Linked List Data Structures

Data Structures for Computer Professionals

Patiwet Wuttisarnwattana, Ph.D.

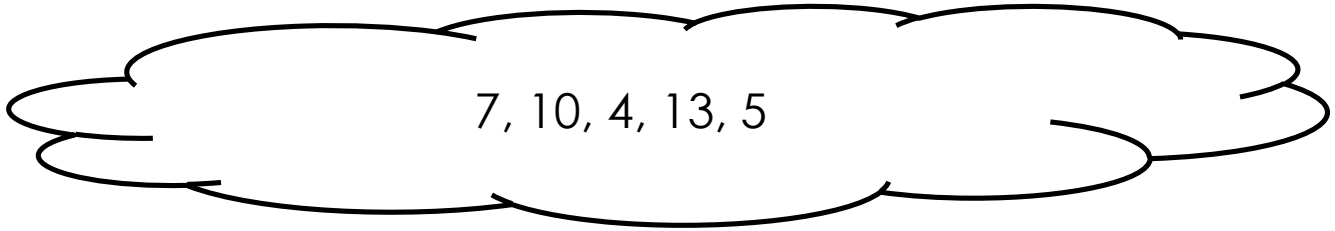
patiwet@eng.cmu.ac.th

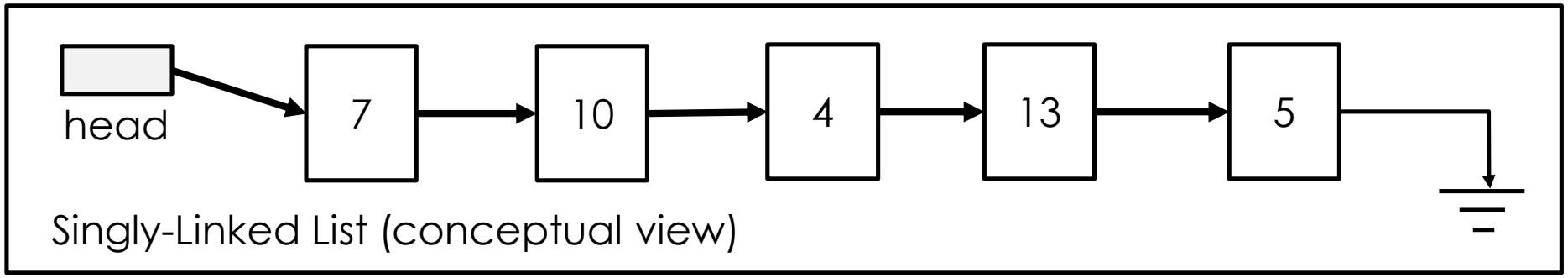
Computer Engineering, Chiang Mai University

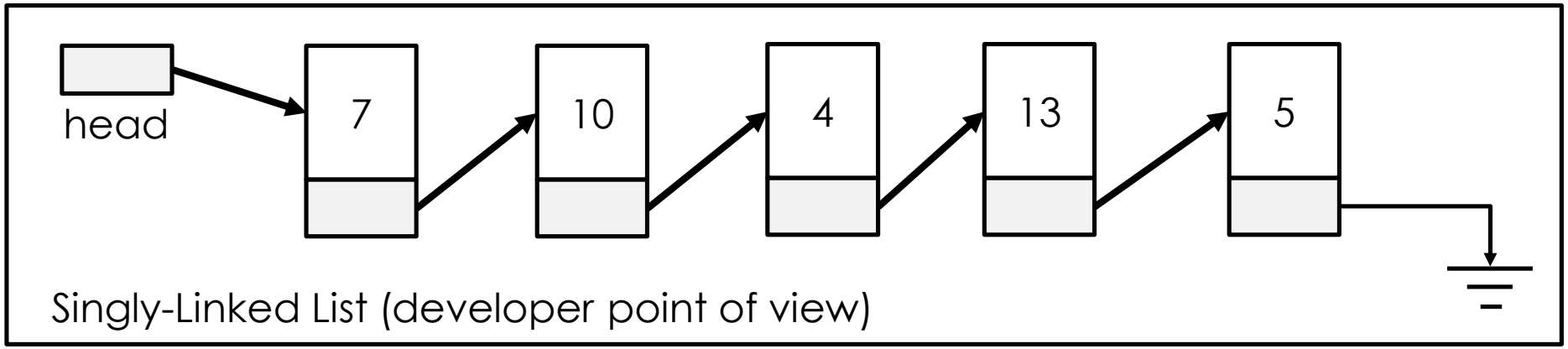
Problem with Arrays

- You need to allocate free space in memory for storing data
- In the static array, "IndexOutOfRange" Exception will raise when the data size exceeds the capacity
- Dynamic array allows you to add new data indefinitely as long as there is free space in memory BUT
  - You need to waste space twice the size of the data structure just only for storing one new object
  - Reallocation cost you O(n)
- Question is "Is there any other data structure that guarantee O(1) every time for adding a new object?"
- How about "Linked List" data structure

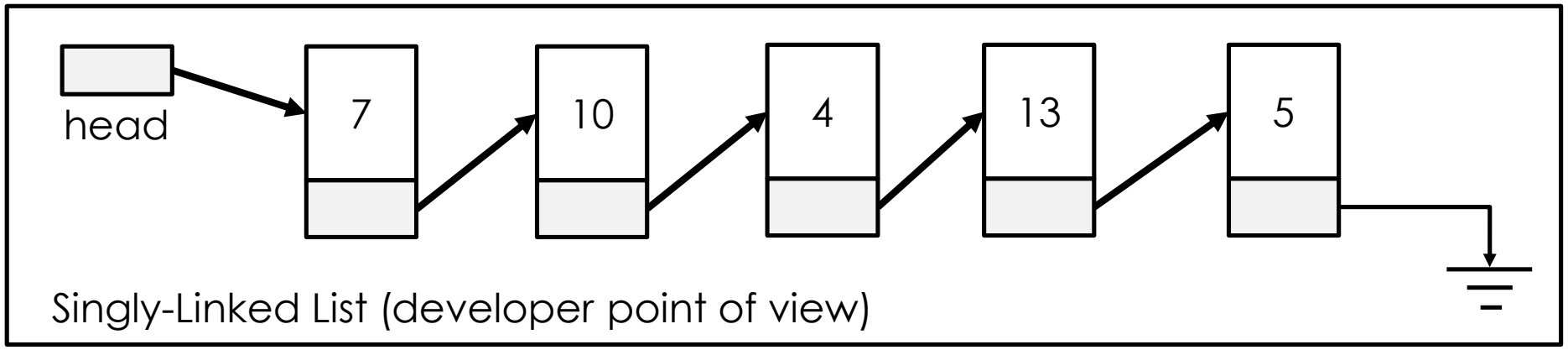
Linked List Data Structures





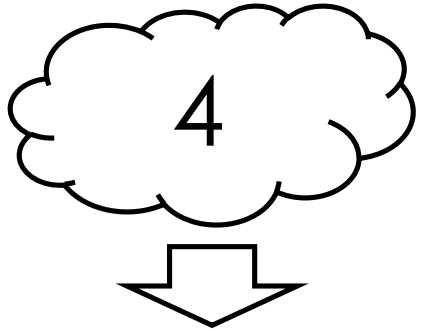


(Singly) Linked List Data Structures

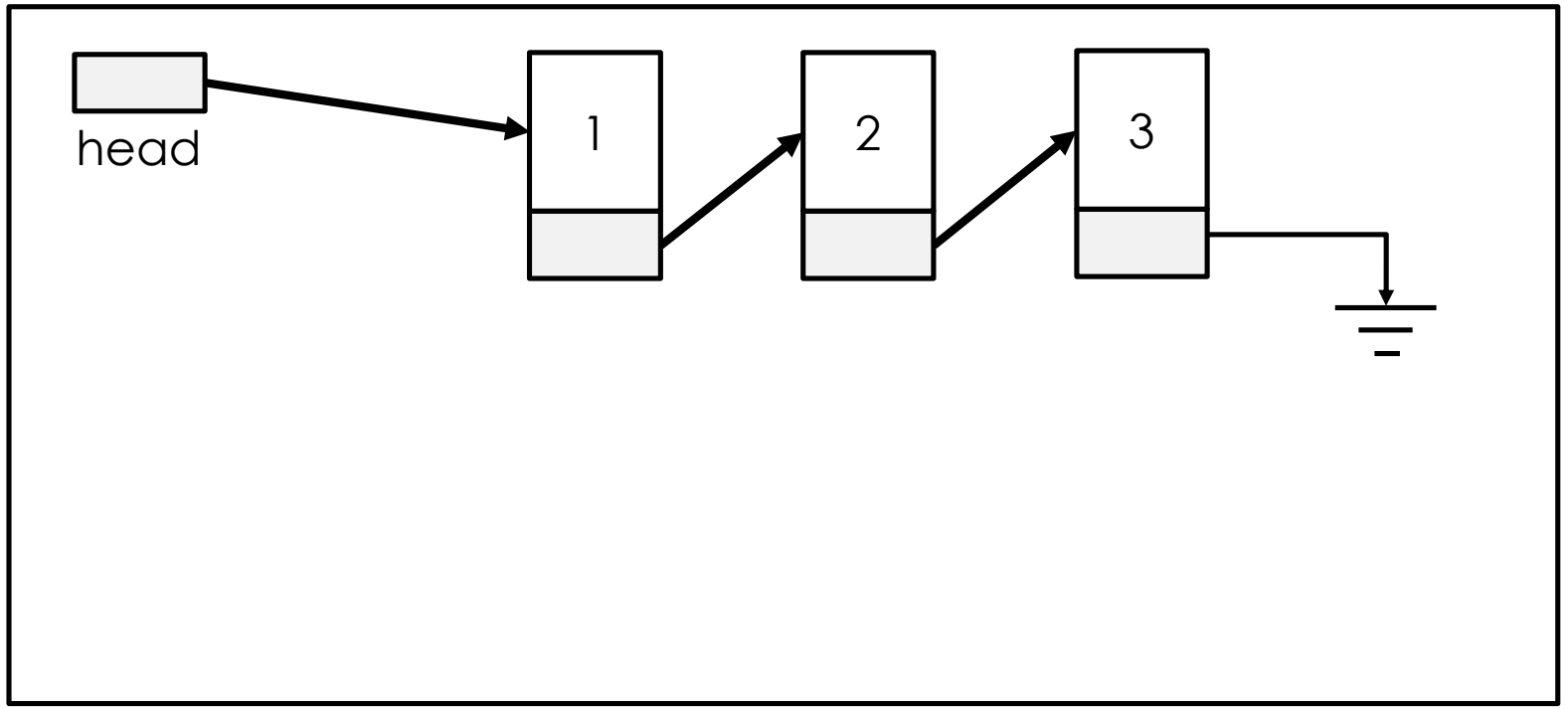


- Linked list consists of the **head** and list of **nodes**
- The **head** is a pointer/reference to the first node
- If Linked List is empty, the head will point to *null*
- Each individual ***key*** contained within a node
- A node consists of a key and the ***next*** pointer
- The ***next*** pointer of a node will point to the next node or *null*

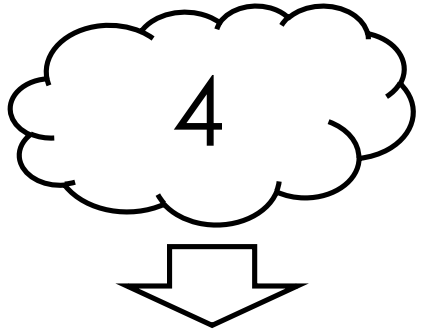
PushFront operation



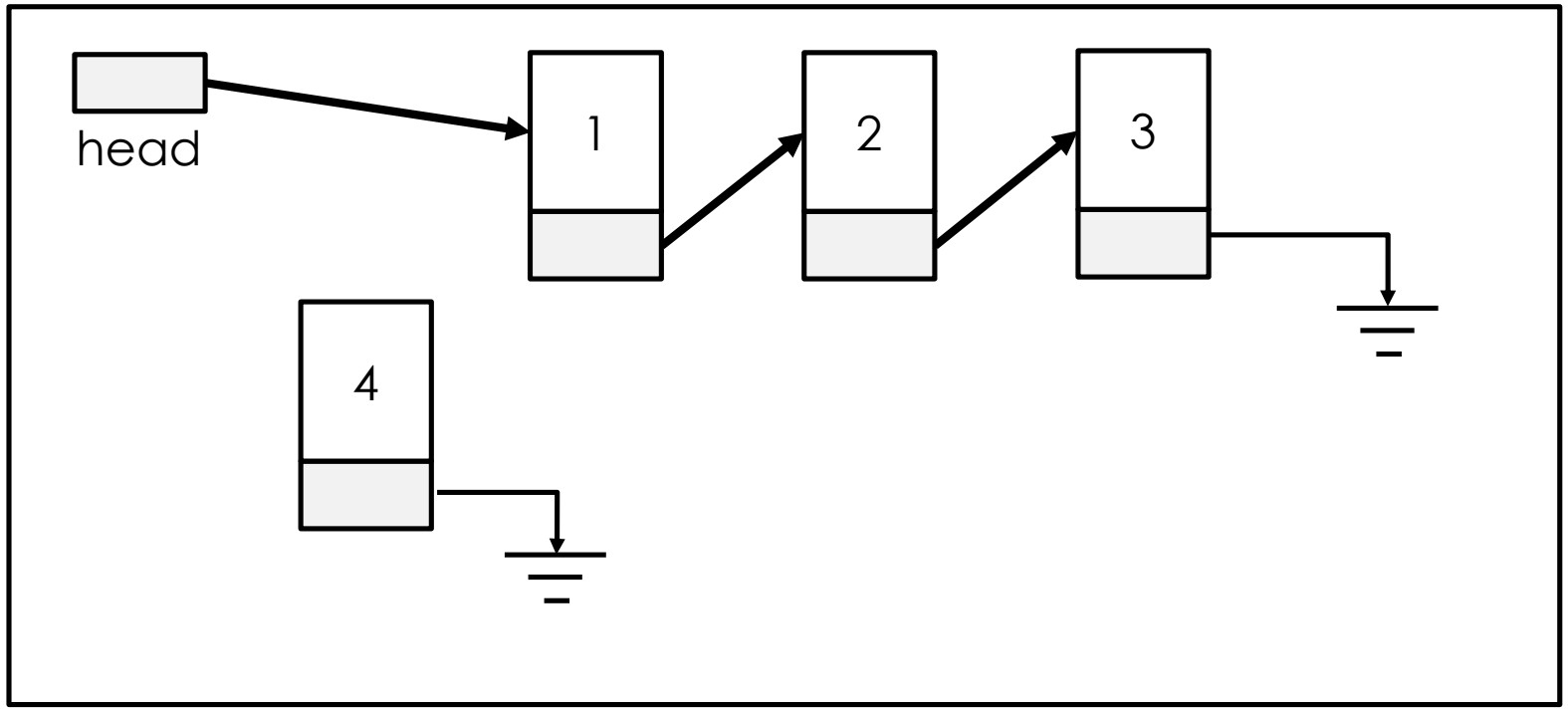
Problem: You want to add new key 4 to the front of the list



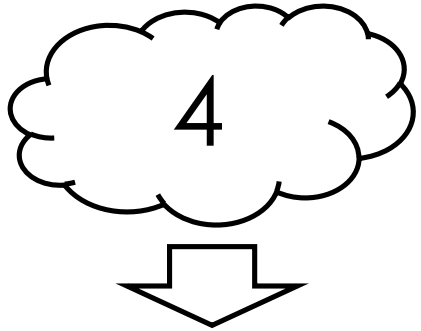
PushFront operation



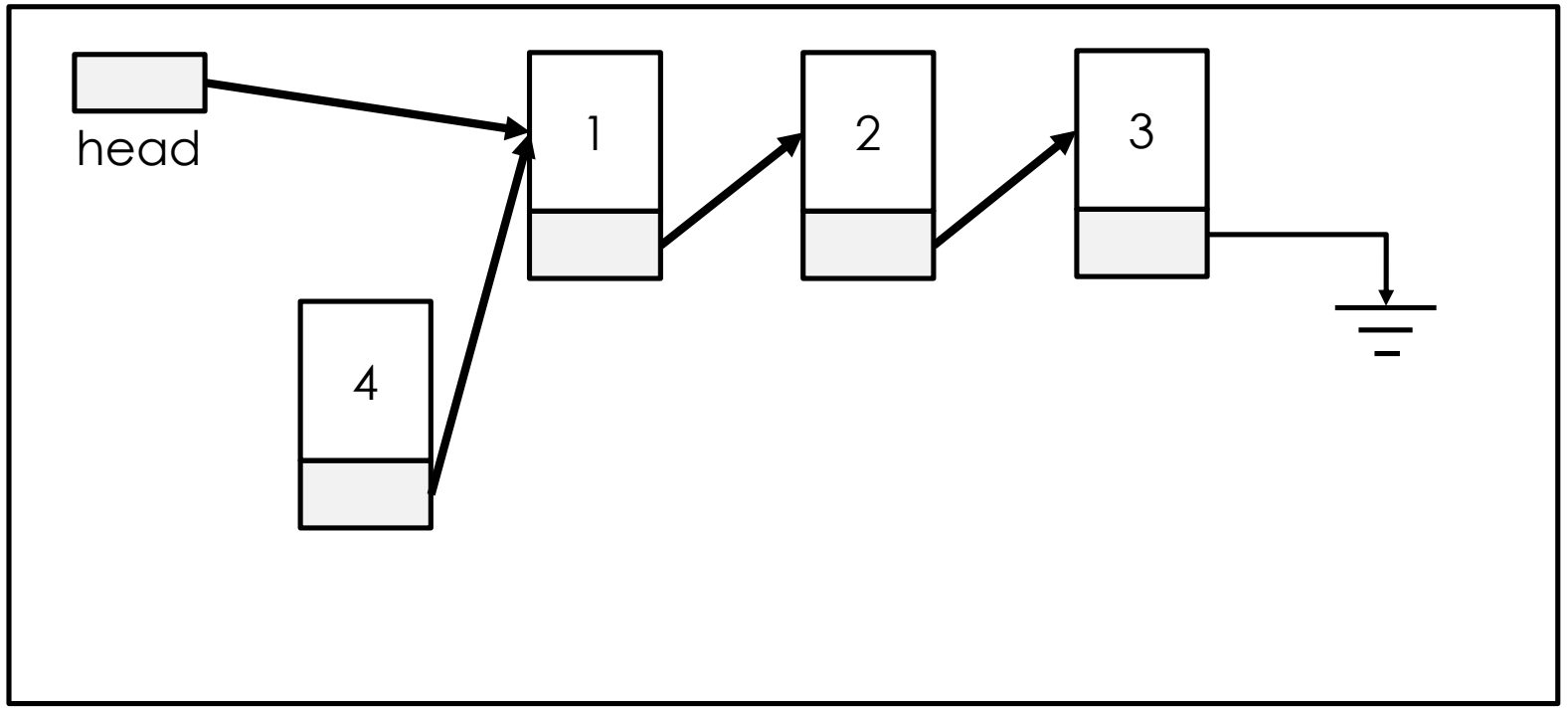
Step 1: Create a node that contains “4”



