CSCE 121

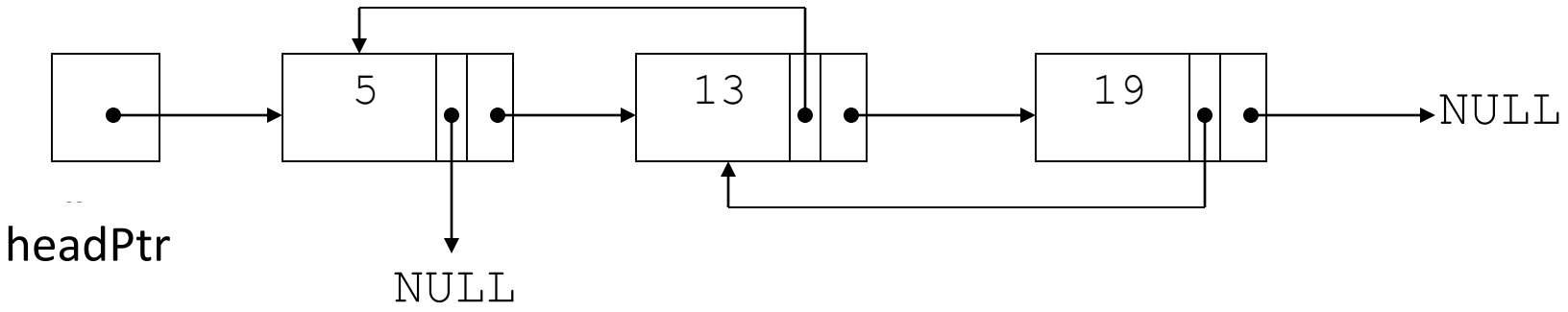
Linked Lists: Implementation

Dr. Tim McGuire

Variations of the Linked List

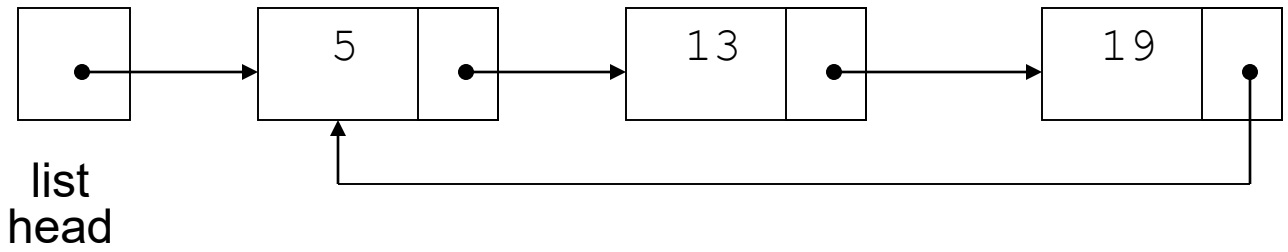
Variations of the Linked List (1)

- Other linked list organizations:
  - doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List (2)

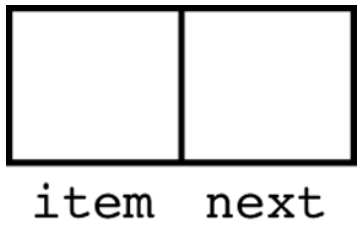
- Other linked list organizations:
  - circular linked list: the last node in the list points back to the first node in the list, not to `nullPtr`
  - Note that the head can move



Pointer-Based Linked Lists

- A node in a linked list is sometimes defined as a template **`struct`**

```
<template <typename T>
struct Node {
   T item;
   Node<T> *next;
};
```



A node

- A node is dynamically allocated

```
Node<T> *p;
p = new Node<T>;
```

Node.h

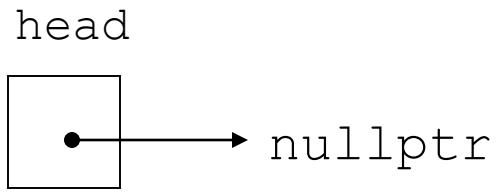
```
#ifndef NODE_H
#define NODE_H

template <typename T>
struct Node
{
   T data;           // Node value
   Node<T> *next;    // Pointer to the next node

   Node (T nodeValue)  // Constructor
   { data = nodeValue;
     next = nullptr;}
};
#endif
```

