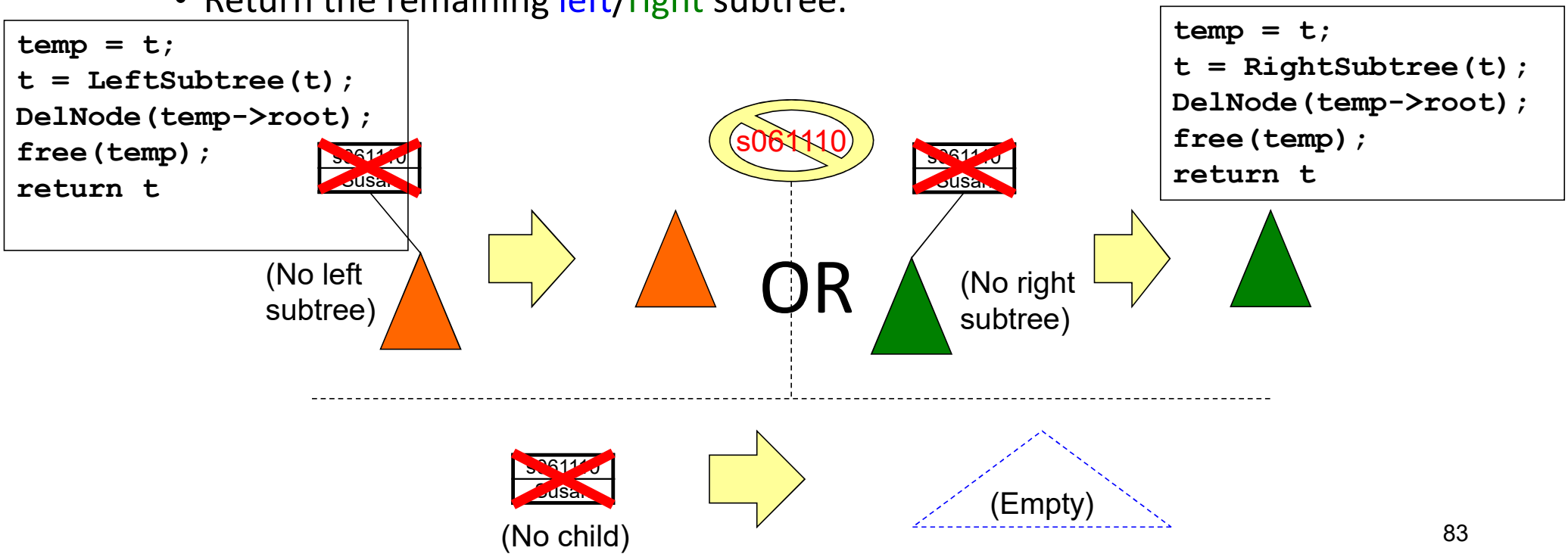



```
if (sign < 0)
        t->left = DeleteNode(LeftSubtree(t), key);
else if (sign > 0)
        t->right = DeleteNode(RightSubtree(t), key);
```

