Revised: 12/17/2019

Linked Lists I

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

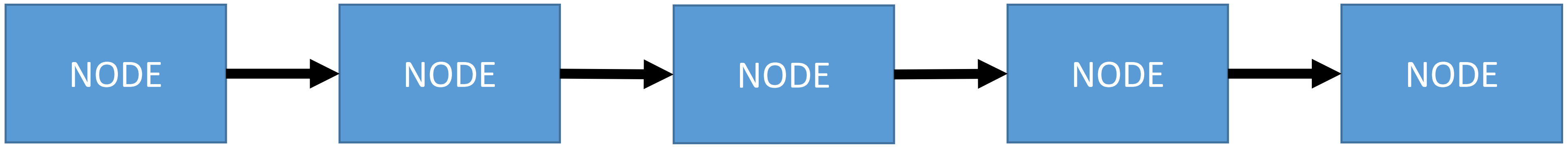
Lecture Topics

- Linked Lists
- Singly Linked Lists
  - Appending
    - Arrow Operator
  - Traversal
  - Prepending
  - Insertion
  - Retrieval
  - Removal
- Doubly Linked Lists
  - Appending
  - Traversal
  - Prepending
  - Insertion
  - Retrieval
  - Removal
- List ADTs

Linked Lists

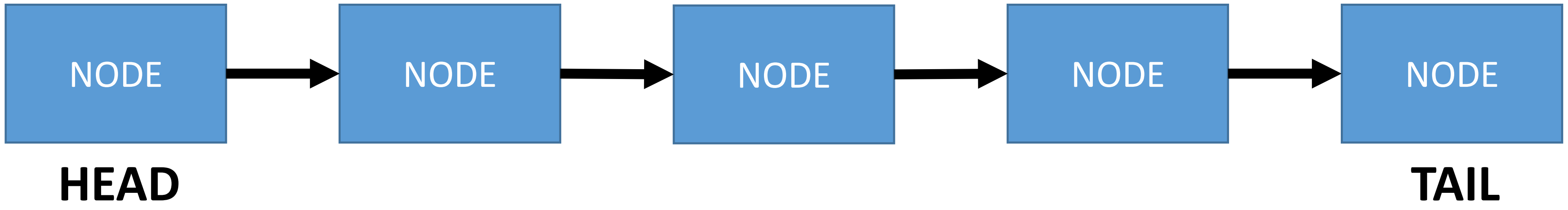
- A **linked list** is a linear data structure where a series of objects ("nodes") are connected to each other using pointers or other forms of references/variables.

- Each node has a reference to the next node (and in come cases the previous node) in the list.



Linked Lists

- The first node in the list is referred to as the list’s **head**.
- The last node in the list is referred to as the list’s **tail**.



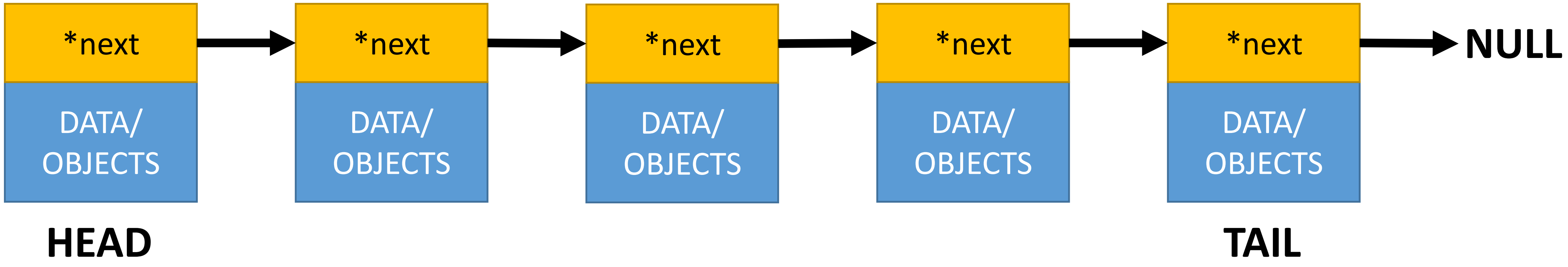
Linked Lists vs Arrays

- Arrays require the use of fixed-length, contiguous memory.

- A list does not have this requirement; Nodes can reside anywhere in memory.
  - Each node in the list knows the location of the next (and sometimes the previous) node.

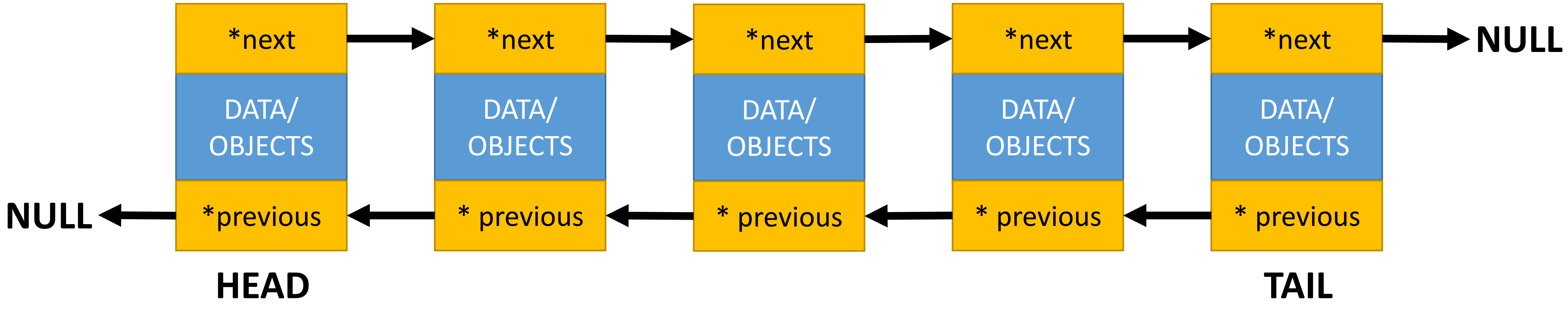
Types of Linked Lists

- Singly Linked List (SLL)
  - Each node contains a reference to the next node in the list.
  - The tail node's next reference is null.



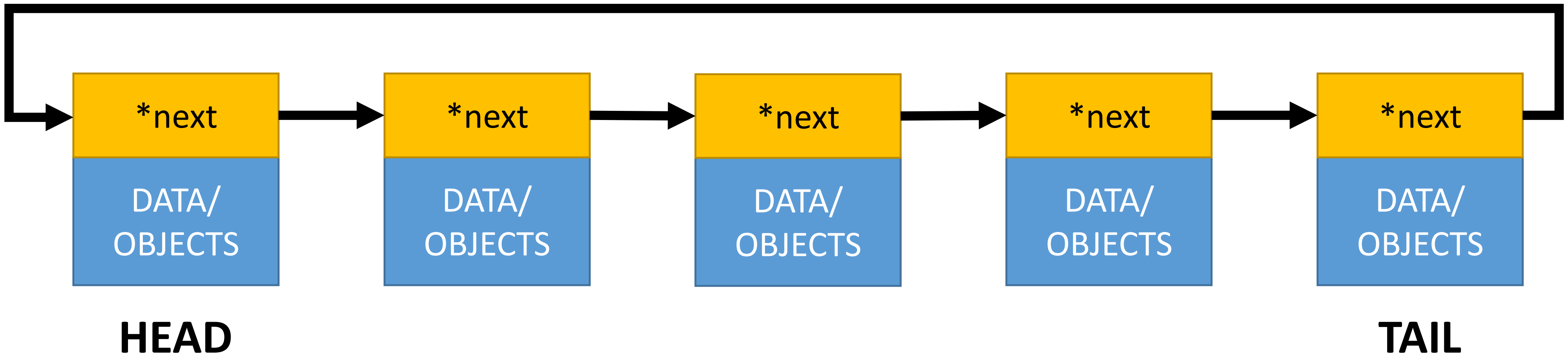
Types of Linked Lists

- Doubly Linked List (DLL)
  - Each node contains a reference to the next **and previous** node in the list.
  - The tail node's next reference is null.
  - The head's previous reference is null.



