BASIC DATA STRUCTURES

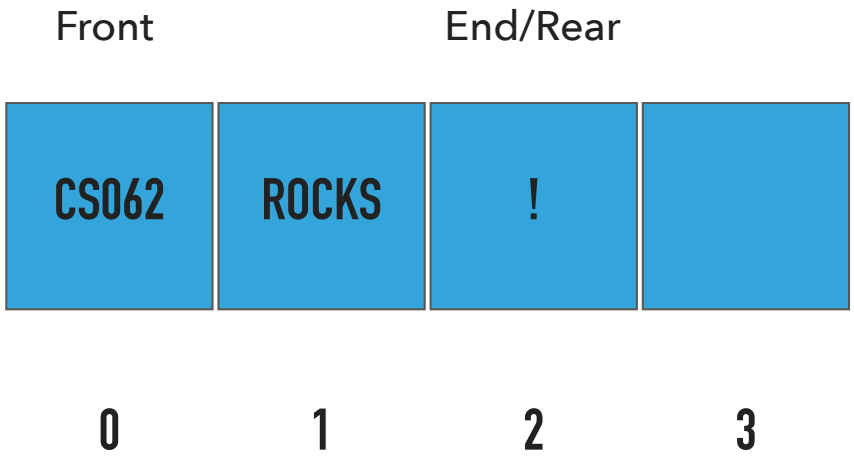
CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

7: Singly Linked Lists

Tom Yeh
he/him/his

`add(Item item)`: Appends the item to the end of the ArrayList



DOUBLE CAPACITY SINCE IT'S FULL AND THEN ADD NEW ITEM INCREASE NUMBER OF ITEMS

```
al.add("!");
```

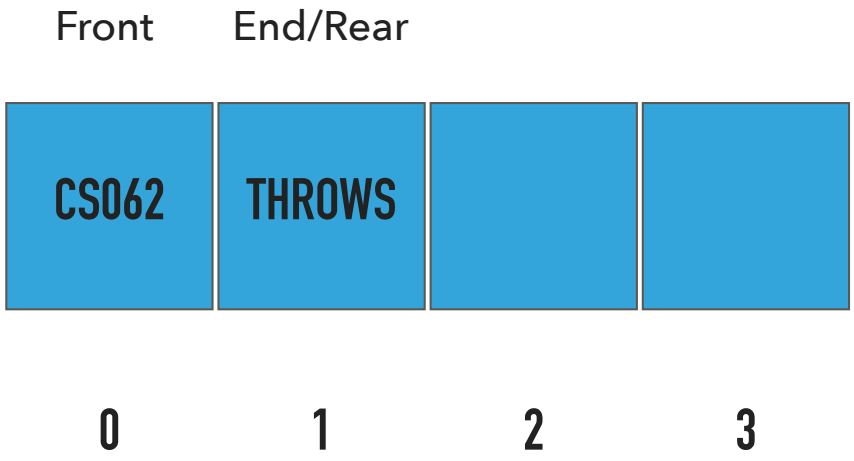
Capacity = 4

Number of items = 3

What should happen?

```
al.add(1, "THROWS");
```

`remove()`: Retrieves and removes item from the end of ArrayList




```
al.remove();
```

Capacity = 4

Number of items = 2

REMOVE ITEM FROM THE END
HALVE CAPACITY WHEN 1/4 FULL

What should happen?

```
al.remove(0);
```

Asymptotic analysis simplifies analyzing worst-case performance

- Sorting them by increasing rate of growth:
  - $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!)$

How to interpret Big O

- $O(1)$ or "order one”: running time does not change as size of the problem changes, that is running time stays constant and independent of problem size.
- $O(\log n)$ or "order log n”: running time increases as problem size grows. Whenever problem size doubles, running time increases by a constant.
- $O(n)$ or "order n”: time increases proportionally to the the rate of growth of the size of the problem, that is in a linear rate. Double the problem size, you get double running time.
- $O(n^2)$ or "order n squared”: Double the problem size you get quadruple running time.

