Chapter 12

Lists, Stacks and Queues

Data Structures

A **data structure** is a particular way of organizing data in a computer so that it can be used effectively throughout our programs.

- Built in arrays
- STL Vectors
- STL Arrays
- more…

Data Structures

3 New Data Structures!

- Linked Lists
- Stacks
- Queues

List ADT

A **List** is an abstract data structure which stores a sequence of items or data and supports a number of different common operations

A user need not have knowledge of the internal implementation of the list ADT.

List (ADT) - Operations

Table 12.1.1: Some common operations for a list ADT.

Operation	Description	Example starting with list: 99, 77
Append(list, x)	Inserts x at end of list	Append(list, 44), list: 99, 77, 44
Prepend(list, x)	Inserts x at start of list	Prepend(list, 44), list: 44, 99, 77
InsertAfter(list, w, x)	Inserts x after w	InsertAfter(list, 99, 44), list: 99, 44, 77
Remove(list, x)	Removes x	Remove(list, 77), list: 99
Search(list, x)	Returns item if found, else returns null	Search(list, 99), returns item 99 Search(list, 22), returns null
Print(list)	Prints list's items in order	Print(list) outputs: 99, 77
PrintReverse(list)	Prints list's items in reverse order	PrintReverse(list) outputs: 77, 99
Sort(list)	Sorts the lists items in ascending order	list becomes: 77, 99
IsEmpty(list)	Returns true if list has no items	For list 99, 77, IsEmpty(list) returns false
GetLength(list)	Returns the number of items in the list	GetLength(list) returns 2

example: vector.cpp

Singly Linked List

A **singly-linked list** is a data structure made up of ***Nodes,*** where each node holds a piece of data, and points to the next node in the sequence.

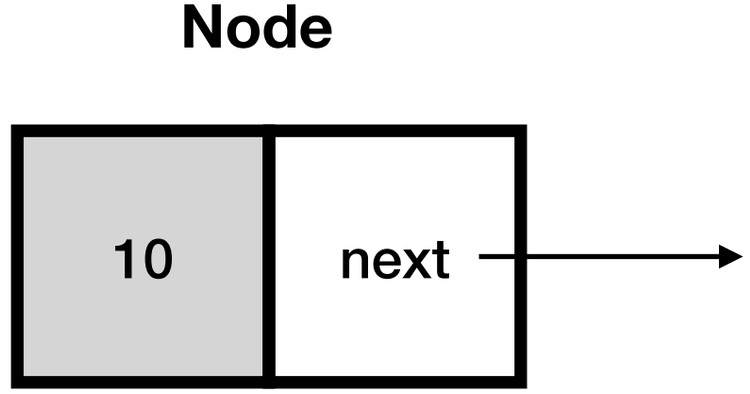
A Linked List utilizes *pointers* to connect or link the data together in one list of information

In a **singly**-linked list, the nodes of the list only point to the next node in the list.

Singly Linked List Node

A **Node** is an object that stores a data type and contains a pointer that holds the memory address of the next node in the sequence, (or nullptr, if it is the last node in the sequence).

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

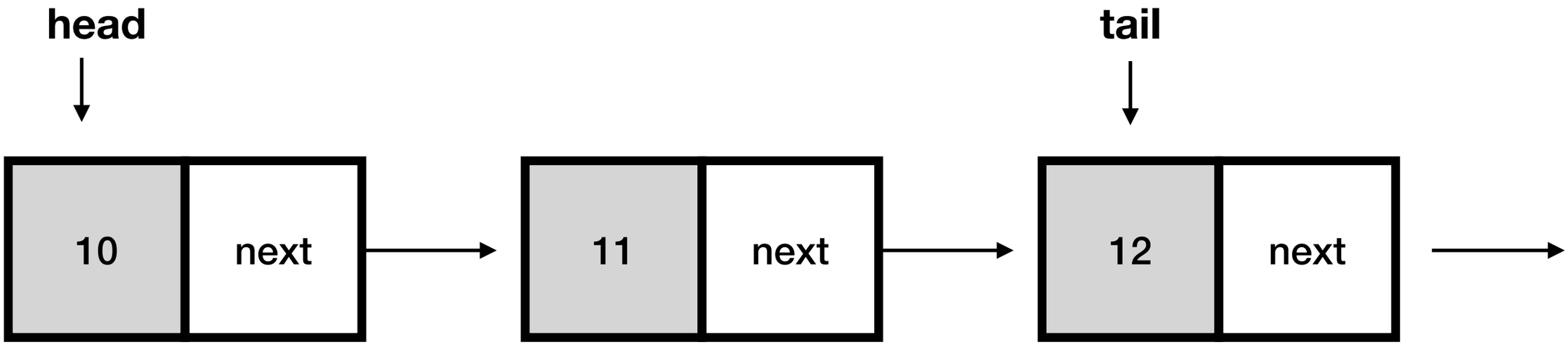


example: SinglyLinkedList/Node.h

Singly Linked List

Linked List is made up of multiple Nodes

- **head** - a pointer to the first Node in the linked list
- **tail -** a pointer to the last Node in the linked list



example: SinglyLinkedList/LinkedList.h

Singly Linked List - insert

To insert into a linked list we must consider the following scenarios.

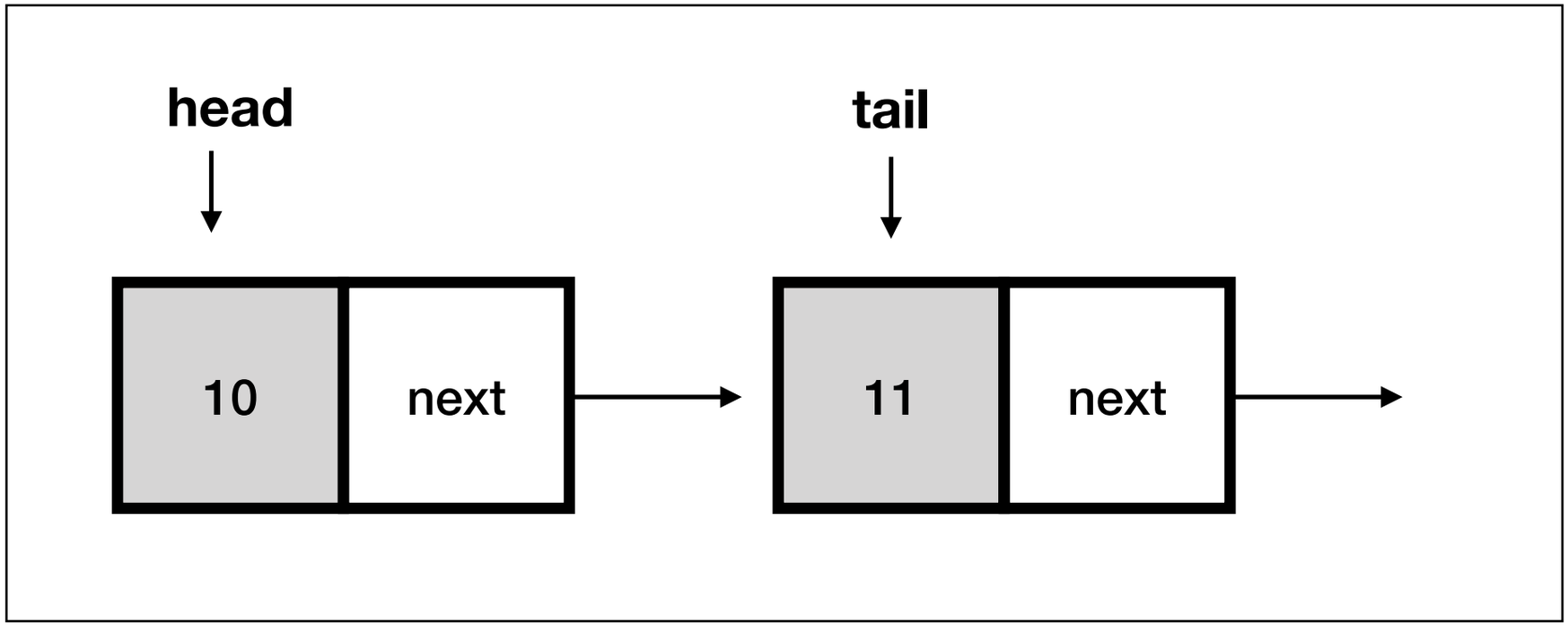
- Insert at the beginning of the list
- Insert at the end of the list
- Inserting in the middle of the list

Singly Linked List - insert

To insert into a linked list we must consider the following scenarios.

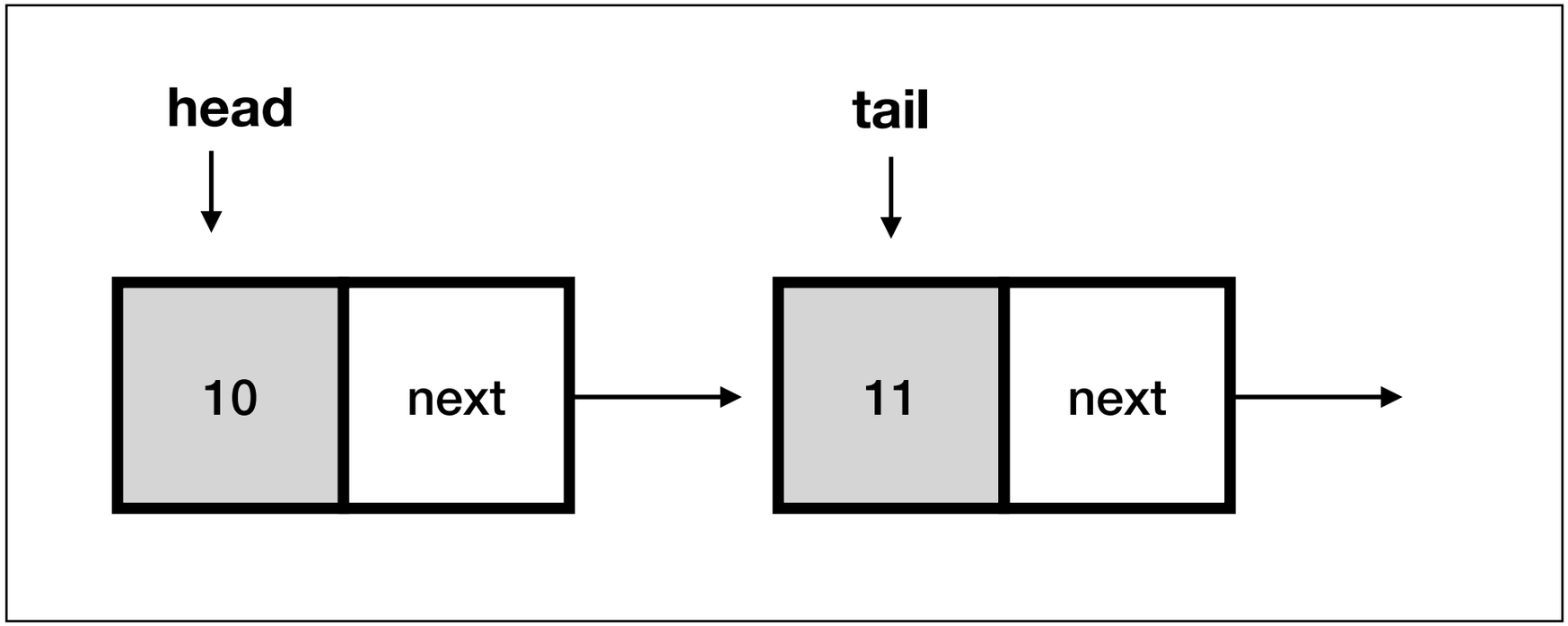
- Insert at the beginning of the list
- **Insert at the end of the list**
- Inserting in the middle of the list

