CSC180: Lecture 33

Wael Aboulsaadat

wael@cs.toronto.edu

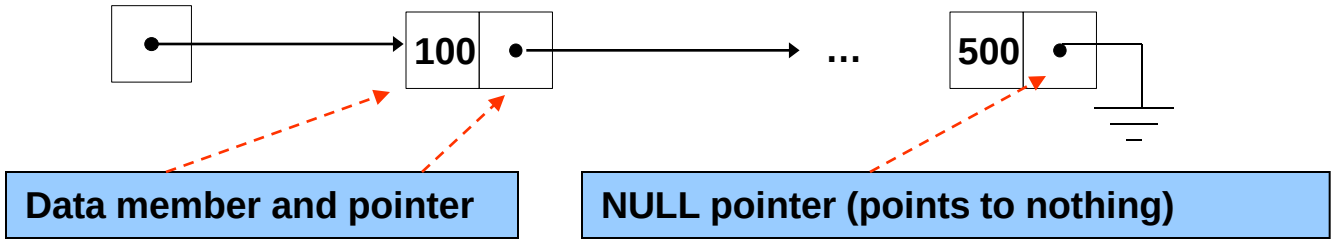
http://portal.utoronto.ca/

Acknowledgement: These slides are partially based on the slides supplied with Prof. Savitch book: Problem Solving with C

Linked Lists

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **`NULL`** pointer (**`0`**)
- Diagram of two self-referential structure objects linked together




```
struct node {
      int data;
      struct node *nextPtr;
   }
```

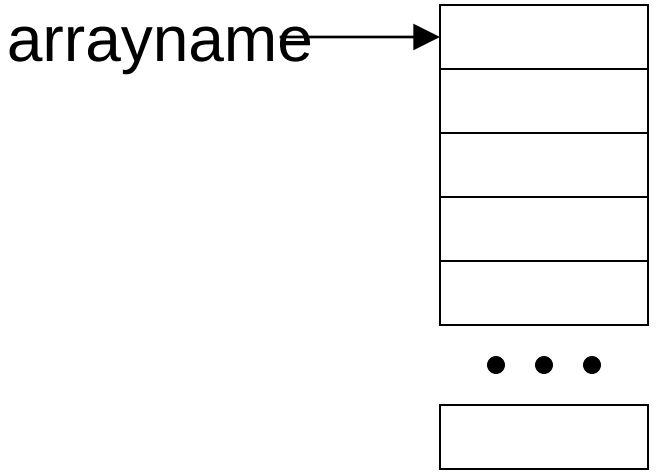
- `nextPtr`
  - Points to an object of type node
  - Referred to as a link

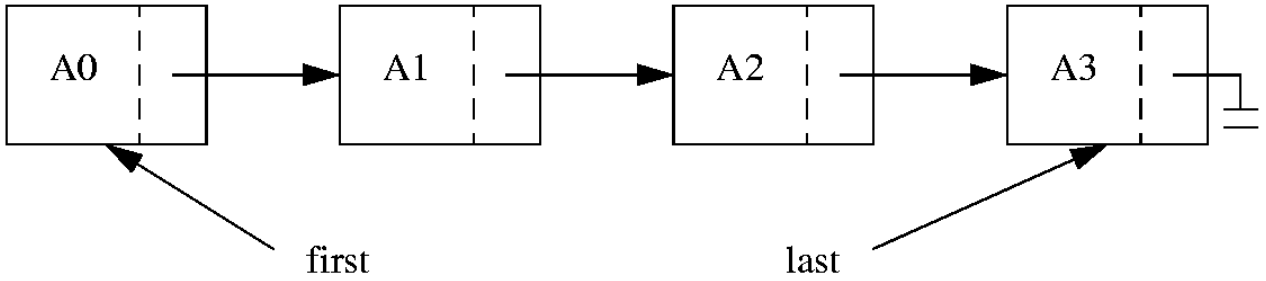
List Implementation using Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Link pointer in the last node is set to null to mark the list’s end

