Intermediate C Programming (2nd)

Yung-Hsiang Lu, Purdue University
Geroge K. Thiruvathukal, Loyola University Chicago

CRC Press
ISBN 9781032189819

Chapter 19 Linked Lists

Linked List 01

Dynamic Structures

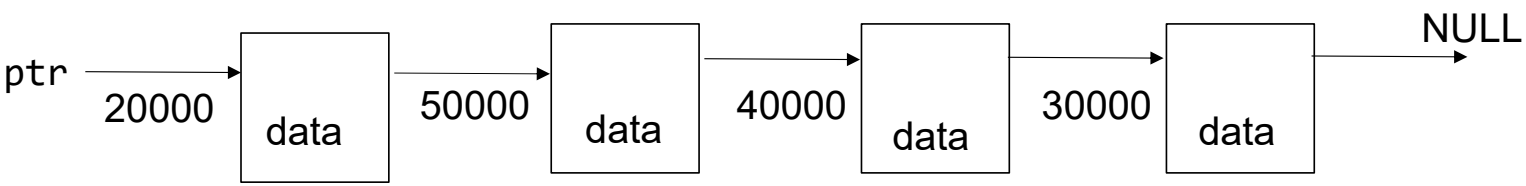
- Memory management:
  - Allocate memory when writing a program
  - Allocate memory after a program starts. Free before the program ends
- Allocate memory when needed. Free when no longer needed.
- Dynamic structures are used widely for problems whose sizes may change over time: database, web users, text editor, …

General Concept

Stack Memory		
	Address	Value
ptr	100	20000

Heap Memory	
Address	Value
	data
50000	40000
	data
40000	30000
	data
30000	NULL
	data
20000	50000

- a pointer `ptr` in the stack memory
- `ptr` points to heap memory
- The structure has a pointer and contains data
- The last piece points to NULL
- Each piece is called a node.

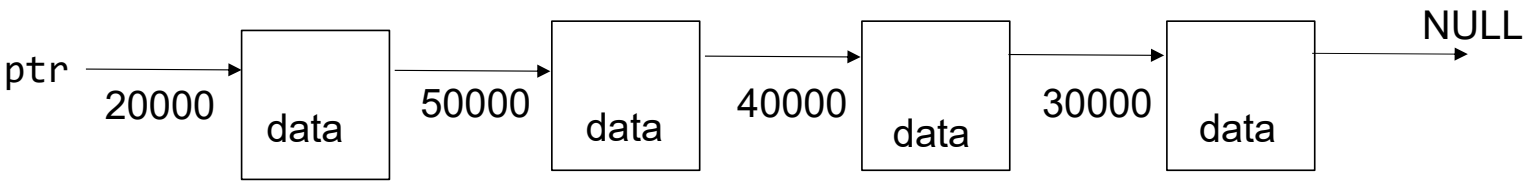


Why Heap or Stack Memory

- Heap memory can be allocated / freed. Stack memory cannot.
- Local variables and arguments are in stack memory
- Heap memory can be accessed by different functions
- malloc returns the allocated heap memory. malloc does not necessary return increasing or decreasing orders
- After malloc / free several times, the memory may be scattered

Container Structure

- The piece of memory may store different types of data.
- The structure is the same.
- The structure acts like “container” of data.
- This structure is called ***linked list***.

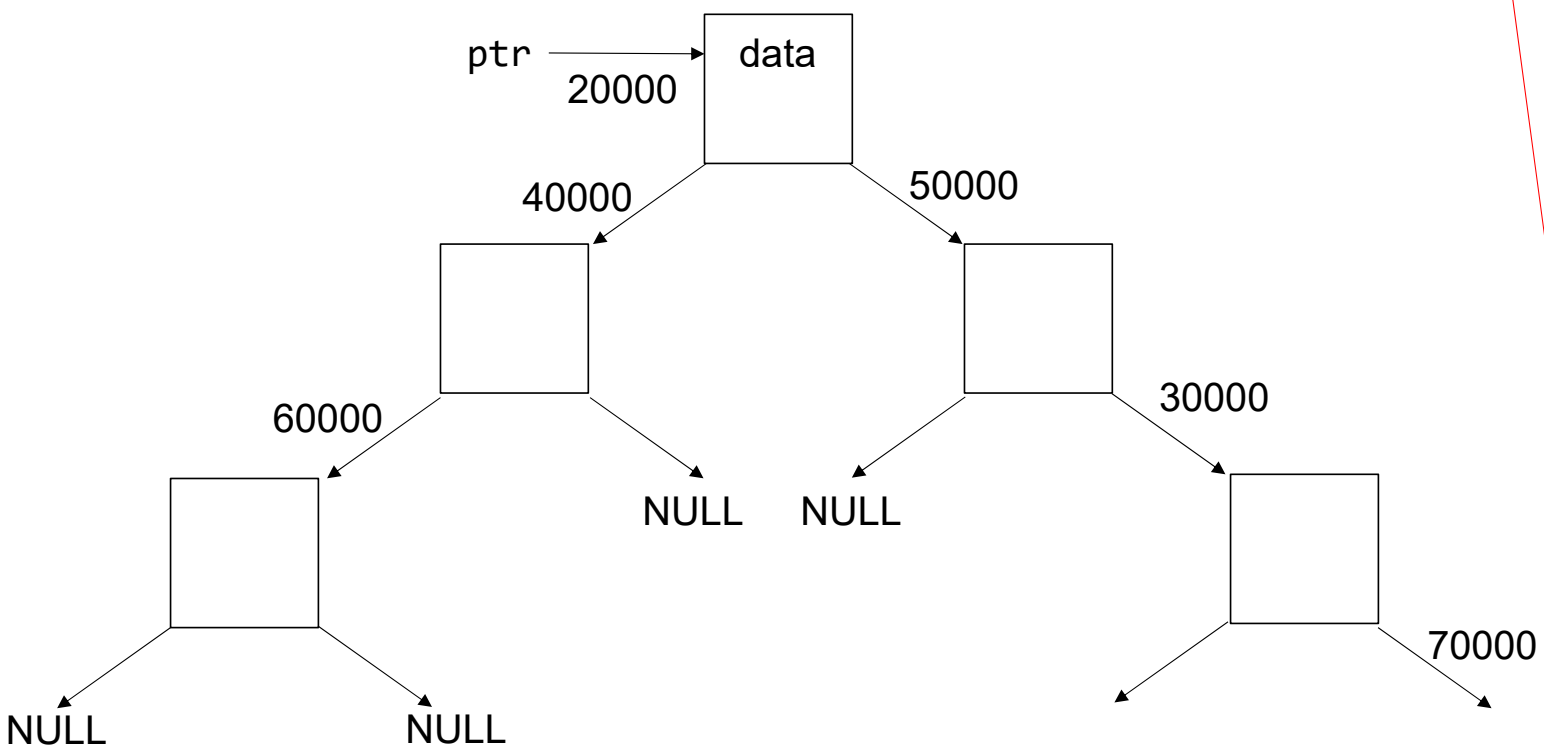


Two Pointers (binary tree)

- Each piece of memory has two pointers

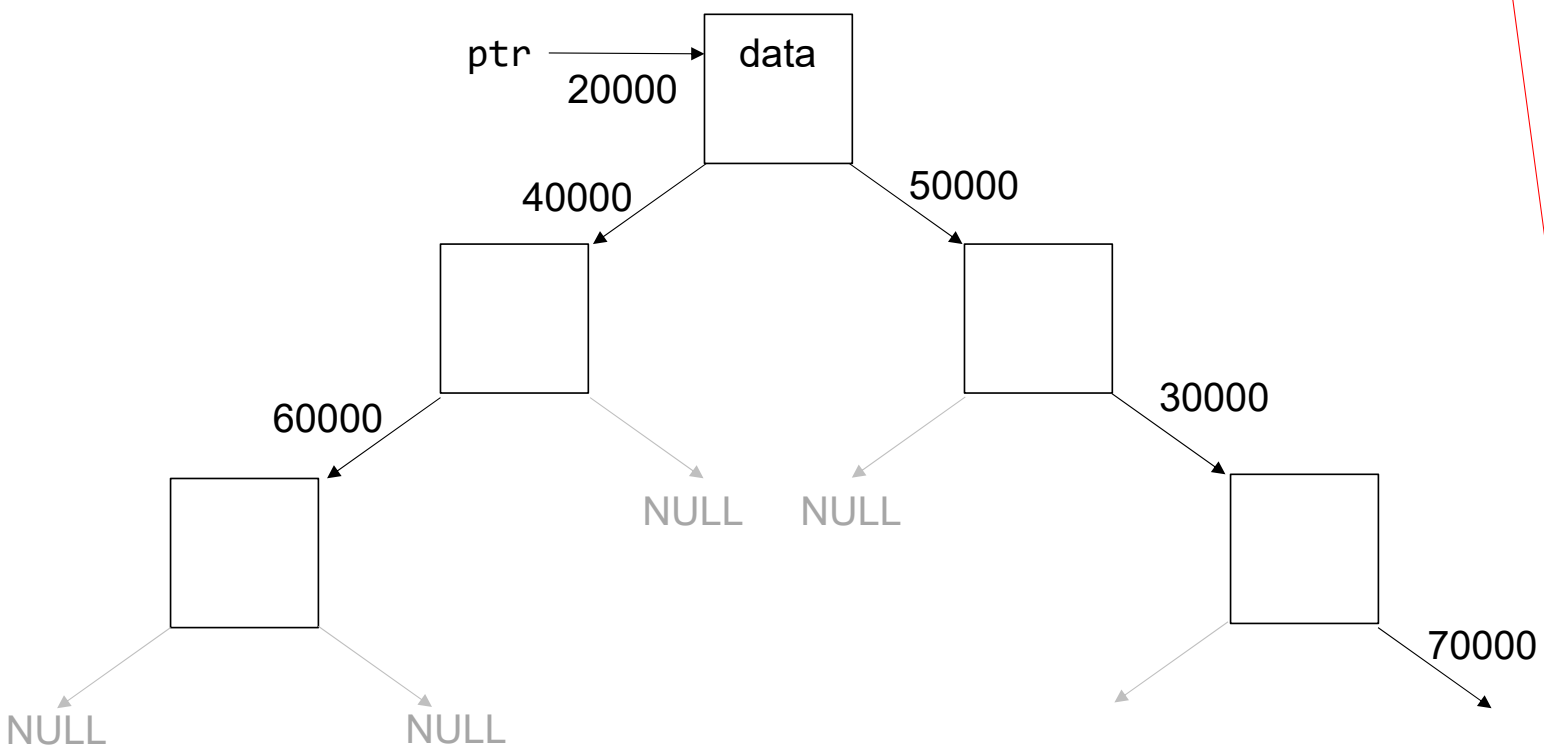
Stack Memory		
	Address	Value
ptr	100	20000

Heap Memory	
Address	Value
	data
	NULL
70000	NULL
	data
	NULL
60000	NULL
	data
	30000
50000	NULL
	data
	NULL
40000	60000
	data
	70000
30000	NULL
	data
	50000
20000	40000



Tree

- Usually, we do not draw → NULL



Stack Memory		
	Address	Value
ptr	100	20000

Heap Memory	
Address	Value
	data
	NULL
70000	NULL
	data
	NULL
60000	NULL
	data
	30000
50000	NULL
	data
	NULL
40000	60000
	data
	70000
30000	NULL
	data
	50000
20000	40000

Difference between one and two

- One and two are “fundamentally different” in their capabilities
- For any positive number n, it is possible finding k such that

$$2^k = n$$

- n = 4 $\Rightarrow$ k =2; n = 8 $\Rightarrow$ k = 3; n = 0.5 $\Rightarrow$ k = -1 …
- The same property holds for 3, 4, … any number greater than 1

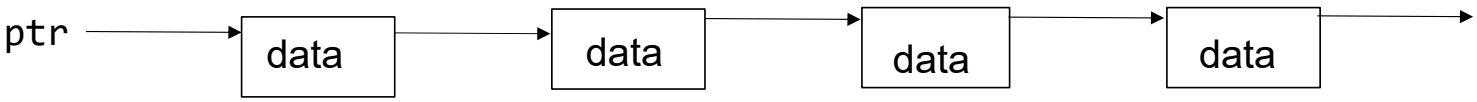
$$3^m = n$$

- n = 9 $\Rightarrow$ m = 2; n = 27 $\Rightarrow$ m = 3; n = $\frac{1}{3}$ $\Rightarrow$ m = -1 …
- It is not possible for one.

