ECE 264 Spring 2023

Advanced C Programming

Aravind Machiry
Purdue University

Linked List

Dynamic Structures

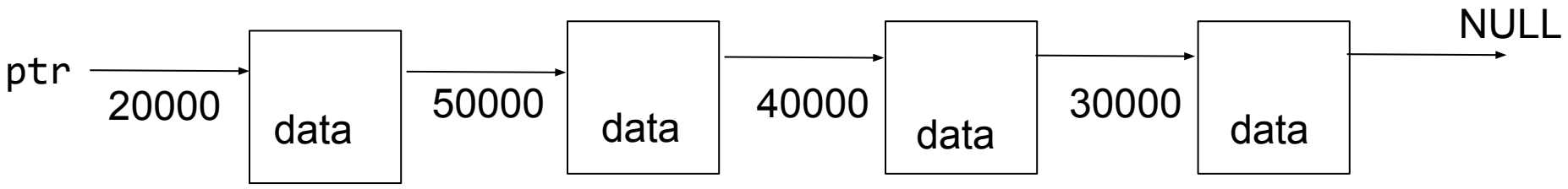
- Memory management:
  - Allocate memory when writing a program
  - Allocate memory after a program starts. Free before the program ends
- Allocate memory when needed. Free when no longer needed.
- Dynamic structures are used widely for problems whose sizes may change over time: database, web users,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

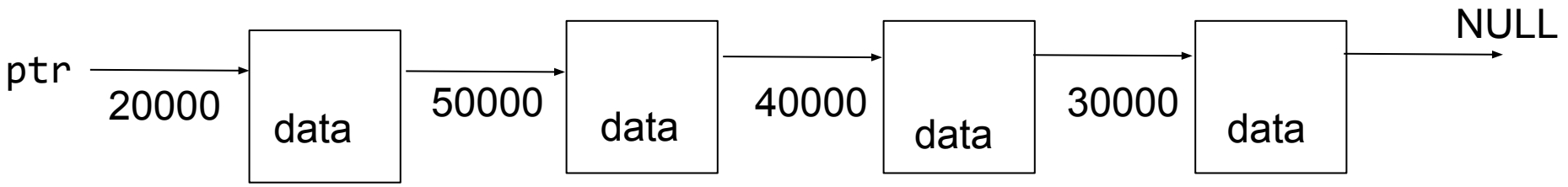


Why Heap or Stack Memory

- 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***.

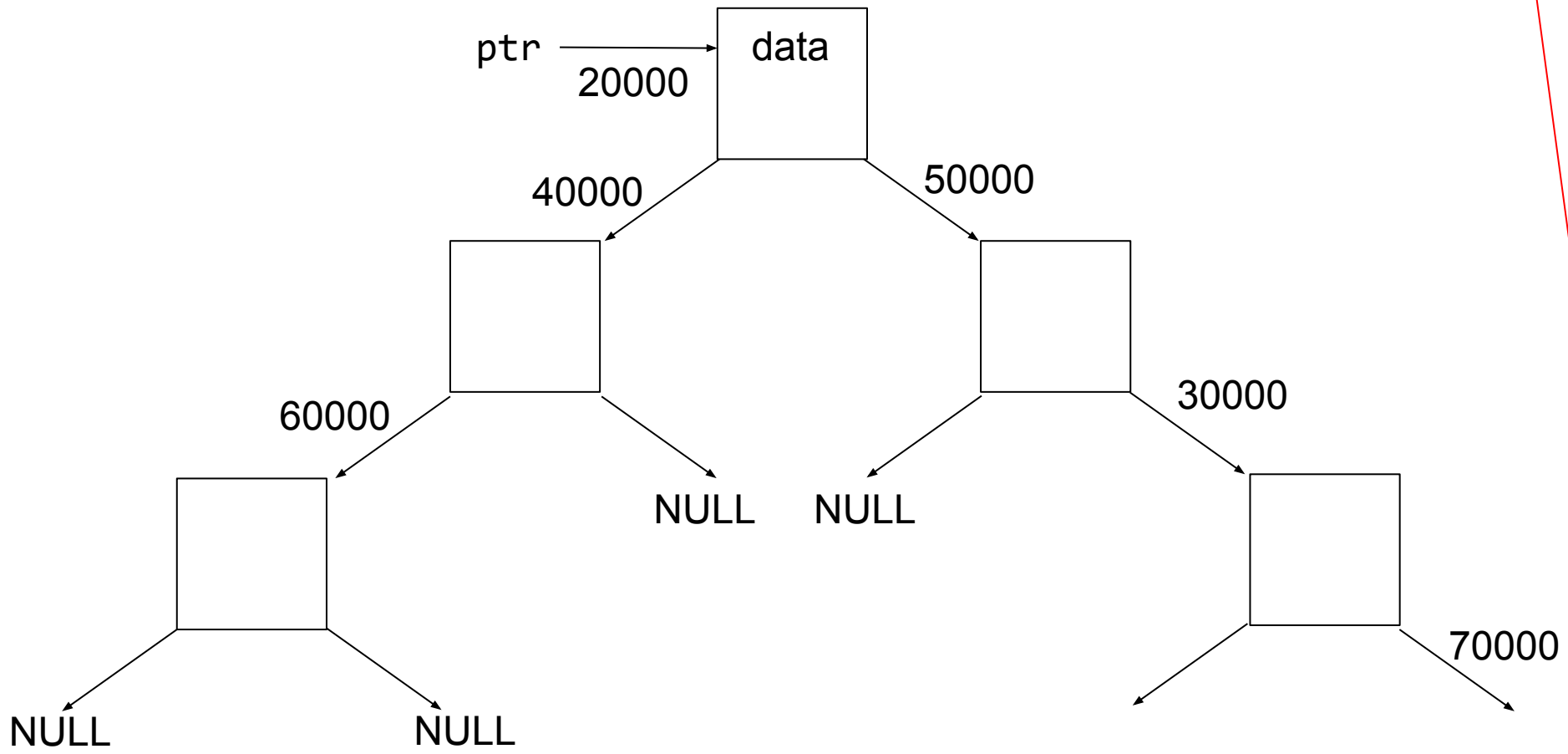


Two Pointers (binary tree)

- Each piece of memory has two pointers

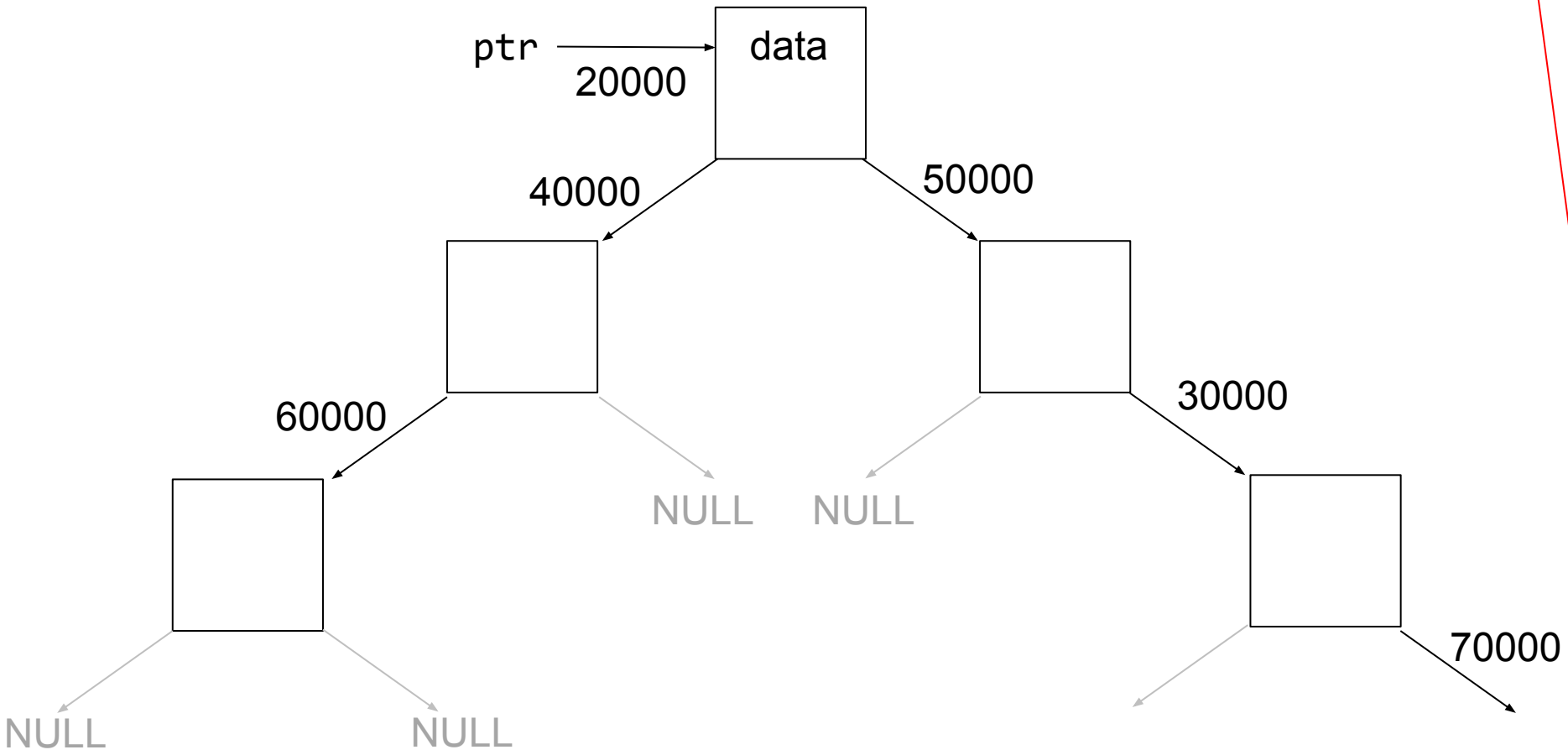
Stack Memory		
	Address	Value
ptr	100	20000

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

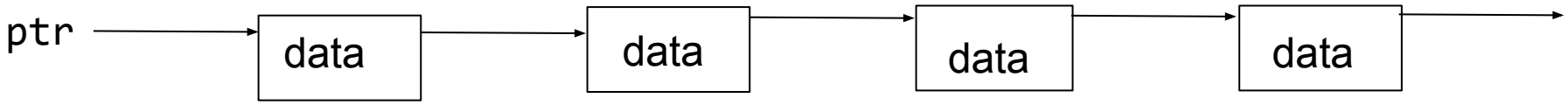


Stack Memory		
	Address	Value
ptr	100	20000

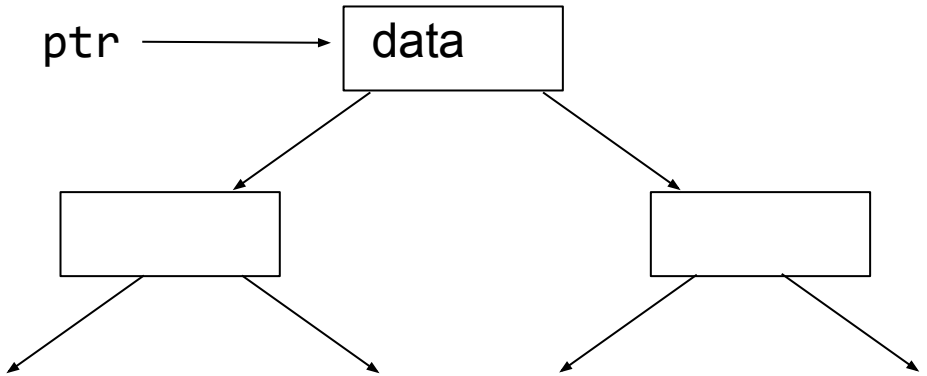
Heap Memory	
Address	Value
	data
	NULL
70000	NULL
	data
	NULL
60000	NULL
	data
	30000
50000	NULL
	data
	NULL
40000	60000
	data
	70000
30000	NULL
	data
	50000
20000	40000

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

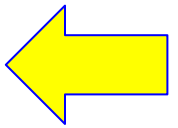
Linked List

```
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

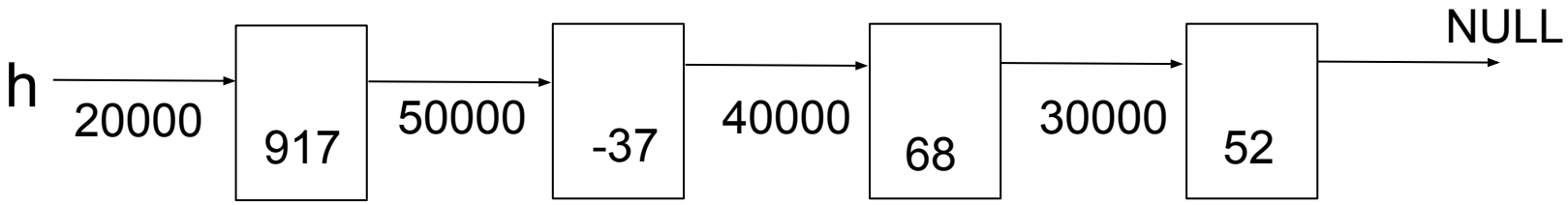
```
Node * head = NULL; /* must initialize it to NULL */
head = List_insert(head, 917);
head = List_insert(head, -504);
head = List_insert(head, 326);
```

```
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);
```

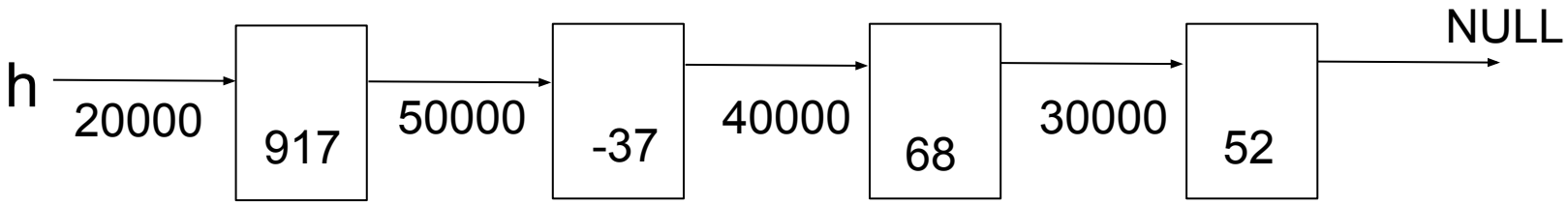
must set to NULL

Frame	Symbol	Address	Value
main	head	200	NULL

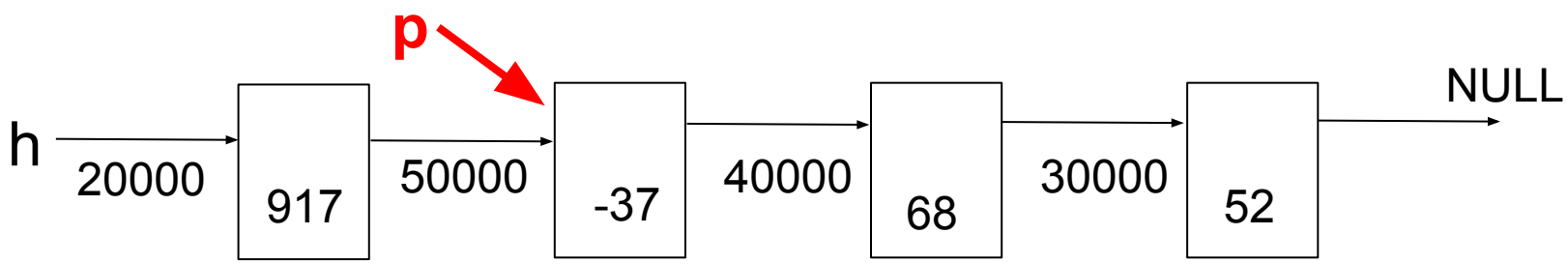
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
…
}
```



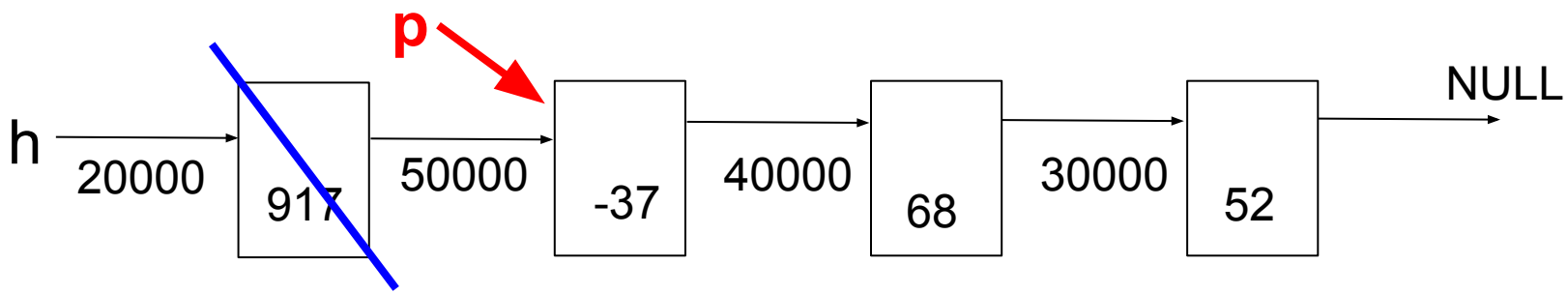
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    // almost every function start with checking NULL
    // if h is NULL, h -> next does not exist
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



