Chapter 5: Linked Lists

Dr. Sirasit Lochanachit

Overall

Complexity

Recursive

Arrays

Linked Lists

Shortest Path Algorithm

Stacks

Searching

Queues

Sorting

Data Structure

Algorithm

Trees

Balancing

Graphs

Hashing
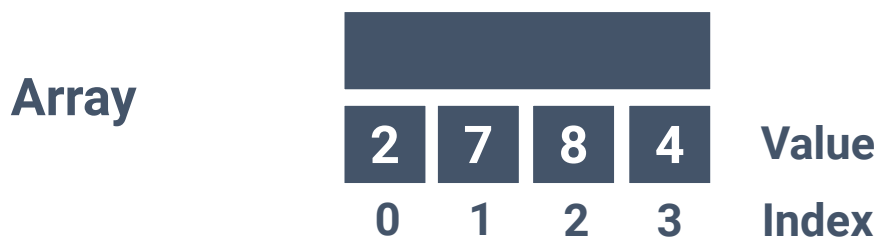
Heaps

Dynamic Programming
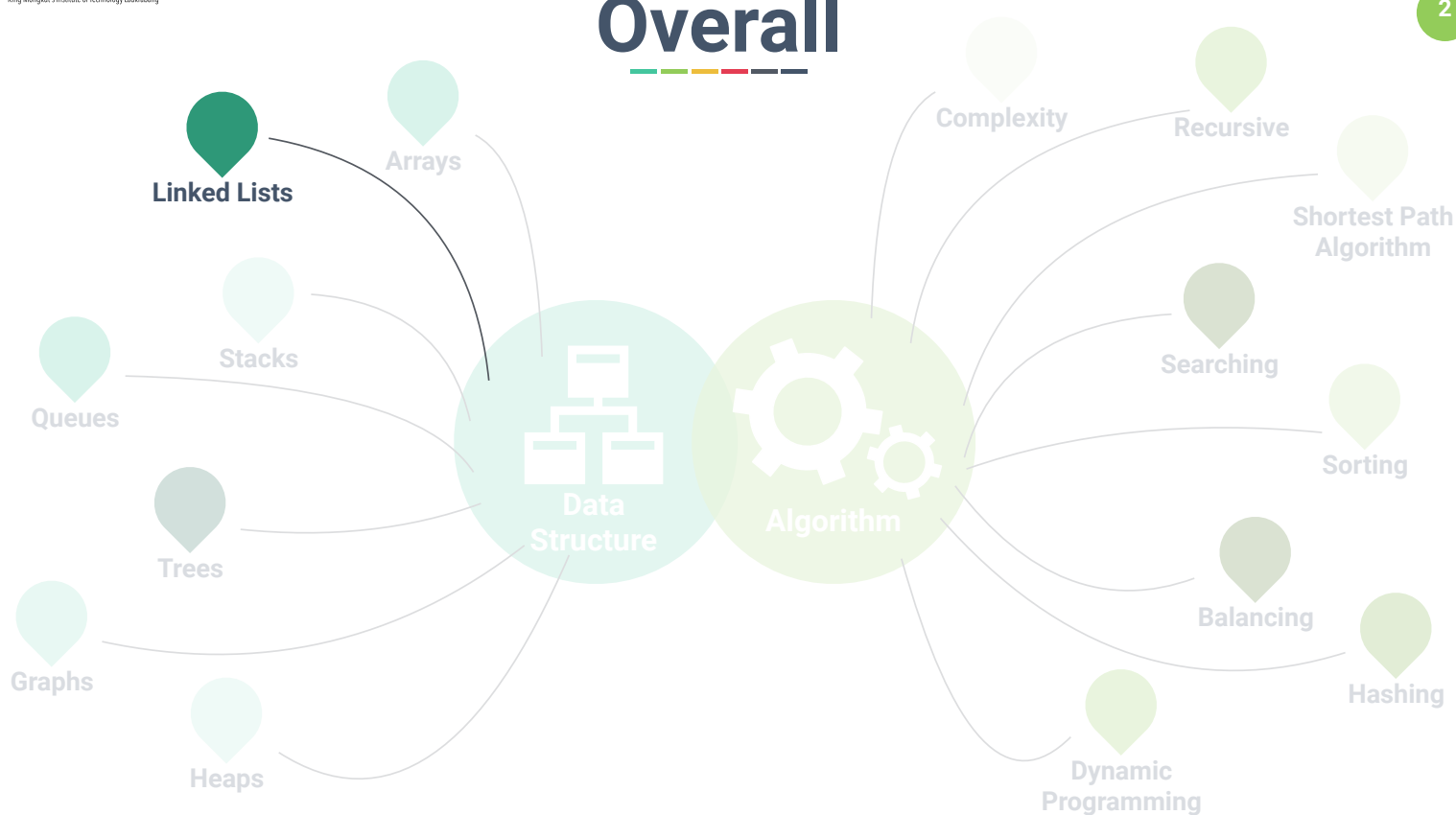
Linked Lists

Disadvantages of array:

- Length of array has to be pre-allocated, empty space wasted.

- Adding or removing elements between values in the array is expensive - $O(n)$

Linked Lists

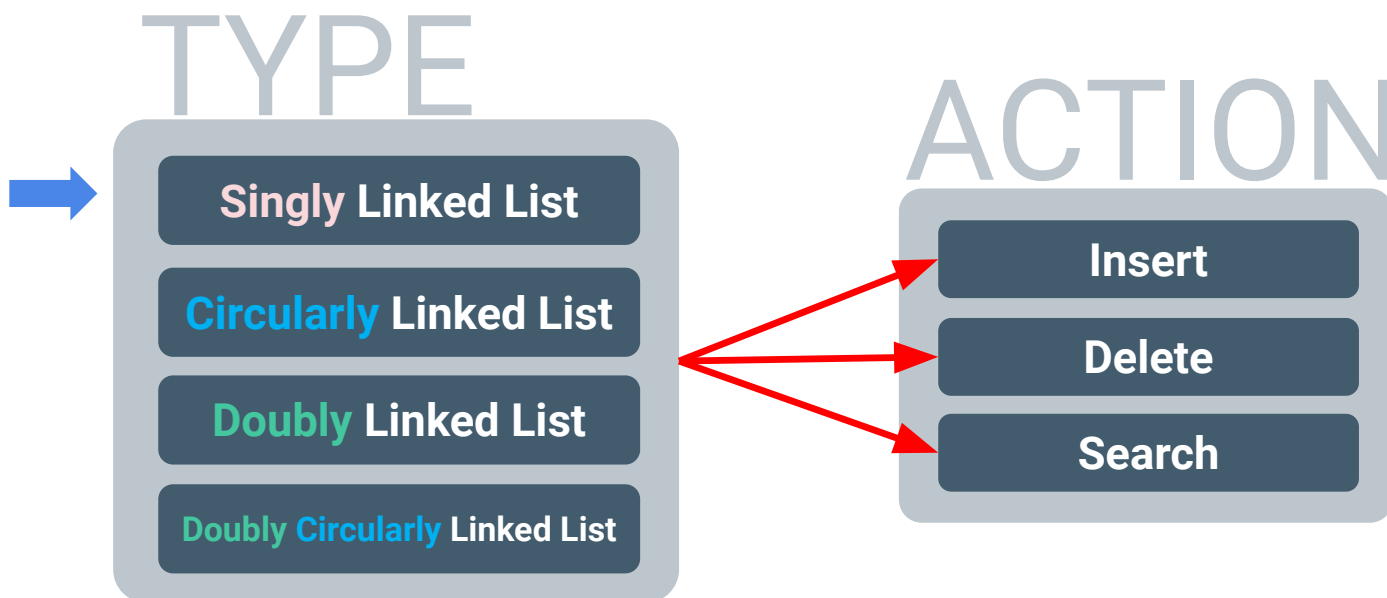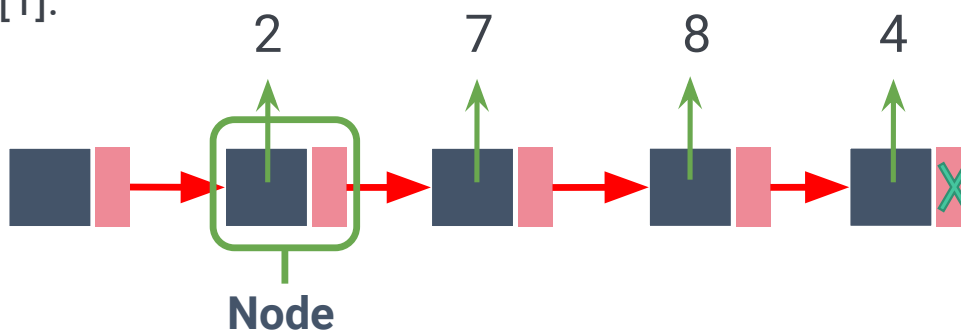To avoid these limitations, an alternative to array is **linked list**.