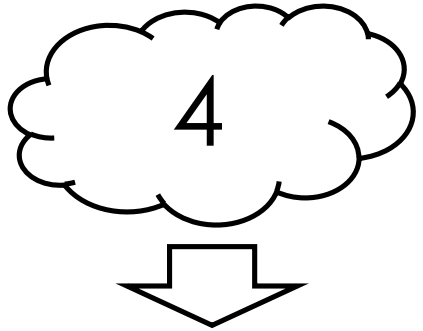
PushFront operation



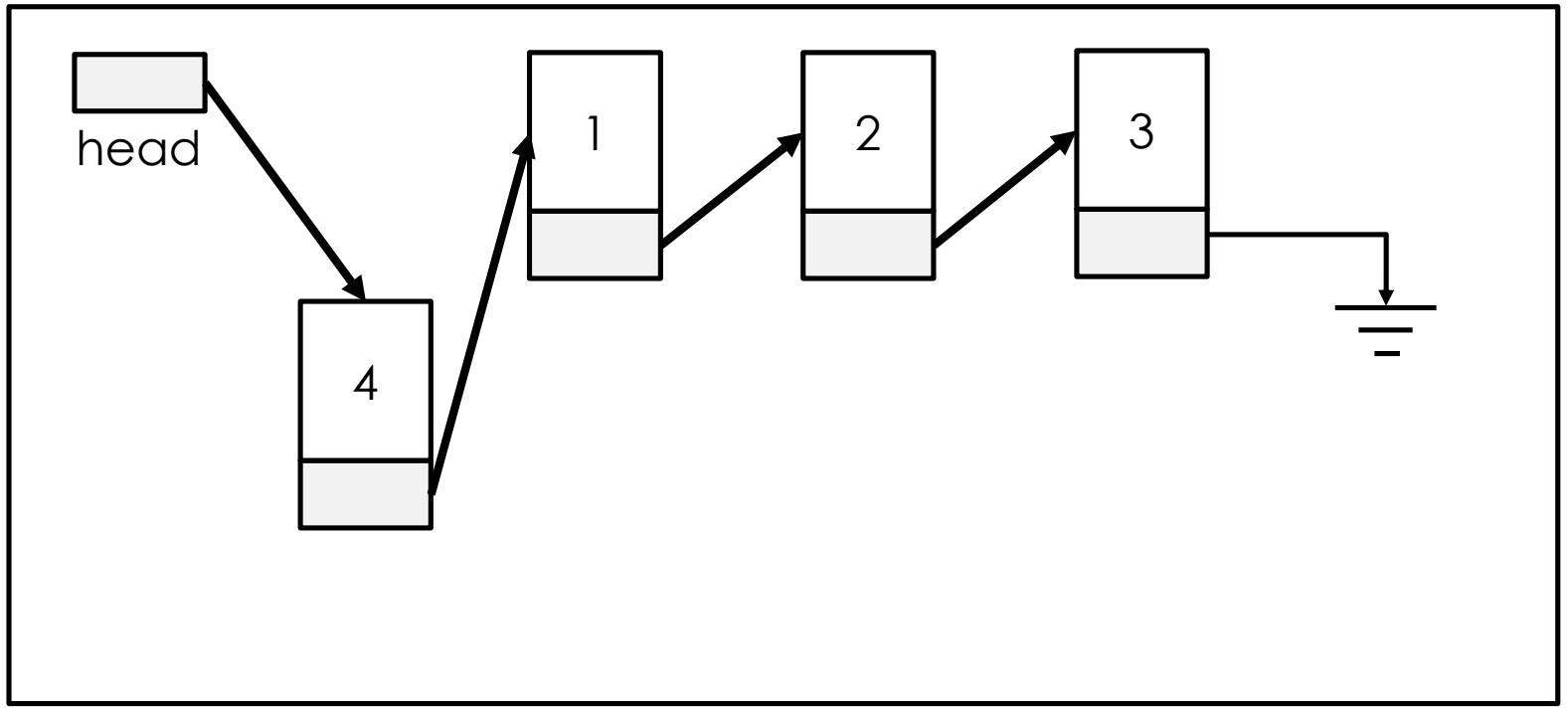
Step 2: Set next pointer of the new node to point to the node that the head points to (or the first node)



PushFront operation

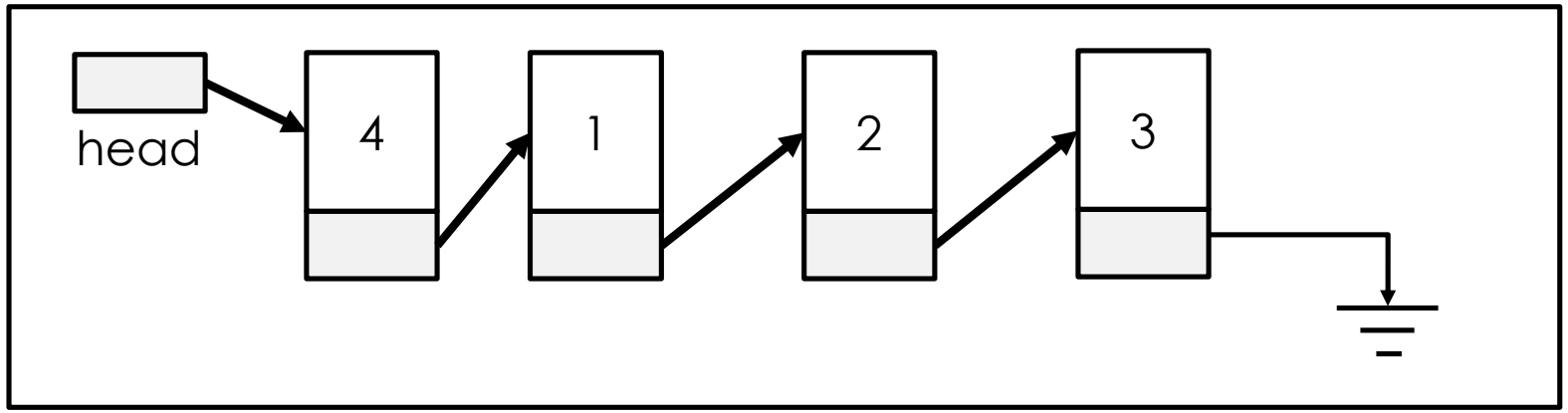


Step 3: Set the *head* to point to the new node



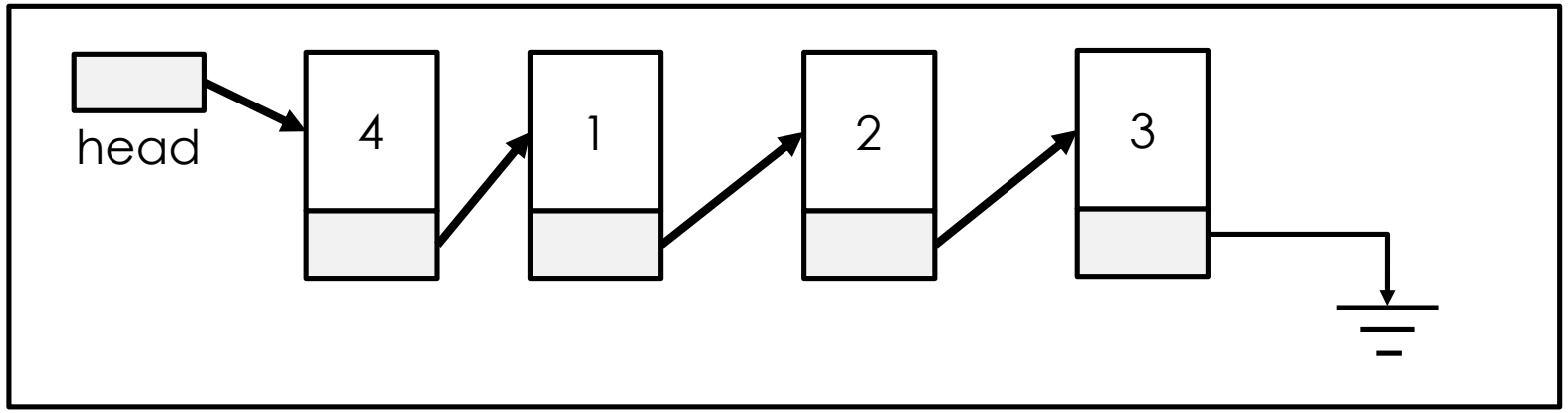
PushFront operation

What is the Big O?
O(1)



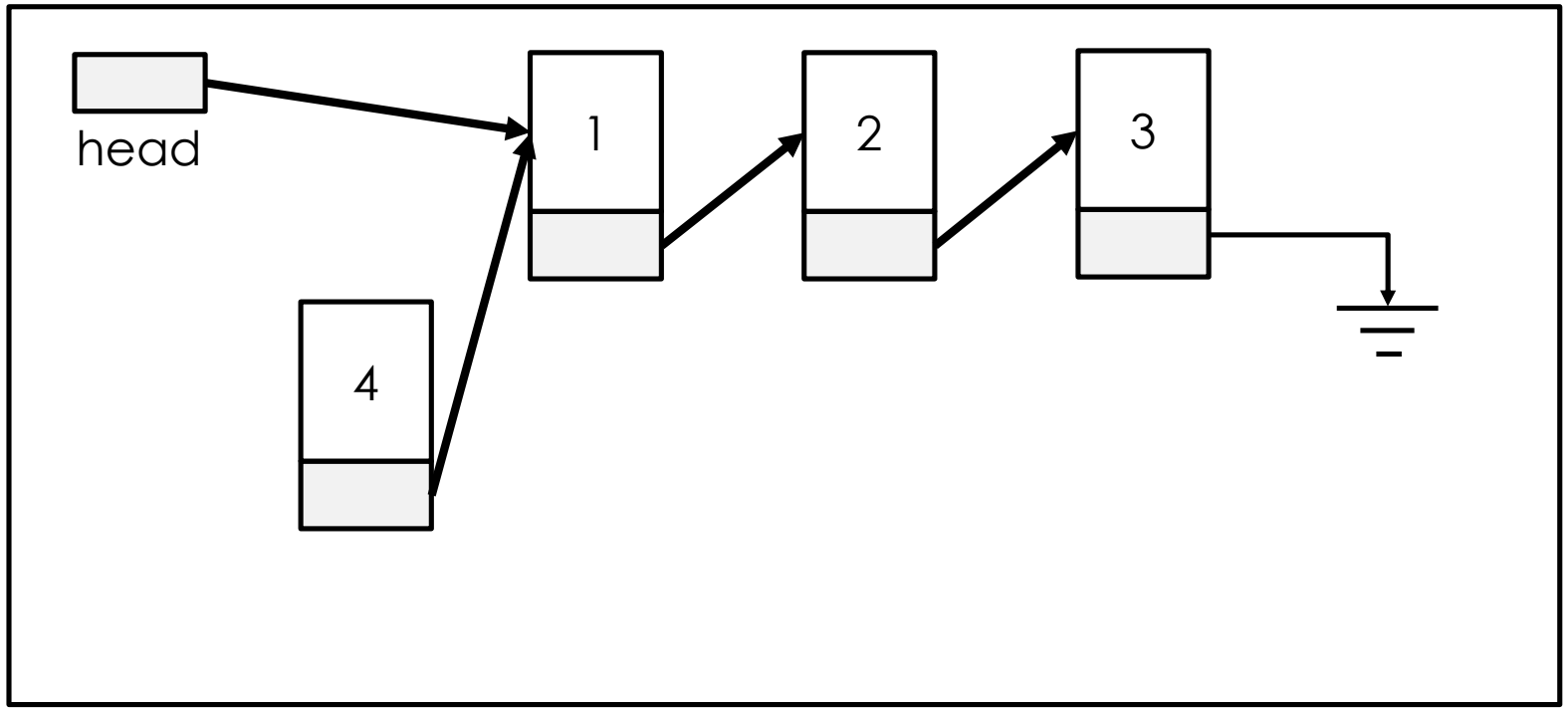
How about remove the front!

The operation is called “PopFront”



PopFront operation

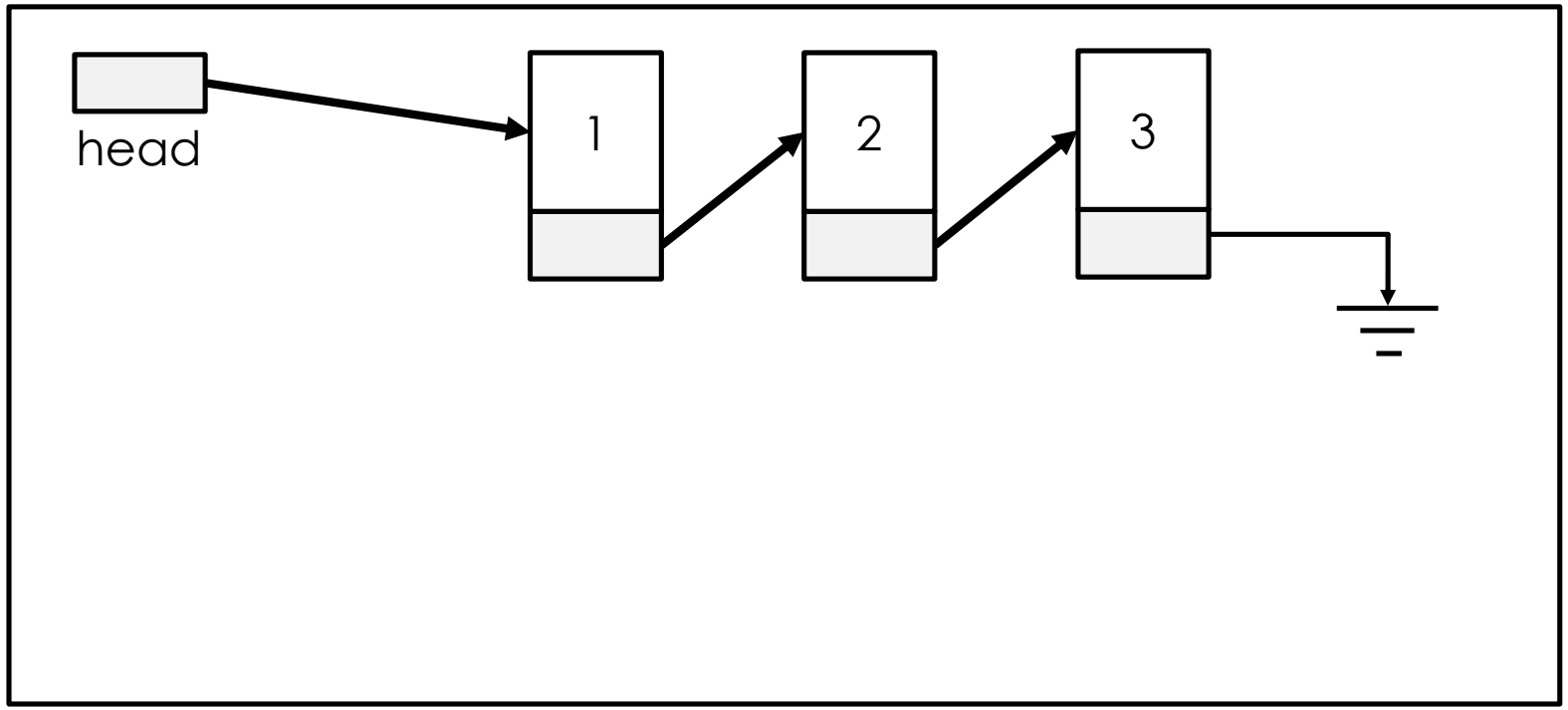
Step 1: Set ***head*** to point to "the node that the first node points to" or "the second node"



PopFront operation

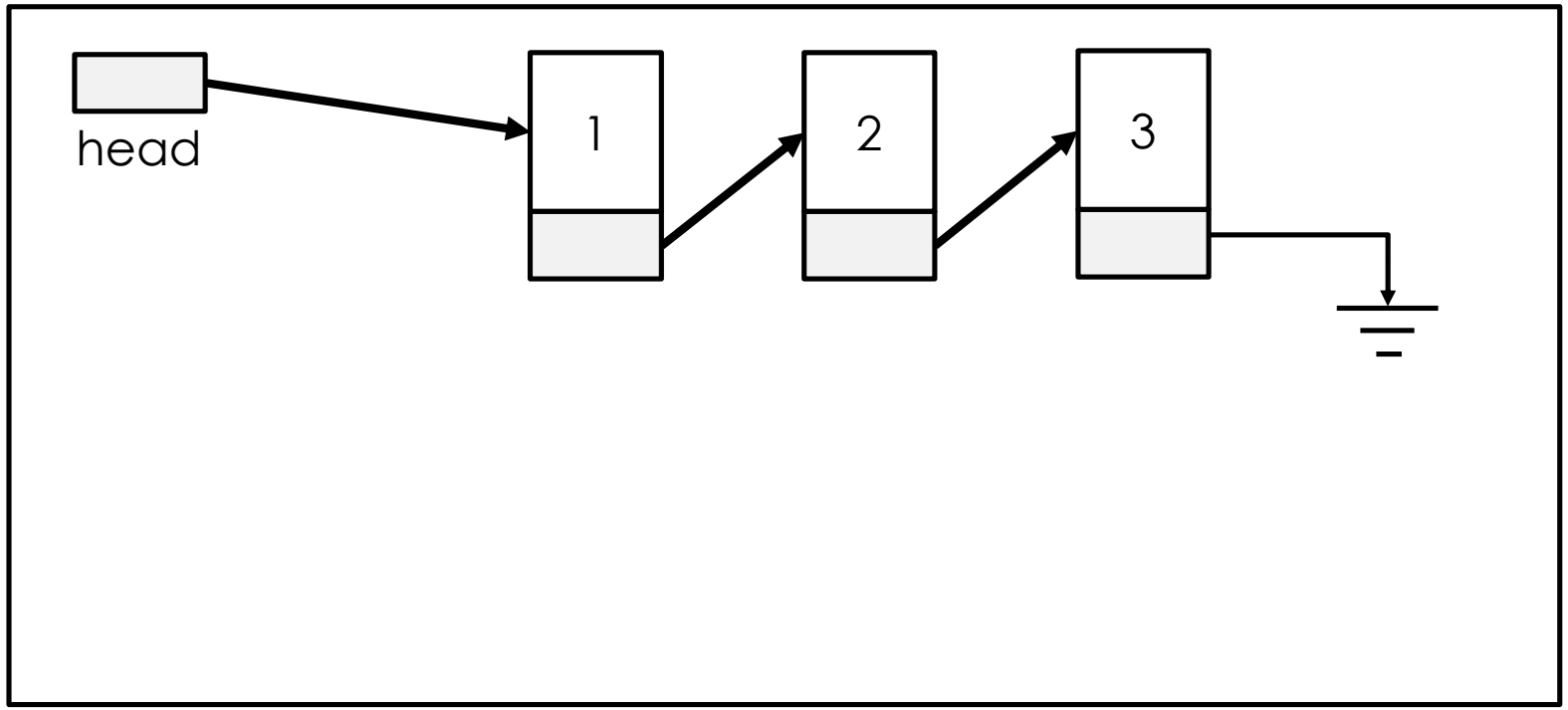
Step 2: Delete the first node

- In C#, *the Garbage Collection Module* will automatically mark and delete any objects that are not referenced
- In C++, you need to manually delete the object using *delete* keyword otherwise there will be "memory leaks"

