Computer Programming II

Ming-Feng Tsai (Victor Tsai)

Dept. of Computer Science
National Chengchi University

Analysis of Algorithms

Analysis of Algorithm

- Understand the performance of an algorithm
  - Worst-case analysis
    - the metric by which most algorithms are compared
  - O-notation
    - the most common notation used to formally express an algorithm's performance
  - Computational complexity
    - the growth rate of the resources (usually time) an algorithm requires with respect to the size of the data it processes

Worst-Case Analysis

- Typically, three cases are used to analyze an algorithm
    - the best case, worst case, and average case
- For example, linear search
    - best case: 1
    - worst case: n
    - average case: n/2

Reasons for Worst-Case Analysis

- There are four reasons why algorithms are generally analyzed by their worst case
    - Many algorithms perform to their worst case a large part of the time
    - The best case is not very informative because many algorithms perform exactly the same in the best case
    - Determining average-case performance is not always easy
    - The worst case gives us an upper bound on performance

O-Notation

- O-notation
  - the most common notation used to express an algorithm’s performance in a formal manner
  - express the upper bound of a function within a constant factor
- We express an algorithm’s performance as a function of the size of the data it processes
- We are only interested in the growth rate of the function

Simple Rules of O-Notation

- We can ignore constant terms
    - $T(n) = n + 50$, when $n = 1024$ => the constant term constitutes less than 5% of the running time
- We can ignore constant multipliers of terms
    - $T_1(n) = n^2$ and $T_2(n) = 10n$
- We need only consider the highest-order term
    - $T(n) = n^2 + n$, when $n = 1024$ => the lesser-order term constitutes less than 0.1% of the running time

Simple Rules of O-Notation

- Constant terms are expressed as O(1)
  - $O(c) = O(1)$
- Multiplicative constants are omitted
  - $O(cT) = cO(T) = O(T)$
- Addition is performed by taking the maximum
  - $O(T_1) + O(T_1+T_2) = \max(O(T_1), O(T_2))$
- Multiplication is not changed but often is rewritten more compactly
  - $O(T_1)\ O(T_2) = O(T_1T_2)$

O-notation example and why it works

- Some examples demonstrate why they work so well in describing a function’s growth rate
    - $T(n) = 3n^2 + 10n + 10$
    - $O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$

O-notation example and why it works

- Some examples demonstrate why they work so well in describing a function’s growth rate
  - $T(n) = 3n^2 + 10n + 10$
  - $O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$

when n = 10
Running time for 3n^2: 3(10)2 / (3(10)2 + 10(10) + 10) = 73.2%
Running time for 10n: 10(10) / (3(10)2 + 10(10) + 10) = 24.4%
Running time for 10: 10 / (3(10)2 + 10(10) + 10) = 2.4%

O-notation example and why it works

- Some examples demonstrate why they work so well in describing a function's growth rate
  - $T(n) = 3n^2 + 10n + 10$
  - $O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$

when n = 10
Running time for 3n^2: 3(10)2 / (3(10)2 + 10(10) + 10) = 73.2%
Running time for 10n: 10(10) / (3(10)2 + 10(10) + 10) = 24.4%
Running time for 10: 10 / (3(10)2 + 10(10) + 10) = 2.4%

when n = 100
Running time for 3n^2: 3(100)2 / (3(100)2 + 10(100) + 10) = 96.7%
Running time for 10n: 10(100) / (3(100)2 + 10(100) + 10) = 3.2%
Running time for 10: 10 / (3(100)2 + 10(100) + 10) < 0.1%

Computational Complexity

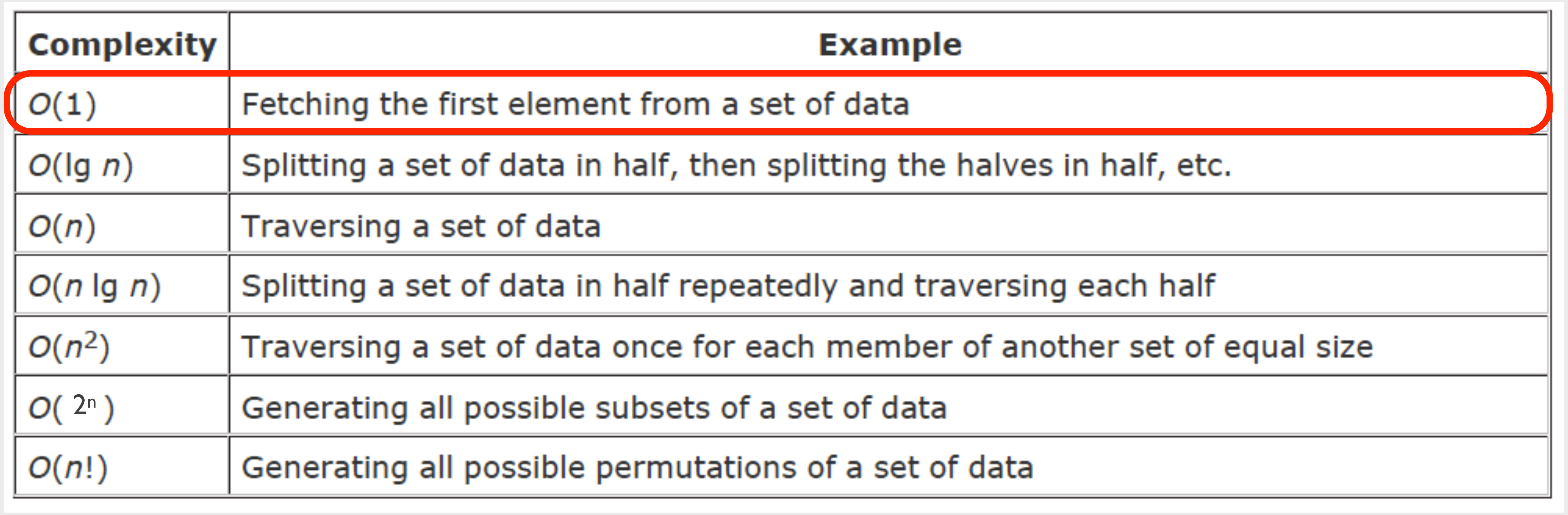
- Speaking of the performance of an algorithm, usually the aspect of interest is its **complexity**
    - the growth rate of the resources (usually time) it requires w.r.t. the size of the data it processes
- For example
    - an algorithm consists of 6 statements
    - if statements 3, 4, and 5 are executed in a loop from 1 to n and the other statements are executed sequentially
    - the overall cost of the algorithm
        - $T(n) = c_1 + c_2 + n(c_3 + c_4 + c_5) + c_6 <= k * n$ (k is a constant factor)
          $= O(n)$

Computational Complexity

- Complexity
  - little info about the actual time the algorithm will take on run
    - a low growth rate does not necessarily mean it will execute in a small amount of time
  - no real units of measurement
    - only describes how the resource being measured will be affected by a change in data size

Computational Complexity

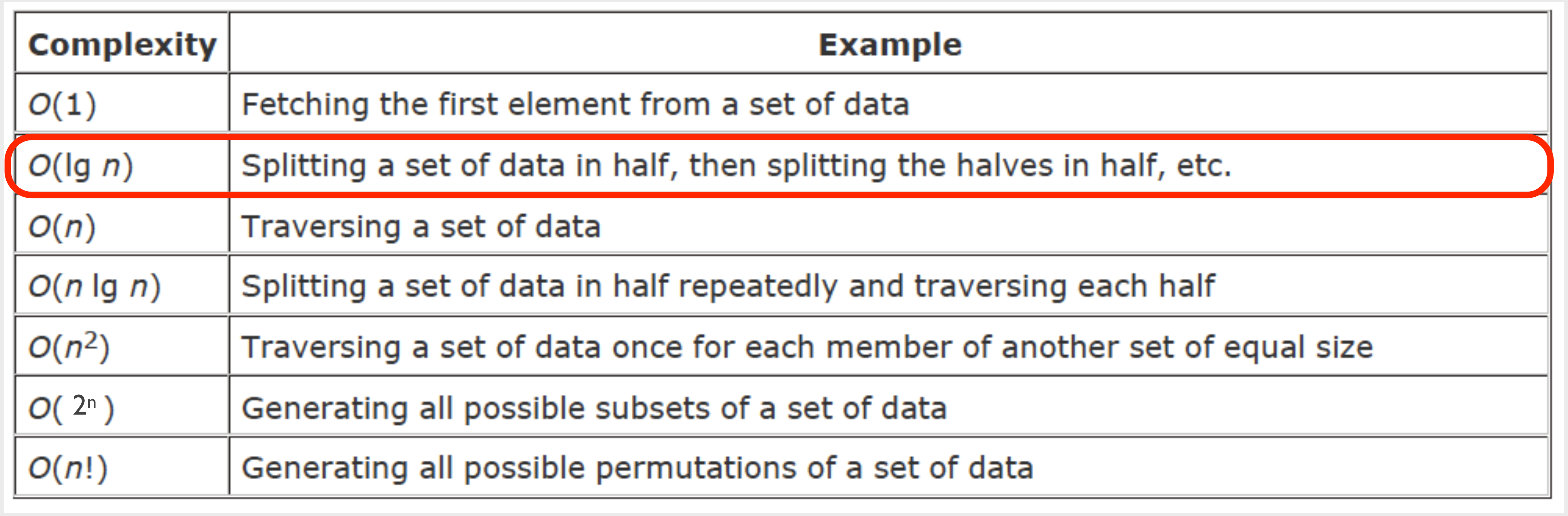
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

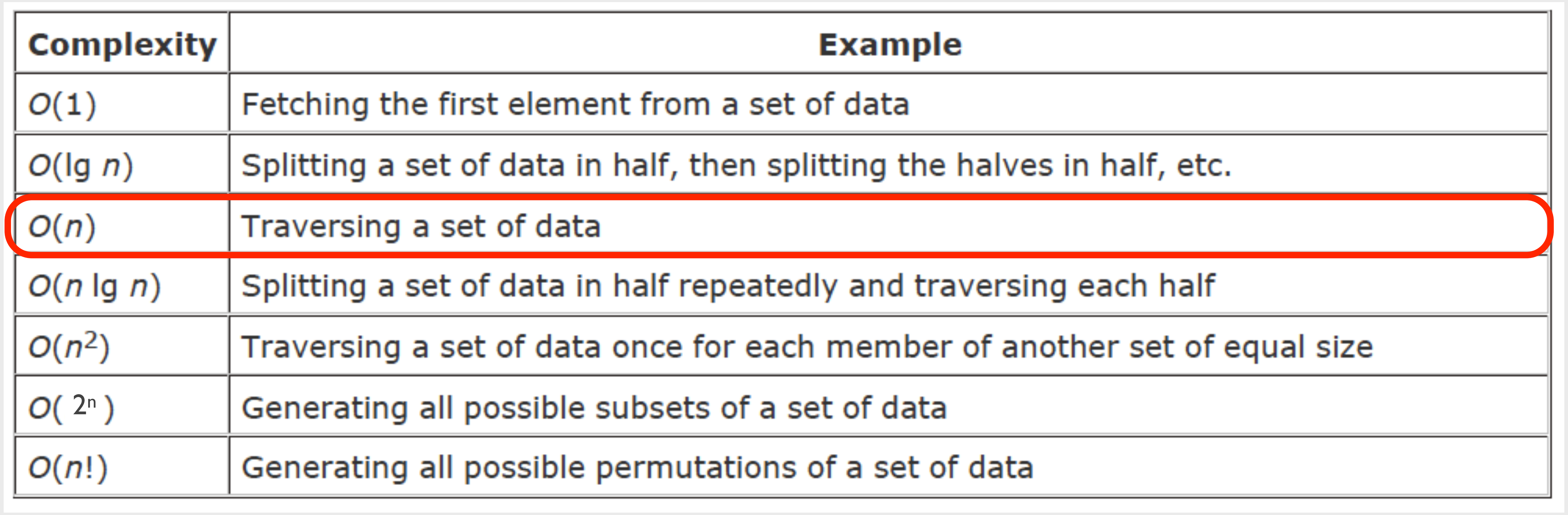
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

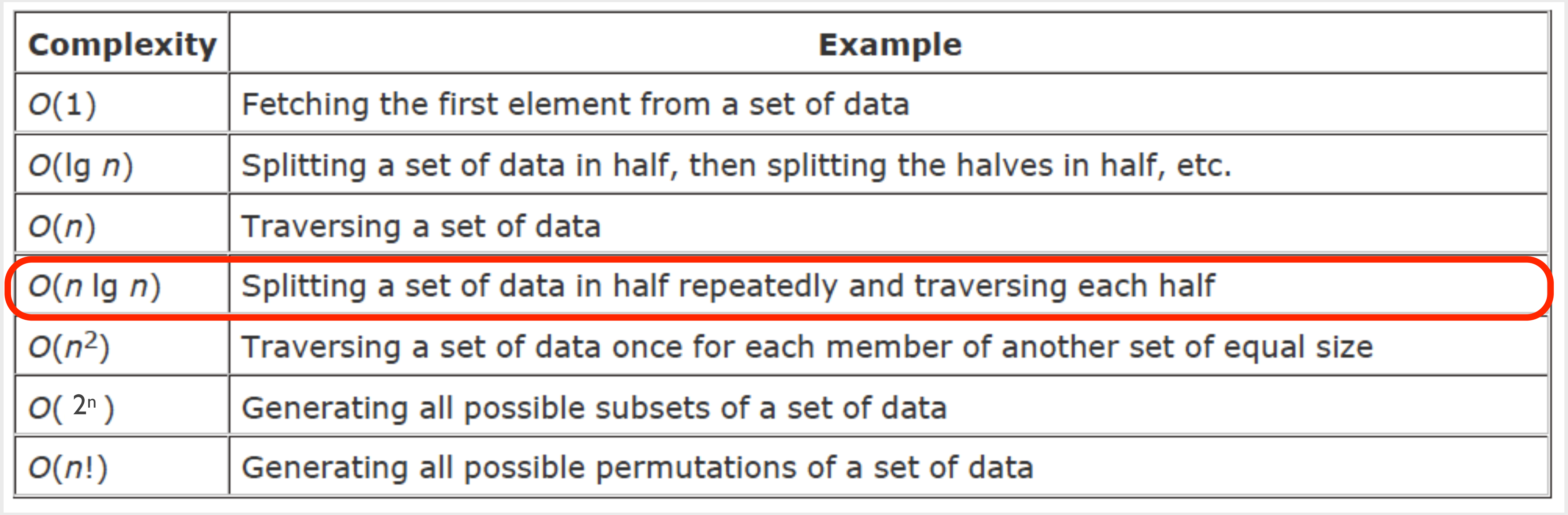
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

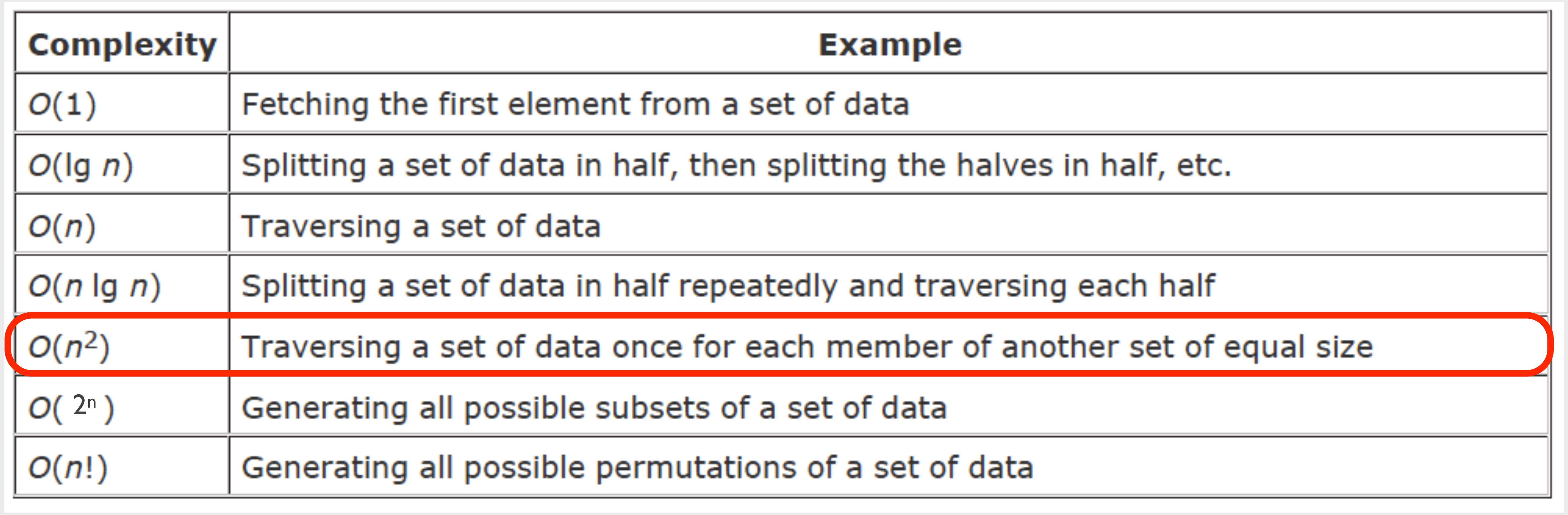
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

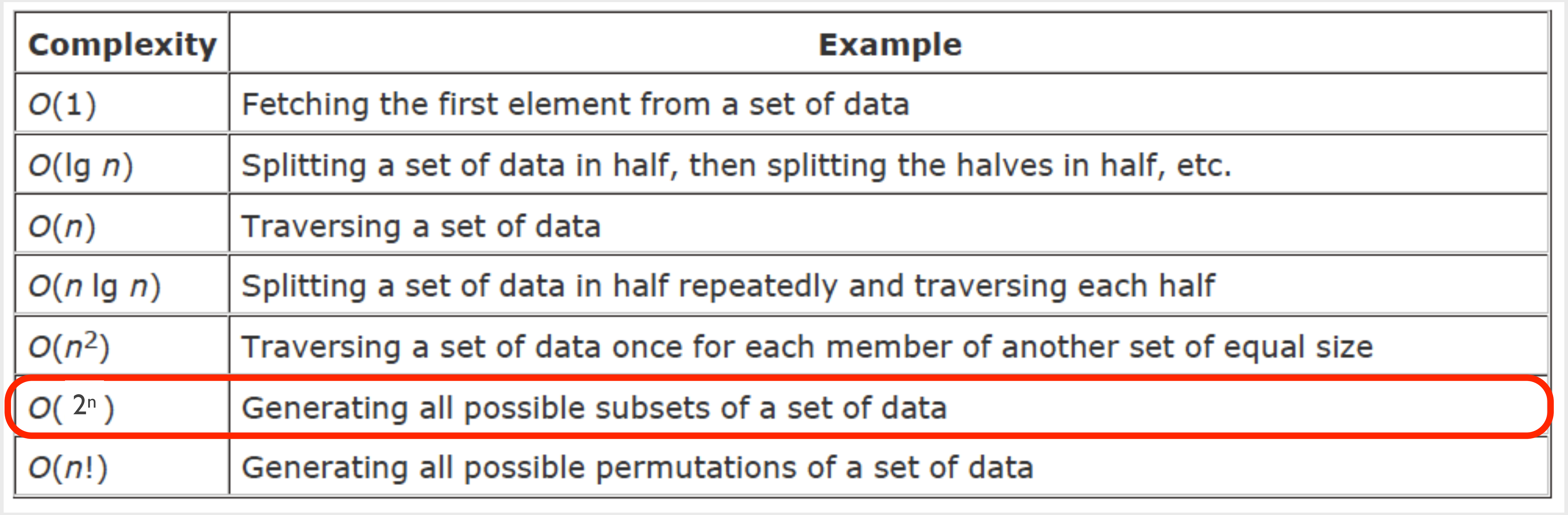
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

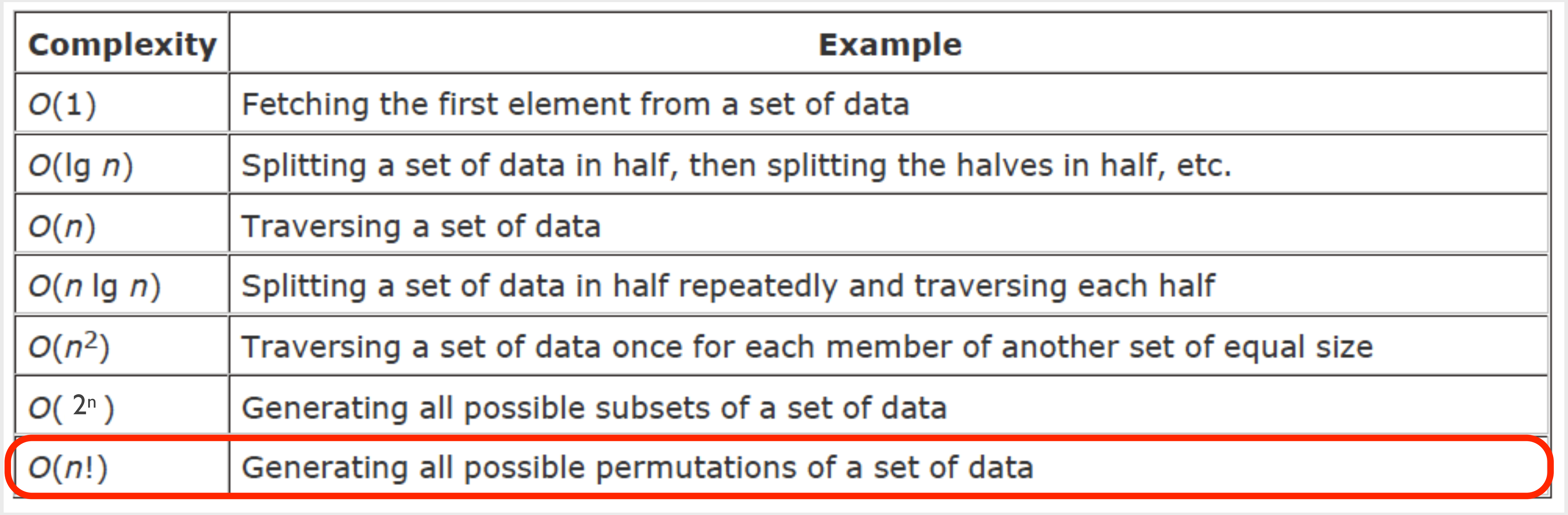
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

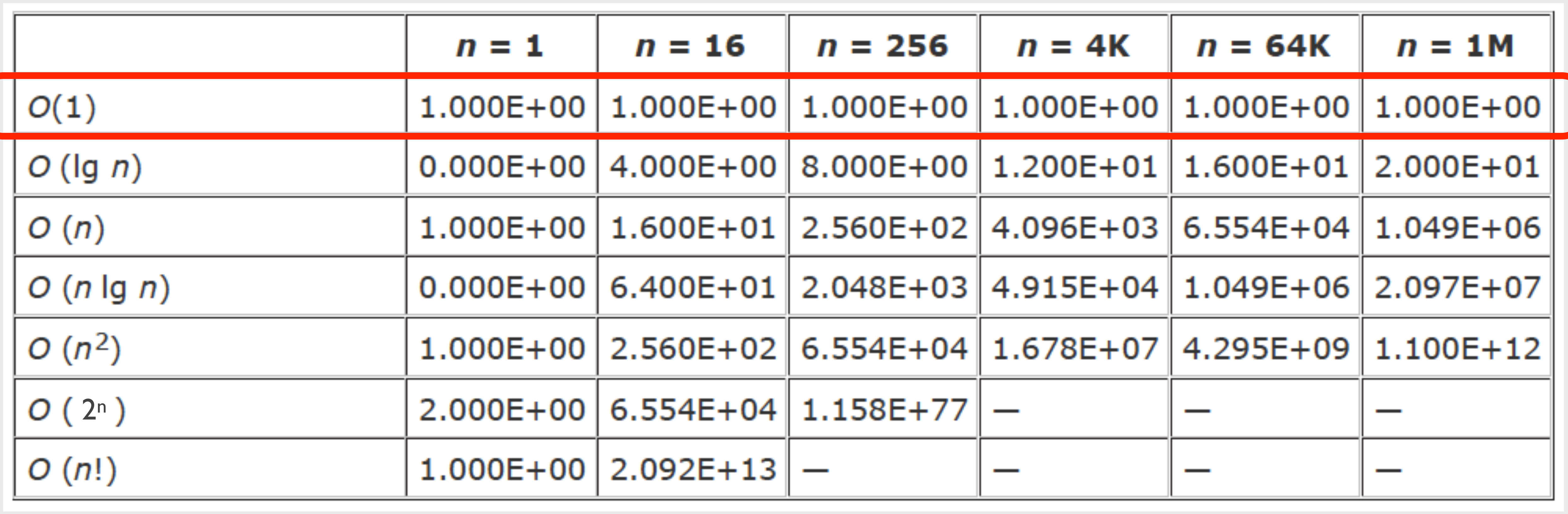
- Common Complexities Occur



Complexity	Example
$O(1)$	Fetching the first element from a set of data
$O(\lg n)$	Splitting a set of data in half, then splitting the halves in half, etc.
$O(n)$	Traversing a set of data
$O(n \lg n)$	Splitting a set of data in half repeatedly and traversing each half
$O(n^2)$	Traversing a set of data once for each member of another set of equal size
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data

Computational Complexity

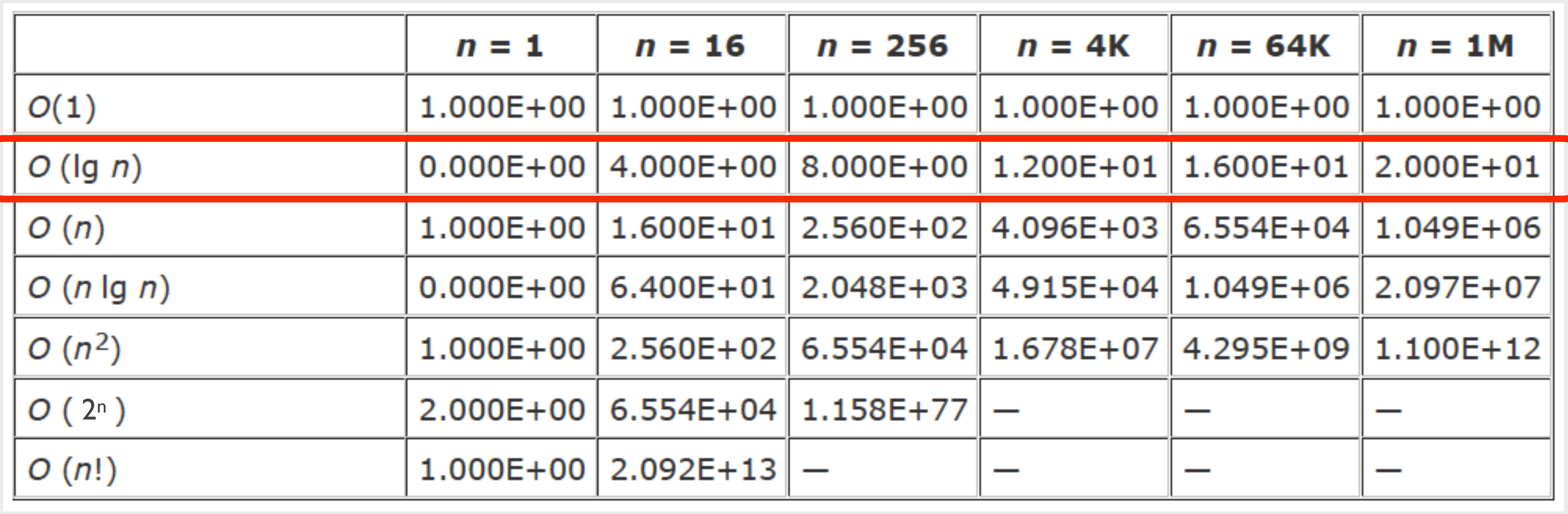
- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

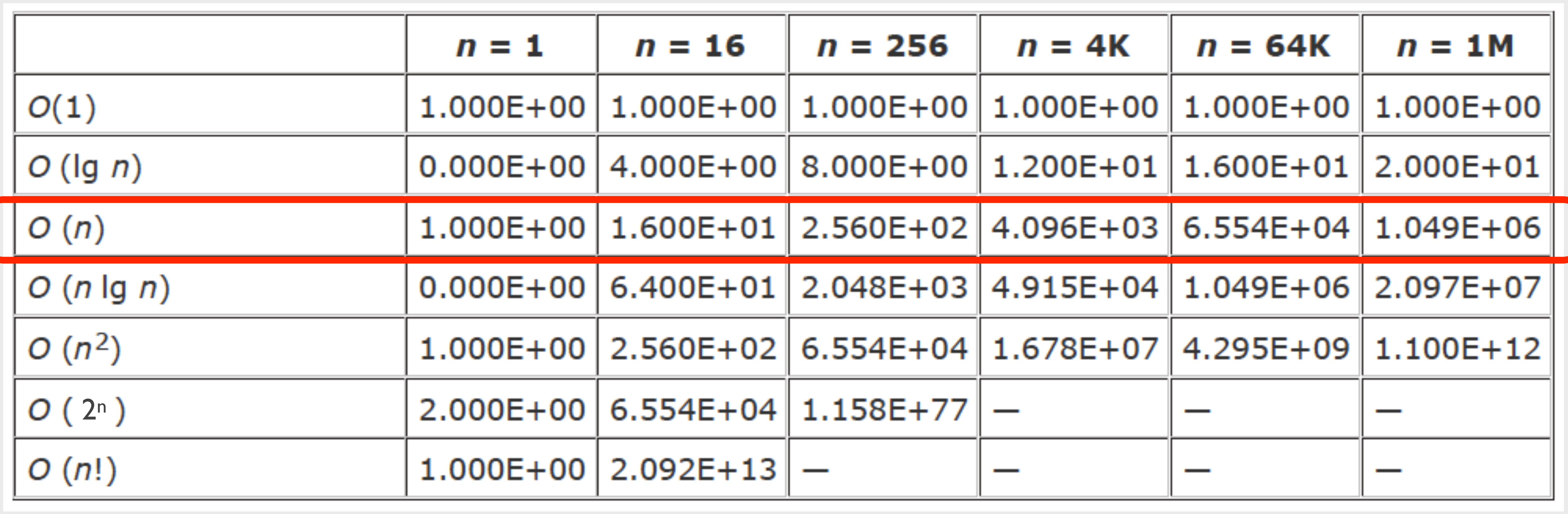
- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

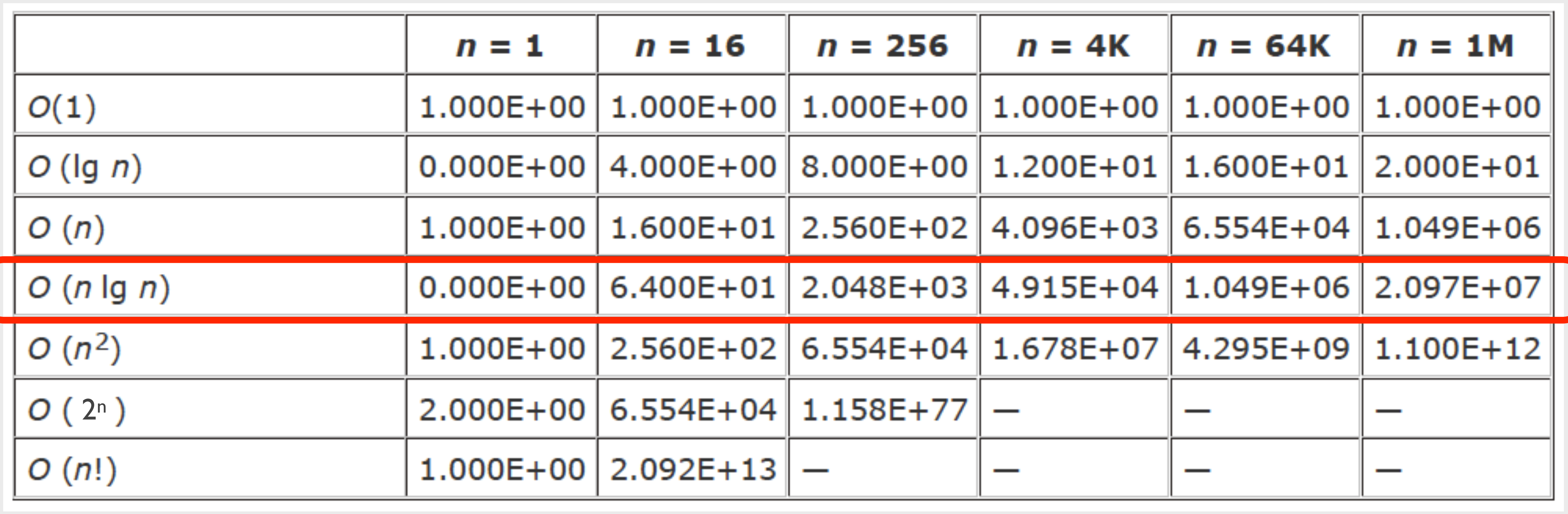
- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

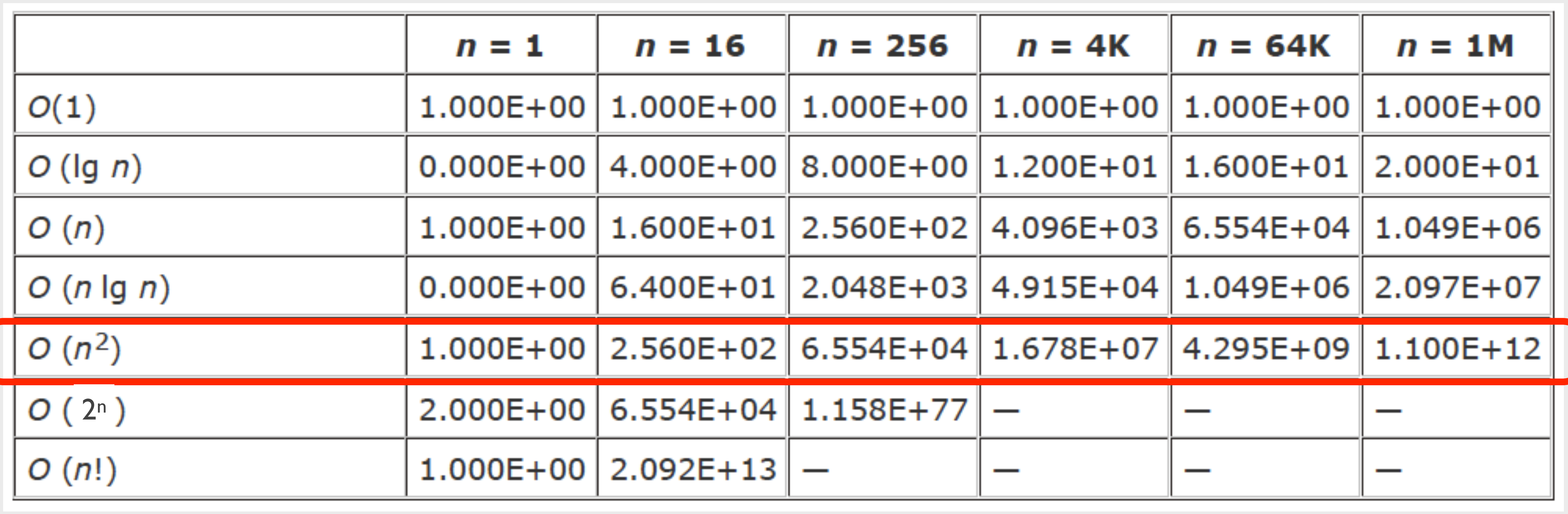
- The growth rate of the common complexities



	$n = 1$	$n = 16$	$n = 256$	$n = 4K$	$n = 64K$	$n = 1M$
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

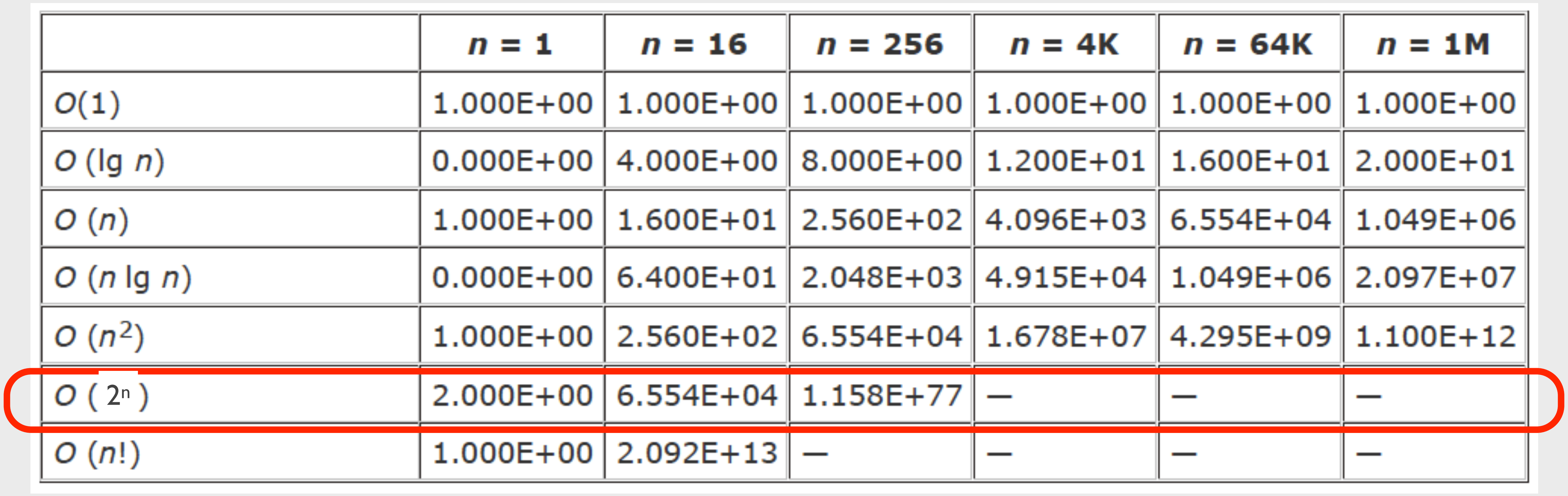
- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

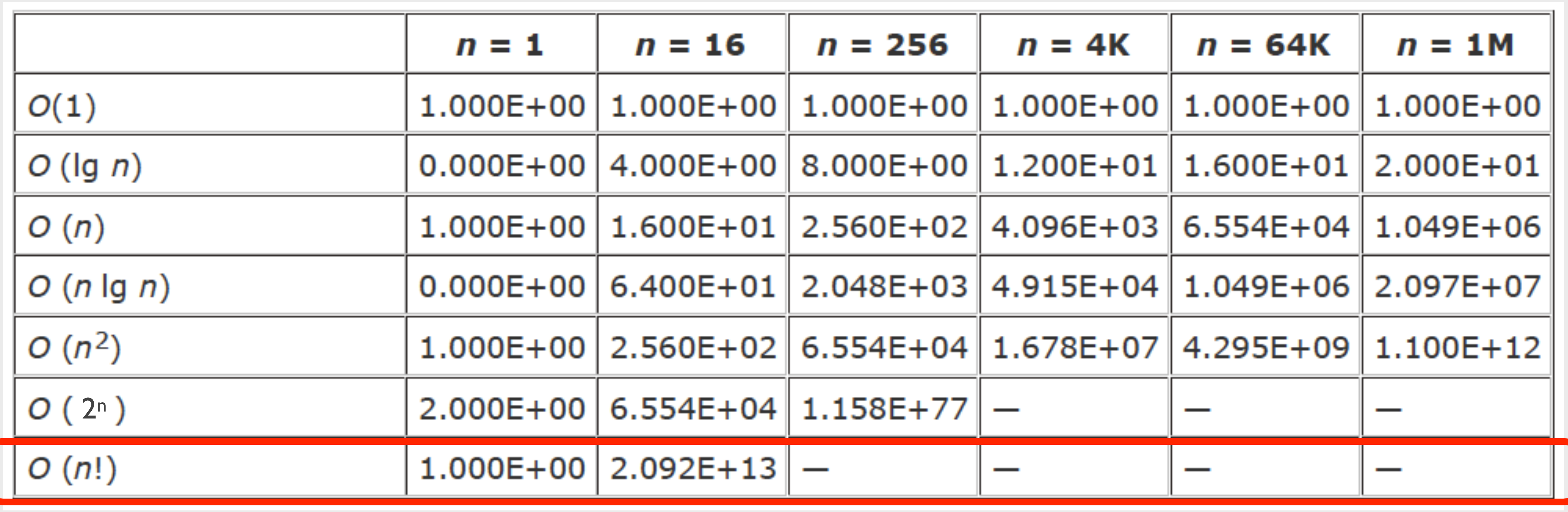
- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

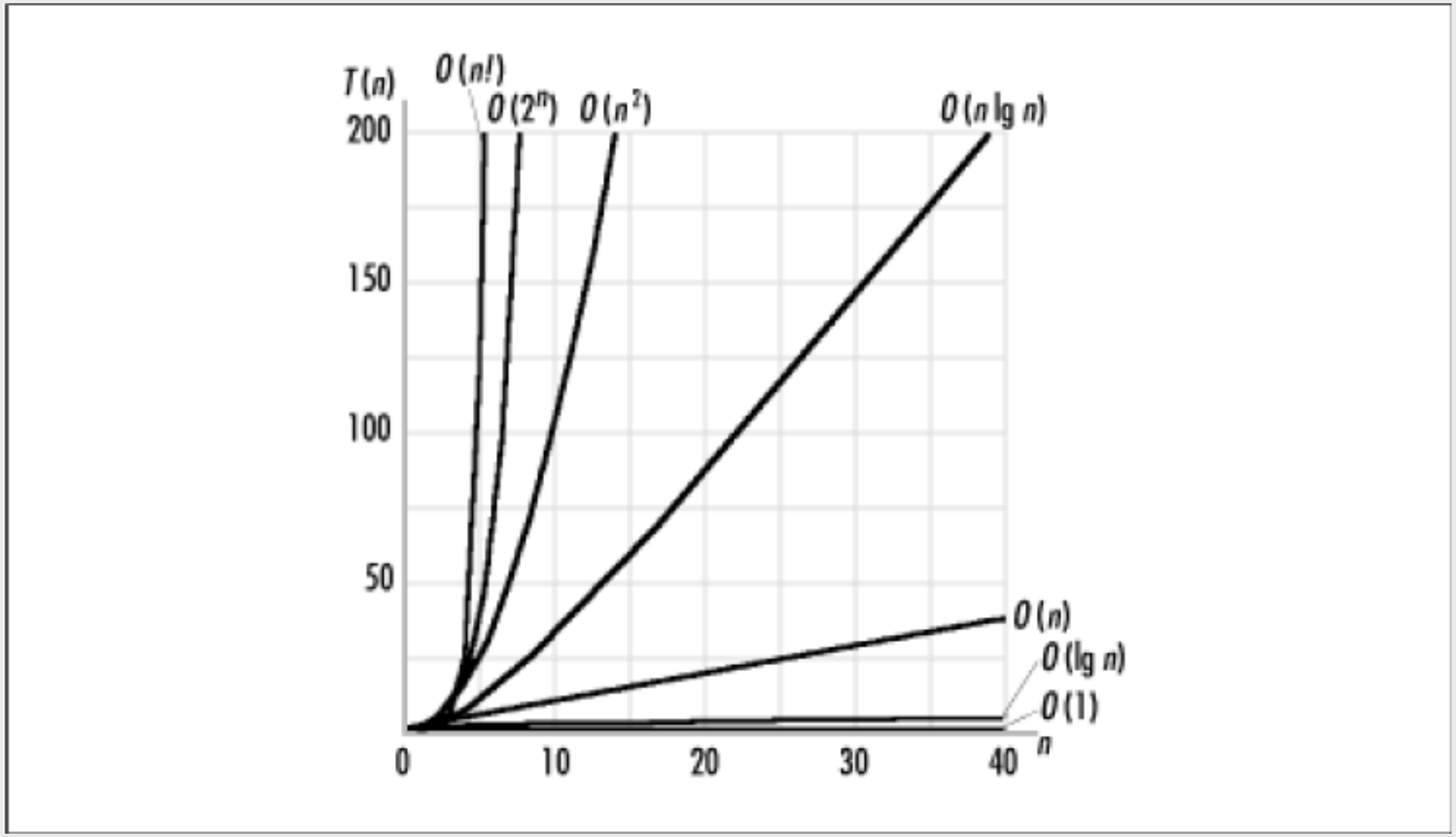
Computational Complexity

- The growth rate of the common complexities



	n = 1	n = 16	n = 256	n = 4K	n = 64K	n = 1M
$O(1)$	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00	1.000E+00
$O(\lg n)$	0.000E+00	4.000E+00	8.000E+00	1.200E+01	1.600E+01	2.000E+01
$O(n)$	1.000E+00	1.600E+01	2.560E+02	4.096E+03	6.554E+04	1.049E+06
$O(n \lg n)$	0.000E+00	6.400E+01	2.048E+03	4.915E+04	1.049E+06	2.097E+07
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

Computational Complexity

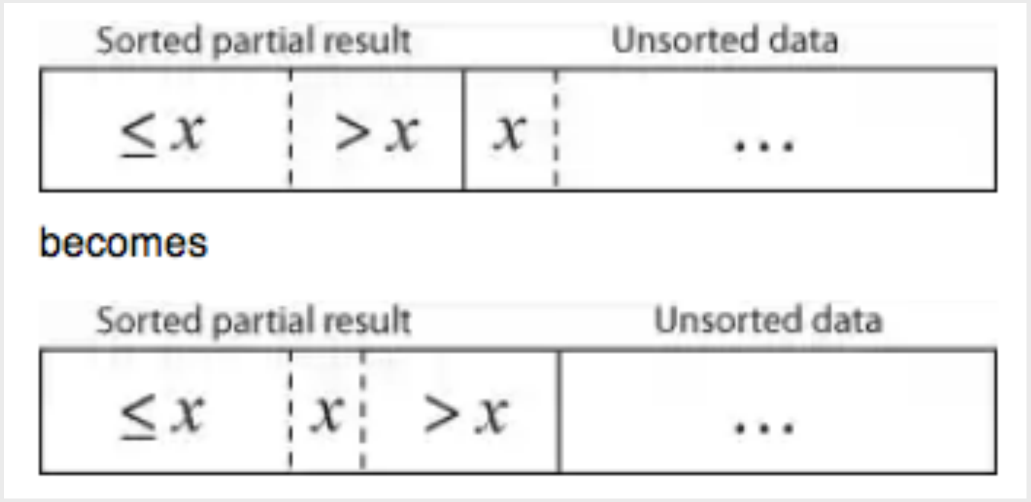


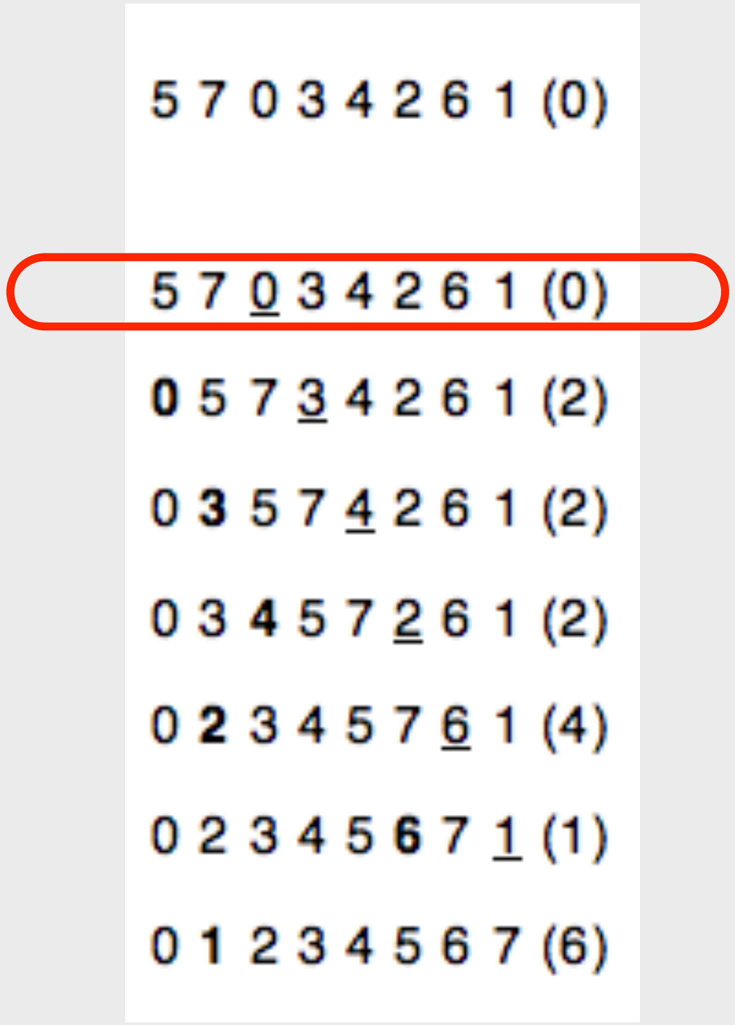
Computational Complexity

- Remarks
  - efficient vs. inefficient algorithms
  - some problems are intractable, so there are no "efficient" solutions => NP-complete problems
  - when two algorithms are of the same complexity, it may be worthwhile to consider their less significant terms and factors

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation





5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 **3** 5 7 4 2 6 1 (2)

0 3 **4** 5 7 2 6 1 (2)

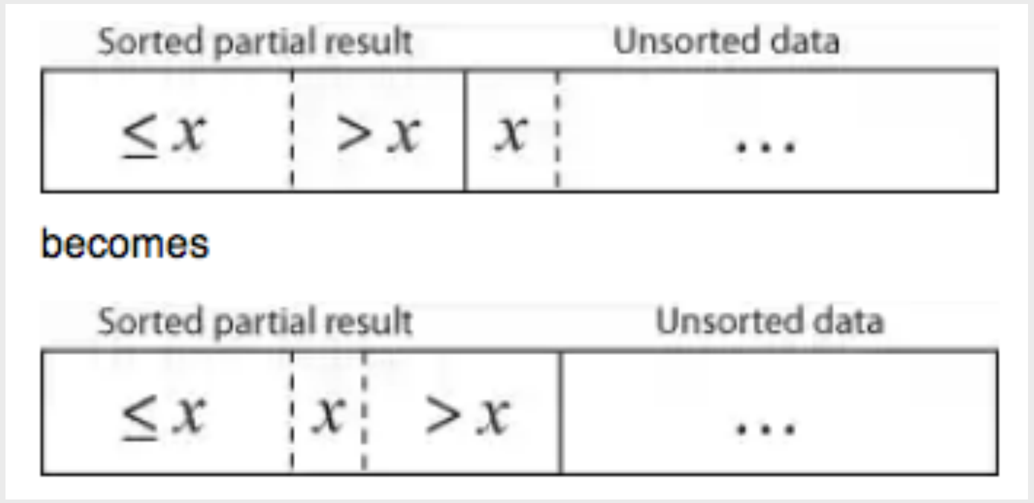
0 **2** 3 4 5 7 6 1 (4)

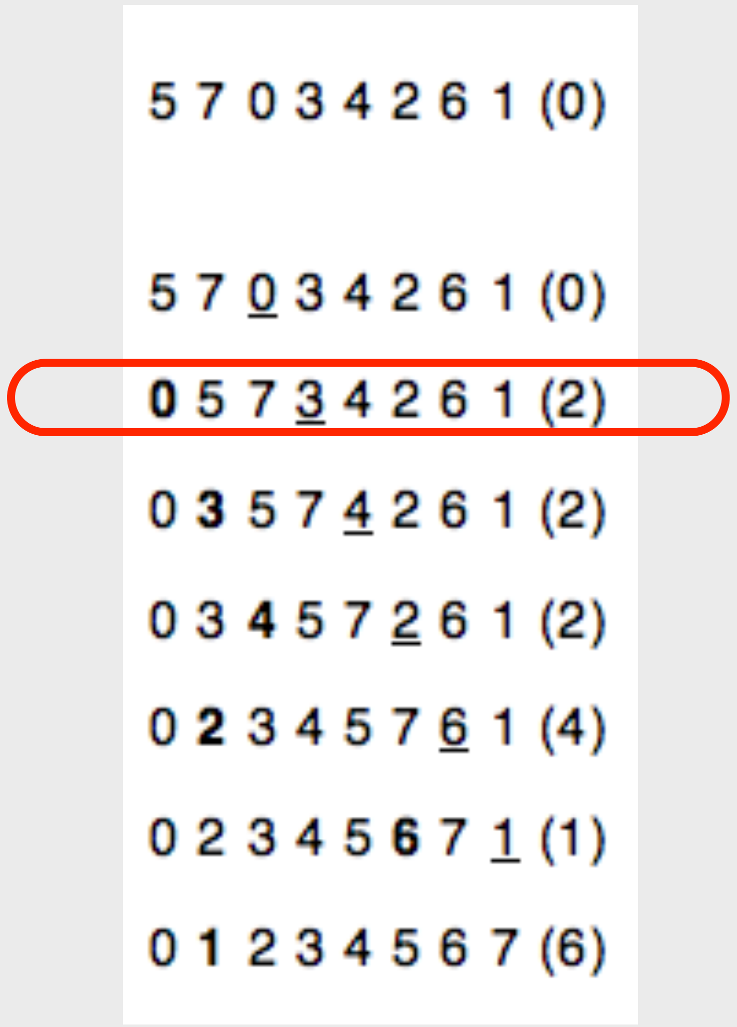
0 2 3 4 5 **6** 7 1 (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation





5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 **3** 5 7 4 2 6 1 (2)

0 3 **4** 5 7 2 6 1 (2)

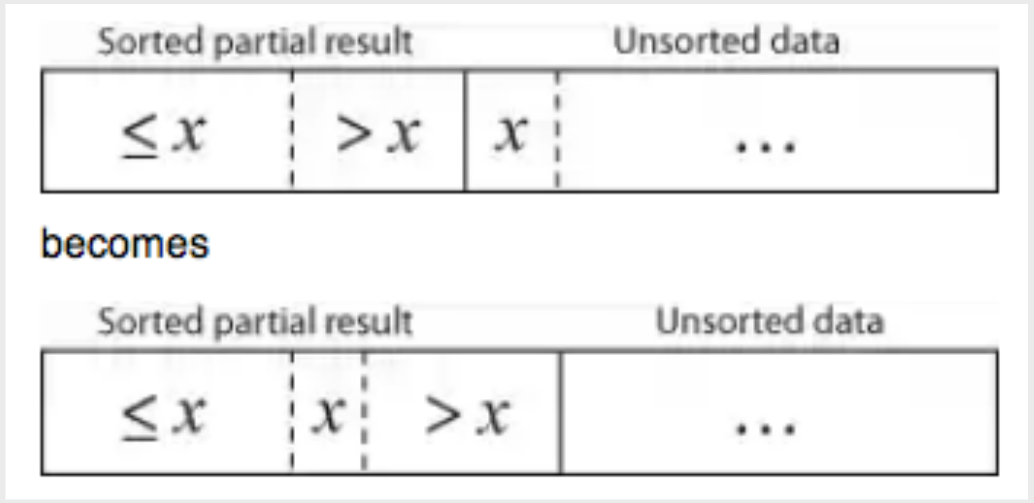
0 **2** 3 4 5 7 6 1 (4)

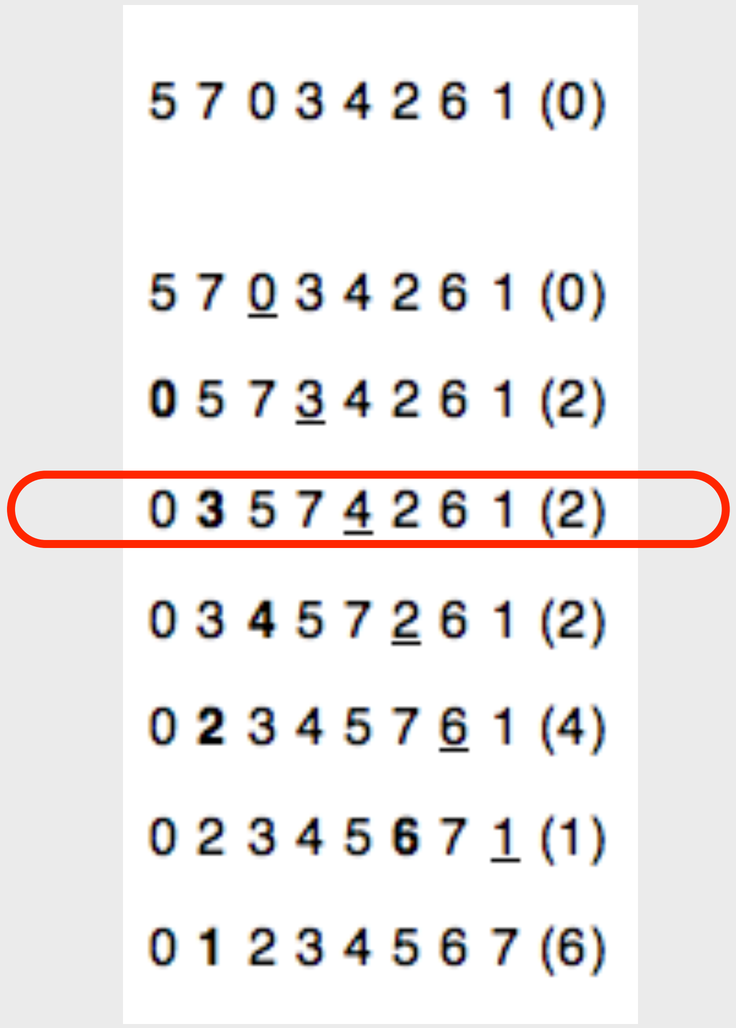
0 2 3 4 5 **6** 7 1 (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation

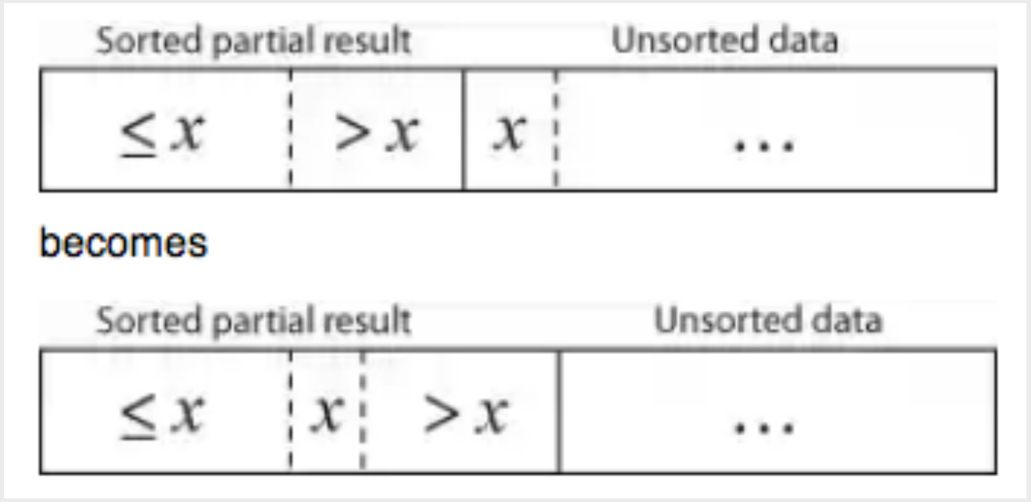


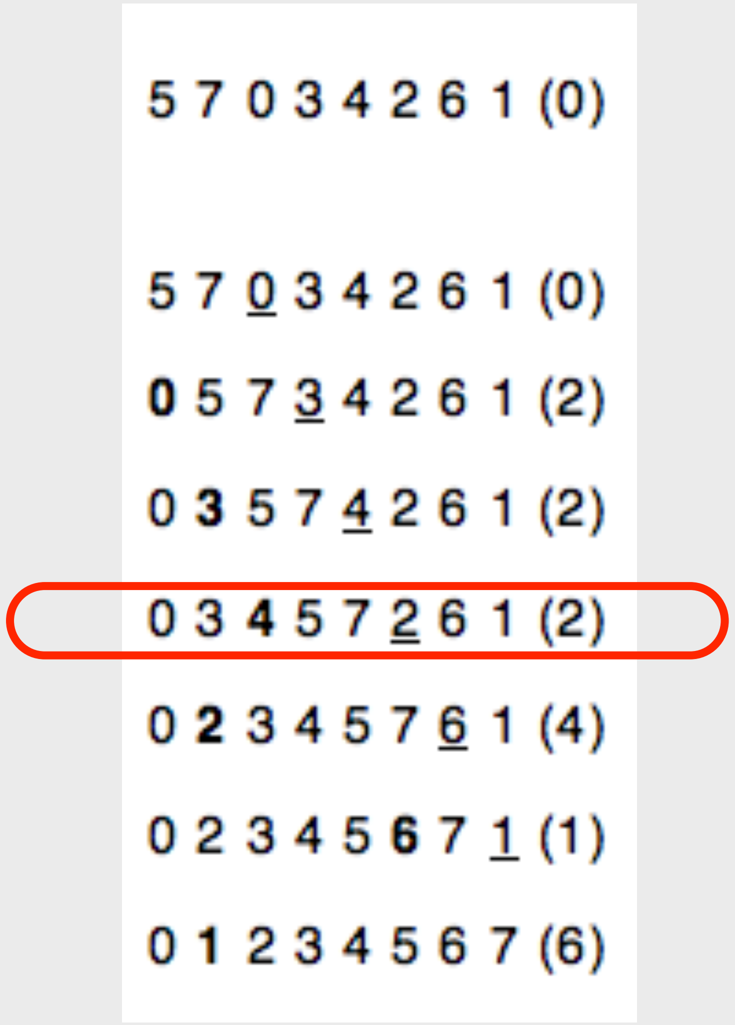


```
5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)
0 5 7 3 4 2 6 1 (2)
0 3 5 7 4 2 6 1 (2)
0 3 4 5 7 2 6 1 (2)
0 2 3 4 5 7 6 1 (4)
0 2 3 4 5 6 7 1 (1)
0 1 2 3 4 5 6 7 (6)
```

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation





5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 **3** 5 7 4 2 6 1 (2)

0 3 **4** 5 7 2 6 1 (2)

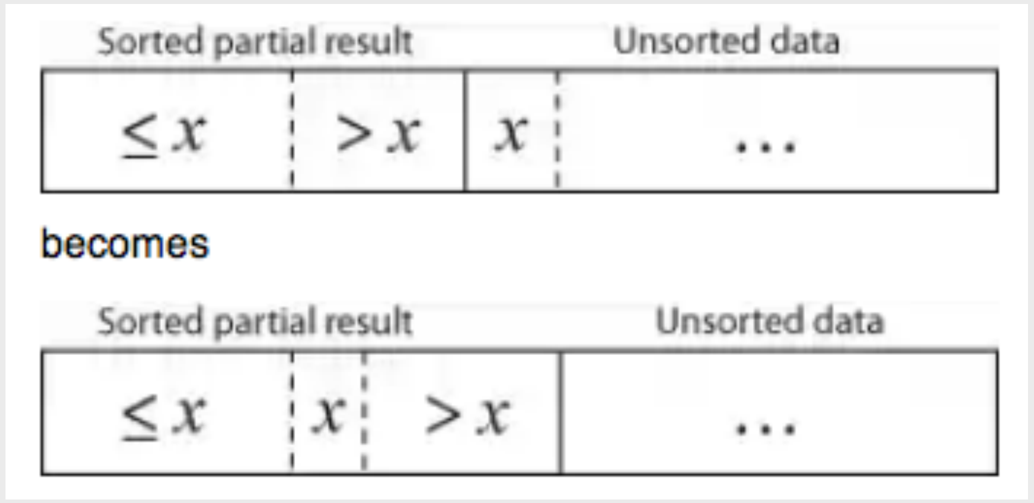
0 **2** 3 4 5 7 6 1 (4)

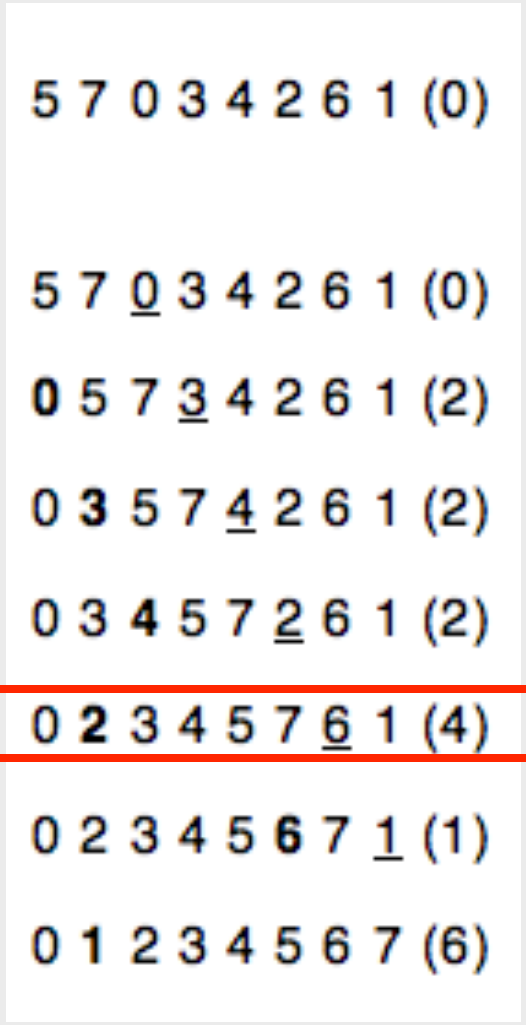
0 2 3 4 5 **6** 7 1 (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation





5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 **3** 5 7 4 2 6 1 (2)

0 3 **4** 5 7 2 6 1 (2)

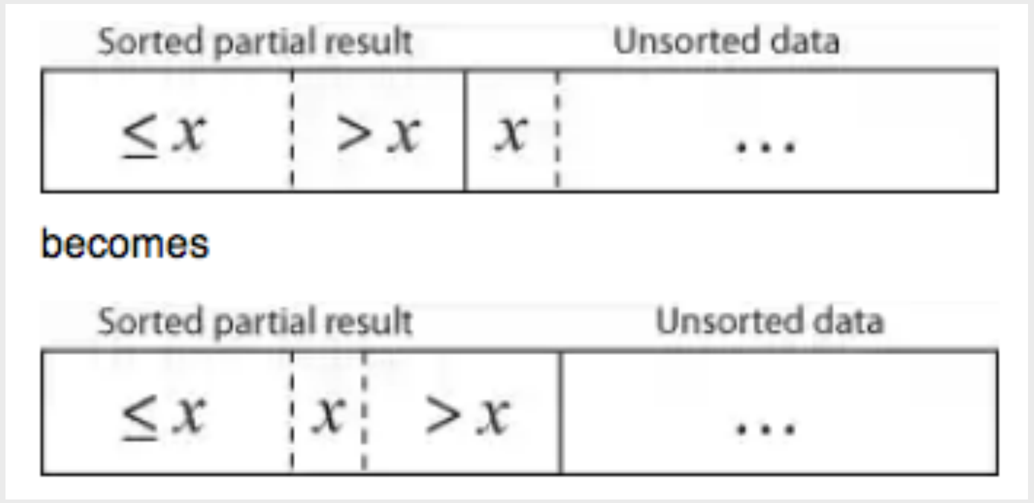
0 **2** 3 4 5 7 6 1 (4)

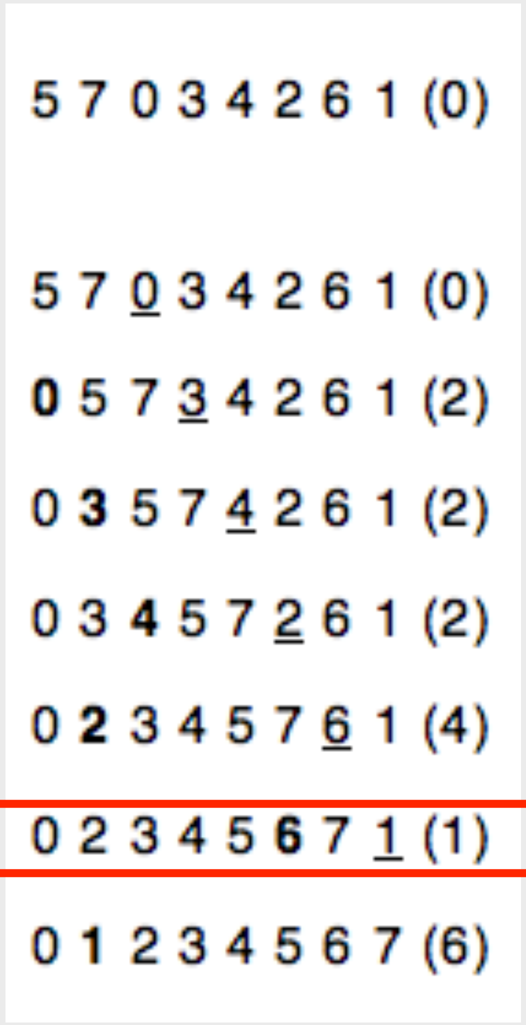
0 2 3 4 5 **6** 7 1 (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

- Insertion Sort
  - Animation





5 7 0 3 4 2 6 1 (0)

5 7 $\underline{0}$ 3 4 2 6 1 (0)

0 5 7 $\underline{3}$ 4 2 6 1 (2)

0 **3** 5 7 $\underline{4}$ 2 6 1 (2)

0 3 **4** 5 7 $\underline{2}$ 6 1 (2)

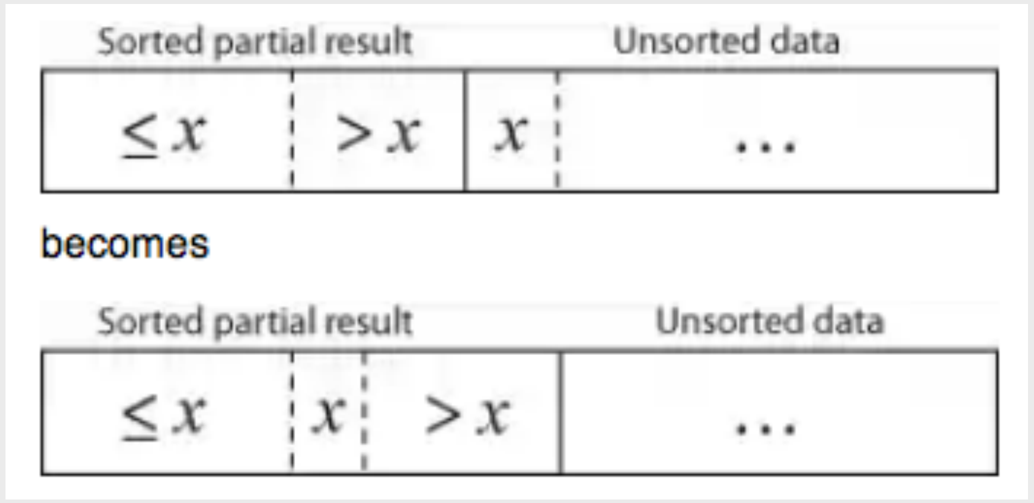
0 **2** 3 4 5 7 $\underline{6}$ 1 (4)

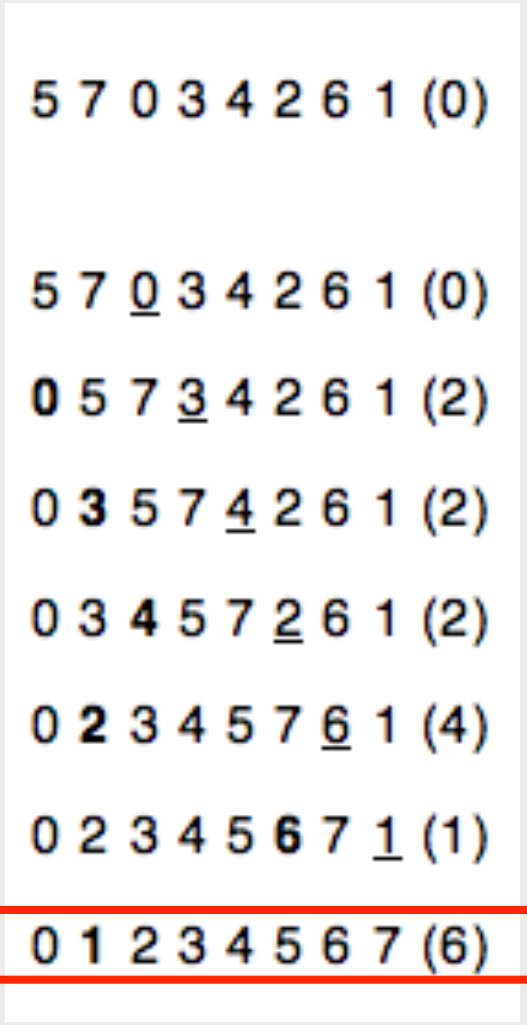
0 2 3 4 5 **6** 7 $\underline{1}$ (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

- Insertion Sort
    - Animation





5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 **3** 5 7 4 2 6 1 (2)

