1/4

Linked Pointers

Data Structures

C++ for C Coders

한동대학교 김영섭 교수

idebtor@gmail.com

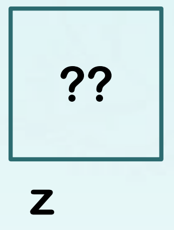
Singly Linked List Concepts

Pointer reviewed – Example 1

```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```

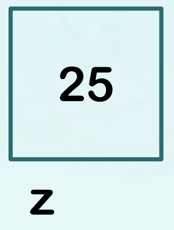
Pointer reviewed – Example 1

```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```



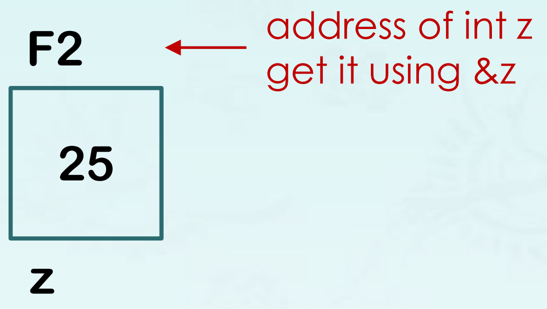
Pointer reviewed – Example 1

```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```



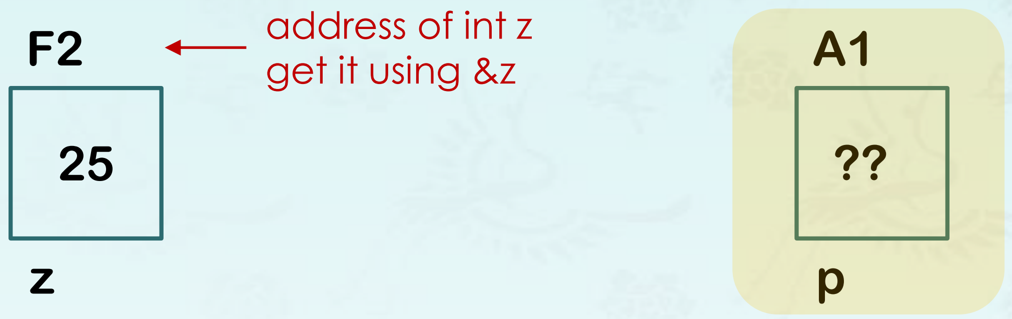
Pointer reviewed – Example 1

```
int z = 25;        // define an int

int* p;            // declare an integer pointer
p = &z;            // p holds the address of z
                   // p points z
```



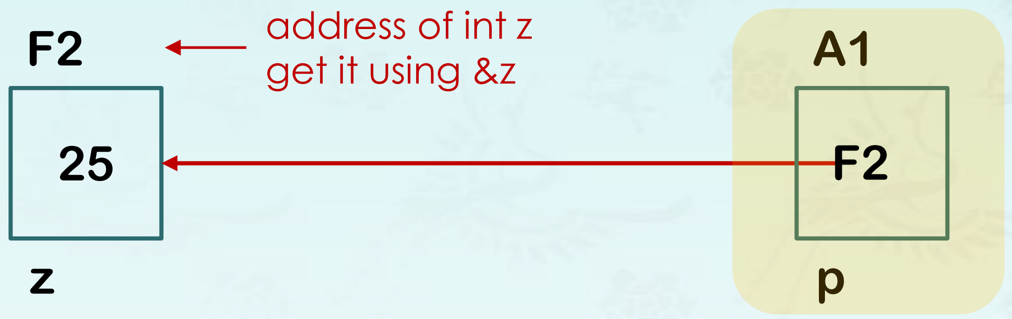
Pointer reviewed – Example 1

```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```



Pointer reviewed – Example 1

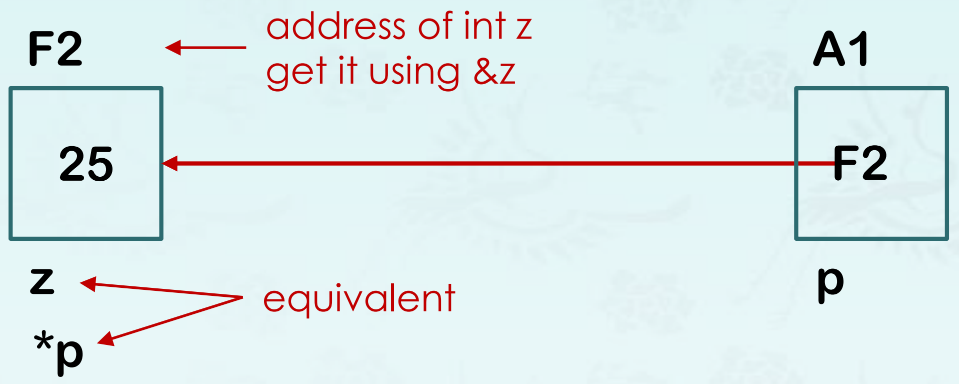
```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```



What is *p?

Pointer reviewed – Example 1

```
int z = 25;     // define an int

int* p;         // declare an integer pointer
p = &z;         // p holds the address of z
                // p points z
```



If p is a pointer, *p is the thing it is pointing at.
Therefore, *p = 25;

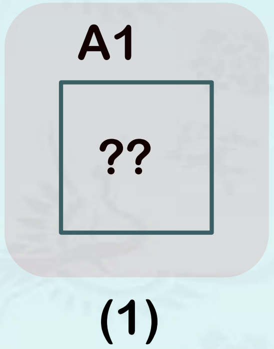
Pointer reviewed – Example 2

Pointer reviewed – Example 2

```
int* p = new int;
```

Pointer reviewed – Example 2

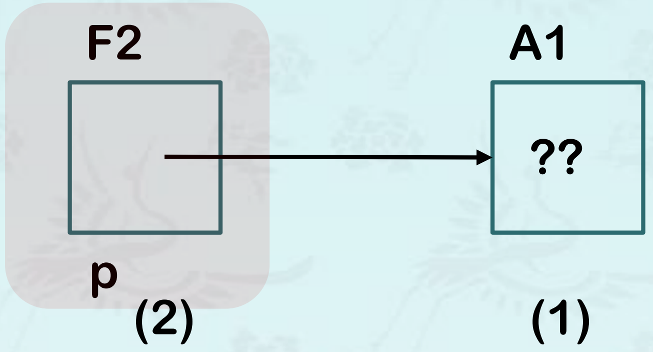
```
int* p = new int;
```



1) `new int;` declares an integer storage space in memory
2) `int *p` makes create a pointer to point an integer storage
3) `=` makes the pointer point at an integer storage.

Pointer reviewed – Example 2

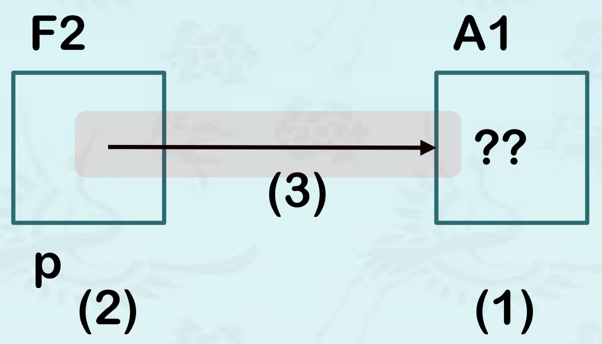
```
int* p = new int;
```



1) `new int;` declares an integer storage space in memory
2) `int *p` makes create a pointer to point an integer storage
3) `=` makes the pointer point at an integer storage.

Pointer reviewed – Example 2

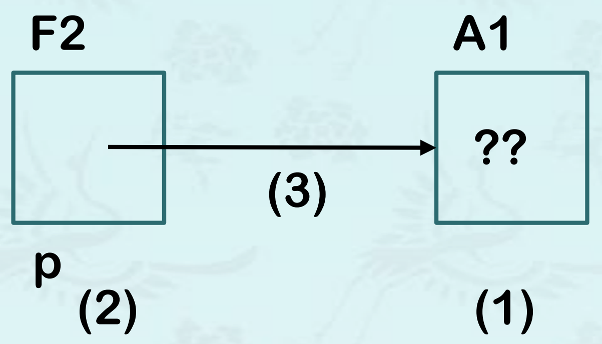
```
int* p = new int;
```



1) `new int;` declares an integer storage space in memory
2) `int *p` makes create a pointer to point an integer storage
3) `=` makes the pointer point at an integer storage.

Pointer reviewed – Example 2

```
int* p = new int;
```

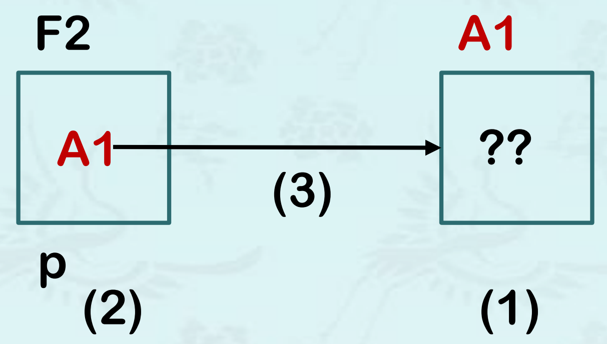


1) `new int;` declares an integer storage space in memory
2) `int *p` makes create a pointer to point an integer storage
3) `=` makes the pointer point at an integer storage.

What is really happening in (3)?

Pointer reviewed – Example 2

```
int* p = new int;
```

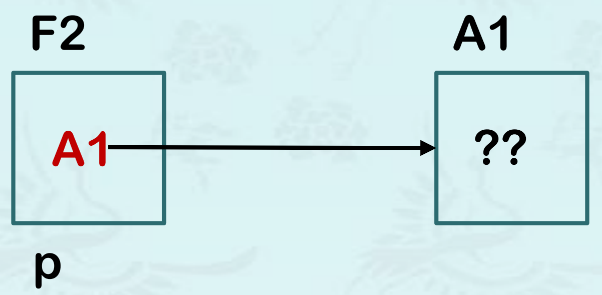


1) `new int;` declares an integer storage space in memory
2) `int *p` makes create a pointer to point an integer storage
3) `=` makes the pointer point at an integer storage.

What is really happening in (3)?

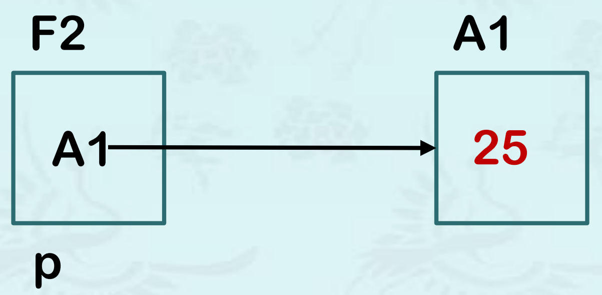
Pointer reviewed – Example 2

```
int* p = new int;
*p = 25;
```



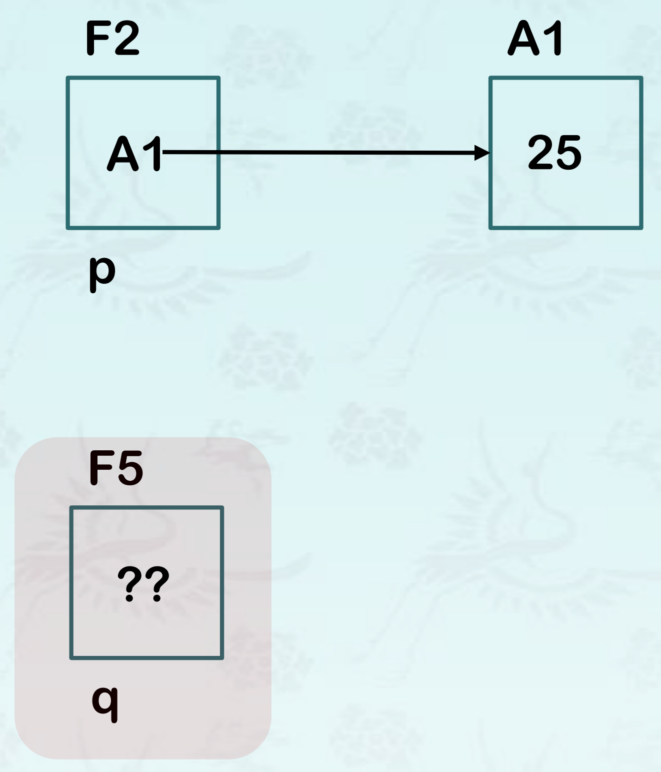
Pointer reviewed – Example 2

```
int* p = new int;
*p = 25;
```



Pointer reviewed – Example 2

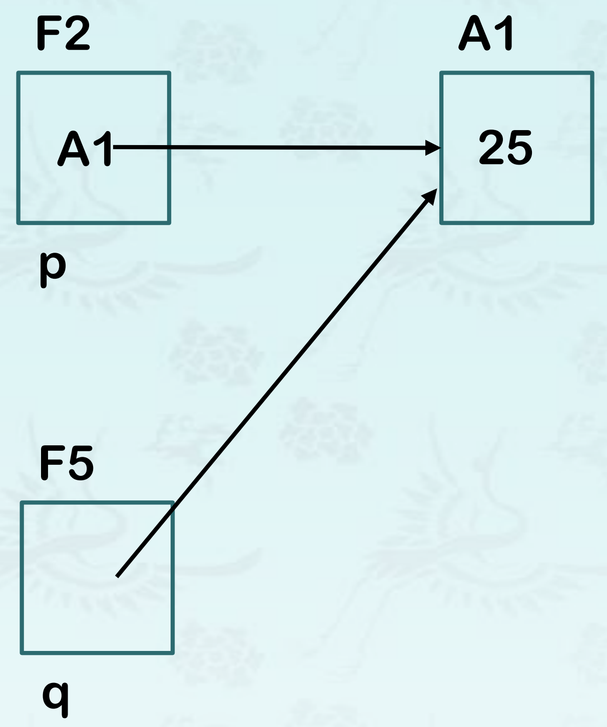
```
int* p = new int;
*p = 25;
cout << *p << endl;
int* q;
```



1) `int* q;` declares a pointer,
2) but it doesn't point anywhere (it's uninitialized) and
3) the statement doesn't assign any memory for the integer data.

Pointer reviewed – Example 2

```
int* p = new int;
*p = 25;
cout << *p << endl;
int* q;
q = p;
```

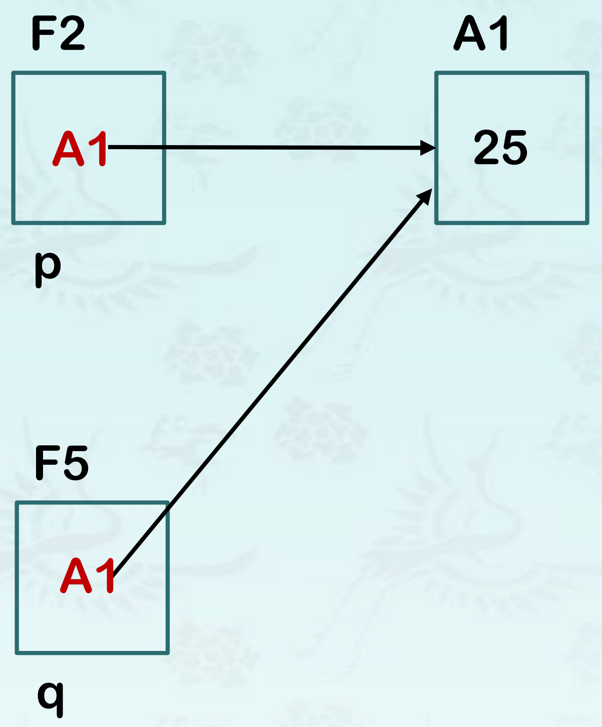


What is really happening in (q = p)?

1) `q = p`; means that `q` is pointing to the same place `p` is pointing at.
2) it does not mean that `q` is pointing at `p`.

Pointer reviewed – Example 2

```
int* p = new int;
*p = 25;
cout << *p << endl;
int* q;
q = p;
```



What is really happening in (q = p)?

1) `q = p`; means that `q` is pointing to the same place `p` is pointing at.
2) it does not mean that `q` is pointing at `p`.

Pointer reviewed

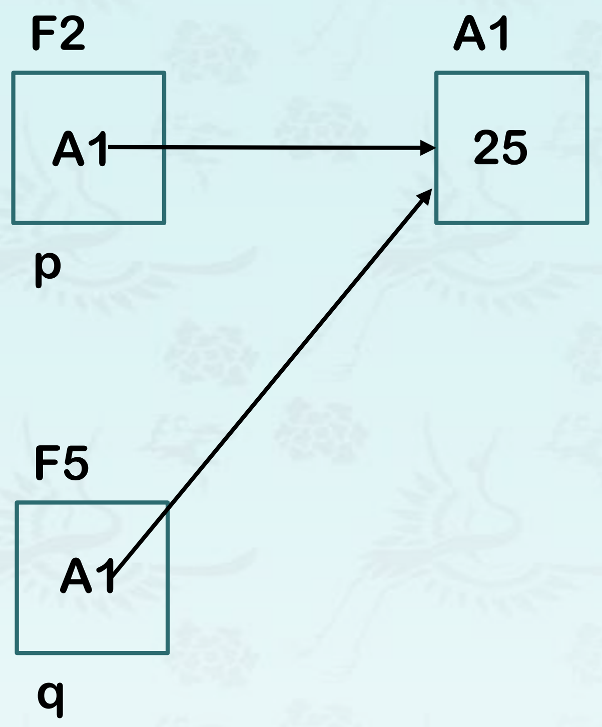
```
int* p = new int;
*p = 25;
cout << *p << endl;
int* q;
q = p;
cout << *q;
```

```
int* p = new int(25);

cout << *p << endl;
int* q = p;

cout << *q;
```

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);   // don't change
p = new int(78);   // don't change
delete p;
delete q;
```

Example 2

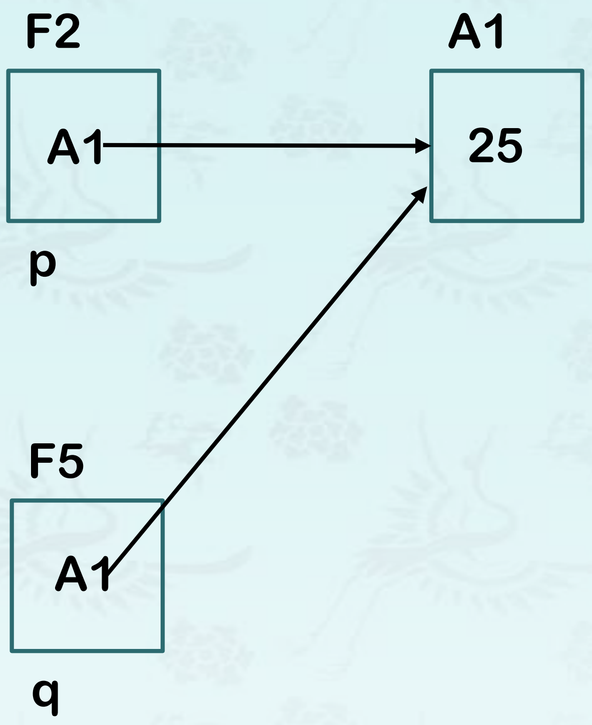


1. Complete the memory diagram based on the code above.
2. Debug code.

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2

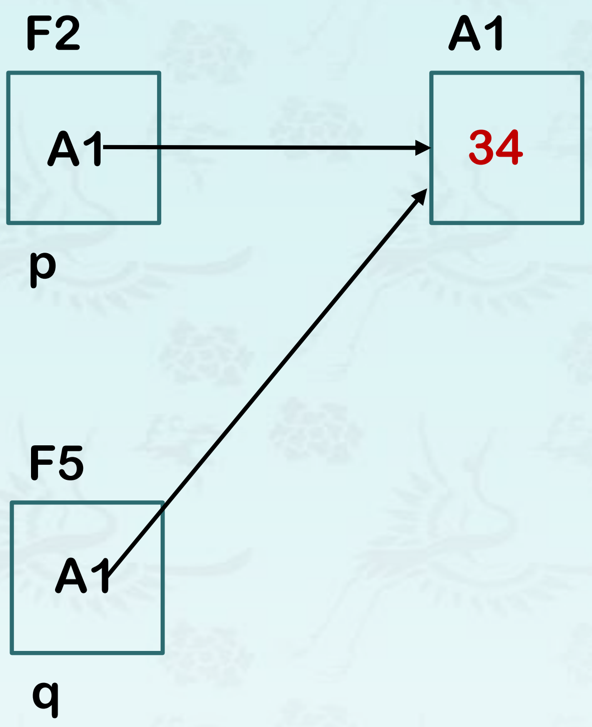


1) What is the effect of assigning a new value to `*q=34`?

