Algorithms

Linear data structures – Linked lists

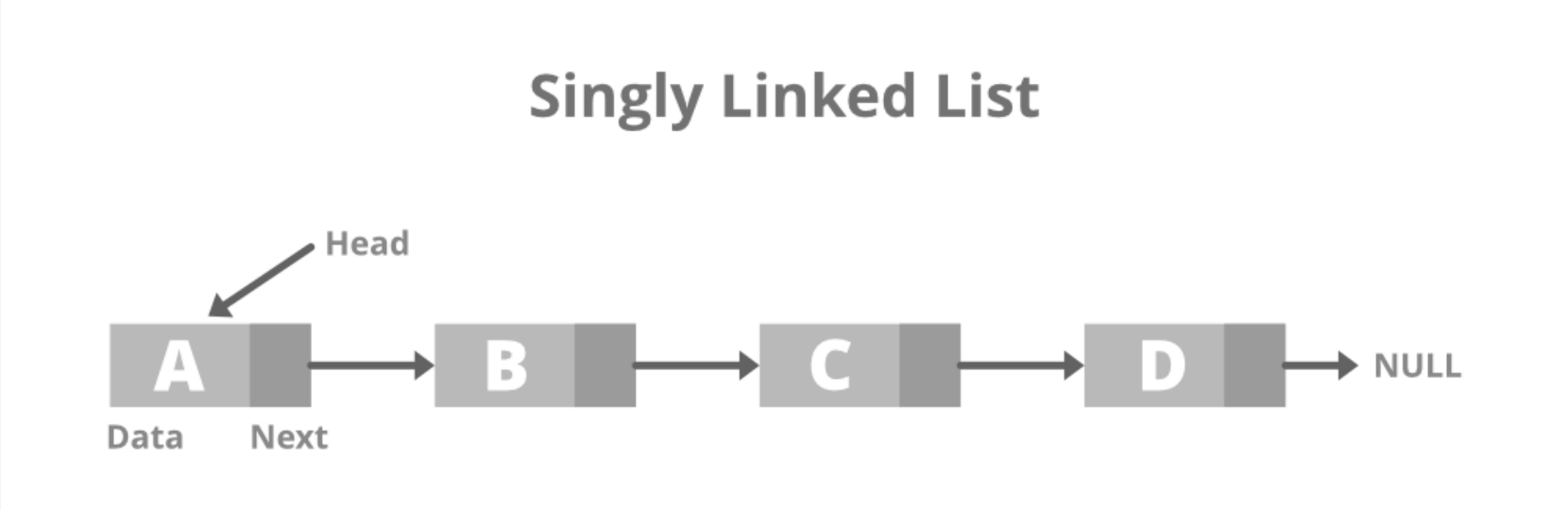
(Implementation & analysis)

Singly Linked List

Introduction

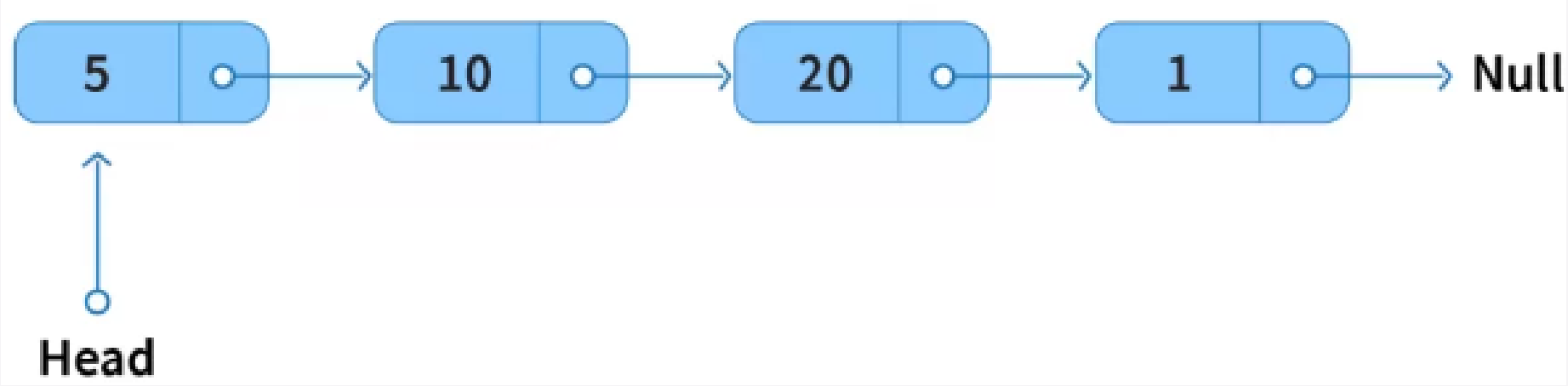
In **computer science**, a linked list is a linear collection of data elements whose **order is not given by their physical placement in memory**.