0 3 **4** 5 7 2 6 1 (2)

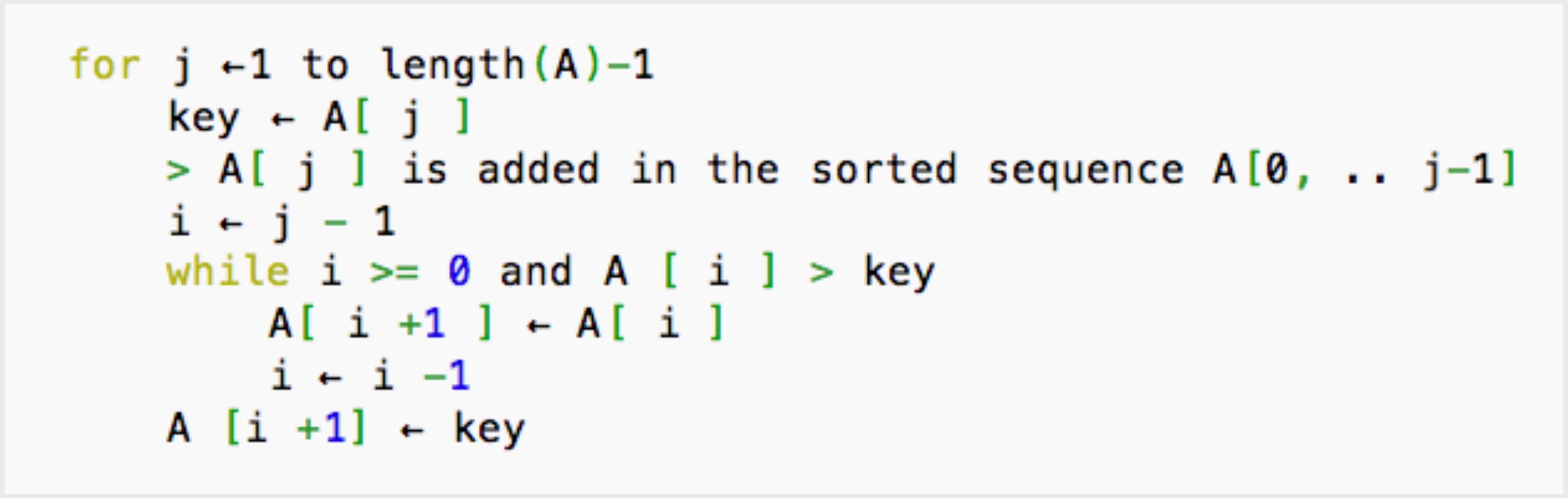
0 **2** 3 4 5 7 6 1 (4)

0 2 3 4 5 **6** 7 1 (1)

0 **1** 2 3 4 5 6 7 (6)

Analysis Example: Insertion Sort

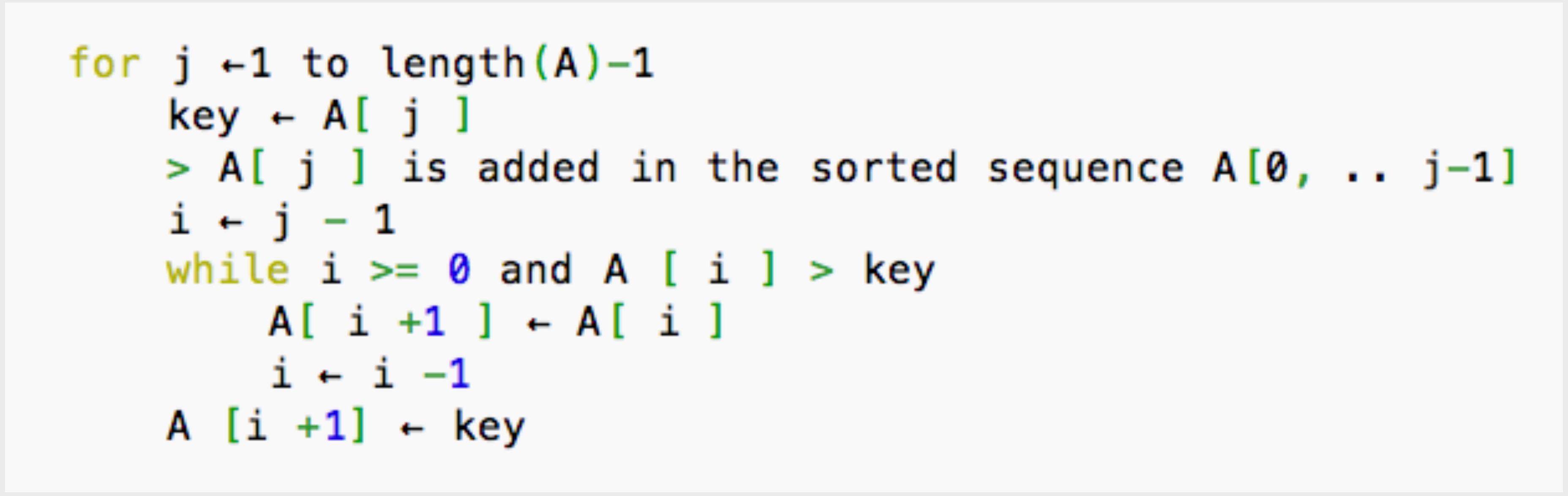
- Pseudocode



```
for j ←1 to length(A)-1
    key ← A[ j ]
    > A[ j ] is added in the sorted sequence A[0, .. j-1]
    i ← j - 1
    while i >= 0 and A [ i ] > key
        A[ i +1 ] ← A[ i ]
        i ← i -1
    A [i +1] ← key
```

Analysis Example: Insertion Sort

- Pseudocode

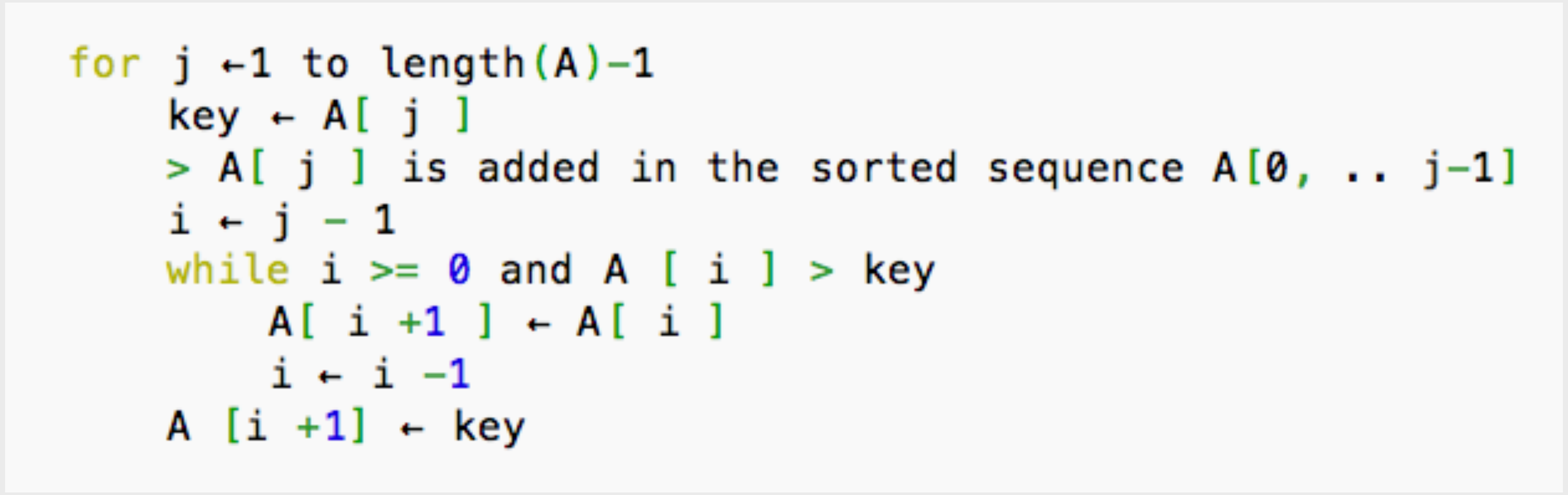


```
for j ←1 to length(A)-1
    key ← A[ j ]
    > A[ j ] is added in the sorted sequence A[0, .. j-1]
    i ← j - 1
    while i >= 0 and A [ i ] > key
        A[ i +1 ] ← A[ i ]
        i ← i -1
    A [i +1] ← key
```

Analysis

Analysis Example: Insertion Sort

- Pseudocode



```
for j ←1 to length(A)-1
    key ← A[ j ]
    > A[ j ] is added in the sorted sequence A[0, .. j-1]
    i ← j - 1
    while i >= 0 and A [ i ] > key
        A[ i +1 ] ← A[ i ]
        i ← i -1
    A [i +1] ← key
```

Analysis

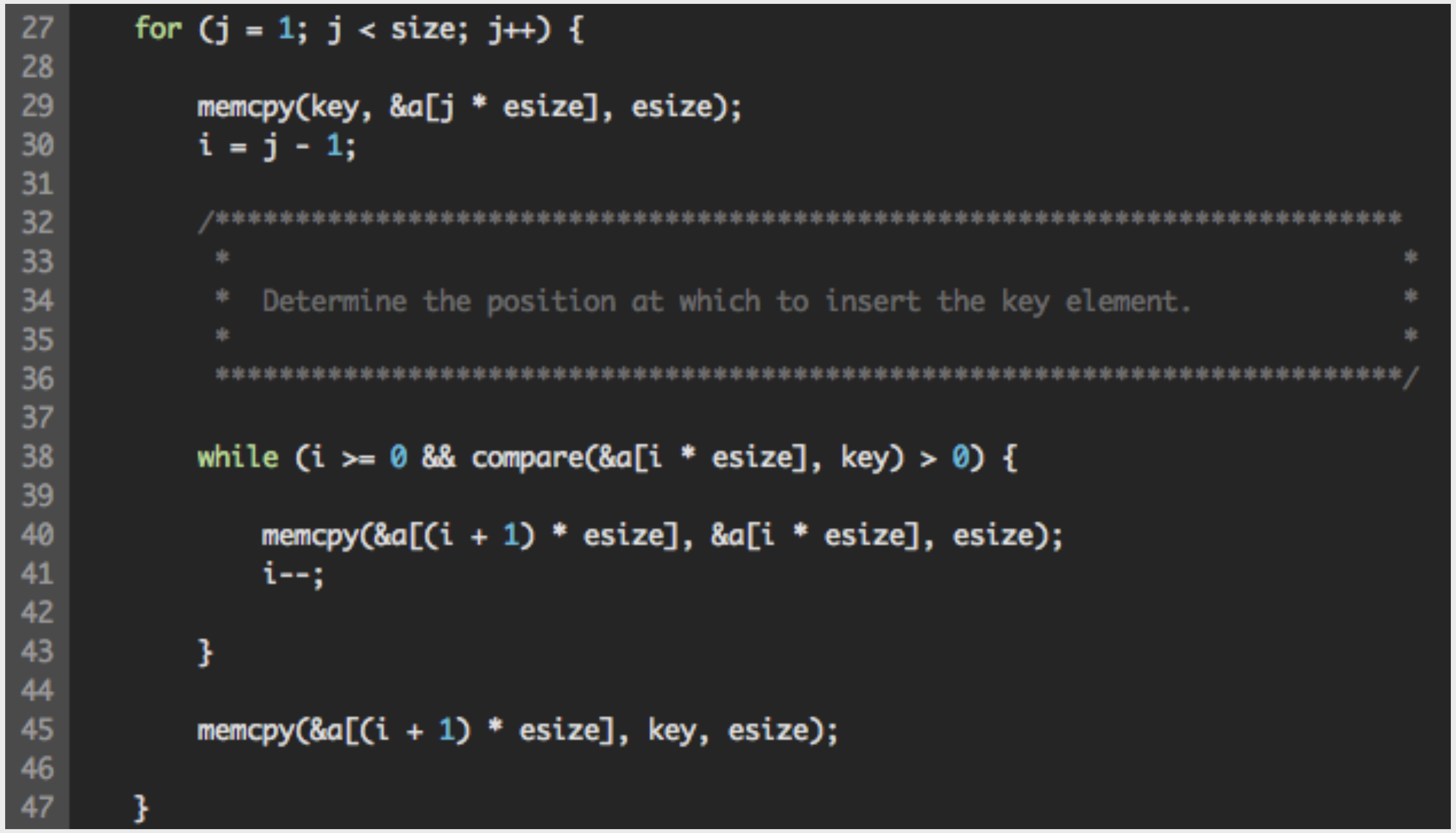
$$O(T(n)) = O(1+2+\cdots+(n-1)) = O(\frac{n(n-1)}{2}) = O(\frac{n^2}{2}) = O(n^2)$$

Analysis Example: Insertion Sort

- Example: issort/example.c

Analysis Example: Insertion Sort

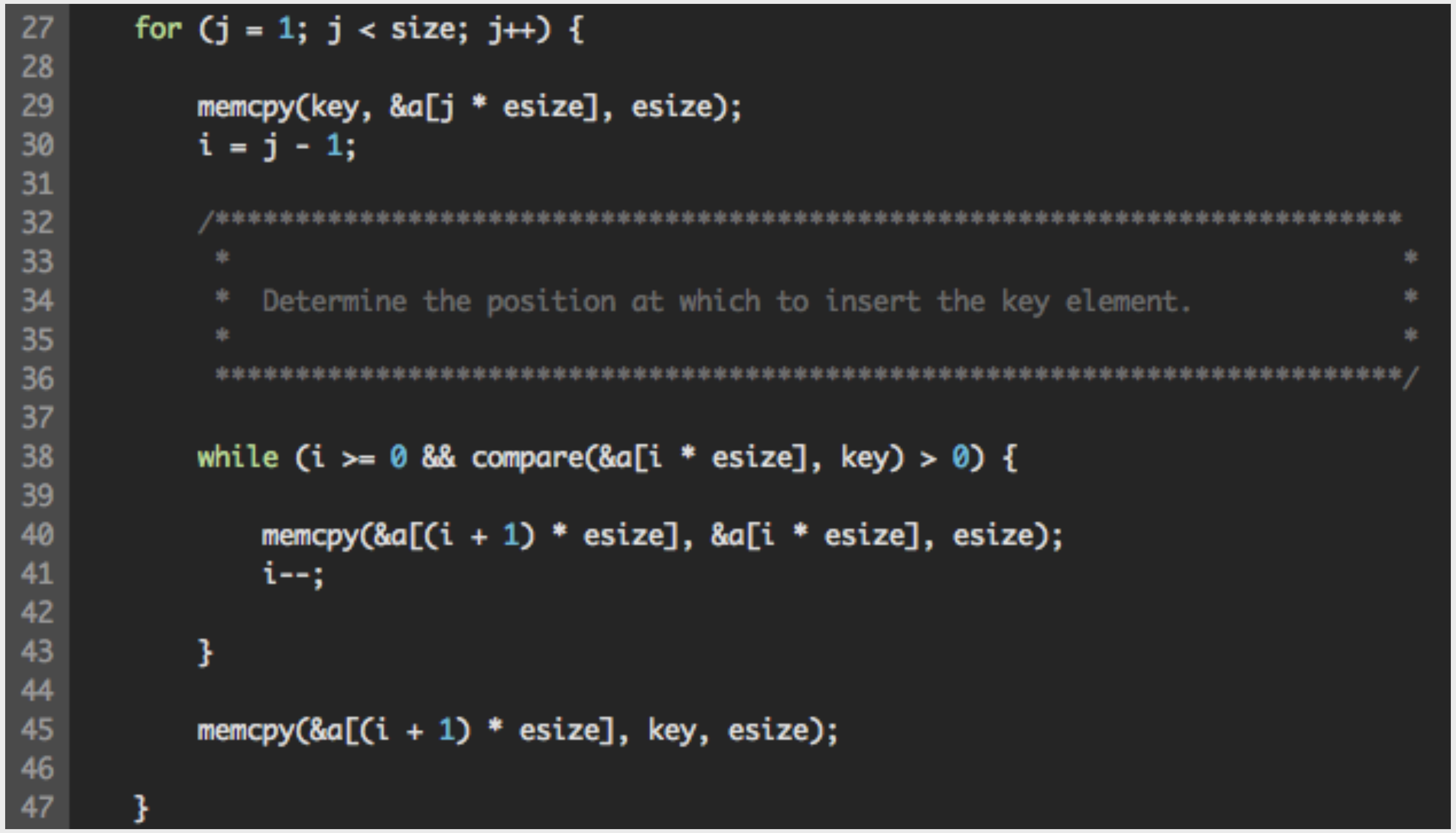
- Example: issort/example.c



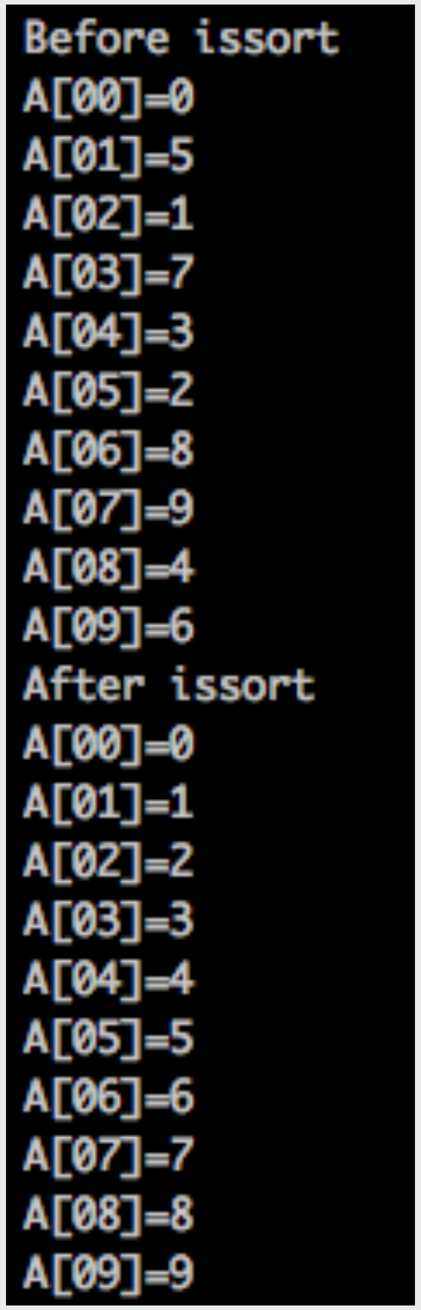
```
for (j = 1; j < size; j++) {

    memcpy(key, &a[j * esize], esize);
    i = j - 1;

    /*****************************************************************************
     *                                                                           *
     *  Determine the position at which to insert the key element.              *
     *                                                                           *
     *****************************************************************************/

    while (i >= 0 && compare(&a[i * esize], key) > 0) {

        memcpy(&a[(i + 1) * esize], &a[i * esize], esize);
        i--;

    }

    memcpy(&a[(i + 1) * esize], key, esize);

}
```

Analysis Example: Insertion Sort

- Example: issort/example.c



```c
    for (j = 1; j < size; j++) {

        memcpy(key, &a[j * esize], esize);
        i = j - 1;

        /*****************************************************************************
         *                                                                           *
         *  Determine the position at which to insert the key element.               *
         *                                                                           *
         *****************************************************************************/

        while (i >= 0 && compare(&a[i * esize], key) > 0) {

            memcpy(&a[(i + 1) * esize], &a[i * esize], esize);
            i--;

        }

        memcpy(&a[(i + 1) * esize], key, esize);

    }
```

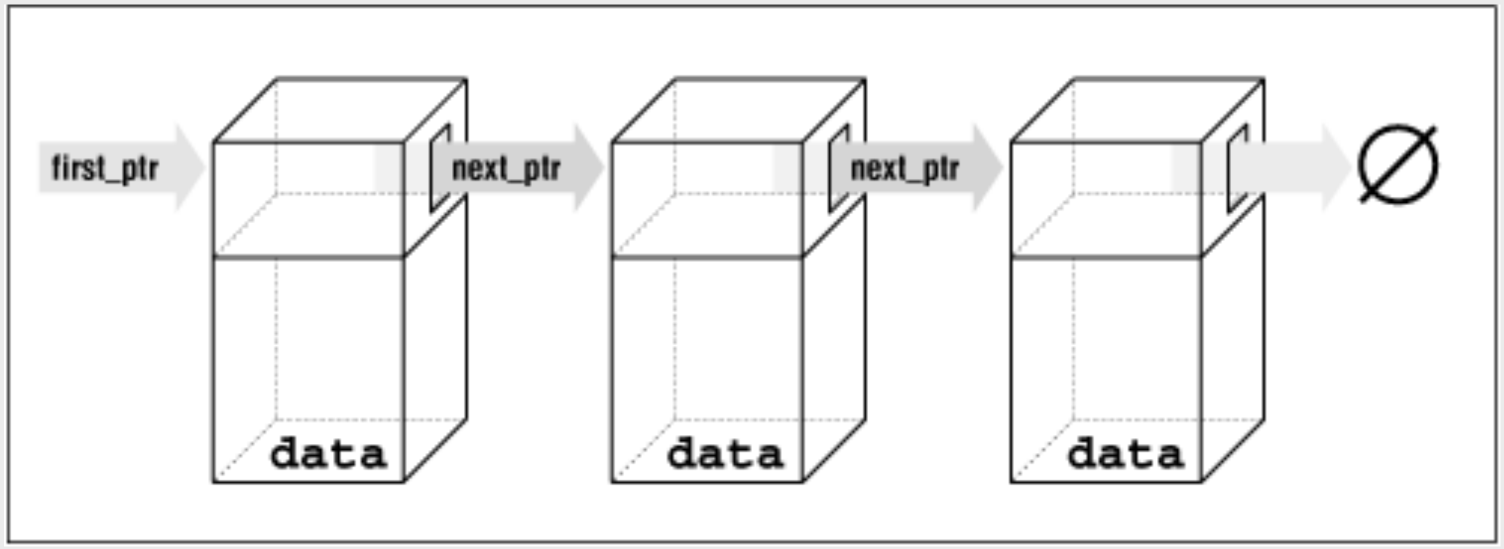


```
Before issort
A[00]=0
A[01]=5
A[02]=1
A[03]=7
A[04]=3
A[05]=2
A[06]=8
A[07]=9
A[08]=4
A[09]=6
After issort
A[00]=0
A[01]=1
A[02]=2
A[03]=3
A[04]=4
A[05]=5
A[06]=6
A[07]=7
A[08]=8
A[09]=9
```

Advanced Linked List

Linked List

- A *linked list* is a chain of items in which each item points to the next one in the chain



```
struct linked_list {
    char    data[30];               /* data in this element */
    struct linked_list *next_ptr; /* pointer to next element */
};
```

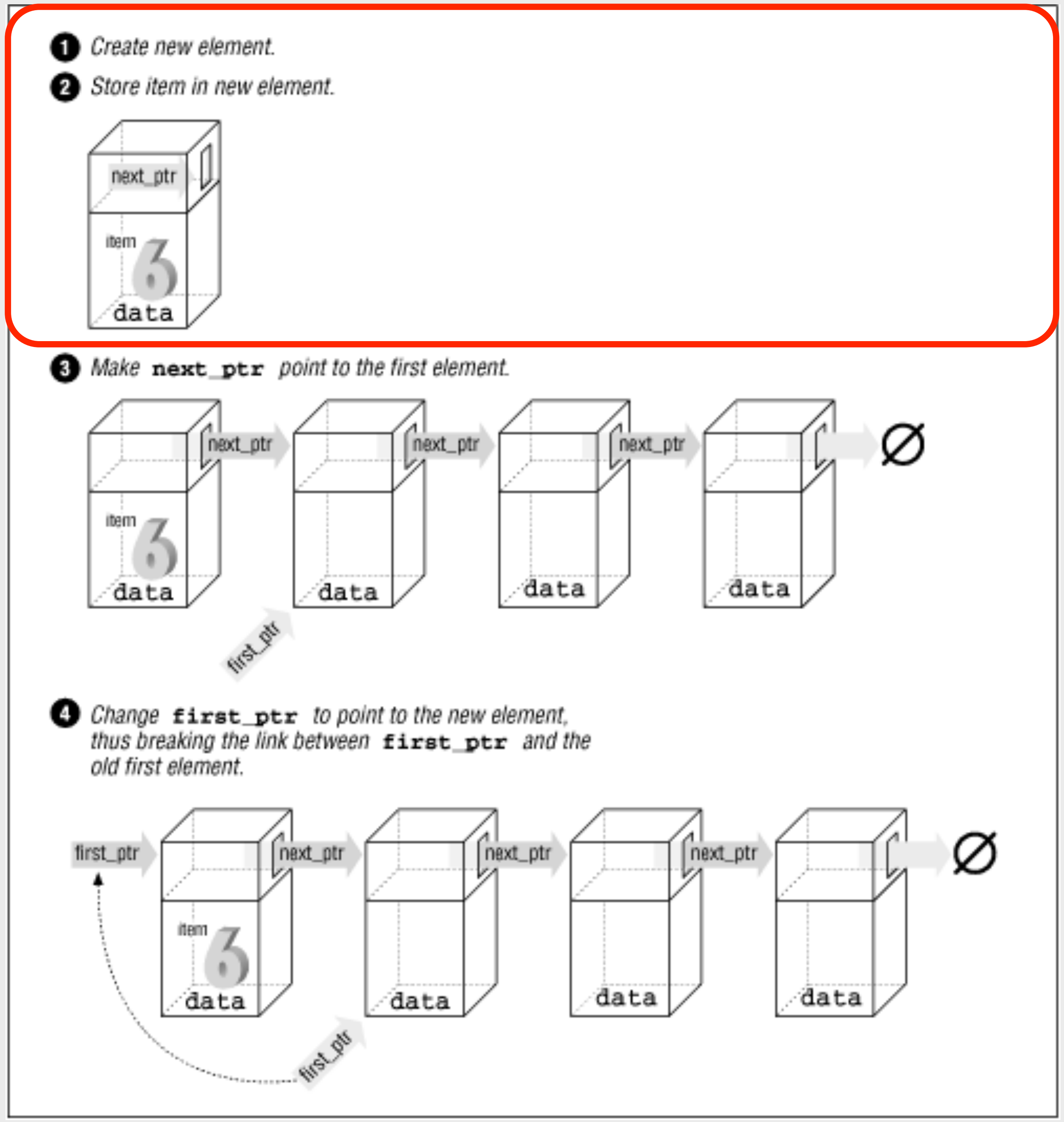
Linked List

- In the beginning, before we insert any elements into a list, the pointer is initialized to ***NULL***

```
struct linked_list *first_ptr = NULL;
```

Linked List

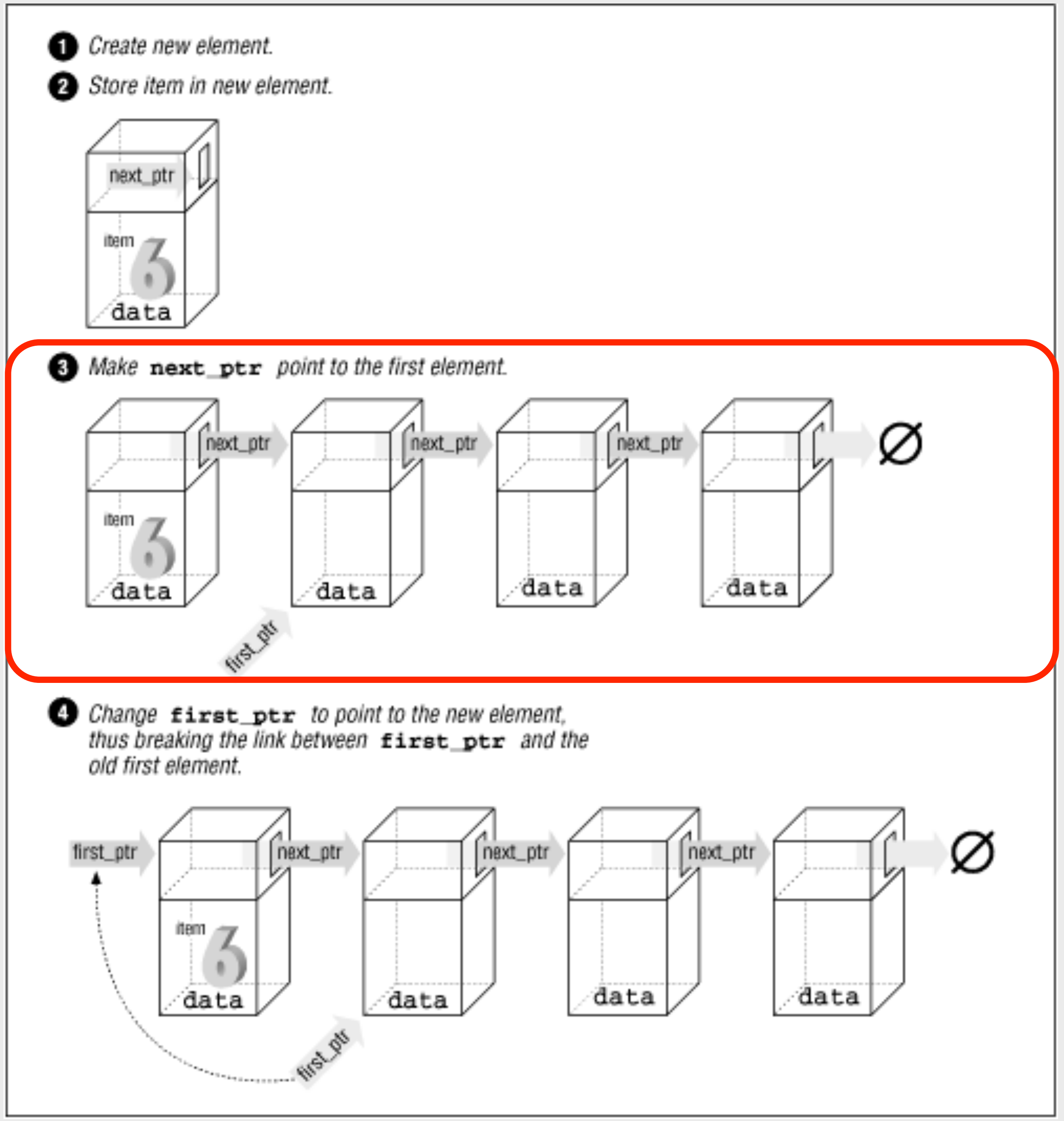
- Adding new element to the beginning of a list



❶ Create new element.
❷ Store item in new element.
next_ptr
item
6
data
❸ Make next_ptr point to the first element.
next_ptr
data
first_ptr
∅
❹ Change first_ptr to point to the new element, thus breaking the link between first_ptr and the old first element.
first_ptr
next_ptr
item
6
data
first_ptr
∅

Linked List

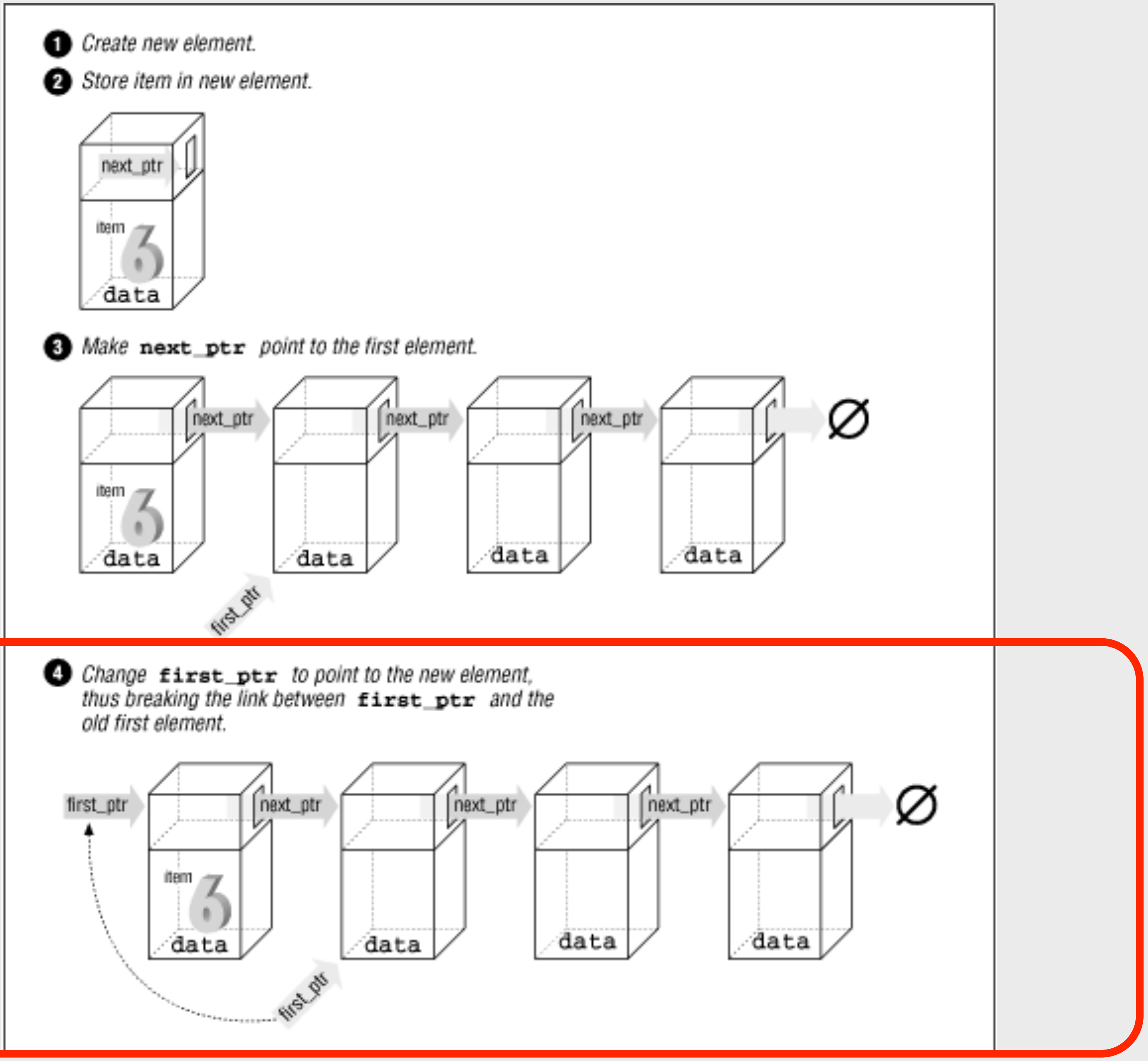
- Adding new element to the beginning of a list



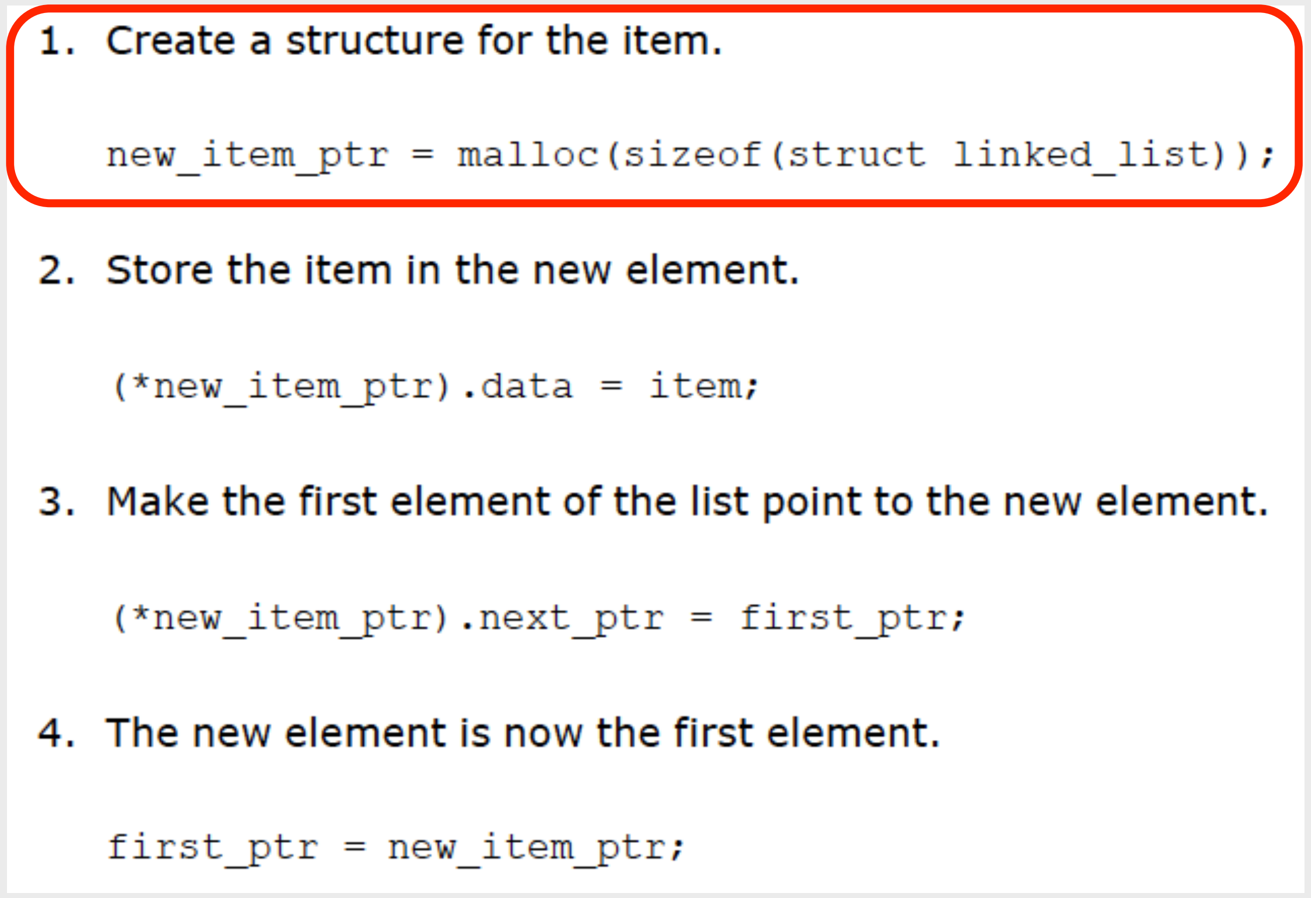
1 Create new element.
2 Store item in new element.
next_ptr
item
6
data
3 Make next_ptr point to the first element.
next_ptr
item
6
data
first_ptr
Ø
4 Change first_ptr to point to the new element, thus breaking the link between first_ptr and the old first element.
first_ptr
next_ptr
item
6
data
first_ptr
Ø

Linked List

- Adding new element to the beginning of a list


❶ Create new element.
❷ Store item in new element.
next_ptr
item
6
data
❸ Make next_ptr point to the first element.
next_ptr
item
6
data
next_ptr
data
next_ptr
data
data
Ø
first_ptr
❹ Change first_ptr to point to the new element, thus breaking the link between first_ptr and the old first element.
first_ptr
next_ptr
item
6
data
next_ptr
data
next_ptr
data
data
Ø
first_ptr

Linked List



1. Create a structure for the item.

   ```
   new_item_ptr = malloc(sizeof(struct linked_list));
   ```

2. Store the item in the new element.

   ```
   (*new_item_ptr).data = item;
   ```

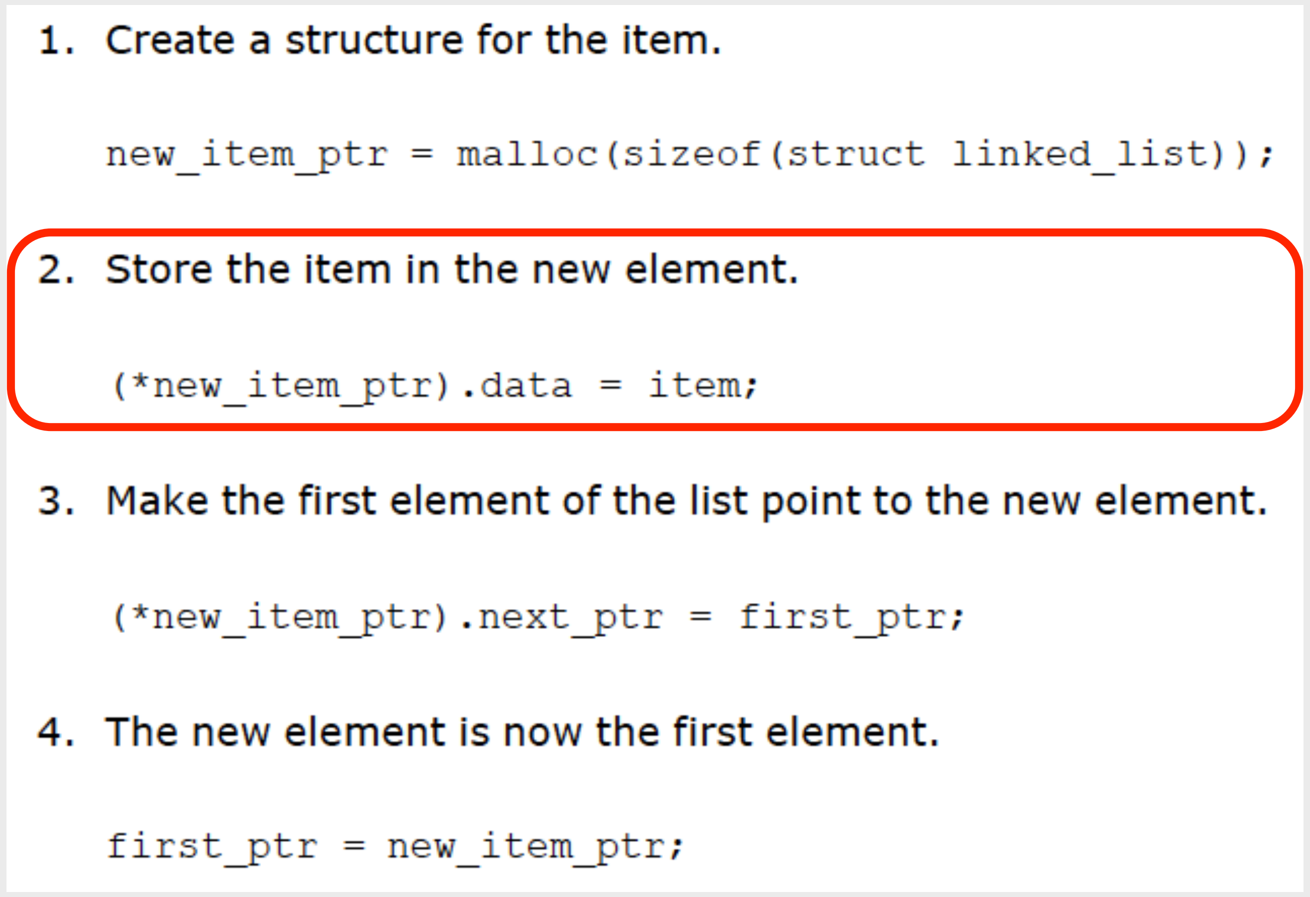
3. Make the first element of the list point to the new element.

   ```
   (*new_item_ptr).next_ptr = first_ptr;
   ```

4. The new element is now the first element.

   ```
   first_ptr = new_item_ptr;
   ```

Linked List



1. Create a structure for the item.

```
new_item_ptr = malloc(sizeof(struct linked_list));
```

2. Store the item in the new element.

```
(*new_item_ptr).data = item;
```

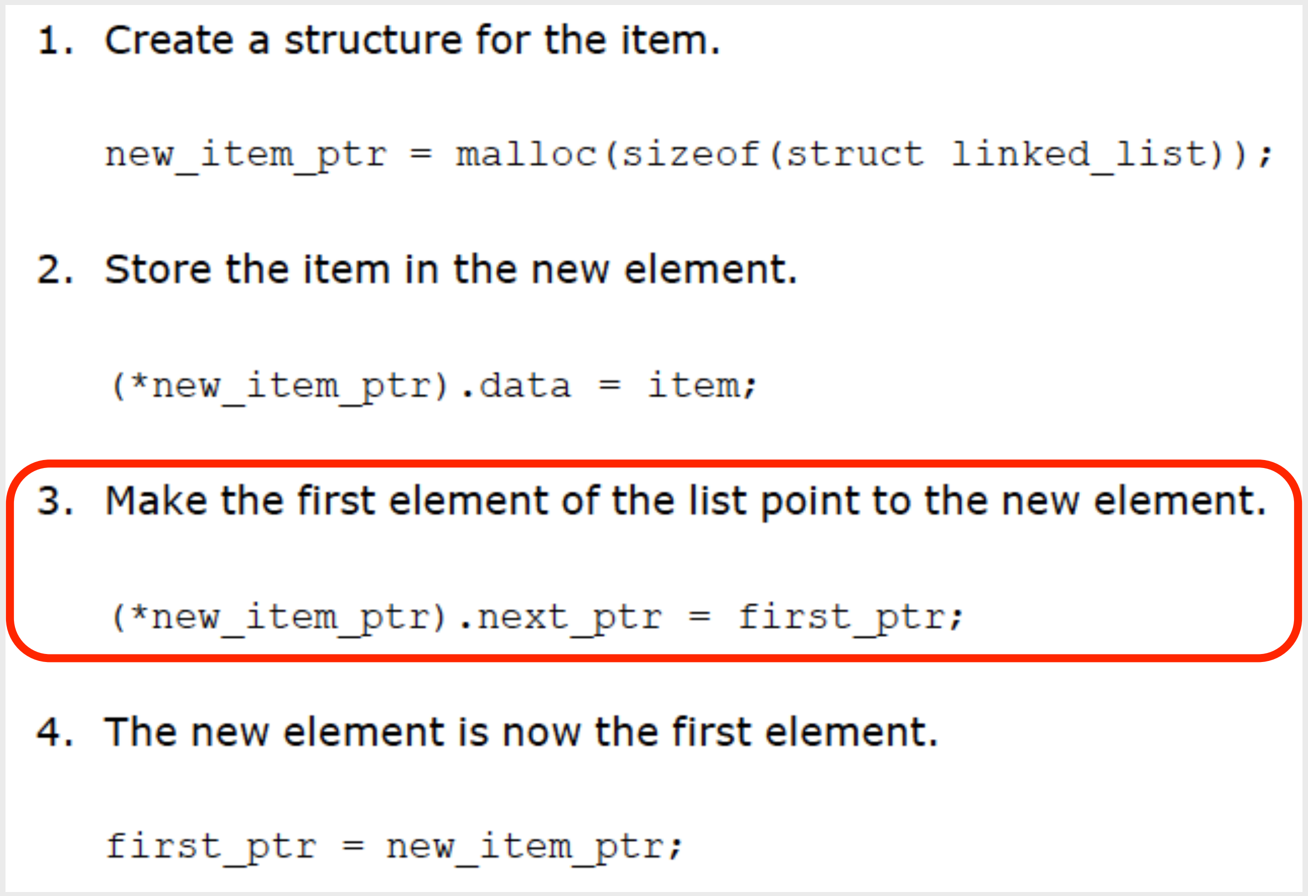
3. Make the first element of the list point to the new element.

```
(*new_item_ptr).next_ptr = first_ptr;
```

4. The new element is now the first element.

```
first_ptr = new_item_ptr;
```

Linked List



1. Create a structure for the item.

```
new_item_ptr = malloc(sizeof(struct linked_list));
```

2. Store the item in the new element.

```
(*new_item_ptr).data = item;
```

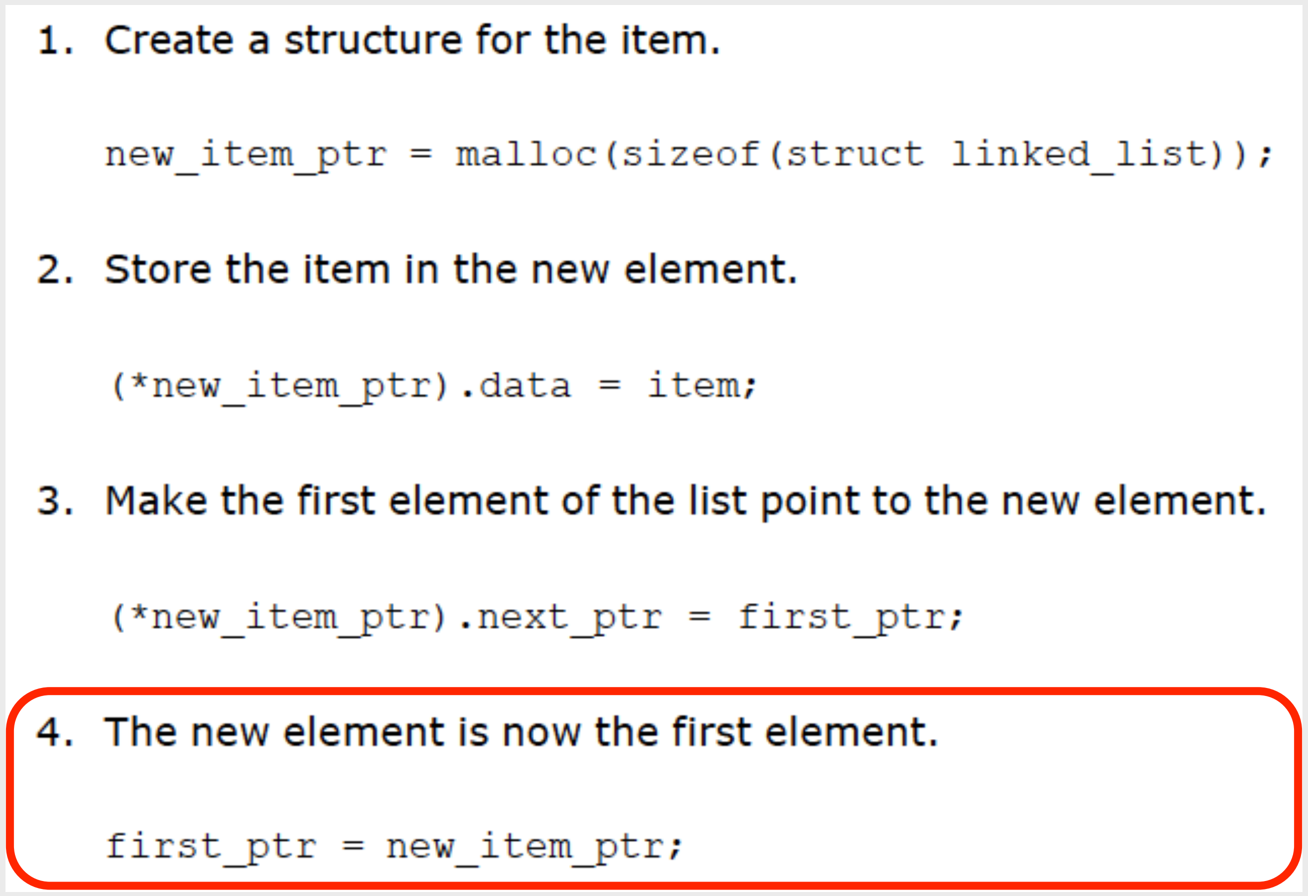
3. Make the first element of the list point to the new element.

```
(*new_item_ptr).next_ptr = first_ptr;
```

4. The new element is now the first element.

```
first_ptr = new_item_ptr;
```

Linked List



1. Create a structure for the item.

   ```
   new_item_ptr = malloc(sizeof(struct linked_list));
   ```

2. Store the item in the new element.

   ```
   (*new_item_ptr).data = item;
   ```

3. Make the first element of the list point to the new element.

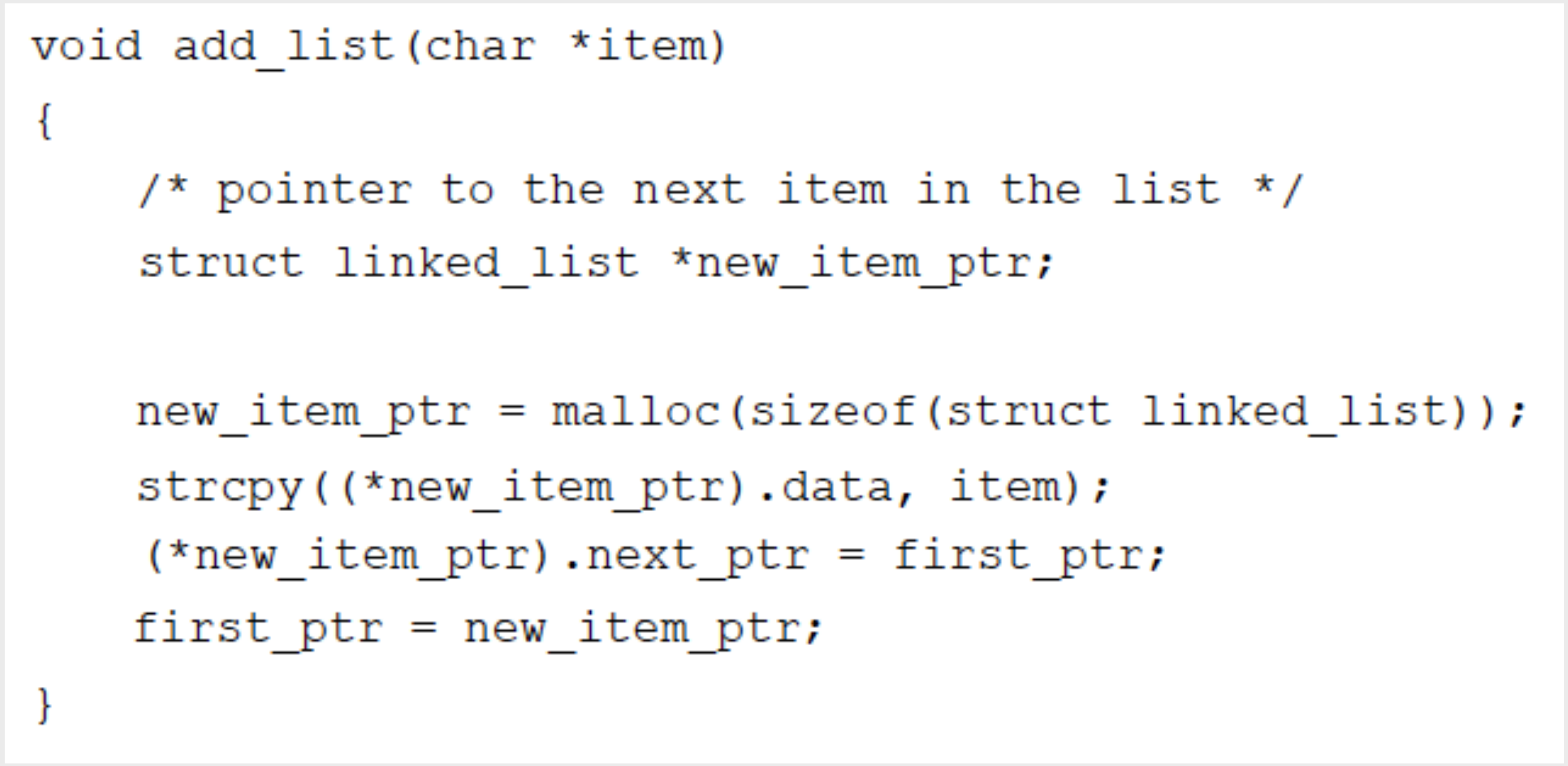
   ```
   (*new_item_ptr).next_ptr = first_ptr;
   ```

4. The new element is now the first element.

   ```
   first_ptr = new_item_ptr;
   ```

Linked List

- Example code of adding a node into a list



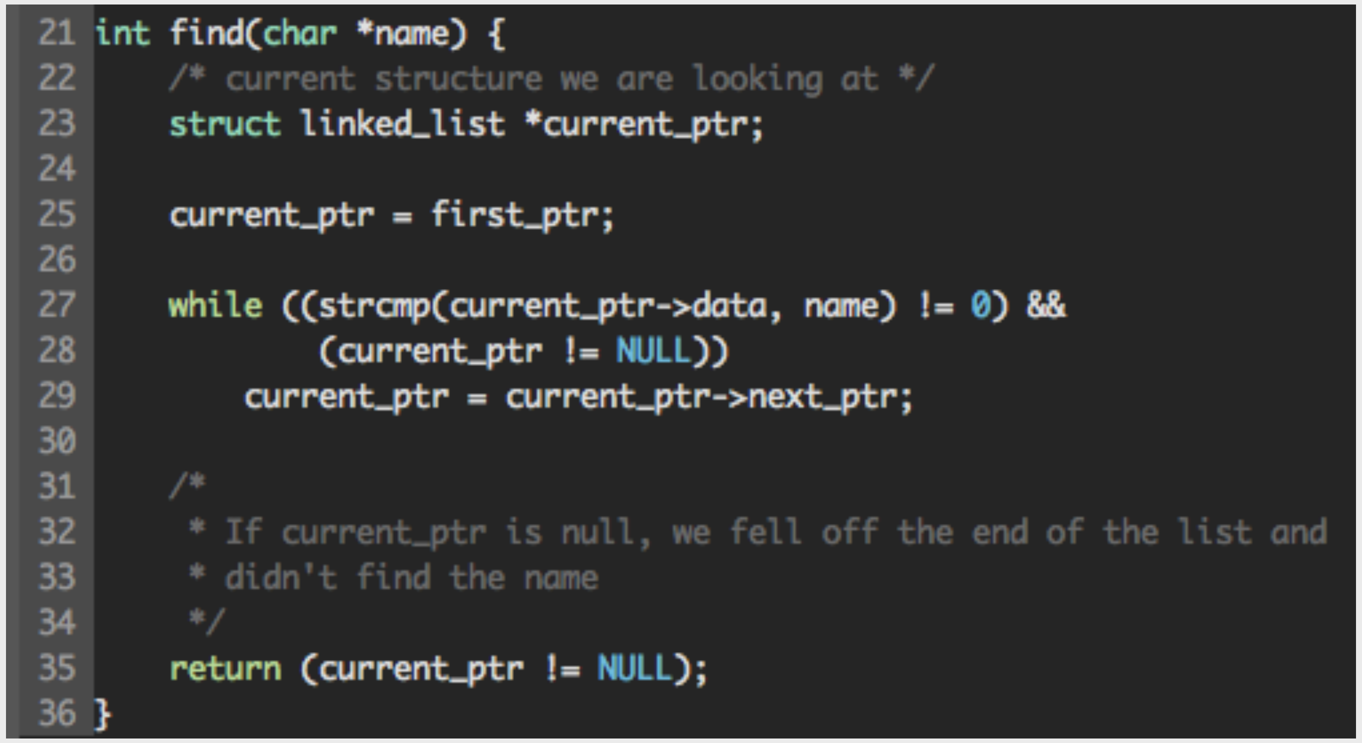
```c
void add_list(char *item)
{
    /* pointer to the next item in the list */
    struct linked_list *new_item_ptr;

    new_item_ptr = malloc(sizeof(struct linked_list));
    strcpy((*new_item_ptr).data, item);
    (*new_item_ptr).next_ptr = first_ptr;
    first_ptr = new_item_ptr;
}
```

Linked List

- To find if an element is in a list
    - search each element of the list until we either find the data or run out of the list

Linked List

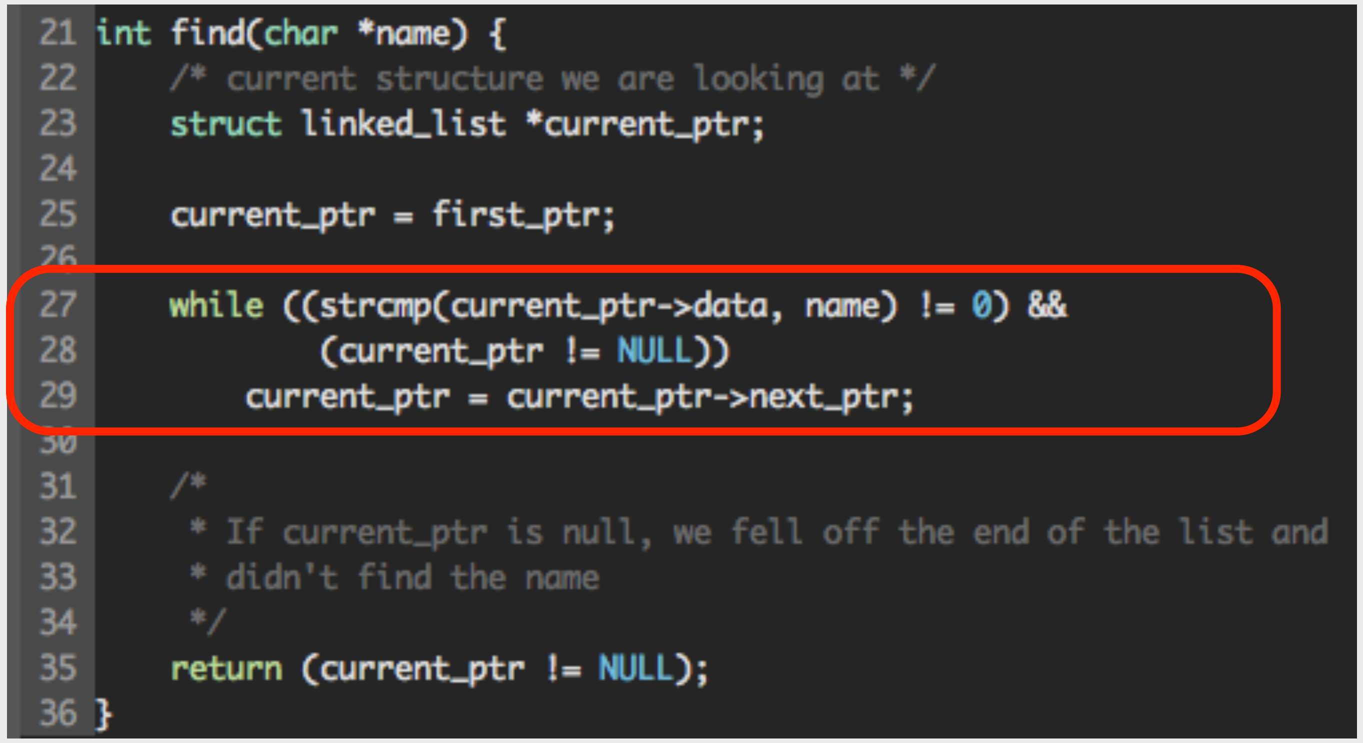
- To find if an element is in a list
    - search each element of the list until we either find the data or run out of the list



```
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

Linked List

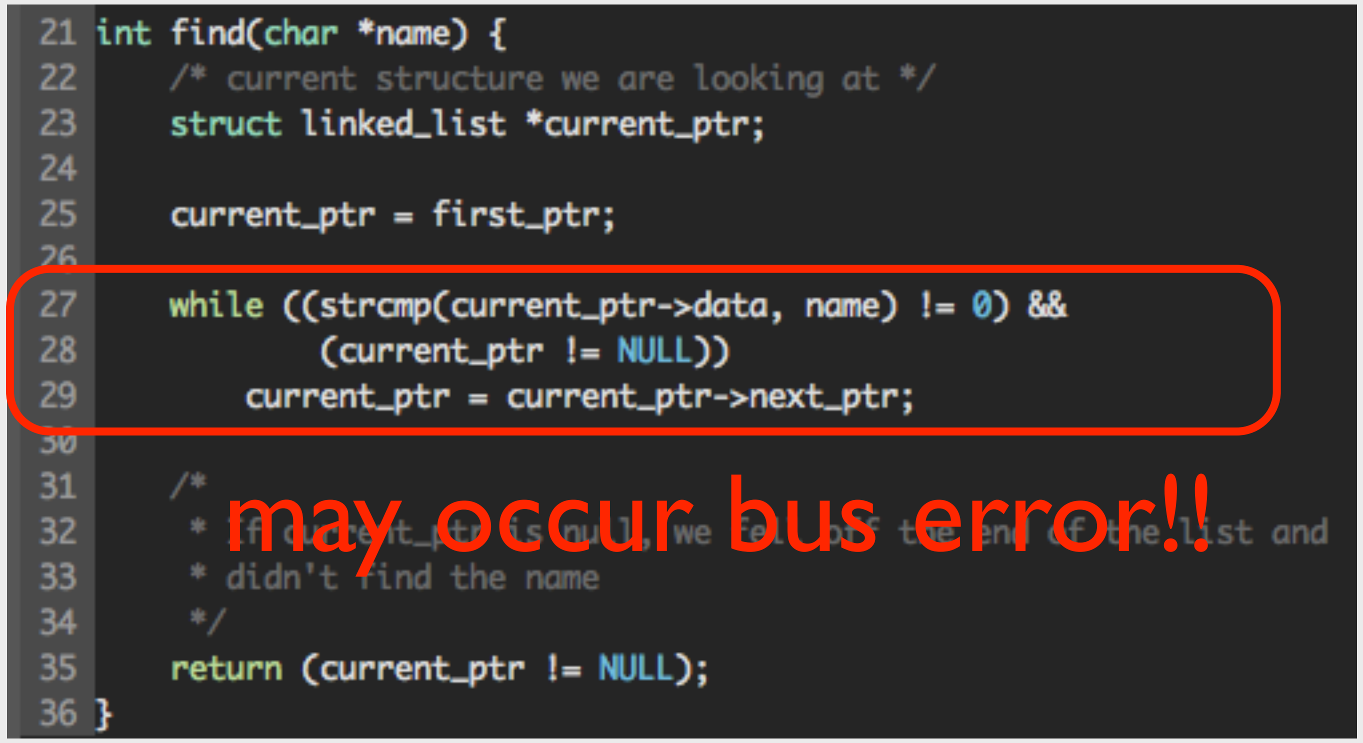
- To find if an element is in a list
    - search each element of the list until we either find the data or run out of the list



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

Linked List

- To find if an element is in a list
  - search each element of the list until we either find the data or run out of the list

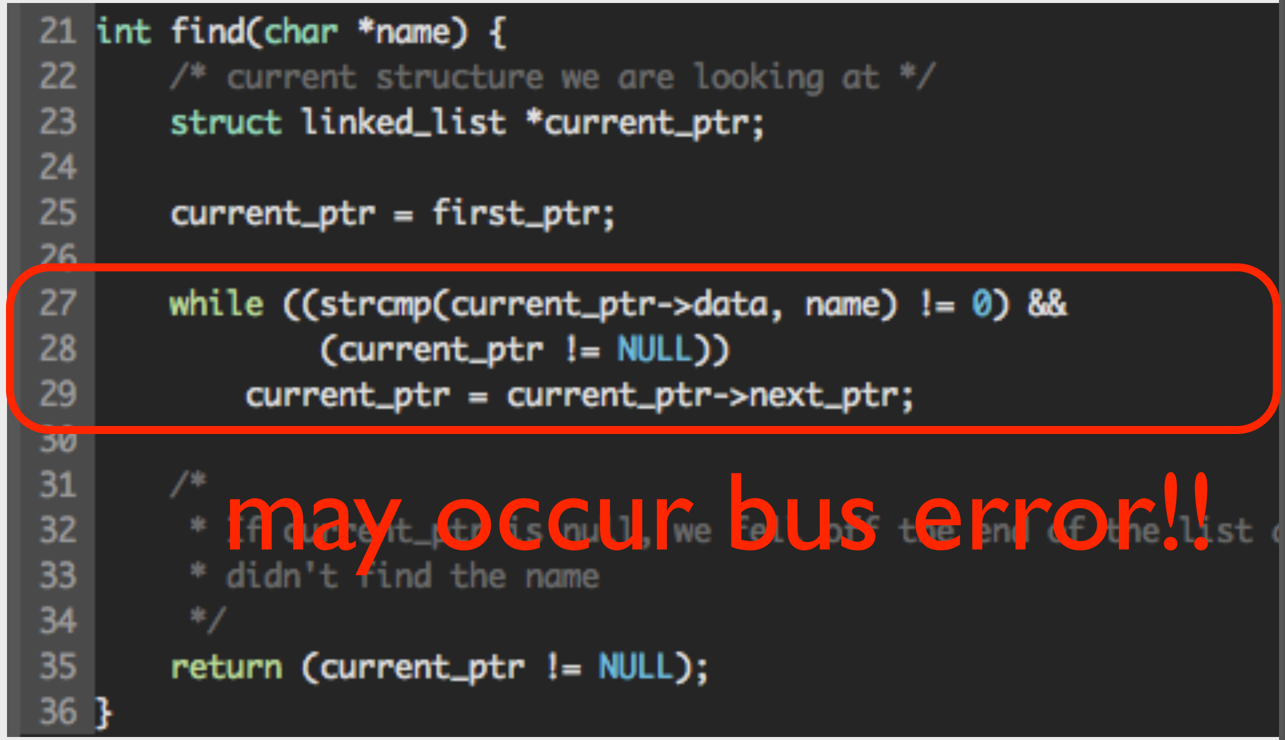


```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

may occur bus error!!

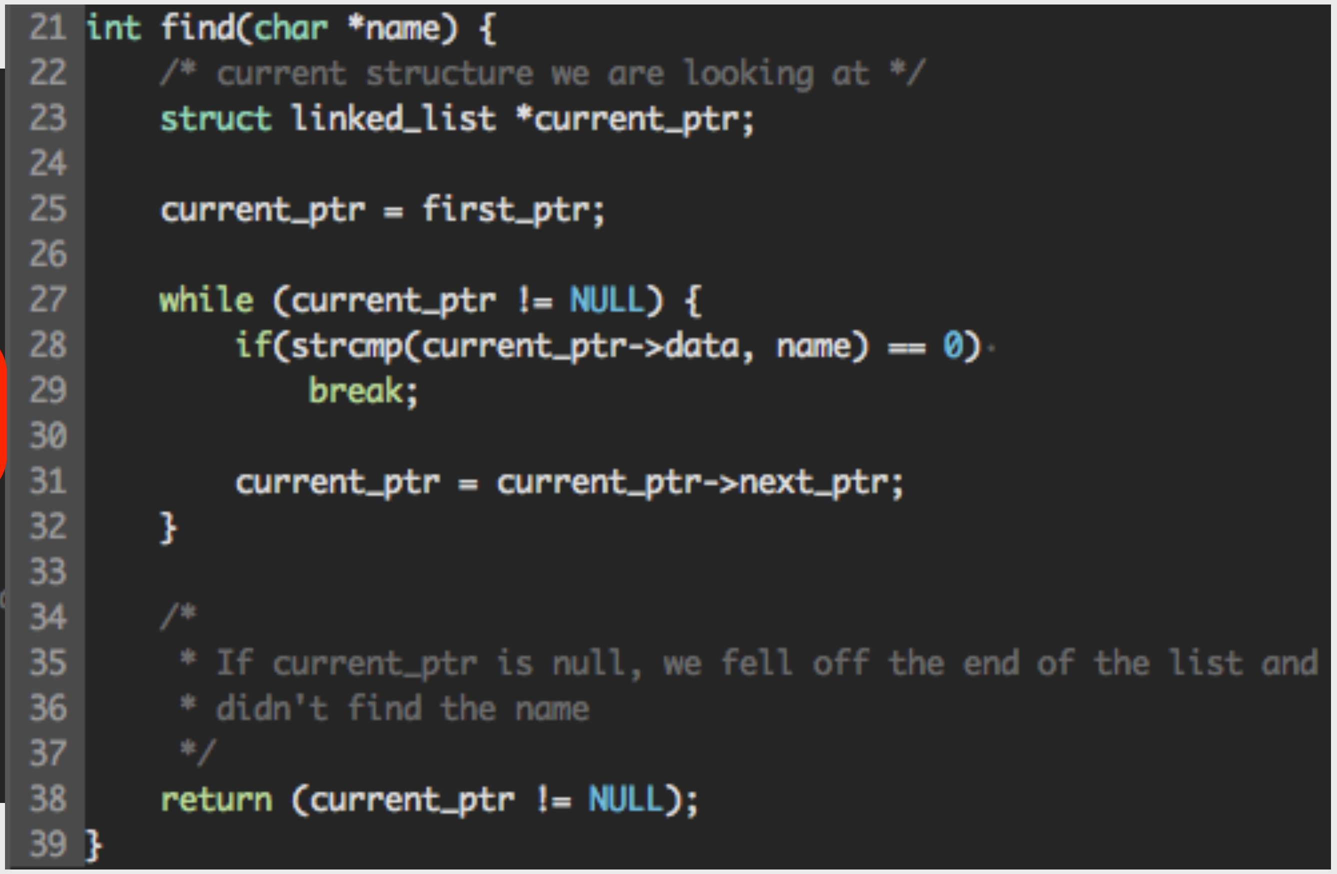
Linked List

- To find if an element is in a list
    - search each element of the list until we either find the data or run out of the list



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list a
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

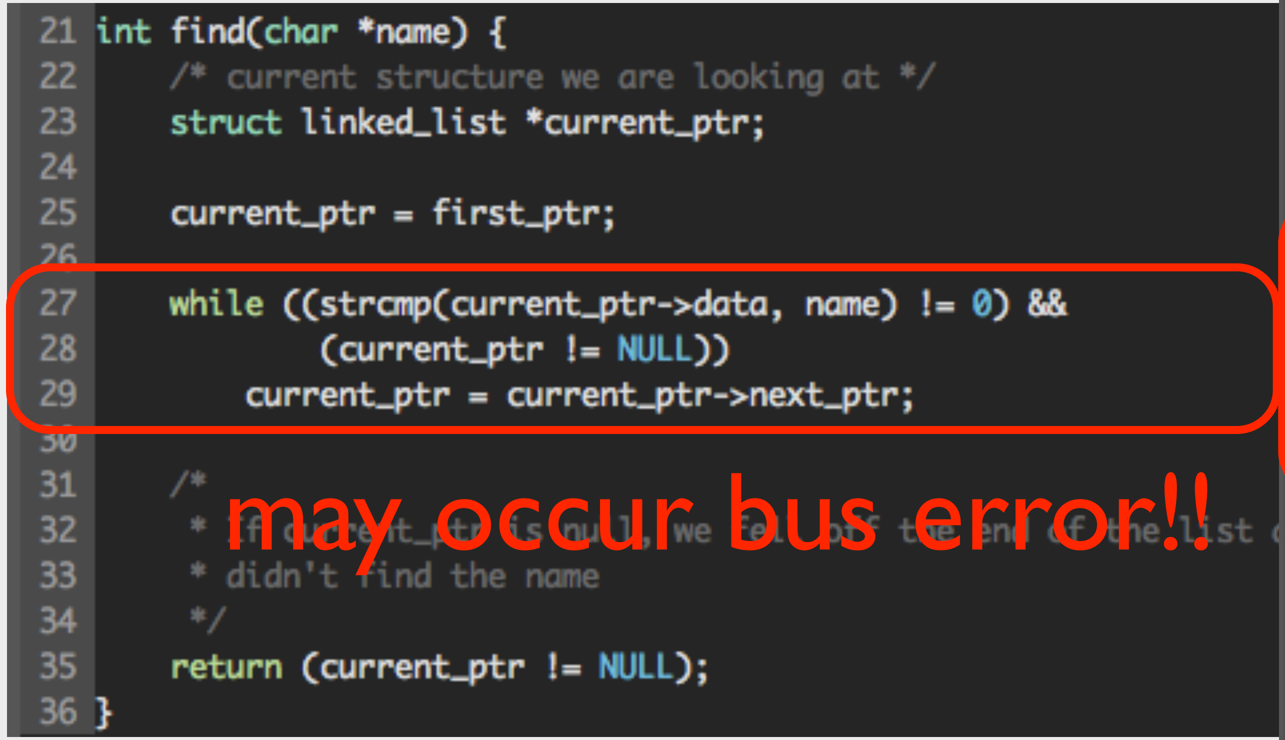
may occur bus error!!



