計算機程式設計

Computer Programming

Dynamic Memory Allocation
Linked List

Instructor: 林英嘉
2024/12/16

W15 Slido: #3259824

GitHub repo

Outline

- Recap for struct
- Lifetime of C Variables
- Dynamic Memory Allocation
  - malloc, free
- Linked List

Why do we need dynamic memory allocation?

- C’s data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we’re forced to choose their sizes when writing a program.

```
char a_string[50];

struct Student {
    char name[50];
    float grade;
};
struct Student students[20];
```

Maybe only 10 characters before ‘\0’, but we need 50 bytes for storage.

There may be only 10 students, but we allocated memory for the size of 20 once we run the program.

[Definition] Dynamic Memory Allocation

- In C, dynamic memory allocation refers to performing manual memory management via functions in the C standard library, such as:
  - malloc (memory allocation)
  - free (deallocate memory)
- **Standard variable declarations**: memory allocation is automatically managed by C and typically fixed until the variable goes out of scope,
- **Dynamic memory allocation**: memory allocated for variables is adjustable during program execution.

[Definition] malloc

https://cplusplus.com/reference/cstdlib/malloc/

- malloc: **m**emory **alloc**ation (defined in stdlib.h)
- Allocates a block of memory, returning a pointer to the beginning of the block.
  - So usually, we need a pointer variable to attain the returned value of malloc.
- Prototype:

```
void* malloc (size_t size);
```

void*: the returned pointer type. void means we can later transform a pointer to any type.

size_t: the return type (long unsigned int) of sizeof, so `size` can not be a negative integer

size: size of the memory block, in bytes

[Usage] malloc

- malloc: **m**emory **alloc**ation (defined in stdlib.h)
- Allocates a block of memory, returning a pointer to the beginning of the block.

```
ptr = (data_type*) malloc (size);
```

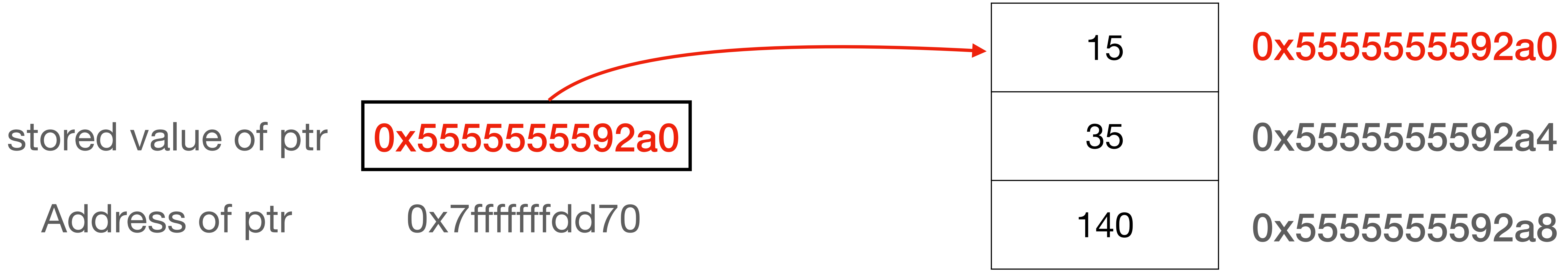
Type casting for the returned pointer.

Example (Code Snippet)

C-course-materials/09-DynamicMemoryAlloc/malloc_example1.c

```
int main(void){
    int *ptr;
    ptr = (int *)malloc(sizeof(12));
    *ptr = 15;
    *(ptr + 1) = 35;
    *(ptr + 2) = 140;
}
```

If we want to allocate three integers, we can set the size as 3x4=12.



stored value of ptr 0x5555555592a0

Address of ptr 0x7ffffffdd70

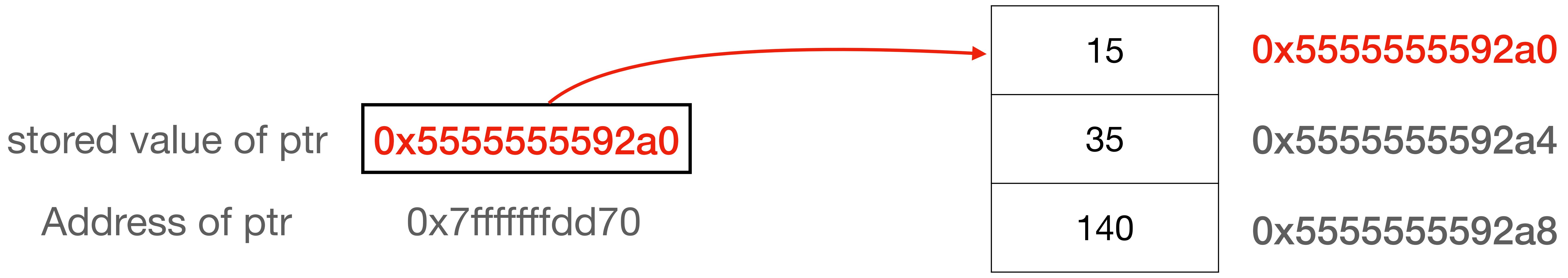
15	0x5555555592a0
35	0x5555555592a4
140	0x5555555592a8

Example (Code Snippet)

C-course-materials/09-DynamicMemoryAlloc/malloc_example1.c

```
int main(void){
    int *ptr;
    ptr = (int *)malloc(3*sizeof(int));
    *ptr = 15;
    *(ptr + 1) = 35;
    *(ptr + 2) = 140;
}
```

If we want to allocate three integers, we also use 3*sizeof(int) to get the size.



stored value of ptr 0x5555555592a0

Address of ptr 0x7fffffffdd70

15	0x5555555592a0
35	0x5555555592a4
140	0x5555555592a8

[Important Notes] malloc

- malloc does not guarantee the allocated memory addresses are continuous.
- The memory allocated with malloc has a lifetime that extends beyond the function where it was allocated.
    - Memory allocated with malloc is not automatically freed when you leave the function or its scope.

Using Dynamic Storage Allocation in String Functions

C-course-materials/09-DynamicMemoryAlloc/malloc_example2.c

```c
// Copy a string using dynamic allocation
char *copy_string_dynamic(const char *source) {
    char *new_string = (char *)malloc((strlen(source) + 1) * sizeof(char));
    strcpy(new_string, source);
    return new_string;
}

void copy_string_static(char *destination, const char *source, size_t size) {
    if (strlen(source) + 1 > size) {
        printf("Error! Destination is too small.\n");
        return;
    }
    strcpy(destination, source);
}
```

Using Dynamic Storage Allocation in String Functions

C-course-materials/09-DynamicMemoryAlloc/malloc_example2.c

```c
int main() {
    const char *original = "Hello, World!";

    // static allocation
    char static_buffer[20];
    copy_string_static(static_buffer, original, sizeof(static_buffer));
    printf("%s\n", static_buffer);

    // dynamic allocation
    char *dynamic_copy_ptr;
    dynamic_copy_ptr = copy_string_dynamic(original);
    printf("Result with malloc: %s\n", dynamic_copy_ptr);
}
```

[Definition] free

https://cplusplus.com/reference/cstdlib/free/

- free (defined in stdlib.h): A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.
- If ptr does not point to a block of memory allocated with the above functions, it causes undefined behavior.
- Prototype:

```
void free (void* ptr);
```

Using void* as the type for a parameter means the function can accept a pointer to any type.

[Usage] free

https://cplusplus.com/reference/cstdlib/free/

- free will be passed a pointer to an unneeded memory block:

```
free (ptr);
```

Example of free

C-course-materials/09-DynamicMemoryAlloc/free_example.c

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    *ptr = 15;
    *(ptr + 1) = 35;
    *(ptr + 2) = 140;
    printf("Value at ptr + 2: %d\n", *(ptr + 2));
    free(ptr);
    printf("Value at ptr + 2: %d\n", *(ptr + 2));
}
```

- Output

```
Value at ptr + 2: 140
Value at ptr + 2: 1431670800
```

Value after being free

[Important Notes] free

https://cplusplus.com/reference/cstdlib/free/

- A block of memory that’s no longer accessible to a program is said to be garbage.
- A program that leaves garbage behind has a **memory leak**.
- Some languages provide a garbage collector that automatically locates and recycles garbage, but C doesn’t.
- Instead, each C program is responsible for recycling its own garbage by calling the free function to release unneeded memory.

Dangling Pointers

- A dangling pointer (迷途指標) is a pointer which points to a non-existing memory location.
- If we forget that **a pointer** no longer points to a valid memory block, undefined behaviors may occur.

Dangling Pointer Example

C-course-materials/09-DynamicMemoryAlloc/dangling_pointer.c

```c
#include <stdio.h>
#include <stdlib.h>
struct s {
    int i;
};

int main(void){
    struct s *p1 = malloc(sizeof(struct s));
    p1->i = 42;
    printf("(p1) before free: p->i = %d\n", p1->i);

    free(p1);
    struct s *p2 = malloc(sizeof(struct s));
    p2->i = 100;
    // Trying to use the dangling pointer p1
    printf("(p1) after free: p->i = %d\n", p1->i);
    printf("(p2): p2->i = %d\n", p2->i);
}
```

- Output

```
(p1) before free: p->i = 42
(p1) after free: p->i = 100
(p2): p2->i = 100
```

p1 is a dangling pointer after free, but p2 may be allocated to the same (or different) memory pointed by p1.

Best Practices for a Dangling Pointer

C-course-materials/09-DynamicMemoryAlloc/dangling_pointer_sol.c

https://cplusplus.com/reference/cstddef/NULL/

1. Set the dancing pointer (p1 here) to NULL

```
int main(void){
    struct s *p1 = malloc(sizeof(struct s));
    p1->i = 42;

    free(p1);
    p1 = NULL;
}
```

NULL is a null pointer. Assigning a pointer with NULL means **the pointer is not pointing to any object**.

Best Practices for a Dangling Pointer

C-course-materials/09-DynamicMemoryAlloc/dangling_pointer_sol.c

2. Reallocate the memory for a dangling pointer (p1 here)

```
int main(void){
    struct s *p1 = malloc(sizeof(struct s));
    p1->i = 42;

    free(p1);
    p1 = malloc(sizeof(struct s));
}
```

Linked List

[Introduction] List (串列)

- Lists can be divided into two types:
    - Sequential list (elements are stored orderly in memory; **Array**)
    - Non-sequential list (elements are not stored orderly in memory; **Linked list**)

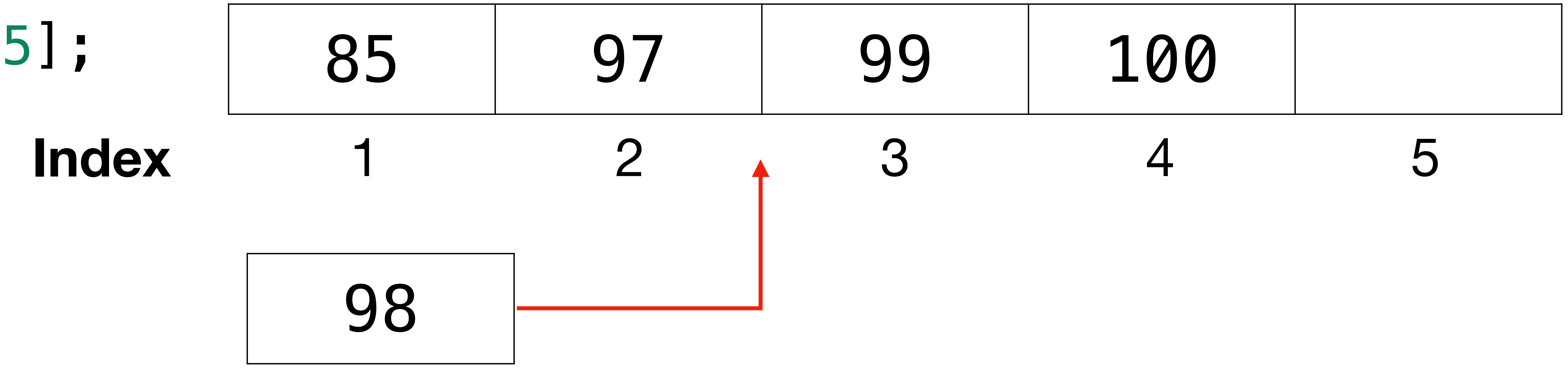
[Illustration] Movement with a Sequential list

