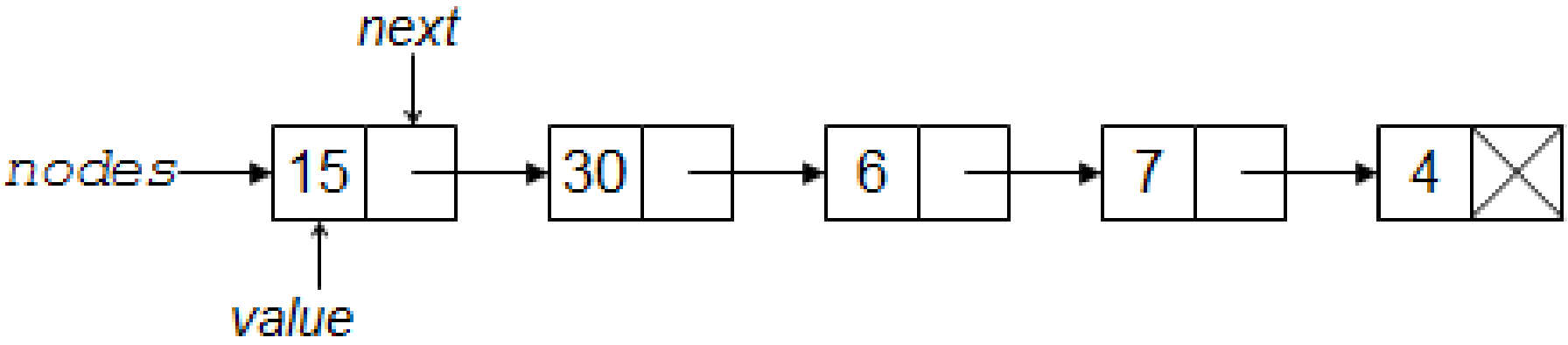
Chapter 9: Linked Lists



Topics

17.1 Introduction to the Linked List ADT

17.2 Linked List Operations

17.3 A Linked List Template

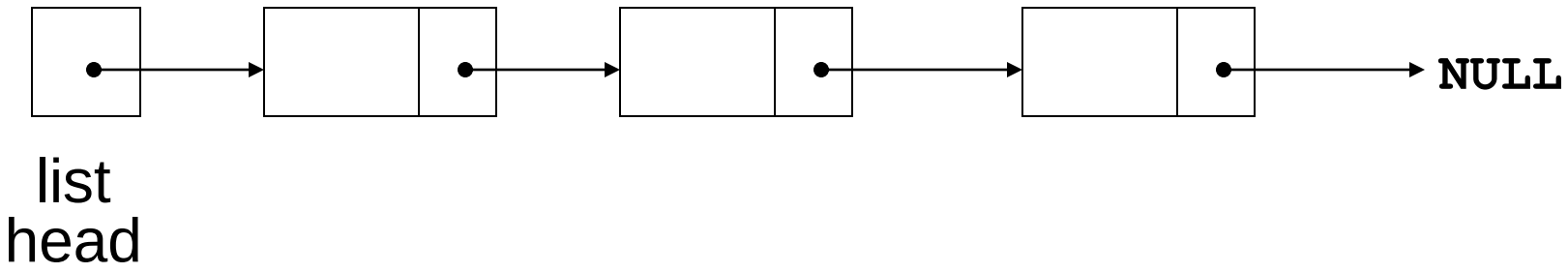
17.4 Recursive Linked List Operations

17.5 Variations of the Linked List

17.6 The STL **`list`** Container

17.1 Introduction to the Linked List ADT

- Linked list: a sequence of data structures (nodes) with each node containing a pointer to its successor
- The last node in the list has its successor pointer set to NULL

