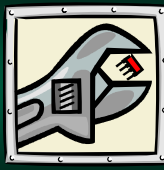


Linked List Data Structure

Part 2

1



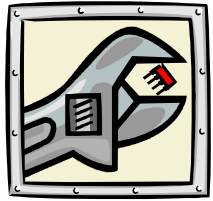
Data Structures

Return to CSC 15 and CSC 20

2

Data Structures

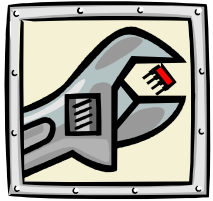
- Arrays and linked-lists are both examples of *data structures*
- These are different *techniques* of storing and organizing data
- In other words, this is *how* data is stored



3

Data Structures

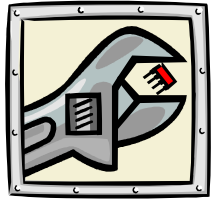
- Depending on *how* data is accessed, some data structures can either excel and falter
- This is true of both arrays and linked lists




4

Data Structures

- We will do a quick review of arrays and linked lists
- There are more data structures than these two
- We will cover them this semester – some which have *incredible* in features



5




Array Data Structure

Hidden math = easy code

6

Array Data Structure

- The array data structure is found in practically every programming language
- This is also one of the fundamental ways data is stored in memory




Fall 2024 Sacramento State - Oak - CS 130 7

7

Behind the Scenes...

- Arrays are just **continuous** blocks of memory containing multiple instances of the same type
- Since the instances are continuous, values can be accessed randomly in **O(1)**



Fall 2024 Sacramento State - Oak - CS 130 8

8

Array Math Example: 64-bit int

- Let's assume the array starts at address **2000**
- Each array element will take 8 bytes (for 64-bit integers)
- Array elements are stored continuous

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353419645000000

Fall 2024 Sacramento State - Oak - CS 130 9

9

Array Math Example: 64-bit int

- array[0]** is 2000
- array[1]** is 2008
- array[2]** is 2016
- array[3]** is 2024
- array[4]** is 2032
- etc...

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353419645000000

Fall 2024 Sacramento State - Oak - CS 130 10

10

Behind the Scenes...

- So, when an array element is read, internally, a mathematical equation is used
- It uses the start array, the array index, and the size of each element

```
start + (index × element_size)
```

Fall 2024 Sacramento State - Oak - CS 130 11

11

Behind the Scenes...

- This is why the C Programming Language uses zero as the first array element*
- If zero is used with this formula, it gets the start of the array


```
start + (index × element_size)
```

Fall 2024 Sacramento State - Oak - CS 130 12

12

Auxiliary Storage in arrays


- Also, because elements are calculated, there is **no** extra storage overhead based on the array size
- So, the *auxiliary storage* overhead is **$O(1)$**



Fall 2024 Sacramento State - CS&E - CS130 13

13

Resizing Arrays




- A *dynamically allocated array* (aka *dynamic array*) is resized anytime an object is added or removed
- Because arrays require **all** elements to be stored continuously...

Fall 2024 Sacramento State - CS&E - CS130 14

14

Resizing Arrays

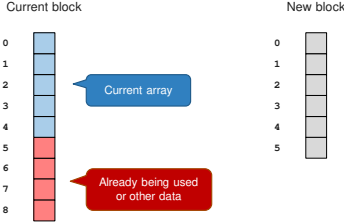


- ...the old block of memory (old array) needs to be **copied** to a new one
- This is extremely costly in both time and resources

Fall 2024 Sacramento State - CS&E - CS130 15

15

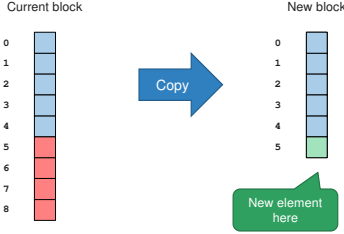
Arrays in Memory



Fall 2024 Sacramento State - CS&E - CS130 16

16


Copy Values to New Block



Fall 2024 Sacramento State - CS&E - CS130 17

17

Resizing Arrays is $O(n)$



- While reading / writing elements takes only $O(1)$...
- ... every time an array is resized, it will require **$O(n)$** time to copy the old array to the new one

Fall 2024 Sacramento State - CS&E - CS130 18

18

Fixed-Sized Arrays

- Arrays can also have a fixed sized called a *capacity*
- The array is **never** resized and often only partially filled
- Also known as:
 - *fixed array*
 - *partially filled array*



Fall 2024

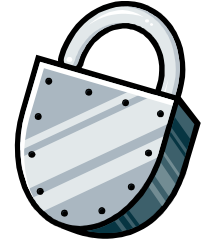
Sacramento State - Oak - CS110

19

19

Fixed-Sized Arrays

- An "end" index is maintained
- This type of array overcomes the $O(n)$ nature of dynamic arrays
- But a cost – it has a limit that cannot be exceeded



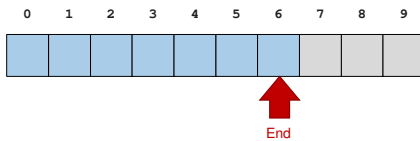
Fall 2024

Sacramento State - Oak - CS110

20

20

Fixed-Size Array



Fall 2024

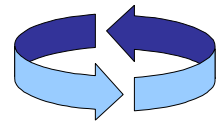
Sacramento State - Oak - CS110

21

21

Fixed-Size Wrapping Around

- Sometimes, you might need an array that wraps
- These are useful if both the first and last items can be removed
- ... or older items can be discarded if space is needed



Fall 2024

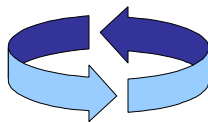
Sacramento State - Oak - CS110

22

22

Fixed-Size Wrapping Around

- In addition to a "end" index, a "start" index is maintained
- Once the end of the array is reached, the array "wraps" to index 0
- ... and continues until end is reached



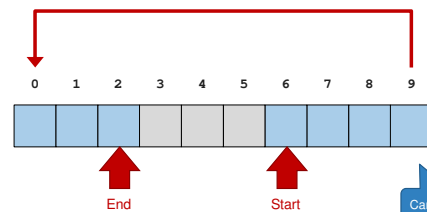
Fall 2024

Sacramento State - Oak - CS110

23

23

Fixed-Size Array



Fall 2024

Sacramento State - Oak - CS110

24

24




Linked Lists

Chain of Data

25

Linked Lists

- The array (and the ArrayList) are just two, of many, ways of storing a collection
- *Linked lists* uses a series of "linked" instances to store a collection



Fall 2024
Sacramento State - Oak - CS110
26

26

How Does It Work?

