

# Pointers and References

## **LAB 13: Manipulating a Linked-List**

Rung-Bin Lin

International Bachelor Program in Informatics

Yuan Ze University

12/27/2022

# Pointers

- Pointer?

A variable used to store the address of other variable. Hence, a pointer variable also has its own address.

```
int count = 10, *countPtr, *aryPtr, *aryElt;
```

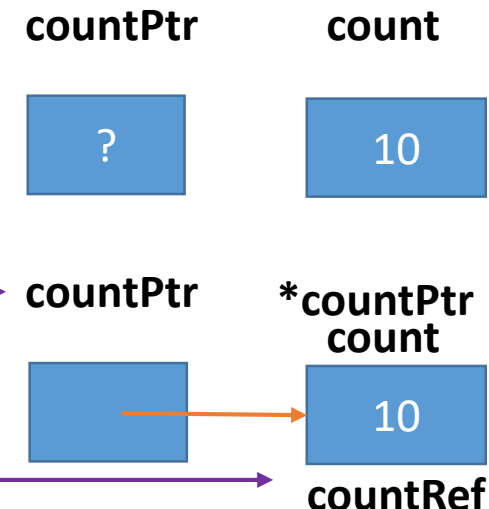
- Reference?

```
int *countPtr=&count;
```

Address-of operator

```
int &countRef = count;
```

Denoting a reference variable



**cout << count << \*countPtr; // what are printed?**

- Array address

```
int intAry[20]; // array name is a constant pointer
```

```
int *aryPtr = intAry;
```

```
int *aryElt = &intAry[12];
```

# Four Types of Pointers

- Nonconstant pointer to nonconstant data

- Pointer and data both can be modified

```
int *countPtr;
```

- Nonconstant pointer to constant data

- Pointer can be modified but data cannot

```
const int *countPtr; // not need initialize because what it declares is  
a nonconstant pointer rather than a constant integer.
```

- Constant pointer to nonconstant data

- Data can be modified but pointer cannot

```
int x;
```

```
int * const countPtr = &x; // must be initialized
```

- Constant pointer to constant data

- Pointer and data cannot be modified

```
const int x=5;
```

```
const int *const countPtr = &x; // must be initialized
```

# Pointers & References

```
int a;
```

```
int *aPtr; // aPtr is a variable whose value is  
           // an address that can hold an integer.
```

```
a=7;
```

```
aPtr = &a;
```

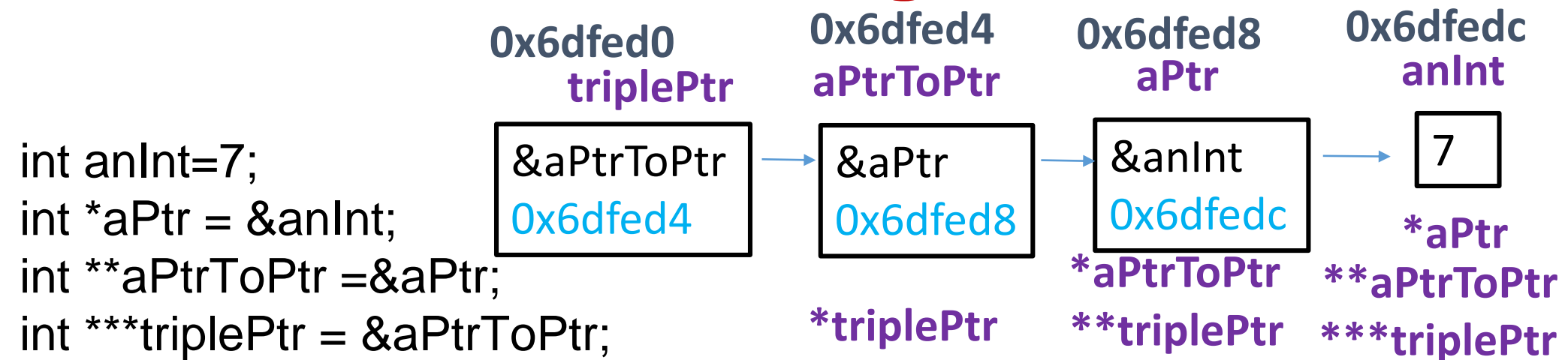
```
cout << &a << aPtr << *&aPtr << &*aPtr;
```

```
cout << a << *aPtr ;
```

**All print out the address of a.**

**The value stored in the address (location)  
pointed by aPtr.**

# Pointer Pointing to a Pointer



**aPtrToPtr** is a pointer that points to the pointer **aPtr**. **aPtr** is a pointer that points to the integer **anInt**.

That is to say, `aPtrToPtr` is an address where stores the address of `aPtr`, and `aPtr` is an address where stores the address of `anInt`. Try below.

```

cout << ***triplePtr << " " << **triplePtr << " " << *triplePtr << " " << triplePtr << " " << &triplePtr << endl;
cout << anInt << " " << aPtr << " " << aPtrToPtr << " " << triplePtr << " " << &triplePtr << endl;
cout << &anInt << " " << aPtr << " " << *aPtrToPtr << " " << **triplePtr << endl;
cout << anInt << " " << *aPtr << " " << **aPtrToPtr << " " << ***triplePtr << endl;

```

This is the output:

```

7 0x6dfedc 0x6dfed8 0x6dfed4 0x6dfed0
7 0x6dfedc 0x6dfed8 0x6dfed4 0x6dfed0
0x6dfedc 0x6dfedc 0x6dfedc 0x6dfedc
7 7 7 7

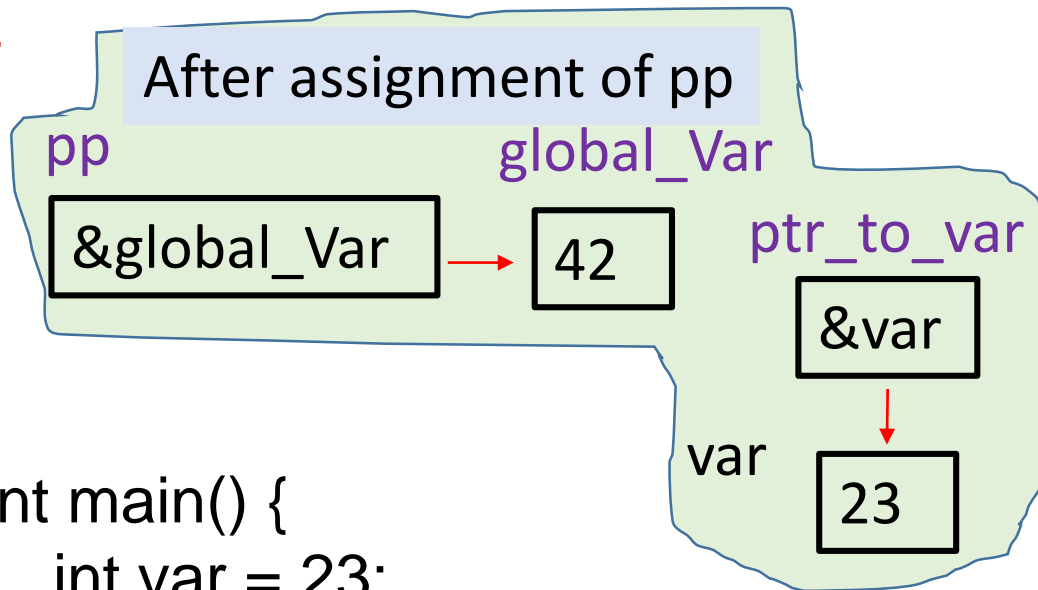
```

