COURSE TECHNOLOGY
CENGAGE Learning™

Data Structures Using C++ 2E

Chapter 5
Linked Lists

Objectives

- 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
- Discover how to use the STL container `list`
- Learn about linked lists with header and trailer nodes
- Become aware of circular linked lists

Linked Lists

- Collection of components (nodes)
  - Every node (except last)
    - Contains address of the next node
- Node components
  - Data: stores relevant information
  - Link: stores address (pointer to the next node)

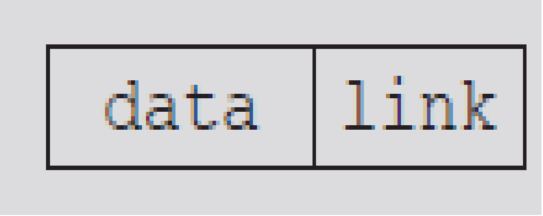


FIGURE 5-1 Structure of a node

Linked Lists (cont'd.)

- Head (first)
  - Address of the first node in the list
- Arrow points to node address
  - Stored in node
- Down arrow in last node indicates `NULL` link field

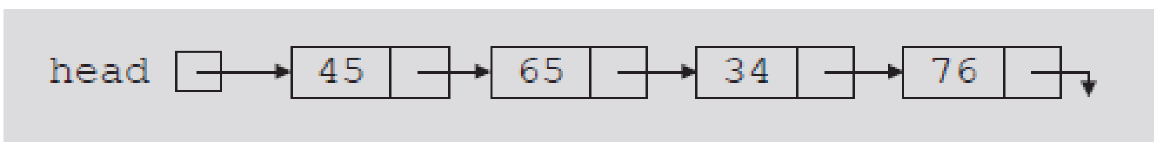


FIGURE 5-2 Linked list

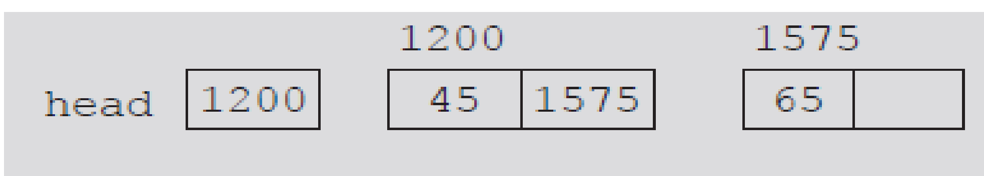


FIGURE 5-3 Linked list and values of the links

Linked Lists (cont'd.)

- Node: Declared as a `class` or `struct`
- Structure of a node: two components
  - info component: information
    - Data type depends on specific application
  - link component: pointer
    - Data type of pointer variable: node type itself

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

The variable declaration is as follows:

```
nodeType *head;
```

Linked Lists: Some Properties

- Pointer `head`: stores address of first node
  - Pointer to the node type
  - `NULL`: empty linked list

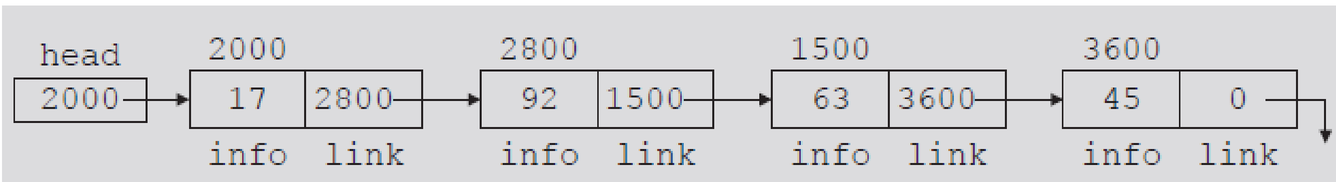


FIGURE 5-4 Linked list with four nodes

TABLE 5-1 Values of head and some of the nodes of the linked list in Figure 5-4

	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.)

- Pointer `current`: same type as pointer `head`
  - `current = head;`
    - Copies value of head into current
  - `current = current->link;`
    - Copies value of `current->link` (2800) into `current`

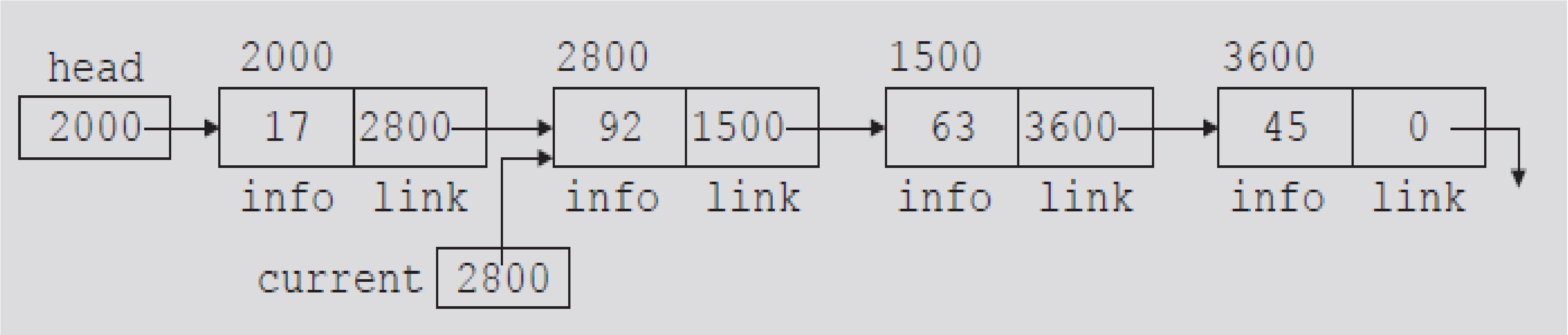


FIGURE 5-5 List after the statement `current = current->link;` executes

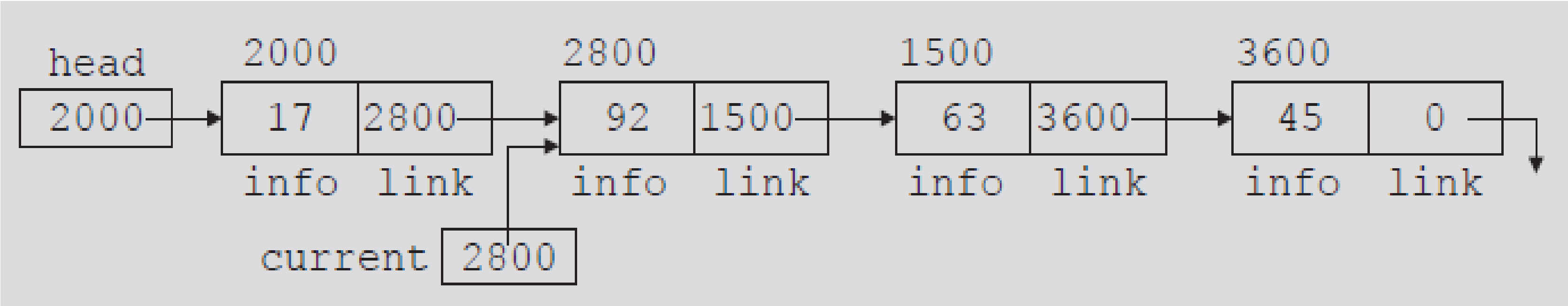


FIGURE 5-5 List after the statement `current = current->link;` executes

TABLE 5-2 Values of `current`, `head`, and some of the nodes of the linked list in Figure 5-5

	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63
`head->link->link`	1500
`head->link->link->info`	63
`head->link->link->link`	3600
`current->link->link->link`	0 (that is, NULL)
`current->link->link->link->info`	Does not exist (run-time error)

Traversing a Linked List

- Basic linked list operations
  - Search list to determine if particular item is in the list
  - Insert item in list
  - Delete item from list
- These operations require list traversal
  - Given pointer to list first node, we must step through list nodes

Traversing a Linked List (cont'd.)

