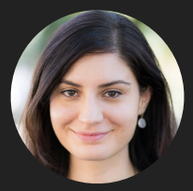
BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

9: Singly Linked Lists



Alexandra Papoutsaki
she/her/hers

Exciting times! We are ready to see linked linear data structures. We will start with singly linked lists.

Lecture 9: Singly Linked Lists

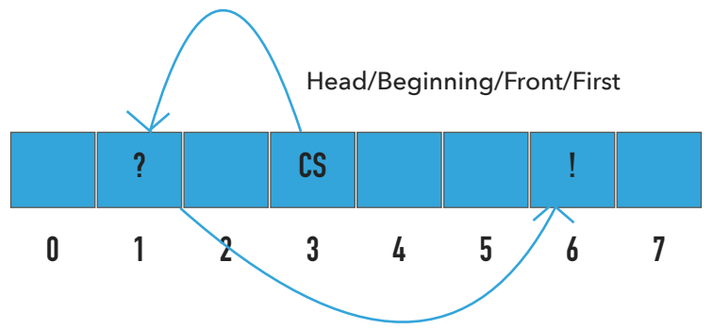
- Singly Linked Lists

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

I will mostly present things today and then next time when we talk about doubly linked data structures you will apply your knowledge from this lecture to build the code for them.

Singly Linked Lists

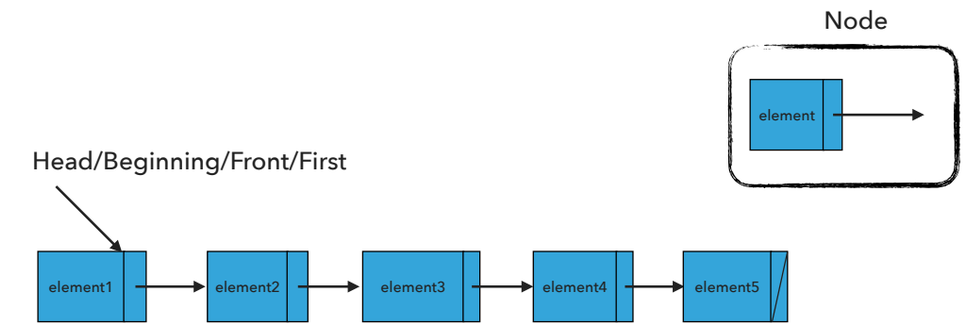
- Dynamic linear data structures.
- In contrast to sequential data structures, linked data structures use pointers/links/references from one object to another.



Singly linked lists are dynamic linear data structures that can accommodate increasing needs for more data. In contrast to sequential data structures, linked data structures use pointers/links/references from one object to another. For example, we might have CS,?,! with CS being the head (or beginning, front, first), and ? and ! being in completely random locations in memory. What allows us to know their sequence is these pointers.

Recursive Definition of Singly Linked Lists

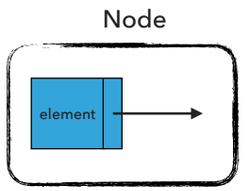
- A singly linked list is either empty (null) or a node having a reference to a singly linked list.
- Node: is a data type that holds any kind of data and a reference to a node.



The recursive definition of singly linked lists is that a singly linked list is either empty (null) or a node having a reference to a singly linked list. A node is a data type that holds any kind of data and a reference to a node. This is how we will visualize a singly linked list. Note that the last node does not point to any node and we mark that with a /.

Node

```
private class Node {
    E element;
    Node next;
  }
```



A node will be represented through the inner* private class Node that has two instance variables. An element of type E and a reference to the next Node. *An inner class is a nested class within another class. It allows us to group things together and create objects of type node without needing to put that in a separate file.

Reminder: Interface List

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

Let’s refresh our memory about the List interface. If we implement it, we promise to implement the methods:

```
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
```

Standard Operations

- `SinglyLinkedList()`: Constructs an empty singly linked list.
- `isEmpty()`:Returns true if the singly linked list does not contain any element.
- `size()`: Returns the number of elements in the singly linked list.
- `E get(int index)`: Returns the element at the specified index.
- `add(E element)`: Inserts the specified element at the head of the singly linked list.
- `add(int index, E element)`: Inserts the specified element at the specified index.
- `E set(int index, E element)`: Replaces the specified element at the specified index and returns the old element
- `E remove()`: Removes and returns the head of the singly linked list.
- `E remove(int index)`: Removes and returns the element at the specified index.
- `clear()`: Removes all elements.

These are the standard operations we expect to have. We will have a constructor and usual methods for checking the size, whether it is empty, a getter, two adds, one set, two removes, and one clear.

`SinglyLinkedList()`: Constructs an empty SLL

head = ?

size = ?

What should happen?

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

Let's say someone creates a singly linked lists of strings. what do you think should happen to the head and size?

`SinglyLinkedList()`: Constructs an empty SLL

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

head = null

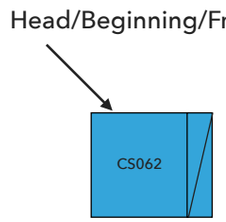
size = 0

What should happen?

