DATA STRUCTURES USING ‘C’

Lecture-8

Data Structures

Lecture Objectives

After studying this chapter, the student should be able to:

❑Define a record as a data structure and how it is used to store attributes
belonging to a single data element.

❑ Distinguish between the name of a record and the names of its fields.

❑ Define a linked list as a data structure and how it is implemented using
pointers.

❑ Understand the mechanism through which the nodes in an array are
accessed.

❑ Describe operations defined for a linked list.

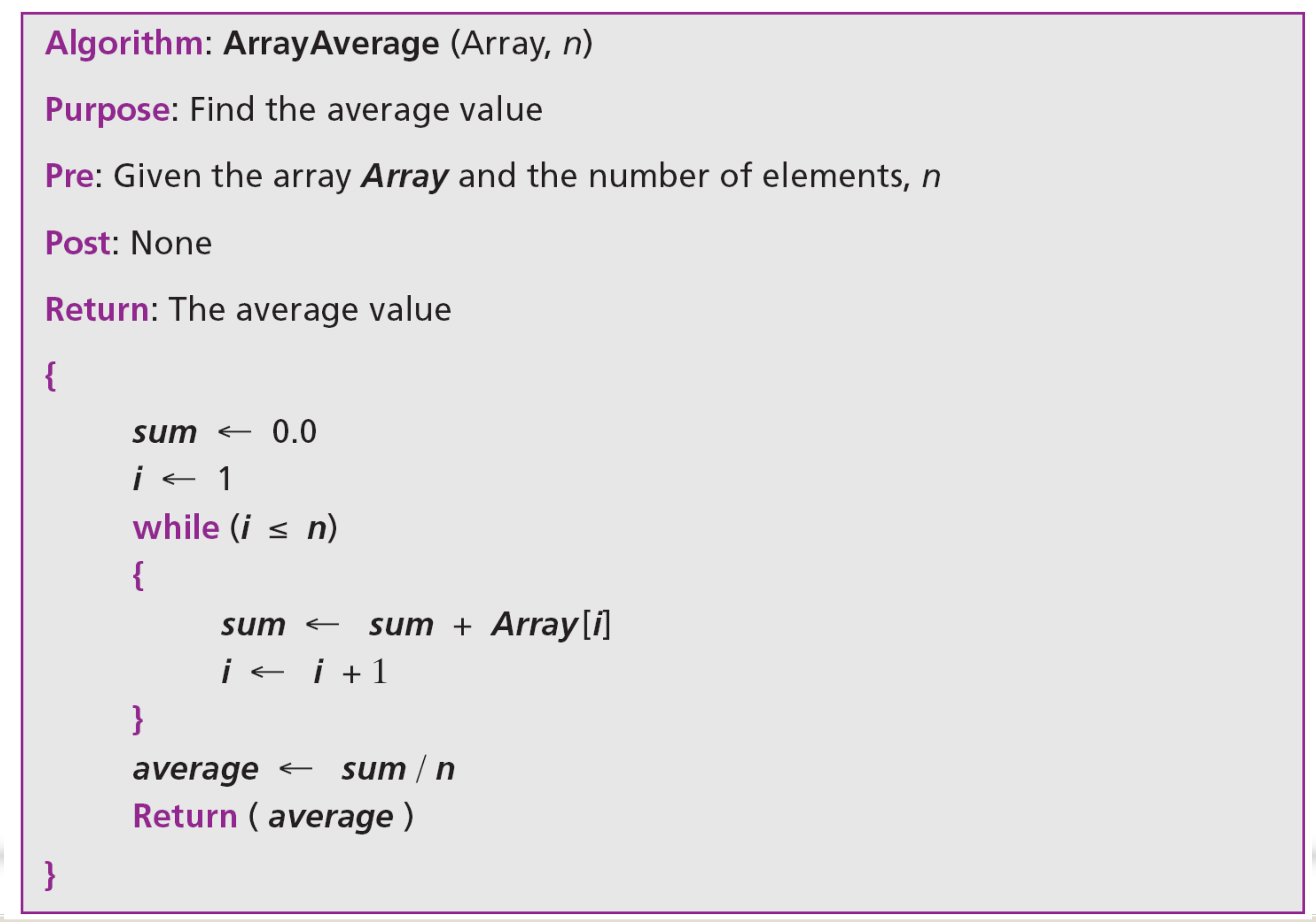
❑ Compare and contrast arrays, records, and linked lists

Operations on array

Although we can apply conventional operations defined for each element of an array (see Chapter 4), there are some operations that we can define on an array as a data structure. The common operations on arrays as structures are **searching**, **insertion**, **deletion**, **retrieval** and **traversal**.

Although searching, retrieval and traversal of an array is an easy job, insertion and deletion is time consuming. The elements need to be shifted down before insertion and shifted up after deletion.

Algorithm 11.1 gives an example of finding the average of elements in array whose elements are reals.



```
Algorithm: ArrayAverage (Array, n)
Purpose: Find the average value
Pre: Given the array Array and the number of elements, n
Post: None
Return: The average value
{
    sum ← 0.0
    i ← 1
    while (i ≤ n)
    {
        sum ← sum + Array[i]
        i ← i + 1
    }
    average ← sum / n
    Return ( average )
}
```

Application

Thinking about the operations discussed in the previous section gives a clue to the application of arrays. If we have a list in which a lot of insertions and deletions are expected after the original list has been created, we should not use an array. An array is more suitable when the number of deletions and insertions is small, but a lot of searching and retrieval activities are expected.

i

An array is a suitable structure when a small number of insertions and deletions are required, but a lot of searching and retrieval is needed.

11-2 RECORDS

A record is a collection of related elements, possibly of different types, having a single name. Each element in a record is called a **field**. A field is the smallest element of named data that has meaning. A field has a type and exists in memory. Fields can be assigned values, which in turn can be accessed for selection or manipulation. A field differs from a variable primarily in that it is part of a record.

Figure 11.7 contains two examples of records. The first example, fraction, has two fields, both of which are integers. The second example, student, has three fields made up of three different types.

