CIT237
Chapter 18: Linked Lists

November 20, 2019

Announcement / Reminder

- Quiz 7 will be held at the start of class on

  Wednesday, December 11.

  The material covered on Quiz 7 will be announced as the date gets closer.

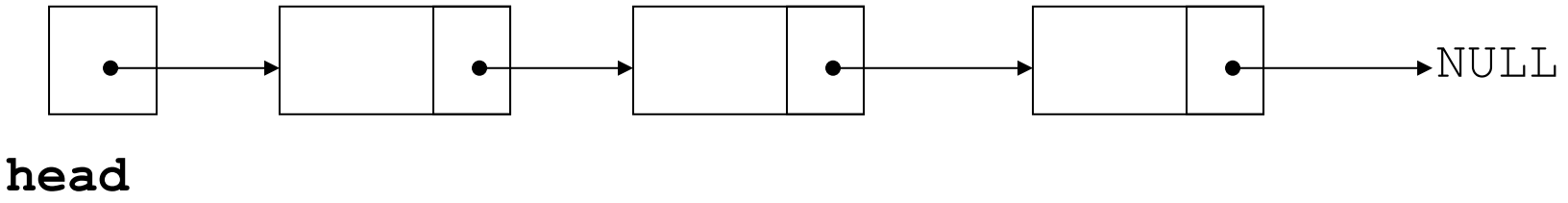
- Project 3:

  Due date is December 2.

Introduction to the Linked List ADT

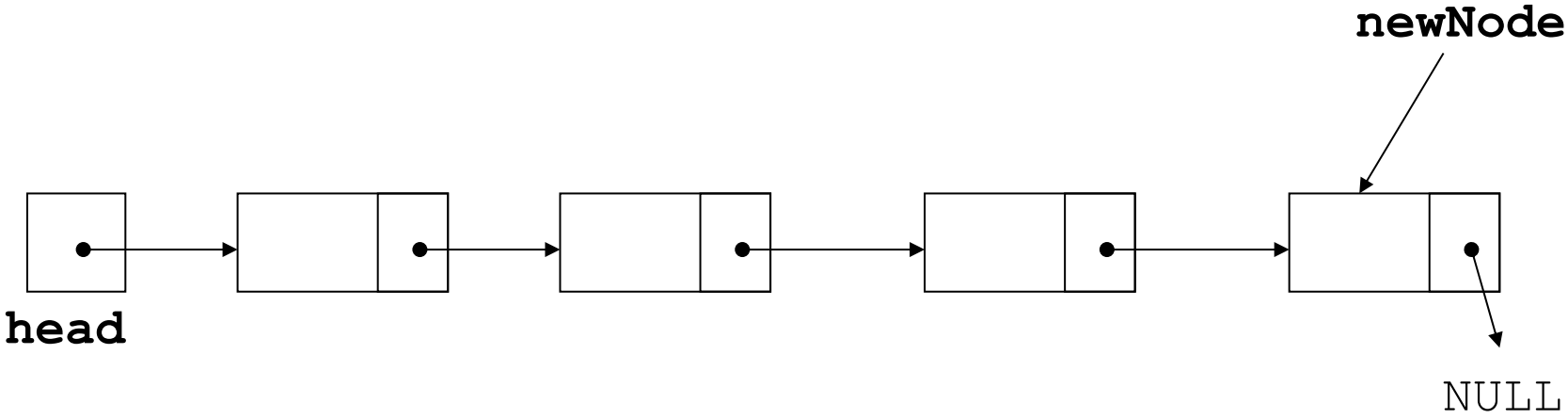
- Linked list: set of data structures (nodes) that contain references to other data structures

("Reference" is meant in a general way here, not necessarily indicating a particular syntax.)



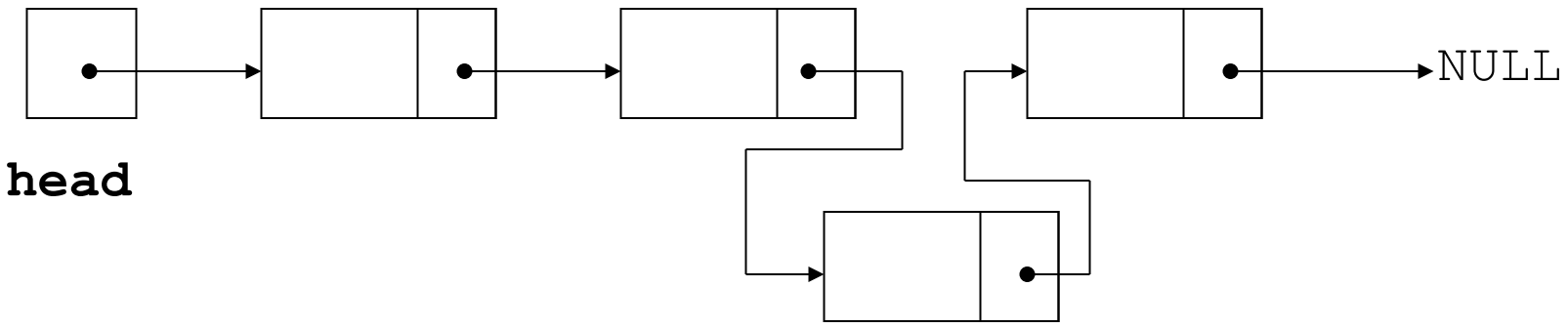
A Dynamic Data Structure

- References may be addresses or array indices (in C++, usually pointers).
- Data structures can be added to or removed from the linked list during execution



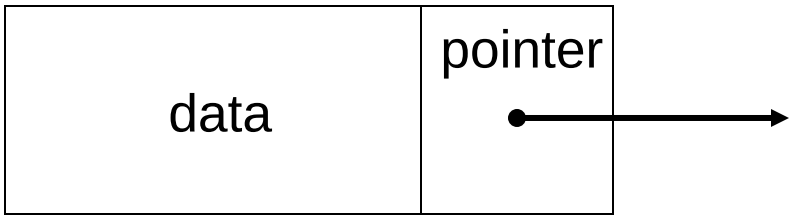
Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size.
- Linked lists are NOT necessarily stored sequentially.
- Linked lists can insert a node between other nodes easily.