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

the head will be null and the size zero. What would happen if we call sll.add("CS062");

`add(E element)`: Inserts the specified element at the head of the singly linked list



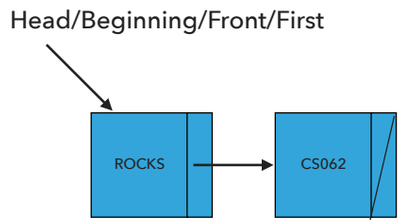

```
sll.add("CS062")

size=1
```

What should happen?

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

The head is the node that contains CS062 and the size is 1. What if we call sll.add("ROCKS");

`add(E element)`: Inserts the specified element at the head of the singly linked list



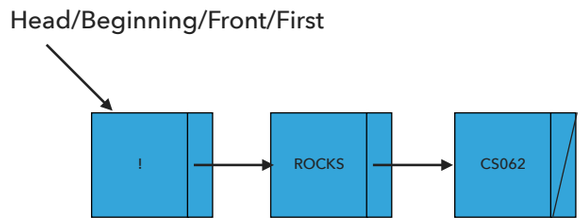

```
sll.add("ROCKS")

size=2
```

What should happen?

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

The addition will happen at the head. The head is now the node that contains ROCKS and the size is 2. What should happen if we type sll.add("!");

`add(E element)`: Inserts the specified element at the head of the singly linked list



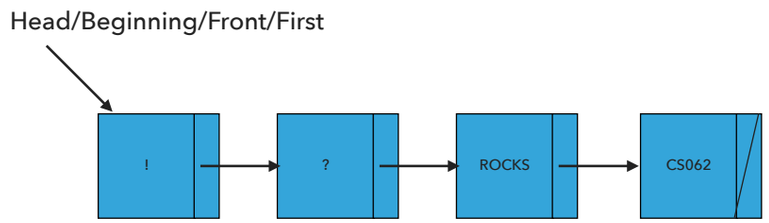

```
sll.add("!")

size=3
```

What should happen?

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

Again, the head is the newly inserted ! and the size is 3. What if we use the alternative form of add by calling sll.add(1,"?");

`add(int index, E element)`: Adds element at the specified index



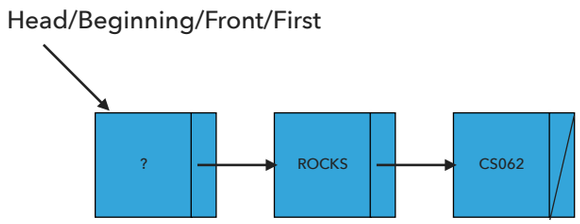

```
sll.add(1,“?”)

size=4
```

What should happen?

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

we are going to make space for a new node that will contain ? and the size will increase by 4. what should happen if we call sll.remove();

`remove()`: Removes and returns the head of the singly linked list



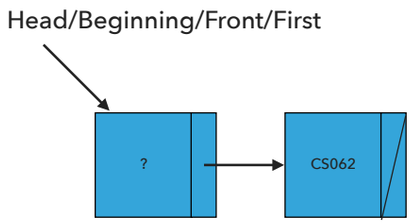

```
sll.remove()

size=3
```

What should happen?

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

The old head will be removed and the head will now be the node that contains ?. What if we call sll.remove(1);

`remove(int index)`: Removes and returns the element at the specified index




```
sll.remove(1)

size=2
```

The node that contains ROCKS will be removed and the size is 2.

Our own implementation of Singly Linked Lists

- We will follow the recommended textbook style.
  - It does not offer a class for this so we will build our own.
- We will work with generics because we want singly linked lists to hold objects of an type.
- We will implement the List interface we defined in past lectures.
- We will use an inner class Node and we will keep track of how many elements we have in our singly linked list.

Our own implementation of singly linked lists will lead us to work with generics. we will use the list interface and an inner class for nodes.

Instance variables and inner class

```
public class SinglyLinkedList<E> implements List<E>{
   private Node head; // head of the singly linked list
   private int size; // 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 {
      E element;
      Node next;
   }
```

That means that we will have two instance variables, head of type Node, and size of type int along with our inner private class for Node.

Check if is empty and how many elements

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

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

isEmpty can either check whether the head is null or the size 0. size is very simple.

Retrieve element from specified index

```
/**
 * Returns element at the specified index.
 *
 * @param index
 *            the index of the element to be returned
 * @return the element at specified index
 * @pre 0<=index<size
 */
public E get(int index) {
    // check whether index is valid
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    // set a temporary pointer to the head
    Node finger = head;
    // search for index-th element or end of list
    while (index > 0) {
        finger = finger.next;
        index--;
    }
    // return the element stored in the node that the temporary pointer points to
    return finger.element;
}
```

The get method will check that the index is within bounds and if not will throw an exception. We will next use a trick: we will create a reference that points to where head points to (NOT A NEW NODE!) We will move index steps to the right by pointing finger to finger.next. Eventually, when finger points to the right node, we will return the element it holds.

Insert element at head of singly linked list

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

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

To add a new element, we will make a reference to the old head. We will create a new node and make head now point to it. we will update the node to hold the given element and then we will make the new head point to the old head. And of course we will increase the size by 1.

