Data Structures
(Linked List)

Lecture#7

Agenda

- Linked-List: Pros and Cons
- Reversing a Linked-List
- Variations of the linked-list

Linked List – Advantages

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

Linked List – Disadvantages (1)

- Overhead of links
  - Extra space for pointers with each item-node, pure overhead
- Dynamic, must provide
  - Destructor (to destroy all the dynamic allocations one-by-one)
- 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

- **Main:** Implement many other abstract datatypes such as stacks, queues, binary trees, and fib. heaps etc.

- 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 processor
  - 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?

Reverse the list

- Input: head ->30->25->20->15->10->5

- Reversed : head ->5->10->15->20->25->30

- Use three pointer (Current, previous, Next)
- While traversing the list just invert the links
- When a final node approaches → store the address of end-node in the head pointer

```
void List :: Reverse(){

    Node *upcoming,*prev, *current;
    current = head;
    prev = NULL;
    while (current!=NULL)
    {
            upcoming = current->next;
            current->next=prev;
            prev = current;
            current = upcoming;
    }
    head = prev;
}
```

Linked List Variations

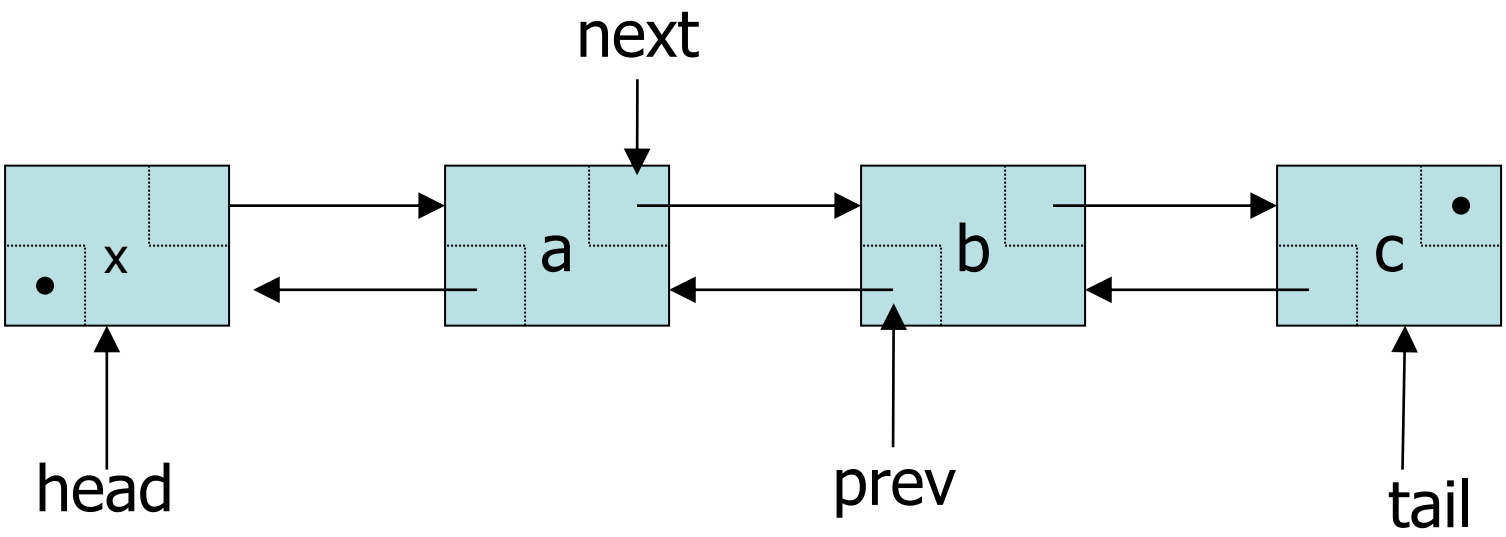
Roadmap

- Variations of linked lists
  - Doubly linked lists
  - Circular Linked lists

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

- Node 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 Node {
public:
        double data;  // data
        Node*  next;  // pointer to next
        Node*  prev; // pointer to previous
};
```

List Class

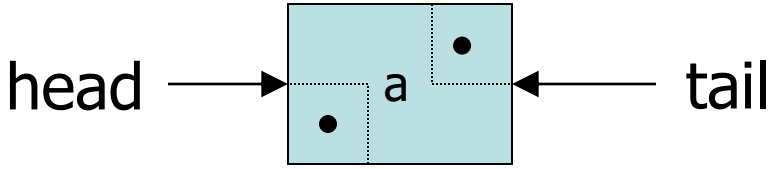
- List class contains two pointers
  - `head`: a pointer to the first node in the list
  - `tail`: a pointer to the last node in the list
  - Since the list is empty initially, `head` and `tail` are set to `NULL`

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

        . . .

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

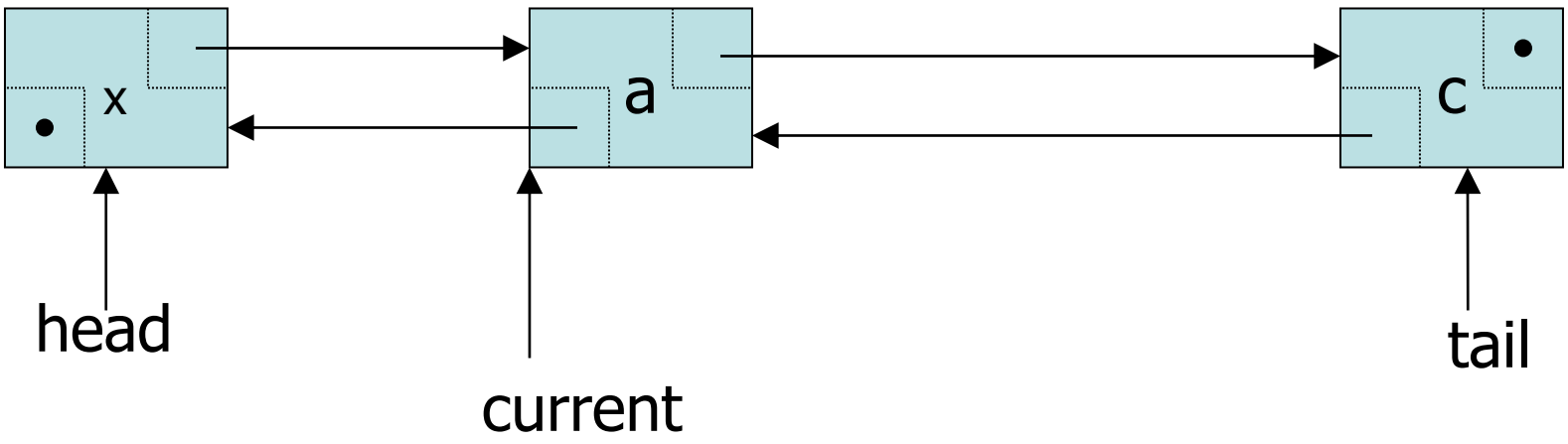
Adding First Node




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

Inserting a Node in Doubly Linked List (1)

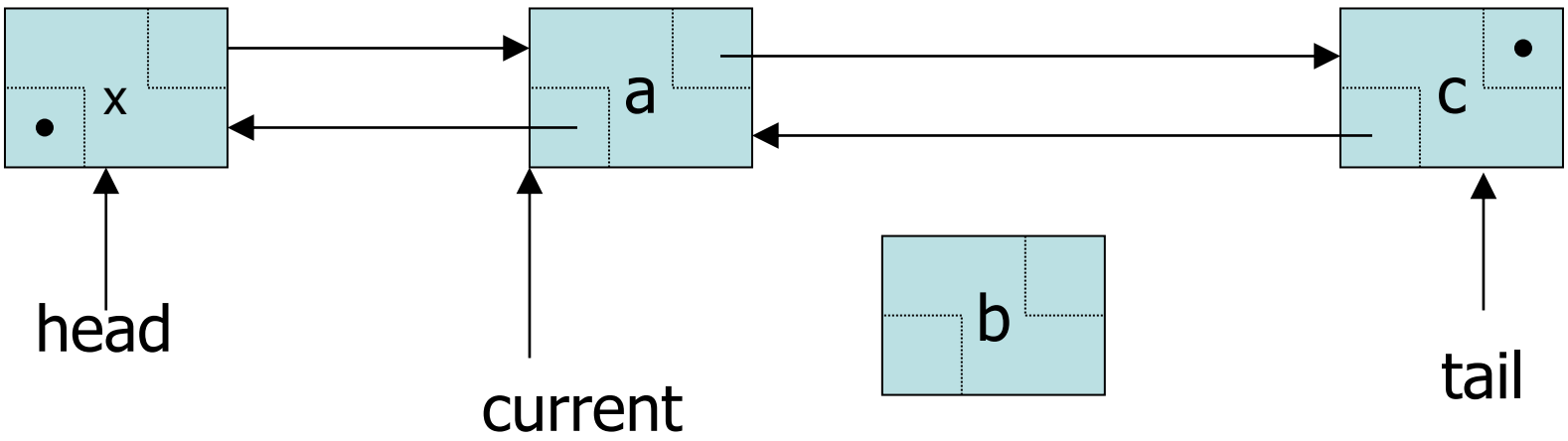
- To add a new item after the linked list node pointed by current




```
newNode = new Node
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)