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?

A **Singly Linked List** 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.



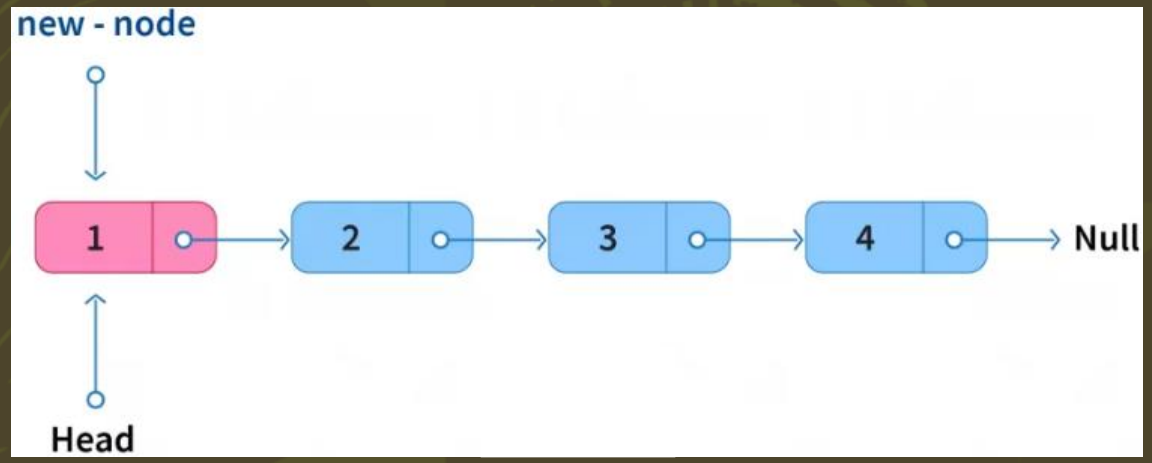
```cpp
#include <iostream>

using namespace std;

template <class T> class Node {
    private:
        T data; // The object information
        Node* next; // Pointer to the next node element

    public:
        Node(T new_data, Node* next_node){
            this->data = new_data;
            this->next = next_node;
        }

        Node(T new_data){
            this->data = new_data;
            this->next = NULL;
        }

        void set_data(T new_data){
            this->data = new_data;
        }

        T get_data(){
            return this->data;
        }

        void set_next(Node *next_node){
            this->next = next_node;
        }

        Node* get_next(){
            return this->next;
        }
};
```

- The Node of the singly linked list, apart from the data, stores the address of only the next element, as shown below.

- For a linked list we maintain a special **pointer** known as **HEAD**. This pointer stores the address of the first node of the list.

