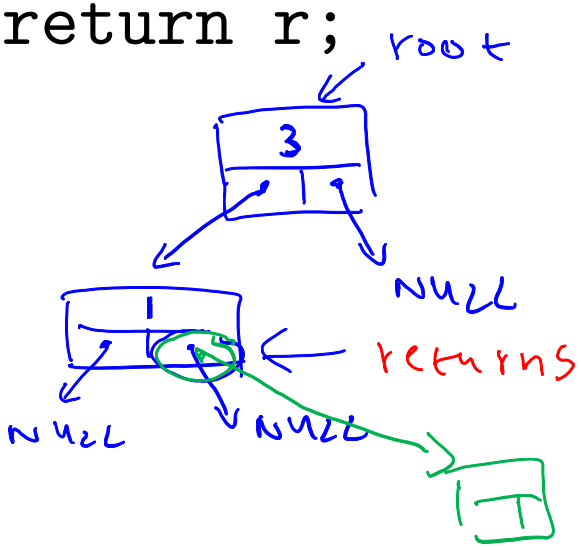
Find a node in a Binary Search Tree

```
Node *& find(KType key, Node *& r) {
  if (r == NULL) return r;
  if (key < r->key)
    return find(key, r->left);
  if (key > r->key)
    return find(key, r->right);
  return r;
}
```



find(z, root) = new Node;

we can use it to update right pointer of 1

reference to right pointer of this node (with value NULL)

Insert into a Binary Search Tree

reference to a pointer

```
void insert(KType key, Node *& root) {
  Node *& target = find(key, root);
  if( target != NULL) {
    cerr << "Duplicate:" << key << "\n";
  }
  target = new Node(key);
}
```

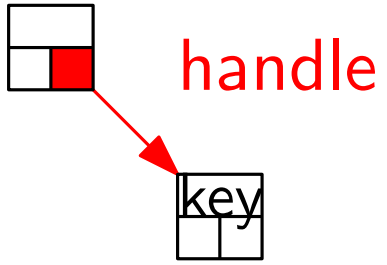
Find node with minimum key in BST

```
Node *& findMin(Node *& root) {
  if( root == NULL || root->left == NULL )
    return root;
  return findMin(root->left);
}
```

Remove a node from a Binary Search Tree

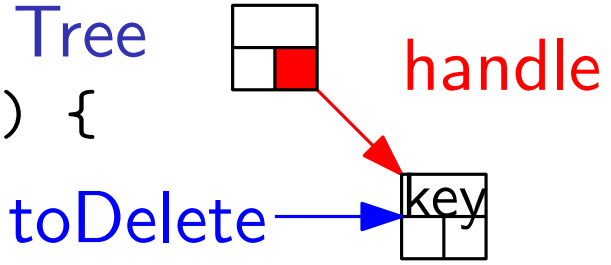
```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

Remove a node from a Binary Search Tree



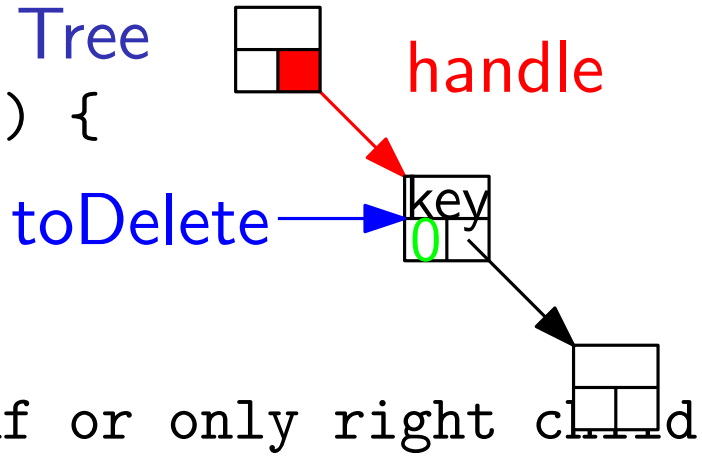

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

Remove a node from a Binary Search Tree



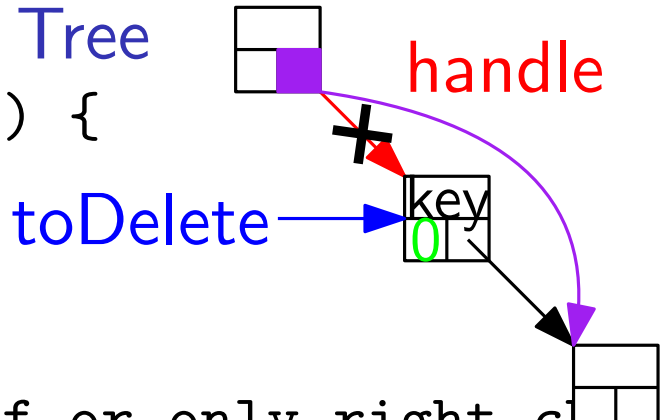

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

Remove a node from a Binary Search Tree



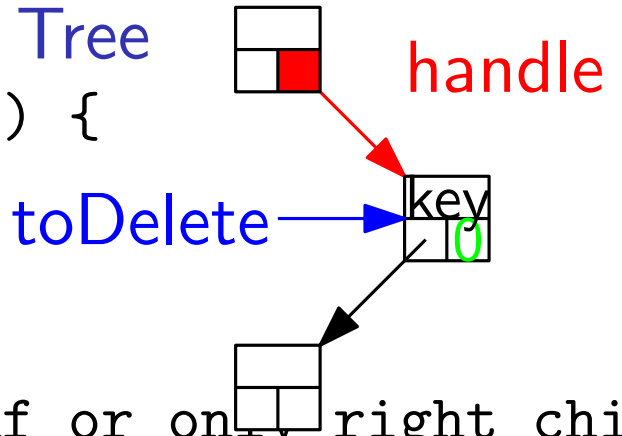

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

Remove a node from a Binary Search Tree



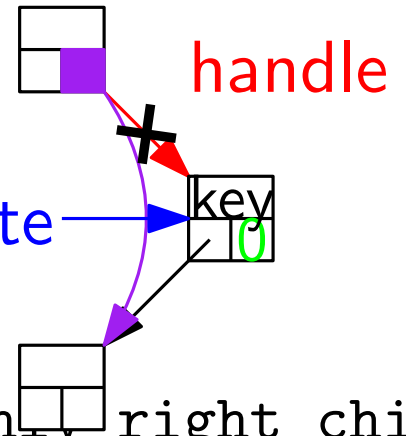

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

Remove a node from a Binary Search Tree




