ECE 217:
Data Structure and Algorithm

Lecture 4: Linked List

Instructor: Shayan (Sean) Taheri, Ph.D.

Assistant Professor

The Department of Electrical and Cyber Engineering (ECE)

The Institute for Health and Cyber Knowledge (I-HACK)

The Gannon University (GU)

Personal Information

- Name: Shayan (Sean) Taheri.
- Date of Birth: July/28/1991.
- Past Position: Postdoctoral Fellow at University of Florida.
- Ph.D. Degree: Electrical Engineering from the University of Central Florida.
- M.S. Degree: Computer Engineering from the Utah State University.
- University Profile: https://www.gannon.edu/FacultyProfiles.aspx?profile=taheri001

Objectives

In this lecture, you will:

- Learn about linked lists
- Become aware of the basic properties of linked lists
- Explore the insertion and deletion operations on linked lists
- Discover how to build and manipulate a linked list
- Learn how to construct a doubly linked list

Introduction

- Data can be organized and processed sequentially using an array, called a sequential list
- Problems with an array
  - Array size is fixed
  - Unsorted array: searching for an item is slow
  - Sorted array: insertion and deletion is slow

Linked Lists

- Linked list: a list of items (nodes), in which the order of the nodes is determined by the address, called the **link**, stored in each node

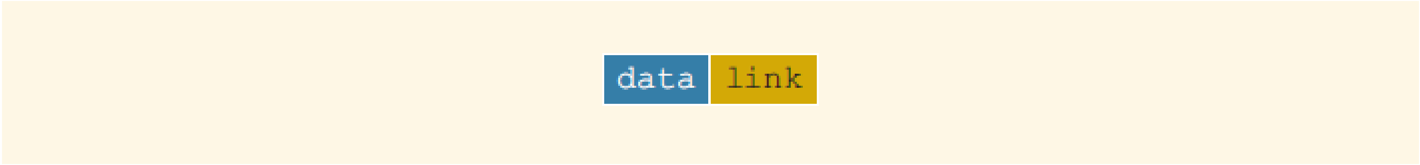


FIGURE 18-1 Structure of a node

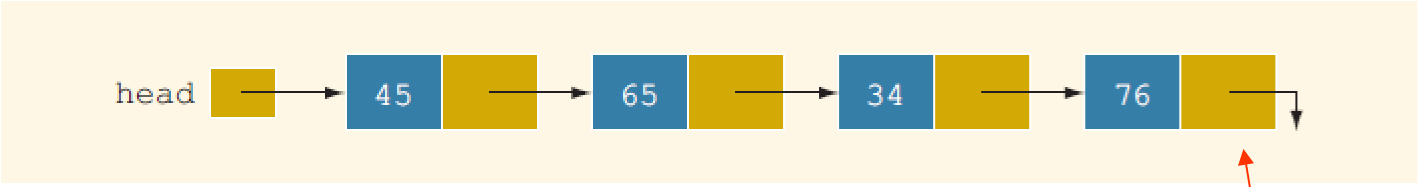
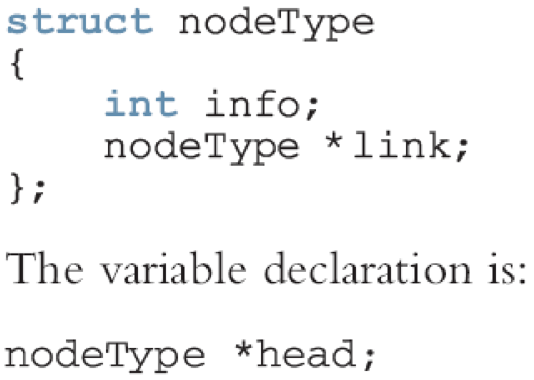


FIGURE 18-2 Linked list

Link field in last node is NULL

Linked Lists (cont'd.)

- Because each node of a linked list has two components, we need to declare each node as a class or struct
  - Data type of a node depends on the specific application
  - The link component of each node is a pointer



```
struct nodeType
{
    int info;
    nodeType *link;
};
```

The variable declaration is:

```
nodeType *head;
```

Linked Lists: Some Properties

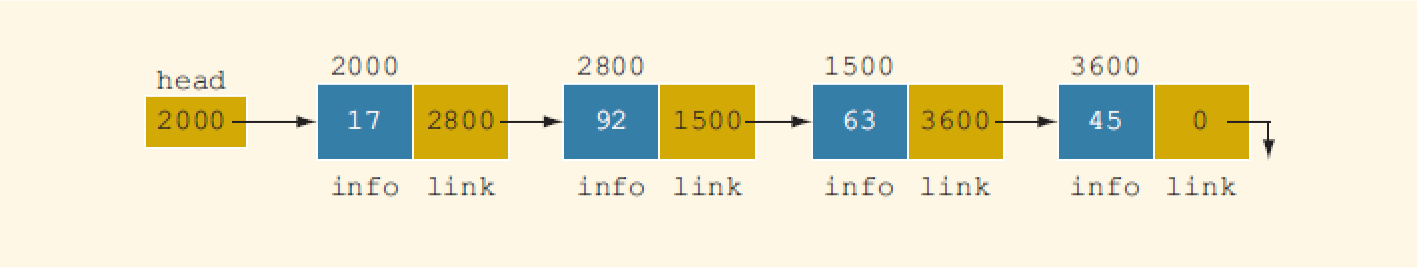
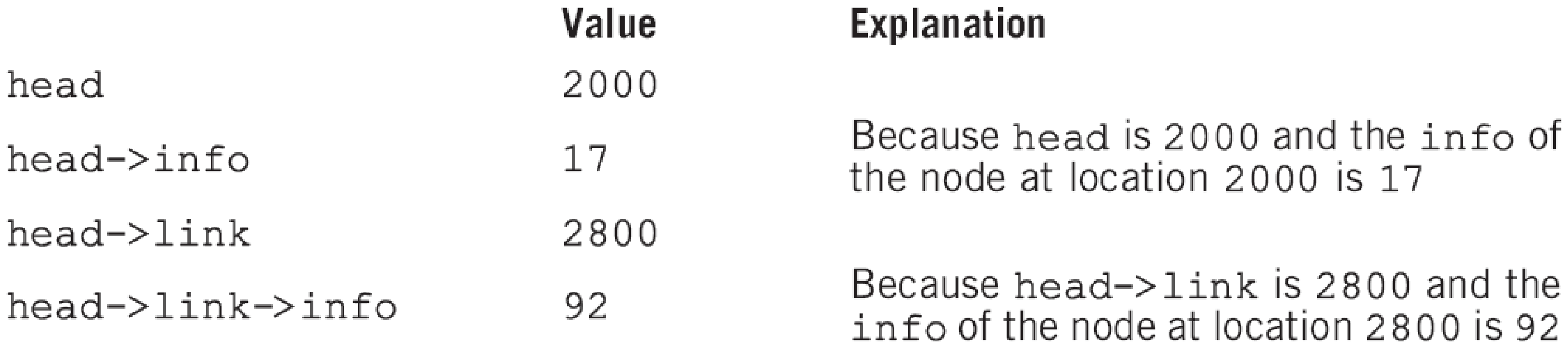


FIGURE 18-4 Linked list with four nodes



	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

Linked Lists: Some Properties (cont'd.)

- current = head;
  - Copies value of head into current

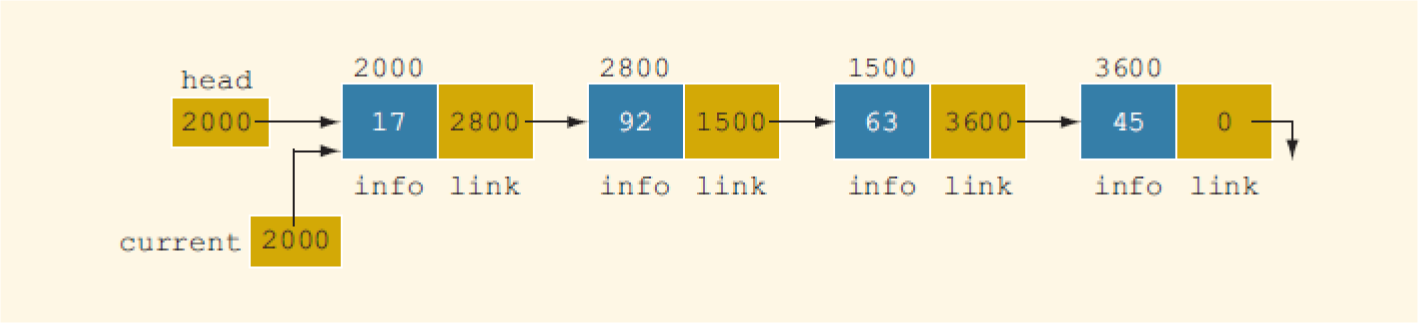


FIGURE 18-5 Linked list after the statement `current = head;` executes

	Value
`current`	2000
`current->info`	17
`current->link`	2800
`current->link->info`	92

Linked Lists: Some Properties (cont'd.)

- current = current->link;

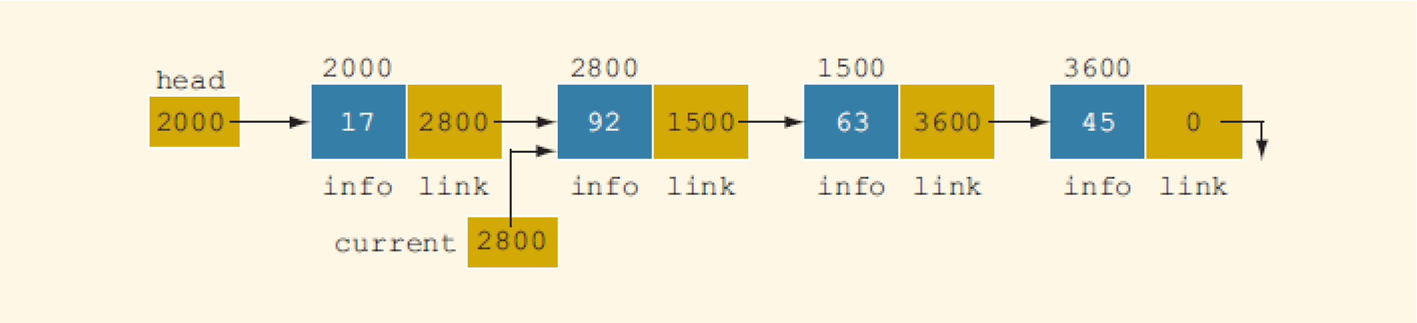
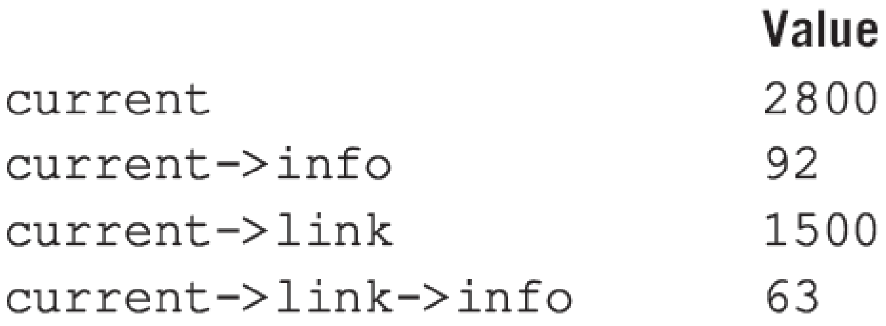


FIGURE 18-6 List after the statement `current = current->link;` executes



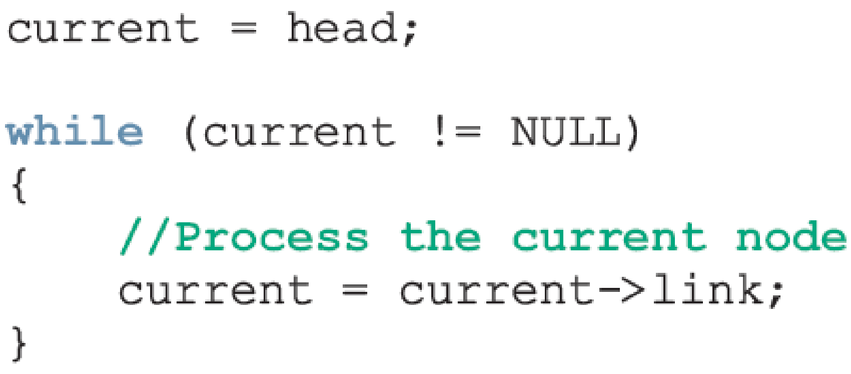
	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63

Traversing a Linked List

- The basic operations of a linked list are:
  - Search to determine if an item is in the list
  - Insert an item in the list
  - Delete an item from the list
- Traversal: given a pointer to the first node of the list, step through the nodes of the list

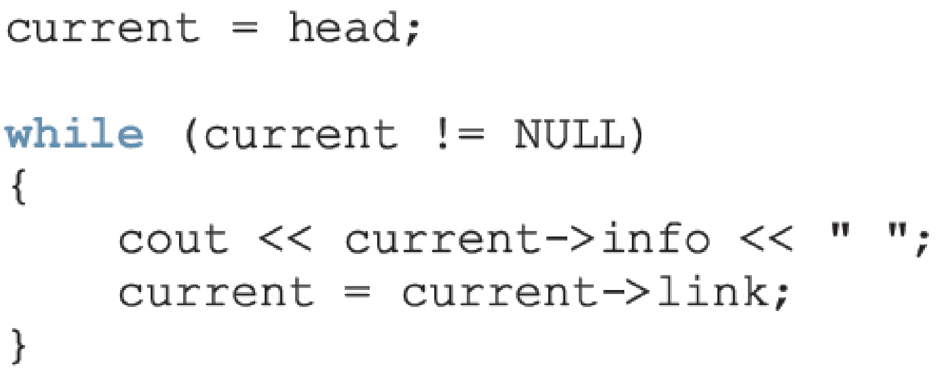
Traversing a Linked List (cont'd.)

- To traverse a linked list:



```
current = head;

while (current != NULL)
{
    //Process the current node
    current = current->link;
}
```

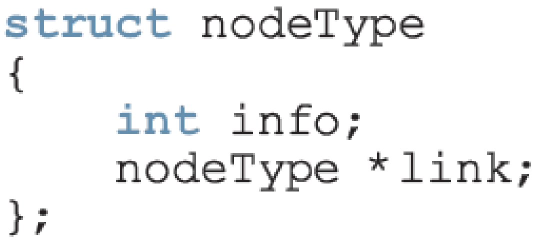
- Example:



```
current = head;

while (current != NULL)
{
    cout << current->info << " ";
    current = current->link;
}
```

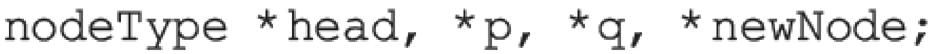
Item Insertion and Deletion

- Consider the following definition of a node:



```
struct nodeType
{
    int info;
    nodeType *link;
};
```

- We will use the following variable declaration:



```
nodeType *head, *p, *q, *newNode;
```

Insertion

- Consider the following linked list:

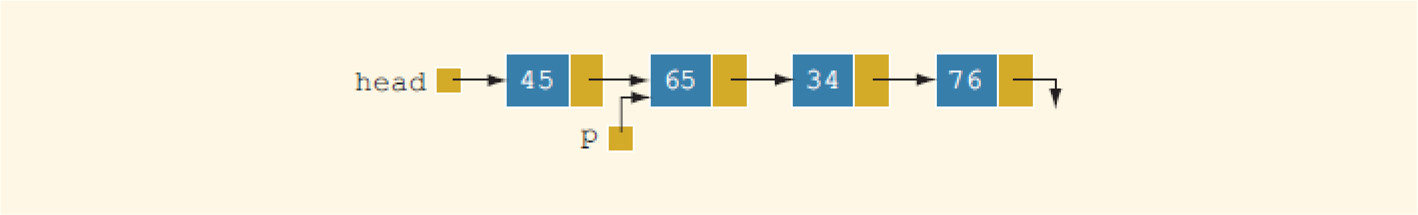
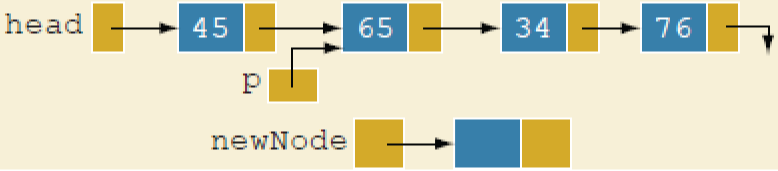
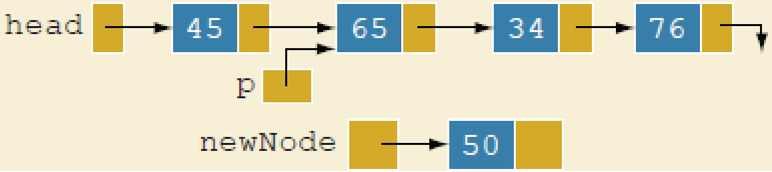
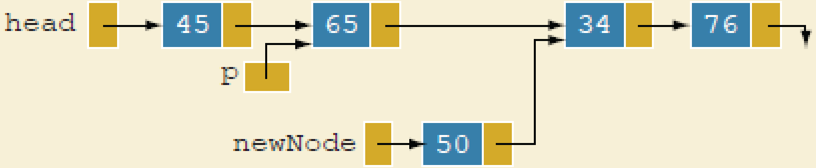
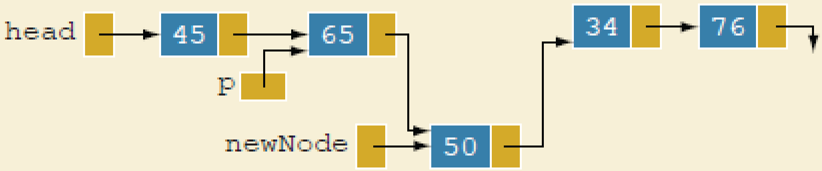


FIGURE 18-7 Linked list before item insertion

- A new node with info 50 is to be created and inserted after p

Insertion (cont'd.)

TABLE 18-1 Inserting a Node in a Linked List

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Insertion (cont'd.)