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while (current_ptr != NULL) {
        if(strcmp(current_ptr->data, name) == 0)
            break;

        current_ptr = current_ptr->next_ptr;
    }

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

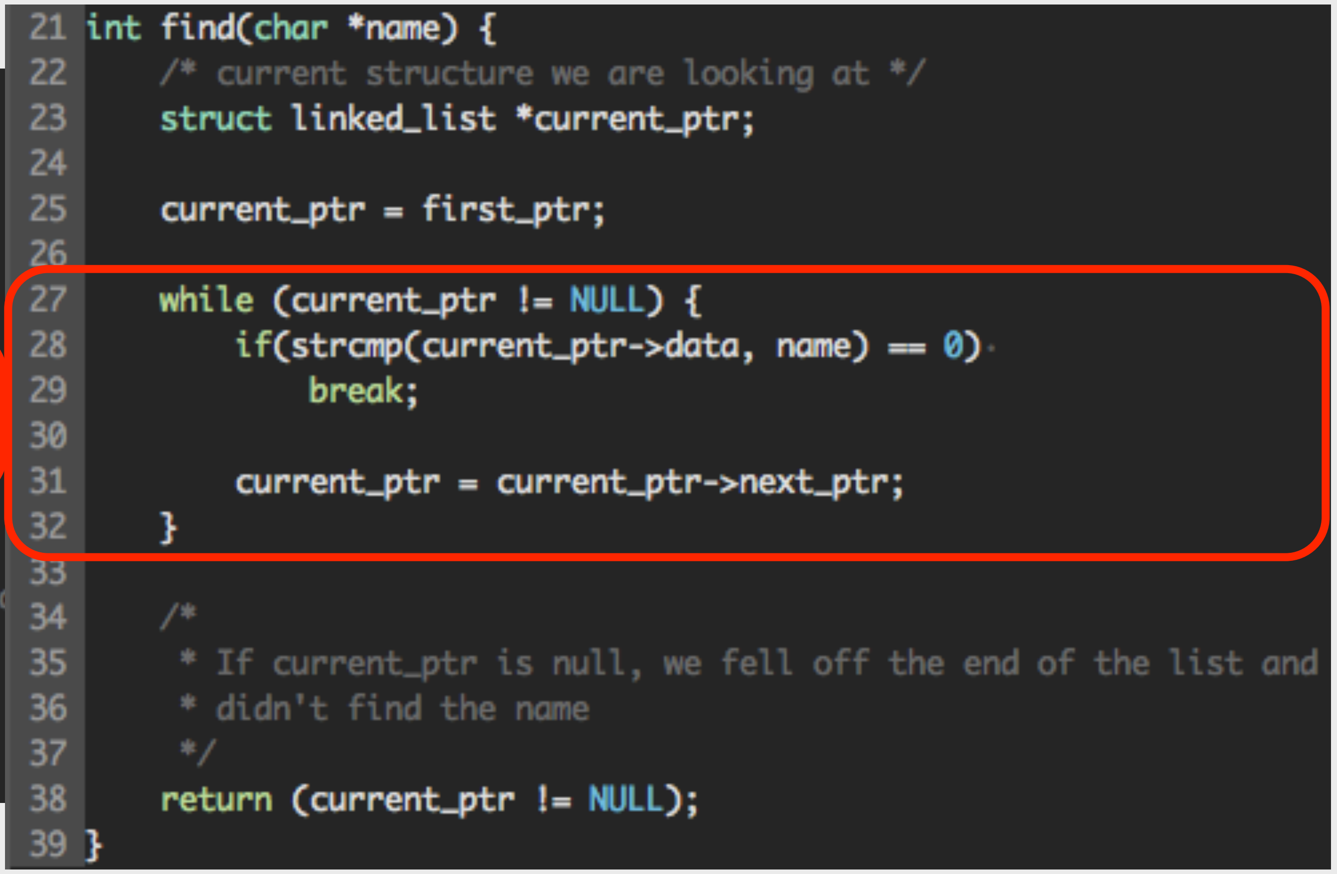
Linked List

- To find if an element is in a list
  - search each element of the list until we either find the data or run out of the list



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

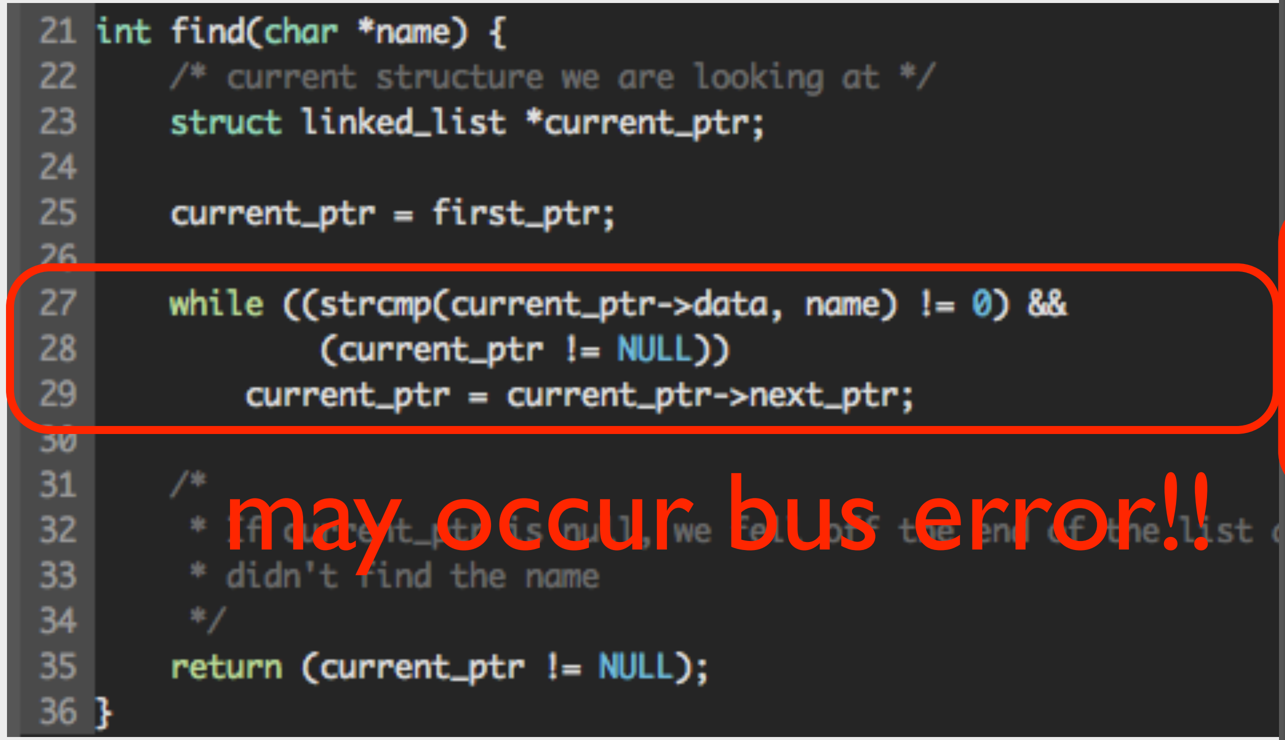
may occur bus error!!



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while (current_ptr != NULL) {
        if(strcmp(current_ptr->data, name) == 0)
            break;

        current_ptr = current_ptr->next_ptr;
    }

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

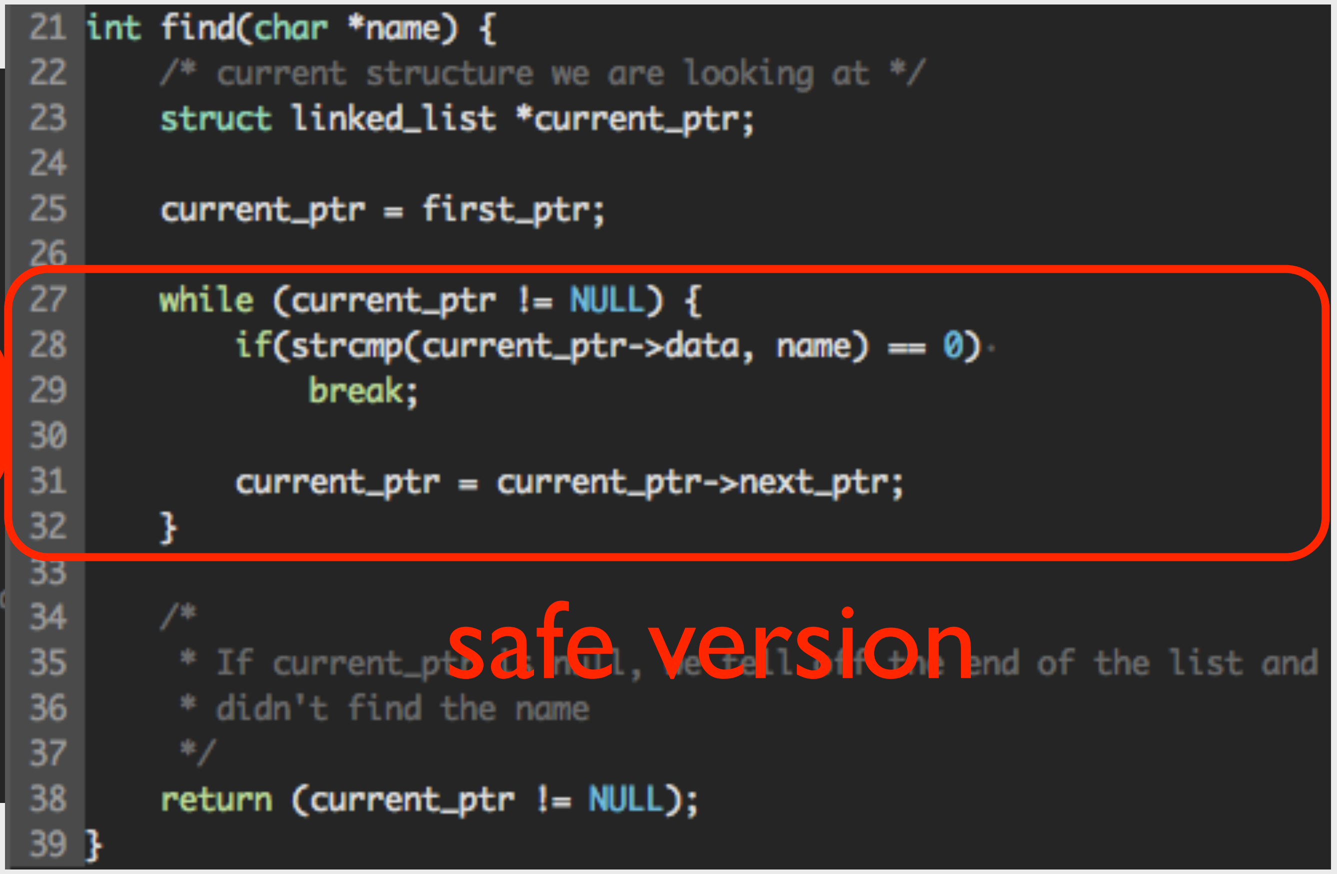
Linked List

- To find if an element is in a list
  - search each element of the list until we either find the data or run out of the list



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
            (current_ptr != NULL))
        current_ptr = current_ptr->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list c
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

may occur bus error!!



```c
int find(char *name) {
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while (current_ptr != NULL) {
        if(strcmp(current_ptr->data, name) == 0)
            break;

        current_ptr = current_ptr->next_ptr;
    }

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

safe version

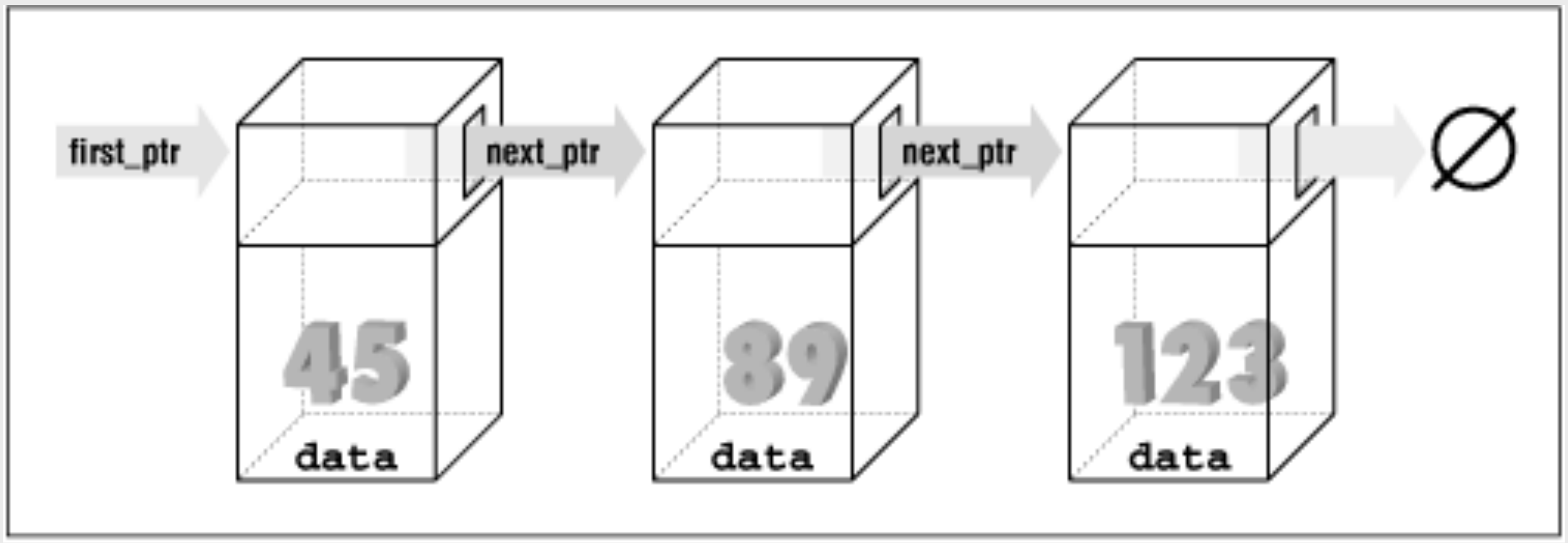
Structure Pointer Operator

- use `(*current_ptr).data` to access the data field of the structure
- C provides a shorthand (`->`)
- The following two expressions are equivalent

```
(*current_ptr).data = value;
current_ptr->data = value;
```

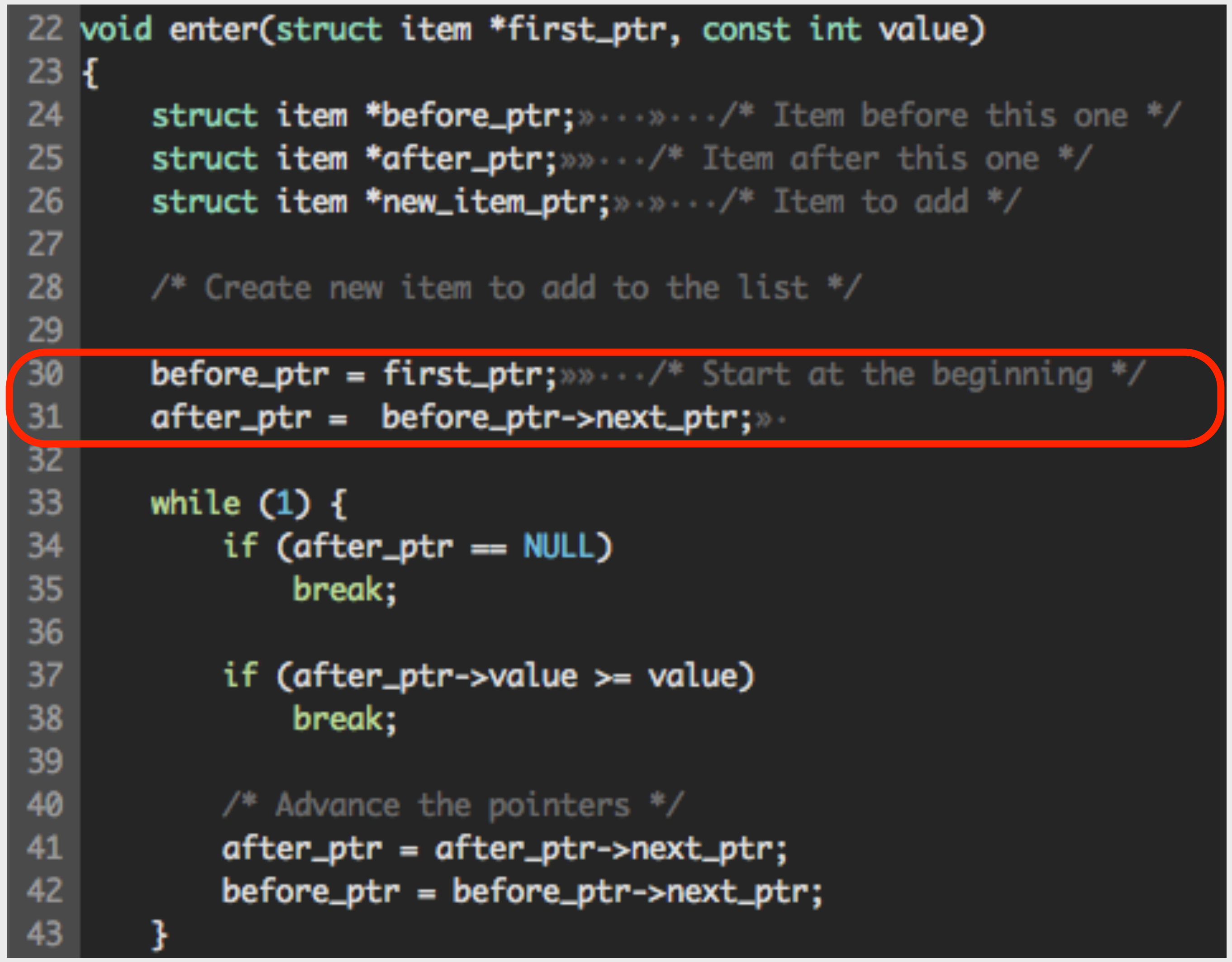
Ordered Linked List

- Suppose we want to add elements in order

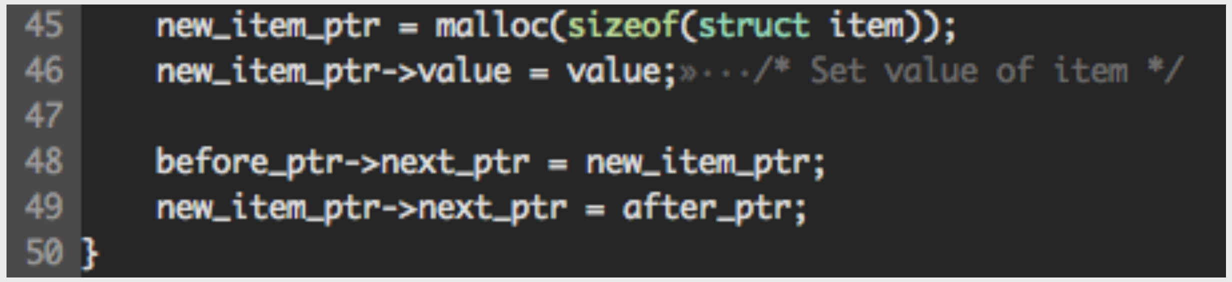


Ordered Linked List

- Example: ordered_list.c



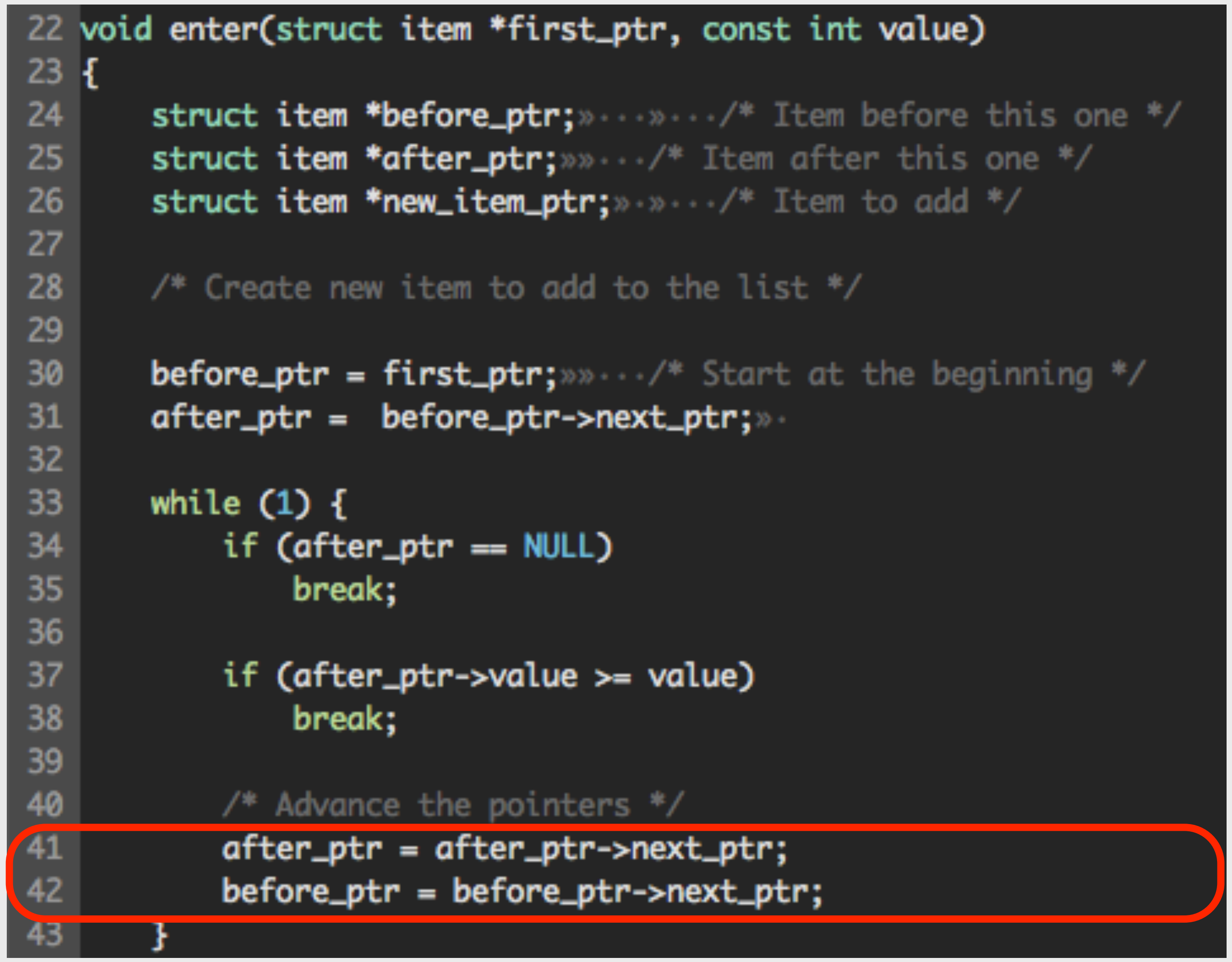
```c
void enter(struct item *first_ptr, const int value)
{
    struct item *before_ptr;        /* Item before this one */
    struct item *after_ptr;     /* Item after this one */
    struct item *new_item_ptr;      /* Item to add */

    /* Create new item to add to the list */

    before_ptr = first_ptr;     /* Start at the beginning */
    after_ptr =  before_ptr->next_ptr;

    while (1) {
        if (after_ptr == NULL)
            break;

        if (after_ptr->value >= value)
            break;

        /* Advance the pointers */
        after_ptr = after_ptr->next_ptr;
        before_ptr = before_ptr->next_ptr;
    }
```



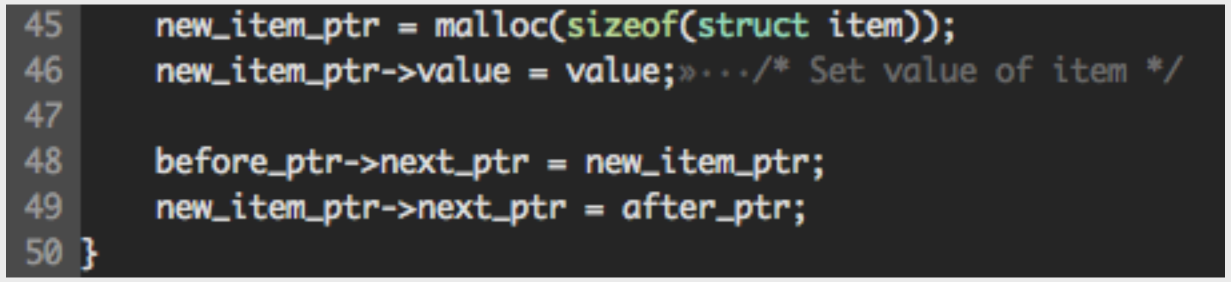
```c
    new_item_ptr = malloc(sizeof(struct item));
    new_item_ptr->value = value;    /* Set value of item */

    before_ptr->next_ptr = new_item_ptr;
    new_item_ptr->next_ptr = after_ptr;
}
```

Ordered Linked List

- Example: ordered_list.c



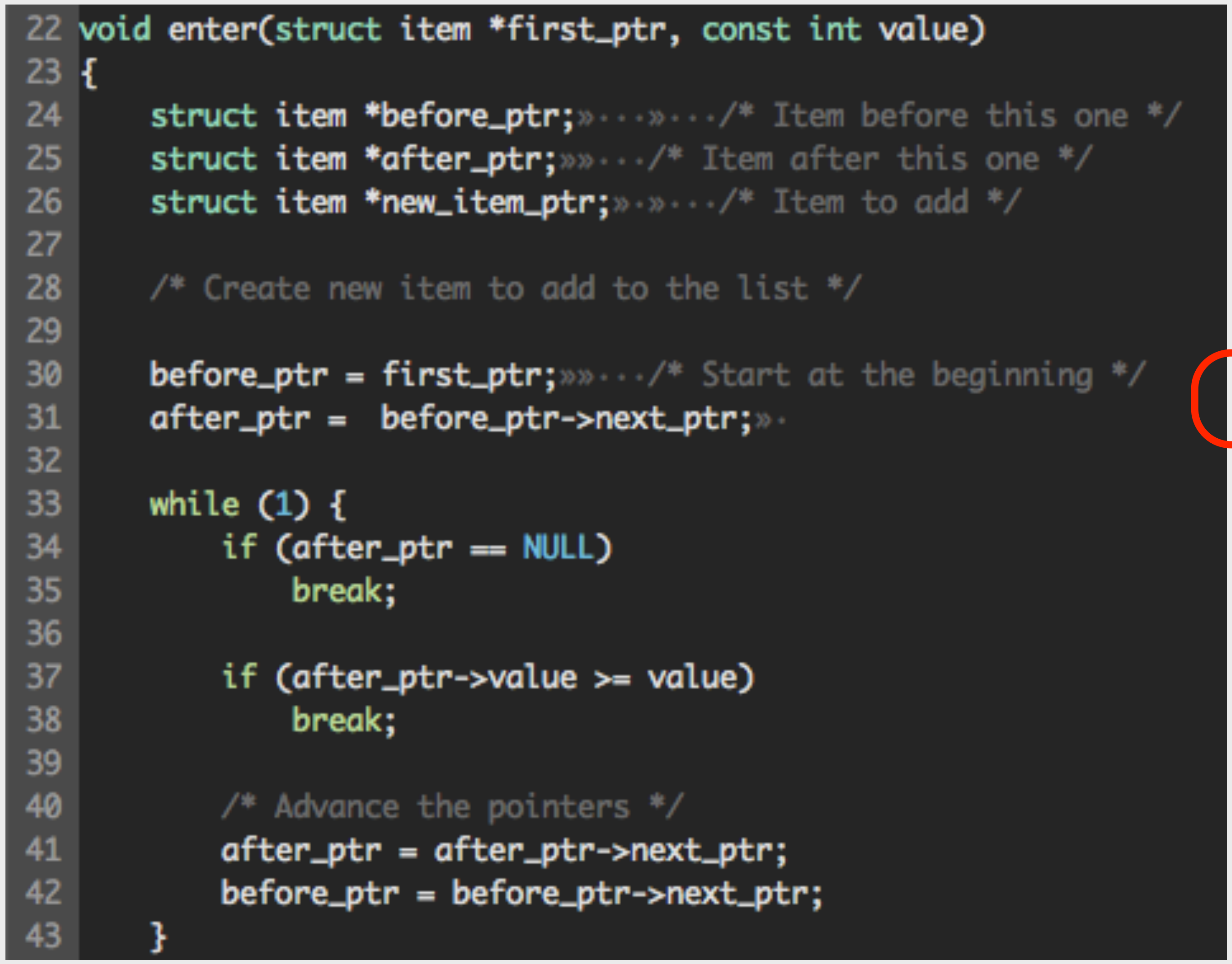
```c
void enter(struct item *first_ptr, const int value)
{
    struct item *before_ptr;        /* Item before this one */
    struct item *after_ptr;         /* Item after this one */
    struct item *new_item_ptr;      /* Item to add */

    /* Create new item to add to the list */

    before_ptr = first_ptr;         /* Start at the beginning */
    after_ptr =  before_ptr->next_ptr;

    while (1) {
        if (after_ptr == NULL)
            break;

        if (after_ptr->value >= value)
            break;

        /* Advance the pointers */
        after_ptr = after_ptr->next_ptr;
        before_ptr = before_ptr->next_ptr;
    }
```



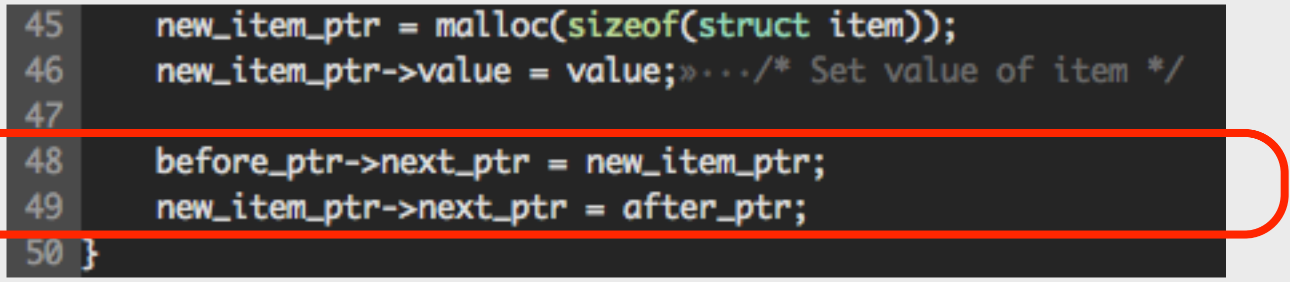
```c
    new_item_ptr = malloc(sizeof(struct item));
    new_item_ptr->value = value;    /* Set value of item */

    before_ptr->next_ptr = new_item_ptr;
    new_item_ptr->next_ptr = after_ptr;
}
```

Ordered Linked List

- Example: ordered_list.c



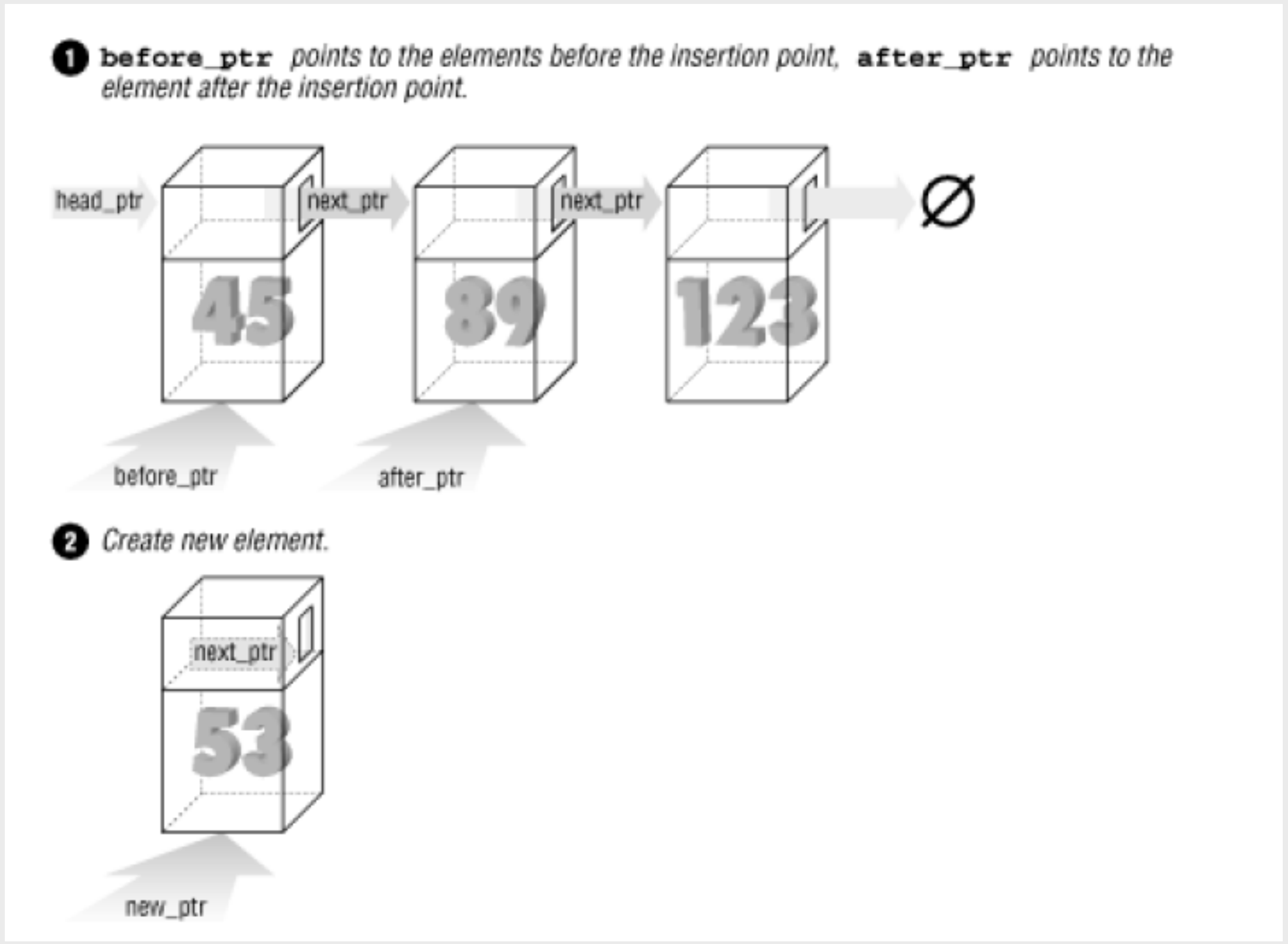
```c
void enter(struct item *first_ptr, const int value)
{
    struct item *before_ptr;        /* Item before this one */
    struct item *after_ptr;         /* Item after this one */
    struct item *new_item_ptr;      /* Item to add */

    /* Create new item to add to the list */

    before_ptr = first_ptr;         /* Start at the beginning */
    after_ptr =  before_ptr->next_ptr;

    while (1) {
        if (after_ptr == NULL)
            break;

        if (after_ptr->value >= value)
            break;

        /* Advance the pointers */
        after_ptr = after_ptr->next_ptr;
        before_ptr = before_ptr->next_ptr;
    }
```



```c
    new_item_ptr = malloc(sizeof(struct item));
    new_item_ptr->value = value;    /* Set value of item */

    before_ptr->next_ptr = new_item_ptr;
    new_item_ptr->next_ptr = after_ptr;
}
```

Ordered Linked List

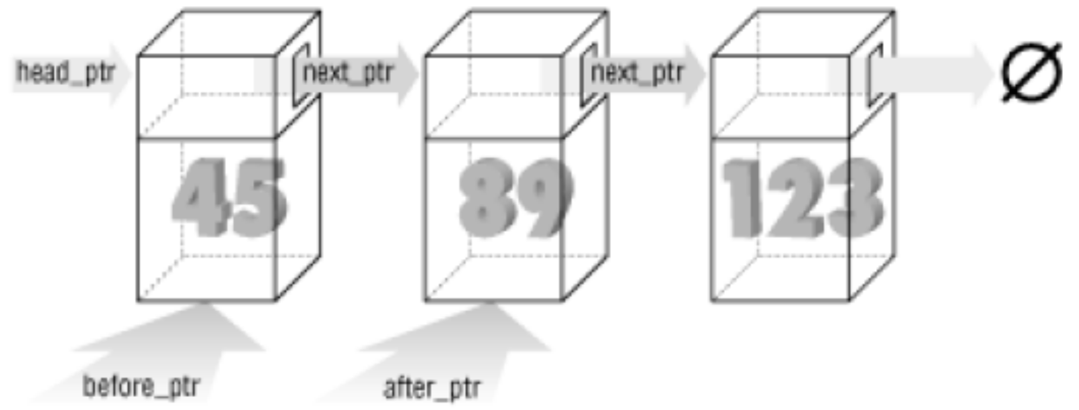
Ordered Linked List



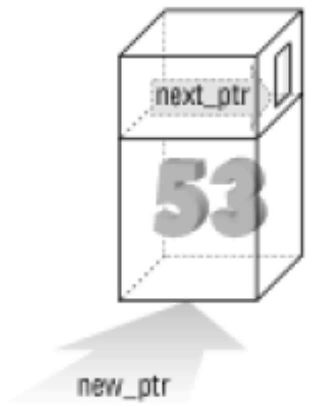
❶ before_ptr points to the elements before the insertion point, after_ptr points to the element after the insertion point.
head_ptr
next_ptr
next_ptr
45
89
123
∅
before_ptr
after_ptr
❷ Create new element.
next_ptr
53
new_ptr

Ordered Linked List

❶ **before_ptr** *points to the elements before the insertion point,* **after_ptr** *points to the element after the insertion point.*

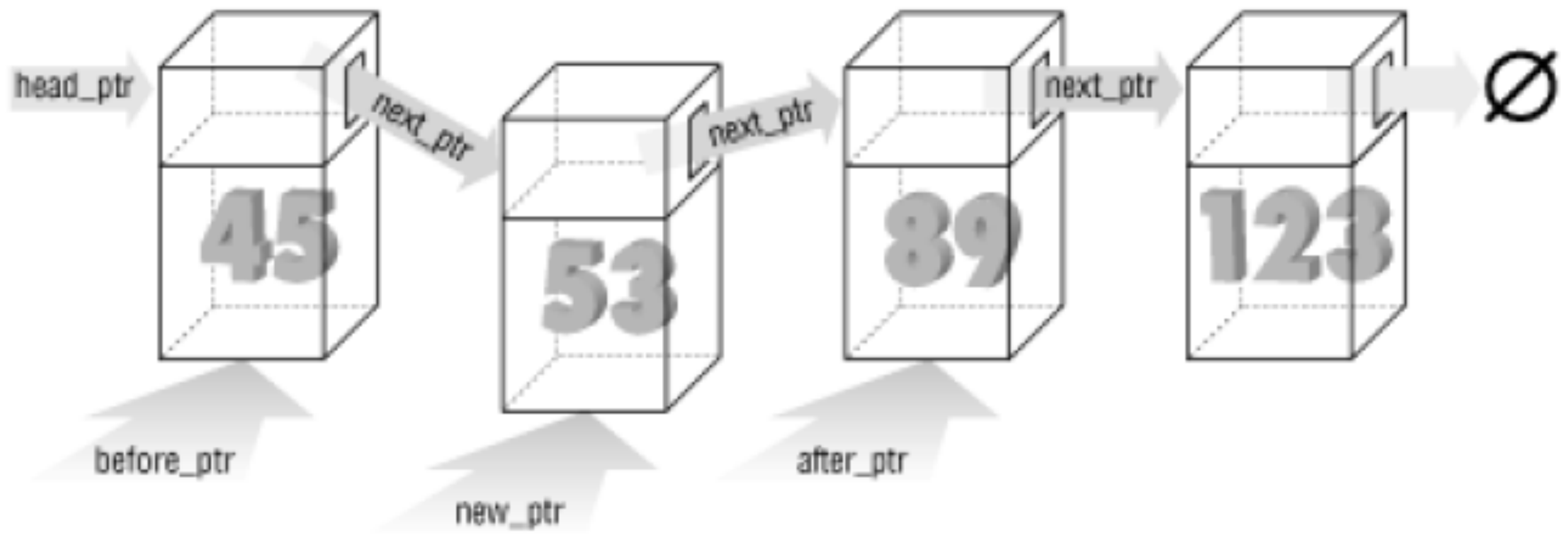


❷ *Create new element.*

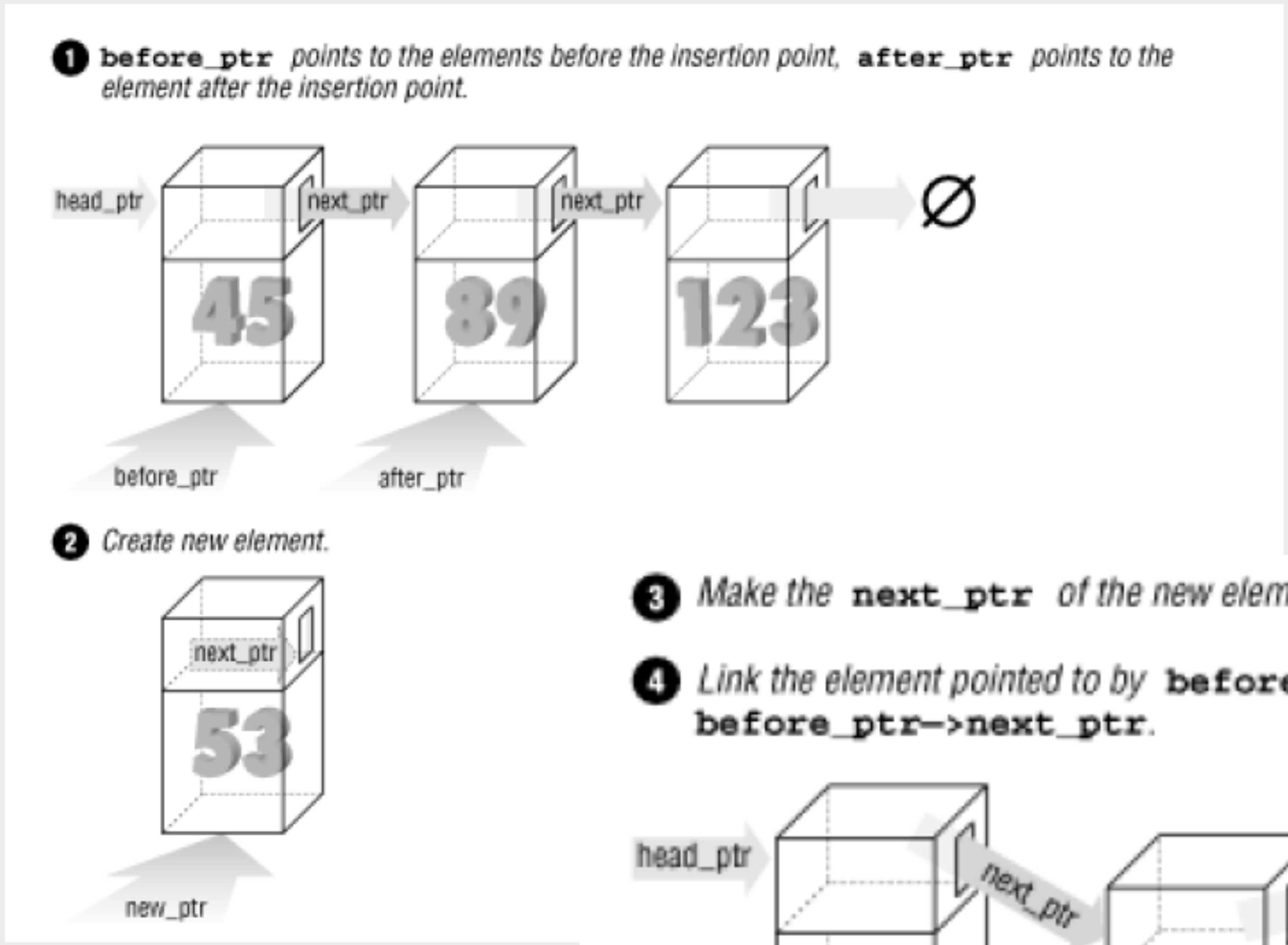


❸ *Make the* **next_ptr** *of the new element point to the same element as* **after_ptr**.

❹ *Link the element pointed to by* **before_ptr** *to our new element by changing* **before_ptr->next_ptr**.

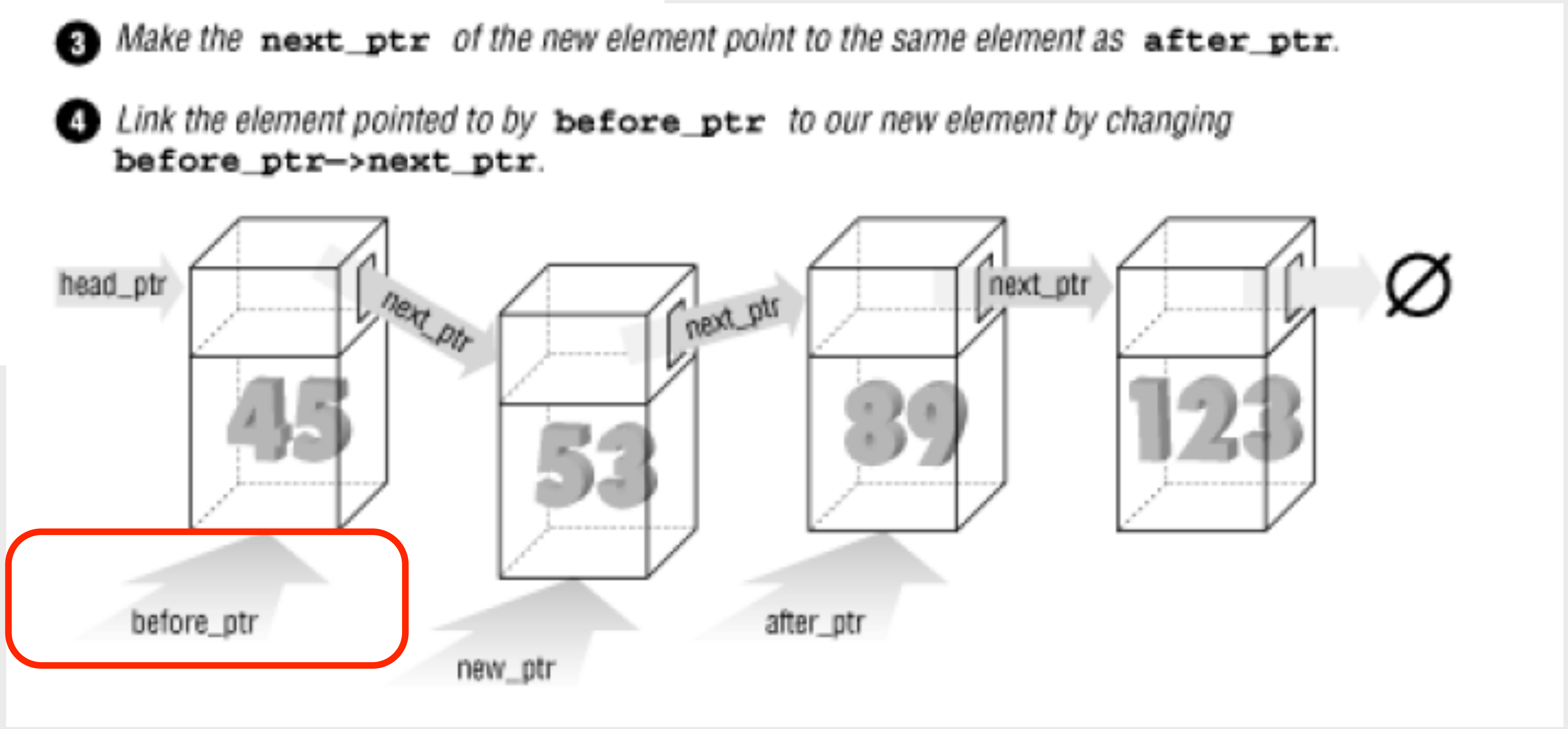


Ordered Linked List



❶ **before_ptr** *points to the elements before the insertion point,* **after_ptr** *points to the element after the insertion point.*

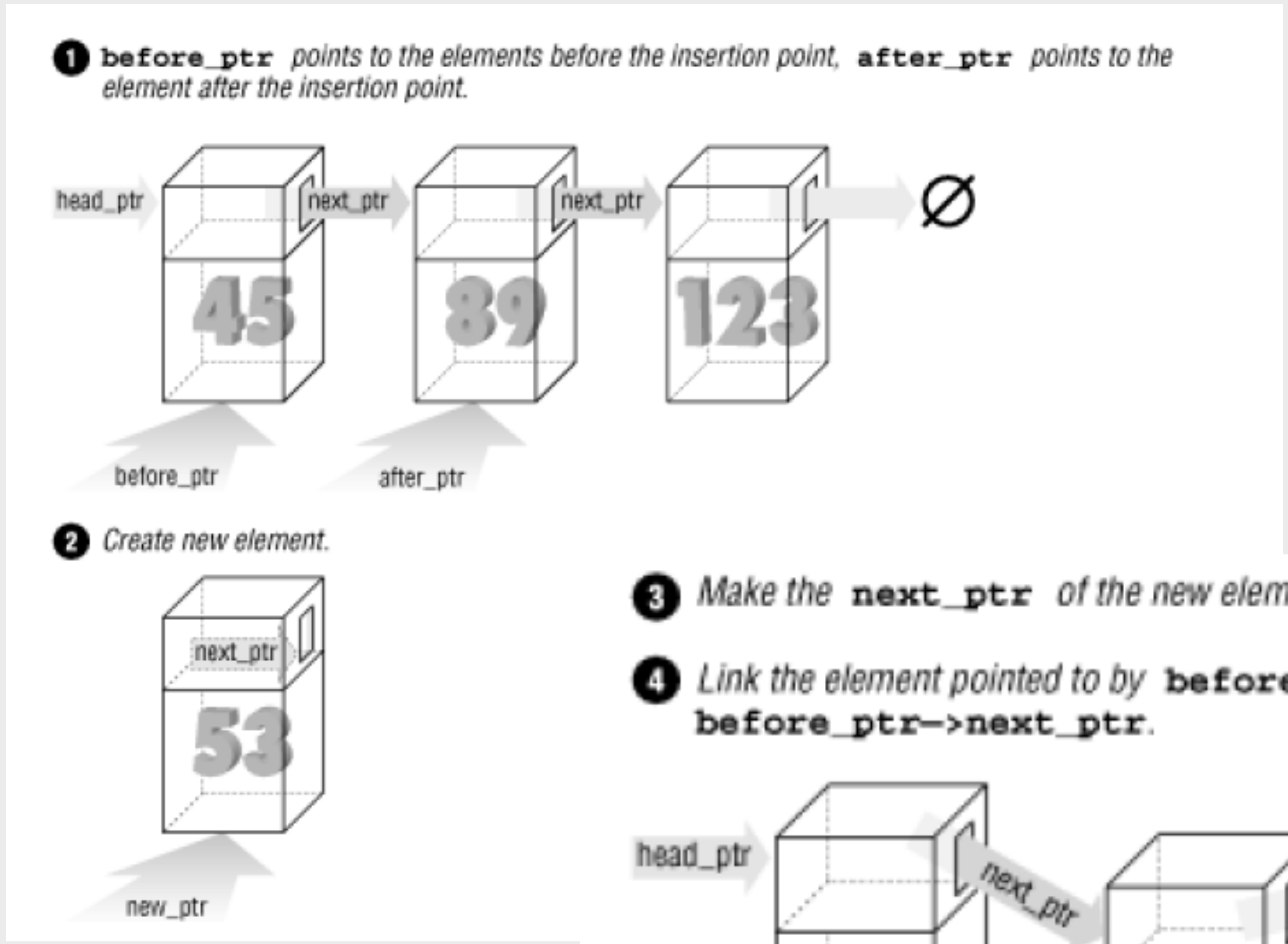
❷ *Create new element.*



❸ *Make the* **next_ptr** *of the new element point to the same element as* **after_ptr***.*

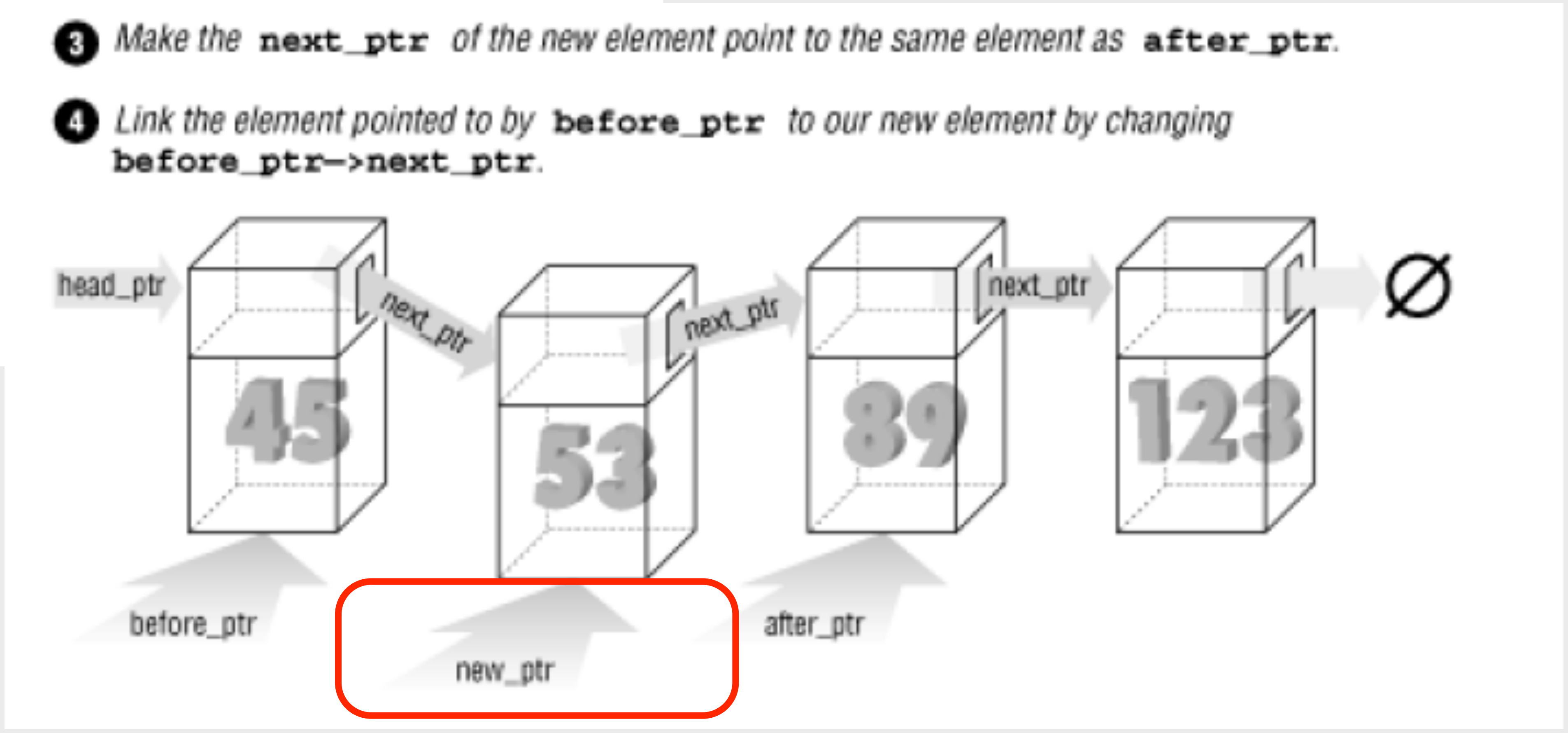
❹ *Link the element pointed to by* **before_ptr** *to our new element by changing* **before_ptr->next_ptr***.*

Ordered Linked List



❶ **before_ptr** *points to the elements before the insertion point,* **after_ptr** *points to the element after the insertion point.*

❷ *Create new element.*

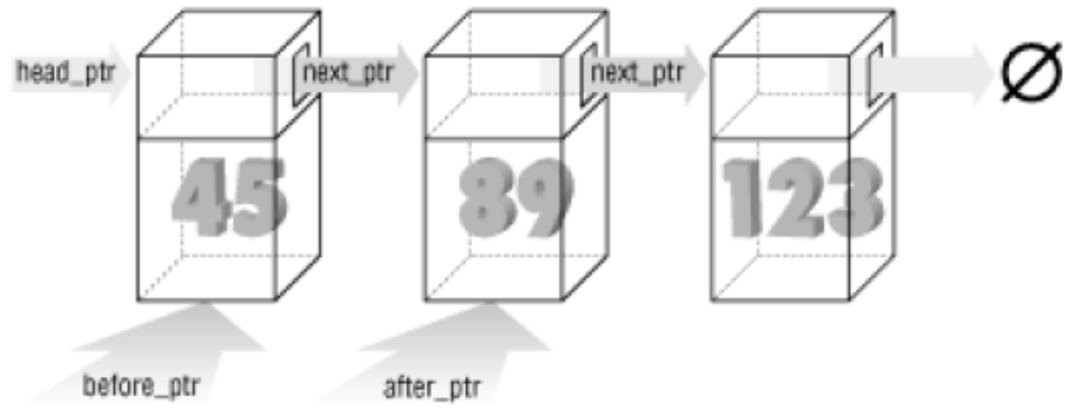


❸ *Make the* **next_ptr** *of the new element point to the same element as* **after_ptr**.

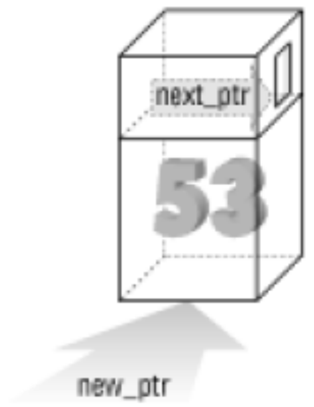
❹ *Link the element pointed to by* **before_ptr** *to our new element by changing* **before_ptr->next_ptr**.

Ordered Linked List

❶ **`before_ptr`** *points to the elements before the insertion point,* **`after_ptr`** *points to the element after the insertion point.*

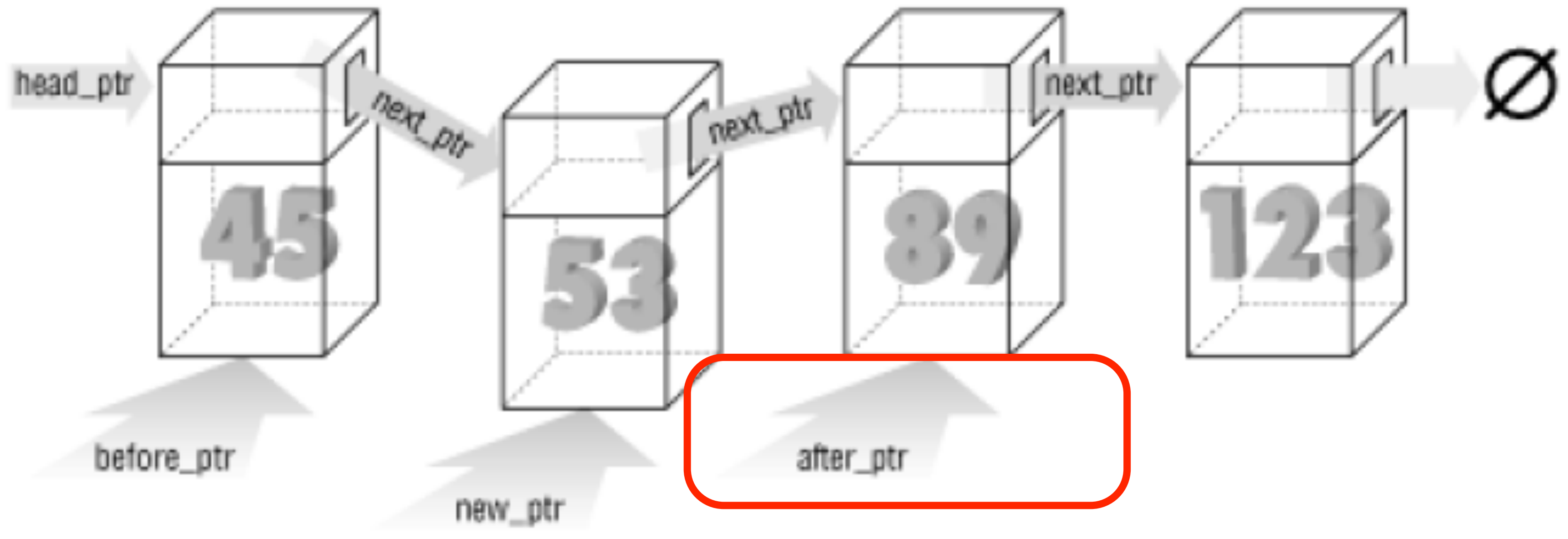


❷ *Create new element.*



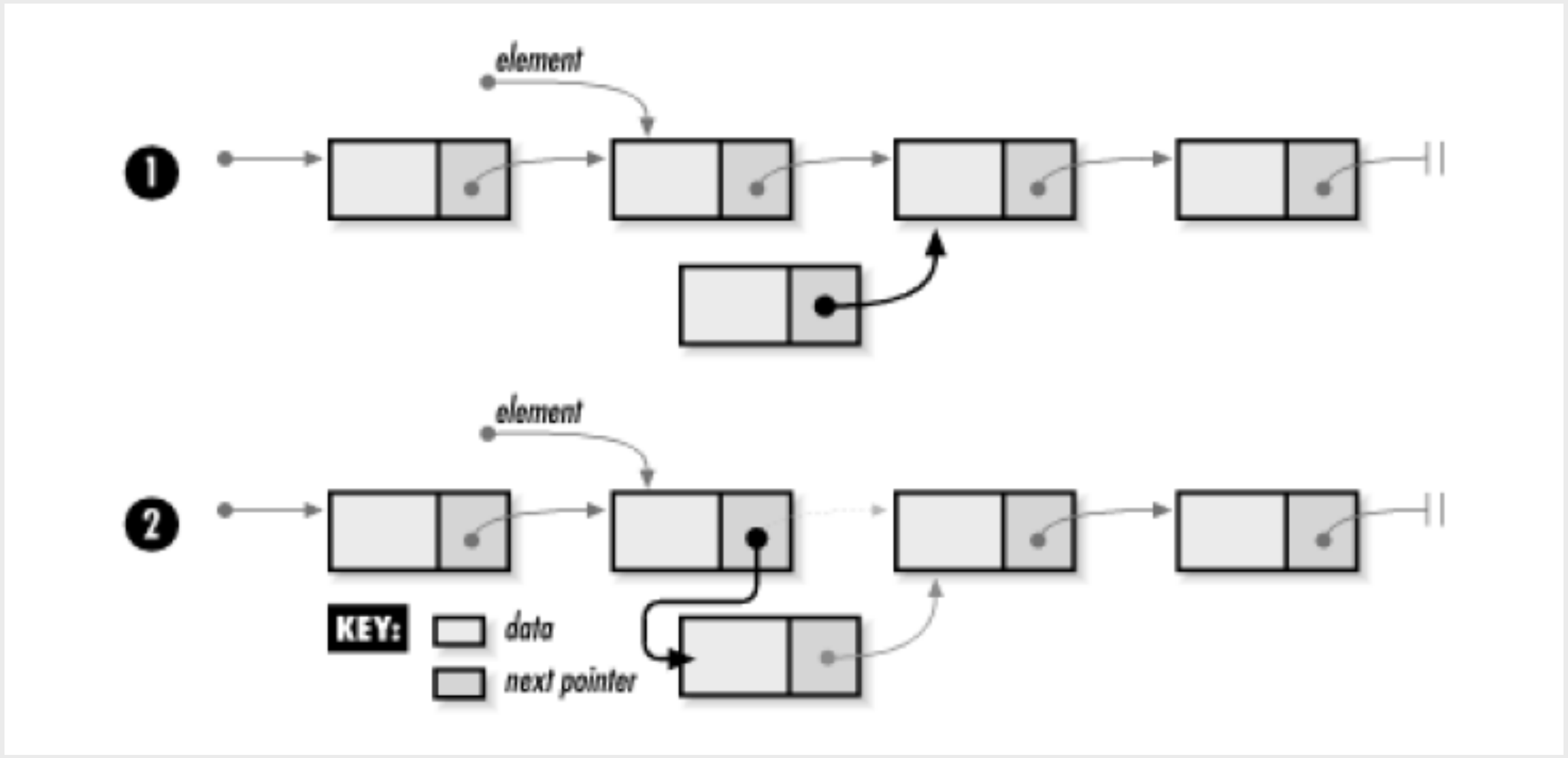
❸ *Make the* **`next_ptr`** *of the new element point to the same element as* **`after_ptr`**.

❹ *Link the element pointed to by* **`before_ptr`** *to our new element by changing* **`before_ptr->next_ptr`**.



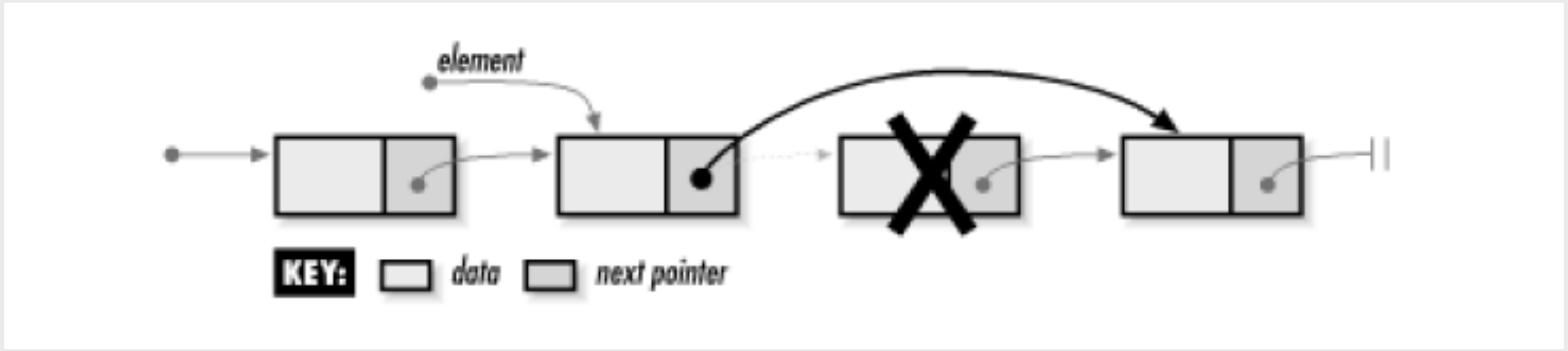
Linked List

- Insert an element from a linked list
  - insertion at the head of the list
  - insertion elsewhere



Linked List

- Remove an element from a linked list
  - remove an element from the head of the list
  - remove one elsewhere



Analysis of Linked List

- Common Interfaces of A More Useful Linked List
  - Initialization
    - Return value: none
    - Complexity: O(1)
  - Destroy
    - Return value: none
    - Complexity: O(n), where n is the number of elements in the linked list

Analysis of Linked List

- Insert next
  - Return value: 0 if inserting the element is successful, or -1 otherwise
  - Complexity: O(1)
- Remove next
  - Return value: 0 if removing the element is successful, or -1 otherwise
  - Complexity: O(1)

Analysis of Linked List

- List size
  - Return value: number of elements in the list
  - Complexity: O(1)
- Head
  - Return value: element at the head of the list
  - Complexity: O(1)
- Tail
  - Return value: element at the tail of the list
  - Complexity: O(1)

Analysis of Linked List

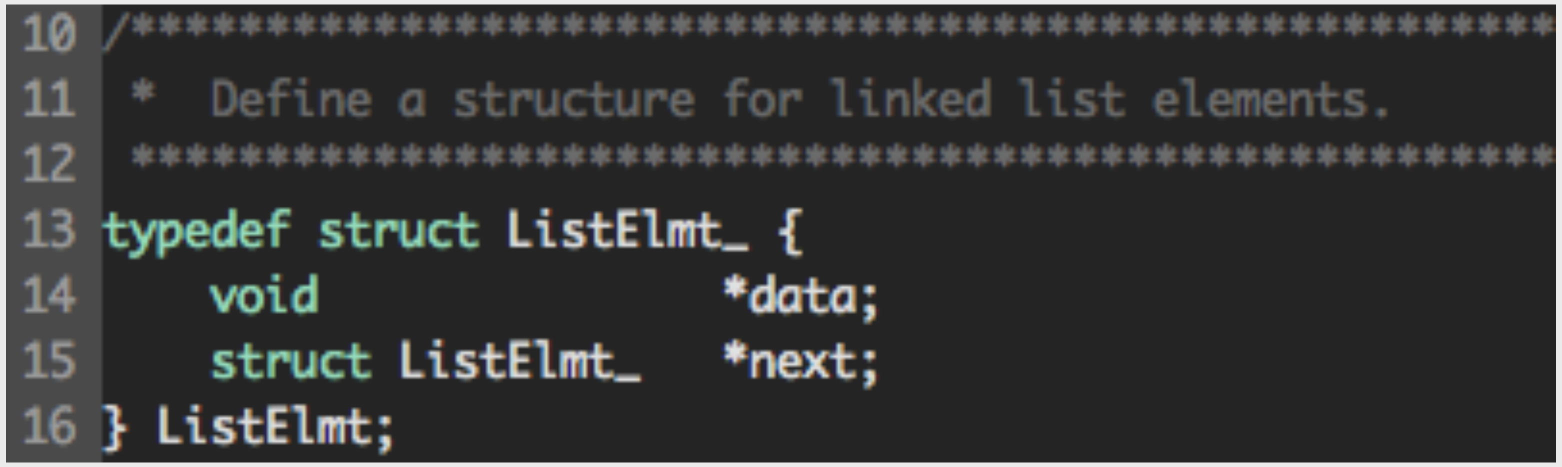
- isHead
  - Return value: 1 if the element is at the head of the list, or otherwise
  - Complexity: O(1)
- isTail
  - Return value: 1 if the element is at the tail of the list, or otherwise
  - Complexity: O(1)

Analysis of Linked List

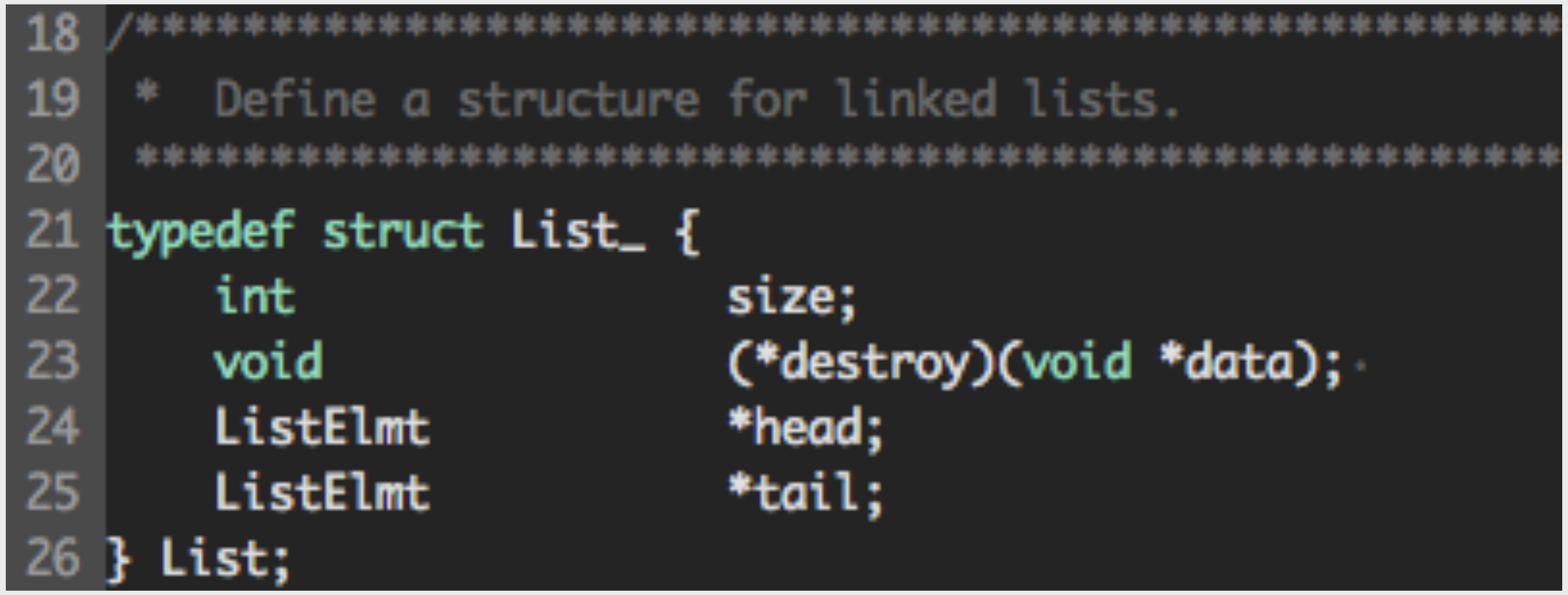
- List data
    - Return value: data stored in the element
    - Complexity: O(1)
- List next
    - Return value: element following the specified element
    - Complexity: O(1)

Advanced Linked List

- Example: advList/advList.h



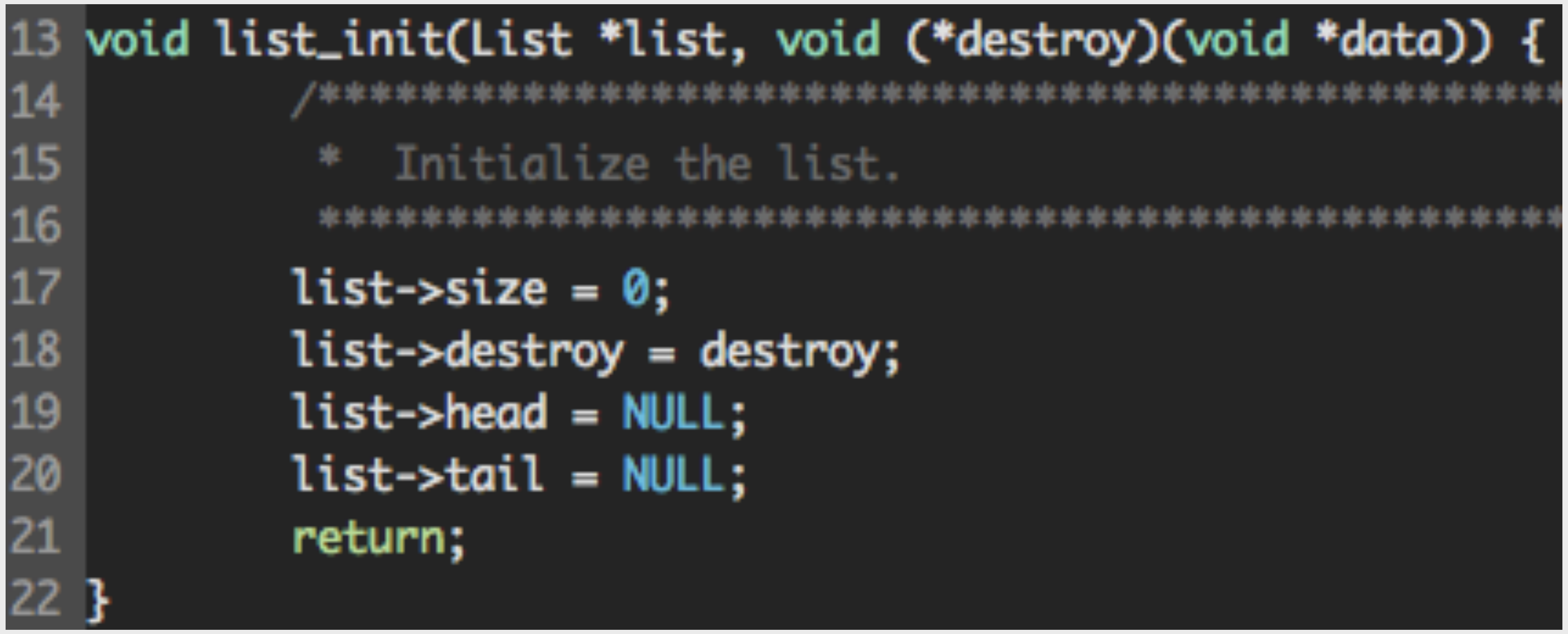
```c
/*****************************************************************
 *  Define a structure for linked list elements.
 *****************************************************************
typedef struct ListElmt_ {
    void                *data;
    struct ListElmt_    *next;
} ListElmt;
```



