Chapter 17 - Linked Lists

CS 202

[TOC]

Objectives (1 of 2)

In this chapter, you will:

- Learn about linked lists
- Become familiar with 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 (2 of 2)

- Learn how to implement linked lists as Abstract Data Types (ADTs)
- Learn how to create linked list iterators (objects for traversing nodes)
- Implement the basic operations on a linked list
- Create unordered and ordered linked lists
- Become familiar with circular and doubly linked lists

Introduction

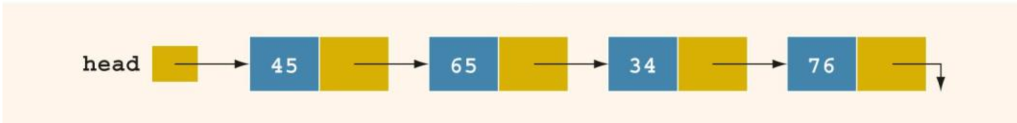
- Data can be organized and processed sequentially using an array, called a sequential list
- Problems with an array:
  - Array size is fixed
  - **Unsorted array**: searching for an item is slow
  - **Sorted array**: insertion and deletion are slow due to data movement

Linked Lists (1 of 3)

- **Linked list**: a collection of items (**nodes**) containing two components:
  - Data
  - Address (**link**) of the next node in the list

Structure of a node

Linked Lists (2 of 3)



Linked List

Linked Lists (3 of 3)

- A node is declared as a `class` or `struct`
  - Data type of a node depends on the specific application
  - Link component of each node is a pointer

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

- Variable declaration:

```
nodeType* head = nullptr; // Initialize head pointer to nullptr
```

Linked Lists: Some Properties (1 of 3) - Example: linked list with four nodes (Figure 17-4)

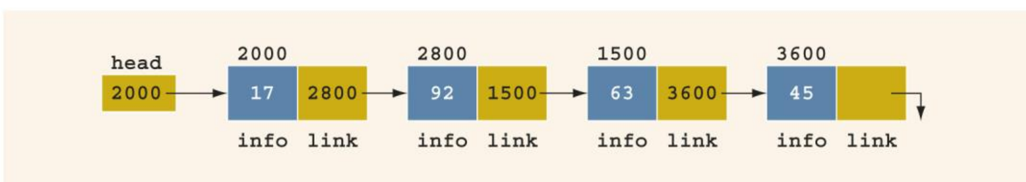


FIGURE 17-4 Linked list with four nodes

	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 (2 of 3)

```
current = head;
```

- Copies value of `head` into `current`

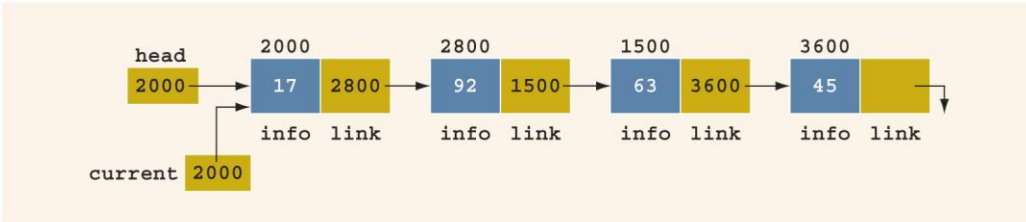


FIGURE 17-5 Linked list after the statement **current = head**; executes

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

Linked Lists: Some Properties (3 of 3)

```
current = current->link;
```

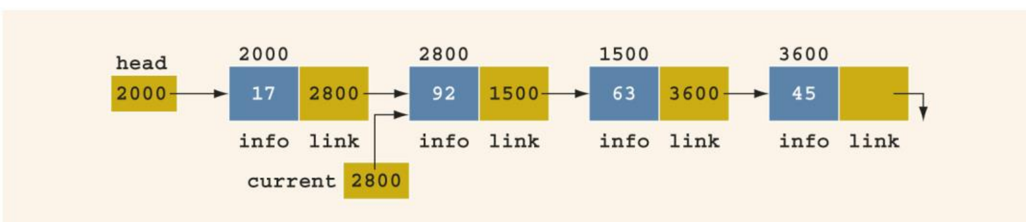


FIGURE 17-6 List after the statement **current = current->link**; executes

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

Traversing a Linked List (1 of 2)

- Basic operations of a linked list:
  - Search for an item 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 (2 of 2)

- To traverse a linked list:

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

- Example:

```
current = head;
while (current != nullptr) {
    cout << current->info << ' ';
    current = current->link;
}
```

Item Insertion and Deletion

- Definition of a node:

```
struct nodeType {
    int       info {};  // default (0)
    nodeType* link {};  // default (nullptr)
};
```

- Variable declaration:

```
nodeType* head = nullptr;  // explicit initialization
nodeType* tail {};         // default value initialization
nodeType* p {};
nodeType* q {};
nodeType* newNode {};
```

Insertion (1 of 4)

- To insert a new node with `info` 50 after p in this list:

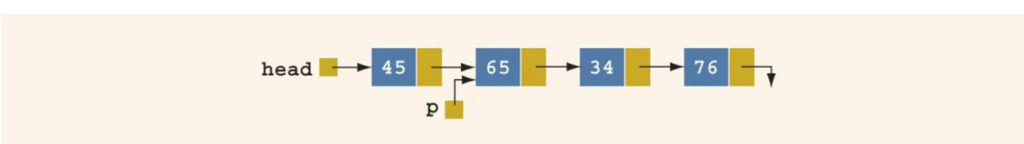


FIGURE 17-7 Linked list before item insertion

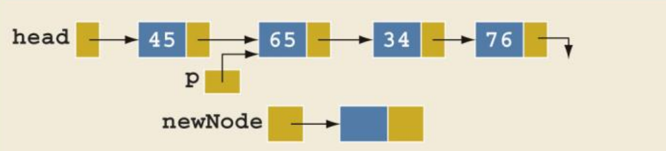
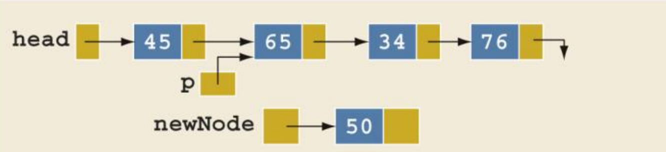
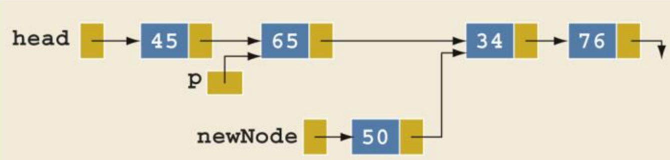
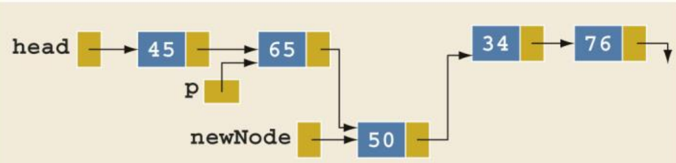
```
newNode         = new nodeType;  // create newNode
newNode->info = 50;              // store 50 in new node
newNode->link = p->link;
p->link         = newNode;
```

or

```
p->link = new nodeType { 50, p->link };
```

Insertion (2 of 4)

TABLE 17-1 Inserting a Node in a Linked List

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

Insertion (3 of 4)

- Using two pointers can simplify insertion code:

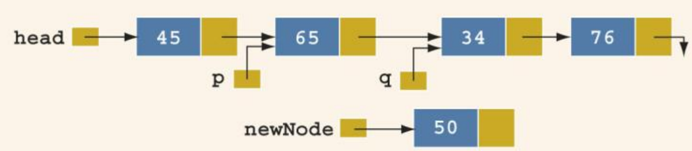


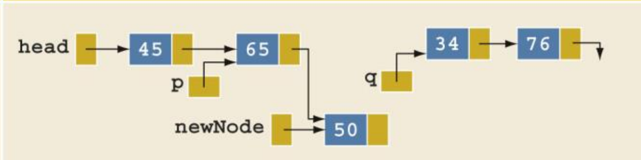
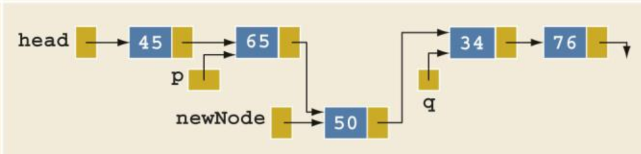
FIGURE 17-9 List with pointers p and q

- To insert newNode between p and q:

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

Insertion (4 of 4)

TABLE 17-2 Inserting a Node in a Linked List Using Two Pointers

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

Deletion (1 of 3)

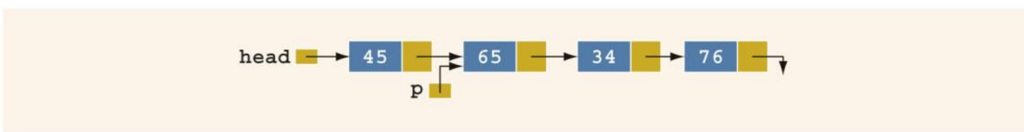


FIGURE 17-10 Node to be deleted is with `info` 34

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

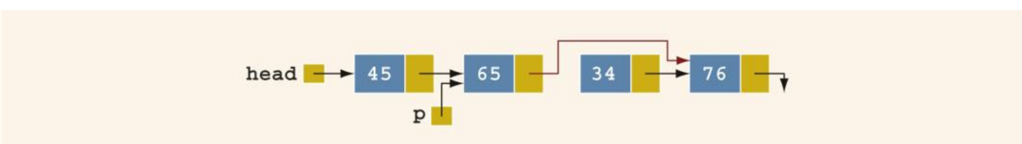


FIGURE 17-11 List after the statement `p->link = p->link->link` executes.

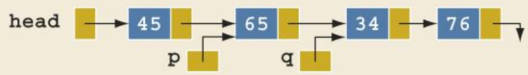
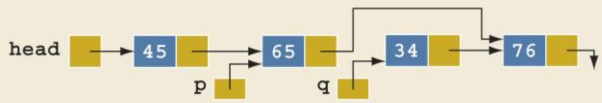
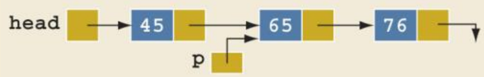
Deletion (2 of 3)

- Node with `info` 34 is removed from the list, but memory is still occupied
  - Node is dangling
  - Must keep a pointer to the node to deallocate memory

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

Deletion (3 of 3)

TABLE 17-3 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
- Linked list can be built 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 (1 of 4)

- Requires three pointers:
  - One to point to the first node in the list
  - One to point to the last node in the list
  - One to create the new node
- Example:
  - Data: 2 15 8 24 34

Building a Linked List Forward (2 of 4)