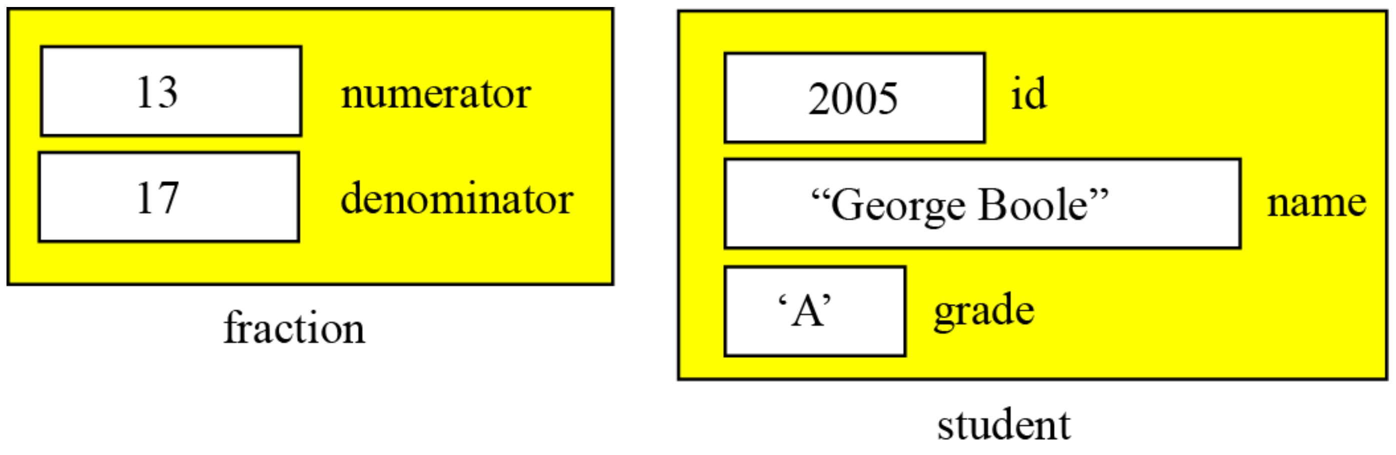


Figure 11.7 Records

Record name *versus* field name

Just like in an array, we have two types of identifier in a record: the name of the record and the name of each individual field inside the record. The name of the record is the name of the whole structure, while the name of each field allows us to refer to that field. For example, in the student record of Figure 11.7, the name of the record is student, the name of the fields are **student.id**, **student.name** and **student.grade**. Most programming languages use a ***period*** (.) to separate the name of the structure (record) from the name of its components (fields). This is the convention we use in this book.

Example 11.5

The following shows how the value of fields in Figure 11.7 are stored.

```
student.id ← 2005        student.name ← "G. Boole"        student.grade  ← 'A'
```

Comparison of records and arrays

We can conceptually compare an array with a record. This helps us to understand when we should use an array and when to use a record. An array defines a combination of elements, while a record defines the identifiable parts of an element. For example, an array can define a class of students (40 students), but a record defines different attributes of a student, such as id, name or grade.

Array of records

If we need to define a combination of elements and at the same time some attributes of each element, we can use an array of records. For example, in a class of 30 students, we can have an array of 30 records, each record representing a student.

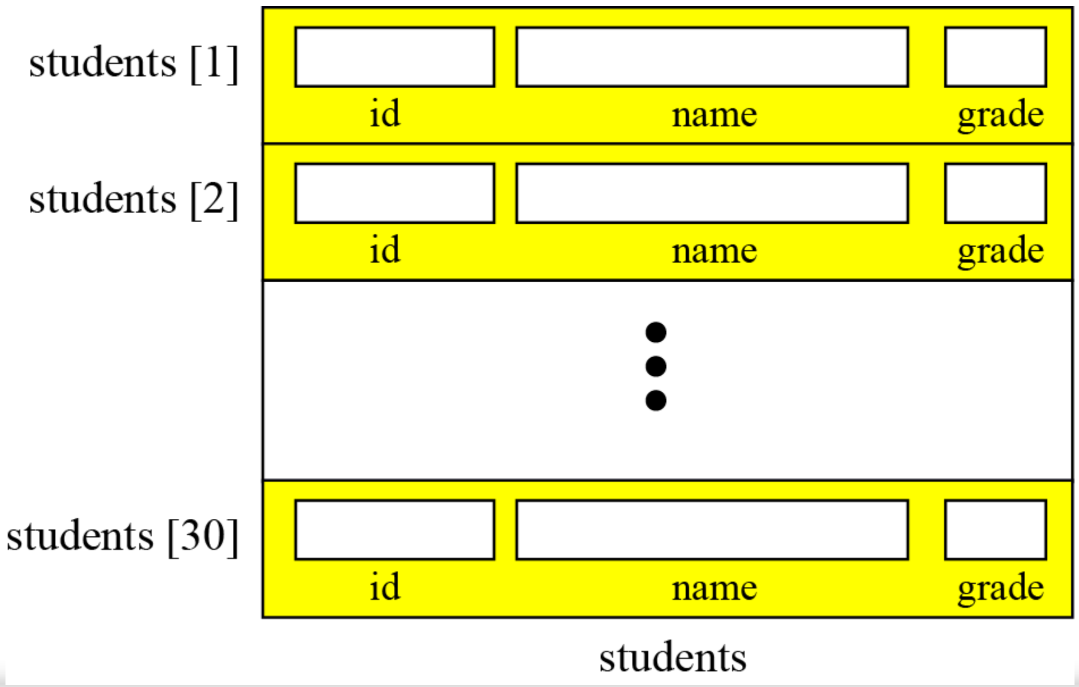
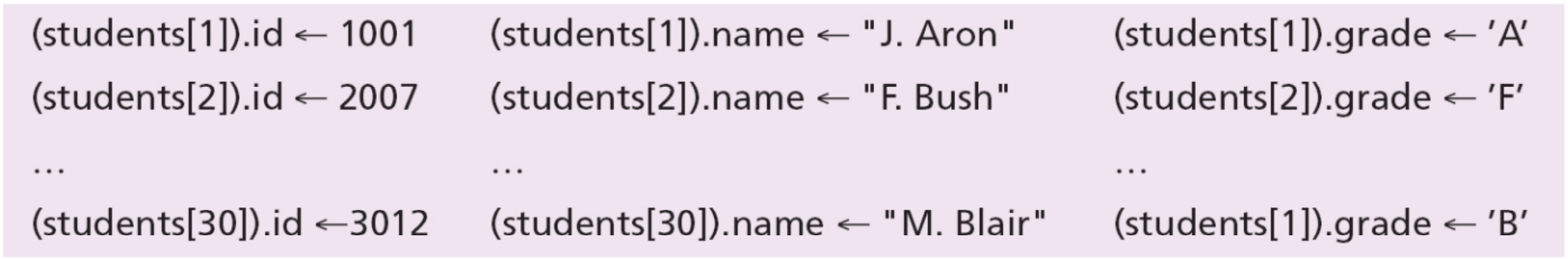


Figure 11.8 **Array of records**

Example 11.6

The following shows how we access the fields of each record in the students array to store values in them.



```
(students[1]).id ← 1001      (students[1]).name ← "J. Aron"      (students[1]).grade ← 'A'
(students[2]).id ← 2007      (students[2]).name ← "F. Bush"      (students[2]).grade ← 'F'
...                          ...                                 ...
(students[30]).id ←3012      (students[30]).name ← "M. Blair"    (students[1]).grade ← 'B'
```

Example 11.7

However, we normally use a loop to read data into an array of records. Algorithm 11.2 shows part of the pseudocode for this process.

Algorithm 11.2 Part of the pseudocode to read student records

```
i ← 1
while (i < 31)
{
        read (students [i]).id
        read (students [i]).name
        read (students [i]).grade
        i ← i + 1
}
```

Arrays versus arrays of records

Both an array and an array of records represent a list of items. An array can be thought of as a special case of an array of records in which each element is a record with only a single field.

