Linked Lists

Faraaz Sareshwala

Arrays

- Arrays are contiguous regions of memory directly accessible with an index

0	1	2	3	4
3	1	4	8	9

```
int[] array = new int[5];

array[0] = 3;

System.out.println(array[2]);
```

Arrays: Time Complexity of Operations

Operation	Time Complexity
Iteration	O(n)
Random Access	O(1)
Addition on end	Amortized O(1)
Addition to front	O(n)
Removal from end	O(1)
Removal from front	O(n)

0	1	2	3	4
3	1	4	8	9

Singly Linked Lists

- **Linked List:** list of dynamically allocated data linked together by pointers to form a chain in memory

- Head pointer indicates the start of the linked list
- Node pointing to null indicates the end of the linked list
- Singly linked because a single link only to the next node

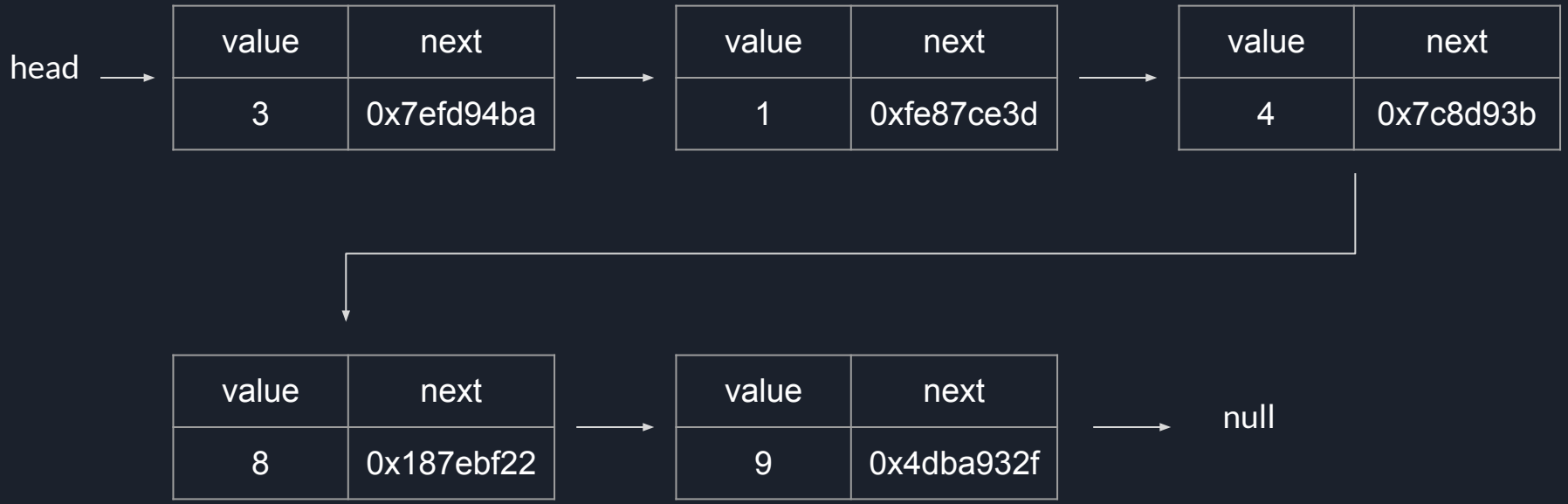
ListNode

```
private class ListNode<T> {
    private T value;
    private ListNode next;

    public ListNode(T value) {
        this(value, null);
    }

    public ListNode(T value, ListNode next) {
        this.value = value;
        this.next = next;
    }
}
```