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2 (first four lines)

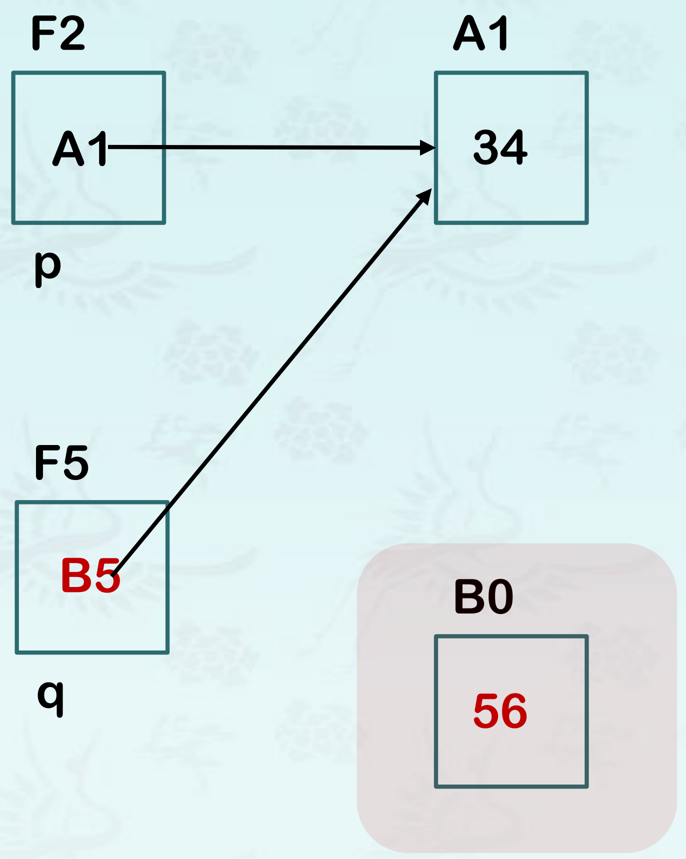


1) What is the effect of assigning a new value to `*q=34`?
Any changes to `*q` will also affect `*p`.

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

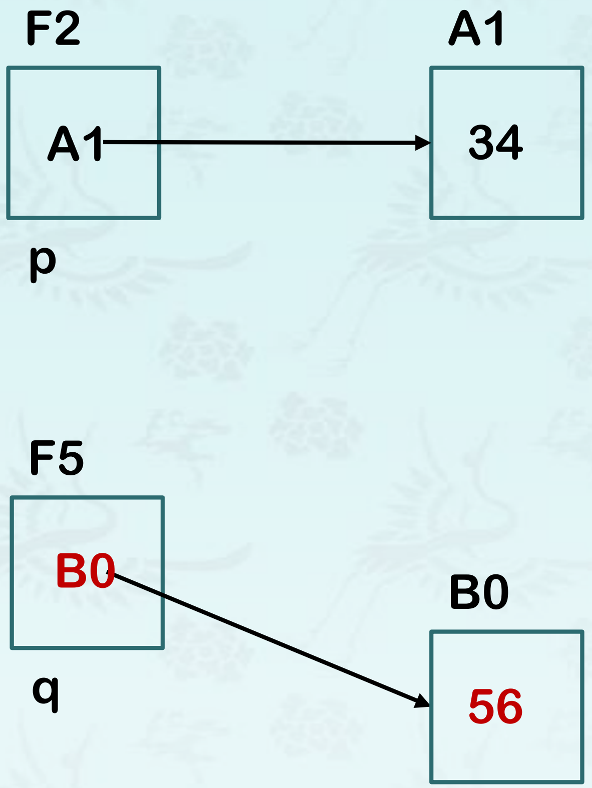
Example 2



Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

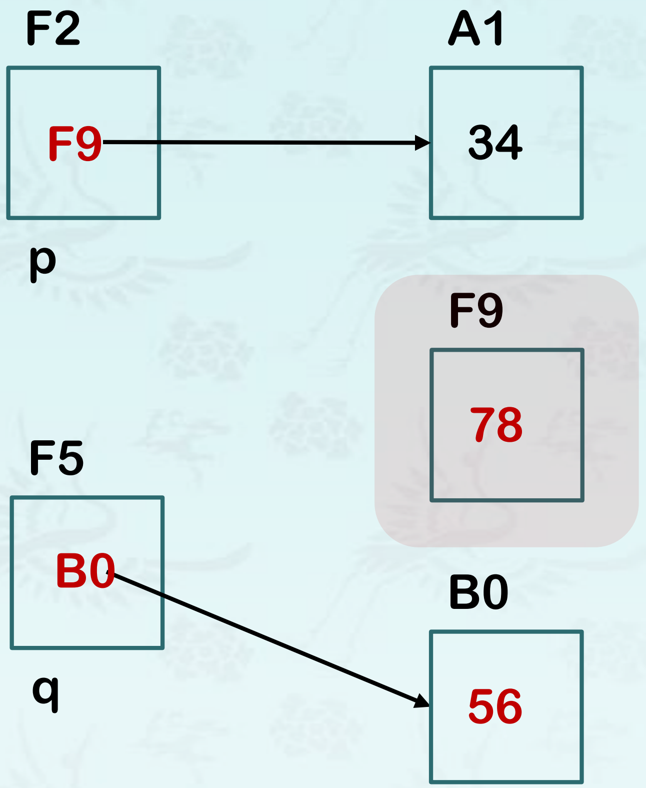
Example 2



Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

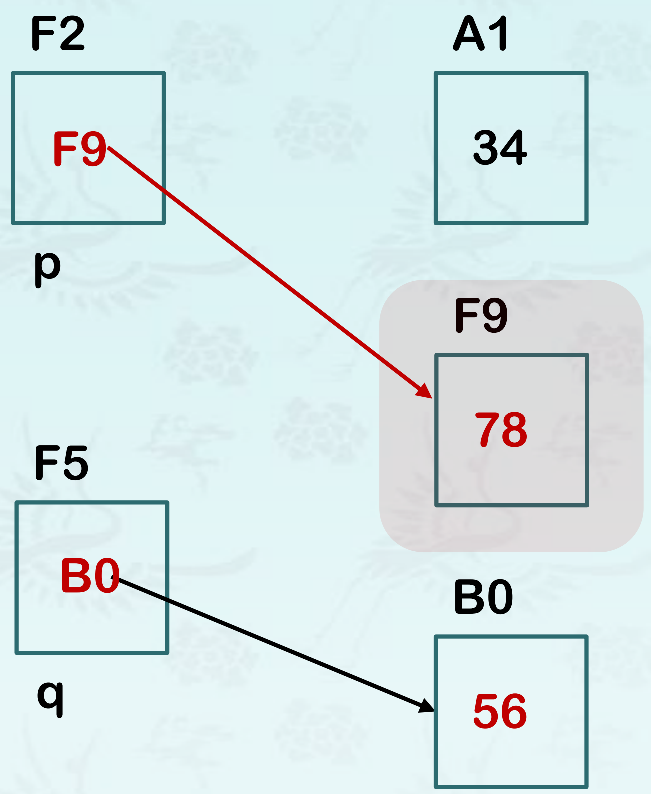
Example 2



Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2

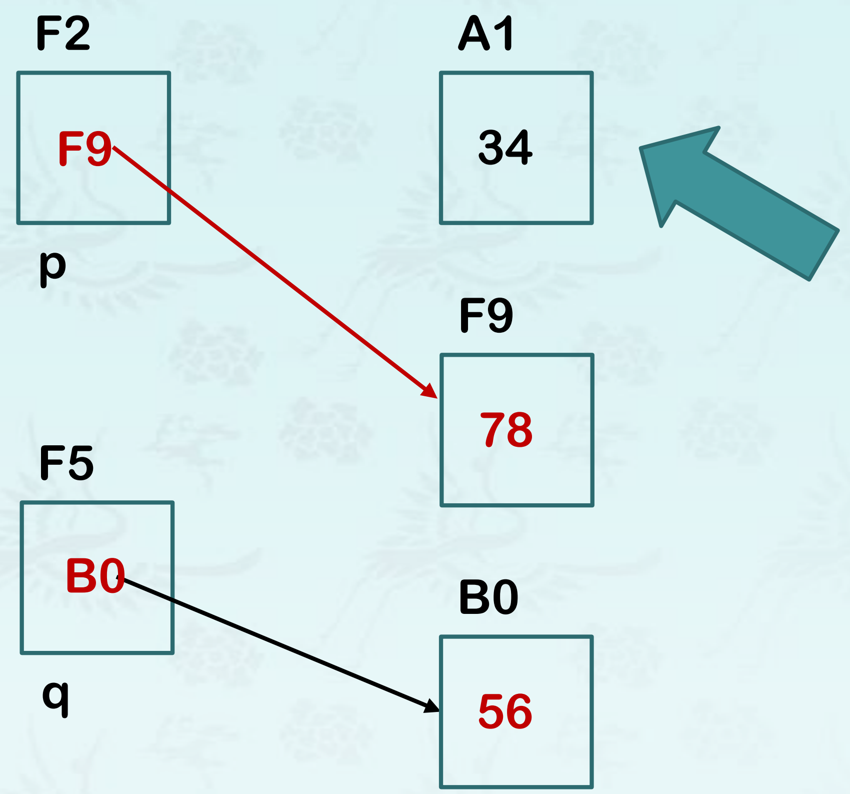


1) What do you observe in result?

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2

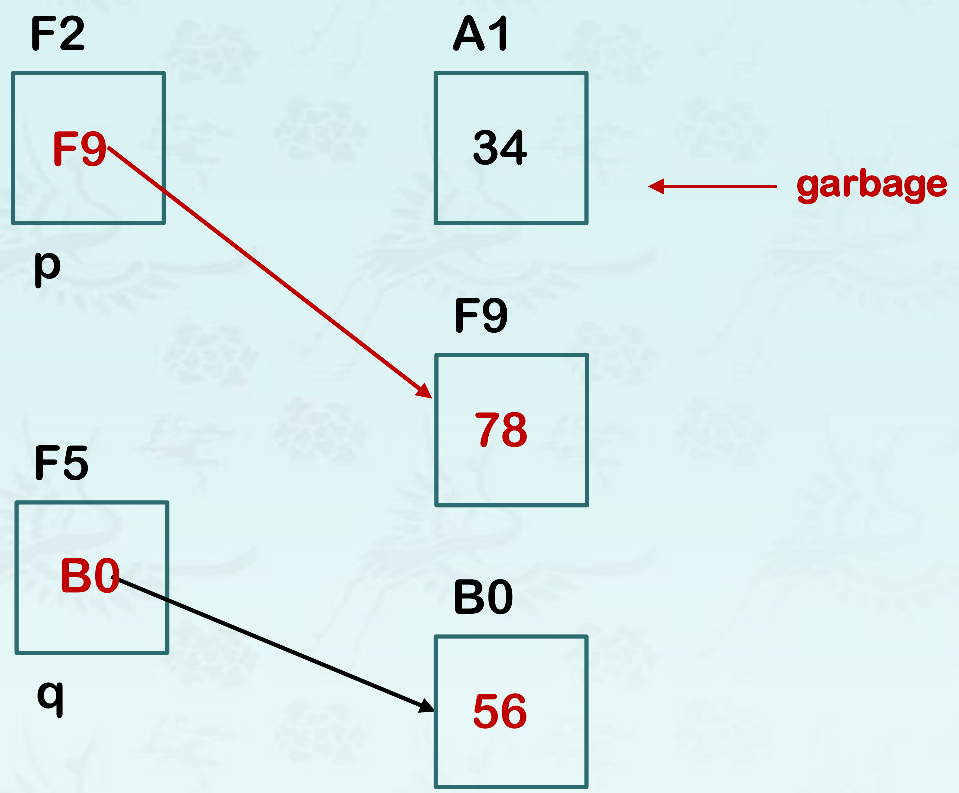


1) What do you observe in result?
Unfortunately by moving `p` from `34` to `78` we have no way of getting back hold of `34`.

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2



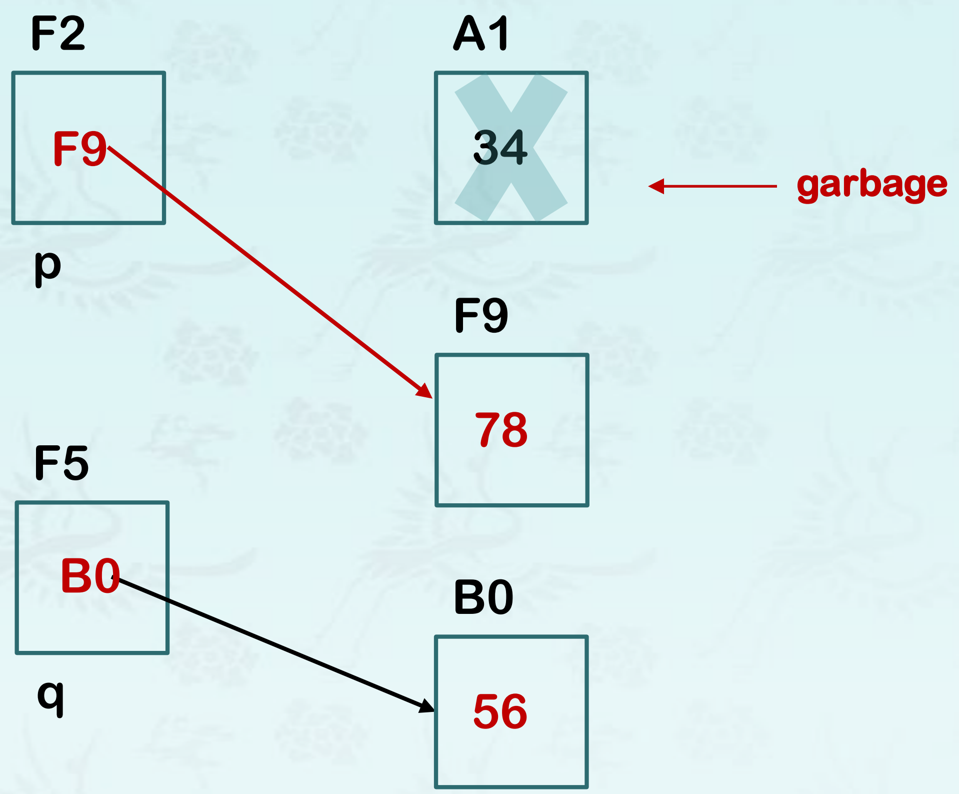
It is now floating in memory, taking up space which we can't re-use. This effect is known as **memory leakage**, and the piece of memory containing the `34` is known as **garbage**. The runtime systems of some languages have garbage collection built in, but C++ doesn't and you have to be careful. If you leak too much memory, the system will run out of RAM and crash!

Pointer reviewed – Quiz

```
int* p = new int(25);
cout << *p << endl;
int* q = p;
cout << *q;
*q = 34;
q = new int(56);
p = new int(78);
delete p;
delete q;
```

Example 2

delete p;

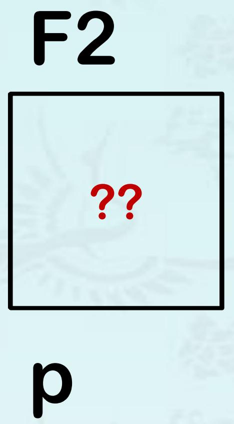


Before you let go of the object you are pointing at, you have to `**delete**` it.

Pointer Linked

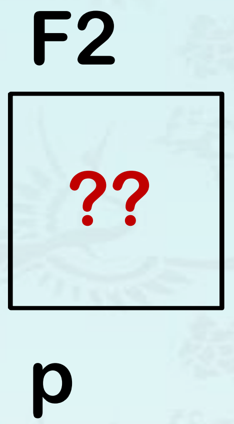
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  ...

}
```



Pointer Linked

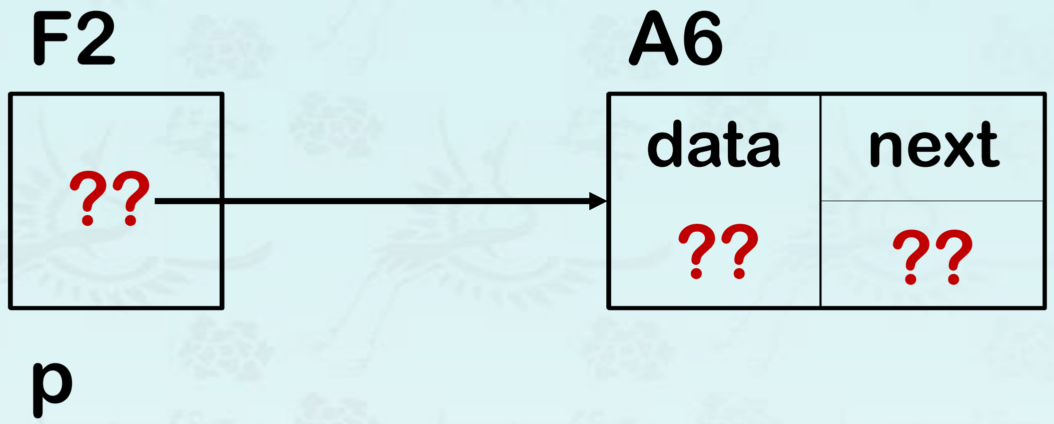
```
class Node {
public:
  int   data;
  Node* next;
};

int main( ) {
  Node* p = new Node;
  ...

}
```



(1) This code declares a **Node pointer**, **p**, and
(2) allocate memory space for a **new Node** and
(3) make **p point at** it.

Pointer Linked

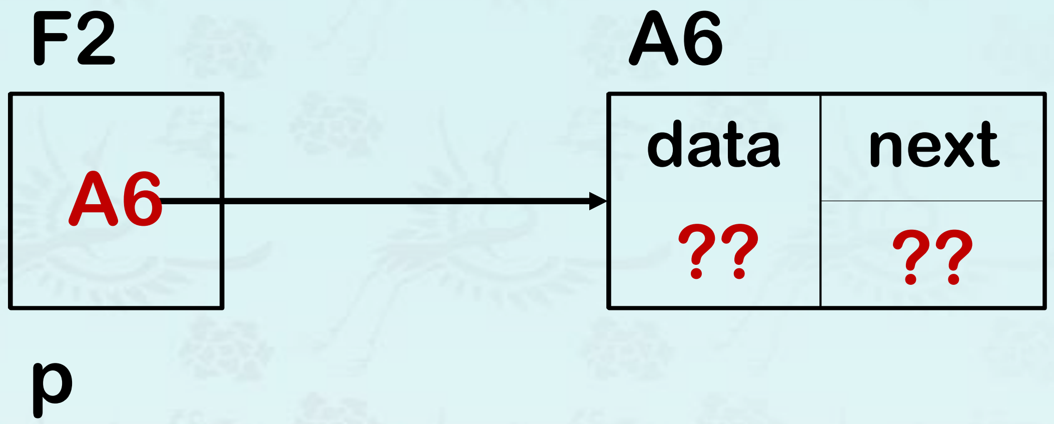
```
class Node {
public:
  int   data;
  Node* next;
};

int main( ) {
  Node* p = new Node;
  ...

}
```



(1) This code declares a **Node pointer, p**, and
(2) allocate memory space for a **new Node** and
(3) make **p point at** it.

Pointer Linked

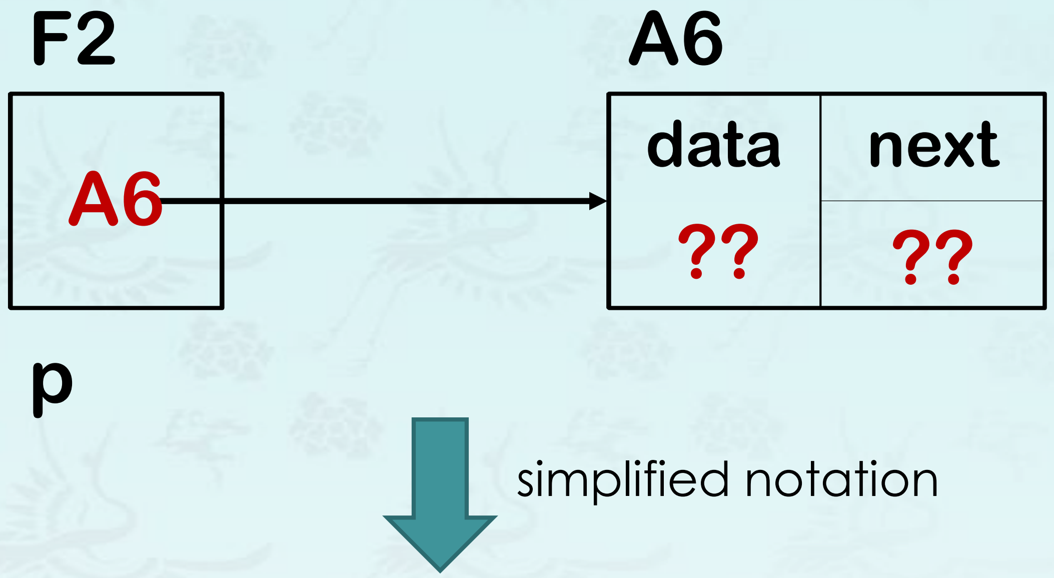
```
class Node {
public:
  int   data;
  Node* next;
};

int main( ) {
  Node* p = new Node;
  ...