Lecture 6: Resizable Arrays

- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

Worst-case performance of `add()` is $O(n)$

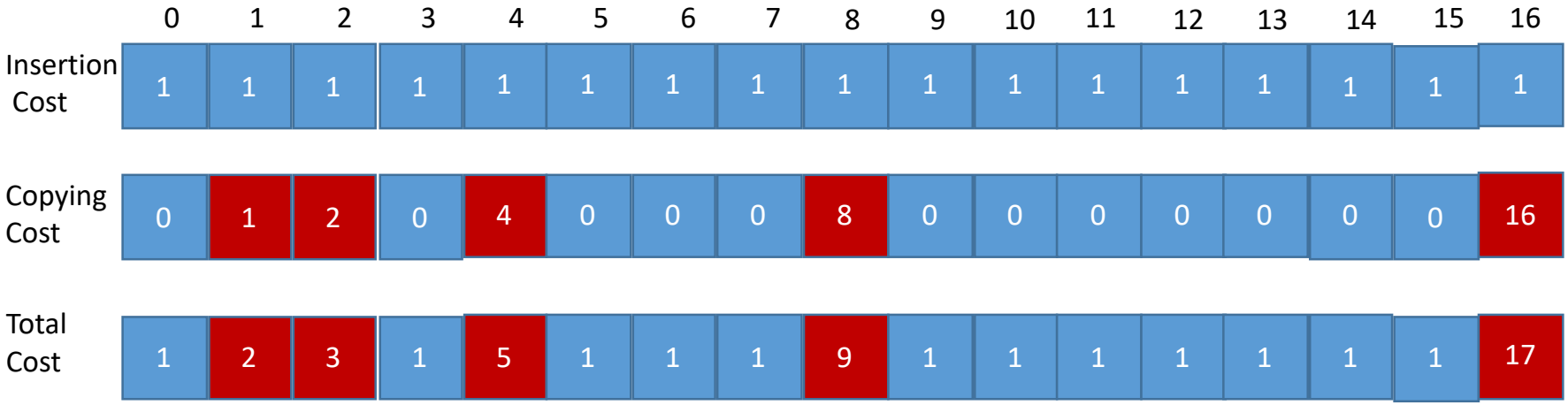
- Cost model: 1 for insertion, n for copying n items to a new array.
- Worst-case: If ArrayList is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- Total cost: $n + 1 = O(n)$.

- Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

Amortized analysis

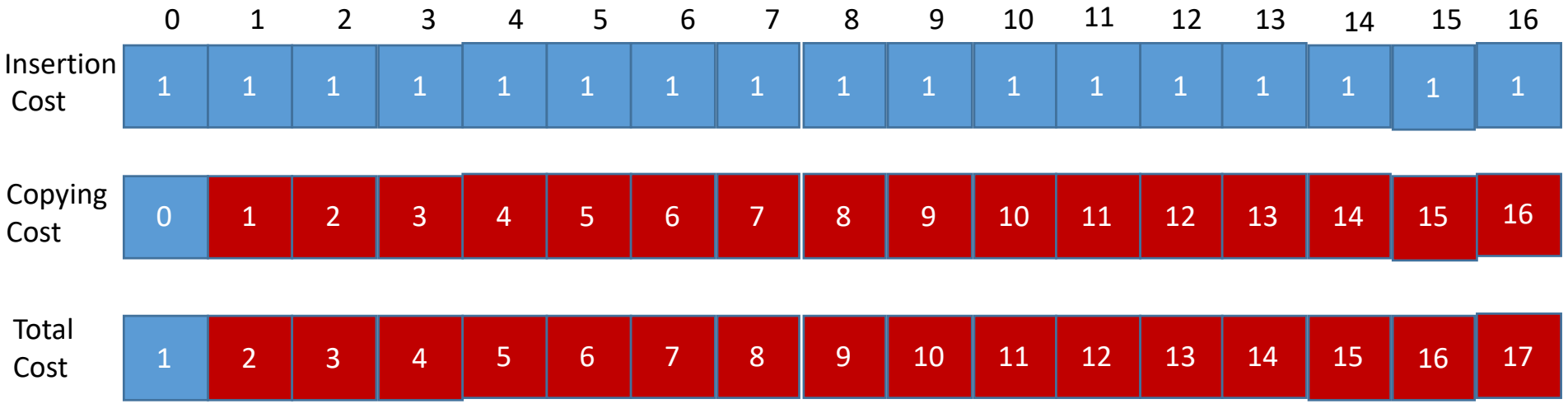
- Amortized cost per operation: for a sequence of n operations, it is the total cost of operations divided by n.
  - Simplest form of amortized analysis called aggregate method. More complicated methods exist, such as accounting (banking) and potential (physicist's).

