

# Data Structures and Algorithms

Trong-Hop Do

University of Information Technology, HCM

# Recap: Reference variable

- A reference variable is a "reference" to an existing variable, and it is created with the & operator:

```
string food = "Pizza"; // food variable
string &meal = food;   // reference to food
```

- Now, we can use either the variable name food or the reference name meal to refer to the food variable:

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

int main() {
    string food = "Pizza";
    string &meal = food;

    cout << food << "\n";
    cout << meal << "\n";
    return 0;
}
```

```
Pizza
Pizza
```

# Recap: Memory Address

- In the example from the previous page, the **& operator** was used to create a reference variable. But it can also be used to get the memory address of a variable; which is the location of where the variable is stored on the computer.
- When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.
- To access it, use the **& operator**, and the result will represent where the variable is stored:

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

int main() {
    string food = "Pizza";

    cout << &food;
    return 0;
}
```

0x6dfed4

# Recap: Reference variable and memory address

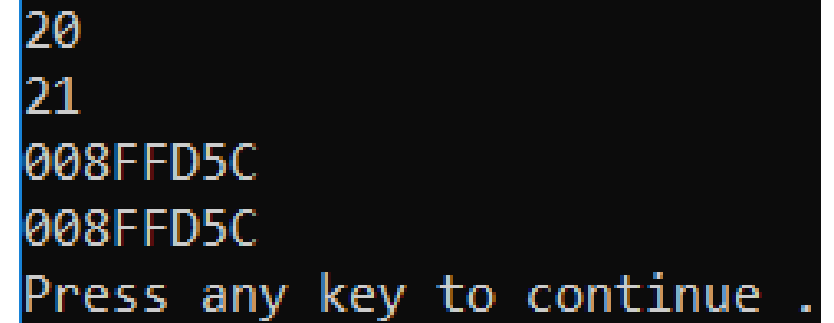
```
#include <iostream>
using namespace std;
int main()
{
    int value = 10;
    int &price = value; // reference to value

    value = 15; // value = 15
    price = 20; // value = 20

    cout << value << "\n"; // 20
    ++price;
    cout << value << "\n"; // 21

    // print address of value and ref
    cout << &value << "\n";
    cout << &price << "\n";

    system("pause");
    return 0;
}
```



20  
21  
008FFD5C  
008FFD5C  
Press any key to continue .

# Recap: Pointer

- We can get the memory address of a variable by using the & operator:

```
string food = "Pizza"; // A food variable of type string

cout << food; // Outputs the value of food (Pizza)
cout << &food; // Outputs the memory address of food (0x6dfed4)
```

- A pointer is a variable that stores the memory address as its value.

```
string food = "Pizza"; // A food variable of type string
string* ptr = &food; // A pointer variable that stores the address of food

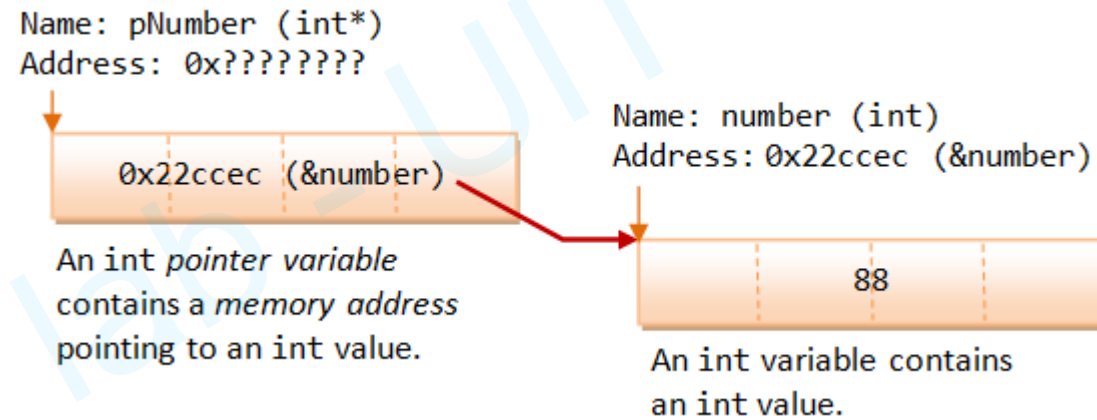
// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";
```

# Recap: Initializing Pointers via the Reference (Address-Of) Operator &

```
int number = 88;  
int * pNumber;  
pNumber = &number;  
int * pAnother = &number;
```



# Recap: Dereference

- In the example from the previous page, we used the pointer variable to get the memory address of a variable (used together with the **& reference operator**). However, you can also use the pointer to get the value of the variable, by using the **\* operator** (the **dereference operator**):

```
string food = "Pizza"; // Variable declaration
string* ptr = &food;   // Pointer declaration

// Reference: Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";

// Dereference: Output the value of food with the pointer (Pizza)
cout << *ptr << "\n";
```

- Note that the **\*** sign can be confusing here, as it does two different things in our code:
- When used in declaration (`string* ptr`), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

# Recap: Modify Pointers

```
string food = "Pizza";  
string* ptr = &food;  
  
// Output the value of food (Pizza)  
cout << food << "\n";  
  
// Output the memory address of food (0x6dfed4)  
cout << &food << "\n";  
  
// Access the memory address of food and output its value (Pizza)  
cout << *ptr << "\n";  
  
// Change the value of the pointer  
*ptr = "Hamburger";  
  
// Output the new value of the pointer (Hamburger)  
cout << *ptr << "\n";  
  
// Output the new value of the food variable (Hamburger)  
cout << food << "\n";
```



# malloc() function

- `malloc()` is a function used for allocating memory for N blocks at run time.
- Whenever a program needs memory to declare at run time we can use this function.

