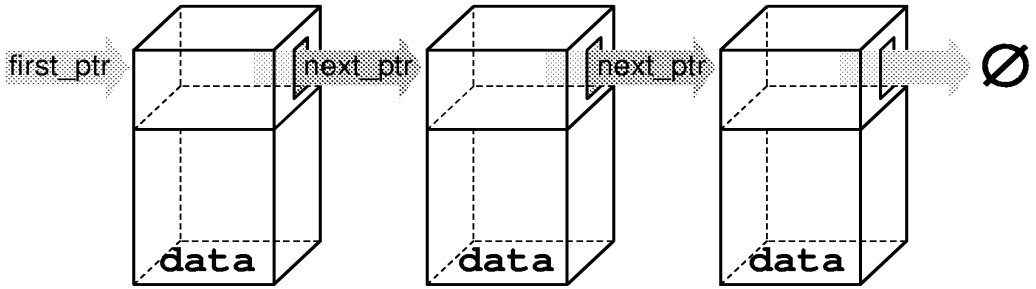
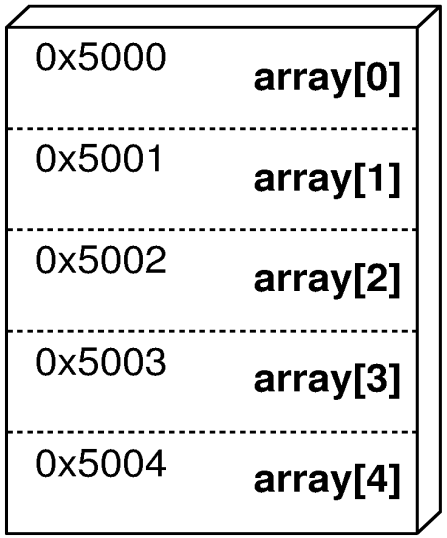
# Chapter - 20
# Advanced Pointers