Linked Lists

TYPE

- **Singly** Linked List
- **Circularly** Linked List
- **Doubly** Linked List
- **Doubly Circularly** Linked List

ACTION

- **Insert**
- **Delete**
- **Search**

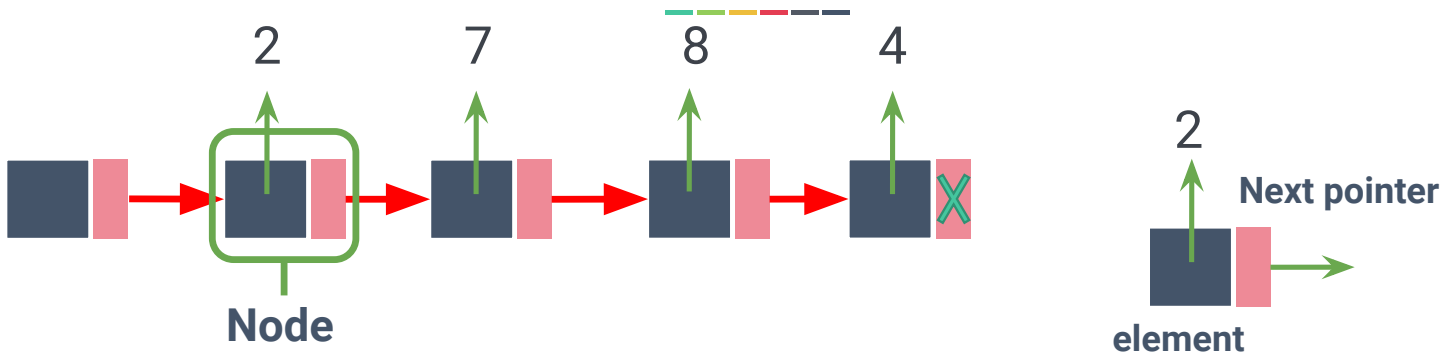# Singly Linked Lists

A singly **linked list** is a collection of nodes that form a linear order of a sequence [1].

2   7   8   4

Node

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Singly Linked Lists
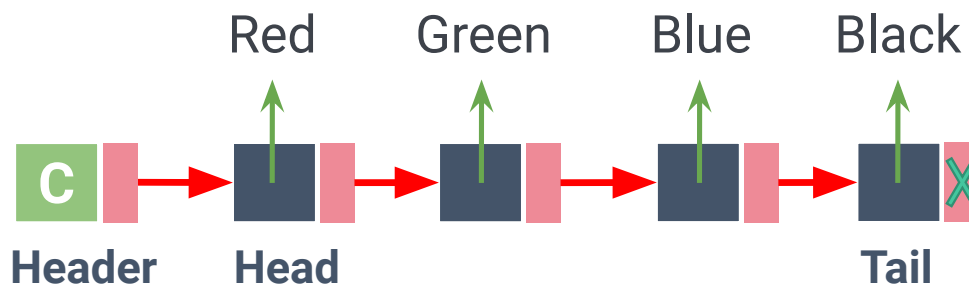
2   7   8   4

2

Next pointer

Node

element

Each node keeps:

- A reference to an object/value which is its element.

- **Link/Pointer**: One or more references to adjacent nodes or subsequent nodes.

    - Reference to **None** if there is no further node.

Singly Linked Lists

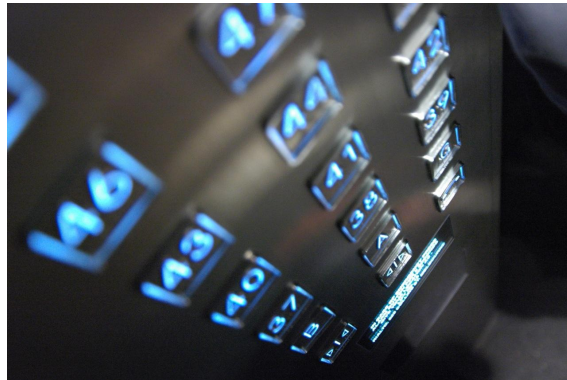Red   Green   Blue   Black

C

Header   **Head**   **Tail**

- **Head** and **tail** identify the first and last node, respectively.

- **Header** node can contain a counter to keep track the number of nodes that form a list.

Linked Lists

Real-life examples of Linked Lists:

Singly Linked Lists

C → Red → Green → Blue → Black X

Head   **Tail**

For simplicity, the linked list illustration will embed element within the node.

Note that each node still contains a reference to the element, not the element itself directly.

Singly Linked Lists

Head

Tail

- **Traversing** or **link hopping** is the process of moving from one node to another according to each node's subsequent pointer.
- Linked Lists provides **sequential access** only.
  - Locating the element in a linked list requires $O(n)$ time to traverse the list from the beginning.

Singly Linked Lists

Address/ Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Suppose that it takes 1 byte to store an integer.

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Counter
Next

Header

C → 2 → 7 → 8 → 4 X

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Counter
Next

Header
Head

C → 2 → 7 → 8 → 4 X

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Counter
Next

Header
Head

C → 2 → 7 → 8 → 4 X

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Counter
Next

Header
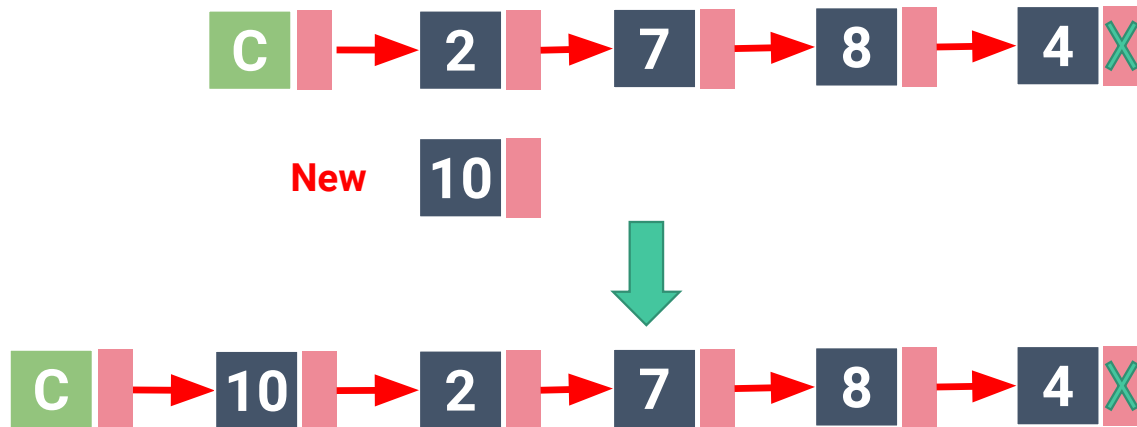**Head**

C → 2 → 7 → 8 → 4 X

Tail

Singly Linked Lists

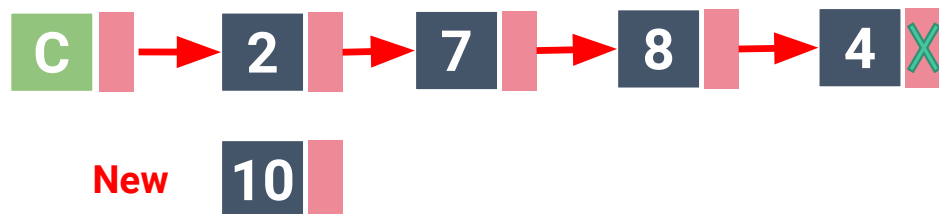**Insert** at the first node

Singly Linked Lists

Insert at the first node

Step 1: Create a new node storing reference to an element.

Step 2: Set new node's next pointer to the <u>current/old head</u>.

Step 3: <u>Set the list's head</u> to reference the new node.

Step 4: Increment the node count.