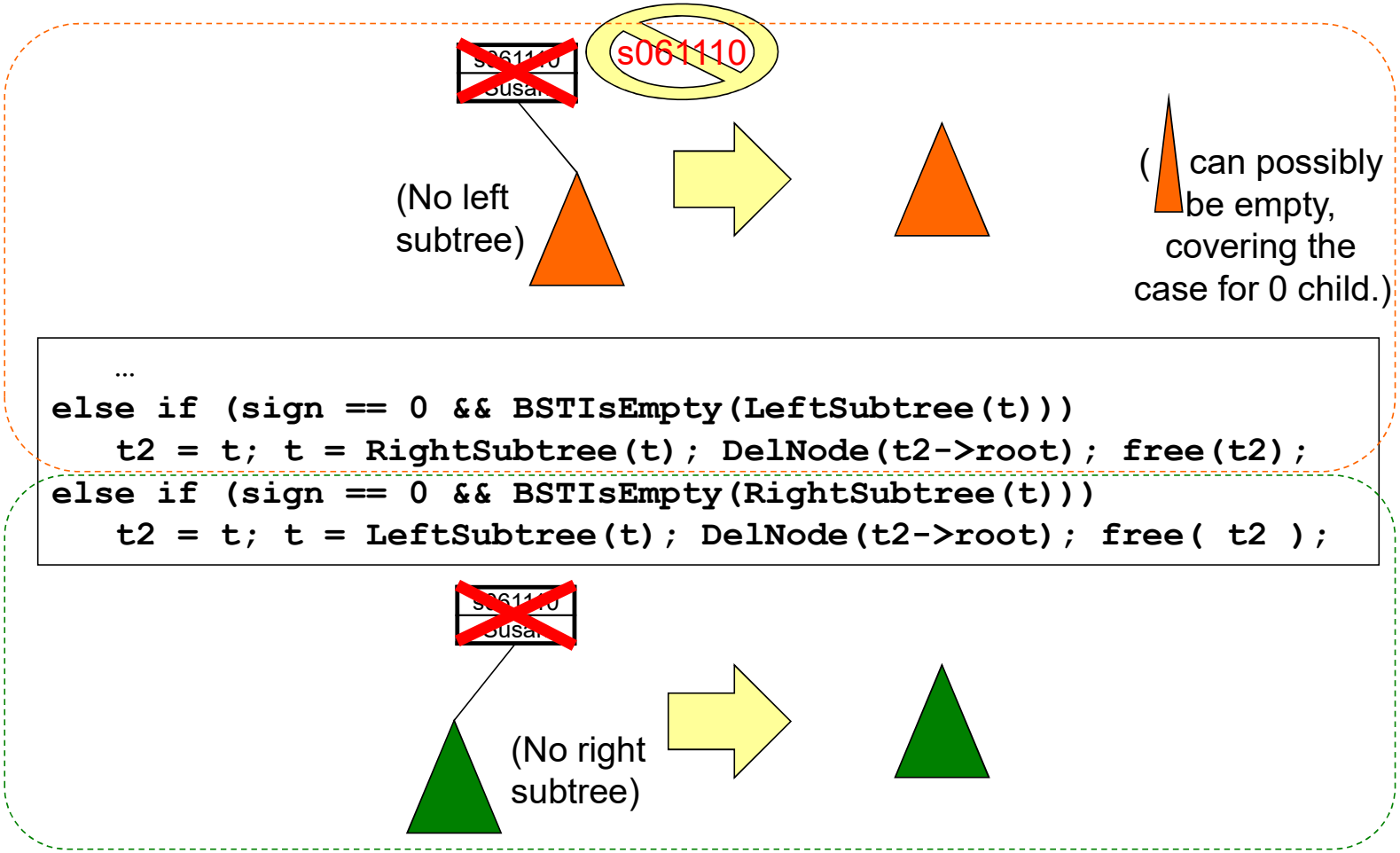


Deleting a Node

- Case 3: deleting a *root* with *0 or 1 child* is also easy.
  - Return the remaining left/right subtree.



Node Deletion: Case 3

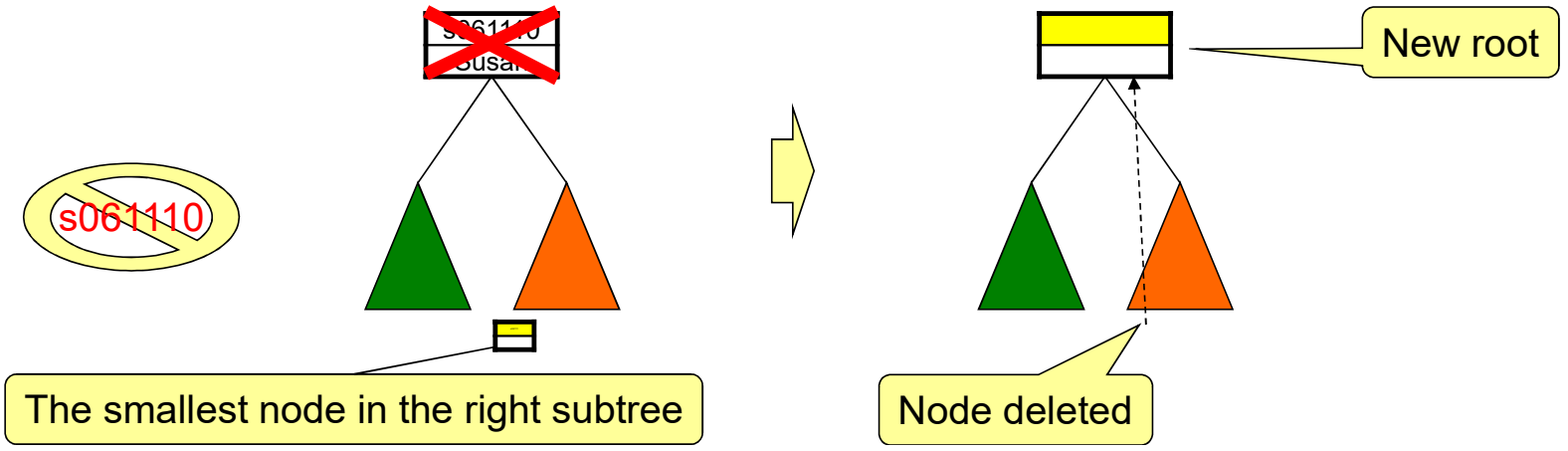



```
   ...
else if (sign == 0 && BSTIsEmpty(LeftSubtree(t)))
   t2 = t; t = RightSubtree(t); DelNode(t2->root); free(t2);
else if (sign == 0 && BSTIsEmpty(RightSubtree(t)))
   t2 = t; t = LeftSubtree(t); DelNode(t2->root); free( t2 );
```