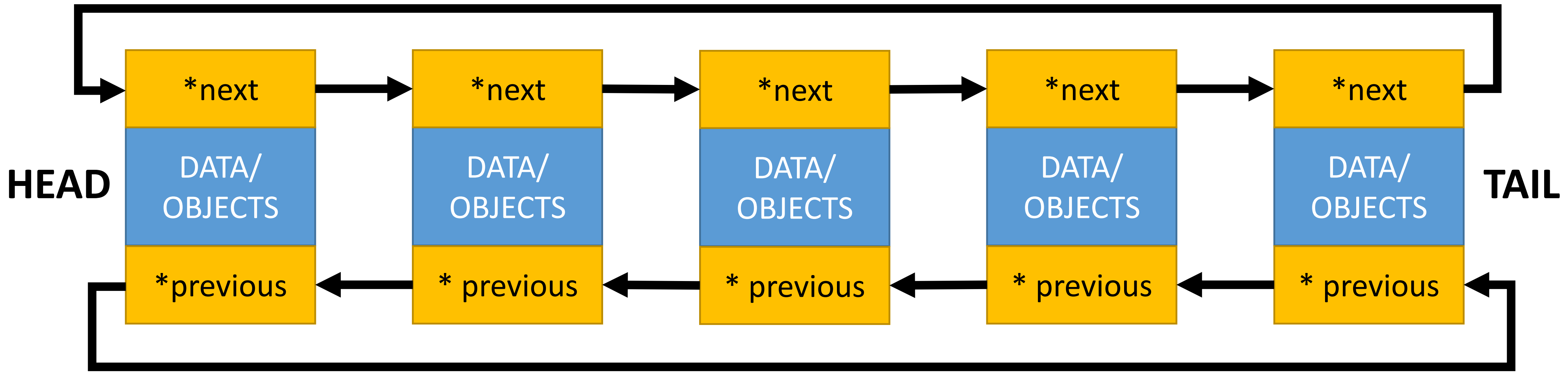
Types of Linked Lists

- Circular Linked List (CLL)
  - Each node contains a reference to the next node in the list.
  - The tail node's next reference is **the head**.



Types of Linked Lists

- Circular (Doubly) Linked List

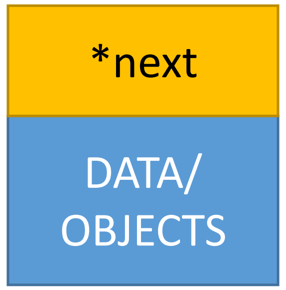


- We’ll see circular linked lists in the next lecture.

Singly Linked Lists

- How a node for a linked list is designed depends on the application of the list.

- In the case of the singly linked list, each node will have a pointer/reference to the next node.
  - The node can contain any data or objects that it needs.



Singly Linked Lists

- In these examples, we’ll use a C++ struct for creating the node.
  - A struct is like a class, but all fields and functions are public by default; In a C++ class, fields and functions are private by default.

- Nodes can just as easily be designed as a class instead of a struct.

Singly Linked Lists

- A sample struct to be used as the nodes in a list:

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

- In addition to the “data” field, this Node struct could contain other values, objects, or functions.
    - They can also be specified as public or private.

Singly Linked Lists

- The class itself for the Linked List will need to maintain pointers to the head and tail of the List.

```
class SLinkedList {
    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
        SLinkedList() {
            head = NULL;
            tail = NULL;
        }
}
```

Singly Linked Lists (Appending)

- With a singly linked list, new nodes are typically added (“pushed”) to the back of the list.

1. Create the new node to be added.
    - Make sure it’s 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)
3. Otherwise, set the current tail’s next pointer to point to the new node.
4. Set the list’s tail pointer to the new node.

The Arrow Operator

- The arrow operator **->** is used to access the members of an object when the variable is a *pointer*.
    - Regular variable referencing an object (use dot operator):

```
someObj.someField = 7;
int value = someObj.getField();
```

    - Pointer to an object (use arrow operator):

```
ptrObj->someField = 7;
int value = ptrObj->getField();
```

Singly Linked Lists (Appending)

```
void push_back(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = NULL;

    if(head == NULL) {
        head = temp;
        tail = temp;
    }
    else {
        tail->next = temp;
        tail = tail->next;
    }
}
```

Singly Linked Lists (Appending)

- The complexity of appending a value to the tail of the list is O(1).
  - For an array, it would be O(n) as this would require creating a new array with extra space and then moving the existing values into the new array.

Singly Linked Lists (Traversal)

- With a singly linked list, nodes are traversed from head to tail.

1. Start with the head node.
2. As long as it’s not null (empty list), get the node’s data.
3. Do any necessary processing with the node.
4. Use the node’s next pointer to get the next node.
5. Once we get a null pointer, it means we’ve reached the tail (since the tail’s next pointer will point to null)

Singly Linked Lists (Traversal)

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

- The complexity of traversing any linked list is O(n).
  - Same as an array.

Singly Linked Lists (Prepending)

- New nodes can also be added to the front of the list (“pushing to the front”).

1. Create the new node to be added.
   - Make sure it’s next pointer points to the current head.
2. Set the list’s head pointer to the new node.
3. Check if the list’s tail is null (meaning the list was empty) and update the tail pointer, if needed.

- Like appending to the back, this also has a complexity of O(1)
  - For an array, this would be O(n) for the same reason as appending to the end of the array.

Singly Linked Lists (Prepending)

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

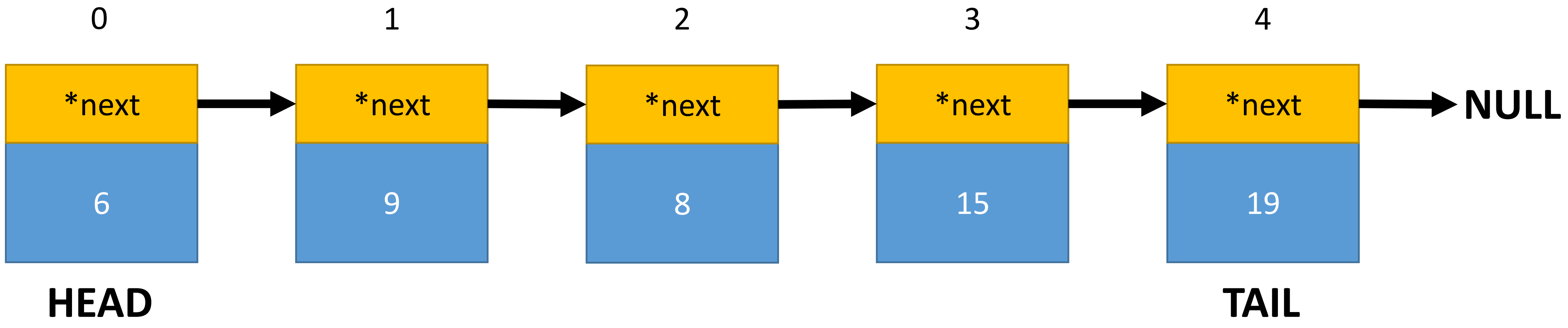
Singly Linked Lists (Insertion)

- While lists don’t have indexes like arrays do, it’s possible to insert a new node at a certain position in the list.

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 current node’s next pointer to the new node.

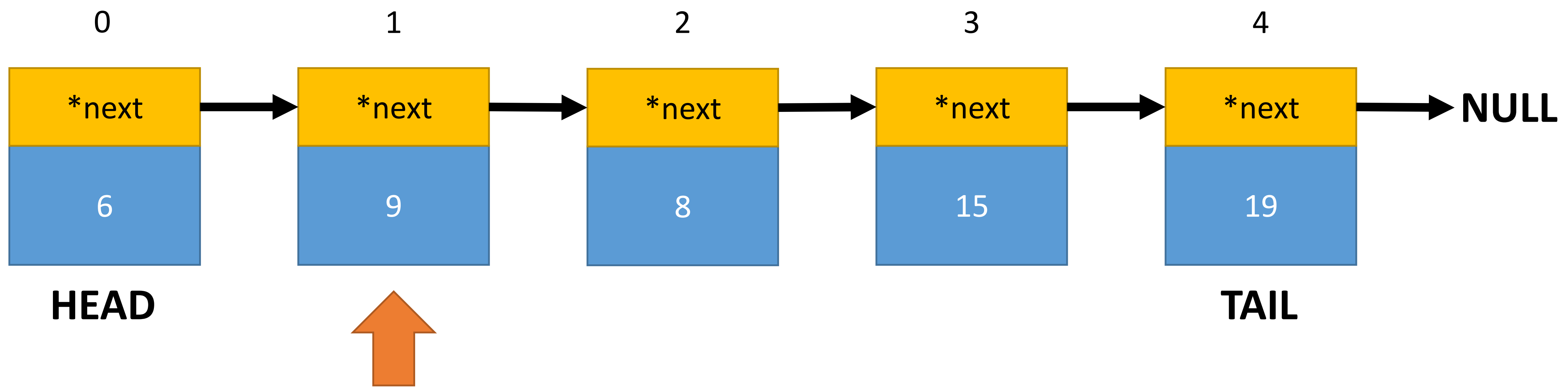
Singly Linked Lists (Insertion)

