Topics

1 Introduction to the Linked List ADT

2 Linked List Operations

3 A Linked List Template

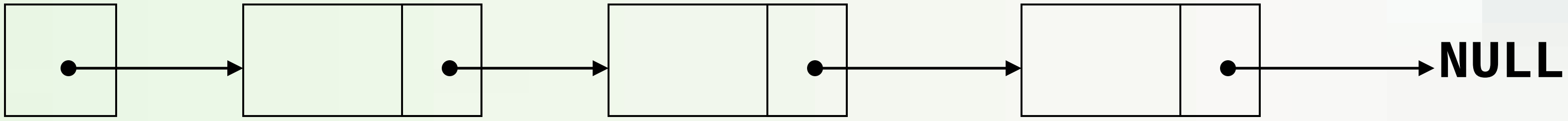
4 Recursive Linked List Operations

5 Variations of the Linked List

6 The STL **`list`** Container

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



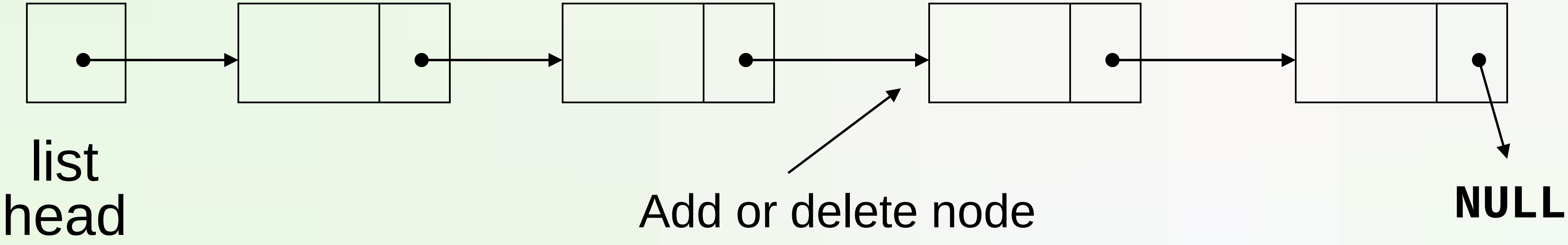
list head

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

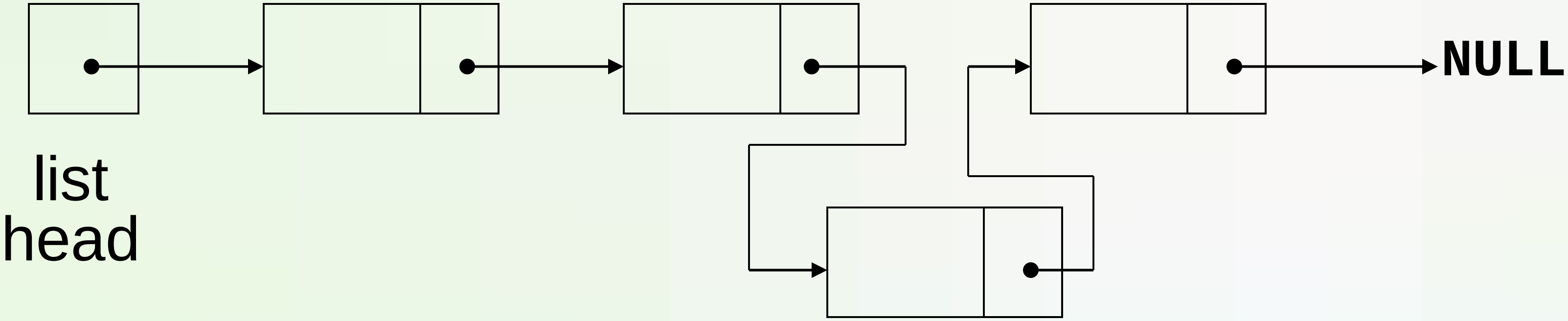
- Nodes can be added or removed from the linked list during execution
- Addition or removal of nodes can take place at beginning, end, or middle of the list



