Chapter 4: Linked Lists

Data Abstraction & Problem Solving with C++

Fifth Edition

by Frank M. Carrano

Preliminaries

- Options for implementing an ADT List
  - Array has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

Preliminaries

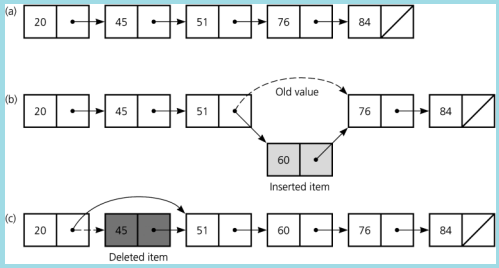

Figure 4-1 *(a)* A linked list of integers; (b) insertion; (c) deletion

Pointers

- A pointer contains the location, or address in memory, of a memory cell
  - Declaration of an integer pointer variable `p`

    ```
    int *p;
    ```
    - Initially undefined, but not *NULL*
    - Static allocation

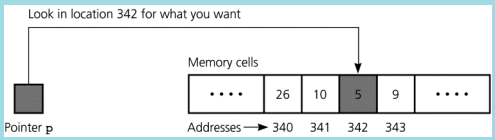

Figure 4-2 A pointer to an integer

Pointers

- The expression `*p` represents the memory cell to which `p` points
- To place the address of a variable into a pointer variable, you can use
  - The address-of operator `&`

    ```
    p = &x;
    ```
  - The `new` operator

    ```
    p = new int;
    ```
    - Dynamic allocation of a memory cell that can contain an integer
    - If the operator `new` cannot allocate memory, it throws the exception `std::bad_alloc` (in the `<new>` header)

Pointers

- The `delete` operator returns dynamically allocated memory to the system for reuse, and leaves the variable's contents undefined

  ```
  delete p;
  ```
  - A pointer to a deallocated memory (`*p`) cell is possible and dangerous

    ```
    p = NULL; // safeguard
    ```

Pointers

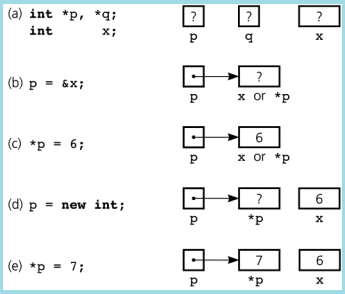

Figure 4-3 (a) Declaring pointer variables; (b) pointing to statically allocated memory; (c) assigning a value; (d) allocating memory dynamically; (e) assigning a value

Pointers

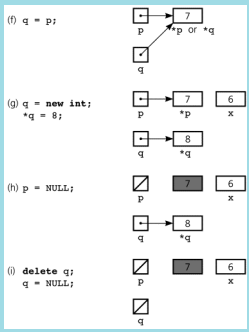

Figure 4-3 continued

(f) copying a pointer;

(g) allocating memory dynamically and assigning a value;

(h) assigning NULL to a pointer variable;

(i) deallocating memory

Dynamic Allocation of Arrays

- You can use the `new` operator to allocate an array dynamically

```
int arraySize = 50;
double *anArray = new double[arraySize ];
```

- An array name is a pointer to the array's first element
- The size of a dynamically allocated array can be increased

```
double *oldArray = anArray;
anArray = new double[2*arraySize];
```


Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

```
struct Node
{ int  item
  Node *next;
}; // end Node
```

Figure 4-6 A node

- The head pointer points to the first node in a linked list

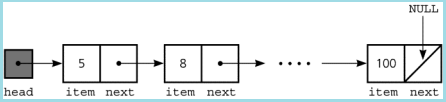

Figure 4-7 A head pointer to a list

Pointer-Based Linked Lists

- If head is *NULL*, the linked list is empty
- A node is dynamically allocated

```
Node *p;       // pointer to node
p = new Node; // allocate node
```


Pointer-Based Linked Lists

- Executing the statement

  `head = new Node`

  before

  `head = NULL`

  will result in a lost cell

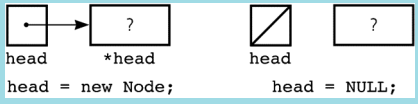

Figure 4-8 A lost cell

Displaying the Contents of a Linked List

- Reference a node member with the -> operator
  `p->item`
- A traverse operation visits each node in the linked list
  - A pointer variable `cur` keeps track of the current node

```
for (Node *cur = head; cur != NULL;
     cur = cur->next)
   cout << cur->item << endl;
```