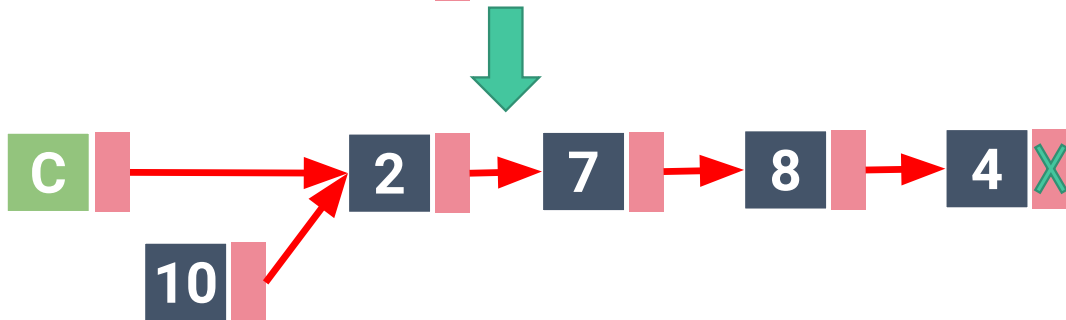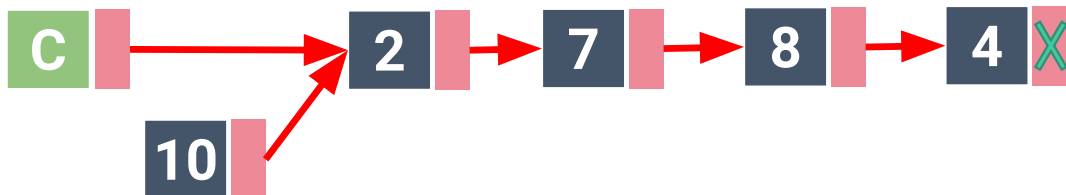# Singly Linked Lists

Insert at the first node

Step 1

**New** 10

Step 2

Singly Linked Lists

Insert at the first node

Step 2

Step 3 & 4

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Step 1

Create a new node

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Step 2, 3 & 4

Set Pointers

Address/Byte#	Value
6000	5
6001	6006
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	6002
6008	7
6009	6004
6010	
6011	
6012	4
6013	None
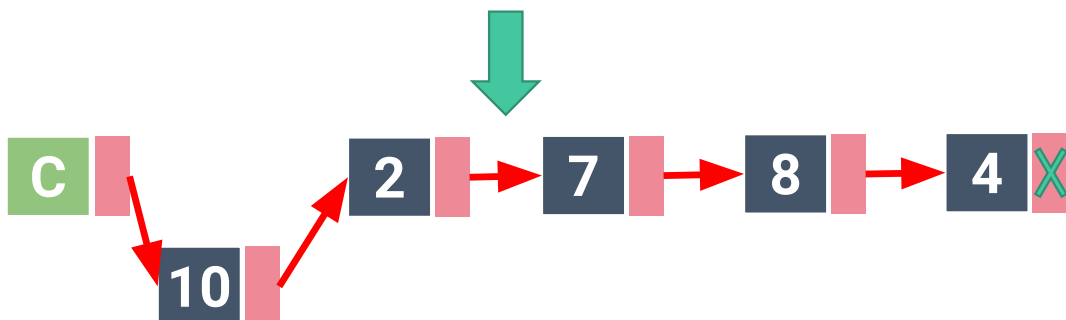
Singly Linked Lists

Insert at the first node

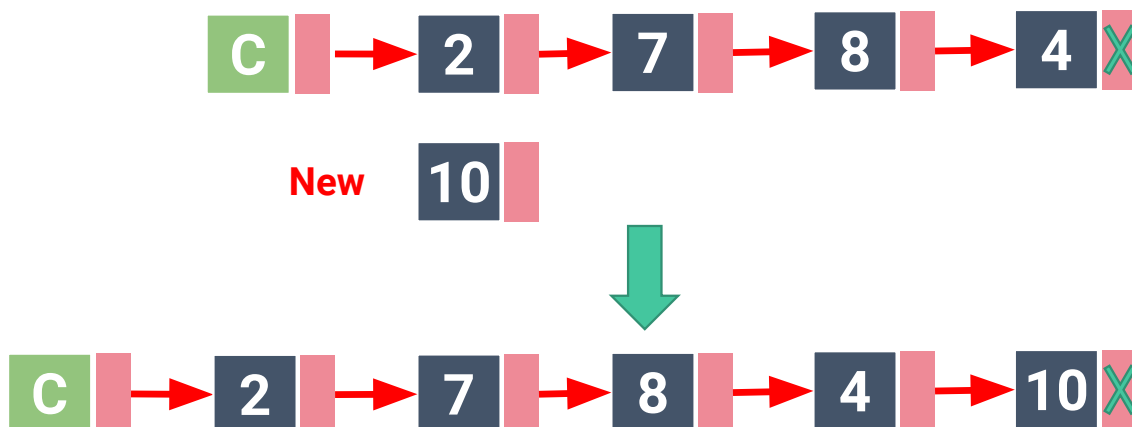Algorithm add_front(L, e):

new_node = Node(e)   # Create new node instance

new_node.next = L.head   # Set new node's next pointer to the old head

L.head = new_node   # Update the list's head to reference the new node

L.size = L.size + 1   # Increment the node count

if L.tail == None: # List was empty

    L.tail = L.head
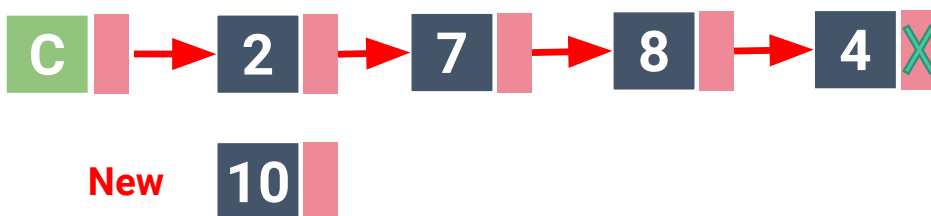
$O(1)$

Singly Linked Lists

Insert at the last node

Singly Linked Lists

Insert at the last node

Step 1: Create a new node storing reference to an element.

Step 2: Set new node's next pointer to <u>None</u>.

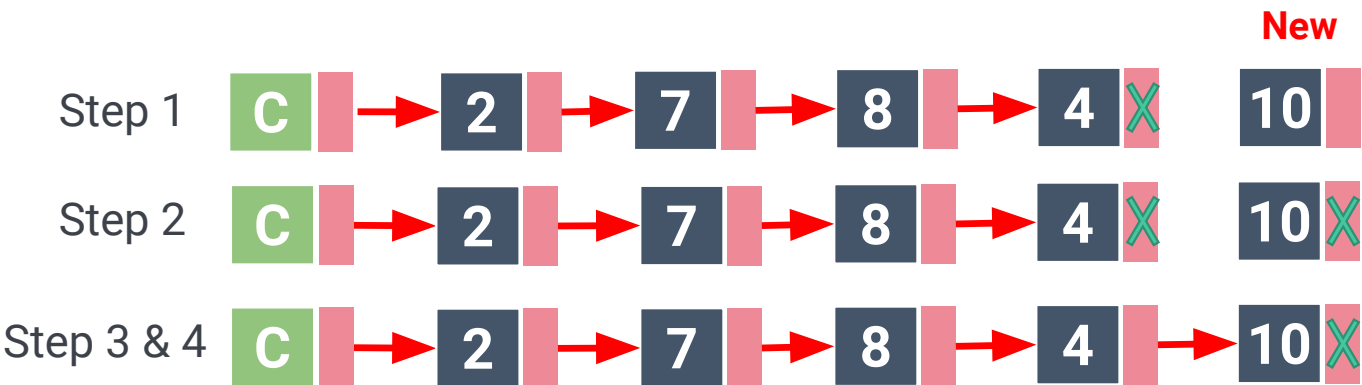Step 3: <u>Update the list's tail</u> to reference the new node.