- Also, the last node can no longer have the next element. Hence, we indicate the end of the linked list by assigning **NULL** to its next.

```
39  template <class T> class CustomLinkedList{
40      Node<T> *head;
41
42      public:
43          CustomLinkedList(){
44              head = NULL;
45          }
46
47          ~CustomLinkedList(){
48          }
49
50          // Method adds data to the begining of the list
51          void add_begin(T data){
52              Node<T>* temp = new Node<T>(data, head);
53
54              this->head = temp;
55          }
```



Insert (1/3)

Insertion in a singly linked list can be performed in the following ways,

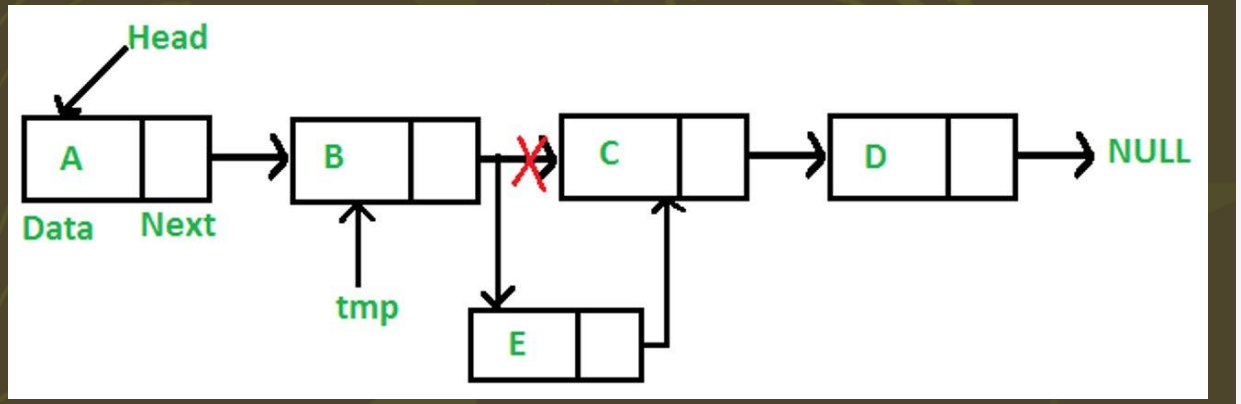
Insertion at the start Insertion of a new node at the start of a singly linked list is carried out in the following manner.

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

What is the time complexity?

Insert (2/3)

```
// Method adds data after a node in the list
void add_after_node(T value_insert, T value_target){
    Node<T>* current_node = this->head;

    // List is empty, the node wasn't found
    if(!current_node){
        return;
    }

    // Search for the node
    while(current_node && current_node->get_data() != value_target){
        current_node = current_node->get_next();
    }

    Node<T>* node_insert = new Node<T>(value_insert, current_node->get_next());

    current_node->set_next(node_insert);
}
```

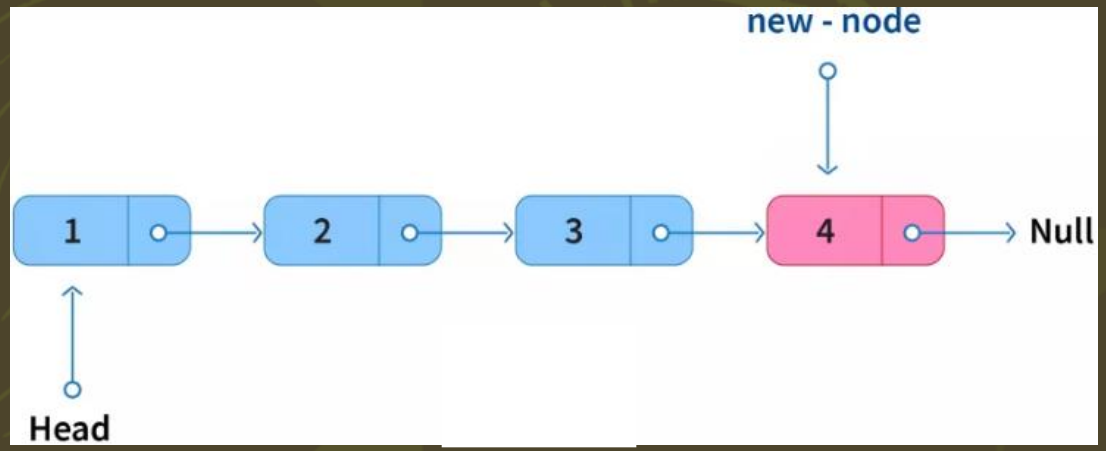


Insertion after some Node Insertion of a new node after some node in a singly linked list is carried out in the following manner,

