Linked Lists

Saikrishna Arcot
M. Hudachek-Buswell

August 18, 2018

Linked List

- A linked list is a data structure consisting of a sequence of nodes.

Linked List

- A linked list is a data structure consisting of a sequence of nodes.
- Each node in the linked list stores the data and a pointer to the next/previous node(s).

Linked List

- A linked list is a data structure consisting of a sequence of nodes.
- Each node in the linked list stores the data and a pointer to the next/previous node(s).
- Nodes are created/destroyed as necessary. When adding a new element, a new node is created, and linked into the linked list. When removing an element, the node containing the element is destroyed.

Linked List

- A head pointer (pointer to the first node in the list) is necessary, while a tail pointer is optional.

Linked List

- A head pointer (pointer to the first node in the list) is necessary, while a tail pointer is optional.
- Linked lists are better for manipulating data, while array lists are better for storing/accessing data.

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:
  - the element

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:
  - the element
  - pointer to the next node

Singly Linked List

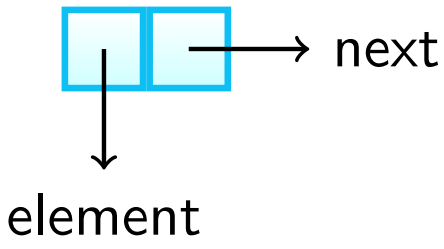
- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:
  - the element
  - pointer to the next node
- Example of a node

Singly Linked List

- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:
  - the element
  - pointer to the next node
- Example of a node

Singly Linked List

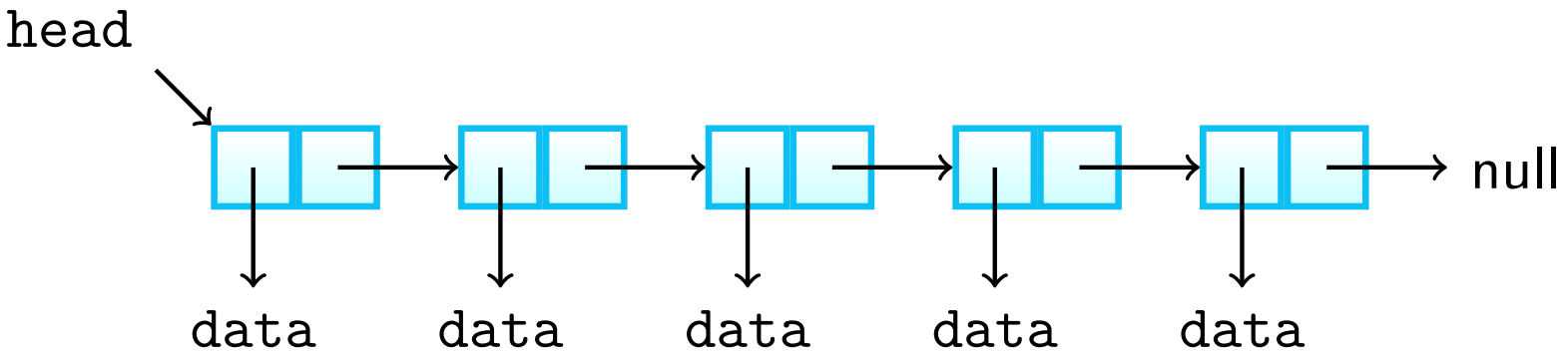
- A singly linked list is a version of a linked list where each node *only* has a pointer to the next node (i.e. it is not possible to go back to the previous node). The singly linked list starts from the head pointer.
- Each node stores:
  - the element
  - pointer to the next node
- Example of a node



Singly Linked List

In this course, we refrain from creating "dummy" nodes that lead to null references at the end of a singly linked list. We just have the next reference of the last node point to null.

Example of a singly linked list:



Inserting at the Head

- Allocate a new node with data and next reference.

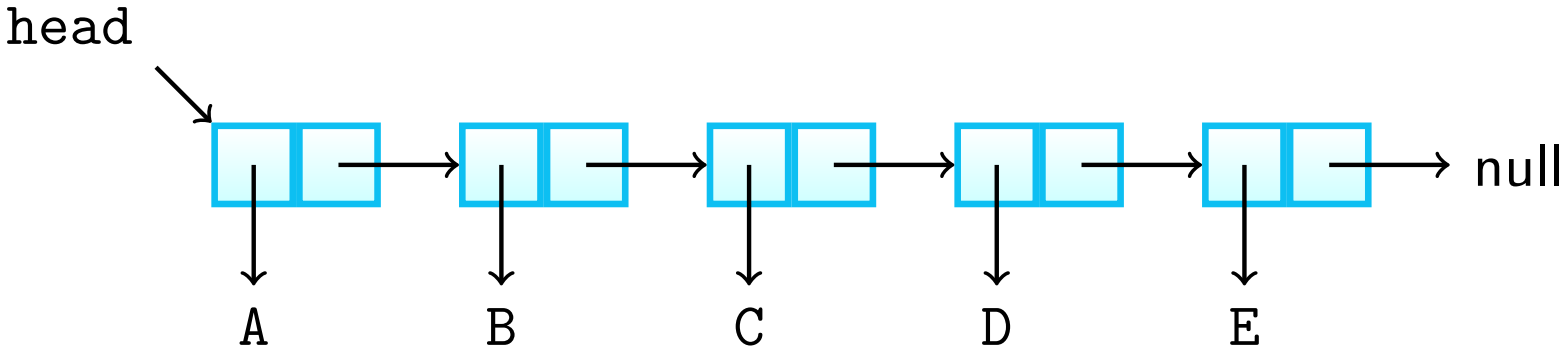
Inserting at the Head

- Allocate a new node with data and next reference.
- Have the new node's next reference point to old head.

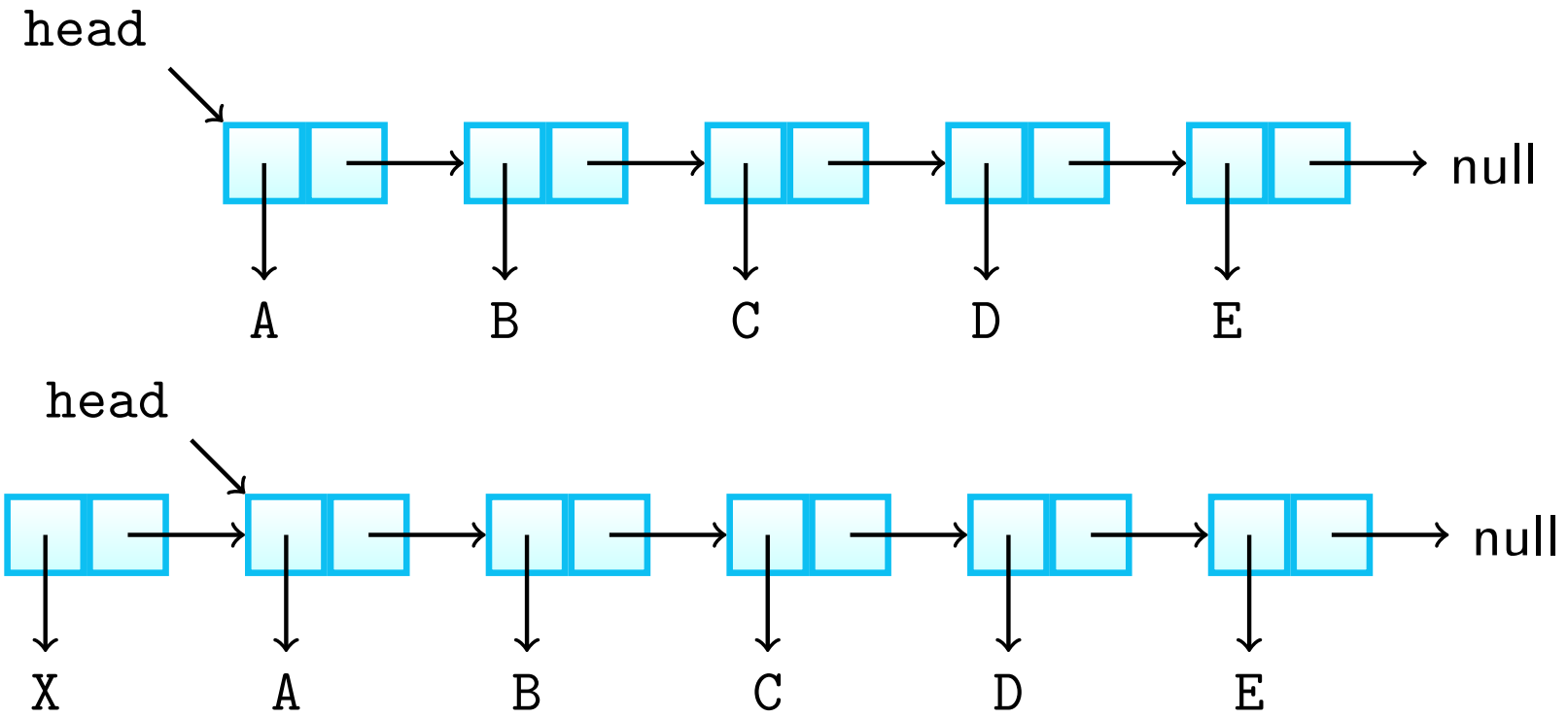
Inserting at the Head

- Allocate a new node with data and next reference.
- Have the new node's next reference point to old head.
- Update the head reference to point to new node.

