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

Objectives (cont’d.)

- 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

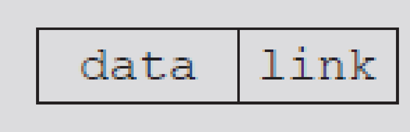


FIGURE 5-1 Structure of a node

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

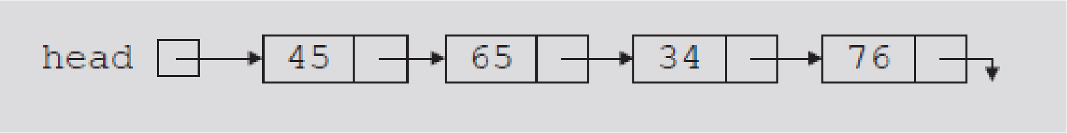


FIGURE 5-2 Linked list

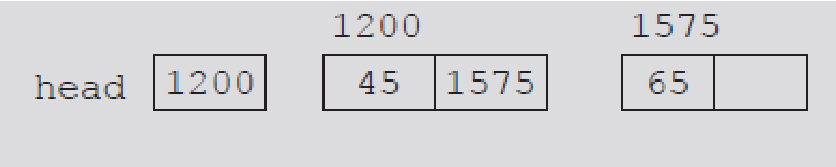
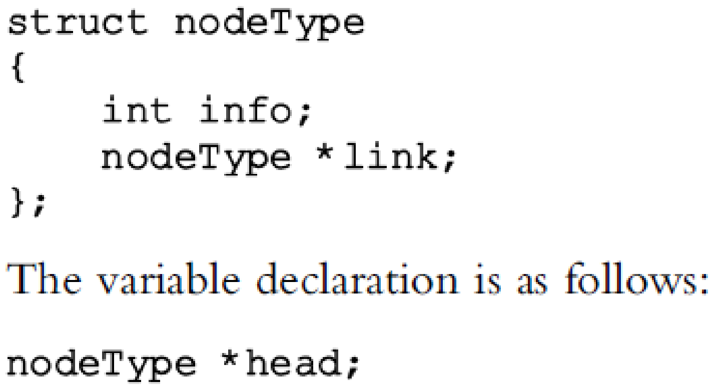


FIGURE 5-3 Linked list and values of the links

Linked Lists (cont'd.)

- Two node components
  - Declared as a `class` or `struct`
    - 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

- Head stores address of first node
- Info stores information
- Link stores address of next node
  - Assume info type `int`

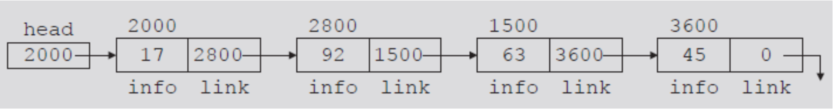
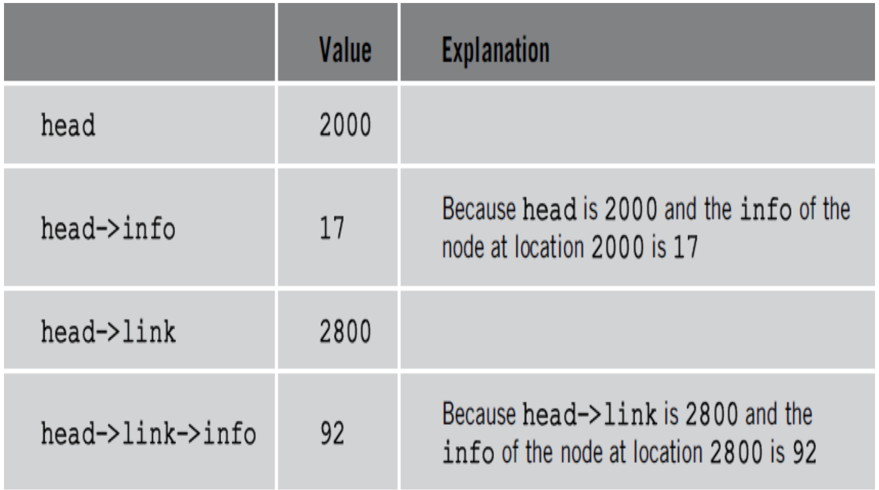