Singly Linked List - insert

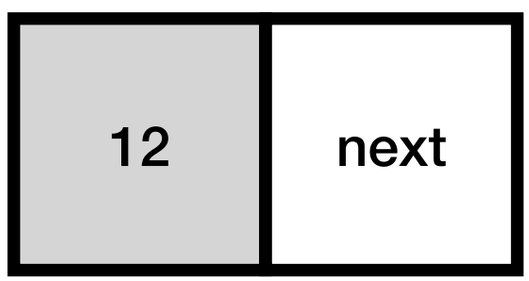


New Value **12**

Singly Linked List - insert

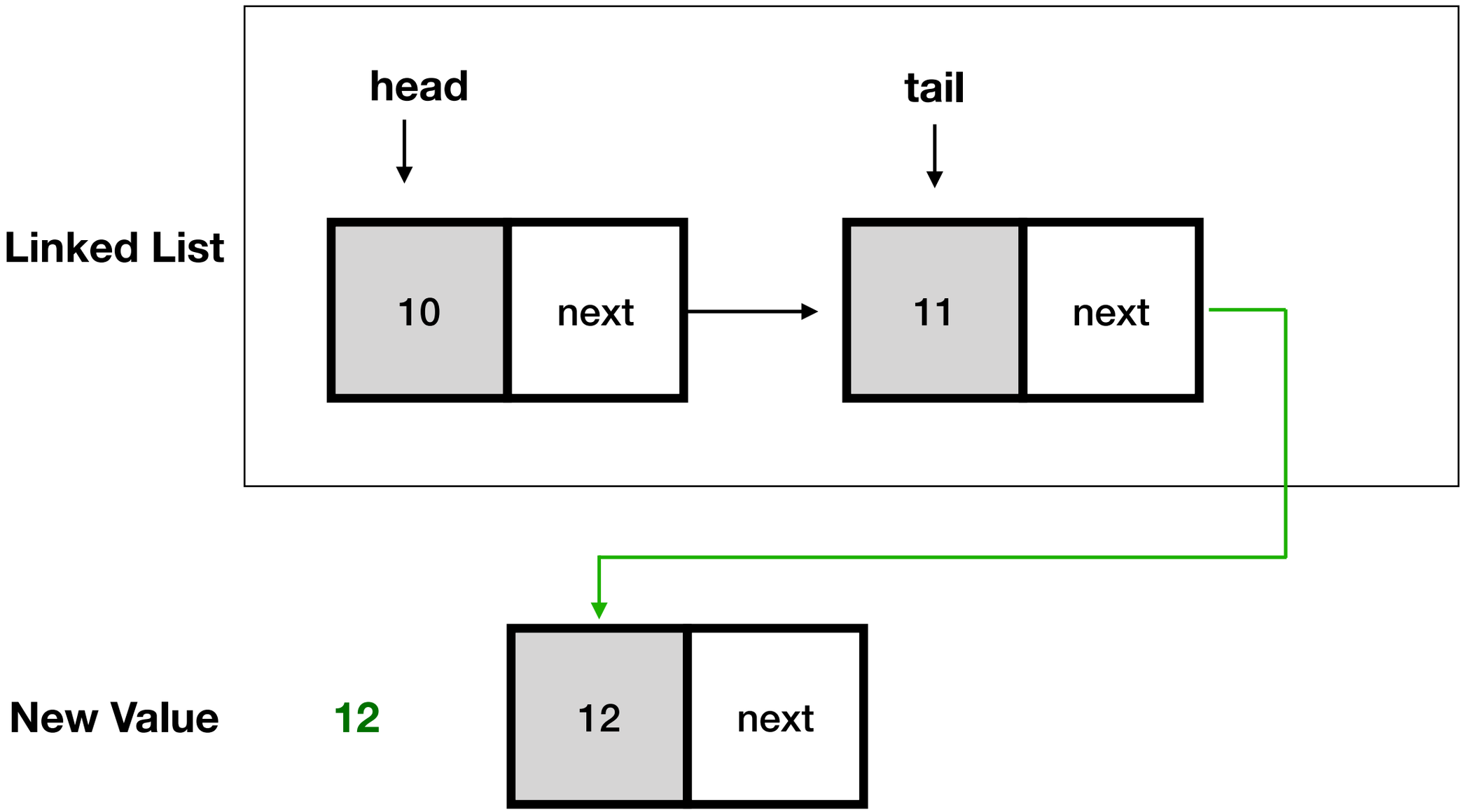


New Value **12**



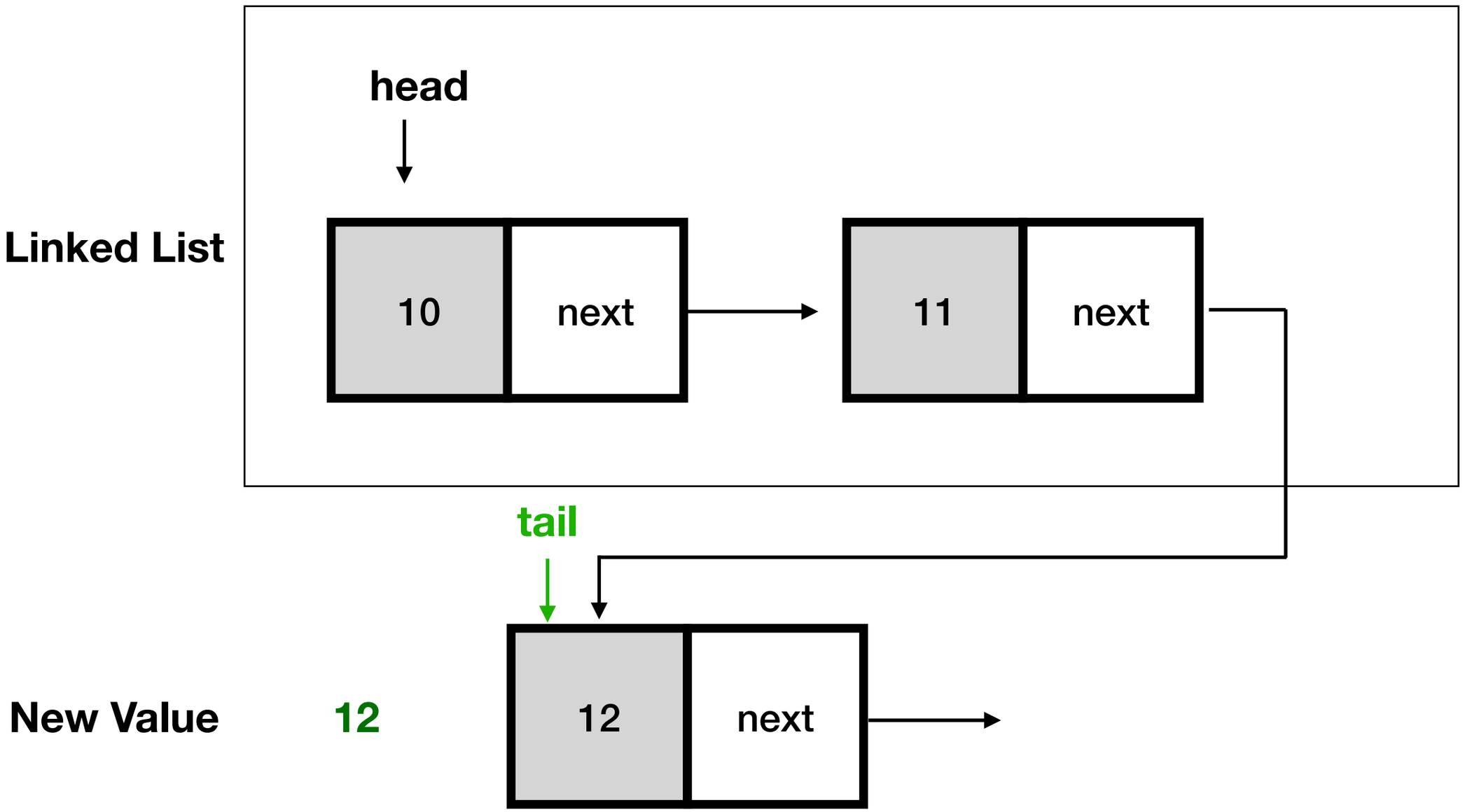
Step 1: Create a Node to store the new value

Singly Linked List - insert



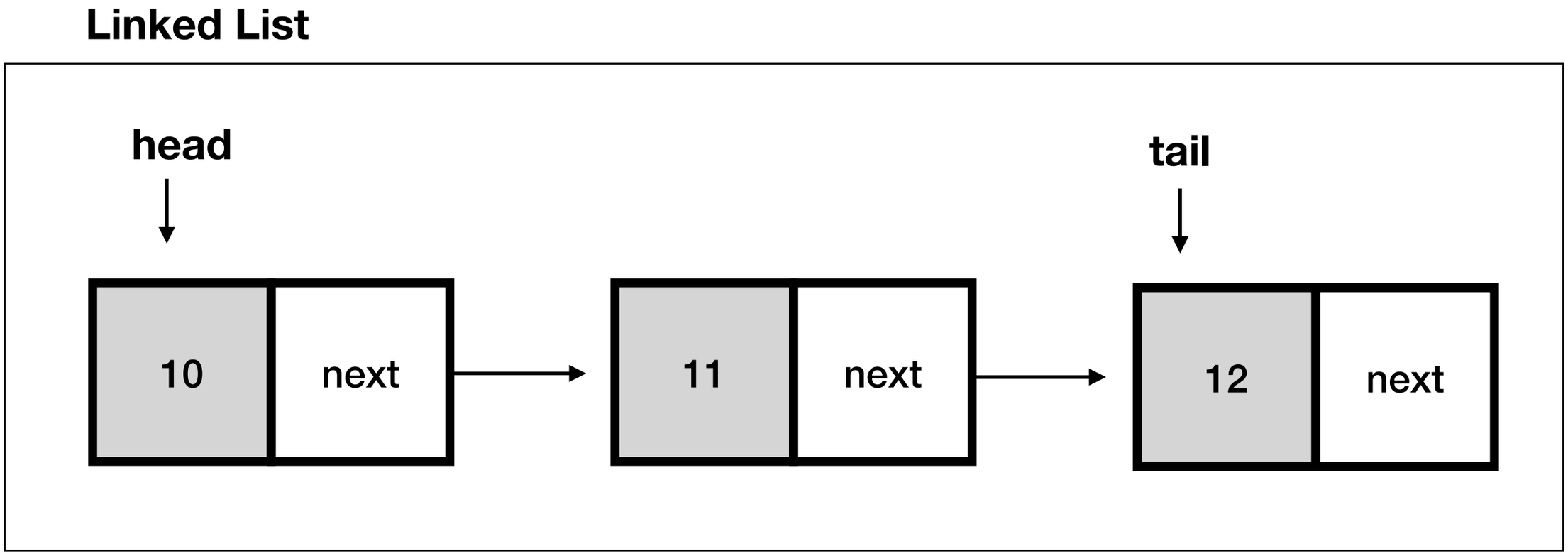
Step 2: Update *tail's* next pointer

Singly Linked List - insert



Step 3: Update *tail pointer*

Singly Linked List - insert



Singly Linked List - append

For this method we will insert a node at the end of our list

```
void LinkedList::pushBack(int data) {
  Node *node = new Node();
  node->data = data;
  node->next = nullptr;

  if (this->head == nullptr) {
    this->head = node;
    this->tail = node;
  } else {
    this->tail->next = node;
    this->tail = node;
  }

  this->length++;
}
```

example: SinglyLinkedList/LinkedList.cpp

Singly Linked List - remove

We can implement a remove method to remove a node from our link list from the following places:

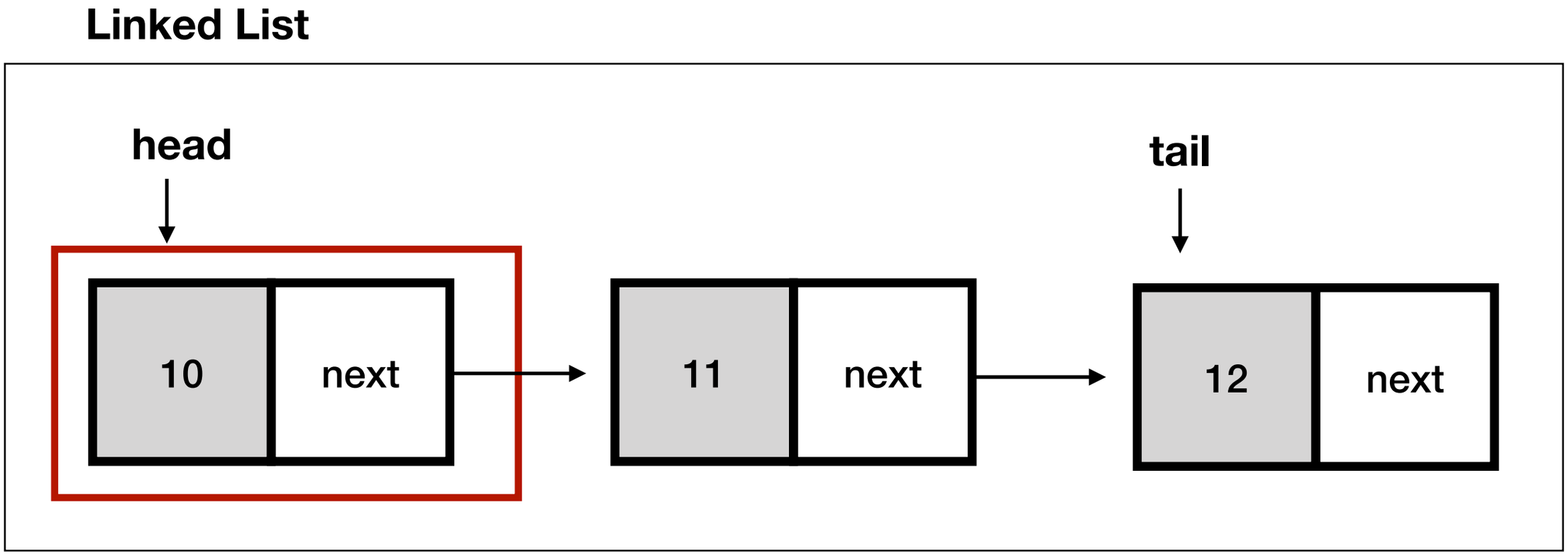
- Remove at beginning of list
- Remove at end of list
- Remove at index

Singly Linked List - remove

We can implement a remove method to remove a node from our link list from the following places:

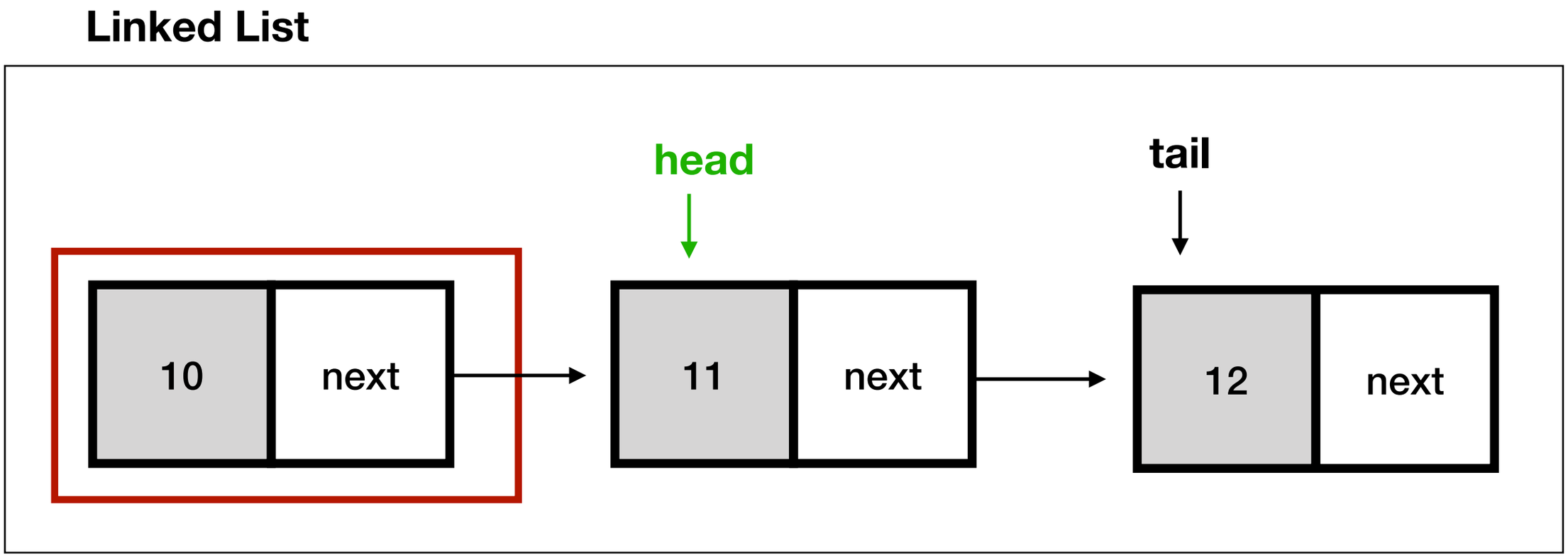
- **Remove at beginning of list**
- Remove at end of list
- Remove at index

Singly Linked List - remove



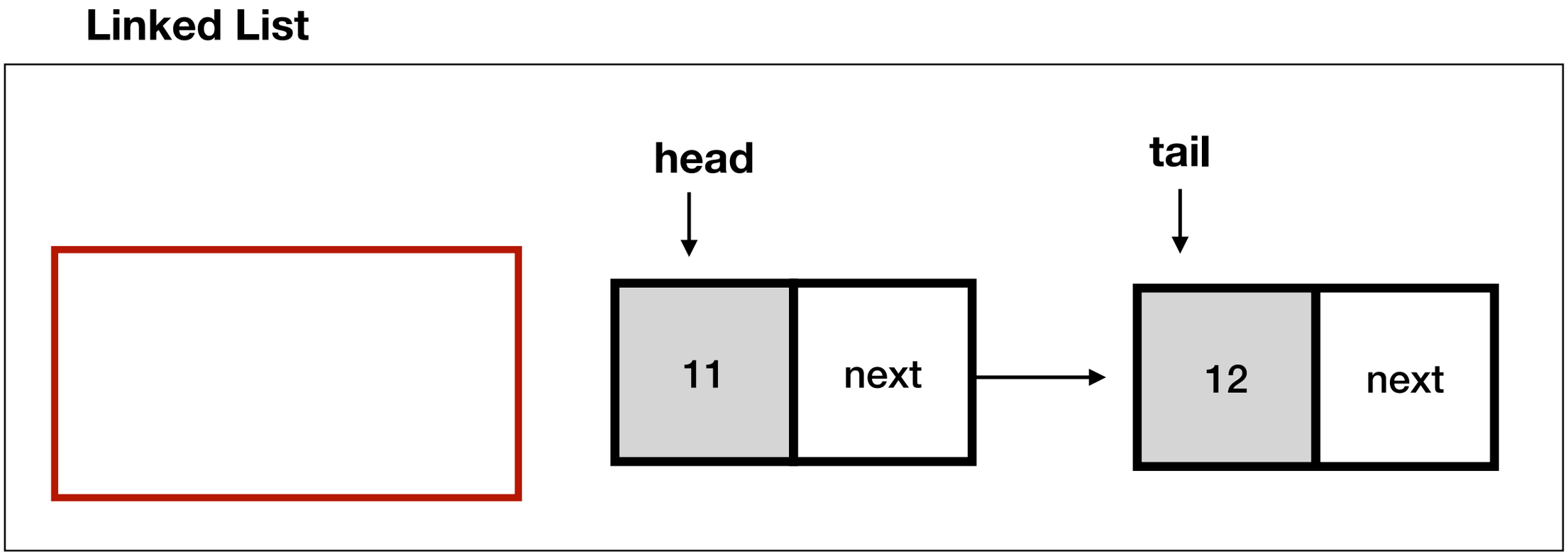
Step 1: Select first node in list

Singly Linked List - remove



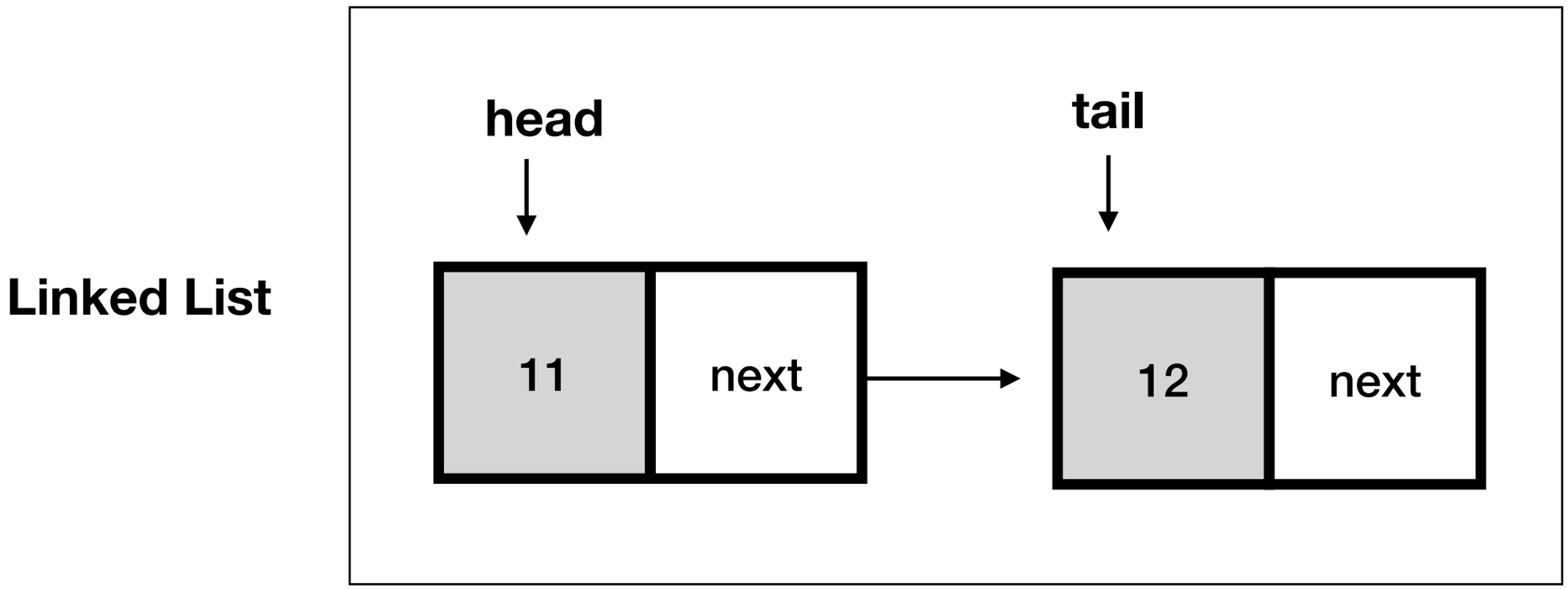
Step 2: Update *head* Pointer

Singly Linked List - remove



Step 3: Delete the node off the heap (use *delete* keyword)

Singly Linked List - remove



Traversing a linked list

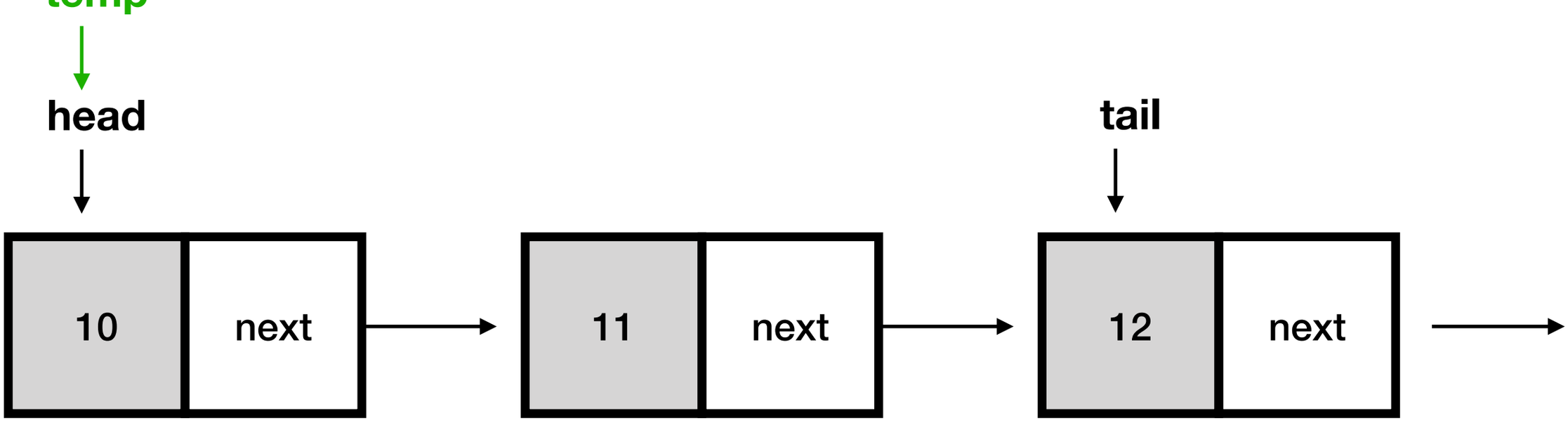
To traverse a linked list we use the pointers to access the next element or item in the list.

Create a *current* pointer that begins at the start of the list (or *head*).

DO NOT use the linked list pointers *(head / tail)* for keeping track of the traversal (we do not want to manipulate the linked list head or tail unless inserting or deleting.

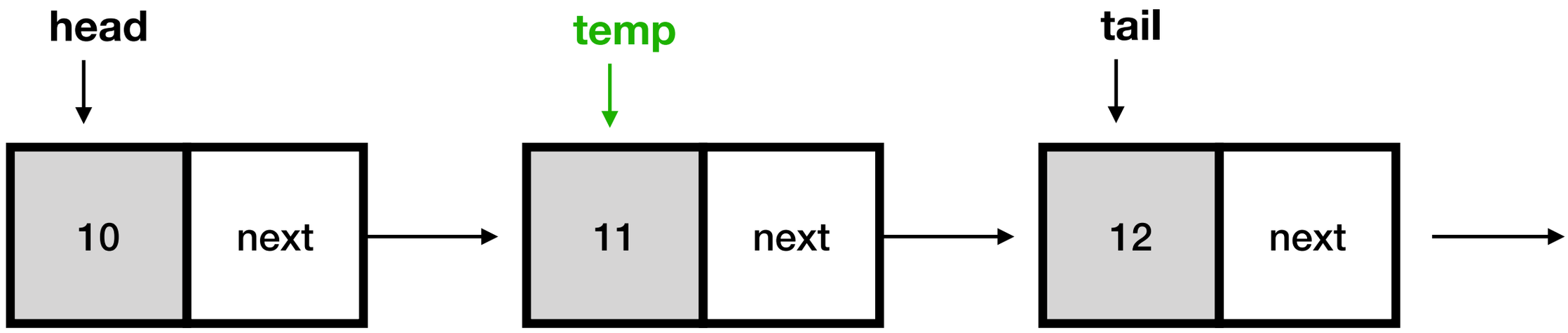
Traversing a linked list

```
void LinkedList::print(){
    Node* temp = this->head;
    while(temp){
          std::cout << temp->data << std::endl;
          temp = temp->next;
    }
}
```



