Data Structures

9. Linked List Variations

Roadmap

- List as an ADT
- An array-based implementation of lists

- Introduction to linked lists
- A pointer-based implementation in C++
- Variations of linked lists

Linked List – Advantages

- Access any item as long as external link to first item maintained
- Insert new item without shifting
- Delete existing item without shifting
- Can expand/contract (flexible) as necessary

Linked List – Disadvantages (1)

- Overhead of links
  - Used only internally, pure overhead
- If dynamic, must provide
  - Destructor
  - Copy constructor
  - Assignment operator
- No longer have direct access to each element of the list
  - Many sorting algorithms need direct access
  - Binary search needs direct access
- Access of nth item now less efficient
  - Must go through first element, then second, and then third, etc.

Linked List – Disadvantages (2)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	

Linked List – Disadvantages (3)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = *null_value*

Linked List – Disadvantages (4)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = *null_value* **If** list is empty Set head to point to new node

Linked List – Disadvantages (5)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = *null_value* **If** list is empty Set head to point to new node **Else** Traverse list to find last node Set next part of last node to point to new node

This is the inefficient part

Some Applications

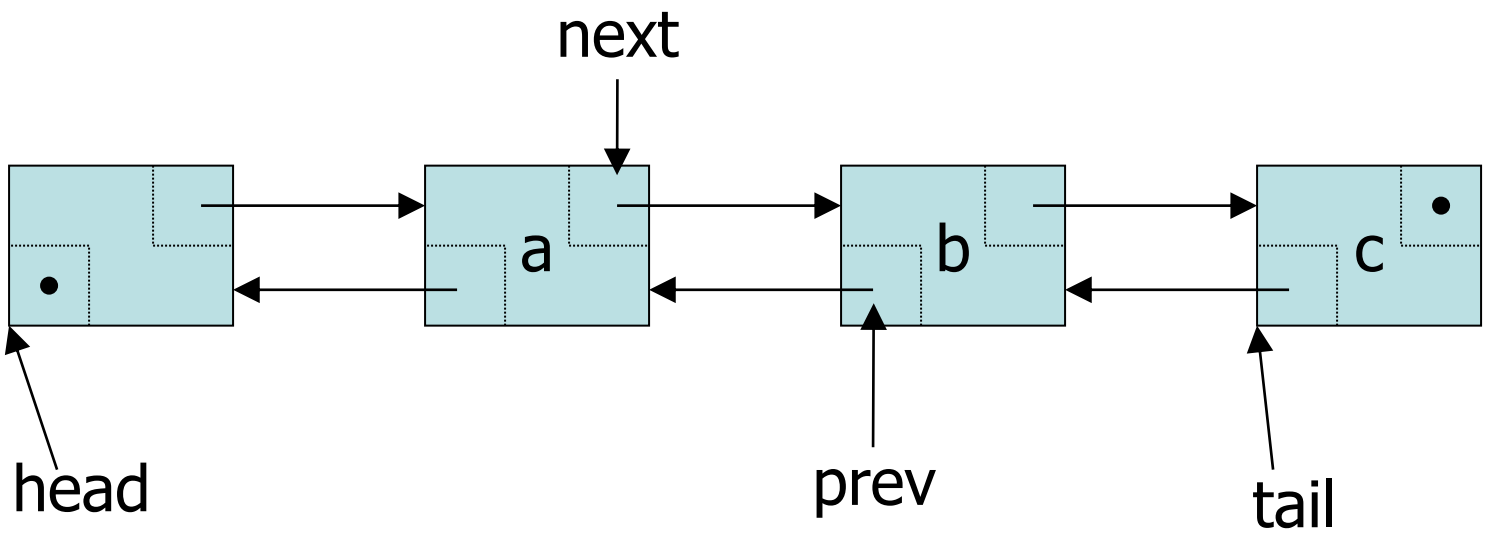
- Applications that maintain a Most Recently Used (MRU) list
  - For example, a linked list of file names
- Cache in the browser that allows to hit the BACK button
  - A linked list of URLs
- Undo functionality in Photoshop or Word
  - A linked list of state
- A list in the GPS of the turns along your route

Can we traverse the linked list in the reverse direction!

Doubly Linked List

Doubly Linked List

- Every node contains the address of the previous node except the first node
  - Both forward and backward traversal of the list is possible



Node Class

- `DoubleListNode` class contains three data members
  - `data`: double-type data in this example
  - `next`: a pointer to the next node in the list
  - `Prev`: a pointer to the pervious node in the list

```
class DoubleListNode {
   public:
      double data;              // data
      DoubleListNode * next;    // pointer to next
      DoubleListNode * prev;    // pointer to previous
};
```

List Class

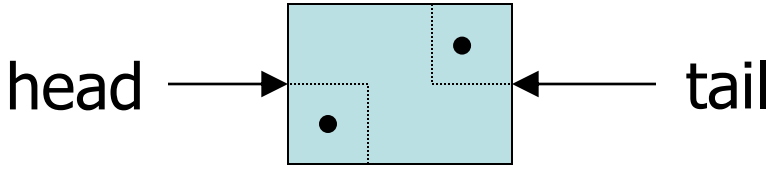
- List class contains two pointers
  - `head`: a pointer to the first node in the list
  - `tail`: a pointer to the last node in the list
  - Since the list is empty initially, `head` and `tail` are set to `NULL`

```
class List {
   public:
      List(void) { head = NULL; tail = NULL; }  // constructor
      ~List(void);                              // destructor

      . . .

   private:
      DoubleListNode * head;
      DoubleListNode * tail;
};
```

