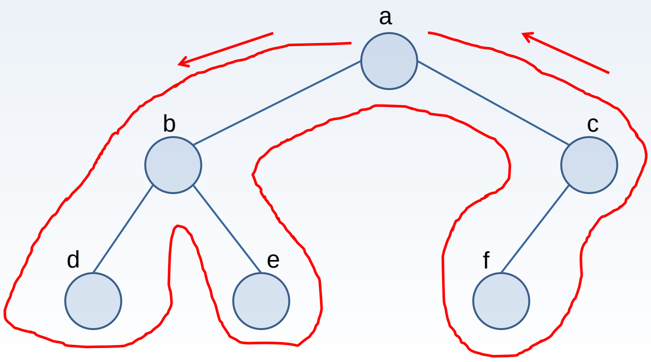
Data Structures

Trees

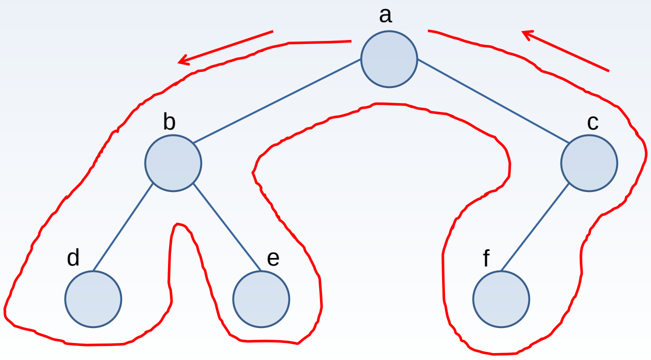
Traversal Orders

- Imagine having a string shrink-wrapped around the tree.
- If we start at the root node and follow the string around the tree, we are traversing the tree.
- In what follows, we will always go around the tree counterclockwise.



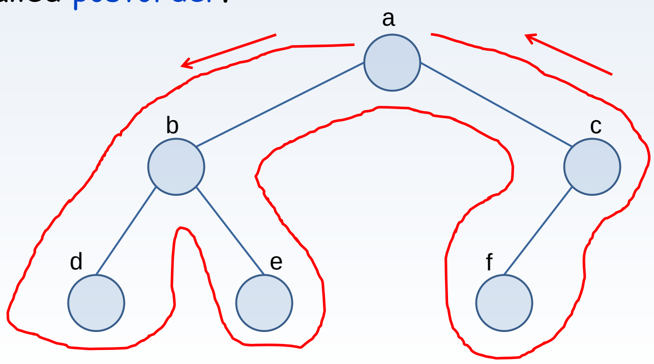
Traversal Orders

- When we traverse the tree, each node is visited three times: on the left, underneath, and on the right.
  - To picture this for nodes with one or no children, just attach imaginary children.

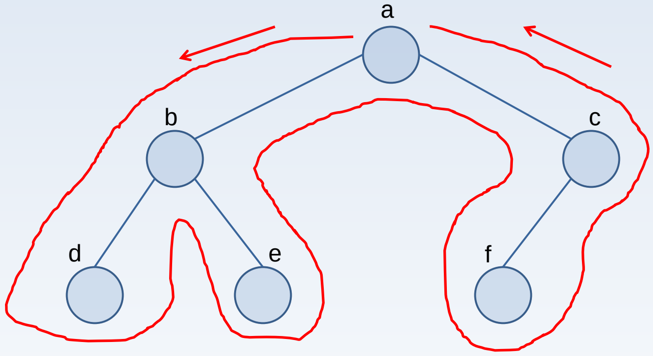


Traversal Orders

- The order of the nodes as they are passed on the left is called preorder.
- The order of the nodes as they are passed underneath is called inorder.
- The order of the nodes as they are passed on the right is called postorder.



Traversal Orders: Example



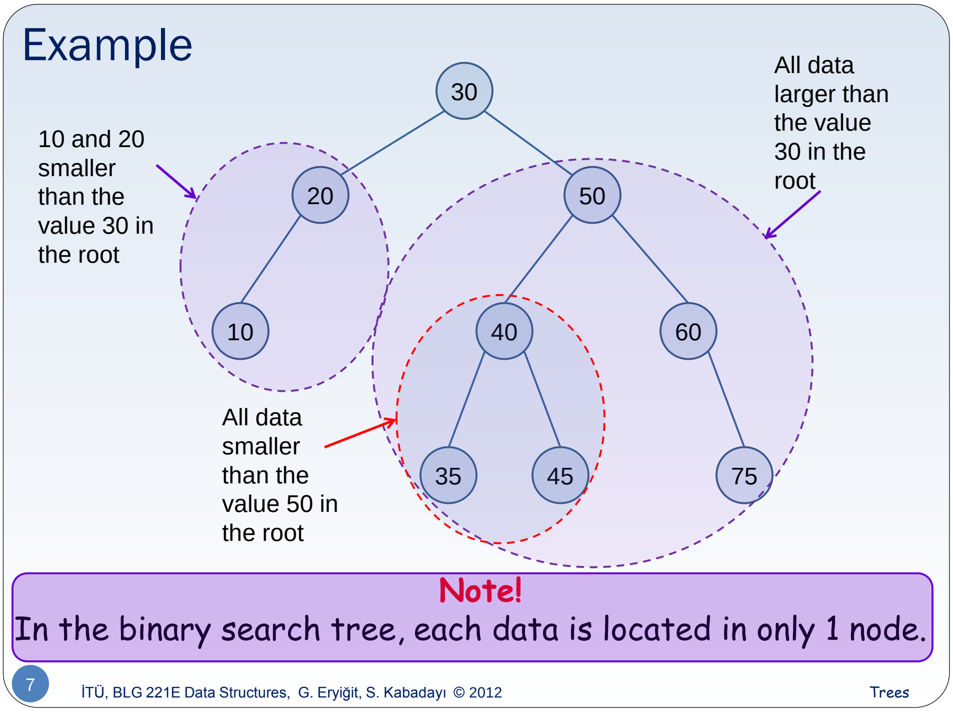
- Preorder: a b d e c f
- Inorder: d b e a f c
- Postorder: d e b f c a

Binary Search Tree

- Trees are generally used in index structures.
- To use them for this purpose, data should be placed in the tree in a certain order.
- The simplest ordering is the binary search tree.

Rule:

- A root node
  - values in the left subtree should be smaller than the value in the root
  - values in the right subtree should be larger than the value in the root.
- The right and left subtrees of a root node have the search tree property.



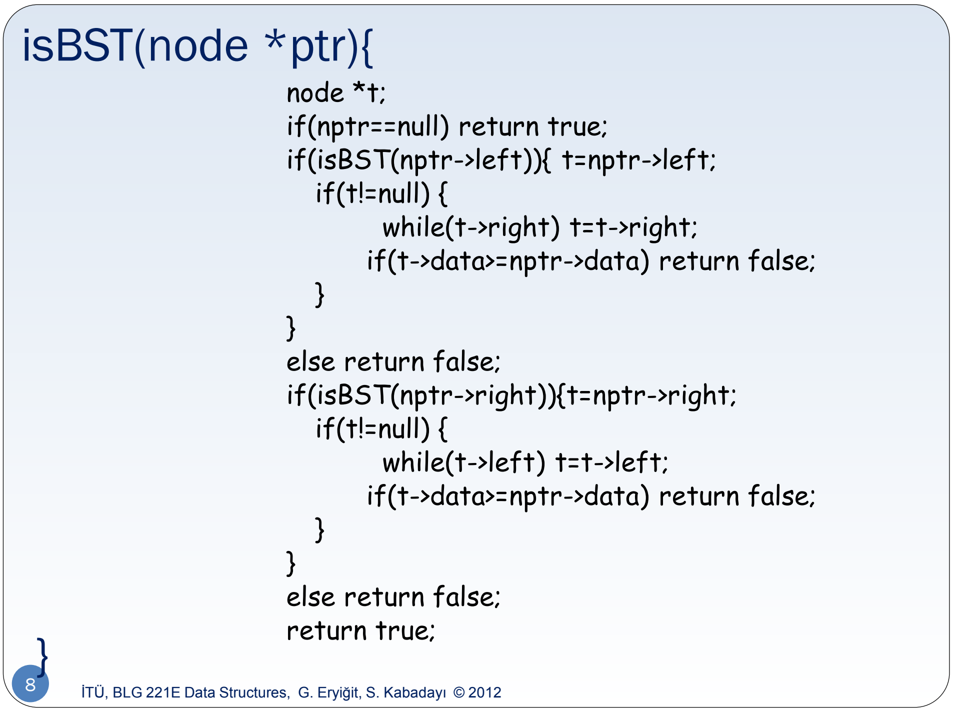
Example

10 and 20 smaller than the value 30 in the root

All data larger than the value 30 in the root

All data smaller than the value 50 in the root

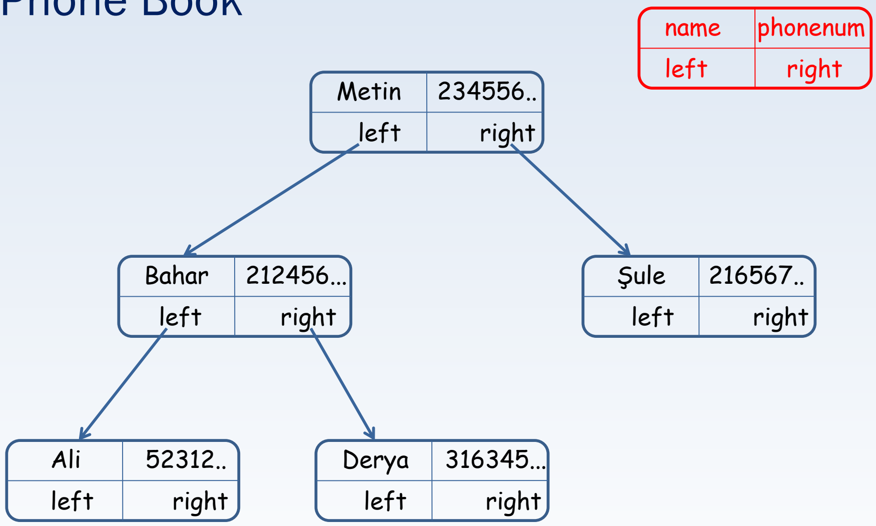
Note!
In the binary search tree, each data is located in only 1 node.