FIGURE 5-4 Linked list with four nodes

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`

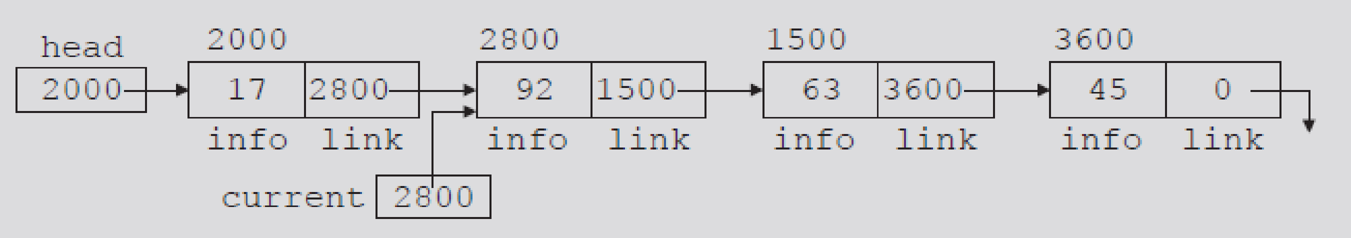
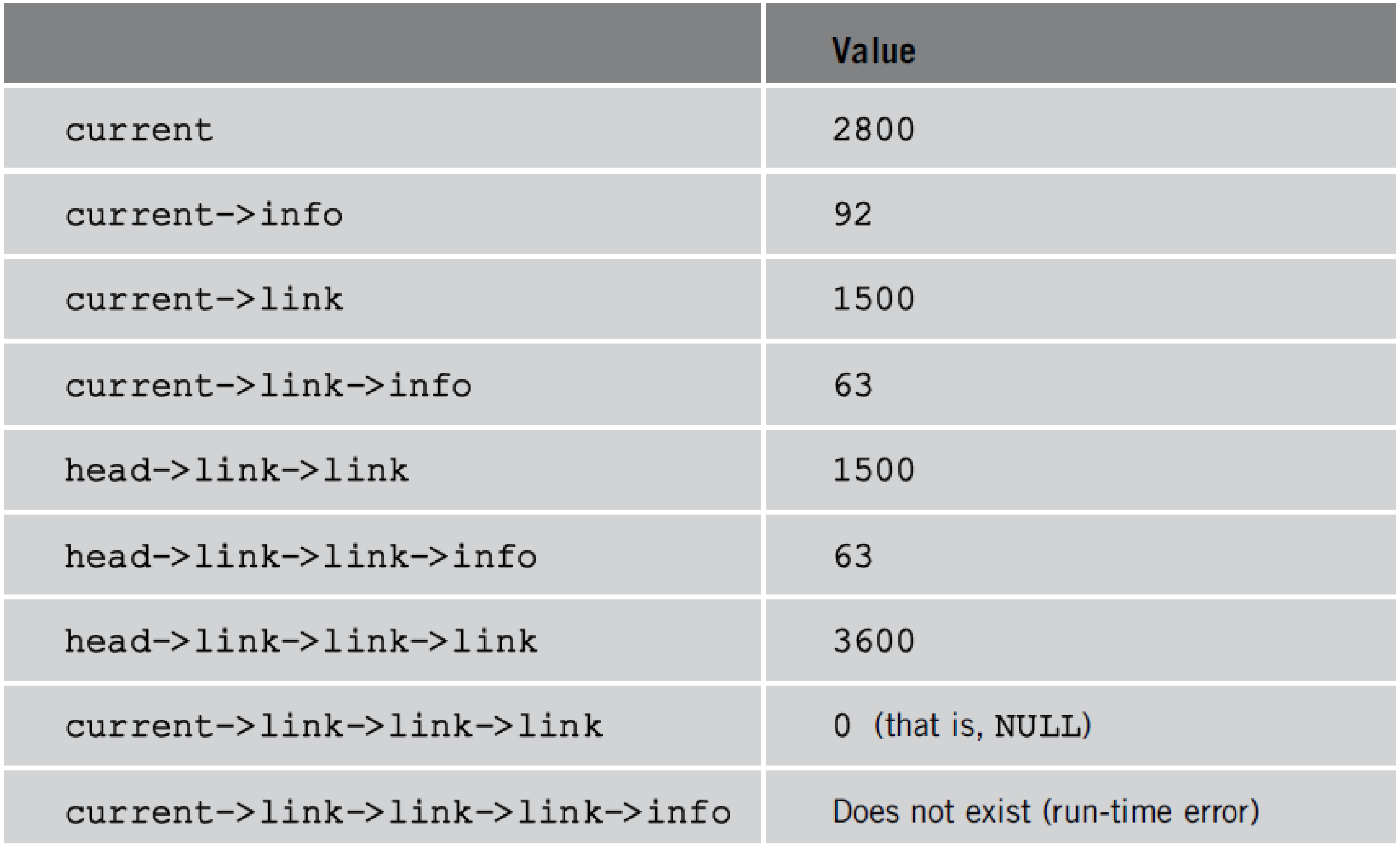


FIGURE 5-5 List after the statement `current = current->link;` executes

Linked Lists: Some Properties (cont’d.)

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



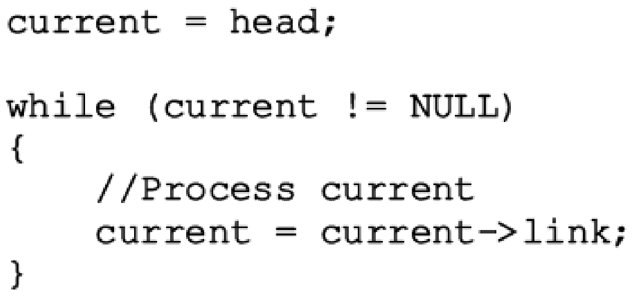
	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63
`head->link->link`	1500
`head->link->link->info`	63
`head->link->link->link`	3600
`current->link->link->link`	0 (that is, NULL)
`current->link->link->link->info`	Does not exist (run-time error)

Traversing a Linked List

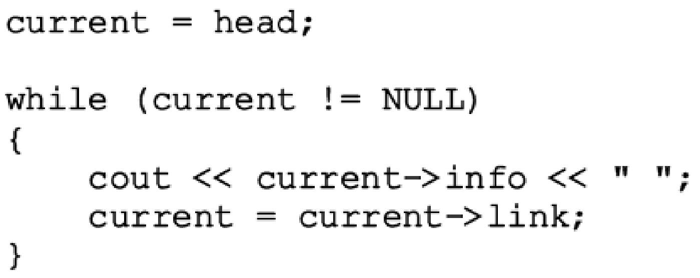
- Basic linked list operations
  - Search list to determine if particular item is in the list
  - Insert item in list
  - Delete item from list
- These operations require list traversal
  - Given pointer to list first node, we must step through list nodes

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;
}
```



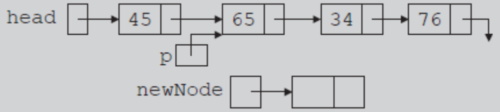
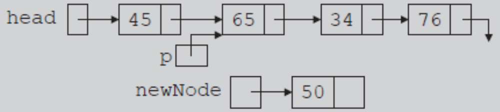
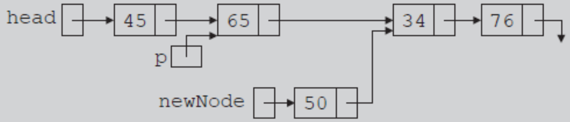
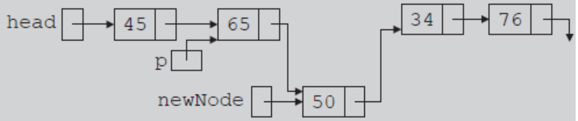
```
current = head;

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

Item Insertion and Deletion

- Generic definition of a node on 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.)