```c
/*****************************************************************
 *  Define a structure for linked lists.
 *****************************************************************
typedef struct List_ {
    int                 size;
    void                (*destroy)(void *data);
    ListElmt            *head;
    ListElmt            *tail;
} List;
```

Advanced Linked List

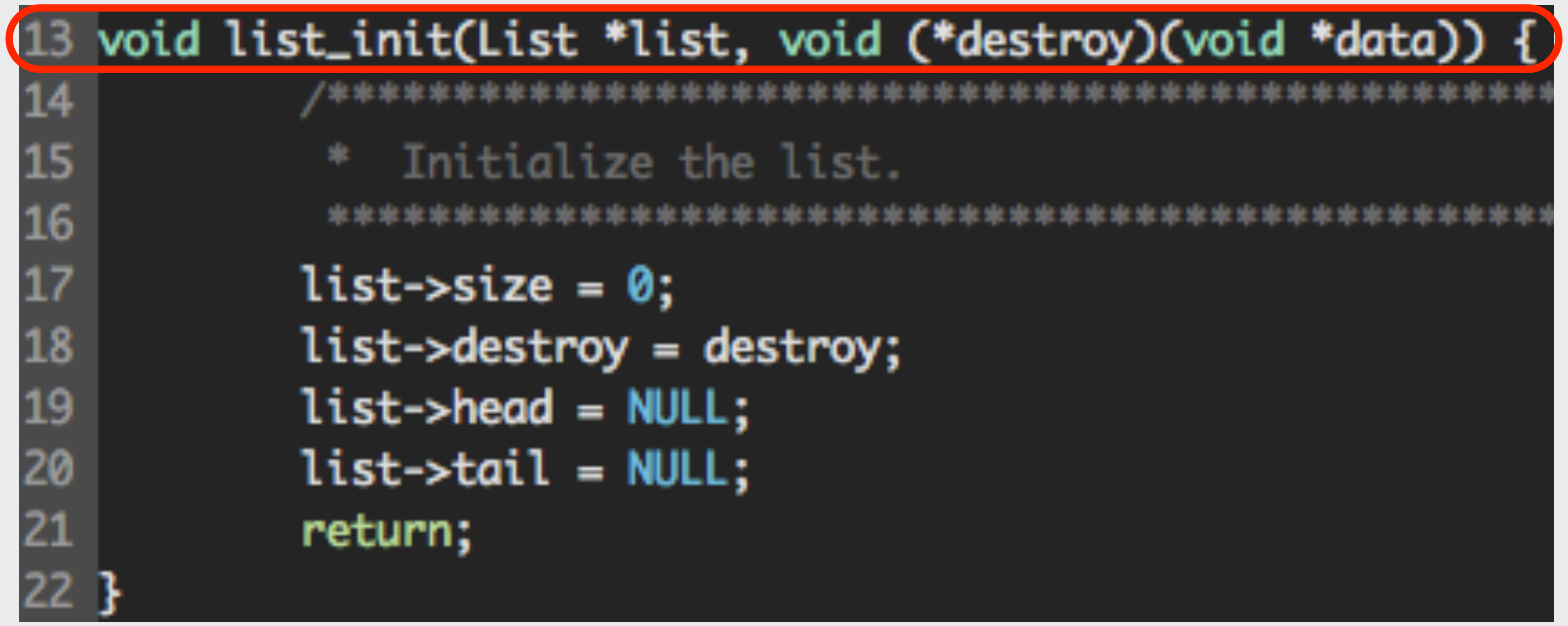
- Example: advList/advList.c



```
void list_init(List *list, void (*destroy)(void *data)) {
        /***********************************************
         *  Initialize the list.
         ***********************************************
        list->size = 0;
        list->destroy = destroy;
        list->head = NULL;
        list->tail = NULL;
        return;
}
```

Advanced Linked List

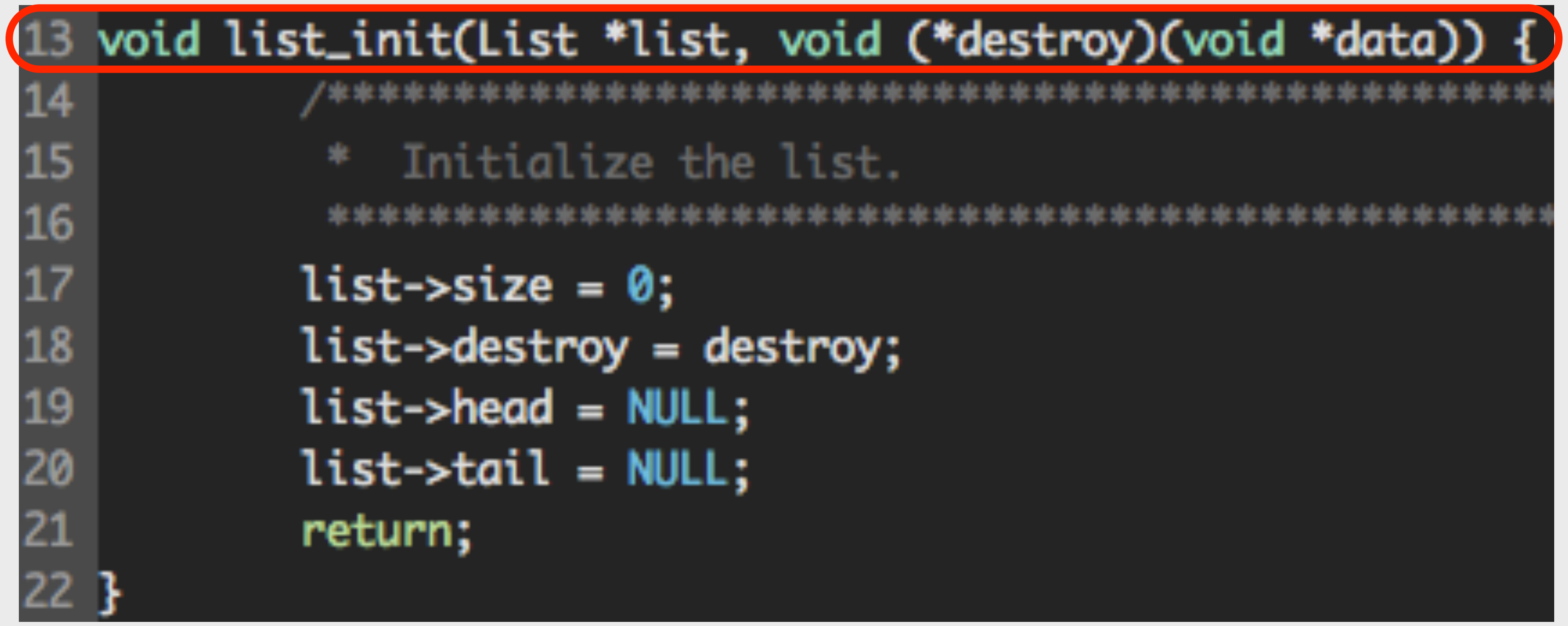
- Example: advList/advList.c



```
void list_init(List *list, void (*destroy)(void *data)) {
        /*********************************************
         *  Initialize the list.
         *********************************************
        list->size = 0;
        list->destroy = destroy;
        list->head = NULL;
        list->tail = NULL;
        return;
}
```

Advanced Linked List

- Example: advList/advList.c

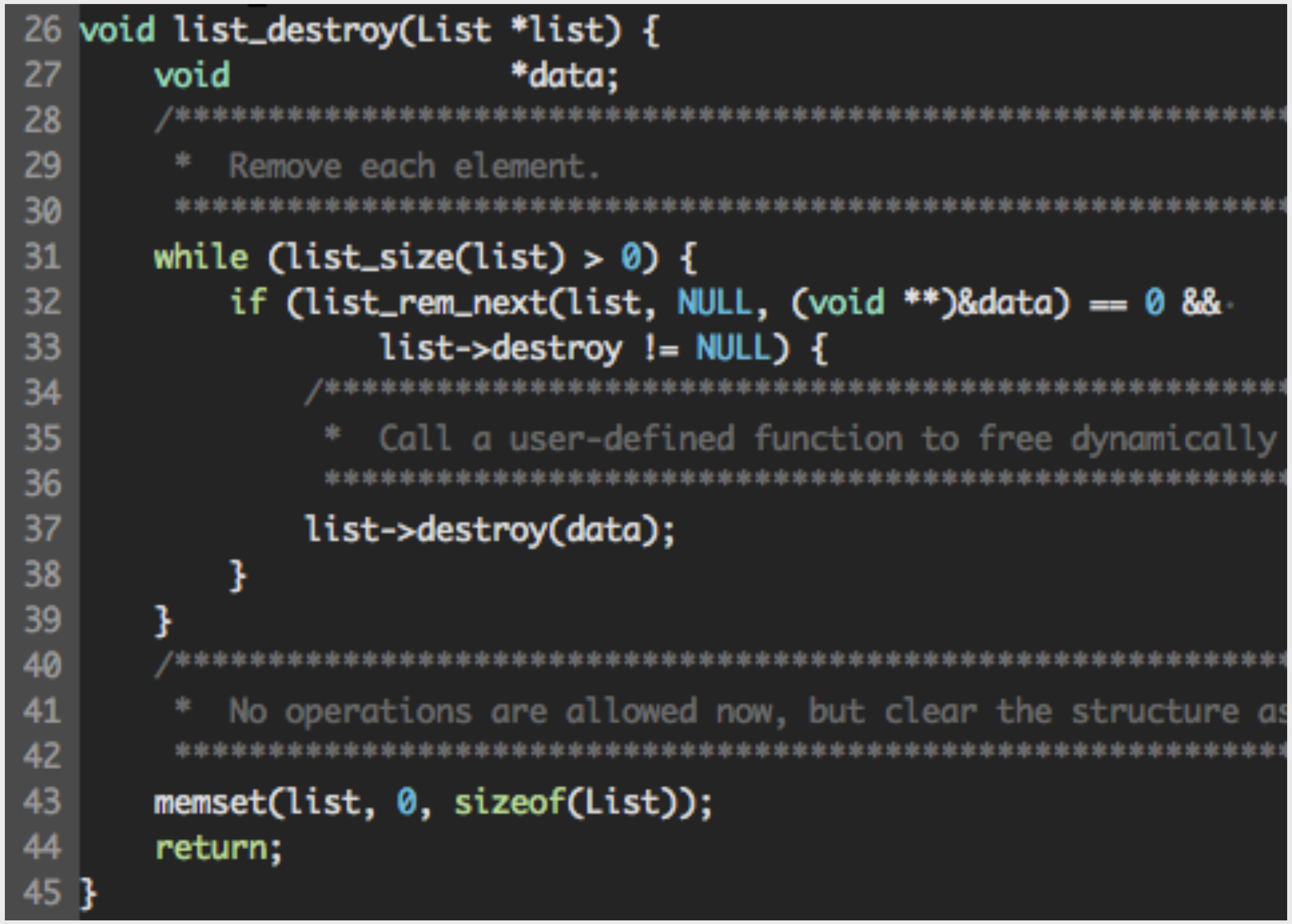


```
void list_init(List *list, void (*destroy)(void *data)) {
        /*****************************************************
         *  Initialize the list.
         *****************************************************
        list->size = 0;
        list->destroy = destroy;
        list->head = NULL;
        list->tail = NULL;
        return;
}
```

use function pointer to pass user-defined destroy function

Advanced Linked List

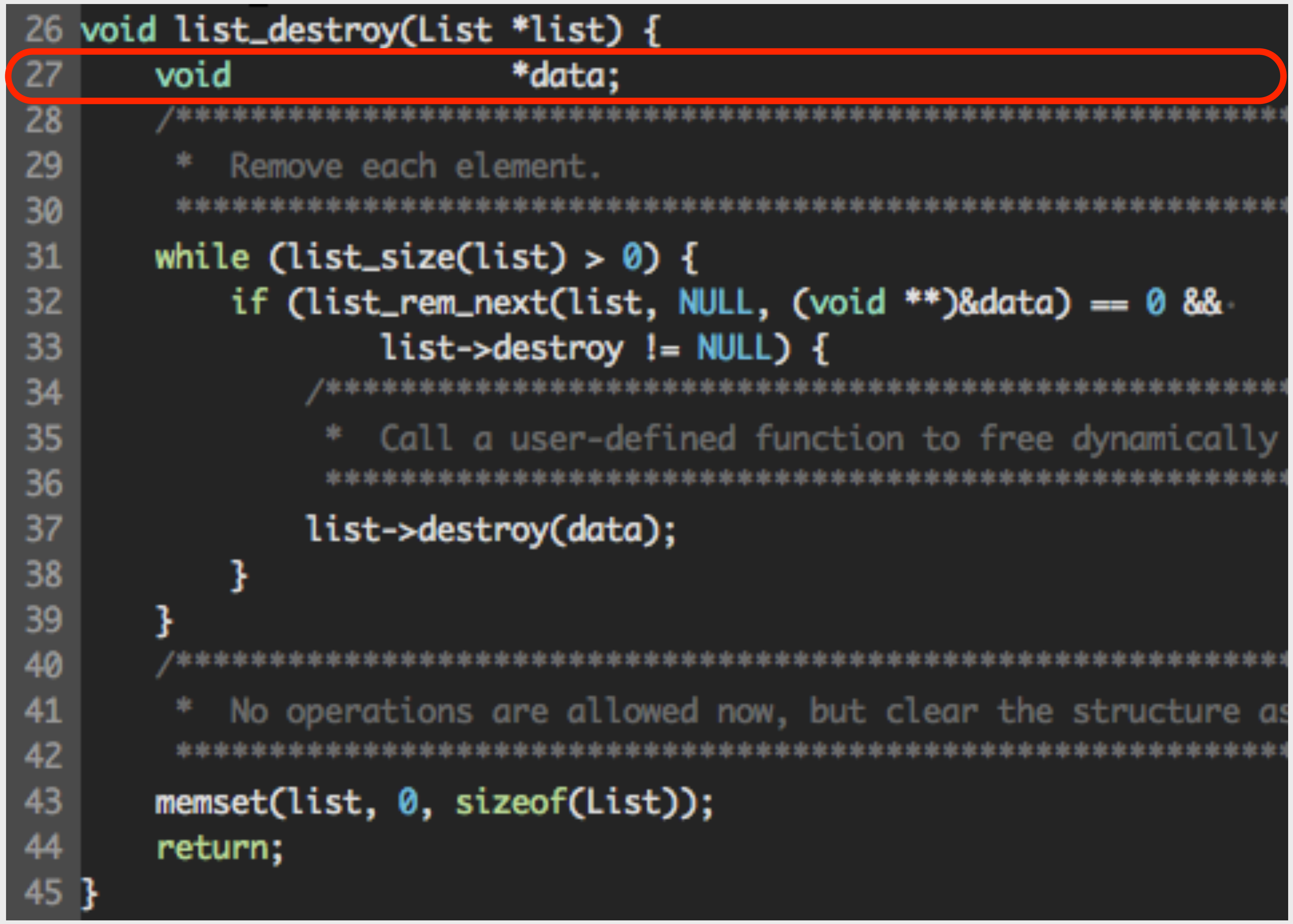
- Example: advList/advList.c



```
void list_destroy(List *list) {
    void                *data;
    /*************************************************************
     *  Remove each element.
     *************************************************************
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 &&
                list->destroy != NULL) {
            /*****************************************************
             *  Call a user-defined function to free dynamically
             *****************************************************
            list->destroy(data);
        }
    }
    /*************************************************************
     *  No operations are allowed now, but clear the structure as
     *************************************************************
    memset(list, 0, sizeof(List));
    return;
}
```

Advanced Linked List

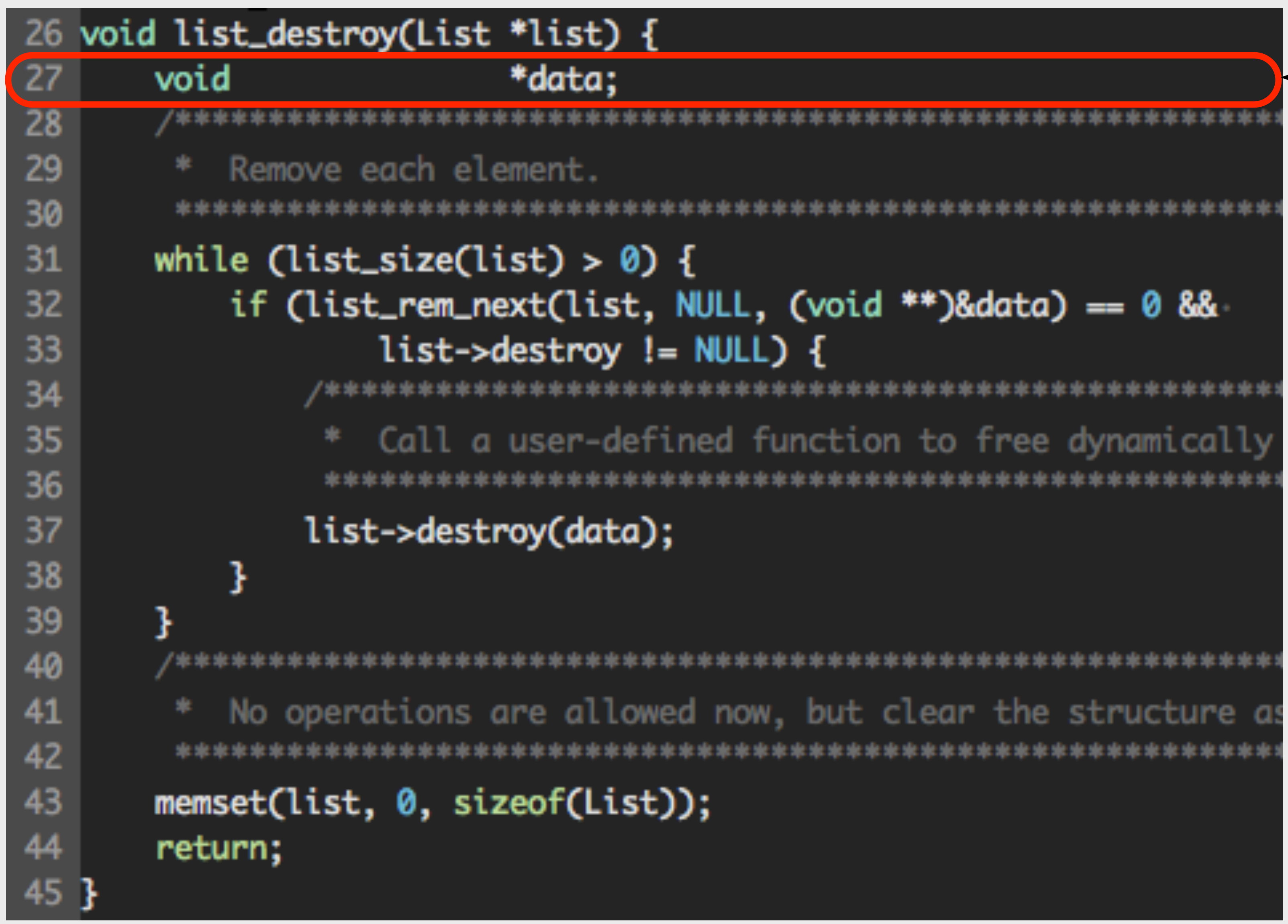
- Example: advList/advList.c



```
void list_destroy(List *list) {
    void                *data;
    /*************************************************************
     *  Remove each element.
     *************************************************************
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 &&
                list->destroy != NULL) {
            /*****************************************************
             *  Call a user-defined function to free dynamically
             *****************************************************
            list->destroy(data);
        }
    }
    /*************************************************************
     *  No operations are allowed now, but clear the structure as
     *************************************************************
    memset(list, 0, sizeof(List));
    return;
}
```

Advanced Linked List

- Example: advList/advList.c

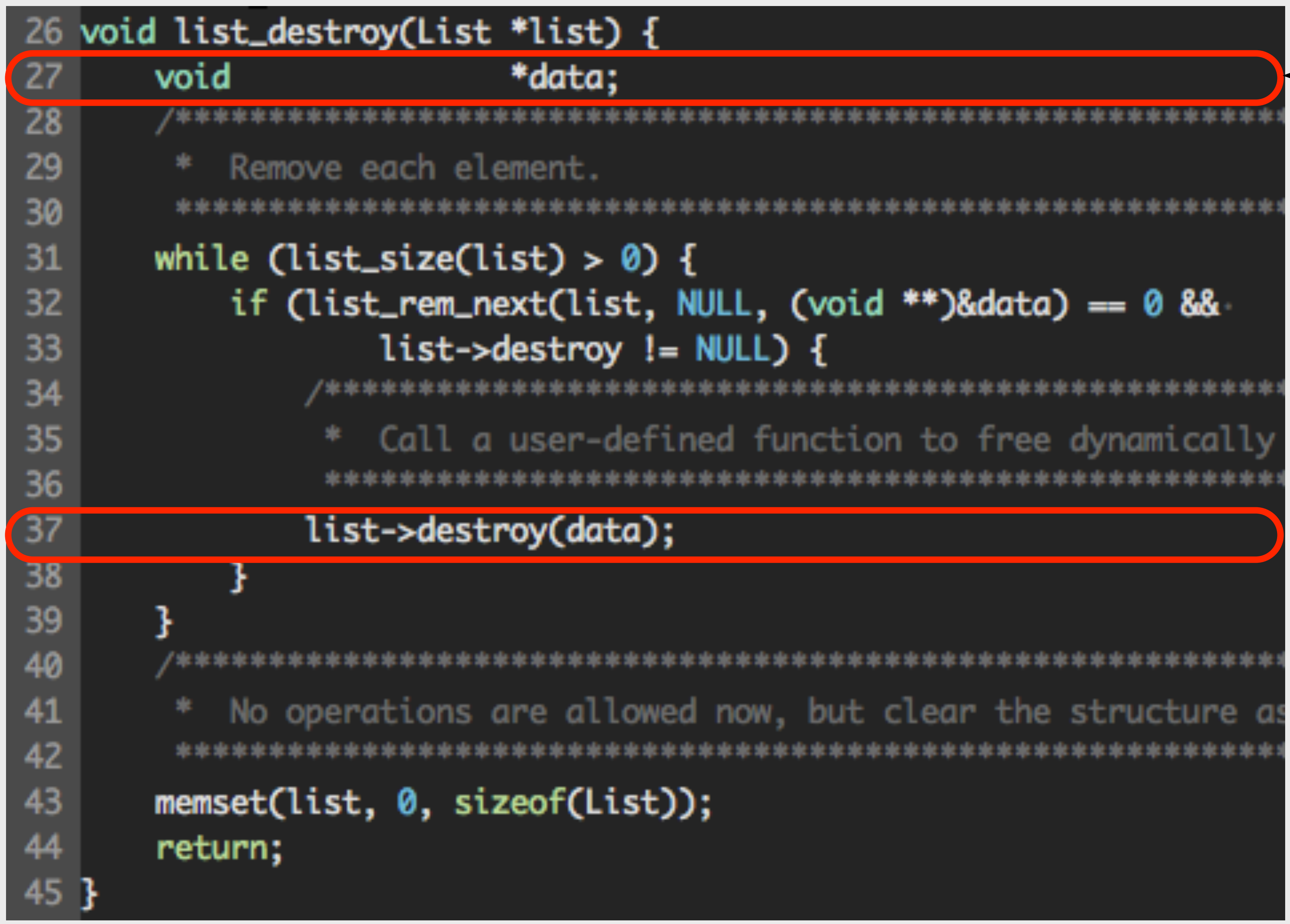


```
void list_destroy(List *list) {
    void                *data;
    /******************************************************
     *  Remove each element.
     ******************************************************
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 &&
                list->destroy != NULL) {
            /**********************************************
             *  Call a user-defined function to free dynamically
             **********************************************
            list->destroy(data);
        }
    }
    /******************************************************
     *  No operations are allowed now, but clear the structure as
     ******************************************************
    memset(list, 0, sizeof(List));
    return;
}
```

use data to store the removed info

Advanced Linked List

- Example: advList/advList.c

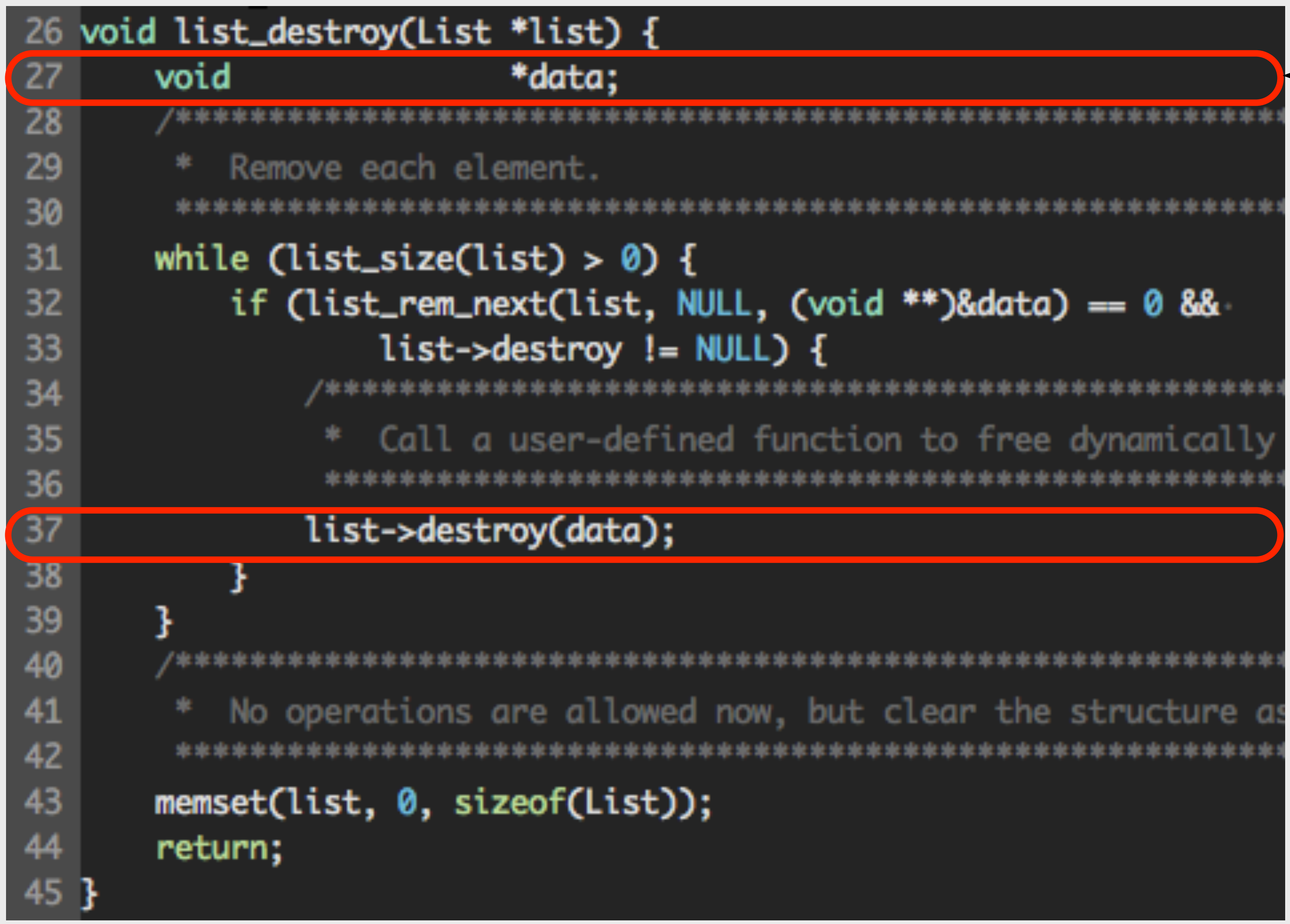


```
void list_destroy(List *list) {
    void                *data;
    /*****************************************************************
     *  Remove each element.
     *****************************************************************
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 &&
                list->destroy != NULL) {
            /*****************************************************
             *  Call a user-defined function to free dynamically
             *****************************************************
            list->destroy(data);
        }
    }
    /*****************************************************************
     *  No operations are allowed now, but clear the structure as
     *****************************************************************
    memset(list, 0, sizeof(List));
    return;
}
```

use data to store the removed info

Advanced Linked List

- Example: advList/advList.c



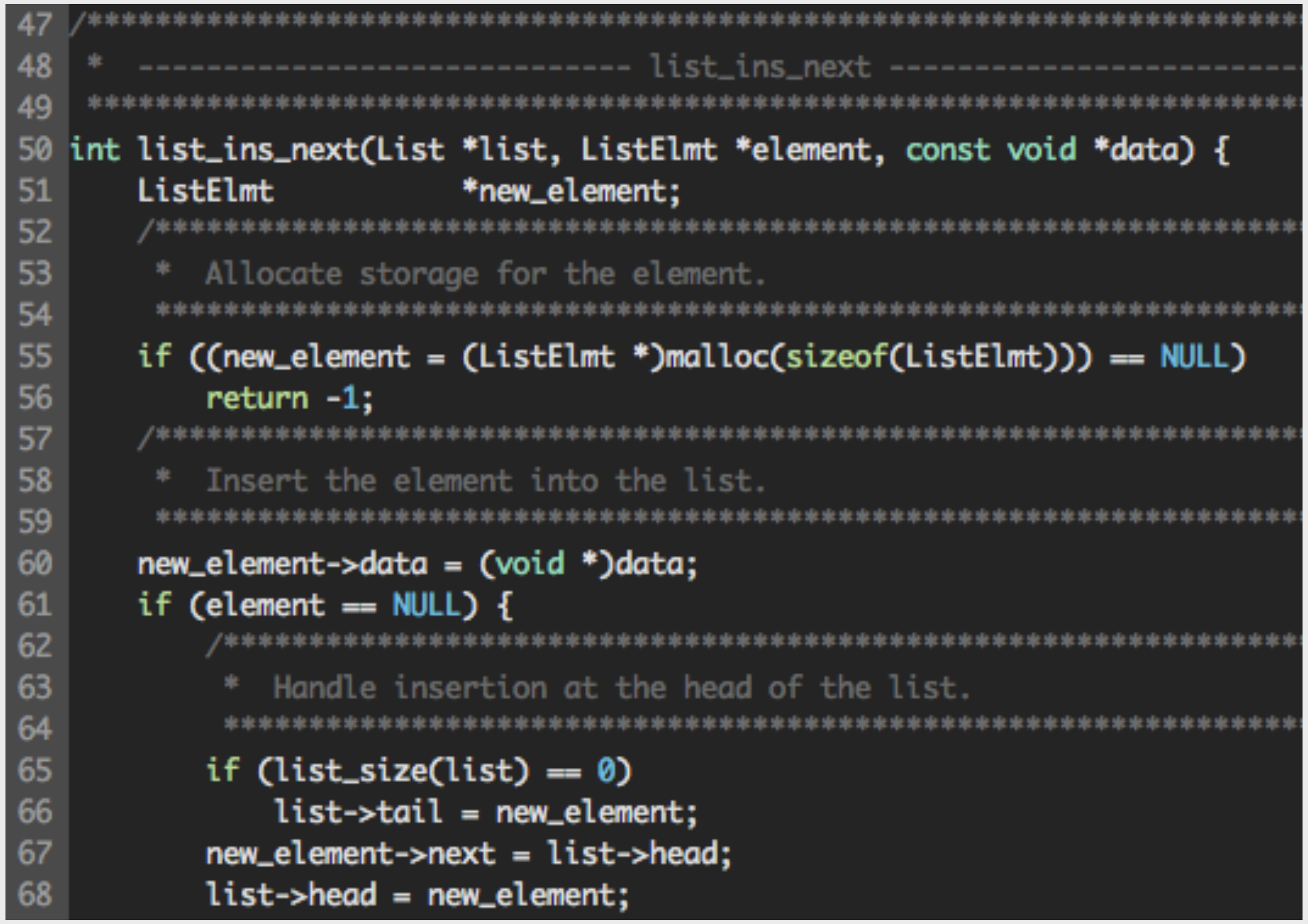
```
void list_destroy(List *list) {
    void                *data;
    /****************************************************************
     *  Remove each element.
     ****************************************************************
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 &&
                list->destroy != NULL) {
            /****************************************************
             *  Call a user-defined function to free dynamically
             ****************************************************
            list->destroy(data);
        }
    }
    /****************************************************************
     *  No operations are allowed now, but clear the structure as
     ****************************************************************
    memset(list, 0, sizeof(List));
    return;
}
```

use data to store the removed info

use user-defined function to free data

Advanced Linked List

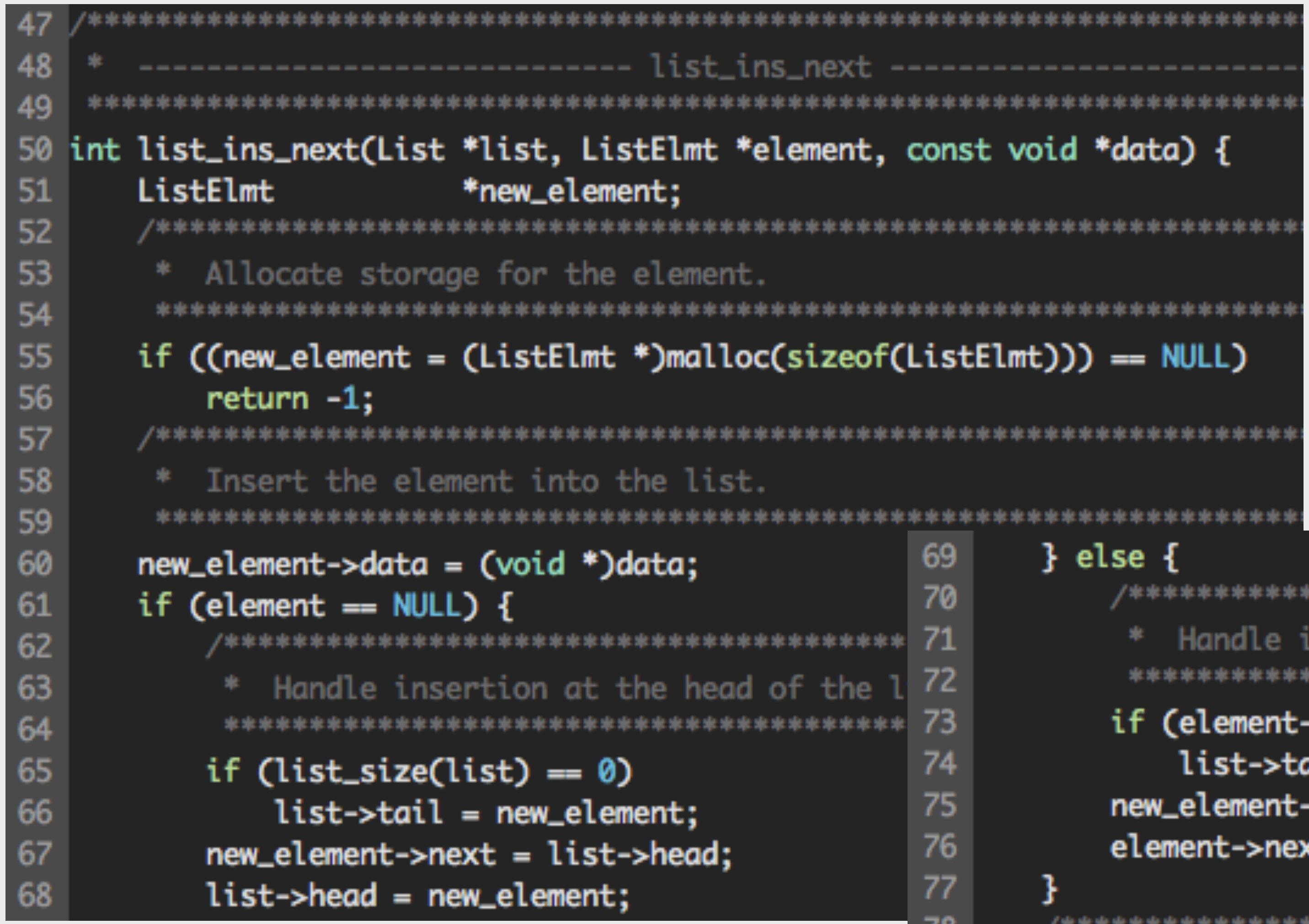
- Example: advList/advList.c



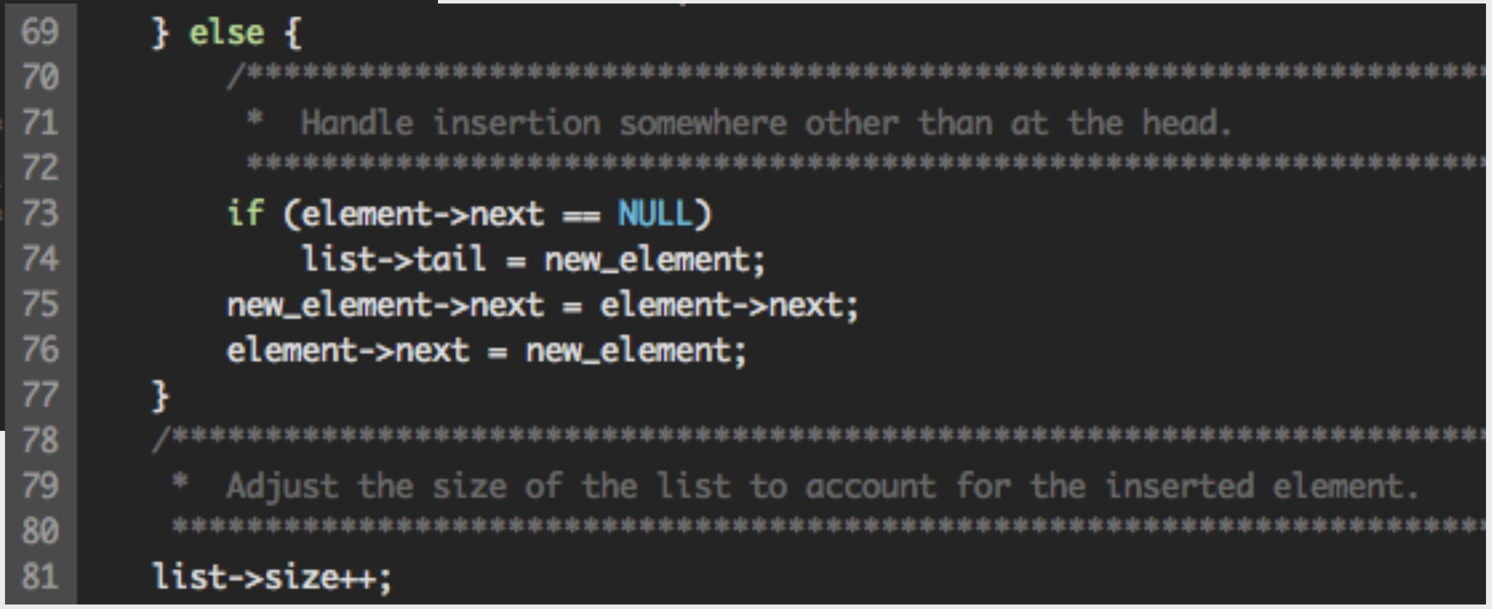
```c
/*****************************************************************************
*  ----------------------------- list_ins_next -----------------------------
*****************************************************************************
int list_ins_next(List *list, ListElmt *element, const void *data) {
    ListElmt           *new_element;
    /*************************************************************************
     *  Allocate storage for the element.
     *************************************************************************
    if ((new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL)
        return -1;
    /*************************************************************************
     *  Insert the element into the list.
     *************************************************************************
    new_element->data = (void *)data;
    if (element == NULL) {
        /*********************************************************************
         *  Handle insertion at the head of the list.
         *********************************************************************
        if (list_size(list) == 0)
            list->tail = new_element;
        new_element->next = list->head;
        list->head = new_element;
```

Advanced Linked List

- Example: advList/advList.c



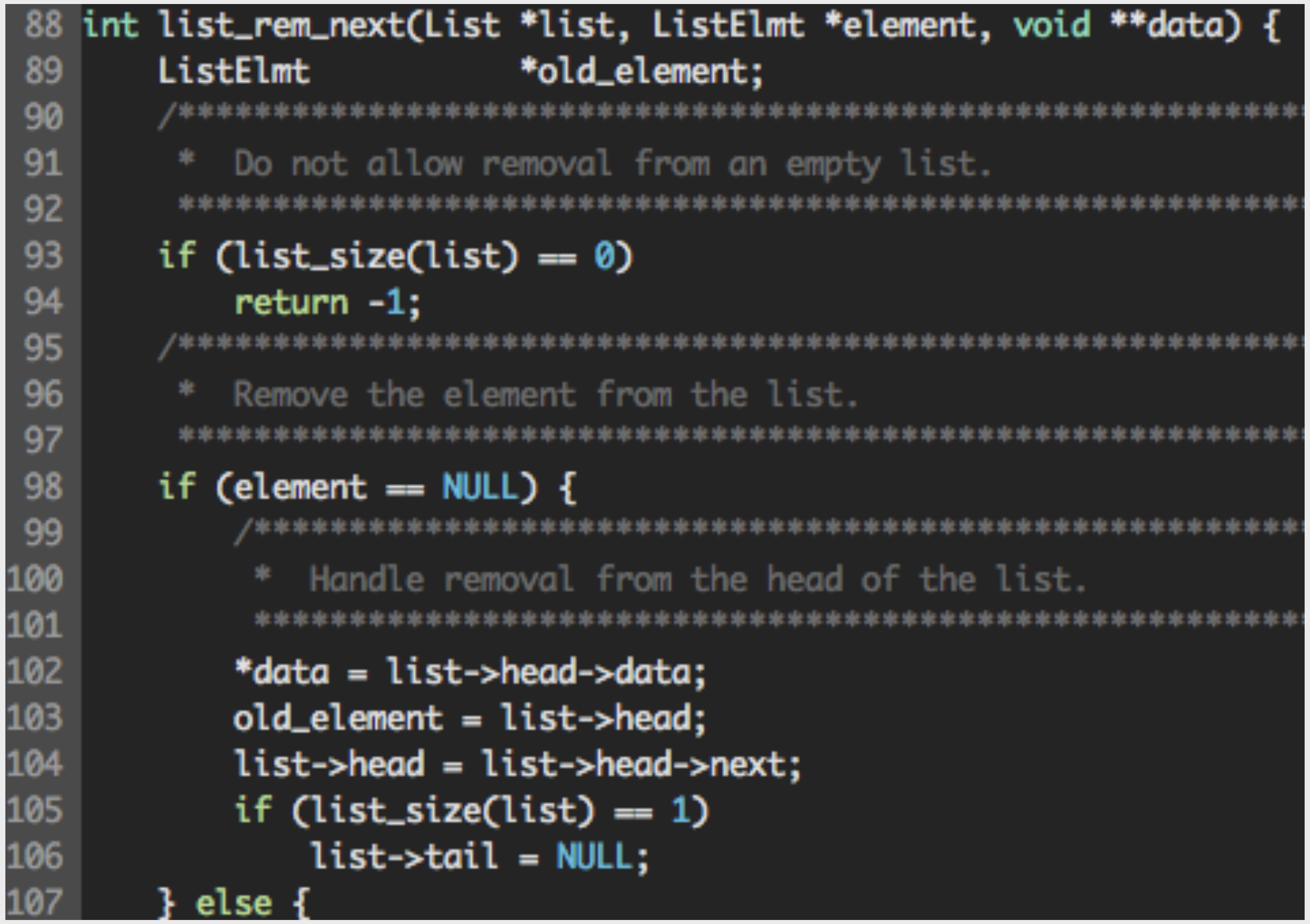
```c
/*****************************************************************************
 *  ---------------------------- list_ins_next -----------------------
 *****************************************************************************
int list_ins_next(List *list, ListElmt *element, const void *data) {
    ListElmt           *new_element;
    /*************************************************************************
     *  Allocate storage for the element.
     *************************************************************************
    if ((new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL)
        return -1;
    /*************************************************************************
     *  Insert the element into the list.
     *************************************************************************
    new_element->data = (void *)data;
    if (element == NULL) {
        /*****************************************
         *  Handle insertion at the head of the l
         *****************************************
        if (list_size(list) == 0)
            list->tail = new_element;
        new_element->next = list->head;
        list->head = new_element;
```



```c
    } else {
        /*********************************************************************
         *  Handle insertion somewhere other than at the head.
         *********************************************************************
        if (element->next == NULL)
            list->tail = new_element;
        new_element->next = element->next;
        element->next = new_element;
    }
    /*************************************************************************
     *  Adjust the size of the list to account for the inserted element.
     *************************************************************************
    list->size++;
```

Advanced Linked List

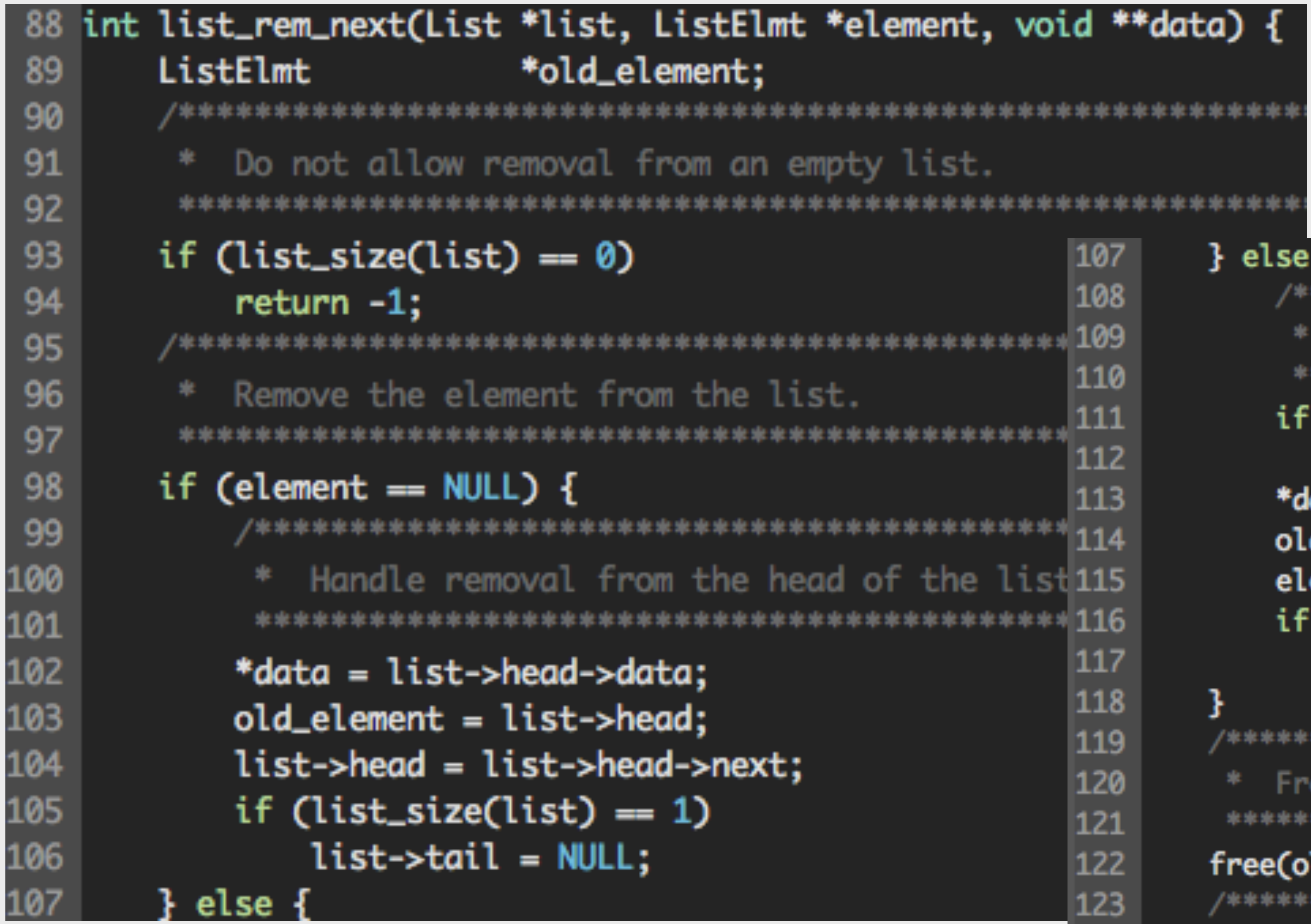
- Example: advList/advList.c



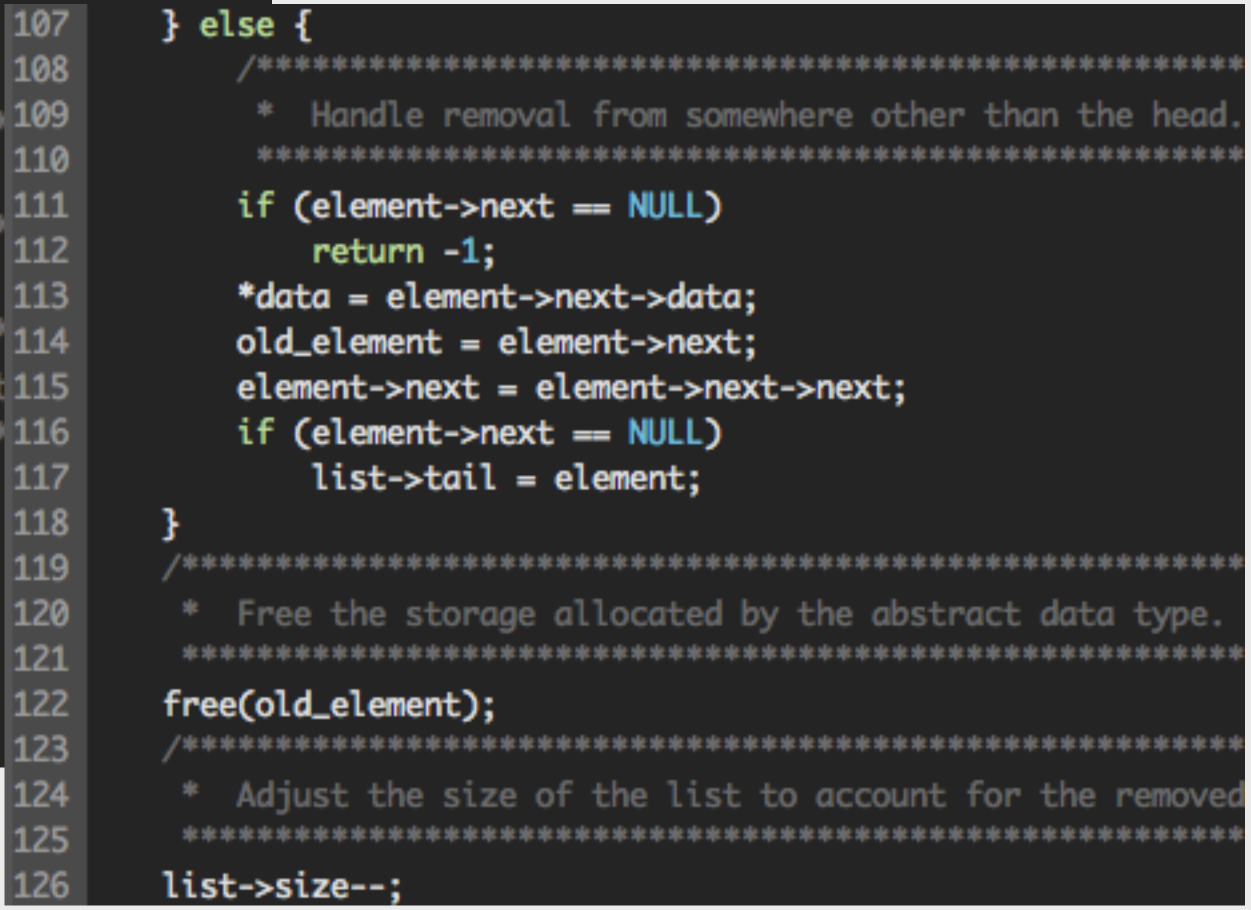
```c
int list_rem_next(List *list, ListElmt *element, void **data) {
    ListElmt            *old_element;
    /************************************************************
     *  Do not allow removal from an empty list.
     ************************************************************
    if (list_size(list) == 0)
        return -1;
    /************************************************************
     *  Remove the element from the list.
     ************************************************************
    if (element == NULL) {
        /********************************************************
         *  Handle removal from the head of the list.
         ********************************************************
        *data = list->head->data;
        old_element = list->head;
        list->head = list->head->next;
        if (list_size(list) == 1)
            list->tail = NULL;
    } else {
```

Advanced Linked List

- Example: advList/advList.c



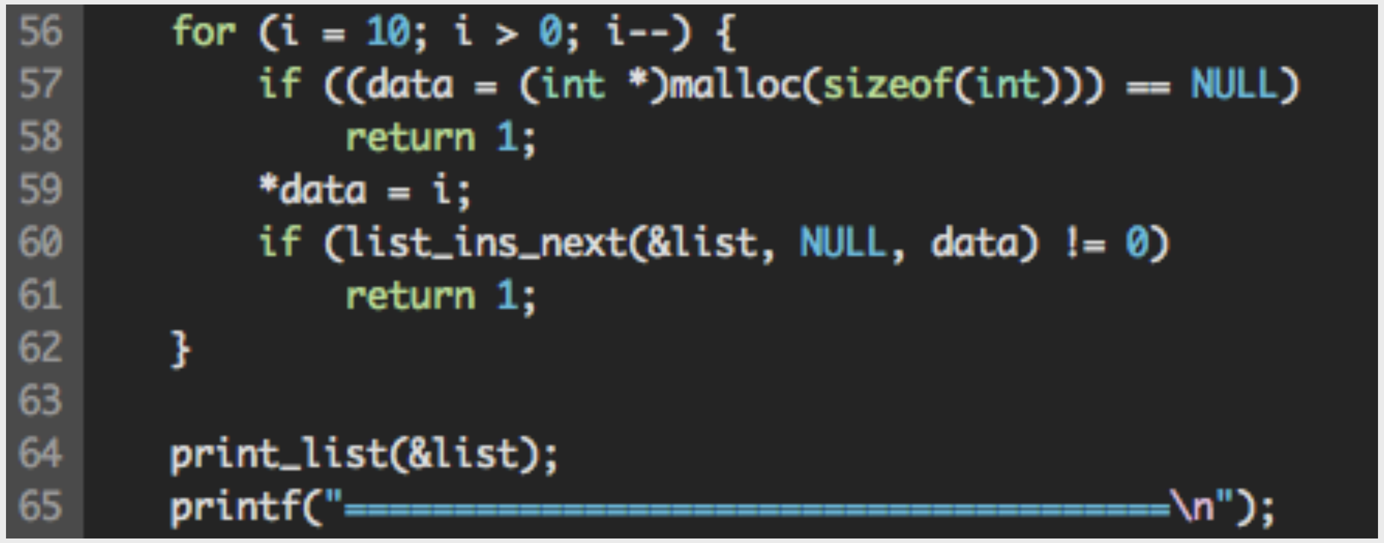
```c
int list_rem_next(List *list, ListElmt *element, void **data) {
    ListElmt            *old_element;
    /*****************************************************
     *  Do not allow removal from an empty list.
     *****************************************************
    if (list_size(list) == 0)
        return -1;
    /*********************************************
     *  Remove the element from the list.
     *********************************************
    if (element == NULL) {
        /*****************************************
         *  Handle removal from the head of the list
         *****************************************
        *data = list->head->data;
        old_element = list->head;
        list->head = list->head->next;
        if (list_size(list) == 1)
            list->tail = NULL;
    } else {
```



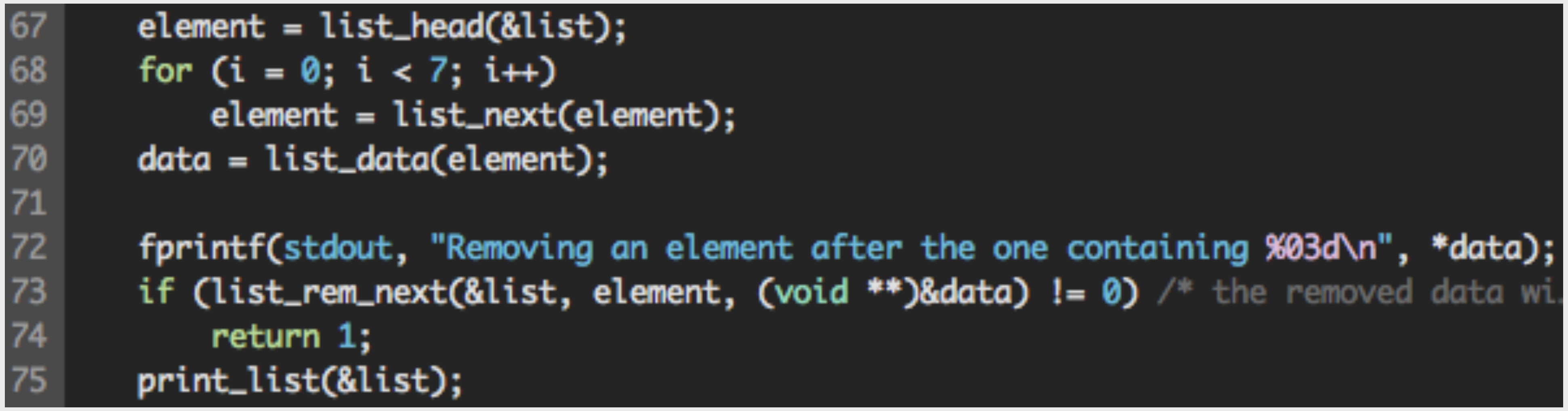
```c
    } else {
        /*************************************************
         *  Handle removal from somewhere other than the head.
         *************************************************
        if (element->next == NULL)
            return -1;
        *data = element->next->data;
        old_element = element->next;
        element->next = element->next->next;
        if (element->next == NULL)
            list->tail = element;
    }
    /****************************************************
     *  Free the storage allocated by the abstract data type.
     ****************************************************
    free(old_element);
    /****************************************************
     *  Adjust the size of the list to account for the removed
     ****************************************************
    list->size--;
```

Advanced Linked List

- Example: advList/advTest.c

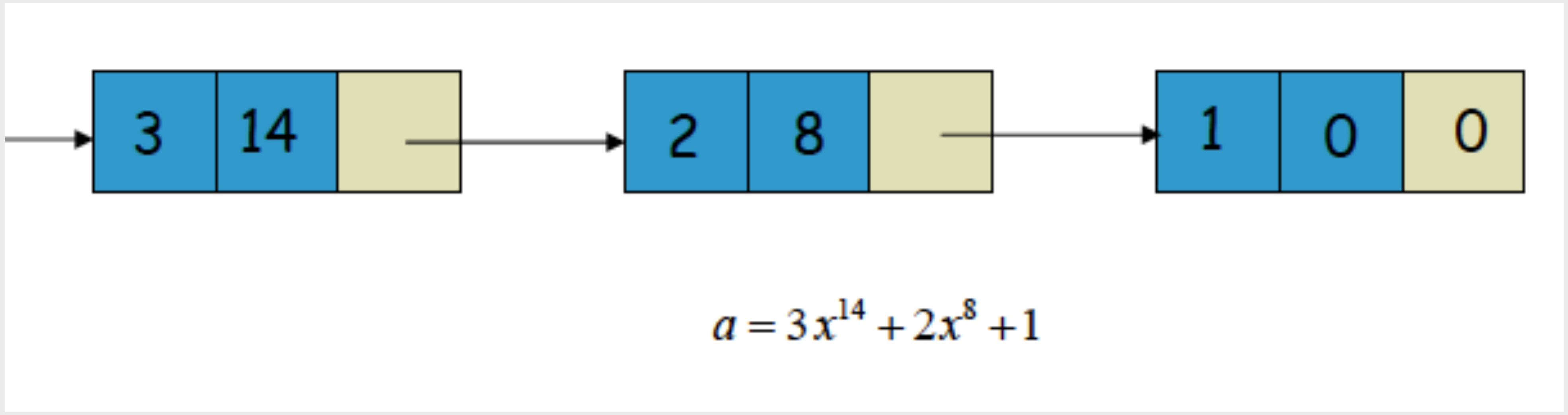


```c
    for (i = 10; i > 0; i--) {
        if ((data = (int *)malloc(sizeof(int))) == NULL)
            return 1;
        *data = i;
        if (list_ins_next(&list, NULL, data) != 0)
            return 1;
    }

    print_list(&list);
    printf("========================================\n");
```

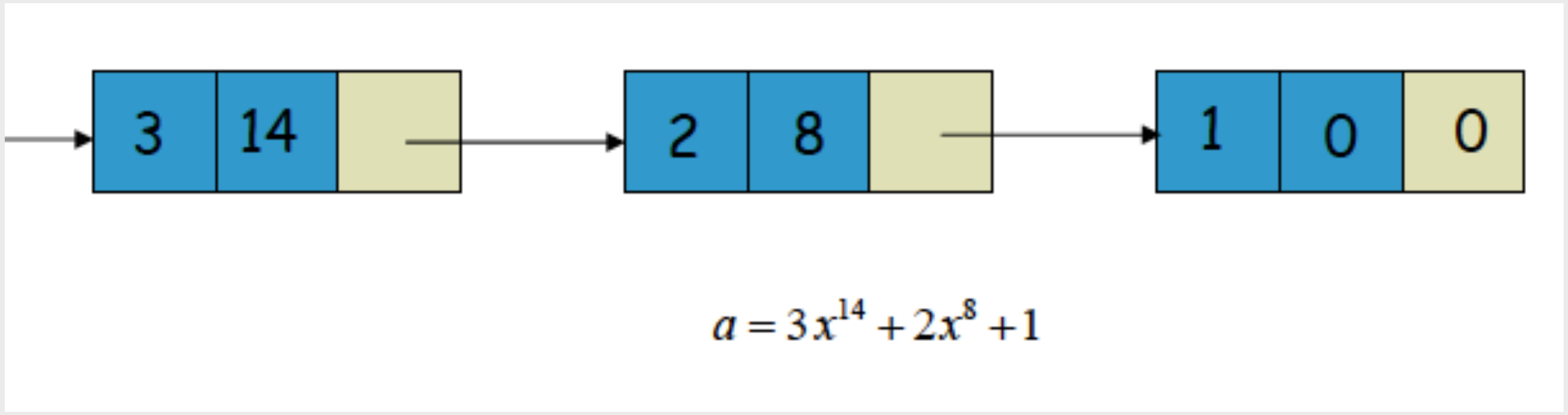


```c
    element = list_head(&list);
    for (i = 0; i < 7; i++)
        element = list_next(element);
    data = list_data(element);

    fprintf(stdout, "Removing an element after the one containing %03d\n", *data);
    if (list_rem_next(&list, element, (void **)&data) != 0) /* the removed data wi
        return 1;
    print_list(&list);
```

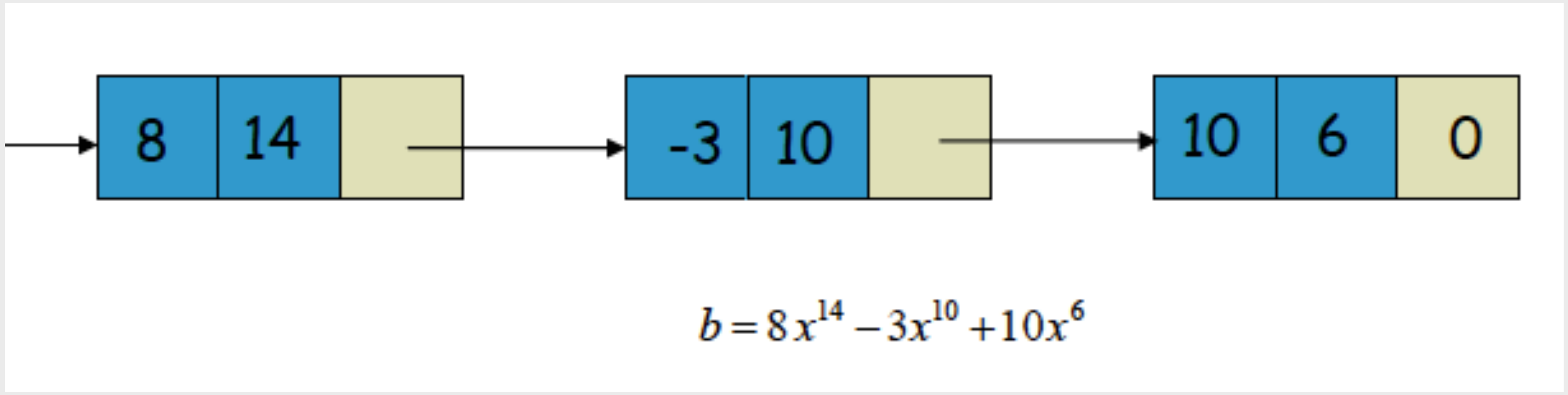
Linked List Example -- Polynomials



Linked List Example -- Polynomials



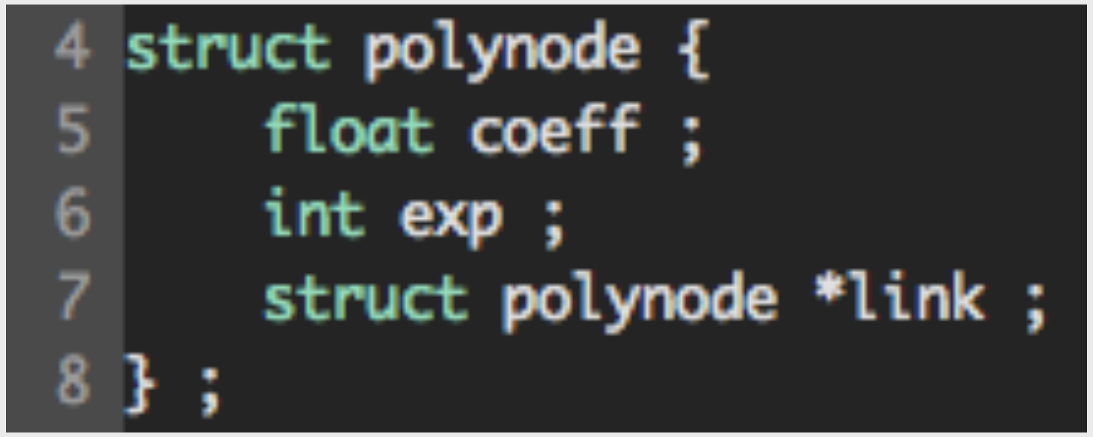
$a = 3x^{14} + 2x^{8} + 1$



$b = 8x^{14} - 3x^{10} + 10x^{6}$

Linked List Example -- Polynomials

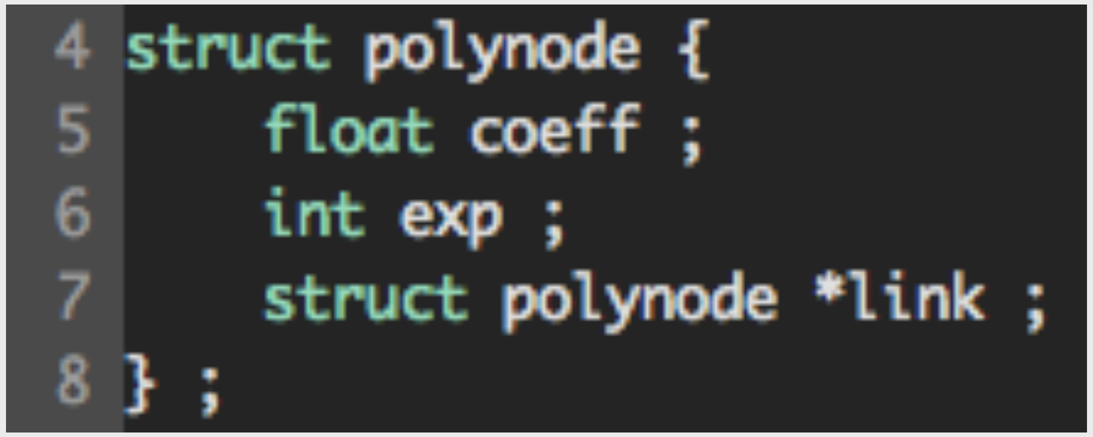
- Example: poly/poly.c



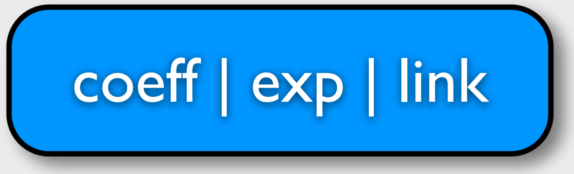
```
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

