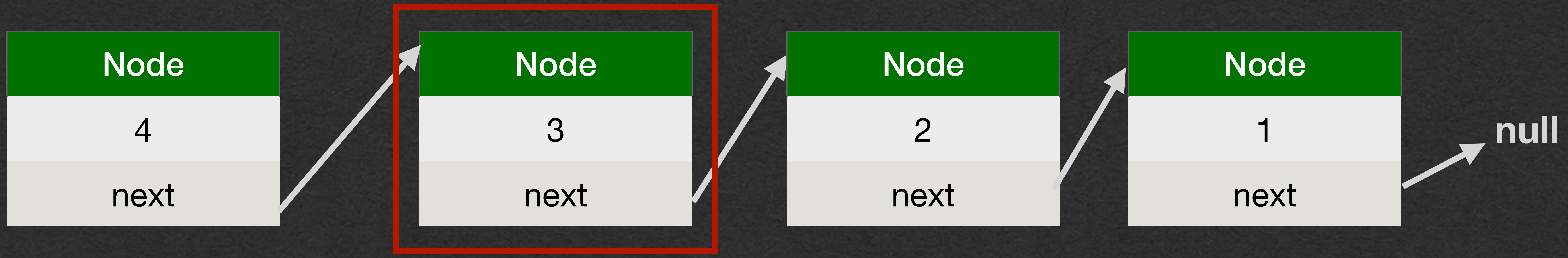
Stack and Queue

But first..
Deleting a Node

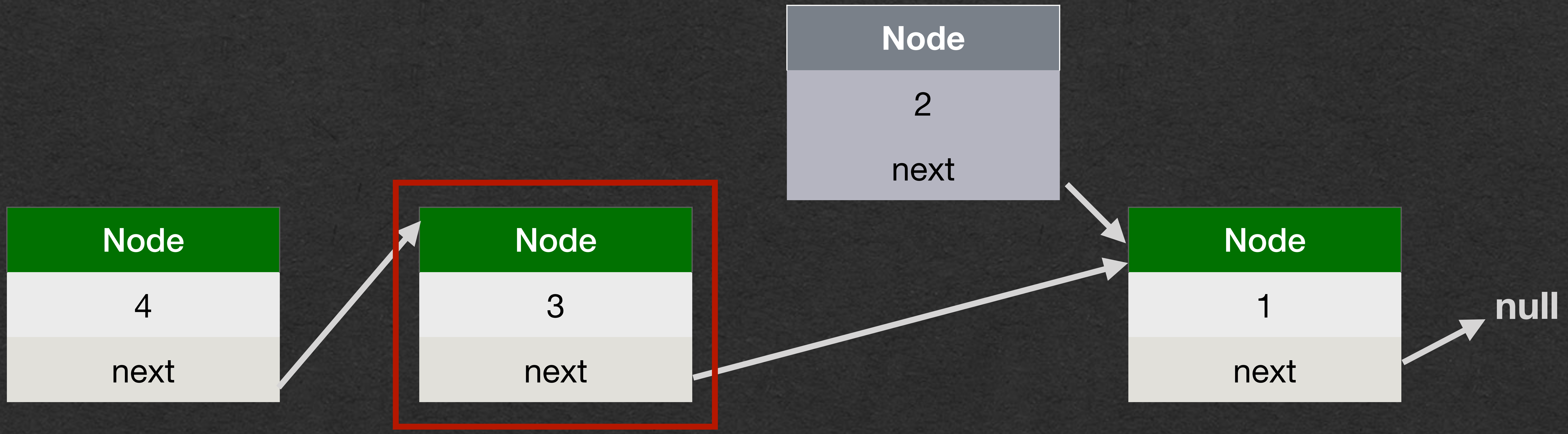
Delete a Node

- Want to delete the node containing 2
- Need a reference to the previous node



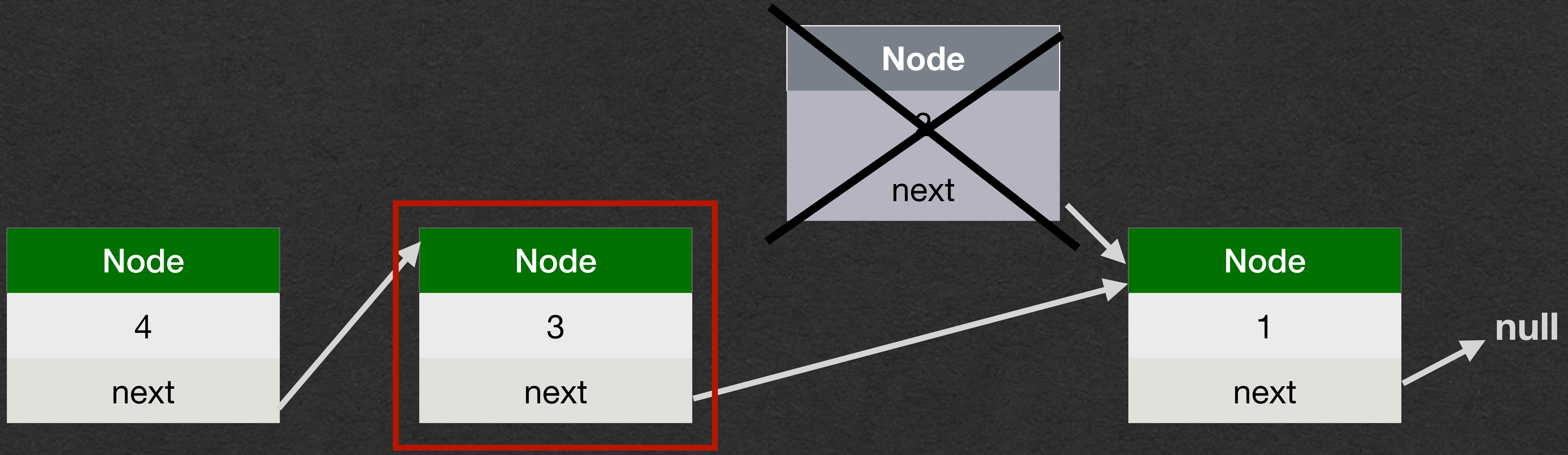
Delete a Node

- Update that node's next to bypass the deleted node
  - Don't have to update deleted node
  - The list no longer refers to this node



Delete a Node

- The deleted node will be garbage collected
- We no longer have a reference to this object



Stack and Queue

- Data structures with specific purposes
  - Restricted features
- All operations are very efficient
  - Inefficient operations are not allowed
- We'll build a stack and queue using linked lists

Stack

Stack

- LIFO
  - Last in First out
  - The last element pushed onto the stack is the first element to be popped off the stack
- Only the element on the top of the stack can be accessed

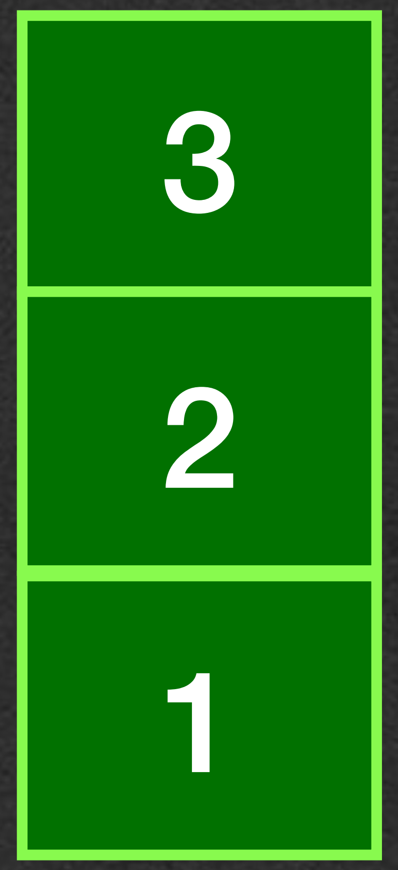
Stack Methods

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the top element of the stack

Stack Usage

- Create a new empty Stack
- Call push to add an element to the top

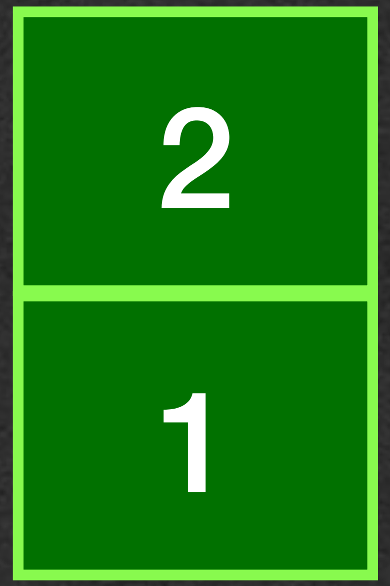
```java
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
}
```



Stack Usage

- Call pop to remove the top element from the Stack

```java
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    int x = stack.pop();
}
```



x == 3

Stack Implementation

```java
public class Stack<T> {
    private LLNode<T> top;

    public Stack() {
        this.top = null;
    }

    public void push(T value) {
        LLNode<T> temp = new LLNode<>(value, this.top);