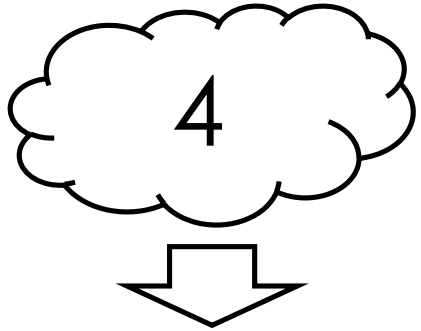


PopFront operation

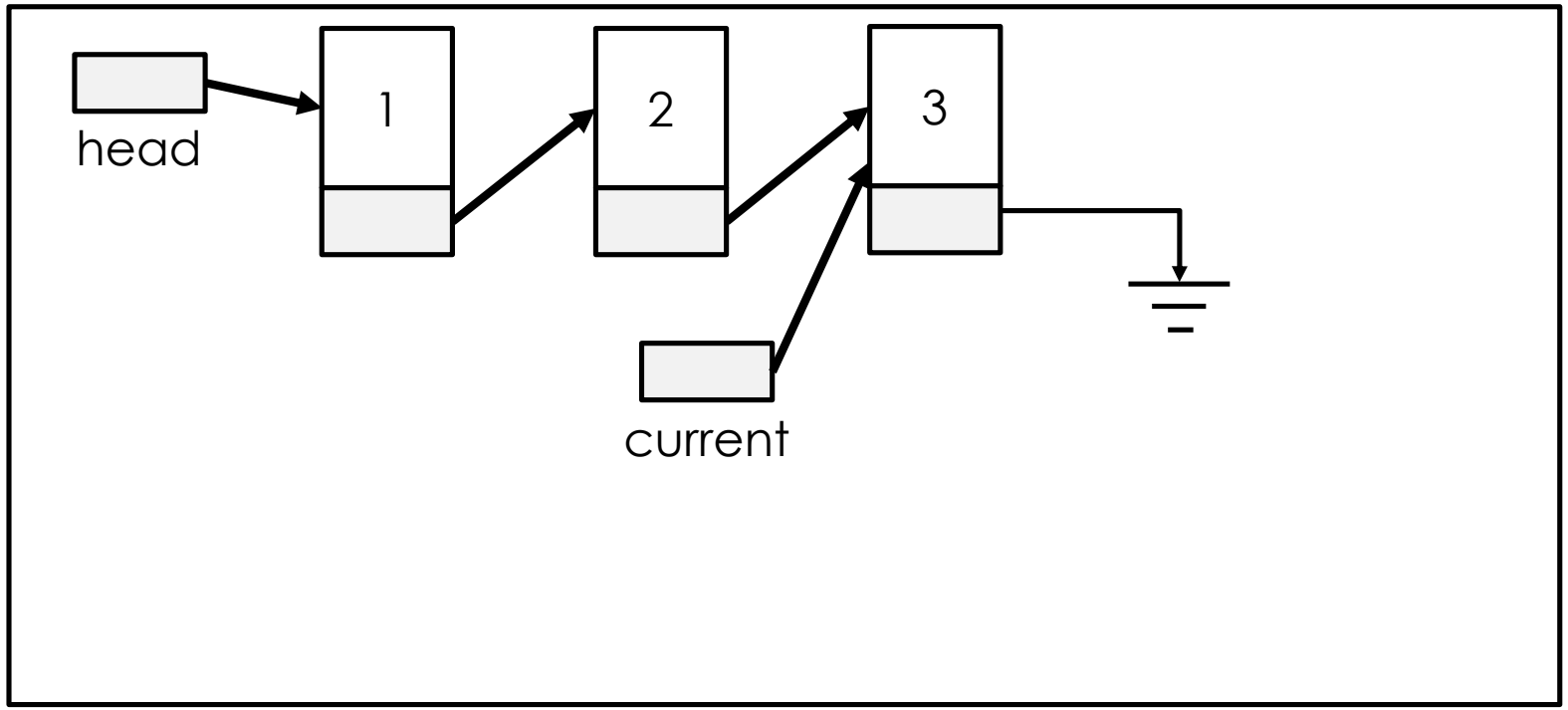
What is the Big O?
O(1)



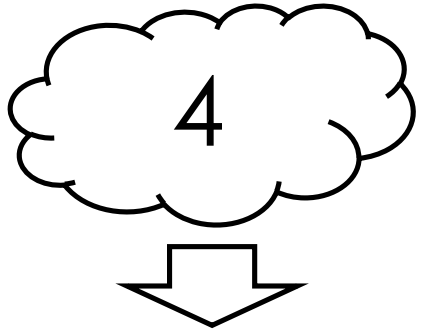
PushBack Operation



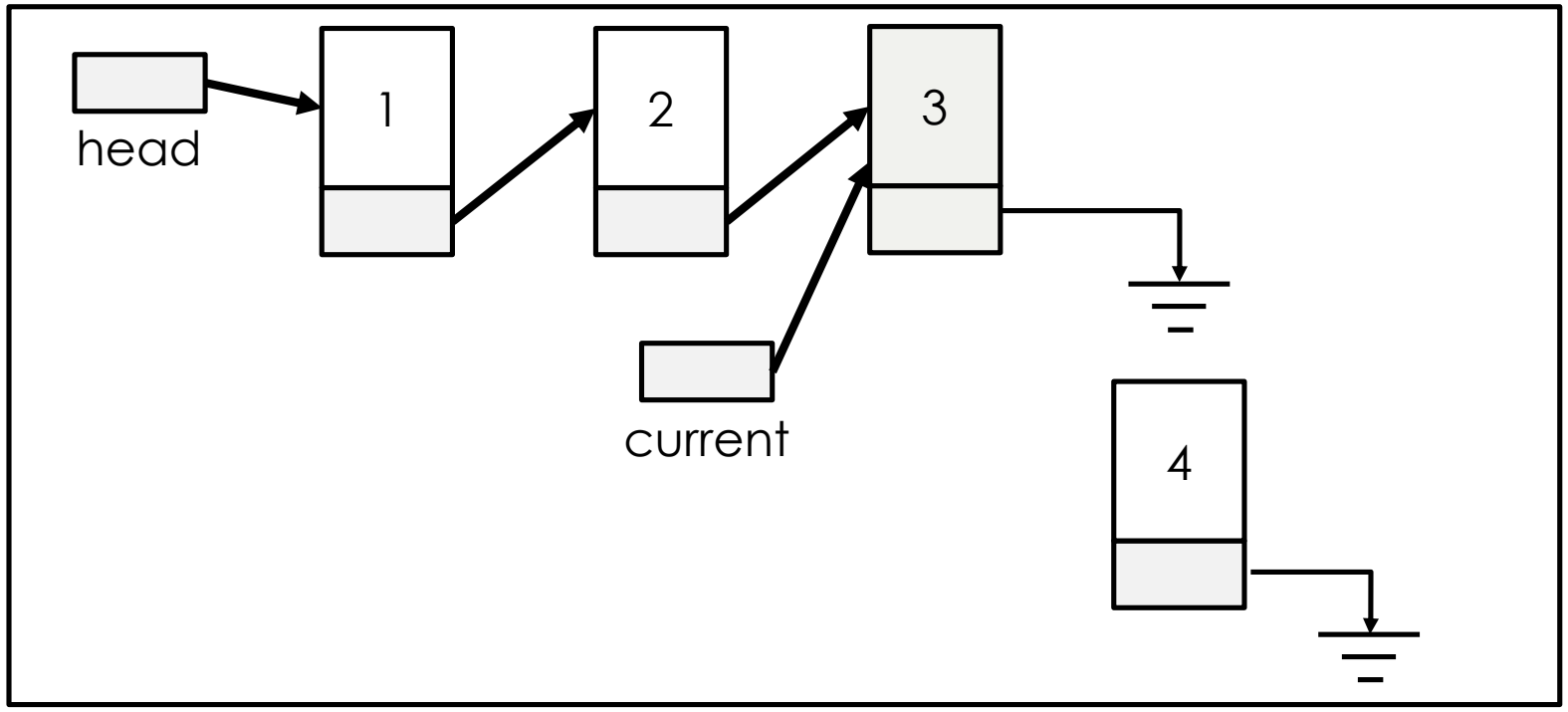
Step 1: Find the last node; start from ***head***



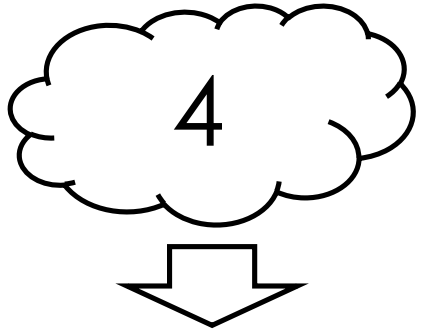
PushBack Operation



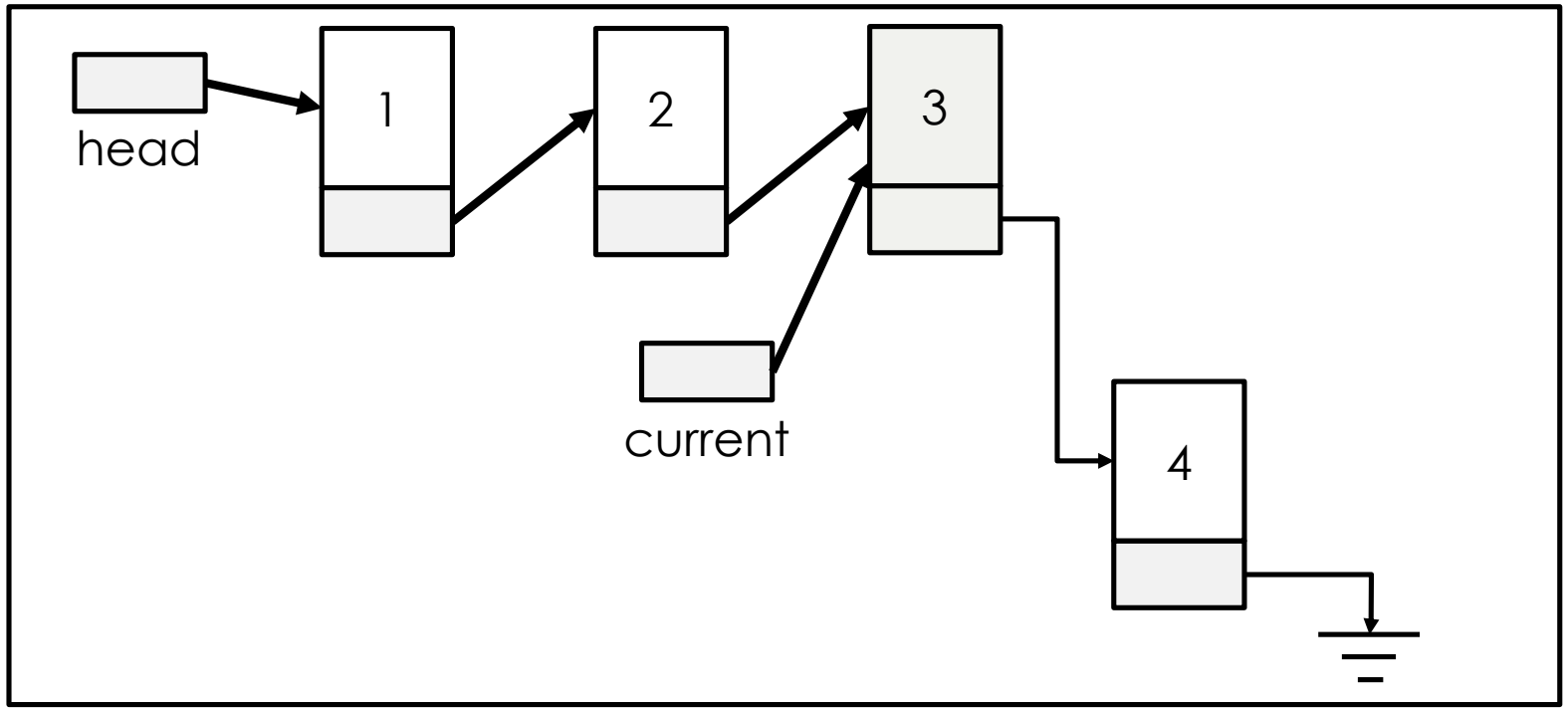
Step 2: Create a node that contains “4”



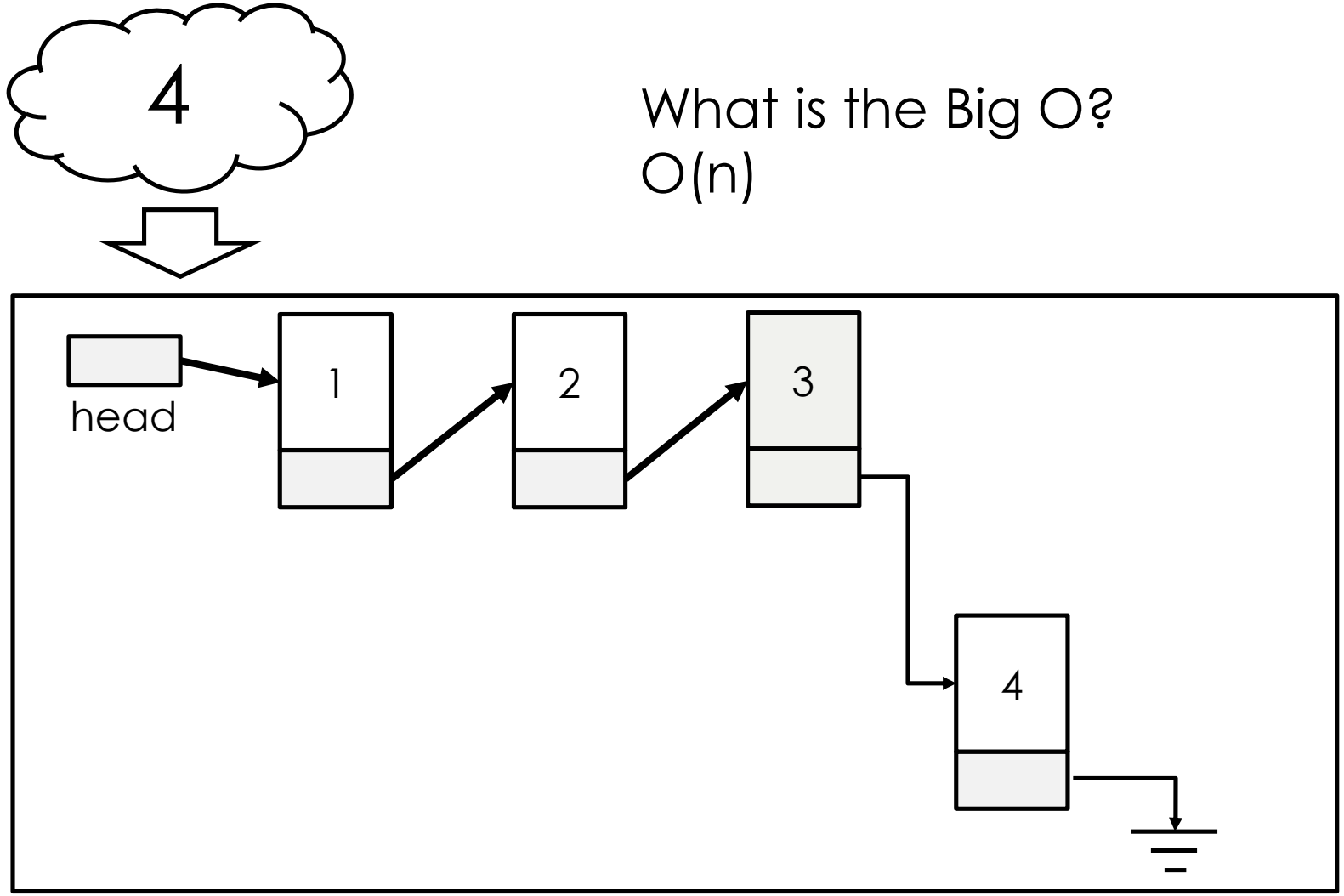
PushBack Operation



Step 3: Set next pointer of the last node to point to the new node



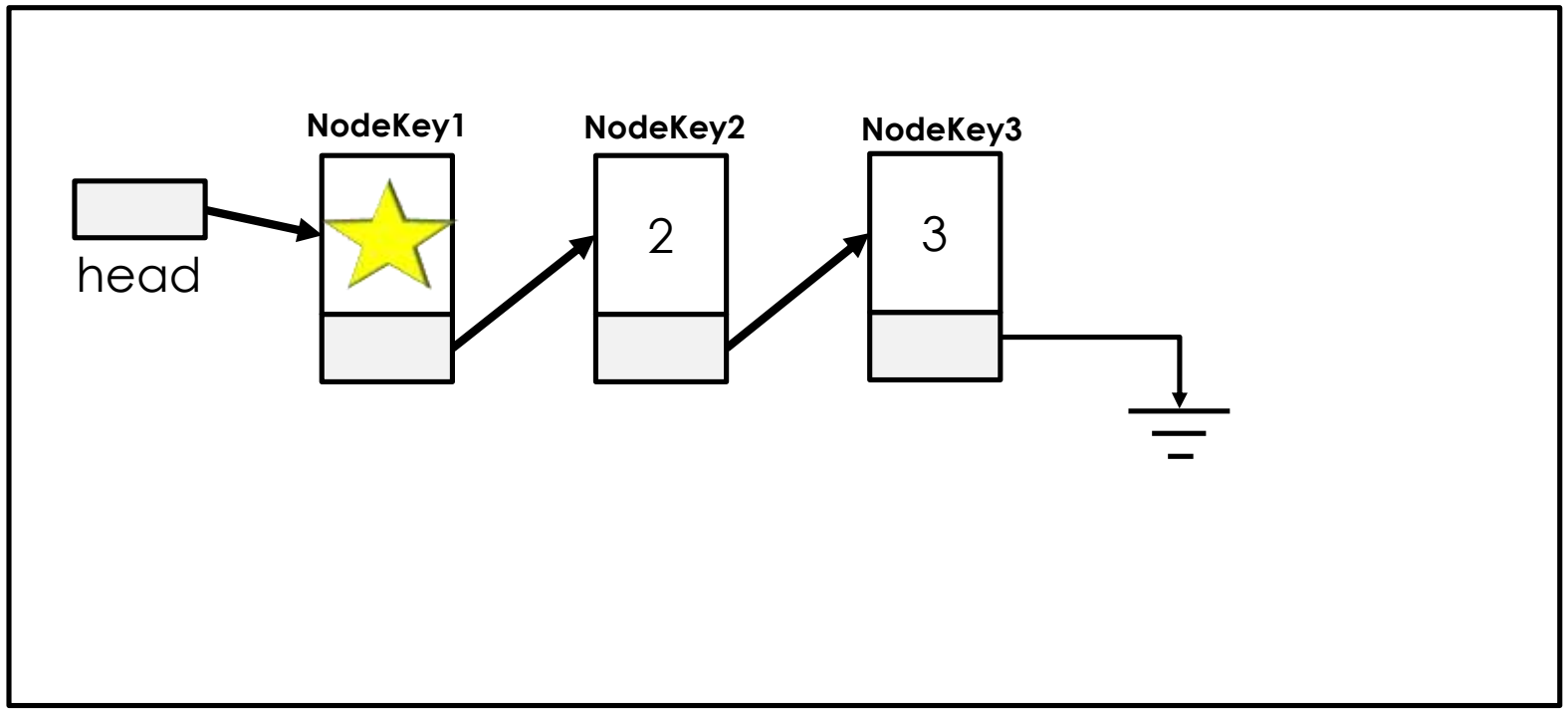
PushBack Operation



What is the Big O?
O(n)

