Lecture 06

List Plus

C++ Templates

- **Generic Data Type:** A type for which only the operators are defined; the ItemType used by our ADTs is generic
- **Template:** A C++ feature by which the compiler generates multiple versions of a class by using parameterized types
- Essentially allows "blanks" in the class definition that clients can fill in to customize the class
- **`template<class ItemType>`**

Example: Stack ADT Template

```
template<class ItemType>
class StackType
{
public:
    StackType();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType item);
    void Pop();
    ItemType Top() const;
private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

Example: Stack ADT Template (cont.)

Clients use the template like so:

```
StackType<int> intStack;
StackType<float> floatStack;
StackType<char> charStack;
```

The compiler will generate a specialized version of StackType for each variable.

Function Templates

- Functions are turned into templates using the template keyword, just like classes
- A template class's methods must also be function templates if they use the type variable in the template
- For example, Push has an ItemType parameter and must be made into a function template

Example: Push Template

```
template<class ItemType>
void StackType<ItemType>::Push(ItemType
newItem)
{
    if (IsFull())
        throw FullStack();
    top++;
    items<top> = newItem;
}
```

Source Files and Templates

- Usually, the class header (StackType.h) and member function definitions (StackType.cpp) are two separate files
- This way, the class's object code can be compiled independently of the client code
- But with templates, the compiler must know the actual type parameter for the template, which appears in the client code

Source Files and Templates (cont.)

- The general solution is to compile the client code and the class's methods at once
  - Write class definition and method implementations in one file
  - Or use separate files, but #include the implementation at the end of the header
- When the client #includes the header, the compiler will have access to all the code it needs

Circular Linked Lists

- A linked list in which every node has a successor
- The “last” element is succeeded by the “first” element
- The class definition doesn’t change, but traversing the list is a little more complex

Circular Linked List (cont.)

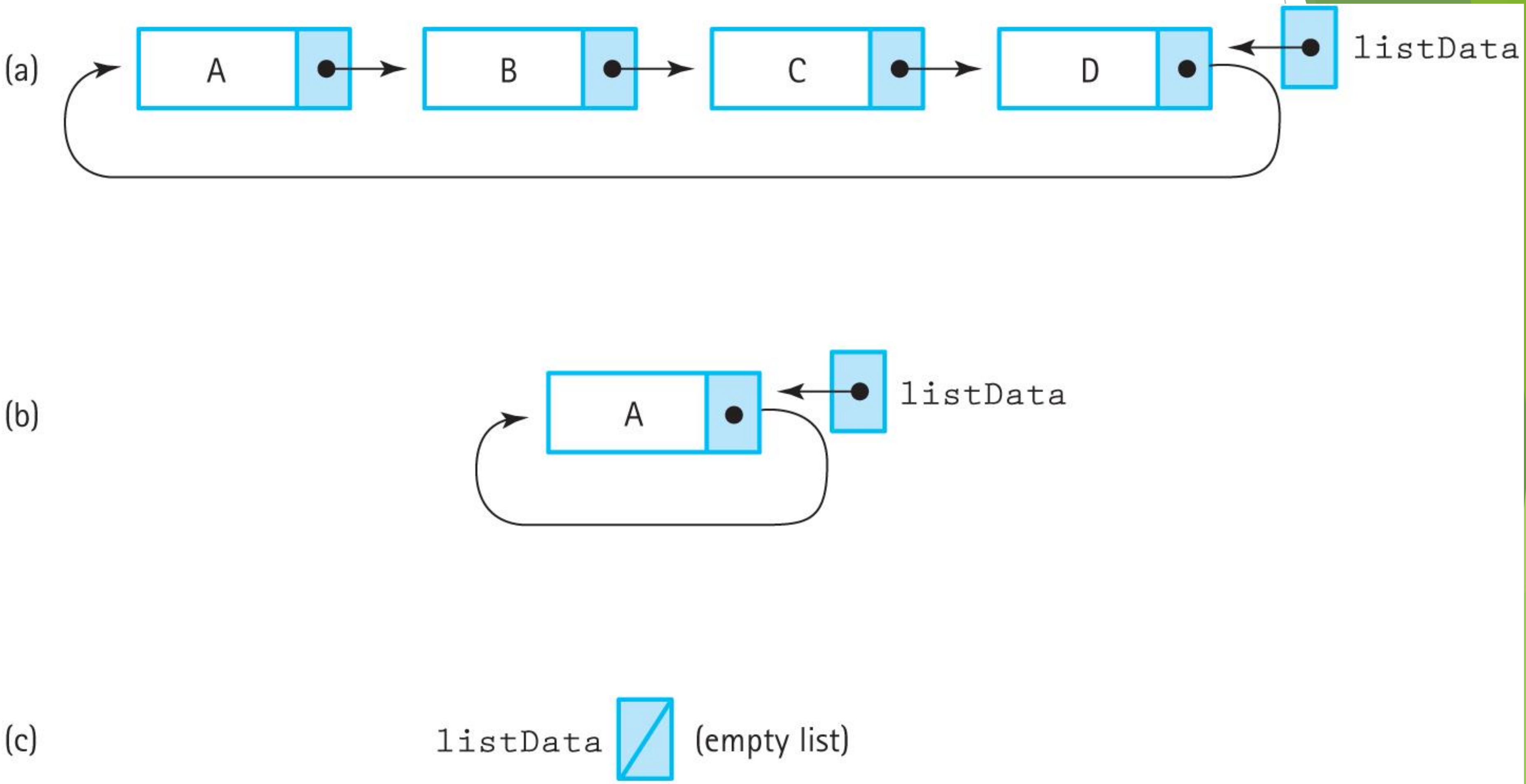


Figure 6.2 Circular linked lists with the external pointer pointing to the rear element

Circular Linked List: Finding Items

- GetItem, PutItem, and DeleteItem all search the list, create a helper function called **FindItem**
- Search stops when:
  - A key greater than or equal to the target item's key is found