- Suppose `head` points to a linked list of numbers
  - Code outputting data stored in each node

```
current = head;

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

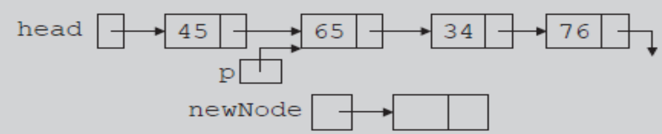
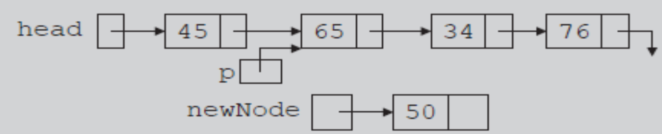
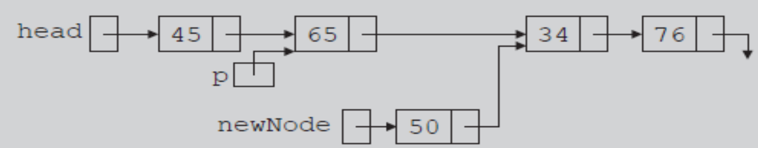
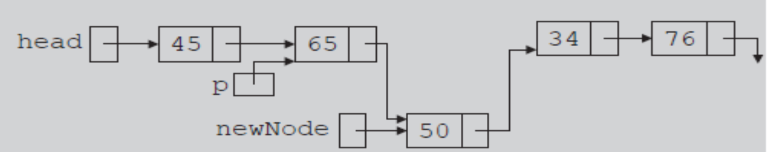
```
current = head;

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

Item Insertion and Deletion

- Insertion: one pointer (page 270)

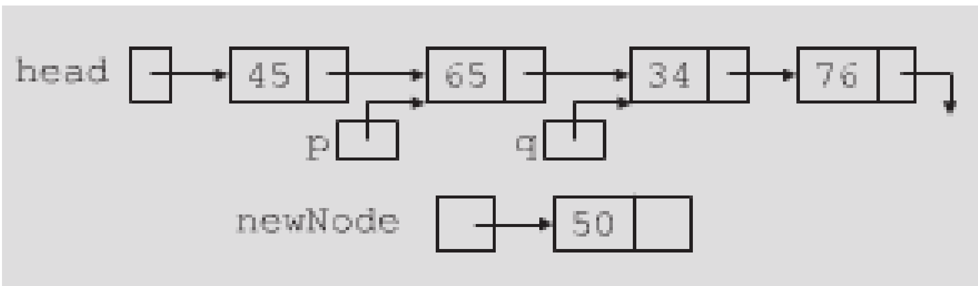
```
newNode = new nodeType;   //create newNode
newNode->info = 50;       //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

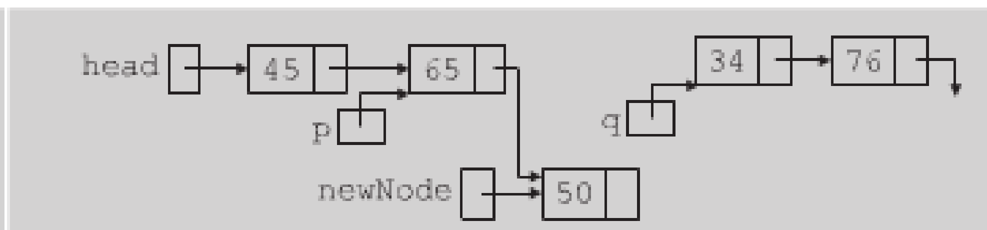
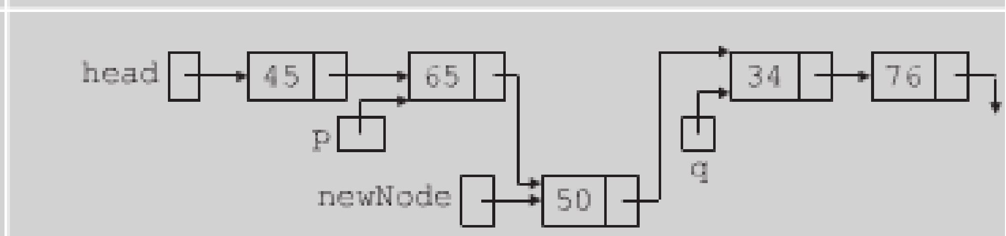
TABLE 5-3 Inserting a node in a linked list

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

Item Insertion and Deletion (cont'd.)

- Insertion: two pointers



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

Item Insertion and Deletion (cont'd.)

- Deletion: Memory still occupied by node after deletion
  - Memory is inaccessible

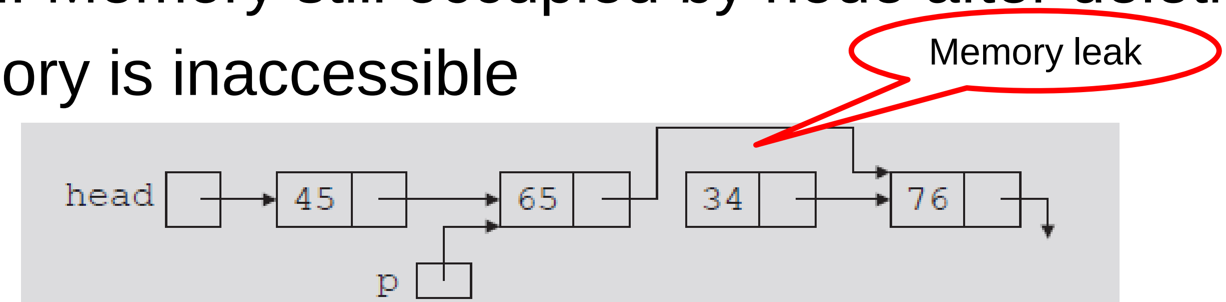
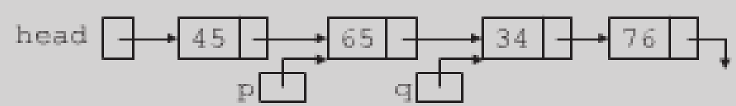
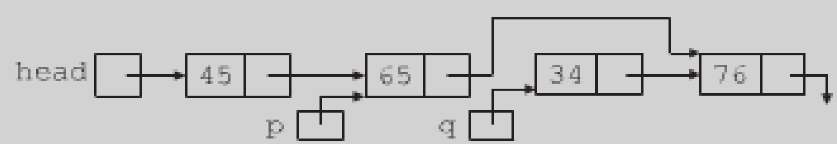
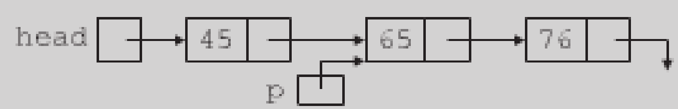


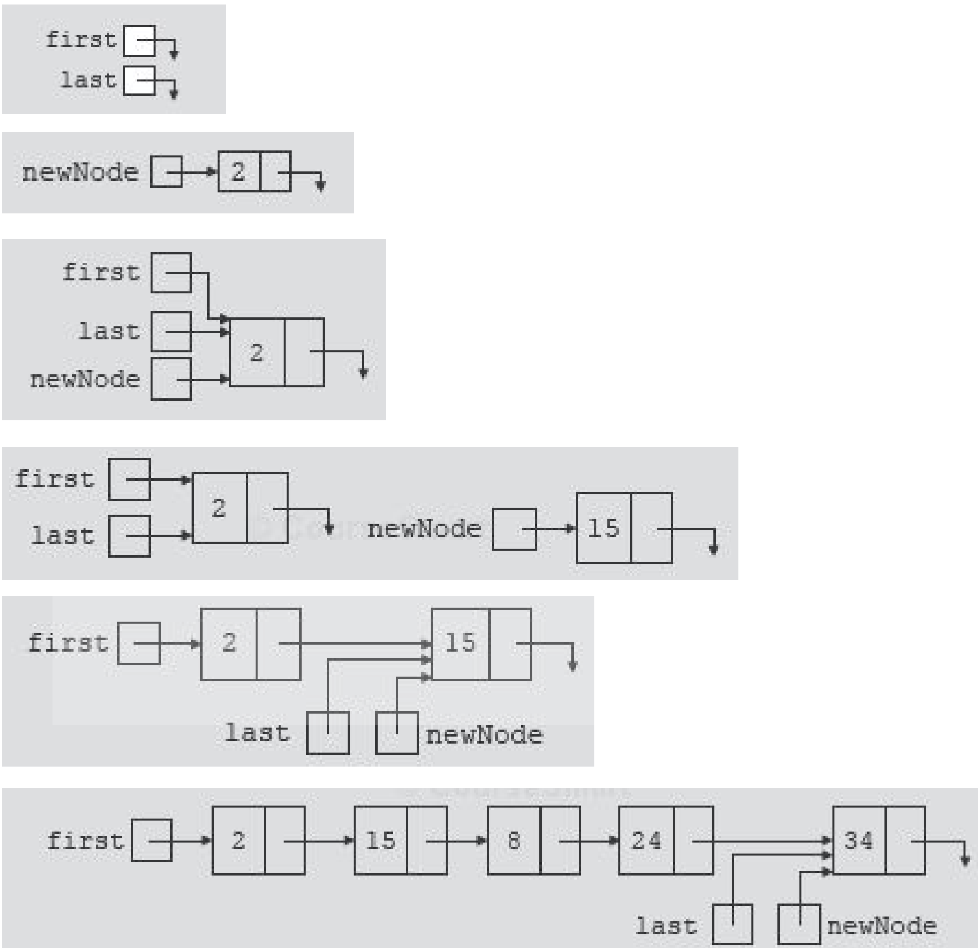
FIGURE 5-10 List after the statement `p->link = p->link->link;` executes

  - Deallocate memory using a pointer to this node

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

Building a Linked List

- Build a linked list with data 2 15 8 24 34



Building a Linked List (cont'd)

- Ways to build linked list
  - Forward
    - New node always inserted at end of the linked list
    - See example on page 274
    - See function `buildListForward` on page 277
  - Backward
    - New node always inserted at the beginning of the list
    - See example on page 277
    - See function `buildListBackward` on page 278

