BITS Pilani
Pilani Campus

Data Structures & Algorithms Design- SS ZG519
Lecture - 2

Dr. Padma Murali

Lecture 2 Topics

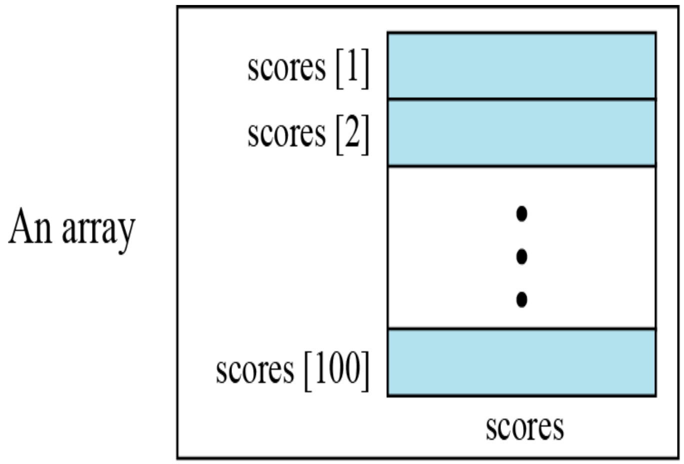
- Arrays, Linked lists
- Analysis of Algorithms -- space and time complexity

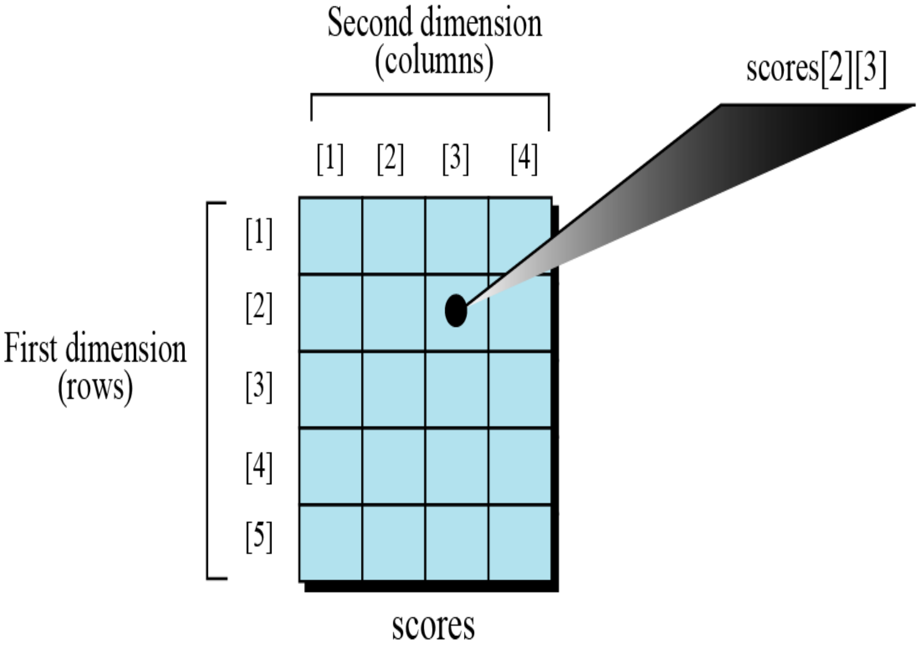
Slides source: 2008 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Lecture notes

Array Data Structure

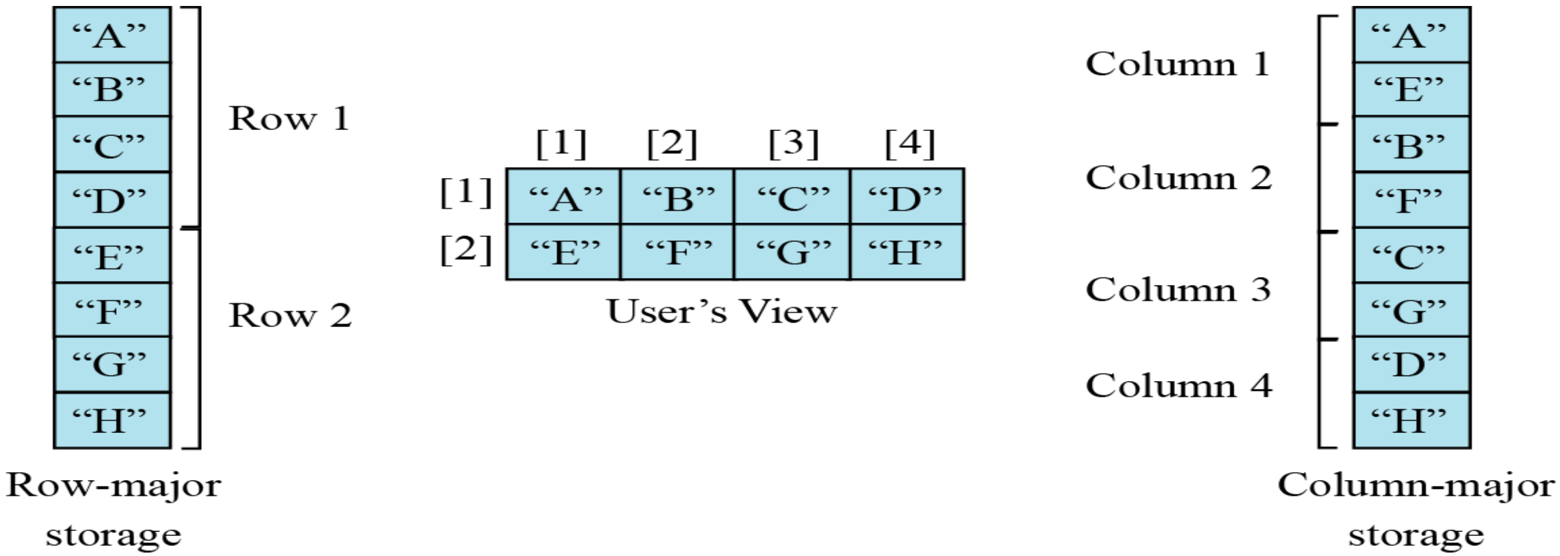
- An array is a sequenced collection of elements, normally of the same data type.
  - **Single-dimensional**
  - **Multi-dimensional**





Memory layout

- The index in an one-dimensional array directly define the relative positions of the element in actual memory.
- two-dimensional array is stored in memory using row-major or column-major storage



We have stored the two-dimensional array students in memory. The array is 100 × 4 (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[1][1] is stored in the memory location with address 1000 and each element occupies only one memory location. The computer uses row-major storage.

Operations on array

- The common operations on arrays as structures are searching, insertion, deletion and traversal.
- An array is more suitable when the number of deletions and insertions is small, but a lot of searching and retrieval activities are expected.

Example

We have stored the two-dimensional array students in memory. The array is 100 × 4 (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[1][1] is stored in the memory location with address 1000 and each element occupies only one memory location. The computer uses row-major storage.

Solution

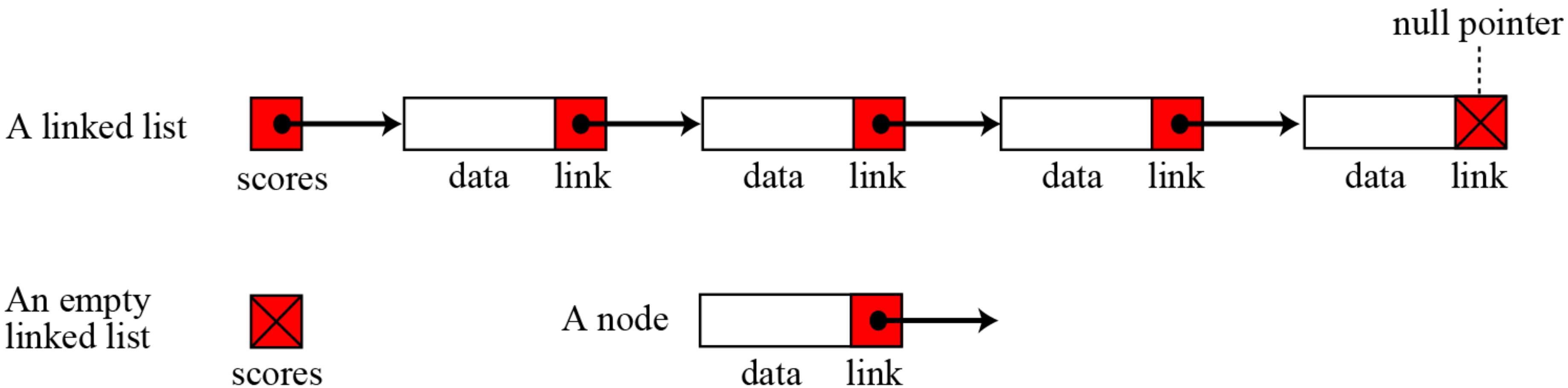
We can use the following formula to find the location of an element, assuming each element occupies one memory location.

$$y = x + \text{Cols} \times (i - 1) + (j - 1)$$

If the first element occupies the location 1000, the target element occupies the location 1018.

