BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

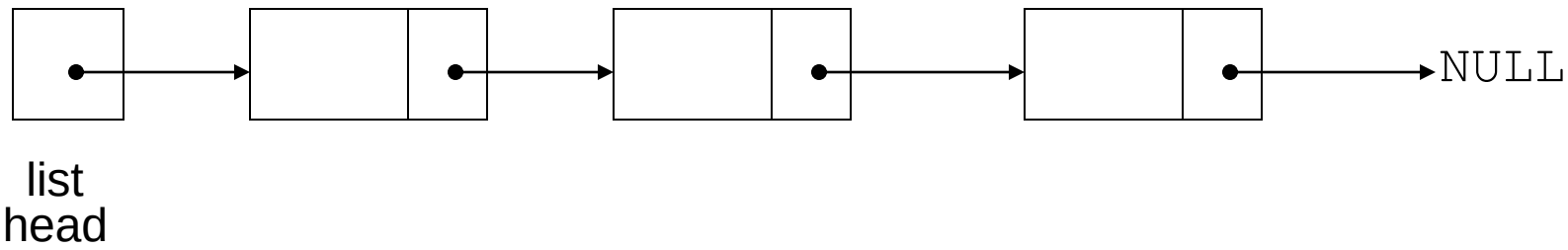
Programming and Computer Applications-2

Linked Lists

Instructor : PhD, Associate Professor Leyla Muradkhanli

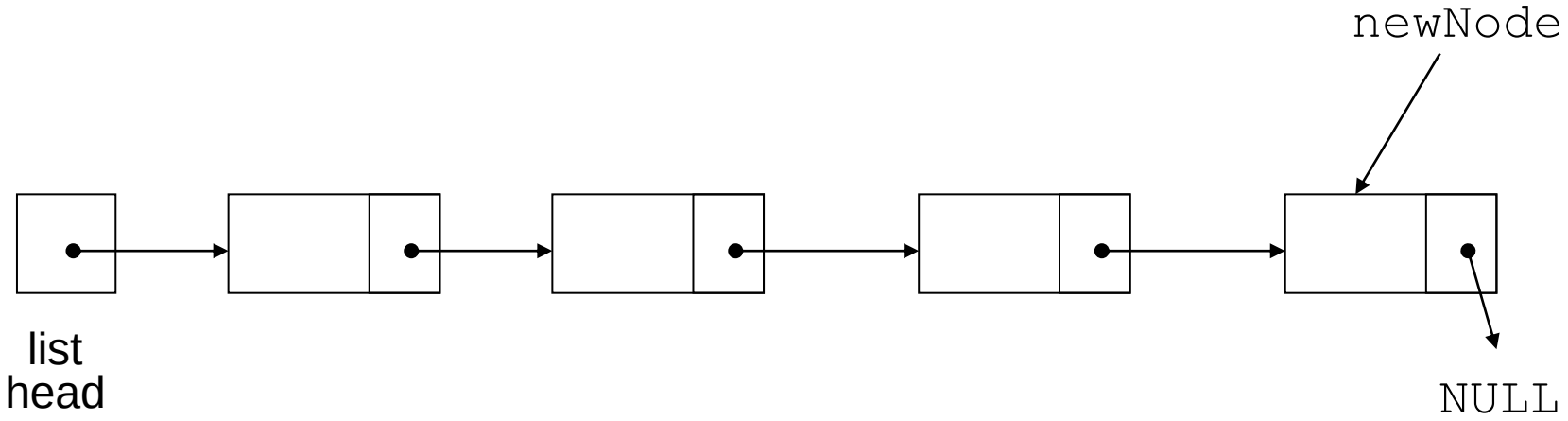
Introduction to the Linked List

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



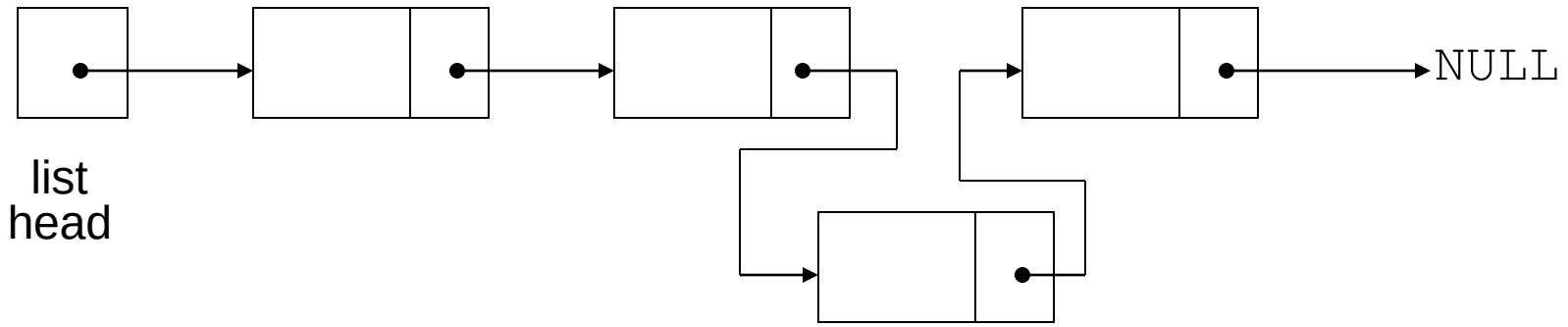
Introduction to the Linked List

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



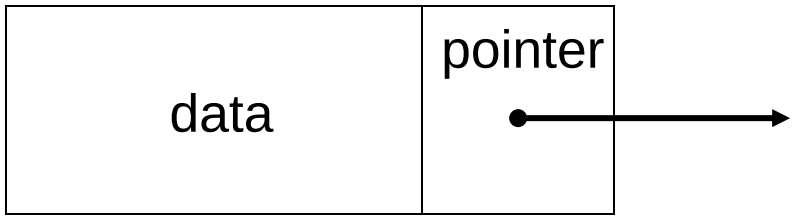
Linked Lists vs. Arrays and Vectors

- 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



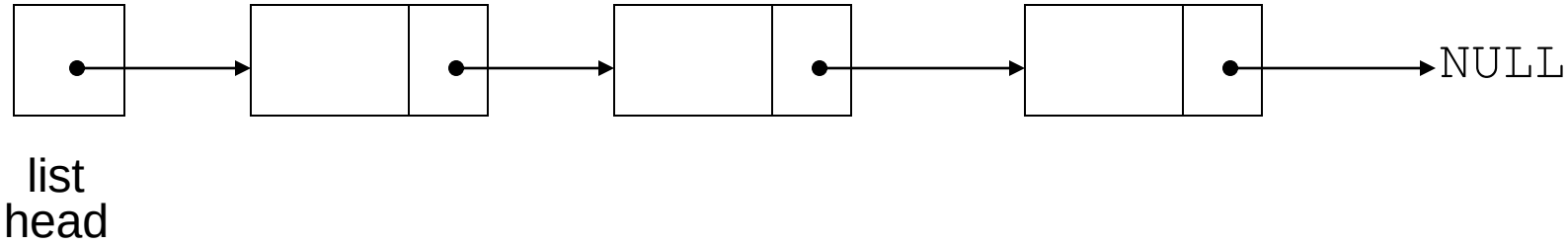
Node Organization

- 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

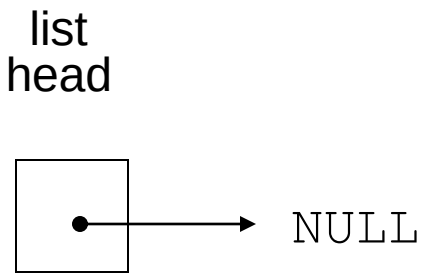
- Linked list contains 0 or more nodes:



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

Empty List

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



Declaring a Node

- Declare a node:

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

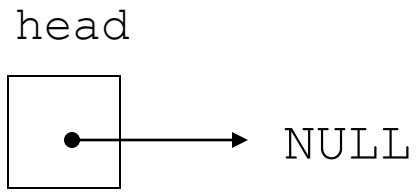
- No memory is allocated at this time

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
```

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:

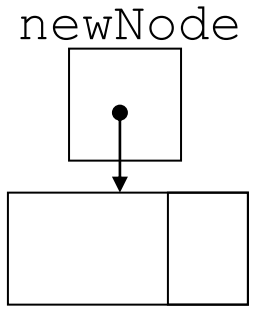
```
newNode = new ListNode;
```

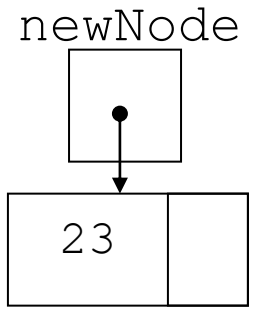
- Initialize the contents of the node:

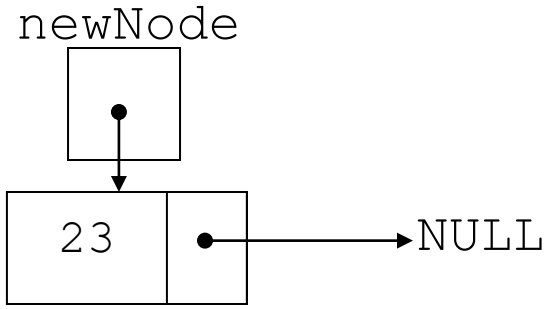
```
newNode->value = num;
```

- Set the pointer field to `NULL`:

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



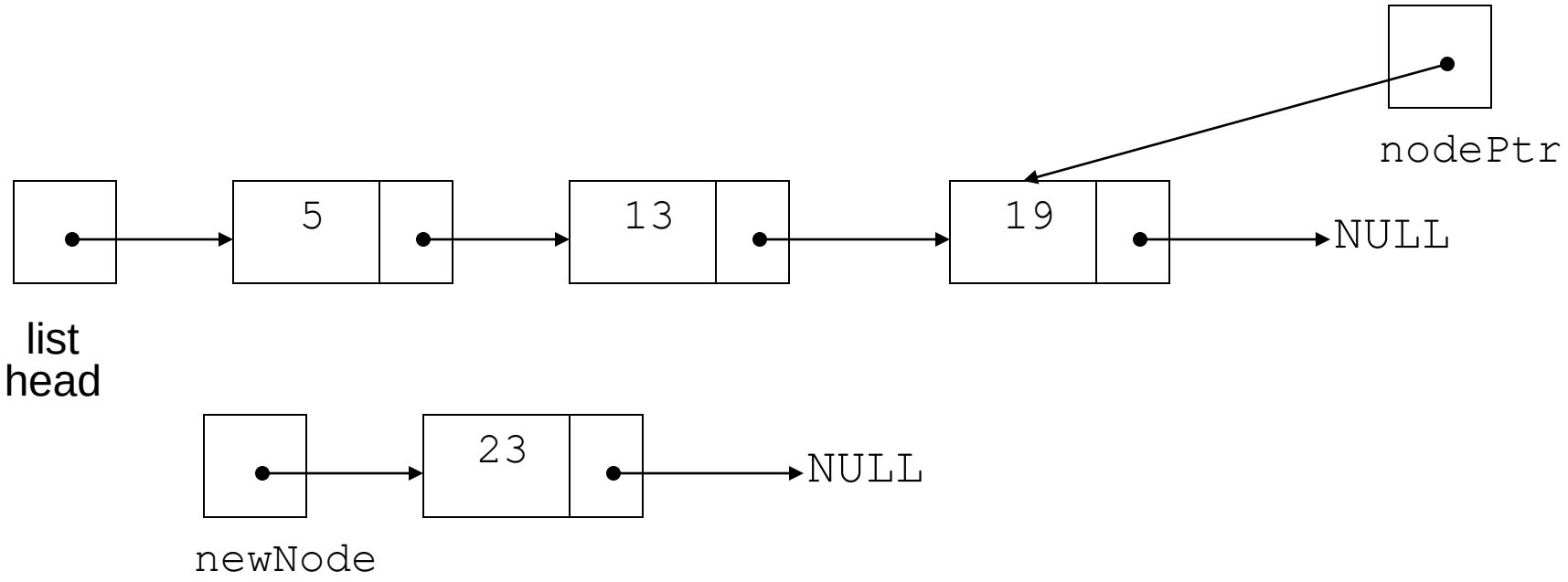




