Lecture 6b

Linked List Variations

Similar but not the same

Linked List Variations: **Overview**

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

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

  ➔ flexible organizations in chaining up the nodes

- Several 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 pointer**:
  1. **Tailed Linked List**
  2. **Circular Linked List**
  3. Linked List with a **dummy head node**

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

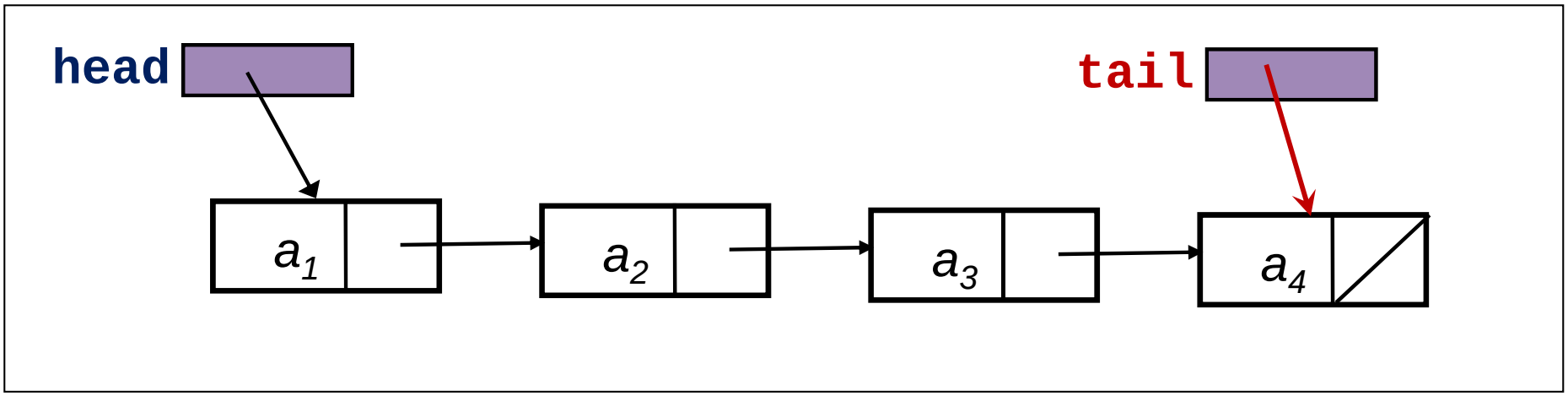
- Other variations are possible:
  - Once you understand the fundamental, it is quite easy to extend to other organizations!

Tailed Linked List

Head and tail: First and last

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 (some applications require this) ➔ very inefficient
- **Simple addition:**
  - Keep an additional pointer to point to the last node