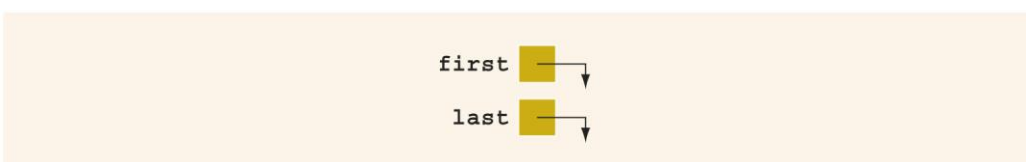


FIGURE 17-12 Empty list

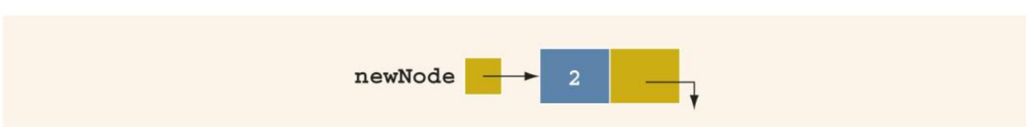


FIGURE 17-13 `newNode` with `info` 2

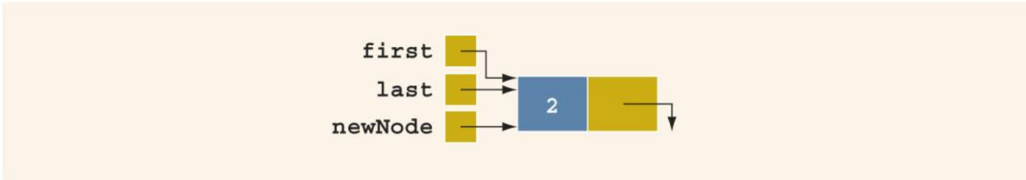


FIGURE 17-14 List after inserting **newNode** in it

Building a Linked List Forward (3 of 4)

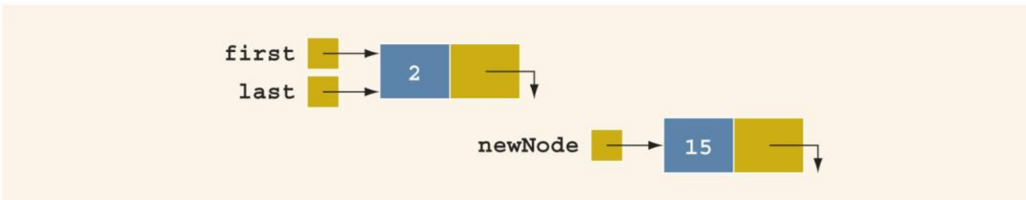


FIGURE 17-15 List and **newNode** with **info 15**

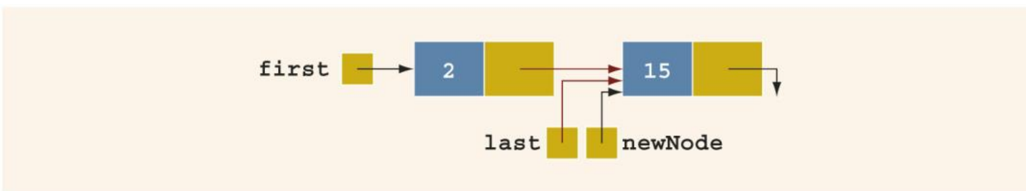


FIGURE 17-16 List after inserting **newNode** at the end

Building a Linked List Forward (4 of 4)

- Repeat this process three more times:

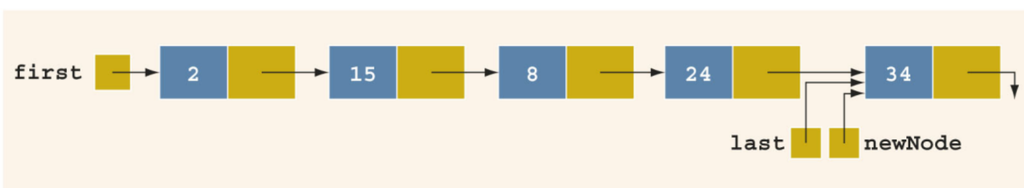


FIGURE 17-17 List after inserting **8**, **24**, and **34**

Building a Linked List Backward

- Algorithm to build a linked list backward:
  - Initialize head to `nullptr`
  - For each item in the list
    - Create new node newNode
    - Store data in newNode
    - Insert newNode before head
    - Update head

Linked List as an ADT

- Basic operations on linked lists:
  - Initialize the list
  - Determine whether the list is empty
  - Print the list
  - Find the list's length
  - Destroy the list
  - Retrieve data in the first or last node
  - Search for a given item
  - Insert an item
  - Delete an item
  - Make a copy of the list

Structure of Linked List Nodes

- Each node has two members:
  - Data
  - Link to next node
- Definition of the struct nodeType:

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

Member Variables of the class linkedListType

- **linkedListType** has three members:
  - Two pointers: **head** and **tail**
  - `count`: the number of nodes, of type `size_type`

```
protected:
    size_type count;          // number of elements
    nodeType<T>* head;        // pointer to first node
    nodeType<T>* tail;        // pointer to last node
```

Linked List Iterators (1 of 3)

- Processing nodes requires traversal from the first node
- **Iterator**: provides each element of a container sequentially, simplifying traversal logic
  - Common iterator operations:
    - ++ (pre-increment)

        - * (dereference)
        - -> (accesses current node directly)

Linked List Iterators (2 of 3)

