

Pointers and References

LAB 1: Manipulating a Linked-List

Rung-Bin Lin

International Bachelor Program in Informatics

Yuan Ze University

02/25/2021

Pointers

- Pointer?

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

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

- Reference?

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

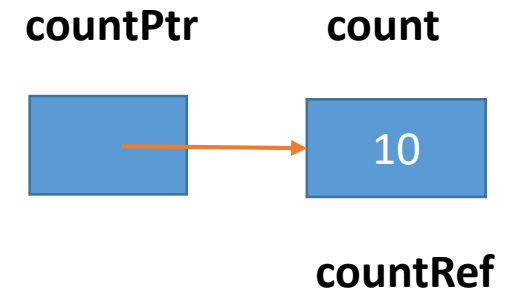
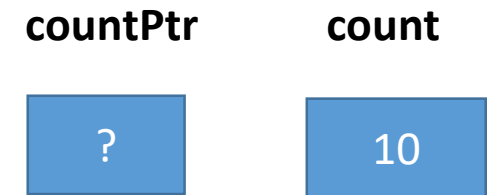
```
int &countRef = count;
```

- 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.**

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';

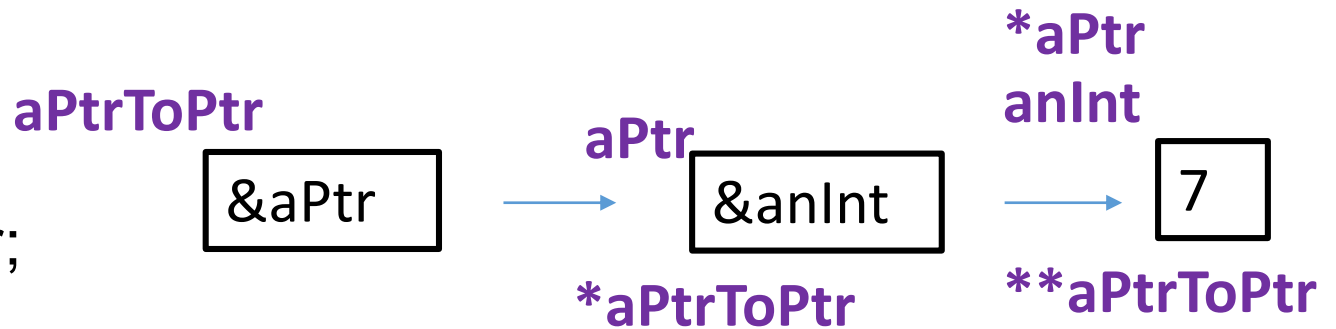
} // end main
```

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 Pointing to a Pointer