isBST(node *ptr){

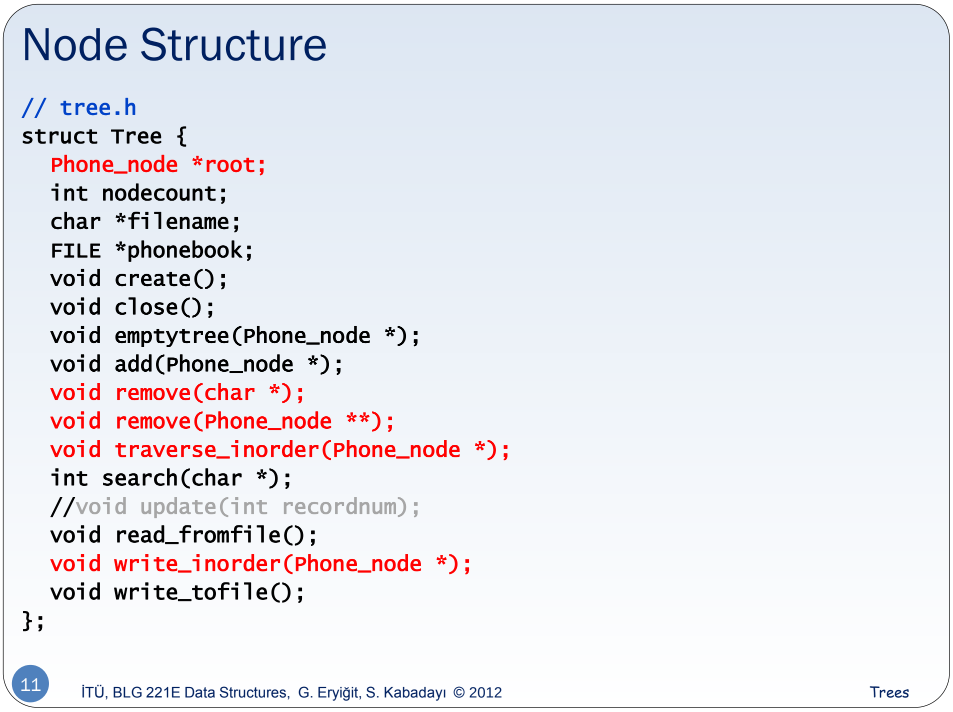
```
node *t;
if(nptr==null) return true;
if(isBST(nptr->left)){ t=nptr->left;
   if(t!=null) {
          while(t->right) t=t->right;
        if(t->data>=nptr->data) return false;
   }
}
else return false;
if(isBST(nptr->right)){t=nptr->right;
   if(t!=null) {
          while(t->left) t=t->left;
        if(t->data>=nptr->data) return false;
   }
}
else return false;
return true;
}
```

Phone Book



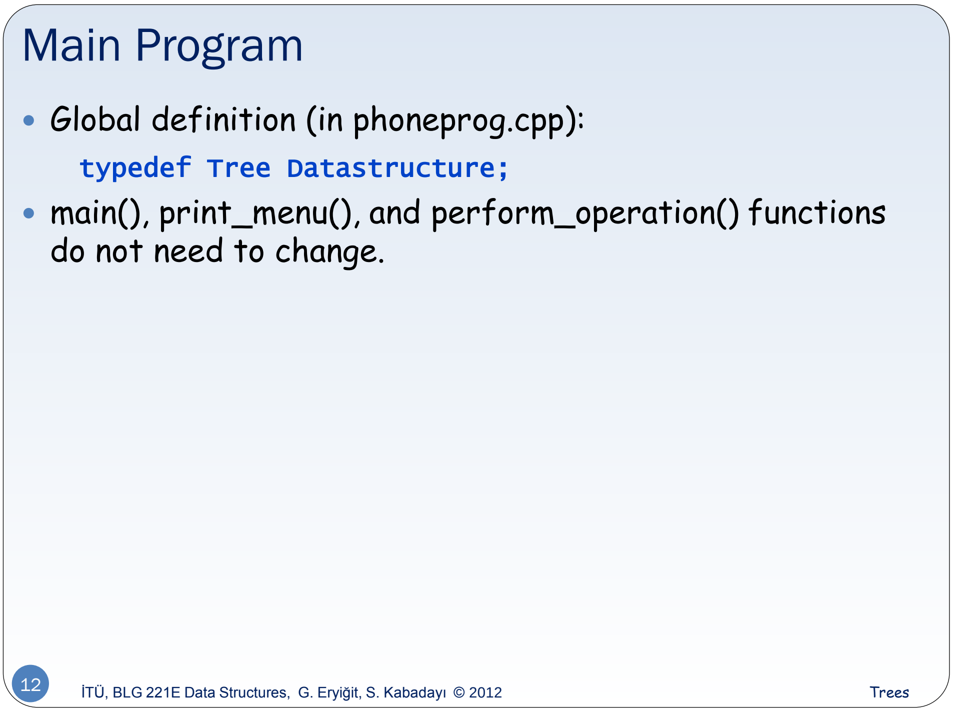
Node Structure

```
// node.h
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15

struct Phone_node{
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
    Phone_node *left;
    Phone_node *right;
};
```



Node Structure

```
// tree.h
struct Tree {
   Phone_node *root;
   int nodecount;
   char *filename;
   FILE *phonebook;
   void create();
   void close();
   void emptytree(Phone_node *);
   void add(Phone_node *);
   void remove(char *);
   void remove(Phone_node **);
   void traverse_inorder(Phone_node *);
   int search(char *);
   //void update(int recordnum);
   void read_fromfile();
   void write_inorder(Phone_node *);
   void write_tofile();
};
```

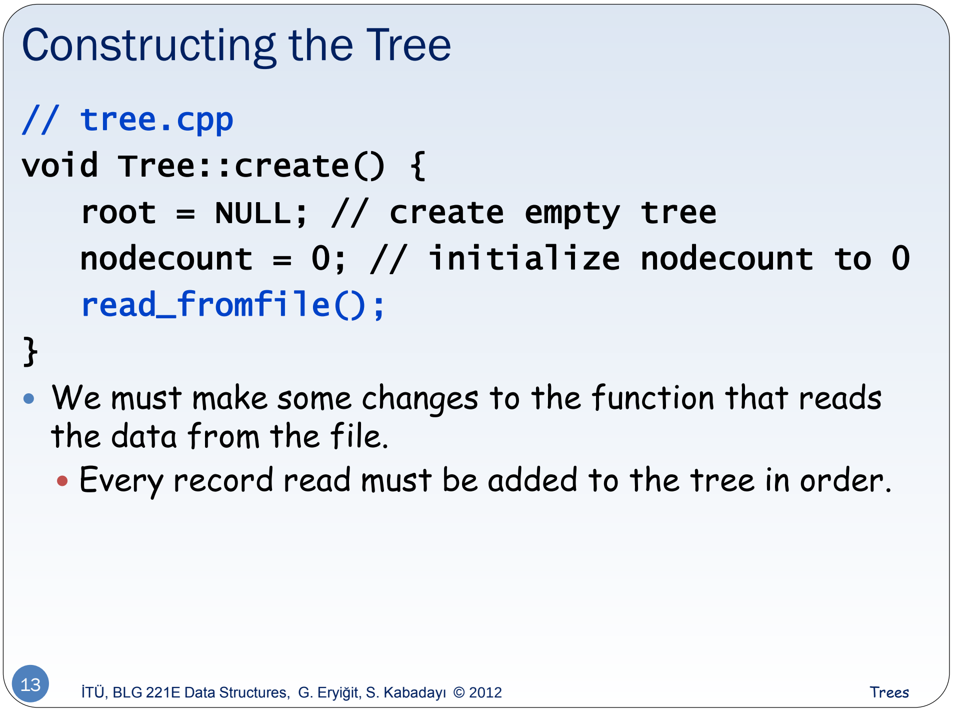


Main Program

- Global definition (in phoneprog.cpp):

```
typedef Tree Datastructure;
```

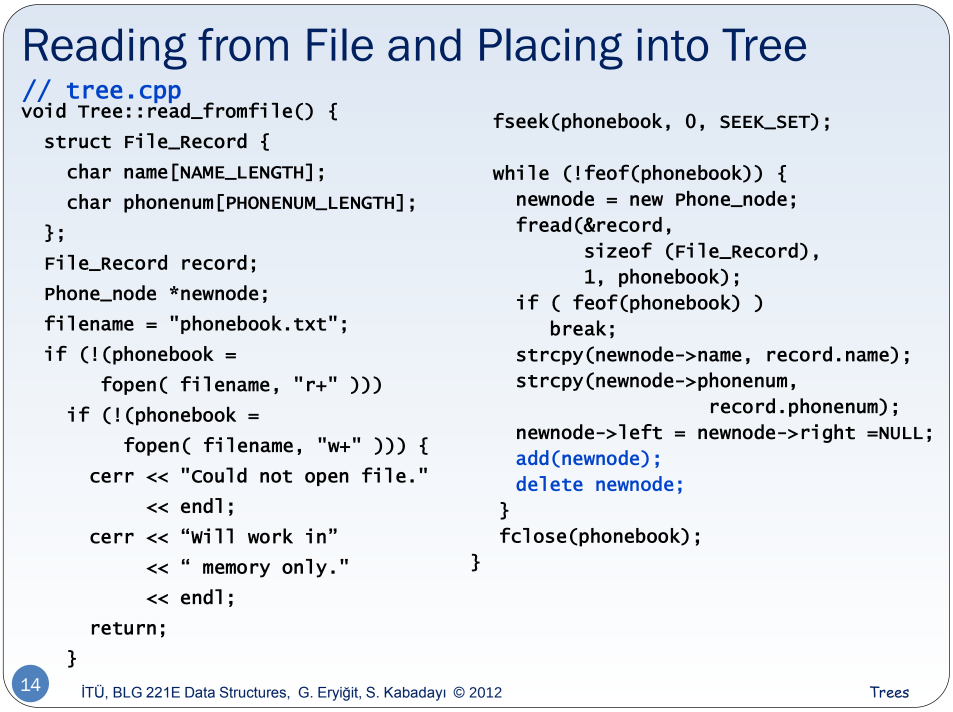
- main(), print_menu(), and perform_operation() functions do not need to change.



Constructing the Tree

```
// tree.cpp
void Tree::create() {
   root = NULL; // create empty tree
   nodecount = 0; // initialize nodecount to 0
   read_fromfile();
}
```

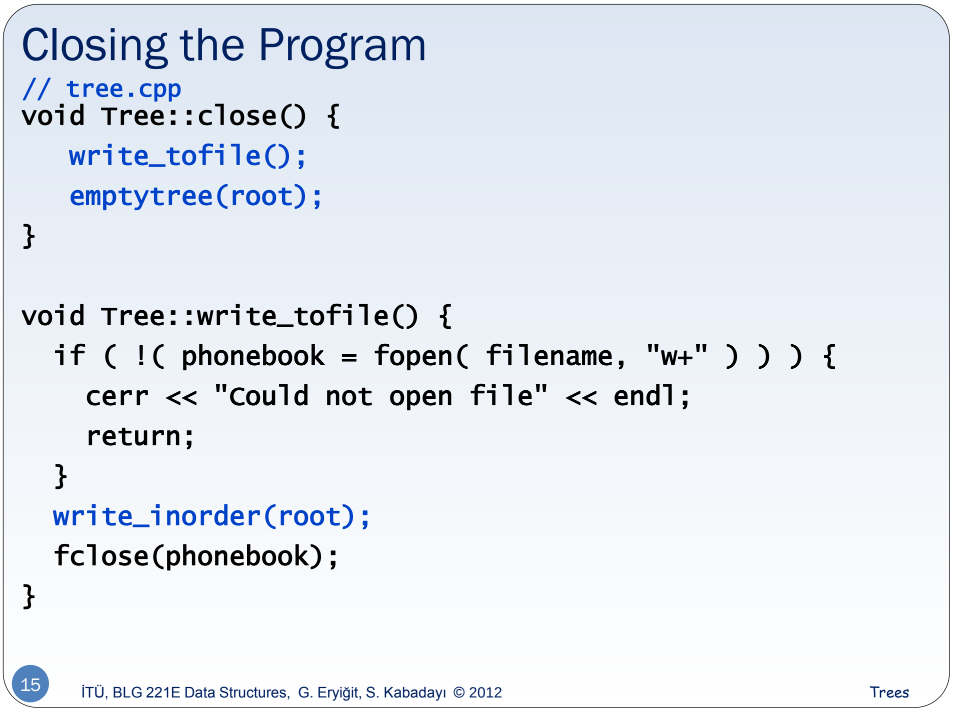
- We must make some changes to the function that reads the data from the file.
  - Every record read must be added to the tree in order.