Linked Lists

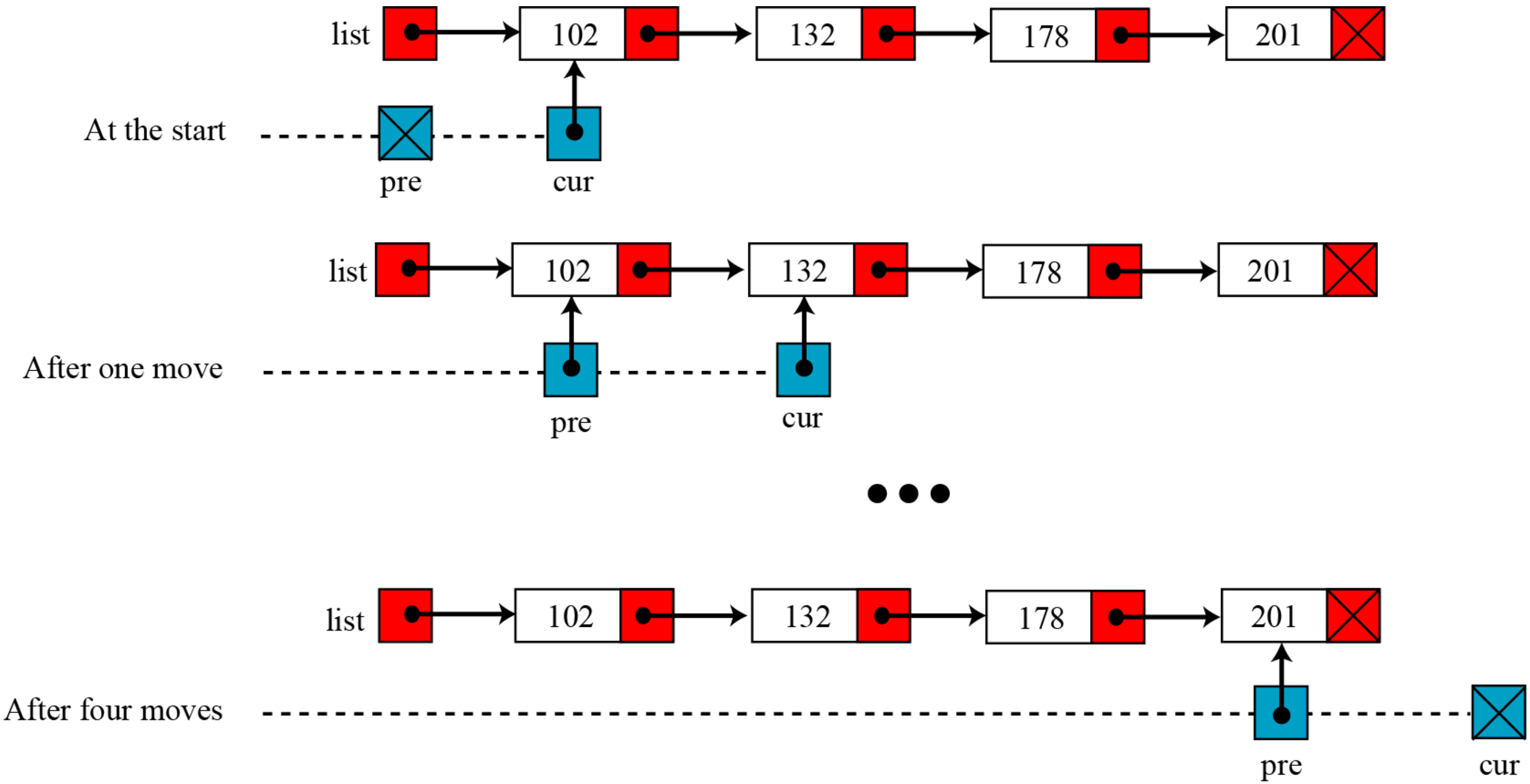
- A linked list is a collection of data in which each element contains the location of the next element.
- Each element contains two parts: data and link. The name of the list is the same as the name of this pointer variable.



Operations on linked lists

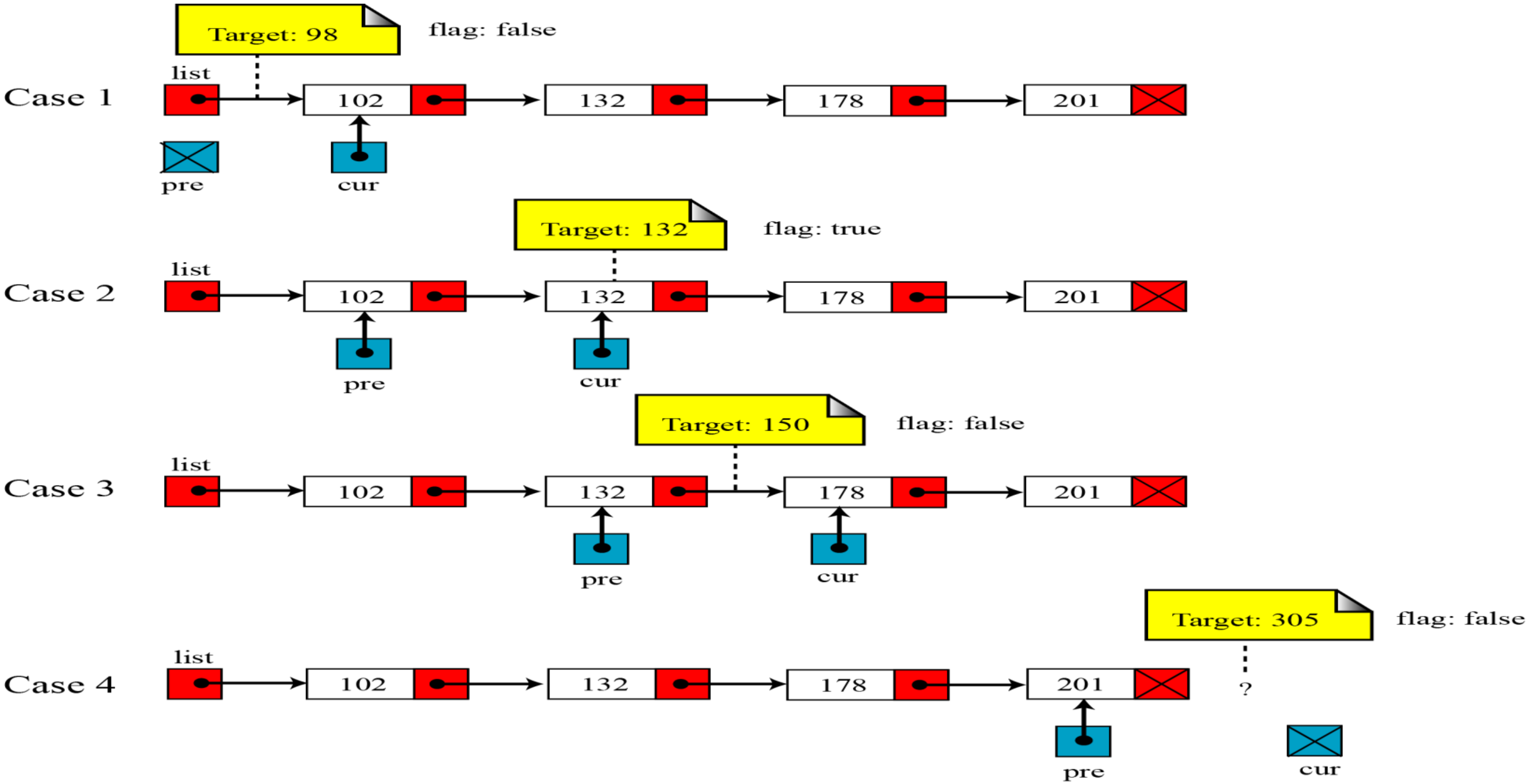
- Search
- Insertion
- Deletion
- Traversal