```
int anInt=7;  
int *aPtr = &anInt;  
int **aPtrToPtr =&aPtr;
```



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

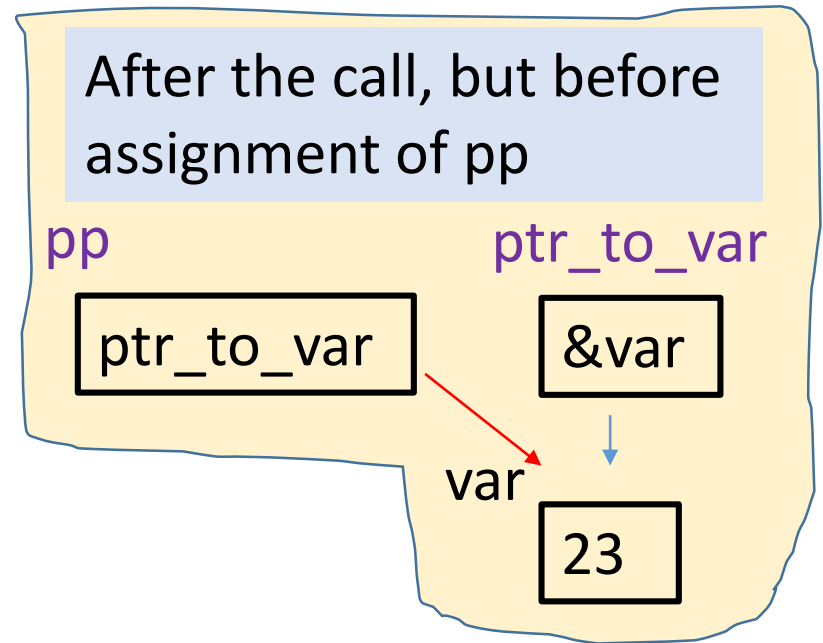
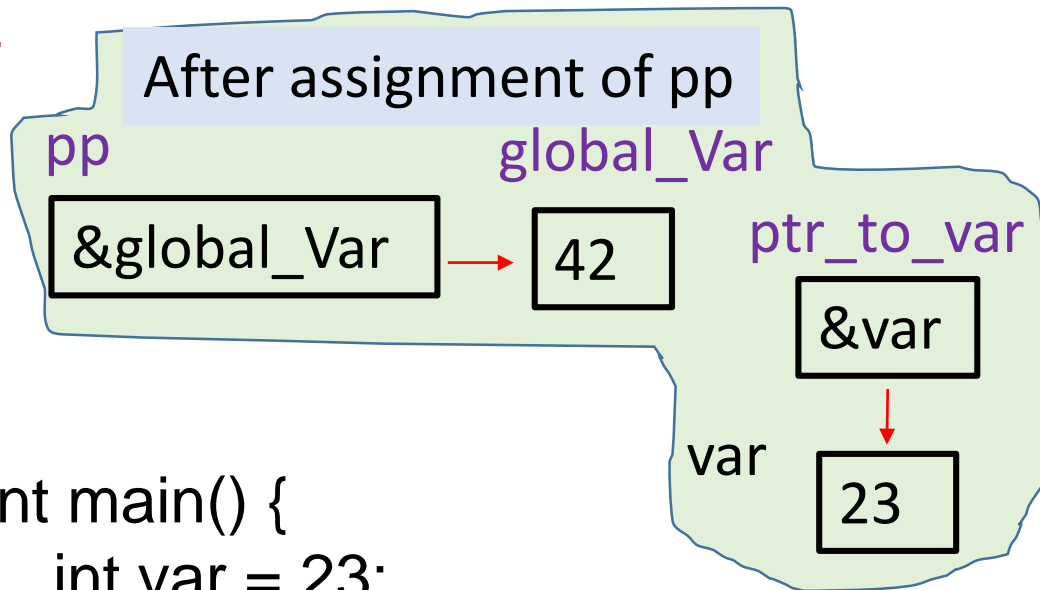
That is to say, **aPtrToPtr** is an address where stores the address of **aPtr**, and **aPtr** is an address where stores the address of **anInt**.

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: Not working

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



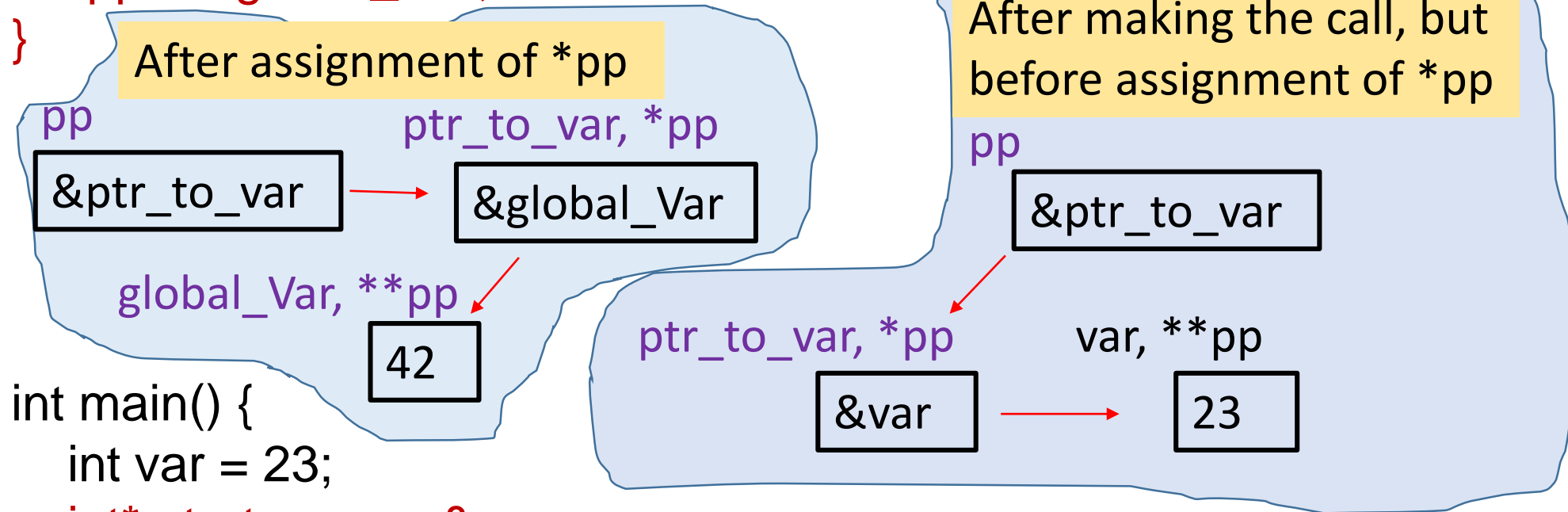
```
int main() {  
    int var = 23;  
    int *ptr_to_var = &var;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr_to_var << endl; // display 23  
    changePointerValue(ptr_to_var);  
    cout << "After :" << *ptr_to_var << endl; // display 23  
}
```

We Intend to modify the pointer stored in ptr_to_var from &var to &global_Var.

Example for Modifying Pointer Pointing to a Pointer: 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`.

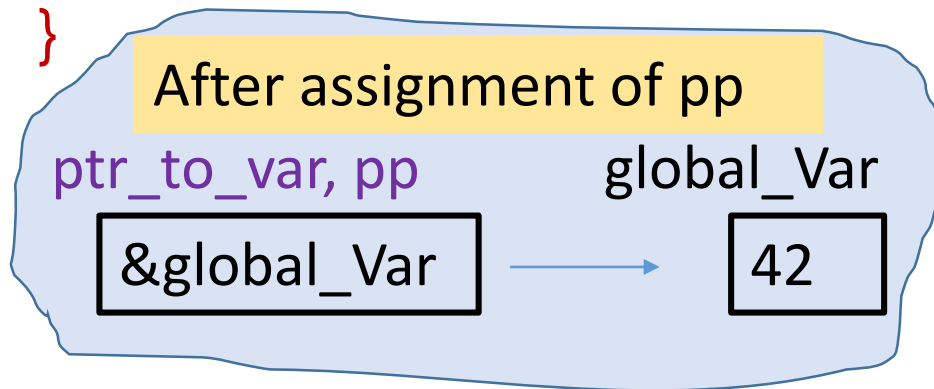


```
int main() {  
    int var = 23;  
    int* ptr_to_var = &var;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr_to_var << endl; // display 23  
    changePointerValue( &ptr_to_var );  
    cout << "After :" << *ptr_to_var << endl; // display 42  
}
```

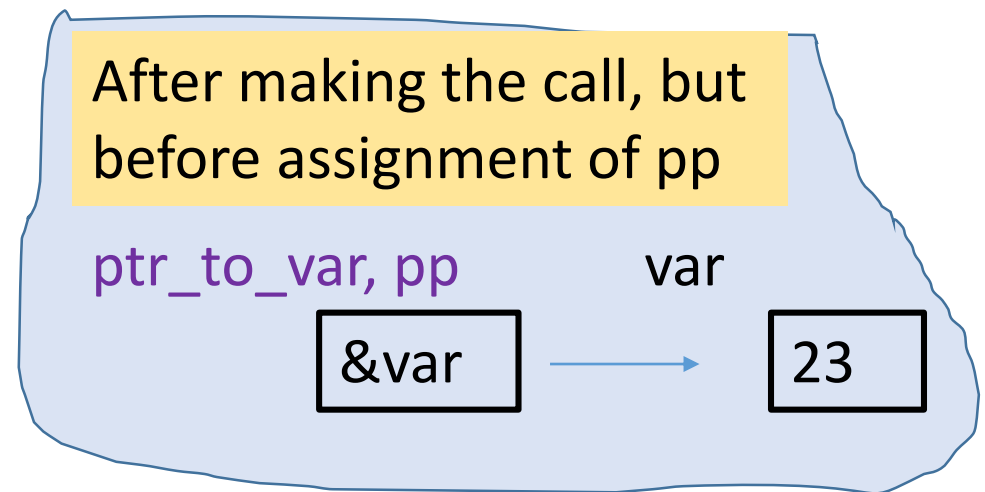
Example for Modifying a Reference to a Pointer : Working Fine

