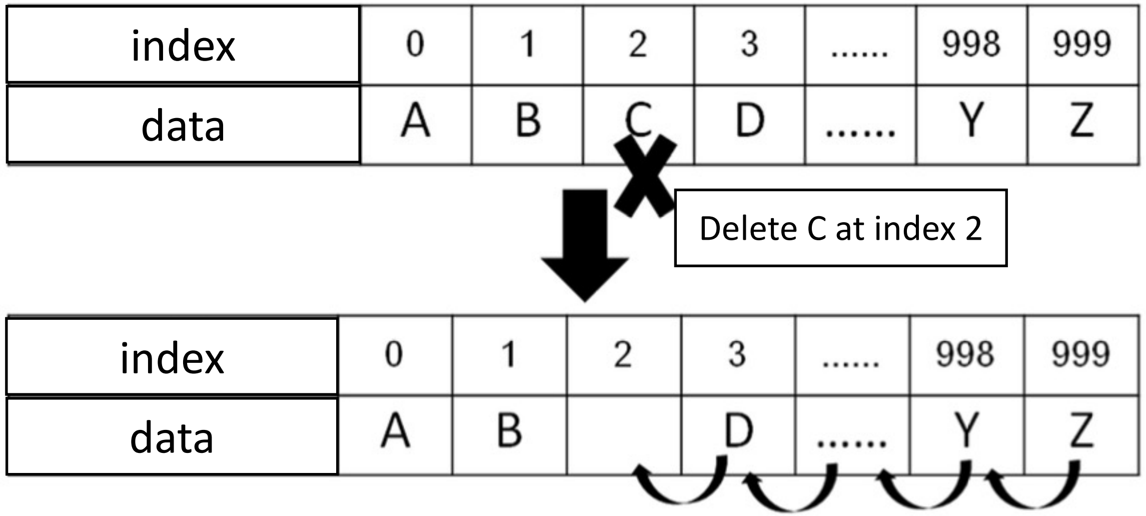
Linked List

Outline

- Linked List
- Stack: Linked List Implementation
- Queue: Linked List Implementation
- Doubly Linked List

Why?

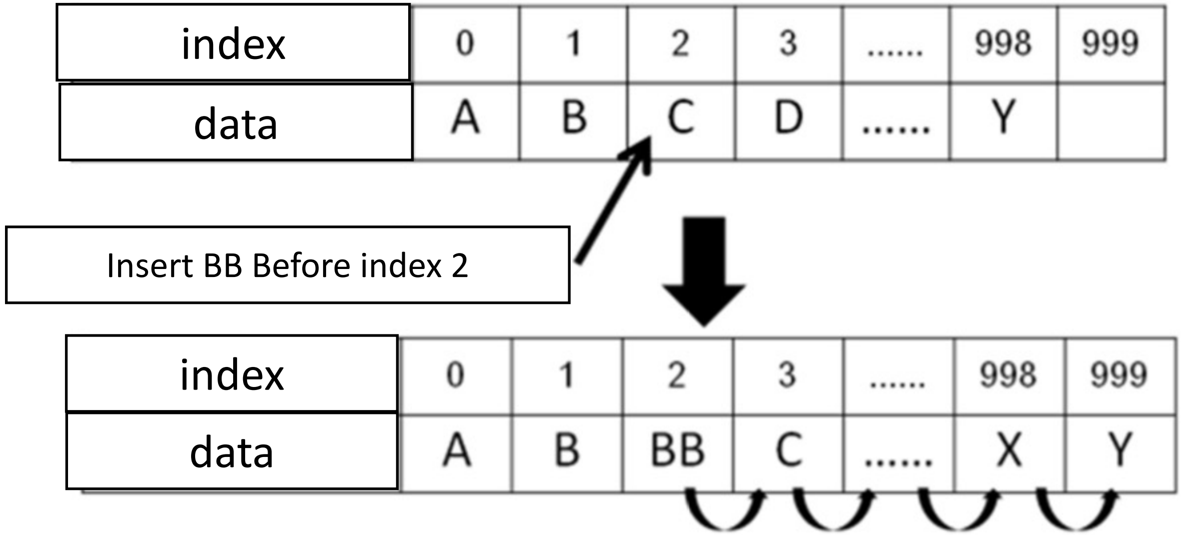


index	0	1	2	3	......	998	999
data	A	B	C	D	......	Y	Z

Delete C at index 2

index	0	1	2	3	......	998	999
data	A	B		D	......	Y	Z

Why?



index	0	1	2	3	......	998	999
data	A	B	C	D	......	Y	

Insert BB Before index 2

index	0	1	2	3	......	998	999
data	A	B	BB	C	......	X	Y

Linked List

A linked list is a data structure where each object is stored in a “*node*”

As well as storing data, the node must also contains a “reference/pointer” to the node containing the next item of data

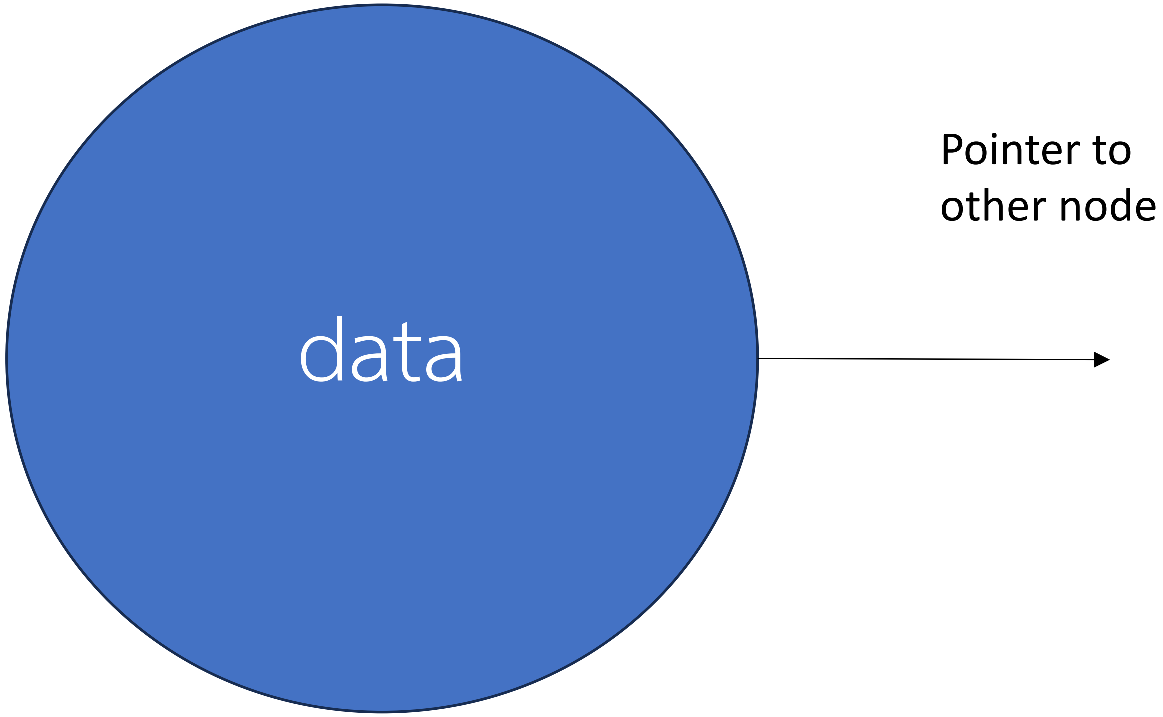
Linked List

We must dynamically create the nodes in a linked list

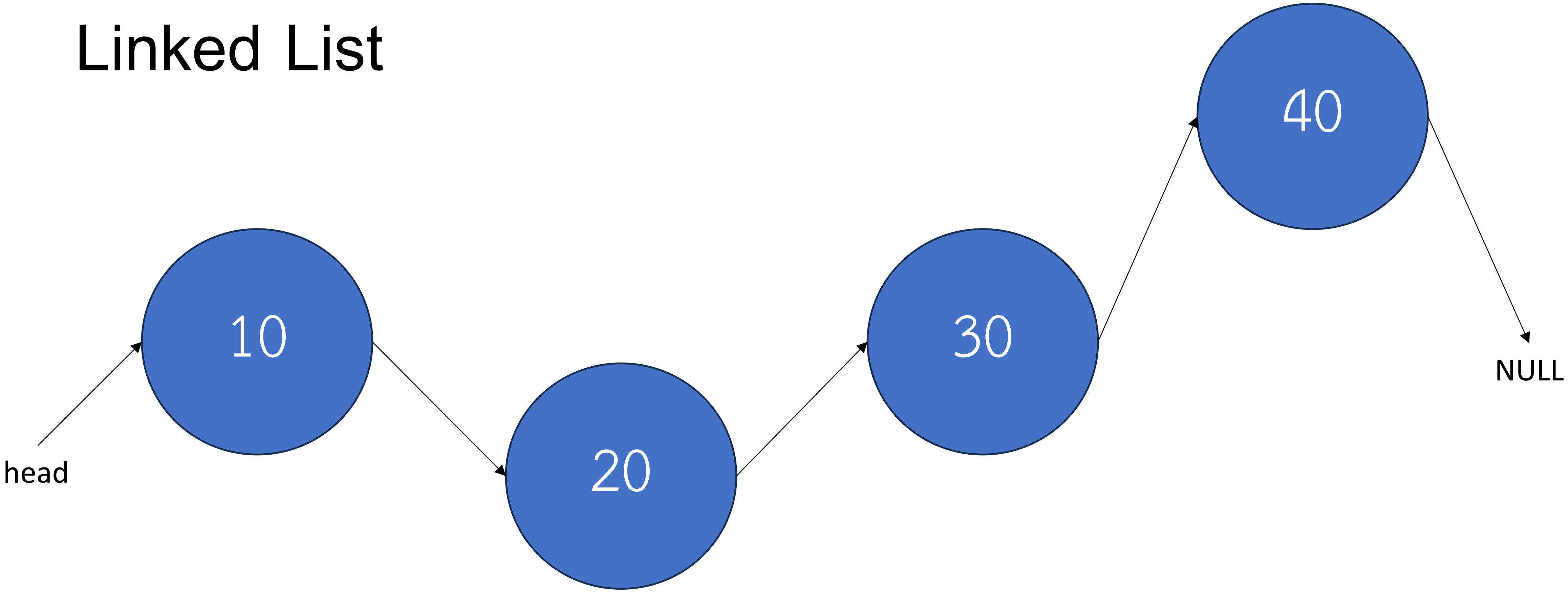
Thus, because new returns a pointer, the logical manner in which to track a linked lists is through a pointer

A Node class must store the data and a reference to the next node (also a pointer)

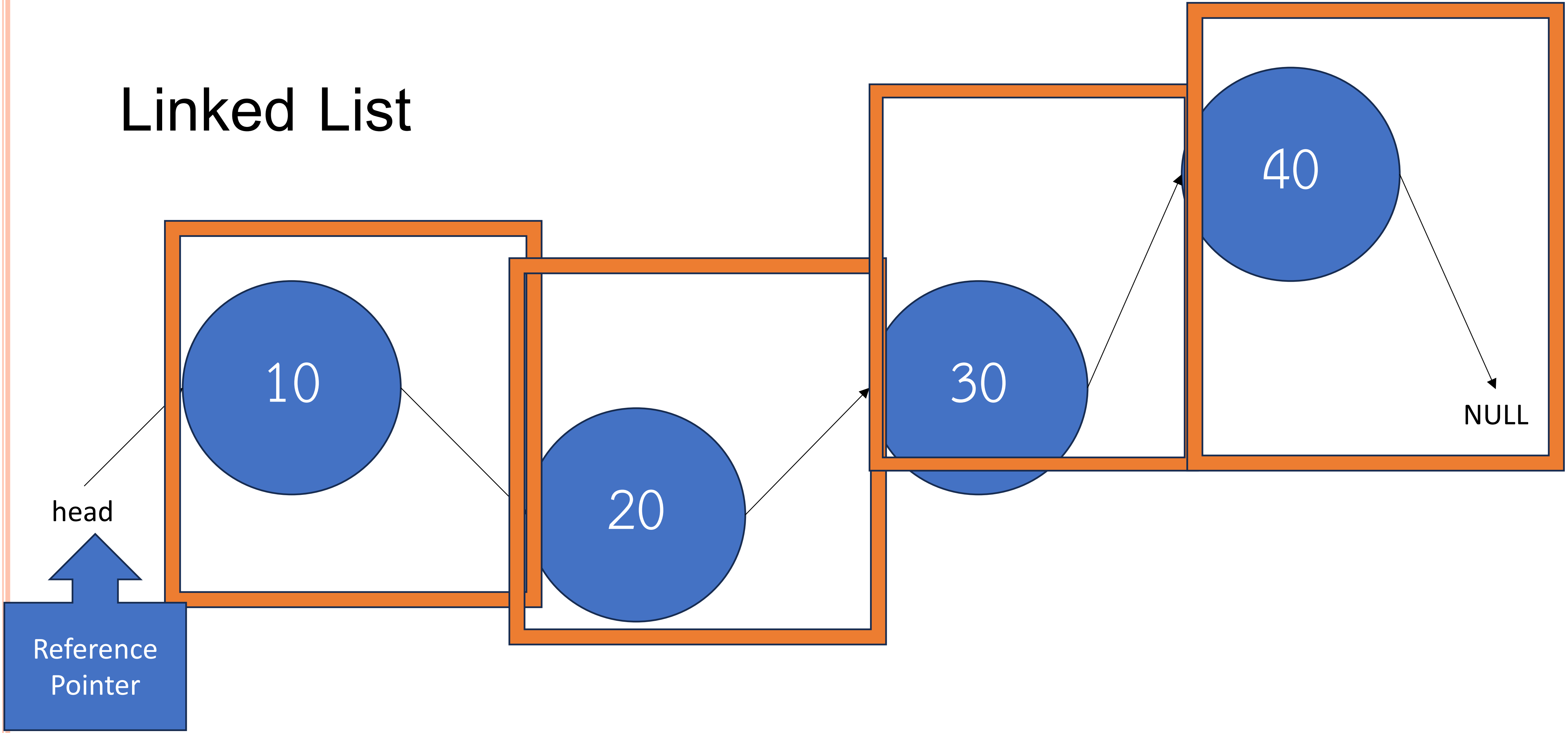
Node



Linked List



Linked List



Node

The node must store data and a pointer:

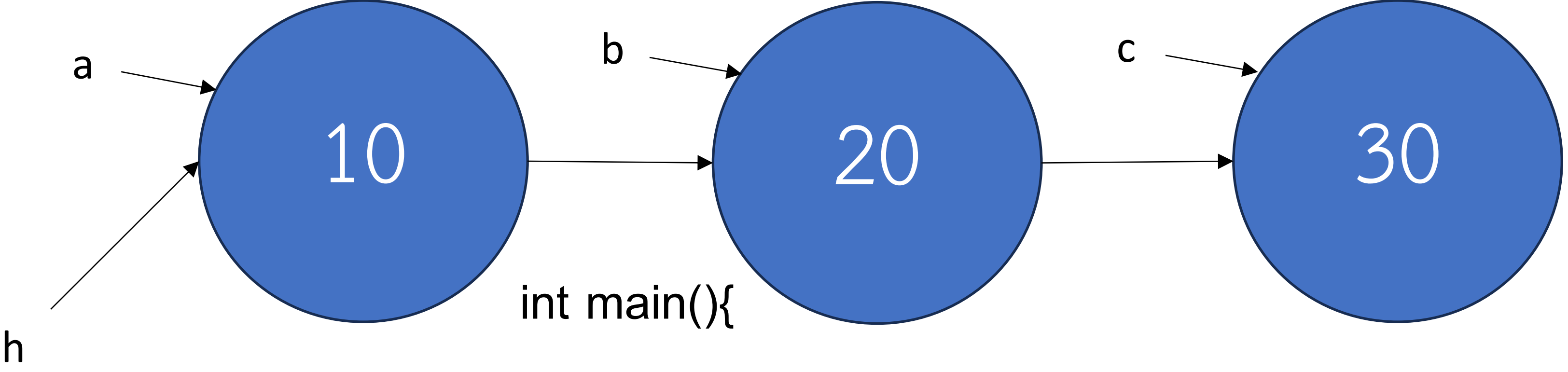
```
class Node {
        public:
        int value;
        Node *next;
};
```