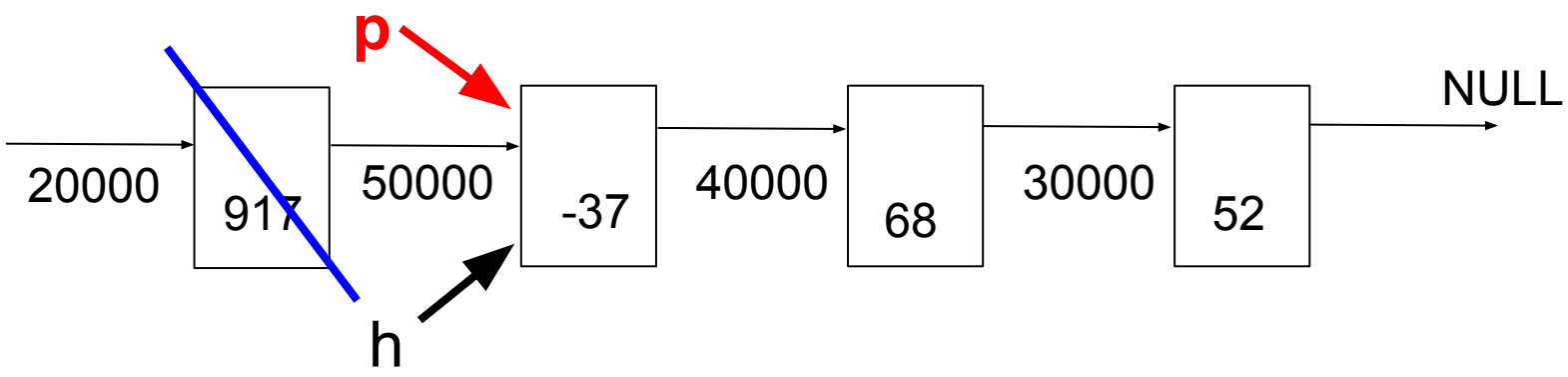
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



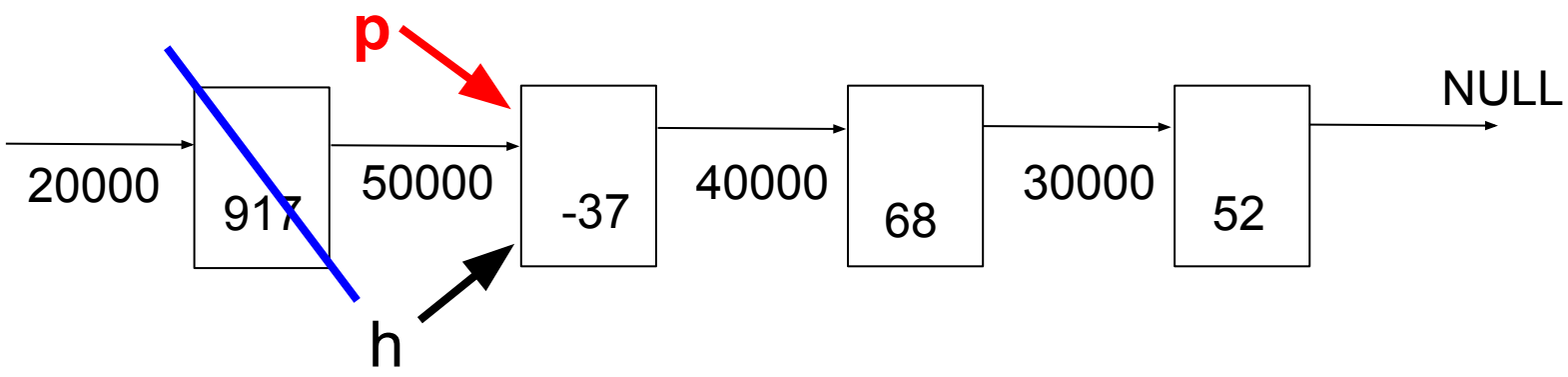
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



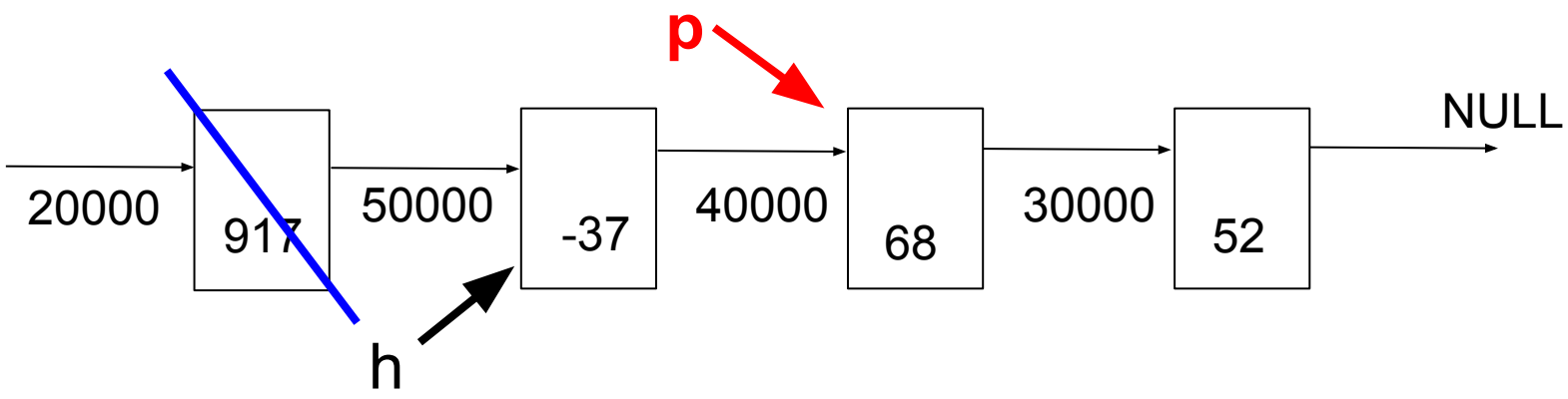
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



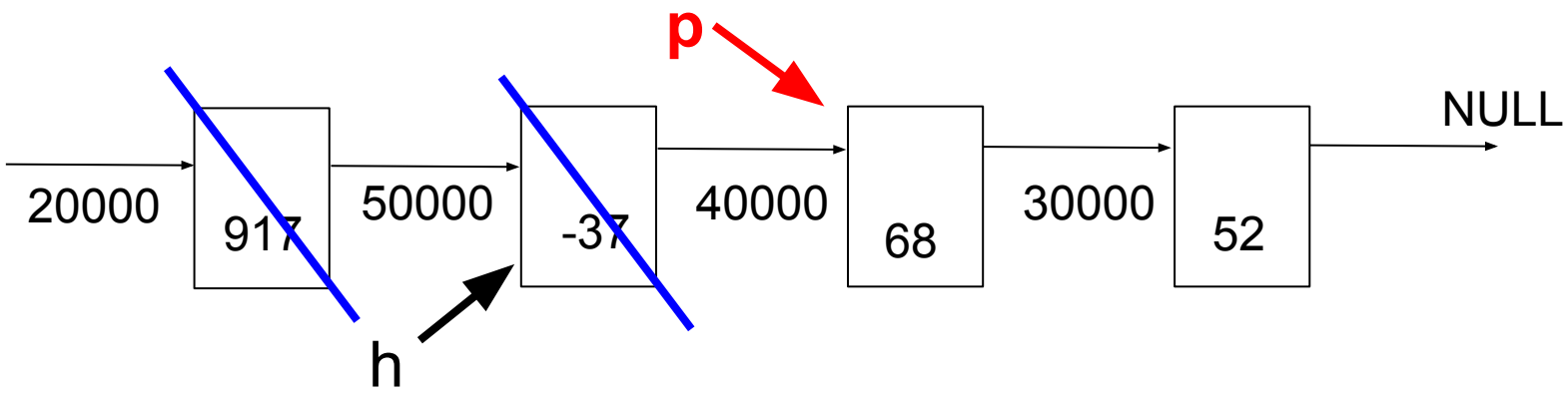
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



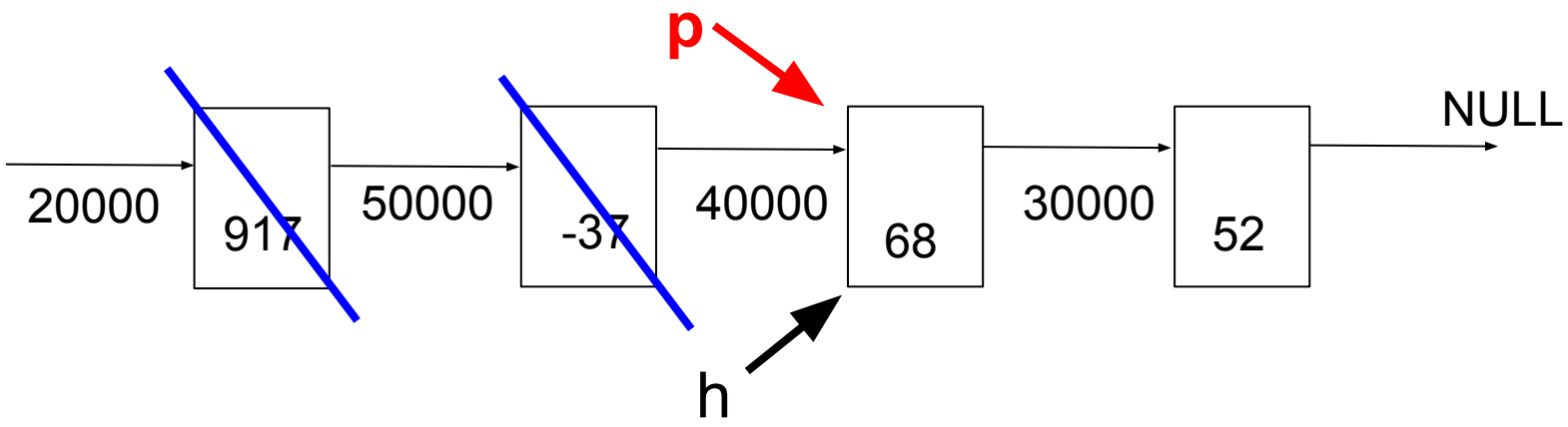
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



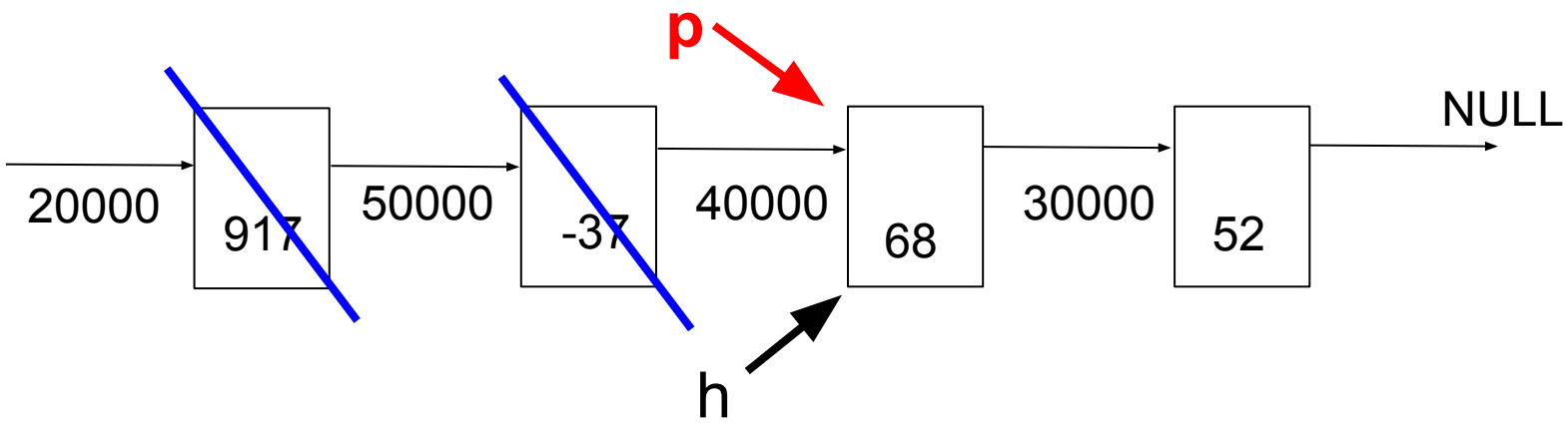
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



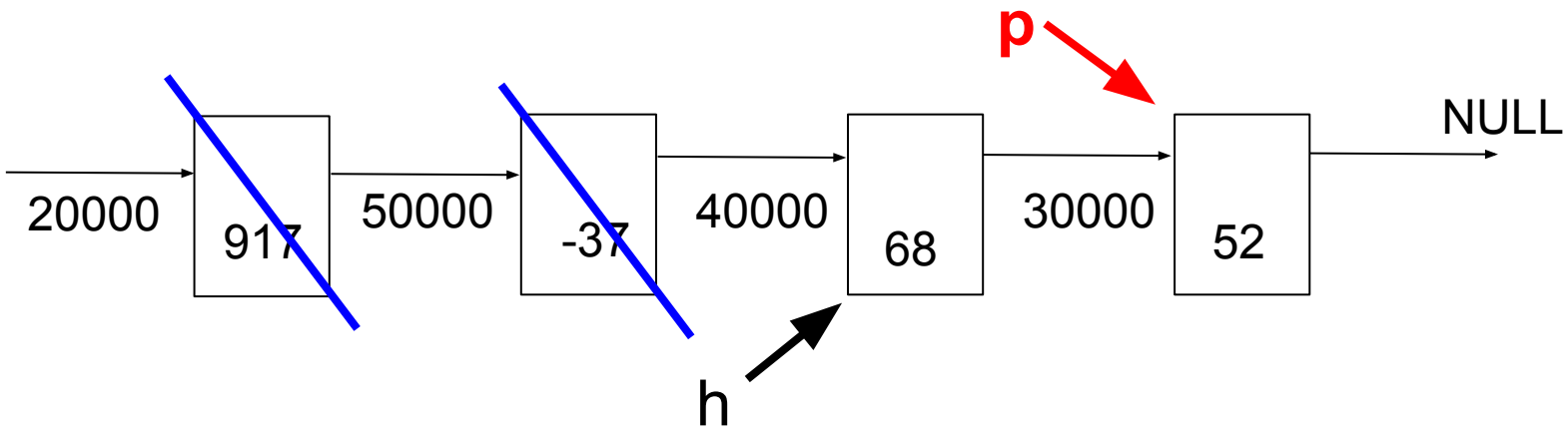
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



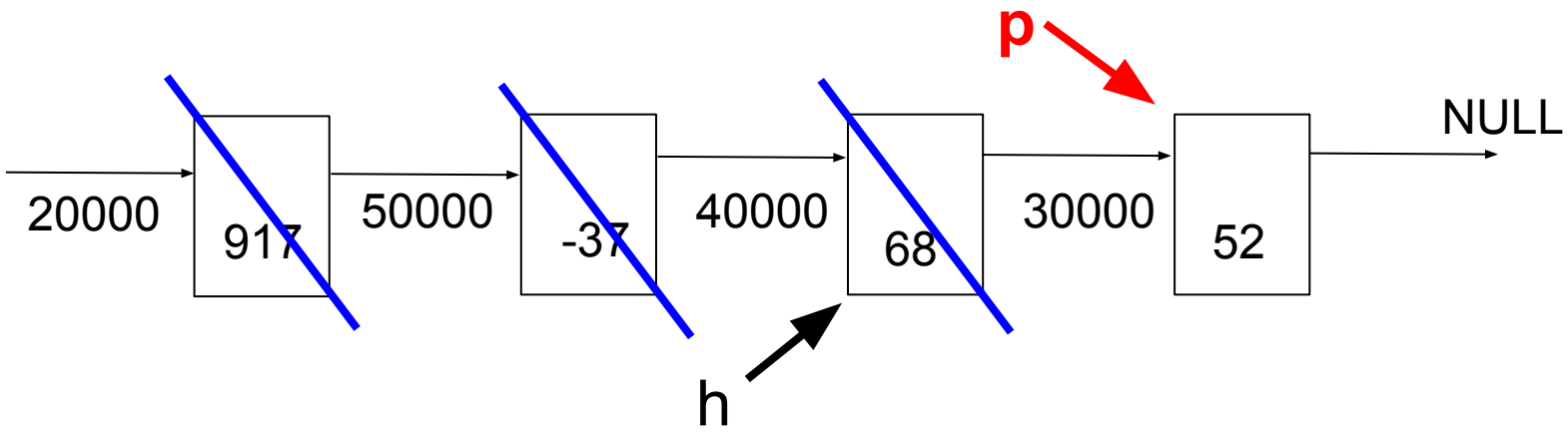
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