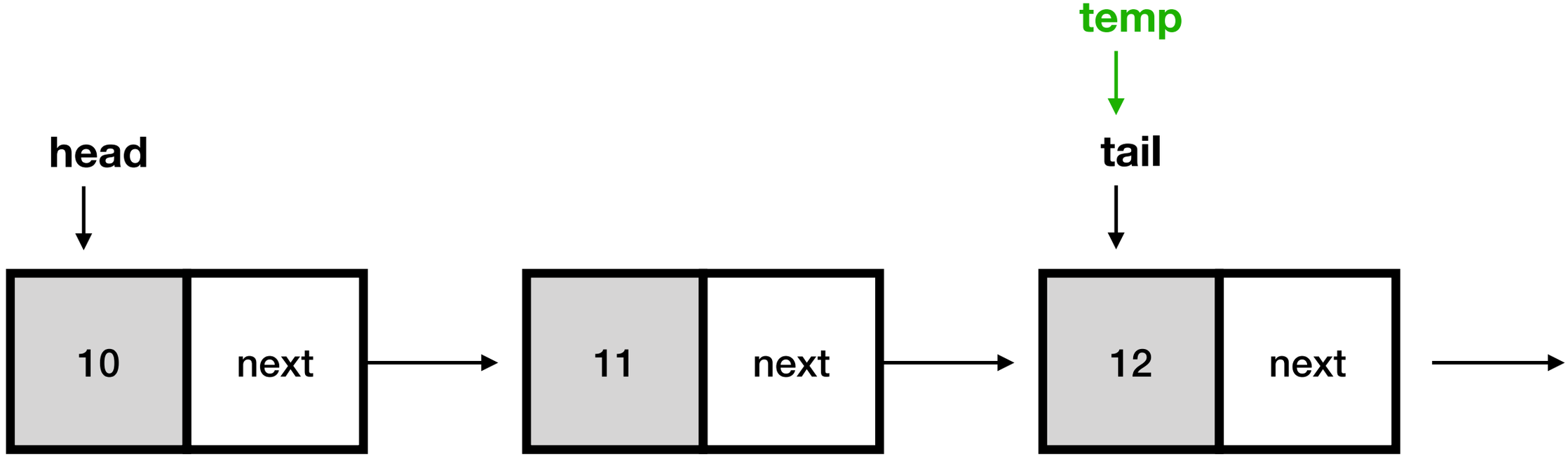
Traversing a linked list

```
void LinkedList::print(){
    Node* temp = this->head;
    while(temp){
          std::cout << temp->data << std::endl;
          temp = temp->next;
    }
}
```



Traversing a linked list

```
void LinkedList::print(){
    Node* temp = this->head;
    while(temp){
          std::cout << temp->data << std::endl;
          temp = temp->next;
    }
}
```



Searching Linked Lists

Searching for an item in a list requires looking for a key. Depending on how we want to search we can either:

1. Return the node where the data was found

2. Return the “index” where the data was found

3. Return some other data about that node

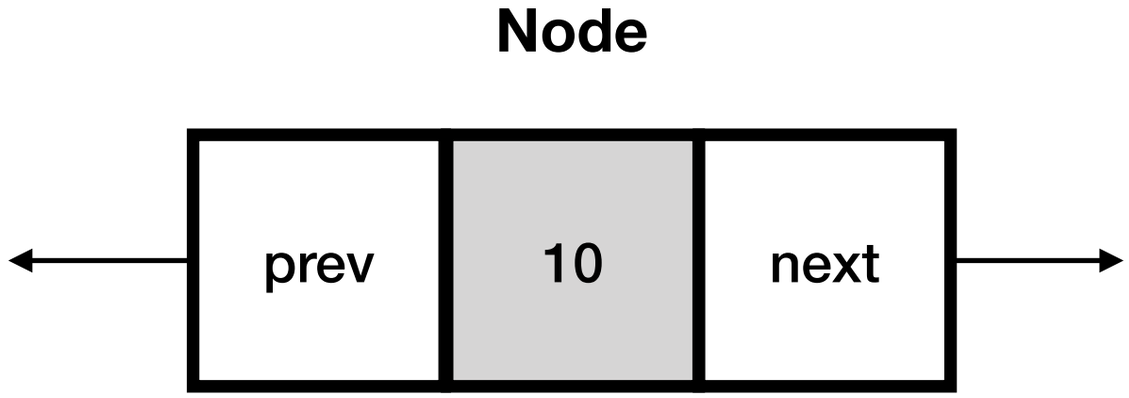
NOTE: most of the time a linked list will actually store an *object.* So we can search for an item in a list that meets some sort of criteria then return the entire object.

Let’s implement option 1.

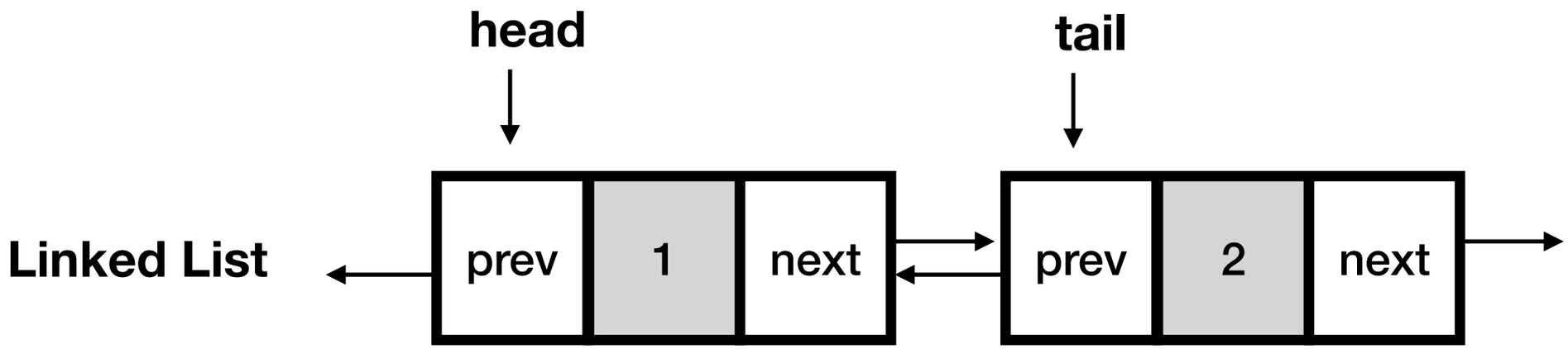
Doubly Linked List

A **doubly-linked list** is a data structure made up of *Nodes,* where each node holds a piece of data, and points to the next node in the sequence ***AND*** the previous node in the sequence.

```
struct Node {
   int data;
   Node* prev;
   Node* next;
};
```



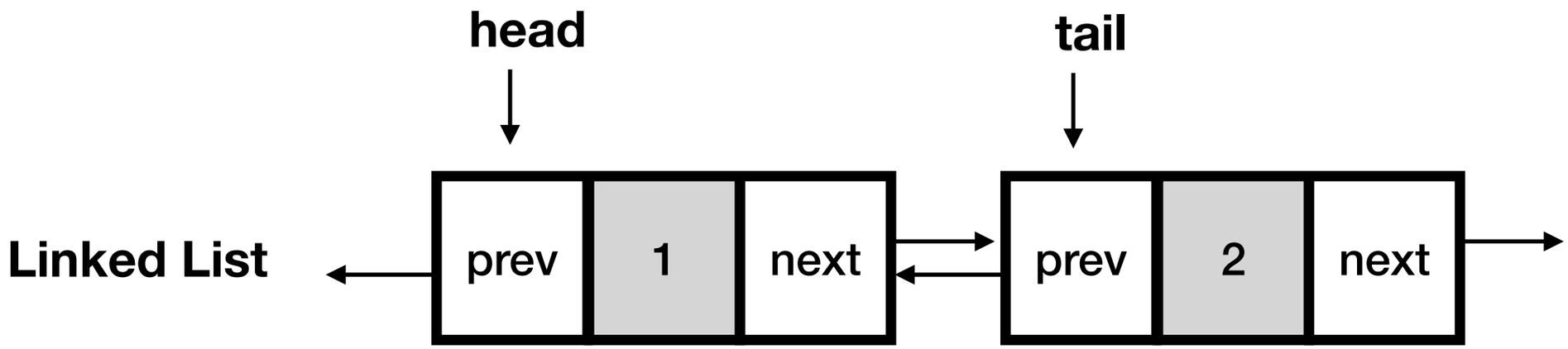
Doubly Linked List - *insert()*

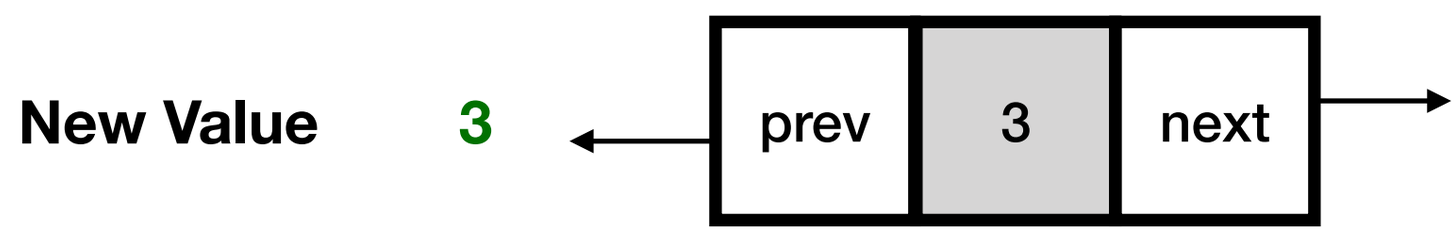


New Value **3**

```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

Doubly Linked List - *insert()*

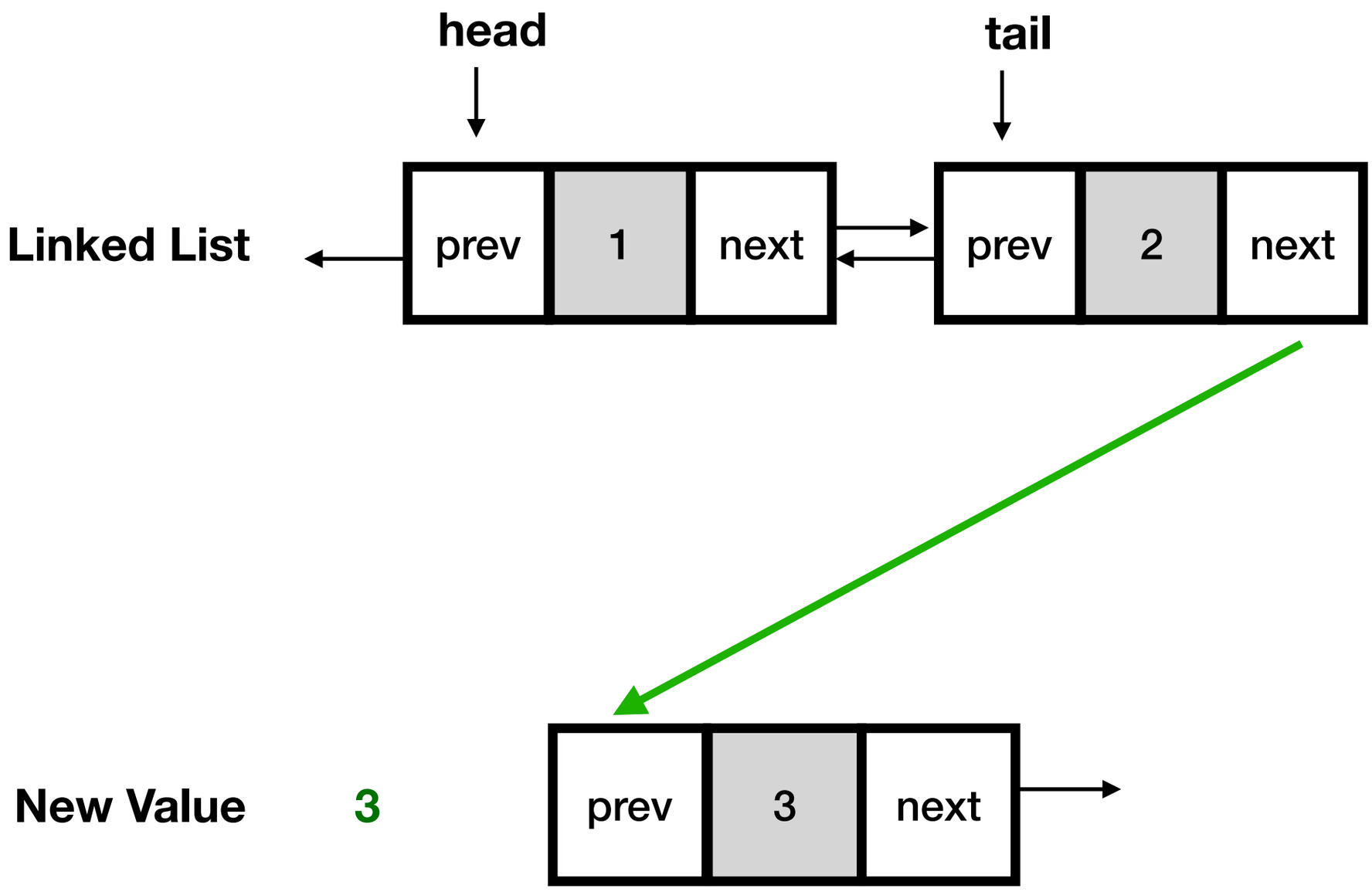





