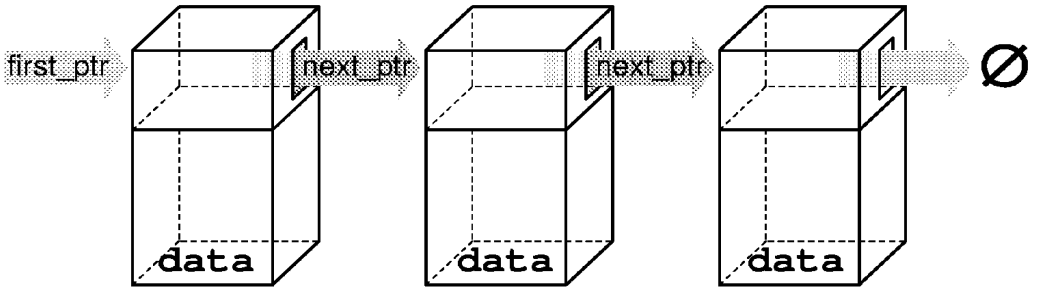
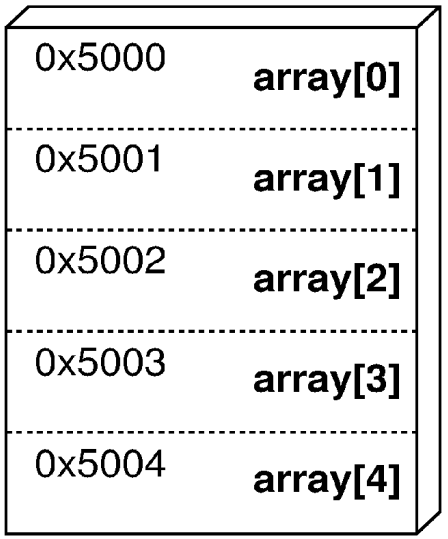
Chapter - 20
Advanced Pointers

