Revised: 6/6/2020

Linked Lists II

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Doubly Linked Lists
    - Appending
    - Traversal
    - Prepending
    - Insertion
    - Retrieval
    - Removal

Doubly Linked Lists

- A sample struct to be used as the nodes in a Doubly Linked List:

```
struct Node {
   int data;                //The value stored in the node
   Node *next;              //Pointer to the next node in the list
   Node *previous;          //Pointer to the previous node in the list
};
```

- In addition to the “data” field, this Node struct could contain other values, objects, or functions- public or private.

Doubly Linked Lists

- The class itself for the doubly linked list will need to maintain pointers to the head and tail of the list, just as the singly linked list did.

```
class DLinkedList {
    private:
        Node *head;             //Pointer to the head of the list
        Node *tail;             //Pointer to the end of the list

    public:
        //Constructor. Sets the head and tail to NULL
        DLinkedList() {
            head = NULL;
            tail = NULL;
        }
}
```

Doubly Linked List Complexities

- The complexities for things like insertion, removal, traversal, etc. are the same as the singly linked list.

- The advantage of the doubly linked list is not in complexity, but that it can iterate forward **and backward**.

Doubly Linked Lists (Appending)

- New nodes are typically added to the back of the list.

1. Create the new node to be added. Make sure its next pointer is null since it will be the new tail.
2. Check if head is null. If so, the list is empty; This new node is now the list’s head and tail (the only node in the list). Set the node’s previous pointer to null.
3. Otherwise, set the current tail’s next pointer to point to the new node. Set the new node’s previous pointer to the old tail. Set the list’s tail pointer to the new node.

Doubly Linked Lists (Appending)

```
void push_back(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = NULL;
    if(head == NULL) {
        temp->previous = NULL;
        head = temp;
        tail = temp;
        return;
    }
    else {
        tail->next = temp;
        temp->previous = tail;
        tail = temp;
    }
}
```

Doubly Linked Lists (Prepending)

- New nodes can also be added to the front of the list.

1. Create the new node to be added. Make sure it’s next pointer points to the current head and it’s previous pointer is set to null.
2. Check to see if there is an existing head. If so, set the current head node’s previous pointer to point to the new node.
3. Set the list’s head pointer to the new node.

Doubly Linked Lists (Prepending)

```
void push_front(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = head;
    temp->previous = NULL;
    if(head != NULL) {
        head->previous = temp;
    }
    head = temp;
    if(tail == NULL) {
        tail = temp;
    }
}
```

Doubly Linked Lists (Forward Traversal)

- No different from a singly linked list.

```
void printListData() {
    Node *tempPtr;
    tempPtr = head;
    while(tempPtr != NULL) {
        cout << tempPtr->data << " ";
        tempPtr = tempPtr->next;
    }
    cout << endl;
}
```

Doubly Linked Lists (Backward Traversal)

- Similar to forward traversal, but we start with the tail and use the previous pointer of each node until reaching the head (where the previous pointer will be null.

```
void printListDataReverse() {
    Node *tempPtr;
    tempPtr = tail;
    while(tempPtr != NULL) {
        cout << tempPtr->data << " ";
        tempPtr = tempPtr->previous;
    }
    cout << endl;
}
```

