CSCI 136
Data Structures &
Advanced Programming

Doubly Linked Lists

This Video

- Continue discussing lists w/ linked structures
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists

Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
  - The list itself

Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself

Recall: Linked List Basics

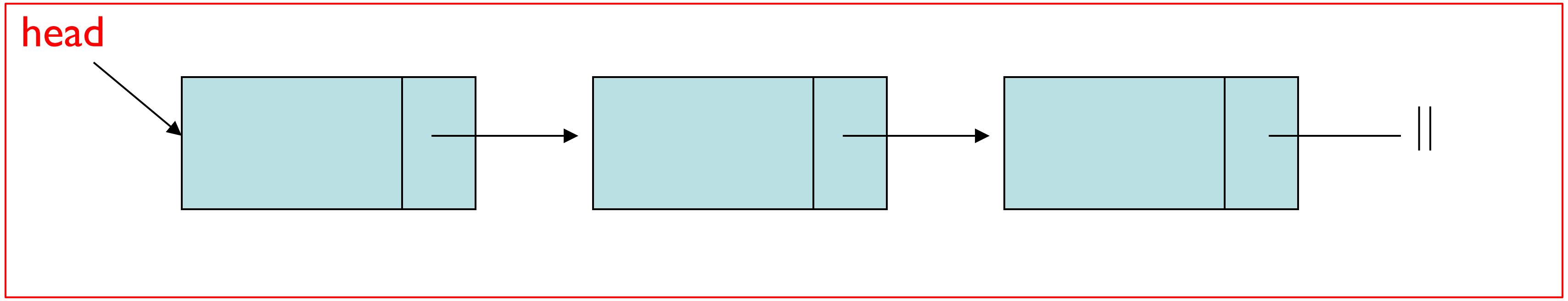
- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list

Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list
- Visualizing linked lists:

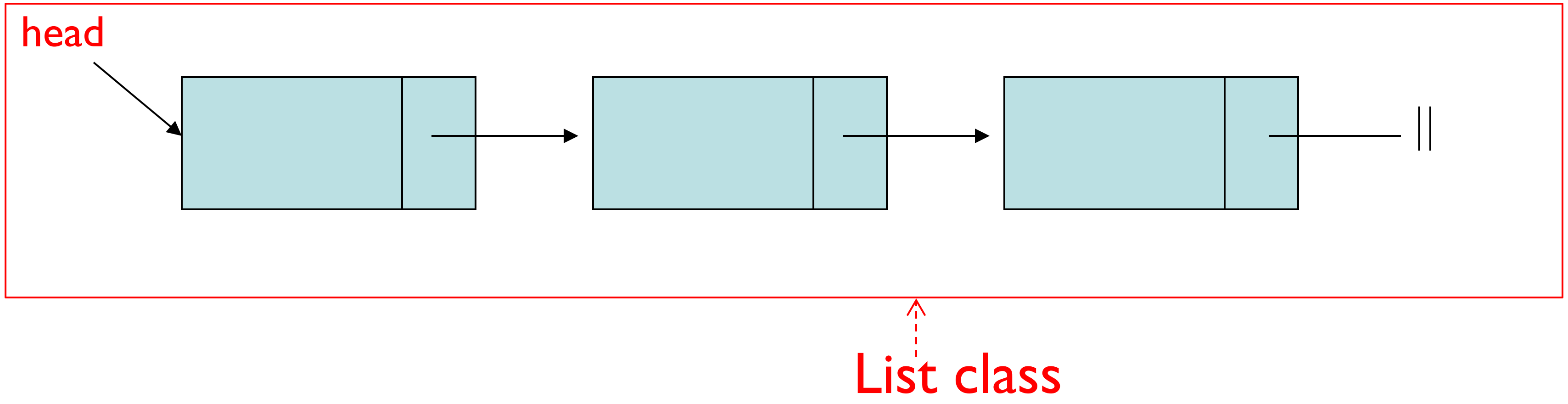
Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list
- Visualizing linked lists:



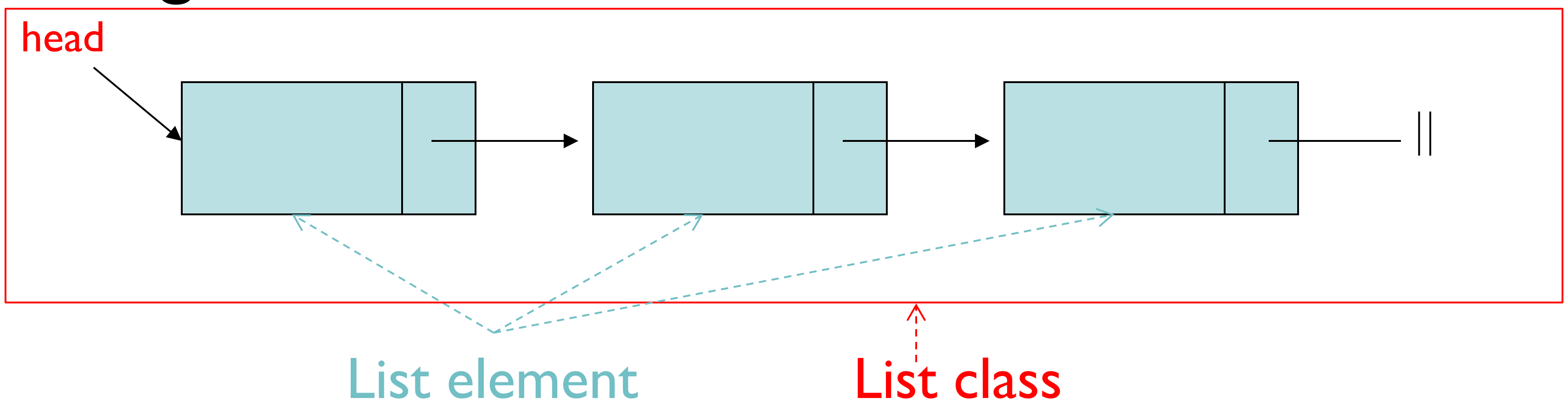
Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list
- Visualizing linked lists:



Recall: Linked List Basics

- There are two key aspects of Linked Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list
- Visualizing linked lists:



Recall: `SinglyLinkedList` Basics

Recall: `SinglyLinkedList` Basics

- `SinglyLinkedList` class has:
  - Public methods that implement `List` interface
  - Internal state/details that are hidden from the user

Recall: `SinglyLinkedList` Basics

- `SinglyLinkedList` class has:
  - Public methods that implement `List` interface
  - Internal state/details that are hidden from the user
- Each "node" has:

Recall: `SinglyLinkedList` Basics

- `SinglyLinkedList` class has:
  - Public methods that implement `List` interface
  - Internal state/details that are hidden from the user
- Each "node" has:
  - A data value

Recall: `SinglyLinkedList` Basics

- `SinglyLinkedList` class has:
  - Public methods that implement `List` interface
  - Internal state/details that are hidden from the user
- Each "node" has:
  - A data value
  - A next variable that identifies the next element in the list

Recall: `SinglyLinkedList` Basics

- `SinglyLinkedList` class has:
  - Public methods that implement `List` interface
  - Internal state/details that are hidden from the user
- Each "node" has:
  - A data value
  - A next variable that identifies the next element in the list
- The `SinglyLinkedList` class keeps a reference only to the first list element (`head`)

`SinglyLinkedList` Summary

`SinglyLinkedList` Summary

- More control over space usage than Vectors

`SinglyLinkedList` Summary

- More control over space usage than Vectors
- Easy to access the front of list: O(1)
  - Direct access to `head`: yay!

`SinglyLinkedList` Summary

- More control over space usage than Vectors
- Easy to access the front of list: O(1)
  - Direct access to `head`: yay!
- Difficult to access later elements: O(n)
  - We must always start our traversals at head
  - We must always go forward

DoublyLinkedLists

DoublyLinkedLists

- Keep reference/links in **both** directions

DoublyLinkedLists

- Keep reference/links in **both** directions
    - Can therefore traverse forwards and backwards!

DoublyLinkedLists

- Keep reference/links in **both** directions
  - Can therefore traverse forwards and backwards!

