Linked List Variation

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Linked List Variations: **Overview**

- The linked list implementation used in List ADT is known as **singly linked list**
  - Each node has one reference / pointer (a single link), pointing to the next node in sequence

- Using references allow the node to be non-contiguous:

  ➔ flexible organizations in chaining up the nodes

- Many high level ADTs utilize variations of linked list as internal data structure

- Let's look at a few common choices

Common Variations: At a glance

- Using list node with **one reference / pointer**:
  1. **Tailed Linked List**
  2. **Circular Linked List**
  3. Linked List with a **dummy head node**

- Using list node with **two references / pointers**:
  1. **Doubly linked list**
  2. **Circular doubly linked list**

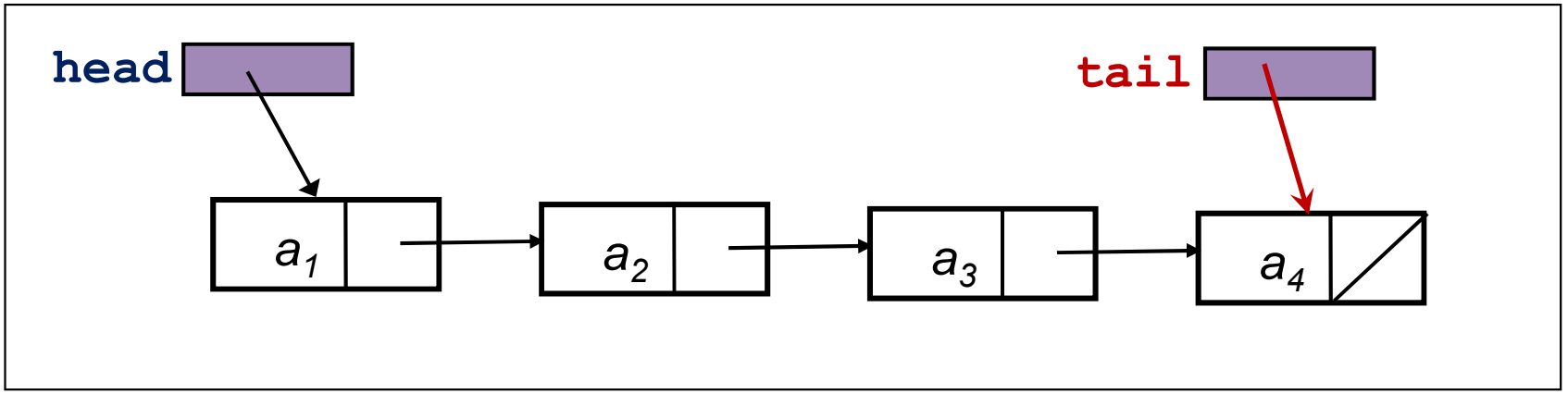
- Other variations are possible:
  - Once you understand the fundamental, it is quite easy to extend to other organizations!

Head and tail: First and last

TAILED LINKED LIST

Tailed Linked List

- **Motivation:**
  - The last node in singly linked list takes the longest time to reach
  - If we keep adding item to the end of list ➔ very inefficient
- **Simple addition:**
  - Keep an additional reference to the last node