Search operation



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

Search operation



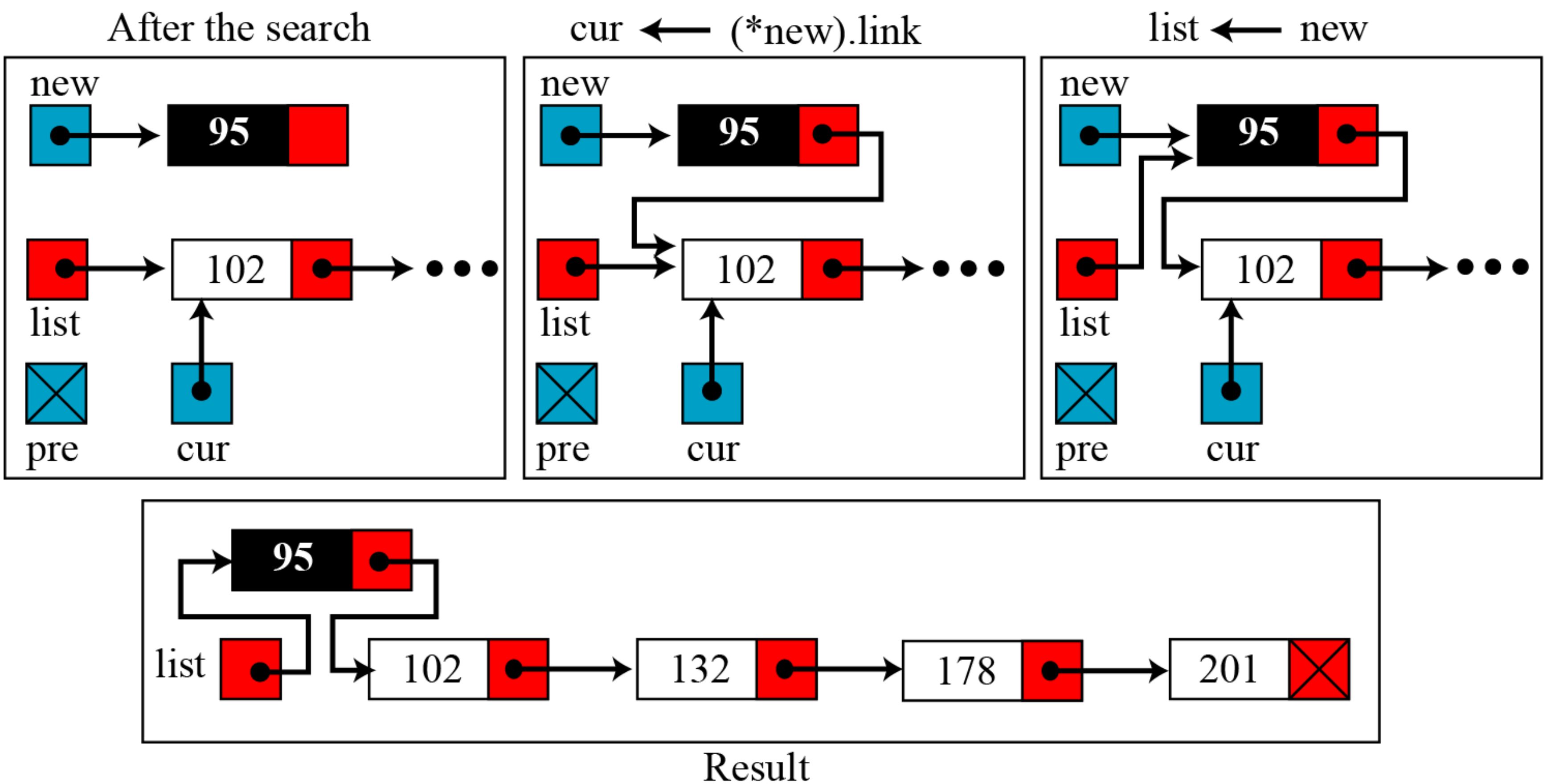
Values of *pre* and *cur* pointers in different cases

Insertion

Four cases can arise:

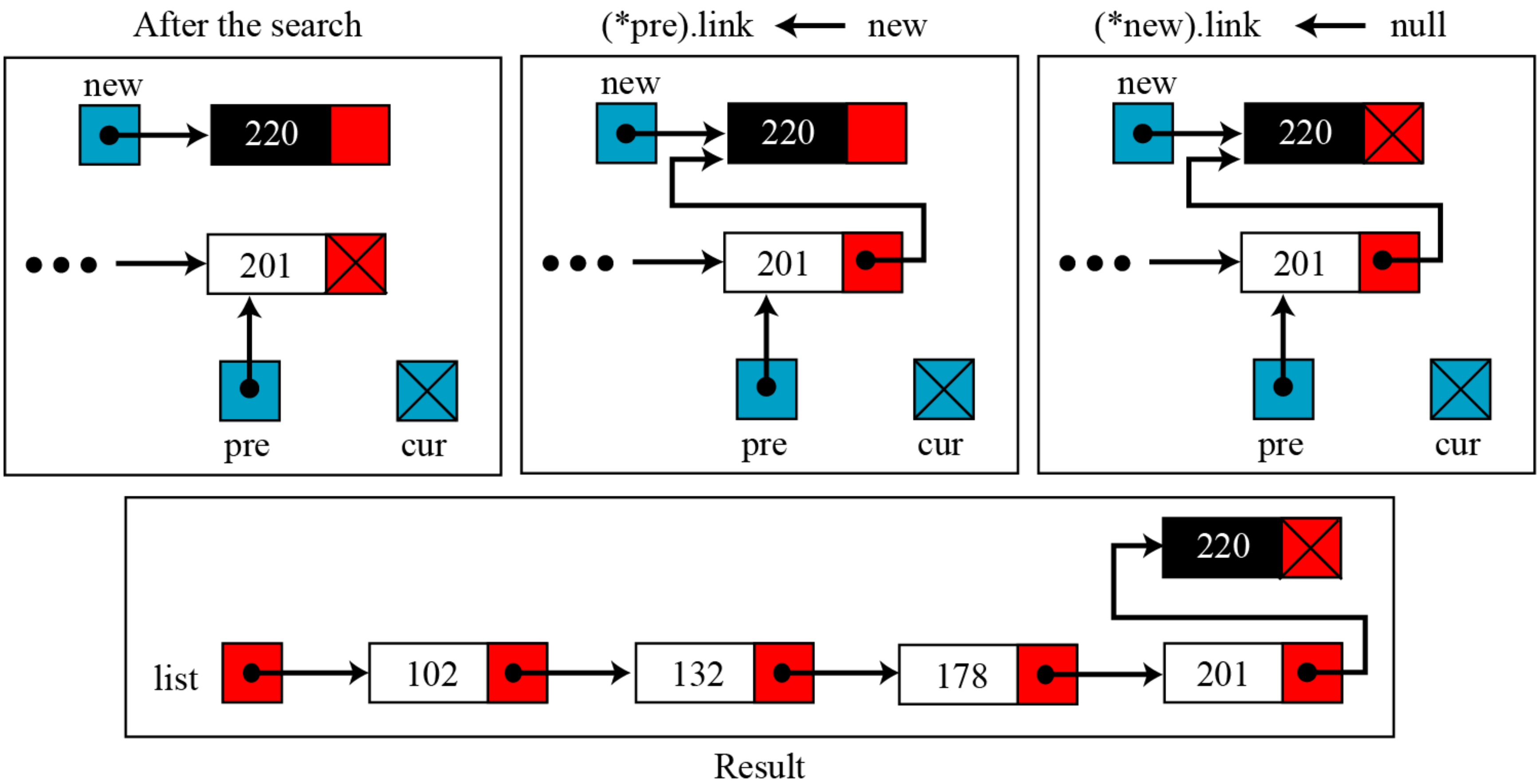
- Inserting into an empty list.
- Insertion at the beginning of the list.
- Insertion at the end of the list.
- Insertion in the middle of the list.