Appending a Node

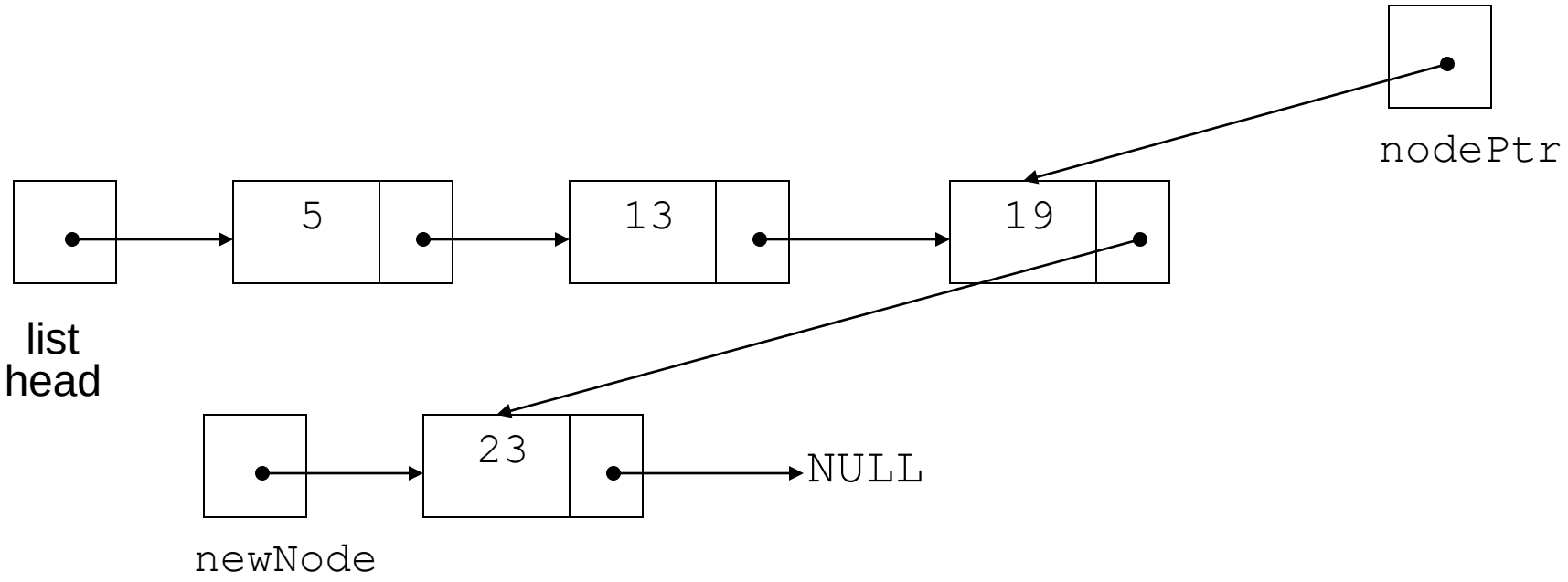
- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



New node created, end of list located

Appending a Node

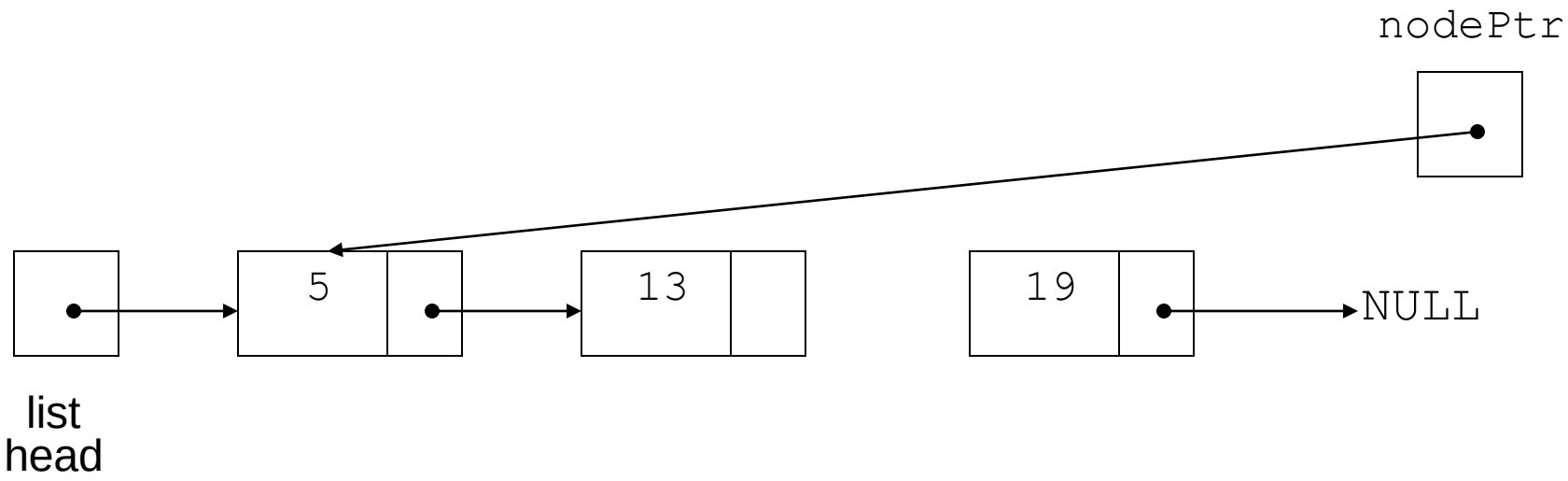


New node added to end of list

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
    - 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

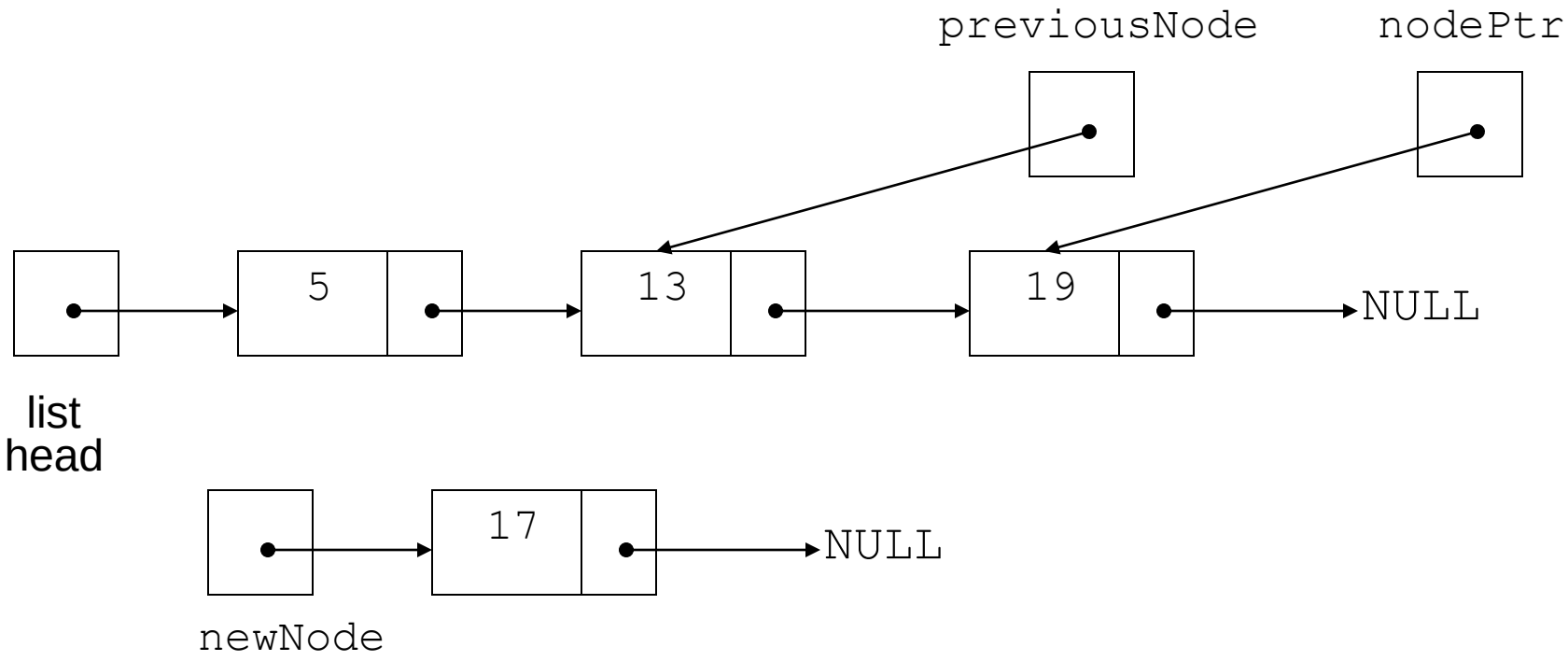


`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to `NULL`, and the list traversal stops

Inserting a Node into a Linked List

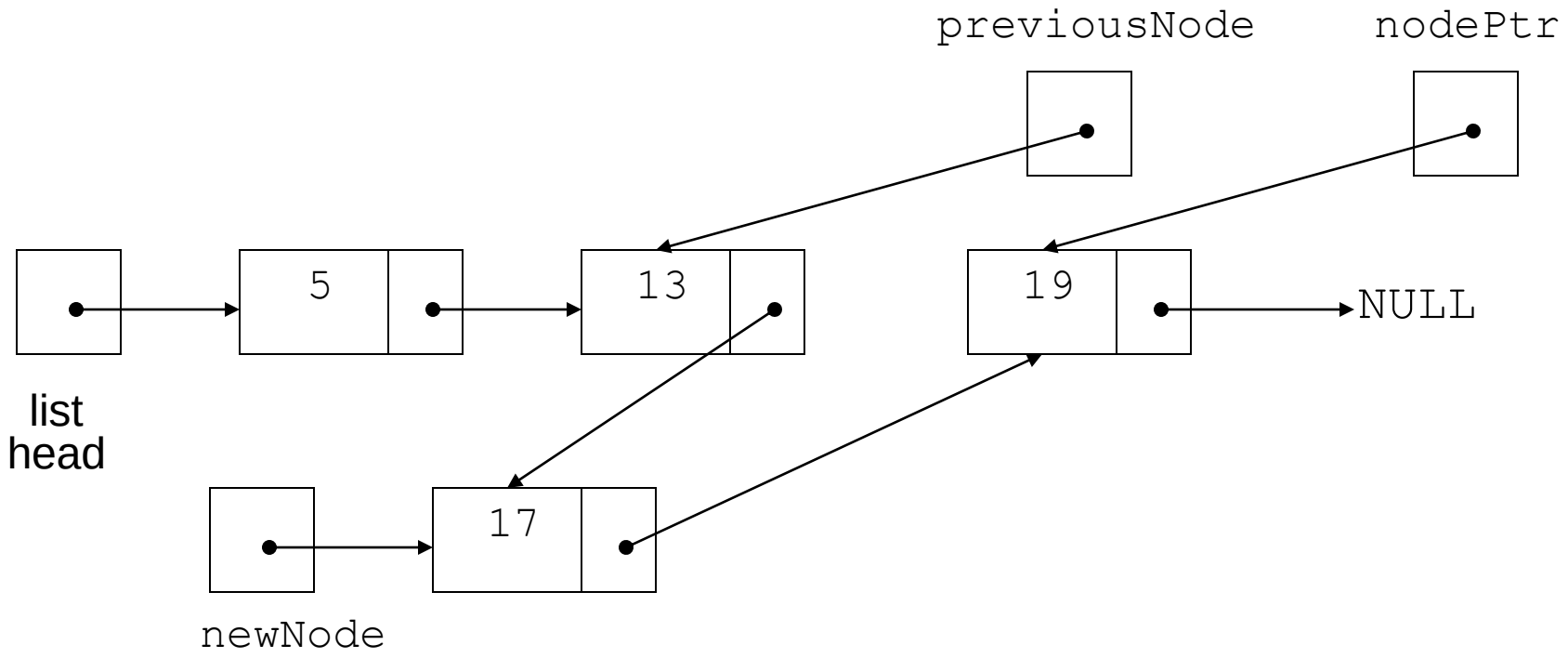
