CHAPTER 18

LINKED LISTS

WARM UP

What do we mean when we say “linked list”?

- Abstract data type
- Nodes linked together
- Head pointer designates beginning of the list

What do we mean when we “append” a node to the list?

- Add at the end

How is the linked list created?

- Individual nodes are allocated dynamically
- Added to the list

How is the linked list destroyed once it is not needed?

- Individual nodes must be visited and memory de-allocated (via the delete statement)

PASSING PHEAD TO A FUNCTION

```
void createList(Node* &, ifstream &);
void addNode(Node* &, Node*);

void createList(Node* &pHead, ifstream &infile)
{
        Node* pNew;
        if (infile.is_open())
        {
                while (!infile.eof())
                {

                }
                infile.close();
        }
}
```

Create a new node
Initialize the new node with data
Add new node to the list

PASSING PHEAD TO A FUNCTION

```
void createList(Node* &, ifstream &);
void addNode(Node* &, Node*);

void createList(Node* &pHead, ifstream &infile)
{
        Node* pNew = new Node;
        pNew->pNext = nullptr;
        if (infile.is_open())
        {
                while (!infile.eof())
                {
                        infile >> pNew->data;
                        addNode(pHead, pNew);
                }
                infile.close();
        }
        else
                cout << "Error, no file found";
}
```

Memory leak
If file not opened

List with only
one node created

PASSING PHEAD TO A FUNCTION

```
void createList(Node* &, ifstream &);
void addNode(Node* &, Node*);

void createList(Node* &pHead, ifstream &infile)
{
        Node* pNew;
        if (infile.is_open())
        {
                while (!infile.eof())
                {
                        pNew = new Node;
                        pNew->pNext = nullptr;
                        infile >> pNew->data;
                        addNode(pHead, pNew);
                }
                infile.close();
        }
        else
                cout << "Error, no file found";
}
```

CHAPTER 18

LINKED LISTS

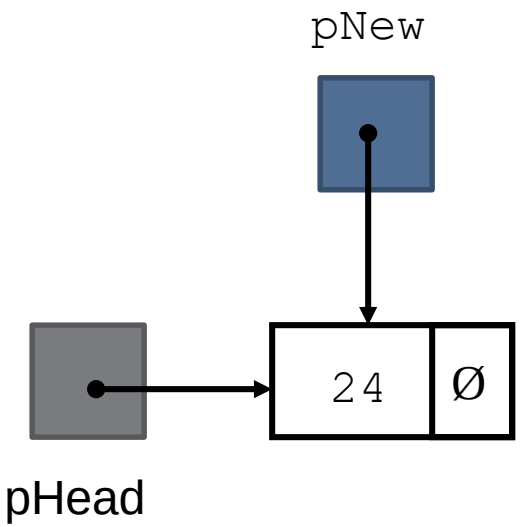
LINKED LIST OPERATIONS

Basic operations:

- append (add at the end)
- prepend (add at the beginning)
- insert (add in the middle)
- destroy the list (deallocate memory)
- delete a node (various locations in the list)
- traverse the linked list (find specific item)

CREATING A NODE

```
Node* pHead = nullptr;
Node* pNew = new Node;
pNew->data = 24;
pNew->pNext = nullptr;

pHead = pNew;
```



APPENDING

1. Append when list is empty

How do we know the list is empty?

```
if (pHead == nullptr)
      // insert first node
      pHead = pNew;
else
      ???
```

APPENDING

1. Append when list is not empty…
   How do we know where the end is?

```
else
{
        pTemp = pHead;
        while (pTemp->pNext != nullptr)
                pTemp = pTemp->pNext;

        // append new node…
        pTemp->pNext = pNew;
}
```

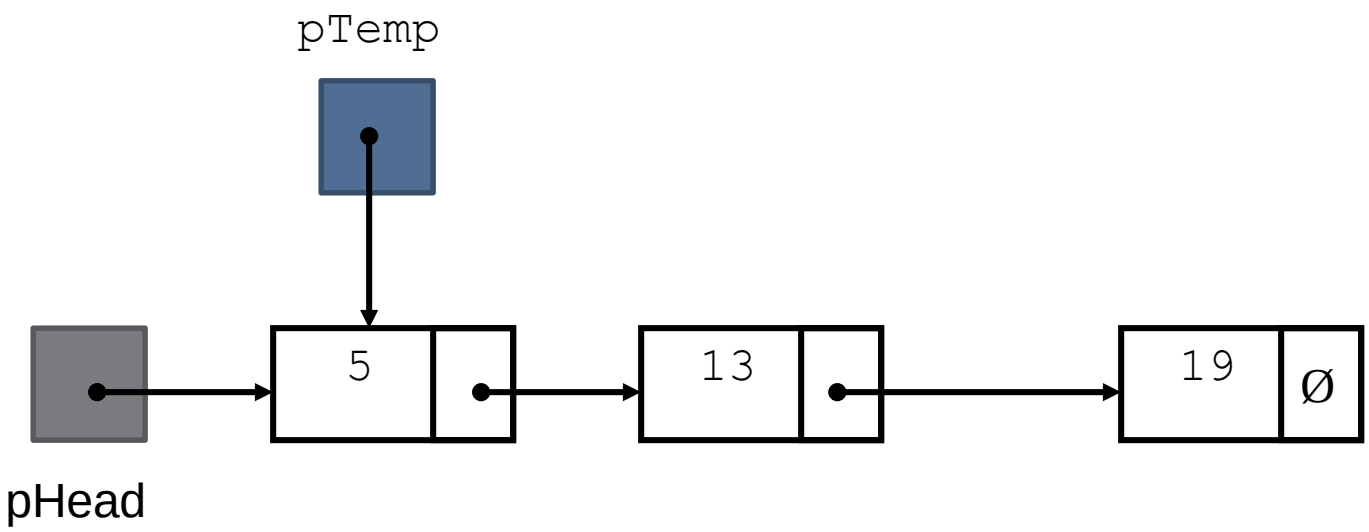
TRAVERSING A LINKED LIST

Visit each node in a linked list: display contents, validate data, etc.

Basic process:

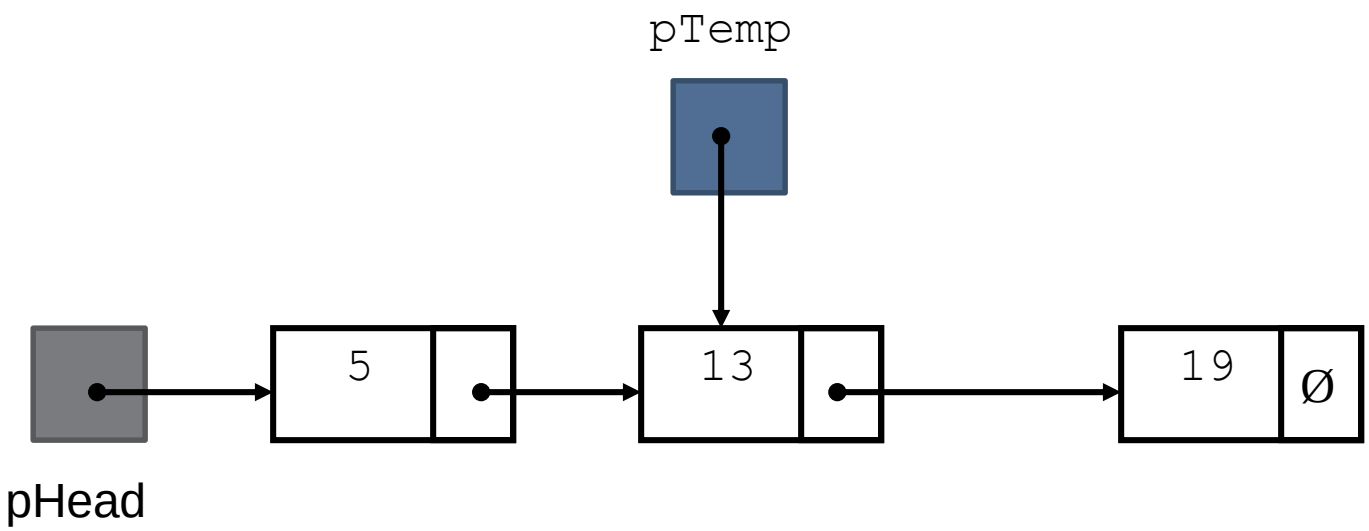
- set a temporary pointer to the contents of the head pointer
- while the temporary pointer is not null
  - do something (can be nothing, process data in current node, etc.)
  - go to the next node by setting the temporary pointer to the pointer field of the current node in the list (pNext)

TRAVERSING A LINKED LIST



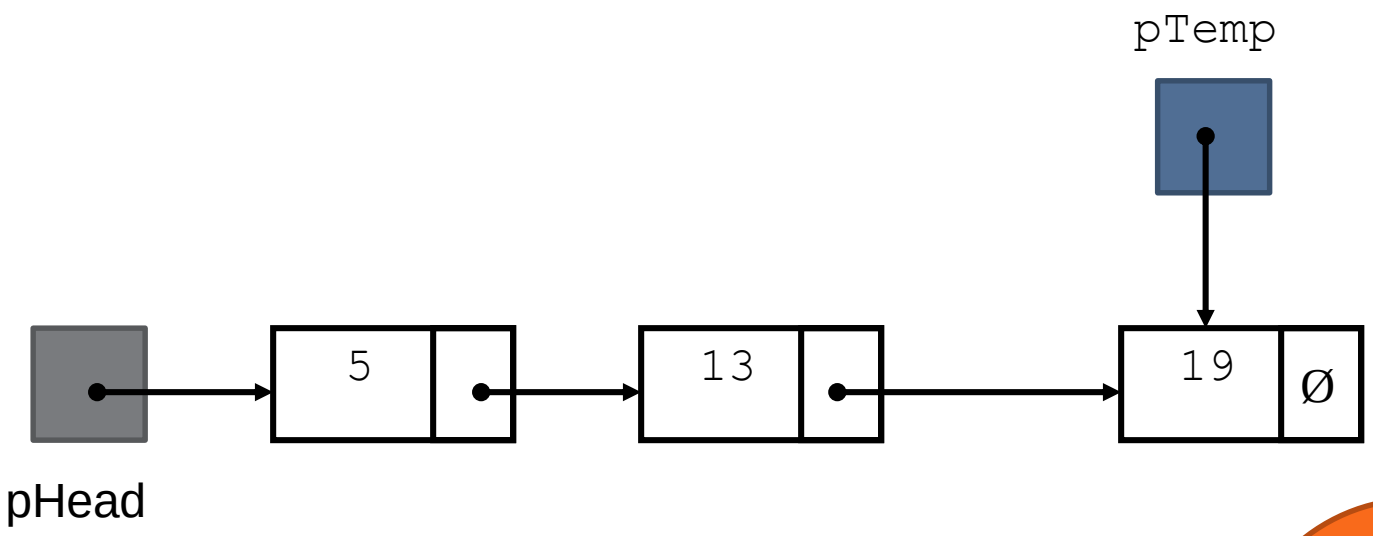
`pTemp` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then the list traversal stops

TRAVERSING A LINKED LIST



`pTemp` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then the list traversal stops

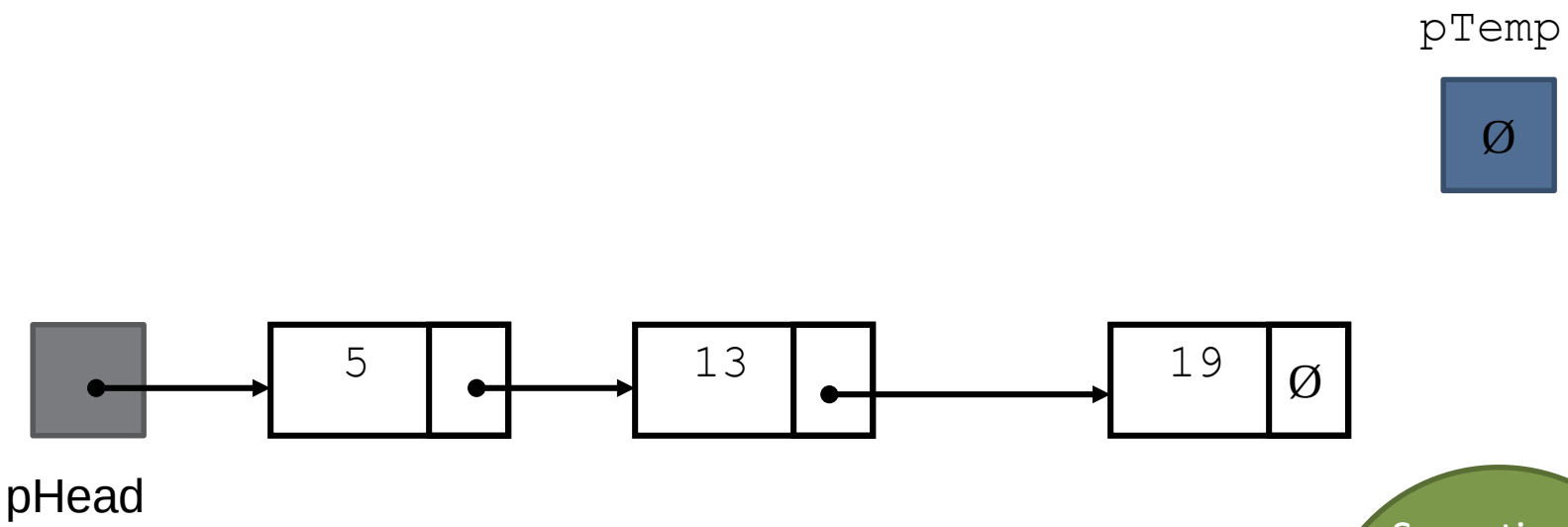
TRAVERSING A LINKED LIST



`pTemp` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then the list traversal stops

Sometimes we want to stop on last node

TRAVERSING A LINKED LIST



Sometimes we want to stop when pTemp is null

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

TRAVERSE A LINKED LIST – TWO WAYS

```
Node* pTemp = pHead;

// stop when pNext of node pointed to by pTemp is null
// pTemp points to the LAST node in the list
while (pTemp->pNext != nullptr)
   pTemp = pTemp->pNext;


// stop when pTemp is null
// pTemp "walks off" the end of the list
while (pTemp != nullptr)
   pTemp = pTemp->pNext;
```