- Since a variable can contain either an instance reference or null, we can do something quite clever
- An instance can contain a reference to another instance – *of the same class*
- This creates a chain of connected instances.
- It ends when the "link" is null

Fall 2024
Sacramento State - Oak - CS110
27

27

How Does That Work?

```

class Book
{
    public String name;
    public Book sequel;
}
    
```

A reference to another Book instance

Fall 2024
Sacramento State - Oak - CS110
28

28

Chain of Books


<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #004a33; color: white;"> <th style="width: 50%;">name</th> <th style="width: 50%;">sequel</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">The Fellowship of the Ring</td> <td style="padding: 2px;">→</td> </tr> </tbody> </table>	name	sequel	The Fellowship of the Ring	→	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #004a33; color: white;"> <th style="width: 50%;">name</th> <th style="width: 50%;">sequel</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">The Two Towers</td> <td style="padding: 2px;">→</td> </tr> </tbody> </table>	name	sequel	The Two Towers	→	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #004a33; color: white;"> <th style="width: 50%;">name</th> <th style="width: 50%;">sequel</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">Return of the King</td> <td style="padding: 2px;">null</td> </tr> </tbody> </table>	name	sequel	Return of the King	null
name	sequel													
The Fellowship of the Ring	→													
name	sequel													
The Two Towers	→													
name	sequel													
Return of the King	null													

Fall 2024
Sacramento State - Oak - CS110
29

29

Chain of Instances

- Notice that the last example is essentially storing Strings
- Can we store other things?
- Yes! This is very simple approach to store any type of data



Fall 2024
Sacramento State - Oak - CS110
30

30

Chain of Instances

- Each link in our chain, that stores a piece of information, is called a *Node*
- The definition of a Node is extremely simple: data and a link to the next node



31

Generic Node Class

```
public class Node
{
    public Object data;
    public Node next;
}
```

We can make this another type

32

Creating a List (not well, though)

```
Node list = new Node();
list.data = "rat";
list.next = new Node();
list.next.data = "owl";
list.next.next = new Node();
list.next.next.data = "cat";
list.next.next.next = null;
```

33

Constructors Will Help

```
public Node(Object initData, Node initNext)
{
    this.data = initData;
    this.next = initNext;
}
public Node(Object initData)
{
    this.data = initData;
    this.next = null;
}
```

34

Constructors Do Help

- Constructors do help
- Though it is still a tad hard to read

```
Node list = new Node("Rat", new Node("Owl", new Node("Cat")));
```


35

We Could Also Do This...

```
Node rat = new Node("Rat");
Node owl = new Node("Owl");
Node cat = new Node("Cat");
rat.next = owl;
owl.next = cat;
```

36

Traversing a Linked List

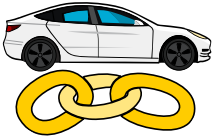


Chain of Data

37

Traversing a Linked List

- Unlike arrays, where the element can be found using a calculation, linked-lists require the list to be traversed
- This is typically done using a while loop and variable representing the current node



38

Node for Integers

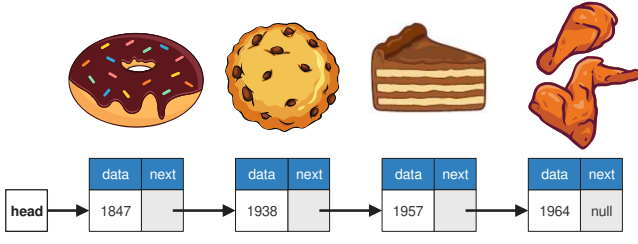
```

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

Let's use int's for now

39

Let's Try This List



40

While Loop – Follow the Links

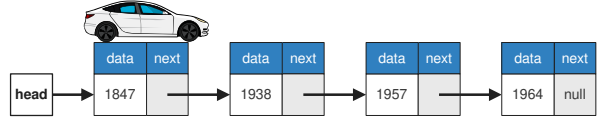
```

current = head;

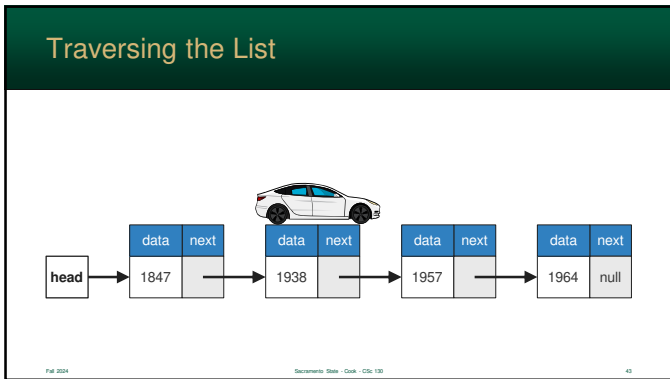
while (current != null)
{
    System.out.println(current.data);
    current = current.next; //Go to next node
}
    