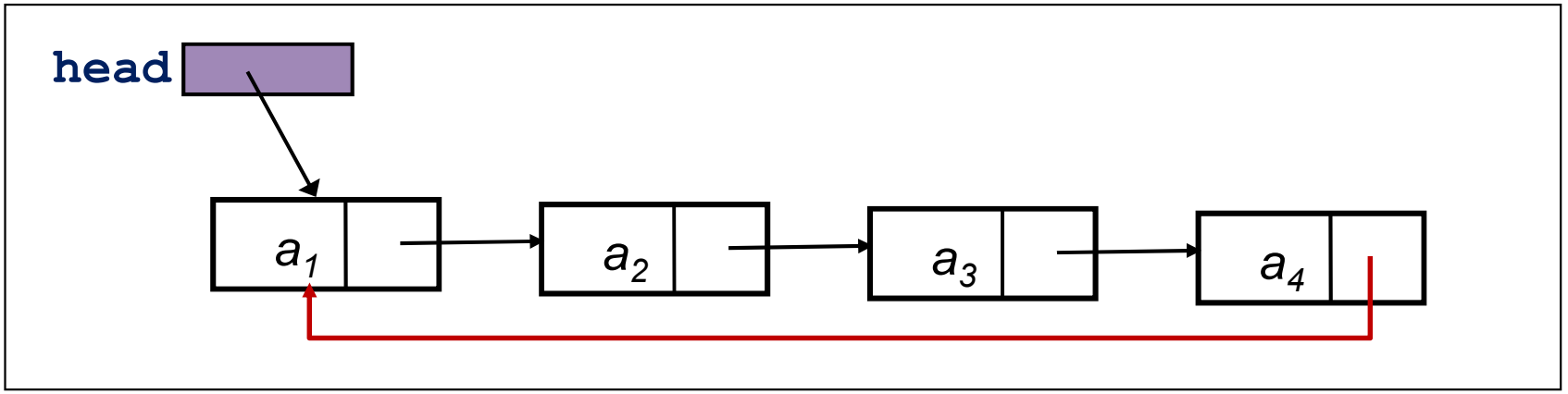
Go round and round

CIRCULAR LINKED LIST

Circular Linked List

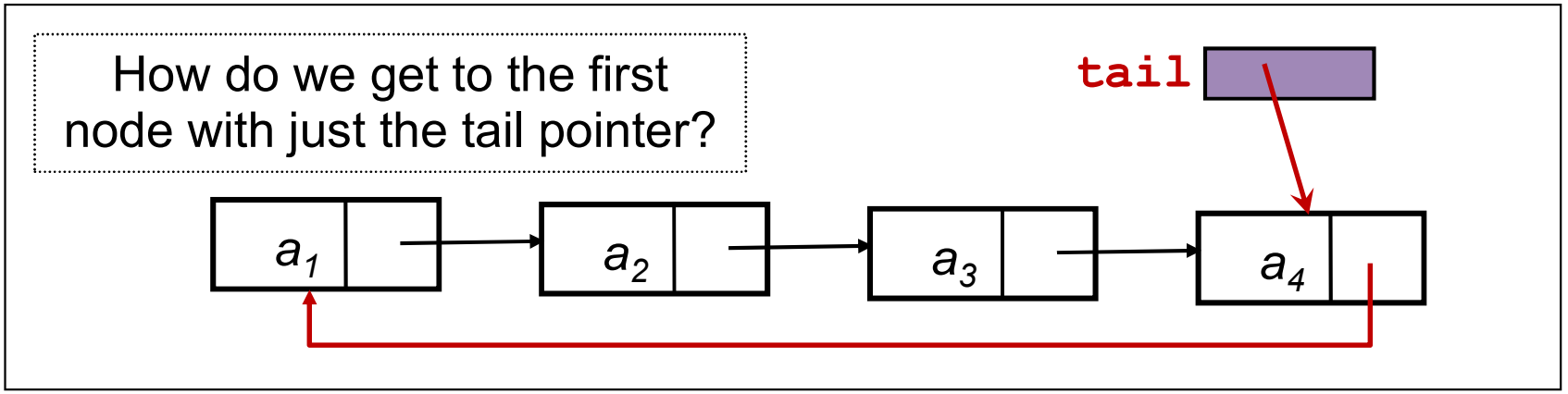
- **Motivation:**
  - Sometimes we need to repeatedly go through the list from 1st node to last node, then restart from 1st node, ….
- **Simple addition:**
  - Just link the last node back to the first node

➔ No `None` reference in the linked list



Circular Linked List: **Even Better**

- Circular Linked List can be made **even better**:
  - Keep the **tail reference** instead of **head**!
  - We now know both the first node and the last node with a single reference
- **Simple addition:**
  - Keep track of the tail reference



