[SWE2015-41] Introduction to Data Structures (자료구조개론)

Linked Lists

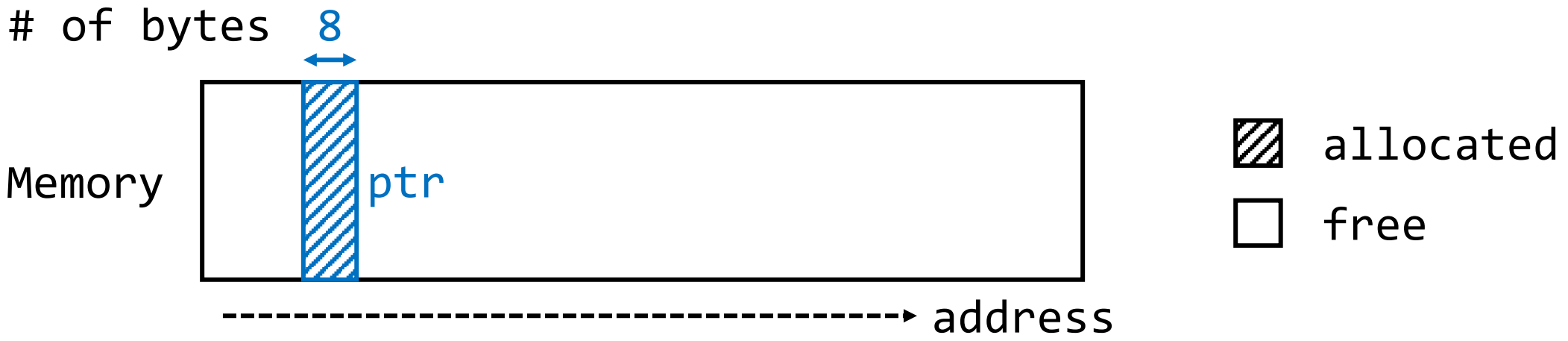
Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) Dynamic Memory Allocation

- `malloc(size)` - allocate `size` bytes consecutively & return its address
- `free(address)` - release the allocated memory
  - You must free the dynamically allocated memory after using it!

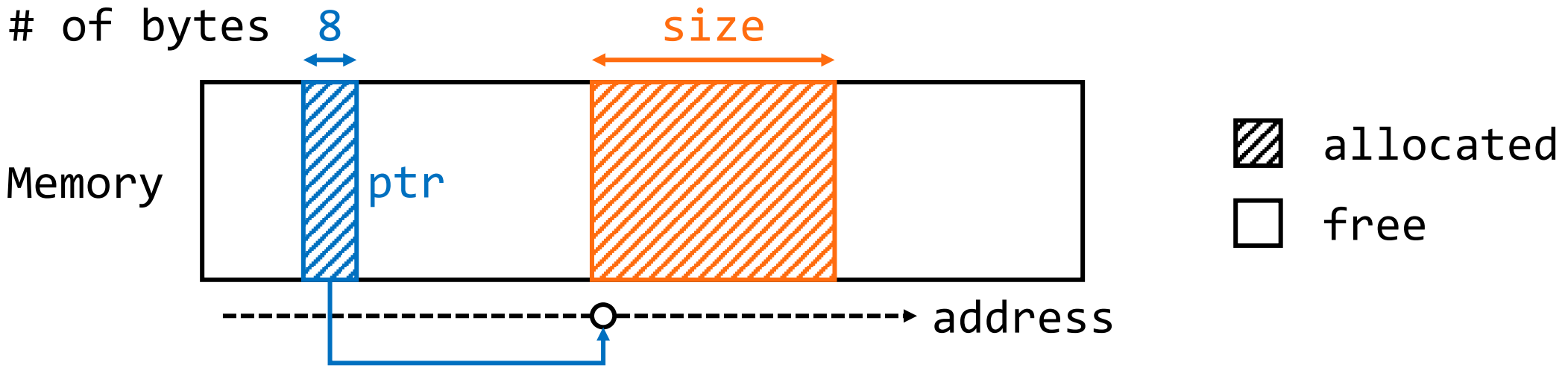
```
int *ptr;
```



(Recap) Dynamic Memory Allocation

- `malloc(size)` - allocate `size` bytes consecutively & return its address
- `free(address)` - release the allocated memory
  - You must free the dynamically allocated memory after using it!

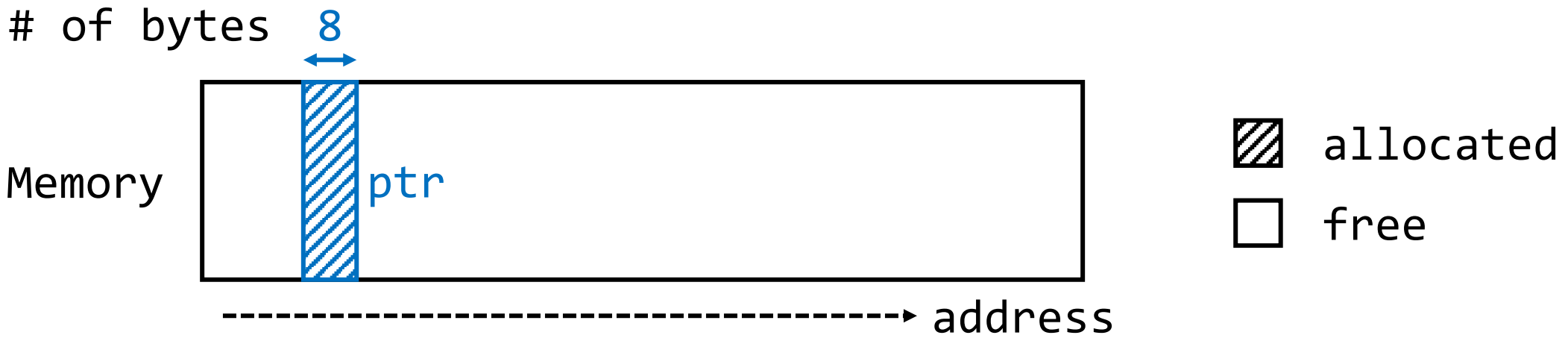
```
int *ptr;
ptr = (int *)malloc(size);
```



(Recap) Dynamic Memory Allocation

- `malloc(size)` - allocate `size` bytes consecutively & return its address
- `free(address)` - release the allocated memory
  - You must free the dynamically allocated memory after using it!

```
int *ptr;
ptr = (int *)malloc(size);
free(ptr);
```



(Recap) Dynamic Memory Allocation

- `malloc(size)` - allocate `size` bytes consecutively & return its address
- `free(address)` - release the allocated memory
  - You must free the dynamically allocated memory after using it!

main.c

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 3, *ptr;
    ptr = (int *)malloc(sizeof(int)*n);
    ptr[0] = 50; ptr[1] = 100; ptr[2] = 150;
    for (int i = 0; i < 3; i++) printf("%d\n", ptr[i]);
    free(ptr);
    return 0;
}
```

(Recap) Arrays

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**

Declaration in C

```
type name[size] = { ... };
```

main.c

```
int numbers[10] = {
    1, 5,  9, -3, 8,
    7, 6, 10, -5, 0
};
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

(Recap) Arrays

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**
- The `i`-th element can be accessed by `arr[i]`
- Time complexity for the access = $O(1)$
  - Address computation requires $O(1)$

```
numbers = &numbers[0] = 0x16aedf320
&numbers[7] = &numbers[0] + 7
            = 0x16aedf33c
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

(Recap) Arrays

- How to insert an item into `arr[]`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	6
3	0x16aedf32c	9
4	0x16aedf330	-3
5	0x16aedf334	8
6	0x16aedf338	7
n=7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Insert 6 at the 3rd position

(Recap) Arrays

- How to insert an item into `arr[]`?

main.c

```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index
    arr[position] = item;
    // 3. Increase size
    size += 1;
    return size;
}
```

  - Time complexity for this insertion = $O(n)$ where n is the number of elements

(Recap) Arrays

- How to delete an item from `arr[]`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	-3
3	0x16aedf32c	8
4	0x16aedf330	7
n=5	0x16aedf334	null
6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Delete 3rd item (value=9, index=2)

(Recap) Arrays

- How to delete an item from `arr[]`?

main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```

  - Time complexity for this deletion = $O(n)$ where n is the number of elements

(Recap) Array Operations

Operation	Time Complexity
Insertion	$O(n)$
Deletion	$O(n)$
Search by Index (Access)	$O(1)$
Search by Value	$O(n)$

- When array structures are inefficient?
  - When `Insertion` or `Deletion` at the middle frequently occurs
  - When `Search by Value` is frequently required

- When array structures are useful?
  - When `Insertion` and `Deletion` only occurs at the end
  - When `Access` frequently occurs

(Recap) Abstraction for Implementation

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  (Q) What operations are necessary?

(Recap) Abstraction for Implementation

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  - `items[]` - the physical memory allocated for storing elements
  - `size` - the number of the elements stored

  (Q) What operations are necessary?

(Recap) Abstraction for Implementation

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  - `items[]` - the physical memory allocated for storing elements
  - `size` - the number of the elements stored

  (Q) What operations are necessary?

  - `insert()` - insert an element to the array
  - `delete()` - delete an element from the array
  - `getSize()` - count the number of elements in the array
  - `isEmpty()` - check whether the array is empty or not
  - `isFull()` - check whether the array is full or not
  - …

(Recap) Abstraction for Implementation