ListNode

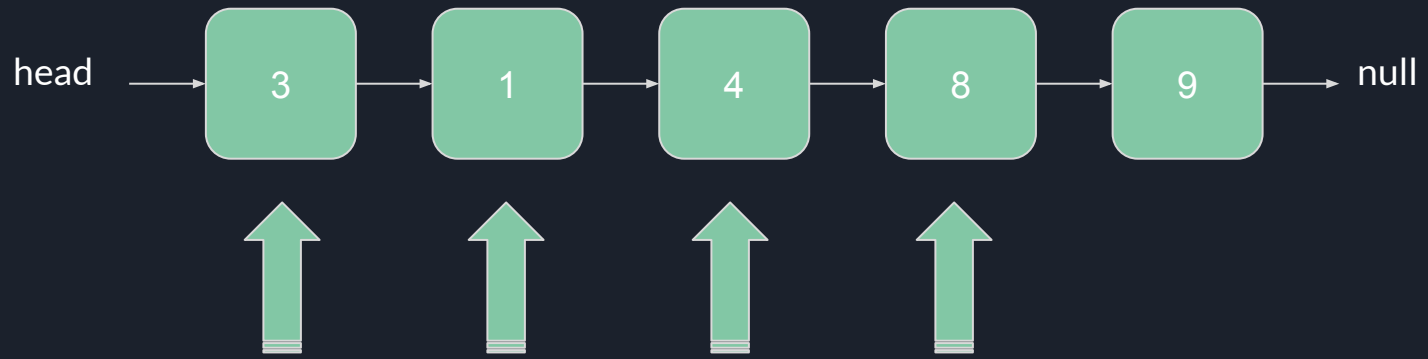
value	next
3	0x7efd94b

Under the Hood



Properties of a Linked List: Traversal

- Only the previous node knows where the next node lives in memory
- Cannot randomly access elements like we could in arrays



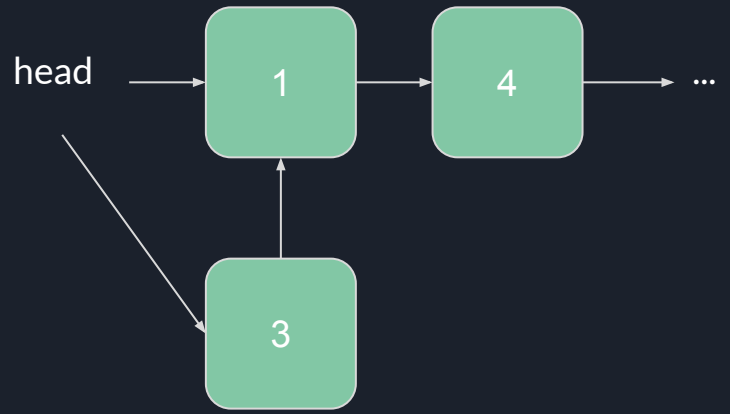
Properties of a Linked List: Dynamic Resize

- Arrays must shift all values to the right in order to insert at the front

0	1	2	3	4
1	4	8	9	

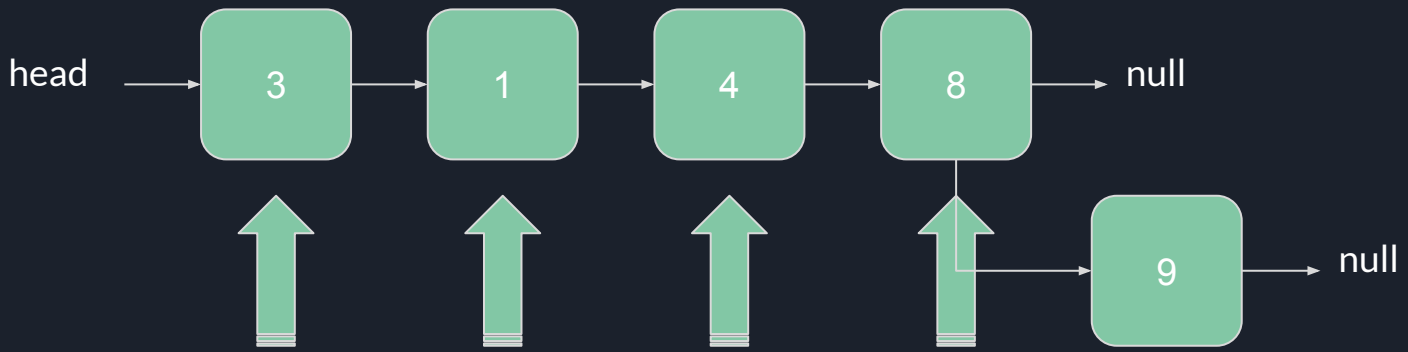
0	1	2	3	4
3	1	4	8	9

- Linked lists can modify pointers to magically add or remove a node to or from the front