  - It encounters the first item in the list again (this is the "end" of the list)
- Returns location, previous location, and a flag
  - If the flag is true, location points to the found item; if false, it points to the item's successor

Circular Linked List: Finding Items

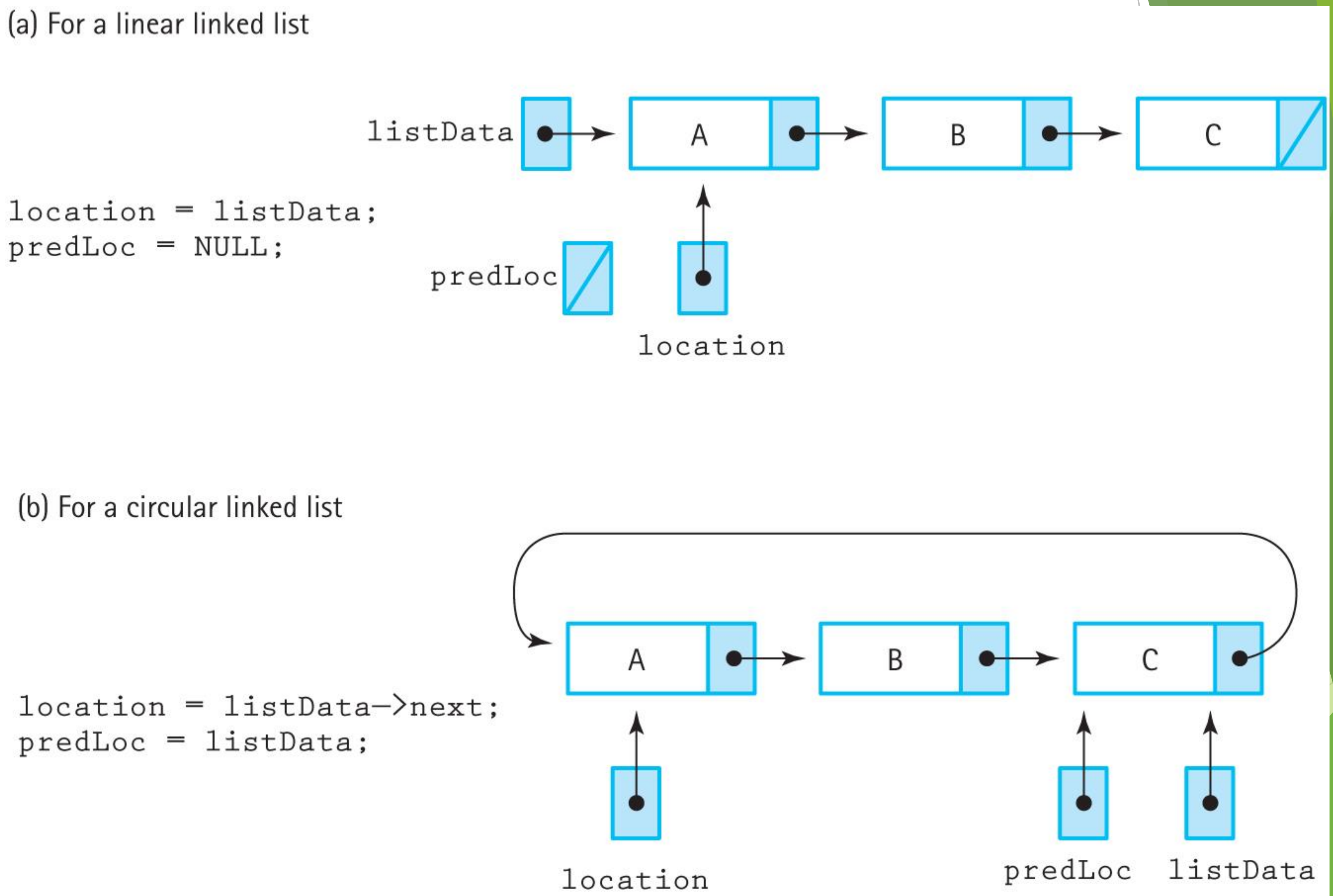
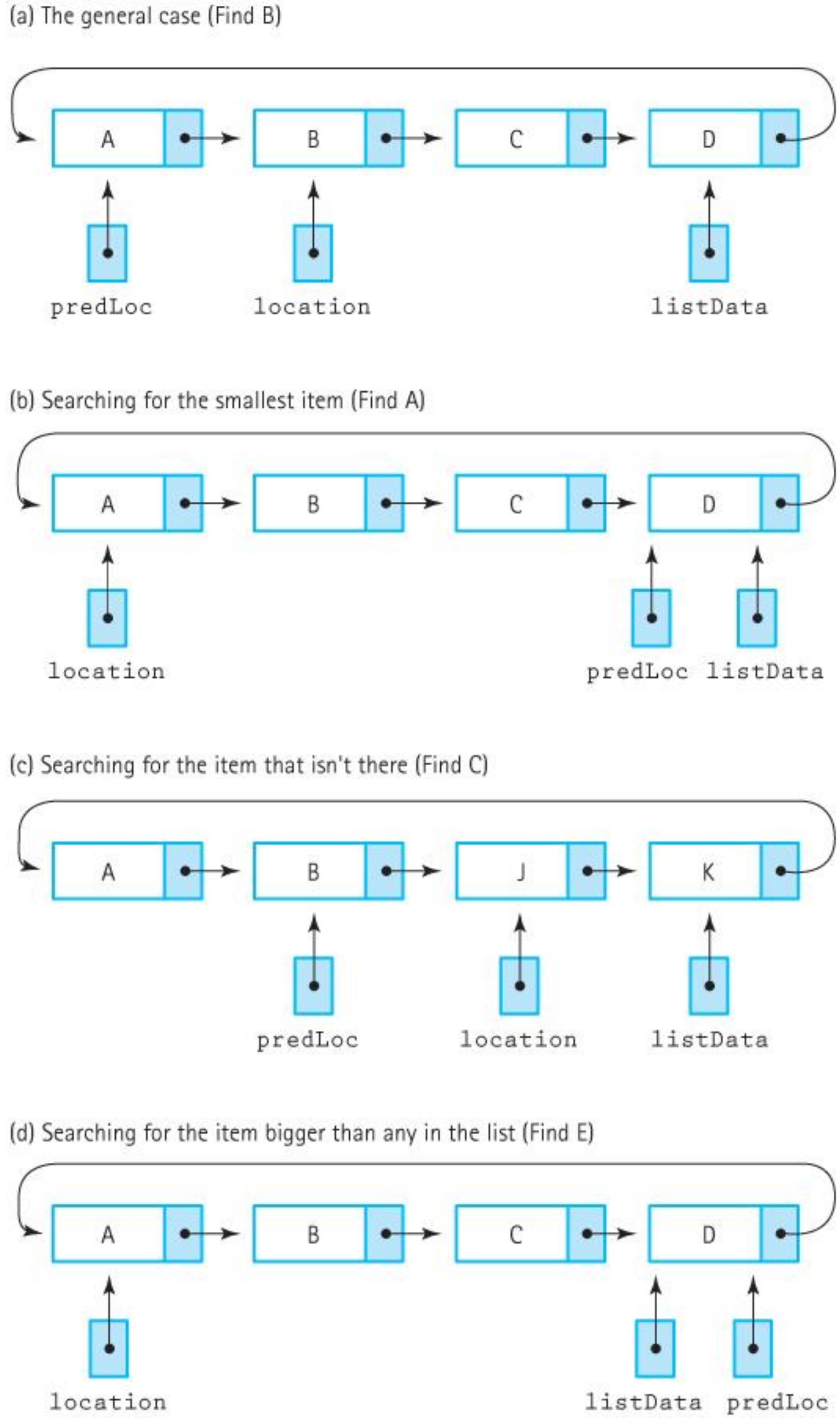


Figure 6.3 Circular linked lists with the external pointer pointing to the rear element

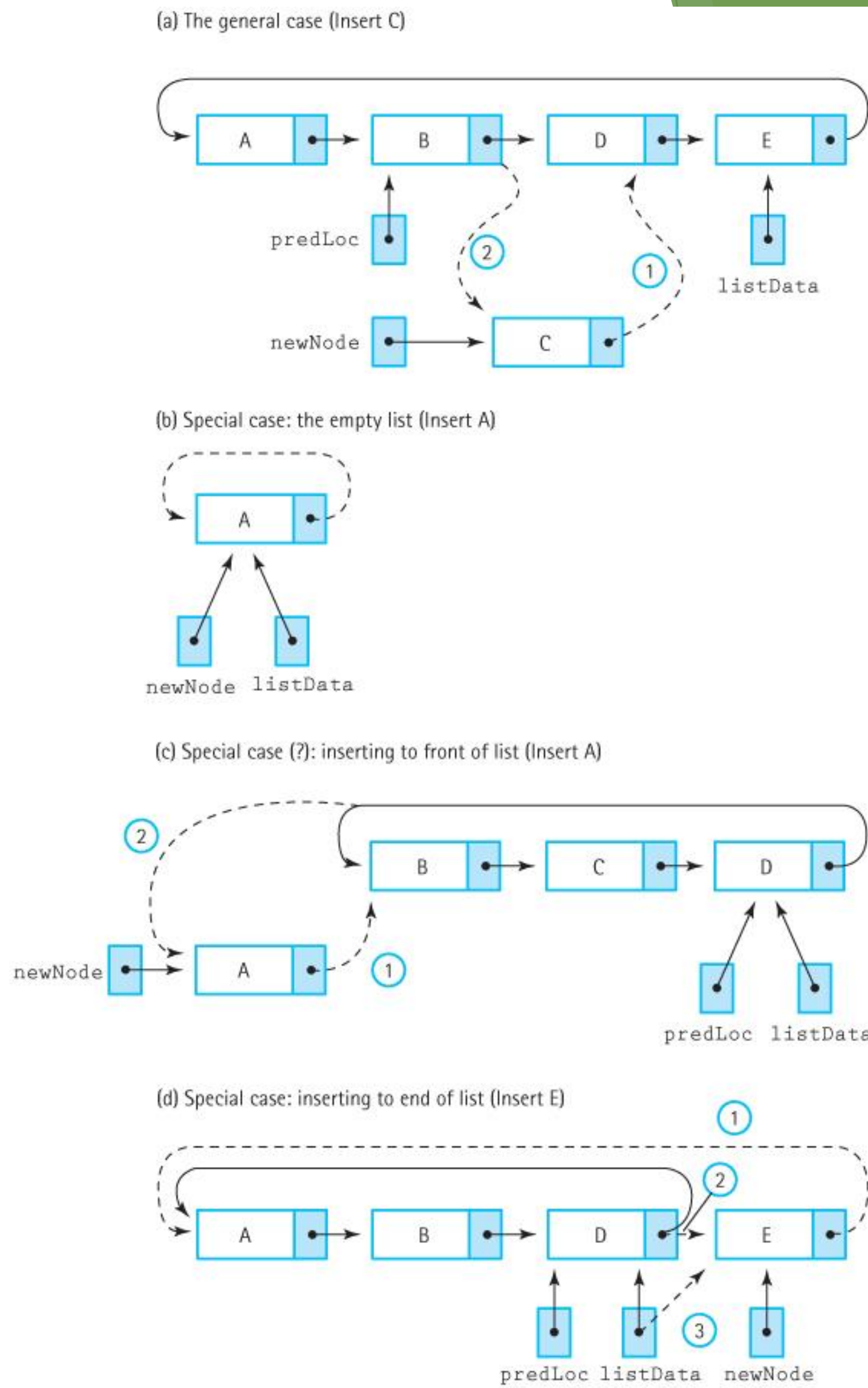
Figure 6.4 The FindItem operation for a circular list (a) The general case (Find B); (b) Searching for the smallest item (Find A); (c) Searching for the item that isn't there (Find C); (d) Searching for the item bigger than any in the list (Find E)



Circular Linked List: Inserting Items

- General case: Link predecessor to new node and new node to successor
- Inserting into an empty list: The new node points to itself
- Inserting into the front of a list: Only special in regular linked lists
- Inserting at the end of a list: Update the external pointer

Figure 6.5 Inserting into a circular linked list (a) The general case (Insert C); (b) Special case: the empty list (Insert A); (c) Special case (?): inserting to front of list (Insert A); (d) Special case: inserting to end of list (Insert E)



Circular Linked List: Deleting Items