```c
#include <stdbool.h>        // This enables to use bool type
#define MAX_SIZE 10000      // Maximum size of our array structure

typedef struct _IntArray { // Array structure for integer values
    int items[MAX_SIZE];
    int size;
} IntArray;

// IntArray operations:
void insert(IntArray *arr, int item, int index);
void delete(IntArray *arr, int index);
bool isFull(IntArray *arr);
bool isEmpty(IntArray *arr);
int getSize(IntArray *arr);
int getMax(IntArray *arr);
int getMin(IntArray *arr);
int getSum(IntArray *arr);
...
```

(Recap) Abstraction for Implementation


```
void insert(IntArray *arr, int item, int index) {
    if (index < 0 || index > arr->size || !isFull(arr)) return;
    for (int i = arr->size-1; i >= index; i --)
        arr->items[i+1] = arr->items[i];
    arr[index] = item;
    arr->size ++;
}

void delete(IntArray *arr, int index) {
    if (index < 0 || index >= arr->size) return;
    for (int i = index; i < arr->size-1; i ++)
        arr->items[i] = arr->items[i+1];
    arr->size --;
}

bool isFull(IntArray *arr) {
    return arr->size == MAX_SIZE;
}
```

Code Abstraction Principle

Each significant piece of functionality in a program
should be implemented in just one place in the source code.

Where similar functions are carried out by distinct pieces of code,
it is generally beneficial to combine them into one
by abstracting out the varying parts.

by Benjamin C. Pierce in Types and Programming Languages (2002)
https://en.wikipedia.org/wiki/Abstraction_principle_(computer_programming)

Code Abstraction Principle

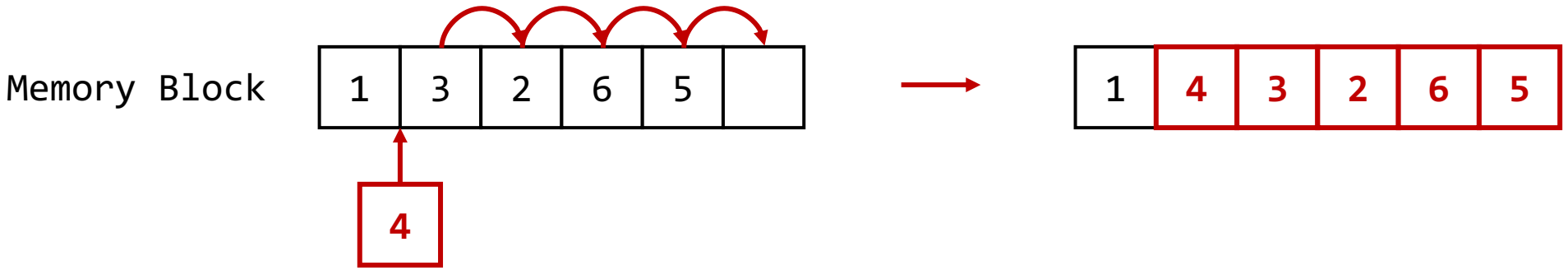
- **How to effectively implement** a data structure?
  - **(Q)** What type of data should be stored?
  - **(Q)** What operations are necessary?

  - **(Q)** How to simplify code?
  - **(Q)** How to reduce code duplication?
  - **(Q)** How to make code reusable and portable?

- Before implementation, you must think how to implement it abstractly!

Linked Lists - Motivation

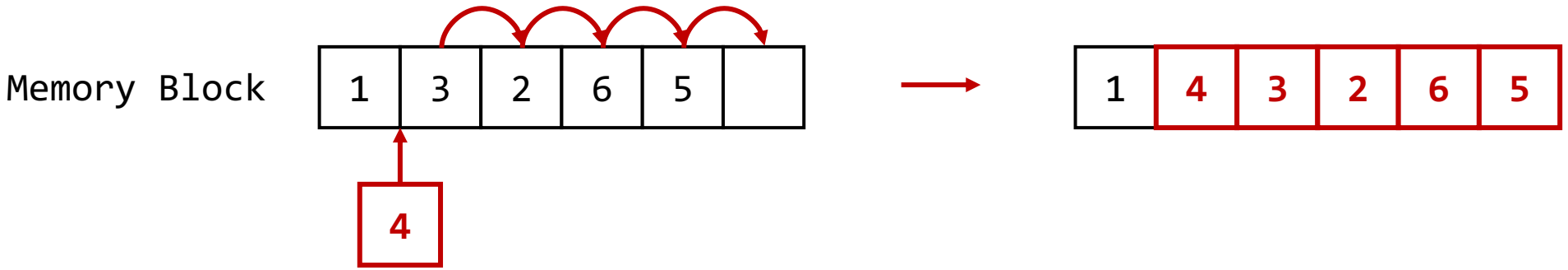
- **An array** is a collection of elements in **a contiguous block of memory**