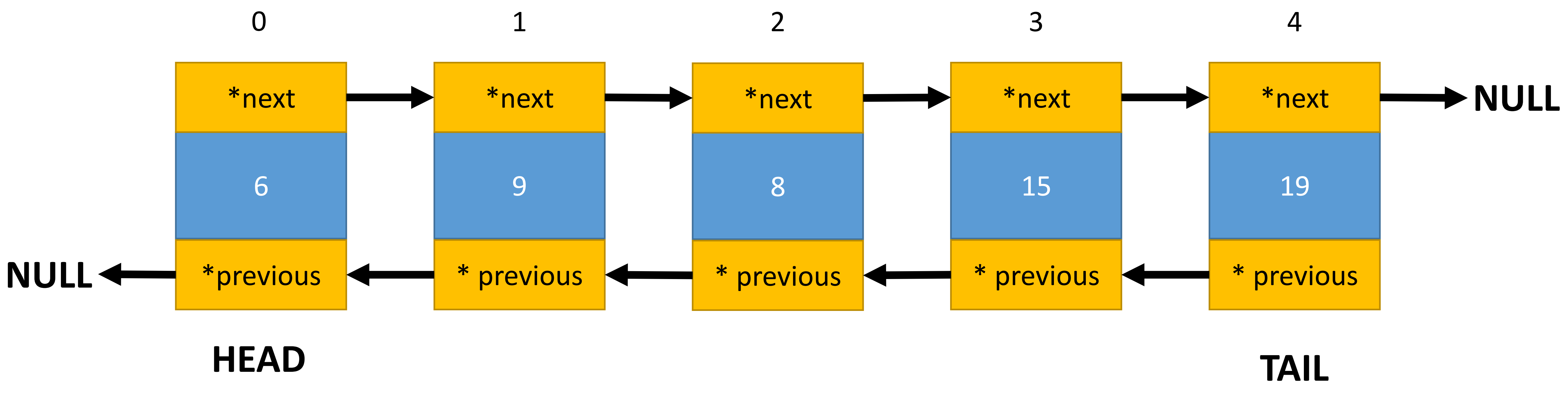
Doubly Linked Lists (Insertion)

- The process is similar to insertion in a singly linked list, but we must also consider the previous pointer.

1. Iterate to the node one place before the position where the insertion will take place
2. Check to see if it is null (meaning we reached the end of the list/tried to go beyond the tail)
3. If it’s not, create the new node.
4. Set the new node’s next pointer to the current node’s next pointer.
5. Set the new node’s previous pointer to the current node.
6. Set the previous pointer of the node after the current node to the new node.
7. Set the current node’s next pointer to the new node.

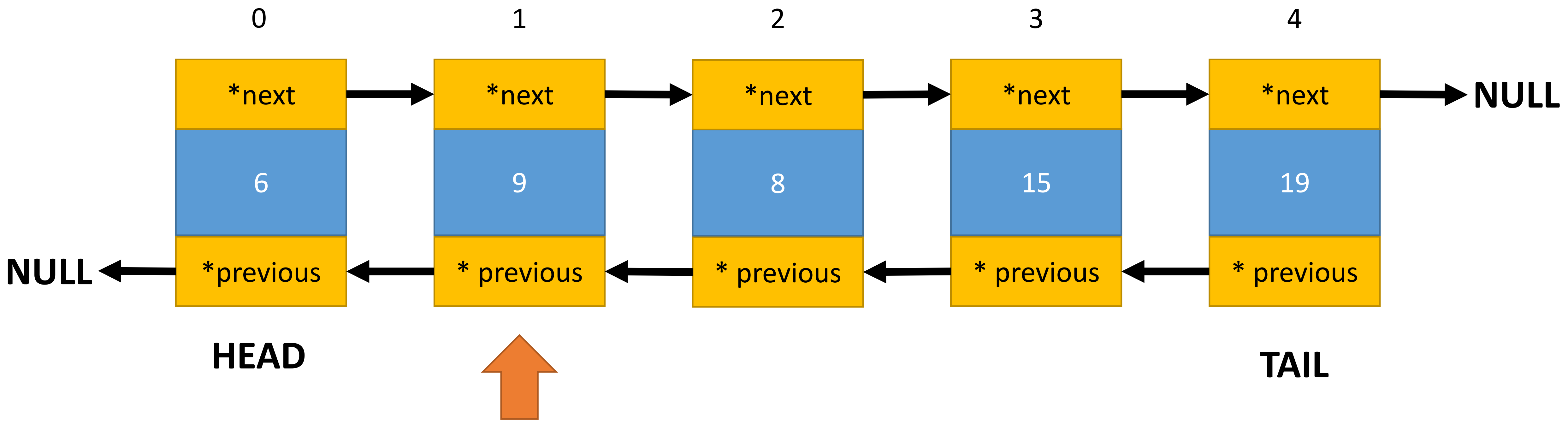
Doubly Linked Lists (Insertion)