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

What is the time complexity?

```
// Method adds data to the end of the list
void add_end(T data){
    if(head == NULL){ //if our list is currently empty
        head = new Node<T>(data); //Create new node of type T
    }
    else{ //if not empty add to the end and move the tail
        Node<T>* temp = new Node<T>(data);

        Node<T>* current_node = this->head;
        while (current_node->get_next() != NULL){
            current_node = current_node->get_next();
        }

        current_node->set_next(temp);
    }
}
```



Insert (3/3)

Insertion at the end Insertion of a new node at the end of a singly linked list isperformed in te followin way,

- 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?

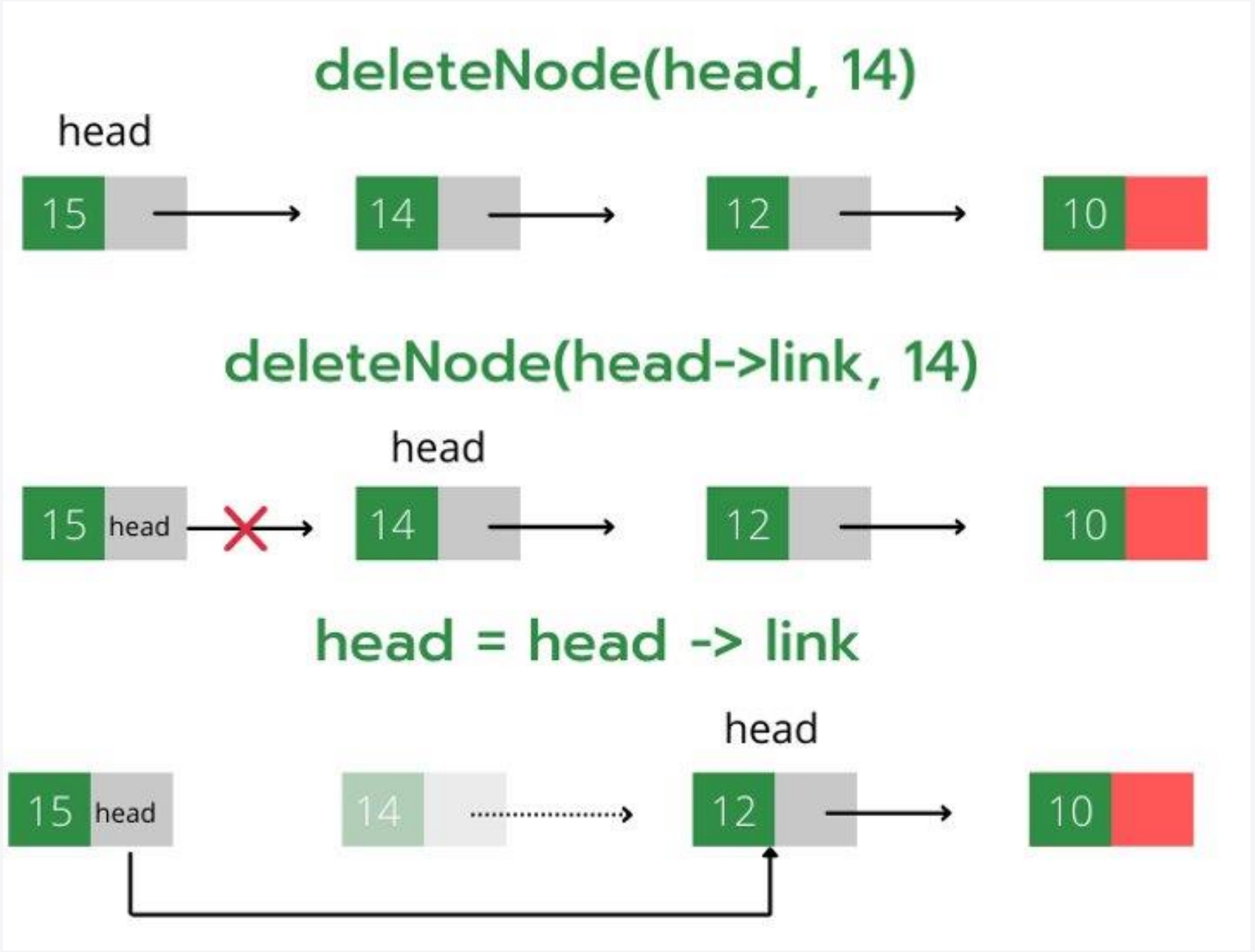
```
void delete_node(T value){
    if(!this->head){
        // Empty linked list, no values to delete
        return;
    }

    // Check if the node to delete is the head
    if(this->head->data == value){
        this->head = this->head->get_next();
        return;
    }

    Node<T>* current_node = this->head;
    // Search for the node to delete
    while ( current_node && current_node->next->get_data() != value){
        current_node = current_node->get_next();
    }

    // current node is empty the node wasn't found
    if(!current_node){
        return;
    }

    Node<T>* node_to_delete = current_node->get_next();
    Node<T>* next_node = node_to_delete->get_next();

    current_node->set_next(next_node);
    delete node_to_delete;
}
```