        this.top = temp;
    }

    public T pop() {
        if (this.top == null) {
            return null;
        } else {
            T temp = this.top.getValue();
            this.top = this.top.getNext();
            return temp;
        }
    }

    public String toString() {
        return this.top.toString();
    }
}
```

- Implement a Stack class using a linked list as an instance variable
- Stack uses the linked list and adapts its methods to implement push and pop

Stack Implementation

```java
public class Stack<T> {
    private LLNode<T> top;

    public Stack() {
        this.top = null;
    }

    public void push(T value) {
        LLNode<T> temp = new LLNode<>(value, this.top);
        this.top = temp;
    }

    public T pop() {
        if (this.top == null) {
            return null;
        } else {
            T temp = this.top.getValue();
            this.top = this.top.getNext();
            return temp;
        }
    }

    public String toString() {
        return this.top.toString();
    }
}
```

push

- Prepend the new element to the front of the list

Stack Implementation

```
public class Stack<T> {
    private LLNode<T> top;

    public Stack() {
        this.top = null;
    }

    public void push(T value) {
        LLNode<T> temp = new LLNode<>(value, this.top);
        this.top = temp;
    }

    public T pop() {
        if (this.top == null) {
            return null;
        } else {
            T temp = this.top.getValue();
            this.top = this.top.getNext();
            return temp;
        }
    }

    public String toString() {
        return this.top.toString();
    }
}
```

