Algorithms

Linear data structures – Linked lists

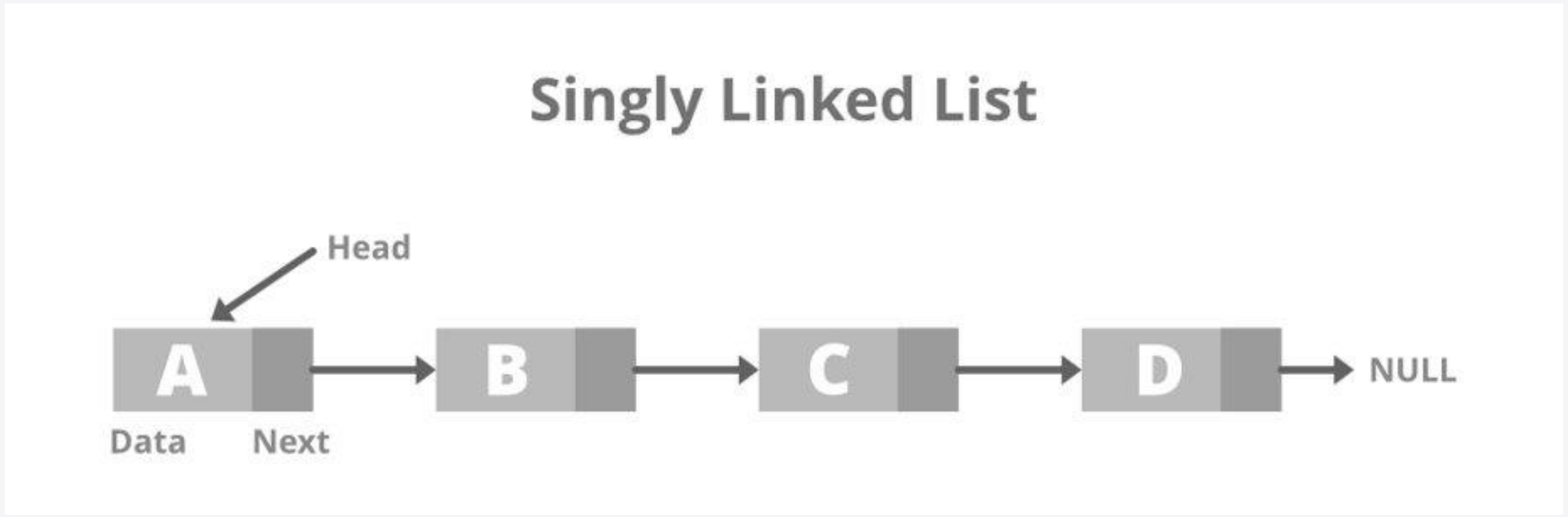
(Implementation & analysis)

Singly Linked List

Introduction

In **computer science**, a linked list is a linear collection of data elements whose **order is not given by their physical placement in memory**.

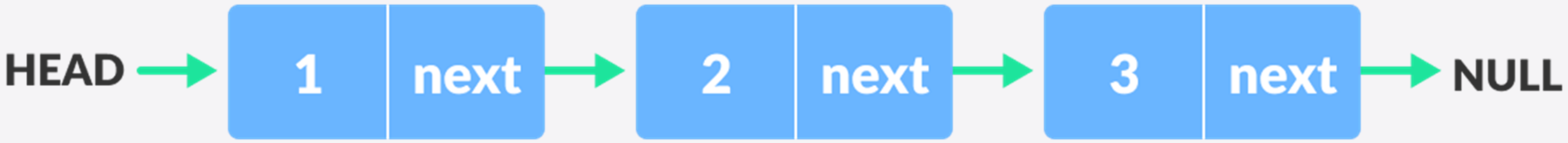
Instead, each element "**points**" to the next. It is a data structure consisting of a collection of **nodes** which together represent a sequence.

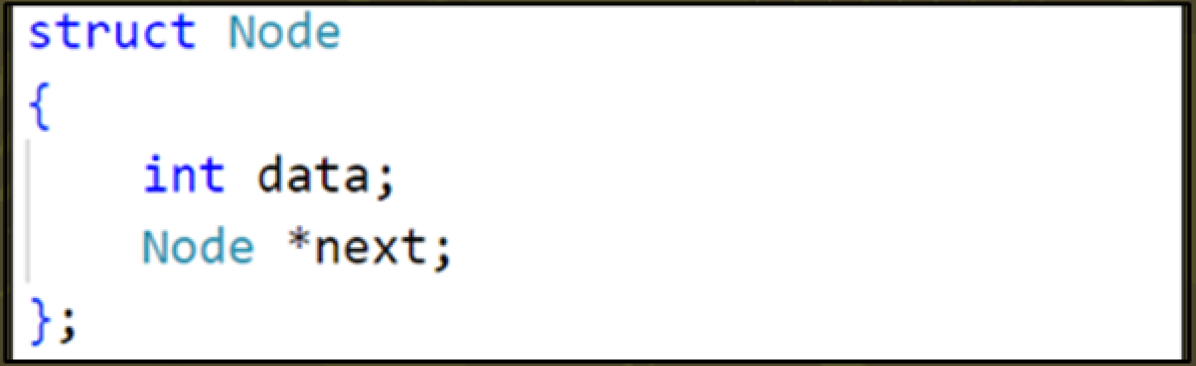


What is a singly linked list?

It is a specialized case of a generic linked list.

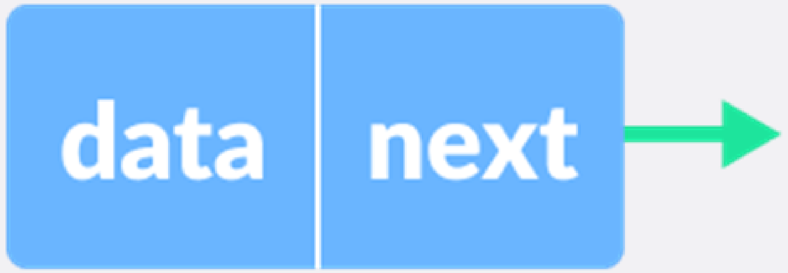
In a **singly linked list**, each node links to only the next node in the sequence, i.e., **if we start traversing from the first node of the list, we can only move in one direction**.



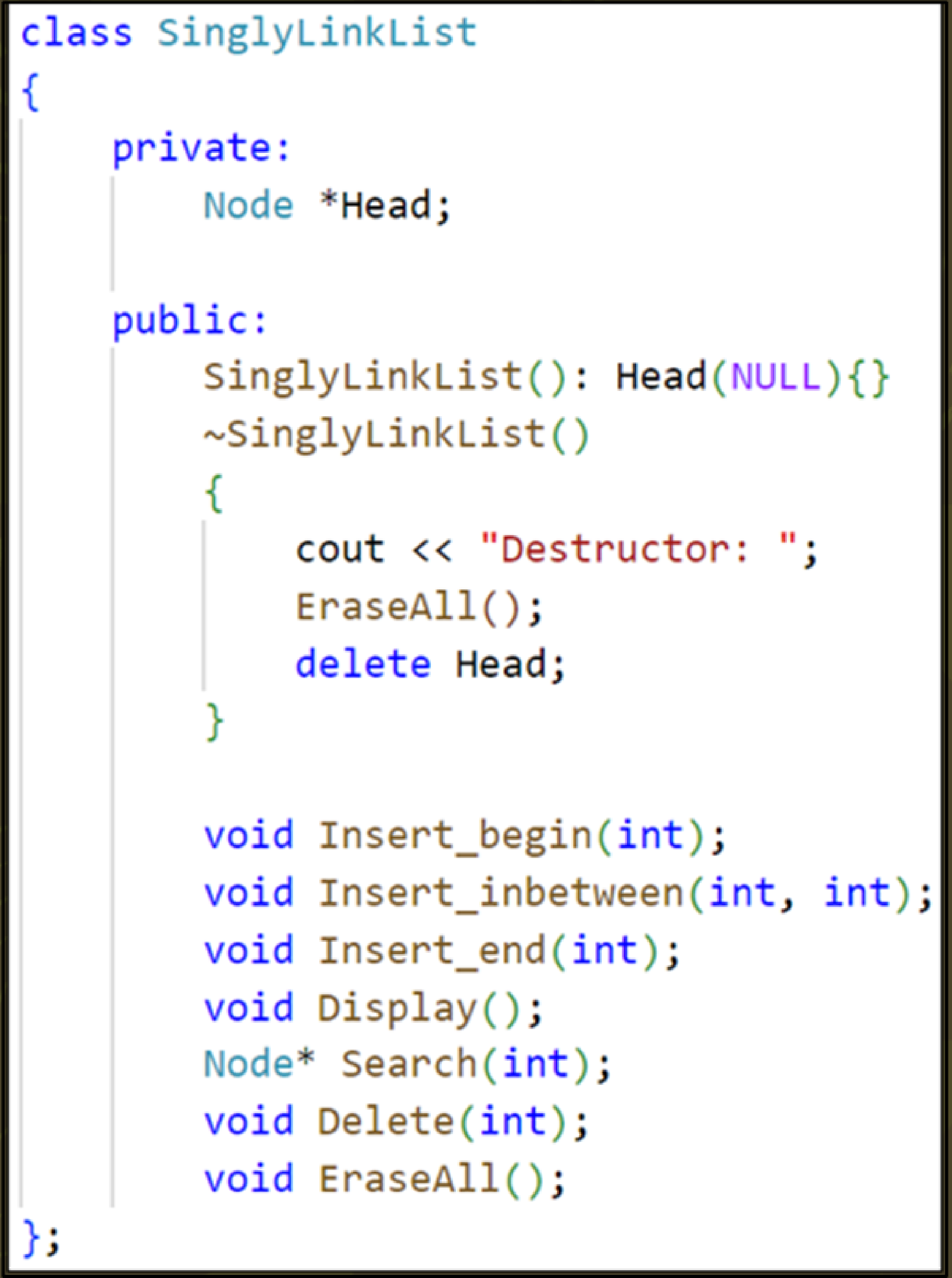


```
struct Node
{
    int data;
    Node *next;
};
```

- The **Node structure** represent each element in the singly linked list.
- Here, the **single node** is represented as

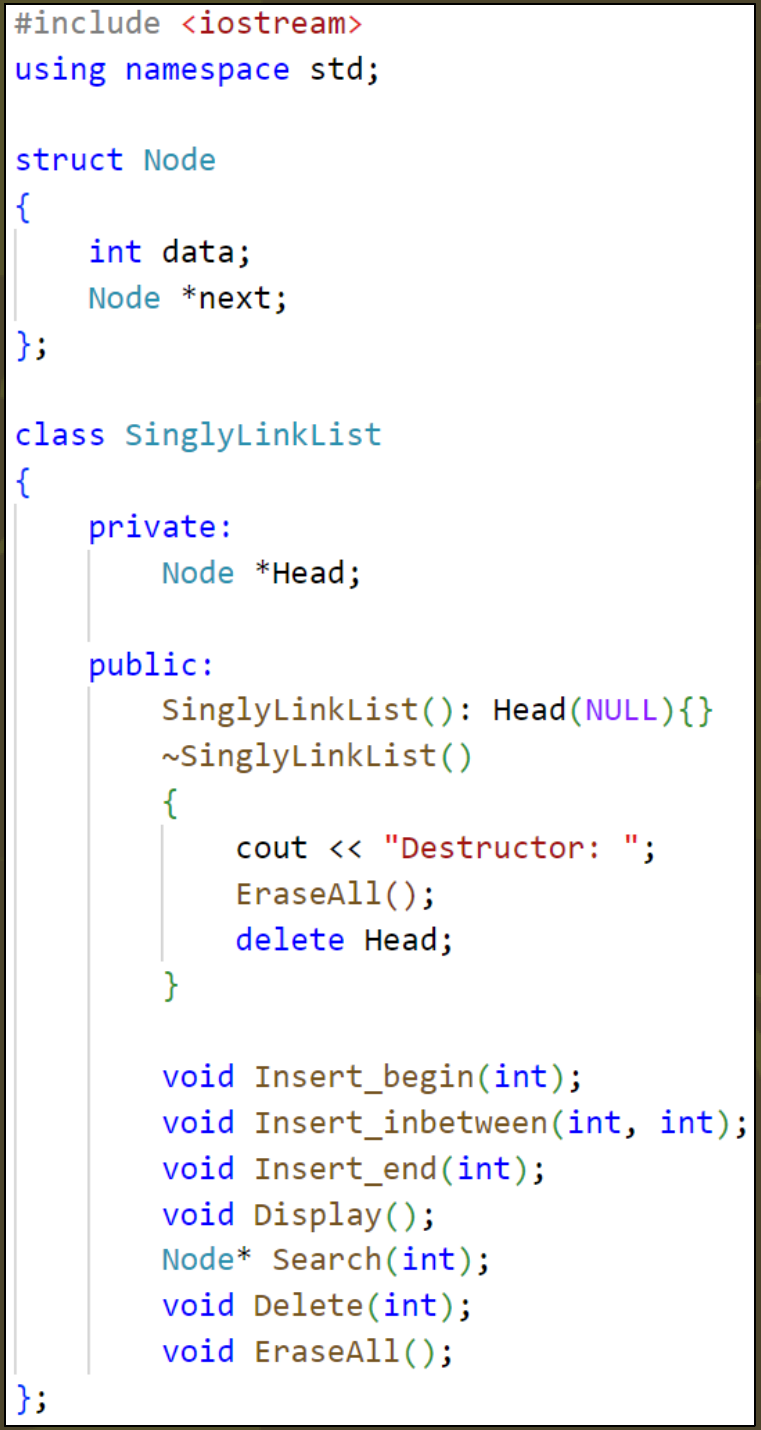


- Each **struct node** contains a **data item** and a **pointer to the next node**.



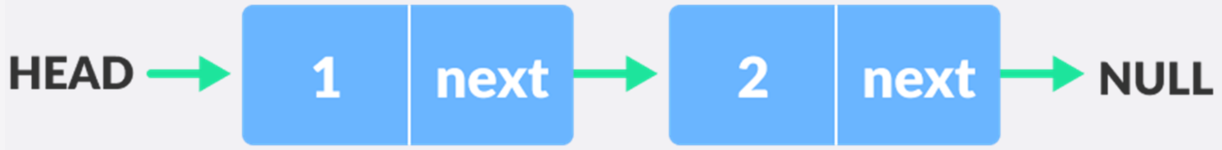
```
class SinglyLinkList
{
    private:
        Node *Head;

    public:
        SinglyLinkList(): Head(NULL){}
        ~SinglyLinkList()
        {
            cout << "Destructor: ";
            EraseAll();
            delete Head;
        }

        void Insert_begin(int);
        void Insert_inbetween(int, int);
        void Insert_end(int);
        void Display();
        Node* Search(int);
        void Delete(int);
        void EraseAll();
};
```

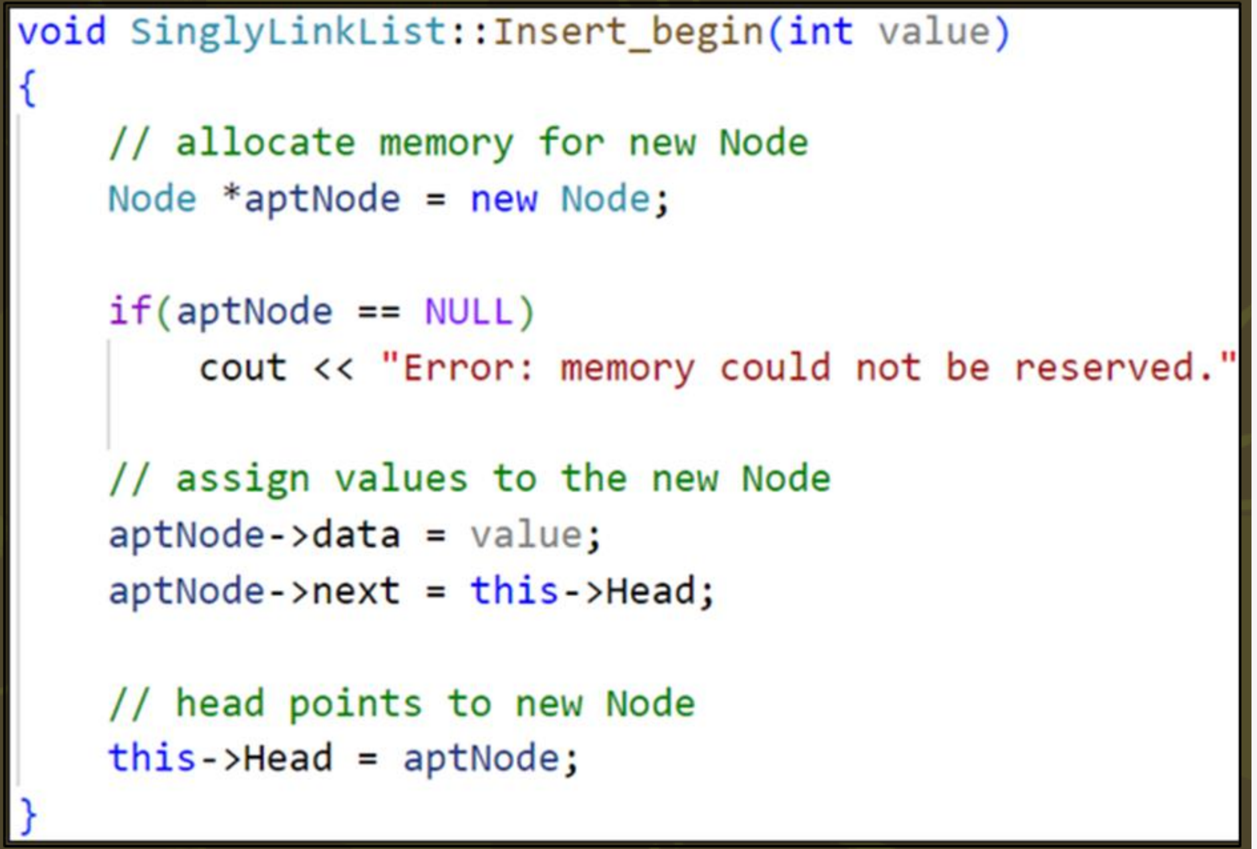
Now we will recreate a **simple linked list** through a **SinglyLinkList class** where we implement all the functionalities of this data structure.


```cpp
#include <iostream>
using namespace std;

struct Node
{
    int data;
    Node *next;
};

class SinglyLinkList
{
    private:
        Node *Head;

    public:
        SinglyLinkList(): Head(NULL){}
        ~SinglyLinkList()
        {
            cout << "Destructor: ";
            EraseAll();
            delete Head;
        }

        void Insert_begin(int);
        void Insert_inbetween(int, int);
        void Insert_end(int);
        void Display();
        Node* Search(int);
        void Delete(int);
        void EraseAll();
};
```

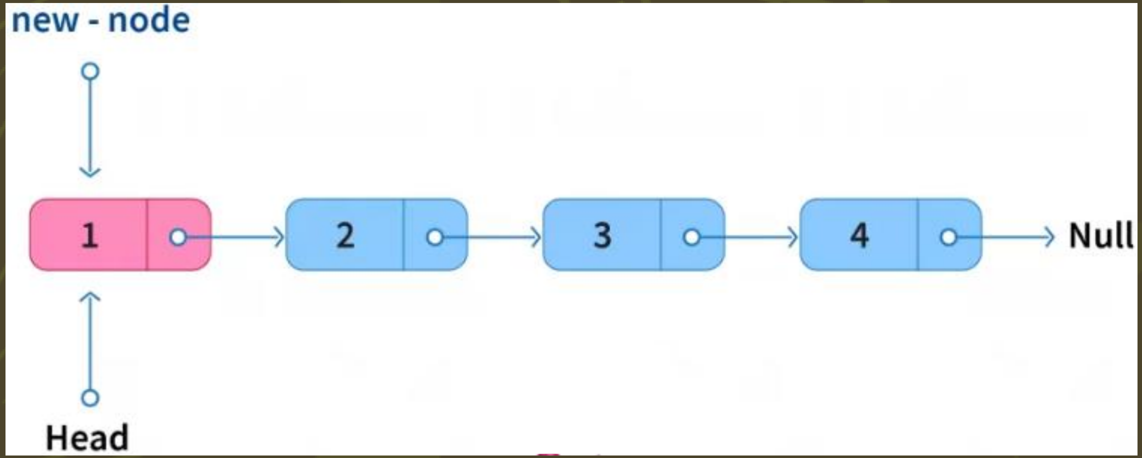
- You have to start somewhere, so we give the address of the **first node** a special name called **HEAD**.