```

41

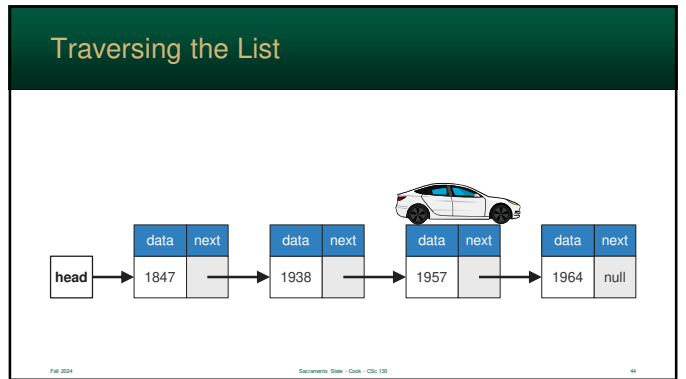
Traversing the List



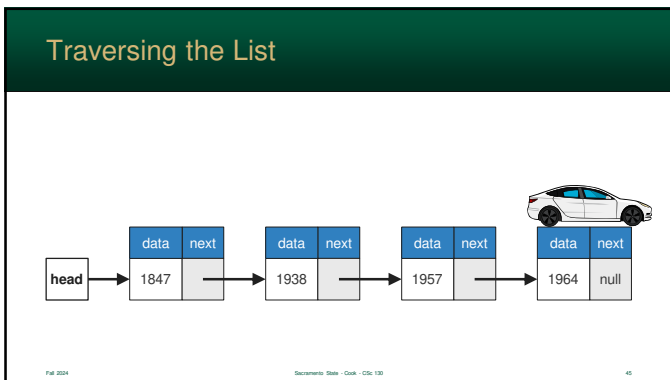
42



43



44



45

Adding to Linked Lists

Chain of Data

46

Adding to Linked Lists

- Adding to Linked Lists is easy to do, but must be done with considerable care
- The links (references) need to be updated in a specific order
- ... or a link will be lost

Fall 2024 Sacramento State - Oak - CS130 47

47

Adding to Linked Lists

- The first item in a linked list is referred to as the **Head** (alternatively *Front*)
- The last item, in which the next field is null, is called the *Tail*

Fall 2024 Sacramento State - Oak - CS130 48

48

Adding to Linked Lists

- In this section, we will add a new node to the front, middle, and end of a linked list
- Most of these actions require just two steps



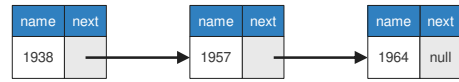
Fall 2024

Sacramento State - CS&E - CS110

49

49

Let's Assume We Have This List



Fall 2024

Sacramento State - CS&E - CS110

50

50

Adding to the Tail of a List



1. Link the tail node (who's next field is null) to the newly added node
2. If a reference to the tail is being maintained, it is linked to the newly added node

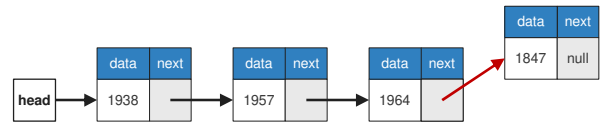
Fall 2024

Sacramento State - CS&E - CS110

51

51

Add Tail: 1. Link Tail to the New Node



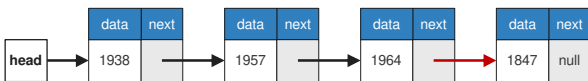
Fall 2024

Sacramento State - CS&E - CS110

52

52

Add Tail: Resulting List



Fall 2024

Sacramento State - CS&E - CS110

53

53

Adding to the Tail of a List

```
// add is the new node
// tail is the last node in the list

tail.next = add;
```

Fall 2024

Sacramento State - CS&E - CS110

54

54

Adding to the Head of a List



1. The newly added node is linked to the head of the list
2. The head is then linked to the newly added node

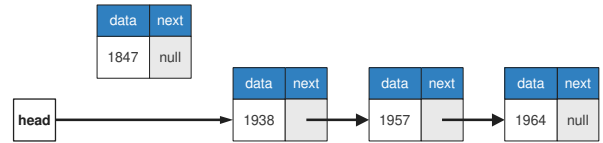
Fall 2024

Sacramento State - CS&E - CS110

55

55

Adding to the Head of a List



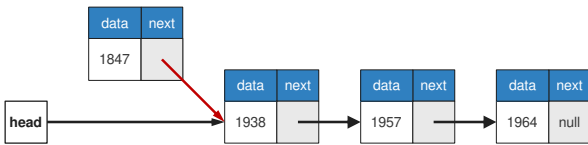
Fall 2024

Sacramento State - CS&E - CS110

56

56

Add Head: 1. Link Node to Head's Reference



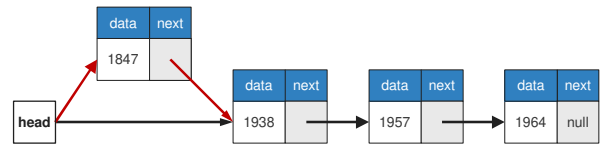
Fall 2024

Sacramento State - CS&E - CS110

57

57

Add Head: 2. Set Head Reference to the Node



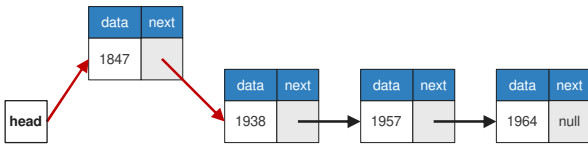
Fall 2024

Sacramento State - CS&E - CS110

58

58

Add Head: 2. Set Head Reference to the Node



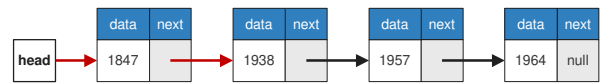
Fall 2024

Sacramento State - CS&E - CS110

59

59

Add Head: Resulting List



Fall 2024

Sacramento State - CS&E - CS110

60

60

Adding to the Head of a List

```


// add is the new node
// head is the first node in the list

add.next = head;
head = add;

```

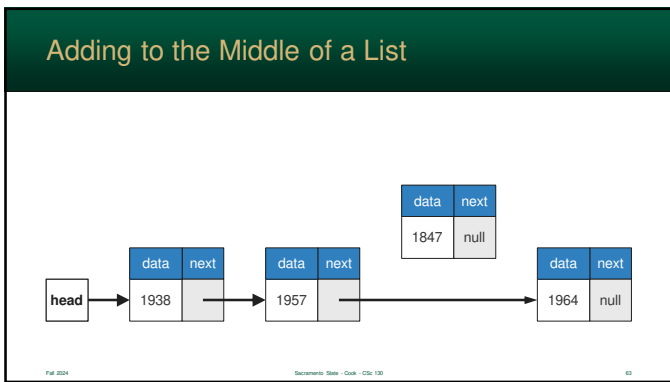
61

Adding to the Middle of a List

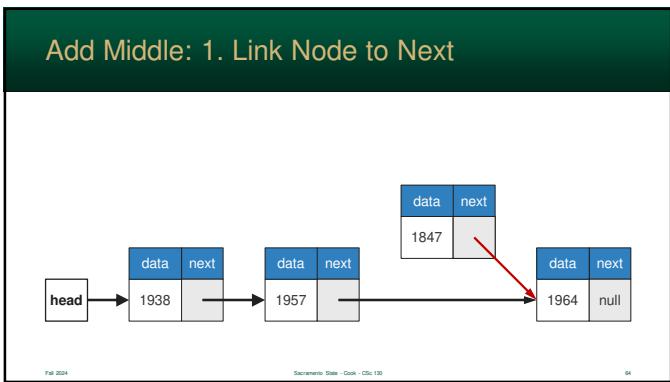


1. The new node is linked to target of the previous node (before where we want to insert)
2. The previous node is then linked to the new node

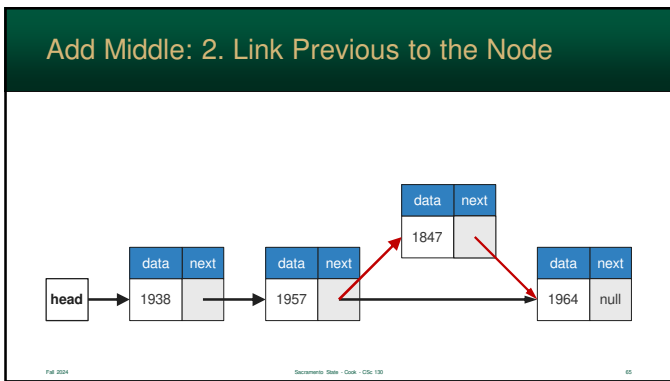
62



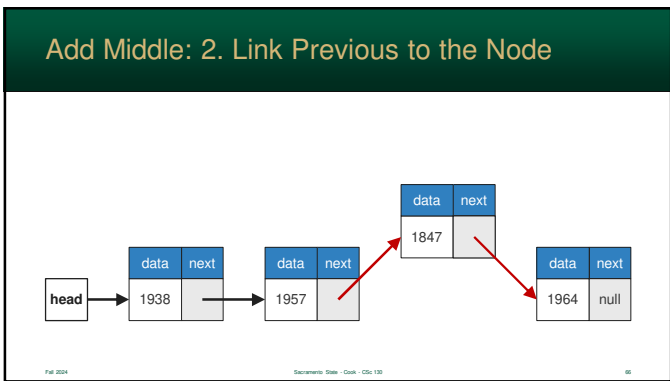
63



64

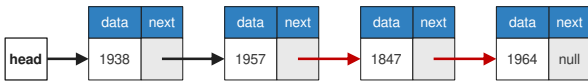


65



66

Add Middle: Resulting List



67

Adding to the Middle of a List