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

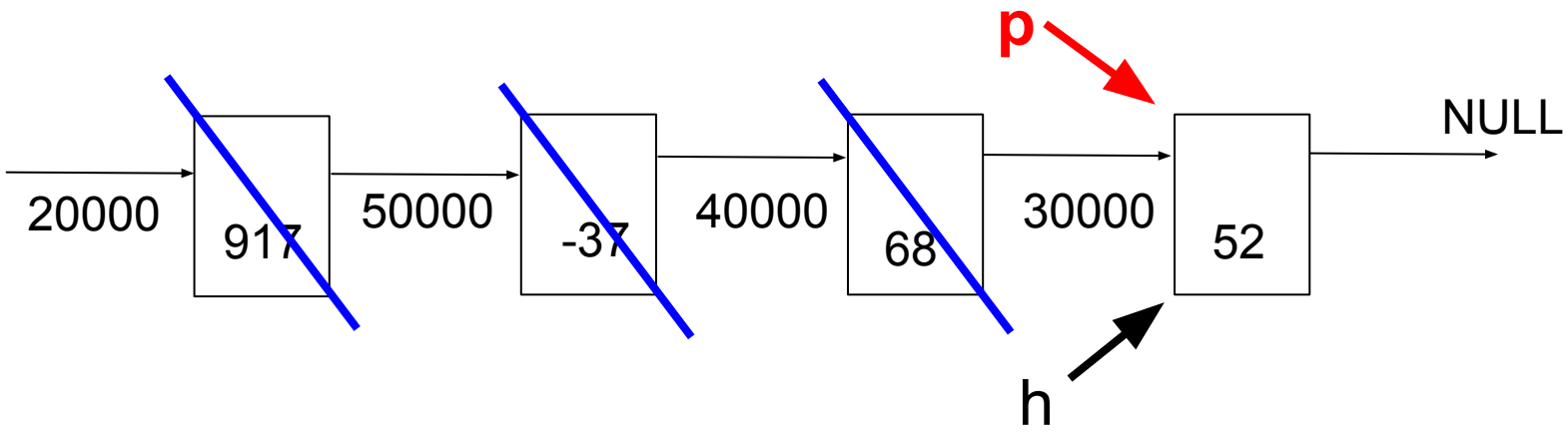


```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```

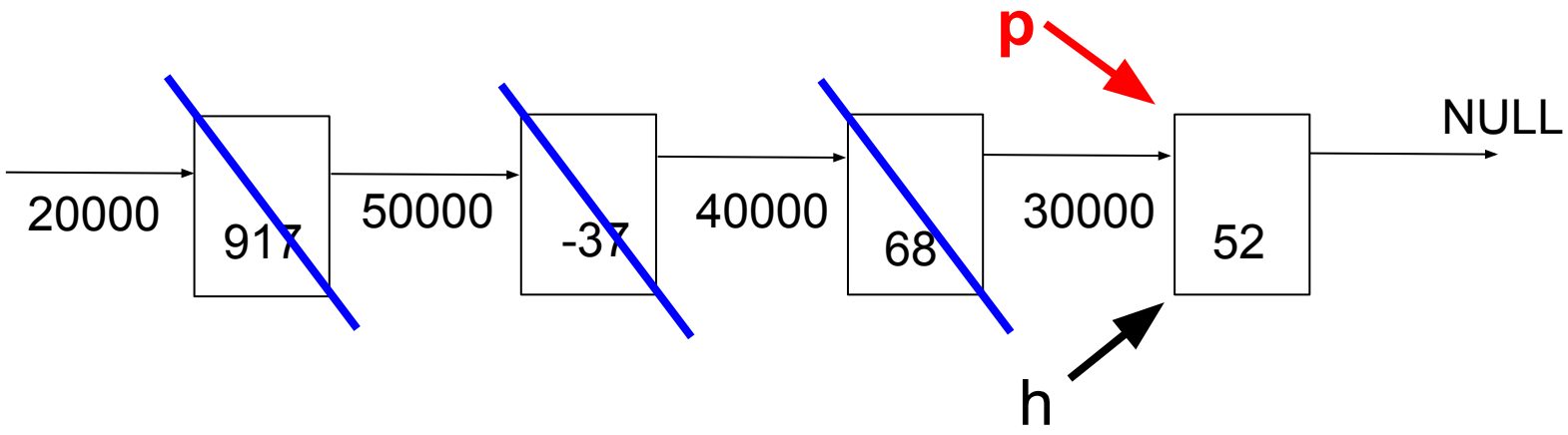


yunglu@purdue.edu

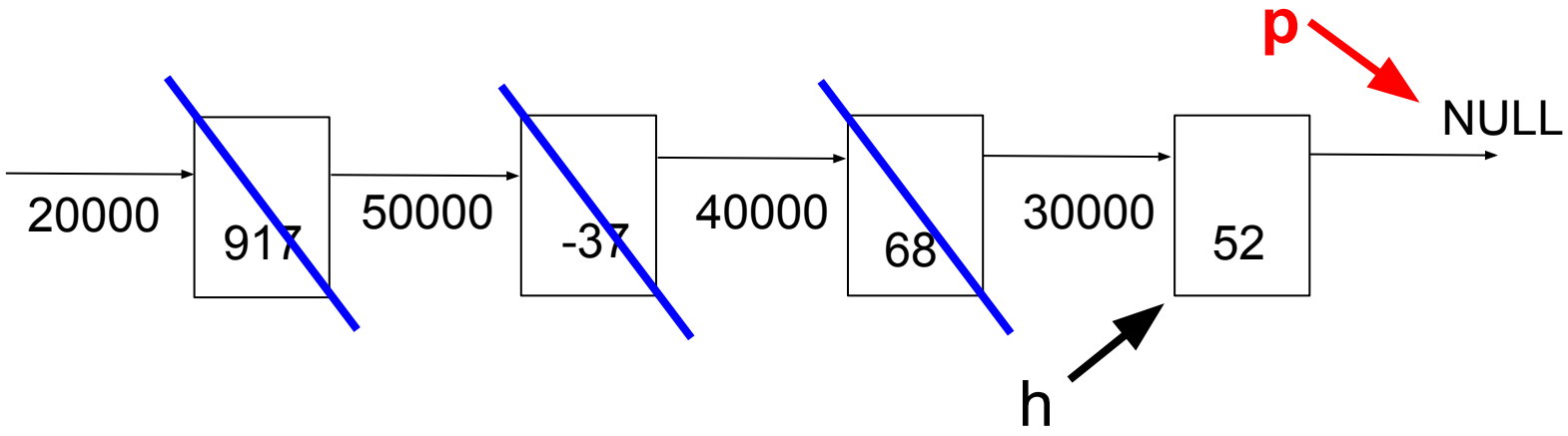
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



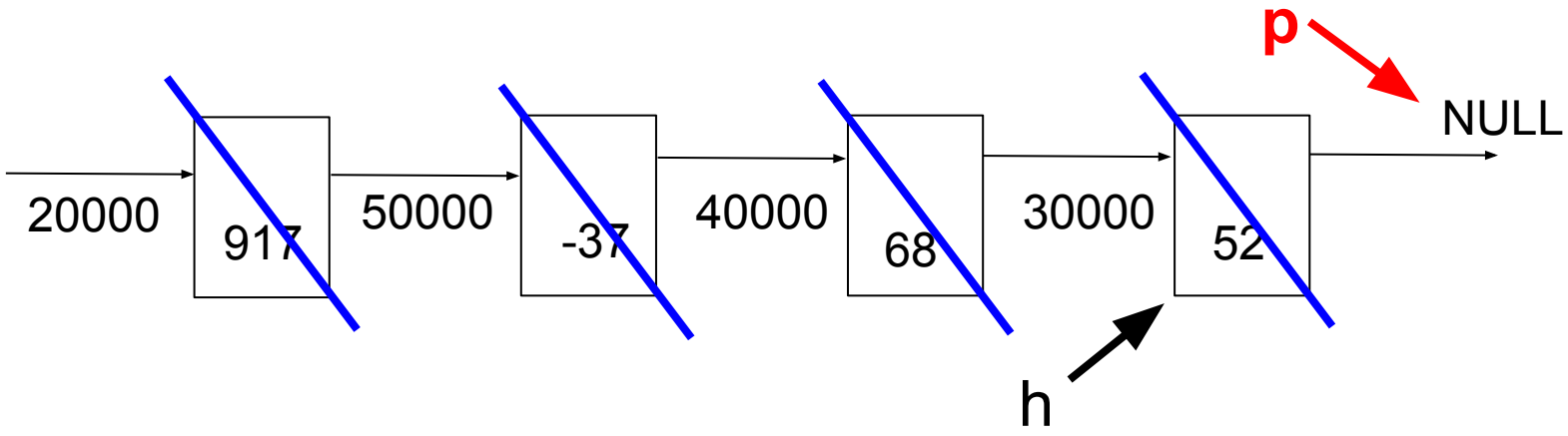
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



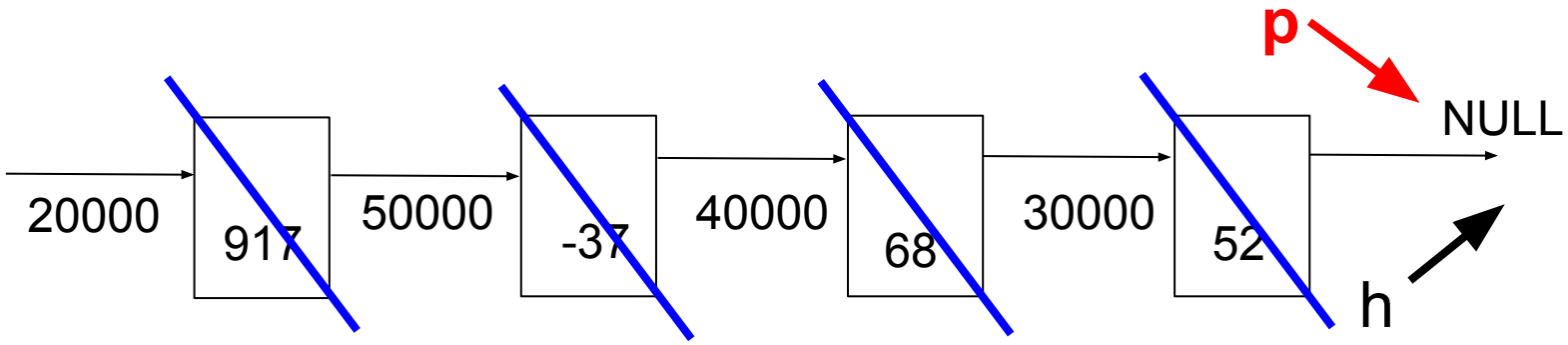
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



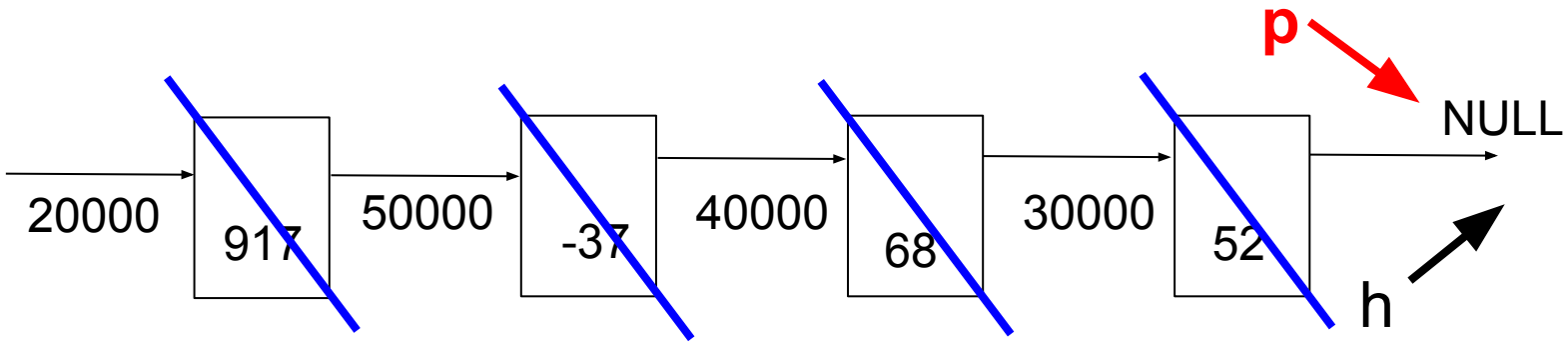
```
/* delete all nodes in a linked list*/
void List_destroy(Node * h)
{
  while (h != NULL)
    {
      Node * p = h -> next;
      free (h);
      h = p;
    }
}
```



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

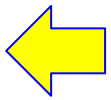
Can I move p's defection outside while? Yes

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

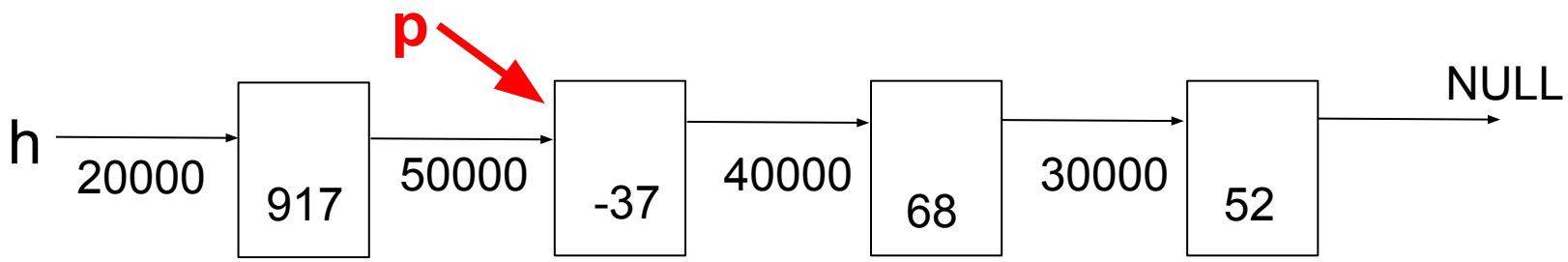
Delete a Node in a Linked List

- 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

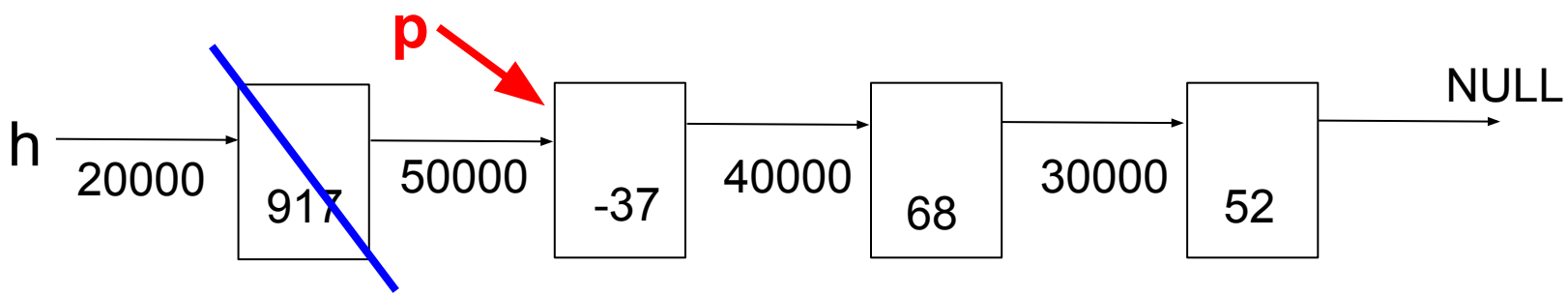
```
/* delete the node whose value is v in a linked list starting
with h, return the head of the remaining list, or NULL if the
list is empty. If multiple nodes contains v, delete the first
one. */

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;
  }
```

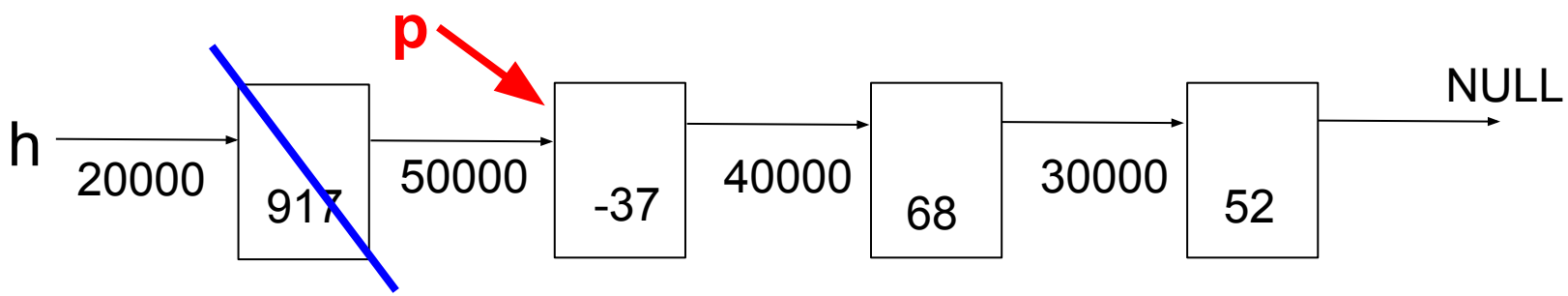
```
/* delete the first node (i.e. head)? */
if ((h -> value) == v)
  {                                  // p may be NULL, that's ok
    Node * p = h -> next;
    free (h);
    return p;
  }
```



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



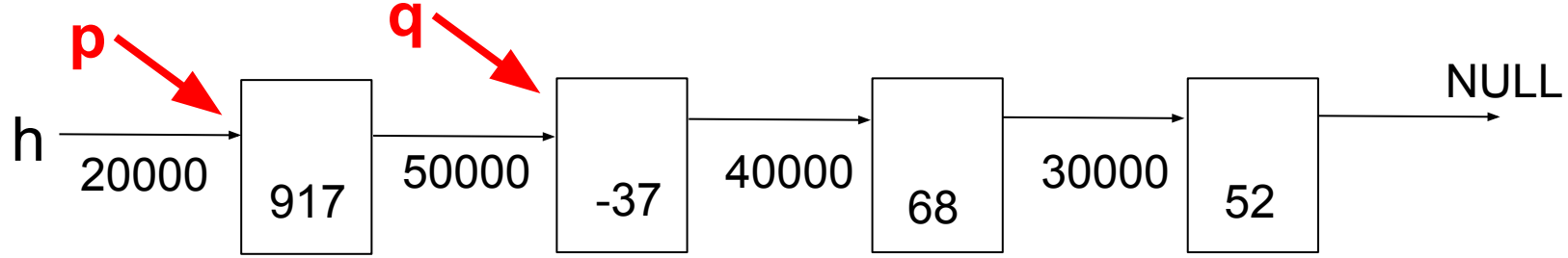
```
/* delete the first node (i.e. head)? */
if ((h -> value) == v)
  {
    Node * p = h -> next;
    free (h);
    return p;
  }
```