- Using two pointers, we can simplify the insertion code somewhat

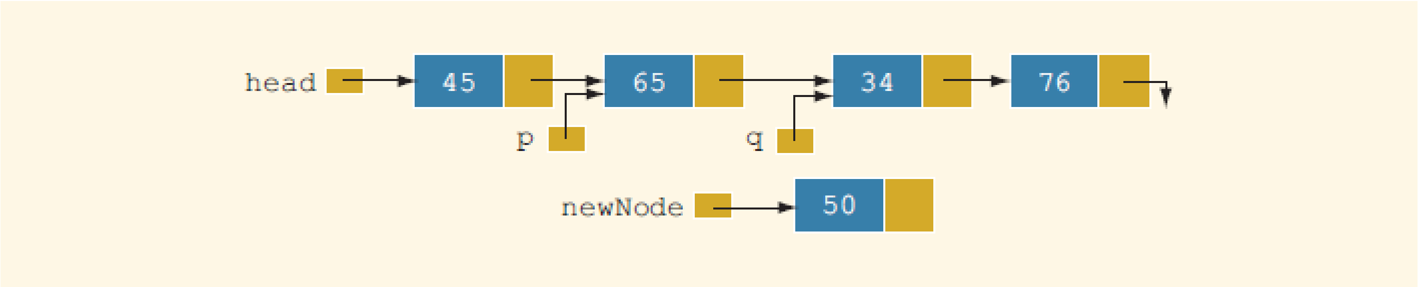


FIGURE 18-9 List with pointers p and q

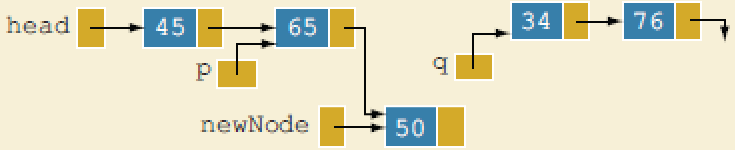
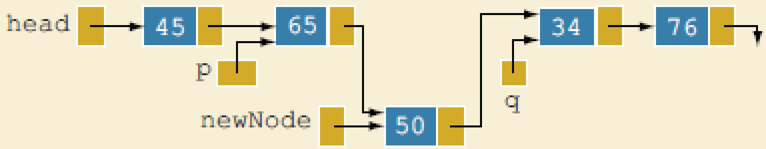
- To insert newNode between p and q:

```
newNode->link = q;
p->link = newNode;
```

The order in which these statements execute does not matter

Insertion (cont'd.)

TABLE 18-2 Inserting a Node in a Linked List Using Two Pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

Deletion

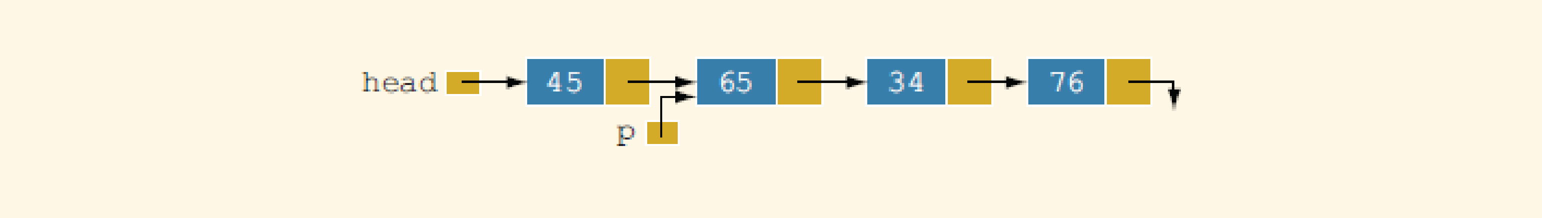


FIGURE 18-10 Node to be deleted is with info 34

```
p->link = p->link->link;
```

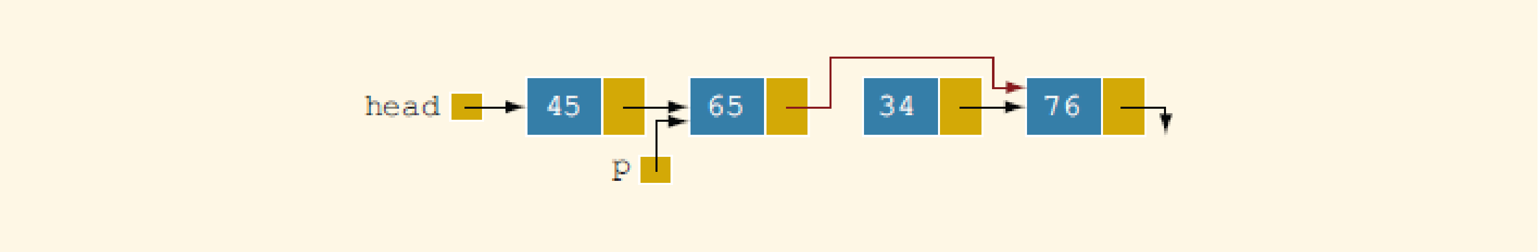


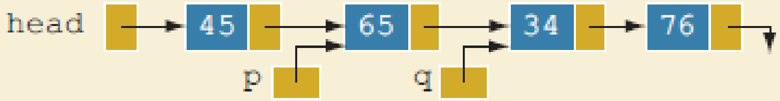
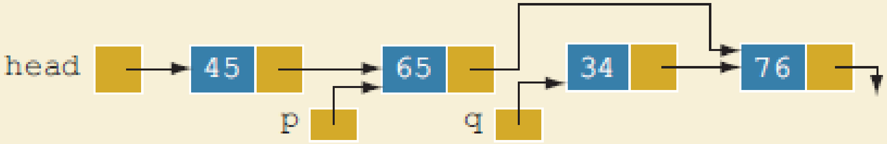
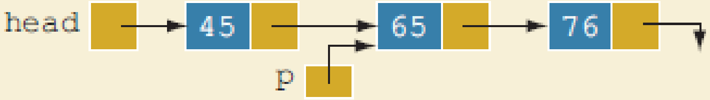
FIGURE 18-11 List after the statement newNode->link = q; executes

Node with info 34 is removed from the list, but memory is still occupied; node is dangling

Deletion (cont’d.)

```
q = p->link;
p->link = q->link;
delete q;
```

TABLE 18-3 Deleting a Node from a Linked List

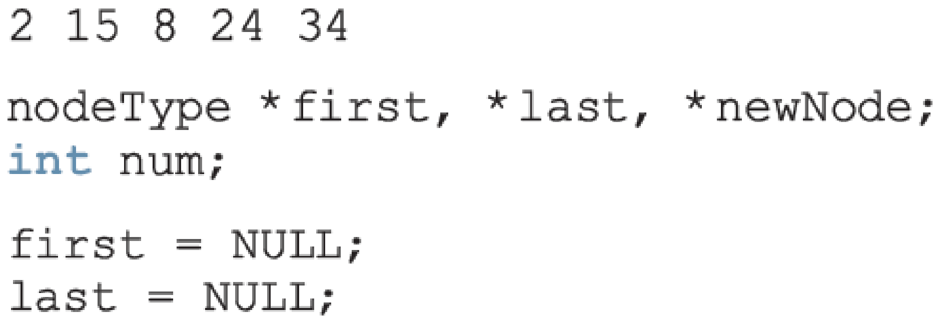
Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

Building a Linked List

- If data is unsorted
  - The list will be unsorted
- Can build a linked list forward or backward
  - Forward: a new node is always inserted at the end of the linked list
  - Backward: a new node is always inserted at the beginning of the list

Building a Linked List Forward

- You need three pointers to build the list:
  - One to point to the first node in the list, which cannot be moved
  - One to point to the last node in the list
  - One to create the new node



```
2 15 8 24 34

nodeType *first, *last, *newNode;
int num;

first = NULL;
last = NULL;
```

Building a Linked List Forward (cont'd.)

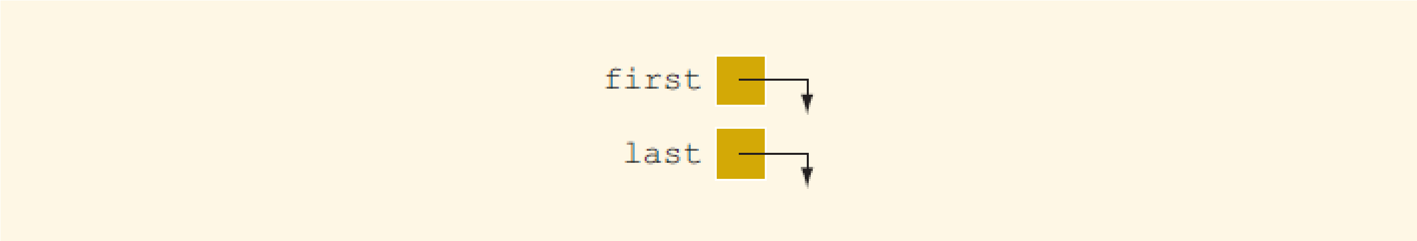


FIGURE 18-12 Empty list

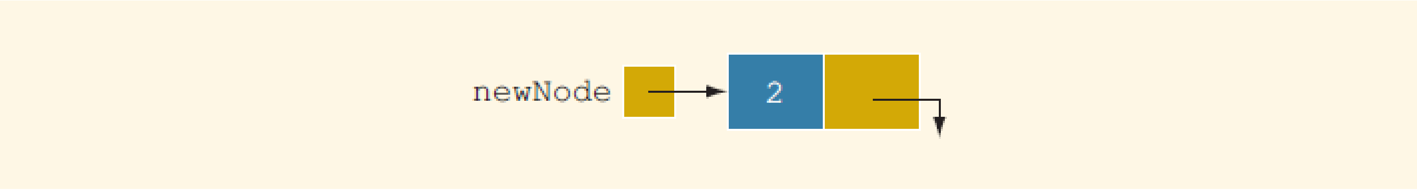


FIGURE 18-13 newNode with info 2

Building a Linked List Forward (cont'd.)

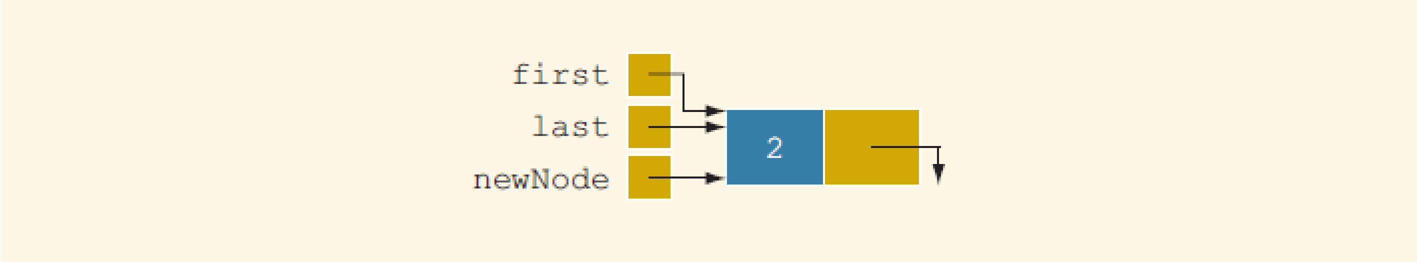


FIGURE 18-14 List after inserting newNode in it

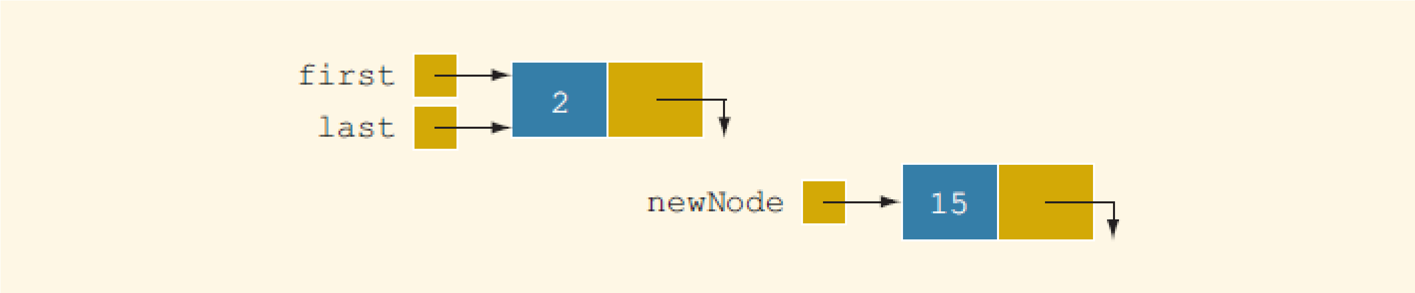


FIGURE 18-15 List and newNode with info 15

Building a Linked List Forward (cont'd.)

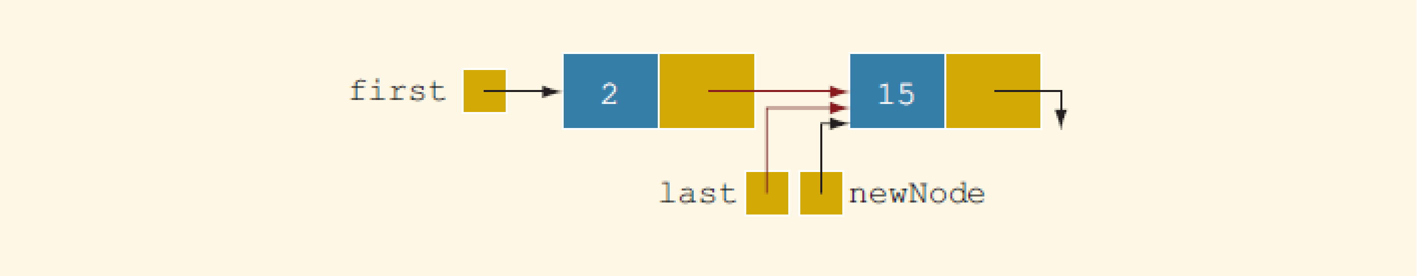


FIGURE 18-16 List after inserting newNode at the end

We now repeat statements:

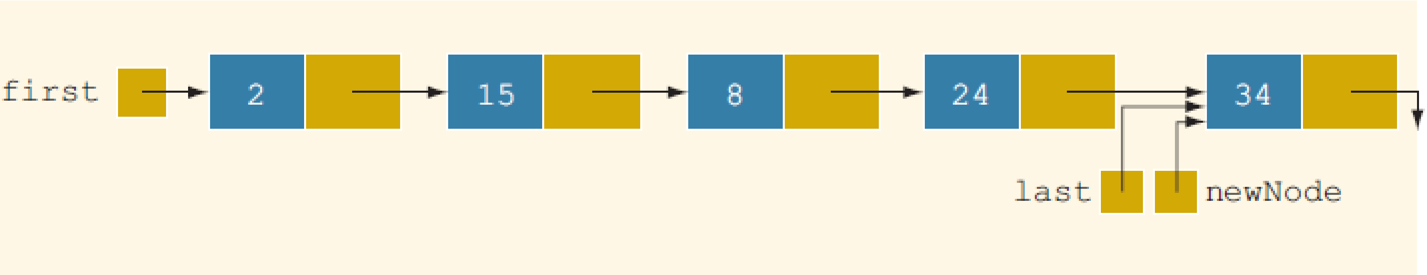
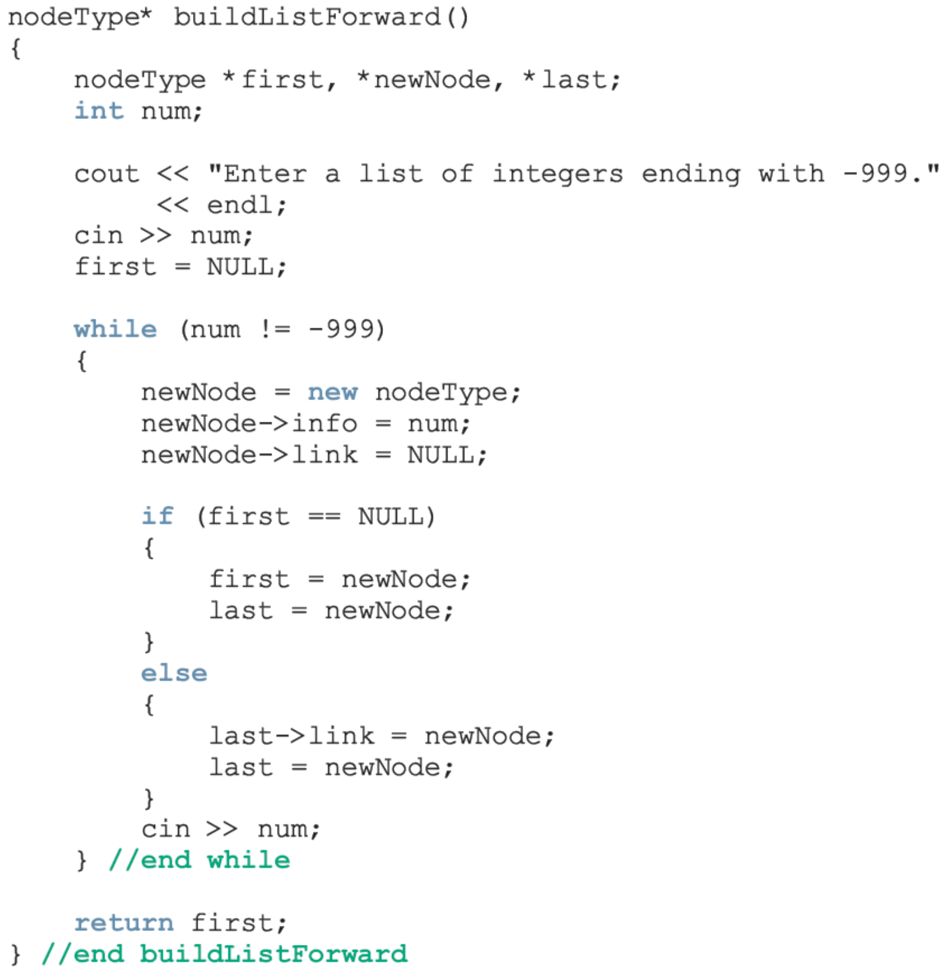


FIGURE 18-17 List after inserting 8, 24, and 34

Building a Linked List Forward (cont'd.)