```
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;  
}
```

This also 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  
}
```



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)**)

{

}

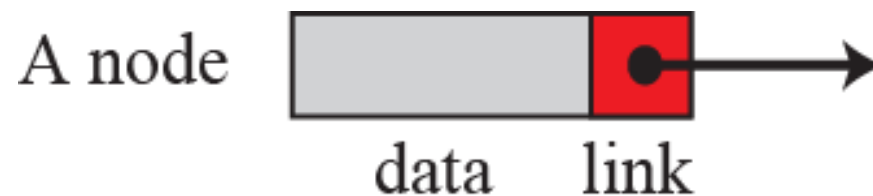
LAB 1: Manipulating a Linked-List

- This lab has two parts. In the first part, you will write a program to create a sorted linked-list. There should be no duplicate numbers in the linked-list. The numbers are sorted in the descending order in the linked-list. However, they are unsorted originally in a file. The file name should be obtained from keyboard. The file consists of integers separated by spaces or newlines. The number of integers in the file is unknown. After creating the linked list, you have to print out the linked-list.
- In the second part, you have to delete some elements from the linked-list. The data to be deleted are first read from a file. The file name should be obtained from keyboard. If an integer read from the file can be found in the linked-list, the node that stores the integer should be deleted. If it is not found, nothing has to be done. The number of integers being checked for deletion in the file is unknown. After deleting all the integers found in the linked-list, display the numbers that remain in the linked-list.

Use of **struct** for Defining 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.

Hints

- To work out this lab, you can refer to the pseudo code in Chapter 11, **Foundations of Computer Science by Behrouz Forouzan**, 4th Edition. However, for your convenience, the pseudo code for **SearchLinkedList**, **InsertLinkedList**, and **DeleteLinkedList** is presented at the end of this slide set.
- You can use pointers pointing 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. This will be given first.
- You can also use references to a pointer to solve the problems. This will be given next. If you understand how to use references to a pointer, this approach will be a simpler way.

Using Functions with parameters of Pointers Pointing to Pointers (1)

- **void createList(ifstream &inFile, NODE **head);**
 - Read data from the input file and create a linked list. The data must maintain in the descending order.
- **void deleteElements(ifstream &inFileDel, NODE **head);**
 - Read data being checked for deletion from a file. The data item is read one by one from a file until end of file. If a data item can be found in the linked list, the node storing this data item should be removed.
- **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** are each a pointer pointing to a pointer. The reason for this is that cuPtr and prePtr may be updated in this function.

Using Functions with parameters of Pointers Pointing 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.

Main() Function if Functions with parameters of Pointers Pointing to Pointers are used

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

```
int main() {
    string dataFilename;
    ifstream inFile;
    string delDataFilename;
    ifstream inFileDel;
    // First part
    cout << "File name for creating linked list: ";
    cin >> dataFilename;
    cout << endl;
    inFile.open(dataFilename);
    NODE *listHead=NULL;
    if(!inFile){
        cout << "File open error!" << endl;
        exit(1);
    }
    createList(inFile, &listHead);
    inFile.close();
    displayList(listHead);
```

```
    // Second part
    cout << "\nFile name for linked list deletion: ";
    cin >> delDataFilename;
    inFileDel.open(delDataFilename);
    if(!inFileDel){
        cout << "File open error!" << endl;
        exit(1);
    }

    cout << "\nList after deletion:" << endl;
    deleteElements(inFileDel, &listHead);
    displayList(listHead);
    inFileDel.close();
    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 allocate 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;
    else
        (*prePtr)->link = (*curPtr)->link;

    (*curPtr)->link = NULL;
    delete *curPtr;
}
```

Function for Displaying a Linked-List

```
void displayList(NODE *head)
{
    int count=0; // Used for counting the number of nodes
    NODE *tempPtr = head; // Set tempPtr to the head of the linked-list

    while(tempPtr != NULL) // Check whether get to the end of the linked-list
    {
        cout << tempPtr->data << " "; // Print out the data stored in the node
        tempPtr = tempPtr->link; // Move to the next node
        count++;
        if(count%15 == 0) // Change line if already printing 15 numbers
            cout << endl;
    }
    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(ifstream &, NODE **);
void deleteElements(ifstream &, NODE **);
bool searchLinkedList(NODE *, int, NODE **, NODE **);
void insertLinkedList(NODE **, int);
OR
void createList(ifstream &, NODE *&);
void deleteElements(ifstream &, NODE *&);
bool searchLinkedList(NODE *, int, NODE *&, NODE *&);
void insertLinkedList(NODE *&, int);
- The output should be correct.

Input Example 1

◆ The content of a file for creating a linked list:

120 30 50 80 -2 90 90 88 100 55 75

300 100 -65 76 1000 -5 0 83 2001

◆ The content of a file for deleting elements in the linked list:

120 30 80 -2 90 90 55 75 0

300 -65 76 1000 -5 0 83 2001

Output Example 1