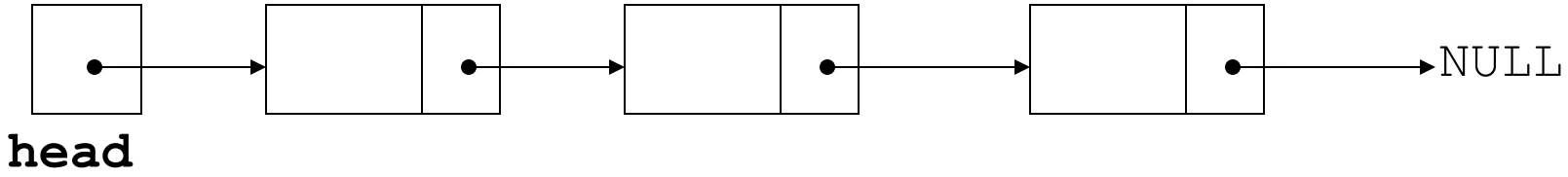
Node Organization

- A node contains:
  - data: one or more data fields – may be organized as structure, object, etc.
  - a pointer that can point to another node



Linked List Organization

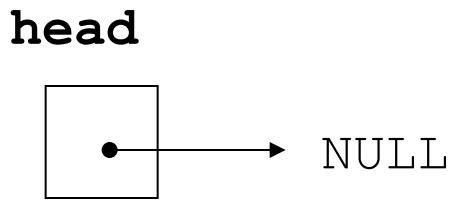
- Linked list contains zero or more nodes:



- Has a list head to point to first node
- Last node points to `NULL`

Empty List

- If a list currently contains zero nodes, it is the empty list
- In this case the list head points to `NULL`



Declaring a Node

- Declare a node:

```
struct ListNode
{
        int value;
        ListNode *next;
};
```

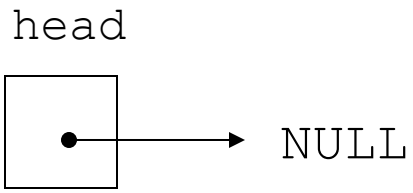
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

- Head pointer initialized to `NULL` to indicate an empty list



NULL Pointer

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

```
ListNode *p;
while (p != NULL) ...
```

- Can also test the pointer itself:

```
while (p)  ... // same meaning
               // as above
```

- Some environments use **`nullptr`** instead of **`NULL`**.

Linked List Operations

- Basic operations:
  - Insert a node at beginning of the list.
  - Traverse the linked list.
  - Append a node to the end of the list.
  - Insert a node within the list.
  - Delete a node.
  - Delete/destroy the entire list.

Create a New Node

- Allocate memory for the new node:

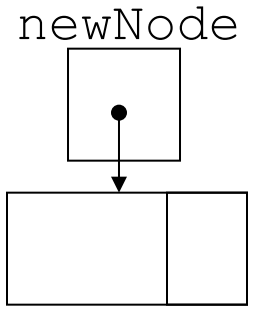
```
newNode = new ListNode;
```

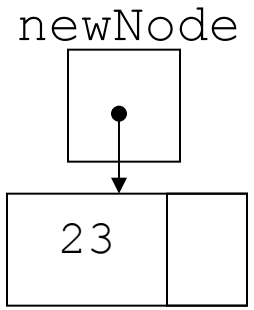
- Initialize the contents of the node:

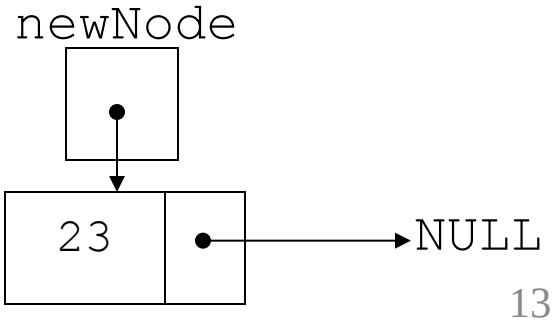
```
newNode->value = num;
```

- Set the pointer field to `NULL`:

```
newNode->next = NULL;
```



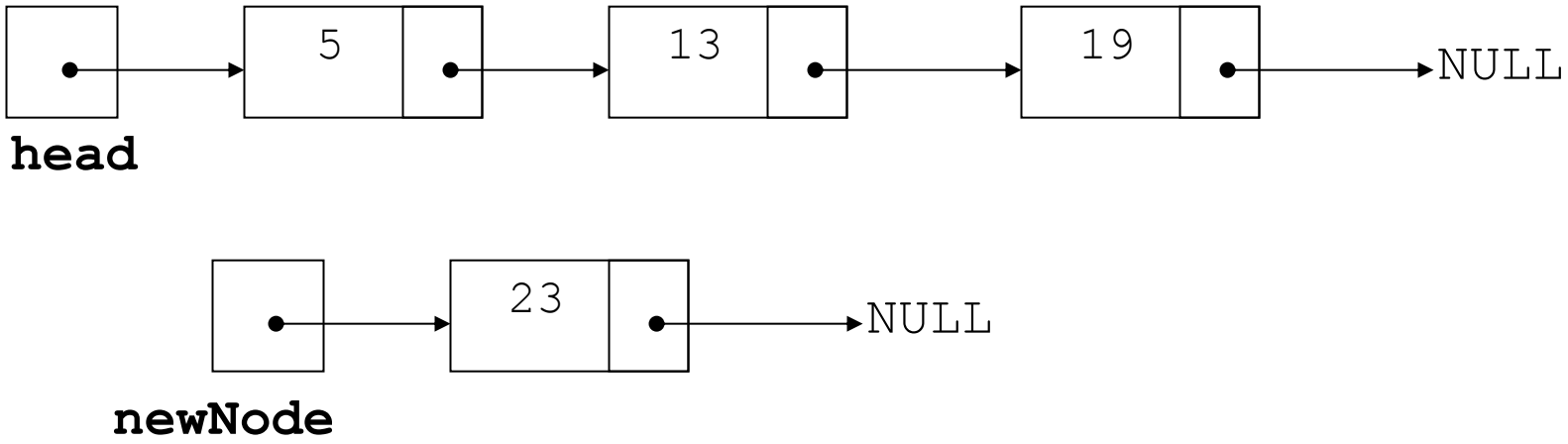




Add a Node at Beginning of List

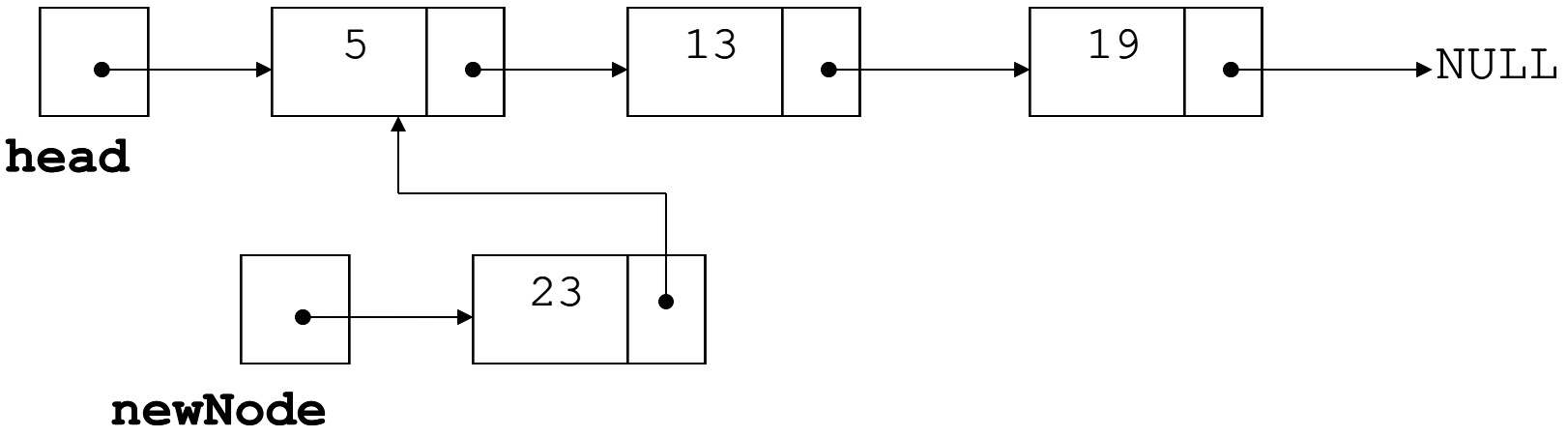
- Basic process:
  - Create the new node (as already described)
  - Add node to the beginning of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - set the **`next`** pointer of new node to point to the previously existing *first* node of the list
      - set list **`head`** to point to the new node.