Reading from File and Placing into Tree

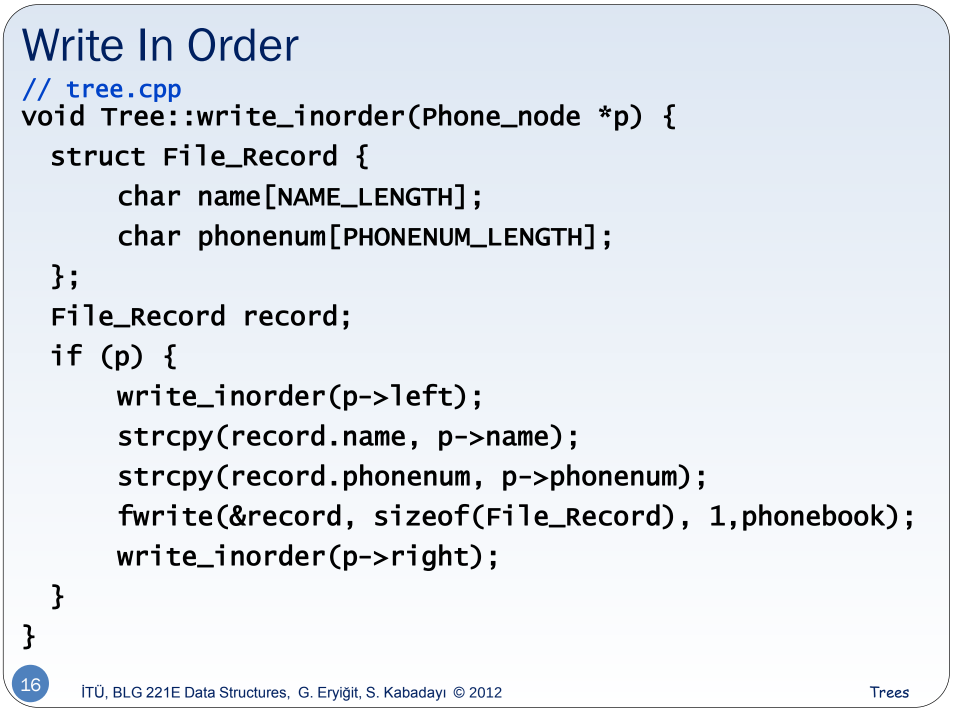
```
// tree.cpp
void Tree::read_fromfile() {
  struct File_Record {
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
  };
  File_Record record;
  Phone_node *newnode;
  filename = "phonebook.txt";
  if (!(phonebook =
       fopen( filename, "r+" )))
    if (!(phonebook =
         fopen( filename, "w+" ))) {
      cerr << "Could not open file."
           << endl;
      cerr << “Will work in”
           << “ memory only."
           << endl;
      return;
    }
```

```
  fseek(phonebook, 0, SEEK_SET);

  while (!feof(phonebook)) {
    newnode = new Phone_node;
    fread(&record,
          sizeof (File_Record),
          1, phonebook);
    if ( feof(phonebook) )
       break;
    strcpy(newnode->name, record.name);
    strcpy(newnode->phonenum,
                 record.phonenum);
    newnode->left = newnode->right =NULL;
    add(newnode);
    delete newnode;
   }
   fclose(phonebook);
}
```



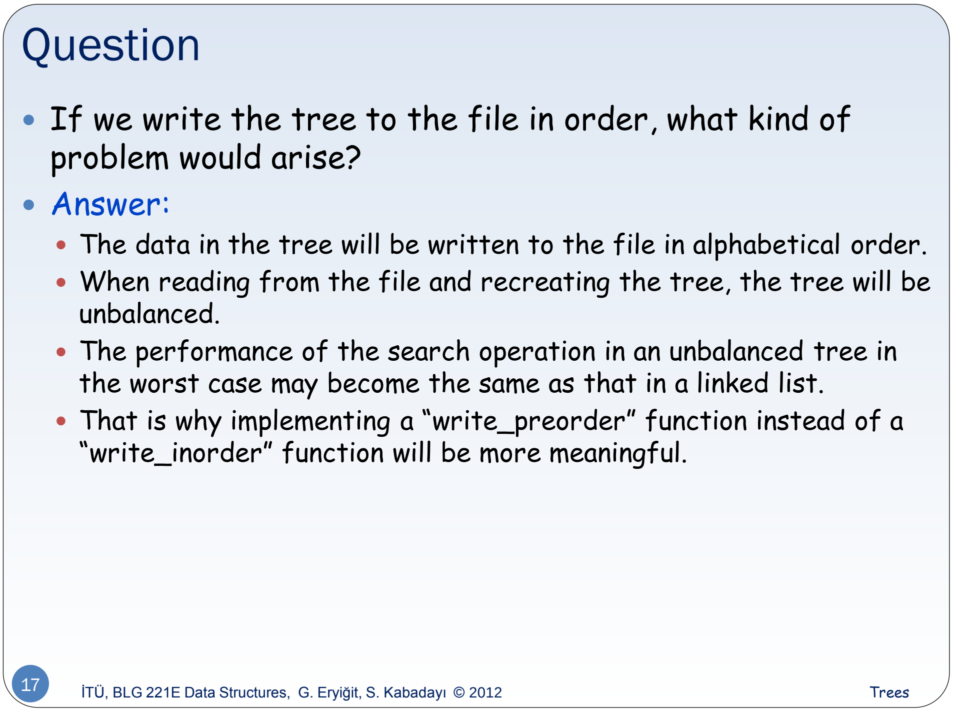
Closing the Program

```
// tree.cpp
void Tree::close() {
   write_tofile();
   emptytree(root);
}

void Tree::write_tofile() {
  if ( !( phonebook = fopen( filename, "w+" ) ) ) {
    cerr << "Could not open file" << endl;
    return;
  }
  write_inorder(root);
  fclose(phonebook);
}
```



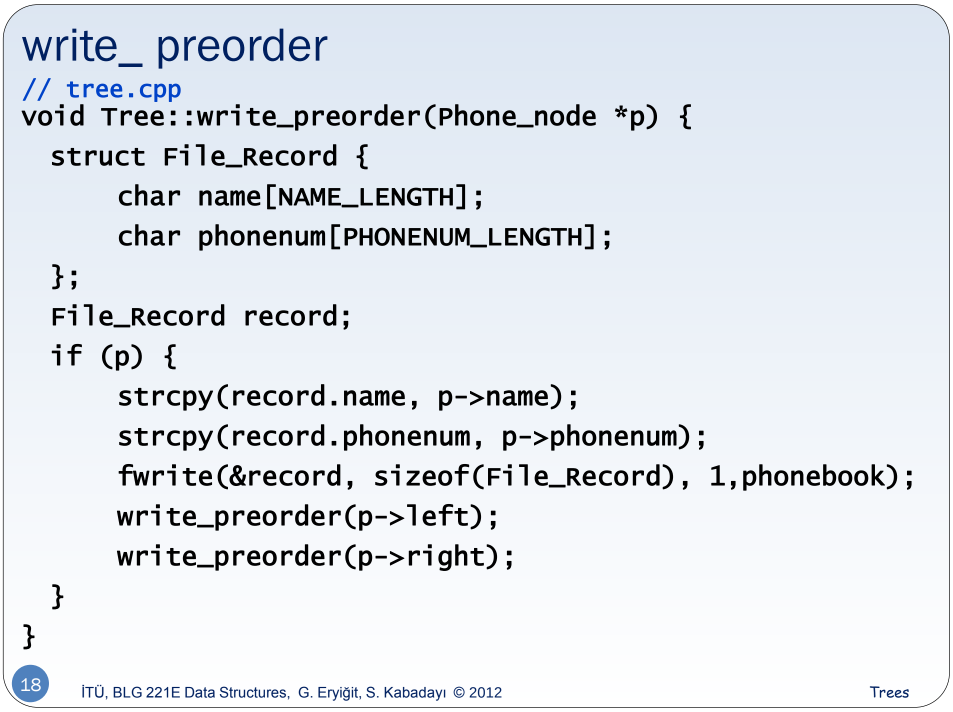
Write In Order

```
// tree.cpp
void Tree::write_inorder(Phone_node *p) {
  struct File_Record {
      char name[NAME_LENGTH];
      char phonenum[PHONENUM_LENGTH];
  };
  File_Record record;
  if (p) {
      write_inorder(p->left);
      strcpy(record.name, p->name);
      strcpy(record.phonenum, p->phonenum);
      fwrite(&record, sizeof(File_Record), 1,phonebook);
      write_inorder(p->right);
  }
}
```



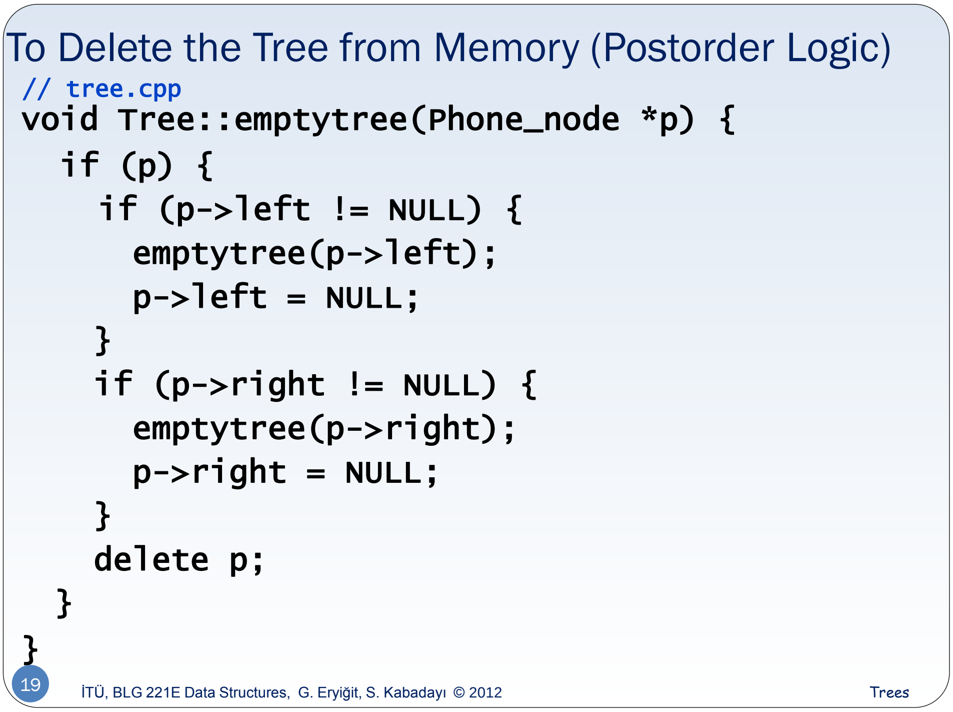
Question

- If we write the tree to the file in order, what kind of problem would arise?
- *Answer:*
  - The data in the tree will be written to the file in alphabetical order.
  - When reading from the file and recreating the tree, the tree will be unbalanced.
  - The performance of the search operation in an unbalanced tree in the worst case may become the same as that in a linked list.
  - That is why implementing a "write_preorder" function instead of a "write_inorder" function will be more meaningful.