- Inserting a node (data = 77) at position 2



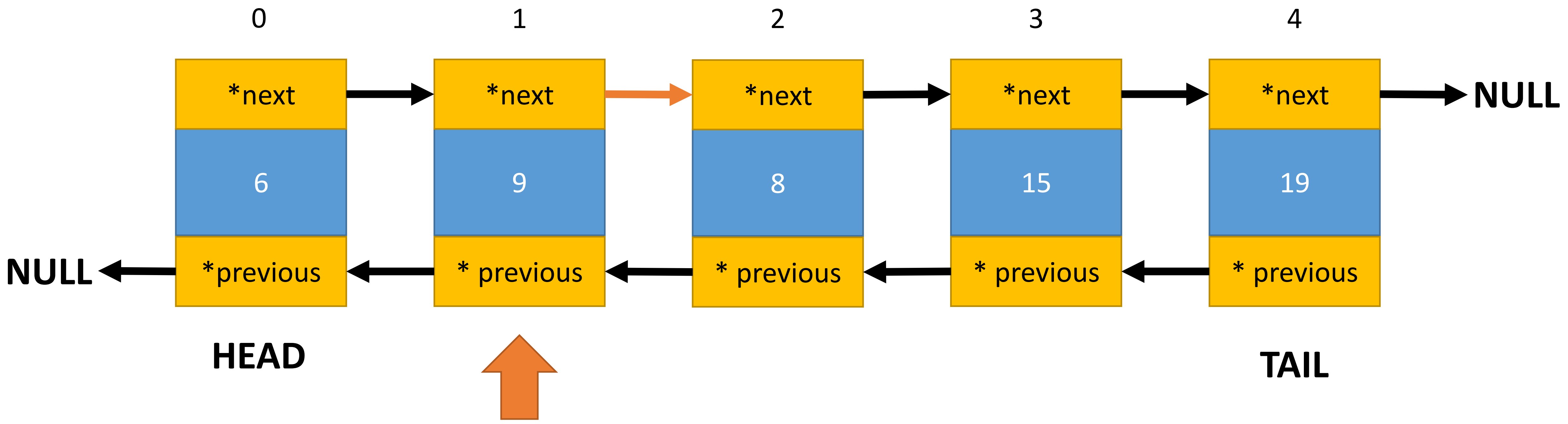
Doubly Linked Lists (Insertion)

- Iterate to position 1 (one place before the insertion point)