Node: Constructor

The constructor assigns the value variables

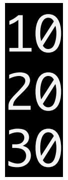
```
Node(int v){
   value = v;
   next = NULL;
}
```

Example



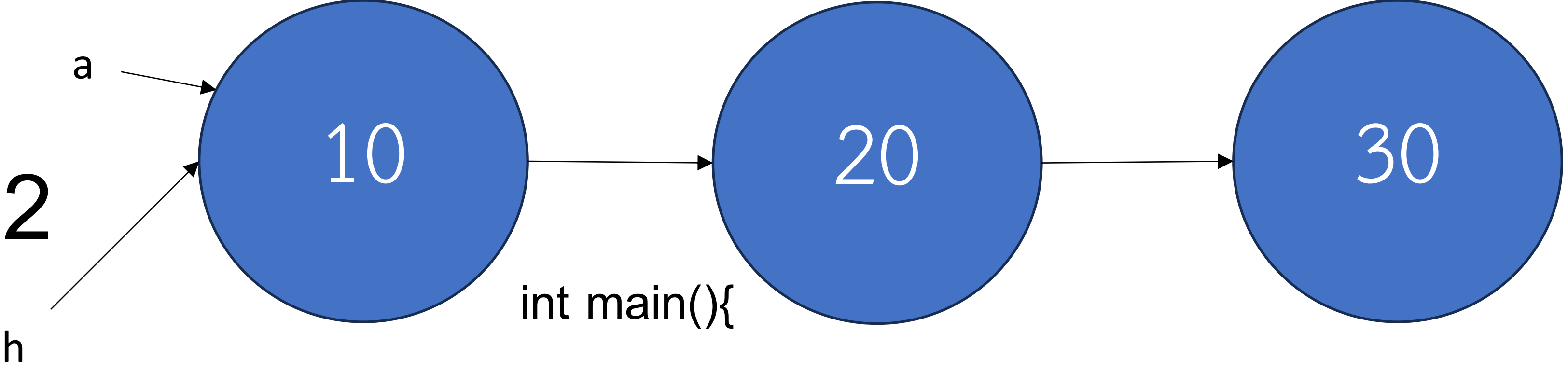
```
class Node {
  public:
  int value;
  Node *next;
  Node(int v){
        value = v;
        next = NULL;
  }
};
```

```
int main(){
  Node *a = new Node(10);
  Node *b = new Node(20);
  Node *c = new Node(30);
  a->next = b;
  b->next = c;
  for(Node *h = a; h!=NULL;h=h->next){
        cout<<h->value<<" ”;
  }
  return 0;
}
```



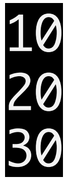
```
10
20
30
```

Example 2



```
class Node {
   public:
   int value;
   Node *next;
   Node(int v){
            value = v;
            next = NULL;
   }
};
```

```
int main(){
   Node *a = new Node(10);
   a->next = new Node(20);
   a->next->next = new Node(30);
   for(Node *h = a; h!=NULL;h=h->next){
            cout<<h->value<<" ”;
   }
   return 0;
}
```



```
10
20
30
```

Linked List Class

Because each node in a linked lists refers to the next, the linked list class need only link to the first node in the list

The linked list class requires member variable: a pointer to a node

```
class Node {...}
class LinkedList {
    public:
            Node *list_head;
    // ...
};
```

Linked List Class

To begin, let us look at the internal representation of a linked list

Suppose we want a linked list to store the values

42 95 70 81

in this order

Structure

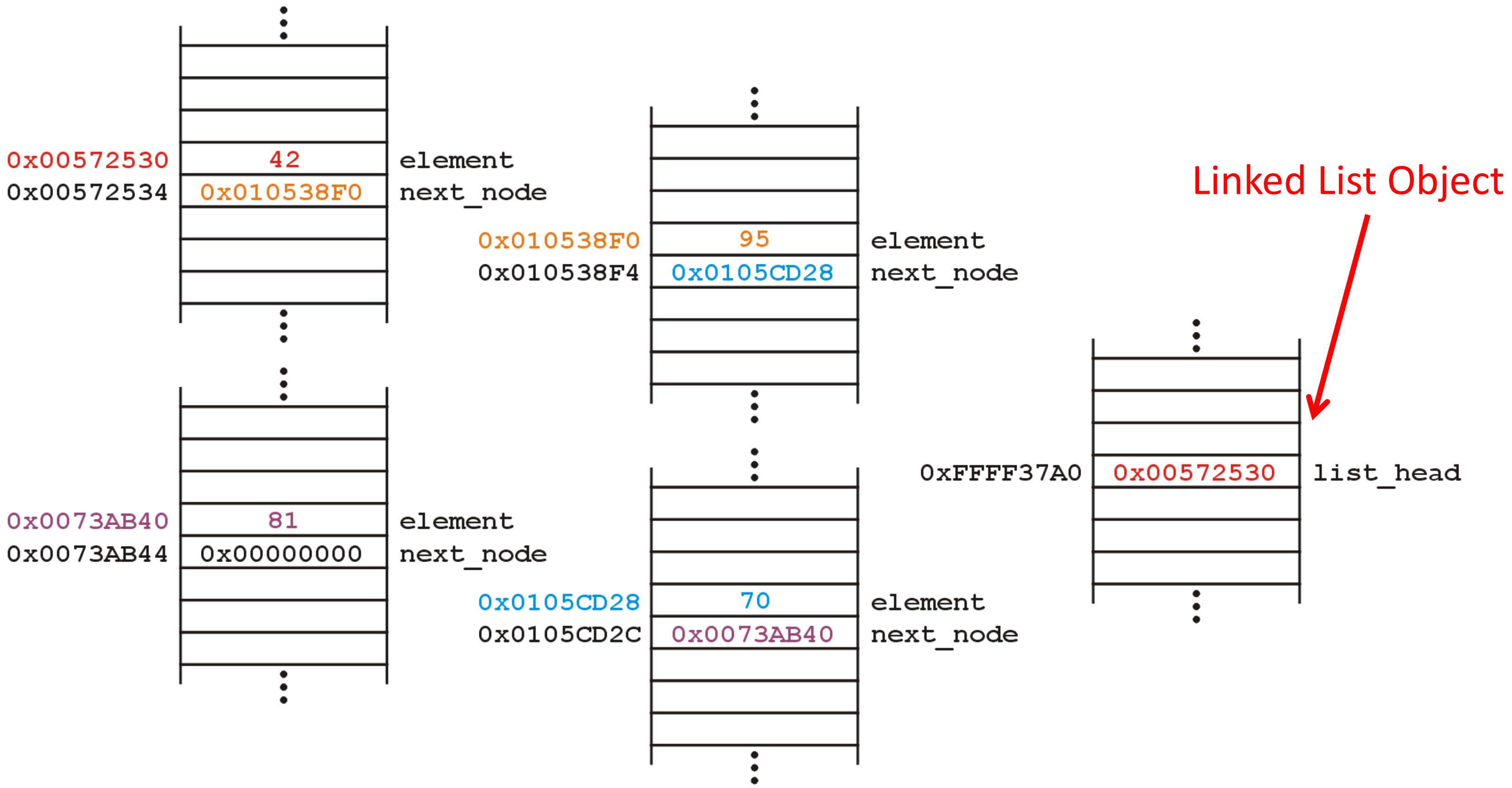
A linked list uses linked allocation, and therefore each node may appear anywhere in memory

Also the memory required for each node equals the memory required by the member variables

- 4 bytes for the linked list (a pointer)
- 8 bytes for each node (an **int** and a pointer)
    - We are assuming a 32-bit machine

Structure

Such a list could occupy memory as follows:



0x00572530 42 element
0x00572534 0x010538F0 next_node

0x010538F0 95 element
0x010538F4 0x0105CD28 next_node

0x0073AB40 81 element
0x0073AB44 0x00000000 next_node

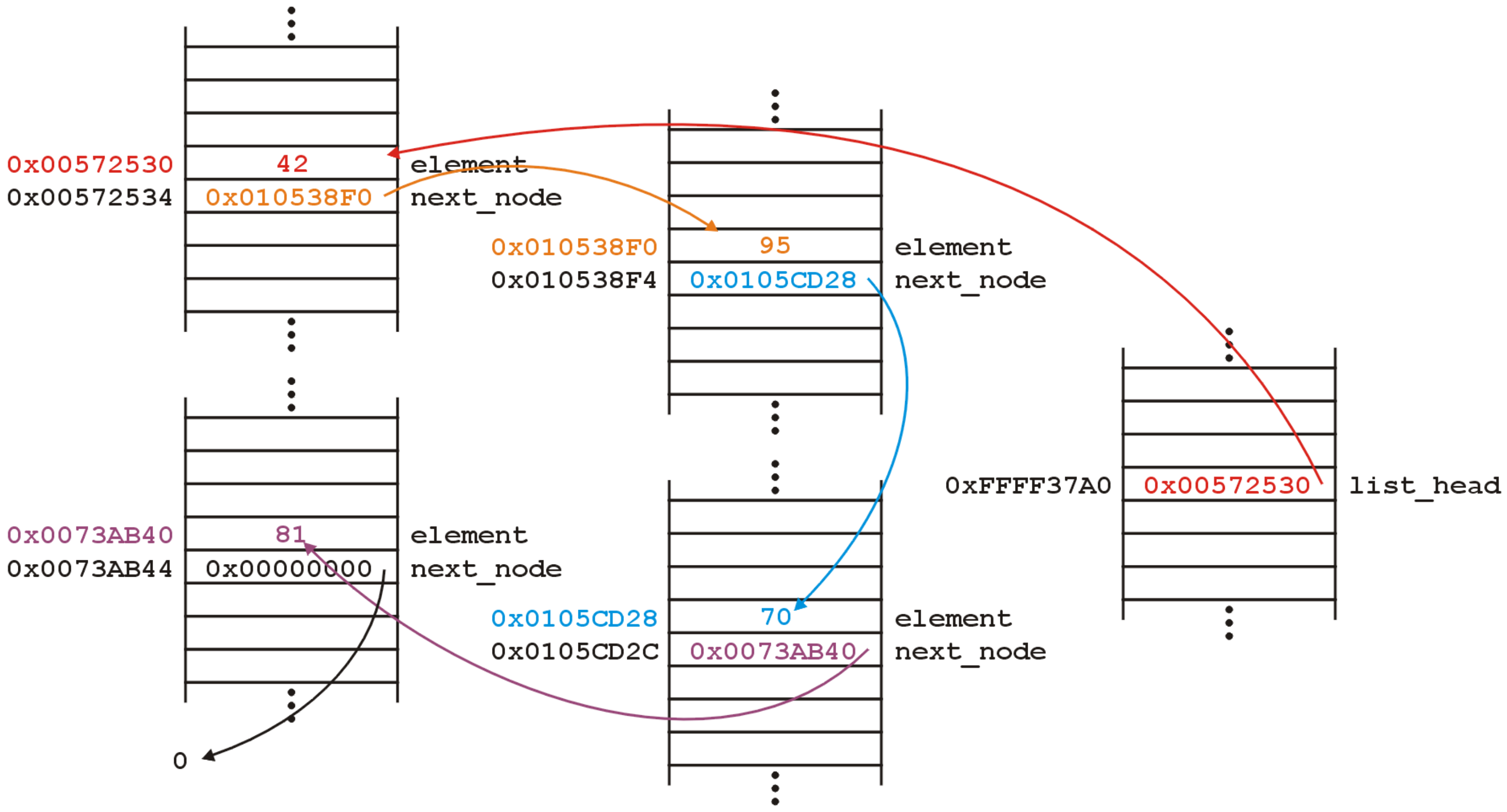
0x0105CD28 70 element
0x0105CD2C 0x0073AB40 next_node

Linked List Object

0xFFFF37A0 0x00572530 list_head

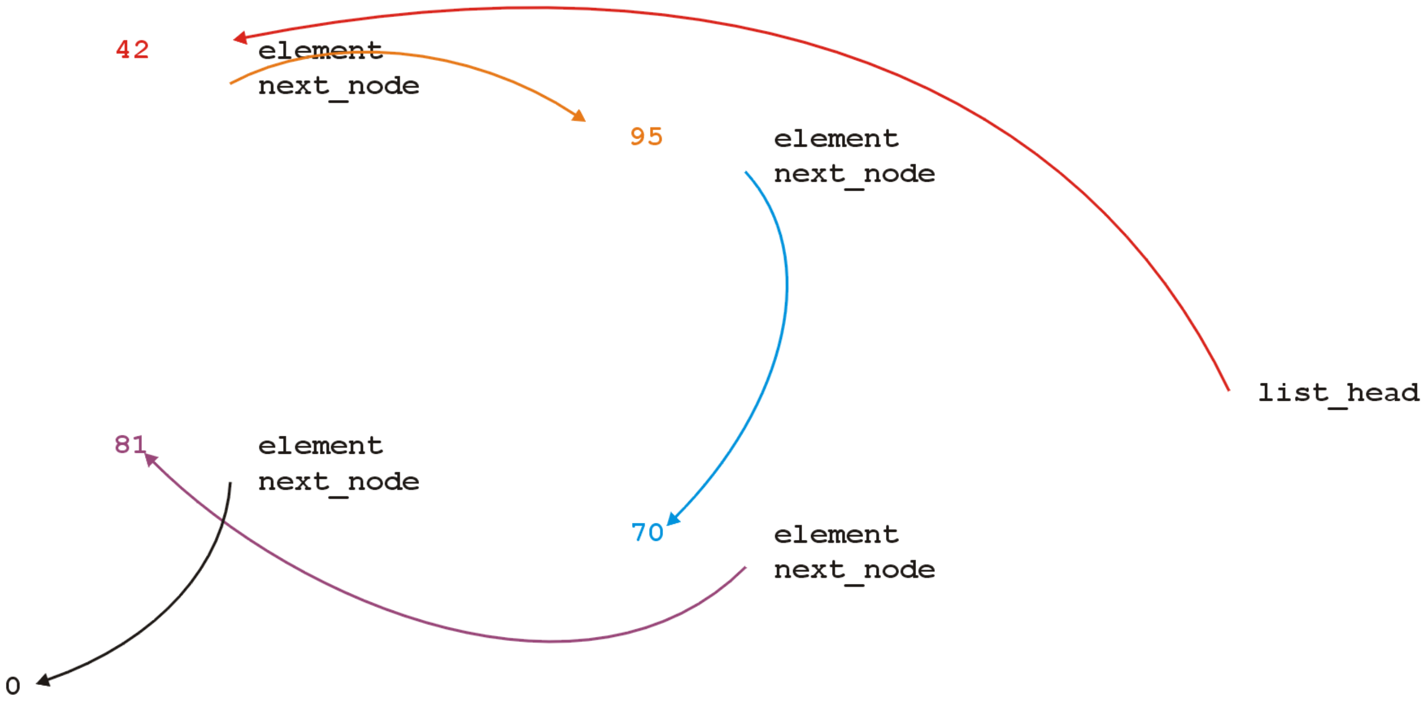
Structure

The **next_node** pointers store the addresses of the next node in the list



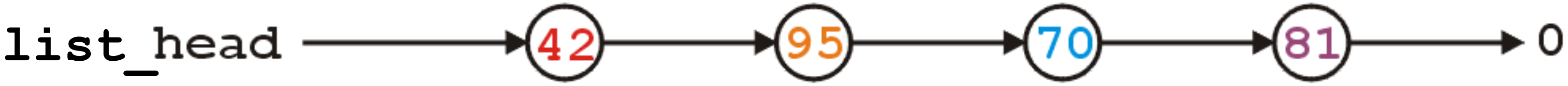
Structure

Because the addresses are arbitrary, we can remove that information:



Structure

We will clean up the representation as follows:



We do not specify the addresses because they are arbitrary and:

- The contents of the circle is the value
- The next_node pointer is represented by an arrow

Operations

First, we want to create a linked list

We also want to be able to:

- Insert before/after
- Append
- Access
- Delete

the values stored in the linked list

Operations

We can do them with the following operations:

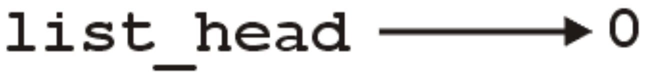
- Adding, retrieving, or removing the value at the front of the linked list

```
void push_front( int );

void pop_front();

...
```

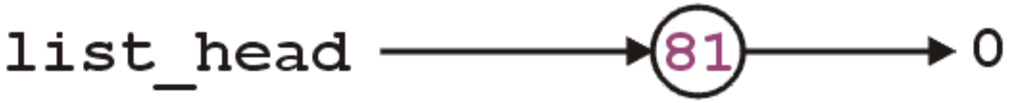
void push_front(int)

Next, let us add a value to the list

If it is empty, we start with:



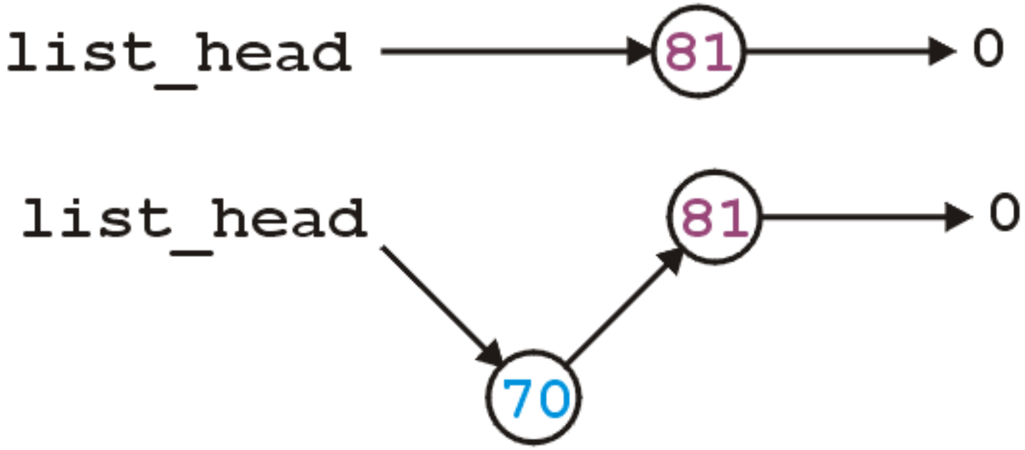
and, if we try to add 81, we should end up with:



void push_front(int)

Suppose however, we already have a non-empty list

Adding 70, we want:



void push_front(int)

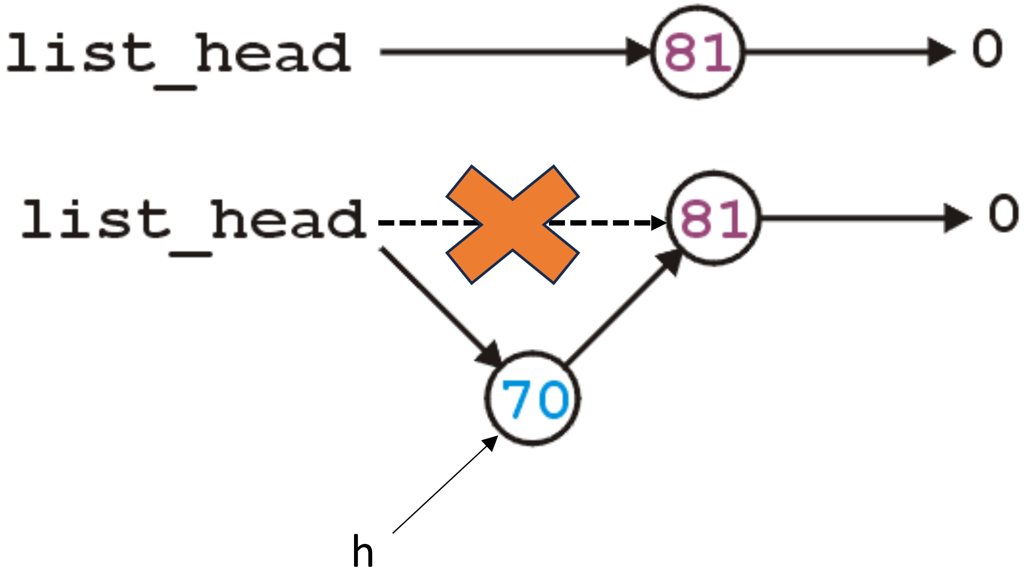
To achieve this, we must we must create a new node which:

- stores the value 70, and
- is pointing to the current list head

void push_front(int value)

Thus, our implementation could be:

```
Node *h = new Node(value);

h->next = list_head;

list_head = h;
```

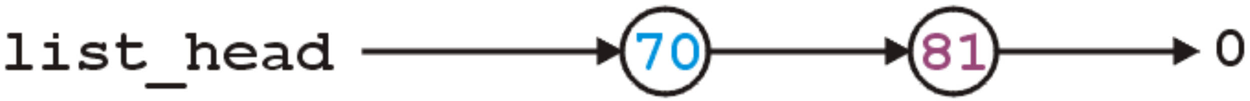


void pop_front()

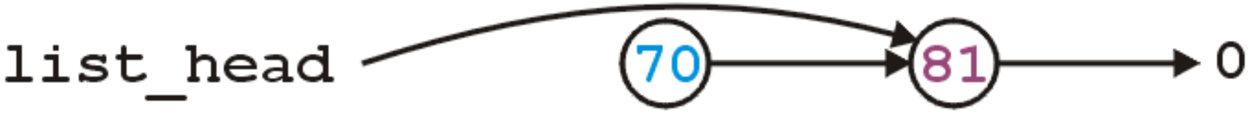
Erasing from the front of a linked list is even easier:

- We assign the list head to the next pointer of the first node

Graphically, given:



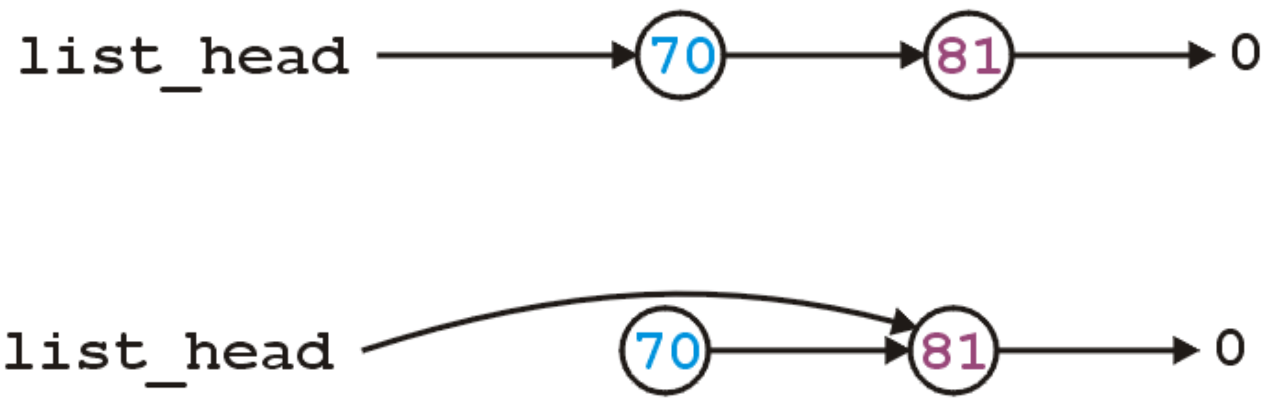
we want:



void pop_front()

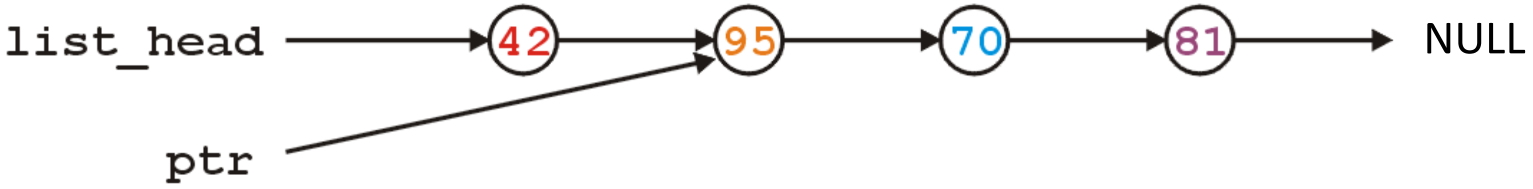
Easy enough:

```
list_head = list_head->next;
```



Stepping through a Linked List

ptr != NULL and thus we evaluate the loop and increment the pointer



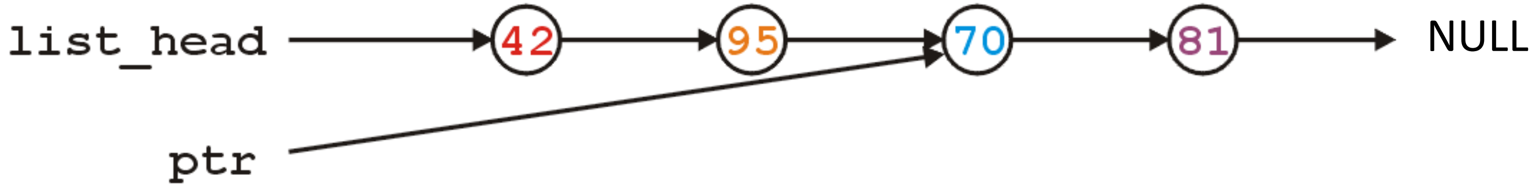
In the loop, we can access the value being pointed to by using

ptr->value

Stepping through a Linked List

ptr != NULL and thus we evaluate the loop and increment the

pointer

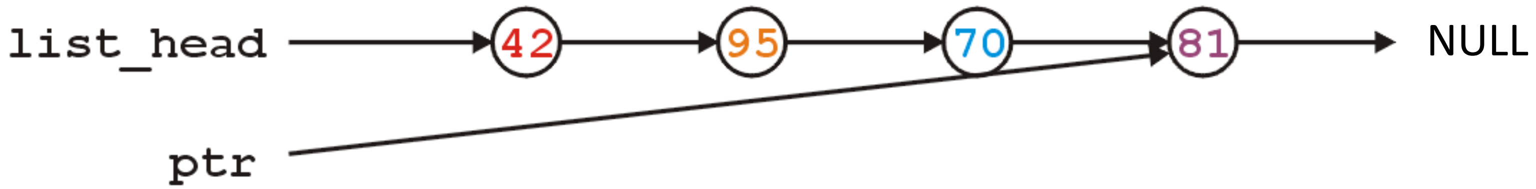


Also, in the loop, we can access the next node in the list by using ptr->next()

Stepping through a Linked List

ptr != nullptr and thus we evaluate the loop and increment the

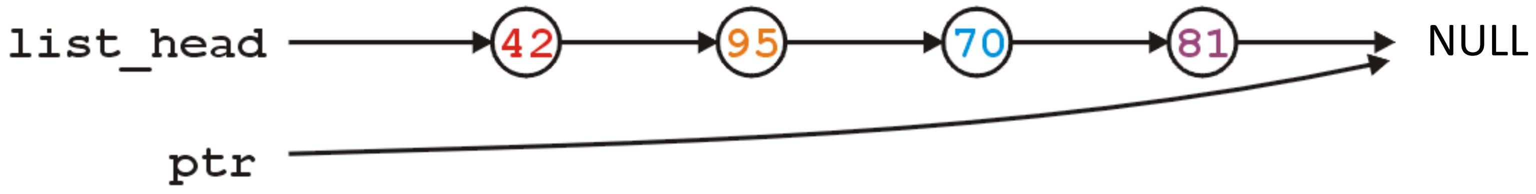
pointer



This last increment causes ptr == NULL

Stepping through a Linked List

Here, we check and find ptr != NULL is false, and thus we exit the loop



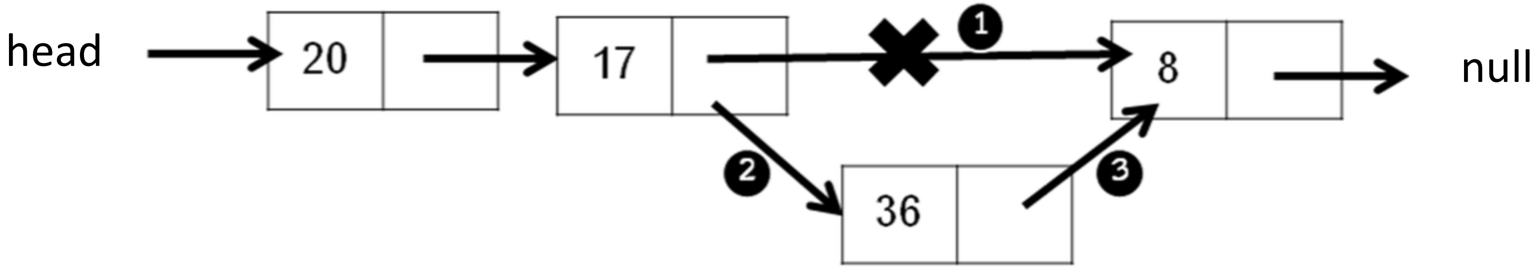
Because the variable ptr was declared inside the loop, we can no longer access it

Stepping through a Linked List

```
 for ( Node *ptr = list_head; ptr != NULL; ptr = ptr->next ) {
     // do something
}
```

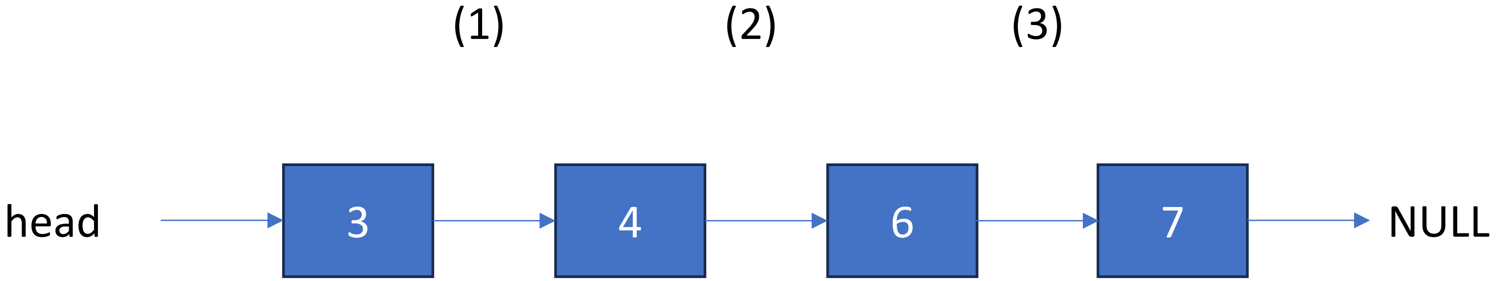
Insert()

