Lecture 10 - February 6

Arrays and Linked Lists

SLL: List Constructions
SLL: getSize and getTail
Trading Space for Time: tail and size

Announcements/Reminders

- **Assignment 2** (on **SLL**) released
  + **Required** studies: Generics in Java (Slides 33 – 36)
  + **Recommended** studies: extra SLL problems
- **Assignment 1** solution released
- ***splitArrayHarder***: an **extended** version released
- Lecture notes template available
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

SLL: Constructing a Chain of Nodes

Alan → Mark → Tom

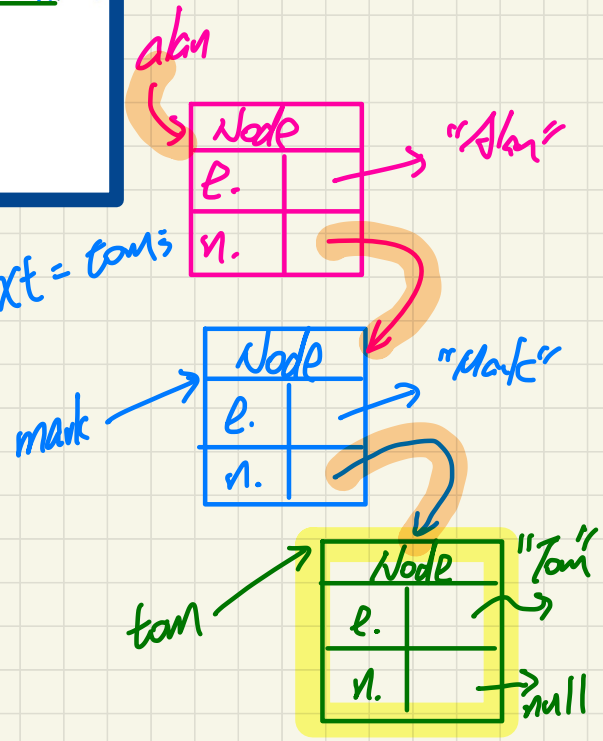
```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node n) { next = n; }
}
```

Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
```

mark.next = tom;

alan.next = mark

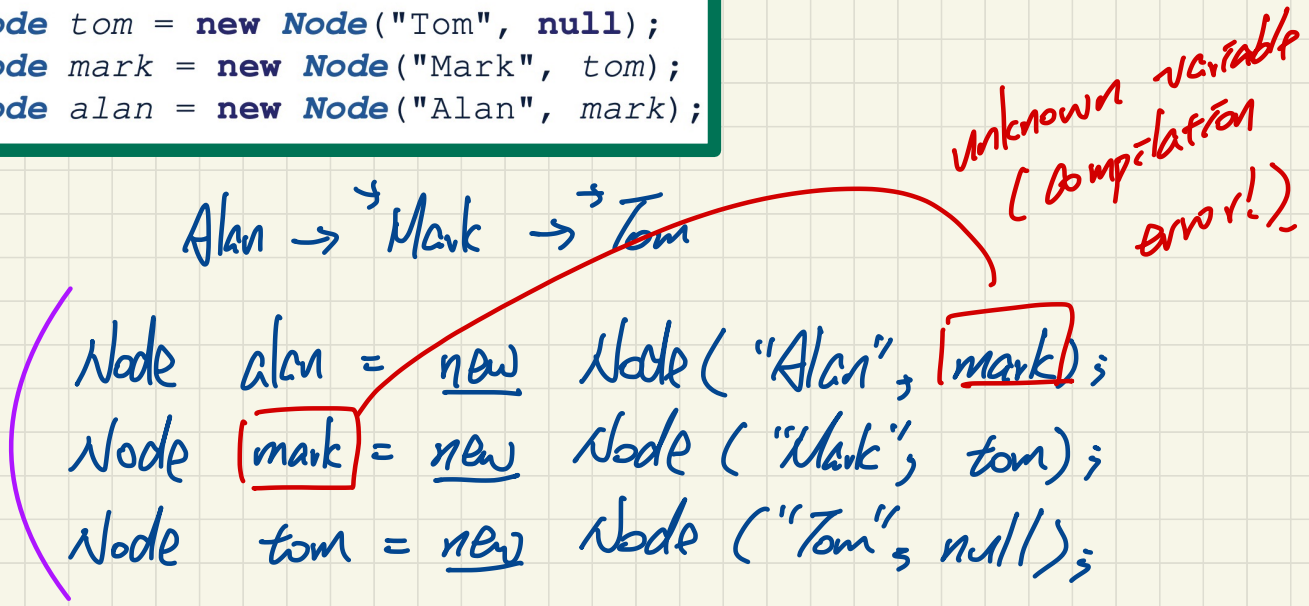


Aliasing

1. tom
2. mark.next → 3. alan.next.next

Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
```