Comparison with Array

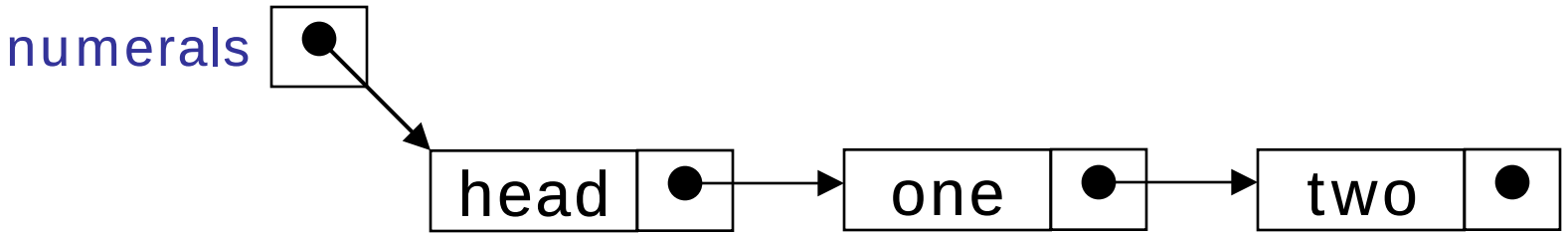
1. Arrays
   contiguous
   direct access of elements
   insertion / deletion difficult
2. Linked Lists
   noncontiguous
   must scan for element
   insertion /deletion easy



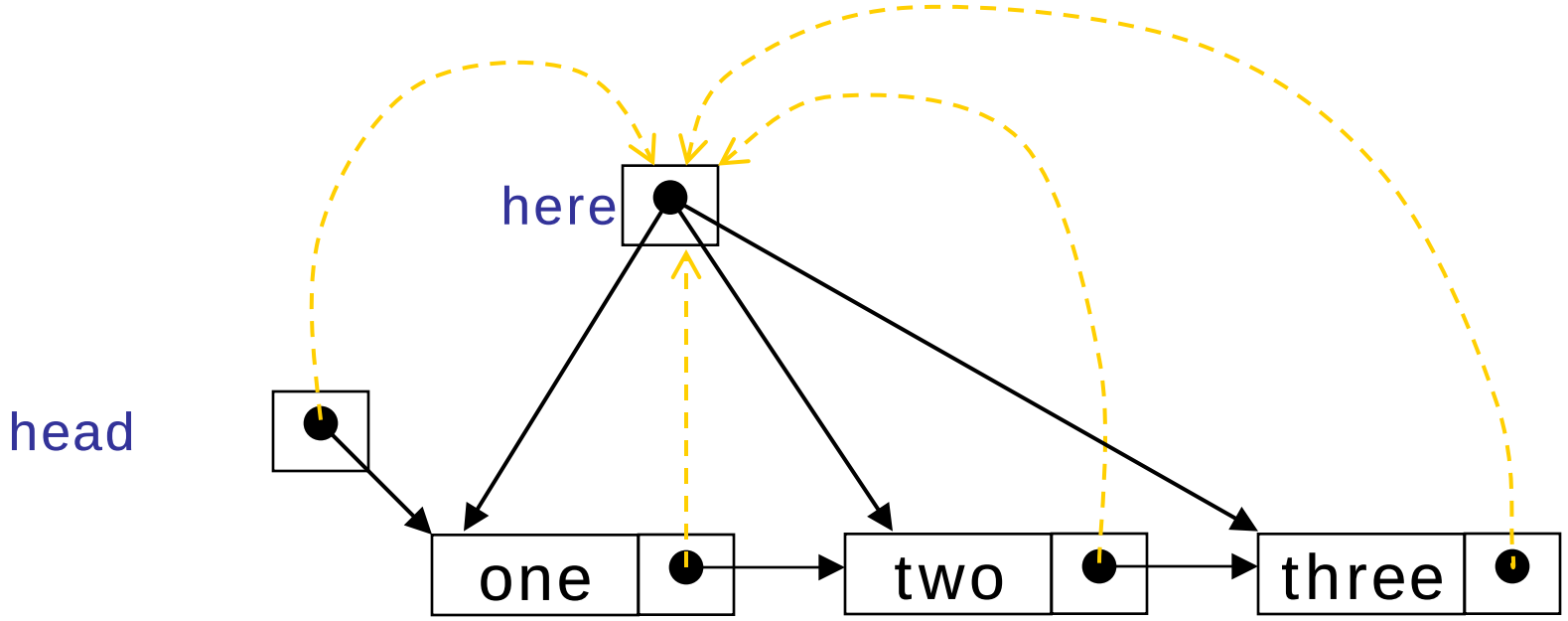


Using a header node

- A header node is just an initial node that exists at the front of every list, even when the list is empty
- The purpose is to keep the list from being null, and to point at the first element