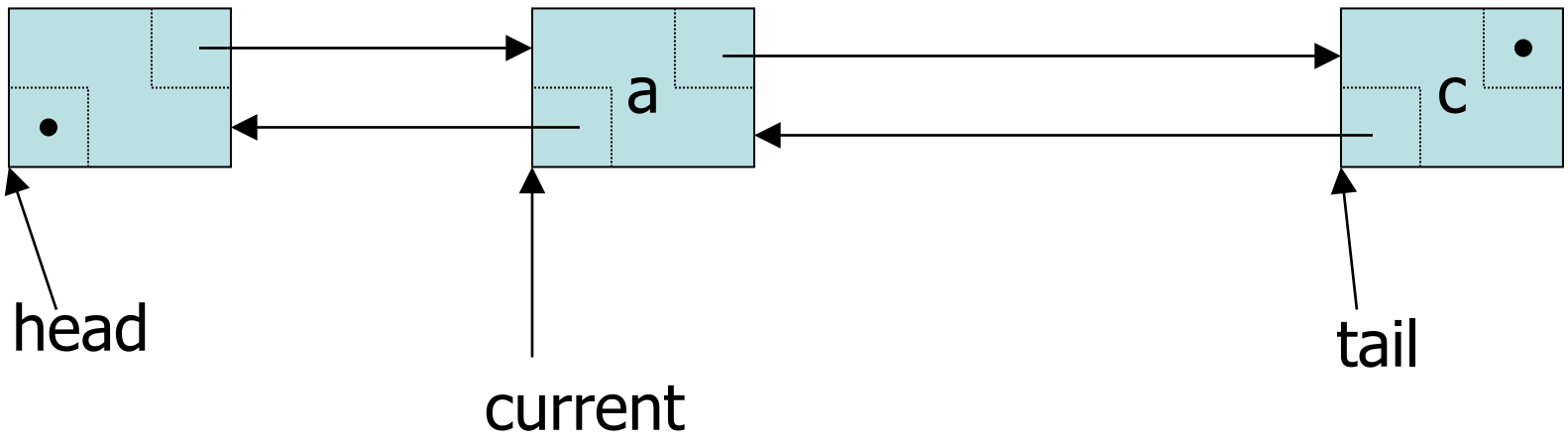
Adding First Node




```
// Adding first node
head = new DoubleListNode;
head->next = null;
head->prev = null;
tail = head;
```

Inserting a Node in Doubly Linked List (1)

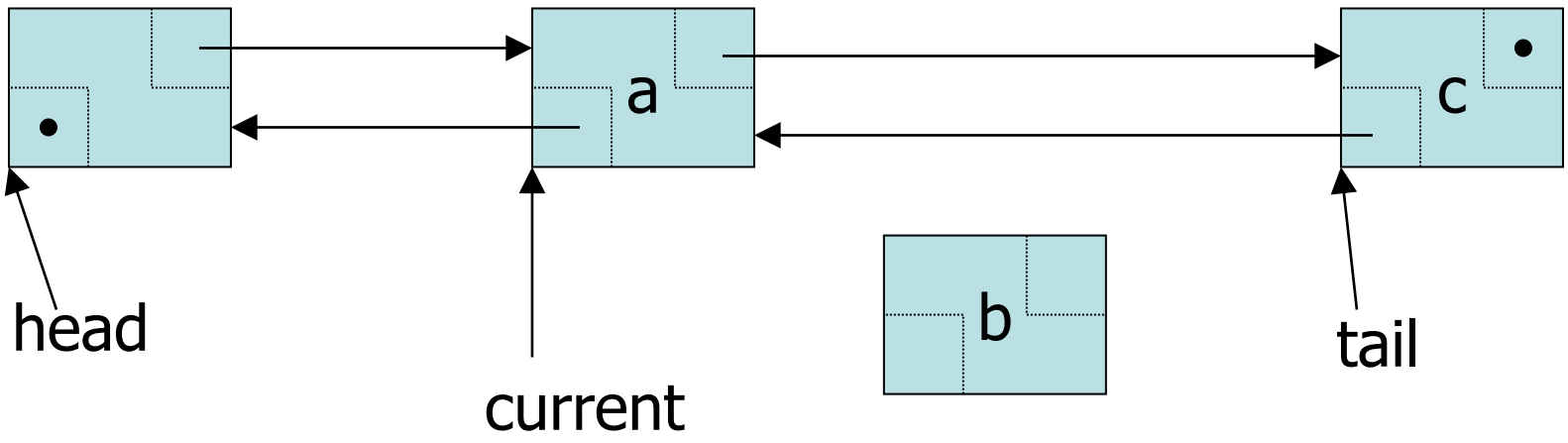
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (2)