11-3 LINKED LISTS

A linked list is a collection of data in which each element contains the location of the next element—that is, each element contains two parts: **data** and **link**. The name of the list is the same as the name of this pointer variable. Figure 11.9 shows a linked list called *scores* that contains four elements. We define an empty linked list to be only a null pointer: Figure 11.9 also shows an example of an empty linked list.

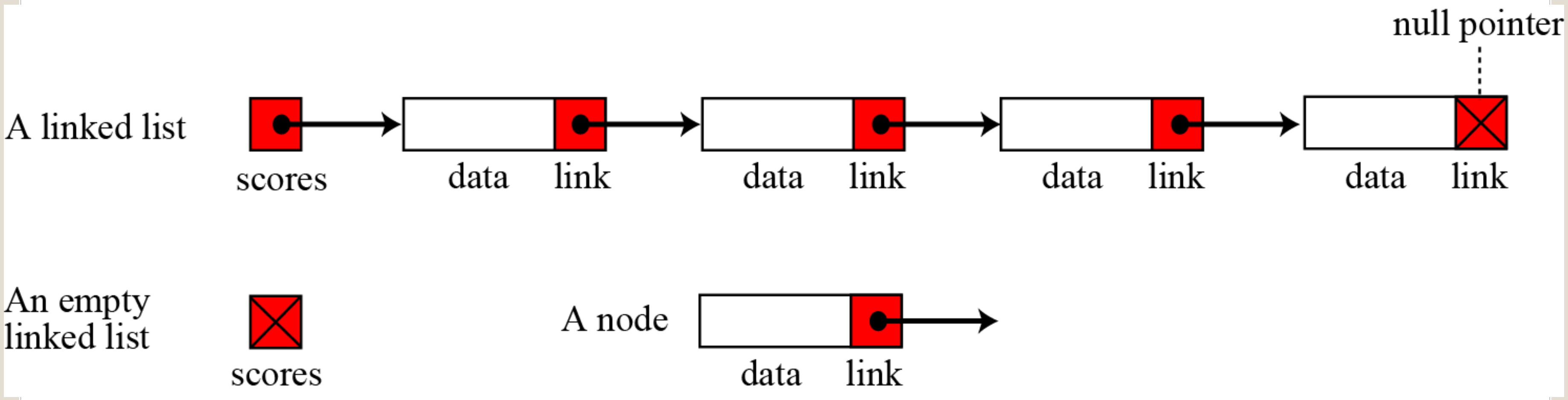


Figure 11.9 **Linked lists**

Before further discussion of linked lists, we need to explain the notation we use in the figures. We show the connection between two nodes using a line. One end of the line has an arrowhead, the other end has a solid circle.

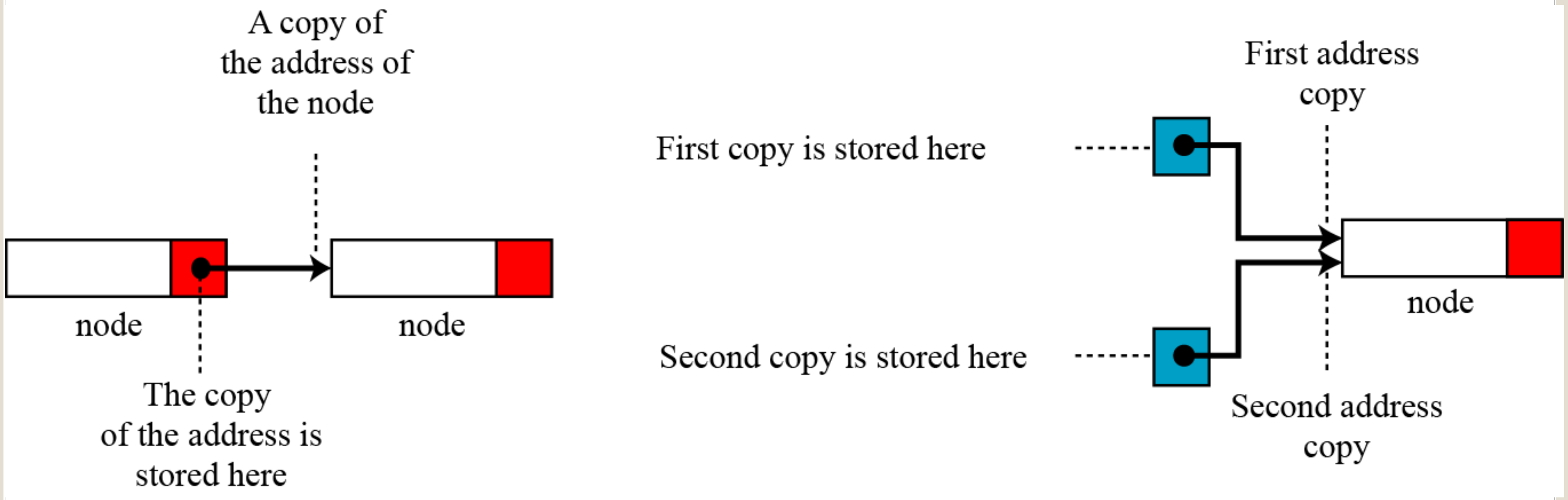

A copy of the address of the node
node
node
The copy of the address is stored here
First copy is stored here
First address copy
node
Second copy is stored here
Second address copy

Figure 11.10 **The concept of copying and storing pointers**

Arrays versus linked lists

Both an array and a linked list are representations of a list of items in memory. The only difference is the way in which the items are linked together. Figure 11.11 compares the two representations for a list of five integers.