Step 4: Increment the node count.
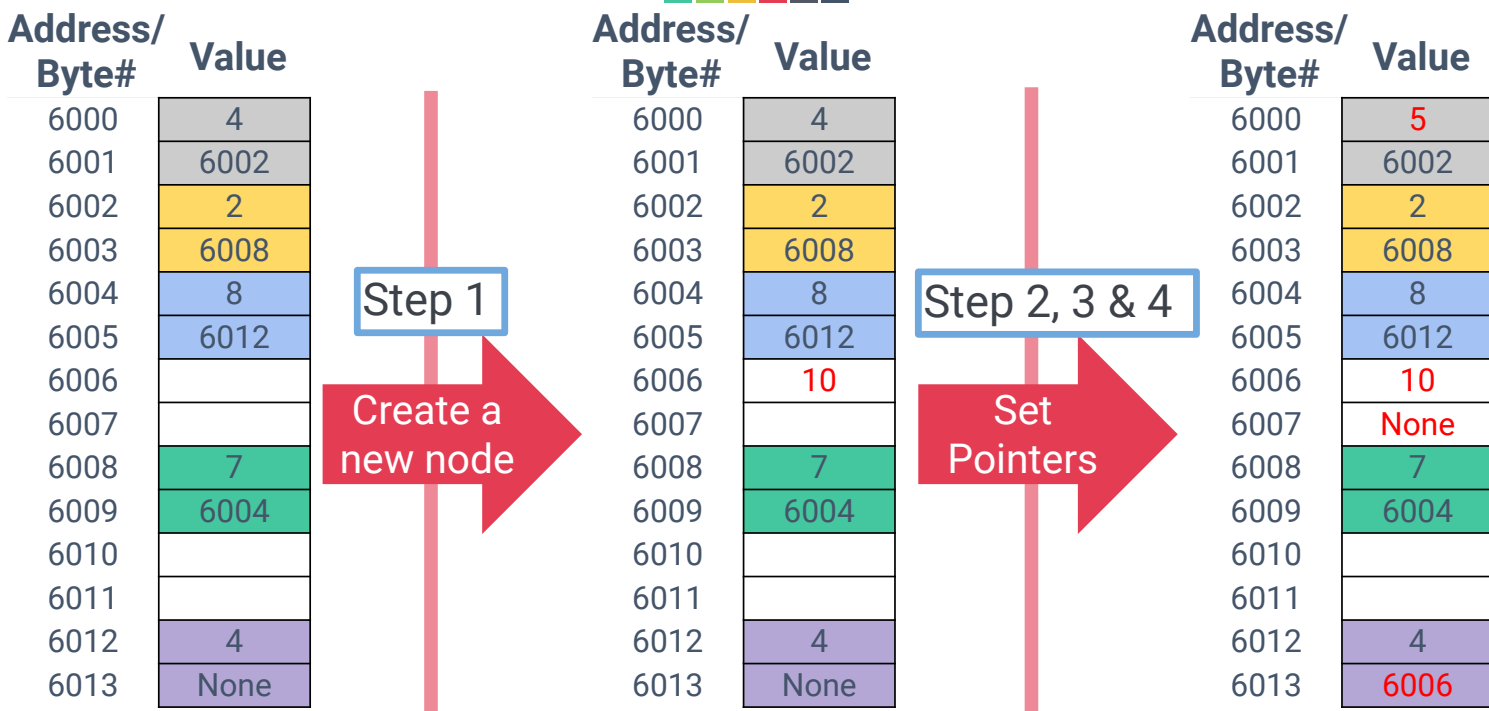
O(1)

Singly Linked Lists

Insert at the last node

New

Step 1: C → 2 → 7 → 8 → 4 ✗   10

Step 2: C → 2 → 7 → 8 → 4 ✗   10 ✗

Step 3 & 4: C → 2 → 7 → 8 → 4 → 10 ✗

Singly Linked Lists

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Step 1 Create a new node →

Address/Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Step 2, 3 & 4 Set Pointers →

Address/Byte#	Value
6000	5
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	6006

Singly Linked Lists

Insert   at the last node

```
Algorithm add_last(L, e):

    new_node = Node(e)              # Create new node instance

    new_node.next = None            # Set new node's next pointer to None

    if L.tail == None:              # List was empty

        L.head & L.tail = new_node

    else:

        L.tail.next = new_node      # Make old tail point to new node

        L.tail = new_node           # Update the list's tail to reference the new node

    L.size = L.size + 1             # Increment the node count
```
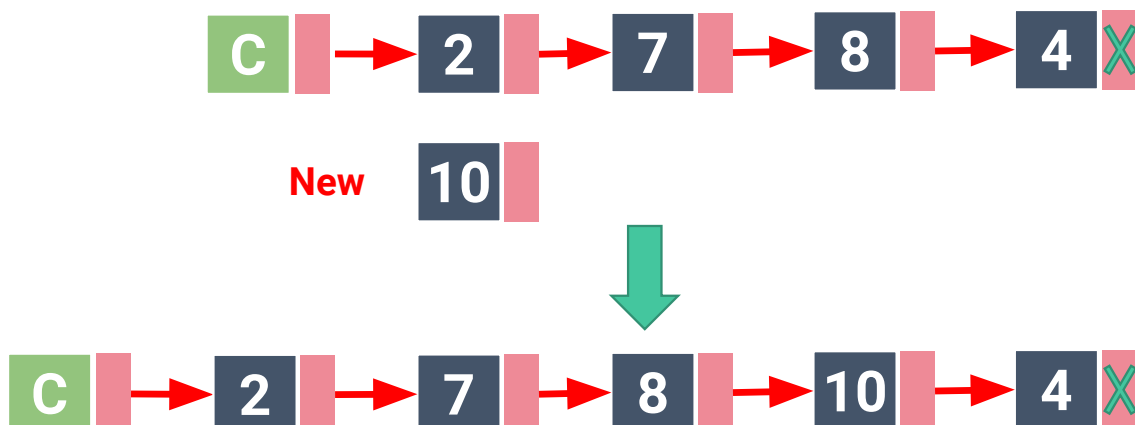
$O(1)$

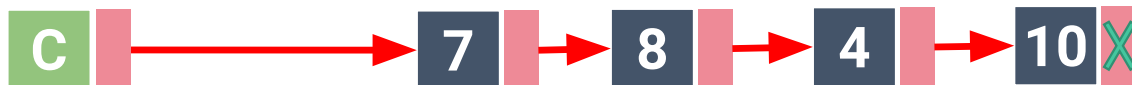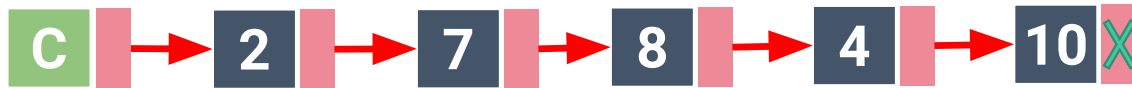# Singly Linked Lists

Insert   between nodes

Only update the pointers of neighbouring nodes.

Singly Linked Lists

Delete at the first node

C → 2 → 7 → 8 → 4 → 10 ✗

⬇

C → 7 → 8 → 4 → 10 ✗

Singly Linked Lists

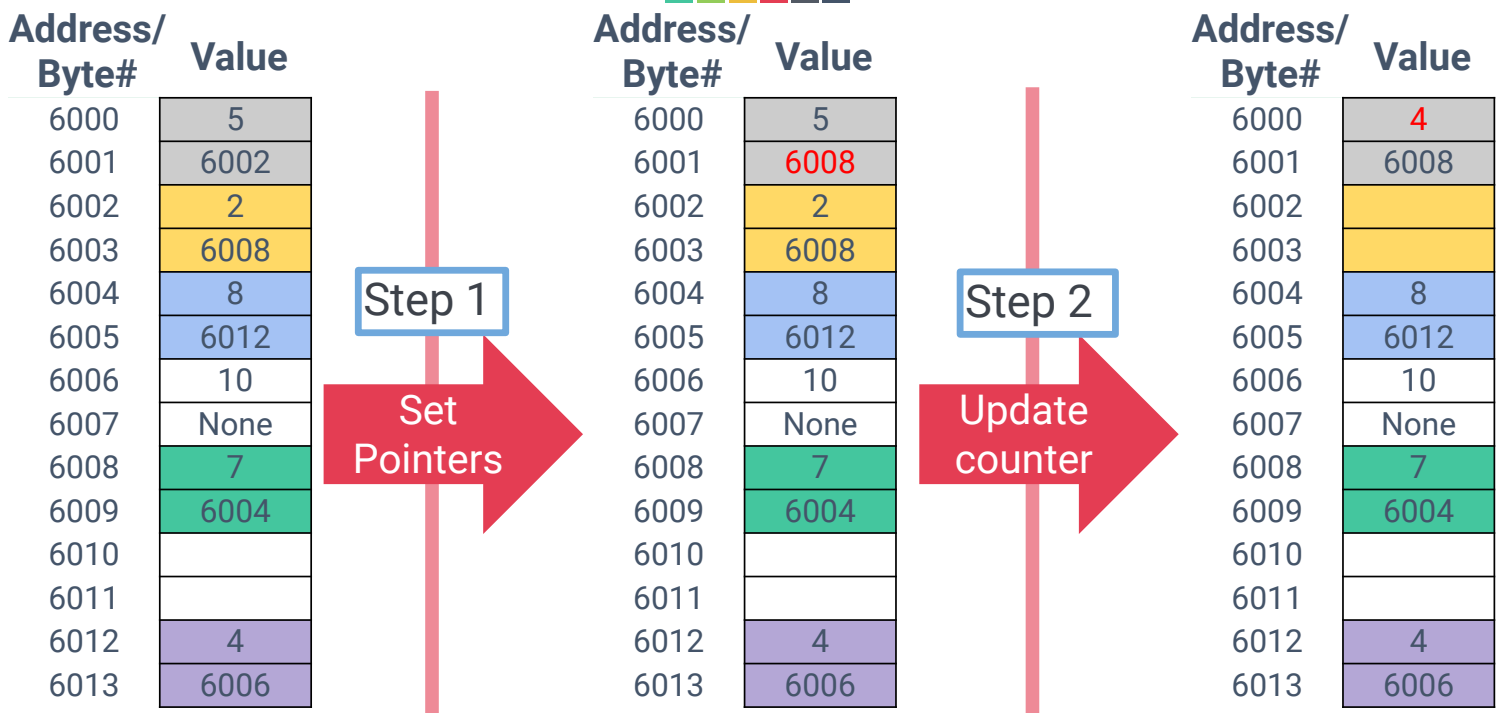**Delete** at the first node

C → 2 → 7 → 8 → 4 → 10 ✗
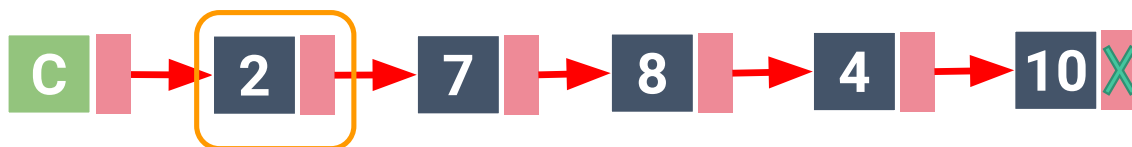
Step 1: Set head node's next pointer to the subsequent node.