}
```



(1) This code declares a **Node pointer, p**, and
(2) allocate memory space for a **new Node** and
(3) make **p point at** it.

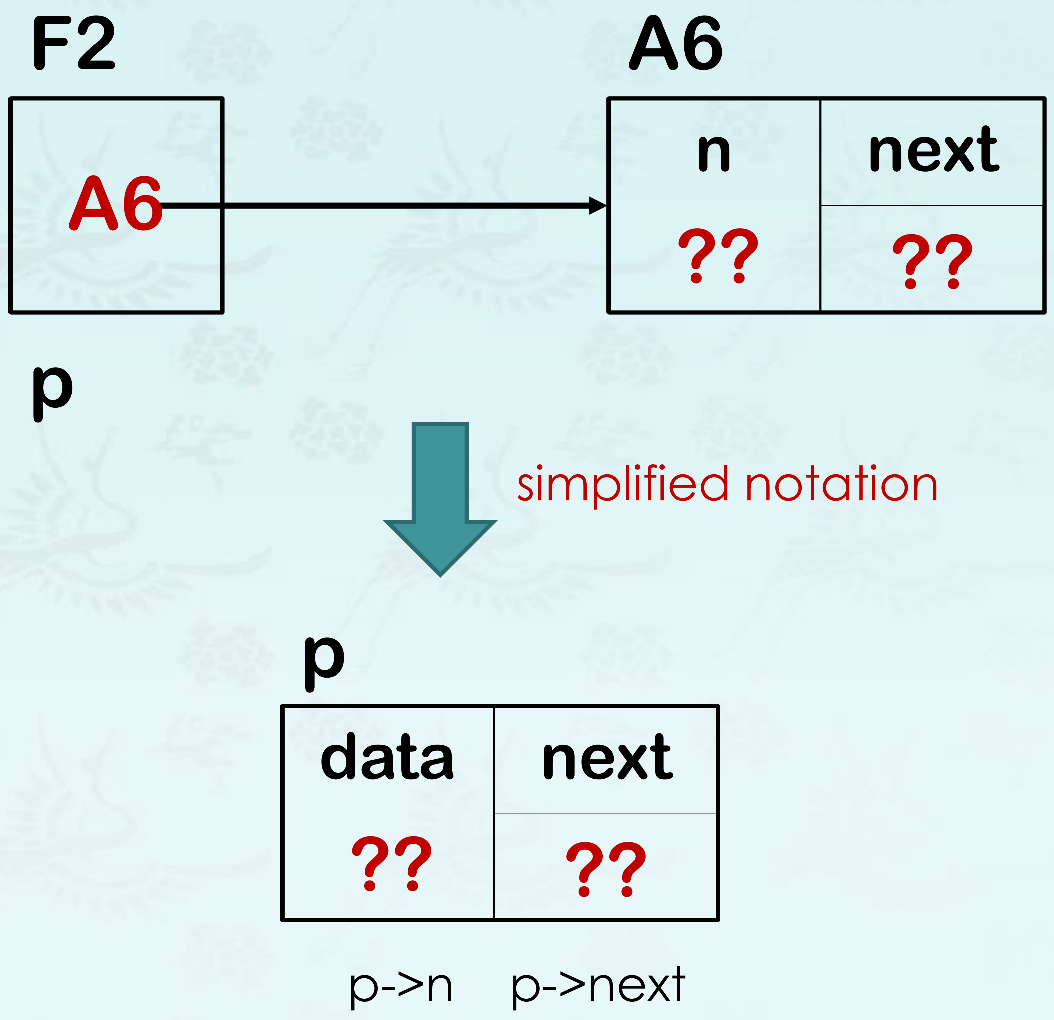
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



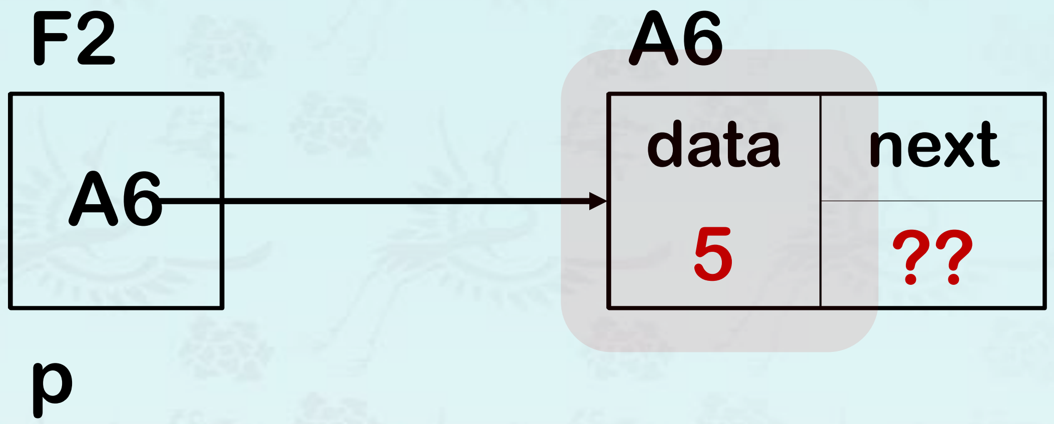
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



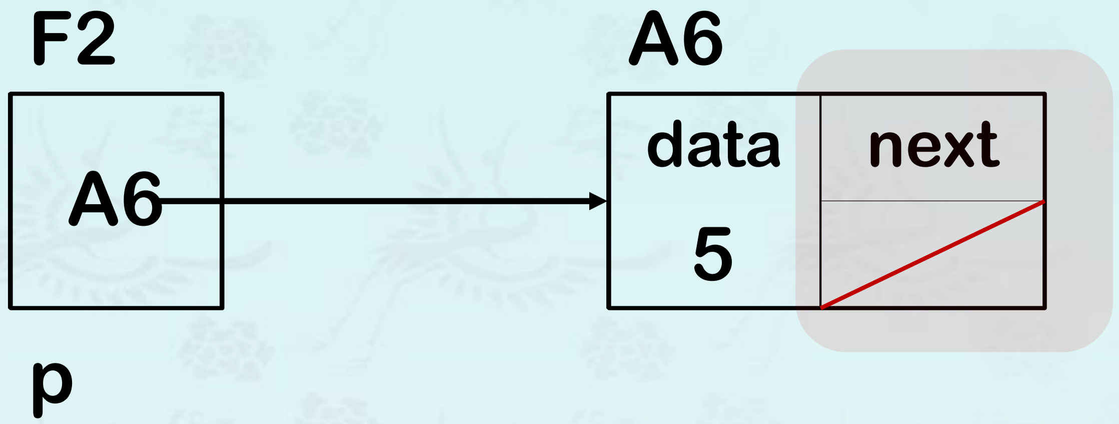
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



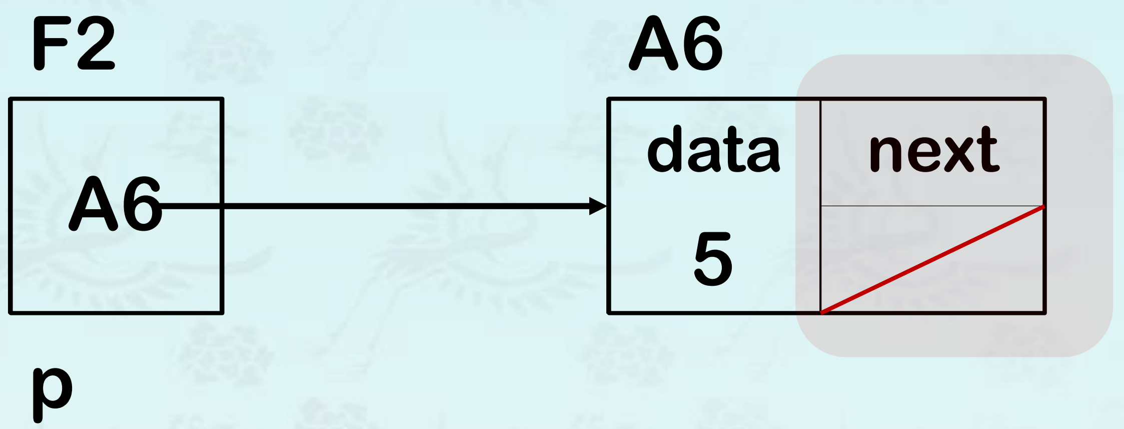
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



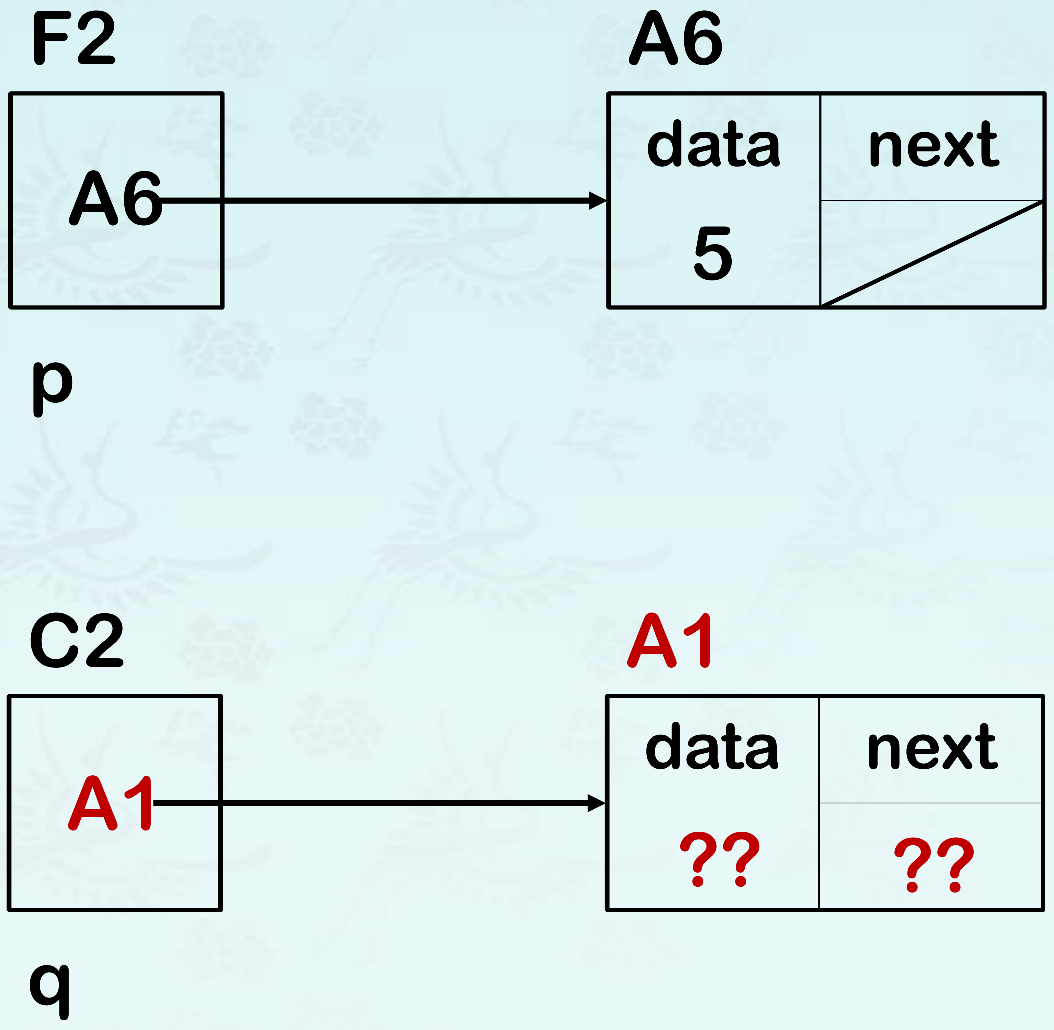
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



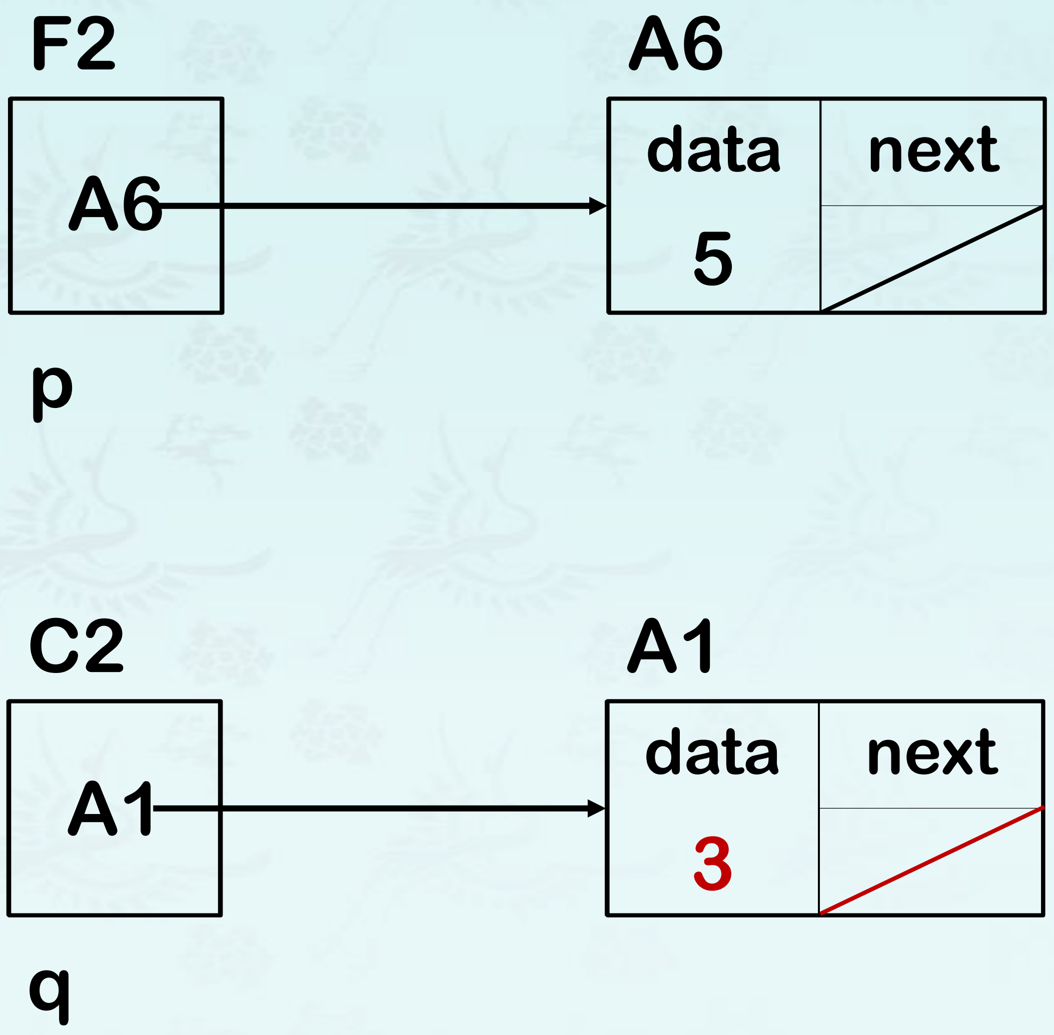
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



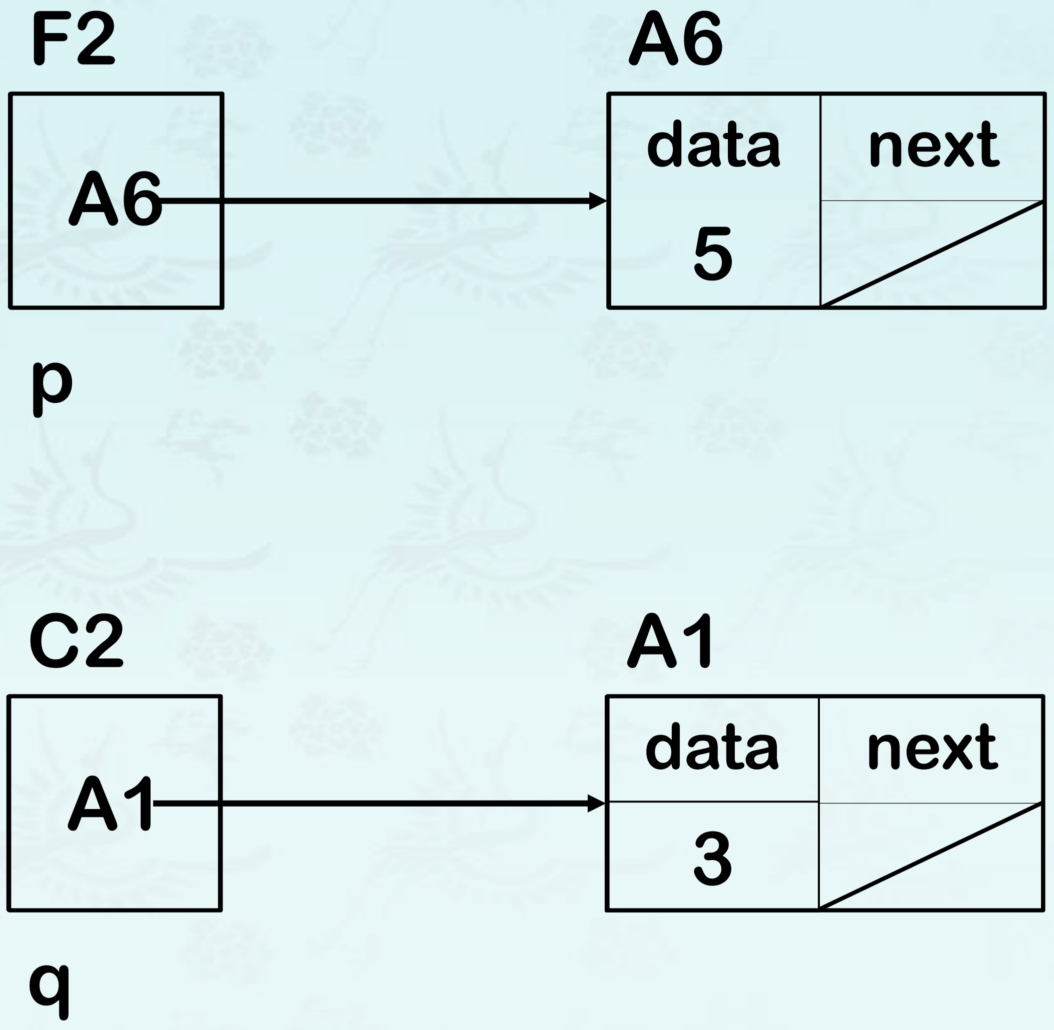
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



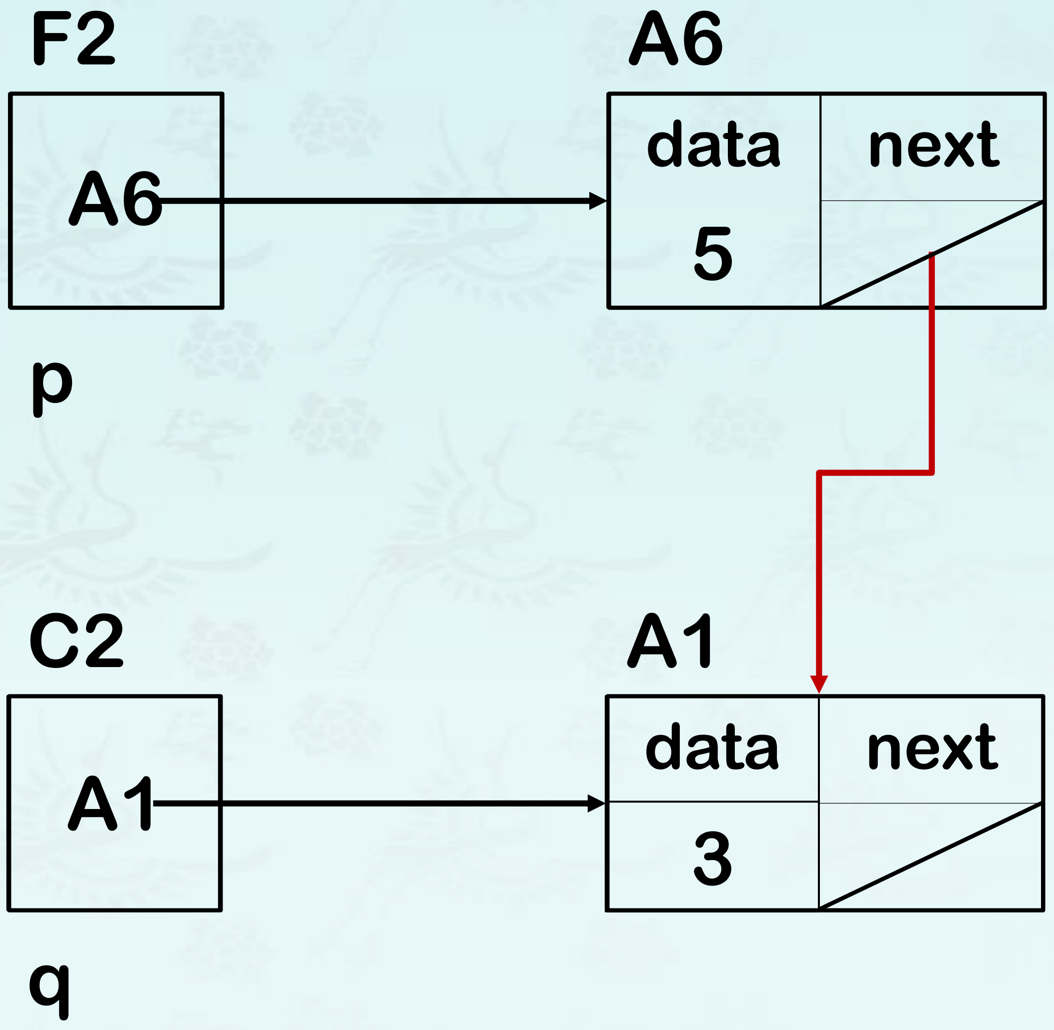
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



Pointer Linked

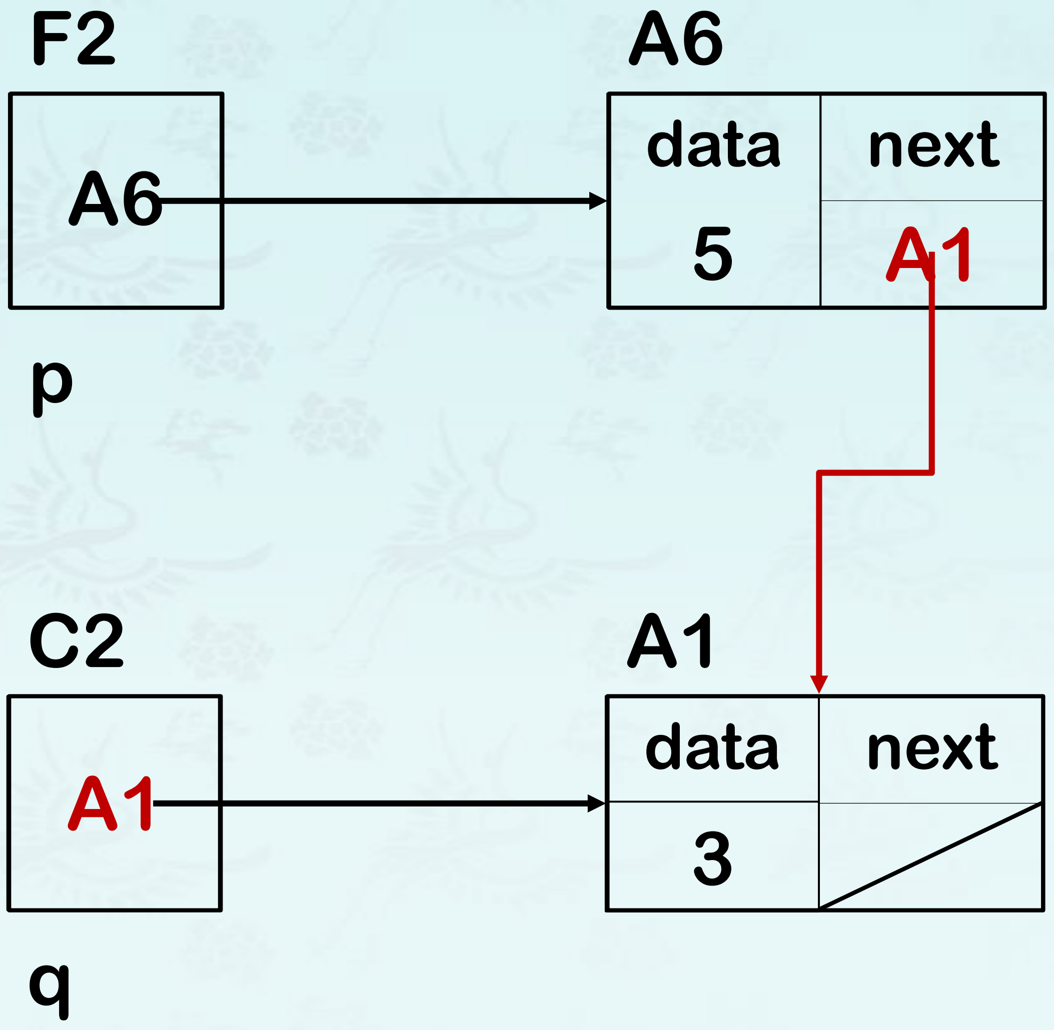
```
class Node {
public:
  int   data;
  Node* next;
};