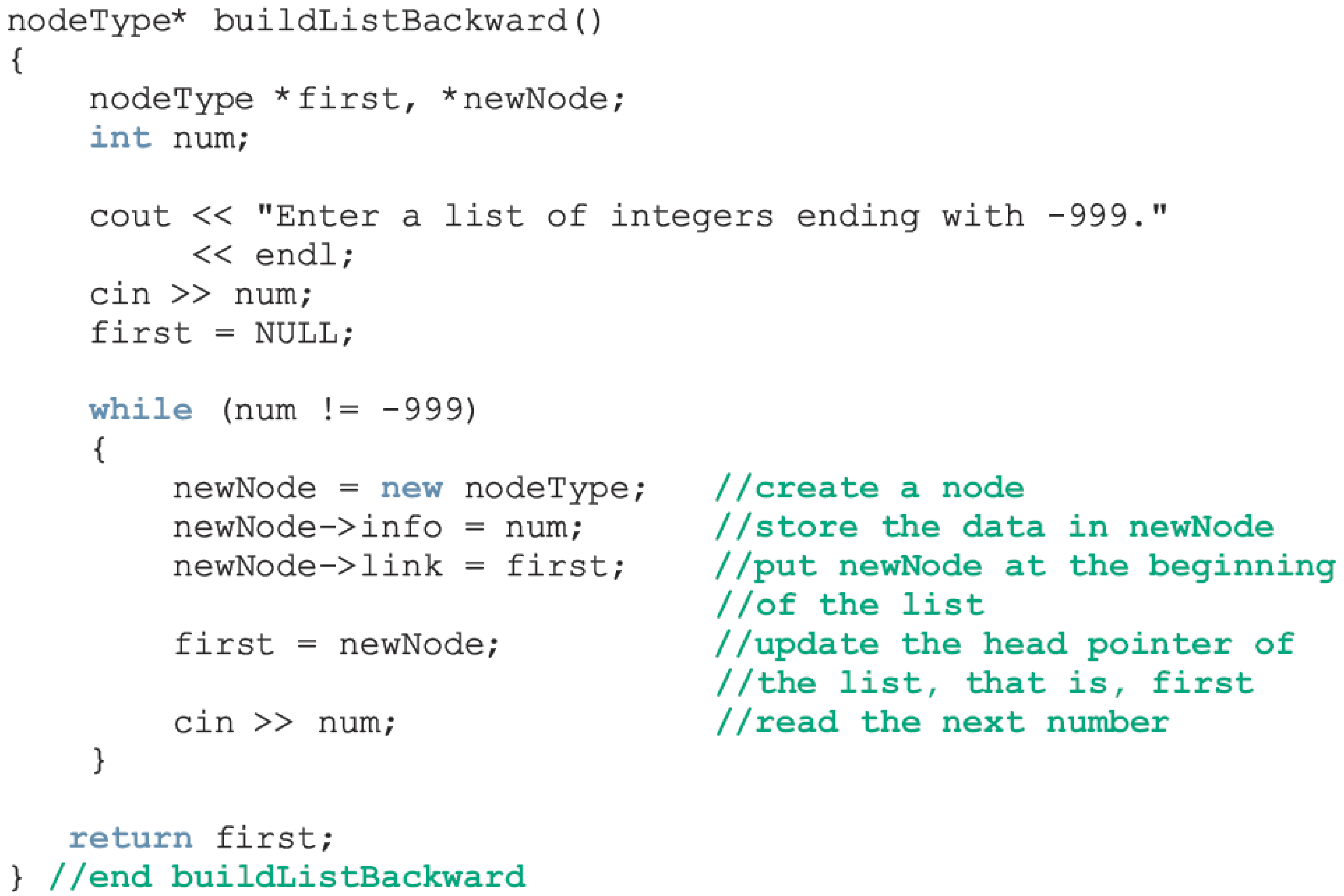


```
nodeType* buildListForward()
{
    nodeType *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
            last = newNode;
        }
        cin >> num;
    } //end while

    return first;
} //end buildListForward
```

Building a Linked List Backward

- The algorithm is:
  - Initialize first to NULL
  - For each item in the list
    - Create the new node, newNode
    - Store the item in newNode
    - Insert newNode before first
    - Update the value of the pointer first

Building a Linked List Backward (cont’d.)



```
nodeType* buildListBackward()
{
    nodeType *first, *newNode;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;   //create a node
        newNode->info = num;      //store the data in newNode
        newNode->link = first;    //put newNode at the beginning
                                  //of the list
        first = newNode;          //update the head pointer of
                                  //the list, that is, first

        cin >> num;               //read the next number
    }

   return first;
} //end buildListBackward
```

Linked List as an ADT

- The basic operations on linked lists are:
  - Initialize the list
  - Determine whether the list is empty
  - Print the list
  - Find the length of the list
  - Destroy the list

Linked List as an ADT (cont'd.)

- – Retrieve the info contained in the first node
- – Retrieve the info contained in the last node
- – Search the list for a given item
- – Insert an item in the list
- – Delete an item from the list
- – Make a copy of the linked list

Linked List as an ADT (cont'd.)

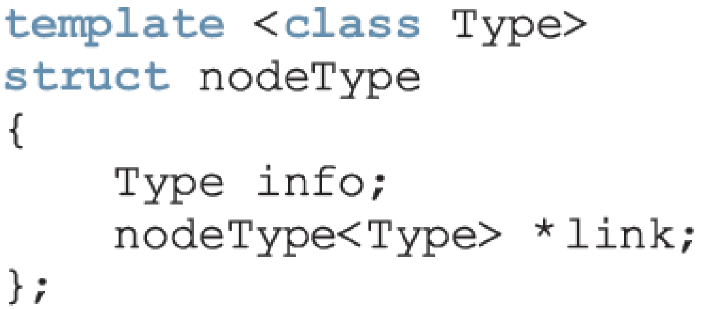
- In general, there are two types of linked lists:
  - Sorted and unsorted lists
- The algorithms to implement the operations search, insert, and remove slightly differ for sorted and unsorted lists
- The abstract class linkedList Type will implement the basic linked list operations
  - Derived Classes: unorderedLinkedList and orderedLinkedList

Linked List as an ADT (cont'd.)

- If a linked list is unordered, we can insert a new item at either the end or the beginning
  - buildListForward inserts item at the end
  - buildListBackward inserts new item at the beginning
- To accommodate both operations, we will write two functions:
  - insertFirst and insertLast
- We will use two pointers in the list:
  - first and last

Structure of Linked List Nodes

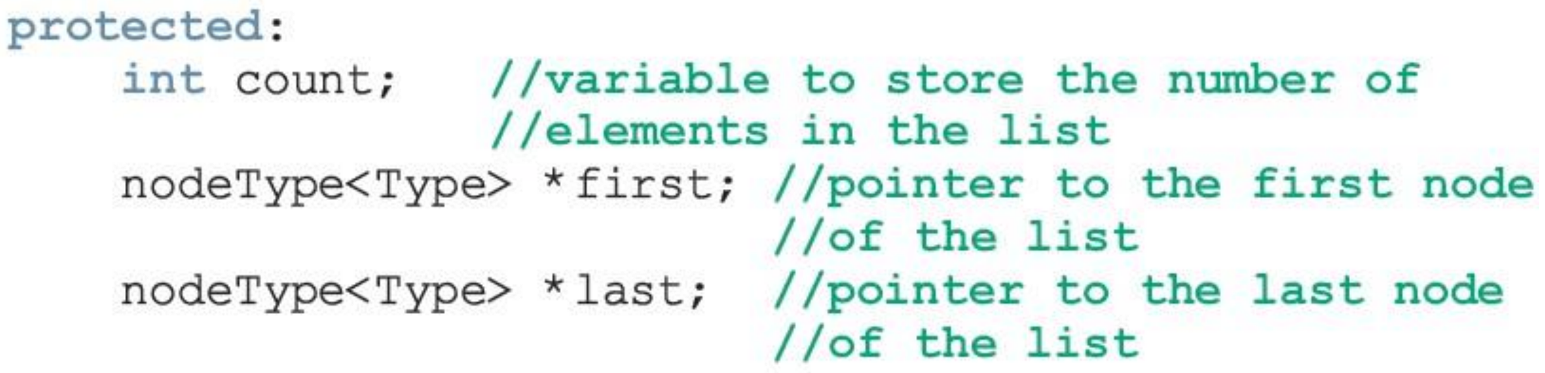
- The node has two member variables
- To simplify operations such as insert and delete, we define the class to implement the node of a linked list as a struct
- The definition of the struct nodeType is:



```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

Member Variables of the class linkedListType

- We use two pointers: first and last
- We also keep a countof the number of nodes in the list
- linkedListType has three member variables:



```
protected:
    int count;    //variable to store the number of
                  //elements in the list
    nodeType<Type> *first; //pointer to the first node
                           //of the list
    nodeType<Type> *last;  //pointer to the last node
                           //of the list
```

Linked List Iterators

- One of the basic operations performed on a list is to process each node of the list
  - List must be traversed, starting at first node
  - Common technique is to provide an iterator
- Iterator: object that produces each element of a container, one element at a time
  - The two most common operations are:

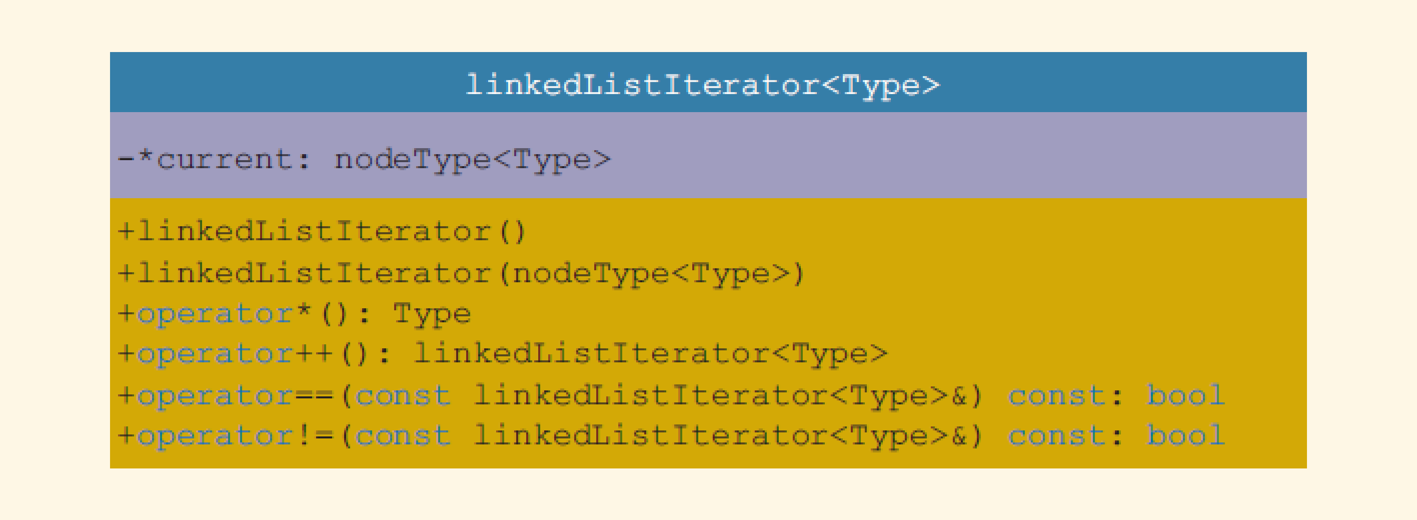
    ++(the increment operator)

    *(the dereferencing operator)

Linked List Iterators (cont'd.)

- Note that an iterator is an object
- We need to define a class (linkedListIterator) to create iterators to objects of the class linkedListType
  - Would have two member variables:
    - One to refer to (the current) node
    - One to refer to the node just before the (current) node

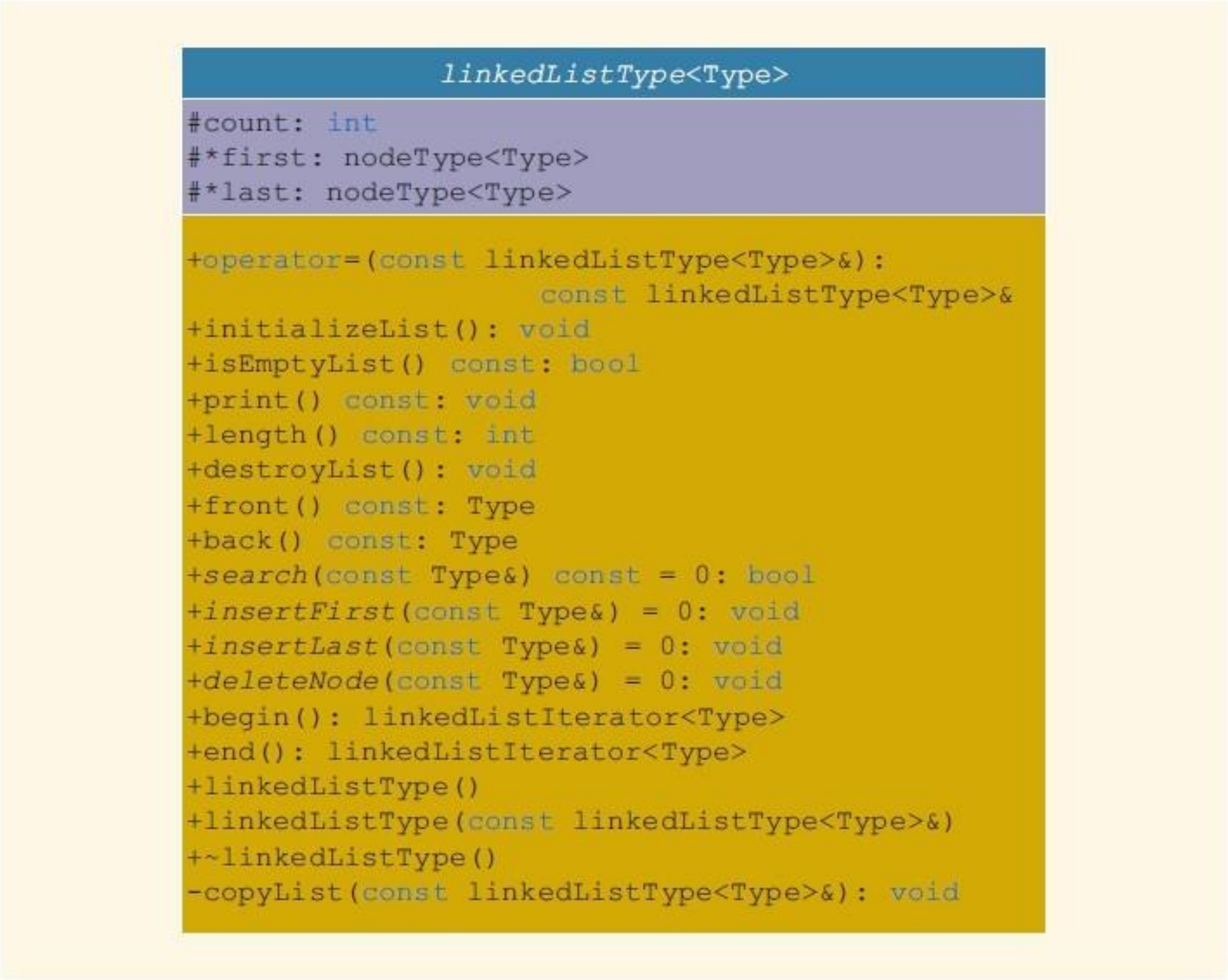
Linked List Iterators (cont'd.)



```
linkedListIterator<Type>
-*current: nodeType<Type>
+linkedListIterator()
+linkedListIterator(nodeType<Type>)
+operator*(): Type
+operator++(): linkedListIterator<Type>
+operator==(const linkedListIterator<Type>&) const: bool
+operator!=(const linkedListIterator<Type>&) const: bool
```

FIGURE 18-19 UML class diagram of the **class** `linkedListIterator`

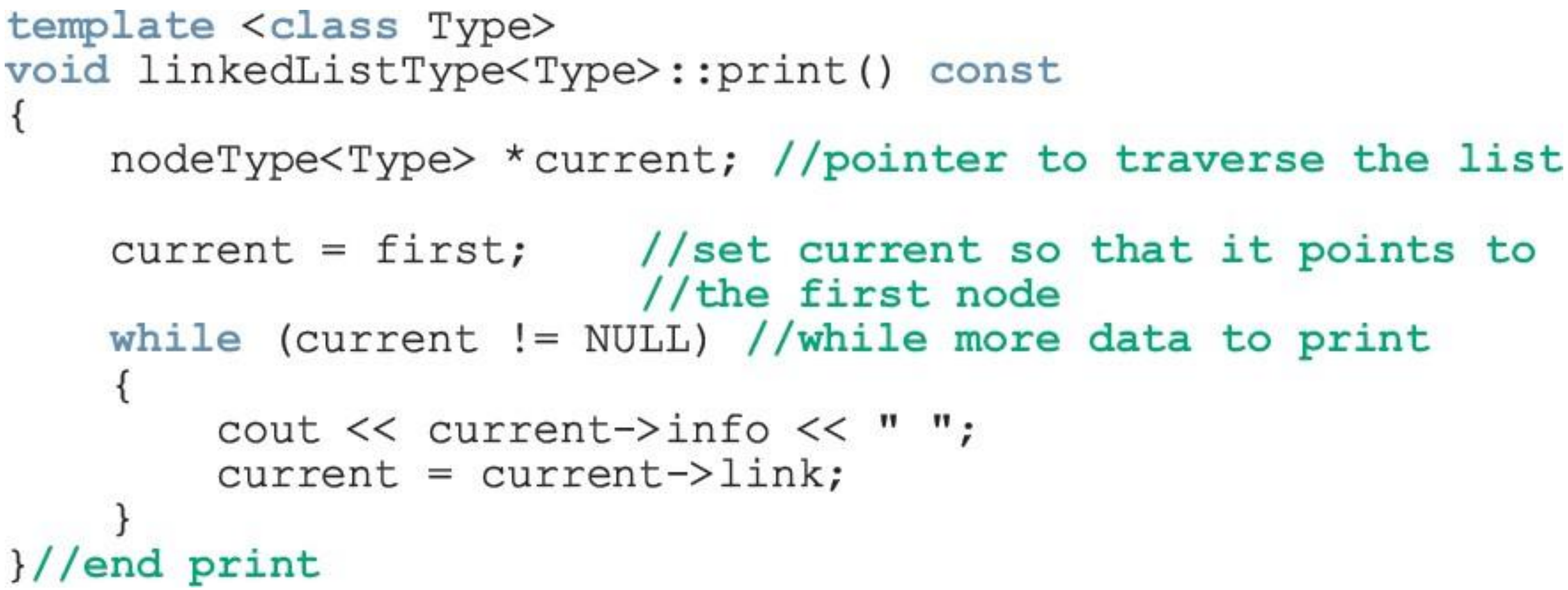
Linked List Iterators (cont'd.)