```
int score[5];
```

	85	97	99	100	
Index	1	2	3	4	5

[Illustration] Movement with a Sequential list

```
int score[5];
```

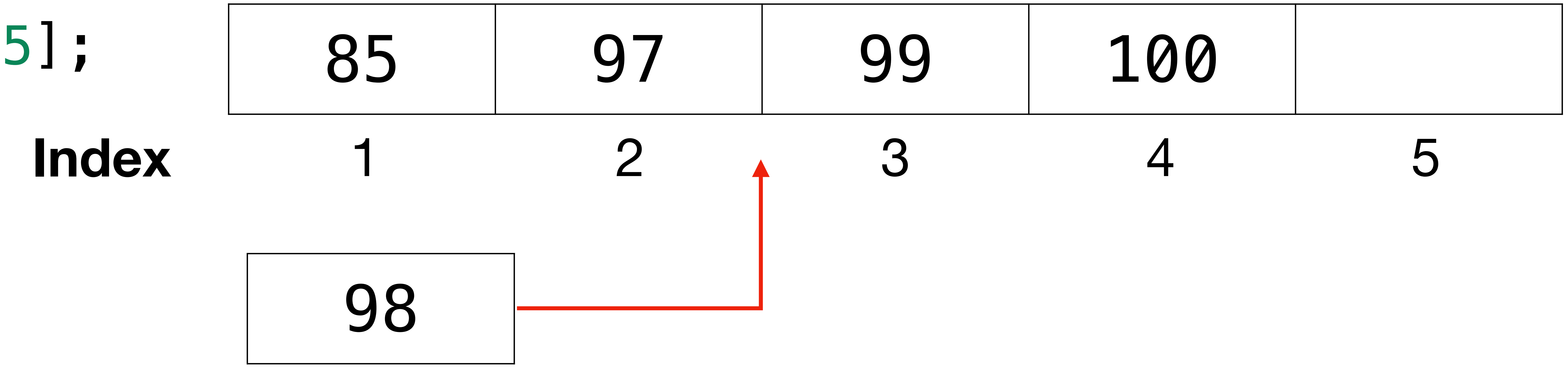


	85	97	99	100	
Index	1	2	3	4	5

98

[Illustration] Movement with a Sequential list

```
int score[5];
```



	85	97	99	100	
Index	1	2	3	4	5

98

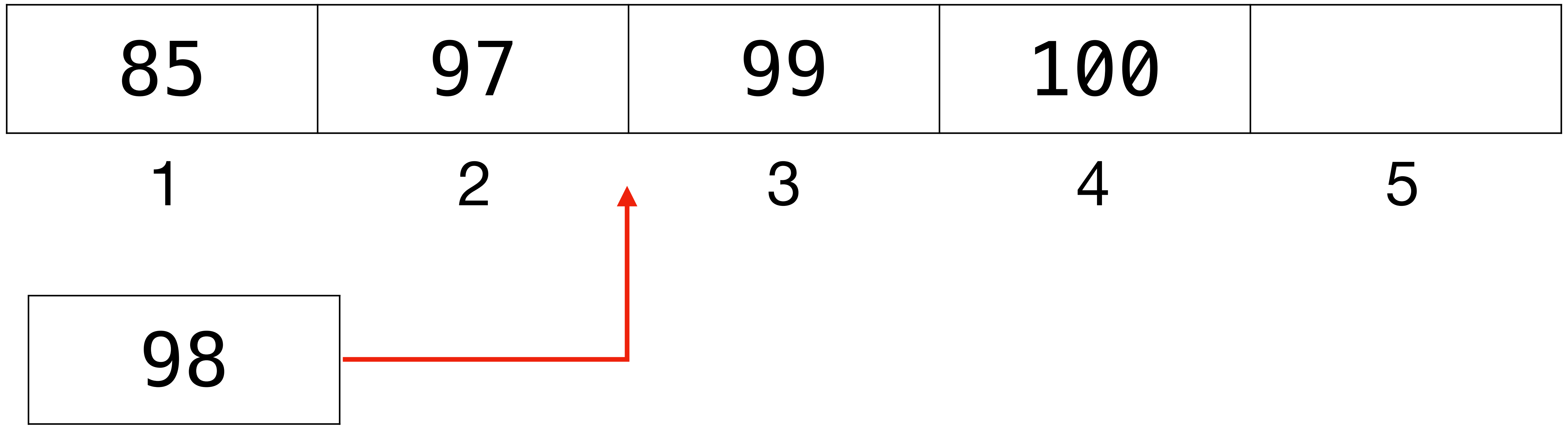
Step1: Move 100 to index5

Step2: Move 99 to index4

Step3: Put 98 to index3, and we can finally get the array:

[Illustration] Movement with a Sequential list