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



What should be corrected in the figure?

Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```

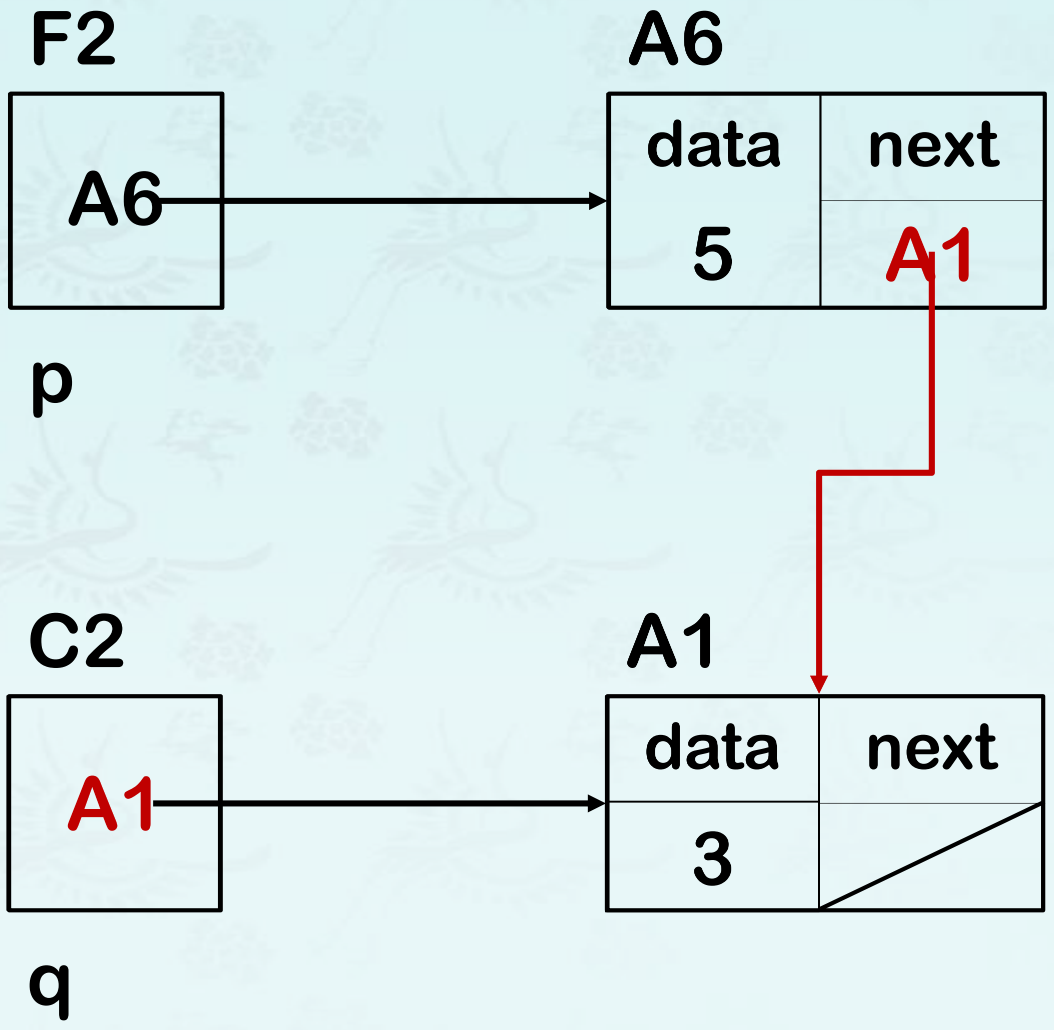


What should be corrected in the figure?

`p->next = A1;`

Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```



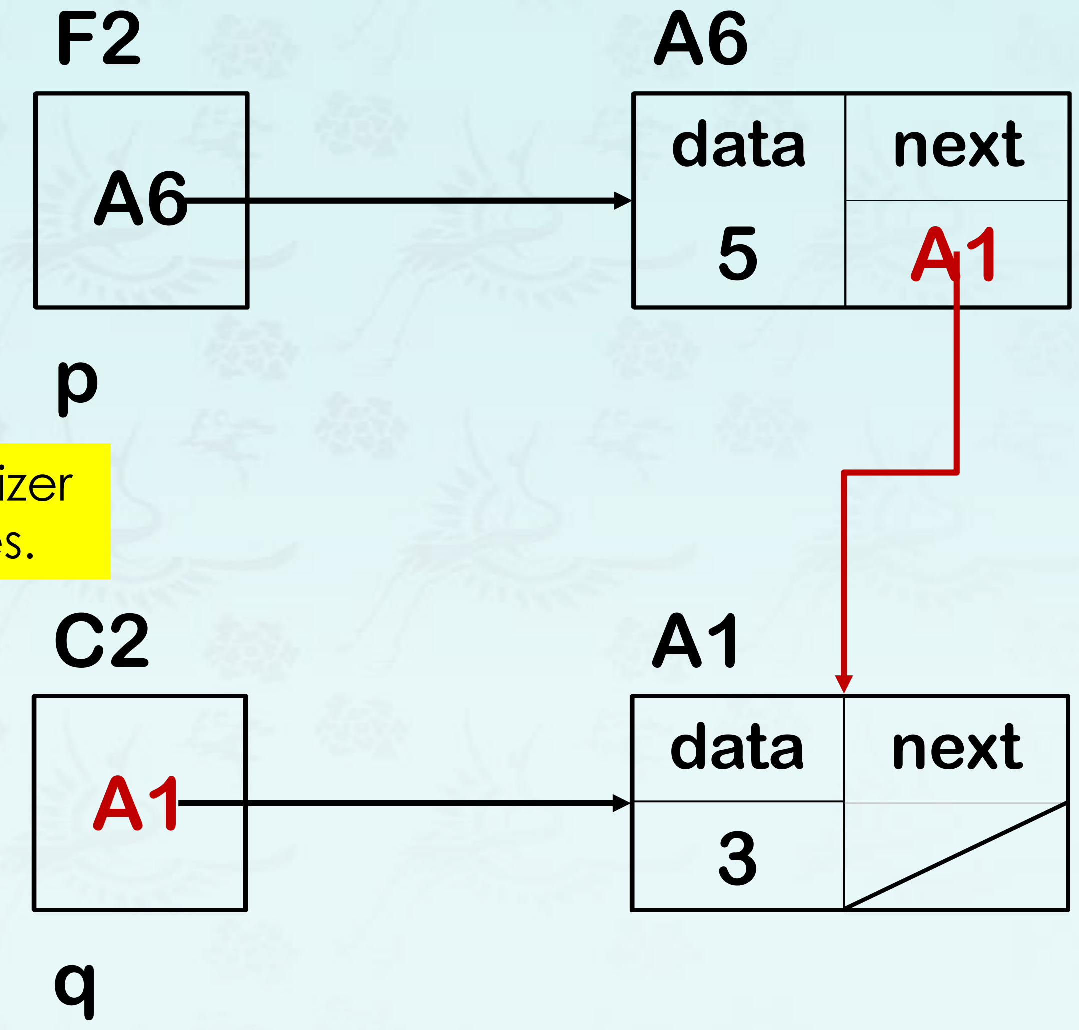
What should be corrected in the figure?

`p->next = A1;        p->next = q;`

Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```

Recode this using initializer (constructor) in two lines.

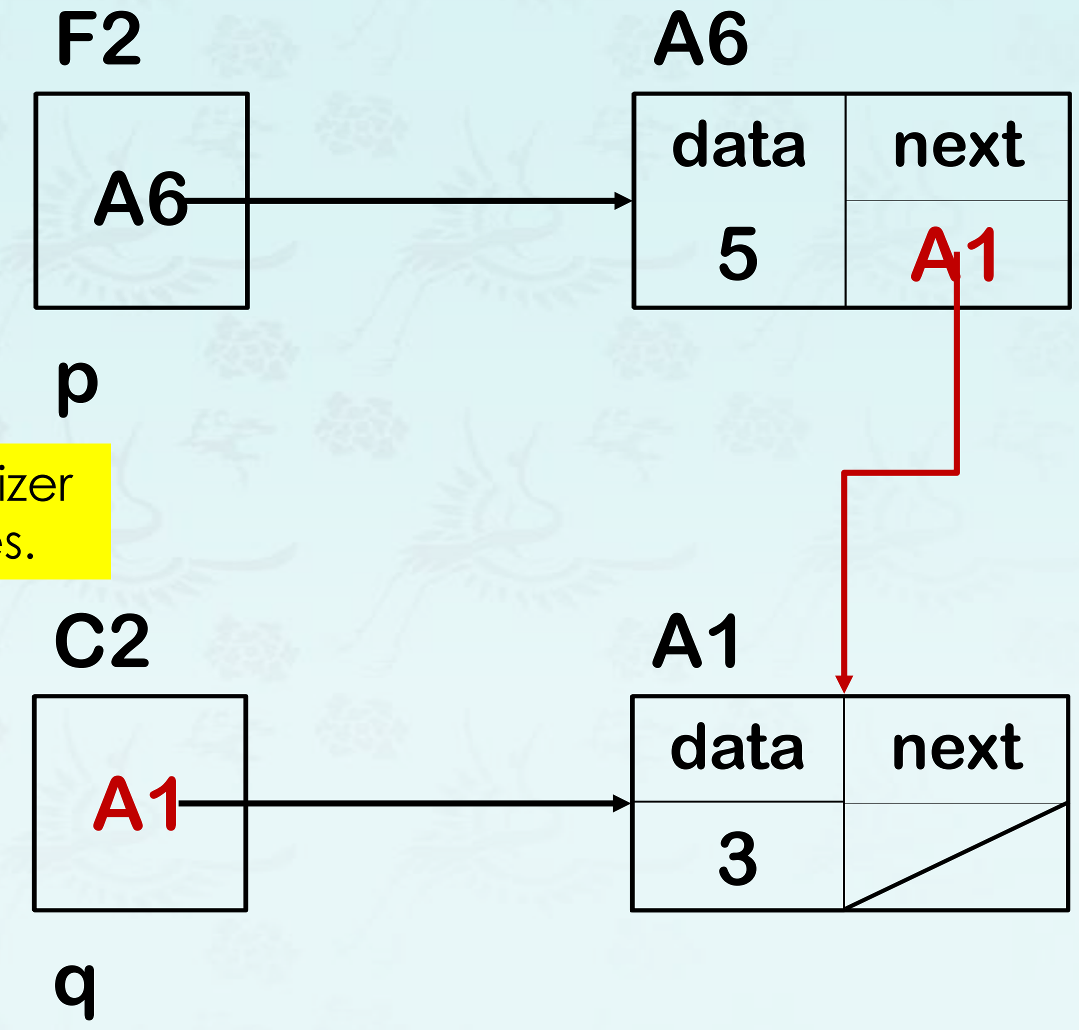


Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```

Recode this using initializer (constructor) in two lines.

```
int main( ) {
  Node* p = new Node{5, nullptr};
  Node* q = new Node{3, nullptr};
  p->next = q;
}
```



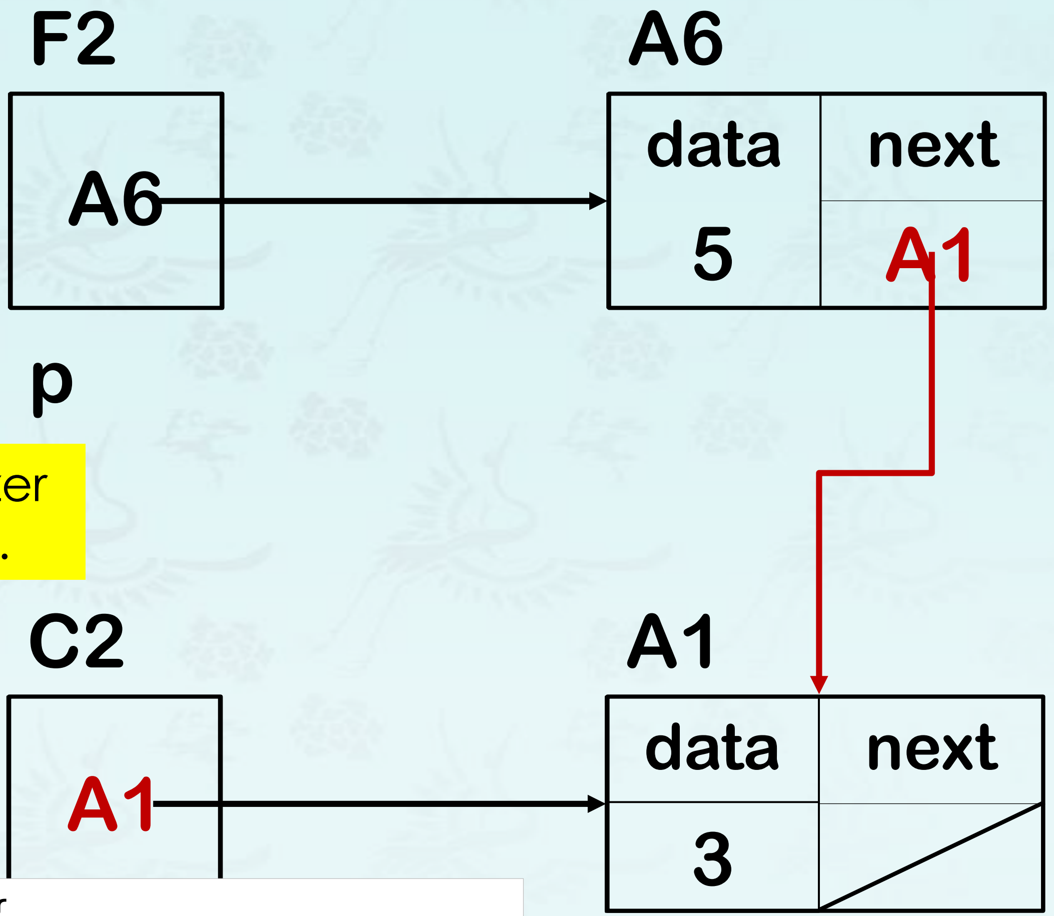
Pointer Linked

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

int main( ) {
  Node* p = new Node;
  p->data = 5;
  p->next = nullptr;
  Node* q = new Node;
  q->data = 3;
  q->next = nullptr;
  p->next = q;
}
```

Recode this using initializer (constructor) in two lines.

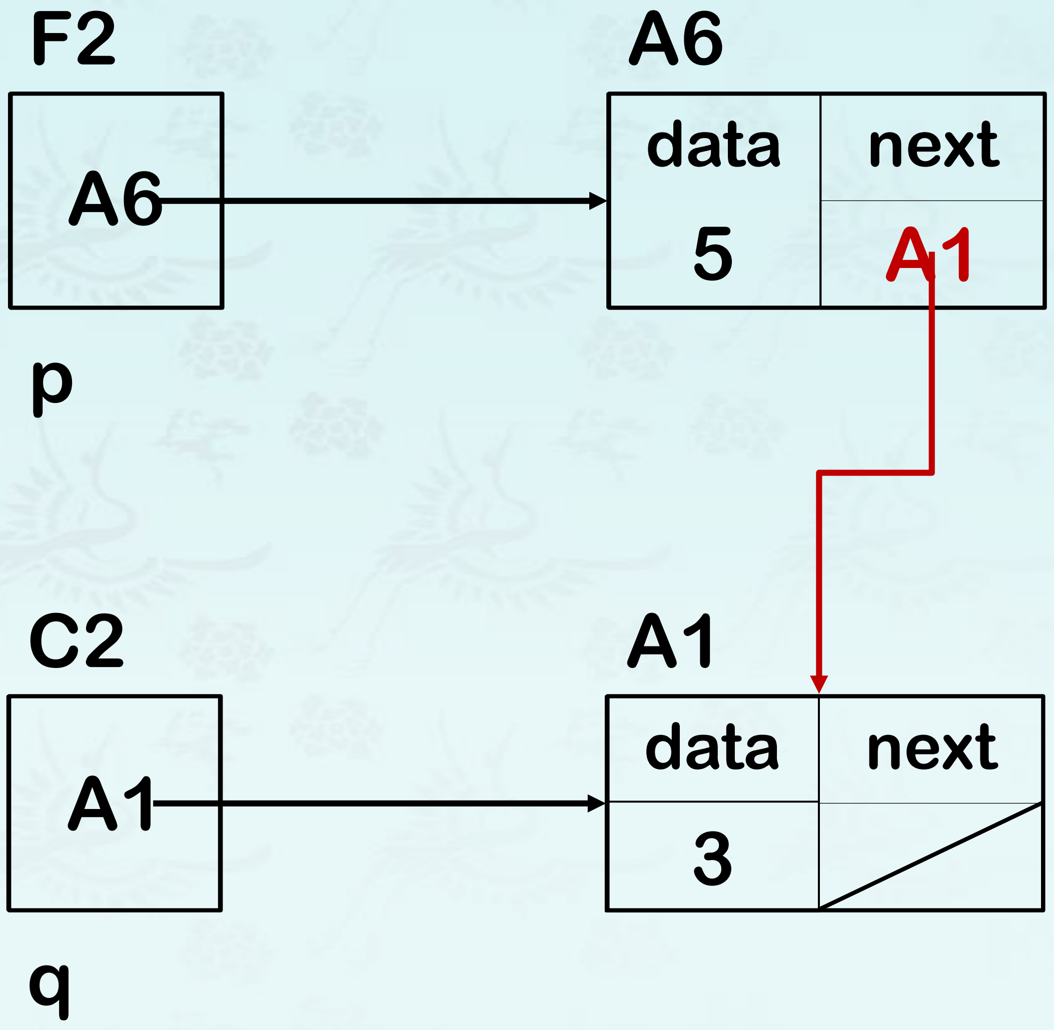
```
int main( ) {
  Node* p = new Node{5, nullptr};
  Node* q = new Node{3, nullptr};
  p->next = q;
}
```

