Chapter 18:

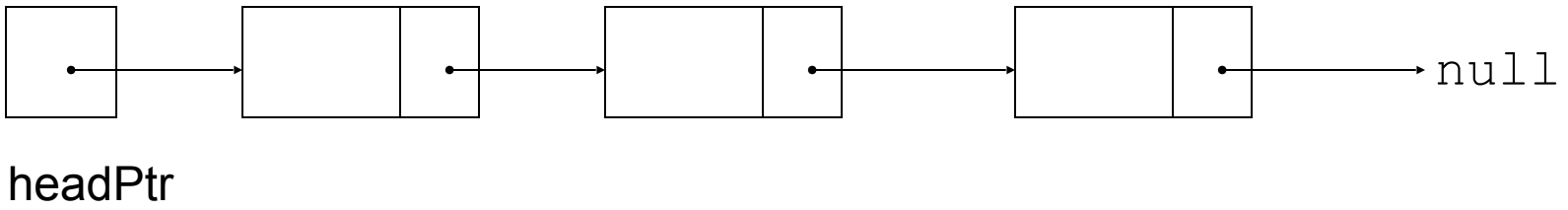
Linked Lists

18.1

Introduction to the Linked List ADT

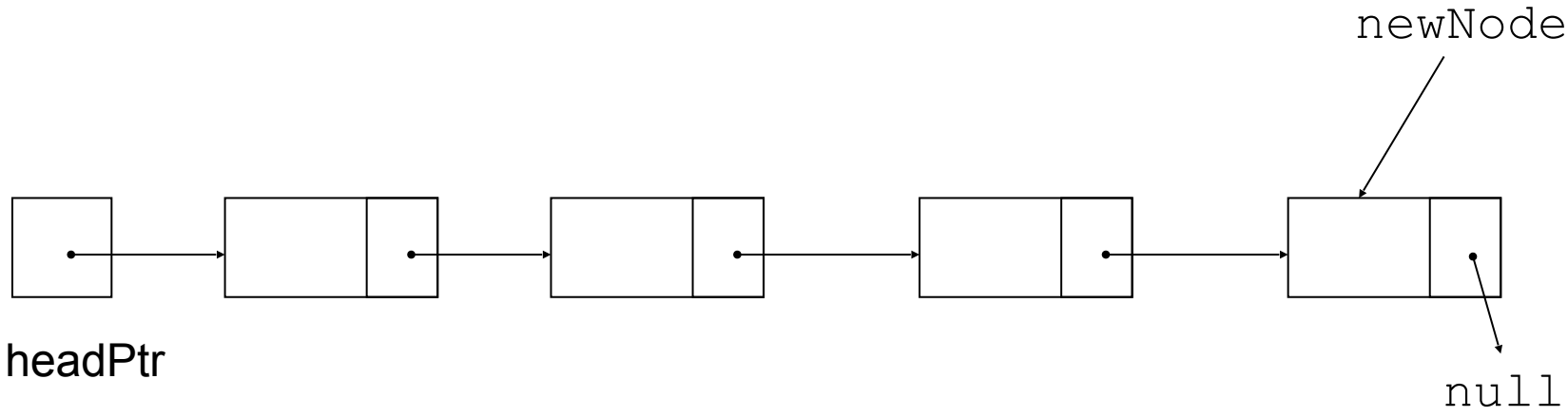
Introduction to the Linked List ADT

- Linked list is a linear collection of data structures (nodes) whose order is not given by their physical placement in memory.
  - Each element points (references) to the next element.
  - It is a collection of nodes which together represent a sequence.



