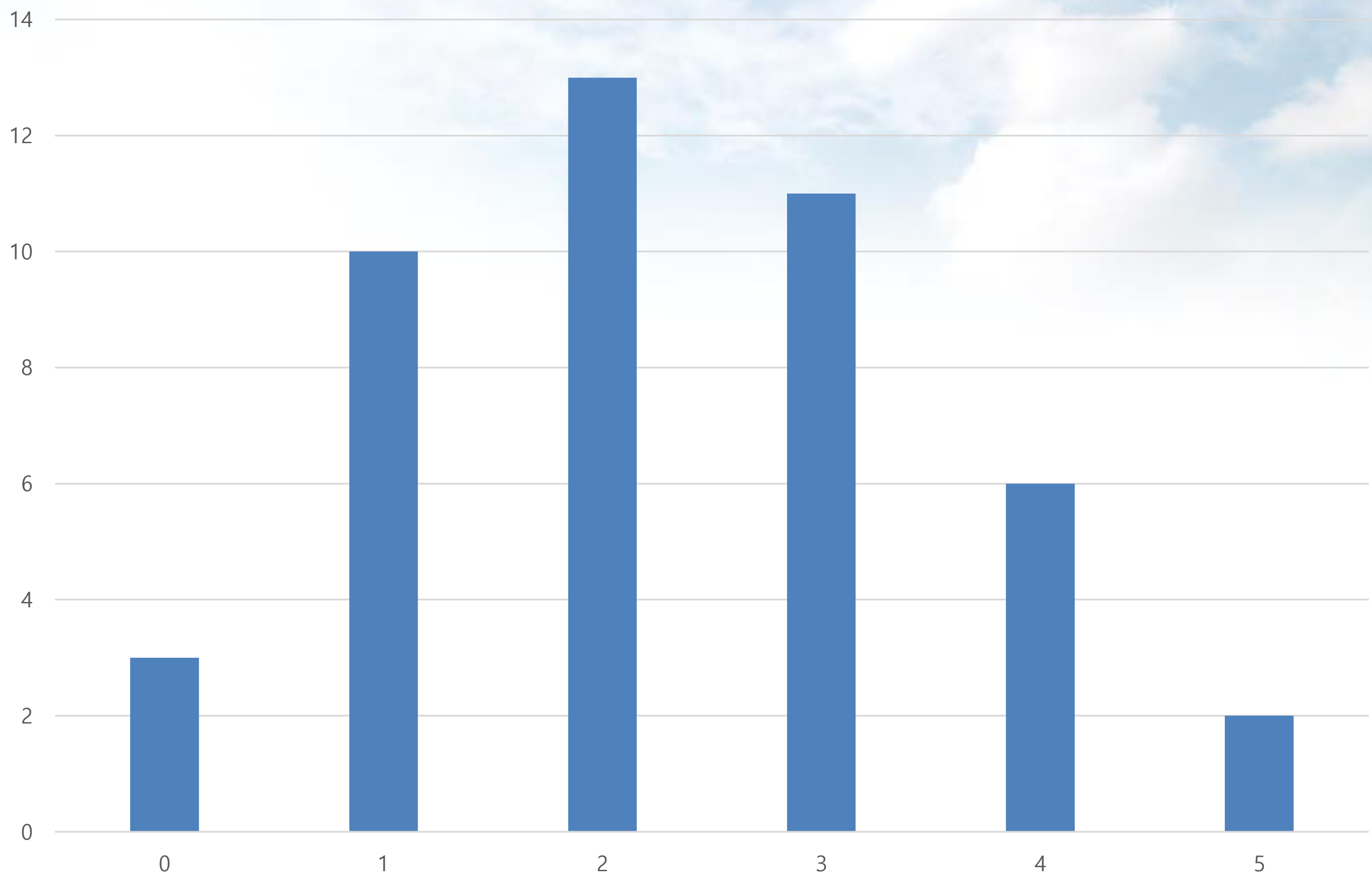
AAI2007 Introduction to Algorithms

Week 2: Order of Complexity, List

Instructor: Jinyoung Han (jinyounghan@skku.edu)

Level Test Result

Average: 2.28



In This Lecture

Outline

1. Order of Complexity
2. List

01. Order of Complexity

Algorithm Analysis

- Execution time measurement
  - Measures actual execution times of two algorithms
  - Requires actual implementation
  - Should use identical hardware

- Complexity analysis
  - (Roughly) Analyze without actual implementation
  - Count the number of operations during algorithm
  - Time complexity
  - Space complexity

01. Order of Complexity

Measurement

- An example code of measuring computing time

```
void main( void )
{
    clock_t start, finish;
    double  duration;
    start = clock();
     // algorithm...
     // ....
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("%f seconds.₩n", duration);
}
```

01. Order of Complexity

Complexity Analysis

- How many algorithms can you imagine for solving a problem?
  - Many!
- Among them, what algorithm should we choose?
  - An efficient one!
  - So algorithm analysis is important!
- How to analyze algorithm (from an efficiency perspective)?
  - Complexity analysis!
    - Without actual implementation, roughly we can compare two algorithms
    - Independent for hardware or software environment

01. Order of Complexity

Complexity Analysis

- Computing time complexity
  - Count the number of operations
    - Basic operations: comparison, assignment, arithmetic, etc.
  - Not measure actual execution time!

- Represented by a time complexity function -> T(n)
  - A function of n (input size)
  - Roughly estimate time for running algorithm

01. Order of Complexity

An Example

- Problem: sum n for n times
  - Let's count the number of operations
  - Let's not consider the for loop control operations