```
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
```

```
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

- Basic operations
  - Initialize the list
  - Is list empty
  - Print the list
  - Find the length
  - Destroy the list
  - Retrieve the info from the 1^{st} node
  - Retrieve the info from the last node
  - Search the list for a given item
  - Insert an item
  - Delete an item
  - Make a copy

Linked List as an ADT (cont'd)

- Two types of linked lists: sorted, unsorted
  - the implement search, insert, and remove are different
- `class linkedListType`
  - Implements basic linked list operations as an ADT
  - Defined as an abstract class
  - Derive two classes using inheritance
    - `unorderedLinkedList` and `orderedLinkedList`
- Unordered linked list functions
  - `buildListForward` and `buildListBackward`
  - Two more functions accommodate both operations
    - `insertFirst` and `insertLast`

Linked List as an ADT (cont'd)

- Definition of the `struct nodeType`

```
//Definition of the node

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

- Member variables of `class linkedListType`
  - Three instance 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
```

Abstract class

```
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
```

Pure virtual function

Linked List Iterators

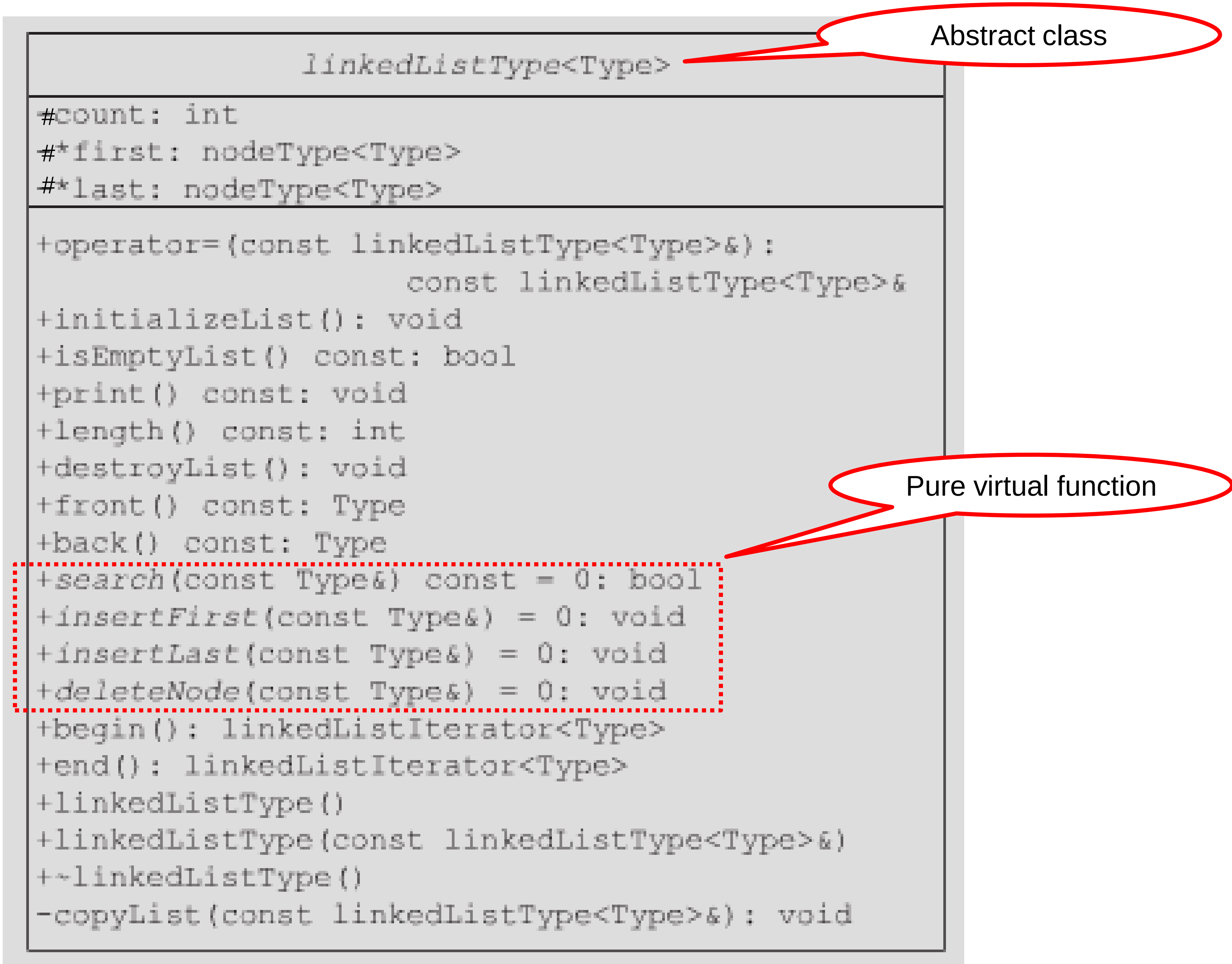
- To process each node
  - Must traverse list starting at first node
- Iterator
  - Object producing each element of a container
  - One element at a time
  - Operations on iterators: `++` and `*`
  - See code on pages 280-281
  - Functions of `class linkedListIterator`

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
```

O(1) for each function

```
template <class Type>
class linkedListIterator
{
public:
    linkedListIterator();
      //Default constructor
      //Postcondition: current = NULL;

    linkedListIterator(nodeType<Type> *ptr);
      //Constructor with a parameter.
      //Postcondition: current = ptr;

    Type operator* ();
      //Function to overload the dereferencing operator *.
      //Postcondition: Returns the info contained in the node.