ADDING NODES AND MAINTAINING ORDER

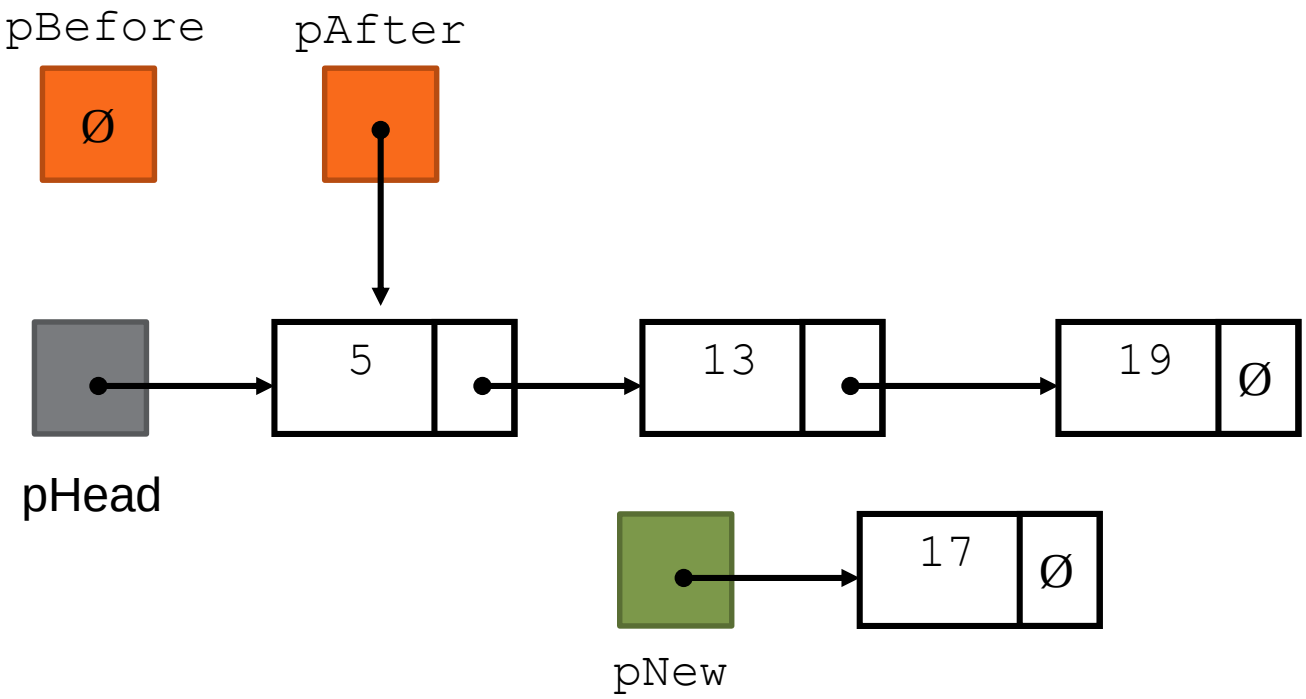
Used to maintain a linked list in some order (assume ascending)

Requires two pointers to traverse the list:

- pointer to locate the node with data value greater than that of node to be inserted… pAfter
- pointer to 'trail behind' one node, to point to node before the point of insertion… pBefore

New node is inserted between the nodes pointed to by these two pointers

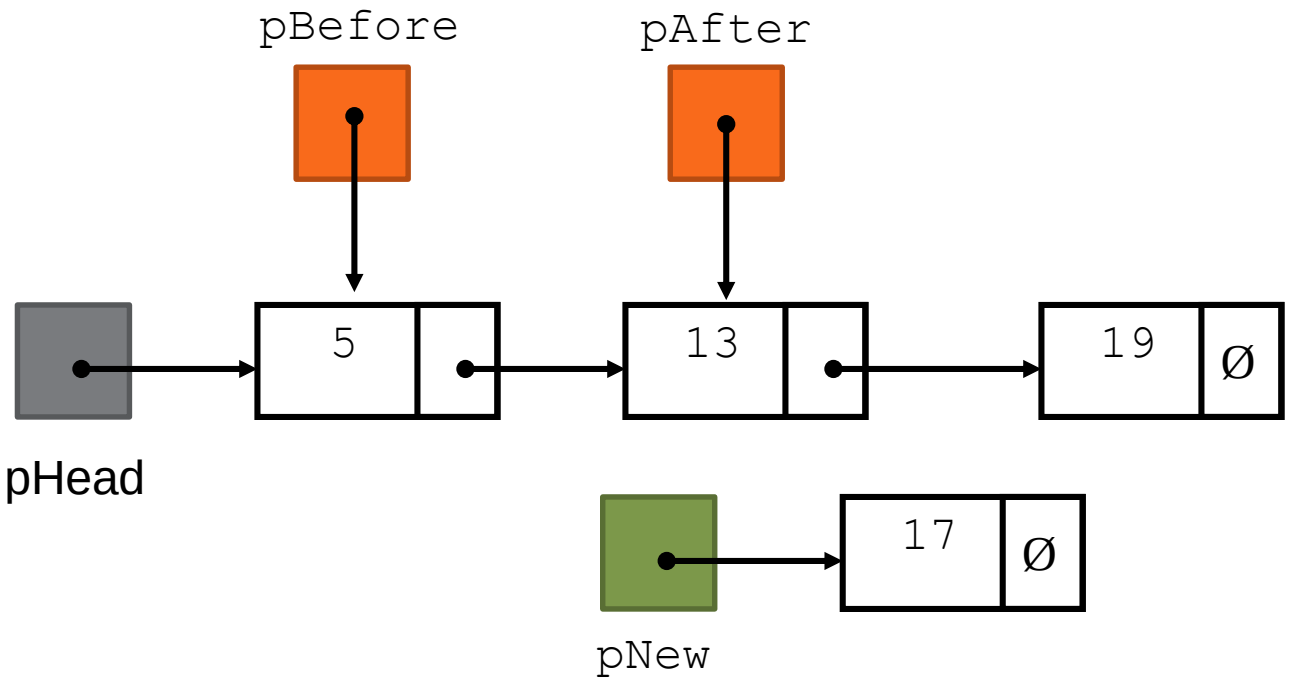
TRAVERSE TO FIND INSERTION POINT



New node created

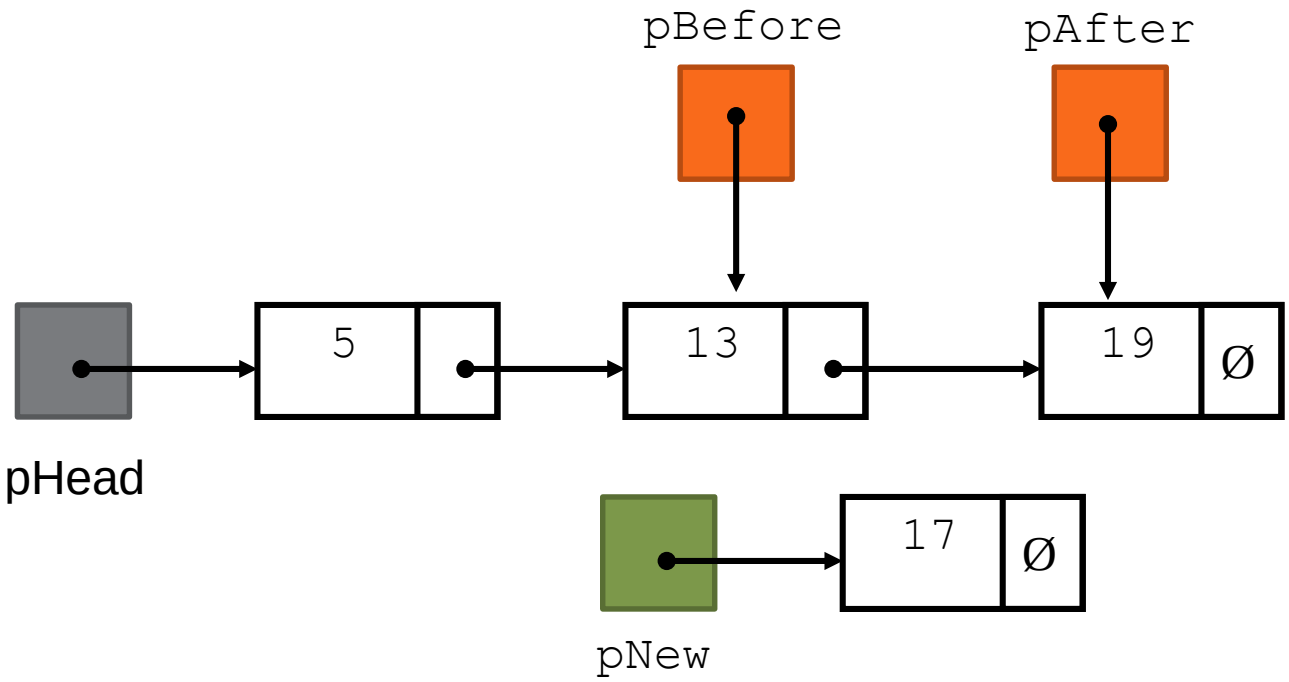
TRAVERSE TO FIND INSERTION POINT

```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}
```



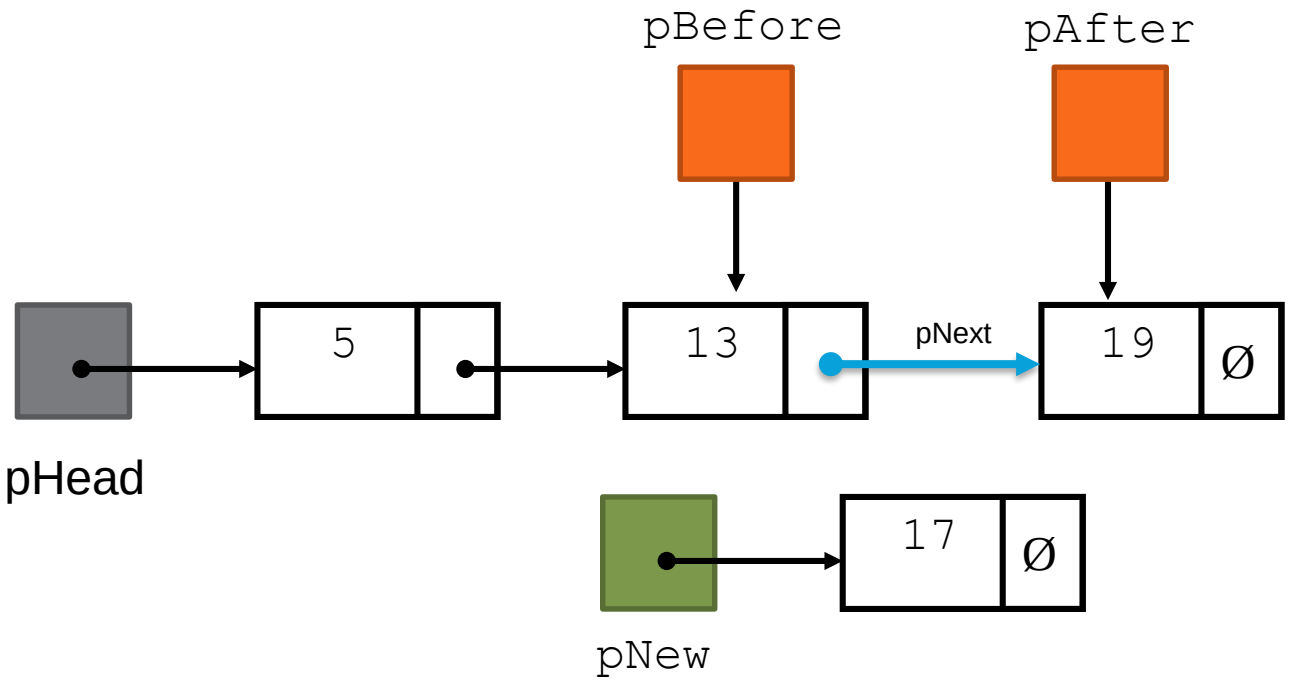
TRAVERSE TO FIND INSERTION POINT

```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}
```