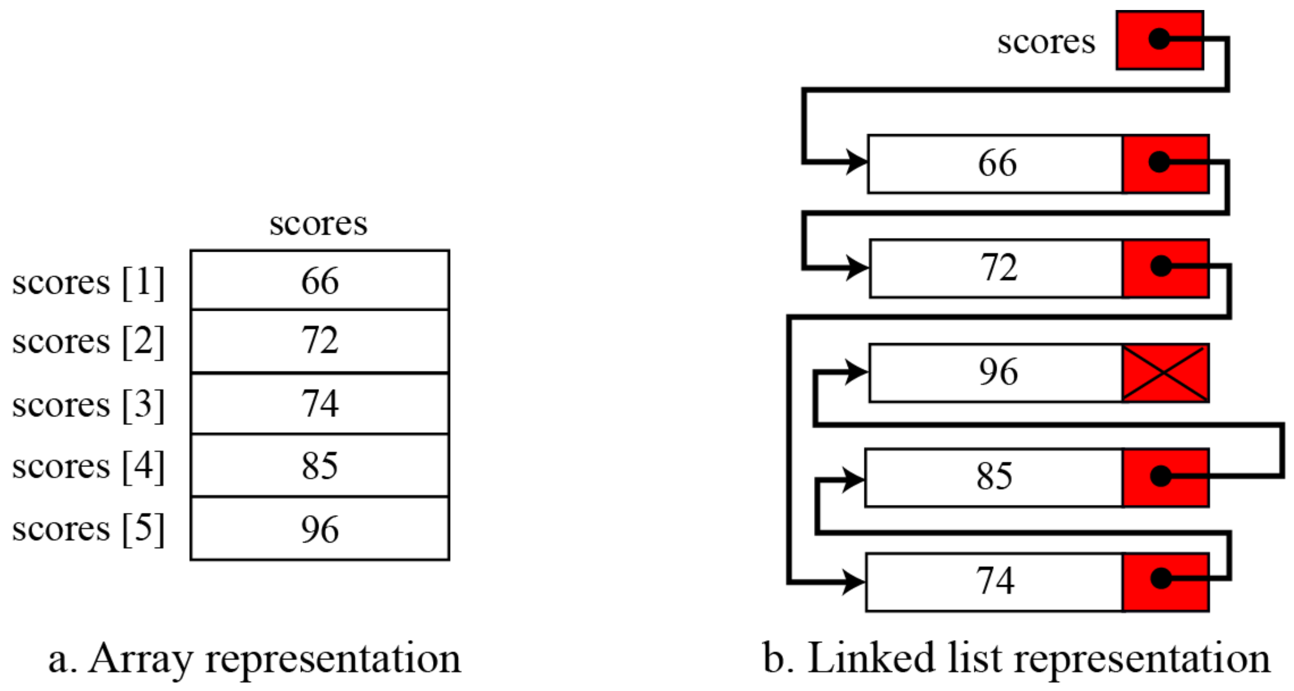


Figure 11.11 **Array versus linked list**

Linked list names versus nodes names

As for arrays and records, we need to distinguish between the name of the linked list and the names of the nodes, the elements of a linked list. A linked list must have a name. The name of a linked list is the name of the head pointer that points to the first node of the list. Nodes, on the other hand, do not have an explicit names in a linked list, just implicit ones.

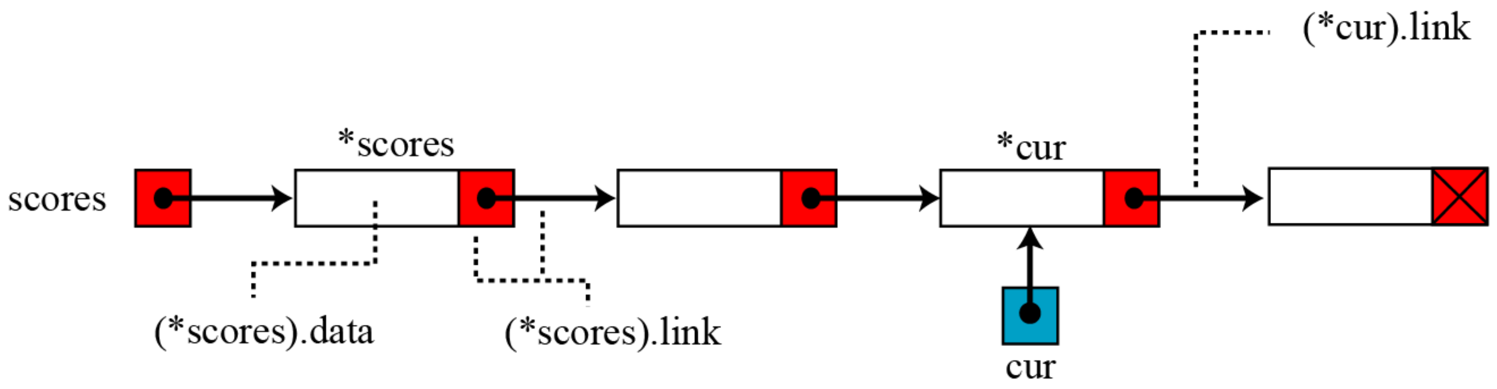


Figure 11.12 **The name of a linked list versus the names of nodes**

Operations on linked lists

The same operations we defined for an array can be applied to a linked list.

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.