To insert an arbitrary value which are not head or tail positions,



Insert() example

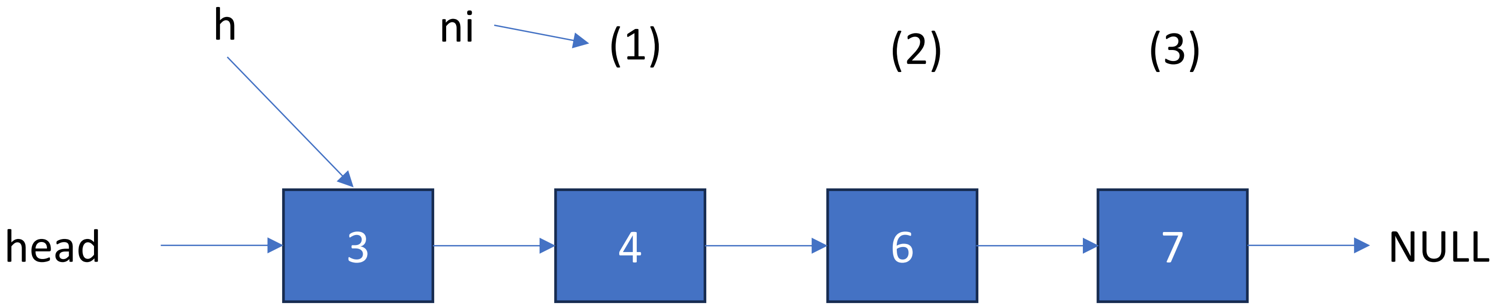
Input: position=2, value=5



Insert() example

Input: position=2, value=5

Traversal pointer (ni=1)

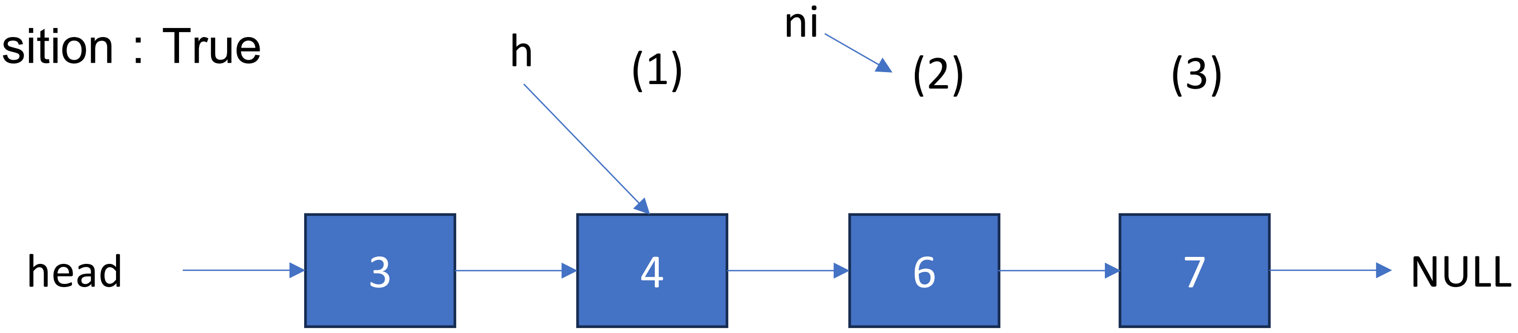


Insert() example

Input: position=2, value=5

Traversal pointer (ni=2)

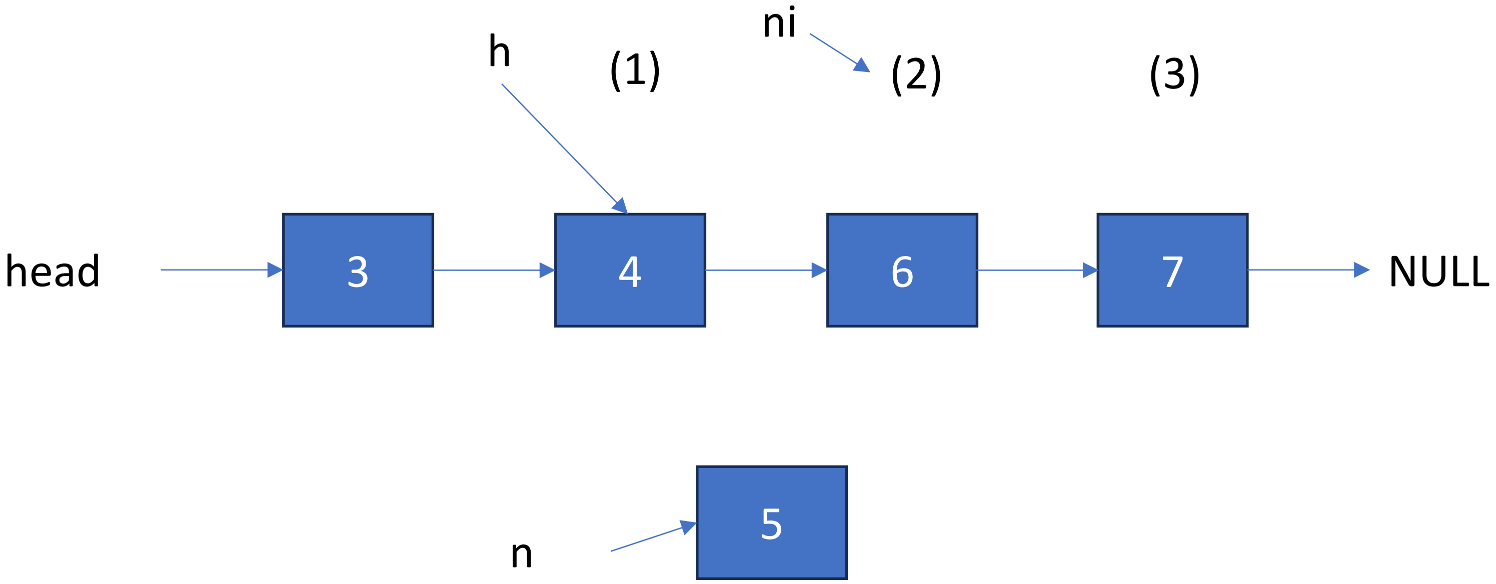
ni == position : True



Insert() example

Input: position=2, value=5

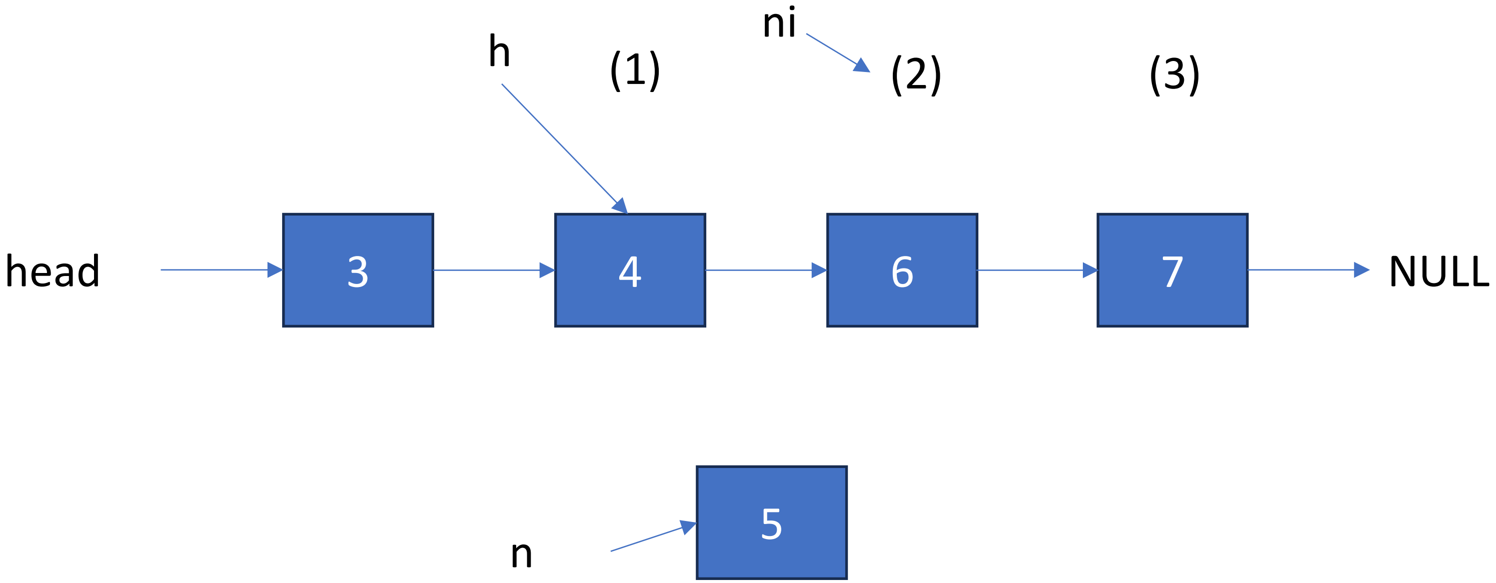
Create new Node(5)



Insert() example

Input: position=2, value=5

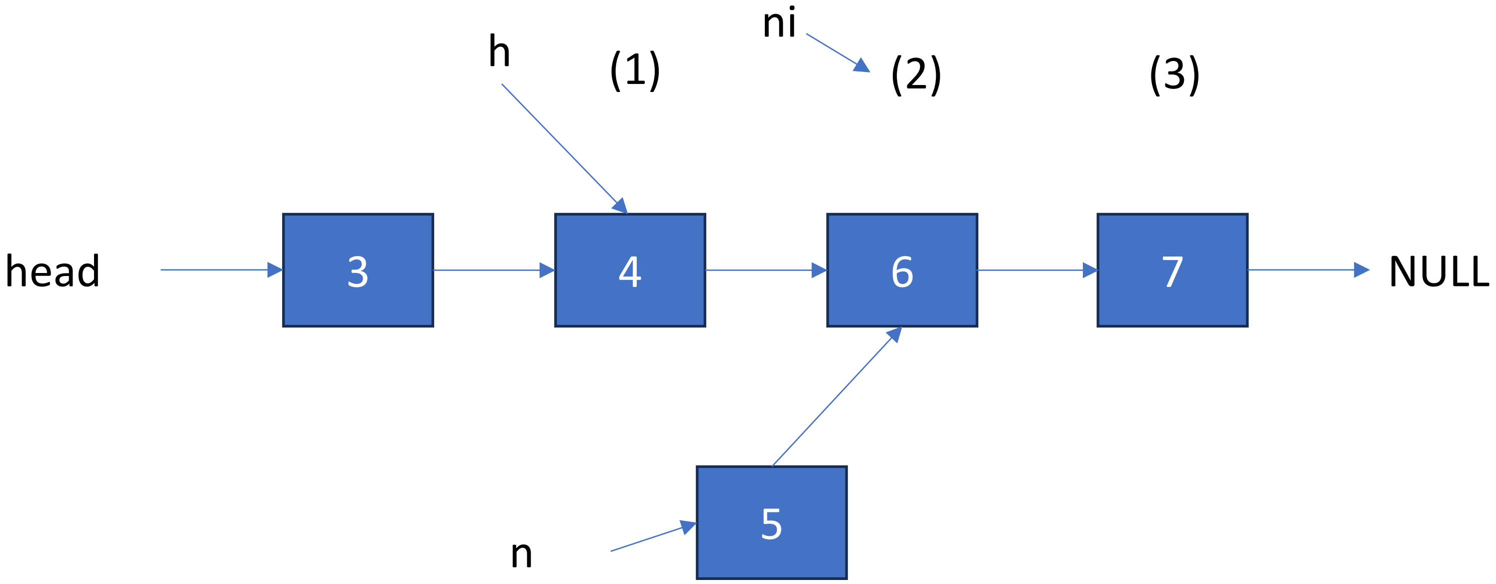
Create new Node(5)



Insert() example

Input: position=2, value=5

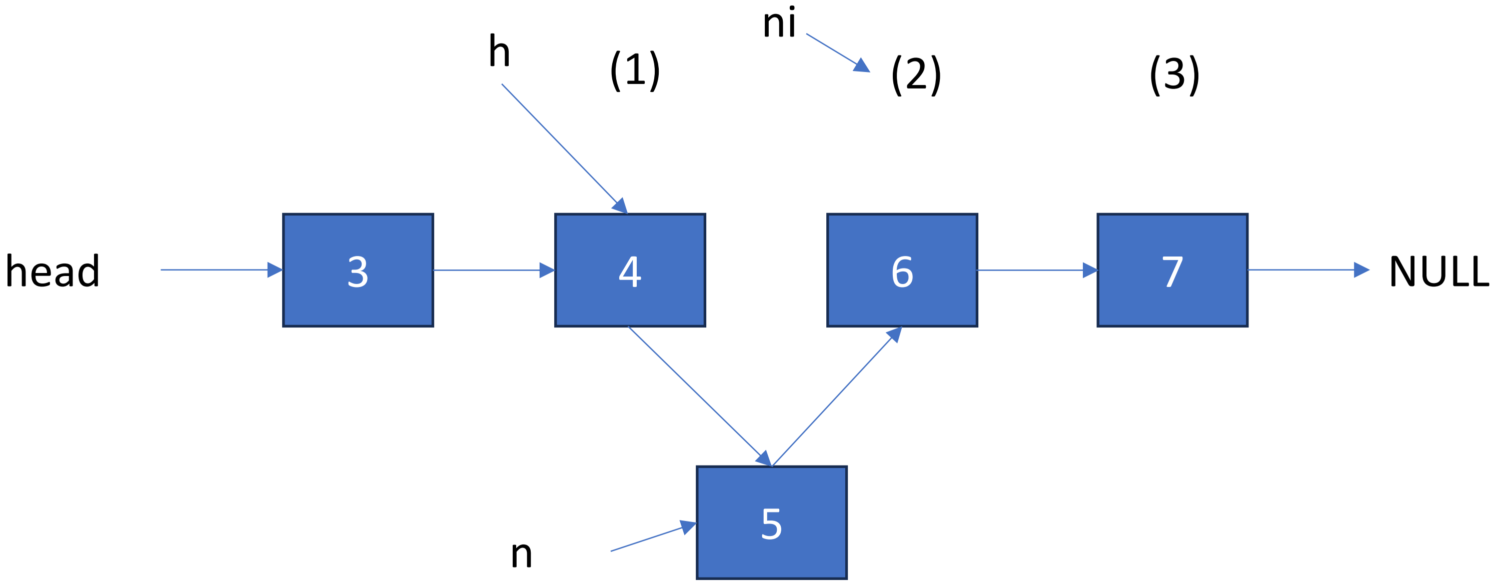
n->next = h->next



Insert() example

Input: position=2, value=5

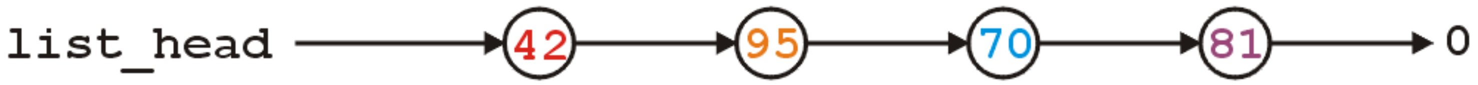
h->next = n



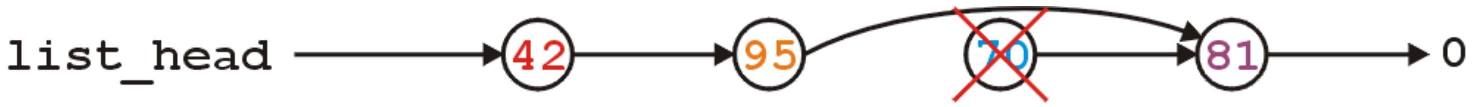
delete

To remove an arbitrary value, *i.e.*, to implement void delete(int), we must update the previous node