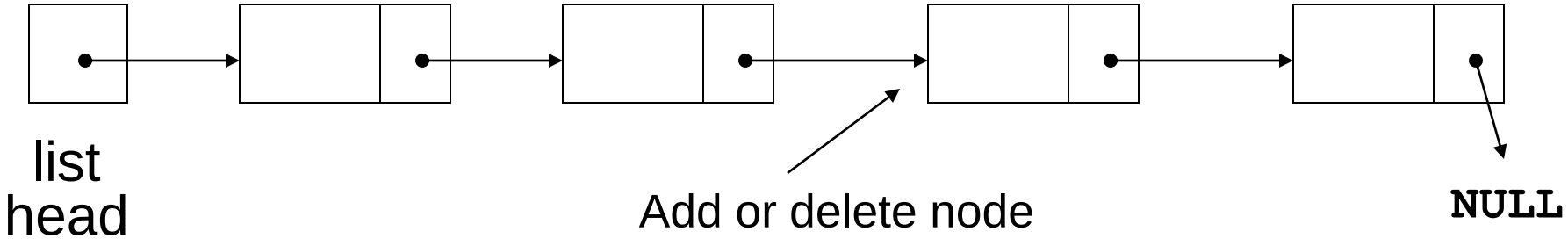


Linked List Terminology

- The node at the beginning is called the head of the list
- The entire list is identified by the pointer to the head node. This pointer is called the list head.

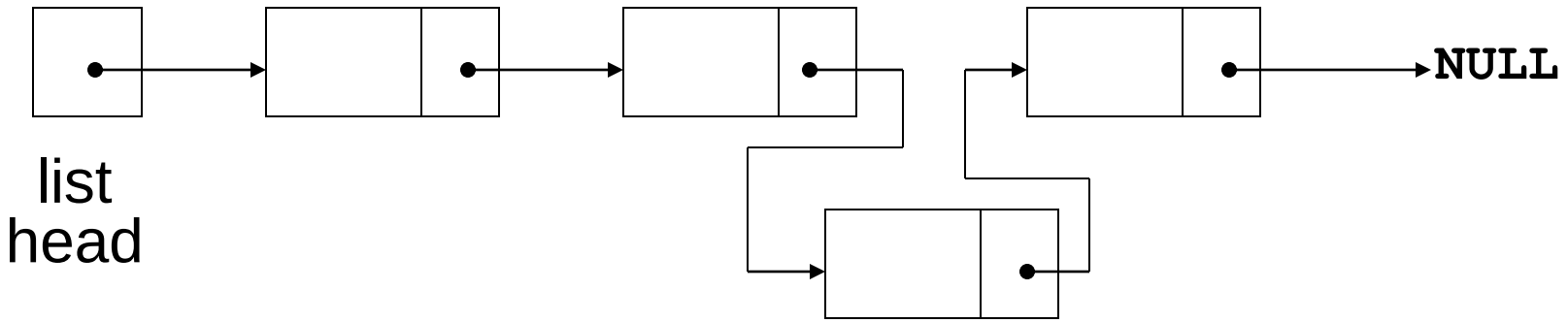
Linked Lists

- Nodes can be added or removed from the linked list during execution
- Addition or removal of nodes can take place at beginning, end, or middle of the list



Linked Lists vs. Arrays and Vectors

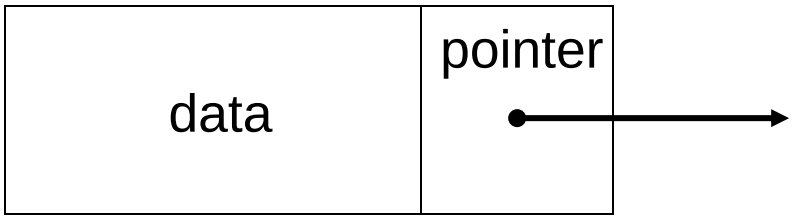
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Unlike vectors, insertion or removal of a node in the middle of the list is very efficient



Node Organization

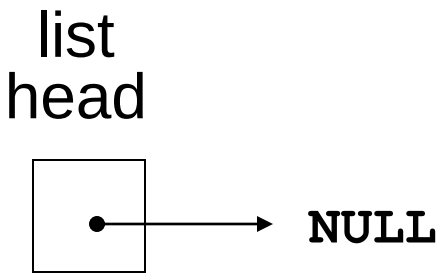
A node contains:

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



Empty List

- A list with no nodes is called the empty list
- In this case the list head is set to **NULL**



C++ Implementation

Implementation of nodes requires a structure containing a pointer to a structure of the same type (a self-referential data structure):

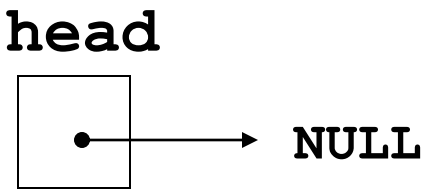
```
struct ListNode
{
   int data;
   ListNode *next;
};
```

Creating an Empty List

- Define a pointer for the head of the list:

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

- Head pointer is initialized to **NULL** to indicate that this is an empty list



C++ Implementation

Nodes can be equipped with constructors:

```
struct ListNode
{
  int data;
  ListNode *next;
  ListNode(int d, ListNode* p=NULL)
   {data = d; next = p;}
};
```

Building a List from a File of Numbers

```
ListNode *head = NULL;

int val;

while (inFile >> val)

{

  // add new nodes at the head

  head = new ListNode(val, head);



  // Note that assignment is right-to-

  // left.  The present value of head

  // is used when the node is created,

  // then the address of the new node

  // is assigned to head.

};
```

Traversing a Linked List

- List traversals visit each node in a linked list to display contents, validate data, etc.
- Basic process of traversal:

  set a pointer to the head pointer

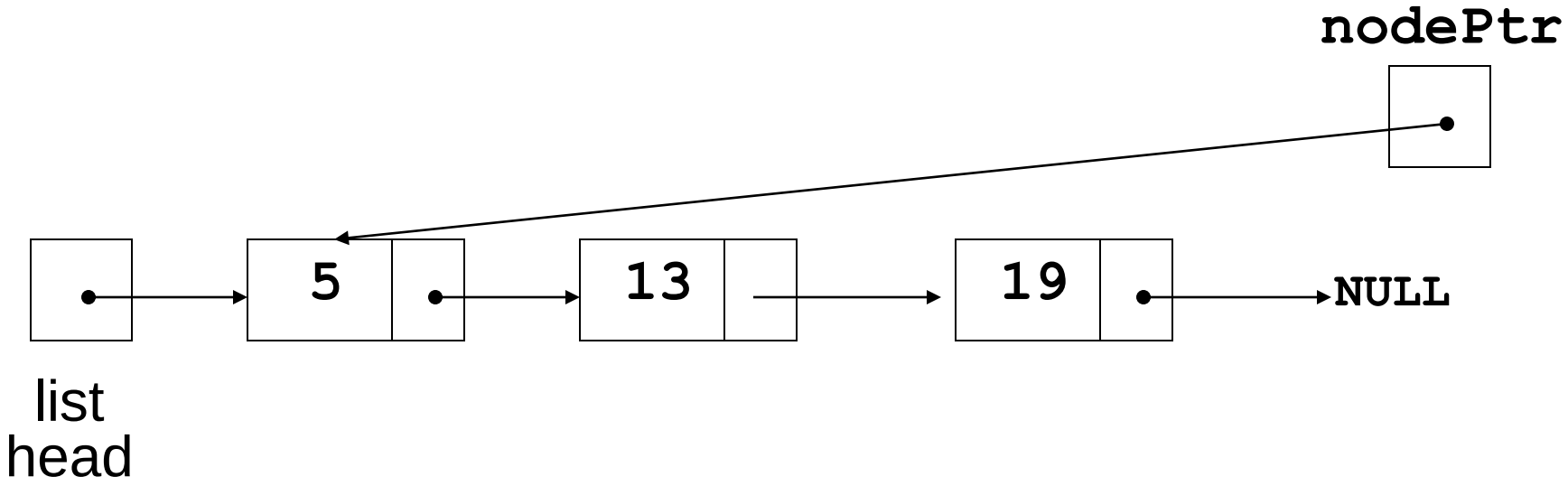
  while pointer is not ***NULL***

  process data

  set pointer to the successor of the current node

  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 **`NULL`**, and the list traversal stops