Linked Lists vs. Arrays and Vectors

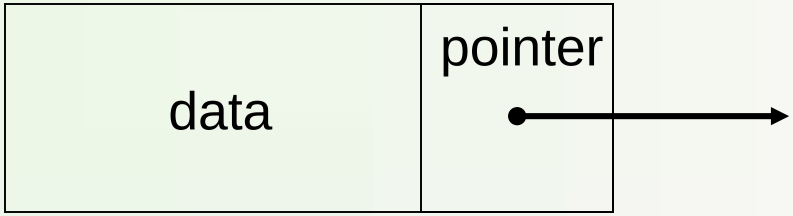
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size

- Unlike vectors, insertion or removal in the middle of the list is very efficient



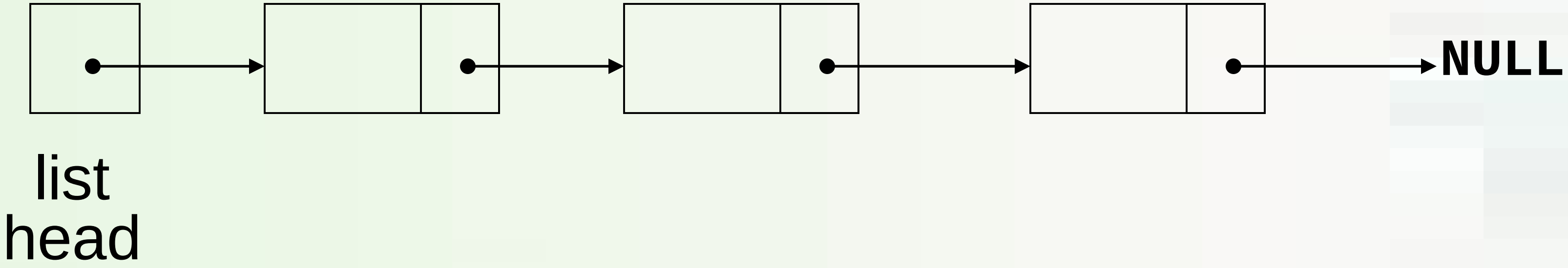
Node Organization

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



Linked List Organization

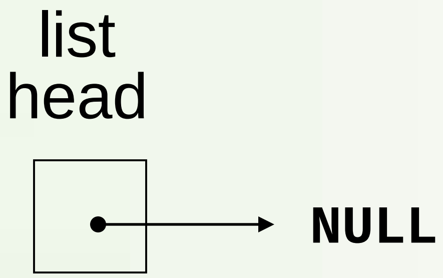
- Linked list contains 0 or more nodes:



- Has a list head to point to first node
- Successor pointer for the last node is set to **NULL**

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:

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

C++ Implementation

- Nodes can be equipped with constructors:

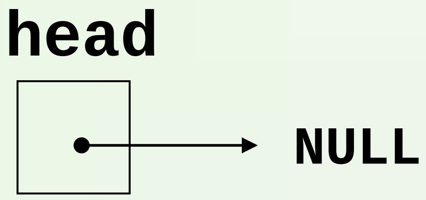
```
struct ListNode
{
  int data;
  ListNode *next;
  ListNode(int d, ListNode* p=0)
   {data = d; next = p;}
};
```

Creating an Empty List

- Define a pointer for the head of the list:

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

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



head

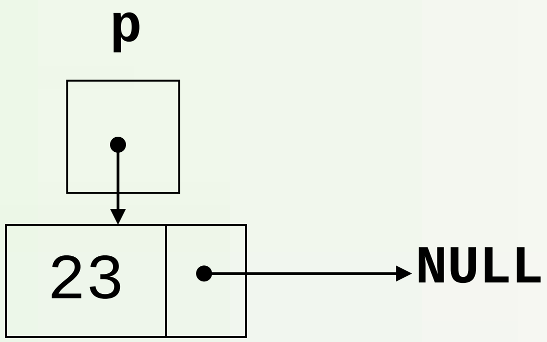
NULL

2 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

Creating a Node

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



Appending an Item

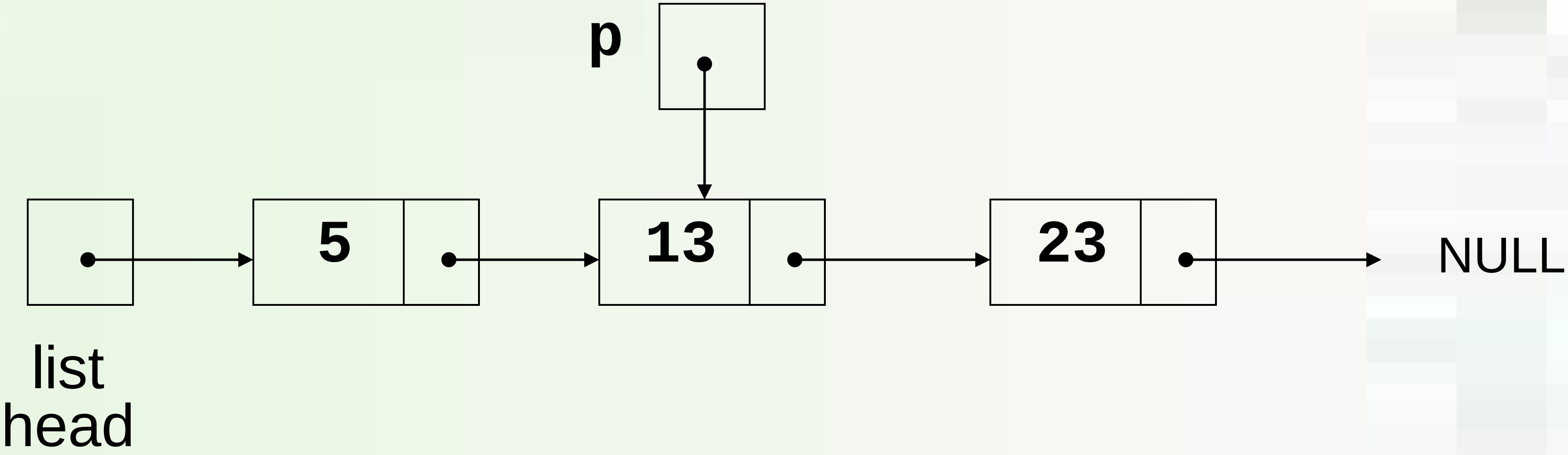
- To add an item to the end of the list:
  - If the list is empty, set **head** to a new node containing the item

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

  - If the list is not empty, move a pointer **p** to the last node, then add a new node containing the item