- `DoublyLinkedListNode` class's instance variables:

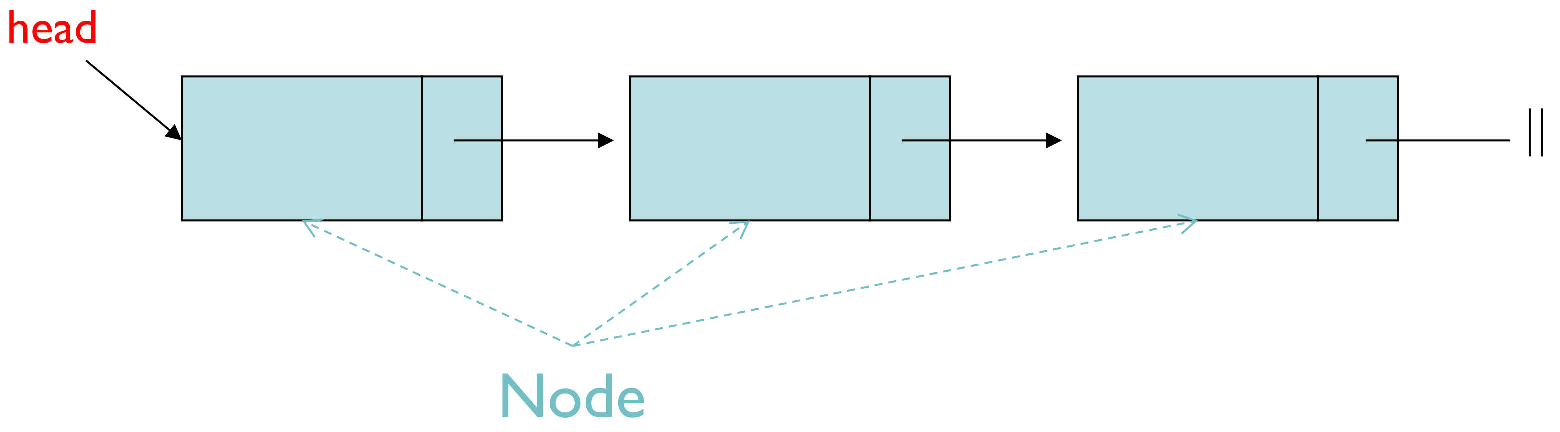
DoublyLinkedLists

- Keep reference/links in **both** directions
  - Can therefore traverse forwards and backwards!

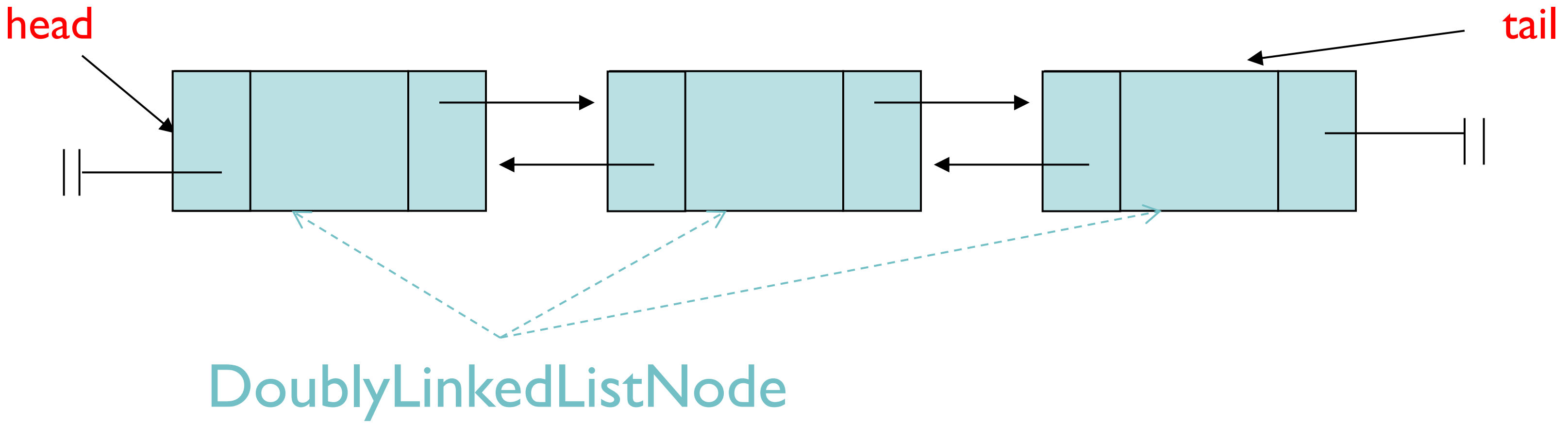
- `DoublyLinkedListNode` class's instance variables:
  - ```
    E value;
    DoublyLinkedListNode next;
    DoublyLinkedListNode prev;
    ```
    ```

DoublyLinkedLists

- Keep reference/links in **both** directions
  - Can therefore traverse forwards and backwards!

- `DoublyLinkedListNode` class's instance variables:
  - ```
    E value;
    DoublyLinkedListNode next;
    DoublyLinkedListNode prev;
    ```