- General case: Update predecessor to point to deleted item’s successor, then delete item
- Deleting only item in list: Set external pointer to NULL
- Deleting item at the end of a list: Update external pointer to point to deleted node’s predecessor

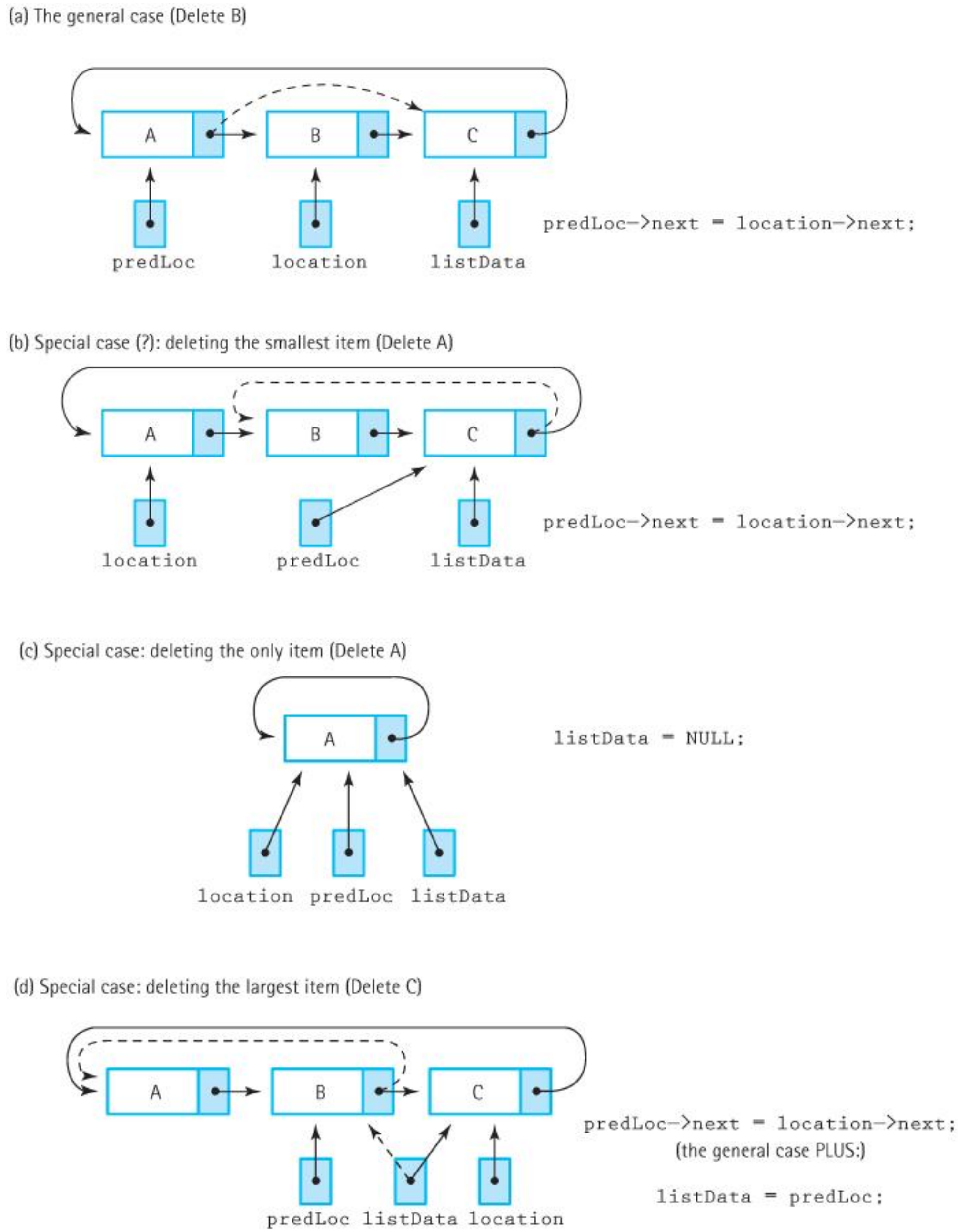


Figure 6.6 Deleting from a circular linked list (a) The general case (Delete B) (b) Special case (?): deleting the smallest item (Delete A) (c) Special case: deleting the only item (Delete A) (d) Special case: deleting the largest

Doubly Linked List

- A linked list in which every node has 2 pointers, linking it to its successor and predecessor
  - First node has NULL predecessor pointer
  - Last node has NULL successor pointer
- Can walk forward or backward through the list

Figure 6.7 A linear doubly linked list

Doubly Linked List: Finding Items

- “Inchworm” search no longer needed, since the previous element can be accessed directly
- FindItem only needs to return the pointer to the item or the item’s successor

Doubly Linked List Operations

- Insertion and deletion are slightly more complex due to the additional pointers
- Both the predecessor and the successor of the target node must have their pointers updated
- Operating on items at either end of the list is similar to singly linked lists

Doubly Linked List Operations (cont.)