Circular Linked List: **Common Code**

- Given a circular linked list:
  - How do we know we have passed through every nodes in the list?
- Idea:
  - If we land on a node again (e.g. the first node), then we have finished one round

```
current = head
visit the current node
current = current.next

while current != head:
    visit the current node
    current = current.next
```

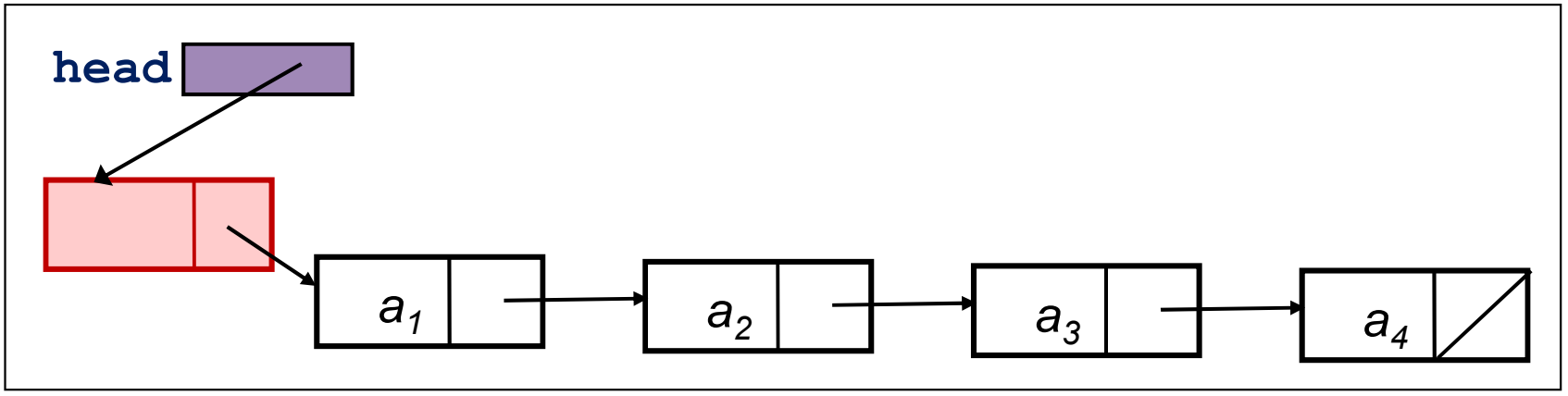
Simple solution as long as the list is not empty

There is a dummy in front!!

DUMMY HEAD NODE

Linked List with Dummy Head Node

- **Motivation:**
  - Insert/Remove the first node in linked list is a special case:
    - because we need to update the **head reference**
- **Idea:**
  - Maintain an extra node at the beginning of the list
    - **Not** used to store real element
    - Only to simplify the coding