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



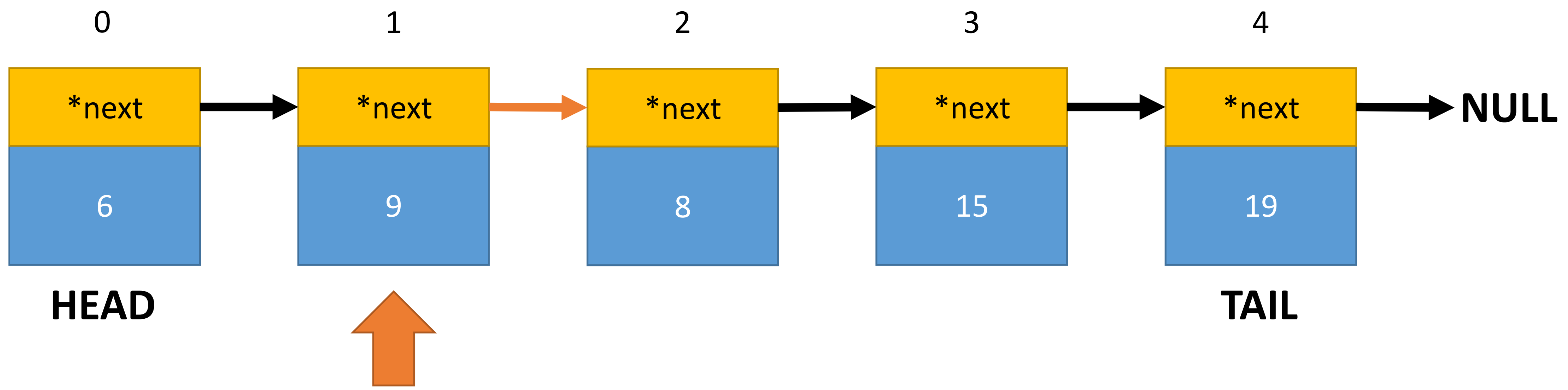
Singly Linked Lists (Insertion)

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



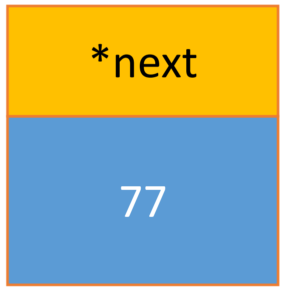
Singly Linked Lists (Insertion)

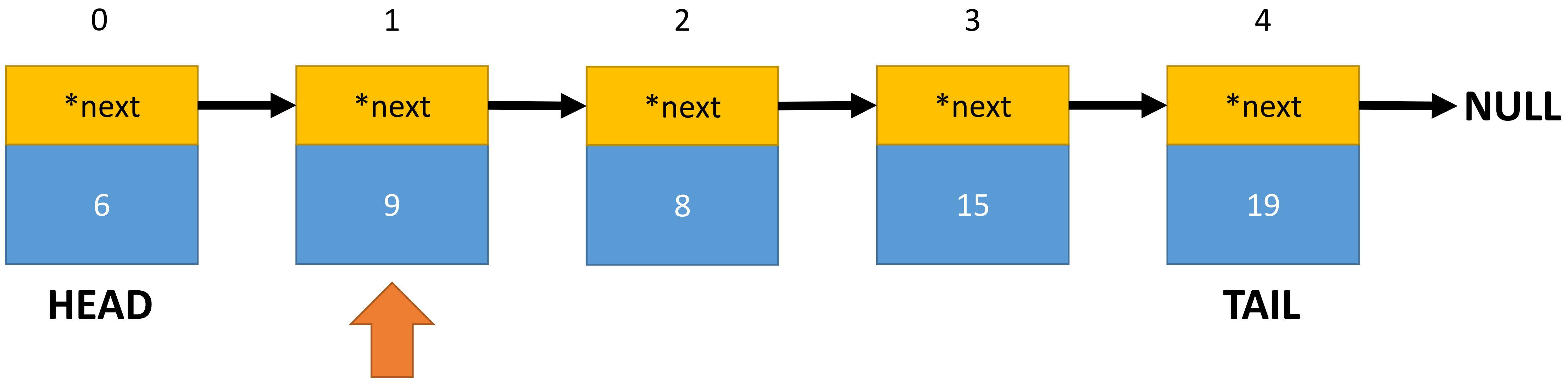
- Check that it’s next pointer is not null



Singly Linked Lists (Insertion)