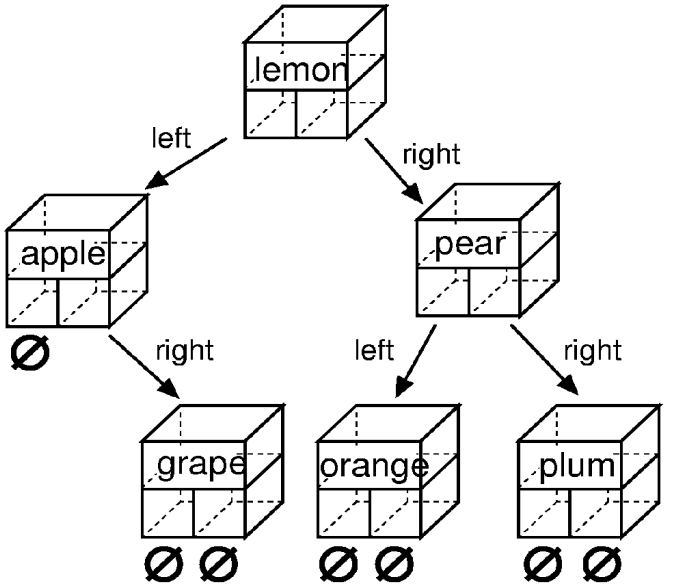
Advanced Data Structures



Linked list



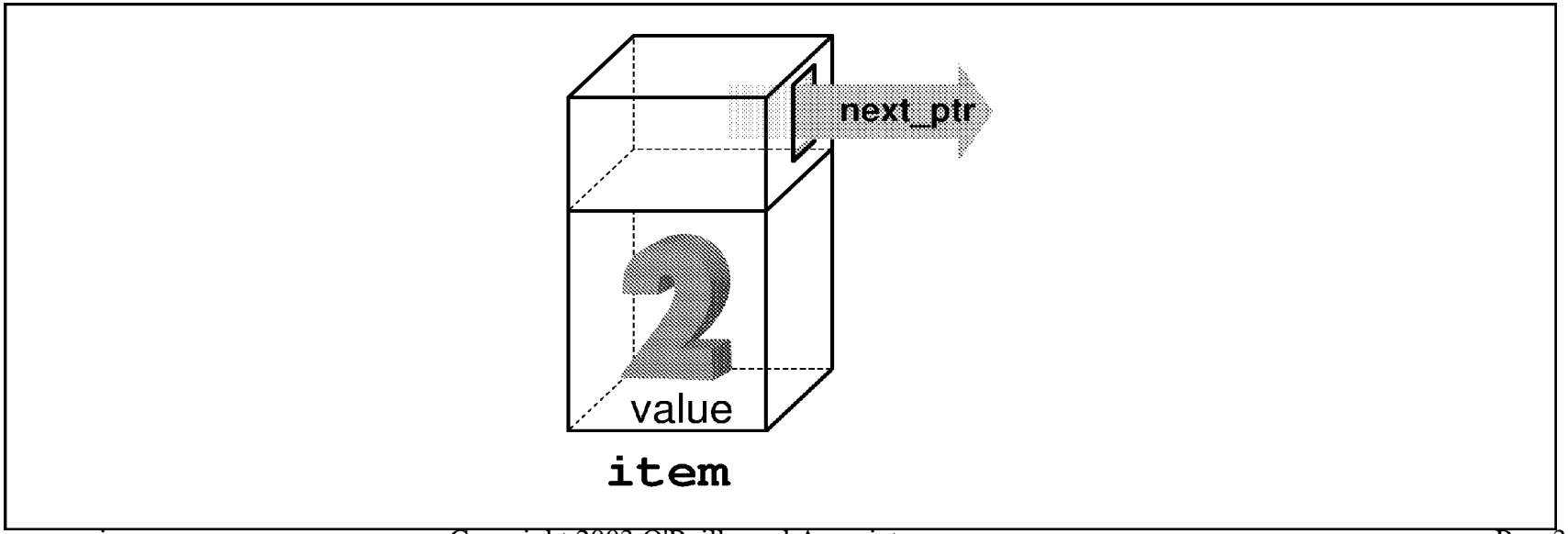
Array



Tree

Pointers, Structures, and Classes

```
public:

};
```



new operator

The **new** operator creates a new variable from a section of memory called the *heap* and returns a pointer to it.

```
    public:



};
```

delete operator