Amortized analysis for n add() operations



- As the ArrayList increases, doubling happens *half as often* but *costs twice as much*.
- O(total cost)= $\sum$("cost of insertions") + $\sum$("cost of copying")
- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") = $1 + 2 + 2^2 + \ldots 2^{\log_2(n-1)} \leq 2n$.
- O(total cost) $\leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O(1)$, but "lumpy".

Amortized analysis for n add() operations when increasing ArrayList by 1.



- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") $= 1 + 2 + 3 + \ldots + n - 1 = n(n-1)/2$.
- O(total cost) $= n + n(n-1)/2 = n(n+1)/2$, therefore amortized cost is $(n+1)/2$ or $O(n)$.
- Same idea when increasing ArrayList size by a constant.
  - This is why in the lab yesterday, we saw that doubling was the fastest and linear(1) the slowest

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Textbook:
  - Chapter 1.3 (Page 136–137)
  - Chapter 1.4 (pages 197–199)
- Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Practice Problems:

- 1.4.1, 1.4.5 - 1.4.7, 1.4.32, 1.4.35-1.4.36.

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Textbook:
  - Chapter 1.3 (Page 136–137)
  - Chapter 1.4 (pages 197–199)
- Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Practice Problems:

- 1.4.1, 1.4.5 - 1.4.7, 1.4.32, 1.4.35-1.4.36.

