Chapter 17 - Linked Lists

Spring 2022

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
- Learn how to implement linked lists as ADTs

Objectives (2 of 2)

- Learn how to create linked list iterators
- Learn how to implement the basic operations on a linked list
- Learn how to create unordered linked lists
- Learn how to create ordered linked lists
- Learn how to construct a
- Become familiar with circular linked lists doubly linked list

Introduction

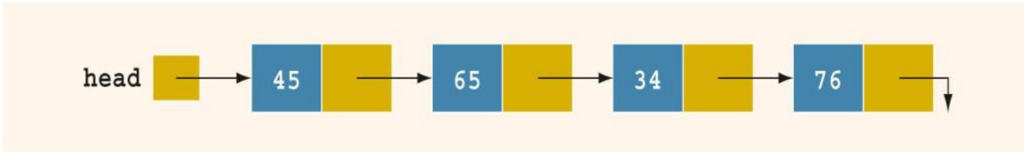
- Data can be organized and processed sequentially using an array, called a sequential list
- Problems with an array
  - Array size is fixed
  - **Unsorted array**: searching for an item is slow
  - **Sorted array**: insertion and deletion is slow because it requires data movement

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;
```

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

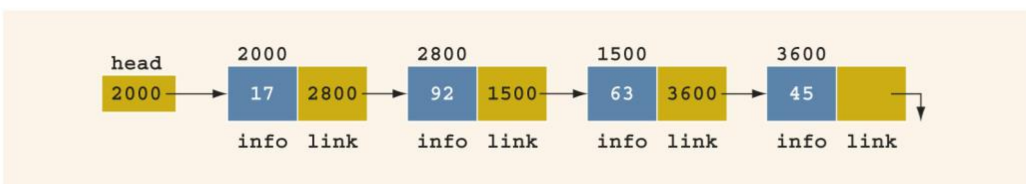


FIGURE 17-4 Linked list with four nodes

	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`

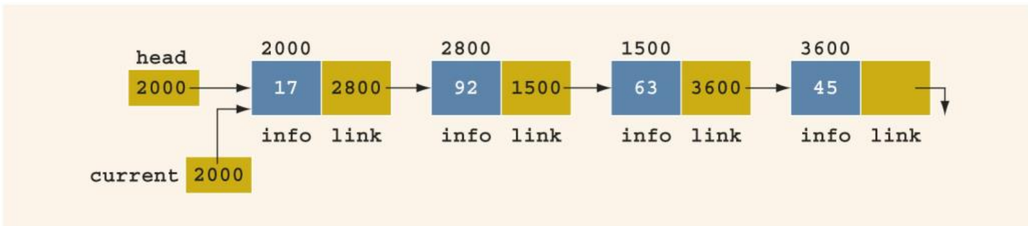


FIGURE 17-5 Linked list after the statement `current = head;` executes

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

Linked Lists: Some Properties (3 of 3)

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

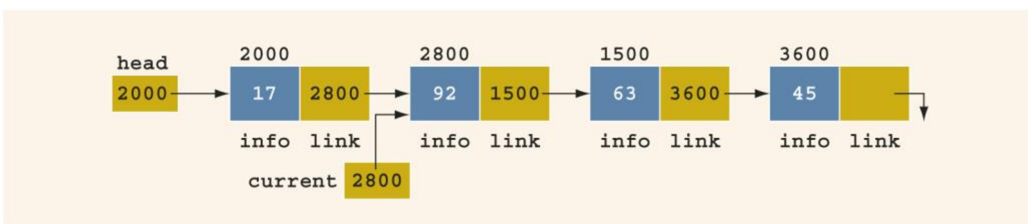


FIGURE 17-6 List after the statement `current = current->link;` executes

	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;
nodeType* tail{};
nodeType* p{};
nodeType* q{};
nodeType* newNode{};
```

Insertion (1 of 4)

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

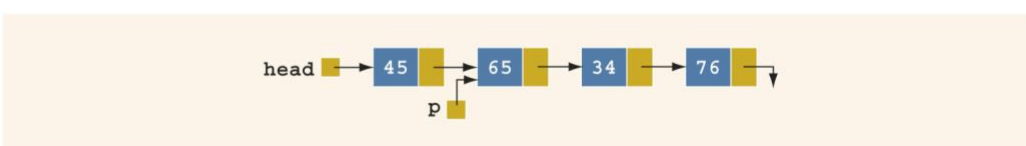
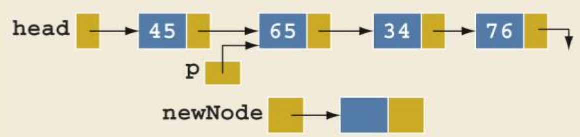
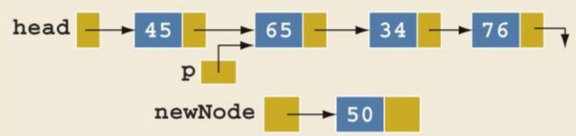
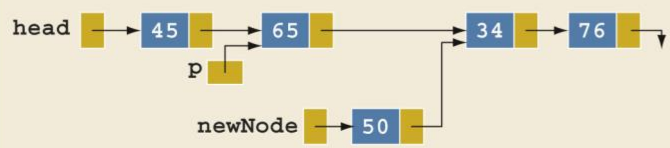
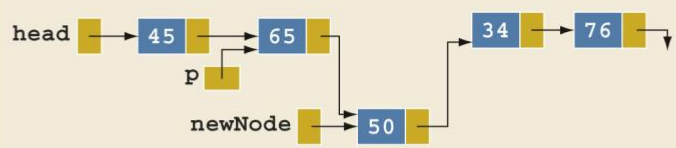


FIGURE 17-7 Linked list before item insertion

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

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)

- Can use two pointers to simplify the insertion code somewhat:

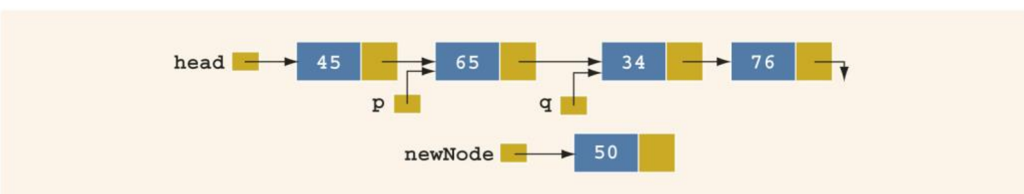


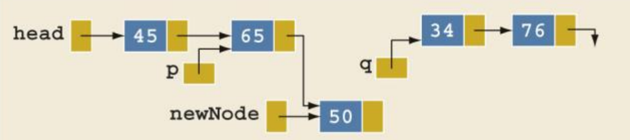
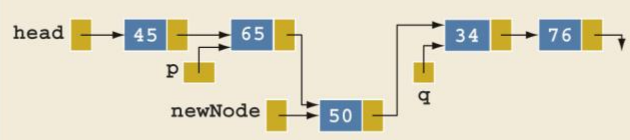
FIGURE 17-9 List with pointers p and q

- 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)

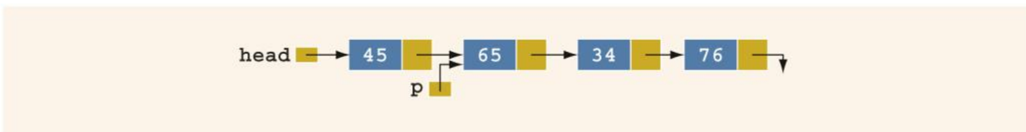


FIGURE 17-10 Node to be deleted is with `info 34`

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

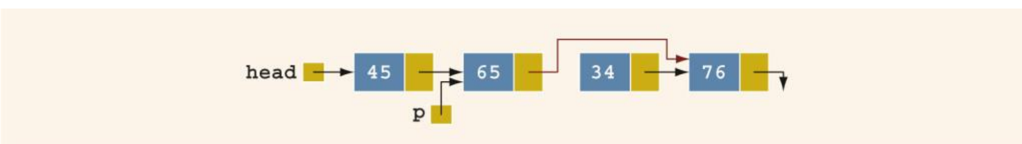


FIGURE 17-11 List after the statement `p->link = p->link->link` executes.

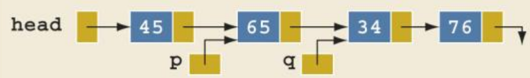
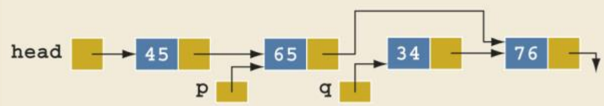
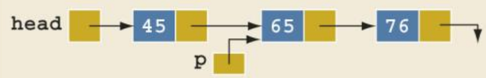
Deletion (2 of 3)

- 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 be able to deallocate its 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
- Can build a linked list forward or backward
  - **Forward**: a new node is always inserted at the end of the linked list
  - **Backward**: a new node is always inserted at the beginning of the list

Building a Linked List Forward (1 of 4)

- Need three pointers to build the list:
  - One to point to the first node in the list, which cannot be moved
  - One to point to the last node in the list
  - One to create the new node
- Example:
  - Data: 2 15 8 24 34

Building a Linked List Forward (2 of 4)

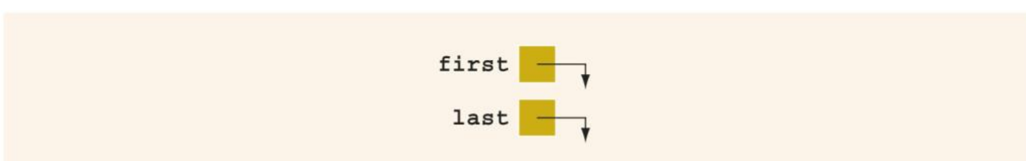


FIGURE 17-12 Empty list

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

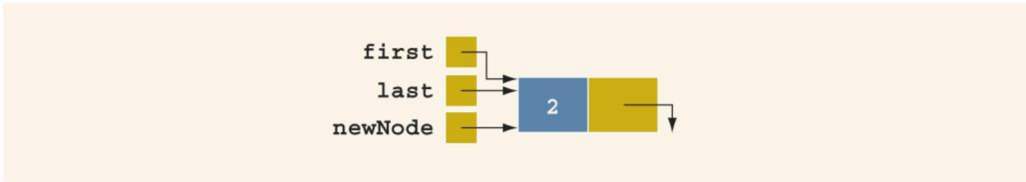


FIGURE 17-14 List after inserting **newNode** in it

Building a Linked List Forward (3 of 4)

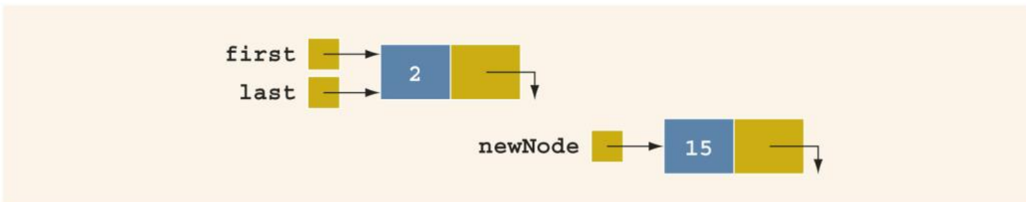


FIGURE 17-15 List and **newNode** with **info** 15

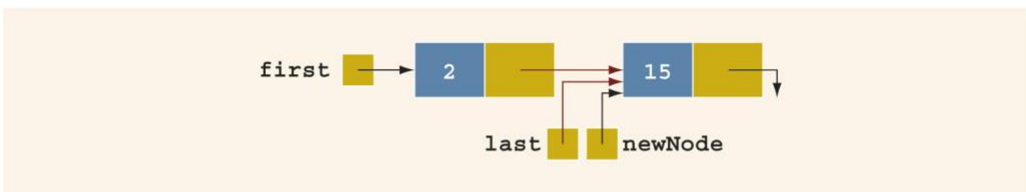


FIGURE 17-16 List after inserting **newNode** at the end

Building a Linked List Forward (4 of 4)