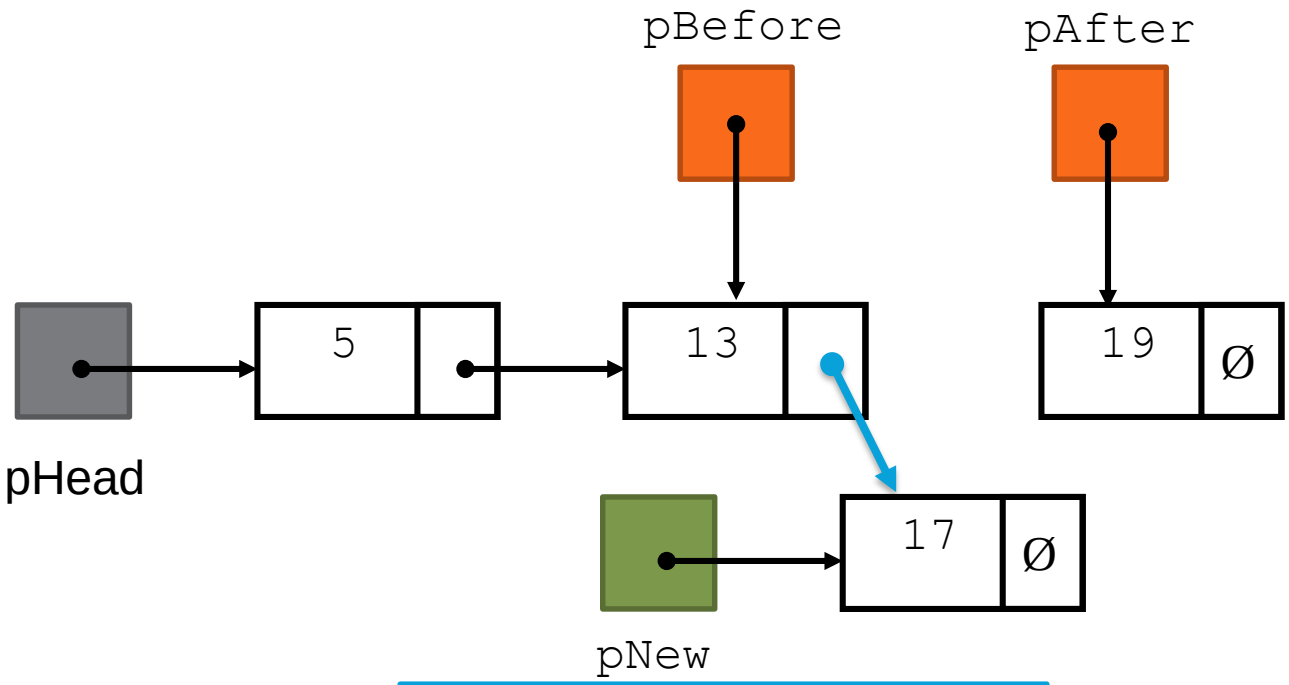
Insert position located

INSERT THE NEW NODE



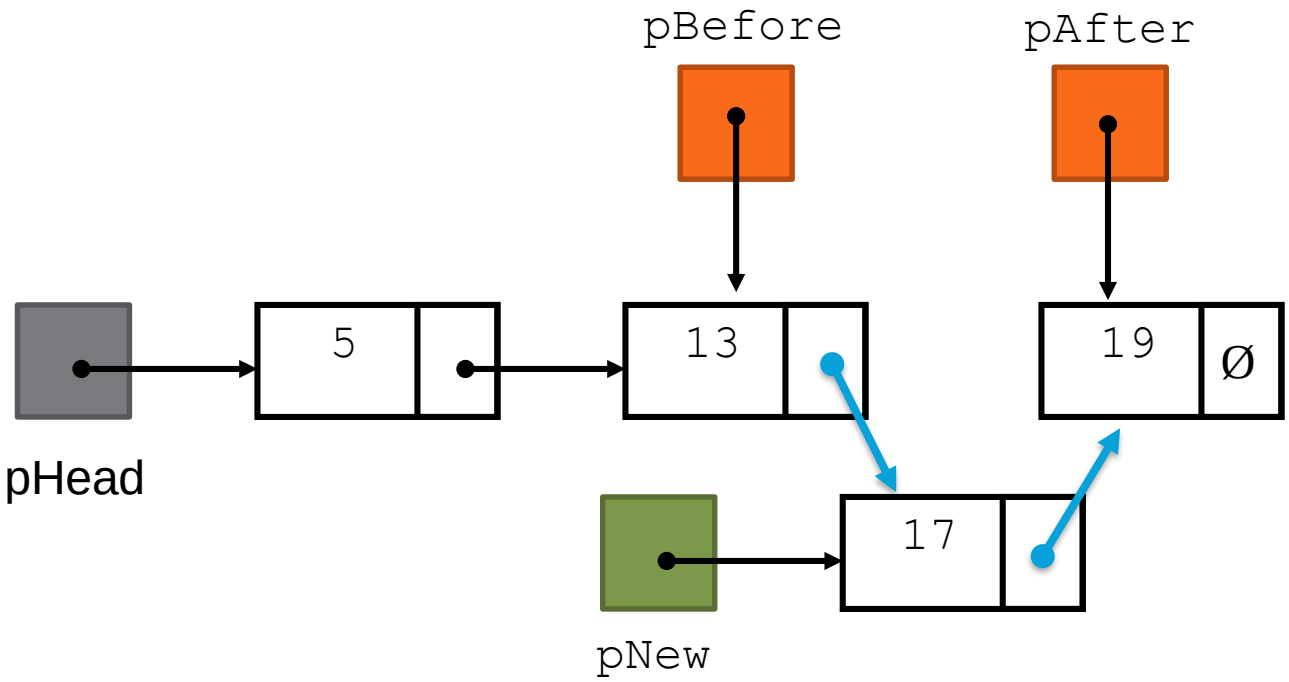
Inserting the new node

INSERT THE NEW NODE



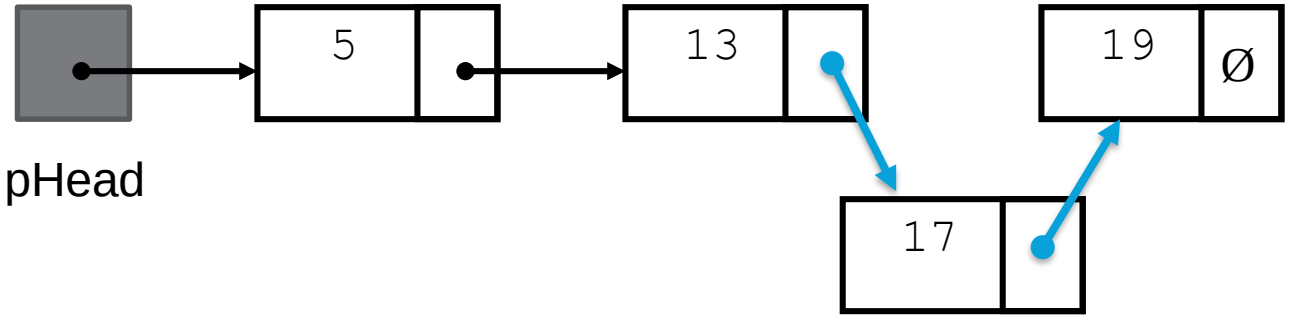

```
pBefore->pNext = pNew
```

TRAVERSE TO FIND INSERTION POINT



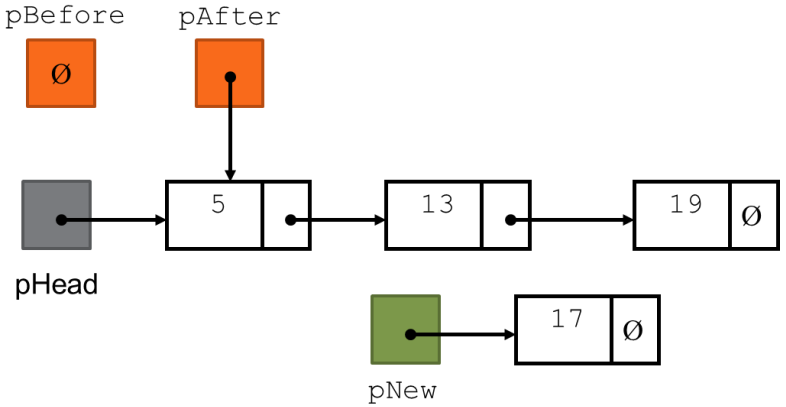

```
pNew->pNext = pAfter
```

TRAVERSE TO FIND INSERTION POINT



New node has been inserted

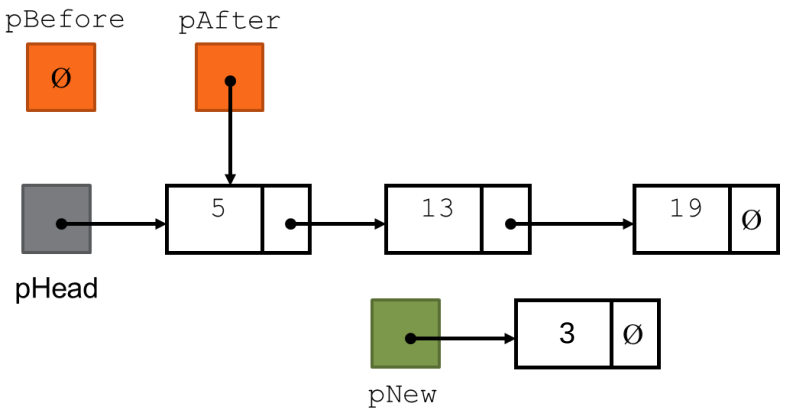
```
Node* pNew;
Node* pAfter;
Node* pBefore;

pNew = new Node;
pNew->data = 24;
pNew->pNext = nullptr;

if (!pHead)
        pHead = pNew;
else
{
        // start at the head
        pAfter = pHead;
        pBefore = nullptr;
```



TRUE && FALSE

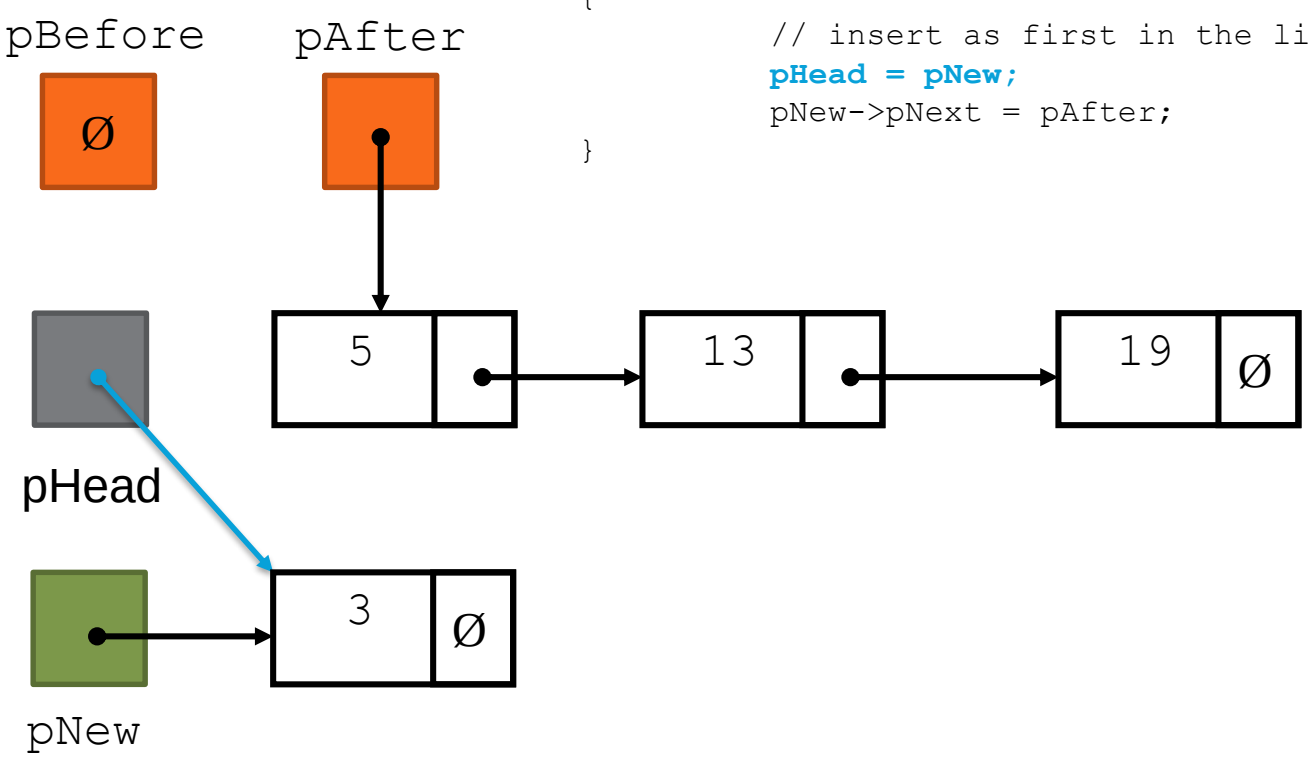
```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}

if (pBefore == nullptr)
{
        // prepend new node to the list
        pHead = pNew;
        pNew->pNext = pAfter;
}
else
{
        …
```

1



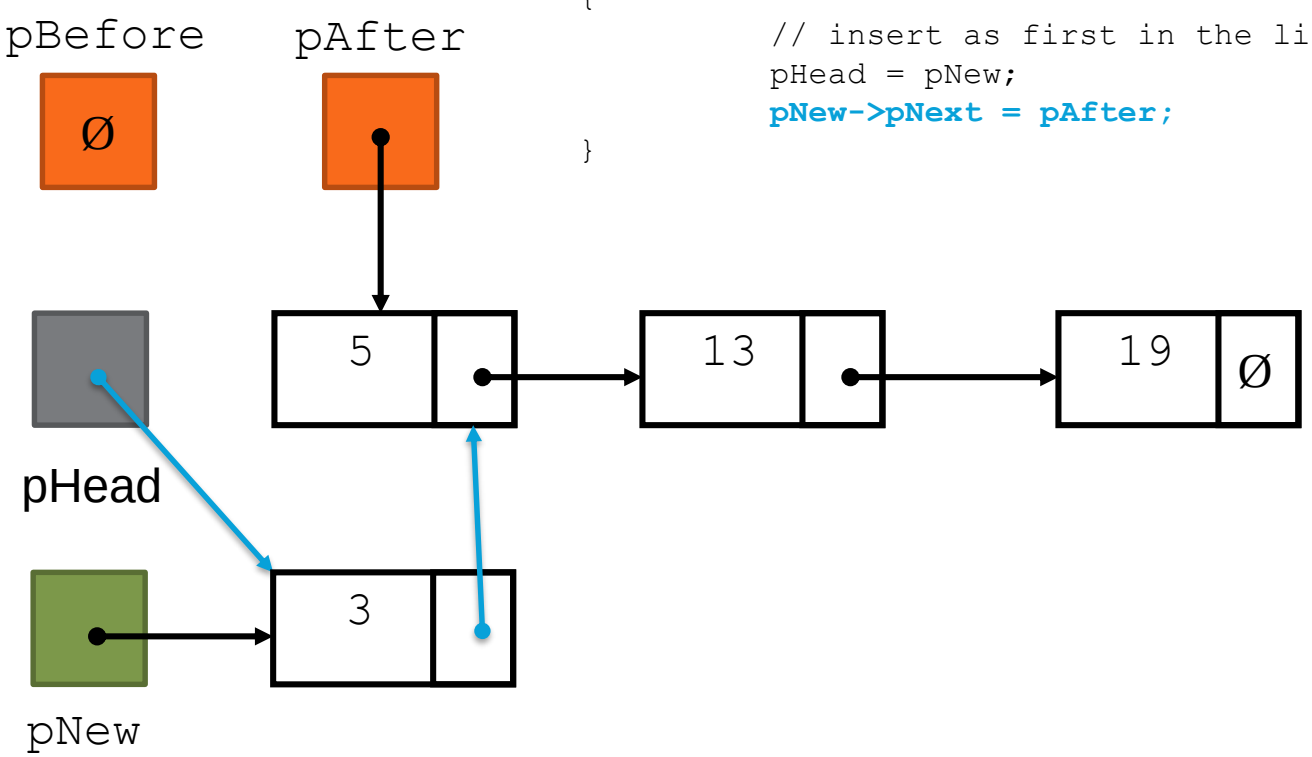
PREPEND NEW NODE

```
if (pBefore == nullptr)
{
        // insert as first in the list
        pHead = pNew;
        pNew->pNext = pAfter;
}
```

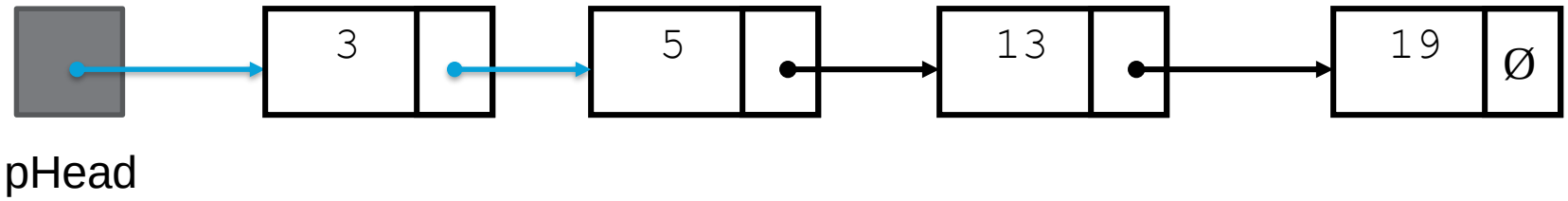


PREPEND NEW NODE

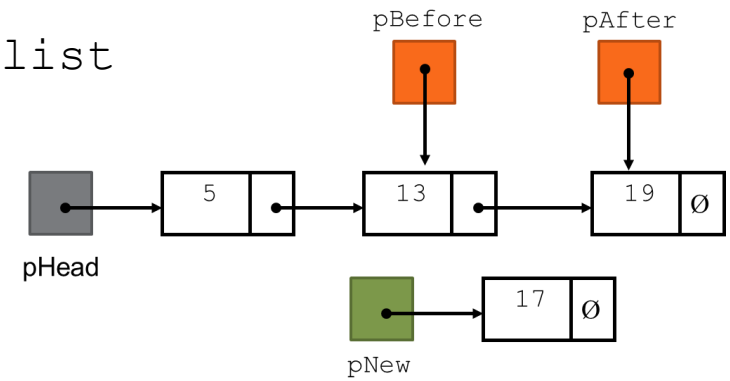
```
if (pBefore == nullptr)
{
        // insert as first in the list
        pHead = pNew;
        pNew->pNext = pAfter;
}
```