Displaying the Contents of a Linked List

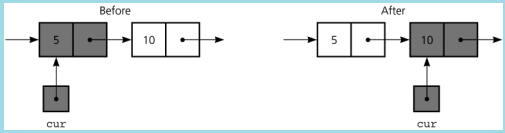

Figure 4-9

The effect of the assignment ***`cur = cur->next`***

Deleting a Specified Node from a Linked List

- Deleting an interior node

```
prev->next = cur->next;
```

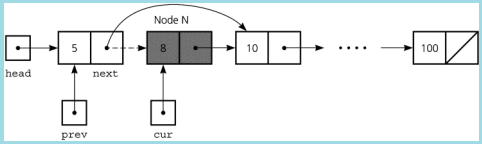

Figure 4-10 Deleting a node from a linked list

Deleting a Specified Node from a Linked List

- Deleting the first node

```
head = head->next;
```

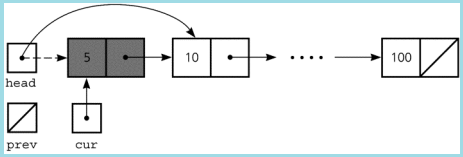

Figure 4-11 Deleting the first node

Deleting a Specified Node from a Linked List

Return deleted node to system

```
cur->next = NULL;
delete cur;
cur = NULL;
```


Inserting a Node into a Specified Position of a Linked List

- To insert a node between two nodes

```
newPtr->next = cur;
prev->next = newPtr;
```

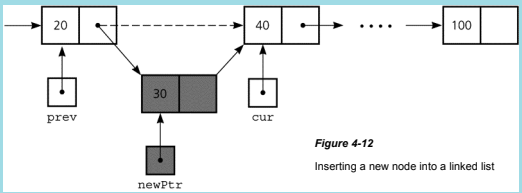

Figure 4-12

Inserting a new node into a linked list

Inserting a Node into a Specified Position of a Linked List

- To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```

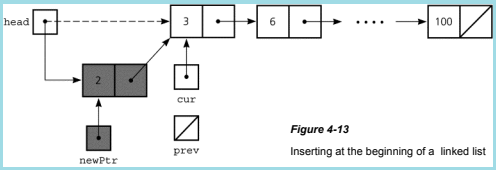

Figure 4-13
Inserting at the beginning of a linked list

Inserting a Node into a Specified Position of a Linked List

- Inserting at the end of a linked list is not a special case if `cur` is *NULL*

```
newPtr->next = cur;
prev->next = newPtr;
```

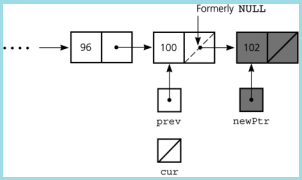

Figure 4-14
Inserting at the end of a linked list

Inserting a Node into a Specified Position of a Linked List

- Finding the point of insertion or deletion for a sorted linked list of objects

```
Node *prev, *cur;
for (prev = NULL, cur = head;
    (cur != NULL) && (newValue > cur->item);
     prev = cur, cur = cur->next);
```


A Pointer-Based Implementation of the ADT List

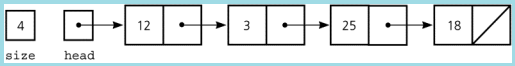

Figure 4-17 A pointer-based implementation of a linked list

A Pointer-Based Implementation of the ADT List

- Public methods
  - `isEmpty`
  - `getLength`
  - `insert`
  - `remove`
  - `retrieve`
- Private method
  - `find`
- Private data members
  - `head`
  - `size`
- Local variables to methods
  - `cur`
  - `prev`

Constructors and Destructors

- Default constructor initializes `size` and `head`
- A destructor is required for dynamically allocated memory

```
List::~List()
{
   while (!isEmpty())
      remove(1);
}  // end destructor
```