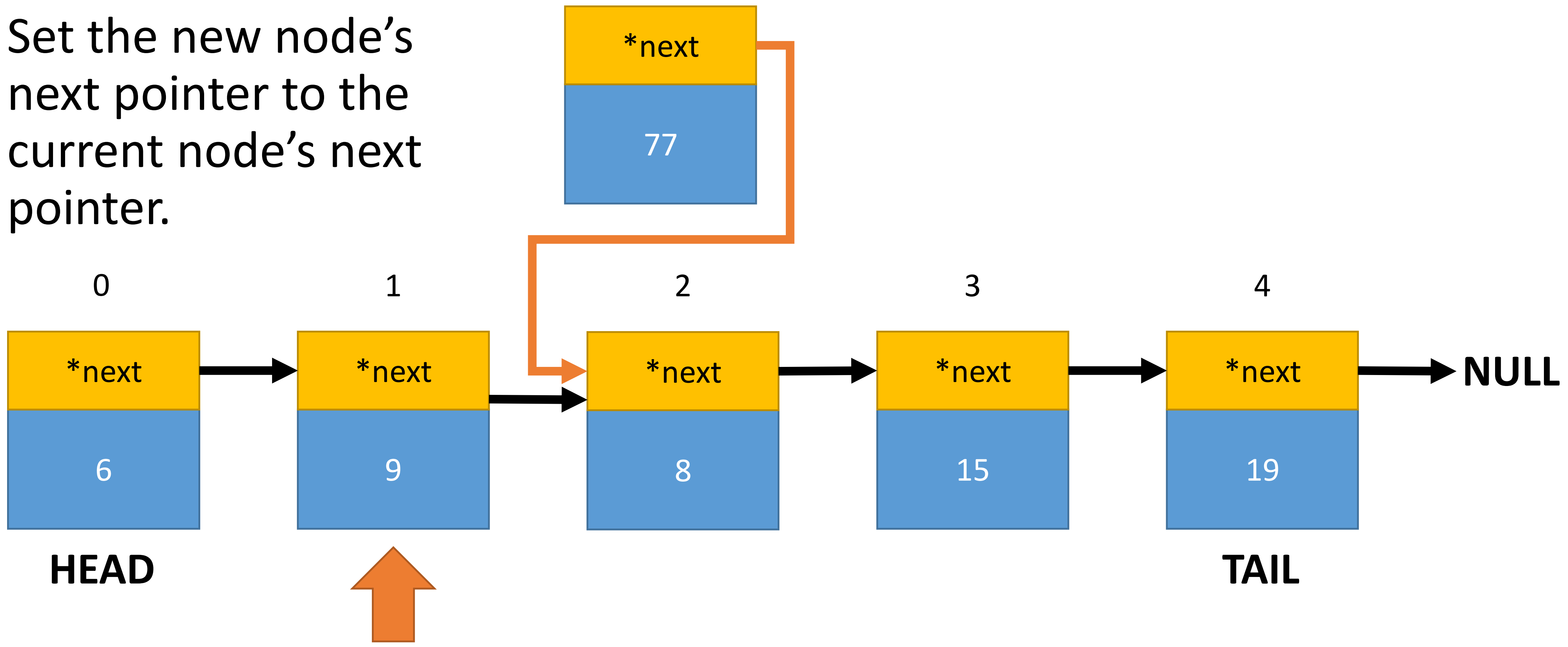
- Create the new node





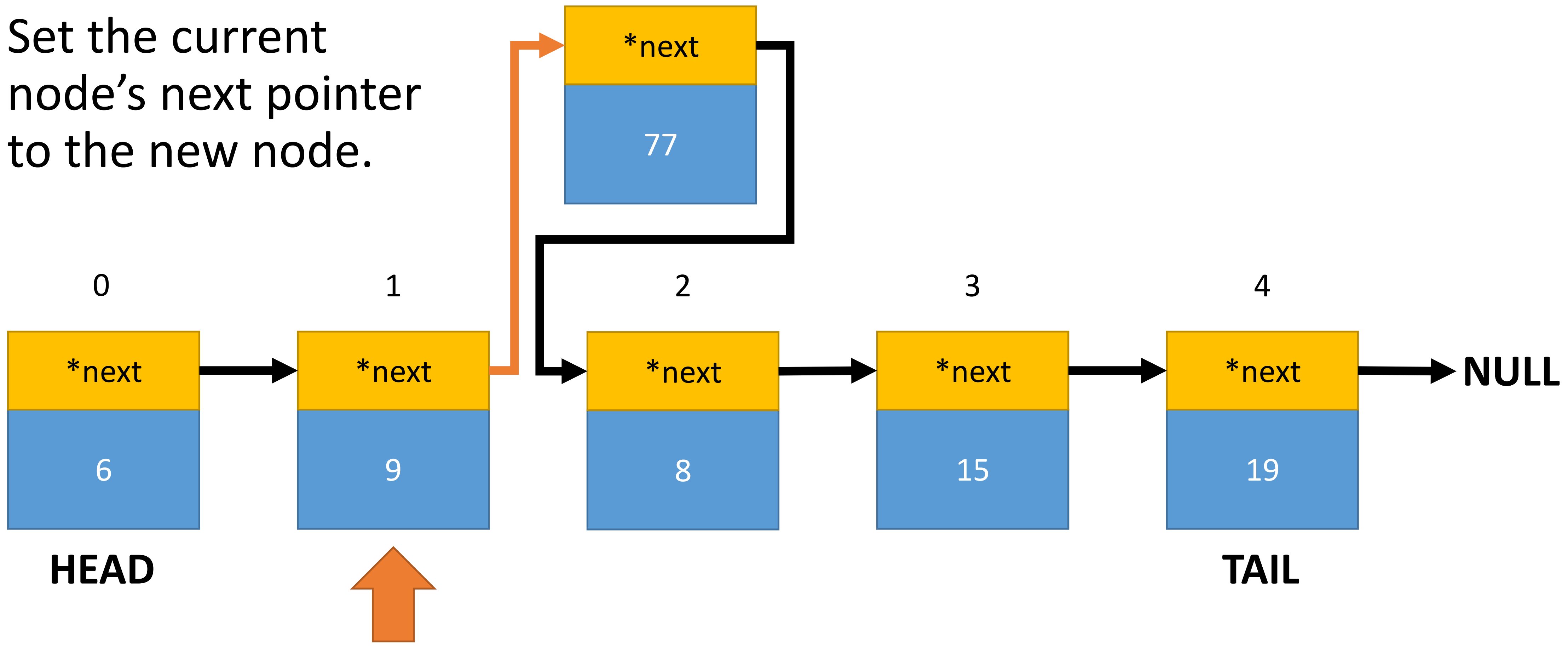
Singly Linked Lists (Insertion)

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

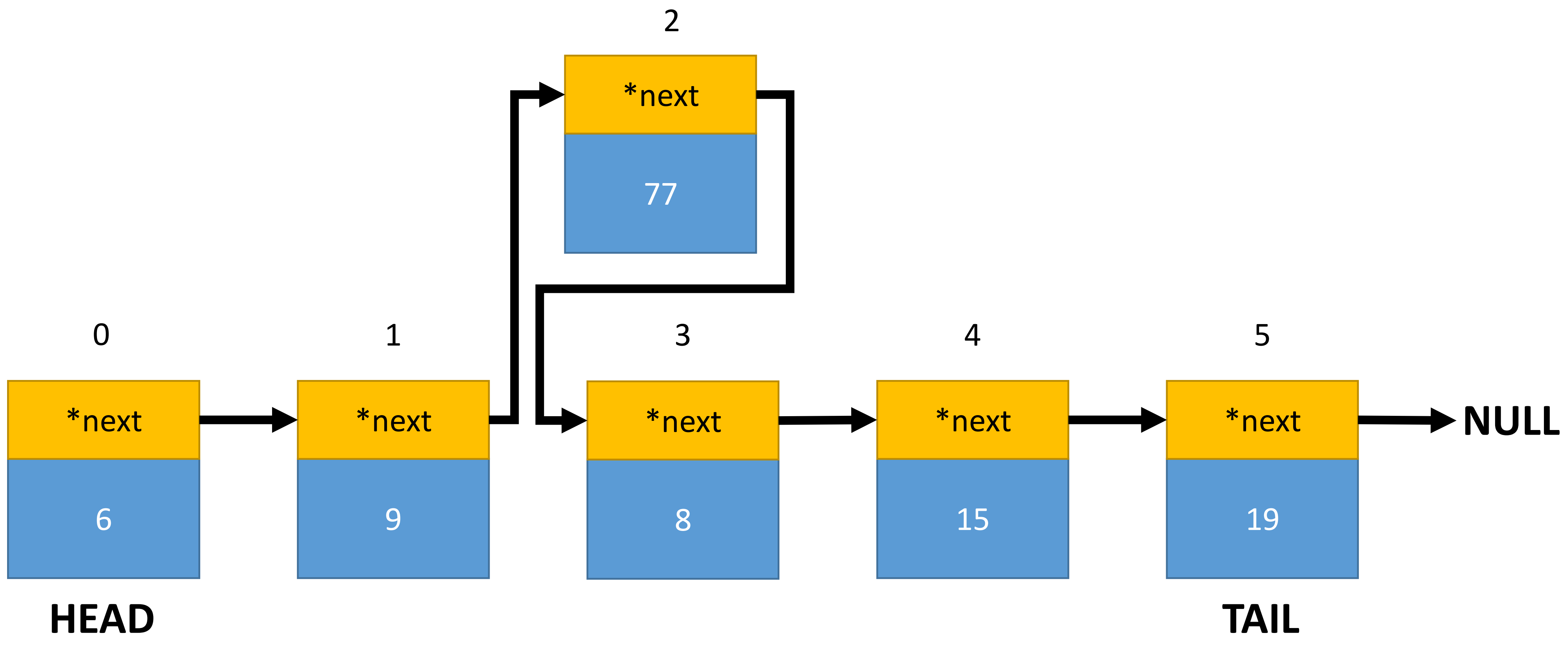


Singly Linked Lists (Insertion)

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



Singly Linked Lists (Insertion)



Singly Linked Lists (Insertion)

- The complexity of inserting a value (not front or back) in the list is O(n).
  - This would be the same complexity for a comparable operation on an array.

- Although, with an array every value would always need to be visited; A list only needs to go as far as the insertion point.