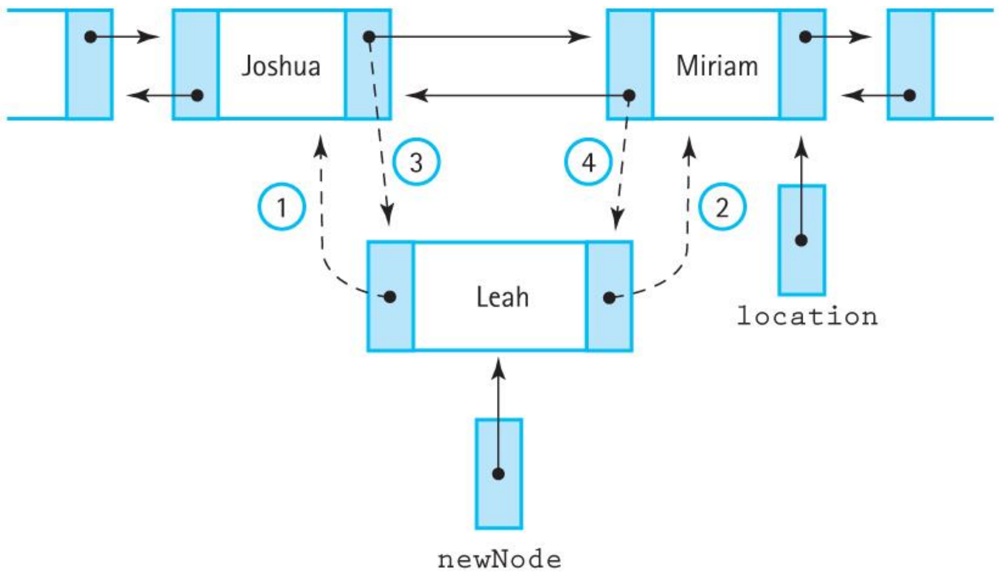


Figure 6.9 Linking the new node into the list

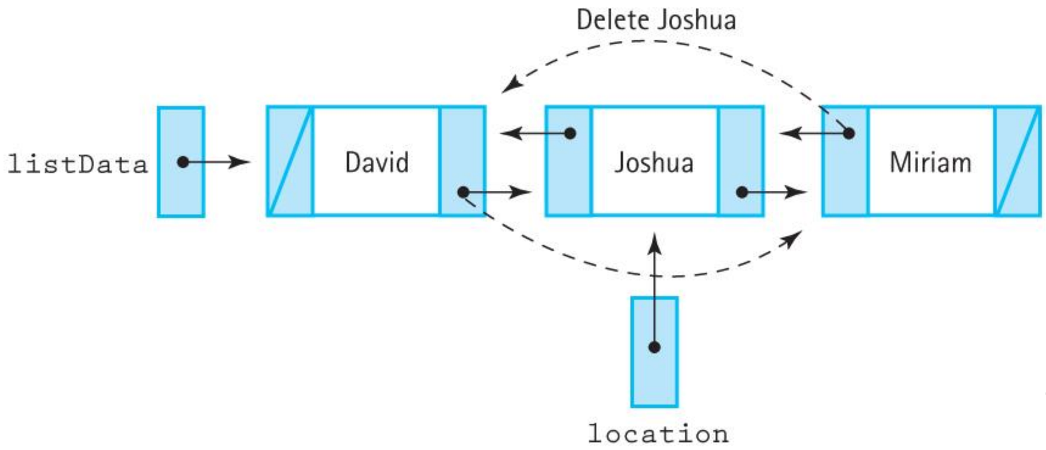


Figure 6.10 Deleting from a doubly linked list

Lists with Headers and Trailers

- Insertion and deletion can be simplified if they never have to choice the first and last nodes
- This can be accomplished using dummy nodes whose values are outside the expected range
- **Header node:** A placeholder at the beginning of a list
- **Trailer node:** A placeholder at the end of a list

Lists with Headers and Trailers (cont.)

- For example, a list of students probably won’t have students named “AAAAA” and “ZZZZZ”
- Could also use blank nodes and always skip processing those nodes

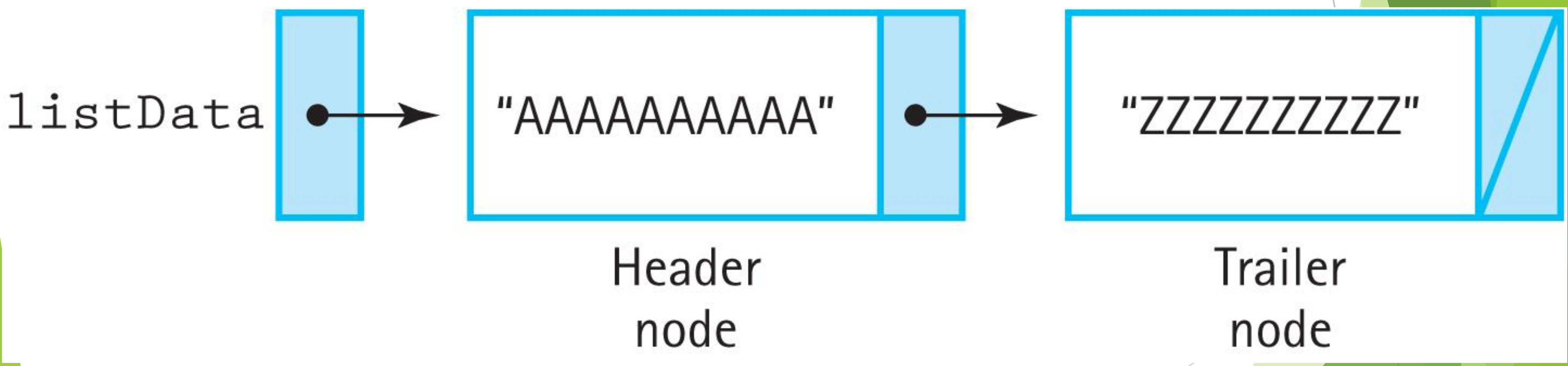


Figure 6.11 An “empty” list with a header and a trailer

Copying Structures

- What would happen if we tried to create a copy of a stack?
- Can’t just pop off elements and push them into a new stack, since that could destroy the original stack
- We could try to recreate the stack, but it wouldn’t be sent back to the caller
- Solution: Use a **copy constructor**

Shallow and Deep Copying

- **Shallow copy:** An operation that copies a class object without copying any pointed-to data
- **Deep copy:** An operation that copies a class object and any additional data it points to
- C++ uses shallow copying by default when passing objects by value, returning an object from a function, and for the assignment operator (e.g., `stack1 = stack2;`)

Shallow and Deep Copying (cont.)