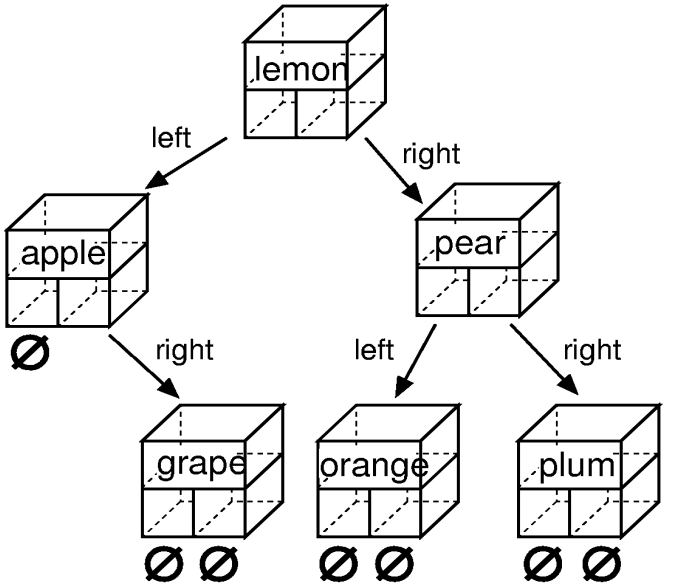
# Advanced Data Structures




*Linked list*



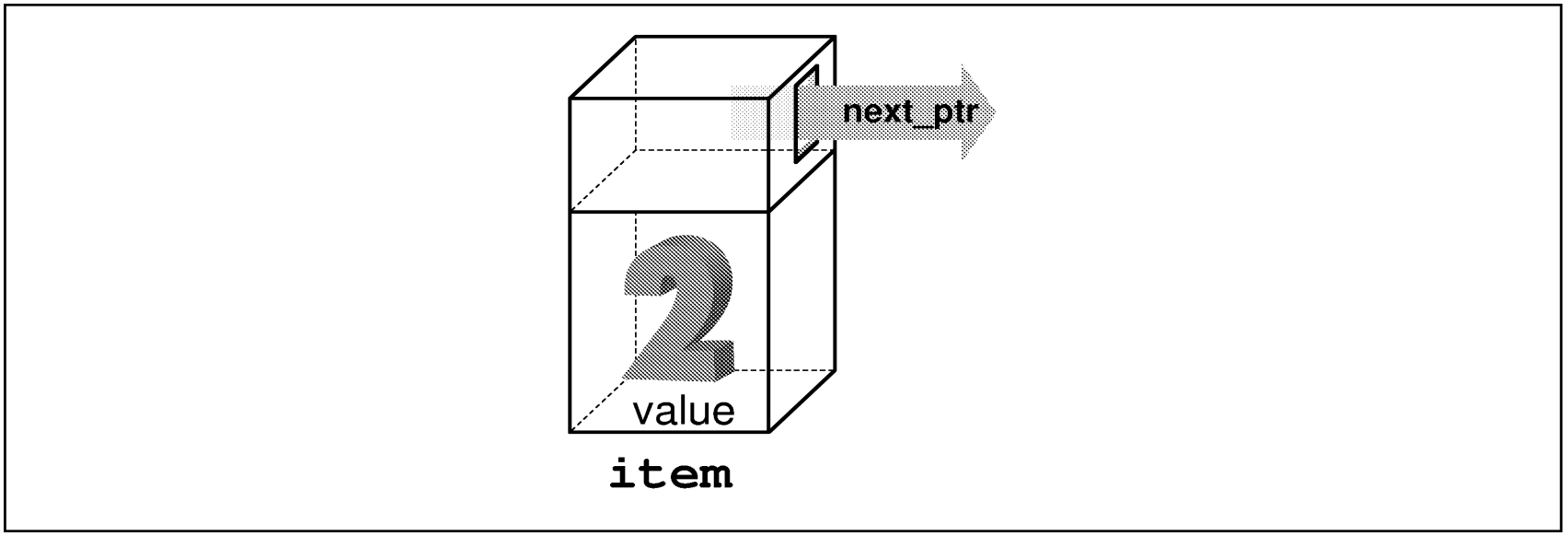

*Array*




*Tree*

# Pointers, Structures, and Classes

```
class item {
    public:
        int value;
       item *next_ptr;
};
```



# *new* operator

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

```
int *element_ptr;   // Pointer to an integer

element_ptr = new int;   // Get an integer from the heap

class person {
    public:
        std::string name;           // name of the person
        std::string address;        // Where he lives
        std::string city_state_zip; // Part 2 of address
        int     age;                // his age
        float   height;             // his height in inches
};
struct person *new_ptr;
new_ptr = new person;
char *string_ptr;
string_ptr = new char[80];
```

