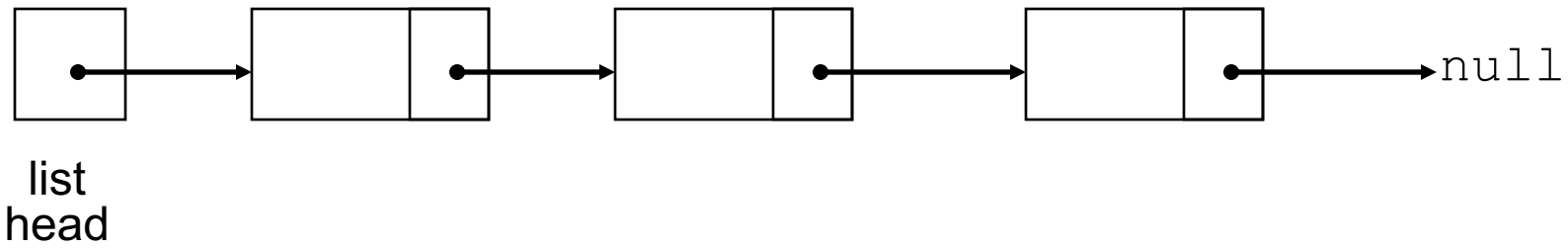
Introduction to the Linked List ADT

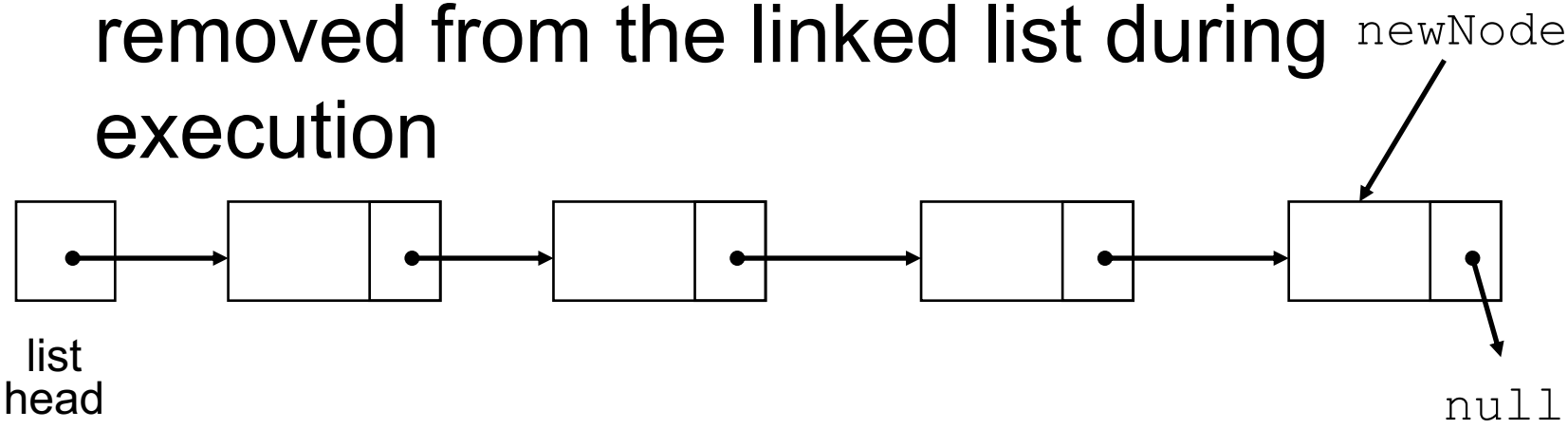
Introduction to the Linked List ADT

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



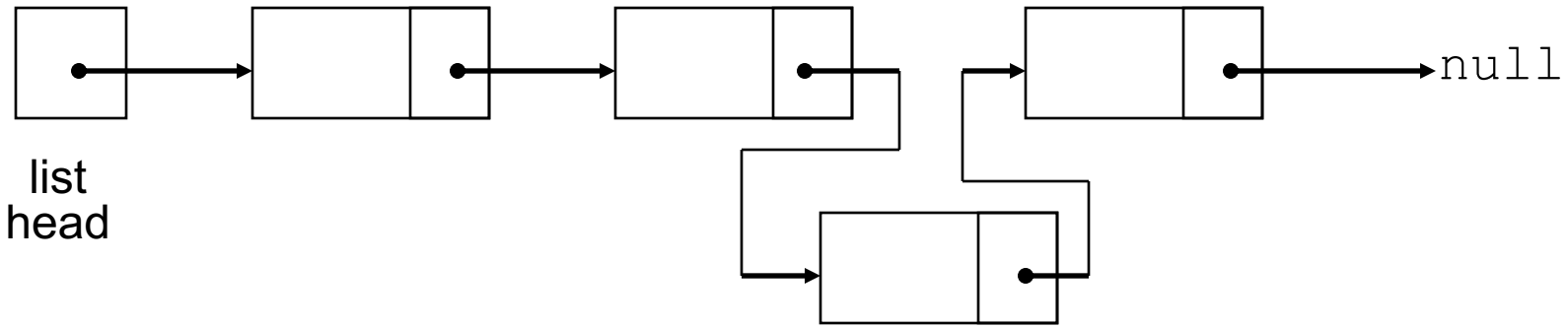
Introduction to the Linked List ADT

- References may be addresses or array indices
- Data structures can be added to or removed from the linked list during execution