```
linkedListType<Type>

#count: int
#*first: nodeType<Type>
#*last: nodeType<Type>

+operator=(const linkedListType<Type>&):
                      const linkedListType<Type>&
+initializeList(): void
+isEmptyList() const: bool
+print() const: void
+length() const: int
+destroyList(): void
+front() const: Type
+back() const: Type
+search(const Type&) const = 0: bool
+insertFirst(const Type&) = 0: void
+insertLast(const Type&) = 0: void
+deleteNode(const Type&) = 0: void
+begin(): linkedListIterator<Type>
+end(): linkedListIterator<Type>
+linkedListType()
+linkedListType(const linkedListType<Type>&)
+~linkedListType()
-copyList(const linkedListType<Type>&): void
```

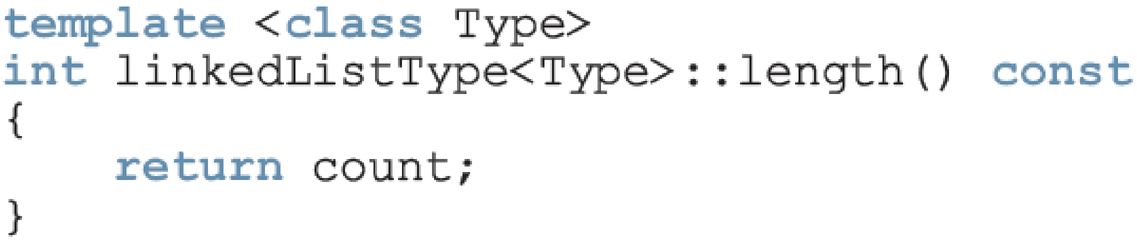
FIGURE 18-20 UML class diagram of the **class** linkedListType

Print the List



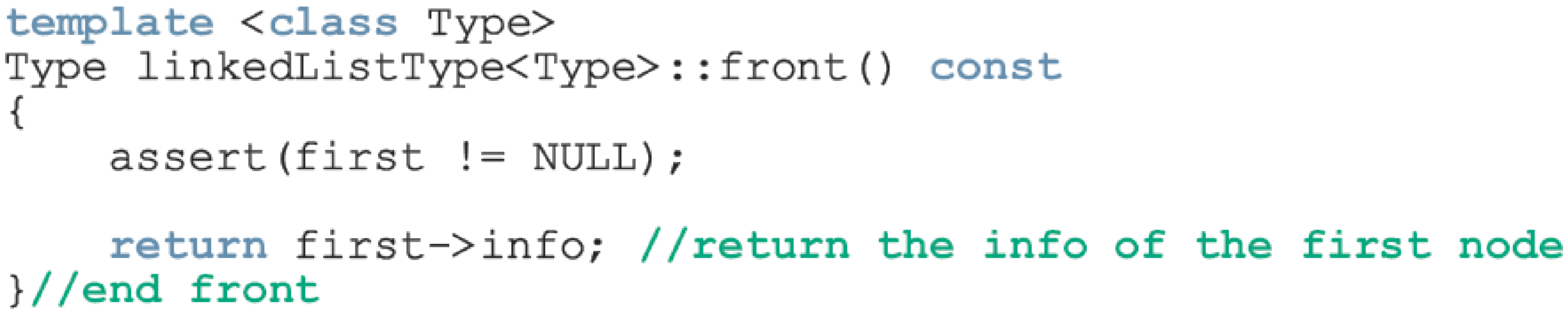
```
template <class Type>
void linkedListType<Type>::print() const
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first;    //set current so that it points to
                        //the first node
    while (current != NULL) //while more data to print
    {
        cout << current->info << " ";
        current = current->link;
    }
}//end print
```

Length of a List



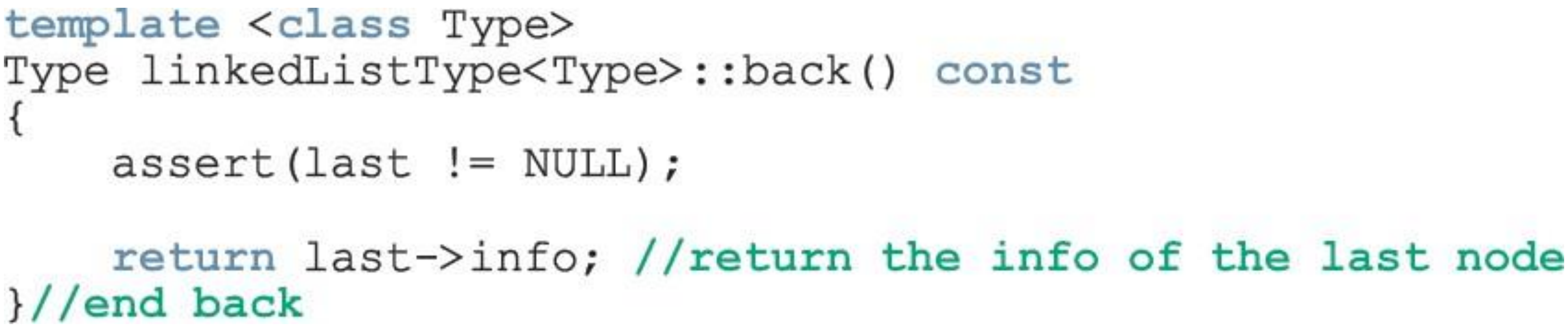
```
template <class Type>
int linkedListType<Type>::length() const
{
    return count;
}
```

Retrieve the Data of the First Node



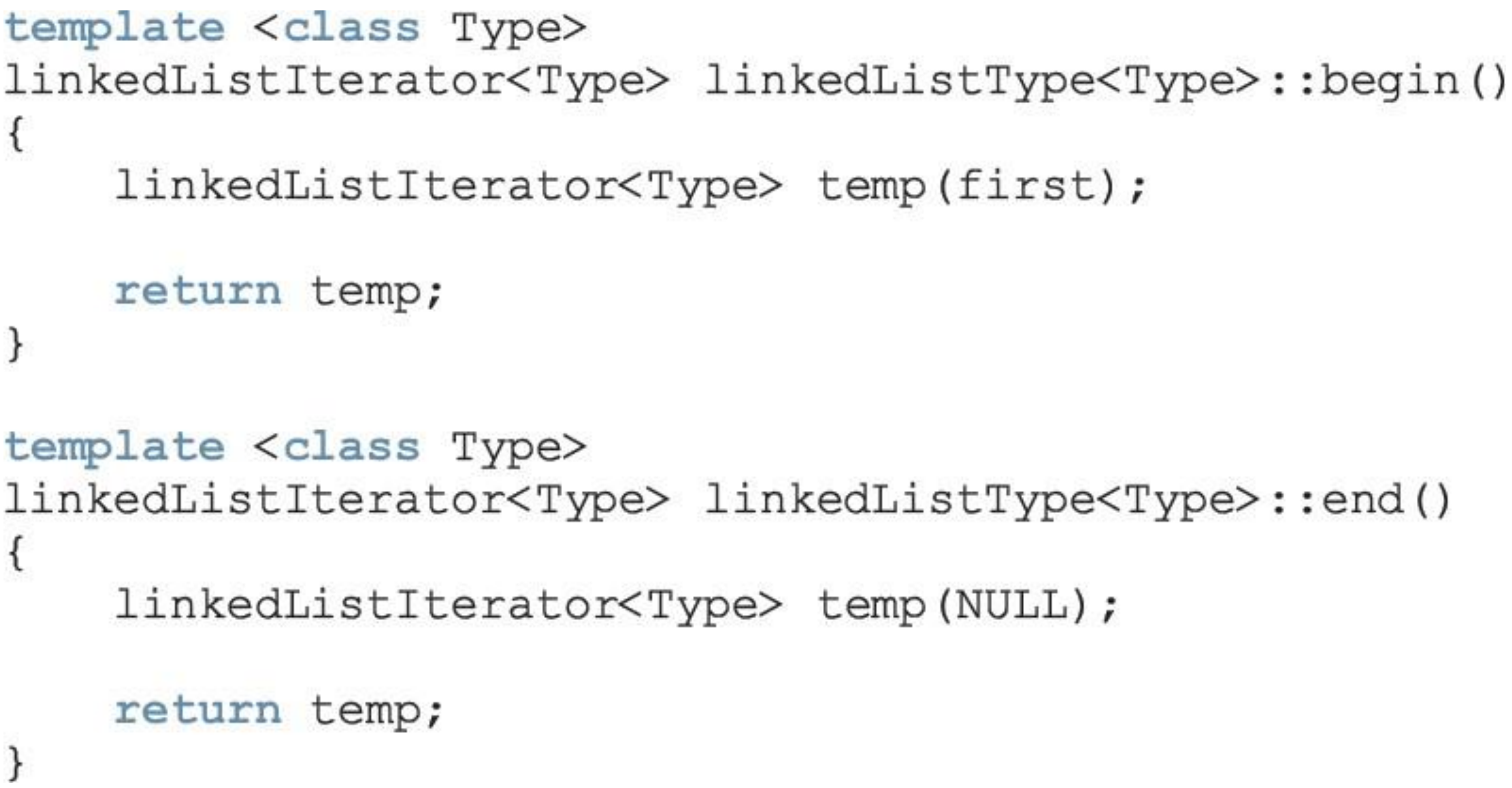
```
template <class Type>
Type linkedListType<Type>::front() const
{
    assert(first != NULL);

    return first->info; //return the info of the first node
}//end front
```

Retrieve the Data of the Last Node



```
template <class Type>
Type linkedListType<Type>::back() const
{
    assert(last != NULL);

    return last->info; //return the info of the last node
}//end back
```

Begin and End

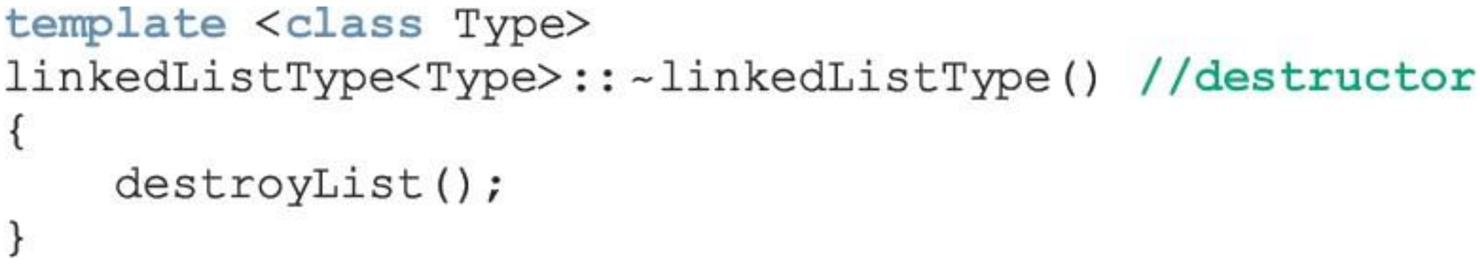


```
template <class Type>
linkedListIterator<Type> linkedListType<Type>::begin()
{
    linkedListIterator<Type> temp(first);

    return temp;
}

template <class Type>
linkedListIterator<Type> linkedListType<Type>::end()
{
    linkedListIterator<Type> temp(NULL);

    return temp;
}
```

Copy the List

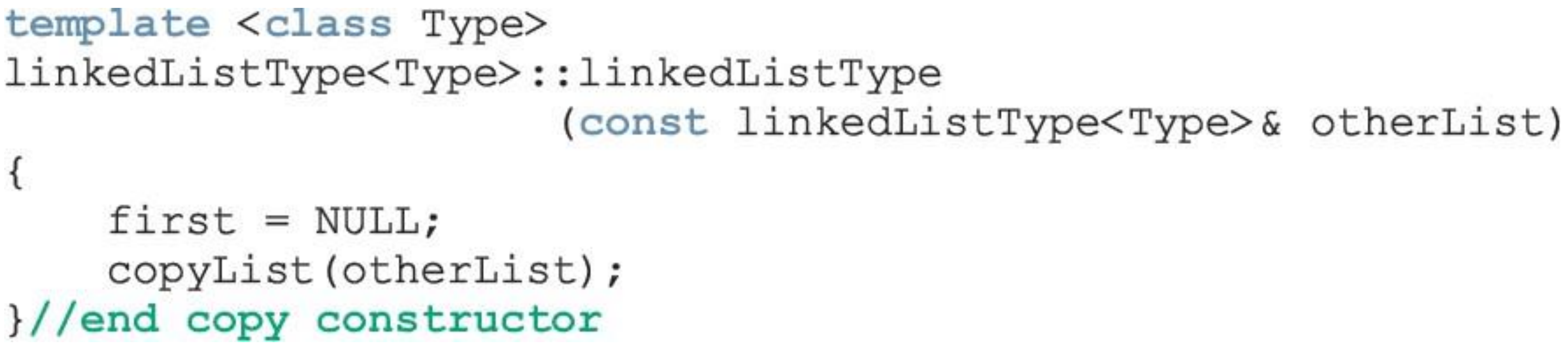
- Steps:
  - Create a node, and call it newNode
  - Copy the info of the node (in the original list) into newNode
  - Insert newNode at the end of the list being created

Destructor



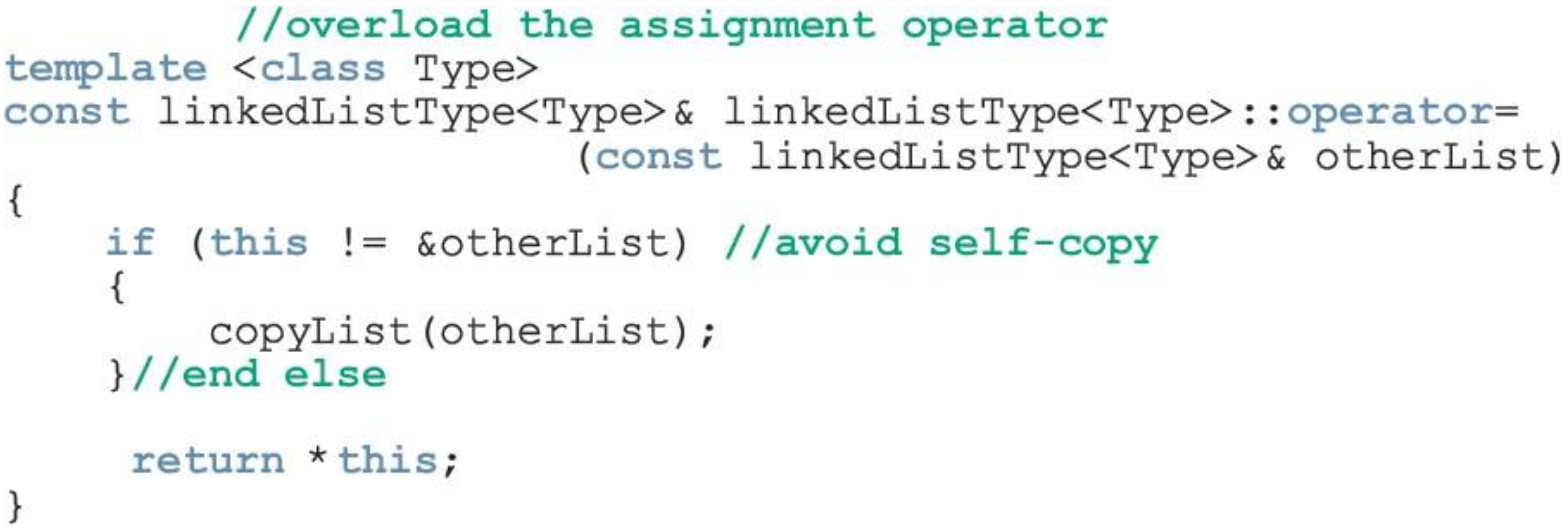
```
template <class Type>
linkedListType<Type>::~linkedListType() //destructor
{
    destroyList();
}
```

Copy Constructor



```
template <class Type>
linkedListType<Type>::linkedListType
                      (const linkedListType<Type>& otherList)
{
    first = NULL;
    copyList(otherList);
}//end copy constructor
```

Overloading the Assignment Operator

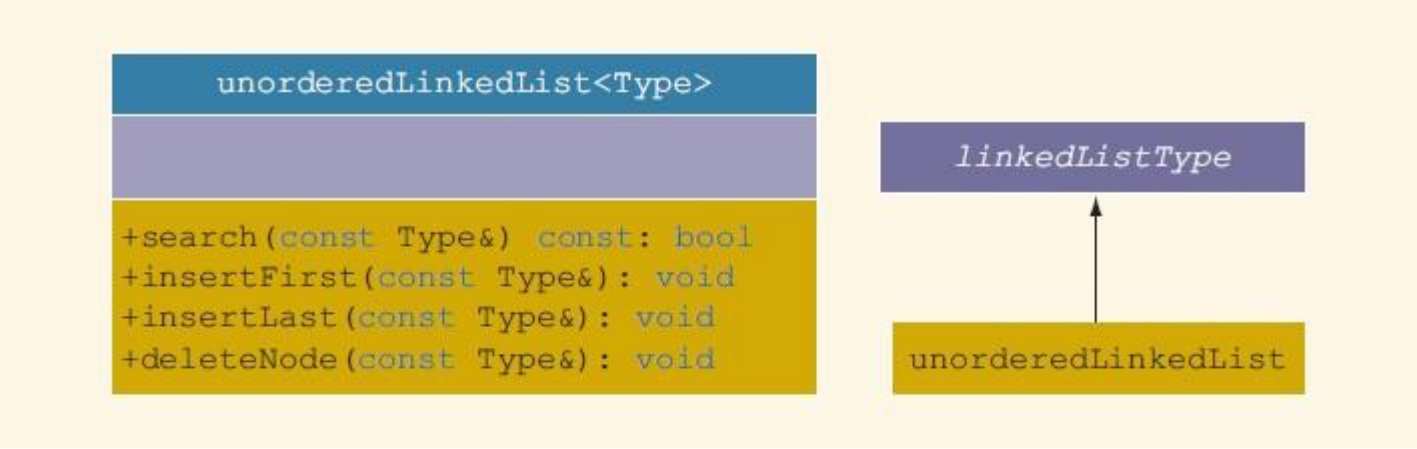