# ***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:

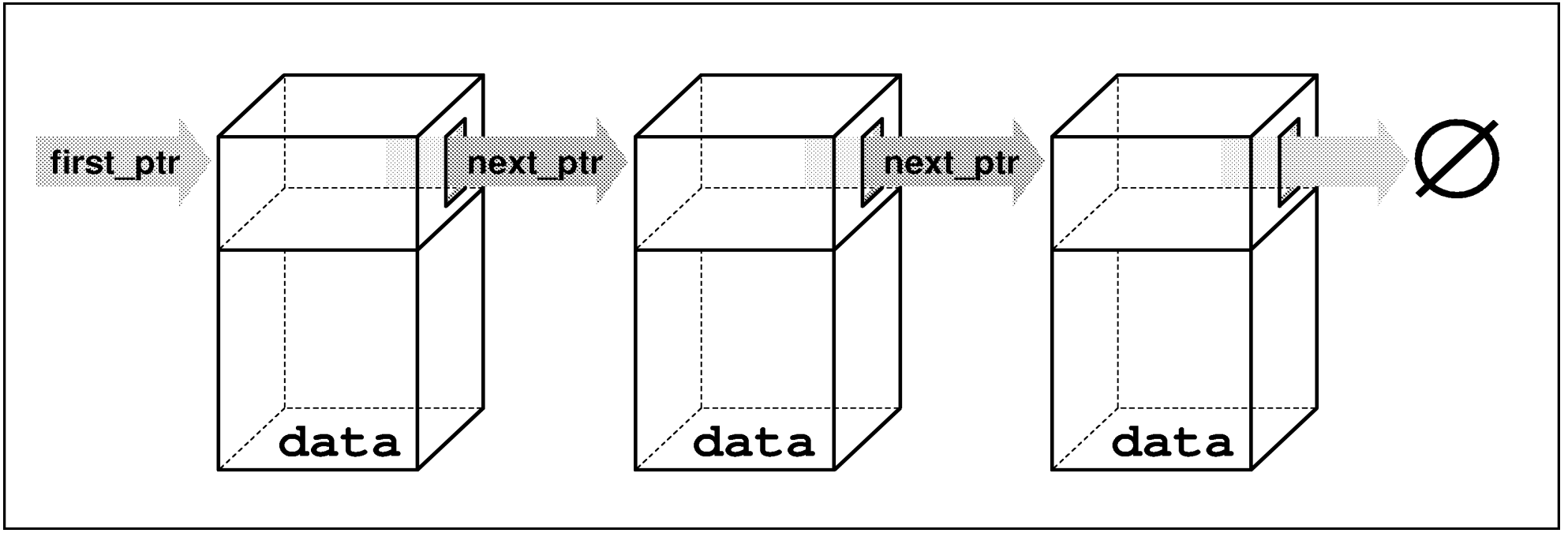
```
delete pointer[];       // Where pointer is a
                        // pointer to an array
pointer = NULL;
```

# Example

```
const DATA_SIZE = (16 * 1024);

void copy(void)
{
    char *data_ptr;     // Pointer to large data buffer

    data_ptr = new char[DATA_SIZE];       // Get the buffer

    /*
     * Use the data buffer to copy a file
     */
    delete[] data_ptr;
    data_ptr = NULL;
}
```