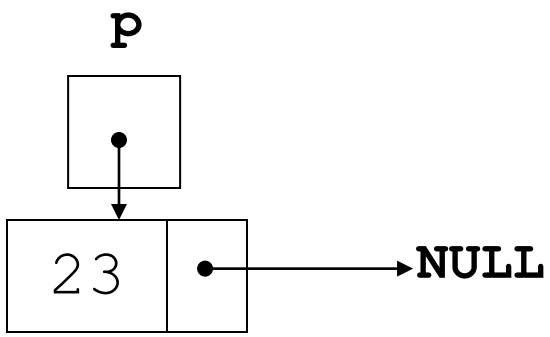
17.2 Linked List Operations

Basic operations:

- add a node to the list
- traverse the linked list
- Delete/remove a node from the list
- delete/destroy the list

Creating a Node

```
ListNode *p;
int num = 23;
p=new ListNode(num);
```



Adding an Item

To add an item to the end of the list:

- If the list is empty, set **`head`** to a new node containing the item

```
head = new ListNode(num);
```

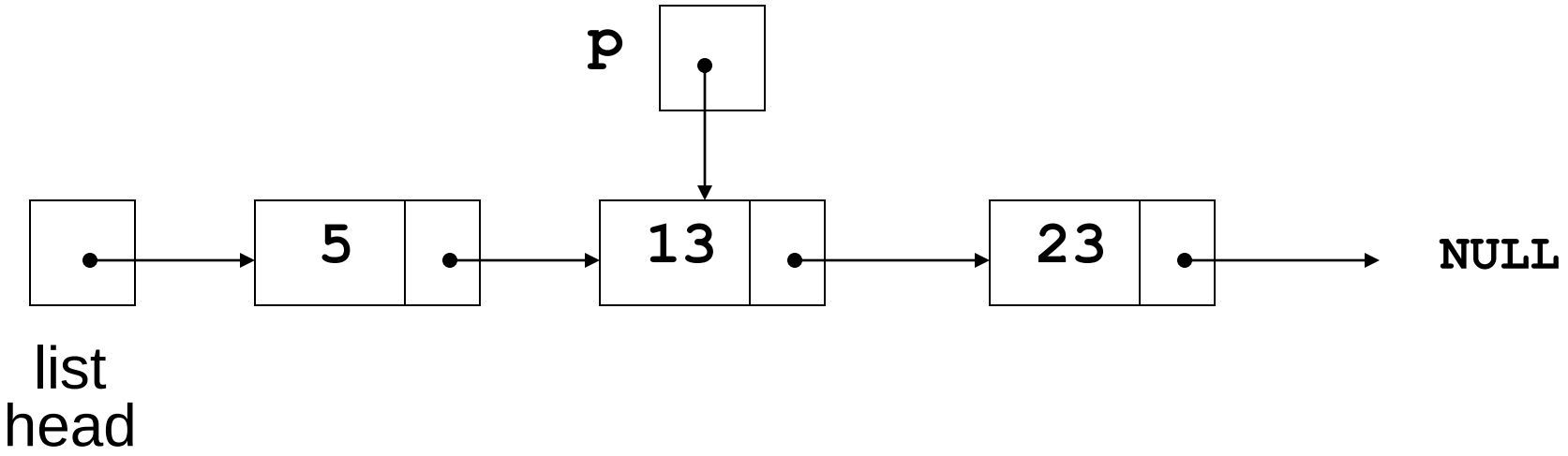
- If the list is not empty,
  - move a pointer **`p`** to the last node using

```
while(p->next != NULL)
    p = p->next;
```

  - then add a new node containing the item

```
p->next = new ListNode(num);
```

Adding an Item



List originally has nodes with 5 and 13.
`p` locates the last node, then a node with a new item, 23, is added

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
  - Free the node's memory
- Finally, set the list head to **`NULL`**