```
// add is the new node
// prev is the node before where
// add is to be inserted

add.next = prev.next;
prev.next = add;
```

68

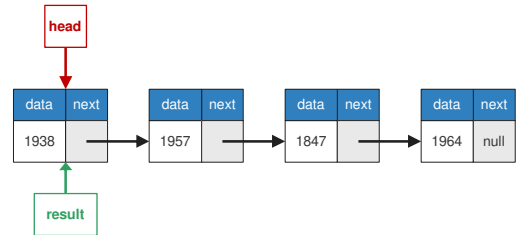
Removing The Head Node



1. Save Link to the Old Head
2. Update the Head Reference to the Head's next link
3. Remove the link from the old head to the new head

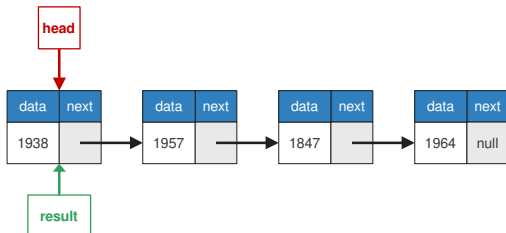
69

Remove Head: 1. Save Link to the Old Head



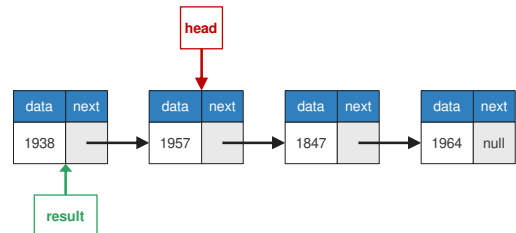
70

Remove Head: 2. Update the Head Reference



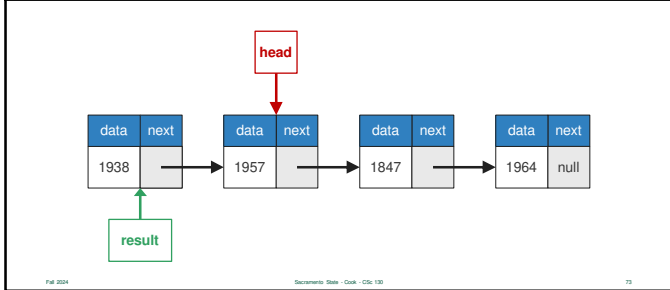
71

Remove Head: 2. Update the Head Reference



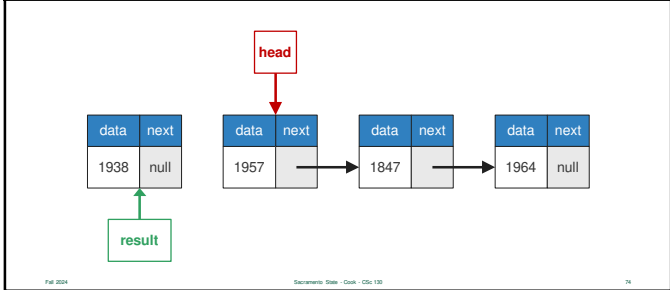
72

Remove Head: 3. Remove the Link



73

Remove Head: Complete



74

Remove Head

```
// Save a reference to the head
result = head;

//Set head to the head's next link
head = head.next;

//Remove link between old head and new head
result.next = null;
```

Exactly the same as a singly linked list

75

Maintaining a Tail Node

One should keep track the caboose

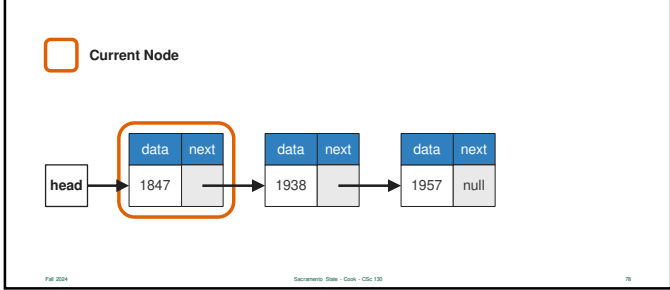
76

Head and Tail Nodes

- Linked lists maintain a link to the head node
 - Often, in well-written linked lists, a link to the tail node is also maintained
 - It is far more efficient
-

77

Adding to the End



78

Adding to the End

□ Current Node

```

graph LR
    head --> n1["data: 1847, next: 1938"]
    n1 --> n2["data: 1938, next: 1957"]
    n2 --> n3["data: 1957, next: null"]
  
```

Fall 2024 Sacramento State - Oak - CS 130 79

79

Adding to the End

□ Current Node

Found the last node in the list. next == null

```

graph LR
    head --> n1["data: 1847, next: 1938"]
    n1 --> n2["data: 1938, next: 1957"]
    n2 --> n3["data: 1957, next: null"]
  
```

Fall 2024 Sacramento State - Oak - CS 130 80

80

Adding to the End

□ Current Node

The new node is added

```

graph LR
    head --> n1["data: 1847, next: 1938"]
    n1 --> n2["data: 1938, next: 1957"]
    n2 --> n3["data: 1957, next: 1964"]
    n3 --> n4["data: 1964, next: null"]
  
```

Fall 2024 Sacramento State - Oak - CS 130 81

81

While Loop – Follow the Links