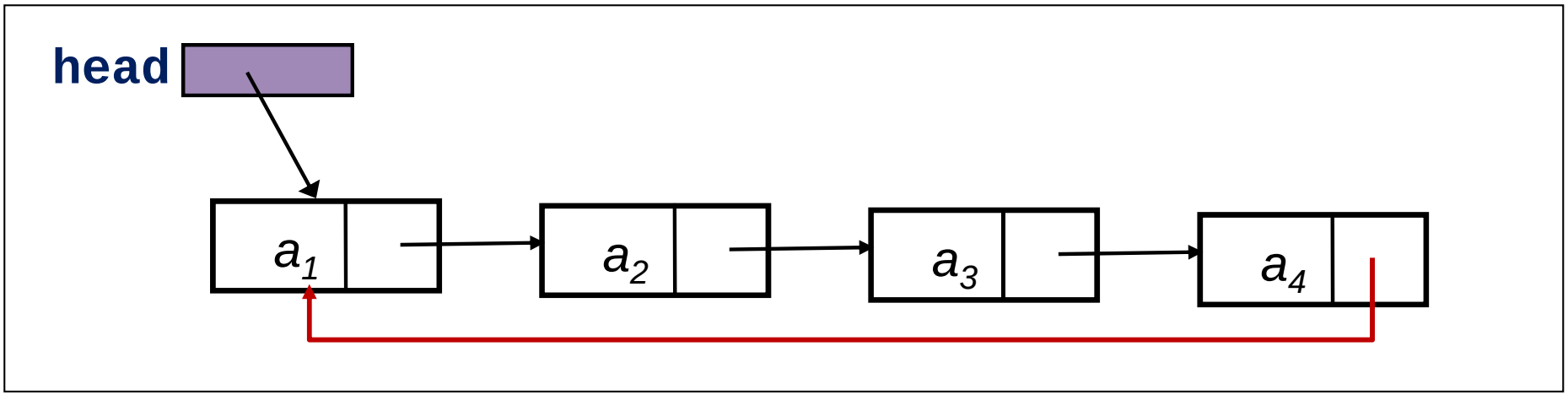
Circular Linked List

Go round and round

Circular Linked List

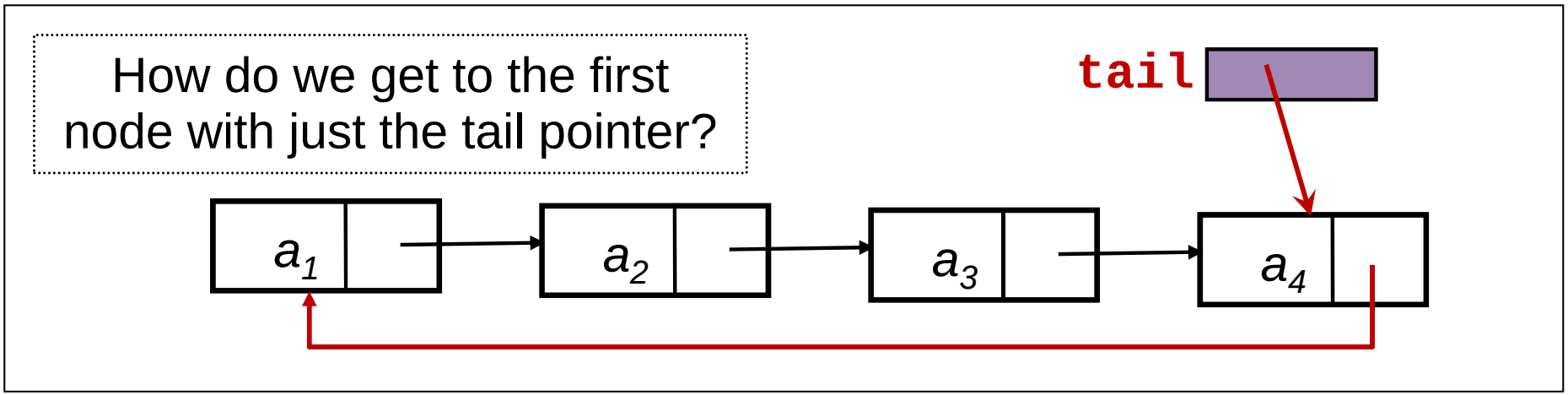
- **Motivation:**
  - Sometimes we need to repeatedly go through the list from 1st node to last node, then restart from 1st node, ….
- **Simple addition:**
  - Just link the last node back to the first node

➔ No **`NULL`** pointer in the linked list



Circular Linked List: **Even Better**

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



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

```
curPtr = head;
do {
    // visit the node curPtr points to
    curPtr = curPtr->next;
} while (curPtr != head);
```

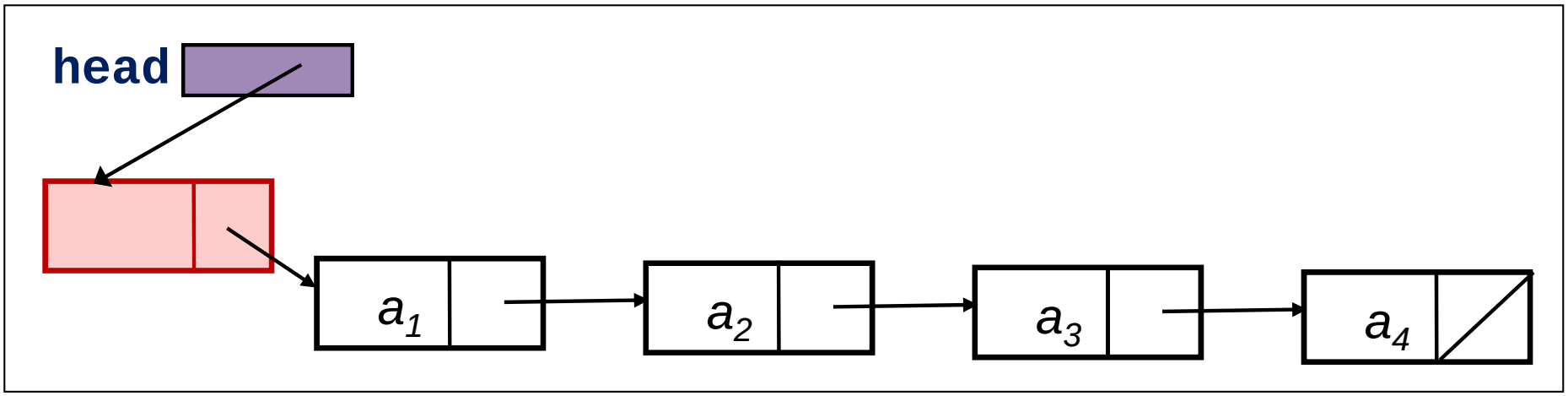
Simple solution as long as the list is not empty