```
int score[5];
```



	85	97	99	100	
Index	1	2	3	4	5

98

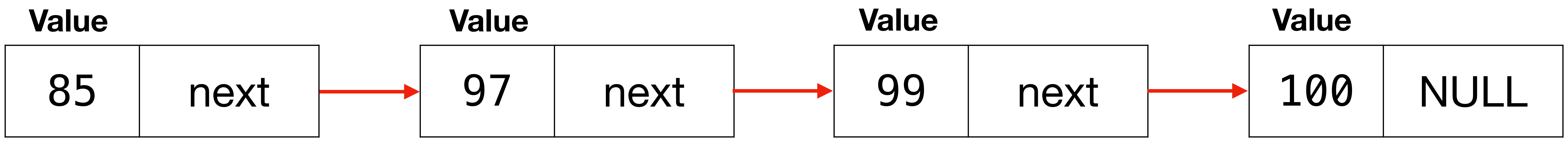
Step1: Move 100 to index5

Step2: Move 99 to index4

Step3: Put 98 to index3, and we can finally get the array:

85	97	98	99	100

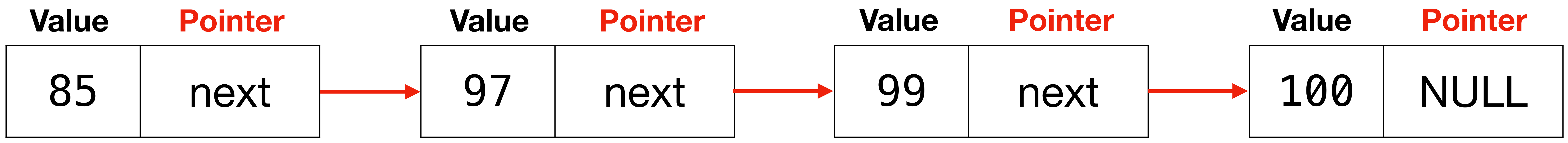
[Illustration] Linked List



Value 85 next → Value 97 next → Value 99 next → Value 100 NULL

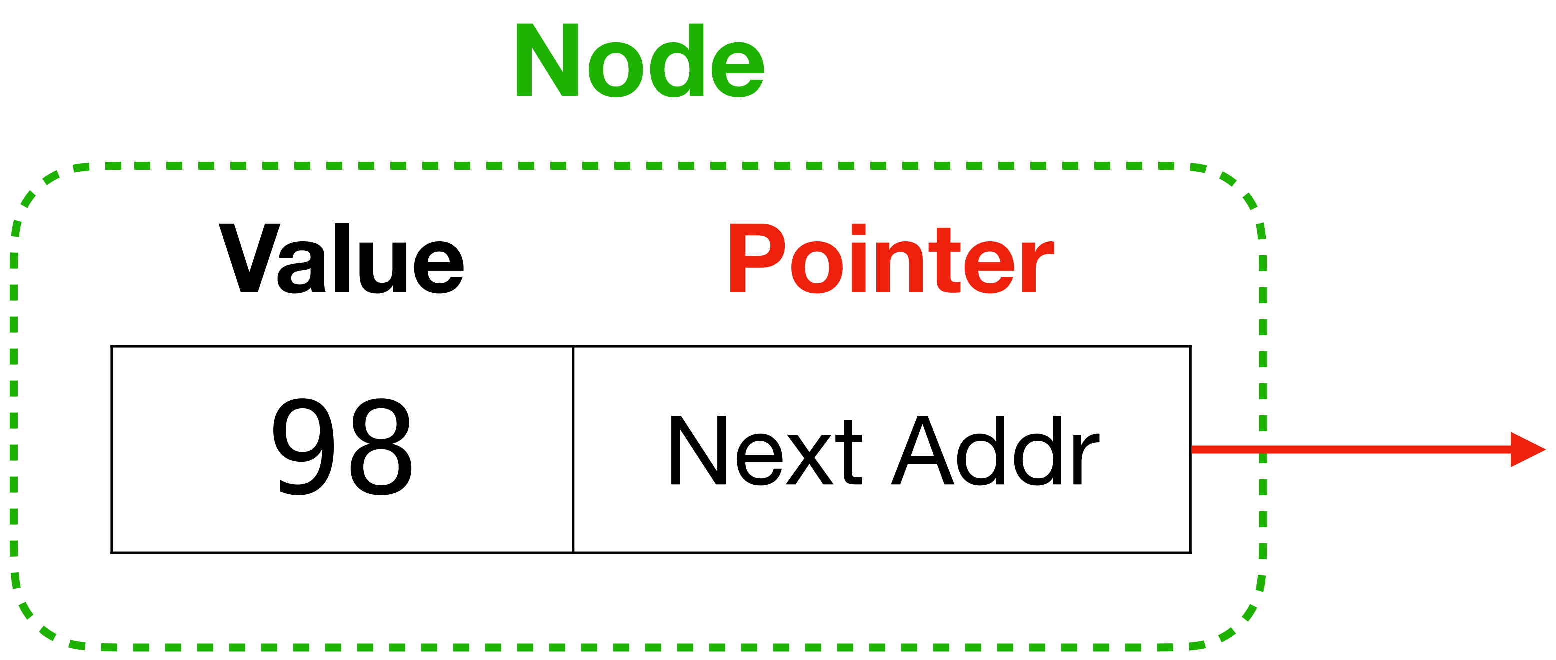
In linked lists, the memory addresses of all the values do not have to be continuous!

[Illustration] Linked List



In linked lists, the memory addresses of all the values do not have to be continuous!

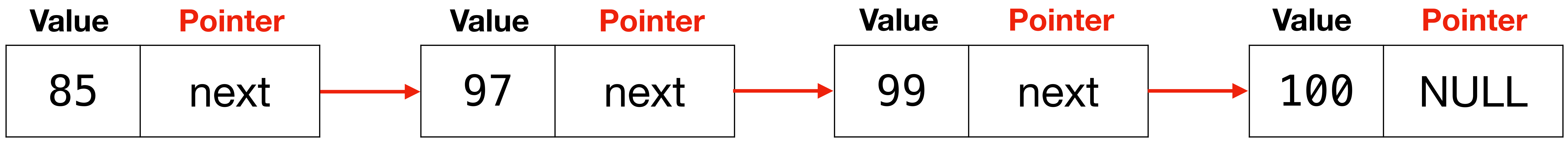
[Illustration] Linked List



Node

Value	Pointer
98	Next Addr

Point to the next element's address



Value	Pointer
85	next

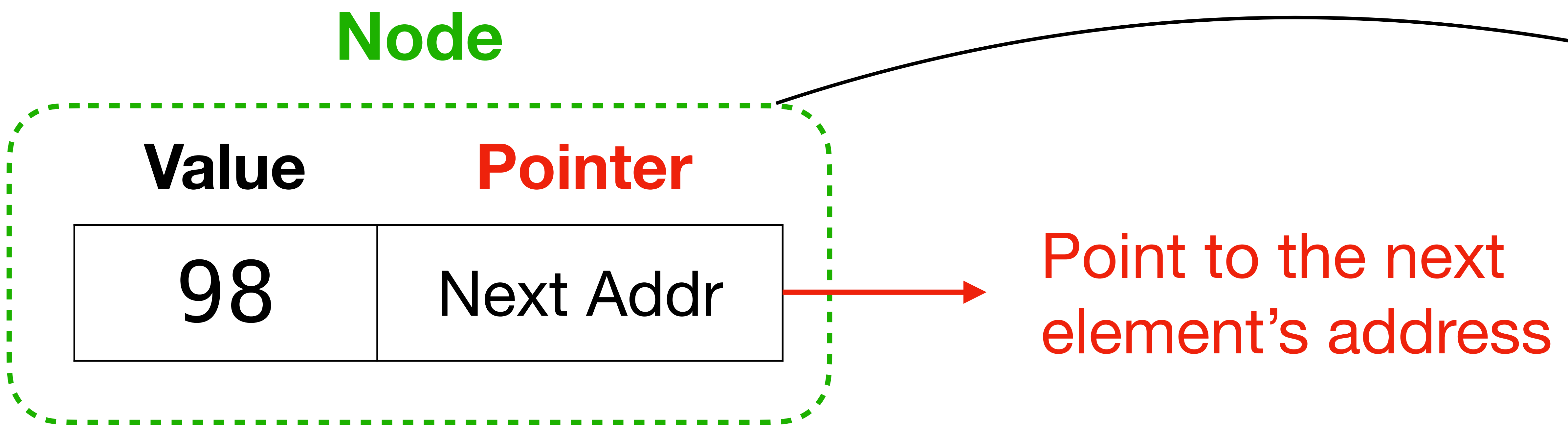
Value	Pointer
97	next

Value	Pointer
99	next

Value	Pointer
100	NULL

In linked lists, the memory addresses of all the values do not have to be continuous!

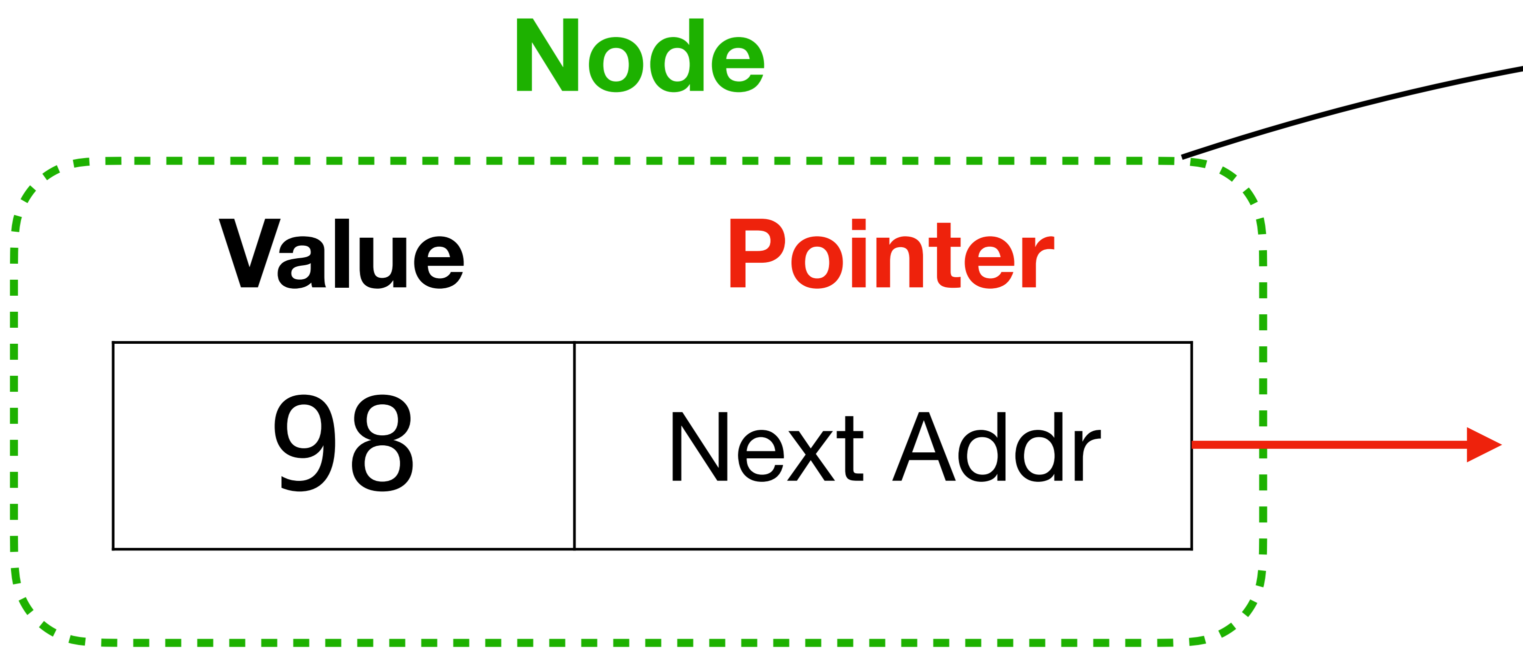
[Illustration] Implementation of Linked Lists (1/2)



Point to the next element's address

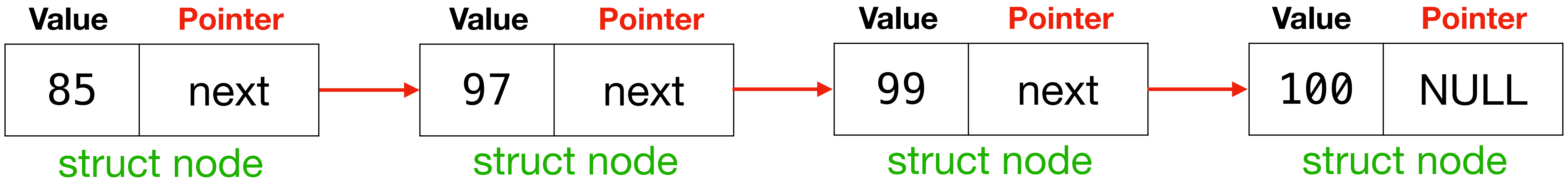
```
struct node {
    int val;
    struct node *next;
};
```

[Illustration] Implementation of Linked Lists (1/2)

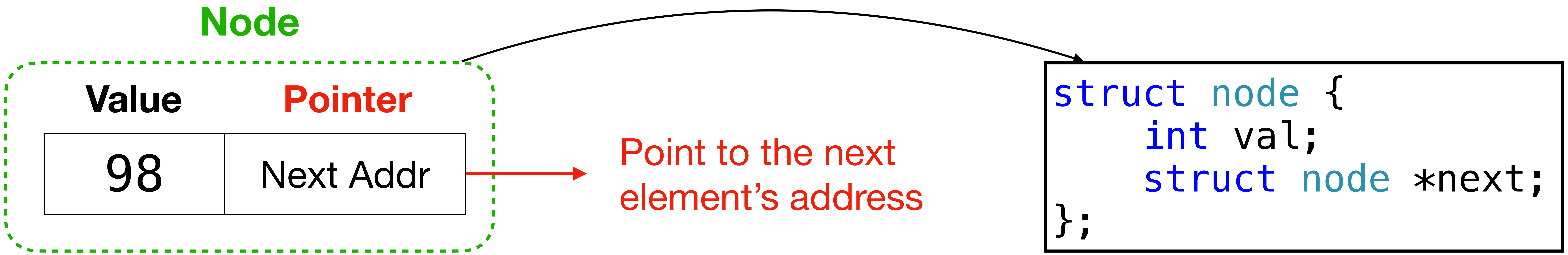


Point to the next element's address

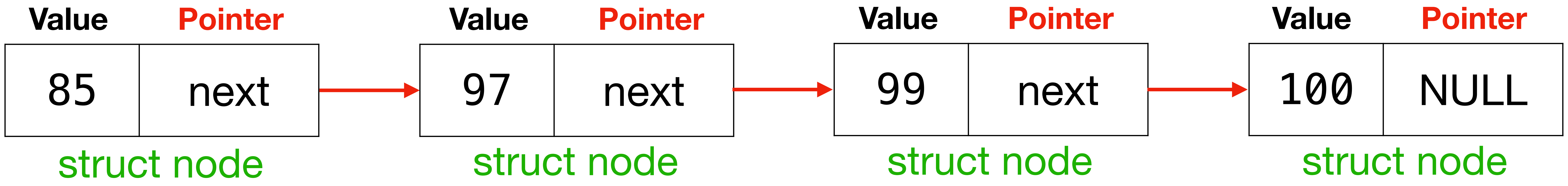
```
struct node {
    int val;
    struct node *next;
};
```



[Illustration] Implementation of Linked Lists (1/2)



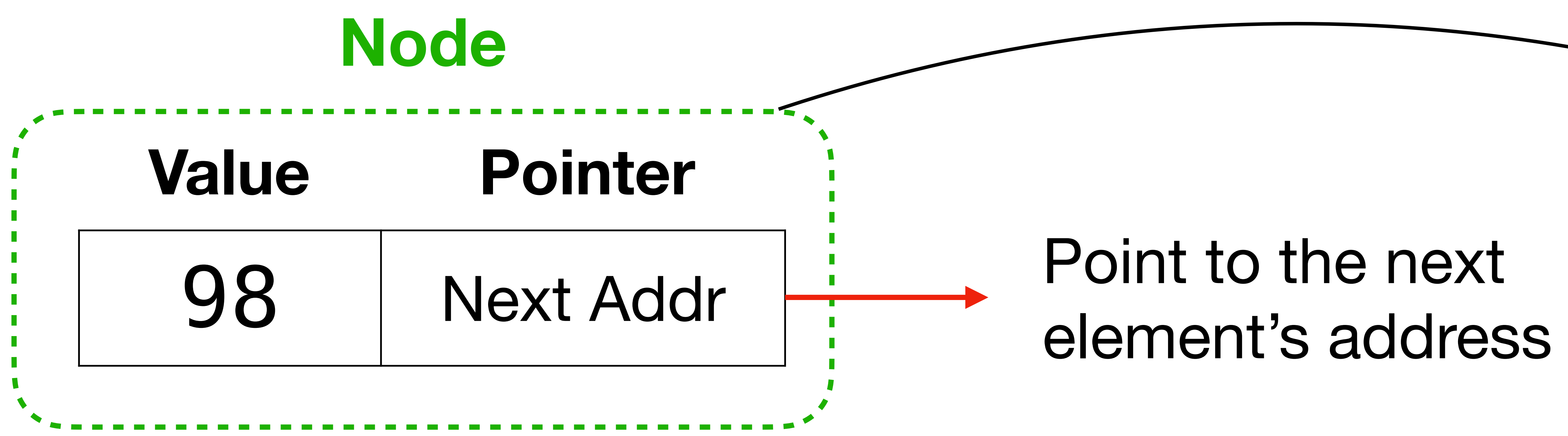
```
struct node {
    int val;
    struct node *next;
};
```



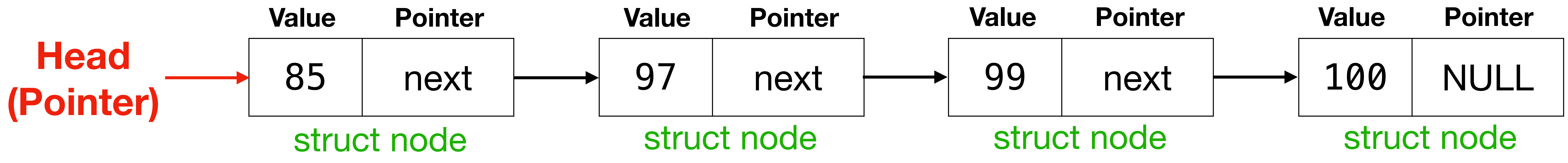
We can also use typedef to give struct node a nickname as **Node**.