Maintaining a Sorted List

- You may want to keep the nodes in a linked list in order according to their data fields.
- In this case, adding new nodes to the end will not work.
- Here are two possibilities (ascending order):
  - The add point is at the head of the list (because the item at the head is already greater than the item being added, or because the list is empty
  - The add point is after an existing node in a non-empty list

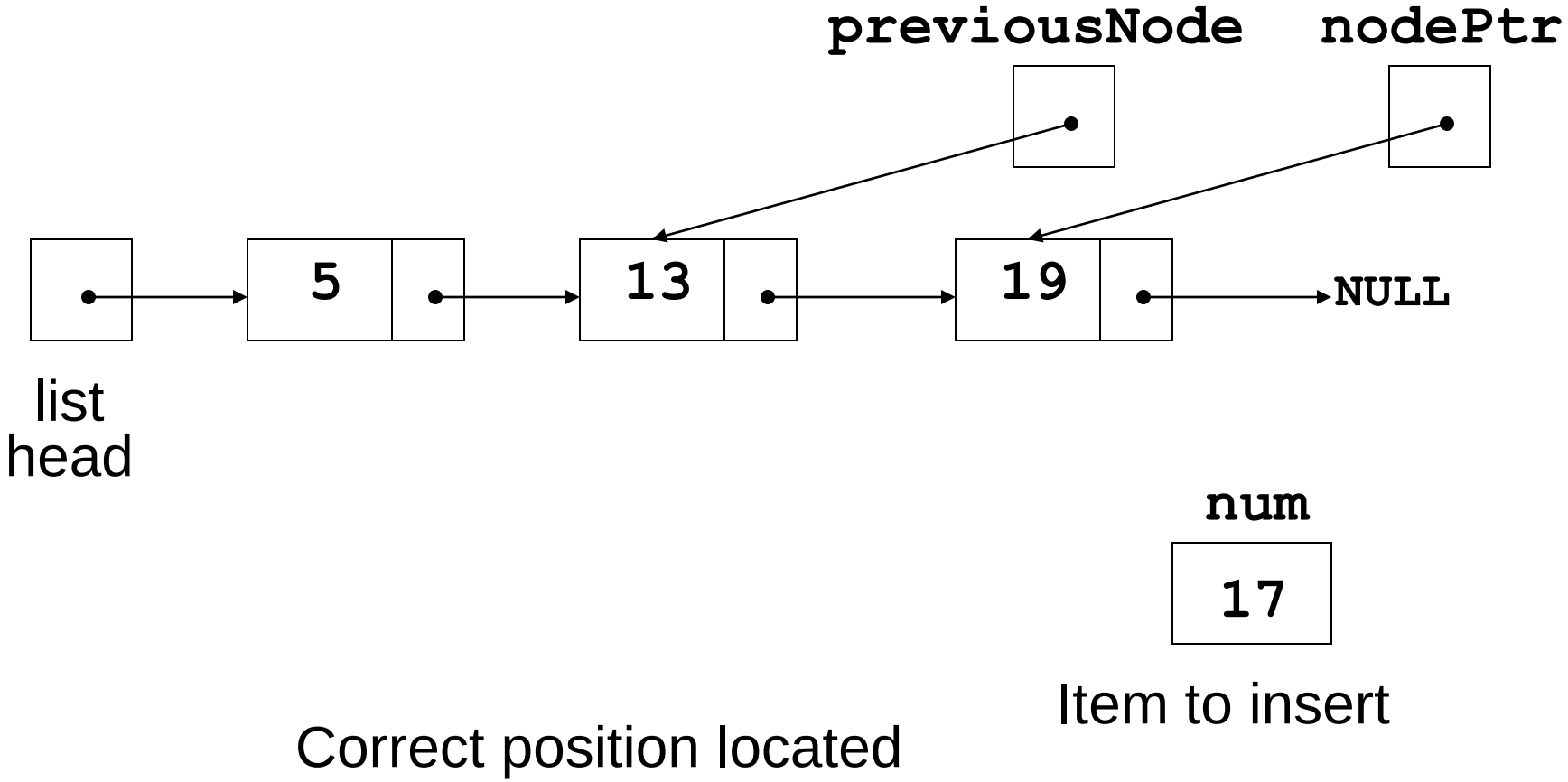
Adding a Node at the Head of a List

- Test to see if
  - head pointer is **`NULL`**, or
  - node value pointed at by head is greater than value to be inserted
- You must test in this order: the results are unpredictable if the second test is attempted on an empty list
- Create new node, set its next pointer to head, then point head to it