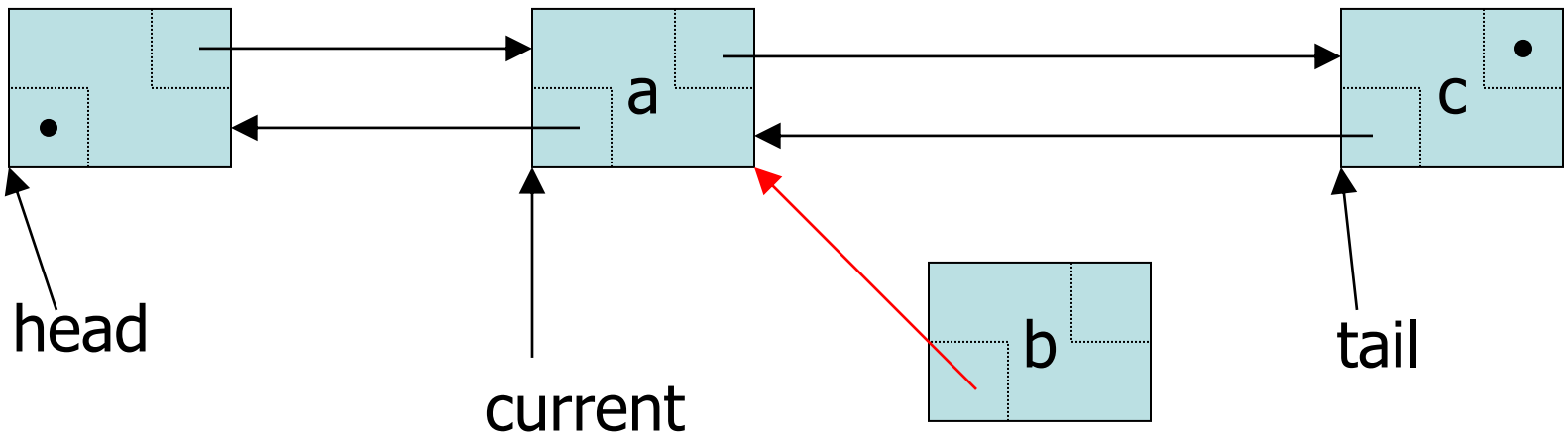
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (3)

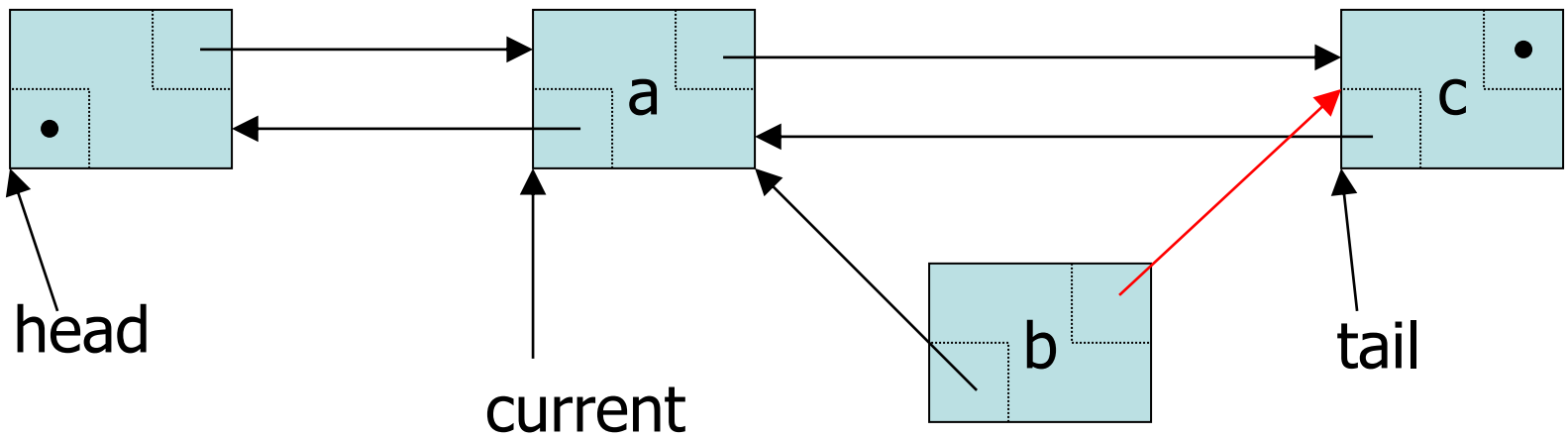
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (3)