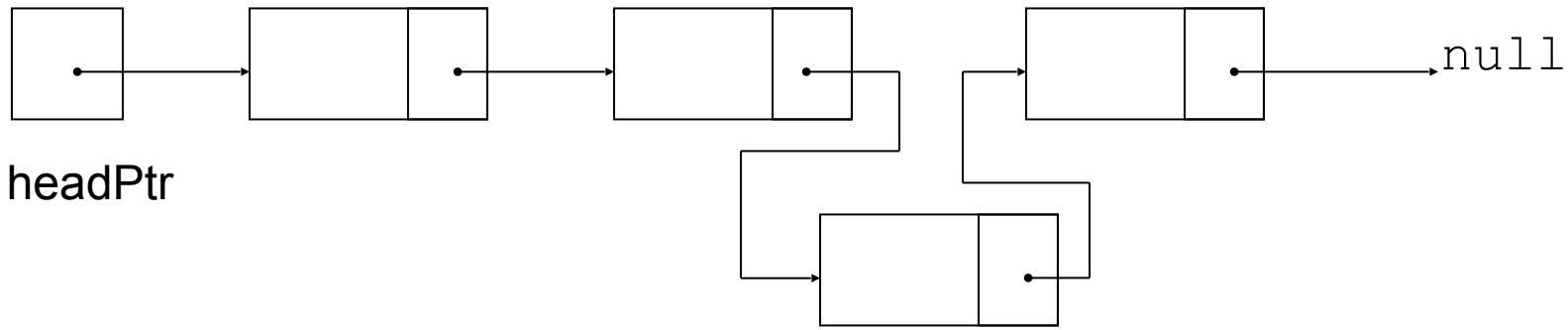
Introduction to the Linked List ADT

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



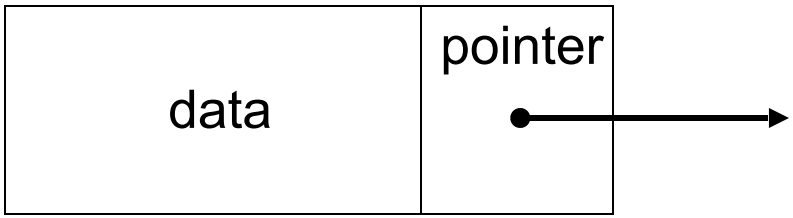
Linked Lists vs. Arrays and Vectors

- 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



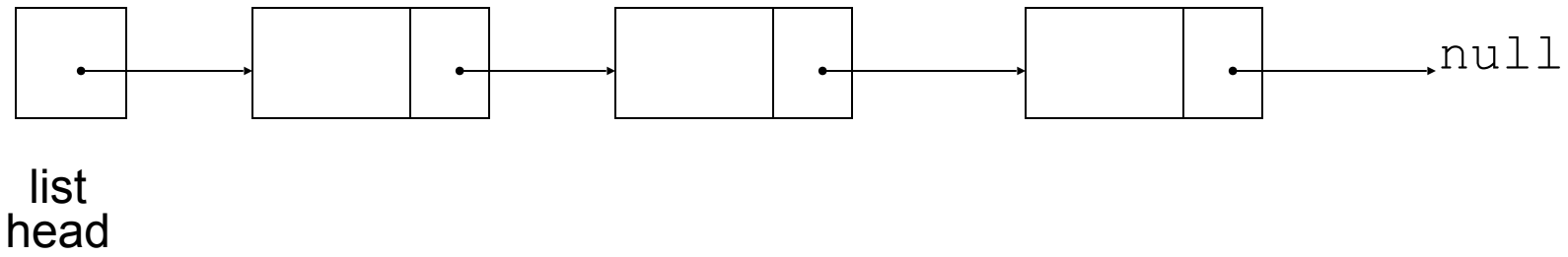
Node Organization

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



Linked List Organization

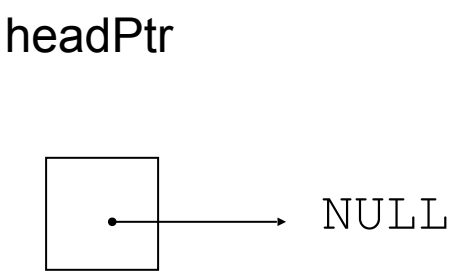
- Linked list contains 0 or more nodes:



  - Has a list head to point to the 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 Node
{
    int value;
    Node *next;
};
```

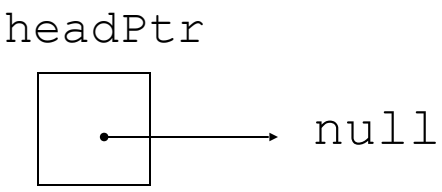
- A node has a data field, and a pointer to the next node. The pointer field acts as a link.

Defining a Linked List

- Define a pointer for the head of the list:

```
Node *head = nullptr;
```

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



The Null Pointer

- nullptr is used to indicate end-of-list
- We should always test whether a given node is nullptr before using it in the program:

```
Node *p = headPtr;
while (p != nullptr){
  p = p->next;
}
```

18.2

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 all the nodes in the list

Linked List Operations

```
class NumberList
{   private:

// Declare a structure for the list
    struct Node
    {
        double value;           // The value in this node
        struct Node *next;  // To point to the next node
    };

    Node *headPtr;          // List head pointer

public:
    NumberList()
    { headPtr = nullptr; }
// Destructor
    ~NumberList();