**Passing Reference to a Pointer in C++**

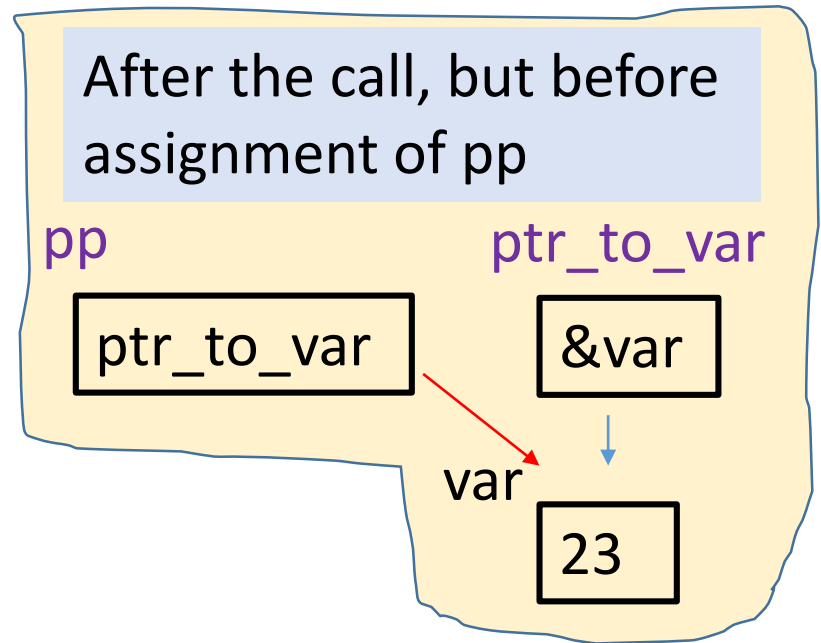
<https://www.geeksforgeeks.org/passing-reference-to-a-pointer-in-c/>

# Example for Modifying Pointer Pointing to a Pointer in a Function: Not working

```
int global_Var = 42;  
// function to change pointer value  
void changePointerValue(int *pp) {  
    pp = &global_Var;  
}
```



```
int main() {  
    int var = 23;  
    int *ptr_to_var = &var;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr_to_var << endl; // display 23  
    changePointerValue(ptr_to_var);  
    cout << "After :" << *ptr_to_var << endl; // display 23  
}
```

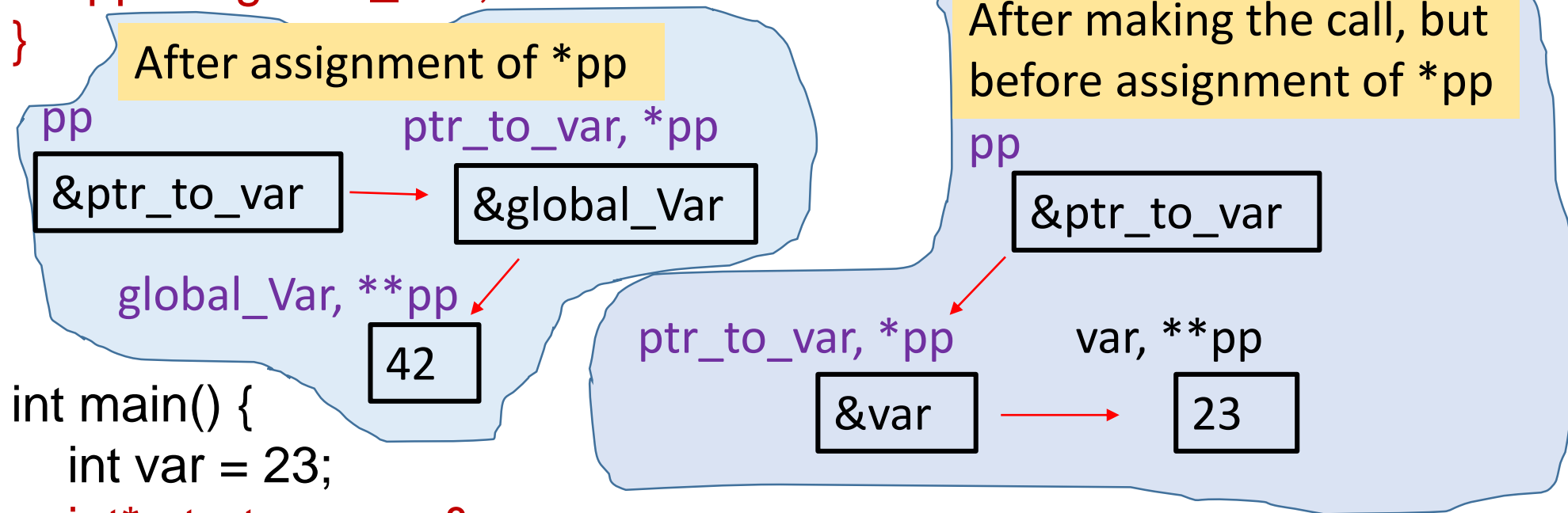


**We Intend to modify the pointer stored in ptr\_to\_var from &var to &global\_Var.**

# Example for Modifying Pointer Pointing to a Pointer in a Function: Working Fine

```
int global_Var = 42;  
// function to change pointer value  
void changePointerValue( int** pp) {  
    *pp = &global_Var;  
}
```

This enables us to modify the pointer stored in `ptr_to_var` from `&var` to `&global_Var`.



```
int main() {  
    int var = 23;  
    int* ptr_to_var = &var;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr_to_var << endl; // display 23  
    changePointerValue( &ptr_to_var );  
    cout << "After :" << *ptr_to_var << endl; // display 42  
}
```

# Example for Modifying a Reference to a Pointer in a Function: Working Fine

```
int global_Var = 42;  
// function to change pointer value  
// pp is a reference to a pointer to an integer  
void changePointerValue( int *&pp) {  
pp = &global_Var;  
}
```

PP is a reference to an int pointer.

This also enables us to modify the pointer stored in ptr\_to\_var from &var to &global\_Var.