Insertion



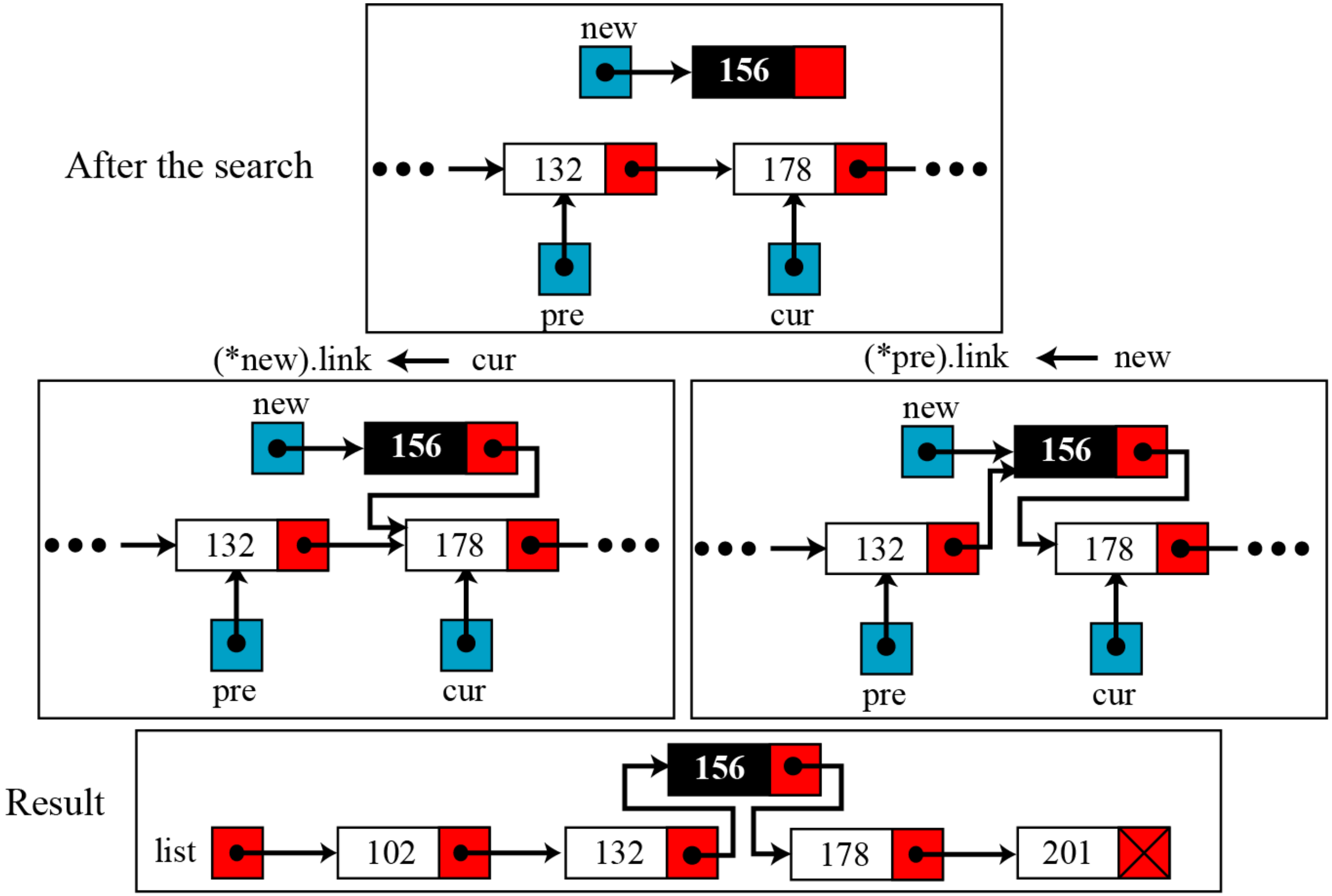
Inserting a node at the beginning of a linked list

Insertion



Inserting a node at the end of the linked list

Insertion



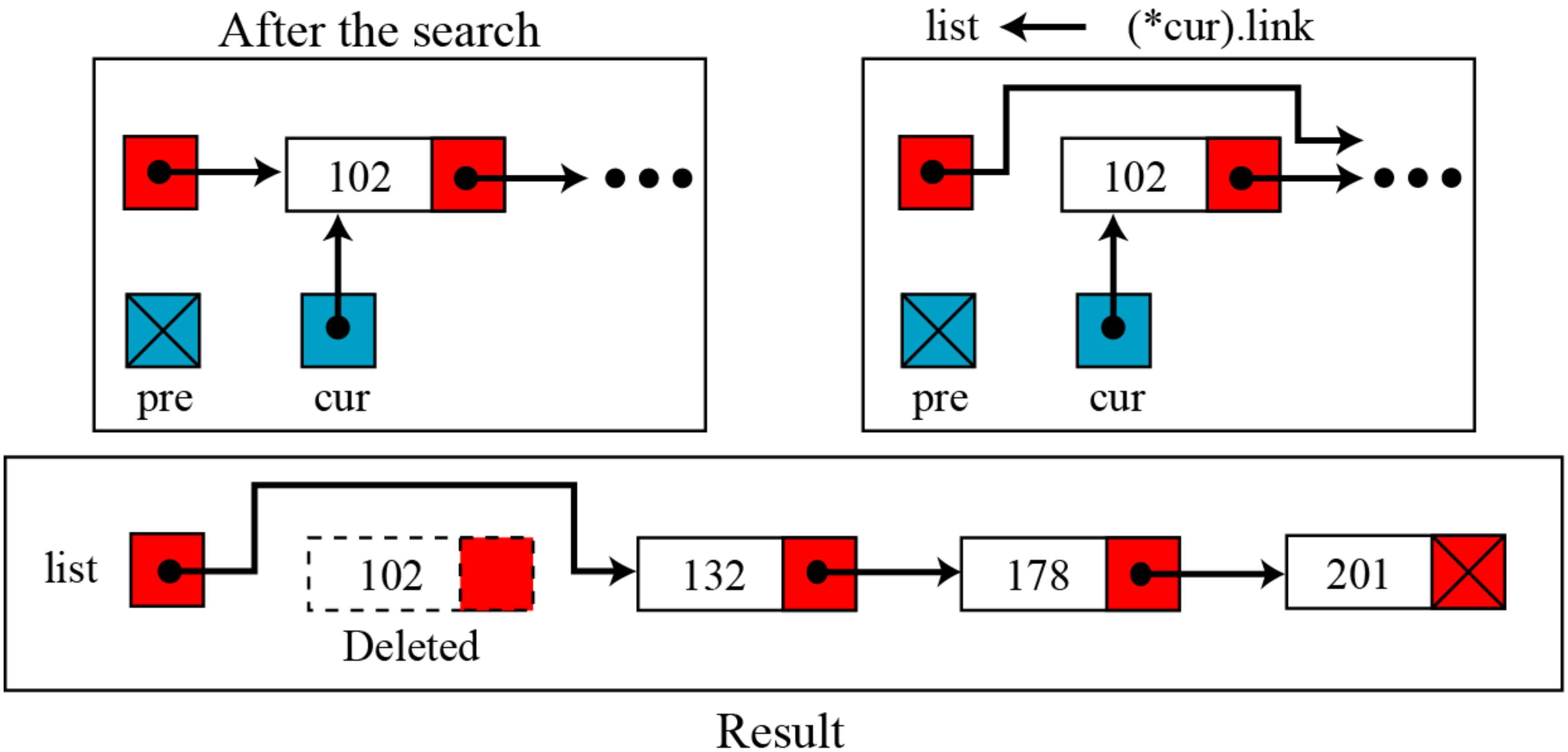
Inserting a node in the middle of the linked list

Deletion

Two cases are:

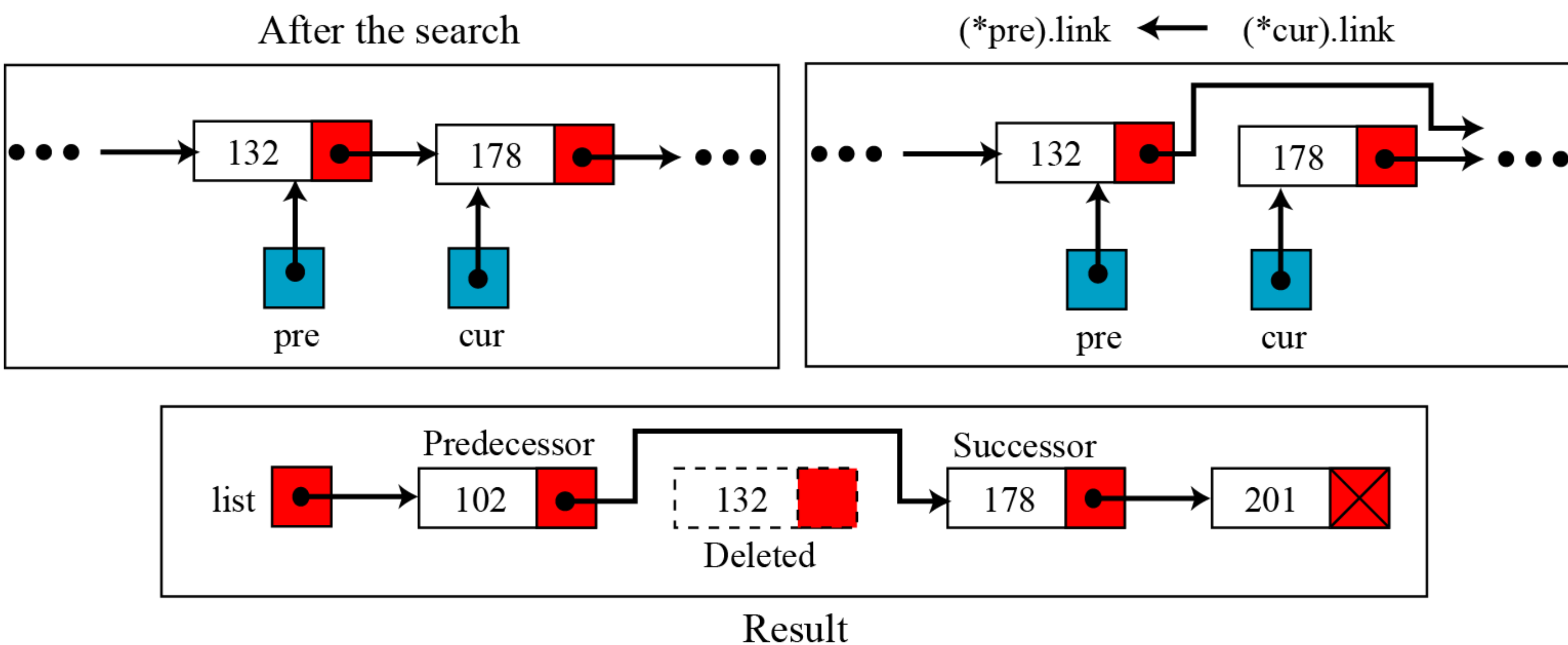
- deleting the first node
- deleting any other node.