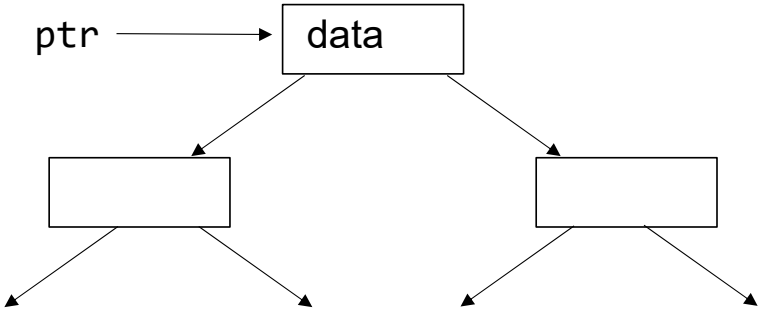
$$1^k = 1$$

Linked List vs Binary Tree

- Linked list is one-dimensional. Going to the middle has to pass half of the list.



- Binary tree is two dimensional and can eliminate (about) half data in a single step.



Linked List

must be the same

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  // data below
  int value;
  char name[20];
  double height; // meter
} Node;
```

Linked List

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  // data below
  int value;
  char name[20];
  double height; // meter
} Node;
```

Node is a new type

Linked List

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  // data below
  int value;
  char name[20];
  double height; // meter
} Node;
```

Can include many types of data

Linked List

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  // data below
  int value;
  char name[20];
  double height; // meter
} Node;
```

Can be later in the list of attributes

Container Structure

- insert: insert data
- delete: delete (a single piece of) data
- search: is a piece of data stored
- destroy: delete all data

Linked List Node storing int

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  int value; // for simplicity, each node stores int
} Node;
```

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL; // important, do not forget
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;   /* insert at the beginning */
}
```

Forgetting NULL is a common mistake

Forgetting NULL is a common mistake

```
Node * head = NULL; /* must initialize it to NULL */
head = List_insert(head, 917);
head = List_insert(head, -504);
head = List_insert(head, 326);
```

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504);
  head = List_insert(head, 326);
```

must set to NULL

Frame	Symbol	Address	Value
main	head	200	NULL

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917); // RL
  head = List_insert(head, -504);
  head = List_insert(head, 326);
```

Frame	Symbol	Address	Value
insert	p	312	U
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Frame	Symbol	Address	Value
construct	n	404	U
	v	400	917
	value address 312		
	return location		
insert	p	312	U
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	U
next	10000	U 917

Frame	Symbol	Address	Value
construct	n	404	A10000
	v	400	917
	value address 312		
	return location		
insert	p	312	U
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	917
next	10000	U

NULL

Frame	Symbol	Address	Value
construct	n	404	A10000
	v	400	917
	value address 312		
	return location		
insert	p	312	U
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
construct	n	404	A10000
	v	400	917
	value address 312		
	return location		
insert	p	312	U
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
construct	n	404	A10000
	v	400	917
	value address 312		
	return location		
insert	p	312	A10000
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
insert	p	312	A10000
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> value = v;
  n -> next = NULL;
  return n;
}
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
```

Symbol	Address	Value
value	10008	917
next	10000	~~NULL~~

NULL

Frame	Symbol	Address	Value
insert	p	312	A10000
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	NULL

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917); // RL
  head = List_insert(head, -504);
  head = List_insert(head, 326);
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
insert	p	312	A10000
	v	308	917
	h	300	NULL
	value address 200		
	return location		
main	head	200	~~NULL~~ A10000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917); // RL
  head = List_insert(head, -504);
  head = List_insert(head, 326);
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
main	head	200	A10000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

Symbol	Address	Value
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
insert	p	312	U
	v	308	-504
	h	300	A10000
	value address 200		
	return location		
main	head	200	A10000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

Symbol	Address	Value
value	20008	-504
next	20000	NULL
value	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	-504
	h	300	A10000
	value address 200		
	return location		
main	head	200	A10000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

The new node (p) is
in front of the previous node

Symbol	Address	Value
value	20008	-504
next	20000	~~NULL~~
value	10008	917 **A10000**
next	10000	NULL

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	-504
	h	300	A10000
	value address 200		
	return location		
main	head	200	A10000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

Symbol	Address	Value
v	20008	-504
next	20000	A10000
v	10008	917
next	10000	NULL

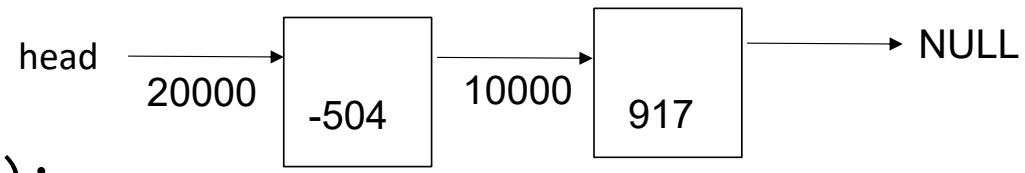
Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	-504
	h	300	A10000
	value address 200		
	return location		
main	head	200	~~A10000~~

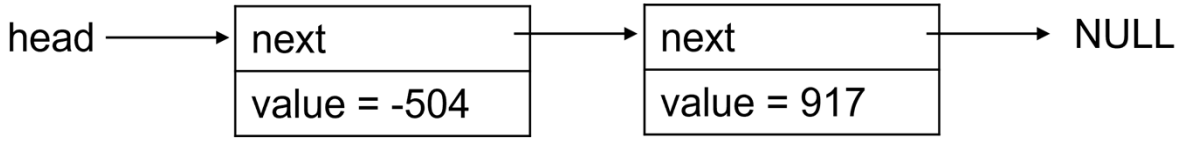
A20000

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

Symbol	Address	Value
v	20008	-504
next	20000	A10000
v	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
main	head	200	A20000



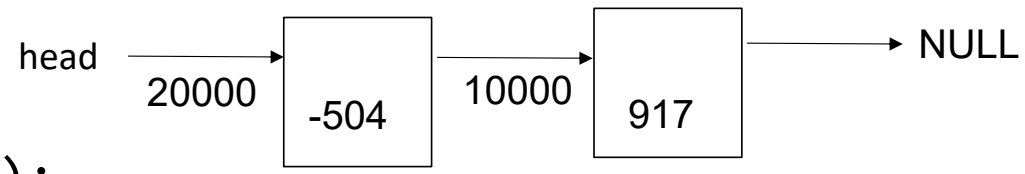

head
next
value = -504
next
value = 917
NULL

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  head = List_insert(head, 917);
  head = List_insert(head, -504); // RL
  head = List_insert(head, 326);
```

The new node (p) is
in front of the previous node

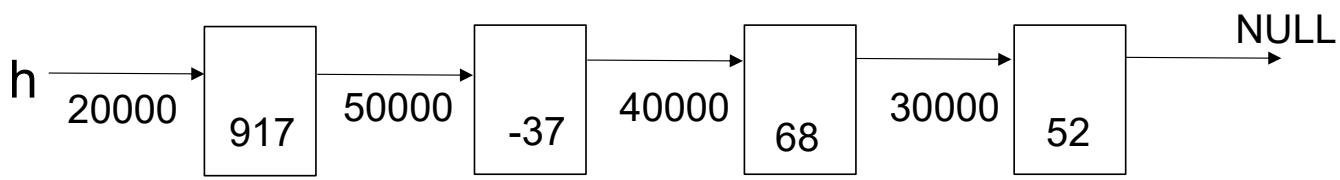
Symbol	Address	Value
v	20008	-504
next	20000	A10000
v	10008	917
next	10000	NULL

Frame	Symbol	Address	Value
main	head	200	A20000

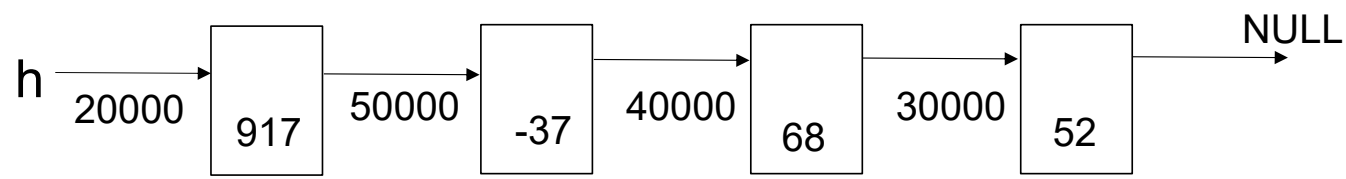


Linked List 02

```
/* search a value in a linked list starting with head, return
the node whose value is v, or NULL if no such node exists */
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
```