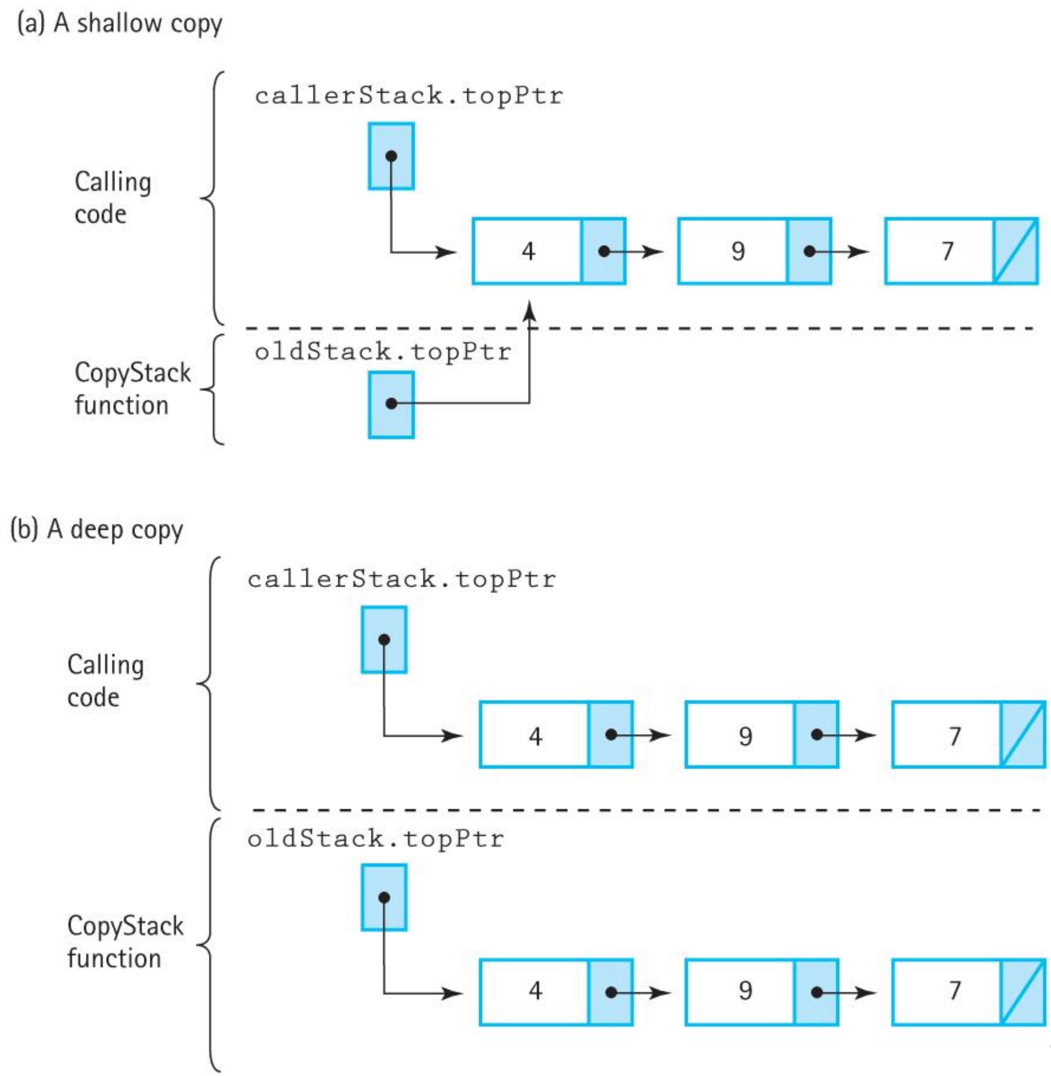


Figure 6.13 Shallow copy versus deep copy of a stack (a) A shallow copy (b) A deep copy

Copy Constructors

- **Copy constructor:** A constructor that is invoked when a copy of a class is created, such as when an object is passed by value, returned from a function or initialized in a declaration
  - Assignment is a special case that requires other methods to override
- A copy constructor looks like a regular constructor but takes an object of the same type as a reference parameter

Copying in Assignment

- There are two approaches for allowing copying in an assignment statement, such as stack1 = stack2
- One is to write a member function that will handle the copying instead of using assignment
- The other is to overload the assignment operator

Copy Function

- Example: stack1.CopyStack(stack2)
  - Should this copy stack2 into stack1, or stack1 into stack2? The syntax is ambiguous
- Another option is: CopyStack(stack1, stack2)
  - Can clearly say or stack1 is copied into stack2
- This can be accomplished using a *friend function*

C++ Friend Functions

- **Friend function:** A function that is not a member of a class but has direct access to all of the private members of a class
- They must be declared within a class definition:
  - `friend void Copy(StackType<ItemType>, StackType<ItemType>&);`
- Copy has no implicit self like member functions do, and must access private members using the parameters

Operator Overloading

- It would be nice if **`myStack = yourStack`** created a deep copy instead of a shallow copy
- C++ allows classes to overload operators such as “=” using member functions:
  - Definition: void operator=(StackType<ItemType>);
  - Operators can be overloaded with different parameters as many times as needed
- Some operators can’t be overloaded: `::`, `sizeof`, `.`, and `?:`

Operator Overloading Examples

- **Operator=**: Used for deep copying; should behave the same as a Copy(dest, source) function
- **Operator< and Operator>:** Relational operators; should *not* modify either argument
- **Operator==:** Check if two objects are equal

Operator Overloading Guidelines

- At least one operand of the overloaded operator must be a class instance
- You cannot change the order of operators, define new operators, or change the number of operands of an operator
- In some situations, it is clearer to implement the overloading function as a friend function instead of a member function

Operator Overloading Guidelines (cont.)

- Overloading ++ and -- requires client code to use the prefix form: ++someObject
- Operator functions must be member functions when overloading =, (), <> and ->
  - Think very carefully when overloading (), <> and ->
- The stream operators << and >> must be overloaded using a friend function
- The compiler must be able to distinguish between the data types of the operands

Linked List as an Array of Records

- "Array vs. Linked List" is not the same as "static vs. dynamic allocation"
  - Dynamically allocated arrays, for example
  - But what about statically allocated linked lists?