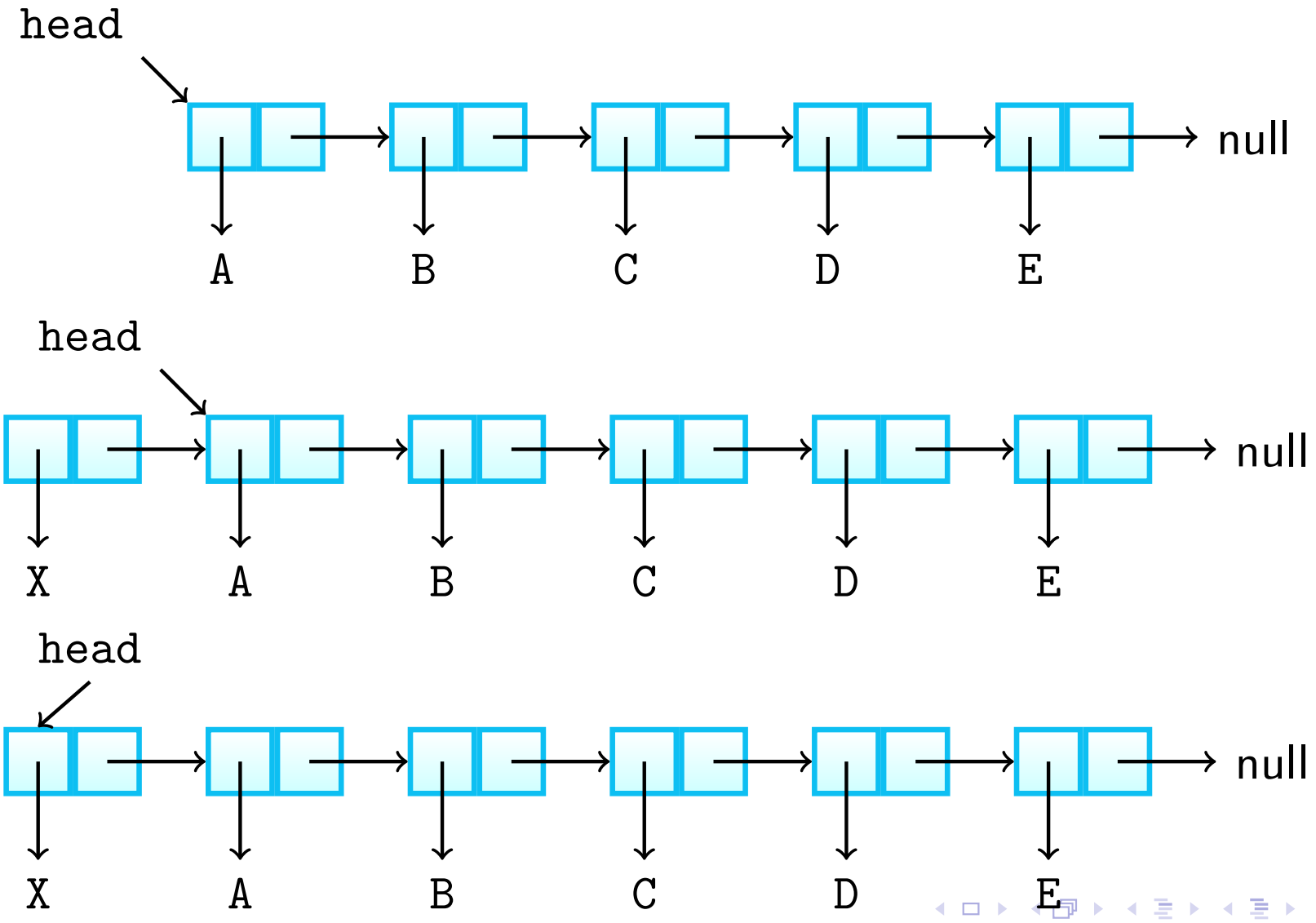
Inserting at the Head



Inserting at the Head



Inserting at the Head



Inserting at the Tail

- Allocate a new node with data and next reference.

Inserting at the Tail

- Allocate a new node with data and next reference.
- Have new node's next reference point to `null`.

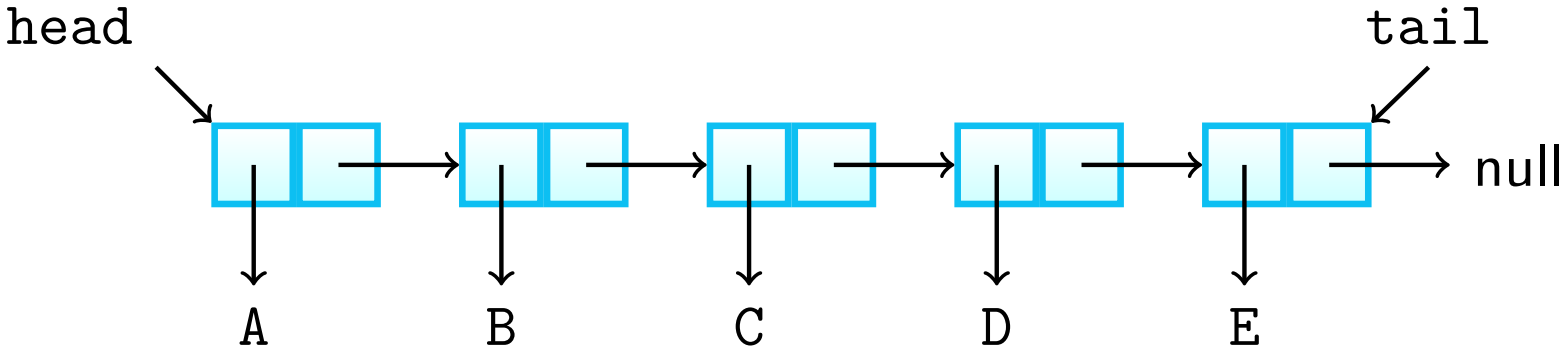
Inserting at the Tail

- Allocate a new node with data and next reference.
- Have new node's next reference point to `null`.
- Have the last node's next reference point to new node.