Traversing a SLL (animation)

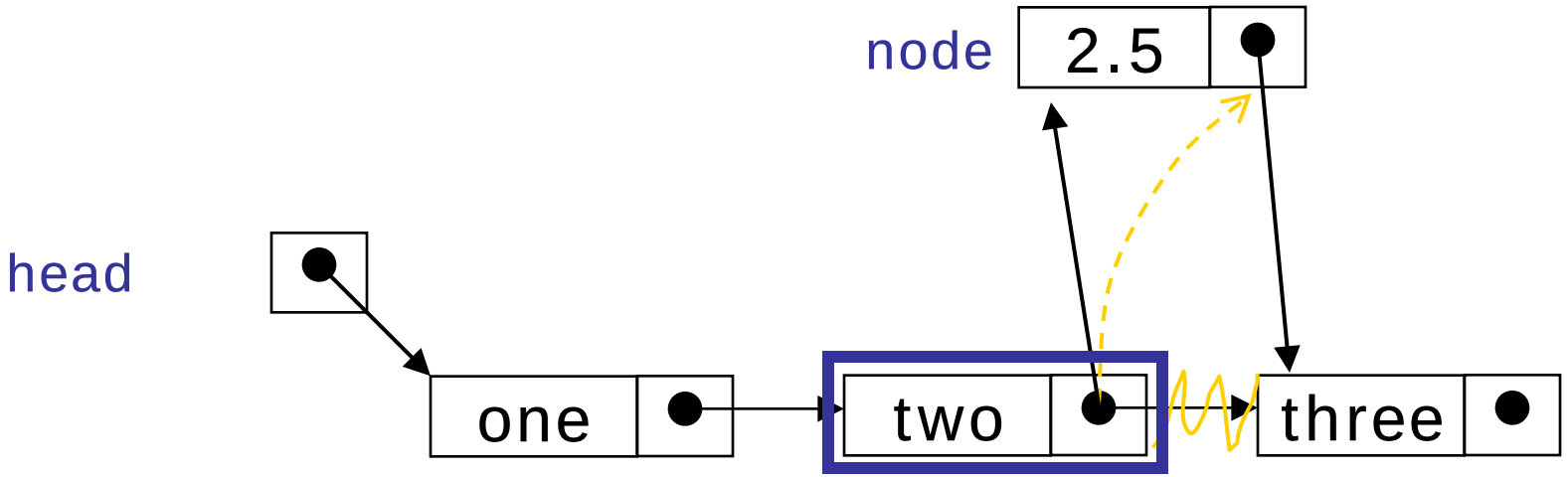



```
here = head;
while(here != NULL)
{
      // do something
       here = here->pNext;
}
```

Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting after (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

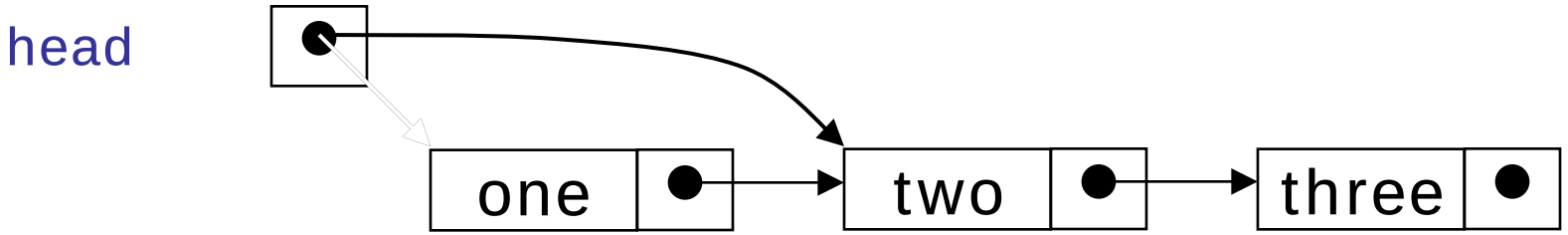
Then, change the link in the node that's already in the list