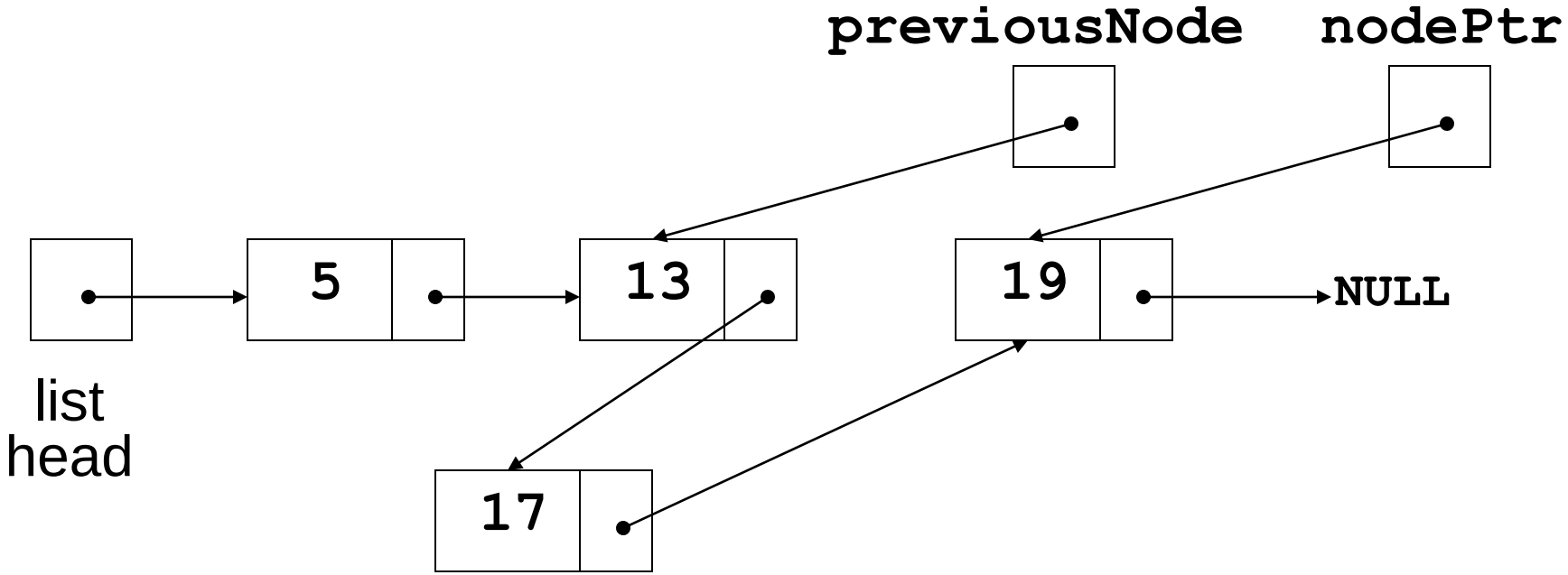
Inserting a Node after Head in a List

- This requires two pointers to traverse the list:
  - a pointer to locate the node with data value greater than that of node to be inserted
  - a pointer to 'trail behind' one node, to point to node before point of insertion
- The new node is inserted between the nodes pointed at by these pointers