- This adds one more reference *per node*, but overall space overhead still proportional to number of elements
    ```

Linked List Visualization

SinglyLinkedList:



DoublyLinkedList:



DoublyLinkedList Tradeoffs

DoublyLinkedList Tradeoffs

- ALL tail operations (including `removeLast`) are fast!
  - Why? We have direct access to the tail node & *its predecessor*

DoublyLinkedList Tradeoffs

- <u>ALL</u> tail operations (including `removeLast`) are fast!
  - Why? We have direct access to the tail node & *its predecessor*
- But, additional *code complexity* in each list operation
  - Example: `add(E d, int index)` has four cases to consider now:
    - empty list
    - add to front
    - add to tail
    - add in middle

DoublyLinkedList Tradeoffs

- ALL tail operations (including `removeLast`) are fast!
  - Why? We have direct access to the tail node & *its predecessor*
- But, additional *code complexity* in each list operation
  - Example: `add(E d, int index)` has four cases to consider now:
    - empty list
    - add to front
    - add to tail
    - add in middle
- Some additional space consumption (`previous`)
  - but space overhead is still O(n) like SLL and Vector

```
public class DoublyLinkedNode<E> {

}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;















}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor “stitches“ new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {









   }
}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor “stitches“ new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {
      data = v;








   }
}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor “stitches“ new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {
      data = v;

      nextElement = next;






   }
}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor “stitches“ new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {
      data = v;

      nextElement = next;
      if (nextElement != null)
         nextElement.previousElement = this;




   }
}
```

```
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor "stitches" new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {
      data = v;

      nextElement = next;
      if (nextElement != null)
         nextElement.previousElement = this;

      previousElement = previous;


   }
}
```

```java
public class DoublyLinkedNode<E> {
   protected E data;
   protected DoublyLinkedNode<E> nextElement;
   protected DoublyLinkedNode<E> previousElement;

   // Constructor "stitches" new node btwn existing nodes
   public DoublyLinkedNode(E v,
                           DoublyLinkedNode<E> next,
                           DoublyLinkedNode<E> previous) {
      data = v;

      nextElement = next;
      if (nextElement != null)
         nextElement.previousElement = this;

      previousElement = previous;
      if (previousElement != null)
            previousElement.nextElement = this;
   }
}
```

```
public void add(int i, E value) {




}
```

```java
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head




}
```

```
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail




}
```

```java
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {









    }
}
```

```
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {
        // find items before and after insertion point










    }
}
```

```
public void add(int i, E value) {
     if (i == 0) addFirst(value); // head
     else if (i == size()) addLast(value); // tail
     else {
        // find items before and after insertion point
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;









     }
}
```

```
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {
        // find items before and after insertion point
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;
        while (i > 0) {
            before = after;
            after = after.next();
            i--;
        }




    }
}
```

```java
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {
        // find items before and after insertion point
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;
        while (i > 0) {
            before = after;
            after = after.next();
            i--;
        }
        // create new value to "splice" into list
        // note: constructor properly updates neighbors
        DoublyLinkedNode<E> insertedNode =
              new DoublyLinkedNode<E>(value,after, before);

    }
}
```

```
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {
        // find items before and after insertion point
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;
        while (i > 0) {
            before = after;
            after = after.next();
            i--;
        }
        // create new value to ”splice” into list
        // note: constructor properly updates neighbors
        DoublyLinkedNode<E> insertedNode =
              new DoublyLinkedNode<E>(value,after, before);
        count++;
    }
}
```

Vectors vs. SLL vs. DLL

Operation	Vector	SLL	DLL
size	O(1)	O(1)	O(1)
addLast	O(1) or O(n)(if resize)	O(n)	O(1)
removeLast	O(1)	O(n)	O(1)
getLast	O(1)	O(n)	O(1)
addFirst	O(n)	O(1)	O(1)
removeFirst	O(n)	O(1)	O(1)
getFirst	O(1)	O(1)	O(1)
get(i)	O(1)	O(n)	O(n)
set(i)	O(1)	O(n)	O(n)
remove(i)	O(n)	O(n)	O(n)
contains	O(n)	O(n)	O(n)
remove(o)	O(n)	O(n)	O(n)