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

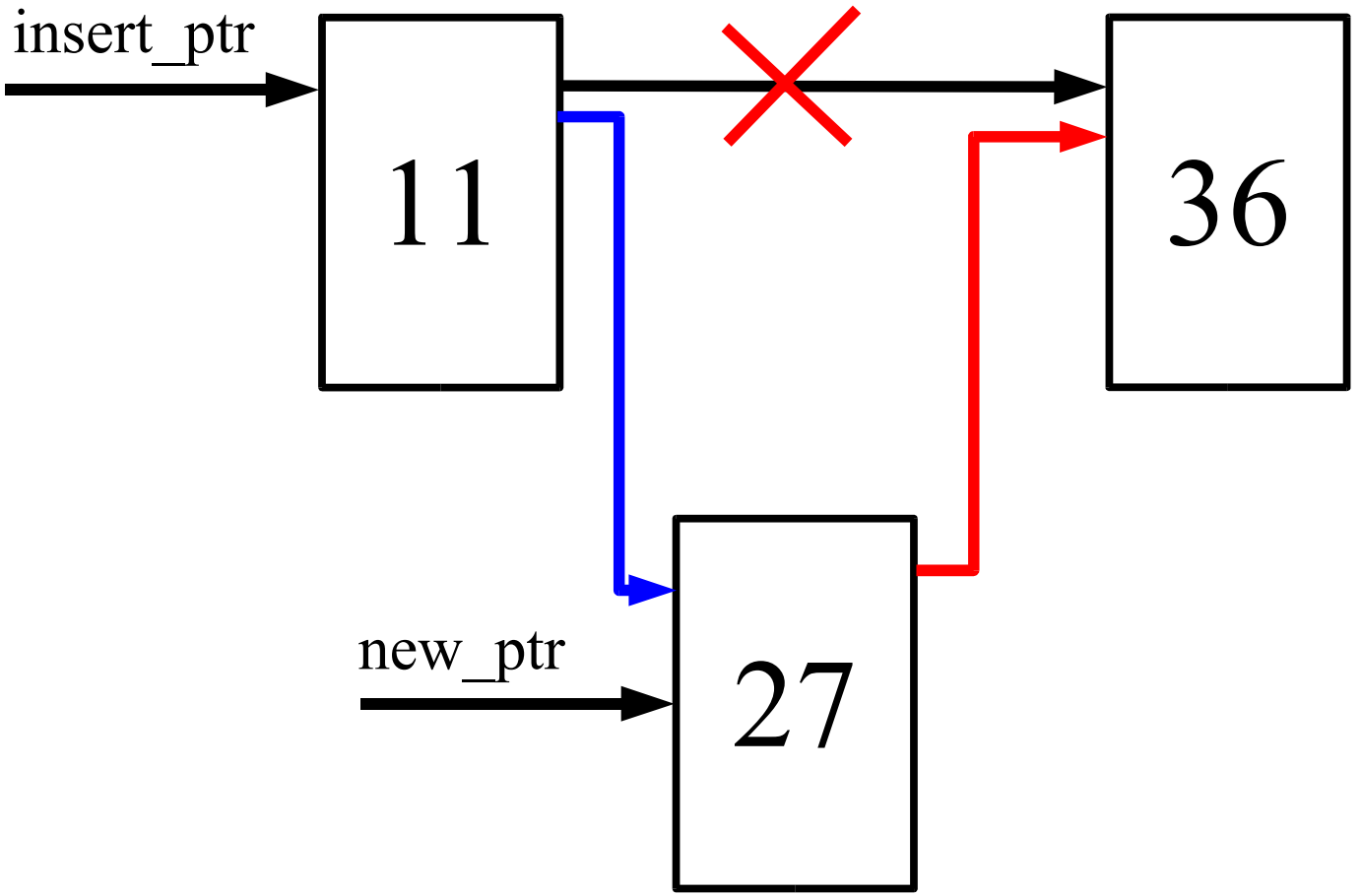
# Linked List



# Linked List

```
    public:
        class linked_list_element {
            public:
                std::string data;      // data in this element
            private:
                linked_list_element *next_ptr; // pointer to
                                               // next element
            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

```
void linked_list::add_list(int item)
{
    // pointer to the next item in the list
    linked_list_element *new_ptr;

    new_ptr = new linked_list;

