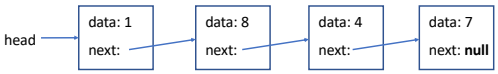
Linked list visualized

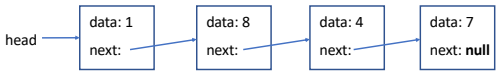
[1, 8, 4, 7]



1

Adding to the front

[1, 8, 4, 7]

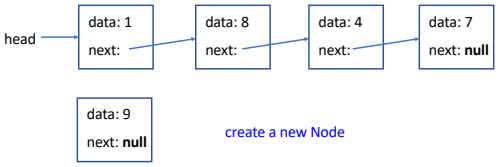


How can we add a value to the front of the list?

2

Adding to the front: add 9 to front

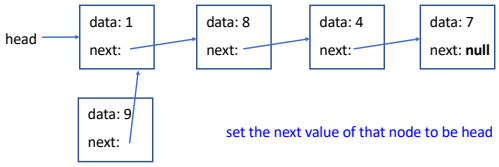
[1, 8, 4, 7]



3

Adding to the front: add 9 to front

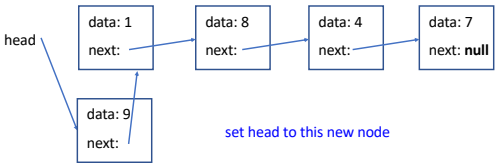
[1, 8, 4, 7]



4

Adding to the front: add 9 to front

[1, 8, 4, 7]



5

Adding to the front: add 9 to front

[1, 8, 4, 7]

6

Adding to the front: add 9 to front

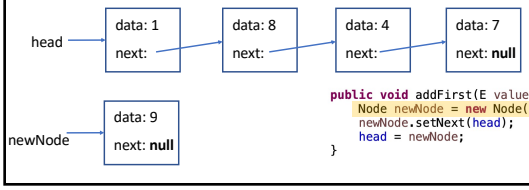
[1, 8, 4, 7]


```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

7

Adding to the front: add 9 to front

[1, 8, 4, 7]

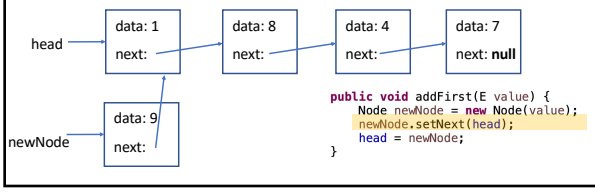



```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

8

Adding to the front: add 9 to front

[1, 8, 4, 7]

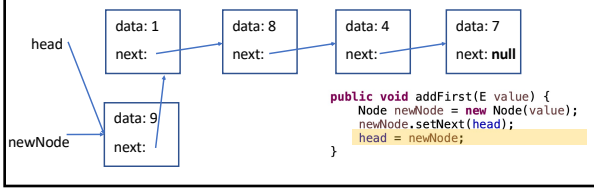



```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

9

Adding to the front: add 9 to front

[9, 1, 8, 4, 7]




```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

10

Adding to the front

Does this work if the linked list is empty?

head: null

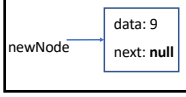
```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

11

Adding to the front

Does this work if the linked list is empty?

head: null



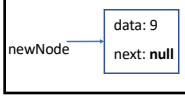

```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

12

Adding to the front

Does this work if the linked list is empty?

head: null

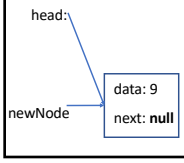



```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

13

Adding to the front

Does this work if the linked list is empty?

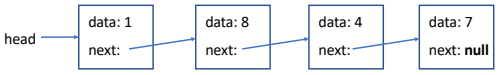