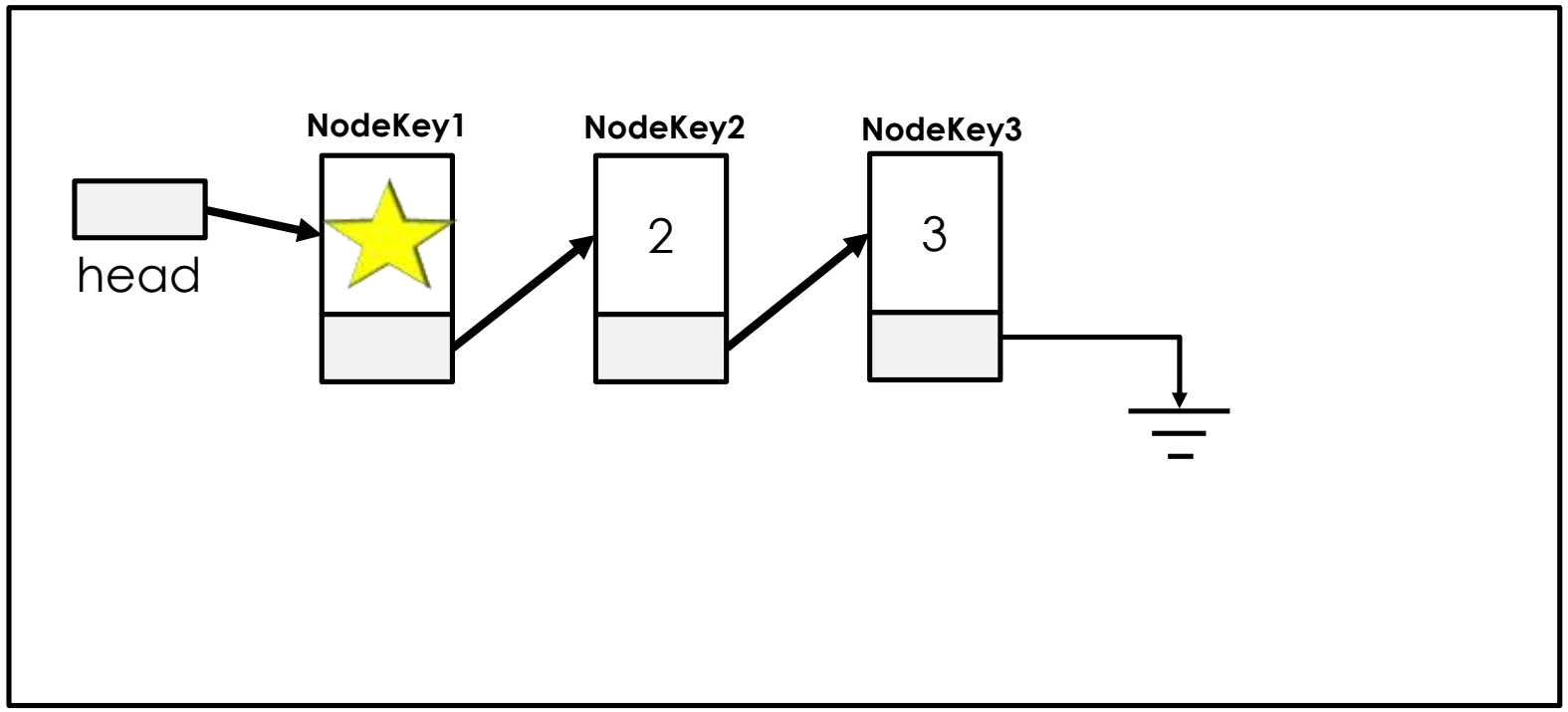
PopBack Operation

Step 1: Start from ***head***, check if the next node (NodeKey1) points to *null* or not



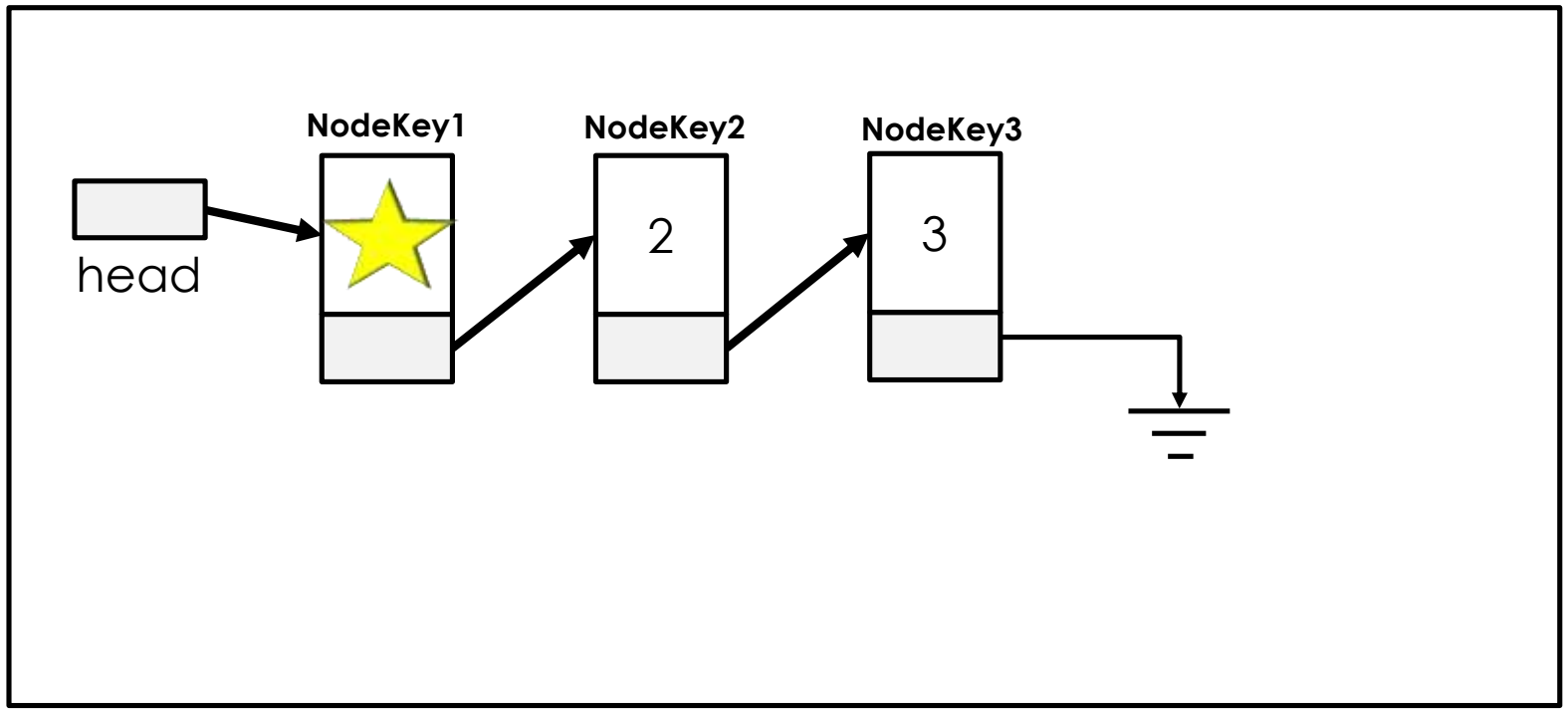
PopBack Operation

Step 2: If not, move to the next node (NodeKey1).



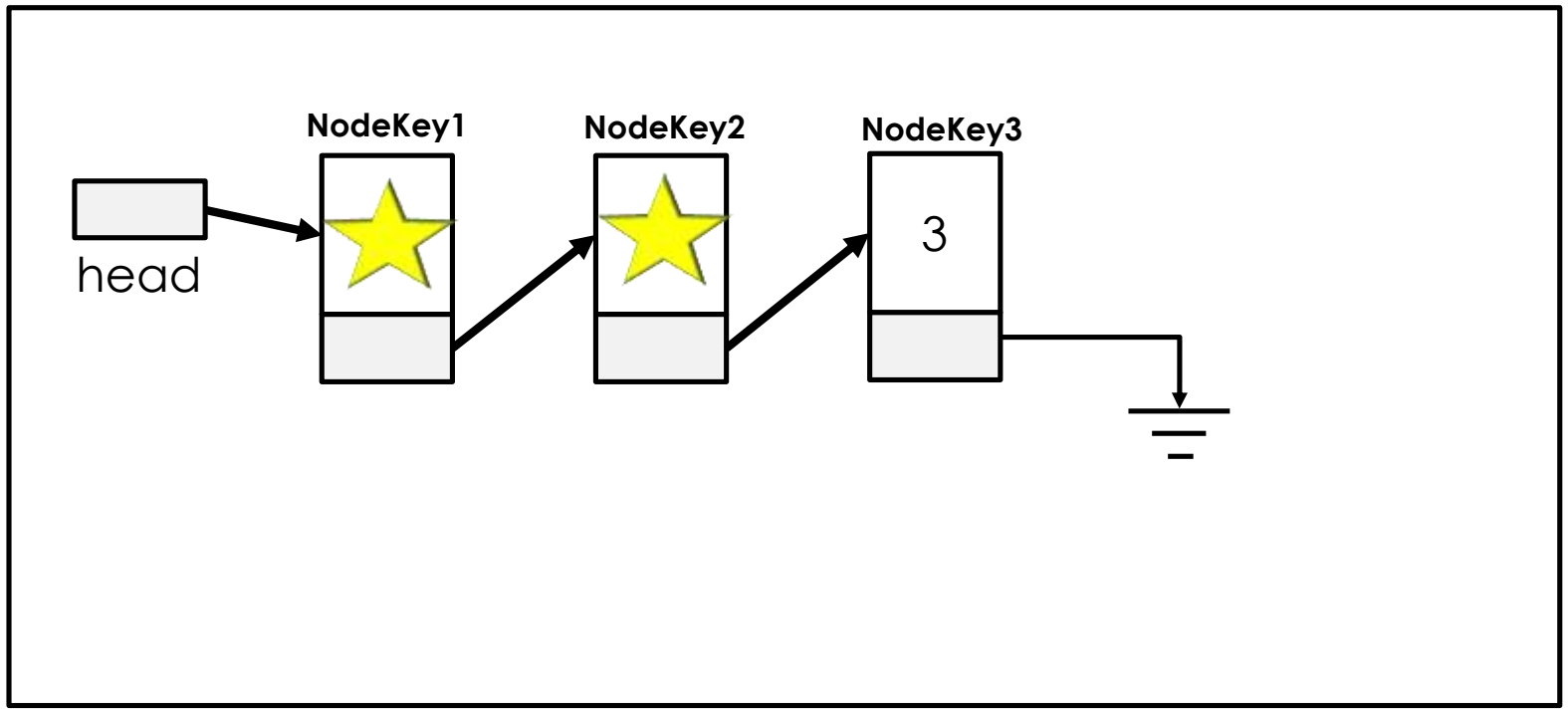
PopBack Operation

Step 2: Repeat the process.
Check if the next node (NodeKey2) points to *null* or not?



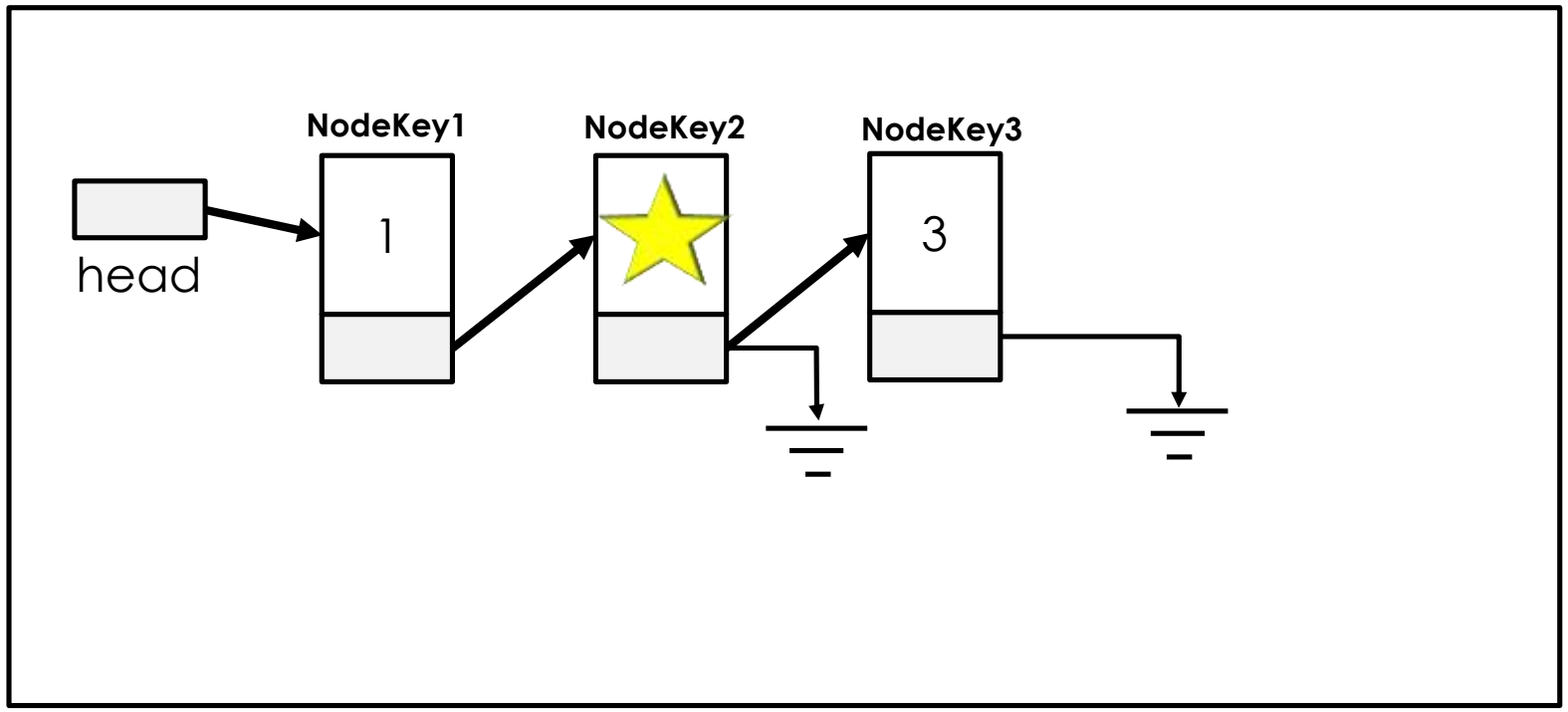
PopBack Operation

Step 2: If not, repeat the process.
Move and then check if the next node (NodeKey3) points to *null* or not?