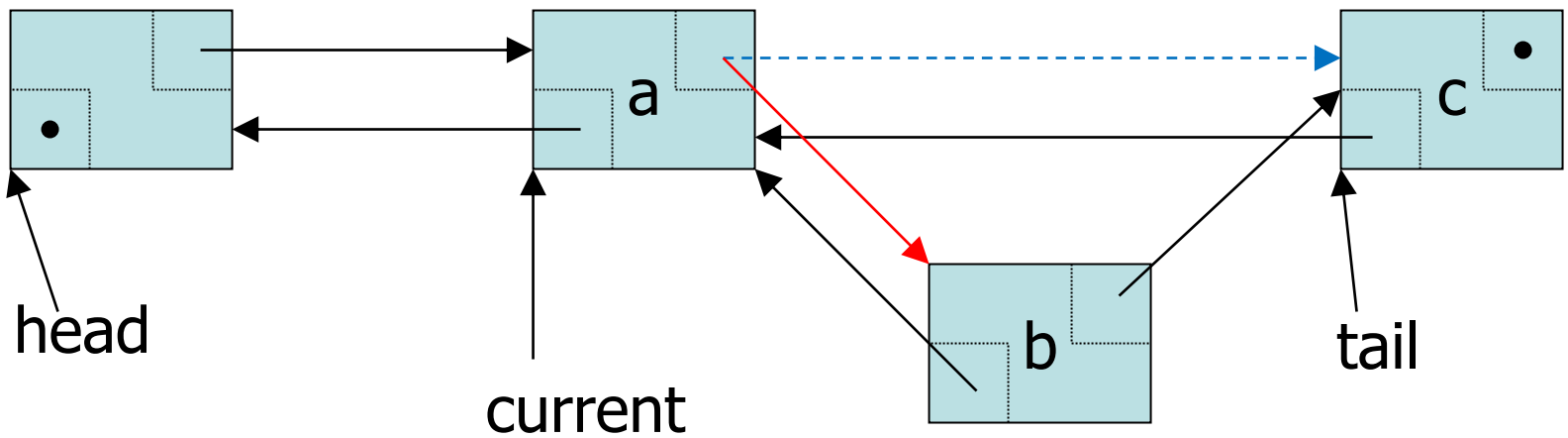
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (3)

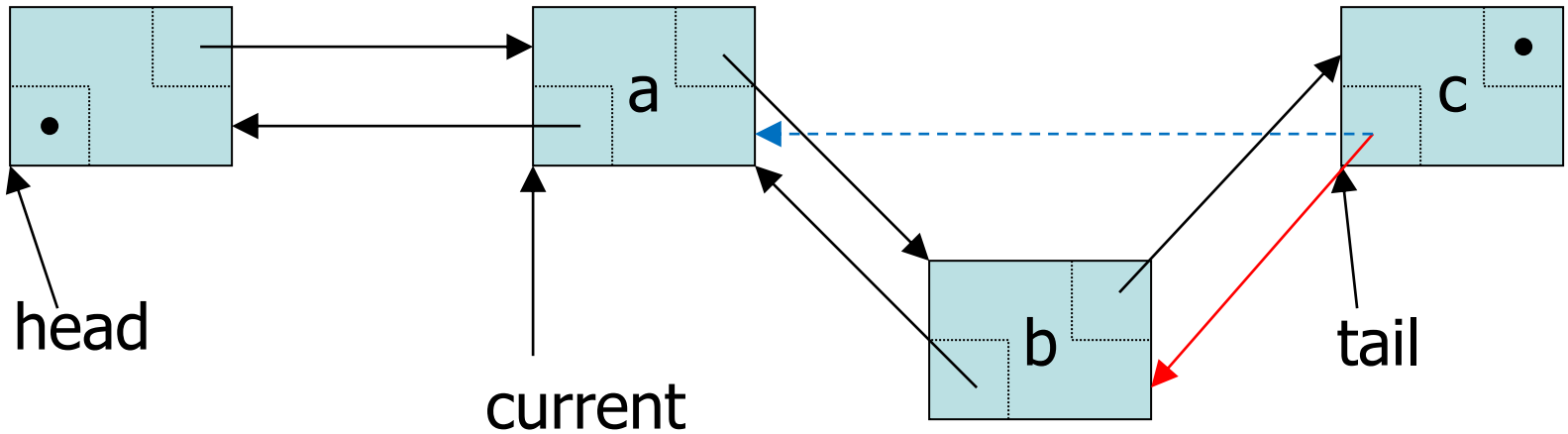
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;   //Current->next = newNode
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (3)