- An iterator is an object
    - Define class **linkedListIterator** for iterating over **linkedListType**
    - Has member variable for the current node

Linked List Iterators (3 of 3)

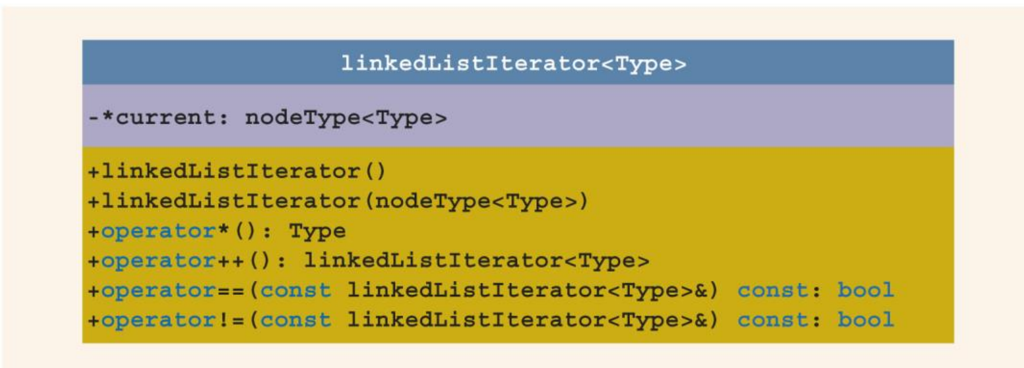


FIGURE 17-19 UML class diagram of the class linkedListIterator

Default Constructor

- Initializes the list to an empty state

```
template <class T>
doublyLinkedList<T>::doublyLinkedList() {
    count = 0;
    head = nullptr;
    tail = nullptr;
}
```

Or, with member initializer list:

```
template <class T>
doublyLinkedList<T>::doublyLinkedList()
: count(0), head(nullptr), tail(nullptr) {}
```

Length of a List

- **length** (size):
    - Returns the node count as `size_type`

Retrieve the Data of the First or Last Node

- **front**:
    - Returns data in the first node

- Terminates if list is empty

- **back**:
    - Returns data in the last node
    - Terminates if list is empty

Begin and End

- **begin**:
    - Returns iterator to first node
- **end**:
    - Returns iterator past last node

Destructor & Copy Constructor

- **Destructor**:
    - Frees memory as object goes out of scope
- **Copy constructor**:
    - Copies linked list

Overloading the Copy Assignment Operator

- Overload copy assignment operator, similar to copy constructor

Move semantics

- Constructing and destructing individual nodes is expensive.
- Implementing a Move constructor and a move assignment operator is particularly important with linked lists.

Ordered Linked Lists (1 of 2)

- **orderedLinkedList**: derived from **linkedListType**
    - Includes **insert** to maintain order

Ordered Linked Lists (2 of 2)

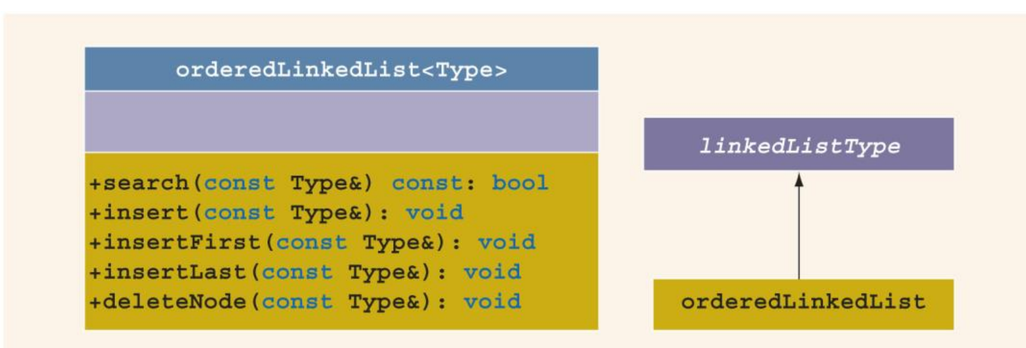


FIGURE 17-29 UML class diagram of the class orderedLinkedList and the inheritance hierarchy

Insert a Node (1 of 4)

- **Case 1**: Empty list

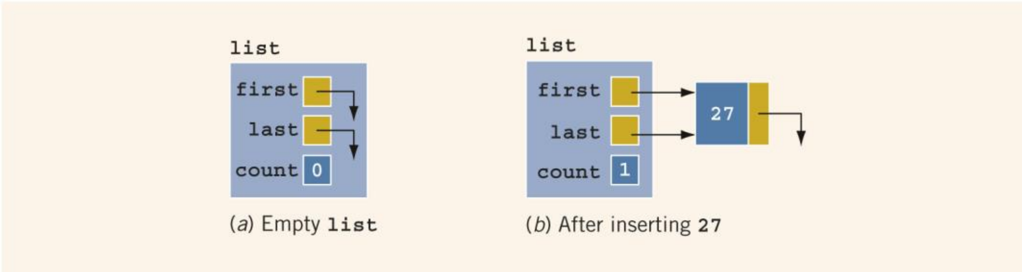


FIGURE 17-30 list

Insert a Node (2 of 4)