NODE HAS BEEN PREPENDED



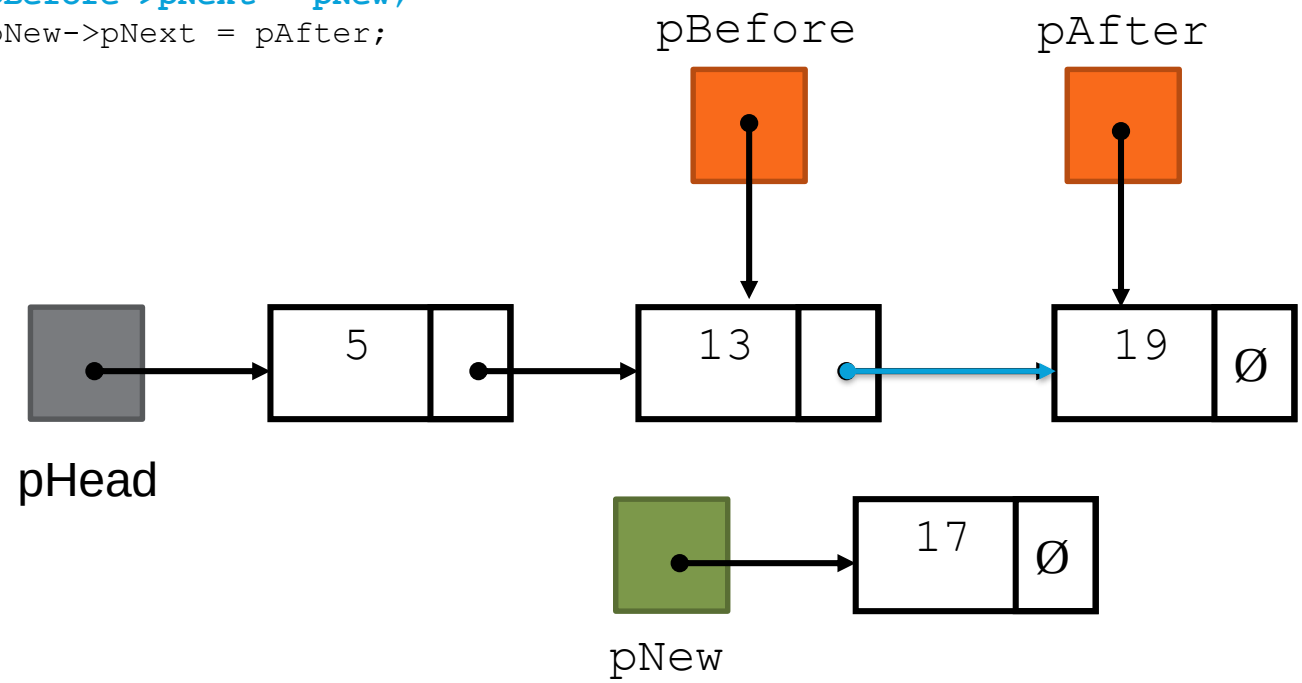
```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}

if (pBefore == nullptr)
{
        // insert as first in the list
        pHead = pNew;
        pNew->pNext = pAfter;
}
else
{
        // insert between pBefore and pAfter
        pBefore->pNext = pNew;
        pNew->pNext = pAfter;
}
```



2

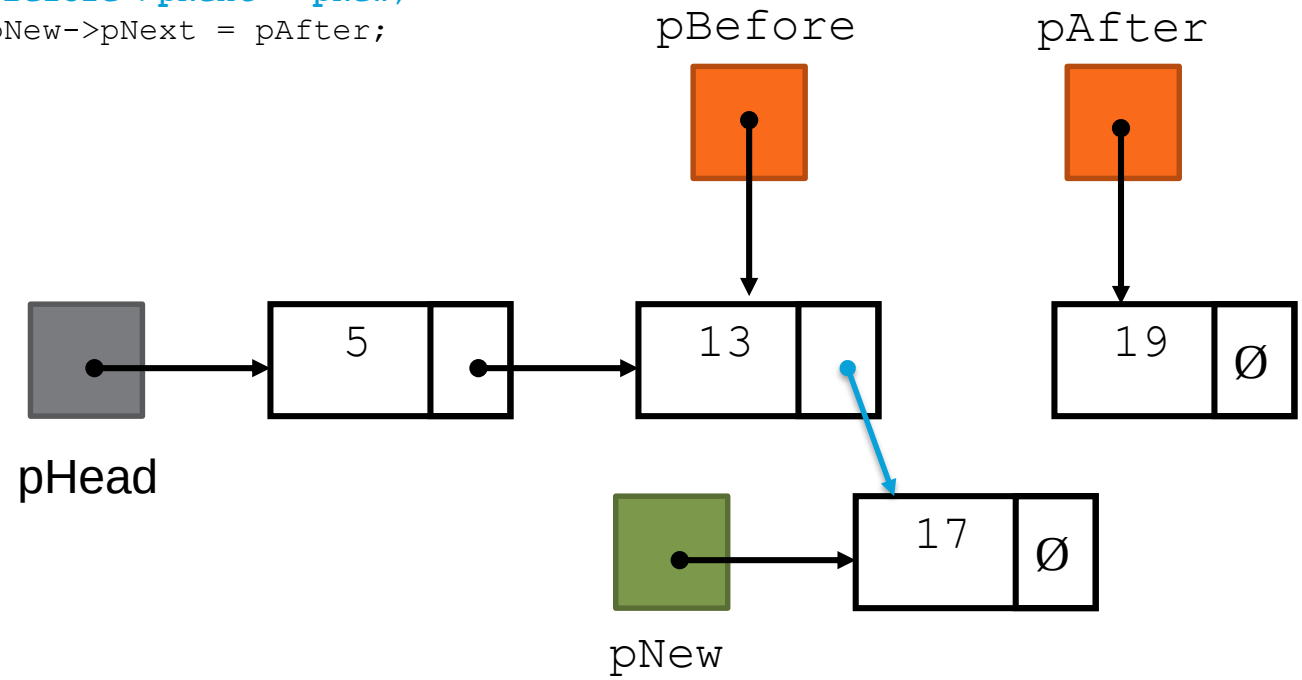
INSERT

```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



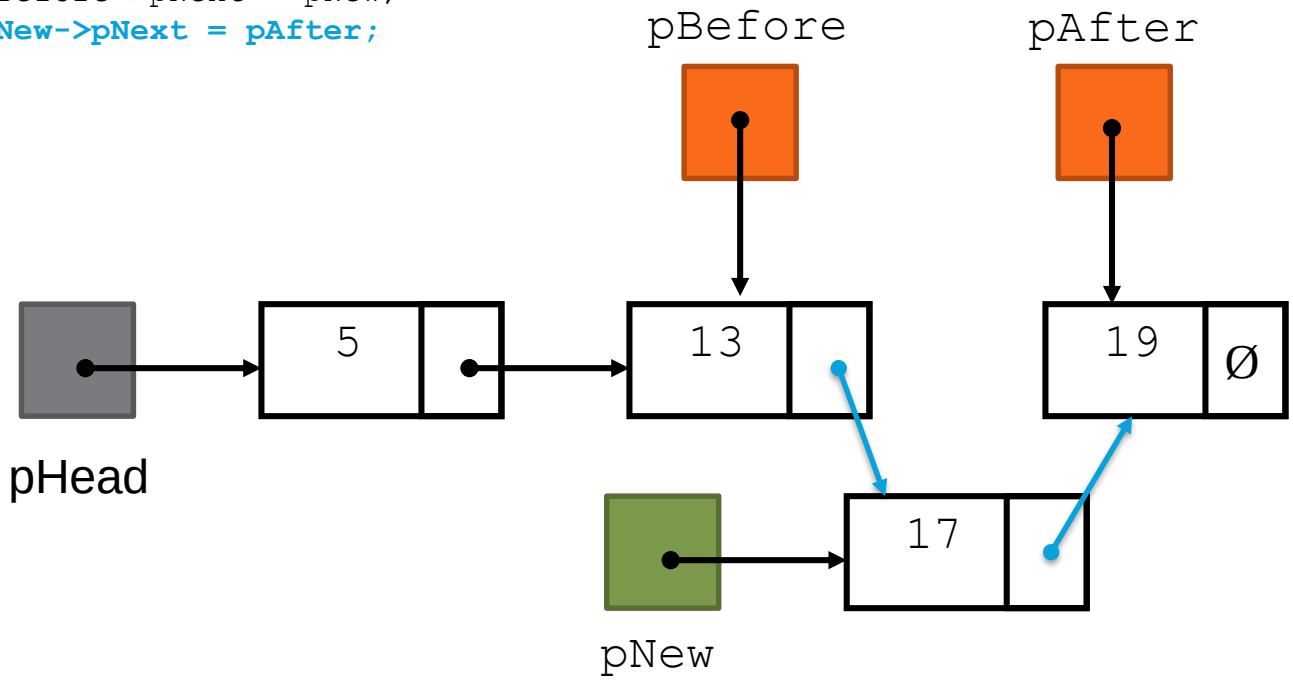
INSERT

```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



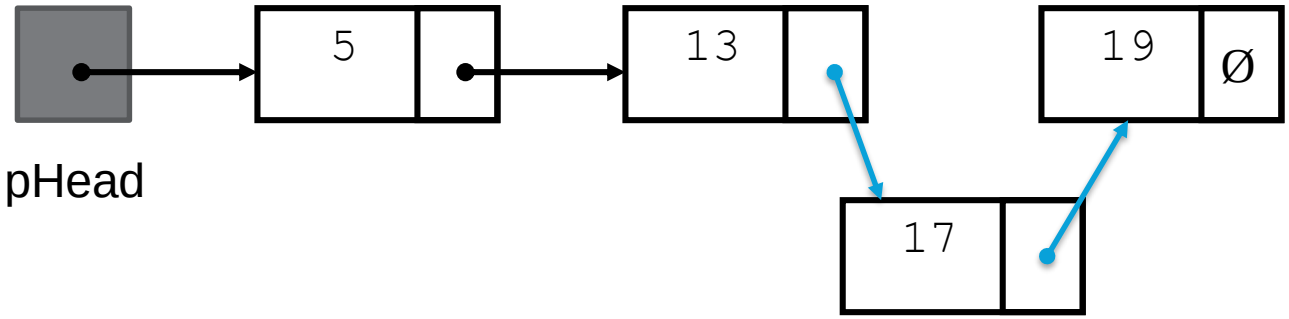
INSERT

```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



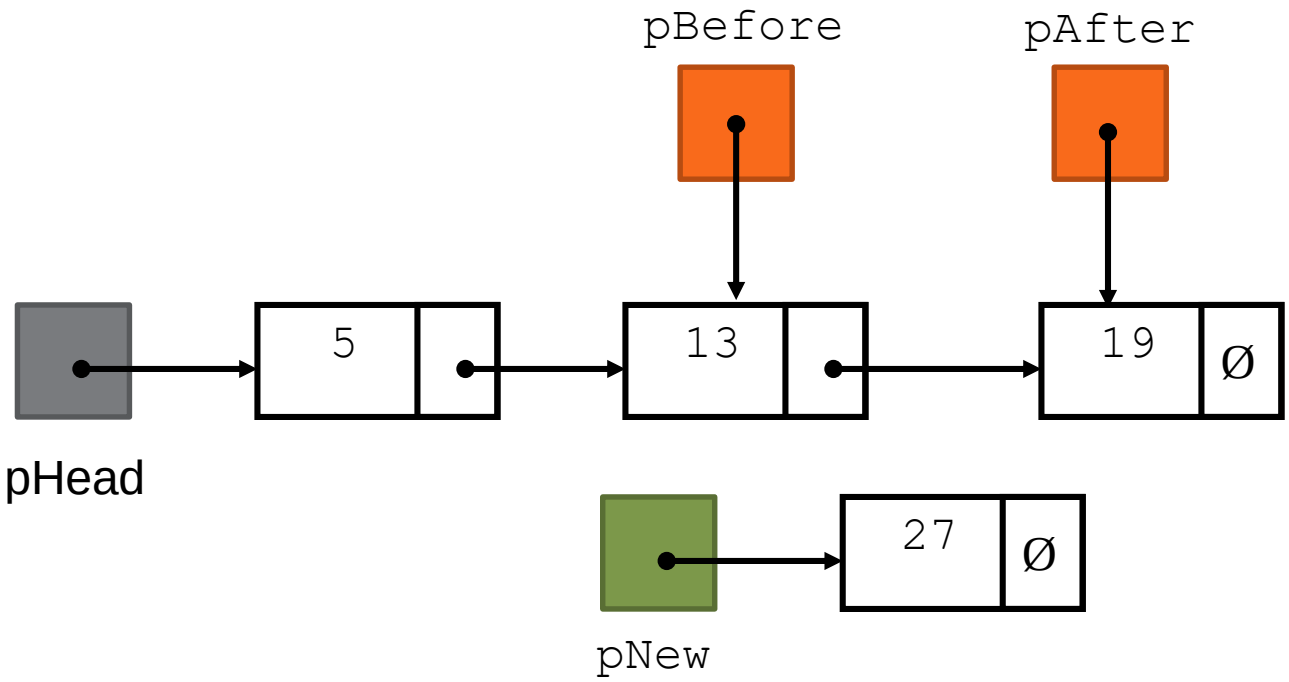
New node created; position located

INSERT



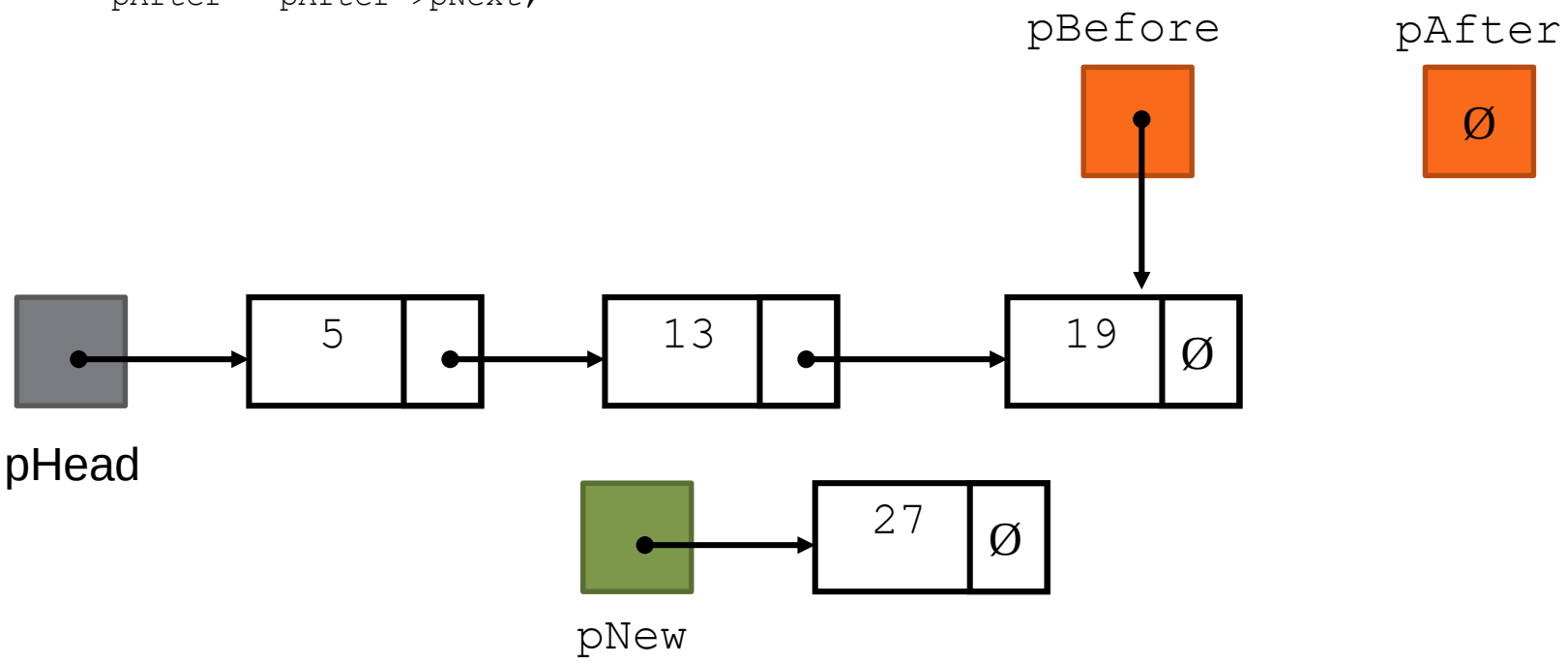
TRAVERSE TO FIND INSERTION POINT