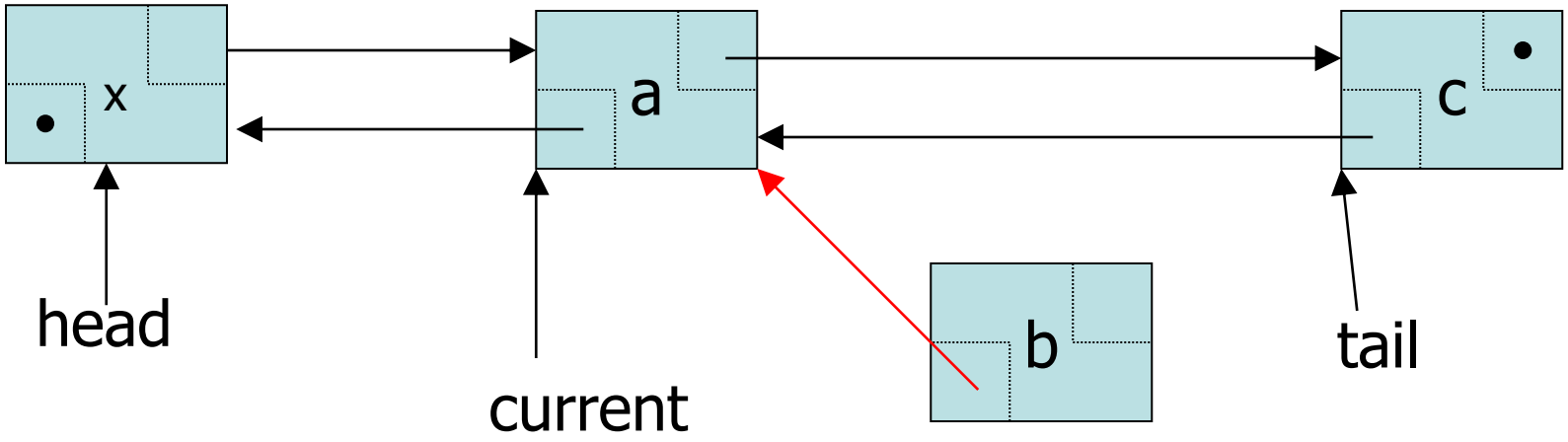
- To add a new item after the linked list node pointed by `current`




```
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)

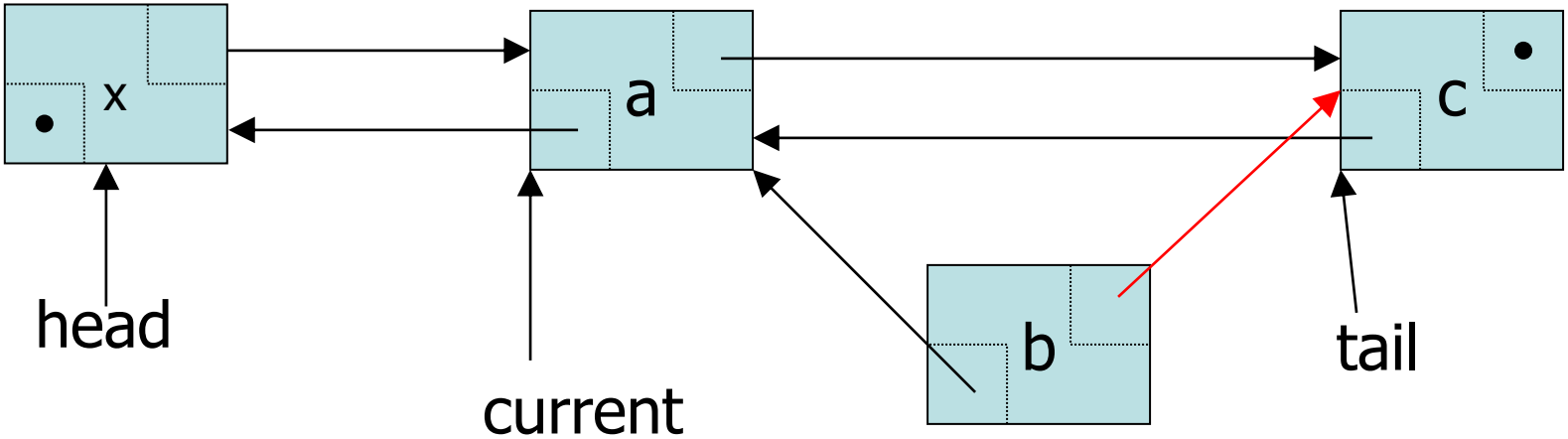
- To add a new item after the linked list node pointed by current




```
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)