```
         //overload the assignment operator
template <class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                       (const linkedListType<Type>& otherList)
{
    if (this != &otherList) //avoid self-copy
    {
        copyList(otherList);
    }//end else

     return *this;
}
```

Unordered Linked Lists

- Derive the class unorderedLinkedList from the abstract class linkedListType and implement the operations search, insertFirst, insertLast, and deleteNode

Unordered Linked Lists (cont'd.)



unorderedLinkedList<Type>

+search(const Type&) const: bool
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void

linkedListType

unorderedLinkedList

FIGURE 18-21 UML class diagram of the **class** unorderedLinkedList and inheritance hierarchy

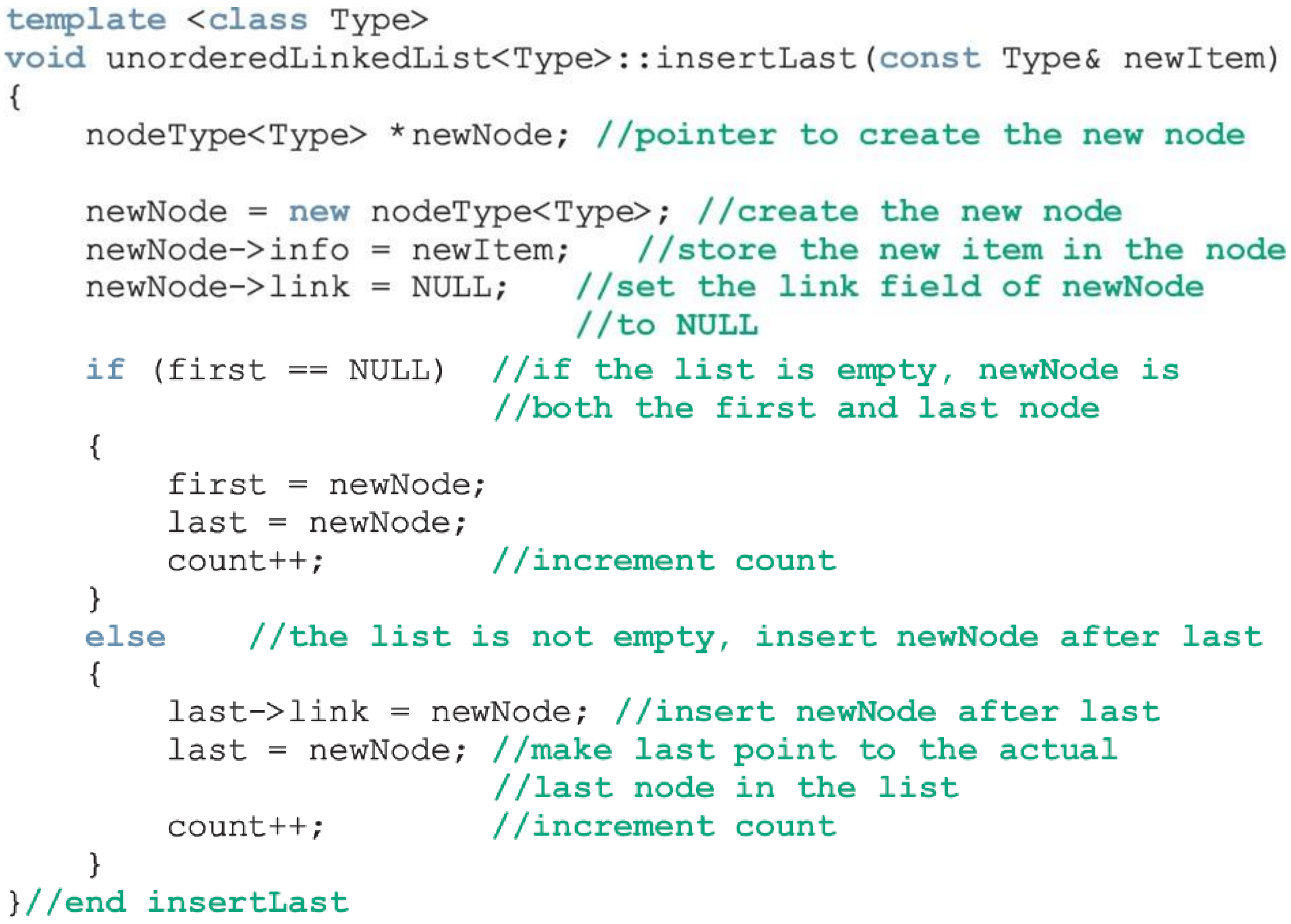
Search the List

- Steps:
  - Compare the search item with the current node in the list
    - If the info of the current node is the same as the search item, stop the search
    - Otherwise, make the next node the current node
  - Repeat Step 1 until the item is found
    - Or, until no more data is left in the list to compare with the search item

Insert the First Node

- Steps:
  - Create a new node
  - Store the new item in the new node
  - Insert the node before first
  - Increment count by 1

Insert the Last Node



```
template <class Type>
void unorderedLinkedList<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem;   //store the new item in the node
    newNode->link = NULL;   //set the link field of newNode
                            //to NULL
    if (first == NULL)  //if the list is empty, newNode is
                        //both the first and last node
    {
        first = newNode;
        last = newNode;
        count++;        //increment count
    }
    else    //the list is not empty, insert newNode after last
    {
        last->link = newNode; //insert newNode after last
        last = newNode; //make last point to the actual
                        //last node in the list
        count++;        //increment count
    }
}//end insertLast
```

Delete a Node

- Case 1: List is empty
  - – If the list is empty, output an error message
- Case 2: List is not empty
  - – The node to be deleted is the first node
  - – First scenario: List has only one node

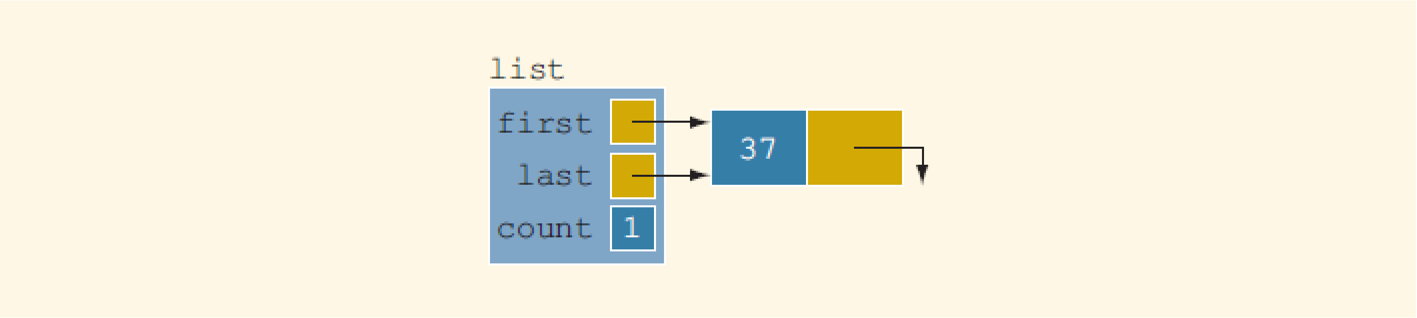


FIGURE 18-22 list with one node

Delete a Node (cont'd.)

– Second scenario: List of more than one node

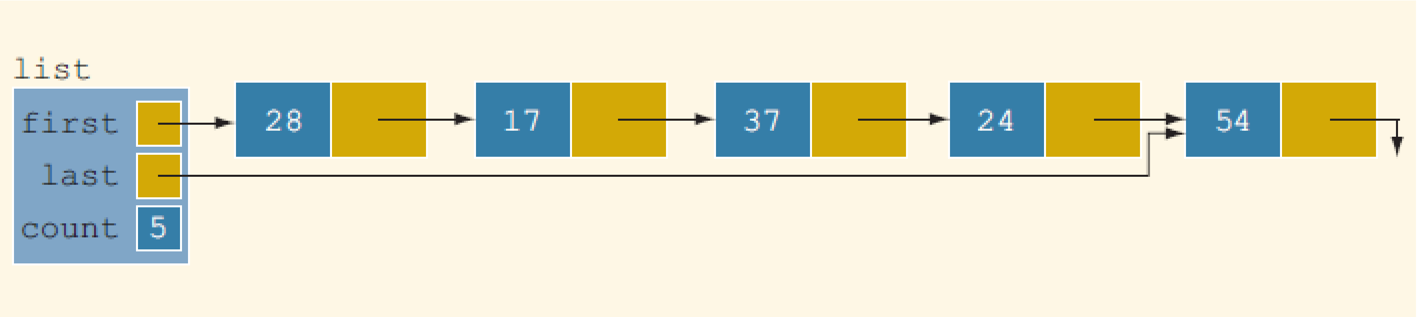


FIGURE 18-23 list with more than one node

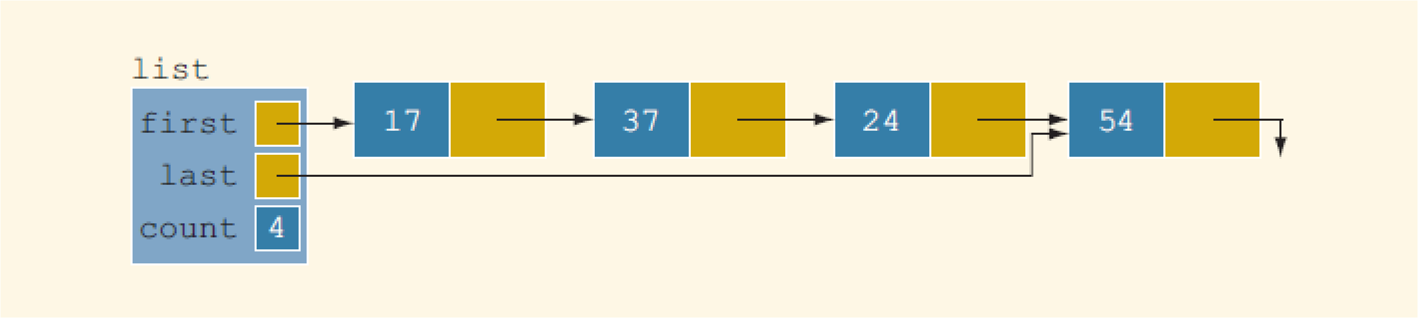


FIGURE 18-24 list after deleting node with info 28

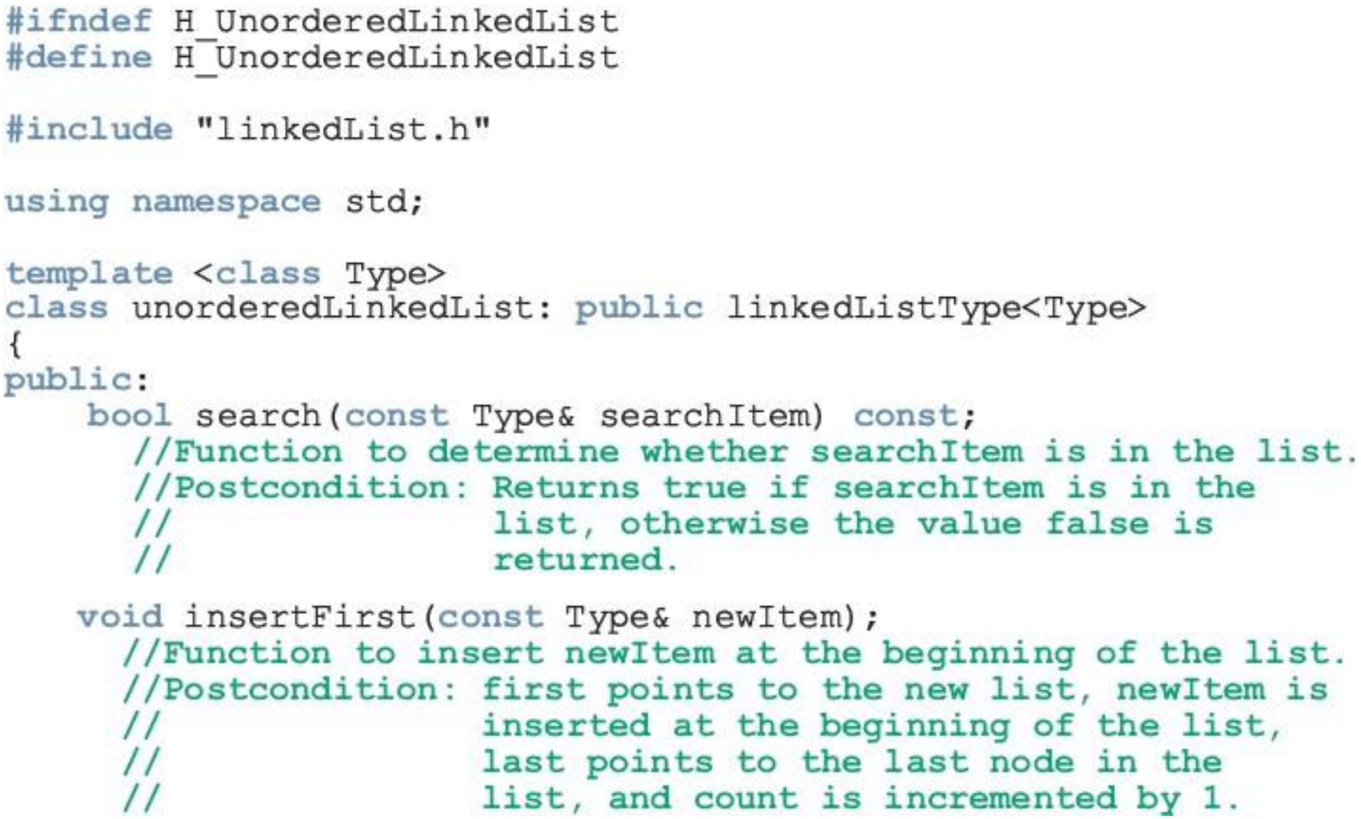
Delete a Node (cont'd.)

- Case 3: Node to be deleted is not the first one
  - Case 3a: Node to be deleted is not last one
  - Case 3b: Node to be deleted is the last node

Delete a Node (cont'd.)

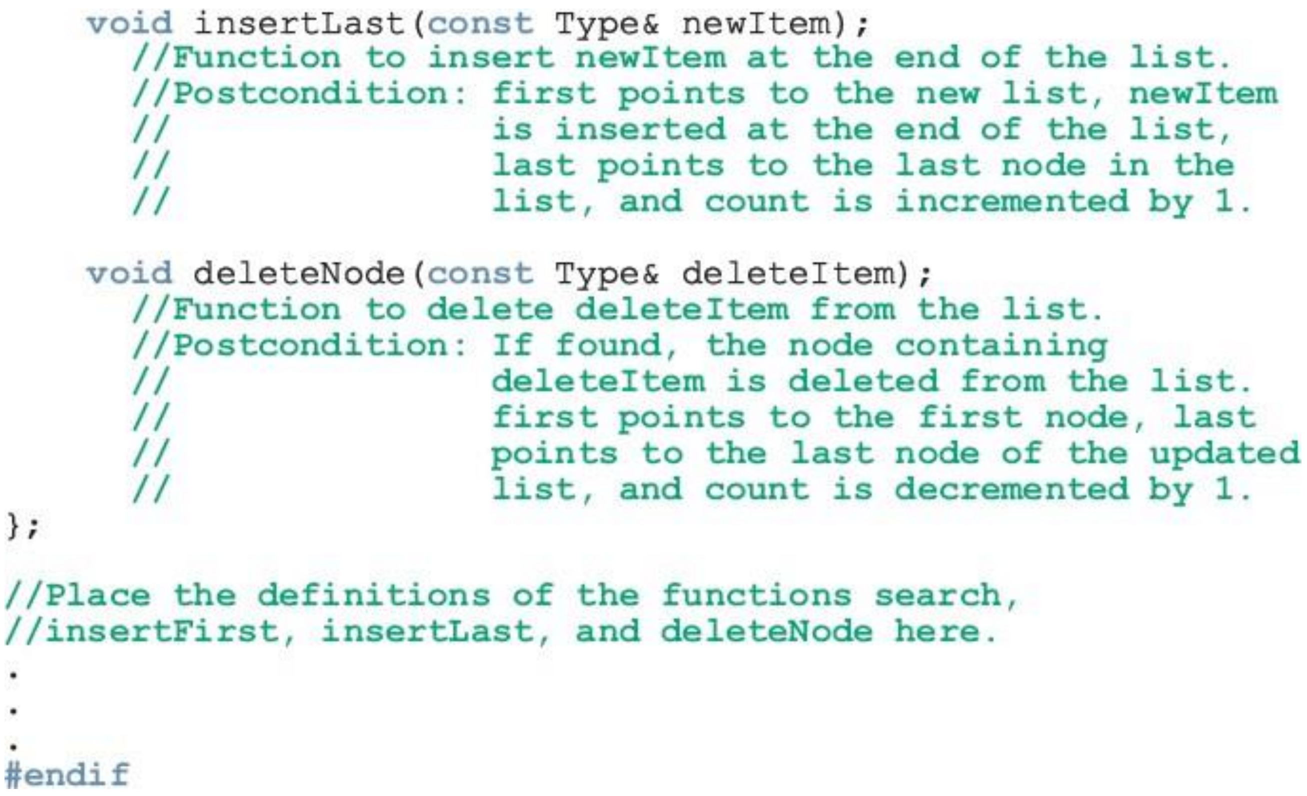
- Case 4: Node to be deleted is not in the list
  - The list requires no adjustment
  - Simply output an error message

Header File of the Unordered Linked List



```
#ifndef H_UnorderedLinkedList
#define H_UnorderedLinkedList

#include "linkedList.h"

using namespace std;

template <class Type>
class unorderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the
      //               list, otherwise the value false is
      //               returned.

    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the beginning of the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.
```

Header File of the Unordered Linked List (cont'd.)