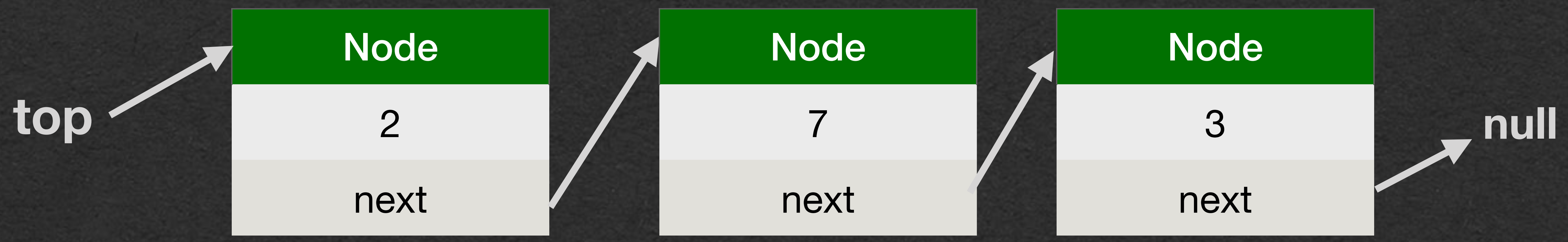
pop

- Remove and return the first element in the list
- If the Stack is empty, return null
- Remove the element by updating this.top to refer to the second node in the list

Stack Usage

- Pushing to a Stack creates a linked list with the pushed elements

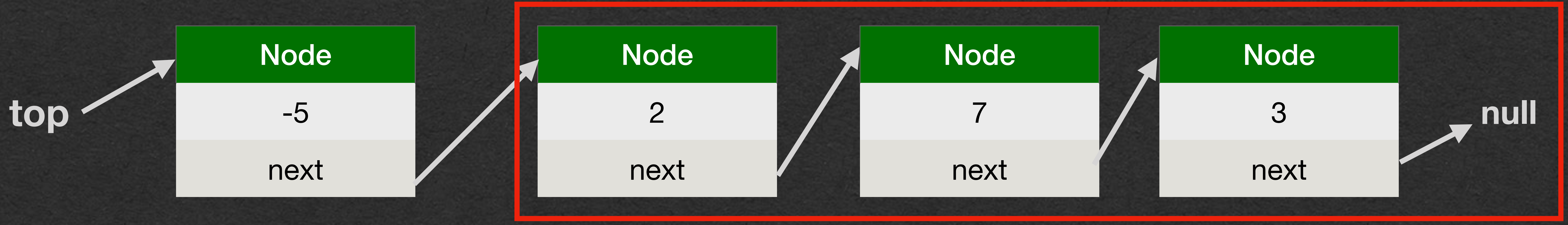
```java
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(3);
    stack.push(7);
    stack.push(2);
}
```



Stack Usage

- To push a value, prepend a new node with that value
- The only list, in the red box, is reused and unchanged
  - Only created a node and updated a reference

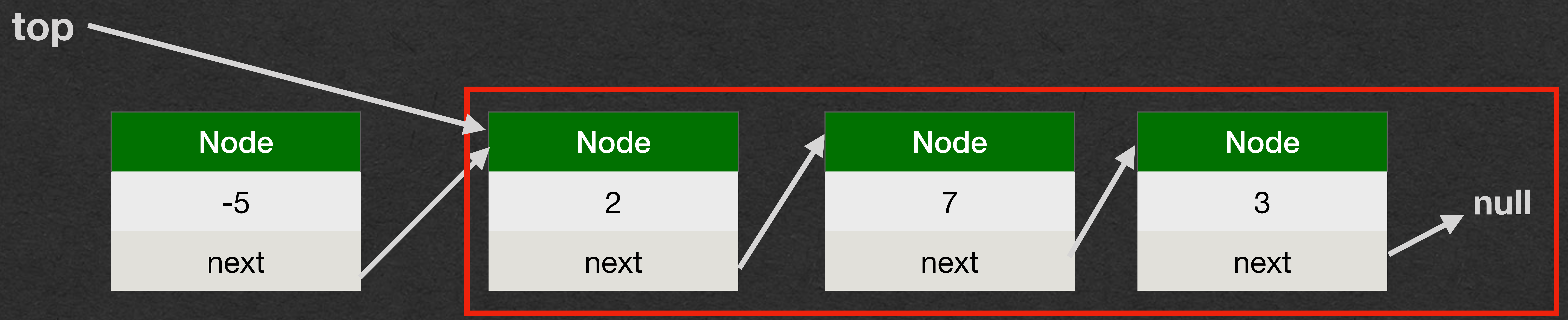
```java
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(3);
    stack.push(7);
    stack.push(2);
    stack.push(-5);
}
```



