Lecture 9: 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.
private class Node {
    E element;
    Node next;
}
Reminder: Interface List

```java
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();
}
```
SINGLY LINKED LISTS

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.
SinglyLinkedList(): Constructs an empty SLL

head = ?

size = ?

What should happen?

SinglyLinkedList<String> sll = new SinglyLinkedList<String>();
SinglyLinkedList(): Constructs an empty SLL

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

head = null
size = 0

What should happen?

sll.add(“CS062”);
add(E element): Inserts the specified element at the head of the singly linked list

```java
sll.add("CS062");
size=1
```

What should happen?

```java
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("!");
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,"? ");
```
add(int index, E element): Adds element at the specified index

sll.add(1, "?")
size = 4

What should happen?

sll.remove();
remove(): Removes and returns the head of the singly linked list

```
sll.remove()
size=3

What should happen?

sll.remove(1);
```
remove(int index): Removes and returns the element at the specified index
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.
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;
    }
}
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;
}
```
/**
 * 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;
}
Insert element at head of singly linked list

```java
/**
 * 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++;
}
```
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 {  
        // 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 nxt;  
            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++;  
    }
}
/**
 * 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;
}
Retrieve and remove head

```java
/**
 * 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;
}
```
/**
 * 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 {
        // 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;
    }
}
Clear the singly linked list of all elements

```java
/**
 * Clears the singly linked list of all elements.
 */

public void clear(
    head = null;
    size = 0;

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

```java
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.
}
```
get() in singly linked lists is $O(n)$ for worst case

```java
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;
}
```
SINGLY LINKED LISTS

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

```java
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++;
    }
}
```
**set(int index, E element)** in singly linked lists is \( O(n) \) for worst case

```java
/**
 * 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;
}
```
remove() in singly linked lists is $O(1)$ for worst case

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

    size--;

    return temp.element;
}
```
```
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;
    }
}
```
clear() in singly linked lists is $O(1)$ for worst case

```java
/**
 * Clears the singly linked list of all elements.
 *
 */
public void clear(
    head = null;
    size = 0;
}
```
Lecture 9: Singly Linked Lists

- Singly Linked Lists
Readings:

- Recommended Textbook:
  - Chapter 1.3 (Page 142-146)

- Recommended Textbook Website:

Code

- Lecture 9 code

Practice Problems:

- 1.3.18-1.3.27