```

current = head;
while (current.next != null)
{
    current = current.next; //Go to next node
}

// Current is now the tail. Link tail to new node
current.next = add;
  
```

Fall 2024 Sacramento State - Oak - CS 130 82

82

So, that took awhile...

- Notice that, to get the tail now, we had to write loop to traverse all the nodes
- If we *knew* where the tail was beforehand, we wouldn't need a loop

Fall 2024 Sacramento State - Oak - CS 130 83

83

Adding to the End (with a tail)

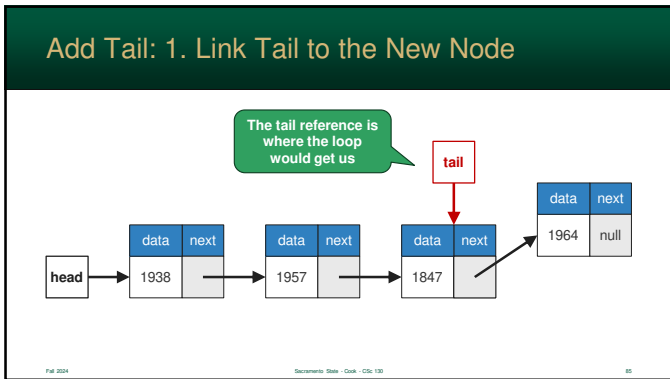
The tail reference is where the loop would get us

```

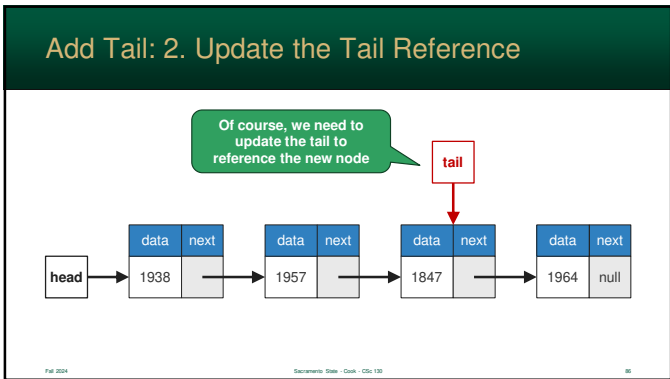
graph LR
    head --> n1["data: 1938, next: 1957"]
    n1 --> n2["data: 1957, next: 1847"]
    n2 --> n3["data: 1847, next: null"]
    tail --> n3
  
```

Fall 2024 Sacramento State - Oak - CS 130 84

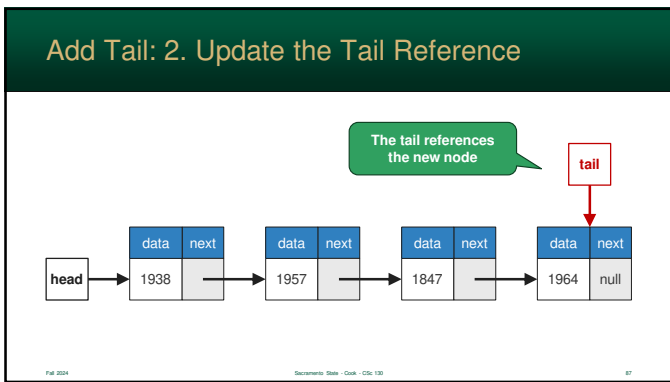
84



85



86



87

Adding to the end – with a Tail Node

```
tail.next = add;
tail = add;
```

88

Finding Second-to-Last

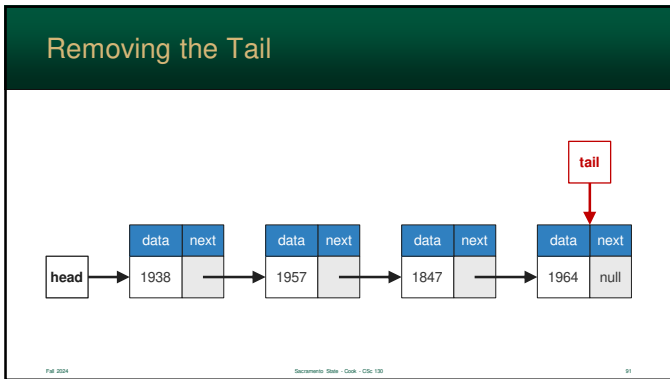
- As we noticed with the Singly-Linked list, finding the last item (to add at the end) required a loop
- ... or was immediate if we maintained a tail node reference

89

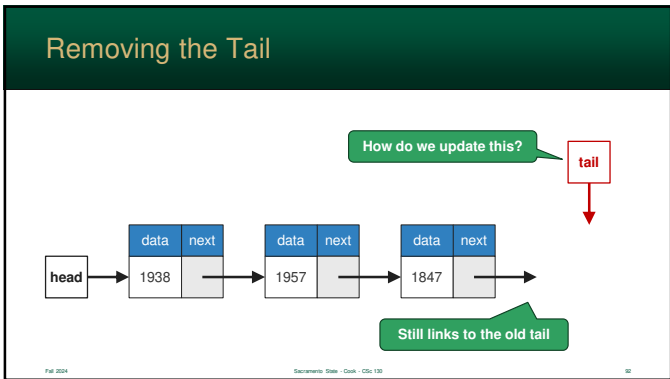
Finding Second-to-Last

- But, how do we remove the last item?
- We can find the last item immediately, but that would make second-to-last the new tail

90



91



92

Remove Last –Linked List

```

current = head;
while (current != null)
{
    if (current.next == last)
    {
        nextToLast = current;
    }
    current = current.next; //Go to next node
}

// Now remove last
result = last;
nextToLast.next = null; //Remove the old last
last = nextToLast;
    
```

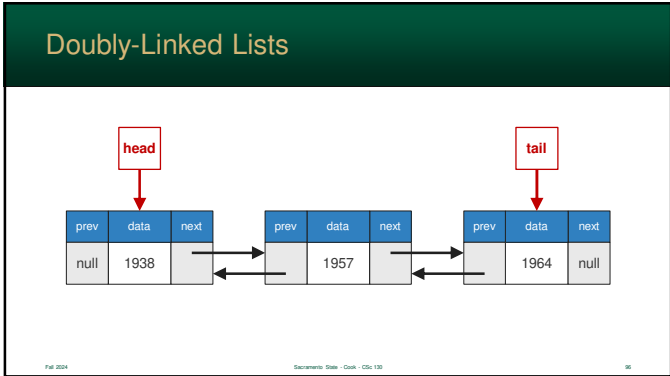
93



94

- ### Doubly-Linked Lists
- Another variation of a linked list is the *doubly-linked list*
 - As the name implies, there are two sets of links – one that points to the next node and one that points to the previous
-

95



96

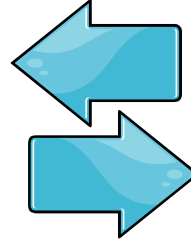
Doubly-Linked List Node