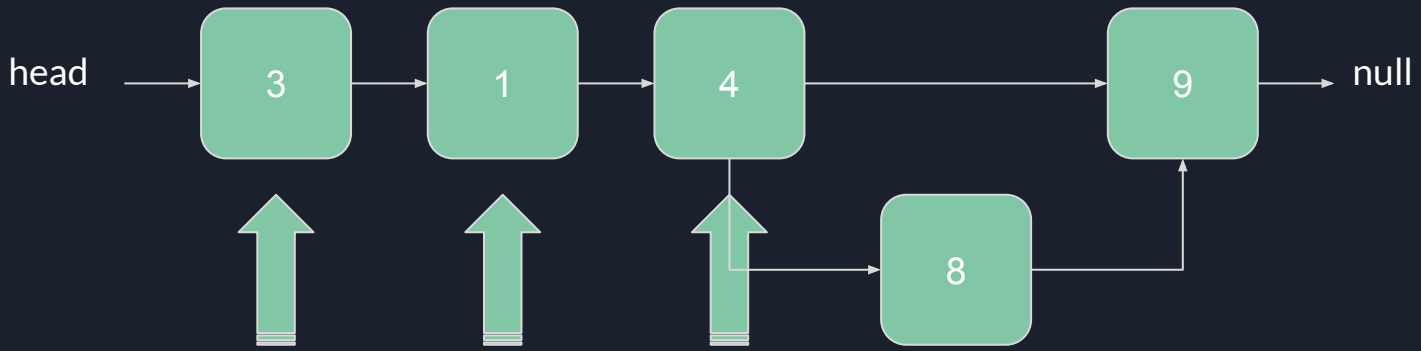
Singly Linked Lists: Insert at End

- Iterate through nodes
- If last element, add new element as next



Singly Linked Lists: Insert in Middle

- Iterate through nodes, until we find where we want to insert
- New node's next is current node's next
- Current node's next is new node

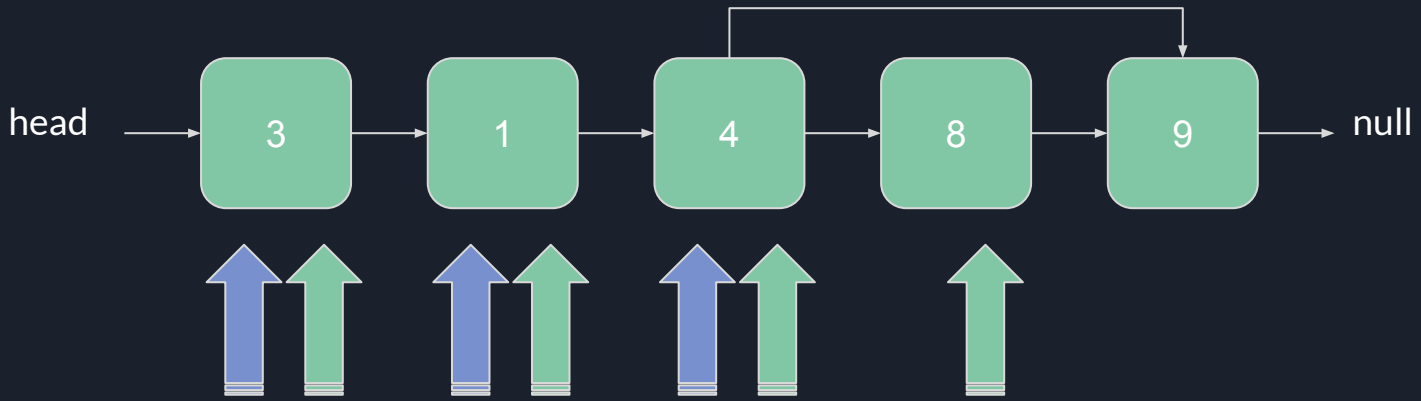


Singly Linked Lists: Remove from Front

- Head points to current head's next node

Singly Linked Lists: Remove from Middle

- Iterate until node is found, keeping track of previous node
- Previous node's next becomes current node's next