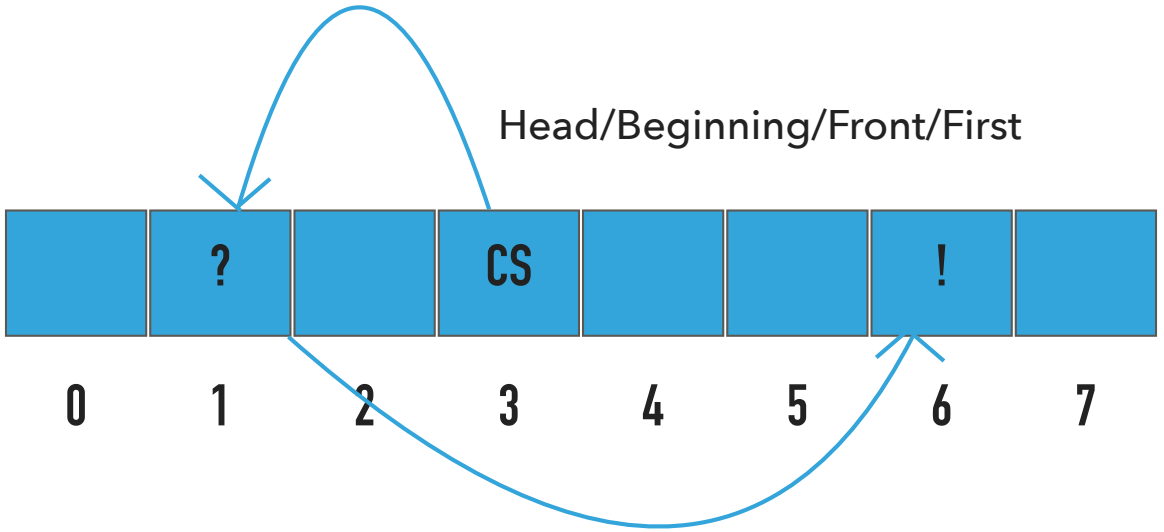
Lecture 7: Singly Linked Lists

- Singly Linked Lists

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

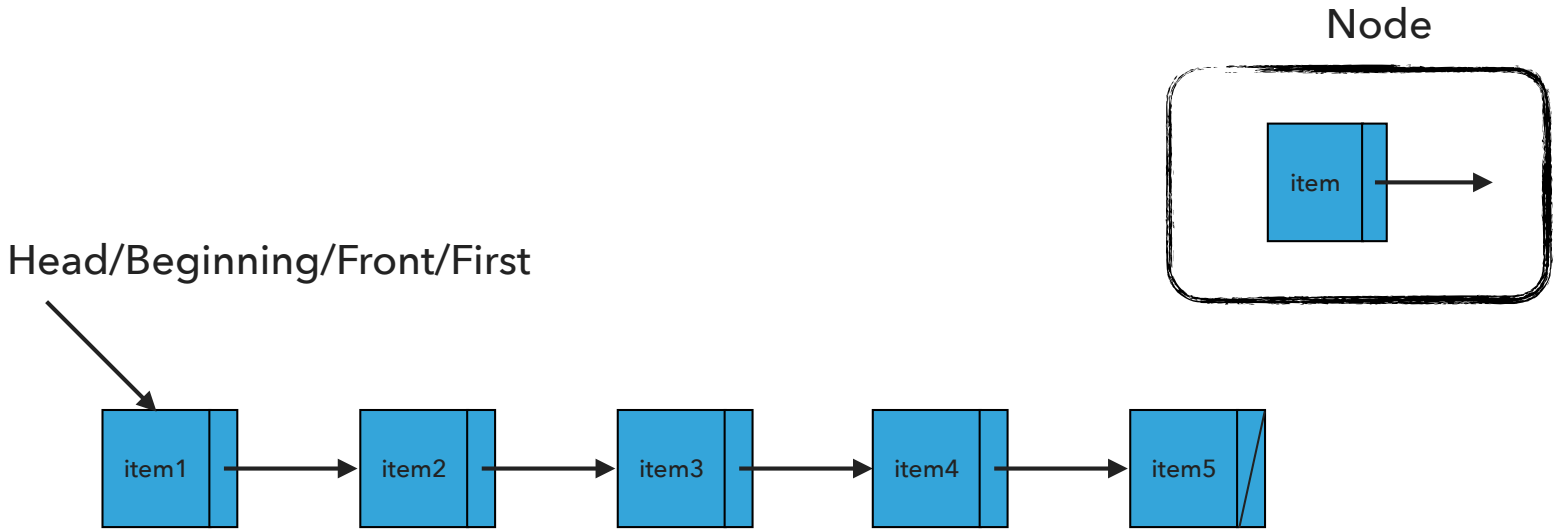
Singly Linked Lists

- Dynamic linear data structures.
- In contrast to sequential data structures, linked data structures use pointers/links/references from one object to another.



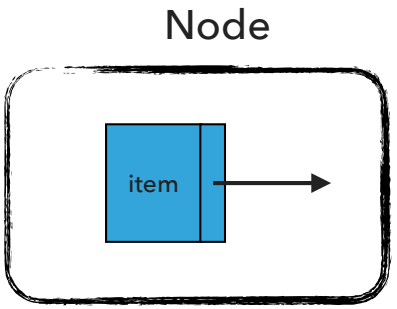
Recursive Definition of Singly Linked Lists

- A singly linked list is either empty (null) or a node having a reference to a singly linked list.
- Node: is a data type that holds any kind of data and a reference to a node.



Node

```
private class Node {
    Item item;
    Node next;
  }
```



Standard Operations

- `SinglyLinkedList()`: Constructs an empty singly linked list.
- `isEmpty()`:Returns true if the singly linked list does not contain any item.
- `size()`: Returns the number of items in the singly linked list.
- `Item get(int index)`: Returns the item at the specified index.
- `add(Item item)`: Inserts the specified item at the head of the singly linked list.
- `add(int index, Item item)`: Inserts the specified item at the specified index.
- `Item remove()`: Removes and returns the head of the singly linked list.
- `Item remove(int index)`: Removes and returns the item at the specified index.

`SinglyLinkedList()`: Constructs an empty SLL

first = ?

n = ?

What should happen?

```
SinglyLinkedList<String> sll = new SinglyLinkedList<String>();
```

`SinglyLinkedList()`: Constructs an empty SLL

```
SinglyLinkedList<String> sll = new SinglyLinkedList<String>();
```

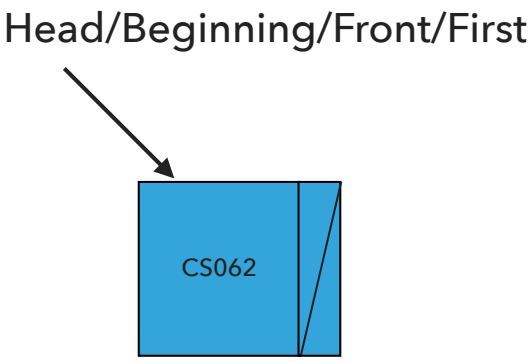
first = null

n = 0

What should happen?

```
sll.add("CS062");
```

`add(Item item)`: Inserts the specified item at the head of the singly linked list