```
typedef struct node Node;
```

[Illustration] Implementation of Linked Lists (2/2)



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



Usually, we need a Head, which is also a pointer, for getting the value of the first element in a linked list.

An Example of a Linked List (static memory)

C-course-materials/09-DynamicMemoryAlloc/linked_list_static.c

```
struct node {
    int val;
    struct node *next;
};
typedef struct node Node;
```

```
int main(void){
    Node stuA, stuB, stuC, stuD;
    Node *head = &stuA;
    stuA.val = 85;
    stuA.next = &stuB;
    stuB.val = 97;
    stuB.next = &stuC;
    stuC.val = 99;
    stuC.next = &stuD;
    stuD.val = 100;
    stuD.next = NULL;
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d ,", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

An Example of a Linked List (static memory)

C-course-materials/09-DynamicMemoryAlloc/linked_list_static.c

- Output

```
Current: 0x7ffffffdd10, 85 ,Next: 0x7ffffffdd20
Current: 0x7ffffffdd20, 97 ,Next: 0x7ffffffdd30
Current: 0x7ffffffdd30, 99 ,Next: 0x7ffffffdd40
Current: 0x7ffffffdd40, 100 ,Next: (nil)
```

NULL, not pointing another address.

int: 4 bytes

pointer: 8 bytes

padding: 4 bytes

Total storage for each structure: 16 bytes

```
struct node {
    int val;
    struct node *next;
};
typedef struct node Node;
```

Problems of a Linked List with static memory

Problems of a Linked List with static memory

1. Adjustment for the number of nodes is not flexible.

All nodes must be manually declared as variables during compilation, making it difficult to adjust the size dynamically.

Problems of a Linked List with static memory

1. Adjustment for the number of nodes is not flexible.

All nodes must be manually declared as variables during compilation, making it difficult to adjust the size dynamically.

2. The allocated memory is fixed for the program's lifetime.

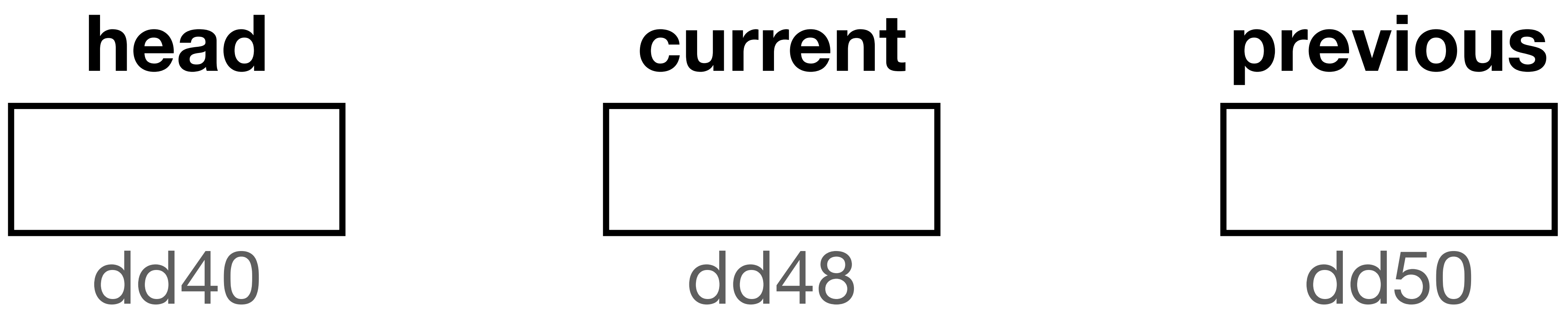
We cannot reclaim the allocated memory even if a node is not used.

An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

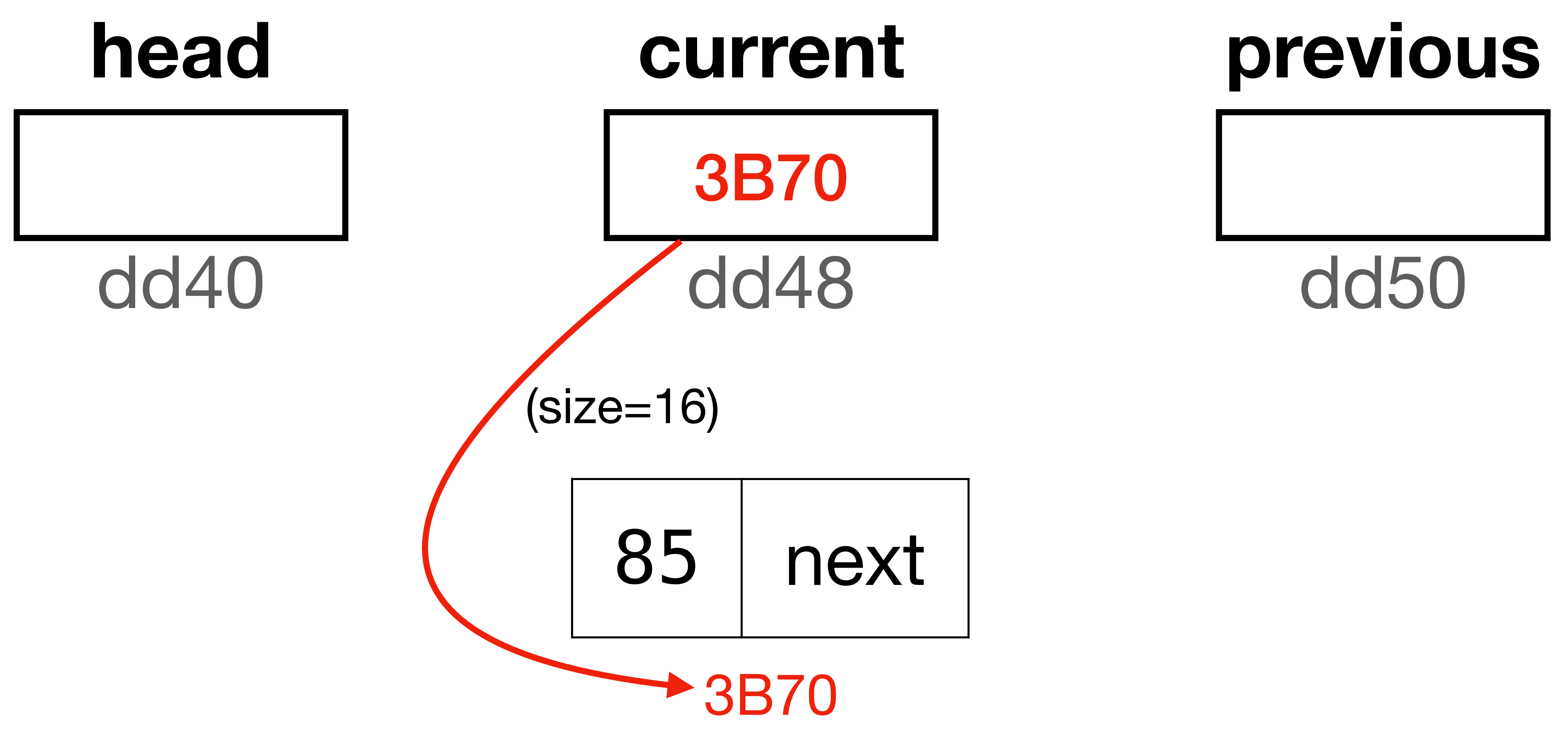


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```



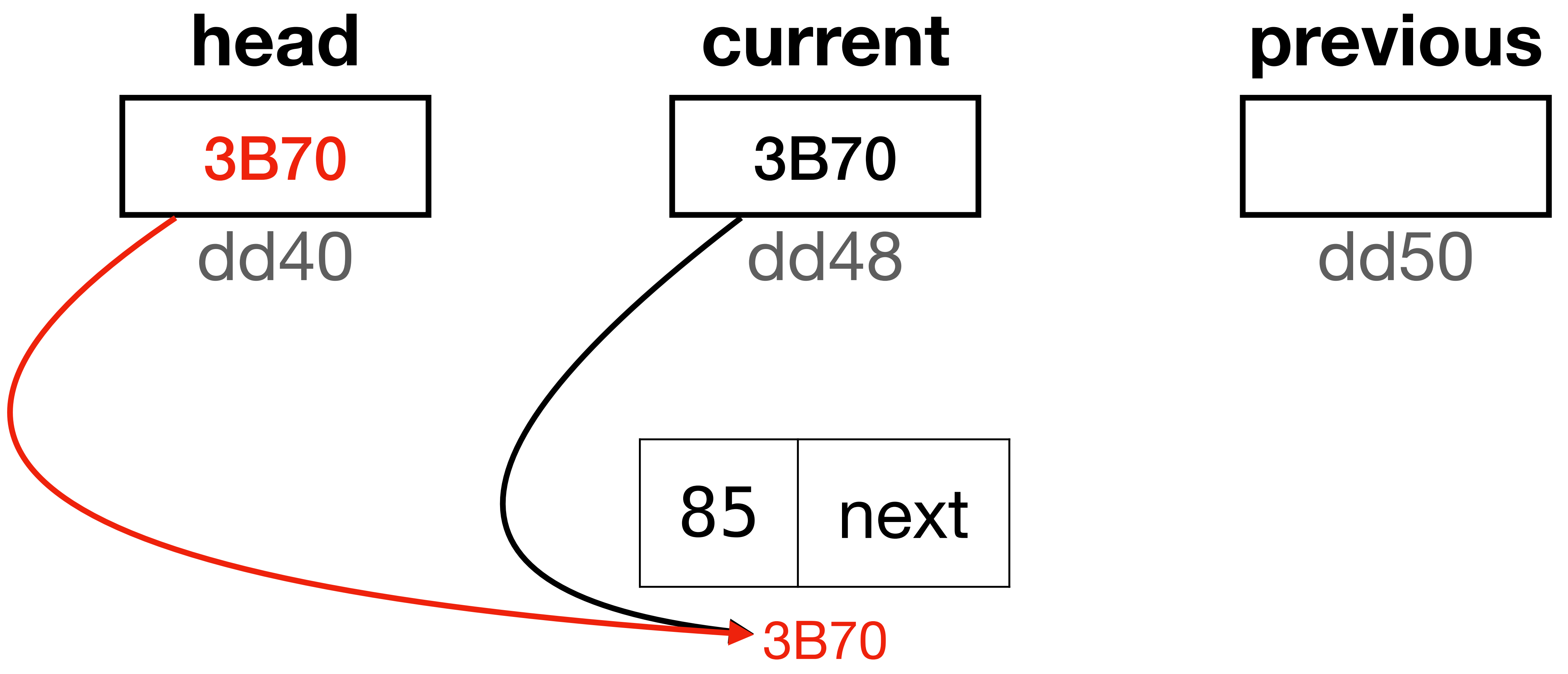
An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