Singly 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++;
    }

    if(temp == NULL || temp->next == NULL) {
        return;
    }
    else {
        Node *newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
```

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

Singly 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.
    - Same complexity as insertion, O(n).

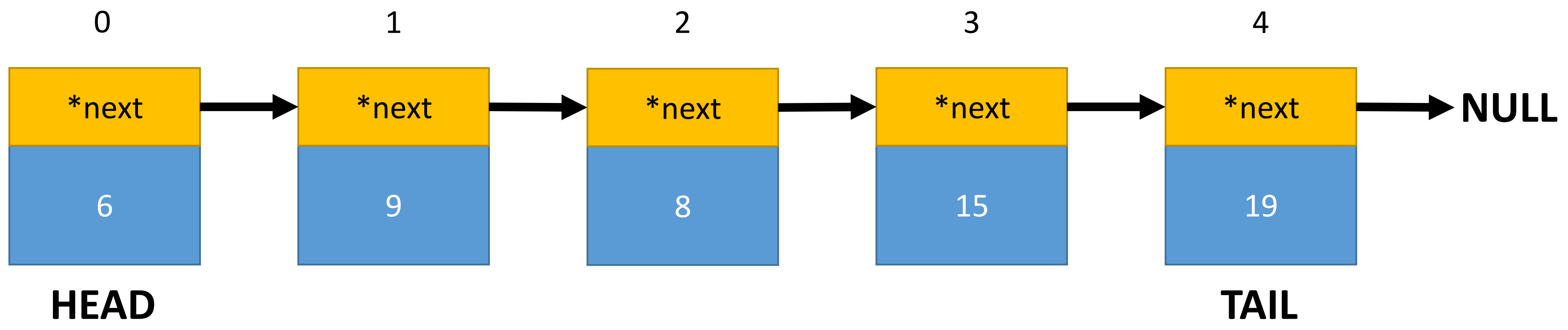
- Will only need to go as far as the deletion point.

Singly 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.

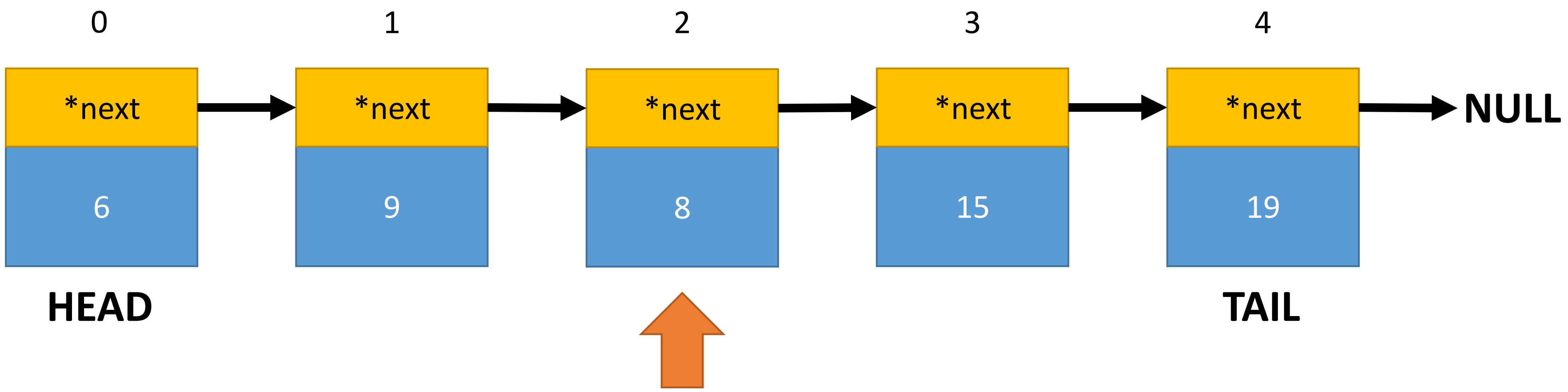
Singly Linked Lists (Removal)

- Removing the node at position 3



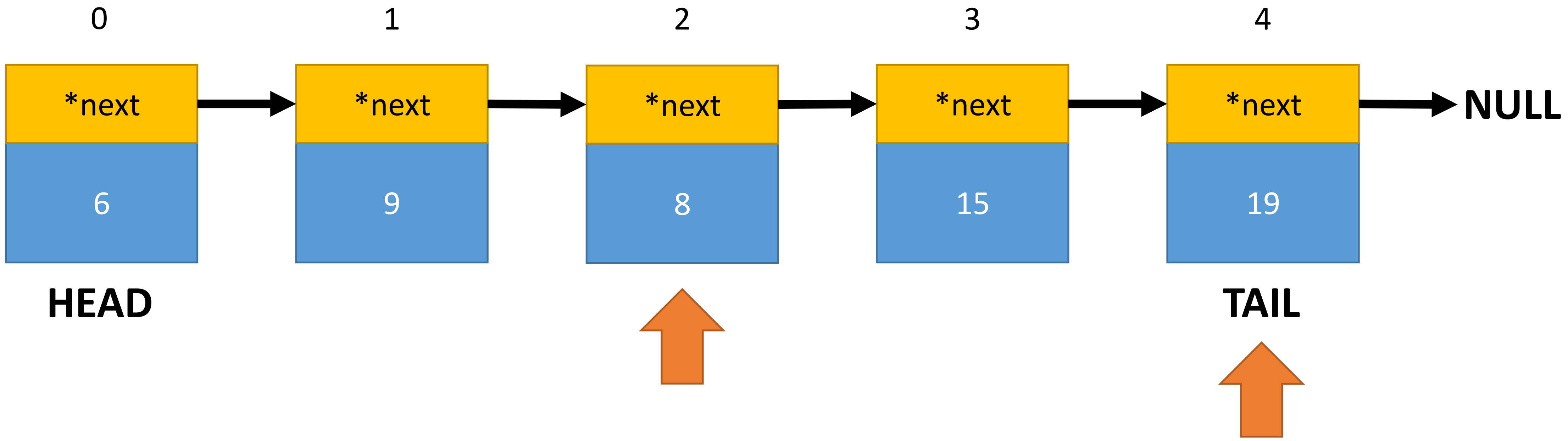
Singly Linked Lists (Removal)

- Iterate to position 2 (one place before the removal point)



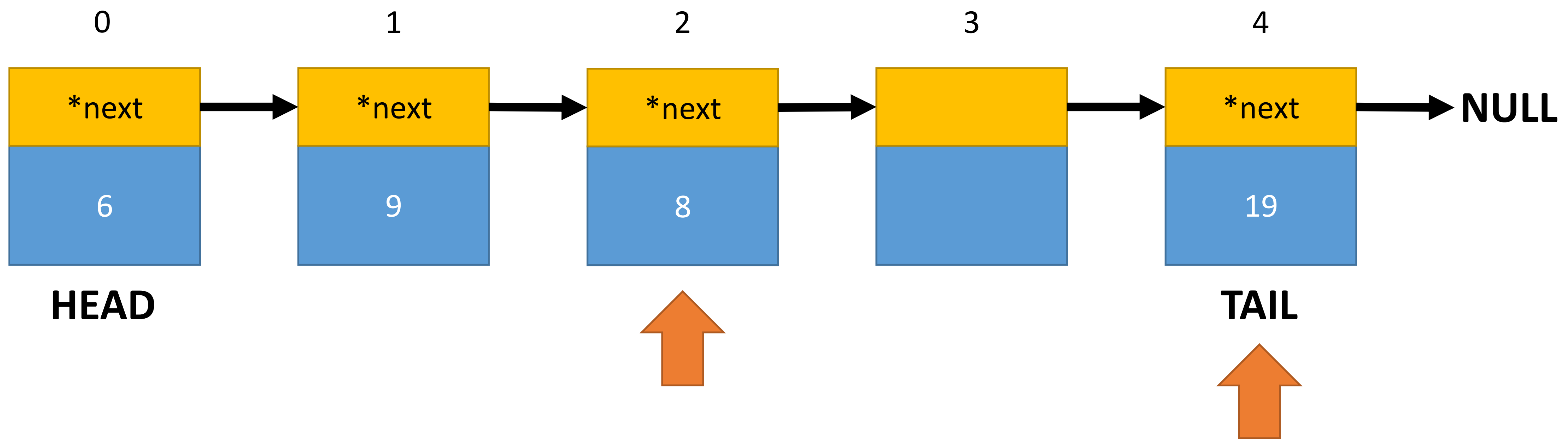
Singly Linked Lists (Removal)

- Get the node after the node to be deleted



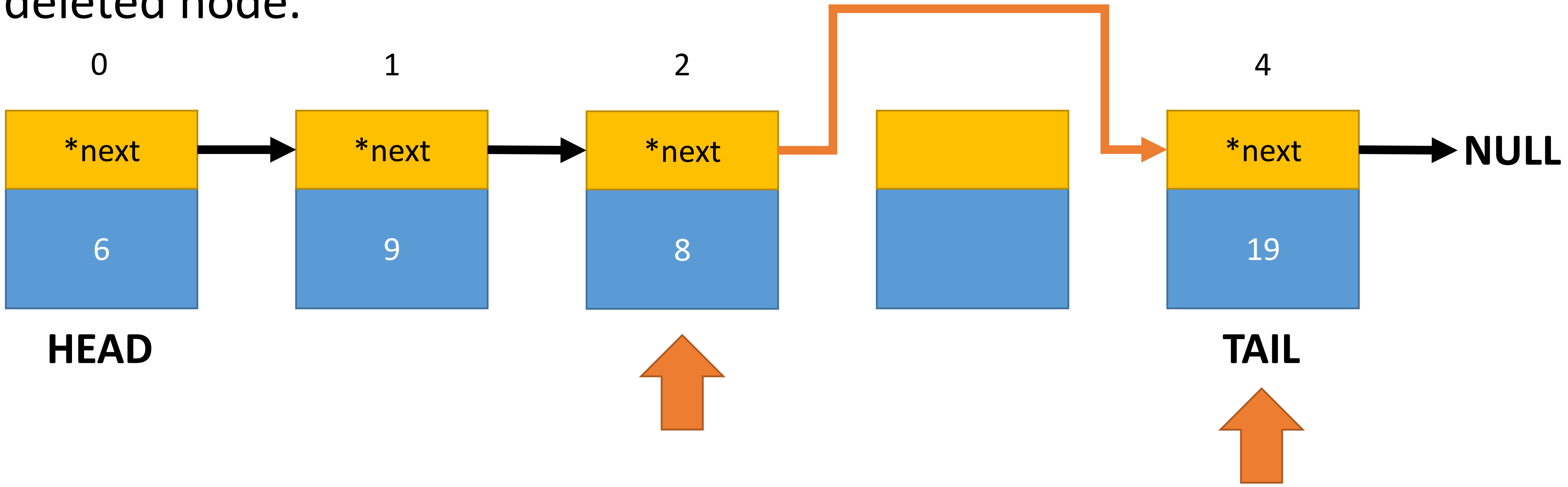
Singly Linked Lists (Removal)

- Free the node to be deleted

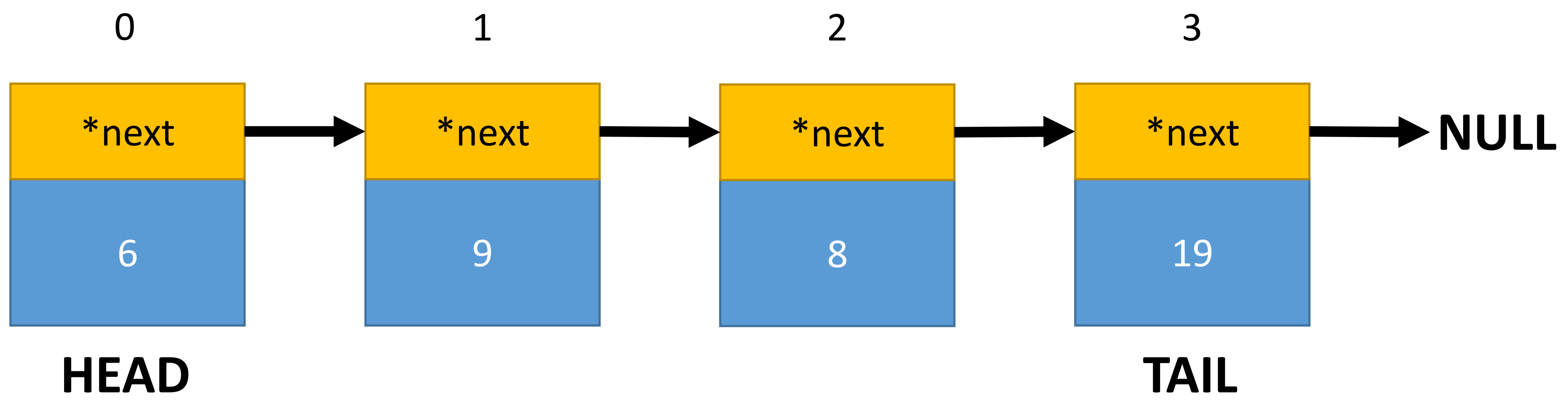


Singly Linked Lists (Removal)

- Set the previous node’s next pointer to the node after the now deleted node.



Singly Linked Lists (Removal)



Singly Linked Lists (Removal)

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

Singly Linked Lists (Retrieval)

- Lists aren’t indexed, and unlike arrays that use contiguous memory we can’t access elements with subscript notation like list[4]

- Lists need to iterate to the desired node/position.
  - This gives retrieval from a list the complexity of O(n)
  - Array retrieval is always O(1)

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

Singly 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

Singly 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;
}
```