Output

```
Enter elements :
10
20
30
40
50
Input elements are :
10
20
30
40
50
```

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    int *p; //pointer declaration
    int i=0;

    //allocating space for 5 integers
    p = (int*) malloc(sizeof(int)*5);

    cout<<"Enter elements :\n";
    for(i=0;i<5;i++)
        cin>>p[i];

    cout<<"Input elements are :\n";
    for(i=0;i<5;i++)
        cout<<p[i]<<endl;

    free(p);
    return 0;
}
```

# New operation

- **new** is an operator used to declare memory for N blocks at run time.

Output

```
Enter elements :
10
20
30
40
50
Input elements are :
10
20
30
40
50
```

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

int main()
{
    int *p; //pointer declaration
    int i=0;

    //allocating space for 5 integers
    p = new int[5];

    cout<<"Enter elements :\n";
    for(i=0;i<5;i++)
        cin>>p[i];

    cout<<"Input elements are :\n";
    for(i=0;i<5;i++)
        cout<<p[i]<<endl;

    delete p;
    return 0;
}
```

# Difference between new and malloc()

Both are used for same purpose, but still they have some differences, the differences are:

1. `new` is an operator whereas `malloc()` is a library function.
2. `new` allocates memory and calls constructor for object initialization. But `malloc()` allocates memory and does not call constructor.
3. Return type of `new` is exact data type while `malloc()` returns `void*`.
4. `new` is faster than `malloc()` because an operator is always faster than a function.

# delete and free()

- Delete is an Operator in C++ which is used to free the memory allocated by new operator or for a NULL Pointer . It also calls the destructor of the class.

```
delete ptr;
```

- Free function is use to deallocate memory allocated by malloc function.

```
free(ptr);
```

- **Difference Between Delete and Free:**

- **delete** is an **operator** while **free** is a **function**.
- delete frees the allocated memory and also calls the destructor of the class in c++ while free() does not calls any destructor and only free the allocated memory .
- free() uses C run time heap whereas Delete may be overloaded on class basis to use private heap.

# Recap: dot & arrow Operators

- The . (dot) operator and the -> (arrow) operator are used to reference individual members of classes, structures, and unions.
- Difference between Dot(.) and Arrow(->) operator:
  - The Dot(.) operator is used to normally access members of a structure or union.
  - The Arrow(->) operator exists to access the members of the structure or the unions using pointers.

```
struct Employee {  
    char first_name[16];  
    int age;  
} emp;
```

```
struct student* emp = NULL;
```

```
// Assigning memory to struct variable emp  
emp = (struct student*) malloc(sizeof(struct student));
```

```
// Assigning value to age variable of emp  
emp->age = 18;
```

# Static variable vs dynamic variable

A dynamic variable is a variable whose address is determined when the program is run. In contrast, a static variable has memory reserved for it at compilation time.

```
int x;  
x = 5 ;
```

**Static variable x**

5

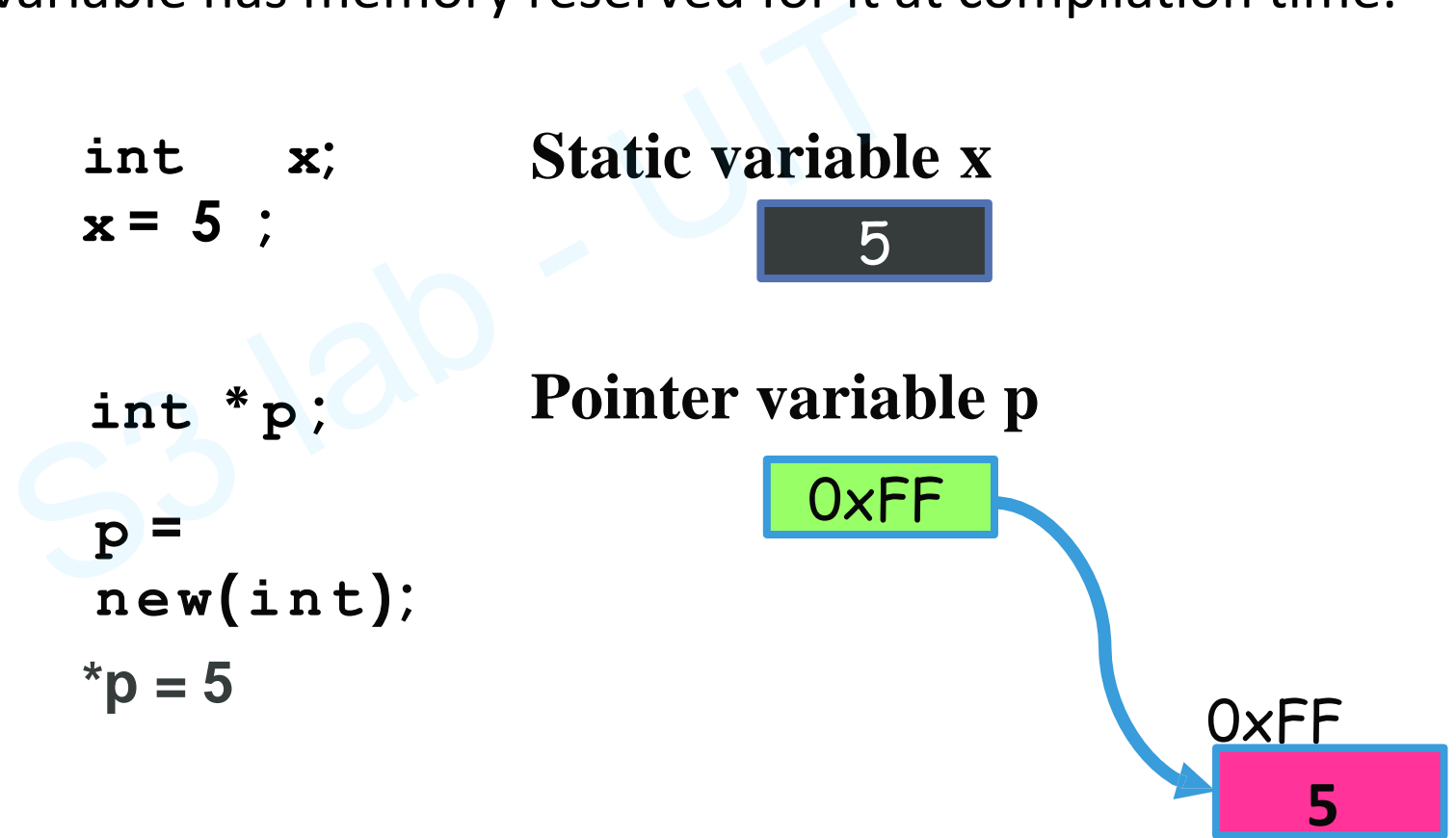
```
int *p;  
p =  
new(int);  
*p = 5
```