Deleting a node from a SLL

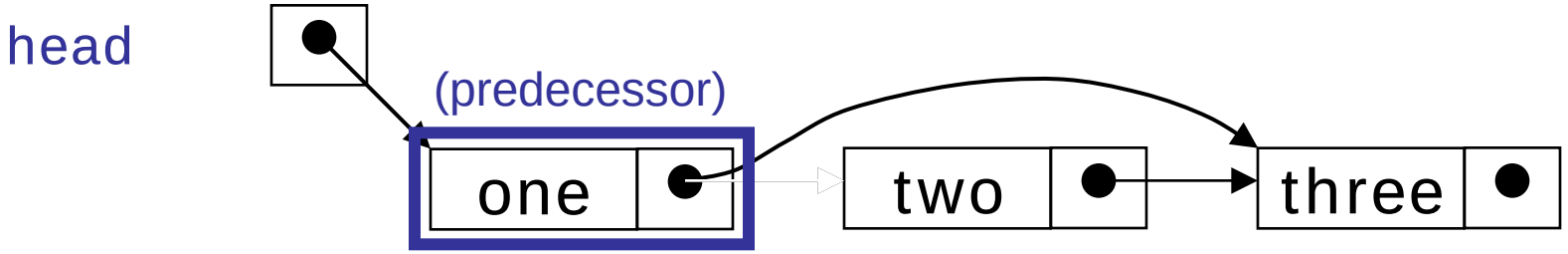
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor

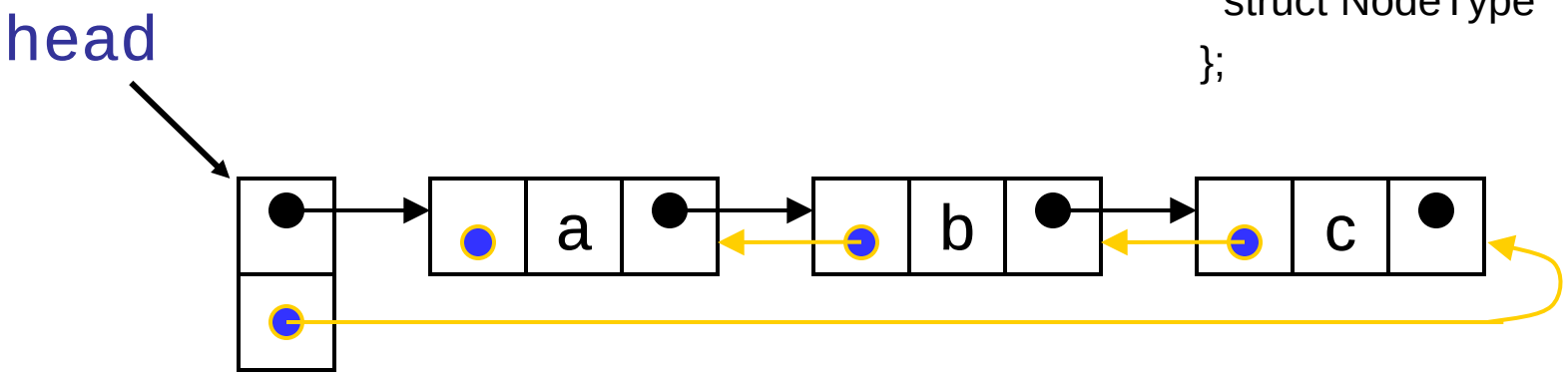


- Space occupied by deleted node(s) still have be returned to OS by calling free

Doubly-linked lists

- Here is a doubly-linked list (DLL):

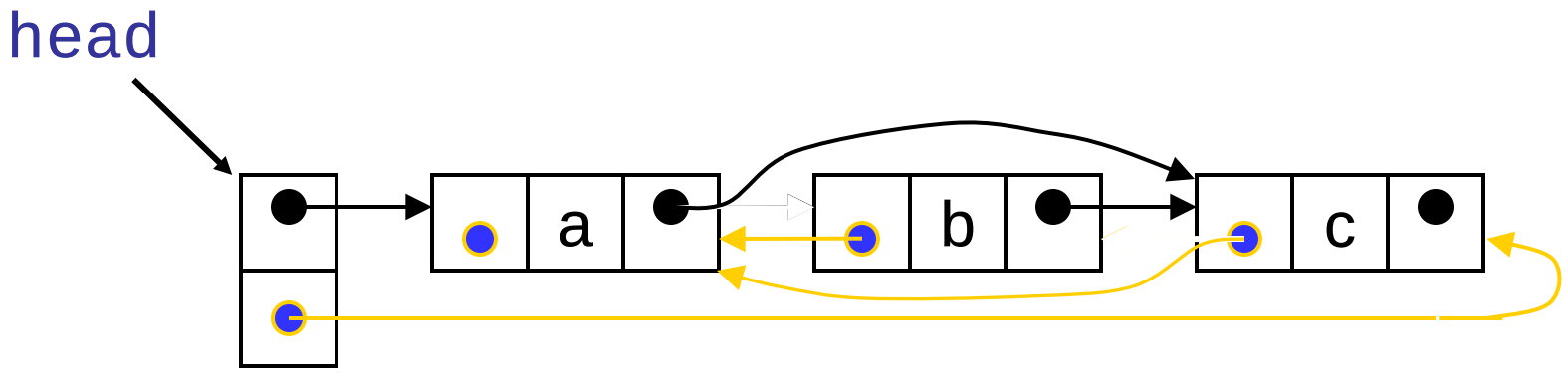
```
struct NodeType {
  ItemType info;
  struct NodeType   *next;
  struct NodeType   *back;
};
```



- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

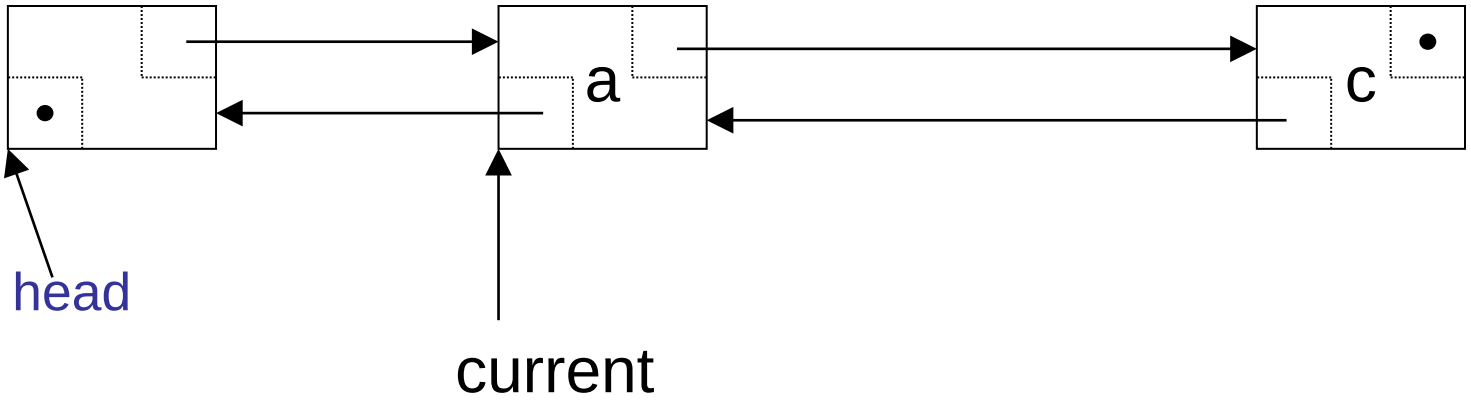
Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



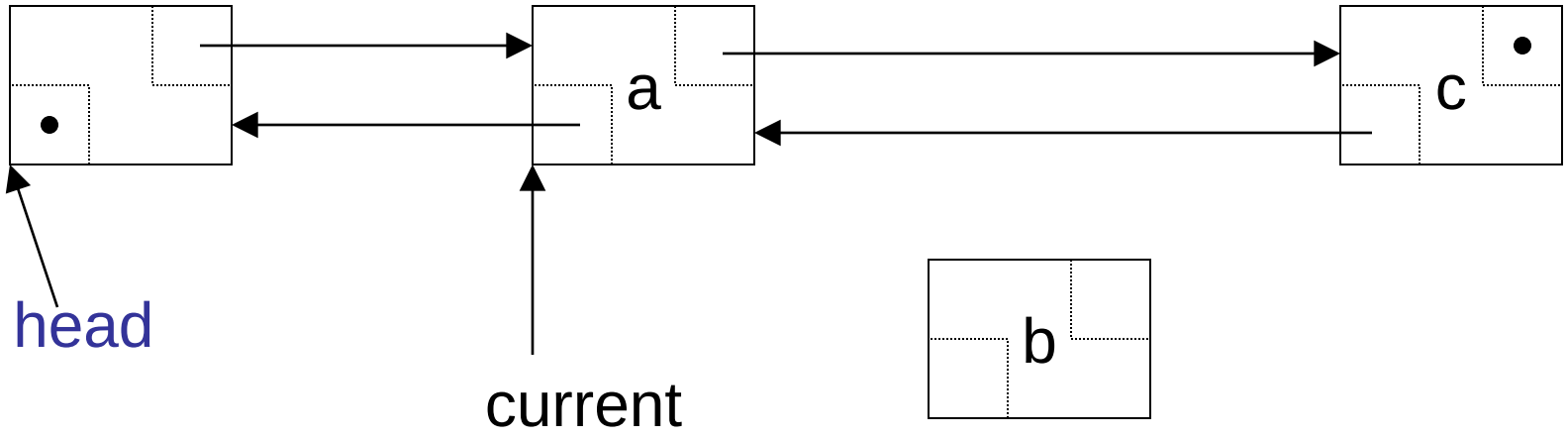
  - We don’t have to do anything about the links in node b
  - Still have to call free on b
  - Deletion of the first node or the last node is a special case