The “delete” operator returns the storage to the heap. Only data allocated by “new” can be returned this way.

Normal variables:

```
delete pointer;   // Where pointer is a pointer to
                  // a simple object
pointer = NULL;
```

Array variables:

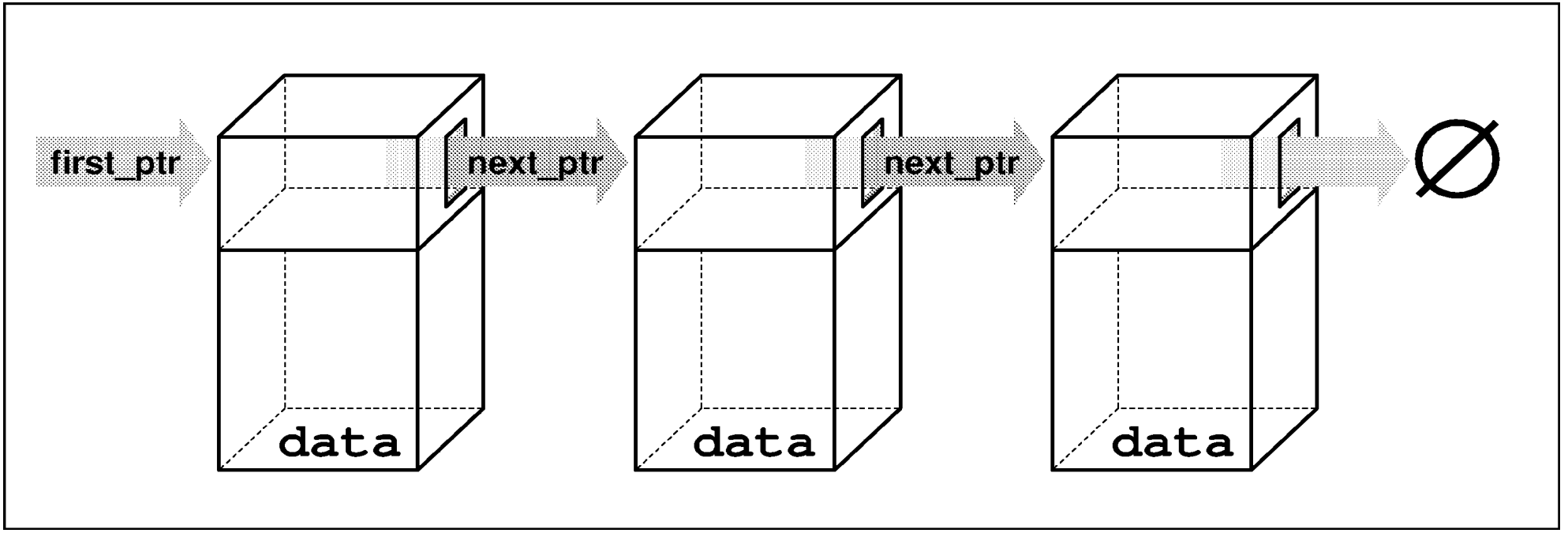
```
delete pointer[];        // Where pointer is a
                         // pointer to an array
pointer = NULL;
```