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

Appending an Item



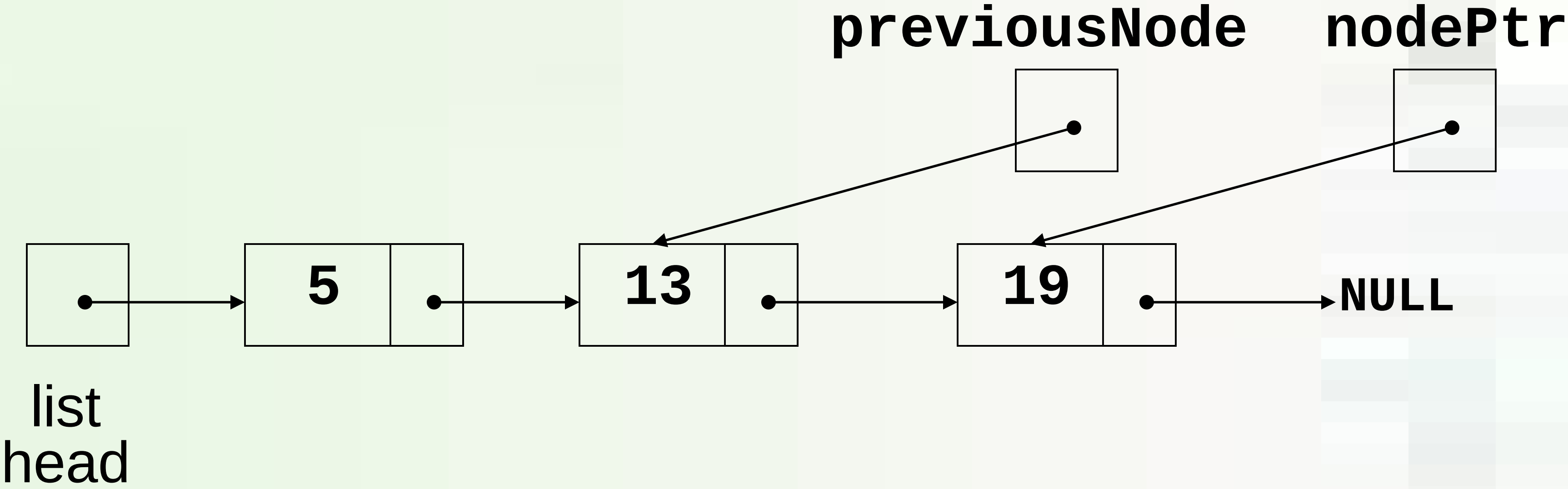
List originally has 5, 13

p locates last node, new item, 23, is added

Inserting a Node

- Used to insert an item into a sorted list, keeping the list sorted.
- 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