Deletion



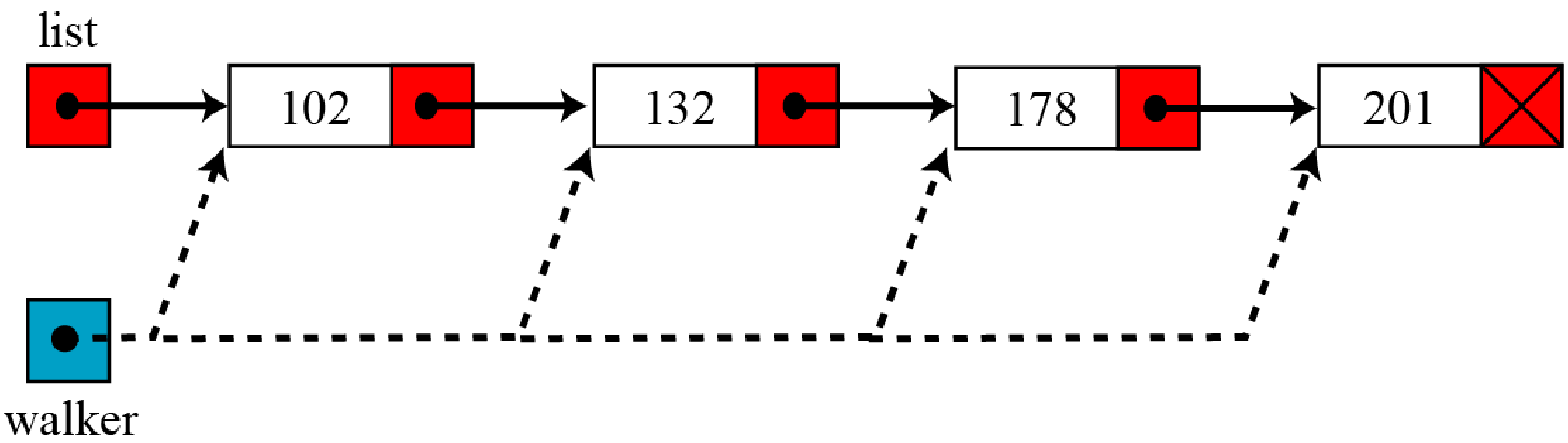
Deleting the first node of a linked list

Deletion



Deleting a node at the middle or end of a linked list

Traversal



Linked List - Applications

- It is a dynamic data structure in which the list can start with no nodes and then grow as new nodes are needed
- It 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.
- It is a very efficient data structure for sorted list that will go through many insertions and deletions

Linked List - Operations

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)
}
```

Linked List - Operations

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
}
```

Linked List - Operations

```
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
}
```

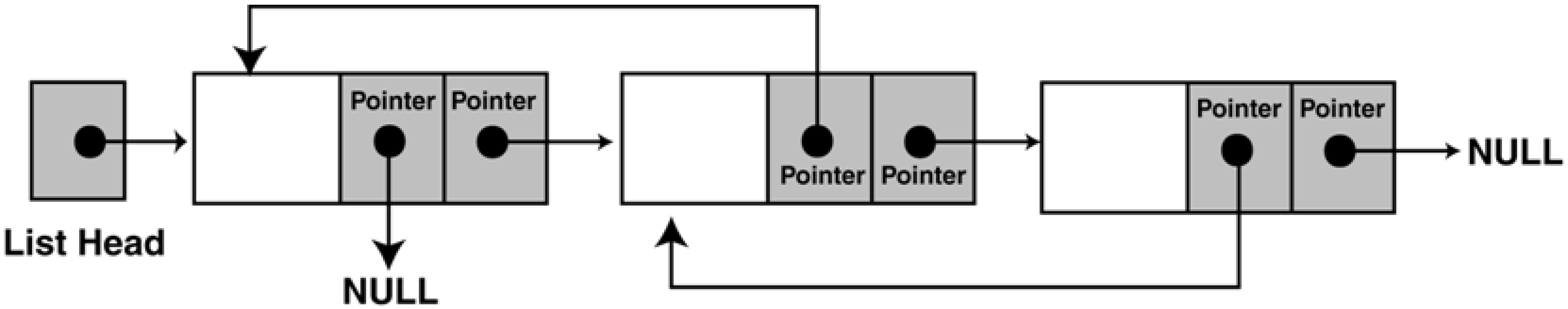
Variations of the Linked List

Singly linked list: It has only head part and corresponding references to the next nodes.

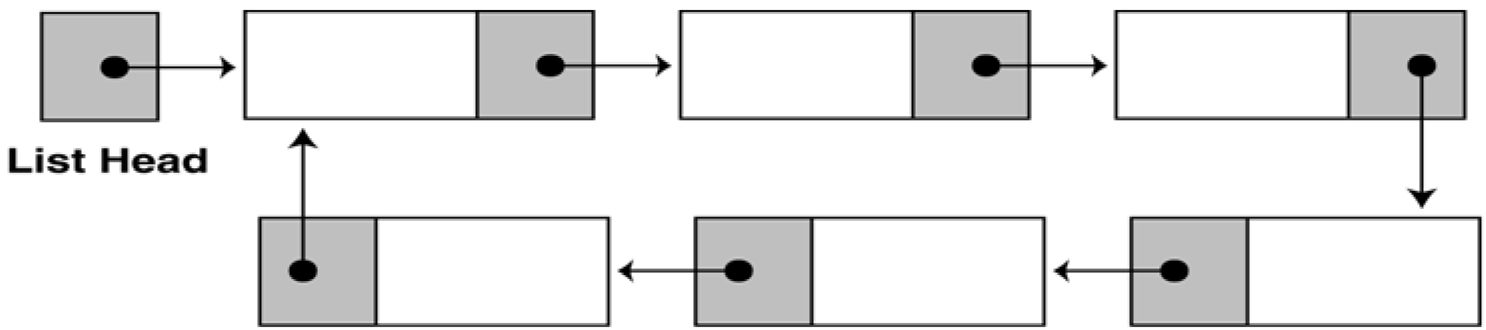
Doubly linked list: A linked list which has both head and tail parts, thus allowing the traversal in bi-directional fashion. Except the first node, the head node refers to the previous node.

Circular linked list: A linked list whose last node has reference to the first node.

