Linked Lists

Lecture 3 - SLLists, Nested Classes, Sentinel Nodes

```
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

While functional, "naked" linked lists like this are hard to use as they are recursively designed. You can also nest this class into SLL.

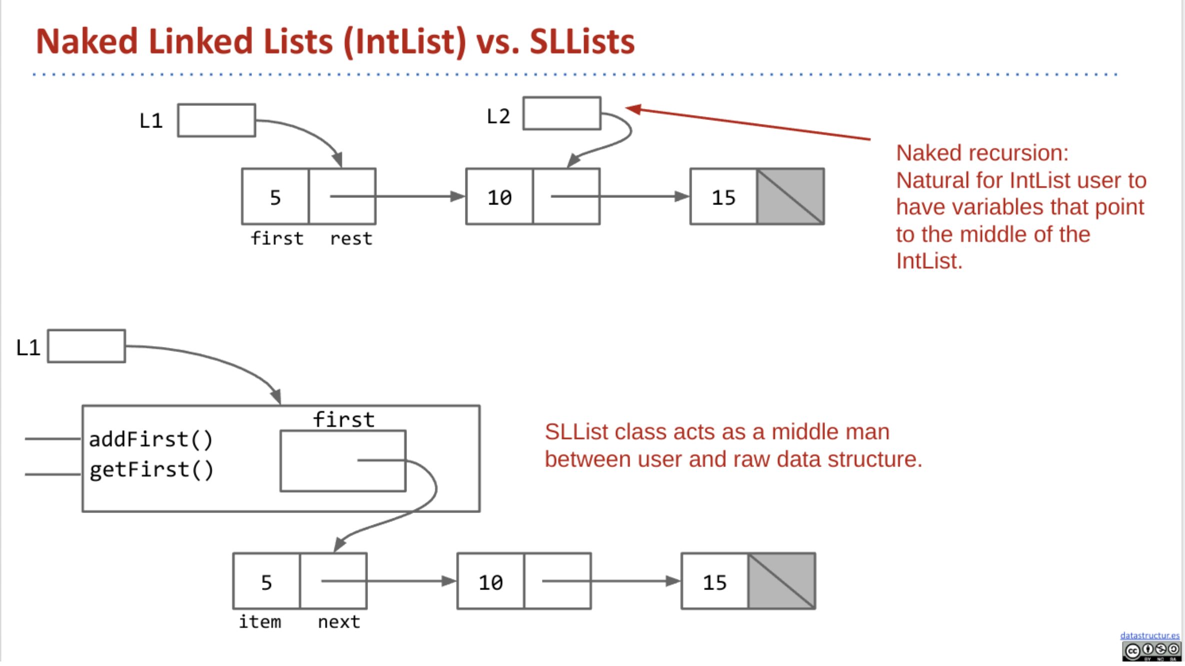
Singly Linked List

```
public class SLList {

    /* Nest IntNode class into SLL, to create a fully singly linked list */
    private static class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
        }
    }
    private IntNode first;
    public SLList(int x) {
        first = new IntNode(x, null);
    }
    public static void main(String[] args) {
        SLList L = new SLList(10);
        L.addFirst(10);
        L.addFirst(5);
        L.addLast(20);
     /* L.getFirst() -> 5
        L.size() -> 4   */
    }
    //adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }
    //returns first item in list
    public int getFirst() {
        return first.item;
    }
```