- Also, the **last node** in the linked list can be identified because its next portion points to **NULL**.





```
void SinglyLinkList::Insert_begin(int value)
{
    // allocate memory for new Node
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved."

    // assign values to the new Node
    aptNode->data = value;
    aptNode->next = this->Head;

    // head points to new Node
    this->Head = aptNode;
}
```



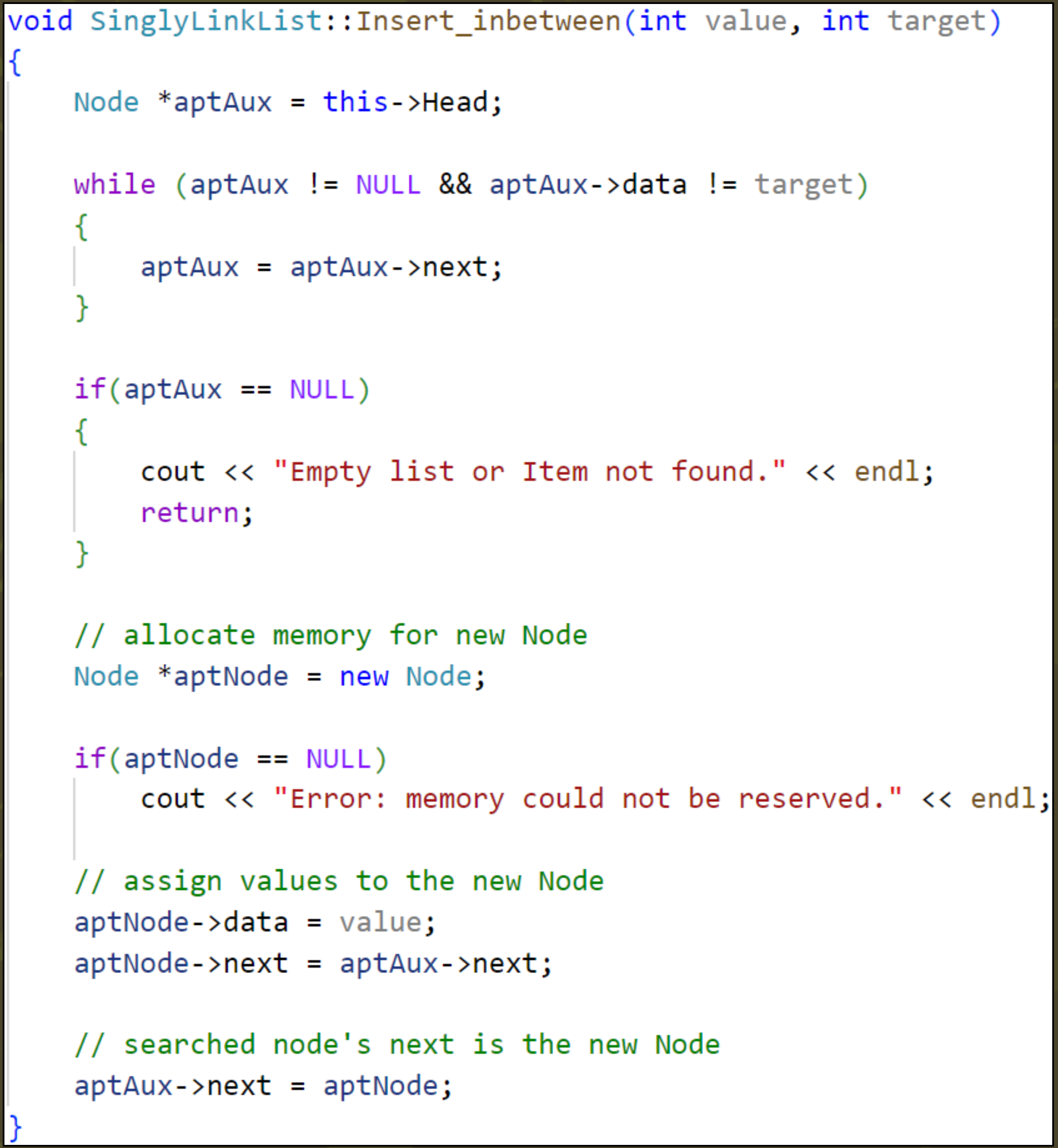
Insert (1/3)

Insertion at the start

Insertion of a new node at the start of a singly linked list is carried out in the following manner.

- Make the **new node** to **HEAD**.
- Make the **HEAD** point to the **new node**.

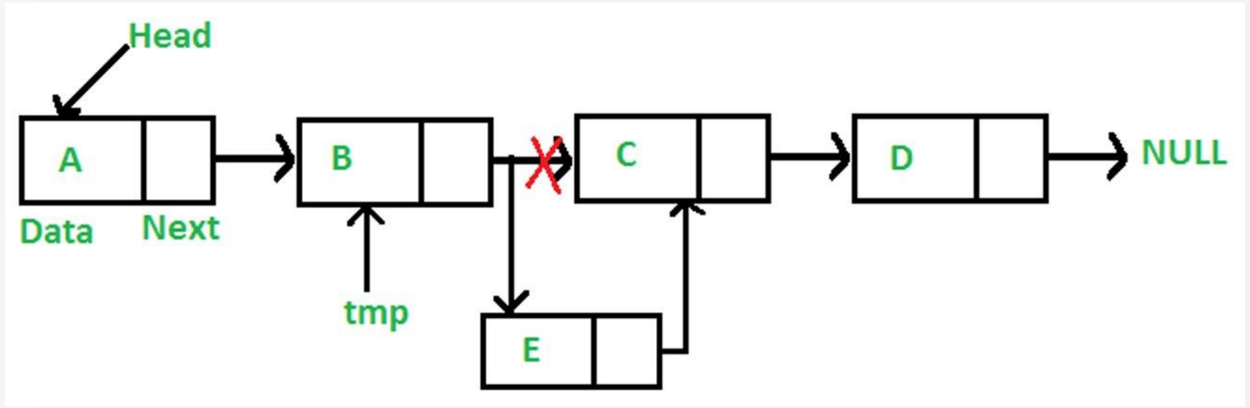
What is the time complexity?


```
void SinglyLinkList::Insert_inbetween(int value, int target)
{
    Node *aptAux = this->Head;

    while (aptAux != NULL && aptAux->data != target)
    {
        aptAux = aptAux->next;
    }

    if(aptAux == NULL)
    {
        cout << "Empty list or Item not found." << endl;
        return;
    }

    // allocate memory for new Node
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved." << endl;

    // assign values to the new Node
    aptNode->data = value;
    aptNode->next = aptAux->next;

    // searched node's next is the new Node
    aptAux->next = aptNode;
}
```

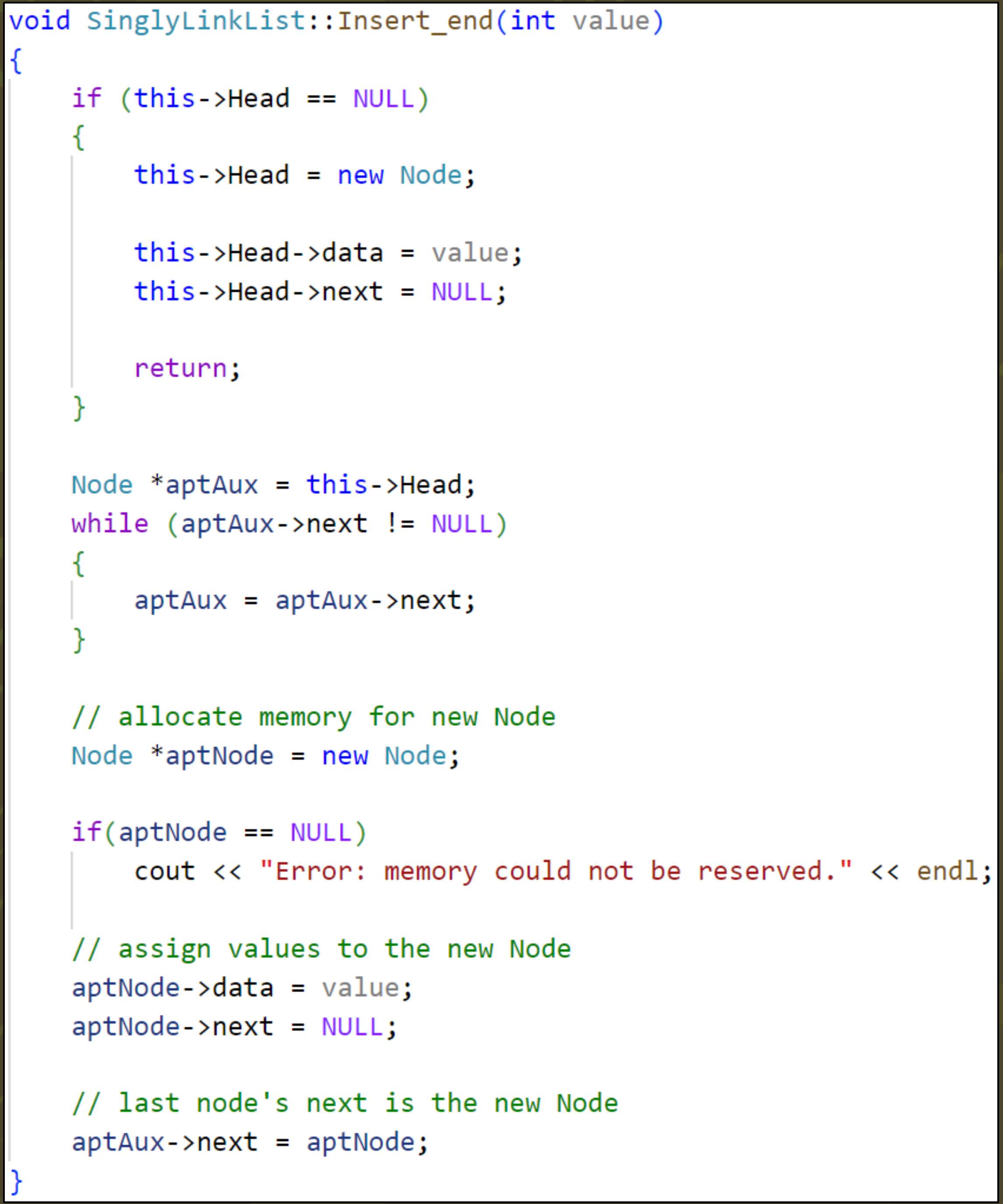
Insert (2/3)

Insertion after some Node

- Reach the **desired node** after which the **new node** is to be inserted.
- Make the **new node** to the next element of the **current node**.



What is the time complexity?

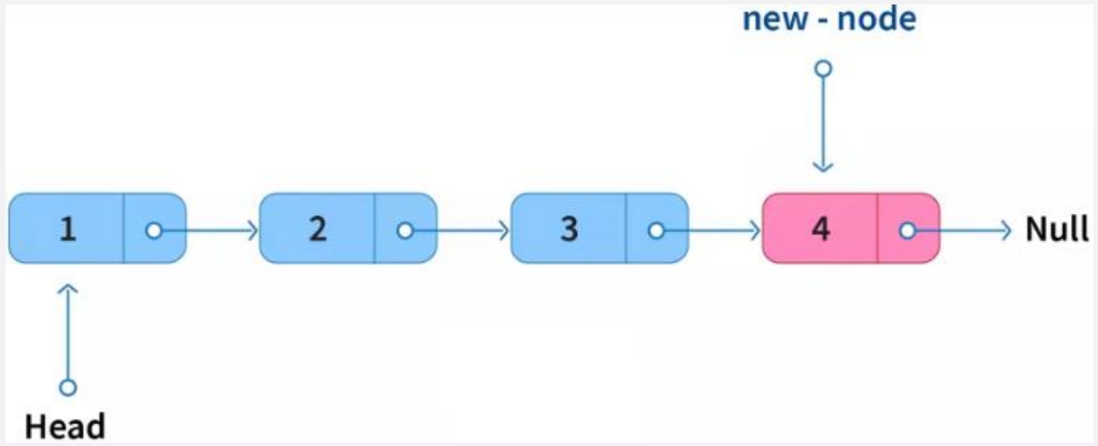


```cpp
void SinglyLinkList::Insert_end(int value)
{
    if (this->Head == NULL)
    {
        this->Head = new Node;

        this->Head->data = value;
        this->Head->next = NULL;

        return;
    }

    Node *aptAux = this->Head;
    while (aptAux->next != NULL)
    {
        aptAux = aptAux->next;
    }

    // allocate memory for new Node
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved." << endl;

    // assign values to the new Node
    aptNode->data = value;
    aptNode->next = NULL;

    // last node's next is the new Node
    aptAux->next = aptNode;
}
```

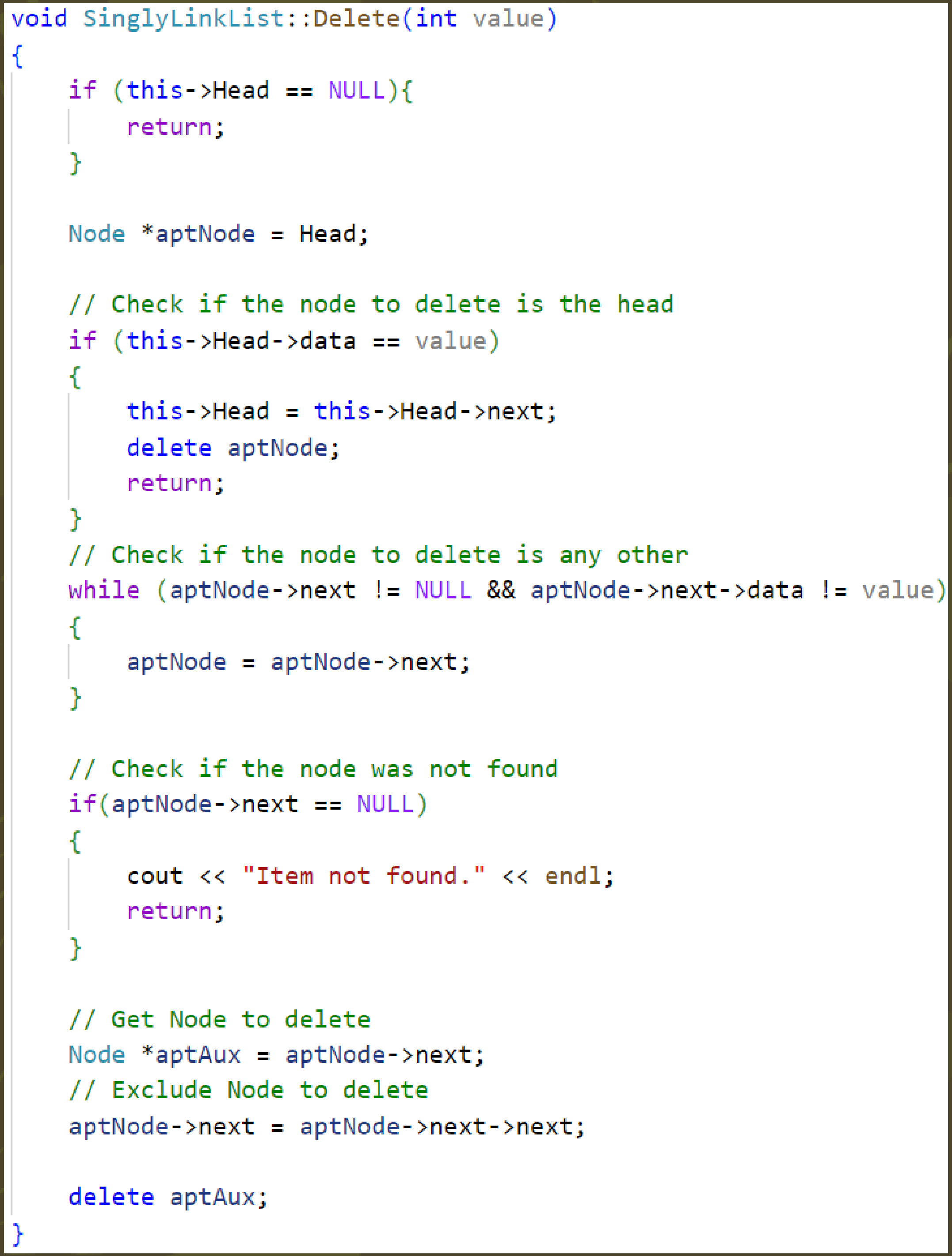
Insert (3/3)

Insertion at the end

- Traverse the list from start and reach the last node.
- Make the last node point to the new node.
- Make the new node point to null, marking the end of the list.