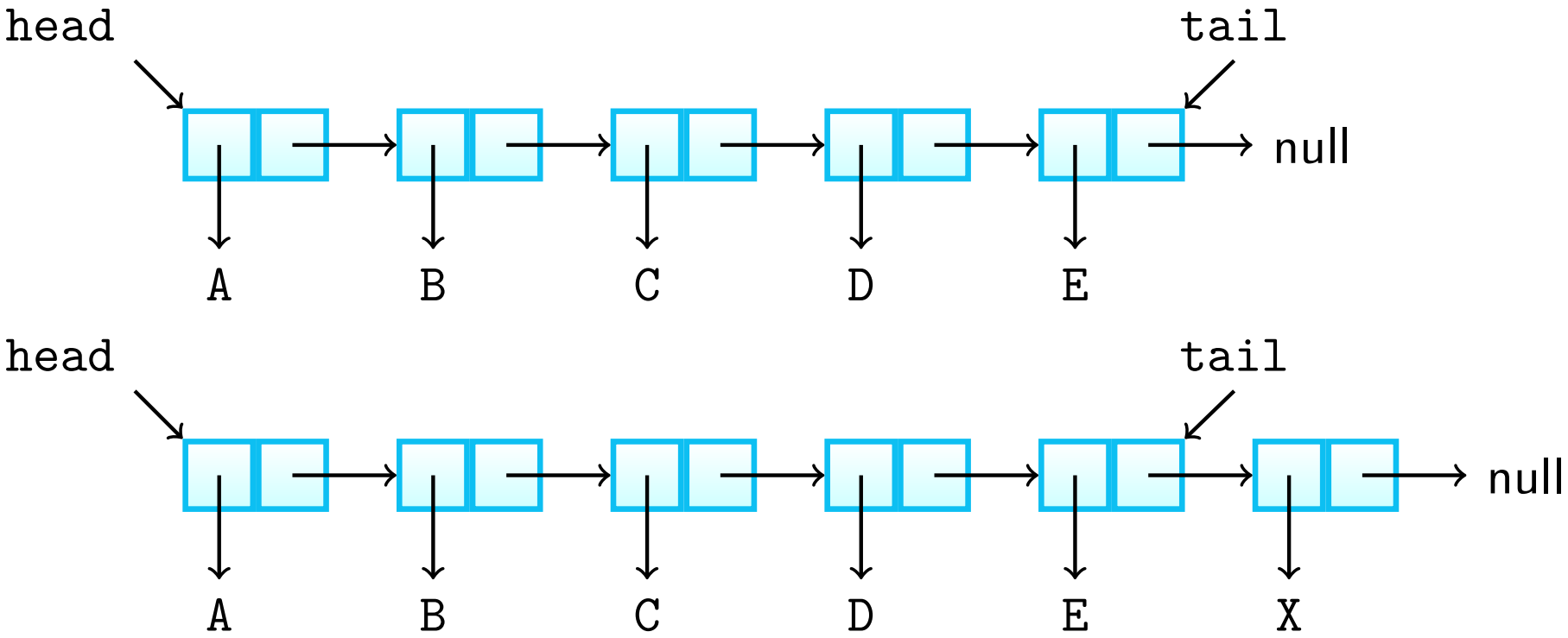
Inserting at the Tail

- Allocate a new node with data and next reference.
- Have new node's next reference point to `null`.
- Have the last node's next reference point to new node.
- Update tail (if there is a tail pointer) to point to new node.

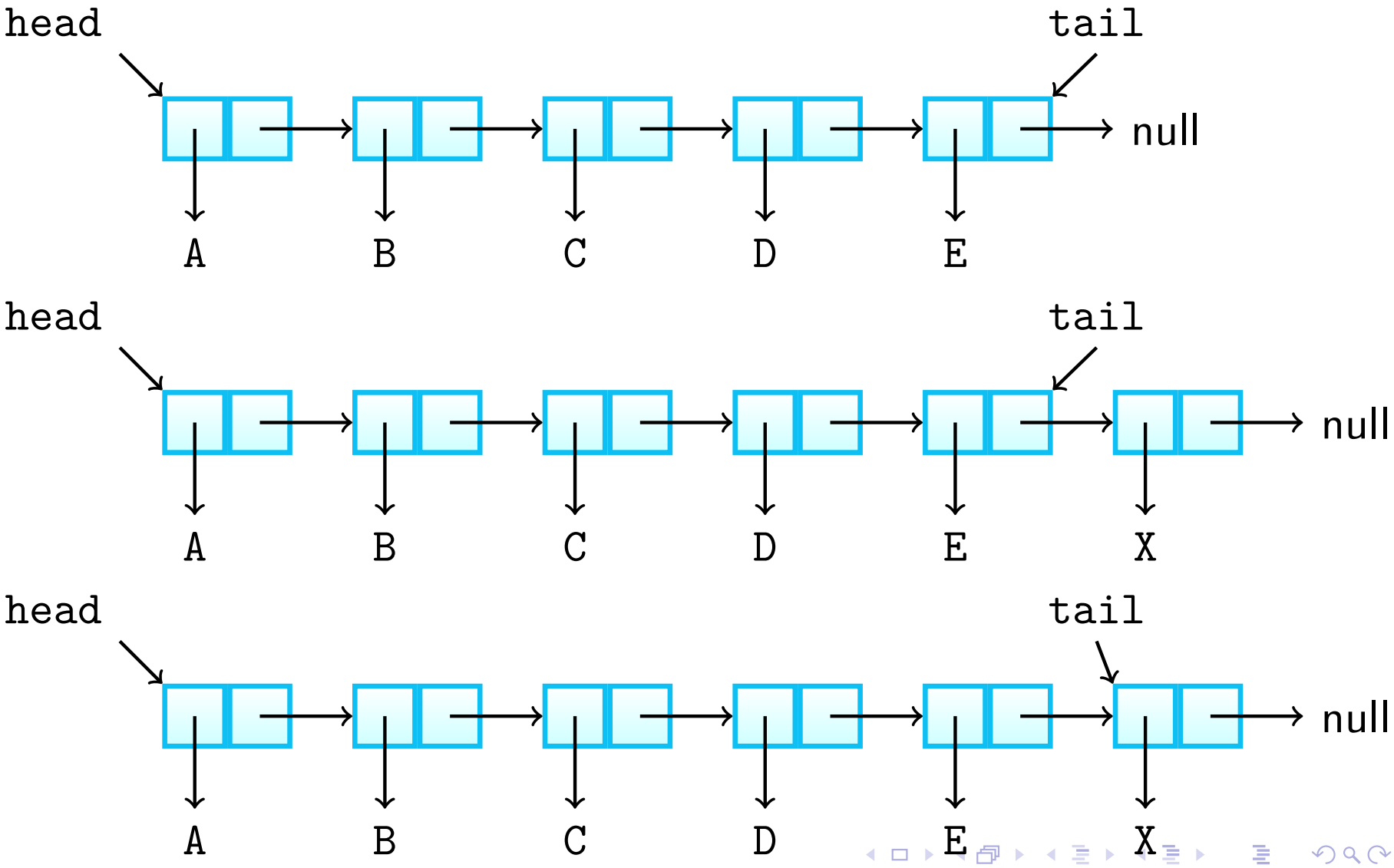
Inserting at the Tail



Inserting at the Tail



Inserting at the Tail



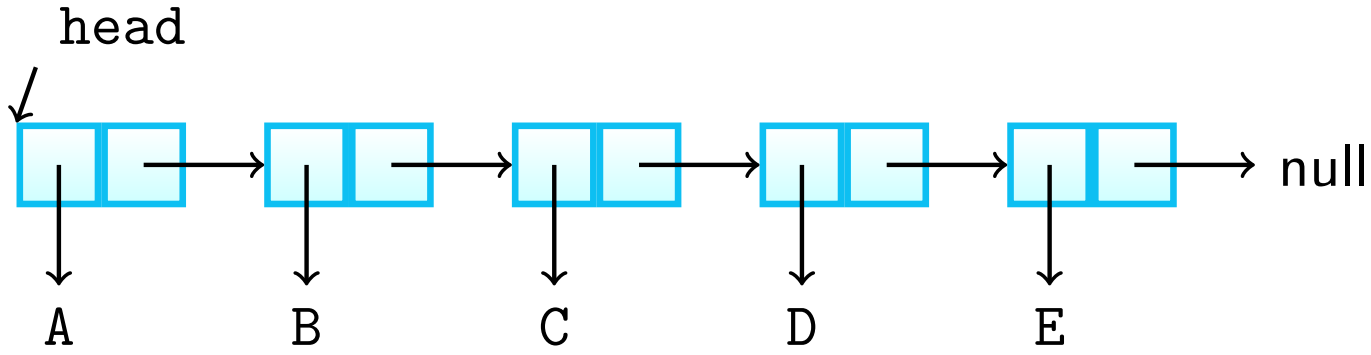
Removing at the Head

- Update head to point to the next node in the list.