```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

Step 1: Create Doubly Linked List Node

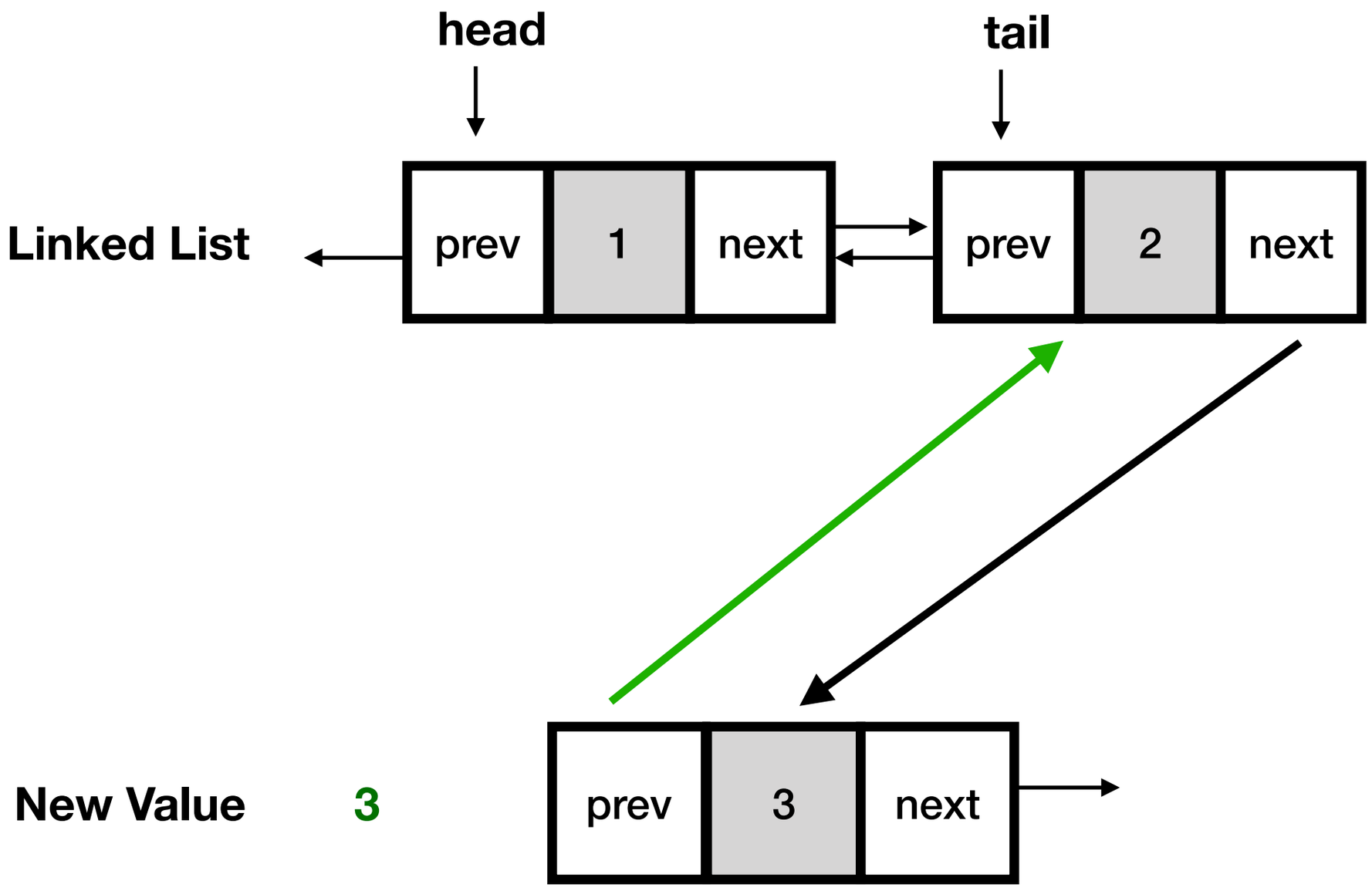
Doubly Linked List - *insert()*



Step 2: Update *tail* next pointer

```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

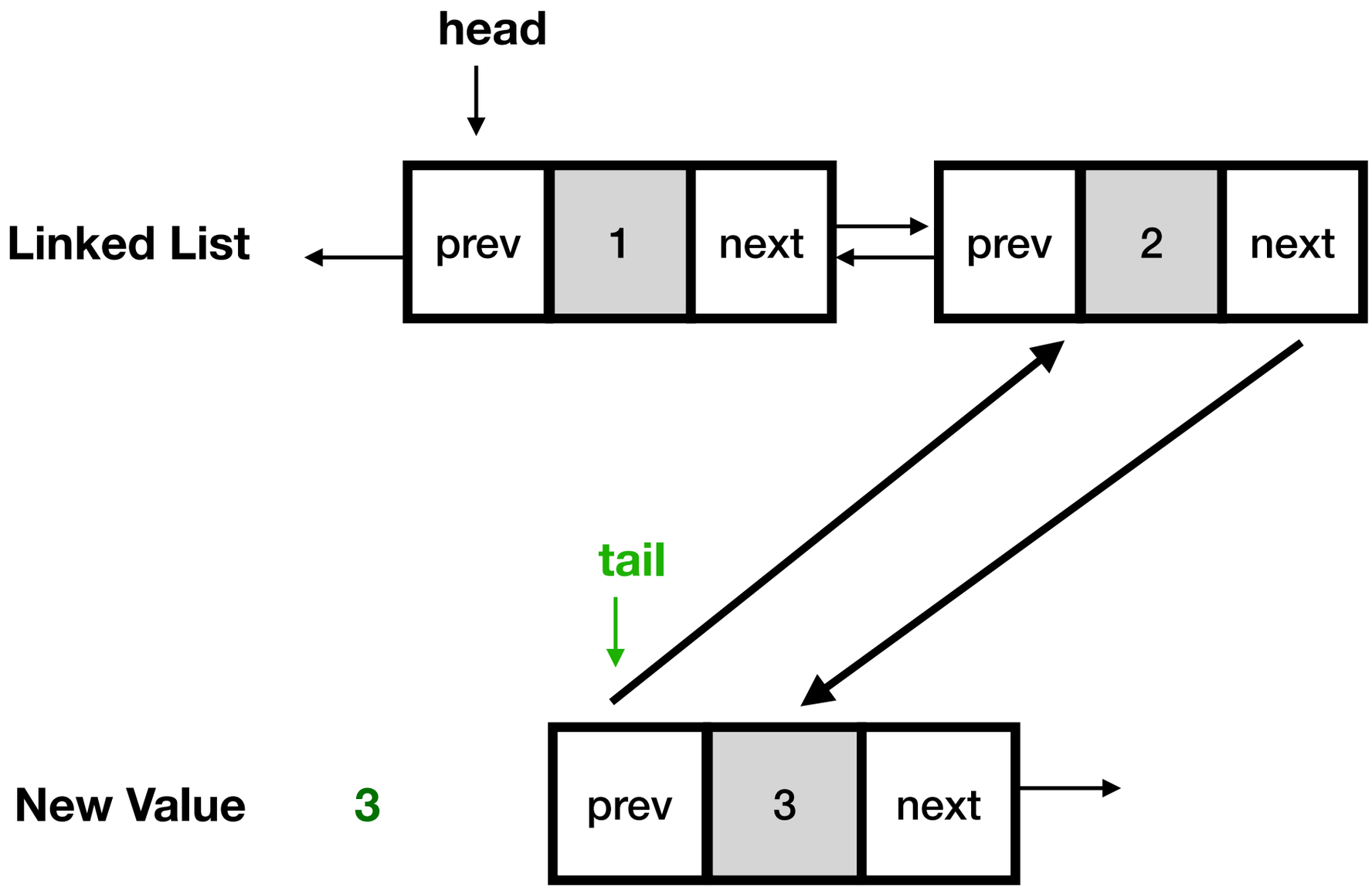
Doubly Linked List - *insert()*



Step 3: Update *new nodes* **prev** pointer

```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

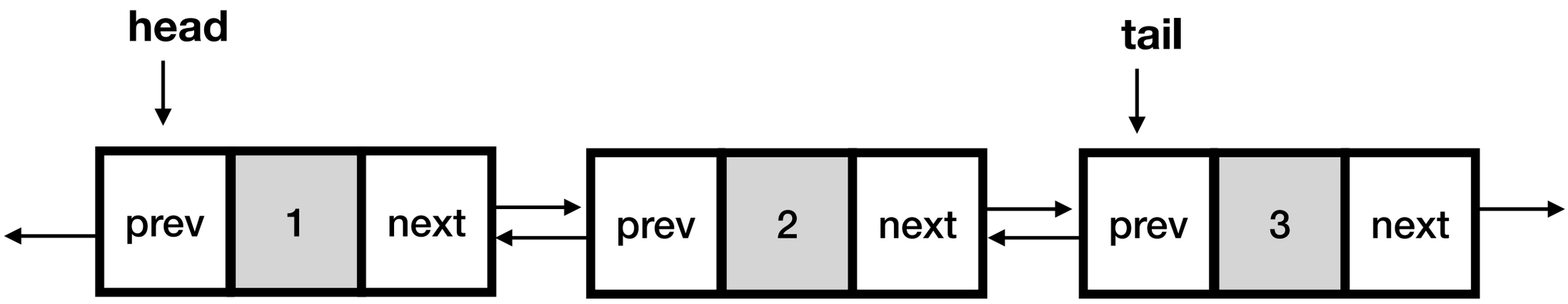
Doubly Linked List - *insert()*



Step 4: Update *tail* pointer

```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

Doubly Linked List - *insert()*




```
struct Node {
  int data;
  Node* prev;
  Node* next;
};
```

Doubly Linked List - append()

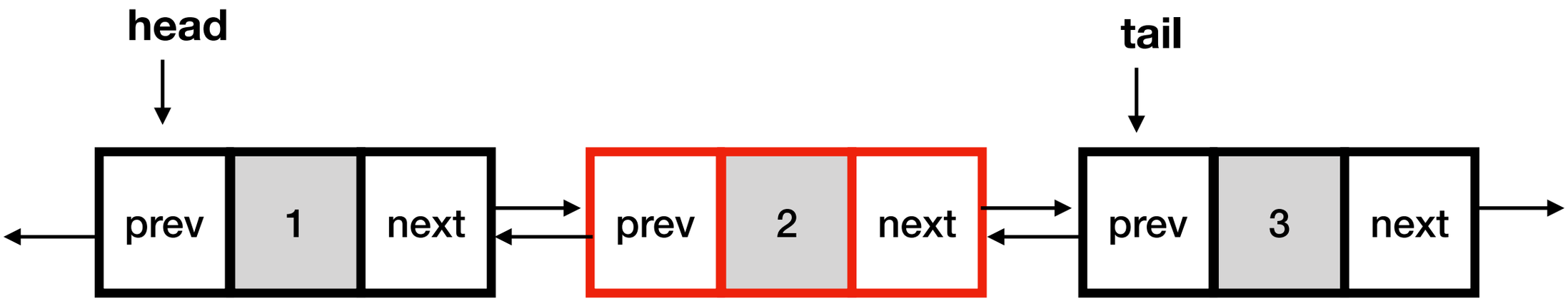
We can implement the append function in the same way as a Singly Linked List, however we also need to account for the *prev* pointers.

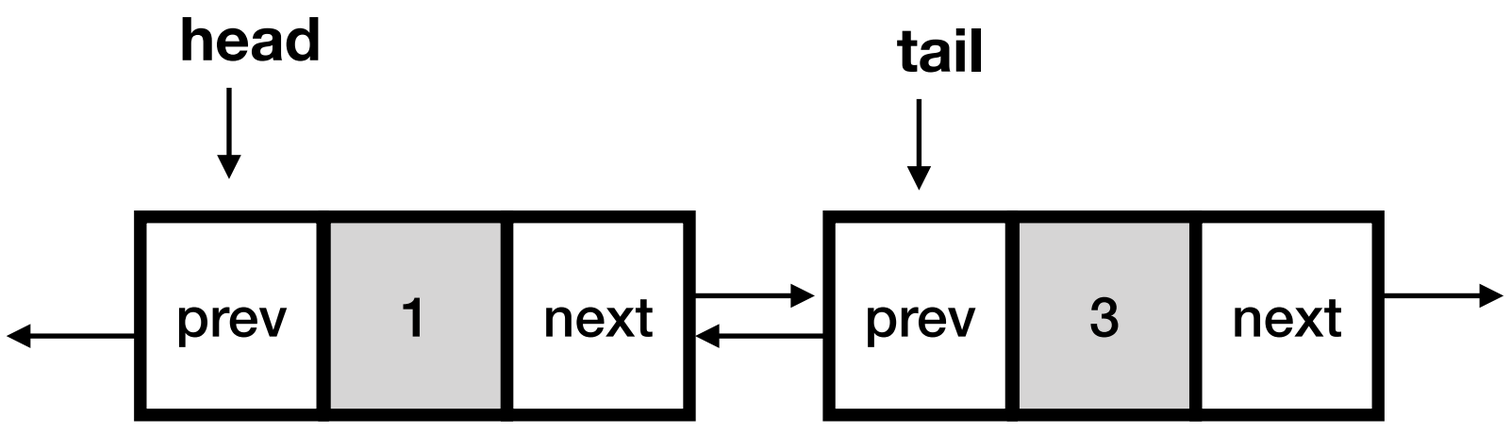
```
void LinkedList::append(int data) {
  Node *newNode = new Node();
  newNode->data = data;