```
File name for creating linked list: rawData.txt
2001 1000 300 120 100 90 88 83 80 76 75 55 50 30 0
-2 -5 -65
Total number of elements in the linked list: 18
File name for linked list deletion: manipData.txt
List after deletion:
100 88 50
Total number of elements in the linked list: 3
```

Input Example 2

◆The content of a file for creating a linked list:

43 120 30 50 80 -2 90 90 88 100 1 2 -100 -49 76 99 100008 75 66
234 300 100 -65 76 1000 -5 0 -9 -165 -1876 654 97235 450 888 943
1034 385 87 6524 927 -609 -8959 0 89 176 548 928 912 7236 2139
1298 21909 2183 39 19 -127 -18 -1293 -278 -398 1932 9398 -218
1890 99 66

◆The content of a file for deleting elements in the linked list:

43 120 30 50 80 -2 90 90 100 1 2 -100 -49 76 99 100008 75 234
300 100 -65 76 1000 -5 0 -9 -165 -1876 654 97235 450 943 1034
385 87 6524 -609 -8959 0 89 176 548 928 912 7236 2139
1298 21909 2183 19 -127 -18 -1293 -278 -398 1932 9398 -218
1890 99 66

Output Example 2

```
File name for creating linked list: rawData2.txt
```

```
100008 97235 21909 9398 7236 6524 2183 2139 1932 1890 1298 1034 1000 943 928  
927 912 888 654 548 450 385 300 234 176 120 100 99 90 89  
88 87 80 76 75 66 50 43 39 30 19 2 1 0 -2  
-5 -9 -18 -49 -65 -100 -127 -165 -218 -278 -398 -609 -1293 -1876 -8959
```

```
Total number of elements in the linked list: 60
```

```
File name for linked list deletion: manipData2.txt
```

```
List after deletion:
```

```
927 888 88 39
```

```
Total number of elements in the linked list: 4
```

Using Functions with Parameters of Pointers

Pointing to References of Pointers (1)

- **void createList(ifstream &inFile, NODE *&head);**
 - Read data from the input file and create a linked list. The data must maintain in the descending order.
- **void deleteElements(ifstream &inFileDel, NODE *&head);**
 - Read data being checked for deletion from a file. The data item is read one by one from a file until end of file. If a data item can be found in the linked list, the node storing this data item should be removed.
- **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 should set up two pointers **curPtr** and **prePtr** which will be employed to insert the data element into the linked-list. Note that **curPtr** and **prePtr** are each a pointer pointing to a reference of a pointer. The reason for this is that **cuPtr** and **prePtr** may be updated in this function.

Using Functions with Parameters of Pointers

Pointing to References of 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 is the same as the one given before.

Main() Function if Functions with parameters of Pointers Pointing to References of Pointers are used

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

```
int main() {
    string dataFilename;
    ifstream inFile;
    string delDataFilename;
    ifstream inFileDel;
    // First part
    cout << "File name for creating linked list: ";
    cin >> dataFilename;
    cout << endl;
    inFile.open(dataFilename);
    NODE *listHead=NULL;
    if(!inFile){
        cout << "File open error!" << endl;
        exit(1);
    }
    createList(inFile, listHead);
    inFile.close();
    displayList(listHead);

    // Second part
    cout << "\nFile name for linked list deletion: ";
    cin >> delDataFilename;
    inFileDel.open(delDataFilename);
    if(!inFileDel){
        cout << "File open error!" << endl;
        exit(1);
    }

    cout << "\nList after deletion:" << endl;
    deleteElements(inFileDel, listHead);
    displayList(listHead);
    inFileDel.close();
    return 0;
}
```

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;

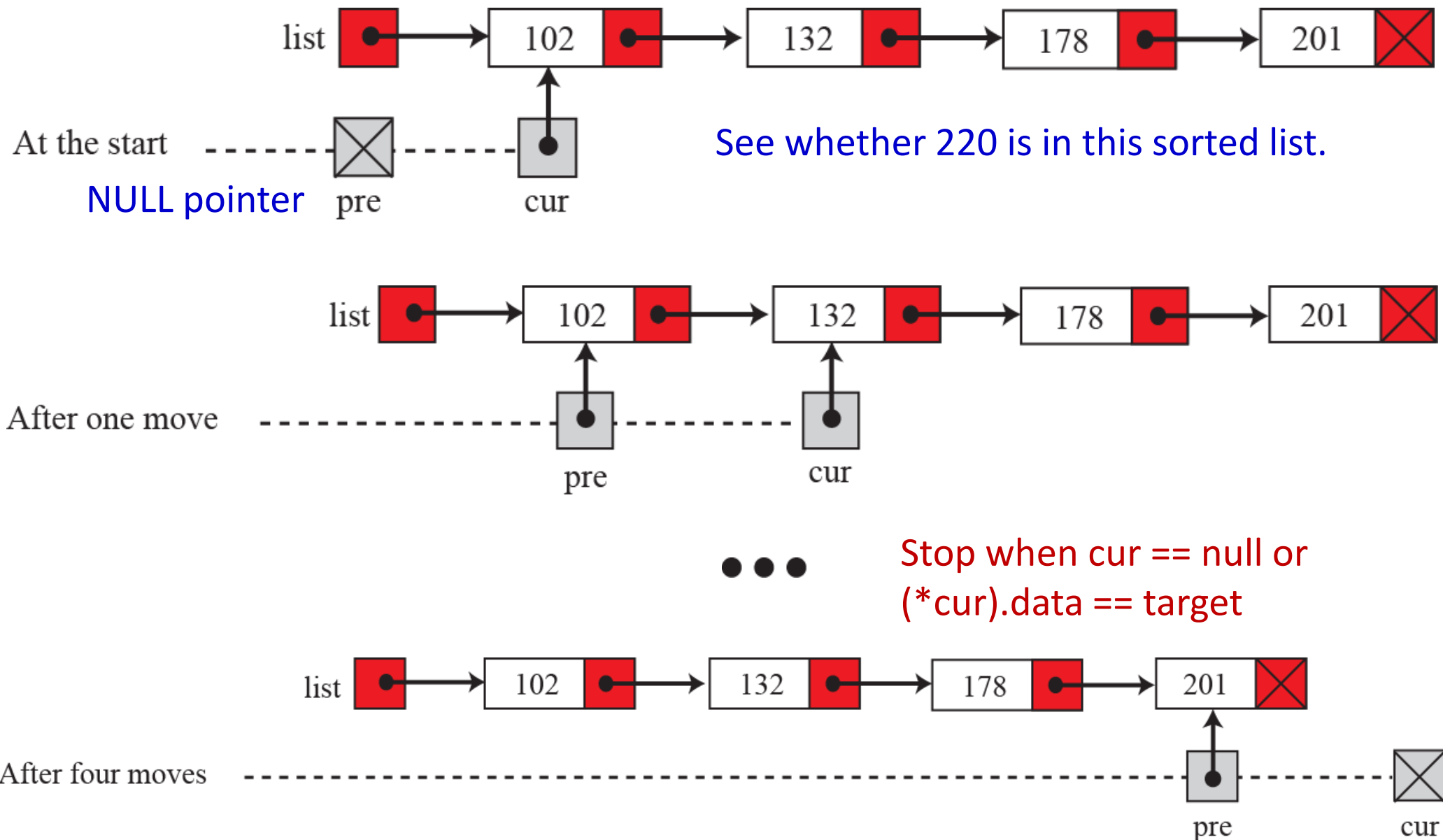
    if(prePtr == NULL)
        listHead = curPtr->link;
    else
        prePtr->link = curPtr->link;

    curPtr->link = NULL;
    delete curPtr;
}
```