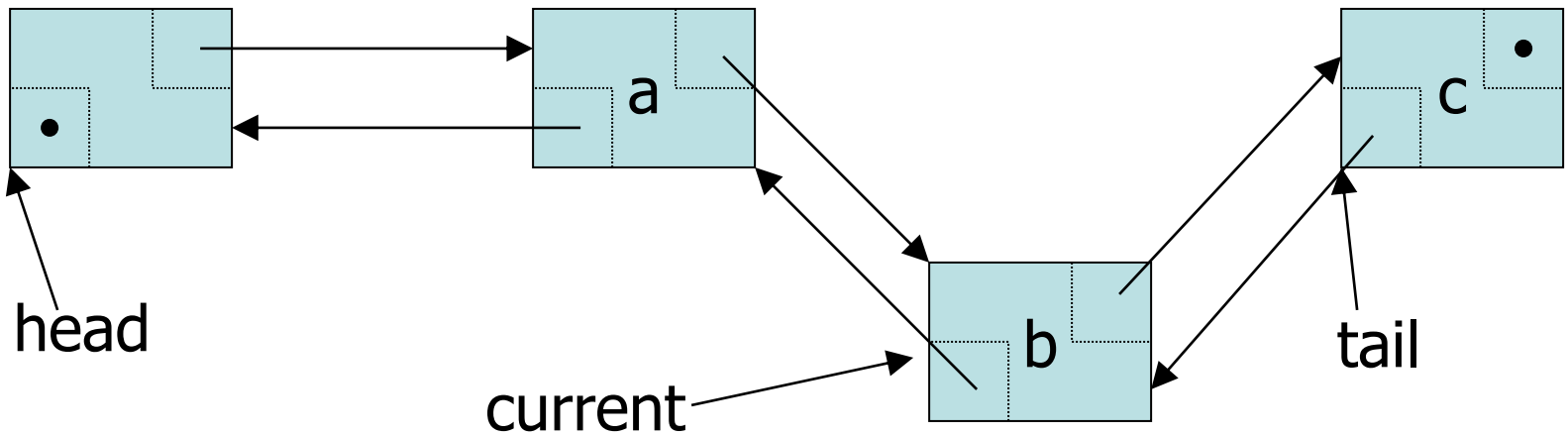
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting a Node in Doubly Linked List (3)