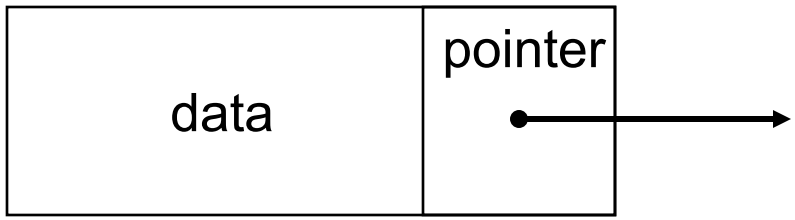
Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



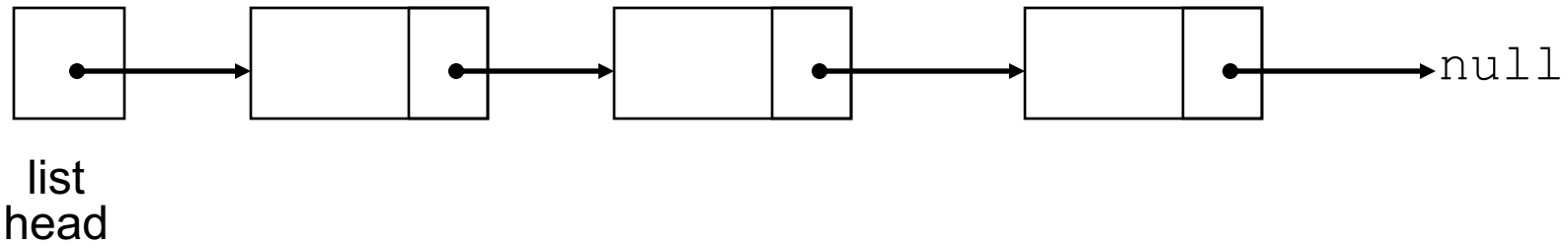
Node Organization

- 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

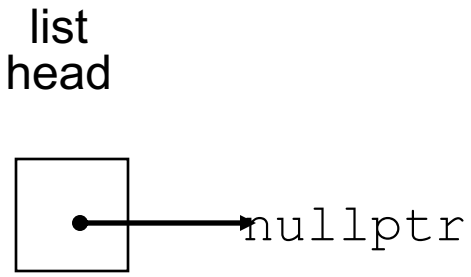
- Linked list contains 0 or more nodes:



- Has a list head to point to first node
- Last node points to `null(address 0)`

Empty List

- If a list currently contains 0 nodes, it is the empty list