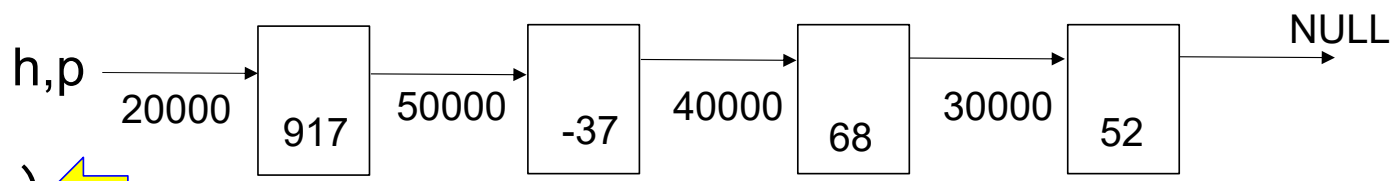
```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



must not use head in both

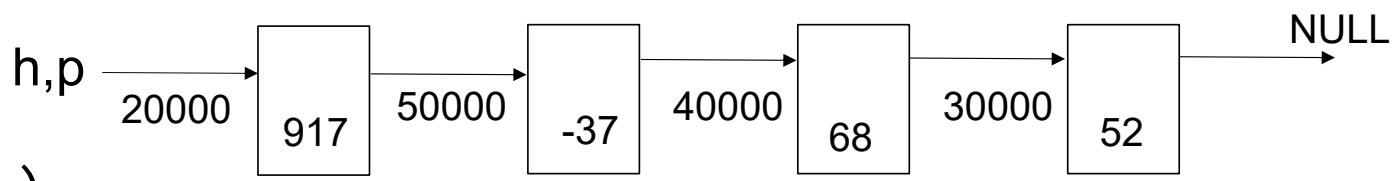
Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



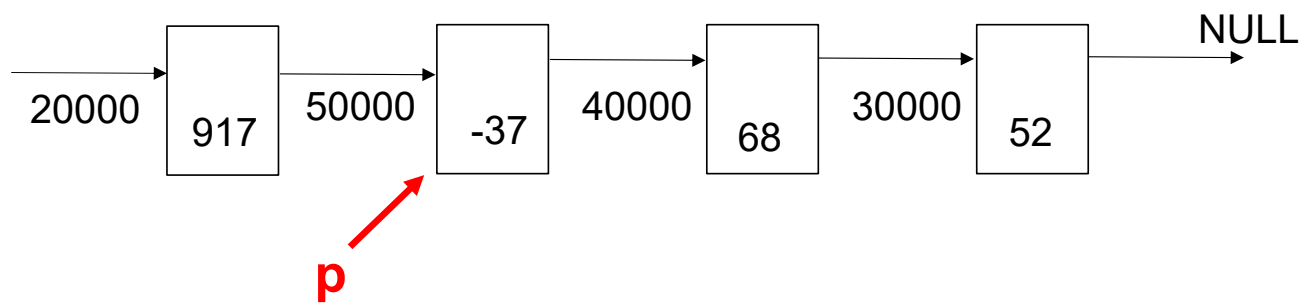
Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



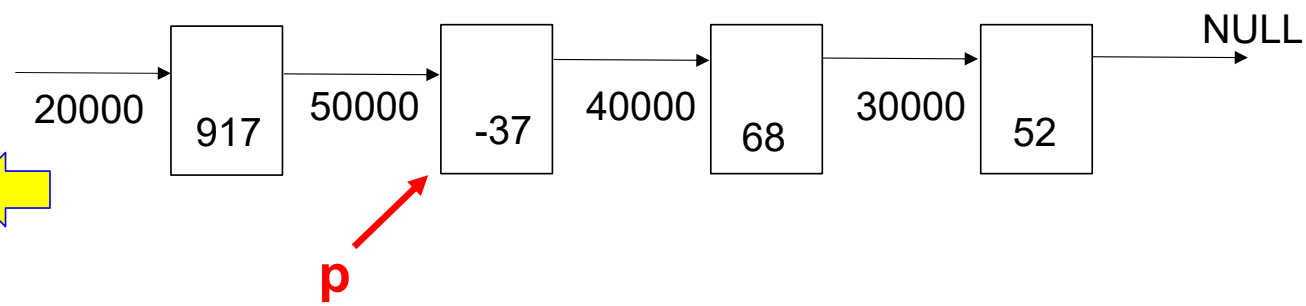
Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



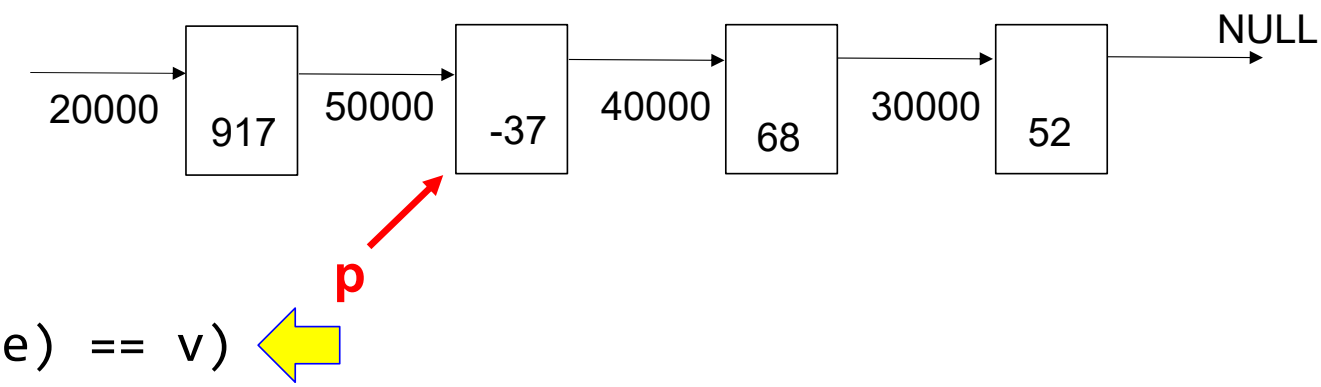
Frame	Symbol	Address	Value
insert	p	312	A50000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



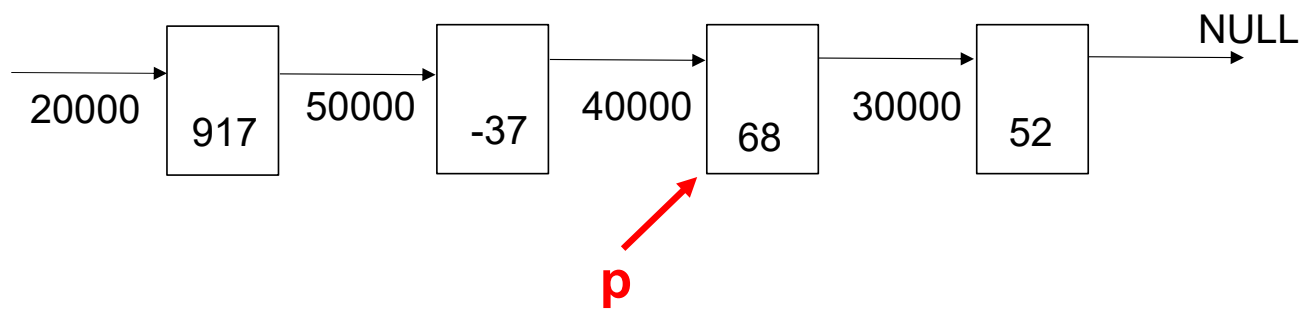
Frame	Symbol	Address	Value
insert	p	312	A50000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



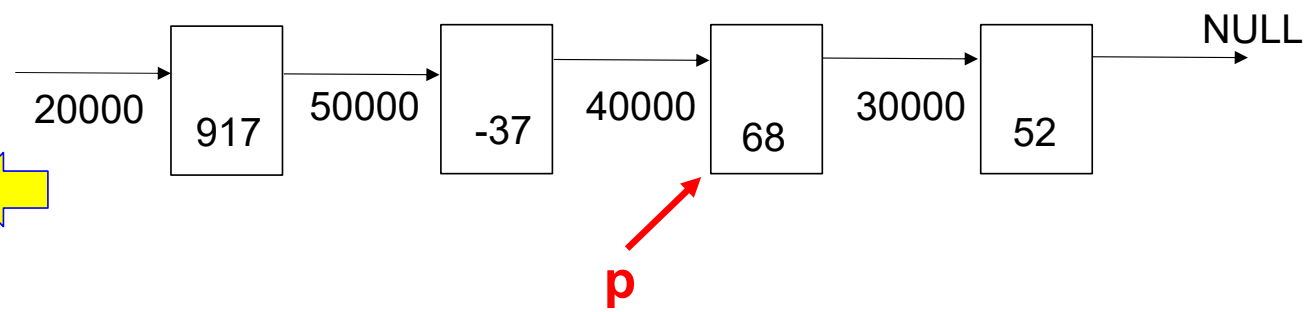
Frame	Symbol	Address	Value
insert	p	312	A50000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



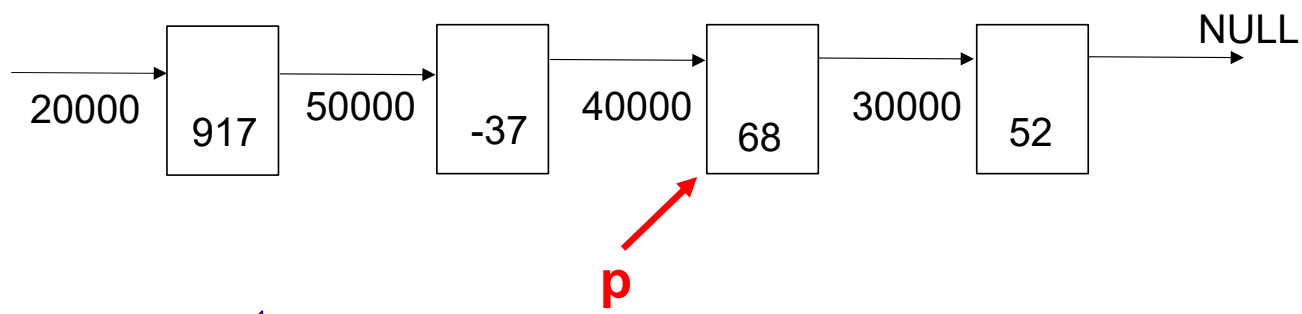
Frame	Symbol	Address	Value
insert	p	312	A40000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



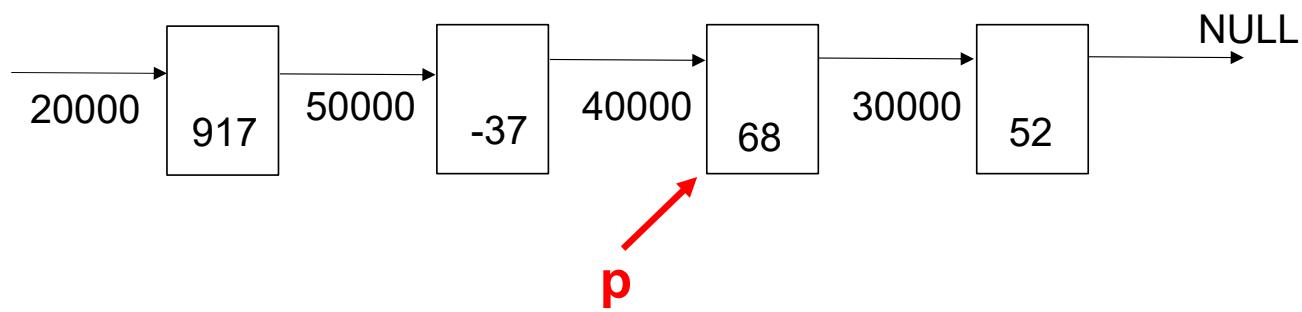
Frame	Symbol	Address	Value
insert	p	312	A40000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



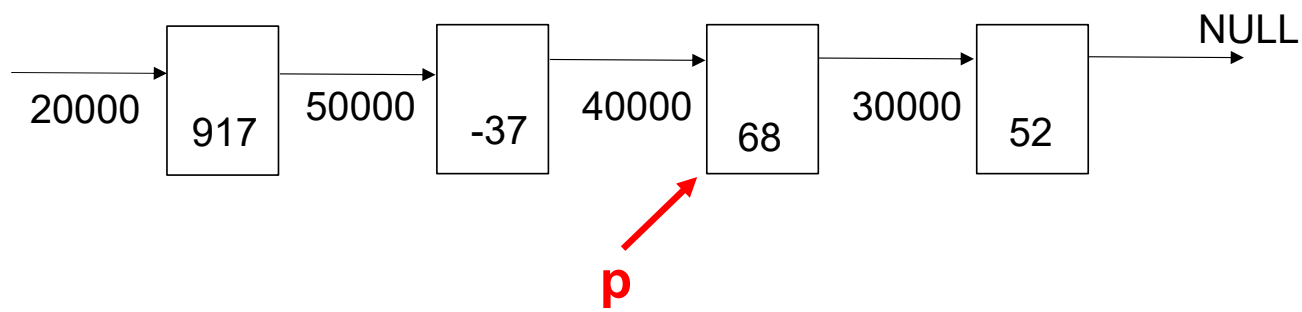
Frame	Symbol	Address	Value
insert	p	312	A40000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



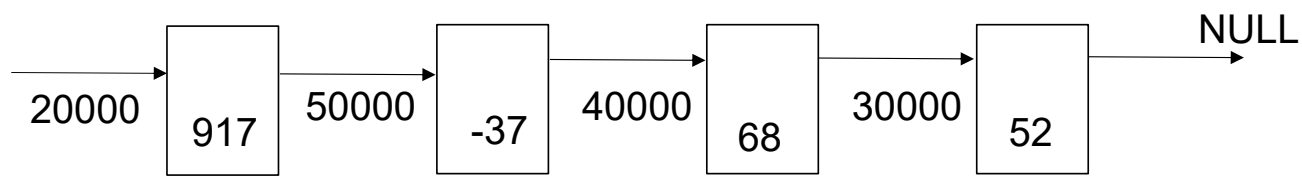
Frame	Symbol	Address	Value
insert	p	312	A40000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



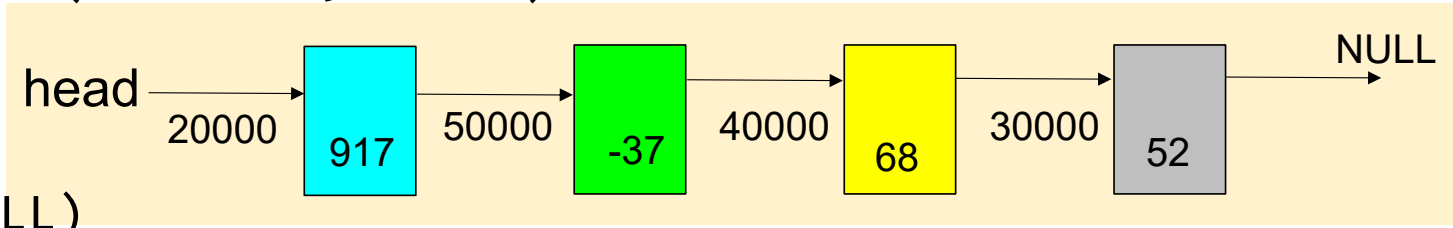
Frame	Symbol	Address	Value
insert	p	312	A40000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	A40000
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



Frame	Symbol	Address	Value
main	q	208	A40000
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```



Stack Memory			
Frame	Symbol	Address	Value
main	q	208	U
	head	200	A20000

Heap Memory		
Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

Frame	Symbol	Address	Value
insert	p	312	A50000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

Do we need p here? No
Can we use h? Yes

Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (p != NULL)
    {
      if ((p -> value) == v)
     { return p; }
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

Do we need q here? Yes
Can we use head? No

Symbol	Address	Value
value	50008	-37
next	50000	A40000
value	40008	68
next	40000	A30000
value	30008	52
next	30000	NULL
value	20008	917
next	20000	A50000

Frame	Symbol	Address	Value
insert	p	312	A20000
	v	308	68
	h	300	A20000
	value address 208		
	return location		
main	q	208	U
	head	200	A20000

```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while ((p != NULL) && ((p -> value) != v))
    {
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

```
if (A && B)
```

When A is false, B is not checked

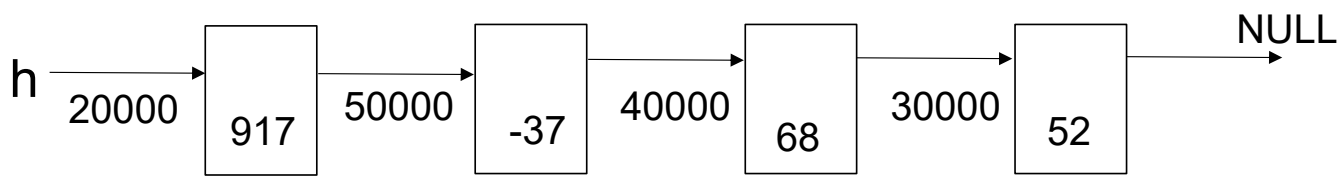
```
Node * List_search(Node * h, int v)
{
  Node * p = h;
  while (((p -> value) != v) && (p != NULL))
    {
      p = p -> next;
    }
  return p;
}
…
Node * q = List_search(head, 68);
```

This is wrong.