  if (this->head == nullptr) {
    newNode->next = nullptr;
    newNode->prev = nullptr;
    this->head = newNode;
    this->tail = newNode;
  } else {
    this->tail->next = newNode;
    newNode->prev = this->tail;
    newNode->next = nullptr;
    this->tail = newNode;
  }
}
```

example: DoublyLinkedList/LinkedList.cpp

Doubly Linked List - delete()

When we delete a node in a doubly linked list, we need to ensure that we remove the node and correctly re-connect the linked list





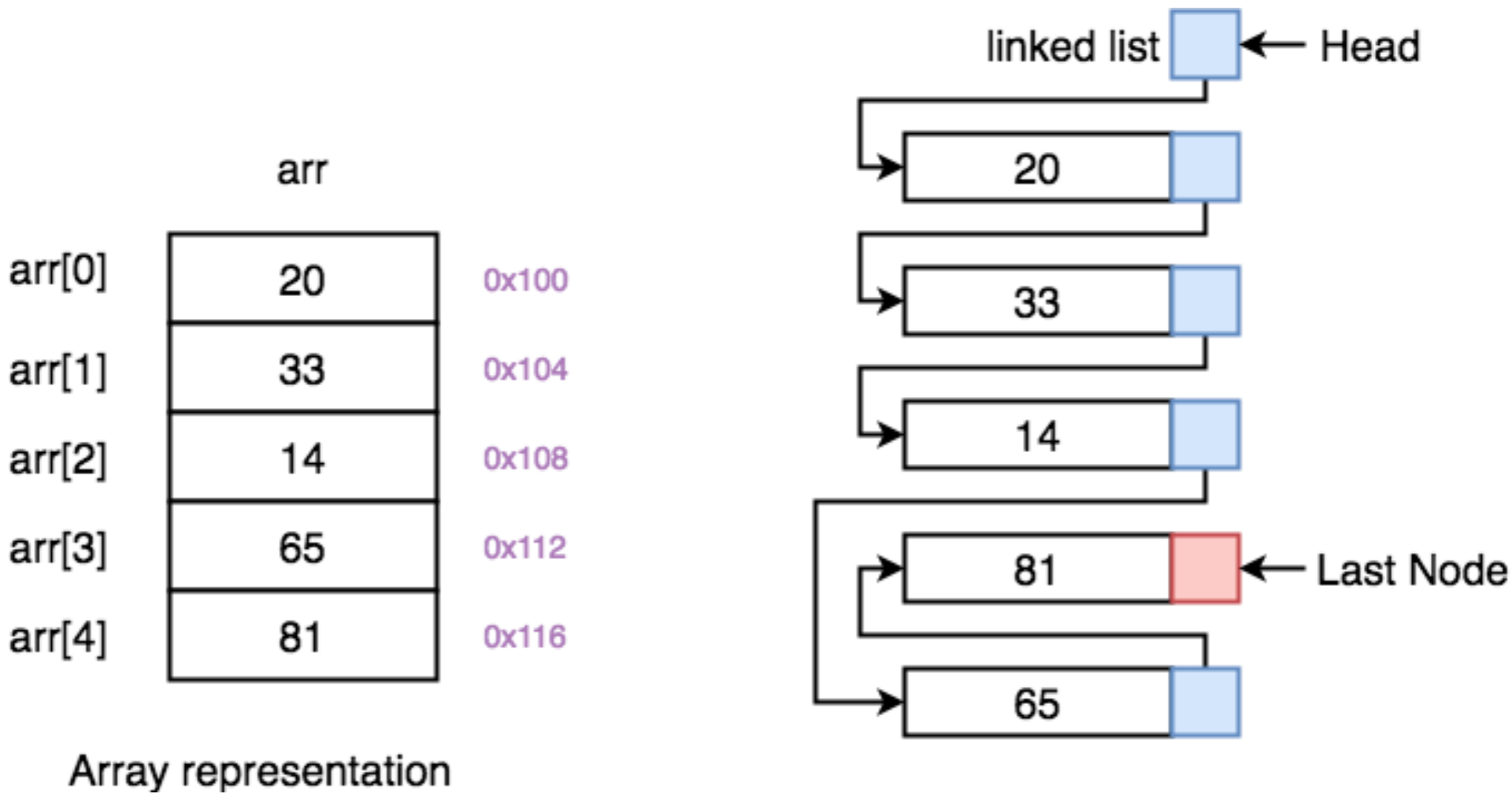
Doubly Linked List - reverse traversal

We can utilize the *tail* pointer and the *prev* pointer on each node to easily traverse a linked list in reverse order without having to know its size

```
void LinkedList::printReverse()
{
  Node *curr = this->tail;