- Now repeat this process three more times:

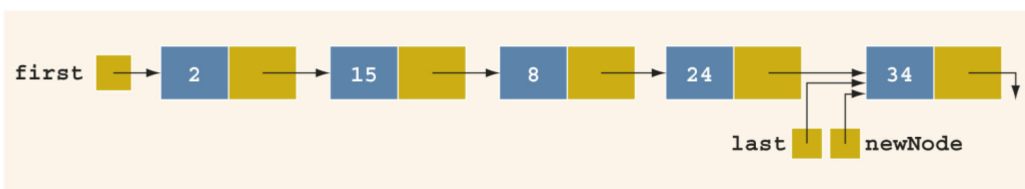


FIGURE 17-17 List after inserting 8, 24, and 34

Building a Linked List Backward

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

Linked List as an ADT (1 of 4)

- Basic operations on linked lists:

- Initialize the list
- Determine whether the list is empty
- Print the list
- Find the length of the list
- Destroy the list
- Retrieve `info` contained in the first or last node
- Search the list for a given item

Linked List as an ADT (2 of 4)

- Basic operations on linked lists (cont'd.):
  - Insert an item in the list
  - Delete an item from the list
  - Make a copy of the linked list

Linked List as an ADT (3 of 4)

- Two general types of linked lists:
  - Sorted and unsorted lists
- Algorithms to implement the operations search, insert, and remove differ slightly for sorted and unsorted lists
- abstract **classlinkedListType** will implement basic linked list operations
  - Derived classes: **unorderedLinkedList** and **orderedLinkedList**

Linked List as an ADT (4 of 4)

- For an unordered linked list, can insert a new item at either the end or the beginning
  - **buildListForward** inserts item at the end
  - **buildListBackward** inserts item at beginning
- Will need two functions:
  - **insertFirst** and **insertLast**
- Will use two pointers in the list:
  - **first** and **last**

Structure of Linked List Nodes

- Each node has two member variables
- We implement the node of a linked list as a struct
- Definition of the struct nodeType:

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

Member Variables of the class linkedListType

- **linkedListType** has three member variables:
  - Two pointers: **first** and **last**
  - count: the number of nodes in the list

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

Linked List Iterators (1 of 4)

- To process each node of the list
  - List must be traversed, starting at first node
- **Iterator**: object that produces each element of a container, one element at a time
  - The two most common iterator operations: ++ (the pre-increment operator) * (the dereferencing operator)

Linked List Iterators (2 of 4)

- An iterator is an object
  - Need to define a class (**linkedListIterator**) to create iterators to objects of the class **linkedListType**
  - Will have one member variable to refer to the current node

Linked List Iterators (3 of 4)

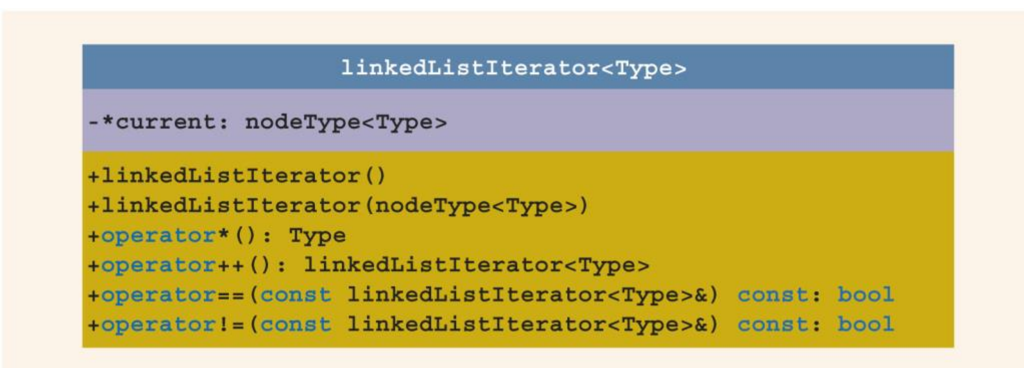


FIGURE 17-19 UML class diagram of the class linkedListIterator

Linked List Iterators (4 of 4)