Existing List and New Node



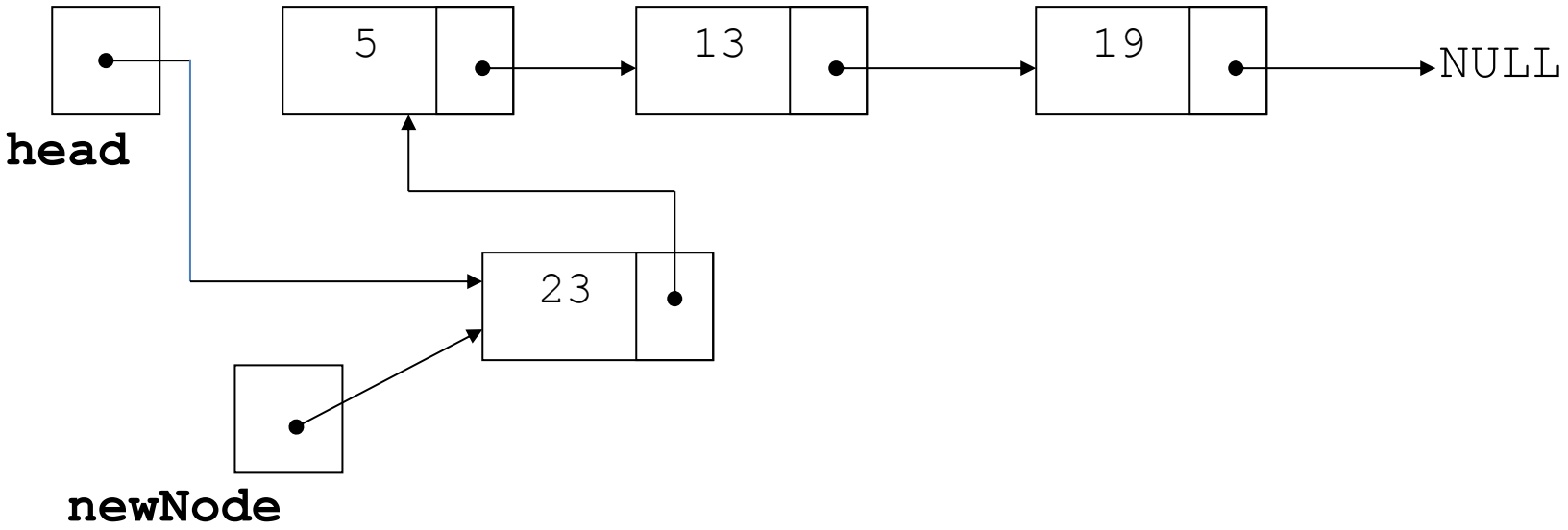
New node created

Link New Node to First Node in List



New node linked to first node in the list

Point List Head to the New Node

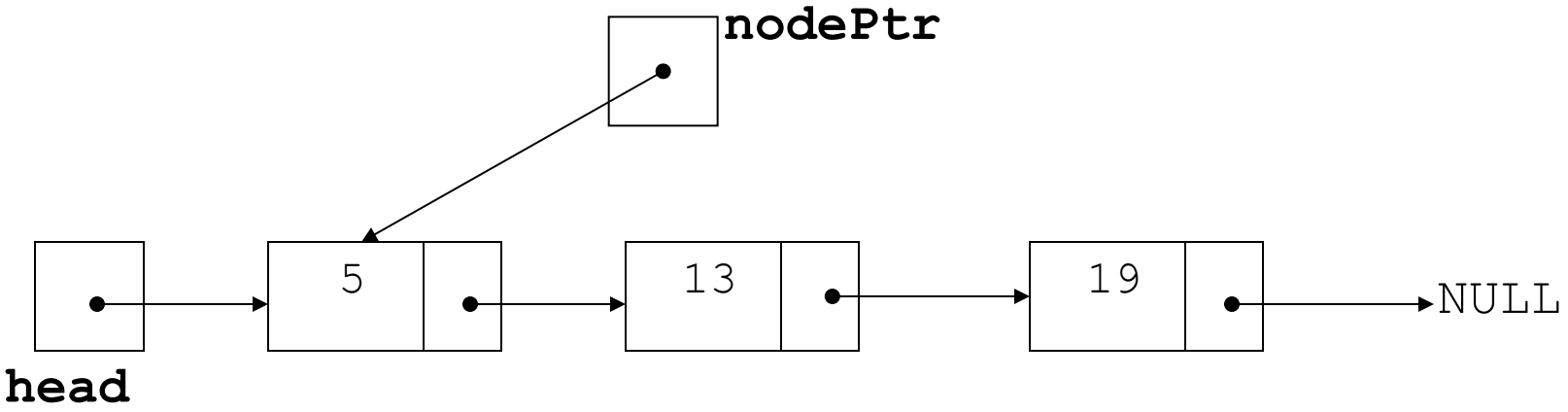


New node is now the first node in the list

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not `NULL`
    - Process data (perform some task)
    - Go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

Traversing a Linked List



The `nodePtr` variable points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to `NULL`, and the list traversal stops.
At each step along the way, some useful task is performed.

Sample code: the **NumberList** class

- Available on Moodle:

  `NumberListDemo.cpp`

  `NumberList.h`

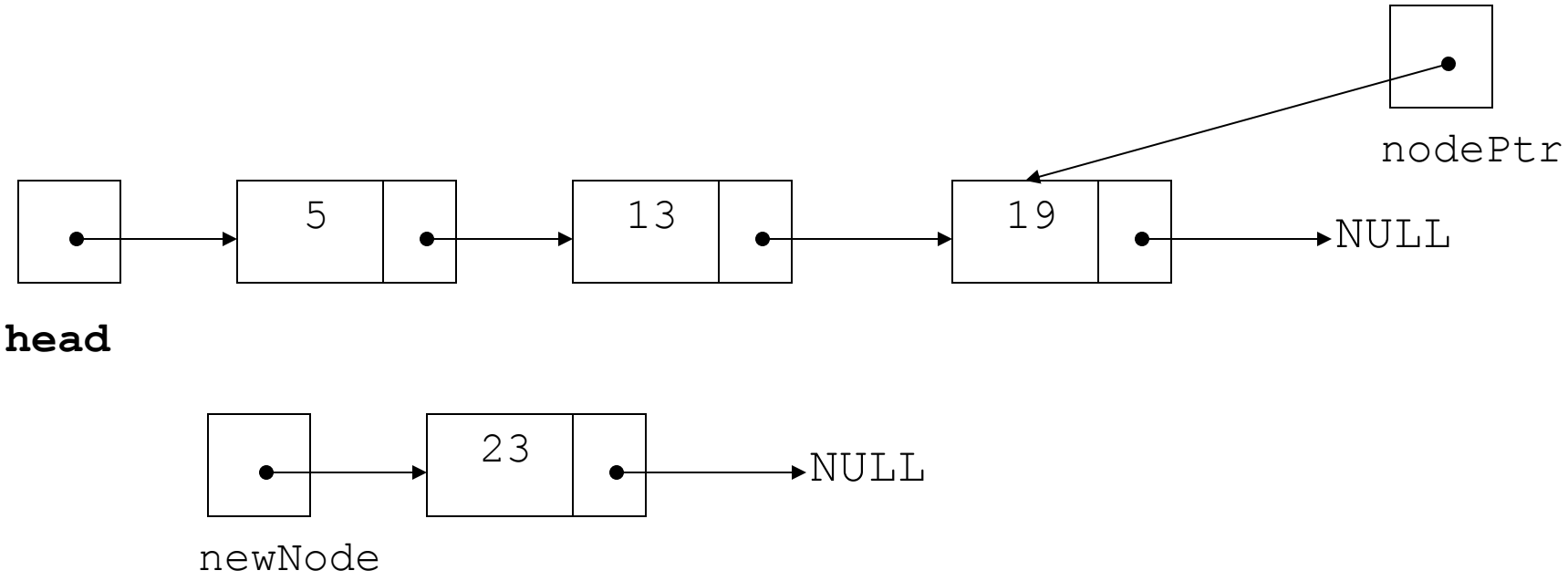
  `NumberList.cpp`

- The **NumberList** class implements a linked list containing floating point numbers.