  - **Main weakness of the array:** $O(n)$ time for insertion & deletion
  - It requires to re-allocate elements for maintaining the contiguous memory block



- **(Q)** Can we insert/delete an element without re-allocating elements?

Linked Lists - Motivation

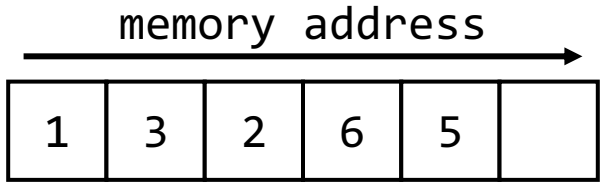
- **An array** is a collection of elements in **a contiguous block of memory**
  - **Main weakness of the array:** $O(n)$ time for insertion & deletion
  - It requires to re-allocate elements for maintaining the contiguous memory block



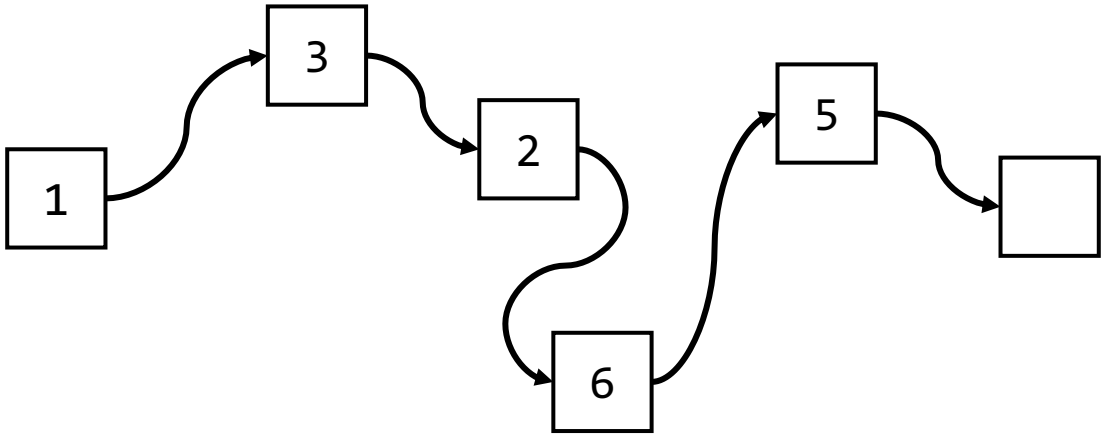
- **(Q)** Can we insert/delete an element without re-allocating elements?
- **(A)** Yes, this can be achieved **using Linked Lists**!
  - It requires only $O(1)$ time for insertion & deletion

Linked Lists - High-level Concept

- **An array** is a collection of elements in **a contiguous block of memory**

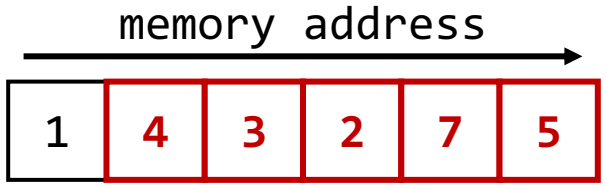


- **A linked list** is a collection of **sequentially-connected** elements
  - The elements are not required to be stored in contiguous memory



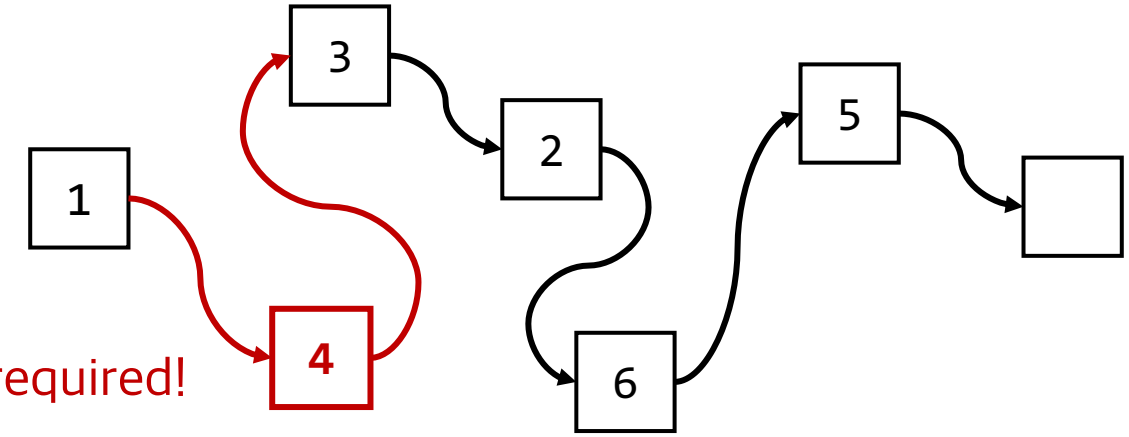
Linked Lists - High-level Concept

- **An array** is a collection of elements in **a contiguous block of memory**



Elements are re-allocated …

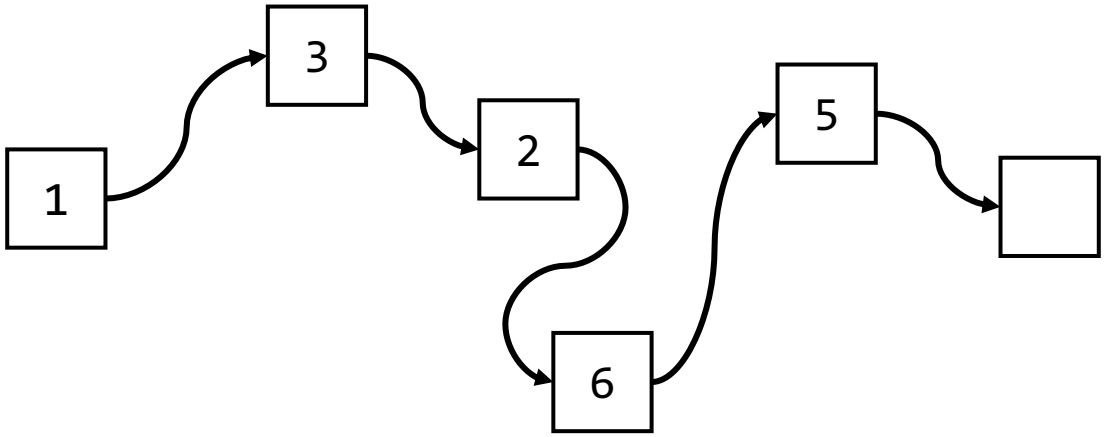
- **A linked list** is a collection of **sequentially-connected** elements
  - The elements are not required to be stored in contiguous memory



Re-allocation is not required!

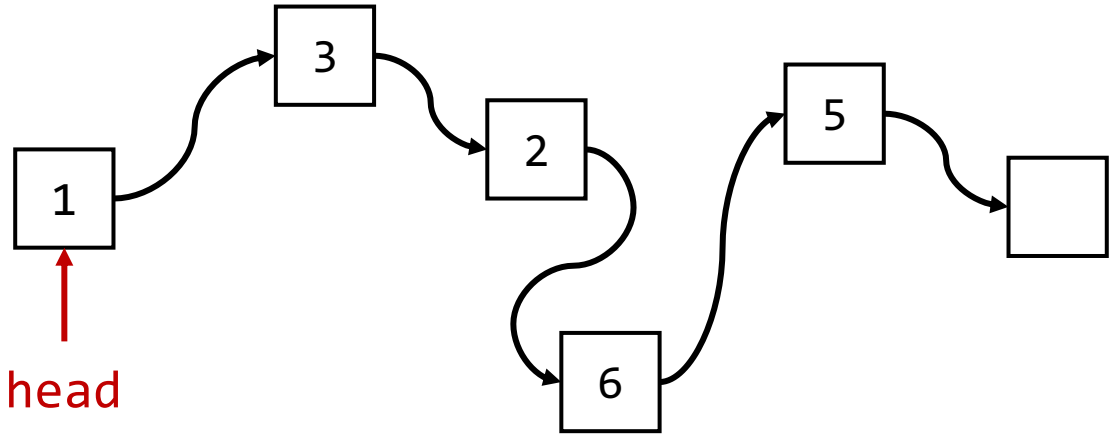
Linked Lists - Implementation

- How to implement the linked list structure?



Linked Lists - Implementation

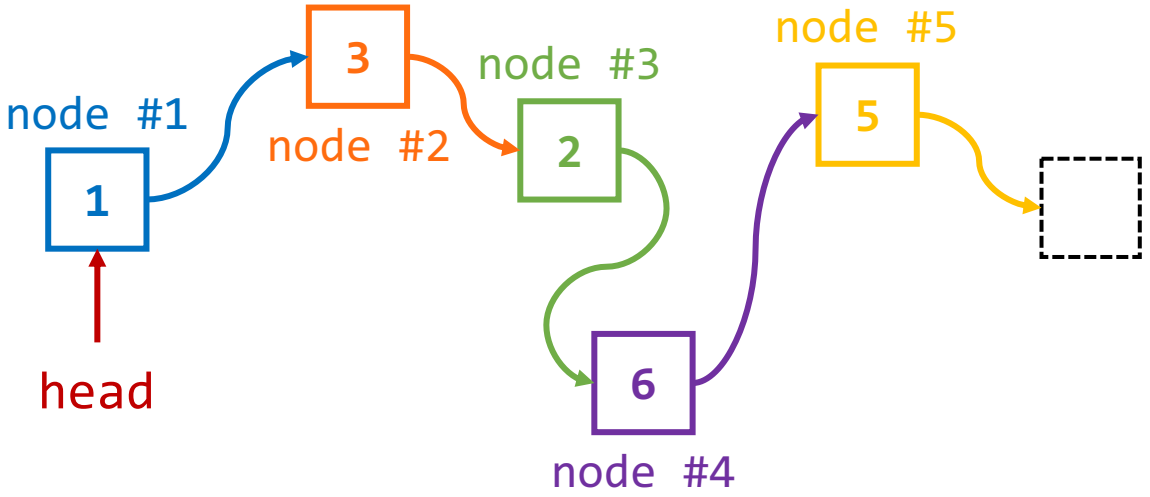
- How to implement the linked list structure?



  - head - the starting point of the linked list

Linked Lists - Implementation

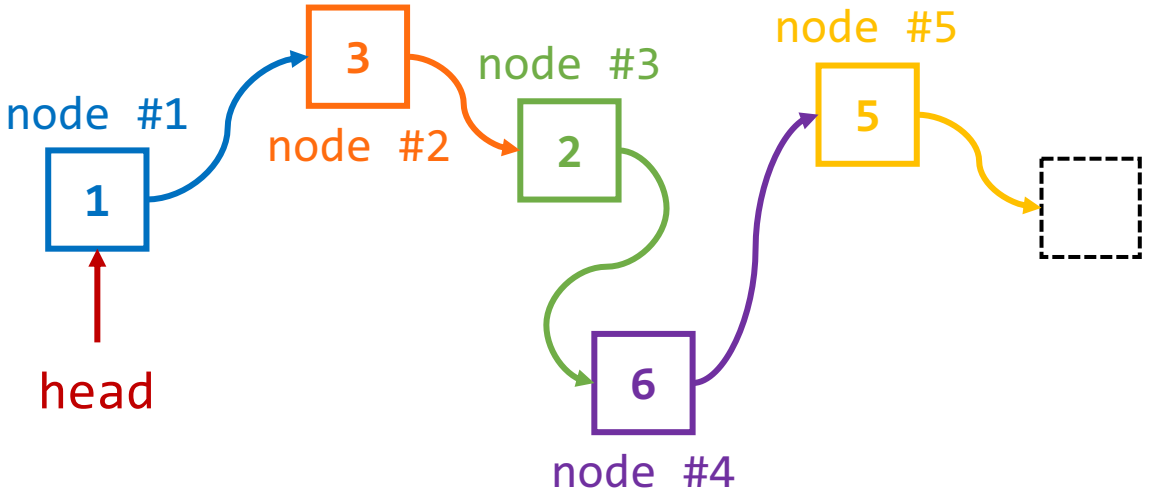
- How to implement the linked list structure?



  - `head` - the starting point of the linked list
  - `nodes` - each node contains its item value and its pointer to the next node

Linked Lists - Implementation

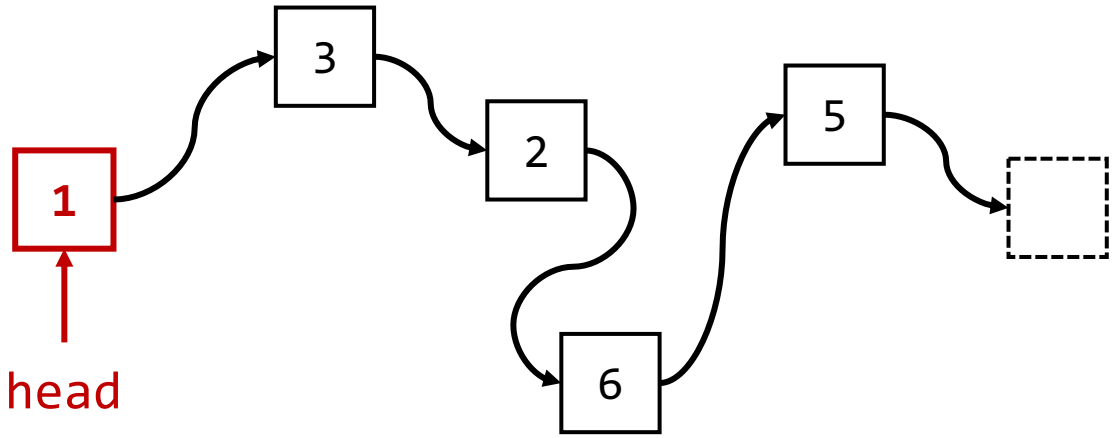
- How to implement the linked list structure?