Dummy Head Node

There is a dummy in front!!

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 pointer
- **Idea:**
  - Maintain an extra node at the beginning of the list
    - **Not** used to store real element
    - Only to simplify the coding



Doubly Linked List

Two is better than one

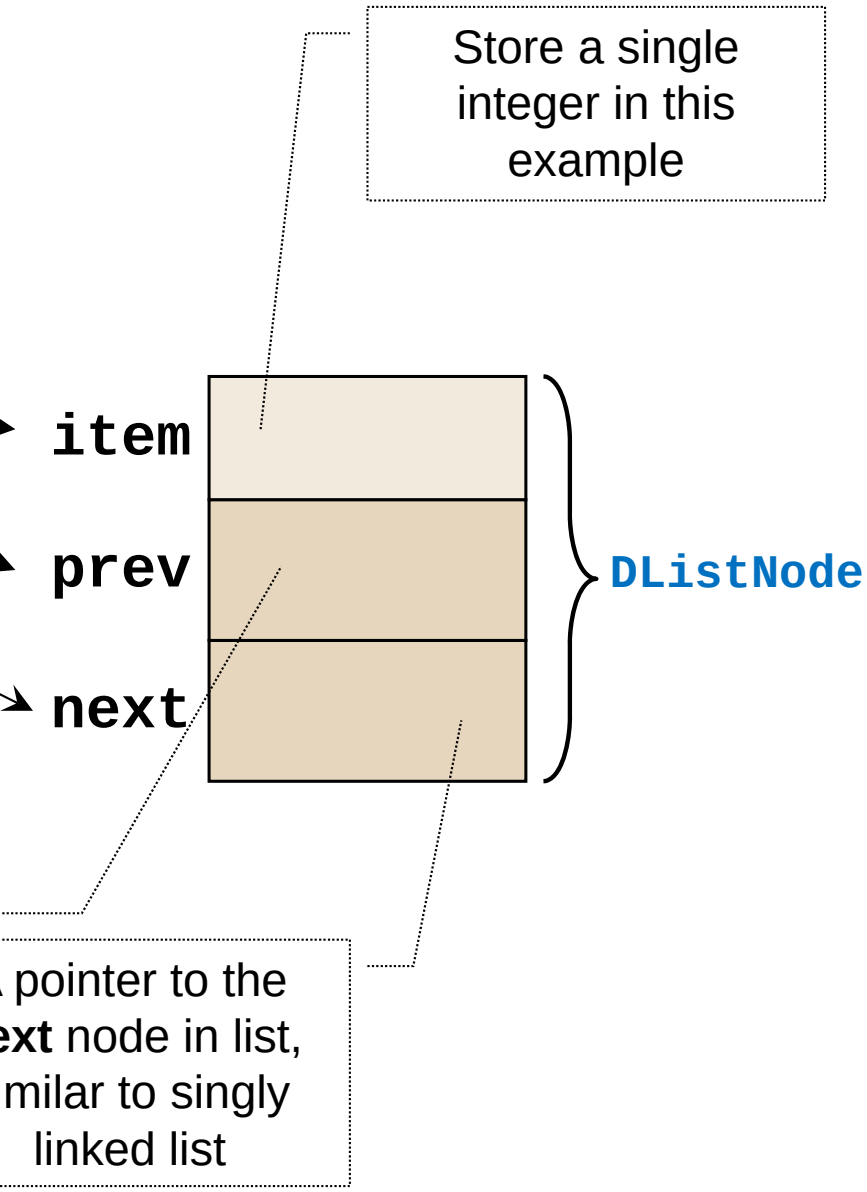
Doubly Linked List: **Motivation**

- Singly Linked List only facilitates movement in one direction
  - Can get to the next node in sequence easily
  - Cannot go to the previous node