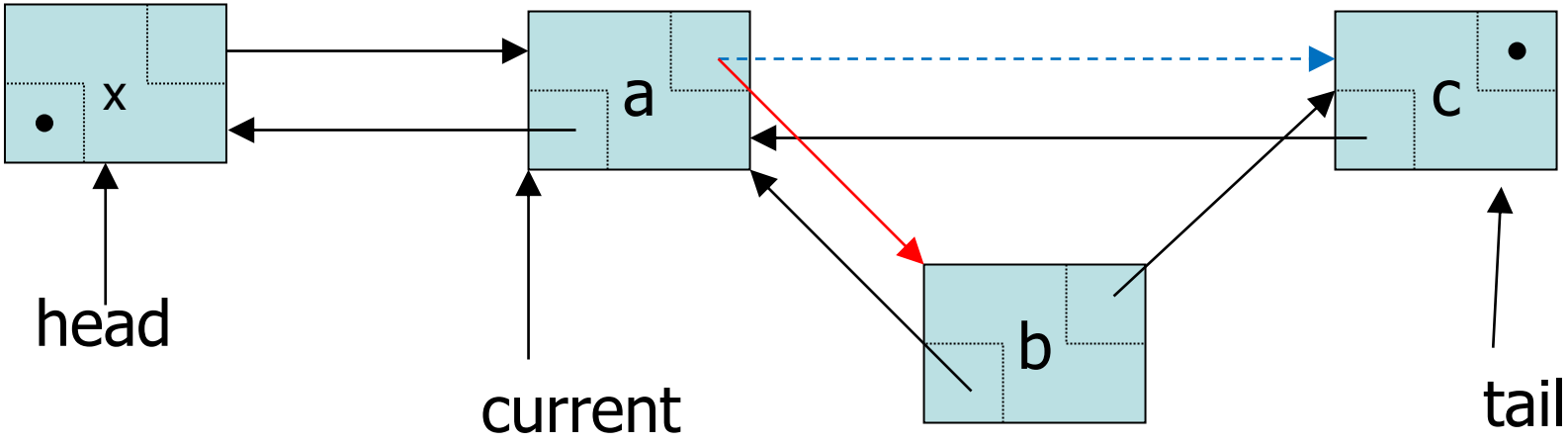
- To add a new item after the linked list node pointed by current




```
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)

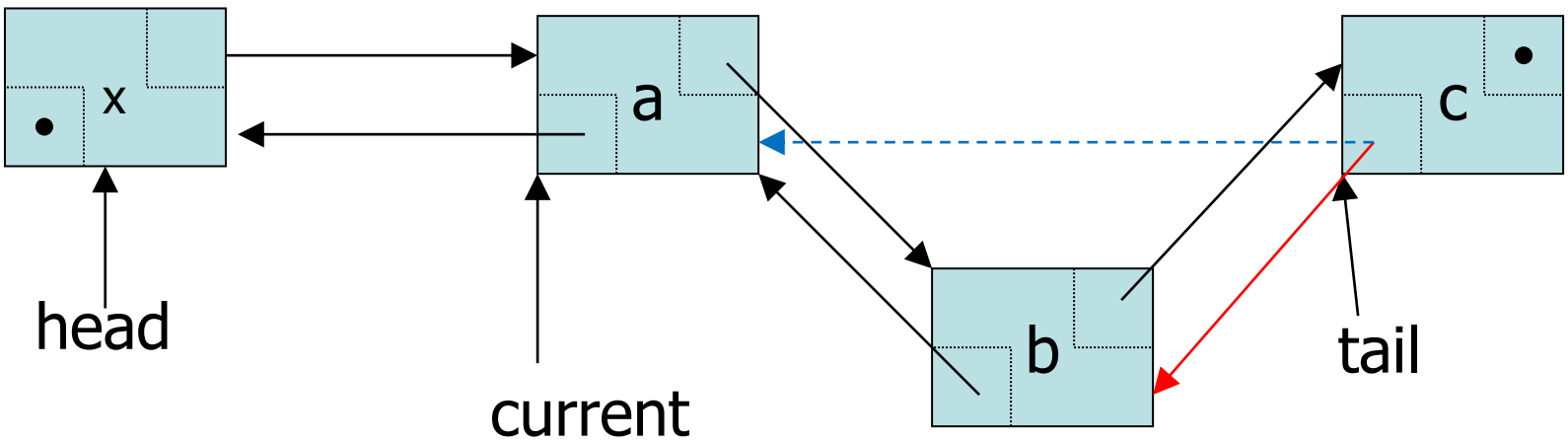
- To add a new item after the linked list node pointed by current




```
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)