    strcpy((*new_ptr).data, item);
    (*new_ptr).next_ptr = first_ptr;
    first_ptr = new_ptr;
}
```

# Finding an element in a list

```
int linked_list::find(char *name) {
    /* current structure we are looking at */
    linked_list_element *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
           (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list
     * and didn't find the name
     */
    return (current_ptr != NULL);
}
```

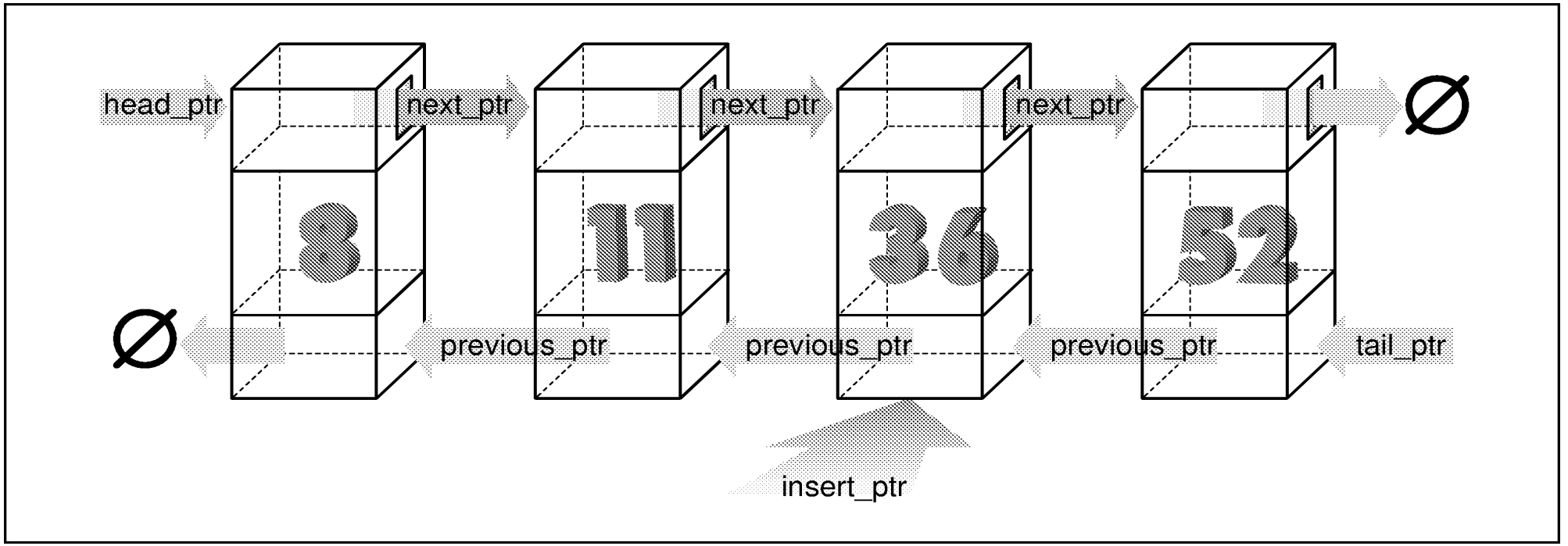
Note: The following two statements are equivalent:

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

# The C++ code