```
public class Node
{
    public Object data;
    public Node prev;
    public Node next;
}
```

Sometimes called last

97

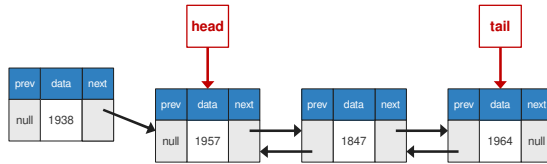
Doubly Linked List: Add to the Head



1. Link New Node to the Head
2. Link Head Back to the New Node
3. Update the Head Reference to new node

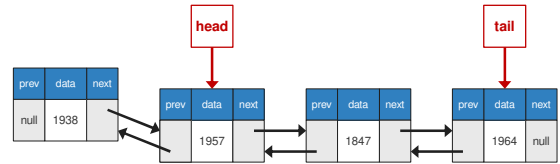
98

Add Head: 1. Link New Node to the Head



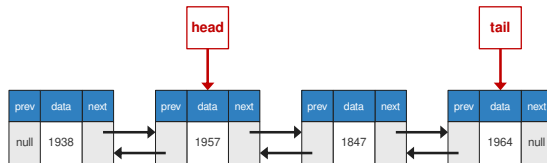
99

Add Head: 2. Link Head Back to the New Node



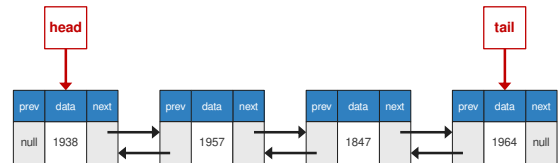
100

Add Head: 3. Update the Head Reference



101

Add Head: 3. Update the Head Reference



102

Add Head – Doubly Linked List

```
// Link the new node to old head
add.next = head;

// Link the old head back to the new node
head.prev = add;

//Set head to the new node
head = add;
```

Also may be wise to check if the head == null

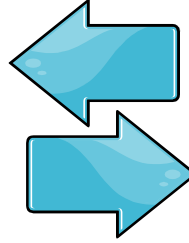
Fall 2024

Sacramento State - Oak - CS 130

103

103

Doubly Linked List: Add to the Tail



1. Link Tail to the New Node
2. Link the New Node to the Old Tail
3. Update the Tail Reference to the New Node

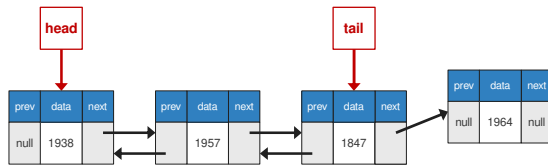
Fall 2024

Sacramento State - Oak - CS 130

104

104

Add Tail: 1. Link Tail to New Node



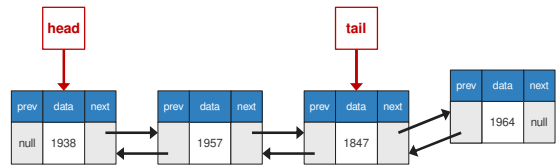
Fall 2024

Sacramento State - Oak - CS 130

105

105

Add Tail: 2. Link the New Node to the Old Tail



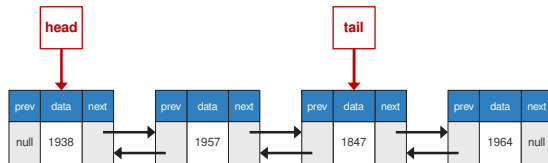
Fall 2024

Sacramento State - Oak - CS 130

106

106

Add Tail: 3. Update the Tail Reference



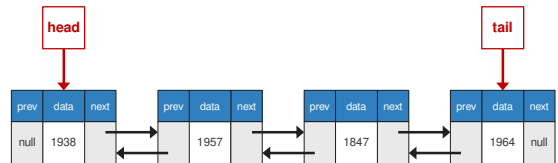
Fall 2024

Sacramento State - Oak - CS 130

107

107

Add Tail: 3. Update the Tail Reference



Fall 2024

Sacramento State - Oak - CS 130

108

108

Add Tail – Doubly Linked List

```
// Link the old tail to the new node
tail.next = add;

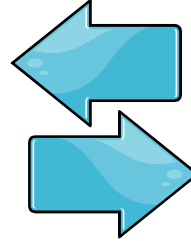
// Link new node back to the old tail
add.prev = tail;

//Set tail to the new node
tail = add;
```

Also may be wise to check if the tail == null

109

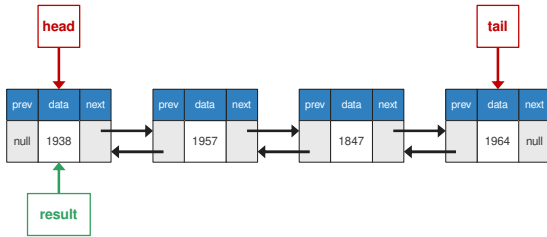
Doubly Linked List: Remove Head



1. Save Link to the Old Head
2. Update the Head Reference to the Head's next reference
3. Remove links between Old Head and New Head

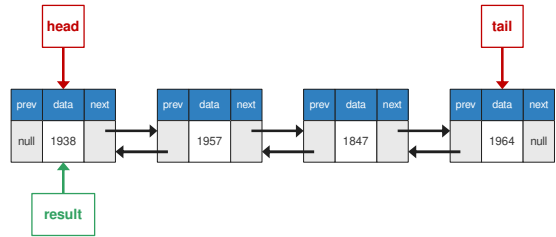
110

Remove Head: 1. Save Link to the Old Head



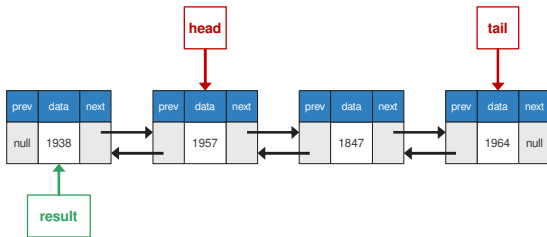
111

Remove Head: 2. Update the Head Reference



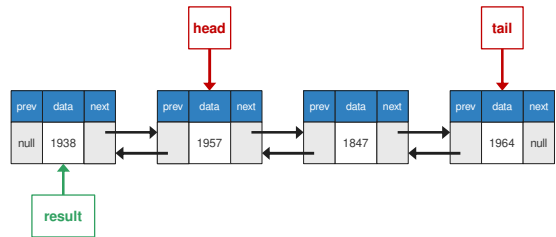
112

Remove Head: 2. Update the Head Reference



113

Remove Head: 3. Remove Links



114

Remove Head: Complete

Diagram illustrating the removal of the head node from a doubly linked list. The list consists of three nodes with data values 1938, 1957, and 1964. The head pointer points to the first node (1938). The result pointer also points to the first node. The diagram shows the head pointer moving to the second node (1957) and the first node's prev pointer becoming null.

115

Remove Head – Doubly Linked List