Two is better than one

DOUBLY LINKED LIST

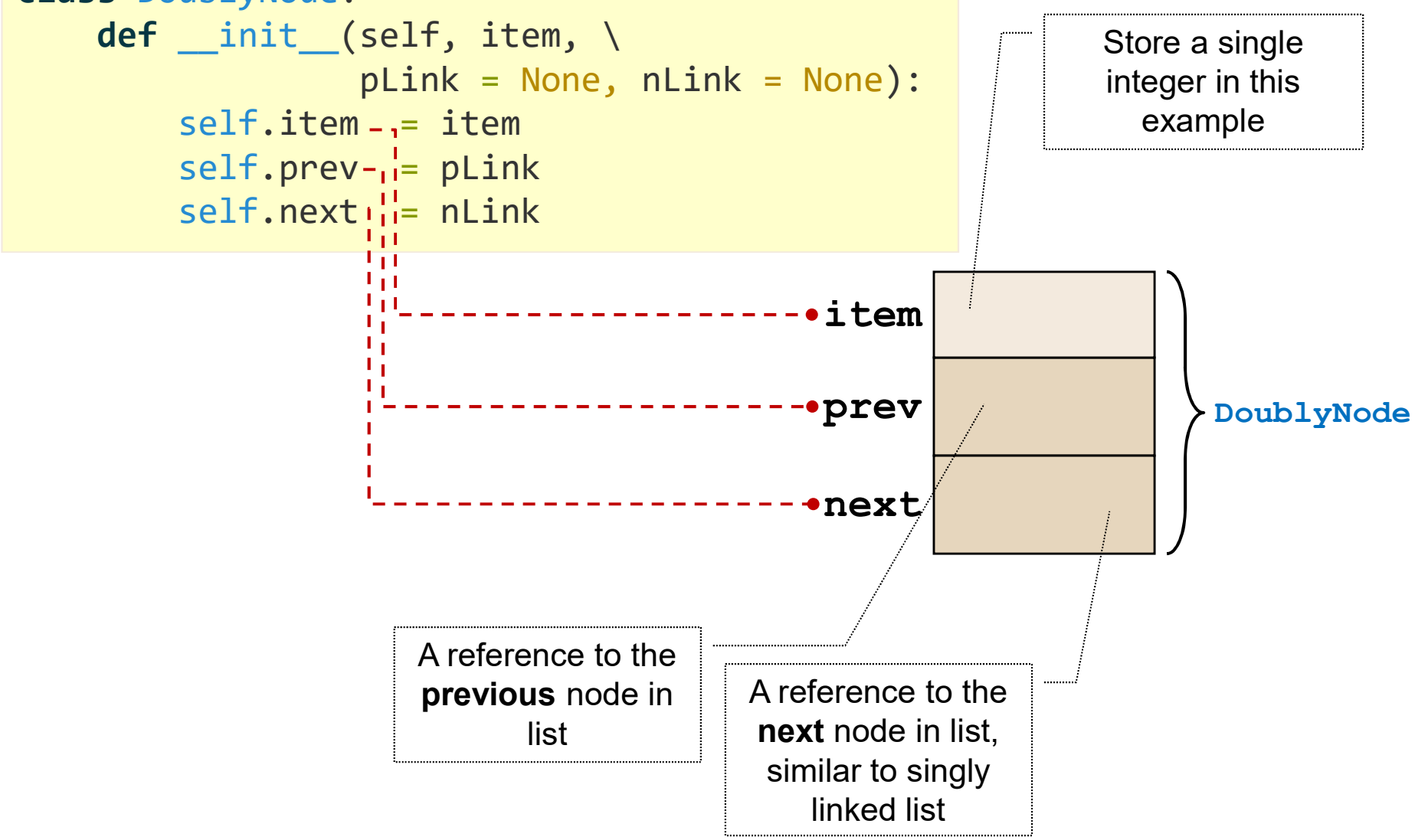
Doubly Linked List: **Motivation**

- Singly Linked List only facilitates movement in one direction
  - Can get to next node in sequence easily
  - Cannot go to the previous node
  - The last node takes the longest time to reach

- Doubly Linked List facilitates movement in both directions
  - Can get to next node in sequence
  - Can get to previous node in sequence