Defining a Linked List

- Define a pointer for the head of the list:

```
Node<T> *head = nullptr;
```

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



NULL Pointer

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

```
Node<T> *p;
while (p != nullptr) ...
```

- Can also test the pointer itself:

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

Linked List Operations

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:

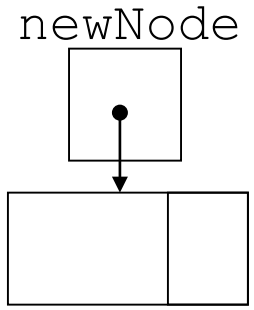
  Node<int>* `newNode = new Node<int>;`

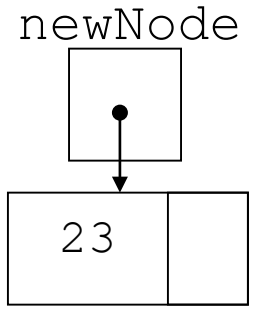
- Initialize the contents of the node:

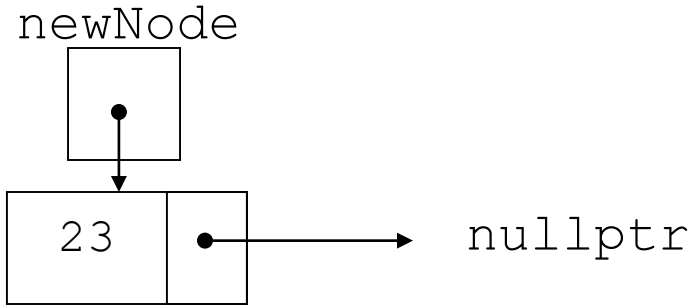
  `newNode->data = num;`

- Set the pointer field to `the null pointer`:

  `newNode->next = nullptr;`



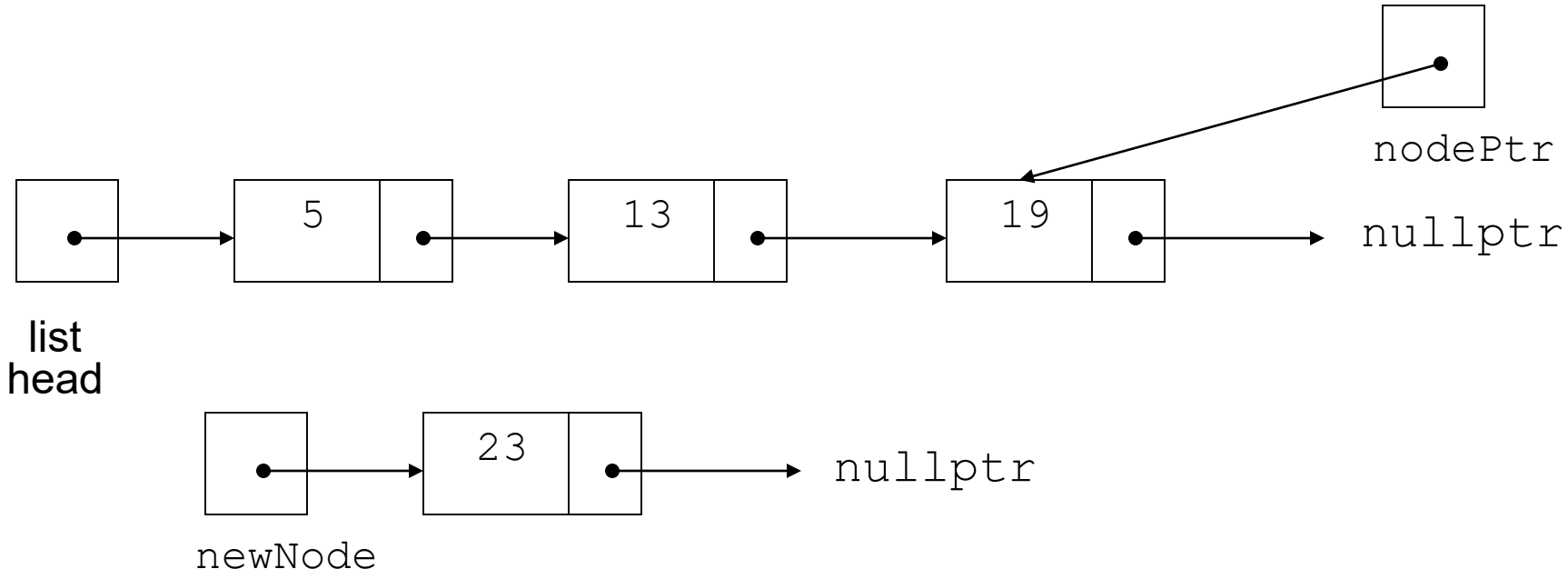




Appending a Node

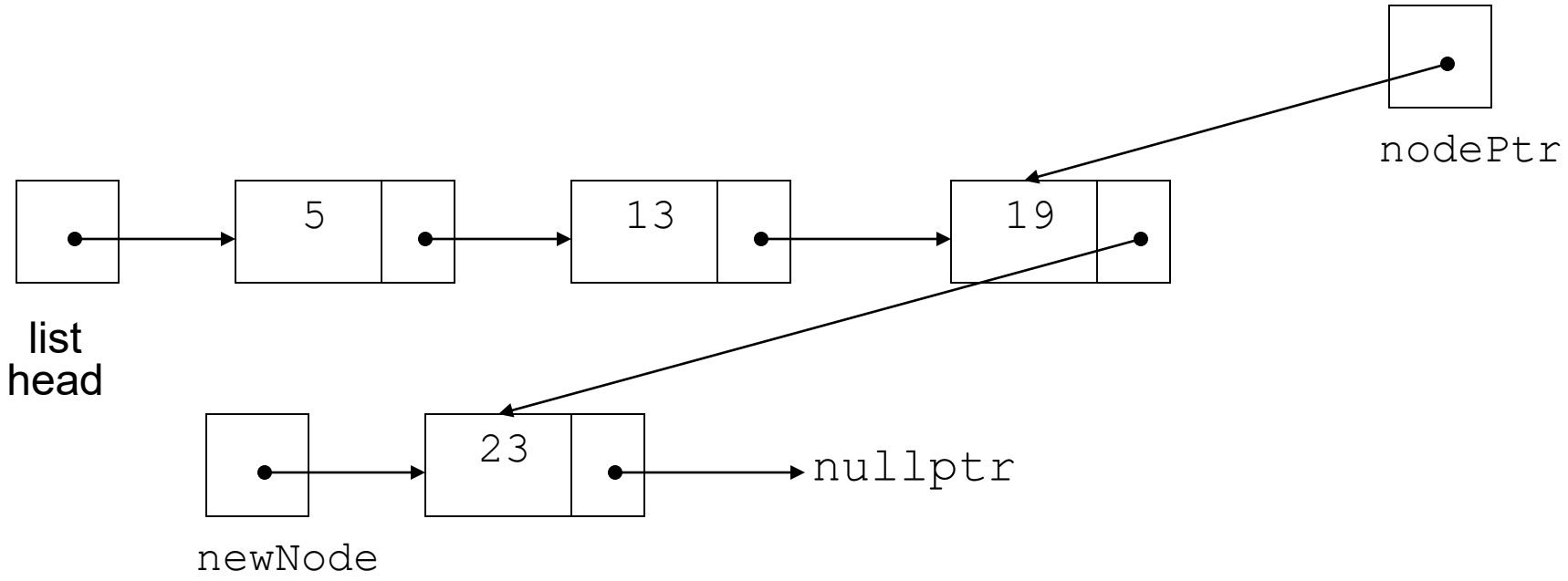
- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