**Pointer variable p**

0xFF

0xFF

5



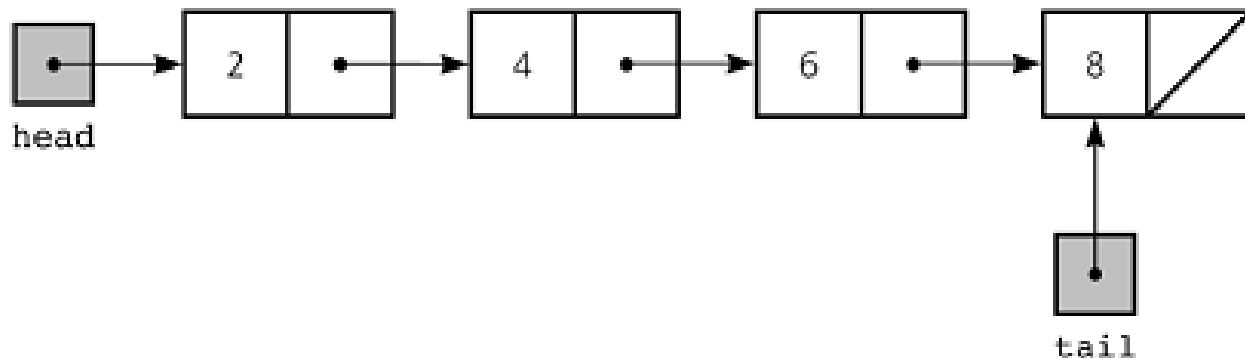
# Static list

```
typedef struct {  char
    Name[20];  int ID;
} STUDENT;  STUDENT
List[50];
```

- Number of student  $< 50 \Rightarrow$  waste of memory
- Number of student  $> 50 \Rightarrow$  not enough space !

# Linked list

- A linked list is a linear data structure where each element is a separate object.
- Linked list elements are not stored at contiguous location; the elements are linked using pointers.
- Each **node** of a list is made up of two items - the **data** and a **link** (reference) to the next node.
- The last node has a link (reference) to null.
- The entry point into a linked list is called the head (or head pointer) of the list.
- It should be noted that head is not a separate node, but pointer that stores the address of the first node.
- If the list is empty then the head is a null reference.





# Linked list terminology

## **linked list**

a list consisting of items in which each item knows the location of the next item.

## **node**

an item in a linked list. Each node contains a piece of list **data** and the **link** to the next node (item).

## **link**

(of a node) the location of the next node.

## **head node**

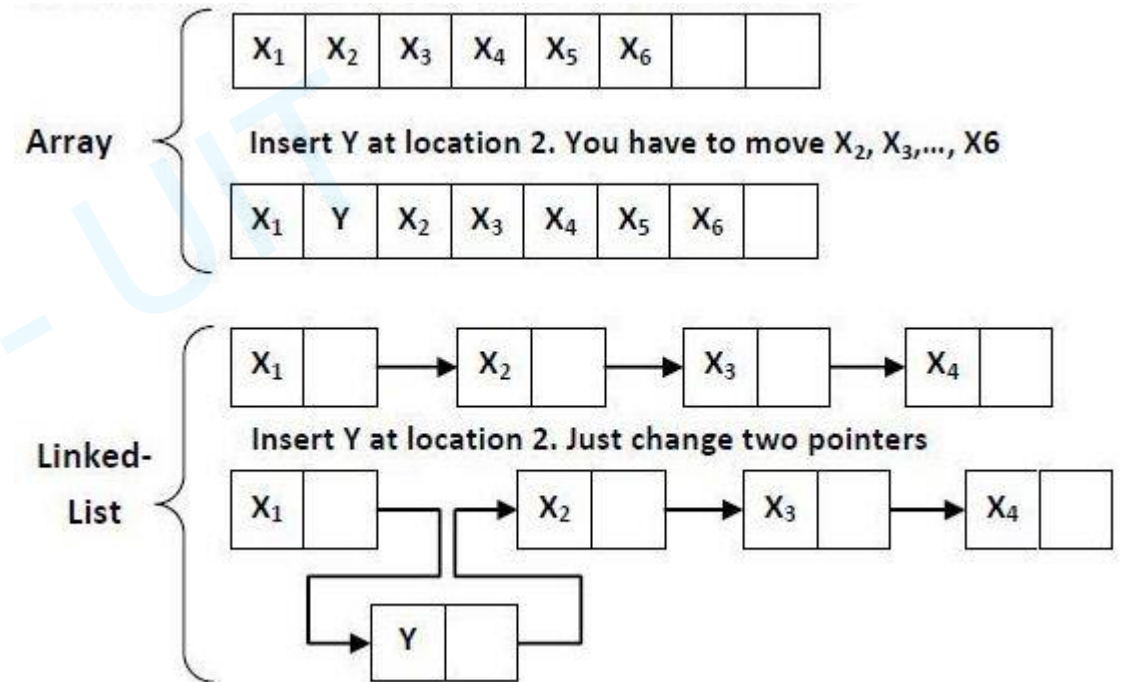
first node in a linked list

## **head pointer**

points to the head node

# Linked list vs Array

| Array   | Linked list  |
|---|--|
| Random access i.e. efficient indexing                               | No random access (no index)                                |
| Sequential access is faster (continuous memory location)            | Sequential access is slow (not continuous memory location) |
| Memory waste (may happen)   | No memory waste  |
| Fixed size: resizing is expensive                                   | Dynamic size   |
| Insertion and deletion are inefficient (need shifting all elements) | Insertion and deletion are efficient (no shifting)         |