```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

Linked Lists - Implementation

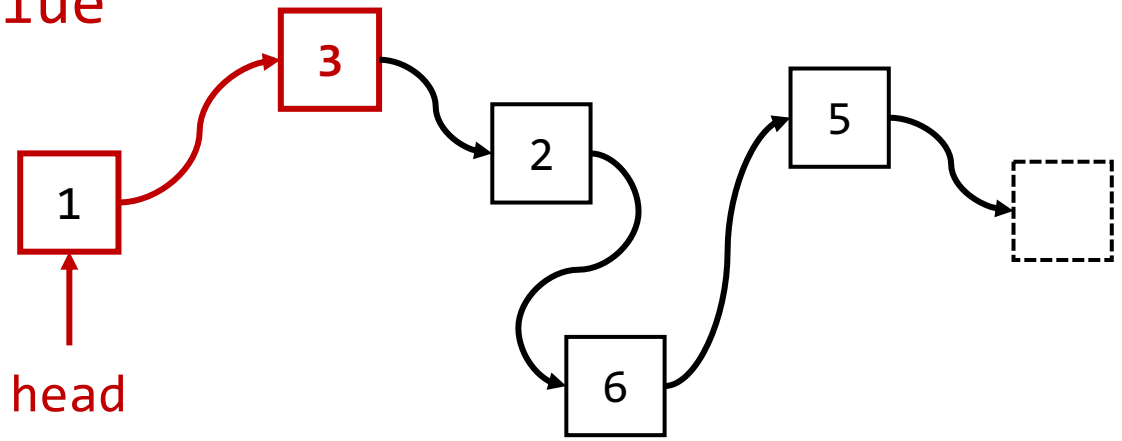
- How to access the 1st element in the linked list structure?
  - `head->value`




```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

Linked Lists - Implementation

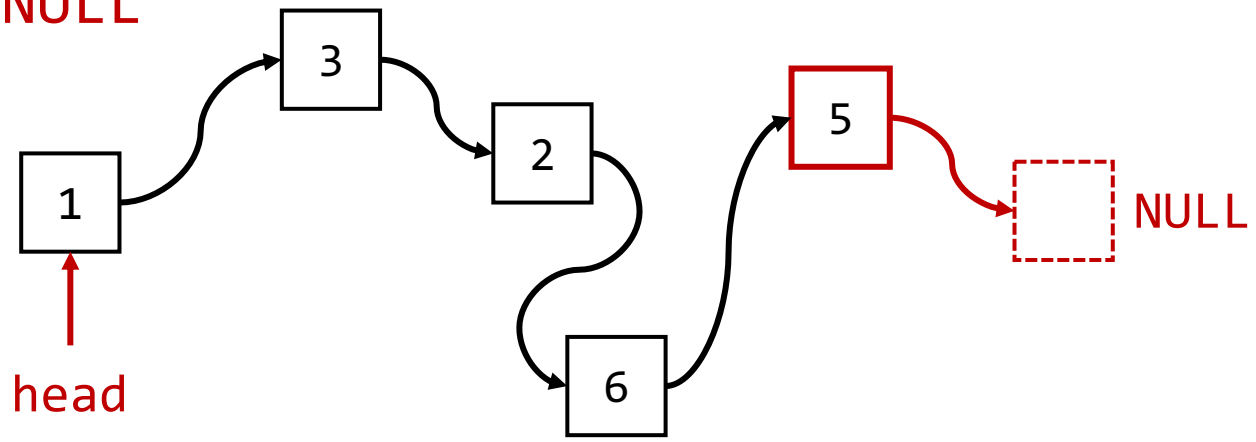
- How to access the 2nd element in the linked list structure?
  - `head->next->value`




```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

Linked Lists - Implementation

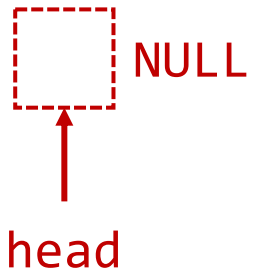
- Where is the end of the linked list structure?
  - `node->next == NULL`




```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

Linked Lists - Implementation

- What is the empty state of the linked list structure?
  - `head == NULL`

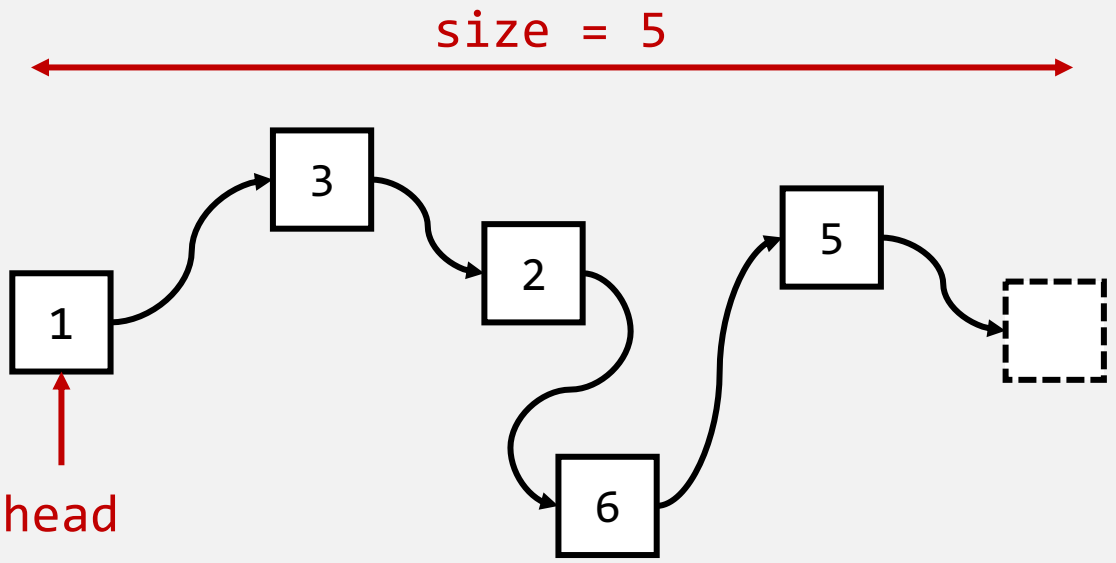



```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

Linked Lists - Length (Size)

- Now, we can compute **the size (# of elements)** of a linked list

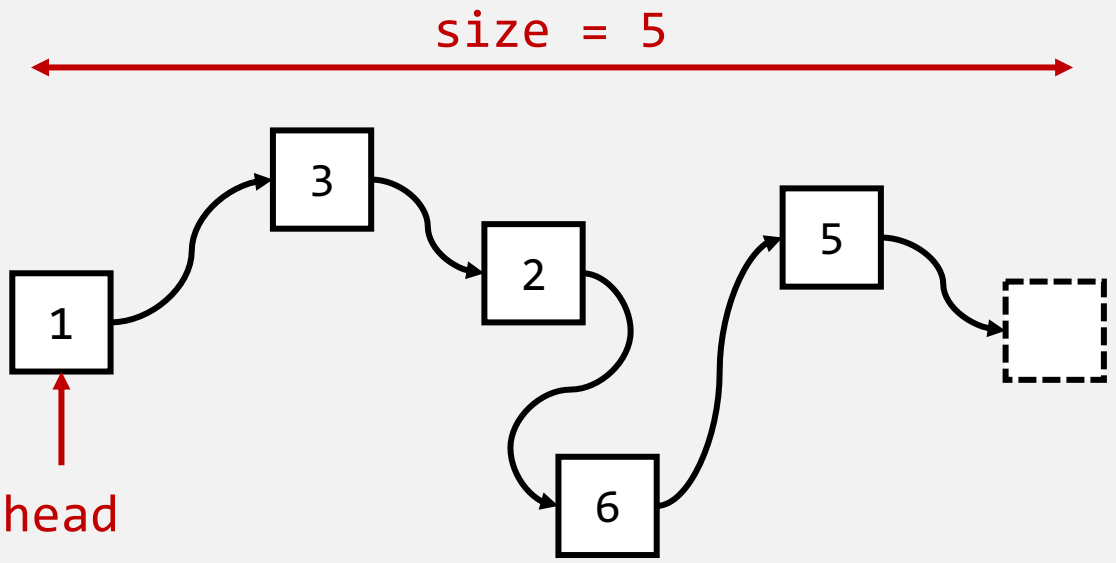
```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

LinkedList createLinkedList() { // This creates an empty list
    LinkedList newList = { NULL };
    return newList;
}

int getSize(LinkedList *list) {

                ?

}
```



Linked Lists - Length (Size)