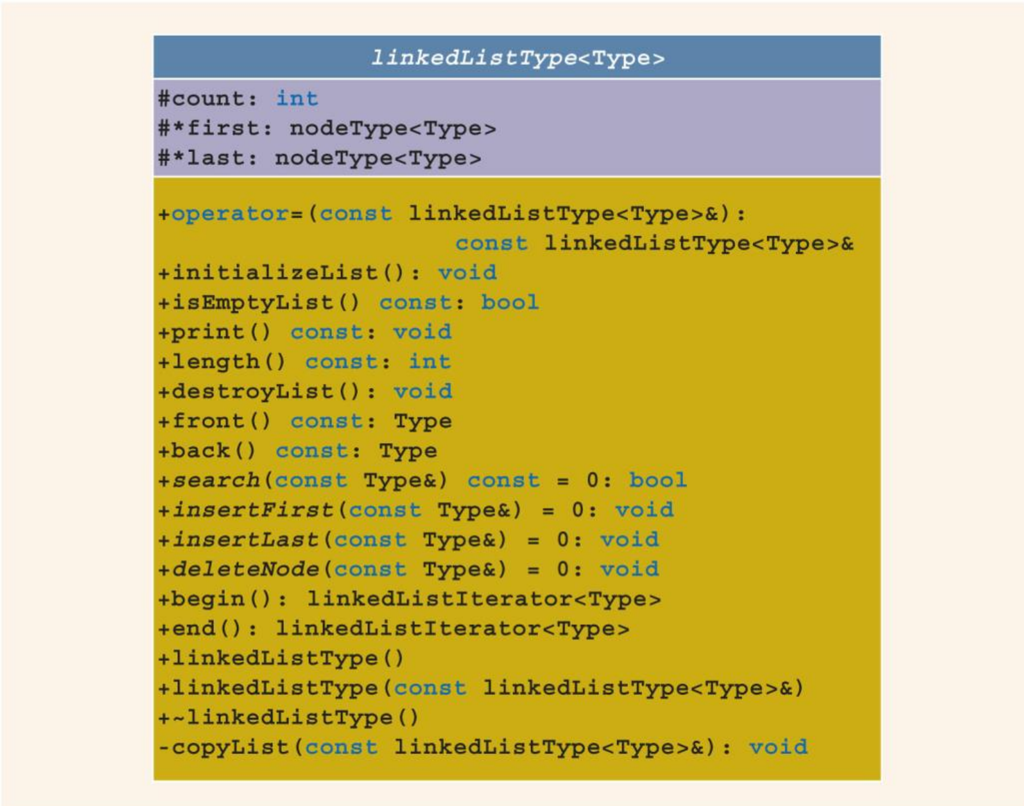


FIGURE 17-20 UML class diagram of the class linkedListType

Default Constructor

- Default constructor:
  - Initializes the list to an empty state

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

  - Or, using a member initialization list:

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

Destroy the List

- Function **destroyList**:
  - Traverses the list to deallocate memory occupied by each node
  - Once list is destroyed, sets pointers **first** and **last** to `nullptr` and count to 0

Initialize the List

- Function **initializeList**:
  - Initializes list to an empty state
- Since constructor already did this, **initializeList** is used to reinitialize an existing list

Print the List

- Function **print**:
  - Prints data contained in each node
  - Traverses the list using another pointer

Length of a List

- Function **length**:
  - Returns the count of nodes in the list
  - Uses the **count** variable

Retrieve the Data of the First or Last Node

- Function **front**:
  - Returns the info contained in the first node
  - If list is empty, program will be terminated
- Function **back**:
  - Returns the info contained in the last node
  - If list is empty, program will be terminated

Begin and End

- Function **begin**:
  - Returns an iterator to the first node in the list
- Function **end**:
  - Returns an iterator to one past the last node in the list

Copy the List

- Function **copyList**:
  - Makes an identical copy of a linked list
- Steps:
  - Create a node called **newNode**
  - Copy the **info** of the original node into **newNode**
  - Insert **newNode** at the end of the list being created

Destructor & Copy Constructor

- Destructor:

  - Deallocates memory occupied by nodes when the class object goes out of scope
  - Calls **destroyList** to traverse the list and delete each node
- Copy constructor:
  - Makes an identical copy of the linked list
  - Calls function **copyList**

Overloading the Assignment Operator

- Definition of the function to overload the assignment operator
  - Similar to the copy constructor

Unordered Linked Lists (1 of 2)

- class **unorderedLinkedList**
  - Derived from the abstract class **linkedListType**
  - Implements the operations **search**, **insertFirst**, **insertLast**, and **deleteNode**

Unordered Linked Lists (2 of 2)

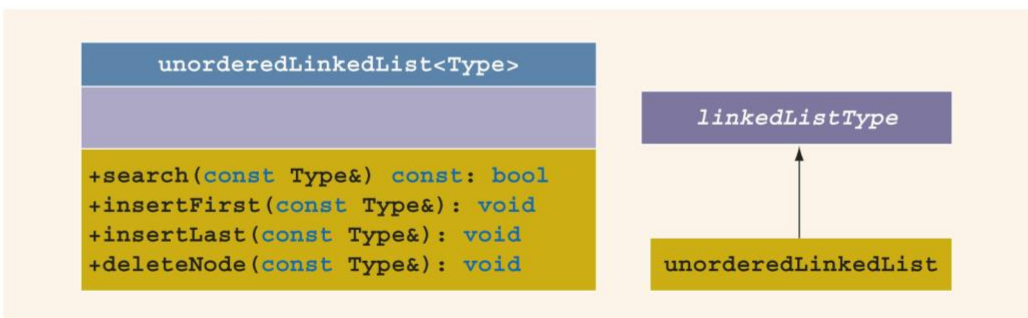


FIGURE 17-21 UML class diagram of the class unorderedLinkedList and inheritance hierarchy

Unordered Linked List: Search the List

- Function **search**:
  - Searches the list for a given item
- Steps:
  - Compare search item with current node in the list
    - If info of current node is the same as search item, stop the search
    - Otherwise, make the next node the current node
  - Repeat Step 1 until item is found or until no more data is left in the list

Insert the First Node

- Function **insertFirst**:
  - Inserts a new item at the beginning of the list
- Steps:
  - Create a new node
  - Store the new item in the new node

– Insert the node before **first**
– Increment **count** by 1

Insert the Last Node

- Function **insertLast**:
  - Inserts a new node after **last**
  - Similar to **insertFirst** function

Delete a Node (1 of 6)

- Function **deleteNode**:
  - Deletes a node with given info from the list
  - Several possible cases to manage
- **Case 1**: List is empty
  - If the list is empty, output an error message
- **Case 2**: Node to be deleted is the first node
  - Adjust the pointer **first** and count
  - If no other nodes, set **first** and **last** to `nullptr`