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.

What is the time complexity?


deleteNode(head, 14)
head
15
14
12
10
deleteNode(head->link, 14)
head
15
head
14
12
10
head = head -> link
head
15
head
14
12
10

Display

To display the entire singly linked list, we need to traverse it from first to last.

```
void print_list(){
    Node<T>* current_node = this->head;
    while (current_node != NULL){
        cout << current_node->get_data() << " -> ";

        current_node = current_node->get_next();
    }
    cout << "NULL" << endl;
}
```

In contrast to arrays, linked list nodes cannot be accessed randomly. Hence to reach the n-th element, we are bound to traverse through all (n-1) elements.

What is the time complexity?

Search

```
Node<T>* search_node(T value_target){
    Node<T>* current_node = this->head;
    // Search for the node to delete
    while ( current_node){
        if(current_node->get_data() == value_target){
            return current_node;
        }
        current_node = current_node->get_next();
    }

    return NULL;
}
```

To search an element in the singly linked list, we need to traverse the linked list right from the start.

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(){
    CustomLinkedList<int> firstList;

    firstList.add_end(32);
    //Pause the program until input is received

    firstList.add_end(33);
    firstList.add_end(34);
    firstList.print_list();

    firstList.add_after_node(10, 33);
    firstList.print_list();

    Node<int>* element = firstList.search_node(11);