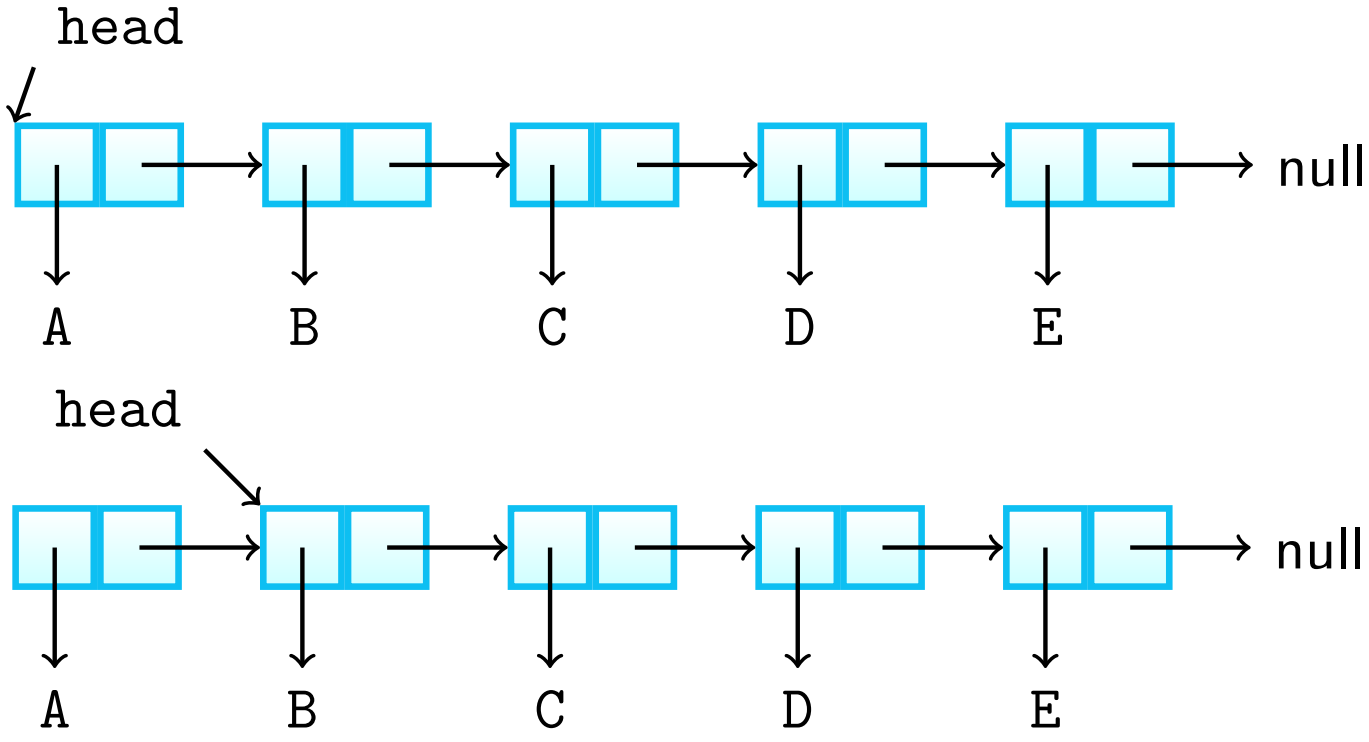
Removing at the Head

- Update head to point to the next node in the list.
- Allow garbage collector to reclaim the former first node.

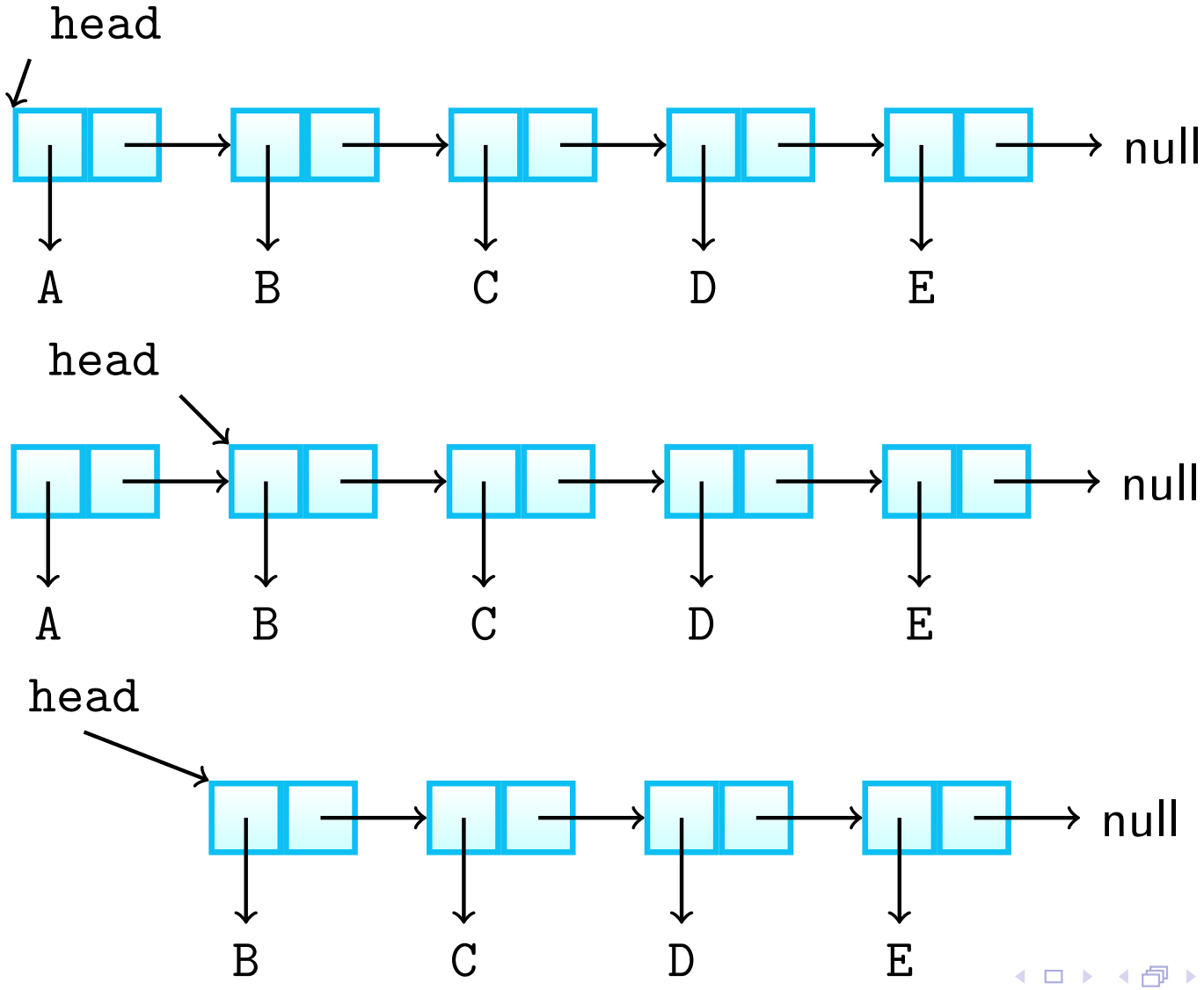
Removing at the Head



Removing at the Head



Removing at the Head



Removing at the Tail

- Removing at the tail of a singly linked list is NOT efficient! There is no constant-time way to update the tail to point to the previous node.

Removing at the Tail

- Removing at the tail of a singly linked list is NOT efficient! There is no constant-time way to update the tail to point to the previous node.
- Iterate to the node before the last node.

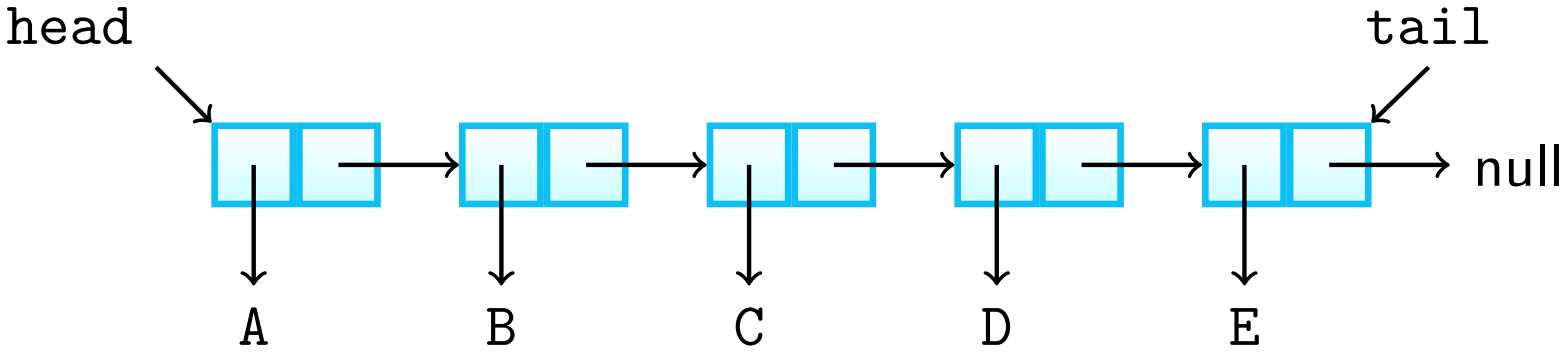
Removing at the Tail

- Removing at the tail of a singly linked list is NOT efficient! There is no constant-time way to update the tail to point to the previous node.
- Iterate to the node before the last node.
- Update tail (if there is a tail pointer).