- In this case the list head points to `nullptr`



Declaring a Node

- Declare a node:

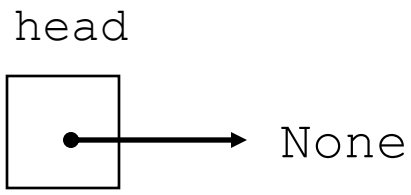
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
head = None
```

- Head pointer initialized to `None` to indicate an empty list



The Null Pointer

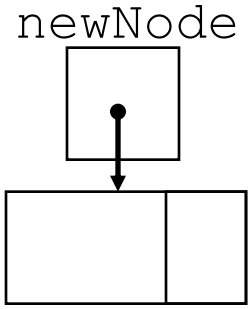
- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

Linked List Operations

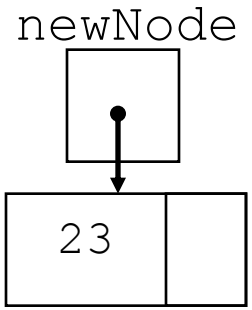
- Basic operations:
  - **append** a node to the end of the list
  - **insert** a node within the list
  - **traverse** the linked list
  - **delete** a node
  - **delete/destroy** the list

Create a New Node

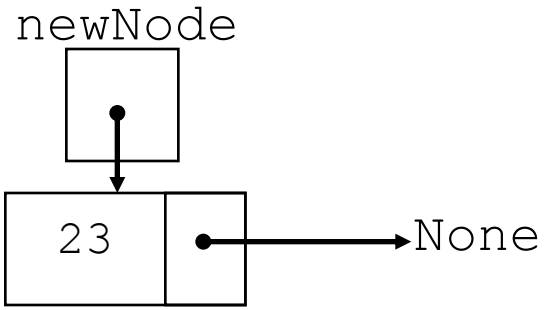
- Allocate memory for the new node:



- Initialize the contents of the node:



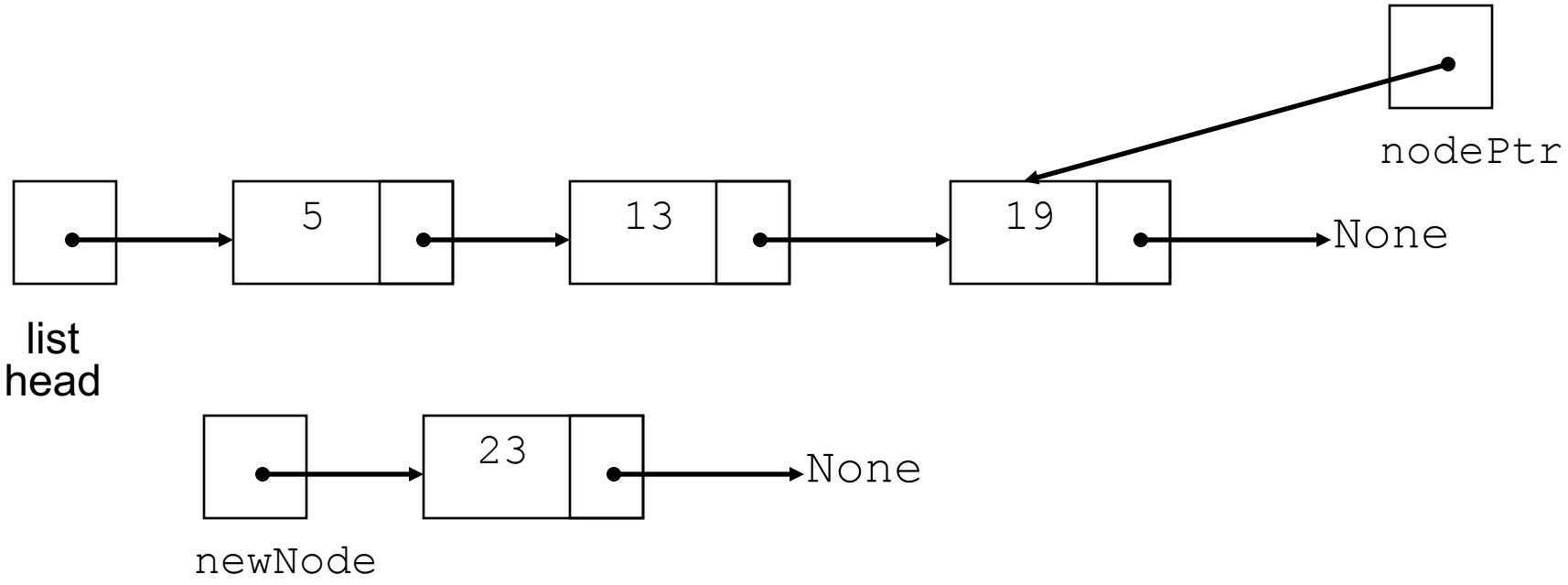