- Sequence of statements to insert node
  - Very important
    - Use only one pointer (`p`) to adjust links of the nodes
- Using two pointers
  - Can simplify insertion code somewhat

Item Insertion and Deletion (cont'd.)

- Memory still occupied by node after deletion
  - Memory is inaccessible
  - Deallocate memory using a pointer to this node

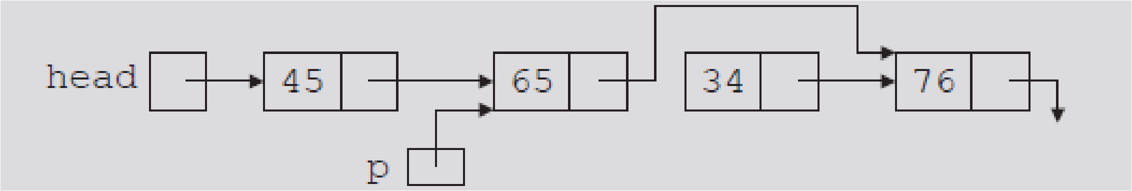


FIGURE 5-10 List after the statement `p->link = p->link->link;` executes

Building a Linked List

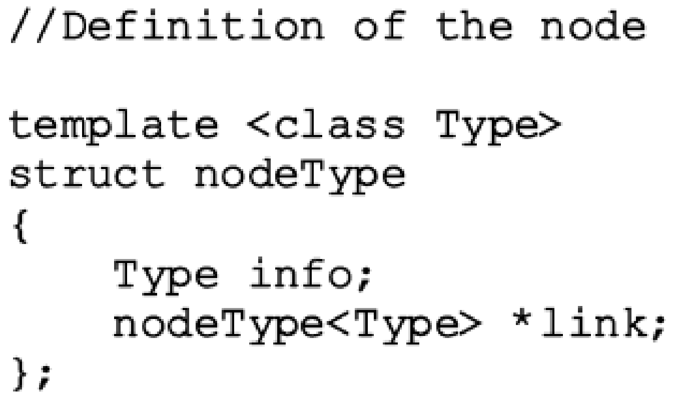
- If linked list data unsorted
  - Linked list unsorted
- Ways to build linked list
  - Forward
    - New node always inserted at end of the linked list
    - See example on page 274
    - See function `buildListForward` on page 277
  - Backward
    - New node always inserted at the beginning of the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

Linked List as an ADT

- 11 basic operations
- Two types of linked lists: sorted, unsorted
- `class linkedListType`
  - Implements basic linked list operations as an ADT
  - Derive two classes using inheritance
    - `unorderedLinkedList` and `orderedLinkedList`
- Unordered linked list functions
  - `buildListForward` and `buildListBackward`
  - Two more functions accommodate both operations
    - `insertFirst` and `insertLast`