# Linked List implementation

- Each node in a list consists of at least two parts:
  1. Data
  2. Pointer to the next node
- In C/C++, we can represent a node of Linked List using structures or classes.

```
// A linked list node
typedef struct tagNode {
    int data;
    struct Node* next; //A pointer that stores the address of the next node
}Node;

// Simple linked list data structure
typedef struct tagList {
    Node *pHead; //A pointer that stores the address of the first node
    Node *pTail; //A pointer that stores the address of the last node
}List;
```

# Recap: struct vs typedef struct

- Basically struct is used to define a structure. But when we want to use it we **have to use the struct keyword** in C.
- If we use the **typedef** keyword, then a new name, we can use the struct by that name, without writing the struct keyword.
- In C++, there is no difference between 'struct' and 'typedef struct' because, in C++, all struct/union/enum/class declarations act like they are implicitly typedef'ed, as long as the name is not hidden by another declaration with the same name.
- Though there is one subtle difference that typedefs cannot be **forward declared**. So for the typedef option, you must include the file containing the typedef before it is used anywhere.

```
struct Node {  
    int data;  
    struct Node* next;  
};  
int main()  
{  
    struct Node* head = NULL;  
    struct Node* second = NULL;  
    struct Node* third = NULL;  
}
```

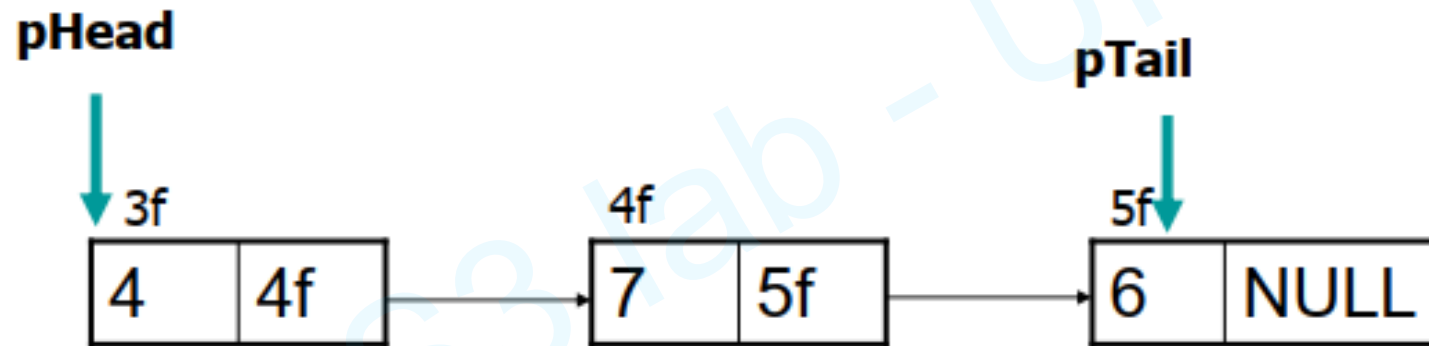
```
typedef struct Node {  
    int data;  
    struct Node* next;  
}Node;  
int main()  
{  
    Node* head = NULL;  
    Node* second = NULL;  
    Node* third = NULL;  
}
```

# Recap: struct forward declarations

- A forward declaration is a declaration of an identifier (denoting an entity such as a type, a variable, a constant, or a function) for which the programmer has not yet given a complete definition.
- A forward declaration allows us to tell the compiler about the existence of an identifier before actually defining the identifier. In the case of struct, this allows us to tell the compiler about the existence of a struct before we define the struct's body.
- When you typedef an anonymous struct then the compiler won't allow you to use its name before the typedef.

```
struct a {  
    struct b * b_pointer;  
    int c;  
};  
struct b {  
    struct a * a_pointer;  
    void * d;  
};
```

# Example of simple linked list



# Basic operations

Following are the basic operations supported by a list.

- **CreateList** – Create empty linked list
- **CreateNode** – Create a node with data x
- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

# Create an empty linked list

The Head and Tail are NULL pointers.

```
void CreateList(List &l)
{
    l.pHead=NULL;
    l.pTail=NULL;
}
```



# Create new Node with data x

This function returns the address of the newly created node

```
Node*CreateNode (Data x)
{
    Node *p;
    p = new Node; //Allocate memory for the new node
    if ( p==NULL)  exit(1);
    p ->Info = x; //Assign data to the node
    p->pNext = NULL;  return p;
}
```

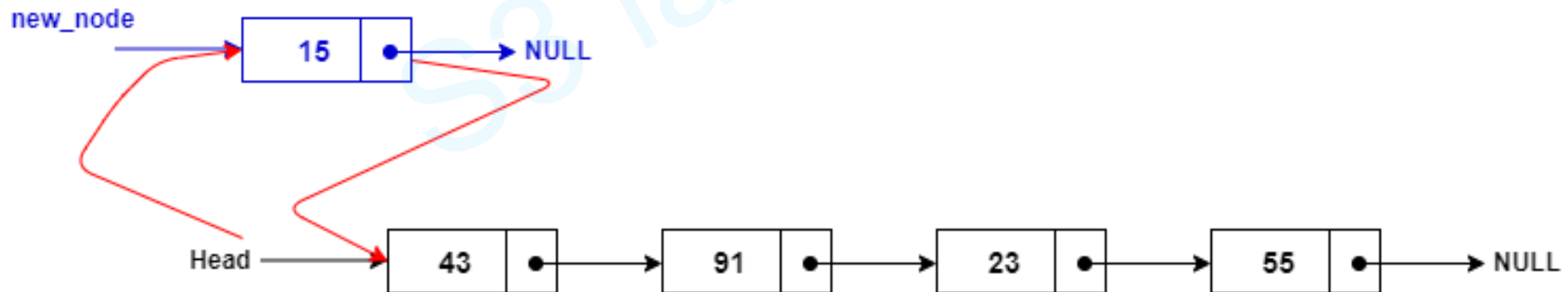
# Insert an Node to the list

Four possible positions for inserting a new element to the list:

- At the front of the list
- At the end of the list
- Before a specified node
- After a specified node

# Insert New Node at the Front of the Linked List

- Head of a linked list always points to the first node if there is at least one element in the list.
- So inserting a new node means the head will point to the newly inserted node.
- And the new node will point where head was pointing to before insertion.
- If the linked list is empty, both the head and the tail will point to the new node