```
int main( ) {
  Node* q = new Node{3, nullptr};
  Node* p = new Node{5, q};
}
```



Pointer Linked

By stringing many of these Node objects together we can create a structure called a **singly-linked list**;

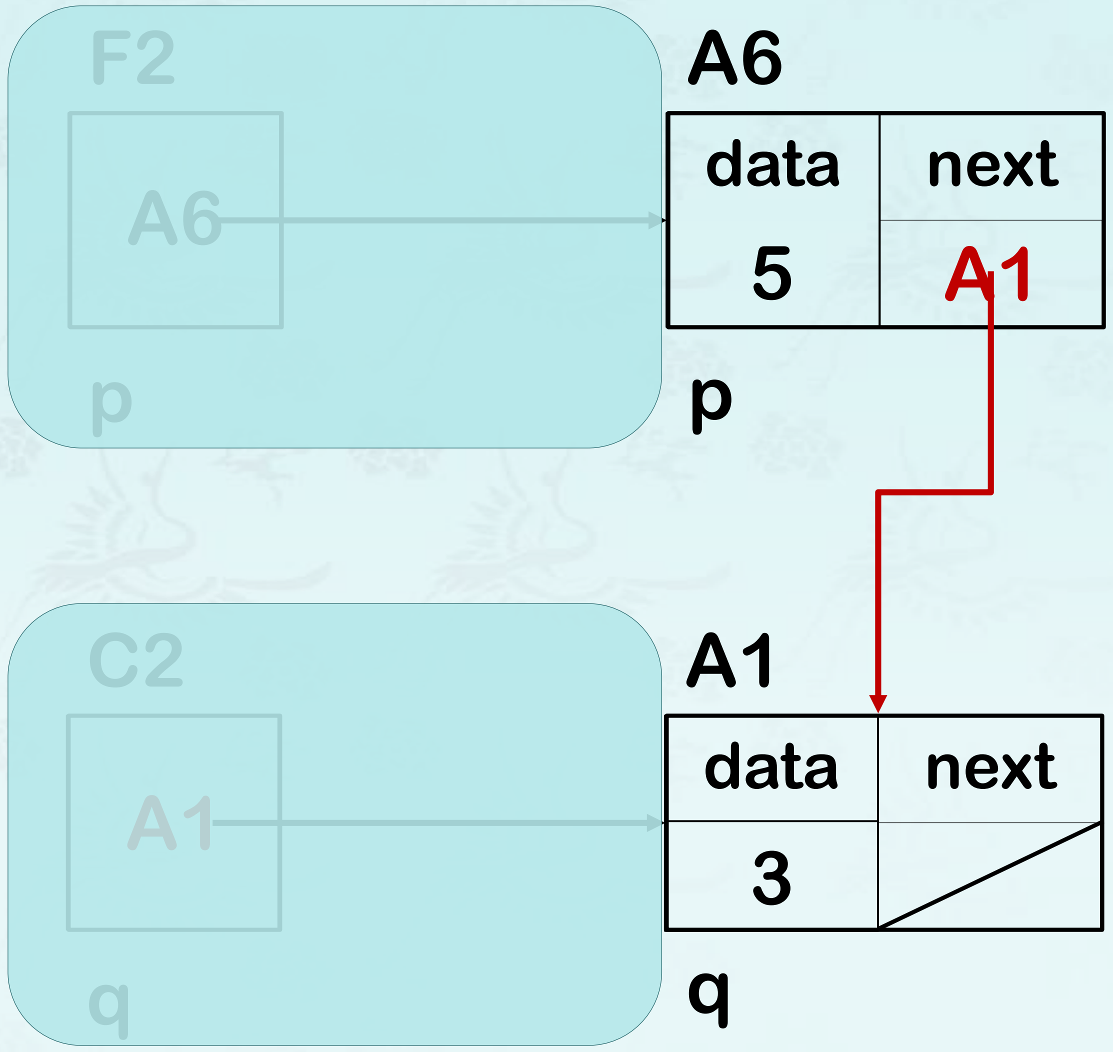


Hide p and q, and you may see a singly linked list clearly.

Pointer Linked

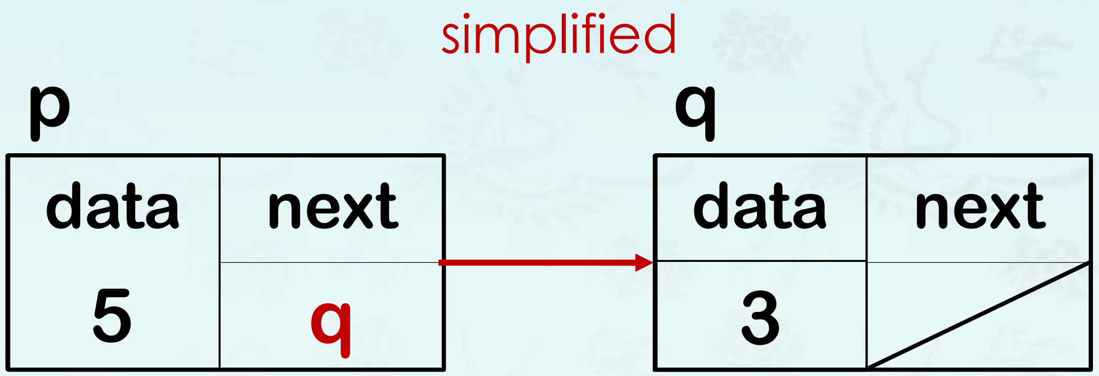
```
class Node {
public:
  int   data;
  Node* next;
};

int main( ) {
  Node* q = new Node{3, nullptr};
  Node* p = new Node{5, q};
}
```

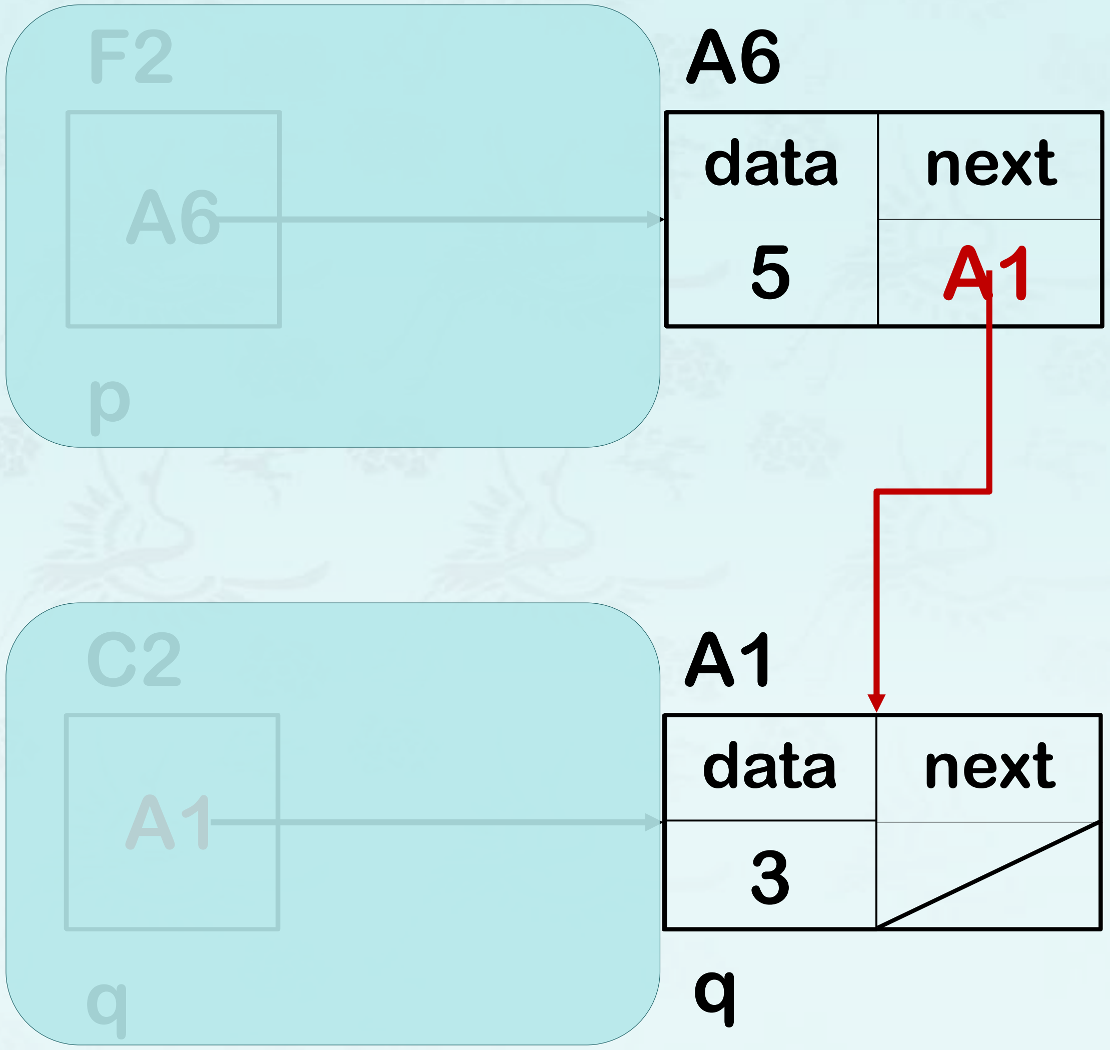
Hide p and q, and you may see a singly linked list clearly.



Pointer Linked

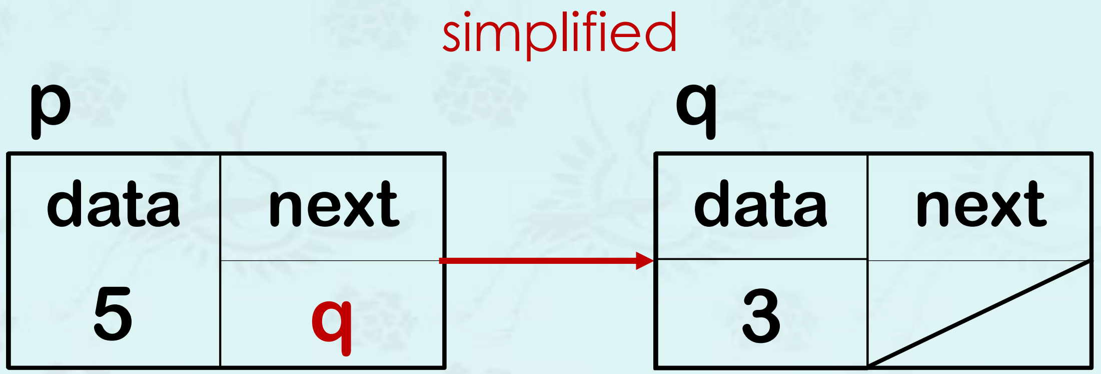
By stringing many of these Node objects together we can create a structure called a **singly-linked list;**





Pointer Linked

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

int main( ) {
  Node* q = new Node{3, nullptr};
  Node* p = new Node{5, q};
}
```



Pointer Linked – Quiz and Lab

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

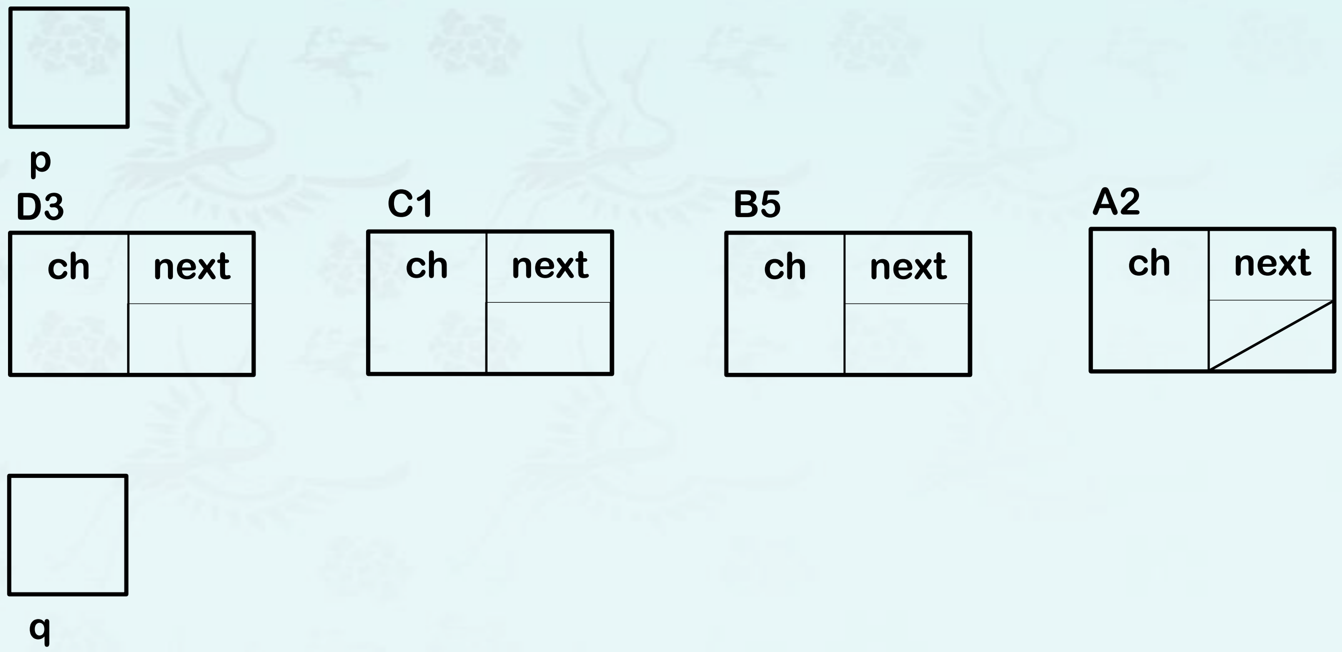
Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.

Pointer Linked – Lab

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



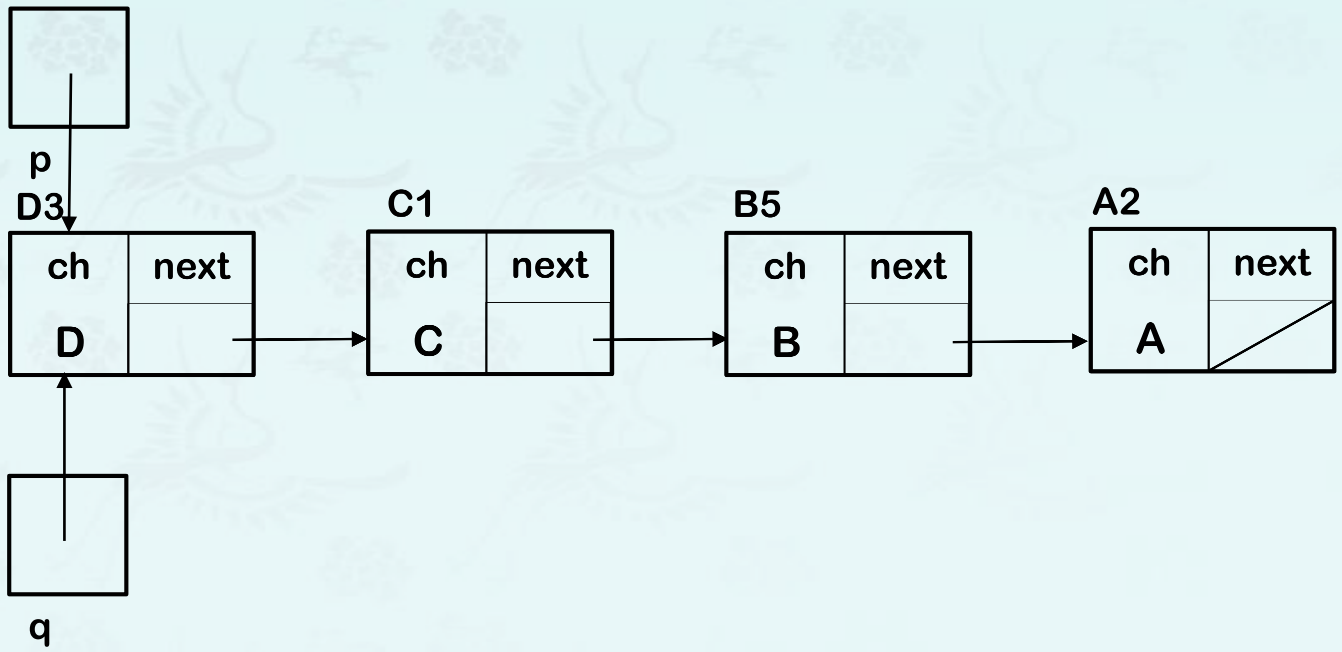
What is missing in the figure?

Pointer Linked – Lab

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



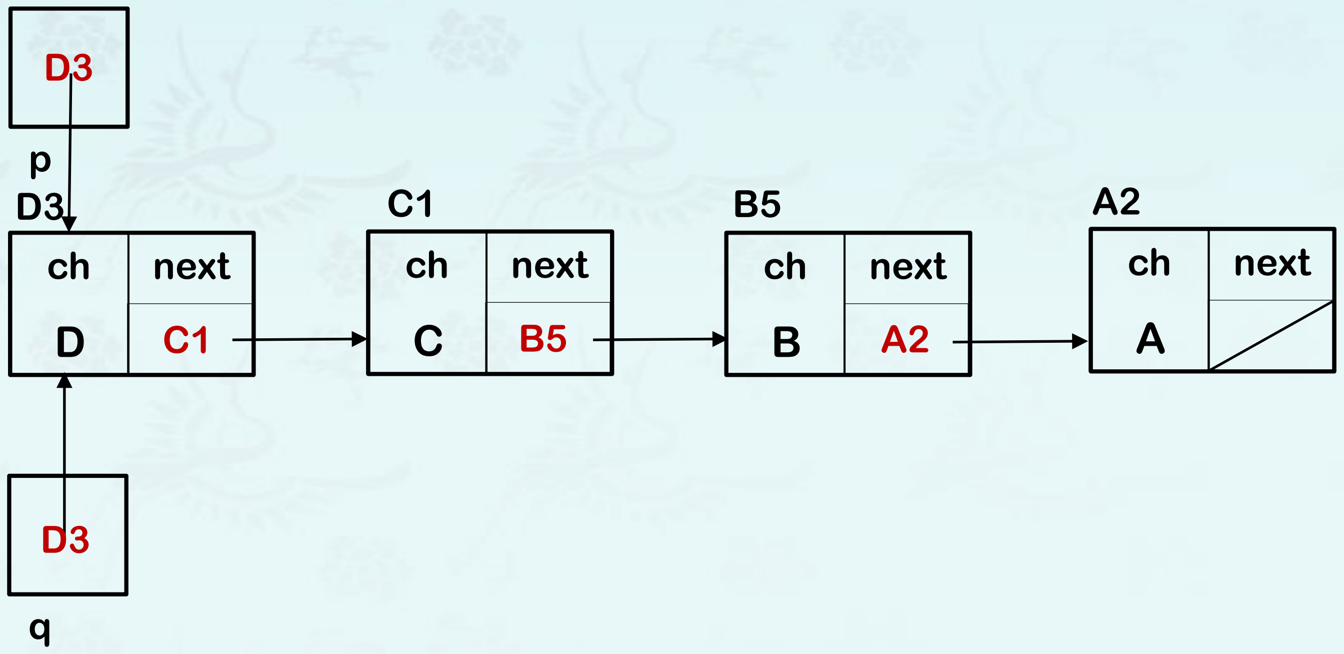
What is missing in the figure?