After assignment of pp



After making the call, but before assignment of pp



```
int main() {  
    int var = 23;  
    int *ptr_to_var = &var;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr_to_var << endl; // display 23  
    changePointerValue( ptr_to_var );  
    cout << "After :" << *ptr_to_var << endl; // display 42  
}
```



# Pass-by-reference VS. Pass-by-value

```
int cubeByValue( int );
```

```
int main()  
{  
    int number = 5;  
    number =  
cubeByValue( number );  
  
} // end main
```

```
int cubeByValue( int n )  
{  
    return n * n * n;  
}
```

```
void cubeByReference( int * );
```

```
int main()  
{  
    int number = 5;  
    cubeByReference( &number );  
} // end main
```

```
void cubeByReference( int *nPtr )  
{  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}
```

# sizeof Operator

```
char c; // variable of type char
short s; // variable of type short
int i; // variable of type int
long l; // variable of type long
float f; // variable of type float
double d; // variable of type double
long double ld; // type long double
int array[ 20 ]; // array of int
int *ptr = array; // type int *
```

**sizeof** is an operator that gives the number of bytes taken by a type or a variable.

```
cout << "sizeof c = " << sizeof c
<< "\\tsizeof(char) = " << sizeof( char )
<< "\\nsizeof s = " << sizeof s
<< "\\tsizeof(short) = " << sizeof( short )
<< "\\nsizeof i = " << sizeof i
<< "\\tsizeof(int) = " << sizeof( int )
<< "\\nsizeof l = " << sizeof l
<< "\\tsizeof(long) = " << sizeof( long )
<< "\\nsizeof f = " << sizeof f
<< "\\tsizeof(float) = " << sizeof( float )
<< "\\nsizeof d = " << sizeof d
<< "\\tsizeof(double) = " << sizeof( double )
<< "\\nsizeof ld = " << sizeof ld
<< "\\tsizeof(long double) = " << sizeof( long
double )
<< "\\nsizeof array = " << sizeof array
<< "\\nsizeof ptr = " << sizeof ptr << endl;
```

# Pointer & Array

```
int main()
{
    int b[] = { 10, 20, 30, 40 };
    int *bPtr = b;

    for ( int i = 0; i < 4; i++ )
        cout << "b[" << i << "] = " << b[ i ] << '\n';


    for ( int offset1 = 0; offset1 < 4; offset1++ )
        cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';

    for ( int j = 0; j < 4; j++ )
        cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

    for ( int offset2 = 0; offset2 < 4; offset2++ )
        cout << "*(bPtr + " << offset2 << ") = "
            << *( bPtr + offset2 ) << '\n';

} // end main
```

This is so-called pointer arithmetic, especially when used along with an array. Hence, **b+offset1** refers to the (offset1)-th element in array b.



# Pointer-Based string Processing

```
char color[ ]="blue";  
const char *colorPtr = "blue";  
char colorx[ ]= {'b', 'l', 'u', 'e', '\0'};  
const char* const suit[4] = {"Hearts", "Diamonds",  
"Clubs", "Spades"}; // Array of pointers
```

# Function Pointers (7.12)

// prototypes

void selectionSort( int [ ], const int, **bool (\*)( int, int )** );

void swap( int \* const, int \* const );

bool ascending( int, int ); // implements ascending order

bool descending( int, int ); // implements descending order

Int main () {

if ( order == 1 )

    selectionSort( a, arraySize, ascending );

else

    selectionSort( a, arraySize, descending );

.....

}

Function pointer



void selectionSort( int work[ ], const int size, **bool (\*compare)( int, int )** )

{

}

# Elaboration

- With the new keyword...
  - `MyClass* myClass = new MyClass();`
  - `myClass->MyField = "Hello world!";`
- Without the new keyword...
  - `MyClass myClass;`
  - `myClass.MyField = "Hello world!";`
- What are the differences between with and without using **new** function?
  - <https://stackoverflow.com/questions/655065/when-should-i-use-the-new-keyword-in-c>

# LAB 13: Manipulating a Linked-List

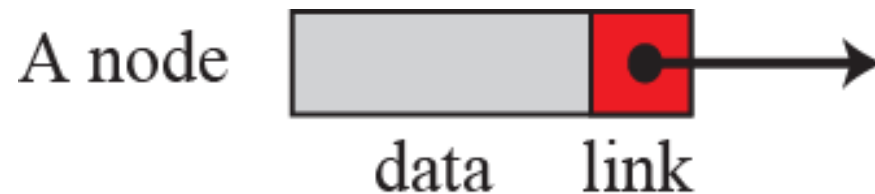
- This lab has two parts. In the first part, you should write a function to create a sorted linked-list from a sequence of integers read from keyboard. There should be no duplicate numbers in the linked-list. The numbers are sorted in the descending order in the linked-list. After creating the linked list, you have to print out the linked-list.
- In the second part, you should write a function to delete some elements from the linked-list. The number to be deleted are first read from keyboard. If a number read can be found in the linked-list, the node that stores the number should be deleted. If it is not found, nothing has to be done. After deleting all the integers found in the linked-list, display the numbers that remain in the linked-list.



# Use of **struct** to Define a node

- **struct** is a language construct used to create a user-defined data type that can hold several data items of different types together as a whole. For example, we can define a type of NODE as follows:

```
struct NODE {  
    int data;  
    NODE *link;  
};
```



The above struct can be used to define a node (variable) that has two fields, **data** and **link**, as the figure shown above.

```
NODE aNode; // aNode is a node.
```

```
NODE *nodePtr; // A pointer pointing to a node of type NODE
```



# Creating a Node

- Two ways: declaration and dynamic allocation

- Declaration

```
NODE aNode; // aNode is a node.  
aNode.data = data; // store data into the node  
aNode.link = NULL; // set the link of the node to NULL
```

- Dynamic allocation

```
NODE *aNode; // aNode is a pointer to a node  
aNode = new NODE; // create a node pointed by aNode  
aNode->data = data; // store data into the node  
aNode->link = NULL; // set the link of the node to NULL
```

- If aNode is a **pointer**, use **->** to get access to the fields pointed by aNode. However, aNode must already point to an existing node. If aNode is a **node**, use **.** to get access to the fields in aNode.

- What are the differences between with and without using **new** function?

- <https://stackoverflow.com/questions/655065/when-should-i-use-the-new-keyword-in-c>

# Hints

- To work out this lab, you can refer to the pseudo code in Chapter 11, **Foundations of Computer Science by Behrouz Forouzan**, 4<sup>th</sup> Edition. However, for your convenience, the pseudo code for **SearchLinkedList**, **InsertLinkedList**, and **DeleteLinkedList** is presented at the end of this slide set.
- You should use a reference to a pointer as the head of a linked list to solve the problems in this lab. The reason for this is that the head of a linked list may be updated in a function.

# Using Functions with parameters being references to pointers (1)

- **void createList(NODE \*&head);**
  - Read data from the keyboard and create a linked list. Reading data stops when 0 is read twice consecutively. The data must maintain in the descending order in the linked list.
- **void deleteElements(NODE \*&head);**
  - Read data being checked for deletion from keyboard. If a data item can be found in the linked list, the node storing this data item should be removed from the linked list. Deleting data stops when 0 is read twice consecutively.
- **bool searchLinkedList(NODE \*head, int, NODE \*&prePtr, NODE \*&curPtr);**
  - find whether a data item is already in the linked-list. If not found, the function returns **false**. Otherwise, it return **true**. In any case, the function should also set up two pointers **curPtr** and **prePtr** which will be employed to insert (delete) a data element into (from) the linked-list. Note that **curPtr** and **prePtr** each are a reference to a pointer. The reason of using reference type is that curPtr and prePtr may be updated in this function. The updated curPtr and prePtr must be returned back to the calling function.