What is the time complexity?



```
void SinglyLinkList::Delete(int value)
{
    if (this->Head == NULL){
        return;
    }

    Node *aptNode = Head;

    // Check if the node to delete is the head
    if (this->Head->data == value)
    {
        this->Head = this->Head->next;
        delete aptNode;
        return;
    }
    // Check if the node to delete is any other
    while (aptNode->next != NULL && aptNode->next->data != value)
    {
        aptNode = aptNode->next;
    }

    // Check if the node was not found
    if(aptNode->next == NULL)
    {
        cout << "Item not found." << endl;
        return;
    }

    // Get Node to delete
    Node *aptAux = aptNode->next;
    // Exclude Node to delete
    aptNode->next = aptNode->next->next;

    delete aptAux;
}
```

Deletion

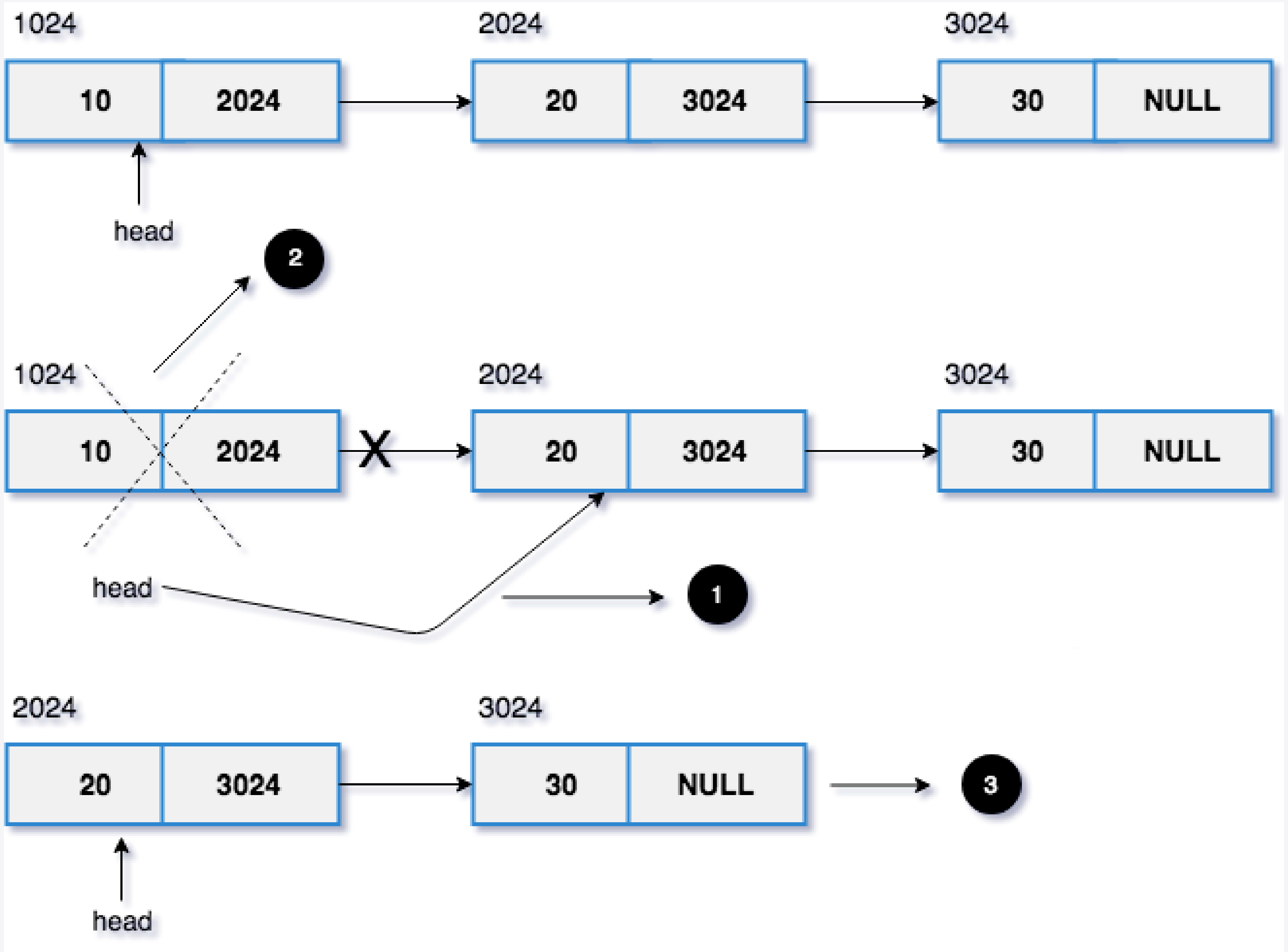
The deletion of a specific node can be formed in the following way,

- Reach the desired node to be deleted.
- Make the next node point the next node of previous node.
- Delete desired node.

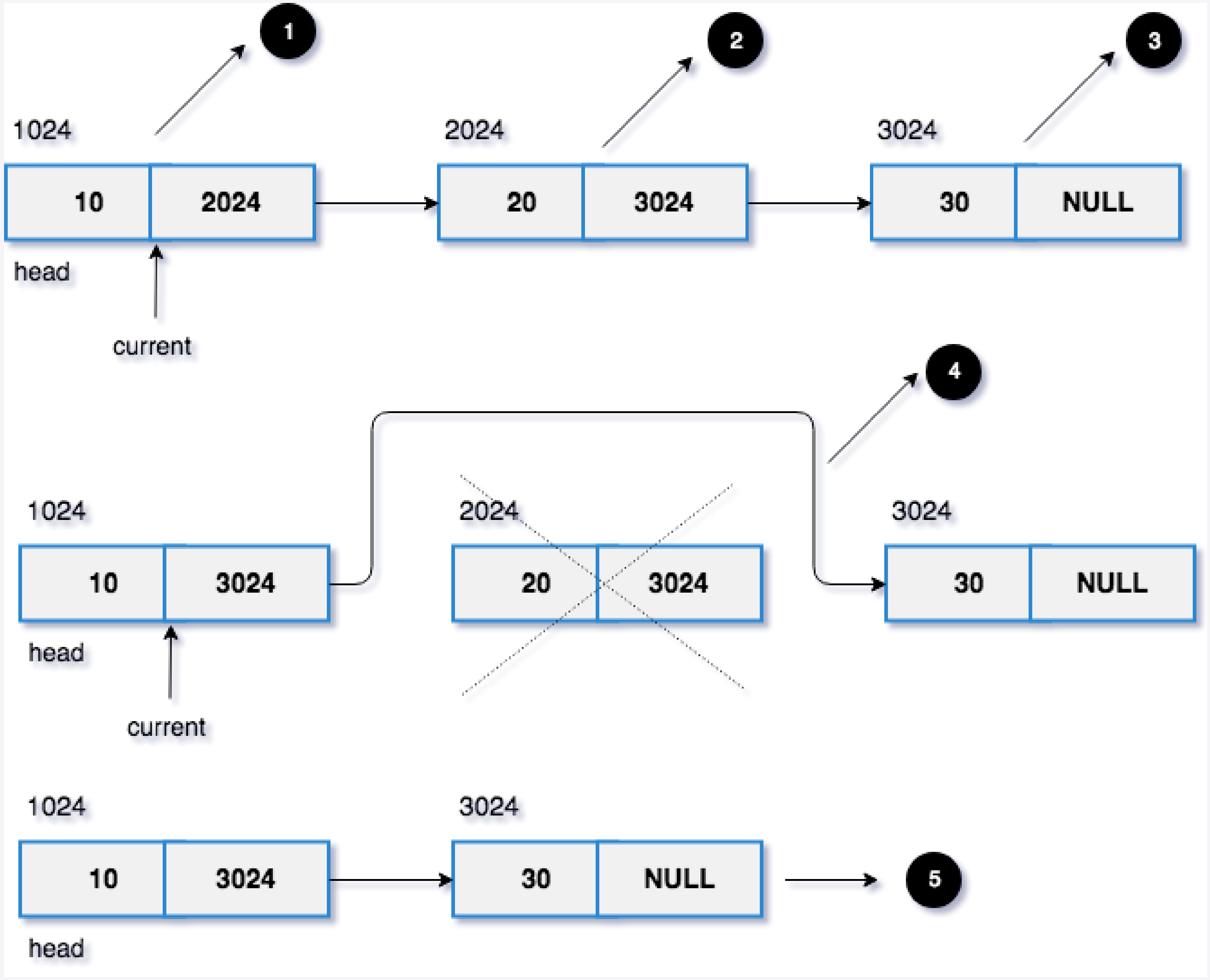
Some examples are shown below.

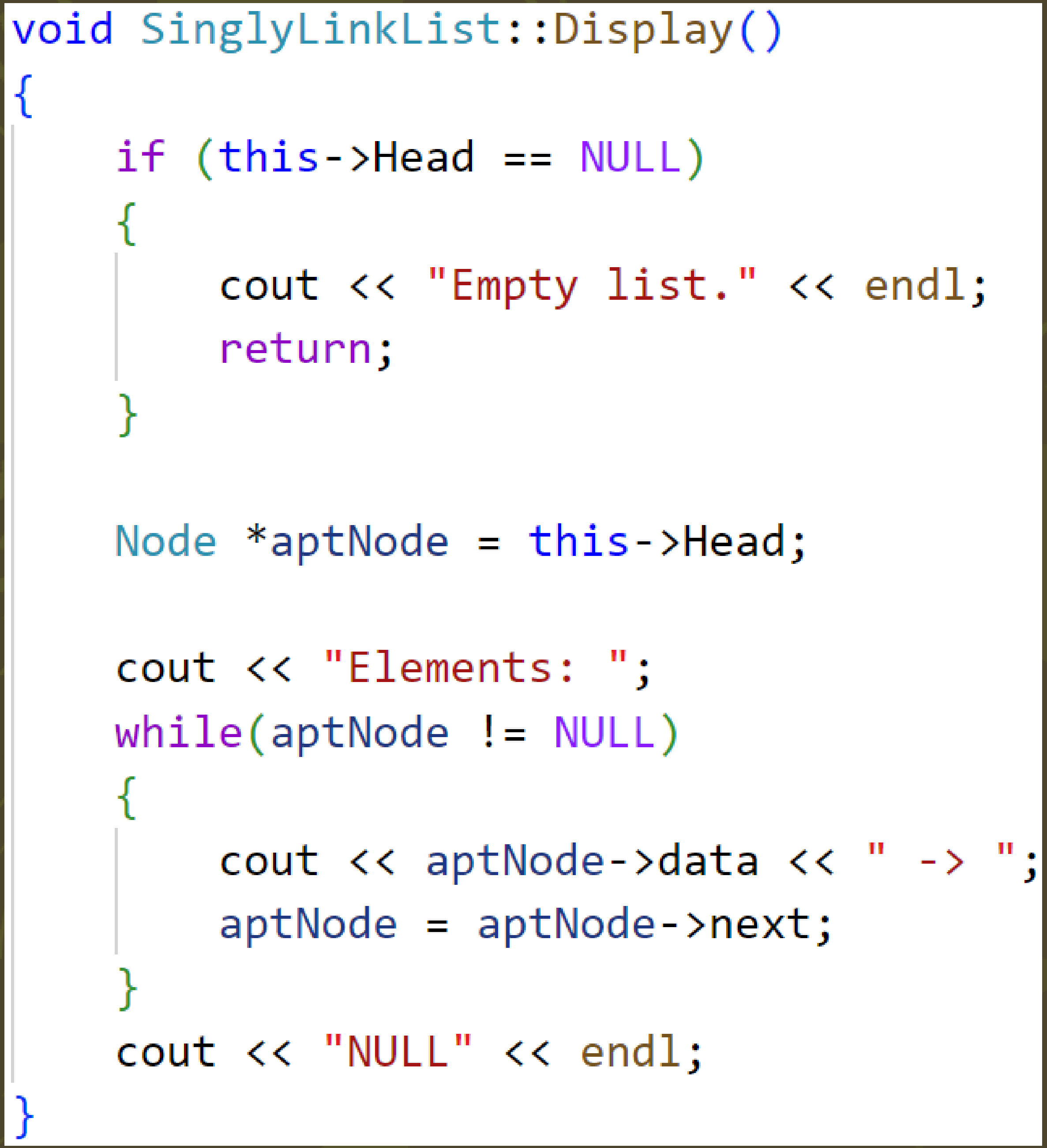
What is the time complexity?

- **Delete Head node (first node)**



- **Delete other nodes (Non-Head)**



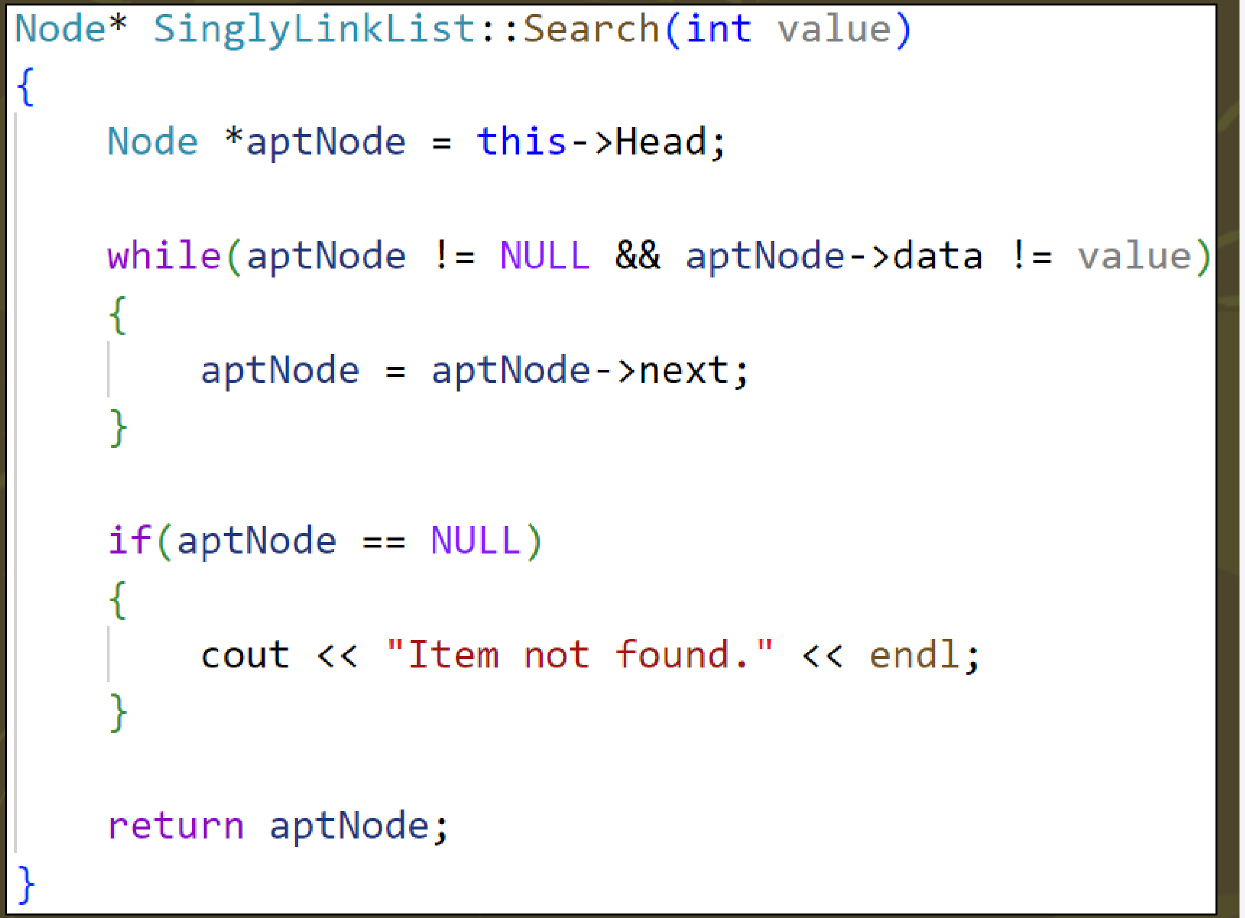


```cpp
void SinglyLinkList::Display()
{
    if (this->Head == NULL)
    {
        cout << "Empty list." << endl;
        return;
    }

    Node *aptNode = this->Head;

    cout << "Elements: ";
    while(aptNode != NULL)
    {
        cout << aptNode->data << " -> ";
        aptNode = aptNode->next;
    }
    cout << "NULL" << endl;
}
```

Display

- To display the entire singly linked list, we need to traverse it from **first** to **last**.
- In contrast to arrays, **linked list nodes cannot be accessed randomly**.

What is the time complexity?



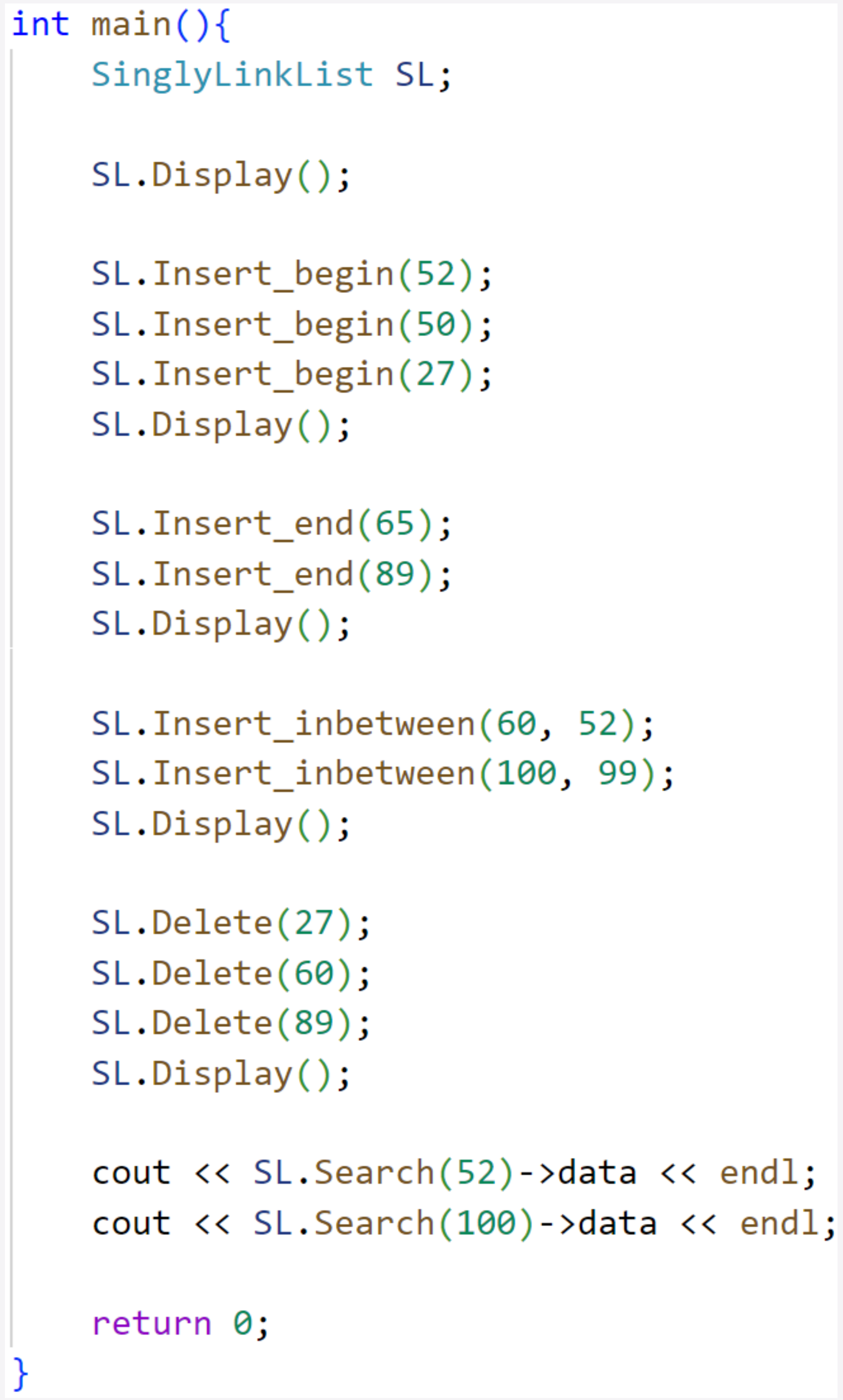
```cpp
Node* SinglyLinkList::Search(int value)
{
    Node *aptNode = this->Head;

    while(aptNode != NULL && aptNode->data != value)
    {
        aptNode = aptNode->next;
    }

    if(aptNode == NULL)
    {
        cout << "Item not found." << endl;
    }

    return aptNode;
}
```

Search

- We need to traverse the linked list right from the start as well.
- At each node, we perform a **lookup** to determine if the target has been found, if **yes**, then we **return the target node else** we **move to the next element**.