Node alan;

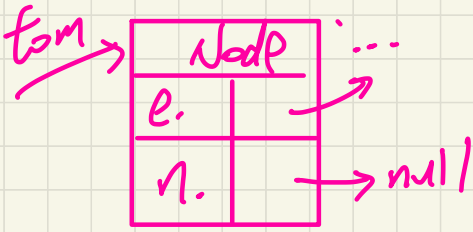
SLL: Constructing a Chain of Nodes

```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node n) { next = n; }
}
```

alan

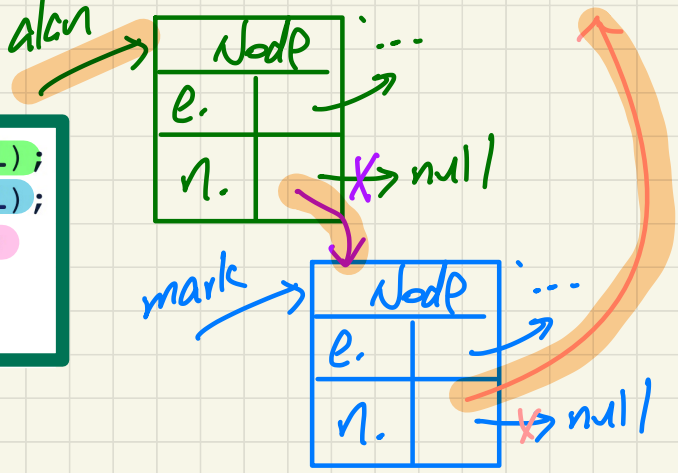
mark

tom



Approach 2

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
```

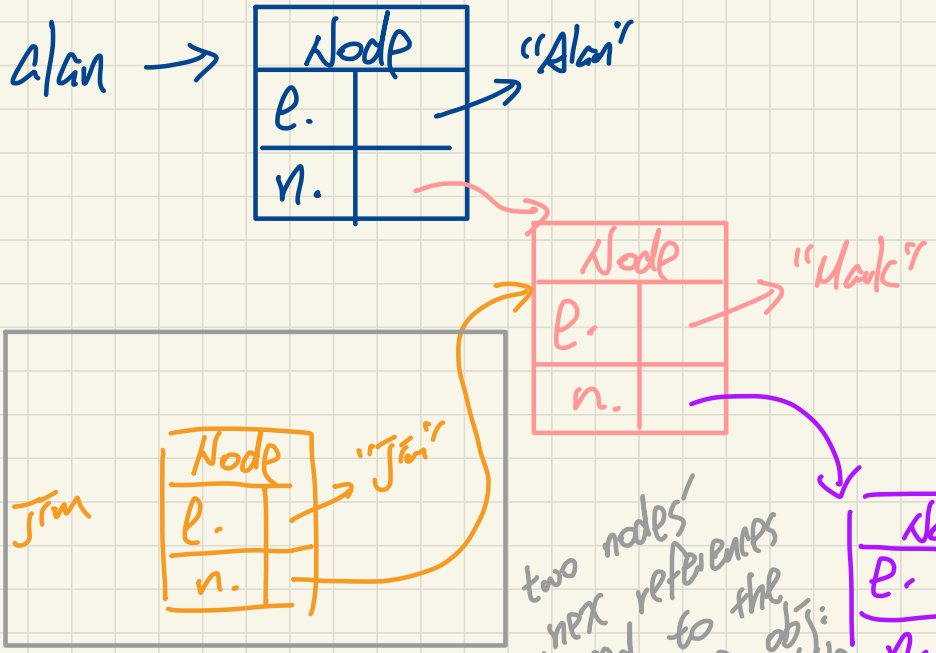


alan.next = mark;
mark.next = tom;