Stack Usage

- Same efficiency when -5 is popped
- Only update a reference to refer to the second node instead of the first

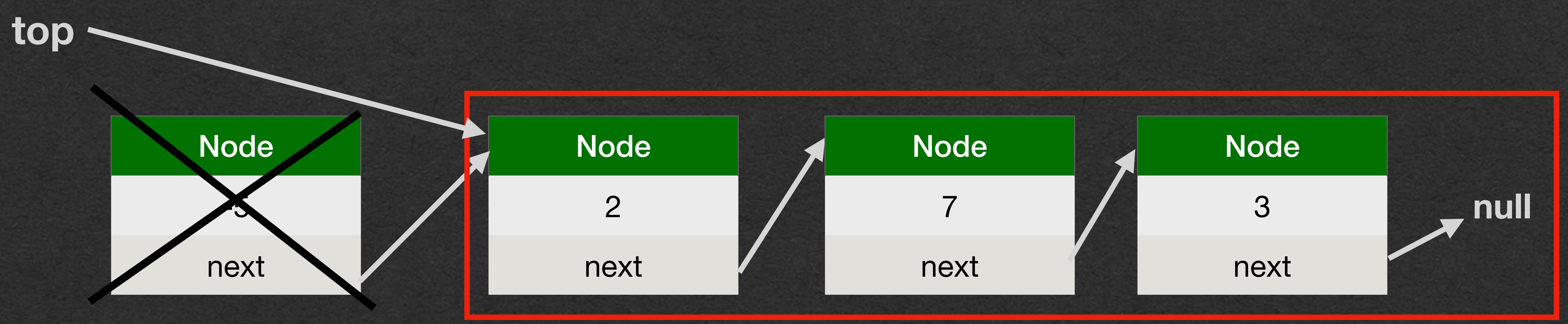
```
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(3);
    stack.push(7);
    stack.push(2);
    stack.push(-5);
    int x = stack.pop();
}
```



Stack Usage

- The node with value 5 is effectively removed from the list by updating the reference stored in top
- The node will be garbage collected

```java
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(3);
    stack.push(7);
    stack.push(2);
    stack.push(-5);
    int x = stack.pop();
}
```



Stack Efficiency

```java
public class Stack<T> {
    private LLNode<T> top;

    public Stack() {
        this.top = null;
    }

    public void push(T value) {
        LLNode<T> temp = new LLNode<>(value, this.top);
        this.top = temp;
    }

    public T pop() {
        if (this.top == null) {
            return null;
        } else {
            T temp = this.top.getValue();
            this.top = this.top.getNext();
            return temp;
        }
    }

    public String toString() {
        return this.top.toString();
    }
}
```

- push and pop are both very efficient
- They take the same amount of time regardless of the size of the Stack
    - We call these O(1) operations
- Compare to linked list algorithms
    - size, toString, find, append all depend on the size of the list and have O(n) runtime

Queue

Queue

- FIFO
  - First in First out
  - The first element enqueued into the queue is the first element to be dequeued out of the queue
- Elements can only be added to the end of the queue
- Only the element at the front of the queue can be accessed

Queue Methods

- Enqueue
  - Add an element to the end of the queue
- Dequeue
  - Remove the front element in the queue

Queue Usage

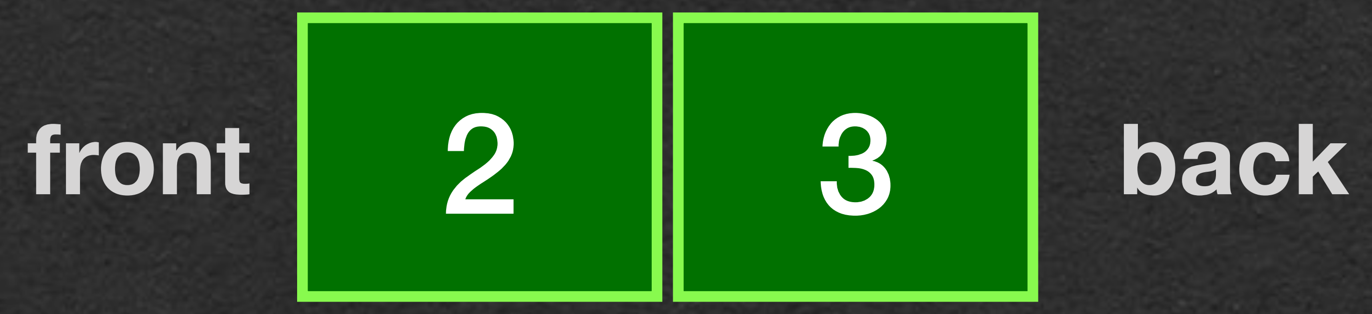
- Create a new empty Queue
- Call enqueue to add an element to the top