Structure of Linked List Nodes

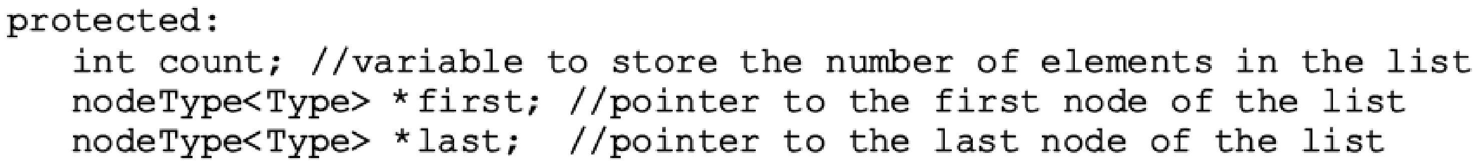
- Node has two instance variables
  - Simplify operations (insert, delete)
    - Define class to implement linked list node as a `struct`
- Definition of the `struct nodeType`



```
//Definition of the node

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

Member Variables of the `class linkedListType`

- `class linkedListType`
  - Three instance variables



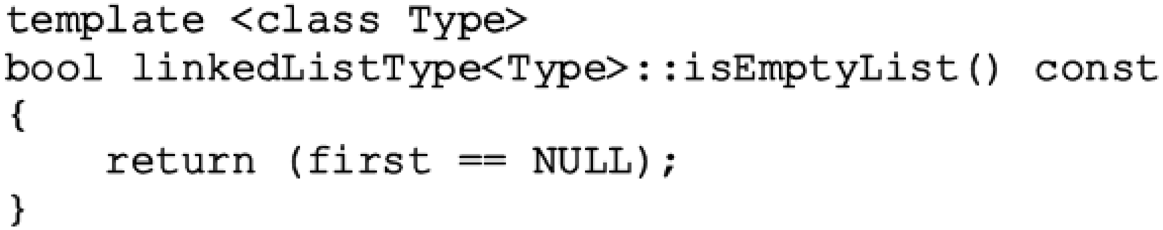
```
protected:
   int count; //variable to store the number of elements in the list
   nodeType<Type> *first; //pointer to the first node of the list
   nodeType<Type> *last;  //pointer to the last node of the list
```

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
  - `class linkedListType`
  - Functions of `class linkedListIterator`

Linked List Iterators (cont'd.)

- 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

Linked List as an ADT (cont'd.)

- Print the list
  - Must traverse the list starting at first node
- 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

Linked List as an ADT (cont'd.)

- Retrieve the data of the last node
  - Function `back`
    - Returns info contained in the last node