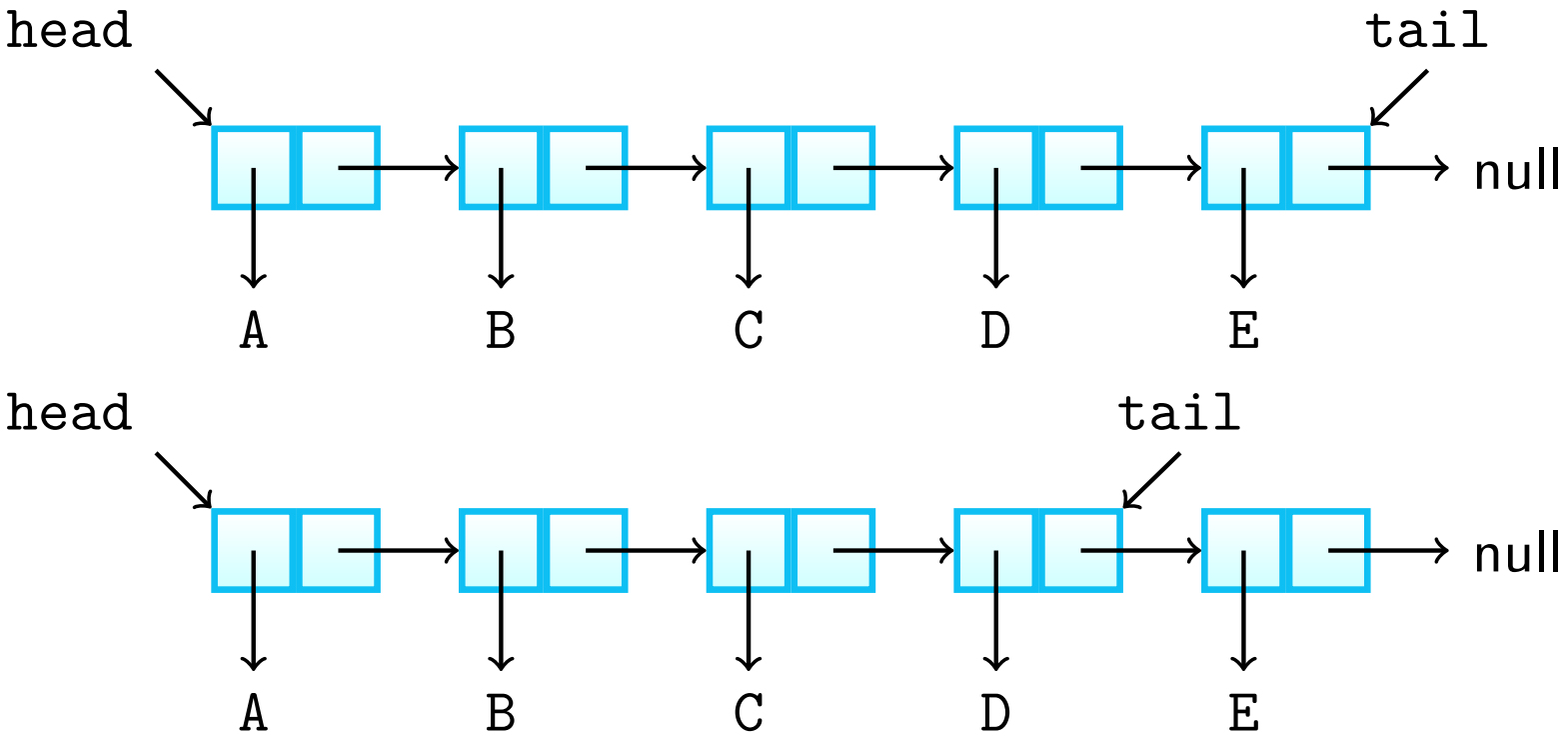
Removing at the Tail

- Removing at the tail of a singly linked list is NOT efficient! There is no constant-time way to update the tail to point to the previous node.
- Iterate to the node before the last node.
- Update tail (if there is a tail pointer).
- Set the current node's next reference to `null`

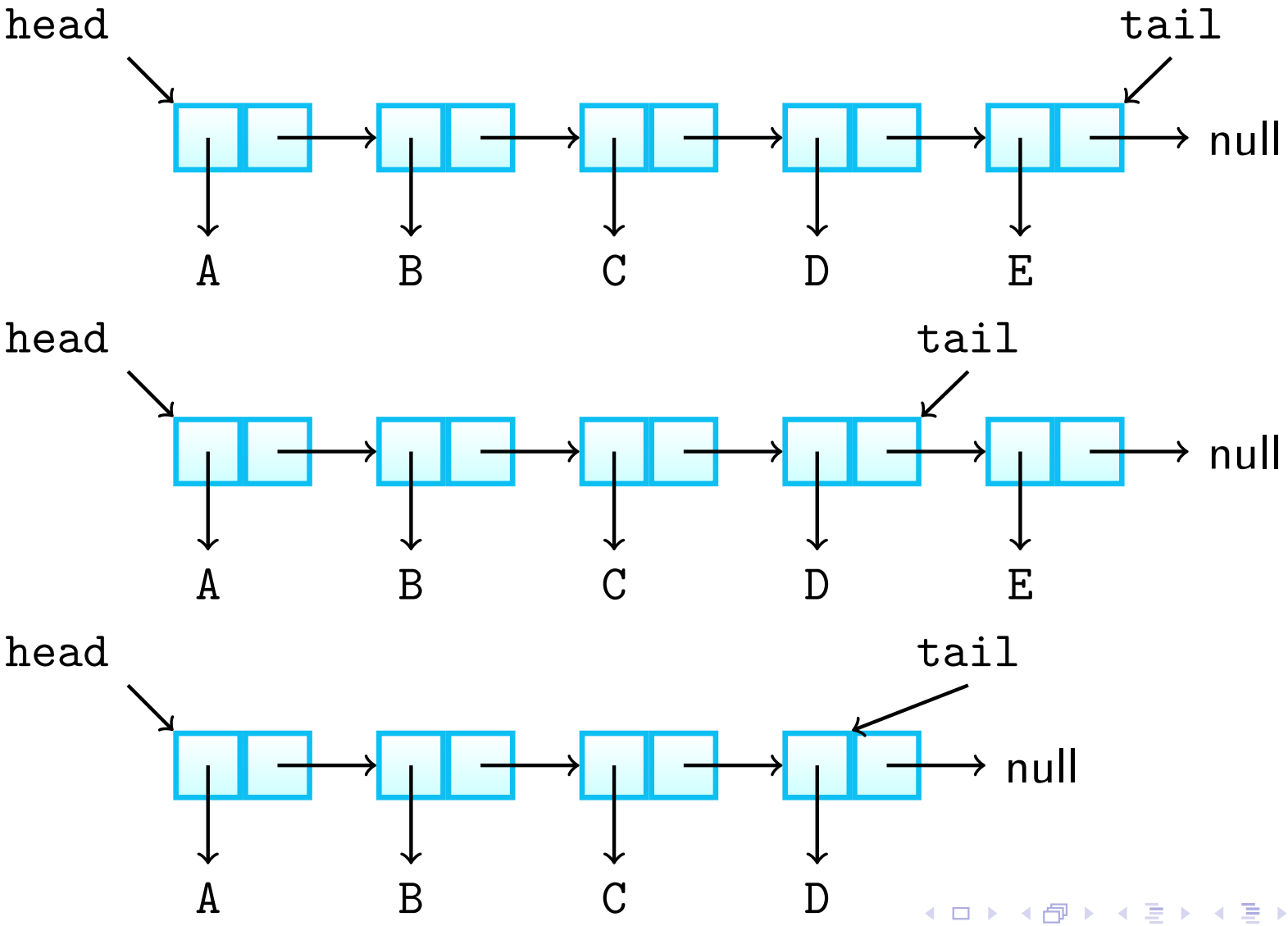
Removing at the Tail



Removing at the Tail



Removing at the Tail



Inserting/Removing at Arbitrary Locations

- In addition to inserting/removing at the head/tail, you can add items at any location in the linked list.

Inserting/Removing at Arbitrary Locations

- In addition to inserting/removing at the head/tail, you can add items at any location in the linked list.
- Note that you need to update the next pointer (and the previous pointer, if it is present) so that the new node is linked in correctly, and that the list can be traversed correctly.

Inserting/Removing at Arbitrary Locations

- In addition to inserting/removing at the head/tail, you can add items at any location in the linked list.
- Note that you need to update the next pointer (and the previous pointer, if it is present) so that the new node is linked in correctly, and that the list can be traversed correctly.
- Examples of these are shown for doubly-linked lists.