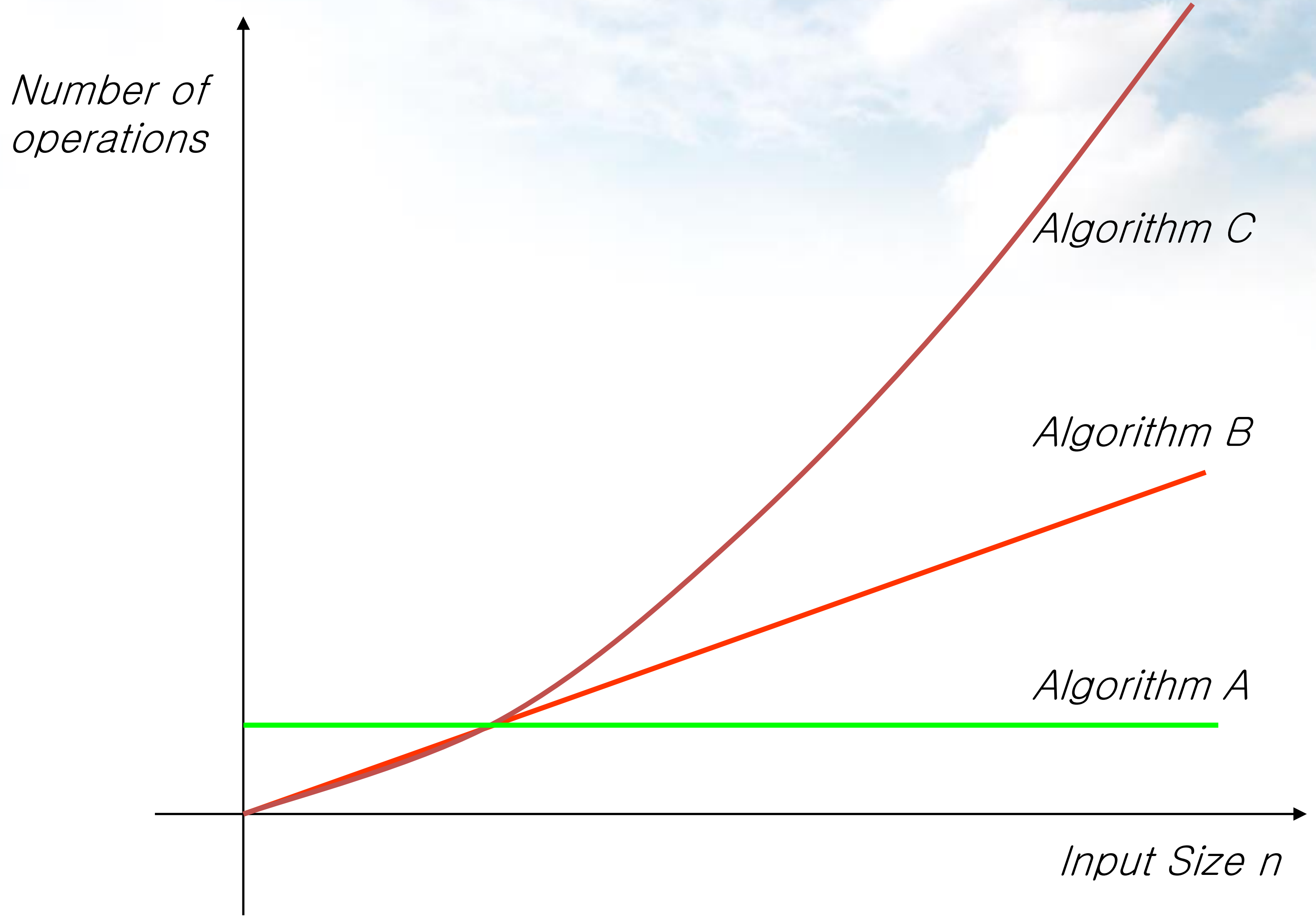
Algorithm A	Algorithm B	Algorithm C

sum ←n*n;	sum ← 0;
	for i ← 1 to n do
	  sum ←sum + n;
		sum ← 0;
		for i←1 to n do
		  for j←1 to n do
		    sum ←sum + 1;

	Algorithm A	Algorithm B	Algorithm C
Assignment	1	n + 1	n*n + 1
Addition		n	n*n
Multiplication	1		
Division			
Total	2	2n + 1	$2n^2 + 1$

01. Order of Complexity

An Example



01. Order of Complexity

Another Example

- By analyzing the code, we can roughly calculate the time complexity for the given algorithm

```
ArrayMax(A,n)

  tmp ← A[0];                          1 assignment
  for i←1 to n-1 do                    Exclude the for operation
        if tmp < A[i] then             n-1 comparisons
              tmp ← A[i];              n-1 assignments (at most)
  return tmp;                          1 return

                                        total = 2n (at most)
```

01. Order of Complexity

Big O Notation

- If n is large, the highest exponent part actually matters, ignoring other parts
  - E.g., n = 1000, T(n) = 1,001,001, first part accounts for about 99%

n=1000

$T(n)=\underbrace{n^2}_{99\%}+\underbrace{n+1}_{1\%}$ *Input size: n*

  - So, typically it is enough to consider the part that most affect

01. Order of Complexity

Big O Notation

- Big O Definition: (Asymptotic Upper Bound)
  - For given f(n) and g(n),

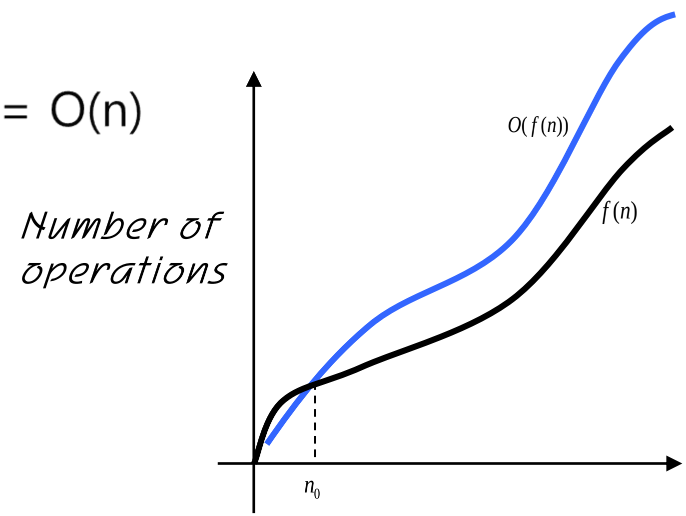
    for all $n \geq n_0$, if there exist two constants c and n_0

    satisfying $|f(n)| \leq c|g(n)|$

    then $f(n)=O(g(n))$

- Big O represents the upper bound
  - E.g., if $n \geq 5$, $2n+1 < 10n$ -> $2n+1 = O(n)$

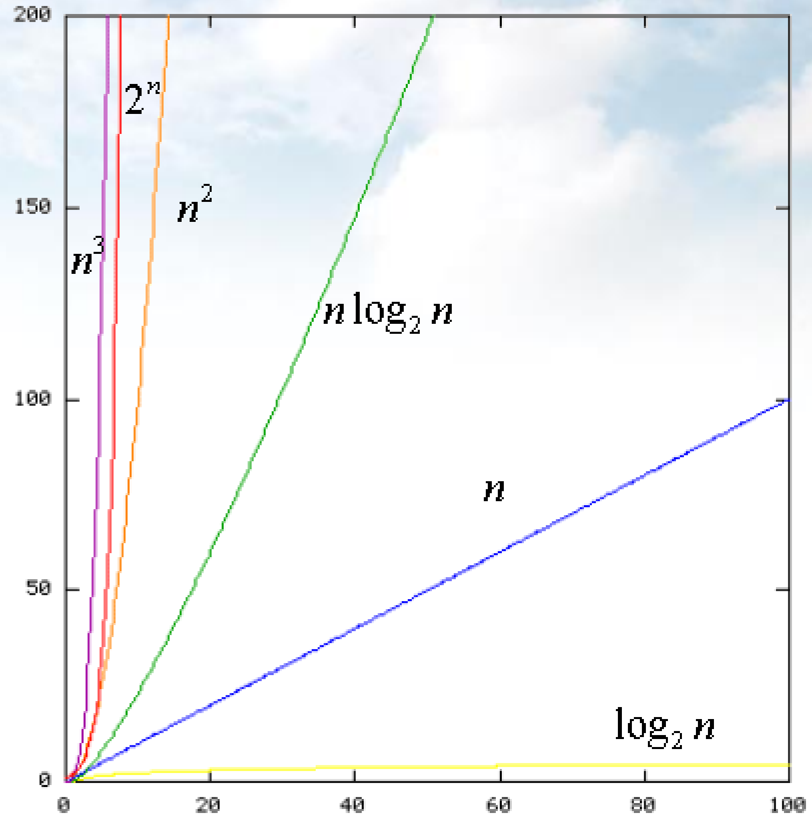
if $n_0 = 2$, $c=2$,
for $n \geq 2$, $2n+1 \leq 2n^2$
→ $O(n^2)$



01. Order of Complexity

Big O Notation

- O(1) : constant
- O(logn) : log
- O(n) : linear
- O(nlogn) : log-linear
- O(n^2) : quadratic
- O(n^3) : cubic
- O(2^n) : exponent
- O(n!) : factorial



01. Order of Complexity

Comparisons

Complexity	n					
	1	2	4	8	16	32
1	1	1	1	1	1	1
$logn$	0	1	2	3	4	5
n	1	2	4	8	16	32
$nlogn$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40326	20922789888000	26313×10^{33}

01. Order of Complexity

Comparisons

	A	B	C	D	E
	$100n$	$10nlog_2n$	$5n^2$	n^3	2^n
10	10^{-3} sec	$1.5*10^{-3}$ sec	$5*10^{-4}$ sec	10^{-3} sec	10^{-3} sec
100	10^{-2} sec	0.03 sec	$5*10^{-2}$ sec	1 sec	$4*10^{14}$ cent
1,000	10^{-1} sec	0.45 sec	5 sec	1.6 min	***
10,000	1 sec	6.1 sec	8.3 min	11.57 d	***
100,000	10 sec	1.5 min	13.8 hour	31.7 y	***