```
public void addFirst(E value) {
    Node newNode = new Node(value);
    newNode.setNext(head);
    head = newNode;
}
```

14

Removing from the front

[1, 8, 4, 7]

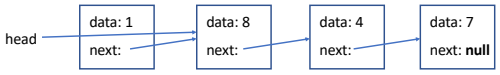


How can we delete a value to the front of the list?
(assuming the list isn't empty)

15

Removing from the front

[1, 8, 4, 7]

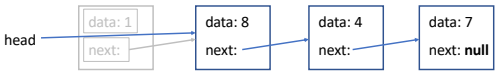


Simply move head down the list!

16

Removing from the front

[1, 8, 4, 7]

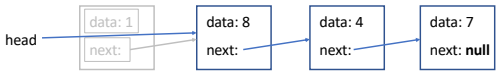


Simply move head down the list!

17

Removing from the front

[1, 8, 4, 7]

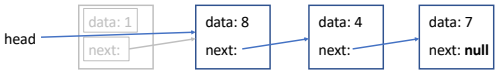


How can do this in code?

18

Removing from the front

[1, 8, 4, 7]

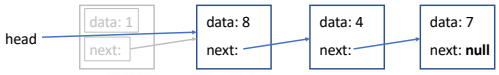


How can do this in code? `head = head.next();`

19

Removing from the front

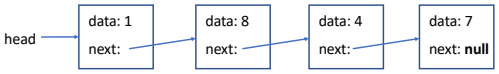
[1, 8, 4, 7]




```
public E removeFirst() {
    if( head == null ) {
        return null;
    } else {
        E returnMe = head.value();
        head = head.next(); // move head down the list
        return returnMe;
    }
}
```

20

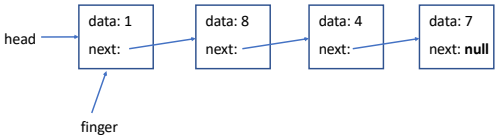
Iterating through a linked list



How can we iterate through a linked list?

21

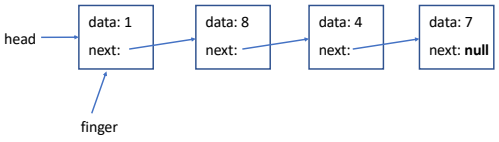
Iterating through a linked list



Use a variable starting at the head and move it down the list

22

Iterating through a linked list

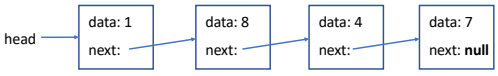


Use a variable starting at the head and move it down the list

How will we know when we're at the end?

23

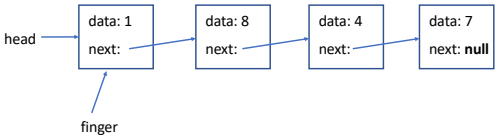
Iterating through a linked list: printing it




```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

24

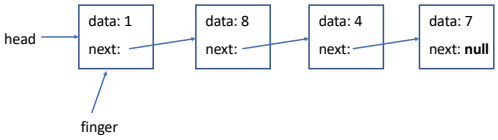
Iterating through a linked list: printing it




```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

25

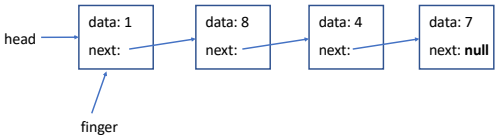
Iterating through a linked list: printing it