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

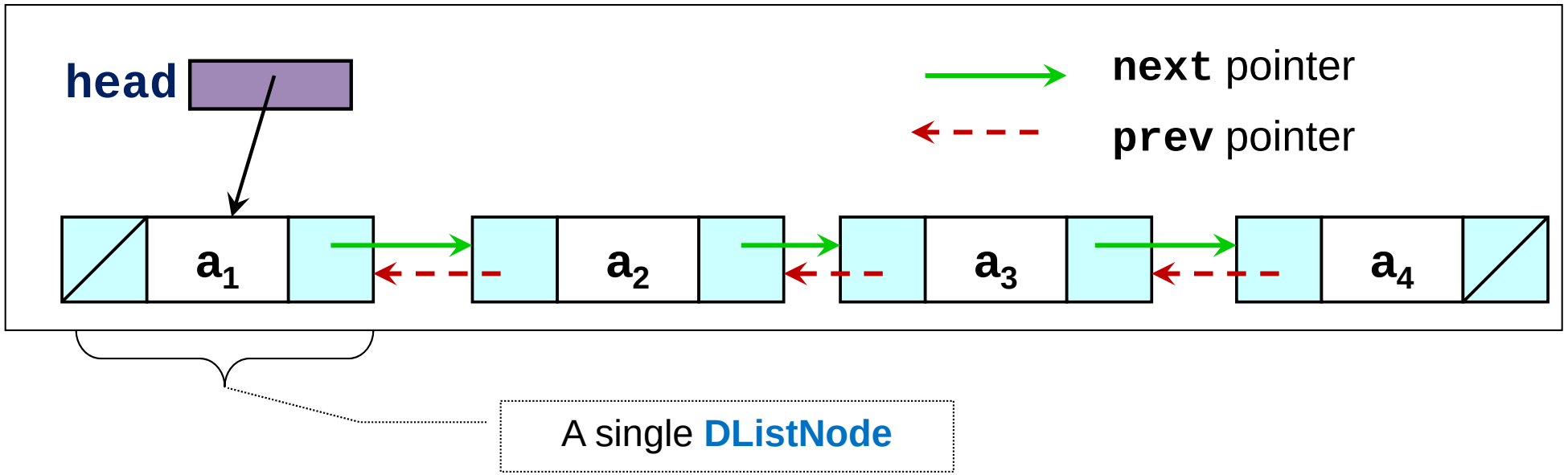
A single node in the Doubly Linked List

```
struct DListNode {
    int item;
    DListNode *prev;
    DListNode *next;
};
```



An example of Doubly Linked List

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



- We need:
  - **head** pointer to indicate the first node
  - **NULL** in the **prev** pointer field of first node
  - **NULL** in the **next** pointer field of last node