01. Order of Complexity

Big Ω Notation

- Big Omega Definition: (Asymptotic Lower Bound)
  - For given f(n) and g(n),

    for all $n \geq n_0$, if there exist two constants c and n_0

    satisfying $|f(n)| \geq c|g(n)|$

    then $f(n)= \Omega(g(n))$

- Big Omega represents the lower bound
  - E.g., if $n \geq 1$, $2n+1 \geq 10n$ -> $2n+1 = \Omega(n)$

01. Order of Complexity

Big θ Notation

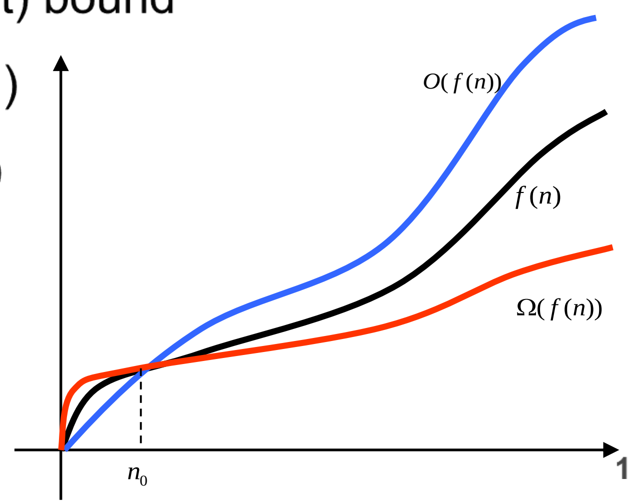
- Big Theta Definition: (Asymptotic Tight Bound)
  - For given f(n) and g(n),

    for all $n \geq n_0$, if there exist three constants c_1, $c_{2,}$, and n_0

    satisfying $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$

    then $f(n) = \theta(g(n))$

- Big Theta represents the lower and upper (tight) bound
  - $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$ -> $f(n)=\theta(n)$
  - E.g., if $n \geq 1$, $n \leq 2n+1 \leq 3n$ -> $2n+1 = \theta(n)$



List

02. List

Definition

- An abstract data type (ADT) that represents a countable number of ordered values, where the same value may occur more than once
- Examples
  - Days (Monday, Tuesday, …)
  - Alphabet (A, B, …)
  - Card (Ace, 2, 3, …)
  - Phone numbers

$$L = (item_0, item_1, \ldots, item_{n-1})$$

02. List

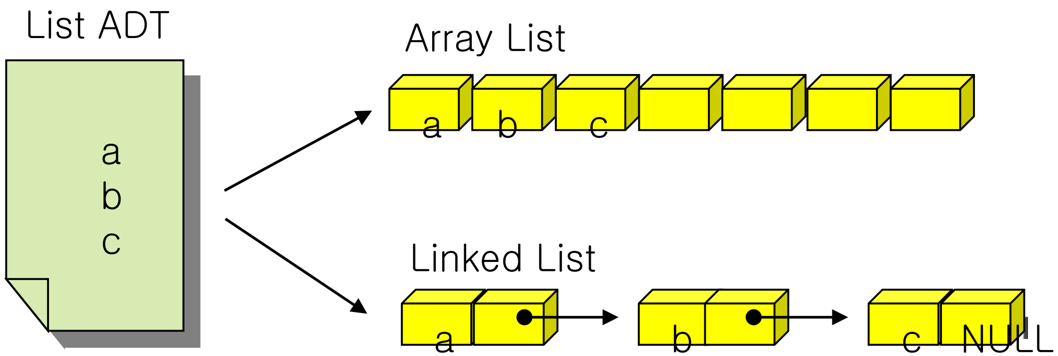
ADT

- Object:
  - A sequence with n values
- Operations:
  - add_last(list, item)
  - add_first(list, item)
  - add(list, pos, item)
  - delete(list, pos)
  - clear(list)
  - replace(list, pos, item)
  - is_in_list(list, item)
  - get_entry(list, pos)
  - get_length(list)
  - is_empty(list)
  - is_full(list)
  - display(list)

02. List

List Implementation

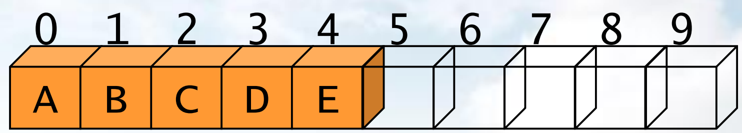
- Array List
  - Simple
  - Insertion and deletion may not be easy
  - Limited capacity
- Linked List
  - Difficult
  - Efficient in insertion and deletion
  - No limitation in capacity



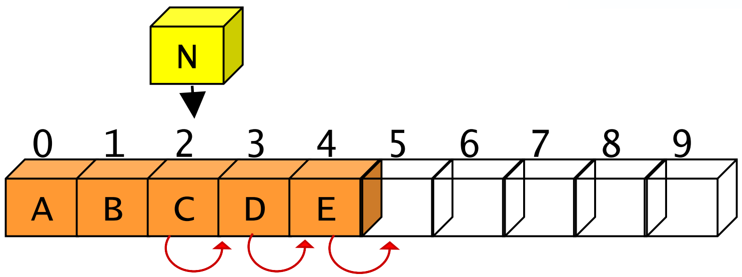
02. List

Array List

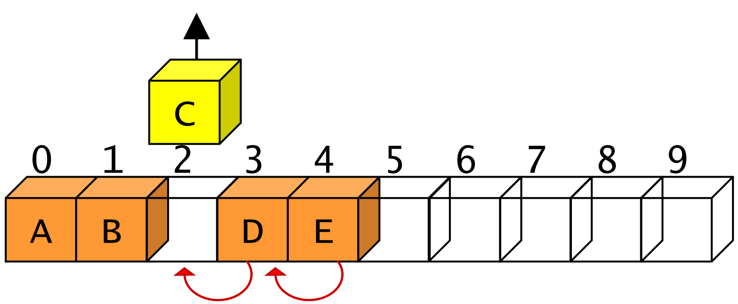
- Store data in one-dimensional array sequentially
  - L = (A, B, C, D, E)