PopBack Operation

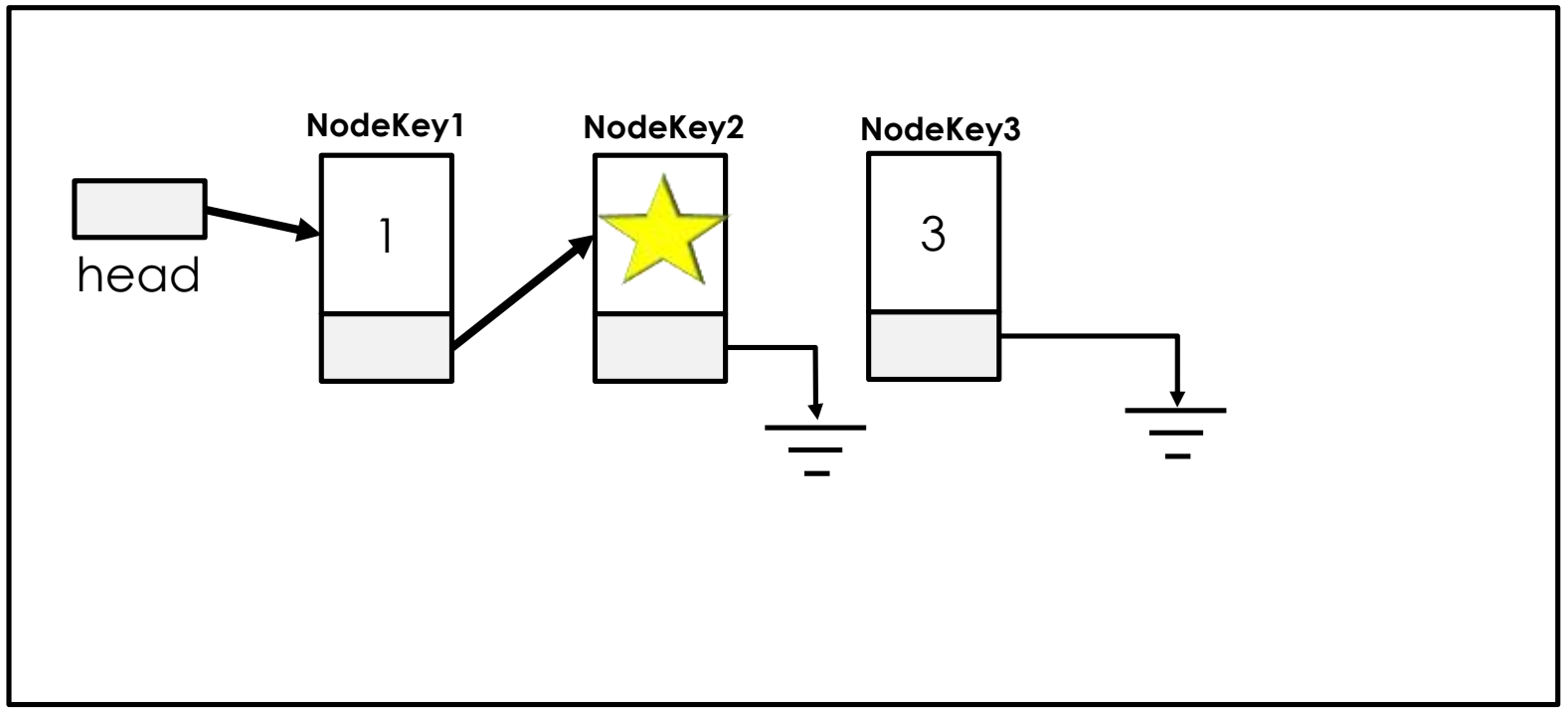
Step 2: If the next node (NodeKey3) points to *null*, set the current next pointer to *null*



PopBack Operation

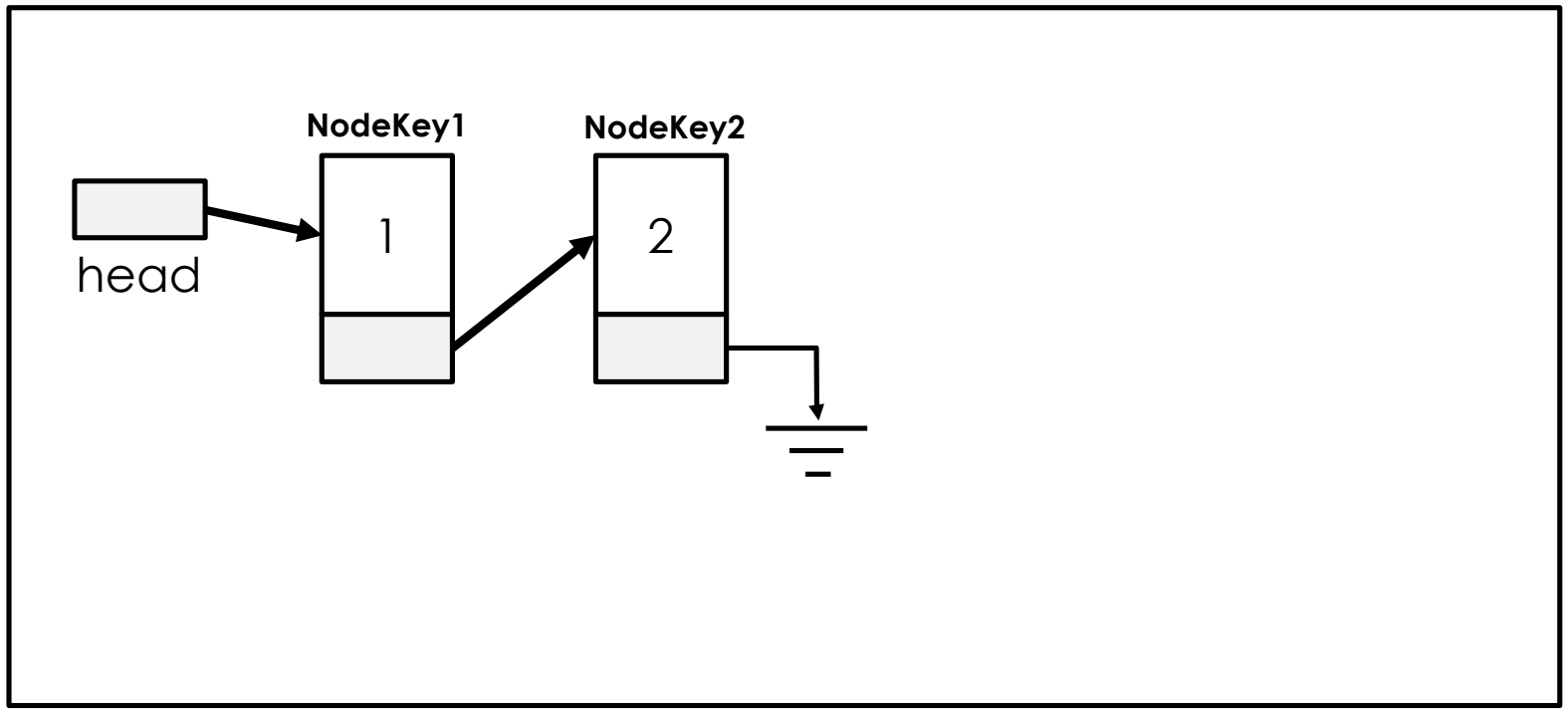
Step 2: Free NodeKey3

In C#, it will be automatically cleared by the .NET Garbage Collection, because it is not referenced by any variables
In C++, you need to clear memory by using *delete* keyword



PopBack Operation

What is the Big O?
O(n)

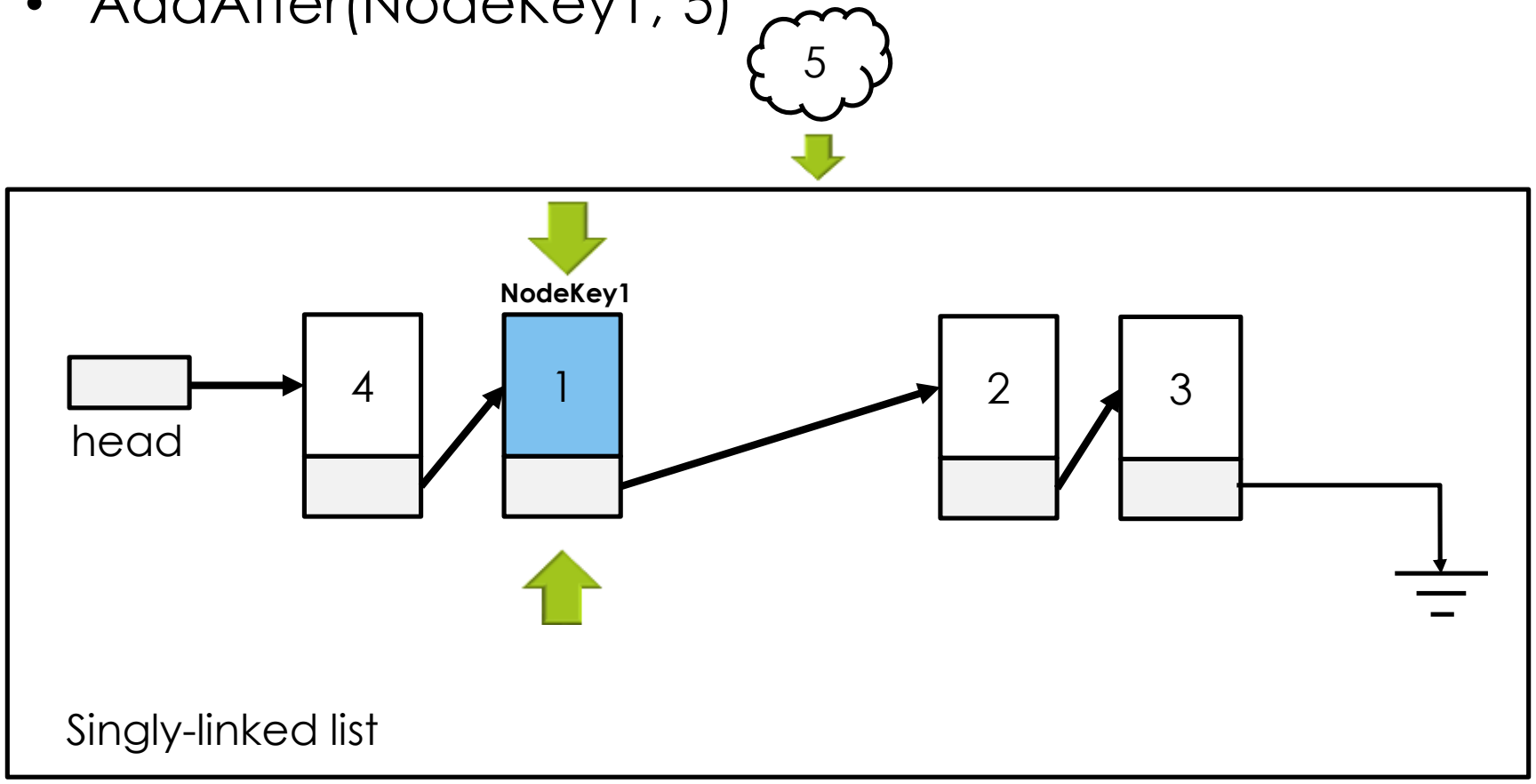


List Application Programming Interface

PushFront(Key)	add to front
PushBack(Key)	add to back
Key PopFront()	remove front item
Key PopBack()	remove back item
Key TopFront()	return front item
Key TopBack()	return back item
Boolean Find(Key)	is key in list?
Erase(Key)	remove key front list
Empty()	empty list?
AddBefore(Node, Key)	Add key before node
AddAfter(Node, Key)	Add key after node

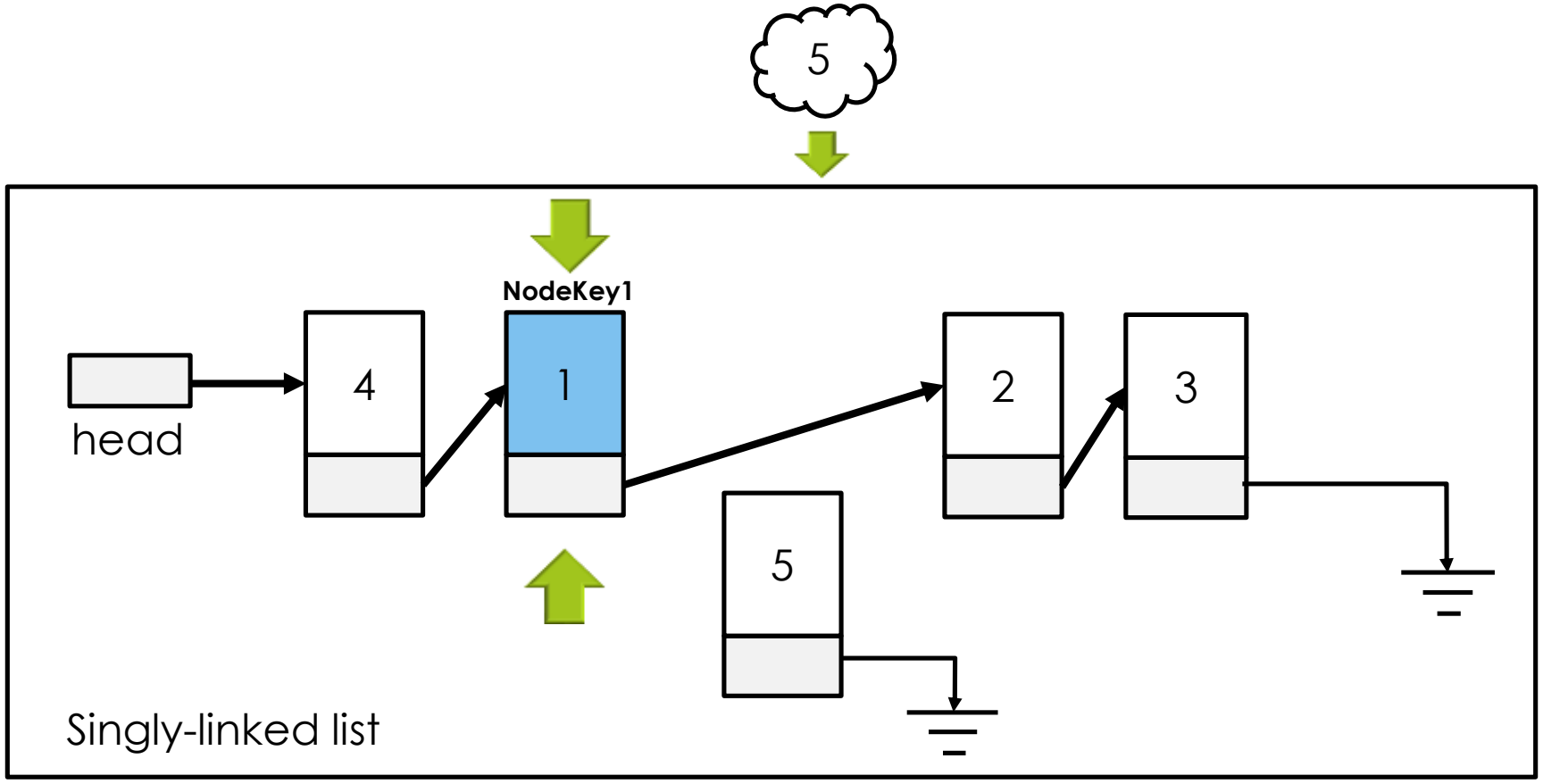
AddAfter(Node, Key) operation

- Given a node and a key, add another with the key after that node
- AddAfter(NodeKey1, 5)



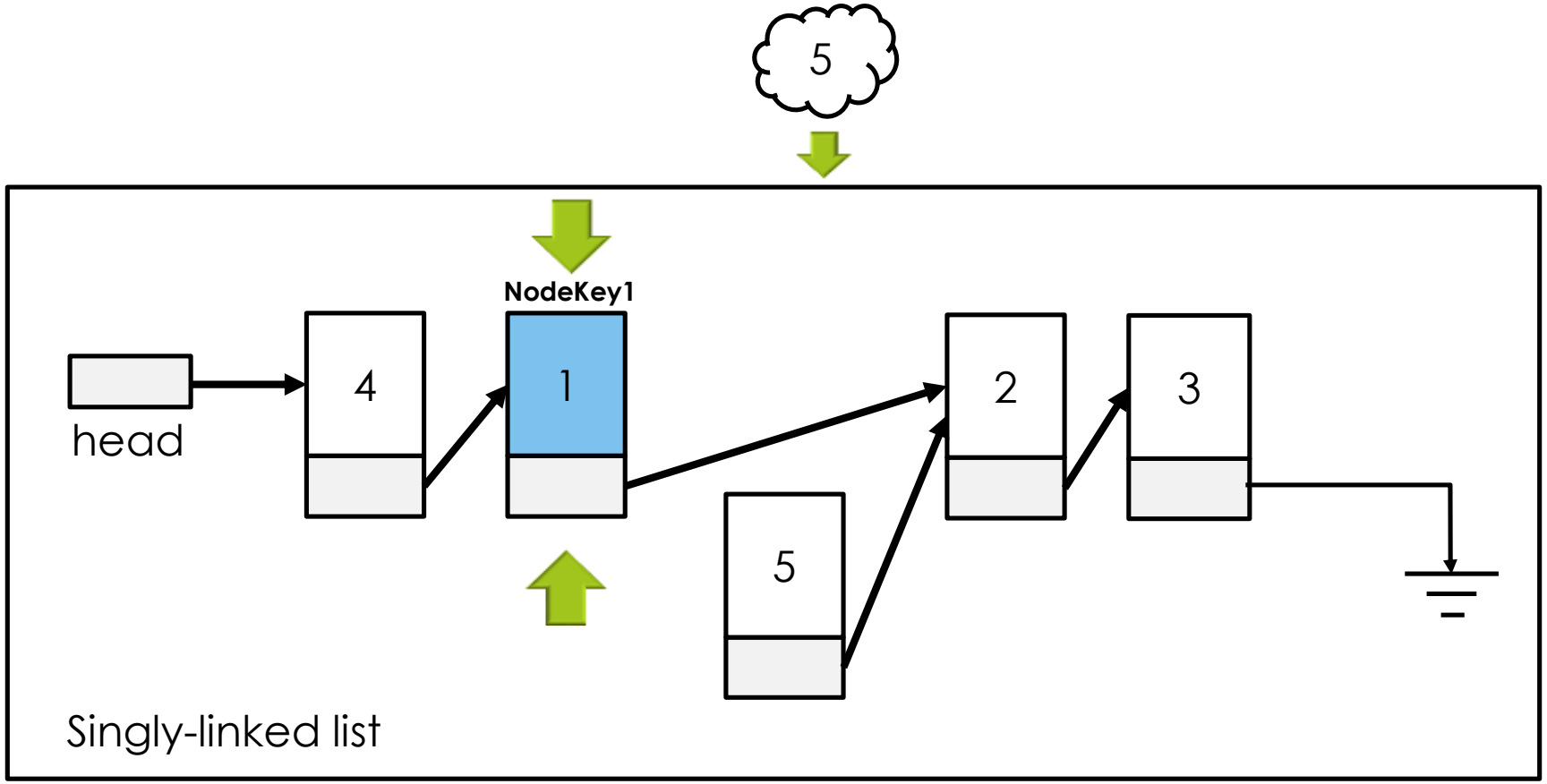
AddAfter(Node, Key) operation

- Step 1: Create a node that contains Key “5”