```
    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the end of the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void deleteNode(const Type& deleteItem);
      //Function to delete deleteItem from the list.
      //Postcondition: If found, the node containing
      //               deleteItem is deleted from the list.
      //               first points to the first node, last
      //               points to the last node of the updated
      //               list, and count is decremented by 1.
};

//Place the definitions of the functions search,
//insertFirst, insertLast, and deleteNode here.
.
.
.
#endif
```

Ordered Linked Lists

- orderedLinkedList is derived from class linkedList Type
  - Provide the definitions of the abstract functions insertFirst, insertLast, search, and deleteNode
- Assume that elements of an ordered linked list are arranged in ascending order
- Include the function insert to insert an element in an ordered list at the proper place

Ordered Linked Lists (cont'd.)

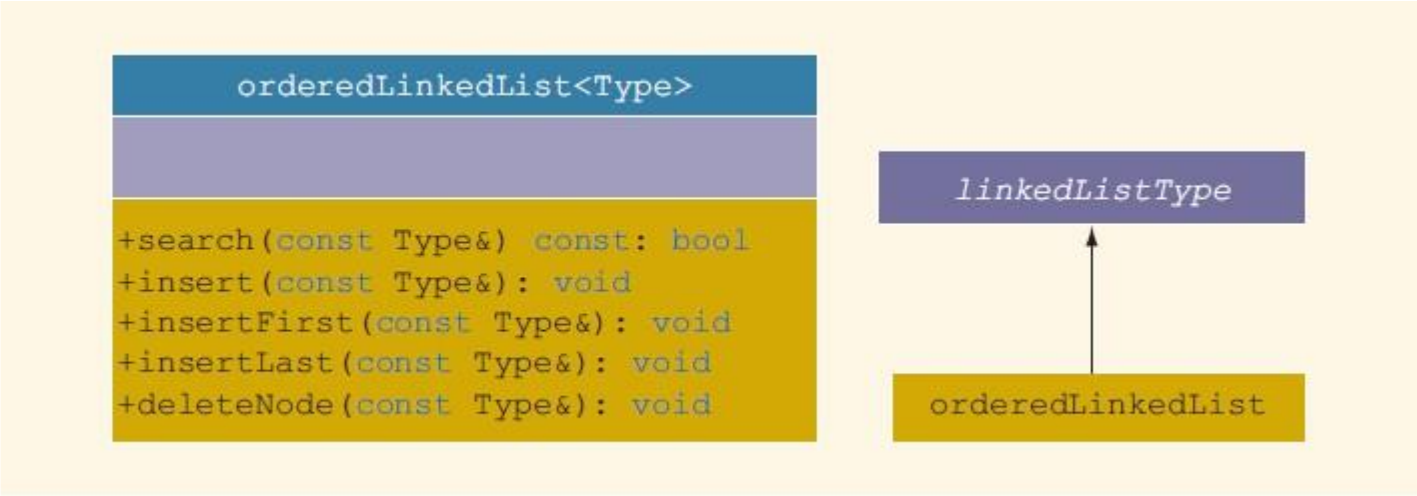

orderedLinkedList<Type>
+search(const Type&) const: bool
+insert(const Type&): void
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void
linkedListType
orderedLinkedList

FIGURE 18-29 UML class diagram of the **class** orderedLinkedList and the inheritance hierarchy

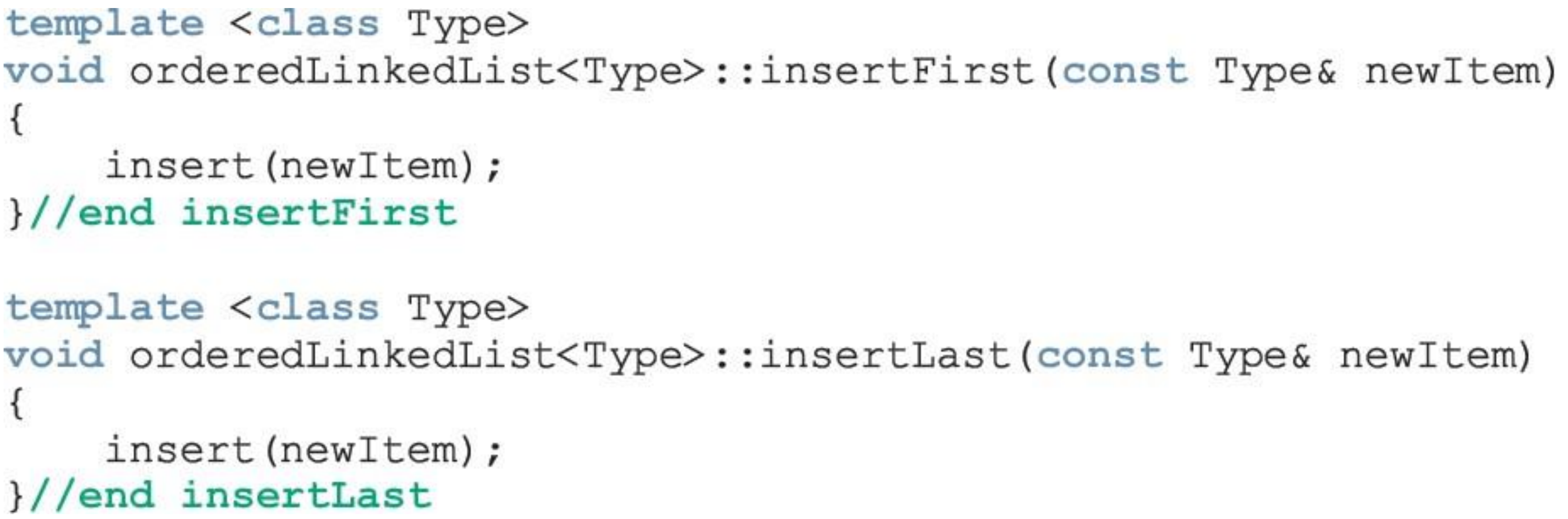
Search the List

- Steps:
  - – Compare the search item with the current node in the list
    - If the info of the current node is >= to the search item, stop the search
    - Otherwise, make the next node the current node
  - – Repeat Step 1 until an item in the list that >= to the search item is found
    - Or, until no more data is left in the list to compare with the search item

Insert a Node

- Case 1: The list is empty
- Case 2: List is not empty, and the item to be inserted is smaller than smallest item in list
- Case 3: New item is larger than first item
  - Case 3a: New item is larger than largest item
  - Case 3b: Item to be inserted goes somewhere in the middle of the list

Insert First and Insert Last

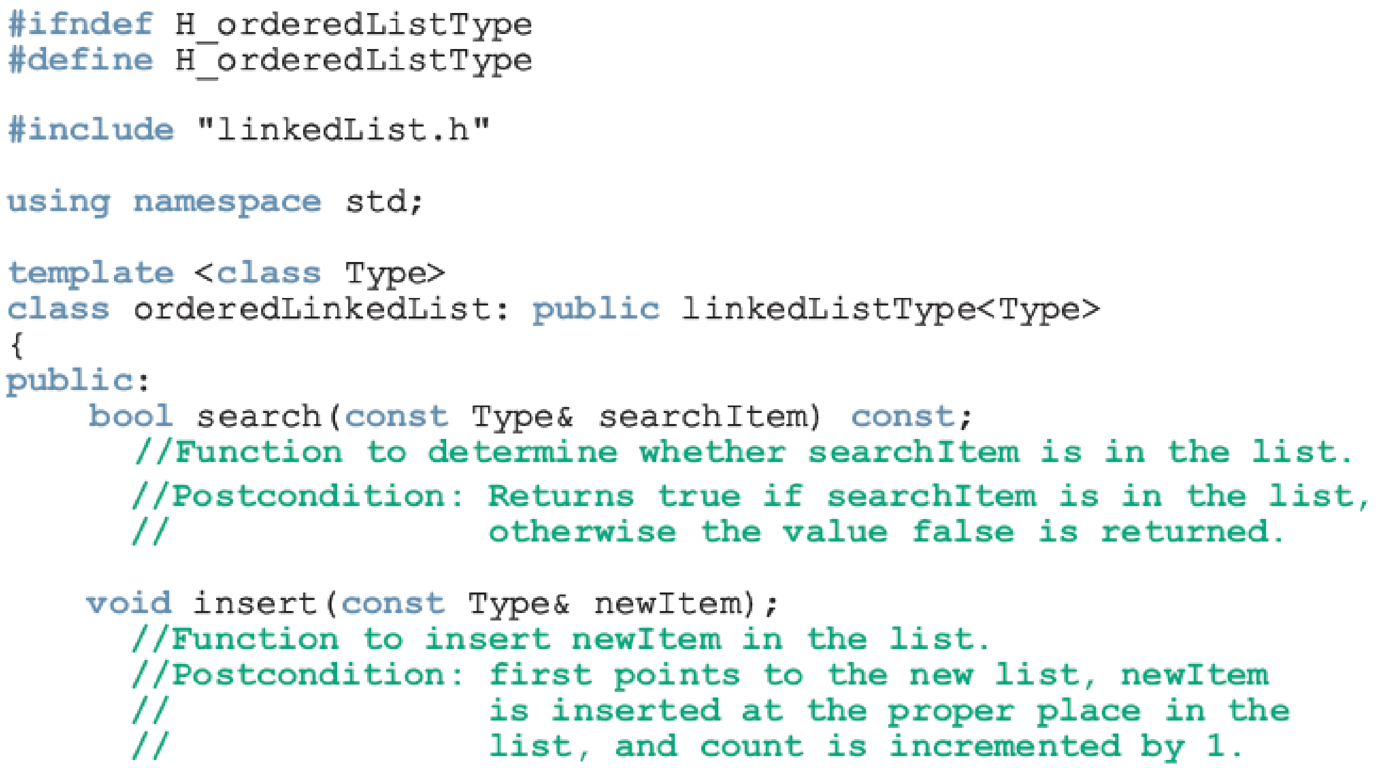


```
template <class Type>
void orderedLinkedList<Type>::insertFirst(const Type& newItem)
{
    insert(newItem);
}//end insertFirst

template <class Type>
void orderedLinkedList<Type>::insertLast(const Type& newItem)
{
    insert(newItem);
}//end insertLast
```

Delete a Node

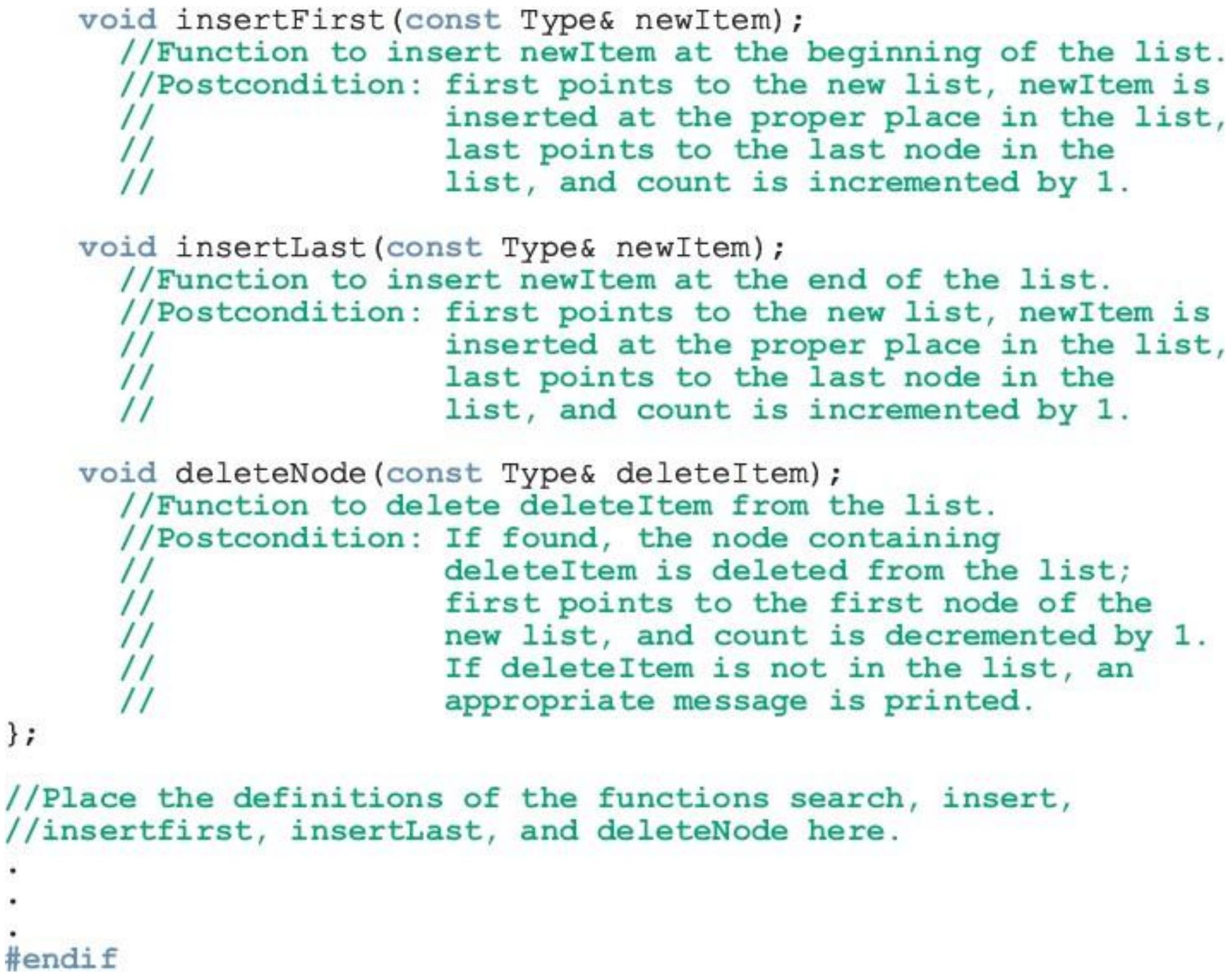
- Case 1: List is initially empty → Error
- Case 2: Item to be deleted is contained in the first node of the list
  - We must adjust the head (first) pointer
- Case 3: Item is somewhere in the list
  - currentpoints to node containing item to be deleted; trailCurrent points to the node just before the one pointed to by current
- Case 4: Item is not in the list → Error

Header File of the Ordered Linked List



```
#ifndef H_orderedListType
#define H_orderedListType

#include "linkedList.h"

using namespace std;

template <class Type>
class orderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the list,
      //               otherwise the value false is returned.

    void insert(const Type& newItem);
      //Function to insert newItem in the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the proper place in the
      //               list, and count is incremented by 1.
```

Header File of the Ordered Linked List (cont’d.)



```
    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the proper place in the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the proper place in the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void deleteNode(const Type& deleteItem);
      //Function to delete deleteItem from the list.
      //Postcondition: If found, the node containing
      //               deleteItem is deleted from the list;
      //               first points to the first node of the
      //               new list, and count is decremented by 1.
      //               If deleteItem is not in the list, an
      //               appropriate message is printed.
};

//Place the definitions of the functions search, insert,
//insertfirst, insertLast, and deleteNode here.
.
.
.
#endif
```

Print a Linked List in Reverse Order (Recursion Revisited)

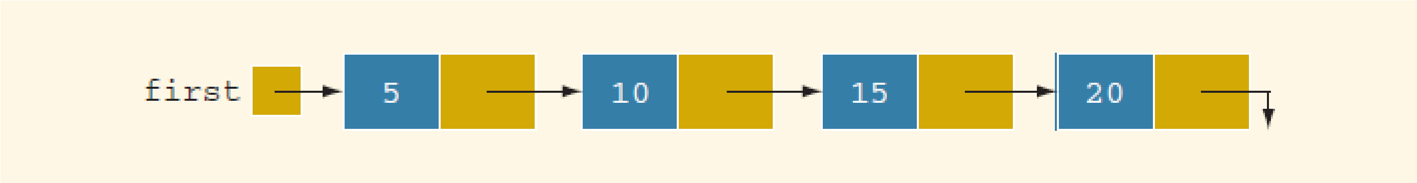
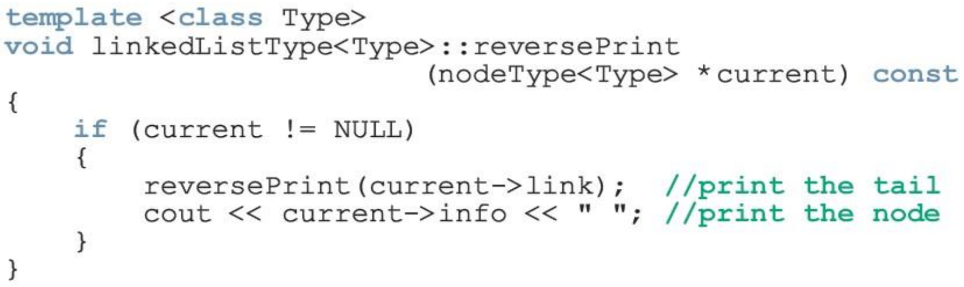


FIGURE 18-37 Linked list

For the list in Figure 18-37, the output should be in the following form:

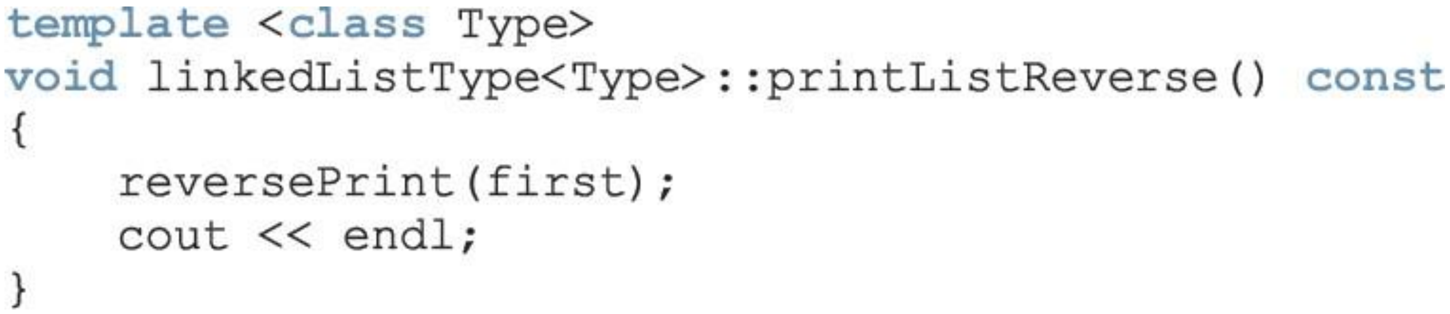
```
20 15 10 5
```

- Assume currentis a pointer to a linked list:



```
template <class Type>
void linkedListType<Type>::reversePrint
                        (nodeType<Type> *current) const
{
    if (current != NULL)
    {
        reversePrint(current->link);   //print the tail
        cout << current->info << " ";  //print the node
    }
}
```

printListReverse