Doubly Linked List: **Operations**

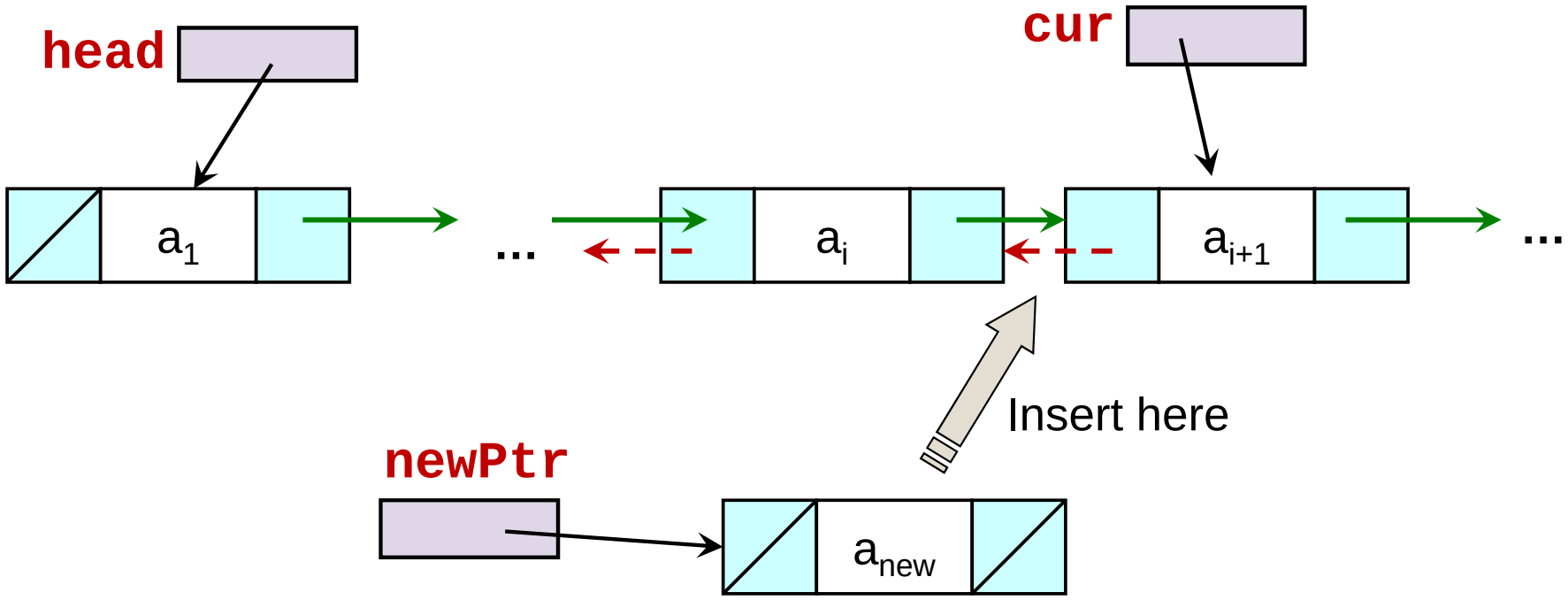
- Insertion and removal in doubly linked list has the similar steps as in singly linked list:
  - Locate the point of interest through list traversal
  - Modify the pointers in affected nodes

- 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
  - Try to figure out the code for other special cases

Doubly Linked List: **General Insertion**

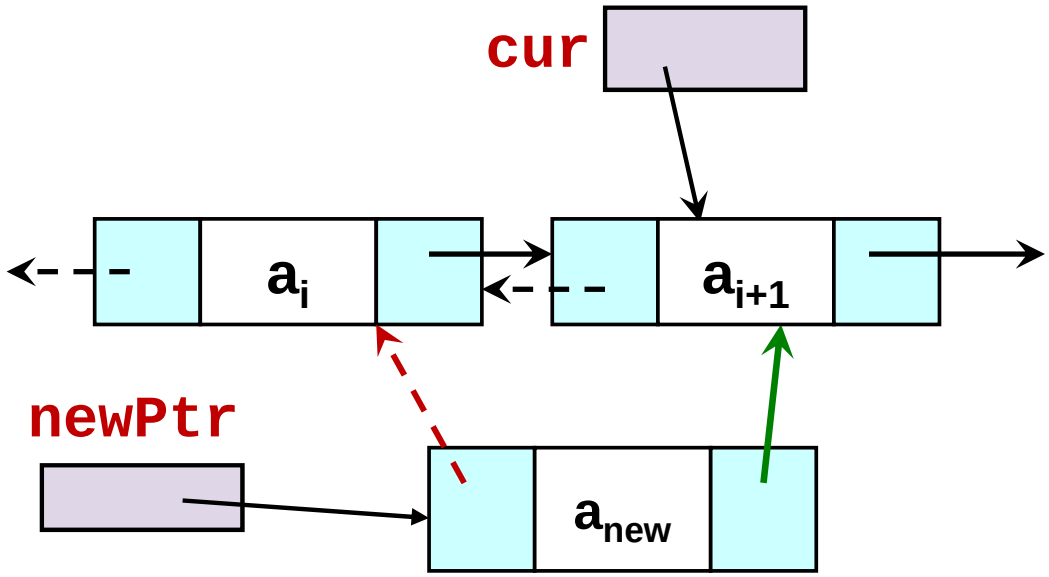
- Assume we have the following:
  - **`newPtr`** pointer:
    - Pointing to the new node to be inserted
  - **`cur`** pointers:
    - Use list traversal to locate this node
    - The new node is to be inserted **before** this node