Delete a Node (2 of 6)

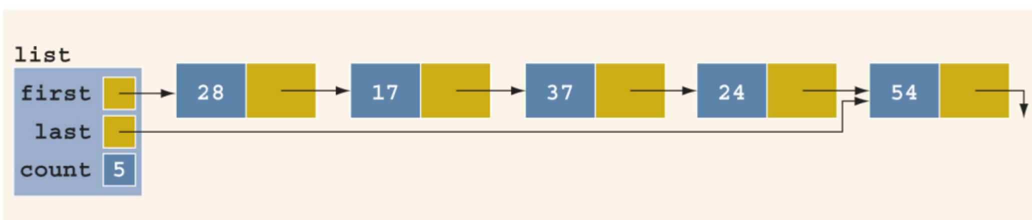


FIGURE 17-23 `list` with more than one node

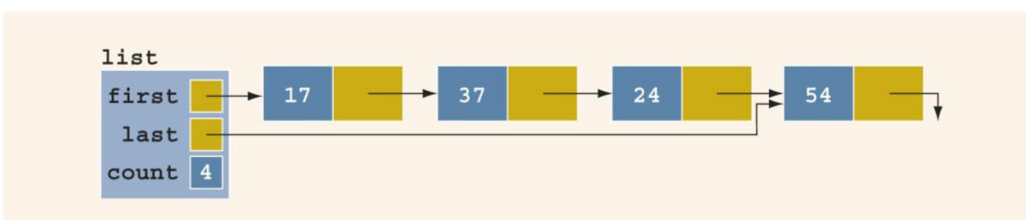


FIGURE 17-24 `list` after deleting node with `info` 28

Delete a Node (3 of 6)

- **Case 3**: Node to be deleted is not the first one – **Case 3a**: Node to be deleted is not last one –Update link field of the previous node – **Case 3b**: Node to be deleted is the last node – Update link field of the previous node to `nullptr` – Update `last` pointer to point to previous node

Delete a Node (4 of 6)

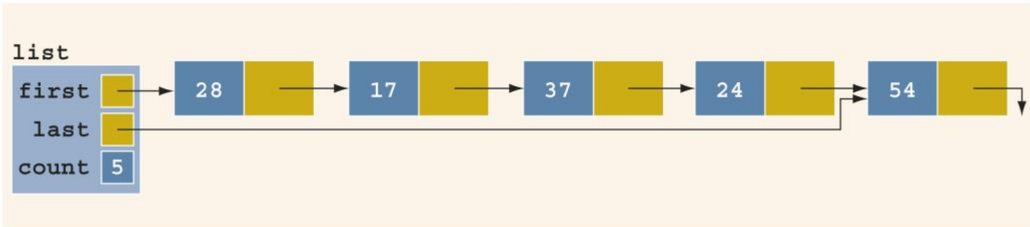


FIGURE 17-25 `list` before deleting 37

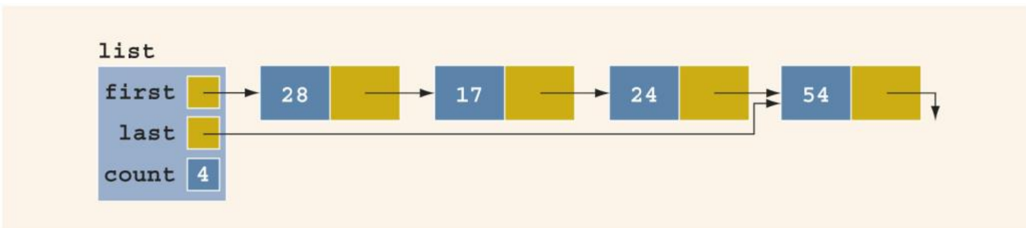


FIGURE 17-26 `list` after deleting 37

Delete a Node (5 of 6)

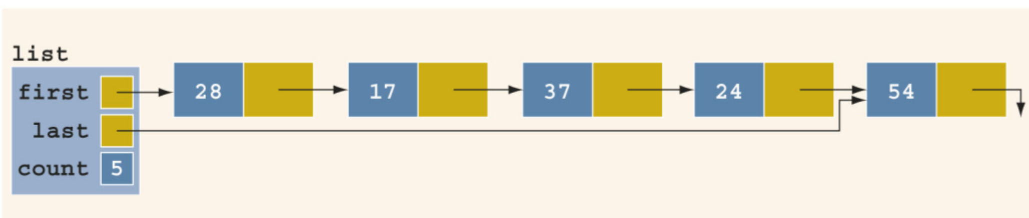


FIGURE 17-27 `list` before deleting 54

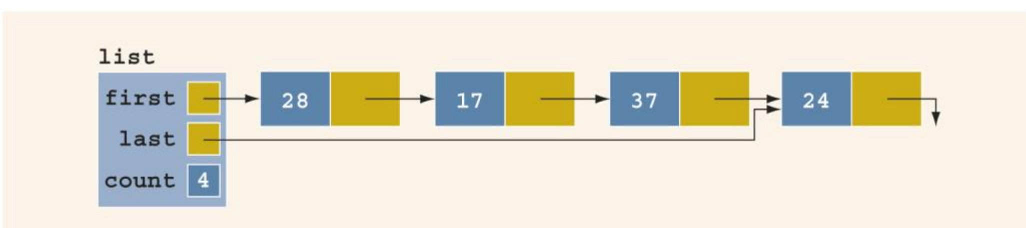


FIGURE 17-28 `list` after deleting 54

Delete a Node (6 of 6)

- **Case 4**: Node to be deleted is not in the list
  - List requires no adjustment
  - Simply fail silently, or output an error message

Ordered Linked Lists (1 of 2)