- If head is **None**, then the list is empty, return error.

Step 2: Decrement the node count.

$O(1)$

Singly Linked Lists

Address/ Byte#	Value
6000	5
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	6006

Step 1 — Set Pointers →

Address/ Byte#	Value
6000	5
6001	6008
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	6006

Step 2 — Update counter →

Address/ Byte#	Value
6000	4
6001	6008
6002	
6003	
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	6006

Singly Linked Lists

Delete **at the first node**

Algorithm remove_first(L):

    if L.head == None: # List is empty

        return Error

    L.head = L.head.next   # Make head point to next node or None if empty

    L.size = L.size - 1   # Decrement the node count

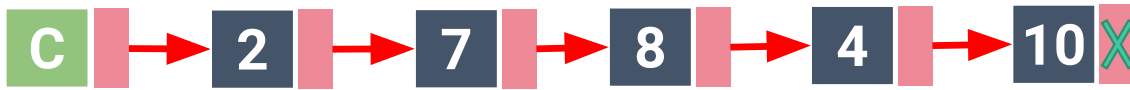   if L.head == None: # List is empty after first node is removed

        L.tail = None

$O(1)$

Singly Linked Lists

Delete at the last node

C → 2 → 7 → 8 → 4 → 10 X

⬇

C → 2 → 7 → 8 → 4 X

Singly Linked Lists

Delete at the last node

C → 2 → 7 → 8 → 4 → 10 X

⬇

C → 2 → 7 → 8 → 4 X   10 X

Step 1: Find the next to last node, then update the next pointer to **None**.

Step 2: Decrement the node count.
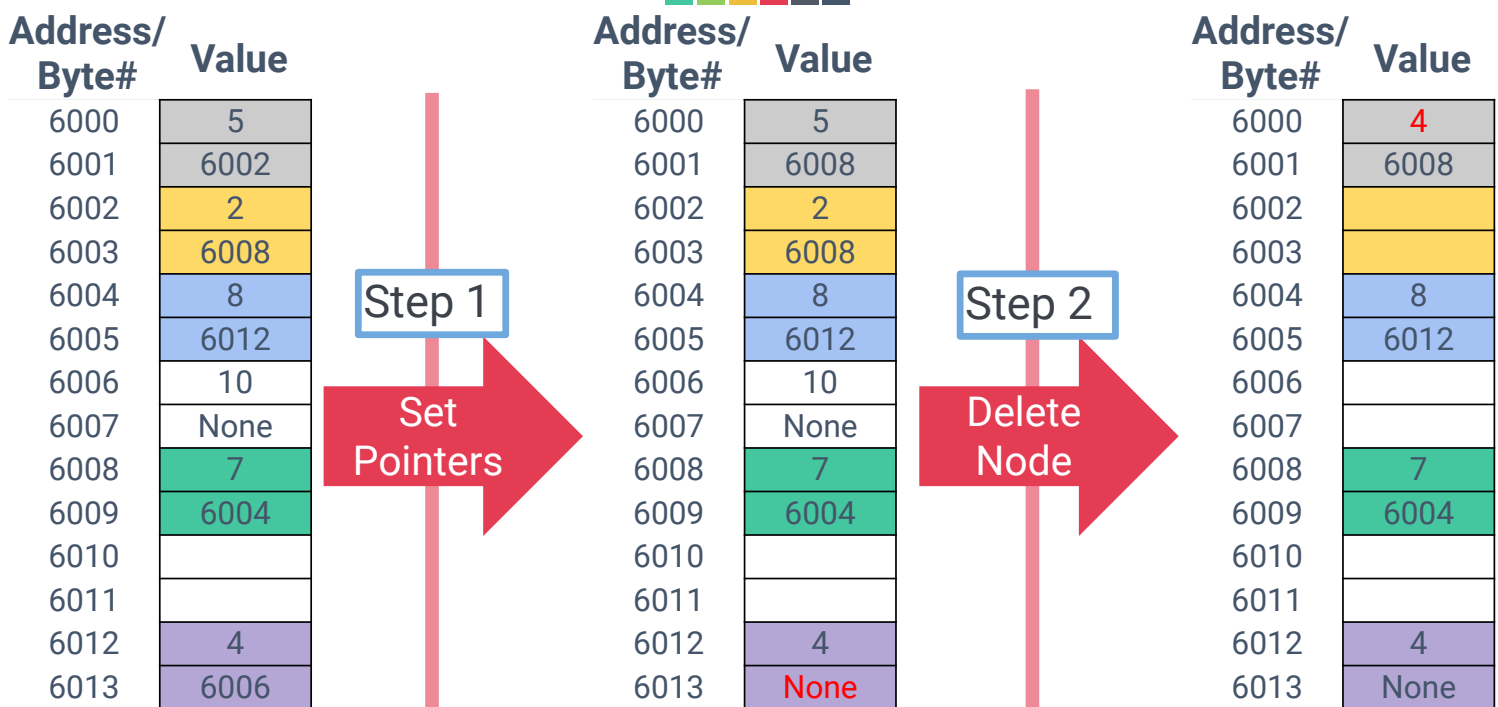
O(n) - why?

Singly Linked Lists

Deletion of the last node in Singly Linked Lists:

- No direct link from the tail node to the node before the tail.
  - There is only a link from the node before the tail to the tail node.
- To access the node before the tail, need to start from the head node and search through the list - $O(n)$.
- To address this problem, **doubly linked list** is proposed as an alternative to singly linked list. also keeps links in backward direction.

Singly Linked Lists

Address/Byte#	Value
6000	5
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	6006

Step 1 → Set Pointers

Address/Byte#	Value
6000	5
6001	6008
6002	2
6003	6008
6004	8
6005	6012
6006	10
6007	None
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Step 2 → Delete Node

Address/Byte#	Value
6000	4
6001	6008
6002	
6003	
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	None

Singly Linked Lists

Delete   at the last node

Algorithm remove_last(L):

      if L.head == None: return Error   # List is empty

      if L.head == L.tail: # List has one node

            L.head & L.tail = None

      else:

            p = L.head   # Initialise pointer to traverse the list

            while p.next.next != None: # Traverse until <u>next to last node</u> is found

                 p = p.next

            p.next = None, L.tail = p   # Update tail pointer

      L.size = L.size - 1   # Decrement the node count

O(n)

Singly Linked Lists: Stacks

How to Implement a Stack?

Array!!

and

Linked Lists!!

- **Singly Linked Lists**

Asymptotic Performance

Operation	Running Time
S.push(element)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Singly Linked Lists: Queues
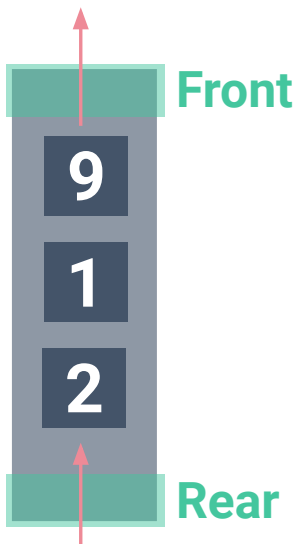
Front

9

1

2

Rear

How to Implement a Queue?

Array!!

and

Linked Lists!!