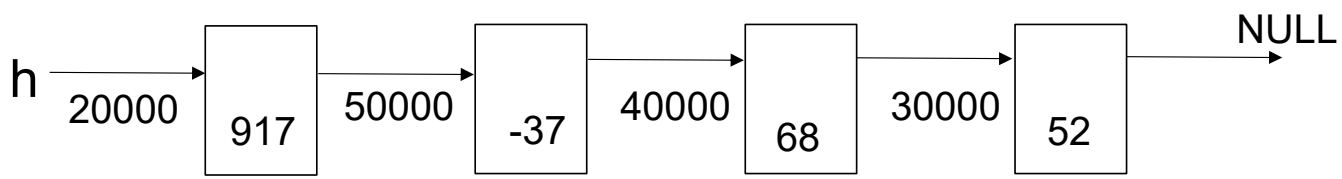
If p is NULL,
p -> value does not exist

Linked List 03

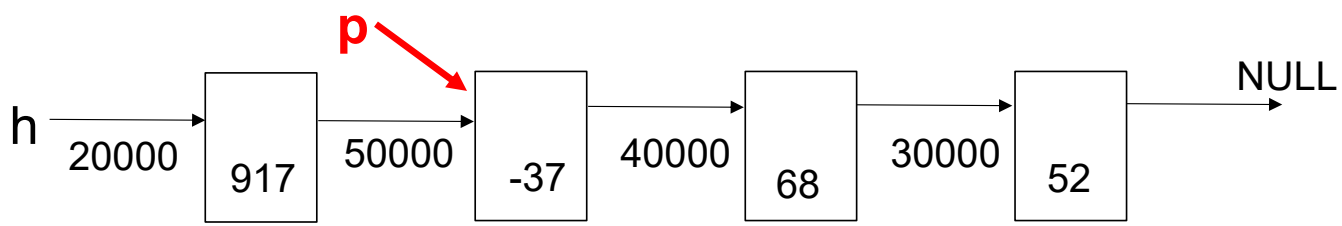
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



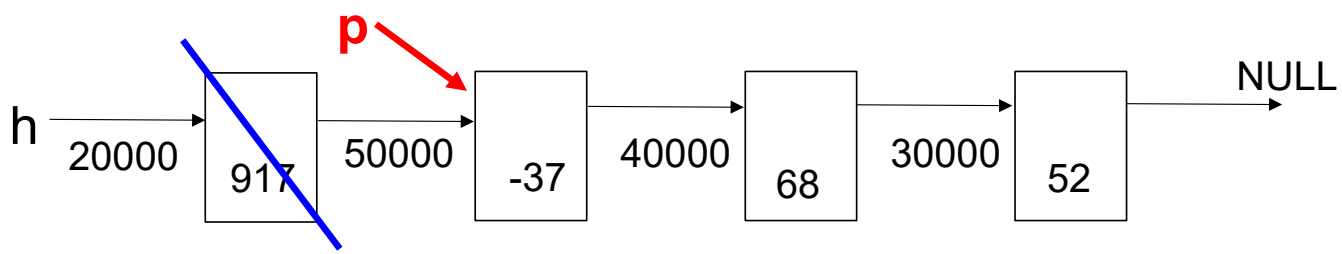
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    // almost every function start with checking NULL
    // if h is NULL, h -> next does not exist
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



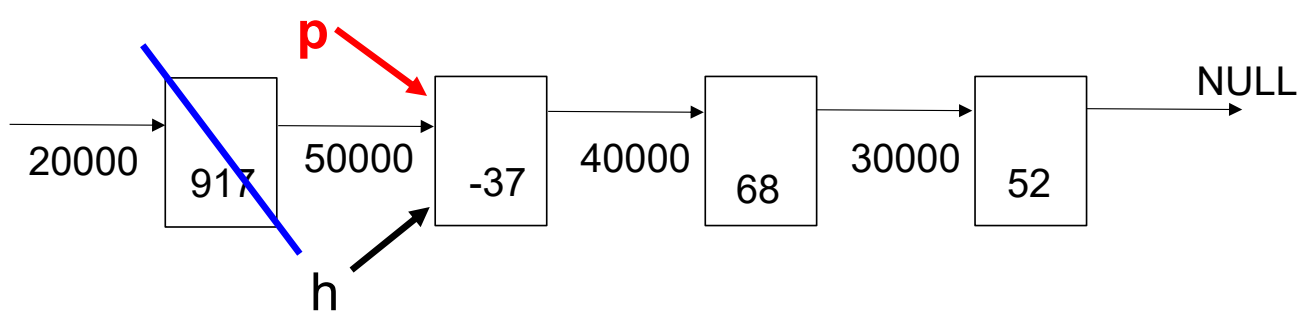
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



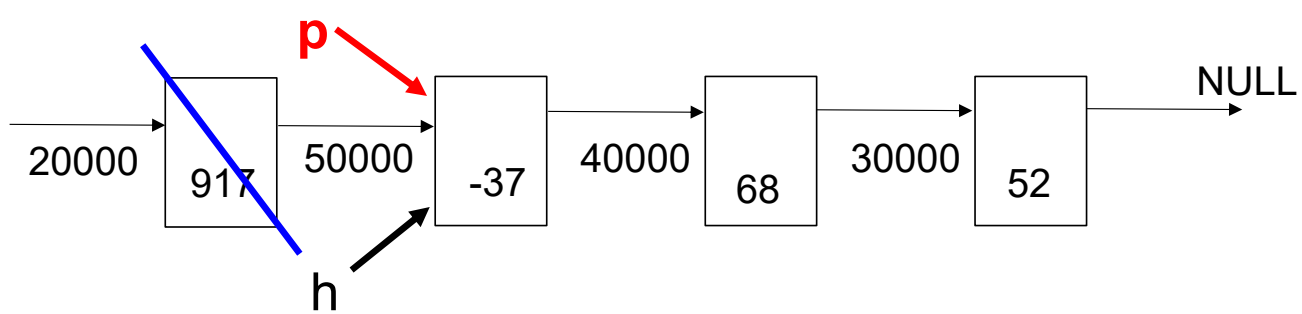
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



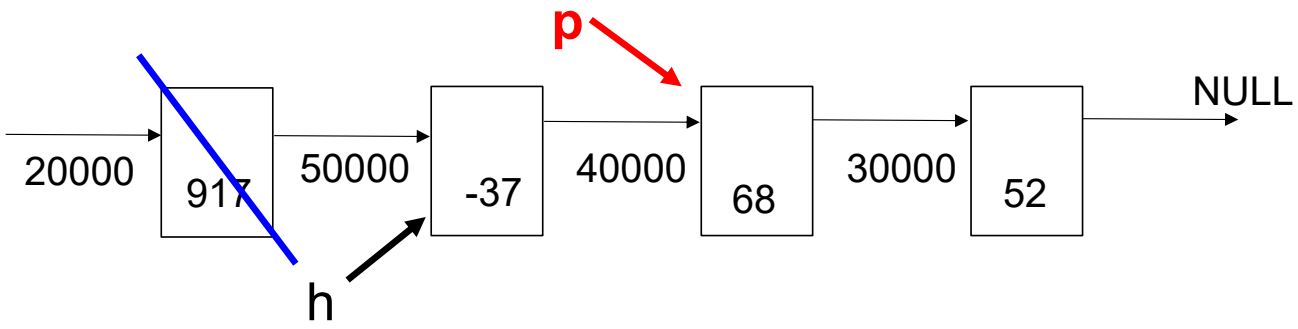
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



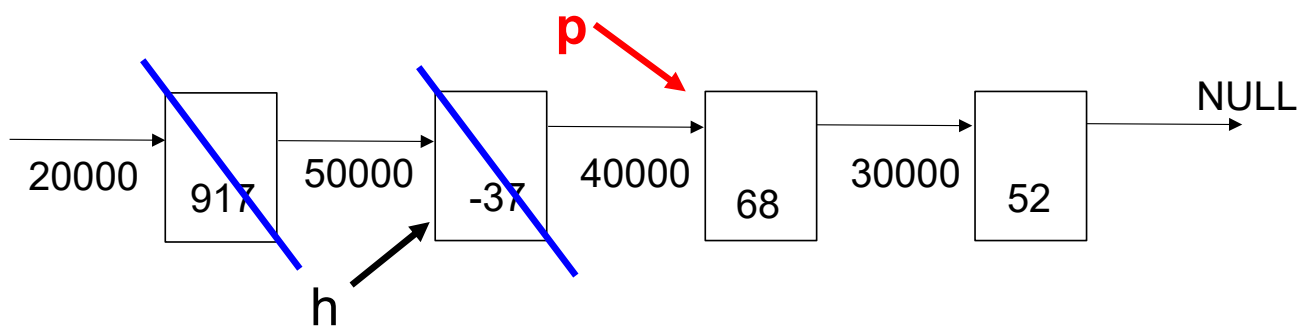
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



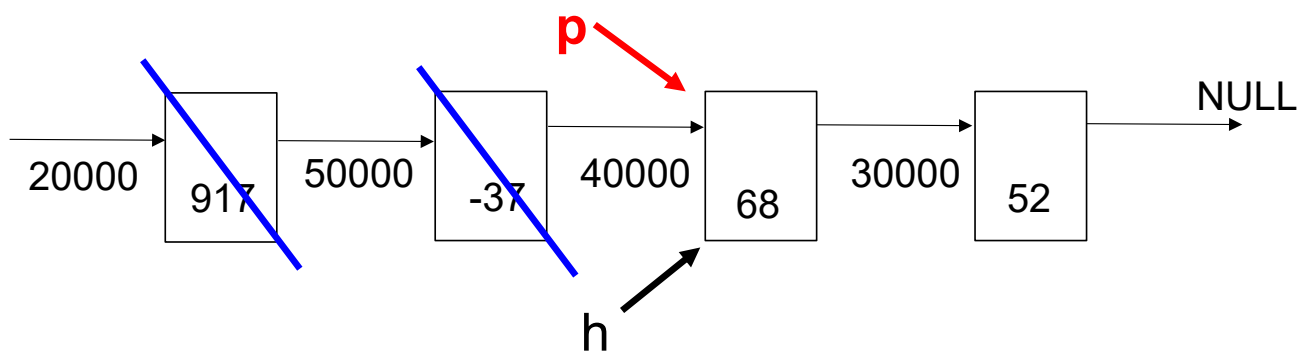
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



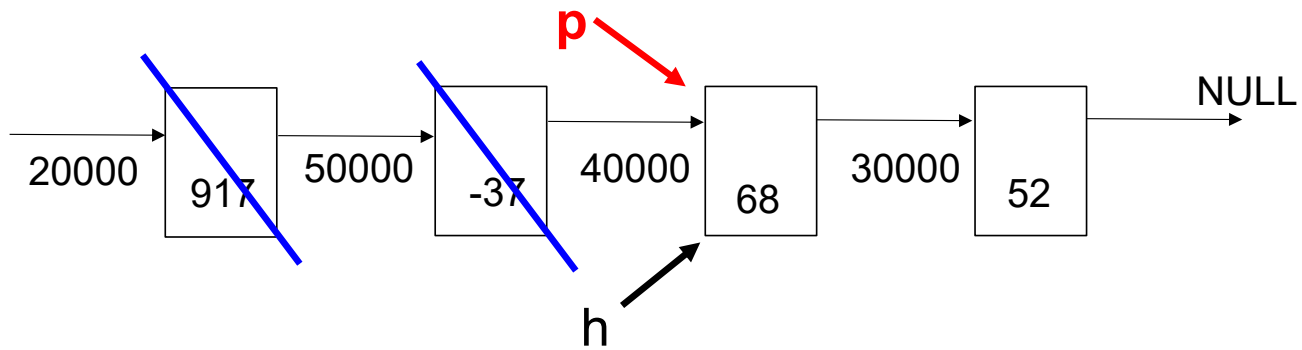
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