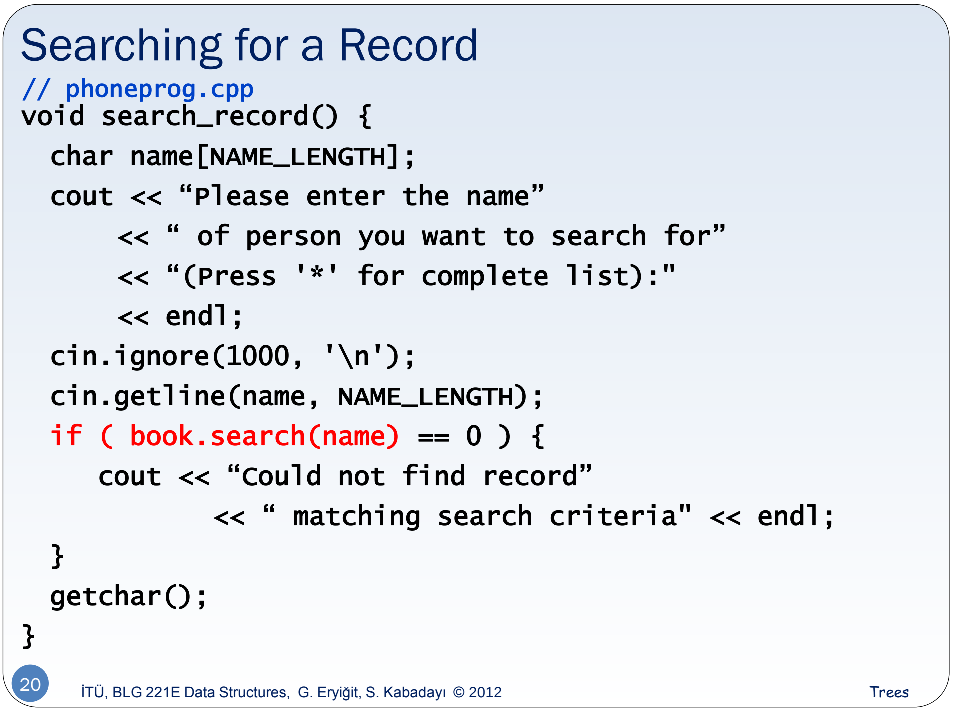
write_ preorder

```
// tree.cpp
void Tree::write_preorder(Phone_node *p) {
  struct File_Record {
       char name[NAME_LENGTH];
       char phonenum[PHONENUM_LENGTH];
  };
  File_Record record;
  if (p) {
       strcpy(record.name, p->name);
       strcpy(record.phonenum, p->phonenum);
       fwrite(&record, sizeof(File_Record), 1,phonebook);
       write_preorder(p->left);
       write_preorder(p->right);
  }
}
```



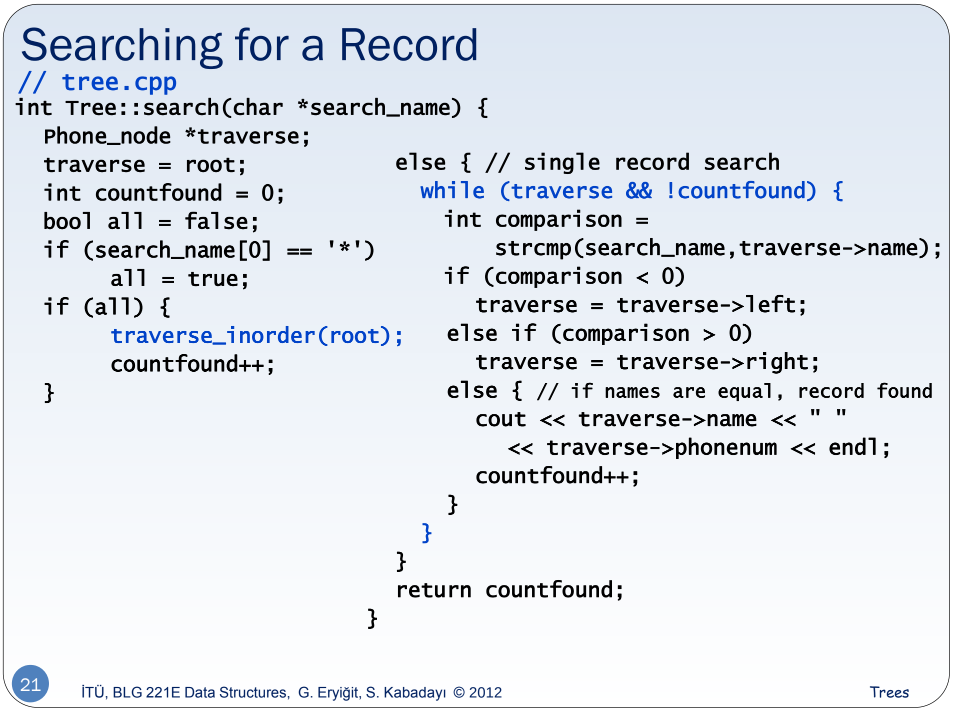
To Delete the Tree from Memory (Postorder Logic)

```
// tree.cpp
void Tree::emptytree(Phone_node *p) {
  if (p) {
    if (p->left != NULL) {
      emptytree(p->left);
      p->left = NULL;
    }
    if (p->right != NULL) {
      emptytree(p->right);
      p->right = NULL;
    }
    delete p;
  }
}
```



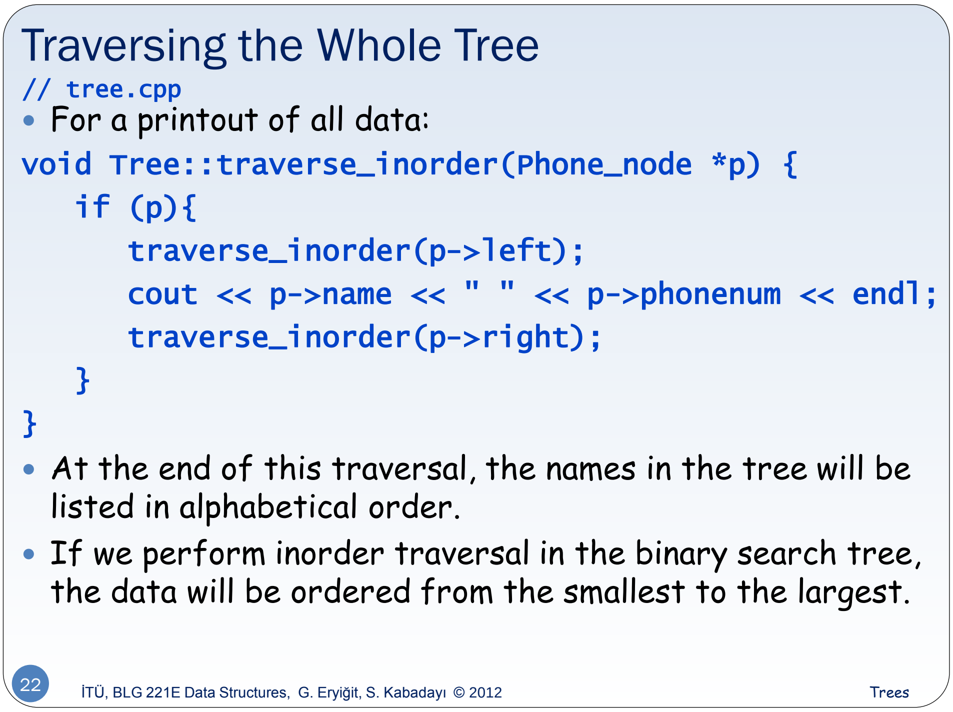
Searching for a Record

```
// phoneprog.cpp
void search_record() {
  char name[NAME_LENGTH];
  cout << “Please enter the name”
      << “ of person you want to search for”
      << “(Press '*' for complete list):"
      << endl;
  cin.ignore(1000, '\n');
  cin.getline(name, NAME_LENGTH);
  if ( book.search(name) == 0 ) {
     cout << “Could not find record”
            << “ matching search criteria" << endl;
  }
  getchar();
}
```



Searching for a Record

```
// tree.cpp
int Tree::search(char *search_name) {
  Phone_node *traverse;
  traverse = root;
  int countfound = 0;
  bool all = false;
  if (search_name[0] == '*')
        all = true;
  if (all) {
        traverse_inorder(root);
        countfound++;
  }
  else { // single record search
    while (traverse && !countfound) {
      int comparison =
          strcmp(search_name,traverse->name);
      if (comparison < 0)
        traverse = traverse->left;
      else if (comparison > 0)
        traverse = traverse->right;
      else { // if names are equal, record found
        cout << traverse->name << " "
           << traverse->phonenum << endl;
        countfound++;
      }
    }
  }
  return countfound;
}
```



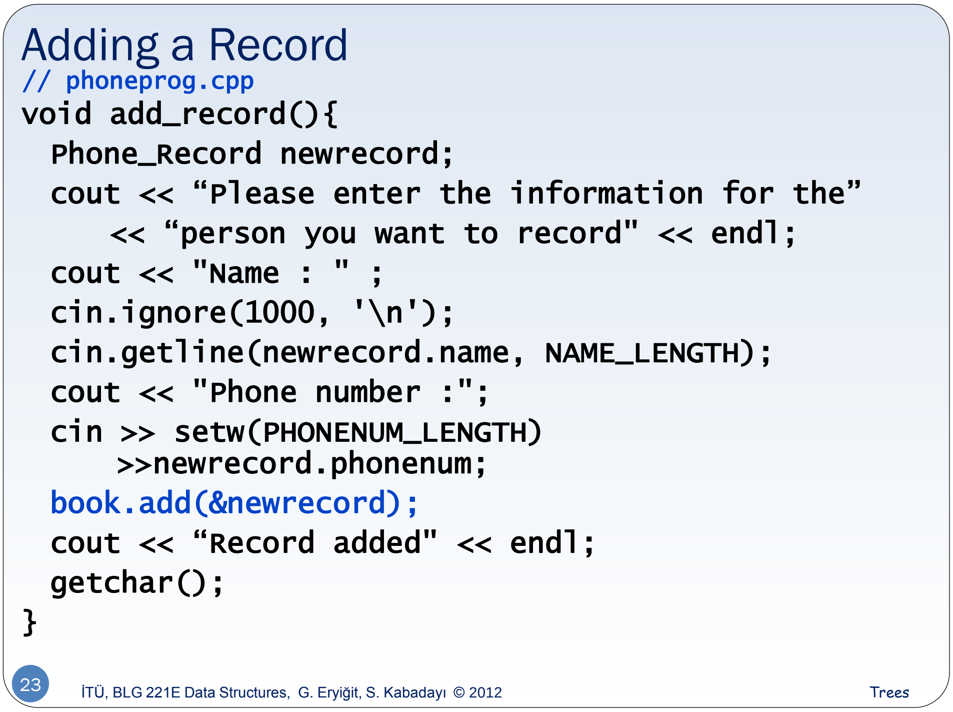
Traversing the Whole Tree

```
// tree.cpp
```