- Set the pointer field to `None`:



Appending a Node

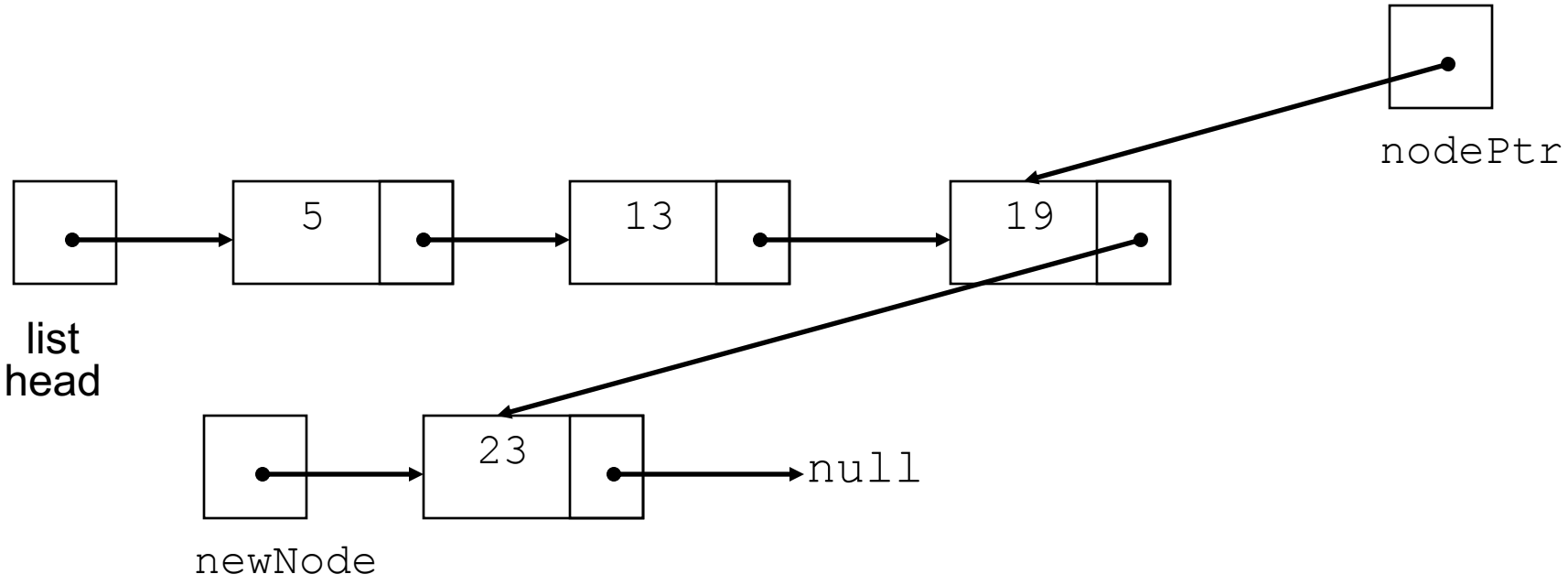
- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



New node created, end of list located

Appending a Node



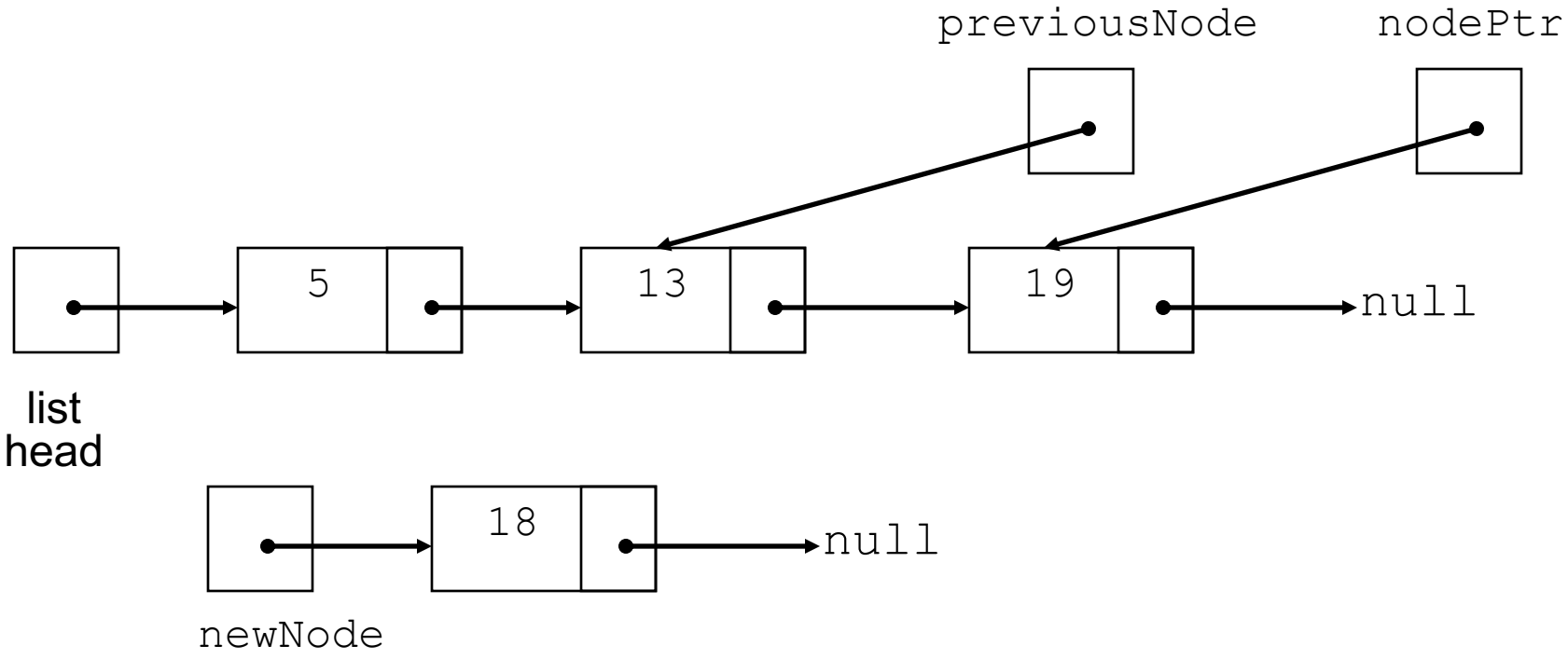
New node added to end of list

Python code for Appending a Node

Inserting a Node into a Linked List

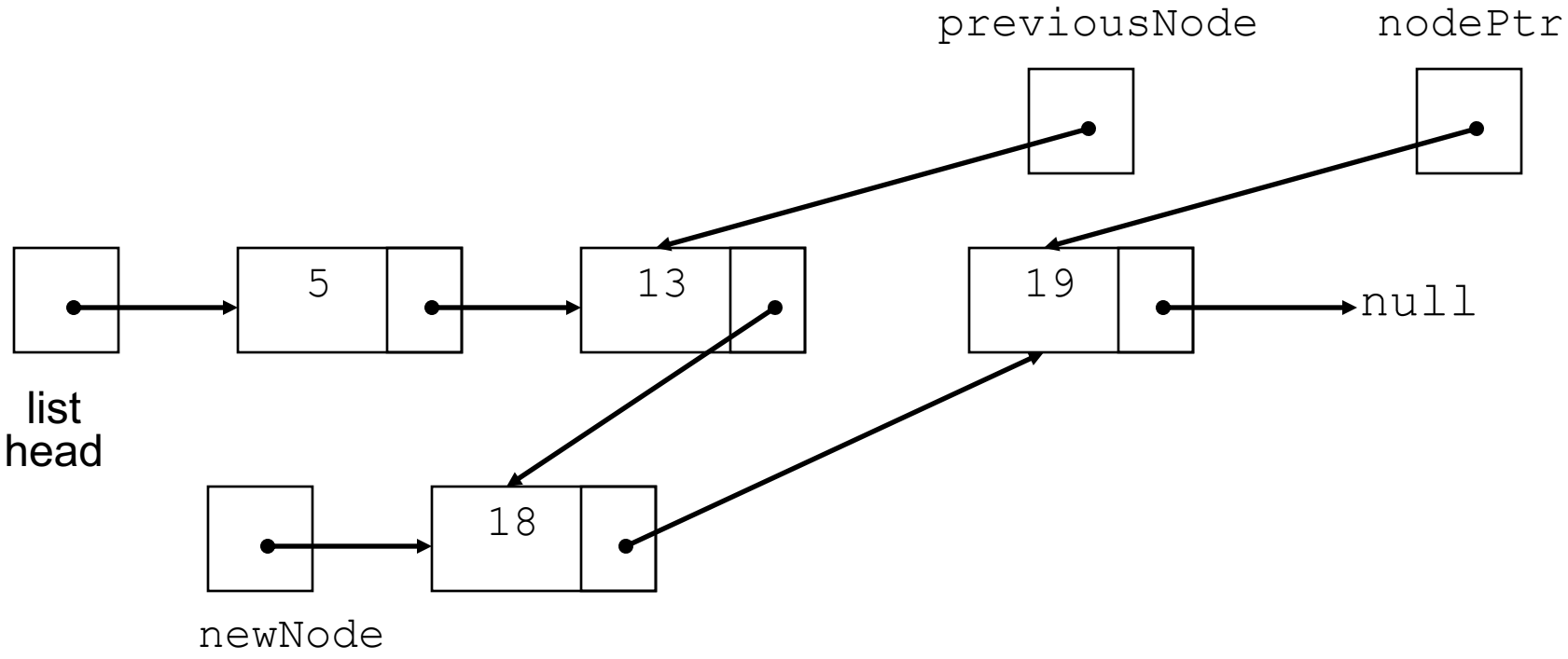