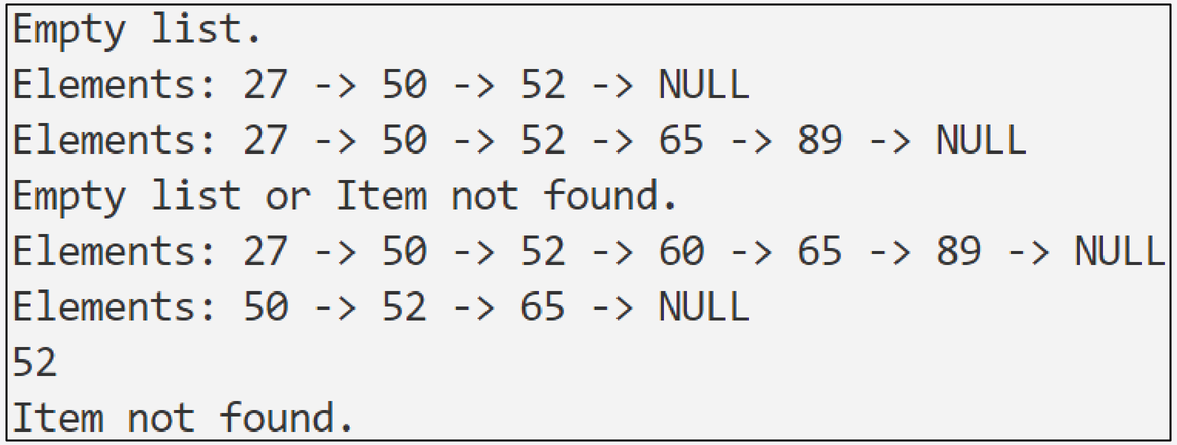
What is the time complexity?

Program body:



```
int main(){
    SinglyLinkList SL;

    SL.Display();

    SL.Insert_begin(52);
    SL.Insert_begin(50);
    SL.Insert_begin(27);
    SL.Display();

    SL.Insert_end(65);
    SL.Insert_end(89);
    SL.Display();

    SL.Insert_inbetween(60, 52);
    SL.Insert_inbetween(100, 99);
    SL.Display();

    SL.Delete(27);
    SL.Delete(60);
    SL.Delete(89);
    SL.Display();

    cout << SL.Search(52)->data << endl;
    cout << SL.Search(100)->data << endl;

    return 0;
}
```

Result:

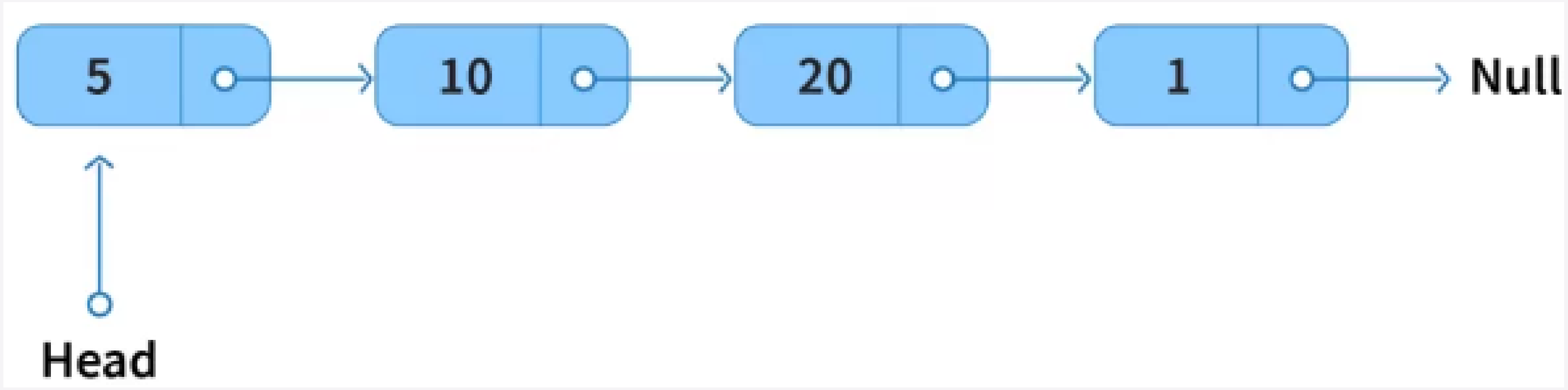


```
Empty list.
Elements: 27 -> 50 -> 52 -> NULL
Elements: 27 -> 50 -> 52 -> 65 -> 89 -> NULL
Empty list or Item not found.
Elements: 27 -> 50 -> 52 -> 60 -> 65 -> 89 -> NULL
Elements: 50 -> 52 -> 65 -> NULL
52
Item not found.
```

Can we do better?

Assume that the entire **singly linked list** is already **sorted** in ascending manner.

- Can we apply the **binary search** on **Linked List** and complete our searching operation in **O(log N) time**?



Can we do better?

Assume that the entire **singly linked list** is already **sorted** in ascending manner.

- Can we apply the **binary search** on **Linked List** and complete our searching operation in **O(log N) time**?

It turns out we can't, even if the linked list is already sorted.

The **binary search require to access to any location** in **constant time**. Any element of a singly linked list can only be accessed in a **sequential manner** making binary search completely ineffective.

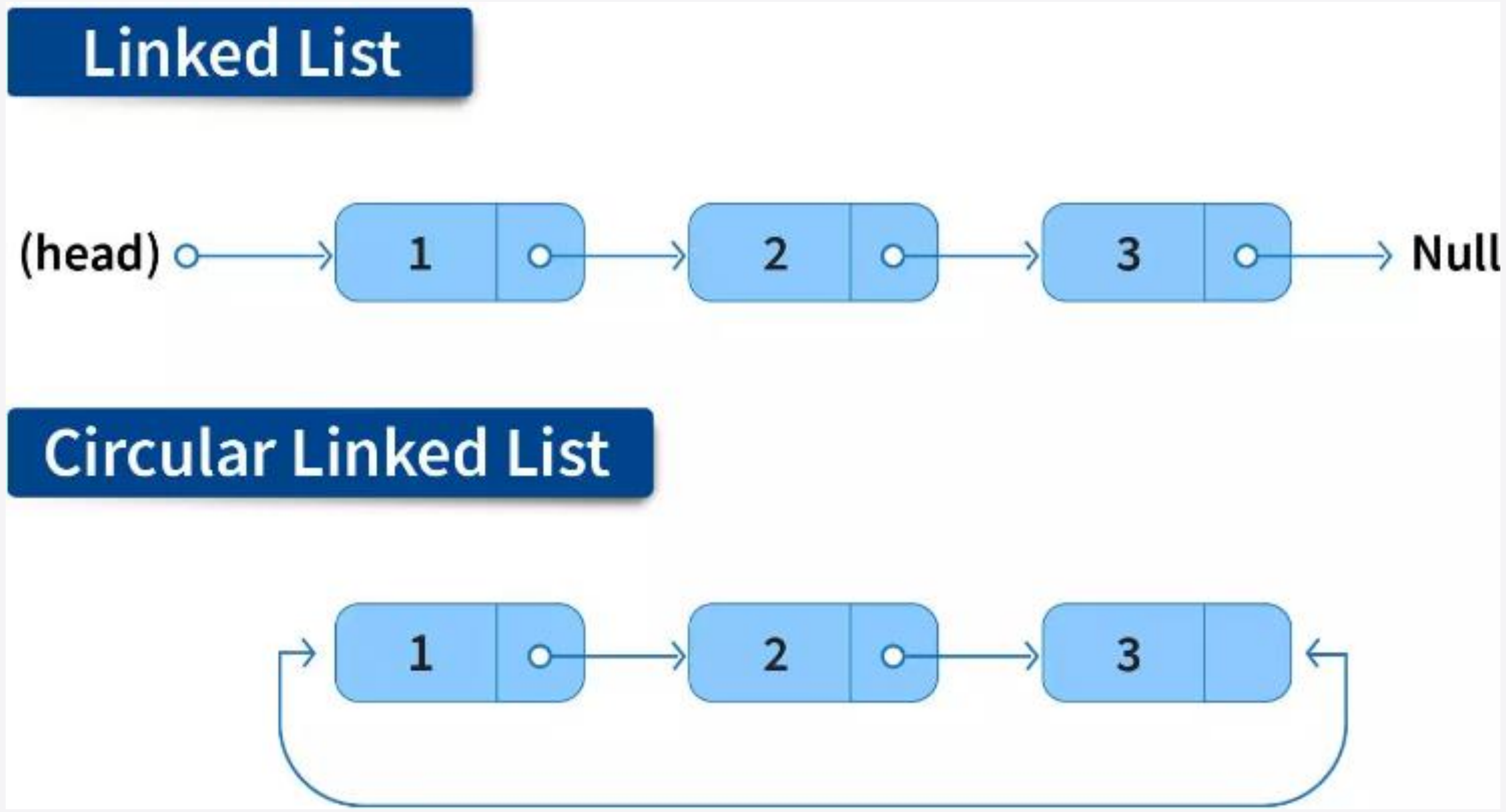
Uses of linked list

Some of the real-life applications of the linked list are as follows:

- Used to store single or bivariable polynomials.
- Act as a base for certain data structures like Queue, Stack, Graph.
- Strategy for file allocation schemes by Operating System.
- Keep track of free space in the secondary disk. All the free spaces can be linked together.
- Turn-based games can use a circular linked list to decide which player is about to be played. Once the player finishes its turn we move to the next player.
- To keep records of items such as music, videos, images, web pages, etc which link to one another and allows to traverse between them sequentially.

Circular linked lists avoids this overhead of **traversing till the Node points to null**, and then using **head pointer** to circle back from start.

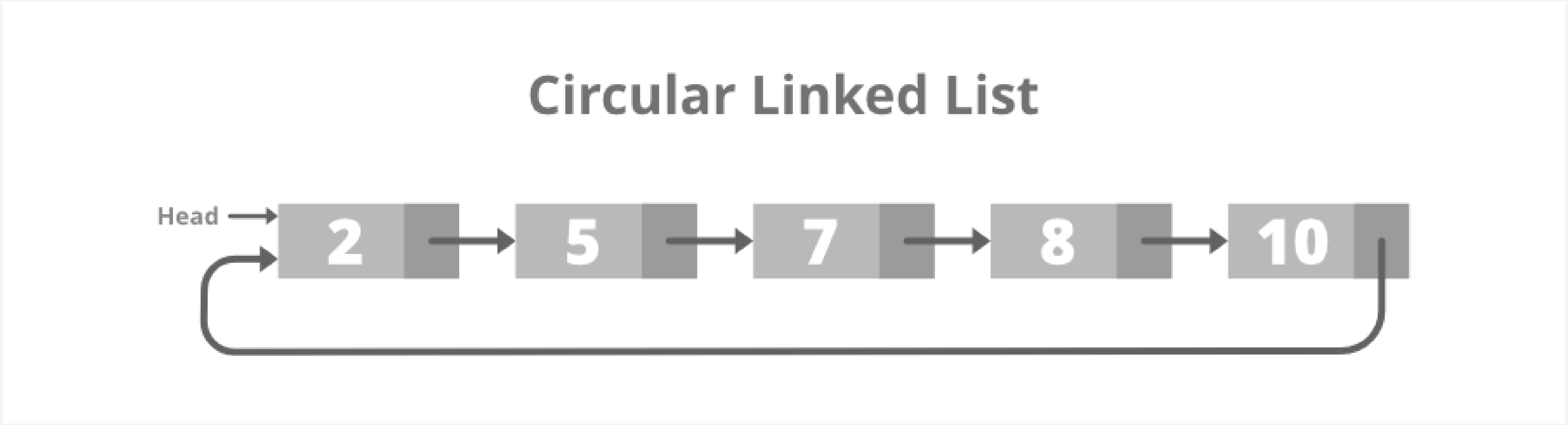
It's the intrinsic property of **Circular Linked lists** which provides this property to circle the Linked List, by default.

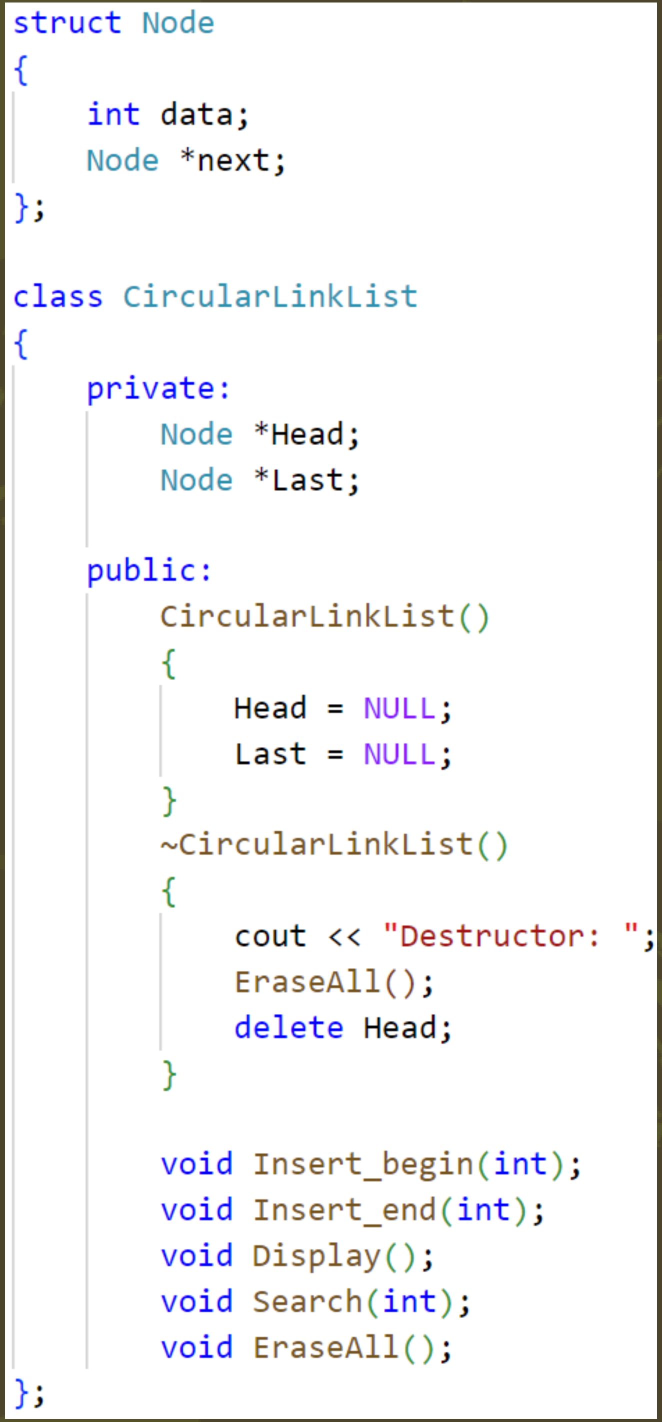


Circular Linked List

Introduction

In a Linked List, it is not very straightforward to **circle around** the list. By circle, we mean that once we have reached the last Node in the list, **we can get the starting Node from the last Node**.

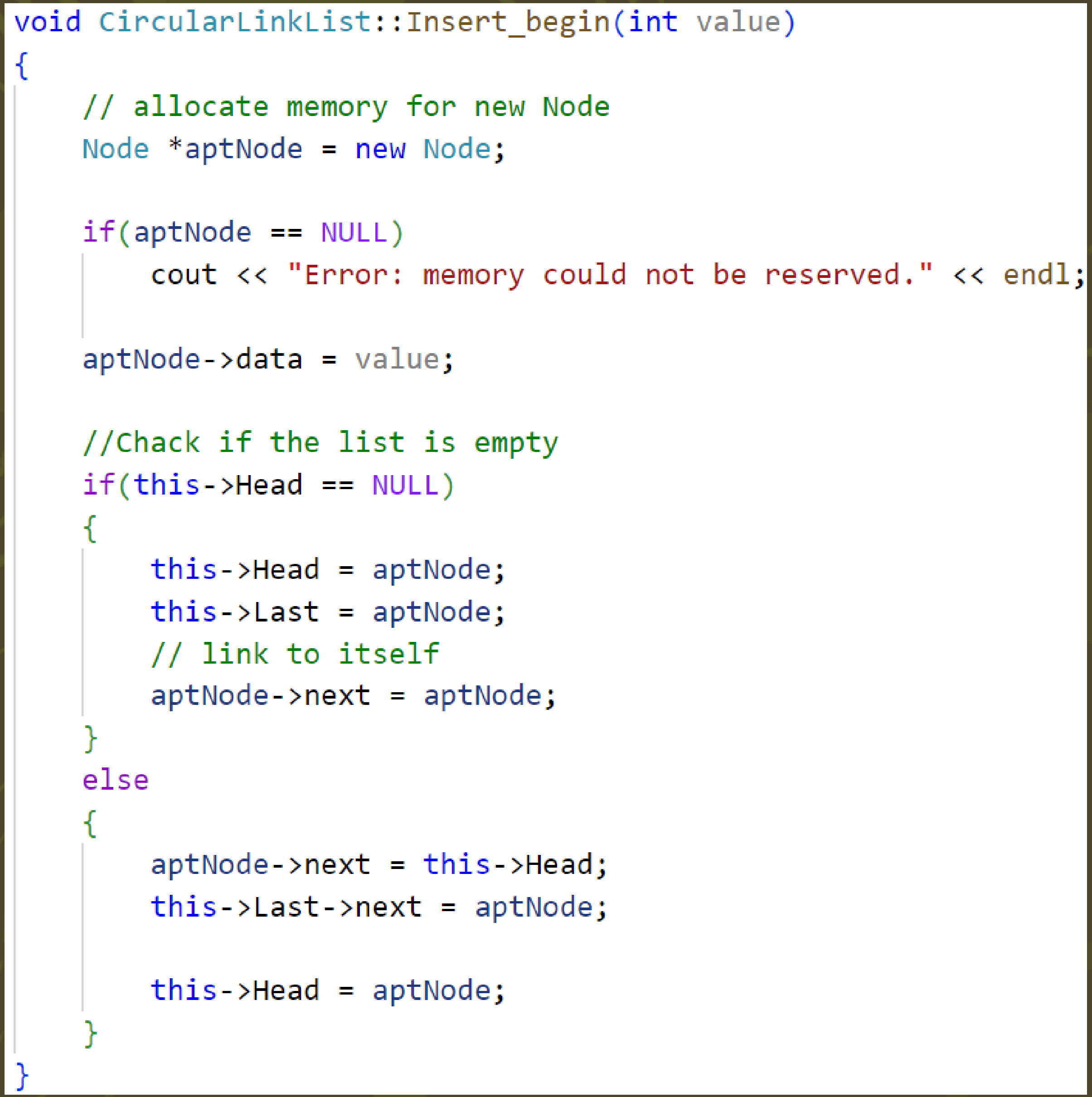




```cpp
struct Node
{
    int data;
    Node *next;
};

class CircularLinkList
{
    private:
        Node *Head;
        Node *Last;

    public:
        CircularLinkList()
        {
            Head = NULL;
            Last = NULL;
        }
        ~CircularLinkList()
        {
            cout << "Destructor: ";
            EraseAll();
            delete Head;
        }

        void Insert_begin(int);
        void Insert_end(int);
        void Display();
        void Search(int);
        void EraseAll();
};
```

- **We can traverse a circular linked list until we reach the same node where we started.**

- The circular linked list has **no beginning** and **no ending** and there is **no NULL value present** in the next part of any of the nodes.