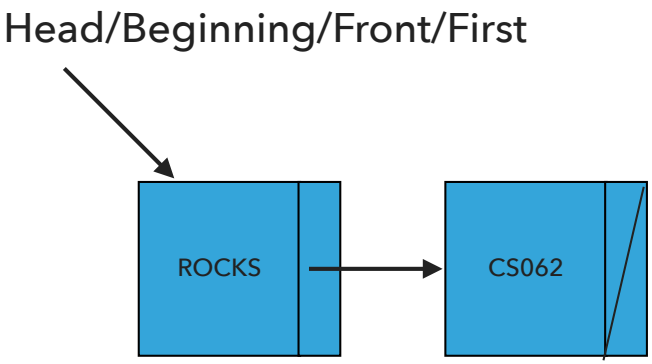

```
sll.add("CS062")

n=1
```

What should happen?

```
sll.add("ROCKS");
```

`add(Item item)`: Inserts the specified item at the head of the singly linked list



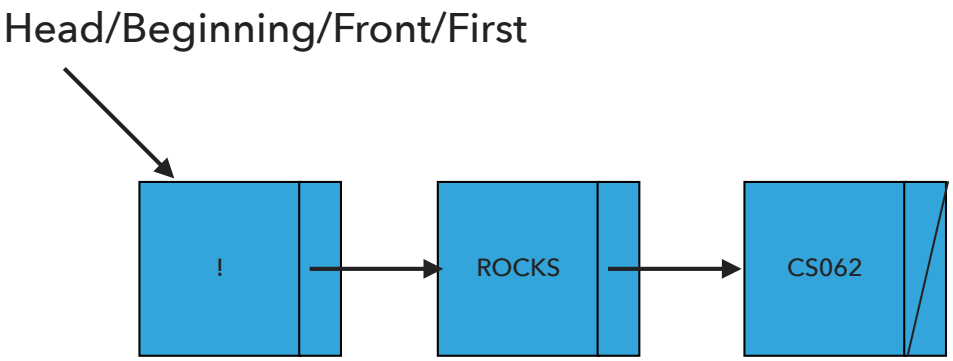

```
sll.add("ROCKS")

n=2
```

What should happen?

```
sll.add("!");
```

`add(Item item)`: Inserts the specified item at the head of the singly linked list