Variations of the Linked List

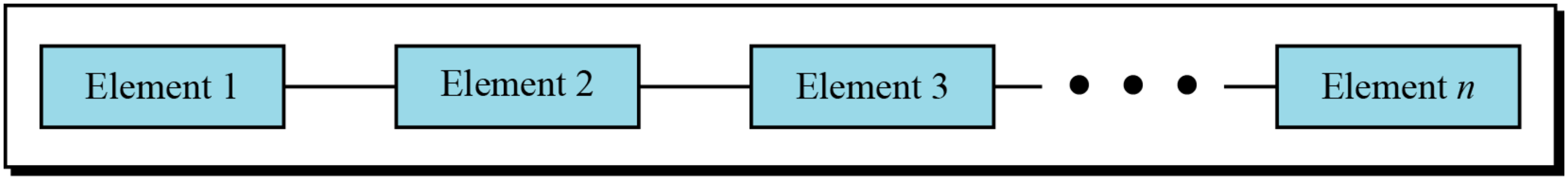


The Doubly-Linked List



The Circular Linked List

Linear Lists

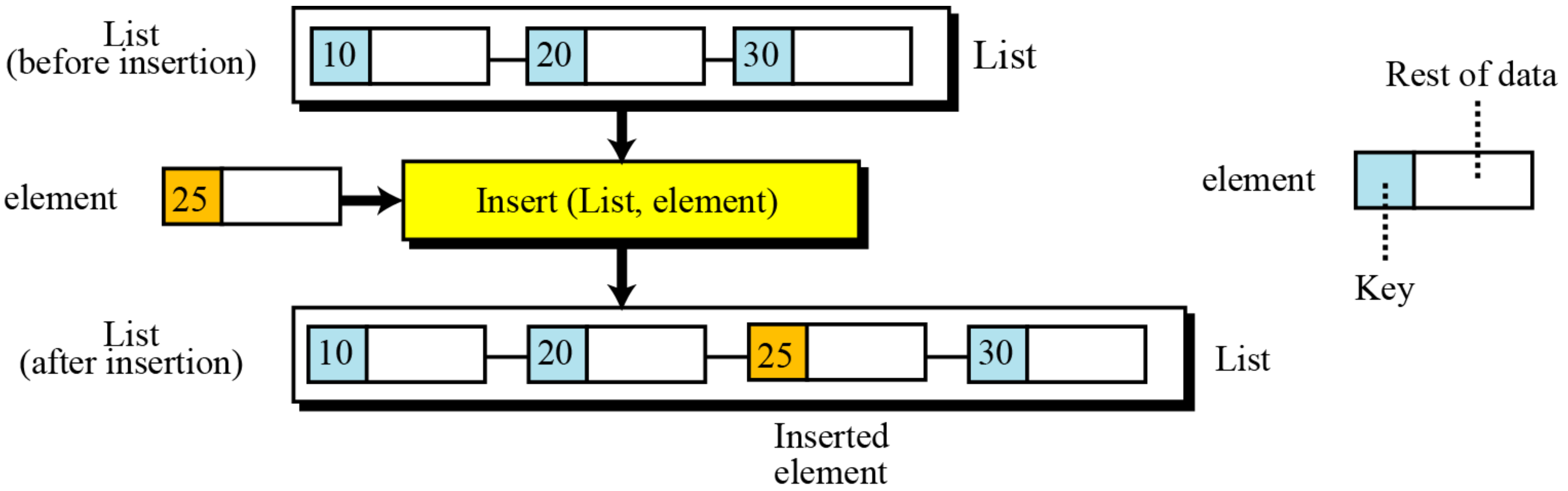


General linear list

Operations on Linear Lists

- List -- **list** (listName)
- Insert --- **insert** (listName, element)
- Delete --- **delete** (listName, target, element)
- Traverse --- **traverse** (listName, action)
- Empty ---- **empty** (listName)

Linear Lists (Insert)



General linear list ADT

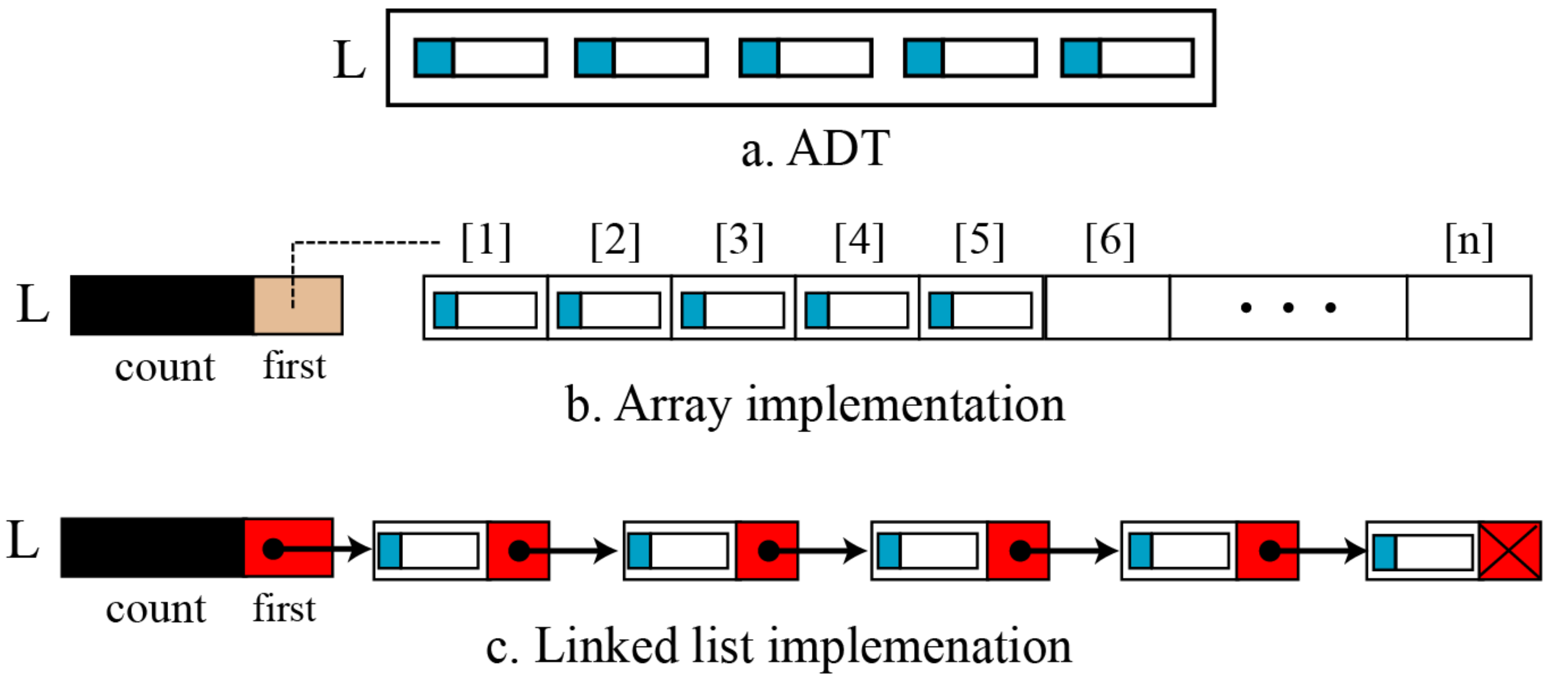
We define a general linear list as an ADT as shown below:

General linear list ADT

Definition	A list of sorted data items, all of the same type.
Operations	**list:** Creates an empty list.
	insert: Inserts an element in the list.
	delete: Deletes an element from the list.
	retrieve: Retrieves an element from the list.
	traverse: Traverses the list sequentially.
	empty: Checks the status of the list.

General linear list implementation

- A general list ADT can be implemented using either an array or a linked list.



Arrays: pluses and minuses

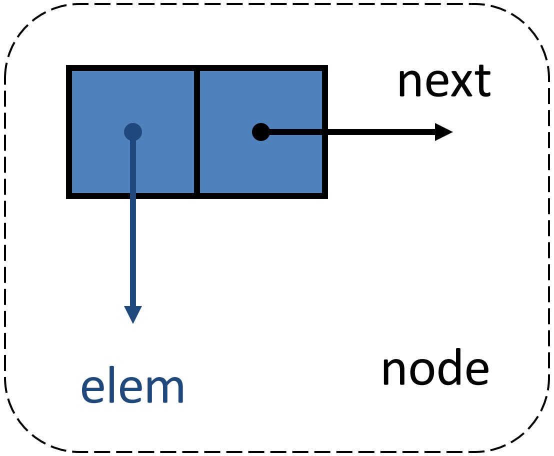
+ Fast element access.

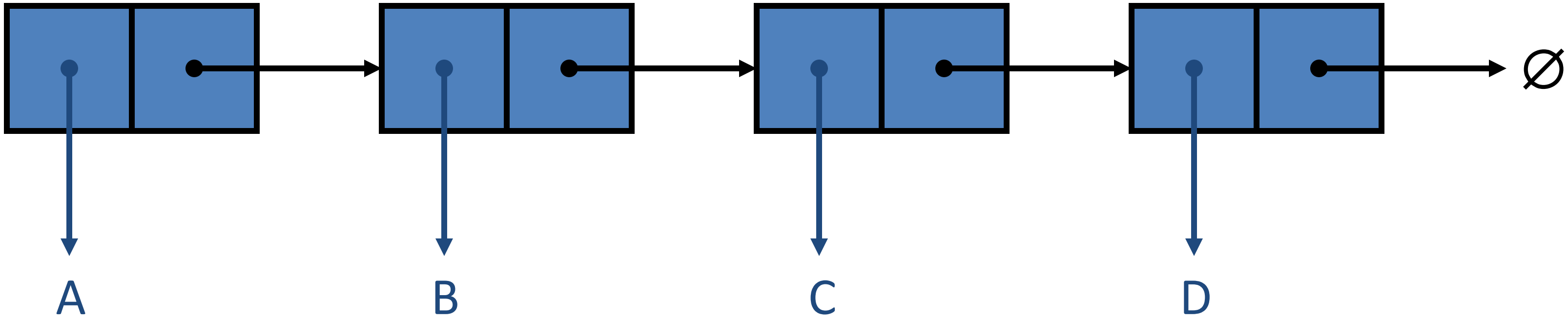
-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.