New node created, end of list located

Appending a Node

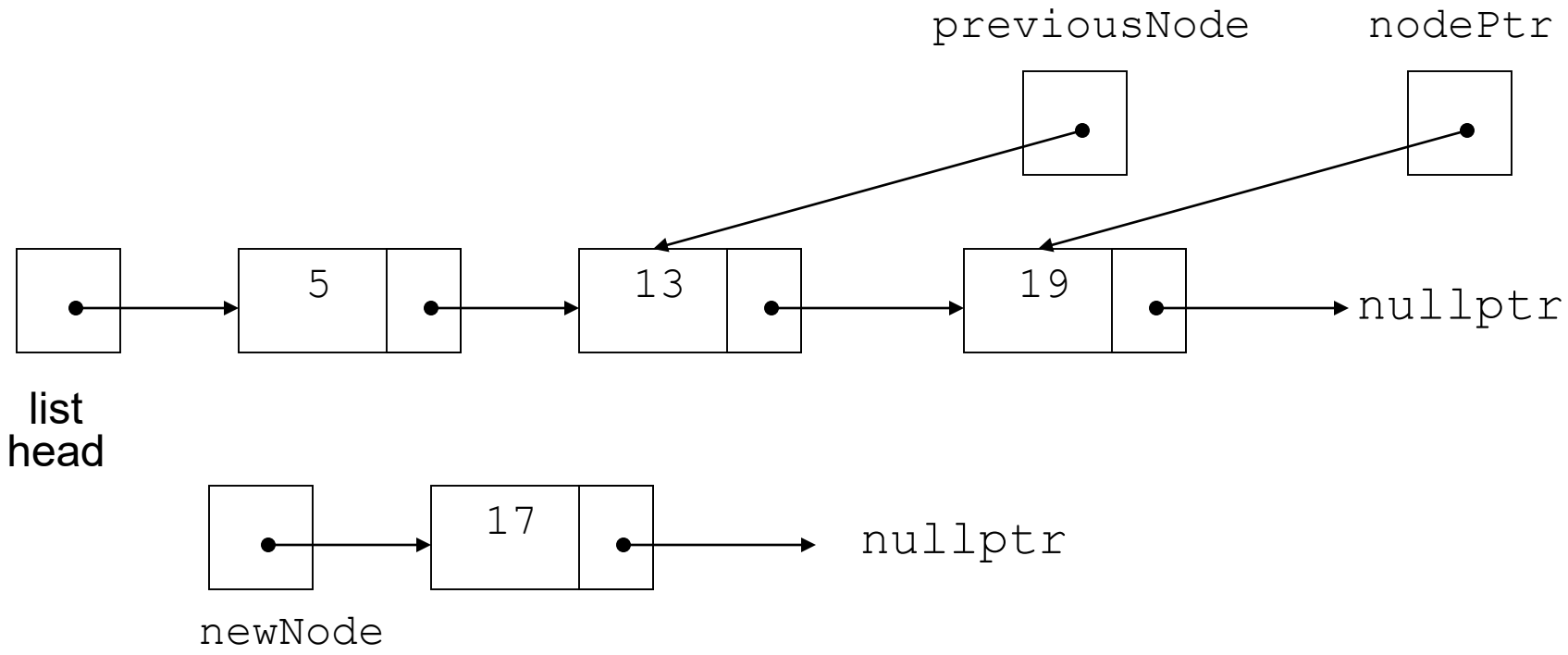


New node added to end of list

Inserting a Node into a Linked List