    - If list is empty, `assert` statement terminates program
- Begin and end
  - Function `begin` returns an iterator to the first node in the linked list
  - Function `end` returns an iterator to the last node in the linked list

Linked List as an ADT (cont'd.)

- Copy the list
  - Makes an identical copy of a linked list
    - Create node called `newNode`
    - Copy node info (original list) into `newNode`
    - Insert `newNode` at the end of list being created
  - See function `copyList` on page 289

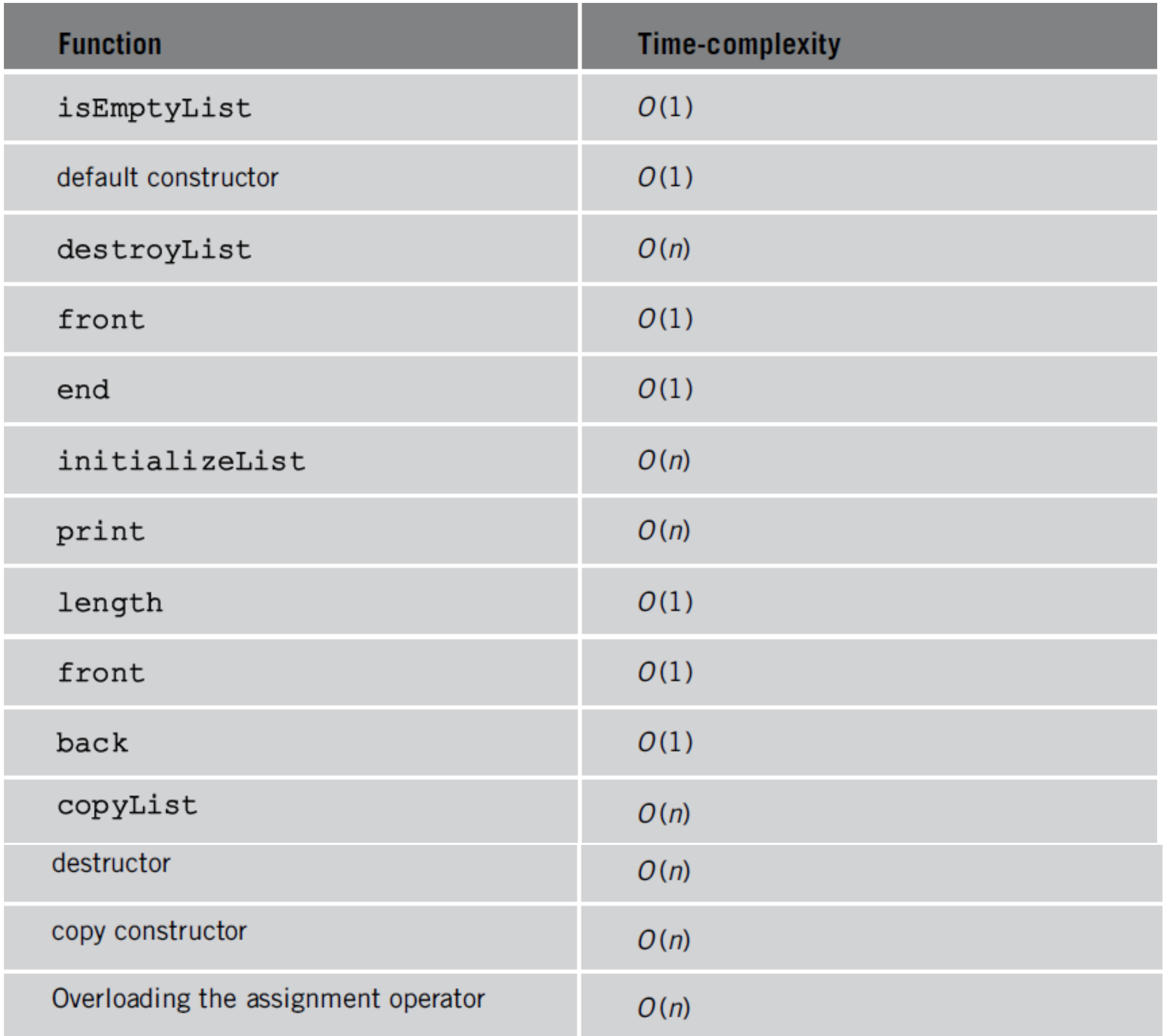
Linked List as an ADT (cont'd.)

- 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

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`
- Overloading the assignment operator
  - Similar to copy constructor definition

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
  - `class unorderedLinkedList`

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

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

Unordered Linked Lists (cont'd.)

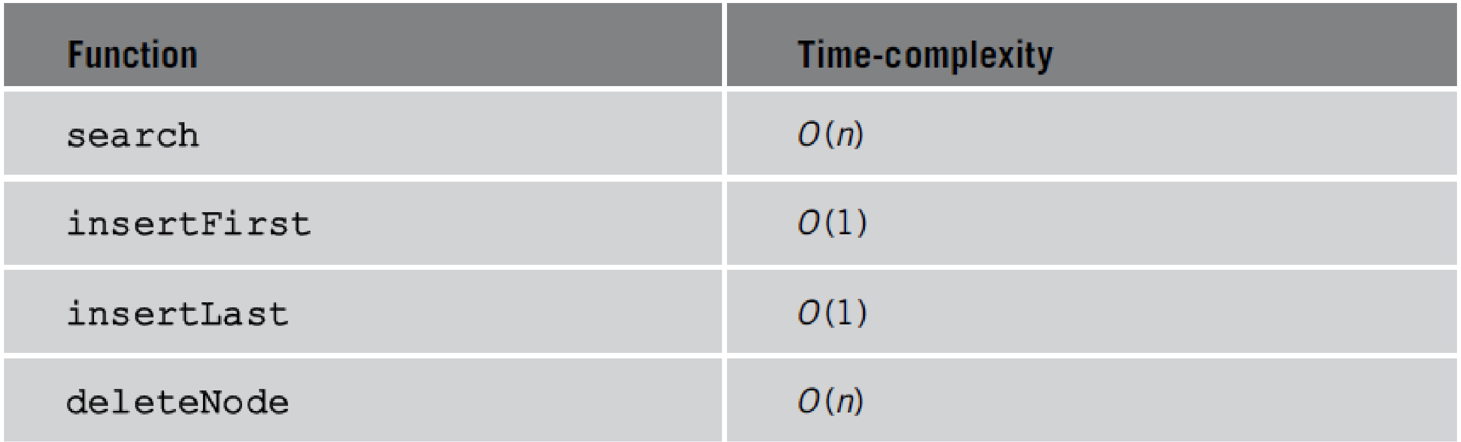
- Insert the last node
  - Similar to definition of member function `insertFirst`
  - Insert new node after last
  - See function `insertLast` on page 294

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