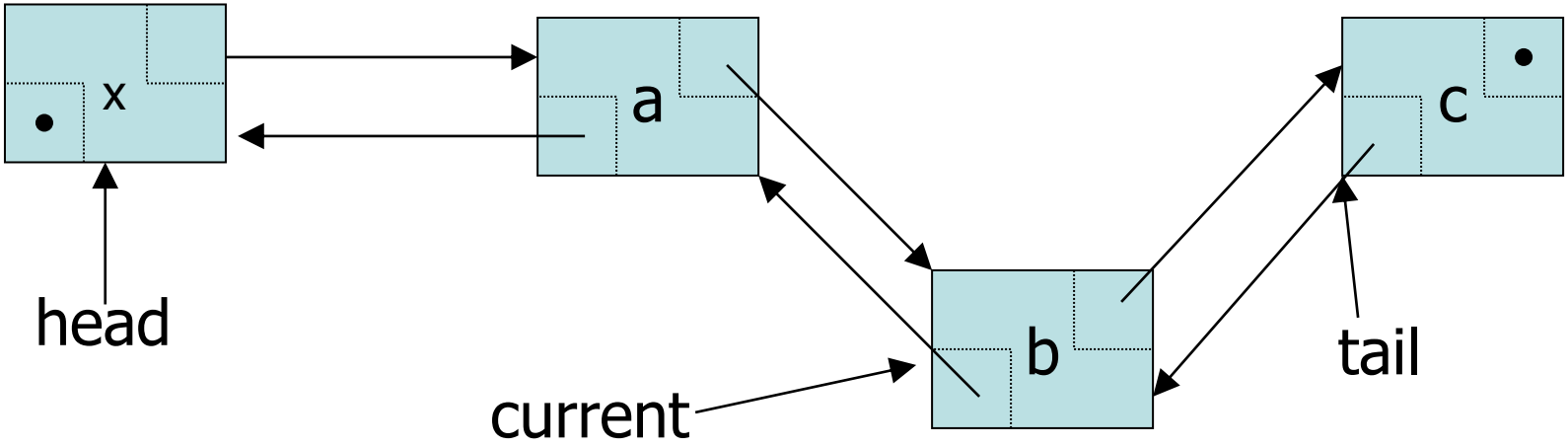
- To add a new item after the linked list node pointed by current




```
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)

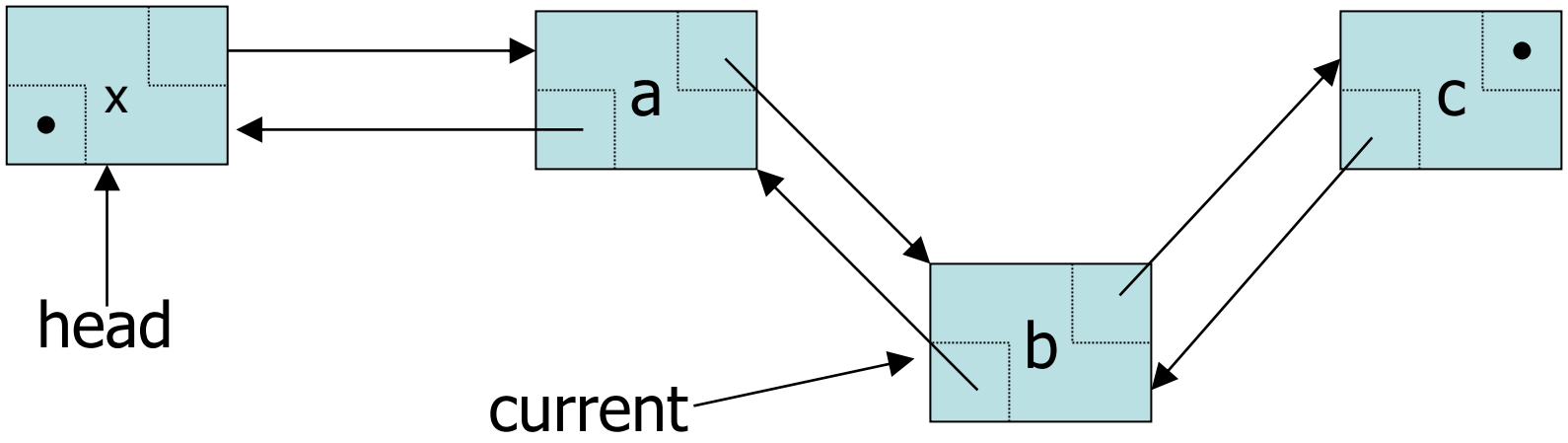
- To add a new item after the linked list node pointed by current




```
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

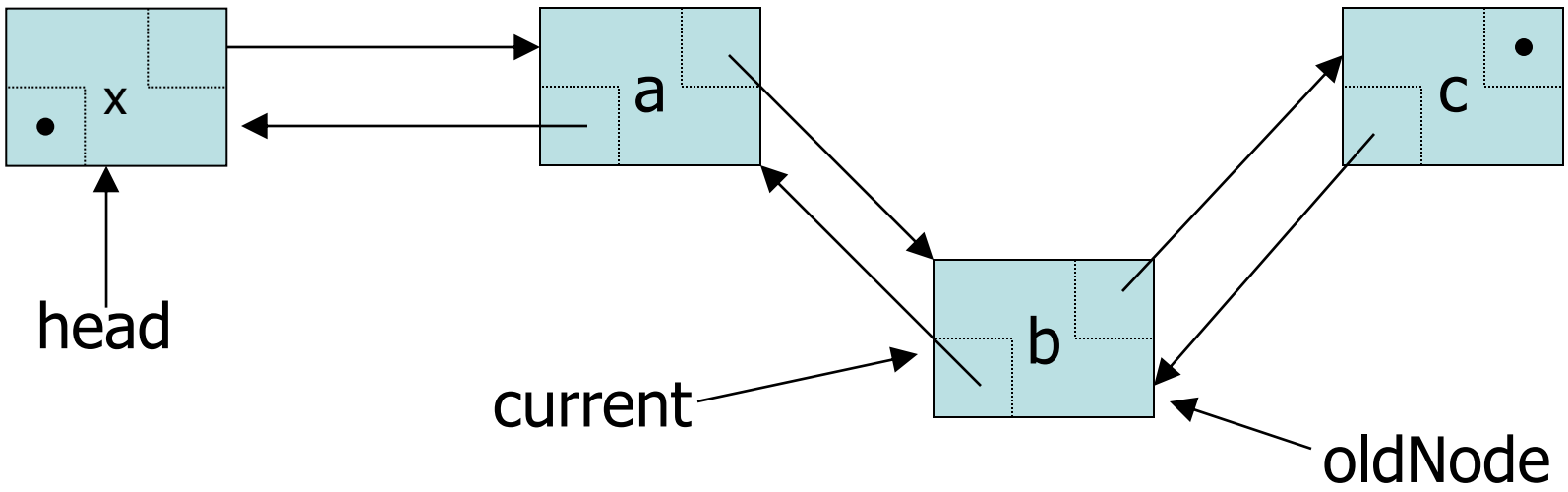
- Suppose `current` points to the node to be deleted from the list