i = 0

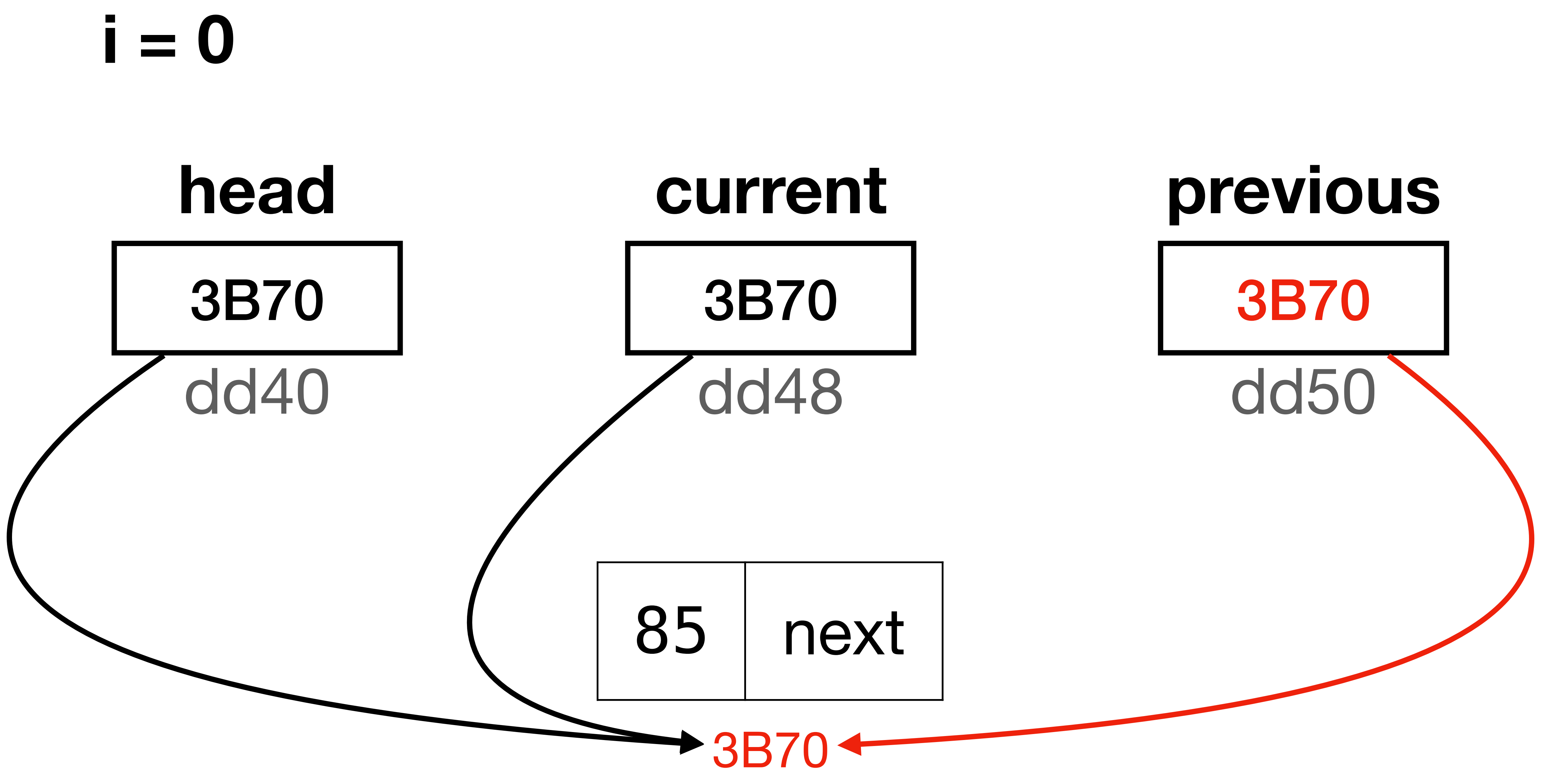


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

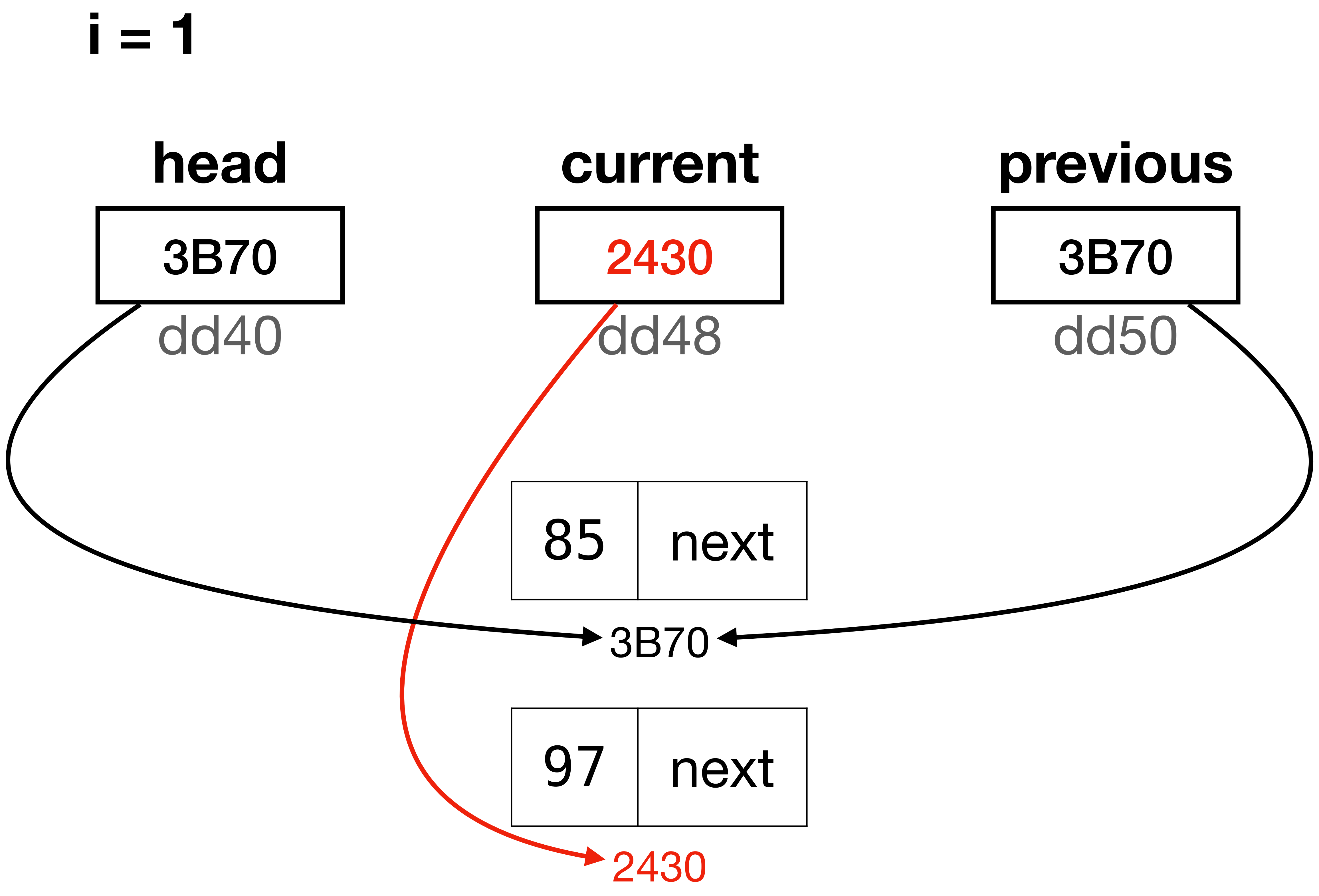


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```



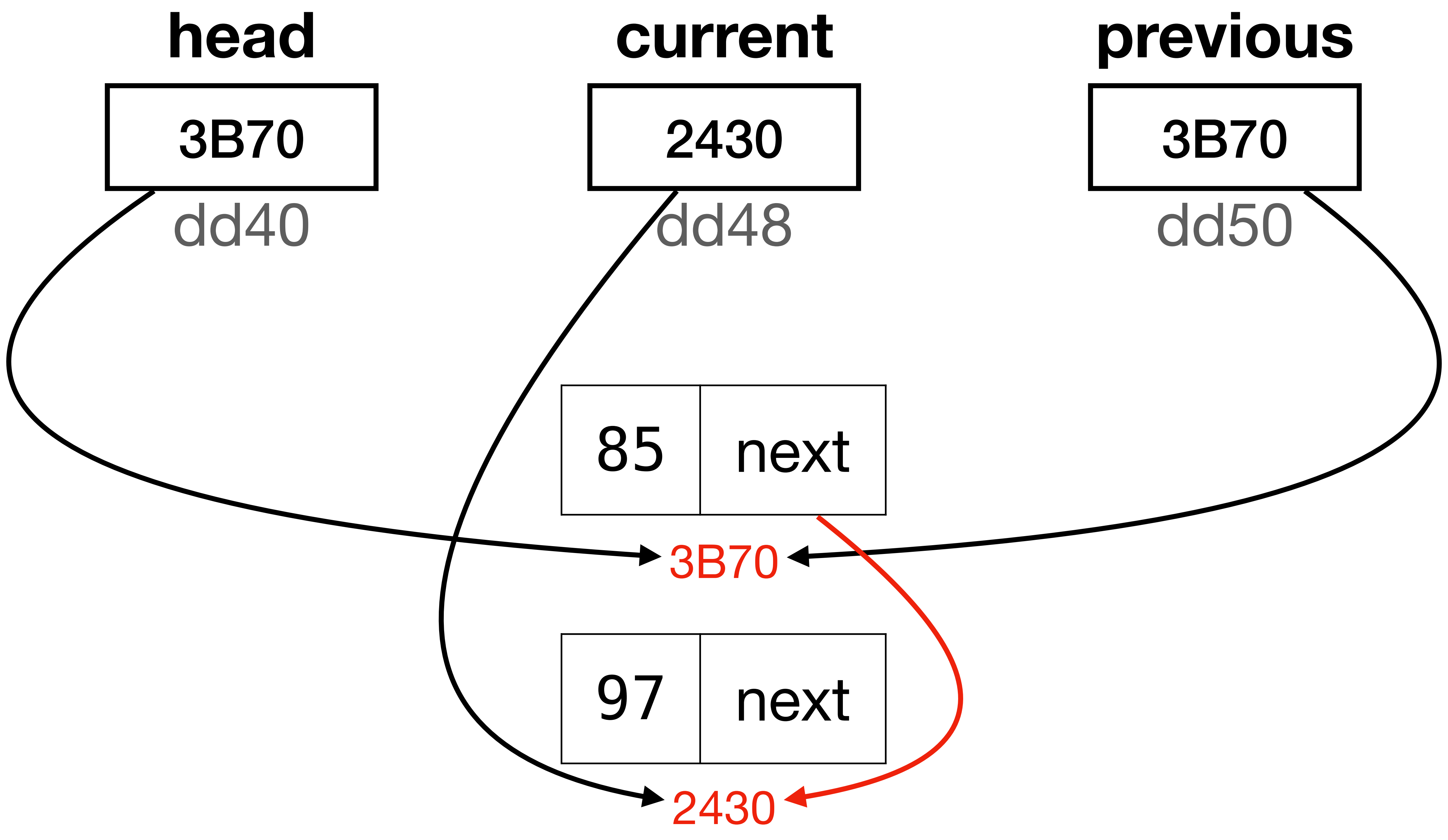
An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