```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}
```



TRAVERSE TO FIND INSERTION POINT

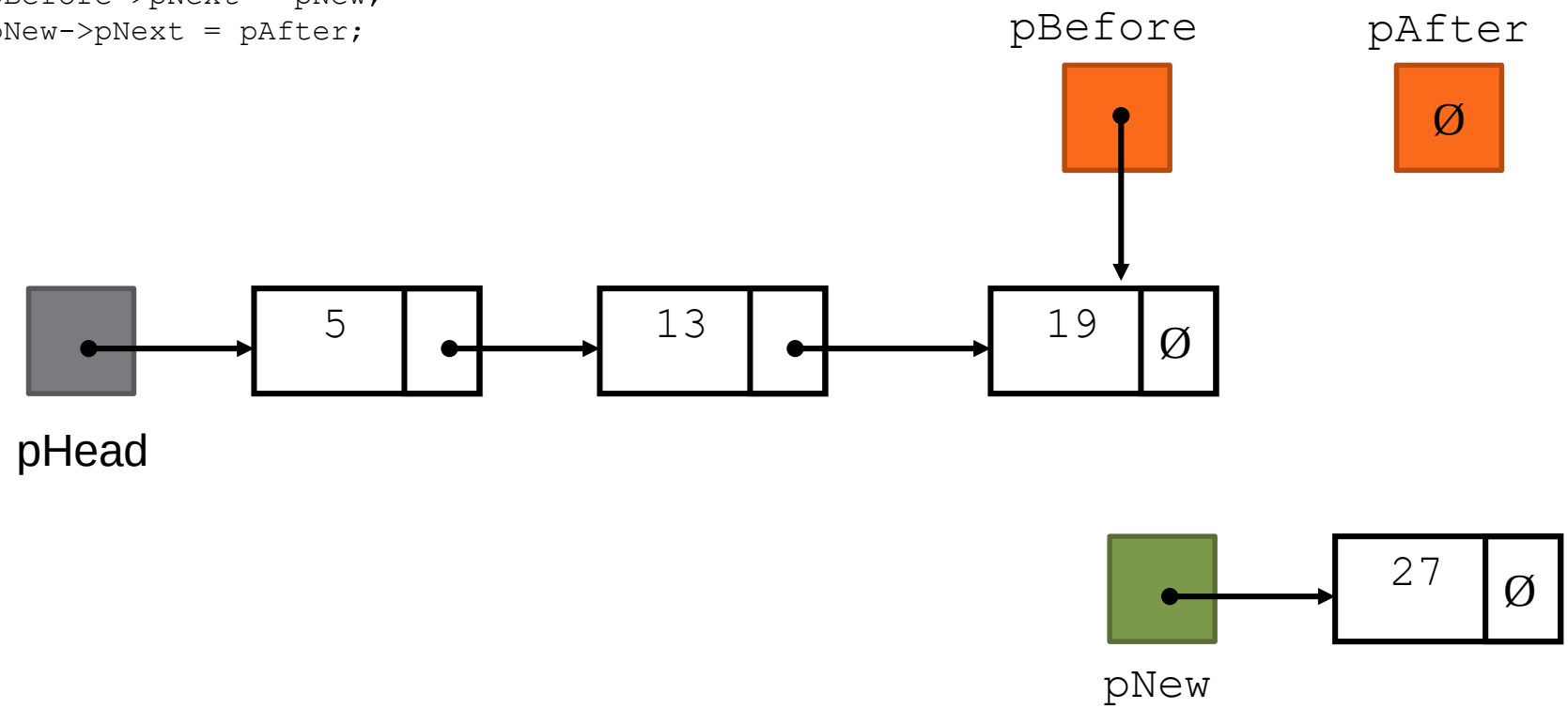
```
while (pAfter != nullptr && pAfter->data < pNew->data)
{
        pBefore = pAfter;
        pAfter = pAfter->pNext;
}
```



New node created; position located

APPEND AT INSERTION POINT

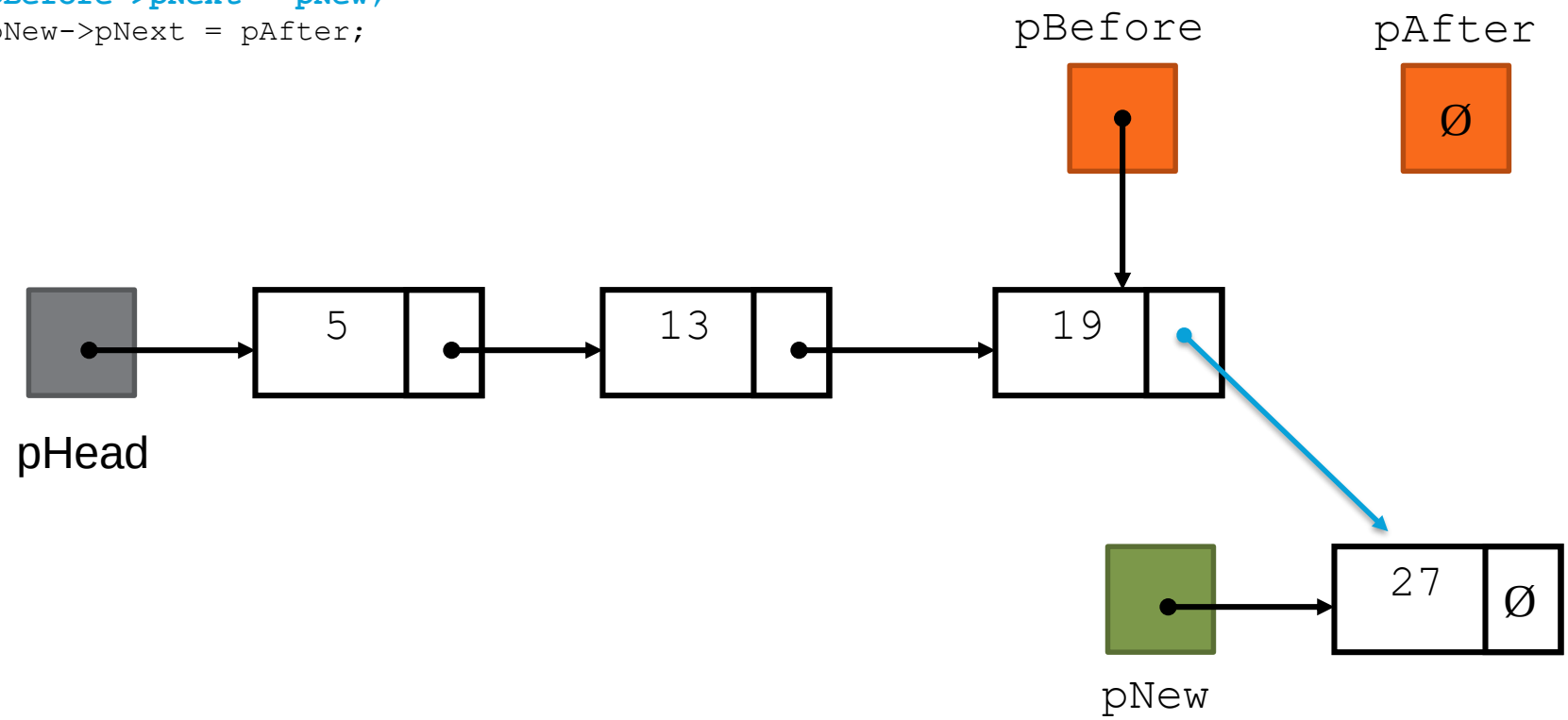
```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



New node created; position located

APPEND AT INSERTION POINT

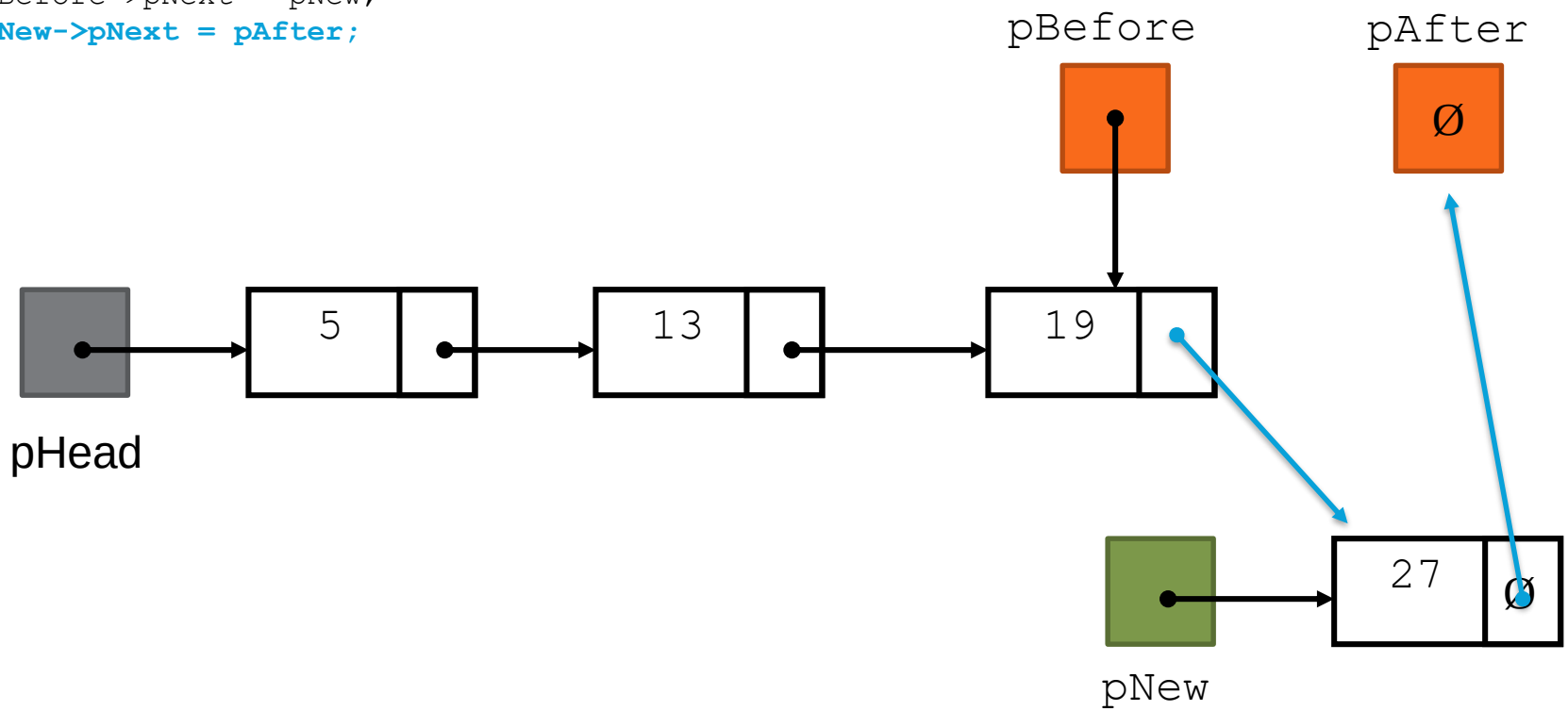
```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



New node created; position located

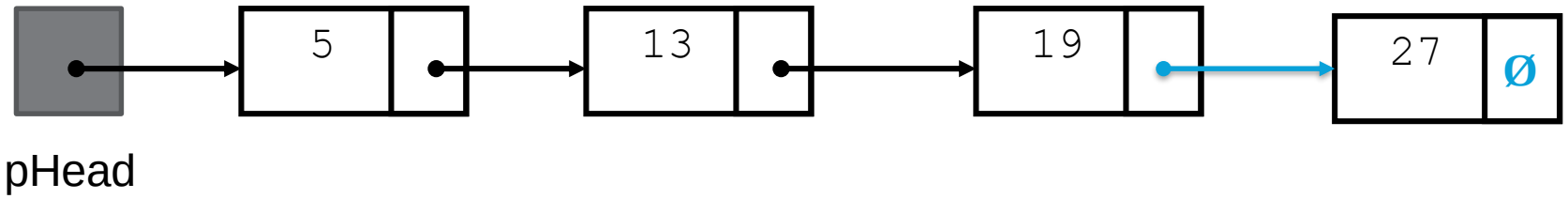
APPEND AT INSERTION POINT

```
else
{
   // insert after the previous node
   pBefore->pNext = pNew;
   pNew->pNext = pAfter;
}
```



New node created; position located

APPEND AT INSERTION POINT



Node appended

LINKED LIST OPERATIONS

Basic operations:

- append (add at the end)
- prepend (add at the beginning)
- insert (add in the middle)
- destroy the list (deallocate memory)
- delete a node (various locations in the list)
- traverse the linked list (find specific item)

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,

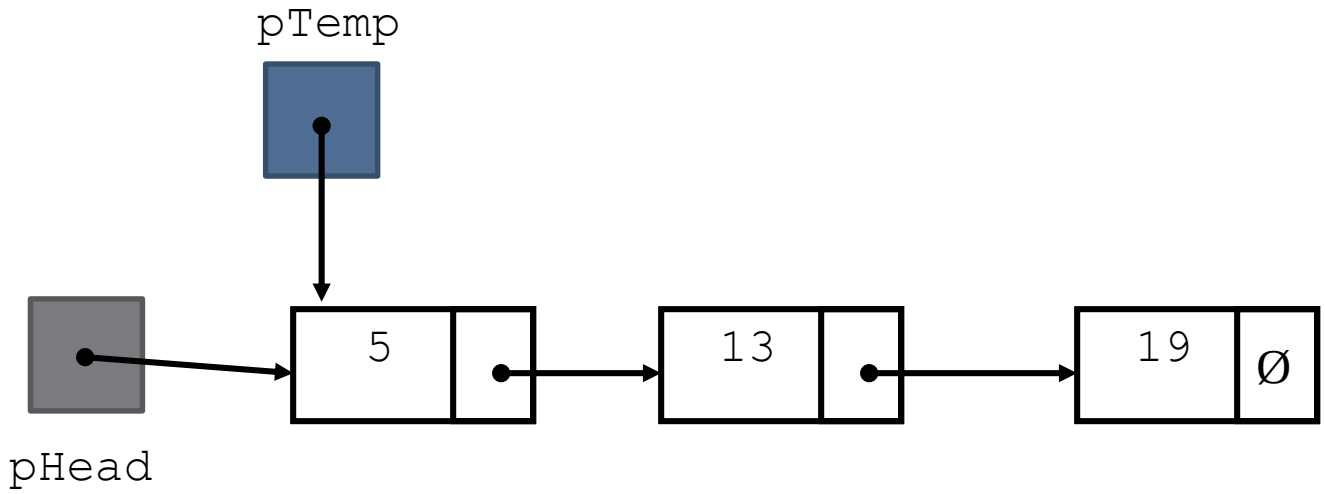
1. **Unlink the node from the list**
2. **If the list uses dynamic memory, then free the node's memory**

Set the list head pointer to `nullptr` when done

DESTROYING A LINKED LIST