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed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

Inserting a Node into a Linked List



New node created, correct position located

Inserting a Node into a Linked List

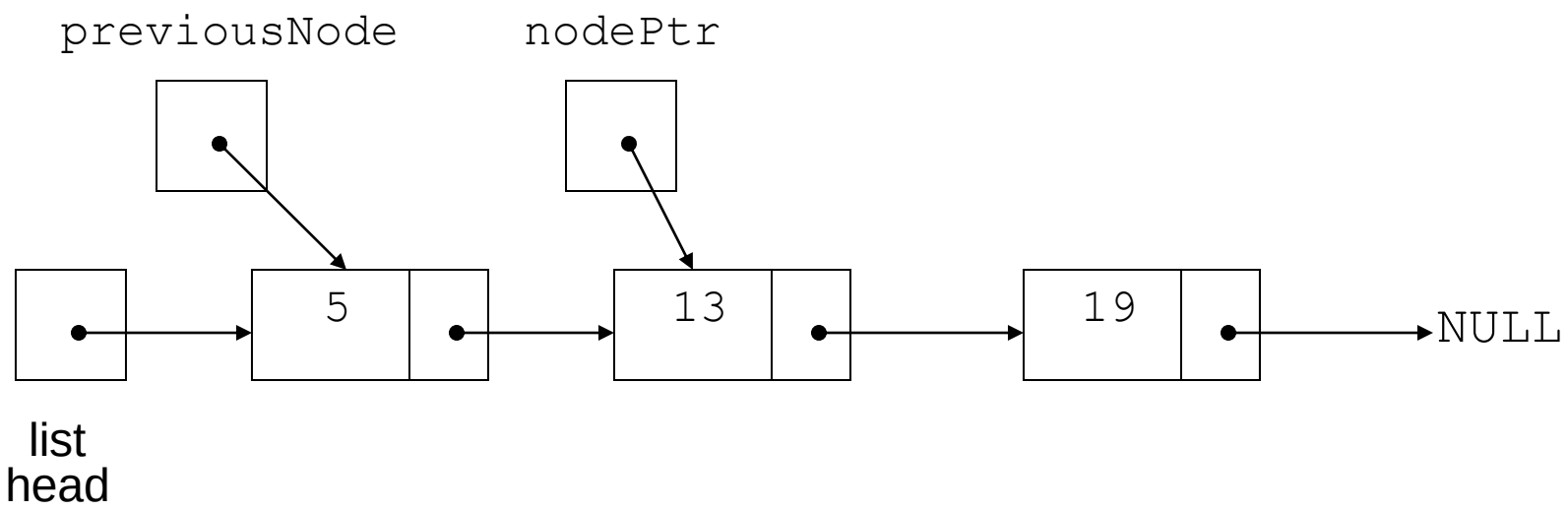


New node inserted in order in the linked list

Deleting a Node

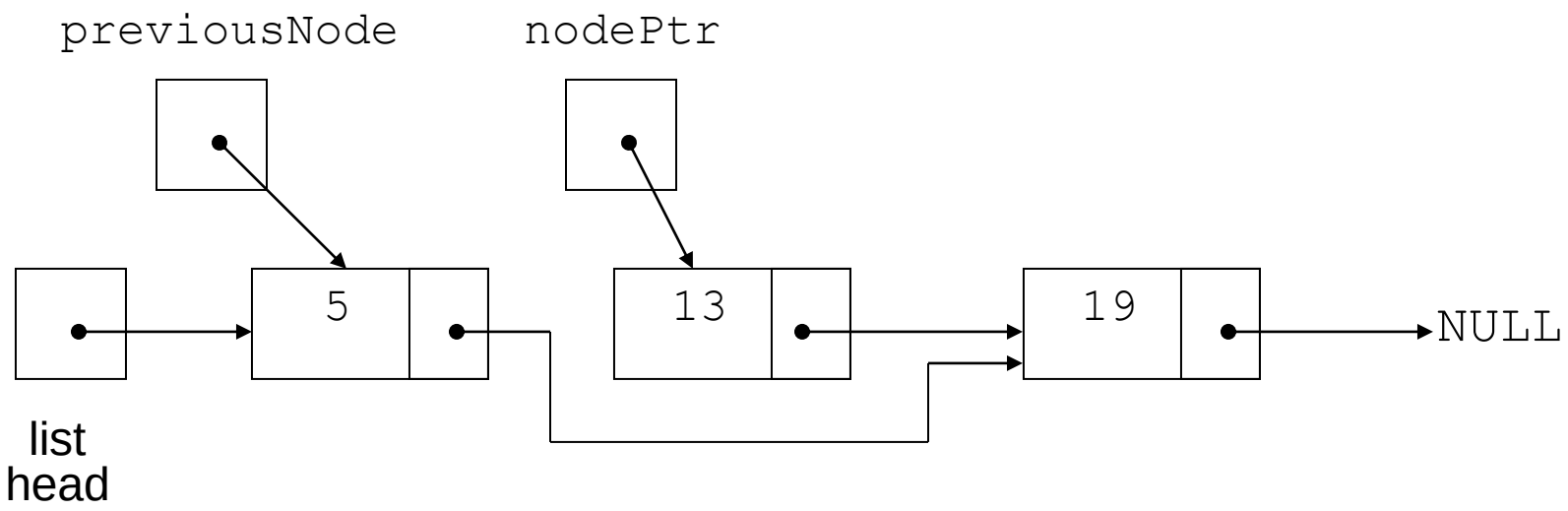
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node



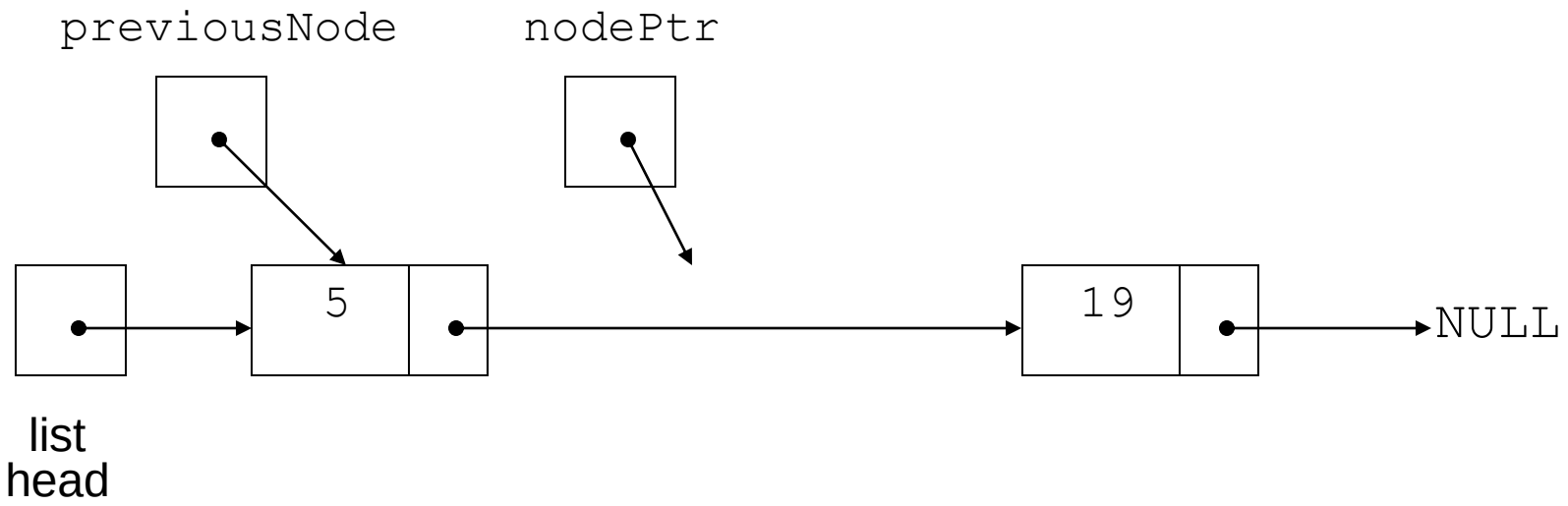
Locating the node containing `13`

Deleting a Node



Adjusting pointer around the node to be deleted

Deleting a Node