- **Case 2**: Insert at beginning

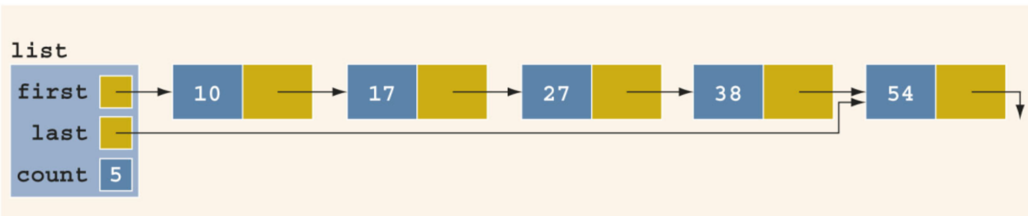


FIGURE 17-32 list after inserting 10

Insert a Node (3 of 4)

- **Case 3**: Insert elsewhere

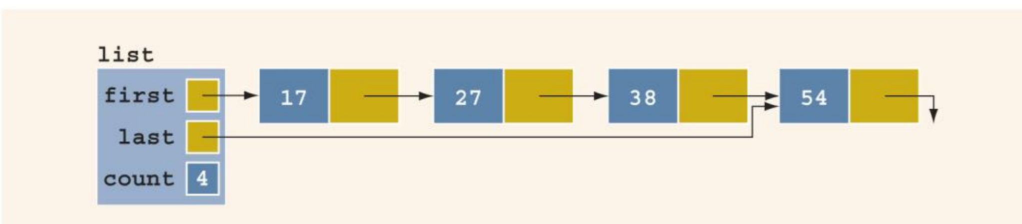


FIGURE 17-33 list before inserting 65

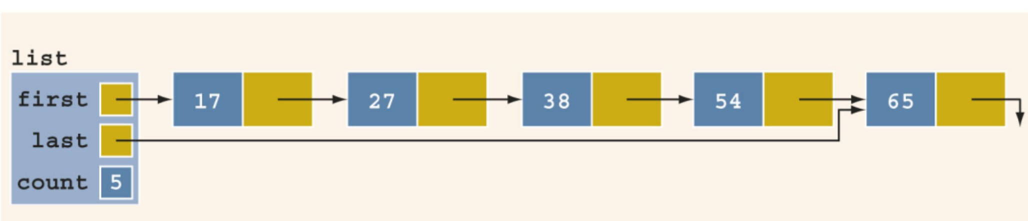


FIGURE 17-34 list after inserting 65

Insert a Node (4 of 4)

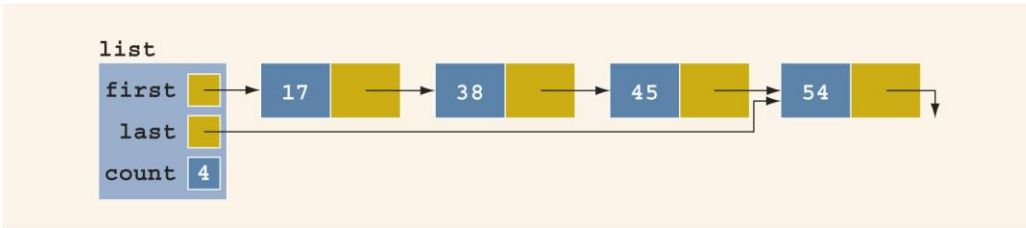


FIGURE 17-35 `list` before inserting `27`

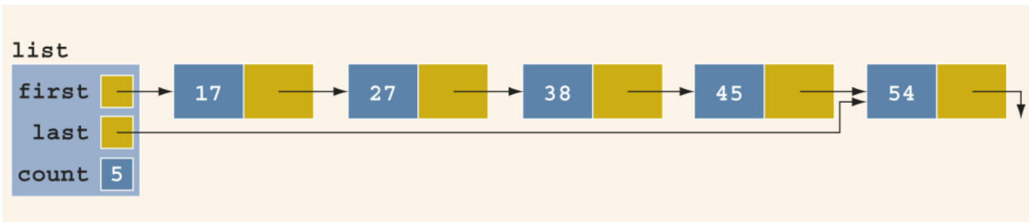


FIGURE 17-36 `list` after inserting `27`

Delete a Node (1 of 4)

- Manages cases:
  - **Case 1:** List empty -> error
  - **Case 2:** First node is deleted
  - **Case 3:** Deleting not-first node
    - **Case 3a**: Deleting non-last
    - **Case 3b**: Deleting last node

Delete a Node (2 of 4)

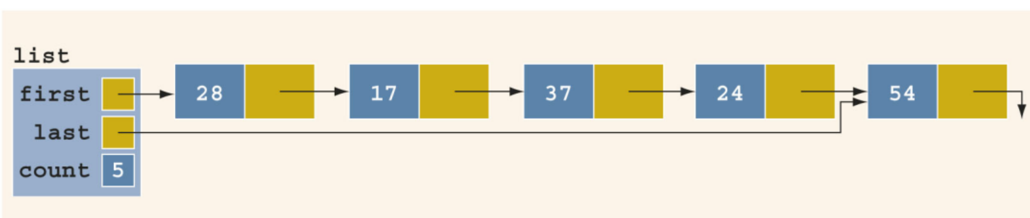


FIGURE 17-23 `list` with more than one node

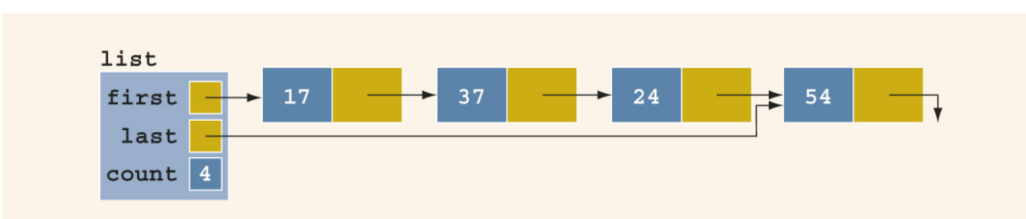


FIGURE 17-24 `list` after deleting node with `info` `28`

Delete a Node (3 of 4)