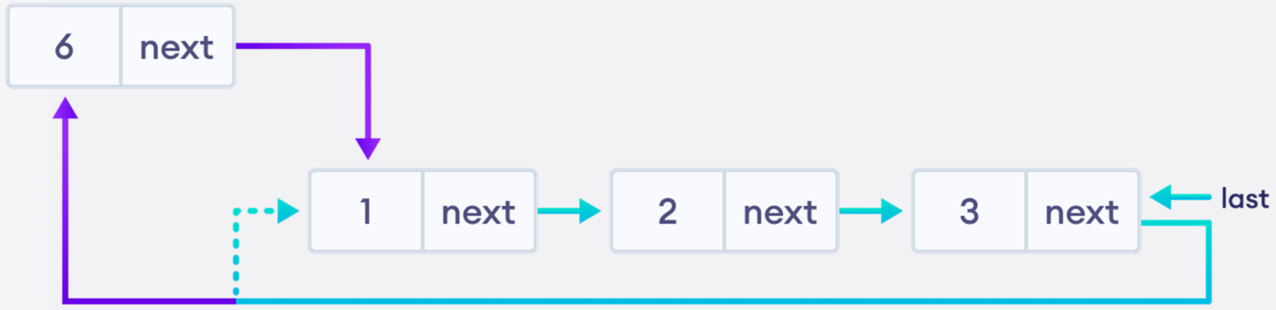


```
void CircularLinkList::Insert_begin(int value)
{
    // allocate memory for new Node
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved." << endl;

    aptNode->data = value;

    //Chack if the list is empty
    if(this->Head == NULL)
    {
        this->Head = aptNode;
        this->Last = aptNode;
        // link to itself
        aptNode->next = aptNode;
    }
    else
    {
        aptNode->next = this->Head;
        this->Last->next = aptNode;

        this->Head = aptNode;
    }
}
```

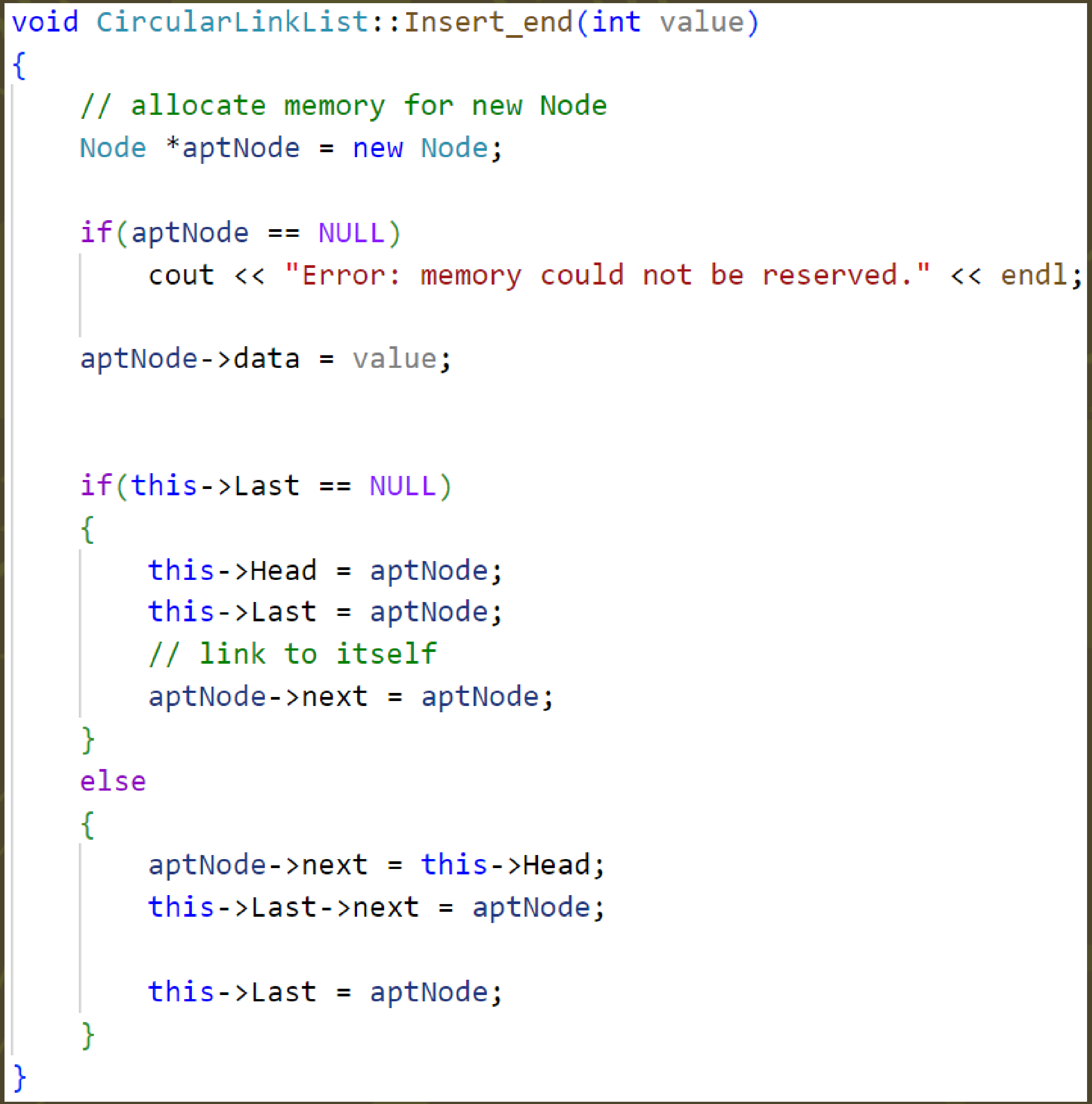
Insert (1/2)

Insertion at the Beginning:

- Get the **current First Node**. The **New Node** can point to this node as its next. Update the **HEAD** pointer.
- The **Last Node** point to the **new Node**, thus honoring the circular property.



What is the time complexity?

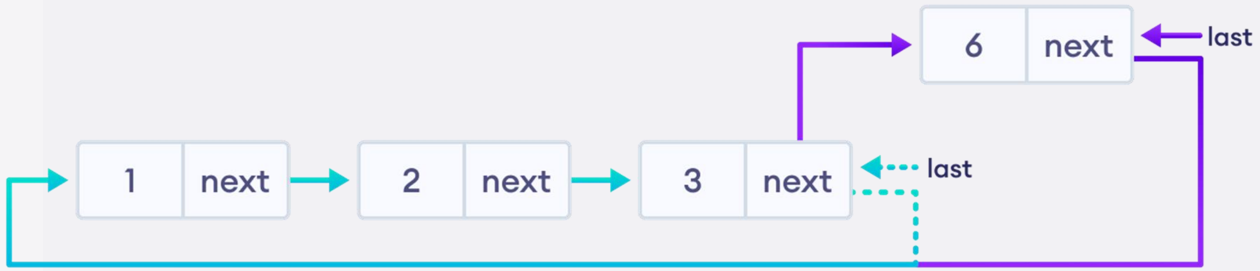


```
void CircularLinkList::Insert_end(int value)
{
    // allocate memory for new Node
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved." << endl;

    aptNode->data = value;


    if(this->Last == NULL)
    {
        this->Head = aptNode;
        this->Last = aptNode;
        // link to itself
        aptNode->next = aptNode;
    }
    else
    {
        aptNode->next = this->Head;
        this->Last->next = aptNode;

        this->Last = aptNode;
    }
}
```

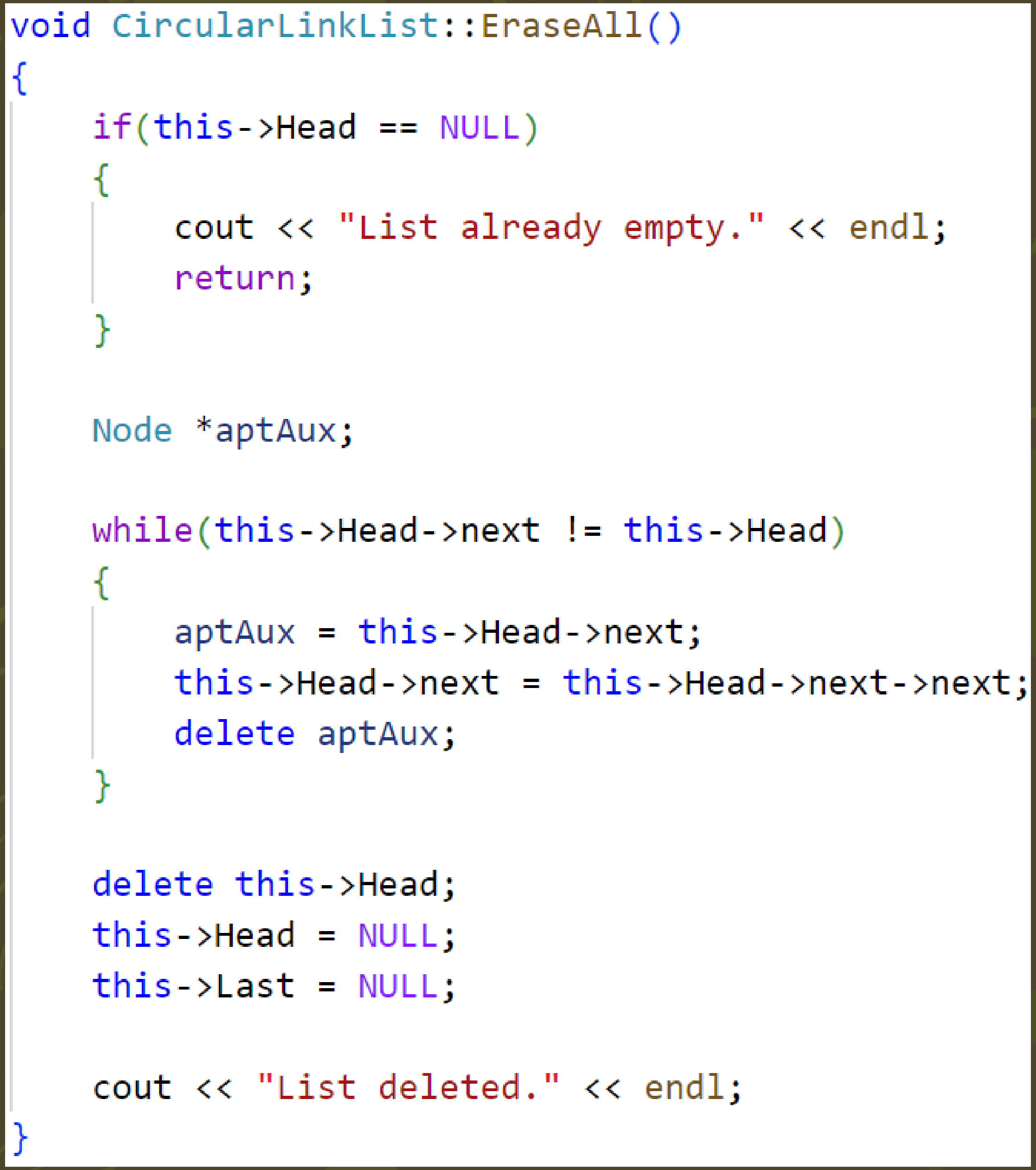
Insert (2/2)

Insertion at the End:

- Similar procedure than **insertion at the Beginning**.
- In this case, the **Last** pointer is updated.



What is the time complexity?



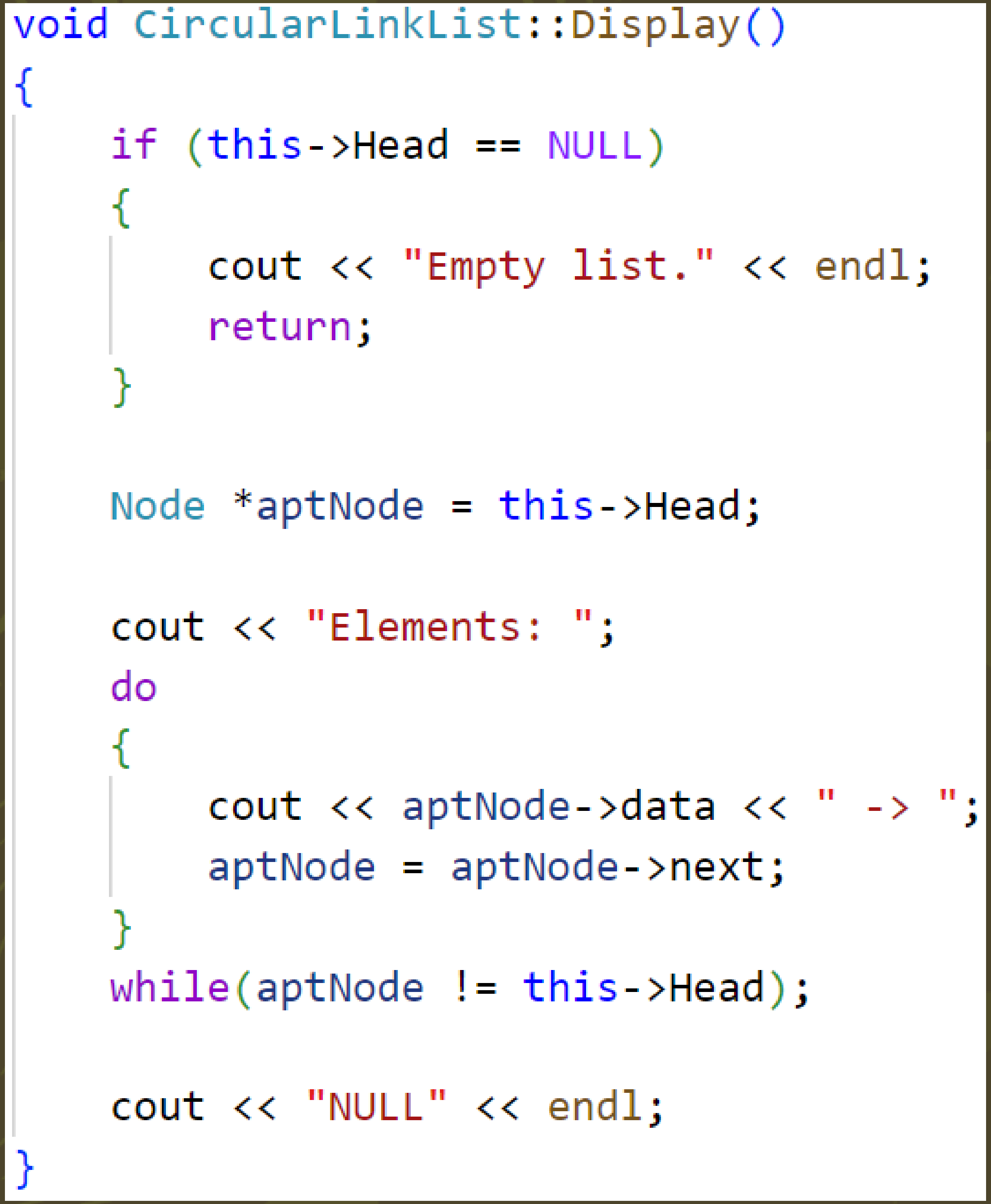
```
void CircularLinkList::EraseAll()
{
    if(this->Head == NULL)
    {
        cout << "List already empty." << endl;
        return;
    }

    Node *aptAux;

    while(this->Head->next != this->Head)
    {
        aptAux = this->Head->next;
        this->Head->next = this->Head->next->next;
        delete aptAux;
    }

    delete this->Head;
    this->Head = NULL;
    this->Last = NULL;

    cout << "List deleted." << endl;
}
```

Erase all

The deletion of all nodes can be formed in the following way,

- From the **Head node**, save the next node.
- Change **the next node of the Head** to the **next node of the saved node**.
- Delete the **saved node**.
- Repeat until **Head** is the only left.
- Delete **Head**.

What is the time complexity?