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
}
```

Queue Usage

- Call dequeue to remove and return the first element in the queue

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
    int x = queue.dequeue();
}
```



x == 1

Queue Implementation

```java
public class Queue<A> {
    private LinkedListNode<A> front;
    private LinkedListNode<A> back;

    public Queue() {
        this.front = null;
        this.back = null;
    }
    public void enqueue(A value) {
        if (this.back == null) {
            this.back = new LinkedListNode<>(value, null);
            this.front = this.back;
        } else {
            this.back.setNext(new LinkedListNode<>(value, null));
            this.back = this.back.getNext();
        }
    }
    public A dequeue() {
        if (this.front == null) {
            return null;
        }else {
            A toReturn = this.front.getValue();
            this.front = this.front.getNext();
            if (this.front == null) {
                this.back = null;
            }
            return toReturn;
        }
    }
    public String toString() {
        if(this.front == null){
            return "";
        }else {
            return this.front.toString();
        }
    }
}
```

- We implement a Queue in a similar way to Stack
- Use a linked list as an instance variable
- Store all the values in the queue in the linked list

```java
public class Queue<A> {
    private LinkedListNode<A> front;
    private LinkedListNode<A> back;

    public Queue() {
        this.front = null;
        this.back = null;
    }
    public void enqueue(A value) {
        if (this.back == null) {
            this.back = new LinkedListNode<>(value, null);
            this.front = this.back;
        } else {
            this.back.setNext(new LinkedListNode<>(value, null));
            this.back = this.back.getNext();
        }
    }
    public A dequeue() {
        if (this.front == null) {
            return null;
        }else {
            A toReturn = this.front.getValue();
            this.front = this.front.getNext();
            if (this.front == null) {
                this.back = null;
            }
            return toReturn;
        }
    }
    public String toString() {
        if(this.front == null){
            return "";
        }else {
            return this.front.toString();
        }
    }
}
```

Queue Implementation

- We need to work with both ends of the linked list to build a queue
- We could just store the head of the list..
    - But we have to append to enqueue a value
    - Append needs to get to the end of the list first
        - That's slow! O(n)

```java
public class Queue<A> {
    private LinkedListNode<A> front;
    private LinkedListNode<A> back;

    public Queue() {
        this.front = null;
        this.back = null;
    }
    public void enqueue(A value) {
        if (this.back == null) {
            this.back = new LinkedListNode<>(value, null);
            this.front = this.back;
        } else {
            this.back.setNext(new LinkedListNode<>(value, null));
            this.back = this.back.getNext();
        }
    }
    public A dequeue() {
        if (this.front == null) {
            return null;
        }else {
            A toReturn = this.front.getValue();
            this.front = this.front.getNext();
            if (this.front == null) {
                this.back = null;
            }
            return toReturn;
        }
    }
    public String toString() {
        if(this.front == null){
            return "";
        }else {
            return this.front.toString();
        }
    }
}
```

Queue Implementation

- Instead, store references to both the first and last node in the list
- enqueue is now O(1)
    - Just update the next reference of the last node to a new node and update back to the new node

Queue Implementation

- Dequeue is very similar to pop
- We need to track and update both references (front and back) if we dequeue the last element

```java
public class Queue<A> {
    private LinkedListNode<A> front;
    private LinkedListNode<A> back;

    public Queue() {
        this.front = null;
        this.back = null;
    }
    public void enqueue(A value) {
        if (this.back == null) {
            this.back = new LinkedListNode<>(value, null);
            this.front = this.back;
        } else {
            this.back.setNext(new LinkedListNode<>(value, null));
            this.back = this.back.getNext();
        }
    }
    public A dequeue() {
        if (this.front == null) {
            return null;
        }else {
            A toReturn = this.front.getValue();
            this.front = this.front.getNext();
            if (this.front == null) {
                this.back = null;
            }
            return toReturn;
        }
    }
    public String toString() {
        if(this.front == null){
            return "";
        }else {
            return this.front.toString();
        }
    }
}
```

Queue Usage

- Let's walk through this usage of a queue
  - *Abbreviated memory diagram*
- front and back initially refer to null