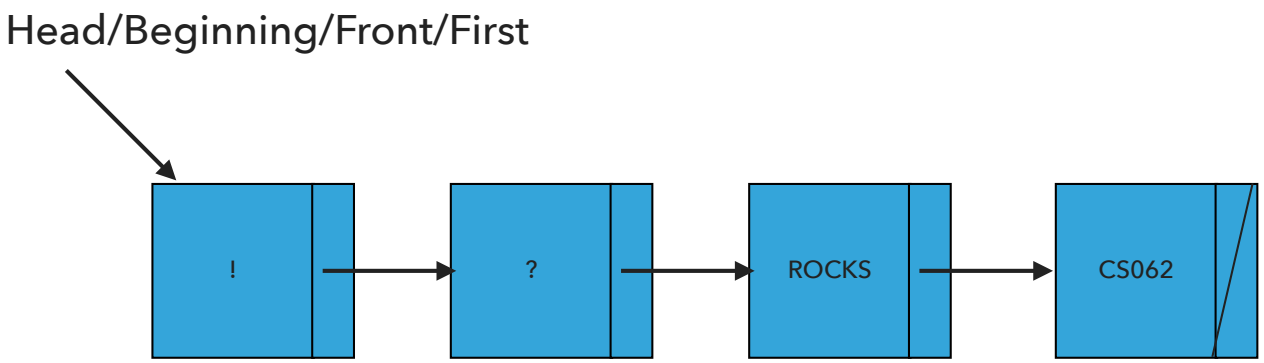

```
sll.add("!")

n=3
```

What should happen?

```
sll.add(1,"?");
```

`add(int index, Item item)`: Adds item at the specified index



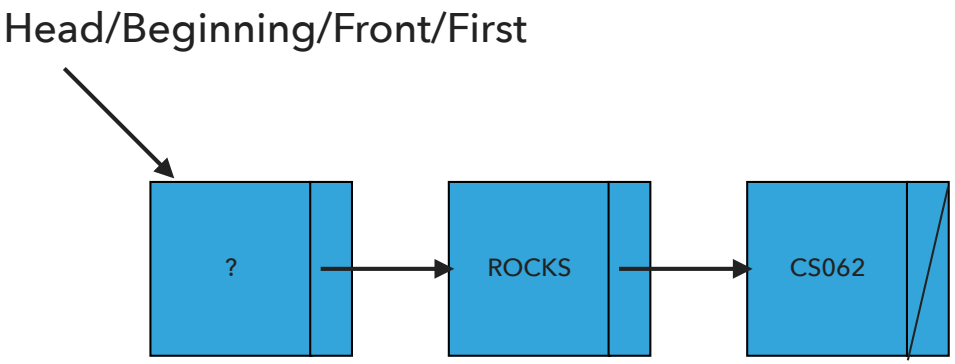

```
sll.add(1,“?”)

n=4
```

What should happen?

```
sll.remove();
```

`remove()`: Retrieves and removes the head of the singly linked list



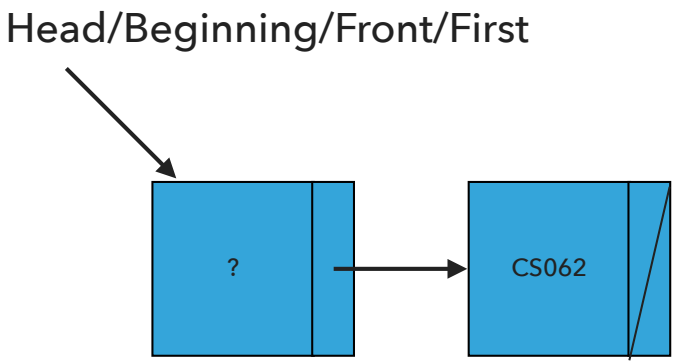

```
sll.remove()

n=3
```

What should happen?

```
sll.remove(1);
```

`remove(int index)`: Retrieves and removes the item at the specified index




```
sll.remove(1)

n=2
```

Our own implementation of Singly Linked Lists

- We will follow the textbook style.
  - It does not offer a class for this so we will build our own.
- We will work with generics because we don’t want to offer multiple implementations.
- We will use an inner class Node and we will keep track of how many elements we have in our singly linked list.

Instance variables and inner class

```
public class SinglyLinkedList<Item> implements Iterable<Item> {
   private Node first; // head of the singly linked list
   private int n; // number of nodes in the singly linked list

   /**
    * This nested class defines the nodes in the singly linked list with a
value
    * and pointer to the next node they are connected.
    */
   private class Node {
      Item item;
      Node next;
   }
```

Check if is empty and how many items

```
/**
 * Returns true if the singly linked list does not contain any item.
 *
 * @return true if the singly linked list does not contain any item
 */
public boolean isEmpty() {
    return first == null; // return size() == 0;
}

/**
 * Returns the number of items in the singly linked list.
 *
 * @return the number of items in the singly linked list
 */
public int size() {
    return n;
}
```

Check if index is >=0 and <n

```
/**
 * A helper method to check if an index is in range 0<=index<n
 *
 * @param index
 *            the index to check
 */
private void rangeCheck(int index) {
    if (index > n || index < 0)
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
}
```

Retrieve item from specified index

```
 /**
 * Returns item at the specified index.
 *
 * @param index
 *            the index of the item to be returned
 * @return the item at specified index
 */
public Item get(int index) {
    // check whether index is valid
    rangeCheck(index);
    // set a temporary pointer to the head
    Node finger = first;
    // search for index-th element or end of list
    while (index > 0) {
        finger = finger.next;
        index--;
    }
    // return the item stored in the node that the temporary pointer points to
    return finger.item;
}
```

Insert item at head of singly linked list

```
 /**
 * Inserts the specified item at the head of the singly linked list.
 *
 * @param item
 *            the item to be inserted
 */
public void add(Item item) {
    // Create a pointer to head
    Node oldfirst = first;

    // Make a new node that will hold the item and assign it to head.
    first = new Node();
    first.item = item;
    // fix pointers
    first.next = oldfirst;
    // increase number of nodes
    n++;
}
```