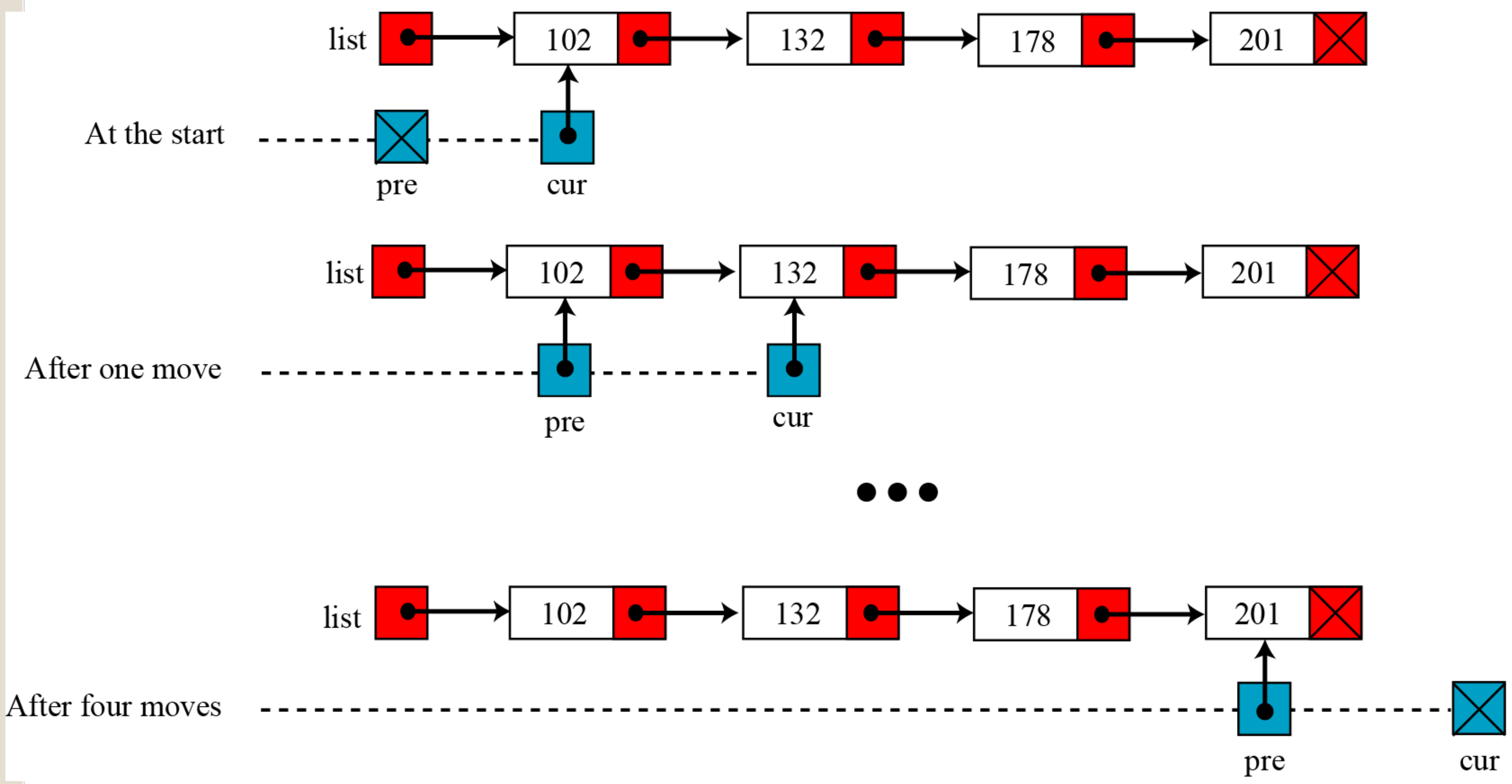


Figure 11.13 **Moving of *pre* and *cur* pointers in searching a linked list**

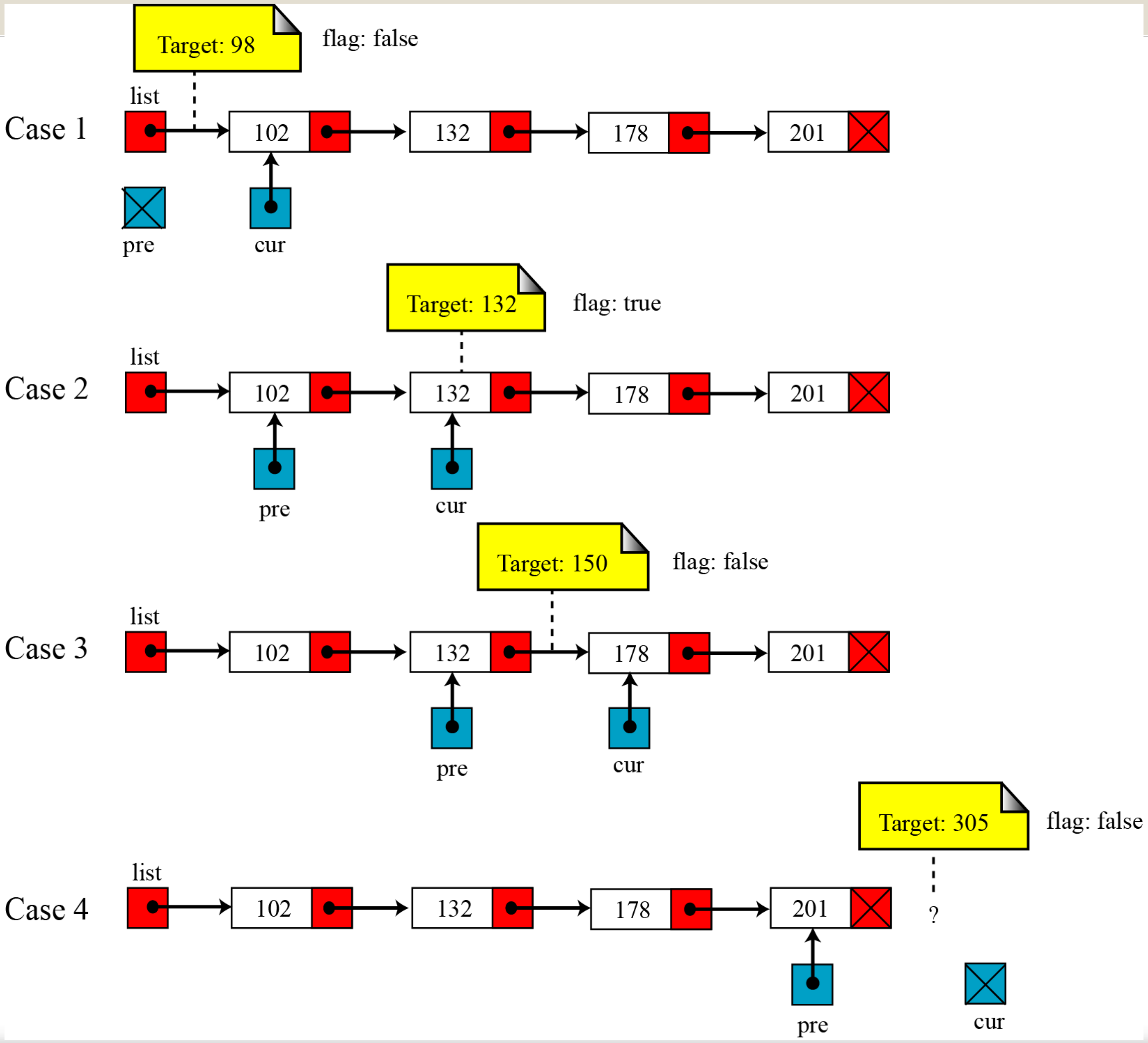


Figure 11.14 **Values of *pre* and *cur* pointers in different cases**

Algorithm 11.3 Searching a linked list

Algorithm: **SearchLinkedList** (list, target)

Purpose: Search the list using two pointers: **pre** and **cur**

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

Post: None

Return: The position of **pre** and **cur** pointers and the value of the flag (*true* or *false*)

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

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.