- Watch out for edge cases (removing the last element)

Time Complexity: Singly Linked Lists

Arrays

Operation	Time Complexity
Iteration	O(n)
Random Access	O(1)
Addition on end	Amortized O(1)
Addition to front	O(n)
Removal from end	O(1)
Removal from front	O(n)

Singly Linked Lists

Operation	Time Complexity
Iteration	O(n)
Random Access	O(n)
Addition on end	O(n) 
Addition to front	O(1)
Removal from end	O(n) 
Removal from front	O(1)

Optimization: Tail Pointer

Time Complexity: Singly Linked Lists

Arrays

Operation	Time Complexity
Iteration	O(n)
Random Access	O(1)
Addition on end	Amortized O(1)
Addition to front	O(n)
Removal from end	O(1)
Removal from front	O(n)

Singly Linked Lists

Operation	Time Complexity
Iteration	O(n)
Random Access	O(n)
Addition on end	O(1) 
Addition to front	O(1)
Removal from end	O(n) 
Removal from front	O(1)

Example Linked List API

```
// insert value at front of list
void insertFront(...) { ... }

// insert value at end of list
void insertEnd(...) { ... }

// remove first element from list
void removeFront() { ... }

// remove last element from list
void removeFront() { ... }
```

```
// remove element from list, if exists
void remove(...) { ... }

// check if the list contains a value
boolean contains(...) { ... }

// get number of elements in the list
int size() { ... }

// check whether there are any elements
boolean isEmpty() { ... }
```

Linked List

```
public class LinkedList<T> {
    private ListNode head;
    private ListNode tail;

    public LinkedList() { ... }

    public void insert(T value) { ... }
    public void remove(T value) { ... }
    public bool contains(T value) { ... }
    public int size() { ... }
    public bool isEmpty() { ... }
}
```

Singly Linked Lists: Insertion

```
public void insertFront(T value) {
    ListNode ptr = new ListNode(value);
    if (isEmpty()) {
        head = ptr;
        tail = ptr;
    }
    else {
        ptr.next = head;
        head = ptr;
    }
}
```

```
public void insertEnd(T value) {
    ListNode ptr = new ListNode(value);
    if (isEmpty()) {
        head = ptr;
        tail = ptr;
    }
    else {
        tail.next = ptr;
        tail = ptr;
    }
}
```

Singly Linked Lists: Remove

```
public void removeFront() {
    if (isEmpty()) {
        return;
    }
    if (head == tail) {
        head = null;
        tail = null;
        return;
    }
    head = head.next;
}
```

```
public void removeEnd() {
    if (isEmpty()) {
        return;
    }

    if (head == tail) {
        head = null;
        tail = null;
        return;
    }

    ListNode curr = head;
    ListNode prev = null;
    while (curr != tail) {
        prev = curr;
        curr = curr.next;
    }

    prev.next = null;
    tail = prev;
}
```

Singly Linked Lists: Remove

```
public void remove(T value) {
    if (isEmpty()) {
        return;
    }
    if (head == tail) {
        head = null;
        tail = null;
        return;
    }

    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        if (curr.value != value) {
            prev = curr;
            curr = curr.next;
            continue;
        }
        if (curr == head) {
            head = head.next;
        }
        else if (curr == tail) {
            prev.next = null;
            tail = prev;
        }
        else {
            prev.next = curr.next;
        }
    }
```

Singly Linked Lists: Contains

```
public boolean contains(T value) {
    ListNode curr = head;
    while (curr != null) {
        if (curr.value == value) {
            return true;
        }
        curr = curr.next;
    }
    return false;
}
```

Singly Linked Lists: Size

```
public int size() {
    ListNode curr = head;

    int size = 0;
    while (curr != null) {
        curr = curr.next;
        size++;
    }

    return size;
}
```