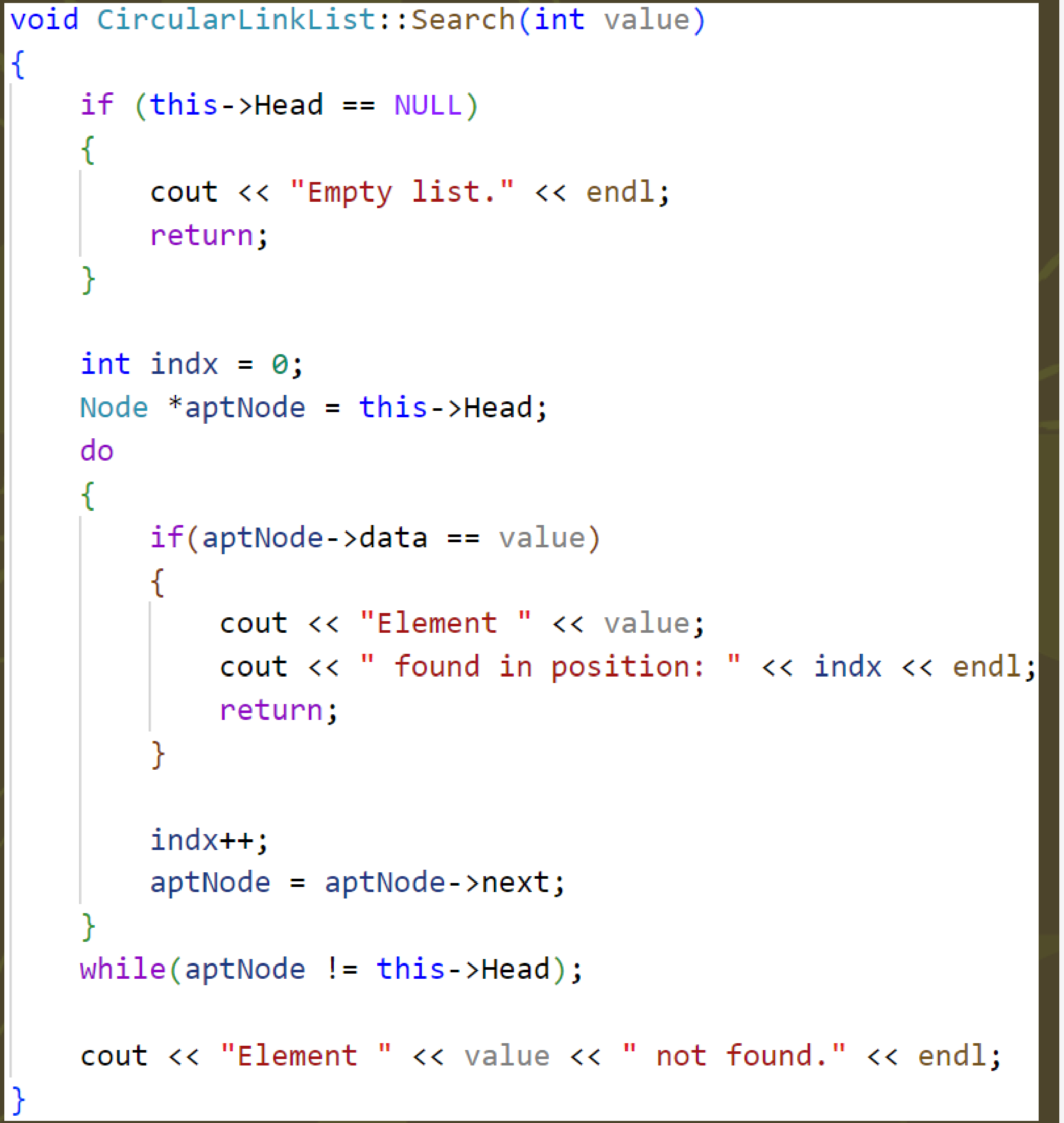
```
void CircularLinkList::Display()
{
    if (this->Head == NULL)
    {
        cout << "Empty list." << endl;
        return;
    }

    Node *aptNode = this->Head;

    cout << "Elements: ";
    do
    {
        cout << aptNode->data << " -> ";
        aptNode = aptNode->next;
    }
    while(aptNode != this->Head);

    cout << "NULL" << endl;
}
```

Display

To display the circular linked list, we need to **transverse from the Head pointer until we reach back to it**.

As we transverse over the Nodes, we can print their values.

What is the time complexity?



```cpp
void CircularLinkList::Search(int value)
{
    if (this->Head == NULL)
    {
        cout << "Empty list." << endl;
        return;
    }

    int indx = 0;
    Node *aptNode = this->Head;
    do
    {
        if(aptNode->data == value)
        {
            cout << "Element " << value;
            cout << " found in position: " << indx << endl;
            return;
        }

        indx++;
        aptNode = aptNode->next;
    }
    while(aptNode != this->Head);

    cout << "Element " << value << " not found." << endl;
}
```

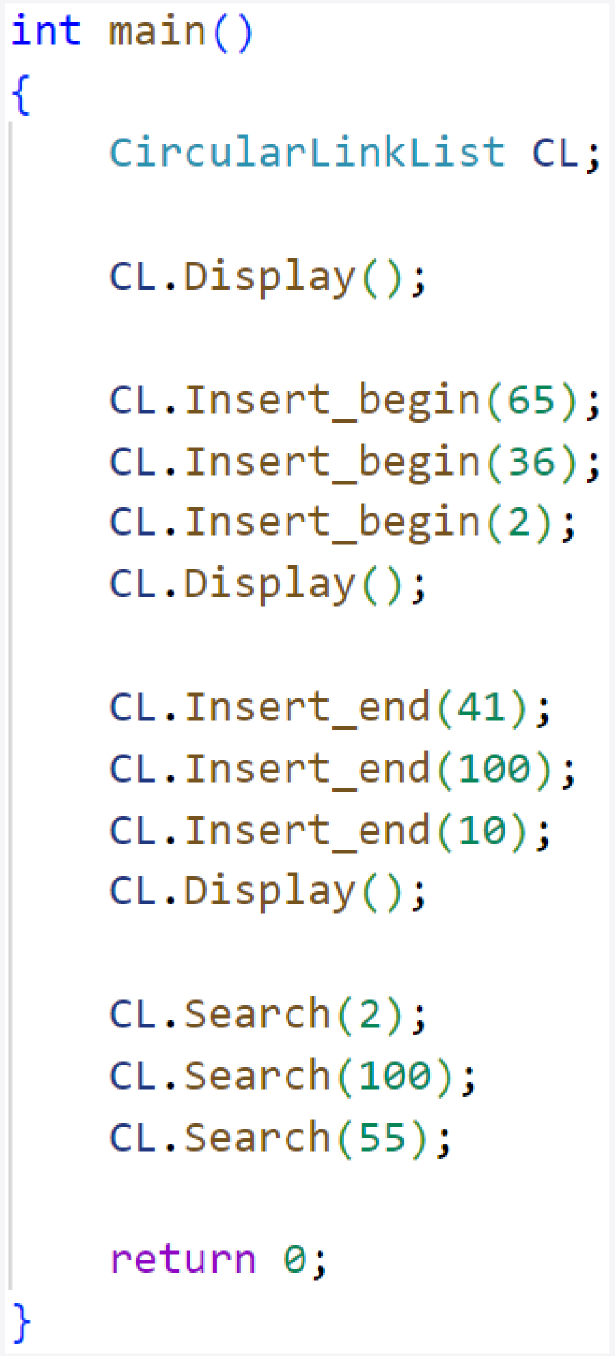
Search

To search an element in the **circular linked list**, we need to **traverse the linked list right from the Head-to-Head**.

At each node, we perform a **lookup** to determine if the target has been found, if **yes**, then **we print the index else we move to the next element**.

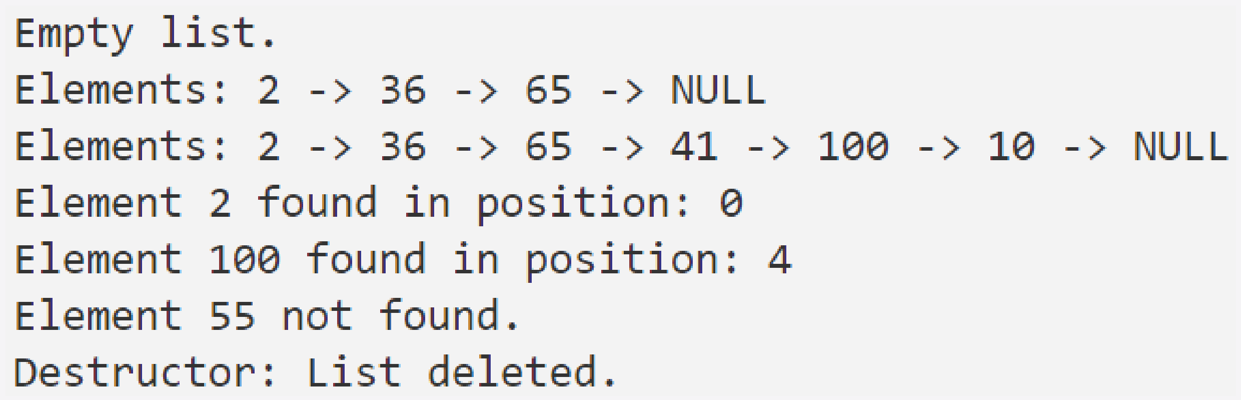
What is the time complexity?

Program body:



```
int main()
{
    CircularLinkList CL;

    CL.Display();

    CL.Insert_begin(65);
    CL.Insert_begin(36);
    CL.Insert_begin(2);
    CL.Display();

    CL.Insert_end(41);
    CL.Insert_end(100);
    CL.Insert_end(10);
    CL.Display();

    CL.Search(2);
    CL.Search(100);
    CL.Search(55);

    return 0;
}
```

Result:



```
Empty list.
Elements: 2 -> 36 -> 65 -> NULL
Elements: 2 -> 36 -> 65 -> 41 -> 100 -> 10 -> NULL
Element 2 found in position: 0
Element 100 found in position: 4
Element 55 not found.
Destructor: List deleted.
```

Applications of Circular linked list

Some applications can be:

- In **multiplayer games**, each player is represented as a Node. This way, each player is given a chance to play, as we can easily shuffle back to the first player from the last player quickly, utilizing the circular property.

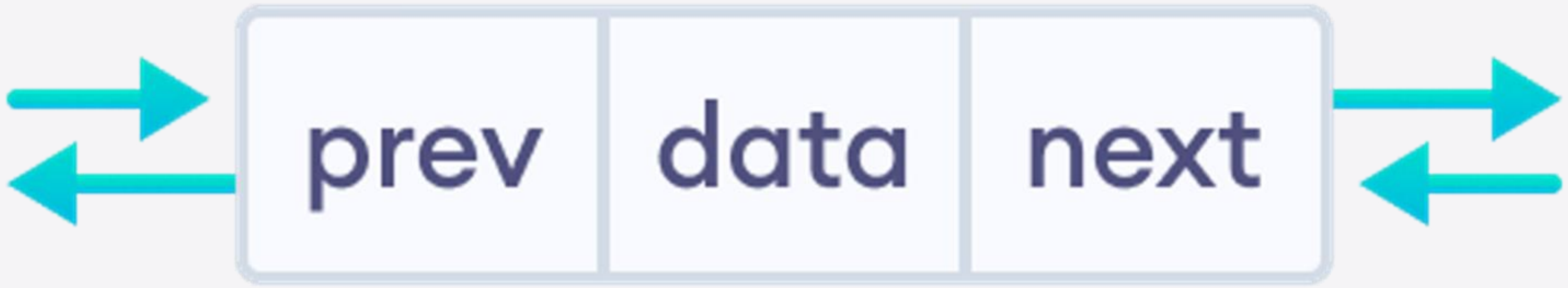
- Similarly, **operation systems** utilizes this concept, where all these applications are placed in the circular linked list, and without worrying about reaching to the end of the running applications, it can easily circle back the applications at the start.

Double Linked List

Introduction

A **doubly linked list** is a type of **linked list** in which each **node** consists of 3 components:

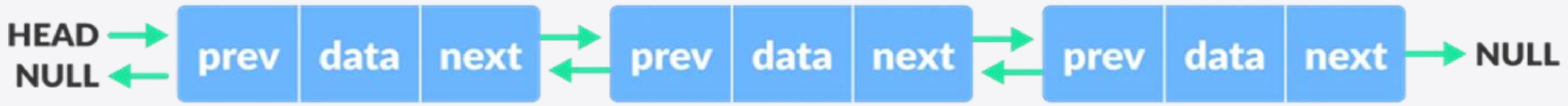
- ***prev** - address of the previous node
- **data** - data item
- ***next** - address of next node

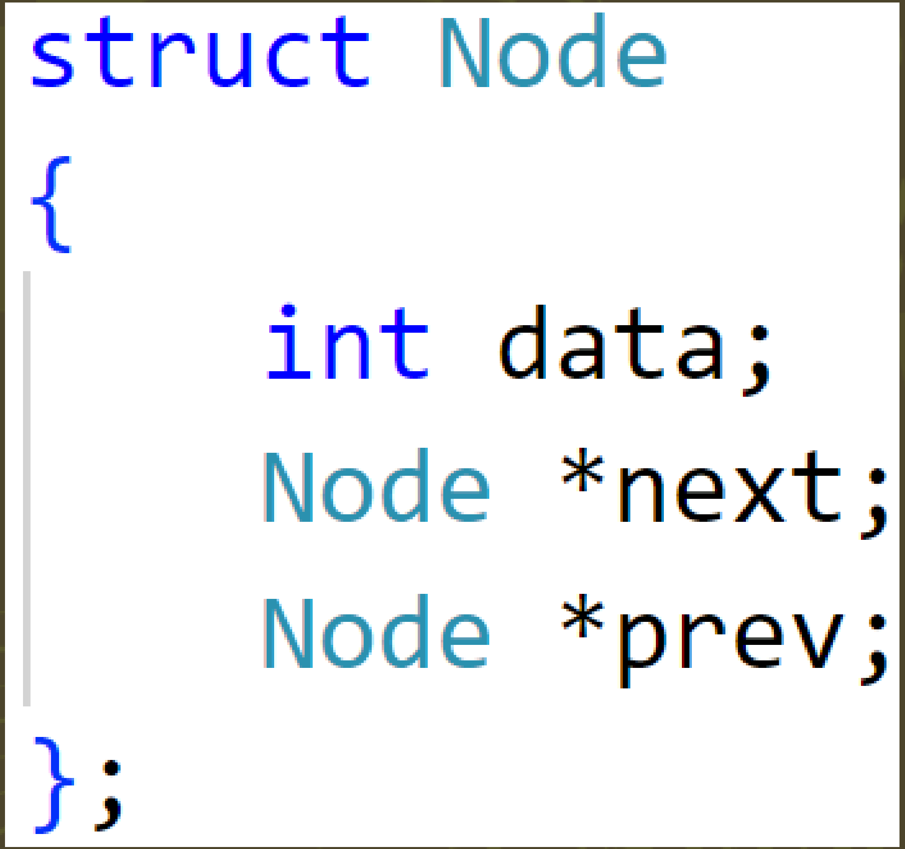


Representation of Doubly Linked List

We **add a pointer to the previous node in a doubly-linked** list. Each struct node has a **data item**, a **pointer to the previous struct node**, and a **pointer to the next struct node**.

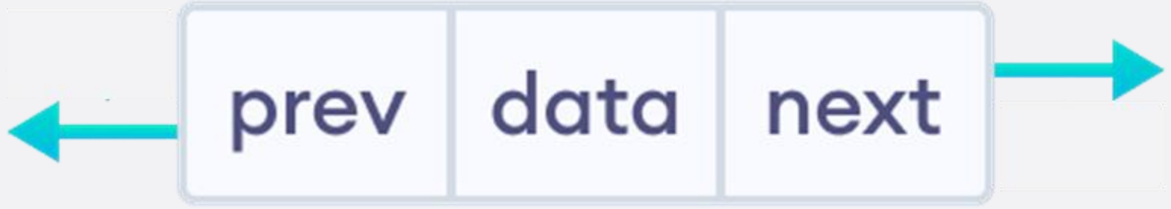
Thus, we can go in either direction: **forward** or **backward**.



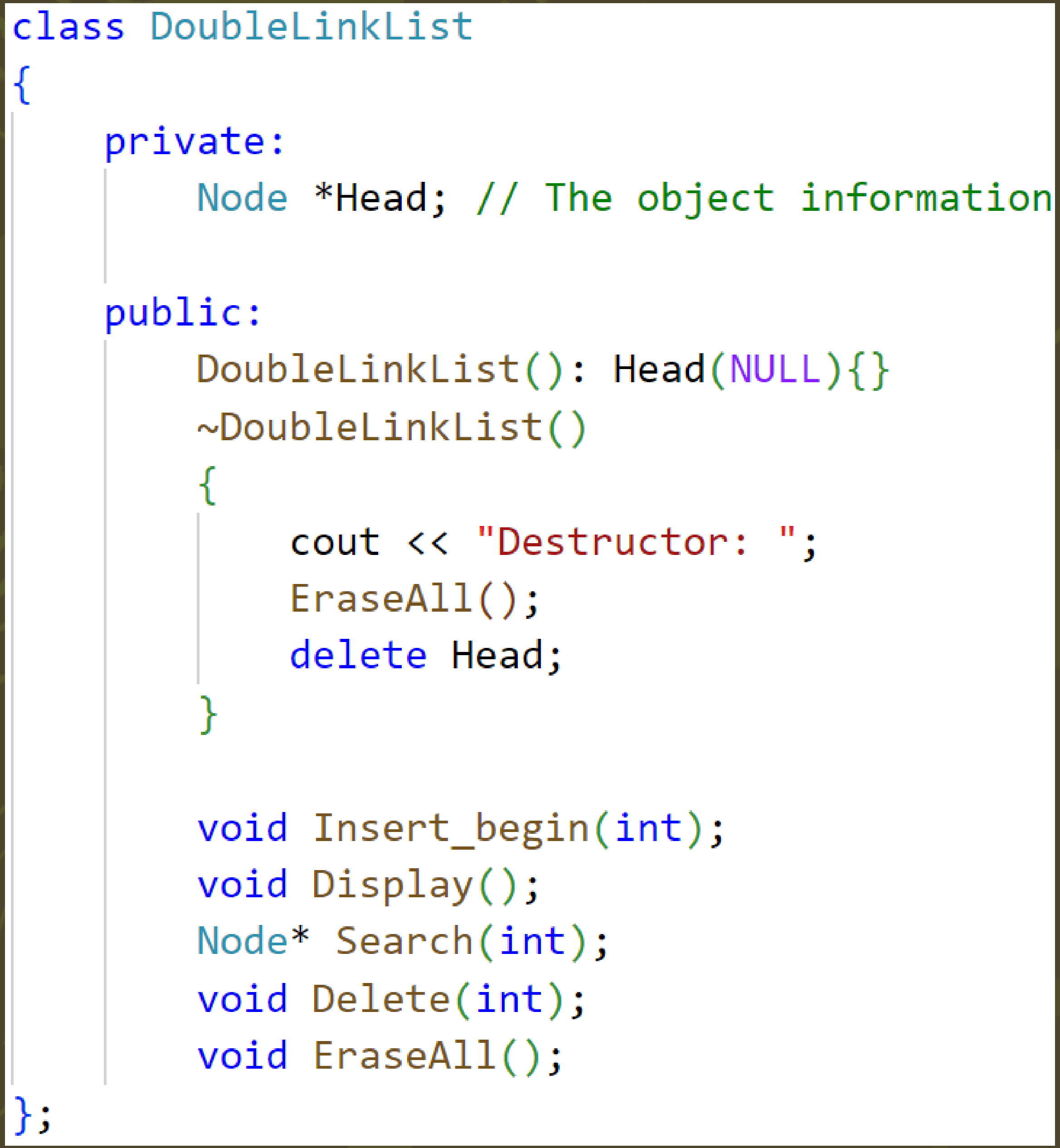


```
struct Node
{
    int data;
    Node *next;
    Node *prev;
};
```

- Here, the **Node structure** represent each single node as



- Each **struct node** has a **data item**, a **pointer to the previous struct node**, and a **pointer to the next struct node**.



```
class DoubleLinkList
{
    private:
        Node *Head; // The object information

    public:
        DoubleLinkList(): Head(NULL){}
        ~DoubleLinkList()
        {
            cout << "Destructor: ";
            EraseAll();
            delete Head;
        }

        void Insert_begin(int);
        void Display();
        Node* Search(int);
        void Delete(int);
        void EraseAll();
};
```

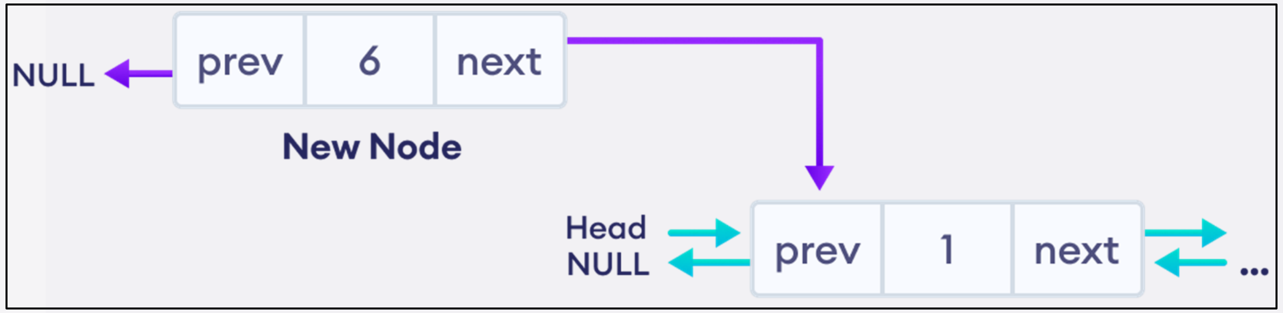
We recreate a **double linked list** through a **DoubleLinkList class** where we implement all the functionalities of this data structure.