    if(element)
        cout << element->get_data() << endl;
    return 0;
}
```

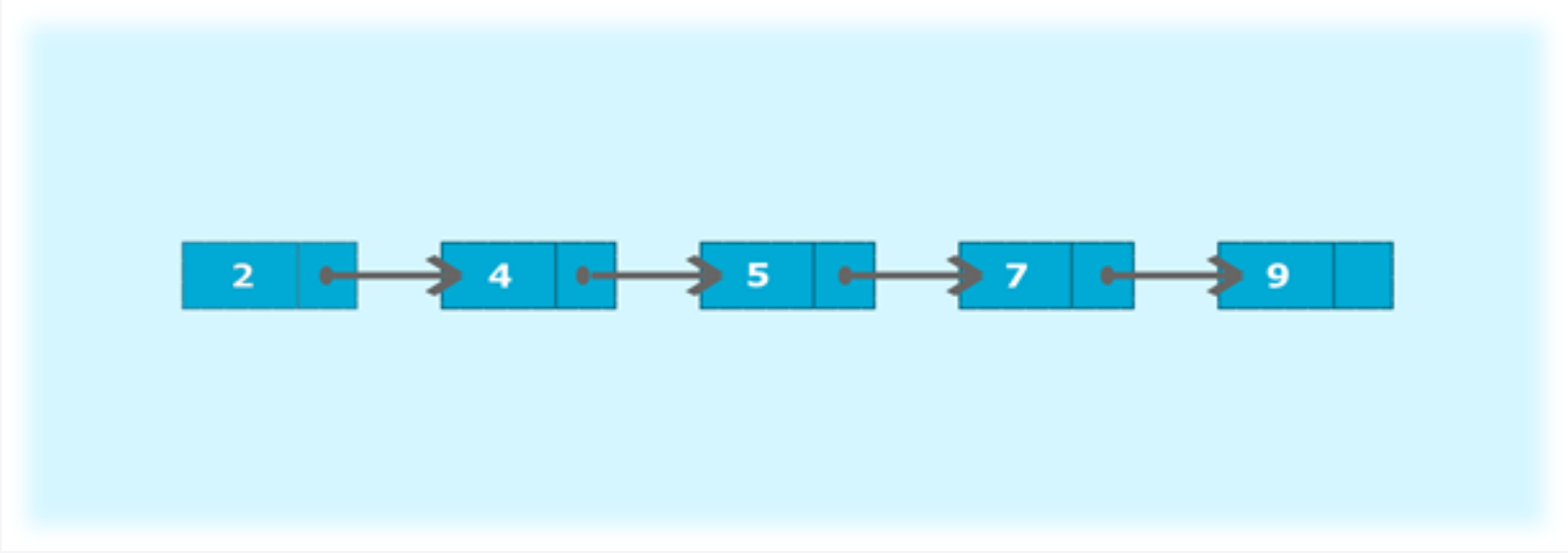
Result:

```
32 -> 33 -> 34 -> NULL
32 -> 33 -> 10 -> 34 -> NULL
```

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, we cannot perform the binary search over it.

The binary search require to access to any location in **constant time**.

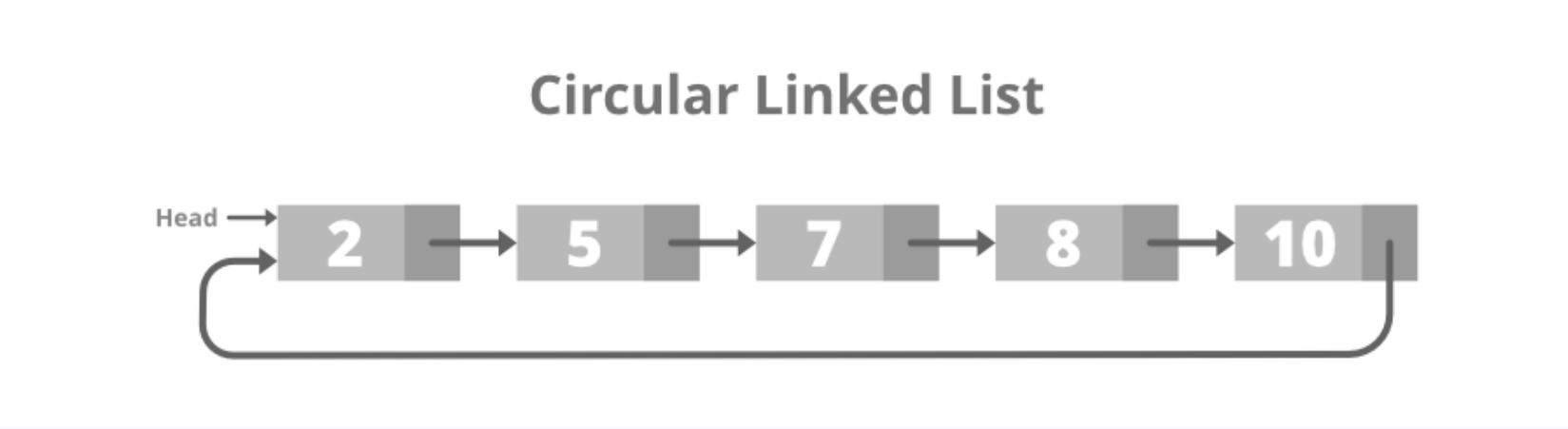
Any element of a singly linked list can only be accessed in a **sequential manner** making binary search completely ineffective.

Circular Linked List

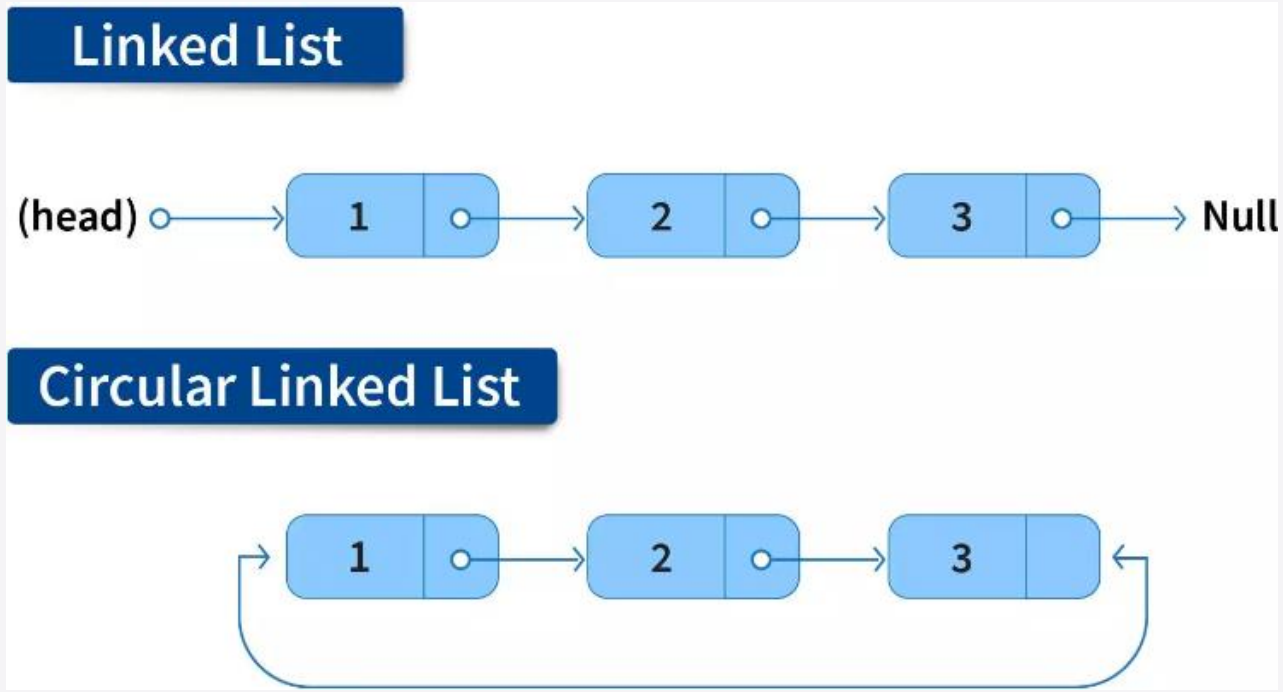
Introduction

As the name suggests, Circular Linked List is a variation of Linked List Data Structures.

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**.



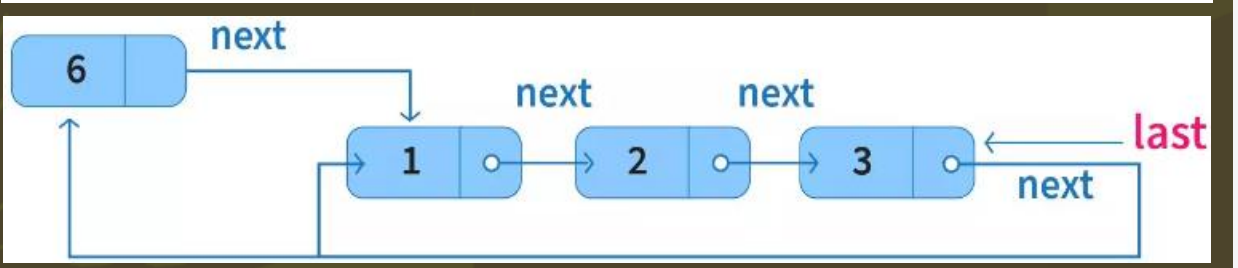
Circular linked lists avoids this overhead of traversing till the Node points to null, and then using head pointer to circle back from start, but it's the intrinsic property of Circular Linked lists which provides this property to circle the Linked List, by default.



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

struct Nodo
{
    int num;
    Nodo *sig;
};

class CircularList
{
    private:
        Nodo *Head;
        Nodo *Last;

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

        void Insert(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.
- These are some points that differentiate it from the linked list.