i = 1



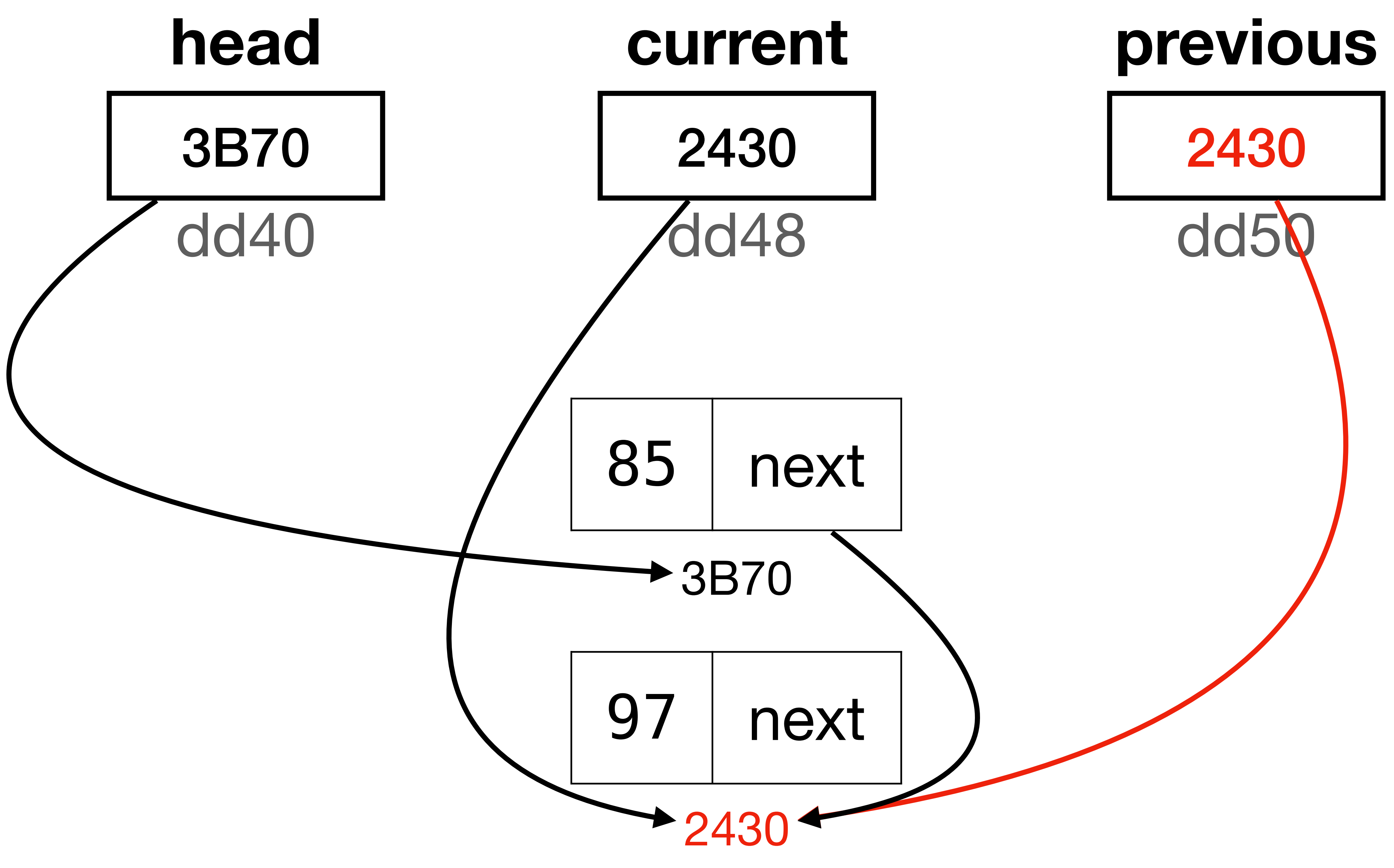
An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

i = 1



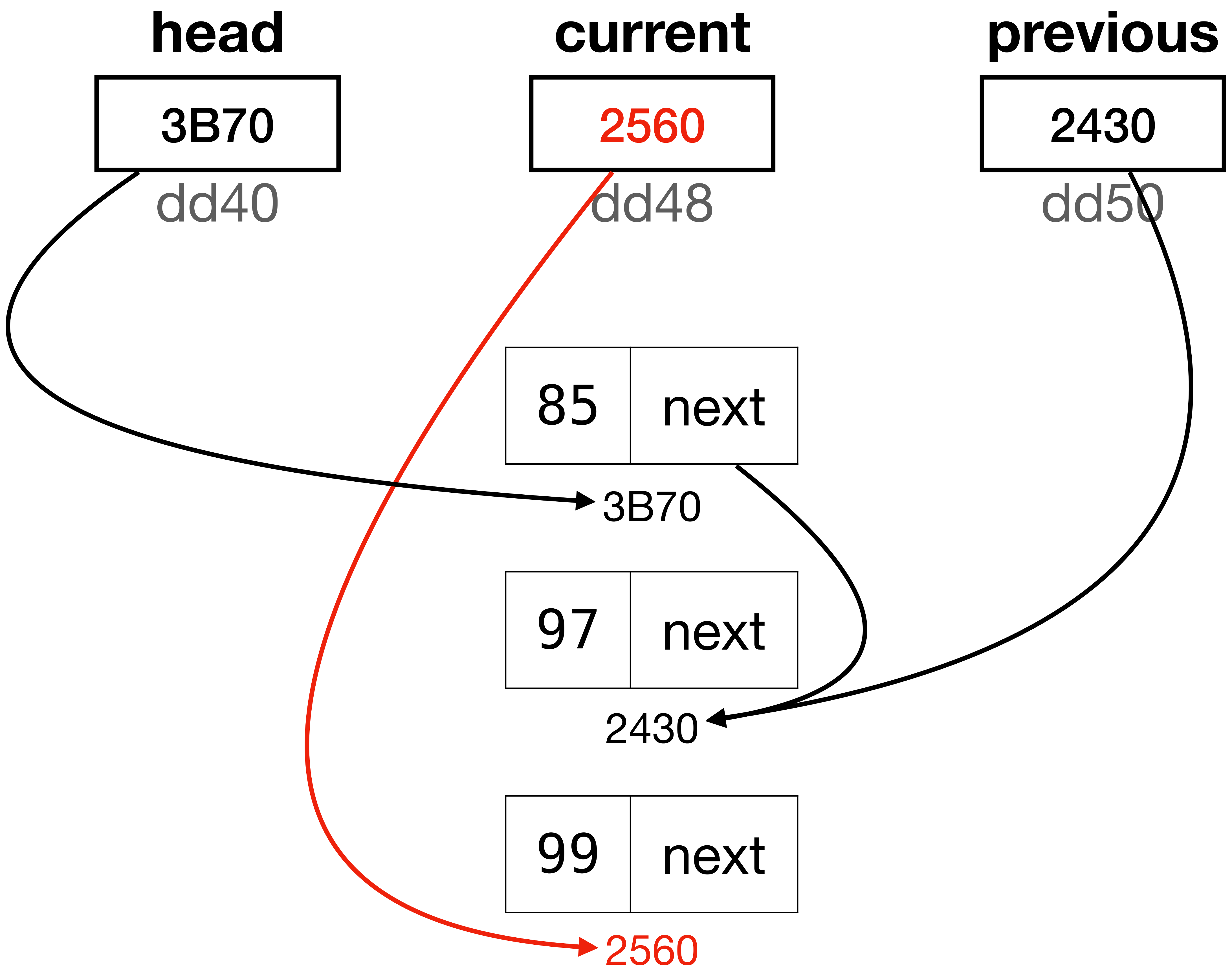
An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            head = current;
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