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.

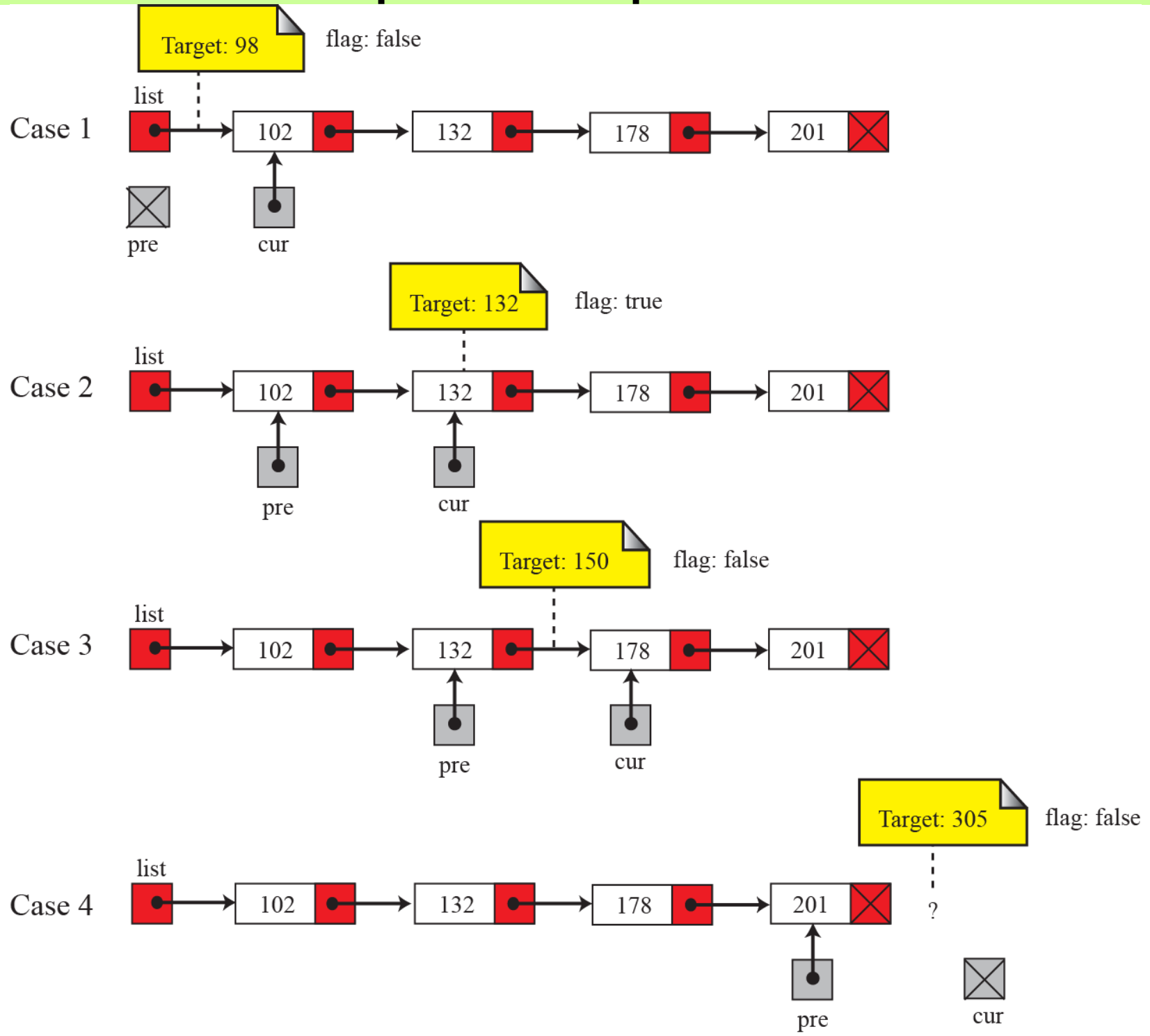
Figure 11.13: Moving of pre and cur pointers for searching



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.