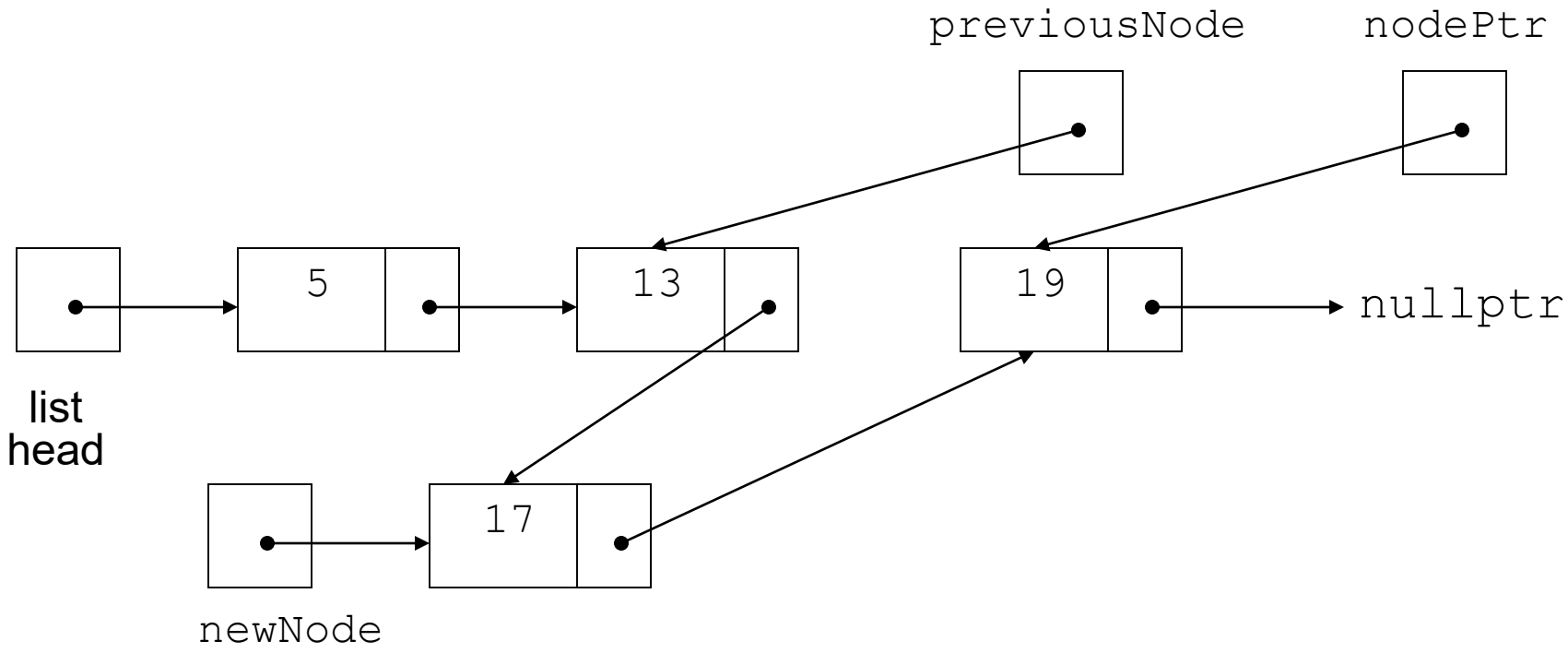
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed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