# Insert New Node at the Front of the Linked List

If List empty

- `pHead = p;`
- `pTail = pHead;`

Else

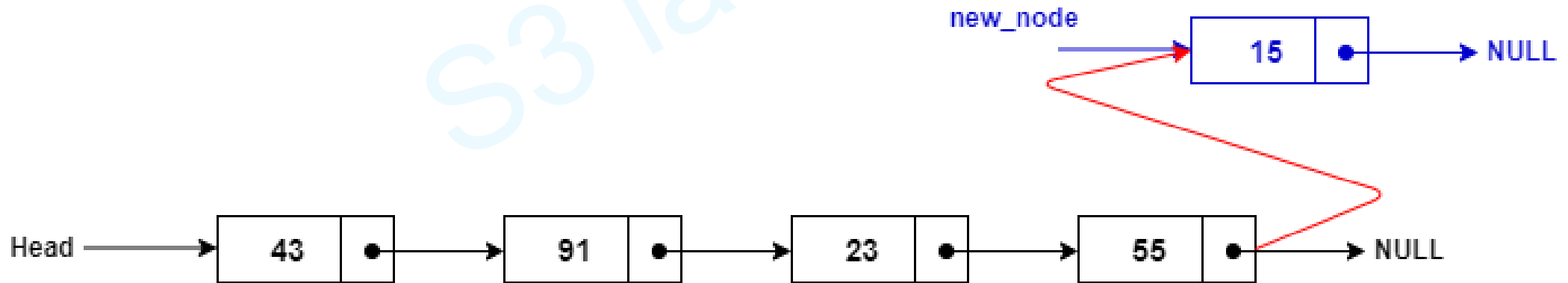
- `p->pNext = pHead;`
- `pHead = p`

# Insert New Node at the Front of the Linked List

```
void AddHead(LIST &l, Node* p)
{
    if (l.pHead==NULL)
    {
        l.pHead = p;  l.pTail = l.pHead;
    }
    else
    {
        p->pNext = l.pHead;  l.pHead = p;
    }
}
```

# Insert New Node at the End of the Linked List

- Last node of a linked points to NULL. To insert a new element at the end of the list, you have to point the current last node to the new node.
- The new node will point to NULL. So the newly inserted node becomes the last node.
- If the list is empty, both the head and the tail will point to the new node



# Insert New Node at the End of the Linked List

If List empty

- `pHead = p;`
- `pTail = pHead;`

Else

- `pTail->pNext=p;`
- `pTail=p`

# Insert New Node at the End of the Linked List

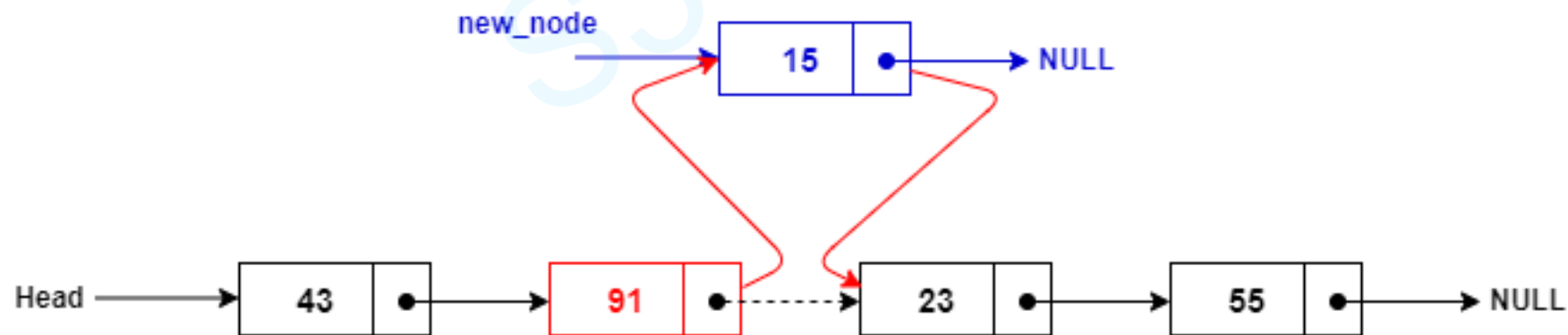
```
void AddTail (LIST &l, Node *p)
{
    if (l.pHead==NULL)
    {
        l.pHead = p;  l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = p;  l.pTail = p;
    }
}
```



# Insert New Node after a Specified Node

To insert an element after a specific node, you have to do the following.

- Point the new node where the specified node is currently pointing to.
- Point the specified node's pointer (next) to the new node.
- If that specified node is empty, than insert at the Front of the List



# Insert New Node after a Specified Node

If (q!=NULL)

p->pNext = q->pNext

q->pNext = p

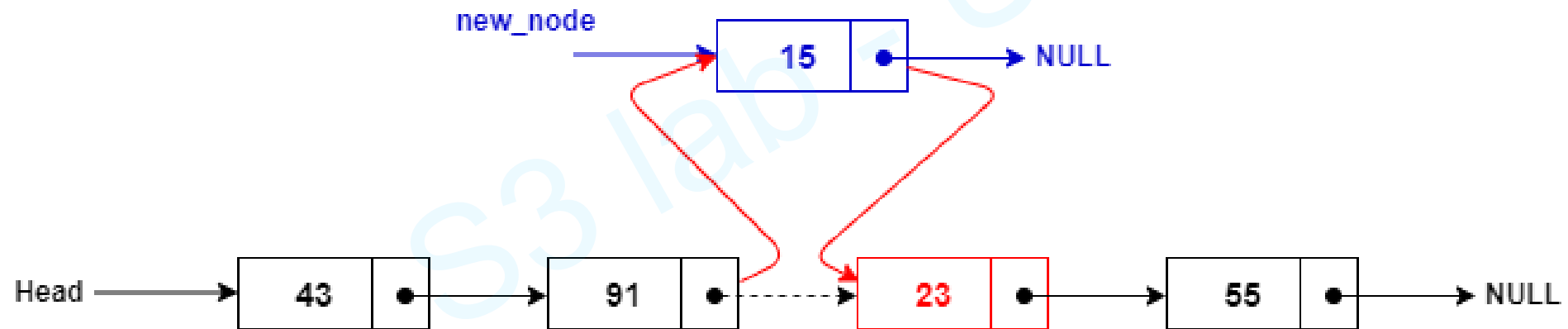
If q = pTail

pTail=p

# Insert New Node after a Specified Node

```
void InsertAfterQ(List &l, Node *p, Node *q)
{
    if (q!=NULL)
    {
        p->pNext=q->Next;
        q->pNext=p;
        if (l.pTail==q)
            l.Tail=q;
    }
    else
        AddHead(l,q); // insert q at the Front of the List
}
```