- **Singly Linked Lists**

Asymptotic Performance

Operation	Running Time
Q.enqueue(e)	$O(1)$
Q.dequeue()	$O(1)$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

Linked Lists

TYPE

- **Singly Linked List**
- **Circularly Linked List**
- **Doubly Linked List**
- **Doubly Circularly Linked List**

ACTION

- **Insert**
- **Delete**
- **Search**

Circular Queue

Start **End**

9 1 2 | | | | | | |

Front **Rear**
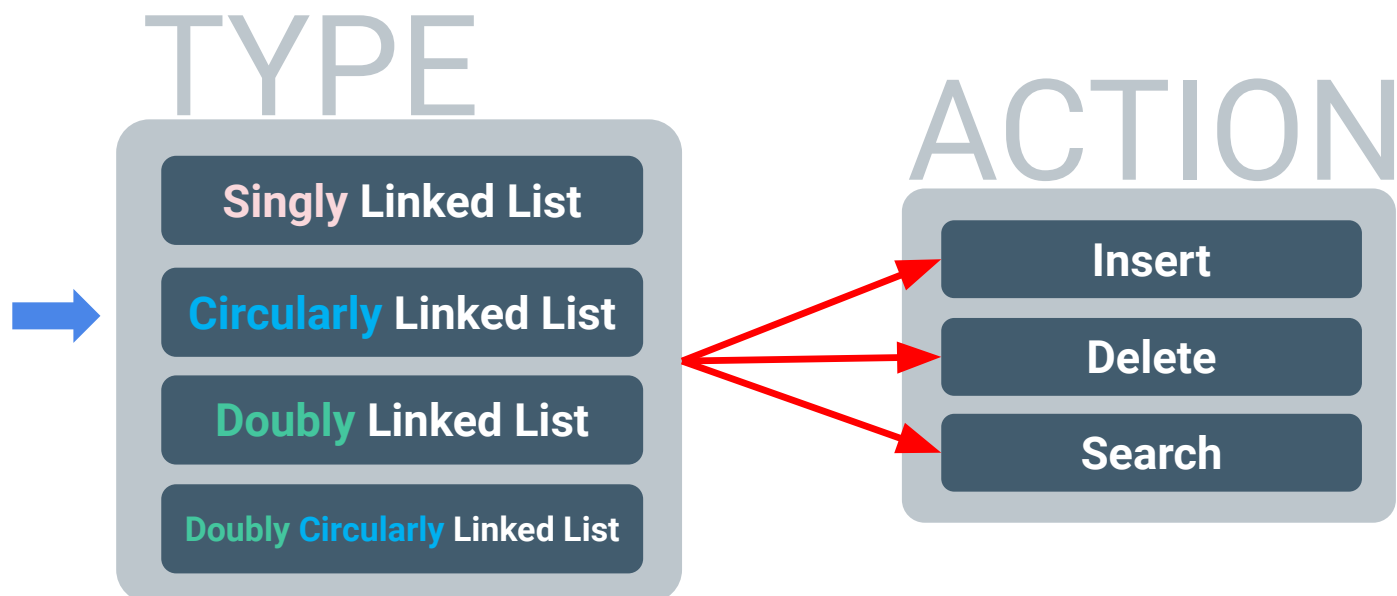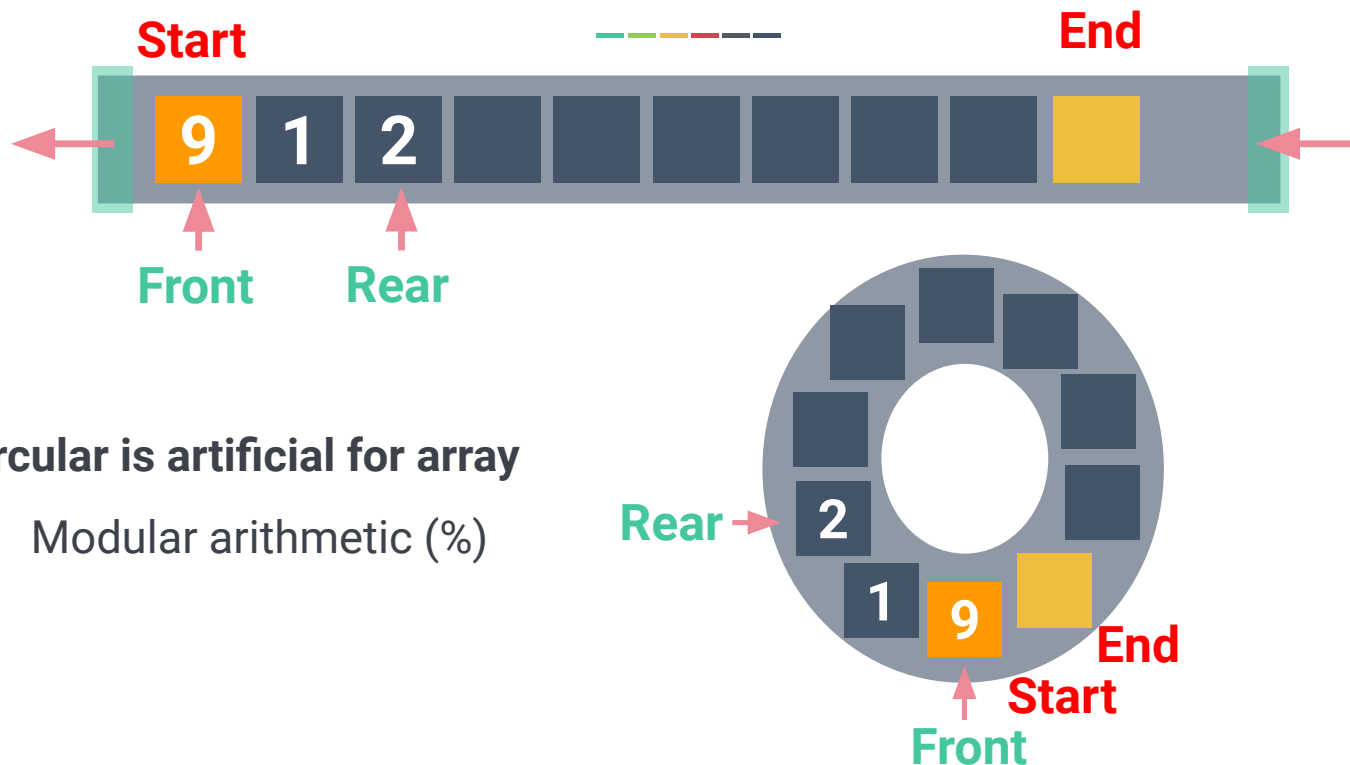
Circular is artificial for array

- Modular arithmetic (%)

Rear → 2

1 | 9

End
Start

Front

Circularly Linked Lists

A **circularly linked list** adds the notion of having the tail of the list to point back to the head of the list as the next node.
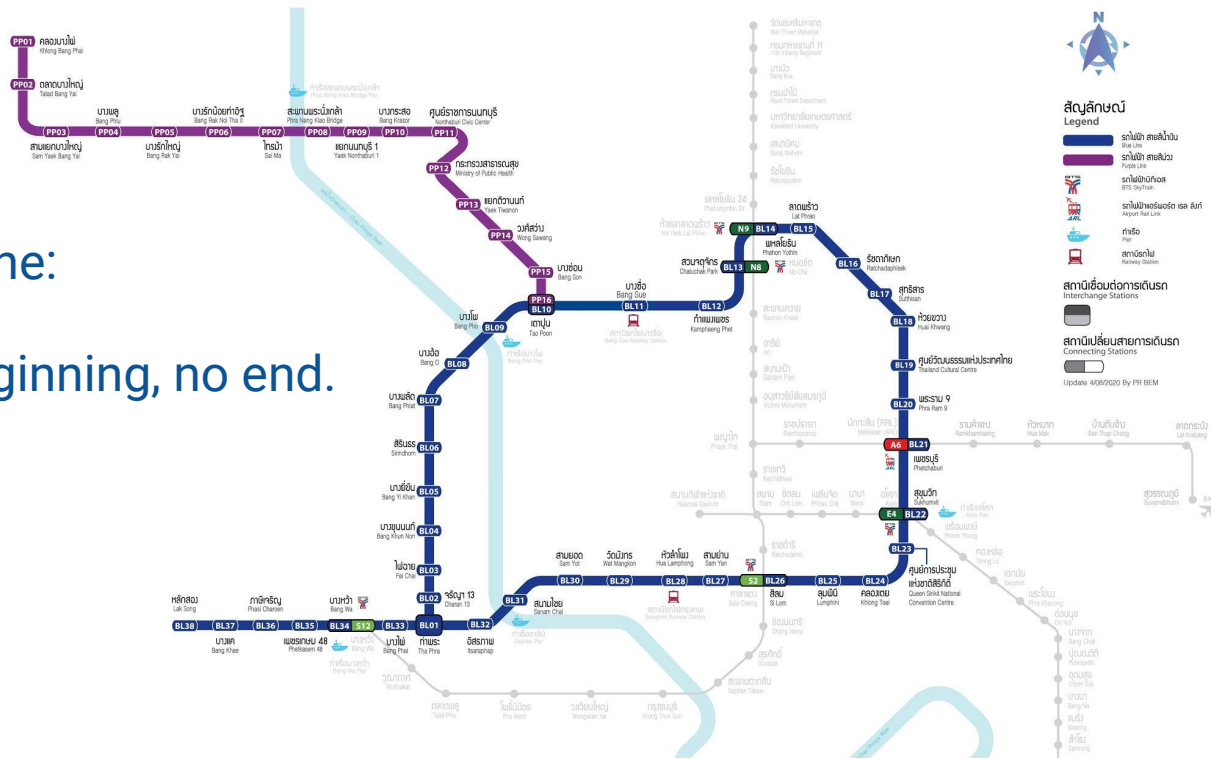
2   7   8   4