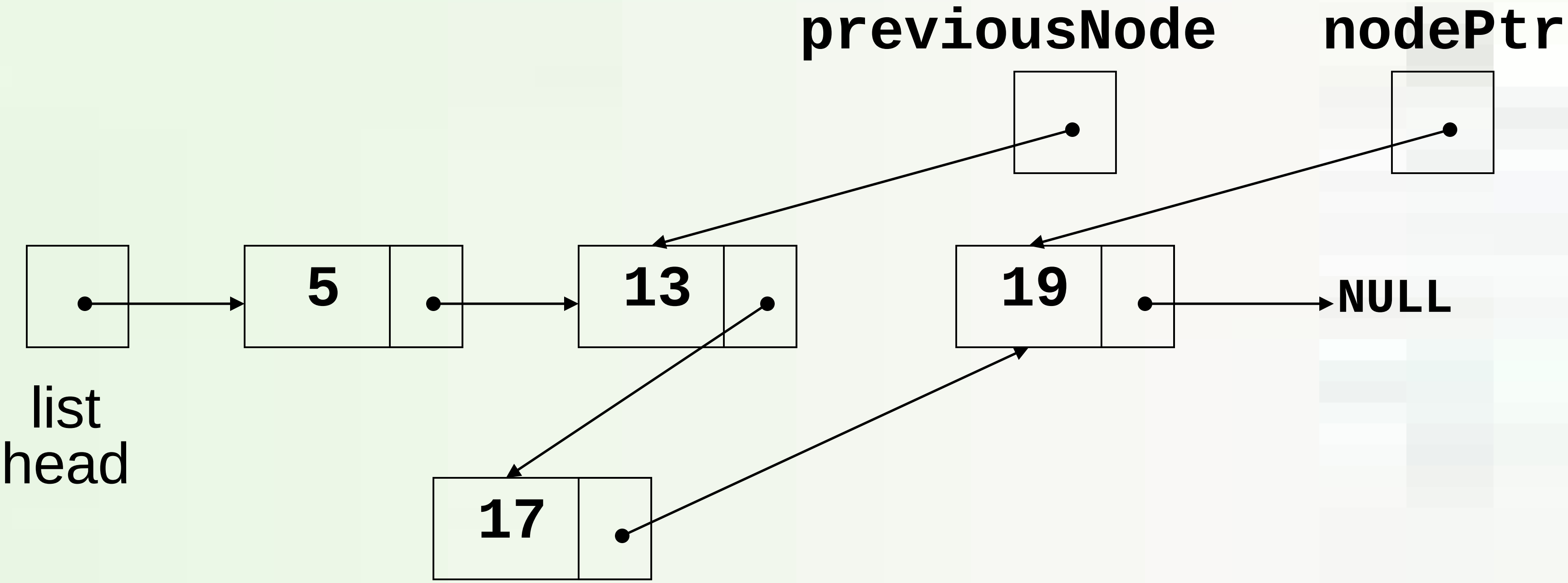


17

Item to insert

Correct position located

Inserting a Node into a Linked List



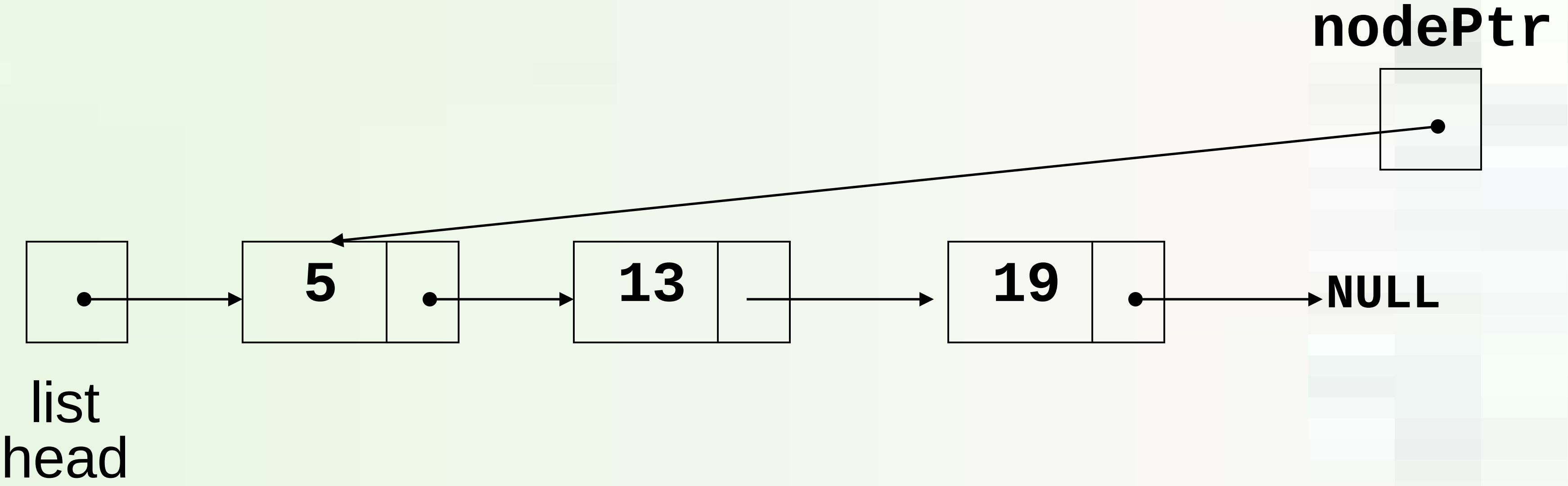
New node created and inserted in order in the linked list