Pointer Linked – Lab

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

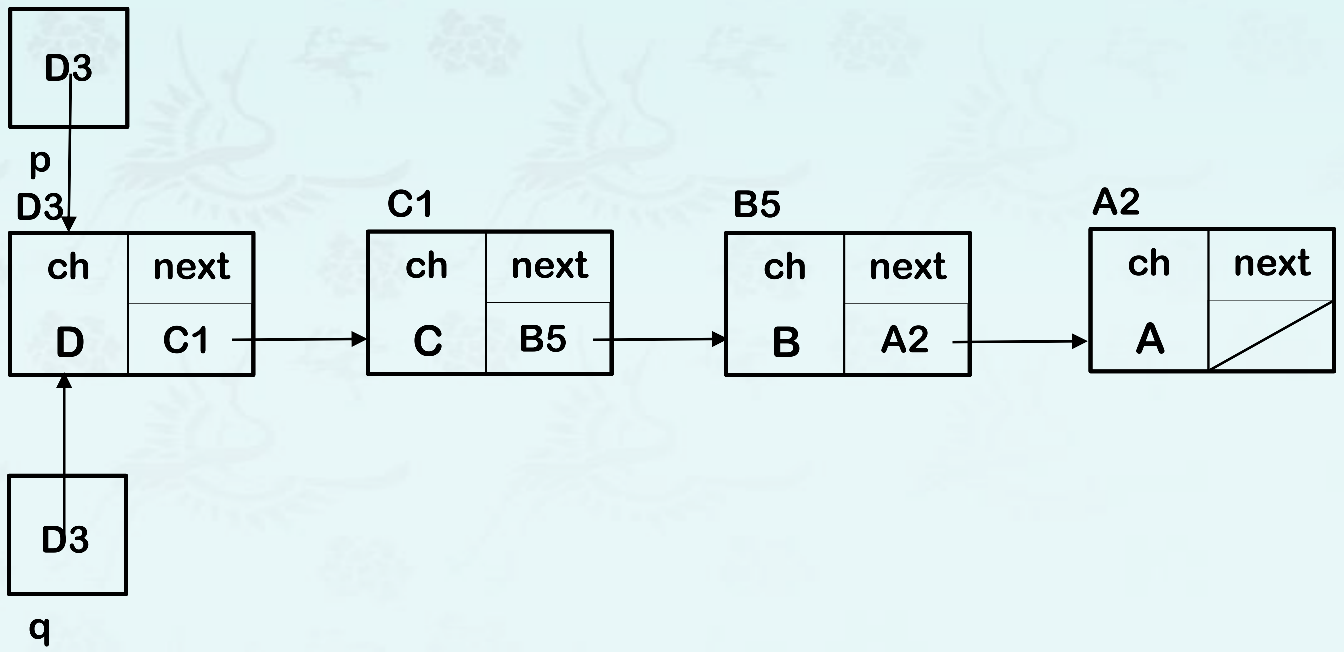
Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



Pointer Linked – Quiz

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

After executing the while loop,
What is the output?
What is the values of p and q?

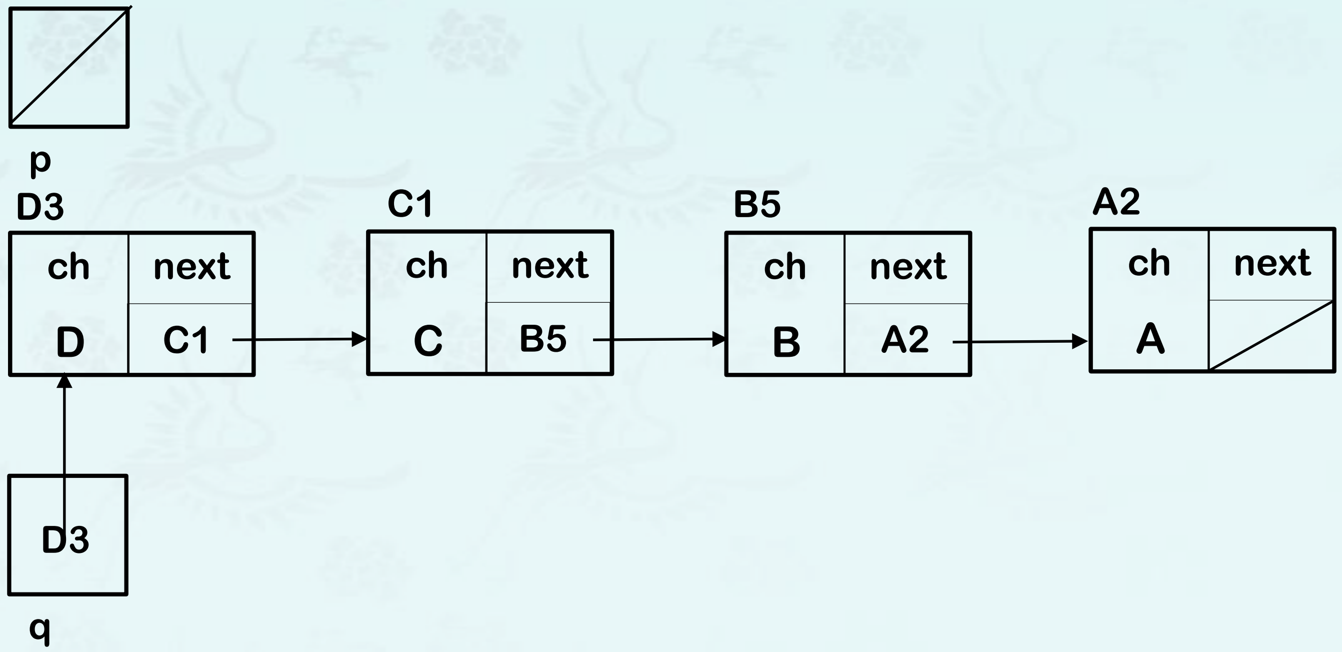


Pointer Linked – Quiz

```
#include <iostream>
using namespace std;
class Node {
public:
  char ch;
  Node* next;
};

int main( ) {
  Node* p = nullptr, *q = nullptr;
  char ch;
  while (cin.get(ch) && ch != '\n') {
    p = new Node;
    p->ch = ch;
    p->next = q;
    q = p;
  }
  while (p != nullptr) {
    cout.put(p->ch);
    p = p->next;
  }
  cout << endl;
}
```

If you run the code shown below at the end, what would be output?

```
cout << q->next->ch;
```

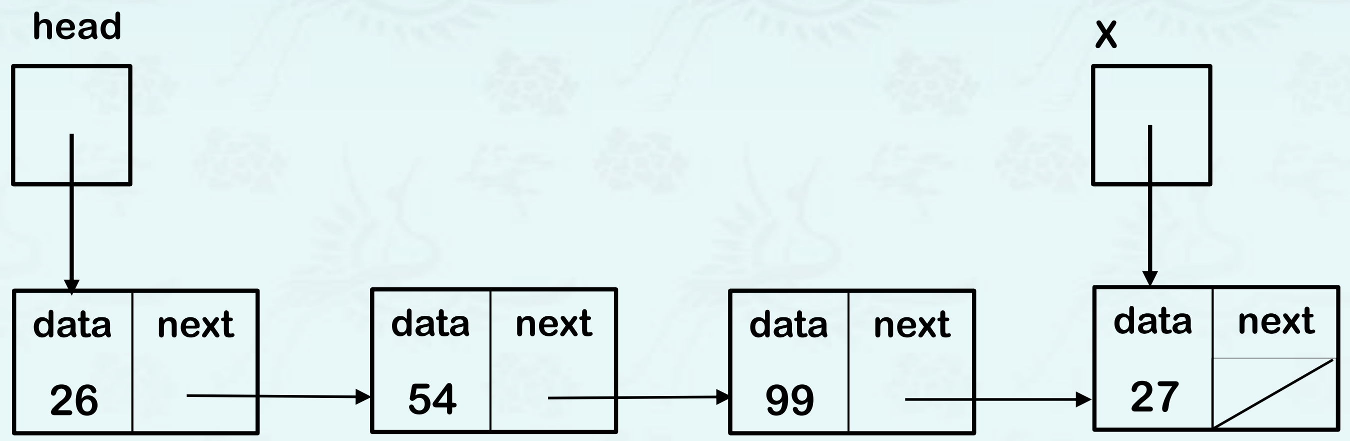


Dynamic Data Structures

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

...

Node* head, *x, *y;
```

basic member functions

- push_front()
- push_back()
- pop_front()
- pop_back()
- insert()
- remove()
- clear()

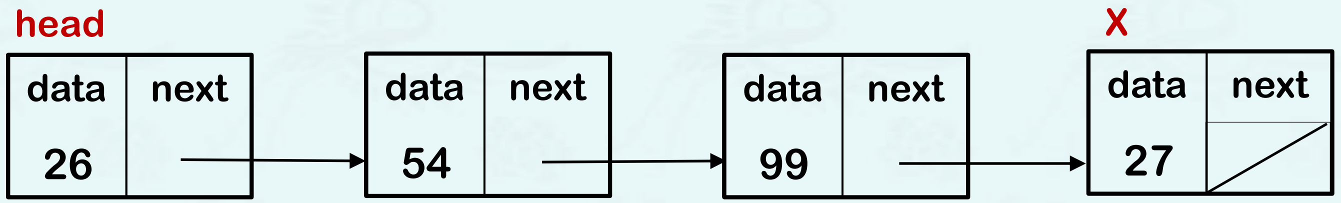


Dynamic Data Structures

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


...

Node* head, *x, *y;
```

basic member functions

- push_front()
- push_back()
- pop_front()
- pop_back()
- insert()
- remove()
- clear()

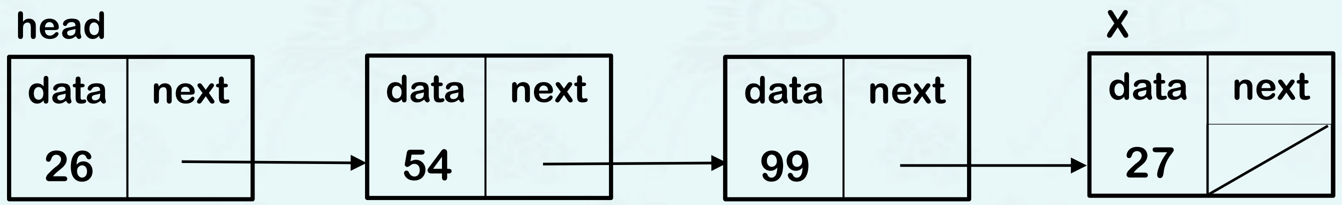


Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 10) at the head of list.



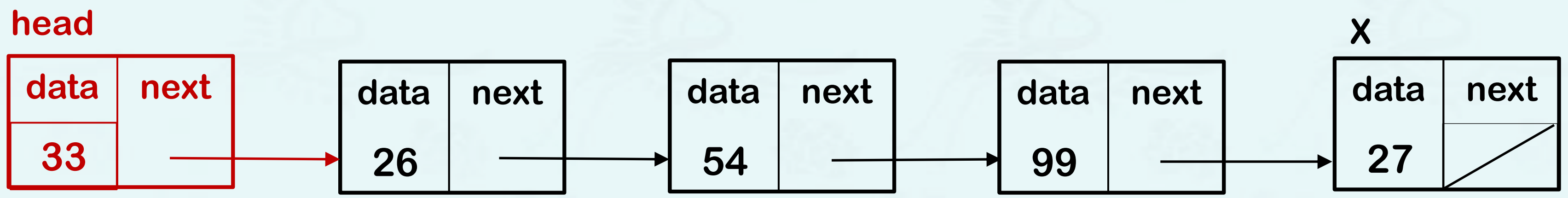
Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next =
```



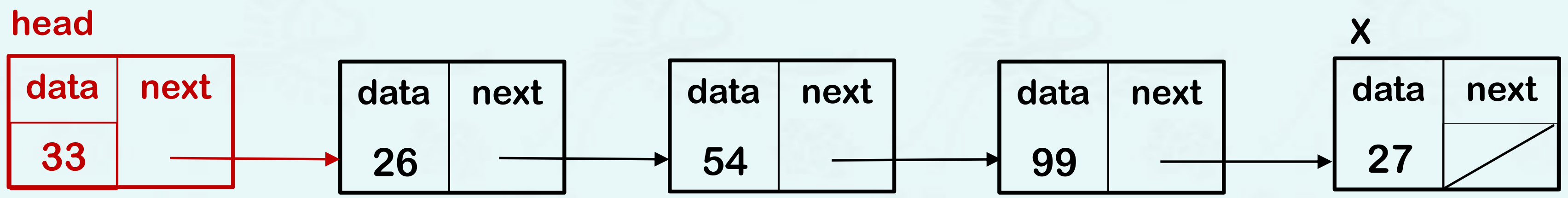
Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
```



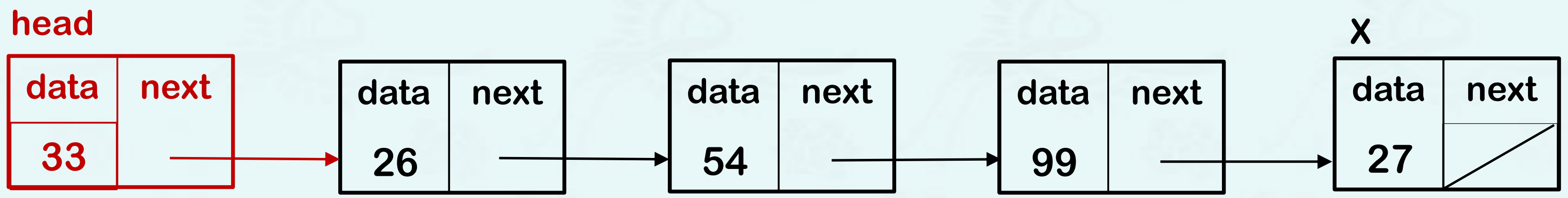
Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
head =
```



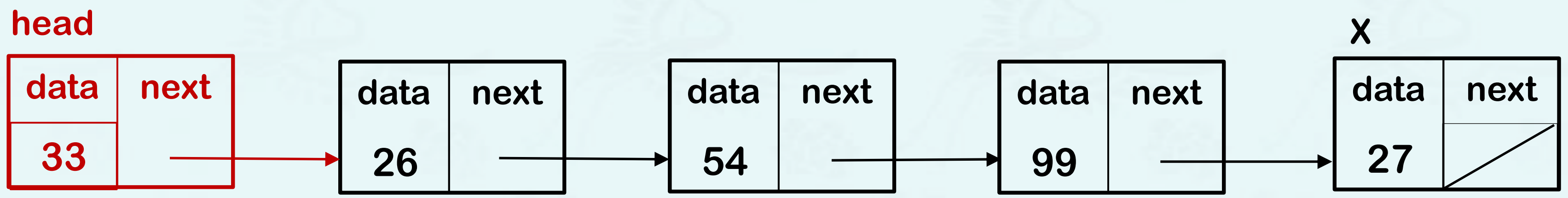
Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
head = y;
```



Dynamic Data Structures – push_front()

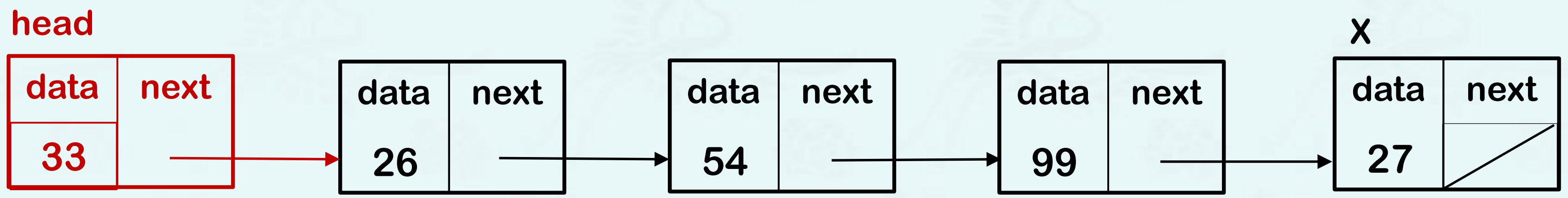
```
class Node {
public:
  int   data;
  Node* next;
};

...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
head = y;
```

```
y = new Node {33, head};
head = y;
```



Dynamic Data Structures – push_front()

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

...

Node* head, *x, *y;
```

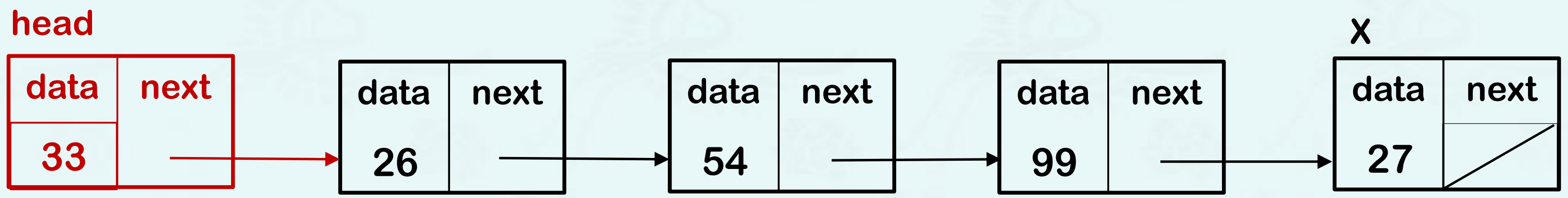
Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
head = y;
```

```
y = new Node {33, head};
head = y;
```

How do you code it in a function, push_front()?



Dynamic Data Structures – push_front()

```
class Node {
public:
  int   data;
  Node* next;
};
...
Node* head, *x, *y;
...
head = push_front(head, 33);
```

```
Node* push_front(Node h, int d) {
  ...
  Node y = new Node{d, h};
  ...
  return y;
}
```

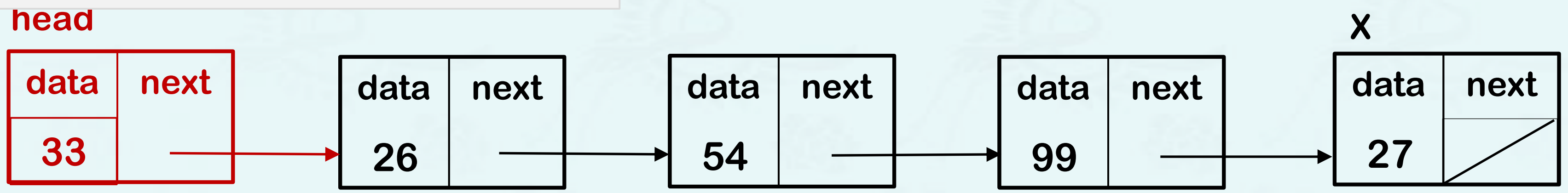
Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the head of list.
  - create a node and initialized with n = 10.
  - let head point to the new node

```
y = new Node;
y->data = 33
y->next = head;
head = y;
```

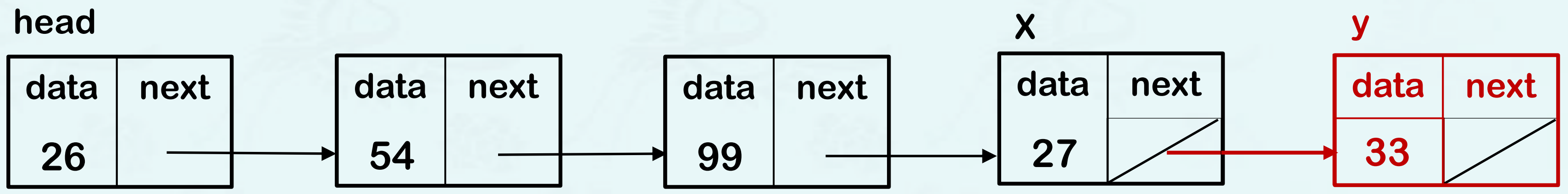
```
y = new Node {33, head};
head = y;
```

How do you code it in a function, push_front()?