C

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013
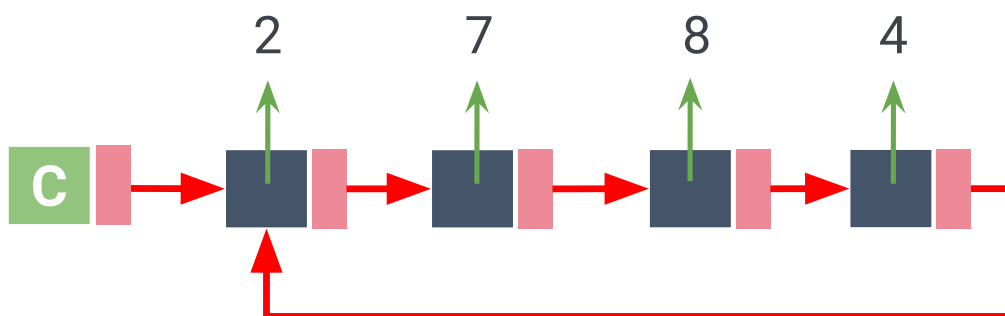
Circularly Linked Lists

Blue line:

No beginning, no end.

Circularly Linked Lists

Suppose that it takes 1 byte to store an integer.

Address/ Byte#	Value
6000	4
6001	6002
6002	2
6003	6008
6004	8
6005	6012
6006	
6007	
6008	7
6009	6004
6010	
6011	
6012	4
6013	6002
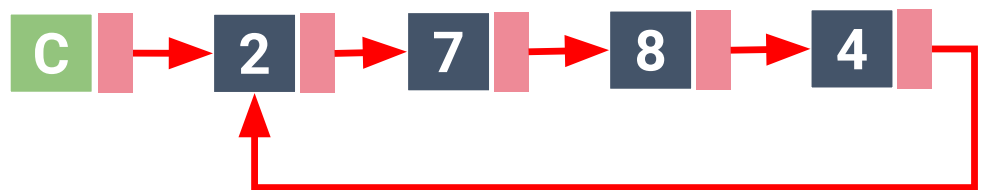
Circularly Linked Lists: Queues

Front

9

1

2

Rear

How to Implement a Queue?

Array!!

and
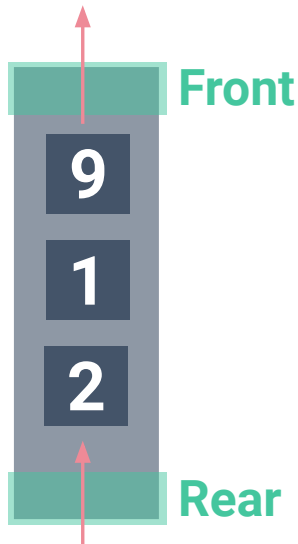
Linked Lists!!

- Singly Linked Lists
- **Circularly Linked Lists**

Asymptotic Performance

Operation	Running Time
Q.enqueue(e)	$O(1)$
Q.dequeue()	$O(1)$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

Linked Lists

TYPE

Singly Linked List

Circularly Linked List

Doubly Linked List

Doubly Circularly Linked List
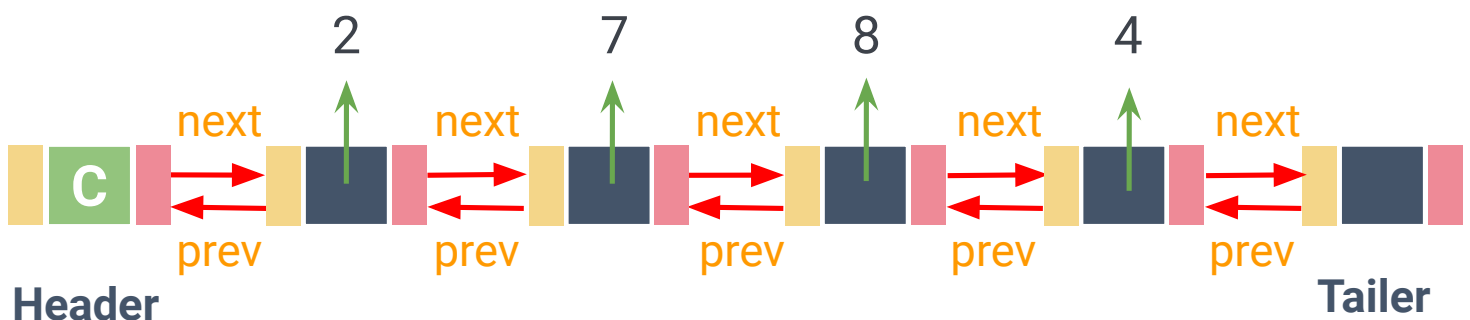
ACTION

Insert

Delete

Search

Doubly Linked Lists

To add more symmetry to the list, ***doubly linked lists*** allow each node to keep a reference to the node <u>before</u> it and a pointer to the node <u>after</u> it.

2   7   8   4

next   next   next   next   next

C

prev   prev   prev   prev   prev

Header

Tailer

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Doubly Linked Lists

Address/ Byte#	Value	
6000	4	
6001	6002	next
6002	2	
6003	6008	next
6004	6000	prev
6005	8	
6006	6012	next
6007	6008	prev
6008	7	
6009	6005	next
6010	6002	prev
6011		
6012	4	
6013	6015	next
6014	6005	prev
6015	6012	prev
6016		

Header next
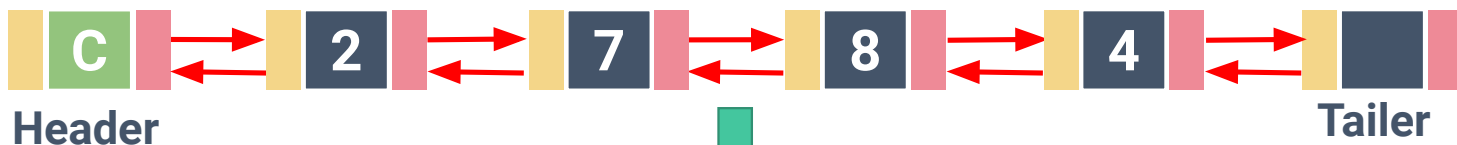
Tailer prev

Suppose that it takes 1 byte to store an integer.
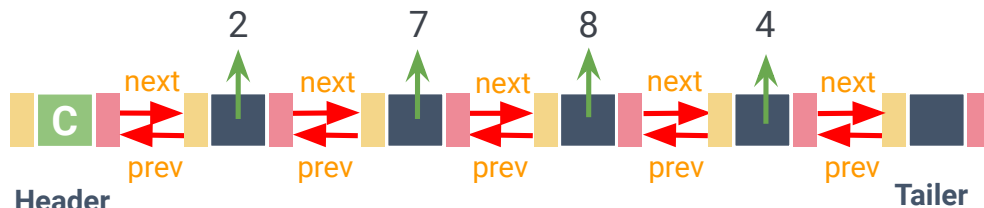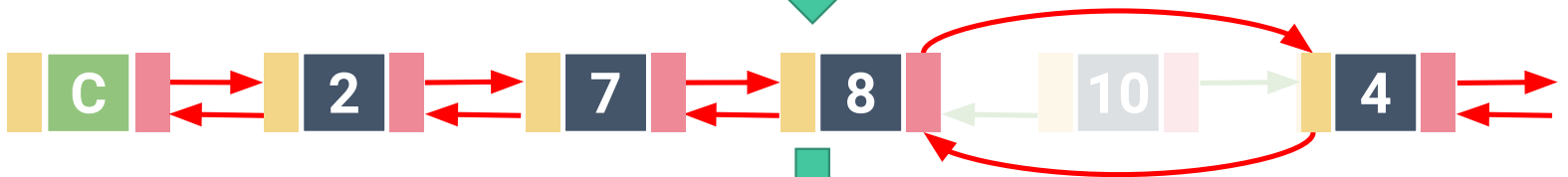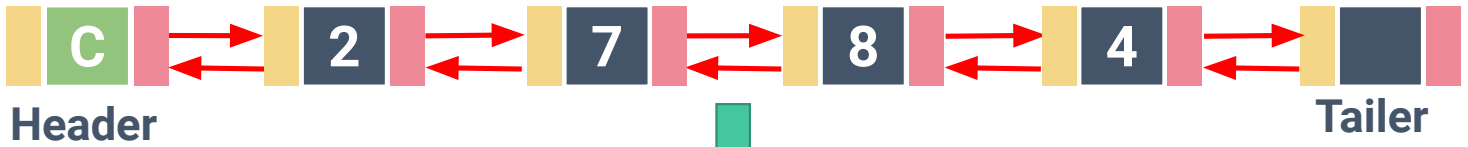