Deleting a Node

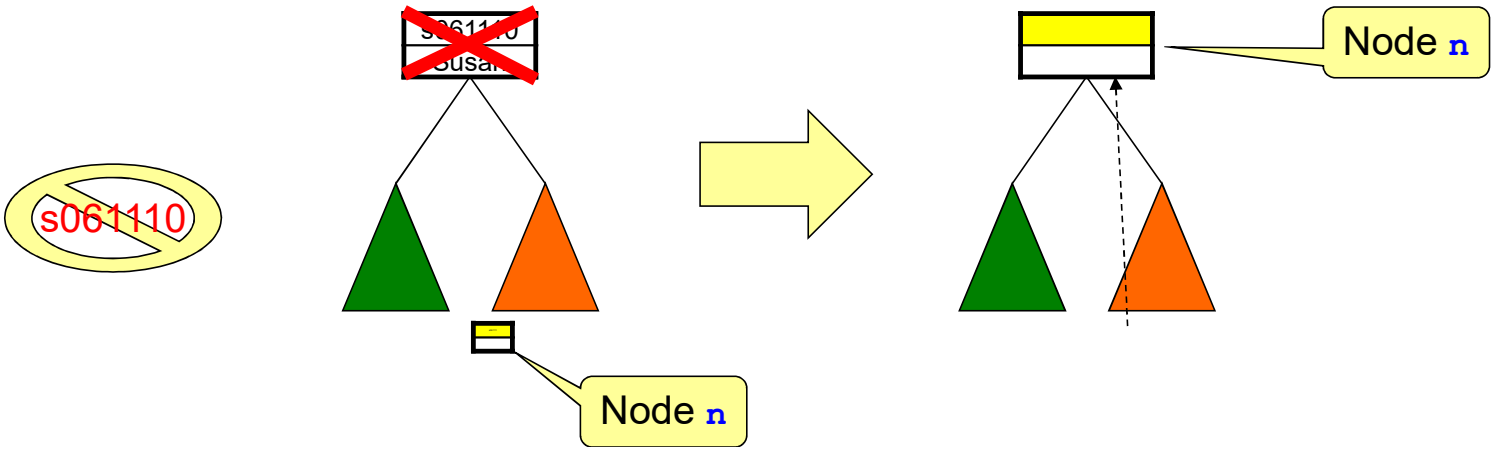
```
n = GetMinNode(RightSubtree(t));
DelNode( t->root );
t->root = CopyNode(n);
t->right = DeleteNode(RightSubtree(t), GetNodeKey(n));
```

- Case 4: the remaining case is to delete a *root* with *2 children*.
  - Find the node with the *smallest* key in the *right* substree. (Alternatively, find the node with the *largest* key in the *left* subtree.)
  - Delete that node from the right subtree
  - Use that node as the new root.