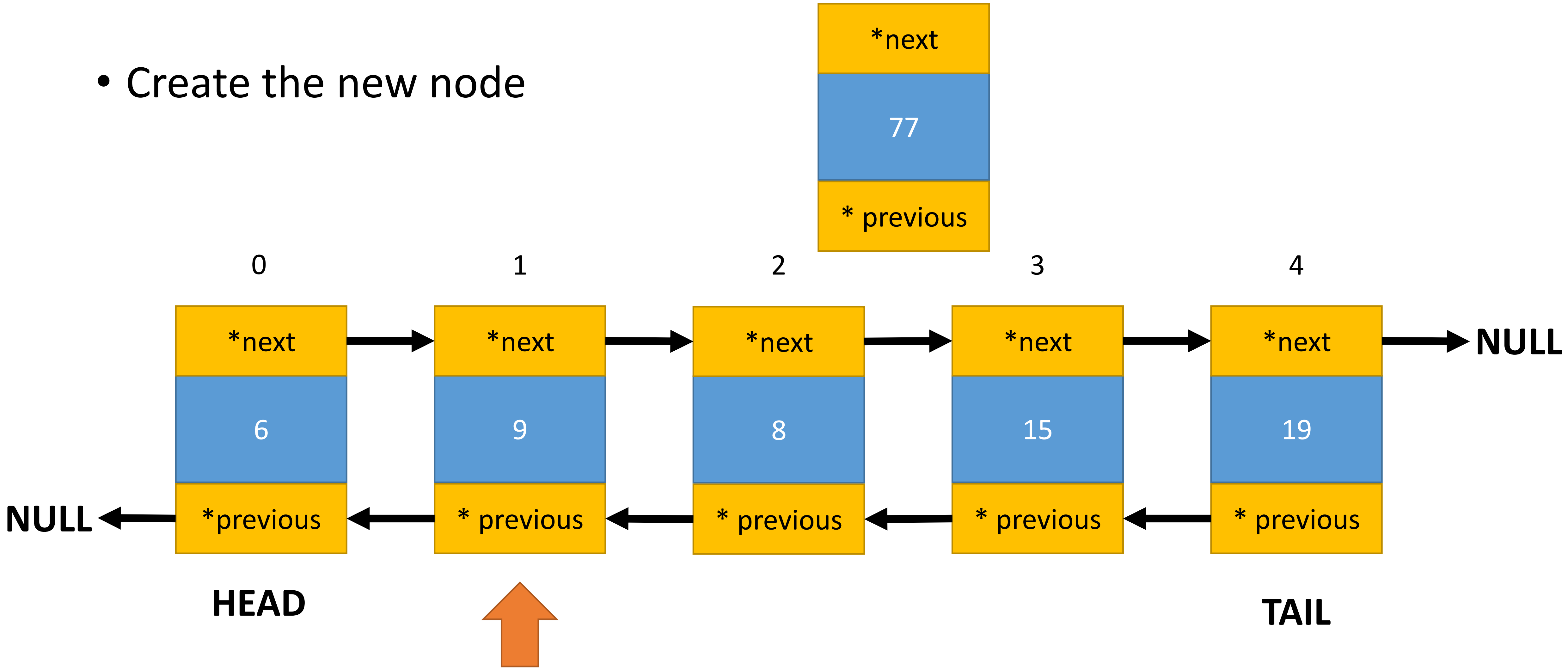
Doubly Linked Lists (Insertion)