```
  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;
}
```

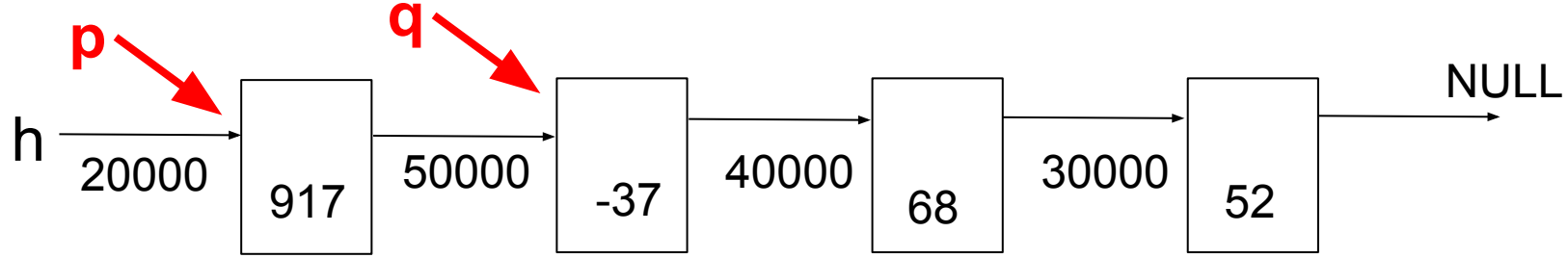
```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

Suppose we want to delete the node that stores 68



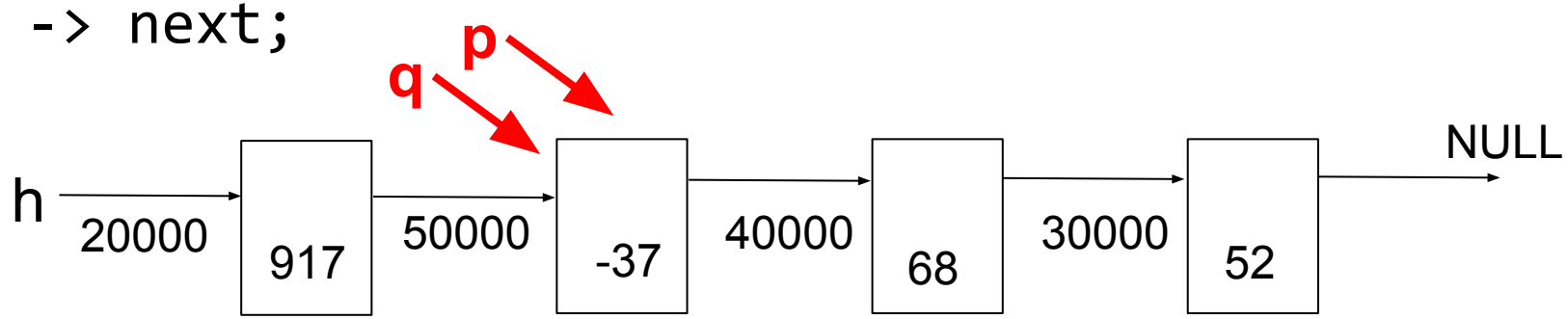
Suppose we want to delete the node that stores 68

```
  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;
}
```



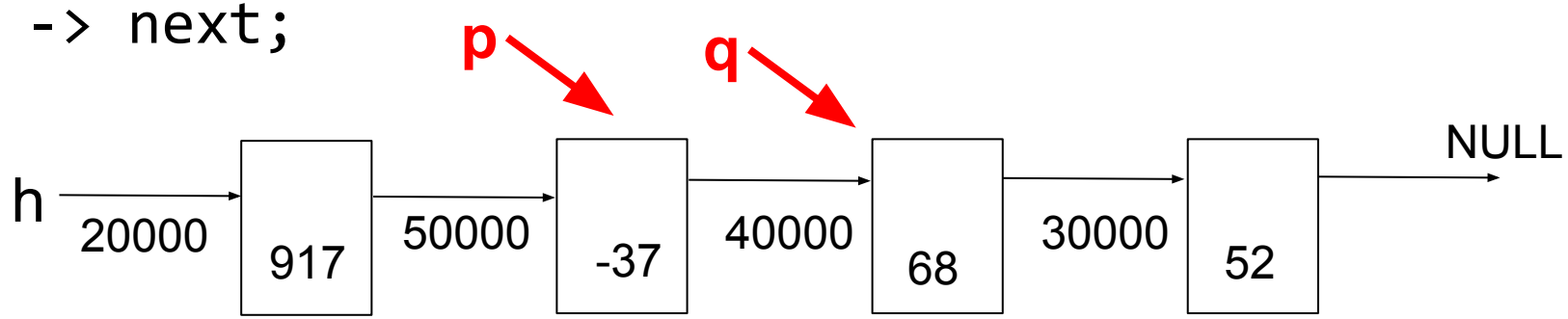
Suppose we want to delete the node that stores 68

```
  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;
}
```



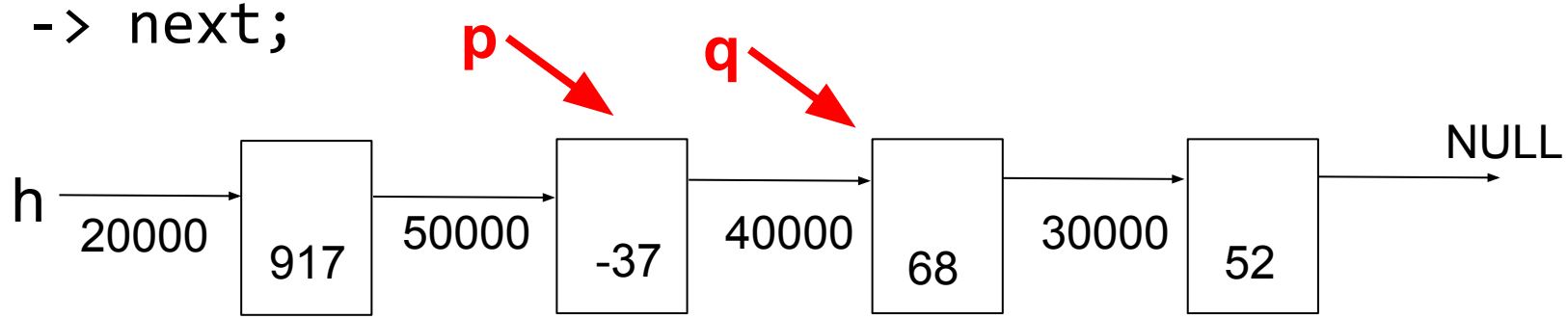
Suppose we want to delete the node that stores 68

```
  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;
}
```



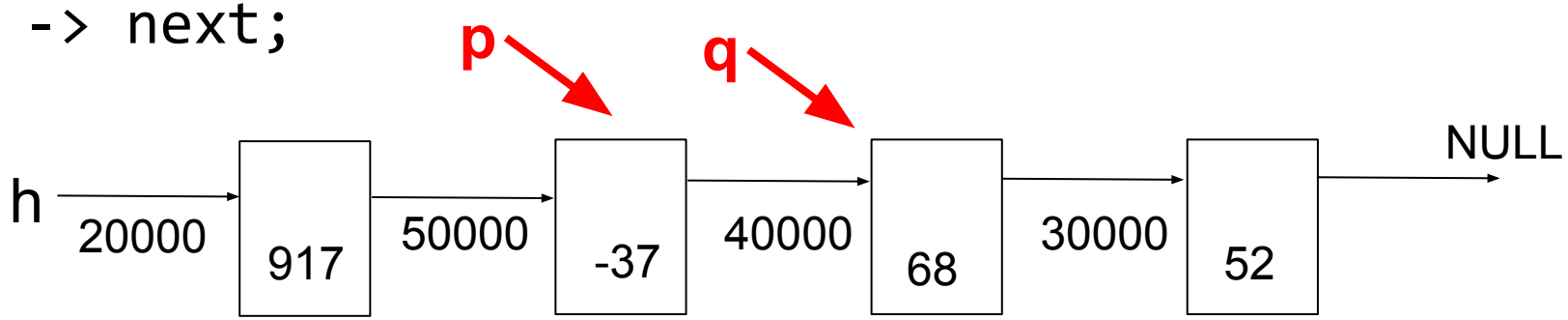
Suppose we want to delete the node that stores 68

```
  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;
}
```



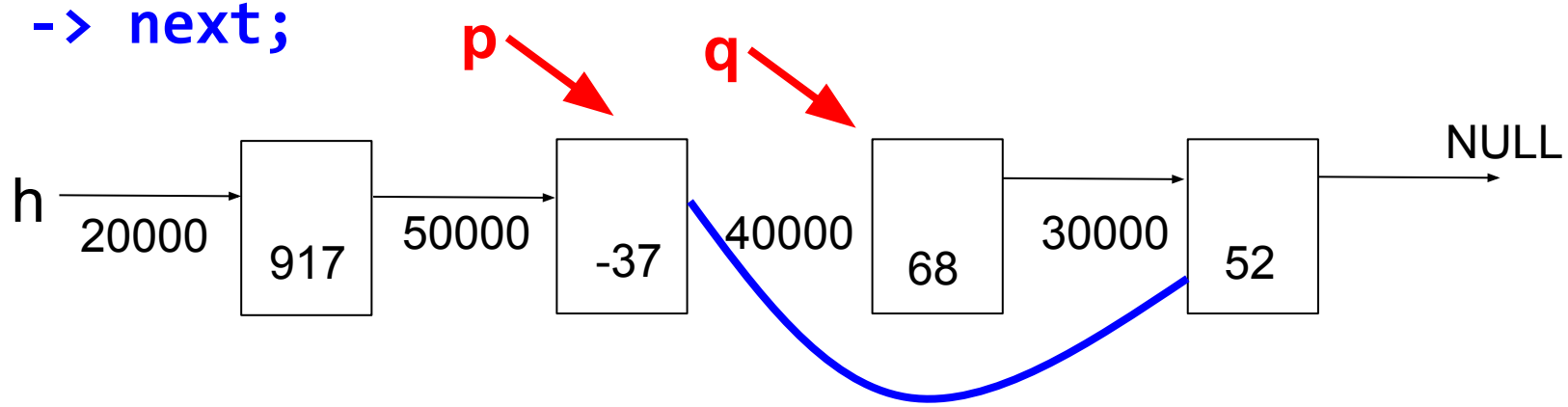
Suppose we want to delete the node that stores 68

```
    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;
}
```



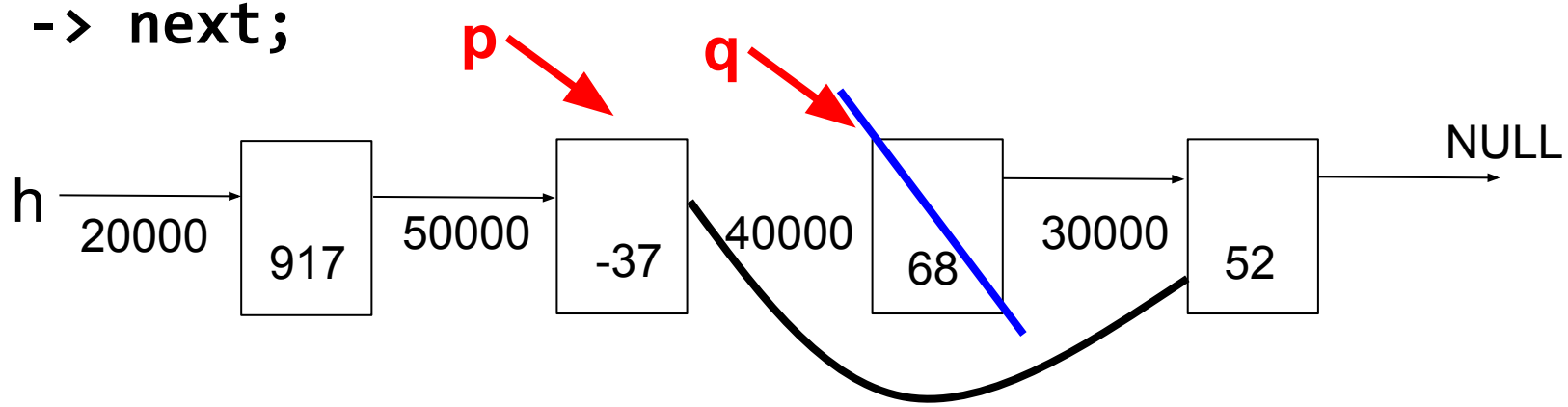
Suppose we want to delete the node that stores 68

```
  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;
}
```



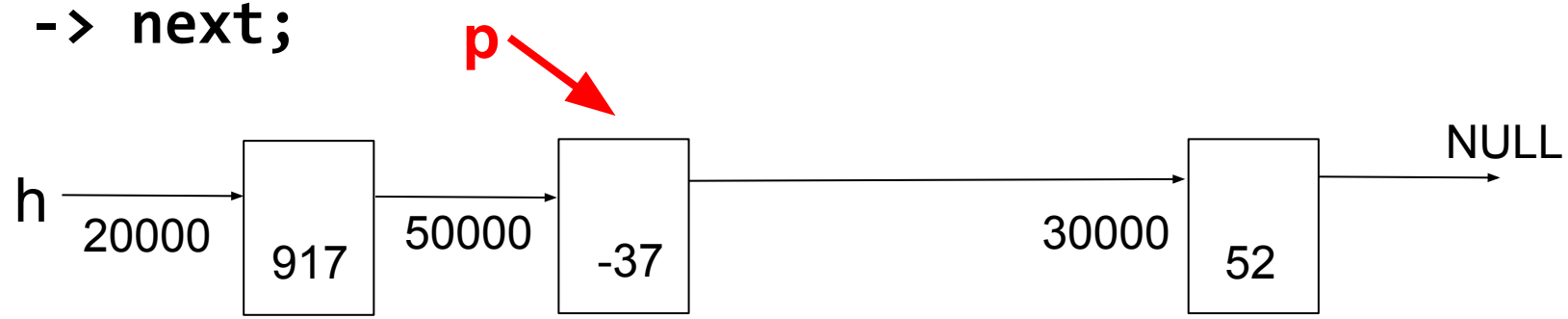
Suppose we want to delete the node that stores 68

```
  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

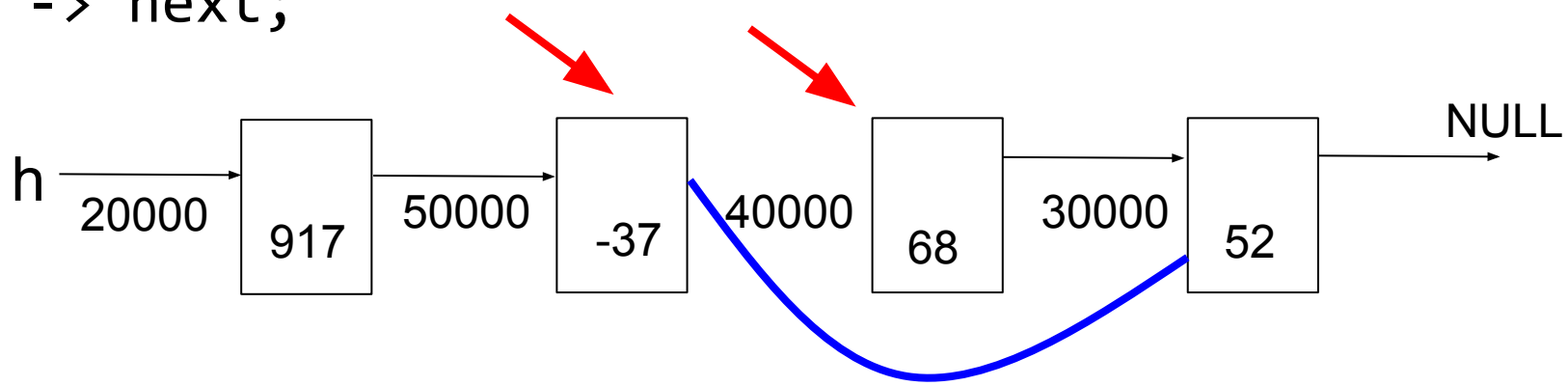
```
/* delete the first node (i.e. head)? */
 if ((h -> value) == v)
   {
     Node * p = h -> next;
     free (h);
     return p;
   }
```

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;
}
```