```
void linked_list::enter(int item)
{
    list *before_ptr; // insert before this element
    list *after_ptr;  // insert after this element

    /*
     * Warning: This routine does not take
     *   care of the case where the element is
     *   inserted at the head of the list
     */
    before_ptr = first_ptr;
    while (1) {
        insert_ptr = before_ptr;
        insert_ptr = insert_ptr->next_ptr;

        // did we hit the end of the list?
        if (insert_ptr == NULL)
            break;

        // did we find the place?
        if (item >= insert_ptr->data)
            break;
    }
    before_ptr->next_ptr = new_ptr;
    new_ptr->next_ptr = after_ptr;
    // create new item
    new_ptr = new list;
    new_ptr->data = item;

    // link in the new item
    before_ptr->next_ptr = new_ptr;
    new_ptr->next_ptr = after_ptr;
}
```

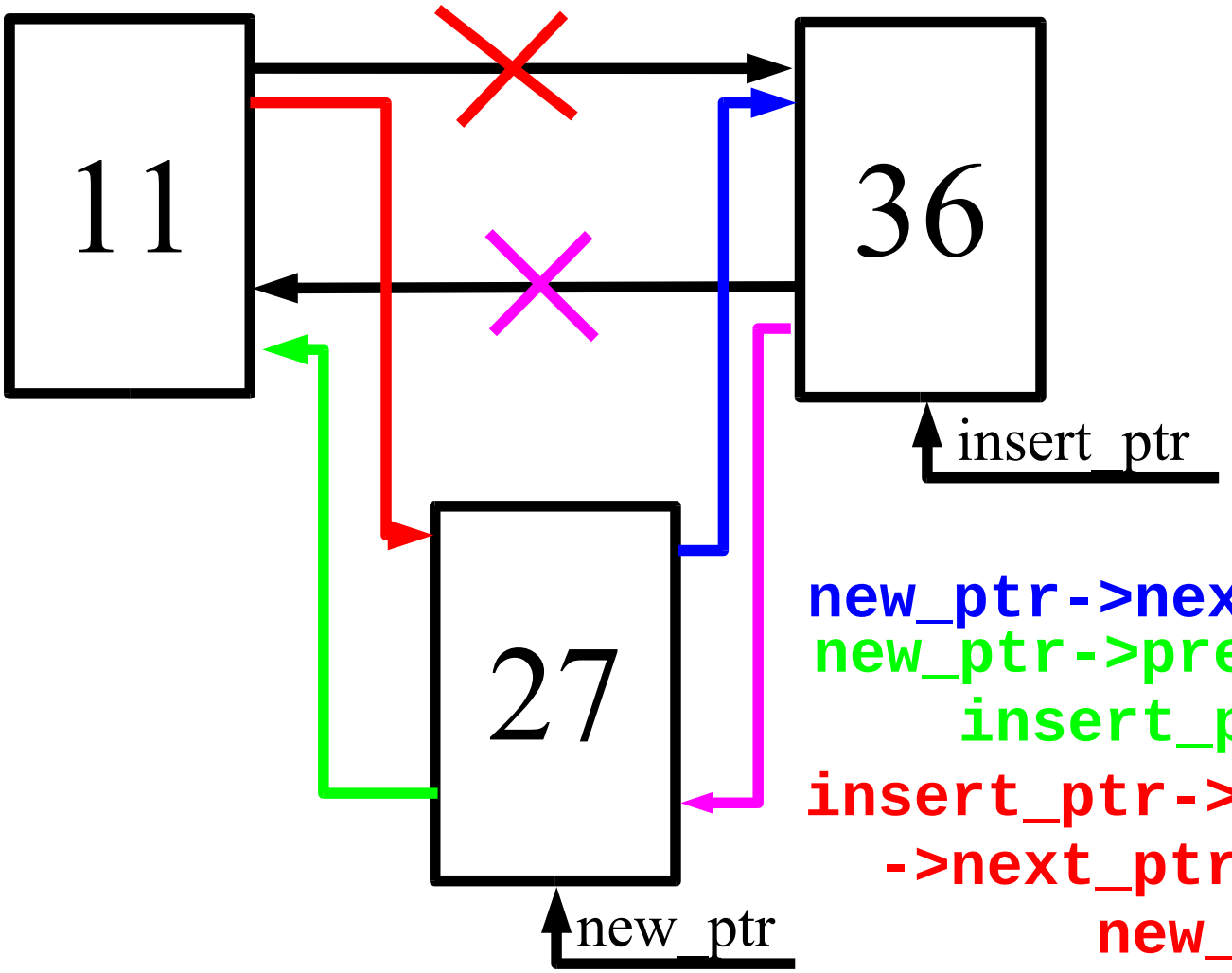
# Double Linked List



# Double Linked List

```
private:
    class double_list_element {
        public:
            int data;                    // data item
        private:
            double_list_element *next_ptr;    // forward link
            double_list_element *previous_ptr;// backward link
        friend class double_list;
    };
public:
    double_list_element *head_ptr;   // Head of the list