- Check that it’s next pointer is not null



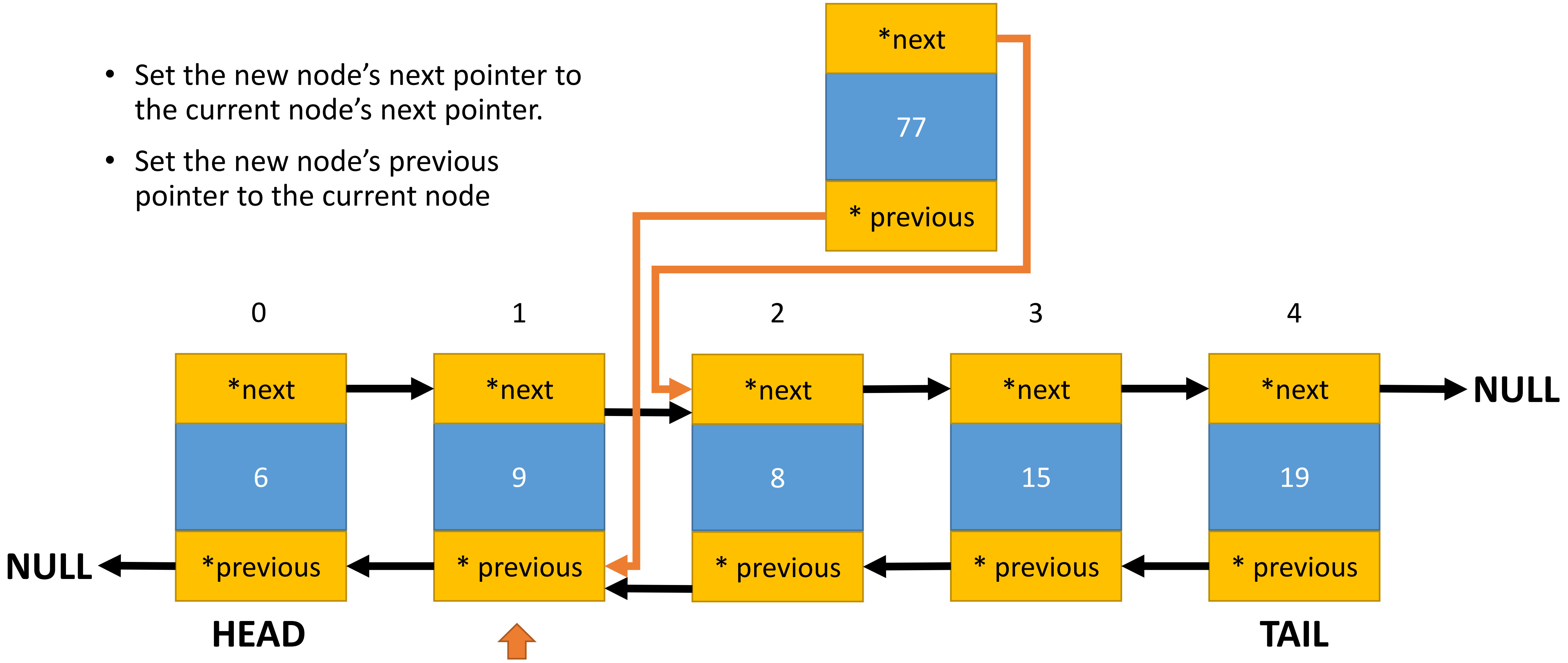
Doubly Linked Lists (Insertion)