For example, given



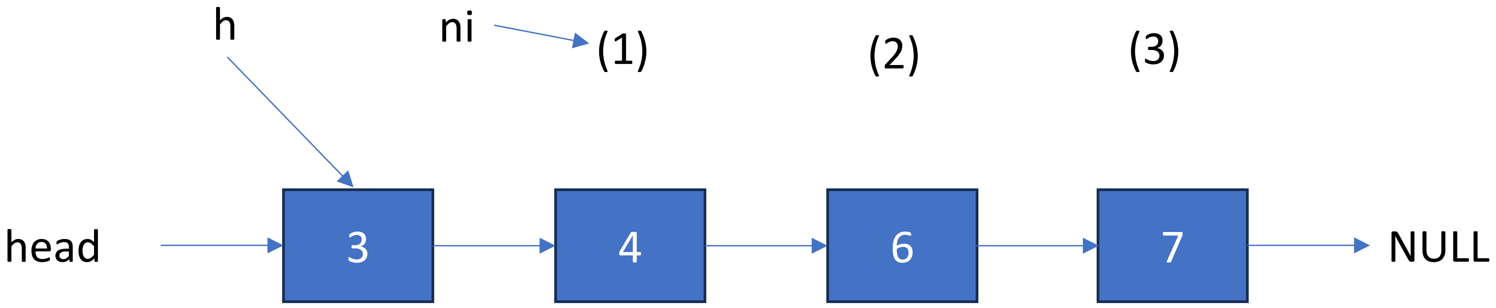
if we delete **70**, we want to end up with



delete example

Input: position=2

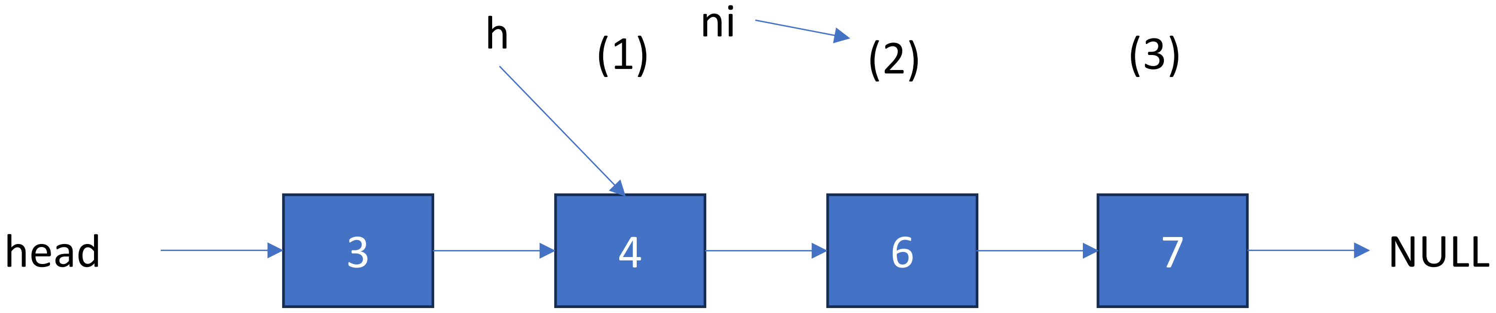
Traversal pointer (ni=1)



delete example

Input: position=2

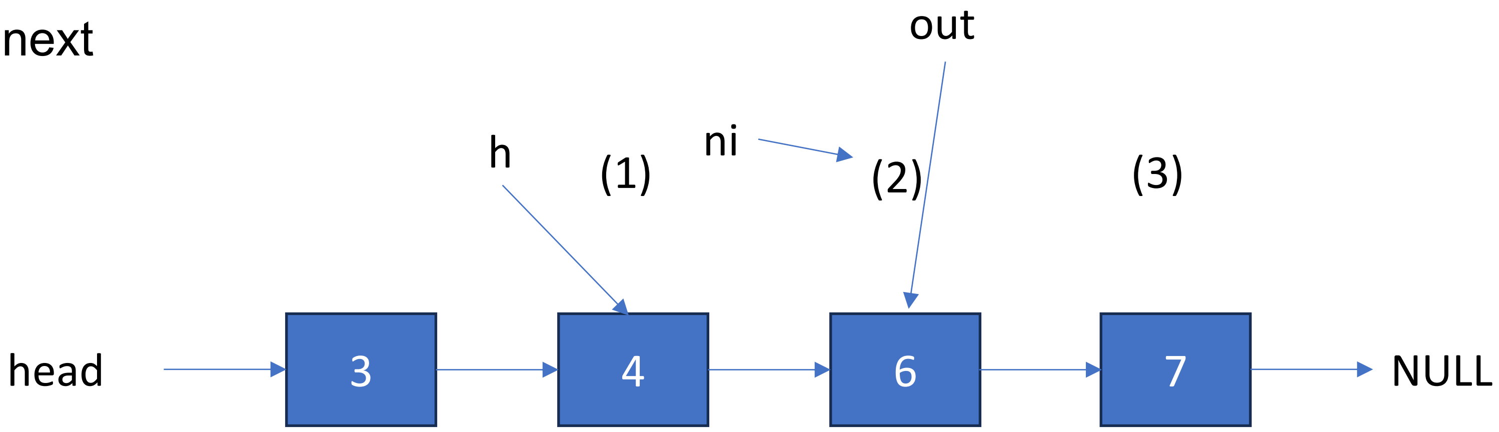
Traversal pointer (ni=2)



delete example

Input: position=2

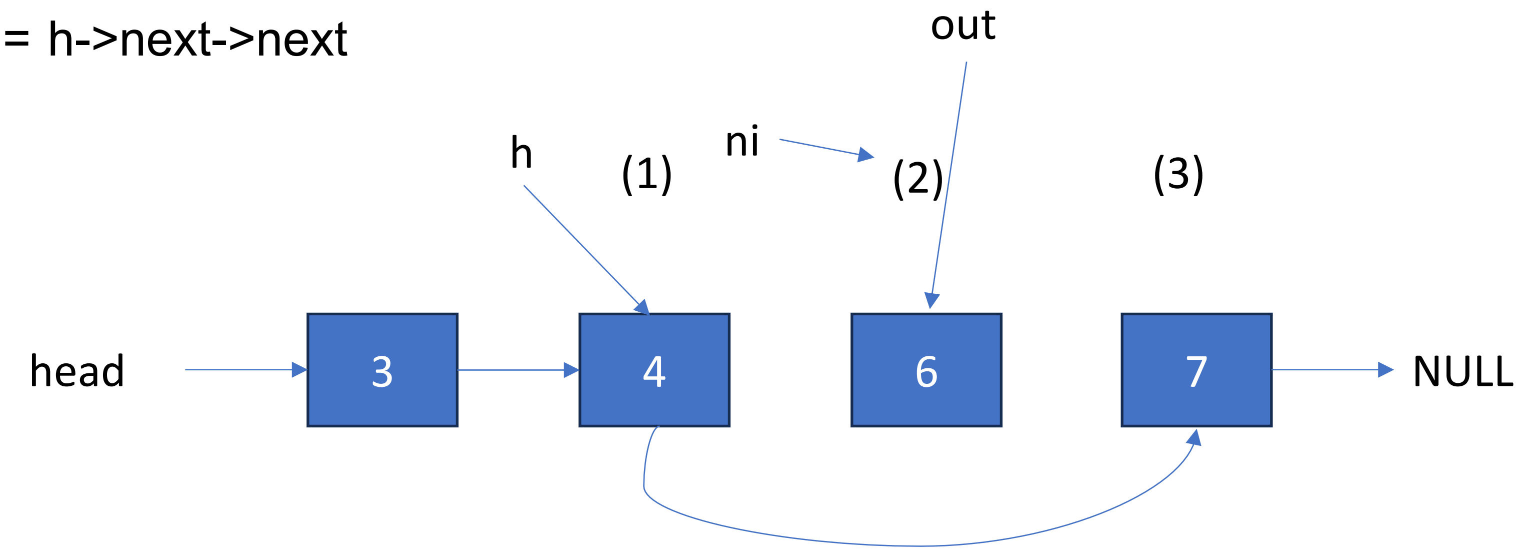
out = h->next



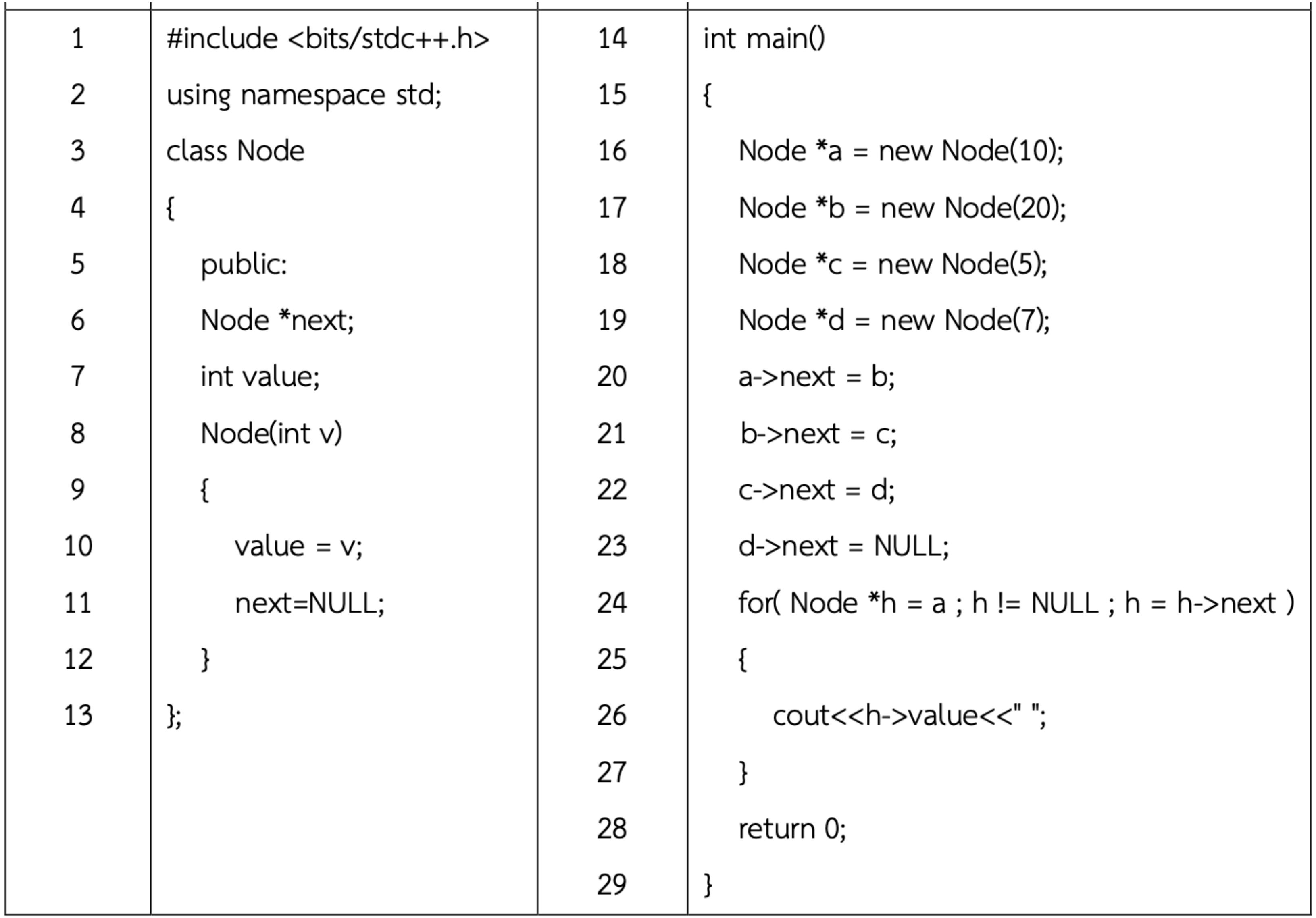
delete example

Input: position=2

h->next = h->next->next



Example: Class Node



```
#include <bits/stdc++.h>
using namespace std;
class Node
{
    public:
    Node *next;
    int value;
    Node(int v)
    {
        value = v;
        next=NULL;
    }
};
int main()
{
    Node *a = new Node(10);
    Node *b = new Node(20);
    Node *c = new Node(5);
    Node *d = new Node(7);
    a->next = b;
    b->next = c;
    c->next = d;
    d->next = NULL;
    for( Node *h = a ; h != NULL ; h = h->next )
    {
        cout<<h->value<<" ";
    }
    return 0;
}
```

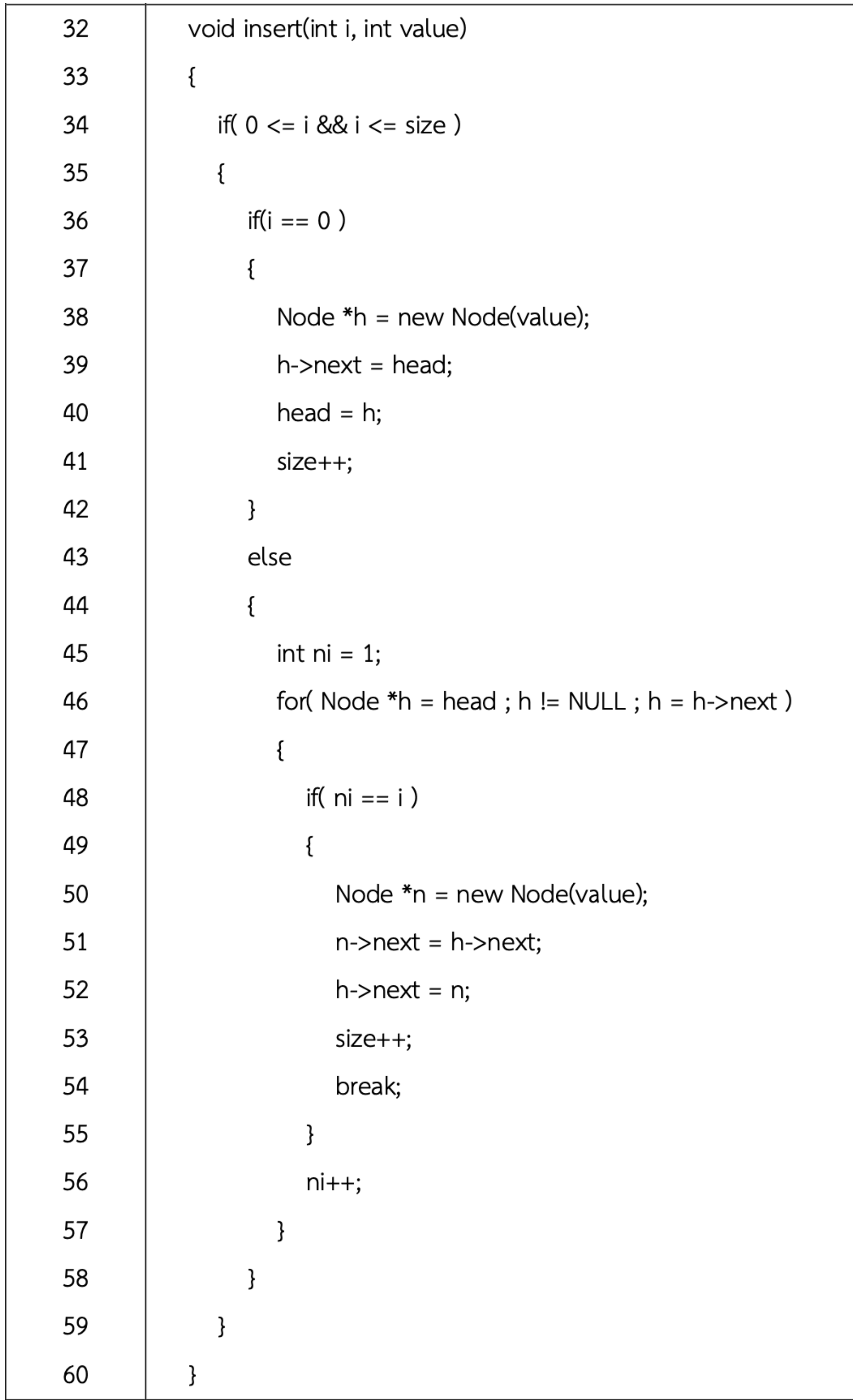
Example: Linked List

```
#include <bits/stdc++.h>
using namespace std;
class Node
{
    public:
    Node *next;
    intvalue;
    Node(int v)
    {
        value = v;   next=NULL;
    }
};
class LinkList
{
    public:
    Node *head;
    int size = 0;
    LinkList(int value)
    {
        head = new Node(value);
        head->next = NULL;
        size = 1;
    }
    void print()
    {
        for( Node *h = head ; h != NULL ; h = h->next )
        {
            cout<<h->value<<" ";
        }
        cout<<endl;
    }
```