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`

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.)

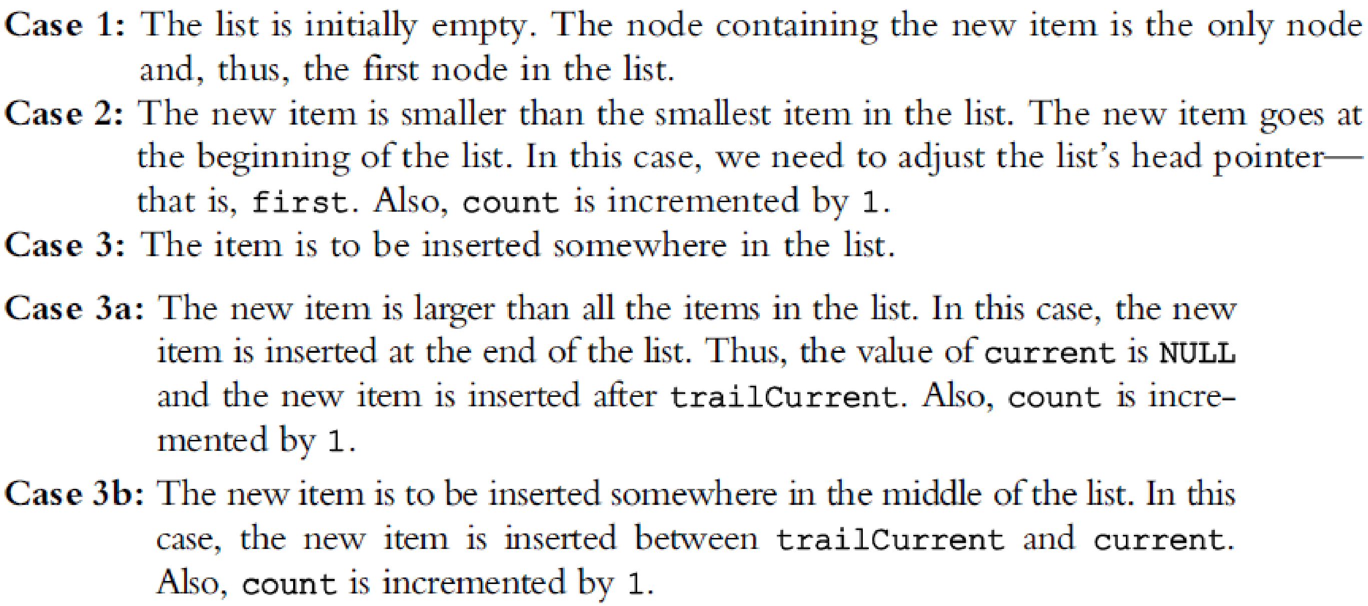
- Search the list
  - Steps describing algorithm
    - Step one: Compare the search item with the current node in the list. If the info of the current node is greater than or equal to the search item, stop the search; otherwise, make the next node the current node
    - Step two: Repeat Step one until 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

Ordered Linked Lists (cont'd.)

- Insert a node
  - Find place where new item goes
    - Insert item in the list
  - See code on page 304
    - Definition of the function `insert`

Ordered Linked Lists (cont'd.)

- Insert a node (cont'd.)
  - Consider the following cases



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`.

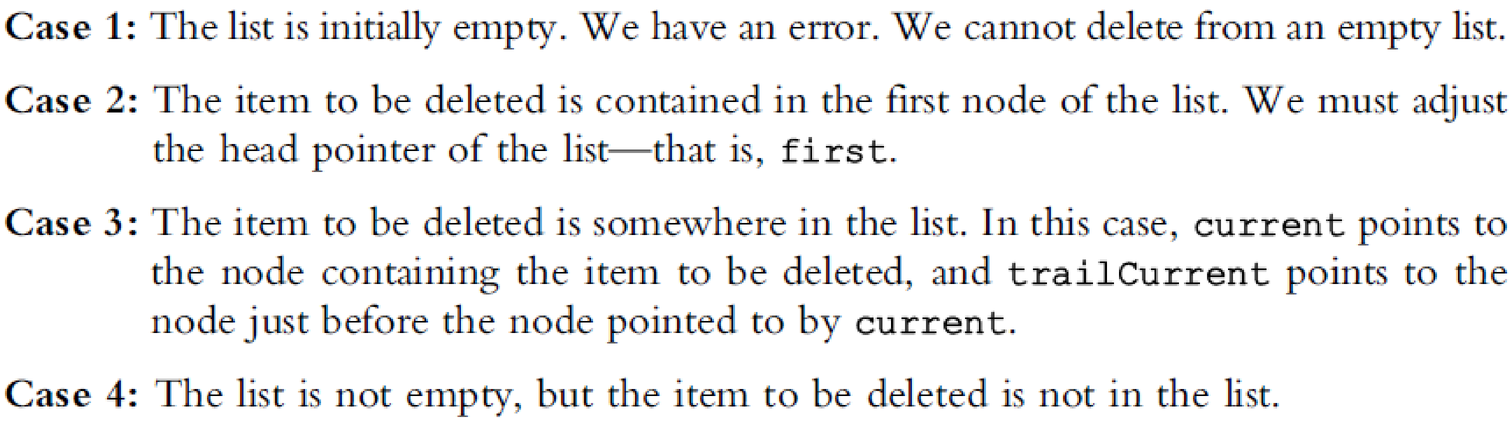
Case 3b: The new item is to be inserted somewhere in the middle of the list. In this case, the new item is inserted between `trailCurrent` and `current`. Also, `count` is incremented by `1`.

Ordered Linked Lists (cont’d.)

- Insert first and insert last
  - Function `insertFirst`
    - Inserts new item at beginning of the list