Appending a Node

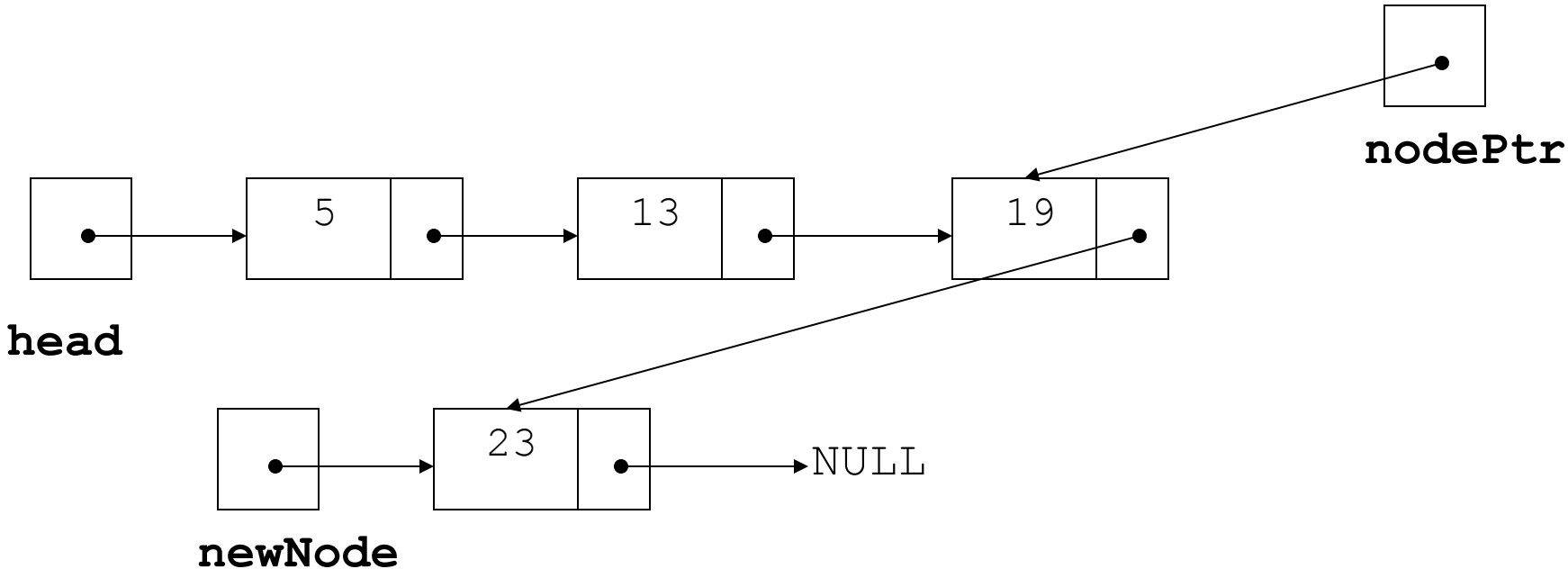
- Sometimes, the program may need to add a node to the end of the list.
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node (1)



New node created, end of list located

Appending a Node (2)



New node added to end of list

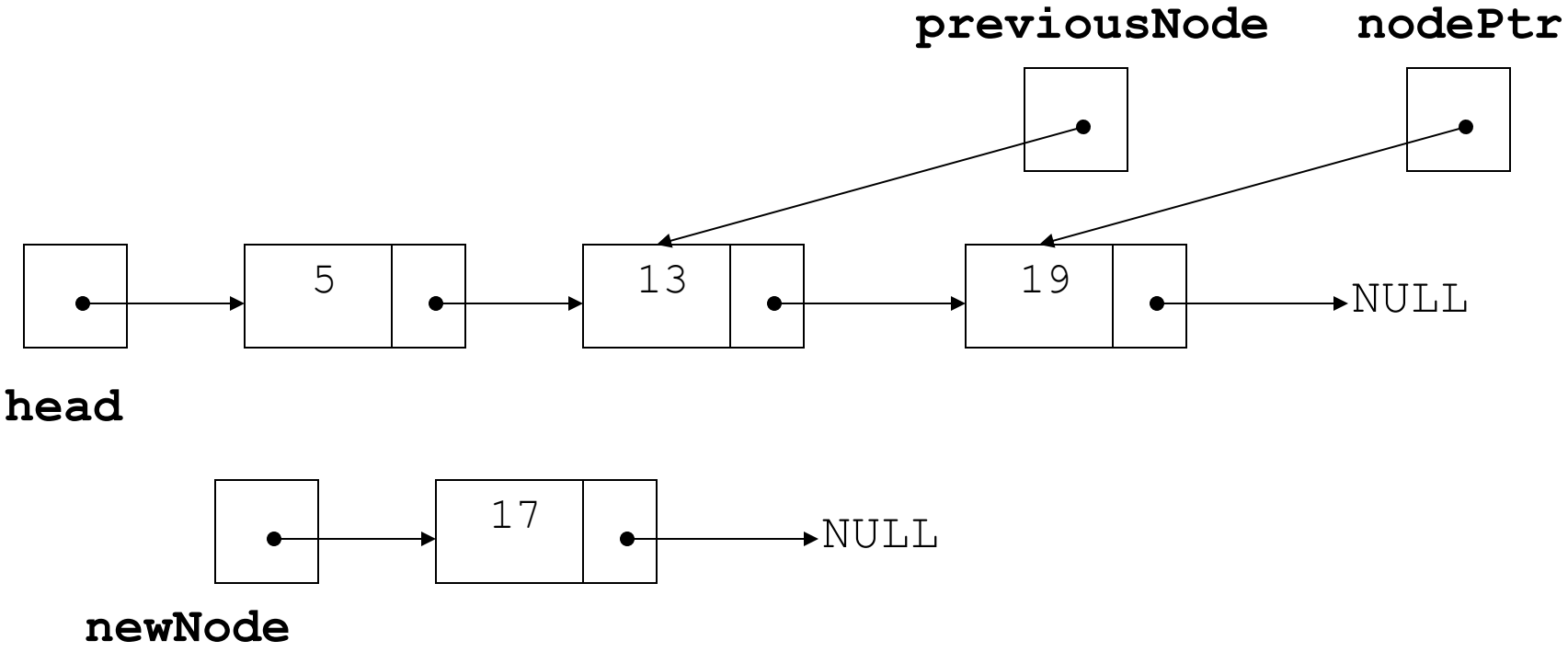
CAUTION about Appending a Node

- If your intention is to maintain a list which is always “sorted”, then a misused Append operation could violate that policy.
- In such a case, using Append could be appropriate if you know that the input data is *already* sorted.