- Insertion

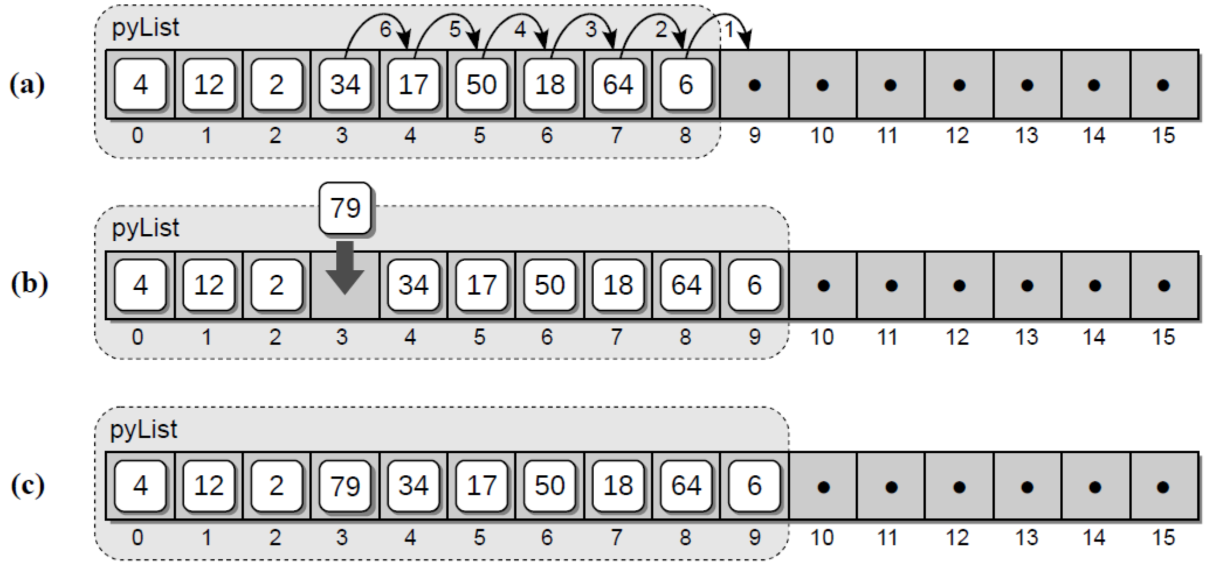


- Deletion



Array List: Insertion

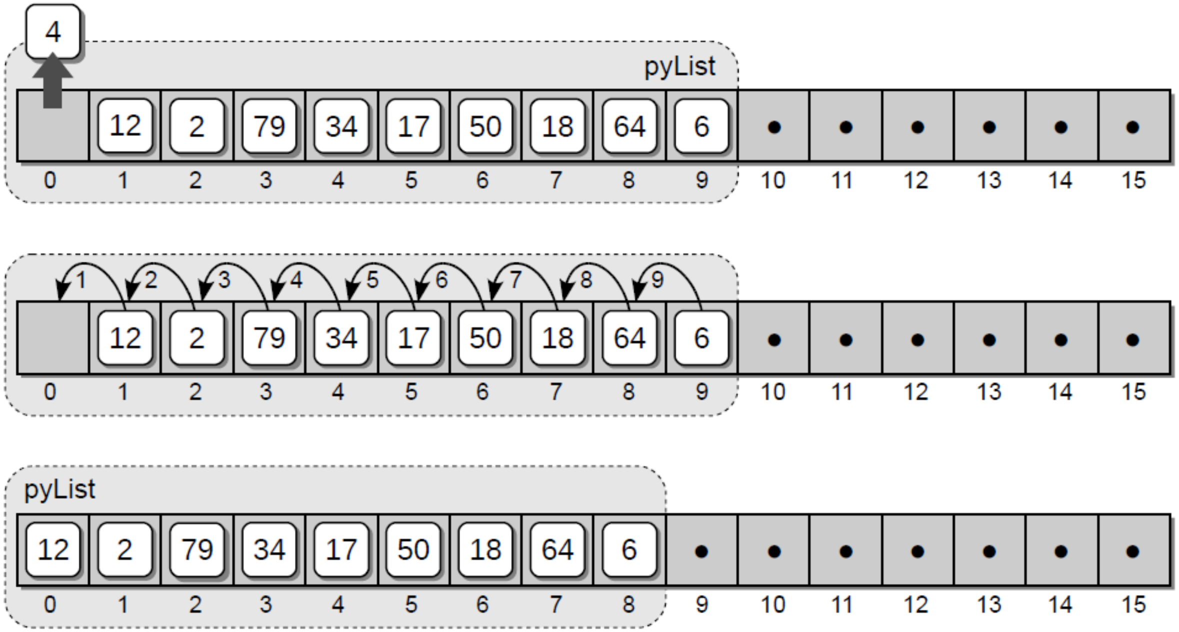
- Inserting an item into a list
  - (a) the array elements are shifted to the right one at a time, traversing from right to left
  - (b) the new value is then inserted into the array at the given position
  - (c) the result after inserting the item



Array List: Deletion

- Removing an item from a list
  - (a) a copy of the item is saved
  - (b) the array elements are shifted to the left one at a time, traversing left to right
  - (c) the size of the list is decremented by one

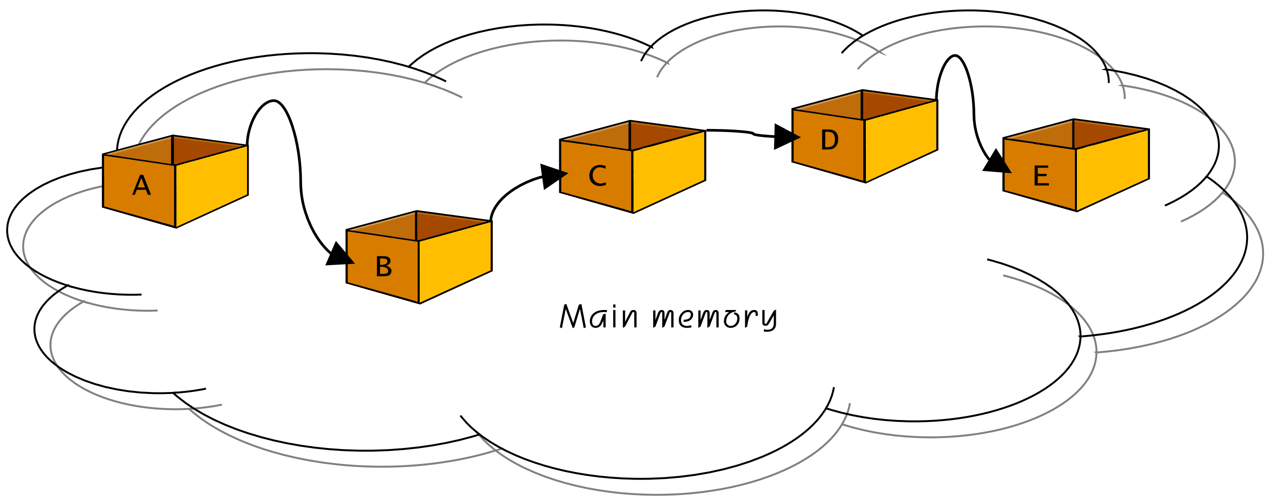
(a)
(b)
(c)



02. List

Linked Representation

- Linked Representation
  - Node: Data & Link
    - Data: data value
    - Link: next node
- The sequence of link may not be identical to that in physical memory

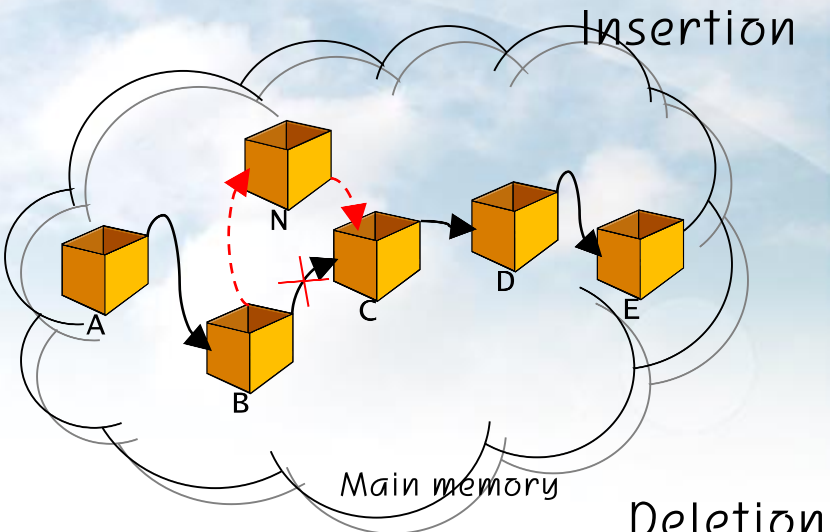


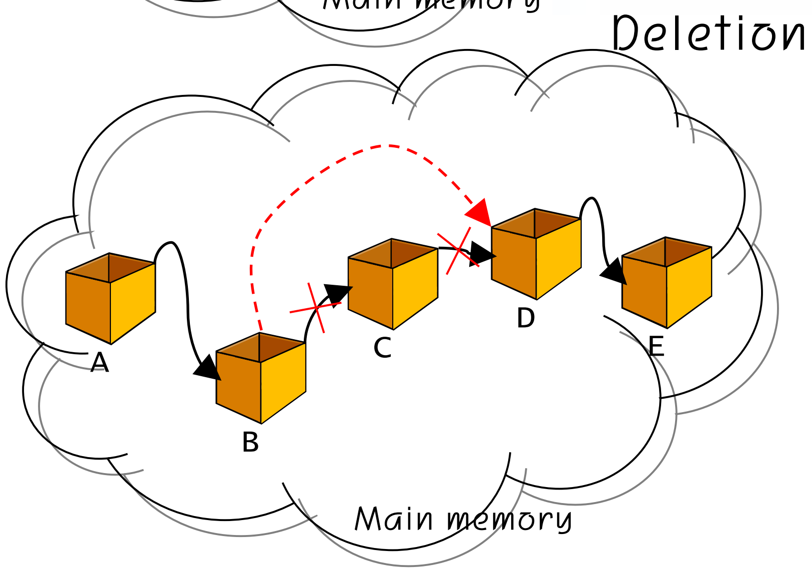
02. List

Linked Representation

- Pros
  - Insertion/deletion are easy
  - Need not continuous memory space
  - No space limitation