    linkedListIterator<Type> operator++();
      //Overload the preincrement operator.
      //Postcondition: The iterator is advanced to the next node.
    bool operator==(const linkedListIterator<Type>& right) const;
      //Overload the equality operator.
      //Postcondition: Returns true if this iterator is equal to
      //    the iterator specified by right, otherwise it returns
      //    false.

    bool operator!=(const linkedListIterator<Type>& right) const;
      //Overload the not equal to operator.
      //Postcondition: Returns true if this iterator is not equal to
      //    the iterator specified by right, otherwise it returns
      //    false.

private:
    nodeType<Type> *current; //pointer to point to the current
                             //node in the linked list
};
```


linkedListType<Type>
Abstract class
#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
Pure virtual function

Linked List as an ADT

- Abstract `class linkedListType`
  - Defines basic properties of a linked list as an ADT
  - See code on page 282
  - Empty list: `first` is `NULL`
    - Definition of function `isEmptyList`

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

Linked List as an ADT (cont'd.)

- Default constructor
  - Initializes list to an empty state
- Destroy the list
  - Deallocates memory occupied by each node
- Initialize the list
  - Reinitializes list to an empty state
    - Must delete the nodes (if any) from the list
  - Default constructor, copy constructor
    - Initialized list when list object declared
- Print the list
  - Must traverse the list starting at first node

Linked List as an ADT (cont'd.)

- Length of a list
  - Number of nodes stored in the variable `count`
  - Function `length`
    - Returns value of variable `count`
- Retrieve the data of the first node
  - Function `front`
    - Returns the `info` contained in the first node
    - If list is empty, `assert` statement terminates program
- Retrieve the data of the last node
  - Function `back`
    - Returns `info` contained in the last node
    - If list is empty, `assert` statement terminates program

Linked List as an ADT (cont'd.)

- Begin and end
  - Function `begin` returns an *iterator* to the first node in the linked list
  - Function `end` returns an *iterator* to <u>one element past the last node</u> in the linked list
- Copy the list
  - Makes an identical copy of a linked list
    - Create node called `newNode`
    - Copy node info from original list into `newNode`
    - Insert `newNode` at the end of list being created
  - See function `copyList` on page 289

```
template <class Type>
void linkedListType<Type>::copyList
                    (const linkedListType<Type>& otherList)
{
    nodeType<Type> *newNode; //pointer to create a node
    nodeType<Type> *current; //pointer to traverse the list

    if (first != NULL) //if the list is nonempty, make it empty
       destroyList();

    if (otherList.first == NULL) //otherList is empty
    {
        first = NULL;
        last = NULL;
        count = 0;
    }
    else
    {
        current = otherList.first; //current points to the
                                   //list to be copied
        count = otherList.count;

            //copy the first node
        first = new nodeType<Type>;  //create the node
        first->info = current->info; //copy the info
        first->link = NULL; //set the link field of the node to NULL
        last = first;    //make last point to the first node
        current = current->link; //make current point to the next
                                 // node

           //copy the remaining list
        while (current != NULL)
        {
            newNode = new nodeType<Type>;  //create a node
            newNode->info = current->info; //copy the info
            newNode->link = NULL; //set the link of newNode to NULL

            last->link = newNode; //attach newNode after last
            last = newNode; //make last point to the actual last
                            //node
            current = current->link; //make current point to the
                                     //next node
        }//end while
    }//end else
}//end copyList
```

Linked List as an ADT (cont'd.)

- Destructor
  - When class object goes out of scope
    - Deallocates memory occupied by list nodes
  - Memory allocated dynamically
    - Resetting pointers first and last
      - Does not deallocate memory
    - Must traverse list starting at first node
      - Delete each node in the list
  - Calling `destroyList` destroys list
  - Time complexity?

Linked List as an ADT (cont'd.)

- Copy constructor
  - Makes identical copy of the linked list
  - Function `copyListc` checks whether original list empty
    - Checks value of `first`
  - Must initialize `first` to `NULL`
    - Before calling the function `copyList`

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

Linked List as an ADT (cont'd.)

- Overloading the assignment operator
  - Similar to copy constructor definition

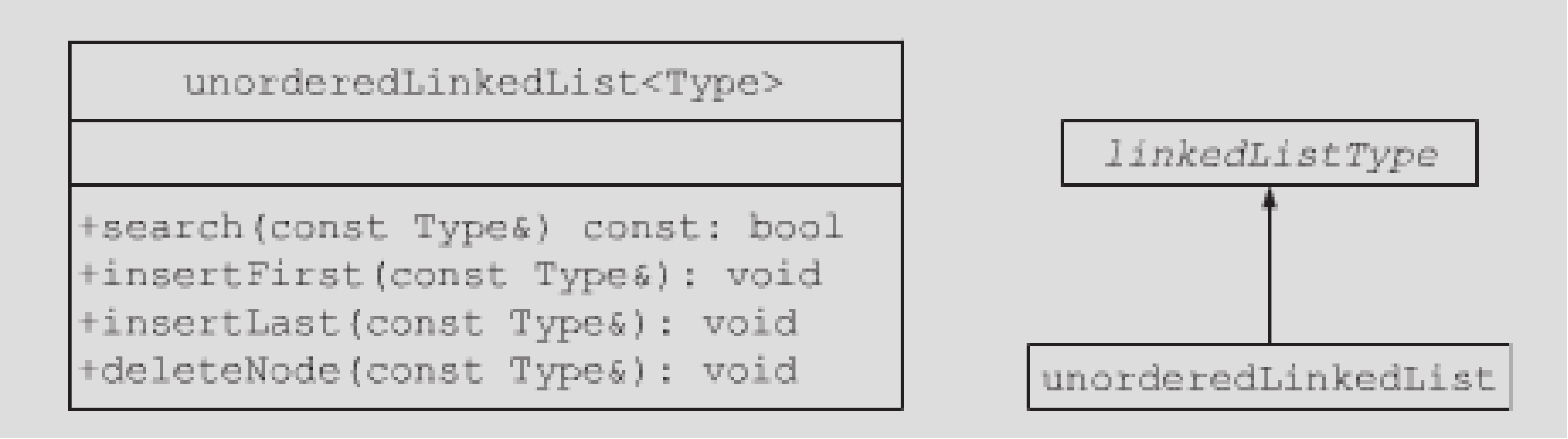
```
    //overload the assignment operator
template <class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                      (const linkedListType<Type>& otherList)
{
    if (this != &otherList) //avoid self-copy
    {
        copyList(otherList);
    }//end else

    return *this;
}
```

TABLE 5-6 Time-complexity of the operations of the `class linkedListType`

Function	Time-complexity
`isEmptyList`	$O(1)$
default constructor	$O(1)$
`destroyList`	$O(n)$
`front`	$O(1)$
`end`	$O(1)$
`initializeList`	$O(n)$
`print`	$O(n)$
`length`	$O(1)$
`front`	$O(1)$
`back`	$O(1)$
`copyList`	$O(n)$
destructor	$O(n)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

Unordered Linked Lists

- Derive `class unorderedLinkedList` from the abstract `class linkedListType`
  - Implement the operations `search`, `insertFirst`, `insertLast`, `deleteNode`



- See code on page 292
  - Defines an unordered linked list as an ADT

Unordered Linked Lists (cont'd.)

- Search the list
  - Steps
    - Step one: 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
    - Step two: Repeat Step one until either the item is found or no more data is left in the list to compare with the search item
  - See function `search` on page 293

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

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

    while (current != NULL && !found)     //search the list
        if (current->info == searchItem) //searchItem is found
            found = true;

        else
            current = current->link; //make current point to
                                     //the next node
    return found;
}//end search
```

Unordered Linked Lists (cont'd.)