Doubly Linked List: **General Insertion**

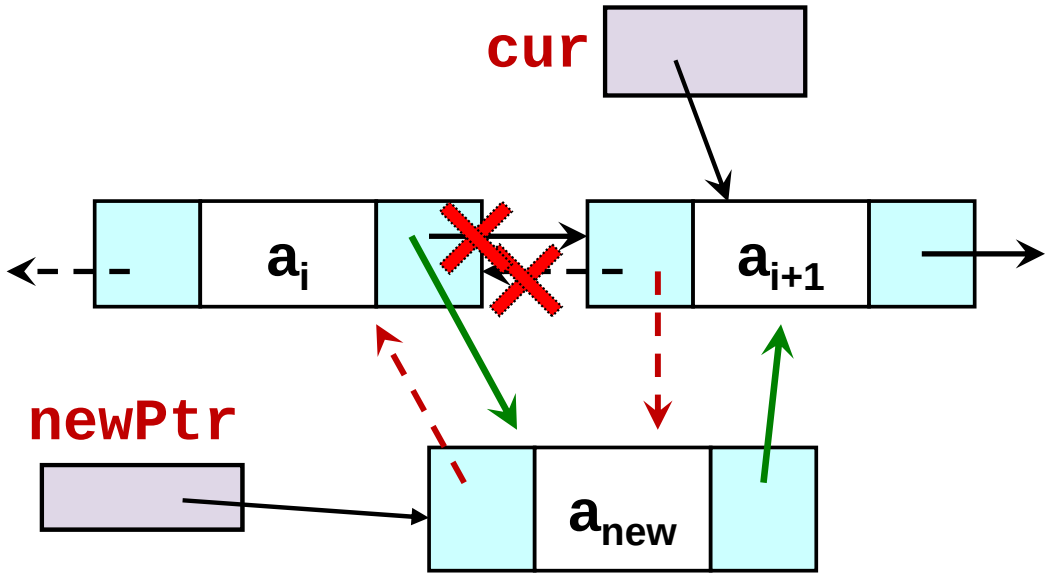
Step 1:

```
newPtr->next = cur;
newPtr->prev = cur->prev;
```



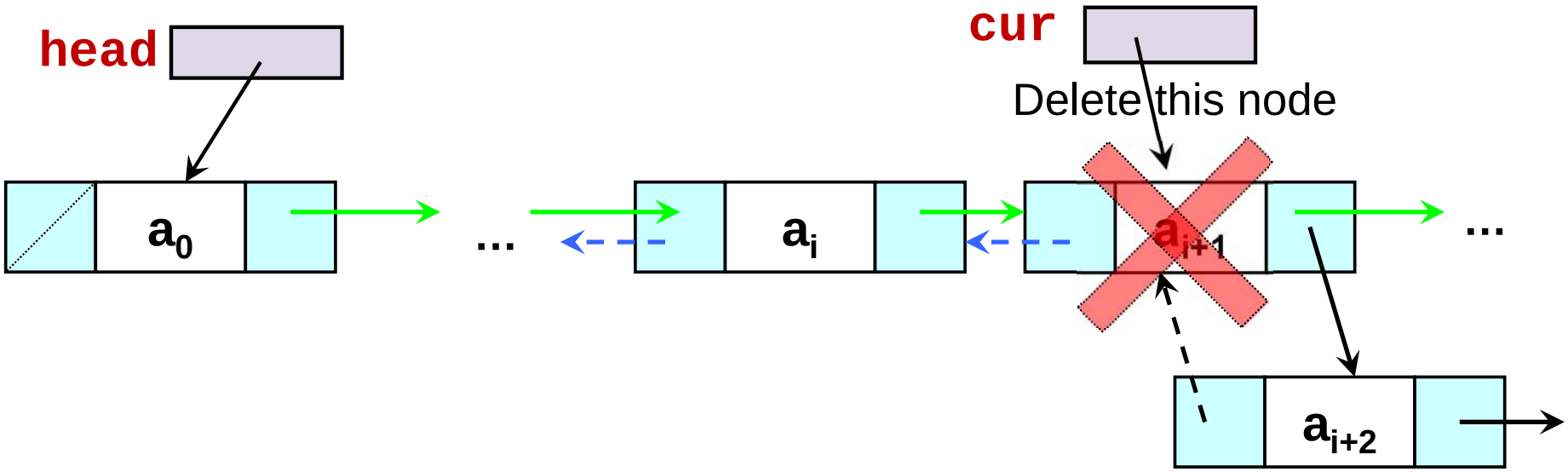
Step 2:

```
cur->prev->next = newPtr;
cur->prev = newPtr;
```