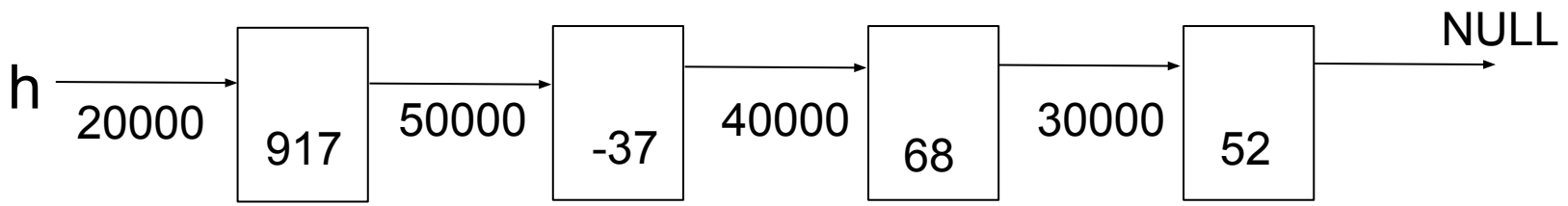
Can the order be changed? Yes

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

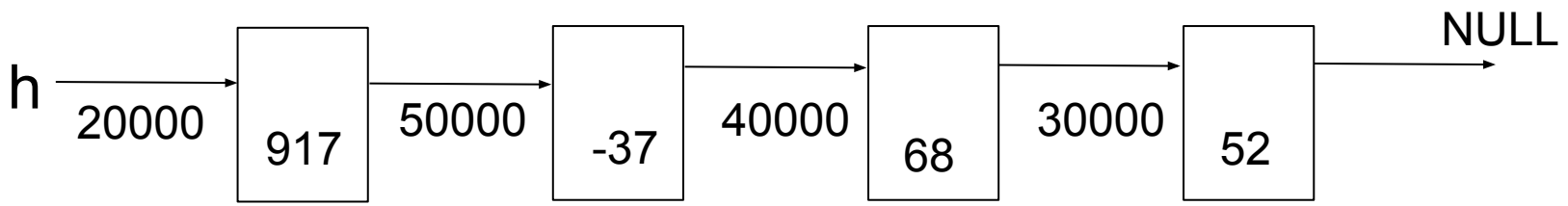
```
  Node * p = h;
  Node * q = p -> next;
  while ((q != NULL) && ((q -> value) != v))
    {
      p = p -> next;
      q = q -> next;
    }
  if (q != NULL) // if q is NULL, v is not in the linked list
    {
      p -> next = q -> next;
      free (q);
    }
  return h;
}
```

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

```
// 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");
}
```

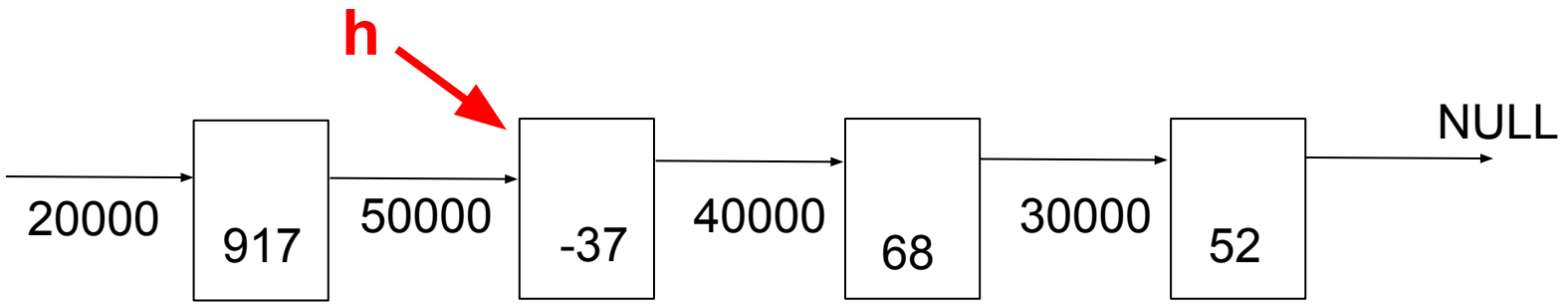


```
// print every node's value. do not change the linked list
void List_print(Node * h)
{
   while (h != NULL)
    {
      printf("%d ", h -> value);
      h = h -> next;
    }
  printf("\n\n");
}
```



```
// 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

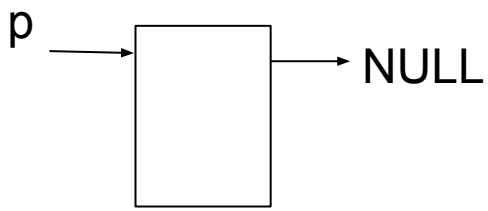


Review: Insert at the beginning

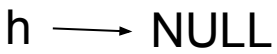
```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  p -> next = h;
  return p;   /* insert at the beginning */
  // this is a “stack”: first inserted node will
  // the last node
}
```

Insert at the end (create a “queue”)

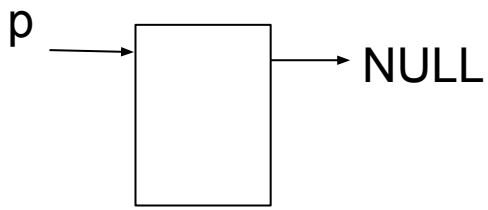
```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

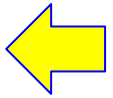


Insert at the end

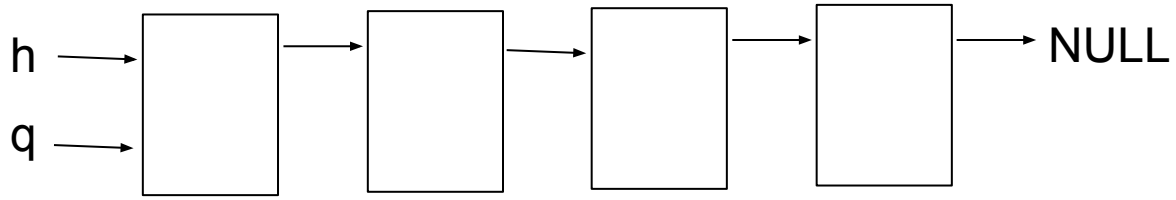



```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

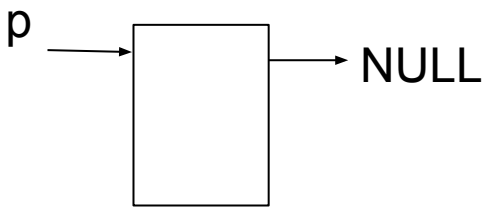


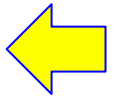


Insert at the end

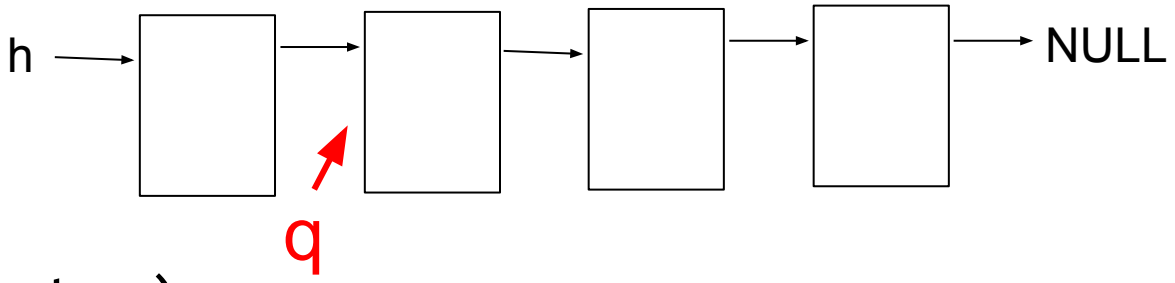



```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

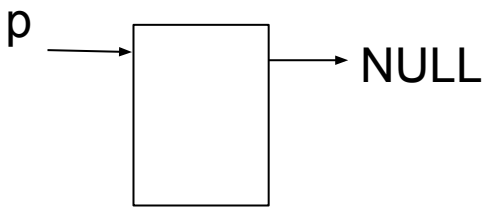


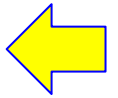


Insert at the end

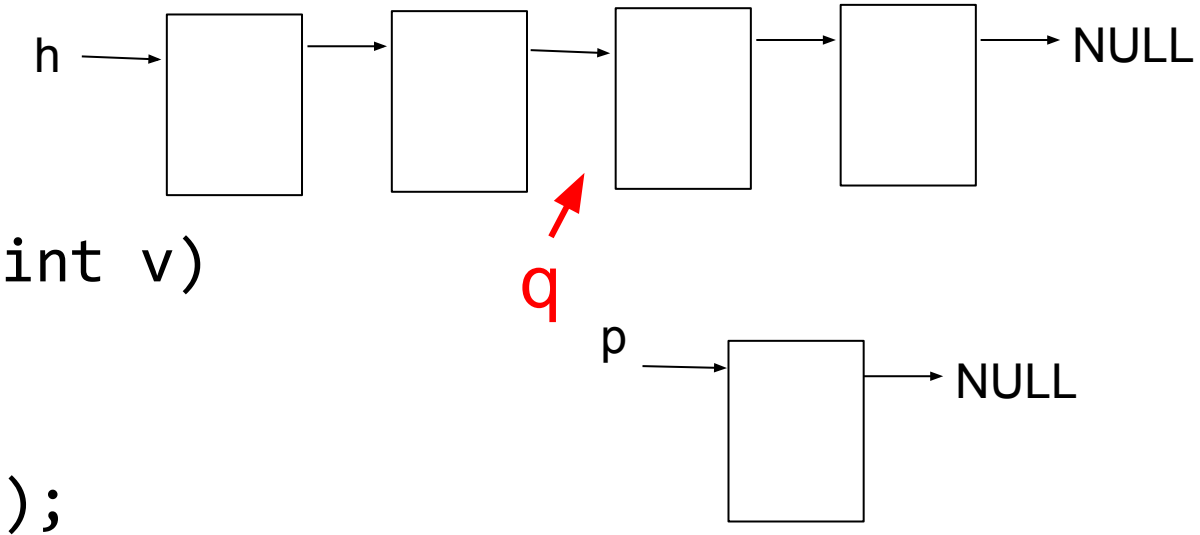



```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```



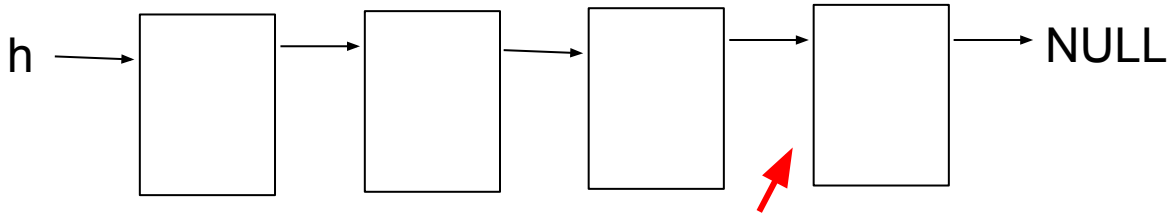


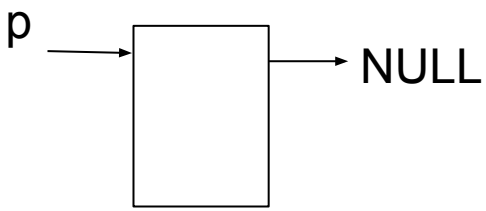
Insert at the end



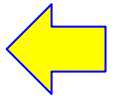

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

Insert at the end

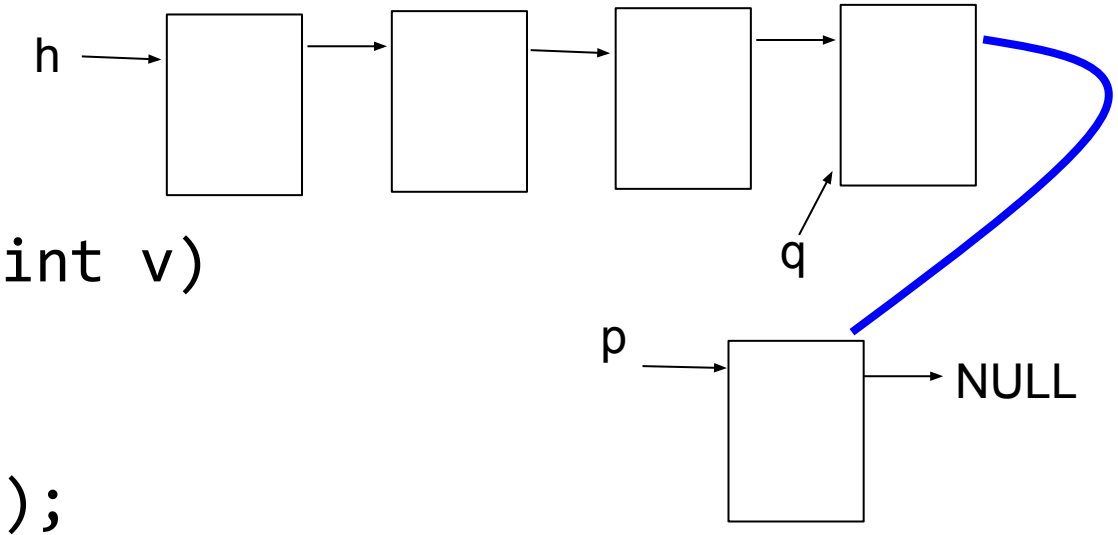




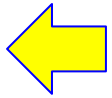

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```



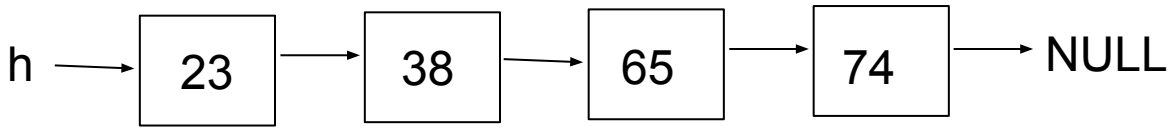
Insert at the end



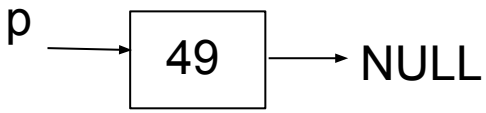

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  Node * q = h;
  while ((q -> next) != NULL) { q = q -> next; }
  q -> next = p;
  return h;
}
```

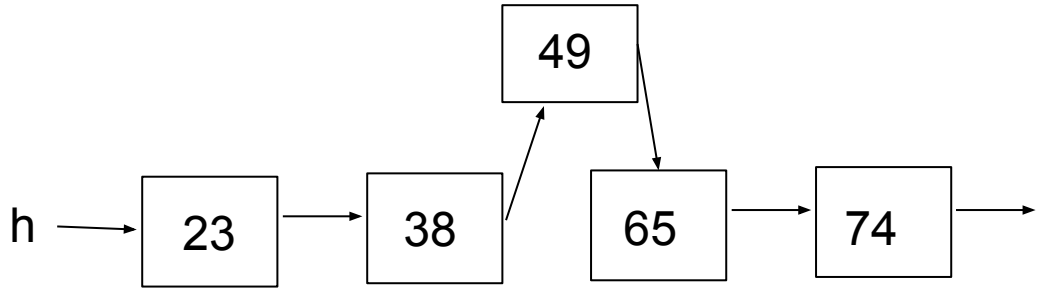


Question: Sort



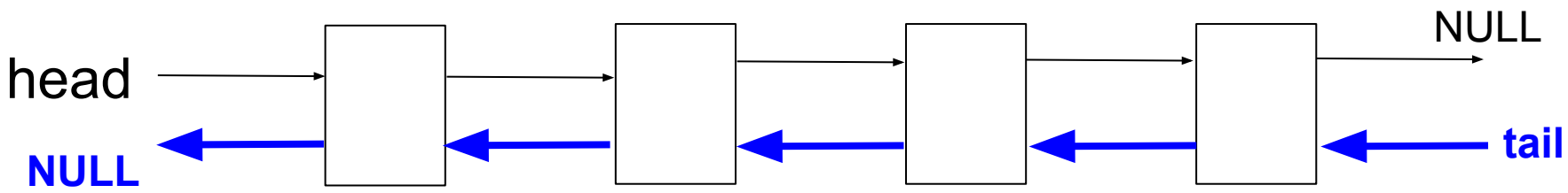

```
Node * List_insert(Node * h, int v)
{
  printf("insert %d\n", v);
  Node * p = Node_construct(v);
  if (h == NULL) { return p; } // first node
  ????
}
```





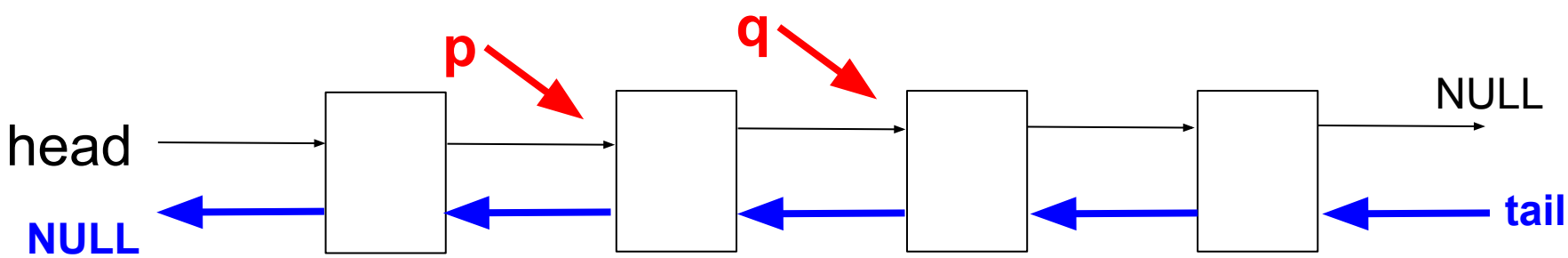
Doubly Linked List

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



Doubly Linked List

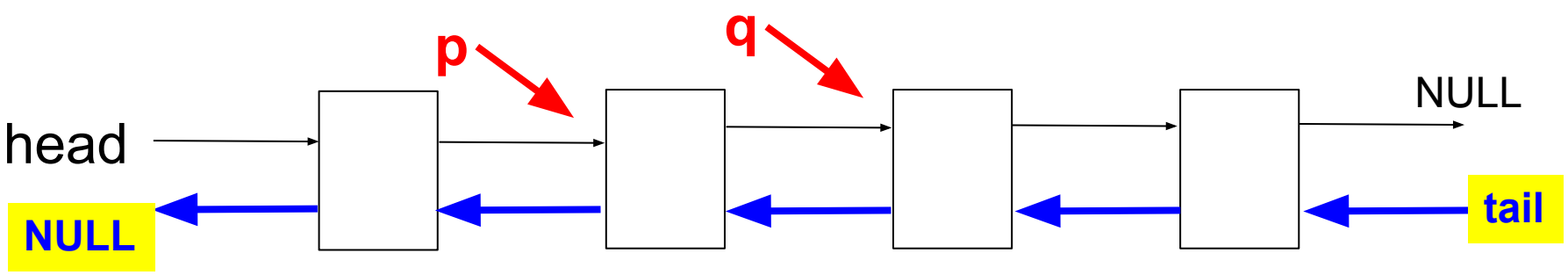
```
typedef struct listnode
{
   struct listnode * next; // must be a pointer
   struct listnode * prev; // must be a pointer
   // data
   // ...
} Node;
```