Insert item at a specified index

```
/**
 * Inserts the specified item at the specified index.
 *
 * @param index
 *            the index to insert the node
 * @param item
 *            the item to insert
 */
public void add(int index, Item item) {
      // check that index is within range
      rangeCheck(index);
      // if index is 0, then call one-argument add
      if (index == 0) {
            add(item);
      // else
      } else {
            // make two pointers, previous and finger. Point previous to null and finger to head
            Node previous = null;
            Node finger = first;
            // search for index-th position by pointing previous to finger and advancing finger
            while (index > 0) {
                  previous = finger;
                  finger = finger.next;
                  index--;
            }
            // create new node to insert in correct position. Set its pointers and contents
            Node current = new Node();
            current.next = finger;
            current.item = item;
            // make previous point to newly created node.
            previous.next = current;
            // increase number of nodes
            n++;
      }
}
```

Retrieve and remove head

```
/**
     * Retrieves and removes the head of the singly linked list.
     *
     * @return the head of the singly linked list.
     */
    public Item remove() {
        // Make a temporary pointer to head
        Node temp = first;
        // Move head one to the right
        first = first.next;
        // Decrease number of nodes
        n--;
        // Return item held in the temporary pointer
        return temp.item;
    }
```

Retrieve and remove element from a specific index

```
/**
 * Retrieves and removes the item at the specified index.
 *
 * @param index
 *            the index of the item to be removed
 * @return the item previously at the specified index
 */
public Item remove(int index) {
     // check that index is within range
      rangeCheck(index);
     // if index is 0, then call remove
      if (index == 0) {
            return remove();
     // else
      } else {
     // make two pointers, previous and finger. Point previous to null and finger to head
            Node previous = null;
            Node finger = first;
            // search for index-th position by pointing previous to finger and advancing finger
            while (index > 0) {
                  previous = finger;
                  finger = finger.next;
                  index--;
            }
            // make previous point to finger's next
            previous.next = finger.next;
            // reduce number of items
            n--;
            // return finger's item
            return finger.item;
      }

}
```

add() in singly linked lists is $O(1)$ for worst case

```
public void add(Item item) {
   // Save the old node
   Node oldfirst = first;

   // Make a new node and assign it to head. Fix pointers.
   first = new Node();
   first.item = item;
   first.next = oldfirst;

   n++; // increase number of nodes in singly linked list.
}
```

get() in singly linked lists is $O(n)$ for worst case

```
public Item get(int index) {
    rangeCheck(index);

    Node finger = first;
    // search for index-th element or end of list
    while (index > 0) {
        finger = finger.next;
        index--;
    }
    return finger.item;
}
```

`add(int index, Item item)` in singly linked lists is $O(n)$ for worst case

```
public void add(int index, Item item) {
        rangeCheck(index);

        if (index == 0) {
            add(item);
        } else {

            Node previous = null;
            Node finger = first;
            // search for index-th position
            while (index > 0) {
                previous = finger;
                finger = finger.next;
                index--;
            }
            // create new value to insert in correct position.
            Node current = new Node();
            current.next = finger;
            current.item = item;
            // make previous value point to new value.
            previous.next = current;

            n++;
        }
    }
```

remove() in singly linked lists is $O(1)$ for worst case

```
public Item remove() {
   Node temp = first;
   // Fix pointers.
   first = first.next;

   n--;

   return temp.item;
}
```

`remove(int index)` in singly linked lists is $O(n)$ for worst case

```
public Item remove(int index) {
    rangeCheck(index);

    if (index == 0) {
        return remove();
    } else {
        Node previous = null;
        Node finger = first;
        // search for value indexed, keep track of previous
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        previous.next = finger.next;

        n--;
        // finger's value is old value, return it
        return finger.item;
    }

}
```

Lecture 7: Singly Linked Lists

- Singly Linked Lists

Readings:

- Textbook:
  - Chapter 1.3 (Page 142–146)
- Textbook Website:
  - Linked Lists: https://algs4.cs.princeton.edu/13stacks/

Practice Problems:

- 1.3.18–1.3.27