```
Node* pTemp;

pTemp = pHead;

while (pTemp != nullptr)
{
        // save pointer to next node in the list
        pHead = pTemp->pNext;

        delete pTemp;

        // start traversing at next node
        pTemp = pHead;
}
```

DELETING THE LIST ONE NODE AT A TIME

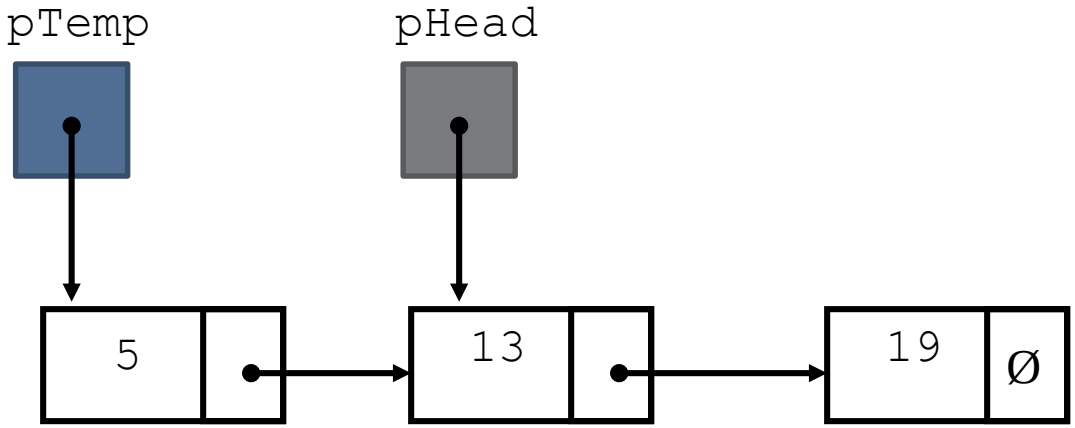


Start traversing the list, while pTemp!=nullptr

DESTROYING A LINKED LIST

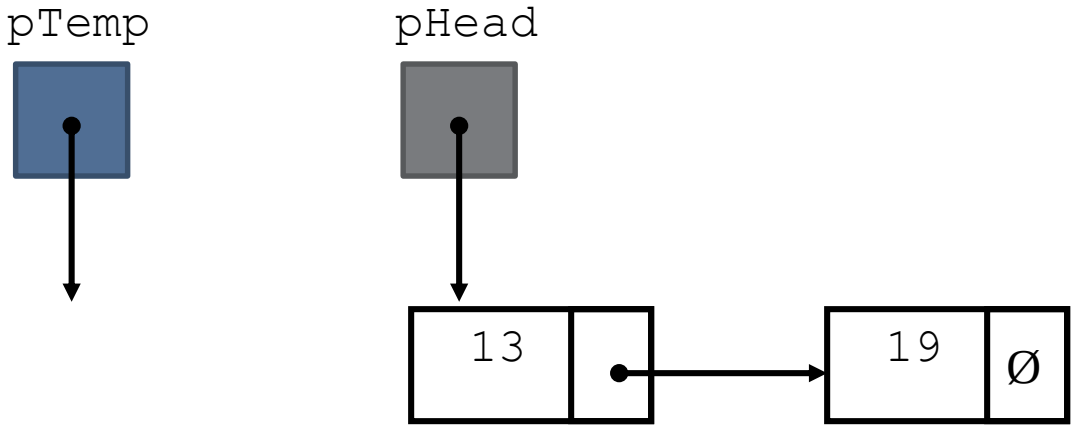
```
Node* pTemp;

pTemp = pHead;

while (pTemp != nullptr)
{
        // save pointer to next node in the list
        pHead = pTemp->pNext;

        delete pTemp;

        // start traversing at next node
        pTemp = pHead;
}
```

DELETING THE LIST ONE NODE AT A TIME



Mark the next node with pHead

DELETING THE LIST ONE NODE AT A TIME

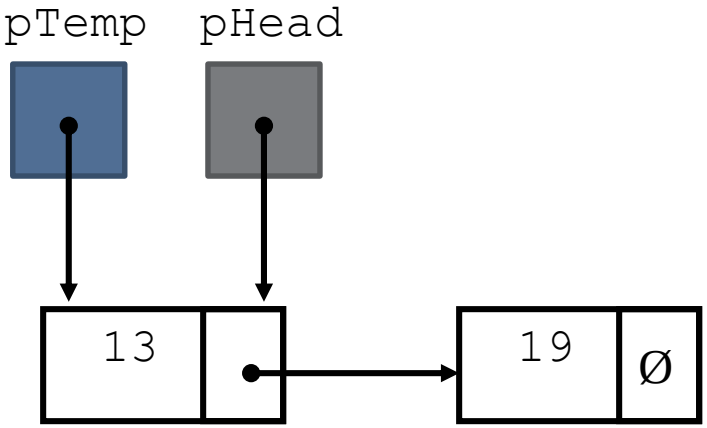


delete pTemp

DESTROYING A LINKED LIST

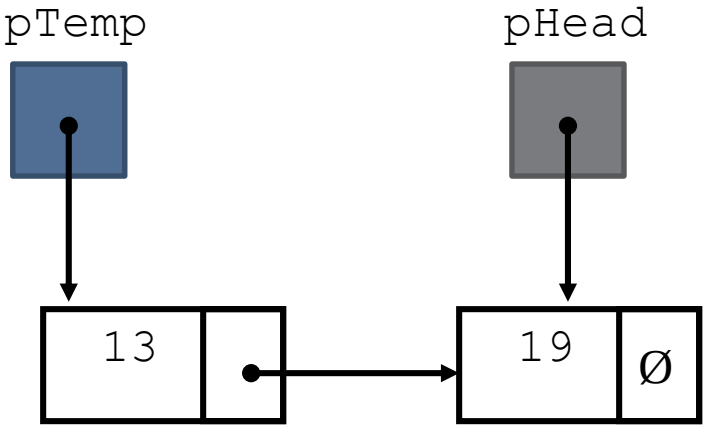
```
Node* pTemp;
Node* pHold;

pTemp = pHead;

while (pTemp != nullptr)
{
        // save pointer to next node in the list
        pHead = pTemp->pNext;

        delete pTemp;

        // start traversing at next node
        pTemp = pHead;
}
```

DELETING THE LIST ONE NODE AT A TIME



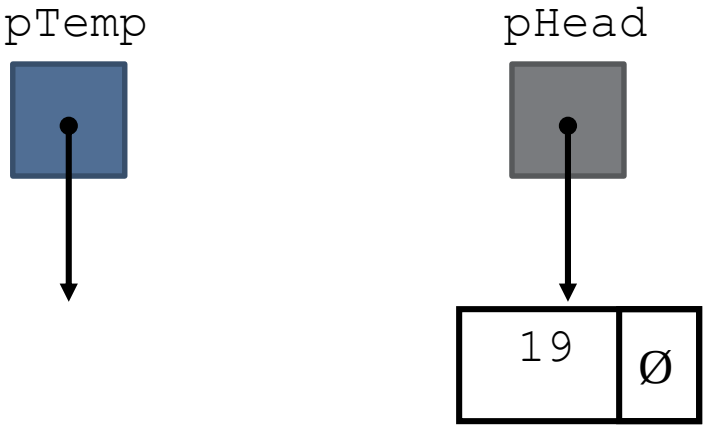
Set pTemp to pHead;

DELETING THE LIST ONE NODE AT A TIME



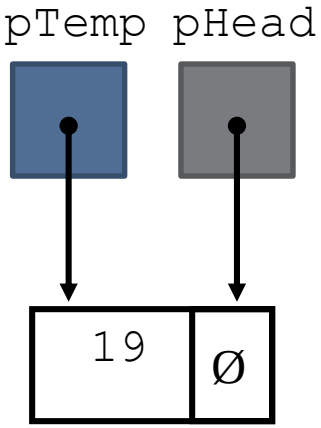
Mark next node;

DELETING THE LIST ONE NODE AT A TIME

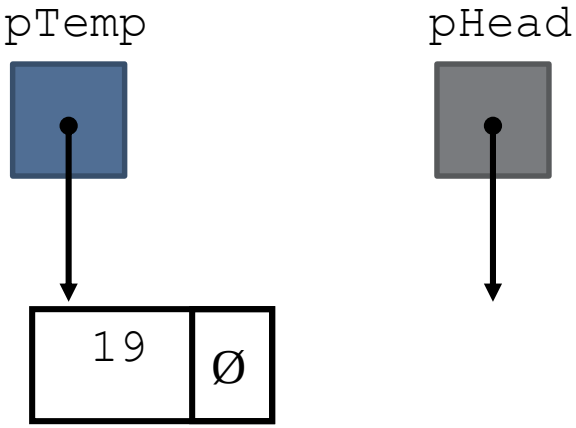


delete pTemp

DELETING THE LIST ONE NODE AT A TIME

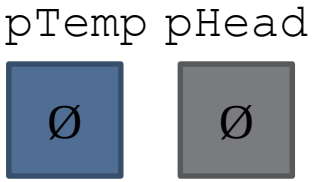


DELETING THE LIST ONE NODE AT A TIME



DELETING THE LIST ONE NODE AT A TIME

DELETING THE LIST ONE NODE AT A TIME



DESTROYING A LINKED LIST

```
Node* pTemp;
Node* pHold;

pTemp = pHead;

while (pTemp != nullptr)
{
        // save pointer to next node in the list
        pHead = pTemp->pNext;

        delete pTemp;

        // start traversing at next node
        pTemp = pHead;
}
```

DESTROYING A LINKED LIST – RECURSIVE FUNCTION

```
void DestroyList(Node* &pHead)
{
        Node* pTemp;

        if (pHead == nullptr)
                return;

        pTemp = pHead;
        pHead = pHead->pNext;
        delete pTemp;
        DestroyList(pHead);
}
```

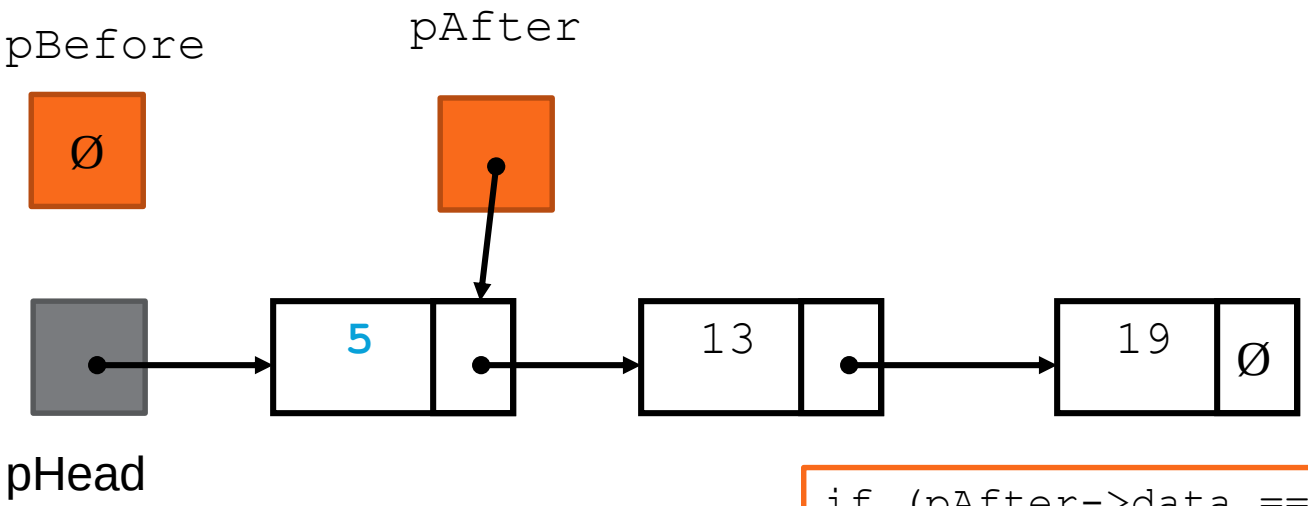
DELETING A NODE

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 – CASE 1

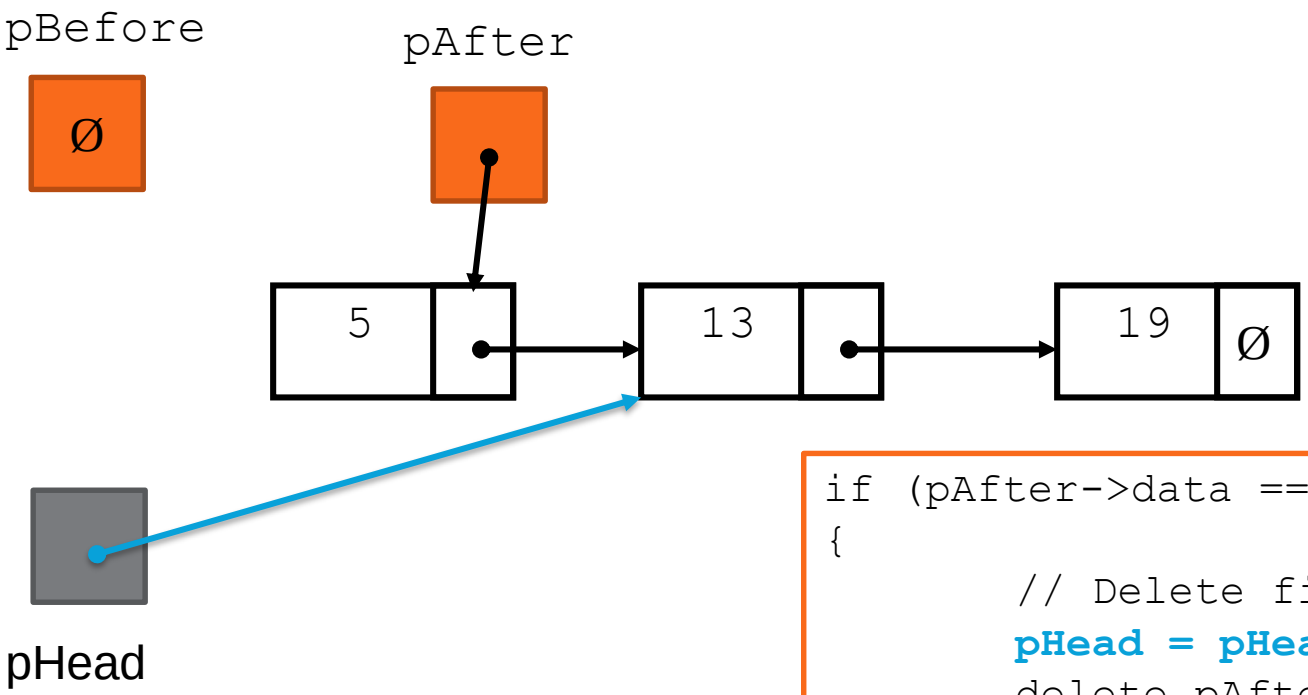
Node to be deleted: the one that contains 5