Example

```
{

    /*

     */
    delete[] data_ptr;
    data_ptr = NULL;
}
```

What would happen if we didn't *free* the memory?

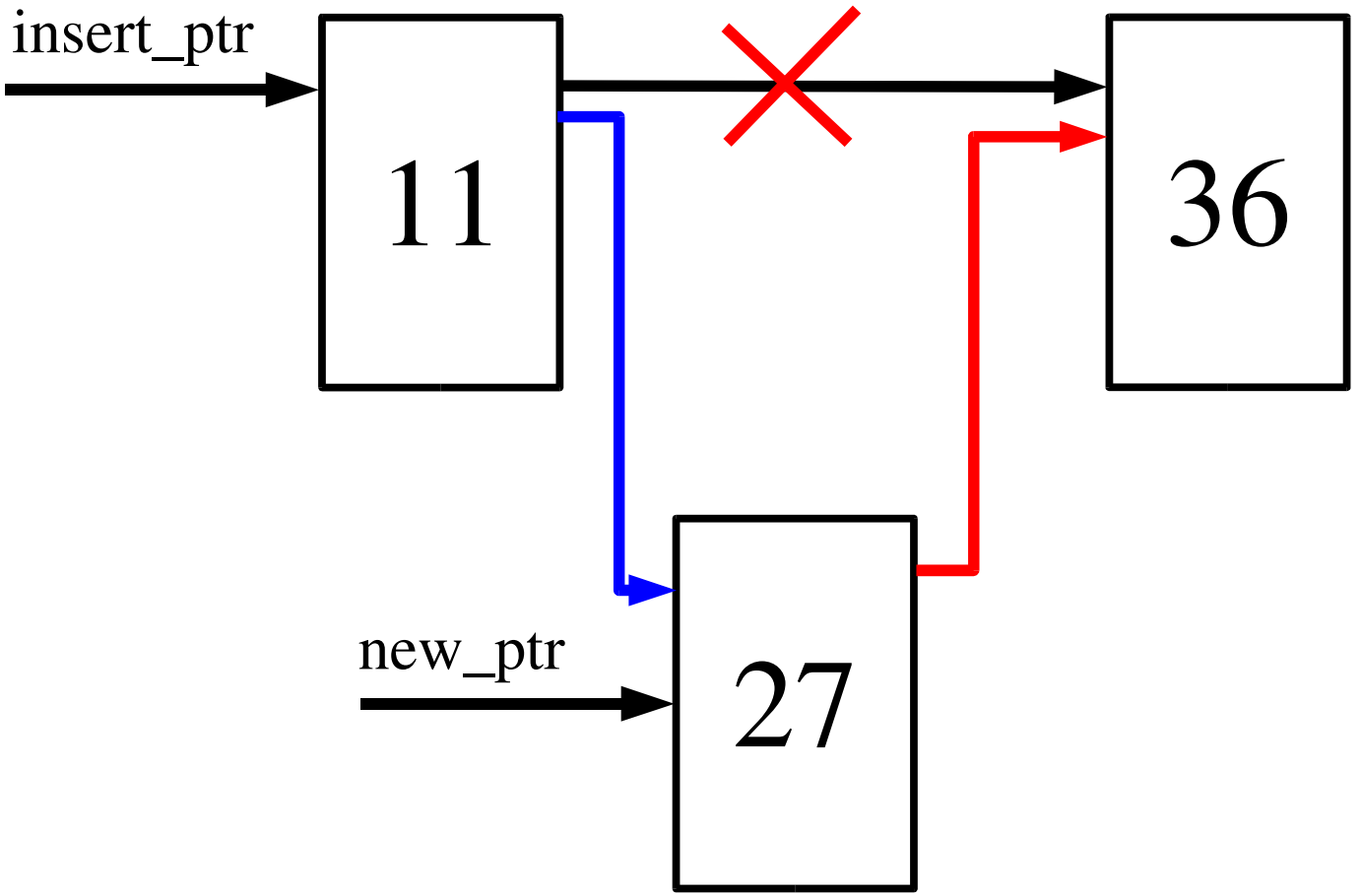
Linked List



Linked List

```
    public:

            public:

            private:


            friend class linked_list;
        };

    public:
        linked_list_element *first_ptr;// First element

        // Initialize the linked list
        linked_list(void): first_ptr(NULL) { }

        // ... Other member functions
};
```

Adding an element




```
new_ptr = insert_ptr->next;
insert_ptr->next = new_ptr;
```

C++ code to add an element

```
{



}
```

Finding an element in a list

```
        /*

          */
}
```

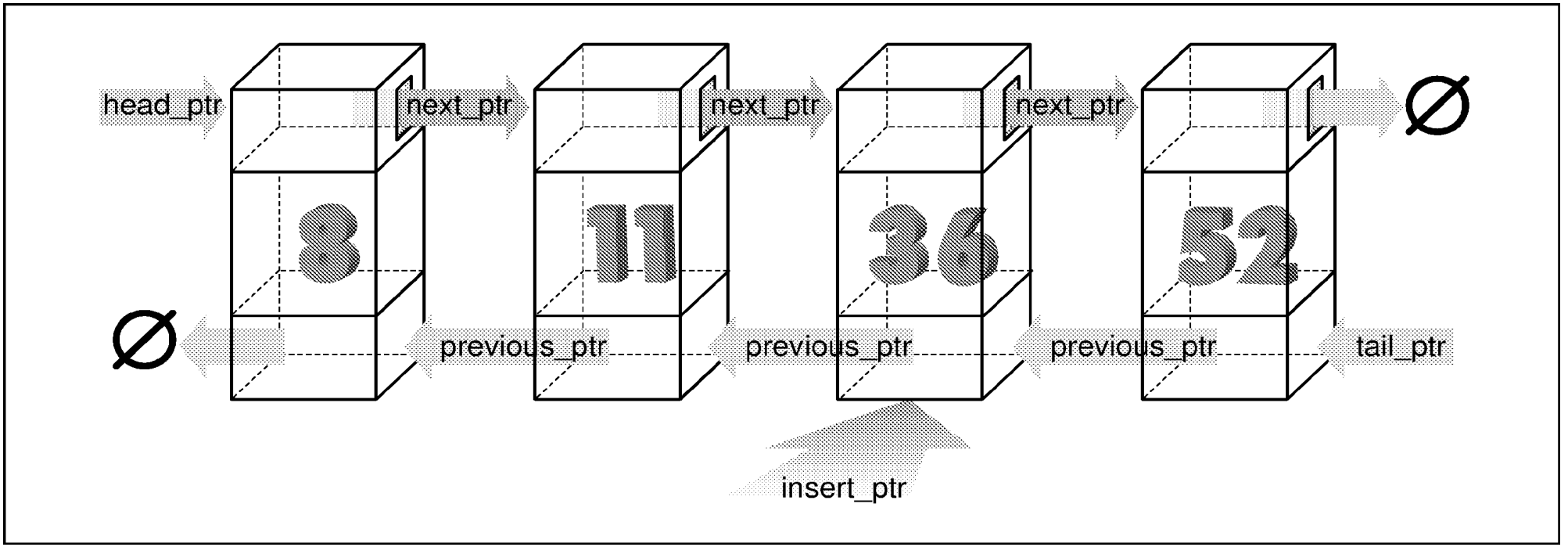
Note: The following two statements are equivalent:

```
(*current_ptr).data = value;
current_ptr->data = value;
```