// Linked list operations

    void appendNode(double);
    void insertNode(double);
    void deleteNode(double);
    void displayList() const;
};
```

Create a New Node

- 1. Allocate memory for the new node:

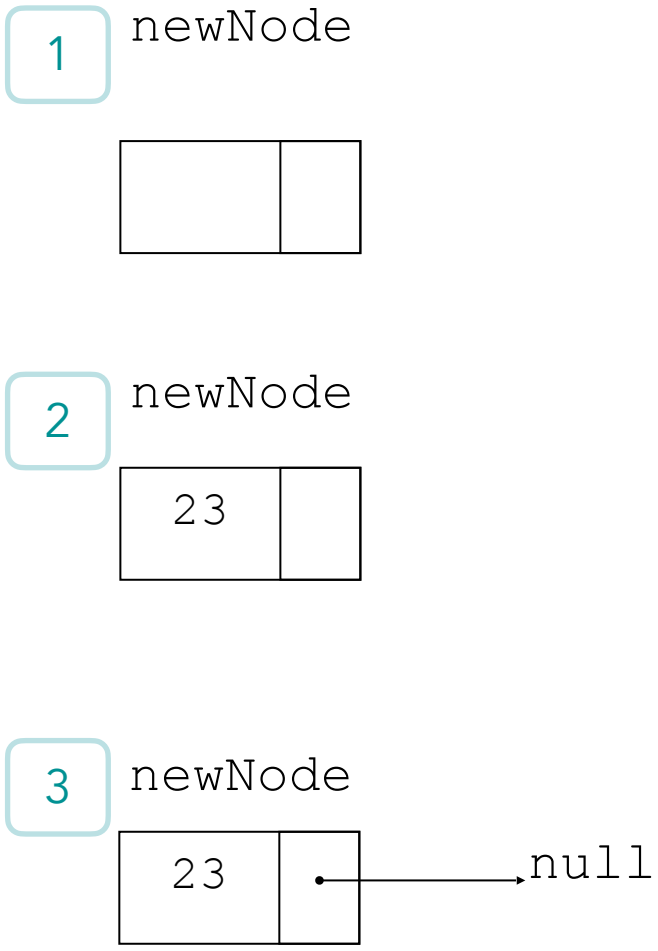
```
newNode = new Node;
```

- 2. Initialize the contents of the node:

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

- 3. Set the pointer field to `nullptr`:

```
newNode->next = nullptr;
```

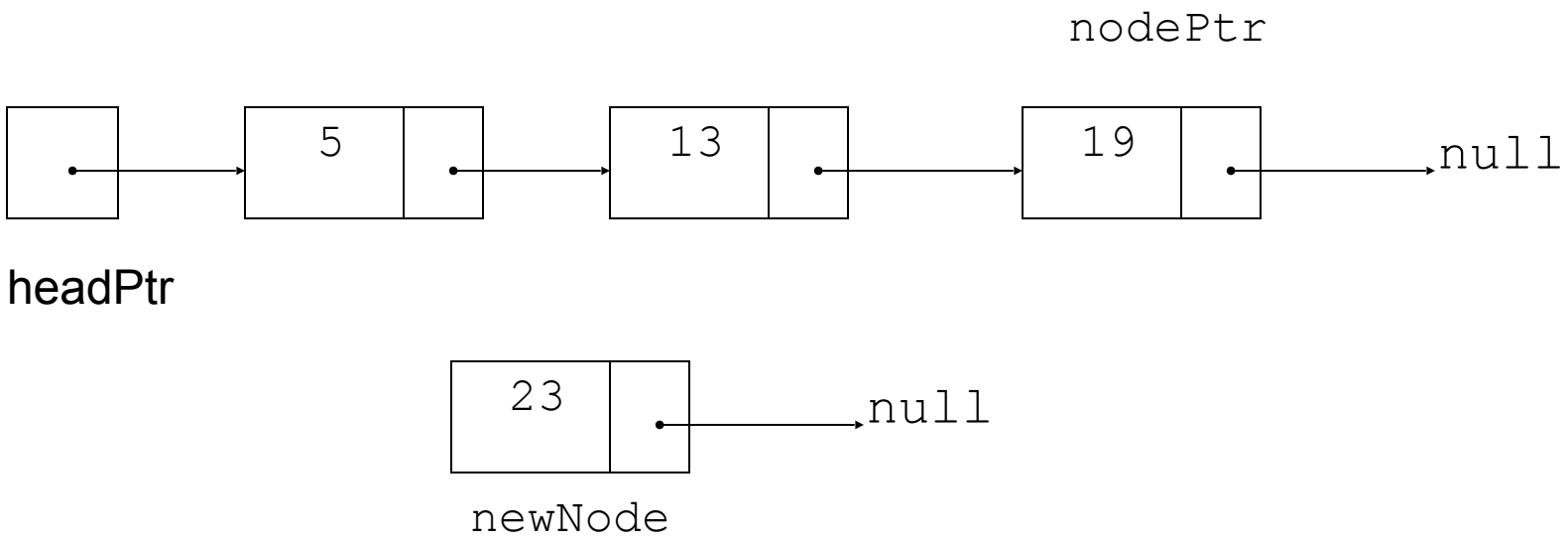


Appending a Node

- Add a node to the end of the list

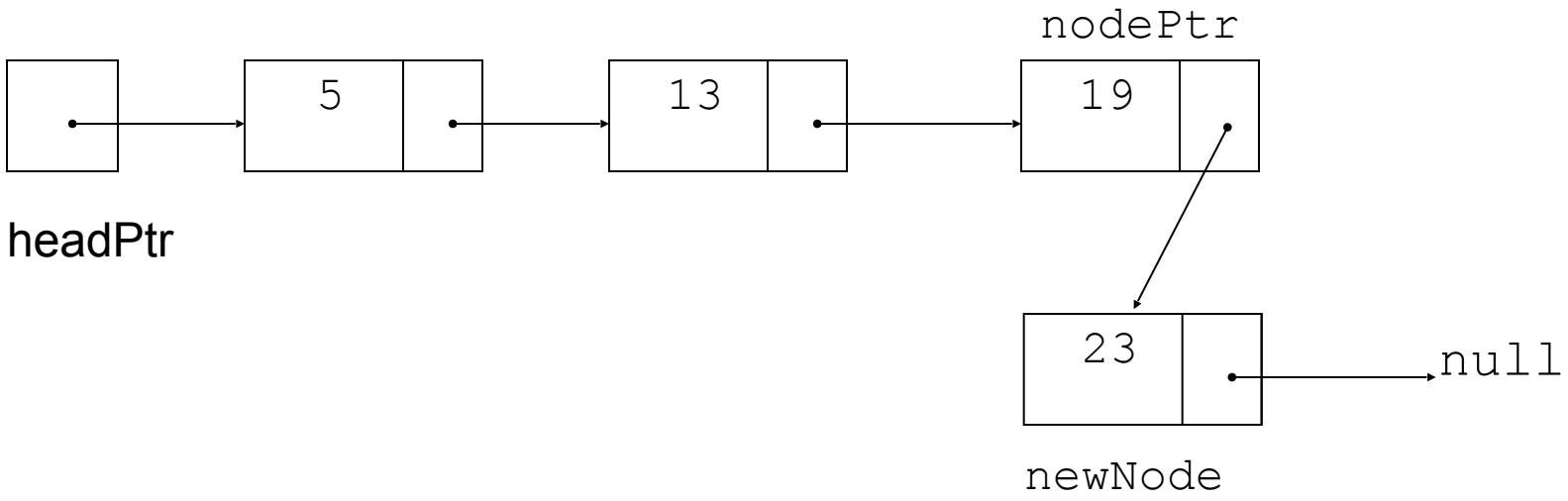
- Basic process:
  - Create the newNode (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else
      - traverse to the ends of the list
      - make the last node to point to the new node

Appending a Node



New node created, end of list located

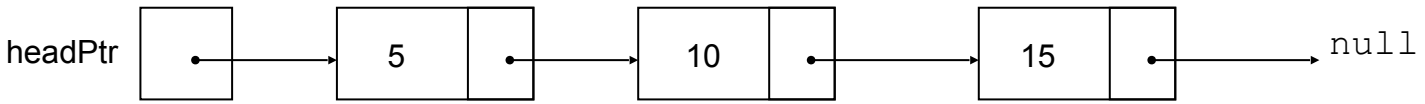
Appending a Node



New node added to end of list

C++ code for Appending a Node

```
void append(double data)
{
    Node *newNode = new Node;
    newNode->value = data;
    newNode->next = nullptr;

    // is the list empty, then headPtr should point to the newNode
    if (headPtr == nullptr)
        headPtr = newNode;
    else // otherwise, navigate to the last node of the list
    {
        Node *currentPtr = headPtr;
        while (currentPtr->next != nullptr)
            currentPtr = currentPtr->next;