Inserting a Node into a Linked List

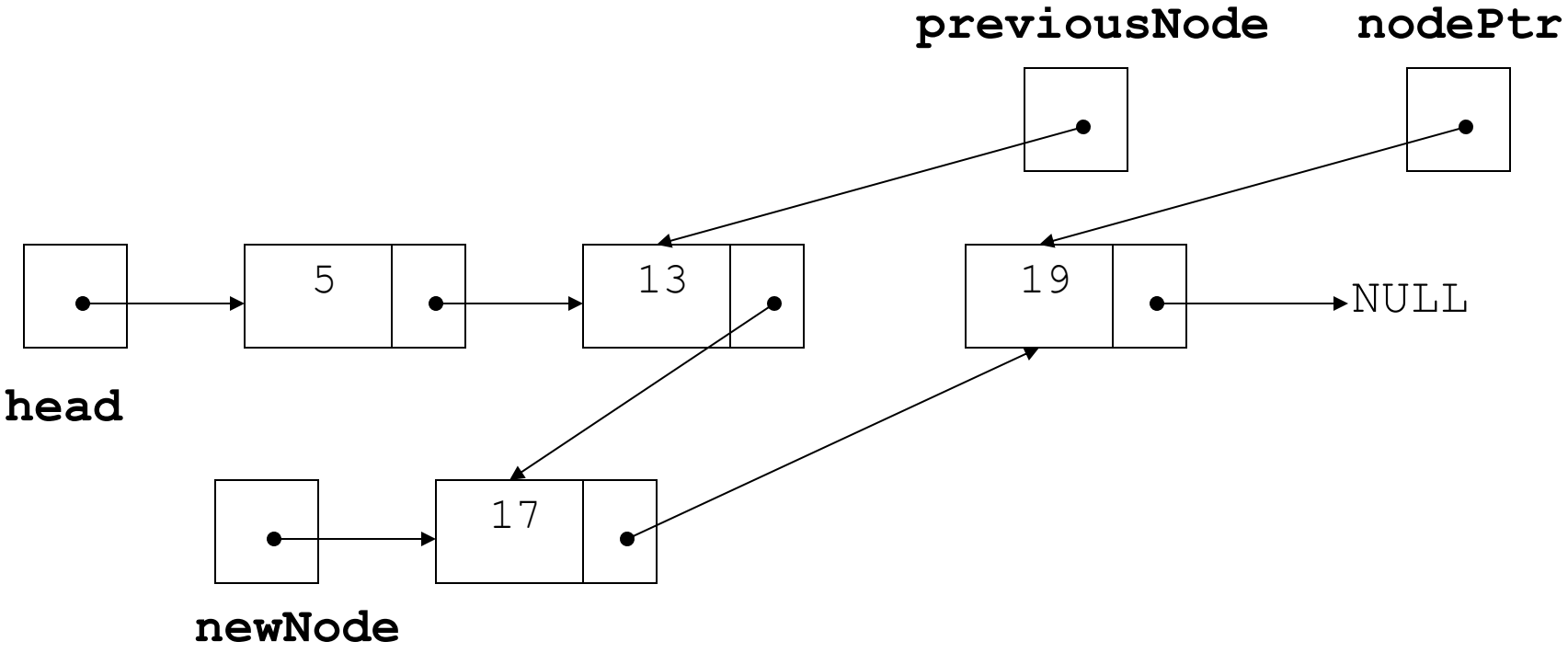
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

Inserting a Node (1)



New node created, correct position located

Inserting a Node (2)