Linked List Example -- Polynomials

- Example: poly/poly.c

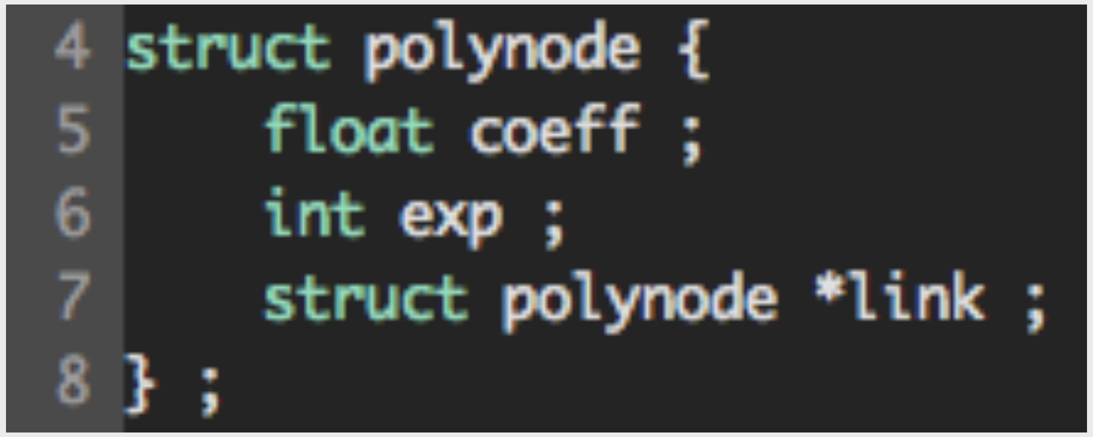


```c
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

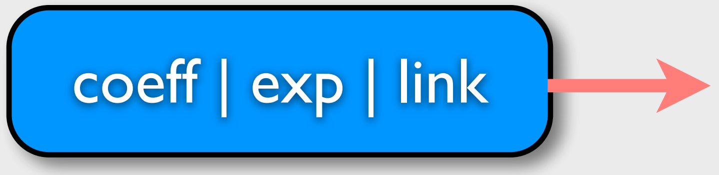


Linked List Example -- Polynomials

- Example: poly/poly.c

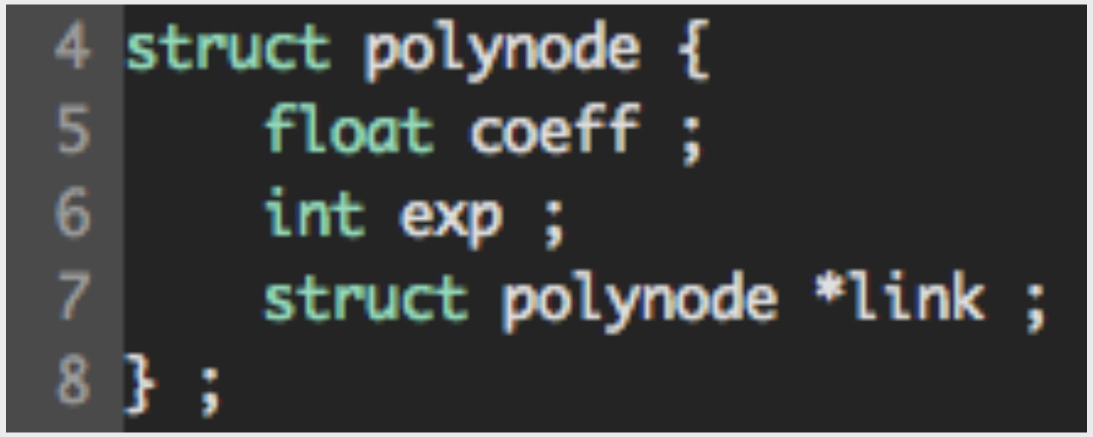


```c
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

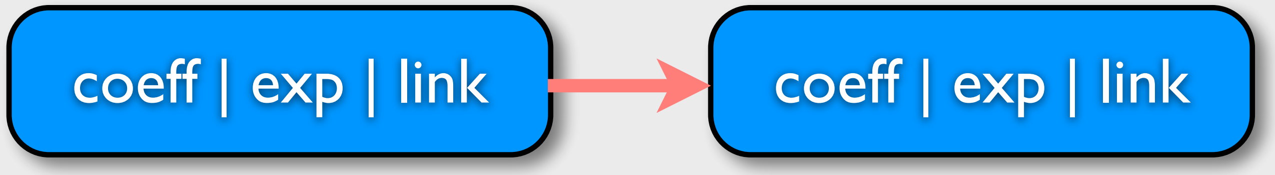


Linked List Example -- Polynomials

- Example: poly/poly.c

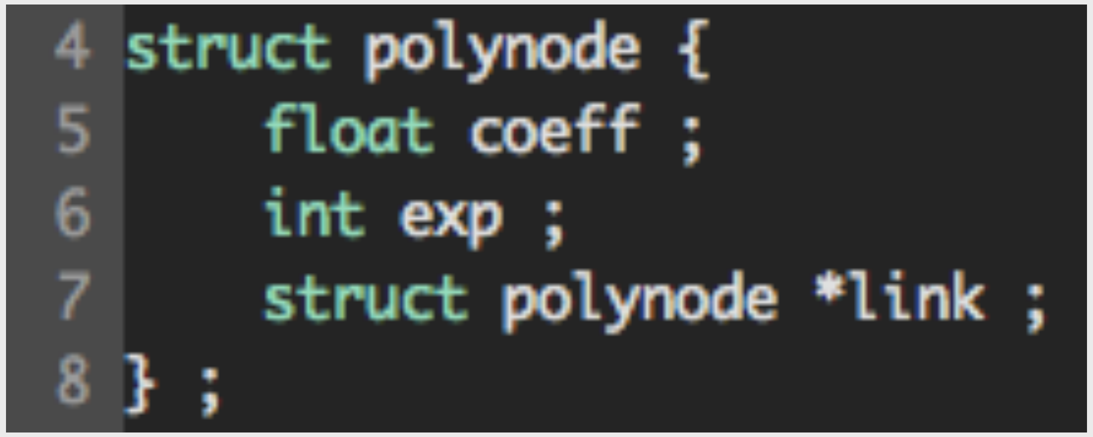


```c
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

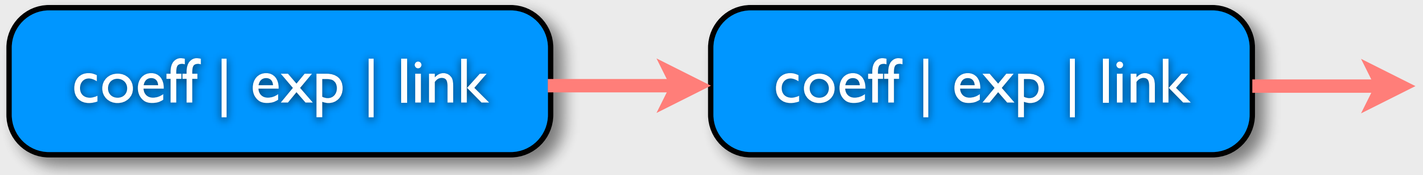


Linked List Example -- Polynomials

- Example: poly/poly.c

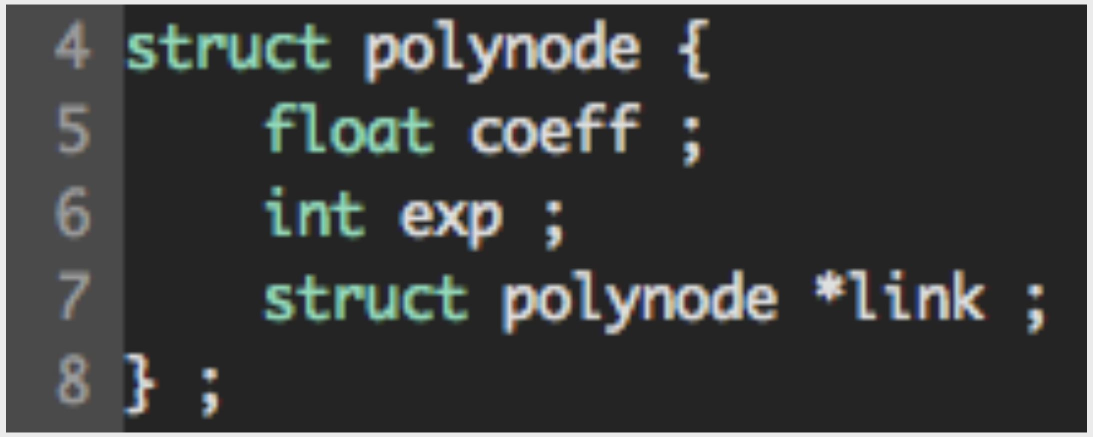


```c
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

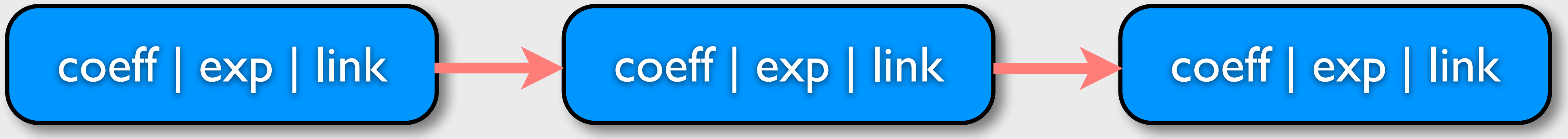


Linked List Example -- Polynomials

- Example: poly/poly.c



```c
struct polynode {
    float coeff ;
    int exp ;
    struct polynode *link ;
} ;
```

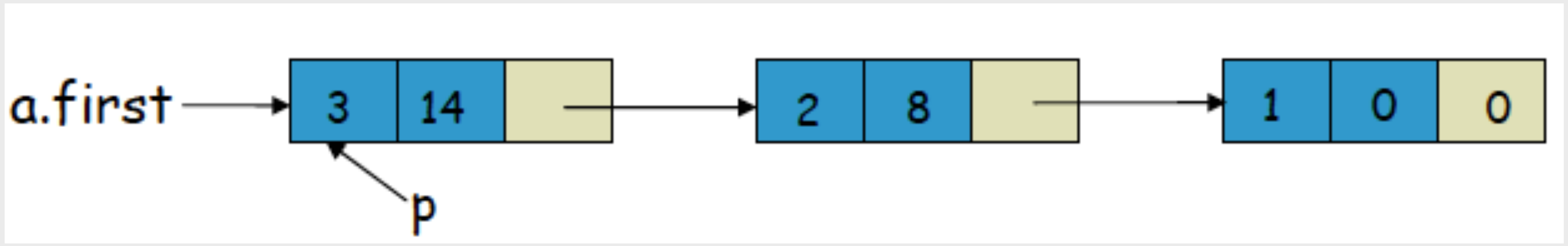


Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting
    - e.g., adding two polynomials a and b

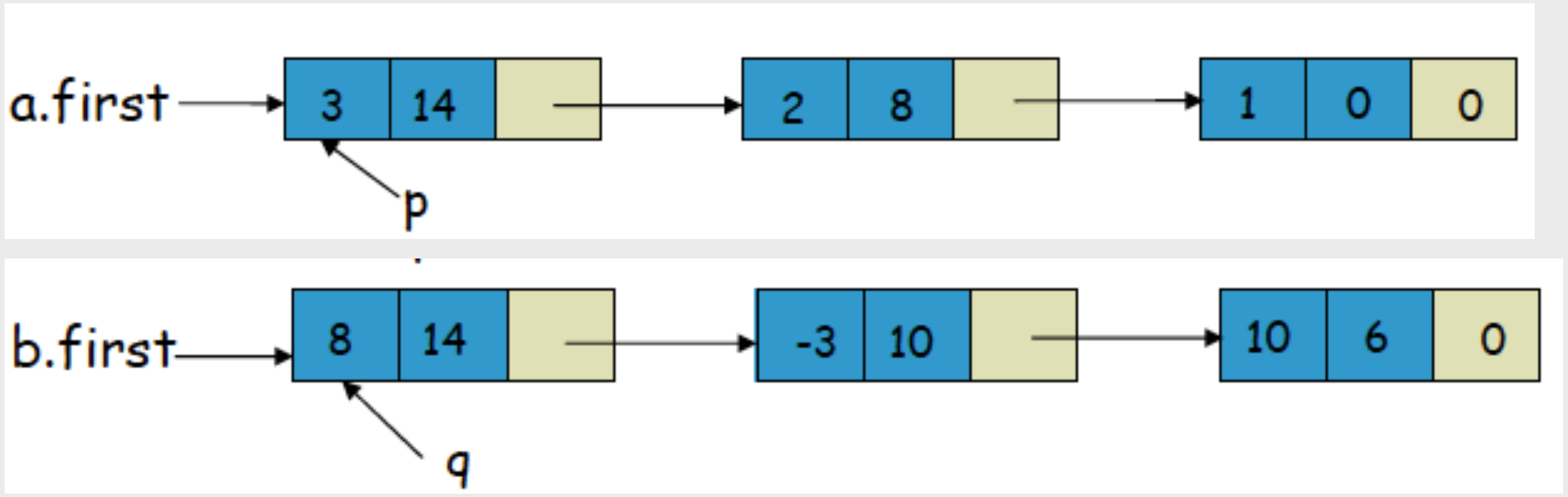
Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting
  - e.g., adding two polynomials a and b



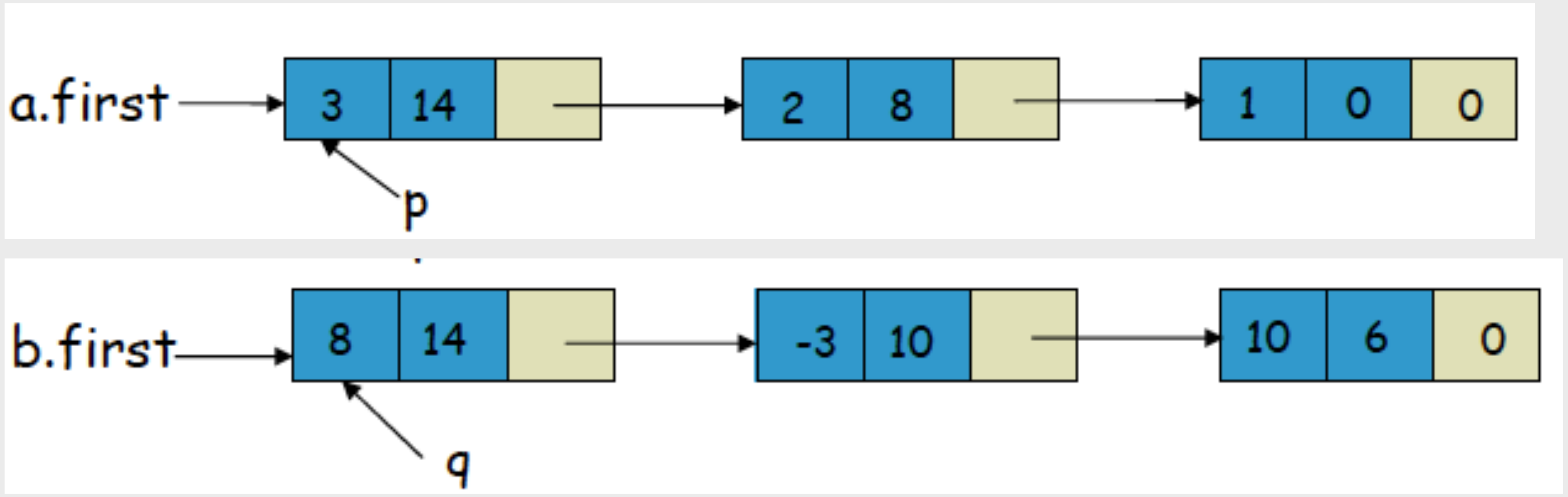
Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting
  - e.g., adding two polynomials a and b



Operating on Polynomials

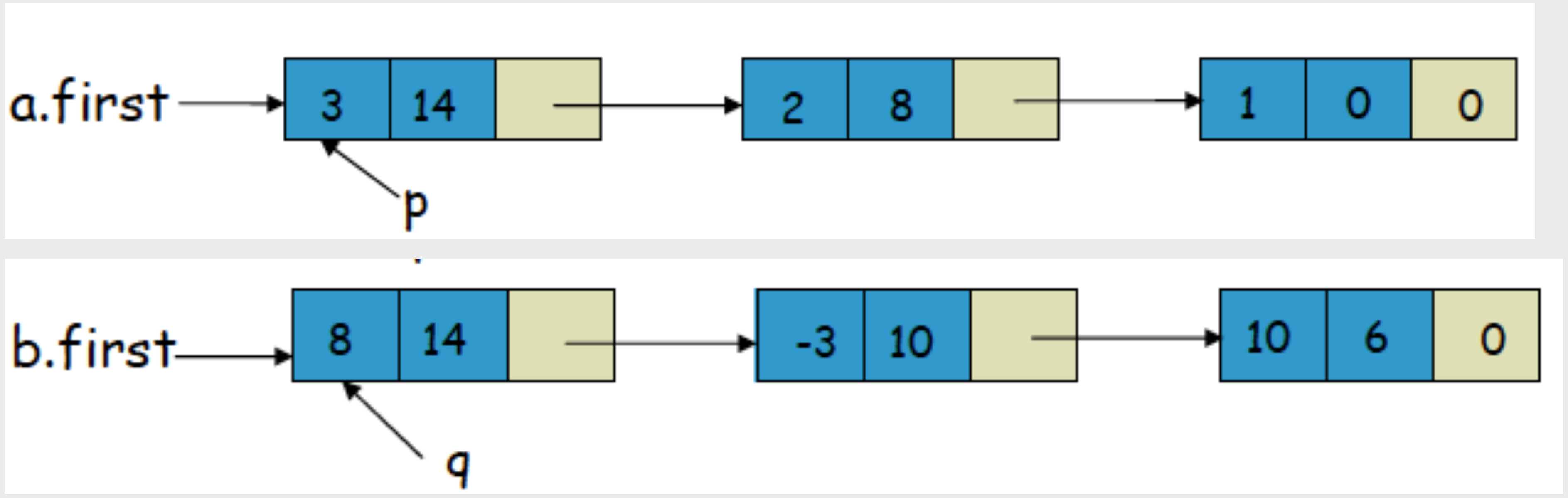
- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting
  - e.g., adding two polynomials a and b



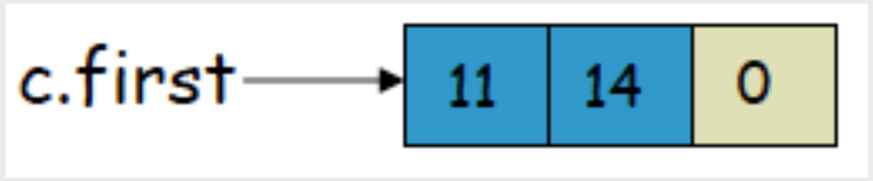
```
p->exp == q->exp
```

Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting
  - e.g., adding two polynomials a and b