The C++ code

```
{

    /*

     */

            break;

            break;
    }

}
```

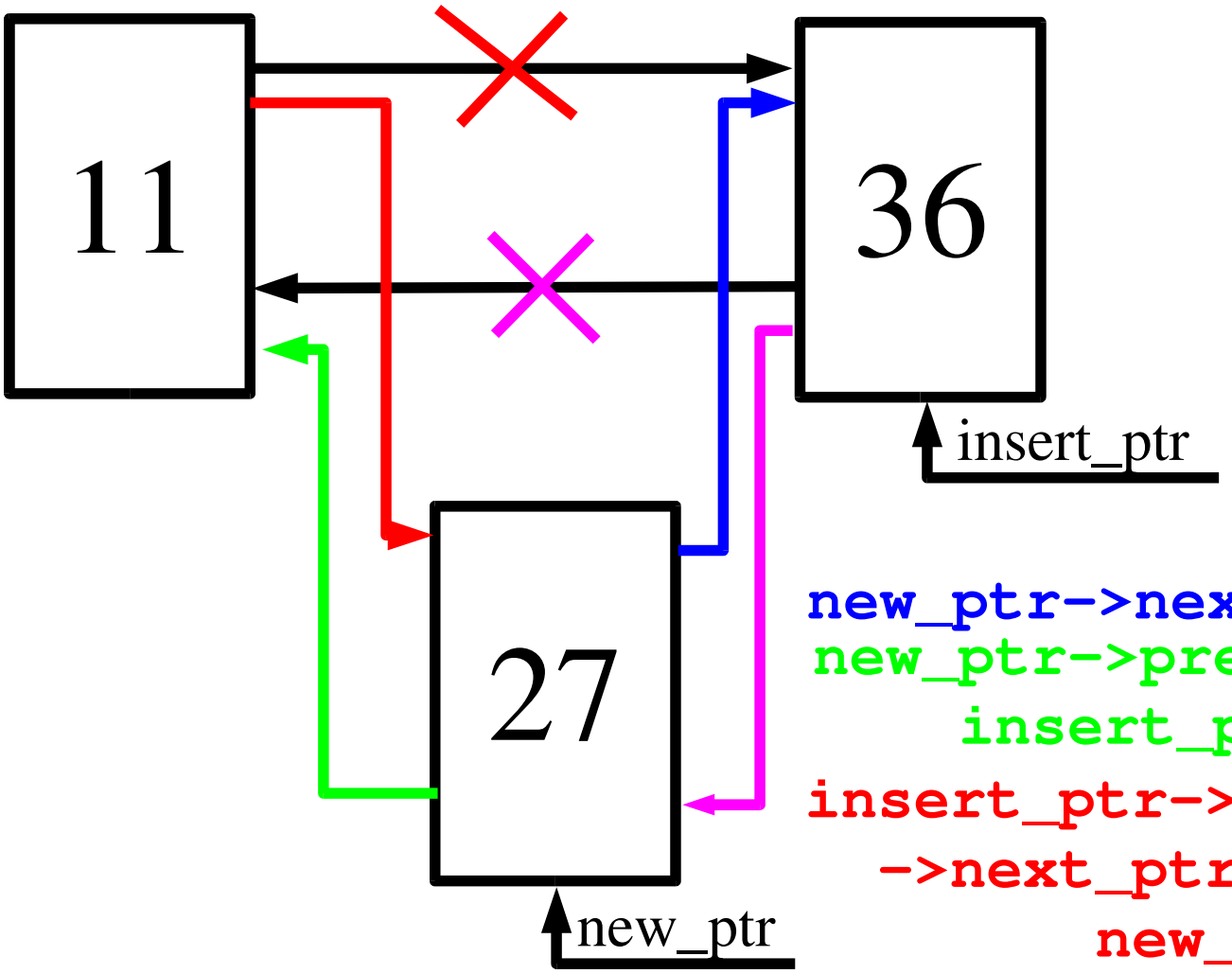
Double Linked List



Double Linked List

```
private:

        public:

        private:


        friend class double_list;
    };
public:


    double_list(void) { head_ptr = NULL; }

    // ... other member functions
```