# Insert New Node before a Specified Node



# Delete a Node in Linked List

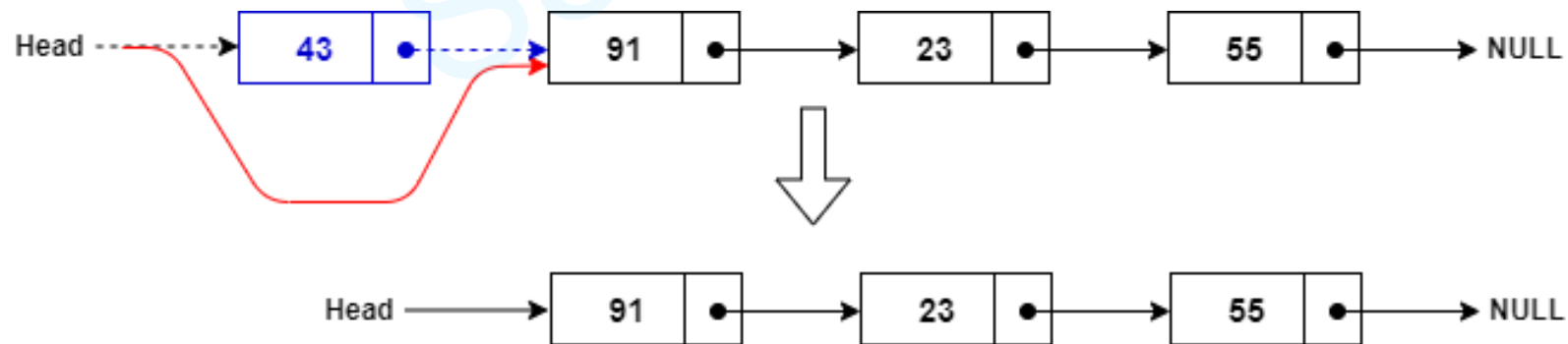
Positions for deleting a Node in Linked List

- First Node
- Last Node
- After a specified Node
- A Node with specified data x

Remember to free the memory allocated for the Node.

# Delete the First Node of a Linked List

- Head always points to the first node. Head will move to where the first node is currently pointing to.
- First node can either point to the second node or to NULL in case the first node is the only node in the linked list.
- After moving the head, we can free up the memory of the first node.



# Delete the First Node of a Linked List

Start:

If (pHead!=NULL) then

Step 1: p=pHead

Step 2: pHead = pHead->pNext

delete (p)

Step 3:

If pHead==NULL then pTail=NULL

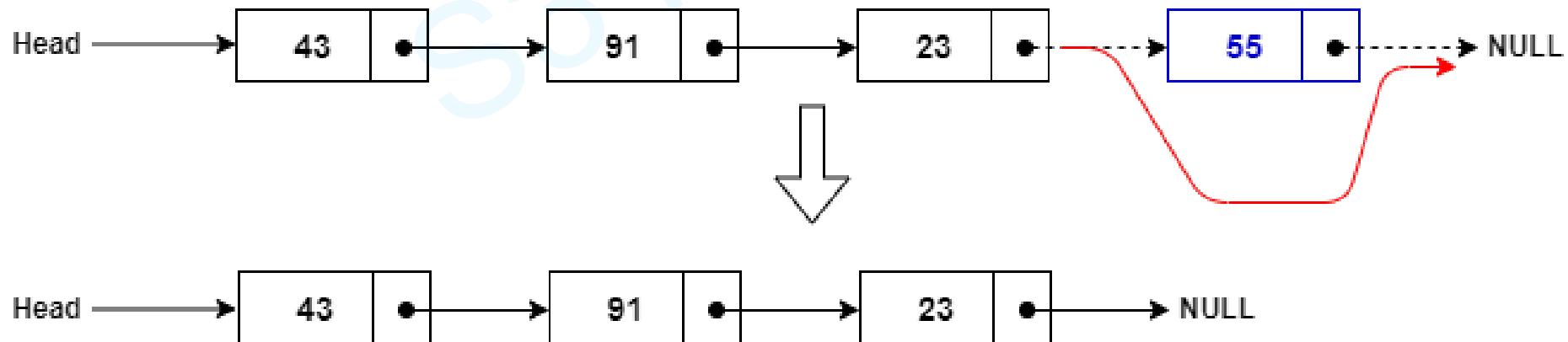
# Delete the First Node of a Linked List

```
int RemoveHead(List &l, int &x)
{
    Node *p;
    if(l.pHead!=NULL)
    {
        p=l.pHead;
        x=p->Info; //save Data of the node before delete
        l.pHead=l.pHead->pNext; delete p; if(l.pHead==NULL)
        l.pTail=NULL;
        return 1; //sucefffully deleted
    }
    return 0; //failed to delete
}
```



# Delete the Last Node of a Linked List

- The last node always points to NULL. To delete that, we have to traverse up to last node.
- We have to point the second last node to NULL and free memory of the last node.
- After deletion the second last node will become the new last node.



# Delete the Last Node of a Linked List

- 1.If head points to *NULL*, return. The linked list is empty – there is no node.
- 2.If the first node points to *NULL* – there is only one node in the linked list, delete the first node and assign *NULL* to *head*.
- 3.Point *prev* to first node and *cur* to second node.
- 4.Keep moving *prev* and *cur* (*prev* = *prev->next*, *cur* = *cur->next*) until *cur->next* is *NULL*.
- 5.If *cur->next* is *NULL*, set *prev->next* equals to *NULL* and delete *cur*.