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed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

Inserting a Node into a Linked List



New node created, correct position located

Inserting a Node into a Linked List



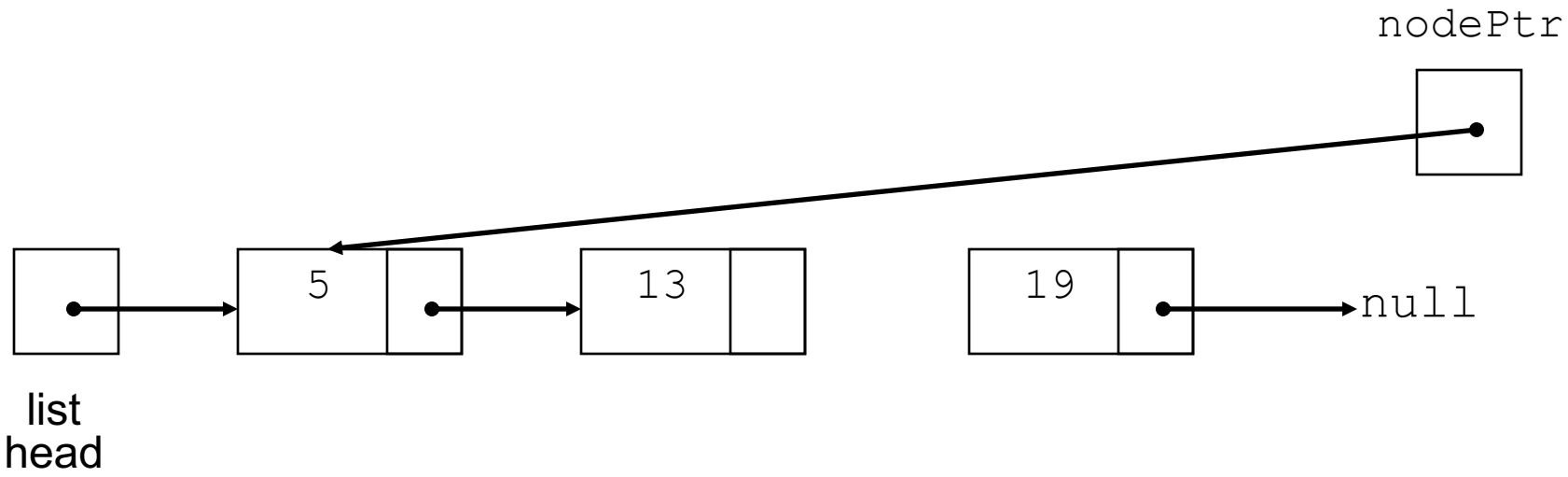
New node inserted in order in the linked list

Python code for Inserting a Node

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 a null pointer
    - process data