Example: Linked List (insert)

push_front()

Insert at i position



```
void insert(int i, int value)
{
    if( 0 <= i && i <= size )
    {
        if(i == 0 )
        {
            Node *h = new Node(value);
            h->next = head;
            head = h;
            size++;
        }
        else
        {
            int ni = 1;
            for( Node *h = head ; h != NULL ; h = h->next )
            {
                if( ni == i )
                {
                    Node *n = new Node(value);
                    n->next = h->next;
                    h->next = n;
                    size++;
                    break;
                }
                ni++;
            }
        }
    }
}
```

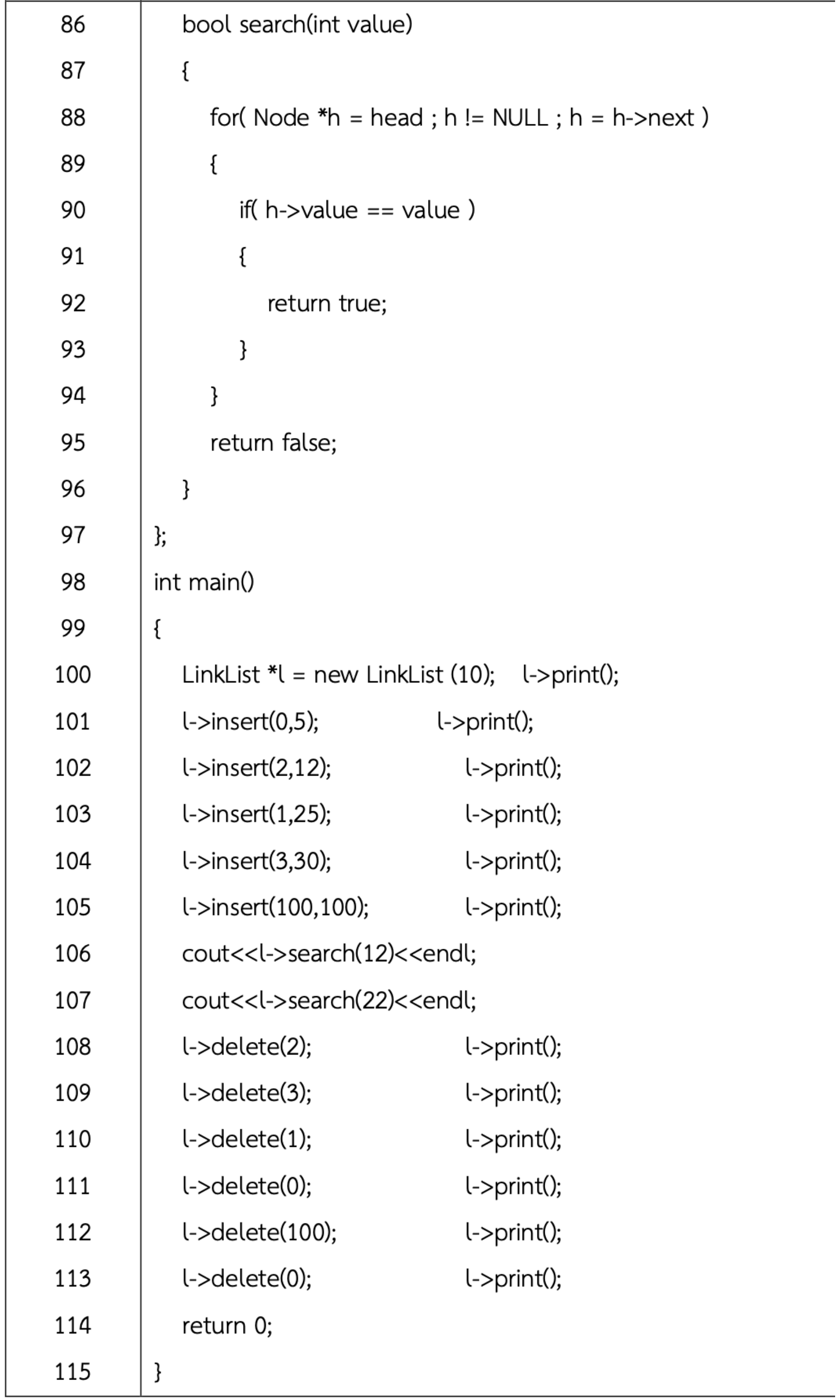
Example: Linked List (delete)

pop_front()

erase() at i position

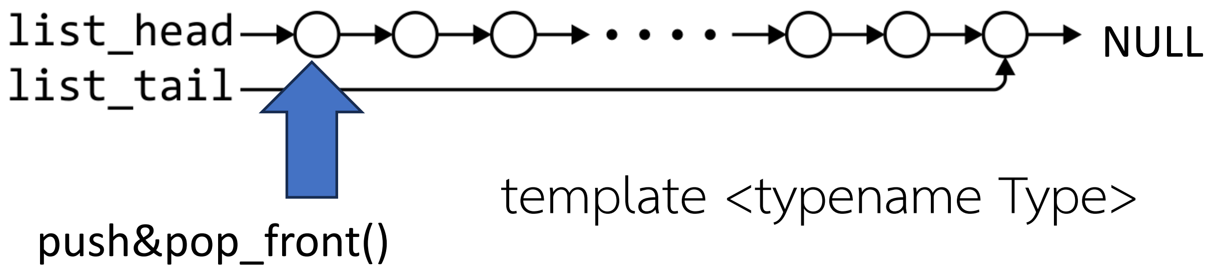
```
void delete(int i)
{
    if( 0 <= i && i <= size && size > 1 )
    {
        if(i == 0)
        {
            head = head->next;
            size--;
        }
        else
        {
            int ni = 1;
            for(Node *h = head ; h != NULL ; h = h->next)
            {
                if(ni == i )
                {
                    h->next = h->next->next;
                    size--;
                    break;
                }
                ni++;
            }
        }
    }
}
```

Example: Linked List (search)



```
    bool search(int value)
    {
        for( Node *h = head ; h != NULL ; h = h->next )
        {
            if( h->value == value )
            {
                return true;
            }
        }
        return false;
    }
};
int main()
{
    LinkList *l = new LinkList (10);    l->print();
    l->insert(0,5);                 l->print();
    l->insert(2,12);                    l->print();
    l->insert(1,25);                    l->print();
    l->insert(3,30);                    l->print();
    l->insert(100,100);                 l->print();
    cout<<l->search(12)<<endl;
    cout<<l->search(22)<<endl;
    l->delete(2);                       l->print();
    l->delete(3);                       l->print();
    l->delete(1);                       l->print();
    l->delete(0);                       l->print();
    l->delete(100);                     l->print();
    l->delete(0);                       l->print();
    return 0;
}
```

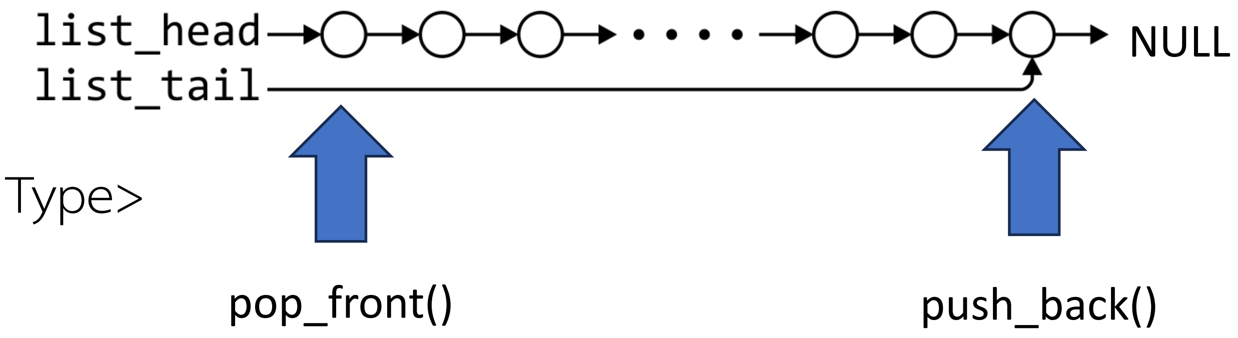
Stack: Linked List Implementation



- The desired behavior of an Abstract Stack may be reproduced by performing all operations at the front

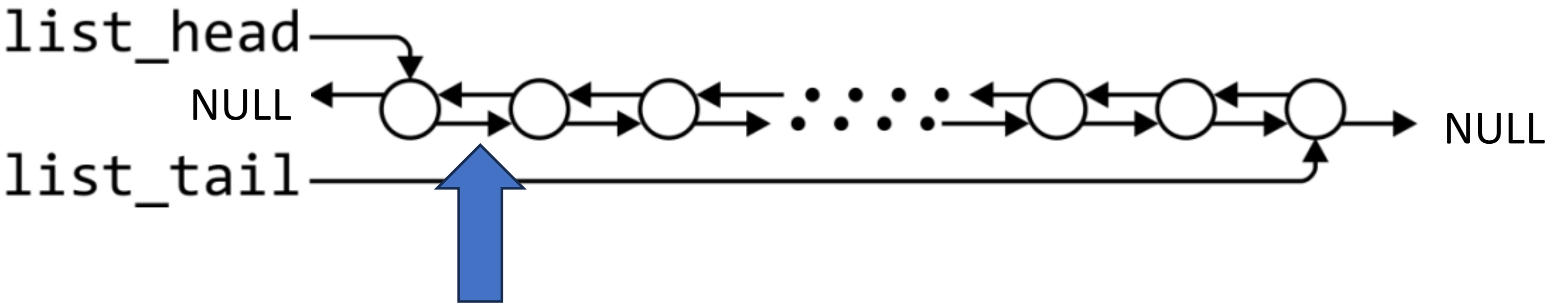
```
template <typename Type>
class Node{ … };
class Stack_list
{
        Node *list_head;
        void push_front(int value);
        Type pop_front();
        …
};
```

Queue: Linked List Implementation



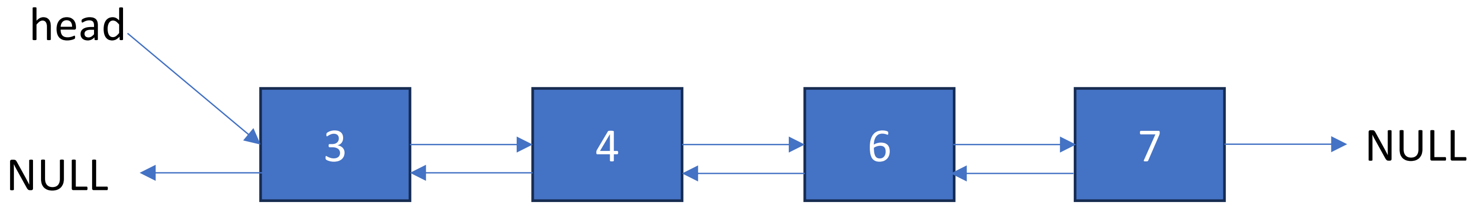
```
template <typename Type>

class Node{ … };

class Queue_list

{

        Node *list_head;

        void push_back(int value);

        Type pop_front();

        …

};
```

Doubly linked list: class Node



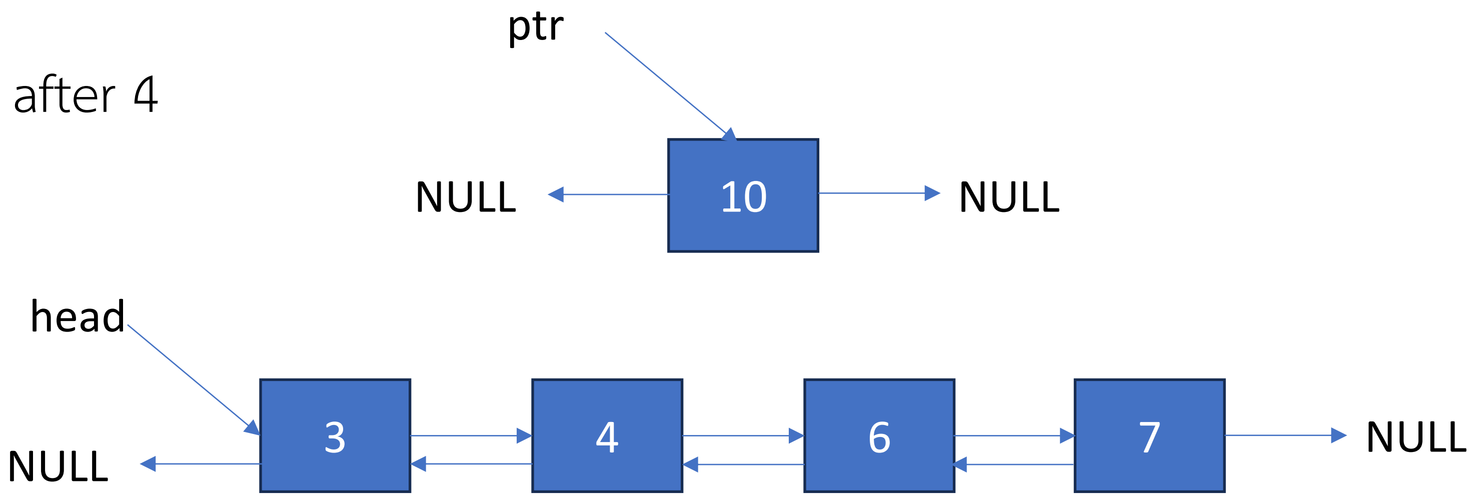
```
class Node{
        int value;
        Node* next;
        Node* prev;
};
```