# Delete the Last Node of a Linked List

```
void delete_last_node(struct node **head) {
    struct node *prev = NULL, *cur = NULL;
    /*Linked list does not exist or the list is empty*/
    if(head == NULL || *head == NULL) return;
    /*If there is only one node in the list*/
    if((*head)->next == NULL) {
        free(*head);
        *head = NULL;
    }
    prev = *head;
    cur = prev->next;
    while(cur->next) {
        prev = prev->next;
        cur = cur->next;
    }
    prev->next = NULL;
    free(cur);
}
```

# Delete N-th Node of a Linked List

N starts with 0. If N is 0, then delete the head node, if 1 then the second node and so on.

- 1.If *head* points to *NULL*, return. The linked list is empty.
- 2.If *N* is 0, delete the first node, point *head* to *NULL*.
- 3.Move to (*N-1*)-th node. Variable *tmp* points to the (*N-1*)-th node.
- 4.If *tmp* points to *NULL*, return. Enough nodes are not available.
- 5.De-link the *N-th* node. Point *tmp->next* to *tmp->next->next*.
- 6.Delete the N-the node.

```

/*Delete the N-th node of a linked list.*/
void delete_nth_node(struct node **head, int n) {
    struct node *tmp = NULL;
    struct node *del_node = NULL;
    /*Linked list does not exist or the list is empty*/
    if(head == NULL || *head == NULL) return;
    /*Storing the head to a temporary variable*/
    tmp = *head;
    /*Special case: we have to delete the first node.*/
    if (n == 0) {
        *head = (*head)->next;
        free(tmp);
        return;
    }
    /*Go to the (n-1)-th node.*/
    while(--n > 0 && tmp->next) tmp = tmp->next;
    /*List does not have enough nodes.*/
    if(tmp->next == NULL) return;
    /*Node to be deleted.*/
    del_node = tmp->next;
    /*De-link the n-th node*/
    tmp->next = tmp->next->next;
    free(del_node);
}

```

```

struct node{
    int val;
    struct node *next;
};

```

<https://qnaplus.com/delete-n-th-node-of-a-linked-list/>

# Delete N-th Node from End of a Linked List

$N$  starts with 0. Deleting 0-th node from end means deleting the last node.

- 1.If head points to *NULL*, then return. The list is empty.
- 2.Point 2 pointers, *prev* and *end* to head.
- 3.Move *end* pointer to  $(N+1)$ -th position.
- 4.Move *prev* and *end* pointers until *end* becomes *NULL*.
- 5.If *end* becomes *NULL* before  $(N+1)$ -th iteration, return. Enough nodes do not exist.
- 6.Delete the next node of *prev*. and point *prev* where the deleted node was pointing.

# Delete a Node after a specified Node (q)

Start

If **(q!=NULL)** thì //q exist in List

□ B1: **p=q->pNext;** // p is the Node to delete

□ B2: If **(p!=NULL)** thì // q is not the Last Node

+ **q->pNext=p->pNext;** // remove p from the List

+ If **(p== pTail)** // if p is the Last Node

**pTail=q;**

+ **delete** p;

# Delete a Node after a specified Node (q)

```
int RemoveAfterQ(List &l, Node *q, int &x)
{
    Node *p;
    if (q != NULL)
    {
        p = q->pNext; // p is the node to delete
        if (p != NULL) // q is not the Last Node
        {
            if (p == l.pTail) // p is the Last Node
                l.pTail = q; // update pTail
            q->pNext = p->pNext;
            x = p->Info;
            delete p;
        }
        return 1;
    }
    else
        return 0;
}
```



# Delete a Node with specified data x

```
int RemoveX(List &l, int x)
{
    Node *p,*q = NULL; p=l.Head;
    while((p!=NULL) && (p->Info!=x)) //searching
    {
        q=p;
        p=p->Next;
    }
    if(p==NULL) //cannot find Node with data x
        return 0;
    if(q!=NULL) //if the predecessor node exist
        DeleteAfterQ(l,q,x);
    else //p is the First Node
        RemoveHead(l,x);
    return 1;
}
```

# Find a Node with specified data x

Step 1: `p=pHead;`

Step 2:

While `p!=NULL and p->Info!=x`

`p=p->pNext;` // examine the next node

Step 3:

If `p!=NULL` //a Node with data x found

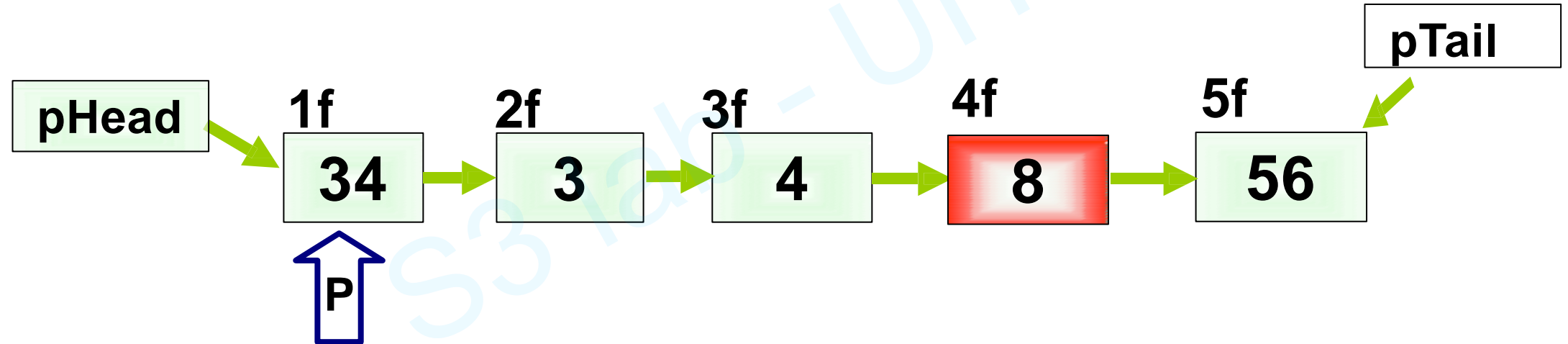
Else: //no Node with data x found

# Find a Node with specified data x

```
//Search for a Node with specified data x
//return the adress of that Node if found, otherwise return
NULL
Node *Search (LIST l, Data x)
{
Node *p;
p = l.pHead;
while ( (p != NULL) && (p->Info != x) )
p = p->pNext;
return p;
}
```