Adding an element

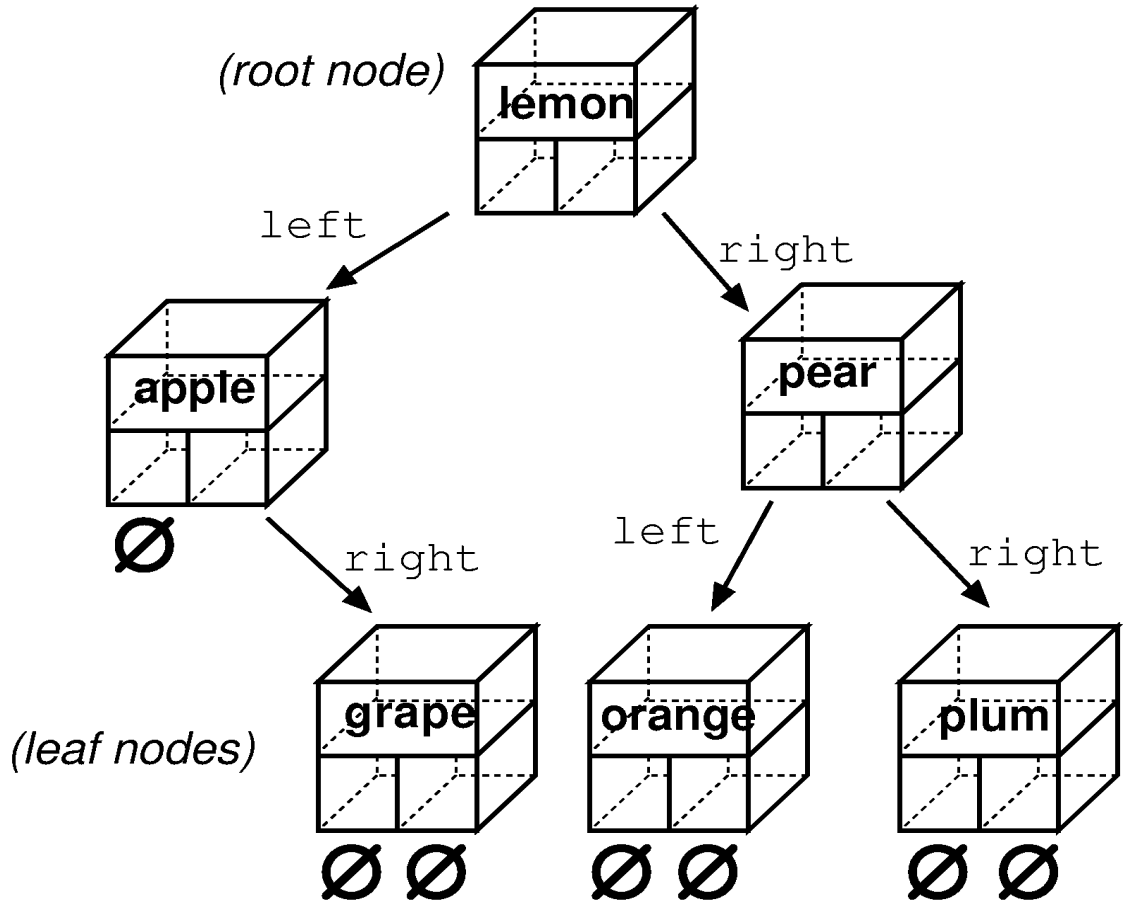



```
new_ptr->next_ptr = insert_ptr
new_ptr->previous_ptr =
     insert_ptr->previous_ptr
insert_ptr->previous_ptr
  ->next_ptr =
          new_ptr;

insert->previous_ptr = new_ptr;
```

Adding an element

```
{

    /*

     */

            break;

            break;
    }

}
```