```

// Save a reference to the tail
result = head;

//Set head to the head's next link
head = head.next;

//Remove links between old head and new head
head.prev = null;
result.next = null;

```

Exactly the same as a singly linked list

116

Doubly Linked List: Remove Tail

1. Save Link to the Old Tail
2. Update the Tail Reference to the previous reference of the current Tail
3. Remove links between the New Tail and the Old Tail

117

Remove Tail: 1. Save Link to the Old Tail

Diagram illustrating the first step of removing the tail node: saving a reference to the old tail. The tail pointer points to the last node (1964). The result pointer also points to the last node. The diagram shows the tail pointer moving to the second-to-last node (1957).

118

Remove Tail: 2. Update the Tail Reference

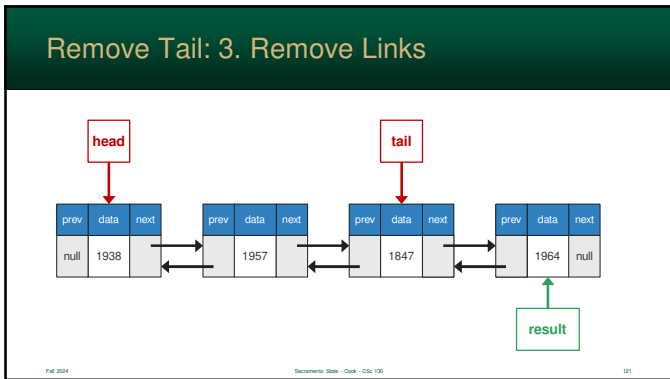
Diagram illustrating the second step of removing the tail node: updating the tail reference. The tail pointer points to the second-to-last node (1957). The diagram shows the tail pointer moving to the second-to-last node (1957).

119

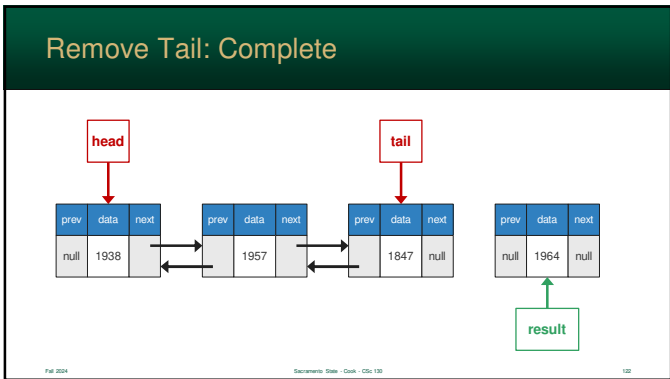
Remove Tail: 2. Update the Tail Reference

Diagram illustrating the second step of removing the tail node: updating the tail reference. The tail pointer points to the second-to-last node (1957). The diagram shows the tail pointer moving to the second-to-last node (1957).

120



121



122

Remove Tail – Doubly Linked List

```

// Save a reference to the tail
result = tail;

//Set tail to the previous of the old tail
tail = tail.prev;

//Remove links between old tail and new tail
tail.next = null;
result.prev = null;

```

123

Singly-Linked List Class

Creating a train of nodes

124

LinkedList Class

- Maintaining both a head and tail node can be a tad difficult
- So, we can place them into a LinkedList class
- Then we can write methods to add to the end (the tail) and the front (the head)

125

LinkedList Class

```

class LinkedList
{
    public Node head;
    public Node tail;
}

```

126

Linked List Class

```
public void AddTail(Node node)
{
    if (head == null) //Add first node
    {
        head = node; //Link both
        tail = node;
    }
    else
    {
        tail.next = node; //Link old tail to the new node
        tail = node; //Now the new node is the tail
    }
}
```

127

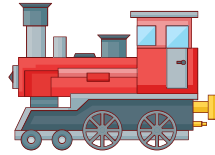
Linked List Class

```
public void AddHead(Node node)
{
    if (head == null) //Add first node
    {
        head = node; //Link both
        tail = node;
    }
    else
    {
        node.next = head; //Link new node to the current head
        head = node; //Now the new node is the head
    }
}
```

128

Linked List Class

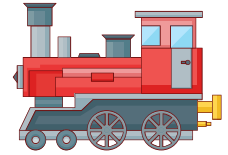
- Now that we have compensated for the head/tail being null, we can also add a method to remove the head
- But there are more cases that need to be considered



129

Linked List Class

- There are 2+ nodes (the head and tail are different)
- There is only one node (the head and tail are the same)
- There are no nodes



130

```
public Node RemoveHead()
{
    Node result;

    if (head == null)
    {
        result = null; //We could also throw an error
    }
    else if (head == tail) //Just one node. Set both head/tail to null
    {
        result = head; //...or tail - it doesn't matter here. Note, we are saving the reference in 'result'.
        head = null; //Deferenece both
        tail = null;
    }
    else //2 or more nodes.
    {
        result = head;
        head = head.next; //Link new node to the current head
    }