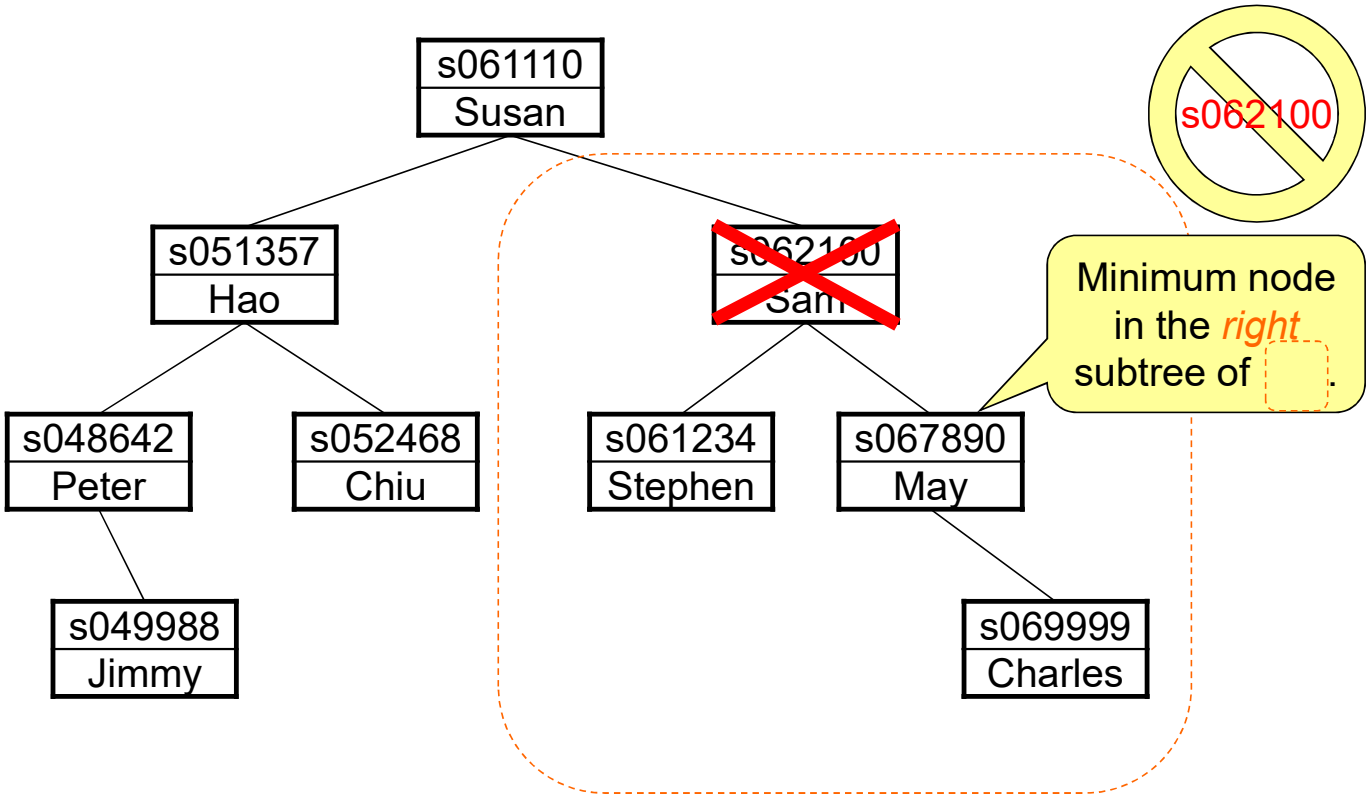


Node Deletion: Case 4

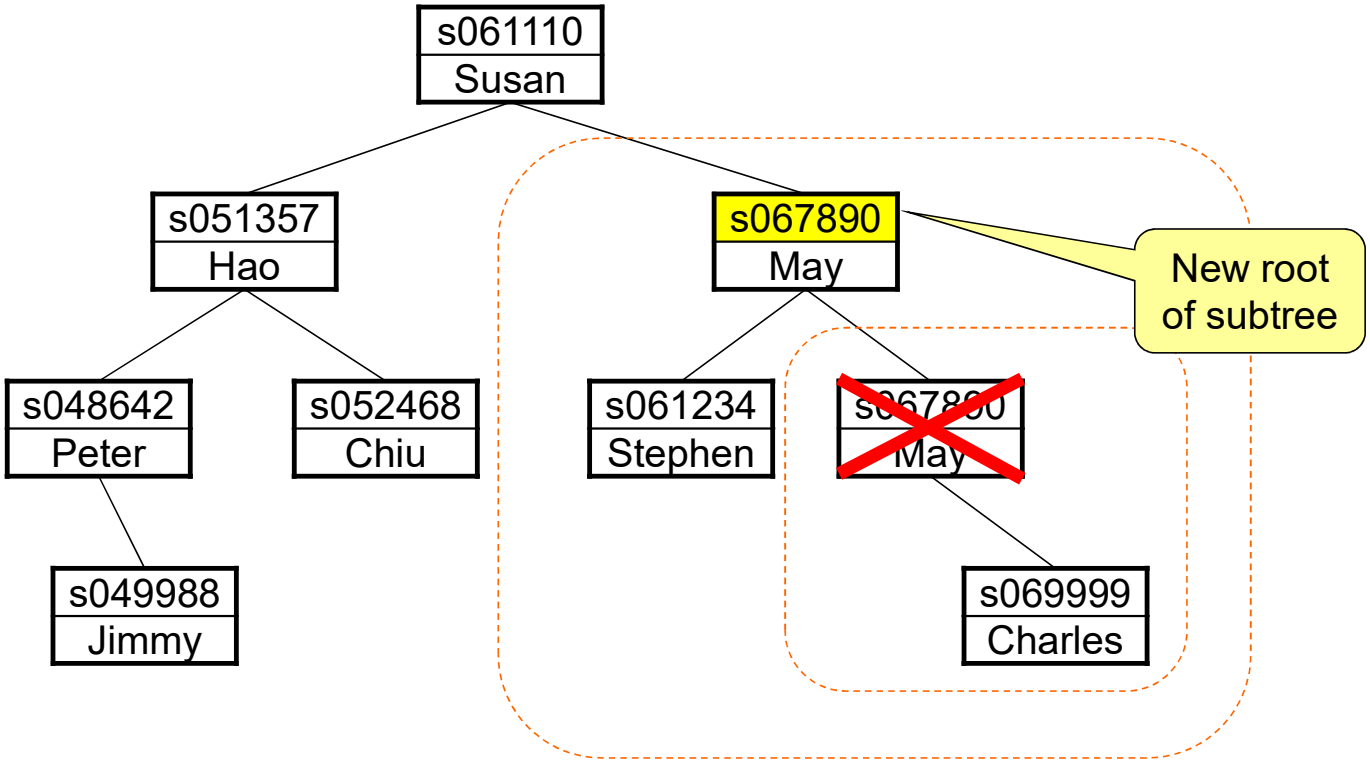
```
    …
else {
       n = GetMinNode(RightSubtree(t));
       DelNode( t->root );
       t->root = CopyNode(n);
       t->right = DeleteNode(RightSubtree(t), GetNodeKey(n));
}
```