- Cons
  - Difficult to implement
  - Possible errors

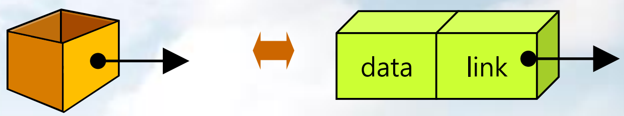




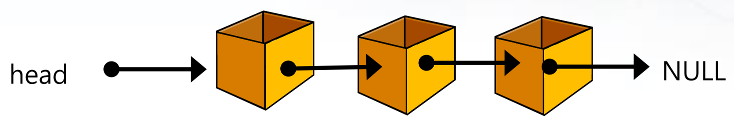
Structure

- Node = (data, link)

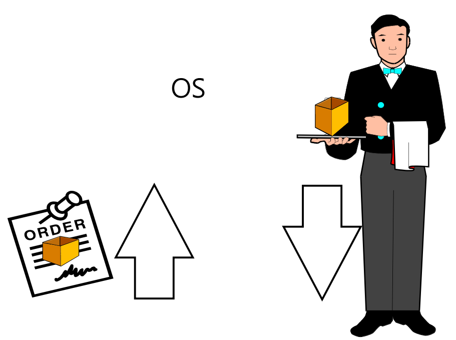
```
class ListNode :
  def __init__( self, data ) :
    self.data = data
    self.next = None
```

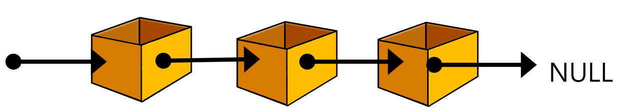


- “Head” indicates the first node



- Node creation
  - a = ListNode(11); a.next = b
  - b = ListNode(52); b.next = c
  - c = ListNode(18)

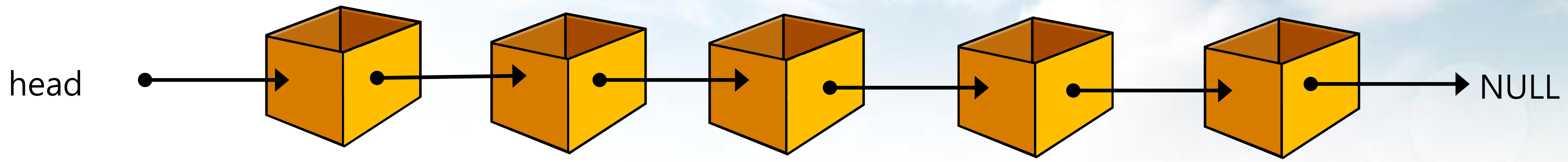




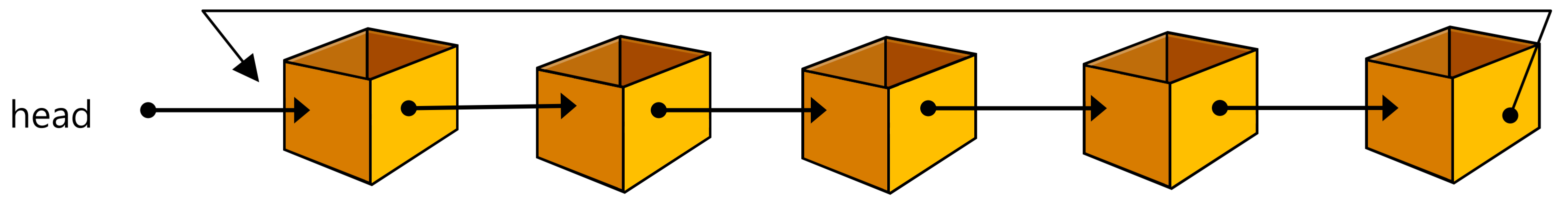
02. List

Linked List Types

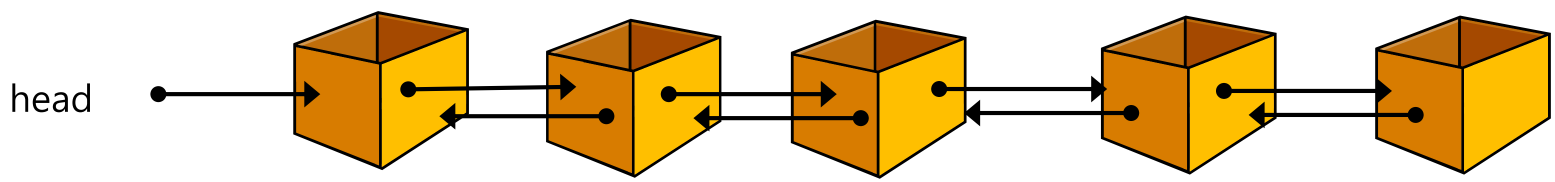
- Singly linked list



- Circular linked list

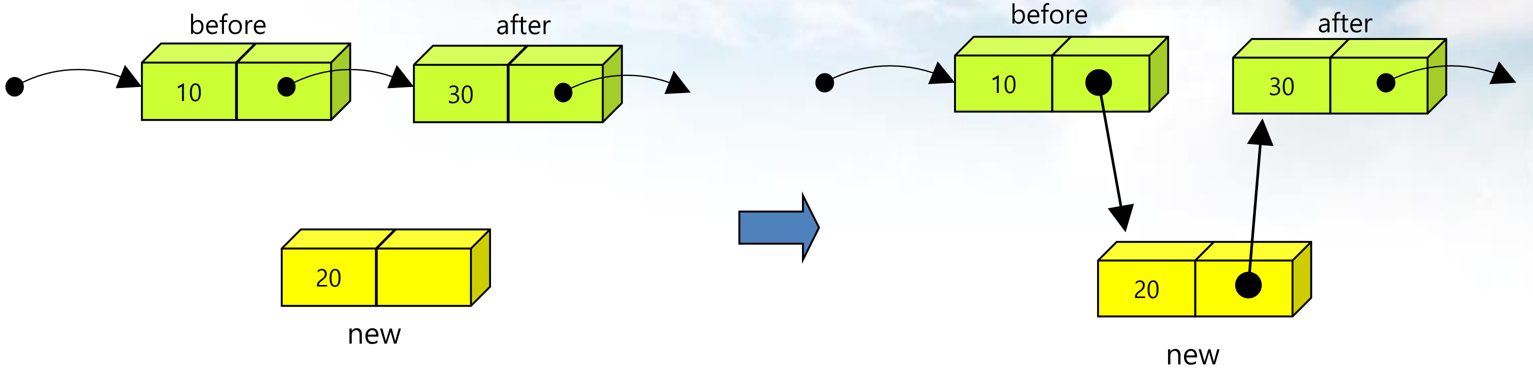


- Doubly linked list



Insertion

- Basic idea

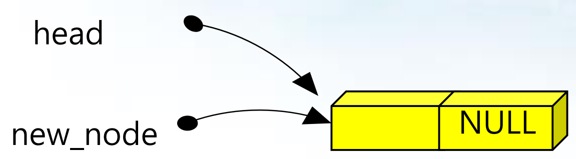



```
insert_node(L, before, new)

if   L = NULL
then  L←new
else   new.link←before.link
       before.link←new
```