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

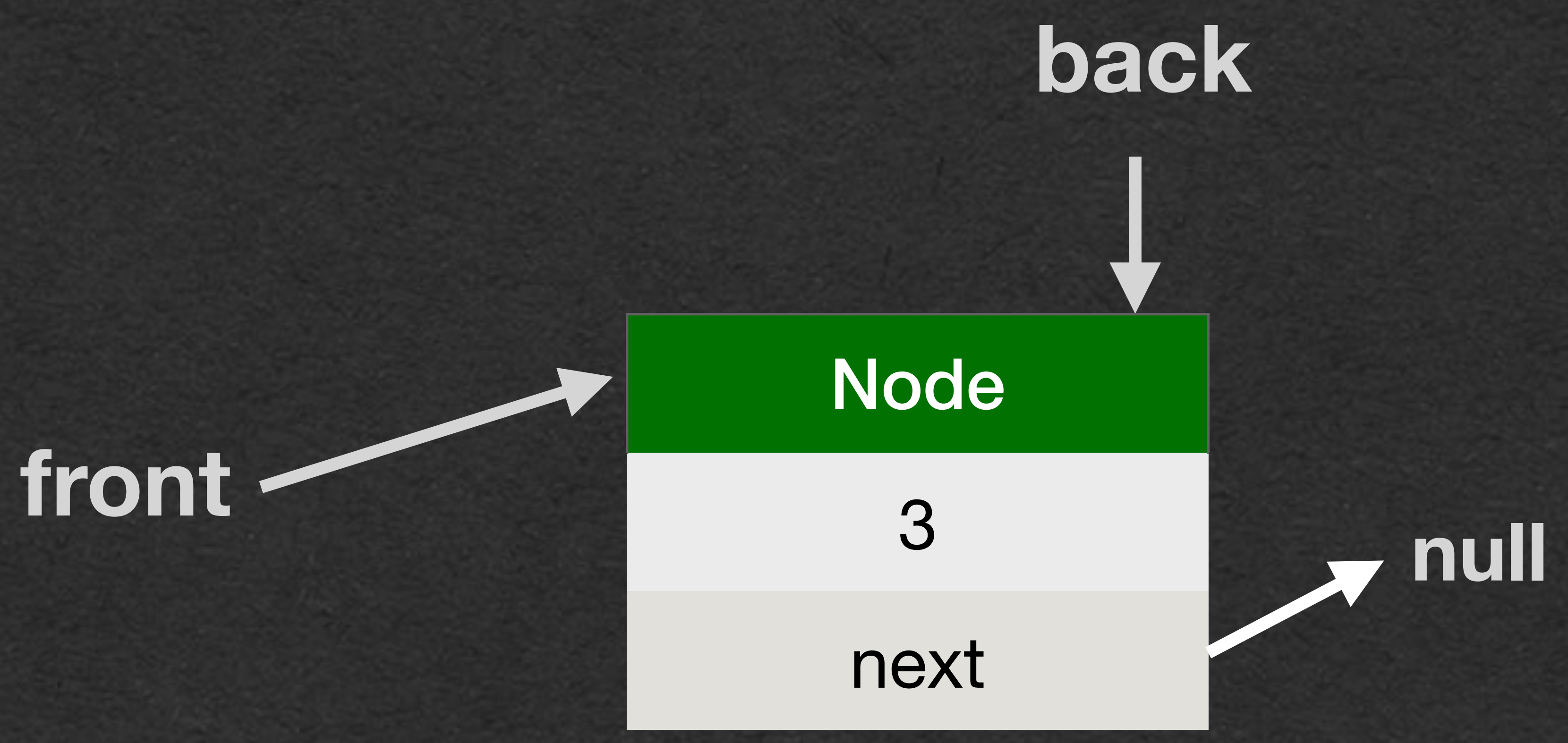
front → null

back → null

Queue Usage

- Enqueueing the first value will replace the nulls stored in fronts and back
- Front and back both refer to the same node