i = 2

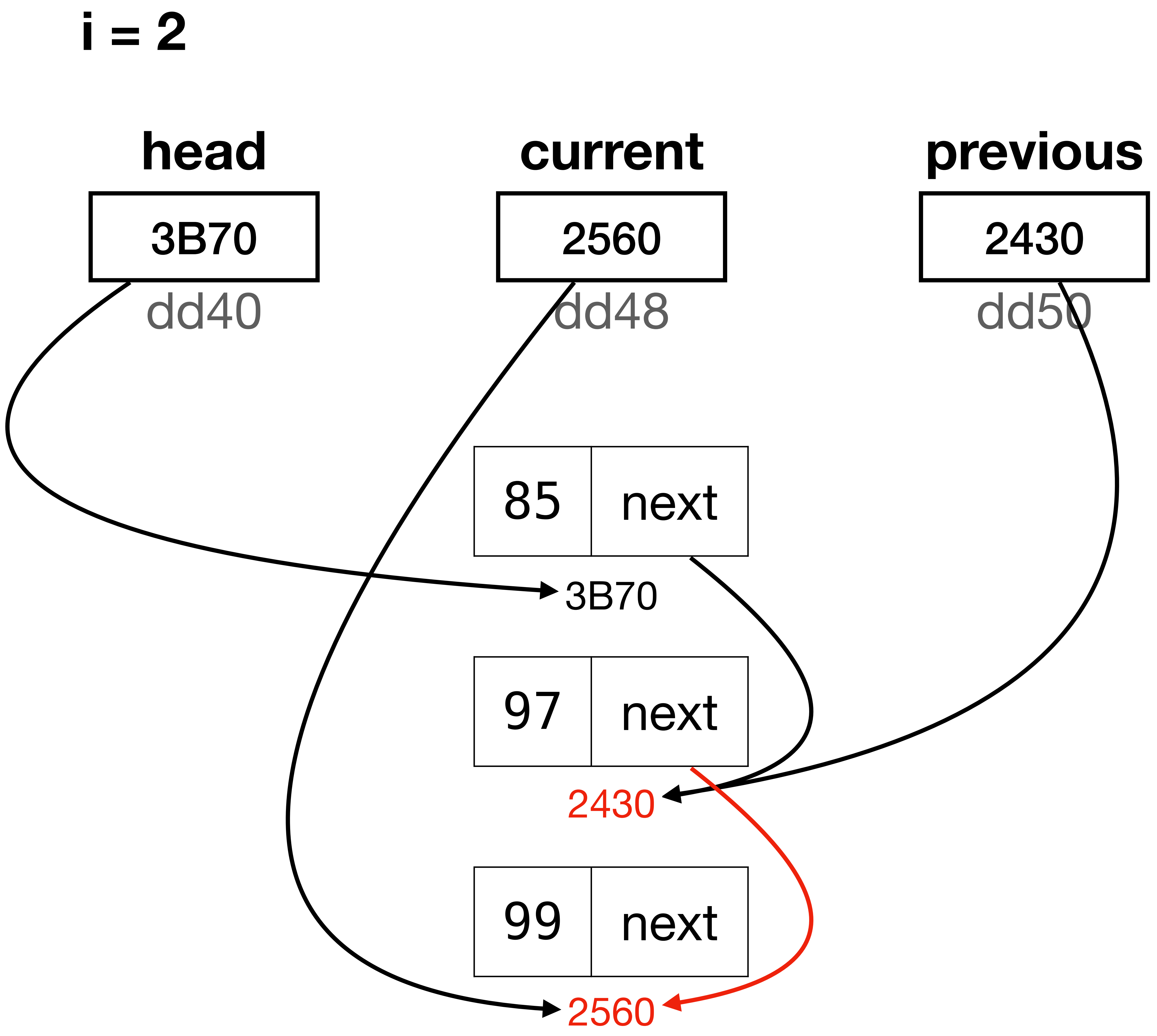


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

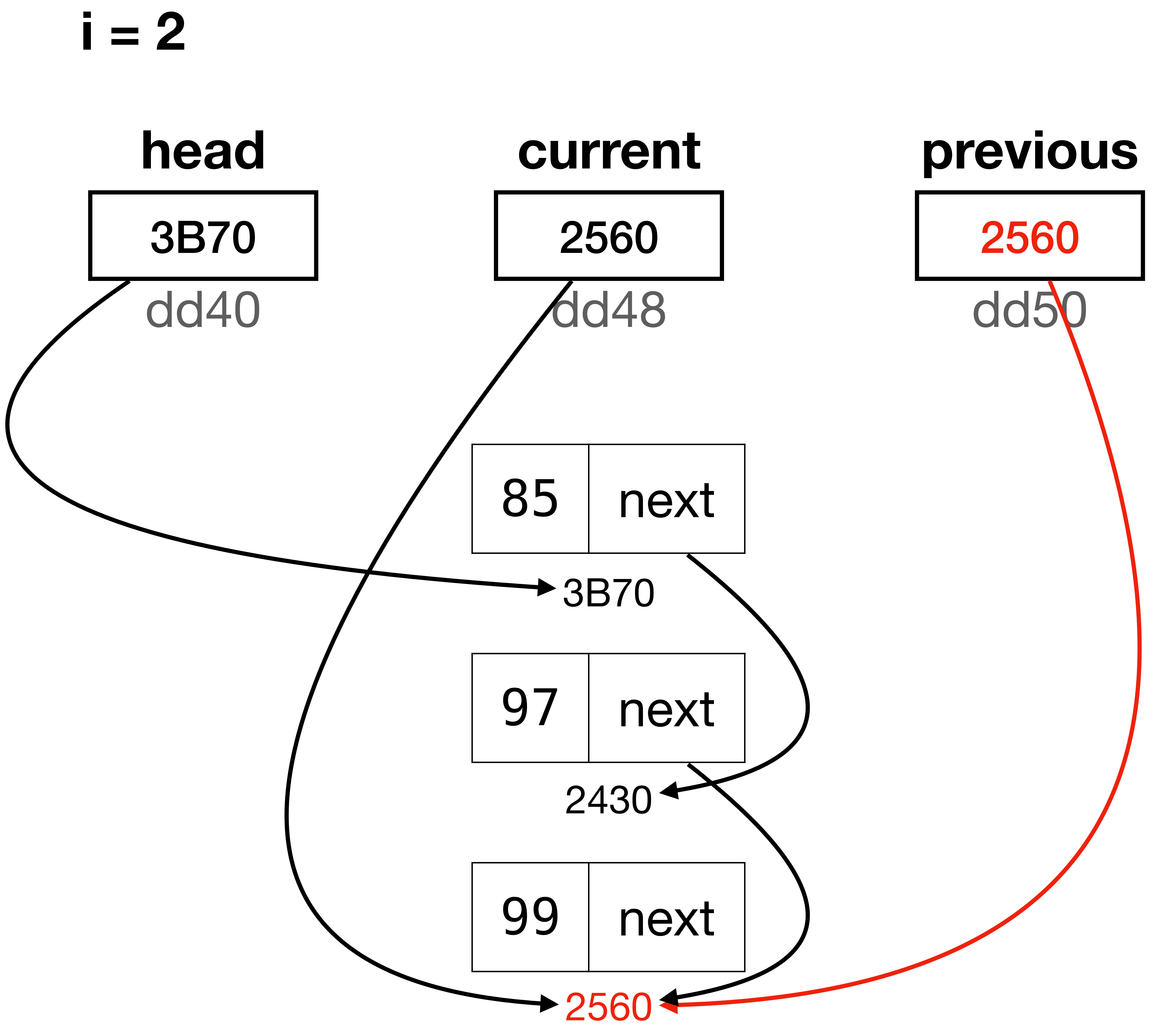


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

```
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```

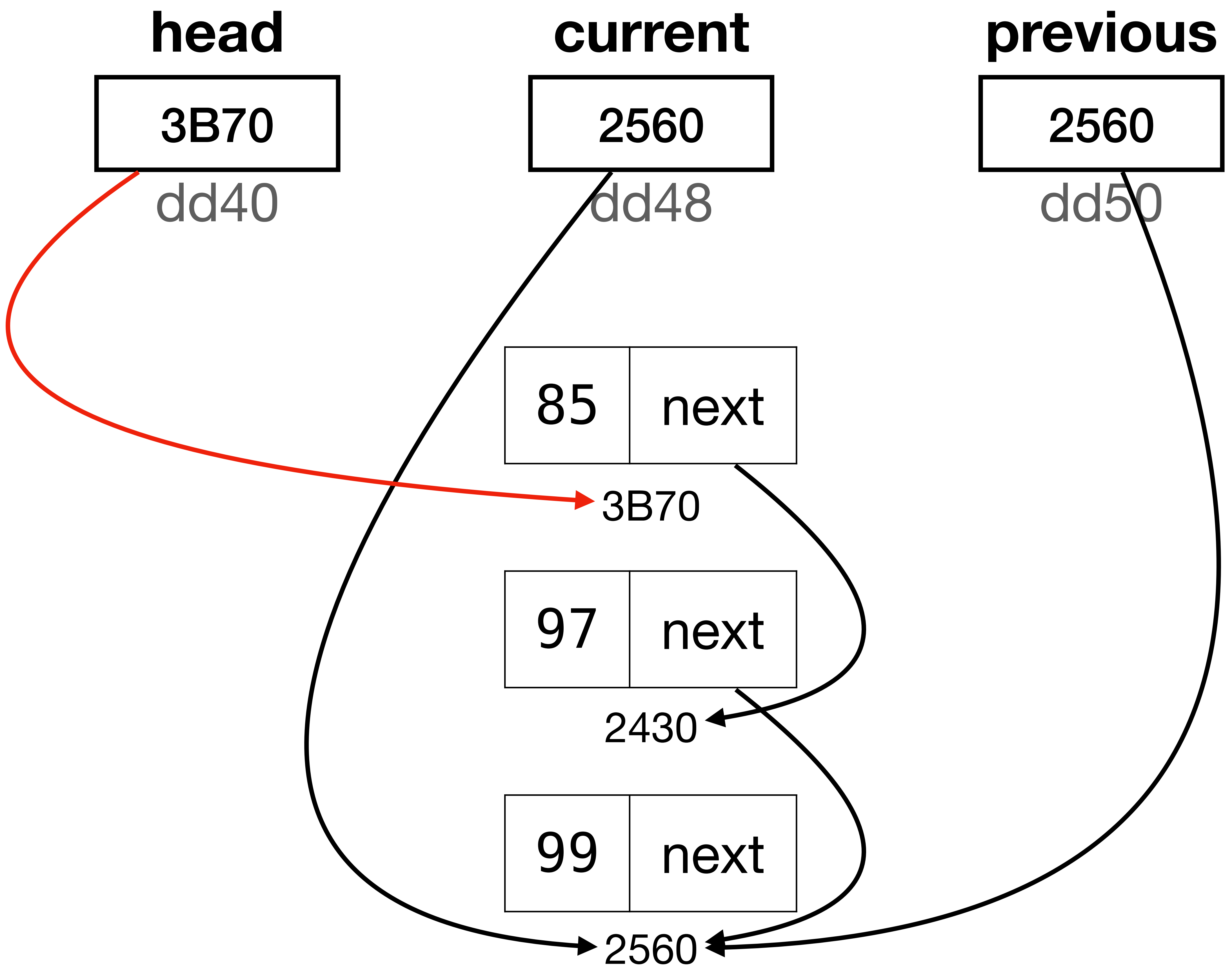


An Example of a Linked List (dynamic memory)

- Input: 85 97 99 100

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

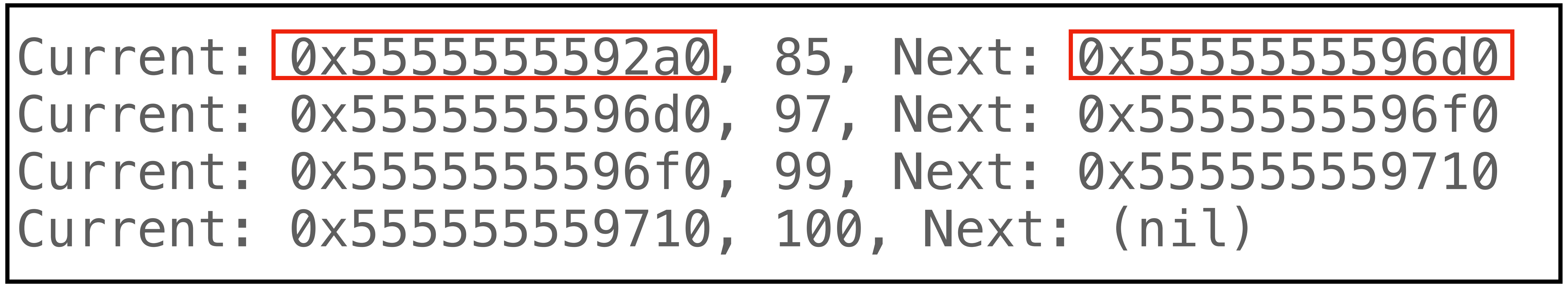
```c
int main(void){
    Node *head, *current, *previous;
    int i, num = 4;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        scanf("%d", &(current->val));
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        previous = current;
    }
    while (head != NULL){
        printf("Current: %p, ", head);
        printf("%d, ", head->val);
        printf("Next: %p\n", head->next);
        head = head->next;
    }
}
```



An Example of a Linked List (dynamic memory)

C-course-materials/09-DynamicMemoryAlloc/linked_list_malloc.c

- Output



```
Current: 0x5555555592a0, 85, Next: 0x5555555596d0
Current: 0x5555555596d0, 97, Next: 0x5555555596f0
Current: 0x5555555596f0, 99, Next: 0x555555559710
Current: 0x555555559710, 100, Next: (nil)
```

malloc does not guarantee the allocated memory addresses are continuous.

[Summary] Why do we need a linked list?

- A linked list is more flexible than an array: **we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed.**
- On the other hand, we lose the access capability of an array:
  - Any element of an array can be accessed in the same amount of time.
  - Accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it’s near the end.

Operations for Linked Lists

1. createList
2. freeList
3. printList
4. searchNode
5. insertNode
6. deleteNode

createList

C-course-materials/09-DynamicMemoryAlloc/createList.c

```
Node *createList(int arr[], int num){
    Node *head, *current, *previous;
    int i;
    for (i = 0; i < num; i++){
        current = (Node *)malloc(sizeof(Node));
        current->val = arr[i];
        if (i == 0){
            head = current;
        }
        else {
            previous->next = current;
        }
        current->next = NULL;
        previous = current;

    }
    return head;
}
```

We use three pointers (head, current, and previous) to build a linked list. See the animation in the previous slides.

printList

C-course-materials/09-DynamicMemoryAlloc/createList.c