- Create the new node



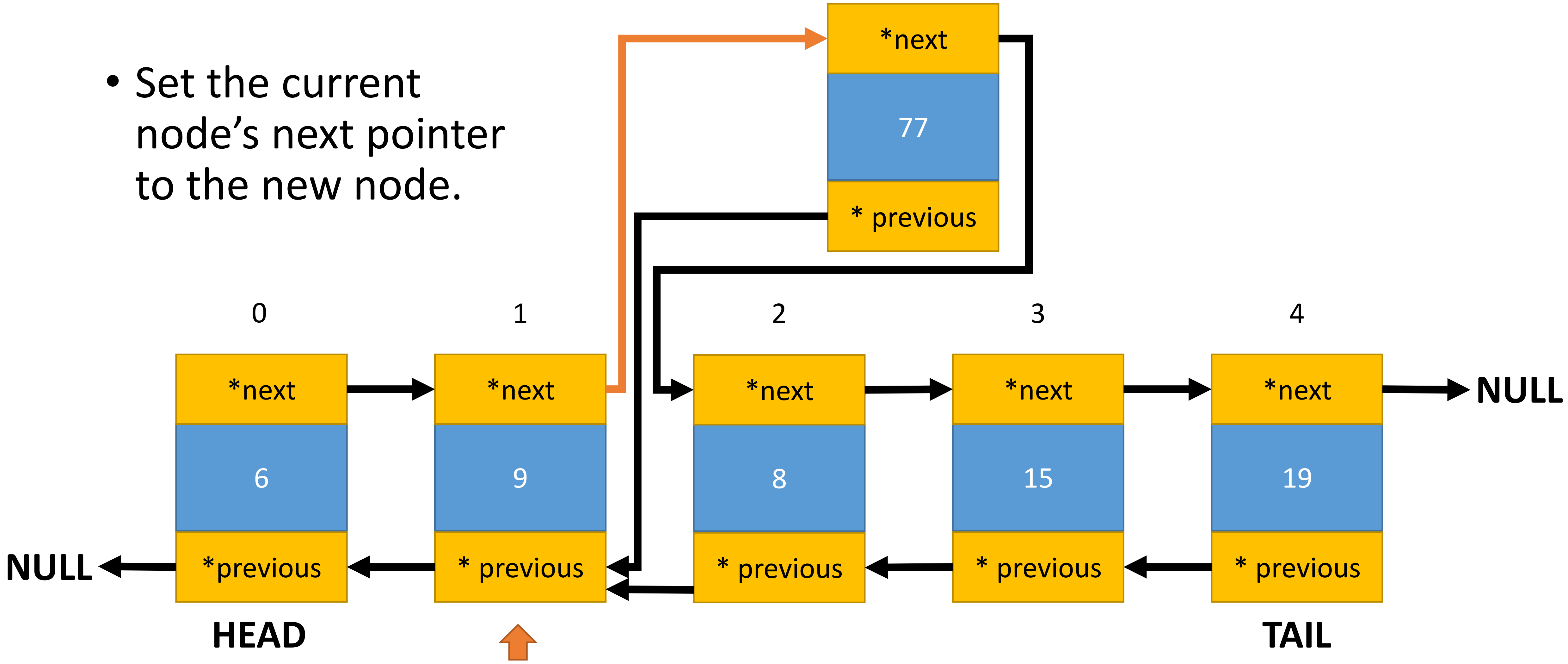
Doubly Linked Lists (Insertion)