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

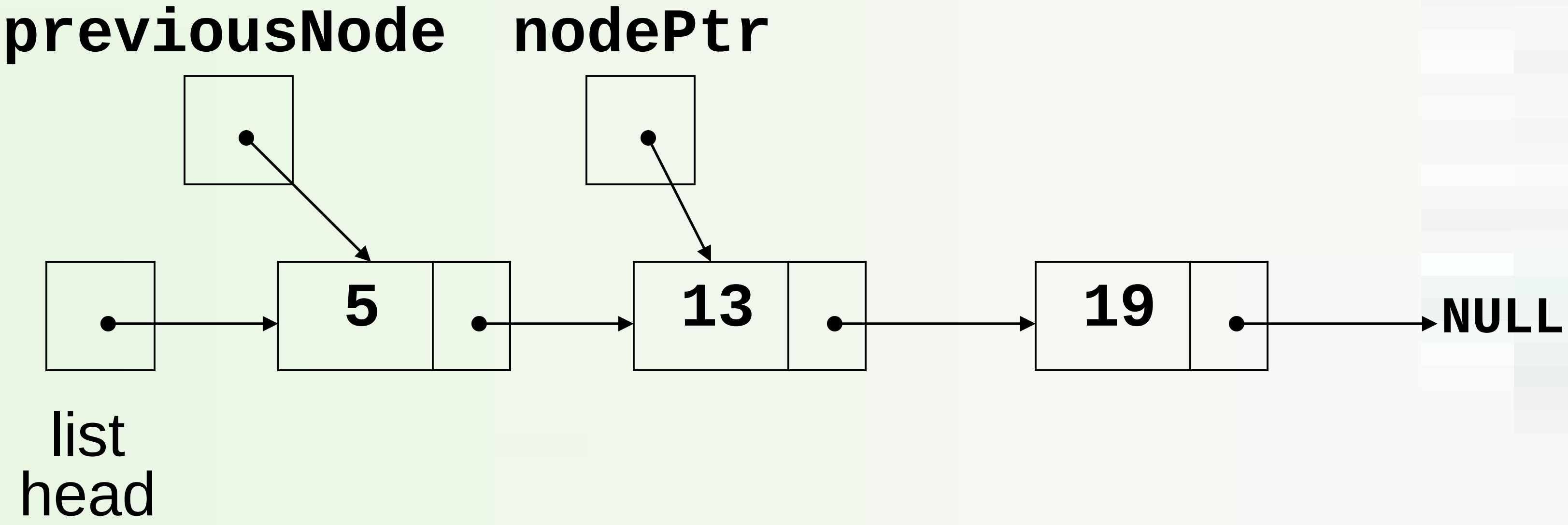


nodePtr points to the node containing **5**, then the node containing **13**, then the node containing **19**, then points to **NULL**, and the list traversal stops

Deleting a Node

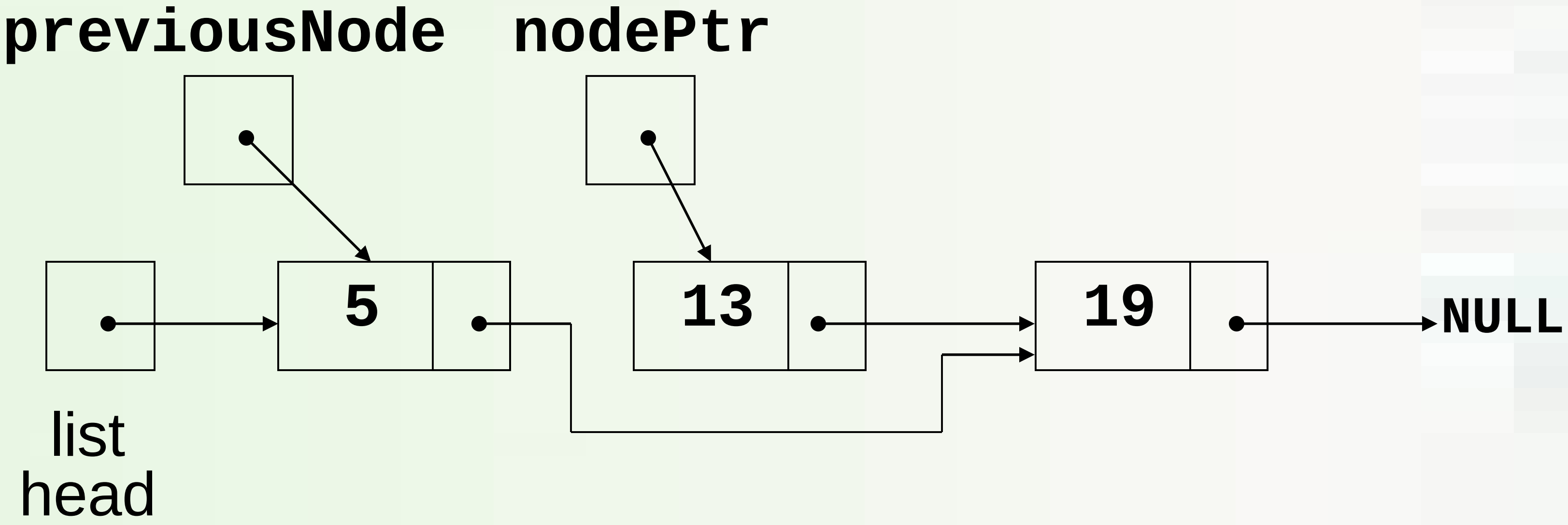
- Used to remove a node from a linked list
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node