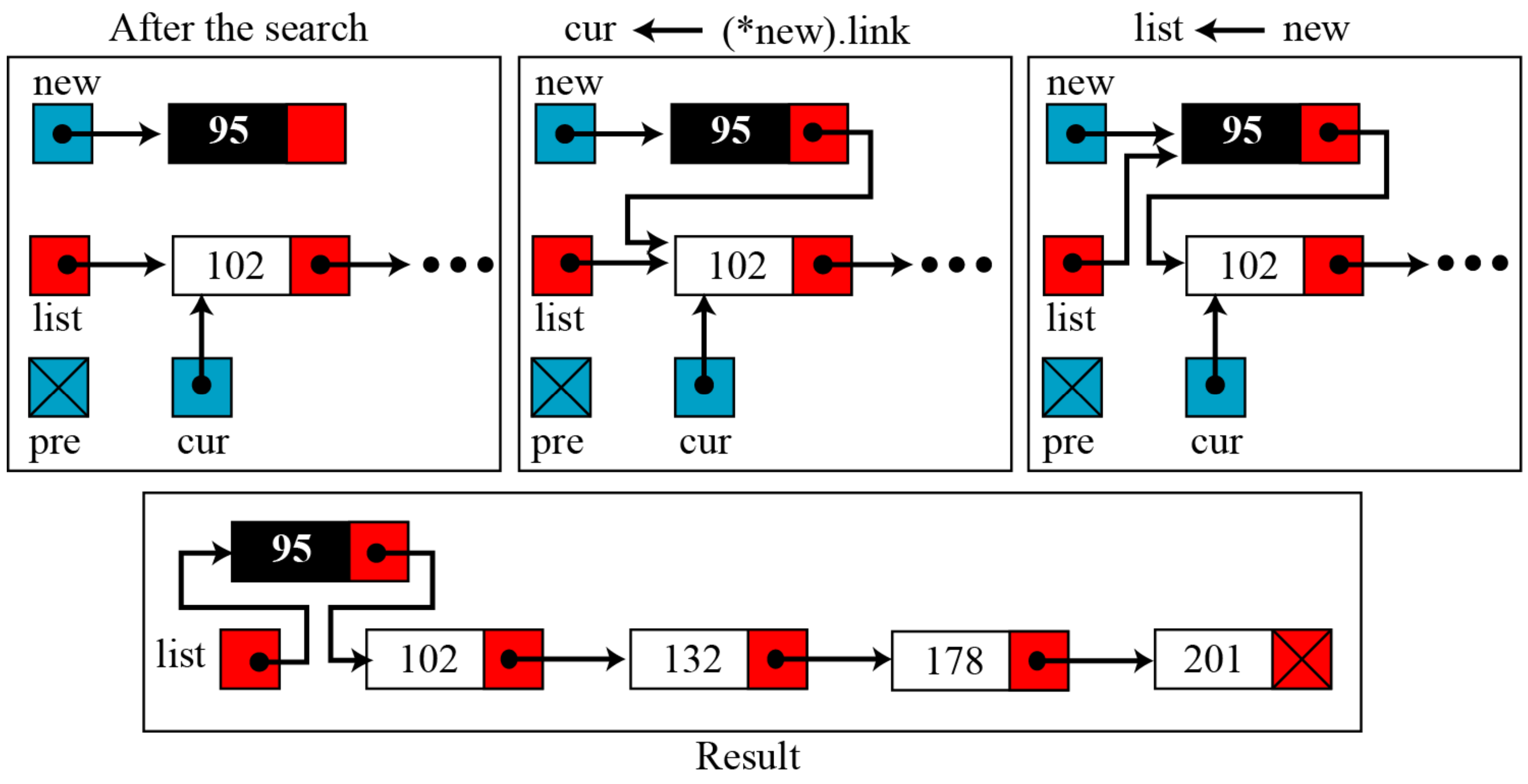


Figure 11.15 **Inserting a node at the beginning of a linked list**

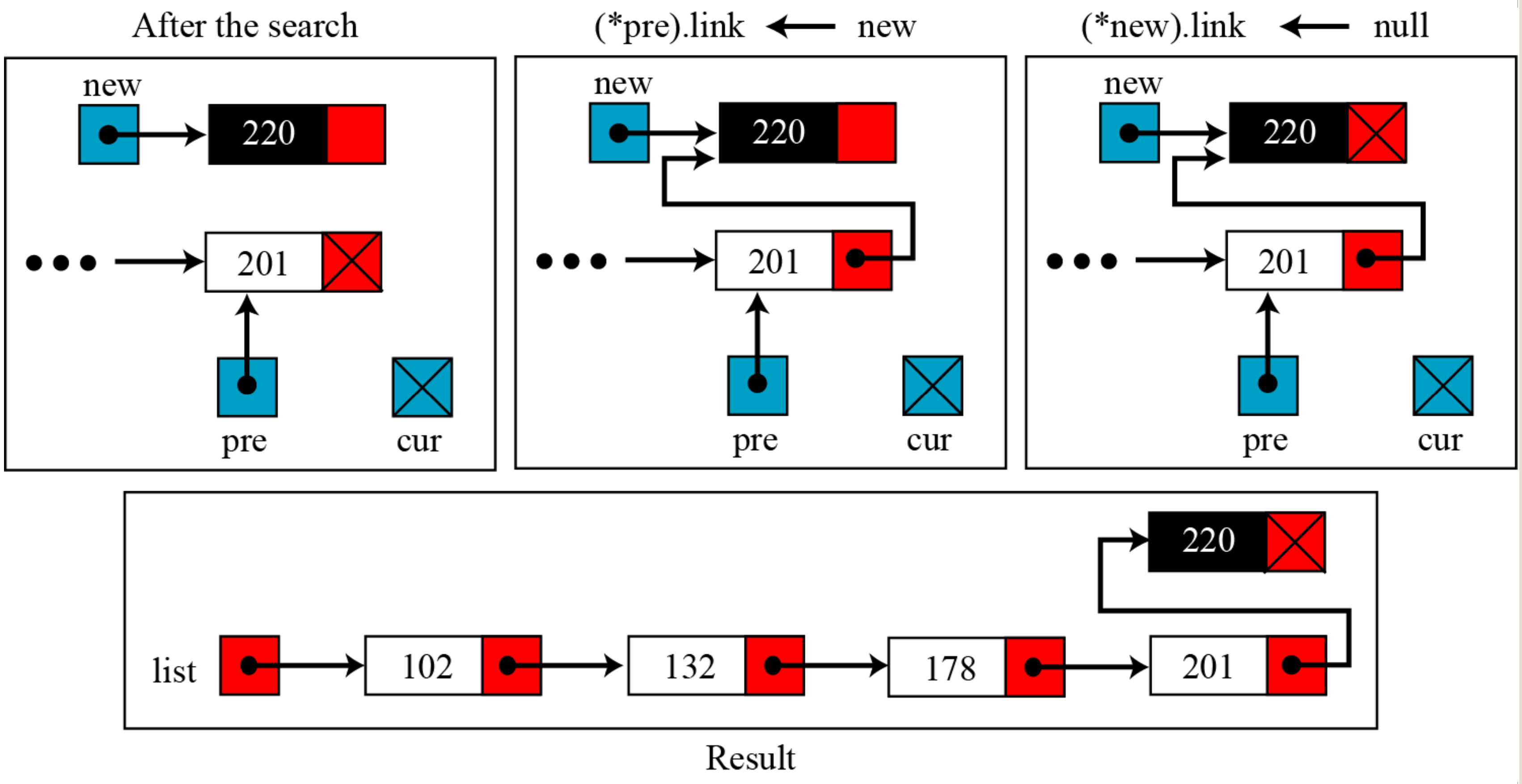


Figure 11.16 **Inserting a node at the end of the linked list**

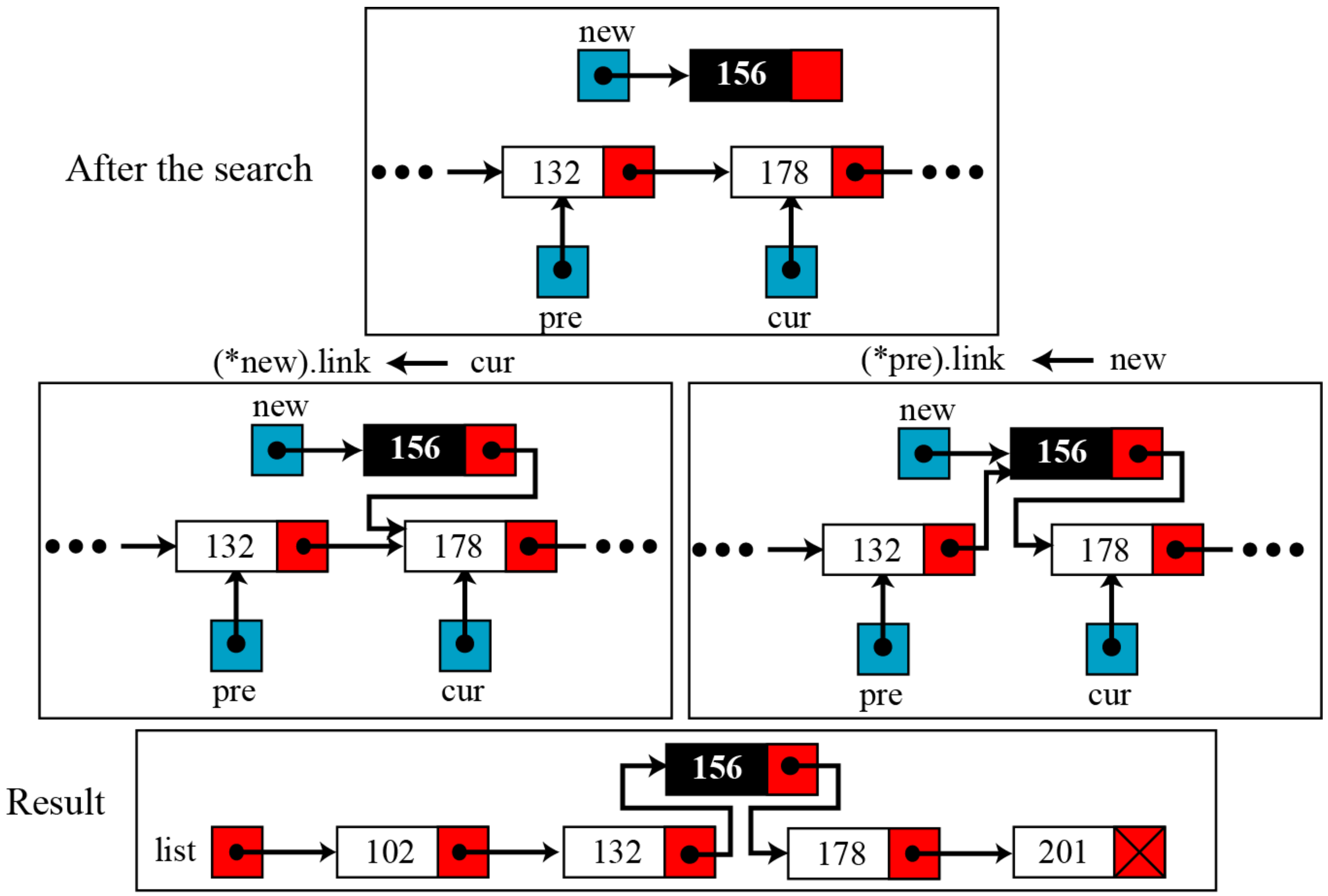


Figure 11.17 **Inserting a node in the middle of the linked list**

Algorithm 11.4 Inserting a node in a linked list

```
Algorithm: InsertLinkedList (list, target, new)
Purpose: Insert a node in the linked list after searching the list for the right
position