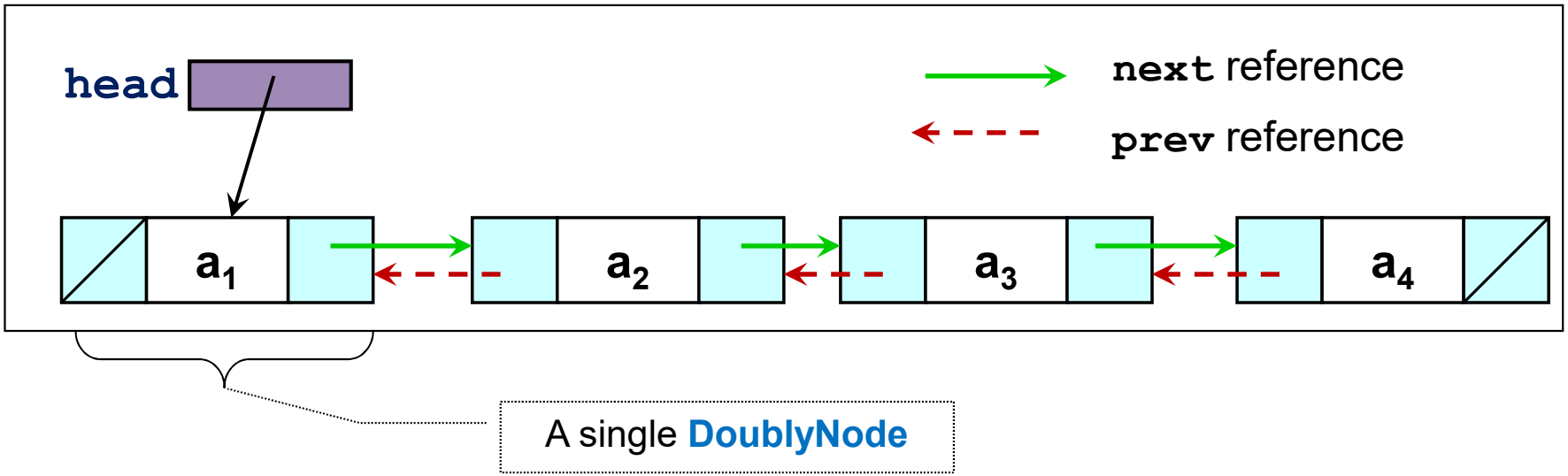
Doubly Linked List: **One Node**

```
class DoublyNode:
    def __init__(self, item, \
                 pLink = None, nLink = None):
        self.item = item
        self.prev = pLink
        self.next = nLink
```



Doubly Linked List: **Example**

- List of four items < a_1, a_2, a_3, a_4 >



- We need:
  - **`head`** reference to indicate the first node
  - **`None`** in the **`prev`** reference field of first node
  - **`None`** in the **`next`** reference field of last node

Doubly Linked List: **Operations**

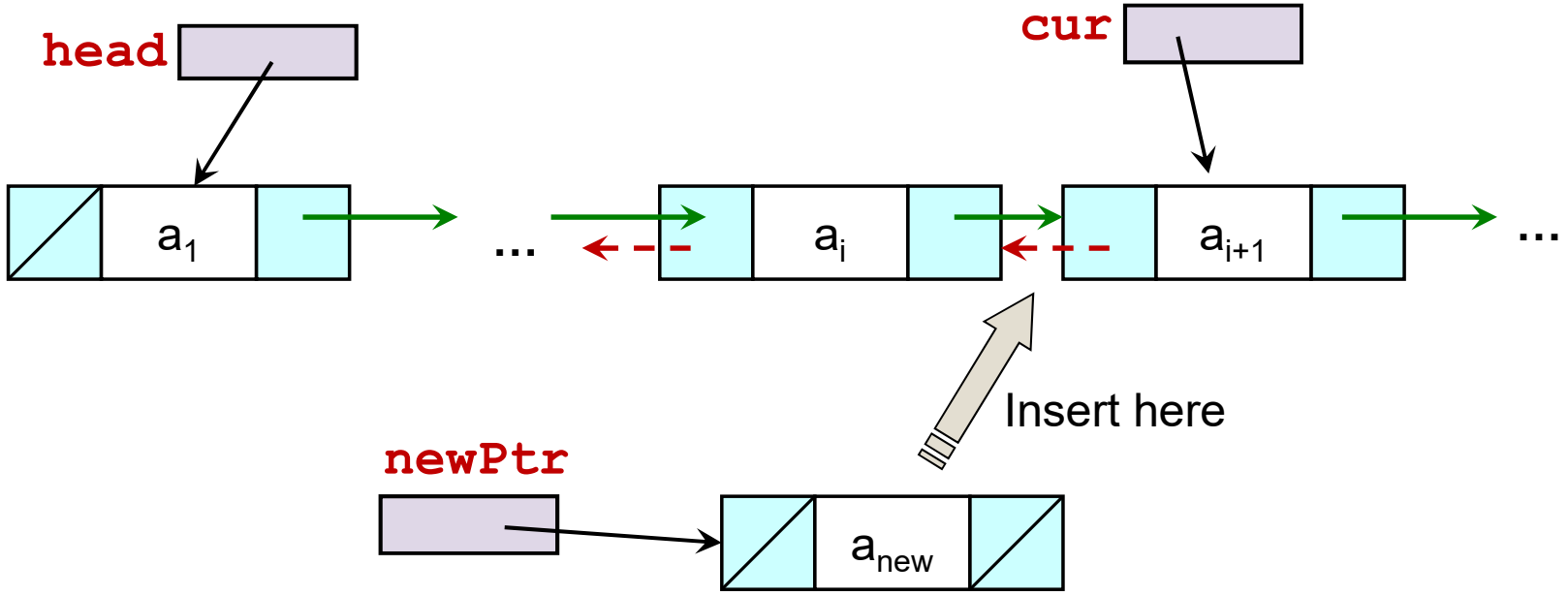
- **Insertion** and **removal** in doubly linked list:
  1. Locate the point of interest through list traversal
  2. Modify the pointers in affected nodes