```
Node alan = new Node("Alan", );
```

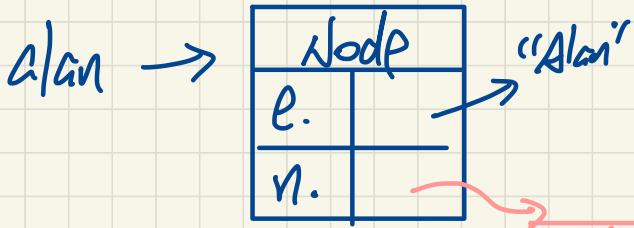
new Node("Mark",);

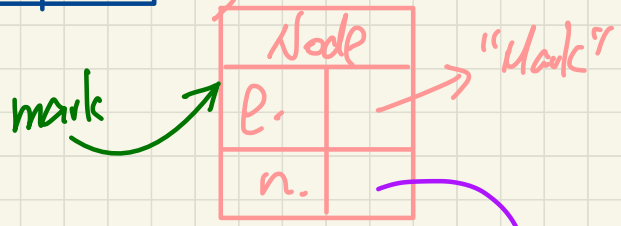
new Node("Tom", null)



two nodes' next references aliased to the same obj: possible but not recommended!

```
Node alan = new Node("Alan", );
```

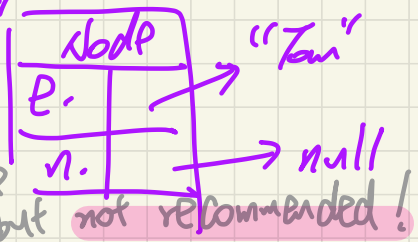





```
Node mark =
    alan.next;
```

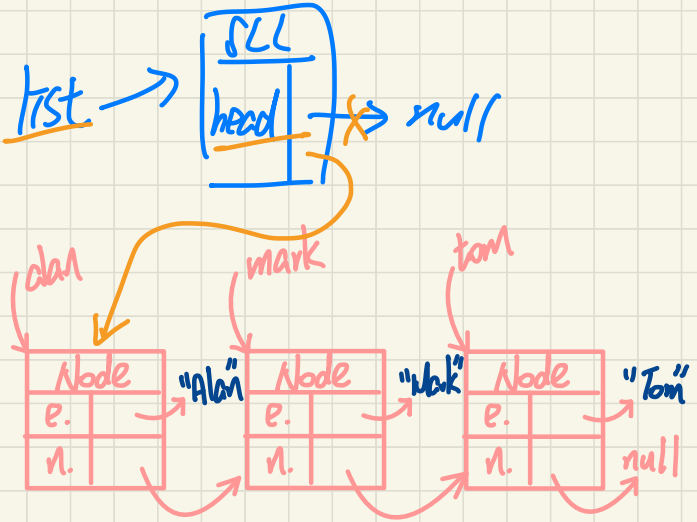
new Node("Tom", null)

two nodes' next references attached to the same obj: possible but not recommended!



SLL: Setting a List's Head to a Chain of Nodes

```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ... }
   public void removeLast() { ... }
}
```



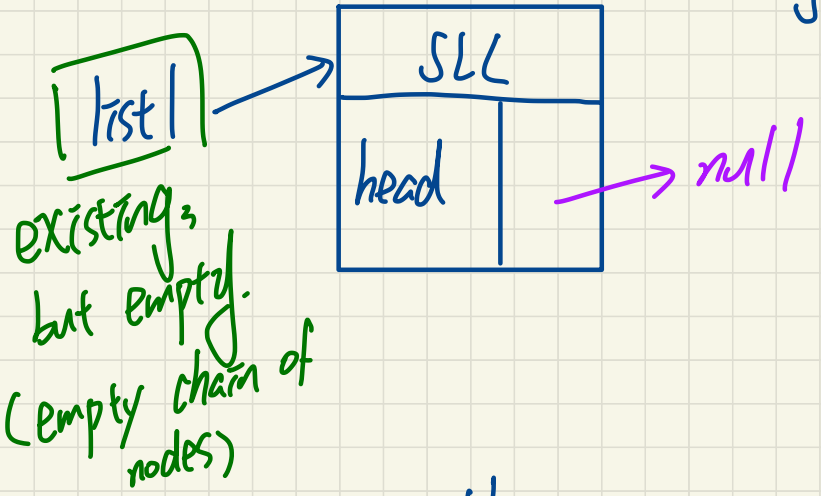
Approach 1

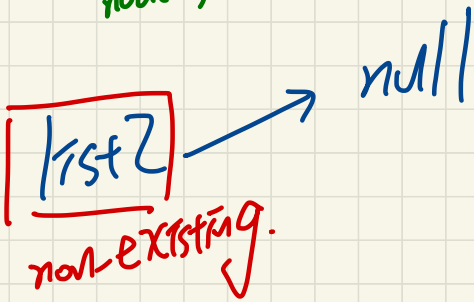
```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

Empty SLL

default const.