Pre: The linked list and 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)   return list             // No duplicate
    if (list  = null                           // Insert into empty list
    {
        list ← new
    }

    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.

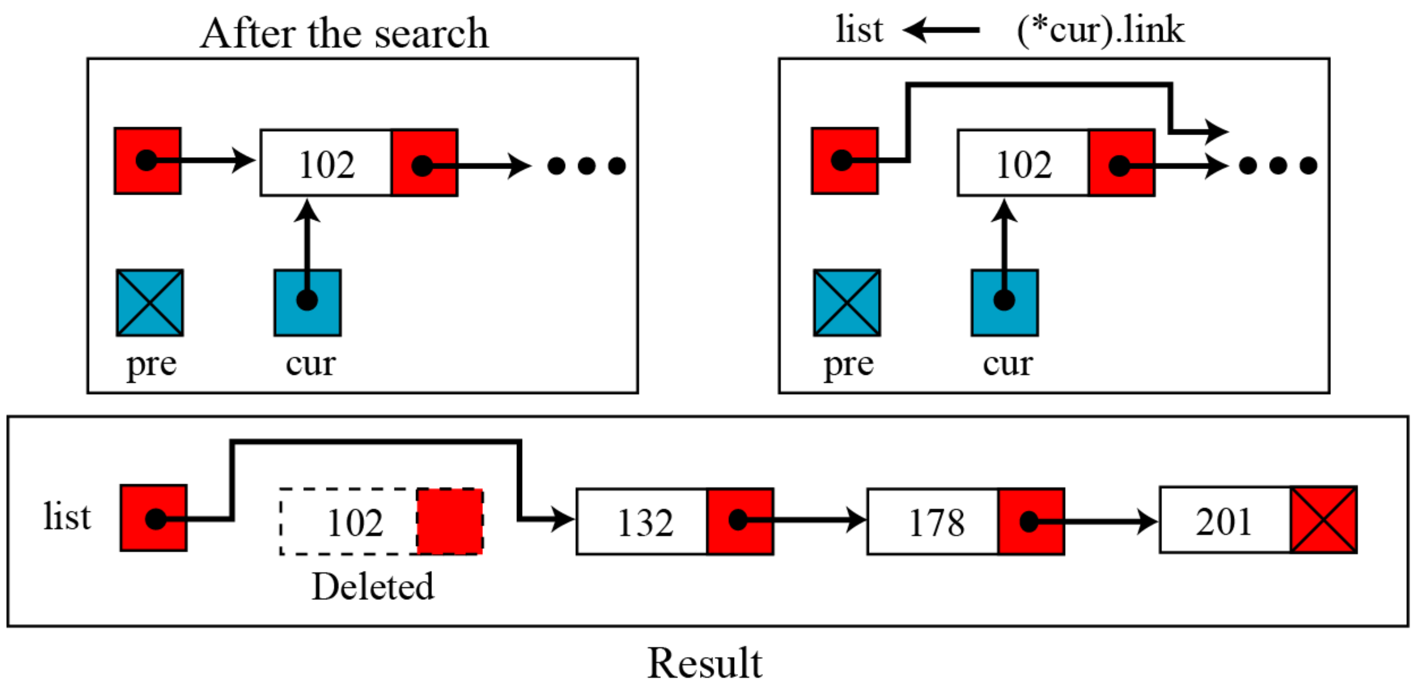


Figure 11.18 **Deleting the first node of a linked list**

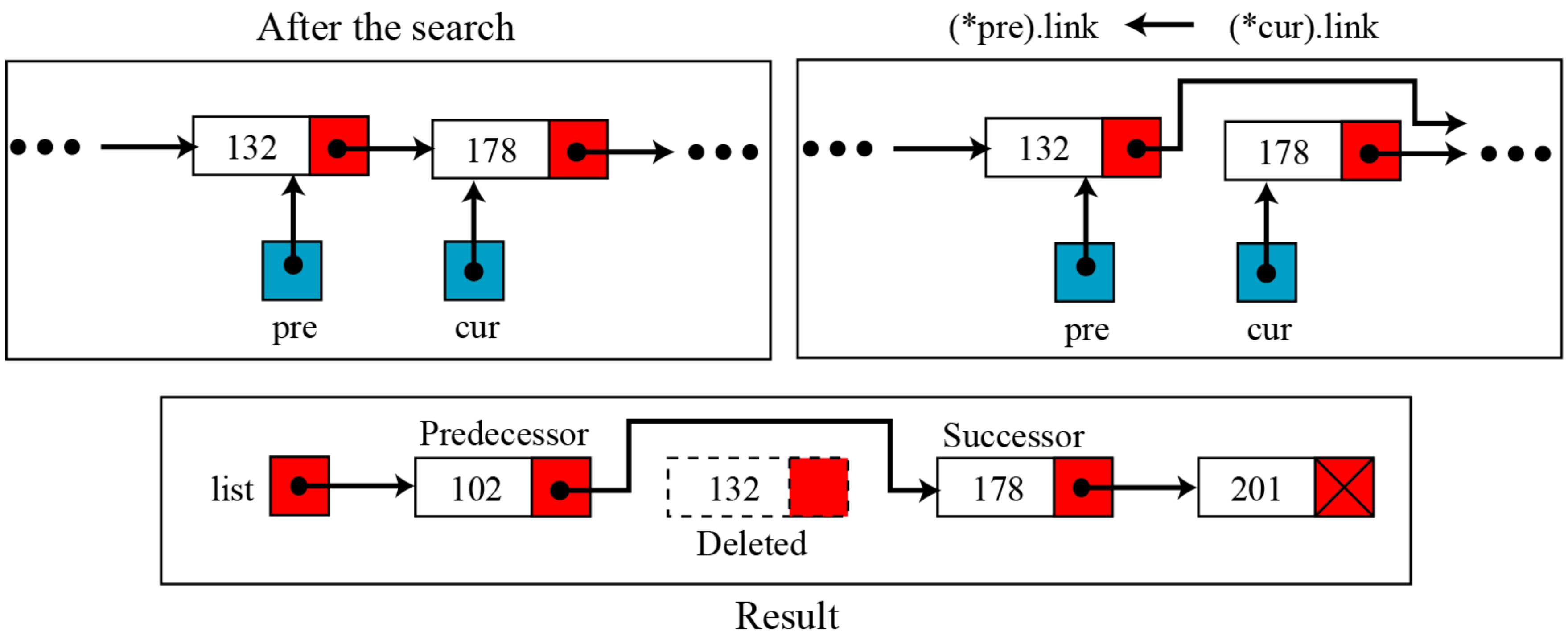


Figure 11.19 **Deleting a node at the middle or end of a linked list**

Algorithm 11.5 Deleting a node in a linked list