Pseudocode For Inserting/Removing

procedure AddFirst(e)
    node $\leftarrow$ Create a new `Node` object, and have its data be *e*, and the next node be *head*
    head $\leftarrow$ *node*
    if list is empty **then**
        tail $\leftarrow$ *node*
    end if
    size $\leftarrow$ *size* $+ 1$
end procedure

Pseudocode For Inserting/Removing

procedure AddLast(e)
    node $\leftarrow$ Create a new `Node` object, and have its data be *e*
    if list is empty **then**
        head $\leftarrow$ *node*
    else
        tail.*next* $\leftarrow$ *node*
    end if
    tail $\leftarrow$ *node*
    size $\leftarrow$ *size* $+ 1$
end procedure

Pseudocode For Inserting/Removing

procedure REMOVEFIRST(e)
    node $\leftarrow$ *head*
    head $\leftarrow$ *head*.*next*
    size $\leftarrow$ *size* $-$ 1
    return *node*
end procedure

Doubly Linked List

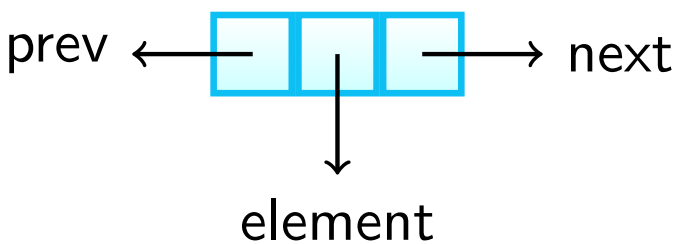
- A doubly linked list is a version of a linked list where each node has a pointer to the next node *and* a pointer to the previous node.
- Doubly linked lists can be traversed forward and backward.

Doubly Linked List

- A doubly linked list is a version of a linked list where each node has a pointer to the next node *and* a pointer to the previous node.
- Doubly linked lists can be traversed forward and backward.
- Nodes store:
  - element
  - link to the previous node
  - link to the next node

Doubly Linked List

- A doubly linked list is a version of a linked list where each node has a pointer to the next node *and* a pointer to the previous node.
- Doubly linked lists can be traversed forward and backward.
- Nodes store:
  - element
  - link to the previous node
  - link to the next node



Generic Code For Doubly Linked List Node

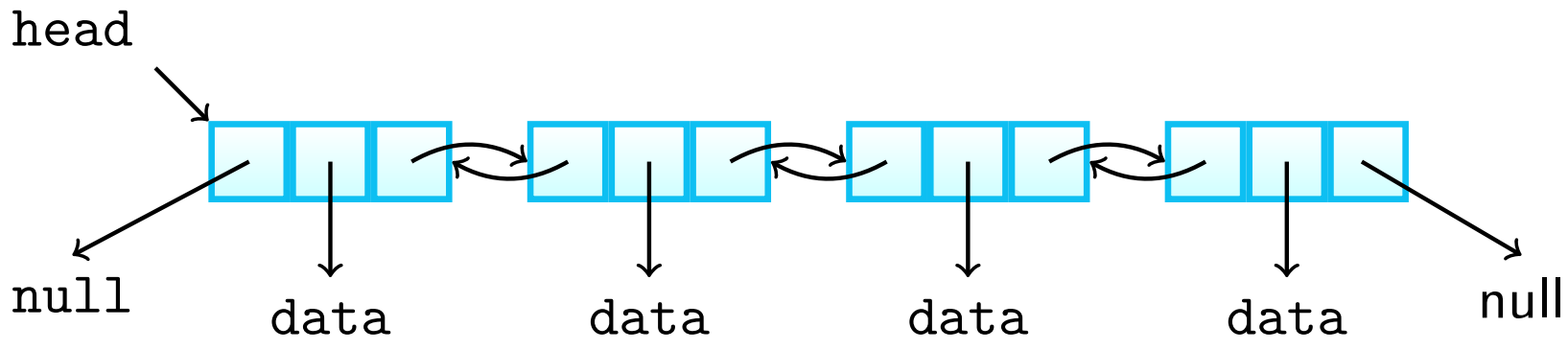
```
public class DoublyLinkedList<Type> {
    private class Node<Type> {
        private Type data;
        private Node<Type> next;
        private Node<Type> prev;

        private Node(Type data, Node<Type> next,
          Node<Type> prev) {
            this.data = data;
            this.next = next;
            this.prev = prev;
        }
    }
}
```

Generic Code For Doubly Linked List Node

```
// Node constructor chaining
private Node(Type data) {
    this(data, null, null)
// this.data = data
// this.next = null
// this.prev = null
}
```

Doubly Linked List

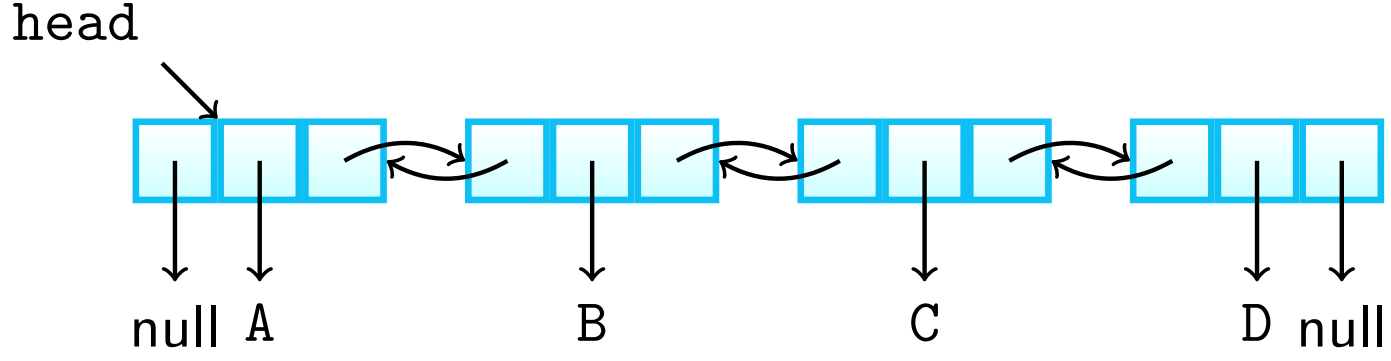
Example of a doubly linked list:



Notice that `prev` reference of the first node points to `null`, as does the `next` reference of the last node