Singly Linked Lists

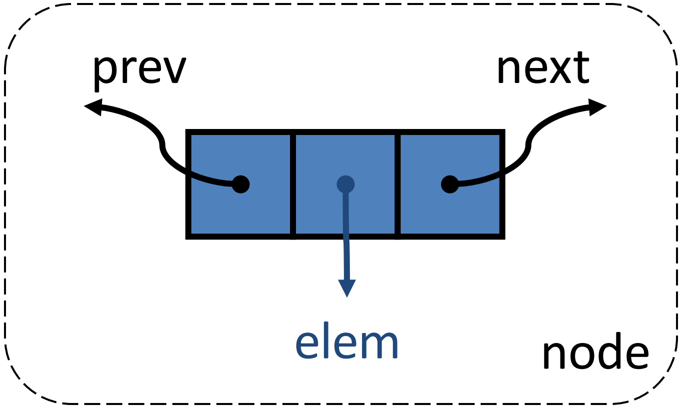
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

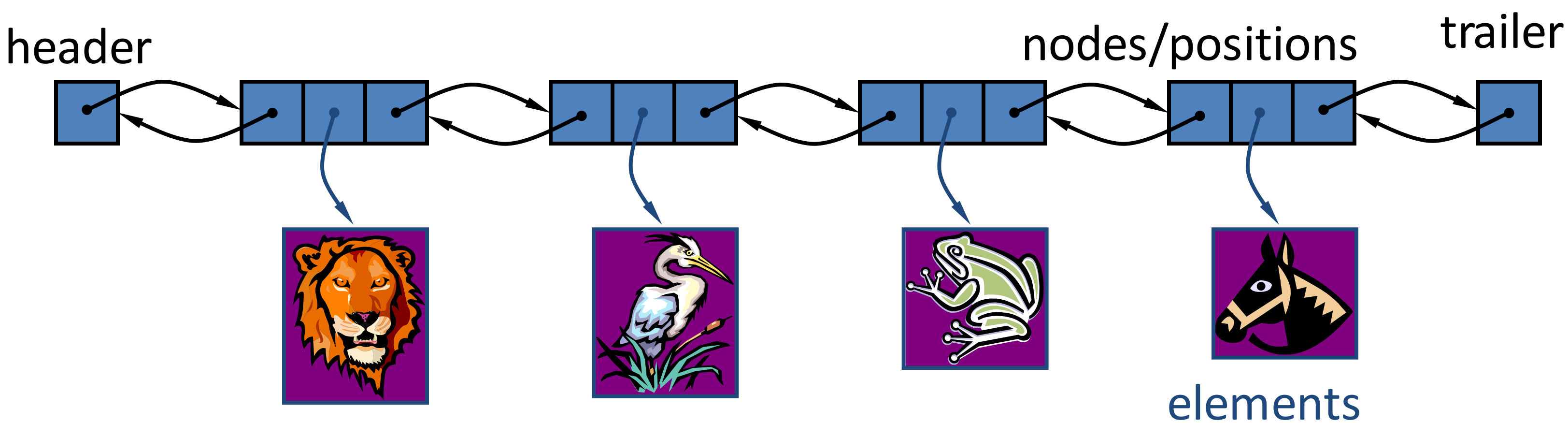




Doubly Linked List

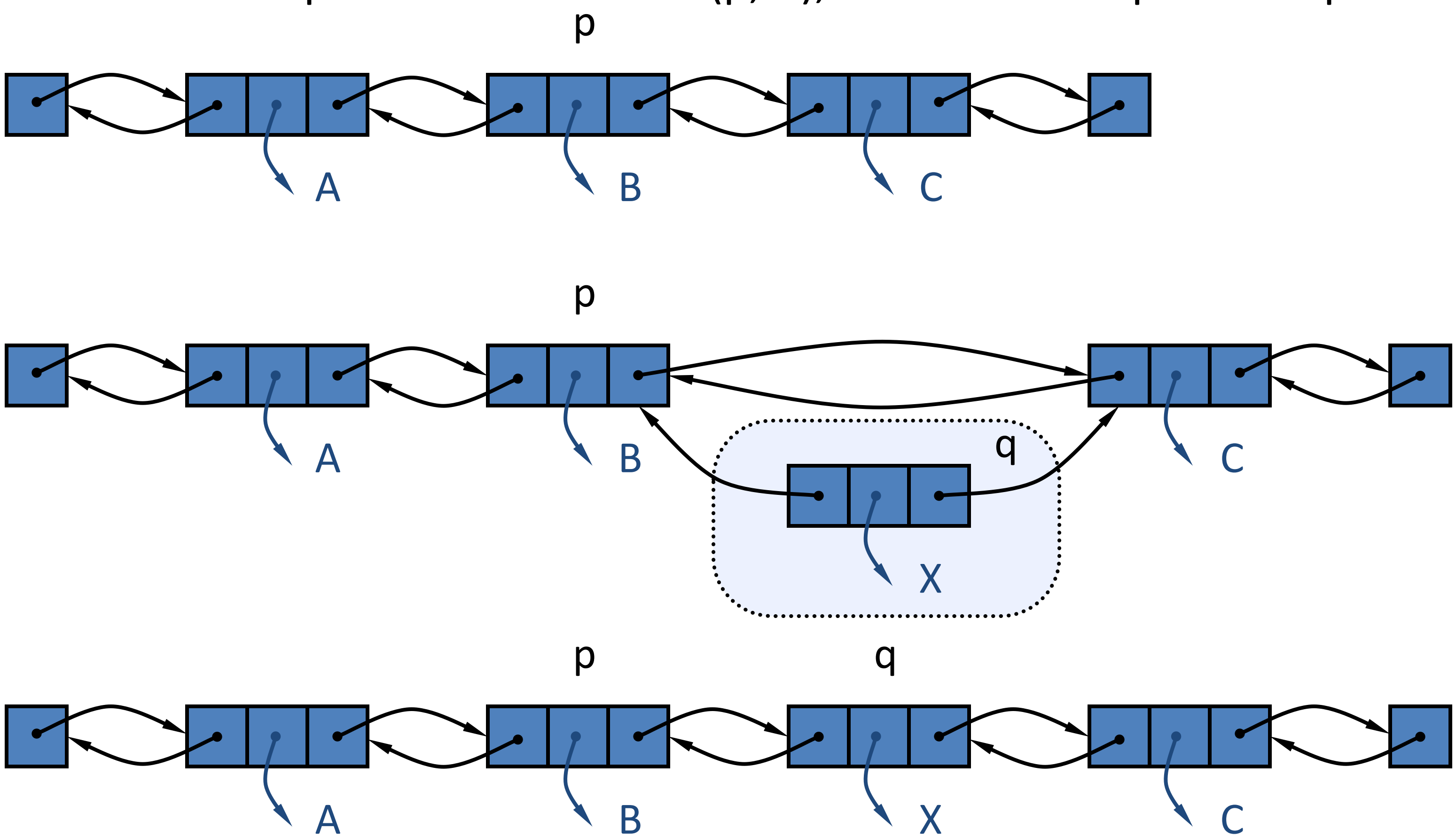
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q



Insertion Algorithm

Algorithm insertAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

v.setPrev(*p*) {link *v* to its predecessor}

v.setNext(*p*.getNext()) {link *v* to its successor}

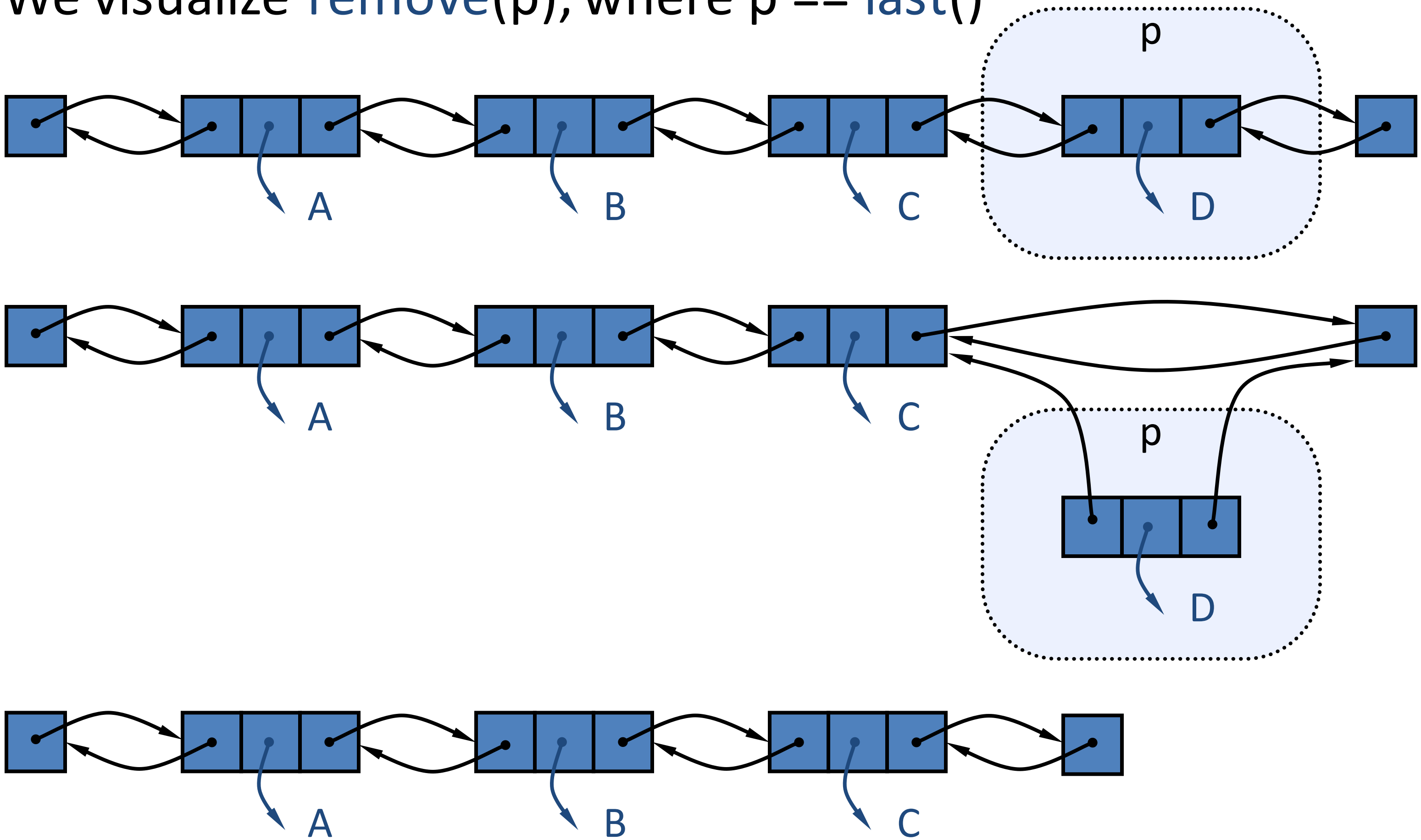
(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}