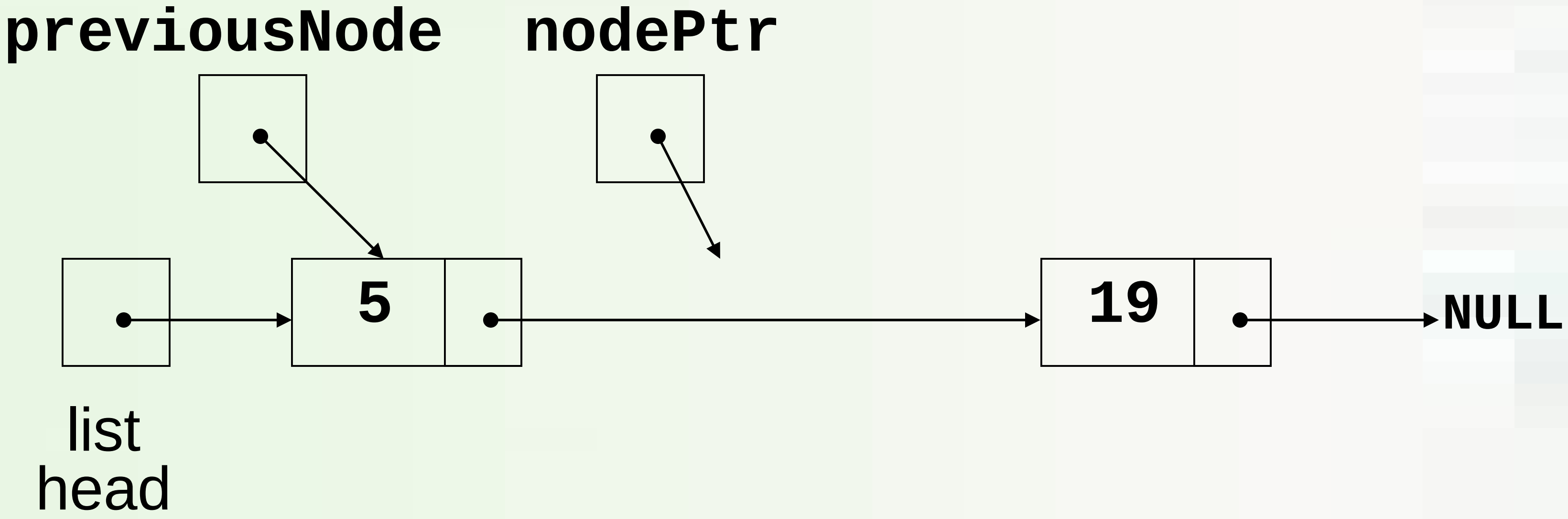
Locating the node containing **13**

Deleting a Node



Adjusting pointer around the node to be deleted

Deleting a Node



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

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
- Set the list head to **`NULL`**

3 A Linked List Template

- A linked list template can be written by replacing the type of the data in the node with a type parameter, say T.