```
Algorithm: DeleteLinkedList (list, target)

Purpose: Delete a node in a linked list after searching the list for the right node

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
}
```

Retrieving a node

Retrieving means randomly accessing a node for the purpose of inspecting or copying the data contained in the node. Before retrieving, the linked list needs to be searched. If the data item is found, it is retrieved, otherwise the process is aborted. Retrieving uses only the cur pointer, which points to the node found by the search algorithm. Algorithm 11.6 shows the pseudocode for retrieving the data in a node. The algorithm is much simpler than the insertion or deletion algorithm.

Algorithm 11.6 Retrieving a node in a linked list

```
Algorithm: RetrieveLinkedList (list, target)
Purpose: Retrieves the data in a node after searching the list for the right node
Pre: The linked list (head pointer) and the target (data to be retrieved)
Post: None
Return: Return the data retrieved
{
    searchlinkedlist (list, target, pre, cur, flag)
    if (flag = false) return error    // The node not found
    return (*cur).data
}
```

Traversing a linked list

To traverse the list, we need a “walking” pointer, which is a pointer that moves from node to node as each element is processed. We start traversing by setting the walking pointer to the first node in the list. Then, using a loop, we continue until all of the data has been processed. Each iteration of the loop processes the current node, then advances the walking pointer to the next node. When the last node has been processed, the walking pointer becomes null and the loop terminates (Figure 11.20).

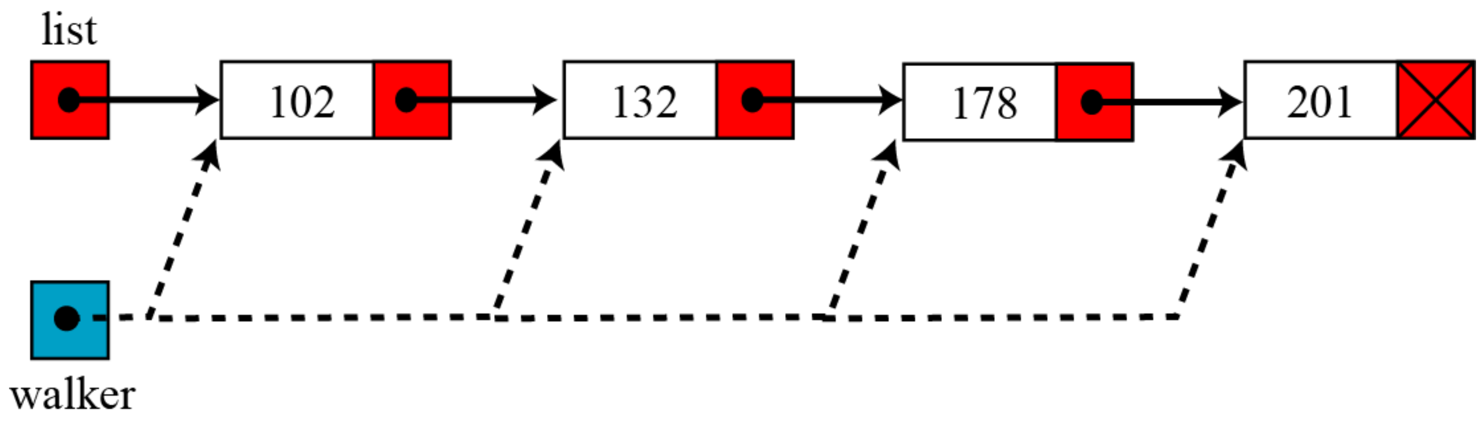


Figure 11.20 **Traversing a linked list**

Algorithm 11.7 Traversing a linked list

```
Algorithm: TraverseLinkedList (list)

Purpose: Traverse a linked list and process each data item

Pre: The linked list (head pointer)

Post: None

Return: The list

{
    walker ← list
    while (walker ≠ null)
    {
        Process (*walker).data
        walker ← (*walker).link
    }
    return list
}
```

Applications of linked lists

A linked list is a very efficient data structure for sorted list that will go through many insertions and deletions. A linked list is a dynamic data structure in which the list can start with no nodes and then grow as new nodes are needed. A node can be easily deleted without moving other nodes, as would be the case with an array. For example, a linked list could be used to hold the records of students in a school. Each quarter or semester, new students enroll in the school and some students leave or graduate.

i

A linked list is a suitable structure if a large number of insertions and deletions are needed, but searching a linked list is slower that searching an array.