- **orderedLinkedList**: derived from class **linkedListType**
  - Provides definitions of the abstract functions **insertFirst**, **insertLast**, **search**, and **deleteNode**

- Assume that elements of an ordered linked list are arranged in ascending order
- Include the function **insert** to insert an element in an ordered list at its proper place

Ordered Linked Lists (2 of 2)

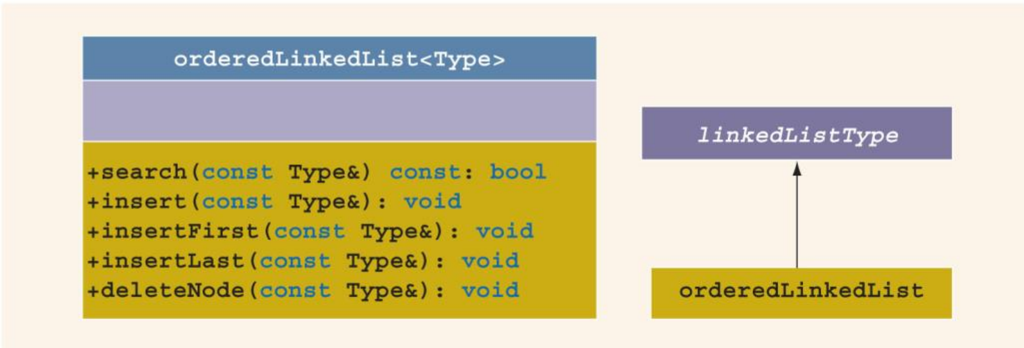


FIGURE 17-29 UML class diagram of the `class` `orderedLinkedList` and the inheritance hierarchy

Ordered Linked Lists: Search the List

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

Insert a Node (1 of 5)

- **Case 1**: The list is empty
  - New node becomes first node

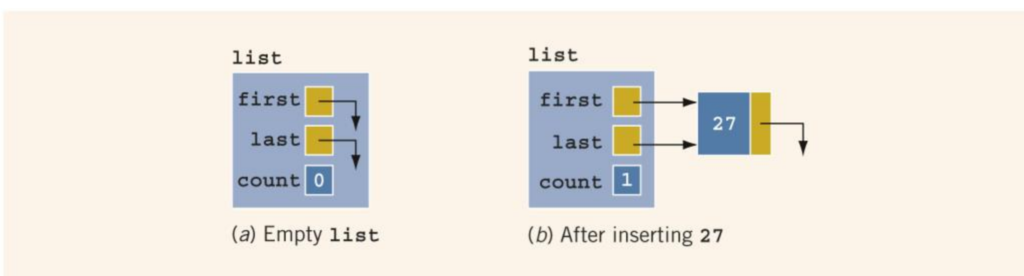


FIGURE 17-30 `list`

Insert a Node (2 of 5)

- **Case 2**: List is not empty, and the item to be inserted is smaller than smallest item in list
  - New node goes at beginning of list

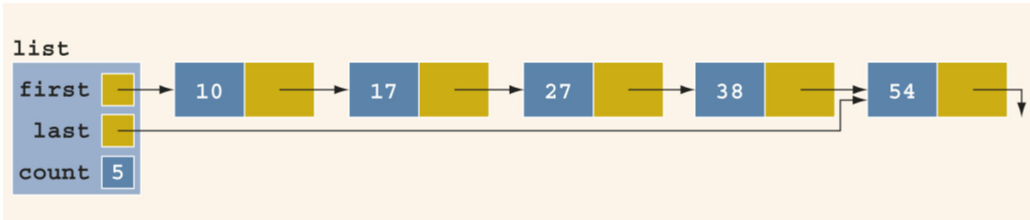


FIGURE 17-32 list after inserting 10

Insert a Node (3 of 5)

- **Case 3**: New item to be inserted somewhere in list
  - **Case 3a**: New item is larger than largest item
    - New item is inserted at end of list
  - **Case 3b**: Item to be inserted goes somewhere in the middle of the list

Insert a Node (4 of 5)

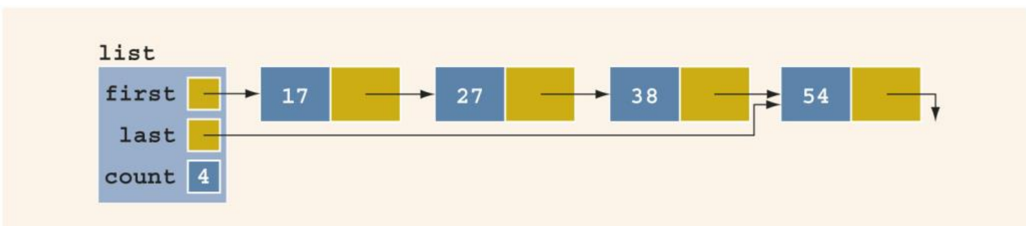


FIGURE 17-33 list before inserting 65

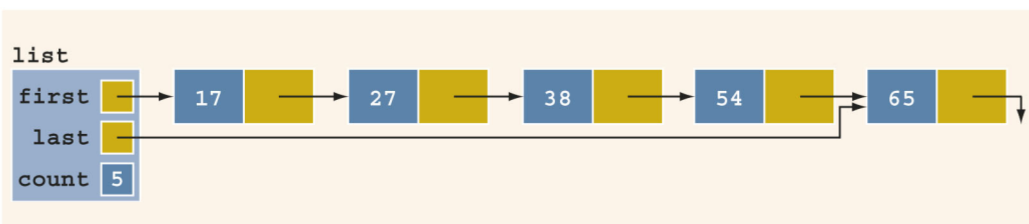


FIGURE 17-34 list after inserting 65

Insert a Node (5 of 5)