    - 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.

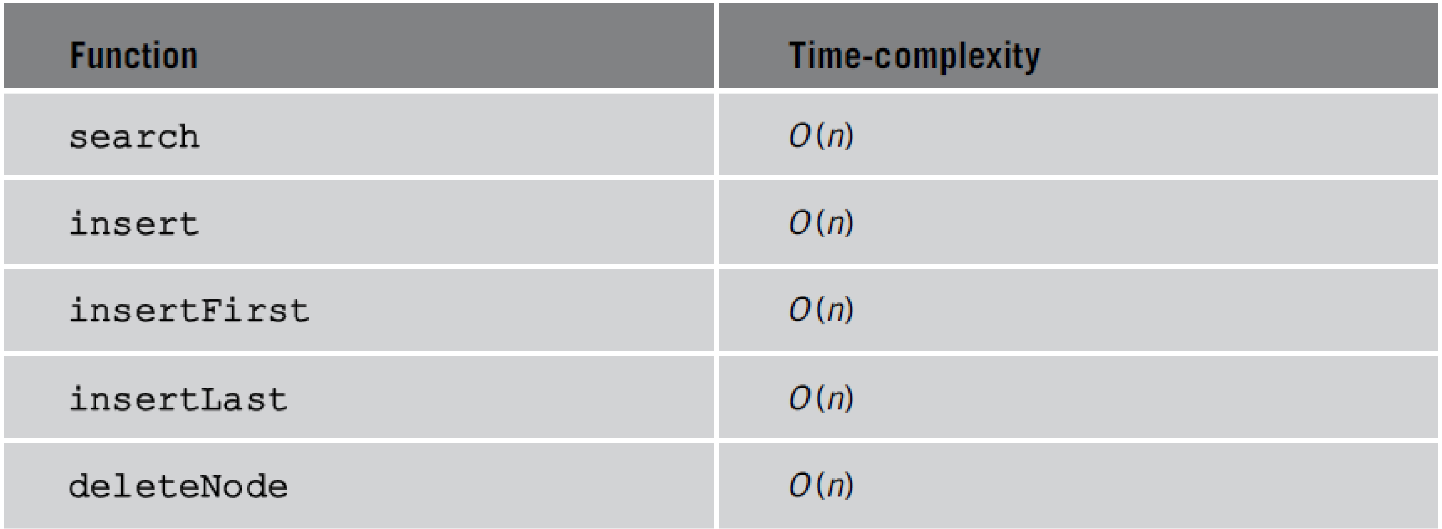
Case 2: The item to be deleted is contained in the first node of the list. We must adjust the head pointer of the list—that is, `first`.

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.

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
- 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

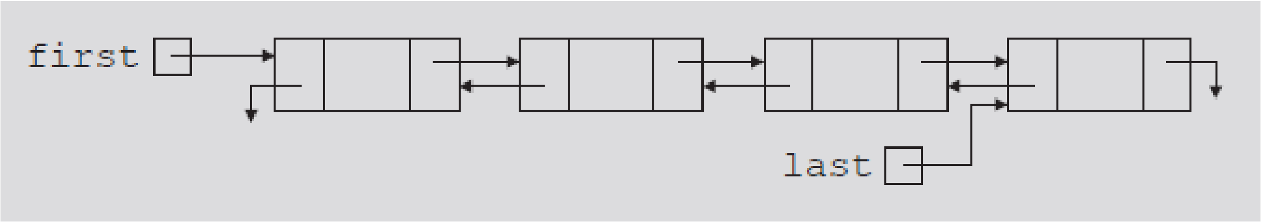


FIGURE 5-27 Doubly linked list

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`

Doubly Linked Lists (cont'd.)

- 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
- 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`
    - Returns value of this variable

Doubly Linked Lists (cont'd.)

- 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
- Search the list
  - – Function `search` returns true if `searchItem` found
    - Otherwise, it returns false
  - – Same as ordered linked list search algorithm

Doubly Linked Lists (cont'd.)

- First and last elements
  - Function `front` returns first list element
  - Function `back` returns last list element
  - If list empty
    - Functions 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 3 and 4: After inserting an item, count incremented by one