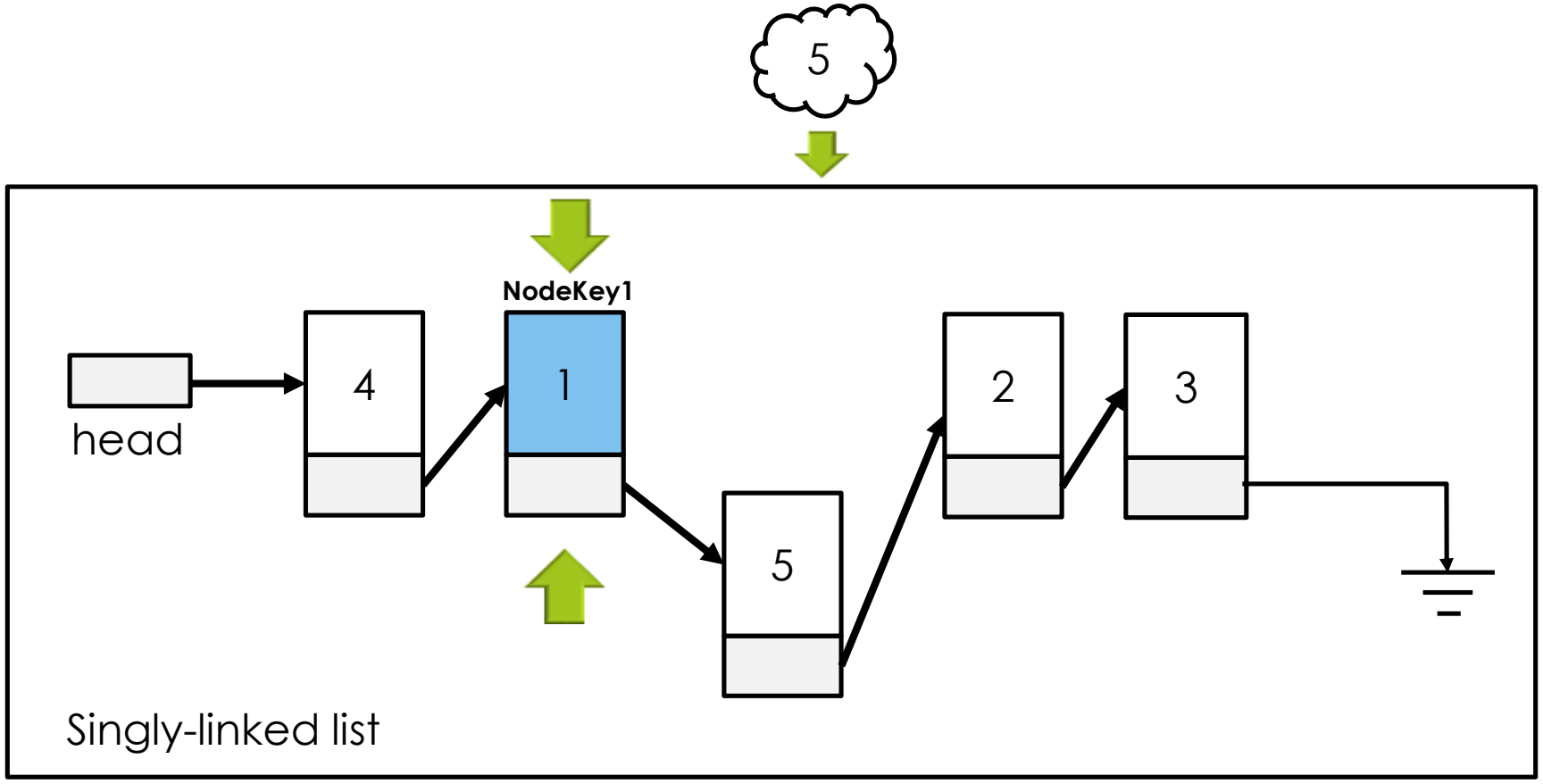
AddAfter(Node, Key) operation

- Step 2: Set pointer of the new node (NodeKey5) to the node pointed by the NodeKey1



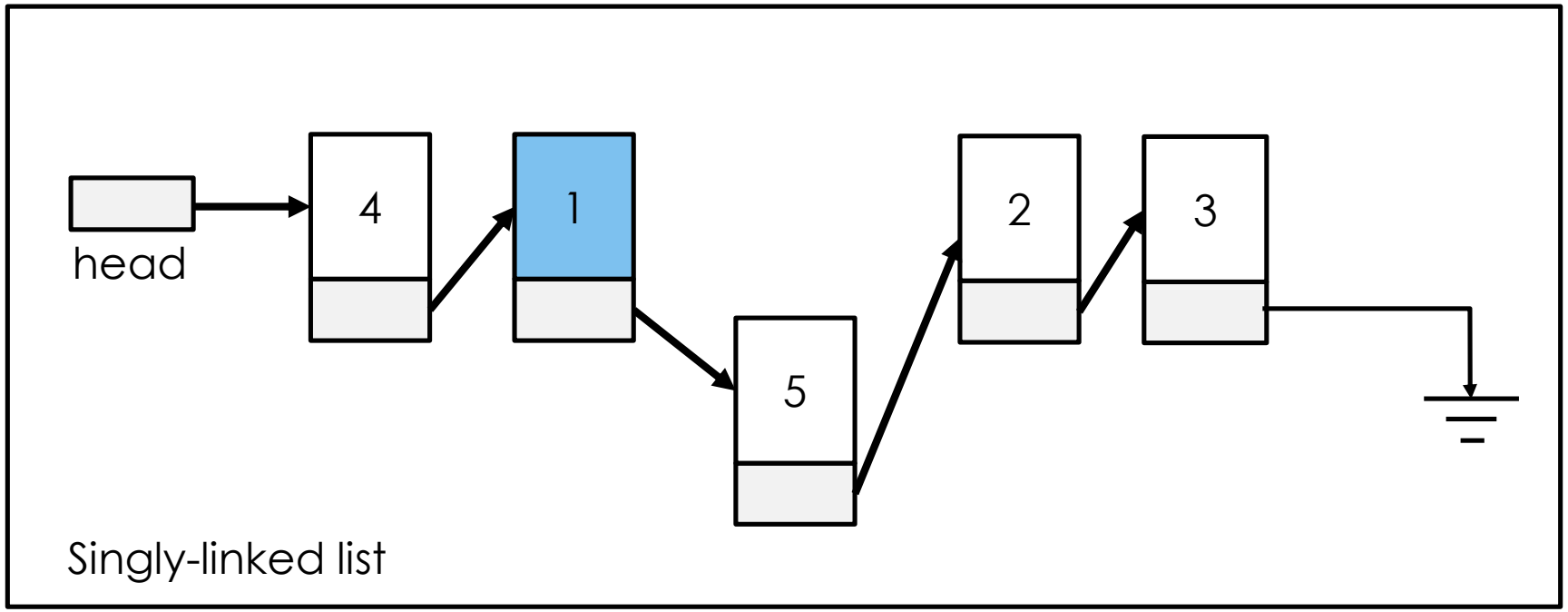
AddAfter(Node, Key) operation

- Step 3: Set pointer of the NodeKey1 to the new node (NodeKey5)



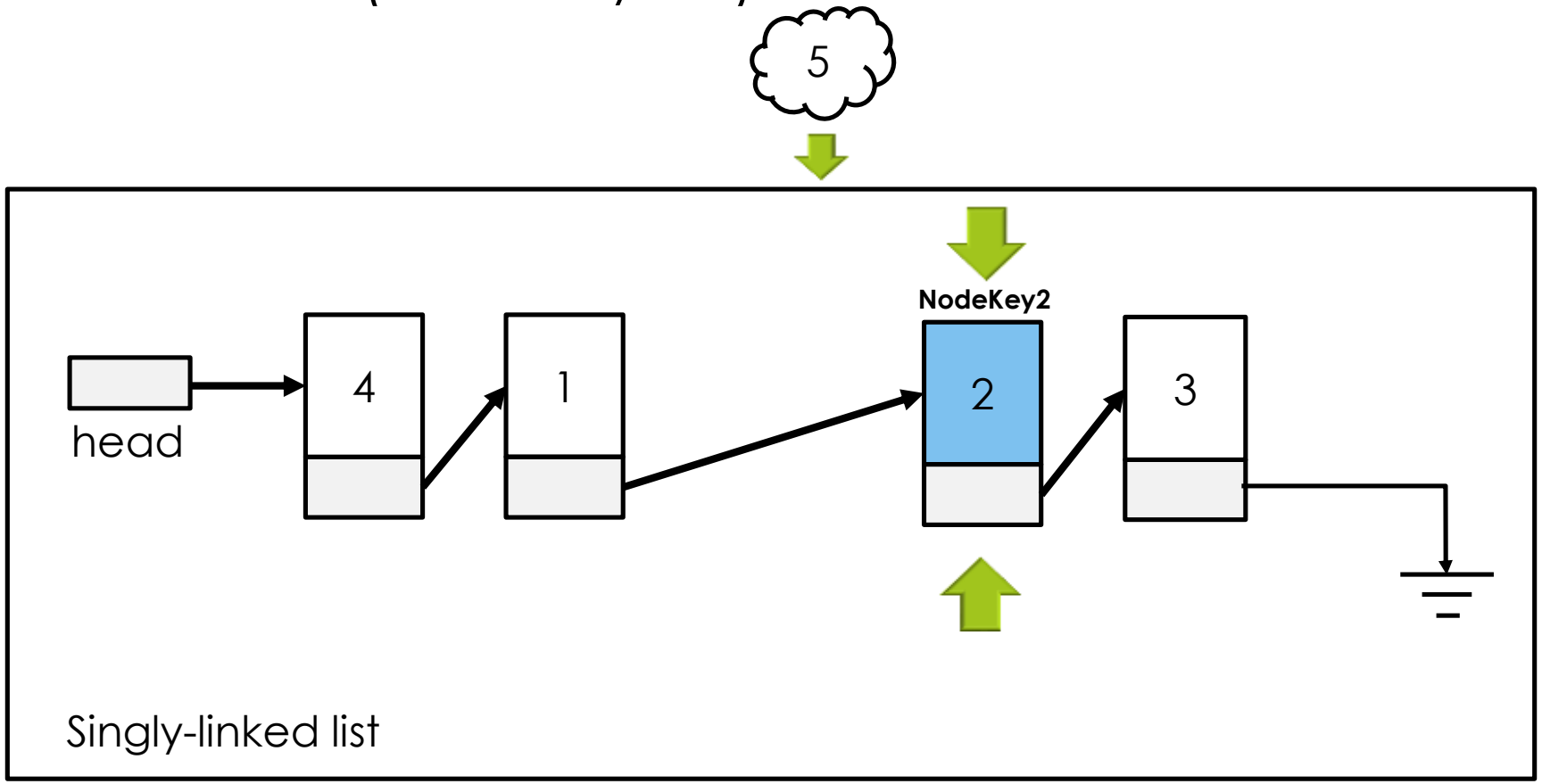
AddAfter(Node, Key) operation

- What is the Big O?
- Ans: O(1)
- or O(n) in what case?



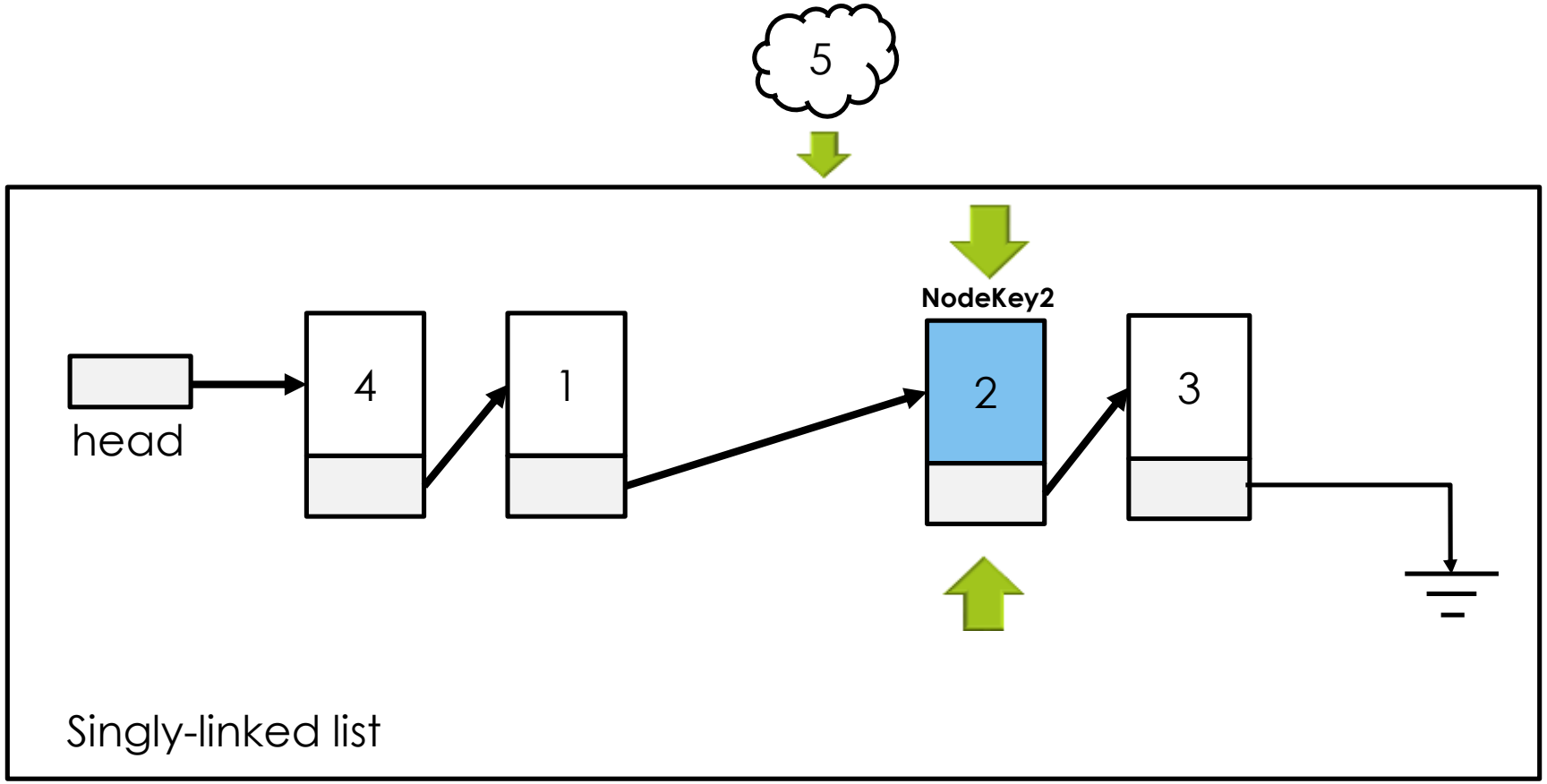
AddBefore(Node, Key) operation

- Given a node and a key, add another with the key before that node
- AddBefore(NodeKey2, 5)



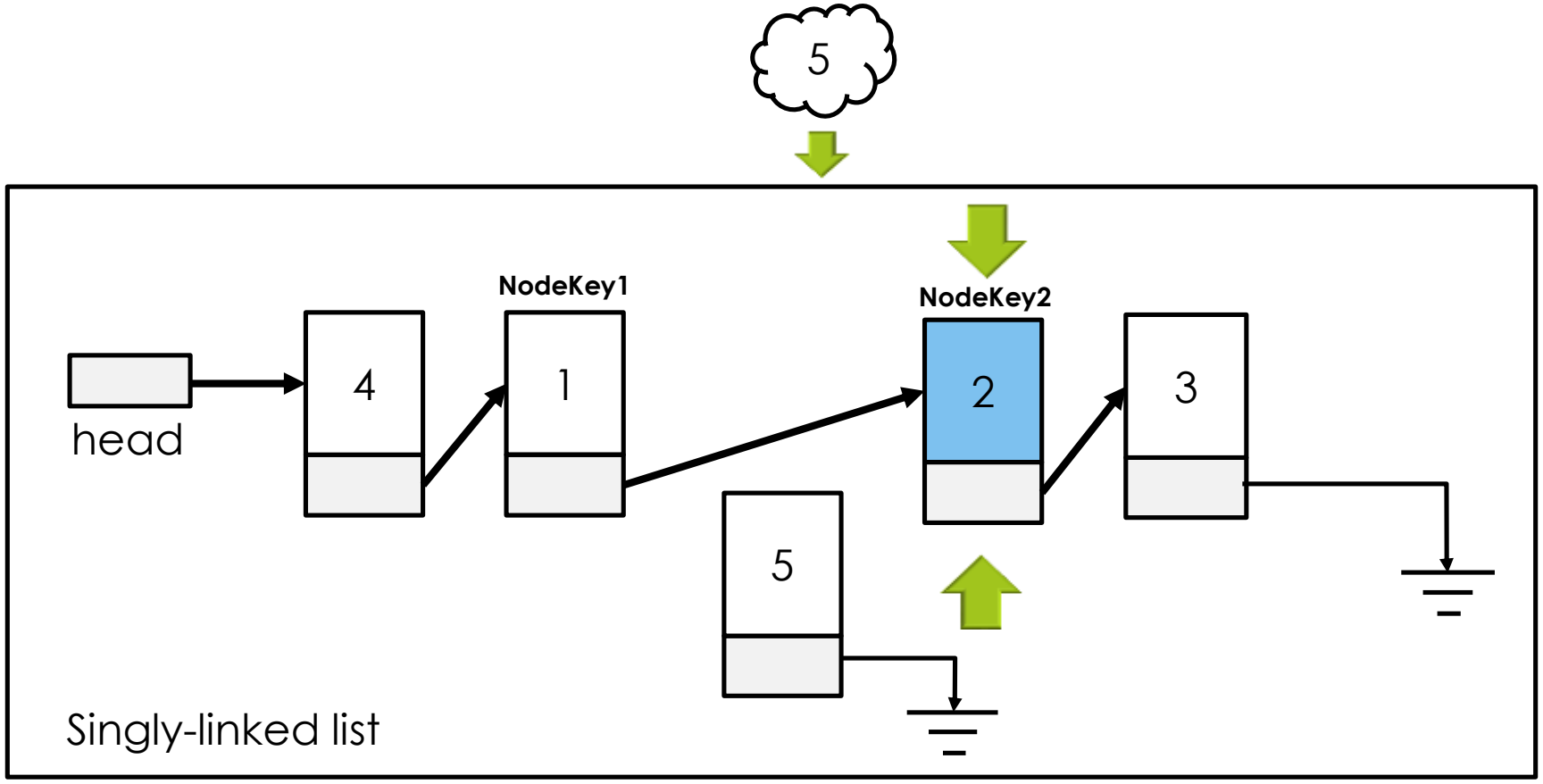
AddBefore(NodeKey2, 5) operation

- Step 1: Start from the head; Go node by node until you find the Node that points NodeKey2