Doubly Linked Lists
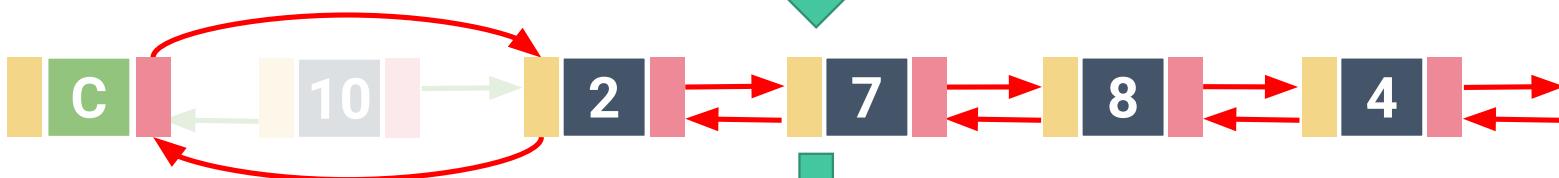
Insert at the first node

Doubly Linked Lists

Insert between nodes

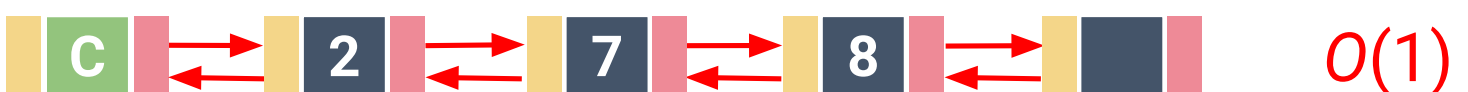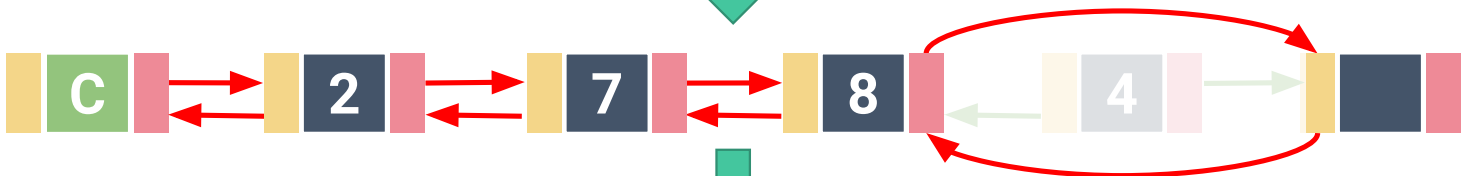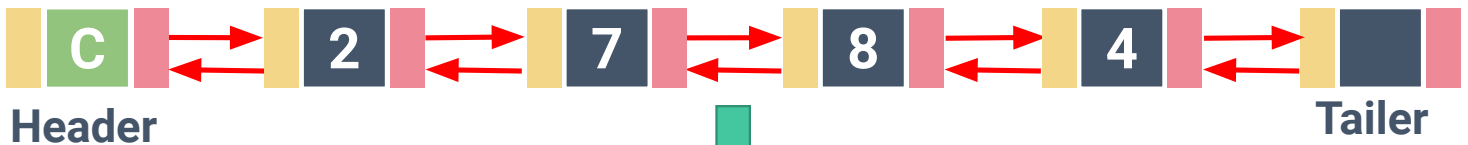Only update the pointers of neighbouring nodes.

C	2	7	8	4

Header

Tailer

C	2	7	8	10	4

C	2	7	8	10	4

Doubly Linked Lists

Delete at the last node

C	2	7	8	4

Header

Tailer

C	2	7	8	4

C	2	7	8

$O(1)$

Linked Lists

TYPE

- **Singly** Linked List
- **Circularly** Linked List
- **Doubly** Linked List
- **Doubly Circularly** Linked List

ACTION

- **Insert**
- **Delete**
- **Search**

Doubly Circularly Linked Lists

prev

next   next   next   next

| C | 2 | 7 | 8 | 4 |

prev   prev   prev   prev

Header   **Tailer**
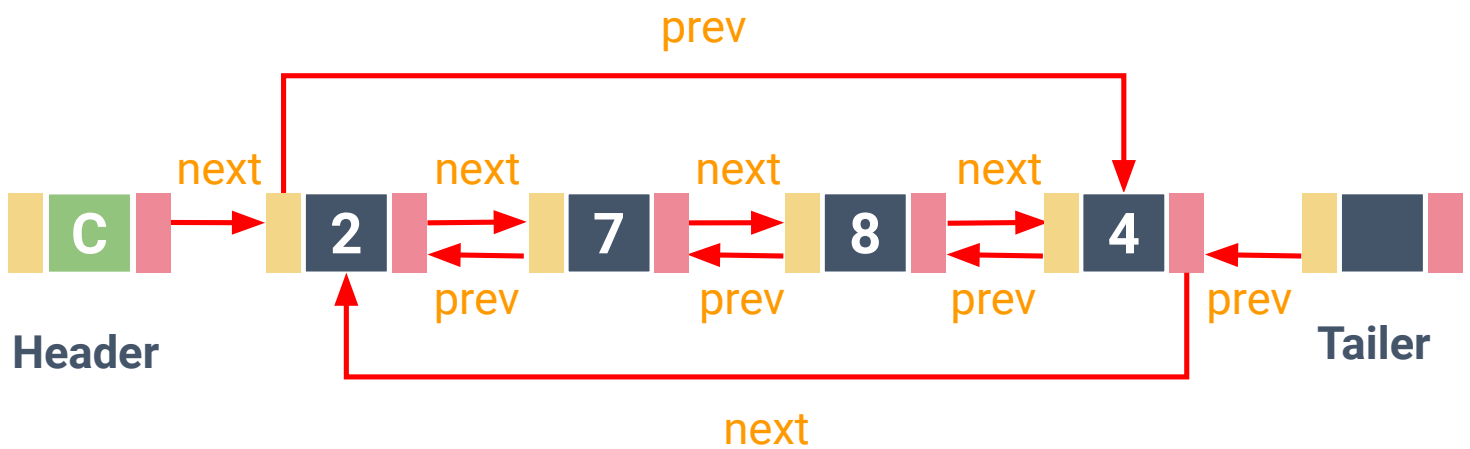
next

Linked Lists

Linked list properties:

- Each node contains an element and a pointer(s) to the next node (and previous node).
- **Sequential access** only: nodes are read from the beginning.
  - Not convenient to have an index, unlike array-based sequences.
- No pre-allocated fixed size of memory, resizeable.
- Insertion and deletion operations are more efficient compared to array.
  - Take $O(1)$ - constant time to add and remove elements at any part in linked lists.

Linked Lists

Linked list's limitations:

- Accessing the data/node in lists takes linear time - $O(n)$
  - To find the item or node at certain location, linked list has to start from the first node and traversing until the target is found.
  - For example, find the 10th node, has to traversing 10 times.
  - Unable to perform binary search.
- Use extra storage than the array to keep next pointers/references.
  - Impractical for storing small data such as characters.

Linked Lists vs Arrays

Operations	Array (Dynamic size)	Linked List
Indexing/searching	$O(1)$	$O(n)$
Add/remove at beginning	$O(n)$	$O(1)$
Add/remove at end	$O(1)$	$O(1)$ when last element is known $O(n)$ when last element is unknown
Add/remove in between	$O(n)$	$O(1)$
Wasted memory space (average)	$O(2n)$	$O(2n)$ - Singly linked list or $O(3n)$ - Doubly linked list

Individual Assignment

- Assignment#3: Queues

- Due 09.00 am, Tuesday 01/09/2020.

- Submission

  - Email: sirasit@it.kmitl.ac.th

  - Paper: in classroom next week

- Can be either written by hand or typing.

- Make sure to submit on time!!

  - Late submission has penalty on the score.

- If unable to submit on time for reasonable reasons, let me know asap.

Group Assignment

- Presentation slides are due to be submitted via email by midnight (12AM) the day before the presentation.
- Make sure to submit on time!!
  - Late submission has penalty on the score.
- If unable to submit on time for reasonable reasons, let me know asap.