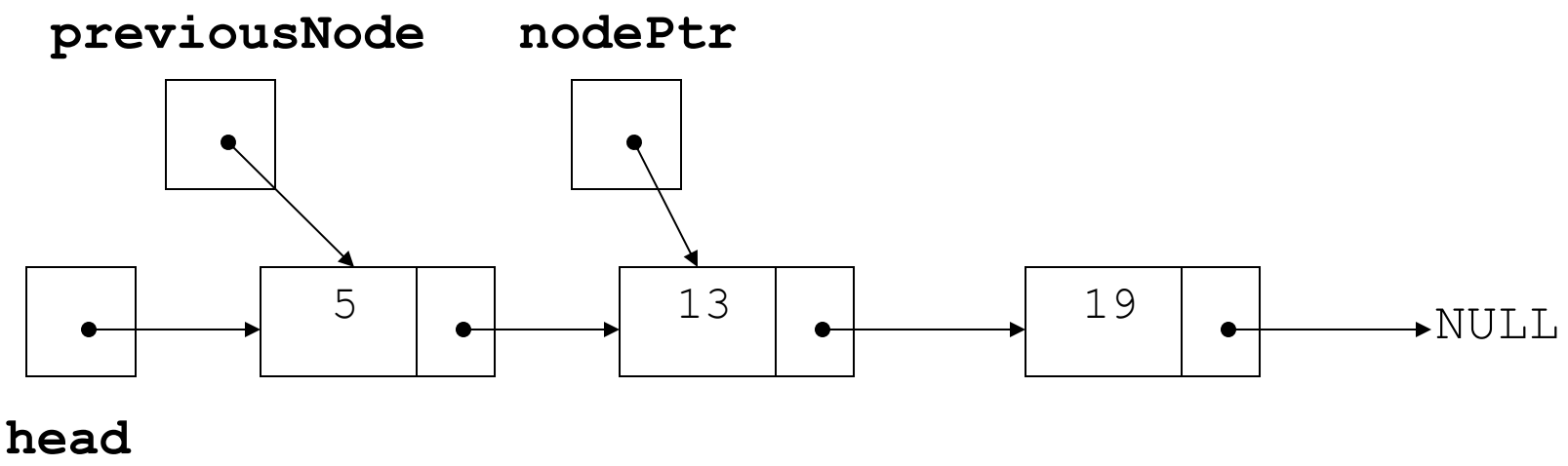


New node inserted in correct order in the linked list

Deleting a Node

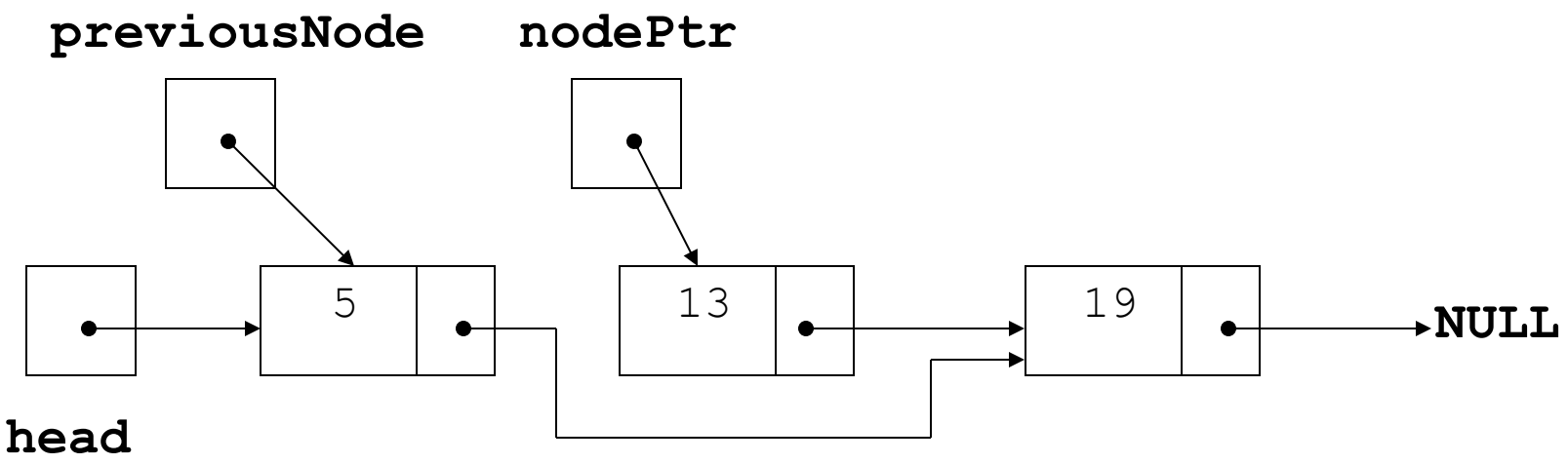
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Prepare for Node Deletion



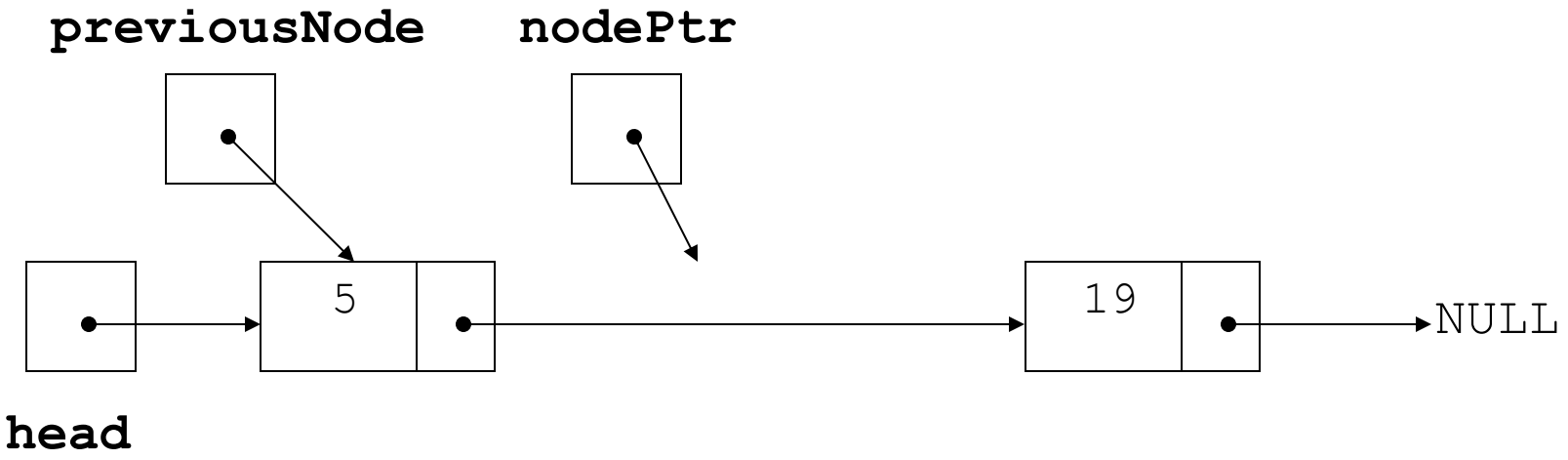
First, we need to locate the node we wish to delete, but also maintain a pointer to the “previous” node.

Unlink the Node



Adjust pointer “around” the node to be deleted.