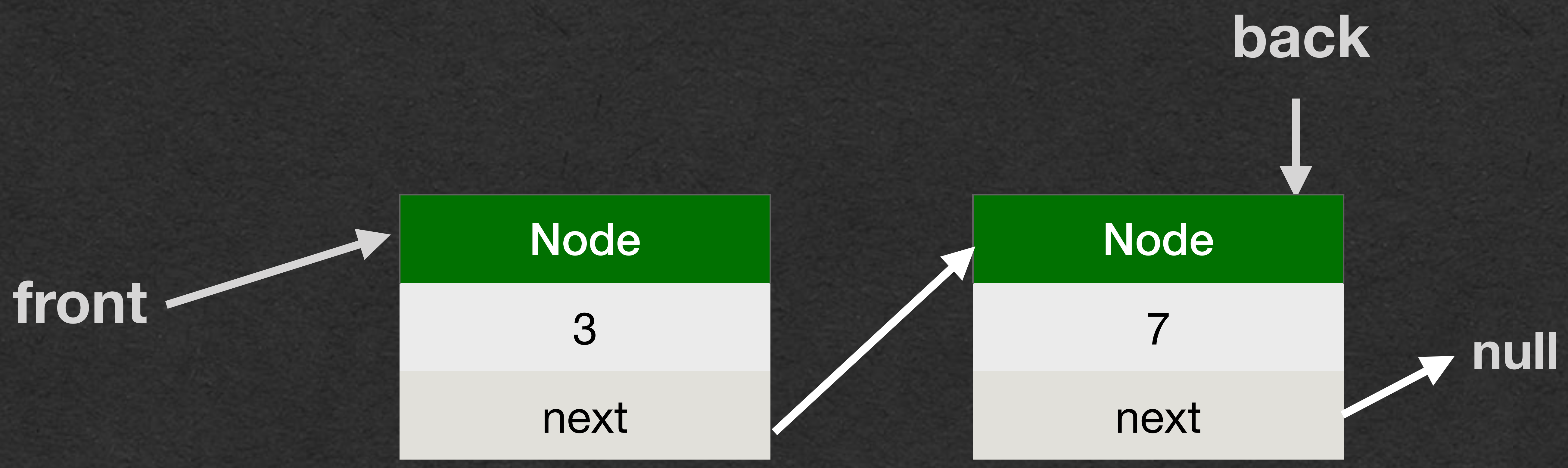
```
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```



Queue Usage

- Enqueue again by creating a new node and updating back to refer to this node

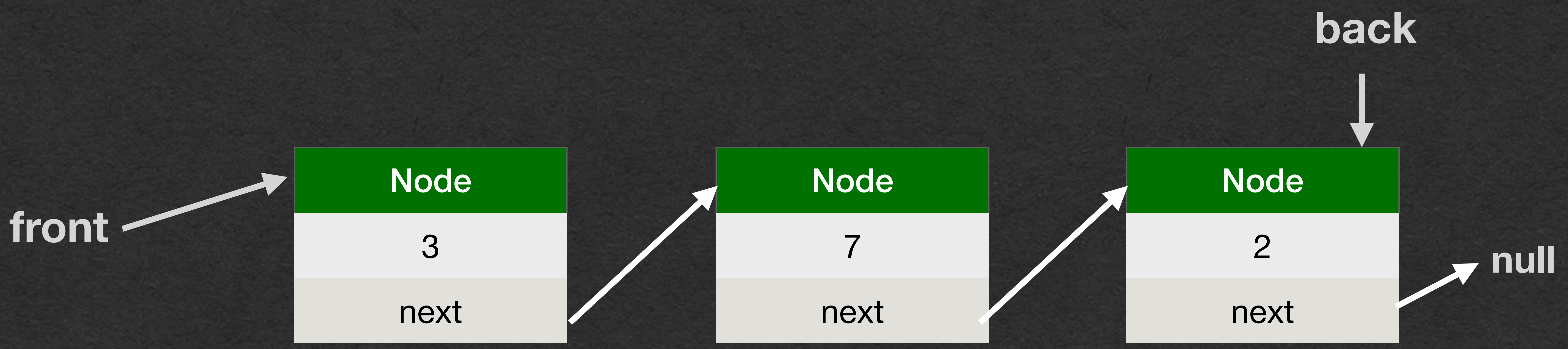
```
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```



Queue Usage

- Notice that each time we enqueue, we do the same amount of work
- Stack and Queue operations do not depend on the size of the data structure