- For a printout of all data:

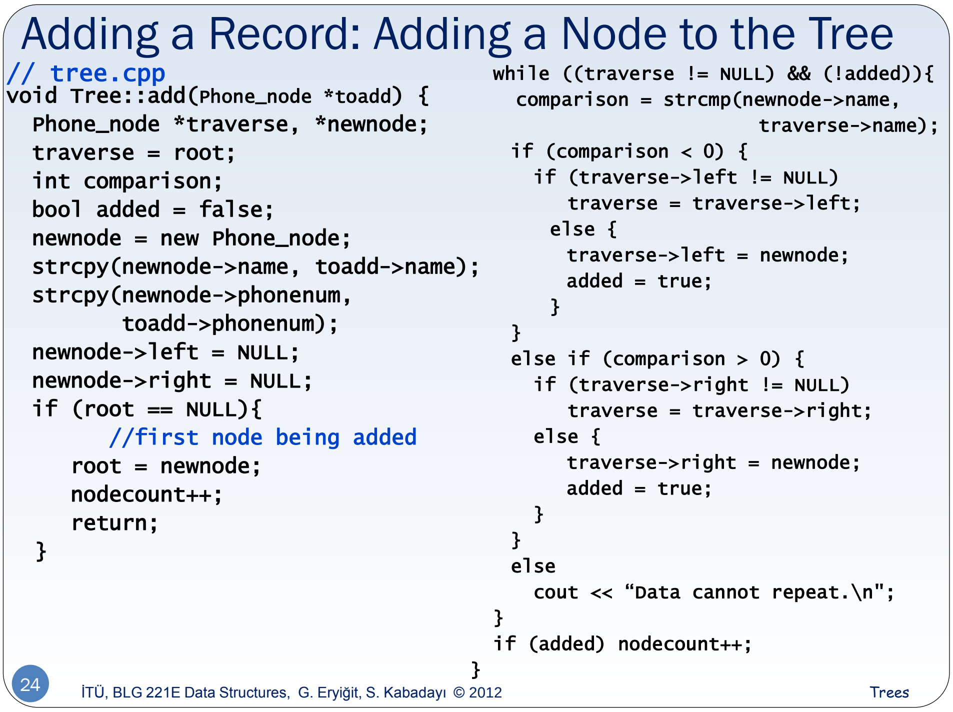
```
void Tree::traverse_inorder(Phone_node *p) {
   if (p){
      traverse_inorder(p->left);
      cout << p->name << " " << p->phonenum << endl;
      traverse_inorder(p->right);
   }
}
```

- At the end of this traversal, the names in the tree will be listed in alphabetical order.
- If we perform inorder traversal in the binary search tree, the data will be ordered from the smallest to the largest.



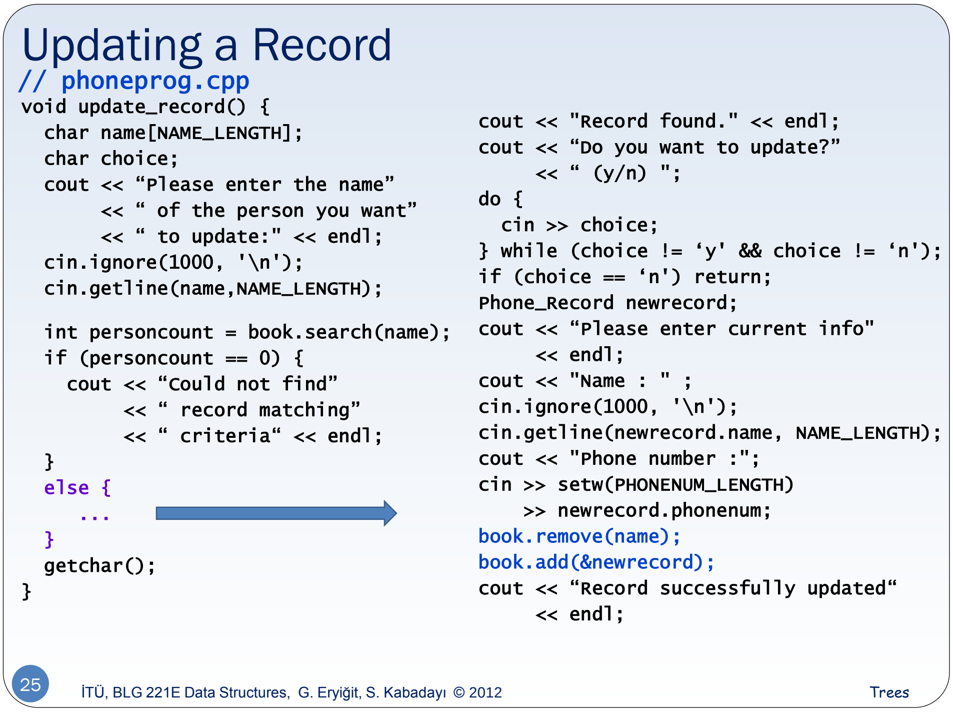
Adding a Record

```
// phoneprog.cpp
void add_record(){
  Phone_Record newrecord;
  cout << “Please enter the information for the”
      << “person you want to record" << endl;
  cout << "Name : " ;
  cin.ignore(1000, '\n');
  cin.getline(newrecord.name, NAME_LENGTH);
  cout << "Phone number :";
  cin >> setw(PHONENUM_LENGTH)
      >>newrecord.phonenum;
  book.add(&newrecord);
  cout << “Record added" << endl;
  getchar();
}
```



Adding a Record: Adding a Node to the Tree

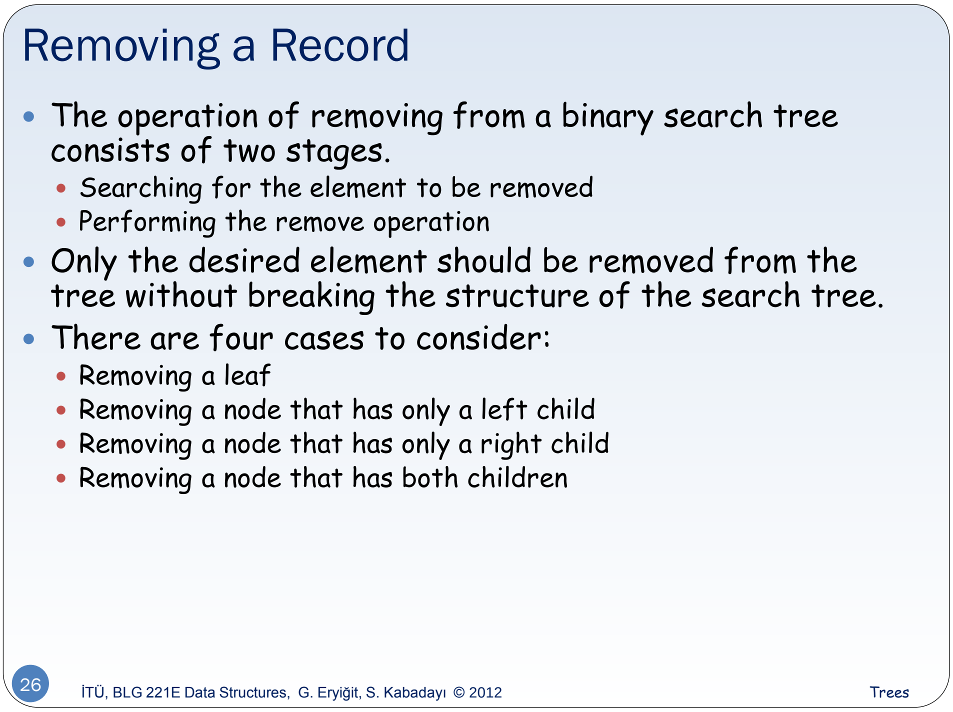
```
// tree.cpp
void Tree::add(Phone_node *toadd) {
  Phone_node *traverse, *newnode;
  traverse = root;
  int comparison;
  bool added = false;
  newnode = new Phone_node;
  strcpy(newnode->name, toadd->name);
  strcpy(newnode->phonenum,
         toadd->phonenum);
  newnode->left = NULL;
  newnode->right = NULL;
  if (root == NULL){
       //first node being added
     root = newnode;
     nodecount++;
     return;
  }
  while ((traverse != NULL) && (!added)){
    comparison = strcmp(newnode->name,
                        traverse->name);
    if (comparison < 0) {
      if (traverse->left != NULL)
         traverse = traverse->left;
       else {
         traverse->left = newnode;
         added = true;
       }
    }
    else if (comparison > 0) {
      if (traverse->right != NULL)
         traverse = traverse->right;
      else {
         traverse->right = newnode;
         added = true;
      }
    }
    else
      cout << “Data cannot repeat.\n";
  }
  if (added) nodecount++;
}
```



Updating a Record

```
// phoneprog.cpp
void update_record() {
  char name[NAME_LENGTH];
  char choice;
  cout << “Please enter the name”
       << “ of the person you want”
       << “ to update:" << endl;
  cin.ignore(1000, '\n');
  cin.getline(name,NAME_LENGTH);

  int personcount = book.search(name);
  if (personcount == 0) {
    cout << “Could not find”
         << “ record matching”
         << “ criteria“ << endl;
  }
  else {
     ...
  }
  getchar();
}
```

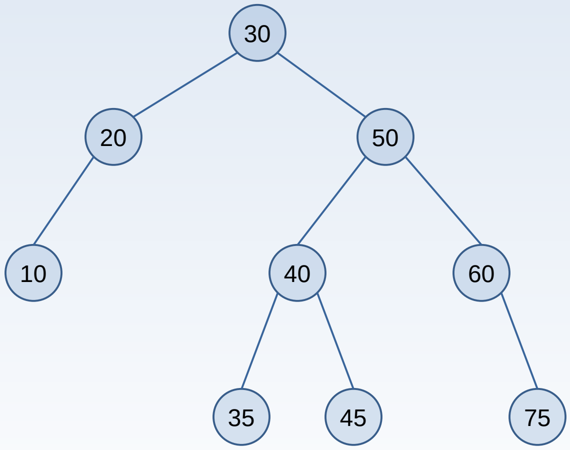
```
cout << "Record found." << endl;
cout << “Do you want to update?”
     << “ (y/n) ";
do {
  cin >> choice;
} while (choice != ‘y' && choice != ‘n');
if (choice == ‘n') return;
Phone_Record newrecord;
cout << “Please enter current info"
     << endl;
cout << "Name : " ;
cin.ignore(1000, '\n');
cin.getline(newrecord.name, NAME_LENGTH);
cout << "Phone number :";
cin >> setw(PHONENUM_LENGTH)
    >> newrecord.phonenum;
book.remove(name);
book.add(&newrecord);
cout << “Record successfully updated“
     << endl;
```