```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

26

Iterating through a linked list: printing it

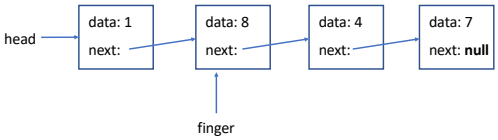


1

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

27

Iterating through a linked list: printing it

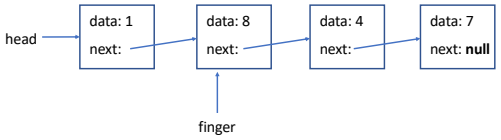


1

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

28

Iterating through a linked list: printing it

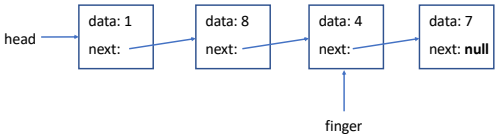


1
8

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

29

Iterating through a linked list: printing it

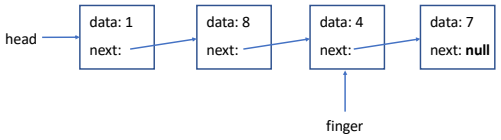


1
8

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

30

Iterating through a linked list: printing it

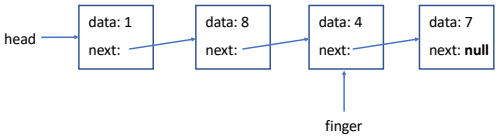


1
8
4

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

31

Iterating through a linked list: printing it

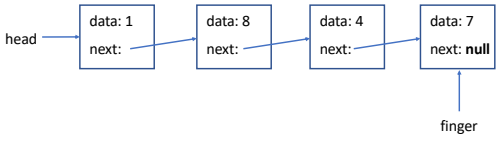


1
8
4

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

32

Iterating through a linked list: printing it

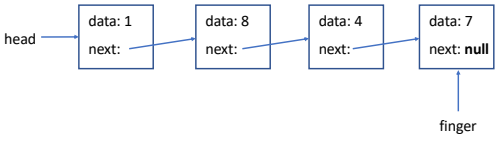



```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

1
8
4

33

Iterating through a linked list: printing it

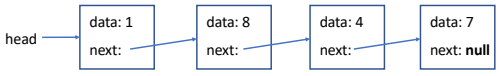



```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

1
8
4
7

34

Iterating through a linked list: printing it



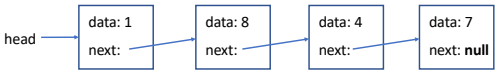
finger:**null**

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

1
8
4
7

35

Iterating through a linked list: printing it



finger:**null**

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

1
8
4
7

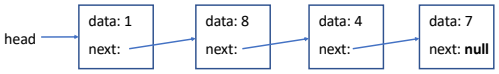
36

Iterating through a linked list


```
Node finger = head;

while (finger != null ) {
    // do something with the current node, finger
    finger = finger.next();
}
```

37

Checking if a value is in the list



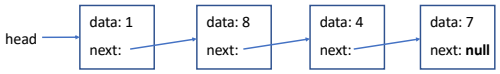
Write a method contains that takes a value as input and returns a boolean indicating whether or not the linked list contains that value:

```
public boolean contains(E value){
```

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

38

Checking if a value is in the list



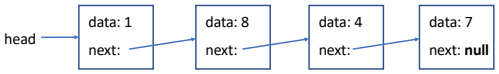
Write a method contains that takes a value as input and returns a boolean indicating whether or not the linked list contains that value:

```
public boolean contains(E value){
    Node finger = head;

    while (finger != null && !finger.value().equals(value)) {
        finger = finger.next();
    }

    return finger != null;
}
```

```
Node finger = head;

while (finger != null ) {
    System.out.println(finger.value());
    finger = finger.next();
}
```

39

Checking if a value is in the list

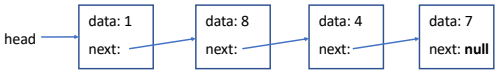



```
public boolean recursiveContains(E value){
    return containsHelper(head, value);
}
```

```
private boolean containsHelper(Node current, E value){
    if( current == null ){
        return false;
    }else if( current.value().equals(value) ){
        return true;
    }else{
        return containsHelper(current.next(), value);
    }
}
```

40

Checking if a value is in the list

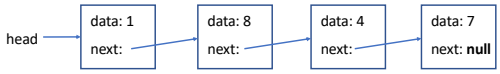



```
public int indexOf(E value){
    int index = 0;

    Node finger = head;

    while( finger != null && !finger.value().equals(value) ){
        finger = finger.next();
        index++;
    }

    return finger != null ? index: -1;
}
```

What does this method do?