```
template <class Type>
void linkedListType<Type>::printListReverse() const
{
    reversePrint(first);
    cout << endl;
}
```

Doubly Linked Lists

- Doubly linked list: every node has next and back pointers

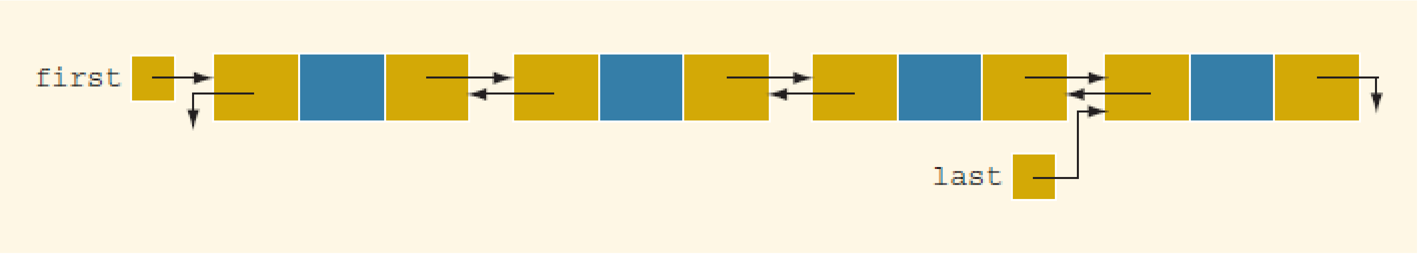


FIGURE 18-39 Doubly linked list

- Can be traversed in either direction

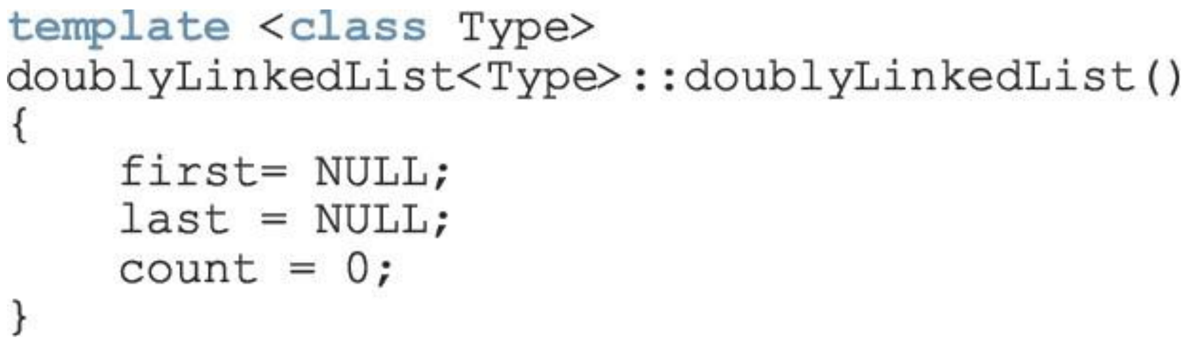
Doubly Linked Lists (cont'd.)

- Operations:
  - Initialize the list
  - Destroy the list
  - Determine whether the list is empty
  - Search the list for a given item
  - Retrieve the first element of the list
  - Retrieve the last element of the list

Doubly Linked Lists (cont'd.)

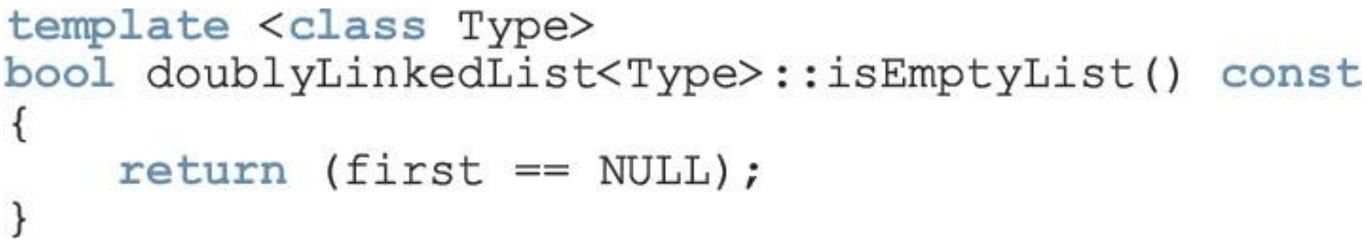
- Insert an item in the list
- Delete an item from the list
- Find the length of the list
- Print the list
- Make a copy of the doubly linked list

Default Constructor



```
template <class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
    first= NULL;
    last = NULL;
    count = 0;
}
```

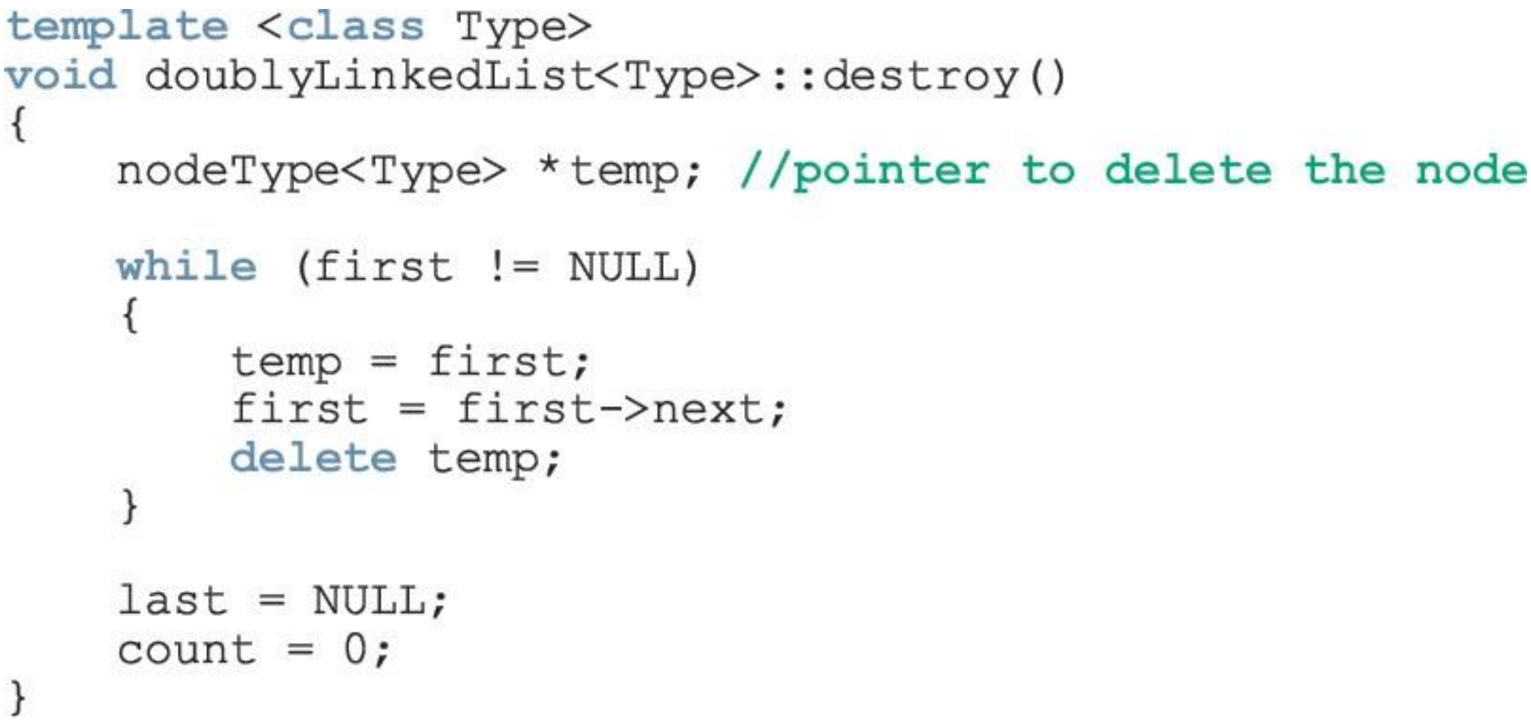
isEmptyList



```
template <class Type>
bool doublyLinkedList<Type>::isEmptyList() const
{
    return (first == NULL);
}
```

Destroy the List

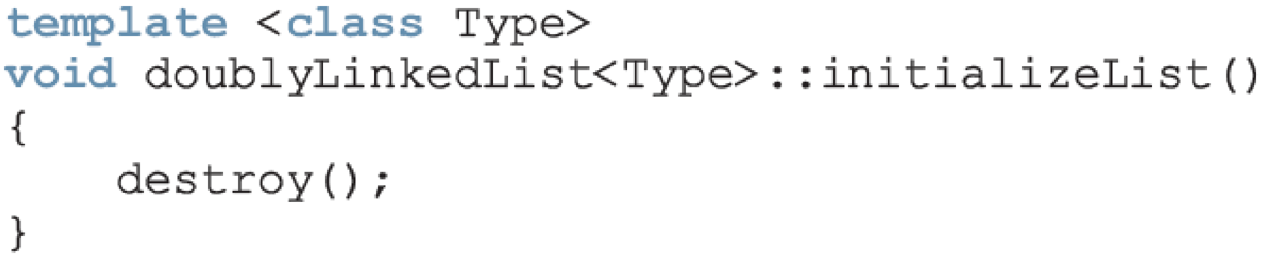
- This operation deletes all the nodes in the list, leaving the list in an empty state



```
template <class Type>
void doublyLinkedList<Type>::destroy()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (first != NULL)
    {
        temp = first;
        first = first->next;
        delete temp;
    }

    last = NULL;
    count = 0;
}
```

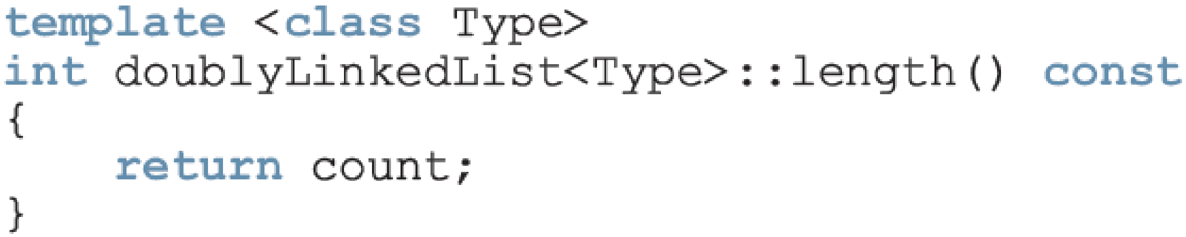
Initialize the List

- This operation reinitializes the doubly linked list to an empty state



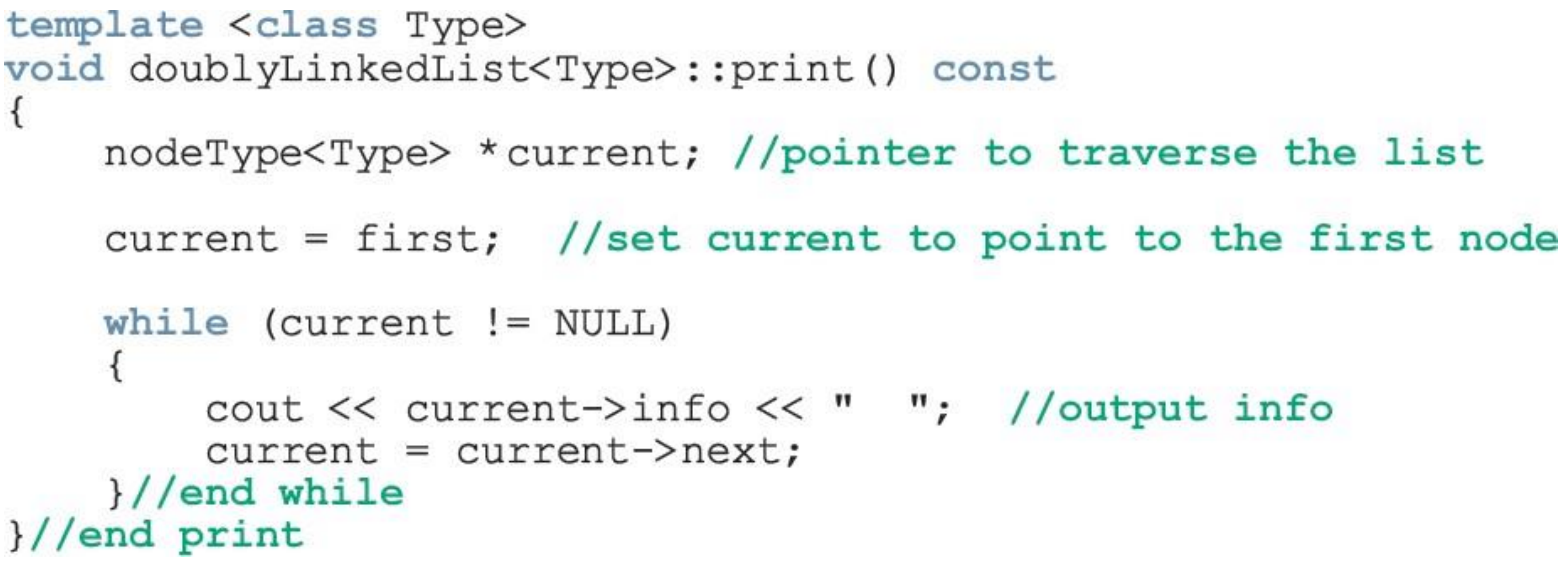
```
template <class Type>
void doublyLinkedList<Type>::initializeList()
{
    destroy();
}
```

Length of the List



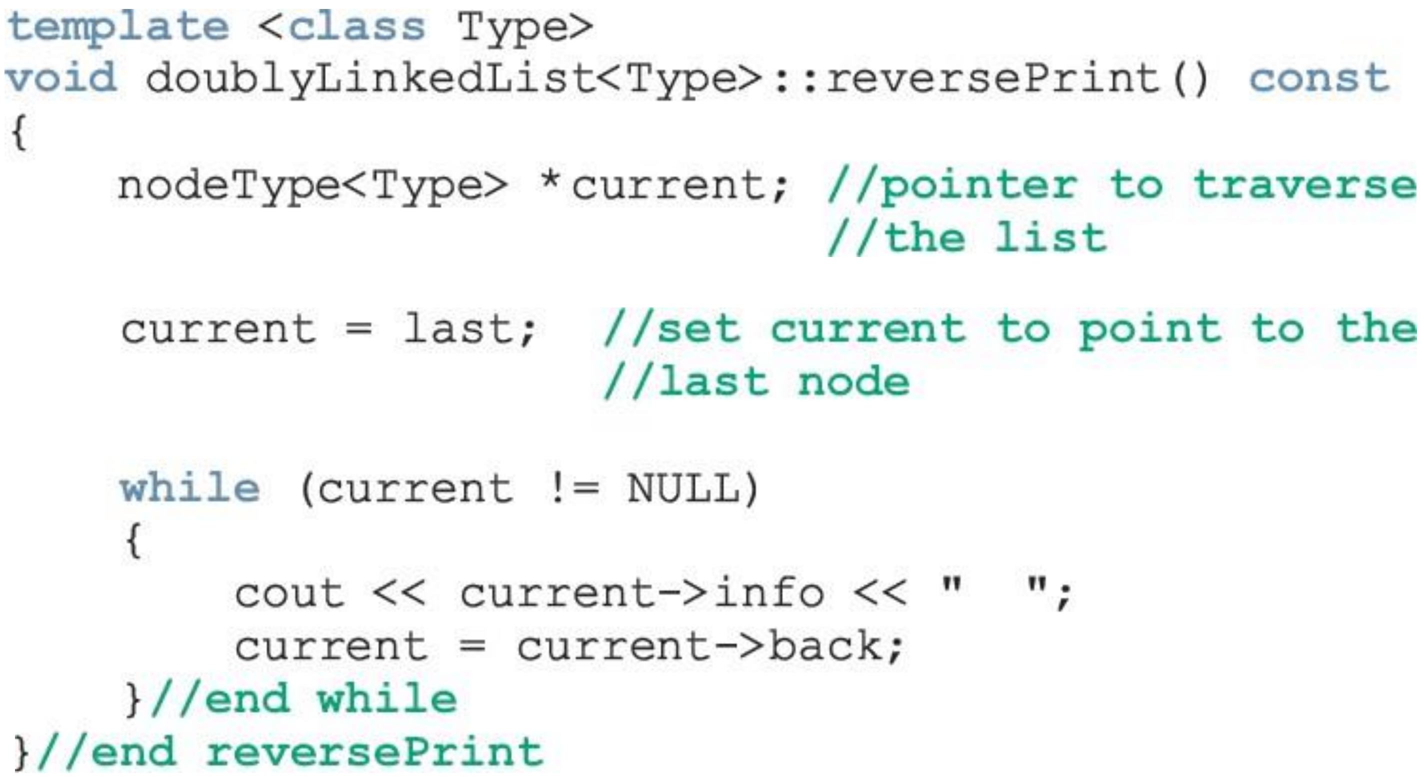
```
template <class Type>
int doublyLinkedList<Type>::length() const
{
    return count;
}
```

Print the List



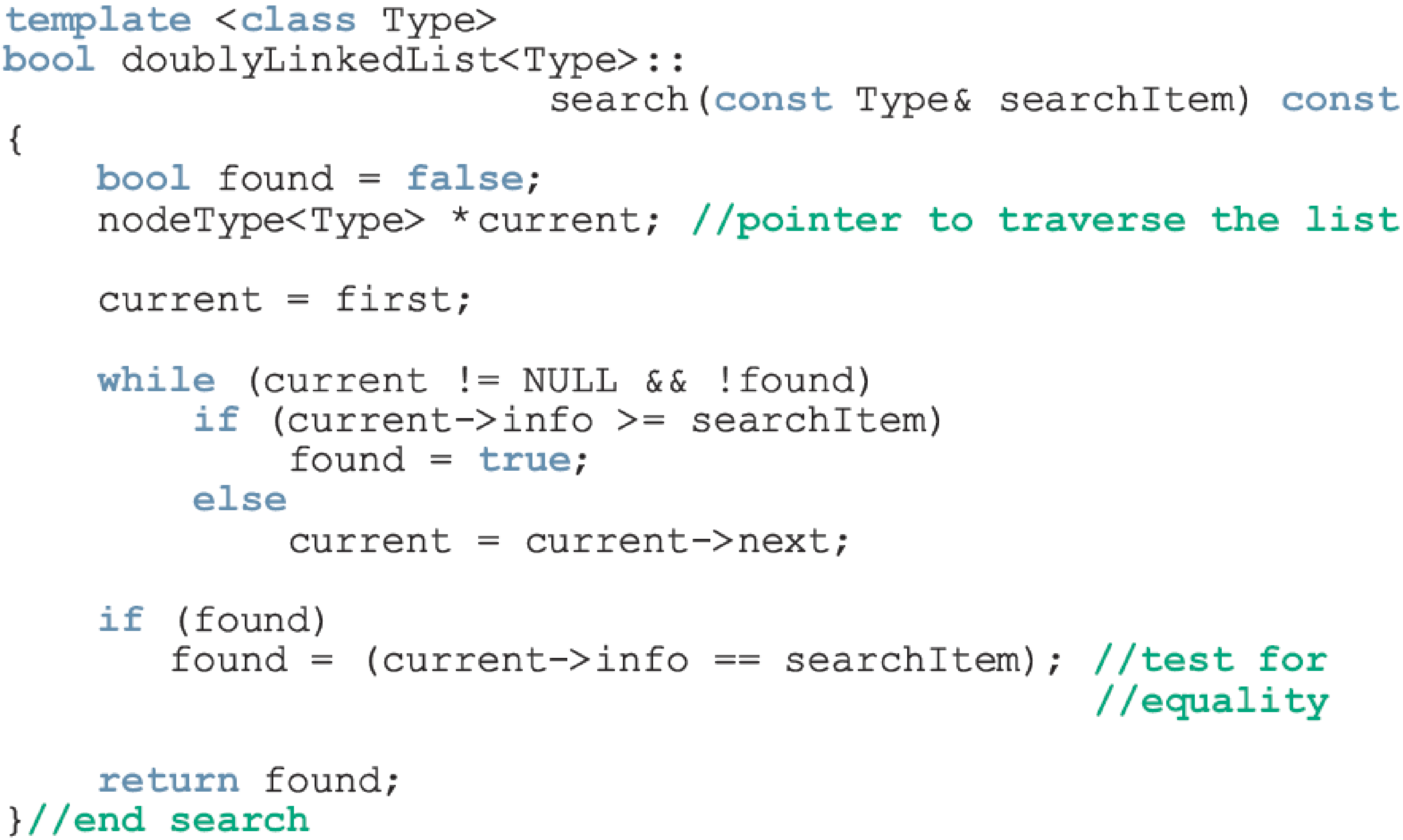
```
template <class Type>
void doublyLinkedList<Type>::print() const
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first;  //set current to point to the first node