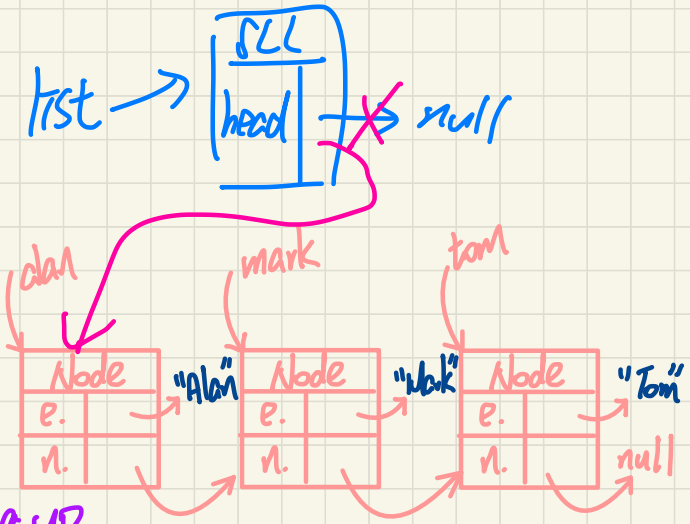
```
SLL list1 = new SLL();
SLL list2 = null;
```





SLL: Setting a List's Head to a Chain of Nodes

```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ... }
   public void removeLast() { ... }
}
```



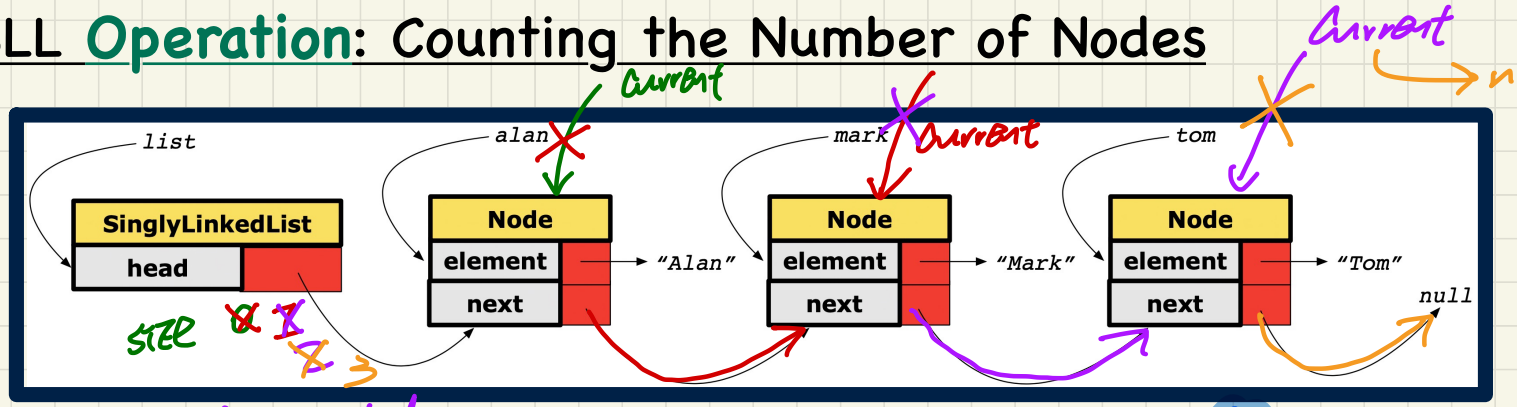
Approach 2

list.setHead(null) → clearing up the list

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

list.setHead(tom)
↳ compiles and runs but potentially logical errors

SLL Operation: Counting the Number of Nodes



a SLL method

```
1  int getSize() {
2     int size = 0;
3     Node current = head;
4     while (current != null) {
5        current = current.getNext();
6        size ++;
7     }
8     return size;
9  }
```

loop counter

exit when current == null

O(n) # iterations = # nodes

Trace: list.getSize()

current	current != null	End of Iteration	size
alan	alan != null	current == mark	1
mark	mark != null	current == tom	2
tom	tom != null	current == null	3
null	null != null		