Removing a Record

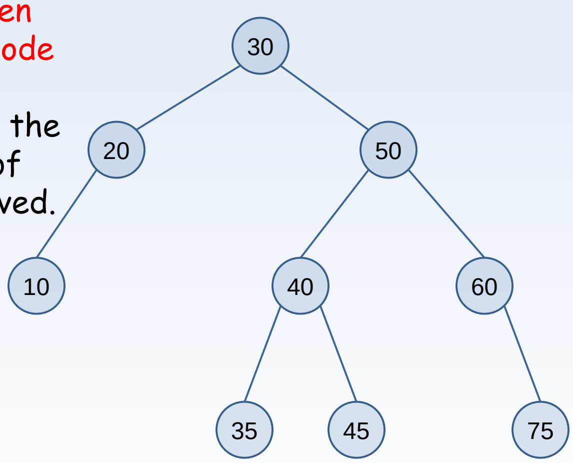
- The operation of removing from a binary search tree consists of two stages.
  - Searching for the element to be removed
  - Performing the remove operation
- Only the desired element should be removed from the tree without breaking the structure of the search tree.
- There are four cases to consider:
  - Removing a leaf
  - Removing a node that has only a left child
  - Removing a node that has only a right child
  - Removing a node that has both children

Removing a Leaf



Removing a Node that Has a Single Child

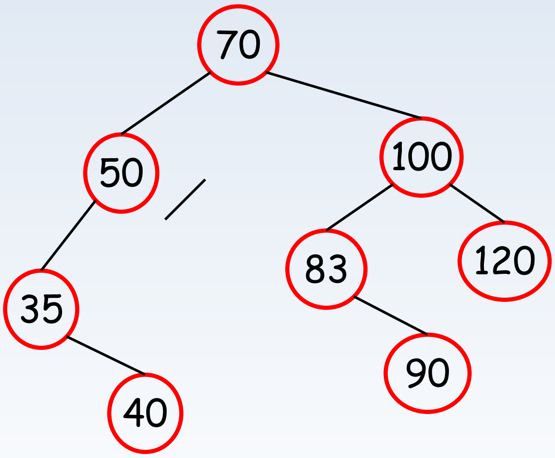
- If the node has a single child, the link between the parent and the node to be removed is established between the parent and subtree of the node to be removed.

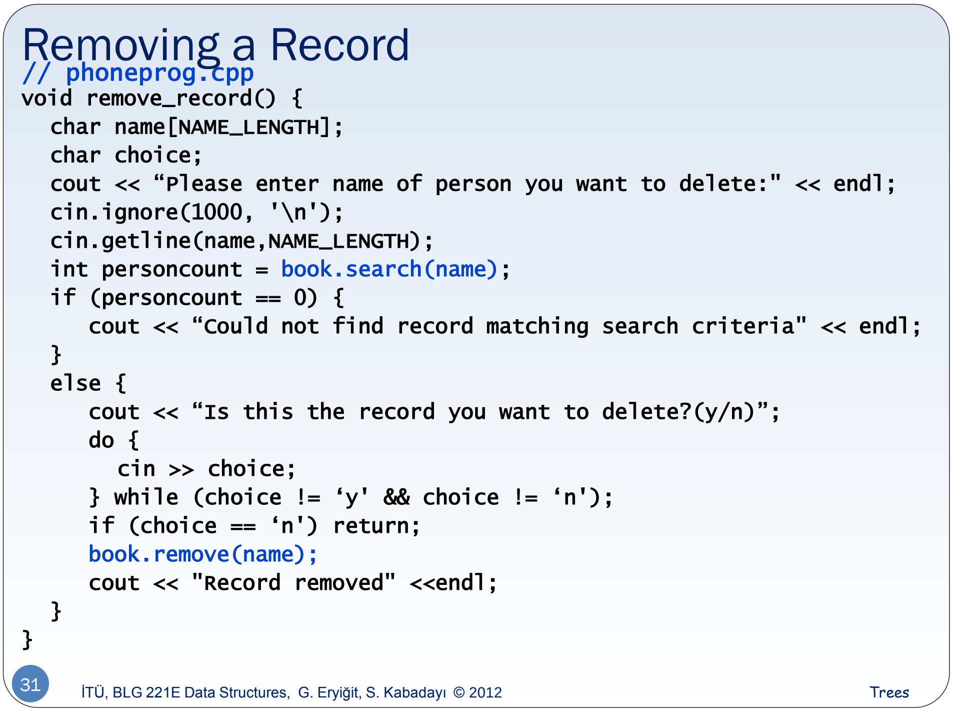


Removing a Node That Has Both Children

- We move the right subtree to where the removed node used to be, and
- We try to link the left subtree to an approriate node in the right subtree.
  - We must link the left subtree to the left of the leftmost child of the right subtree.

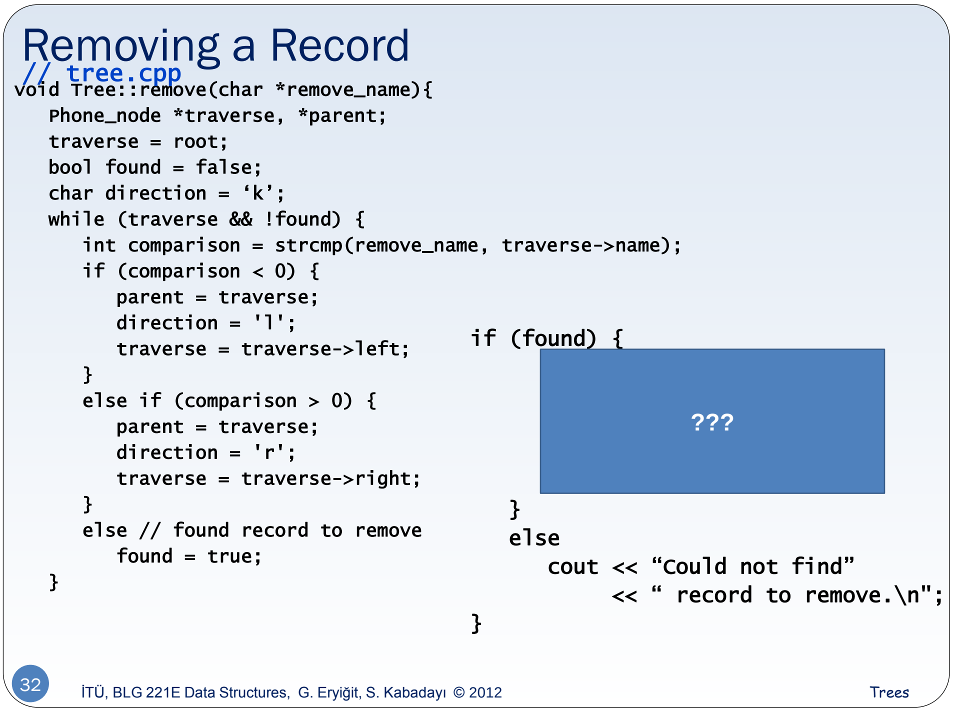
Removing a Node That Has Both Children: Example





Removing a Record

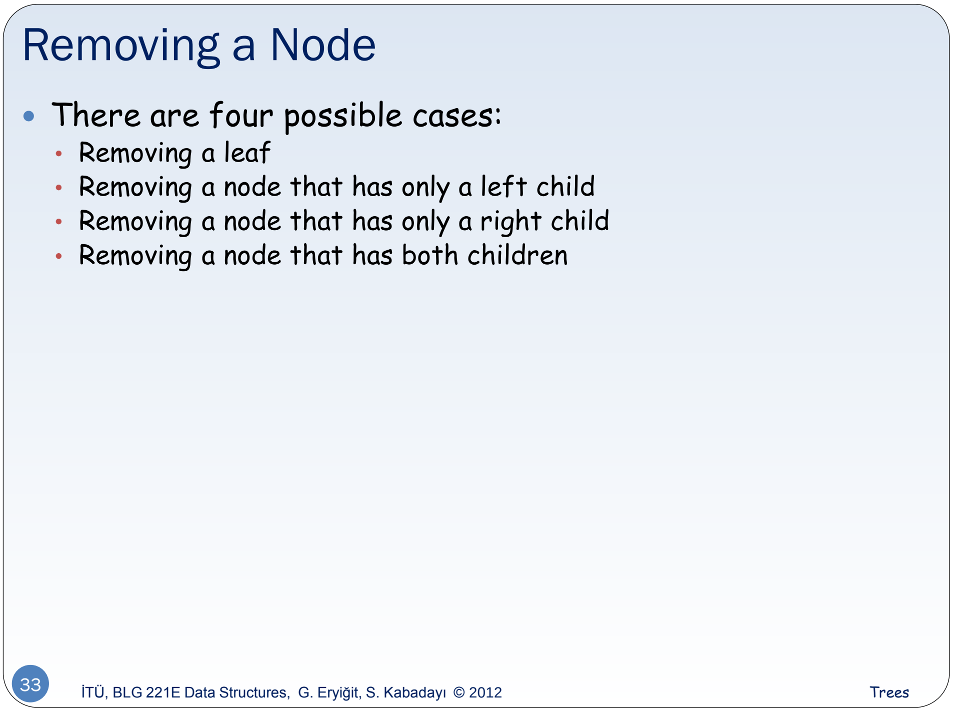
```
// phoneprog.cpp
void remove_record() {
   char name[NAME_LENGTH];
   char choice;
   cout << “Please enter name of person you want to delete:" << endl;
   cin.ignore(1000, '\n');
   cin.getline(name,NAME_LENGTH);
   int personcount = book.search(name);
   if (personcount == 0) {
       cout << “Could not find record matching search criteria" << endl;
   }
   else {
       cout << “Is this the record you want to delete?(y/n)”;
       do {
          cin >> choice;
       } while (choice != ‘y' && choice != ‘n');
       if (choice == ‘n') return;
       book.remove(name);
       cout << "Record removed" <<endl;
   }
}
```