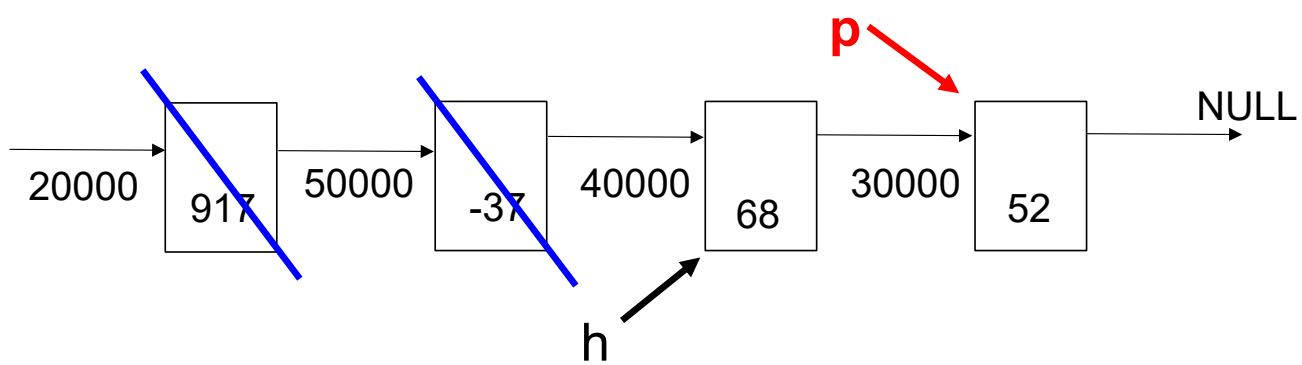
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



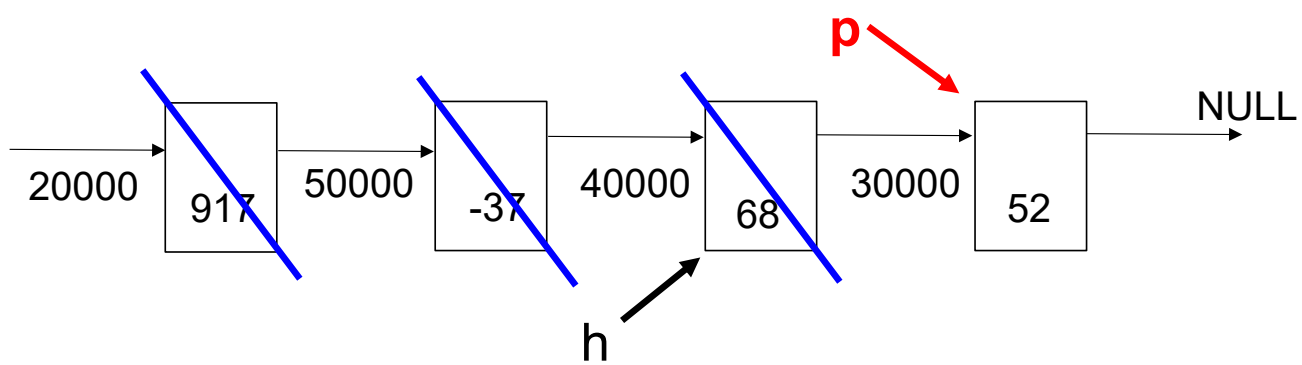
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



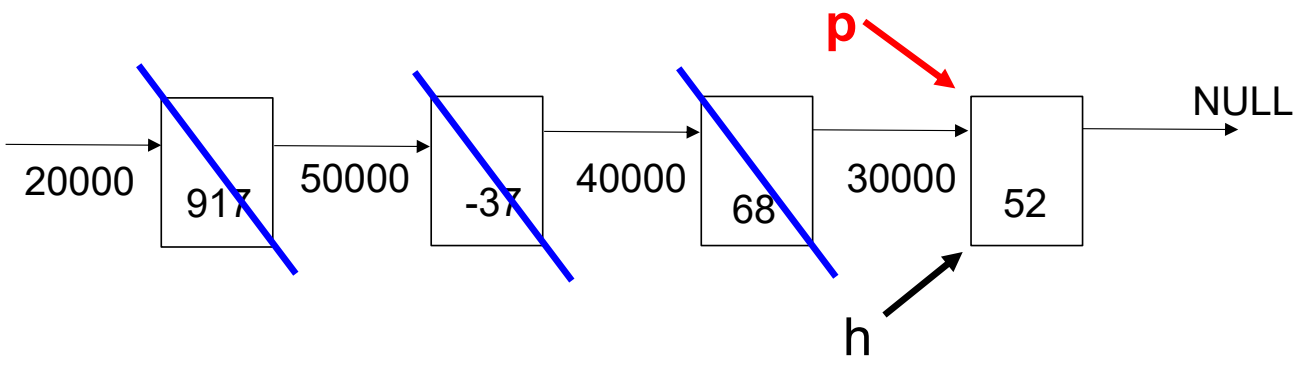
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



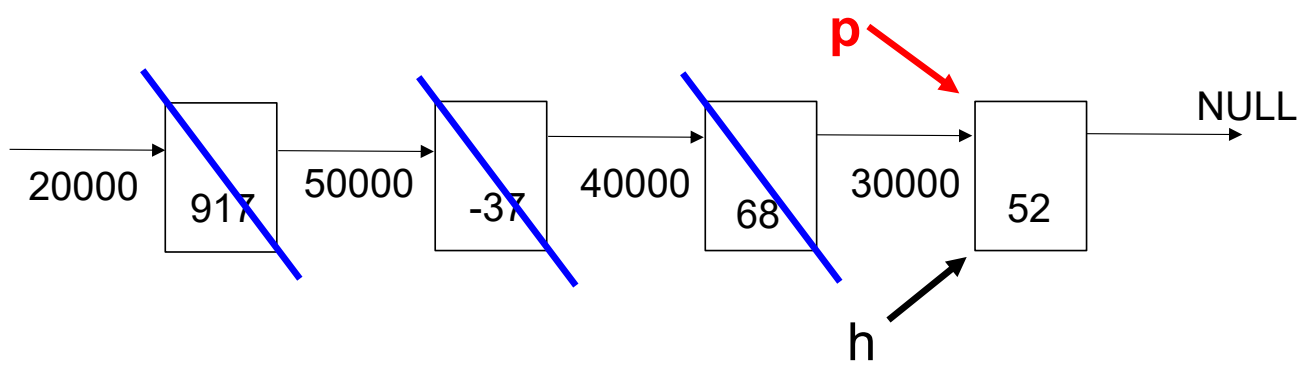
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



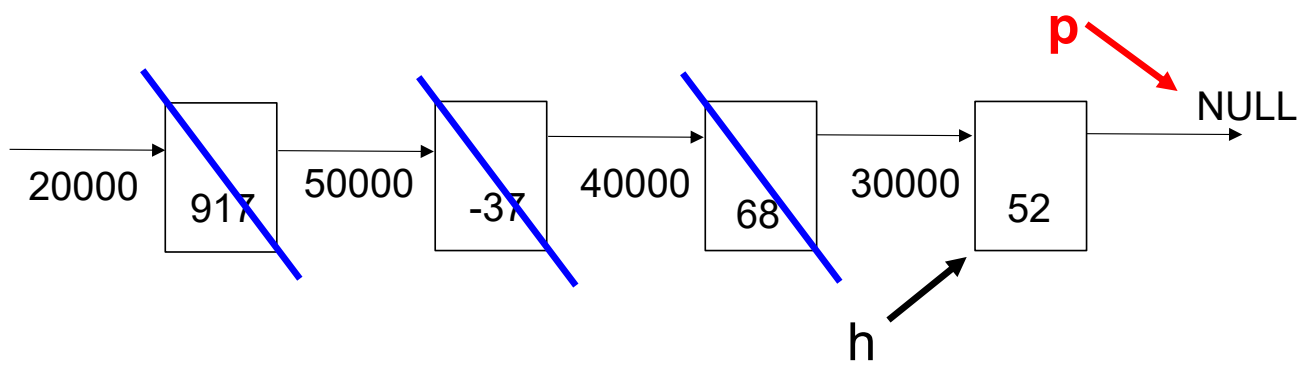
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



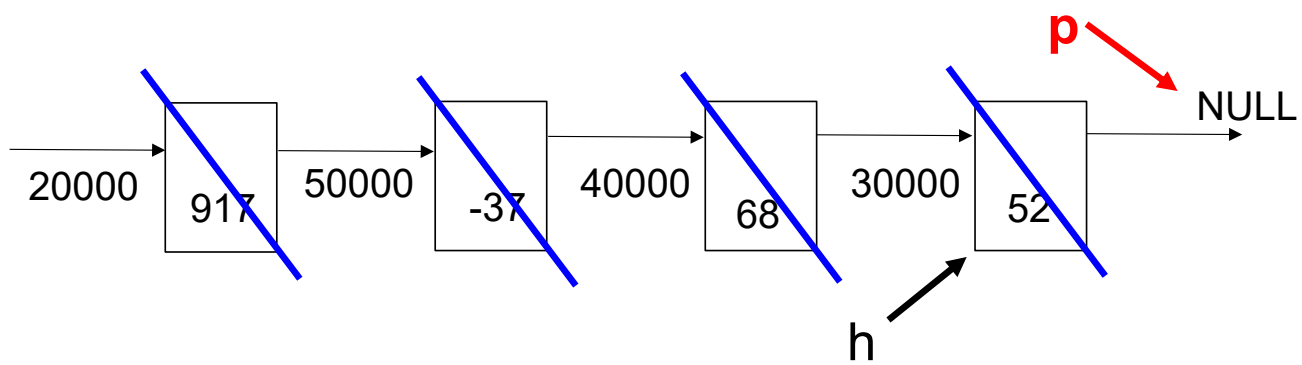
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



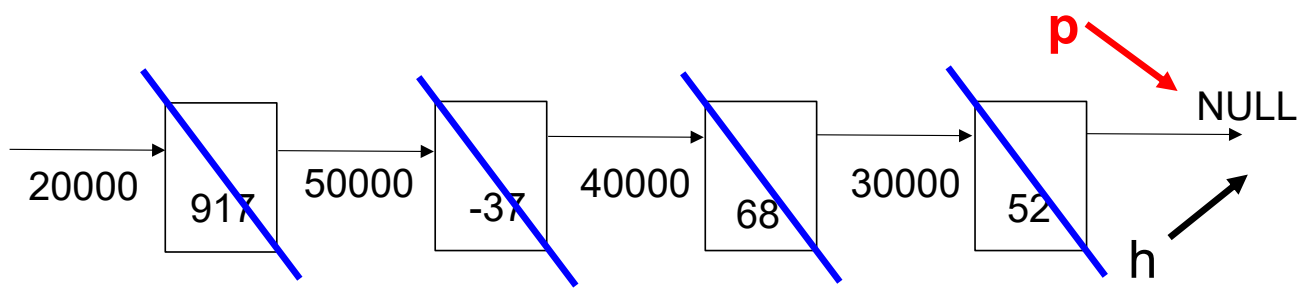
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



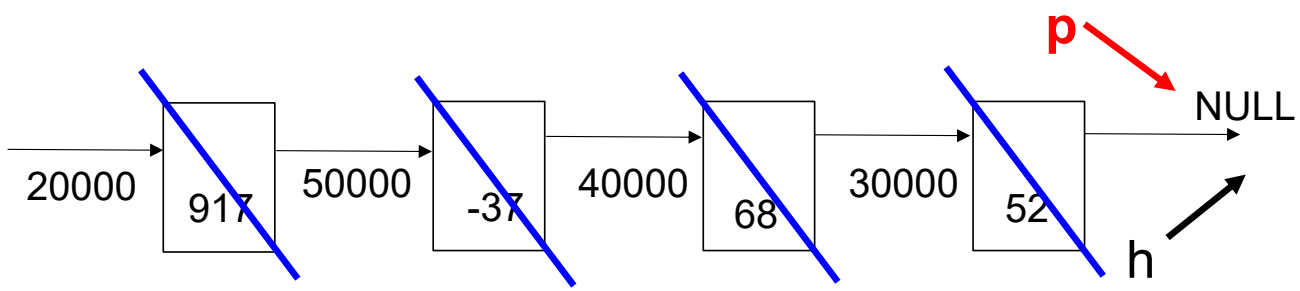
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