```
}
```



Private vs Public

Private - Hide implementation details from users of your class.

- Less for user of class to undertand.
- Safe for you to change private methods (implementation).

Car analogy:

- **Public:** Pedals, Steering Wheel **Private:** fuel line, Rotary Valve

Despite the term 'access control': Has nothing to do with protection against hackers/spies, other entities.

Public should generally stay available forever.

Improving the SLL

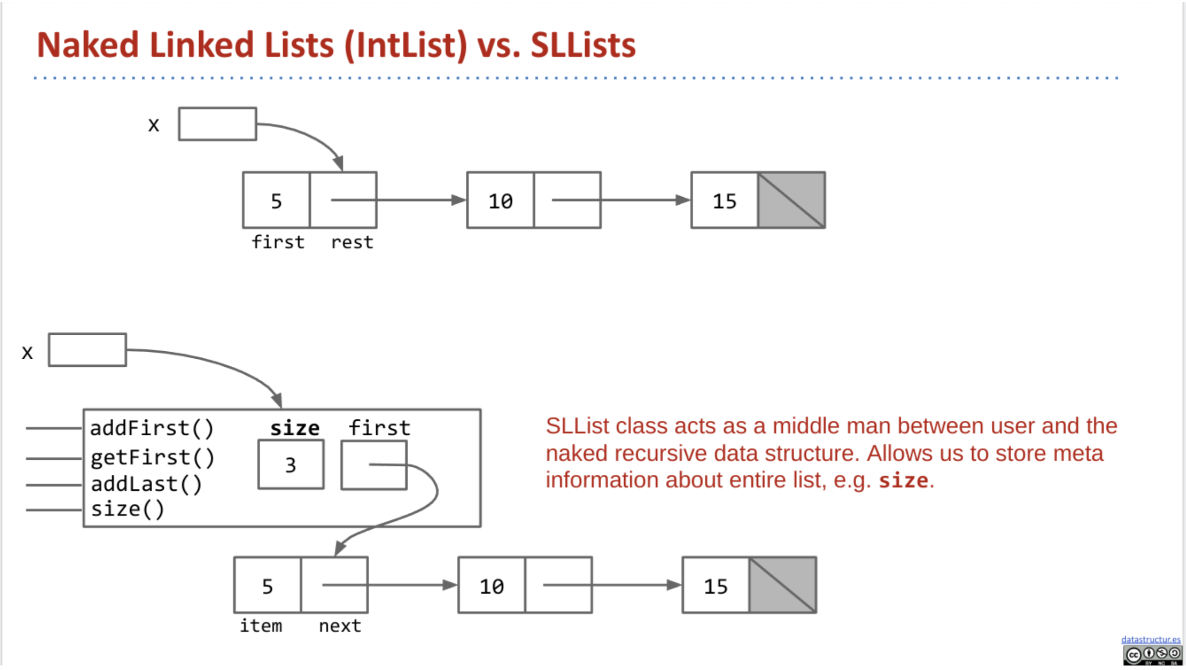
```
/*Adds an item to the end of the list */
public void addLast(int x) {
    IntNode p = first;
    //Move p until it reaches the end of the list
    while (p.next != null) {
        p = p.next;
    }
    p.next = new IntNode(x, null);
}
//returns the size of the list that starts at IntNode p
private static int size(IntNode p) {
    if (p.next == null) return 1;
    return 1 + size(p.next);
}
public int size() {
    return size(first);
}
```

However, the size method is very inefficient. It's calculating the size every single time you call it, which can take a long time. Instead, create a `private int size` that adds to itself every time a new node is created, and modify the methods to include `size += 1;`

This is called ***Caching*** - putting aside data to speed up retrieval.

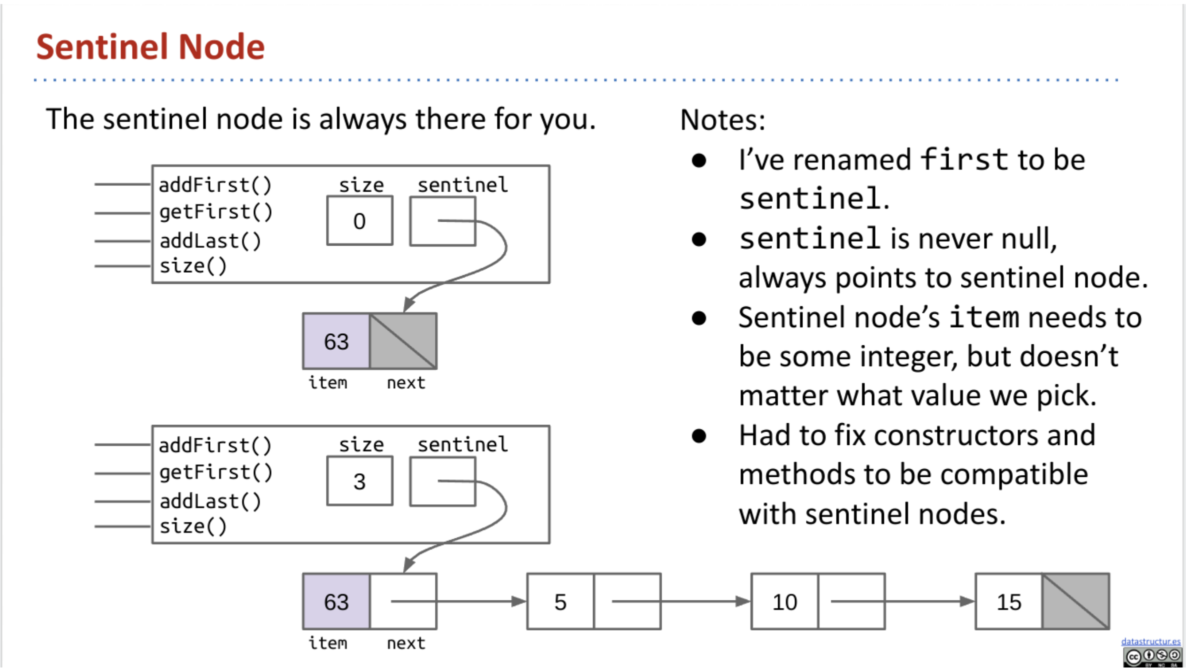
There is no such thing as a free lunch, but spreading the work over each add call is a net win in almost any circumstance.

Current diagram of the linked list we have created so far



Sentinel Nodes