Doubly linked list: example



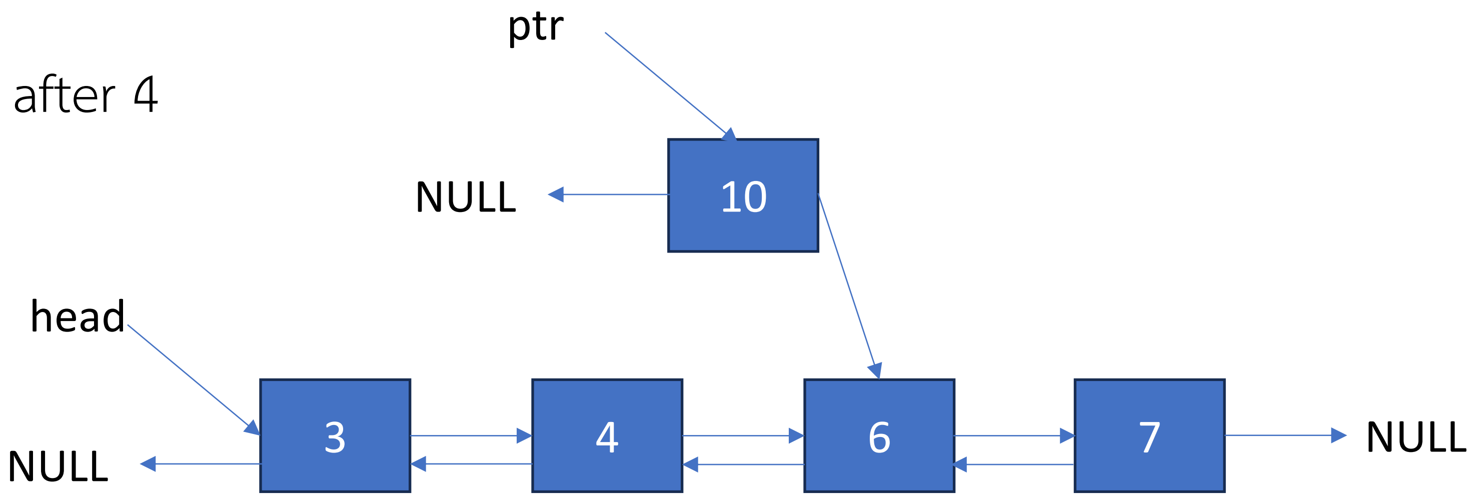
Doubly linked list: example

Insert 10 after 4



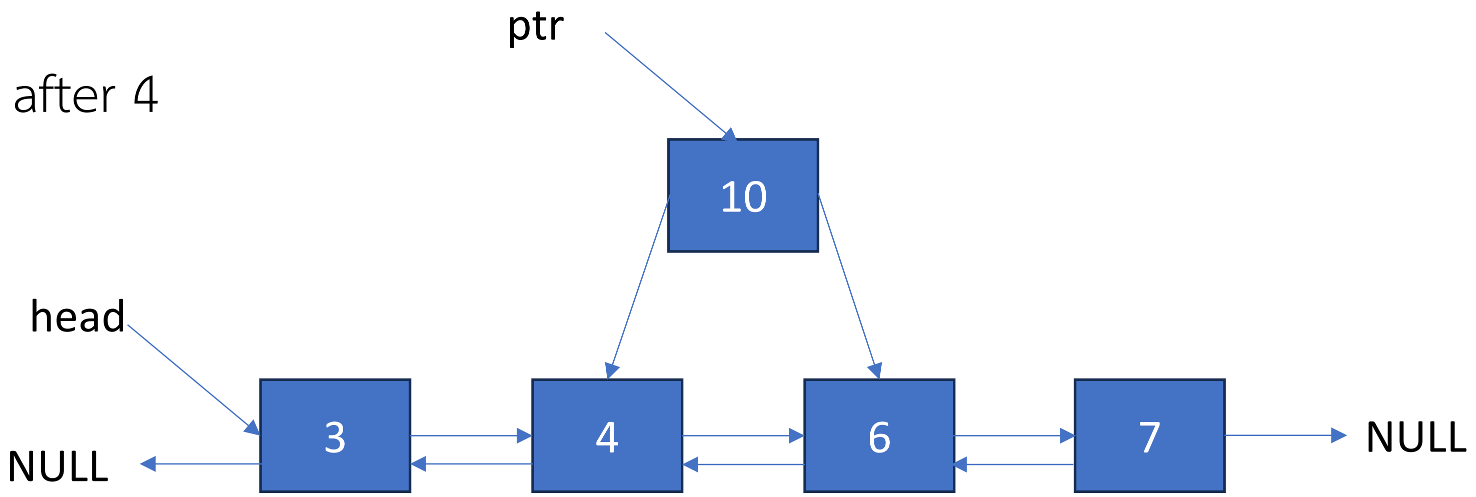
Doubly linked list: example

Insert 10 after 4



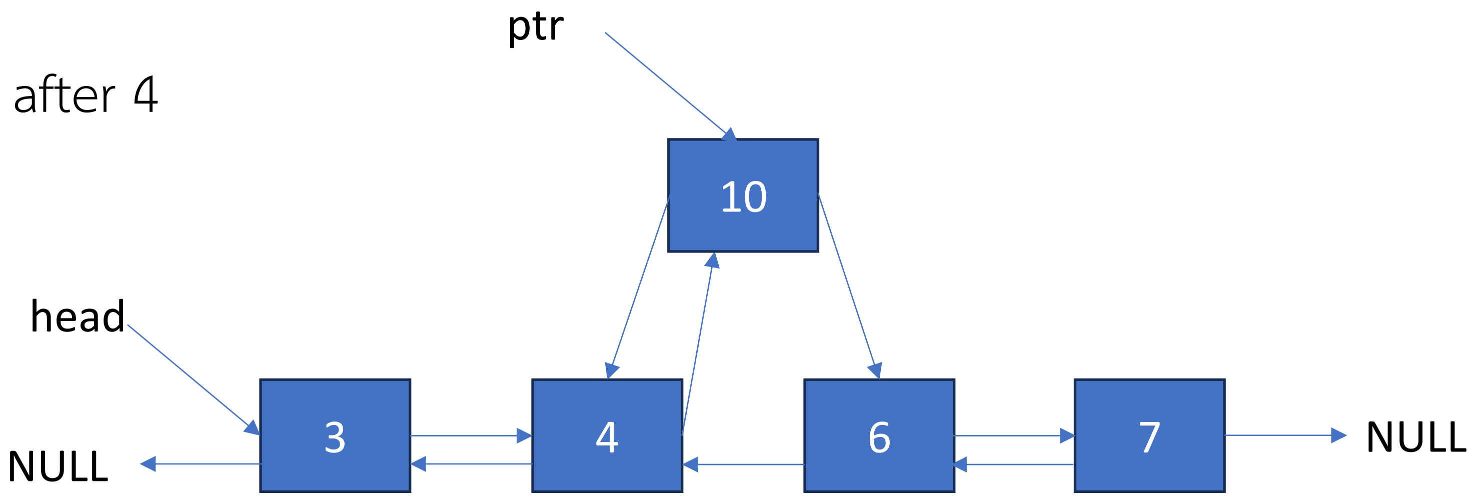
Doubly linked list: example

Insert 10 after 4



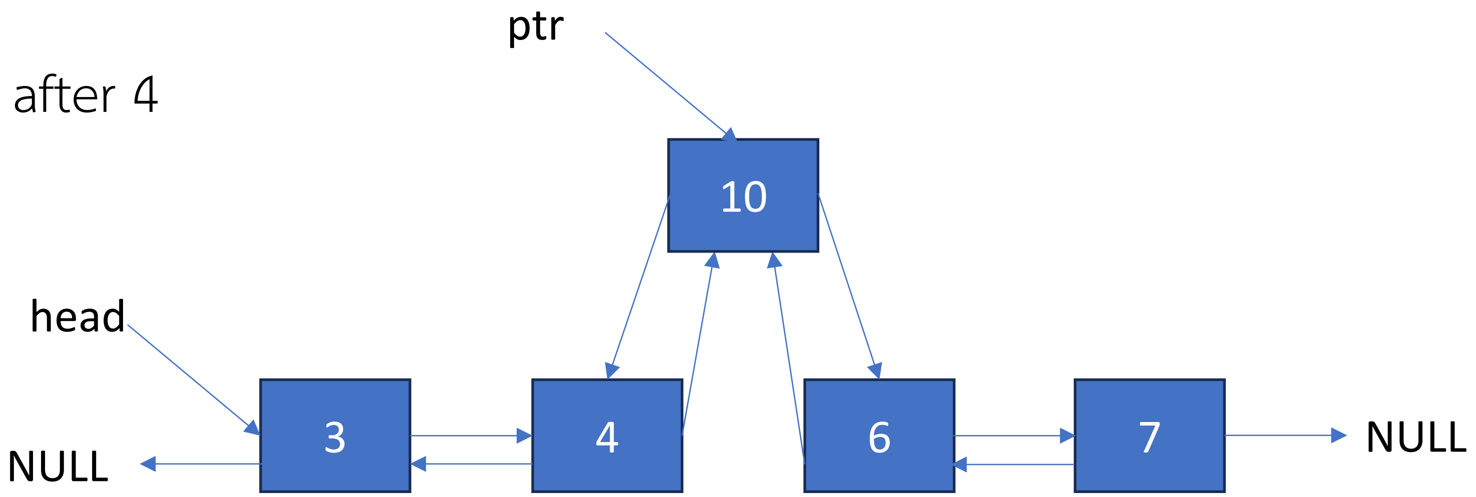
Doubly linked list: example

Insert 10 after 4



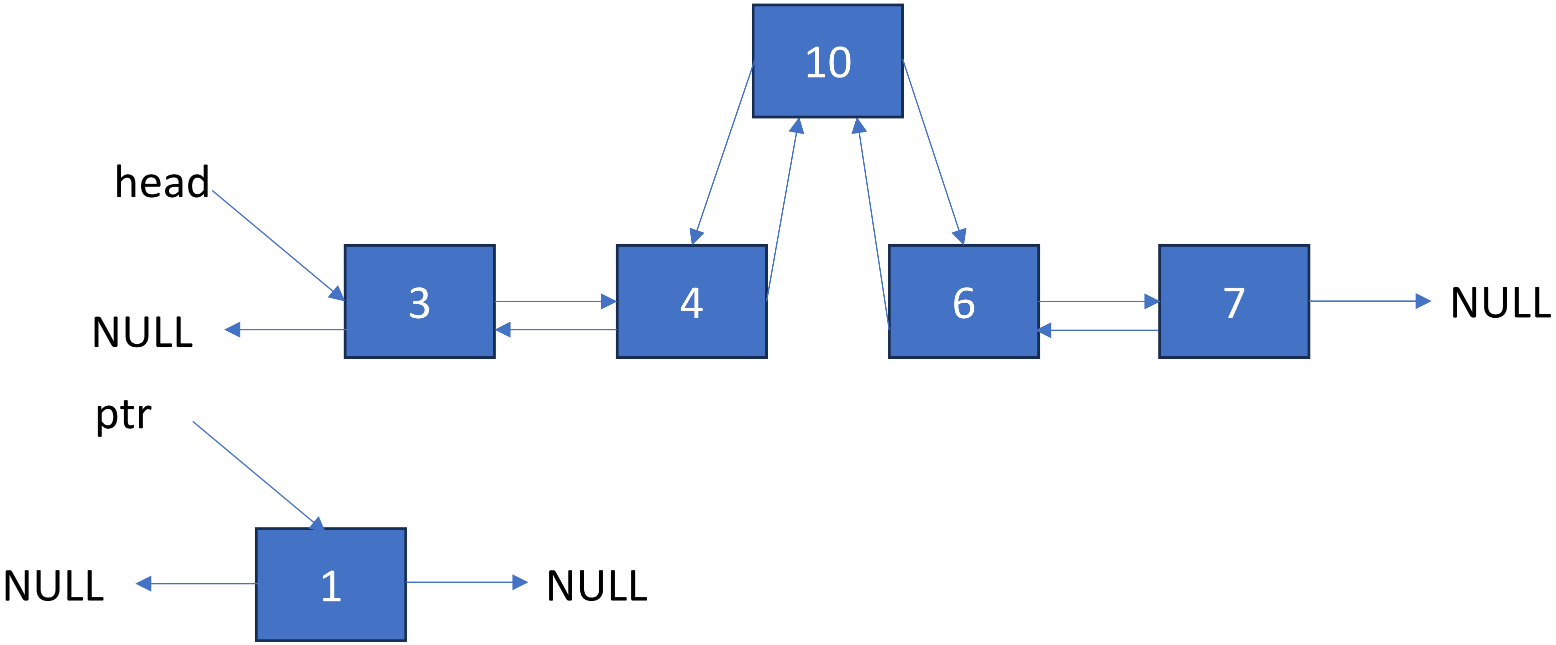
Doubly linked list: example

Insert 10 after 4



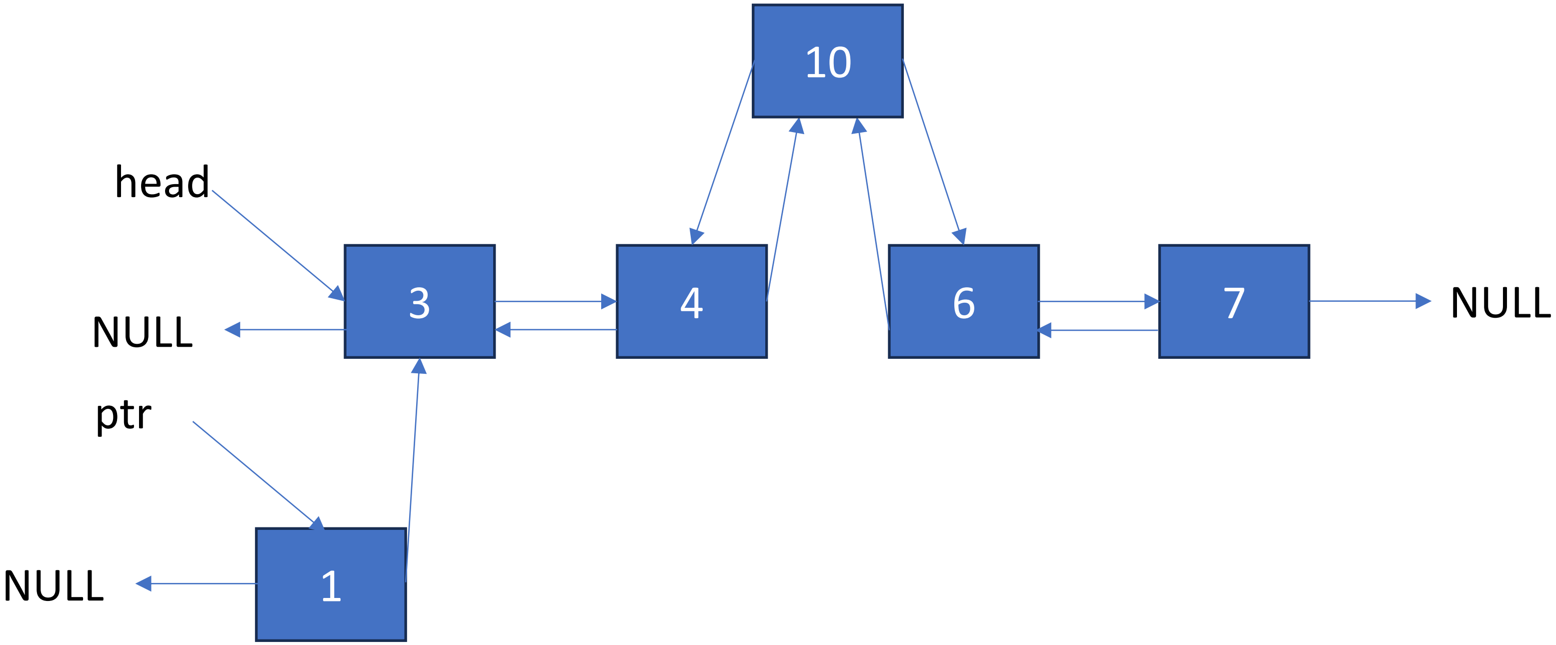
Doubly linked list: example

Insert 1 before 3



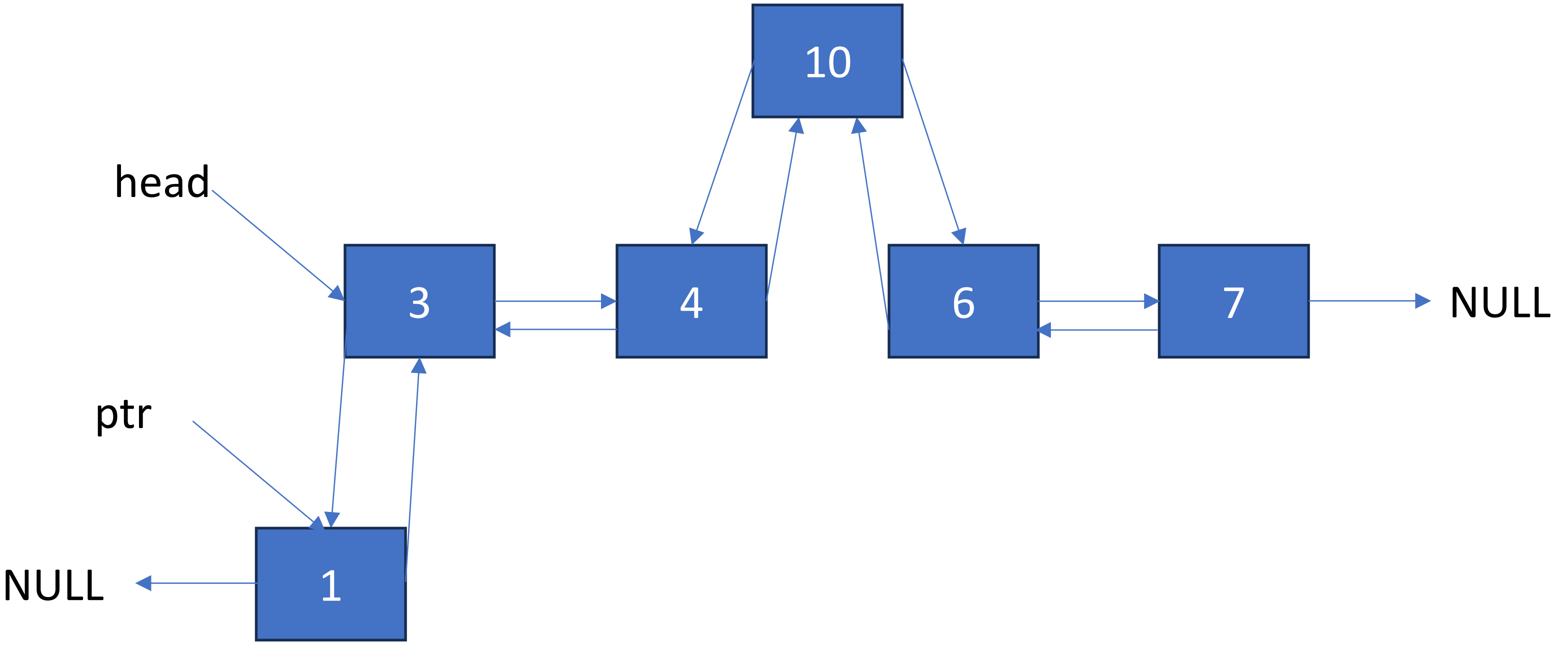
Doubly linked list: example