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;
}
```

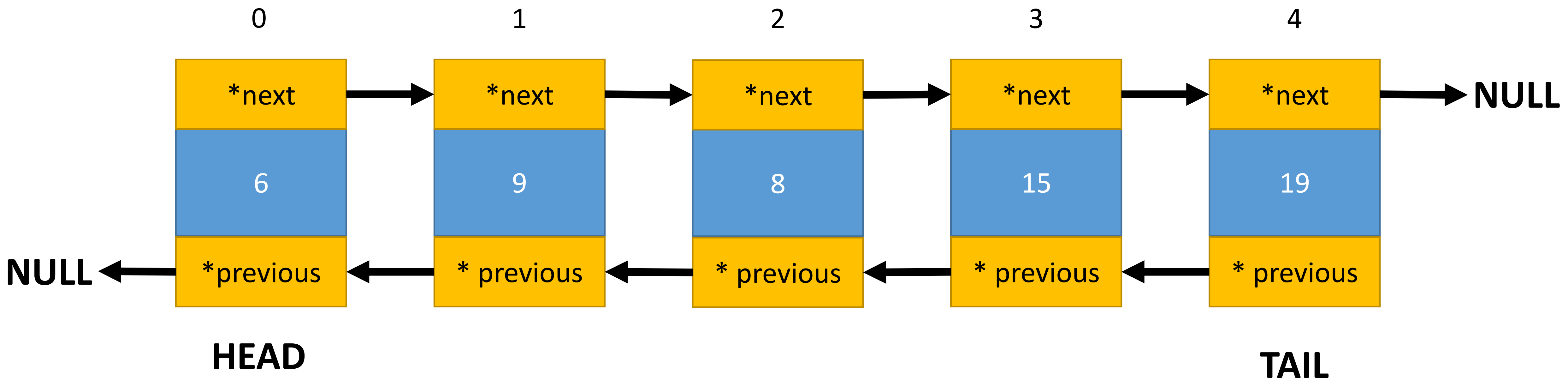
Doubly Linked Lists (Insertion)

- 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.

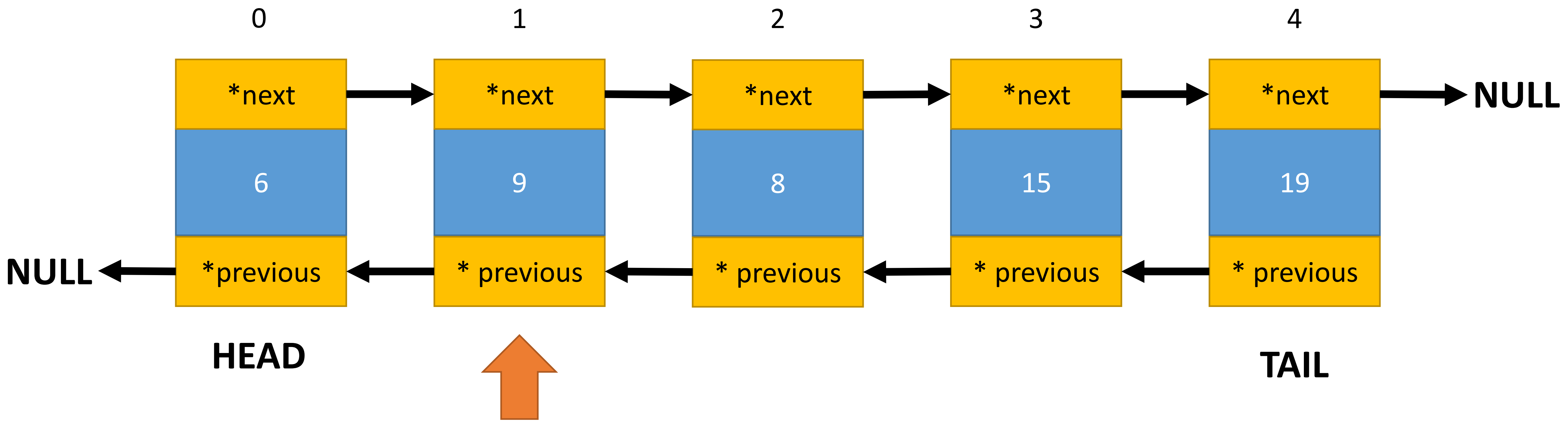
Doubly Linked Lists (Insertion)

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



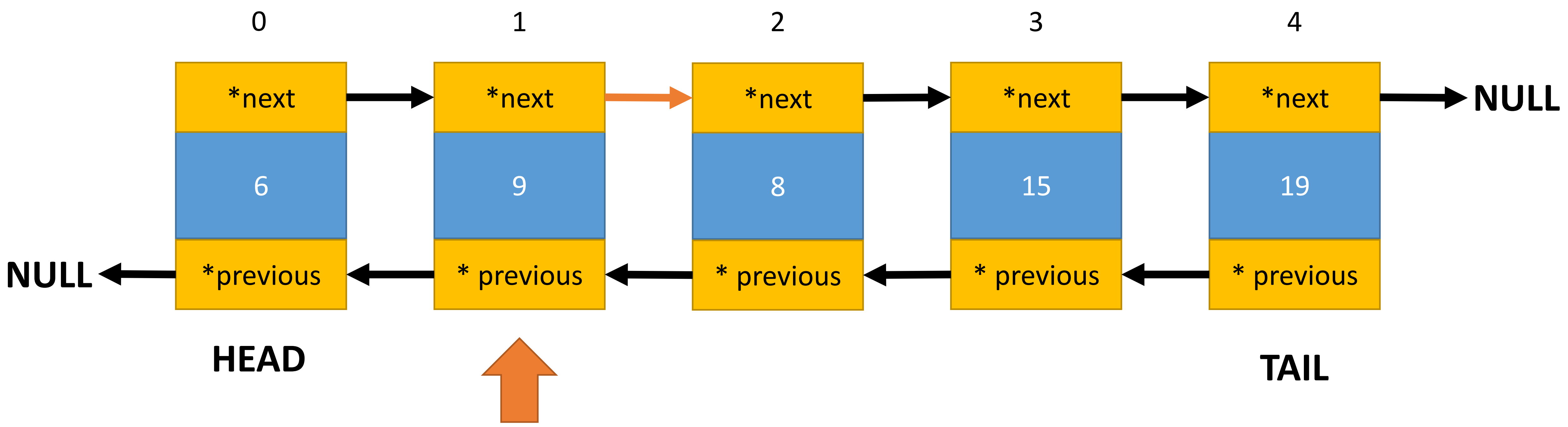
Doubly Linked Lists (Insertion)

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



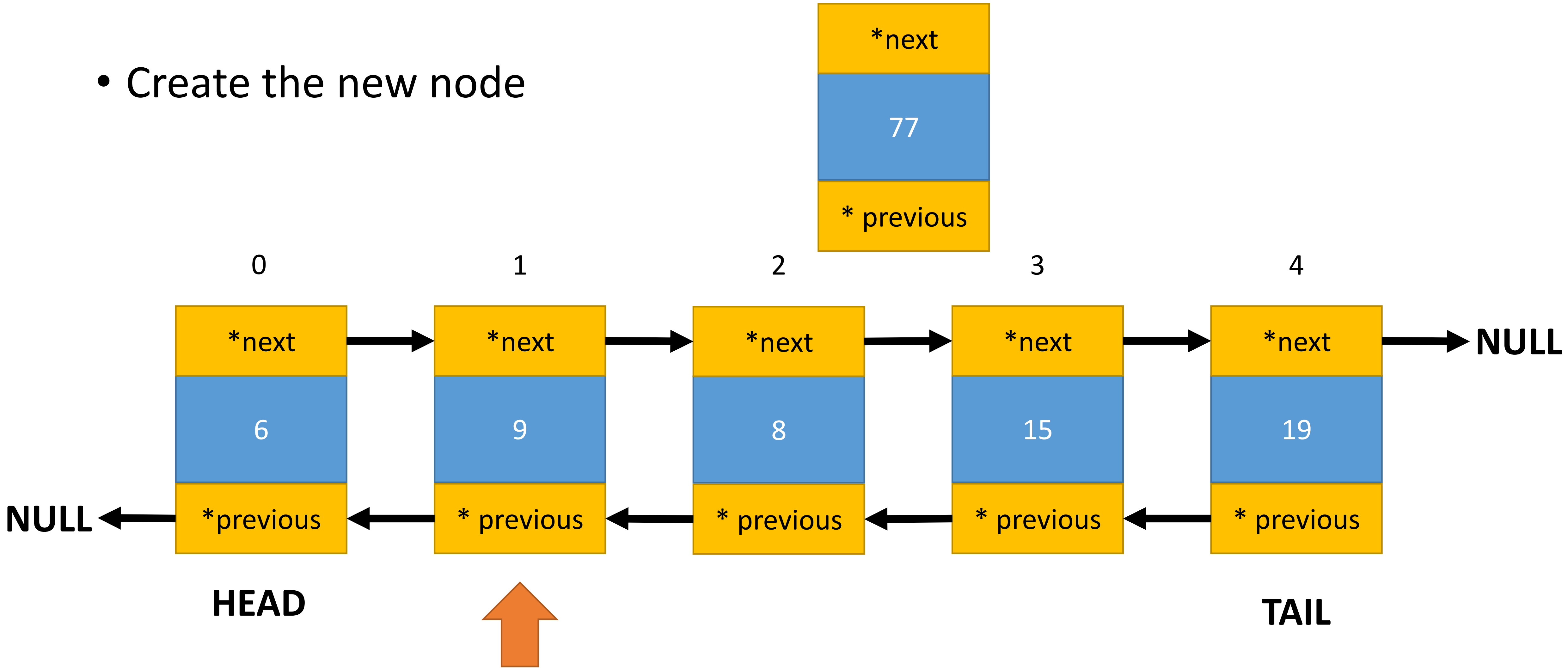
Doubly Linked Lists (Insertion)

- Check that it’s next pointer is not null



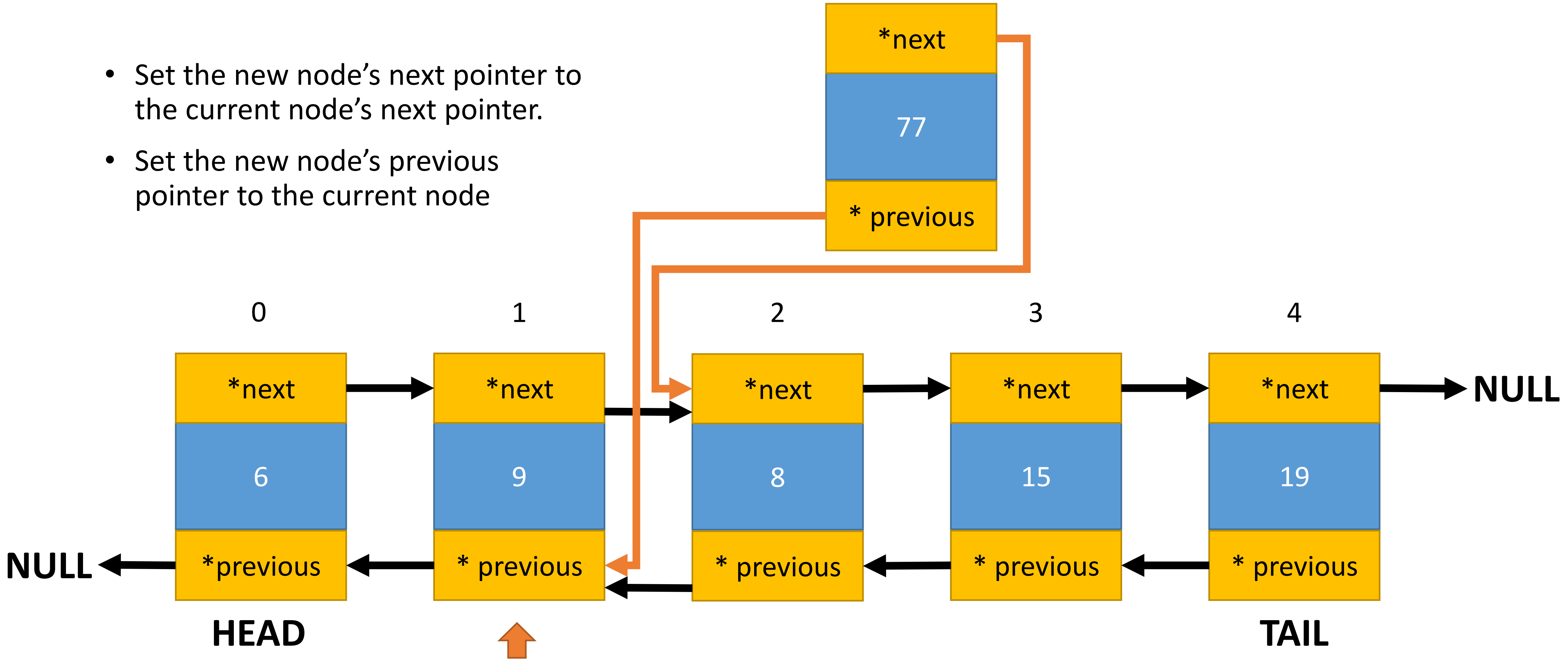
Doubly Linked Lists (Insertion)

- Create the new node