41

Checking if a value is in the list




```
public int indexOf(E value){
    int index = 0;

    Node finger = head;

    while( finger != null && !finger.value().equals(value) ){
        finger = finger.next();
        index++;
    }

    return finger != null ? index: -1;
}
```

```
public boolean contains(E value){
    Node finger = head;

    while (finger != null && !finger.value().equals(value)) {
        finger = finger.next();
    }

    return finger != null;
}
```

Returns the index the value occurs at

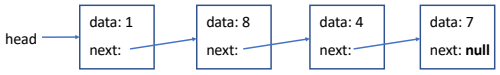
42

```
public int indexOf(E value){
    int index = 0;

    Node finger = head;

    while( finger != null && !finger.value().equals(value) ){
        finger = finger.next();
        index++;
    }

    return index;
}
```

43

Adding to the end

[1, 8, 4, 7]

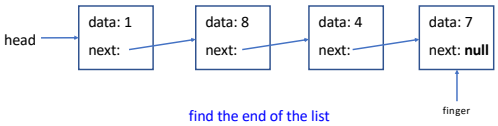


How can we add a value to the end of the list?

44

Adding to the end: add 9 to the end

[1, 8, 4, 7]



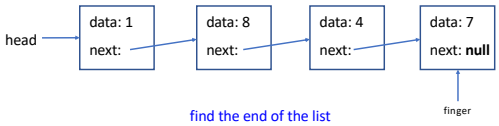
find the end of the list

How do we know we're at the end?

45

Adding to the end: add 9 to the end

[1, 8, 4, 7]



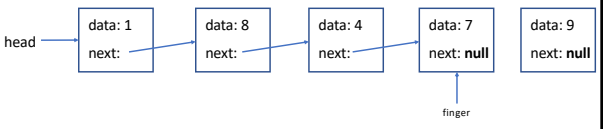
find the end of the list

finger.next() == null

46

Adding to the end: add 9 to the end

[1, 8, 4, 7]

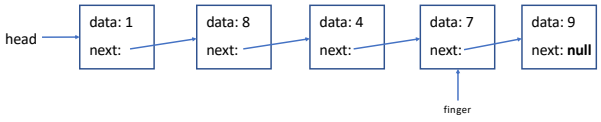


create a new node and add it to the end

47

Adding to the end: add 9 to the end

[1, 8, 4, 7]

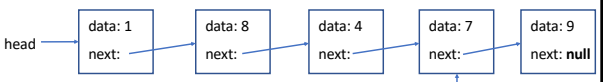


create a new node and add it to the end

48

Adding to the end: add 9 to the end

[1, 8, 4, 7]

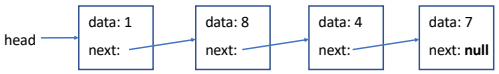



```
Node finger = head;

while( finger.next() != null ){
    finger = finger.next();
}

finger.setNext(new Node(value));
```

49

Adding to the end

[1, 8, 4, 7]




```
Node finger = head;

while( finger.next() != null ){
    finger = finger.next();
}

finger.setNext(new Node(value));
```

Any special cases we need to handle?

50

Adding to the end

[1, 8, 4, 7]

head: **null**

What would happen here?

```
Node finger = head;

while( finger.next() != null ){
    finger = finger.next();
}

finger.setNext(new Node(value));
```

51

Adding to the end

[1, 8, 4, 7]

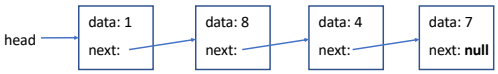
head: **null**

```
public void addLast(E value){
    if( head == null ){
        head = new Node(value);
    }else{
        Node finger = head;

        while( finger.next() != null ){
            finger = finger.next();
        }

        finger.setNext(new Node(value));
    }
}
```

52

Removing a value

[1, 8, 4, 7]

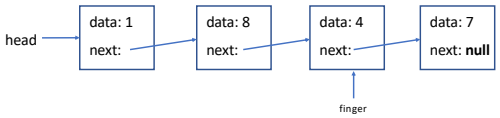


How can we we remove a value from the list?