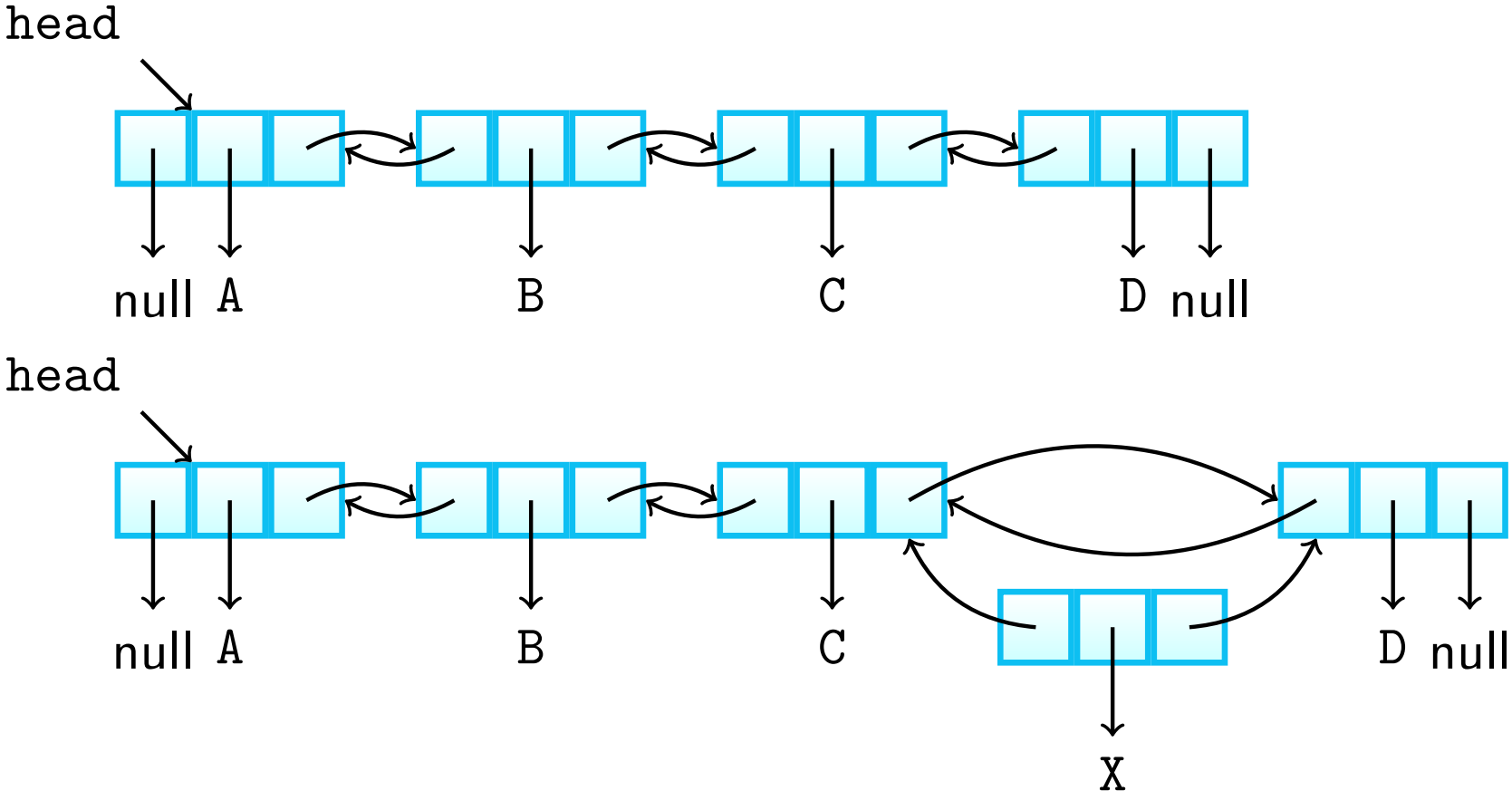
Insertion

Create new node `X`. Insert a new node, `X`, between node `C` and node `D`. Set new node's `prev` reference to `C` and `next` reference to `D`.

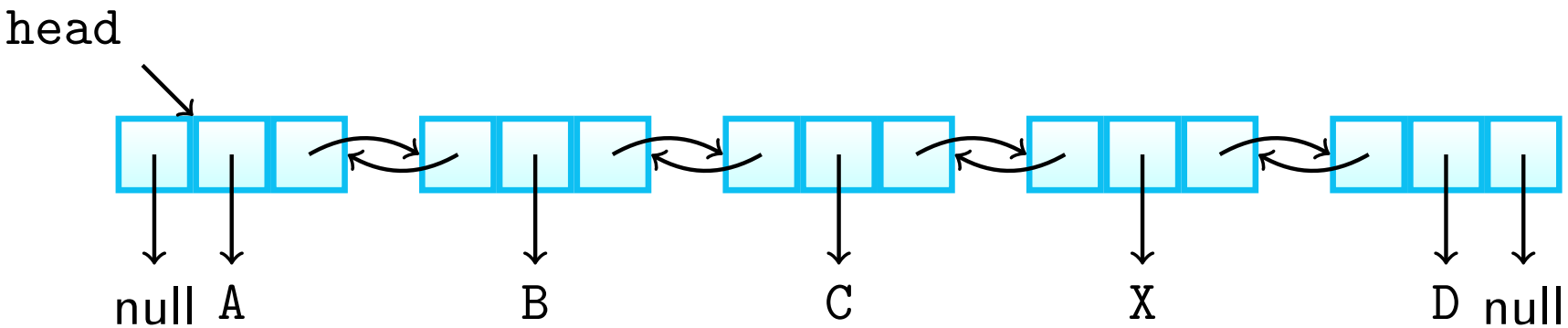


Insertion

Create new node `X`. Insert a new node, `X`, between node `C` and node `D`. Set new node's `prev` reference to `C` and `next` reference to `D`.



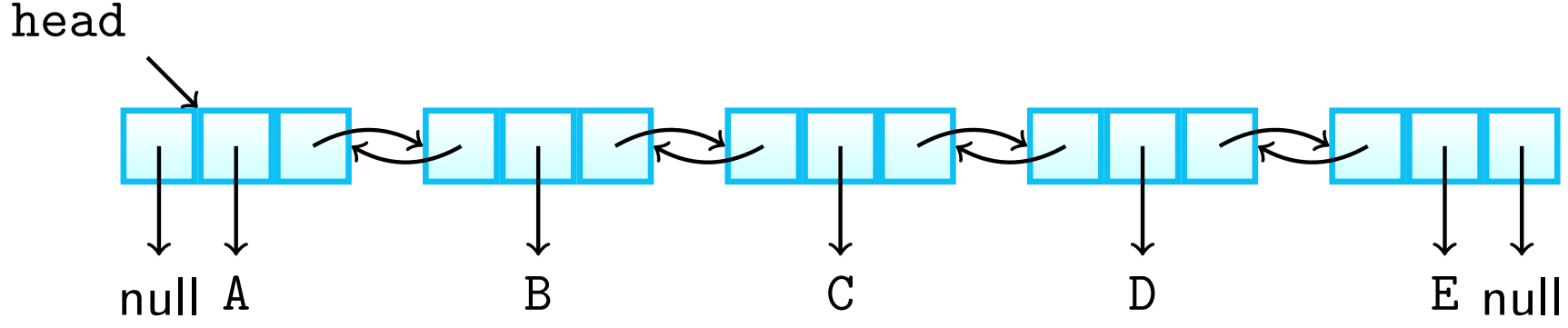
Insertion



Once node `X` is connected, set Node `C`'s `next` reference to node `X`, and Node `D`'s `prev` reference to node `X`.

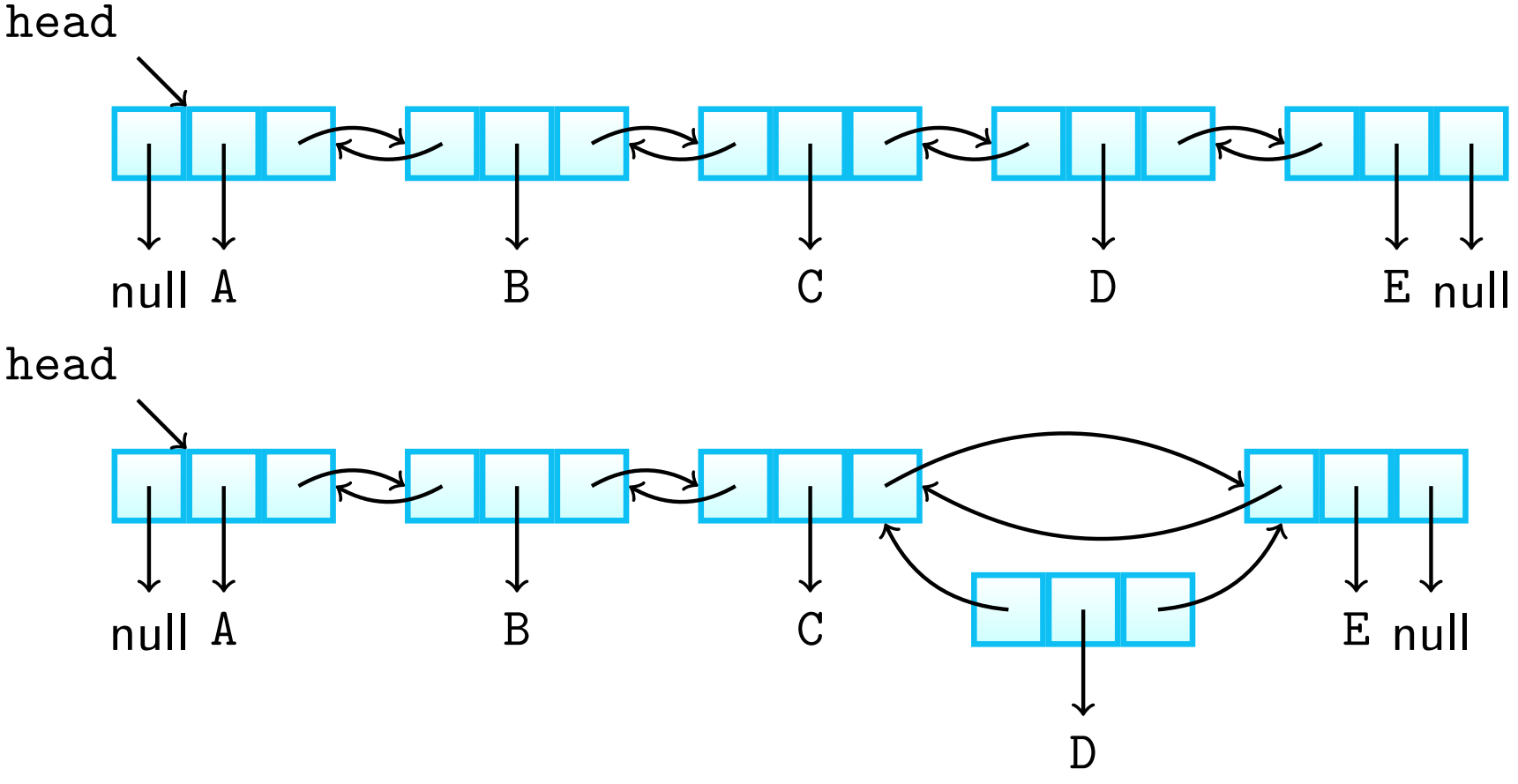
Deletion

Remove node D, between nodes C and E. The order in which you redirect references is important so you do not lose your linked list.