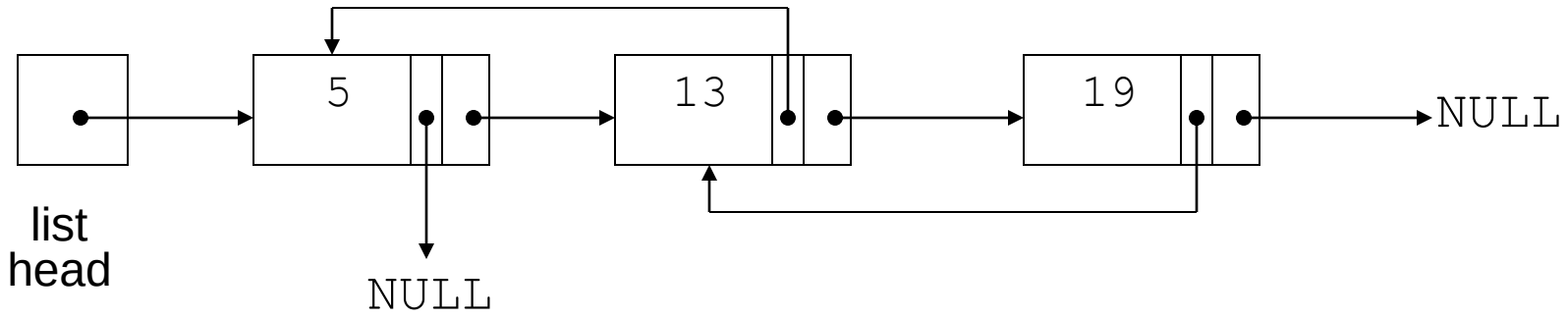
Linked list after deleting the node containing `13`

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`

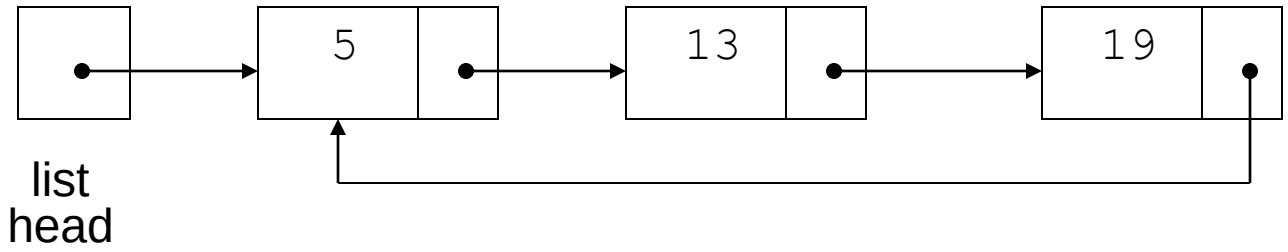
Variations of the Linked List

- 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`



The STL `list` Container

Functions used with List :

front() – Returns reference to the first element in the list

back() – Returns reference to the last element in the list

push_front() – Adds a new element ‘g’ at the beginning of the list

push_back() – Adds a new element ‘g’ at the end of the list

pop_front() – Removes the first element of the list, and reduces size of the list by 1

pop_back() – Removes the last element of the list, and reduces size of the list by 1

The STL `list` Container

begin() – Returns an iterator pointing to the first element of the list

end() – Returns an iterator pointing to the theoretical last element which follows the last element

empty() – Returns whether the list is empty(1) or not(0)

insert() – Inserts new elements in the list before the element at a specified position

erase() – Removes a single element or a range of elements from the list

The STL `list` Container

assign() – Assigns new elements to list by replacing current elements and resizes the list

remove() – Removes all the elements from the list, which are equal to given element

reverse() – Reverses the list

size() – Returns the number of elements in the list

sort() – Sorts the list in increasing order

Examples

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node *next;
};
Node *head=NULL;
void insert(int new_data) {
Node *new_node;
new_node=new Node;
new_node->data=new_data;
new_node->next=head;
head=new_node;
}
void display() {
Node *ptr;
ptr=head;
while(ptr!=NULL) {
cout<<ptr->data<<" ";
ptr=ptr->next;
}
}
```