53

Removing a value: remove 4

[1, 8, 4, 7]

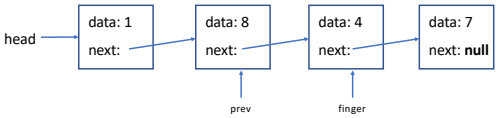


Find it

54

Removing a value: remove 4

[1, 8, 4, 7]

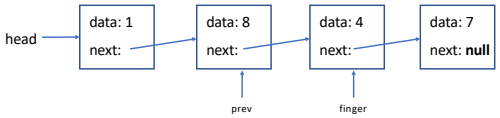


Need the node before it too!

55

Removing a value: remove 4

[1, 8, 4, 7]

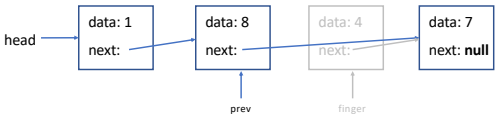


splice it out

56

Removing a value: remove 4

[1, 8, 4, 7]

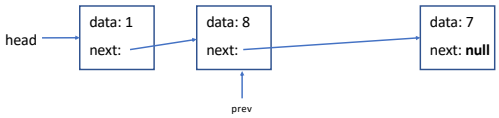


splice it out

57

Removing a value: remove 4

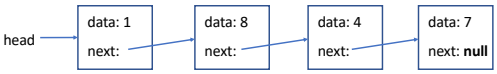
[1, 8, 4, 7]



splice it out

58

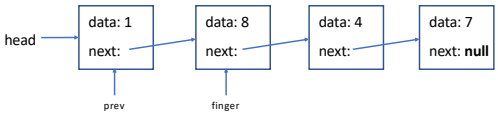
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

59

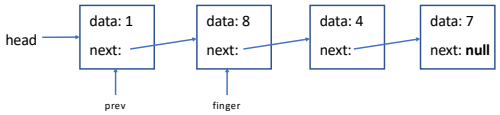
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

60

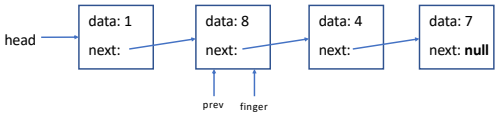
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

61

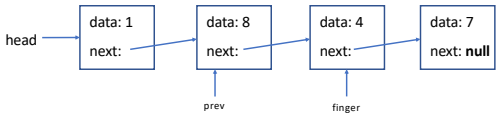
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

62

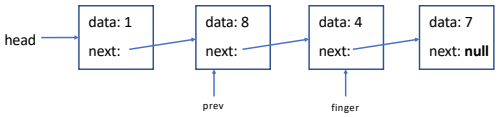
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

63

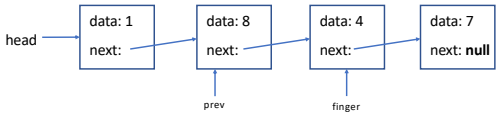
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

64

Removing a value

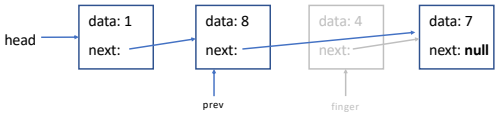



```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

When would finger be null?

65

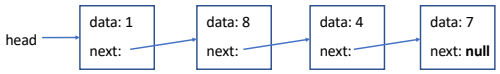
Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

66

Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

Any special cases we need to handle?

67

Removing a value

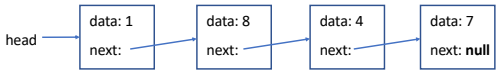
head: **null**

What would happen here?

```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

68

Removing a value

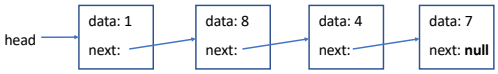



```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

Any *other* special cases we need to handle?

69

Removing a value

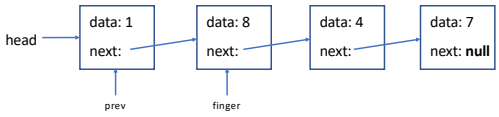



```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

Delete 1. Does it work?