# Using Functions with parameters being references to Pointers (2)

- **void insertLinkedList(NODE \*&head, int data);**
  - Insert a data item (i.e., a node) into the linked-list while still maintaining the linked-list sorted. The two pointers, **curPtr** and **prePtr**, passed back from searchLinkedList() should be used for carrying out the **task**.
- **void deleteLinkedList(NODE \*&head, int data);**
  - Delete a data item (i.e., a node) from the linked-list if it can be found in the list. It will do nothing if the data item is not in the linked-list. **This function serves an example to help you develop the code for other functions.**
- **void displayList(NODE \*head);**
  - Print the data items in the linked list in the descending data values. This function will be provided.

# Input and Output formats

## • Input format

- The first line gives the number of test cases. The input to the first part of each test case takes a line that hold a sequence of numbers ended with two 0's. These numbers are used to create a sorted list. The format of the input to the second part of each test case is the same as that of the first part.

## • Output format

- An output line starts with a #. The format is set by the given function for displaying a linked list.

# Main() Function if Functions with parameters being references to Pointers

Your main function should be the same as that shown below. No global variables should be used.

```
int main()
{
    int numTests; // number of test cases
    cin >> numTests;
    for (int i = 0; i<numTests; i++){
        // First part
        NODE *listHead=NULL;
        createList(listHead);
        displayList(listHead);
        // Second part
        deleteElements(listHead);
        displayList(listHead);
        freeList(listHead); // free memory allocated to the linked list
    }
    return 0;
}
```

## **createList(...) & deleteElements(...)**

- In **createList(...)**, you have to call **searchLinkedList(...)** and **insertLinkedList()**. In **insertLinkedList(...)**, you have to read the data elements one-by-one and create a node dynamically for each data element using **new** function. After this, you do the insertion.
- In **deleteElements(...)**, you have to call **searchLinkedList(...)** and **deleteLinkedList(...)**.

# Function for Deleting a Data Element from Linked List

```
void deleteLinkedList(NODE *&listHead, int data)
{
    NODE *curPtr;
    NODE *prePtr;

    if(!searchLinkedList(listHead, data, curPtr, prePtr)) // not found
        return;

    if(prePtr == NULL)
        listHead = curPtr->link; // the node at the head of the linked list is deleted
    else
        prePtr->link = curPtr->link; // the deleted node is not a head node

    curPtr->link = NULL;
    delete curPtr; // Free the memory given to the deleted node
}
```



# Function for Displaying a Linked-List

```
void displayList(NODE *listHead)
{
    int count=0; // Used for counting the number of nodes
    NODE *curPtr = listHead; // Set tempPtr to the head of the linked-list
    cout << "# "; // The first character in a line
    while(curPtr != NULL) // check whether come to the end of a linked list
    {
        if(count%15 == 0 && count != 0) // Start with a new line
            cout << endl << "# ";
        cout << curPtr->data << " "; // Print out data
        curPtr = curPtr->link; // Move to the next node
        count++;
    }
    cout << endl;
    cout << "# Total number of elements in the linked list: " << count << endl;
}
```

# Key Points for Grading

- The main () function and displayList(...) function should not be changed.
- The following four functions should be created.  
**void createList(NODE \*&);**  
**void deleteElements(NODE \*&);**  
**bool searchLinkedList(NODE \*, int, NODE \*&, NODE \*&);**  
**void insertLinkedList(NODE \*&, int);**
- The output should be correct.

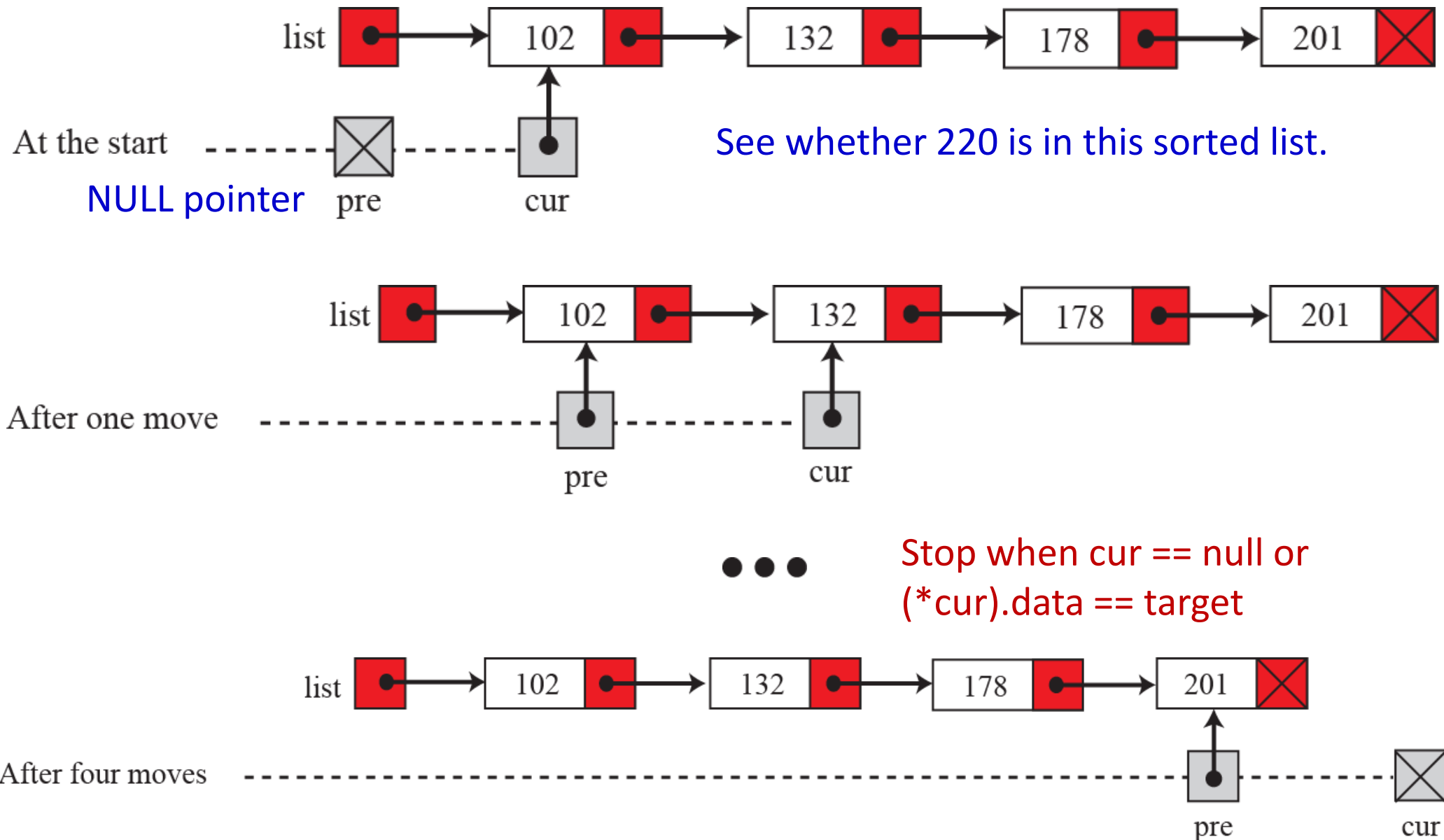
# Input and Output Example

```
3
120 -1 113 65 -43 0 23 44 8 65 -1 0 0
# 120 113 65 44 23 8 0 -1 -43
# Total number of elements in the linked list: 9
65 65 -1 43 44 2 0 0
# 120 113 23 8 -43
# Total number of elements in the linked list: 5
7 9 -5 8 8 -5 0 10 0 0
# 10 9 8 7 0 -5
# Total number of elements in the linked list: 6
10 9 10 7 0 6 0 0
# 8 -5
# Total number of elements in the linked list: 2
1 2 3 4 5 6 7 8 9 10 17 16 16 15 14 11 13 12 0 10 0 0
# 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3
# 2 1 0
# Total number of elements in the linked list: 18
0 1 2 3 4 5 5 6 6 7 8 9 10 11 12 13 15 14 16 17 18 0 0
#
# Total number of elements in the linked list: 0
```