- However, insertion and removal affects more nodes in doubly linked list:
  - Both the nodes before **and after** the point of operation are affected

- We only show the general case for insertion and removal in the next section

General Insertion: Doubly Linked List

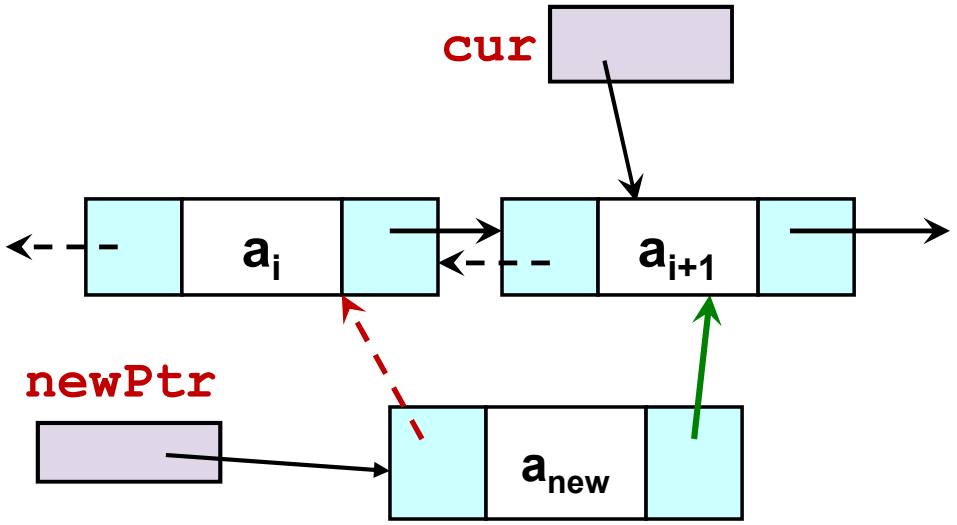
- Assume we have the following:
  - `newPtr` reference:
    - Pointing to the new node to be inserted
  - `cur` reference:
    - Use list traversal to locate this node
    - The new node is to be inserted **before** this node



General Insertion: Doubly Linked List

Step 1:

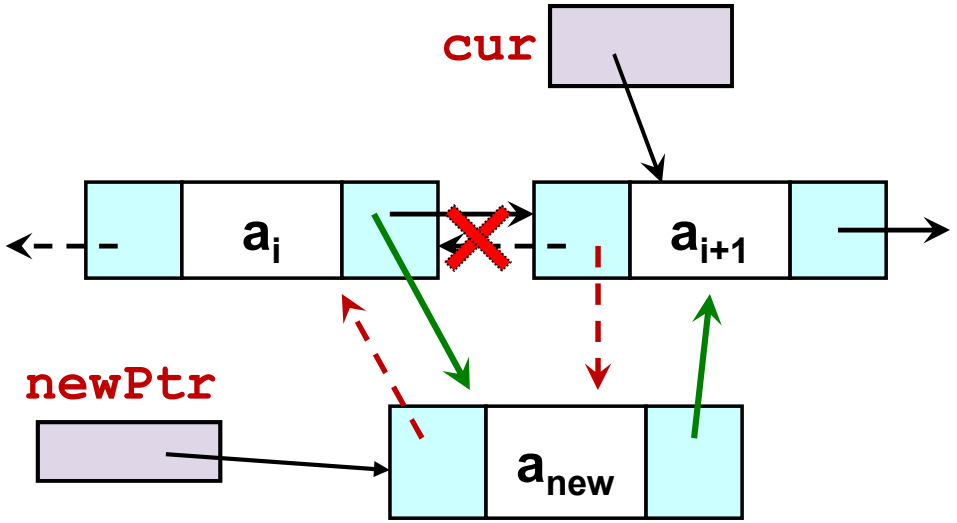
```
newPtr.next = cur
newPtr.prev = cur.prev
```



Step 2:

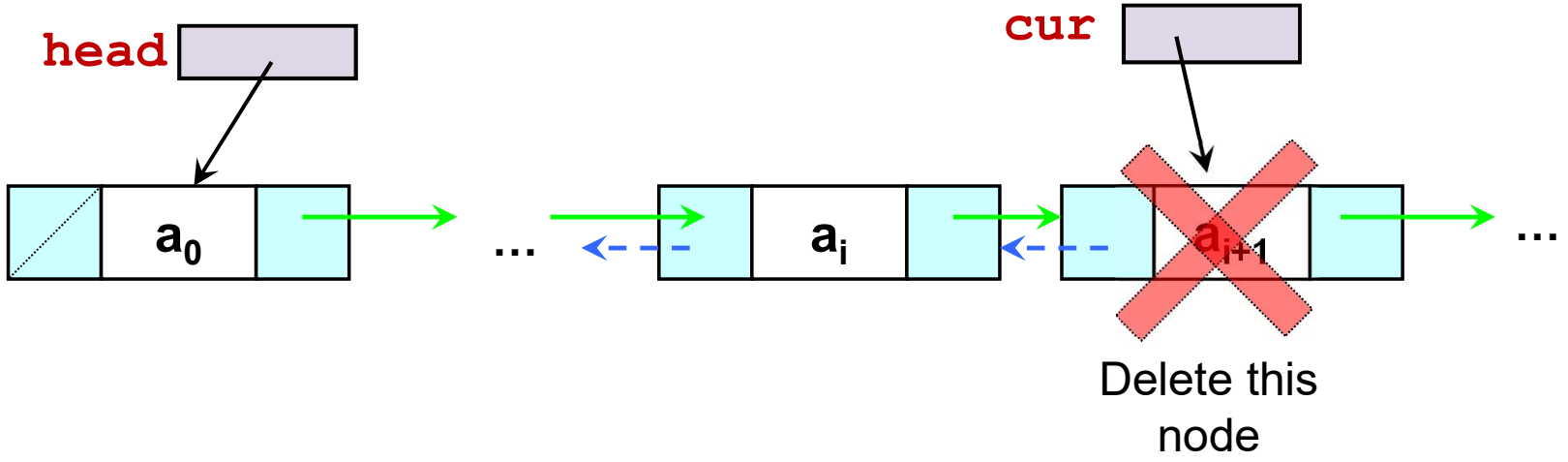
```
cur.prev.next = newPtr;
cur.prev = newPtr;
```

Any other alternatives?