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

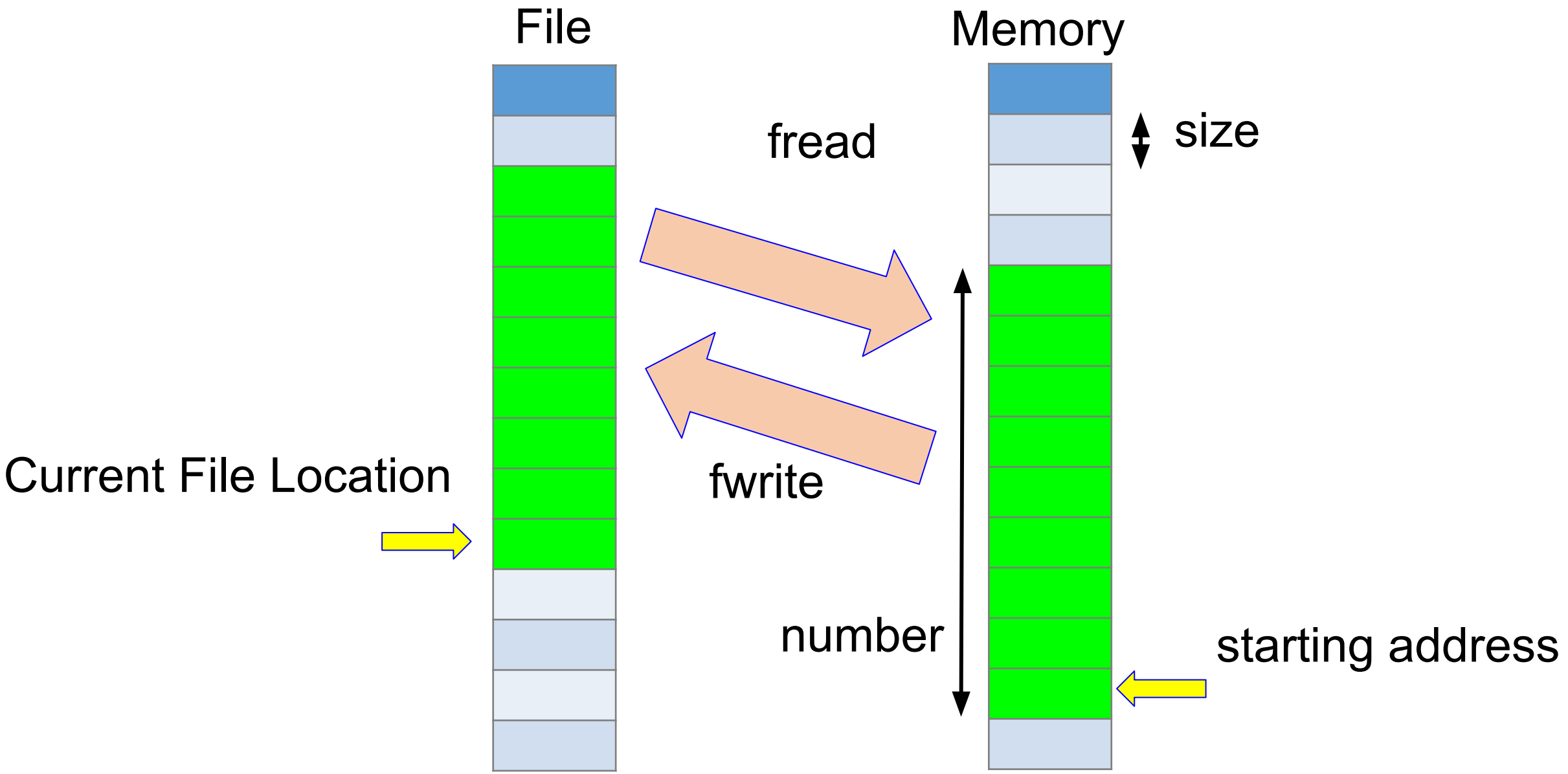
Homework 08
fread and fwrite

```c
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

The function **fread**() reads *nmemb* items of data, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* items of data, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

On success, **fread**() and **fwrite**() return the number of items read or written. This number equals the number of bytes transferred only when *size* is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).


File
Memory
fread
size
fwrite
Current File Location
number
starting address

```
Vector * vecArr;
vecArr = malloc(sizeof (* vecArr) * numElem);
FILE * fptr = fopen(filename, "r");
int numRead = 0;
numRead = fread(& vecArr[0], sizeof(Vector),
          numElem, fptr);
if (numRead != numElem) { /* something is wrong */ }
int nextByte = fgetc(fptr);
if (nextByte != EOF) { /* more data than needed */ }
fclose (fptr);
```

Advantages of fread / fwrite

- Compared with fgetc, fscanf, fgets, fprintf
- fread or fwrite: large amounts of data at once
- fread or fwrite: read / write different data types
- The structure can very complex (many attributes). These two functions take care of all attributes.
- If a structure changes (it will change), the program still works after re-compilation.

File Read / Write Function

- fread / fwrite should be used together
- Do not mix fread with fprint
- Do not mix fwrite with fgetc, fscanf, fgets

```c
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

long ftell(FILE *stream);
```

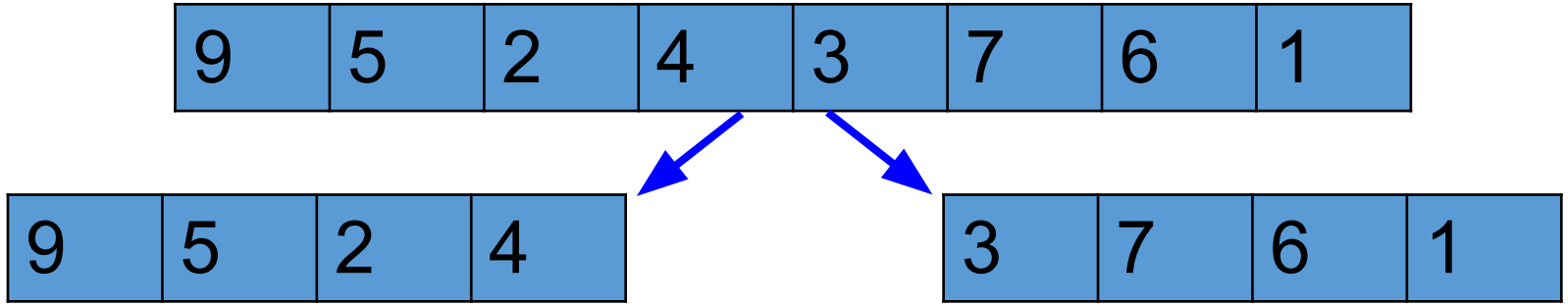
The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) function on the same stream.

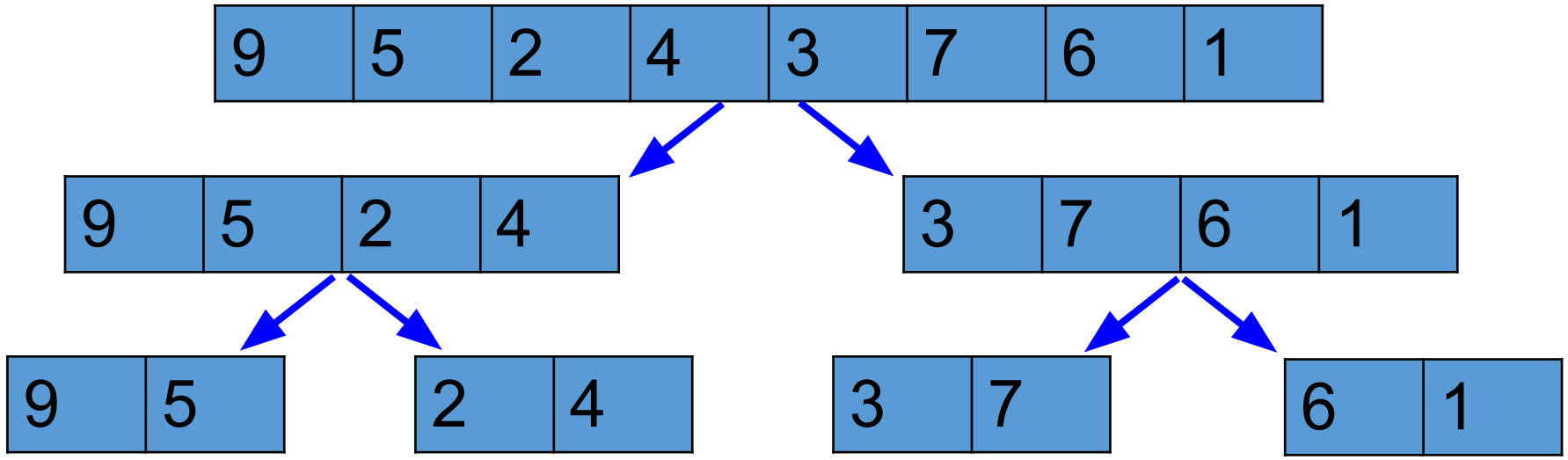
The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

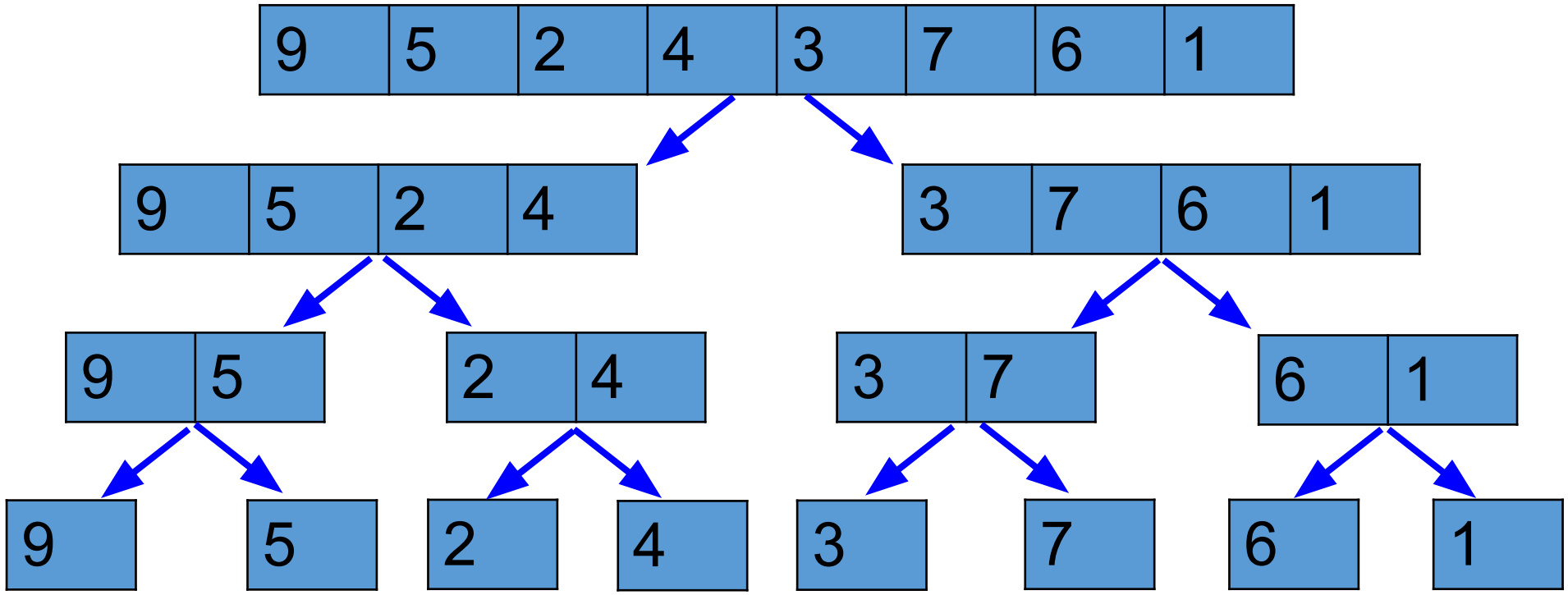
Homework 09 Merge Sort

Merge Sort

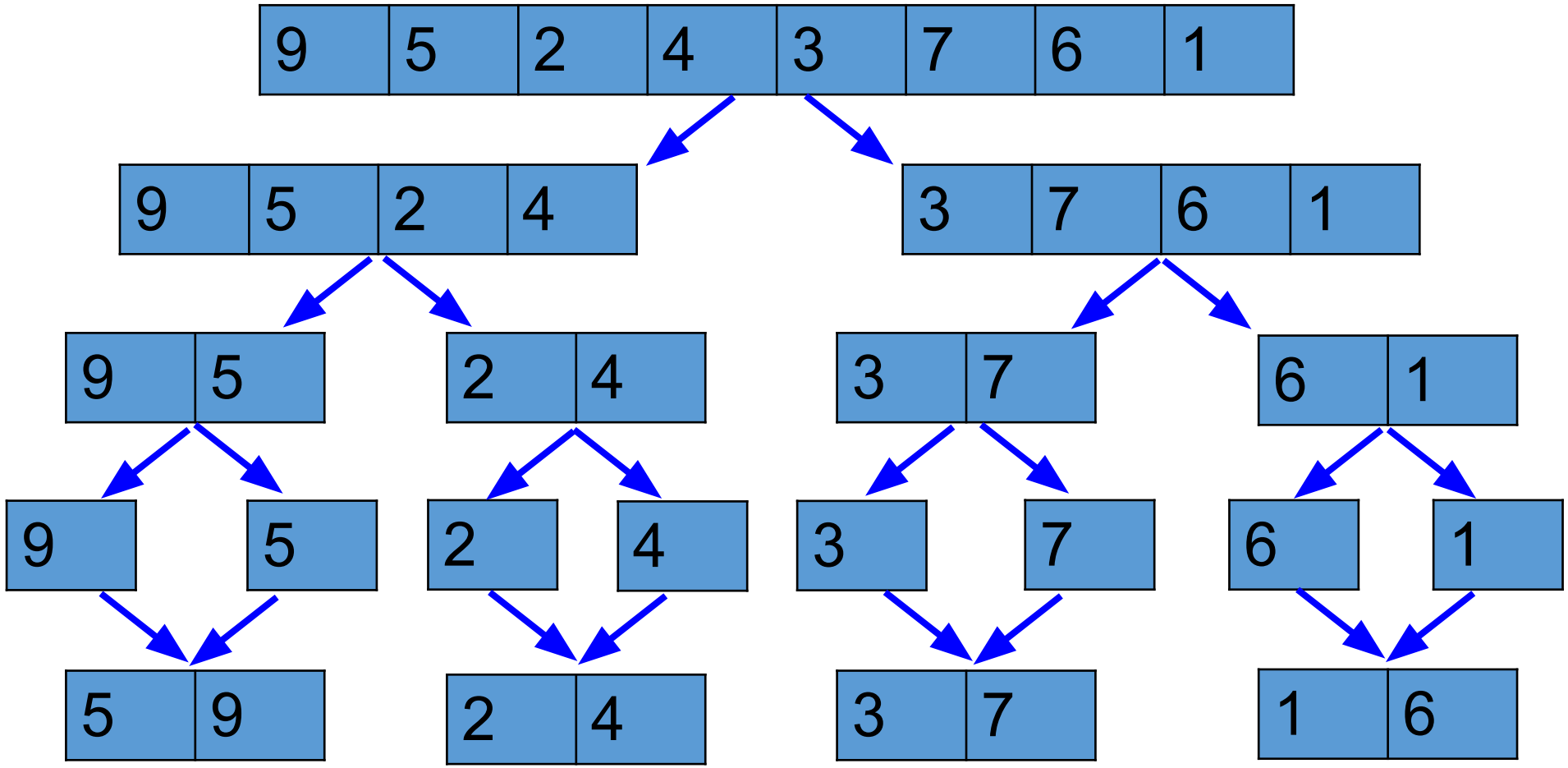
- Divide an array into two halves until one or no element
- Merge and sort the two arrays

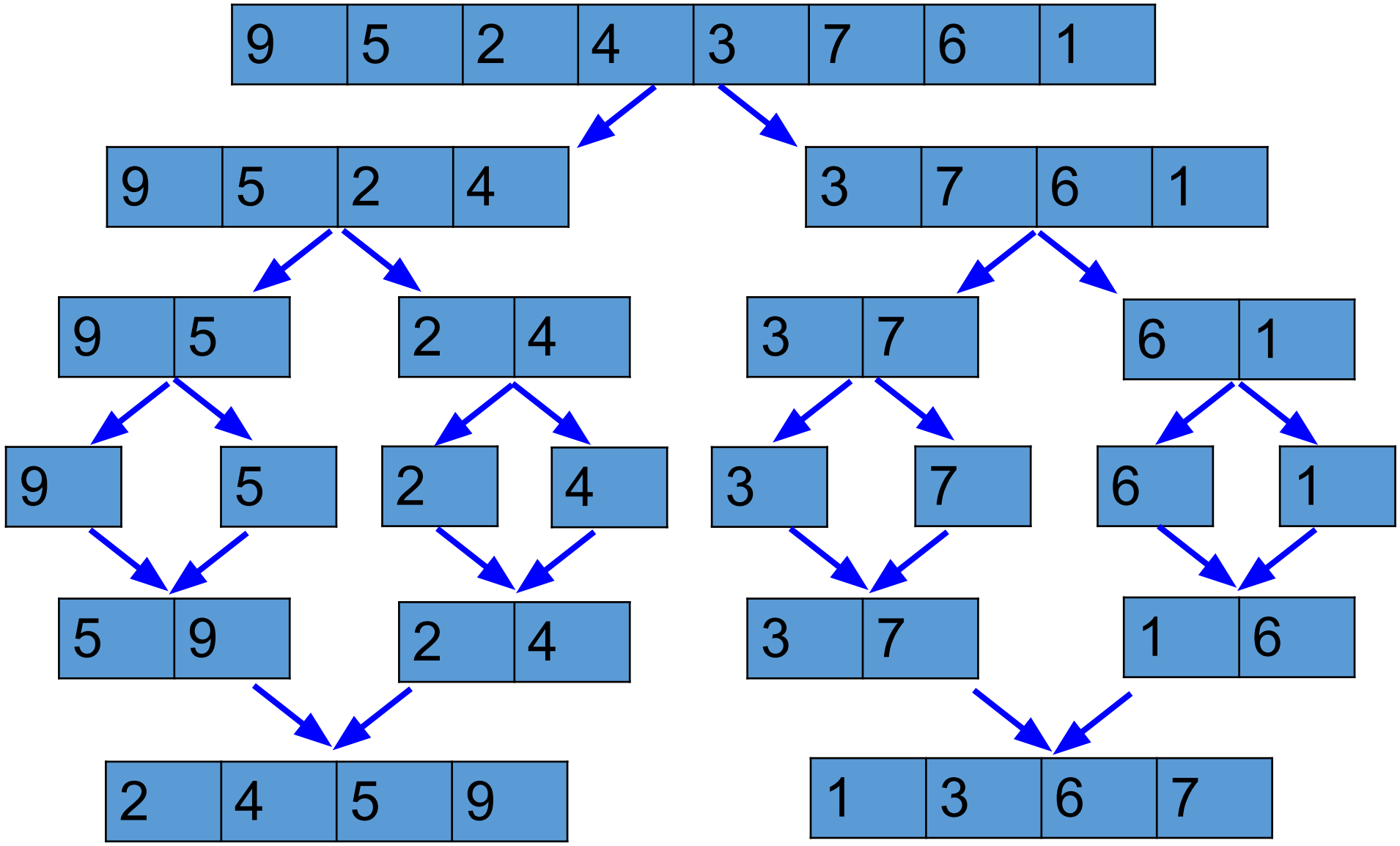

9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1

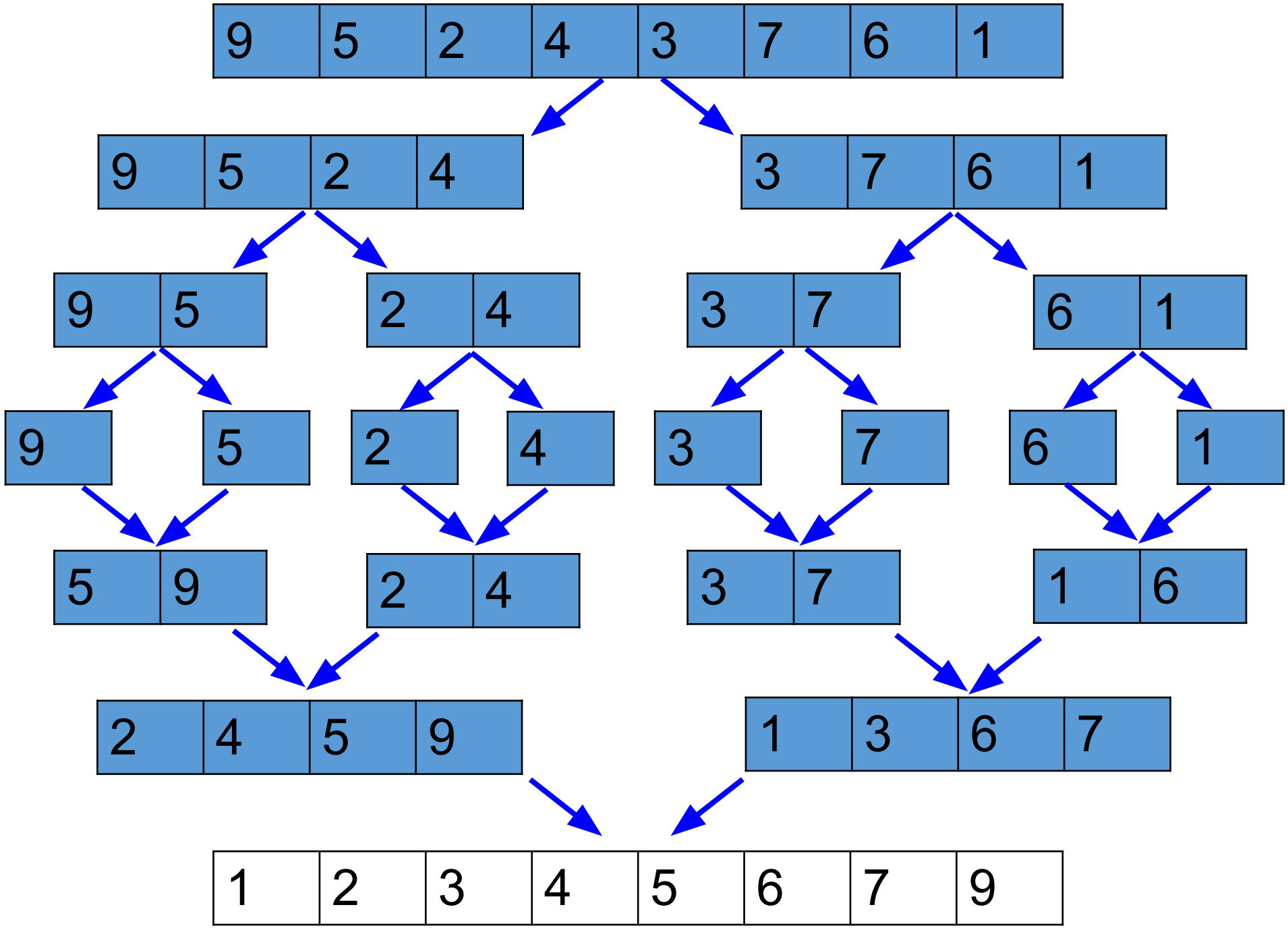

9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1
9 5
2 4
3 7
6 1


9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1
9 5
2 4
3 7
6 1
9
5
2
4
3
7
6
1