        // make the last node link to the newNode
        currentPtr->next = newNode;
    }
}
```



Program 18-1

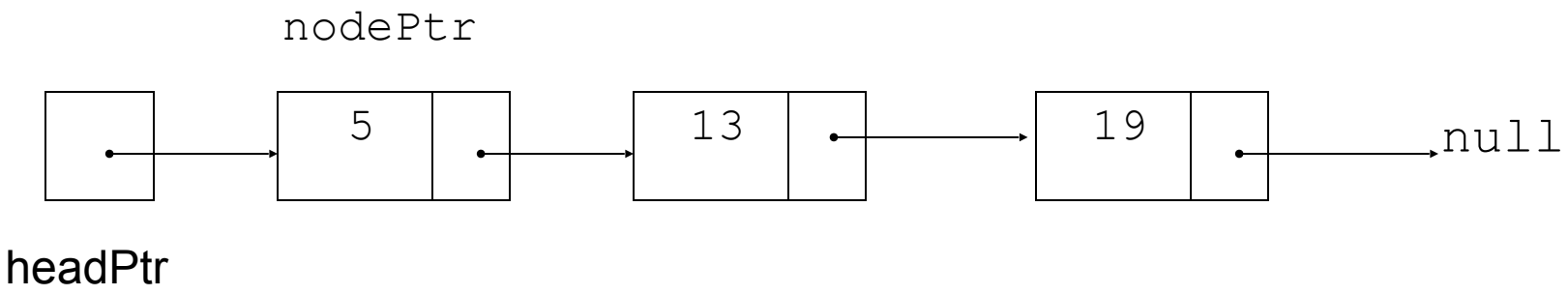
```
1  // This program demonstrates a simple append
2  // operation on a linked list.
3  #include <iostream>
4  #include "NumberList.h"
5  using namespace std;
6
7  int main()
8  {
9     // Define a NumberList object.
10    NumberList list;
11
12    // Append some values to the list.
13    list.appendNode(2.5);
14    list.appendNode(7.9);
15    list.appendNode(12.6);
16    return 0;
17 }
```

(This program displays no output.)

Traversing a Linked List

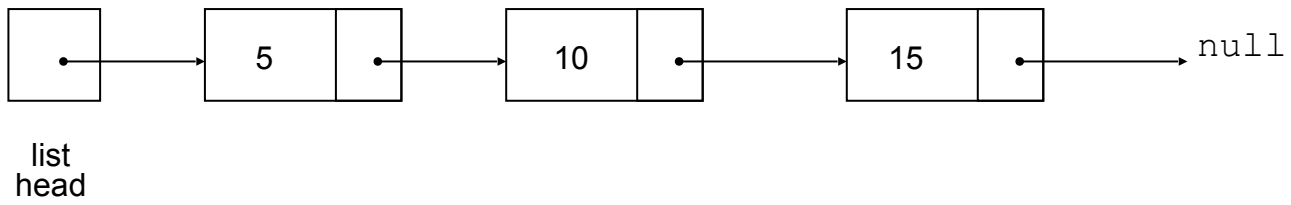
- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not a null pointer
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the 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

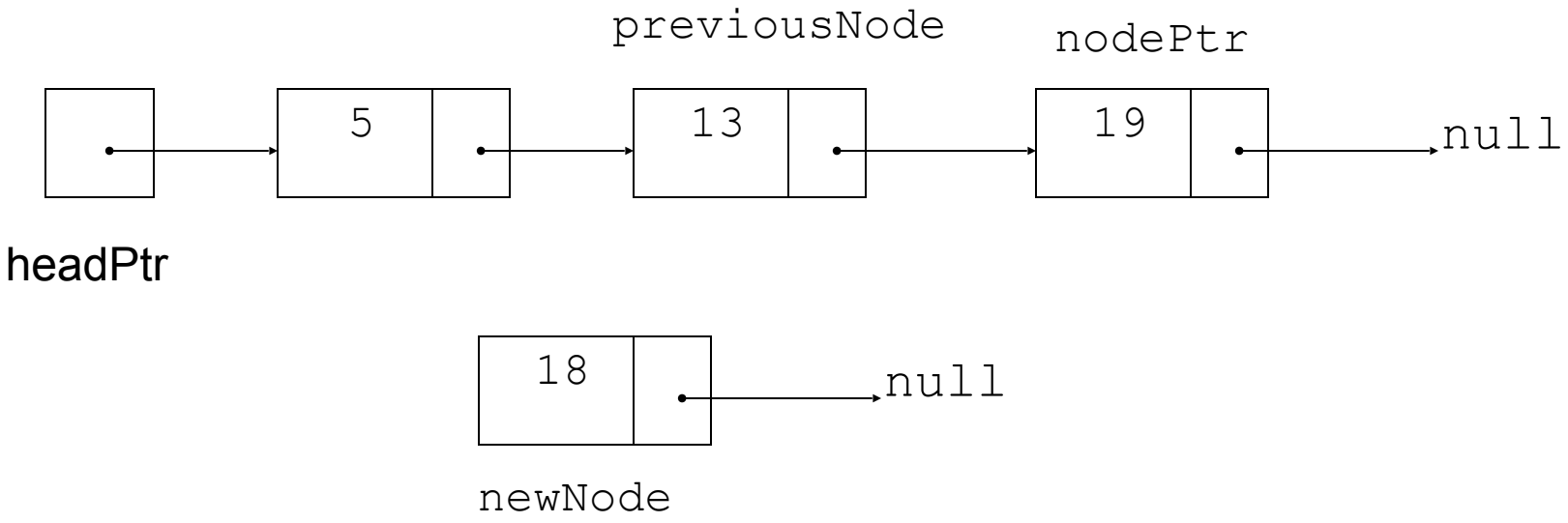
```
// Traverse through all the elements of the list