- Now, we can compute **the size (# of elements)** of a linked list

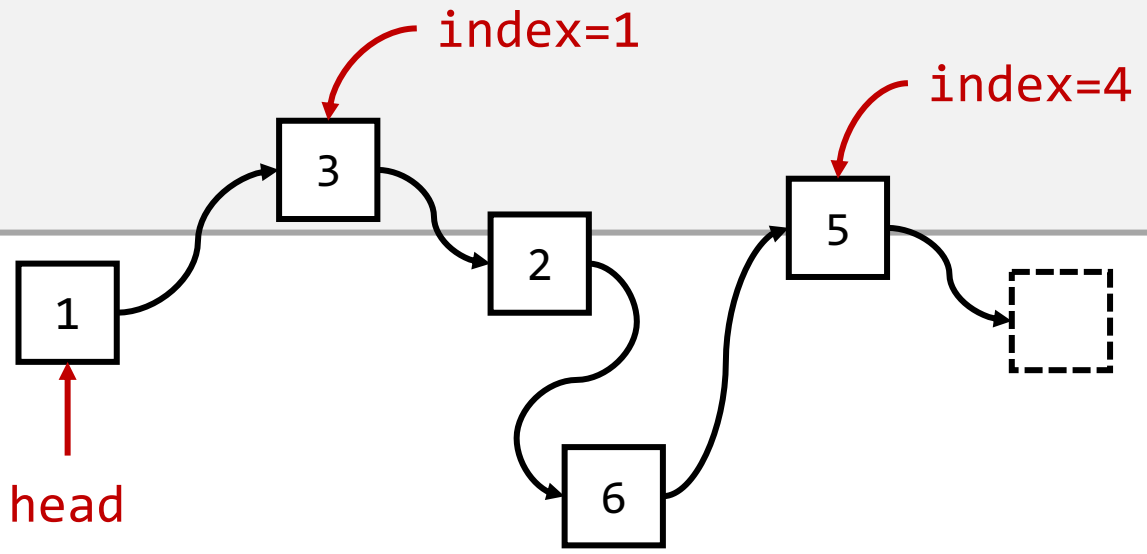
```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

LinkedList createLinkedList() { // This creates an empty list
    LinkedList newList = { NULL };
    return newList;
}

int getSize(LinkedList *list) {
    Node *node = list->head;
    int size = 0;
    while (node != NULL) {
        node = node->next;
        size ++;
    }
    return size;
}
```



Linked Lists - Access

- How to **access** the i-th element?

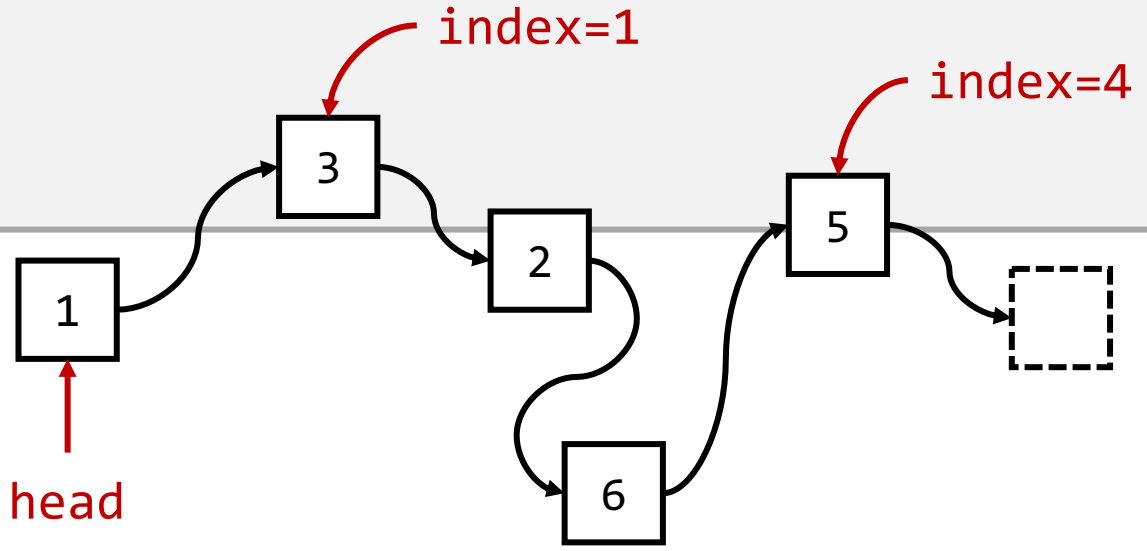
```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

Node* getNode(LinkedList *list, int index) {



                    ?



}
```



Linked Lists - Access

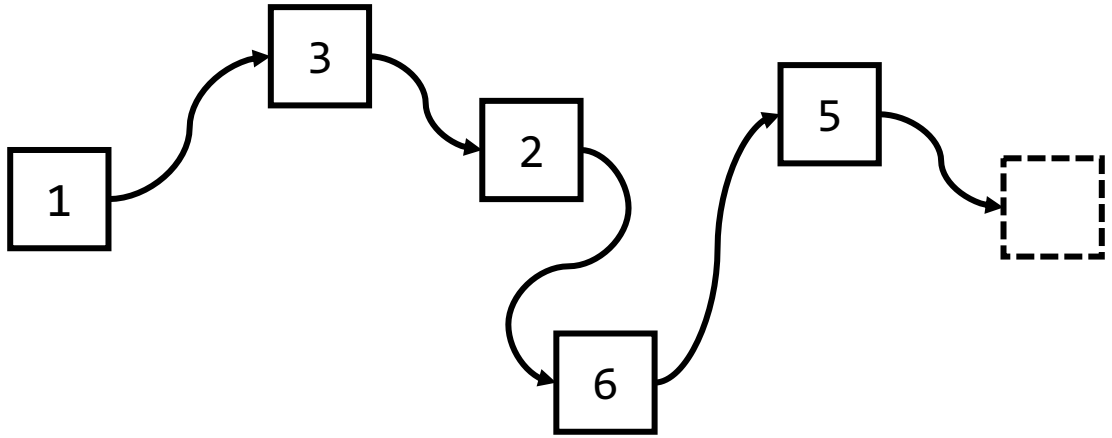
- How to **access** the i-th element?