70

Removing a value




```
Node finger = head.next();
Node prev = head;

while (finger != null && !finger.value().equals(value)){
    prev = finger;
    finger = finger.next();
}

if( finger != null ){
    prev.setNext(finger.next());
}
```

Delete 1. Does it work?

71

Removing a value

```
public void remove(E value){
    if( head != null ) {
        if( head.value().equals(value) ){
            head = head.next();
        }else{
            Node finger = head.next();
            Node prev = head;

            while (finger != null && !finger.value().equals(value)){
                prev = finger;
                finger = finger.next();
            }

            if( finger != null ){
                prev.setNext(finger.next());
            }
        }
    }
}
```

72

Removing a value

```
public void remove(E value){
    if( head != null ) {
        if( head.value().equals(value) ){
            head = head.next();
        }else{
            Node finger = head.next();
            Node prev = head;

            while (finger != null && !finger.value().equals(value)){
                prev = finger;
                finger = finger.next();
            }

            if( finger != null ){
                prev.setNext(finger.next());
            }
        }
    }
}
```

handle empty list

removing first element

all other elements

73

Linked lists: fast or slow?

	Linked list
add to end	
add to front	
insert at index	
contains	
get	
set	
remove	
remove from end	
remove from front	
size	

74

Linked lists: fast or slow?

	Singly linked list
add to end	slow
add to front	fast
insert at index	slow
contains	slow
get	slow
set	slow
remove	slow
remove from end	slow
remove from front	fast
size	fast

fast = O(1)
slow = O(n)

75

Linked lists: fast or slow?

	ArrayList	Singly linked list
add to end		slow
add to front		fast
insert at index		slow
contains		slow
get		slow
set		slow
remove		slow
remove from end		slow
remove from front		fast
size		fast

How does this compare to ArrayLists?

76

Linked lists: fast or slow?

	ArrayList	Singly linked list
add to end	fast (amortized)	slow
add to front	slow	fast
insert at index	slow	slow
contains	slow	slow
get	fast	slow
set	fast	slow
remove	slow	slow
remove from end	fast	slow
remove from front	slow	fast
size	fast	fast

77

Linked lists: fast or slow?

	ArrayList	Singly linked list
add to end	fast (amortized)	slow
add to front	slow	fast
insert at index	slow	slow
contains	slow	slow
get	fast	slow
set	fast	slow
remove	slow	slow
remove from end	fast	slow
remove from front	slow	fast
size	fast	fast

78

Linked lists: fast or slow?

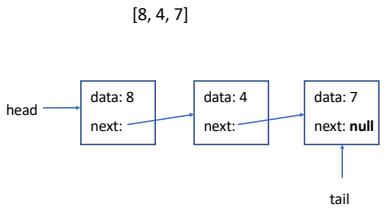
	Singly linked list
add to end	slow
add to front	fast
insert at index	slow
contains	slow
get	slow
set	slow
remove	slow
remove from end	slow
remove from front	fast
size	fast

Can we make any of these faster?

79

Adding a tail reference

	Singly linked list
add to end	slow
add to front	fast
insert at index	slow
contains	slow
get	slow
set	slow
remove	slow
remove from end	slow
remove from front	fast
size	fast



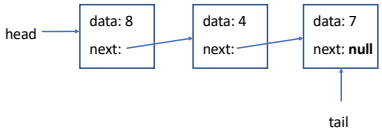
How does this help us?

80

Adding a tail reference

	Singly linked list	Singly linked list (tail)
add to end	slow	fast
add to front	fast	fast
insert at index	slow	slow
contains	slow	slow
get	slow	slow
set	slow	slow
remove	slow	slow
remove from end	slow	slow
remove from front	fast	fast
size	fast	fast

[8, 4, 7]

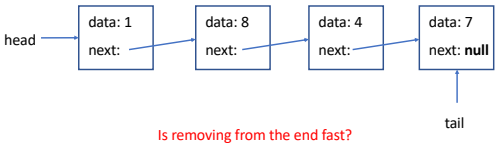


Add to the end quickly!