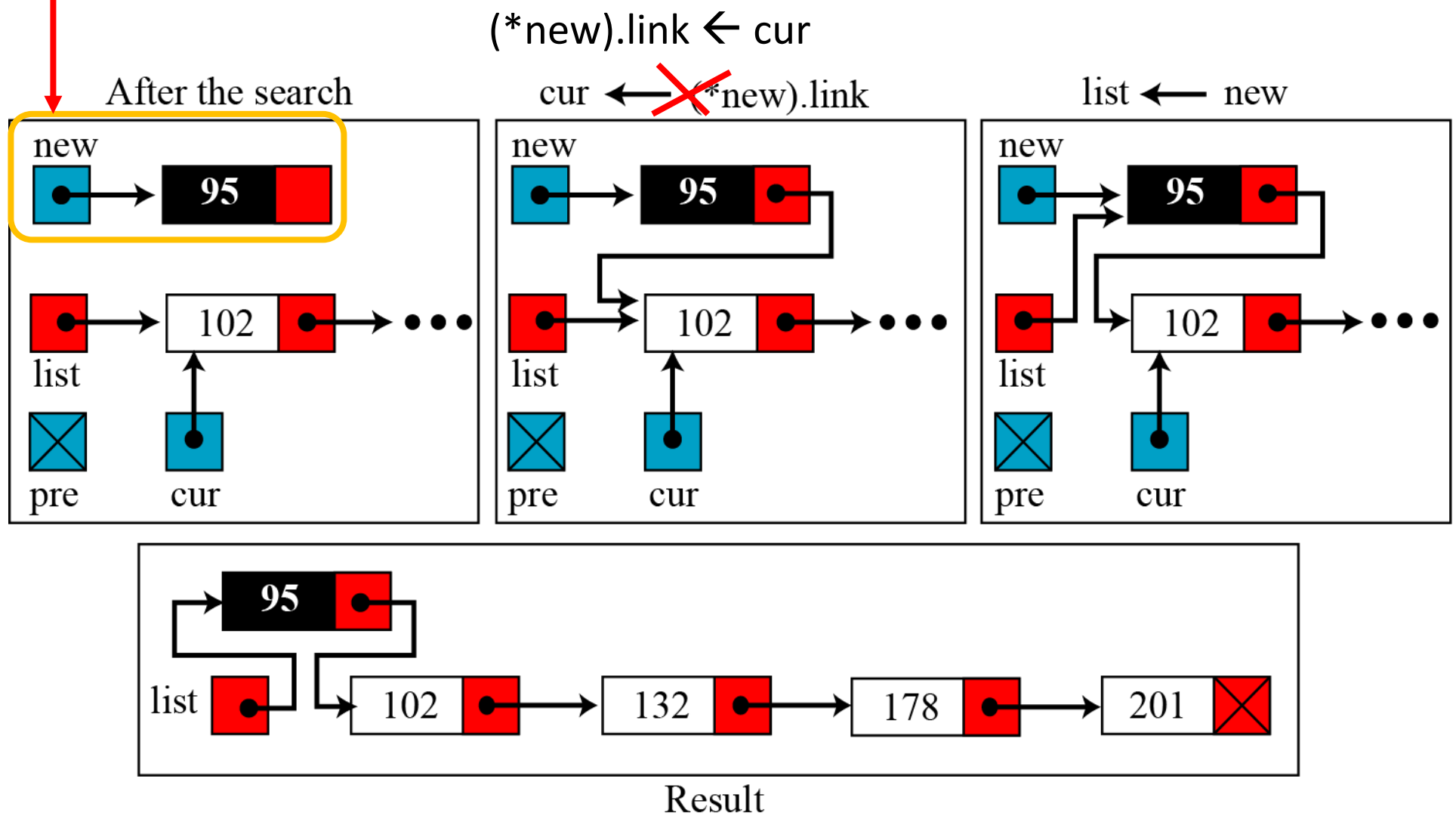


Figure 11.16 Inserting a node at the end of the linked list