Common Questions

```
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```

Do I need to use another pointer p? Yes
Can I use only h? No
After free(h), h -> next does not exist

```
void List_destroy(Node * h)
{
  Node * p = NULL;
  while (h != NULL)
    {
      p = h -> next;
      free (h);
      h = p;
    }
}
```

Can I move p's definition inside while? Yes

p must be updated inside while

```
void List_destroy(Node * h)
{
  Node * p;
  while (h != NULL)
    {
      p = h -> next;
      free (h);
      h = p;
    }
}
```

Do I have to update h here? Yes

```
void List_destroy(Node * h)
{
  Node * p;
  while (h != NULL)
    {
      p = h -> next;
      free (h);
      h = p;
    }
}
```

Is h NULL after this line? No.
h's value is unchanged
free(h) does not set h to NULL

```
void List_destroy(Node * h)
{
  Node * p;
  while (h != NULL)
    {
      p = h -> next;
      free (h);
      h = p;
    }
}
```

The order of these three lines must not be changed

1	p = h -> next;
	free (h);
	h = p;
		correct

2	p = h -> next;
	h = p;
	free (h);
		free wrong node
		h -> next does not exist in the next iteration

3	free (h);
	p = h -> next;
	h = p;
		after free(h),
		h -> next does not exist

4	free (h);
	h = p;
	p = h -> next;
		p’s value is unknown
		h -> next is invalid

5	h = p;
	p = h -> next;
	free (h);
		p’s value is unknown
		h -> next is invalid

6	h = p;
	free (h);
	p = h -> next;
		p’s value is unknown
		free (h) is invalid

```
void List_destroy(Node * h)
{
  Node * p;
  while (h != NULL)
    {
      p = h -> next;
      free (h);
      h = p;
    }
}
```

The order of these three lines must not be changed

Linked List 04

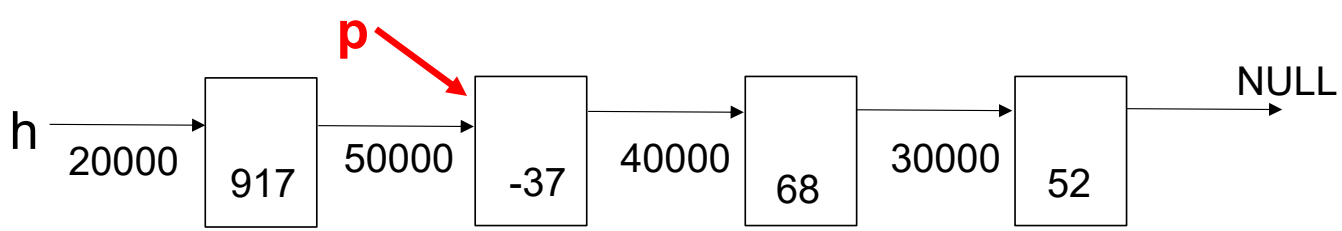
Delete a Node in a Linked List

- If the list is empty (NULL), do nothing, return NULL
- If the node to delete is the first node:
  - Save the second node
  - Free the first node
  - Return the second node (now is the first node)
- If the node to delete is not the first node:
  - Find the node to be deleted and the node in front of it
  - Bypass the node to be deleted
  - Free the node
  - Return the original first node

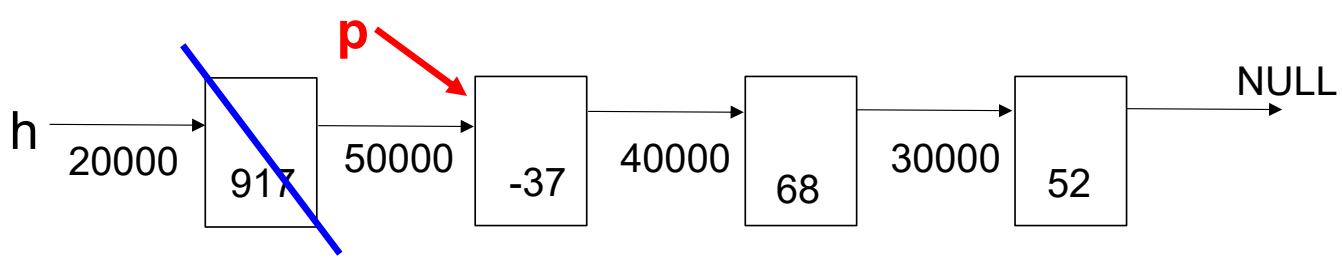
```
/* delete the node whose value is v in a linked list starting
with h, return the head of the remaining list, or NULL if the
list is empty. If multiple nodes contains v, delete the first
one. */

Node * List_delete(Node * h, int v)

{

    if (h == NULL) /* empty list, do nothing */

    {

      return h; // same as return NULL

    }
```

```
// h must not be NULL because it has been checked
// delete the first node (i.e. head)?
if ((h -> value) == v)
  {
    Node * p = h -> next; // p may be NULL, that’s ok
    free (h);
    return p;
  }
```

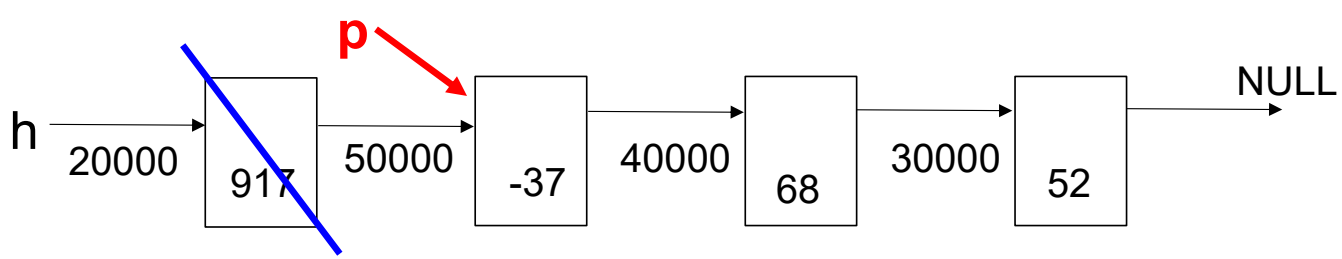
```
/* delete the first node (i.e. head)? */
if ((h -> value) == v)
  {                                  // p may be NULL, that's ok
    Node * p = h -> next;
    free (h);
    return p;
  }
```



```
/* delete the first node (i.e. head)? */
if ((h -> value) == v)
  {
    Node * p = h -> next;
    free (h);
    return p;
  }
```



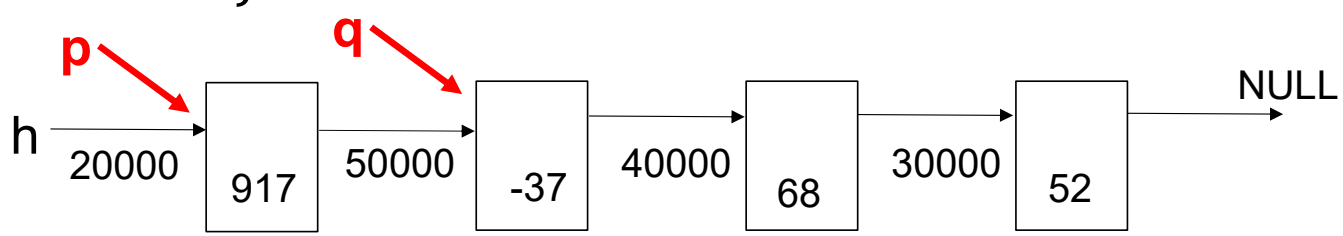
```
/* delete the first node (i.e. head)? */
if ((h -> value) == v)
  {
    Node * p = h -> next;
    free (h);
    return p;
  }
```



```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

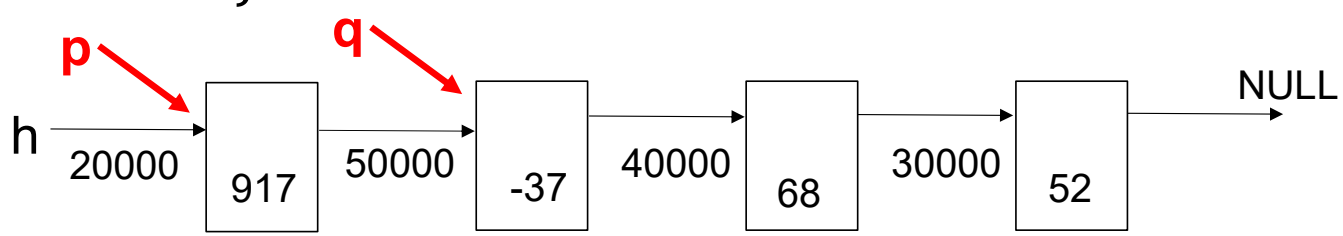
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



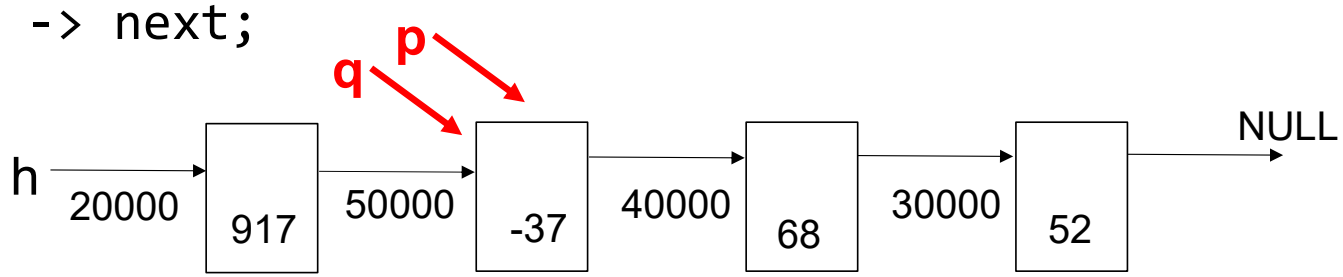
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



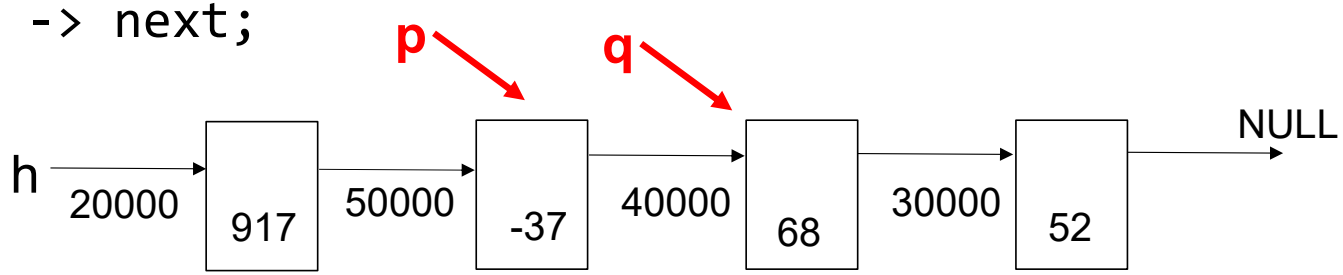
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



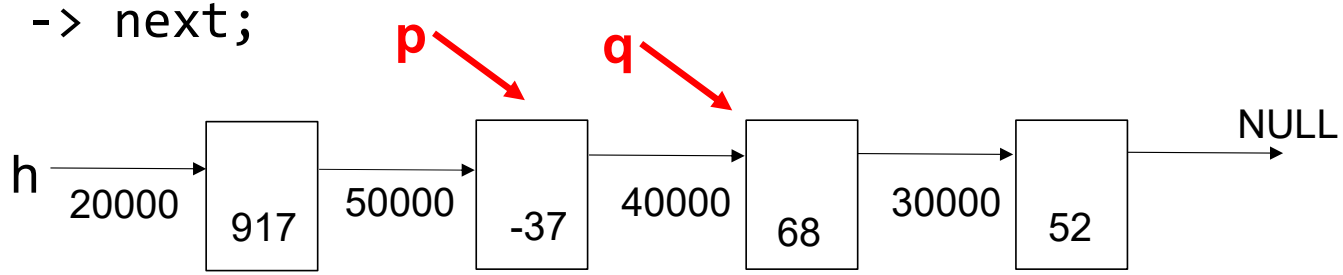
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