Removing a Record

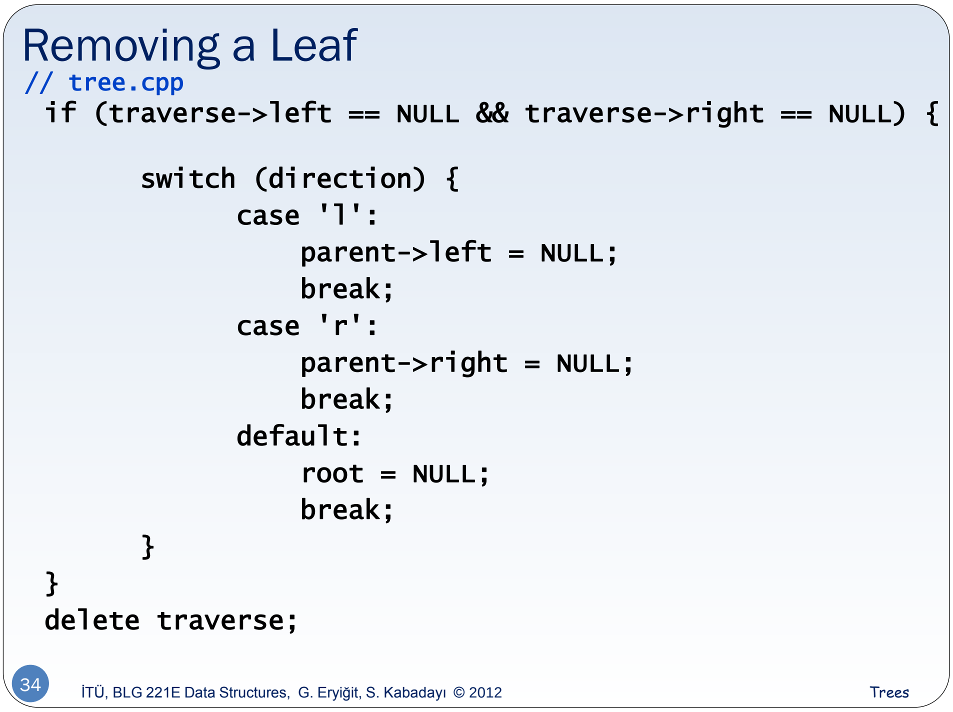
```
// tree.cpp
void Tree::remove(char *remove_name){
   Phone_node *traverse, *parent;
   traverse = root;
   bool found = false;
   char direction = ‘k’;
   while (traverse && !found) {
      int comparison = strcmp(remove_name, traverse->name);
      if (comparison < 0) {
         parent = traverse;
         direction = 'l';
         traverse = traverse->left;
      }
      else if (comparison > 0) {
         parent = traverse;
         direction = 'r';
         traverse = traverse->right;
      }
      else // found record to remove
         found = true;
   }
```

```
if (found) {

      ???

   }
   else
      cout << “Could not find”
           << “ record to remove.\n";
}
```



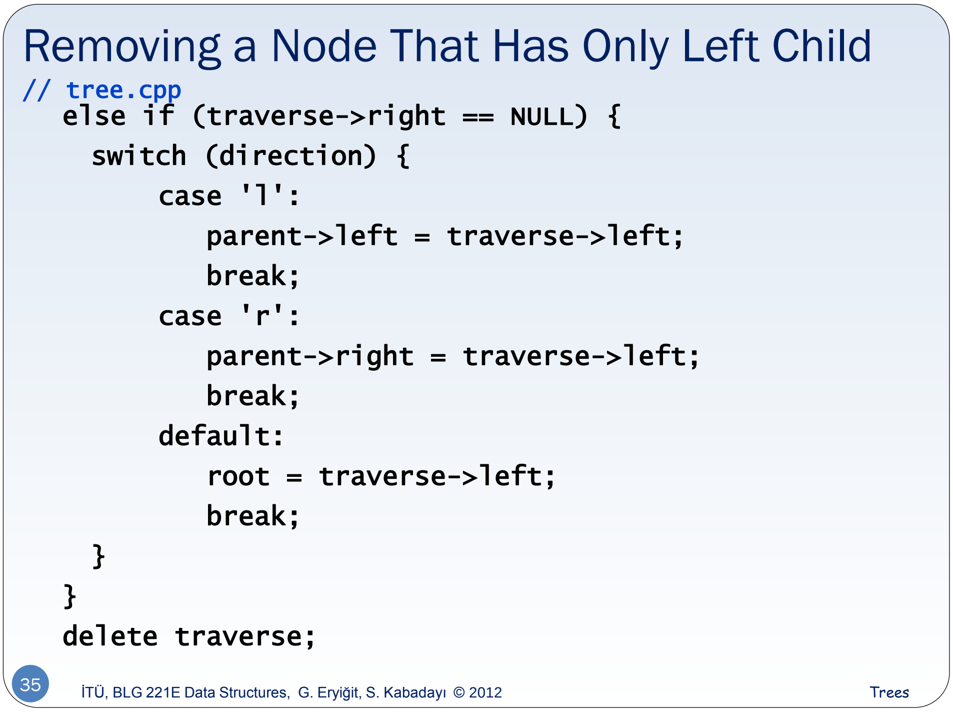
Removing a Node

- There are four possible cases:
  - Removing a leaf
  - Removing a node that has only a left child
  - Removing a node that has only a right child
  - Removing a node that has both children



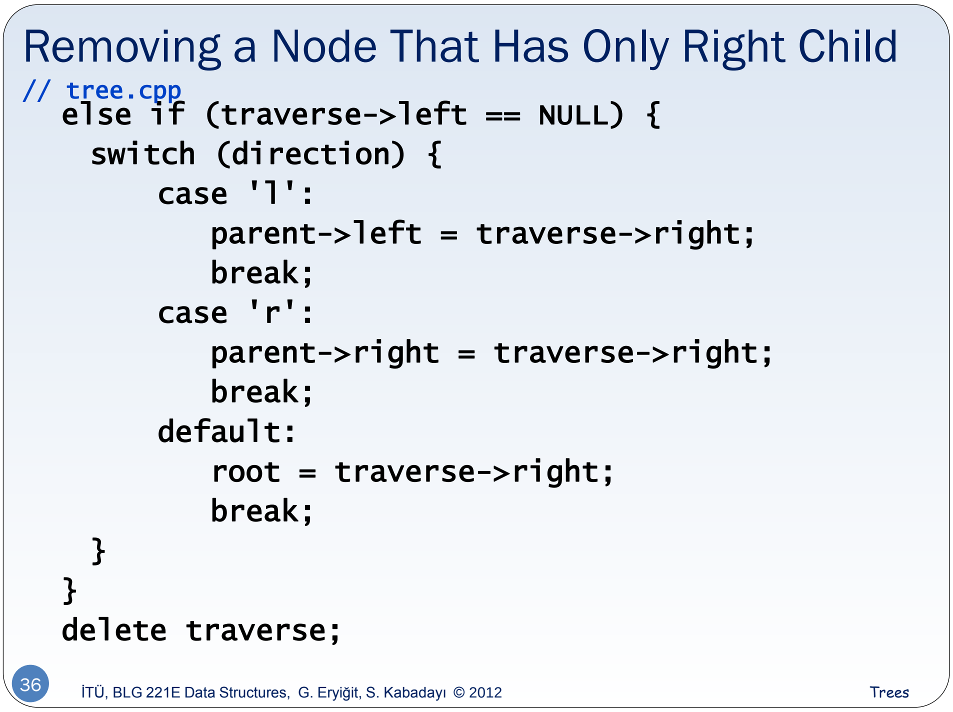
Removing a Leaf

```
// tree.cpp
 if (traverse->left == NULL && traverse->right == NULL) {

       switch (direction) {
             case 'l':
                 parent->left = NULL;
                 break;
             case 'r':
                 parent->right = NULL;
                 break;
             default:
                 root = NULL;
                 break;
       }
 }
 delete traverse;
```



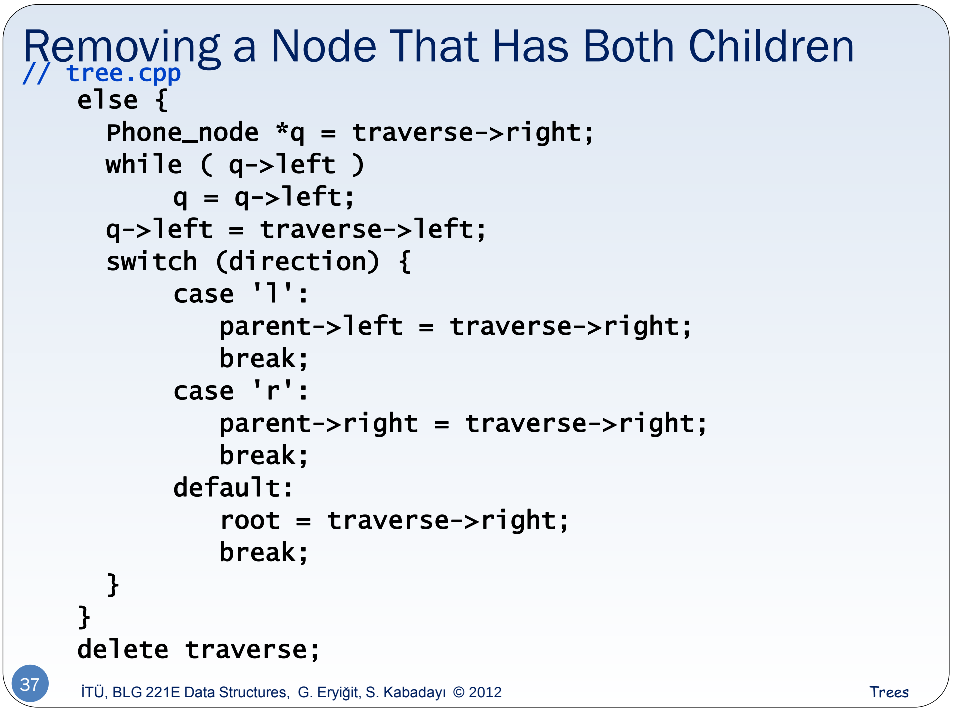
Removing a Node That Has Only Left Child

```
// tree.cpp
   else if (traverse->right == NULL) {
     switch (direction) {
         case 'l':
            parent->left = traverse->left;
            break;
         case 'r':
            parent->right = traverse->left;
            break;
         default:
            root = traverse->left;
            break;
     }
   }
   delete traverse;
```



Removing a Node That Has Only Right Child

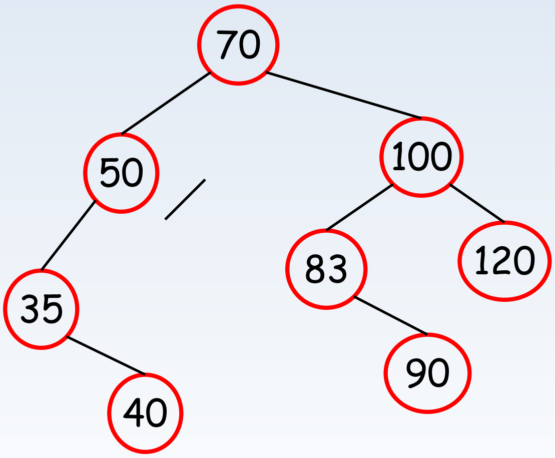
```
// tree.cpp
   else if (traverse->left == NULL) {
     switch (direction) {
         case 'l':
            parent->left = traverse->right;
            break;
         case 'r':
            parent->right = traverse->right;
            break;
         default:
            root = traverse->right;
            break;
     }
   }
   delete traverse;
```