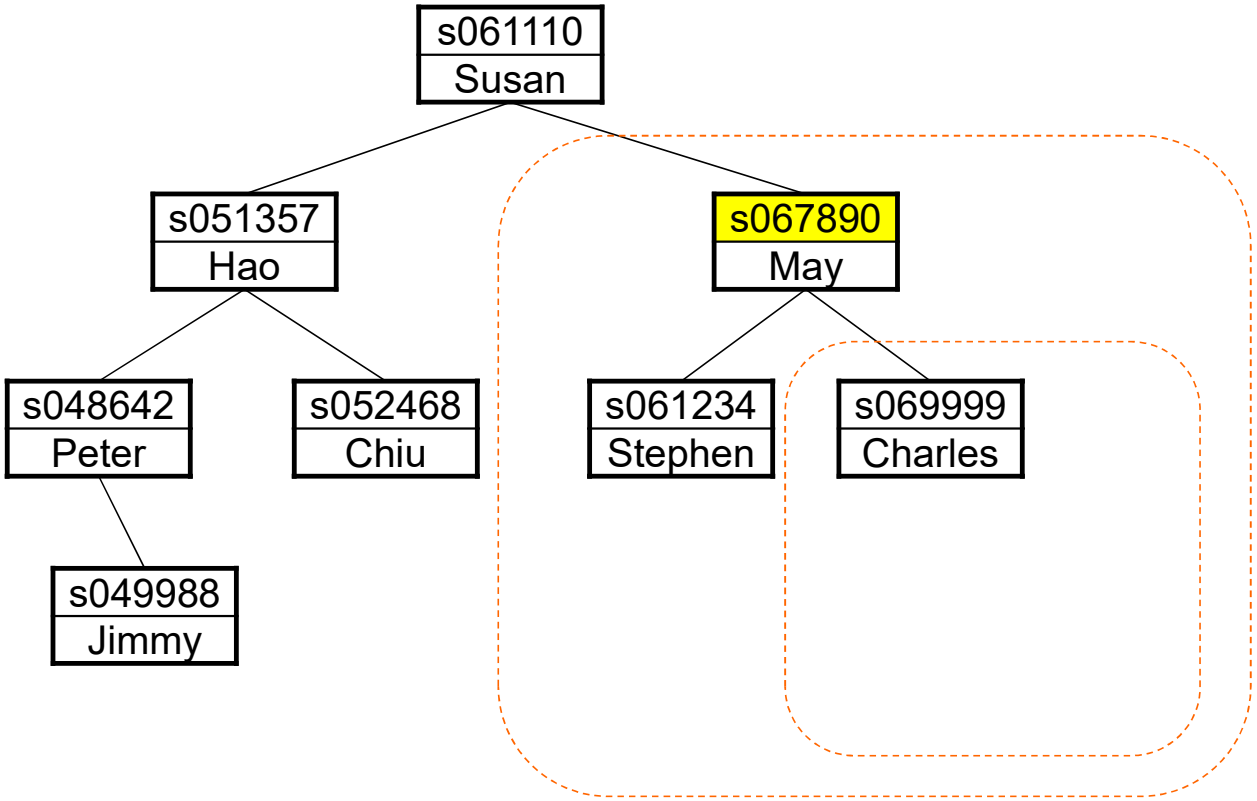
Deleting a Node: Example



Deleting a Node: Example



Deleting a Node: Example



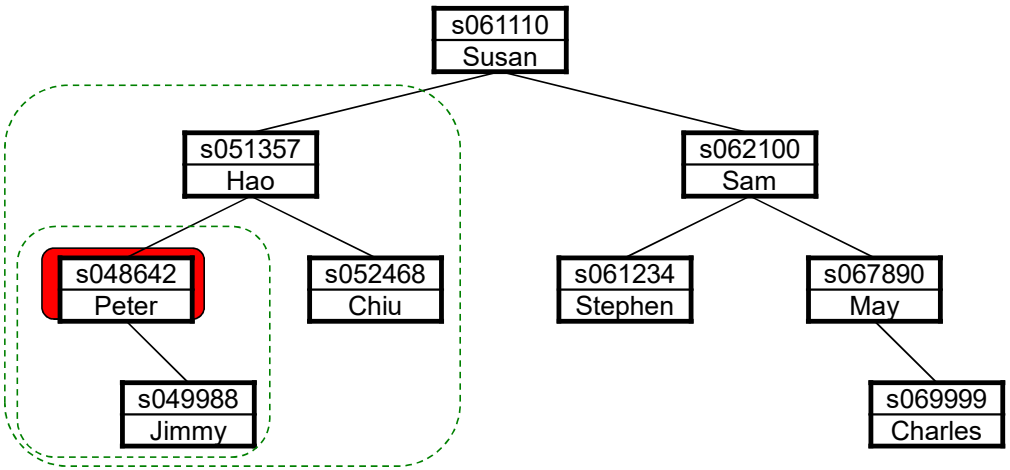
Finding the Minimum Node

bst.c (continue)

```
treeNodeADT GetMinNode(bstADT t) {
   if (BSTIsEmpty(LeftSubtree(t)))
      return t->root;
   else
      return GetMinNode(LeftSubtree(t));
}
```

The node with the minimum key must be the *furthest left descendent*.

No *left* subtree



bst.c (continue)

```
bstADT DeleteNode(bstADT t, char *key) {
  int sign;  treeNodeADT n; bstADT t2;
  if (BSTIsEmpty(t))      return t;
  sign = strcmp(key, GetNodeKey(Root(t)));
  if (sign < 0)
    t->left = DeleteNode(LeftSubtree(t), key);
  else if (sign > 0)
    t->right = DeleteNode(RightSubtree(t), key);
  else{
    if (BSTIsEmpty(LeftSubtree(t))){
      t2 = t; t = RightSubtree(t); DelNode(t2->root); free(t2);
    }
    else if (BSTIsEmpty(RightSubtree(t))){
      t2 = t; t = LeftSubtree(t); DelNode(t2->root); free( t2 );
    }
    else {
      n = GetMinNode(RightSubtree(t));
      DelNode( t->root );
      t->root = CopyNode(n);
      t->right = DeleteNode(RightSubtree(t), GetNodeKey(n));
  }
  return t;
}
```

Case 1: empty BST

Case 2: non-root, go to left tree

Case 2: non-root, go to right tree

Case 3: root with right subtree only

Delete root node

Case 3: root with left subtree only

Case 4: root with 2 children