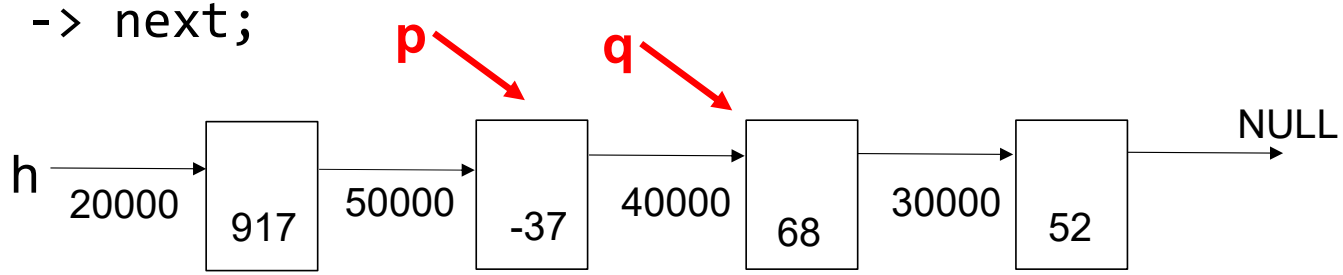
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



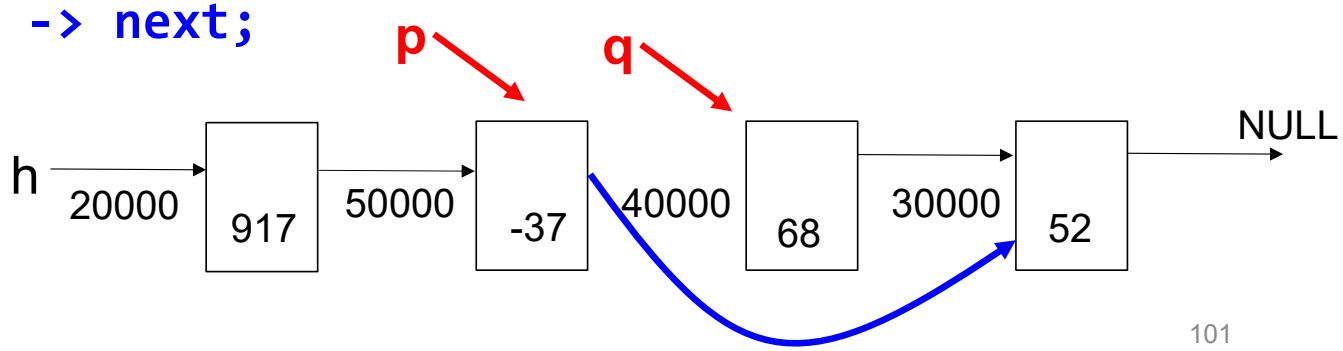
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



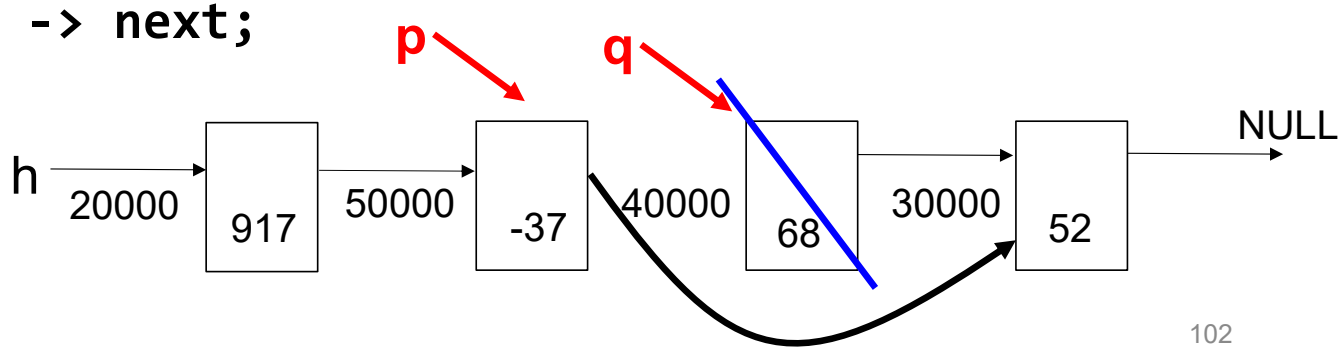
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



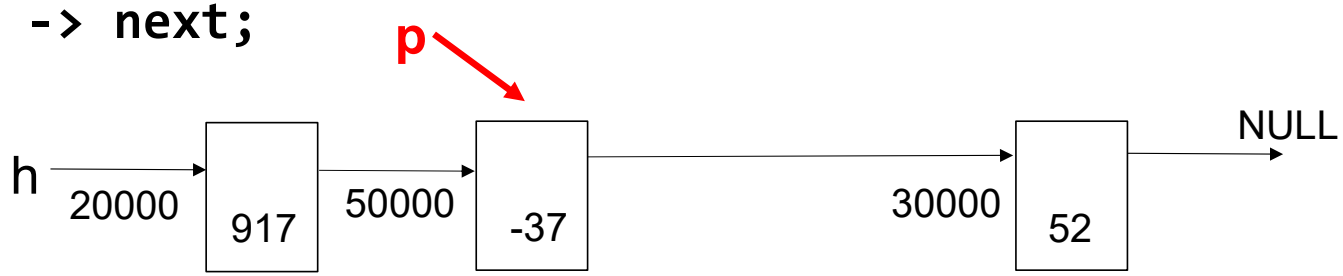
Suppose we want to delete the node that stores 68

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```



Suppose we want to delete the node that stores 68

```
Node * p = h;
Node * q = p -> next;
while ((q != NULL) && ((q -> value) != v))
  {
    p = p -> next;
    q = q -> next;
  }
if (q != NULL) // if q is NULL, v is not in the linked list
  {
    p -> next = q -> next;
    free (q);
  }
return h;
}
```



Delete a Node in a Linked List

- If the list is empty (NULL), do nothing, return NULL
- If the node to delete is the first node:
  - Save the second node
  - Free the first node
  - Return the second node (now is the first node)
- If the node to delete is not the first node:
  - Find the node to be deleted and the node in front of it
  - Bypass the node to be deleted
  - Free the node
  - Return the original first node

Common Questions

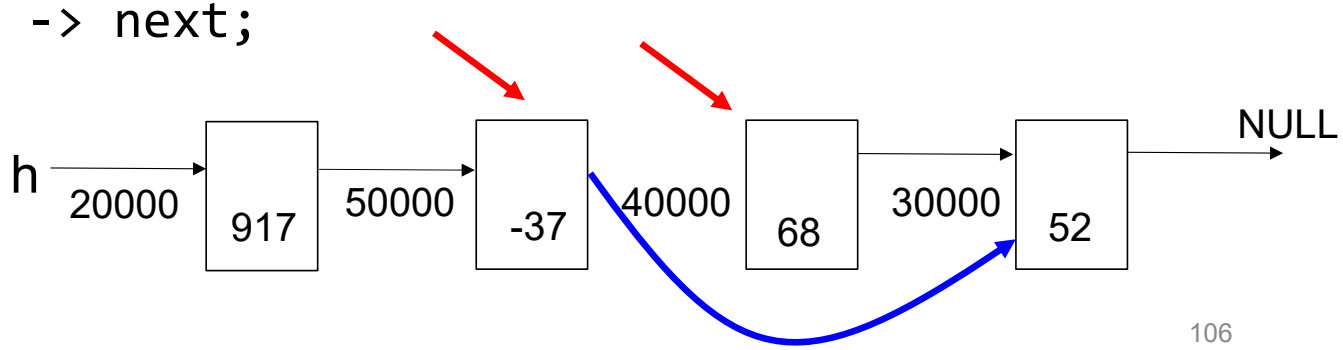
```
/* delete the first node (i.e. head)? */
 if ((h -> value) == v)
   {
     Node * p = h -> next;
     free (h);
     return p;
   }
```

Can the order be changed? No
After free (h), h -> next does not exist
return p stops this function and return to caller

```
Node * p = h;
Node * q = p -> next;
```

Do I need h, p, and q? Yes
h: first; q: to be deleted; p: before q

```
while ((q != NULL) && ((q -> value) != v))
  {
    p = p -> next;
    q = q -> next;
  }
if (q != NULL) // if q is NULL, v is not in the linked list
  {
    p -> next = q -> next;
    free (q);
  }
return h;
}
```



```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

Can the order be changed? No
if q is NULL, q -> value does not exist

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

Can the order be changed? Yes

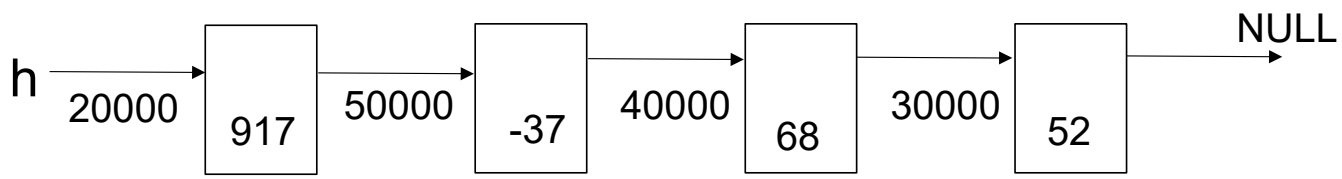
```
q = q -> next;
p = p -> next; // OK
```

```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

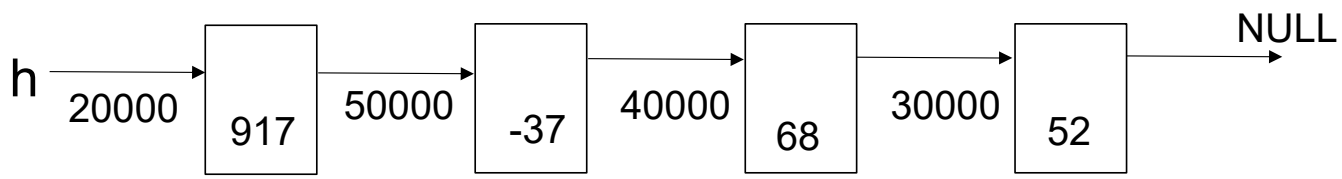
Can the order be changed? No
After free(q),
q-> next does not exist

Linked List 05

```
// print every node's value. do not change the linked list
void List_print(Node * h) // also called "traverse" the list
{
   while (h != NULL)
    {
      printf("%d ", h -> value);
      h = h -> next;
    }
  printf("\n\n");
}
```

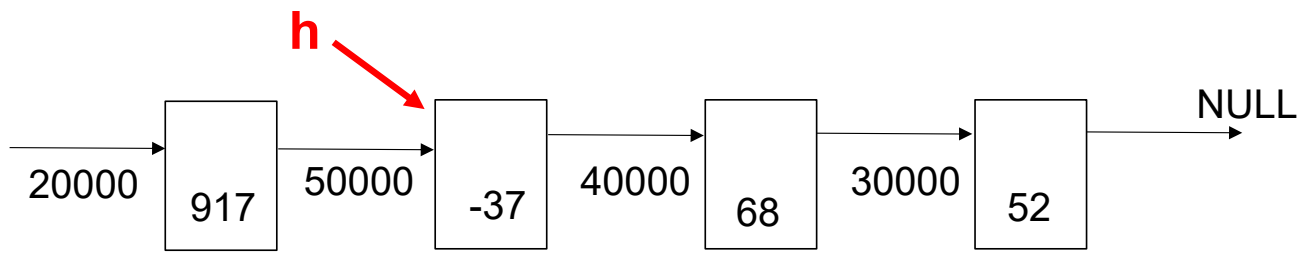


```
// print every node's value. do not change the linked list
void List_print(Node * h)
{
   while (h != NULL)
    {
      printf("%d ", h -> value);
      h = h -> next;
    }
  printf("\n\n");
}
```



```
// print every node's value. do not change the linked list
void List_print(Node * h)
{
   while (h != NULL)
    {
      printf("%d ", h -> value);
      h = h -> next;
    }
  printf("\n\n");
}
```