Constructors and Destructors

- Copy constructor creates a deep copy
  - Copies `size`, `head`, and the linked list
  - The copy of `head` points to the copied linked list
- In contrast, a shallow copy
  - Copies `size` and `head`
  - The copy of `head` points to the original linked list
- If you omit a copy constructor, the compiler generates one
  - But it is only sufficient for implementations that use statically allocated arrays

Shallow Copy vs. Deep Copy

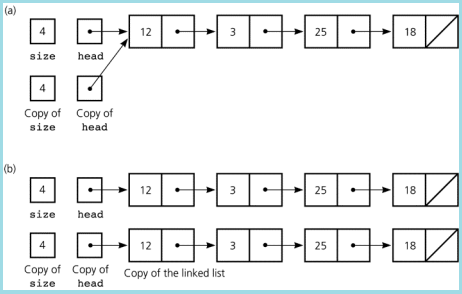

Figure 4-18 Copies of the linked list in Figure 4-17; (a) a shallow copy; (b) a deep copy

Comparing Array-Based and Pointer-Based Implementations

- Size
  - Increasing the size of a resizable array can waste storage and time
  - Linked list grows and shrinks as necessary
- Storage requirements
  - Array-based implementation requires less memory than a pointer-based one for each item in the ADT

Comparing Array-Based and Pointer-Based Implementations

- Retrieval
  - The time to access the i^{th} item
    - Array-based: Constant (independent of i)
    - Pointer-based: Depends on i
- Insertion and deletion
  - Array-based: Requires shifting of data
  - Pointer-based: Requires a traversal

Saving and Restoring a Linked List by Using a File

- Use an external file to preserve the list between runs of a program
- Write only data to a file, not pointers
- Recreate the list from the file by placing each item at the end of the linked list
  - Use a tail pointer to facilitate adding nodes to the end of the linked list
  - Treat the first insertion as a special case by setting the tail to head

Passing a Linked List to a Method

- A method with access to a linked list's `head` pointer has access to the entire list
- Pass the head pointer to a method as a reference argument
  - Enables method to change value of the head pointer itself (value argument would not)

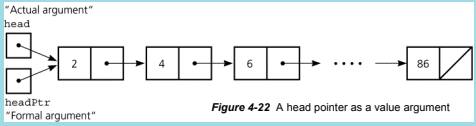

Figure 4-22 A head pointer as a value argument

Processing Linked Lists Recursively

- Recursive strategy to display a list
  - Write the first item in the list
  - Write the rest of the list (a smaller problem)
- Recursive strategies to display a list backward
  - First strategy
    - Write the last item in the list
    - Write the list minus its last item backward

Processing Linked Lists Recursively

  - Second strategy
    - Write the list minus its first item backward
    - Write the first item in the list
- Recursive view of a sorted linked list
  - The linked list to which `head` points is a sorted list if
    - `head` is *NULL or*
    - `head->next` is *NULL or*
    - `head->item < head->next->item`, and `head->next` points to a sorted linked list

Objects as Linked List Data

- Data in a node of a linked list can be an instance of a class

```
typedef ClassName ItemType;
struct Node
{   ItemType item;
    Node *next;
}; //end struct
Node *head;
```


Variations: Circular Linked Lists

- Last node points to the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

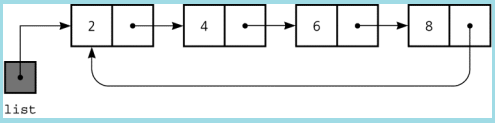

Figure 4-25 A circular linked list

Variations: Circular Linked Lists

- Access to last node requires a traversal
- Make external pointer point to last node instead of first node
  - Can access both first and last nodes without a traversal

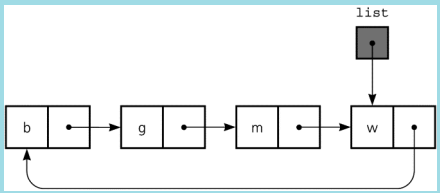

Figure 4-26 A circular linked list with an external pointer to the last node

Variations: Dummy Head Nodes

- Dummy head node
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to point to the dummy head node, rather than to *NULL*
    - Eliminates special case