- Insert the first node
  - Steps
    - Create a new node
    - If unable to create the node, terminate the program
    - Store the new item in the new node
    - Insert the node before first
    - Increment count by one
  - See function `insertFirst` on page 294

1. Create a new node.
2. If unable to create the node, terminate the program.
3. Store the new item in the new node.
4. Insert the node before **first**.
5. Increment **count** by **1**.

```
template <class Type>
void unorderedLinkedList<Type>::insertFirst(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 = first;     //insert newNode before first
    first = newNode;   //make first point to the actual first node
    count++;   //increment count

    if (last == NULL)  //if the list was empty, newNode is also
                       //the last node in the list
        last = newNode;
}//end insertFirst
```

Unordered Linked Lists (cont'd.)

- Insert the last node
  - Similar to definition of member function `insertFirst`
  - Insert new node after last
  - See function `insertLast` on page 294

```
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
```

Unordered Linked Lists (cont'd.)

- Delete a node
  - Consider the following cases:
    - The list is empty
    - The node is nonempty and the node to be deleted is the first node
    - The node is nonempty and the node to be deleted is not the first node, it is somewhere in the list
    - The node to be deleted is not in the list
  - See pseudocode on page 295
  - See definition of function `deleteNode` on page 297

Pseudo Code

```
if list is empty
   Output(cannot delete from an empty list);
else
{
   if the first node is the node with the given info
       adjust the head pointer, that is, first, and deallocate
       the memory;
   else
   {
       search the list for the node with the given info
       if such a node is found, delete it and adjust the
       values of last (if necessary) and count.
   }
}
```

Case 1: The list is empty. If the list is empty, output an error message as shown in the pseudocode.

Case 2: The list is not empty and the node to be deleted is the first node. This case has two scenarios: `list` has only one node, and `list` has more than one node. If list has only one node, then after deletion, the list becomes empty. Therefore, after deletion, both `first` and `last` are set to `NULL` and `count` is set to `0`.

Case 3: The node to be deleted is not the first node, but is somewhere in the list.

This case has two subcases: (a) the node to be deleted is not the last node, and (b) the node to be deleted is the last node. Let us illustrate the first cases.

Case 3a: The node to be deleted is not the last node.

Case 3b: The node to be deleted is the last node. In this case, after deleting the node, the value of the pointer `last` changes. It contains the address of the node just before the node to be deleted. For example, consider the list given in Figure 5-21 and the node to be deleted is `54`. After deleting `54`, `last` contains the address of the node with `info 24`. Also, `count` is decremented by `1`.

Case 4: The node to be deleted is not in the list. In this case, the list requires no adjustment. We simply output an error message, indicating that the item to be deleted is not in the list.

Exercise

- Please implement

```
Template <class Type>
unorderedLinkedList<Type>::deleteNode(const Type &deleteItem)
```

```
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
```

Unordered Linked Lists (cont'd.)

TABLE 5-7 Time-complexity of the operations of the `class unorderedLinkedList`

Function	Time-complexity
`search`	$O(n)$
`insertFirst`	$O(1)$
`insertLast`	$O(1)$
`deleteNode`	$O(n)$

Header File of the Unordered Linked List

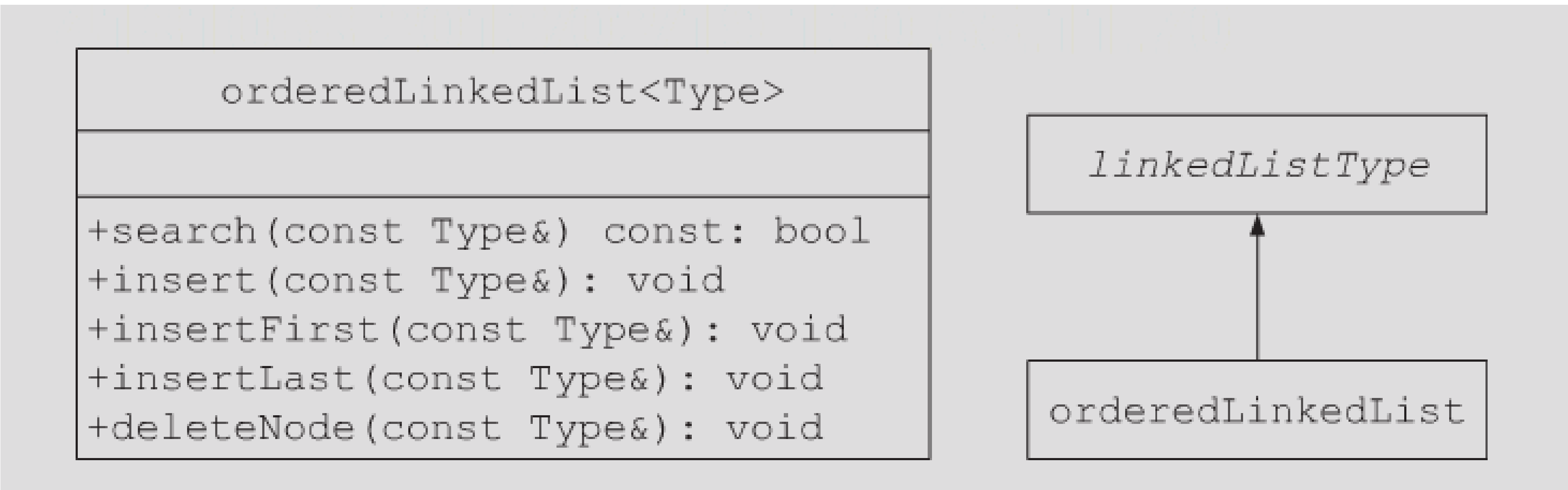
- Create header file defining `class unorderedListType`
  - See `class unorderedListType` code on page 299
    - Specifies members to implement basic properties of an unordered linked list
    - Derived from `class linkedListType`

```
#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, 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 end of the list, last points to the
      //    last node, 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, insertNode,
//insertFirst, insertLast, and deleteNode here.
.
.
```

Ordered Linked Lists

- Derive `class orderedLinkedList` from `class linkedListType`
  - Provide definitions of the abstract functions:
    - `insertFirst`, `insertLast`, `search`, `deleteNode`
  - Ordered linked list elements are arranged using some ordering criteria
    - Assume elements of an ordered linked list arranged in ascending order
- See `class orderedLinkedList` on pages 300-301

Ordered Linked Lists (cont'd)



```
#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.

    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.
      //

    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 of
      //    the new list, and count is decremented by 1. If
      //    deleteItem is not in the list, an appropriate message
      //    is printed.
};
```

Ordered Linked Lists (cont'd.)

- Search the list
  - Steps describing algorithm
    - Step one: Compare the search item with the current node in the list
      - If the info of the current node is greater than or equal to the search item, stop the search
      - otherwise, make the next node the current node
    - Step two: Repeat Step one until either an item in the list that is greater than or equal to the search item is found, or no more data is left in the list to compare with the search item
      - at the end of while loop, check if the found item is equal to the search item

Ordered Linked Lists (cont'd.)

- Insert a node
  - Find place where new item goes
  - Consider the following cases (code on p.304)

Case 1: The list is initially empty. The node containing the new item is the only node and, thus, the first node in the list.

Case 2: The new item is smaller than the smallest item in the list. The new item goes at the beginning of the list. In this case, we need to adjust the list's head pointer—that is, `first`. Also, `count` is incremented by `1`.

Case 3: The item is to be inserted somewhere in the list.

Case 3a: The new item is larger than all the items in the list. In this case, the new item is inserted at the end of the list. Thus, the value of `current` is `NULL` and the new item is inserted after `trailCurrent`. Also, `count` is incremented by `1`.

Case 3b: The new item is to be inserted somewhere in the middle of the list. In this case, the new item is inserted between `trailCurrent` and `current`. Also, `count` is incremented by `1`.

Ordered Linked Lists (cont'd.)

- Insert first and insert last
  - Function insertFirst
    - Must be inserted at the proper place
  - Function insertLast
    - Inserts new item at the proper place

Ordered Linked Lists (cont'd.)

- Delete a node
  - Several cases to consider
  - See function `deleteNode` code on page 306

Case 1: The list is initially empty. We have an error. We cannot delete from an empty list.