Inserting a Node into a Linked List



New node created, correct position located

Inserting a Node into a Linked List

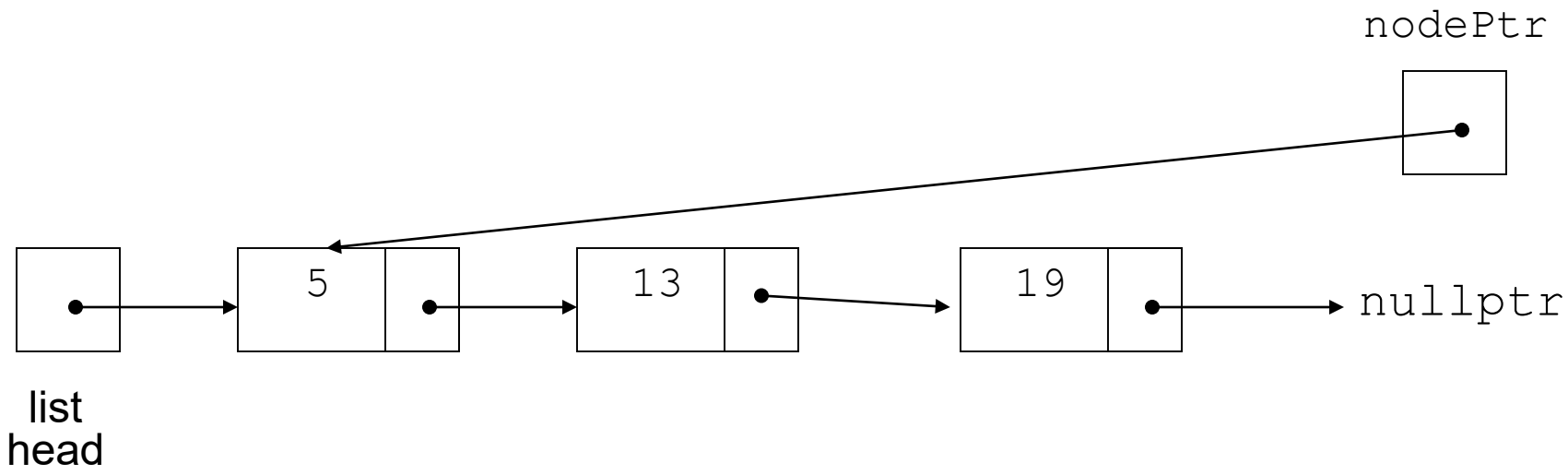


New node inserted in order in the linked list

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 `nullptr`
    - 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

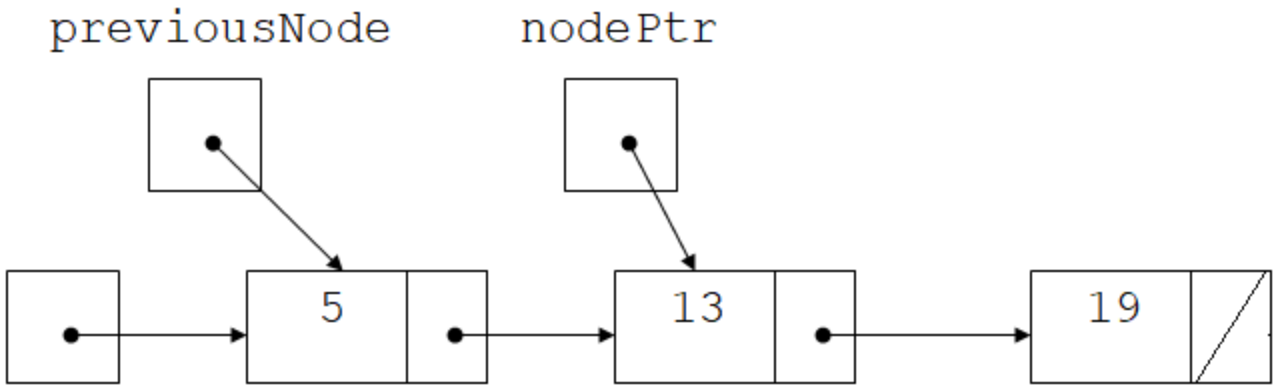


`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to `nullptr`, and the list traversal stops

Deleting a Node

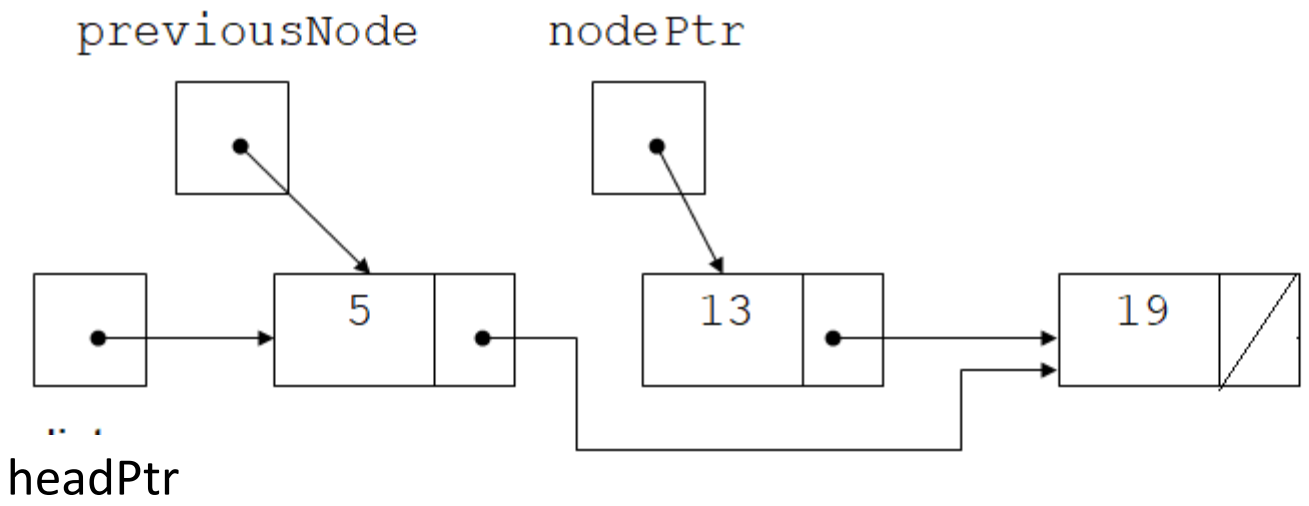
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node (2)