# Find a Node with specified data x

Find Node with data = 8



# Visiting all Nodes in linked list

Traversing the linked list is required when:

- Count the number of Nodes
- Find all Node with specified conditions
- Delete the entire list

# Visiting all Nodes in linked list

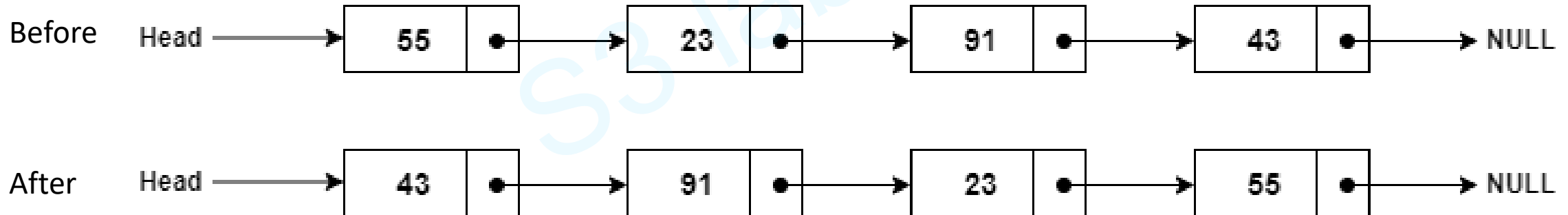
```
void PrintList(List l)
{
    Node *p;
    p=l.pHead;
    while(p!=NULL)
    {
        printf("%d ", p->Info);
        p=p->pNext;
    }
}
```

# Delete the entire linked list

```
void RemoveList(List &l)
{
    Node *p;
    while(l.pHead!=NULL) //còn phần tử trong List
    {
        p = l.pHead;
        l.pHead = p->pNext;
        delete p;
    }
}
```

# Reverse a Linked List

Reversing a linked list means re-arranging the *next* (connection) pointers. *Head* will point to the last node and the first node will point to NULL. Direction of other *next* pointers will change.





# Reverse a Linked List

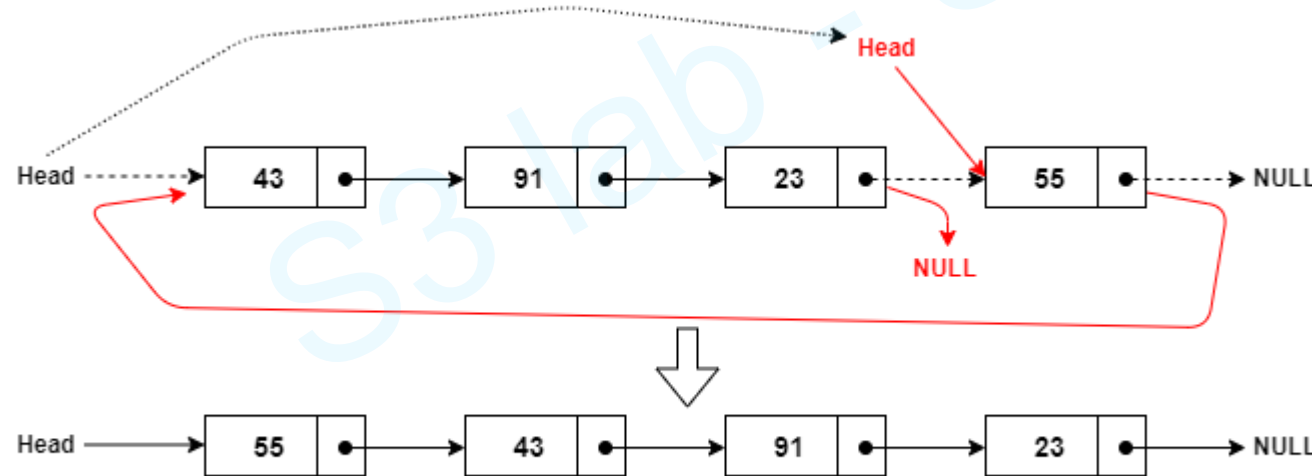
1. Take the first node out of the linked list. One new pointer will point to the first node and the head will move to the next node.
2. Insert the node at the front of the new list.
3. Repeat previous two steps until all nodes move to the new list.
4. Point the head of the original linked list to the first node of the new list.

# Reverse a Linked List

```
void reverse_linked_list(struct node **head) {  
    struct node *new_head = NULL; /*head of the reversed list*/  
    struct node *tmp = NULL;  
  
    while(*head) {  
        tmp = *head; /*tmp points the first node of the current list*/  
        *head = (*head)->next;  
  
        /*Insert tmp at the front of the reversed list.*/  
        tmp->next = new_head;  
        new_head = tmp;  
    }  
  
    *head = new_head;  
}
```

# Move the Last Node to the Front of a Linked List

We can move the last node to the front of a linked list just by rearranging the pointers. No need to create or delete any node.



# Move the Last Node to the Front of a Linked List

- 1.If the linked list is empty or has only one element, then return. If the list has only one element, then the first node and the last node are basically same.
- 2.Point *prev* to the first node and *cur* to the second node.
- 3.Keep moving both *prev* and *cur* to the next nodes until *cur->next* becomes *NULL*. *cur->next* pointing to *NULL* means *cur* pointing to the last node and *prev* pointing to the second last node.
- 4.Point the second last node to *NULL* (*prev->next = NULL*), last node to the first node (*cur->next = head*). And assign *head* where *cur* is pointing to (*head = cur*).

# Homework

Implement other operations on linked list.

S3 lab - UIT