    - 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

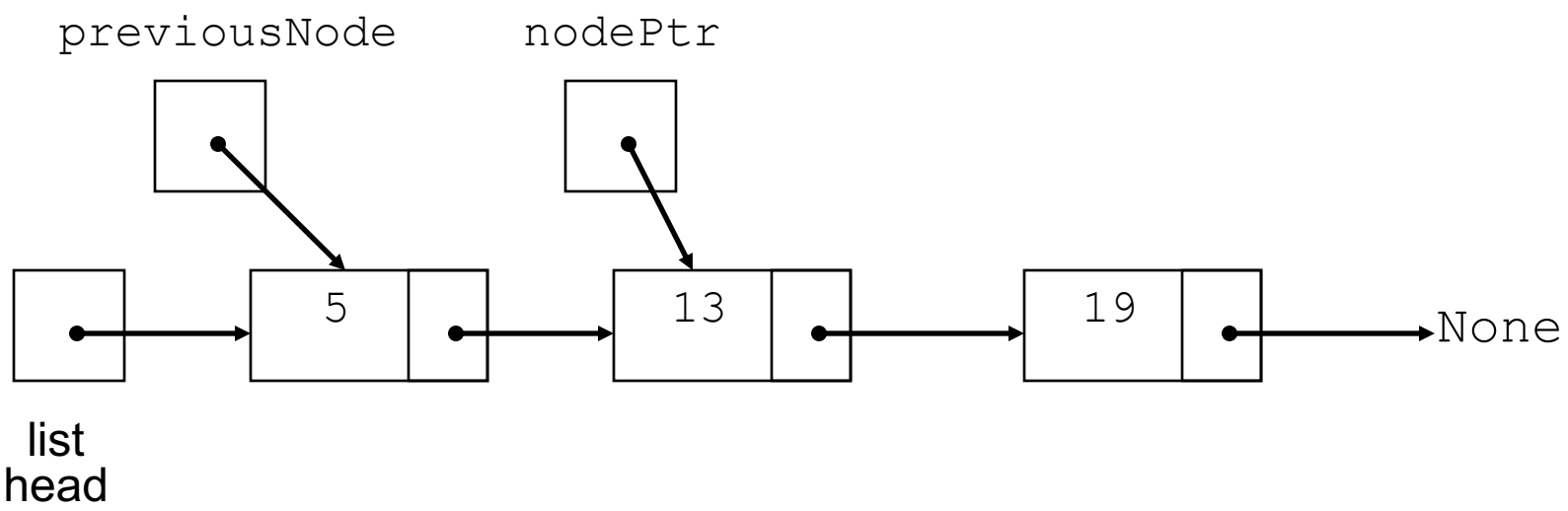


`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to the null pointer, and the list traversal stops

Deleting a Node

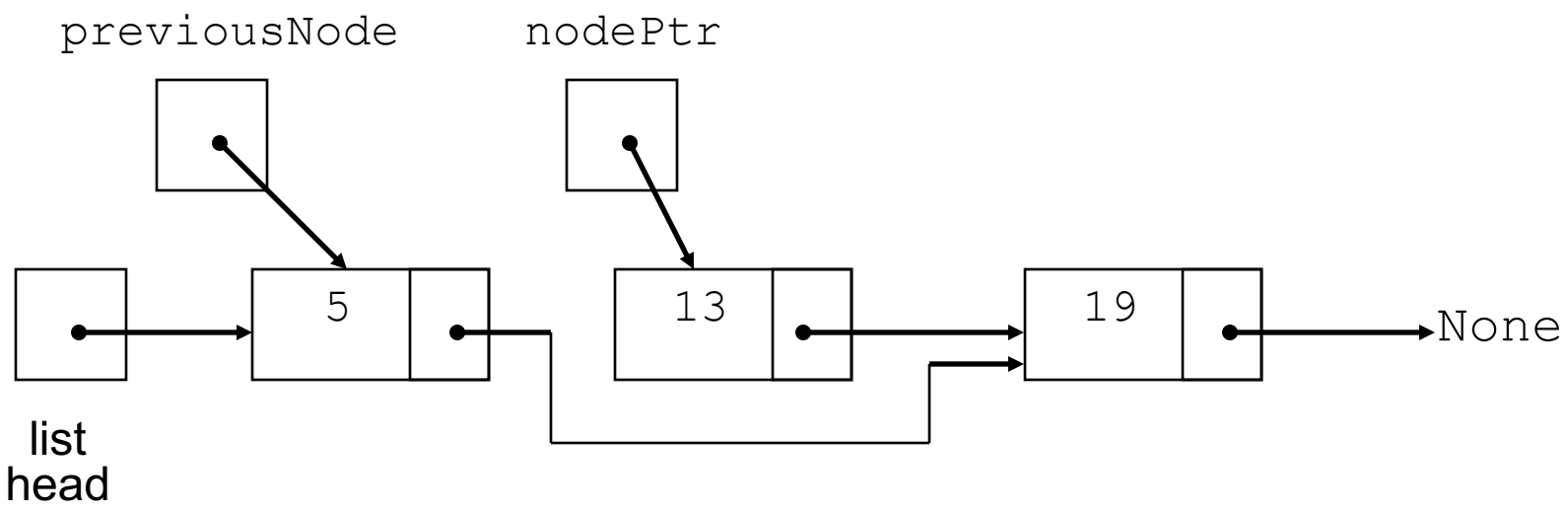
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node



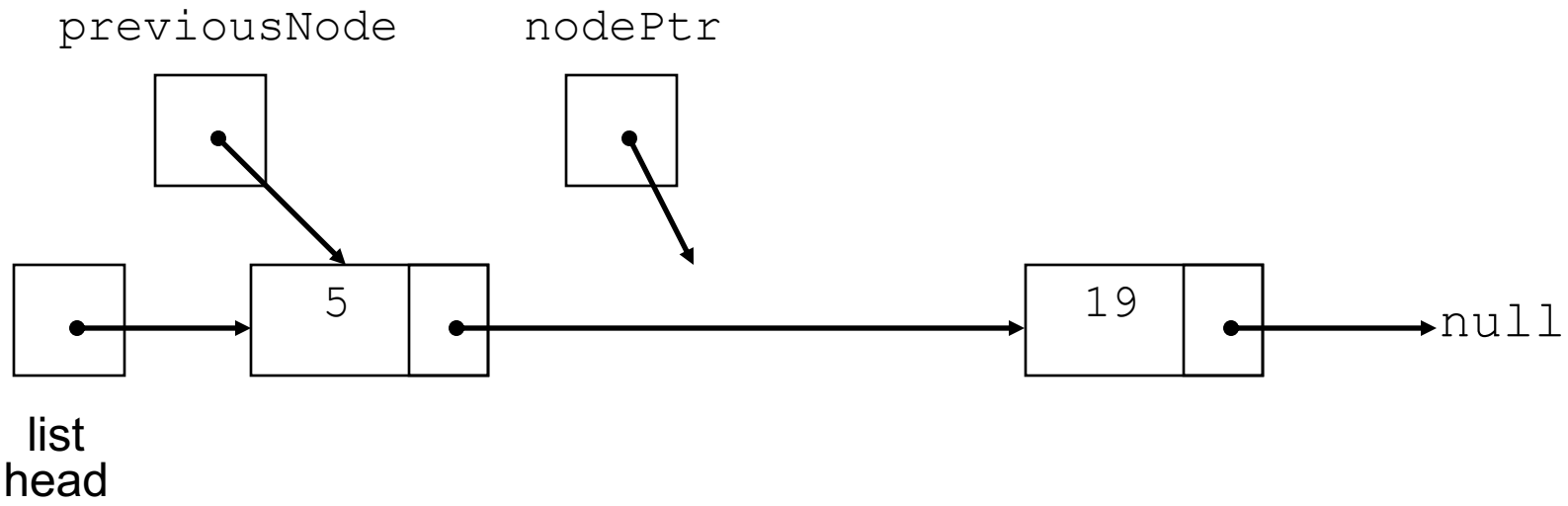
Locating the node containing `13`

Deleting a Node



Adjusting pointer around the node to be deleted