    double_list(void) { head_ptr = NULL; }


    // ... other member functions
```

# Adding an element

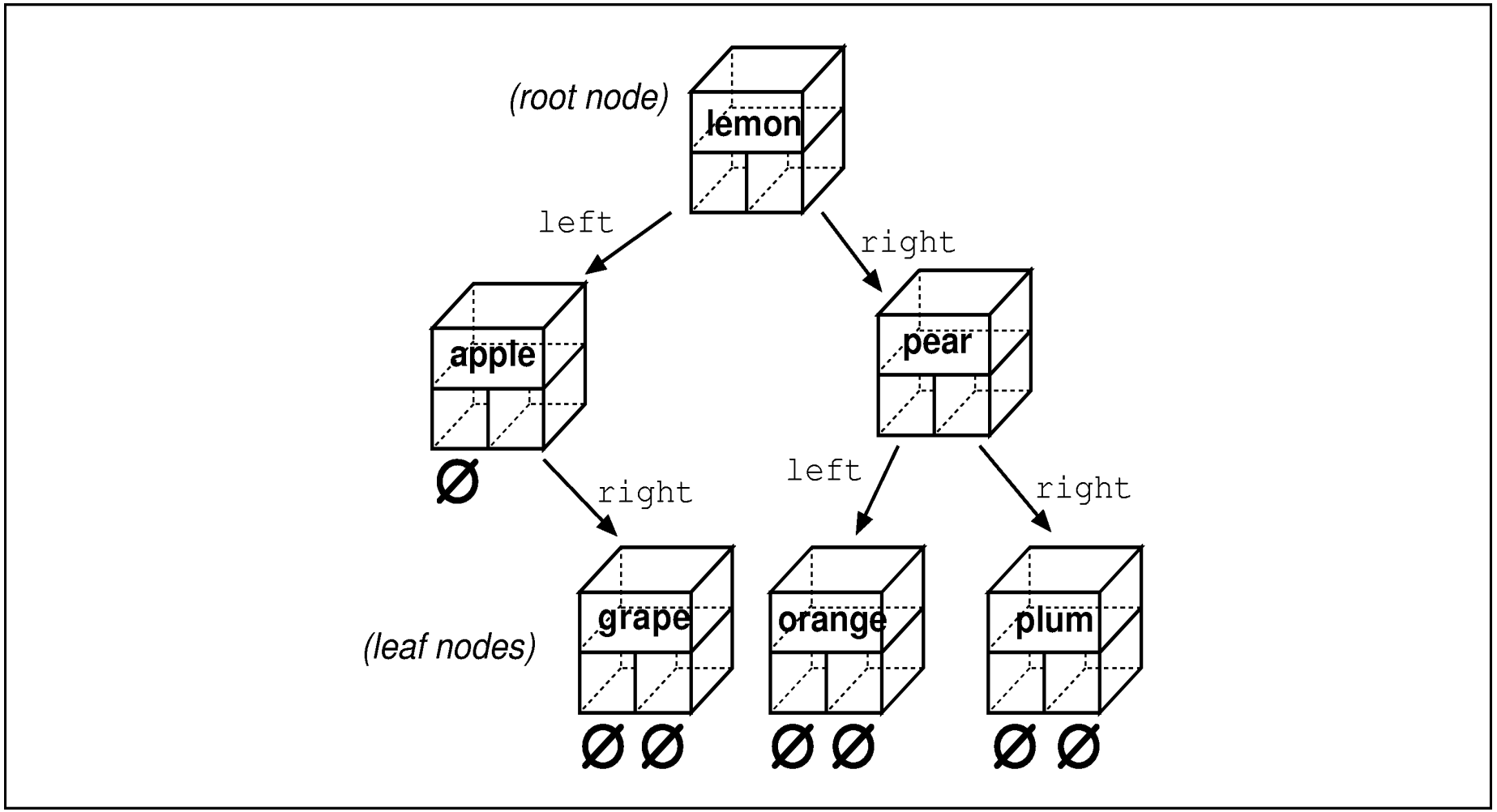



```
new_ptr->next_ptr = insert_ptr
new_ptr->previous_ptr =
    insert_ptr->previous_ptr
insert_ptr->previous_ptr
  ->next_ptr =
        new_ptr;

insert->previous_ptr = new_ptr;
```

# Adding an element

```
void double_list::enter(int item)
{
    double_list_elememt *insert_ptr; // insert b4 this elem

    /*
     * Warning: This routine does not take
     *   care of the case where the element is
     *   inserted at the head of the list
     *   or the end of the list
     */
    insert_ptr = head_ptr;
    while (1) {
        insert_ptr = insert_ptr->next;

        // have we reached the end
        if (insert_ptr == NULL)
            break;

        // have we reached the right place
        if (item >= insert_ptr->data)
            break;
    }
    // create new element
    new_ptr = new double_list;
    new_ptr->next_ptr = insert_ptr;
    new_ptr->previous_ptr = insert_ptr->previous_ptr;