p.setNext(*v*) {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}

Deletion

- We visualize remove(p), where p == last()



Deletion Algorithm

Algorithm remove(*p*):

t = *p*.element {a temporary variable to hold the return value}

(*p*.getPrev()).setNext(*p*.getNext()) {linking out *p*}

(*p*.getNext()).setPrev(*p*.getPrev())

p.setPrev(**null**) {invalidating the position *p*}

p.setNext(**null**)

return *t*

Complexity Example [1]

Example 1 (Y and Z are input)

```
X = Y * Z;
X = Y * X + Z;
// 2 units of time and 1 unit of storage
// Constant Unit of time and Constant Unit of
   storage
```

Complexity Example [2]

Example 2 (a and N are input)

```
j = 0;
while (j < N) do
   a[j] = a[j] * a[j];
   b[j] = a[j] + j;
   j = j + 1;
endwhile;
// 3N + 1 units of time and N+1 units of storage
// time units prop. to N and storage prop. to N
```

Complexity Example [3]

Example 3 (a and N are input)

```
j = 0;
while (j < N) do
    k = 0;
    while (k < N) do
        a[k] = a[j] + a[k];
        k = k + 1;
    endwhile;
    b[j] = a[j] + j;
    j = j + 1;
endwhile;
//??? units of time and ??? units of storage
// time prop. to N^2 and  storage prop. to N
```

Example of sorting

Input Sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

8,4,6,2,1

Sort

Output a Permutation of input of numbers

$b_1, b_2, b_3, \ldots, b_n$

1,2.4.6.8

Correctness(Requirement for the output)

For any input algorithm halts with the output:

- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time of algorithm depends on

- Number of elements n.
- How (partially)sorted they are.

Order Notation

- Purpose
  - Capture proportionality
  - Machine independent measurement
  - Asymptotic growth
    (i.e. large values of input size N)

Motivation for Order Notation

Examples

- $100 * \log_2 N < N$ for $N > 1000$
- $70 * N + 3000 < N^2$ for $N > 100$
- $10^5 * N^2 + 10^6 * N < 2^N$ for $N > 26$

Asymptotic Analysis

- Goal: To simplify analysis of running time of algorithm .eg $3n^2=n^2$.
- Capturing the essence: how the running time of the algorithm increases with the size of the input in the limit.

Asymptotic Notation

- The big O notation

  Definition

  Let f and g be functions from the set of integers to the set of real numbers. We say that $f(x)$ is in $O(g(x))$ if there are constants C > and k such that

  $|f(x)| \leq C\,|g(x)|$, **whenever** $x >= k$.

- This is read as $f(x)$ is ***big-oh*** of $g(x)$

Note: Pair of C and k is never unique.

Order Notation

Examples

$g(n) = 17*N + 5$

$\lim_{n\to\infty} g(n) / f(n) = c$

$\lim_{n\to\infty} (17*N + 5)/N = 17$. The asymptotic complexity is $O(N)$

$g(n) = 5*N^3 + 10*N^2 + 3$

$\lim_{n\to\infty} (5*N^3 + 10*N^2 + 3) / N^3 = 5$. The asymptotic complexity is $O(N^3)$

$g(n) = C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C$

$\lim_{n\to\infty} (C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C) / N^k = C1$.

The asymptotic complexity is $O(N^k)$

$2^N + 4*N^3 + 16$ is $O(2^N)$

$5*N*\log(N) + 3*N$ is $O(N*\log(N))$

1789 is $O(1)$

Linear Search

```
function search(X, A, N)
 j = 0;
while (j < N)
      if (A[j] == X)  return j;
      j++;
 endwhile;
return "Not-found";
```

Linear Search - Complexity

Time Complexity

“if” statement introduces possibilities

- Best-case: O(1)
- Worst case: O(N)
- Average case: ???

Binary Search Algorithm

```
Assume: Sorted Sequence of numbers
low = 1;  high = N;
while (low <= high) do
   mid = (low + high) /2;
   if (A[mid] = = x)  return x;
   else if (A[mid] < x) low = mid +1;
   else high = mid – 1;
endwhile;
  return Not-Found;
```

Binary Search - Complexity

- Best Case
  - O(1)
- Worst case:
  - Loop executes until low <= high
  - Size halved in each iteration
  - N, N/2, N/4, … 1
  - How many steps ?

Binary Search - Complexity

- Worst case:
  – K steps such that $2^K = N$

  i.e. $\log_2 N$ steps is $O(\log(N))$

Algorithm Analysis

Predict the amount of resources required:

- ✓ memory: how much space is needed?
- ✓ computational time: how fast the algorithm runs?

FACT: running time grows with the size of the input

Input size (number of elements in the input)

- Size of an array, polynomial degree, # of elements in a matrix, # of bits in the binary representation of the input, vertices and edges in a graph

Def: *Running time = the number of primitive operations (steps) executed before termination*

Running time is expressed as **T**(*n*) for some function **T** on input size *n*.

Algorithm Analysis

Two approaches to obtaining running time:

- Measuring under standard benchmark conditions.
- Estimating the algorithms performance

Estimation is based on:

- The "size" of the input
- The number of *basic operations*

The time to complete a basic operation does not depend on the value of its operands.

Running Time (Example)

```
sum = 0;
{
for (k=1; k<=n; k++)
   for (j=1; j<=k; j++)
      sum++;
}
```

What is the running time for this code?

Running Time (Example)

Number of executions

k	1	2	3	….	n
j	1	1,2	1,2,3	…	1,2,. n
# runs	1	2	3	…	n

Running Time (Example)

$$\# \text{ runs} = 1 + 2 + 3 + 4 \,..+ n = \sum_{j=1}^{n} j$$

$$\sum_{j=1}^{n} j = n(n+1)/2 \qquad = n^2$$

$T(n) = c1 + c2\ (n+1) + c3(n^2 + 1) + c4\ (n^2\ ) =$ Order of n^2

Running Time (Example)

What is the running time for the following codes?

```
a)   sum1 = 0;
    for (k=1; k<=n; k*=2)
      for (j=1; j<=n; j++)
            sum1++;
```

```
b)   sum2 = 0;
    for (k=1; k<=n; k*=2)
      for (j=1; j<=k; j++)
            sum2++;
```

```
c) sum3 = 0;
    for (k=1; k<=n; k++)
      for (j=1; j<=n; j++)
            sum3++;
```

Running Time (Example a)

Number of executions

k	1	2	4	….	n
j	1,2,..n	1,2,..n	1,2..n	…	1,2,. n
# runs	n	n	n	…	log n

N x log N

Running Time (Example a)

$$\# \text{ runs} = (1 + ..N) \log n = \sum_{j=1}^{\log n} n$$

$$\sum_{j=1}^{\log n} n = n \log n$$

T(n) = Order of n log n.

Running Time (Example b)

Number of executions

k	1	2	4	....	n
j	1	1,2	1,2,3,4	...	1,2,. n
# runs	1	2	4	...	log n

1 + 2 + 4 + 8 + 16 + …… n

Running Time (Example b)

runs = 1+2+4+8+16 ..+ n

$= 1 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots\ldots + 2^{\log n}$

$$\sum_{j=1}^{\log n} 2^i = 2n-1$$

T(n) = Order of n.