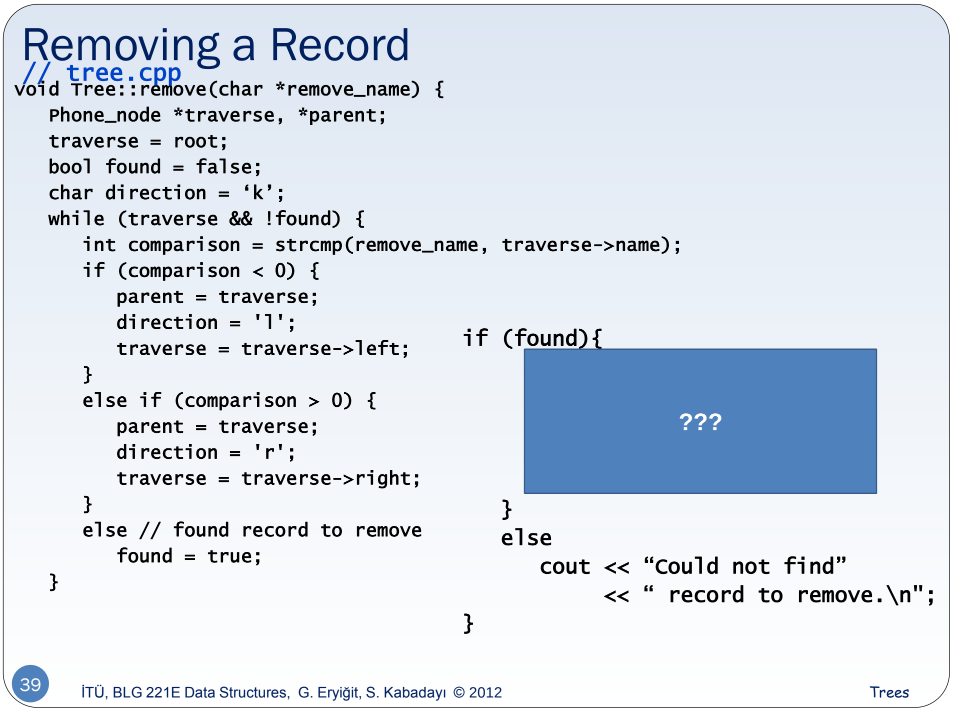


Removing a Node That Has Both Children

```
// tree.cpp
    else {
      Phone_node *q = traverse->right;
      while ( q->left )
           q = q->left;
      q->left = traverse->left;
      switch (direction) {
           case 'l':
              parent->left = traverse->right;
              break;
           case 'r':
              parent->right = traverse->right;
              break;
           default:
              root = traverse->right;
              break;
      }
    }
    delete traverse;
```

Example

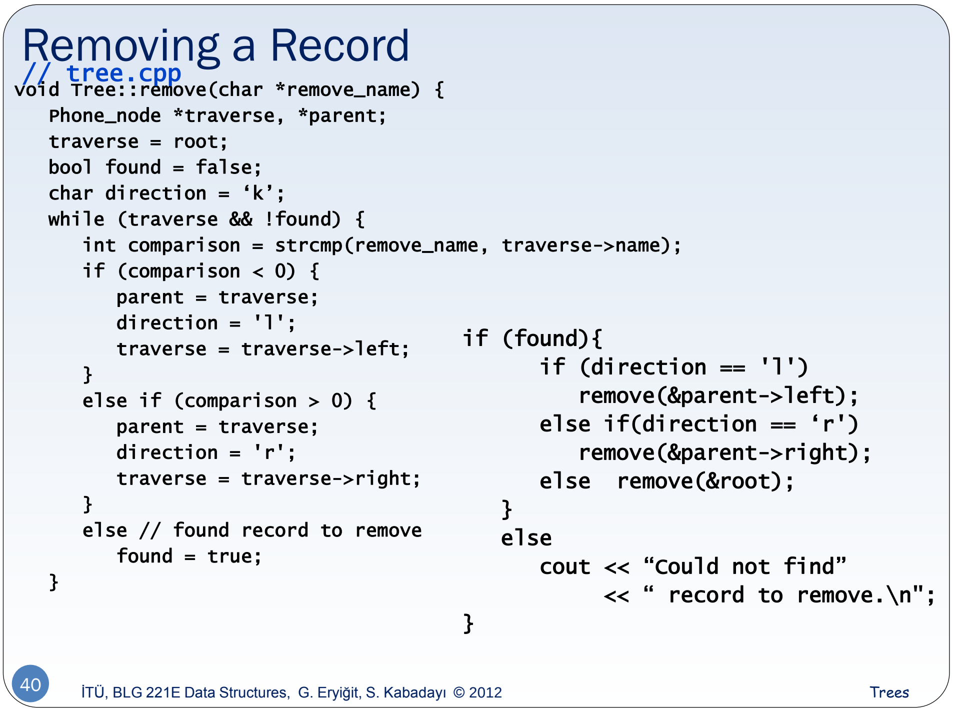




Removing a Record

```
// tree.cpp
void Tree::remove(char *remove_name) {
   Phone_node *traverse, *parent;
   traverse = root;
   bool found = false;
   char direction = ‘k’;
   while (traverse && !found) {
      int comparison = strcmp(remove_name, traverse->name);
      if (comparison < 0) {
         parent = traverse;
         direction = 'l';
         traverse = traverse->left;
      }
      else if (comparison > 0) {
         parent = traverse;
         direction = 'r';
         traverse = traverse->right;
      }
      else // found record to remove
         found = true;
   }
```

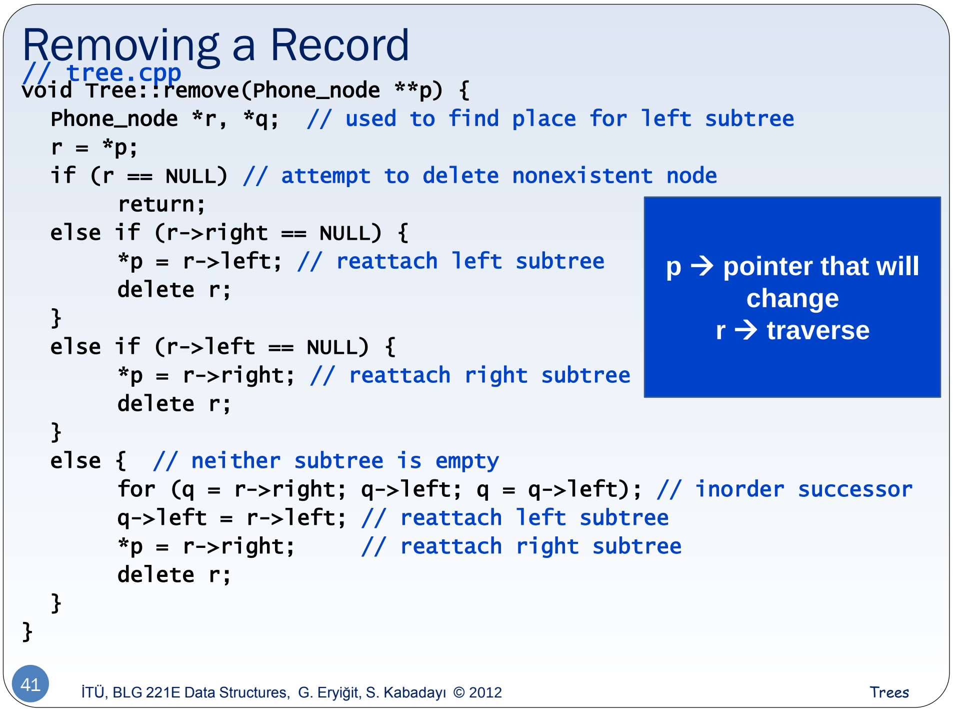
```
if (found){
                ???
   }
   else
      cout << “Could not find”
           << “ record to remove.\n";
}
```



Removing a Record

```
// tree.cpp
void Tree::remove(char *remove_name) {
   Phone_node *traverse, *parent;
   traverse = root;
   bool found = false;
   char direction = ‘k’;
   while (traverse && !found) {
      int comparison = strcmp(remove_name, traverse->name);
      if (comparison < 0) {
         parent = traverse;
         direction = 'l';
         traverse = traverse->left;
      }
      else if (comparison > 0) {
         parent = traverse;
         direction = 'r';
         traverse = traverse->right;
      }
      else // found record to remove
         found = true;
   }
```

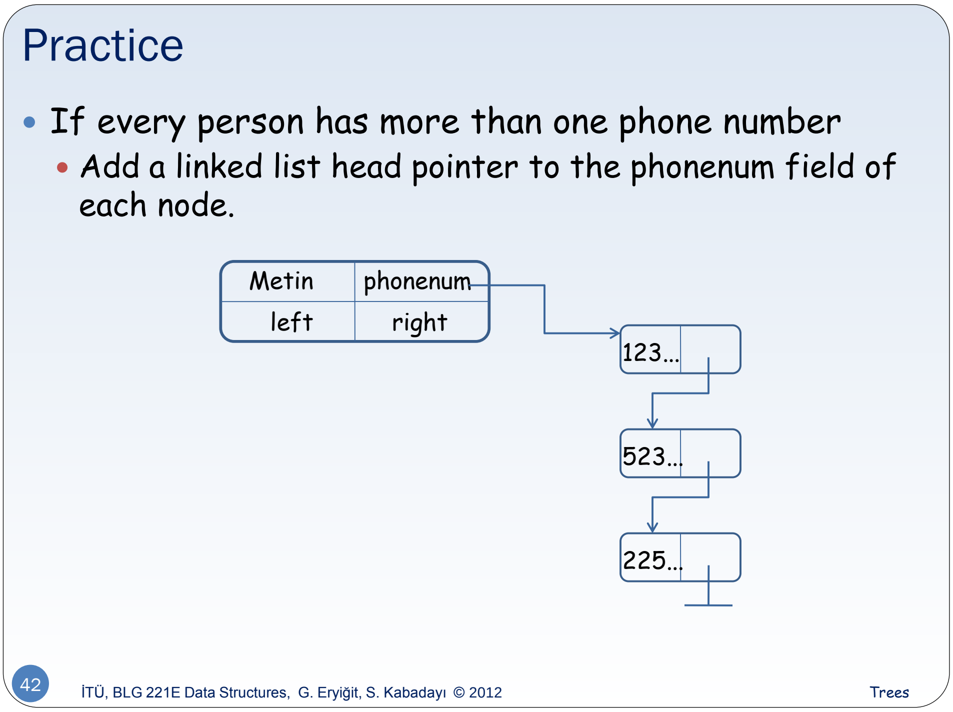
```
if (found){
      if (direction == 'l')
         remove(&parent->left);
      else if(direction == ‘r')
         remove(&parent->right);
      else  remove(&root);
   }
   else
      cout << “Could not find”
           << “ record to remove.\n";
}
```



Removing a Record

```
// tree.cpp
void Tree::remove(Phone_node **p) {
   Phone_node *r, *q;  // used to find place for left subtree
   r = *p;
   if (r == NULL) // attempt to delete nonexistent node
          return;
   else if (r->right == NULL) {
          *p = r->left; // reattach left subtree
          delete r;
   }
   else if (r->left == NULL) {
          *p = r->right; // reattach right subtree
          delete r;
   }
   else {  // neither subtree is empty
          for (q = r->right; q->left; q = q->left); // inorder successor
          q->left = r->left; // reattach left subtree
          *p = r->right;     // reattach right subtree
          delete r;
   }
}
```

p → pointer that will change
r → traverse



Practice

- If every person has more than one phone number
  - Add a linked list head pointer to the phonenum field of each node.