```
void mergeSort(int * arr, int l, int r)
{
  if (l > r) { return;  } // empty array, nothing to do
  if (l == r) { return; } // only one element, already sorted
  int m = (l + r)/2;
  mergeSort(arr, l, m);
  mergeSort(arr, m+1, r); // notice + 1
  merge(arr, l, m, r);
}
```

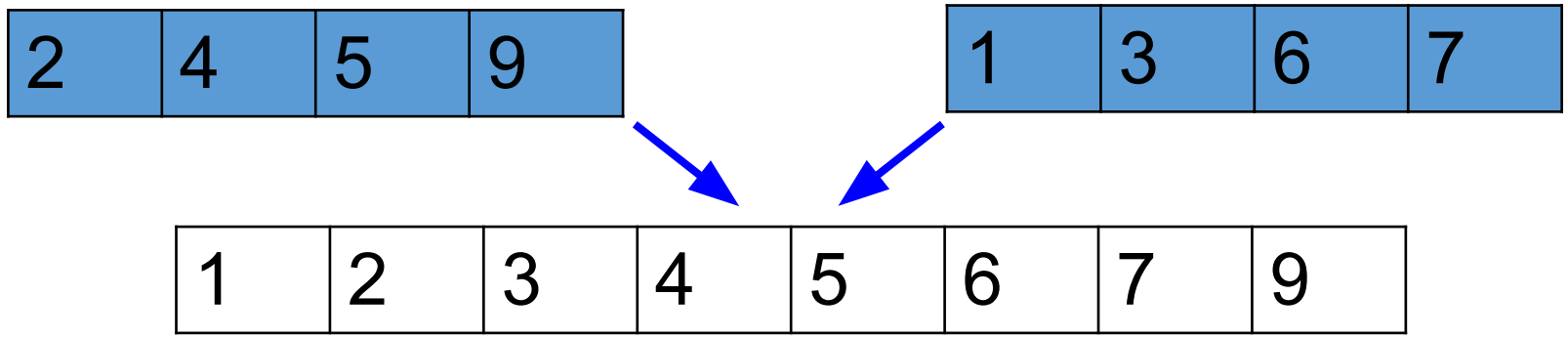

9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1
9 5
2 4
3 7
6 1
9
5
2
4
3
7
6
1
5 9
2 4
3 7
1 6


9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1
9 5
2 4
3 7
6 1
9
5
2
4
3
7
6
1
5 9
2 4
3 7
1 6
2 4 5 9
1 3 6 7


9 5 2 4 3 7 6 1
9 5 2 4
3 7 6 1
9 5
2 4
3 7
6 1
9
5
2
4
3
7
6
1
5 9
2 4
3 7
1 6
2 4 5 9
1 3 6 7
1 2 3 4 5 6 7 9

Transitivity in Merge Sort

If a > b and b > c, then a > c. No need to compare a and c.

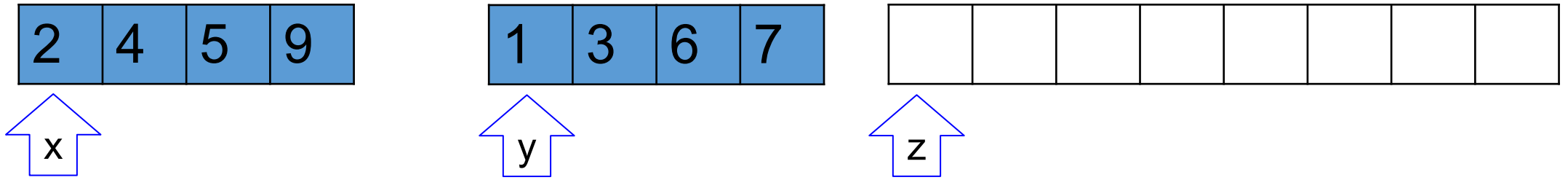


Compare 1 and 2, 1 is smaller, put 1 as the first

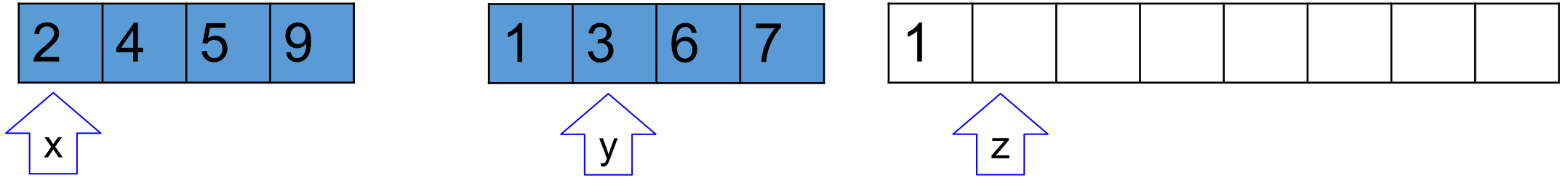
Compare 2 and 3, 2 is smaller, put 2 as the second

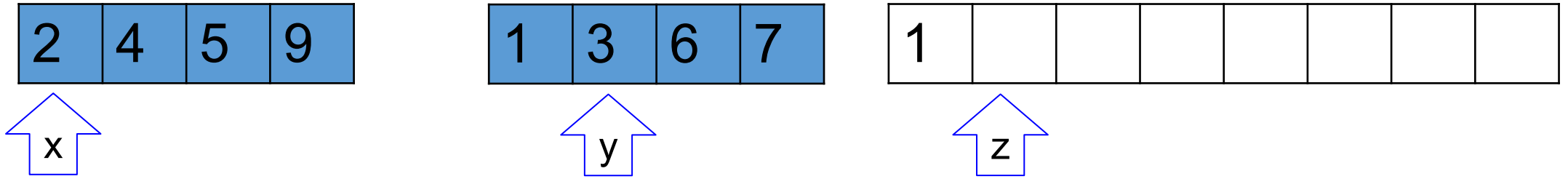
Compare 3 and 4, 3 is smaller, put 3 as the third

…

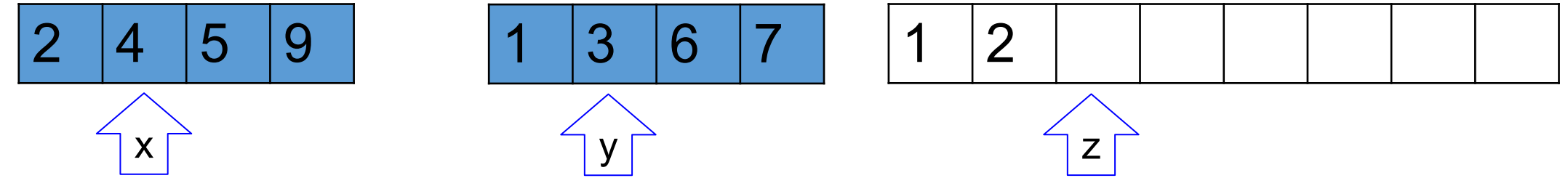


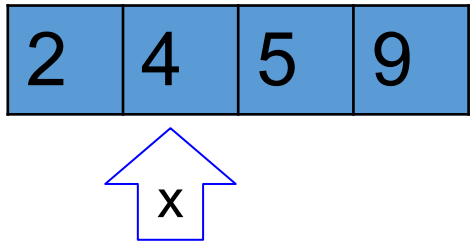

```
if (arr1[x] > arr2[y])
{
   arr3[z] = arr2[y];
   y ++;
   z ++;
}
```

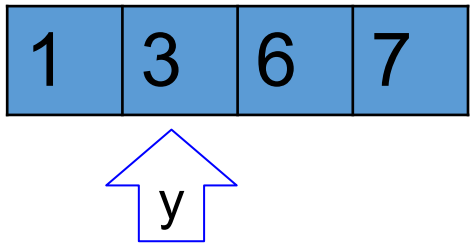


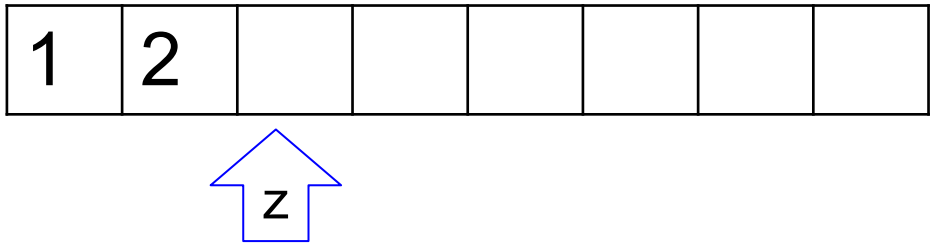



```
if (arr1[x] < arr2[y])
{
   arr3[z] = arr1[x];
   x ++;
   z ++;
}
```

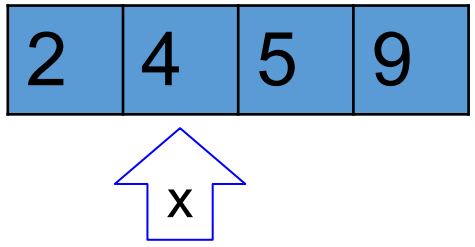


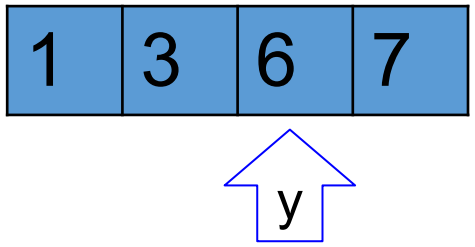


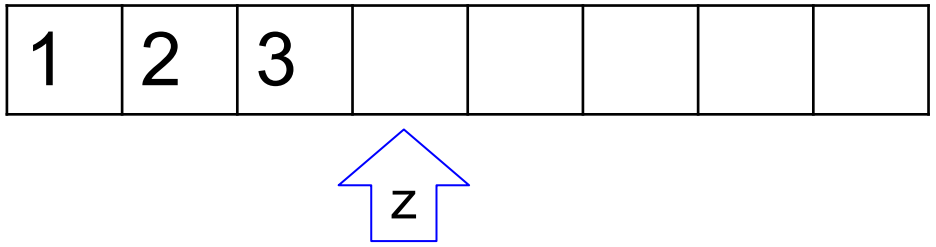