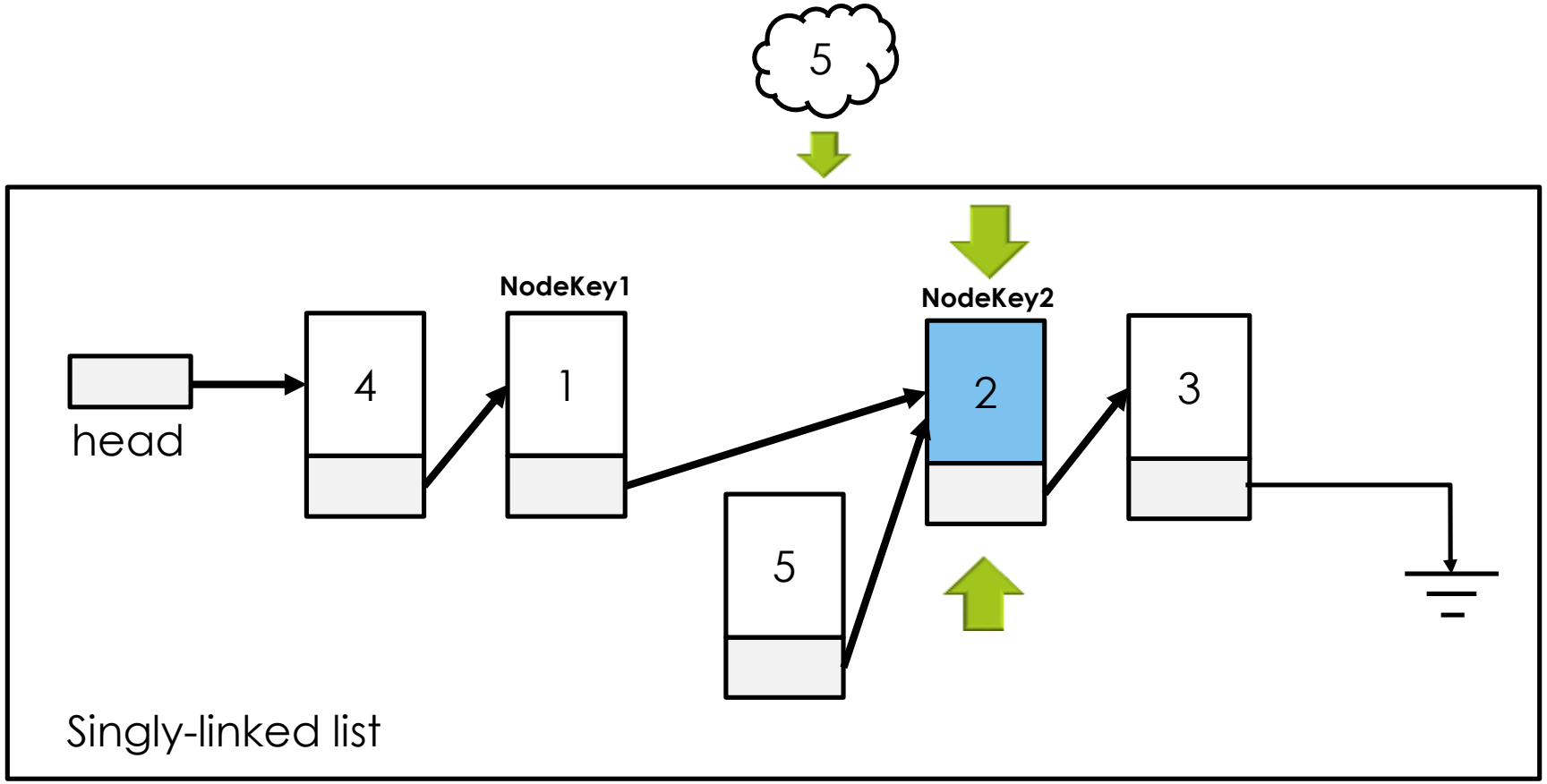
AddBefore(NodeKey2, 5) operation

- Step 2: Create a node that contains Key "5"



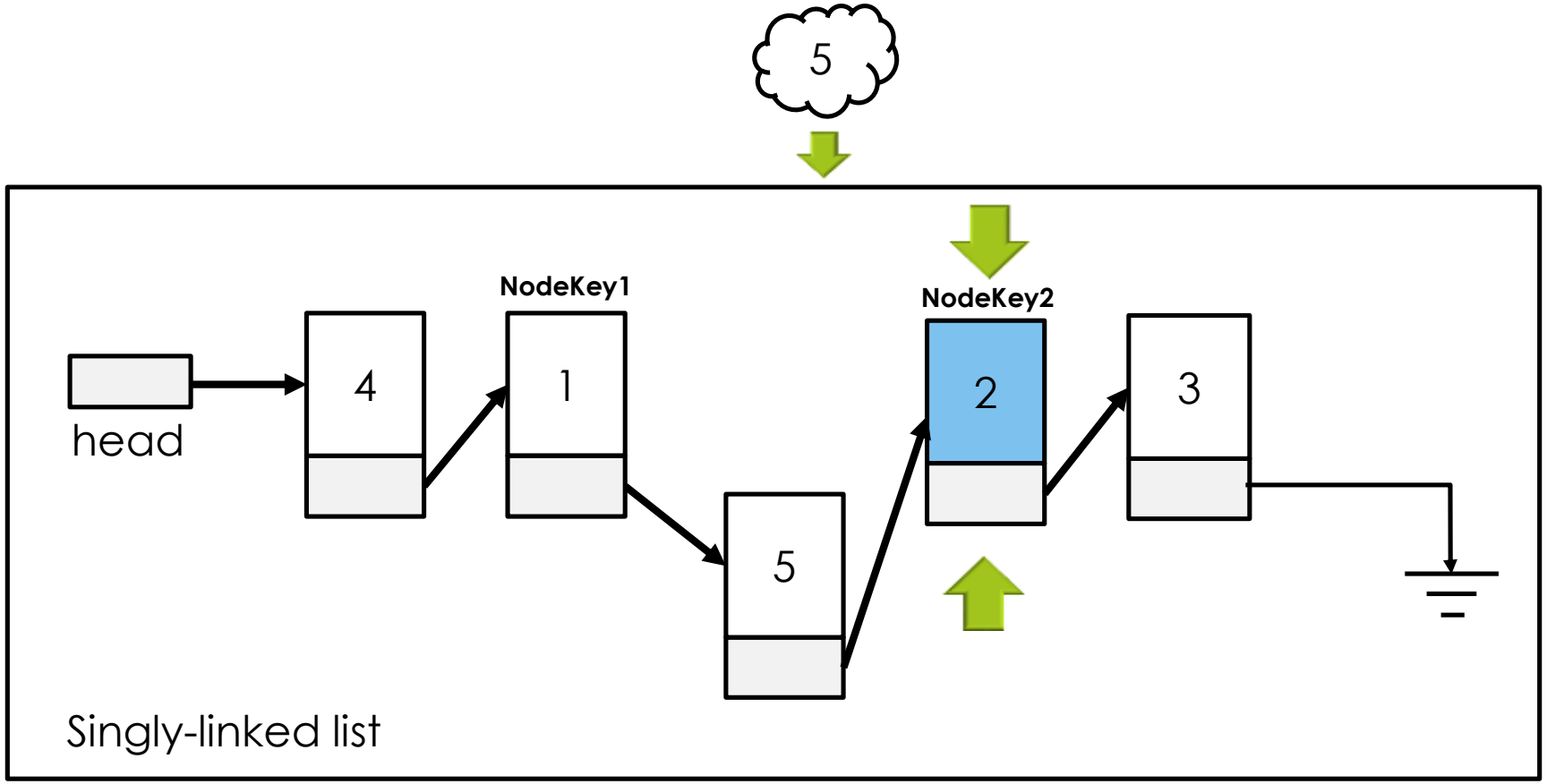
AddBefore(NodeKey2, 5) operation

- Step 3: Set pointer of the new node (NodeKey5) to point to the NodeKey2



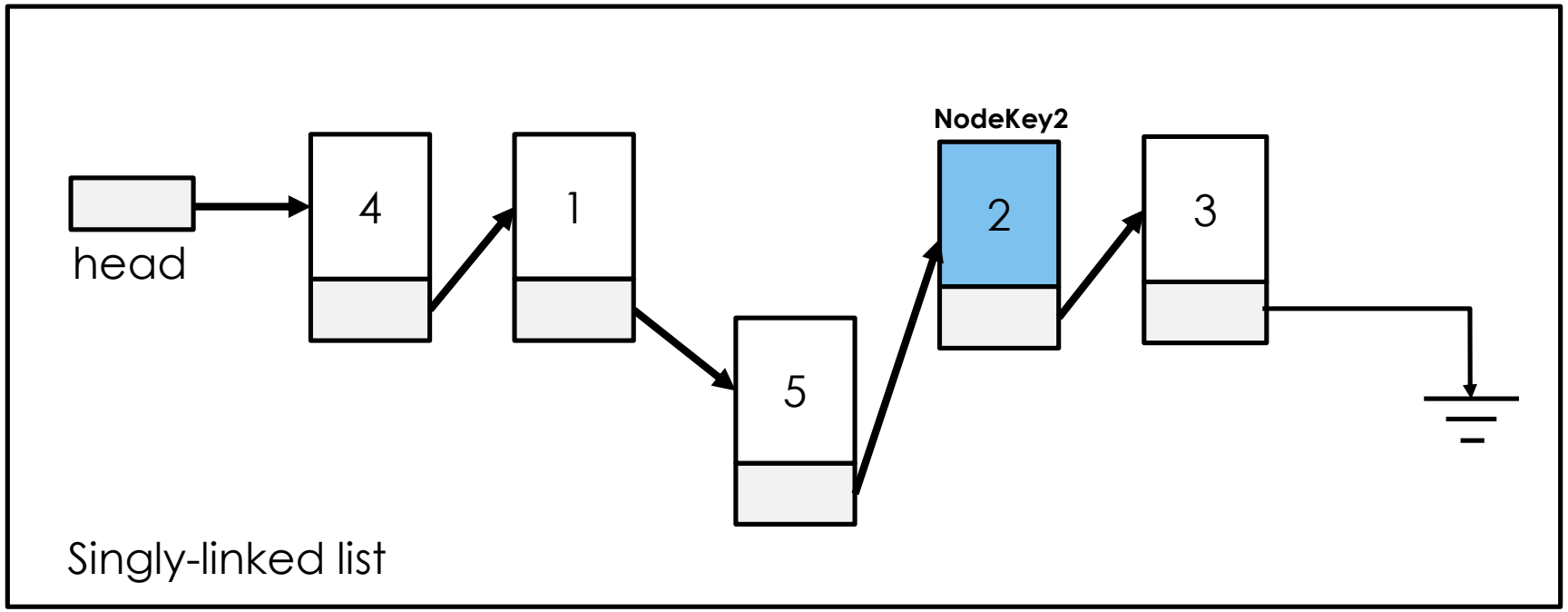
AddBefore(NodeKey2, 5) operation

- Step 4: Set pointer of the NodeKey1 to point to the new node (NodeKey5)



AddBefore(NodeKey2, 5) operation

- What is the Big O?
- Ans: O(n)



Running Time

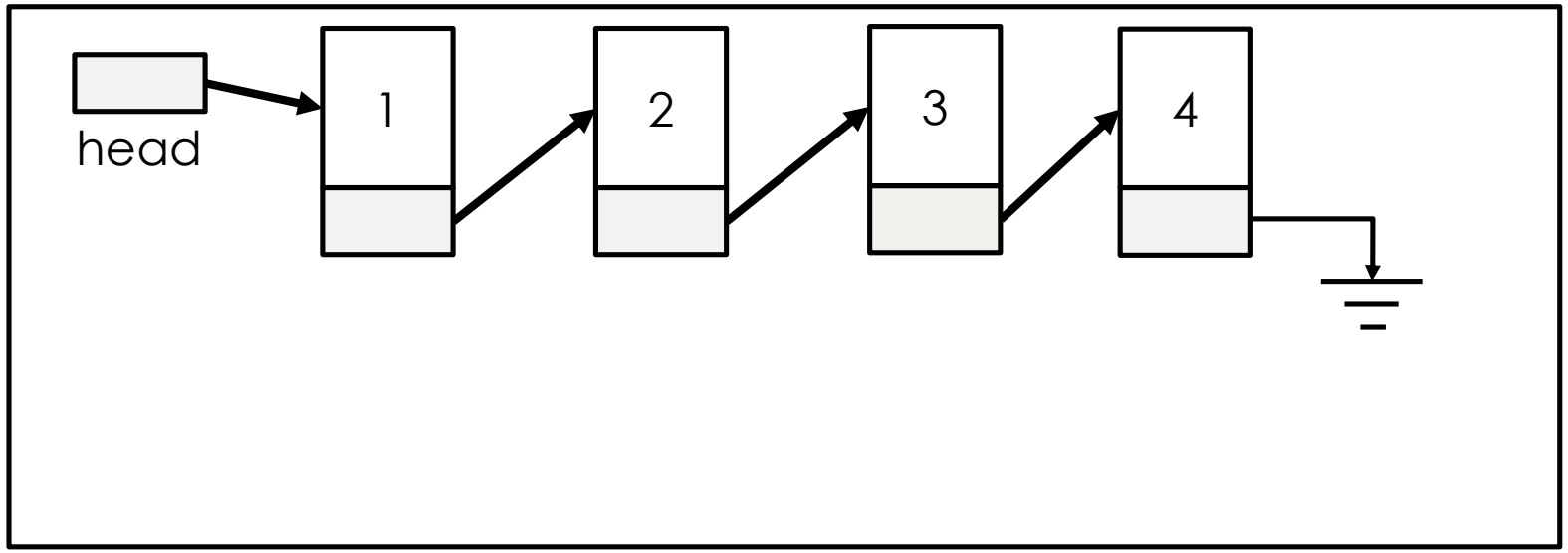
Singly-linked list operations	Running time
PushFront (Key)	O(1)
TopFront()	O(1)
PopFront()	O(1)
PushBack(Key)	O(n)
TopBack()	O(n)
PopBack()	O(n)
Find(Key)	O(n)
Erase(Key)	O(n)
Empty()	O(1)
AddBefore(Node, Key)	O(n)
AddAfter(Node, Key)	O(1)

What if you are allowed to modify the linked-list in order to improve the performance, what will you do?

How to improve PushBack, TopBack, and PopBack operations?

The current complexity is O(n)

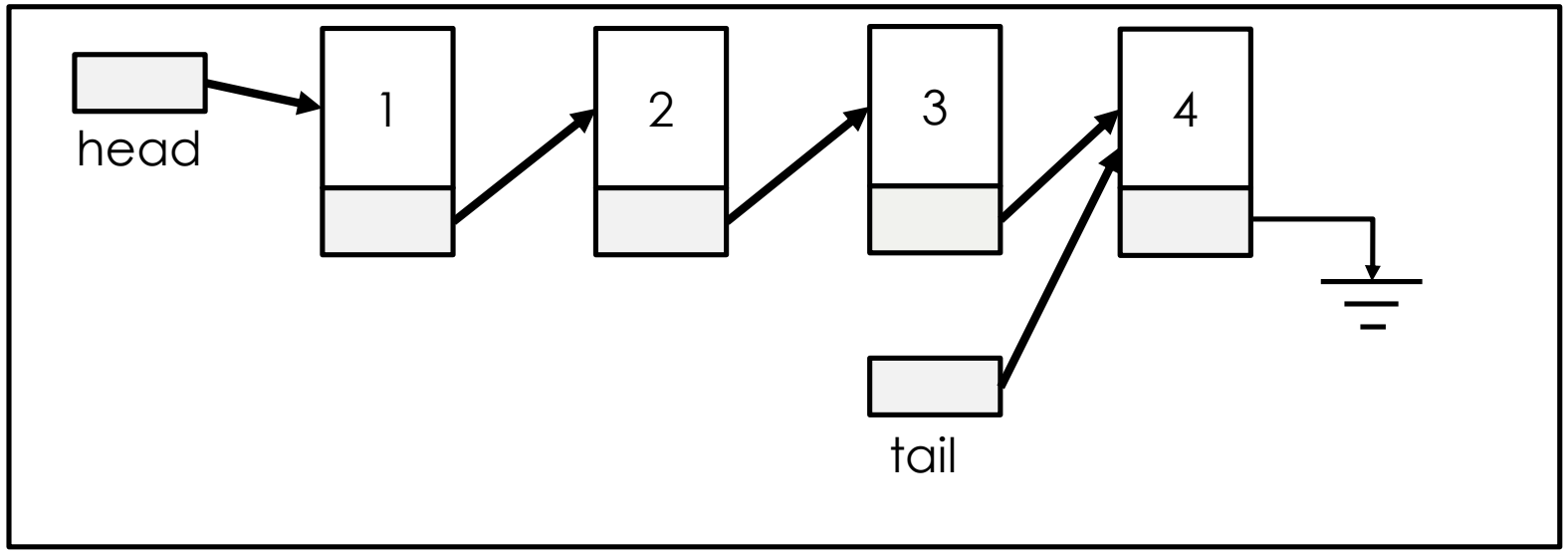
Assume you can add another variable to the structure



How to improve PushBack, TopBack, and PopBack operations?

The current complexity is O(n)

Assume you can add another variable to the structure

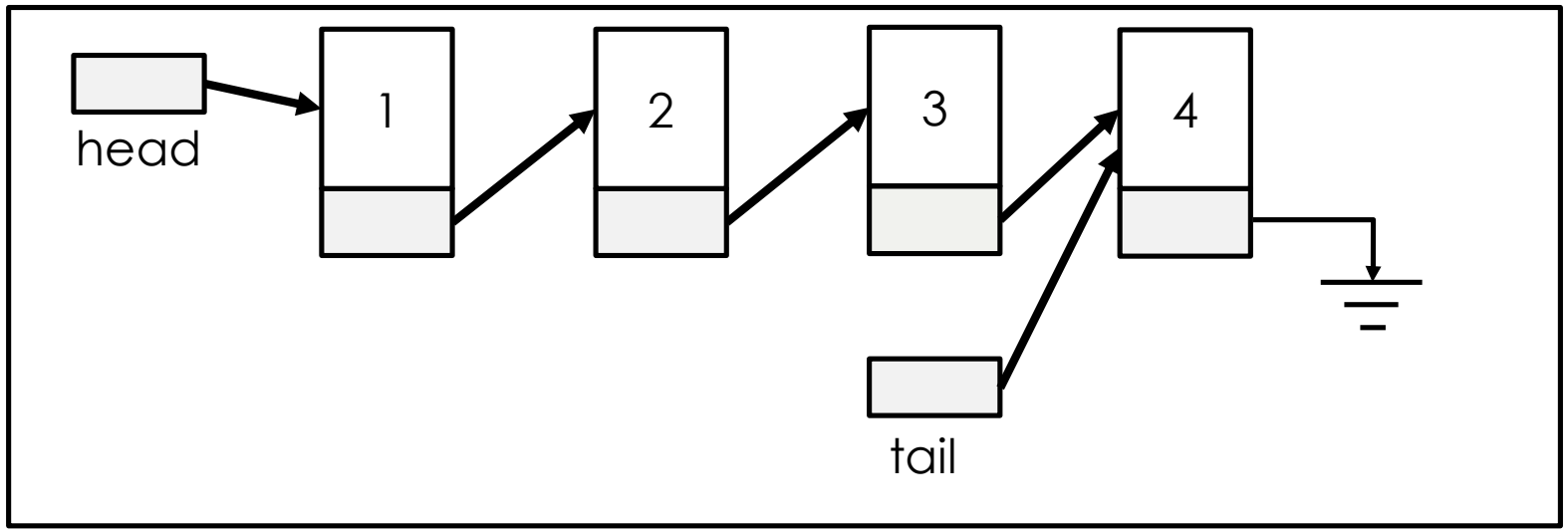


How to improve PushBack, TopBack, and PopBack operations?

PushBack(Key); Key TopBack(); PopBack();
Show Steps how these function works with ***tail*** variable?

What is the complexity of those operations after adding ***tail*** variable?