void NumberList::traverse()
{
    Node * currentNode;
    // Initialize nodePtr to head of list.
    currentNode = head;

    // Does the element pointed by the currentNode exist?
    while (currentNode != nullptr)
    {
        //Display the current element
        cout<< currentNode->value <<" ";

        // Navigate to the next element
        currentNode = currentNode->next;
    }
}
```



Inserting a Node into a Linked List

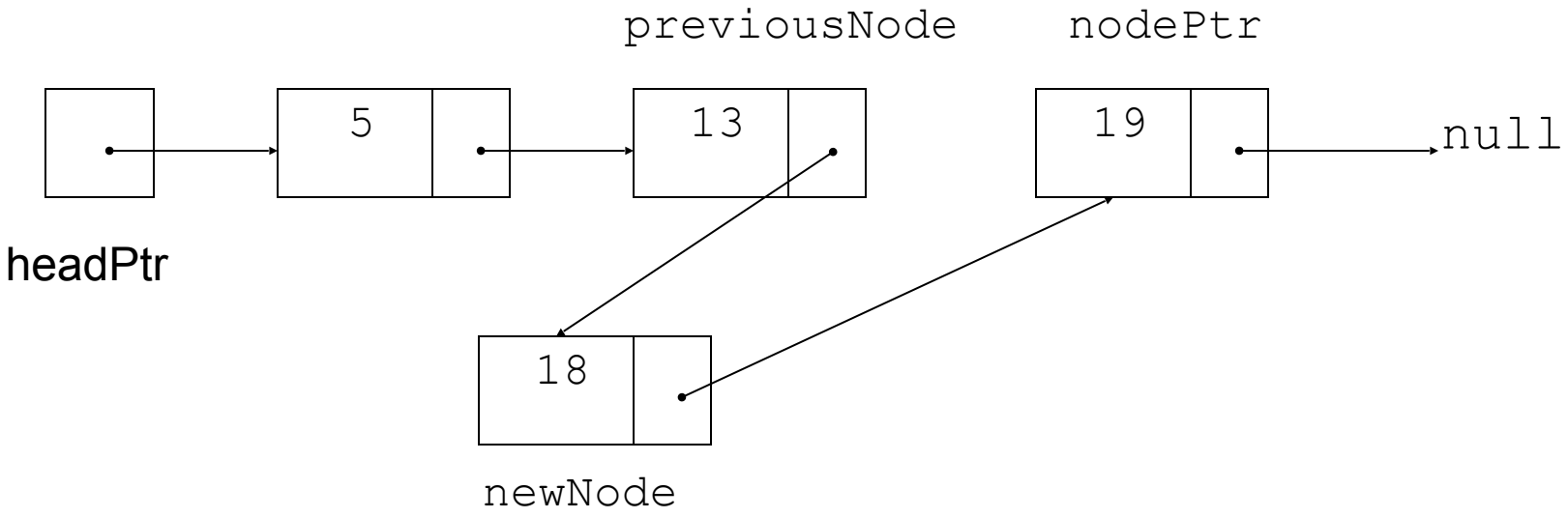
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed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

Thank You

Please send your questions by email!