# Searching a linked list

- Since nodes in a linked list have no names, we use two pointers, *pre* (for previous) and *cur* (for current).
- At the beginning of the search, the *pre* pointer is null and the *cur* pointer points to the first node.
- The search algorithm moves the two pointers together towards the end of the list.
- Figure 11.13 shows the movement of these two pointers through the list in an extreme case scenario: when the target value is larger than any value in the list.
- Note that the pointers *cur* and *pre* in the pseudo code used hereon simply a pointer to a node. However, the pointers *curPtr* and *prePtr* in the parameter list of a function that need be implemented are each a pointer pointing to a pointer which then points to a node. By this way, change of *prePtr* and *curPtr* can be returned to the calling function.

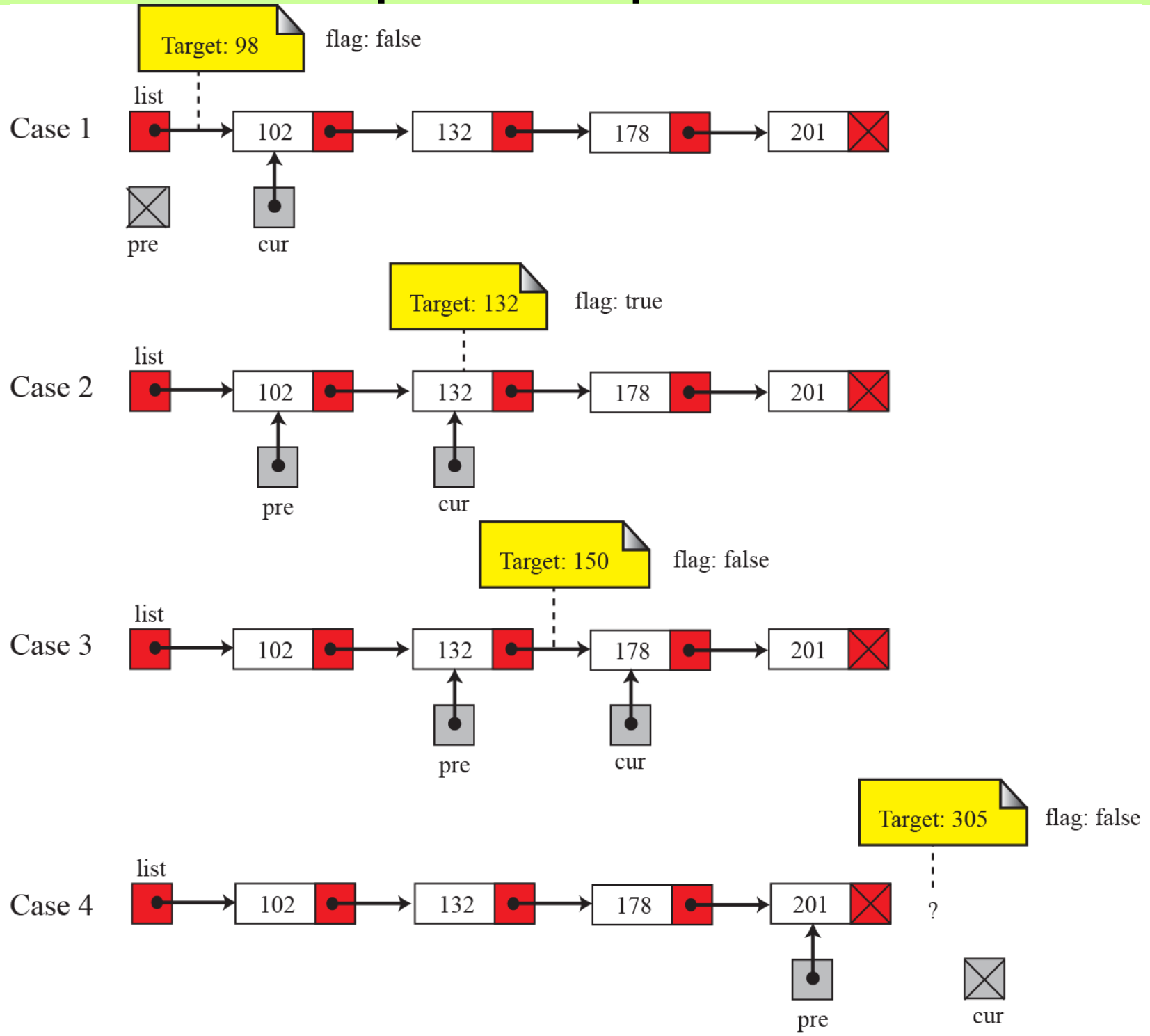
**Figure 11.13: Moving of pre and cur pointers for searching**



Is “pre” pointer needed for this?

ANS: it depends what we are going to do.

**Figure 11.14: Values of pre and cur pointers in different cases**



# Algorithm 11.3: Searching a sorted linked list

## Algorithm: SearchLinkedList (list, target, cur, pre, flag)

Purpose: Search the list using two pointers.

Pre: The linked list (head pointer) and target value

Post: None

Return: pre and cur pointers and flag

```
{
    pre ← null
    cur ← list           // while(cur != null && target > (*cur).data)
    while (target > (*cur).data)
    {
        pre ← cur
        cur ← (*cur).link
    }
    // if (cur != null && target == (*cur).data)
    if ((*cur).data == target)    flag ← true
    else    flag ← false
    return (cur, pre, flag)
}
```