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

Node* getNode(LinkedList *list, int index) {
    Node *node = list->head;
    while (index > 0 && node != NULL) {
        node = node->next;
        index --;
    }
    return node;
}
```



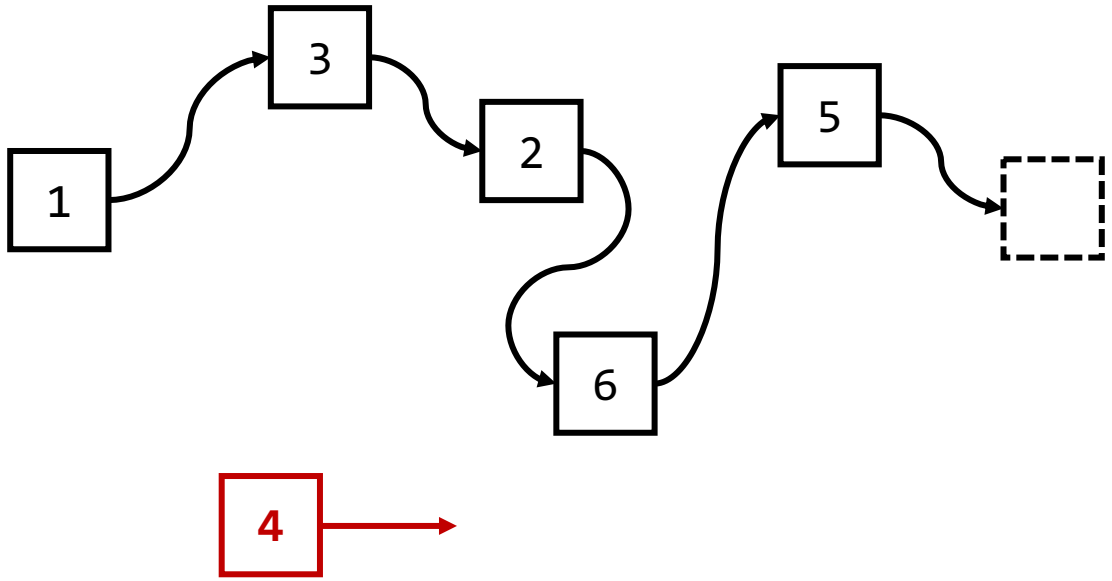
Linked Lists - Insertion

- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



Linked Lists - Insertion

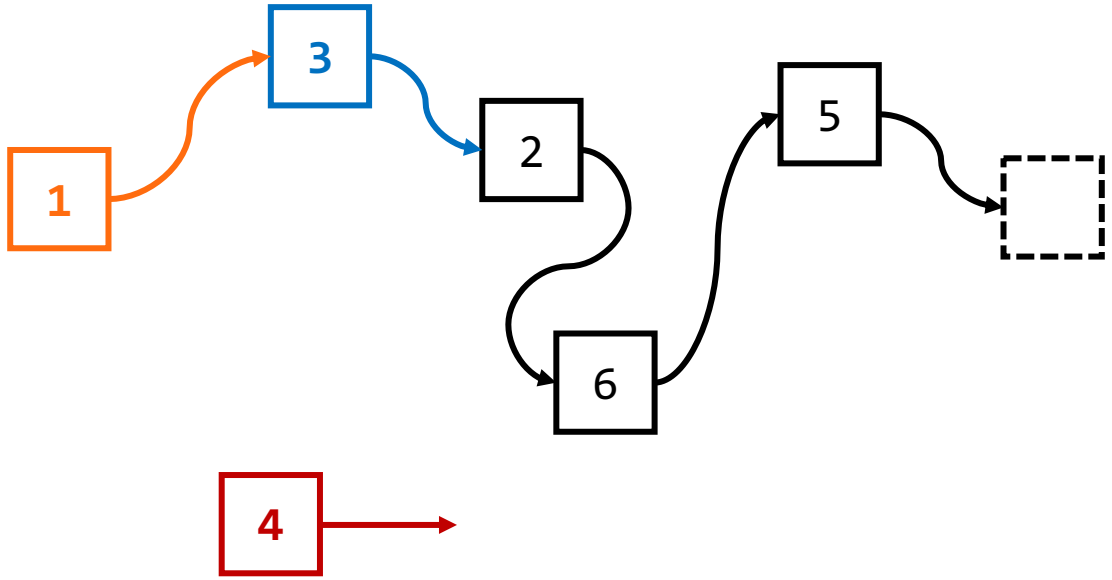
- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



1. Create new node

Linked Lists - Insertion

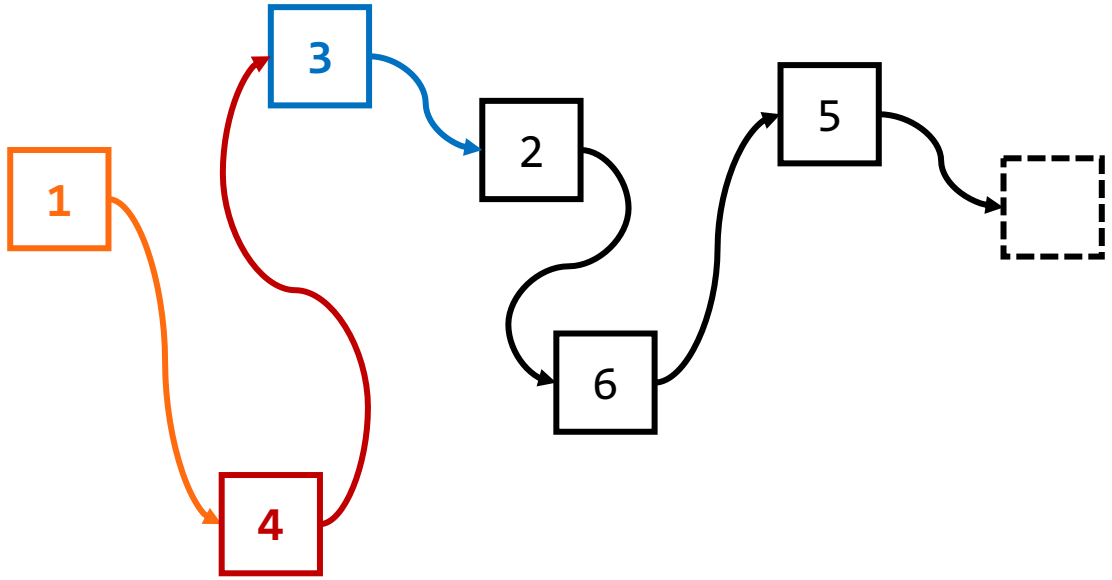
- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



1. Create new node
2. Find the 1st node and its next node

Linked Lists - Insertion

- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



1. Create new node
2. Find the 1st node and its next node
3. Connect them: node → node → node

Linked Lists - Insertion

- How to **insert** an item at the middle of the linked list?

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void insert(LinkedList *list, int item, int index) {
    Node *newNode = (Node *)malloc(sizeof(Node));    // 1.  Create new node
    newNode->value = item;

    Node *prevNode = getNode(list, index-1); // 2a. Find (index-1)-th node
    Node *nextNode = prevNode->next;         // 2b. Find index-th node

    newNode.next = nextNode; prevNode.next = newNode; // 3. Connect Nodes
}
```

1. Create new node
2. Find the 1st node and its next node
3. Connect them: node → node → node

Linked Lists - Insertion

- How to **insert** an item at the middle of the linked list?

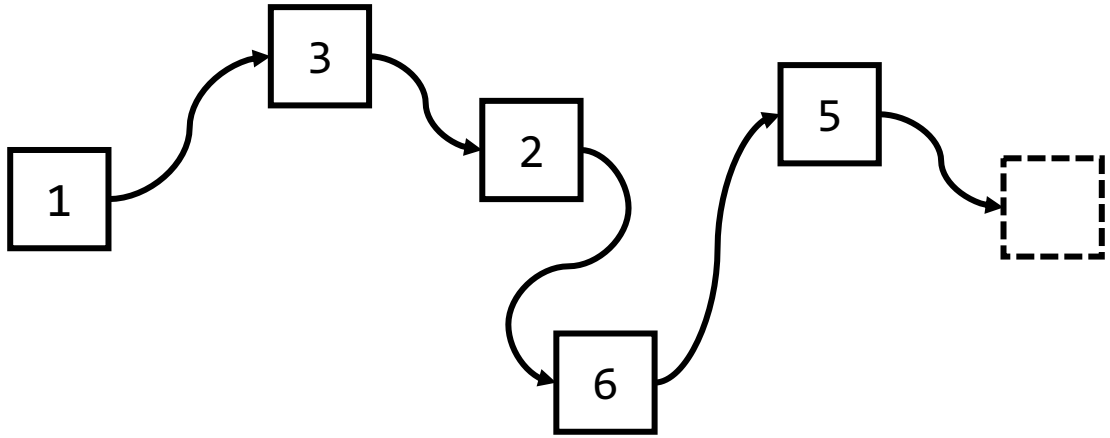
```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void insert(LinkedList *list, int item, int index) {
    Node *newNode = (Node *)malloc(sizeof(Node));    // 1.  Create new node
    newNode->value = item;

    Node *prevNode = getNode(list, index-1); // 2a. Find (index-1)-th node
    Node *nextNode = prevNode->next;         // 2b. Find index-th node

    newNode.next = nextNode; prevNode.next = newNode; // 3. Connect Nodes
}
```

(Q) Is this well-implemented?

- When is `list->head` changed?
- What happens if `getNode()` returns `NULL`?

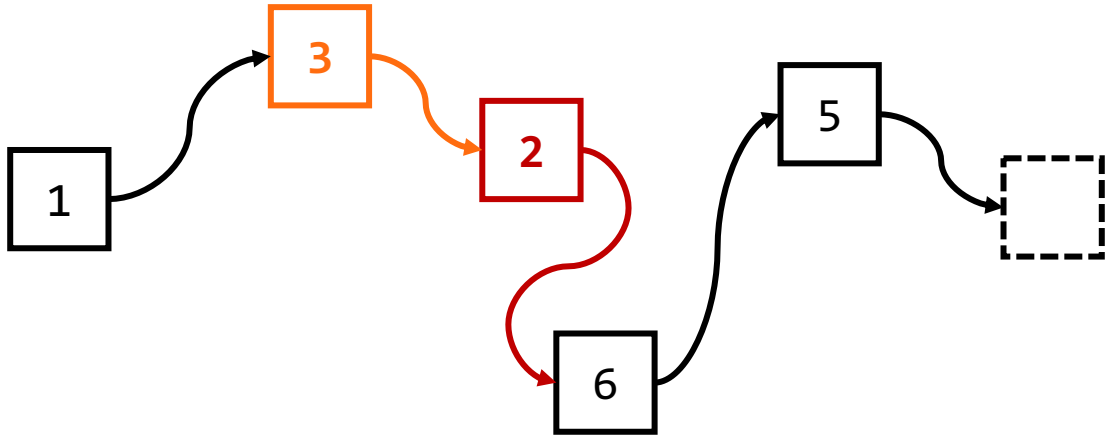
Linked Lists - Deletion

- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



Linked Lists - Deletion

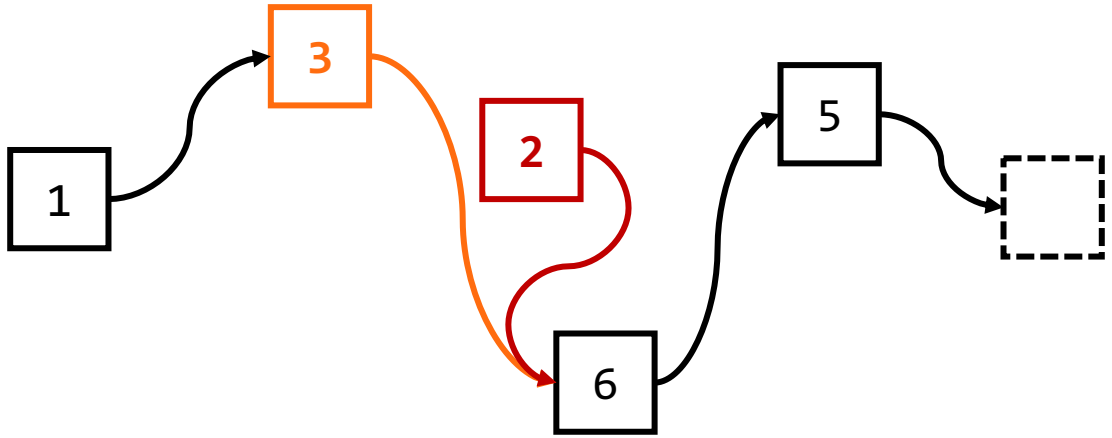
- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