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

Deleting a Node From Doubly Linked List

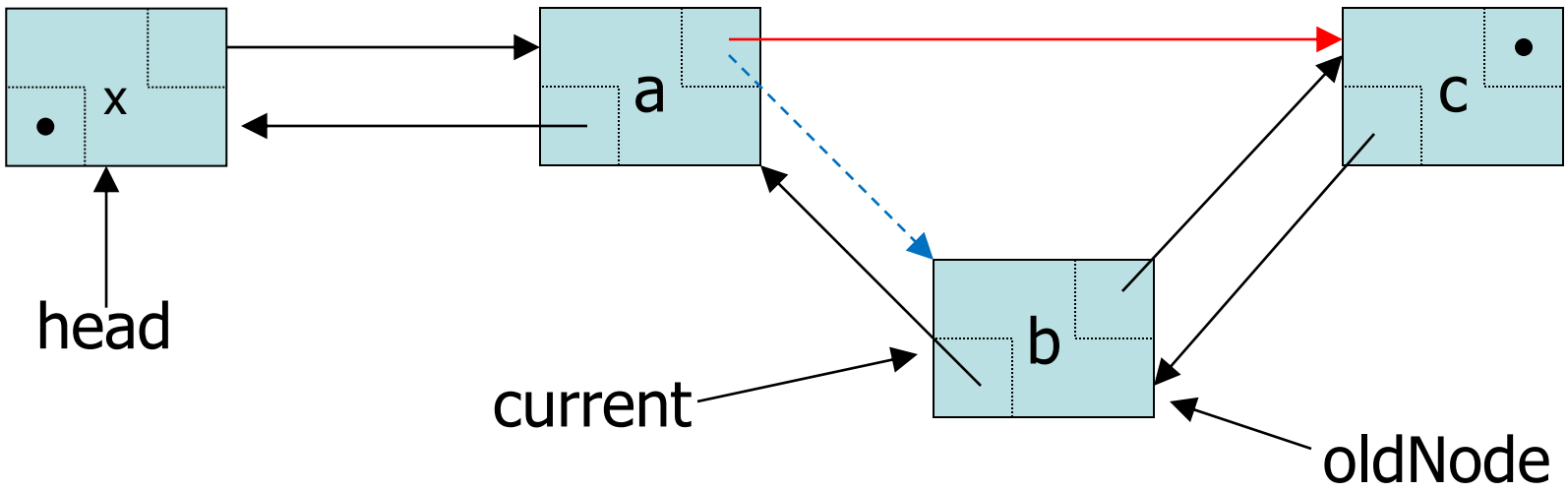
- Suppose current points to the node to be deleted from the list