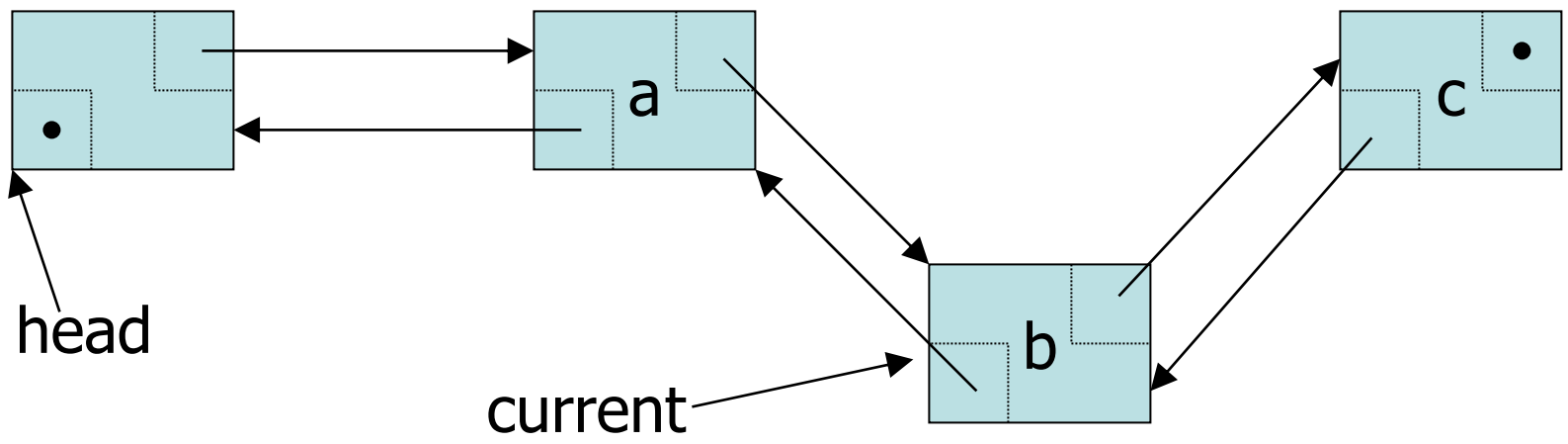
- To add a new item after the linked list node pointed by `current`




```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Deleting a Node From Doubly Linked List

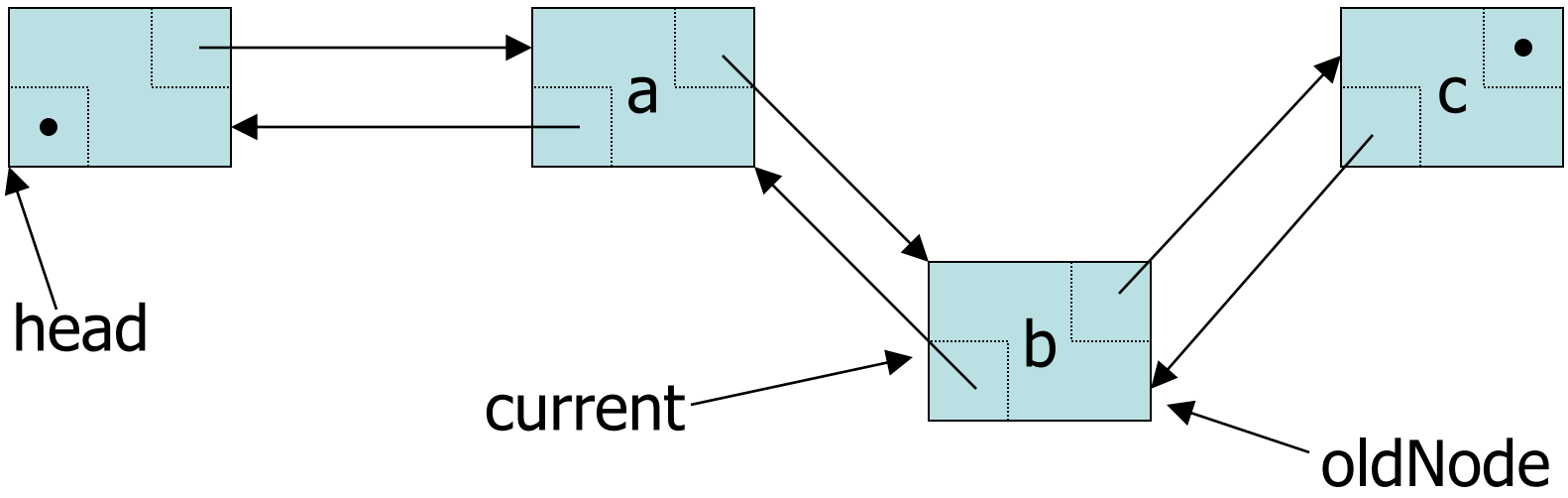
- Suppose `current` points to the node to be deleted from the list




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

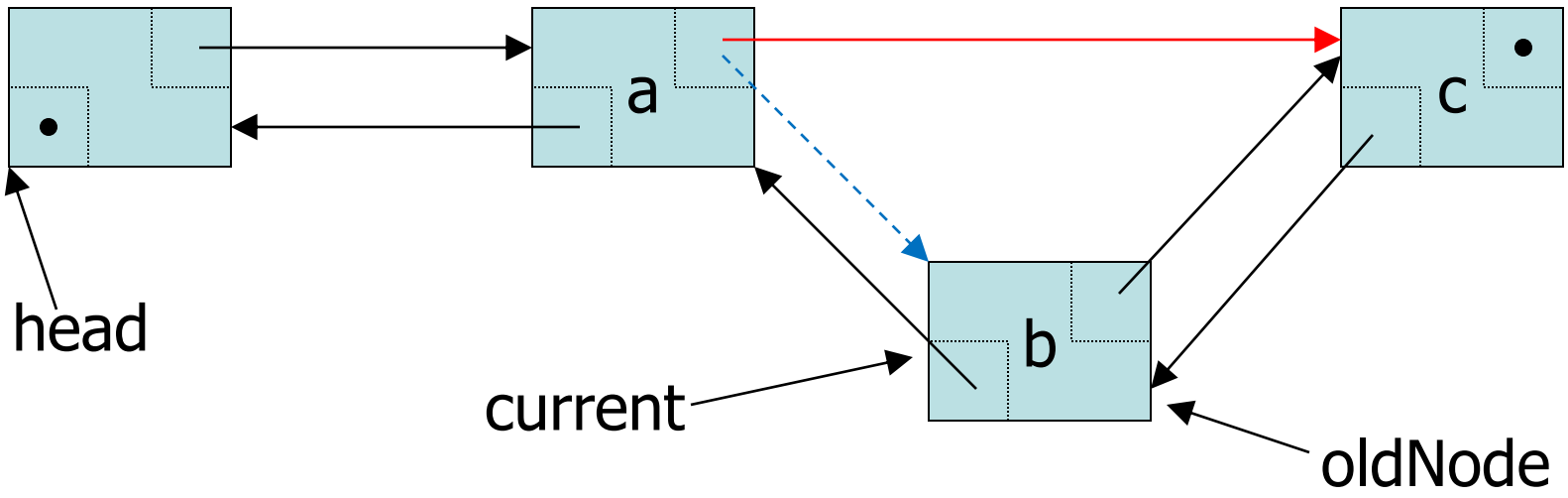
- Suppose `current` points to the node to be deleted from the list




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