```
void CircularList::Insert(int key)
{
    Nodo *apNodo = new Nodo;

    if(apNodo == NULL)
        cout << "ERROR: memory could not be reserved"<<endl;

    apNodo->num = key;

    if(Head == NULL)
    {
        Head = apNodo;
        Last = apNodo;
        apNodo->sig = apNodo; // link to itself
    }
    else
    {

        apNodo->sig = Head;
        Last->sig = apNodo;
        Head = apNodo;
    }
}
```



Insert

Insertion at the Beginning:

- Get the current First Node, and the New Node can point to this node as its next. This takes care of adjusting pointers of the New Node.

- Point the Last Node to the new Node, thus honoring the circular property.

What is the time complexity?

```
void CircularList::EraseAll()
{
    if(Head == NULL)
        cout << "Empty List." << endl;
    else
    {
        Nodo *apAux;

        while(Head->sig != Head)
        {
            apAux = Head->sig;
            Head->sig = apAux->sig;
            delete apAux;
        }
        delete Head;
        Head = NULL;
        Last = NULL;
        cout<< "Deleted List."<<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 CircularList::Display()
{
    Nodo *apAux = Head;

    if(Head == NULL)
        cout << "Empty List." << endl;
    else
    {
        cout << "Elements: ";

        do
        {
            cout << apAux->num << " ";
            apAux = apAux->sig;
        }
        while(apAux != Head);

        cout << 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?

```
void CircularList::Search(int key)
{
    Nodo *apAux = Head;

    if(Head == NULL)
        cout << "Empty List." << endl;
    else
    {
        int indx = 0;
        cout << "Results: ";
        do
        {
            if(apAux->num == key)
                cout << indx << " ";

            apAux = apAux->sig;
            indx++;
        }
        while(apAux != Head);

        cout << endl  << "Search completed." << endl;
    }
}
```

Search

To search an element in the singly 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()
{
    CircularList L;

    L.Display();

    L.Insert(65);
    L.Insert(36);
    L.Insert(2);
    L.Insert(41);

    L.Display();

    L.Search(36);
    L.Search(4);

    L.Display();

    L.EraseAll();

    L.Display();



    return 0;
}
```

Result:

```
Empty List.
Elements: 41 2 36 65
Results: 2
Search completed.
Results:
Search completed.
Elements: 41 2 36 65
Deleted List.
Empty List.
Destructor: Empty List.
```

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.

Applications of Circular linked list

As we have seen, circular linked lists come in handy when we need to have the circular order maintained between the different Nodes. Unlike linked lists, no Node in circular linked lists points to null.

Circular linked lists finds its application in many real live systems where we can easily circle back to the starting Node from the last one. Some of those can be:

- In multiplayer games, each player is represented as a Node in the circular linked list. 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 for running multiple applications, 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.