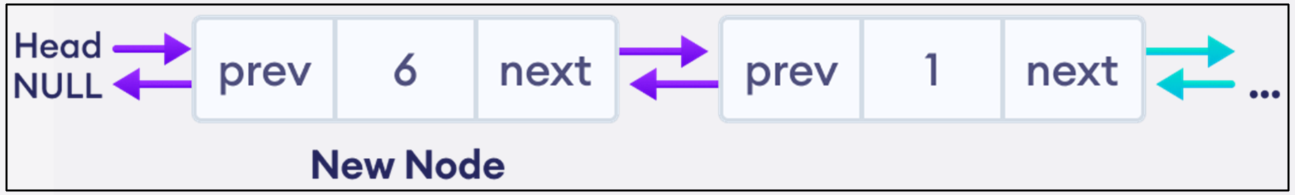
In the case of the **Head** node, **prev** points to **NULL**, and in the case of the **last node**, **next** points to **NULL**.

```cpp
void DoubleLinkList::Insert_begin(int value)
{
    // allocate memory for newNode
    Node *aptNode = new Node;

    if(aptNode == NULL)
        cout << "Error: memory could not be reserved.\n";

    // assign data to newNode
    aptNode->data = value;
    // newNode's next to the first node
    aptNode->next = Head;
    // newNode's prev to NULL
    aptNode->prev = NULL;

    // If list not empty, the first Node's prev to newNode
    if(Head != NULL)
        Head->prev = aptNode;

    // Head to newNode
    Head = aptNode;
}
```

Insert at the beginning

Insertion of a new node at the start is carried out in the following manner:



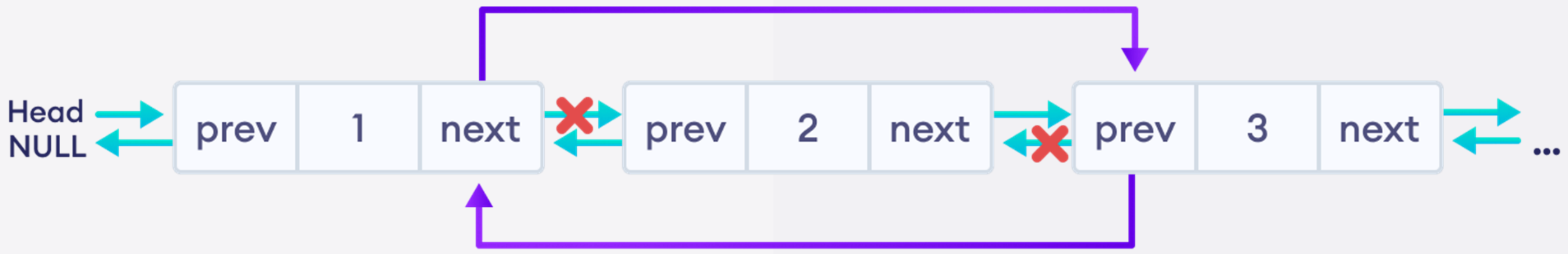


What about ***insertion at the end*** *and* ***after some node****?*

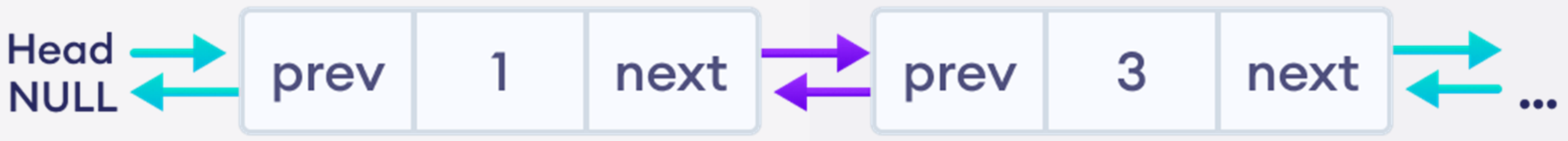
Deletion of a specific node

1. Strictly Delete an Inner Node

If the node to be deleted is an inner node (**second node**)

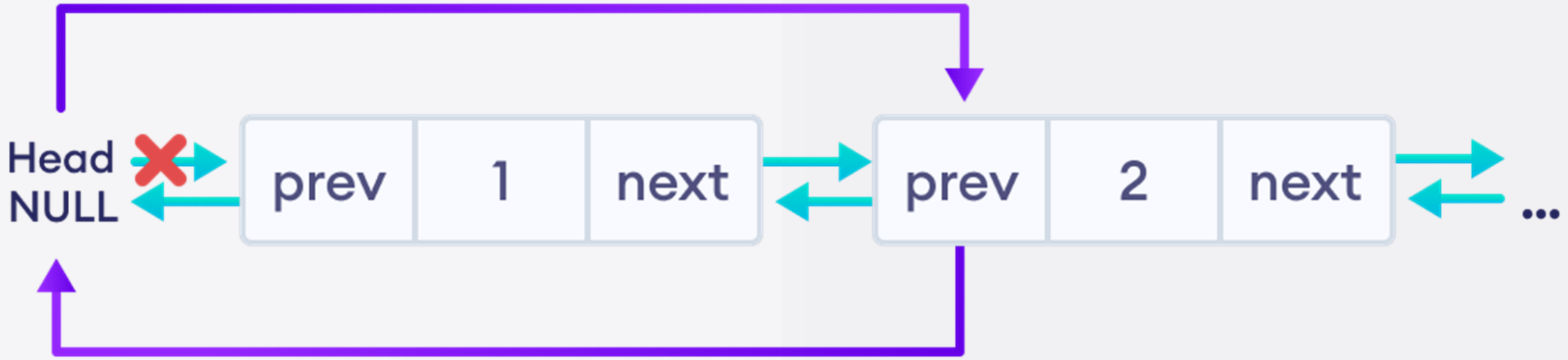


Finally, **free** the memory of the deleted node. And, the final doubly linked list looks like this.

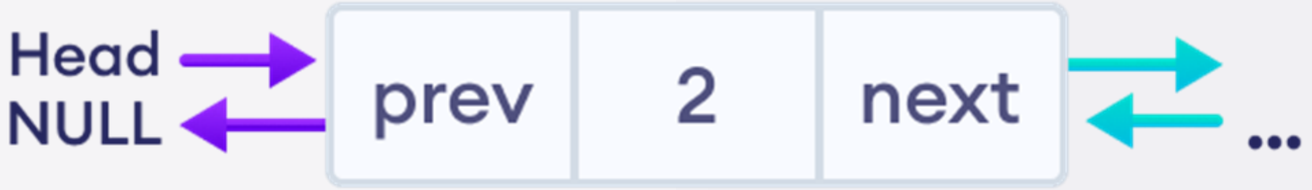


2. Strictly Delete the First Node

If the node to be deleted (i.e., **1**) is at the beginning

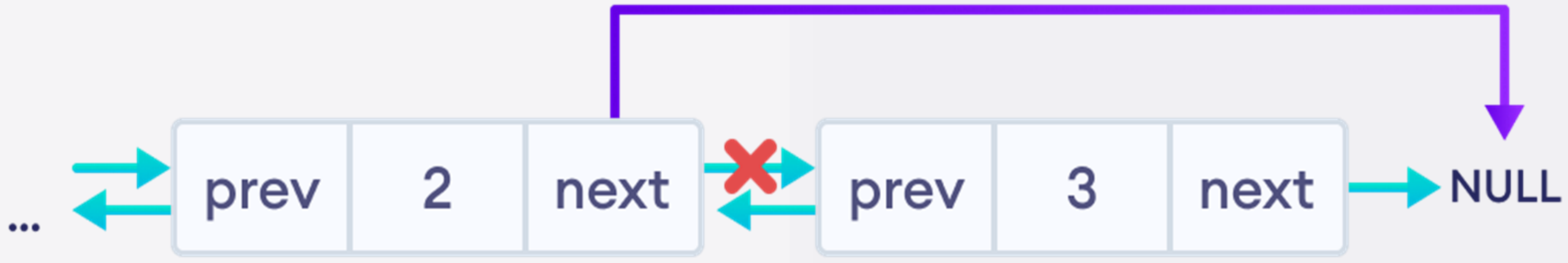


Finally, **free** the memory of the deleted node.

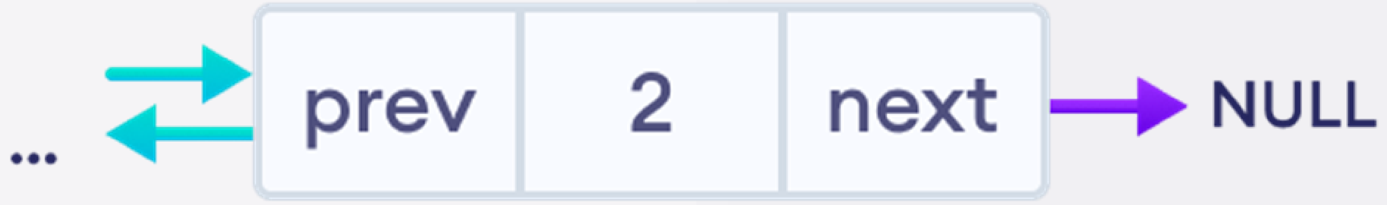


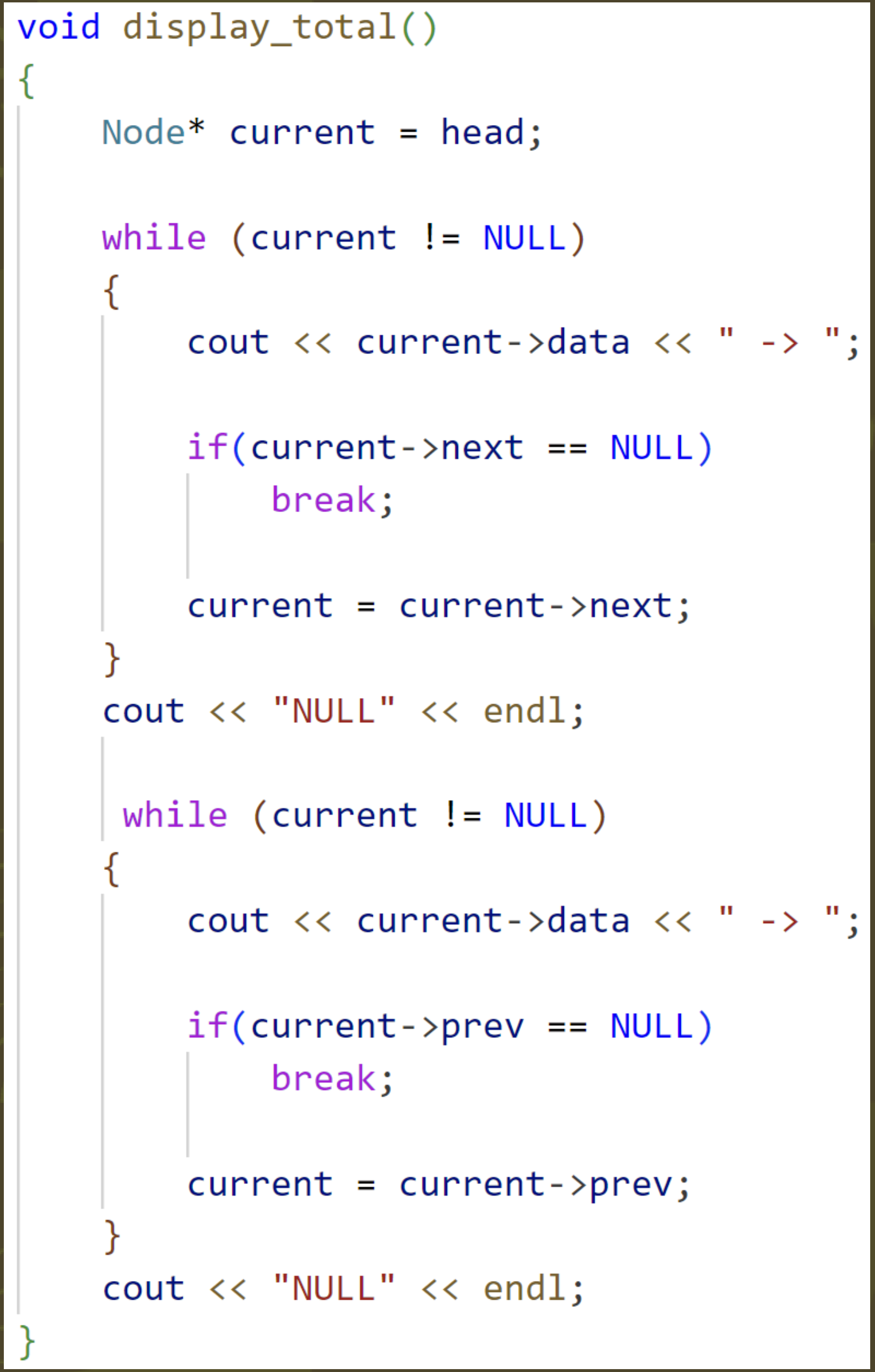
3. Strictly Delete the Last Node

In this case, we are deleting the last node (value **3**) of the doubly linked list.



Finally, **free** the memory of the deleted node. The final doubly linked list looks like this.



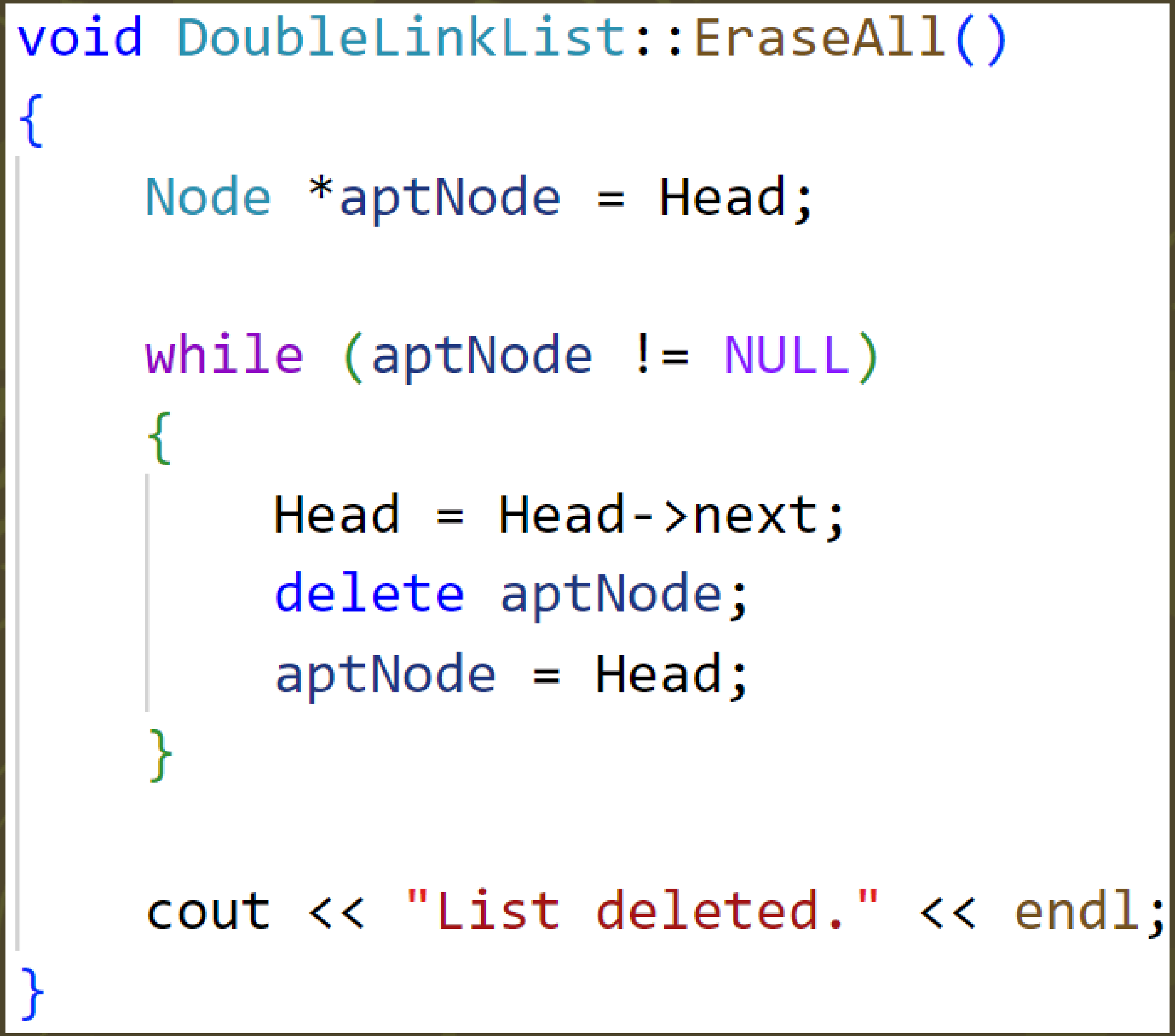


```
void display_total()
{
    Node* current = head;

    while (current != NULL)
    {
        cout << current->data << " -> ";

        if(current->next == NULL)
            break;

        current = current->next;
    }
    cout << "NULL" << endl;

     while (current != NULL)
    {
        cout << current->data << " -> ";

        if(current->prev == NULL)
            break;

        current = current->prev;
    }
    cout << "NULL" << endl;
}
```

Display – Both ways

- To display the entire singly linked list (**both ways**), we need to traverse it from **first** to **last**, then from **last** to **first**.