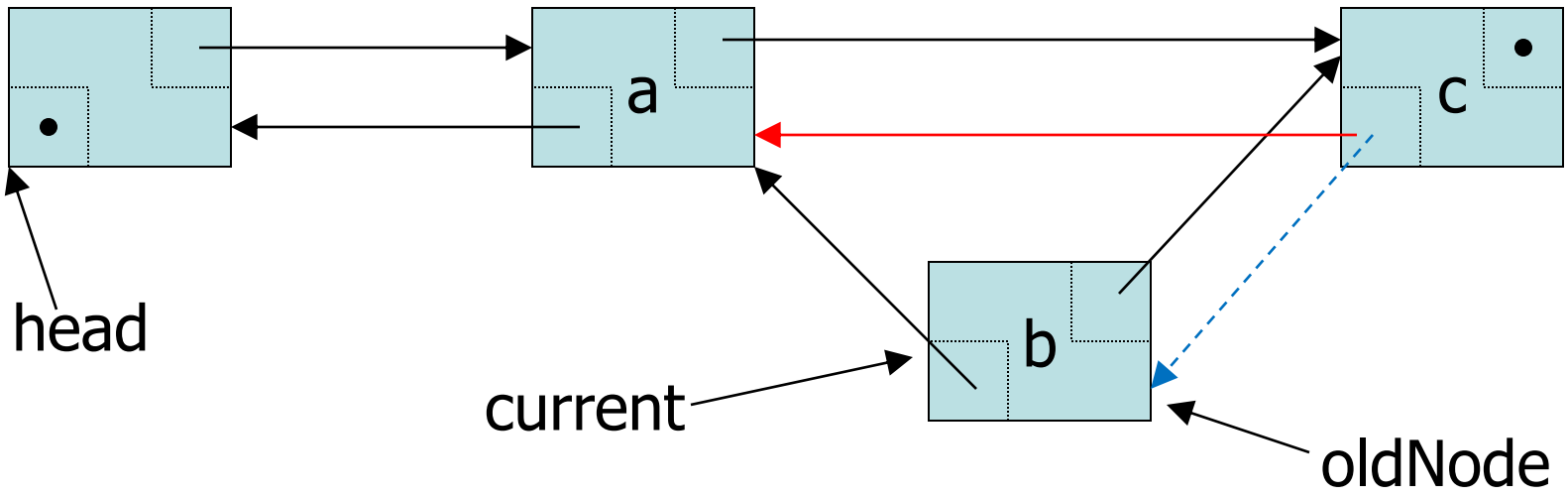
- Suppose `current` points to the node to be deleted from the list




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

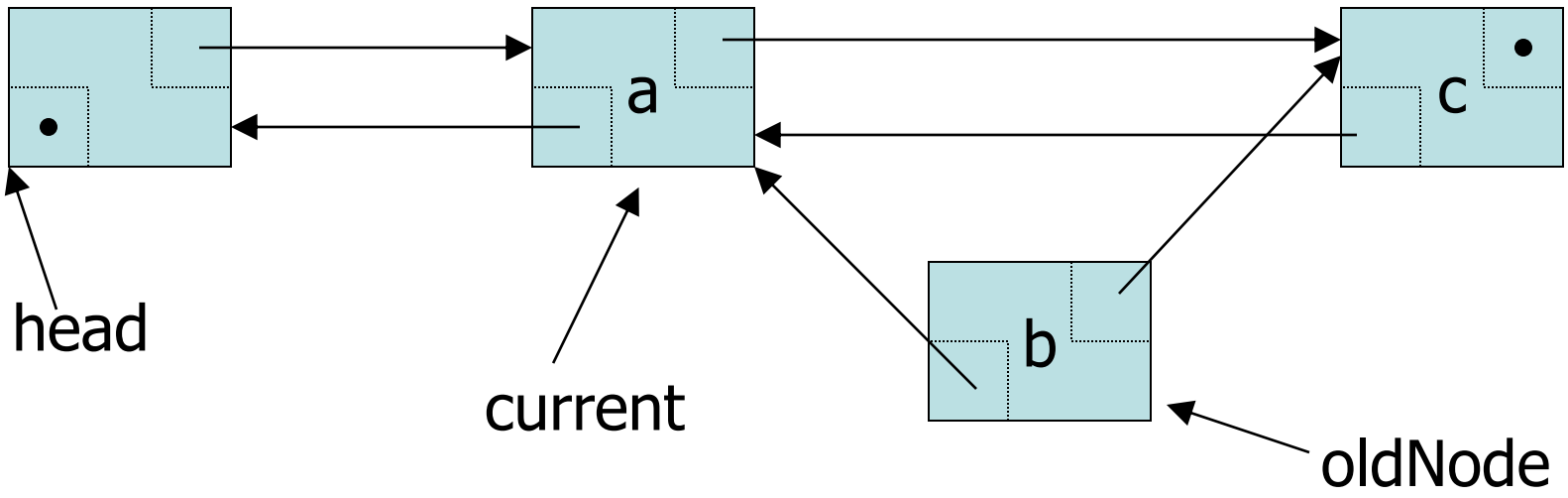
- Suppose `current` points to the node to be deleted from the list




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

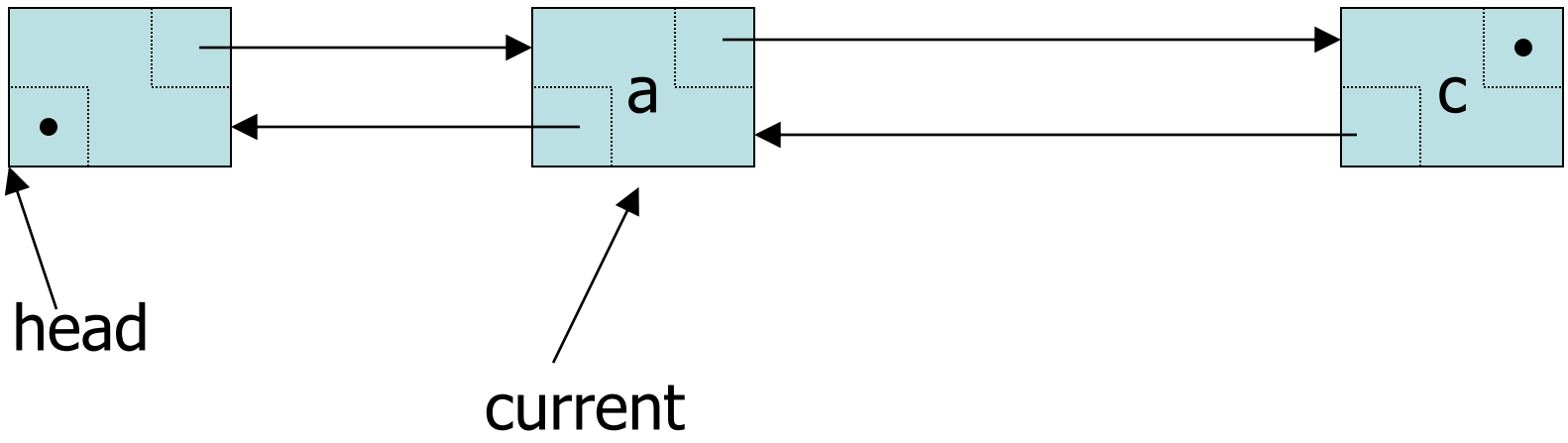
- Suppose `current` points to the node to be deleted from the list




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

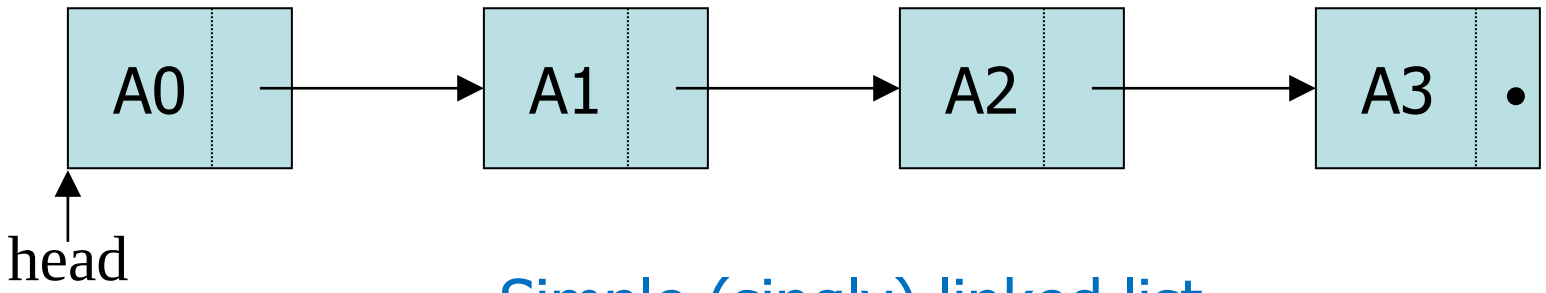