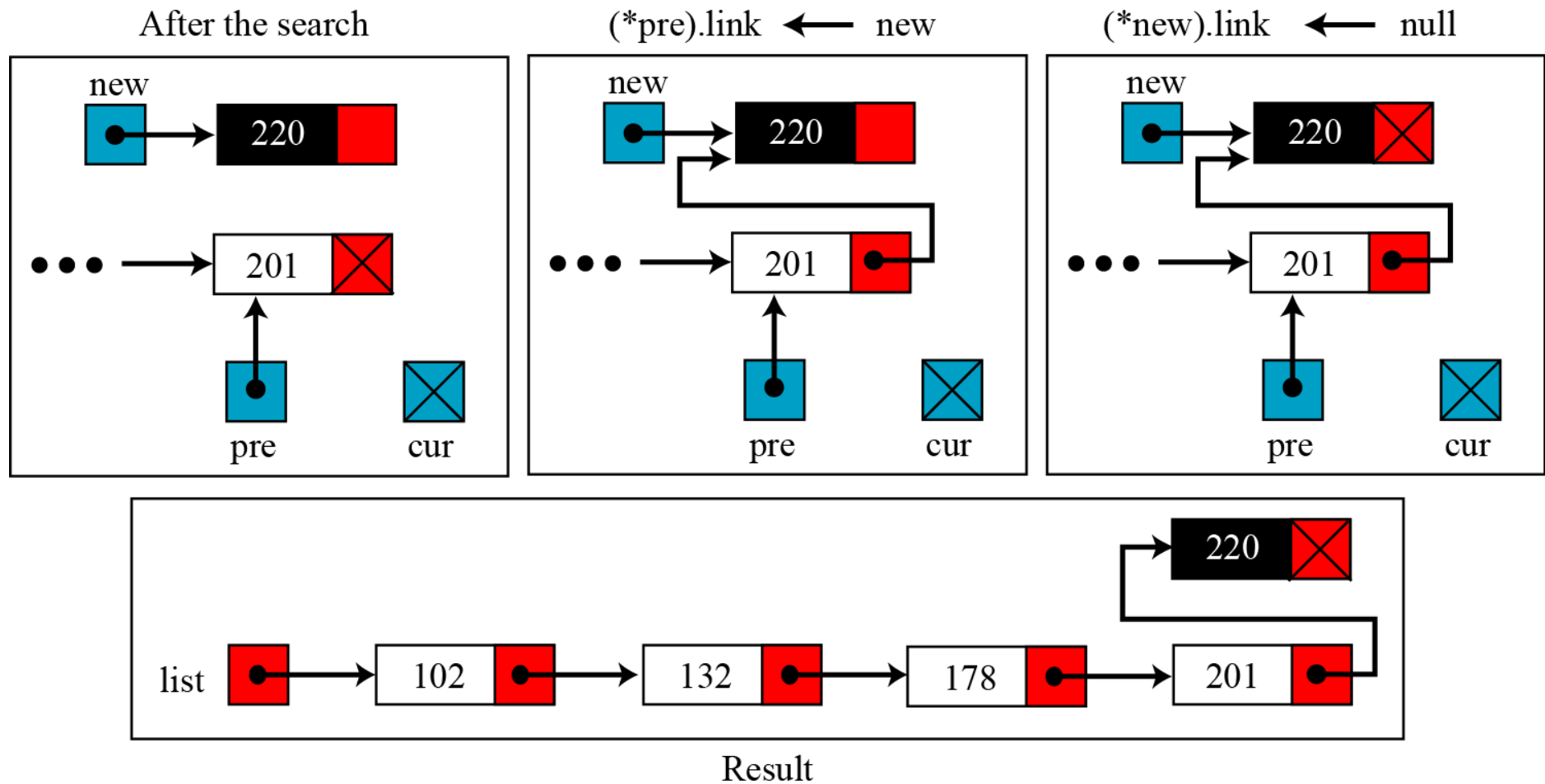
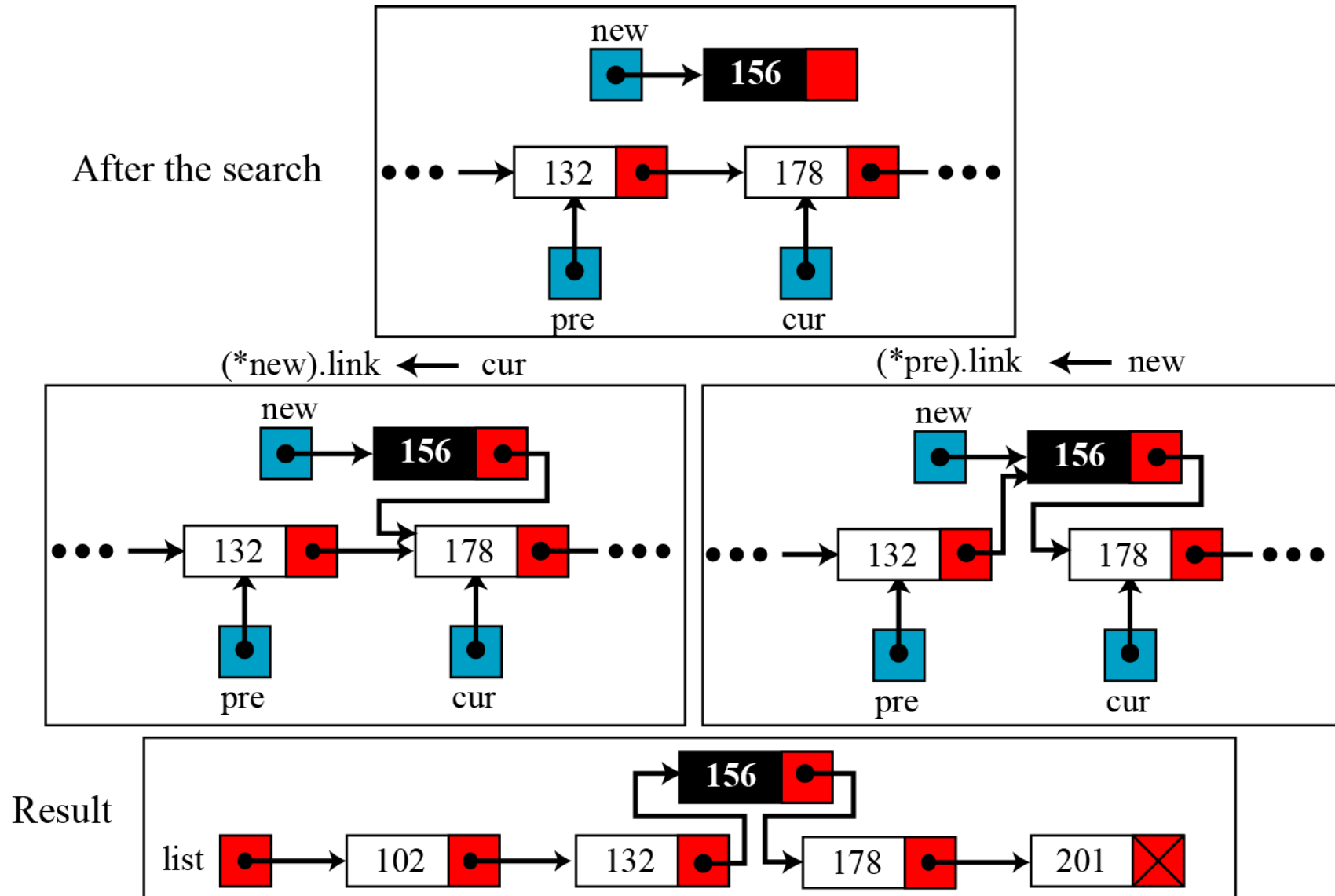