General Deletion: Doubly Linked List

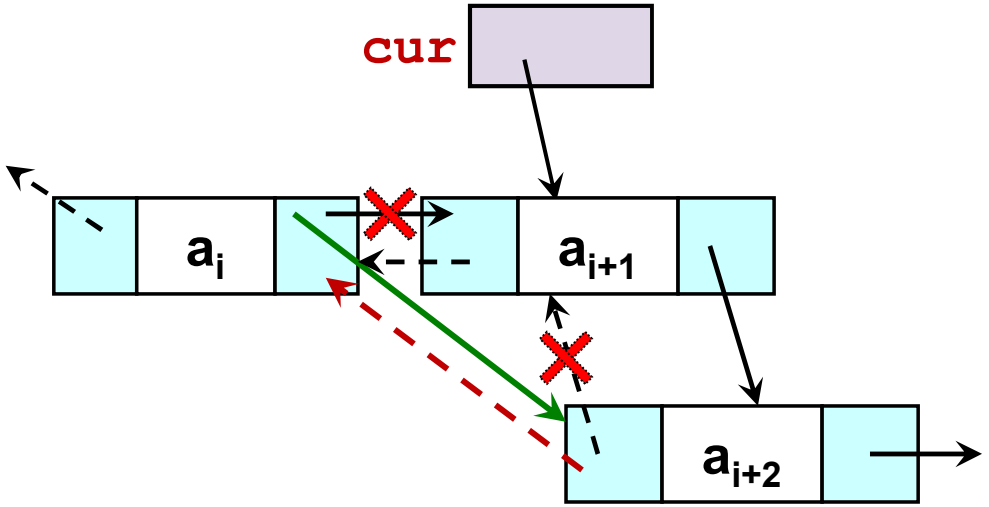
- Assume we have the following:
  - `cur` reference:
    - Points to the node to be deleted



Deletion Example: Doubly Linked List

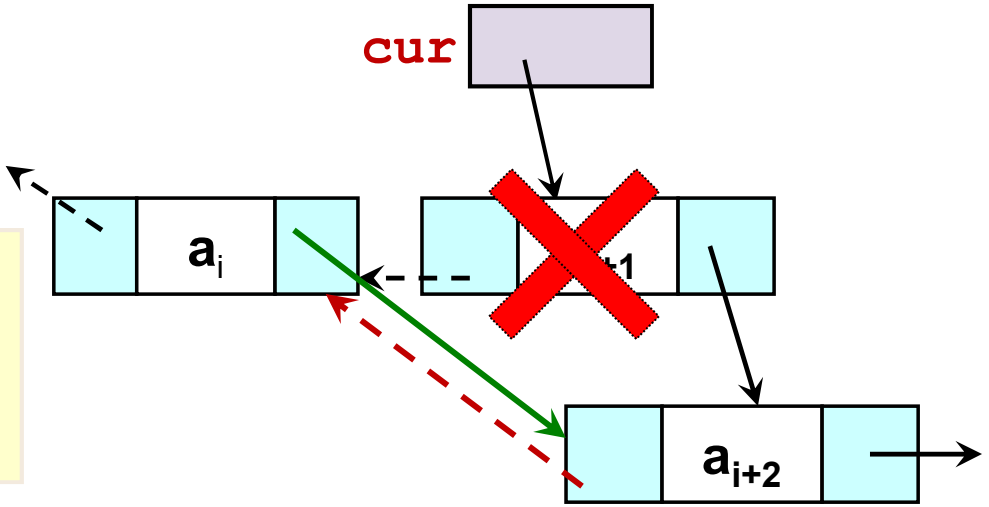
Step 1:

```
cur.prev.next = cur.next
cur.next.prev = cur.prev
```



Step 2:

Remove the node cur points to



Linked List Variation: **More?**

- By using the ideas discussed, we can easily construct:
  - **Doubly linked list** with **dummy head node**
  - **Circular doubly linked list**
  - etc…

- Rather than memorizing the variations:
  - Make sure you understand the basic of pointer manipulation
  - Make sure you can reason about the pros and cons of each type of organization

Summary

- Singly Linked List with Dummy Head Node
- Tailed singly linked list
- Circular singly linked list
- Doubly linked list

END