Textbook is not correct

## Inserting a node

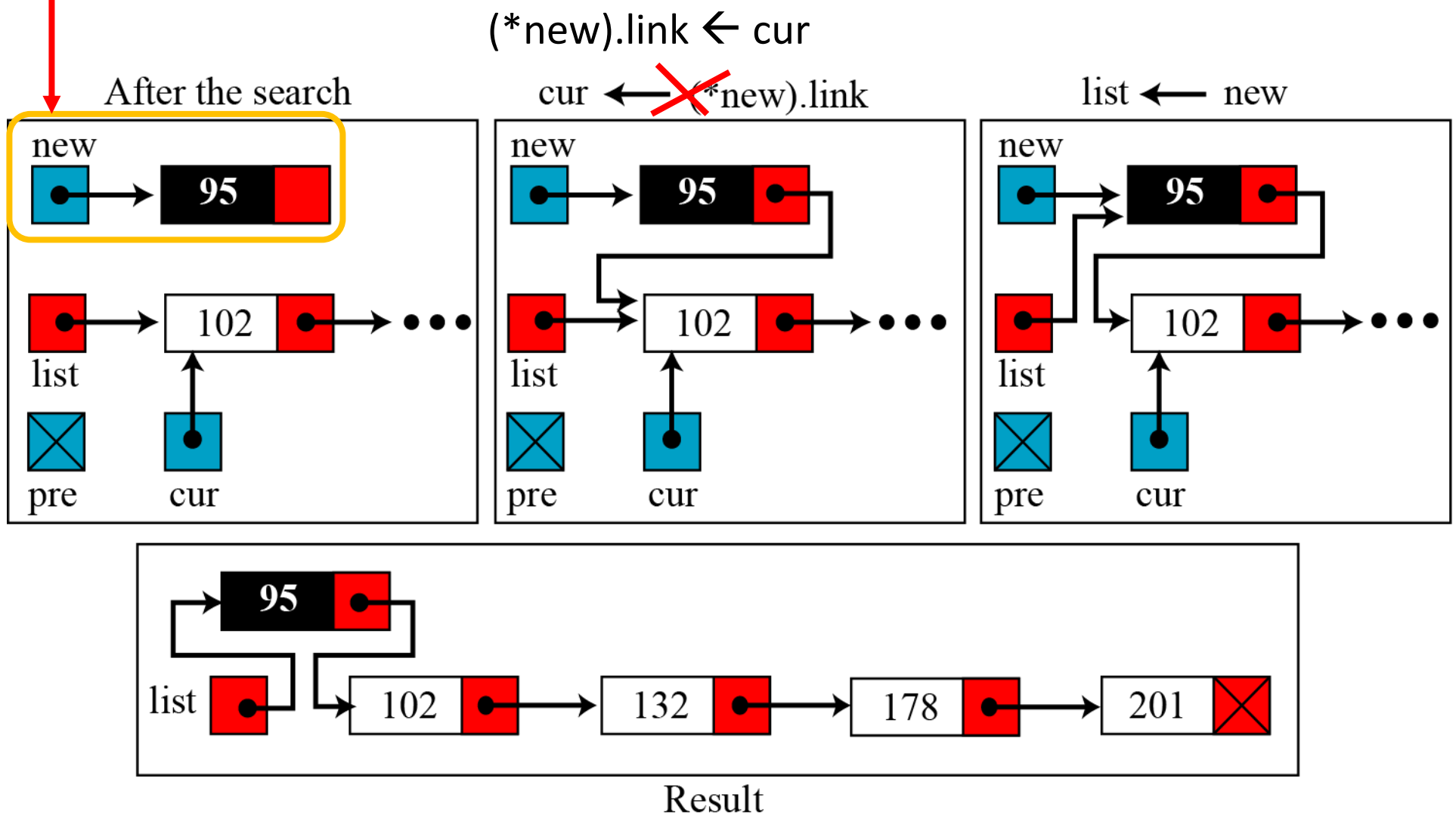
Before insertion into a linked list, we first apply the searching algorithm. If the flag returned from the searching algorithm is false, we will allow insertion, otherwise we abort the insertion algorithm, **because we do not allow data with duplicate values**. Four cases can arise:

- ☐ Inserting into an empty list.
- ☐ Insertion at the beginning of the list.
- ☐ Insertion at the end of the list.
- ☐ Insertion in the middle of the list.