Examples

```
int main()
{
insert(5);
insert(1);
insert(7);
insert(9);
insert(11);
cout<<"The linked list is : ";
display();
return 0;
}
```

The linked list is : 11 9 7 1 5

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l;
for (int i = 1; i < 10; ++i)
l.push_front(i);
cout << "List contains the following elements" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

```
List contains the following elements
9
8
7
6
5
4
3
2
1
```

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l;
for (int i = 1; i < 10; ++i)
l.push_back(i);
cout << "List contains the following elements" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

```
List contains the following elements
1
2
3
4
5
6
7
8
9
```

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l;
for (int i = 1; i < 10; ++i)
l.insert(l.end(), i);
cout << "List contains the following elements" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

```
List contains the following elements
9
8
7
6
5
4
3
2
1
```

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l;
l.insert(l.begin(), 5, 7);
cout << "List contains the following elements" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

```
List contains the following elements
7
7
7
7
7
```

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l;
l.push_front(1);
l.push_front(2);
l.push_front(3);
l.push_front(4);
l.push_front(5);
cout << "List contains the following elements before pop operation" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
l.pop_front();
cout << "List contains the following elements after pop operation" << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

The STL `list` Container

List contains the following elements before pop operation
5
4
3
2
1
List contains the following elements after pop operation
4
3
2
1

The STL `list` Container

```
#include <iostream>
#include <list>
using namespace std;
bool comp(int a, int b) {
return (a > b);
}
int main() {
list<int> l;
l.push_back(10);
l.push_back(2);
l.push_back(9);
l.push_back(4);
l.push_back(7);
```

The STL `list` Container

```
cout << "List contains the following elements " << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
l.sort(comp);
cout << "List contains the following elements after sort " << endl;
for (auto it = l.begin(); it != l.end(); ++it)
cout << *it << endl;
return 0;
}
```

List contains the following elements
10
2
9
4
7
List contains the following elements after sort
10
9
7
4
2