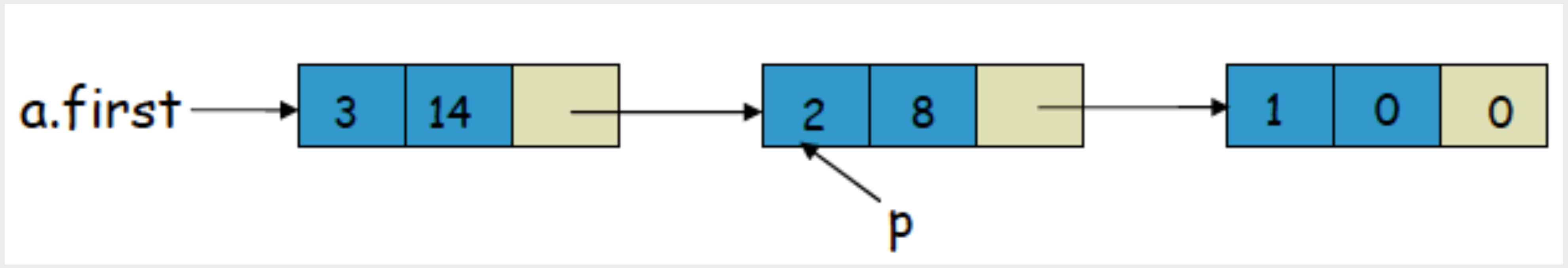


`p->exp == q->exp`

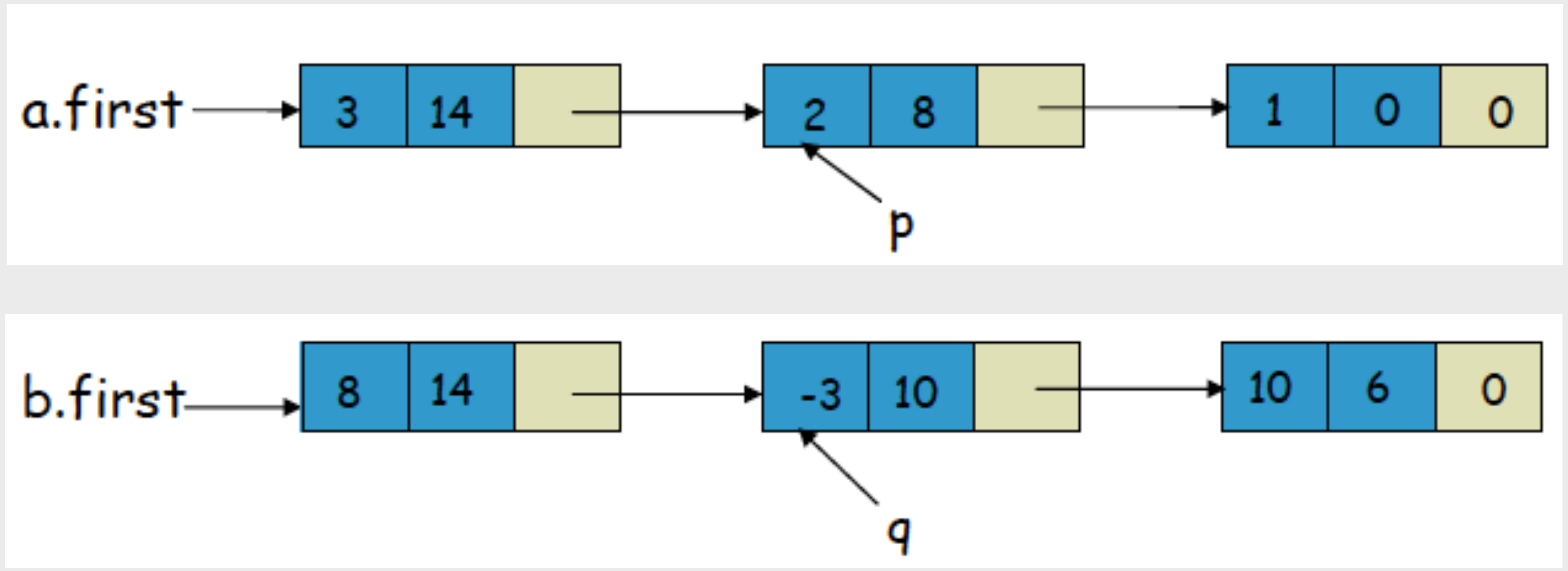


Operating on Polynomials

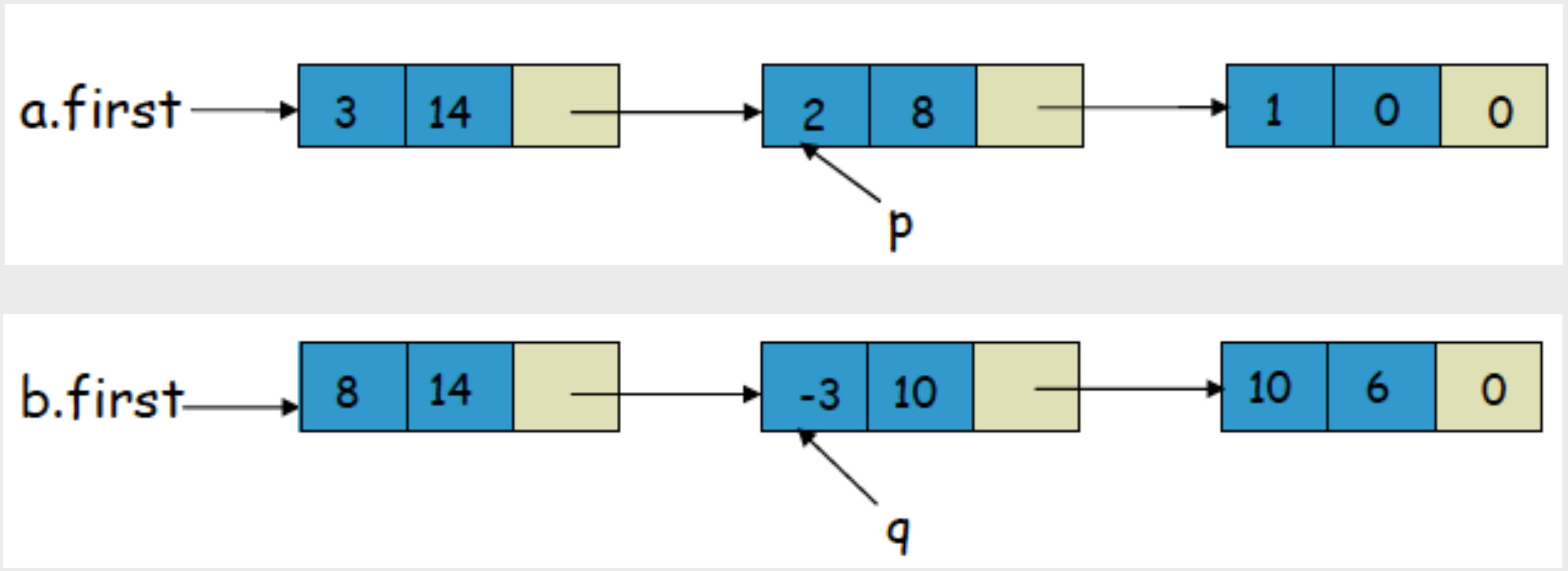
Operating on Polynomials



Operating on Polynomials

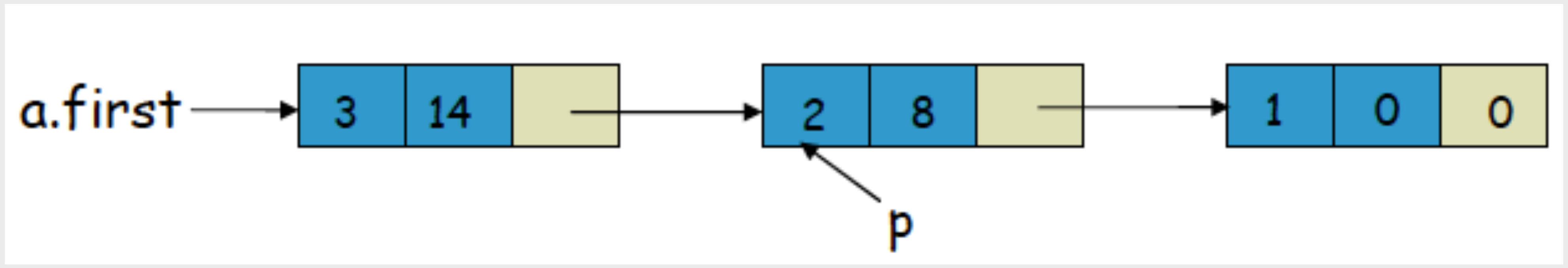


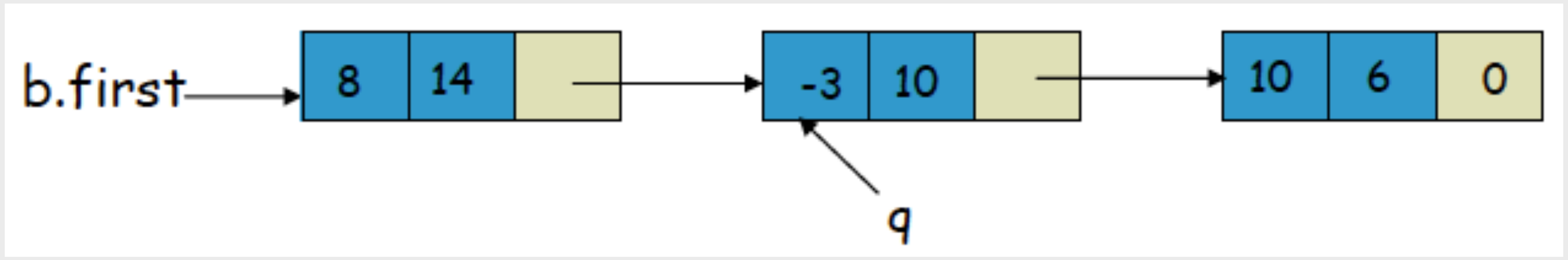
Operating on Polynomials



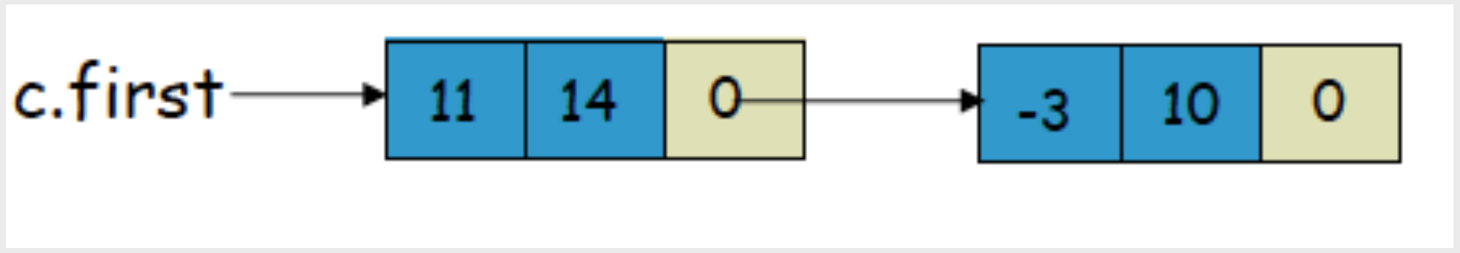
```
p->exp  <  q->exp
```

Operating on Polynomials



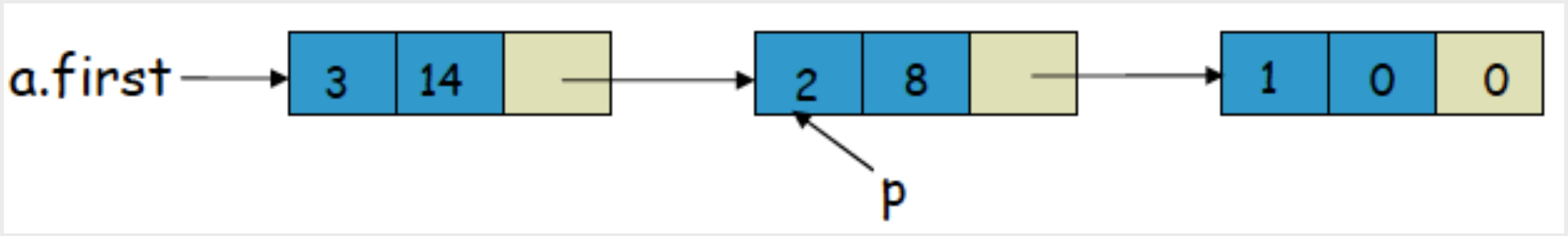


```
p->exp  <  q->exp
```

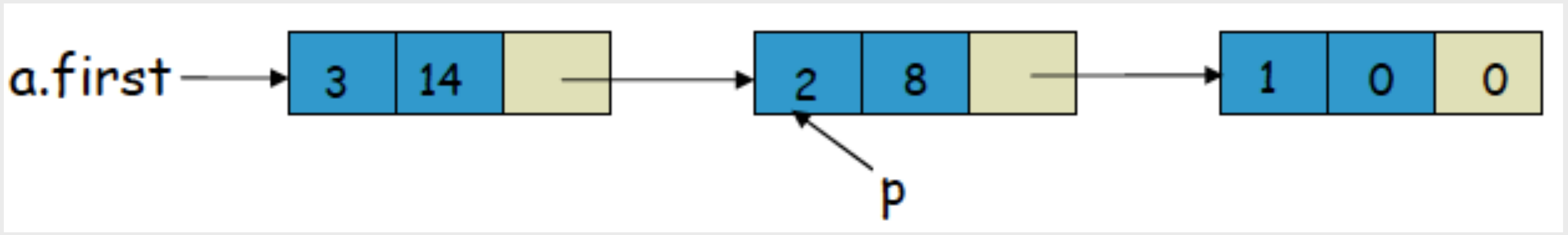


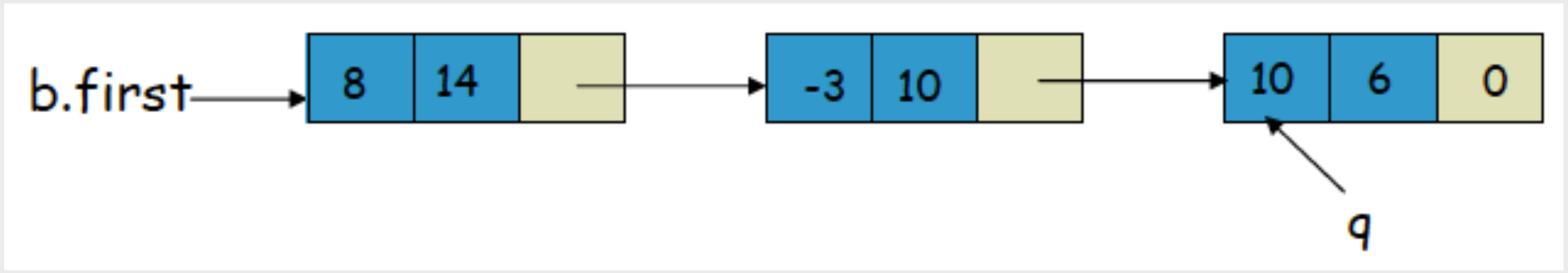
Operating on Polynomials

Operating on Polynomials

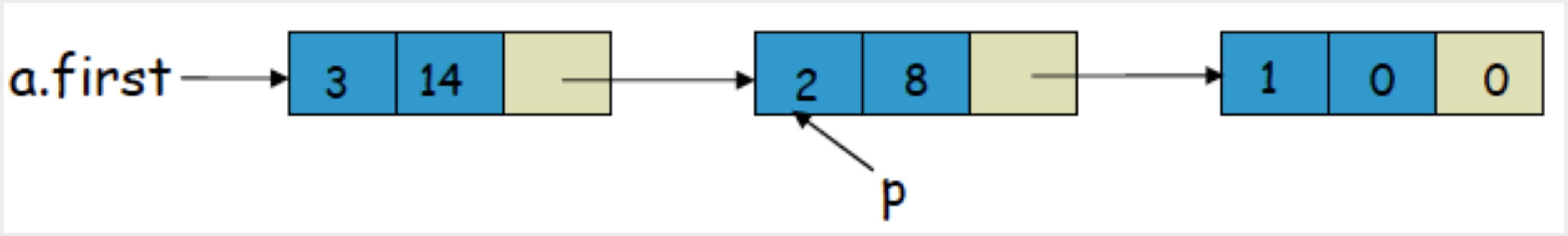


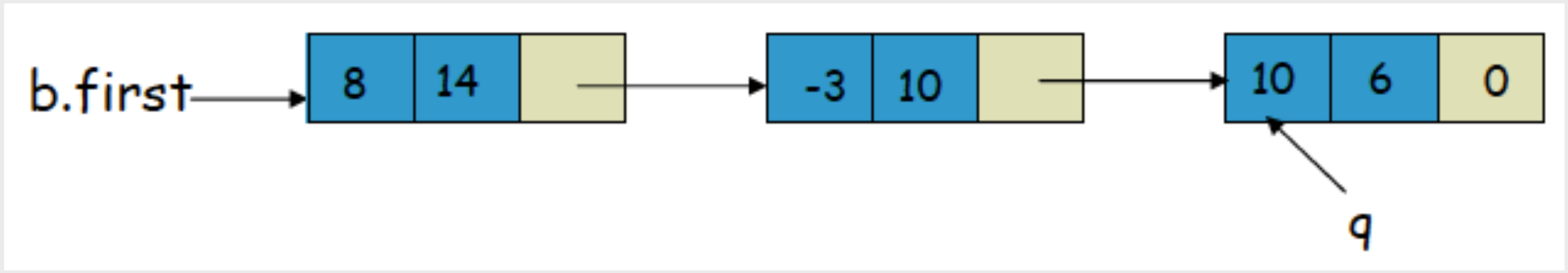
Operating on Polynomials





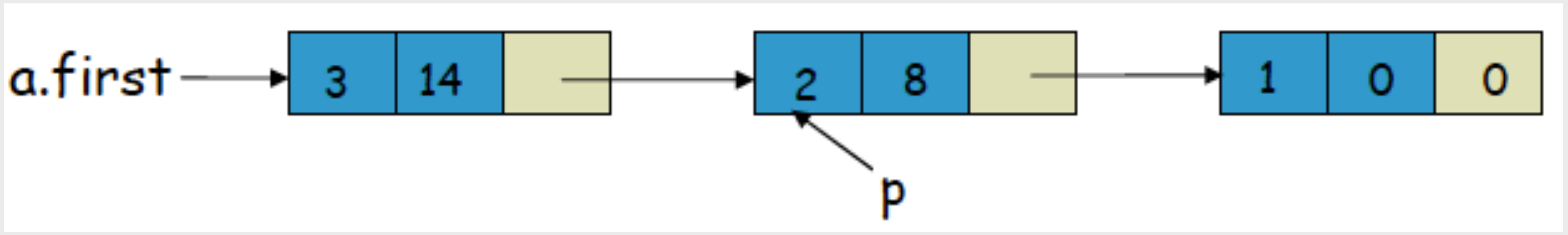
Operating on Polynomials

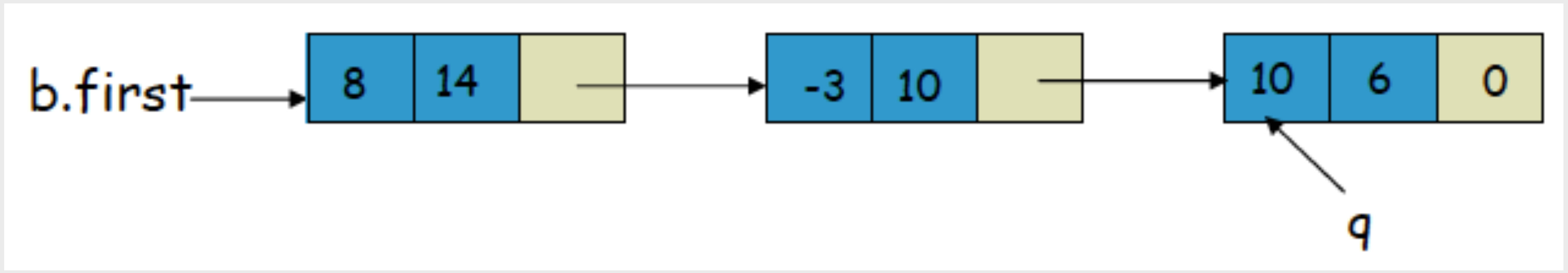




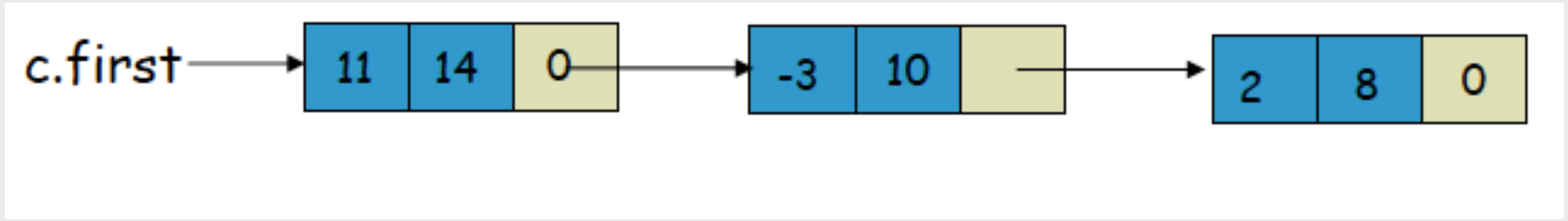
```
p->exp   >   q->exp
```

Operating on Polynomials



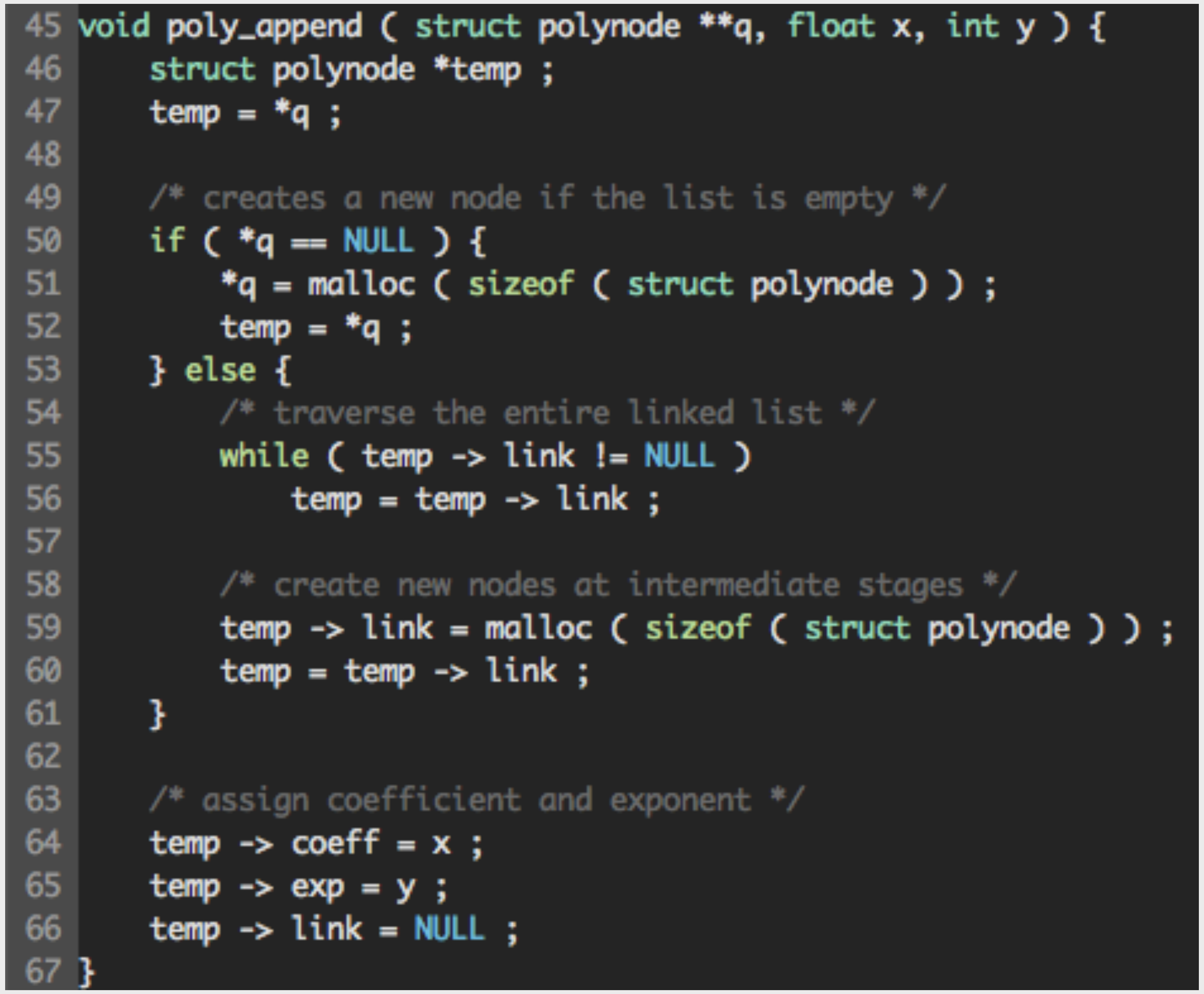


```
p->exp  >  q->exp
```

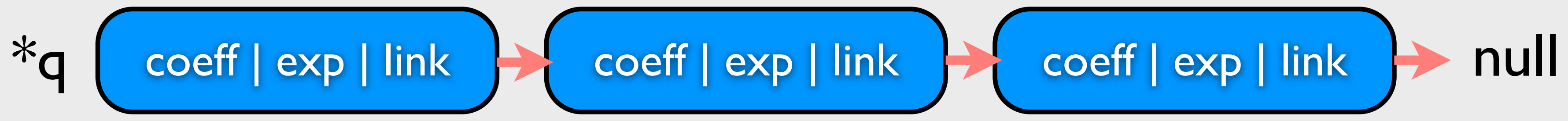


Example: poly/poly.c

- poly_append

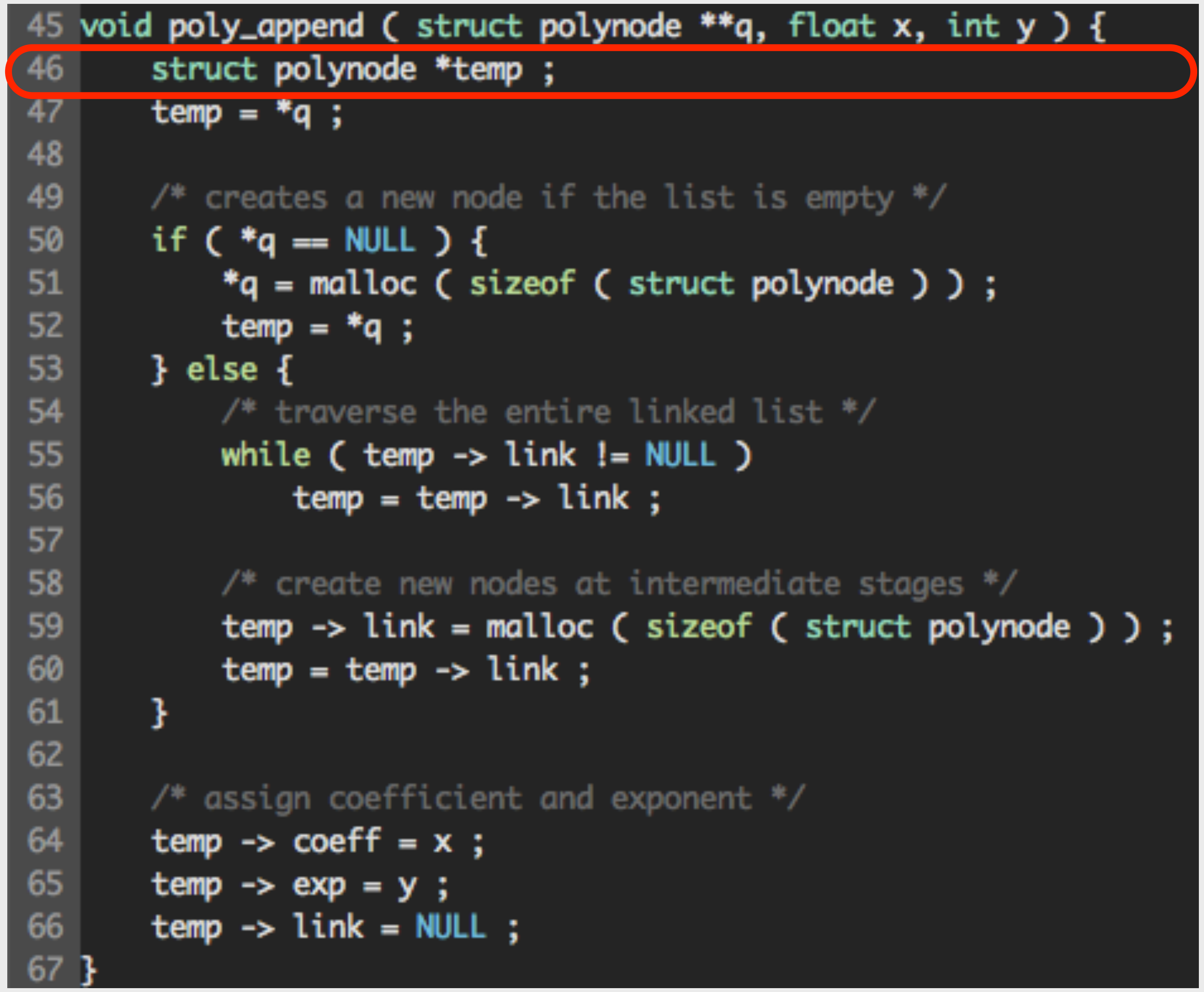


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

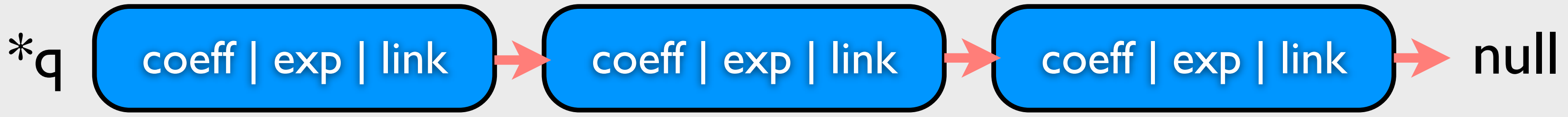


Example: poly/poly.c

- poly_append

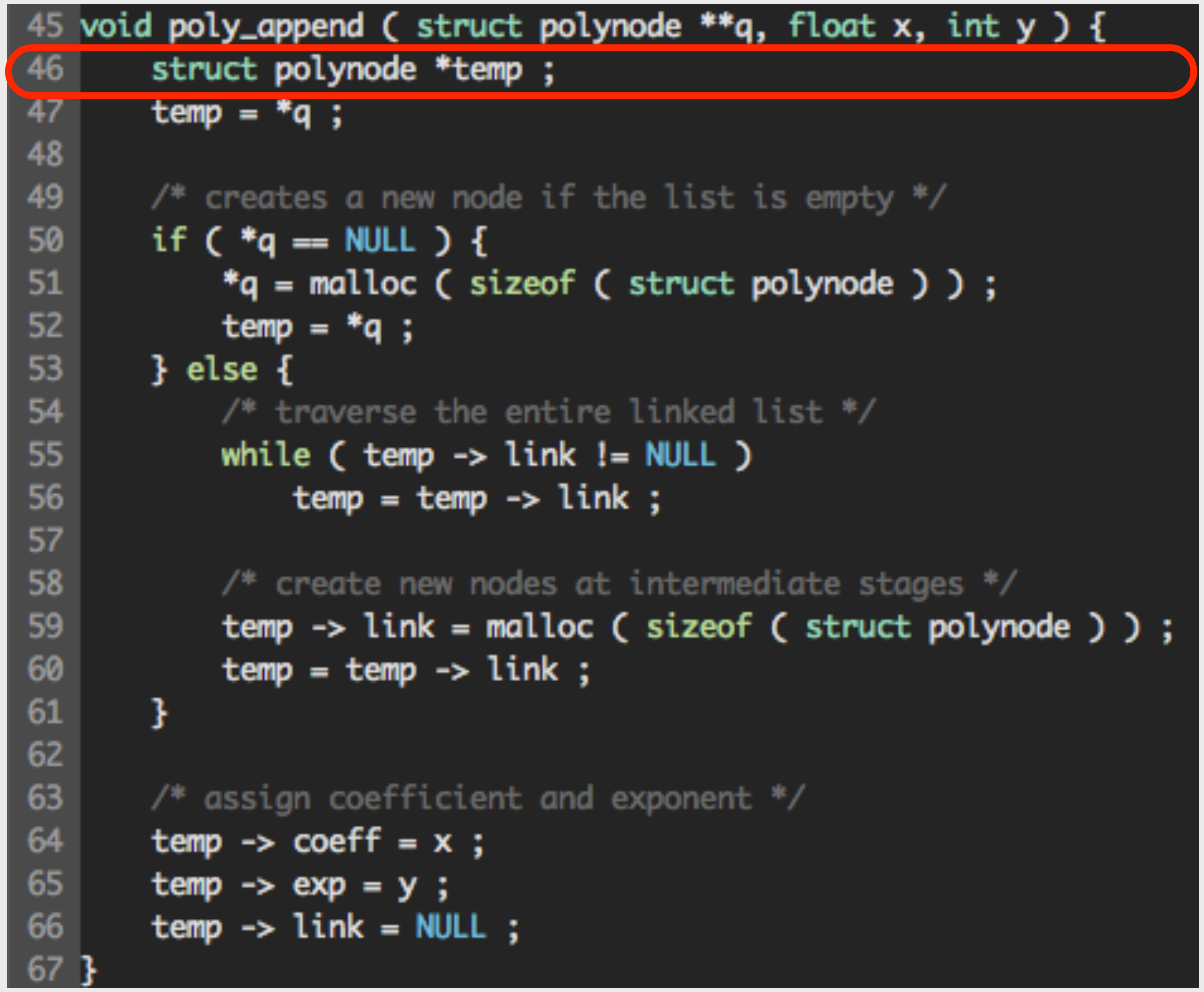


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

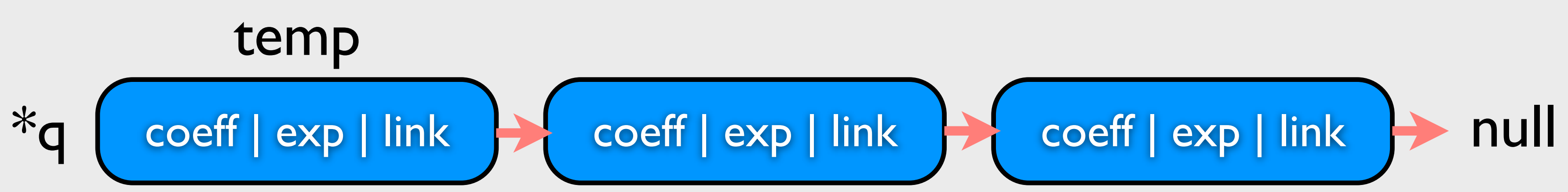


Example: poly/poly.c

- poly_append

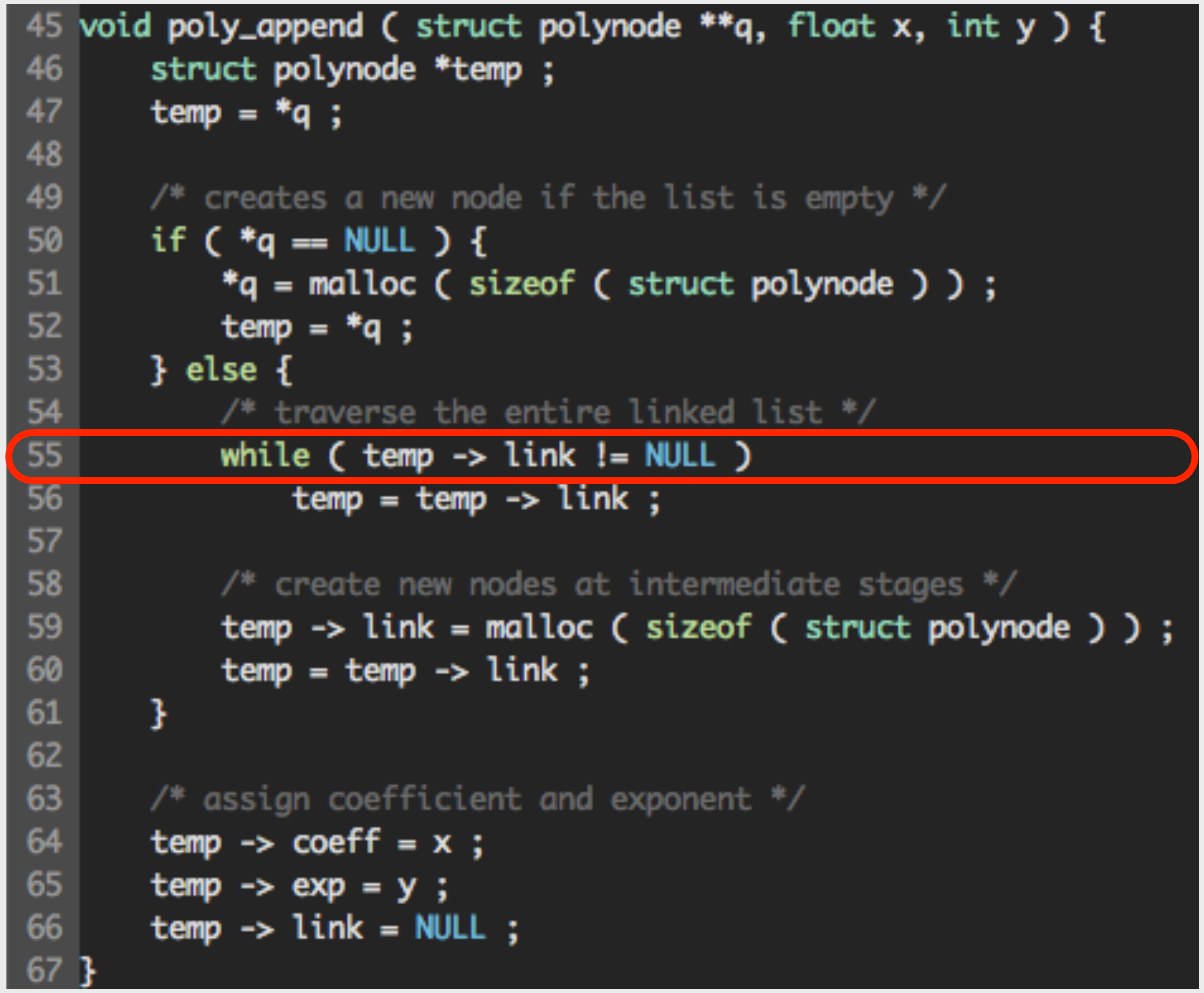


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

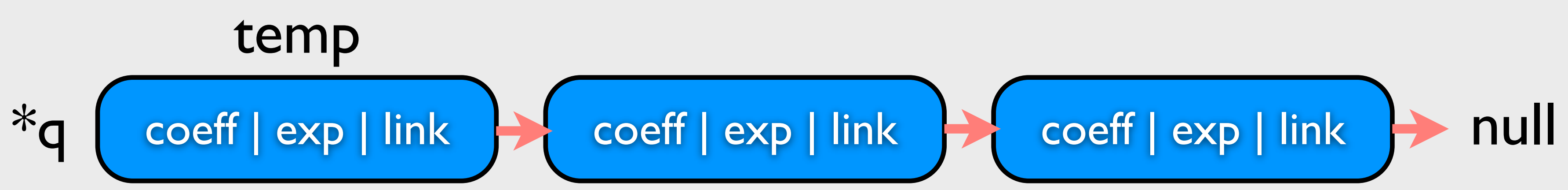


Example: poly/poly.c

- poly_append

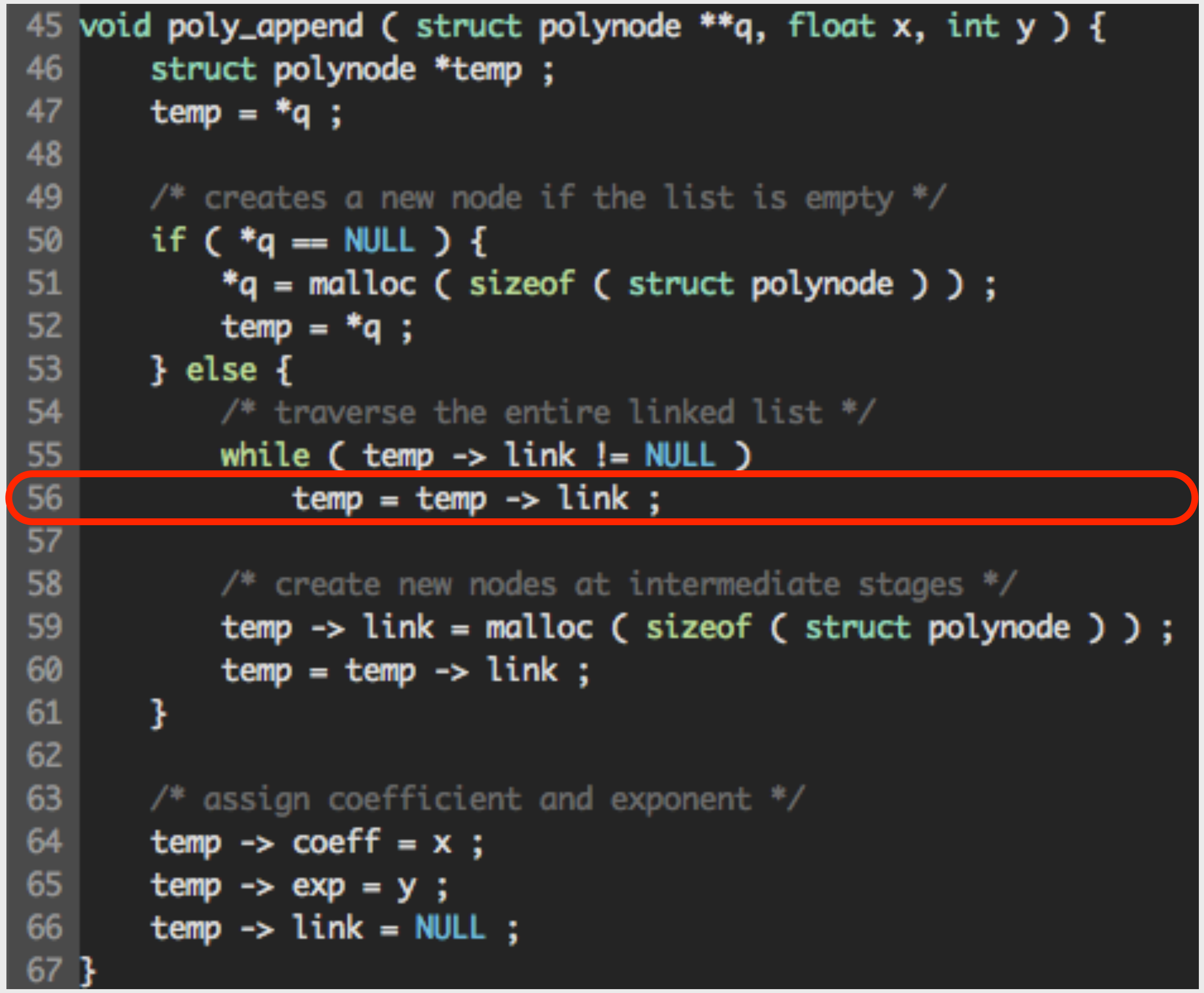


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

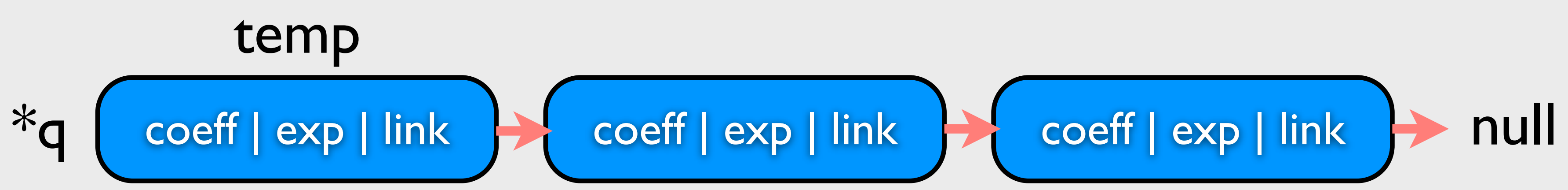


Example: poly/poly.c

- poly_append

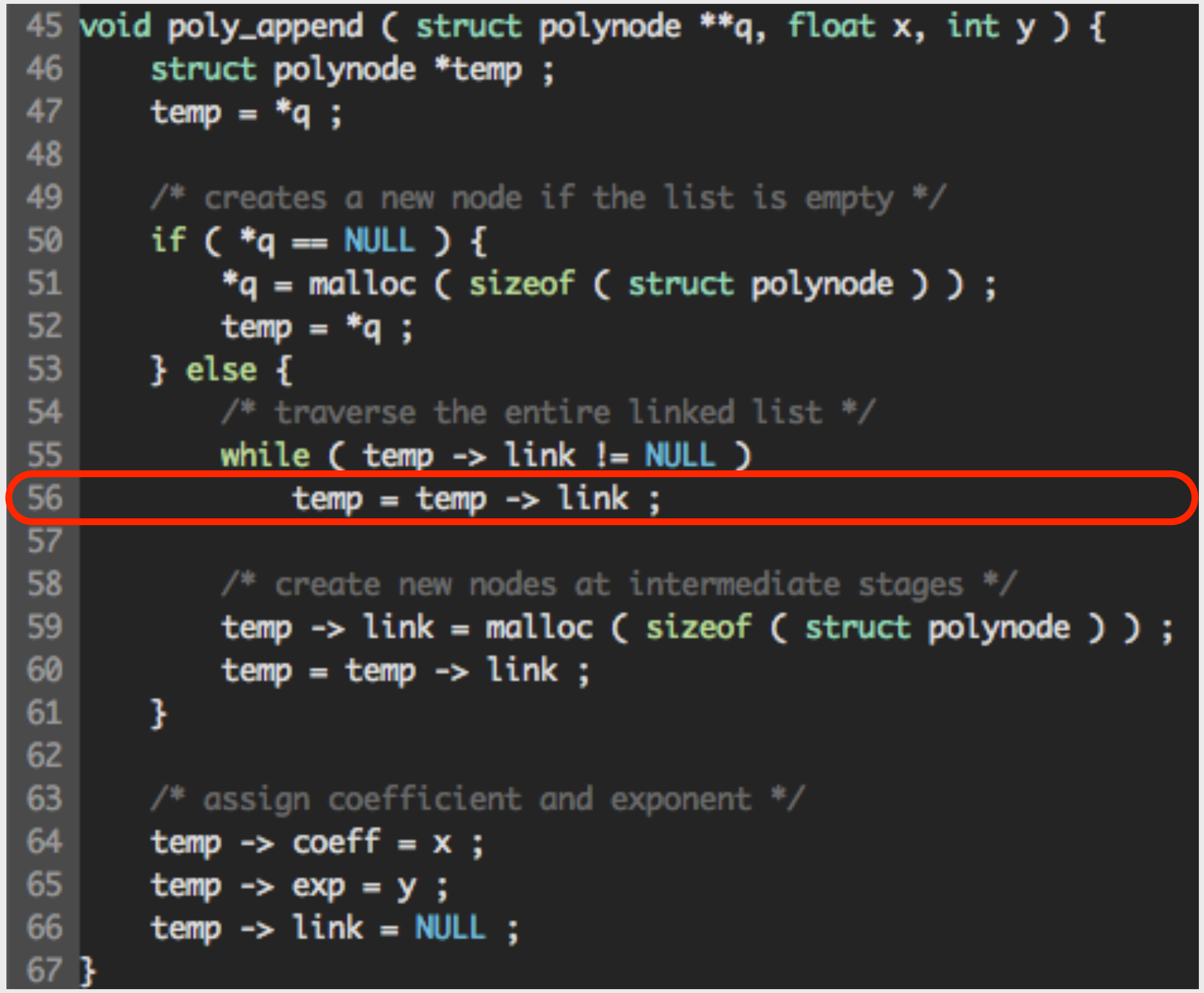


```
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

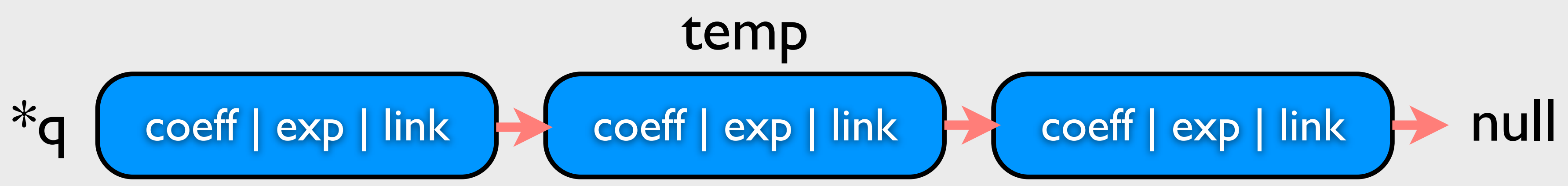


Example: poly/poly.c

- poly_append

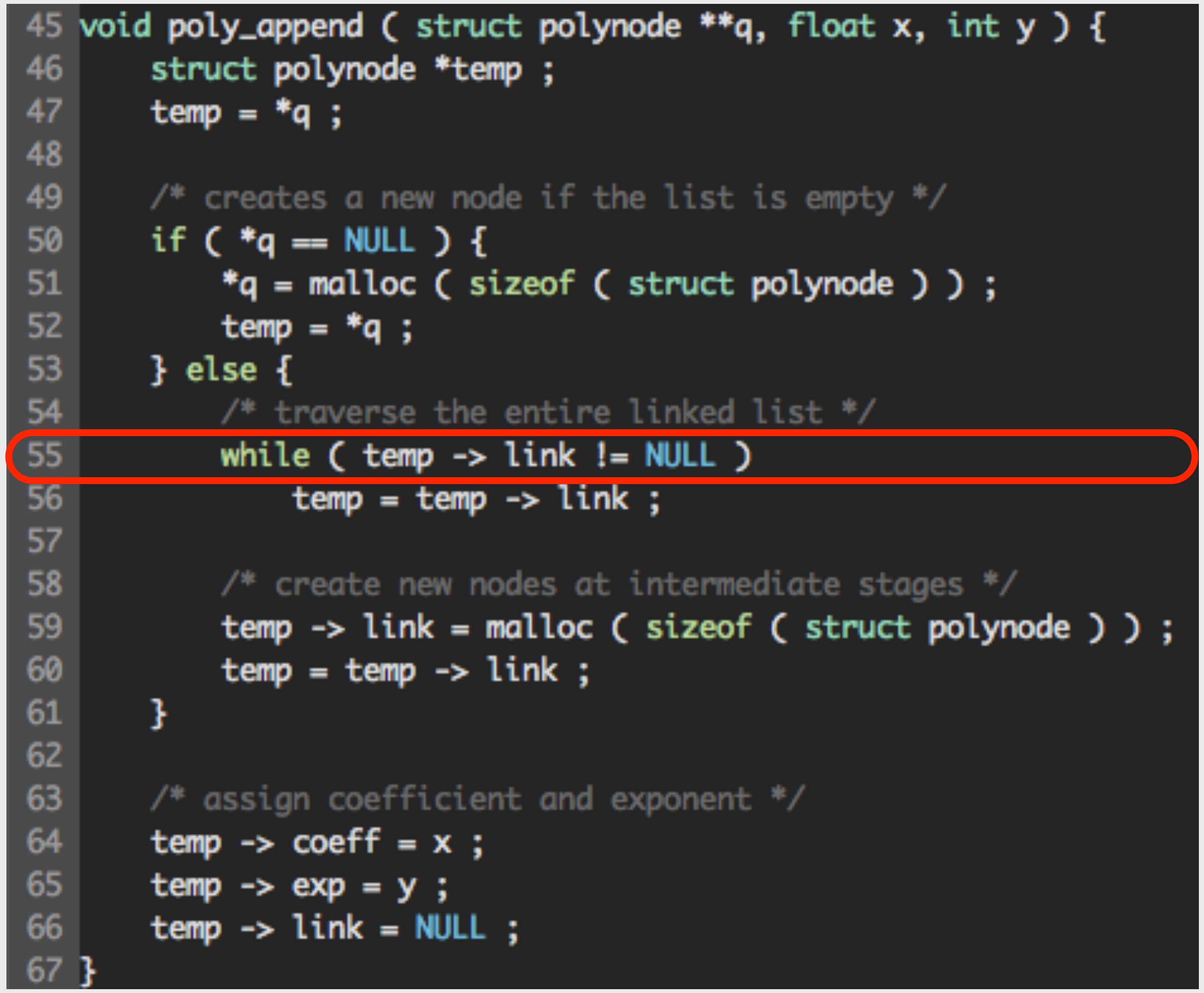


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

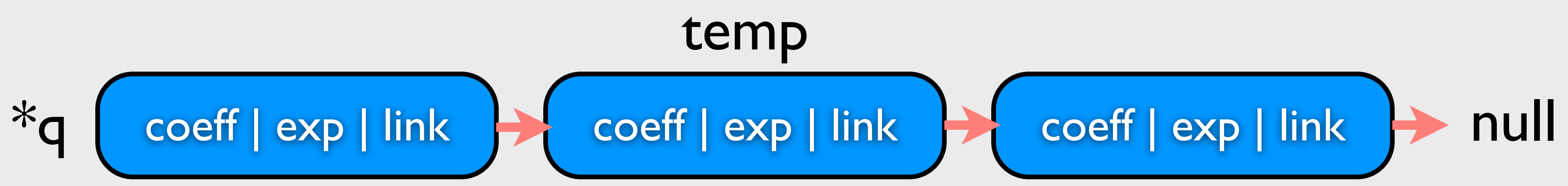


Example: poly/poly.c

- poly_append

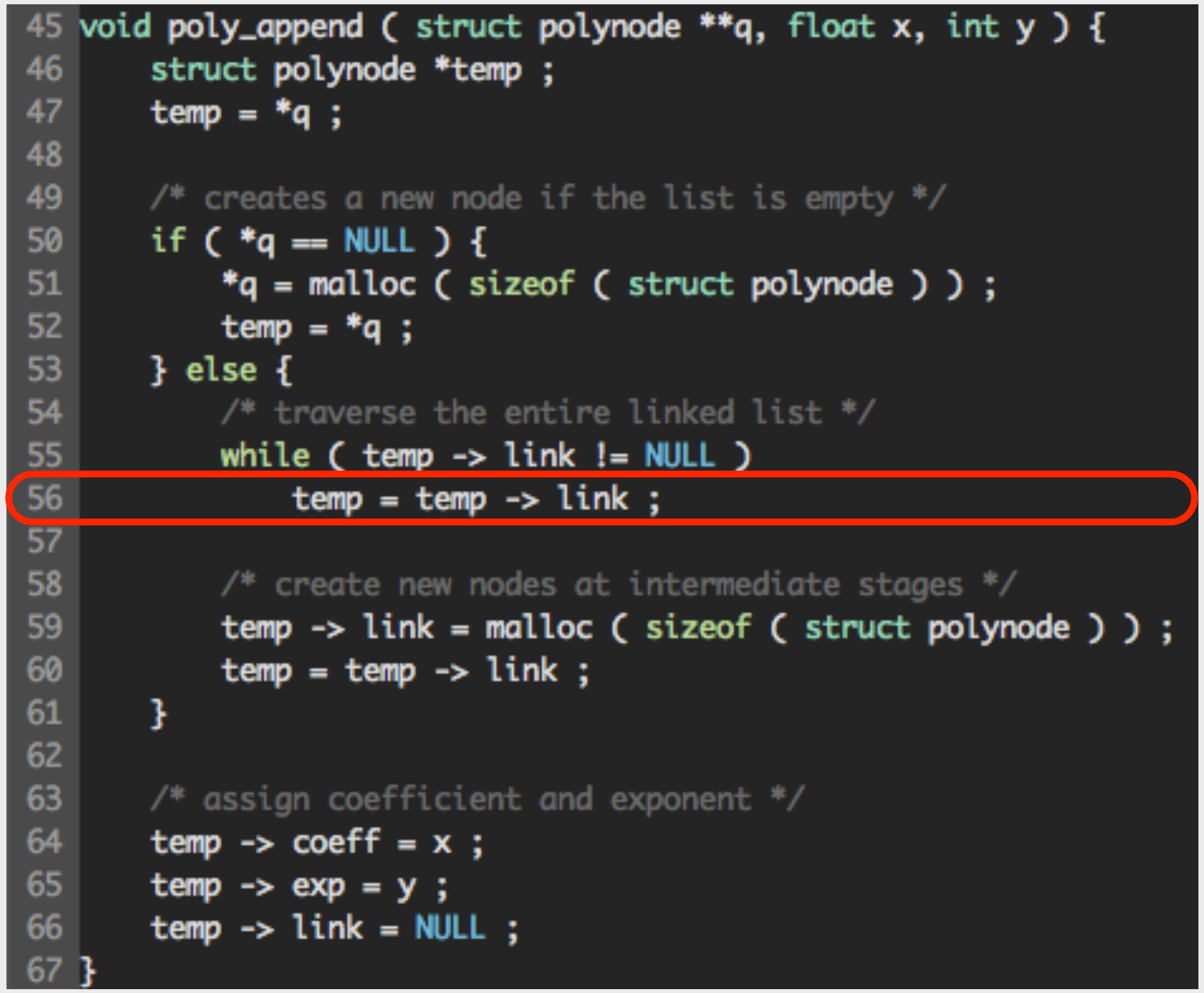


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

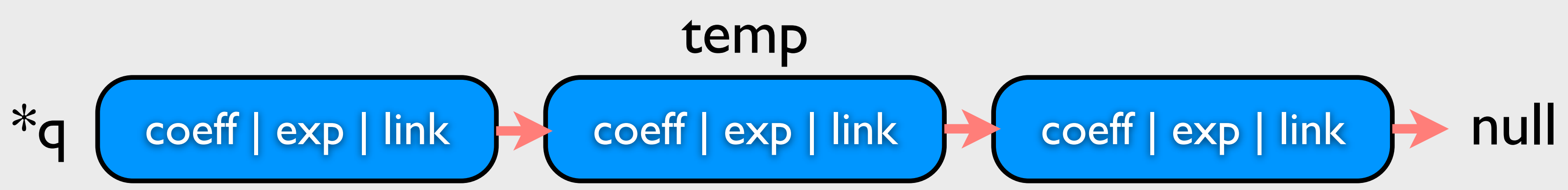


Example: poly/poly.c

- poly_append

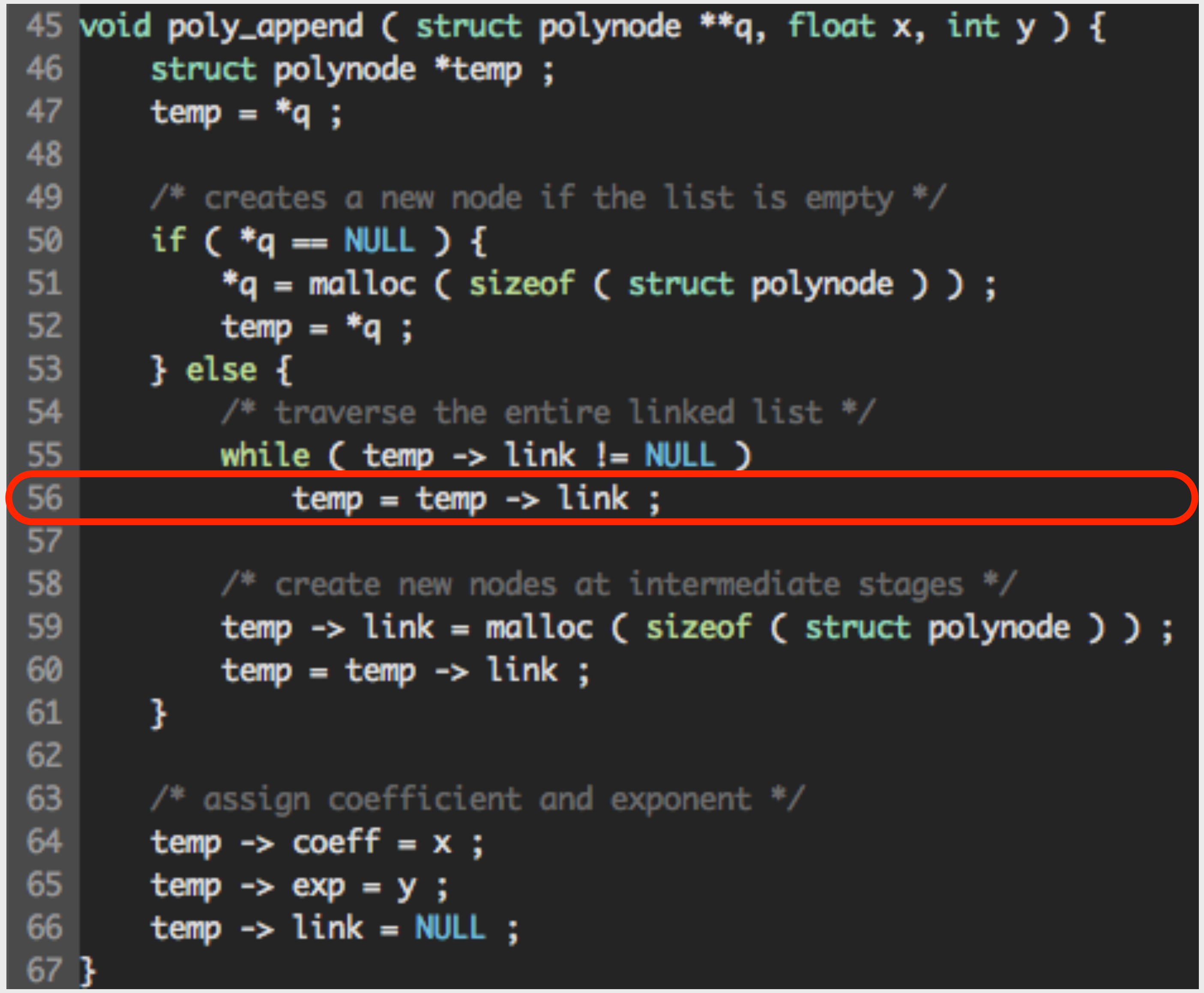


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

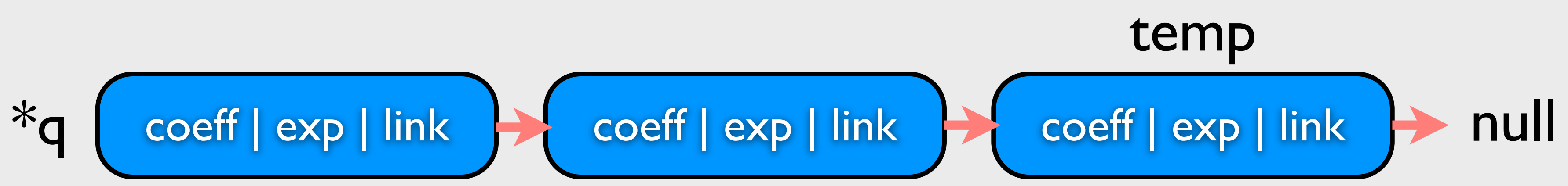


Example: poly/poly.c

- poly_append

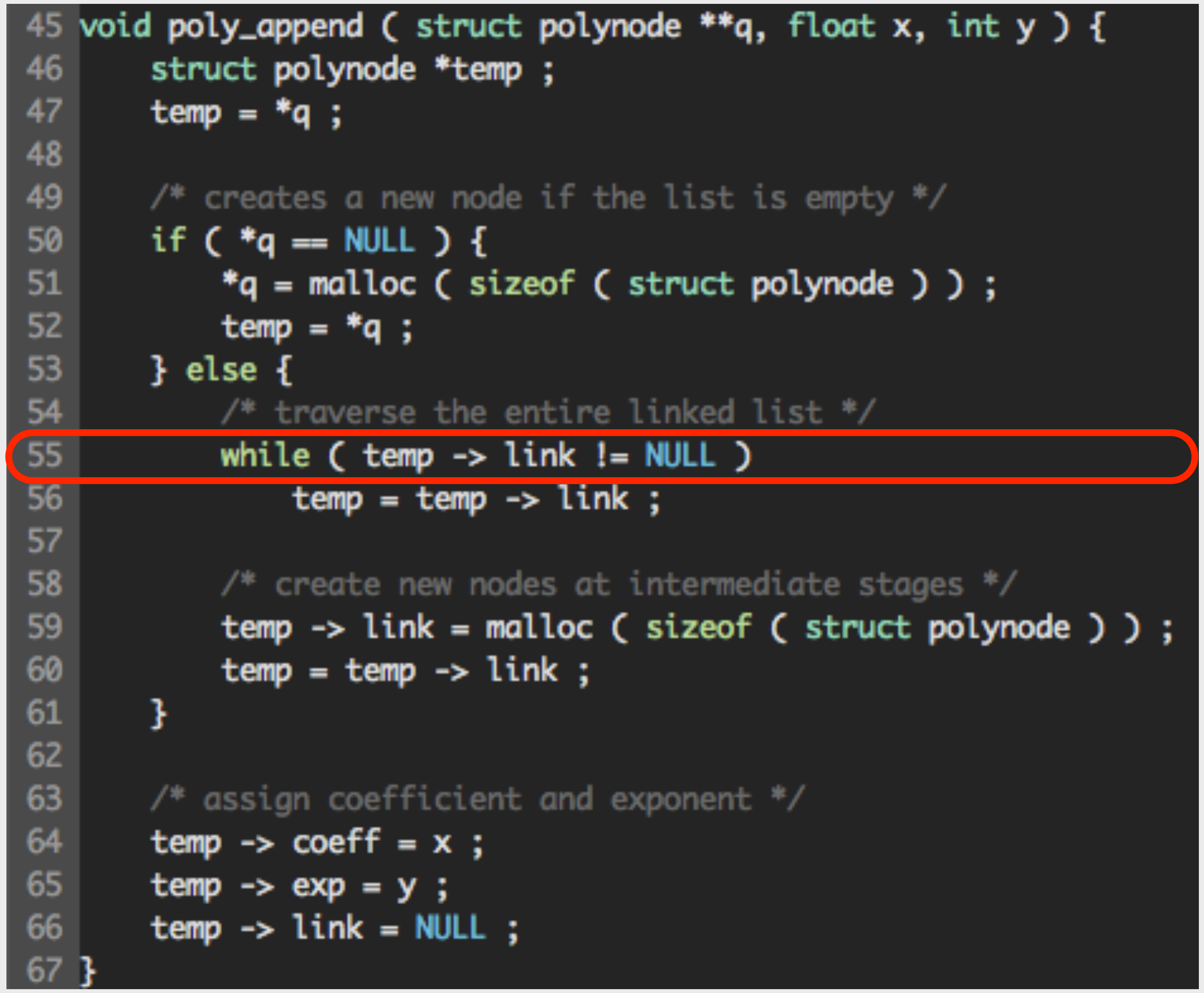


```
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

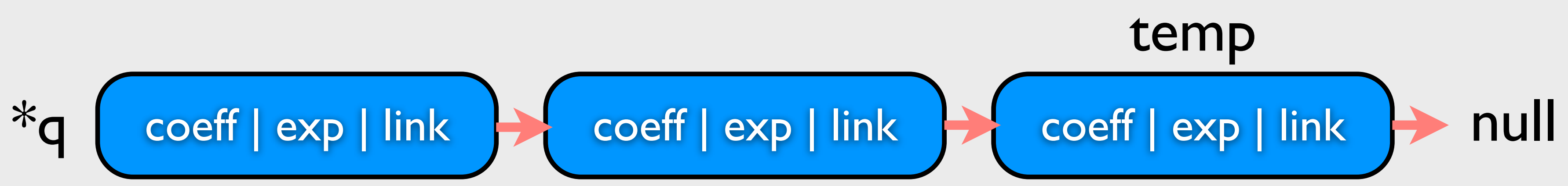


Example: poly/poly.c

- poly_append

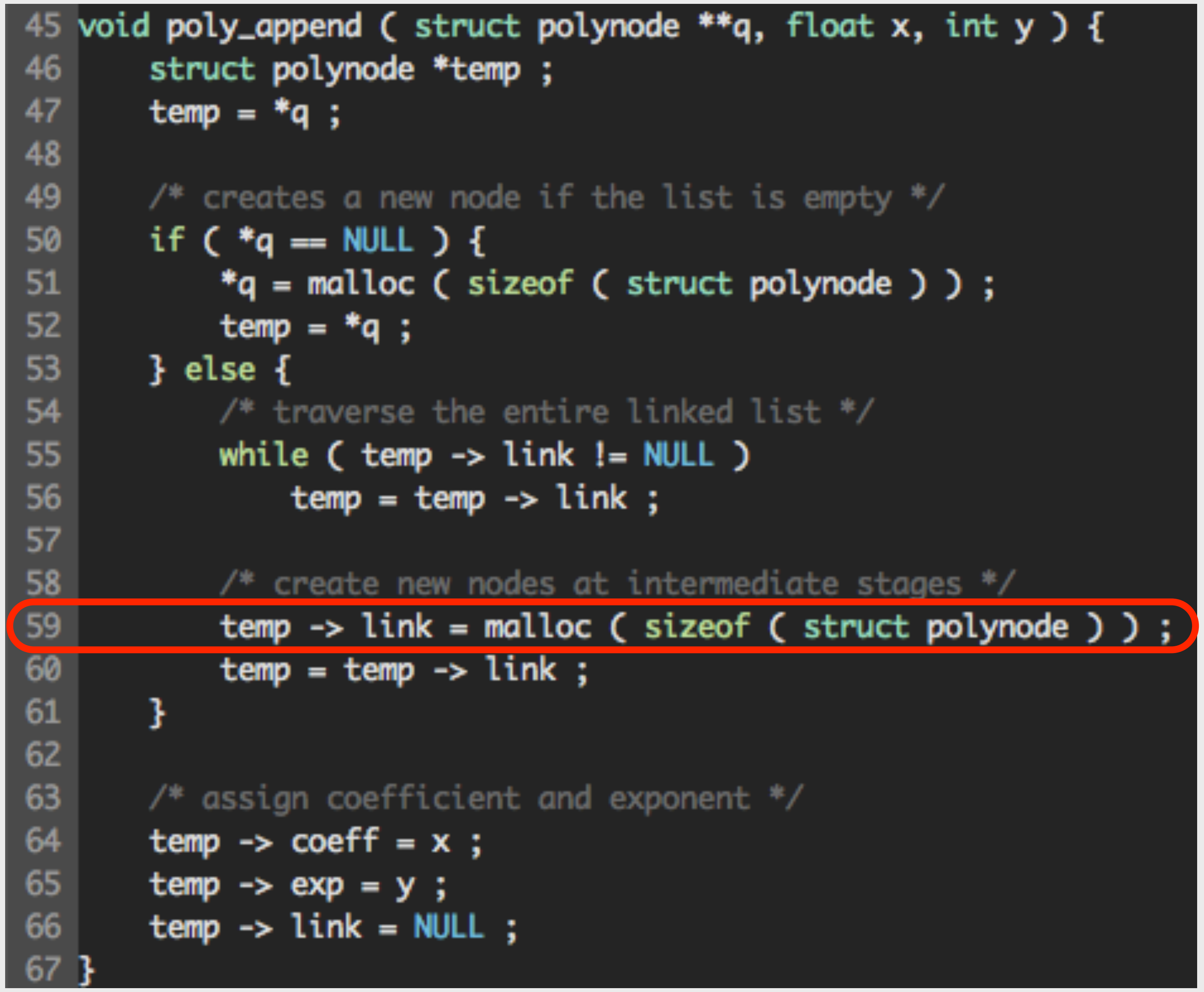


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

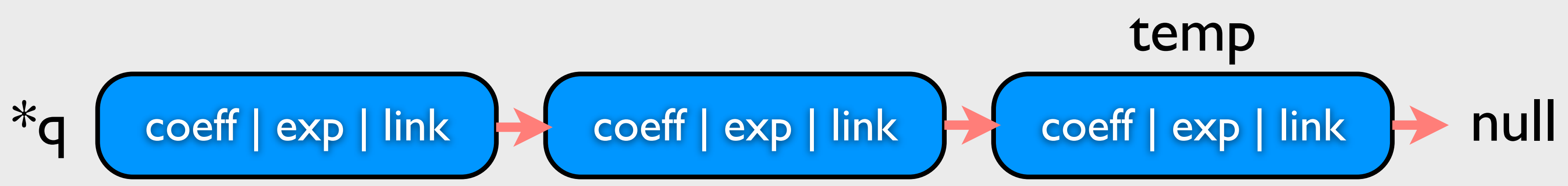


Example: poly/poly.c

- poly_append

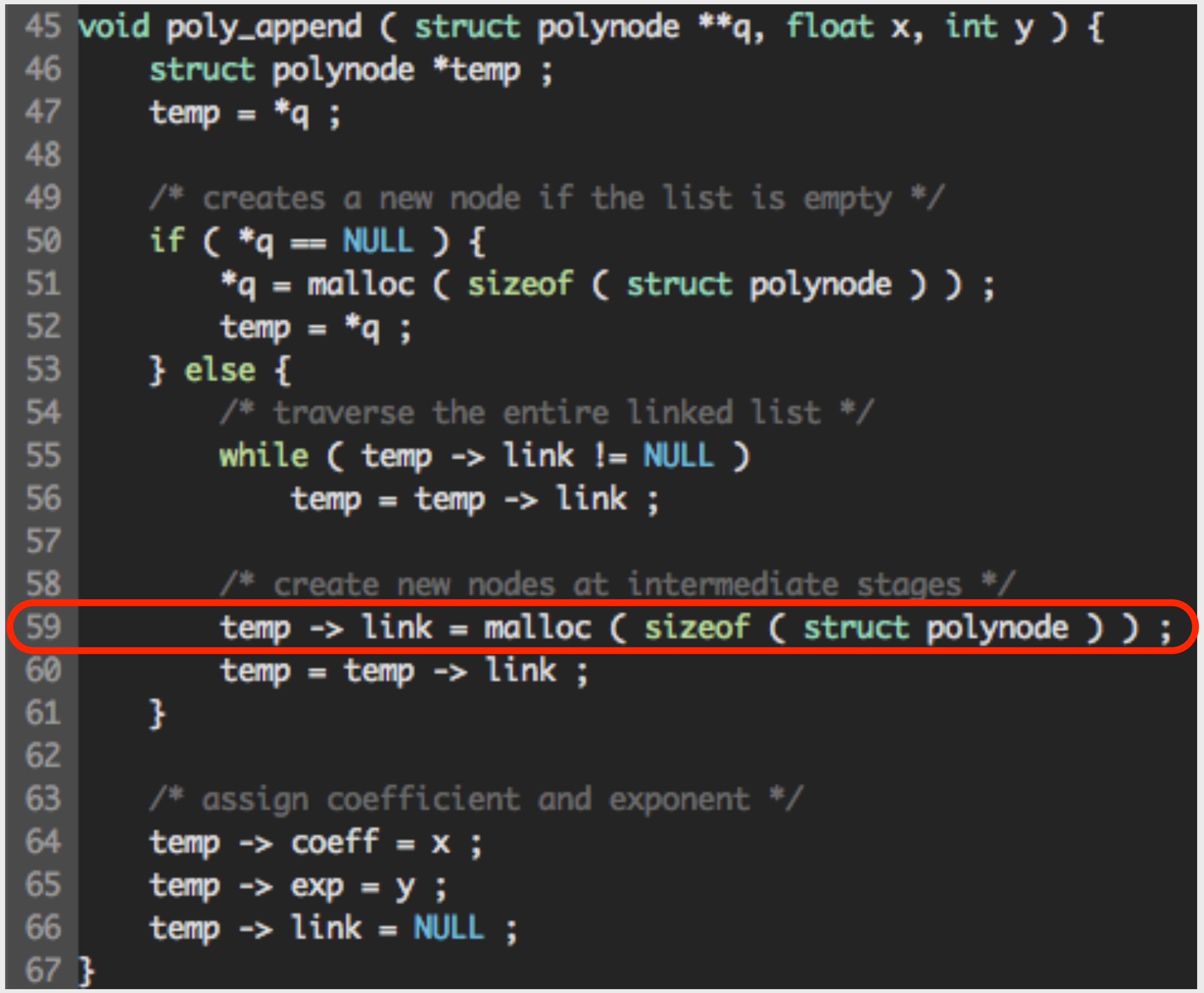


```
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

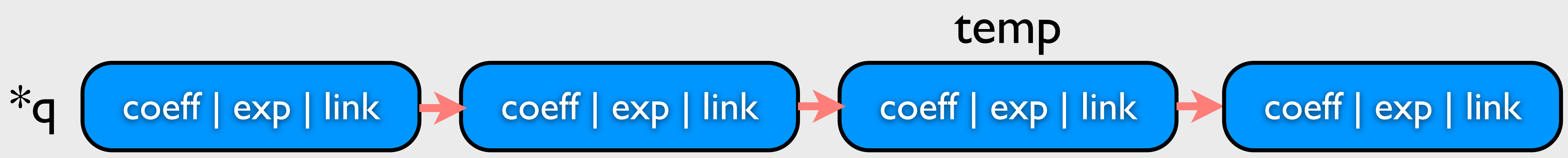


Example: poly/poly.c

- poly_append

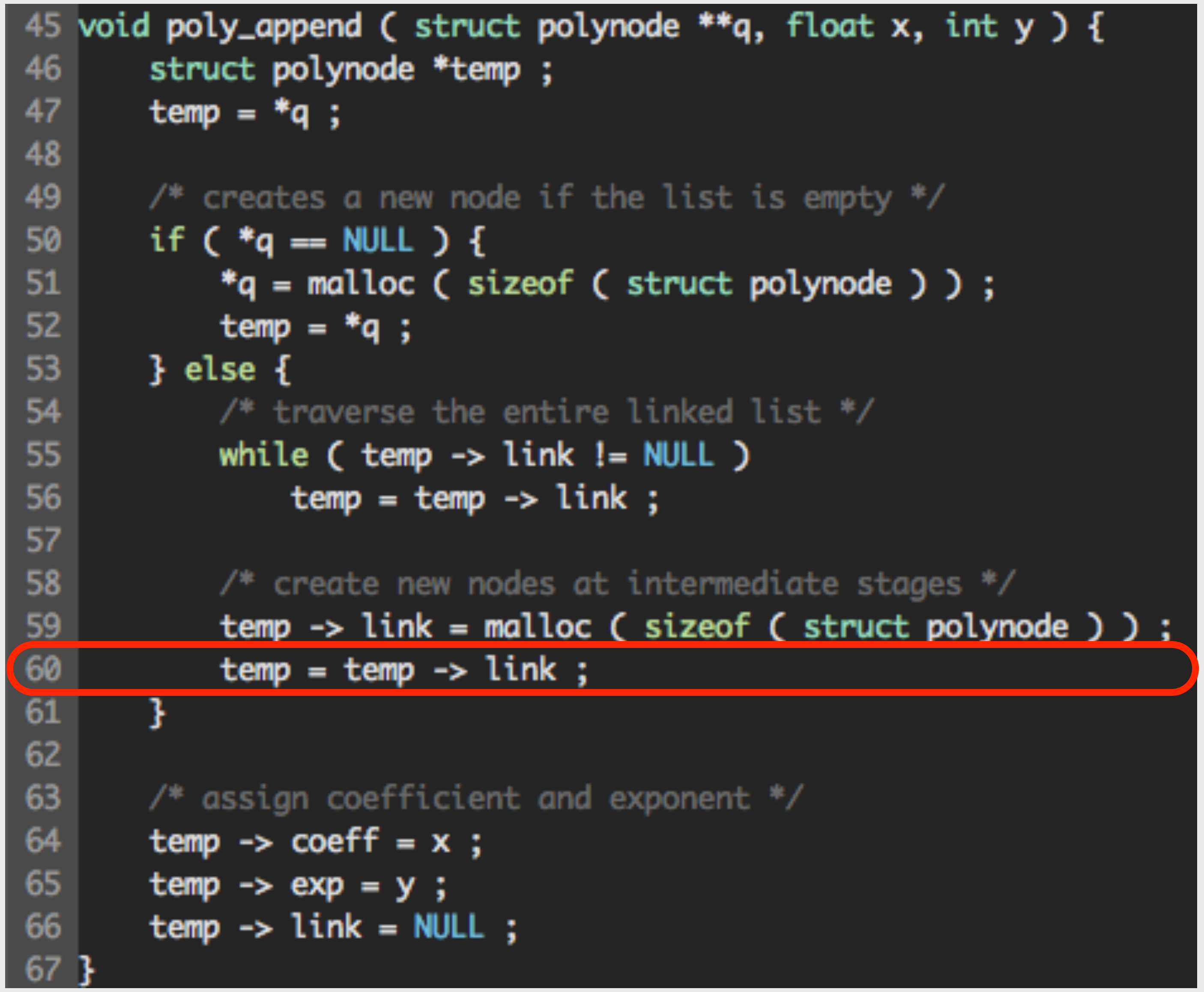


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

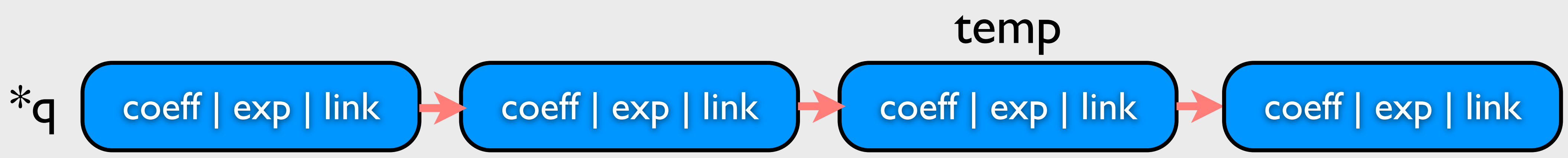


Example: poly/poly.c

- poly_append

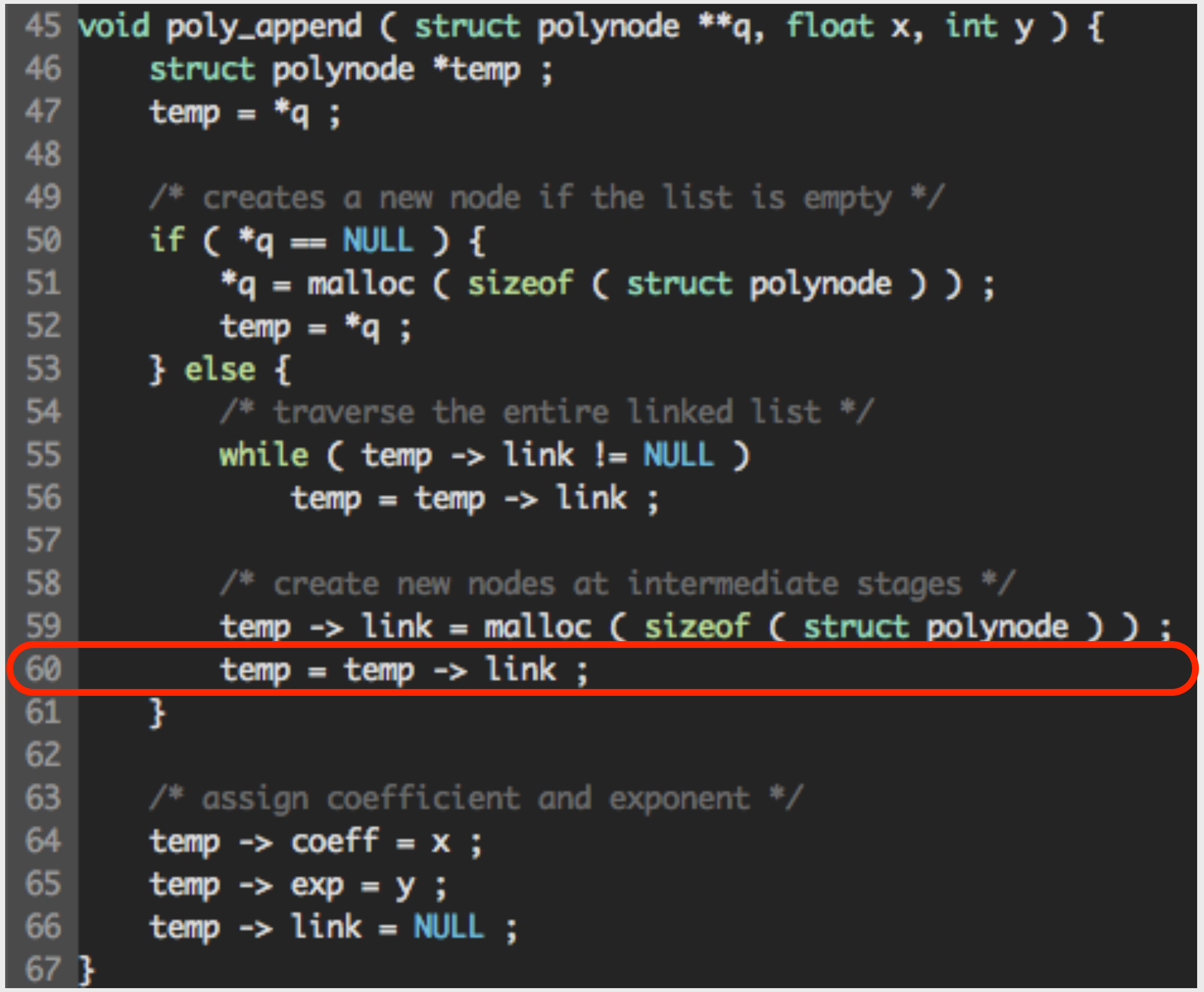


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

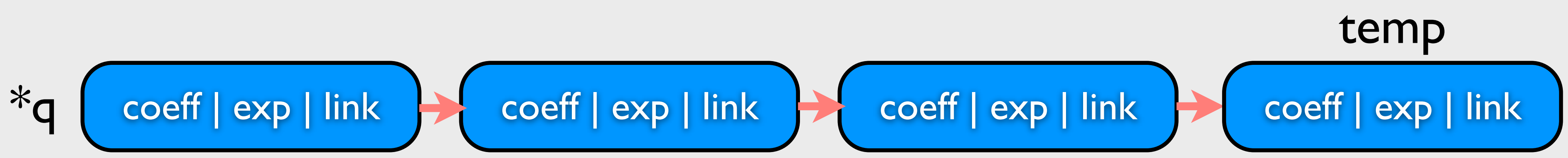


Example: poly/poly.c

- poly_append

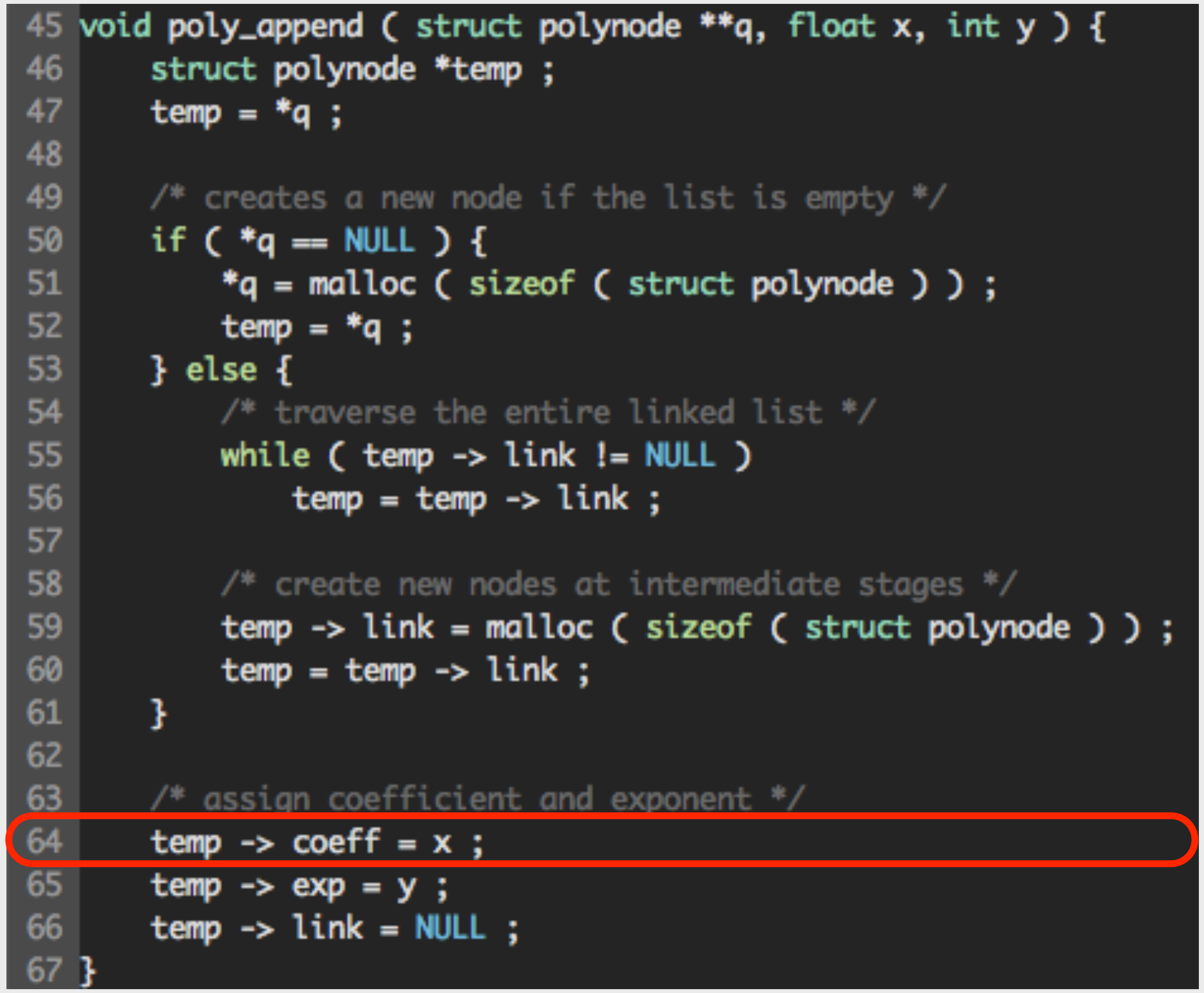


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

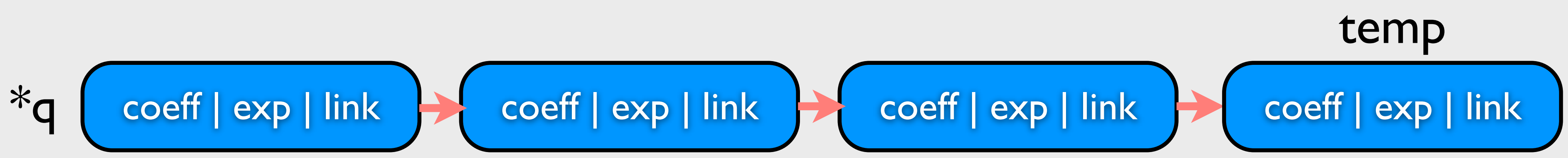


Example: poly/poly.c

- poly_append

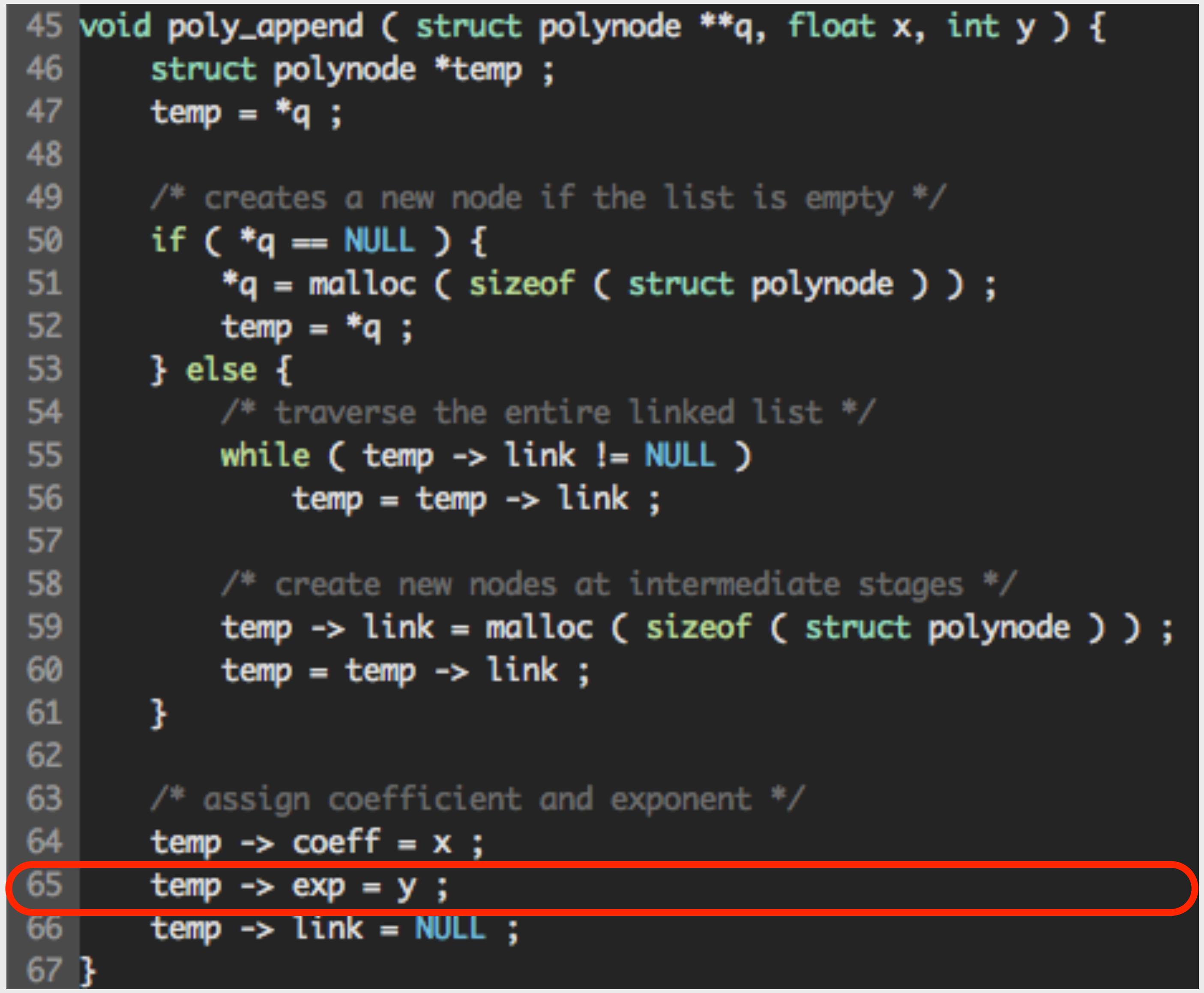


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

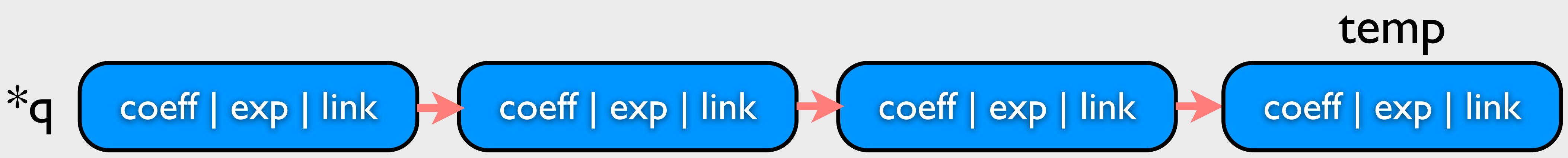


Example: poly/poly.c

- poly_append

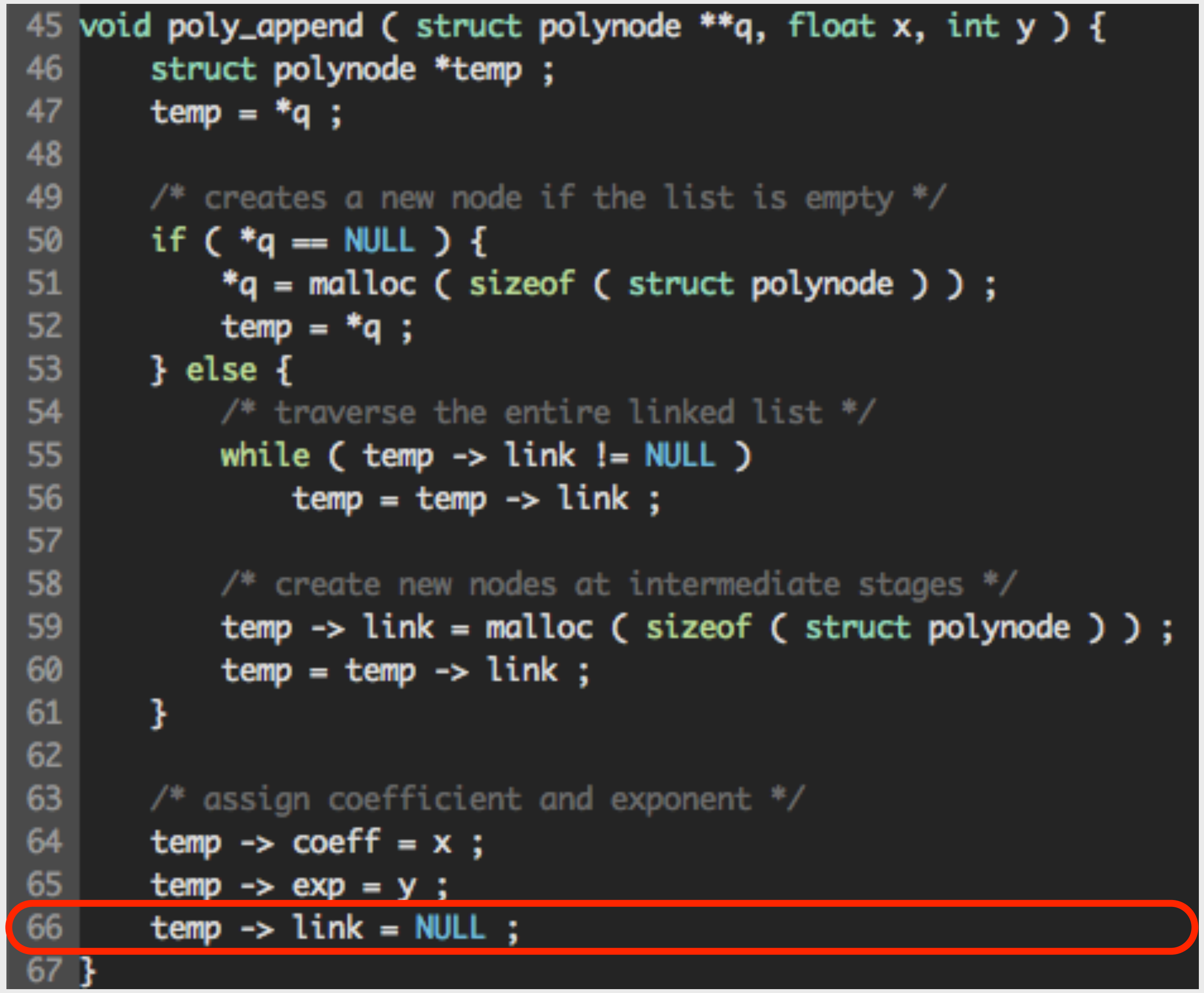


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

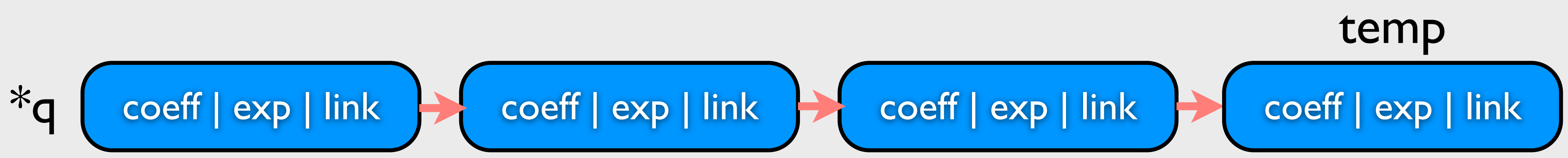


Example: poly/poly.c

- poly_append

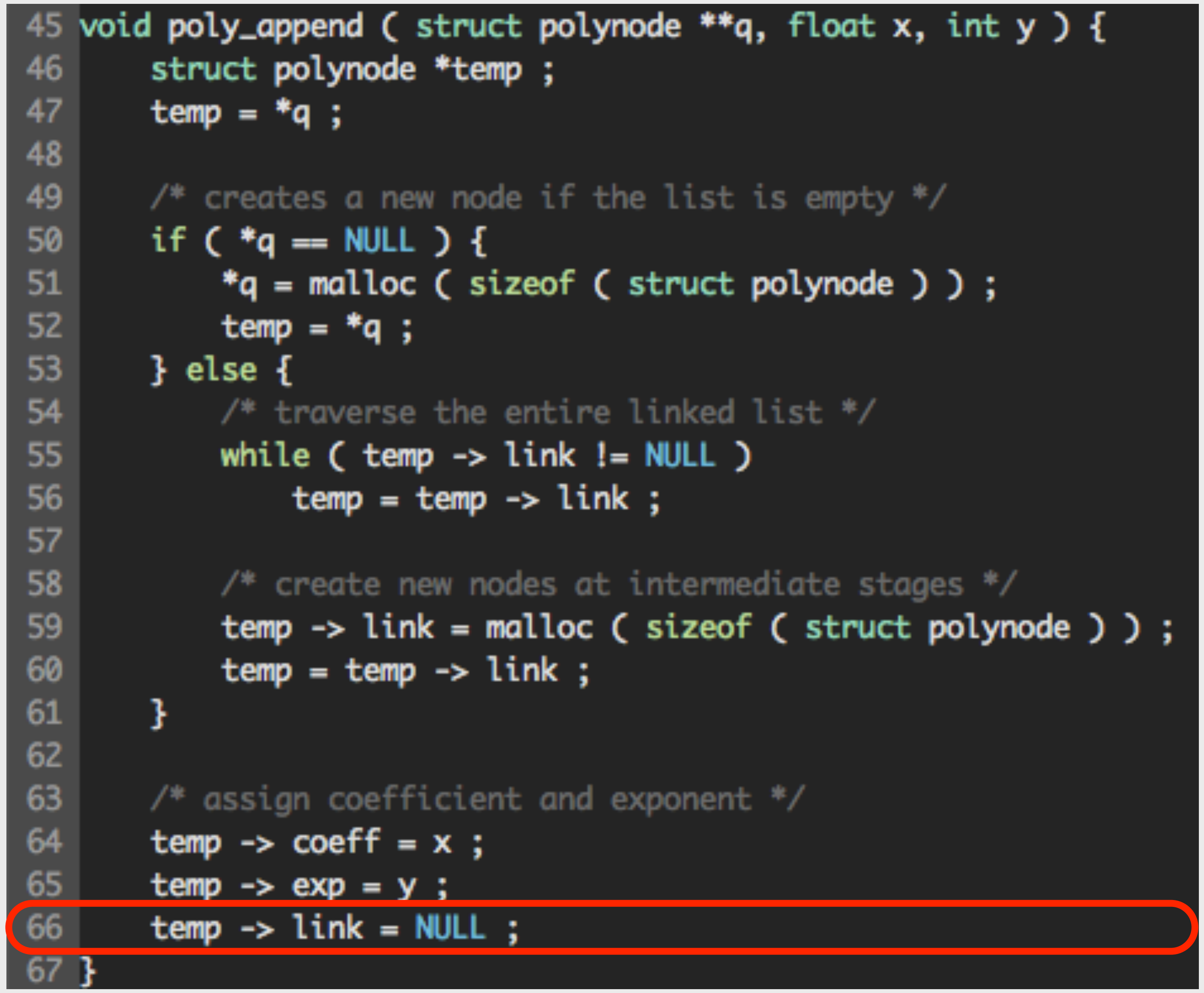


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

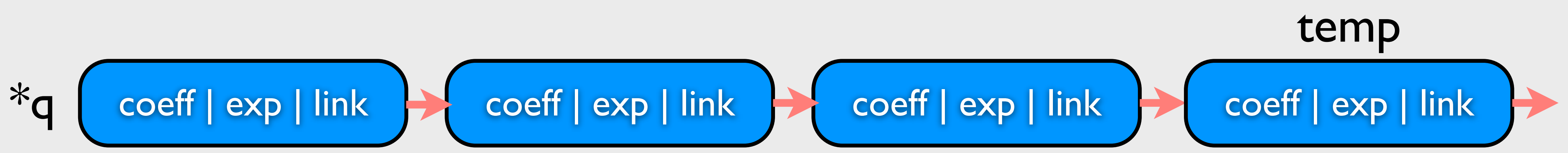


Example: poly/poly.c

- poly_append

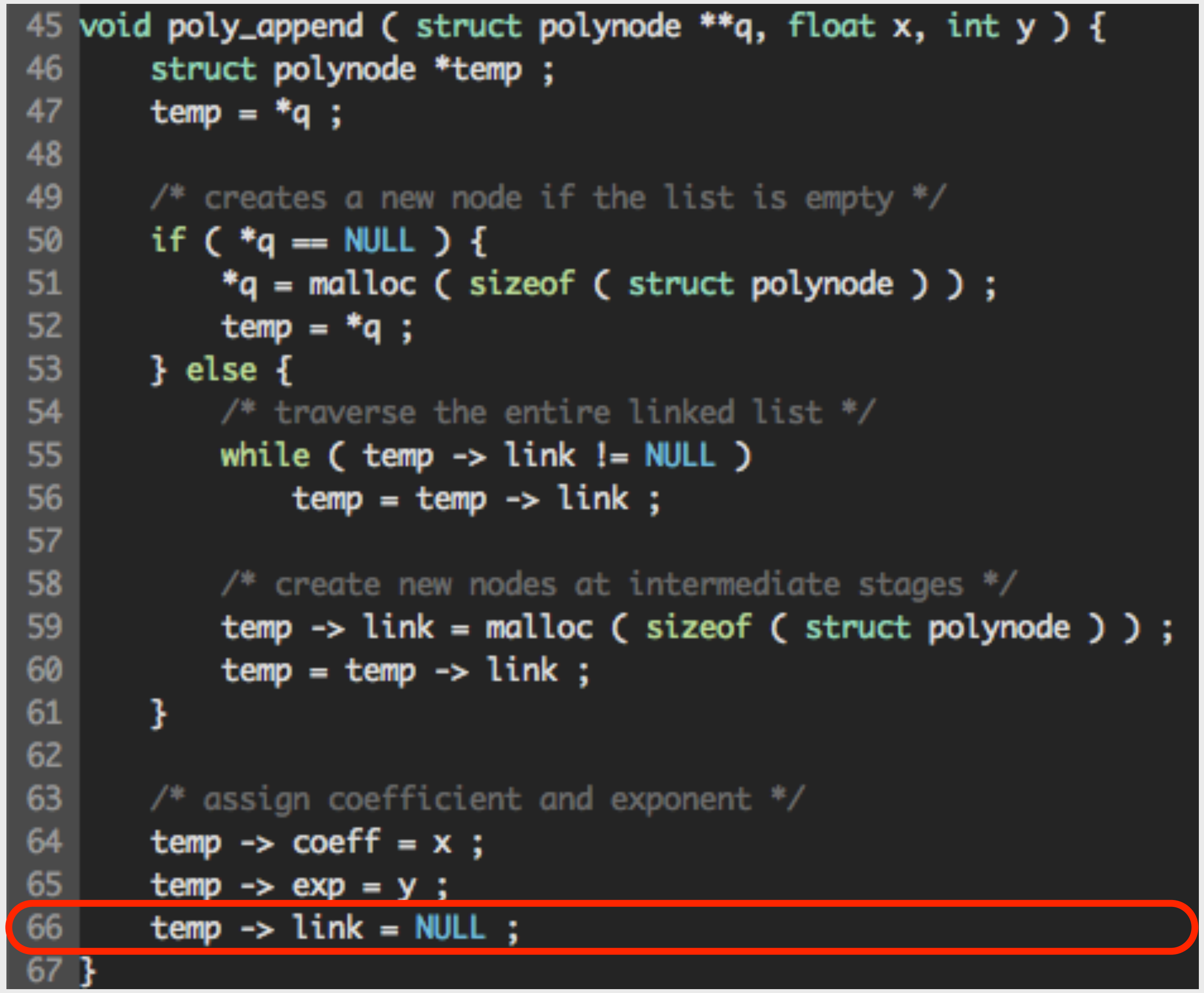


```
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

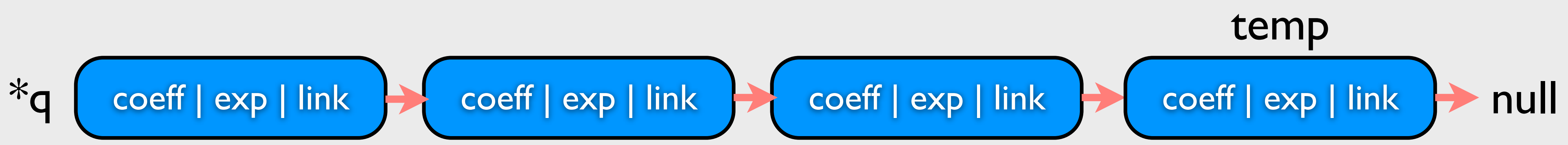


Example: poly/poly.c

- poly_append

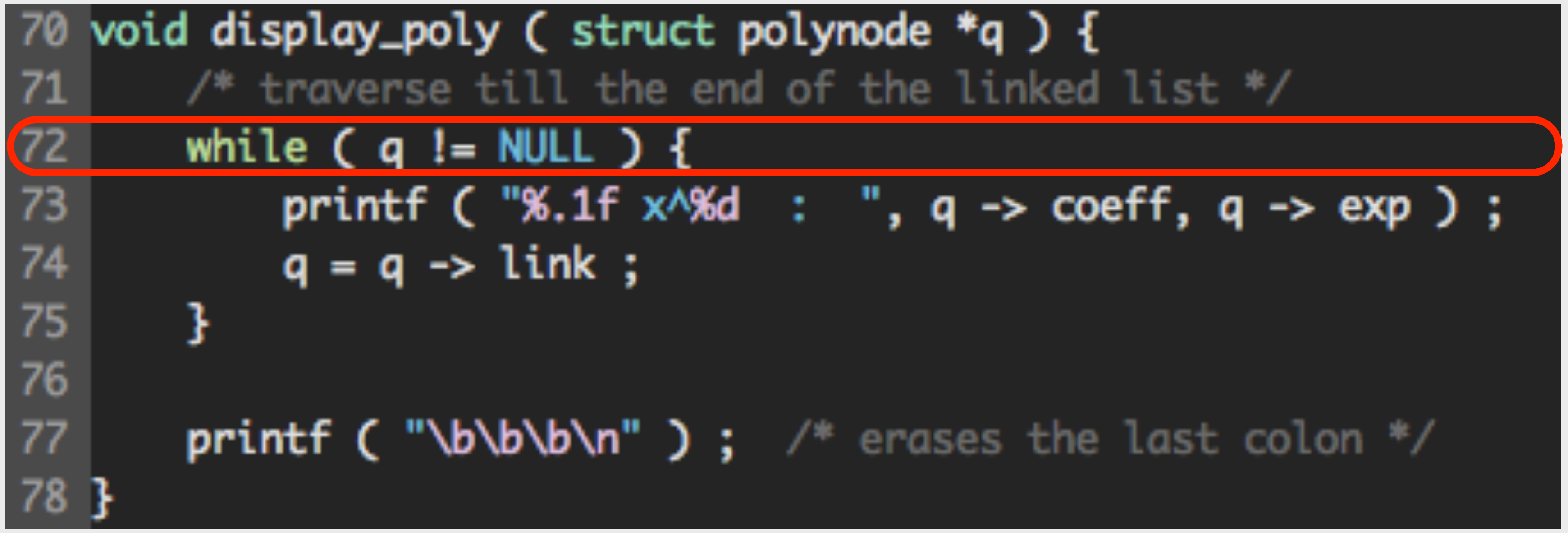


```c
void poly_append ( struct polynode **q, float x, int y ) {
    struct polynode *temp ;
    temp = *q ;

    /* creates a new node if the list is empty */
    if ( *q == NULL ) {
        *q = malloc ( sizeof ( struct polynode ) ) ;
        temp = *q ;
    } else {
        /* traverse the entire linked list */
        while ( temp -> link != NULL )
            temp = temp -> link ;

        /* create new nodes at intermediate stages */
        temp -> link = malloc ( sizeof ( struct polynode ) ) ;
        temp = temp -> link ;
    }

    /* assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> link = NULL ;
}
```

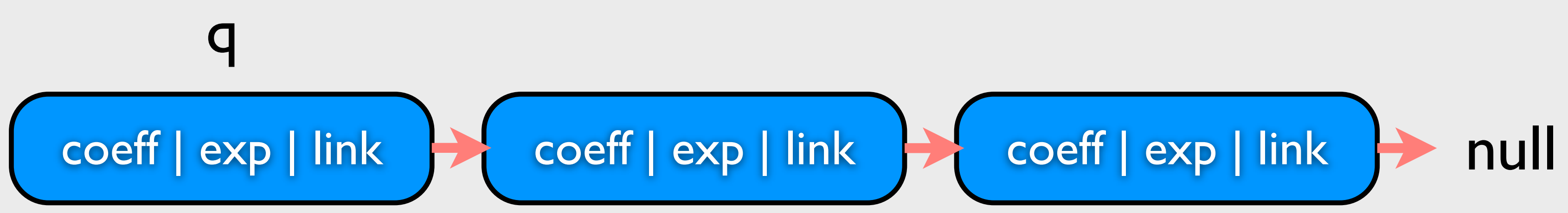


Example: poly/poly.c

- display_poly

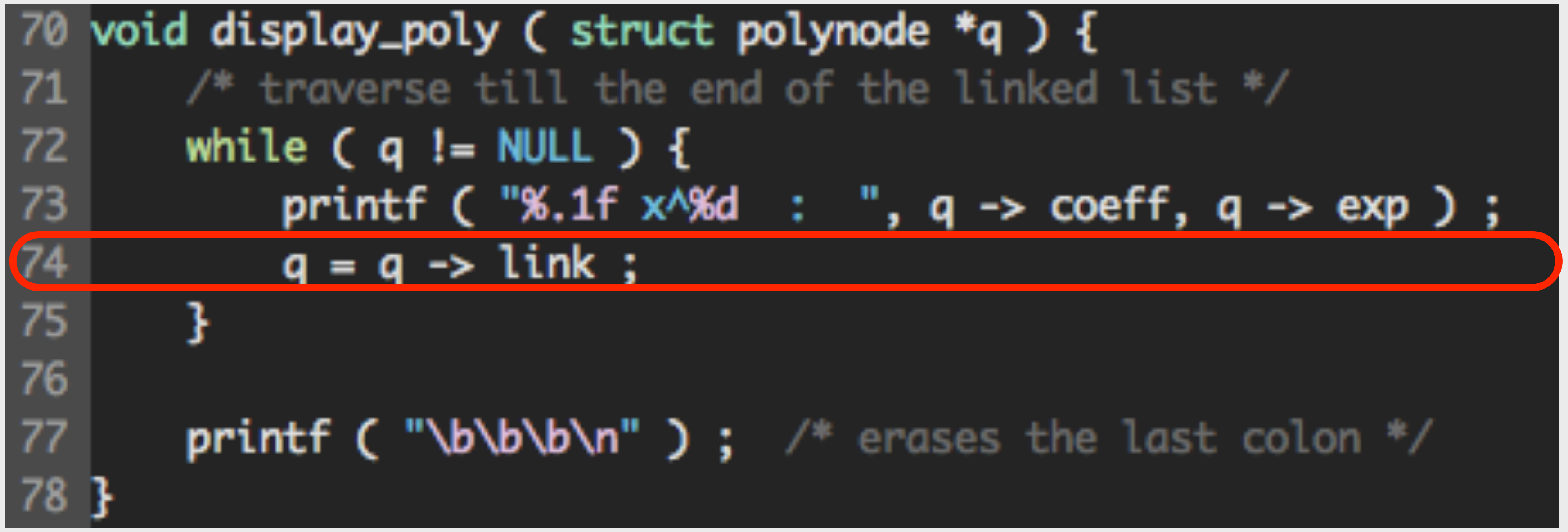


```c
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

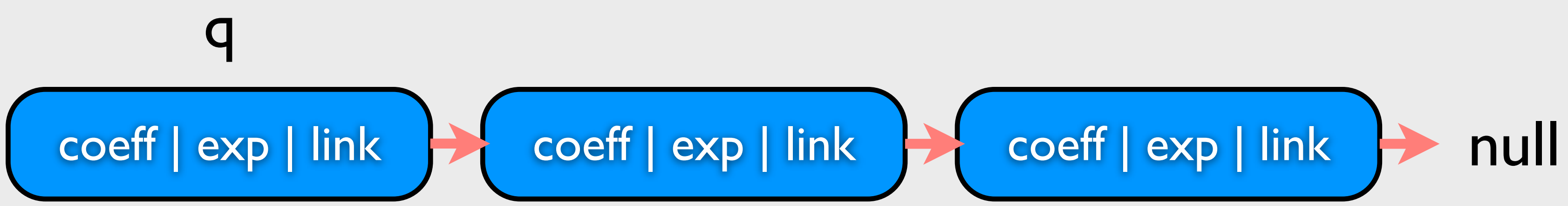


Example: poly/poly.c

- display_poly

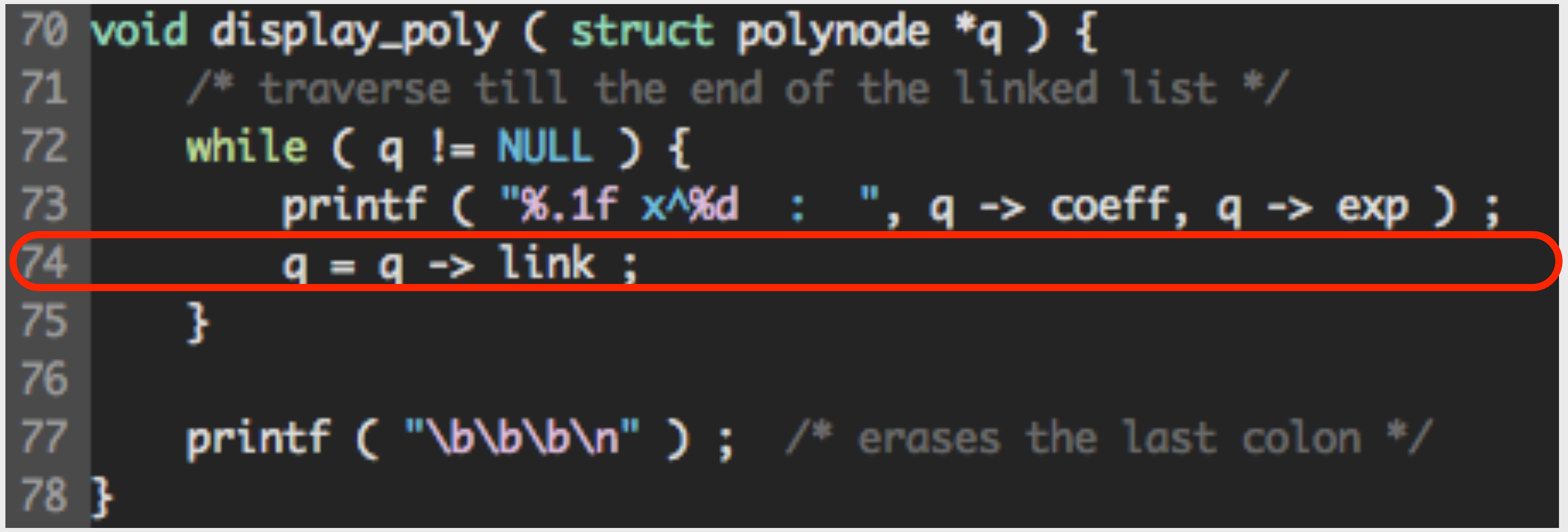


```
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

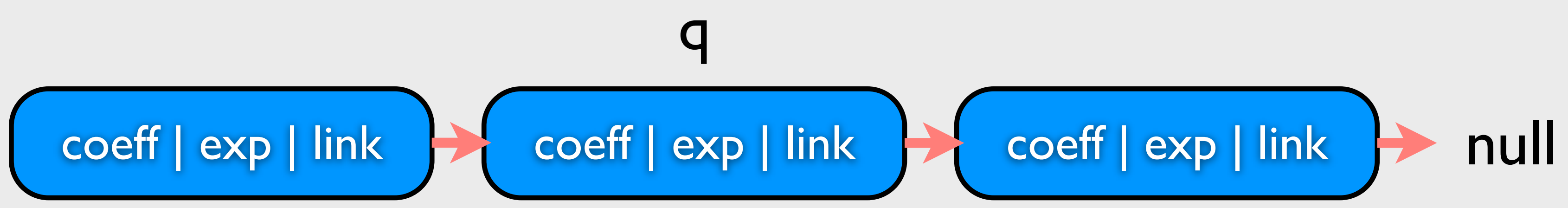


Example: poly/poly.c

- display_poly

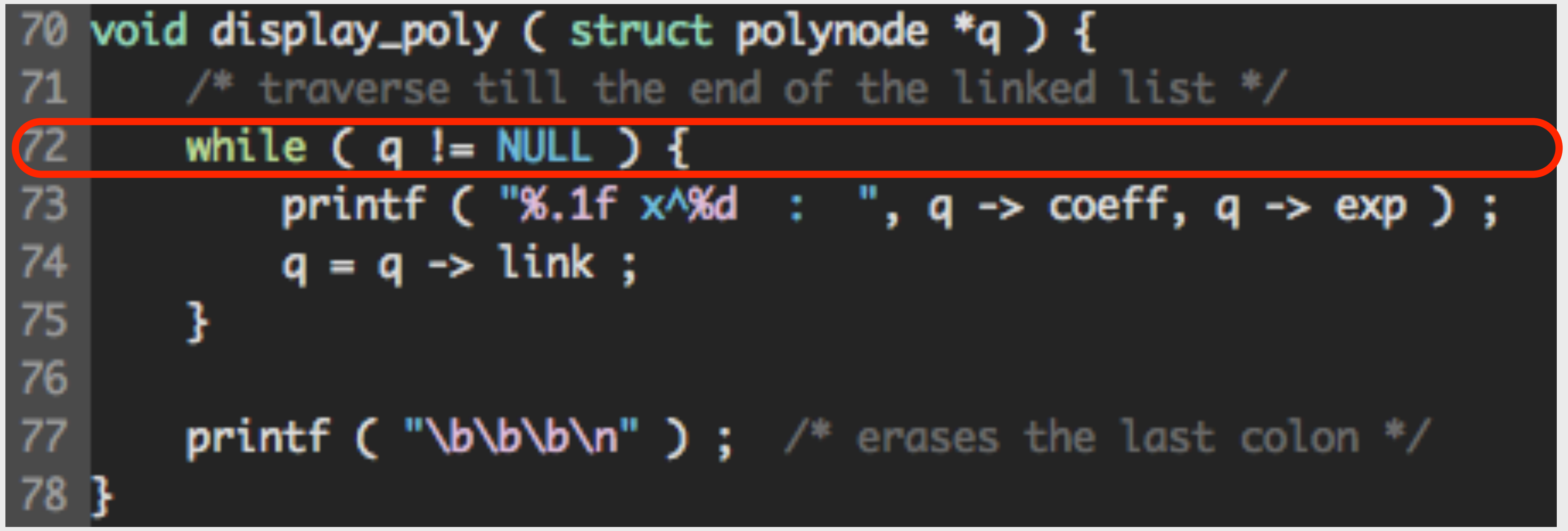


```
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

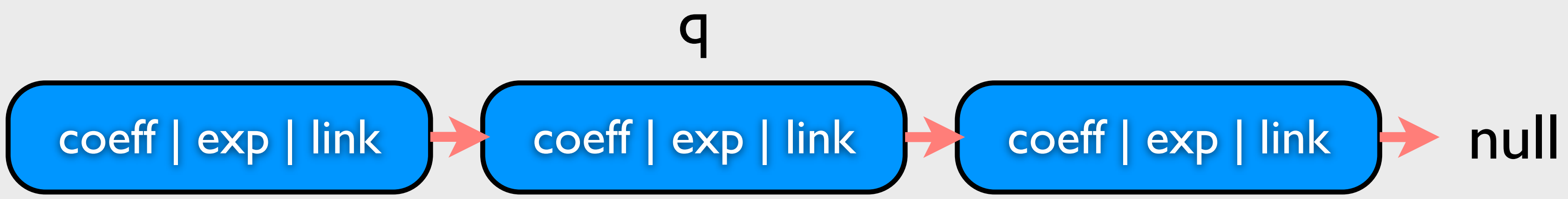


Example: poly/poly.c

- display_poly

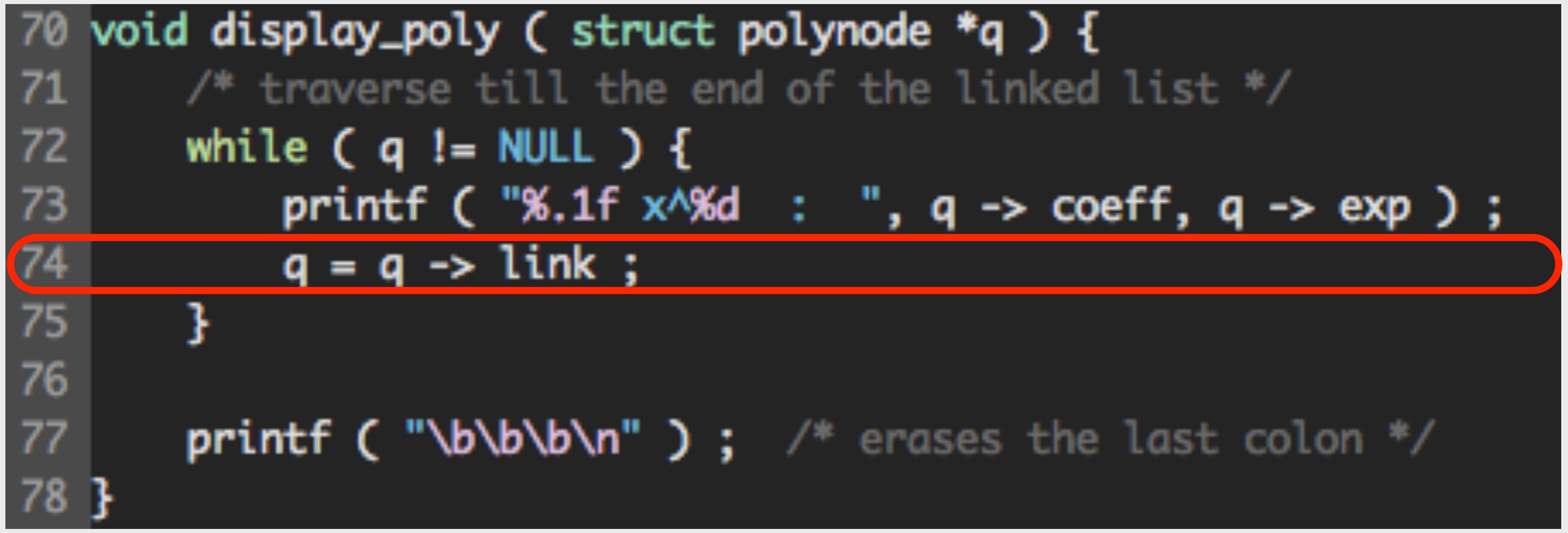