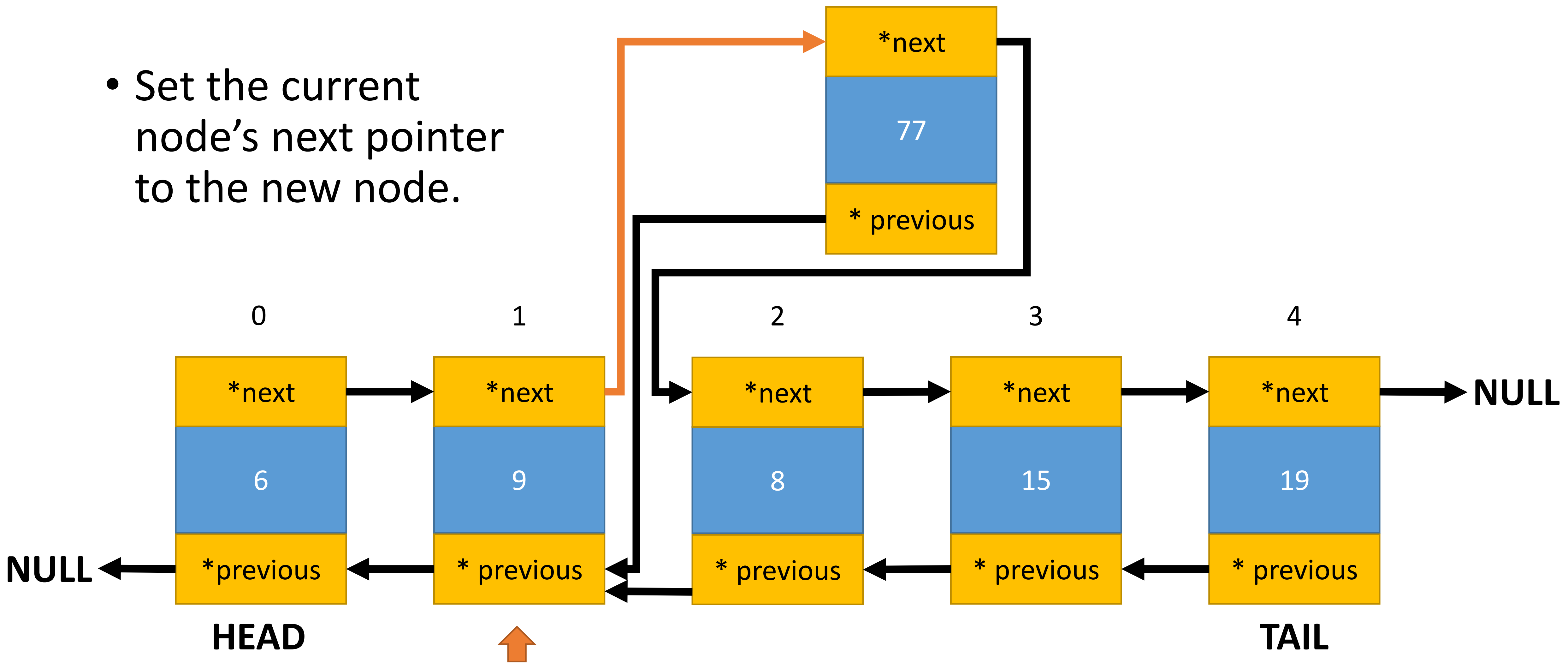
Doubly Linked Lists (Insertion)

- 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



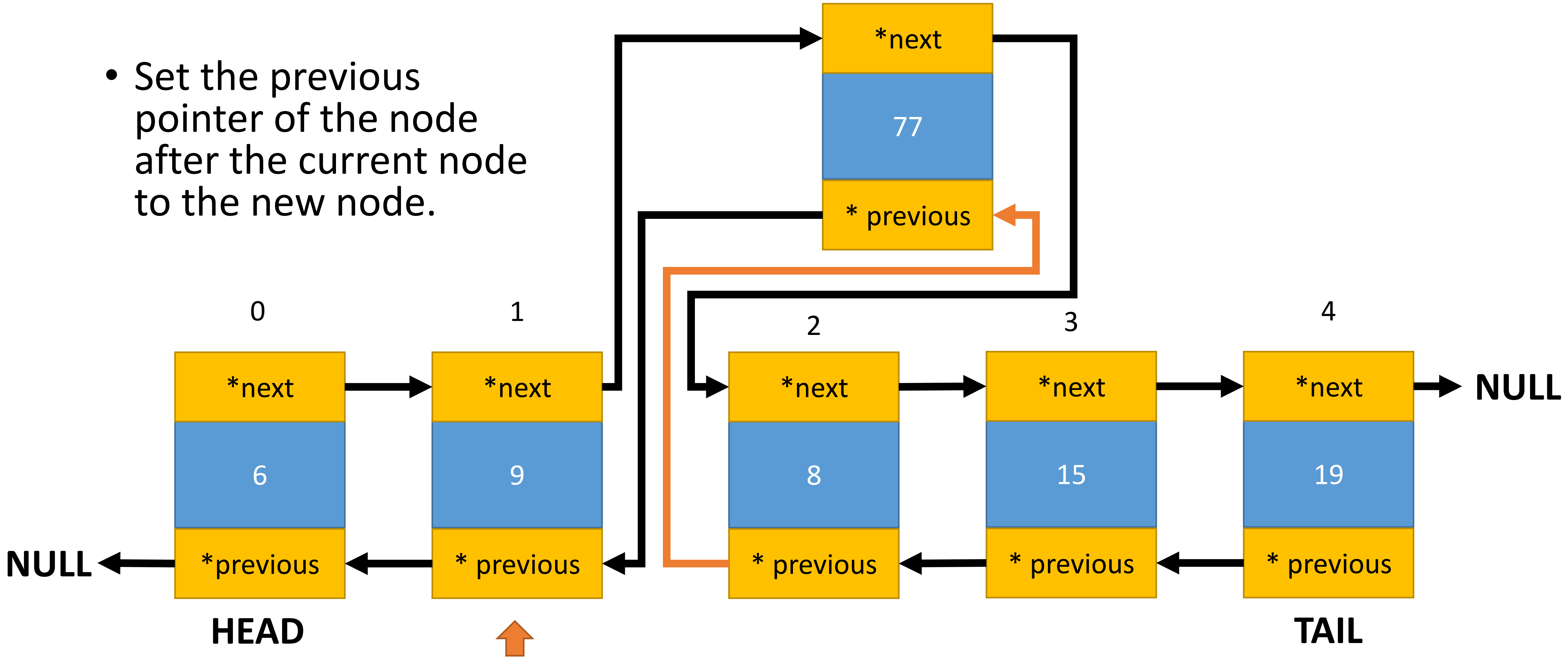
Doubly Linked Lists (Insertion)

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

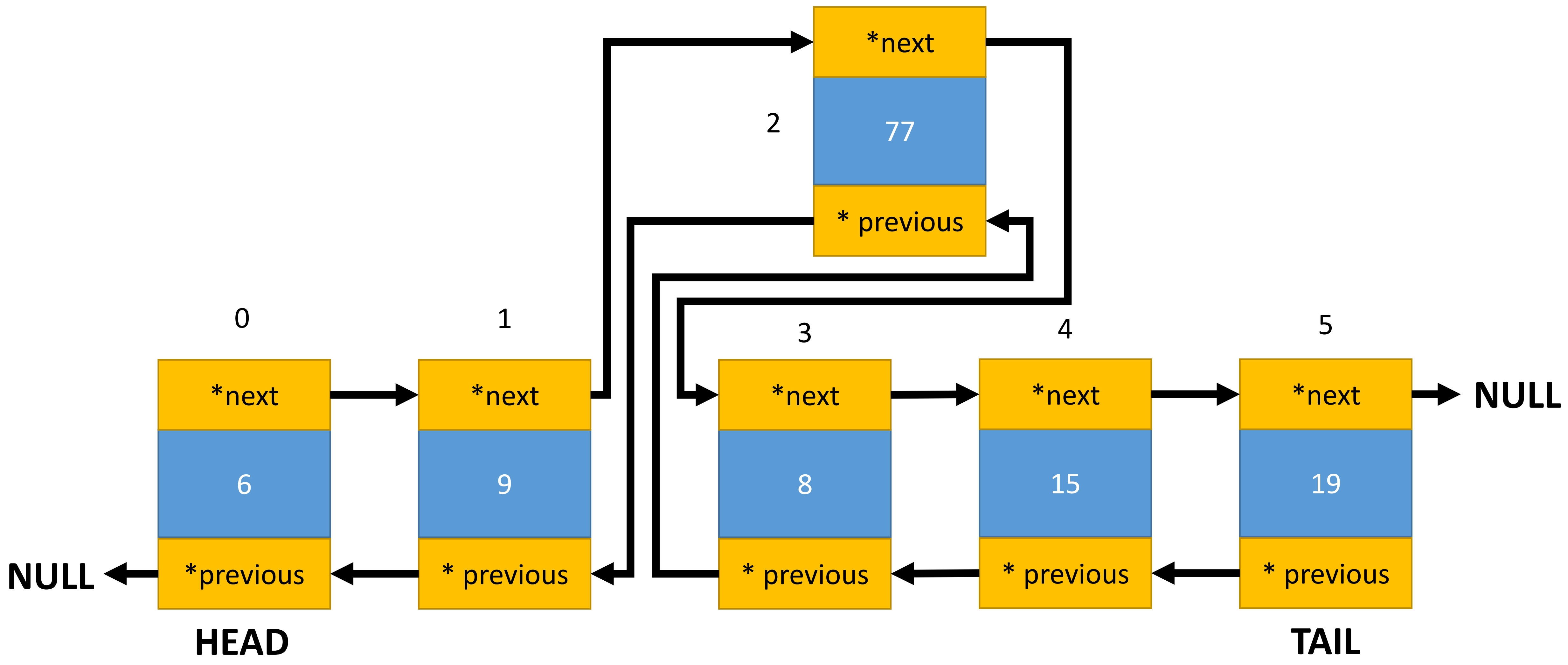


Doubly Linked Lists (Insertion)

- 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)

List ADTs

- Many programming languages include built-in abstract data types for making lists.
  - As of Python 3.6, Python does not have a built-in linked list.

- C++’s List type is one such example
  - Implemented as doubly linked.

- Java’s List interface is implemented by a number of different List-type ADTs
  - Two such ADTs in Java are:
    - LinkedList (doubly linked)
    - ArrayList (NOT a linked list)

- Examples of all three are provided in the Sample Code/Module Download

List ADTs

- Java’s ArrayList is a frequently used replacement for an array.
    - It is essentially a class that encapsulates an array and provides list-like functions such as appending, insertion, deletion, etc.
- The ArrayList will have the same complexities of an array.
    - For example, appending data onto an ArrayList will have a complexity of O(n) while appending data onto a LinkedList will have a complexity of O(1).
- Python’s standard list type is like a mix between a Java array and ArrayList.