Inserting a Node into a Linked List



Inserting a Node into a Linked List



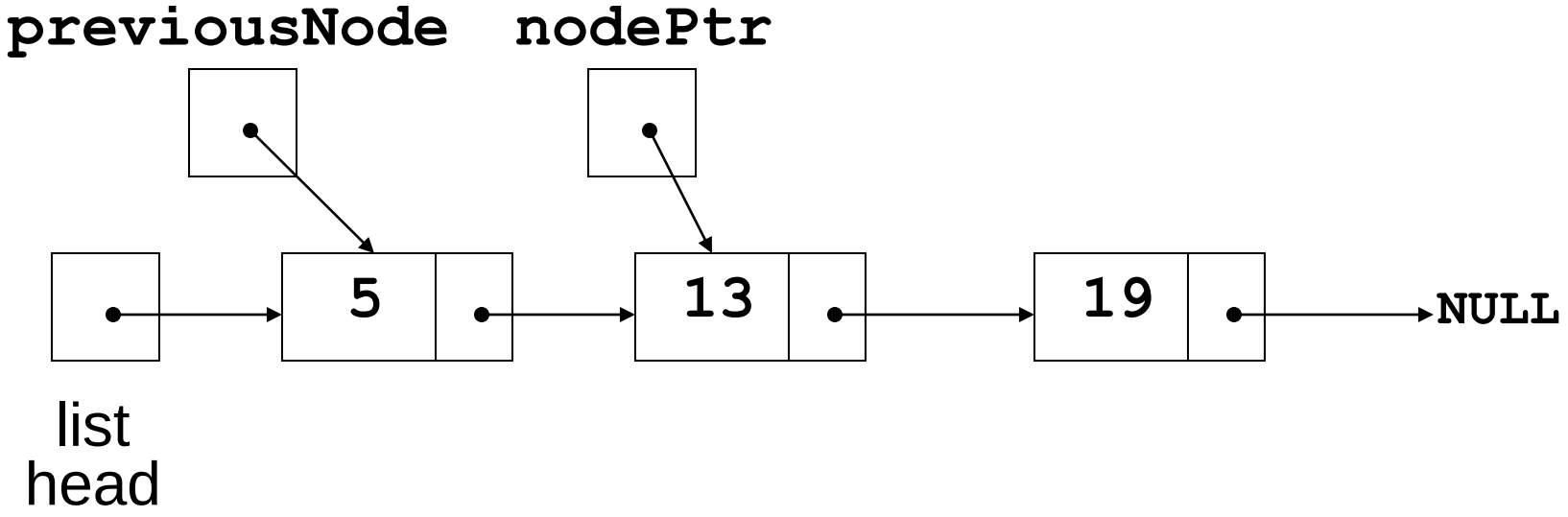
New node created and inserted in order in the linked list

Removing an Element

- Used to remove a node from a linked list
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