Singly Linked Lists: Is Empty

```
public int isEmpty() {
    if (head == null && tail == null) {
      return true;
    }
    return false;
}
```

Time Complexity: Singly Linked Lists

Arrays

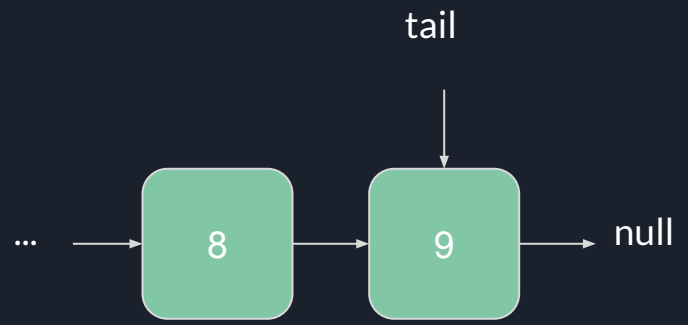
Operation	Time Complexity
Iteration	O(n)
Random Access	O(1)
Addition on end	Amortized O(1)
Addition to front	O(n)
Removal from end	O(1)
Removal from front	O(n)

Singly Linked Lists

Operation	Time Complexity
Iteration	O(n)
Random Access	O(n)
Addition on end	O(1)
Addition to front	O(1)
Removal from end	O(n) 
Removal from front	O(1)

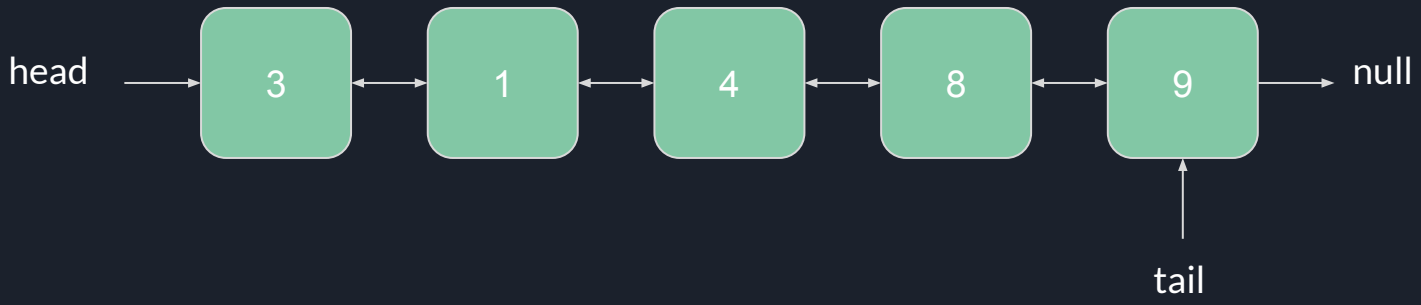
Limitations of a Singly Linked List

- Only the previous node knows where the next node lives in memory
- Tail pointer gives direct access to the last node in the list
- How can we get from the last node of the list to the node before the last?
    - We can’t
- Solution: use links that point backwards as well



Doubly Linked Lists

- Each node has both a next and prev pointer
- Can get to previous node by following the prev pointer



Time Complexity: Doubly Linked Lists

Arrays

Operation	Time Complexity
Iteration	O(n)
Random Access	O(1)
Addition on end	Amortized O(1)
Addition to front	O(n)
Removal from end	O(1)
Removal from front	O(n)

Doubly Linked Lists

Operation	Time Complexity
Iteration	O(n)
Random Access	O(n)
Addition on end	O(1)
Addition to front	O(1)
Removal from end	O(1)
Removal from front	O(1)

Doubly Linked List: `ListNode`

```
private class ListNode<T> {
    private T value;
    private ListNode prev;
    private ListNode next;

    public ListNode(T value) {
        this(value, null, null);
    }

    public ListNode(T value, ListNode prev, ListNode next) {
        this.value = value;
        this.prev = prev;
        this.next = next;
    }
}
```