```c
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

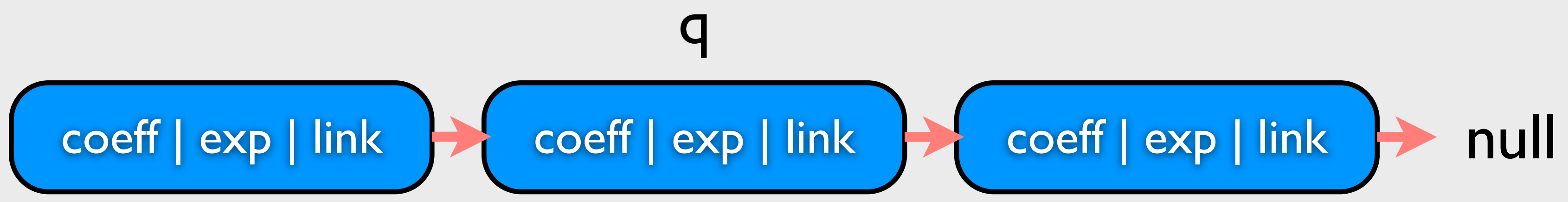


Example: poly/poly.c

- display_poly

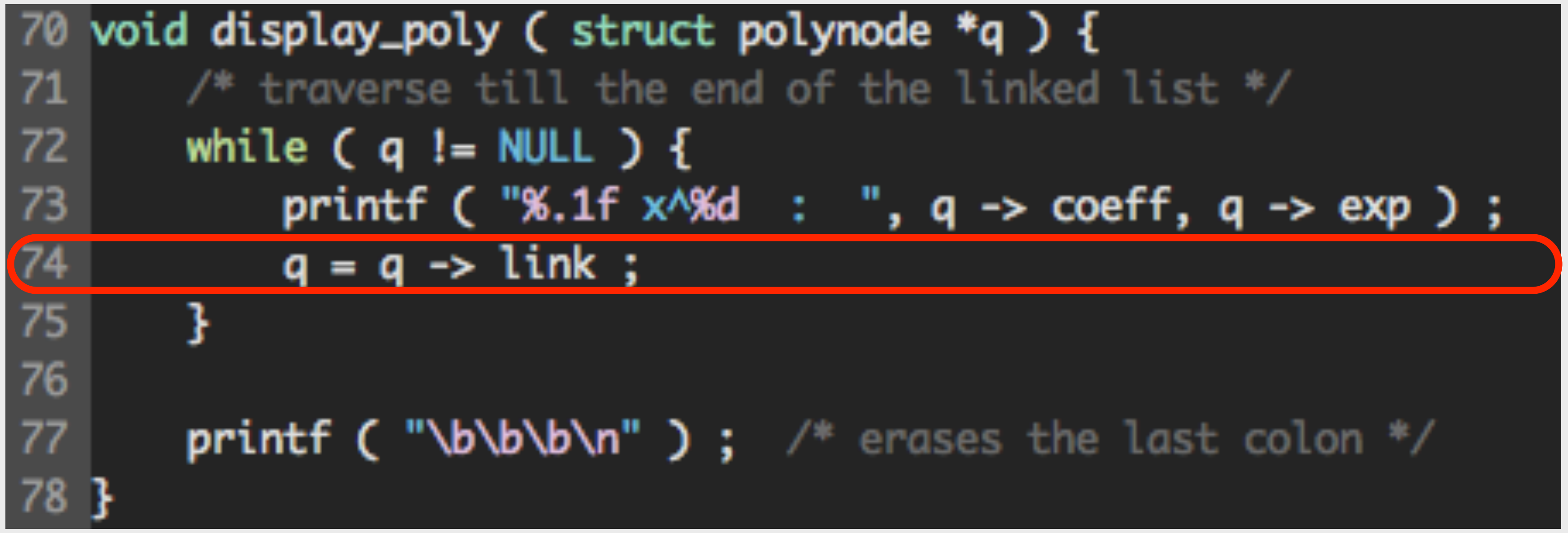


```
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

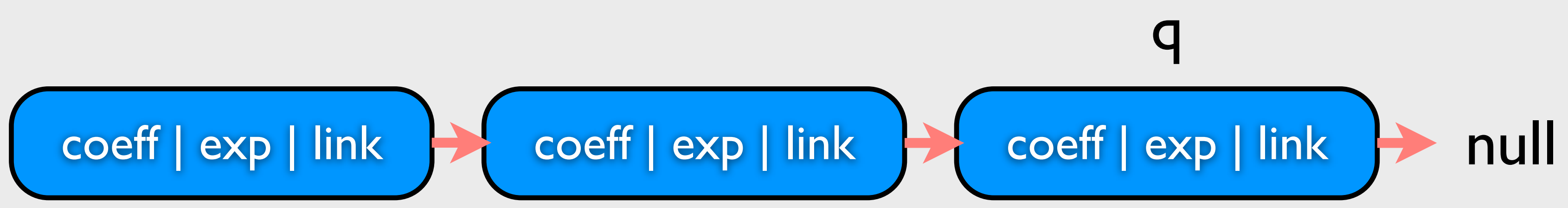


Example: poly/poly.c

- display_poly

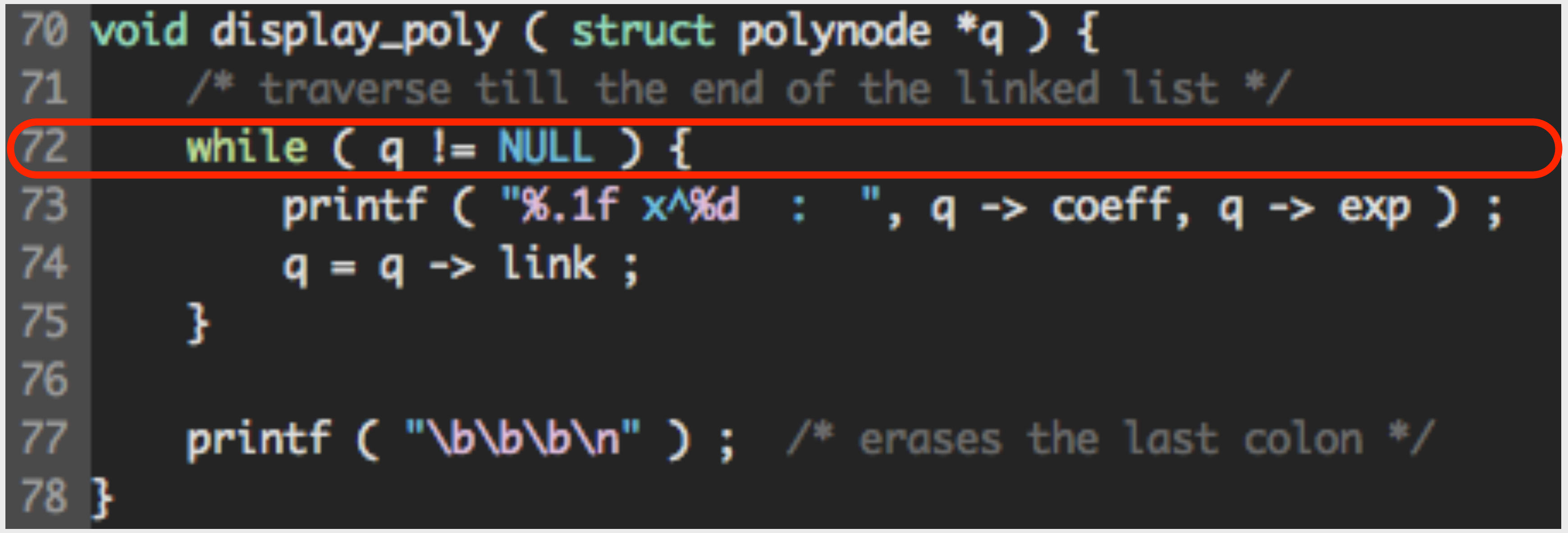


```
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

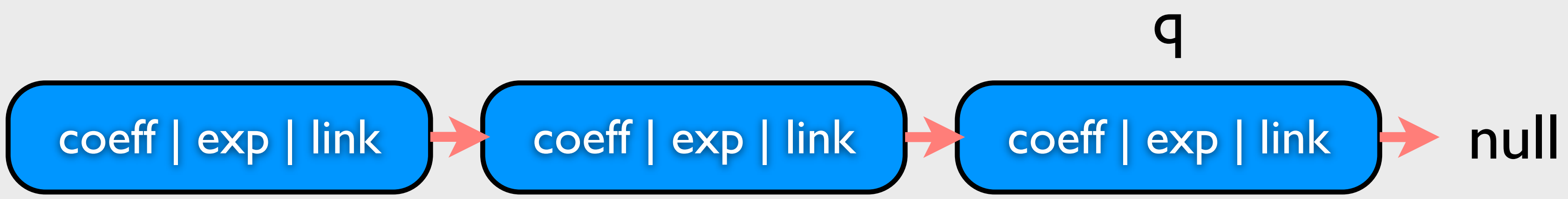


Example: poly/poly.c

- display_poly

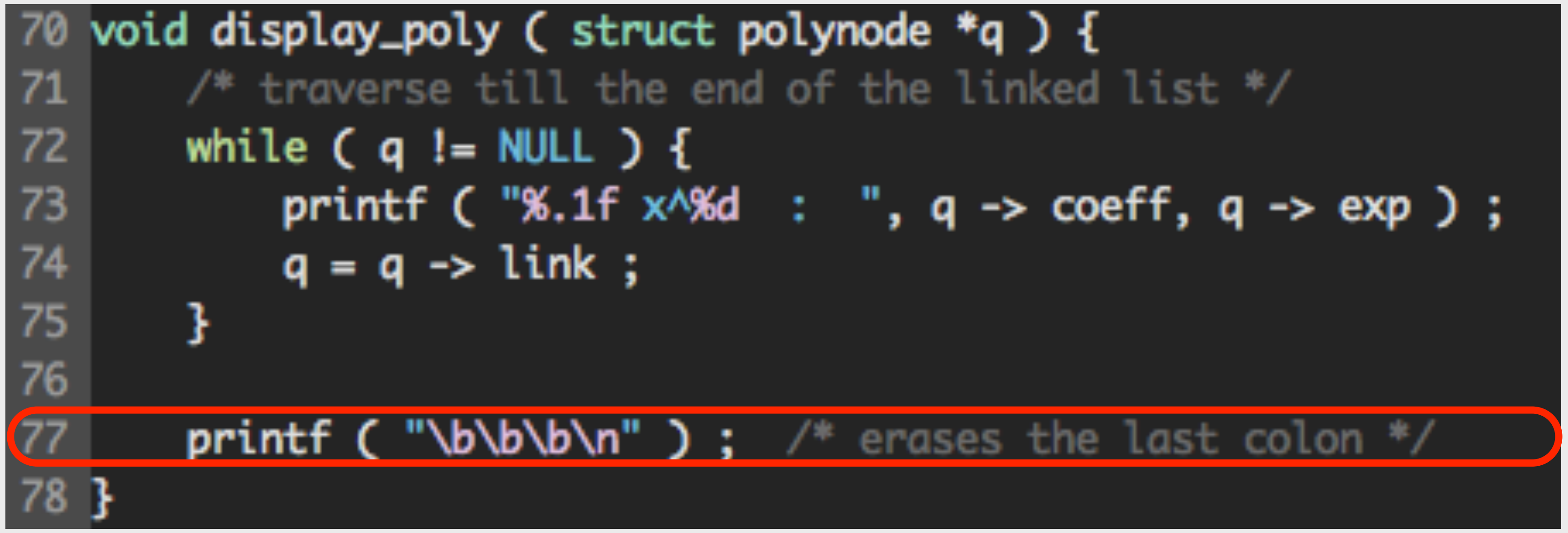


```
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

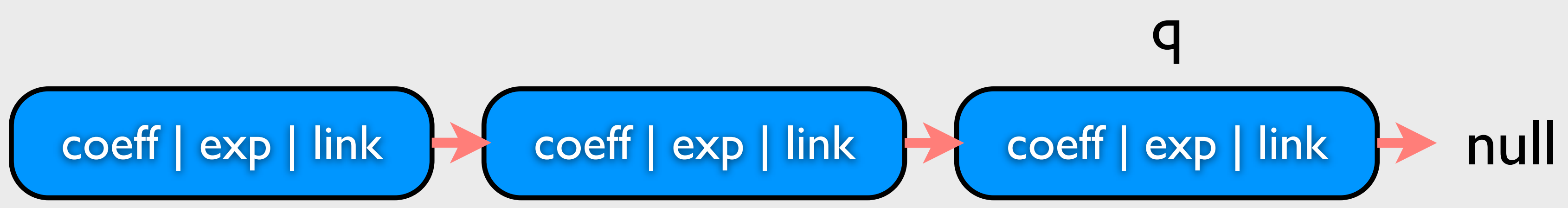


Example: poly/poly.c

- display_poly

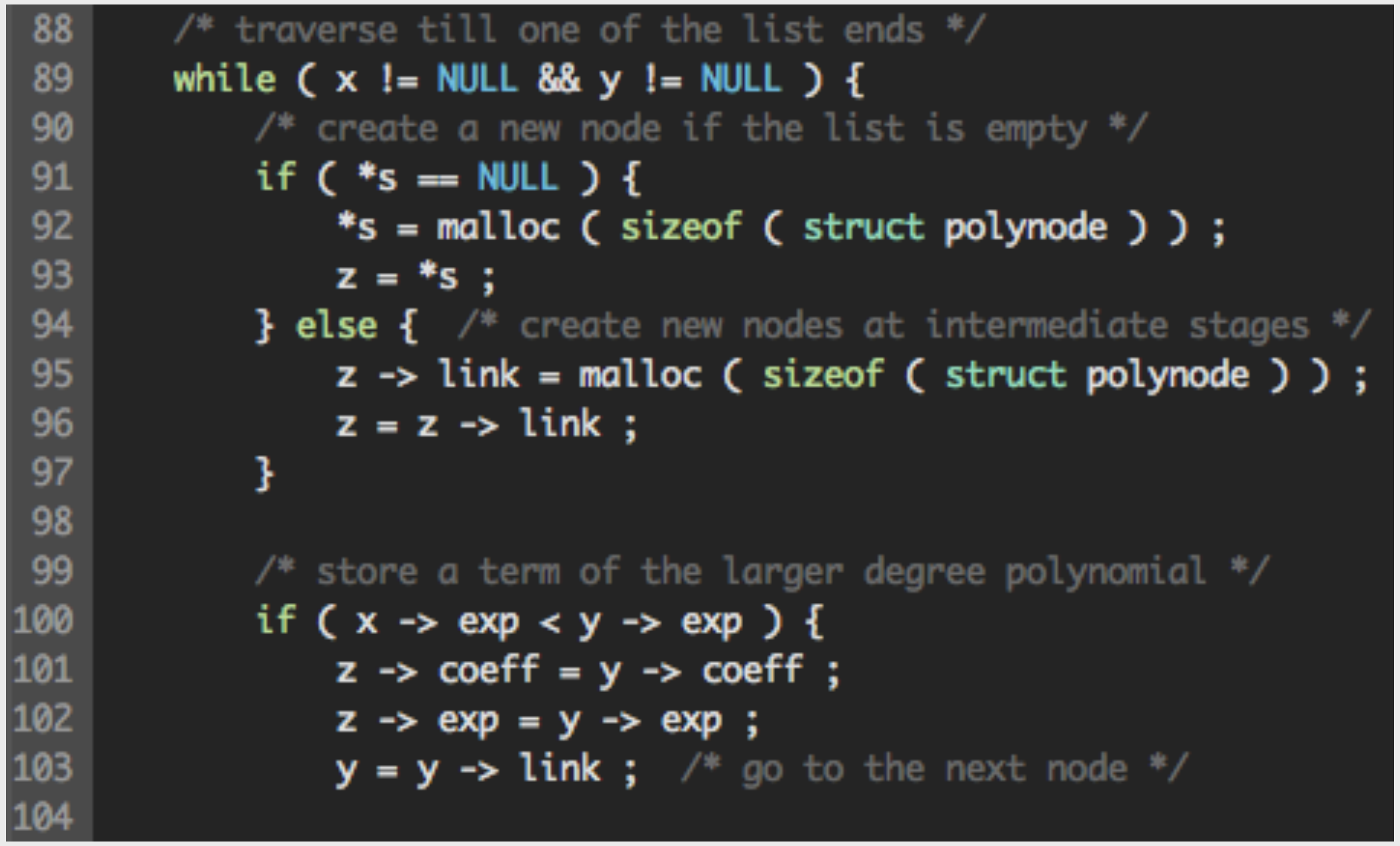


```c
void display_poly ( struct polynode *q ) {
    /* traverse till the end of the linked list */
    while ( q != NULL ) {
        printf ( "%.1f x^%d  :  ", q -> coeff, q -> exp ) ;
        q = q -> link ;
    }

    printf ( "\b\b\b\n" ) ;  /* erases the last colon */
}
```

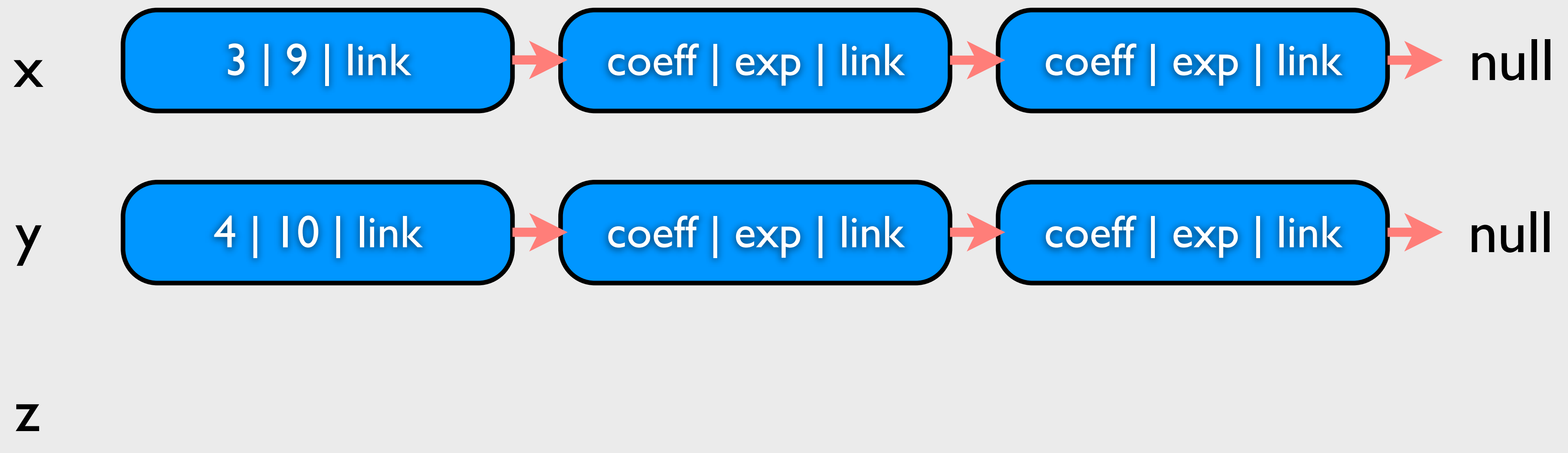


Example: poly/poly.c

- poly_add

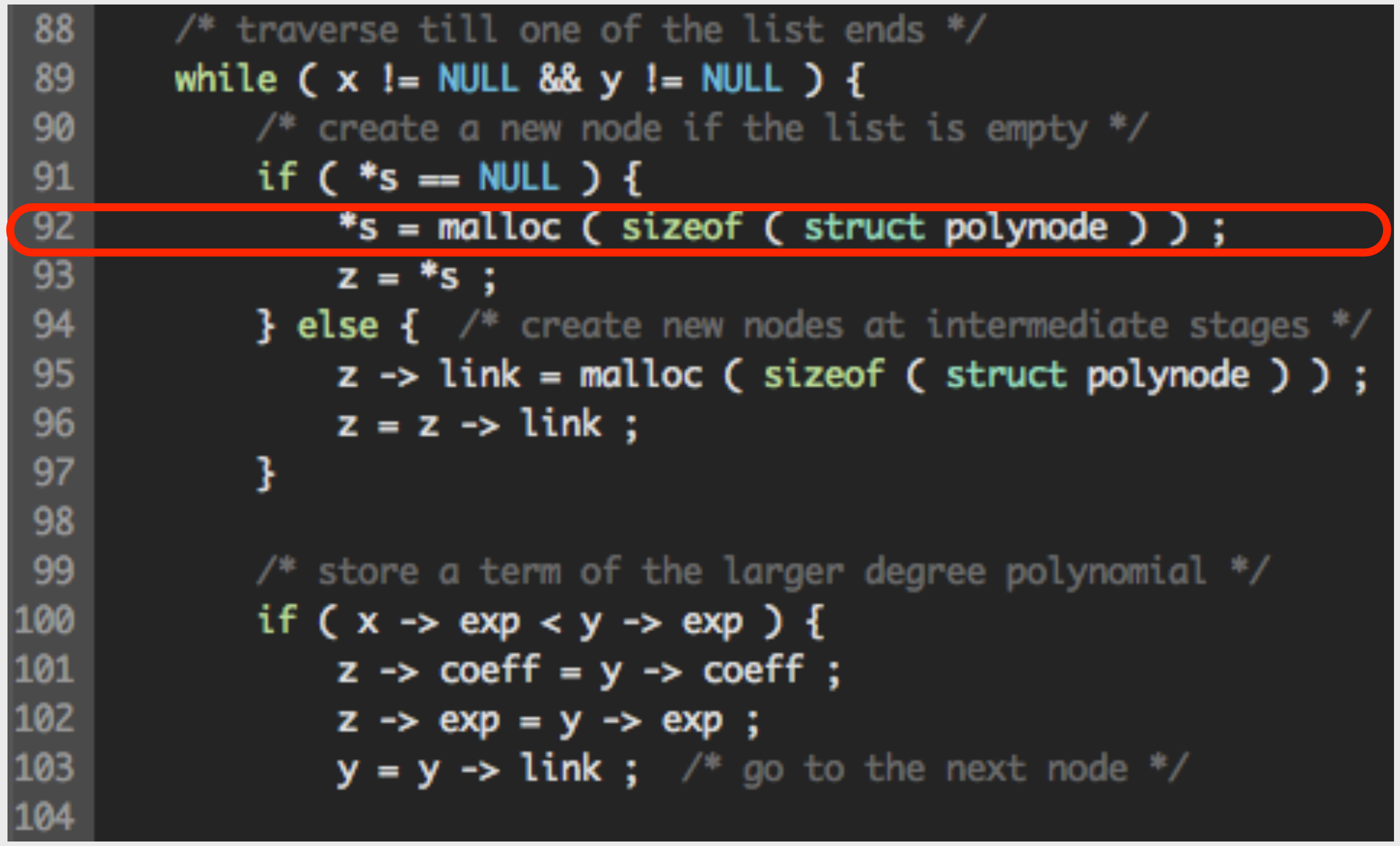


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */
```

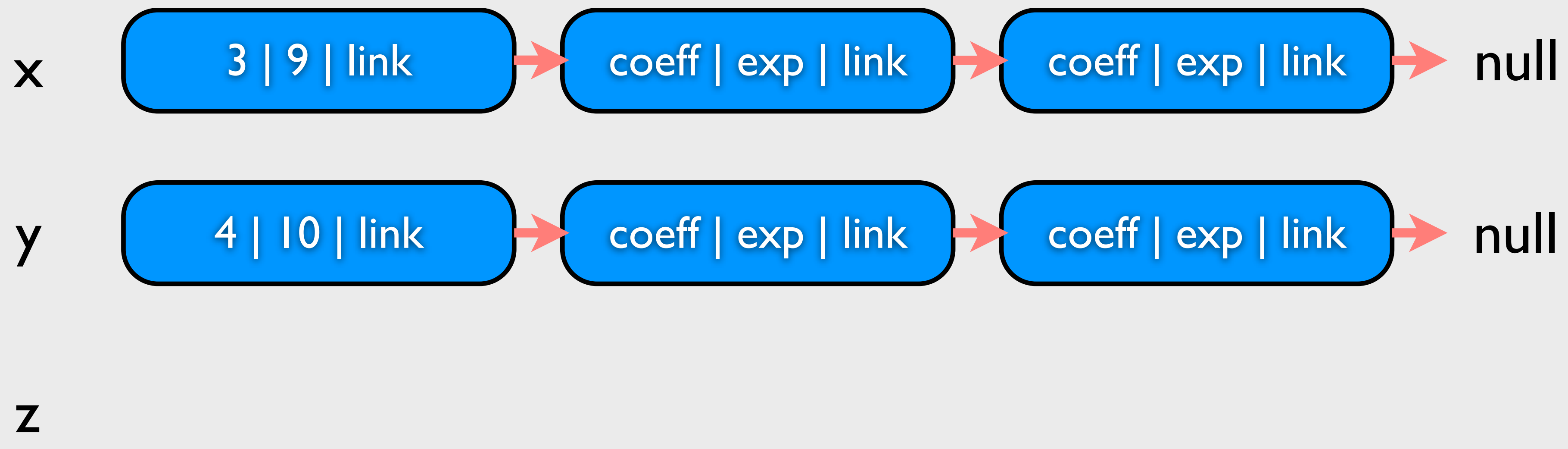


Example: poly/poly.c

- poly_add

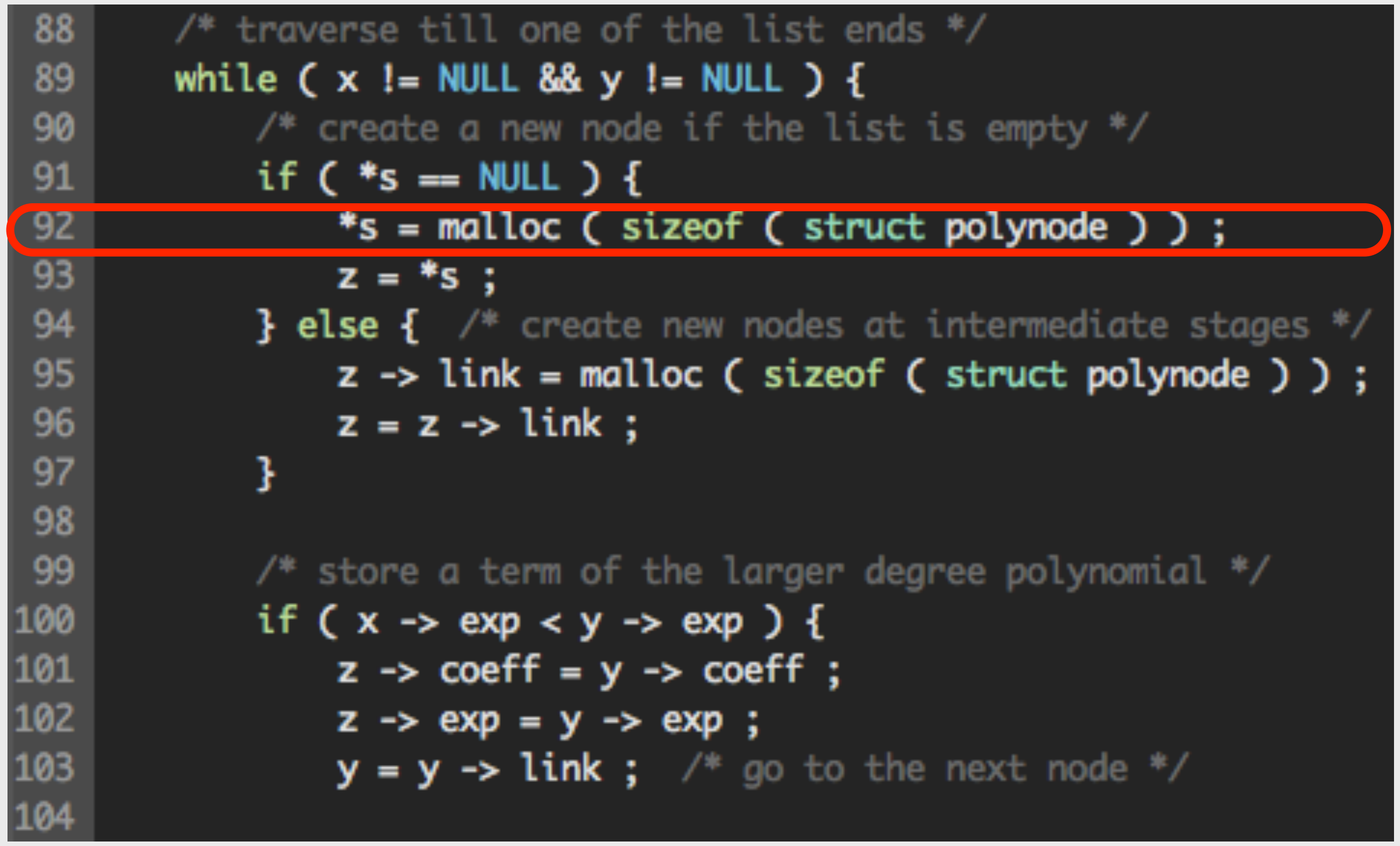


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;   /* go to the next node */

```

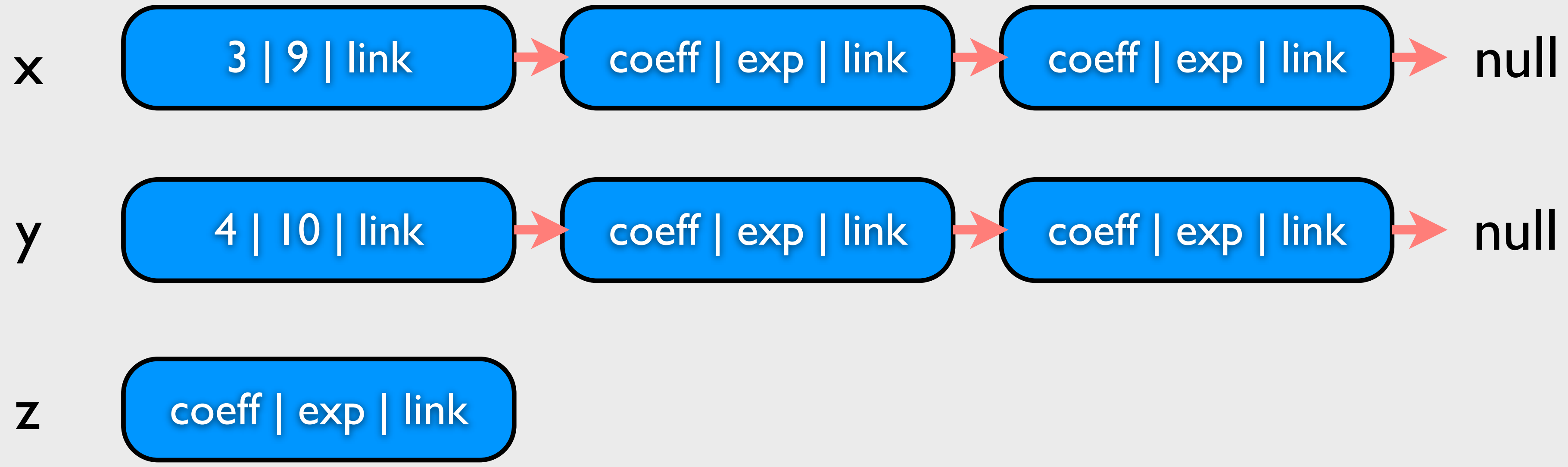


Example: poly/poly.c

- poly_add

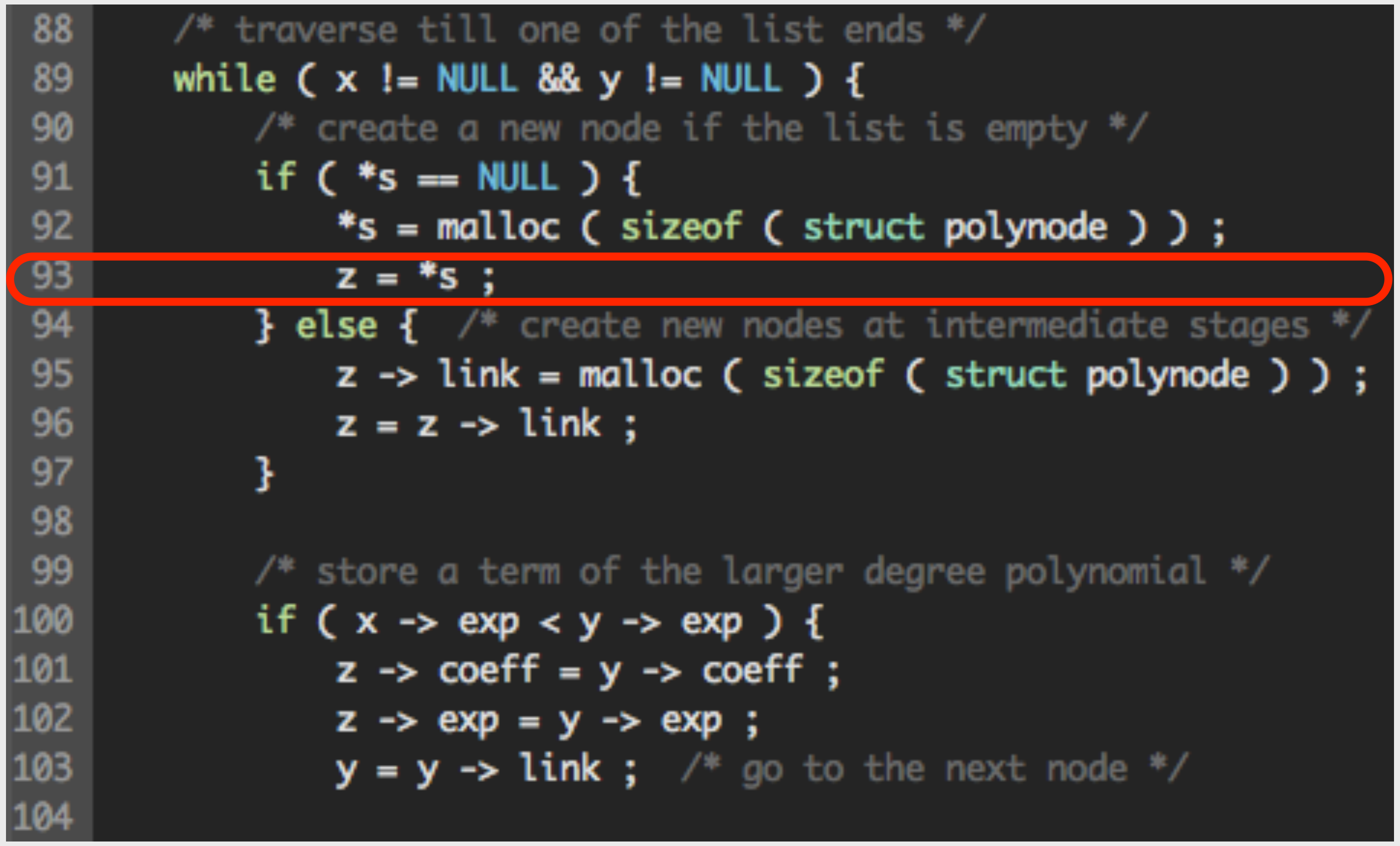


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

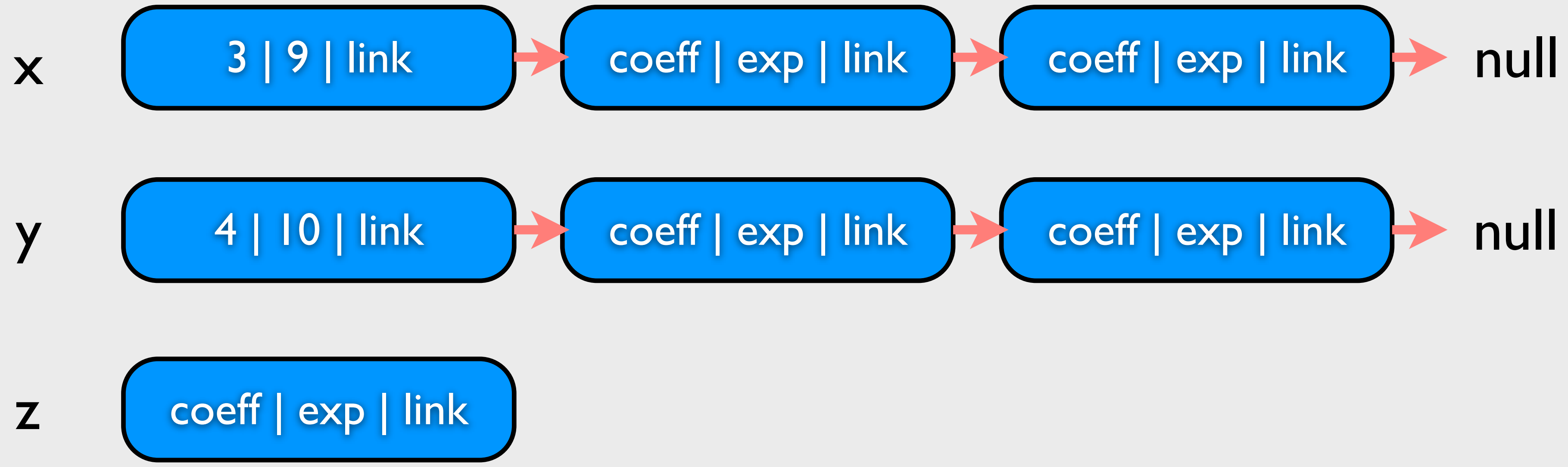


Example: poly/poly.c

- poly_add

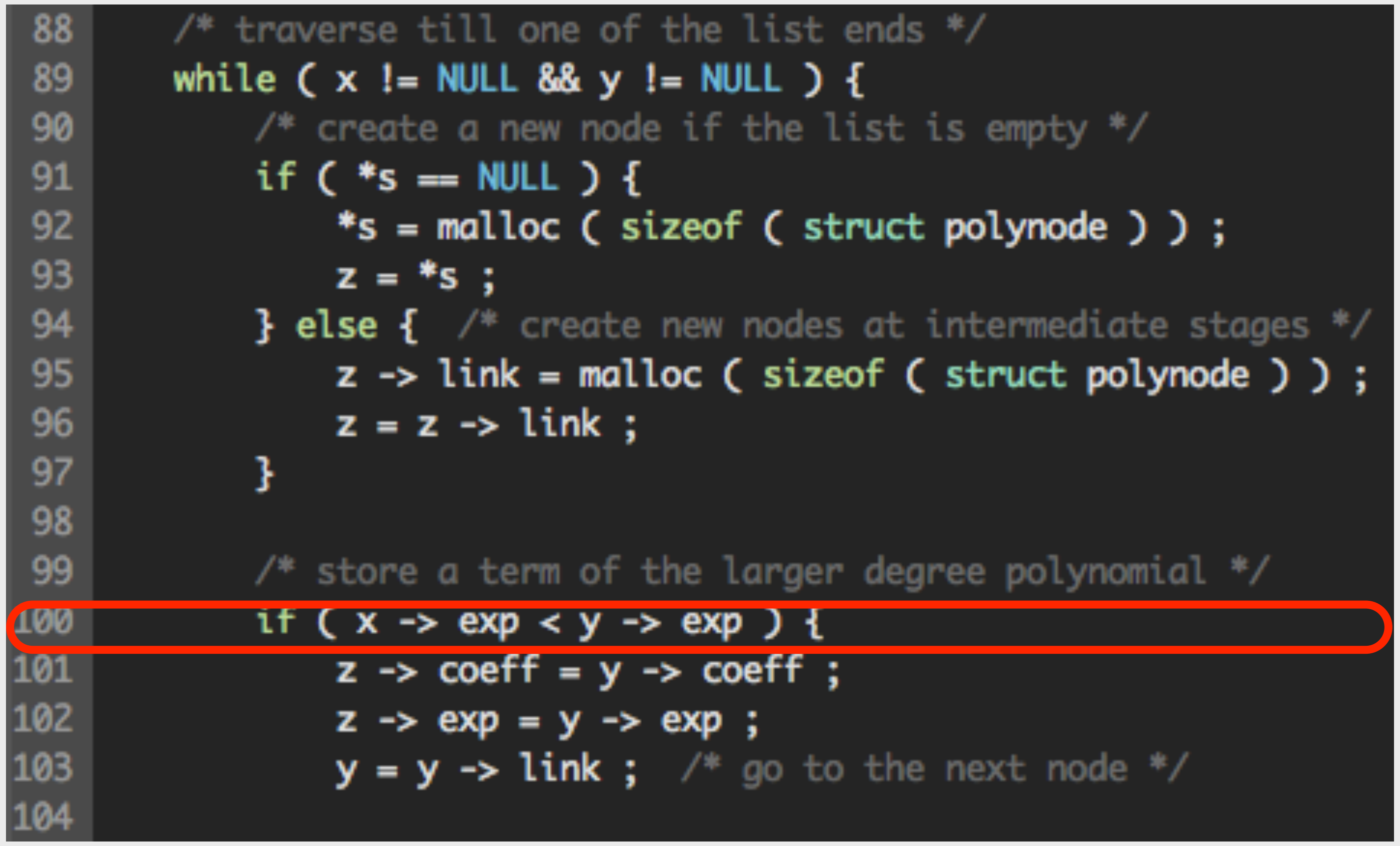


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

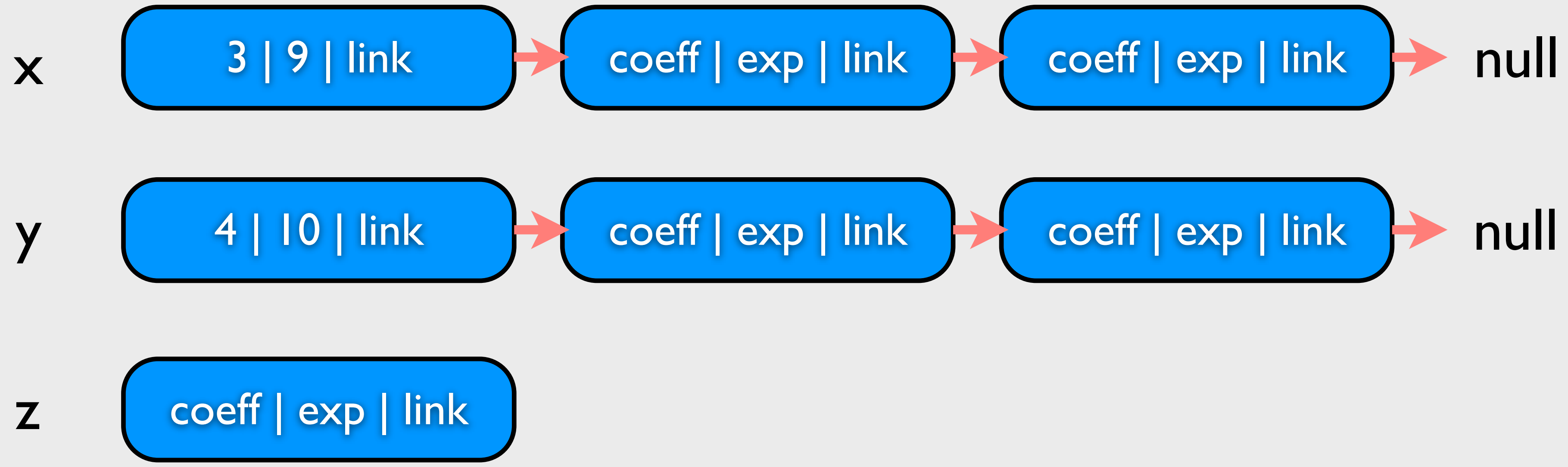


Example: poly/poly.c

- poly_add

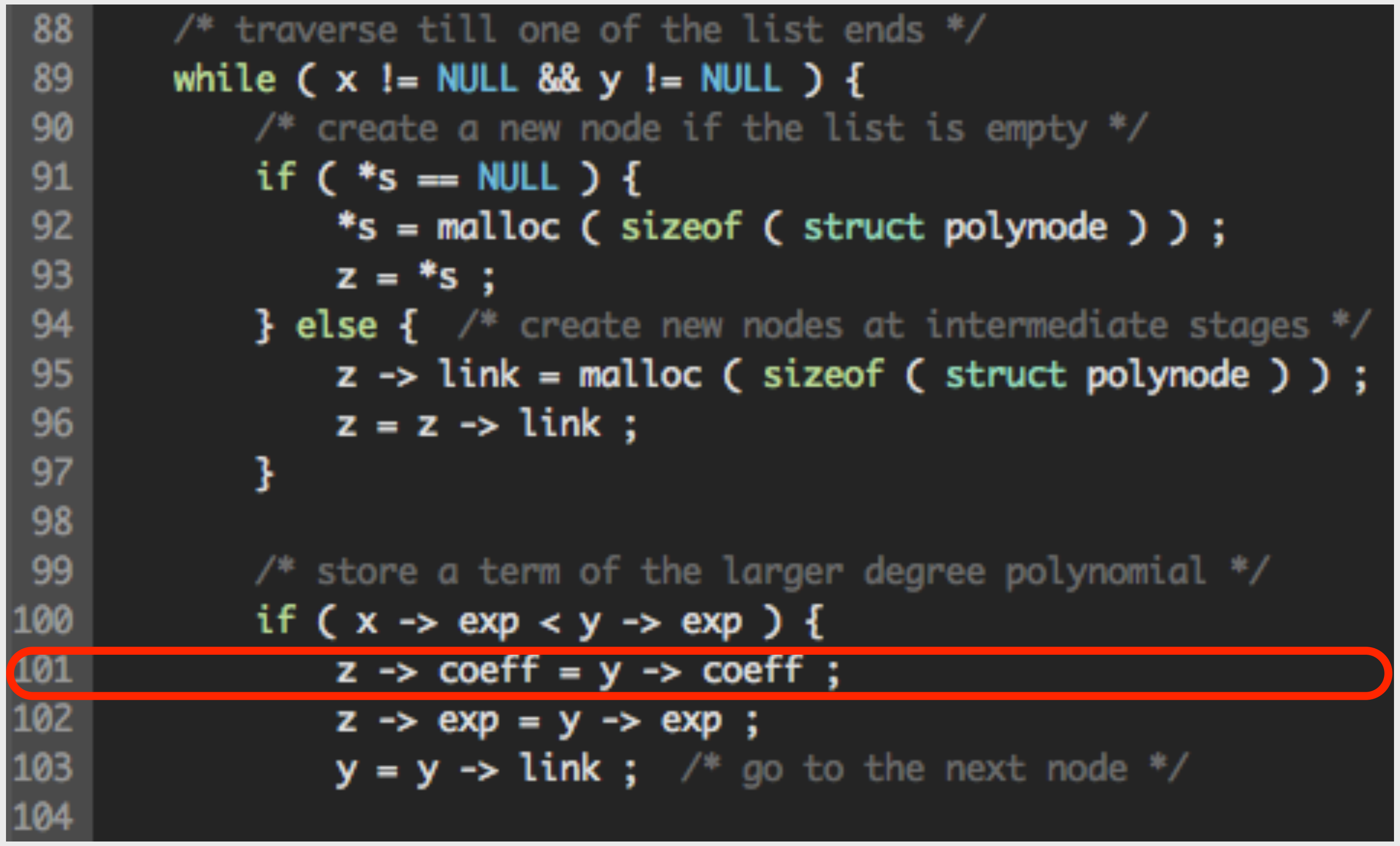


```
/* traverse till one of the list ends */
while ( x != NULL && y != NULL ) {
    /* create a new node if the list is empty */
    if ( *s == NULL ) {
        *s = malloc ( sizeof ( struct polynode ) ) ;
        z = *s ;
    } else {  /* create new nodes at intermediate stages */
        z -> link = malloc ( sizeof ( struct polynode ) ) ;
        z = z -> link ;
    }

    /* store a term of the larger degree polynomial */
    if ( x -> exp < y -> exp ) {
        z -> coeff = y -> coeff ;
        z -> exp = y -> exp ;
        y = y -> link ;  /* go to the next node */

```

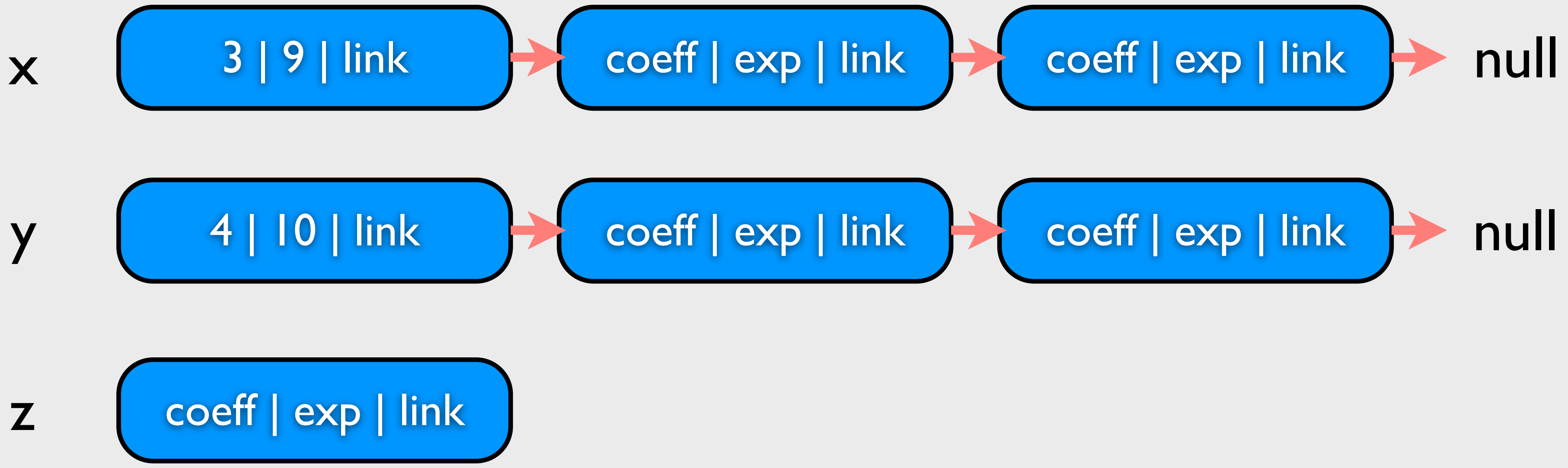


Example: poly/poly.c

- poly_add

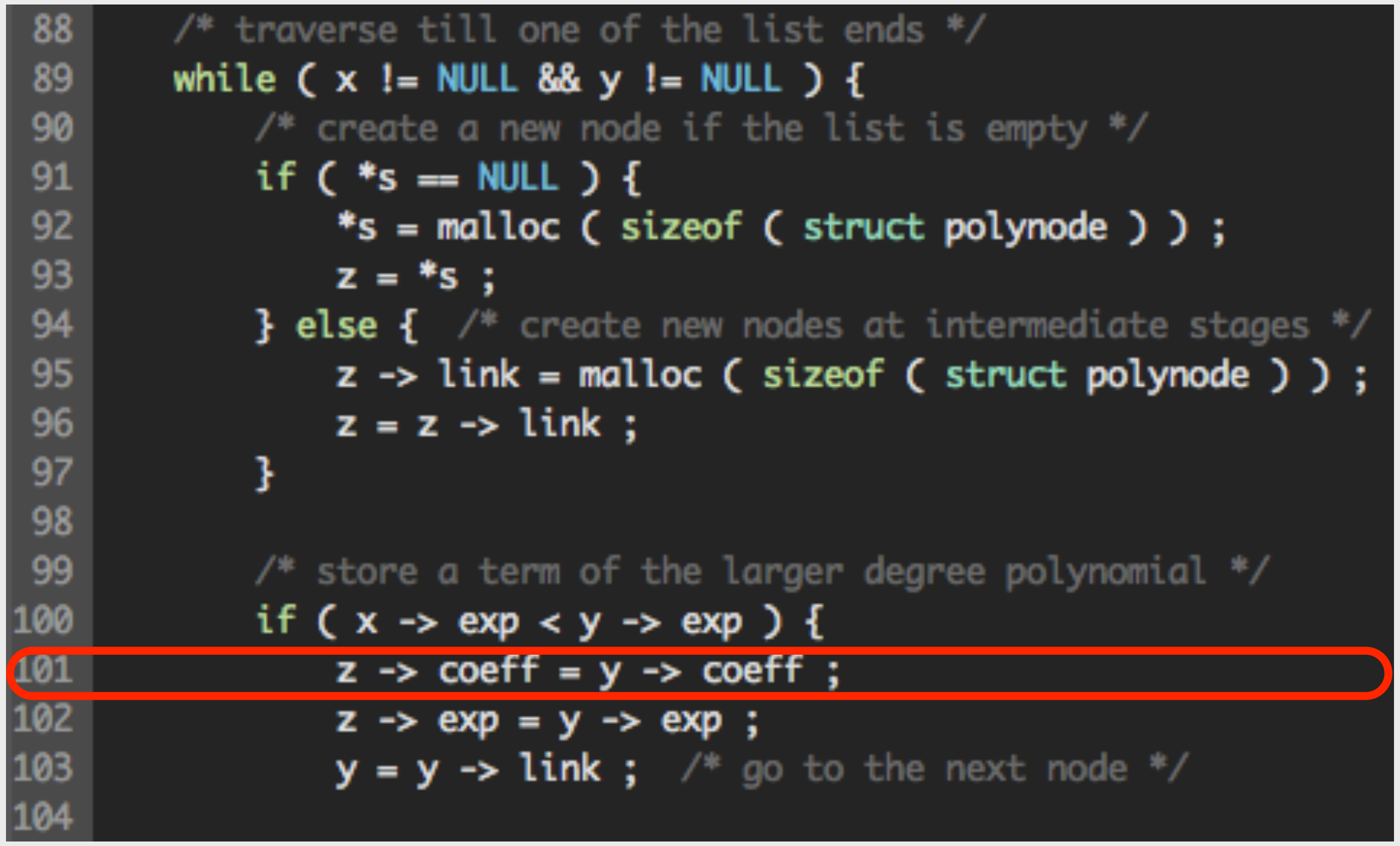


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

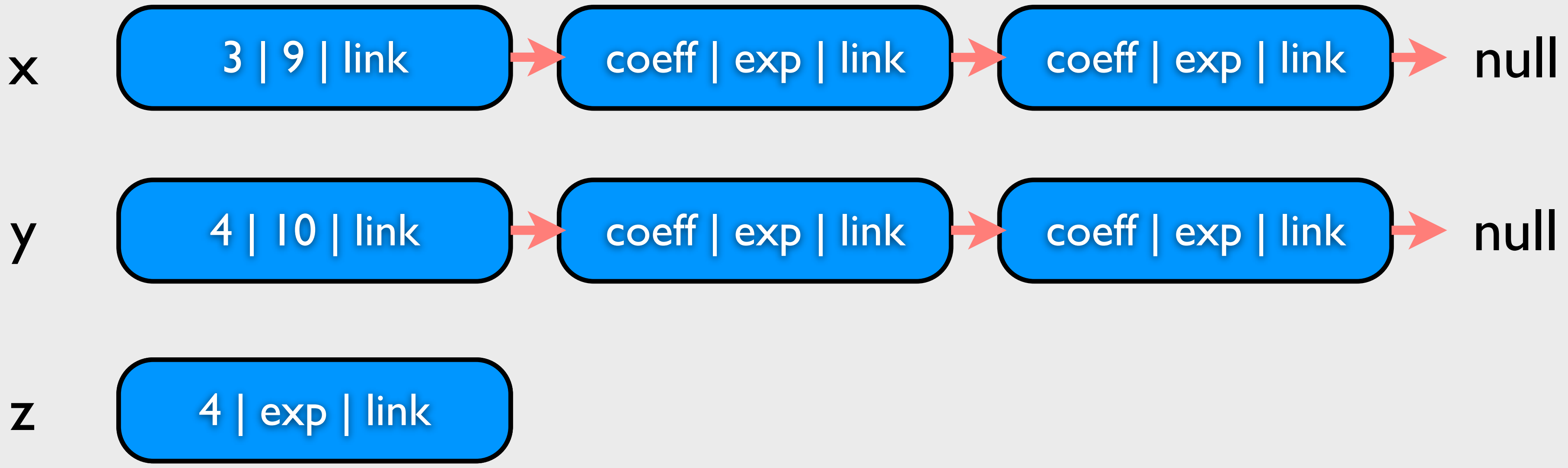


Example: poly/poly.c

- poly_add

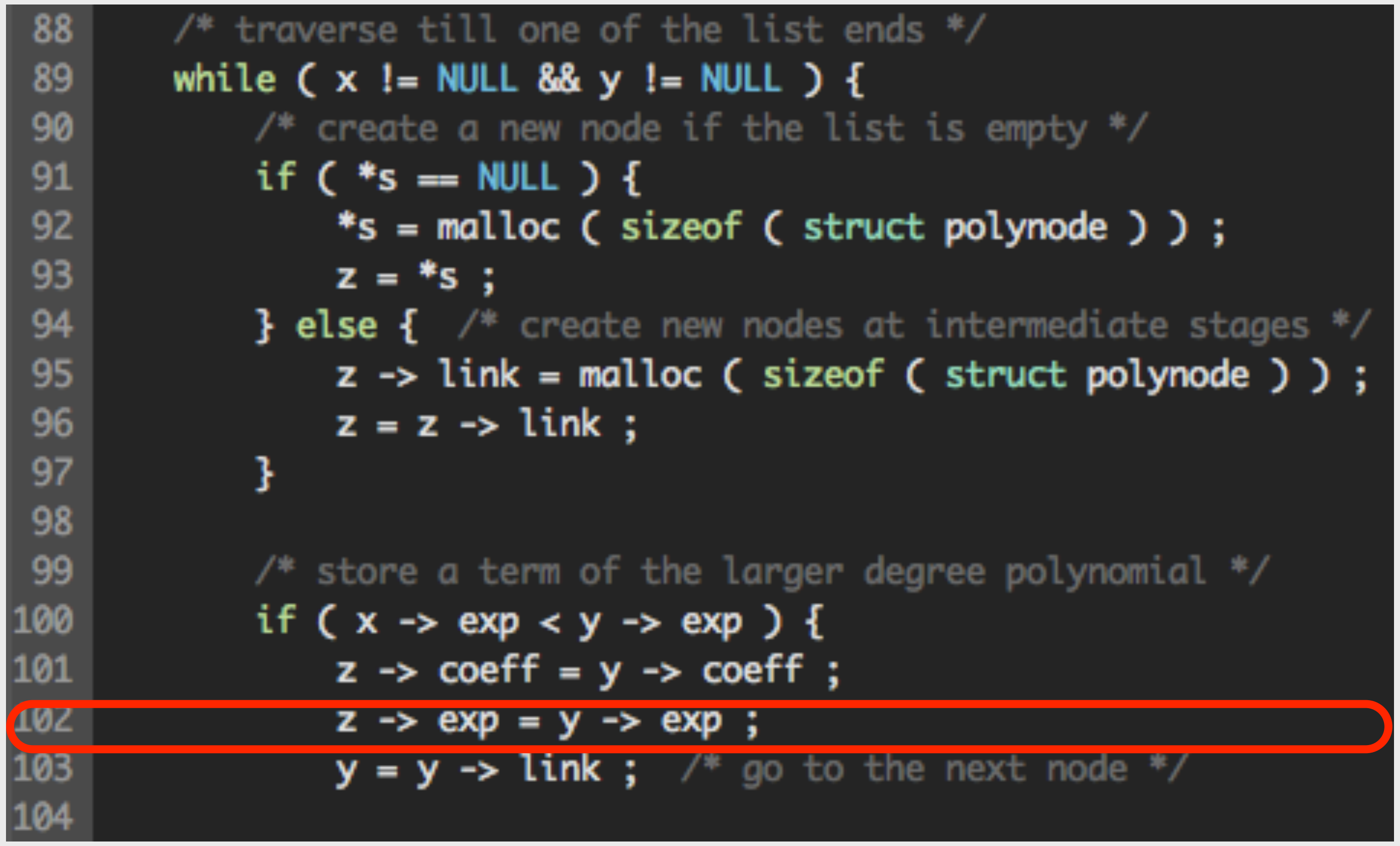


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

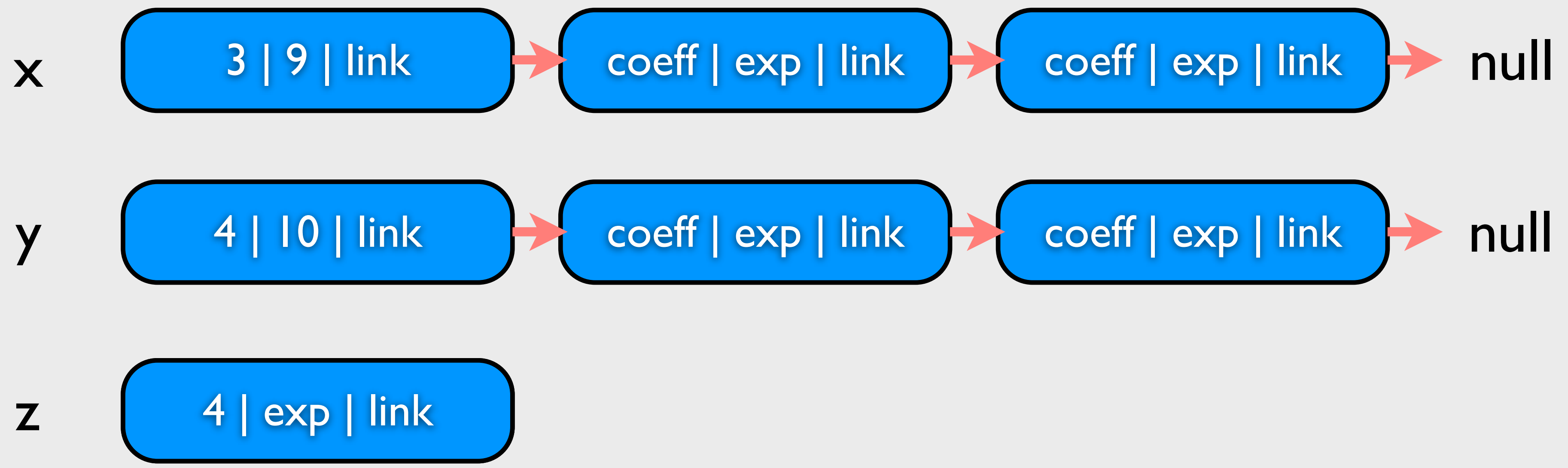


Example: poly/poly.c

- poly_add

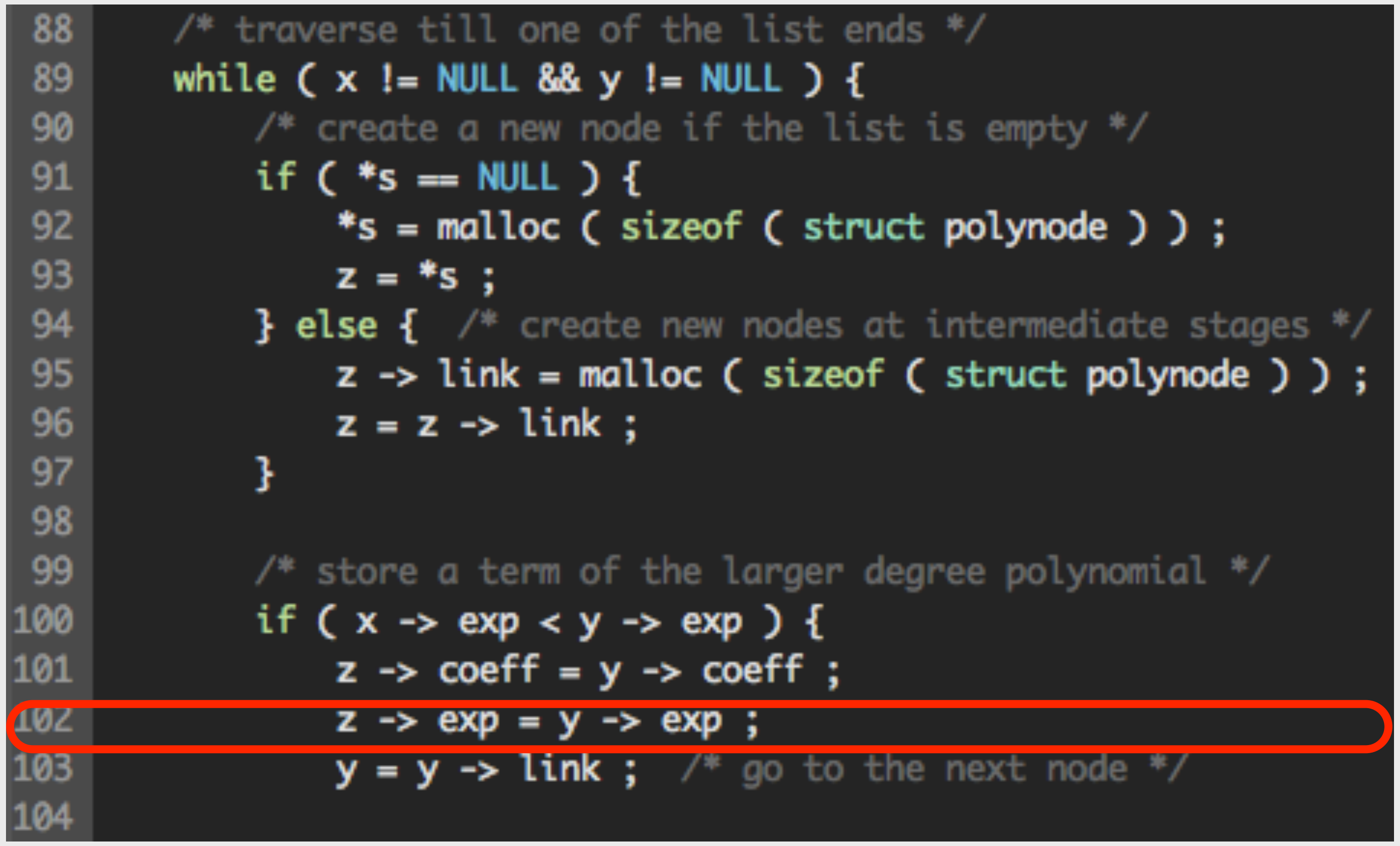


```c
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */
```

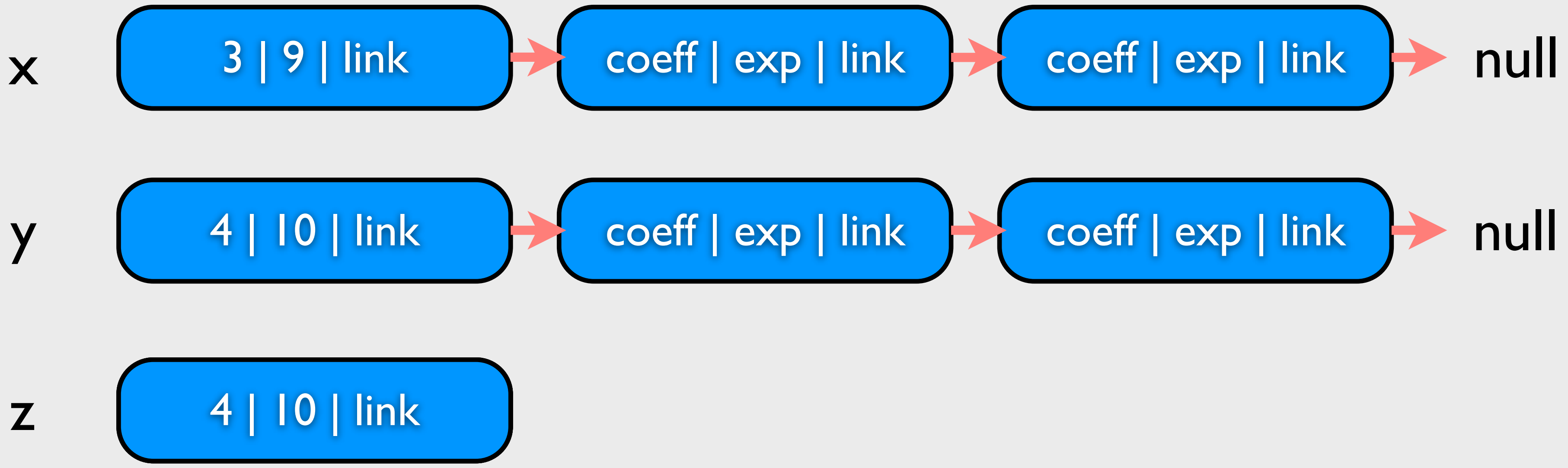


Example: poly/poly.c

- poly_add

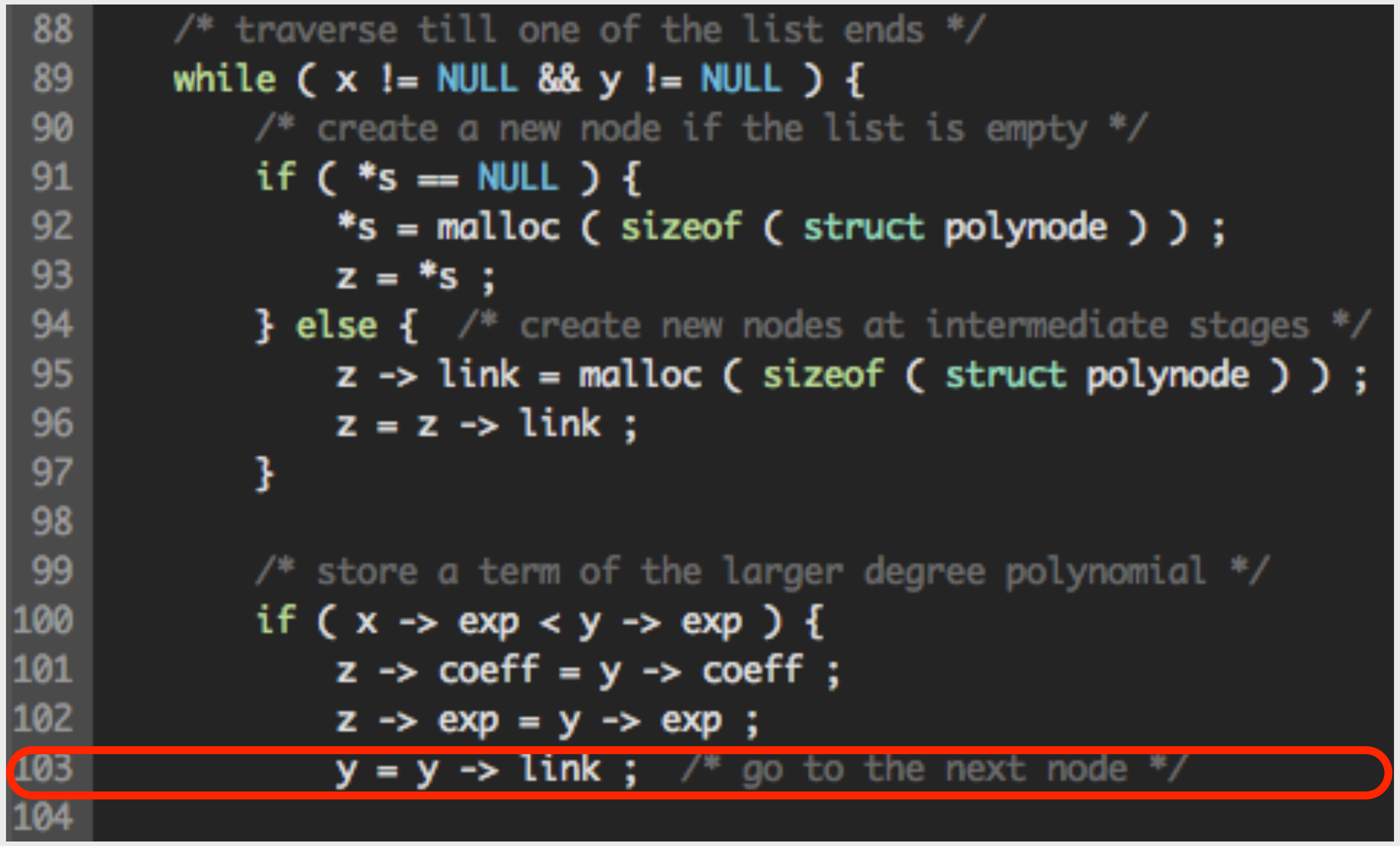