```
if (pAfter->data == value2find)
{
        // Delete first node
        pHead = pHead->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 1

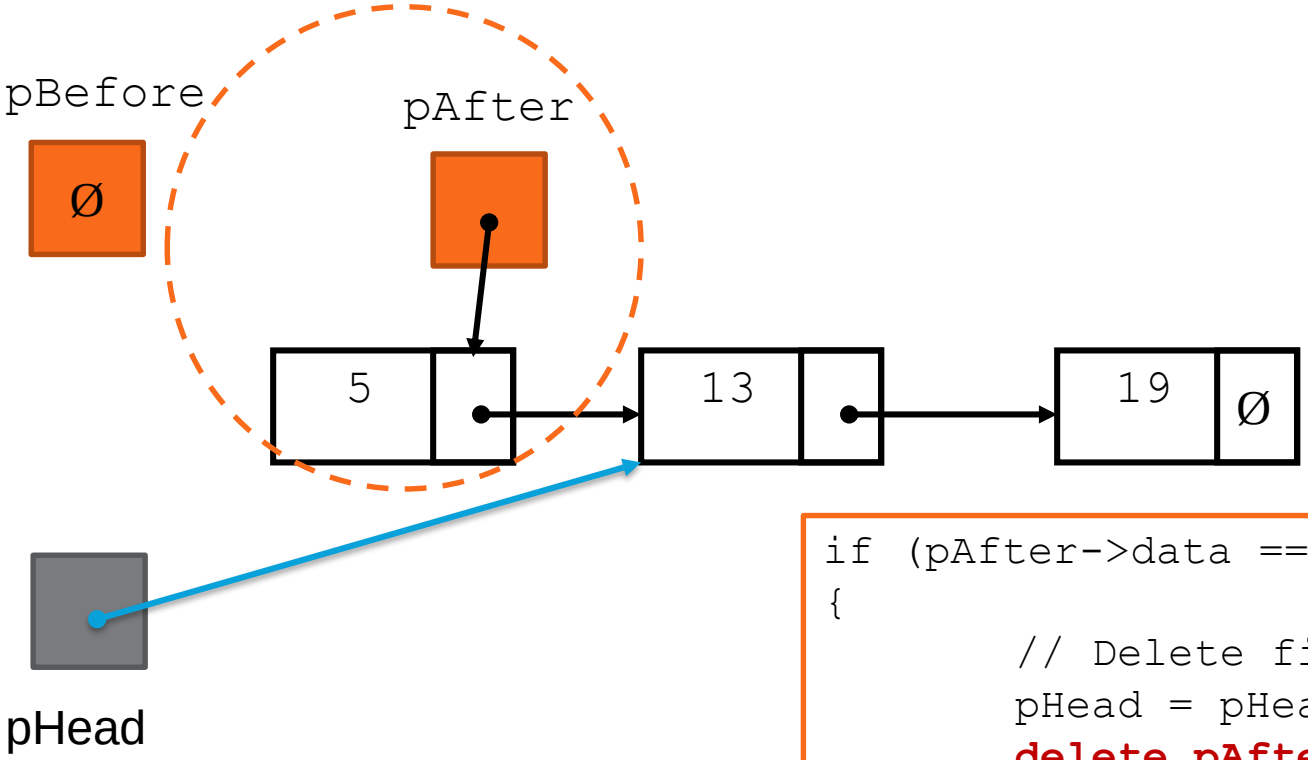
Node to be deleted: the one that contains 5




```
if (pAfter->data == value2find)
{
        // Delete first node
        pHead = pHead->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 1

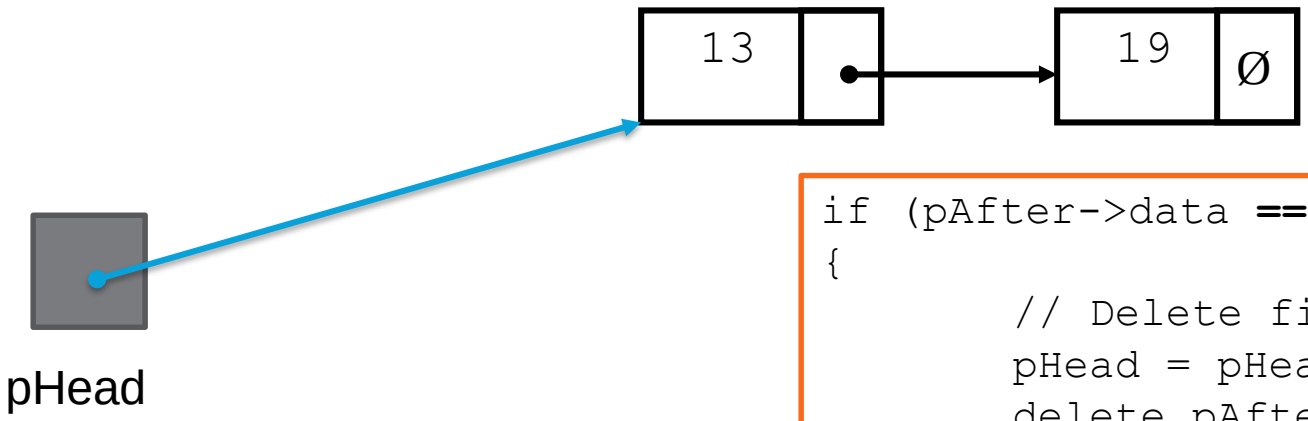
Node to be deleted: the one that contains 5




```
if (pAfter->data == value2find)
{
        // Delete first node
        pHead = pHead->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 1