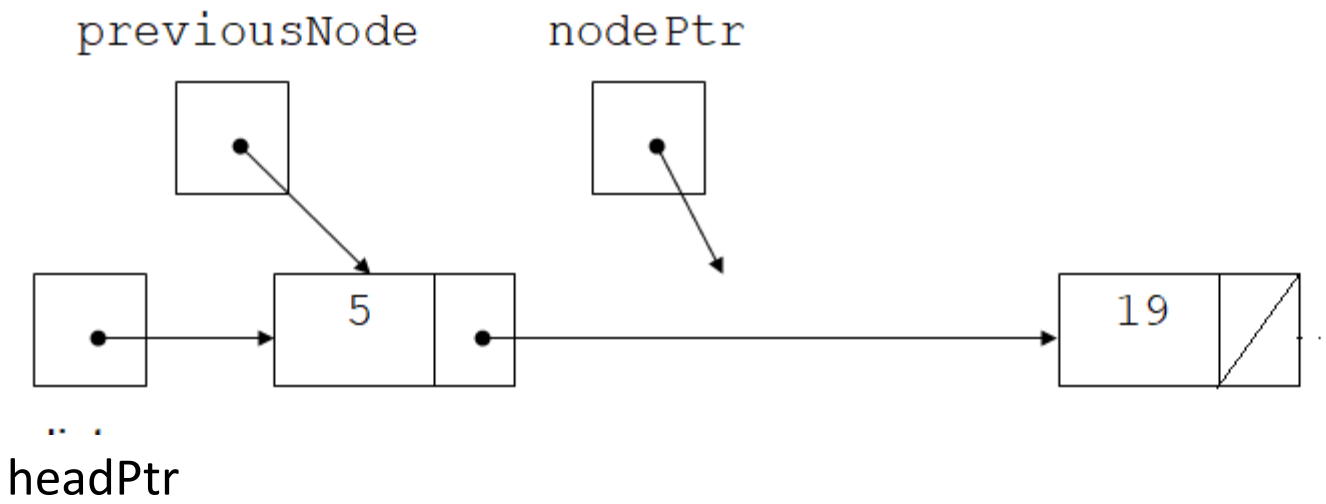
Locating the node containing `13`

Deleting a Node (3)



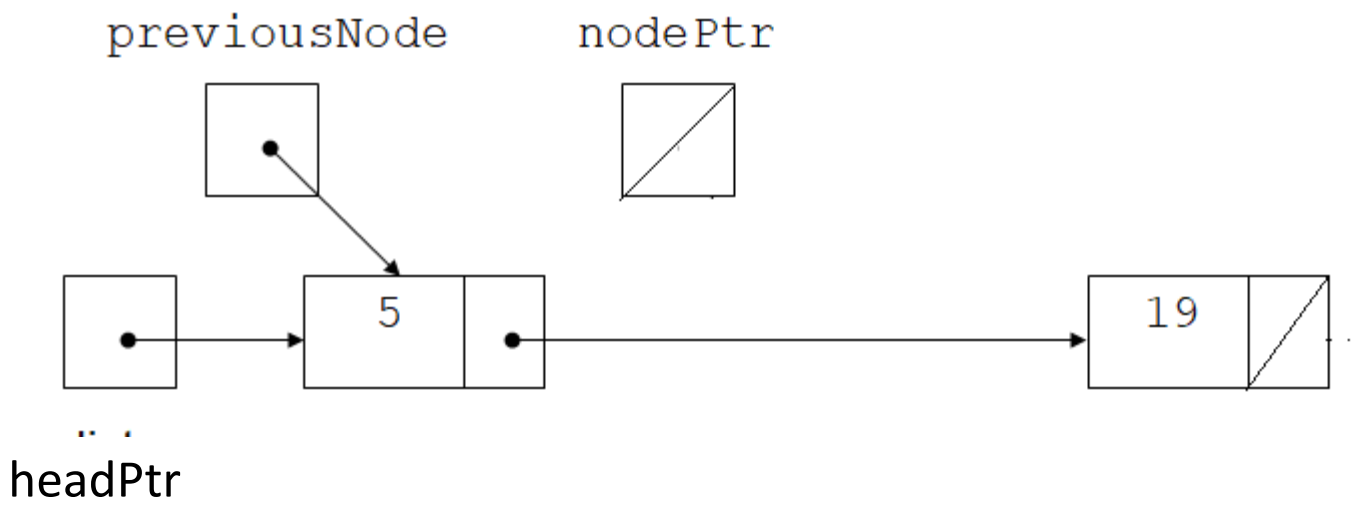
Adjusting pointer around the node to be deleted