1. Find the 2nd node and its next node

Linked Lists - Deletion

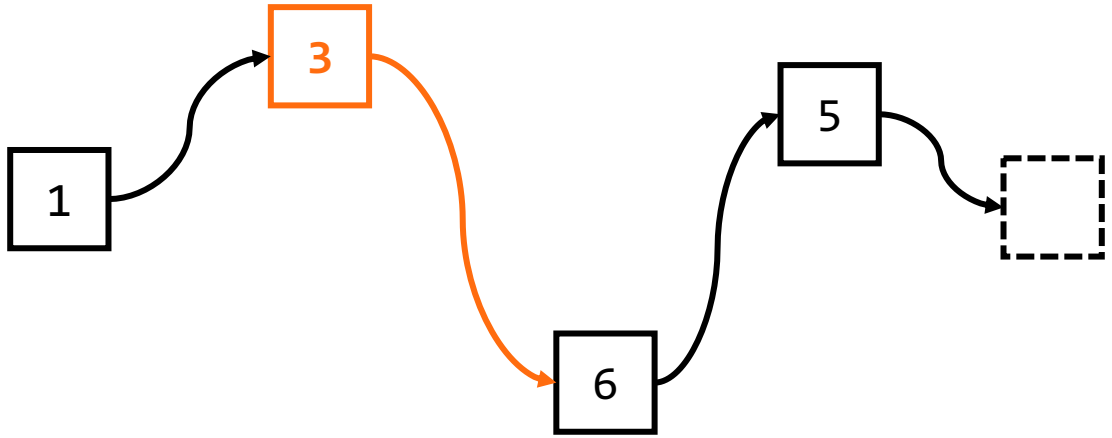
- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



1. Find the 2nd node and its next node
2. Skip the connection between node and node

Linked Lists - Deletion

- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



1. Find the 2nd node and its next node
2. Skip the connection between node and node
3. Free the node’s memory

Linked Lists - Deletion (Practice)

- How to **delete** an item from the linked list?

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void delete(LinkedList *list, int index) {



                    ?



}
```

  - Use `free()` for releasing the deleted node
  - Consider corner cases
    - When is `list->head` changed? What happens if `getNode()` returns `NULL`?

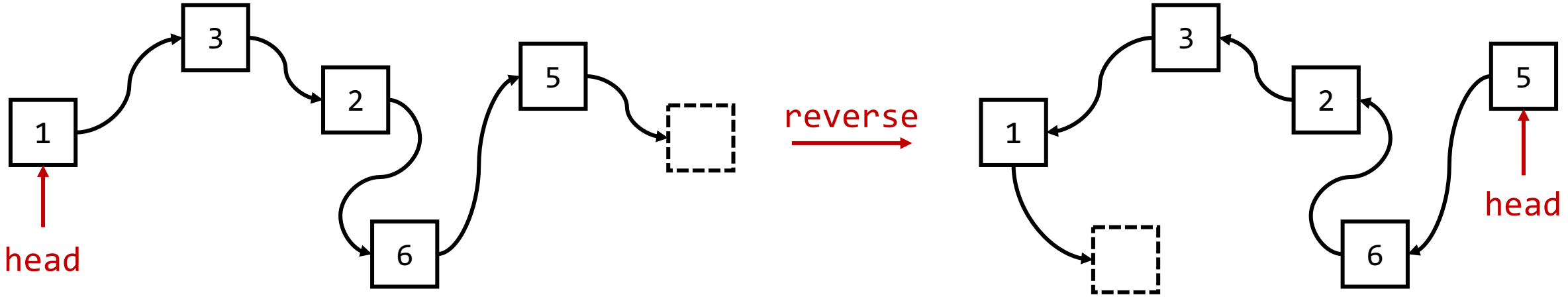
Linked Lists - Deletion (Practice)

- How to **delete all items** from the linked list?

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void deleteAll(LinkedList *list) {

                    ?

}
```

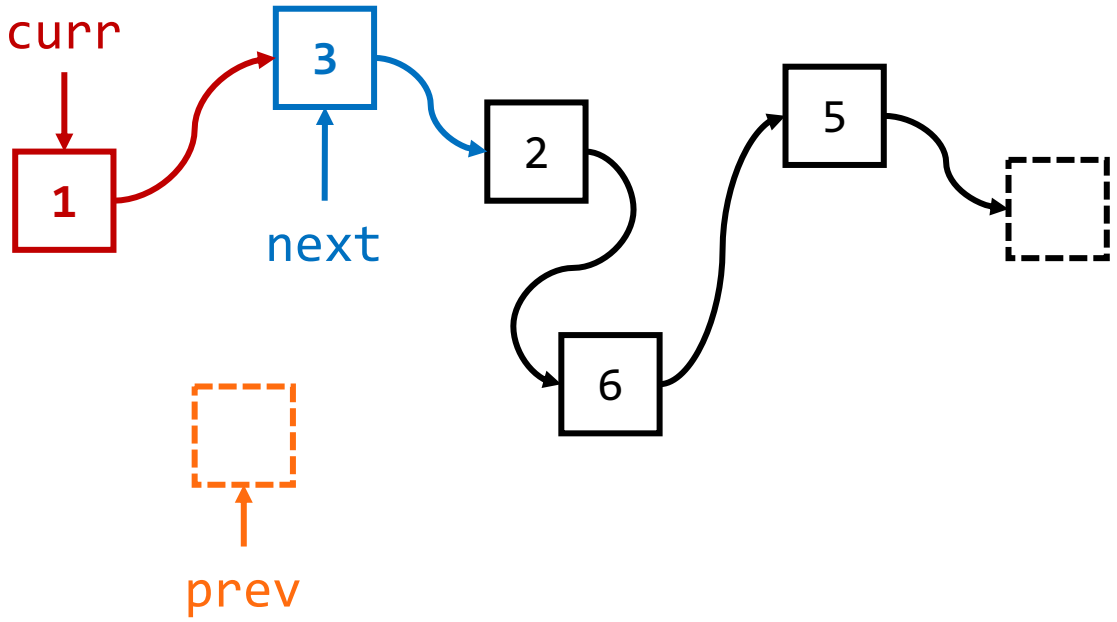
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



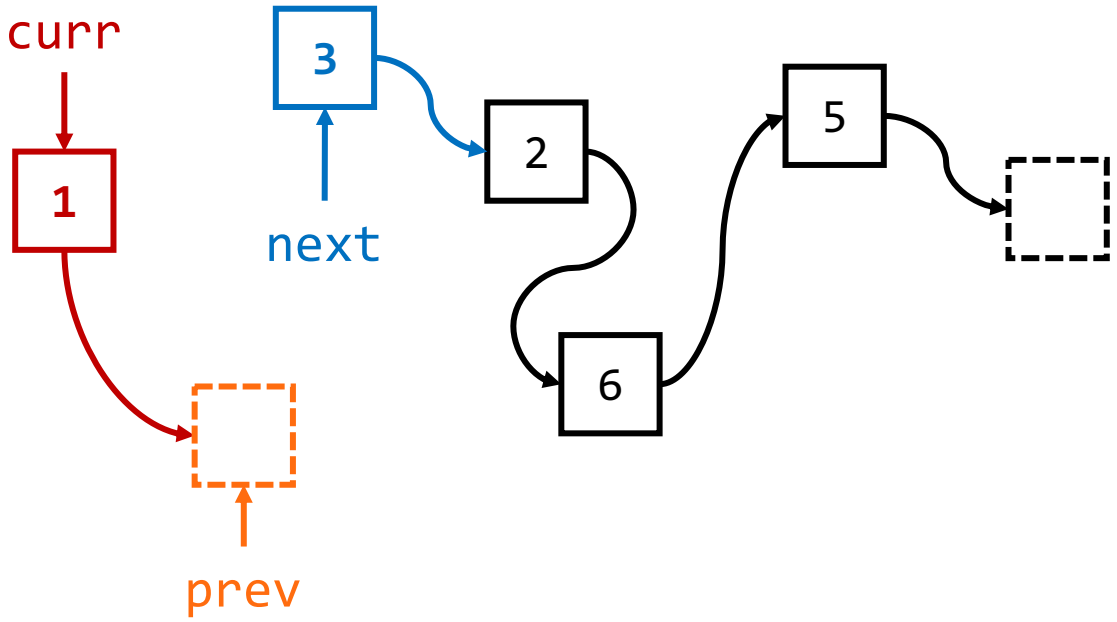
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