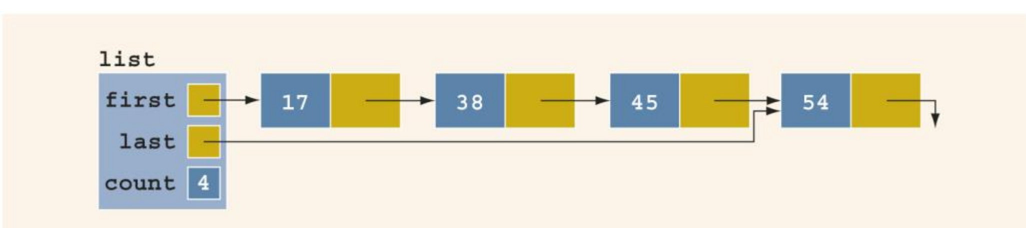


FIGURE 17-35 list before inserting 27

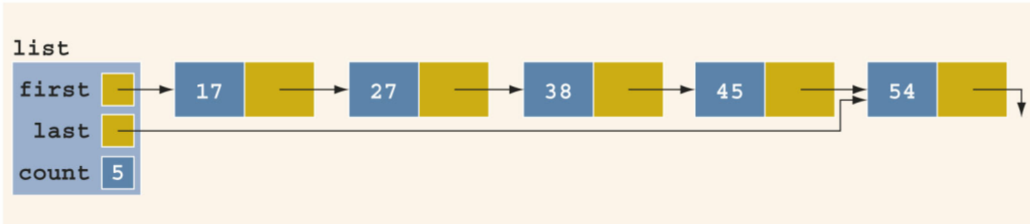


FIGURE 17-36 `list` after inserting `27`

Insert First and Insert Last

- Functions **insertFirst** and **insertLast**
  - Must insert new item at the proper place to ensure resulting list is still sorted
- These functions are not actually used
  - Definitions must be provided because they are declared as abstract in the parent class
- Function **insertNode** is used to insert at the proper place

Delete a Node

- **Case 1**: List is initially empty -> error
- **Case 2**: Item to be deleted is first node in list
  - Adjust the head (**first**) pointer
- **Case 3**: Item is somewhere in the list
  - **current** points to node with item to delete
  - **trailCurrent** points to node previous to the one pointed to by **current**
- **Case 4**: Item is not in the list -> error

Doubly Linked Lists (1 of 2)

- **Doubly linked list**: every node has next and back pointers
  - Can be traversed in either direction

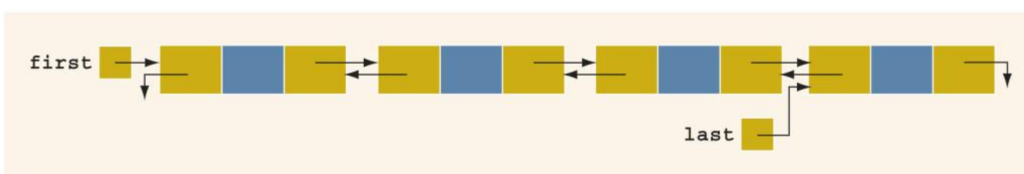


FIGURE 17-39 Doubly linked list

Doubly Linked Lists (2 of 2)

- Operations:
  - Initialize or destroy the list
  - Determine whether the list is empty
  - Search the list for a given item
  - Retrieve the first or last element of the list
  - Insert or delete an item

  - Find the length of the list
  - Print the list
  - Make a copy of the list

Doubly Linked List: Default Constructor

- Default constructor:
  - Initializes doubly-linked list to empty state
  - Sets **first** and **last** to `nullptr`, and `count` to 0
- **isEmptyList**:
  - Returns true if list is empty, false otherwise

Destroy the List & Initialize the List

- Function **destroy**:
  - Deletes all the nodes in the list, leaving list in an empty state
  - Sets `count` to 0
- Function **initializeList**:
  - Reinitializes doubly linked list to an empty state
  - Uses the **destroy** operation

Length of the List & Print the List

- Function **length**
  - Returns the count of nodes in the list
- Function **print**
  - Traverses the list
  - Outputs the `info` contained in each node
- Function **reversePrint**
  - Traverses list in reverse order using back links
  - Outputs the `info` in each node

Doubly Linked List: Search the List

- Function **search**:
  - Returns true if search item is found, otherwise false
  - Algorithm is same as that for an ordered linked list

First and Last Elements

- Function **front**
  - Returns first element of the list
- Function **back**
  - Returns last element of the list
- If list is empty, both functions will terminate the program

Insert a Node (1 of 2)

- Four insertion cases:

- **Case 1**: Insertion in an empty list
- **Case 2**: Insertion at beginning of a nonempty list
- **Case 3**: Insertion at end of a nonempty list
- **Case 4**: Insertion somewhere in nonempty list

- Cases 1 & 2 require update to pointer **first**
- Cases 3 & 4 are similar:
  - After inserting item, increment count by 1

Insert a Node (2 of 2)