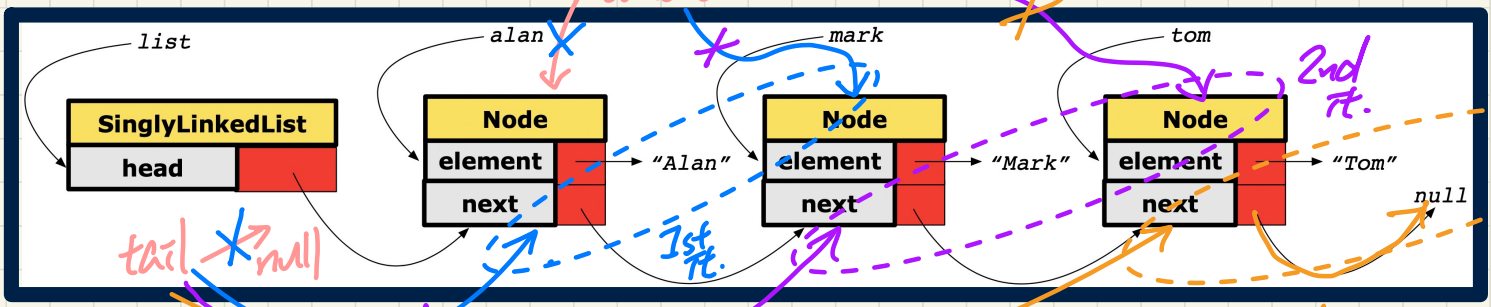
(F)

O(n · 1) = O(n)

ft. ↳ L5, L6

Exercise: Use "current" only to implement getT. current is always one node ahead of tail.

SLL Operation: Finding the Tail of the List




```
1  Node getTail() {
2      Node current = head;
3      Node tail = null;
4      while (current != null) {
5          tail = current;
6          current = current.getNext();
7      }
8      return tail;
9  }
```

Trace: list.getTail()

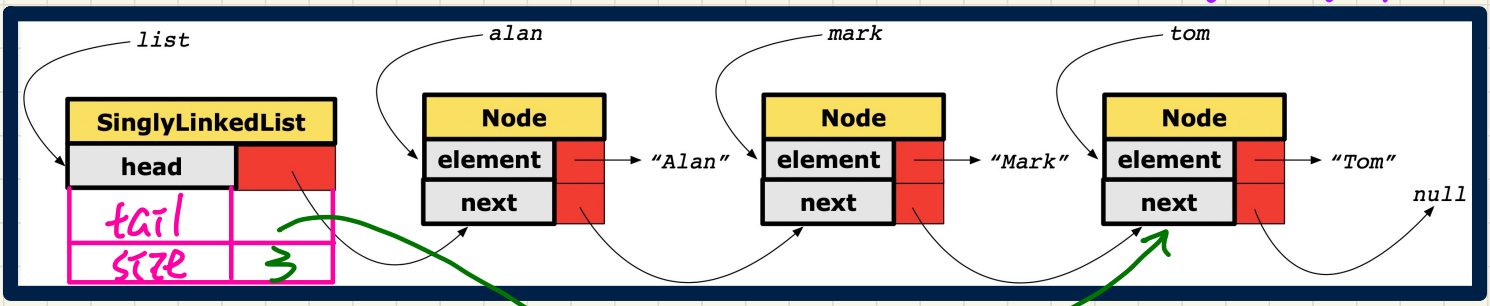
current	current != null	End of Iteration	tail

Iterations = size of list O(n)

O(n · 1) L5, L6 = O(n)

SLL: Trading Space for Time

waste more memory space to make the subsequent computations cheaper



SLL class
- head
- tail
- size

attributes ↓ more cost of space

```
SLL list = new SLL();
    ⋮
```

attribute access ↓ less cost for running time

- list.tail
- list.size

O(1)

↳ turning loops to att. access

Catch for methods that might impact head, tail, or size of the SLL, bodies of imp. must update them properly.