- Set the new node's next pointer to the current node's next pointer.
- Set the new node's previous pointer to the current node



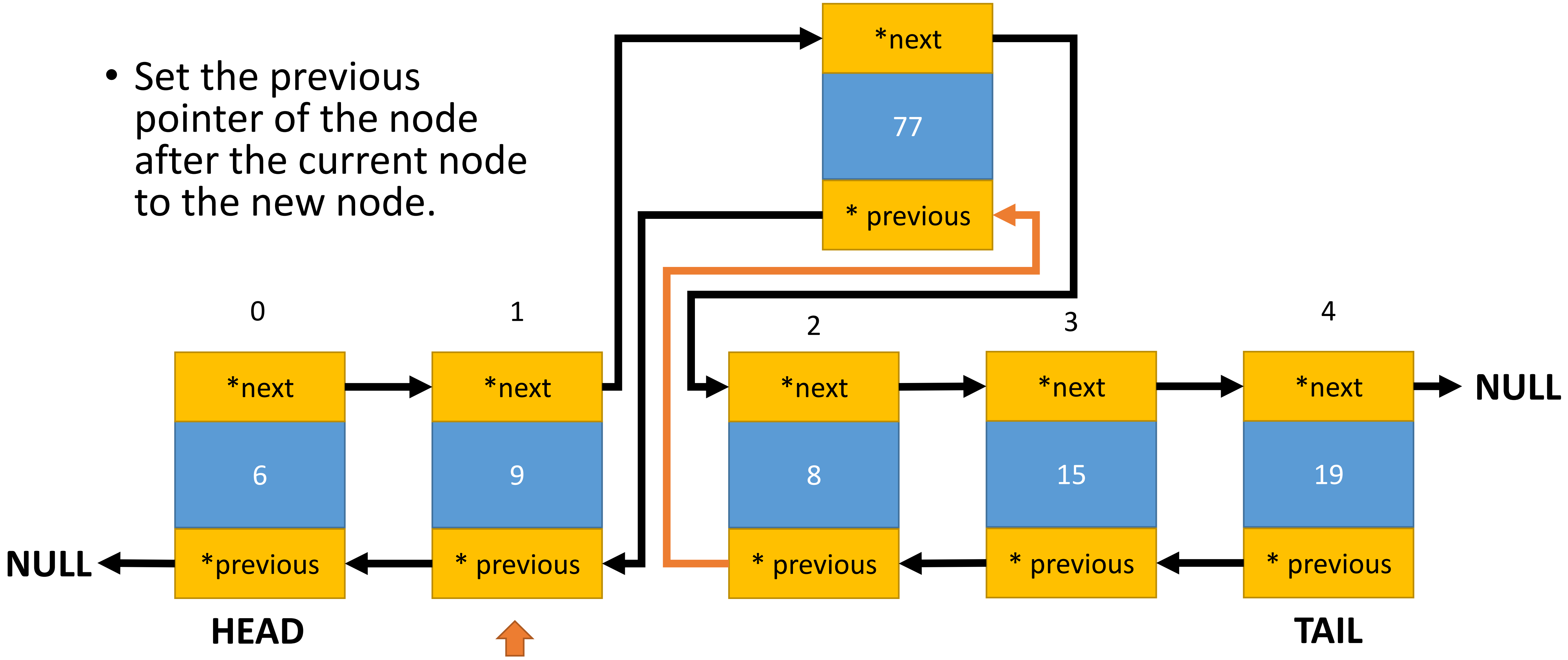
Doubly Linked Lists (Insertion)