Case 2: The item to be deleted is contained in the first node of the list. We must adjust the head pointer of the list—that is, `first`.

Case 3: The item to be deleted is somewhere in the list. In this case, `current` points to the node containing the item to be deleted, and `trailCurrent` points to the node just before the node pointed to by `current`.

Case 4: The list is not empty, but the item to be deleted is not in the list.

```
template <class Type>
void orderedLinkedList<Type>::deleteNode(const Type& deleteItem)
{
    nodeType<Type> *current; //pointer to traverse the list
    nodeType<Type> *trailCurrent; //pointer just before current
    bool found;

    if (first == NULL) //Case 1
        cout << "Cannot delete from an empty list." << endl;
    else
    {
        current = first;
        found = false;

        while (current != NULL && !found)  //search the list
            if (current->info >= deleteItem)
                found = true;
            else
            {
                trailCurrent = current;
                current = current->link;
            }

        if (current == NULL)    //Case 4
            cout << "The item to be deleted is not
                 << endl;
        else
            if (current->info == deleteItem) //the
                                             //deleted is in
```

```
{
    if (first == current)       //Case 2
    {
        first = first->link;

        if (first == NULL)
            last = NULL;

        delete current;
    }
    else                        //Case 3
    {
        trailCurrent->link = current->link;

        if (current == last)
            last = trailCurrent;

        delete current;
    }
    count--;
}
else                            //Case 4
    cout << "The item to be deleted is not in the "
         << "list." << endl;
    }
}//end deleteNode
```

Ordered Linked Lists (cont'd.)

TABLE 5-8 Time-complexity of the operations of the `class orderedLinkedList`

Function	Time-complexity
`search`	$O(n)$
`insert`	$O(n)$
`insertFirst`	$O(n)$
`insertLast`	$O(n)$
`deleteNode`	$O(n)$

Header File of the Ordered Linked List

- See code on page 308
  - Specifies members to implement the basic properties of an ordered linked list
  - Derived from `class linkedListType`
- See test program on page 309
  - Tests various operations on an ordered linked list

Doubly Linked Lists

- Traversed in either direction
- Typical operations
  - Initialize the list
  - Destroy the list
  - Determine if list empty
  - Search list for a given item
  - Insert an item
  - Delete an item, and so on

```
  //Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;
    nodeType<Type> *back;
};
```

- See code on page 311
  - Class specifying members to implement properties of an ordered doubly linked list

Doubly Linked Lists (cont'd.)

- Linked list in which every node has a `next` pointer and a `back` pointer
  - Every node contains address of next node
    - Except last node
  - Every node contains address of previous node
    - Except the first node

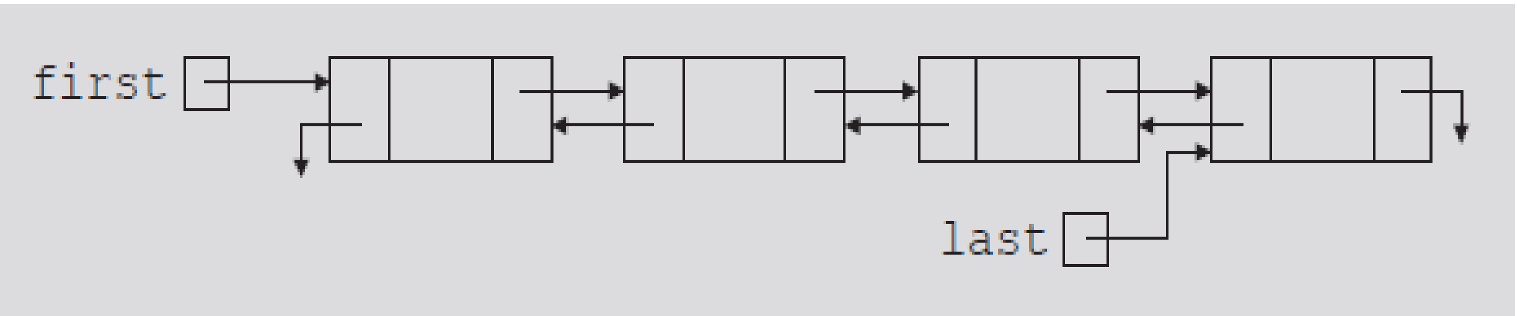


FIGURE 5-27 Doubly linked list

```cpp
template <class Type>
class doublyLinkedList
{
public:
    const doublyLinkedList<Type>& operator=
                        (const doublyLinkedList<Type> &);

    void initializeList();
    bool isEmptyList() const;
    void destroy();
    void print() const;
    void reversePrint() const;
    int length() const;
    Type back() const;
    bool search(const Type& searchItem) const;
    void insert(const Type& insertItem);
    void deleteNode(const Type& deleteItem);
    doublyLinkedList();
    doublyLinkedList(const doublyLinkedList<Type>& otherList);
    ~doublyLinkedList();

protected:
    int count;
    nodeType<Type> *first; //pointer to the first node
    nodeType<Type> *last;  //pointer to the last node

private:
    void copyList(const doublyLinkedList<Type>& otherList);

};
```

Doubly Linked Lists (cont'd.)

- Default constructor
  - Initializes the doubly linked list to an empty state
- `isEmptyList`
  - Returns `true` if the list empty
    - Otherwise returns `false`
  - List empty if pointer `first` is `NULL`
- Destroy the list
  - Deletes all nodes in the list
    - Leaves list in an empty state
    - Traverse list starting at the first node; delete each node
    - `count` set to zero

Doubly Linked Lists (cont'd.)

- Initialize the list
  - Reinitializes doubly linked list to an empty state
    - Can be done using the operation `destroy`
- Length of the list
  - Length of a linked list stored in variable `count`
- Print the list
  - Outputs info contained in each node
    - Traverse list starting from the first node
- Reverse print the list
  - Outputs info contained in each node in reverse order
    - Traverse list starting from the last node

Doubly Linked Lists (cont'd.)

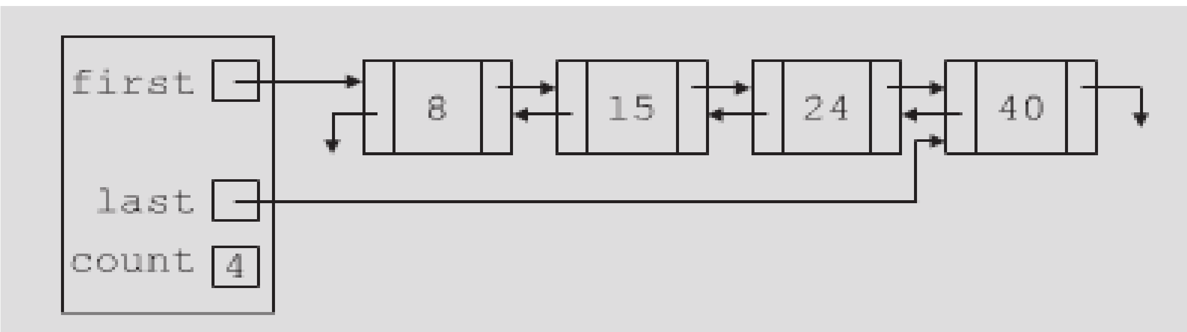
- Search the list
  - Function `search` returns true if `searchItem` found
    - Otherwise, it returns false
  - Same as ordered linked list search algorithm
- First and last elements
  - Function `front` returns first list element
  - Function `back` returns last list element
  - If list empty, terminate the program