Doubly Linked List: **General Deletion**

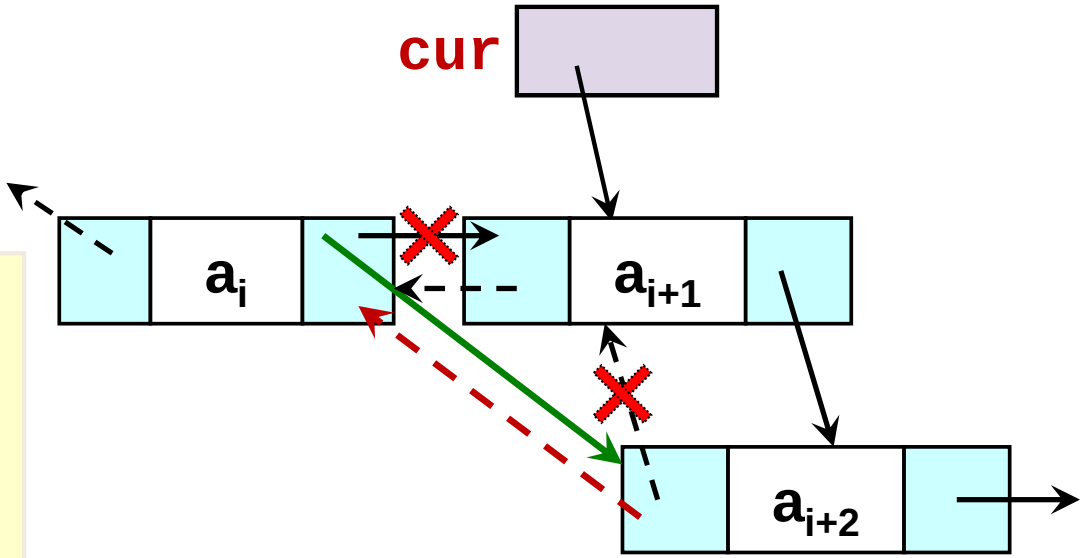
- Assume we have the following:
  - **cur** pointer:
    - Points to the node to be deleted



Deletion: Using Doubly Linked List

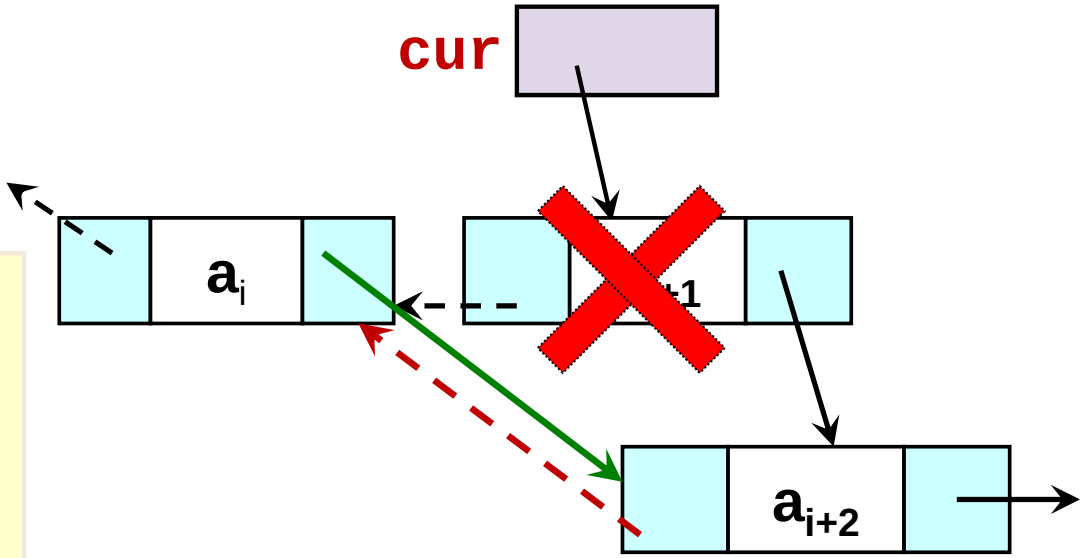
Step 1:

```
cur->prev->next = cur->next;
cur->next->prev = cur->prev;
```