```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

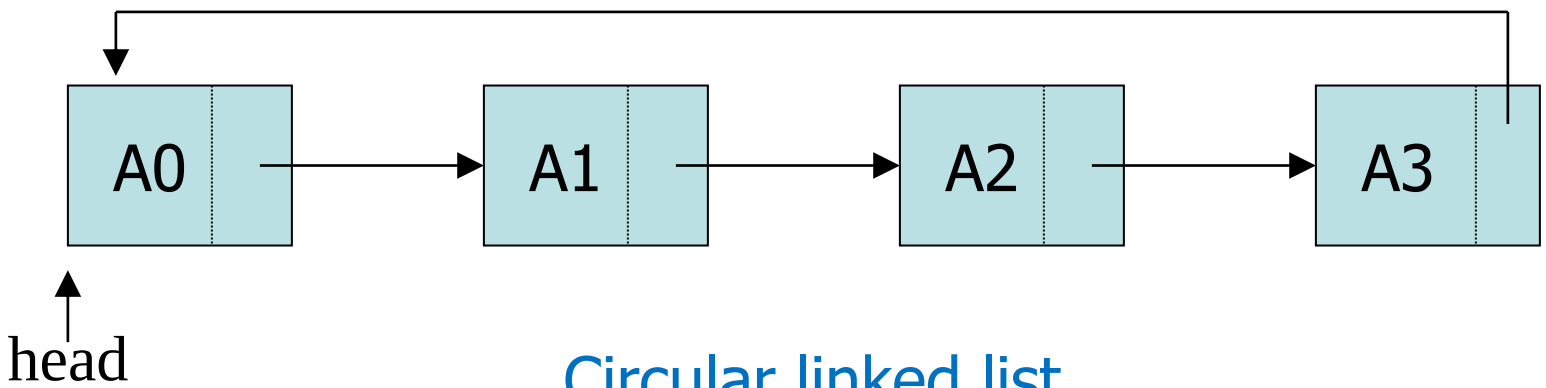
Circular Linked List

Circular Linked List

- A linked list in which the last node points to the first node



Simple (singly) linked list



Circular linked list

Advantages of Circular Linked List

- Whole list can be traversed by starting from any point
  - Any node can be starting point
  - What is the stopping condition?

- Fewer special cases to consider during implementation
  - All nodes have a node before and after them

- Used in the implementation of other data structures
  - Circular linked lists are used to create circular queues
  - Circular doubly linked lists are used for implementing Fibonacci heaps

Disadvantages of Circular Linked List

- Finding end of list and loop control is harder
  - No NULL to mark beginning and end

Any Question So Far?

Linked List – Disadvantages (2)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = *null_value* **If** list is empty Set head to point to new node **Else** Traverse list to find last node Set next part of last node to point to new node

This is the inefficient part