  while (curr != nullptr)
  {
    std::cout << curr->data << " ";
    curr = curr->prev;
  }
  std::cout << std::endl;
}
```

example: DoublyLinkedList/LinkedList.cpp

Linked Lists vs. Arrays / Vectors



- Array's are stored in sequential memory
- Linked Lists are not. We use pointers to get the next element in the list

Linked Lists vs. Arrays / Vectors

Arrays / Vectors	Linked Lists
Fixed Size, Resizing is expensive	Dynamic in size
Accessing elements are constant time *O(1)*	Accessing a random element is expensive, *O(n)*
Insertion and Deletions are inefficient. Elements must be shifted to the correct location in memory (usually involves copying) - *O(n)*	Insertion and Deletion in constant time *O(1)*
Memory must be sequential. If array is not full we waste a lot of space	List nodes are not stored sequentially. Have more memory flexibility and do not waste space
Searching can be done quickly with binary search.	Cannot perform binary search

Stacks

A **stack** is an abstract data structure that contains a collection of elements and implements a LIFO mechanism.

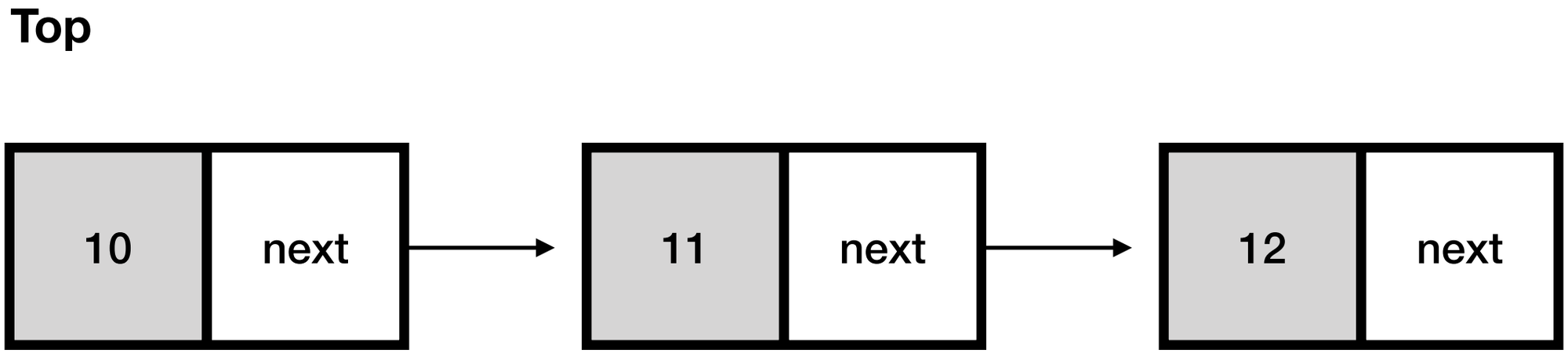
LIFO - Last in first out

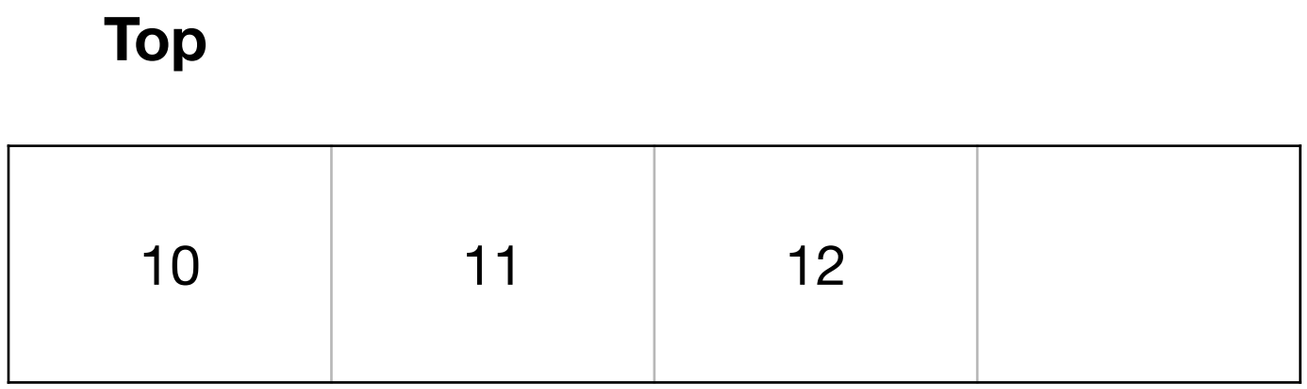
Table 12.17.1: Common stack ADT operations.

Operation	Description	Example starting with stack: 99, 77 (top is 99).
Push(stack, x)	Inserts x on top of stack	Push(stack, 44). Stack: 44, 99, 77
Pop(stack)	Returns and removes item at top of stack	Pop(stack) returns: 99. Stack: 77
Peek(stack)	Returns but does not remove item at top of stack	Peek(stack) returns 99. Stack still: 99, 77
IsEmpty(stack)	Returns true if stack has no items	IsEmpty(stack) returns false.
GetLength(stack)	Returns the number of items in the stack	GetLength(stack) returns 2.

Stack Implementation

A stack can be implemented with a variety of data structures including: *Linked List, Arrays* or *Vectors.*





Stack

Linked List

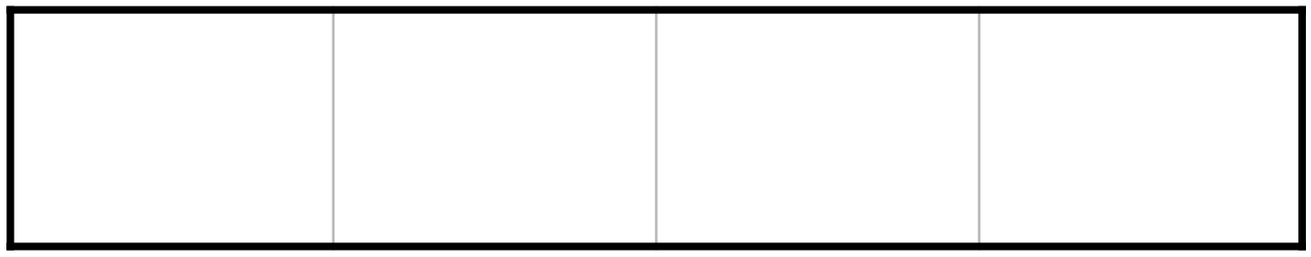
Top

nullptr

Array

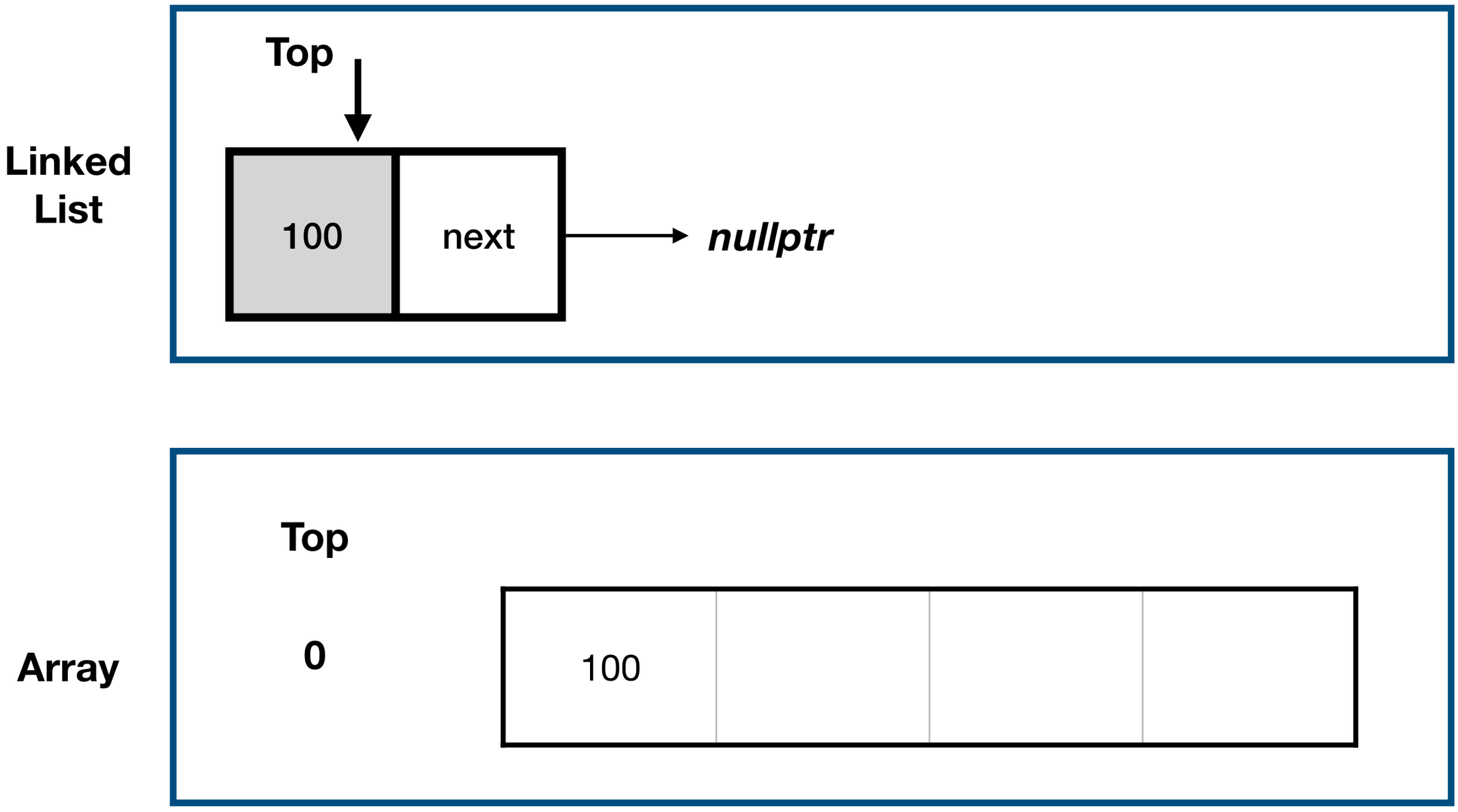
Top

-1



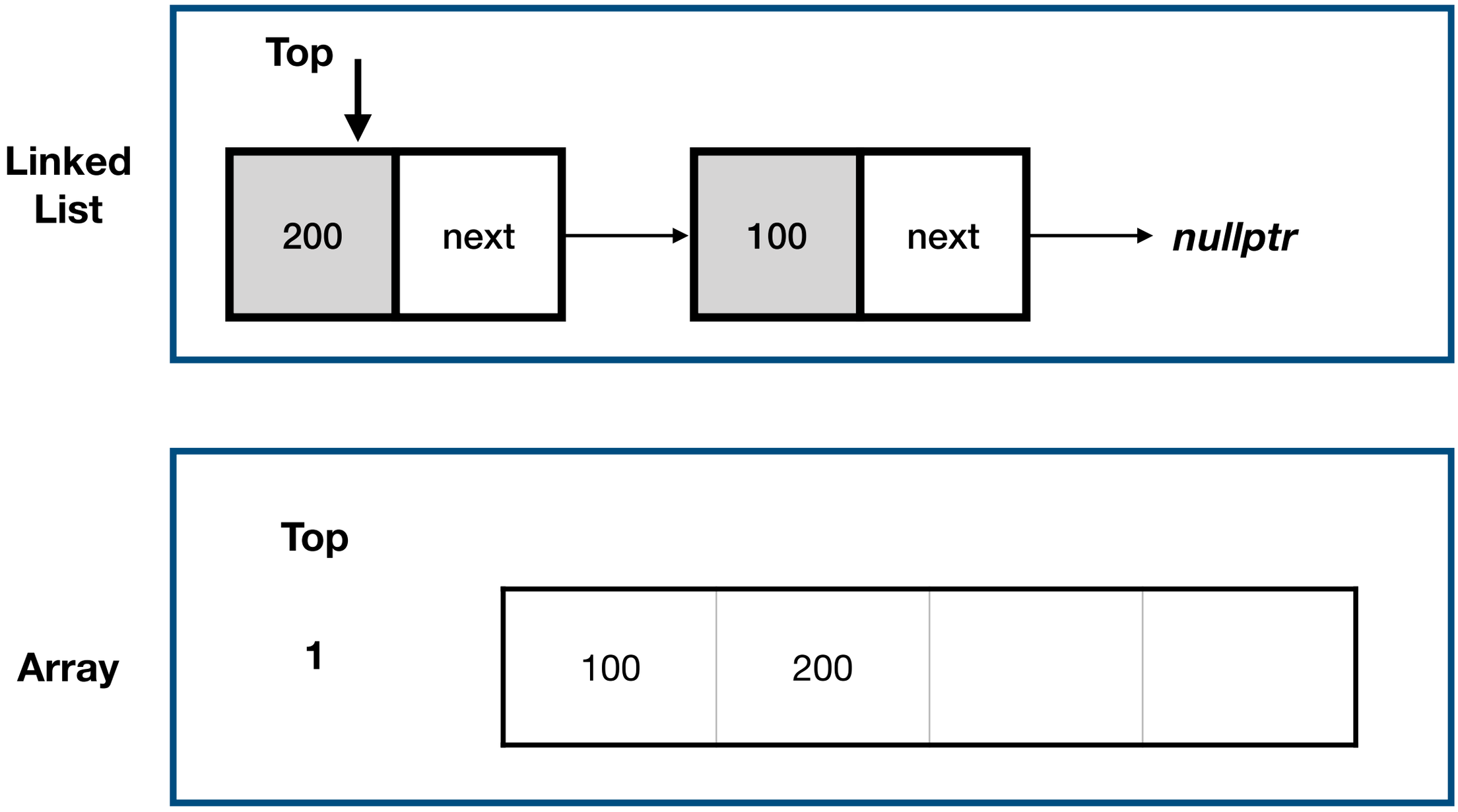
Stack

*push(**100**)*



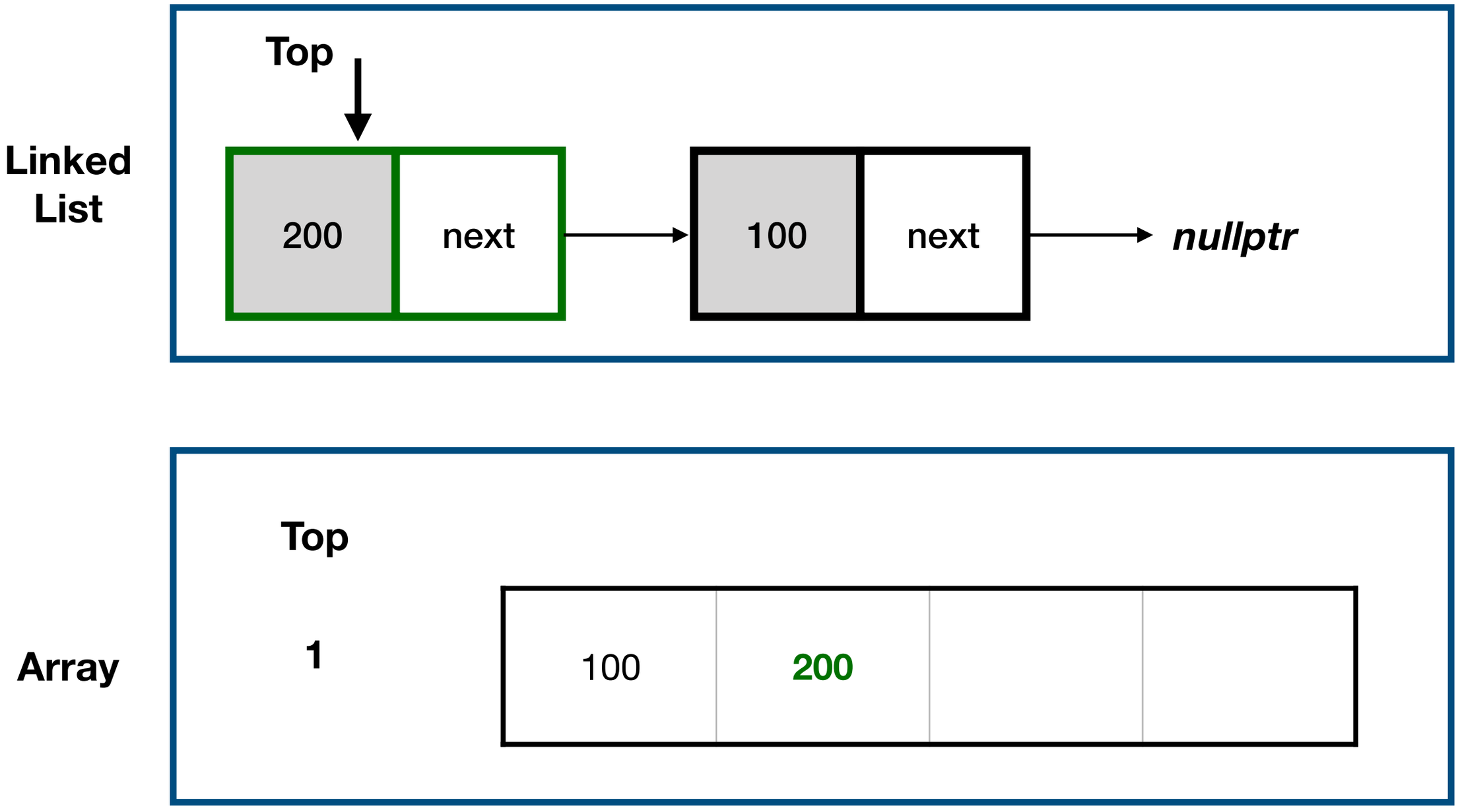
Stack

*push(**200**)*



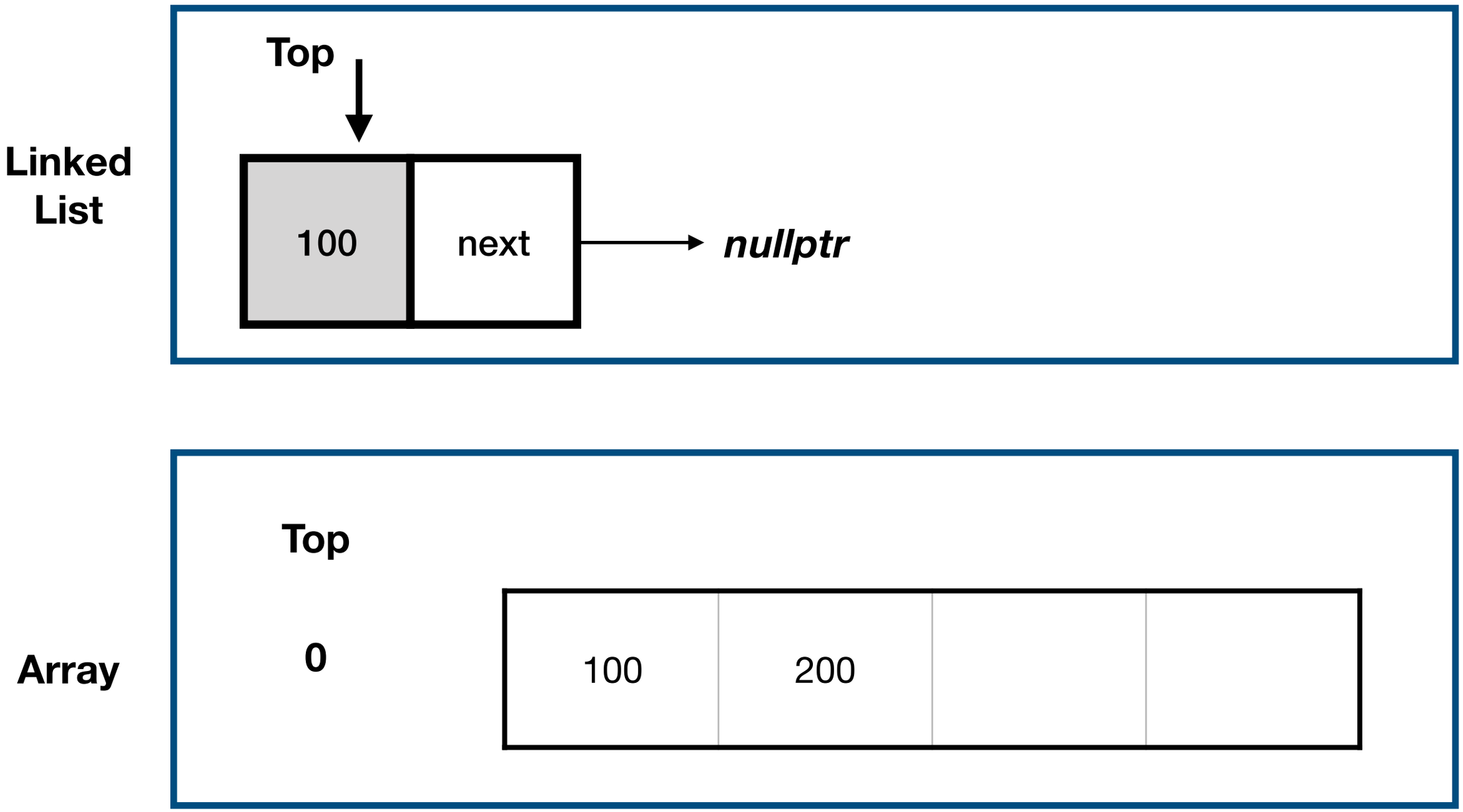
Stack

pop( );



Stack

pop( );



Stack Linked List - push & pop

Push - we want to push a new node onto the stack

Pop - we will pop the top node (head) from the linked list

```
void push(int val) {
  Node *newNode = new Node();
  newNode->data = val;
  newNode->next = head;
  head = newNode;
}
```

```
Node* pop() {
  if (head == nullptr) {
    return nullptr;
  } else {
    Node *top = head;
    head = head->next;
    return top;
  }
}
```

example: Stack/linkedList.cpp

Stack Array - push & pop

Push - add item to end of the array

Pop - pull off item at end of the array