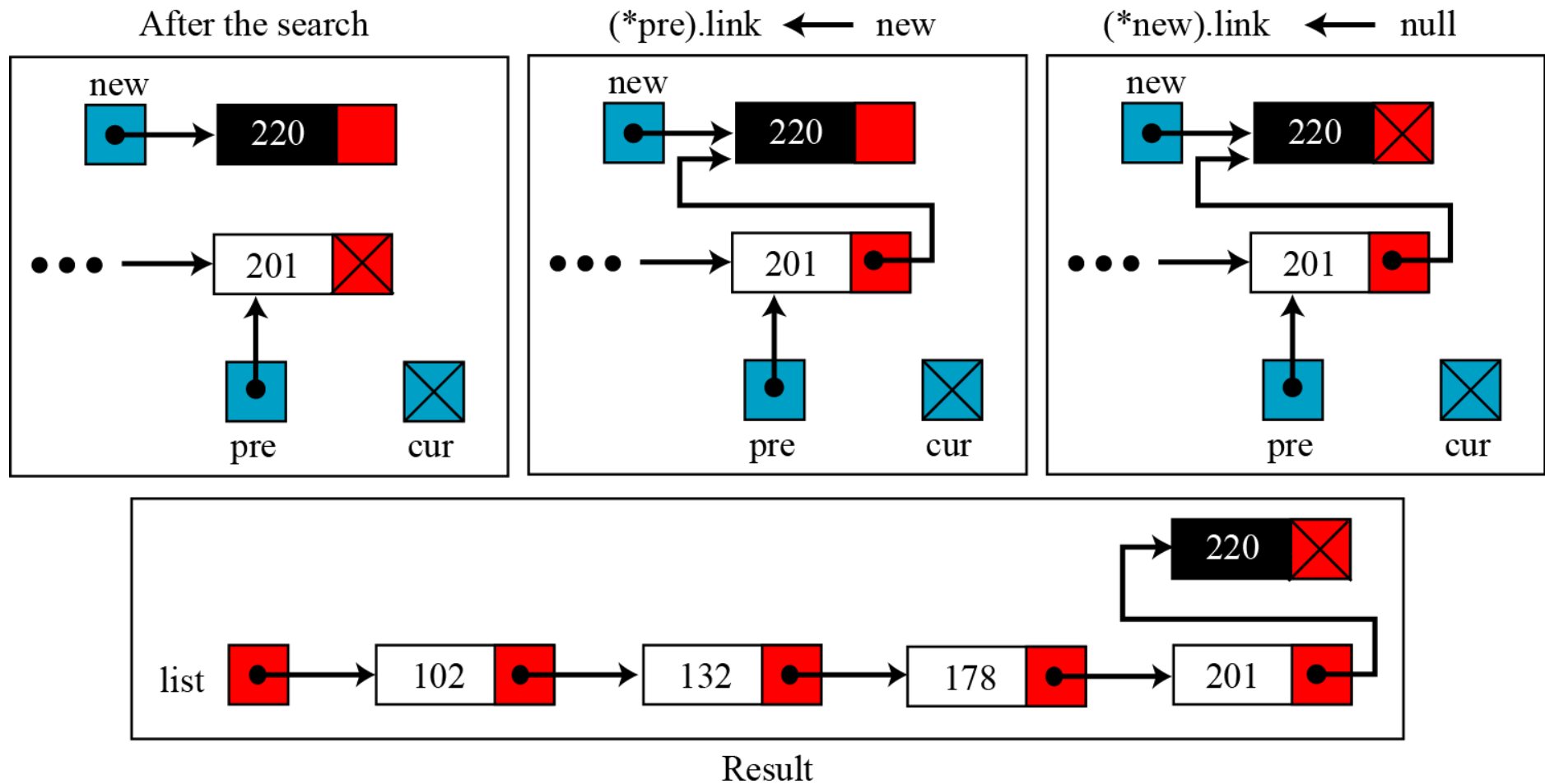


**Figure 11.15** Inserting a node at the beginning of a linked list

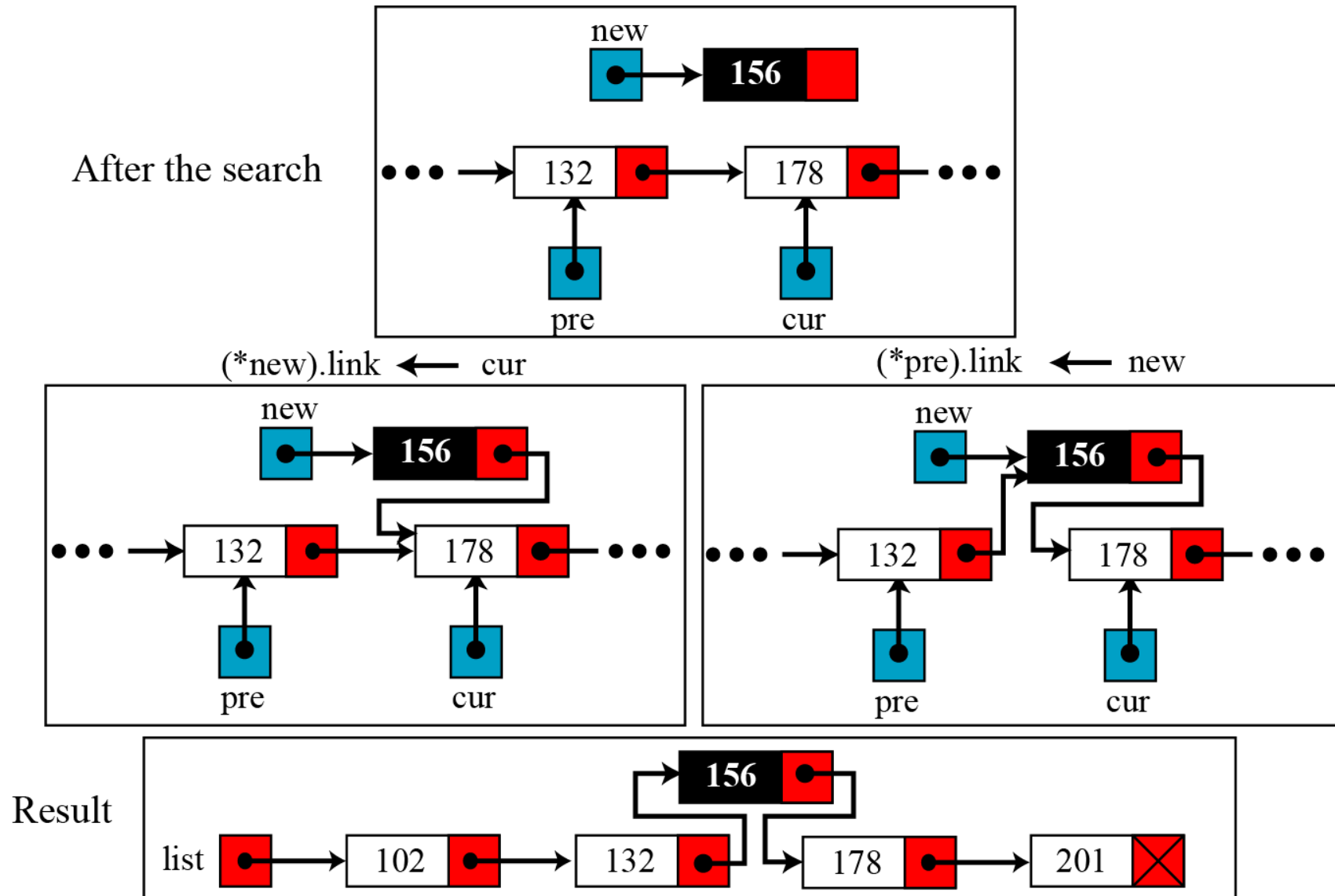
A new node to be inserted.