- Set the current node's next pointer to the new node.

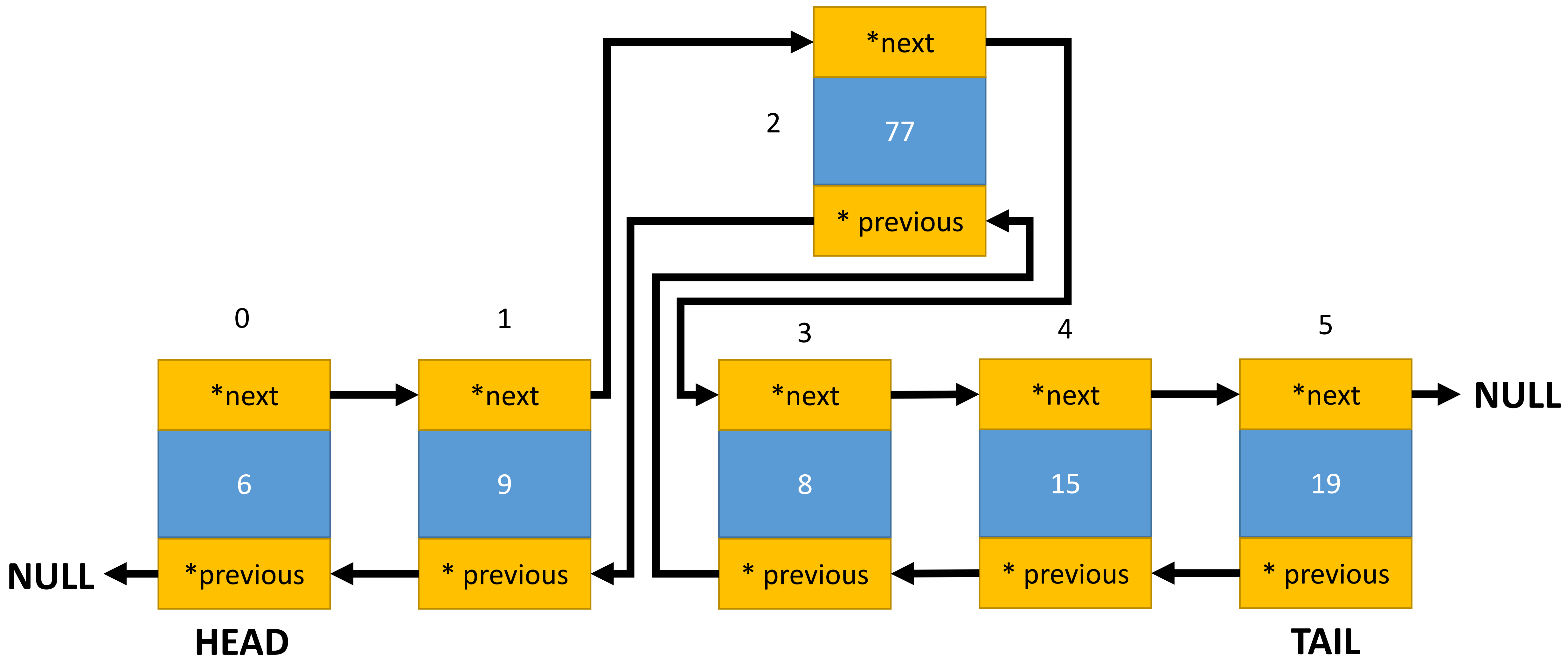


Doubly Linked Lists (Insertion)

- Set the previous pointer of the node after the current node to the new node.



Doubly Linked Lists (Insertion)



Doubly Linked Lists (Insertion)

```
void insert(int newData, int index) {
    Node *temp = head;
    int counter = 0;
    while(counter < index-1 && temp != NULL) {
        temp = temp->next;
        counter++;
    }
    Node *newNode = new Node;
    newNode->data = newData;
    newNode->next = temp->next;
    newNode->previous = temp;
    temp->next->previous = newNode;
    temp->next = newNode;
}
```

- See sample code for additional instructions that handle head and tail insertion.

Double Linked Lists (Removal)

- The process to remove a node is similar to insertion, as we need to iterate to the node immediately before the node to remove.

- Removal process itself is similar to removal of a node from a doubly linked list, but we now have to consider the previous pointers.

Doubly Linked Lists (Removal)

1. Iterate to the node one place before the position where the deletion will take place
2. Using this node, get the node two spots ahead (the one after the node to be deleted)
3. Free the node to be deleted using the free function
4. Set the previous node's next pointer to the node after the deleted node.
5. Set the previous pointer of the after node to the previous node (the one before the deleted node)

Doubly Linked Lists (Removal)

```
void erase(int index) {
      if(index < 0 || head == NULL) {
            return;
      }
      Node *prev = head;
      if(index == 0) {
            head->next->previous = NULL;
            head = prev->next;
            free(prev);
            return;
      }
      for(int i = 0; prev != NULL && i < index-1; i++) {
             prev = prev->next;
      }
      if(prev == NULL || prev->next == NULL) {
            return;
      }
      Node *after = prev->next->next;
      if(after == NULL) {
            tail = prev;
            tail->next = NULL;
            free(prev->next);
      }
      else {
            free(prev->next);
            prev->next = after;
            after->previous = prev;
      }
}
```

Doubly Linked Lists (Retrieval)

- Same process as the singly linked list.
    - Need to iterate to the desired node/position.

- Getting the head will always be O(1) and a specific function for getting the tail would also be O(1).

Doubly Linked Lists (Retrieval)

1. Check the list is not empty
2. Use a for loop to iterate/count to the desired node
3. Check to see if it is null (meaning we reached the end of the list/tried to go beyond the tail)
4. Return the desired data from the node

Doubly Linked Lists (Retrieval)

```
int get(int index) {
    if(index < 0 || head == NULL) {
        return 0;
    }
    Node *current = head;
    for(int i = 0; current != NULL && i < index; i++) {
        current = current->next;
    }
    if (current == NULL) {
        return 0;
    }
    return current->data;
}
```

Array and List Complexity Comparison

Operation	**Array**	**Singly Linked List**	**Doubly Linked List**
Appending/Prepending	O(n)	**O(1)**	
Forward Traversal	O(n)	O(n)	
Backward Traversal	O(n)	N/A	O(n)
Insertion	O(n)	O(n)	
Removal	O(n)	O(n)	
Retrieval	**O(1)**	O(n)*	

- *- Retrieving the head or tail can be done in O(1)