- It's possible to implement a linked list using an array with the indices acting as links

Linked List as an Array of Records

(a) A linked list in static storage

```
struct NodeType
{
  char info;
  int next;
};
struct ListType
{
  NodeType nodes[5];
  int first;
};
ListType list;
```

list .nodes	info	next
[0]	C	4
[1]	B	0
[2]	E	-1
[3]	A	1
[4]	D	2
.first	3	

(b) A linked list in dynamic storage

```
struct NodeType
{
  char info;
  NodeType* next;
};

NodeType* list;
list = new NodeType;
```

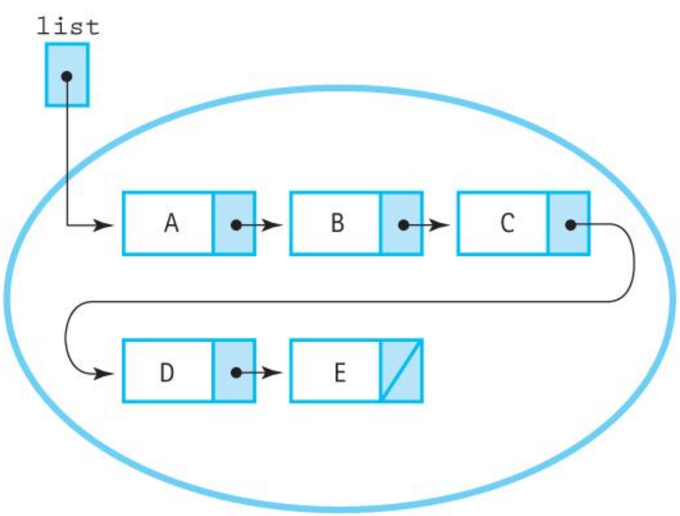


Figure 6.16 Linked lists in static and dynamic storage (a) A linked list in static storage (b) A linked list in dynamic storage

Why Use an Array?

- All the advantages of linked lists without needing to allocate memory or use pointers
  - Some languages or systems don't support these features
  - The overhead can be too high for some applications
- Easier to store an array-based list between runs of the program

How Is an Array Used?

- Each node in the array holds data and contains the index of the next node in the list
  - The last node uses -1 (“NUL”) instead of NULL
- The array contains nodes (data + index) and free space (no data)
- Inserting an item uses a free space, and deleting an item frees up a space
- The free space is handled as a list

Sorted List in an Array of Records

Note that in both there are two lists: the store data and the free space

nodes	.info	.next
[0]	David	4
[1]		5
[2]	Miriam	6
[3]		8
[4]	Joshua	7
[5]		3
[6]	Robert	NUL
[7]	Leah	2
[8]		9
[9]		NUL

list	0
free	1

Figure 6.18 An array with a linked list of values and free space

Array of Records Operations

- Array of records uses three bookkeeping functions
- InitializeMemory: All nodes are added to the free list and linked together
- GetNode: Return the index of the first free node and update the free list pointer
- FreeNode: Return a node to the free list
- The array itself is stored in a struct along with the index of the free list

Array of Records vs. Linked List

The array of records indices and the linked list pointers can be used the same ways:

Design Notation/Algorithm	Dynamic Pointers	Array-of-Records "Pointers"
Node(location)	*location	storage.nodes<location>
Info(location)	location->info	storage.nodes<location>.info
Next(location)	location->next	storage.nodes<location>.next
Set location to Next(location)	location = location->next	location = storage.nodes<location>.next
Set Info(location) to value	location->info = value	storage.nodes<location>.info = value
Allocate a node	nodePtr = new NodeType	GetNode(nodePtr)
Deallocate a node	delete nodePtr	FreeNode(nodePtr)

Array as Dynamic Memory

- A single array is used as a “heap” with multiple lists allocating nodes in the array
- Each list allocates nodes from the free list, similar to how dynamic memory works
- Multiple external pointers point to the lists inside the array

Array as Dynamic Memory (cont.)

free: 7

nodes	.info	.next
[0]	John	4
[1]	Mark	5
[2]		3
[3]		NUL
[4]	Nell	8
[5]	Naomi	6
[6]	Robert	NUL
[7]		2
[8]	Susan	9
[9]	Susanne	NUL

list1: 0

list2: 1

Figure 6.19 An array with three lists (including the free list)

Polymorphism with Virtual Functions

- **Polymorphism:** The ability to determine which function to apply to an object; one of the three fundamentals of object-oriented programming
- **Dynamic binding:** The function is chosen at run time, based on the type and number of arguments

Polymorphism with Virtual Functions (cont.)

- C++ forces formal parameters and actual parameters to have the same type
- Inheritance relaxes this: the actual parameter's class can be a subclass of the formal parameter
- Consider the line: `formalParam.MemberFunc();`
  - If the actual parameter could be any subclass of the formal parameter, which version of the member function is invoked?

Virtual Member Functions

```
formalParam.MemberFunc();
```

- If MemberFunc is **not virtual**, the formal parameter’s version is called
- If MemberFunc is **virtual**, the actual parameter’s version is called

Polymorphism and Parameters

- When will the derived class's methods be used?
  - The actual parameter is passed as a reference
  - The actual parameter is a pointer defined as a pointer to the base class and points to an object of the derived class
- Passing a derived class by value makes only the subobject of the base class available in the function

Polymorphism Example

```
#include <iostream>
class One {
    public:
        virtual void Print() const;
};
class Two : public One {
    public:
        void Print() const;
};
void One::Print() const {
    std::cout << "Print class One" << std::endl;
}
void Two::Print() const {
    std::cout << "Print class Two " << std::endl;
}
```

Polymorphism Example (cont.)

```
// Takes a base-class object by reference
void PrintRef(One& ptr) {
    ptr.Print();
}
// Takes a pointer to a base-class object
void PrintPtr(One* ptr) {
    ptr->Print();
}
// Takes a base-class object by value
void PrintVal(One ptr) {
    ptr.Print();
}
```

Polymorphism Example (cont.)

```
int main() {
    using namespace std;
    One one;
    Two two;
    cout << "Result of printing one:" << endl;
    PrintRef(one);     // Output: "Print class One"
    PrintPtr(&one);    // Output: "Print class One"
    PrintVal(one);          // Output: "Print class One"
    cout << "Result of printing two:" << endl;
    PrintRef(two);     // Output: "Print class Two"
    PrintPtr(&two);    // Output: "Print class Two"
    PrintVal(two);     // Output: "Print class One"
    cout << "Pointer to a derived type:" << endl;
    One *onePtr = new One;
    PrintPtr(onePtr); // Output: "Print class One"
    onePtr = new Two;
    PrintPtr(onePtr); // Output: "Print class Two"
    return 0;
}
```

A Specialized List ADT

- The earlier list ADTs have some restrictions:
  - List elements must be unique – no duplicates!