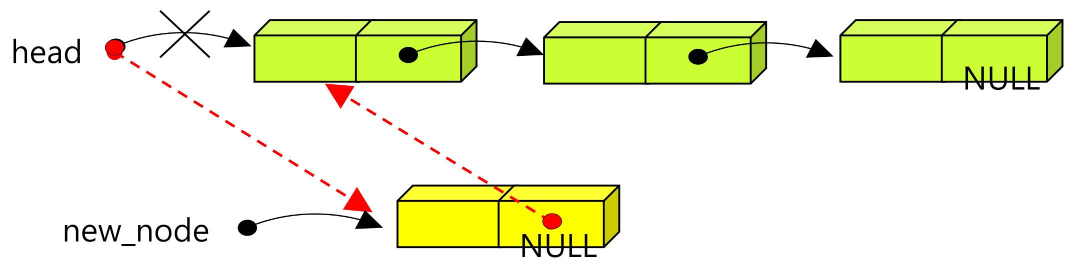
02. List

Insertion

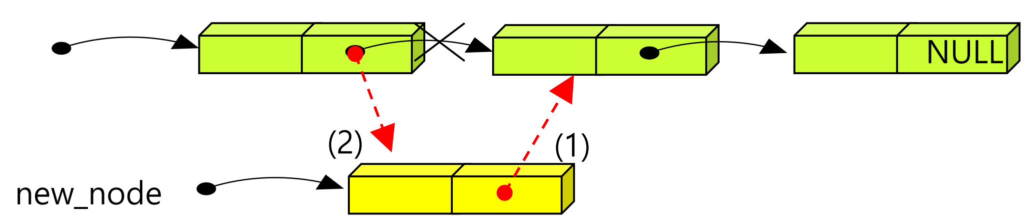
- 3 cases
  - Empty list



  - Add to first

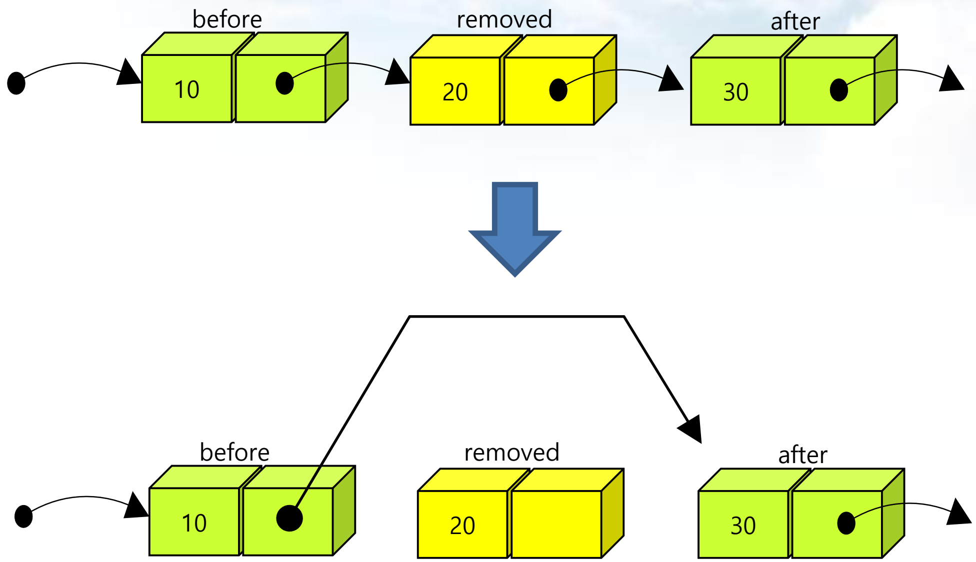


  - General case



Deletion

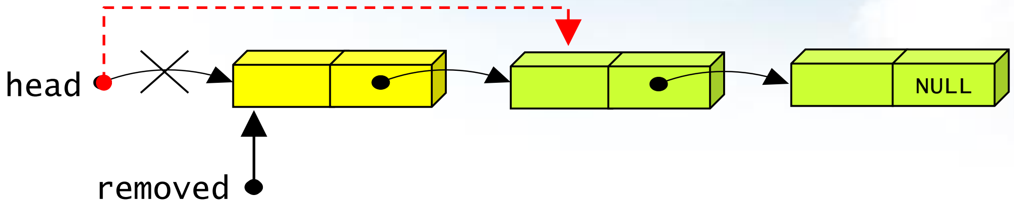
- Basic idea



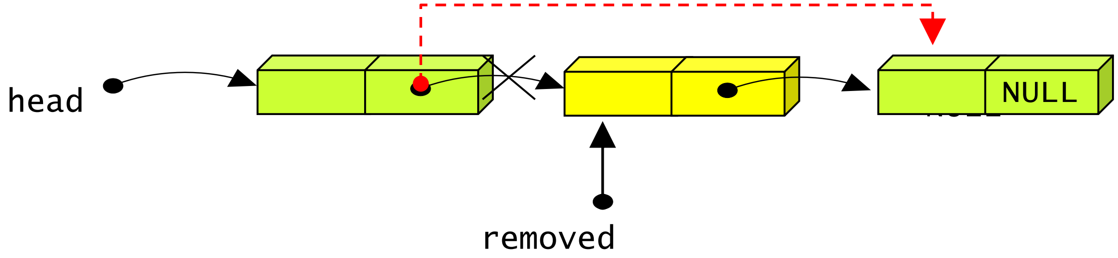

```
remove_node(L, before)

if L ≠ NULL
    then   before.link←removed.link
```