```
void Stack::push(int val) {
  if (top == STACK_SIZE - 1) {
    cout << "Stack Overflow" << endl;
    return;
  }
  stack[++top] = val;
}
```

```
int Stack::pop() {
  if (top == -1) {
    cout << "Stack Underflow" << endl;
    return 0;
  }
  return stack[top--];
}
```

example: Stack/array.cpp

Stacks

Stack Overflow - when we attempt to push an item on the stack that exceeds the stacks size.

Stack Underflow - when an attempt is made to pop an item off an empty stack.

Queue

A **Queue** is an abstract data type which items are inserted at the end of the queue and removed from the front of a queue.

A good example of a queue is a DMV line.

A *Queue* is known as a **First in First out** data structure.

Table 12.19.1: Some common operations for a queue ADT.

Operation	Description	Example starting with queue: 43, 12, 77 (front is 43)
Push(queue, x)	Inserts x at end of the queue	Push(queue, 56). Queue: 43, 12, 77, 56
Pop(queue)	Returns and removes item at front of queue	Pop(queue) returns: 43. Queue: 12, 77
Peek(queue)	Returns but does not remove item at the front of the queue	Peek(queue) return 43. Queue: 43, 12, 77
IsEmpty(queue)	Returns true if queue has no items	IsEmpty(queue) returns false.
GetLength(queue)	Returns the number of items in the queue	GetLength(queue) returns 3.

Queue Linked List - push & pop

Push - we want to push a new node onto the queue (end of the list)

Pop - we will pop the first node (head) from the linked list

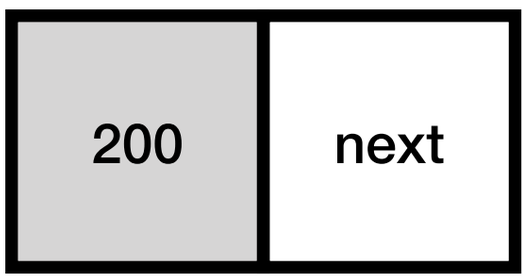
```
void Queue::push(int val) {
  Node *newNode = new Node();
  newNode->data = val;
  newNode->next = nullptr;

  if (head == nullptr) {
    head = newNode;
    tail = newNode;
  } else {
    tail->next = newNode;
    tail = newNode;
  }

  size++;
}
```

```
int Queue::pop() {
  if (size == 0) {
    return -1;
  } else {
    Node *head = this->head;
    int poppedVal = head->data;
    this->head = head->next;
    delete head;
    size--;
    return poppedVal;
  }
}
```

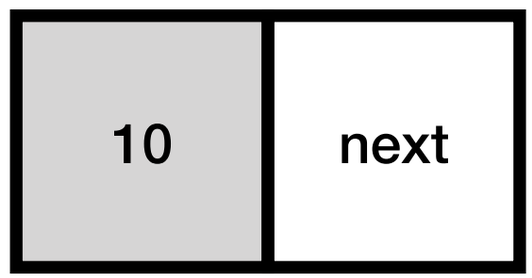
example: Queue/linkedList.cpp

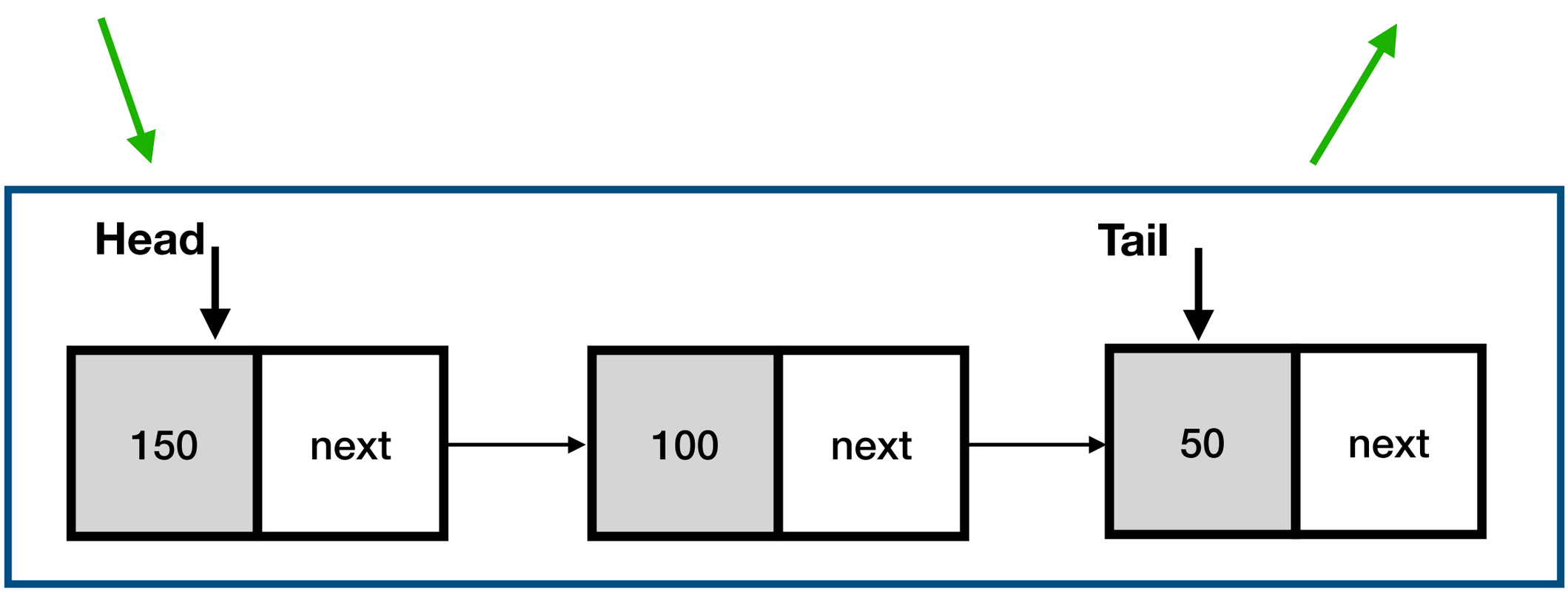
Queue

*push(**200**);*



pop();



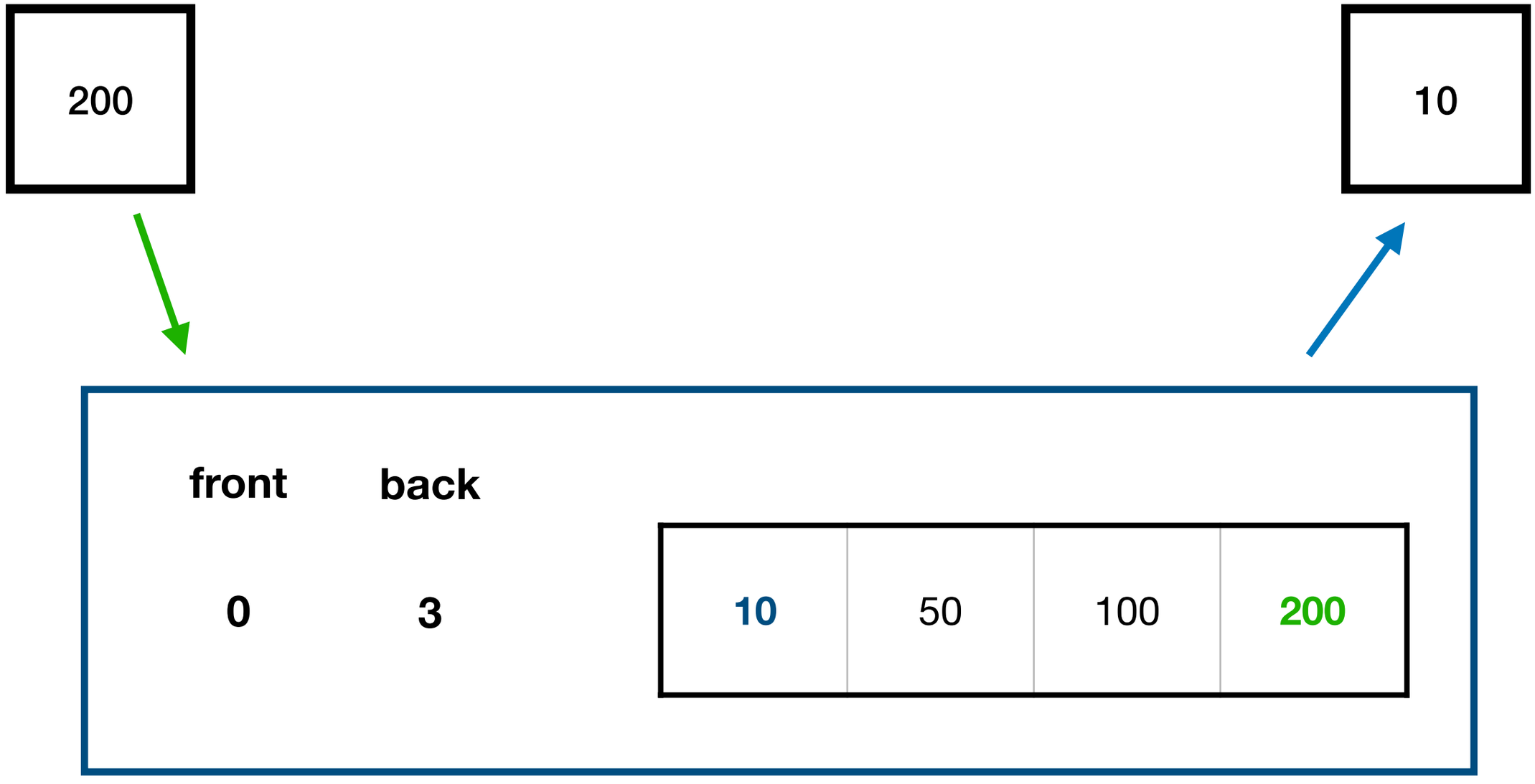


Linked List

Queue

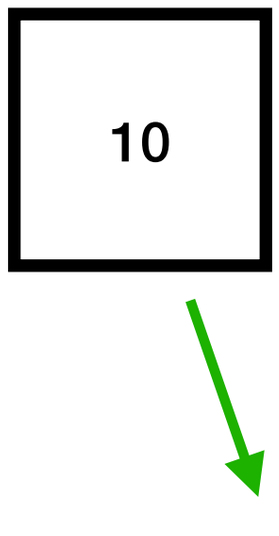
*push(**200**);*

pop( );



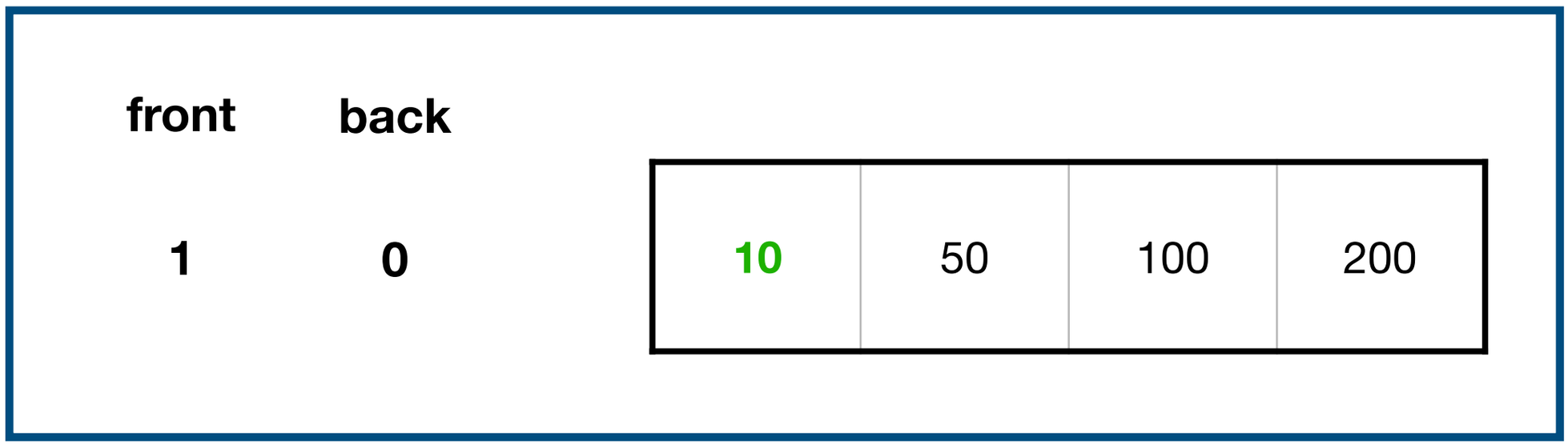
Queue

*push(**200**);*



We can use the % operator so that we can continue to insert values after we extend past the capacity of the array

```
back = (back + 1) % capacity;
```



Queue Array - push & pop

Push - add item to end of the array

Pop - pull off item at beginning of the array

```
void Queue::push(int val) {
  if !(count >= capacity) {
    back = (back + 1) % capacity;
    array[back] = val;
    count++;
  }
}
```

```
int Queue::pop() {
  if (count > 0) {
    int poppedVal = array[front];
    front = (front + 1) % capacity;
    count--;
    return poppedVal;
  }
}
```

example: Queue/array.cpp