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

Deleting a Node From Doubly Linked List

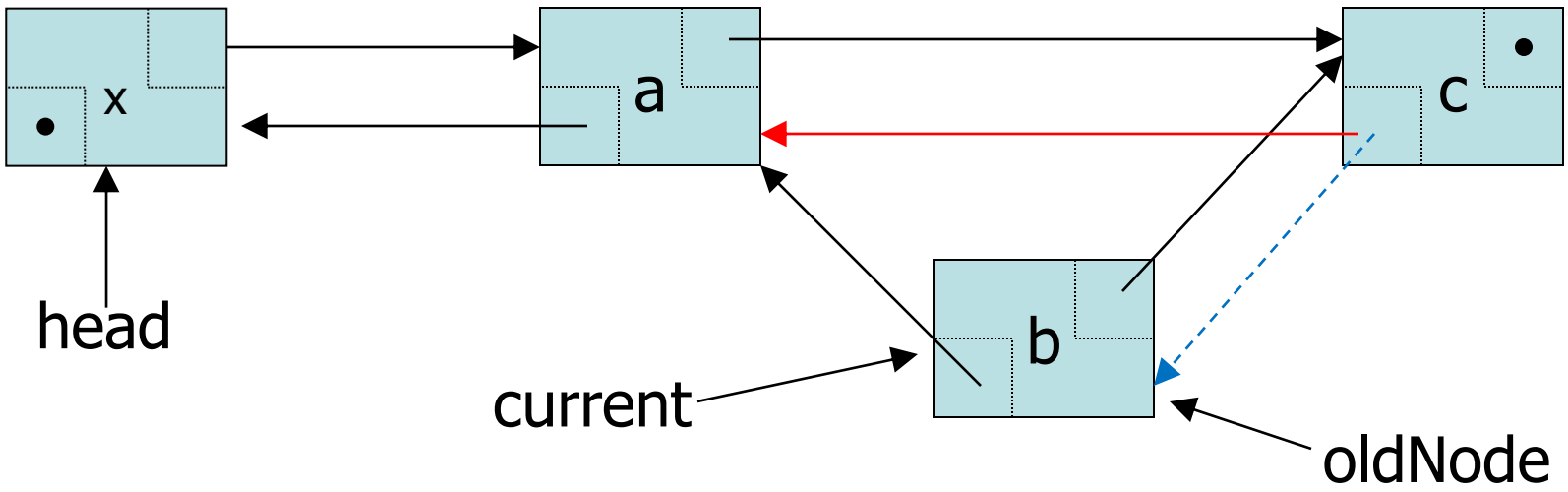
- Suppose current points to the node to be deleted from the list




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

Deleting a Node From Doubly Linked List

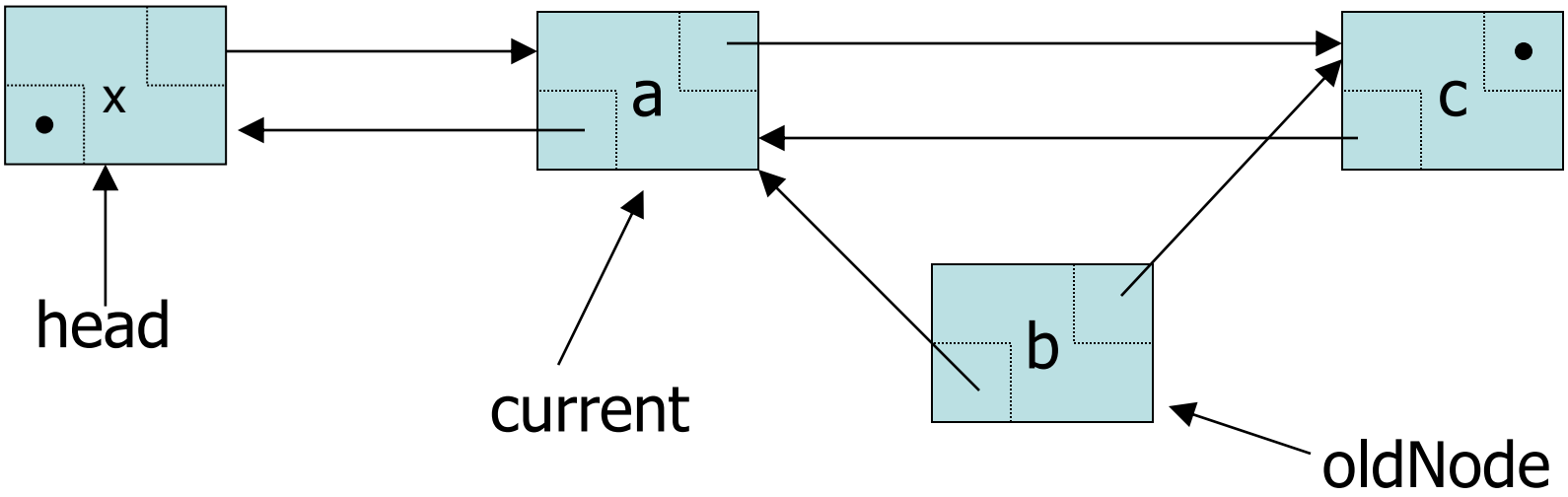
- Suppose current points to the node to be deleted from the list