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

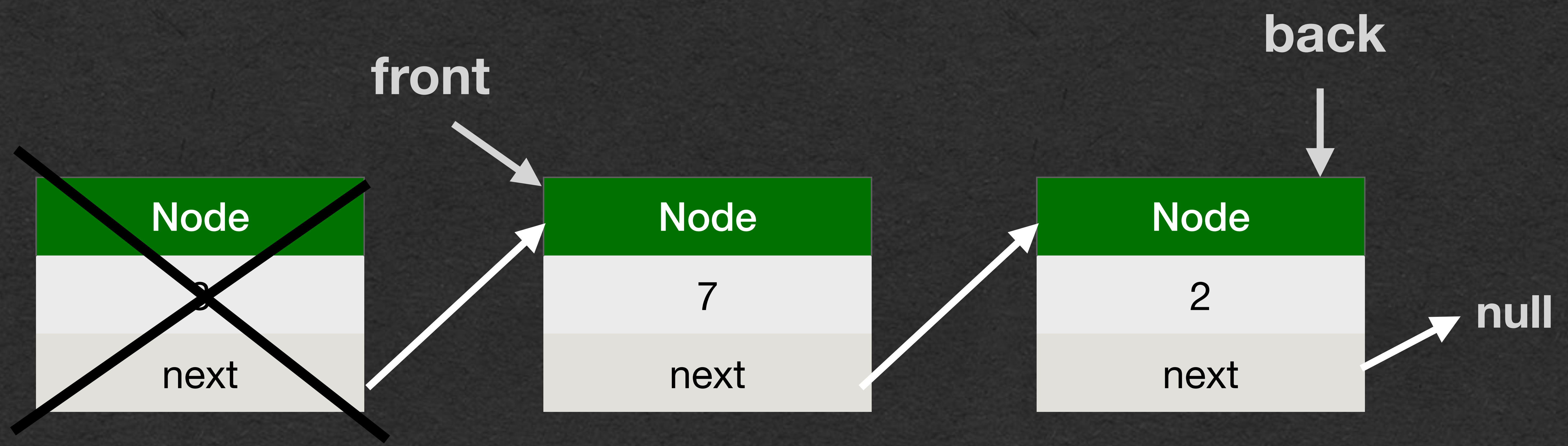


Queue Usage

- When we dequeue, we update the reference stored in front
- The dequeued node will be garbage collected

```
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

x == 3

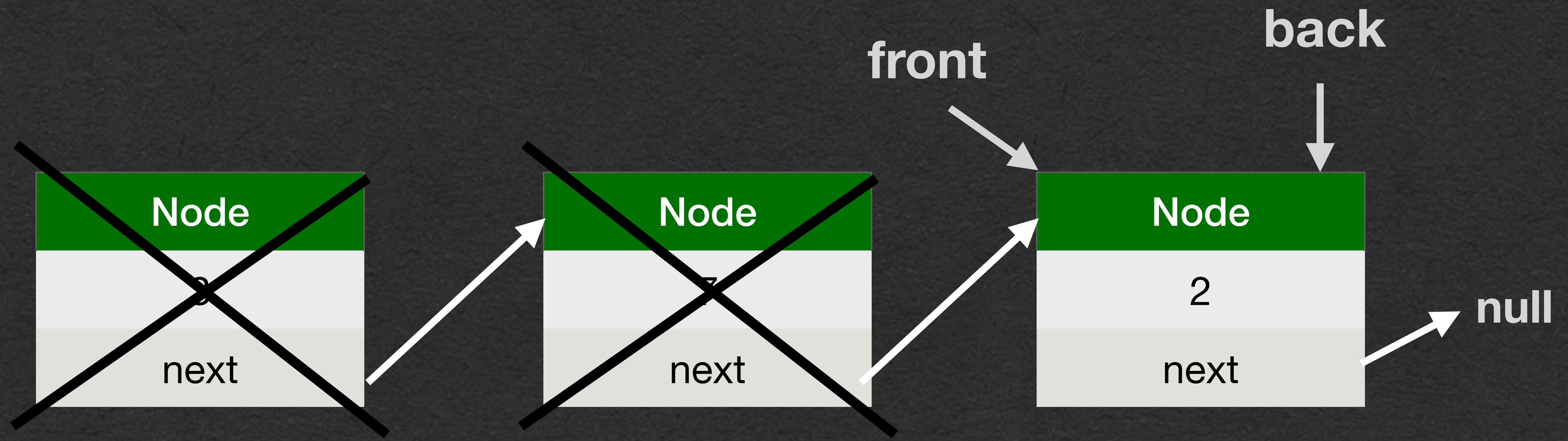


Queue Usage

- Dequeue again to remove 7
- Another node garbage collected

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

x == 7

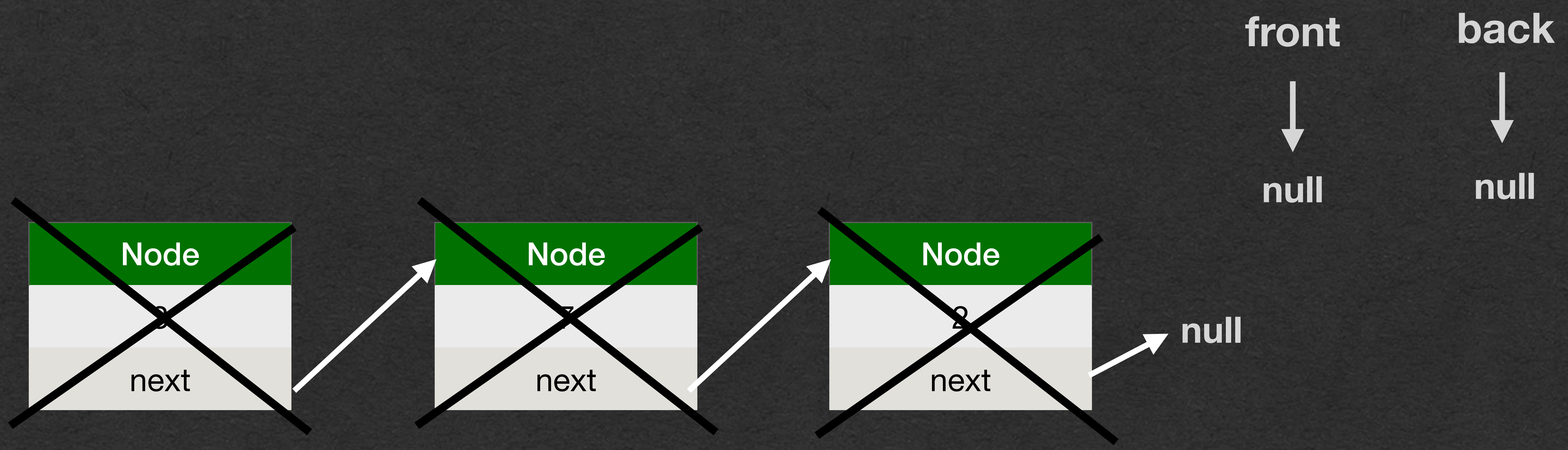


Queue Usage

- When the last value is dequeued, both front and back are set to null
- The whole queue is garbage collected

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

x == 2



Queue Usage

- You should be careful to never dequeue from an empty queue
  - Or pop from an empty stack
- Most implementations will throw an exception
- Ours returns null, then crashes since null can't be stored in a variable of a primitive type
  - null is the lack of a reference. int stores a value, not a reference so it can't be null

```java
public static void main(String[] args) {
    Queue<Integer> queue = new Queue<>();
    queue.enqueue(3);
    queue.enqueue(7);
    queue.enqueue(2);
    int x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
    x = queue.dequeue();
}
```

x == ??
program crashes