Inserting into a Doubly Linked List



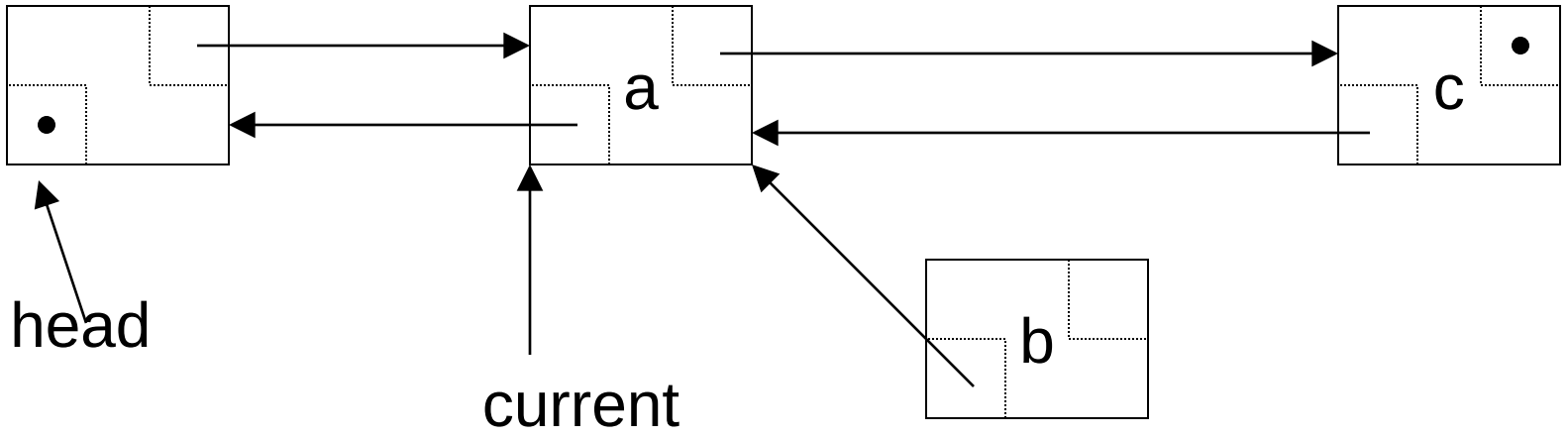

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



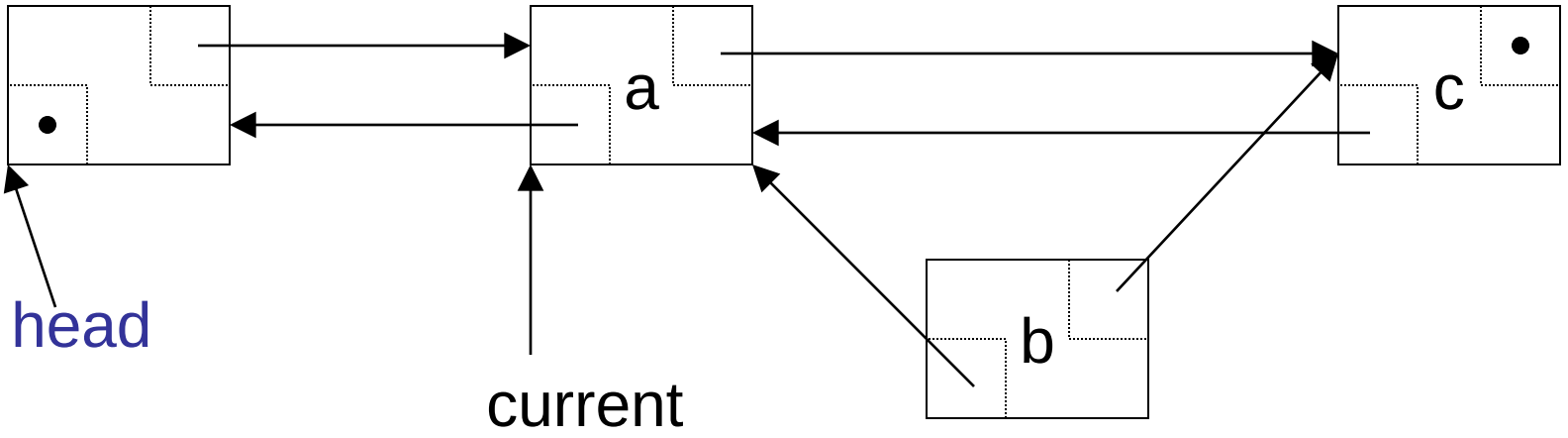

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List



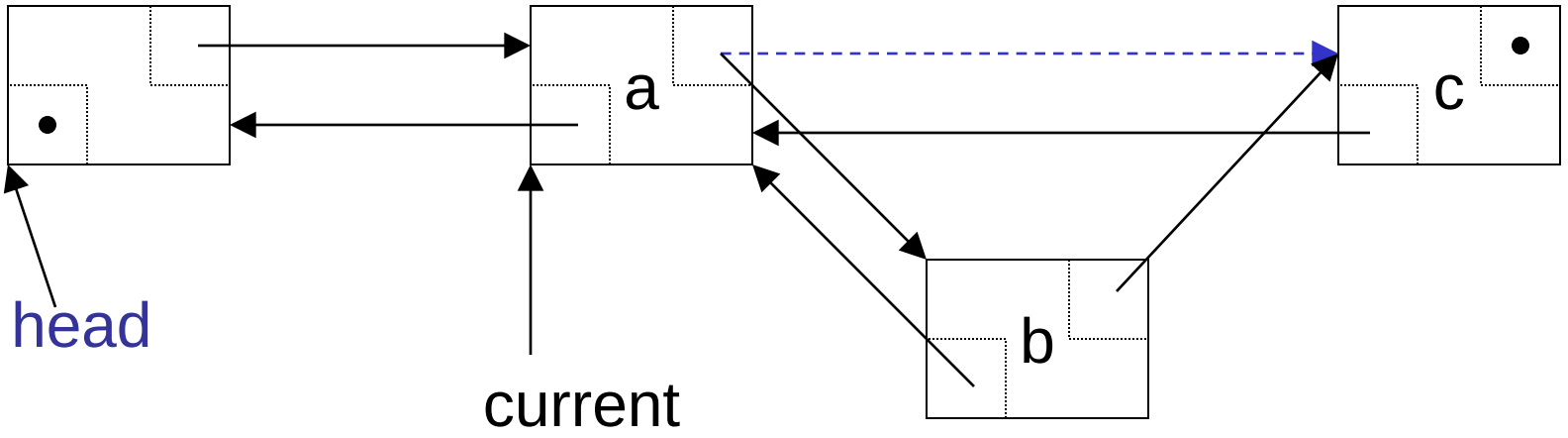

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List



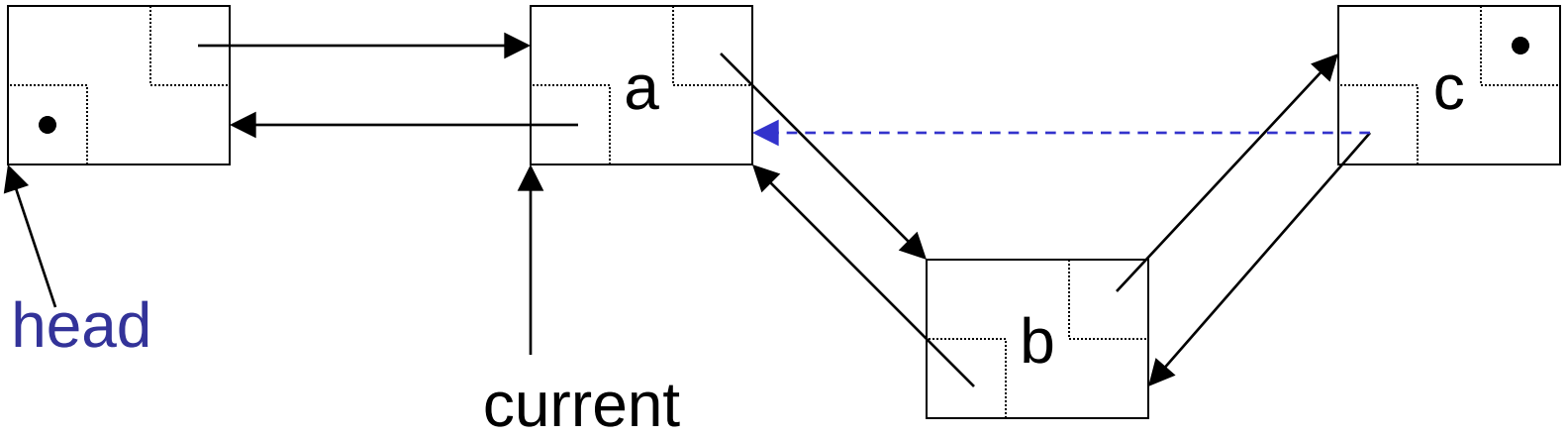