Insert element at a specified index

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index to insert the node
 * @param element
 *            the element to insert
 * @pre 0<=index<=size
 */
public void add(int index, E element) {
      // check that index is within range
      if (index > size || index < 0){
          throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
      }
      // if index is 0, then call one-argument add
      if (index == 0) {
            add(element);
      // else
      } else {
            // make two pointers, previous and finger. Point previous to null and finger to head
            Node previous = null;
            Node finger = head;
            // 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.element = element;
            // make previous point to newly created node.
            previous.next = current;
            // increase number of nodes
            size++;
      }
}
```

The overloaded add will work similarly. We will start by checking that the index is within bounds. If the index is 0, we can call the basic add. Otherwise, we will double down on our trick. We will use two pointers. Finger will start at the head and previous will be right before it (initially at null). As we advance index positions to the right, we will move previous and finger accordingly. Eventually, we will reach with finger, where we need to add the new node. We will create it and make previous point to it, and it to finger. And will increase the size by 1.

Replace element at a specified index

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index of the element to replace
 * @param element
 *            the element to be stored at the specific index
 * @return the old element that was replaced
 * @pre 0<=index<size
 */
public E set(int index, E element) {
       // check that index is within range
      if (index >= size || index < 0){
           throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
       }

       Node finger = head;
       // search for index-th position by pointing previous to finger and advancing finger
       while (index > 0) {
            finger = finger.next;
            index--;
        }
        // reference old element
        E old = finger.element;
        // update element at finger
        finger.element = element;
        // return old element
        return old;
    }
}
```

Replacing an element a specified index will require us to go to that node using the finger trick, keeping track of what the old element is, updating the node, and returning the old element.

Retrieve and remove head

```
/**
 * Removes and returns the head of the singly linked list.
 *
 * @return the head of the singly linked list.
 */
public E remove() {
    // Make a temporary pointer to head
    Node temp = head;
    // Move head one to the right
    head = head.next;
    // Decrease number of nodes
    size--;
    // Return element held in the temporary pointer
    return temp.element;
}
```

Remove needs to keep track of what the old head is, make its subsequent node the next head, reduce the size by 1 and return the element from the old head.

Retrieve and remove element from a specific index

```
/**
 * Removes and returns the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the element previously at the specified index
 * @pre 0<=index<size
 */
public E remove(int index) {
      // check that index is within range
     if (index >= size || index < 0){
               throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
      }
      // 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 = head;
          // 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 elements
          size--;
          // return finger's element
          return finger.element;
     }

}
```

The overloaded remove will check that the index is within bounds and if it is 0 will call the simple remove. Otherwise, it will use the double pointer trick to go to the right node and will sever the pointers accordingly.

Clear the singly linked list of all elements

```
/**
 * Clears the singly linked list of all elements.
 *
 */
public void clear(
        head = null;
        size = 0;

}
```

Clear is super simple. Just set the head to null and the size to 0. The garbage collector will take care of the rest.

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

```
public void add(E element) {
   // Save the old node
   Node oldfirst = head;

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

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

Let's look now into the running time complexity of add. It will be O(1). It does not depend on how many elements already exist in the singly linked list. The fact that we need to do a couple of operations doesn't matter. they don't scale linearly with the size of the singly linked list.

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

```
public E get(int index) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

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

Get is another story. It can take O(n) for worst case if we need to hop n steps to find the desired index.

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

```
public void add(int index, E element) {
        if (index > size || index < 0){
            throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }
        if (index == 0) {
            add(element);
        } else {

            Node previous = null;
            Node finger = head;
            // 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.element = element;
            // make previous value point to new value.
            previous.next = current;

            size++;
        }
    }
```

same idea for add, worst case is O(n).

`set(int index, E element)` in singly linked lists is $O(n)$ for worst case

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index of the element to replace
 * @param element
 *            the element to be stored at the specific index
 * @return the old element that was replaced
 * @pre 0<=index<size
 */
public E set(int index, E element) {
       // check that index is within range
      if (index >= size || index < 0){
           throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
       }

       Node finger = head;
       // search for index-th position by pointing previous to finger and advancing finger
       while (index > 0) {
            finger = finger.next;
            index--;
        }
        // reference old element
        E old = finger.element;
        // update element at finger
        finger.element = element;
        // return old element
        return old;
    }
}
```

and for set.

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

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

    size--;

    return temp.element;
}
```

remove from the head in contrast is O(1) like with add.

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

```
public E remove(int index) {
     if (index >= size || index < 0){
          throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
       }

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

         size--;
         // finger's value is old value, return it
         return finger.element;
    }

}
```

But remove at a specific index can be O(n)

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

```
/**
 * Clears the singly linked list of all elements.
 *
 */
public void clear(
        head = null;
        size = 0;

}
```

Clear is O(1)!

Lecture 9: Singly Linked Lists

- Singly Linked Lists

And that's all for today; stay tuned for doubly linked lists.

Readings:

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

Code

- Lecture 9 code

Practice Problems:

- 1.3.18–1.3.27