**Figure 11.16** Inserting a node at the end of the linked list



**Figure 11.17** Inserting a node in the middle of the linked list



## Algorithm 11.4: Inserting a node in a linked list

### Algorithm: InsertLinkedList (list, target, new)

Purpose: Insert a node in the link list after searching the list

Pre: The linked list and “new” node containing the target data to be inserted

Post: None

Return: The new linked list

{

**SearchLinkedList (list, target, pre, cur, flag)**

**// Given target and returning pre, cur, and flag**

**if (flag = true) // No duplicate (i.e., do not insert)**

**return list**

**if (list != null ) // Insert into an empty list**

**{**

**list ← new**

**return list // This statement is missing in the textbook**

**}**

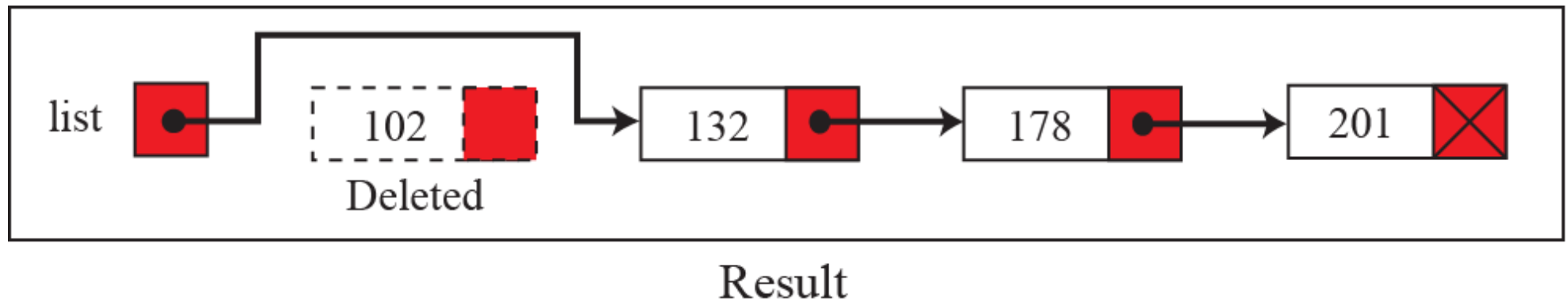
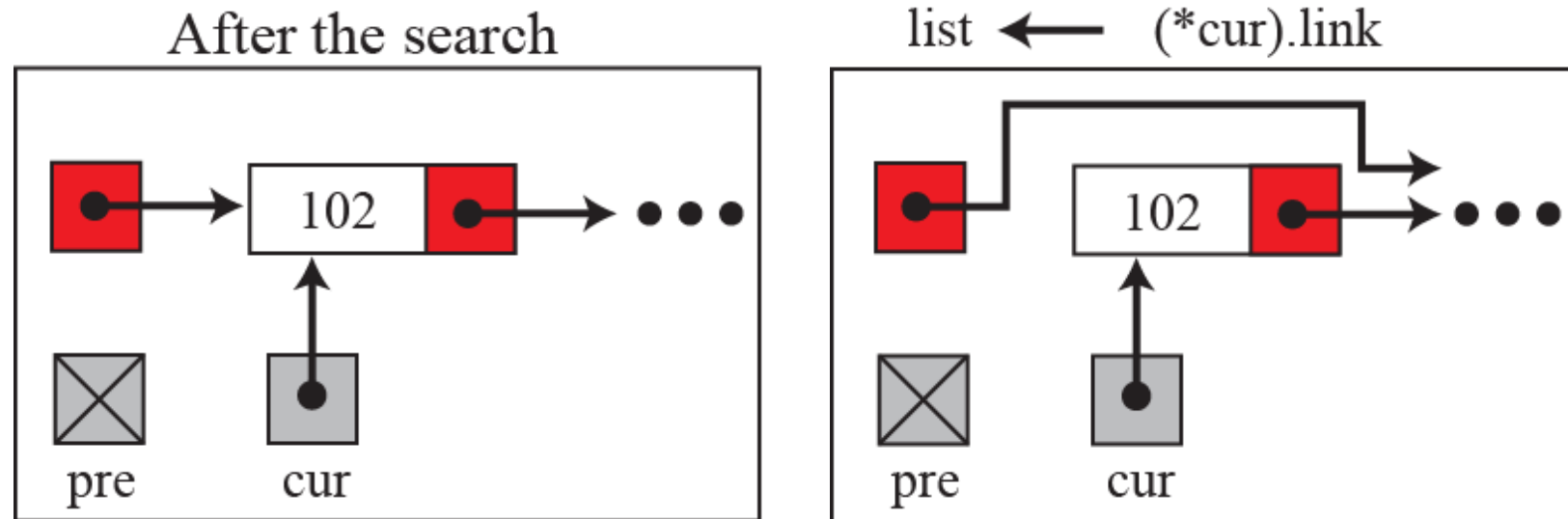
## Algorithm 11.4: Continued

```
if (pre = null)    // Insertion at the beginning
{
    (*new).link ← cur
    list ← new
    return list
}
if (cur = null)    // Insertion at the end
{
    (*pre).link ← new
    (*new).link ← null
    return list
}
(*new).link ← cur  // Insertion in the middle
(*pre).link ← new
return list
}
```

## Deleting a node

Before deleting a node in a linked list, we apply the search algorithm. If the flag returned from the search algorithm is true (the node is found), we can delete the node from the linked list. However, deletion is simpler than insertion: **we have only two cases—deleting the first node and deleting any other node.** In other words, the deletion of the last and the middle nodes can be done by the same process.

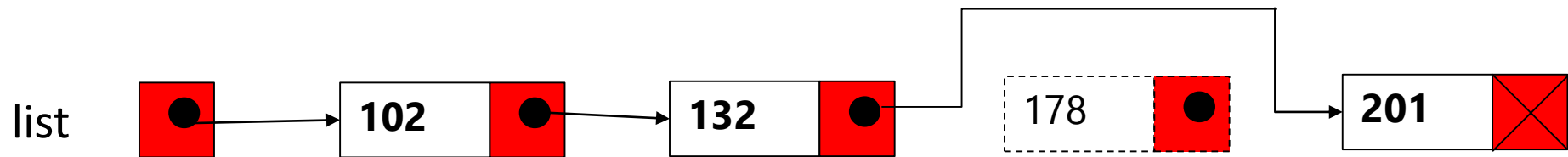
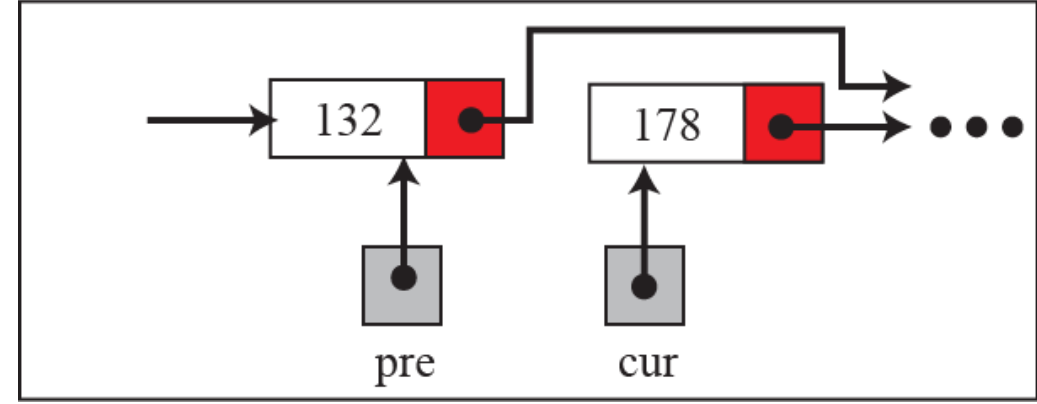
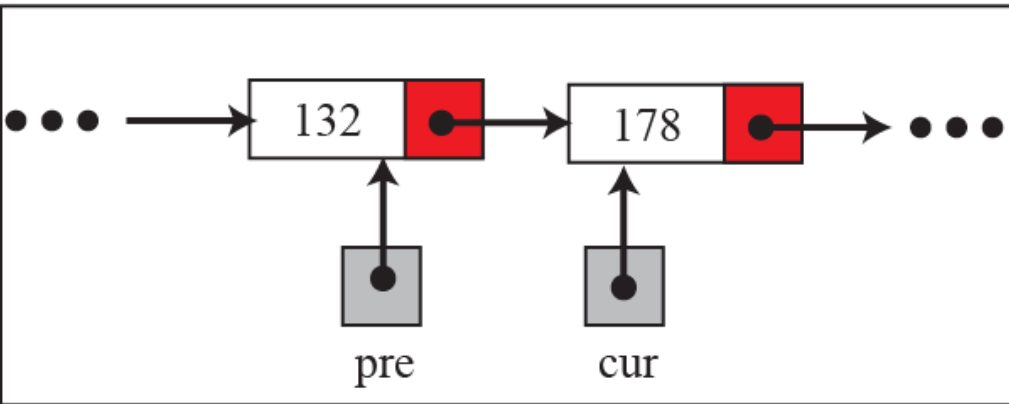
**Figure 11.18: Deleting the first node of a linked list**



**Figure 11.19: Deleting a node at the middle or end**

After the search

$(*pre).link \leftarrow (*cur).link$





# Algorithm 11.5: Deleting a node in a linked list

Algorithm: DeleteLinkedList (list, target)

Purpose: Delete a node in a linked list

Pre: The linked list and the target data to be deleted

Post: None

Return: The new linked list

```
{
    // Given target and returning pre, cur, and flag
    searchLinkedList (list, target, pre, cur, flag)
    if (flag = false)
    {
        return list      // The node to be deleted not found
    }
    if (pre = null)      // Deleting the first node
    {
        list ← (*cur).link
        return list
    }
    (*pre).link ← (*cur).link // Deleting other nodes
    return list
}
```