Figure 11.17 Inserting a node in the middle of the linked list



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

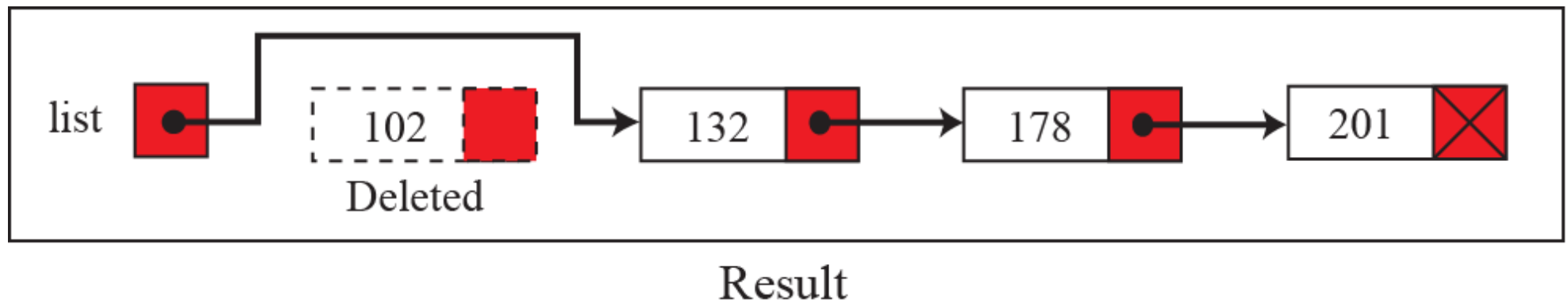
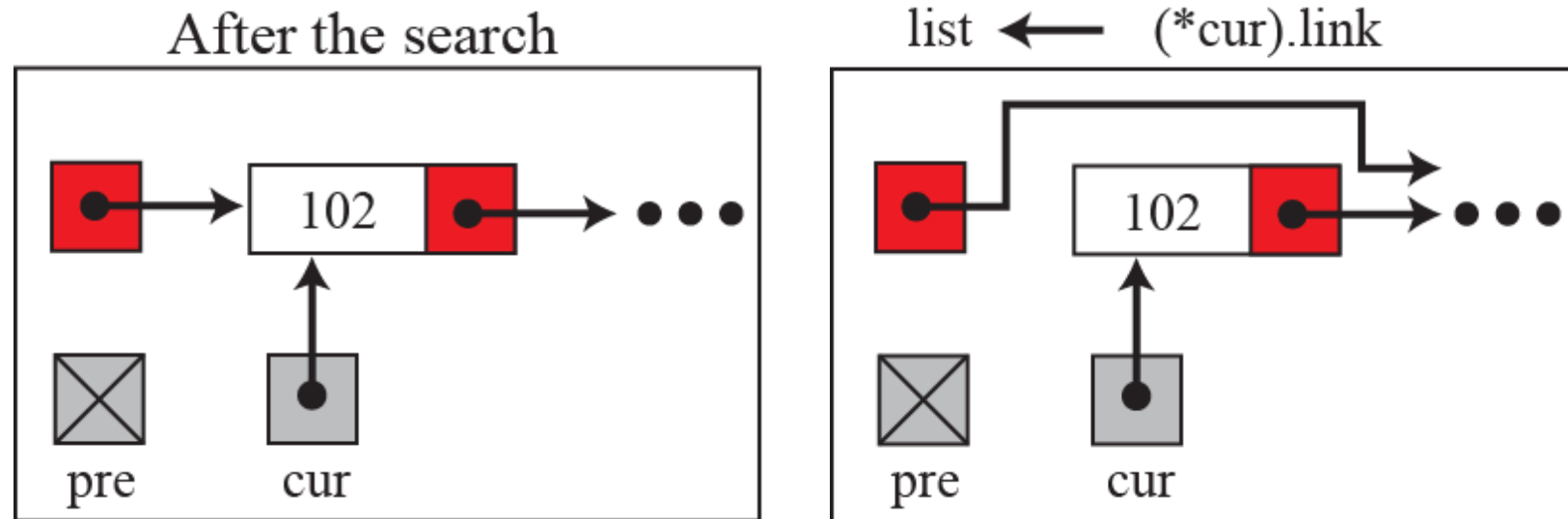
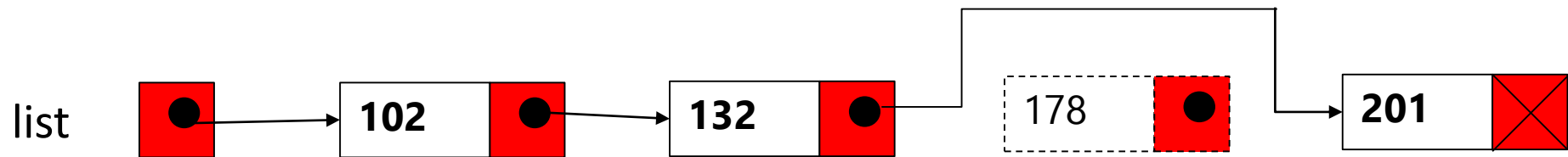
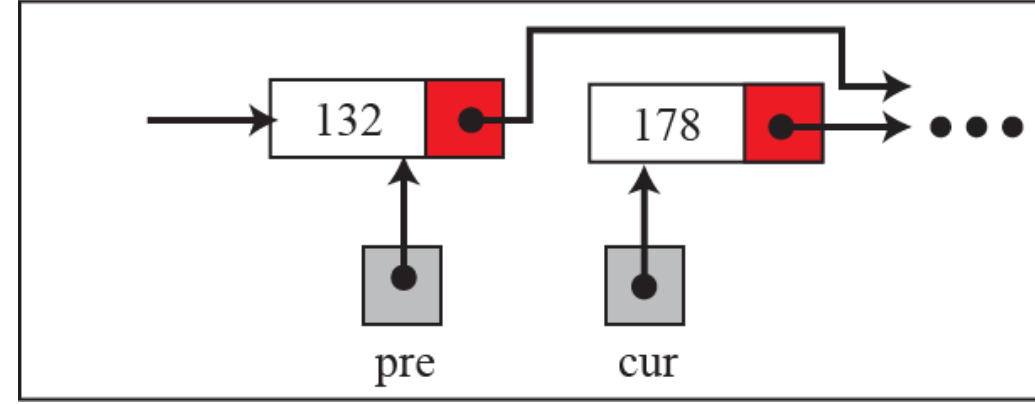
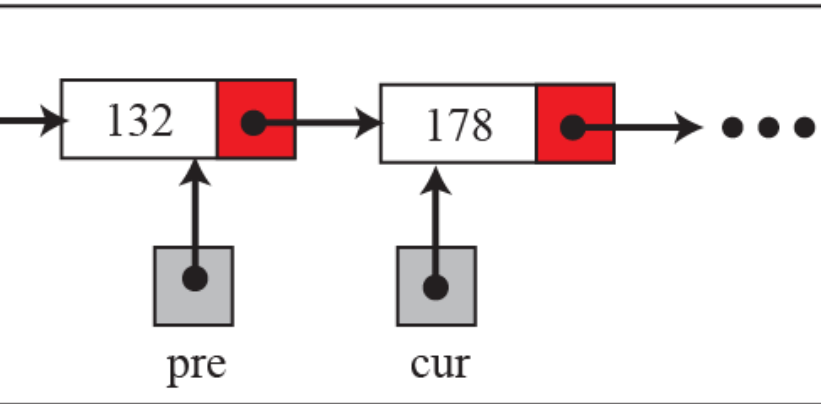


Figure 11.19: Deleting a node at the middle or end

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
}
```