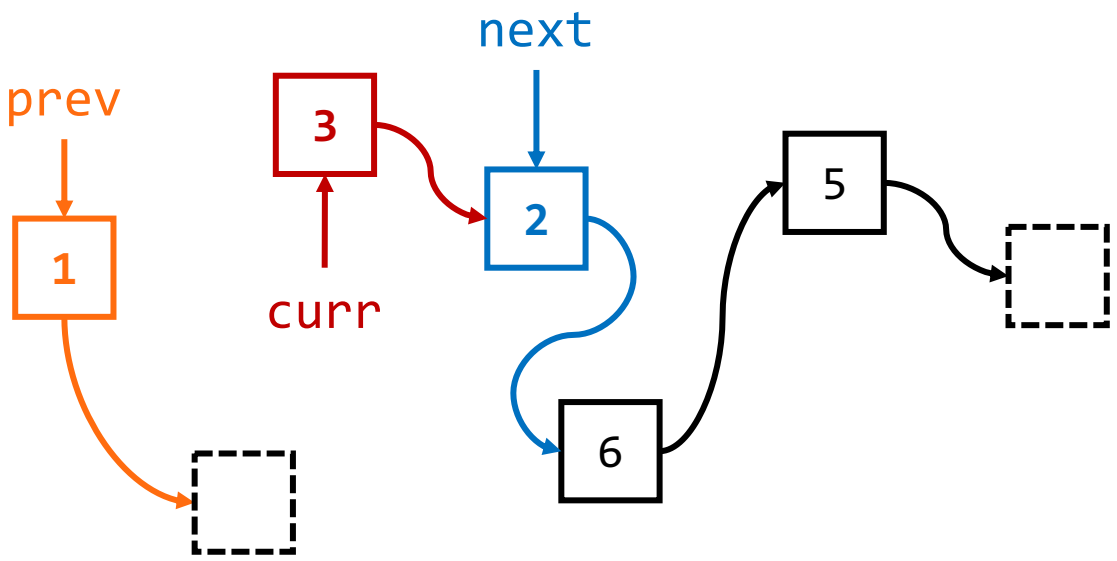
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



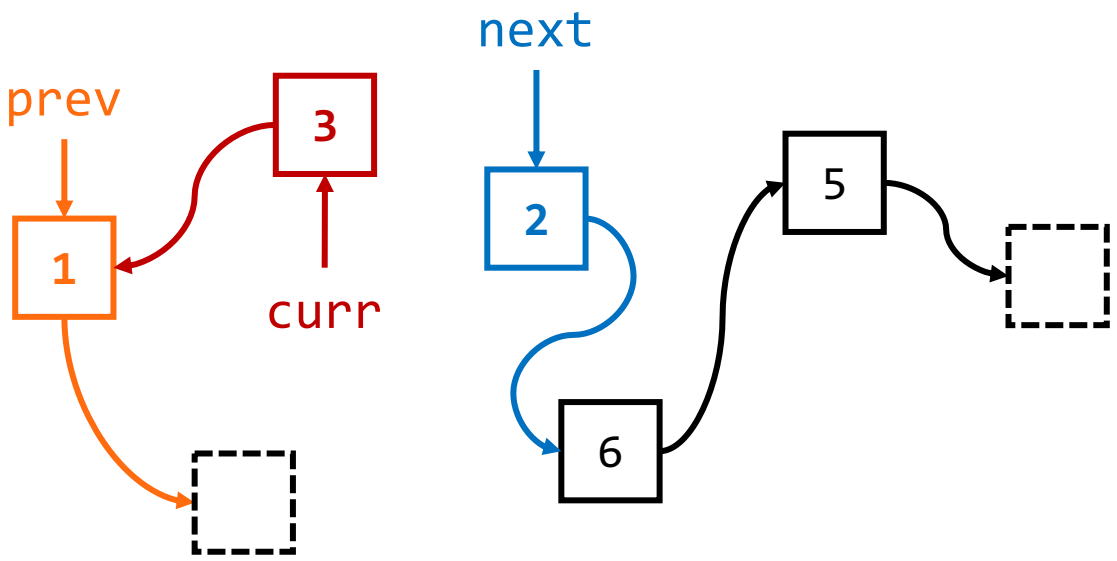
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



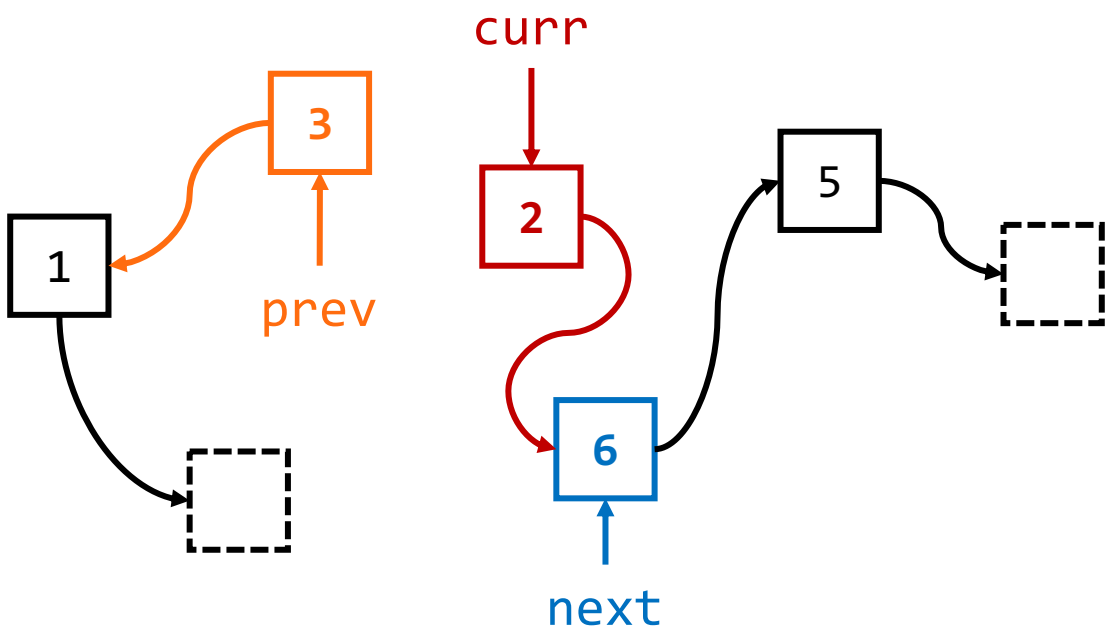
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



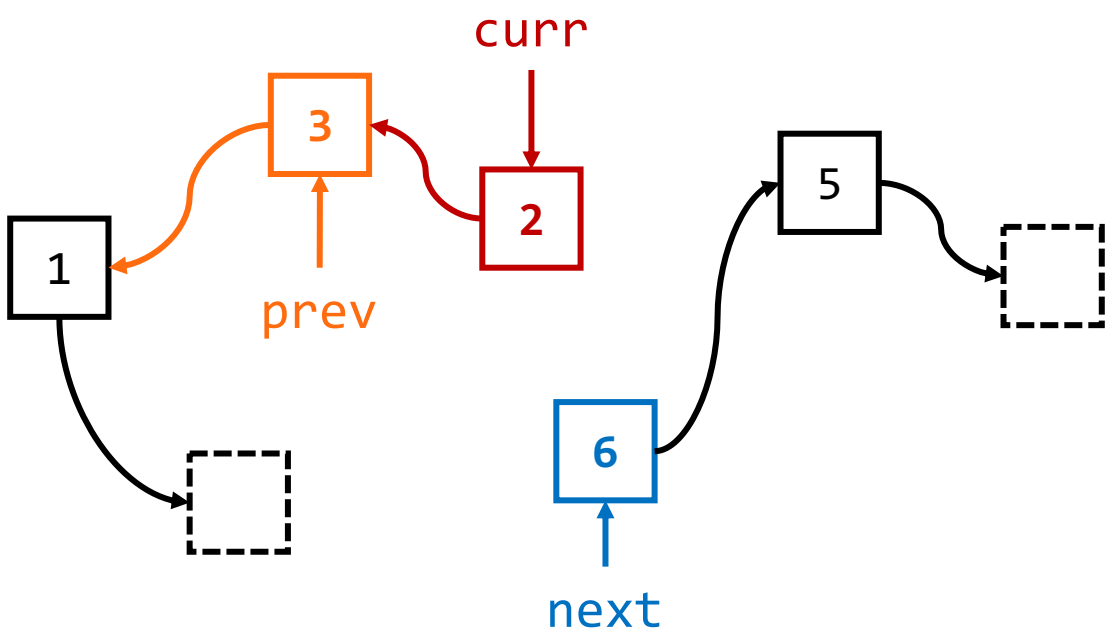
Linked Lists - Reversion

- How to **reverse the order** of the linked list?



Linked Lists - Reversion

- How to **reverse the order** of the linked list?



Linked Lists - Reversion

- How to **reverse the order** of the linked list?

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void reverse(LinkedList *list) {
```

?

```
}
```

Linked Lists - Time Complexity

- Array *vs.* Linked List
  - Array Insertion & Deletion
    - $O(1)$ - Find the target element
    - $O(n)$ - Insert or delete the element
  - Linked List Insertion & Deletion
    - $O(n)$ - Find the target element
    - $O(1)$ - Insert or delete the element

Operation	Array	Linked List
Insertion	$O(n)$	$O(1)$ or $O(n)$
Deletion	$O(n)$	$O(1)$ or $O(n)$
Search by Index (Access)	$O(1)$	$O(n)$
Search by Value	$O(n)$	$O(n)$
Reversion	$O(n)$	$O(n)$

Any Questions?