    return result;
}
```

131




Linked List Big-O

How Good Is This?

132

Linked List Data Structure

- Linked lists are a fundamental data structure that was covered in CSC 20
- Data is stored in a series of nodes which are connected with links




Fall 2024 Sacramento State - CS&E - CSC 130 133

133

Linked List Data Structure

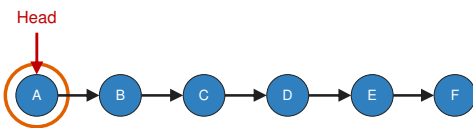
- Unlike arrays, where the element can be found using a calculation, linked-lists require the list to be traversed
- So, finding an item in a linked list requires $O(n)$



Fall 2024 Sacramento State - CS&E - CSC 130 134

134

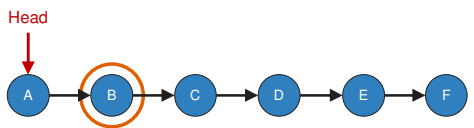
Single-Linked List – Find D



Fall 2024 Sacramento State - CS&E - CSC 130 135

135

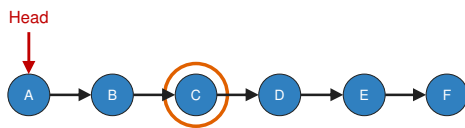
Single-Linked List – Find D



Fall 2024 Sacramento State - CS&E - CSC 130 136

136

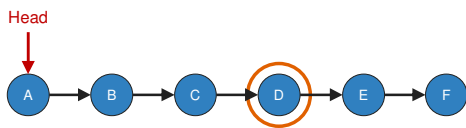
Single-Linked List – Find D



Fall 2024 Sacramento State - CS&E - CSC 130 137

137

Single-Linked List – Find D



Fall 2024 Sacramento State - CS&E - CSC 130 138

138

Head and Tail Nodes



- Linked lists maintain a link to the head node
- Often, in well-written linked lists, a link to the tail node is also maintained
- Why? It has a huge impact on time complexity

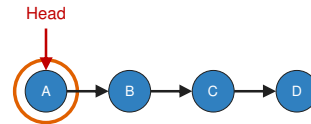
Fall 2024

Sacramento State - CS&E - CS130

139

139

Append Value – No Tail Node



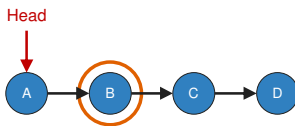
Fall 2024

Sacramento State - CS&E - CS130

140

140

Append Value – No Tail Node



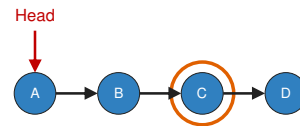
Fall 2024

Sacramento State - CS&E - CS130

141

141

Append Value – No Tail Node



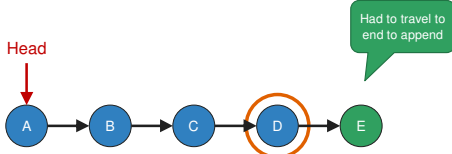
Fall 2024

Sacramento State - CS&E - CS130

142

142

Append Value – No Tail Node



Fall 2024

Sacramento State - CS&E - CS130

143

143

Head and Tail Nodes



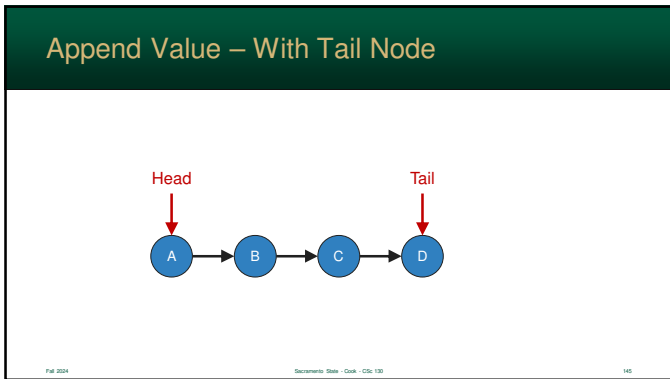
- Without a tail node, the entire list must be traversed to find the end
- This will require $O(n)$
- Adding a tail node, will decrease it to $O(1)$

Fall 2024

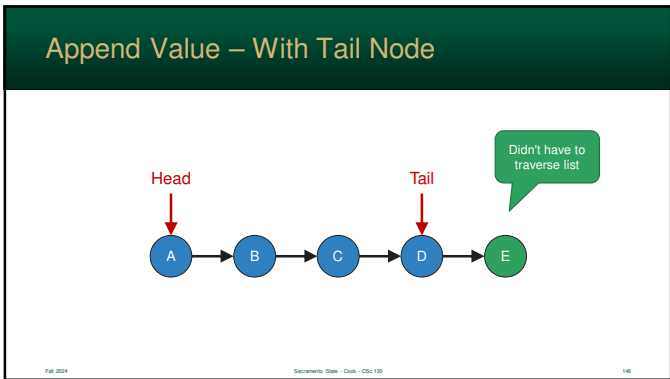
Sacramento State - CS&E - CS130

144

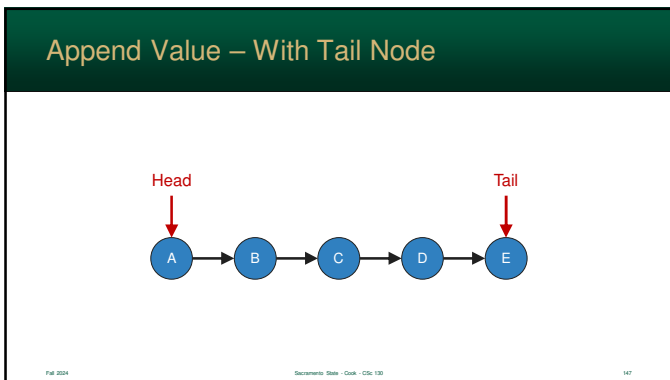
144



145



146



147

Use a Tail Node!

- Unless you are only appending nodes at the head of a linked list, maintain a tail node
- For **all** the examples used in these slides... assume the linked list has a tail node

148

Auxiliary Storage in Linked Lists

- Unlike arrays, linked lists must store the "next" links between nodes
- So, the *auxiliary storage* overhead is $O(n)$
 - ...which is usually the size of an address
 - 64-bit system → 8 bytes

149

Big-O: Test Your Might...

```

LinkedList list;
for(i = 0; i < list.Count; i++)
{
    total += list.Find(i);
}
    
```

$O(n)$ (pointing to list.Count)

$O(n)$ (pointing to list.Find(i))

$O(n^2)$ (pointing to the loop body)

150

Iterators

- To avoid accidental $O(n^2)$, major programming languages support *iterator objects*
- They store information about the current state (e.g. a node) when data is being sequentially read



Fall 2024

Srinivasan Sivas - Osh - CS101

151

151

Iterators

- Iterators maintain $O(n)$ for sequentially accessing all the list's elements
- This is the purpose of the For-Each Statement
- Notation varies greatly between languages (when they are supported)



Fall 2024

Srinivasan Sivas - Osh - CS101

152

152

Dynamic Array vs. Linked List

Operation	Dynamic Array	Linked List
Find (to read or write)	$O(1)$	$O(n)$
Insert (arbitrary)	$O(n)$	$O(n)$
Add first/last	$O(n)$	$O(1)$
Remove first/last	$O(n)$	$O(1)$
Auxiliary storage	$O(1)$	$O(n)$

Fall 2024

Srinivasan Sivas - Osh - CS101

153

153