What is the time complexity?



```
void DoubleLinkList::EraseAll()
{
    Node *aptNode = Head;

    while (aptNode != NULL)
    {
        Head = Head->next;
        delete aptNode;
        aptNode = Head;
    }

    cout << "List deleted." << endl;
}
```

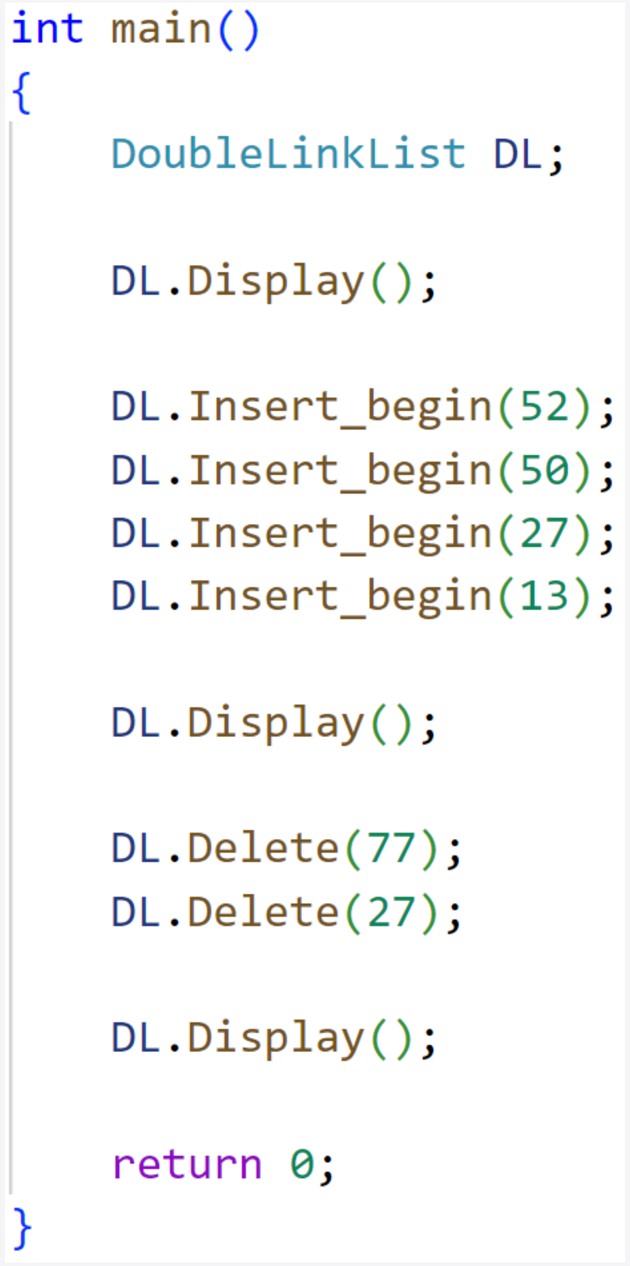
Erase all

The deletion of all nodes can be formed in the following way,

- Save the first node, the **Head node**.
- Change the **Head node** to the **next node**.
- Delete the **saved node**.
- Repeat until **saved node pointer** is **NULL**.

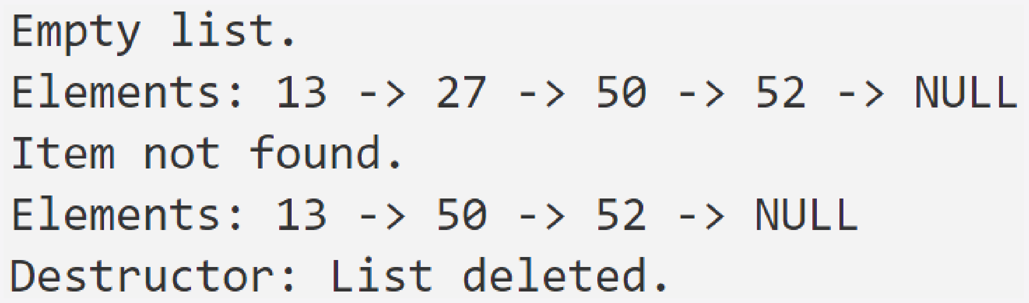
What is the time complexity?

Program body:



```
int main()
{
    DoubleLinkList DL;

    DL.Display();

    DL.Insert_begin(52);
    DL.Insert_begin(50);
    DL.Insert_begin(27);
    DL.Insert_begin(13);

    DL.Display();

    DL.Delete(77);
    DL.Delete(27);

    DL.Display();

    return 0;
}
```

Result:



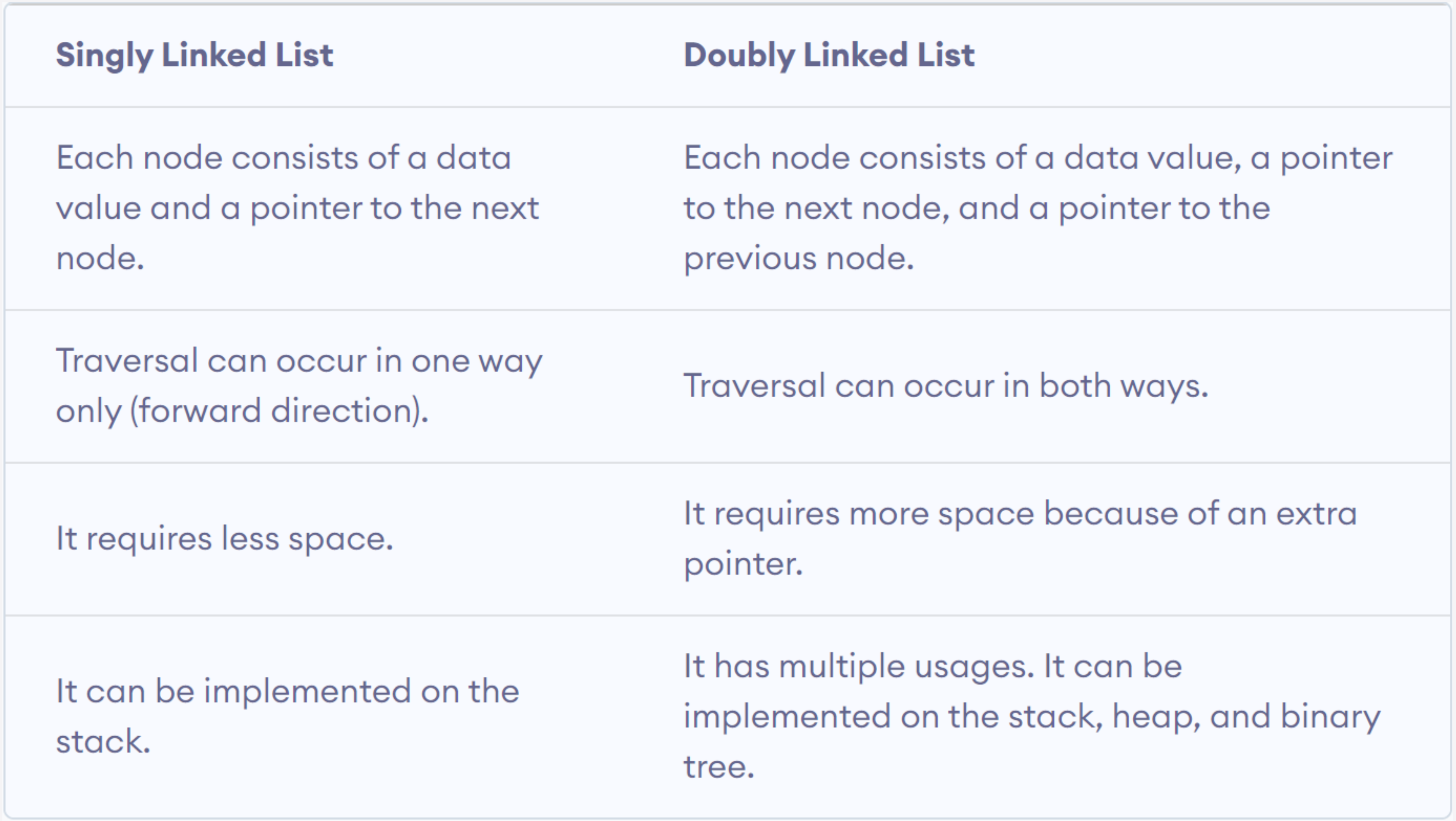
```
Empty list.
Elements: 13 -> 27 -> 50 -> 52 -> NULL
Item not found.
Elements: 13 -> 50 -> 52 -> NULL
Destructor: List deleted.
```

Doubly Linked List Applications

Some of the real-life applications of the double linked list are as follows:

- Redo and undo functionality in software.
- Forward and backward navigation in browsers.
- For navigation systems where forward and backward navigation is required.

Singly Linked List Vs Doubly Linked List



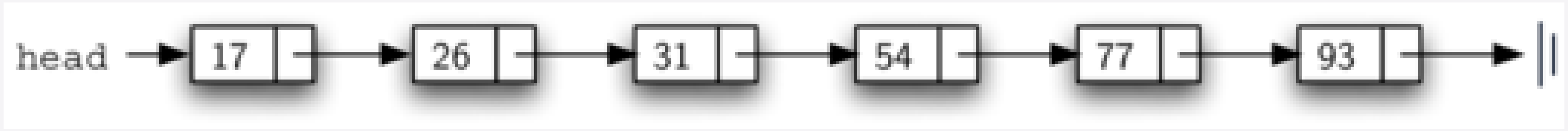
Singly Linked List	Doubly Linked List
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node, and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.

Ordered Linked List

Introduction

In order to implement the ordered linked list, we must remember that the **relative positions of the items**.

The ordered linked list of integers (**17**, **26**, **31**, **54**, **77**, **93**) can be represented by a linked structure as

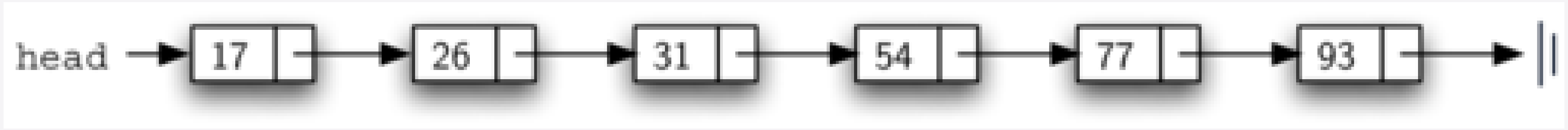


To implement the **OrderedList class**, we will use the same technique as seen previously with unordered linked lists. Once again, an **empty linked list** will be denoted by a **Head** reference to **NULL**.

Search

In **unordered linked list**, we traverse the nodes one at a time until we either **find the item** or **run out of nodes** (**NULL**).

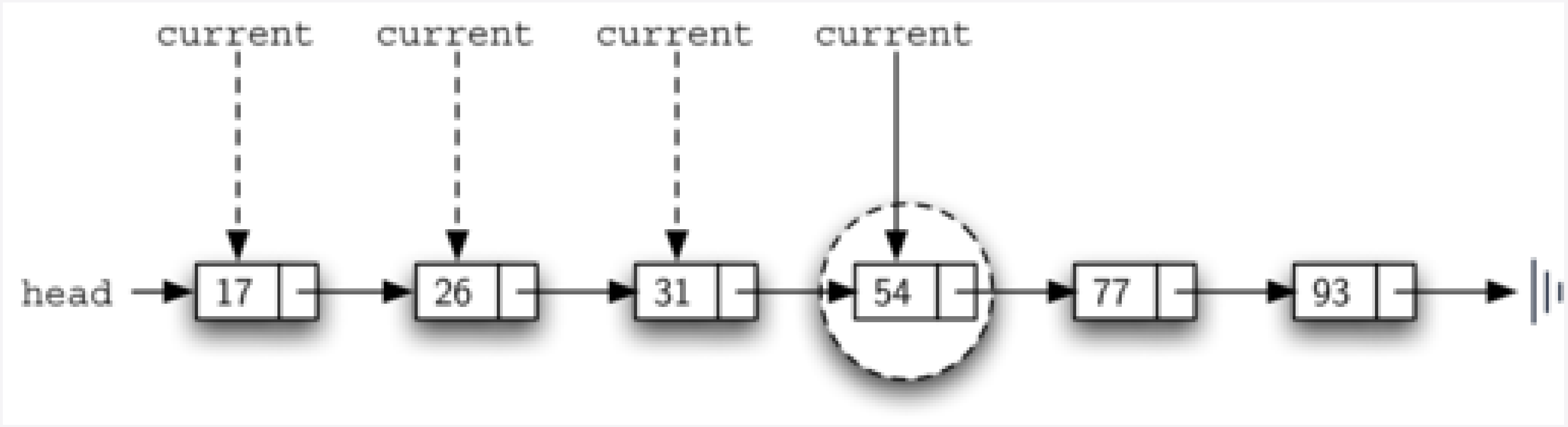
The same approach would work with the **ordered linked list**. However, **if the item is not in the linked list**, we can take advantage of the ordering to stop the search as soon as possible.



Search - Example

If we search for the **value 45** in an **ordered linked list**. As we traverse the list and compare the values. Eventually, we get to the **value 54**.

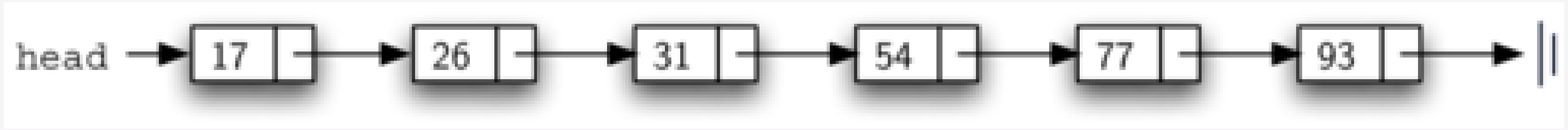
In **unordered linked list**, our former strategy would be to move forward. However, since this is an **ordered linked list**, if the value is greater than the item we are searching for, the **search can stop** and return **False**.



Insertion

In **unordered linked list**, the add method could simply place a new node at the head of the linked list. The easiest point of access.

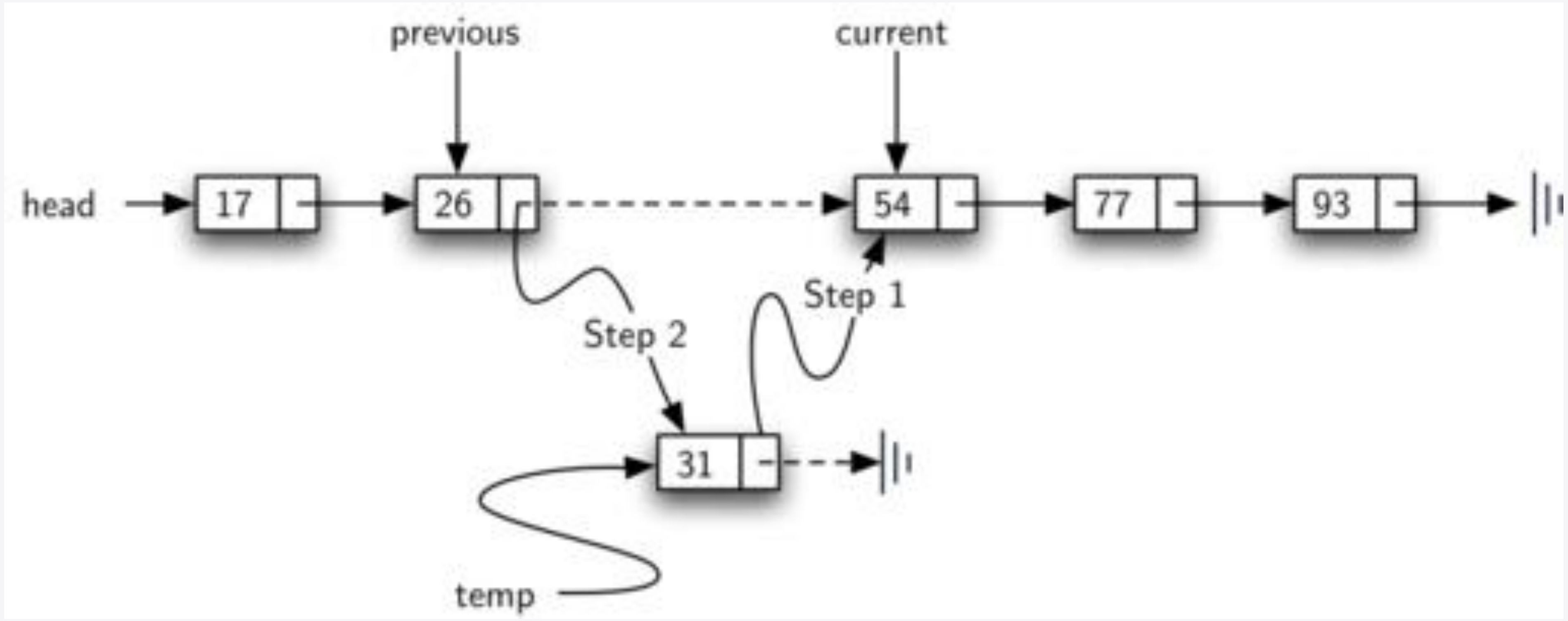
Unfortunately, this will no longer work with **ordered linked lists**. It is now necessary that we discover the specific place where a new item belongs in the existing ordered linked list.



Insertion - Example

If we wanted to add the **value 31**.

We need to traverse the linked list looking for the place where the new node will be added. Here, either we run **out of nodes** (**NULL**) or **the node's value is greater** than the item to add.



Other Operations

Many of the ordered list operations are the same as those of the **unordered linked list**.

- **remove(item)** removes the **item** from the list. It needs the item and modifies the list. Assume the item is present in the list.
- **isEmpty()** tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- **size()** returns the number of items in the list. It needs no parameters and returns an integer.
- **index(item)** returns the position of **item** in the list. It needs the item and returns the index. Assume the item is in the list.
- **pop()** removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- **pop(pos)** removes and returns the item at position **pos**. It needs the position and returns the item. Assume the item is in the list.

Act 2.1
- Linear data structure ADT execution

What do I have to do?

Individually, implement an **Double Link List** following the interface specs:

According to the linear data structure,The operations CRUD (Create, Read (search), Update, Delete) elements must be implemented in the data structure.

In this case, you must implement:

- **Insert**
- **Search**
- **Update**
- **Delete**
- **Delete All (**Free up all the dynamic memory space used**)**

Act 2.3
- Integral activity linear data structure (Competency Evidence)

What do we have to do?

In **teams** of three, make an application that:

- Open the input file **"Bitacora.txt"**, which contains a server access log, with each entry in the format:

Month Day Time IP_address:Port_number Access_error_message

```
Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest
Aug 28 23:07:49 897.53.984.6:6710 Failed password for root
Aug 4 03:18:56 960.96.3.29:5268 Failed password for admin
Jun 20 13:39:21 118.15.416.57:4486 Failed password for illegal user guest
...
```

- Sort the information by **IP address (xxx.xxx.xxx.xxx)** using an **Ordered Double Linked List**. compare **>**xxx, **>**xxx, **>**xxx, **>**xxx.
- Ask the user for the information search in that range of IPs (**Initial IP**, **final IP**).
- Display all the IPs within that range. **Sorted in ascending order**.
- Store the sorting result in a file. **Txt file**.