Dynamic Data Structures – push_back()

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

...

Node* head, *x, *y;
```



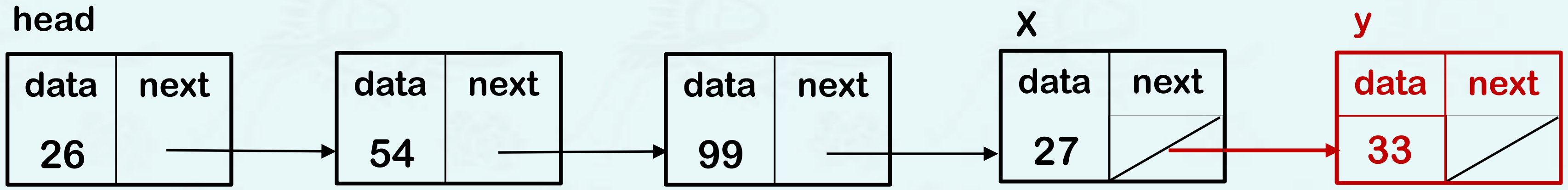
Dynamic Data Structures – push_back()

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


...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the end of list.

```
y = new Node;
```



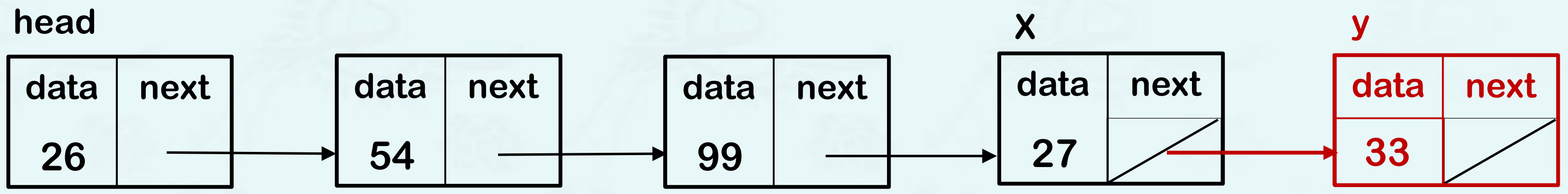
Dynamic Data Structures – push_back()

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


...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the end of list.

```
y = new Node;
y->data = 33
y->next = nullptr;
```



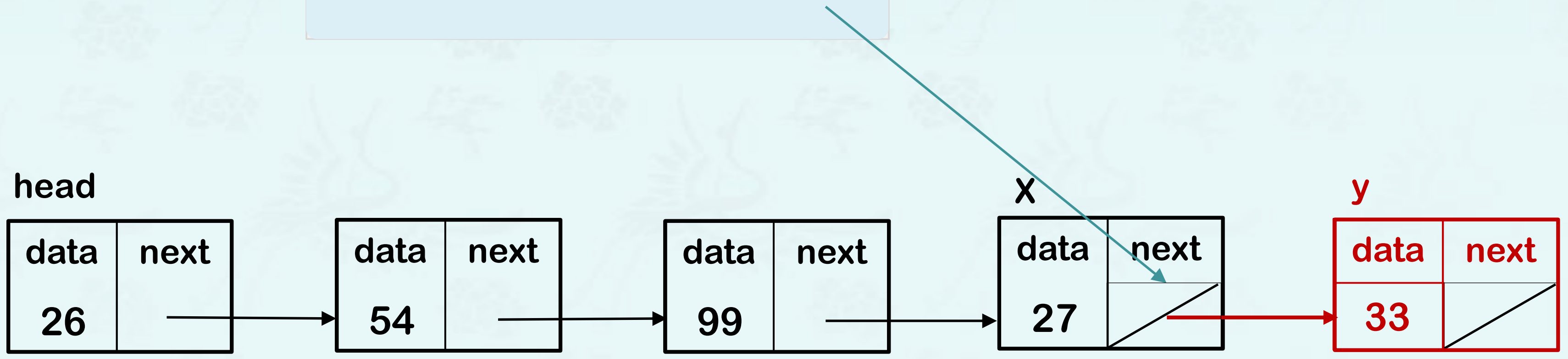
Dynamic Data Structures – push_back()

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


...

Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the end of list.

```
y = new Node;
y->data = 33
y->next = nullptr;
```



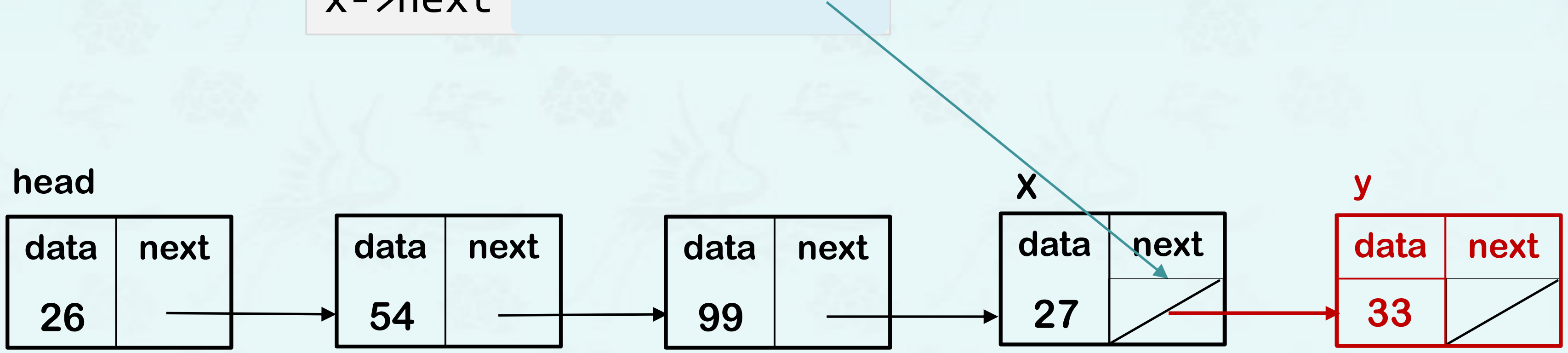
Dynamic Data Structures – push_back()

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


...


Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the end of list.

```
y = new Node;
y->data = 33
y->next = nullptr;
x->next
```



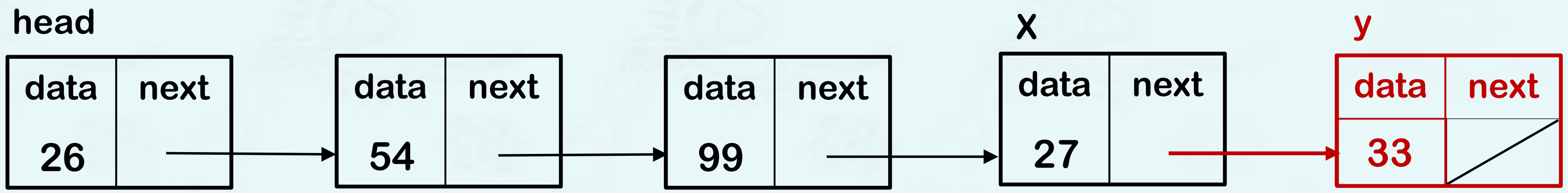
Dynamic Data Structures – push_back()

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


...


Node* head, *x, *y;
```

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (n = 33) at the end of list.

```
y = new Node;
y->data = 33
y->next = nullptr;
x->next = y;
```

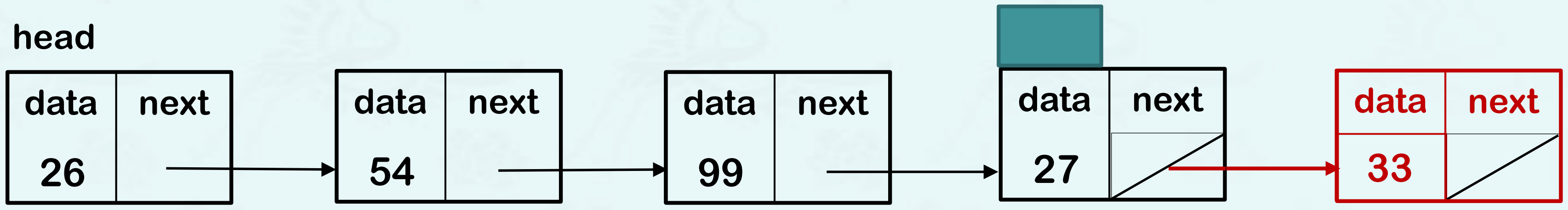


Dynamic Data Structures – push_back()

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

...

Node* head, *x, *y;
```

- Add a node (n = 33) at the end of list, where **only head** of the list is given as shown below.

tail node is not given.

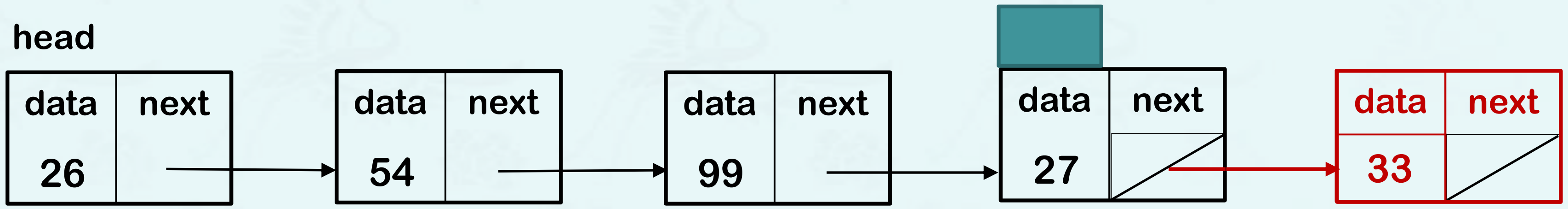


Dynamic Data Structures – push_back()

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

...

Node* head, *x, *y;
```

- Add a node (n = 33) at the end of list, where **only head** of the list is given as shown below.
- To get to the tail we have to scroll along the list until the end. We want a pointer that will stop while still pointing at the last node. Thus our termination condition is that the node's next field is **nullptr**. Once we have a pointer to the end of the list, we can make it point to the node we want to add:

tail node is not given.

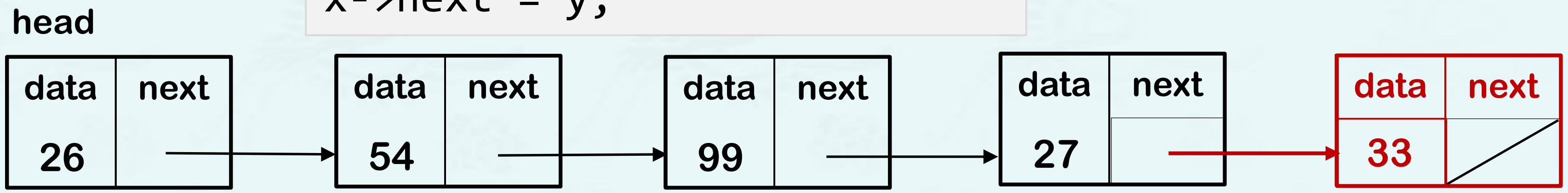


Dynamic Data Structures – push_back()

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

...

Node* head, *x, *y;
```

- Add a node (n = 33) at the end of list, where **only head** of the list is given as shown below.
- To get to the tail we have to scroll along the list until the end. We want a pointer that will stop while still pointing at the last node. Thus our termination condition is that the node's next field is **nullptr**. Once we have a pointer to the end of the list, we can make it point to the node we want to add:

```
y = new Node;
y->data = 33
y->next = nullptr;
x = head;
while (x->next != nullptr)
  x = x->next;
x->next = y;
```



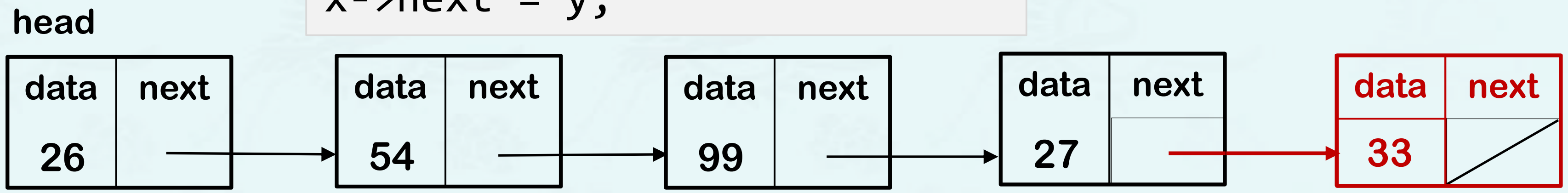
Dynamic Data Structures – push_back()

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

...

Node* head, *x, *y;
```

- Add a node (n = 33) at the end of list, where **only head** of the list is given as shown below.
- To get to the tail we have to scroll along the list until the end. We want a pointer that will stop while still pointing at the last node. Thus our termination condition is that the node's next field is **nullptr**. Once we have a pointer to the end of the list, we can make it point to the node we want to add:

```
y = new Node;
y->data = 33
y->next = nullptr;
x = head;
while (x->next != nullptr)
  x = x->next;
x->next = y;
```

a way to get the last node

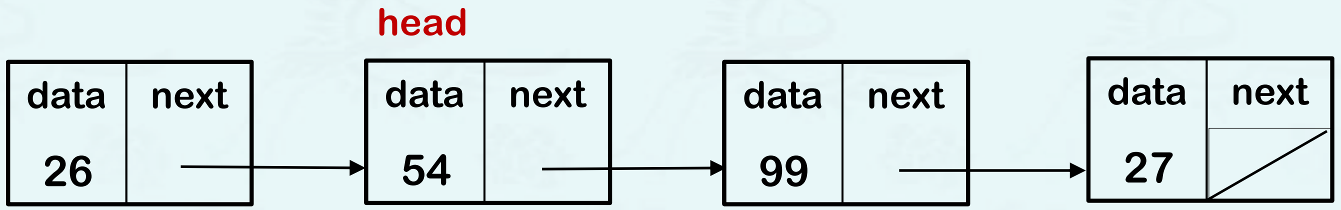


Dynamic Data Structures – pop_front()

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

...

Node* head, *x, *y;
```

- Remove the first node or move head to the next node. Then what is wrong with the following code?

```
head = head->next;
```

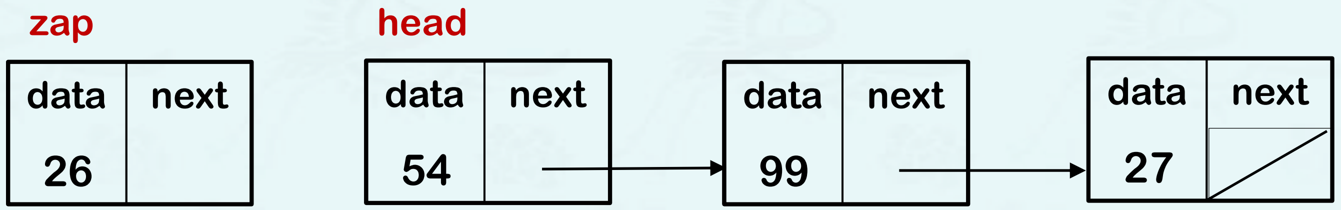


Dynamic Data Structures – pop_front()

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

...

Node* head, *x, *y;
```

- Remove the first node or move head to the next node.
  Then what is wrong with the following code?
- When removing a node, beware of memory leak; remember to give yourself a pointer to the node that is about to be removed before you lose your pointer to it:

```
Node* zap = head;
head = head->next;
```

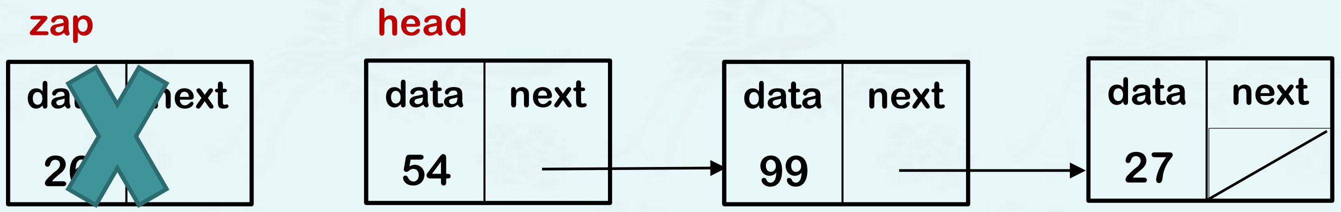


Dynamic Data Structures – pop_front()

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

...

Node* head, *x, *y;
```

- Remove the first node or move head to the next node. Then what is wrong with the following code?
- When removing a node, beware of memory leak; remember to give yourself a pointer to the node that is about to be removed before you lose your pointer to it:

```
Node* zap = head;
head = head->next;
```

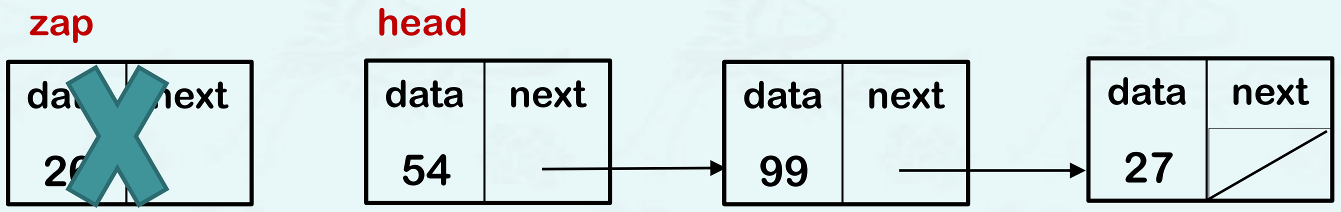


Dynamic Data Structures – pop_front()

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

...

Node* head, *x, *y;
```

- Remove the first node or move head to the next node.
  Then what is wrong with the following code?
- When removing a node, beware of memory leak; remember to give yourself a pointer to the node that is about to be removed before you lose your pointer to it:

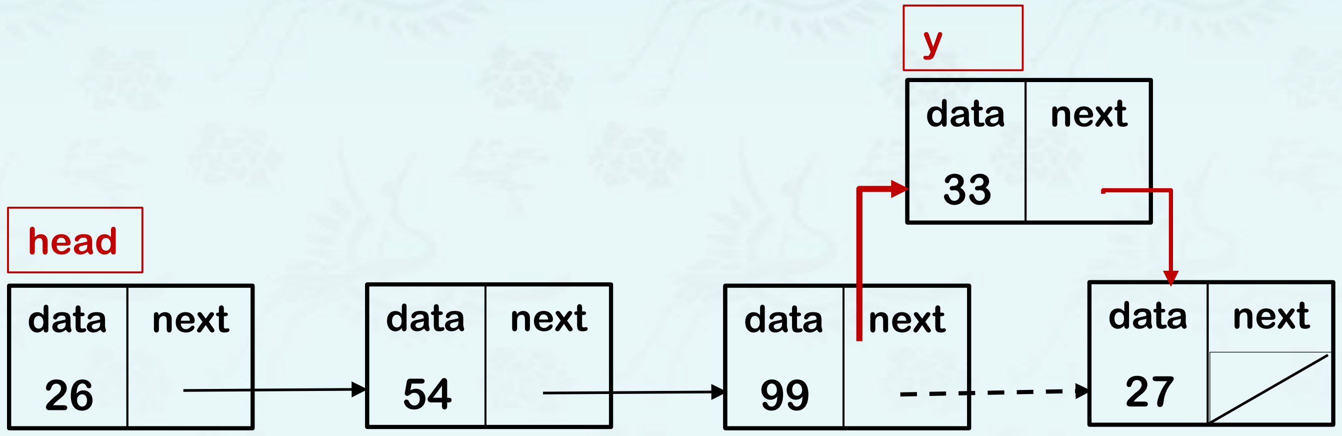
```
Node* zap = head;
head = head->next;
delete zap;
```



Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.