```
if (arr1[x] > arr2[y])
{
   arr3[z] = arr2[y];
   y ++;
   z ++;
}
```







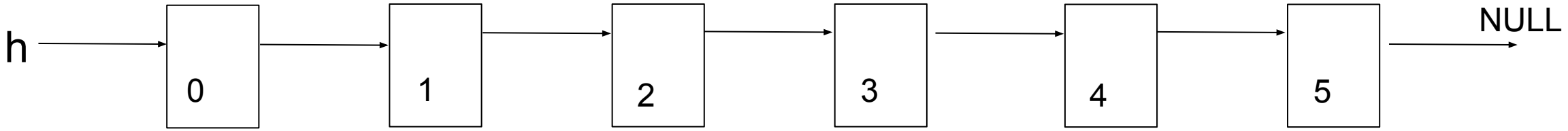
“Best” sorting algorithm?

- We have seen selection, quick, and merge sort.
- What is the “best” sorting algorithm?
- Evaluation metrics:
  - Number of comparisons or data movements
  - Stability: Preserved the order of same values
  - Best, Worst, Average cases
  - Almost sorted array
  - Number of random / sequential data accesses
  - (For very large amounts of data) Number of disk accesses
  - (For distributed data) Number of network packages

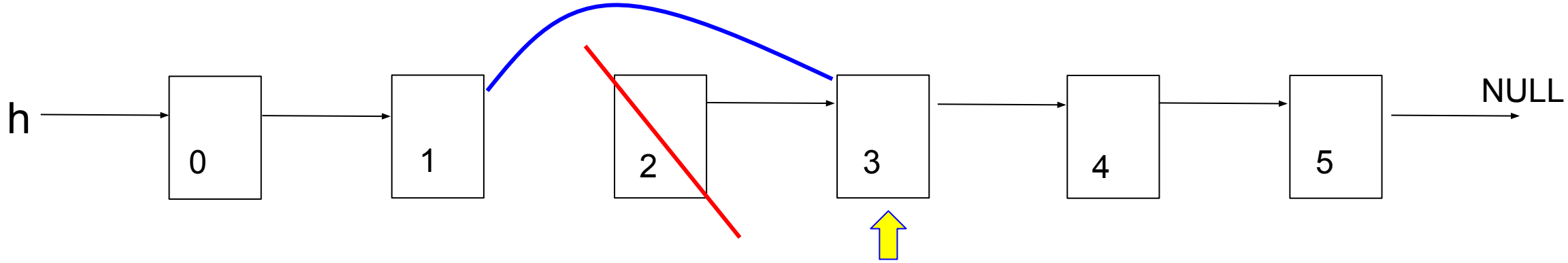
Homework 10

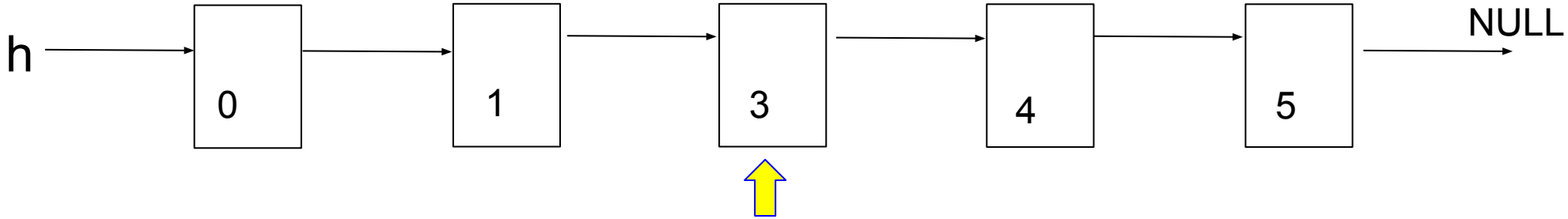
Who Gets the Cake using linked list

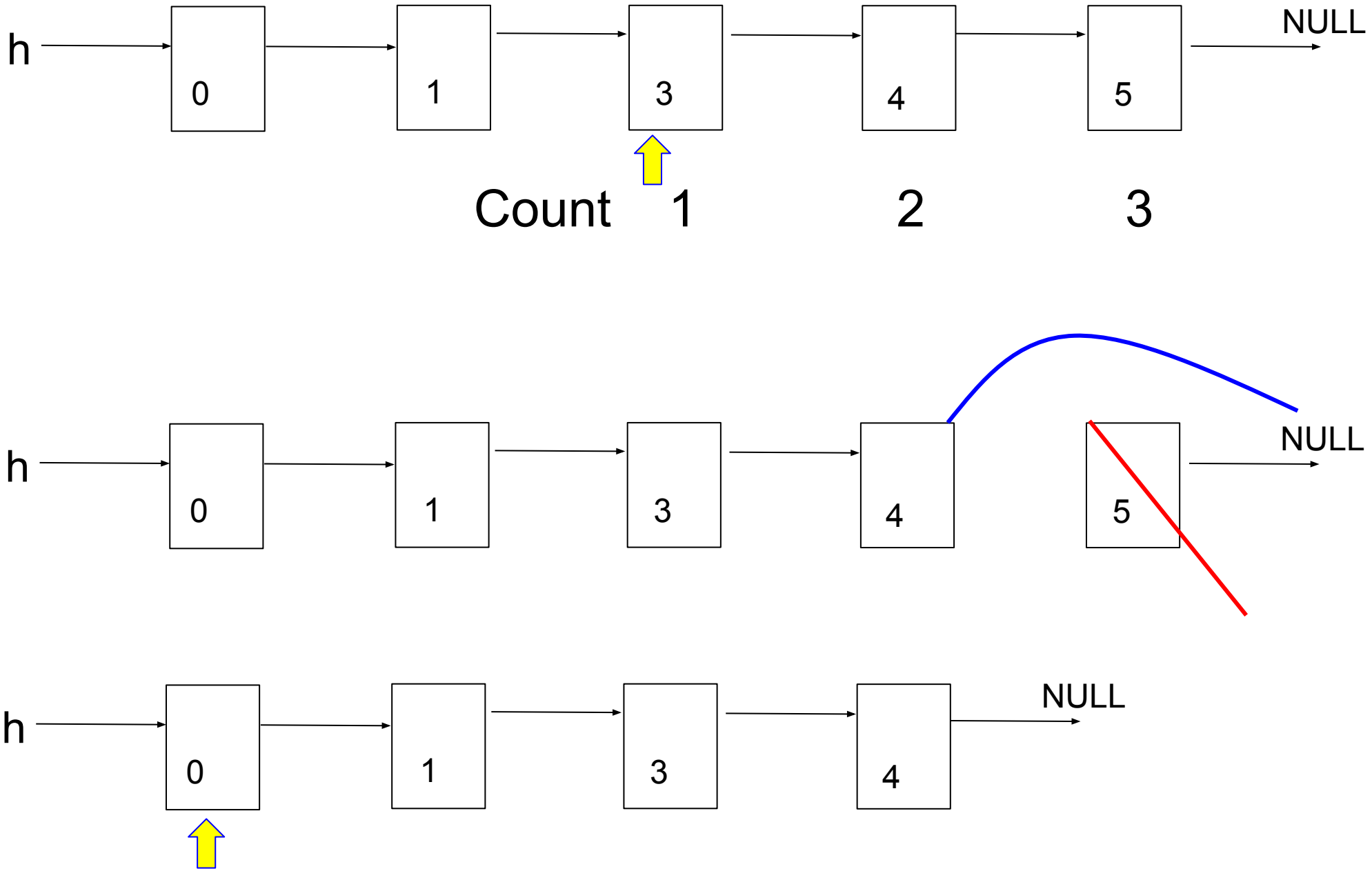
n is 6 and k is 3

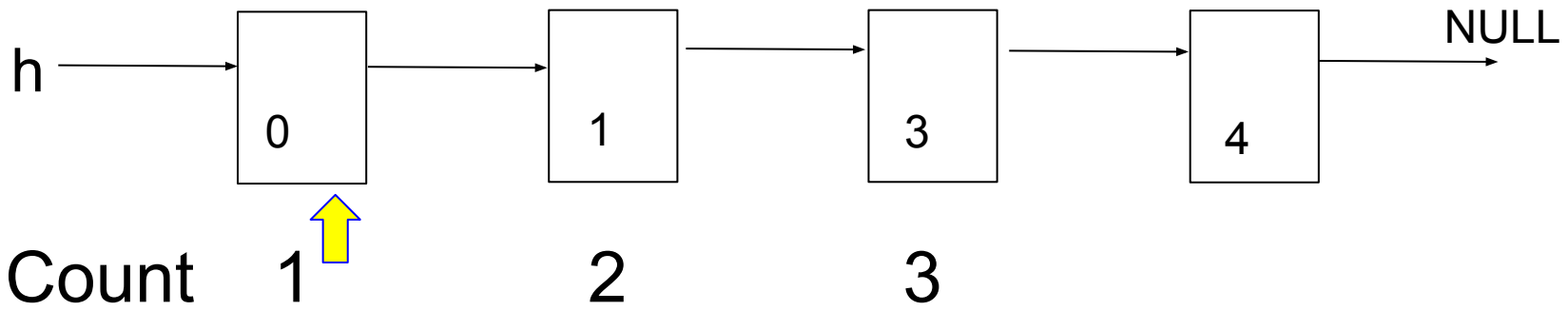


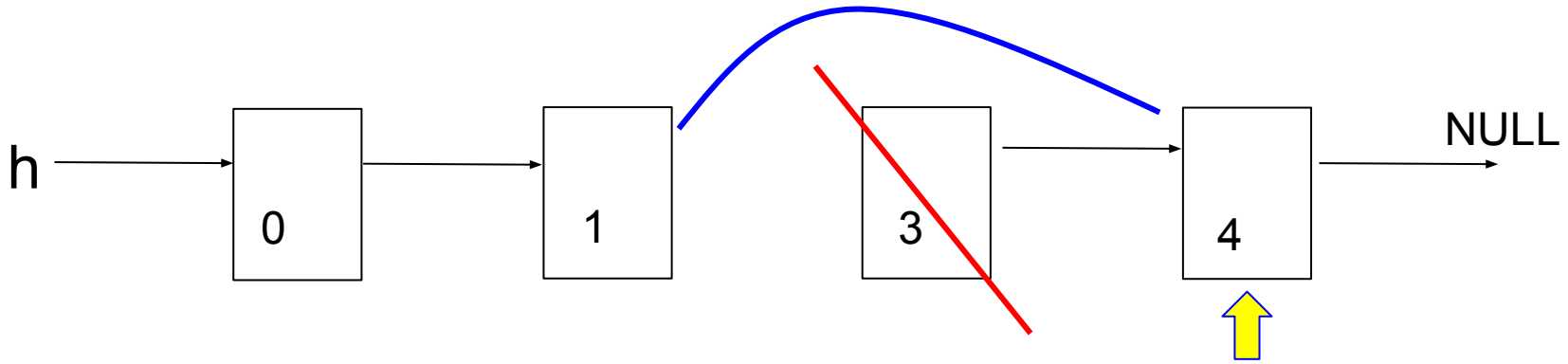
Count 1 2 3

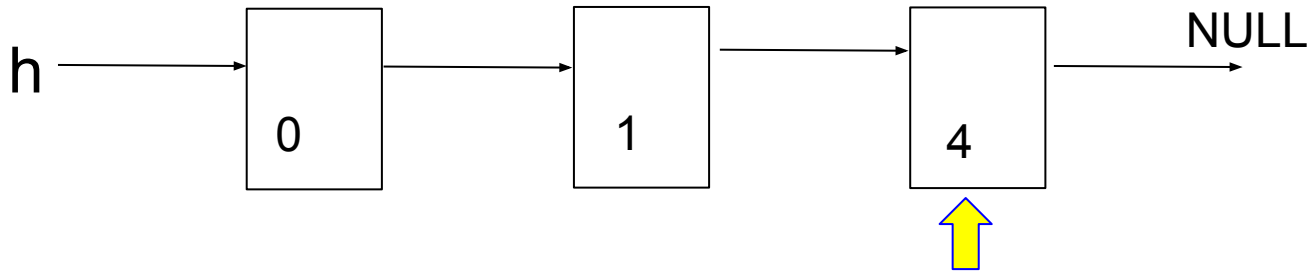


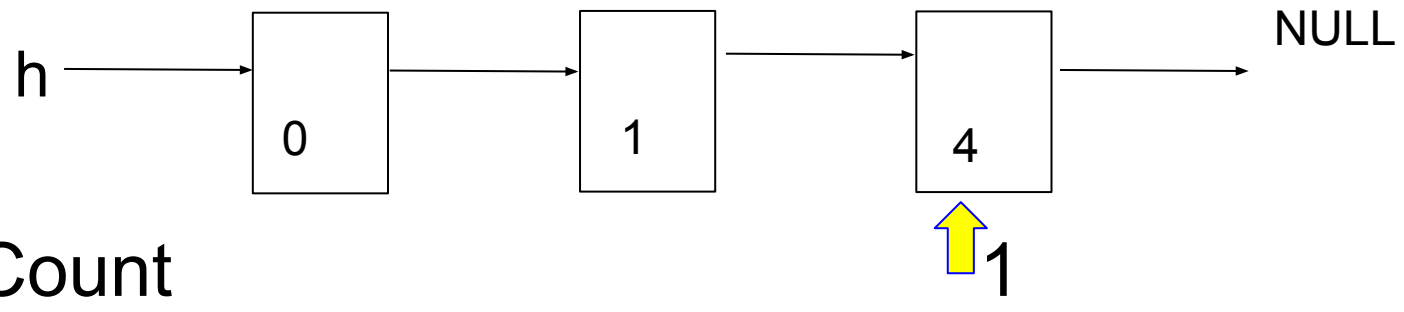



h
0
1
3
4
5
NULL
Count
1
2
3
h
0
1
3
4
5
NULL
h
0
1
3
4
NULL


h
0
1
3
4
NULL
Count
1
2
3


h
0
1
3
4
NULL


h
0
1
4
NULL

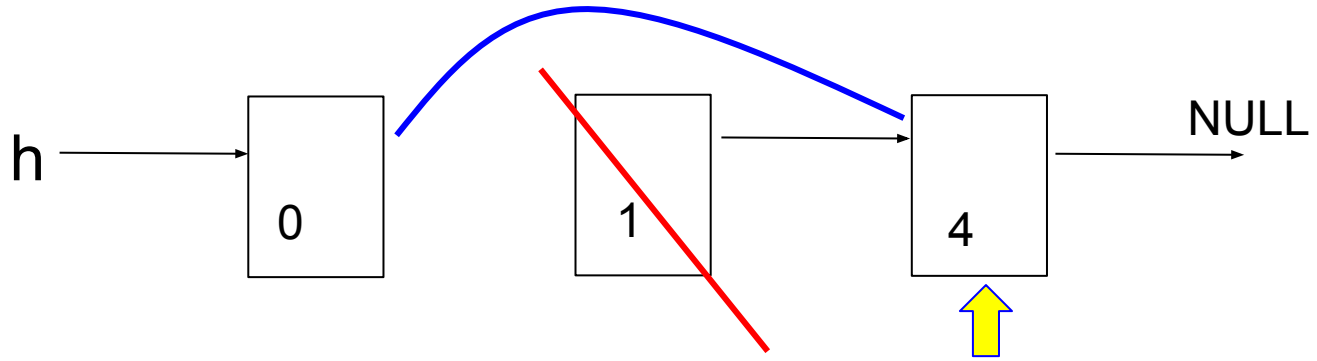


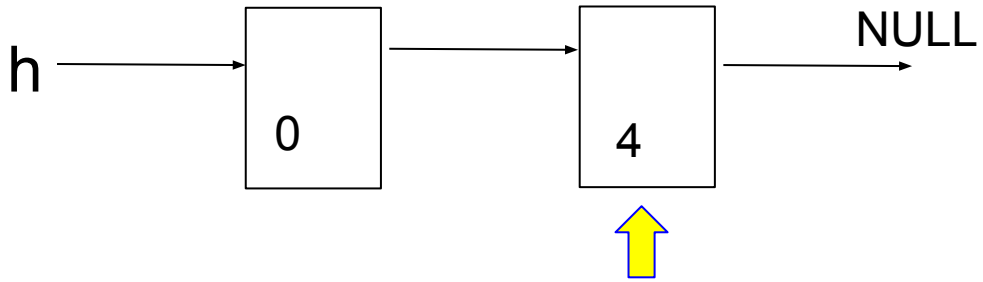
Count

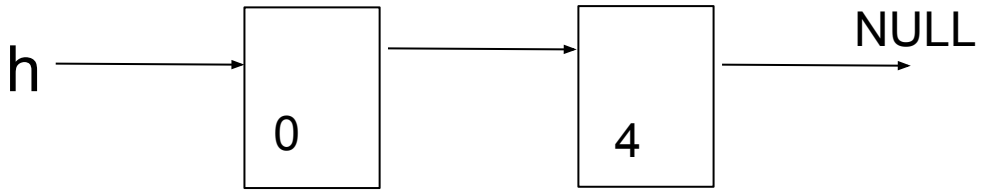
1

2

3

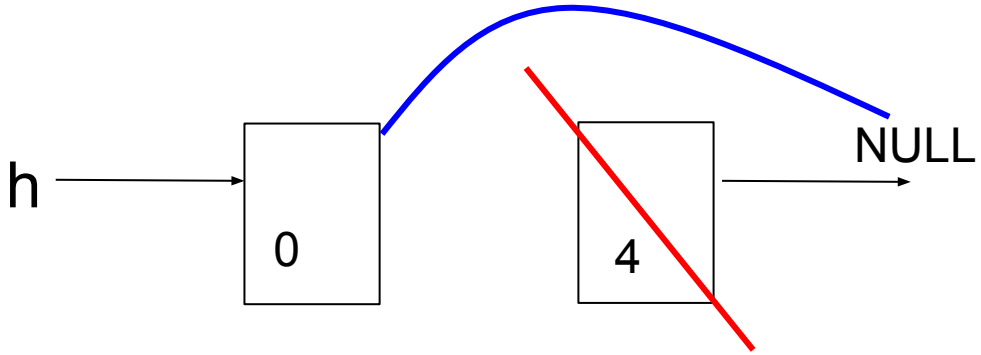






Count

1



2

3