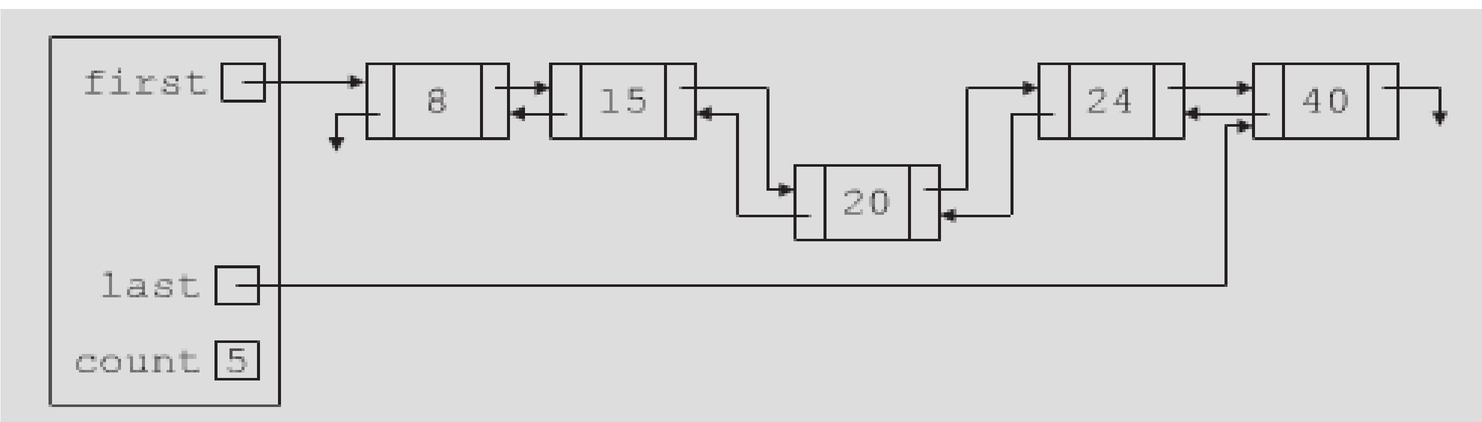
Doubly Linked Lists (cont'd.)

- Insert a node
  - 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 a nonempty list
    - Cases 1 and 2 requirement: Change value of the pointer `first`
    - Cases 1 and 3 requirement: Change value of the pointer `last`

Doubly Linked Lists (cont'd.)

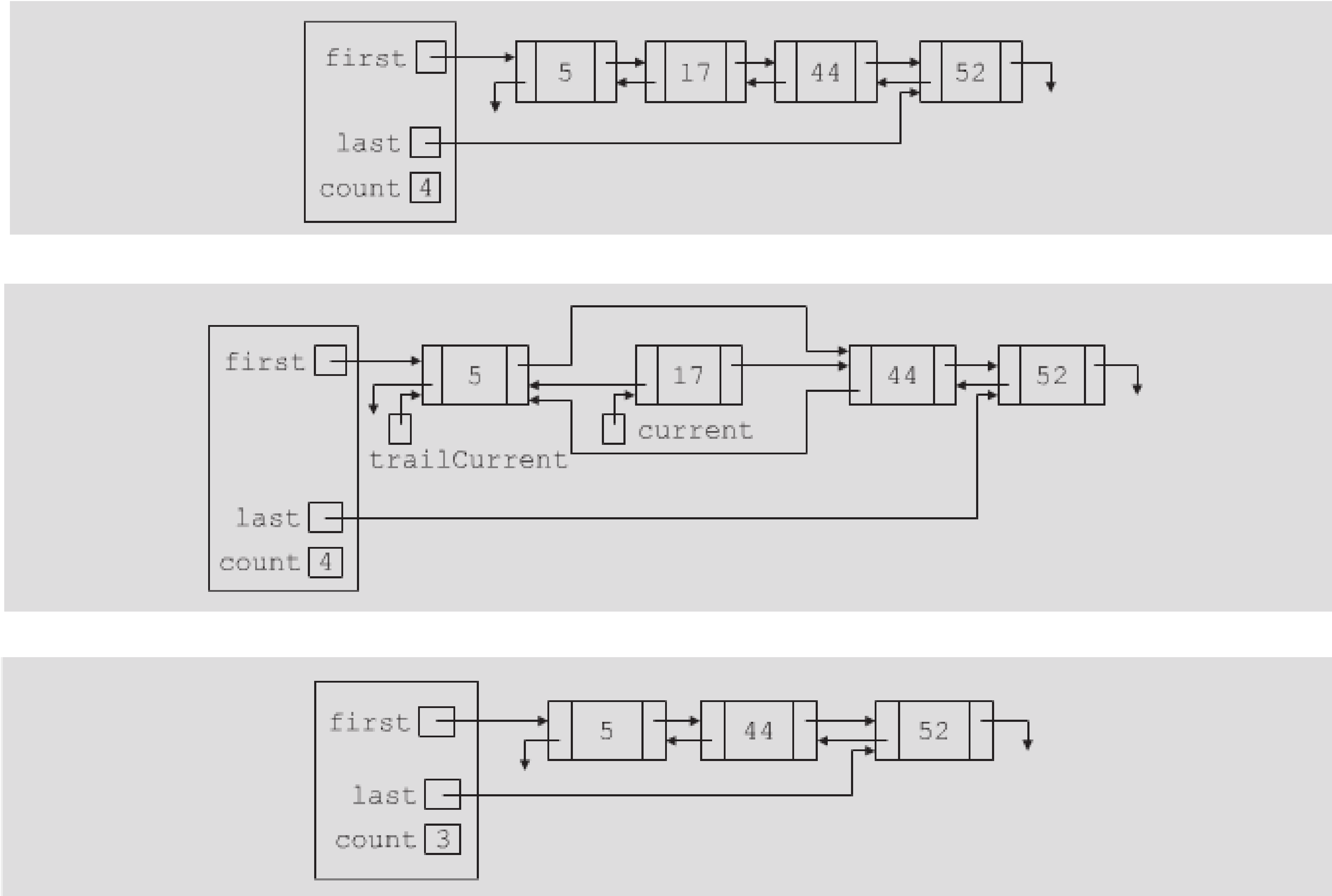
- Insert a node (cont'd.)
  – Figures 5-28 and 5-29 illustrate case 4
  – See code on page 317, definition of function `insert`





Doubly Linked Lists (cont'd.)

- Delete a node
  - Four cases
    - 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: The item to be deleted is somewhere in the list
    - Case 4: The item to be deleted is not in the list
  - See code on page 319, definition of function `deleteNode`


first
last
count 4
5
17
44
52
first
last
count 4
5
17
44
52
trailCurrent
current
first
last
count 3
5
44
52

STL Sequence Container: `list`

- `class list` is a Doubly linked list

`#include <list>`

TABLE 5-9 Various ways to declare a list object

Statement	Description
`list<elemType> listCont;`	Creates the empty `list` container `listCont`. (The default constructor is invoked.)
`list<elemType> listCont(otherList);`	Creates the `list` container `listCont` and initializes it to the elements of `otherList`. `listCont` and `otherList` are of the same type.
`list<elemType> listCont(size);`	Creates the `list` container `listCont` of size `size`. `listCont` is initialized using the default constructor.
`list<elemType> listCont(n, elem);`	Creates the `list` container `listCont` of size `n`. `listCont` is initialized using `n` copies of the element `elem`.
`list<elemType> listCont(beg, end);`	Creates the `list` container `listCont`. `listCont` is initialized to the elements in the range `[beg, end)`, that is, all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.

TABLE 5-10 Operations specific to a `list` container

Expression	Description
`listCont.assign(n, elem)`	Assigns `n` copies of `elem`.
`listCont.assign(beg, end)`	Assigns all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.
`listCont.push_front(elem)`	Inserts `elem` at the beginning of `listCont`.
`listCont.pop_front()`	Removes the first element from `listCont`.
`listCont.front()`	Returns the first element. (Does not check whether the container is empty.)
`listCont.back()`	Returns the last element. (Does not check whether the container is empty.)
`listCont.remove(elem)`	Removes all the elements that are equal to `elem`.
`listCont.remove_if(oper)`	Removes all the elements for which `oper` is `true`.
`listCont.unique()`	If the consecutive elements in `listCont` have the same value, removes the duplicates.
`listCont.unique(oper)`	If the consecutive elements in `listCont` have the same value, removes the duplicates, for which `oper` is `true`.
`listCont1.splice(pos, listCont2)`	All the elements of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`. After this operation, `listCont2` is empty.