```c
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

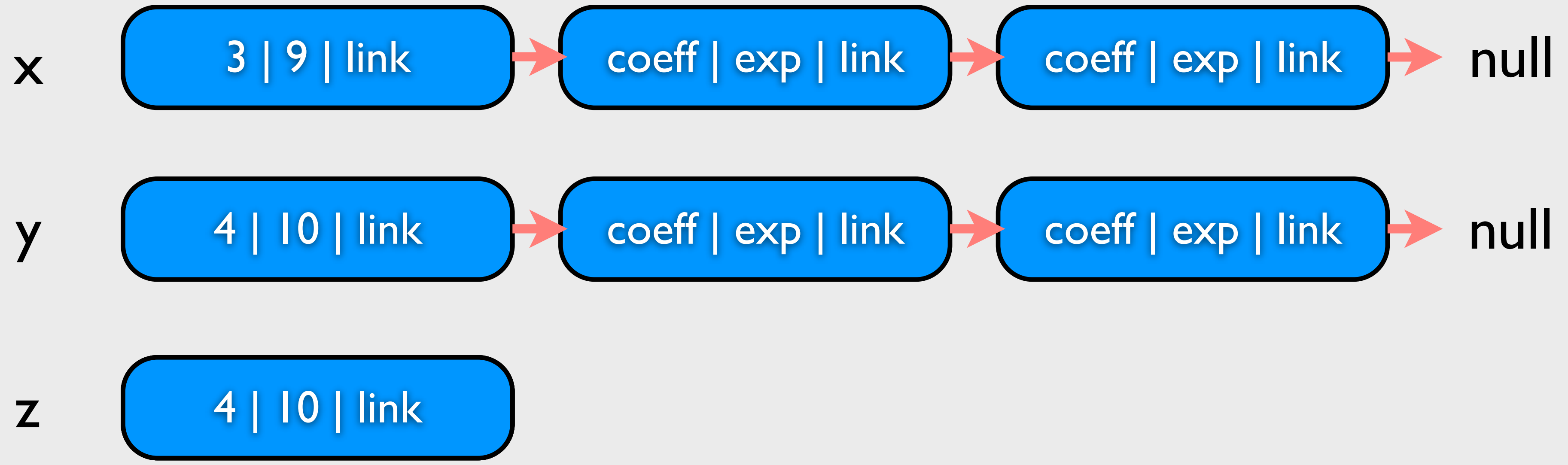


Example: poly/poly.c

- poly_add

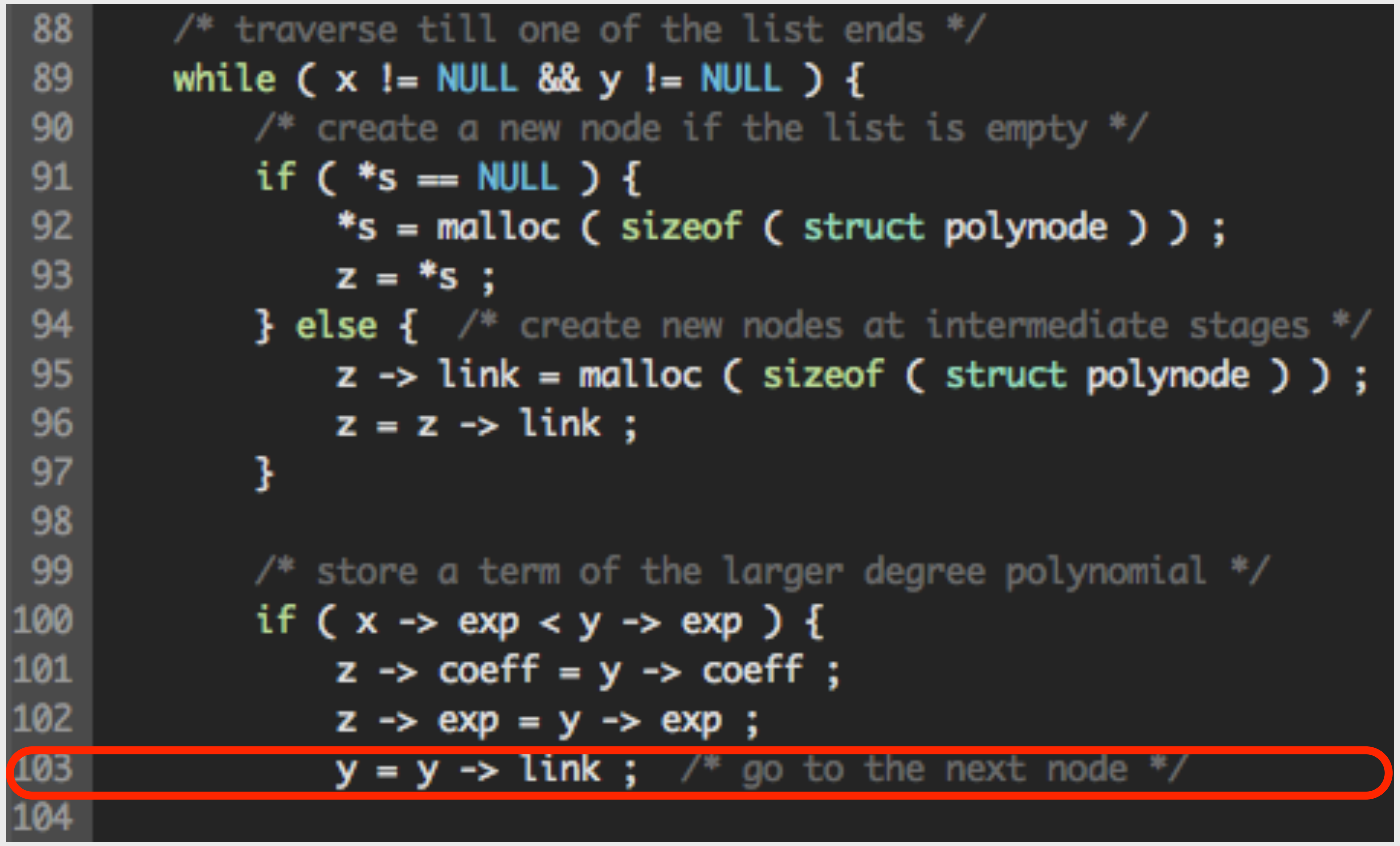


```
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */
```

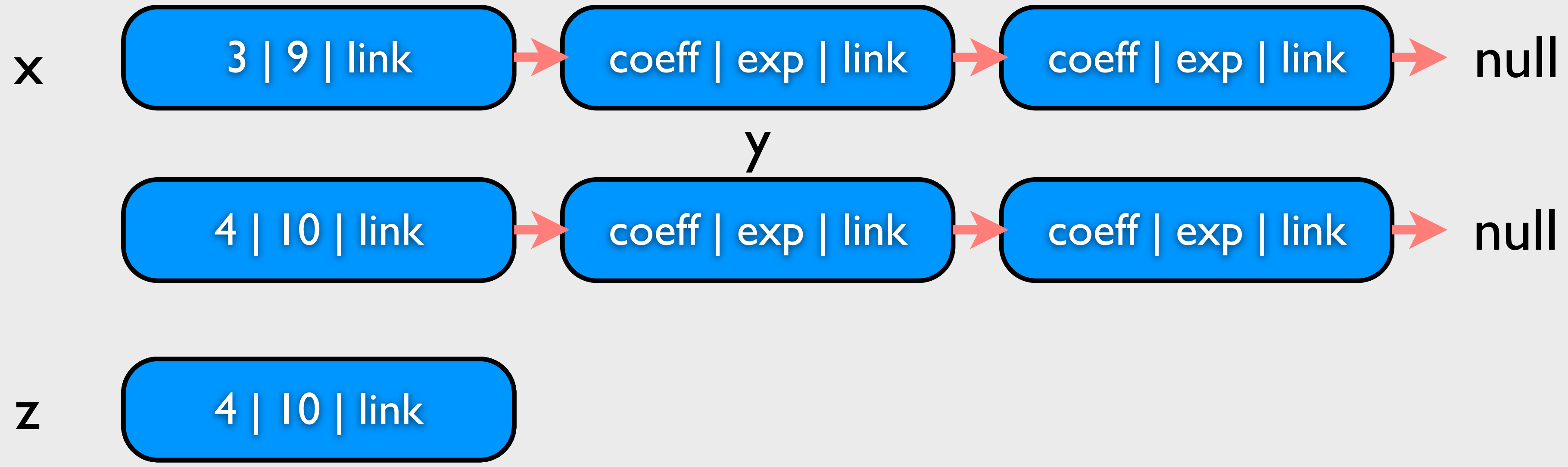


Example: poly/poly.c

- poly_add

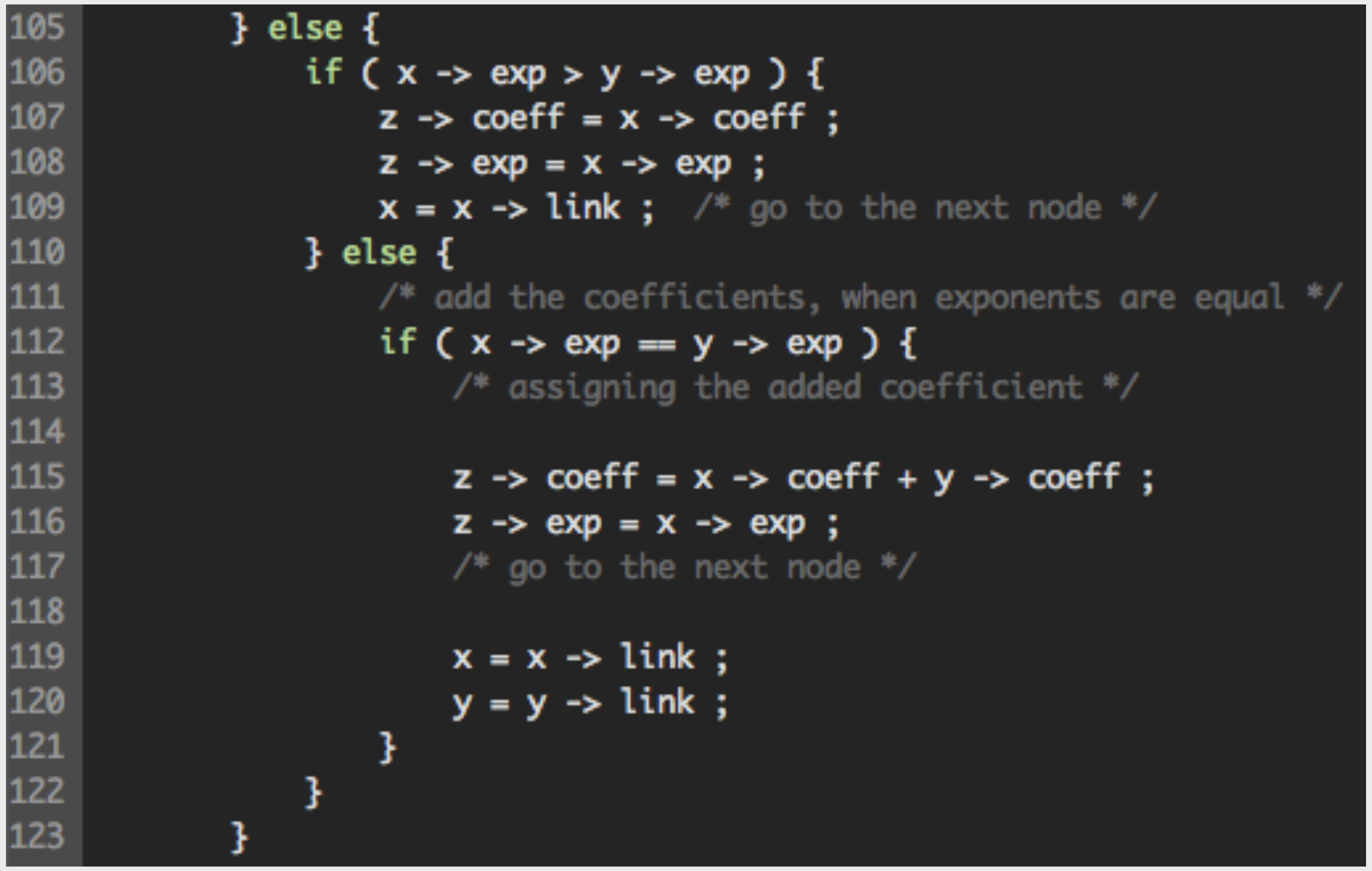


```c
    /* traverse till one of the list ends */
    while ( x != NULL && y != NULL ) {
        /* create a new node if the list is empty */
        if ( *s == NULL ) {
            *s = malloc ( sizeof ( struct polynode ) ) ;
            z = *s ;
        } else {  /* create new nodes at intermediate stages */
            z -> link = malloc ( sizeof ( struct polynode ) ) ;
            z = z -> link ;
        }

        /* store a term of the larger degree polynomial */
        if ( x -> exp < y -> exp ) {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> link ;  /* go to the next node */

```

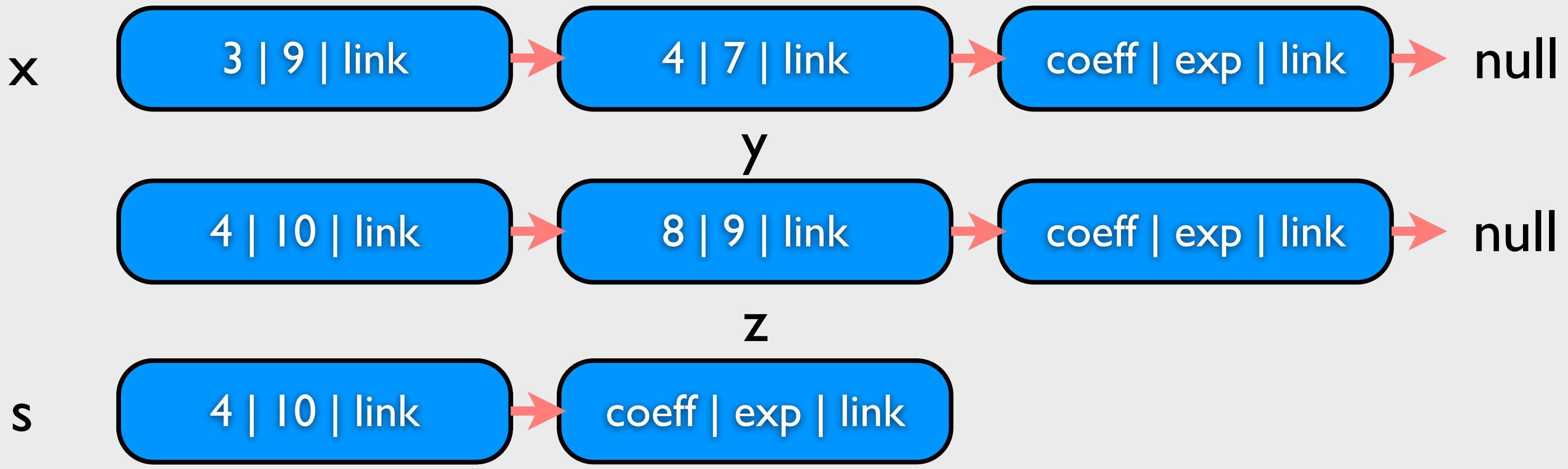


Example: poly/poly.c

- poly_add

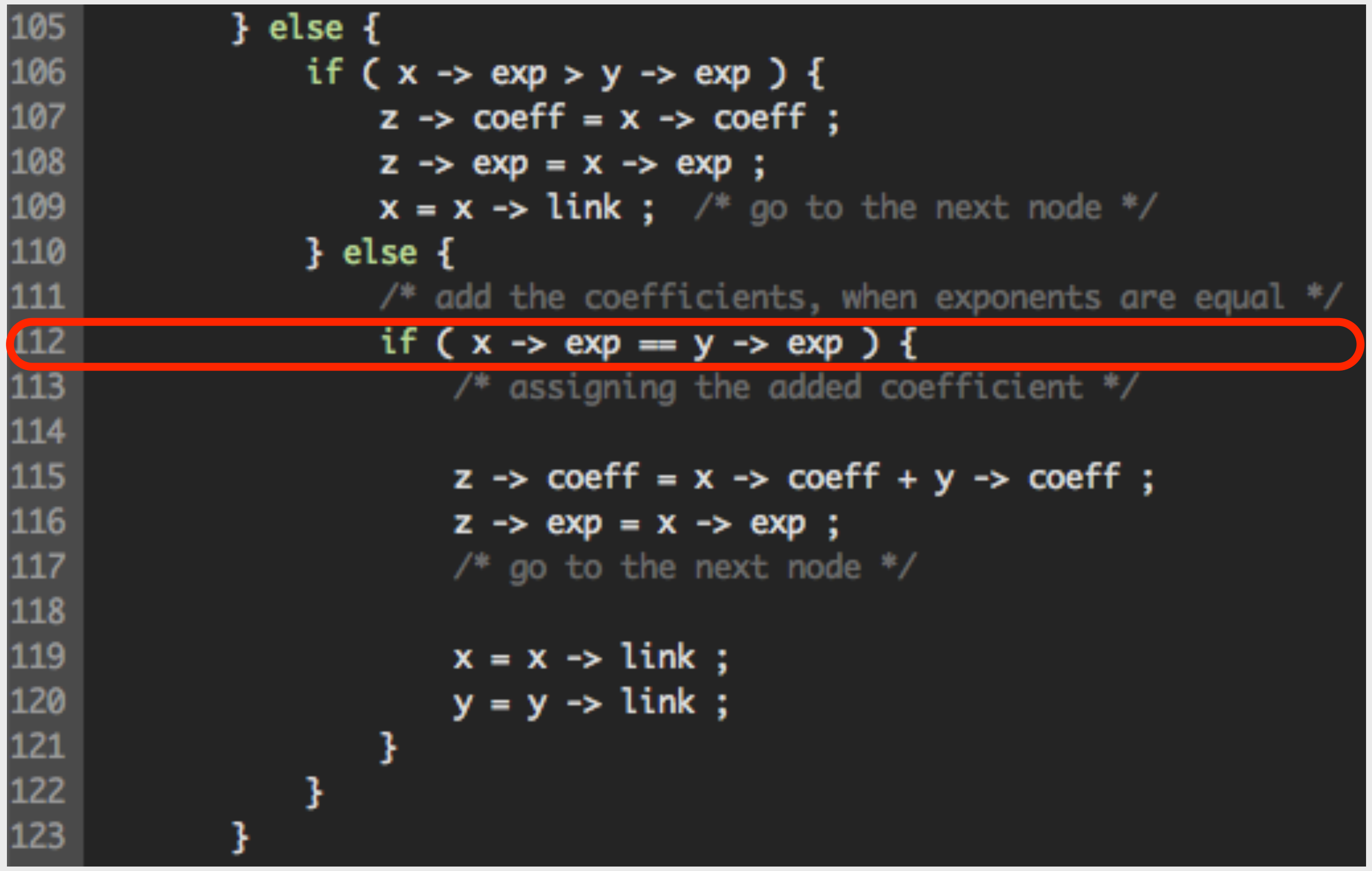


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

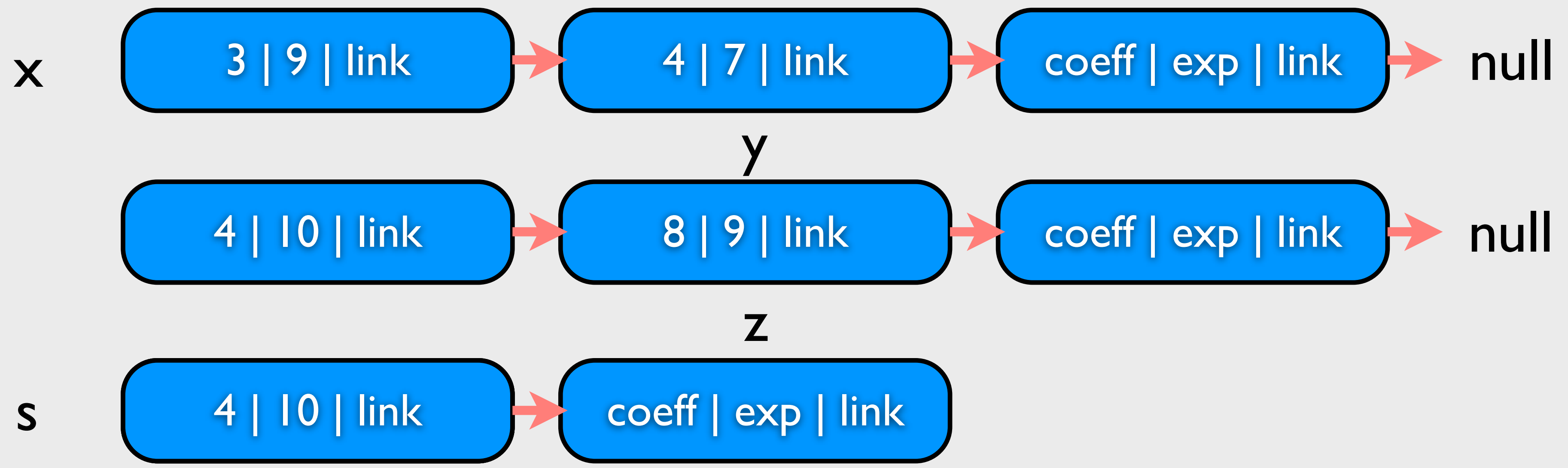


Example: poly/poly.c

- poly_add

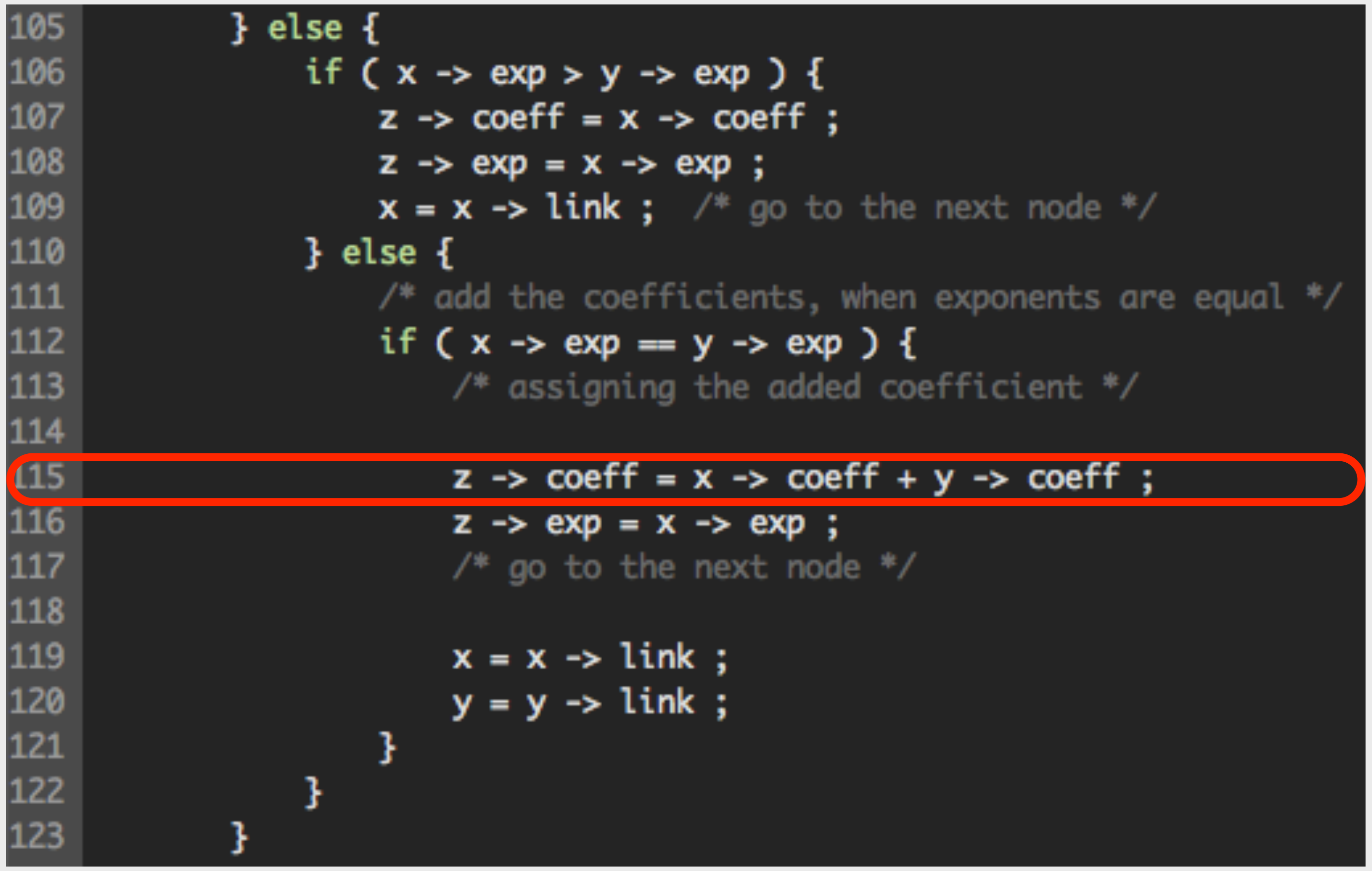


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

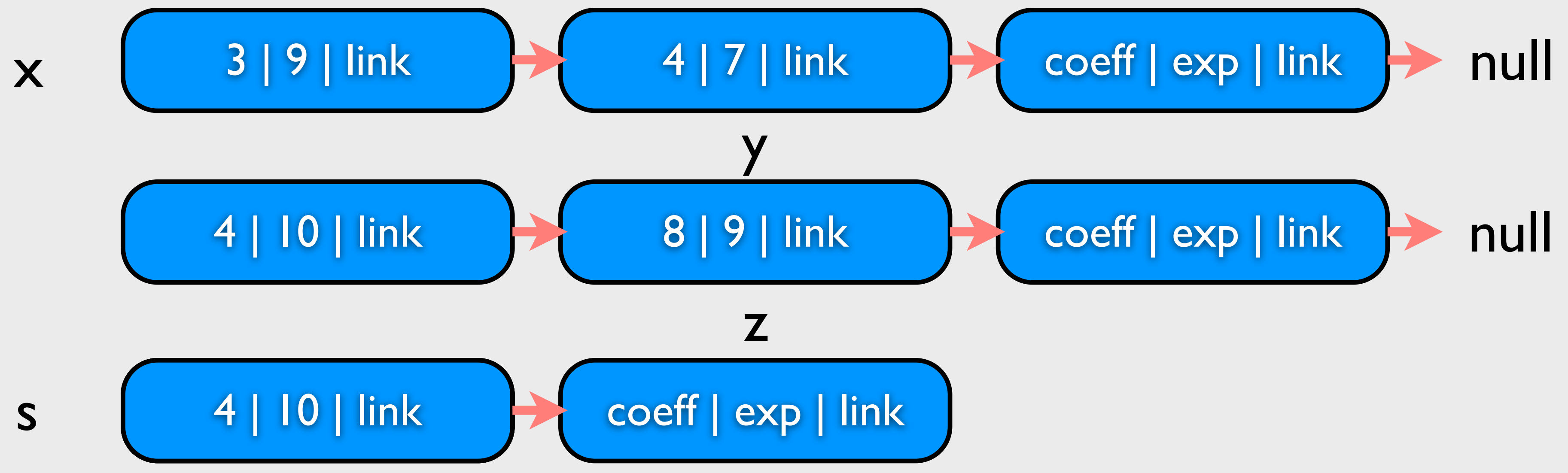


Example: poly/poly.c

- poly_add

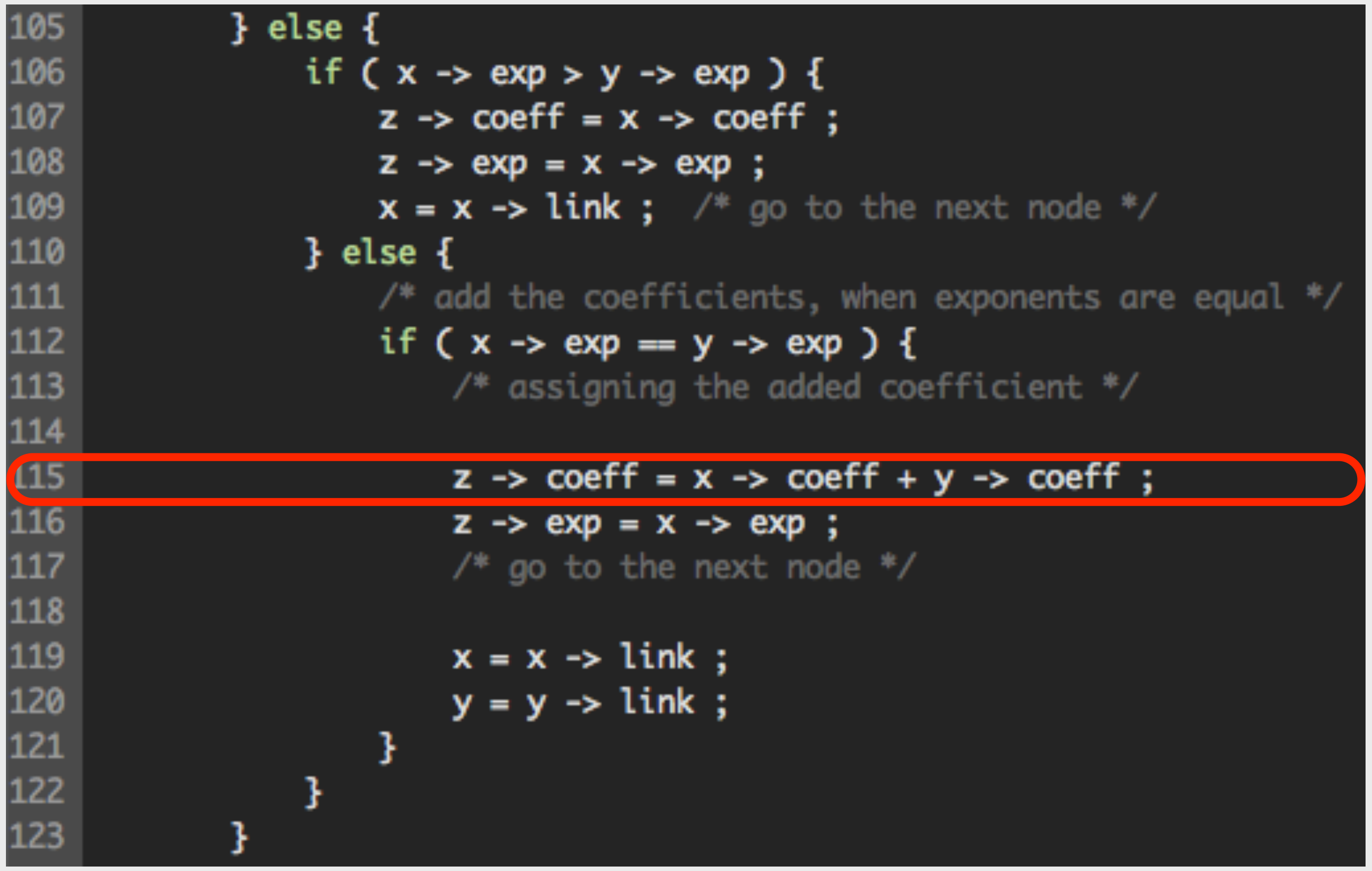


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

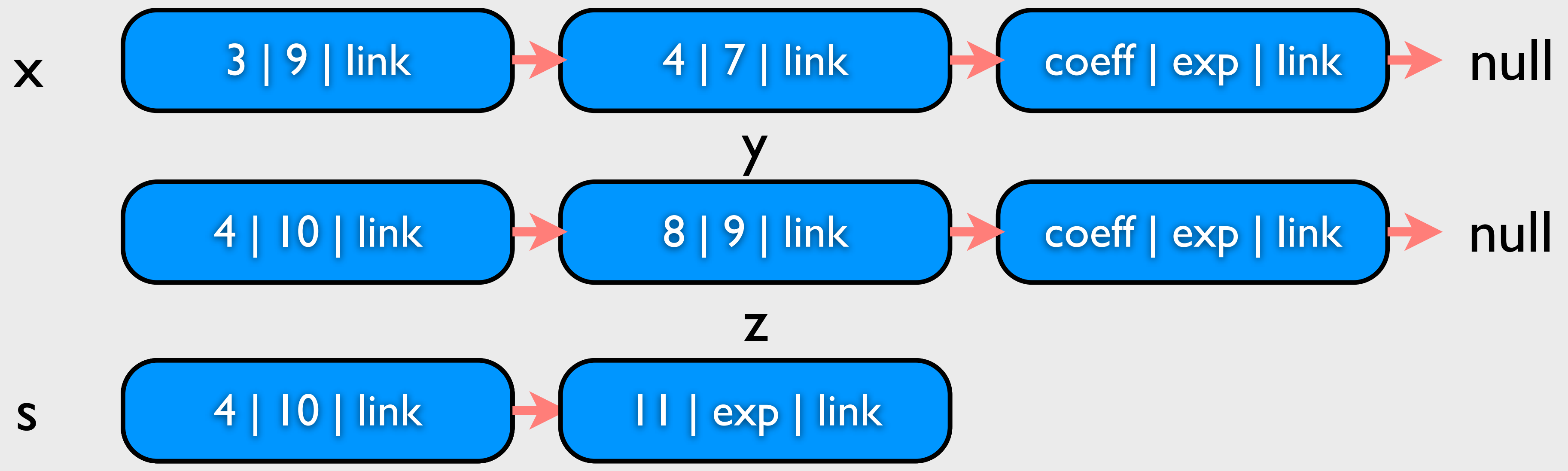


Example: poly/poly.c

- poly_add

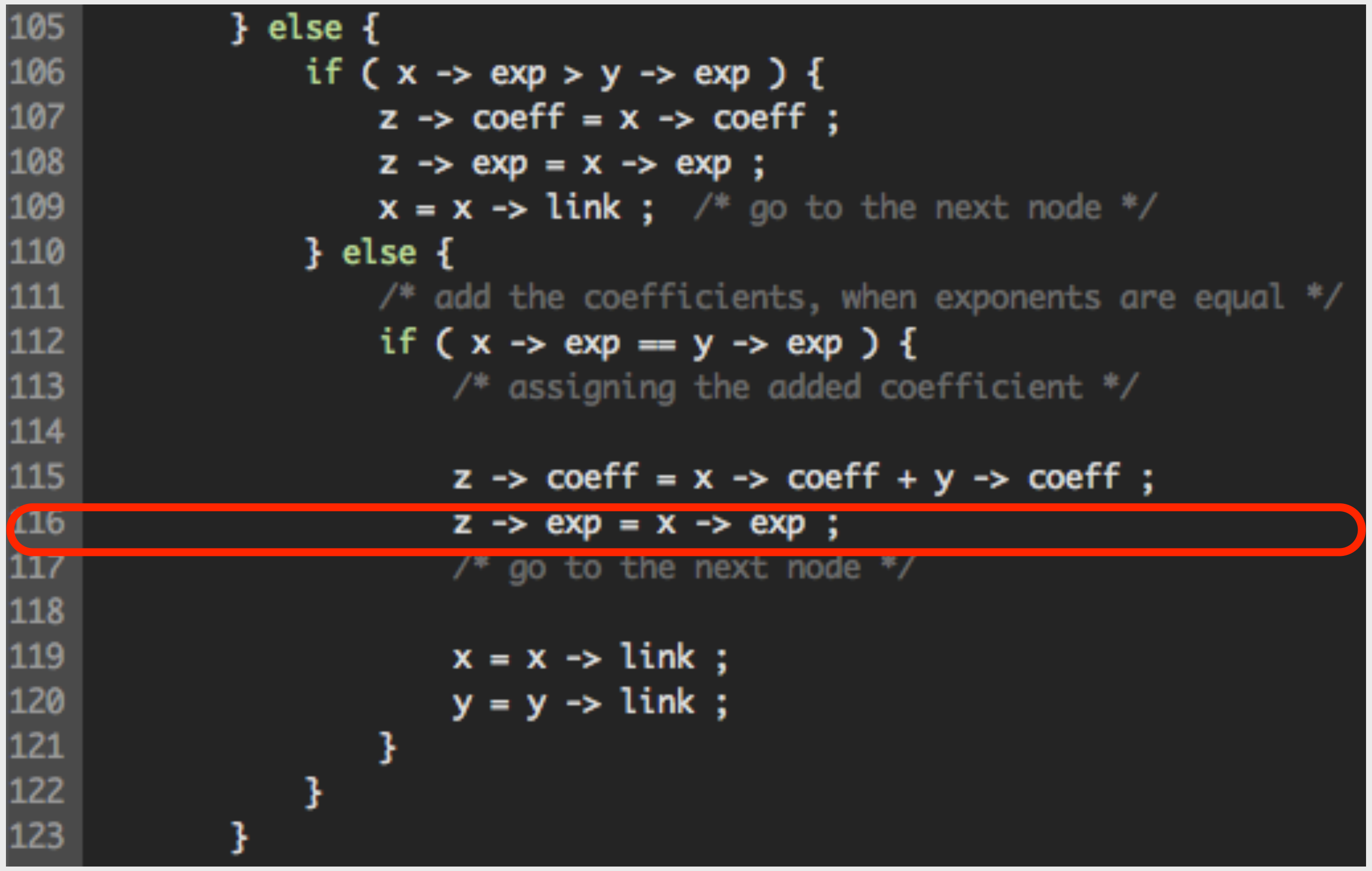


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

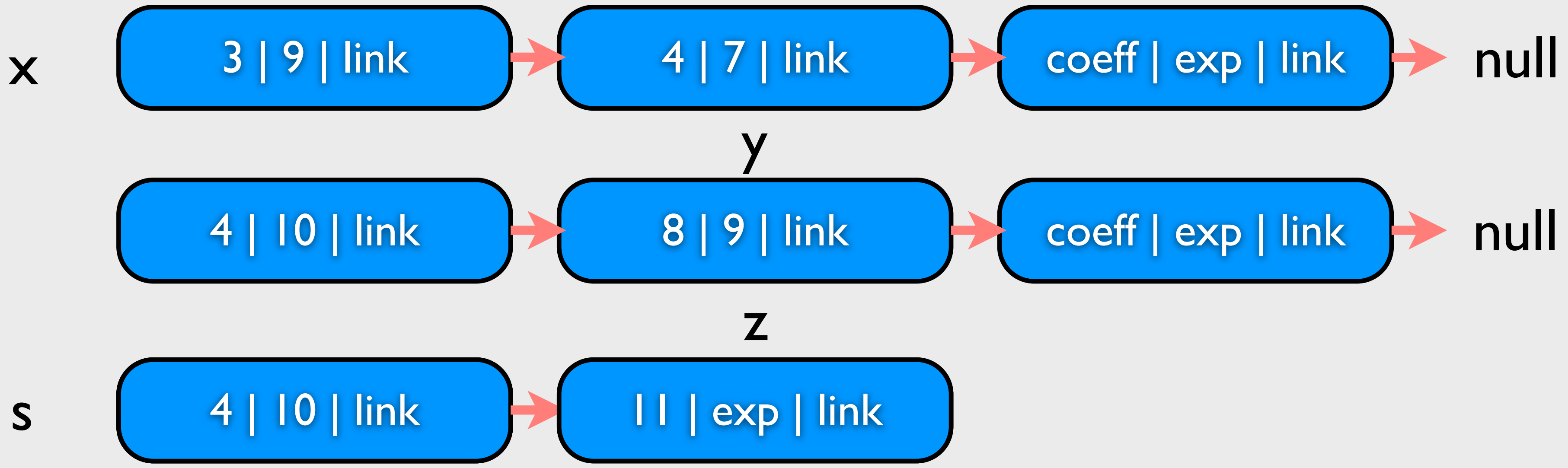


Example: poly/poly.c

- poly_add

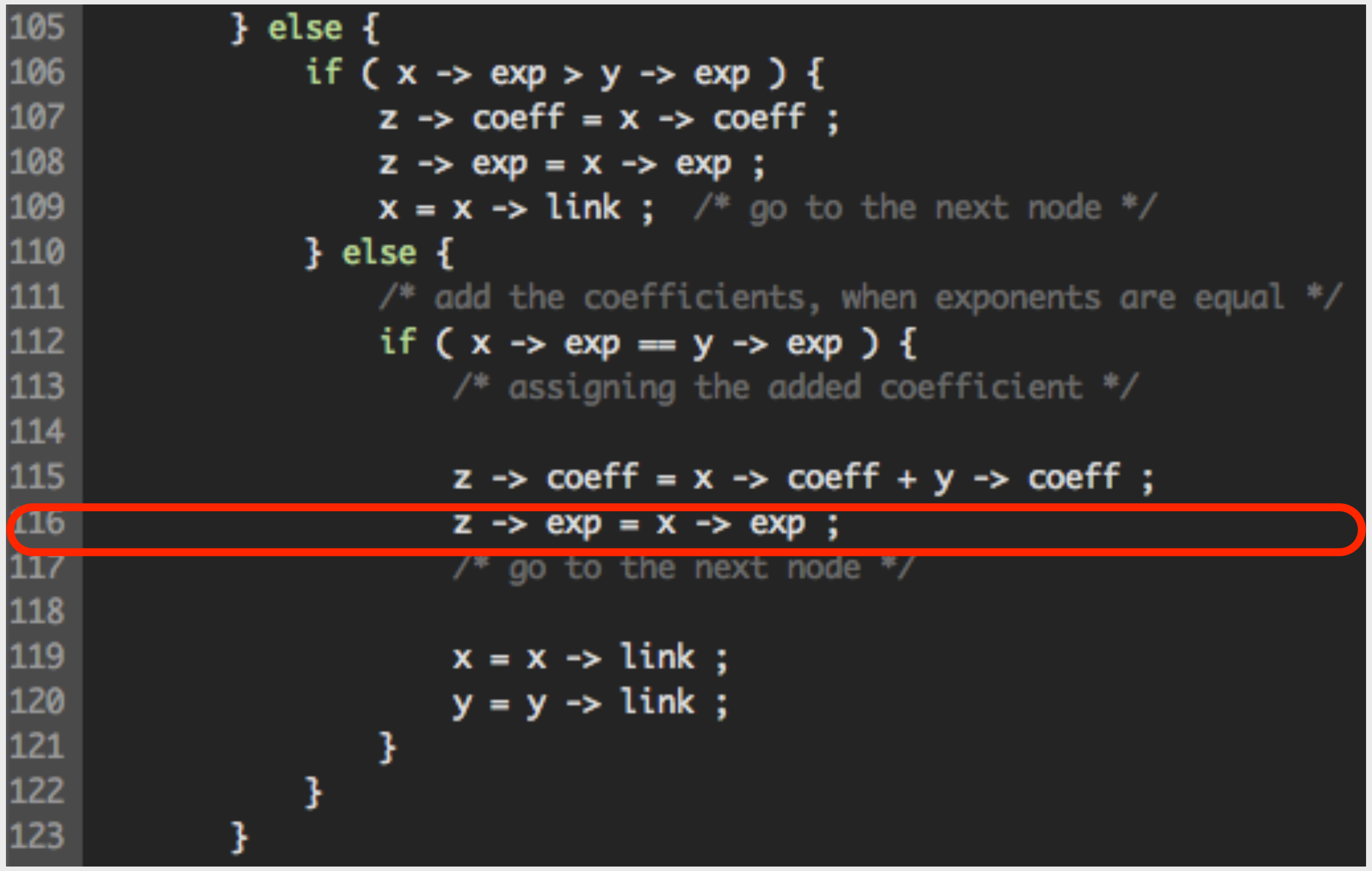


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

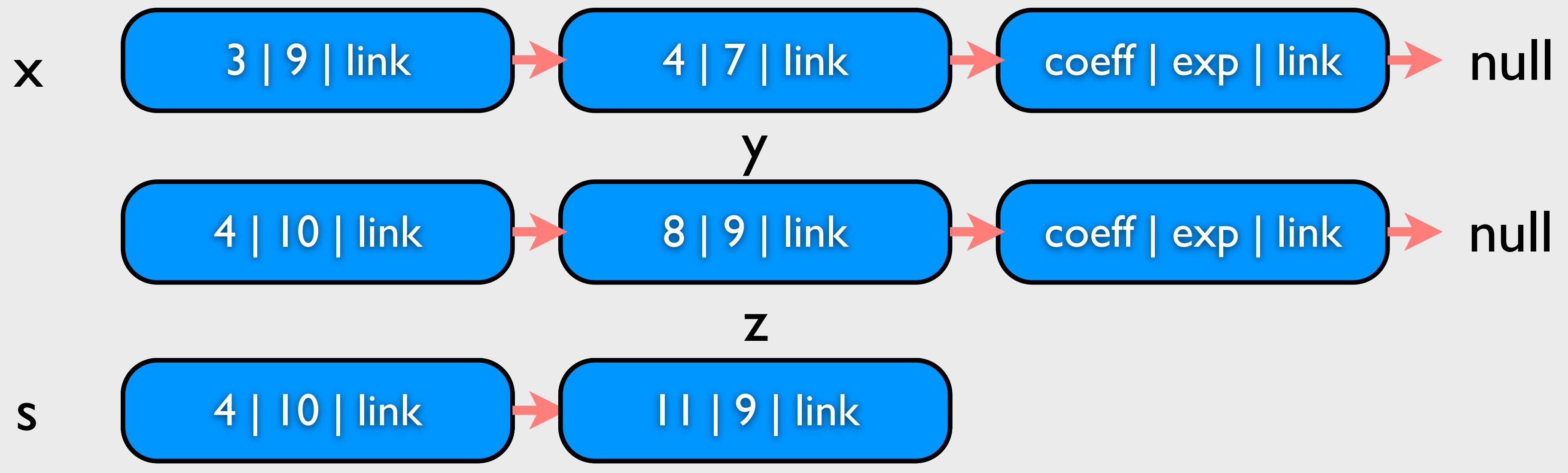


Example: poly/poly.c

- poly_add

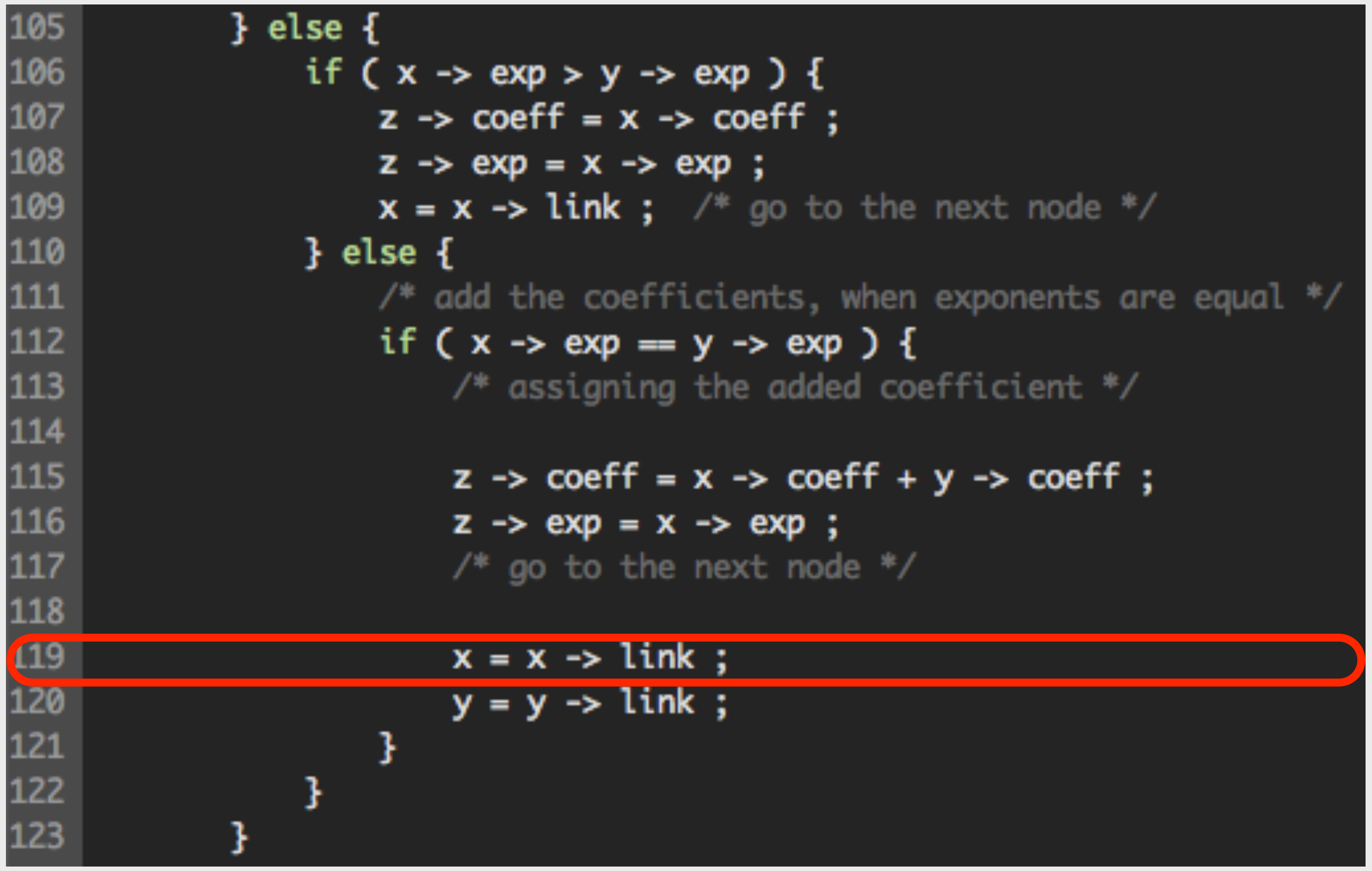


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

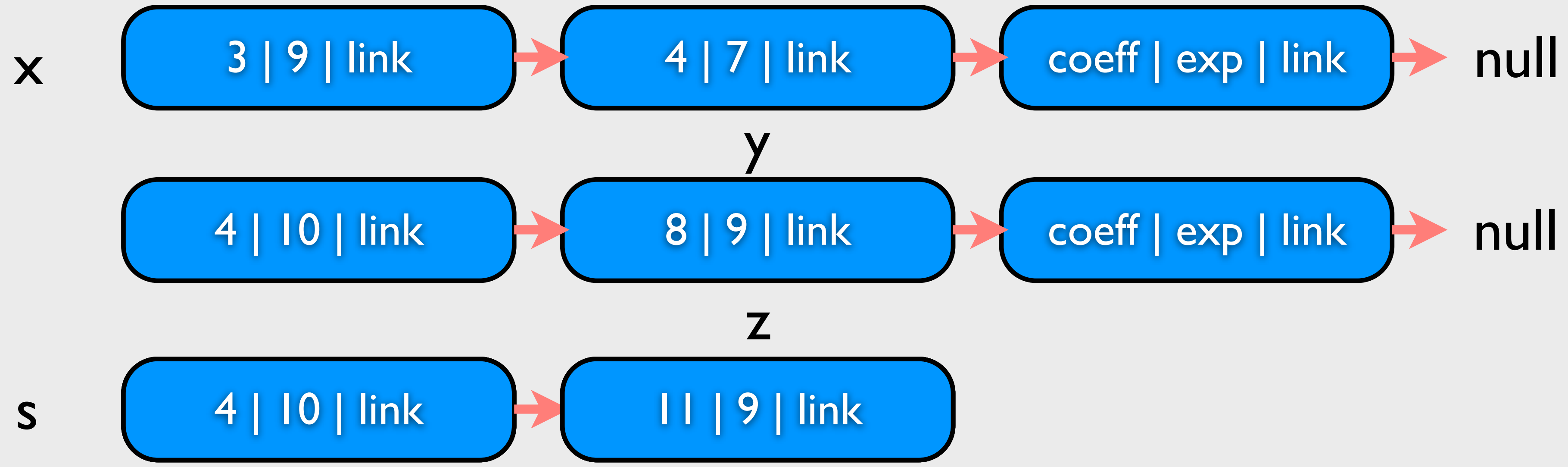


Example: poly/poly.c

- poly_add

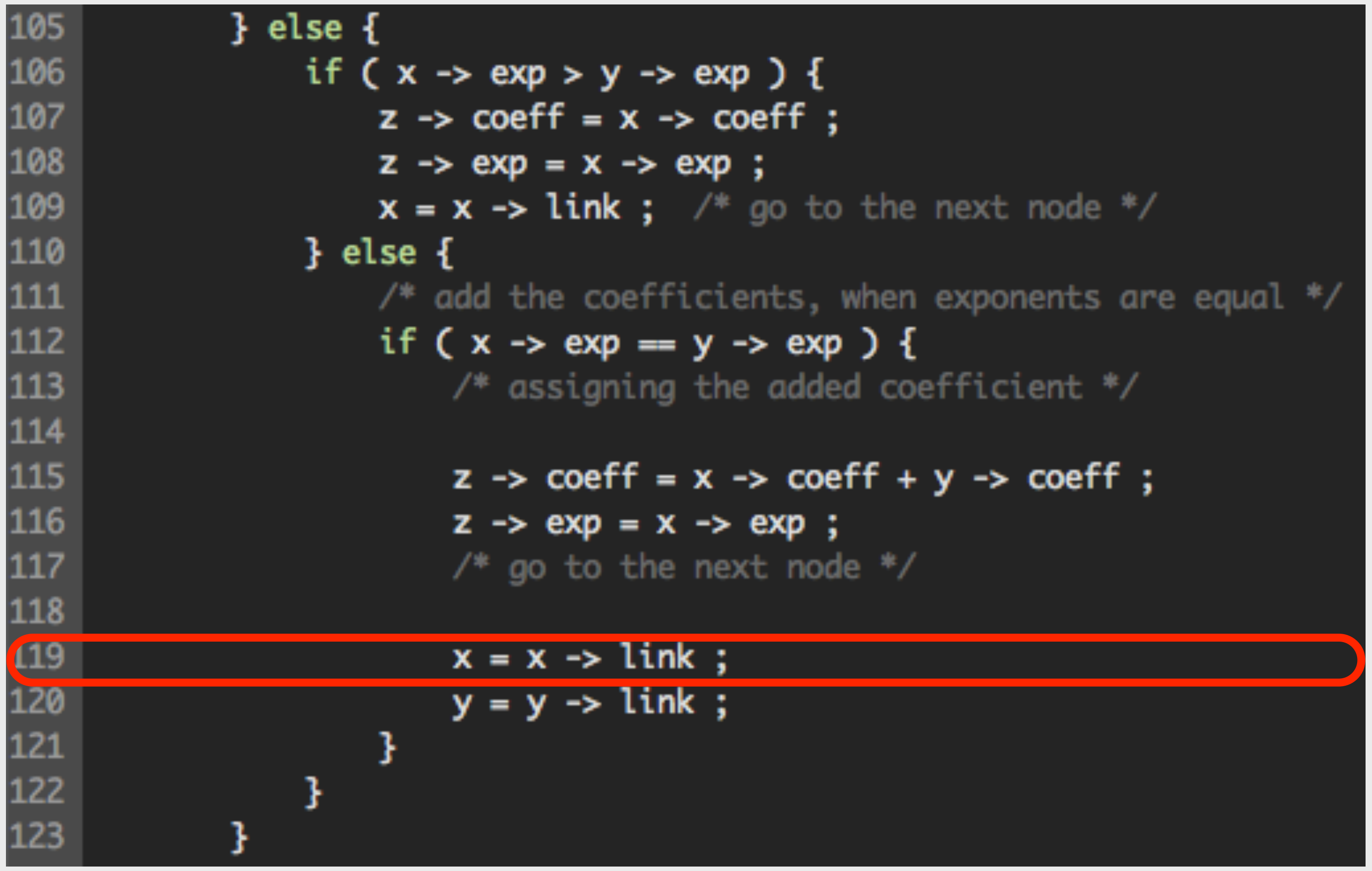


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

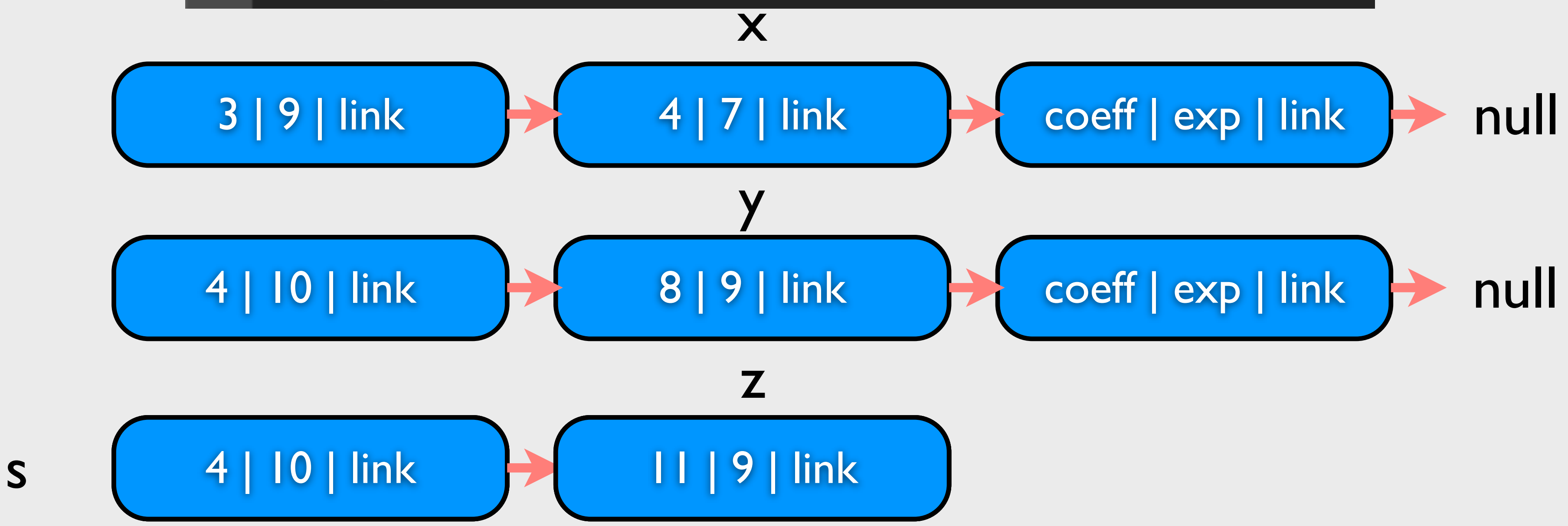


Example: poly/poly.c

- poly_add

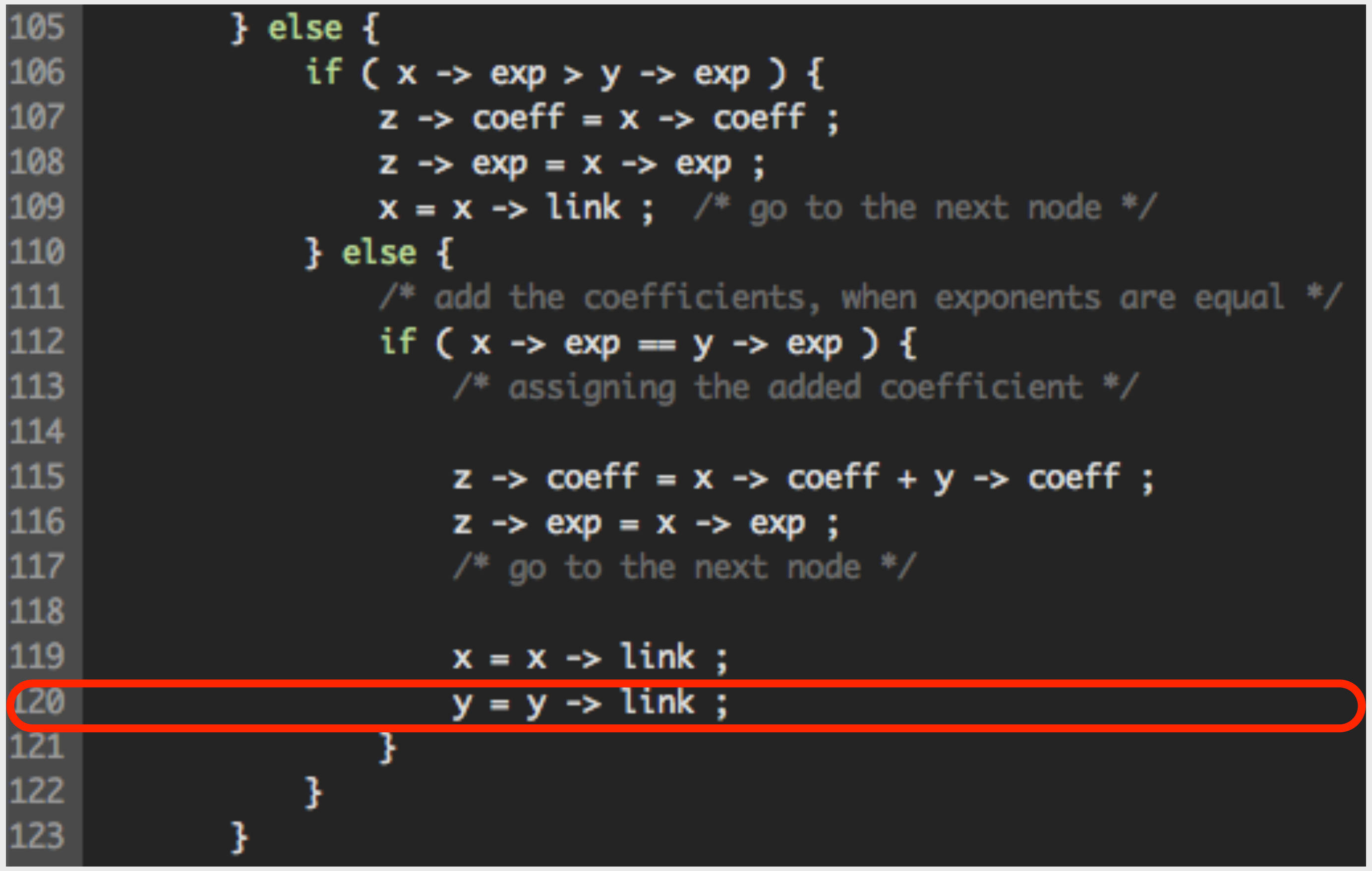


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

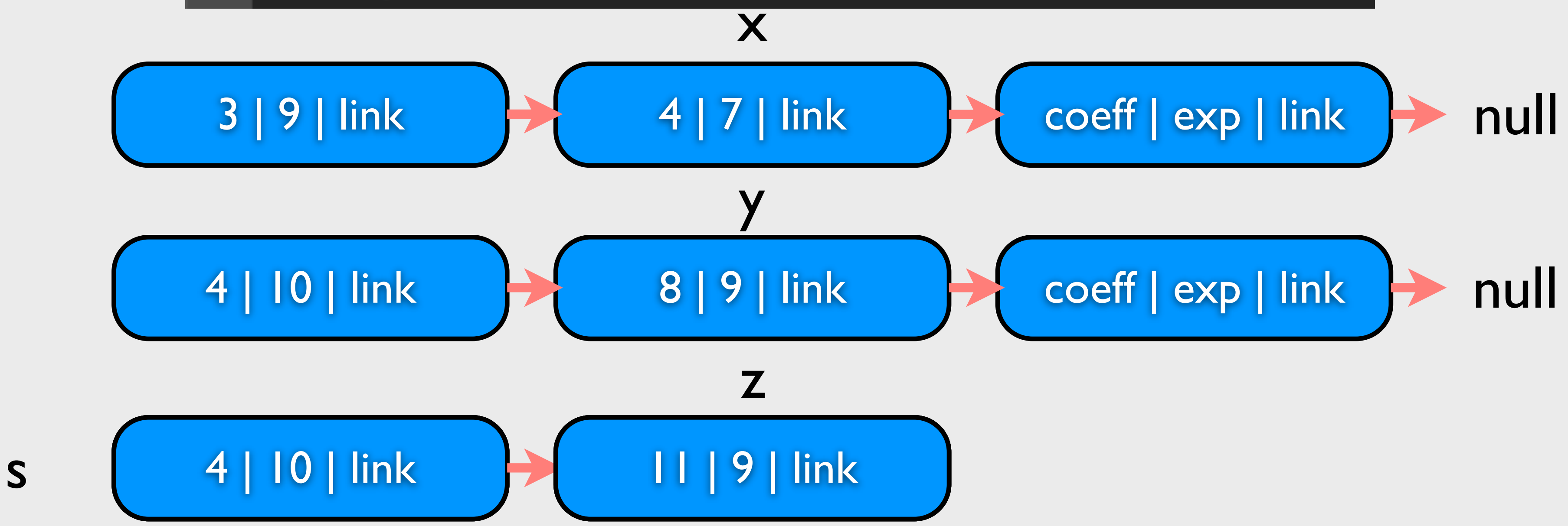


Example: poly/poly.c

- poly_add

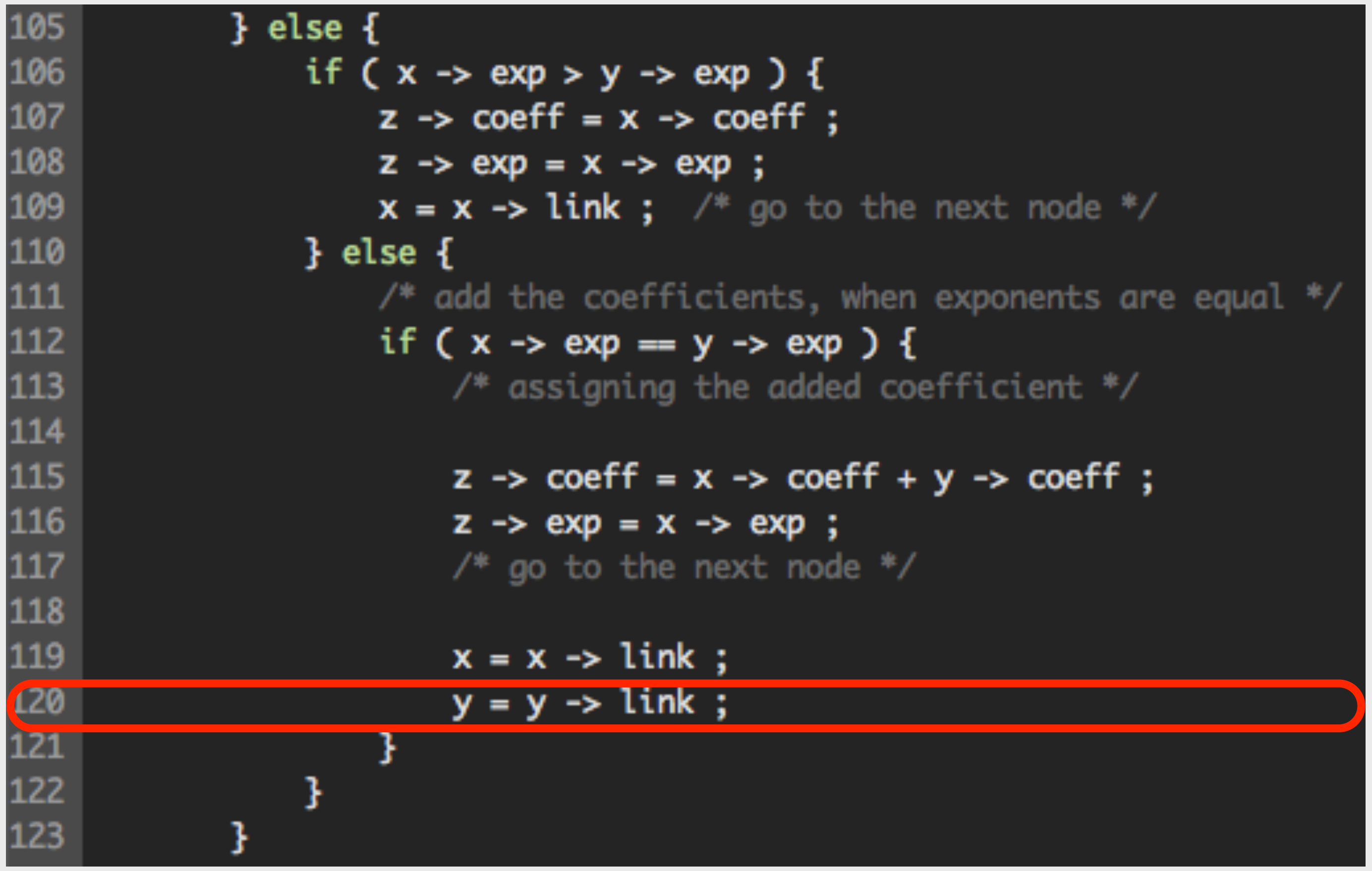


```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```

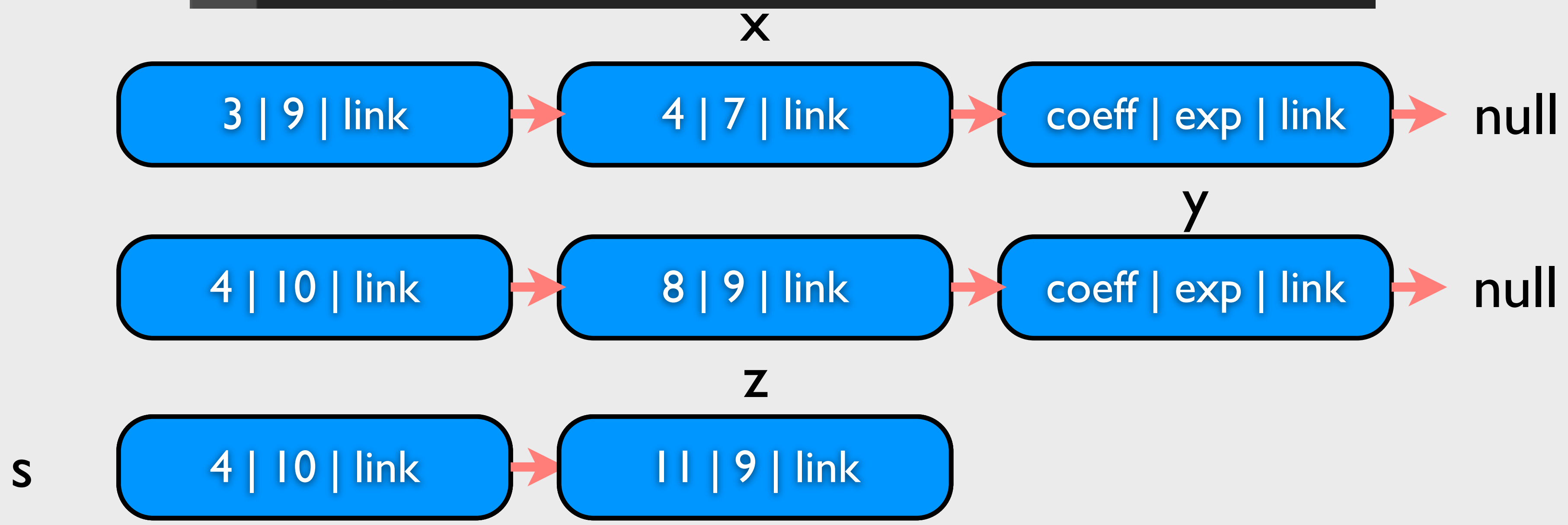


Example: poly/poly.c

- poly_add



```
        } else {
            if ( x -> exp > y -> exp ) {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> link ;  /* go to the next node */
            } else {
                /* add the coefficients, when exponents are equal */
                if ( x -> exp == y -> exp ) {
                    /* assigning the added coefficient */

                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* go to the next node */

                    x = x -> link ;
                    y = y -> link ;
                }
            }
        }
```


x
3 | 9 | link
4 | 7 | link
coeff | exp | link
null
y
4 | 10 | link
8 | 9 | link
coeff | exp | link
null
z
s
4 | 10 | link
11 | 9 | link