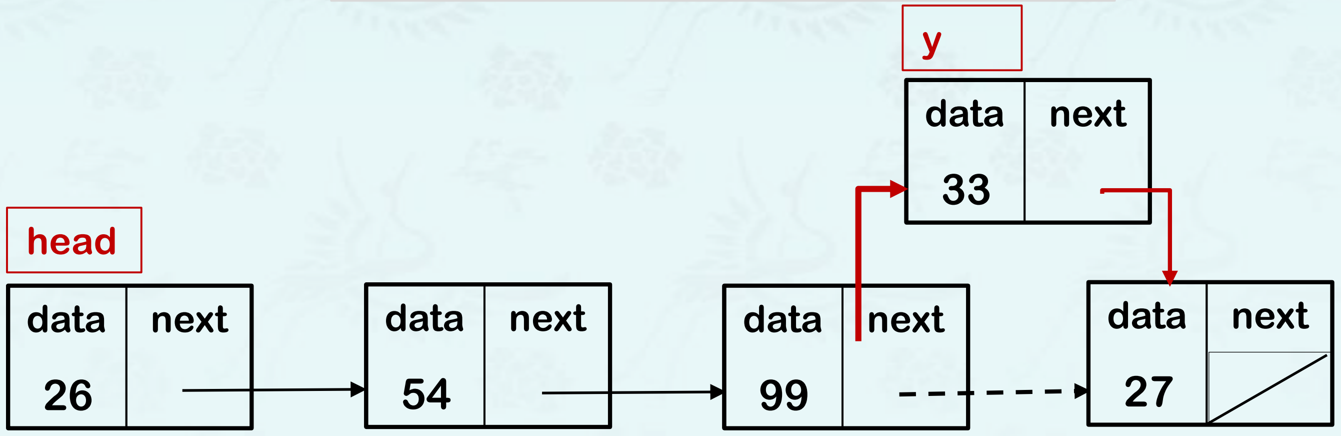


Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.
- Starting from the head node, we have to stop at the node (n = 99) before the insertion point. Remember that a singly-linked list is a one way street!

```
x = head;
```



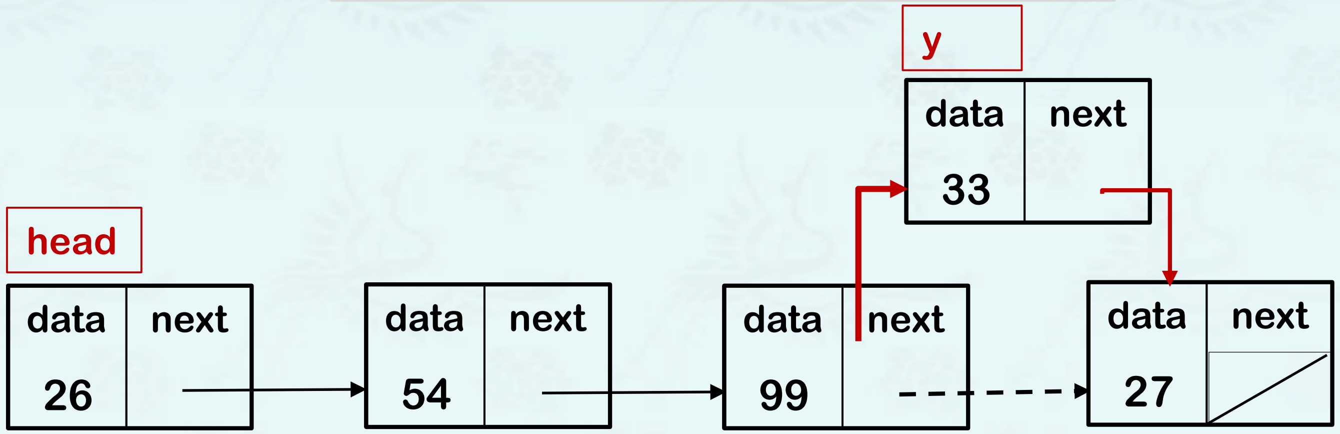
Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.
- Starting from the head node, we have to stop at the node (n = 99) before the insertion point. Remember that a singly-linked list is a one way street!

```
x = head;
while (x->data != 99)
  x = x->next;
```

Where is x pointing after while()?



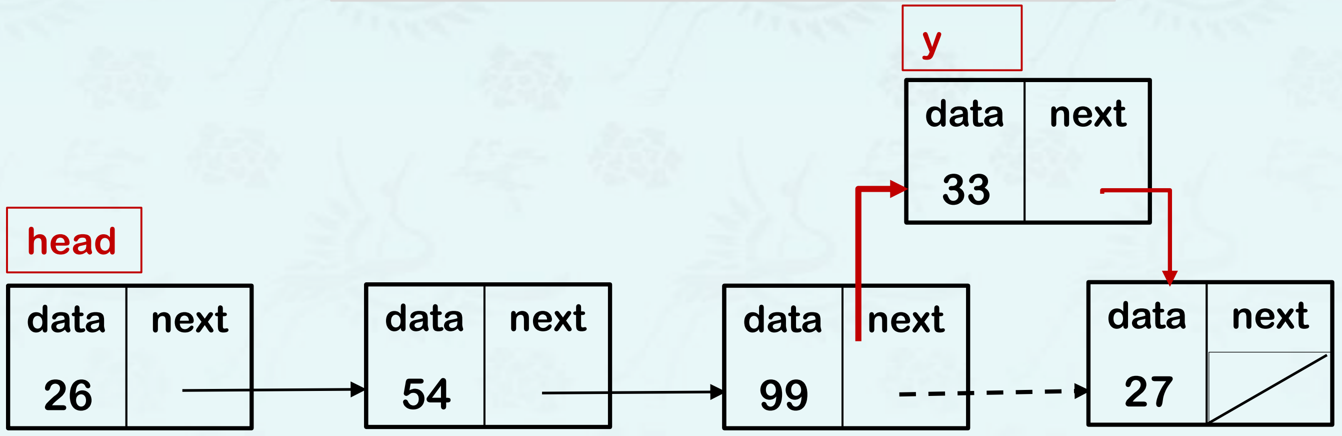
Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.
- Starting from the head node, we have to stop at the node (n = 99) before the insertion point. Remember that a singly-linked list is a one way street!

```
x = head;
while (x->data != 99)
  x = x->next;
y->next =
```

Where is x pointing after while()?

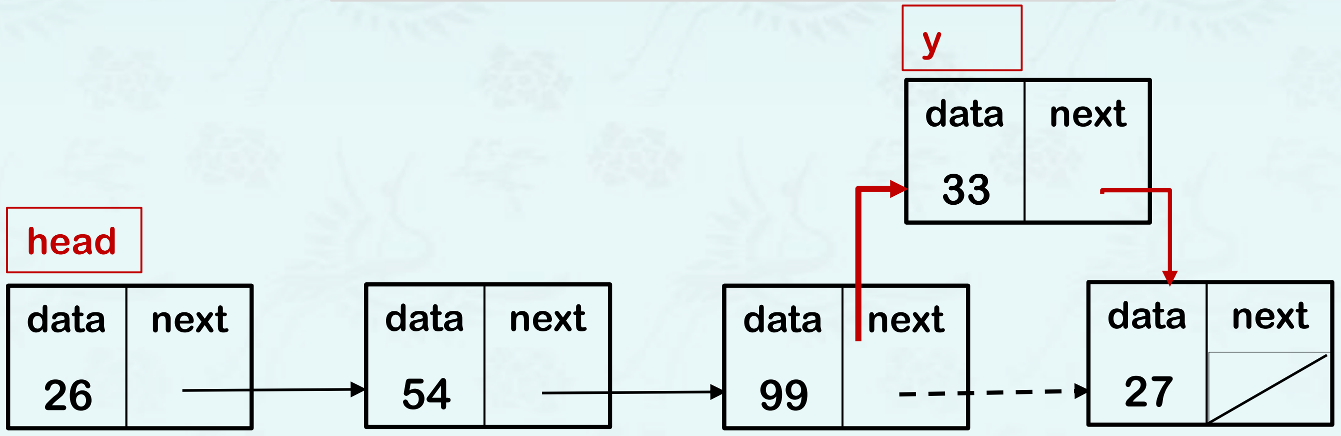


Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.
- Starting from the head node, we have to stop at the node (n = 99) before the insertion point. Remember that a singly-linked list is a one way street!

```
x = head;
while (x->data != 99)
  x = x->next;
y->next = x->next;
x->next = 
```

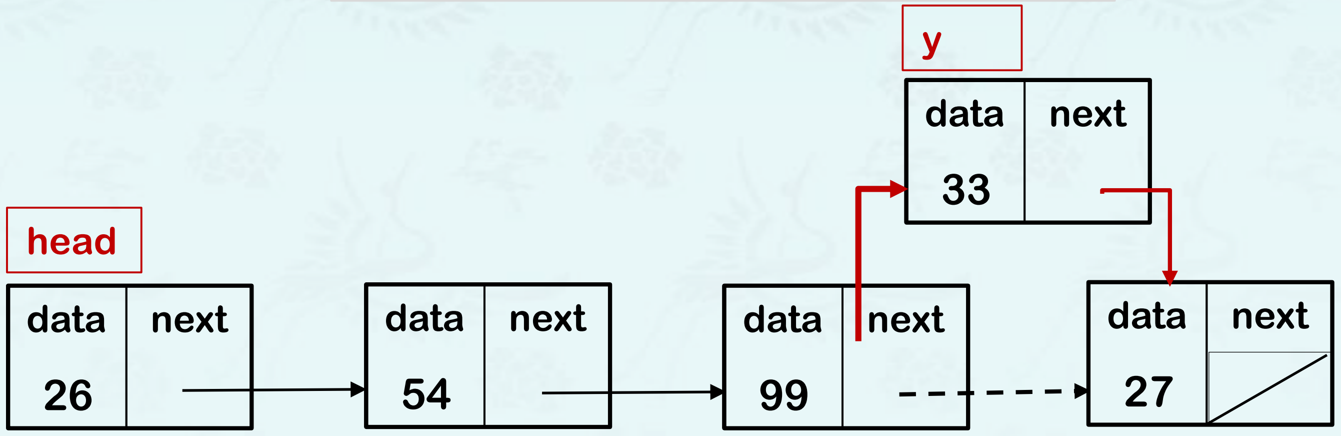


Dynamic Data Structures – insert()

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

...

Node* head, *x, *y;
```

- Insert a node(n = 33) after the node (n = 99) as shown below.
- Starting from the head node, we have to stop at the node (n = 99) before the insertion point. Remember that a singly-linked list is a one way street!

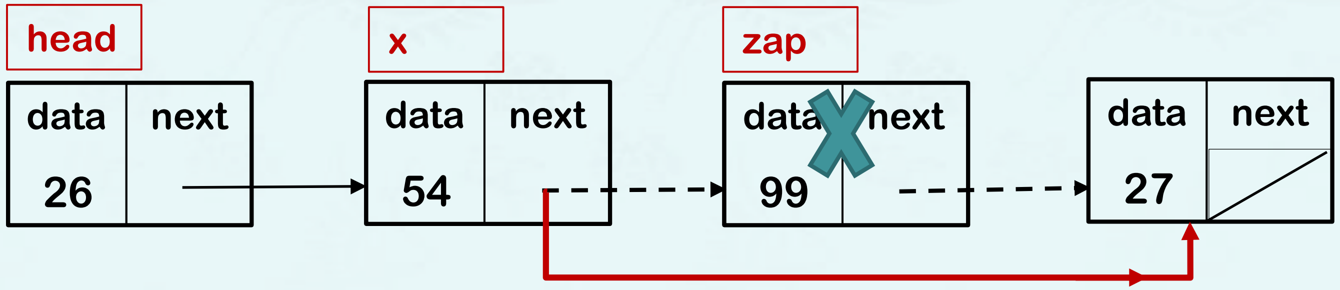
```
x = head;
while (x->data != 99)
  x = x->next;
y->next = x->next;
x->next = y;
```



Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

- Remove a node(n = 99) in the middle of list as shown below.



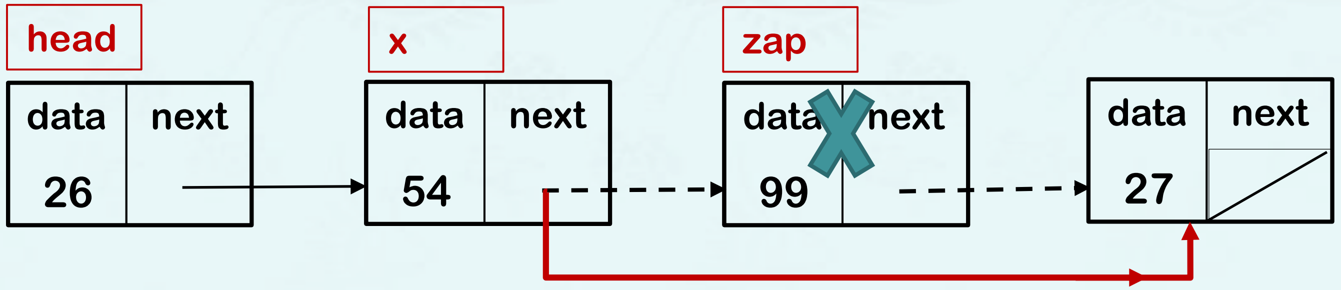
Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

- Remove a node(n = 99) in the middle of list as shown below.

To remove a node from a list we have to do three things:

- use a handle pointer (**zap** here) to keep hold of the unwanted node
- find the node **before** the unwanted node and make links.
- delete the unwanted node



Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

- Remove a node(n = 99) in the middle of list as shown below.

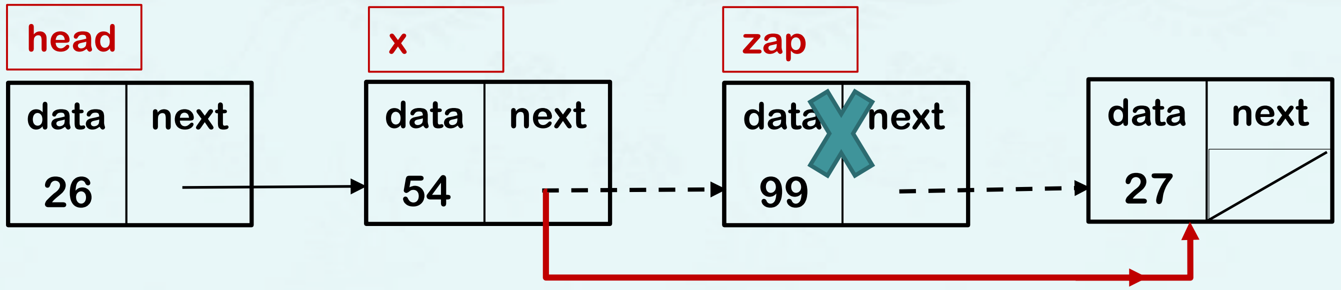
To remove a node from a list we have to do three things:

- use a handle pointer (**zap** here) to keep hold of the unwanted node
- find the node **before** the unwanted node and make links.
- delete the unwanted node

```
node* x = head, *zap = head->next;
while(zap->data!= 99) {
  x = zap;
  zap = zap->next;
}
```

→ To find both x and zap.

Assuming (1) there are at least two nodes, (2) 99 is not the head node, and (3) there is a 99 node.



Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

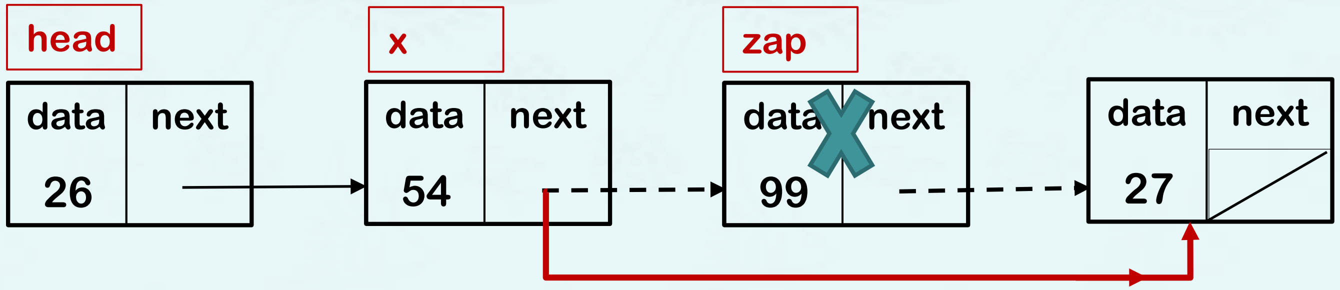
- Remove a node(n = 99) in the middle of list as shown below.

To remove a node from a list we have to do three things:
- use a handle pointer (**zap** here) to keep hold of the unwanted node
- find the node **before** the unwanted node and make links.
- delete the unwanted node

```
node* x = head, *zap = head->next;
while(zap->data!= 99) {
  x = zap;
  zap = zap->next;
}
x->next =
```

→ To find both x and zap.

Assuming (1) there are at least two nodes,
(2) 99 is not the head node, and
(3) there is a 99 node.



Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

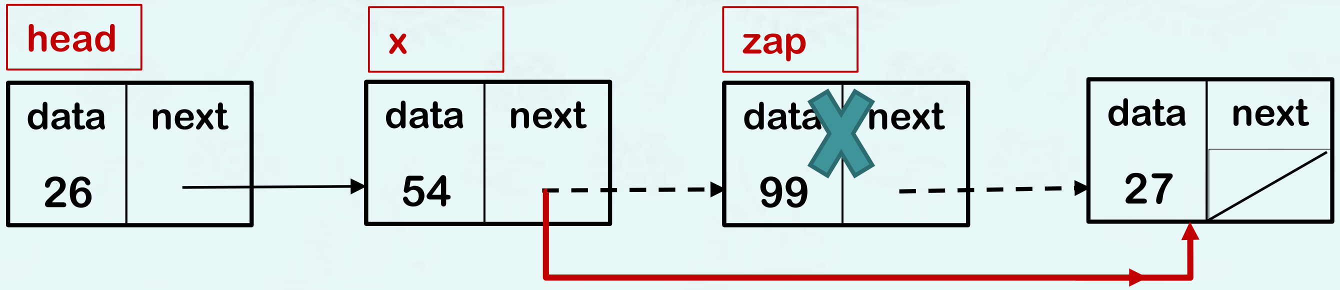
- Remove a node(n = 99) in the middle of list as shown below.

To remove a node from a list we have to do three things:
- use a handle pointer (**zap** here) to keep hold of the unwanted node
- find the node **before** the unwanted node and make links.
- delete the unwanted node

```
node* x = head, *zap = head->next;
while(zap->data!= 99) {
   x = zap;
   zap = zap->next;
}
x->next = zap->next;
```

To find both x and zap.

Assuming (1) there are at least two nodes, (2) 99 is not the head node, and (3) there is a 99 node.



Dynamic Data Structures – remove()

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

...

Node* head, *x, *y;
```

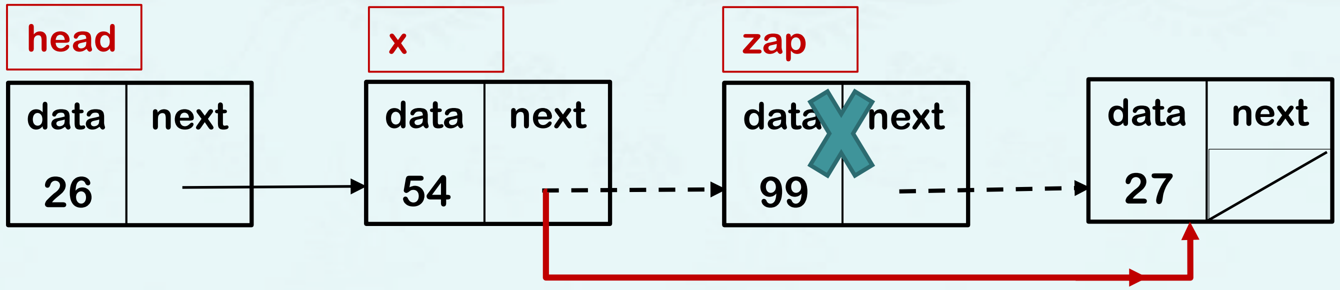
- Remove a node(n = 99) in the middle of list as shown below.

To remove a node from a list we have to do three things:
- use a handle pointer (**zap** here) to keep hold of the unwanted node
- find the node **before** the unwanted node and make links.
- delete the unwanted node

```
node* x = head, *zap = head->next;
while(zap->data!= 99) {
  x = zap;
  zap = zap->next;
}
x->next = zap->next;
delete zap;
```

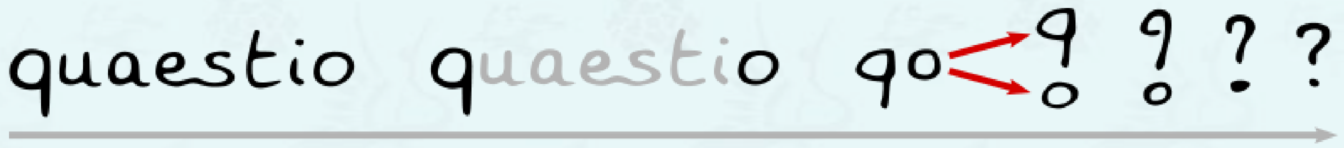
→ To find both x and zap.

Assuming (1) there are at least two nodes, (2) 99 is not the head node, and (3) there is a 99 node.



Summary

&


quaestio
quaestio
qo
?