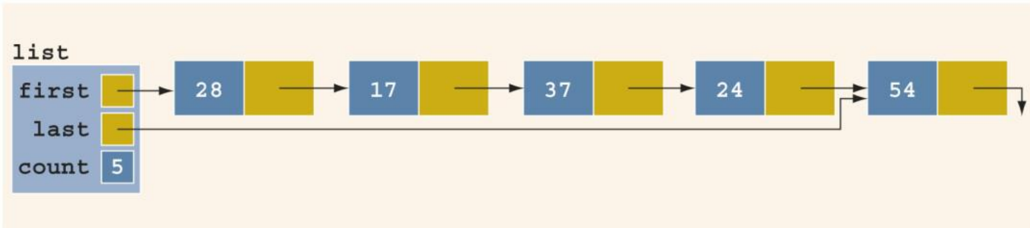


FIGURE 17-25 list before deleting 37

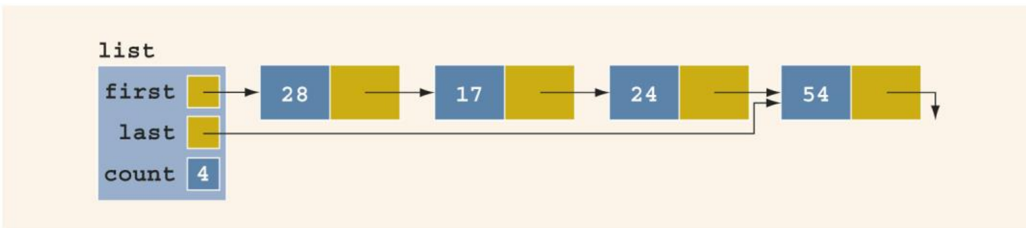


FIGURE 17-26 list after deleting 37

Delete a Node (4 of 4)

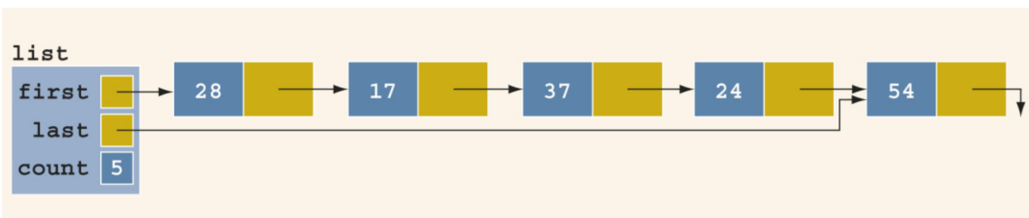


FIGURE 17-27 list before deleting 54

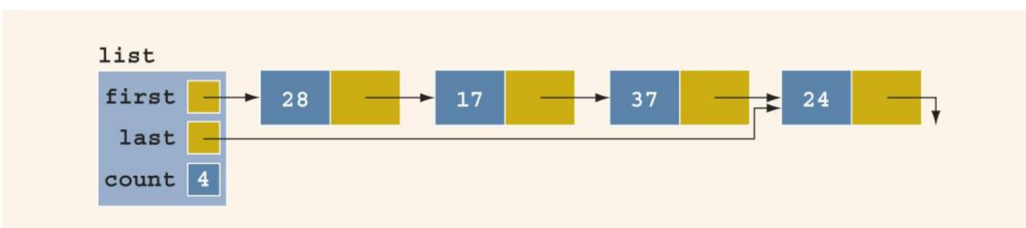


FIGURE 17-28 list after deleting 54

Doubly Linked Lists (1 of 2)

- **Doubly linked list**:
  - Two links: next and back

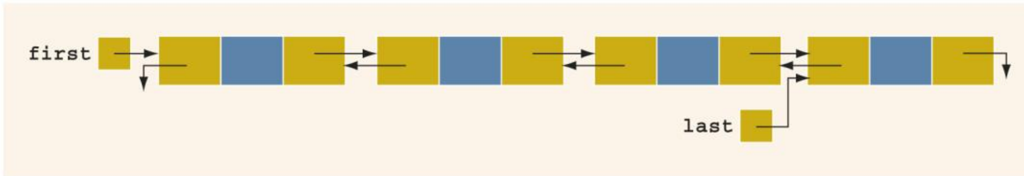


FIGURE 17-39 Doubly linked list

Doubly Linked Lists (2 of 2)

- Common operations:
    - Initialize
    - empty check
    - search
    - retrieve
    - insert
    - delete
    - length
    - print
    - copy

Insert a Node (1 of 2)

- In a doubly-linked list, insertion can occur in several distinct cases, each requiring specific pointer adjustments to maintain the integrity of the list. Here are the primary cases for insertion:

    - **Case 1**: Inserting into an Empty List
    - **Case 2**: Inserting at the Front of the List
    - **Case 3**: Inserting in the Middle of the List
    - **Case 4**: Inserting at the End of the List

Insert a Node (2 of 2)