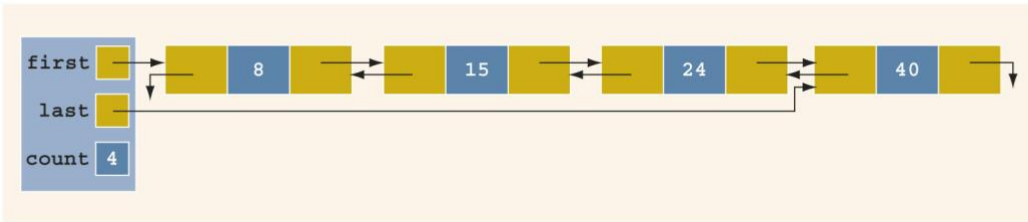


FIGURE 17-40 Doubly linked list before inserting 20

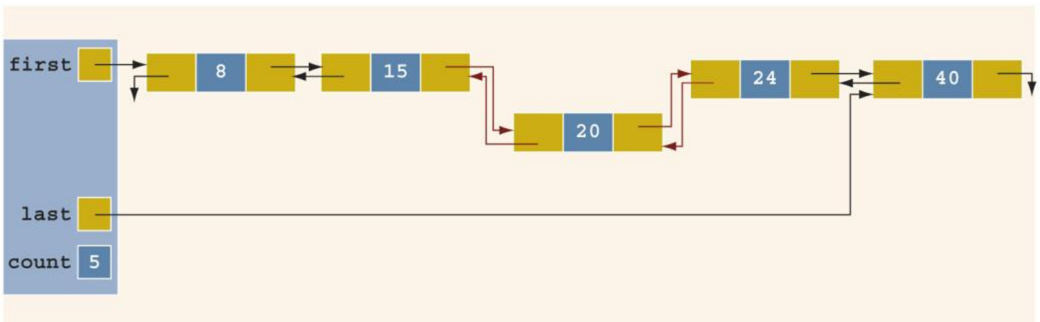


FIGURE 17-41 Doubly linked list after inserting 20

Delete a Node (1 of 3)

- **Case 1**: The list is empty
- **Case 2**: The item to be deleted is first node in list
  - Must update the pointer **first**
- **Case 3**: Item to be deleted is somewhere in the list
- **Case 4**: Item to be deleted is not in the list
- After deleting a node, **count** is decremented by **1**

Delete a Node (2 of 3)

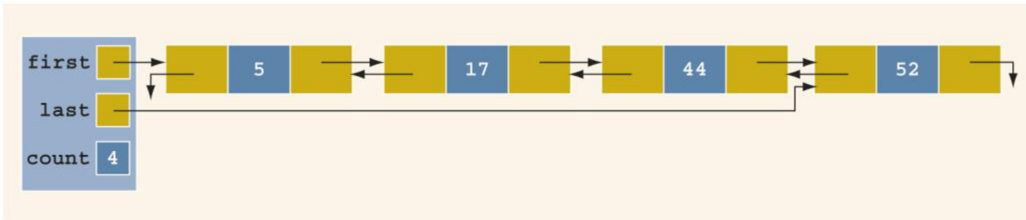


FIGURE 17-42 Doubly linked list before deleting 17

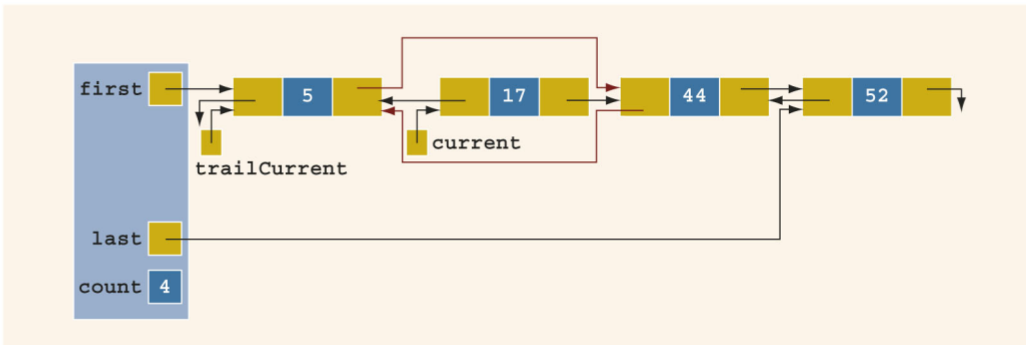


FIGURE 17-43 List after adjusting the links of the nodes before and after the node with info 17

Delete a Node (3 of 3)

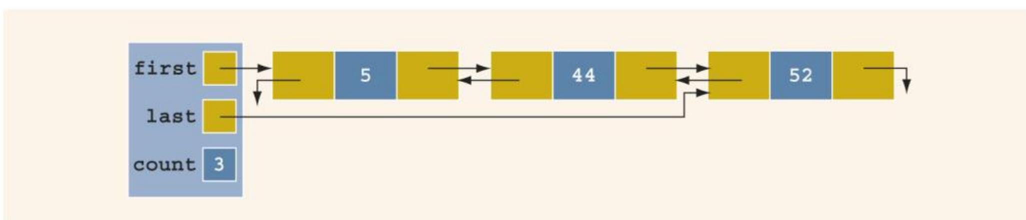


FIGURE 17-44 List after deleting the node with info 17

Circular Linked Lists (1 of 2)

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

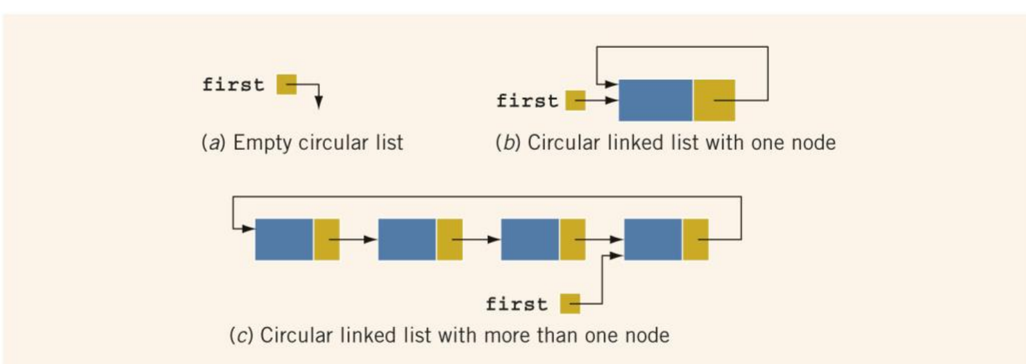


FIGURE 17-45 Circular linked lists

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

Summary (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?