81

Adding a tail reference

[1, 8, 4, 7]

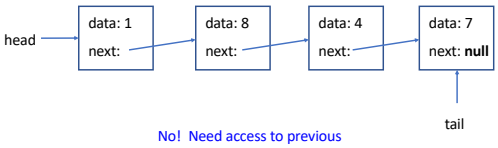


Is removing from the end fast?

82

Adding a tail reference

[1, 8, 4, 7]

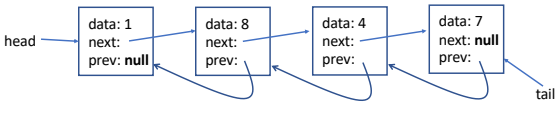


No! Need access to previous

83

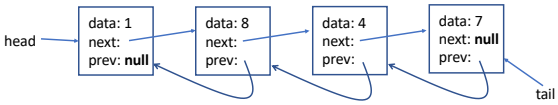
Doubly linked list visualized

Keep track of both the next *and* previous node in the list



84

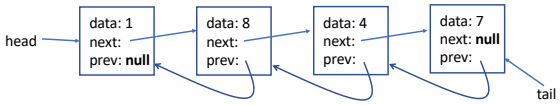
Doubly linked list visualized



Is removing from the end fast?

85

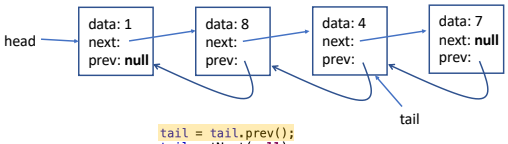
Doubly linked list visualized




```
tail = tail.prev();
tail.setNext(null);
```

86

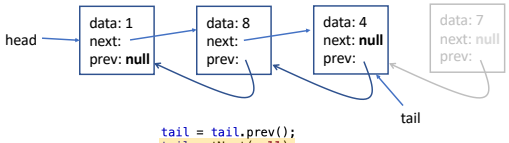
Doubly linked list visualized




```
tail = tail.prev();
tail.setNext(null);
```

87

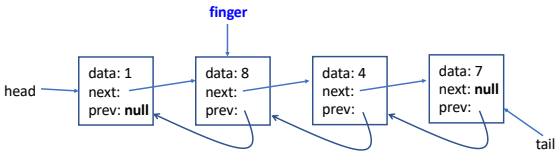
Doubly linked list visualized




```
tail = tail.prev();
tail.setNext(null);
```

88

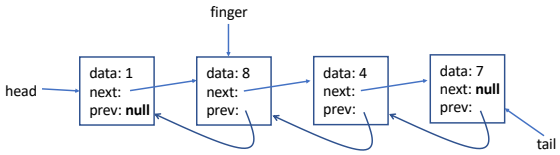
Doubly linked list visualized



Could I remove a node if I had a reference to it?

89

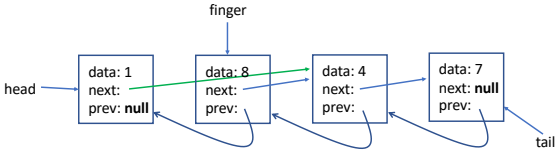
Doubly linked list visualized




```
finger.prev().setNext(finger.next());
finger.next().setPrev(finger.prev());
```

90

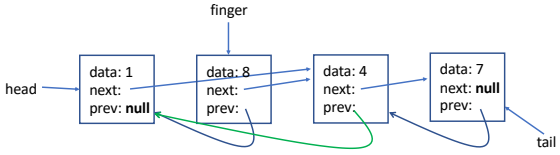
Doubly linked list visualized




```
finger.prev().setNext(finger.next());
finger.next().setPrev(finger.prev());
```

91

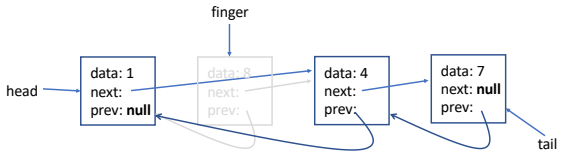
Doubly linked list visualized




```
finger.prev().setNext(finger.next());
finger.next().setPrev(finger.prev());
```