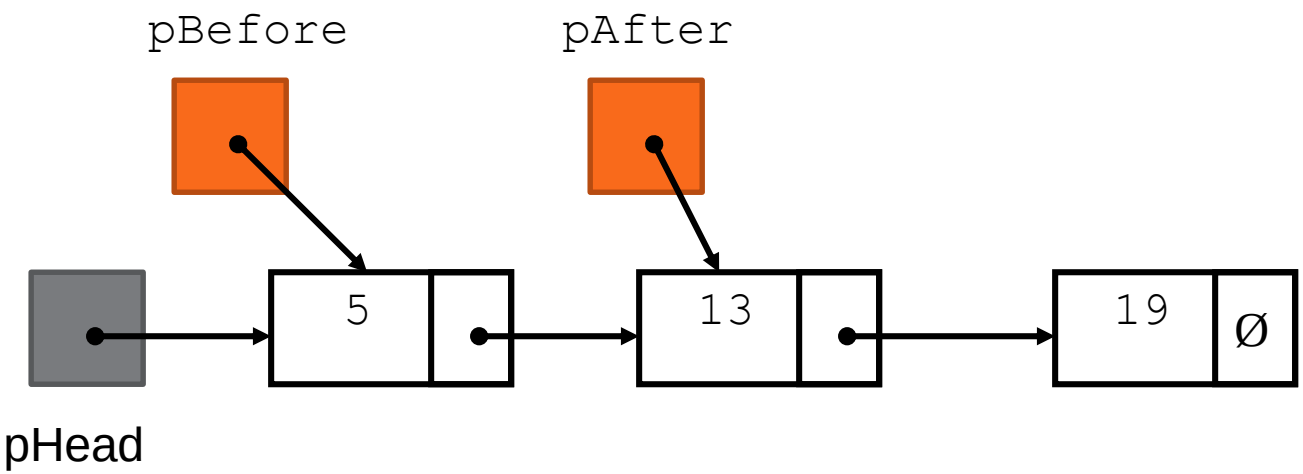
Node to be deleted: the one that contains 5




```
if (pAfter->data == value2find)
{
        // Delete first node
        pHead = pHead->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 2

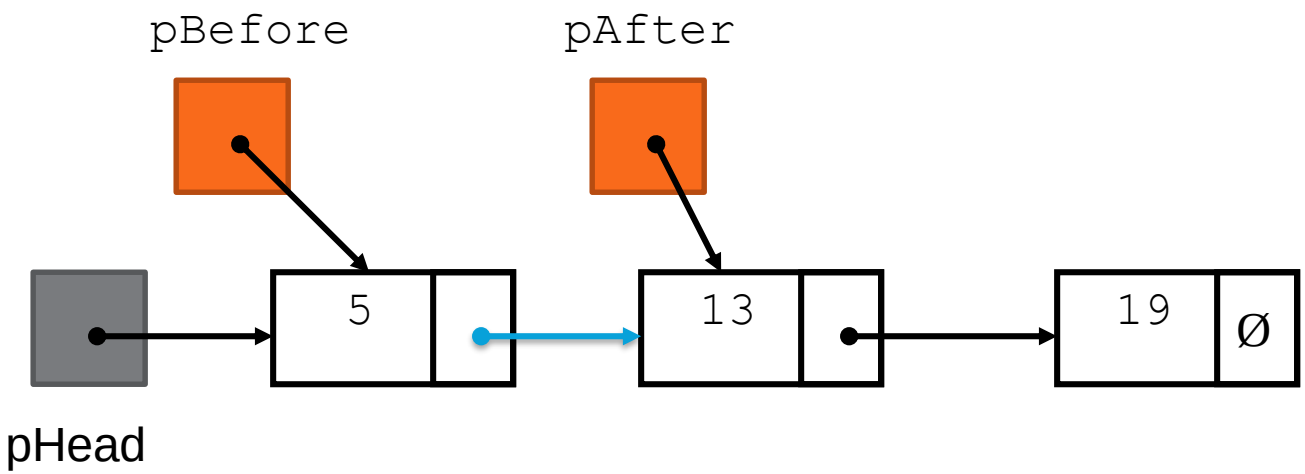
Node to be deleted: the one that contains 13



Locating the node containing `13`

DELETING A NODE – CASE 2

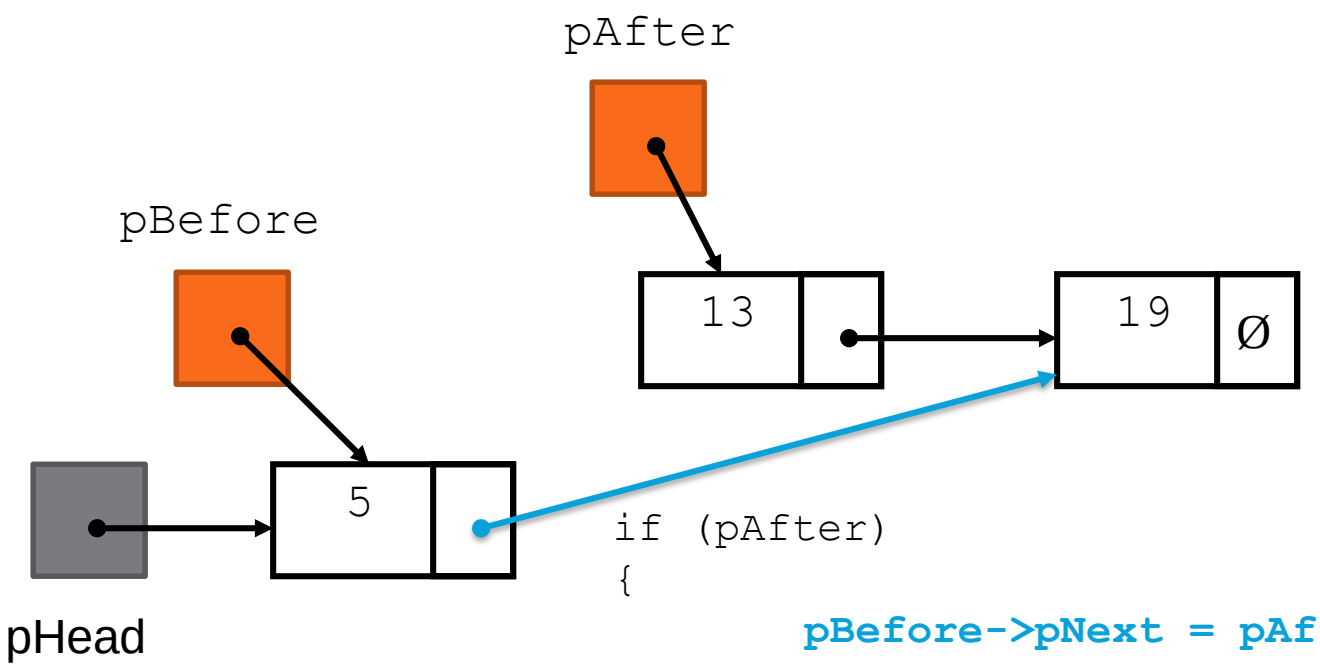
Node to be deleted: the one that contains 13




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 2

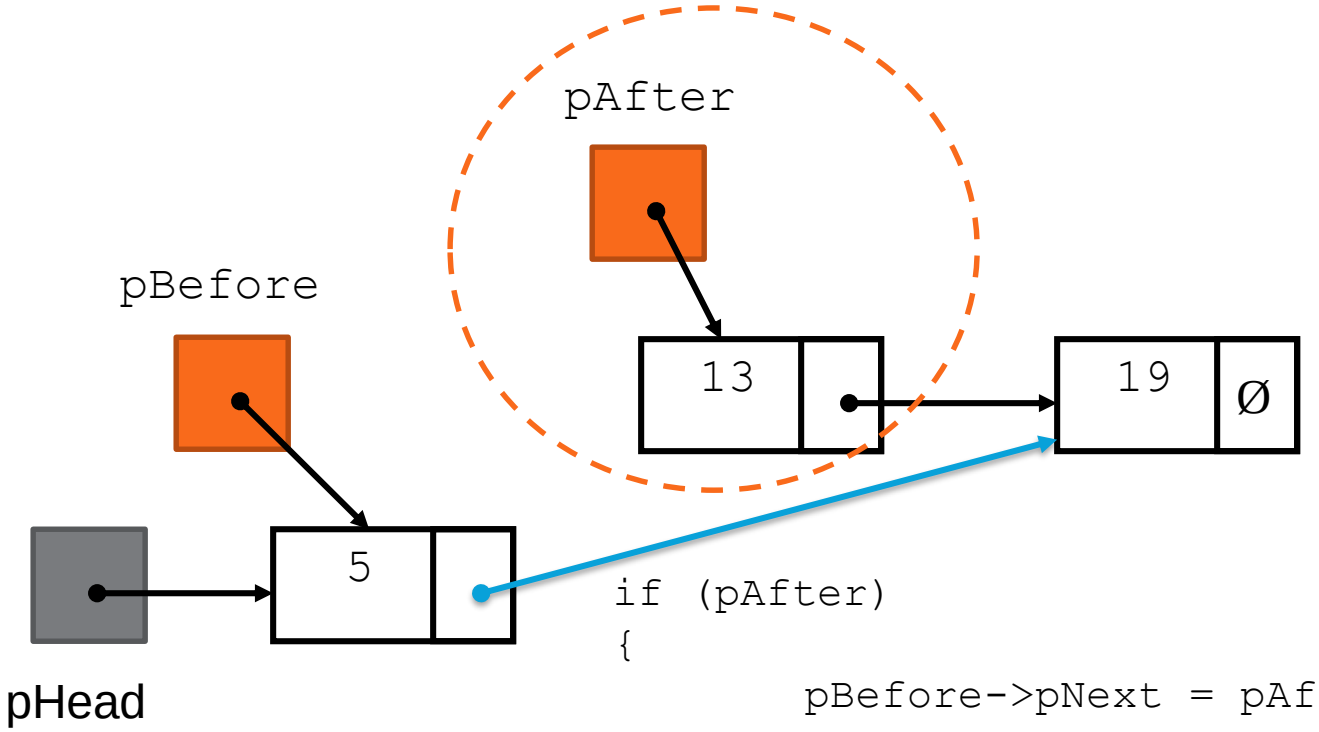
Node to be deleted: the one that contains 13




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 2

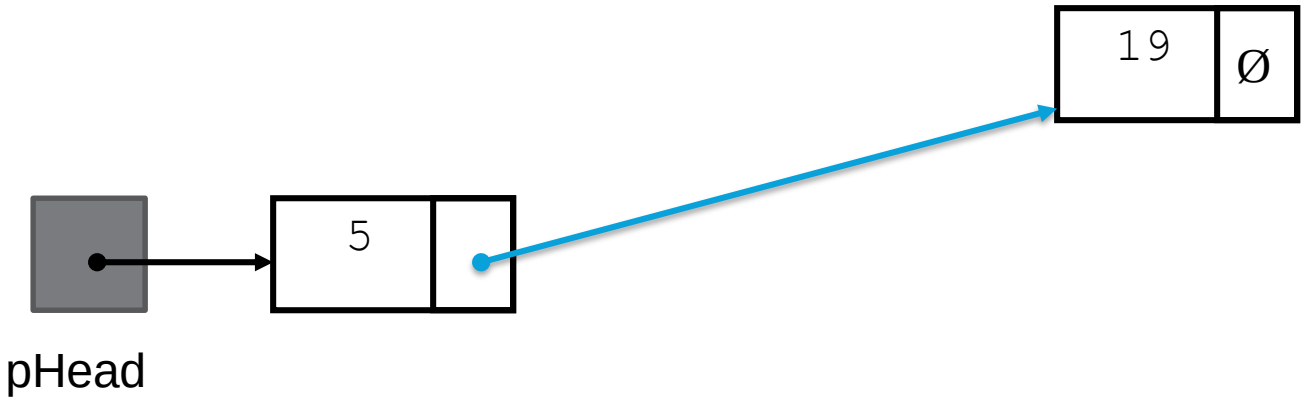
Node to be deleted: the one that contains 13




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

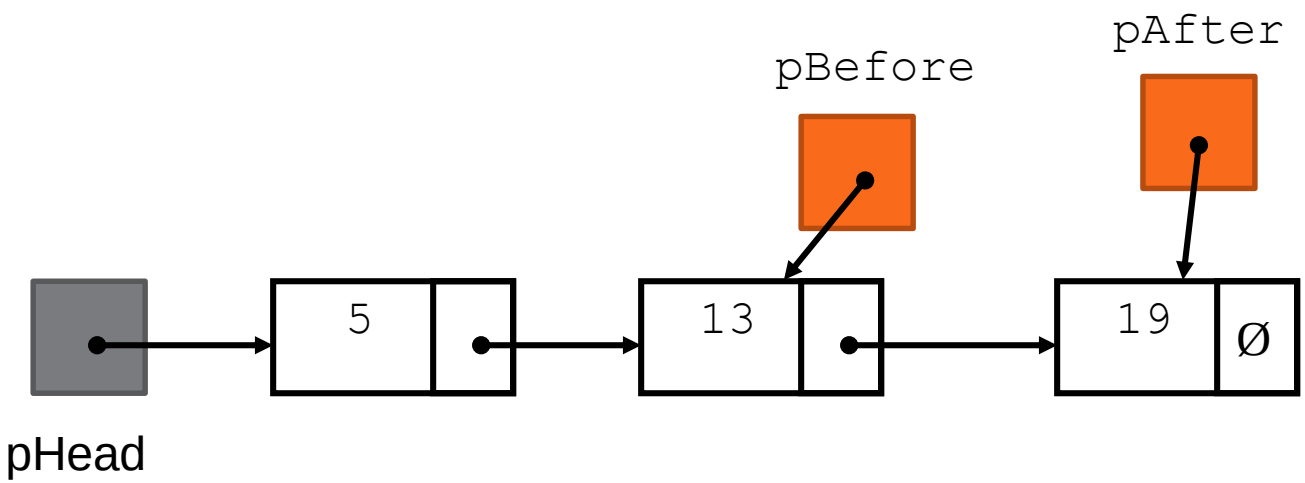
DELETING A NODE – CASE 2

Node to be deleted: the one that contains 13



DELETING A NODE – CASE 3

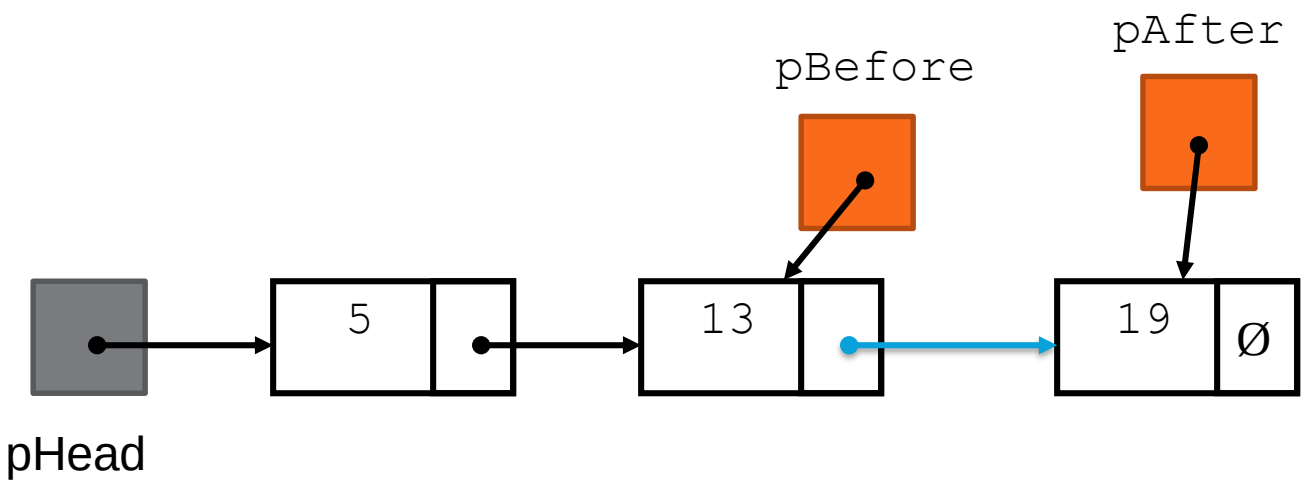
Node to be deleted: the one that contains 19




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 3

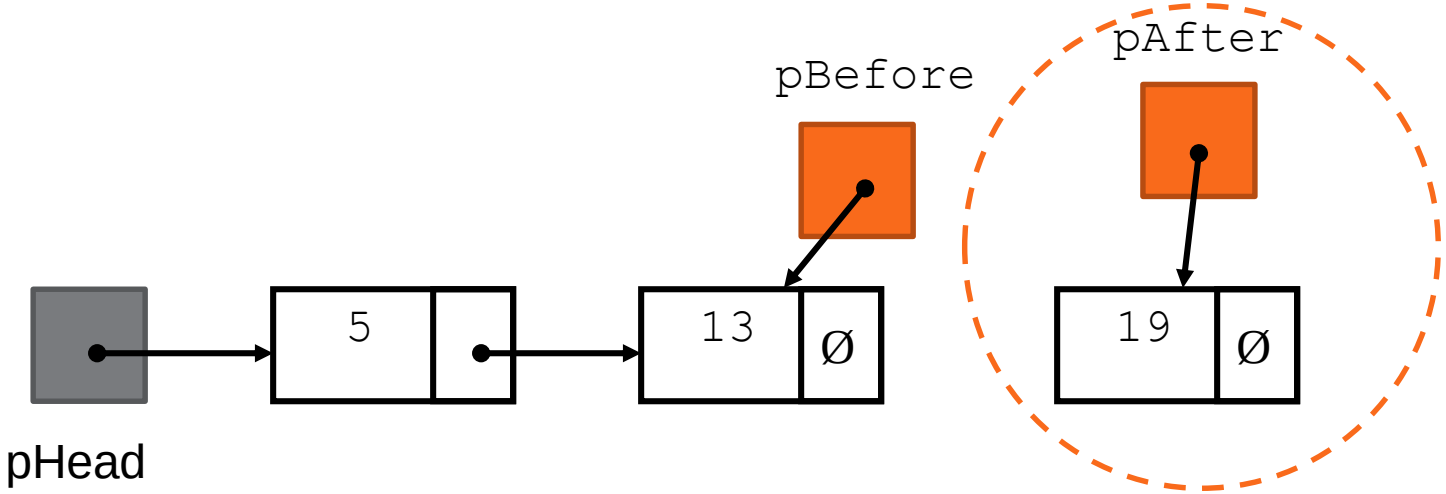
Node to be deleted: the one that contains 19




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

DELETING A NODE – CASE 3

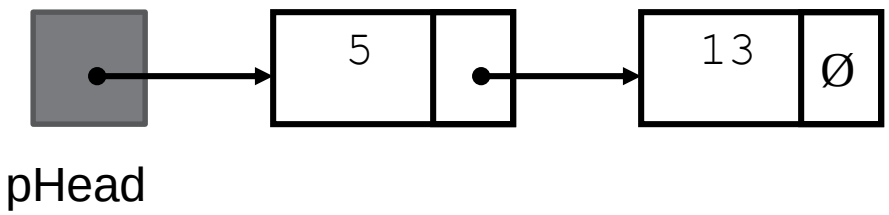
Node to be deleted: the one that contains 19




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

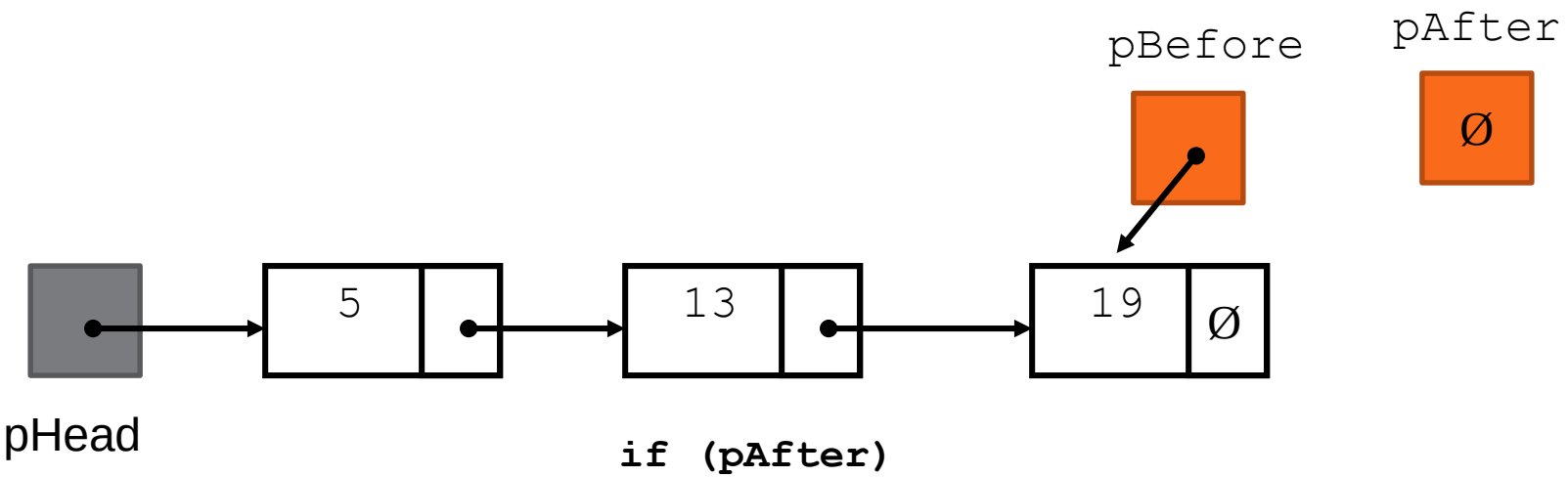
DELETING A NODE – CASE 3

Node to be deleted: the one that contains 19



DELETING A NODE – CASE 4

Node to be deleted: the one that contains **29**




```
if (pAfter)
{
        pBefore->pNext = pAfter->pNext;
        delete pAfter;
}
```

```
Node* pAfter;
Node* pBefore = nullptr;
pAfter = pHead;
if (pAfter->data == value2find)
{
        // Delete first node
        pHead = pHead->pNext;
        delete pAfter;
}
else
{
        // skip all nodes whose data is not value2find
        while (pAfter!=nullptr && pAfter->data != value2find)
        {
                pBefore = pAfter;
                pAfter = pAfter->pNext;
        }
        // continues on next slide…
```

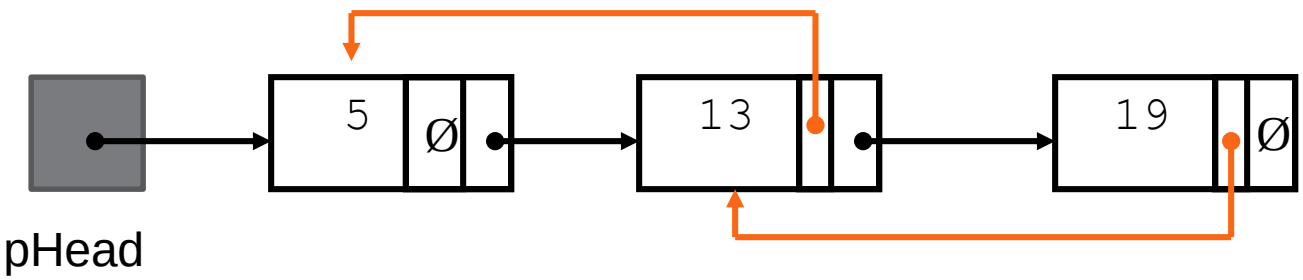
```
    // If pAfter is not null,
    // link the previous node to the node
    // following pAfter, then delete pAfter.
    //
    if (pAfter)
    {
            pBefore->pNext = pAfter->pNext;
            delete pAfter;
    }
}
```

18.4 VARIATIONS OF THE LINKED LIST

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

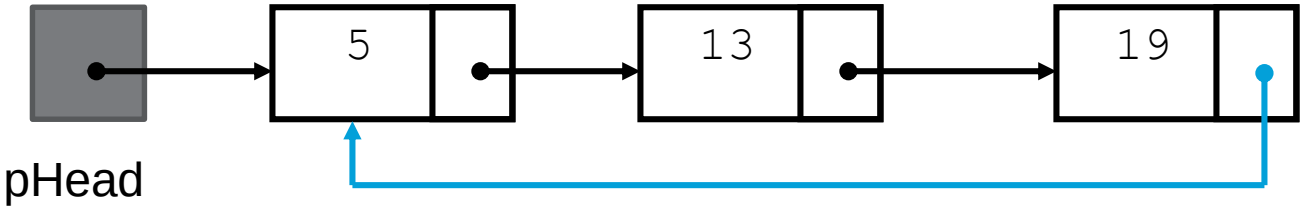



```
struct Node
{
        int data;
        Node* pNext;
        Node* pPrev;
};
```

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 the null pointer



VARIATIONS OF THE LINKED LIST

Other linked list organizations:

- Keeping pHead and pTail pointers always pointing to the beginning and the end of the last.