- More on templates later in this course.

4 Recursive Linked List Operations

- A non-empty linked list consists of a head node followed by the rest of the nodes
- The rest of the nodes form a linked list that is called the tail of the original list

Recursive Linked List Operations

- Many linked list operations can be broken down into the smaller problems of processing the head of the list and then recursively operating on the tail of the list

Recursive Linked List Operations

- To find the length of a list
  - If the list is empty, the length is 0 (base case)
  - If the list is not empty, find the length of the tail and then add 1 to obtain length of original list

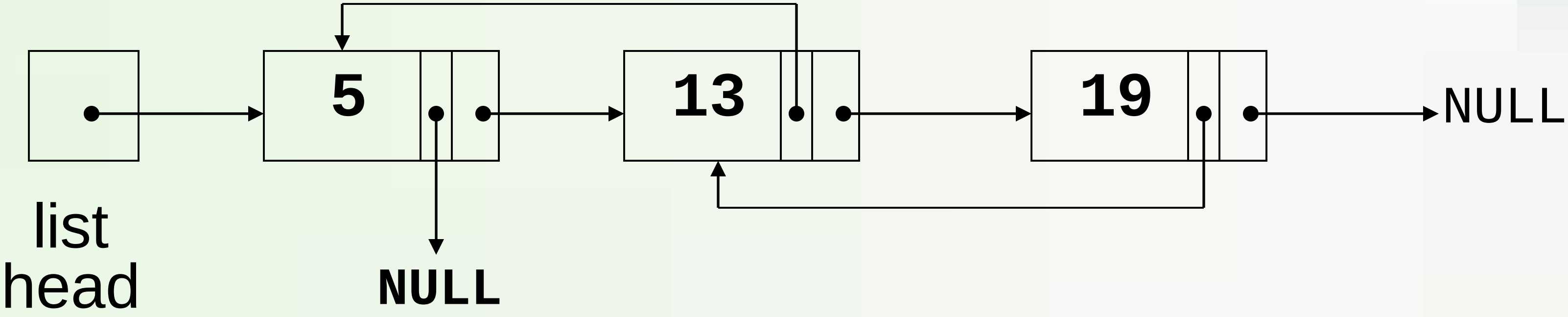
Recursive Linked List Operations

- To find the length of a list

```
int length(ListNode *myList)
{
   if (myList == NULL) return 0;
   else
   return 1 + length(myList->next);
}
```

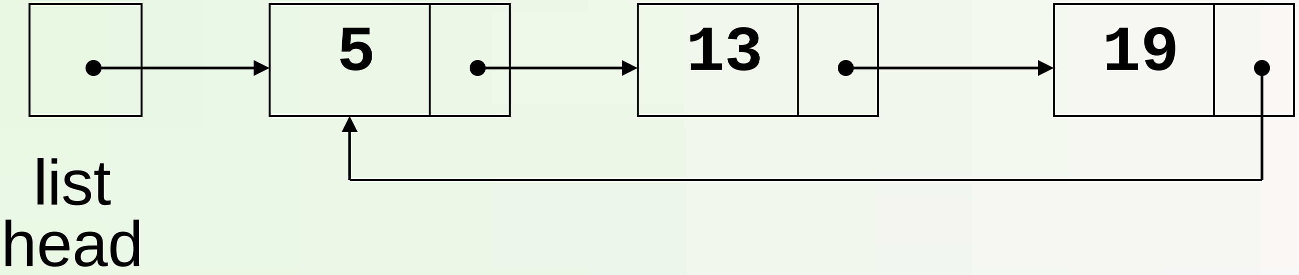
5 Variations of the Linked List

- 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

- Other linked list organizations:
  - – circular linked list: the last node in the list points back to the first node in the list, not to **`NULL`**



6 The STL **`list`** Container

- Template for a doubly linked list
- Member functions include:
  - 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`**