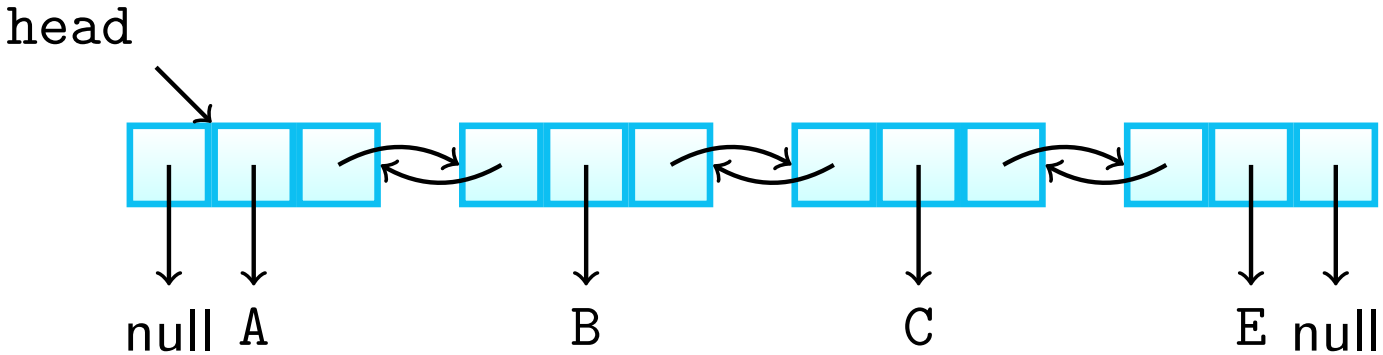
Deletion

Remove node D, between nodes C and E. The order in which you redirect references is important so you do not lose your linked list.



Deletion

Set node `C`'s `next` reference to node `E`, and node `E`'s `prev` reference to `C`. Node `D` will be garbage collected.



Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.

Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.
- Accessing or searching any linked list is a cost of O(n), linear time

Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.
- Accessing or searching any linked list is a cost of O(n), linear time
- Adding or removing *from the front* of any linked list with a *head pointer* is a cost of O(1), constant time

Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.
- Accessing or searching any linked list is a cost of O(n), linear time
- Adding or removing *from the front* of any linked list with a *head pointer* is a cost of O(1), constant time
- Linked lists are useful when you have to manipulate data

Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.
- Accessing or searching any linked list is a cost of O(n), linear time
- Adding or removing *from the front* of any linked list with a *head pointer* is a cost of O(1), constant time
- Linked lists are useful when you have to manipulate data
- Singly linked lists are used in RFID scanning, app wizards or levels in a video game

Linked List Summary and Complexity

- Linked Lists are most definitely dynamic, and can store primitives or objects.
- Accessing or searching any linked list is a cost of O(n), linear time
- Adding or removing *from the front* of any linked list with a *head pointer* is a cost of O(1), constant time
- Linked lists are useful when you have to manipulate data
- Singly linked lists are used in RFID scanning, app wizards or levels in a video game
- Doubly linked lists are used in browser history, scroll bars or forward/back buttons

Circular Linked Lists

- Circular Linked List is a more complicated Linked list where the last node points to the first node, (and the first node could point to the last node).

Circular Linked Lists

- Circular Linked List is a more complicated Linked list where the last node points to the first node, (and the first node could point to the last node).
- Circular linked lists can be either singly or doubly linked.

Circular Linked Lists

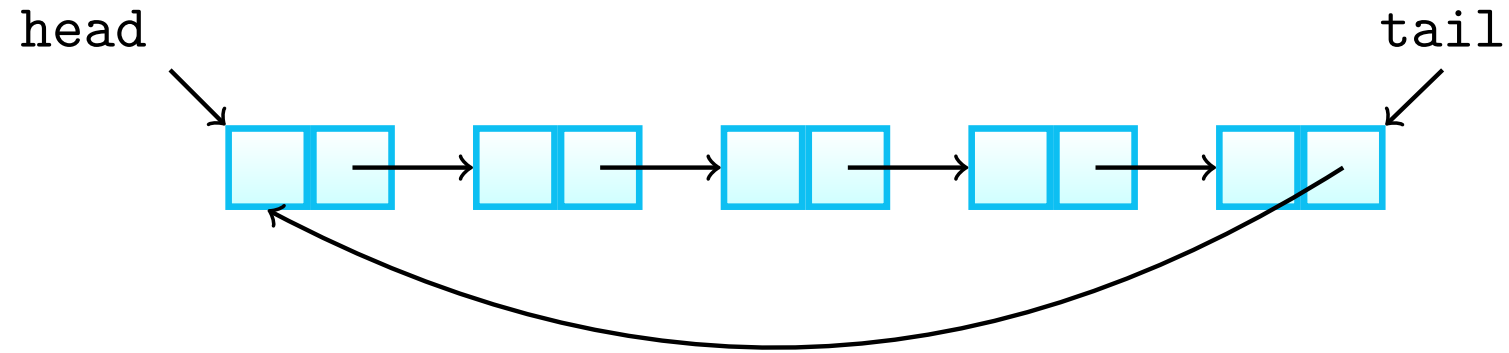
- Circular Linked List is a more complicated Linked list where the last node points to the first node, (and the first node could point to the last node).
- Circular linked lists can be either singly or doubly linked.
- Depending on single or double links, adding or removing is the same as for the type of link

Circular Linked Lists

- Circular Linked List is a more complicated Linked list where the last node points to the first node, (and the first node could point to the last node).
- Circular linked lists can be either singly or doubly linked.
- Depending on single or double links, adding or removing is the same as for the type of link
- Applications that use circular linked lists are gifs, music playlists; round-robin scheduling algorithms for operating systems

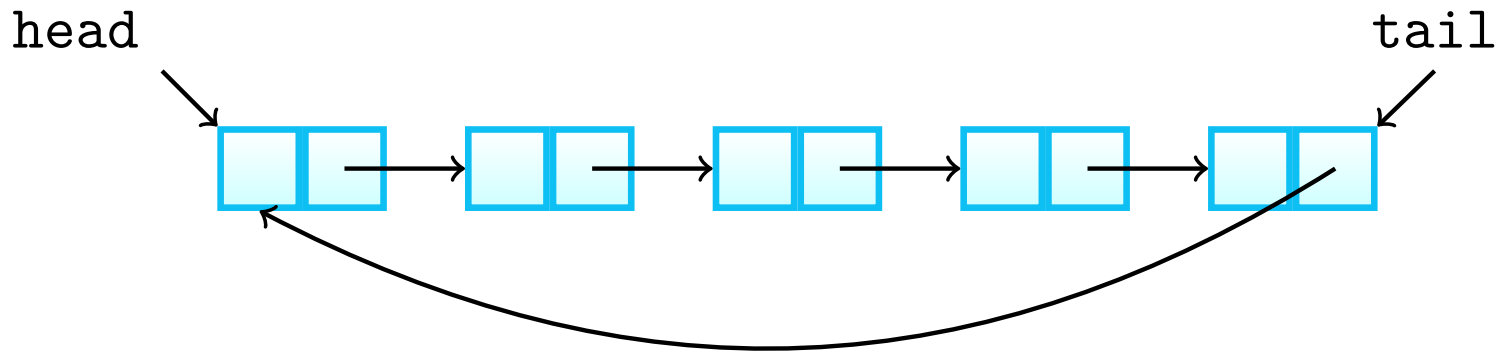
Circular Linked Lists

Example of Singly Circular Linked List



Circular Linked Lists

Example of Singly Circular Linked List



Example of Doubly Circular Linked List