TABLE 5-10 Operations specific to a `list` container (cont'd.)

Expression	Description
`listCont1.splice(pos, listCont2, pos2)`	All the elements starting at `pos2` of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`.
`listCont1.splice(pos, listCont2, beg, end)`	All the elements in the range `beg...end-1` of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`. Both `beg` and `end` are iterators.
`listCont.sort()`	The elements of `listCont` are sorted. The sort criterion is <.
`listCont.sort(oper)`	The elements of `listCont` are sorted. The sort criterion is specified by `oper`.
`listCont1.merge(listCont2)`	Suppose that the elements of `listCont1` and `listCont2` are sorted. This operation moves all the elements of `listCont2` into `listCont1`. After this operation, the elements in `listCont1` are sorted. Moreover, after this operation, `listCont2` is empty.
`listCont1.merge(listCont2, oper)`	Suppose that the elements of `listCont1` and `listCont2` are sorted according to the sort criteria `oper`. This operation moves all the elements of `listCont2` into `listCont1`. After this operation, the elements in `listCont1` are sorted according to the sort criteria `oper`.
`listCont.reverse()`	The elements of `listCont` are reversed.

```
int main()                                              //Line 6
{                                                       //Line 7
    list<int> intList1, intList2;                       //Line 8

    ostream_iterator<int> screen(cout, " ");            //Line 9

    intList1.push_back(23);                             //Line 10
    intList1.push_back(58);                             //Line 11
    intList1.push_back(58);                             //Line 12
    intList1.push_back(36);                             //Line 13
    intList1.push_back(15);                             //Line 14
    intList1.push_back(98);                             //Line 15
    intList1.push_back(58);                             //Line 16

    cout << "Line 17: intList1: ";                      //Line 17
    copy(intList1.begin(), intList1.end(), screen);     //Line 18
    cout << endl;                                       //Line 19

    intList2 = intList1;                                //Line 20

    cout << "Line 21: intList2: ";                      //Line 21
    copy(intList2.begin(), intList2.end(), screen);     //Line 22
    cout << endl;                                       //Line 23

    intList1.unique();                                  //Line 24

    cout << "Line 25: After removing the consecutive "
         << "duplicates," << endl
         << "         intList1: ";                      //Line 25
    copy(intList1.begin(), intList1.end(), screen);     //Line 26
    cout << endl;                                       //Line 27

    intList2.sort();                                    //Line 28

    cout << "Line 29: After sorting, intLis
    copy(intList2.begin(), intList2.end(),
    cout << endl;

    return 0;
}
```

Sample Run:

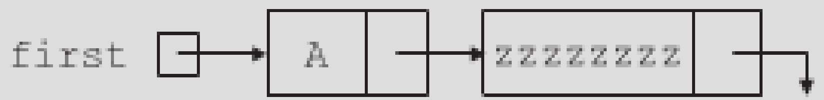
```
Line 17: intList1: 23 58 58 36 15 98 58
Line 21: intList2: 23 58 58 36 15 98 58
Line 25: After removing the consecutive duplicates,
         intList1: 23 58 36 15 98 58
Line 29: After sorting, intList2: 15 23 36 58 58 58 98
```

Linked Lists with Header and Trailer Nodes

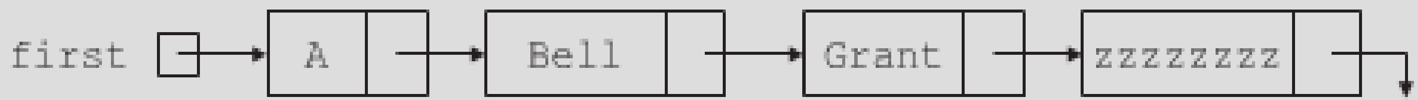
- Simplify insertion and deletion
  - Never insert item before the first or after the last item
  - Never delete the first node
- Set header node at beginning of the list
  - Containing a value smaller than the smallest value in the data set
- Set trailer node at end of the list
  - Containing value larger than the largest value in the data set

Linked Lists with Header and Trailer Nodes (cont'd.)

- Header and trailer nodes
  - Serve to simplify insertion and deletion algorithms
  - Not part of the actual list
- Actual list located between these two nodes



(a) Empty linked list with header and trailer nodes



(b) Nonempty linked list with header and trailer nodes

Circular Linked Lists

- Last node points to the first node
- Basic operations
  - Initialize list (to an empty state), determine if list is empty, destroy list, print list, find the list length, search for a given item, insert item, delete item, copy the list

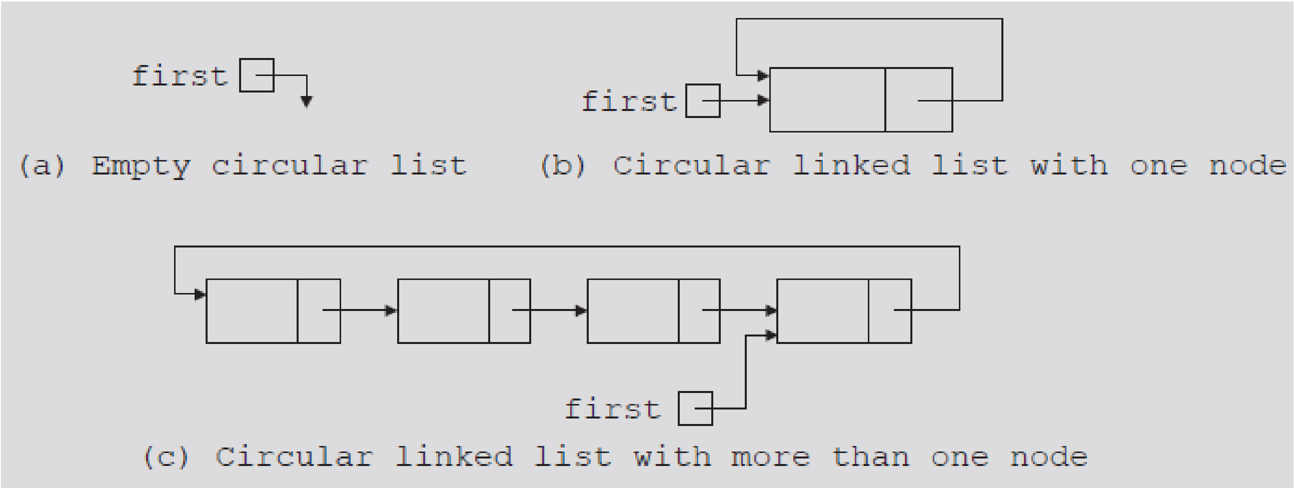


FIGURE 5-34 Circular linked lists

Discussion

- Discuss the STL container list. Describe the benefits of using this container.

- Discuss ordered and unordered linked lists. What are the benefits of each? Think of applications applicable to each type of linked list and determine when to switch from an unordered list to an ordered list.

Summary

- Linked list topics
  - Traversal, searching, inserting, deleting
- Building a linked list
  - Forward, backward
- Linked list as an ADT
- Ordered linked lists
- Doubly linked lists
- STL sequence container `list`
- Linked lists with header and trailer nodes
- Circular linked lists

Resources

- http://xlinux.nist.gov/dads/HTML/linkedList.html
- www.codeproject.com/KB/cpp/linked_list.aspx
- http://xlinux.nist.gov/dads/HTML/doublyLinkedList.html
- http://msdn.microsoft.com/en-us/library/1fe2x6kt.aspx
- http://xlinux.nist.gov/dads/HTML/circularlist.html
- www.devx.com/getHelpOn/10MinuteSolution/16976

Self Exercises

- Programming Exercises: 2, 3, 4, 5, 6, 10, 13