Is this a problem? No.
The caller still keeps the head of the list

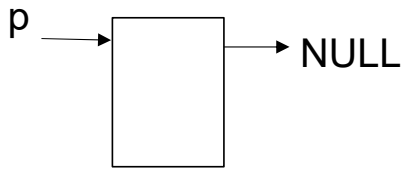


Review: Insert at the beginning

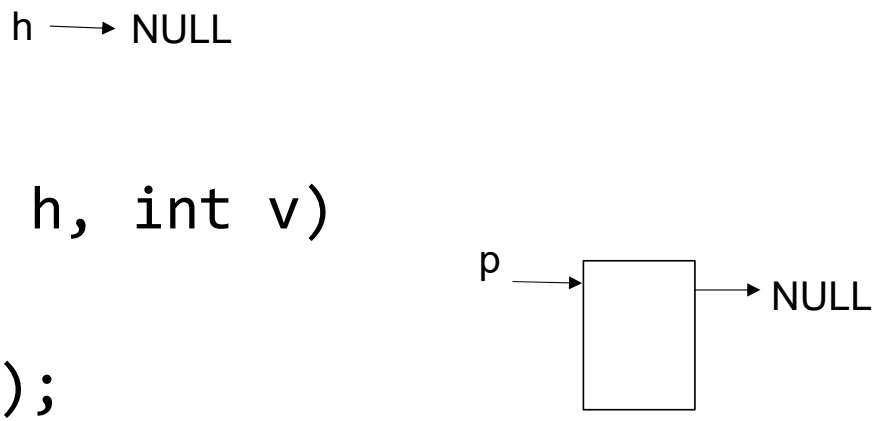
```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;   /* insert at the beginning */
  // this is a “stack”: first inserted node will
  // the last node
}
```

Insert at the end (create a “queue”)

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

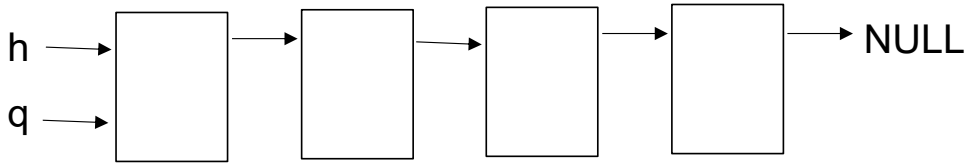


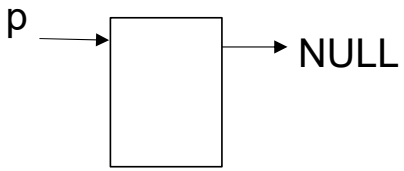
Insert at the end




```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

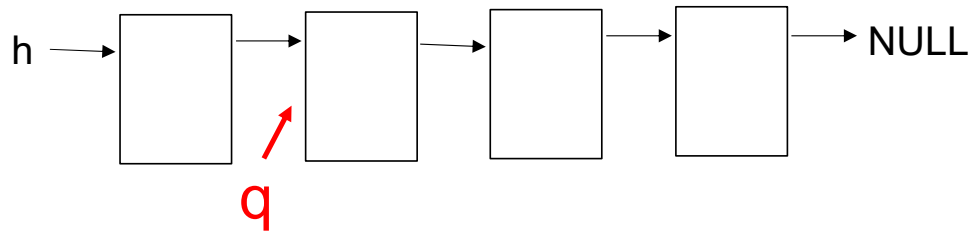
Insert at the end

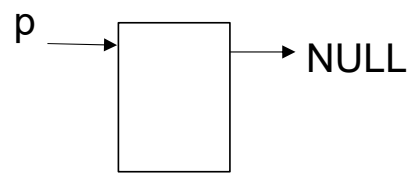





```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

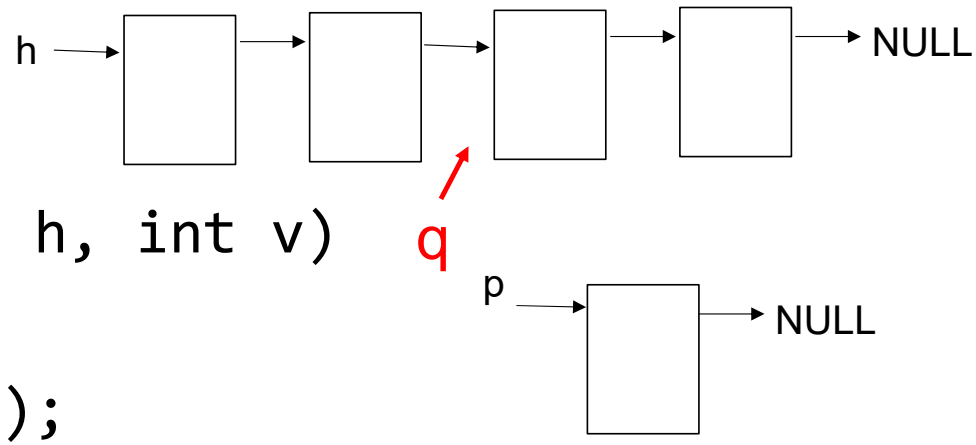
Insert at the end





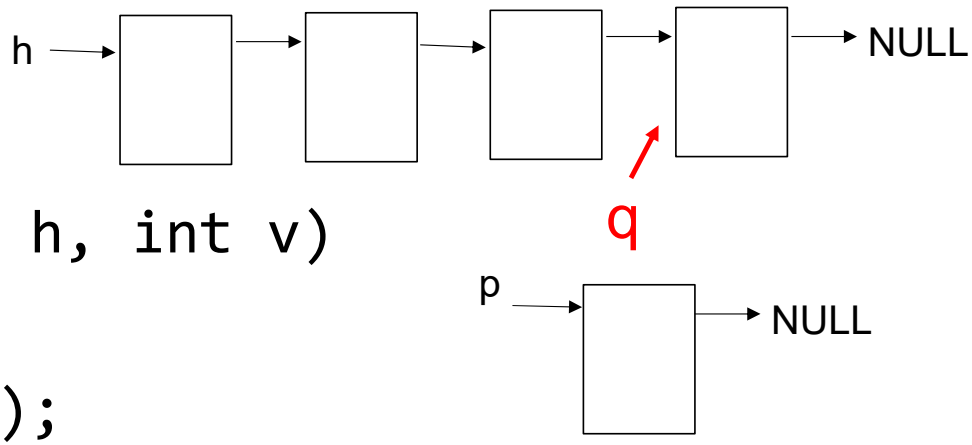

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

Insert at the end



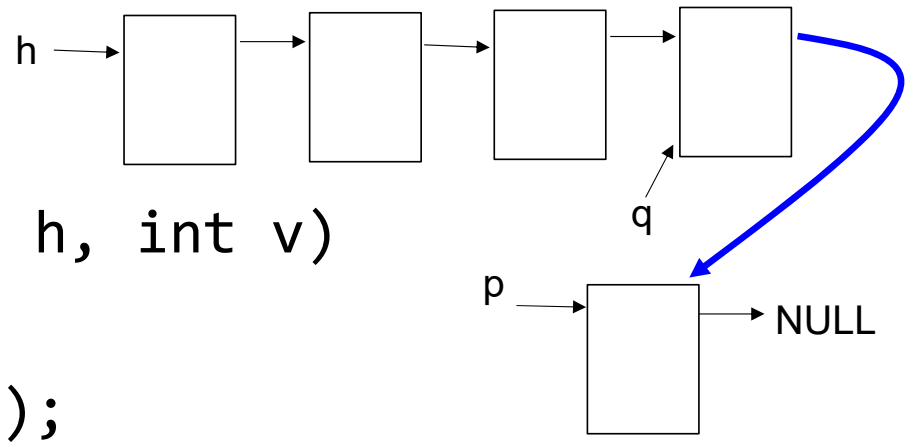

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

Insert at the end



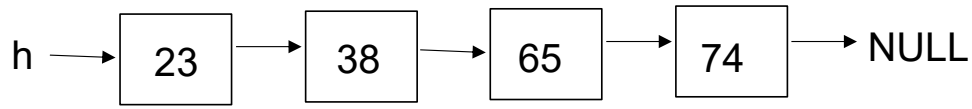

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

Insert at the end

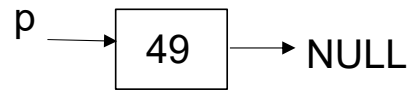


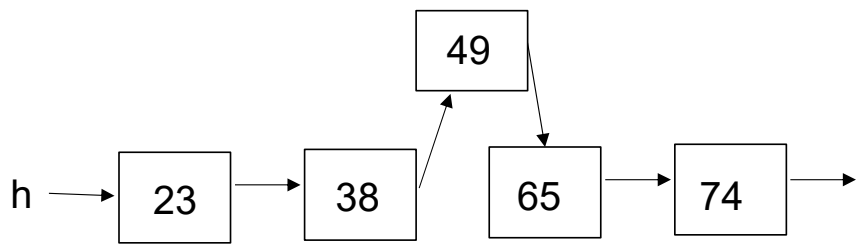

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

Question: Sort



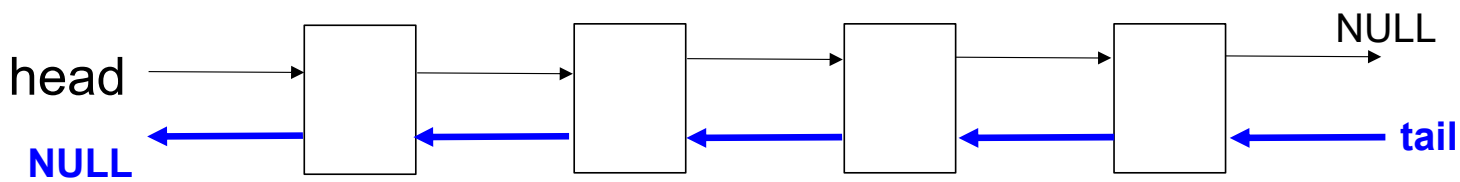

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  ????
}
```





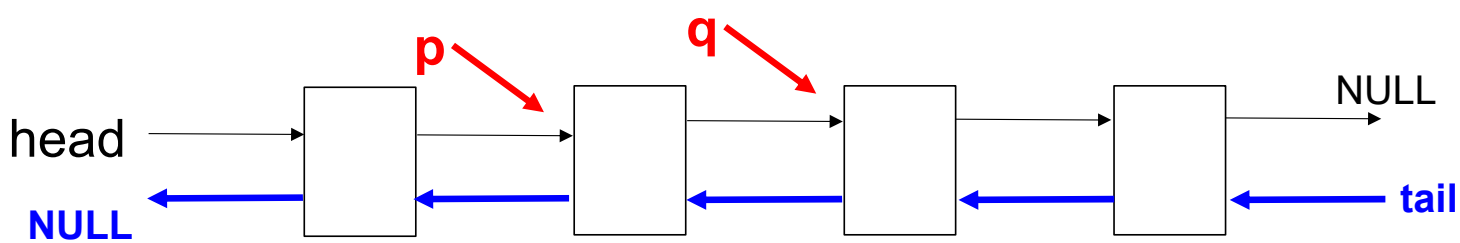
Doubly Linked List

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  struct listnode * prev; // must be a pointer
  // data
  // …
} Node;
```



Doubly Linked List

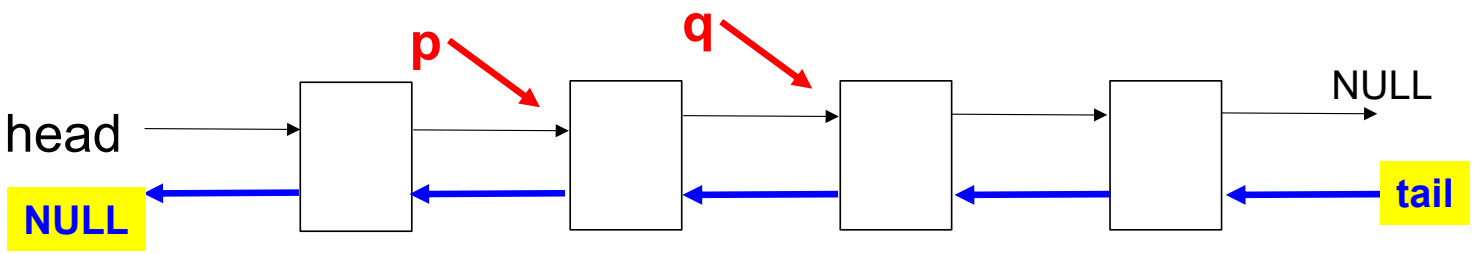
```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  struct listnode * prev; // must be a pointer
  // data
  // …
} Node;
```



If p -> next is q, then
q -> prev is p

Doubly Linked List

```
typedef struct listnode
{
  struct listnode * next; // must be a pointer
  struct listnode * prev; // must be a pointer
  // data
  // …
} Node;
```



Advantage of Doubly Linked List

- It can go forward and backward
- Inserting at the end is fast
- Inserting in the middle no real advantage in speed
- Still one-dimensional, not two-dimensional like binary tree