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List



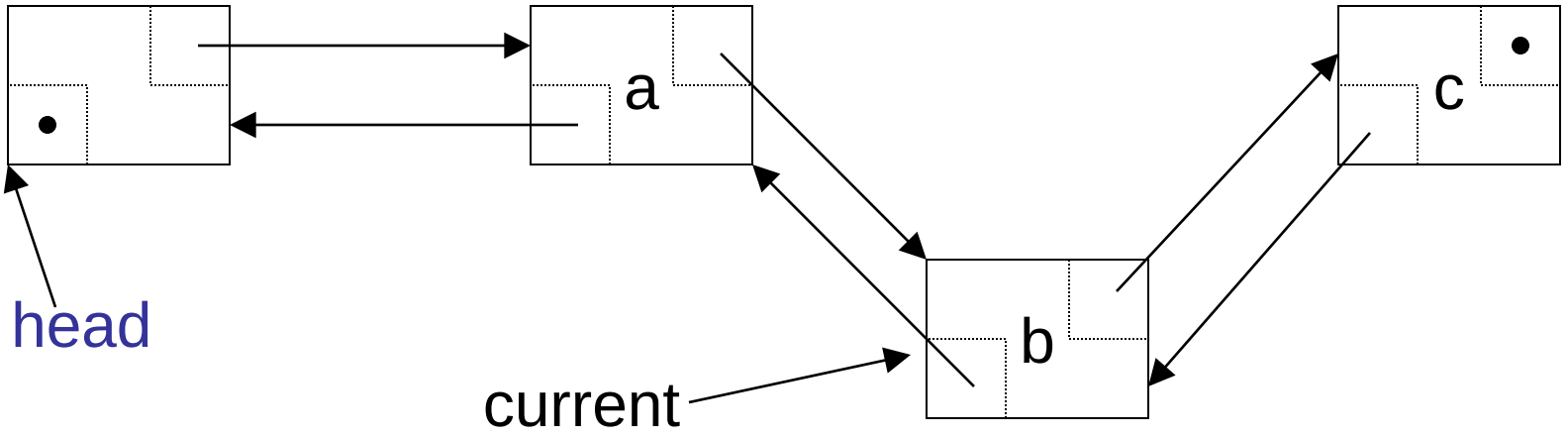

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List




```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List



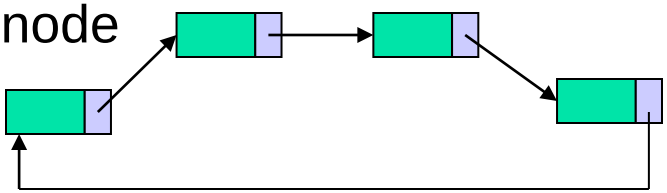

```
newNode = (DLLNode*) malloc ( size_of( DLLNode ) );
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

DLLs compared to SLLs

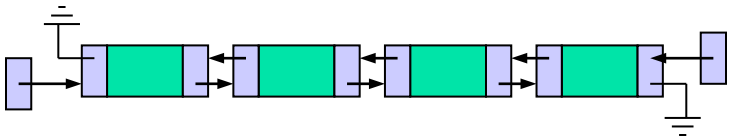
- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Linked Lists Types

- Types of linked lists:
  - Singly linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction



  - Circular, singly linked
    - Pointer in the last node points back to the first node



  - Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - Circular, doubly linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node