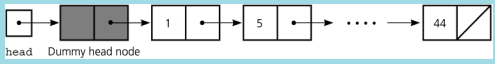

Figure 4-27 A dummy head node

Variations: Doubly Linked Lists

- Each node points to both its predecessor and its successor
  - `precede` pointer and `next` pointer
  - Insertions/deletions more involved than for a singly linked list
  - Often has a dummy head node
  - Often circular to eliminate special cases

Variations: Doubly Linked Lists

- Circular doubly linked list with dummy head node
  - `precede` pointer of the dummy head node points to the last node
  - `next` pointer of the last node points to the dummy head node
  - No special cases for insertions and deletions

Variations: Doubly Linked Lists

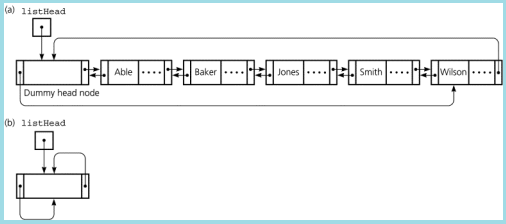

Figure 4-29 (a) A circular doubly linked list with a dummy head node
(b) An empty list with a dummy head node

Variations: Doubly Linked Lists

- To delete the node to which `cur` points

```
(cur->precede)->next = cur->next;
(cur->next)->precede = cur->precede;
```

- To insert a new node pointed to by `newPtr` before the node pointed to by `cur`

```
newPtr->next = cur;
newPtr->precede = cur->precede;
cur->precede = newPtr;
newPtr->precede->next = newPtr;
```


Application: Maintaining an Inventory

- Operations on the inventory
  - List the inventory in alphabetical order by title (L command)
  - Find the inventory item associated with title (I, M, D, O, and S commands)
  - Replace the inventory item associated with a title (M, D, R, and S commands)
  - Insert new inventory items (A and D commands)

The C++ Standard Template Library

- The STL contains class templates for some common ADTs, including the *`list`* class
- The STL provides support for predefined ADTs through three basic items
  - Containers
    - Objects that hold other objects
  - Algorithms
    - That act on containers
  - Iterators
    - Provide a way to cycle through the contents of a container

Summary

- The C++ `new` and `delete` operators enable memory to be dynamically allocated and recycled
- Each pointer in a linked list is a pointer to the next node in the list
- Algorithms for insertions and deletions in a linked list involve traversing the list and performing pointer changes
  - Use the operator `new` to allocate a new node and the operator `delete` to deallocate a node

Summary

- Special cases
  - Inserting a node at the beginning of a linked list
  - Deleting the first node of a linked list
- Array-based lists use an implicit ordering scheme; pointer-based lists use an explicit ordering scheme
  - Pointer-based requires memory to represent pointers
- Arrays enable direct access of an element; Linked lists require a traversal

Summary

- Inserting an item into a linked list does not shift data, an important advantage over array-based implementations
- A class that allocates memory dynamically needs an explicit copy constructor and destructor
- If you omit a copy constructor or destructor, the compiler generates one
  - But such methods are only sufficient for implementations that use statically allocated arrays

Summary

- You can increase the size of a linked list one node at a time more efficiently that you can increase the size of an array by one location
  - Increasing the size of an array involves copying
- A binary search of a linked list is impractical, because you cannot quickly locate its middle item
- You can save the data in a linked list in a file, and later restore the list

Summary

- Recursion can be used to perform operations on a linked list
  - Eliminates special cases and trailing pointer
- Recursive insertion into a sorted linked list considers smaller and smaller sorted lists until the actual insertion occurs at the beginning of one of them

Summary

- In a circular linked list, the last node points to the first node
  - The external pointer points to the last node
- A dummy head node eliminates the special cases for insertion into and deletion from the beginning of a linked list

Summary

- In a doubly linked list, each node points to both its successor and predecessor
  - Enables traversal in two directions
  - Insertions/deletions are more involved than with a singly linked list
    - Both a dummy head node and a circular organization eliminate special cases

Summary

- A class template enables you to defer choosing some data-types within a class until you use it
- The Standard Template Library (STL) contains class templates for some common ADTs
- A container is an object that holds other objects
- An iterator cycles through the contents of a container