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

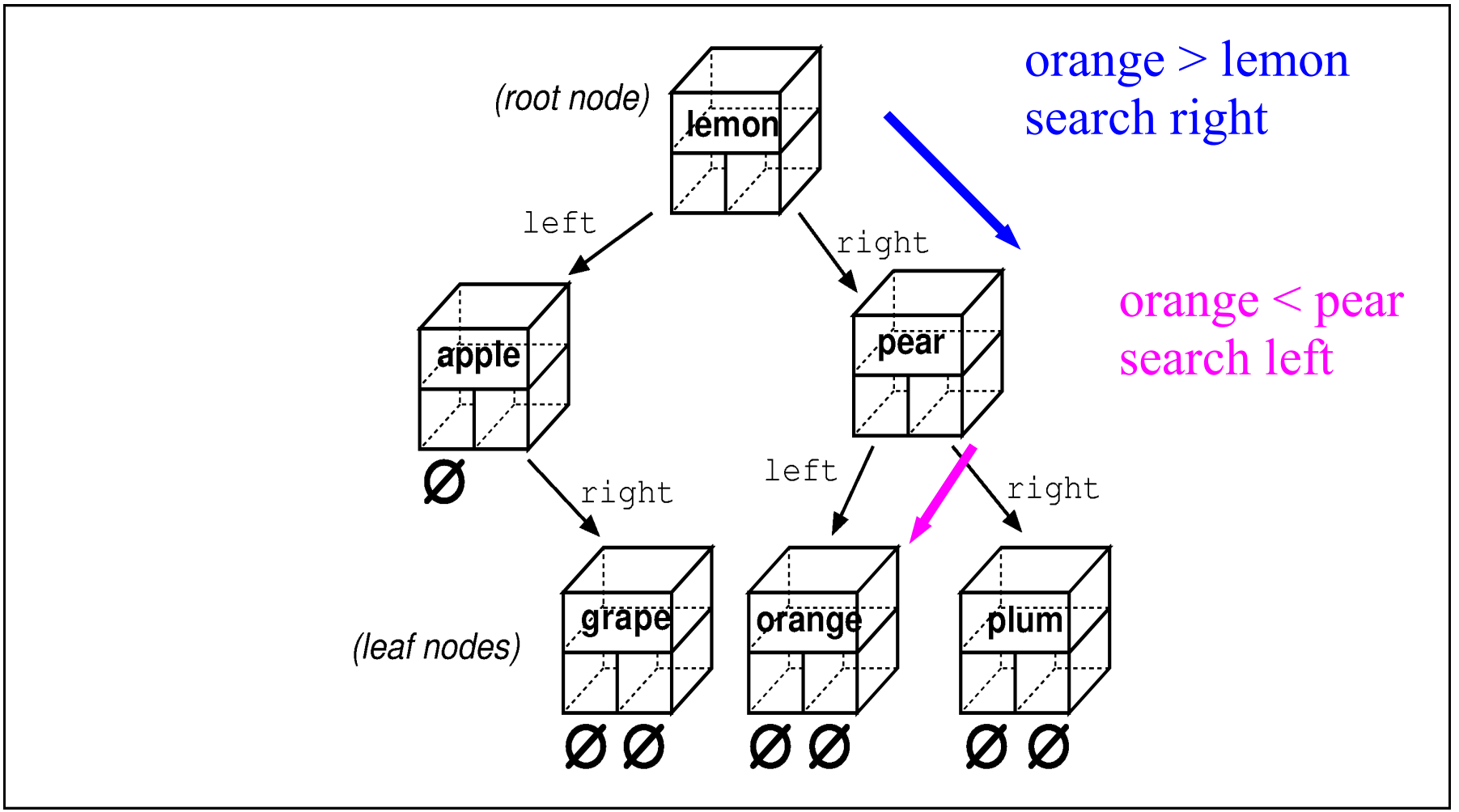
# Trees



# Trees

```
class tree {
    private:
        class node {
            public:
                char *data;      // word for this tree
            private:
                node *right;     // tree to the right
                node *left;      // tree to the left
            friend class tree;
        };
     public:
        node *root;  // Top of the tree (the root)

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

# Tree Search



# 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

```
void tree::enter_one(node *&node, char *word)
{
    int  result;        // result of strcmp

    // see if we have reached the end
    if (node == NULL) {
        node = new node;

        node->left = NULL;
        node->right = NULL;
        node->word = strdup(word);
    }
    result = strcmp(node->word, word);
    if (result == 0)
        return;

    if (result < 0)
        enter(node->right, word);
    else
        enter(node->left, word);
}
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).

```
void tree::print_one(node *top)
{
    if (top == NULL)
        return;                    // short tree

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