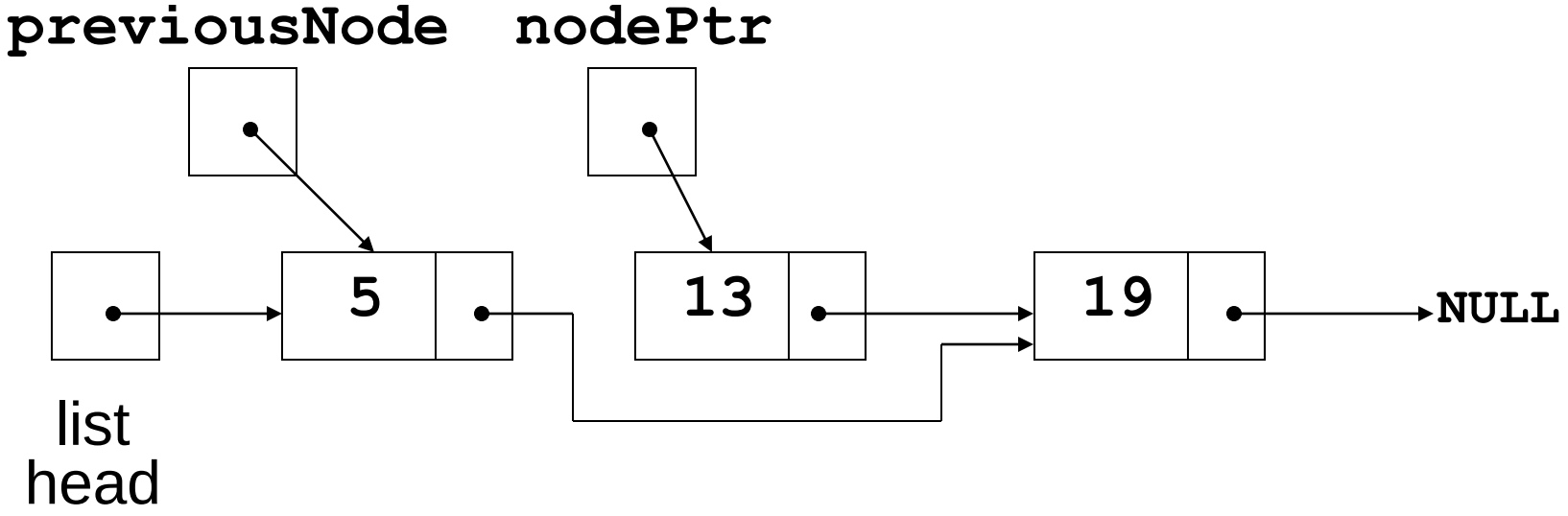
Deleting a Node

Contents of node to be deleted: 13



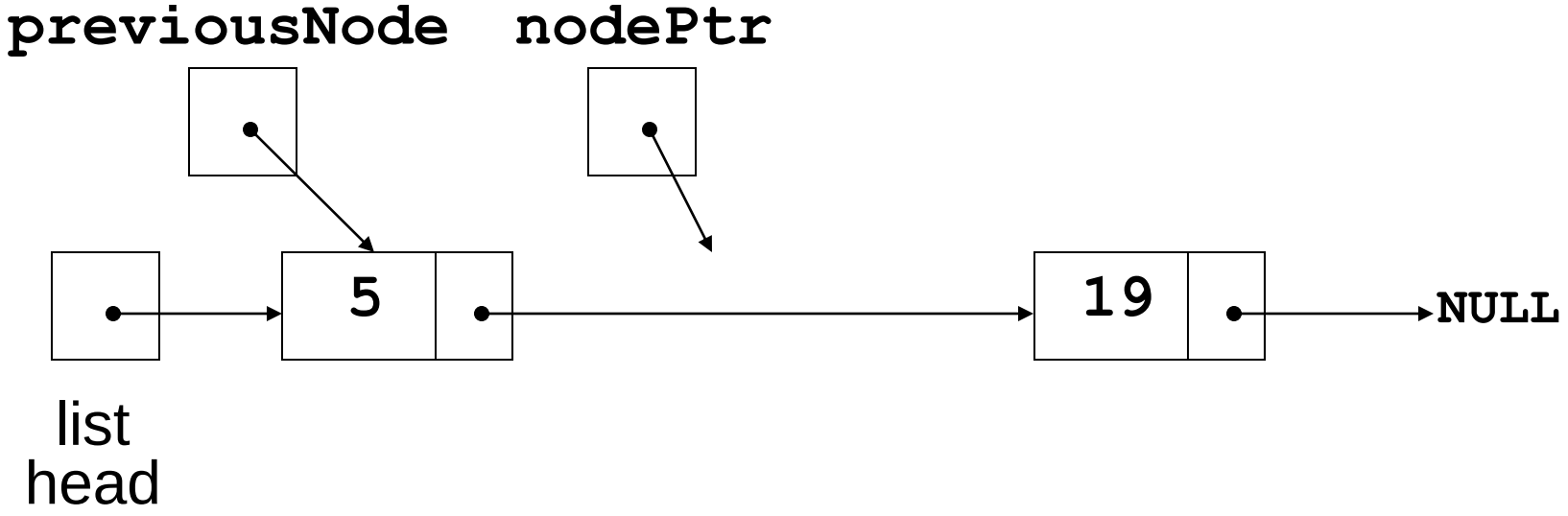
Locating the node containing **13**

Deleting a Node



Adjusting pointer around the node to be deleted

Deleting a Node



Linked list after deleting the node containing **13**