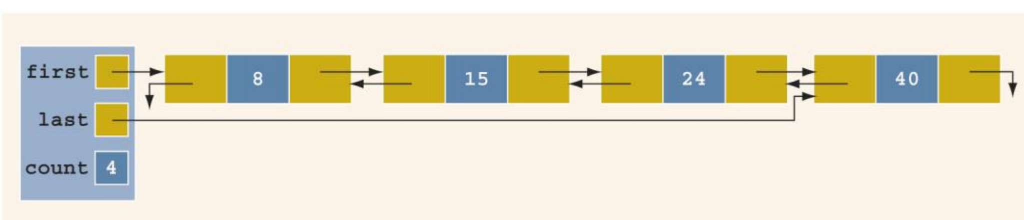


FIGURE 17-40 Doubly linked list before inserting 20

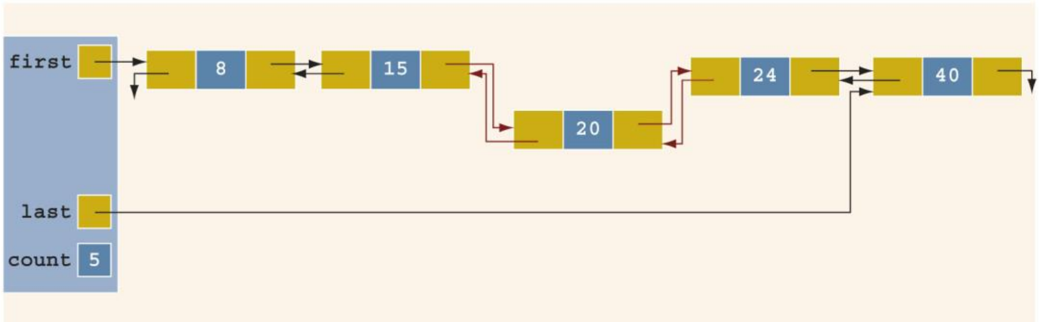


FIGURE 17-41 Doubly linked list after inserting 20

Delete a Node (1 of 3)

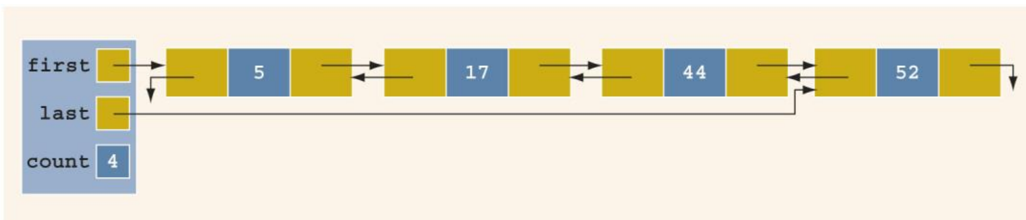


FIGURE 17-42 Doubly linked list before deleting 17

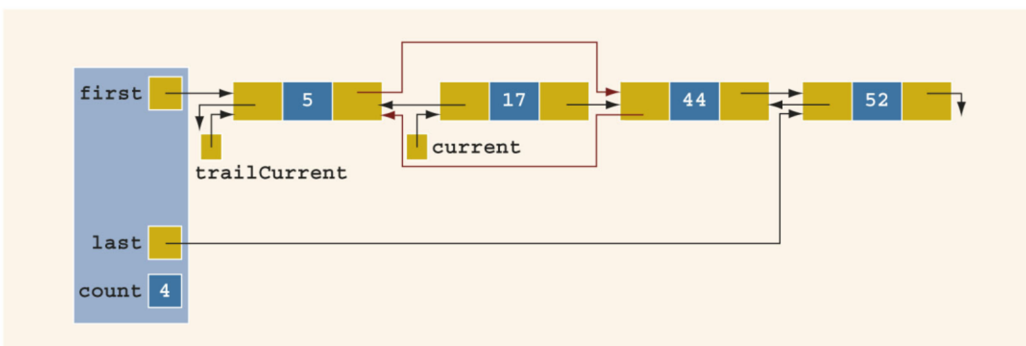


FIGURE 17-43 List after adjusting the links of the nodes before and after the node with info 17

Delete a Node (3 of 3)

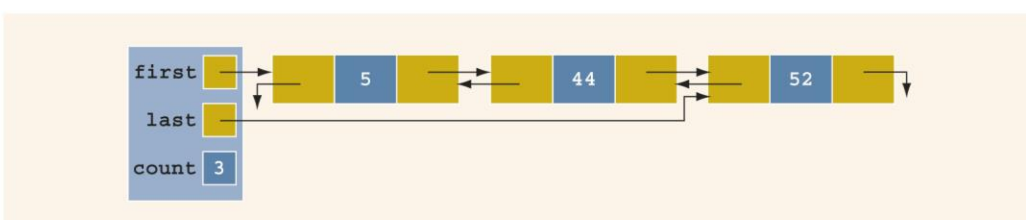


FIGURE 17-44 List after deleting the node with info 17

Circular Linked Lists (1 of 2)