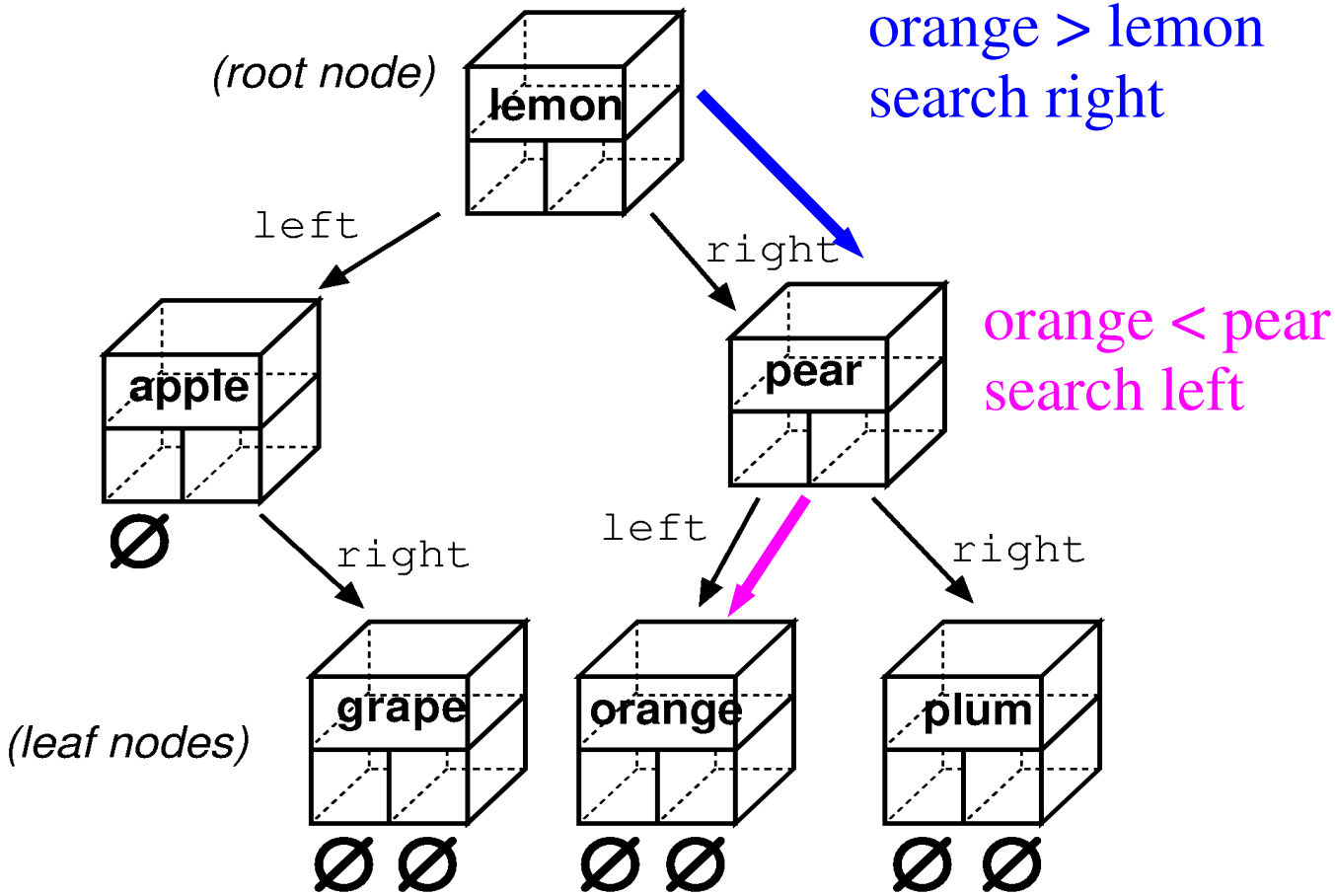
Trees



Trees

```
class tree {
    private:

            public:

            private:


            friend class tree;
        };
     public:


        tree(void) { root = NULL; };
        // ... other member function
};
```

Tree Search



orange = orange
success

Tree Insert Rules

The algorithm for inserting a word in a tree is:

1. If this is a null tree (or sub-tree), create a one-node tree with this word.
2. If this node contains the word, do nothing.
3. Otherwise, enter the word in the left or right sub-tree, depending on the value of the word.

Does this follow the two rules of recursion?

Adding a node

```
{
    }
        return;
    else
}
void tree::enter(char *word) {
    enter_one(root, word);
};
```

Printing a Tree

The printing algorithm is:

1. For the null tree, print nothing.
2. Print the data that comes before this node (left tree).
3. Print this node.
4. Print the data that comes after this node (right tree).

```
{



    print_tree(top->left);
    std::cout << top->word << '\n';
    print_tree(top->right);
}
void tree::print(void) {
     print_one(root);
}
```