92

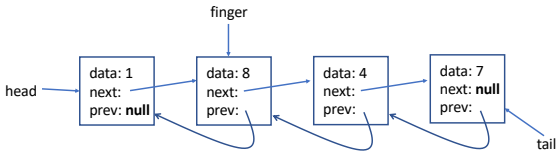
Doubly linked list visualized




```
finger.prev().setNext(finger.next());
finger.next().setPrev(finger.prev());
```

93

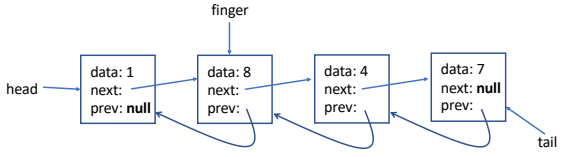
Doubly linked list visualized



How does this help us? We still have to find the node?

94

Doubly linked list visualized



If we're iterating over the data already
Other scenarios (see upcoming assignment ☺)

95

List performance

	ArrayList	Singly linked list	Singly linked list (tail)	Doubly linked list
add to end	fast (amortized)	slow	fast	
add to front	slow	fast	fast	
insert at index	slow	slow	slow	
contains	slow	slow	slow	
get	fast	slow	slow	
set	fast	slow	slow	
remove	slow	slow	slow	
remove from end	fast	slow	slow	
remove from front	slow	fast	fast	
size	fast	fast	fast	

Which of these change with doubly linked lists?

96

List performance

	ArrayList	Singly linked list	Singly linked list (tail)	Doubly linked list
add to end	fast (amortized)	slow	fast	fast
add to front	slow	fast	fast	fast
insert at index	slow	slow	slow	slow
contains	slow	slow	slow	slow
get	fast	slow	slow	slow
set	fast	slow	slow	slow
remove	slow	slow	slow	slow*
remove from end	fast	slow	slow	fast
remove from front	slow	fast	fast	fast
size	fast	fast	fast	fast

97

List performance

	ArrayList	Singly linked list	Singly linked list (tail)	Doubly linked list
add to end	fast (amortized)	slow	fast	fast
add to front	slow	fast	fast	fast
insert at index	slow	slow	slow	slow
contains	slow	slow	slow	slow
get	fast	slow	slow	slow
set	fast	slow	slow	slow
remove	slow	slow	slow	slow*
remove from end	fast	slow	slow	fast
remove from front	slow	fast	fast	fast
size	fast	fast	fast	fast

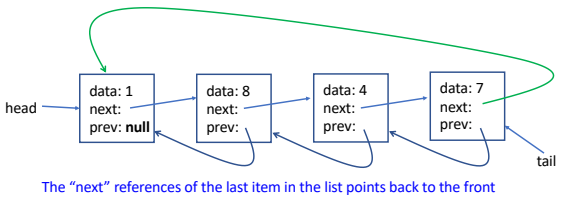
98

Java implementation

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

99

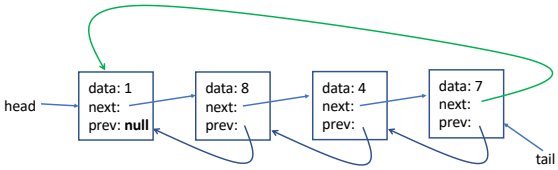
Circularly linked list visualized



The "next" references of the last item in the list points back to the front

100

Circularly linked list visualized

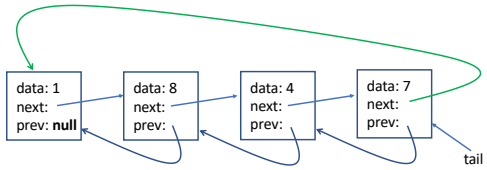


The "next" references of the last item in the list points back to the front

Any advantage of this?

101

Circularly linked list visualized



The "next" references of the last item in the list points back to the front

Can get rid of head! head = tail.next()

102