```
private IntNode sentinel;
```



Invariant

An **invariant** is a condition that is guaranteed to be true during code execution.

An SLList with a sentinel node has at least the following invariants:

- The sentinel reference always points to a sentinel node.
- The first node (if it exists), is always at sentinel.next.
- The size variable is always the total number of items that have been added.

Invariants make it easier to reason about code:

- Can assume they are true to simplify code (e.g. addLast doesn't need to worry about nulls).
- Must ensure that methods preserve invariants.

And now, the Final SLL Class

```
public class SLList {
        /* Nest IntNode class into SLL, to create a fully singly linked list */
        private static class IntNode {
            public int item;
            public IntNode next;

            public IntNode(int i, IntNode n) {
                item = i;
                next = n;
            }
        }
        //the first item (if it exists) is at sentinel.next
        private IntNode sentinel;
        //set the counter of size for linked list
        private int size;
        // Creates an empty SLList - instantiate a list w/ no ints
        public SLList() {
            sentinel = new IntNode(1, null);
            size = 0;
        }
        public SLList(int x) {
            sentinel = new IntNode(1, null);
            sentinel.next = new IntNode(x, null);
            size = 1;
        }
        public static void main(String[] args) {
            SLList L = new SLList(10);
            L.addFirst(10);
            L.addFirst(5);
            L.addLast(20);
     /* L.getFirst() -> 5
        L.size() -> 4   */
        }
        //adds x to the front of the list
        public void addFirst(int x) {
            sentinel.next = new IntNode(x, sentinel.next);
            size += 1;
```

```
        }
        //returns first item in list
        public int getFirst() {
            return sentinel.next.item;
        }
    /*Adds an item to the end of the list */
    public void addLast(int x) {
        size += 1;
        IntNode p = sentinel;
        //Move p until it reaches the end of the list
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }
    //returns the size of the list
    public int size() {
        return size;
    }
}
```