Step 2:

```
delete cur;
cur = NULL;
```

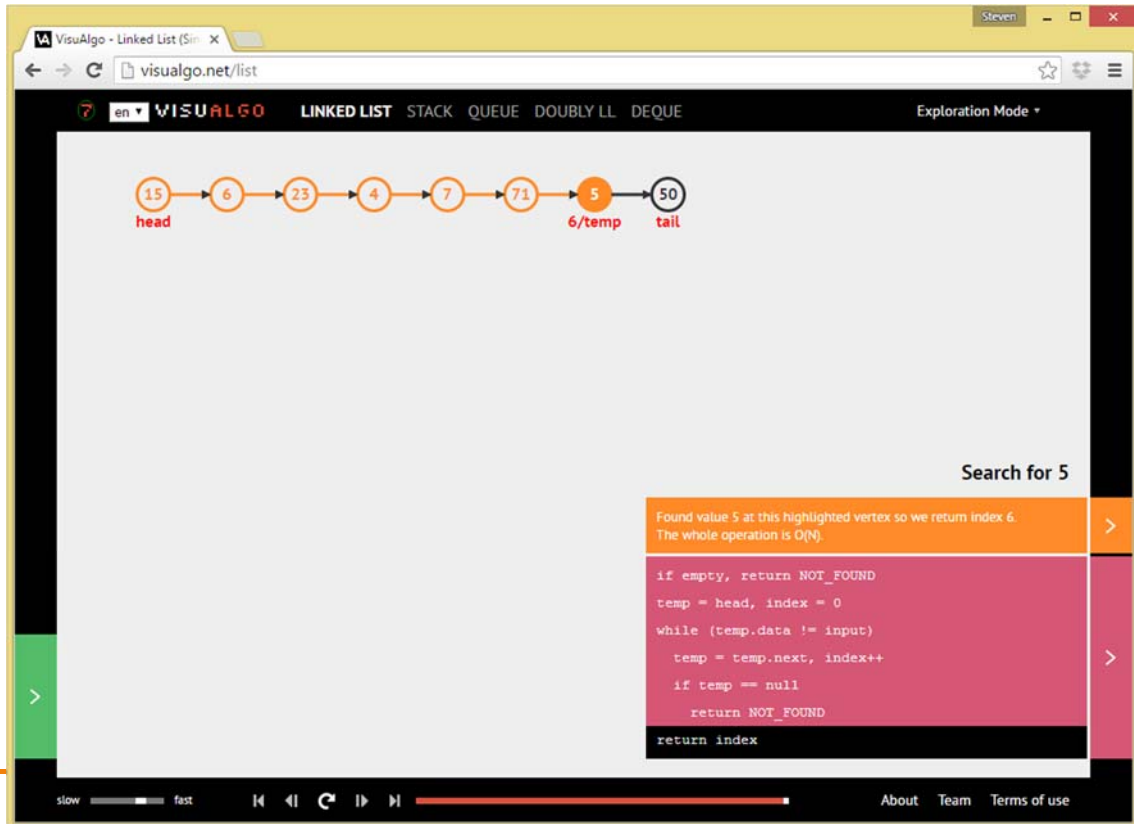


Linked List Variation: **More?**

- By using the ideas discussed,
  we can easily construct:
  - Tailed Double Linked List
  - 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

http://visualgo.net/list

- VisuAlgo version:
  - With Tail Pointer, Not Circular, Without Dummy Head
  - Operations Supported (integer list only):
    - Create List: Random, R Sorted, R Fixed Size, User Defined List
    - Insert: At Head, At Tail, At Index K
    - Remove: At Head, At Tail, At Index K
    - Search
- Please explore:
  - http://visualgo.net/list, Single Linked List
  - http://visualgo.net/list?mode=DLL, Doubly Linked List



C++ STL list

- Do we have to code ListLL.cpp (and all these variations and special cases) every time we need to use a Linked List?

- Fortunately, no ☺
- We can use C++ STL list
  - http://en.cppreference.com/w/cpp/container/list

Summary

- Singly Linked List with Dummy Head Node
- Tailed Singly Linked List
- Circular Singly Linked List
- Doubly Linked List
- Exposure to http://visualgo.net/list
- Exposure to C++ STL list
  - http://en.cppreference.com/w/cpp/container/list