Deleting a Node



Linked list after deleting the node containing `13`

Python code for Deleting a Node

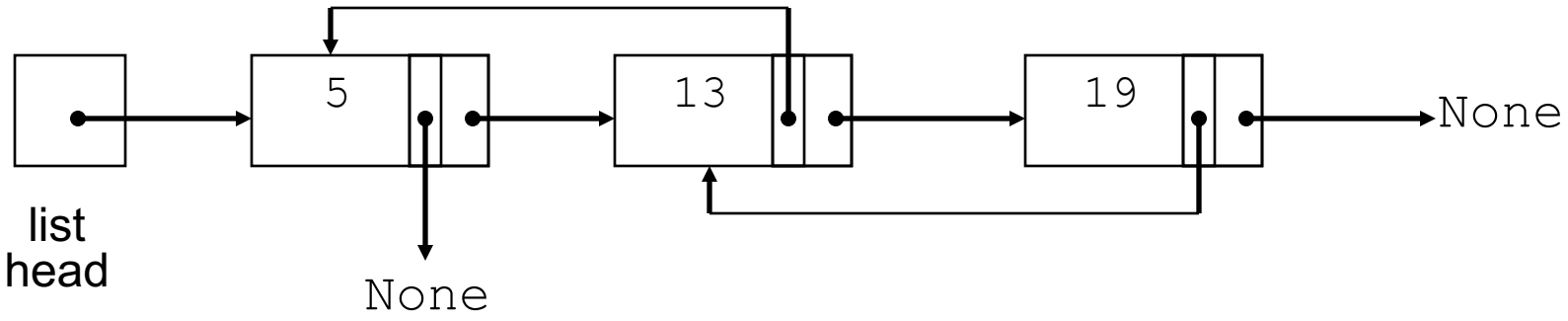
Destroying a Linked List

- 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 `None`

Variations of the Linked List

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

The last node in the list points back to the first node in the list, not to the None pointer