  - Clients can only iterate forward (from beginning to end) through the list
- For some applications, this is not enough
- Case study: A list of integers that supports duplicate elements, bidirectional iteration, and inserting elements at either end of the list

A Specialized List ADT (cont.)

- Which structure is best for this?
  - Double linked allows iteration in both directions, except it doesn’t give access to the last element, which is where reverse iteration begins
  - A circular linked list does provide access to the last element, making inserting at either end easier, but only supports iteration in one direction
  - Both support duplicate elements
- Combine them: doubly linked circular list

List Operations

- Don’t need IsFull, GetItem, or DeleteItem
- PutFront and PutEnd: Insert items at desired end of the list
- GetLength: Return number of items in the list
- ResetForward and ResetBackward: Reset iteration for the desired direction
- GetNextItem and GetPriorItem: Advance iteration of the desired direction

Inserting Items

PutEnd is the same as PutFront, except the external pointer is updated.

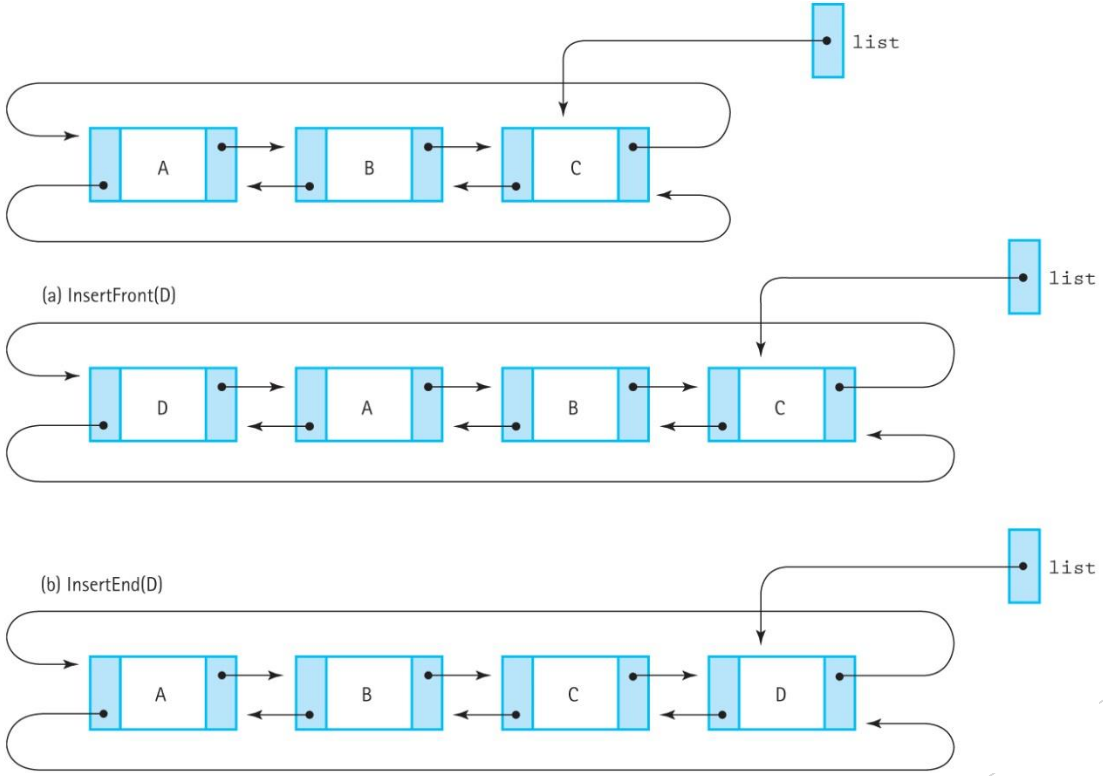


Figure 6.22 Putting at the front and at the rear (a) InsertFront(D) (b) InsertEnd(D)

List Iterations

- Allow simultaneous iteration in both directions
- Keep track of iteration position going forward and going backward using two fields
- Calling a forward iteration method (ResetForward or GetNextItem) has no effect on the backward iteration methods (ResetBackward and GetPriorItem) and vice versa

Range-Based Iteration

- Basic C++ for loop:
  - `for (j = 0; j < length; j++)`
  - Several possible errors: initialize j to the wrong value (j = 0 or j = 1?), or wrong terminating condition (j < length or j <= length?)
  - Tight control over the index j, such as allowing j += 2 instead of just j++

Ranged-Based Iteration (cont.)

- Ranged-based C++ for loop:
  - `for (ElementType e : list)`
  - Abstracts away the iteration index, eliminating common errors
  - Only allows processing list in order
  - Actually just **syntactic sugar** for the regular *for* loop
- **Syntactic Sugar:** An alternative form or syntax that makes programs easier to read or write

C++ Iterators

- **Iterator:** A class that implements iteration over a particular ADT; must overload dereference (*), increment (++), and inequality (!=)
- The ADT provides begin() and end() methods that return iterators representing the beginning and end of the collection
- The range-based *for* loop unfolds into a *for* loop that uses these tools

C++ Iterators (cont.)

```
// a list of numbers to process
list<int> numbers(1,2,3,4);
for (int e : numbers) {process(i);}
// The loop is compiled into
// something like:
for (iterator it = numbers.begin();
  it != numbers.end(); ++it)
    {process(i);}
```

Implementing Iterators

- Iterators are named after the ADT they iterate over, e.g., SortedTypeIterator for the SortedType list implementation
- They implement the following operators:
  - Dereference: Returns the current element of the list
  - Increment: Advances the iterator forward one step
  - Inequality: Compares two iterators (e.g., checking if the iterator has reached the end of the list)

SortedListIterator

- Encapsulates a pointer into the list as a `NodeType<T>* item`
- Dereference: Returns `item->info`
- Increment: Updates `item = item->next`, or sets item to NULL if it’s the end of the list
- Inequality: Checks if both iterators point to the same item