Doubly Linked Lists (cont’d.)

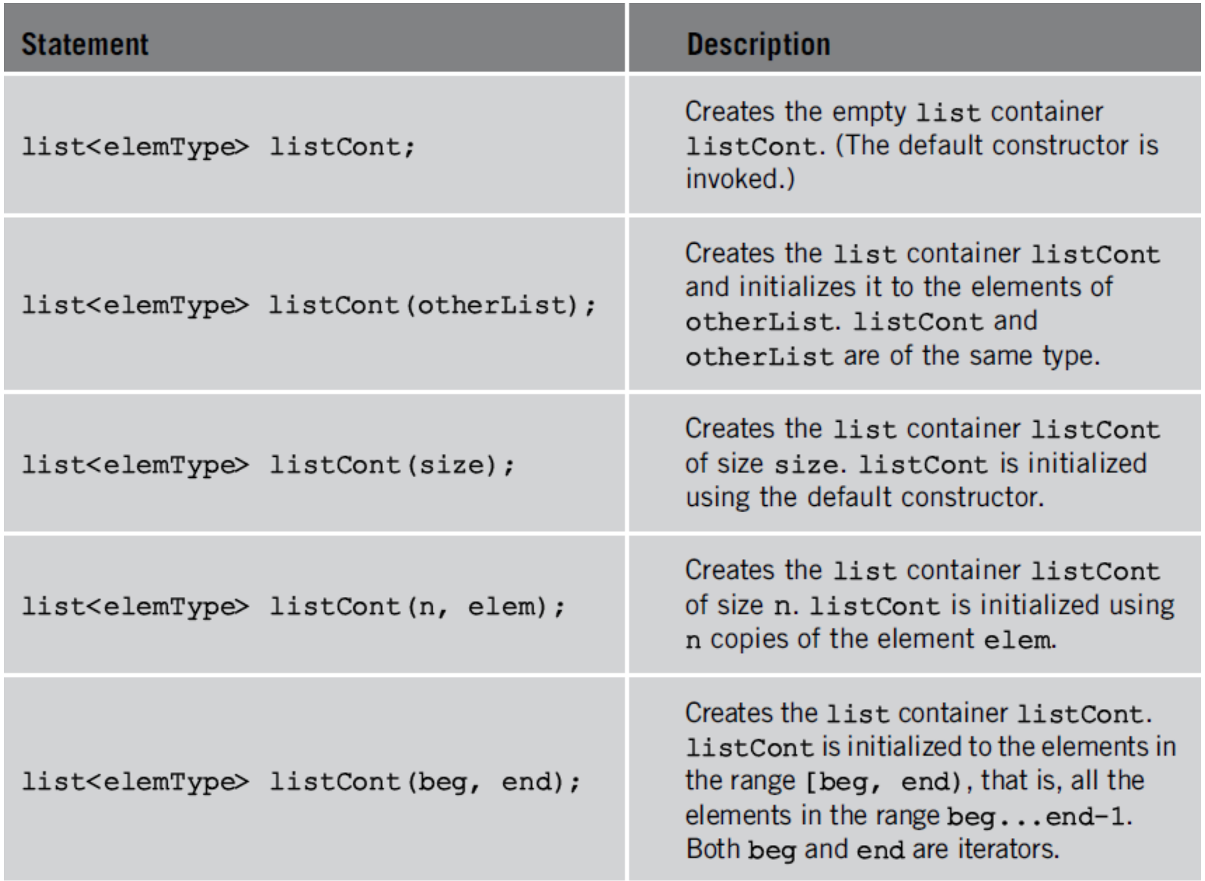
- Insert a node (cont’d.)
  - Figures 5-28 and 5-29 illustrate case 4
  - See code on page 317
    - Definition of the function `insert`

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`

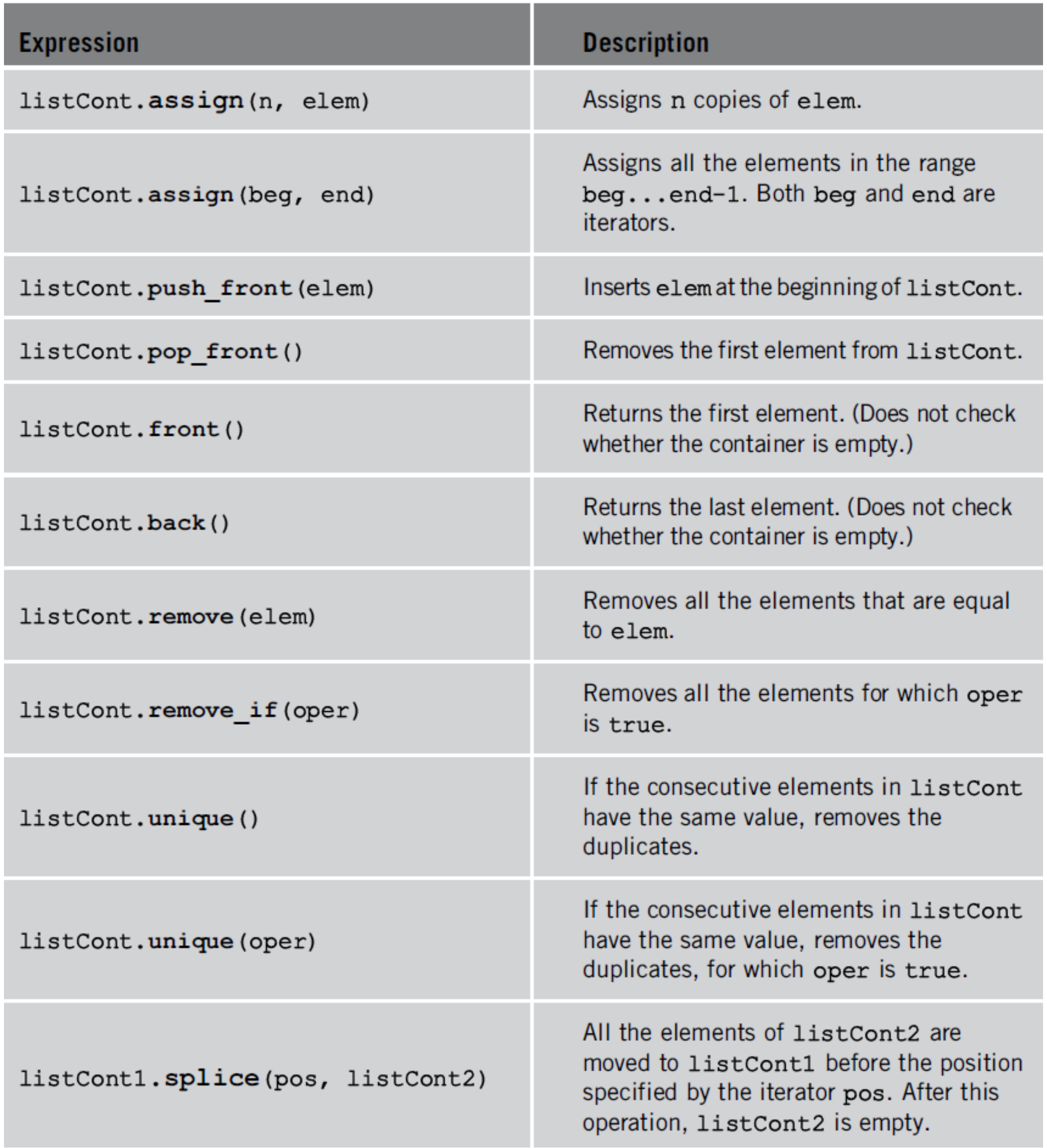
STL Sequence Container: `list`

TABLE 5-9 Various ways to declare a list object



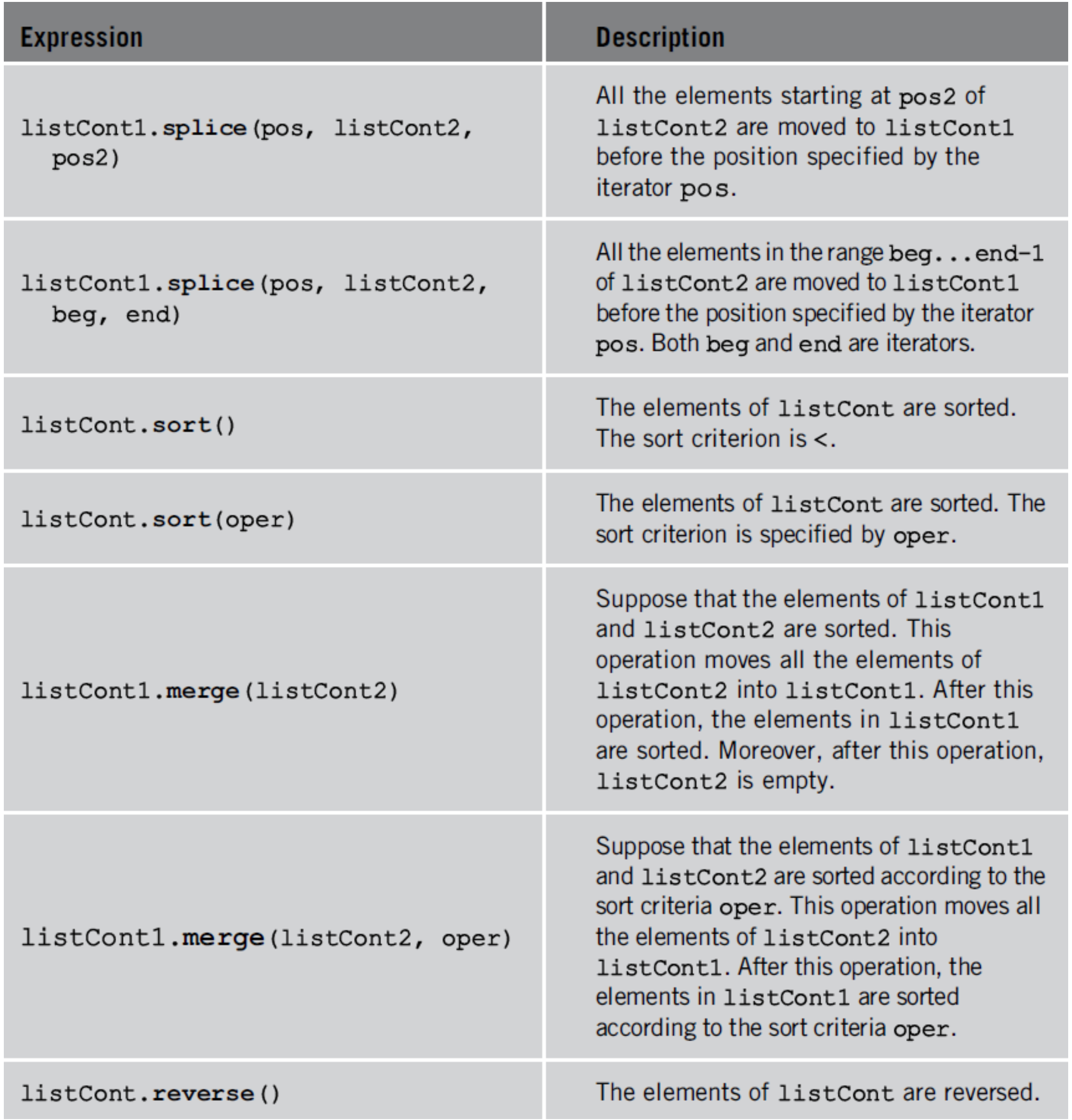
Statement	Description
`list<elemType> listCont;`	Creates the empty `list` container `listCont`. (The default constructor is invoked.)
`list<elemType> listCont(otherList);`	Creates the `list` container `listCont` and initializes it to the elements of `otherList`. `listCont` and `otherList` are of the 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.

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

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

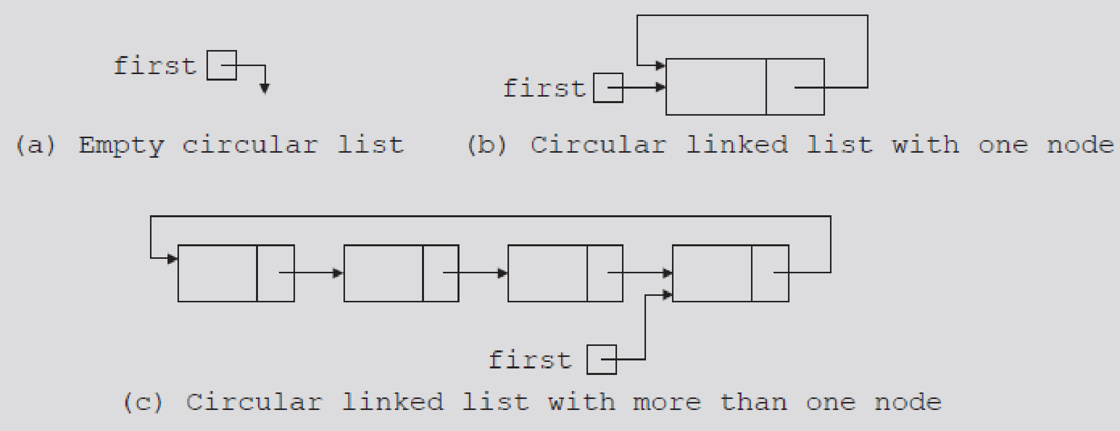

first
(a) Empty circular list
first
(b) Circular linked list with one node
first
(c) Circular linked list with more than one node

FIGURE 5-34 Circular linked lists

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