    while (current != NULL)
    {
        cout << current->info << "  ";  //output info
        current = current->next;
    }//end while
}//end print
```

Reverse Print the List



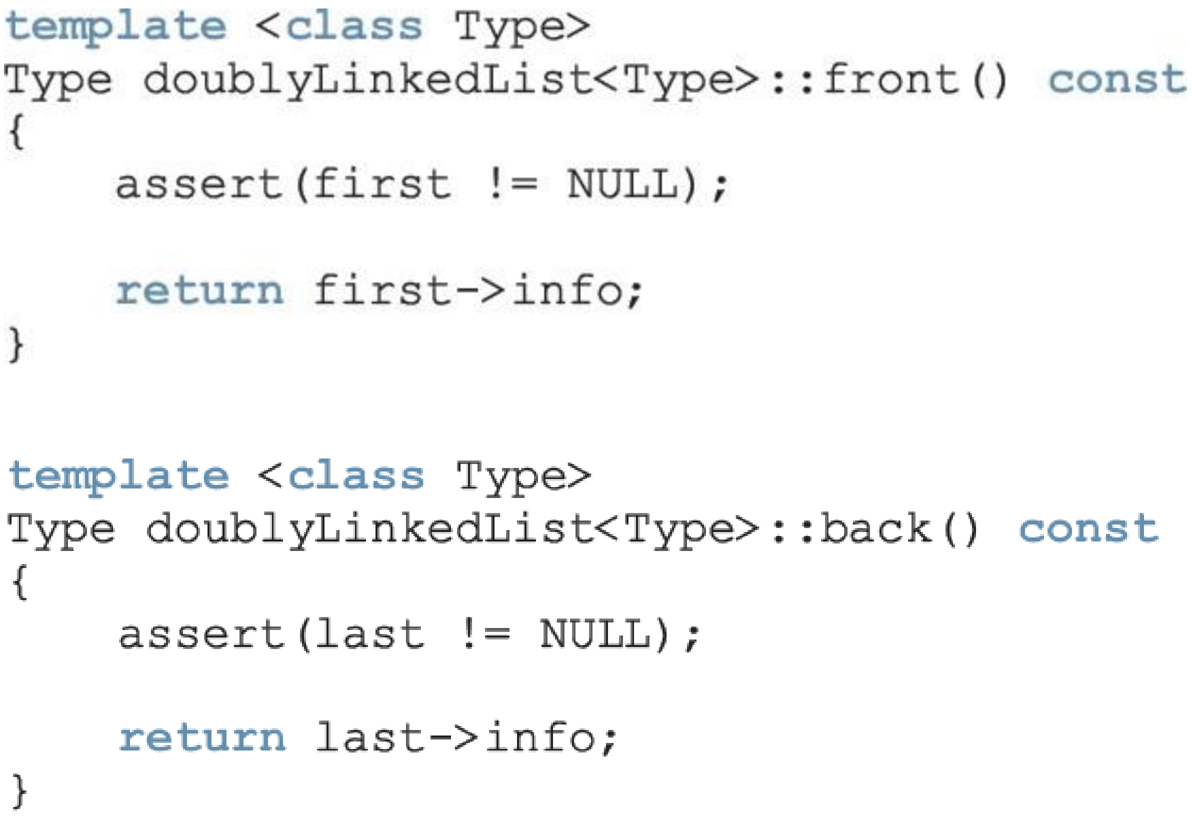
```
template <class Type>
void doublyLinkedList<Type>::reversePrint() const
{
    nodeType<Type> *current; //pointer to traverse
                             //the list

    current = last;  //set current to point to the
                     //last node

    while (current != NULL)
    {
        cout << current->info << "  ";
        current = current->back;
    }//end while
}//end reversePrint
```

Search the List



```
template <class Type>
bool doublyLinkedList<Type>::
                       search(const Type& searchItem) const
{
    bool found = false;
    nodeType<Type> *current; //pointer to traverse the list

    current = first;

    while (current != NULL && !found)
        if (current->info >= searchItem)
            found = true;
        else
            current = current->next;

    if (found)
       found = (current->info == searchItem); //test for
                                              //equality

    return found;
}//end search
```

First and Last Elements



```
template <class Type>
Type doublyLinkedList<Type>::front() const
{
    assert(first != NULL);

    return first->info;
}

template <class Type>
Type doublyLinkedList<Type>::back() const
{
    assert(last != NULL);

    return last->info;
}
```

Insert a Node

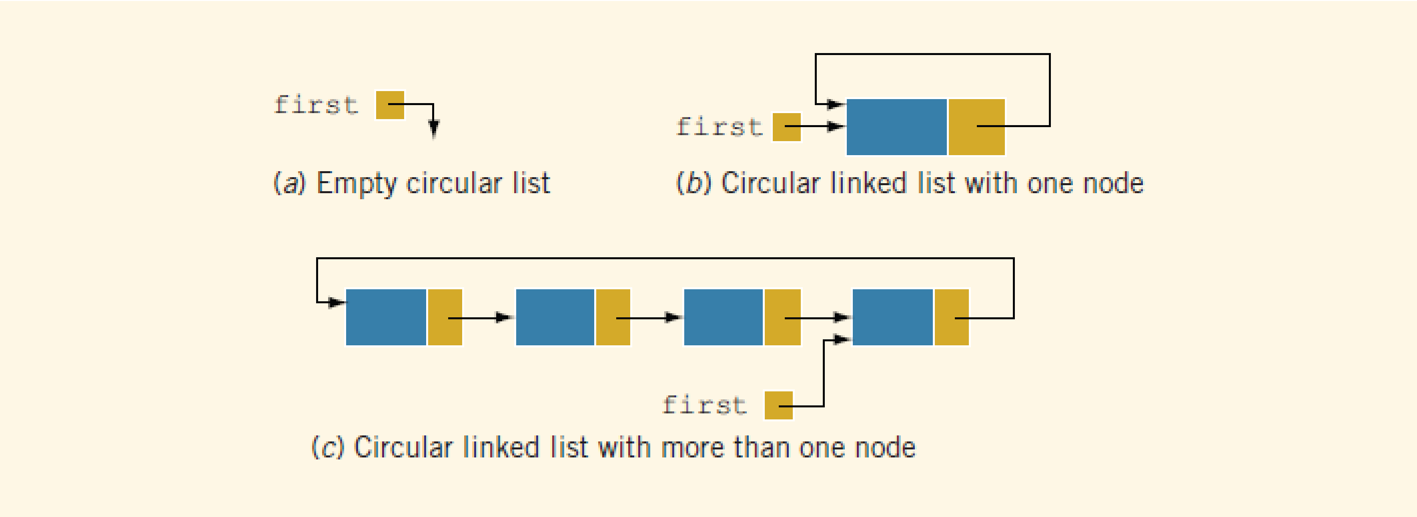
- There are four cases:
  - Case 1: Insertion in an empty list
  - Case 2: Insertion at the beginning of a nonempty list
  - Case 3: Insertion at the end of a nonempty list
  - Case 4: Insertion somewhere in nonempty list
- Cases 1 and 2 require us to change the value of the pointer first
- Cases 3 and 4 are similar (after inserting an item, count is incremented by 1)

Delete a Node

- Case 1: The list is empty
- Case 2: The item to be deleted is in the first node of the list, which would require us to change the value of the pointer first
- Case 3: Item to be deleted is somewhere in the list
- Case 4: Item to be deleted is not in the list
- After deleting a node, count is decremented by 1

Circular Linked Lists

- Circular linked list: a linked list in which the last node points to the first node



(a) Empty circular list

(b) Circular linked list with one node

(c) Circular linked list with more than one node

FIGURE 18-45 Circular linked lists

Circular Linked Lists (cont'd.)

- Operations on a circular list are:
  - Initialize the list (to an empty state)
  - Determine if the list is empty
  - Destroy the list
  - Print the list
  - Find the length of the list
  - Search the list for a given item
  - Insert an item in the list
  - Delete an item from the list
  - Copy the list

Programming Example: Video Store

- A new video store in your neighborhood is about to open
- However, it does not have a program to keep track of its videos and customers
- The store managers want someone to write a program for their system so that the video store can operate

Programming Example: Video Store (cont'd.)

- The program should enable the following:
  - Rent a video; that is, check out a video
  - Return, or check in, a video
  - Create a list of videos owned by the store
  - Show the details of a particular video
  - Print a list of all videos in the store
  - Check whether a particular video is in the store
  - Maintain a customer database
  - Print list of all videos rented by each customer

Programming Example: Video Store (cont'd.)

- Two major components of the video store:
  - Videos
  - Customer
- Maintain the following lists:
  - A list of all videos in the store
  - A list of all the store’s customers
  - Lists of the videos currently rented by the customers

Part 1: Video Component Video Object

- The common things associated with a video are:
  - Name of the movie
  - Names of the stars
  - Name of the producer
  - Name of the director
  - Name of the production company
  - Number of copies in the store

Part 1: Video Component Video List

- This program requires us to:
  - Maintain a list of all videos in the store
  - Be able to add a new video to our list
- Use a linked list to create a list of videos:

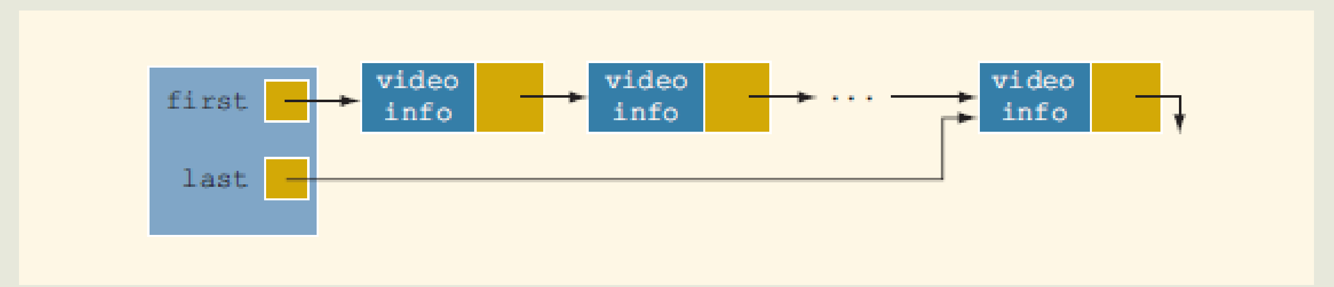


FIGURE 18-46 videoList

Part 1: Video Component Video List (cont'd.)

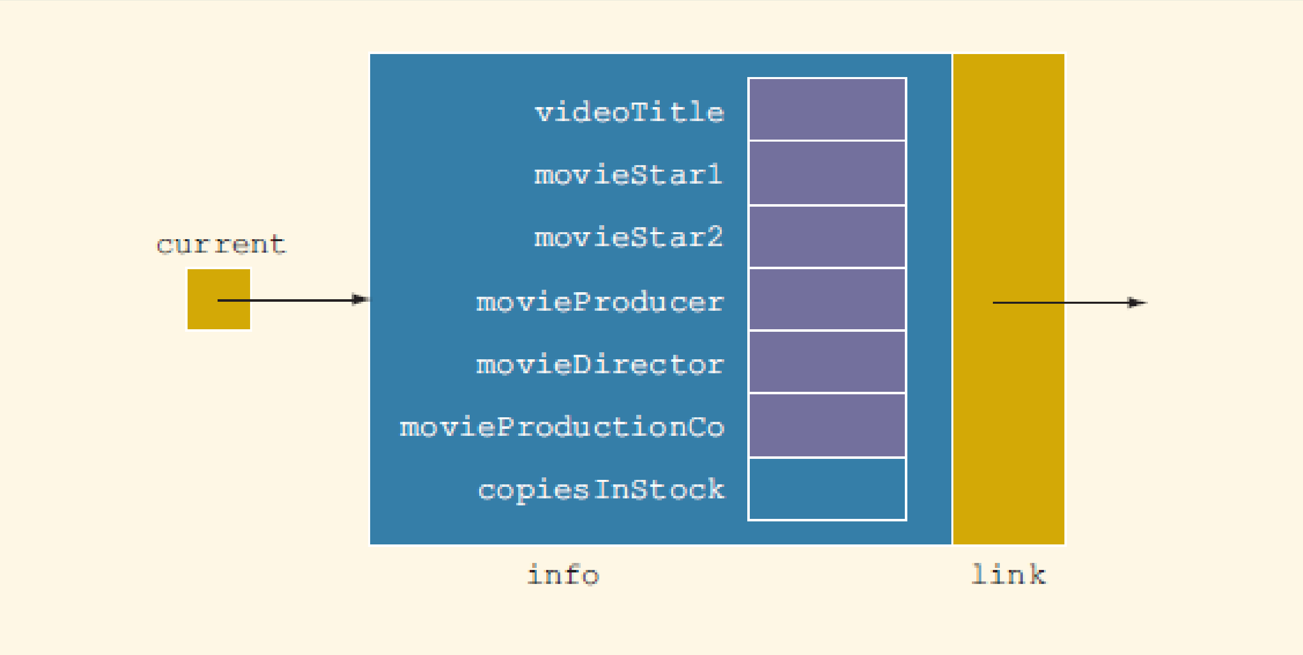
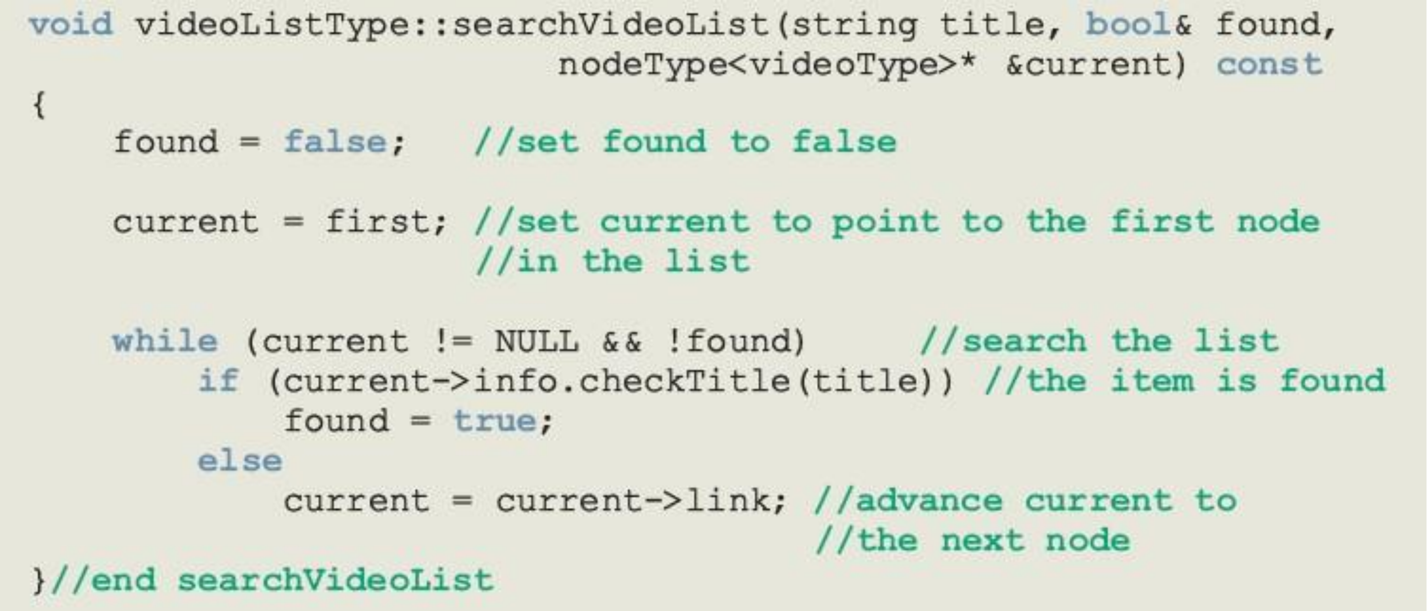


FIGURE 18-49 Pointer `current` and video list node

Part 1: Video Component Video List (cont’d.)



```
void videoListType::searchVideoList(string title, bool& found,
                         nodeType<videoType>* &current) const
{
    found = false;   //set found to false

    current = first; //set current to point to the first node
                     //in the list

    while (current != NULL && !found)     //search the list
        if (current->info.checkTitle(title)) //the item is found
            found = true;
        else
            current = current->link; //advance current to
                                     //the next node
}//end searchVideoList
```

Part 2: Customer Component Customer Object

- Primary characteristics of a customer:
  - First name
  - Last name
  - Account number
  - List of rented videos

Part 2: Customer Component Customer Object (cont'd.)

- Basic operations on personType:
  - Print the name
  - Set the name
  - Show the first name
  - Show the last name

Part 2: Customer Component Customer Object (cont'd.)

- Basic operations on an object of the type customerType are:
  - Print name, account #, and list of rentals
  - Set the name and account number
  - Rent a video (i.e., add video to the list)
  - Return a video (i.e., delete video from the list)
  - Show the account number

Part 2: Customer Component Main Program

- The data in the input file is in the form:

video title (that is, the name of the movie) movie star1
movie star2 movie
producer movie
director
movie production co.
number of copies
.
.
.

Part 2: Customer Component Main Program (cont'd.)

- Open the input file
  - Exit if not found
- createVideoList: create the list of videos
- displayMenu: show the menu
- While not done
  - Perform various operations

Part 2: Customer Component createVideoList

- Read data and store in a video object
- Insert video in list
- Repeat steps 1 and 2 for each video in file

Part 2: Customer Component displayMenu

- Select one of the following:
  - Check if the store carries a particular video
  - Check out a video
  - Check in a video
  - Check whether a particular video is in stock
  - Print the titles of all videos
  - Print a list of all videos
  - Exit

Summary

- A linked list is a list of items (nodes)
  - Order of the nodes is determined by the address, called a link, stored in each node
- The pointer to a linked list is called head or first
- A linked list is a dynamic data structure
- The list length is the number of nodes

Summary (cont'd.)

- Insertion and deletion does not require data movement
  - Only the pointers are adjusted
- A (single) linked list is traversed in only one direction
- Search of a linked list is sequential
- The head pointer is fixed on first node
- Traverse: use a pointer other than head

Summary (cont'd.)

- Doubly linked list
  - Every node has two links: next and previous
  - Can be traversed in either direction
  - Item insertion and deletion require the adjustment of two pointers in a node
- A linked list in which the last node points to the first node is called a circular linked list