Deleting a Node (4)



Linked list after deleting the node containing `13`

Deleting a Node



Linked list after deleting the node containing `13`

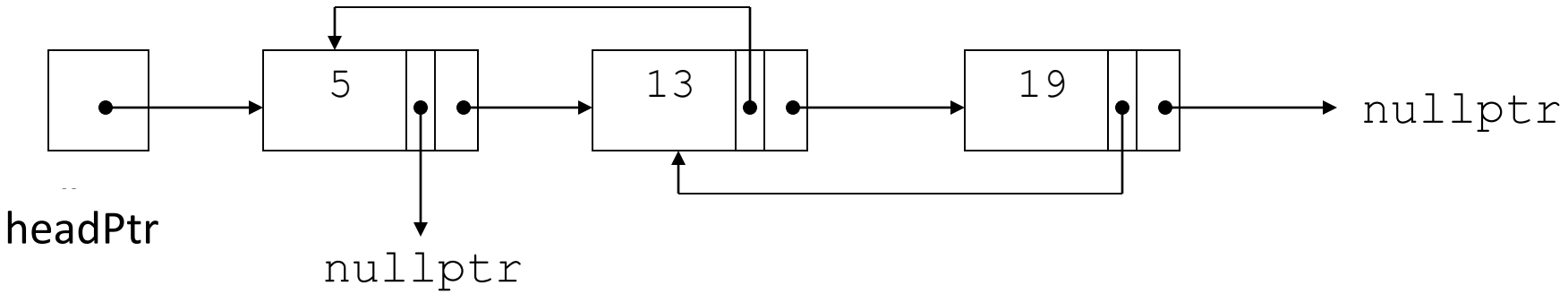
Destroying a Linked List

- 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 `nullptr`

Variations of the Linked List

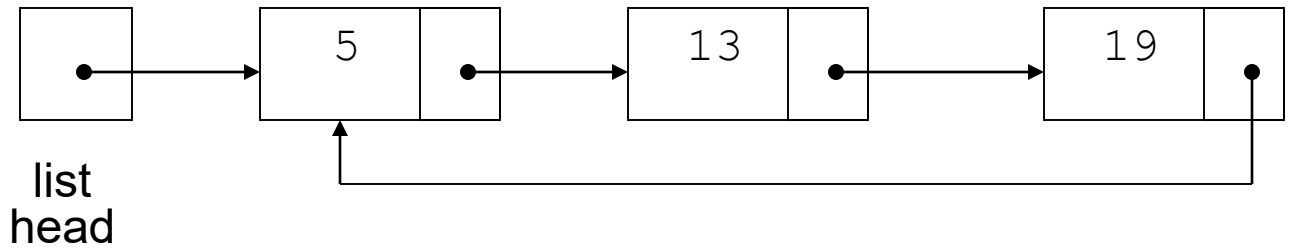
Variations of the Linked List (1)

- Other linked list organizations:
  - doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List (2)

- Other linked list organizations:
  - circular linked list: the last node in the list points back to the first node in the list, not to `nullptr`
  - Note that the head can move



The STL `list` Container

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`