Insert 1 before 3



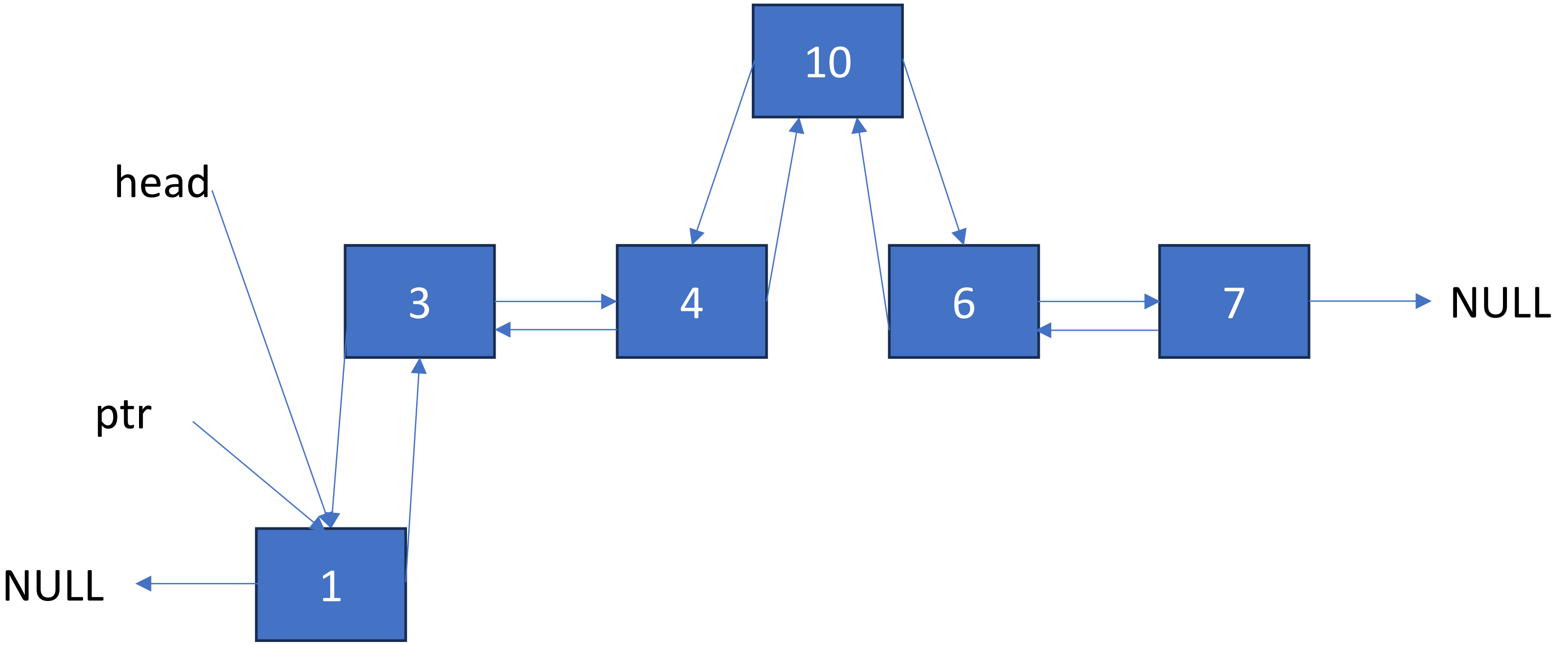
Doubly linked list: example

Insert 1 before 3



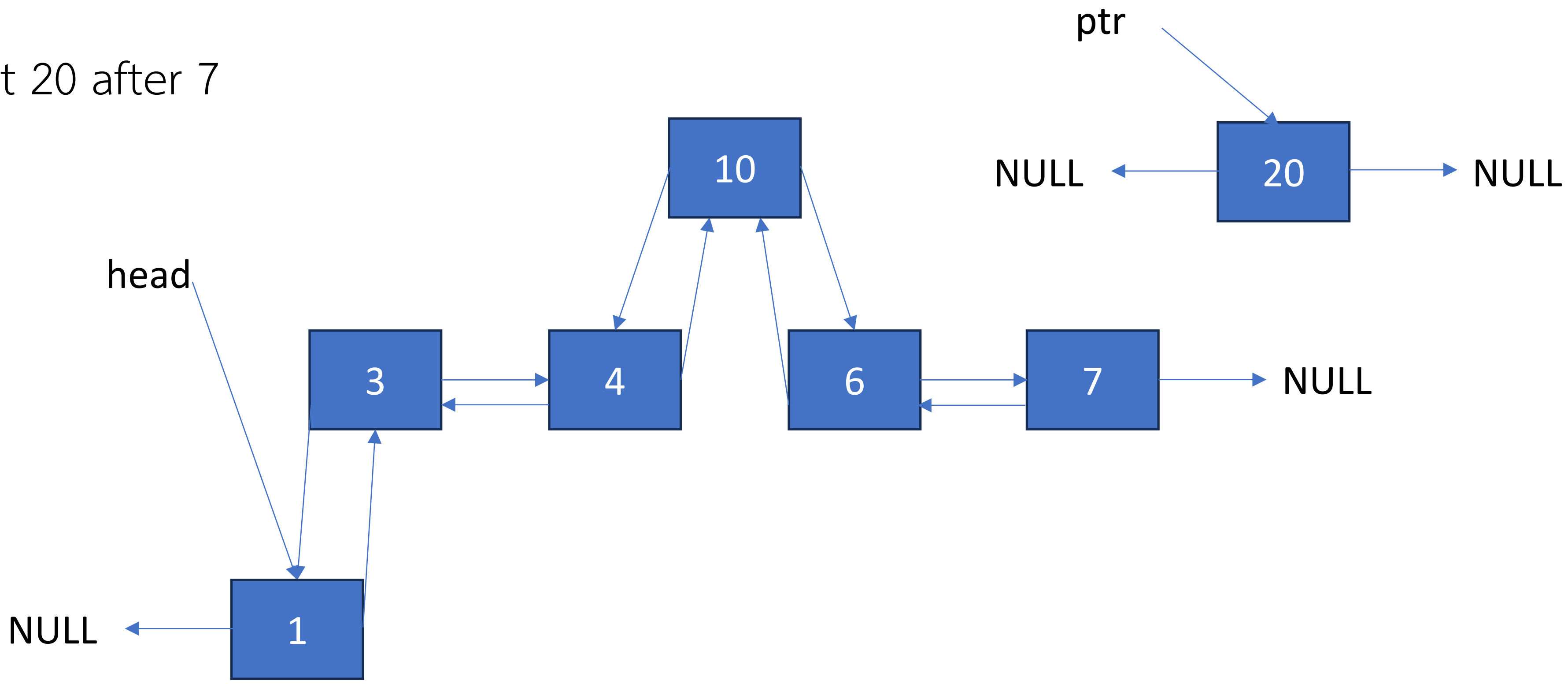
Doubly linked list: example

Insert 1 before 3



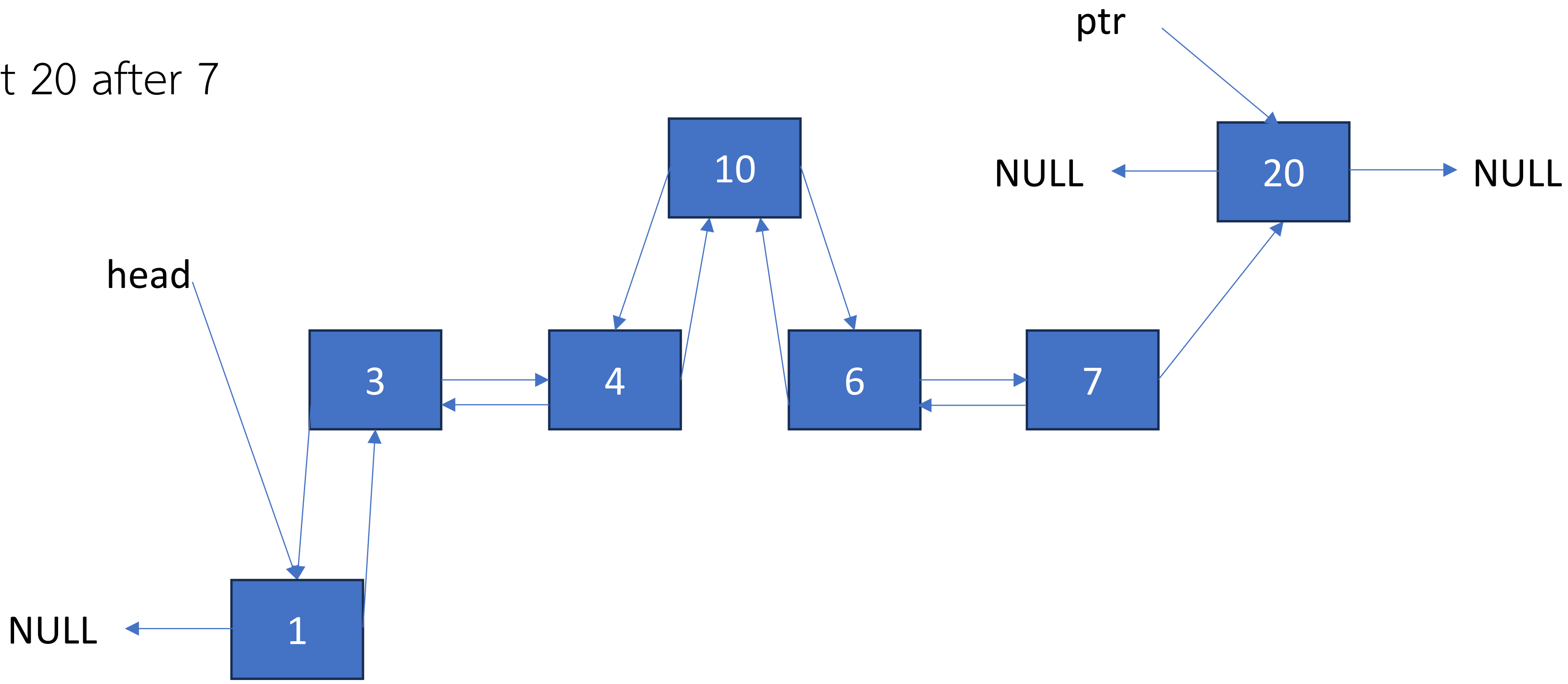
Doubly linked list: example

Insert 20 after 7



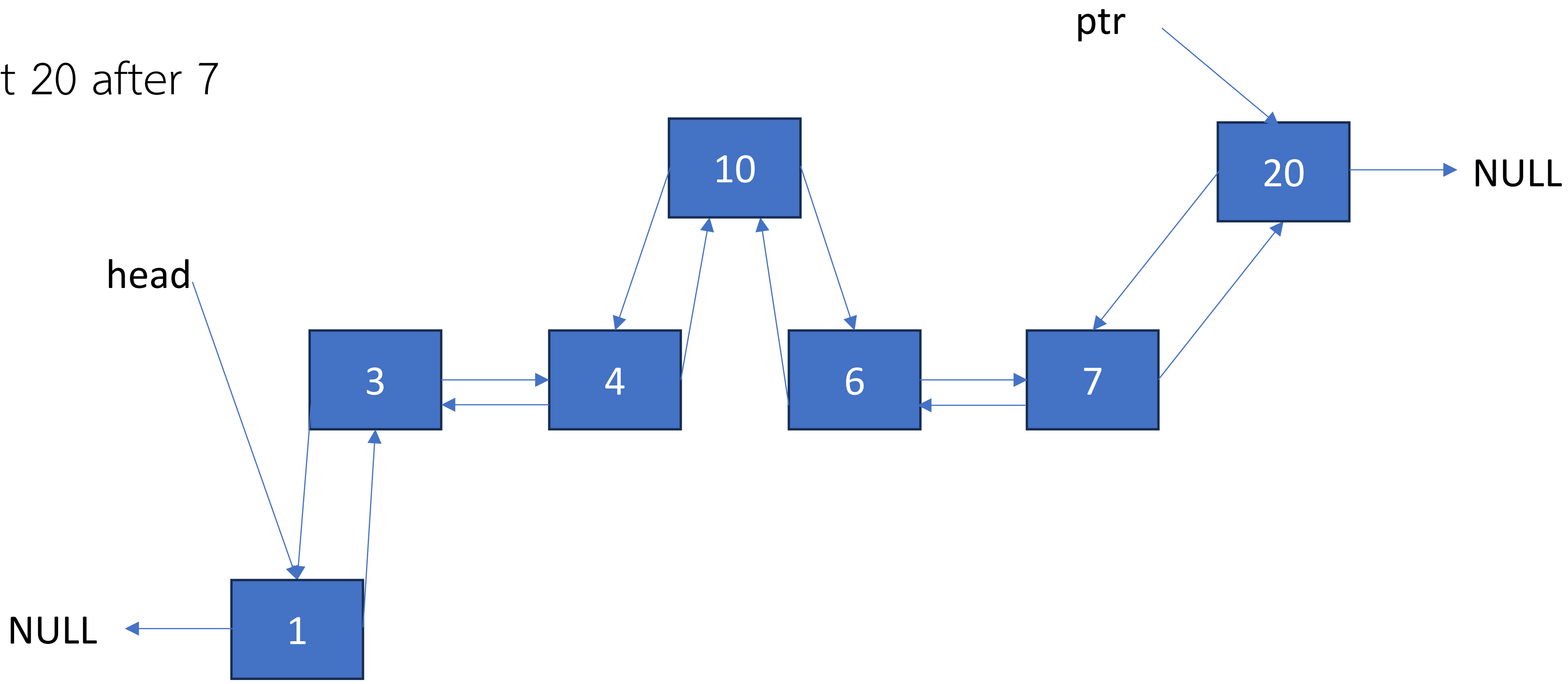
Doubly linked list: example

Insert 20 after 7



Doubly linked list: example

Insert 20 after 7



Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

Stallings, W., & Paul, G. K. (2012). Operating systems: internals and design principles (Vol. 9). New York: Pearson.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/