```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

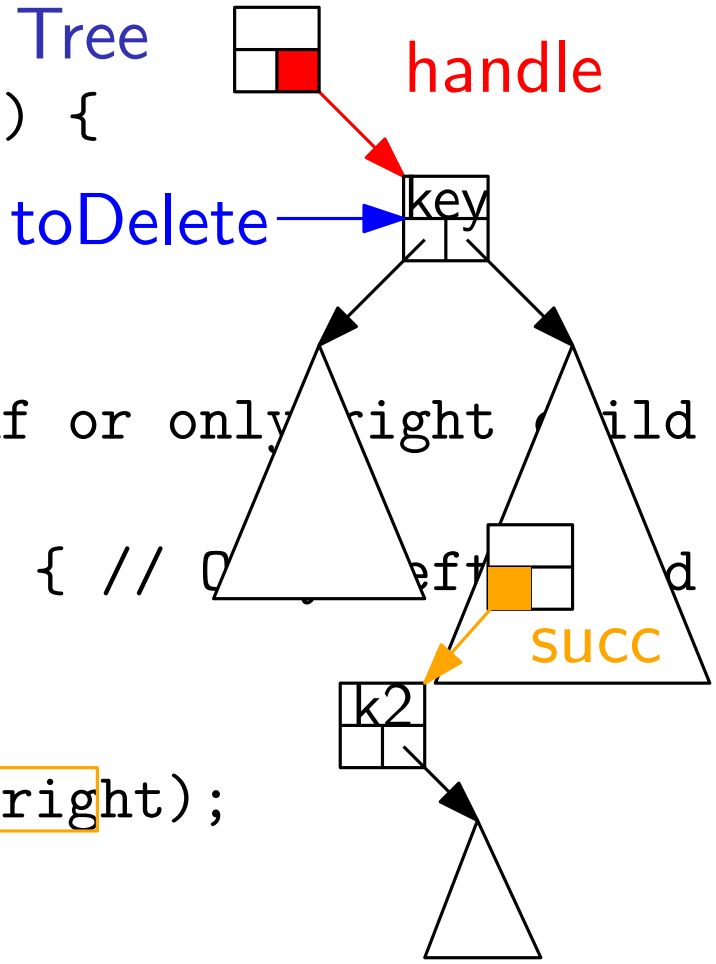
Remove a node from a Binary Search Tree




```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```

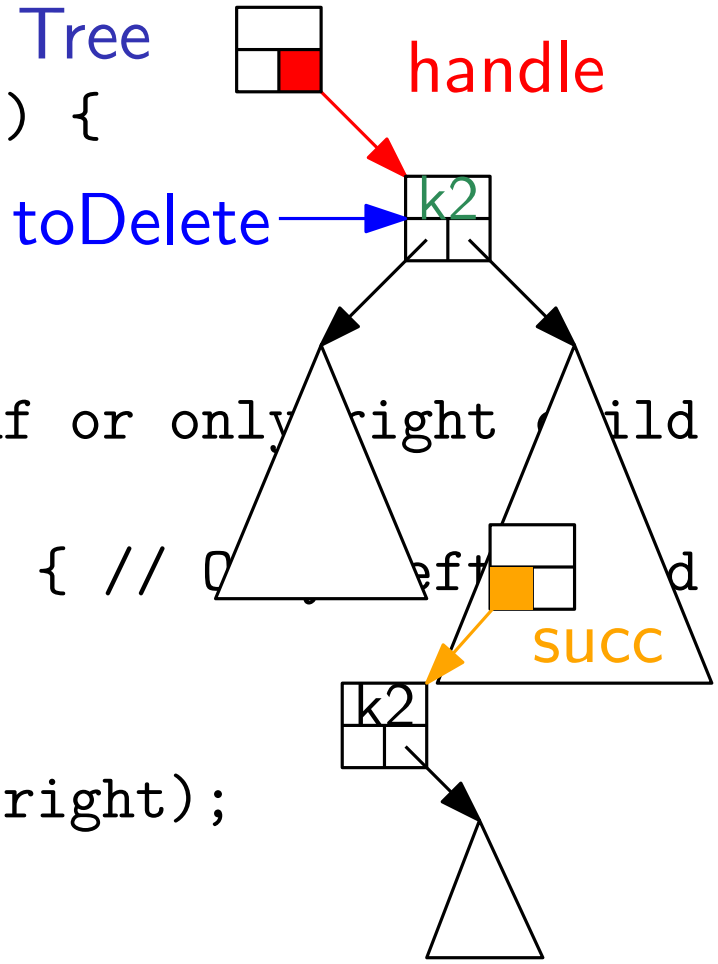
Remove a node from a Binary Search Tree

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```



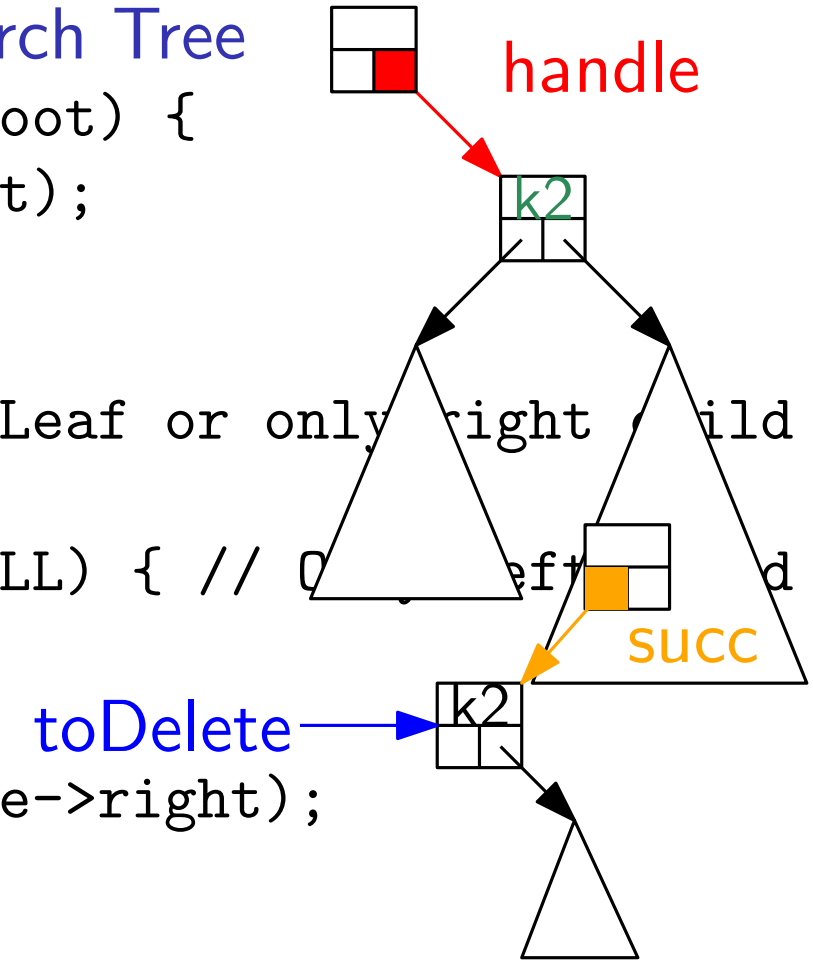
Remove a node from a Binary Search Tree

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```



Remove a node from a Binary Search Tree

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```



Remove a node from a Binary Search Tree

```
void remove(KType key, Node *& root) {
  Node *& handle = find(key, root);
  if (handle == NULL) return;
  Node * toDelete = handle;
  if (handle->left == NULL) {// Leaf or only right child
    handle = handle->right;
  } else if (handle->right == NULL) { // Only left child
    handle = handle->left;
  } else {  // Two children
    Node *& succ = findMin(handle->right);
    handle->key = succ->key;
    toDelete = succ;
    succ = succ->right;          // succ has no left child
  }
  delete toDelete;
}
```