- **Circular linked list:** last node points to first node

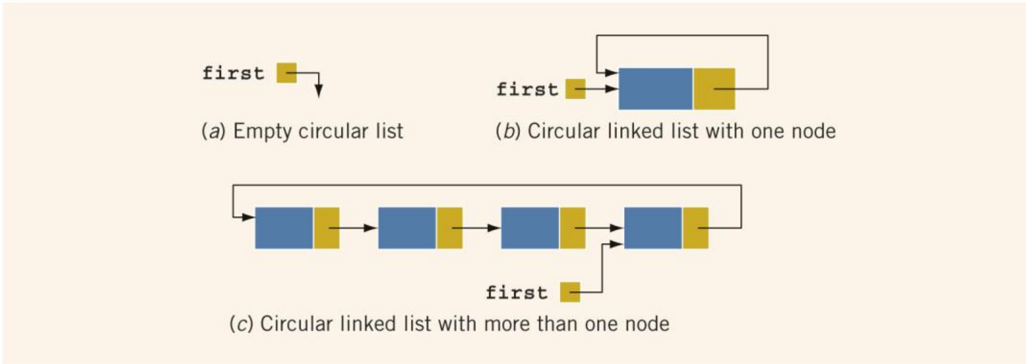


FIGURE 17-45 Circular linked lists

Circular Linked Lists (2 of 2)

- Operations on a circular list:
  - 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 or delete an item
  - Copy the list

Summary (1 of 3)

- A linked list is a list of items (nodes)
  - Order of the nodes is determined by the address (link) stored in each node
- 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 (2 of 3)

- Insertion and deletion do 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 the first node
- Traverse: use a pointer other than head

Summary (3 of 3)

- 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

Questions?

Additional Slides: Implementing Custom Doubly-Linked List (LList)

LList Class Overview

- **LList**: Custom implementation of a doubly-linked list, similar to `std::list`.
- **Characteristics**:
    - **O(1)** insertion and removal at any position.
    - Supports bidirectional traversal but lacks random access.
- **Components**:
    - **Node Structure**:
        - `T data`: Element of type `T`.
        - `Node* prev`, `Node* next`: Links to neighboring nodes.
    - **Instance Variables**:
        - `head`, `tail`, and `count` (size_type) for tracking nodes.

Node Struct

The `Node` structure is a member of `LList<T>`, supporting doubly-linked structure.

```
struct Node {
    T     data;  // Element of type T
    Node* prev;  // Previous node
    Node* next;  // Next node
};
```

- **Purpose**: Stores data and links to previous/next nodes.
- Enables efficient insertion and traversal in both directions.

BiDirectionalIterator Class

BiDirectionalIterator enables forward and backward traversal.

```
class BiDirectionalIterator {
public:
    using iterator_category = std::bidirectional_iterator_tag;
    using difference_type   = std::ptrdiff_t;
    using value_type        = T;
```

```cpp
    using pointer           = T*;
    using reference         = T&;

    BiDirectionalIterator(Node* ptr = nullptr);

    // Increment/decrement operators
    BiDirectionalIterator& operator++();
    BiDirectionalIterator operator++(int);
    BiDirectionalIterator& operator--();
    BiDirectionalIterator operator--(int);

    reference operator*() const;  // Dereference
    Node* operator->();           // Access Node
    bool operator==(const BiDirectionalIterator& rhs) const;
    bool operator!=(const BiDirectionalIterator& rhs) const;

private:
    Node* current;
};
```

Member Types in LList

Define these member types in **LList** for compatibility with C++ standards:

- **`value_type`**: Alias for `T` (element type).
- **`size_type`**: Alias for `std::size_t` (node count).
- **`reference`**: `value_type&`, allowing reference access.
- **`pointer`**: `value_type*`, allowing pointer access.
- **`iterator`**: Alias for `BiDirectionalIterator`, enabling bidirectional traversal.

Compatibility: Makes **LList** accessible to generic algorithms and functions.

Constructors and Destructor

LList provides the following constructors and a destructor:

1. **Default Constructor**: Initializes an empty list.
2. **Copy Constructor**: Creates a deep copy of an existing list.
3. **Move Constructor**: Transfers ownership from another list.
4. **Initializer List Constructor**: Constructs from `std::initializer_list<T>`.
5. **Destructor**: Clears all nodes to free memory.

Accessors and Capacity Functions

Functions for element access and list properties:

- **front()**: Returns a reference to the first element. Throws if empty.

- **back()**: Returns a reference to the last element. Throws if empty.
- **begin()**: Returns an iterator to the first element.
- **end()**: Returns an iterator past the last element.
- **empty() const**: Checks if the list has no elements.
- **size() const**: Returns the number of elements.

Basic Modifiers

Functions for adding and removing elements at the ends of the list:

- **clear() noexcept**: Clears all nodes, leaving the list empty.
- **push_back(const T& value)**: Adds `value` at the end of the list.
- **pop_back()**: Removes the last element. Throws if empty.
- **push_front(const T& value)**: Adds `value` at the beginning.
- **pop_front()**: Removes the first element. Throws if empty.

Advanced Modifiers: insert() and erase()

1. **insert(iterator pos, const T& value)**:
   - Creates a new `Node` with `data = value`.
   - Updates links for neighboring nodes.
   - Adjusts `count` and returns iterator to the new node.
2. **erase(iterator pos)**:
   - Removes node at `pos`, updating neighbor links.
   - Decrements `count` and returns iterator to the following node.

Comparison Operators

Friend template functions for comparisons:

1. **operator==()**:
   - Compares two lists for equality.
   - Returns true if sizes match and elements are equal.
2. **operator!=()**:
   - Returns true if lists differ in size or elements.

Why Friend Template Functions? - Accesses `LList`'s private members directly. - Defined as templates to work with any instantiation of `LList<T>`.

```
template <class U>
friend bool operator==(const LList<U>& lhs, const LList<U>& rhs);

template <class U>
friend bool operator!=(const LList<U>& lhs, const LList<U>& rhs);
```

Questions?