Ans: O(1)



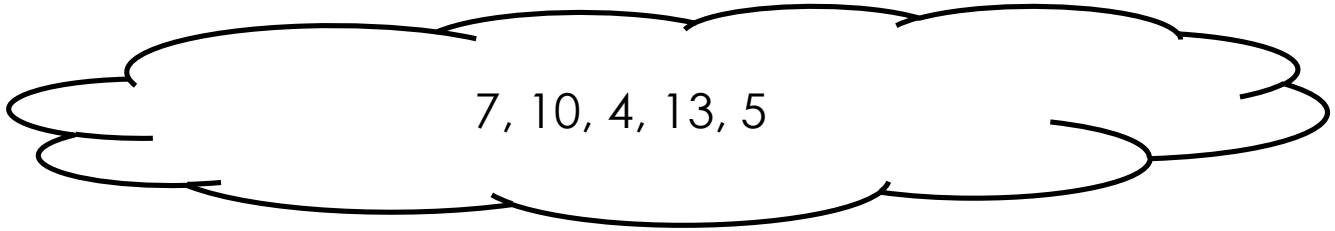
Running Time with Tail variable

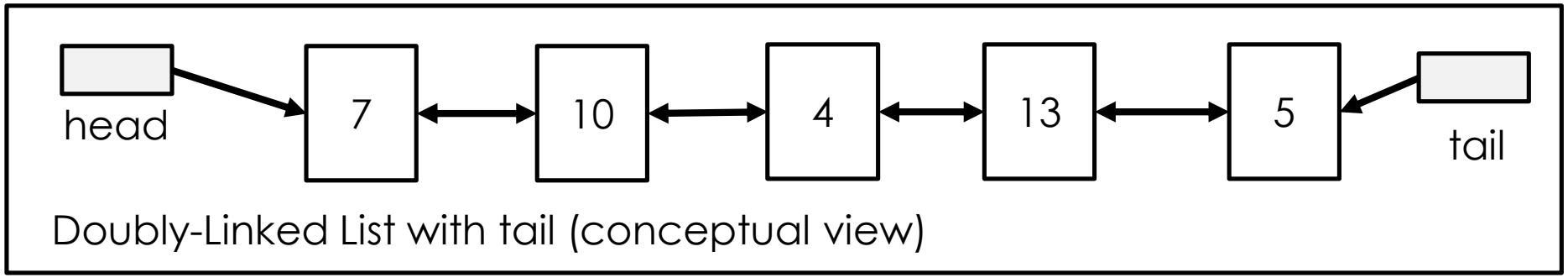
Singly-linked list operations	No tail	With tail
PushFront (Key)	O(1)	
TopFront()	O(1)	
PopFront()	O(1)	
PushBack(Key)	O(n)	O(1)
TopBack()	O(n)	O(1)
PopBack()	O(n)	O(n)
Find(Key)	O(n)	
Erase(Key)	O(n)	
Empty()	O(1)	
AddBefore(Node, Key)	O(n)	?
AddAfter(Node, Key)	O(1)	

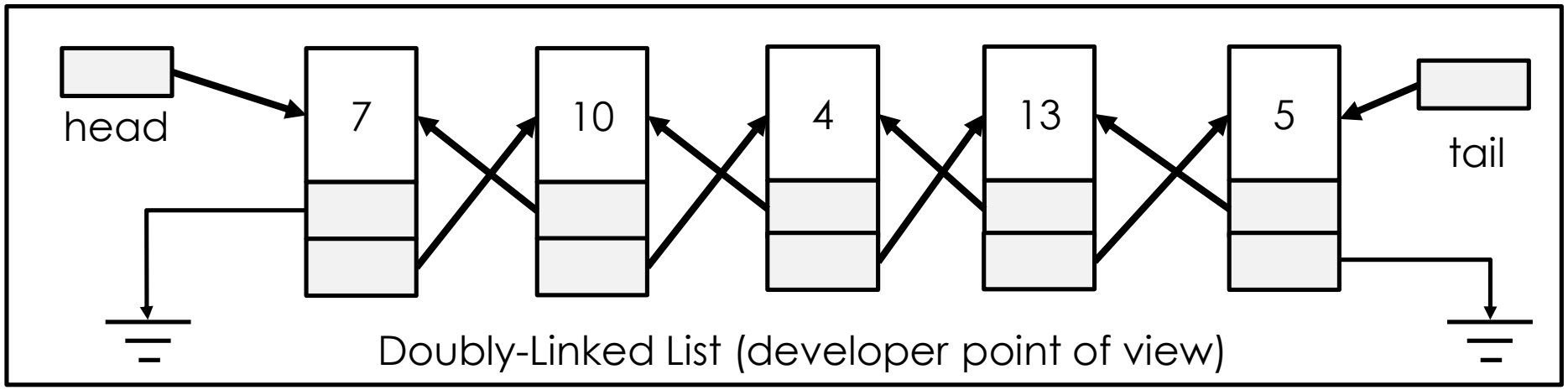
Problem with Singly-linked list

- One-way traveling
  - Go right only
  - Given a node, we always know who is on the right
  - So AddAfter is super fast
  - But AddBefore is much slower because we do not know who is on the left (we must start from the beginning)
- Solution?
- Doubly-linked list

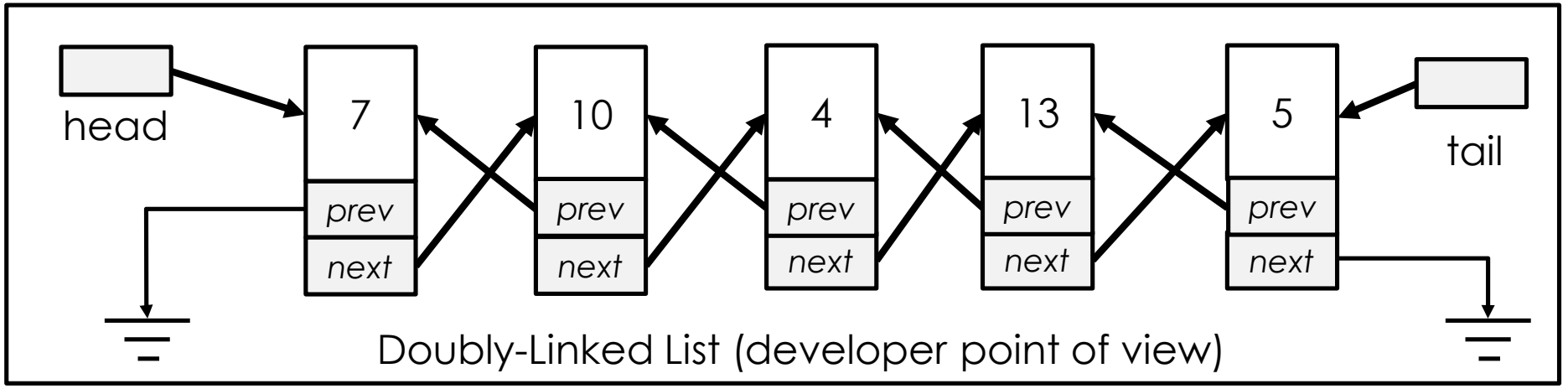
Doubly-linked List with Tail







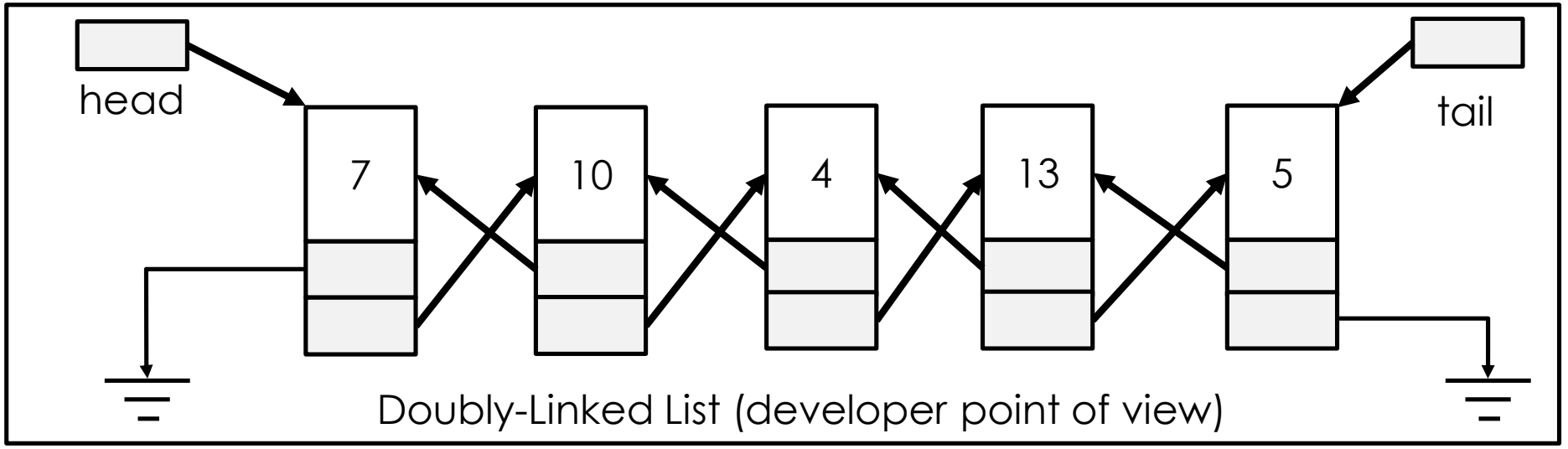
Doubly-linked List with Tail



Node contains

- ***key***
- ***next*** pointer
- ***prev*** pointer

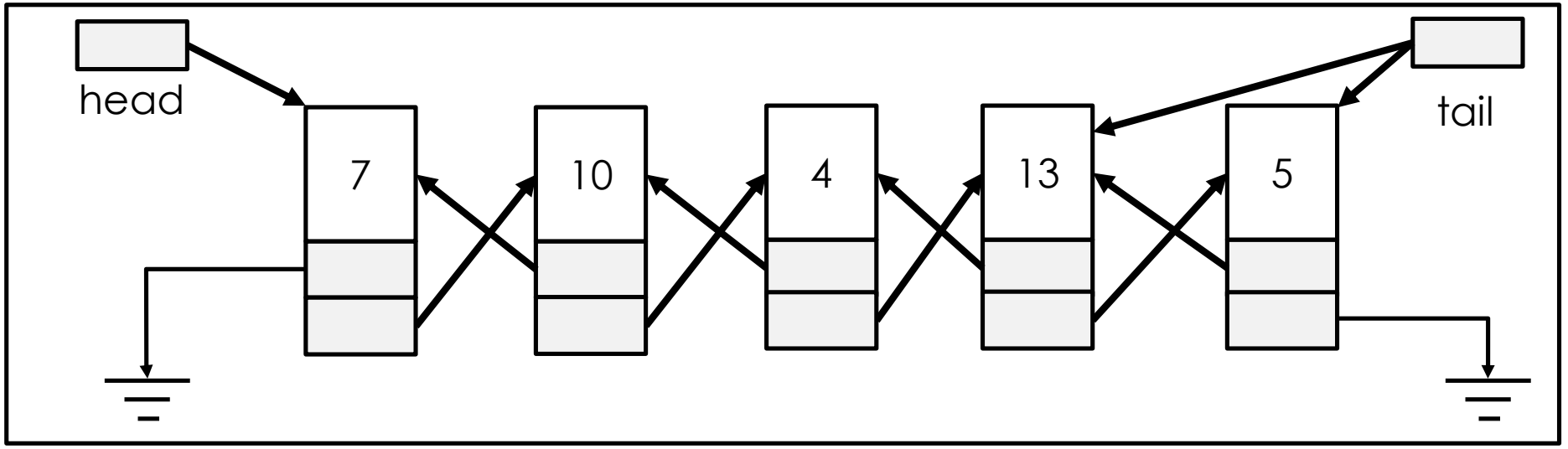
PopBack() in Doubly-linked List with Tail



PopBack() removes the last node

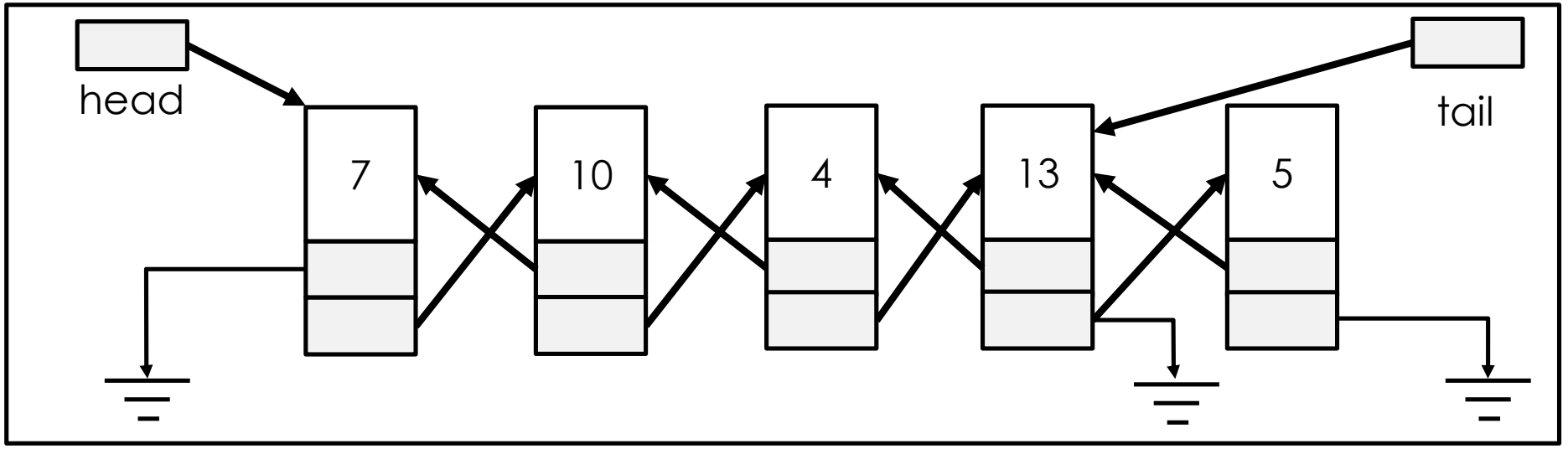
In the singly-linked list, the operation cost O(n)
How about the doubly-linked list?

PopBack() in Doubly-linked List with Tail



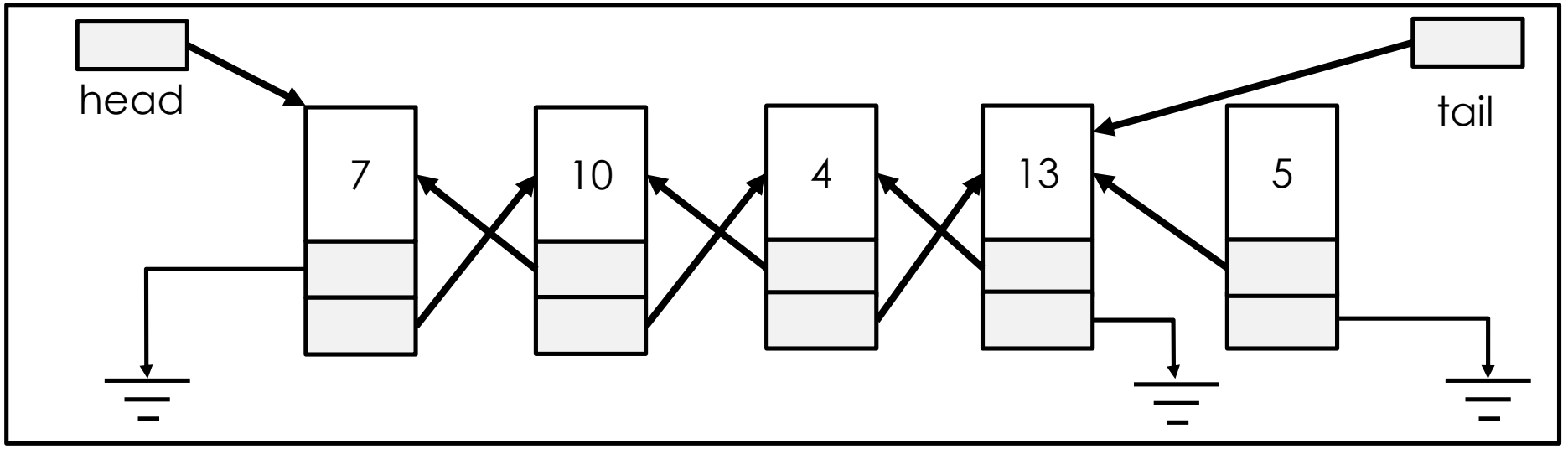
Step 1: Set ***tail*** to the node that the *prev* pointer of the last node points to

PopBack() in Doubly-linked List with Tail



Step 2: Set *prev* pointer of the node pointed by the ***tail*** to null

PopBack() in Doubly-linked List with Tail



Step 3: Delete the last node

This process is automatically done in C#

For other languages such as C++, you may create temp pointer to point to the last node in the step 2, then you can use "delete temp" command

What can be improved if you use doubly-linked lists instead

Singly-linked list operations	No tail	With tail
PushFront (Key)	O(1)	
TopFront()	O(1)	
PopFront()	O(1)	
PushBack(Key)	O(n)	O(1)
TopBack()	O(n)	O(1)
PopBack()	O(n)	O(n)
Find(Key)	O(n)	
Erase(Key)	O(n)	
Empty()	O(1)	
AddBefore(Node, Key)	O(n)	
AddAfter(Node, Key)	O(1)	

What can be improved if you use doubly-linked lists instead

Doubly-linked list operations	No tail	With tail
PushFront (Key)	O(1)	
TopFront()	O(1)	
PopFront()	O(1)	
PushBack(Key)	O(n)	O(1)
TopBack()	O(n)	O(1)
PopBack()	O(n)	O(1)
Find(Key)	O(n)	
Erase(Key)	O(n)	
Empty()	O(1)	
AddBefore(Node, Key)	O(1)	
AddAfter(Node, Key)	O(1)	

Summary

- Constant time to insert at or remove from the front.
- With tail and doubly-linked, constant time to insert at or remove from the back.
- O(n) time to find arbitrary element.
- List elements need not be contiguous.
- With doubly-linked list, constant time to insert between nodes or remove a node.

Demo

- Singly-linked list, pushFront(X), printHead2Tail()
- Your job is to do the rest for singly-linked list and doubly-linked list

Demo

- Array of Objects (Demo X1)
- List of Objects (Demo X2)
- Object consists of student_ID, GPA, name
- Find the top GPA student; Show student_ID and Name

Student_ID (string)	Name (string)	GPA (double)
5906001	Matthew	3.50
5906002	Mark	2.75
5906003	Luke	3.00
5906004	John	3.75
5906005	James	3.25