```
void printList(Node *head){
    if (head == NULL){
        printf("List is empty\n");
    }
    else {
        while (head != NULL){
            printf("%d, %p\n", head->val, head);
            head = head->next;
        }
    }
}
```

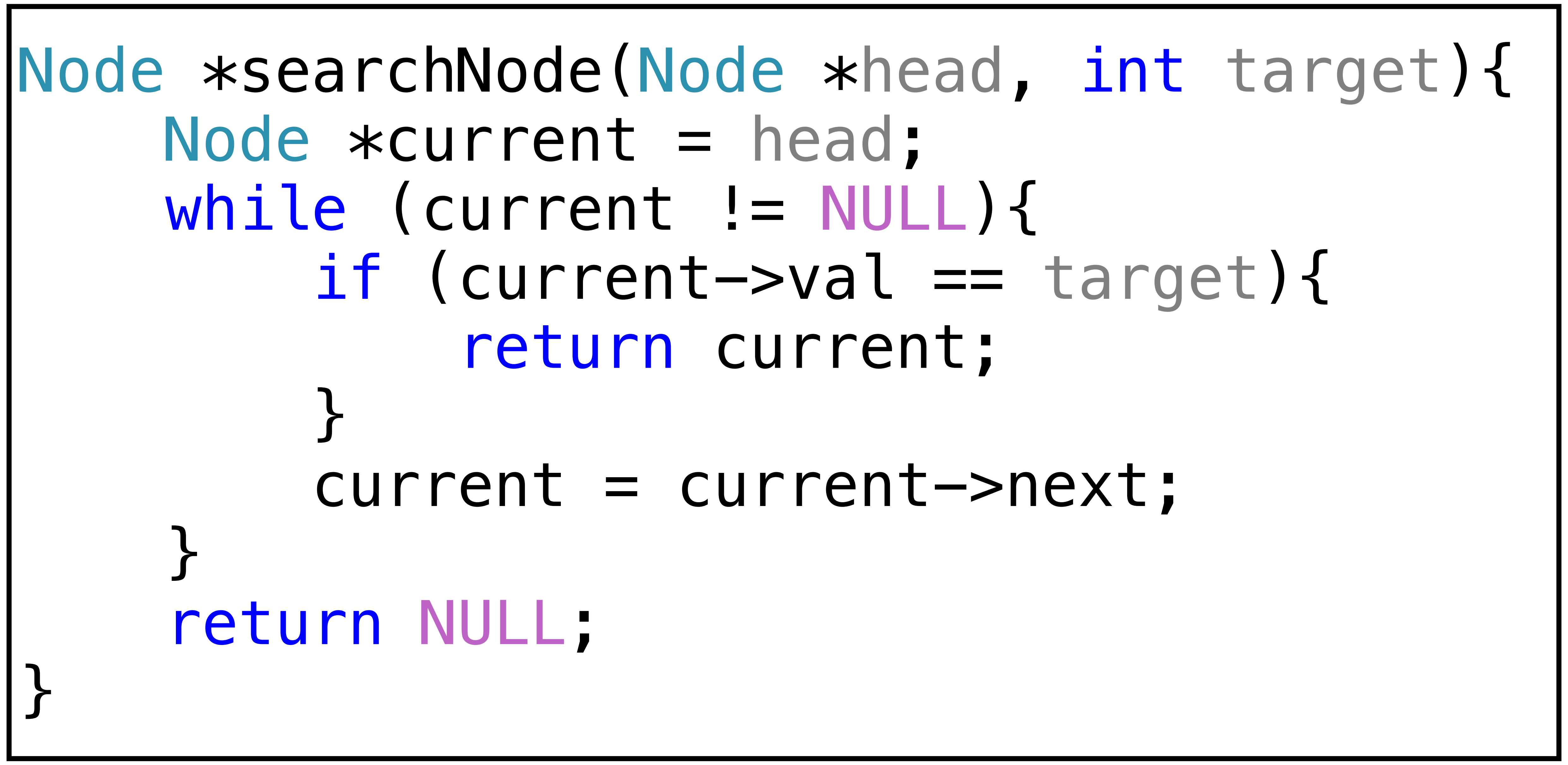
freeList

C-course-materials/09-DynamicMemoryAlloc/freeList.c

```
void freeList(Node *head){
    Node *current, *tmp;
    current = head;
    while (current != NULL){
        tmp = current;
        current = current->next;
        free(tmp);
    }
}
```

free the pointers in a linked list orderly.

searchNode

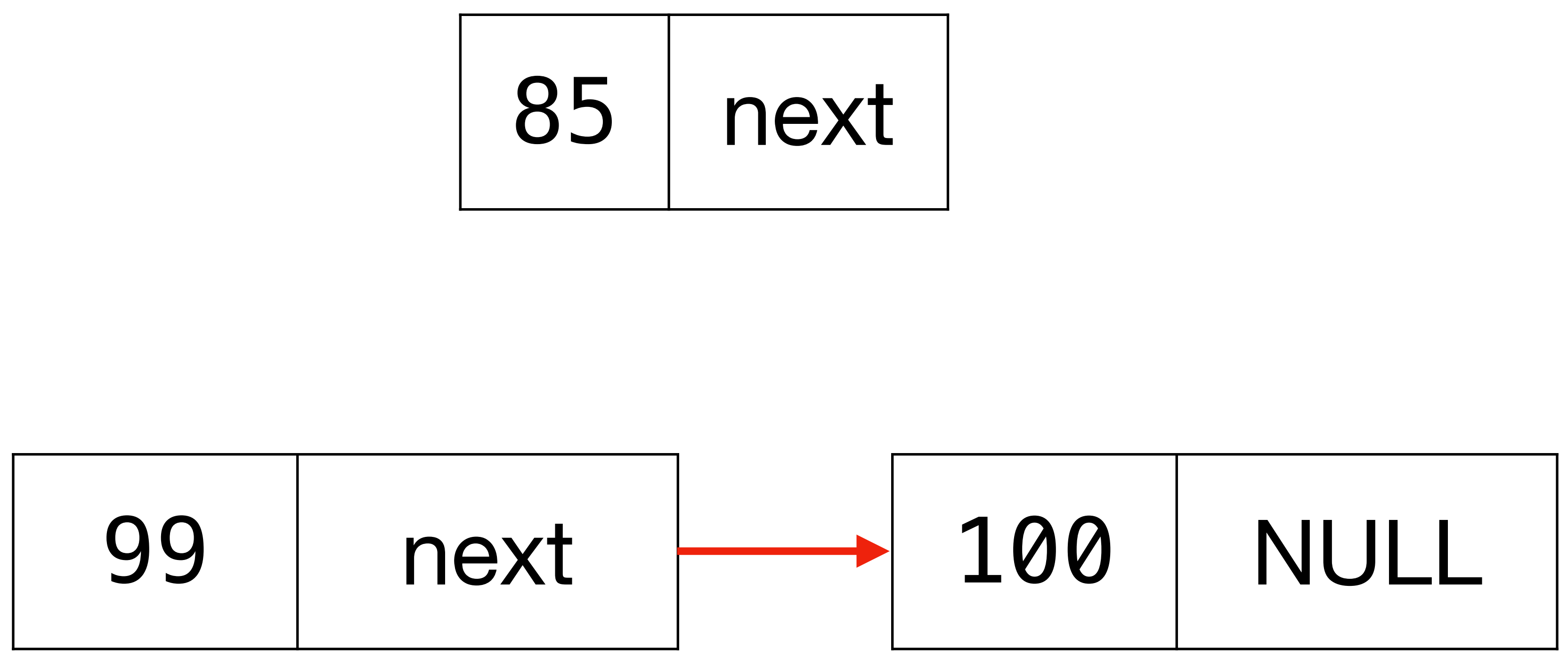


```
Node *searchNode(Node *head, int target){
    Node *current = head;
    while (current != NULL){
        if (current->val == target){
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

Question: In this case, do we need to create an additional Node (current)?

insertNode

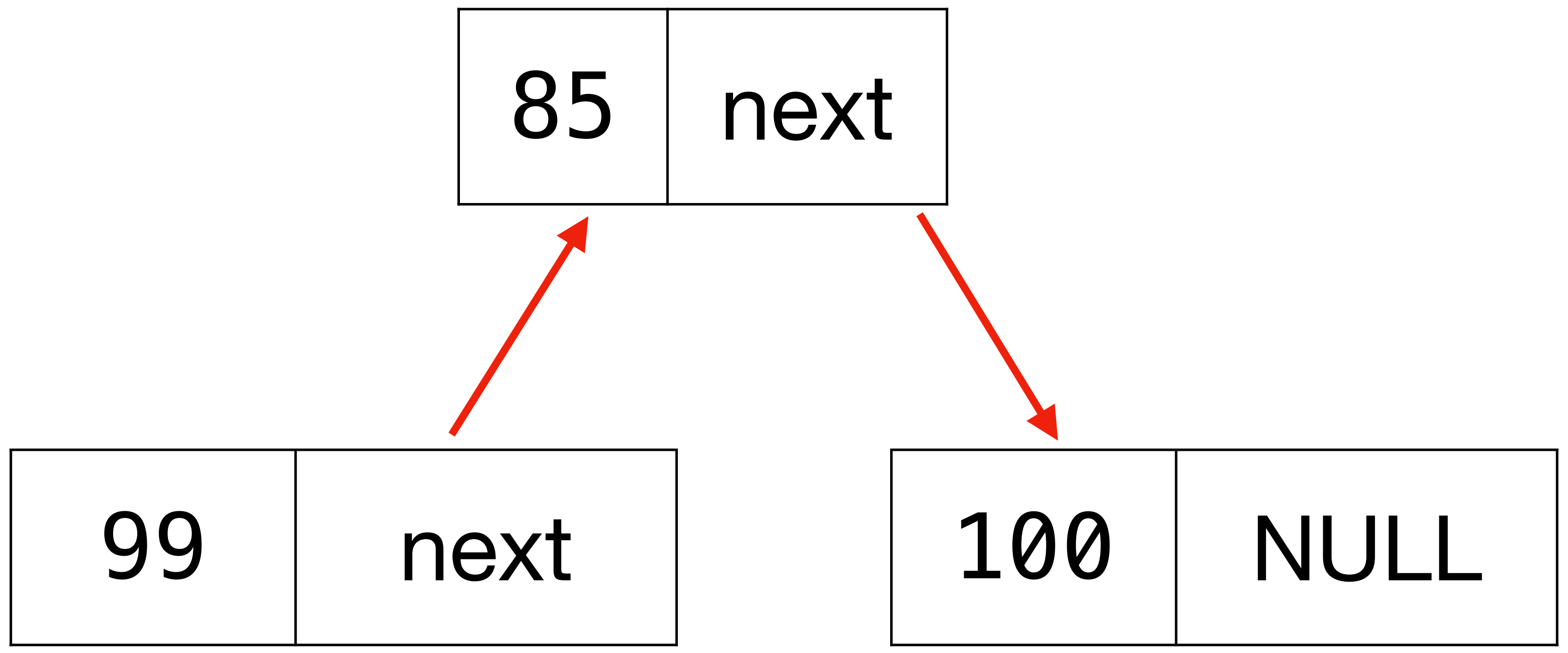
```
Node *insertNode(Node *node, int item){
    Node *newNode;
    newNode = (Node *)malloc(sizeof(Node));
    newNode->val = item;
    newNode->next = node->next;
    node->next = newNode;
}
```



To insert a node, typically, we need to first create a new node.

insertNode

```
Node *insertNode(Node *node, int item){
    Node *newNode;
    newNode = (Node *)malloc(sizeof(Node));
    newNode->val = item;
    newNode->next = node->next;
    node->next = newNode;
}
```



To insert a node, typically, we need to first create a new node.

deleteNode

```
Node *deleteNode(Node *head, Node *node){
    if (head == NULL){
        return NULL;
    }
    if (node == head){
        head = head->next;
    }
    else{
        Node *current = head;
        while (current->next != node){
            current = current->next;
        }
        // found the node before the node to be deleted
        current->next = node->next;
    }
    free(node);
    return head;
}
```