Deletion

- 2 cases
  - First node

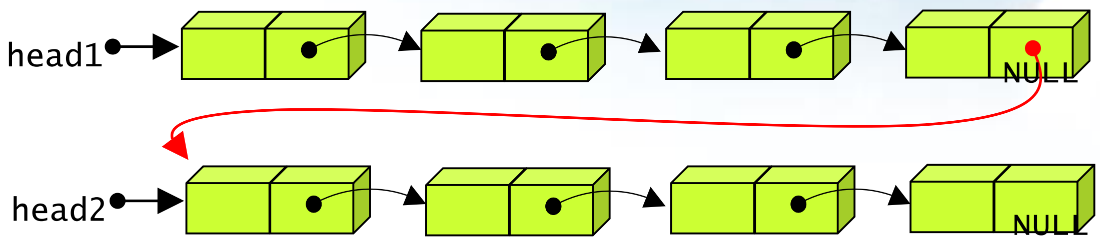


  - General case



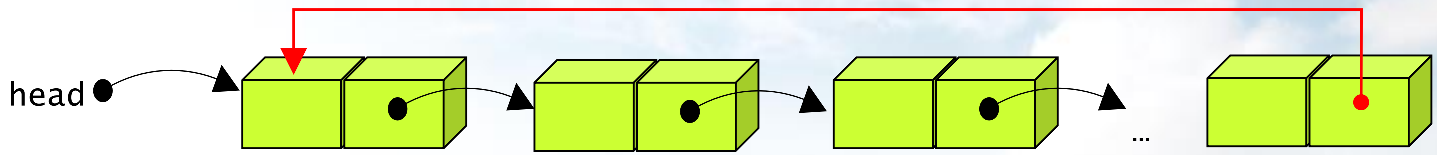
Merge Two Lists

- An example of list operations: merging two lists

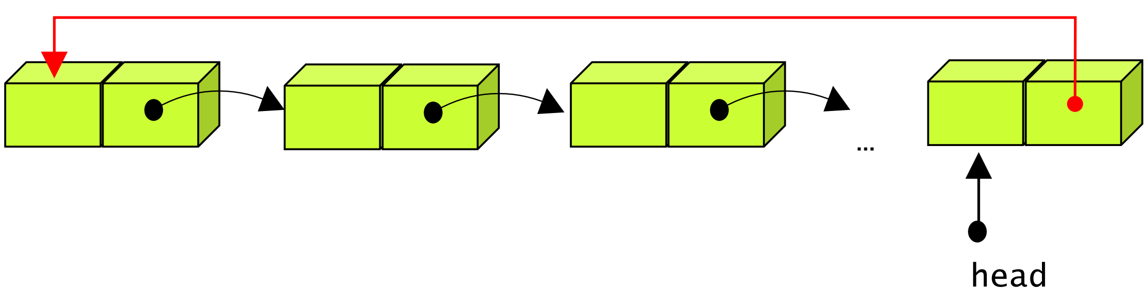


Circular Linked List

- The last node points the first node

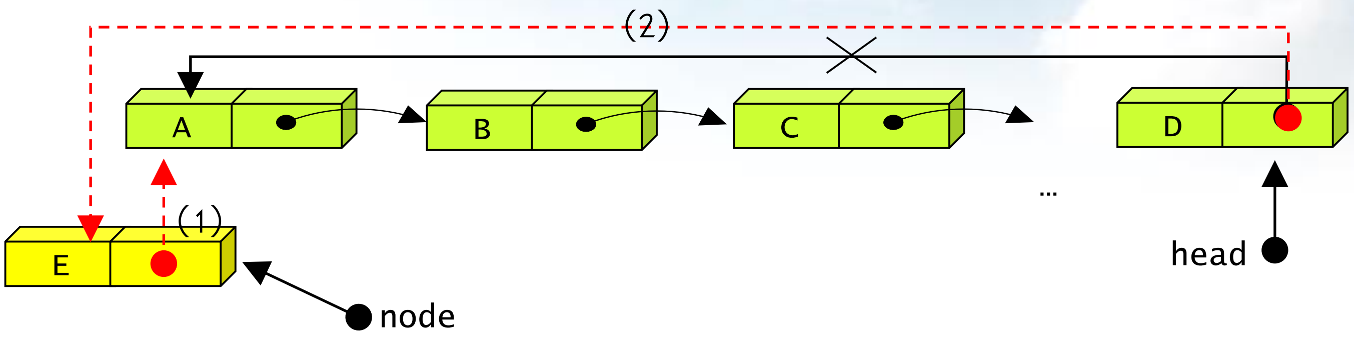


- We can traverse the list starting from any node
- Easier than single linked list in insertion / deletion
- If head points to the last node, “addFirst” and “addLast” can be easily implemented compared to single linked list

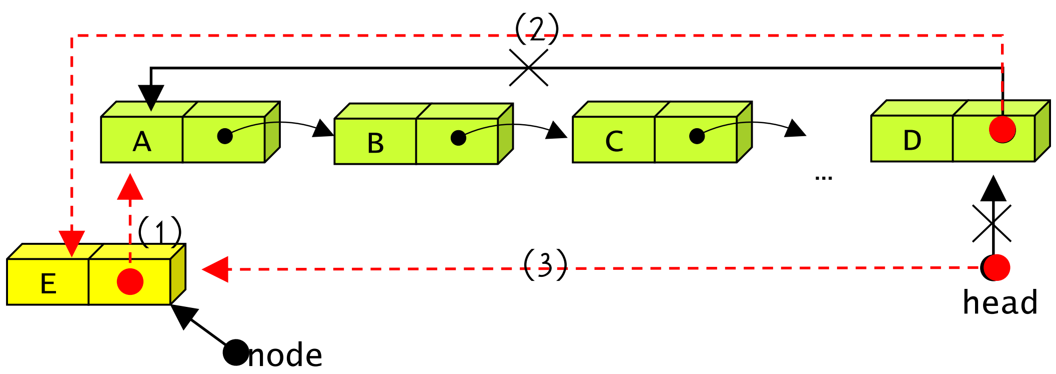


Insertion

- Two cases
  - Insertion in the first



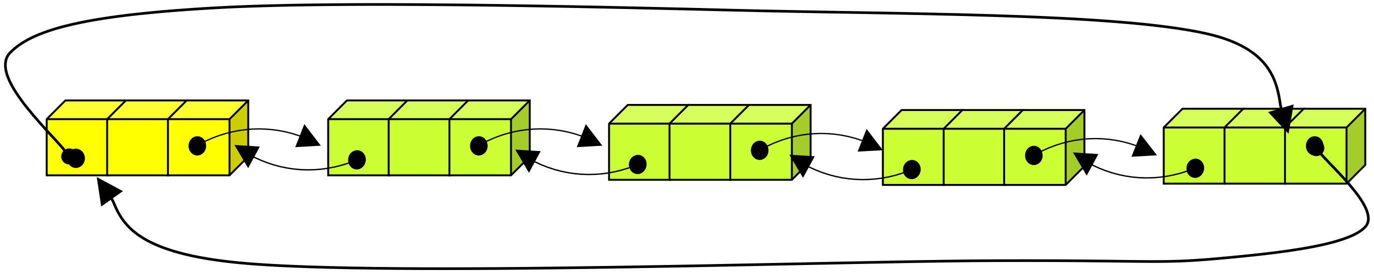
  - Insertion in the last



Double Linked List

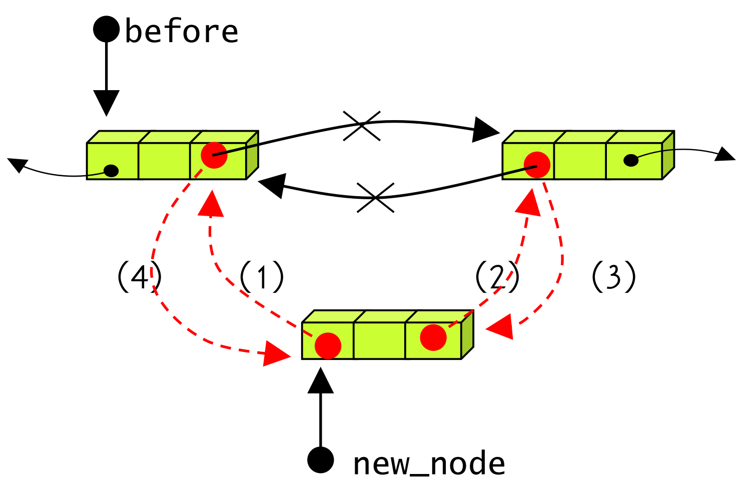
- Double linked list
  - Node has two links for previous and next data
  - Link => bidirectional

- Practically, “double linked list + circular linked list” type is widely used



Insertion

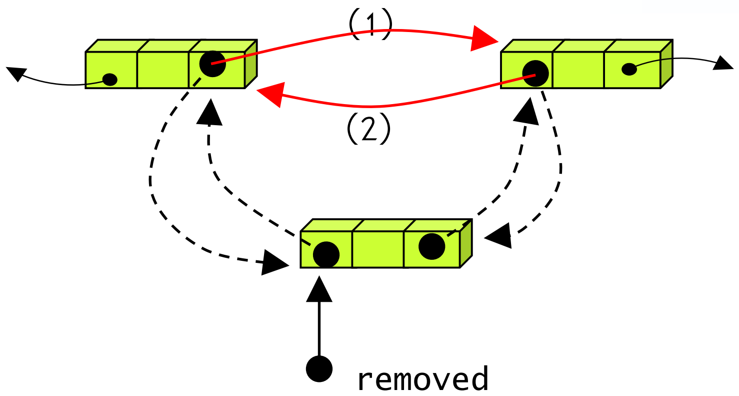
- new_node.llink = before; (1)
- new_node.rlink = before.rlink; (2)
- before.rlink.llink = new_node; (3)
- before.rlink = new_node; (4)