Delete the Node



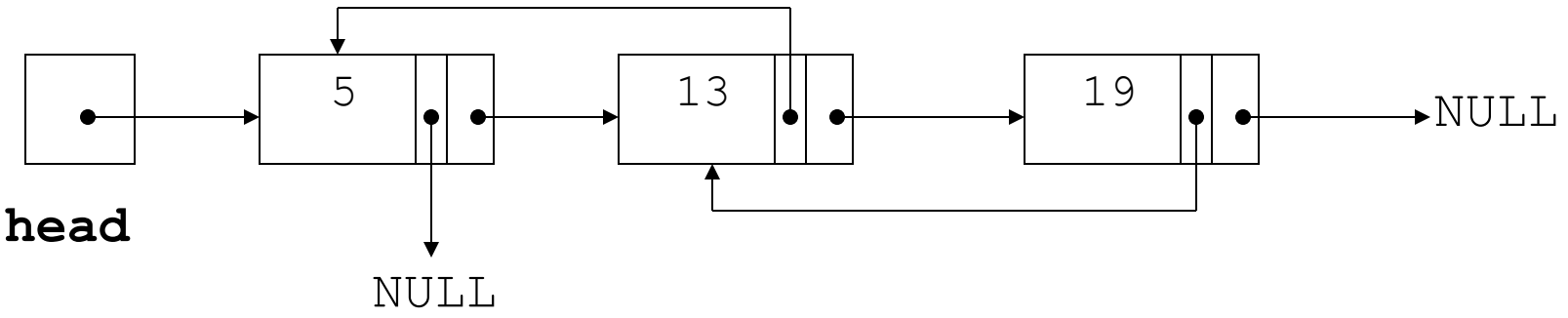
Linked list after deleting the node containing `13`

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `NULL`

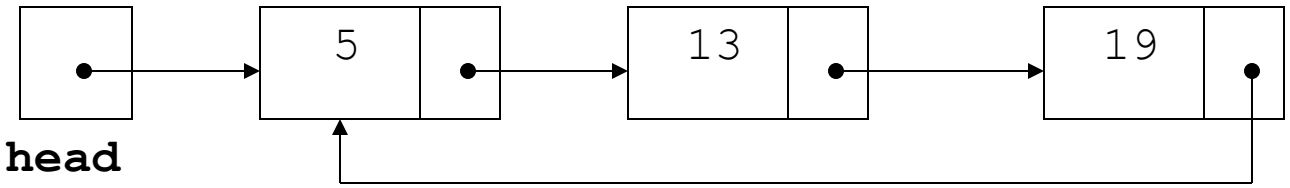
A Variation of the Linked List

- Some linked lists have forward and backward links:
  - Doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list

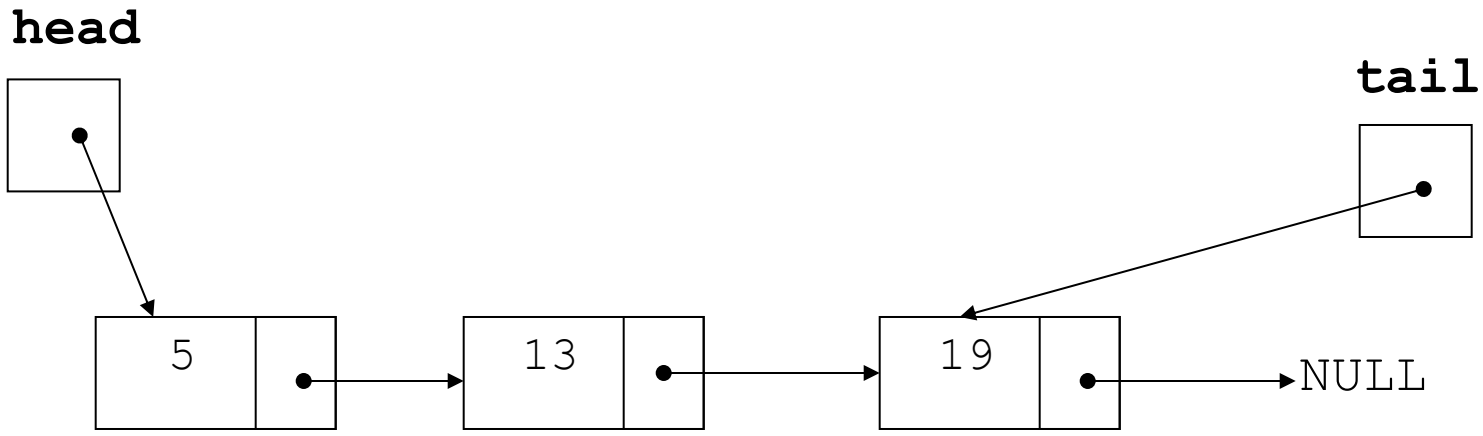


“Circular” Linked List

- Some linked lists have a pointer from the “tail” back to the “head”:
  - Circular linked list: the last node in the list points back to the first node in the list, not to `NULL`



Special “Tail” node



The STL `list` Container

- Template for a *doubly linked* list.
- Member functions for
  - locating beginning, end of list: `front, back, end`
  - adding elements to the list: `insert, merge, push_back, push_front`
  - removing elements from the list: `erase, pop_back, pop_front, unique`
- See Table 18-2 (pages 1155-1157) for a partial list of member functions.
- The STL also includes a *singly linked* list container:

`forward_list`

Examples of Linked Lists

- The textbook develops a class called **NumberList**: a list of nodes containing **double** values.
- In the lab exercise, we are developing a class called **IntList**: a list of nodes containing **int** values.
- It would be nice to create a re-usable class that can create a linked list of *any* type.

A Linked List Template

- The **LinkedList.h** file (in the sample code) defines a class template that can be used to create a linked list of any data type.
- The sample code is available on Moodle:

  Ch18_LinkedListTemplate1_sample_code.