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

Deleting a Node From Doubly Linked List

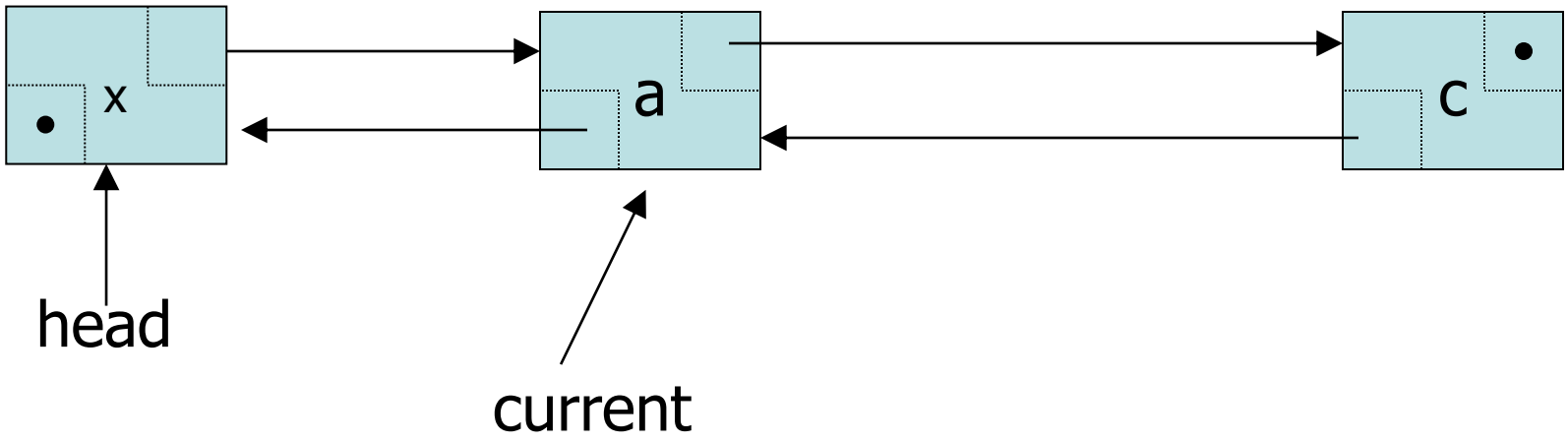
- Suppose current points to the node to be deleted from the list




```
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

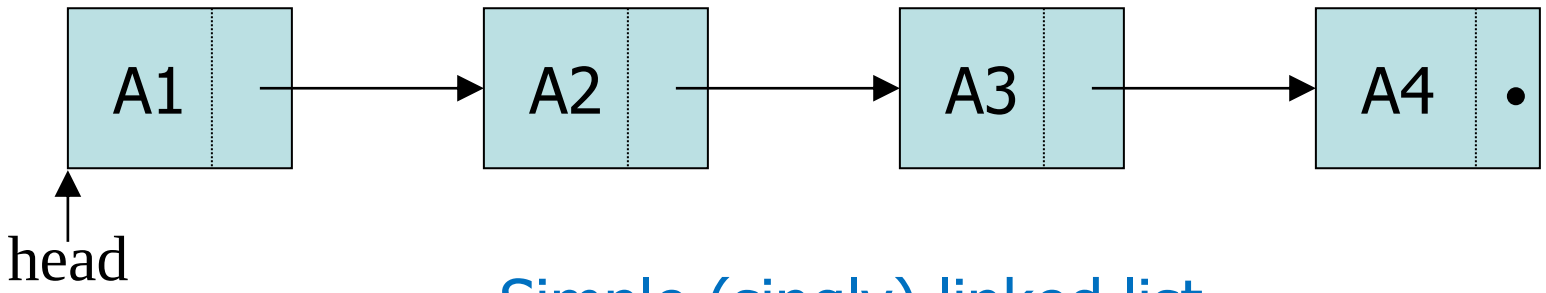



```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

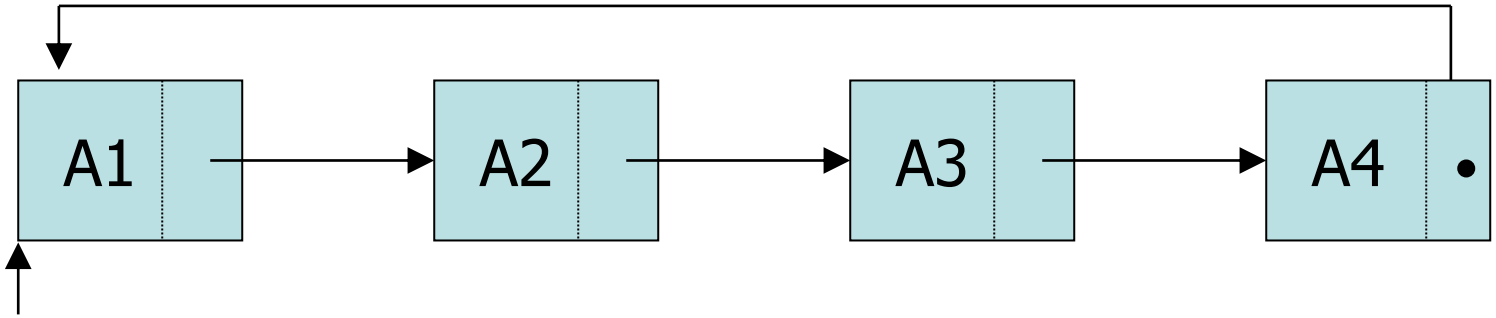
Circular Linked List

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?
    - If node from which traversing was started, is encountered again then, the traversing will stop at this point.

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

- 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

Any Question So Far?