02. List

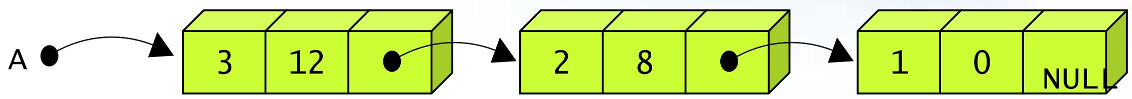
Deletion

- removed.llink.rlink = removed.rlink (1)
- removed.rlink.llink = removed.llink (2)



Application: Polynomial

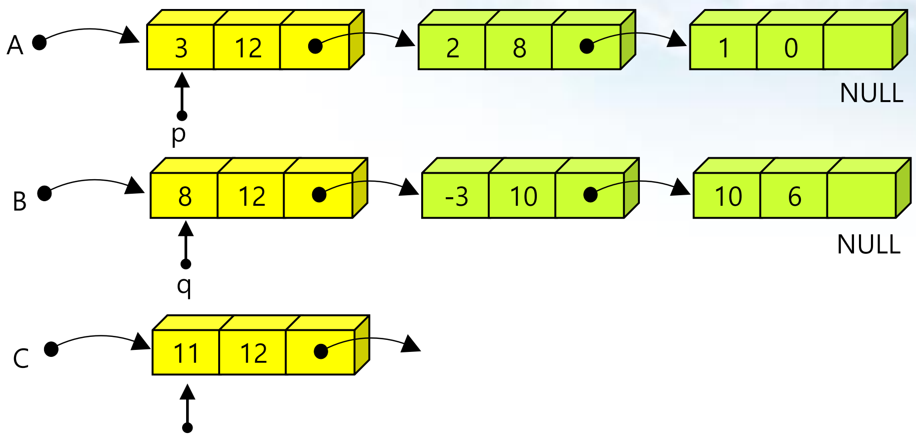
- A polynomial (in one variable) can be expressed as a list
  - $A=3x^{12}+2x^{8}+1$



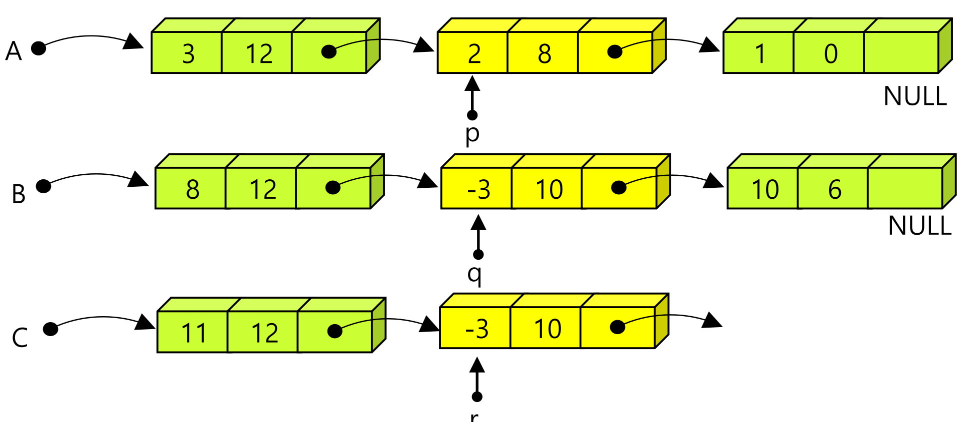

```
class _PolyTermNode( object ):
  def __init__( self, degree, coefficient ):
      self.degree = degree
      self.coefficient = coefficient
      self.next = None
```

Polynomial Addition

① p.expon == q.expon

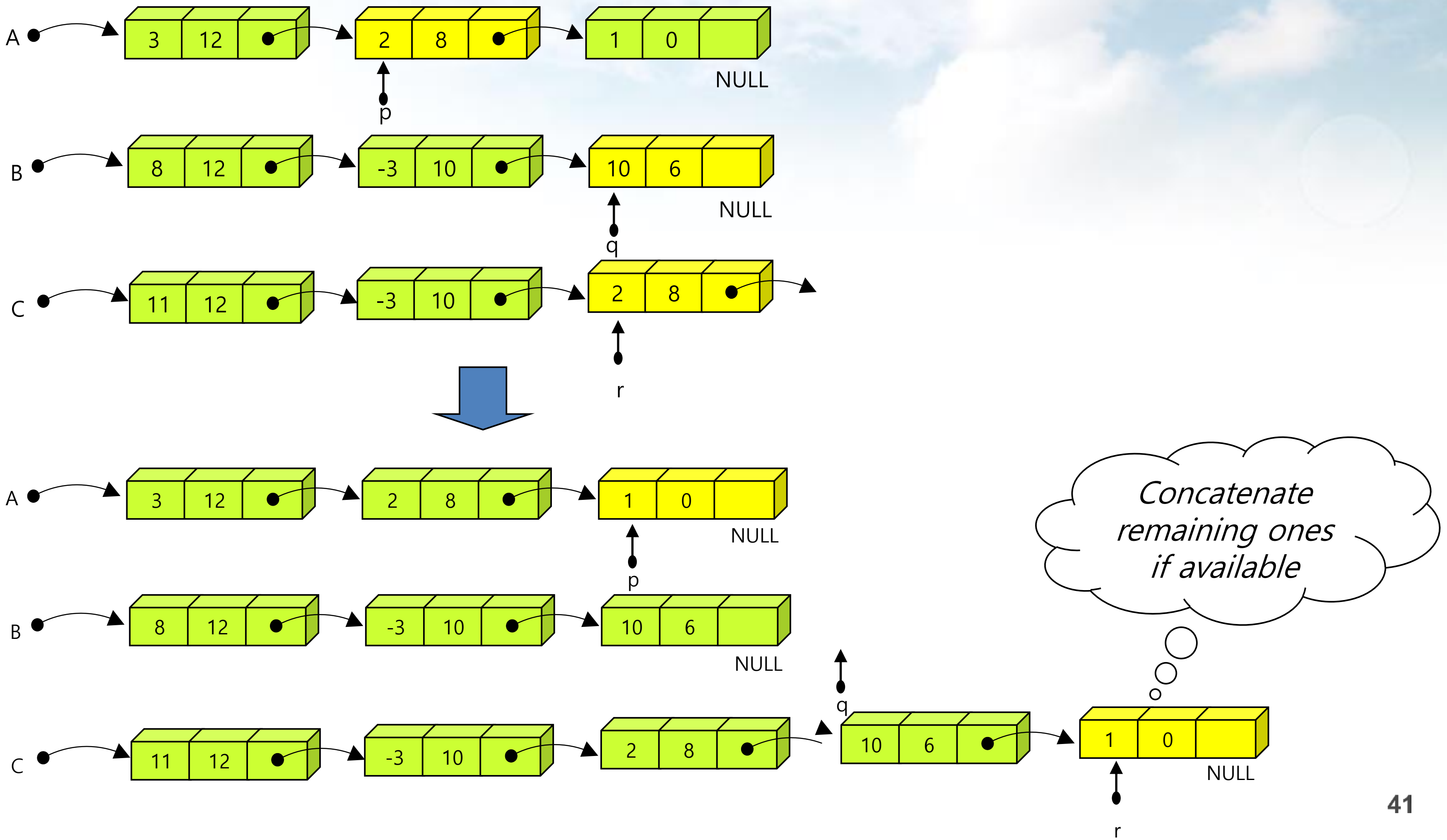


② p.expon < q.expon



Polynomial Addition

③ p.expon > q.expon



What You Need to Know

Summary

- Algorithm analysis
  - Time complexity
- List
  - Array list
  - Linked list
    - Linked representations
    - Singly linked list
    - Circular linked list
    - Double linked list
  - Application: Polynomials

Thanks

Week 2: Order of Complexity, List
Instructor: Jinyoung Han (jinyounghan@skku.edu)